View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

ACM 2007 Turing Award
Edmund Clarke, Allen Emerson, and Joseph Sifakis
Model Checking: Algorithmic Verification and Debugging

ACM Turing Award Citation

In 1981, Edmund M. Clarke and E. Allen Emerson, working in the USA, and Joseph Sifakis working independently in France, authored seminal papers that founded what has become the highly successful field of Model Checking. This verification technology provides an algorithmic means of determining whether an abstract model—representing, for example, a hardware or software design—satisfies a formal specification expressed as a temporal logic formula. Moreover, if the property does not hold, the method identifies a counterexample execution that shows the source of the problem.

The progression of Model Checking to the point where it can be successfully used for complex systems has required the development of sophisticated means of coping with what is known as the state explosion problem. Great strides have been made on this problem over the past 27 years by what is now a very large international research community. As a result many major hardware and software companies are beginning to use Model Checking in practice. Examples of its use include the verification of VLSI circuits, communication protocols, software device drivers, real-time embedded systems, and security algorithms.

The work of Clarke, Emerson, and Sifakis continues to be central to the success of this research area. Their work over the years has led to the creation of new logics for specification, new verification algorithms, and surprising theoretical results. Model Checking tools, created by both academic and industrial teams, have resulted in an entirely novel approach to verification and test case generation. This approach, for example, often enables engineers in the electronics industry to design complex systems with considerable assurance regarding the correctness of their initial designs. Model Checking promises to have an even greater impact on the hardware and software industries in the future.

Allen Emerson: A Bird's Eye View

1. Formal Verification

Formal verification of program correctness hinges on the use of mathematical logic. A program is a mathematical object with well-defined, although possibly complex and intuitively unfathomable, behavior. Mathematical logic can be used to precisely describe what constitutes correct behavior. This makes it possible to contemplate mathematically establishing that the program behavior conforms to the correctness specification. In most early work, this entailed constructing a formal proof of correctness. In contradistinction, Model Checking avoids proofs.

Hoare-style verification was the prevailing mode of formal verification going back from the late-1960s until the 1980s. This classic and elegant approach entailed manual proof construction, using axioms and inference rules in a formal deductive system, oriented toward sequential programs. Such proof construction was tedious, difficult, and required human ingenuity. This field was a great academic success, spawning work on compositional or modular proof systems, soundness of program proof systems, and their completeness; see section 4. Case studies confirmed that this approach worked at least for small programs. A very short program might require a lengthy multi-page paper. However, manual verification did not scale up to large or industrial strength programs. The proofs were just too hard to construct.

2. Temporal Logics

In view of the difficulties in trying to construct program proofs it seemed like there ought to be a better way. The way was inspired by the use of *Temporal Logic* (TL), a formalism for describing change over time. If a program can be specified in TL, it can be realized as a finite state system. This suggested the idea of model checking — to check if a finite state graph is a model of a temporal logic specification.

The critical suggestion of using temporal logic for reasoning about ongoing concurrent programs was made in Pnueli's landmark paper [37]. Such systems ideally exhibit nonterminating behavior so that they do not conform to the Hoare-style paradigm. They are also typically nondeterministic. Examples include hardware circuits, microprocessors, operating systems, banking networks, communication protocols, automotive electronics, and many modern medical devices. He used a temporal logic with basic temporal operators **F** (*sometimes*) and **G** (*always*). Augmented with **X** (*next-time*) and **U** (*until*), this is today known as LTL (Linear Time Logic).

Another widely used logic is CTL (Computation Tree Logic) [8] (cf. [18]) Its basic temporal modalities are **A** (for all futures) or **E** (for some future) followed by one of **F** (sometime), **G** (always), **X** (next-time), and **U** (until); compound formulae are built up from nestings and propositional combinations of CTL subformulae. CTL is a branching time logic as it can distinguish between $\mathbf{AF}P$ (along all futures, P eventually holds and is thus inevitable) and $\mathbf{EF}P$ (along some future, P eventually holds and is thus possible). The branching time logic CTL* subsumes both CTL and LTL.

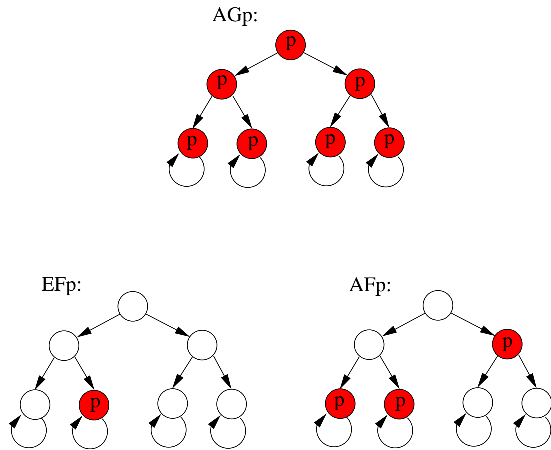

Figure 1: Basic Temporal Operators

Temporal logic formulae are interpreted over a given finite *state graph*, also called a *(Kripke) structure*, M comprised of a set S of *states*, a total binary *transition relation* $R \subseteq S \times S$, and a *labelling* L of states with atomic facts (propositions) true there. We may have also a distinguished (start) state s_0. As usual in mathematical logic, to be precise in defining a logic we use the meta-notation $M, s_0 \models f$ as shorthand for "in structure M at state s_0 formula f is true", for f a CTL (or CTL*) formula. When s_0 is understood, we may write $M \models f$. For example, $M, s_0 \models \mathbf{AF}p$ iff for all paths $x = s_0, s_1, s_2, \ldots$ in M we have $\exists i \geq 0\ P \in L(s_i)$.

When doing specification in practice we may write just $\mathbf{AF}p$ to assert that p is inevitable. An LTL formula h is interpreted over a path and then over a structure by implicit universal path quantification: in practical specification we write h but mean $\mathbf{A}h$.

The LTL formula $\mathbf{G}\neg(C_1 \wedge C_2)$ captures mutual exclusion for the critical sections, corresponding to assertions C_1 and C_2, of processes 1 and 2, respectively. In CTL, we would write $\mathbf{AG}\neg(C_1 \wedge C_2)$ for mutual exclusion, and $\mathbf{AG}(T_1 \Rightarrow \mathbf{AF}\, C_1)$ for "whenever process 1 enters its trying region (T_1) it inevitably enters its (C_1) critical section." The CTL formula $\mathbf{AG}\,\mathbf{EF}\, start$ asserts the system can aways be re-started; this is not expressible in LTL. The CTL* formula $\mathbf{EGF}send$ asserts the existence of a *fair* behavior along which the *send* condition occurs repeatedly. Such fairness conditions are important in ensuring that goals are fulfilled in concurrent systems.

The logics LTL, CTL, and CTL* have turned out to be very influential, spawning industrial extensions and uses plus many academic applications as well as theoretical results. There are prominent industrial logics, tailored for hardware verification using special "macros", i.e. compact high-level operators that expand into longer combinations of basic operators. These include IBM Sugar based on CTL, Intel For-Spec based on LTL, and Accellera-IEEE-1850 PSL incorporating features from CTL*.

Finally, there is also the (propositional) mu-calculus [28] (cf. [18]), a particular but very general temporal logic. It permits temporal correctness properties to be characterized as fixed points or *fixpoints* of recursive definitions. For example $\mathbf{EF}p = p \vee \mathbf{EX}(\mathbf{EF}\ p)$. The mu-calculus plays a vital role in model checking. It is very expressive: CTL, CTL*, as well as LTL, can be encoded in the Mu-calculus. The fixed point characterizations of temporal correctness properties underly many conventional and symbolic model checking algorithms, as well as tools used in practice.

3. Model Checking

In the early 1980s Ed Clarke and I proposed Model Checking, a method for automatic (and algorithmic) verification of finite state concurrent systems [8]; independently J.-P. Quielle and Joseph Sifakis proposed essentially the same method [39]. In Model Checking, temporal logic is used to specify correct system behavior. An efficient, flexible search procedure is used to find correct temporal patterns in the finite state graph of the concurrent system. The orientation of the method is to provide a practical verification method. The technical formulation of the Model Checking problem is simply: Given structure M, state s, and TL formula f, does $M, s \models f$? An alternative formulation is: given M and f, calculate $\{s : M, s \models f\}$. In [8] we made several contributions. We introduced Model Checking (cf. [39]). We formulated the logic CTL. We argued that concurrent programs can be abstracted to finite state synchronization skeletons, suppressing behavior irrelevant to concurrency. We gave a CTL Model Checking algorithm that ran in time $O(|f| \cdot |S|^2)$.

Our algorithm was based on fixpoint characterizations of basic temporal modalities. For example, let $f(Z)$ denote $p \vee \mathbf{AX}Z$. We see that $\mathbf{AF}p = f(\mathbf{AF}p)$ is a fixpoint of $f(Z)$, since AFp holds iff p holds or $AXAFp$ holds. In general, there may be multiple fixpoints. It can be shown that $\mathbf{AF}p$ is the *least fixpoint* which we shall write $\mu Z = f(Z)$, with $f(Z)$ as above. Intuitively, least fixpoints capture only well-founded or finite behaviors. (as in $\mathbf{AF}p$). The fixpoint characterization $\mu Z = f(Z)$ of property makes It is possible to calculate *iteratively* the set of states where $\mathbf{AF}p$ is true. We compute the maximum of the ascending chain of increasingly larger under-approximations to the desired set of states: $false \subseteq f(false) \subseteq f^2(false) \subseteq \ldots \subseteq f^k(false) = f^{k+1}(false)$, where k is at most the size of the (finite) state space. More generally, the *Tarski-Knaster Theorem* [42] permits the ascending iterative calculation $\bigcup f^i(false)$ of any temporal property r characterized as a least fixpoint $\mu Z = f(Z)$, provided that $f(Z)$ is *monotone*, which is ensured by Z only appearing un-negated. For greatest fixpoints, one starts the calculation at $true$. Essentially the same algorithm was given in [39].

We also described a method for efficient Model Checking of basic fairness properties. We showed that Model Checking certain formulae not in CTL was NP-hard. Moreover, we described an algorithmic method to synthesize synchronization skeletons from CTL specifications.

The following are noteworthy extensions. CTL Model Checking can be done in time $O(|M| \cdot |f|)$ [9], i.e., linear in the size of the state graph and linear in the size of the formula. LTL Model Checking can be done in time $O(|M| \cdot exp(|f|)$; since M is usually very large while f is small, the exponential factor may be tolerable [31]. The automata-theoretic approach to LTL Model Checking is described in [44]. A succinct fix-

point characterization of fairness from [21] is used to make LTL Model Checking more efficient in practice. Branching time CTL* Model Checking can be efficiently reduced to linear time LTL Model Checking for the same overall bound [22].

4. Expressivness

An important criterion for a logic is expressiveness, reflecting what correctness properties can and cannot be captured by the logic. Interesting properties include safety properties ("nothing bad happens": e.g., $\mathbf{G}\neg \mathit{bad}$), liveness properties ("something good happens": e.g., $\mathbf{F}\,\mathit{goal}$), and fairness properties ("something is recurrent", e.g., $\mathbf{GF}\,\mathit{try}$). It is arguable that expressiveness in Model Checking is the most fundamental characteristic, perhaps even more critical than efficiency. It is imperative that one be able to actually express all the correctness properties that are needed. If this basic requirement is not met, there is no point in using the verification method in the first place. In actual usage, a particular formalism, commonly a system of temporal logic, provides the needed expressive power. It includes a few basic temporal operators, which can be combined to yield virtually limitless assertions. Another benefit of temporal logic is that it is related to natural language, which can facilitate its use.

The ability to describe complex patterns of system behavior is basic. LTL is naturally suited to the task. Along paths, it is in a sense expressively complete, equivalent to the First Order Language of Linear Order [17], e.g. $\mathbf{G}\,P = \forall t(t \geq 0 \Rightarrow P(t))$. A property such as $\mathbf{G}_2\,P$ meaning that P holds at all even moments 0,2,4,... is not expressible in LTL. It can be useful in hardware verification applications where it is needed to count clock cycles. The (linear time) mu-calculus as well as PSL can express this property. (cf. [45]).

CTL is well suited to capture correctness over computation trees. The branching time capability of distinguishing between necessary and possible behaviors using explicit path quantifiers $\mathbf{A}$, $\mathbf{E}$ provides significant expressive power. The existence of a bad path, $\mathbf{EF}\,\mathit{bad}$, is not expressible by any formula $\mathbf{A}h$ where h is in LTL, nor even any universal CTL* formula where all path quantifiers are $\mathbf{A}$ (and only atomic propositions appear negated). Thus, LTL is not closed under semantic negation: writing the invariant $\mathbf{G}\neg \mathit{bad}$ means $\mathbf{AG}\neg \mathit{bad}$ whose semantic negation is $\mathbf{EF}\,\mathit{bad}$ which, as above, is not expressible by any $\mathbf{A}h$ formula [19]. There has been an ongoing debate as to whether linear time logic or branching time logic is better for program reasoning. Leading proponents of linear time include L. Lamport, A. Pnueli, M. Vardi, and P. Wolper while proponents of branching time include the authors, as well as P. Sistla. Linear time offers the advantage of simplicity, but at the cost of significantly less expressiveness. Branching time's potentially greater expressiveness may incur greater conceptual (and computational) complexity.

A related criterion is succinctness, reflecting how compactly a property can be expressed. The CTL* formula $\mathbf{E}(\mathbf{F}P_1 \wedge \mathbf{F}P_2)$ is not a CTL formula, but is semantically equivalent to the longer CTL formula $\mathbf{EF}(P_1 \wedge \mathbf{EF}P_2) \vee \mathbf{EF}(P_2 \wedge \mathbf{EF}P_1)$. For n conjuncts, the translation is exponential in n. In practice, the most important is the criterion of convenience, reflecting how easily and naturally properties can be expressed. Expressiveness and succinctness may be partially amenable to mathematical definition and investigation. Succinctness and convenience often correlate but not always. Convenience, however, is inherently informal. Yet it is extremely important in actual use. That is why, e.g., many person-years were devoted to formulating industrial strength logics such as PSL.

5. Efficiency

Another important criterion is efficiency, related to questions of the complexity of the Model Checking problem for a logic and the performance of Model Checking algorithms for the logic. An algorithm that has potentially high complexity in theory but is repeatedly observed to exhibit significantly lower complexity in actual use is likely to be preferred to one better theoretical complexity but inferior observed performance. Moreover, there are tradeoffs. For instance, a more expressive logic is likely to be less efficient. A more succinct logic is likely to be more convenient yet even less efficient. Some experience is required to reach a good tradeoff. For many Model Checking applications M is sufficiently small that it can be explicitly represented in computer memory. Such basic enumerative Model Checking may be adequate for systems with 10^6 states.

However, many more systems M have an astronomically or even infinitely large state space. There are some fundamental strategies to cope with large state spaces. Foremost, is the use of abstraction where the original, large, complex system M is simplified, by suppressing inessential detail (cf. [8]), to get a (representation of a) smaller and simpler system $\overline{M}$. Compact representations of the state graph yield another important strategy.

The advent of symbolic Model Checking combining CTL, fixpoint computation, and data structures for compact representation of large state sets, made it possible to check many systems with an astronomical number of states (cf. [6])

If there are many replicated or similar subcomponents, it is often possible to factor out the inherent symmetry in the original M resulting in an expononentially reduced abstract $\overline{M}$ [41] (cf. [7]). Most work on symmetry has required the use of explicit representation of M. Natural attempts to combine symmetry and symbolic representation were shown inherently infeasible [12]. However, a very advantageous combination based on dynamically reorganizing the symbolic representation overcomes these limitations [23]. Finally, one may have an infinite state system comprised of, e.g., a (candidate) dining philosophers solution M_n for all sizes $n > 1$. In many situations, this parameterized correctness problem is reducible to Model Checking a fixed finite size system M_c (cf. [20]).

6. Evolution of Model Checking

The early reception of Model Checking was restrained. Model Checking originated in the theoretical atmosphere of the early 1980s. There was a field of study known as Logics of Programs, which dealt with the theory and sometime use of logic for reasoning about programs. Various modal and temporal logics played a prominent role. The key techni-

cal issue under investigation for such a logic was satisfiability: Given any formula of f, determine whether there exists some structure M such at $M \models f$. Analyzing the decidability and complexity of satisfiability for these logics was the major focus. However, Model Checking refers to the truth under *one* given interpretation M of a given formula f. This notion was implicit in the Tarskian definition of truth but, classically, was not viewed as an interesting problem. The idea that Model Checking should provide for verification of finite state systems was not appreciated. The early reaction to Model Checking then was mostly one of confusion and disinterest. It seemed a disconcerting novelty. It was not satisfiability. It was not validity. What was it? It was even dubbed "disorienting". Many felt it couldn't possibly work well in practice. In more recent times, some more favorable comments have been made. Model Checking is "an acceptable crutch". — Edsger W. Dijkstra; It is "a first step towards engineerization of the field". — A. Pnueli [38].

What factors contributed to Model Checking's successful deployment? First, the initial framework was feasible and comprehensible. It built on a helpful combination of TL and algorithms. It provided a "push-button", i.e., automated, method for verification. It permitted bug detection as well as verification of correctness. Since most programs are wrong, this is enormously important in practice. Incidently, the limited support for bug detection in proof-theoretic verification approaches contributes to their slower adoption rate. Moreover, while a methodology of constructing a program hand-in-hand with its proofs certainly has its merits, it is not readily automatable. This hampers its deployment. With Model Checking, the separation of system development from verification and debugging (see sect. 3) has undoubtedly facilitated Model Checking's industrial acceptance. The development team can go ahead and produce various aspects of system under design. The team of verifiers or verification engineers can conduct verification independently. Hopefully, many subtle bugs will be detected and fixed. As a practical matter, the system can go into production at whatever level of "acceptable correctness" prevails at deadline time. Lastly, Moore's Law has engendered larger computer main memory, which enabled the development of ever more powerful model checking tools.

7. Discussion and Summary

What are the key accomplishments of Model Checking? In my judgement, the key contribution is that verification using model checking is now done *routinely* on a widespread basis for many large systems including industrial-strength systems. Large organizations from hardware vendors to government agencies depend on model checking to facilitate achieving their goals. In contrast to 27 years ago, we no longer just talk about verification; we do it. The somewhat surprising conceptual finding is that verification can be done extremely well by automated search rather than manual proofs.

Model Checking realizes in small part the *Dream of Leibniz [1646-1716]* (cf. [16]). This was a proposal for a universal reasoning system. It was comprised of a *lingua characteristica universalis*, a language in which all knowledge could be formally expressed. Temporal logic plays a limited formulation of this role. There was also a *calculus ratiocinator*, a method of calculating the truth value of such a formalized assertion. Model Checking algorithms provide the means of calculating truth. We hope that, over time, Model Checking will realize an increasingly large portion of Leibniz' Dream.

Edmund Clarke — My 27-Year Quest To Conquer The State Explosion Problem

1. Model Checkers and Debugging

Model Checkers typically have three main components: (1) a *specification language*, based on propositional temporal logic [37], (2) a way of encoding a state machine representing the system to be verified, and (3) a *verification procedure*, that uses an *intelligent* exhaustive search of the state space to determine if the specification is true or not. If the specification is not satisfied, then most Model Checkers will produce a counterexample execution trace that shows why the specification does not hold. It is impossible to overestimate the importance of this feature. The counterexamples are invaluable in debugging complex systems. Some people use Model Checking just for this feature. The EMC Model Checker [9] did not give counterexamples for universal CTL properties that were false or witnesses for existential properties that were true. Michael C. Browne added this feature to the MCB Model Checker in 1984. It has been an important feature of Model Checkers ever since.

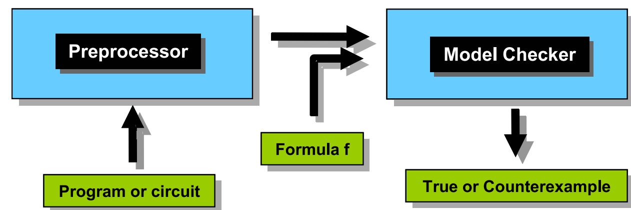

Figure 2: A Model Checker with Counterexamples

2. State Explosion Problem

State explosion is the major problem in Model Checking. The number of global states of a concurrent system with many processes can be enormous. It is easy to see why this is true. The asynchronous composition of n processes, each having m states, may have m^n states. A similar problem occurs with data. The state-transition system for an n-bit counter will have 2^n states. All Model Checkers suffer from this problem. Complexity-theoretic arguments can be used to show that the problem is unavoidable in the worst case. Fortunately, steady progress has been made over the past 27 years for special types of systems that occur frequently in practice. In fact, the state explosion problem has been the driving force behind much of the research in Model Checking and the development of new Model Checkers. We discuss below the major breakthroughs that have been made and some of the important cases where additional research is needed.

3. Major Breakthroughs

3.1 Symbolic Model Checking with OBDDs

In the original implementation of the Model Checking algorithm, transition relations were represented explicitly by adjacency lists [9]. For concurrent systems with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical. In systems with many concurrent parts the number of states in the global state-transition system was too large to handle. In the fall of 1987, McMillan, then a graduate student at Carnegie Mellon, realized that by using a symbolic representation for the state-transition systems, much larger systems could be verified. The new symbolic representation was based on Bryant's *ordered binary decision diagrams* (OBDDs). OBDDs provide a canonical form for Boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states—many orders of magnitude larger than could be handled by the explicit-state algorithms. With the new representation for state-transition systems, we could verify some examples that had more than 10^{20} states [6, 33]. Since then, various refinements of the OBDD-based techniques have pushed the state count up to more than 10^{120}.

3.2 Partial Order Reduction

Verifying software poses significant problems for Model Checking. Software tends to be less structured than hardware. In addition, concurrent software is usually *asynchronous*, i.e., most of the activities taken by different processes are performed independently, without a global synchronizing clock. For these reasons, the state explosion problem is particularly serious for software. Consequently, Model Checking has been used less frequently for software verification than for hardware verification. One of the most successful techniques for dealing with asynchronous systems is the *partial order reduction*. These techniques exploit the independence of concurrently executed events. Intuitively, two events are *independent* of each other when executing them in either order results in the same global state. In this case, it is possible to avoid exploring certain paths in the state-transition system. Model Checking algorithms that incorporate the partial order reduction are described in several different papers. The *stubborn sets* of Valmari [43], the *persistent sets* of Godefroid [25] and the *ample sets* of Peled [36] differ on the actual details, but contain many similar ideas. The SPIN Model Checker developed by Holzmann uses the ample-set reduction to great advantage.

3.3 Bounded Model Checking with SAT

Although Symbolic Model Checking with OBDDs was the first big breakthrough on the state explosion problem and is still widely used, OBDDs have a number of problems that limit the size of the models that can be checked with this technique. The ordering of variables on each path from the root of the OBDD to a leaf has to be the same. Finding an ordering that results in a small OBDD is quite difficult. In fact, for some Boolean formulas no space-efficient ordering is possible. A simple example is the formula for the middle output bit of a combinational multiplier for two n-bit numbers. It is possible to prove that the OBDD for this formula has size that is exponential in n for all variable orderings.

Propositional satisfiability (SAT) is the problem of determining whether a propositional formula in conjunctive normal form ("product of sums form" for Boolean formulas) has a truth assignment that makes the formula true. The problem is NP-complete (in fact, it is usually the first example of this class that students see). Nevertheless, the increase in power of modern SAT solvers over the past 15 years on problems that occur in practice has been phenomenal. It has become the key enabling technology in applications of Model Checking to both computer hardware and software. Bounded Model Checking (BMC) of computer hardware using a fast SAT solver is now probably the most widely used Model Checking technique. The counterexamples that it finds are just the satisfying instances of the propositional formula obtained by unwinding to some fixed depth the state-transition system for the circuit and the negation of its specification in linear temporal logic.

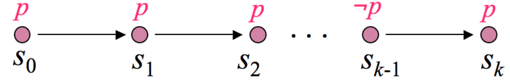

Figure 3: Counterexample of length at most k

The basic idea for BMC is quite simple. The extension to full LTL obscures the simplicity so we will just describe how to check properties of the form $\mathbf{G}\,P$ where the property P is an atomic proposition (e.g., "signal_a = signal_b"). BMC determines whether there is a counterexample of length less than a fixed bound. In other words, it checks if there is a state labeled with $\neg P$ that is reachable by a path consisting of at most k transitions. See Figure 3. Assume that the state-transition system M has n states. Each state can be encoded by a vector $\vec{v}$ of $\lceil log(n) \rceil$ Boolean variables. The set of initial states can be specified by a propositional formula $I(\vec{v})$ which holds for exactly those assignments to $\vec{v}$ that correspond to initial states. Likewise, the transition relation can be given by a propositional formula $R(\vec{v}, \vec{v}\,')$. A path of length k starting in an initial state can be encoded by means of the following formula:

$$I(\vec{v}_0) \wedge R(\vec{v}_0, \vec{v}_1) \wedge \ldots \wedge R(\vec{v}_{k-1}, \vec{v}_k). \quad (1)$$

The property P fails in one of the k steps if and only if

$$\neg P(\vec{v}_0) \vee \neg P(\vec{v}_1) \vee \ldots \vee \neg P(\vec{v}_k). \quad (2)$$

Thus, the safety property $\mathbf{G}\,P$ has a counterexample of length at most k if and only if the conjunction $\Omega(k)$ of Formulas 1 and 2 is satisfiable:

$$\Omega(k) = I(\vec{v}_0) \wedge \bigwedge_{i=0}^{k-1} R(\vec{v}_i, \vec{v}_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(\vec{v}_i). \quad (3)$$

If the formula $\Omega(k)$ is satisfiable, we know that $\mathbf{G}\,P$ has a counterexample of length at most k. A counterexample execution trace can be extracted from the satisfying assignment to $\Omega(k)$. If the formula $\Omega(k)$ is not satisfiable, then it could

be the case that either the temporal formula $\mathbf{G}\,P$ holds on all paths starting from an initial state (and our specification is true) or there is a counterexample that is longer than k. When $\Omega(k)$ is unsatisfiable, we can do one of two things: Either increase the value of k and look for longer counterexamples or stop if time or memory constraints are exceeded.

In practice, BMC can often find counterexamples in circuits with thousands of latches and inputs. Armin Biere recently reported an example in which the circuit had 9510 latches and 9499 inputs. This resulted in a propositional formula with 4×10^6 variables and 1.2×10^7 clauses. The shortest bug of length 37 was found in 69 seconds! Many others have reported similar results.

Can BMC ever be used to prove correctness if no counterexamples are found? It can be argued that for safety properties, specified as a set of bad states, if there is a counterexample, then there is one that is less than the diameter (i.e., the longest shortest path between any two states) of the state-transition system. So, the diameter could be used to place an upper bound on how much the transition relation would need to be unwound. Unfortunately, it appears to be computationally difficult to compute the diameter when the state-transition system is given implicitly as a circuit or in terms of propositional formulas for the set of initial states, the transition relation, and the set of bad states. Other ways for making BMC complete are based on cube enlargement [34], circuit co-factoring [24], induction [40], and Craig interpolants [35]. But, the problem remains a topic of active research. Meanwhile, an efficient way of finding subtle counterexamples is still quite useful in debugging circuit designs.

3.4 The Abstraction Refinement Loop

This technique uses counterexamples to refine an initial abstraction. We begin by defining what it means for one state-transition system to be an abstraction of another. We write $M_\alpha = \langle S_\alpha, s_0^\alpha, R_\alpha, L_\alpha \rangle$ to denote the *abstraction* of state-transition system $M = \langle S, s_0, R, L \rangle$ with respect to an *abstraction mapping* α. (Here we include the start states s_0 and s_0^α as parts of the state-transition systems.) We assume that the states of both M and M_α are labeled with atomic propositions from the set AP. We call M the *concrete* system and M_α the *abstract* system.

DEFINITION 1. *A function $\alpha : S \rightarrow S_\alpha$ is an* abstraction mapping *from the concrete system M to the abstract system M_α with respect to the propositions in A_α if and only if*

- $\alpha(s_0) = s_0^\alpha$
- *If there is a transition from state s to state t in M, then there is a transition from $\alpha(s)$ to $\alpha(t)$ in M_α.*
- *For all states s, $L(s) \cap A_\alpha = L_\alpha(\alpha(s)) \cap A_\alpha$*

The three conditions ensure that M_α *simulates* M. Note that only identically labeled states of the concrete model (modulo propositions absent from A_α) will be mapped into the same state of the abstract model (see Figure 4). The key theorem relating concrete and abstract systems is the *Property Preservation Theorem*:

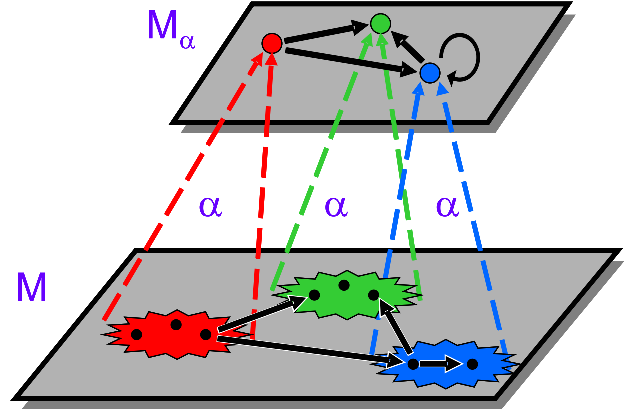

Figure 4: A concrete system and its abstraction

THEOREM 1 (CLARKE, GRUMBERG, AND LONG[11]). *If a universal CTL* property holds on the abstract model, then it holds on the concrete model.*

Here, a universal CTL* property is one that contains no existential path quantifiers when written in negation-normal form. For example, $\mathbf{AF}\,P$ is a universal property but $\mathbf{EF}\,P$ is not.

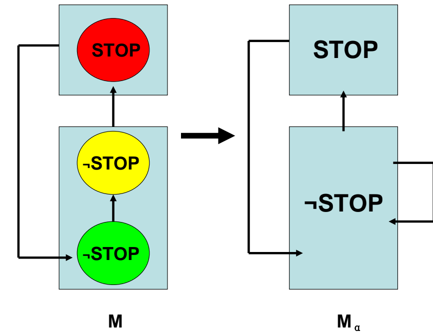

Figure 5: Spurious Counterexample

The converse of the theorem is not true as Figure 5 illustrates. A universal property that holds in the concrete system may fail to hold in the abstract system. For example, the property $\mathbf{A}\,\mathbf{GF}\,STOP$ (*infinitely often* STOP) holds in M, but not in M_α. Thus, a counterexample to the property in the abstract system may fail to be a counterexample in the concrete system. Such counterexamples are said to be *spurious* counterexamples. This leads to a verification technique called *Counterexample Guided Abstraction Refinement* (CEGAR) [10]. Universal properties are checked on a series of increasingly precise abstractions of the original system. If the property holds, then by the Property Preservation Theorem, it must hold on the concrete system and we can stop. If it does not hold and we get a counterexample, then we must check the counterexample on the concrete system in order to make sure that it is not spurious. If the counterexample checks on the concrete system, then we have found an error and can also stop. If the counterexample is spurious, then we use information in the counterexample to refine the abstraction mapping and repeat the loop. The CEGAR Loop in Figure 6 generalizes an earlier abstraction technique

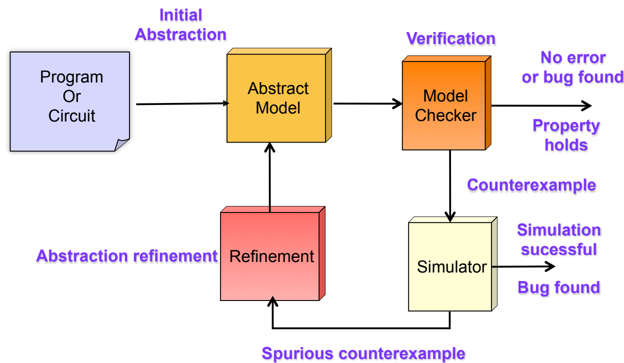

Figure 6: The CEGAR Loop

for sequential circuits called the *localization reduction*, which was developed by R. Kurshan [29]. CEGAR is used in many software Model Checkers including the SLAM Project at Microsoft [1].

4. State Explosion Challenges for the Future

The state explosion problem is likely to remain the major challenge in Model Checking. There are many directions for future research on this problem, some of which are listed below.

- Software Model Checking, in particular, combining Model Checking and Static Analysis
- Effective Model Checking Algorithms for Real-time and Hybrid Systems
- Compositional Model Checking of Complex Systems
- Symmetry Reduction & Parameterized Model Checking
- Probabilistic and Statistical Model Checking
- Combining Model Checking and Theorem Proving
- Interpreting long counterexamples
- Scaling up even more!!

Joseph Sifakis — The Quest for Correctness: Challenges and Perspectives

Where are we today?

Verification techniques have definitely found important applications. After the first two decades of intensive research and development, recent years have been characterized by a shift in focus and intensity.

Algorithmic verification involves three different tasks: (1) requirements specification, (2) building executable system models, and (3) developing scalable algorithms both for checking requirements and for providing diagnostics when requirements are not met. The status for each of these tasks is discussed below.

Requirements specification. Requirements characterize the expected behavior of a system. They can be expressed following two paradigms. *State-based* requirements specify a system's observable behavior by using transition systems. *Property-based* requirements use a declarative style. These requirements are expressed as sets of formulas in a formalism such as a temporal logic. A combination of the two paradigms is necessary for enhanced expressiveness, such as in the PSL language. The state-based paradigm is adequate for characterizing causal dependencies between events, e.g., sequences of actions. In contrast, the property-based paradigm is more appropriate for global properties, e.g., liveness and mutual exclusion. For concurrent systems, an important trend is towards semantic variations of state-based formalisms such as Live Sequence Charts [15].

Using temporal logics has certainly been a breakthrough in understanding and formalizing requirements for concurrent systems. Nonetheless, subtle differences in the formulation of common concepts such as liveness and fairness, which depend on the underlying time model (e.g., branching or linear time), show that writing rigorous logic specifications is not trivial.

Furthermore, the declarative and dense style in the expression of property-based requirements is not always easy to master and understand. Requirements must be *sound*. That is, they are satisfiable by some model. In addition they must be *complete*. That is, no important information is omitted about the specified system. In contrast to soundness which is a well-understood property and can be checked automatically by using decision procedures, there is no consensus as to what precisely constitutes completeness in requirements specifications, nor how to go about achieving it. Absolute completeness, which means that specifications describe the system exactly, has only a theoretical interest and is probably unattainable for non-trivial systems [30].

Existing requirements specification formalisms are mainly appropriate for expressing functional requirements. We lack rigorous formalisms for extra-functional requirements for security properties (e.g., privacy), reconfigurability properties (e.g., non-interference of configurable features), and quality of service (e.g., degree of jitter).

Building executable models. Successful application of verification methods requires techniques for building executable models that *faithfully* represent a system or an abstraction of it. Faithfulness means that the system to be verified and its model are related through a checkable semantics-preserving relation. This will ensure soundness of the model. In other words, any property that we can verify for the model will hold for the real system. Furthermore, to avoid errors in building models and cope with their complexity, models should be generated automatically from system descriptions.

For hardware verification, it is relatively straightforward to generate exact logical finite state models, expressed as systems of boolean equations, e.g., from RTL descriptions. This probably explains the strong and immediate success of Model Checking in the area. For software, the problem is more difficult. In contrast to logical hardware models, we need to formally define the semantics of the programming language. This may not be an easy task for languages such as C or Java, as it requires some clarification of concepts and additional assumptions about their semantics. Once the se-

mantics is fixed, tractable models can be extracted from real software through abstraction. This allows us to cope with complexity of data and dynamic features. Currently, we do not know how to build faithful models for systems consisting of hardware *and* software, at the same level of detail as for pure hardware or software. Ideally, for a system consisting of application software running on a platform, the corresponding model could be obtained as the composition of models for the software and the platform. The main difficulty is in understanding and formalizing the interaction between these two types of models, in particular by taking into account timing aspects and resources such as memory and energy. In addition, this should be done at some adequate level of abstraction, allowing tractable models.

Today, we can specify and verify only high-level timed models with tools such as Uppaal [3] for schedulability analysis. These models take into account hardware timing aspects and some abstraction of the application software. The validation of even relatively simple systems such as a node in a wireless sensor network is carried out by testing physical prototypes or by ad-hoc simulation. We need theory, methods, and tools for modeling complex heterogeneous systems [2]. Weaknesses in the state-of-the-art are also seen in standards and languages for system modeling. Efforts for extending UML to cover scheduling and resource management issues have failed to provide a rigorous basis for this. At the same time, extensions of hardware description languages to encompass more asynchronous execution models such as SystemC and TLM can be used only for simulation, due to a lack of formal semantic foundations.

Scalable verification methods. Today we have fairly efficient verification algorithms. However, all suffer from well-known inherent complexity limitations when applied to large systems. To cope with this complexity, I see two main avenues.

The first avenue is to develop new abstraction techniques, in particular for specific semantic domains depending on the data handled by the system and on the properties to be verified. The convergence between Model Checking and Abstract Interpretation [14] could lead to significant breakthroughs. These two main algorithmic approaches, which have developed rather independently for almost three decades, have a common foundation: solving fixpoint equations in specific semantic domains.

Initially, Model Checking focused on the verification of finite state systems such as hardware or complex control-intensive reactive systems such as communication protocols. Later, research on Model Checking addressed verification of infinite state systems by using abstractions [11, 32]. The evolution of abstract interpretation is driven by the concern for finding adequate abstract domains for efficient verification of program properties by computing approximations of reachability sets. Model Checking has had a broader application scope, including hardware, software and systems. Furthermore, depending on the type of properties to be checked, Model Checking algorithms may involve computation of multiple fixed points. I believe that the combination of the two algorithmic approaches can still lead to significant progress in the state-of-the-art, e.g., by using libraries of abstract domains in Model Checking algorithms.

The second avenue addresses significant long-term progress in defeating complexity. It involves moving from monolithic verification to compositional techniques. We need divide-and-conquer approaches for inferring global properties of a system from the properties of its components. The current state-of-the-art does not meet our initial expectations. The main approach is by "assume-guarantee", where properties are decomposed into two parts. One is an assumption about the global behavior of the system within which the component resides; the other is a property guaranteed by the component when the assumption about its environment holds. As discussed in a recent paper [13], many issues make it difficult to apply assume-guarantee rules, in particular because synthesis of assumptions (when feasible) may cost as much as monolithic verification.

In my opinion, any *general* compositional verification theory will be highly intractable and will be of theoretical interest only. We need to study compositionality results for particular classes of properties and/or particular classes of systems as explained below.

From *a posteriori* verification to constructivity

A big difference between Computer Engineering and more mature disciplines based on Physics, e.g., Electrical Engineering, is the importance of verification for achieving correctness. These disciplines have developed theory guaranteeing by construction the correctness and predictability of artifacts. For instance, the application of Kirchoff's laws allows building circuits that meet given properties.

My vision is to investigate links between compositional verification for specific properties and results allowing constructivity. Currently, there exists in Computer Science an important body of constructivity results about architectures and distributed algorithms.

1) We need theory and methods for building faithful models of complex systems as the composition of heterogeneous components, e.g., mixed software/hardware systems. This is a central problem for ensuring correct interoperation, and meaningful refinement and integration of heterogeneous viewpoints. Heterogeneity has three fundamental sources which appear when composing components with different (a) execution models, e.g., synchronous and asynchronous execution, (b) interaction mechanisms such as locks, monitors, function calls, and message passing, and (c) granularity of execution, e.g., hardware and software [27].

We need to move from composition frameworks based on the use of a single low-level parallel composition operator, e.g., automata-based composition, to a unified composition paradigm encompassing architectural features such as protocols, schedulers, and buses.

2) In contrast to existing approaches, we should investigate compositionality techniques for high-level composition operators and specific classes of properties. I propose to investigate two independent directions:

- One direction is studying techniques for specific classes of properties. For instance, finding compositional verification rules guaranteeing deadlock-freedom or mutual exclusion instead of investigating rules for safety properties in general. Potential deadlocks can be found by analysis of dependencies induced by interactions between components [26]. For proving mutual exclusion, a different type of analysis is needed.
- The other direction is studying techniques for particular architectures. Architectures characterize the way interaction among a system's components is organized. For instance, we might profitably study compositional verification rules for ring or star architectures, for real-time systems with preemptable tasks and fixed priorities, for time-triggered architectures, etc. Compositional verification rules should be applied to high-level coordination mechanisms used at the architecture level, without translating them into a low-level automata-based composition.

The results thus obtained should allow us to identify "verifiability" conditions (i.e., conditions under which verification of a particular property and/or class of systems becomes scalable). This is similar to finding conditions for making systems testable, adaptable, etc. In this manner, compositionality rules can be turned into correct-by-construction techniques.

Recent results implemented in the D-Finder tool [4, 5] provide some illustration of these ideas. D-Finder uses heuristics for proving compositionally global deadlock-freedom of a component-based system, from the deadlock-freedom of its components. The method is compositional and proceeds in two steps.

- First, it checks that individual components are deadlock-free. That is, they may block only at states where they are waiting for synchronization with other components.
- Second, it checks if the components' interaction graph is acyclic. This is a sufficient condition for establishing global deadlock-freedom at low cost. It depends only on the system architecture. Otherwise, D-Finder symbolically computes increasingly strong global invariants of the system, based on results from the first step. Deadlock-freedom is established if there exists some invariant that is satisfied by the system's initial state.

Benchmarks published in [5] show that such a specialization for deadlock-freedom, combined with compositionality techniques, leads to significantly better performance than is possible with general-purpose monolithic verification tools.

A posteriori verification is not the only way to guarantee correctness. System designers develop complex systems, by carefully applying architectural principles that are operationally relevant and technically successful. Verification should advantageously take into account architectures and their features. There is a large space to be explored, between full constructivity and *a posteriori* verification. This vision can contribute to bridging the gap between Formal Methods and the body of constructivity results in Computer Science.

1. References

[1] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Computer Aided Verification (CAV'01)*, LNCS 2102, pages 260–264, 2001.

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006.

[3] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *CAV*, pages 121–125, 2007.

[4] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen. Compositional verification for component-based systems and application. In *ATVA*, pages 64–79, 2008.

[5] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, 2009.

[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. 98(2):142–170, June 1992. Originally presented at the 1990 Symposium on Logic in Computer Science (LICS'90).

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131, 1981.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986. Originally presented at the 1983 Symposium on Principles of Programming Languages (POPL'83).

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic Model Checking. *J. ACM*, 50(5):752–794, 2003. Originally presented at the 2000 Conference on Computer-Aided Verification (CAV'00).

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[12] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic Model Checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[13] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[15] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[16] M. Davis. *The Universal Computer: The Road from*

Leibniz to Turing. W. W. Norton & Co., 2000.

[17] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[18] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Lecture Notes in Computer Science 85*, pages 169–181. Automata, Languages and Programming, July 1980.

[19] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *J. ACM*, 33:151–178, 1986.

[20] E. A. Emerson and V. Kahlon. Reducing Model Checking of the many to the few. In D. A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254. Springer, 2000.

[21] E. A. Emerson and C.-L. Lei. Efficient Model Checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings, Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, Massachusetts, USA*, pages 267–278, 1986.

[22] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[23] E. A. Emerson and T. Wahl. Dynamic symmetry reduction. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2005.

[24] M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based unbounded symbolic Model Checking using circuit cofactoring. In *International conference on Computer-aided design (ICCAD'04)*, pages 510–517, 2004.

[25] P. Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification (CAV'90)*, LNCS 531, 1990.

[26] G. Gößler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[27] T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[28] D. Kozen. Results on the propositional mu-calculus. *Theoretical Comput. Sci.*, 27:333–354, Dec. 1983.

[29] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[30] N. G. Leveson. Completeness in formal specification language design for process-control systems. In *FMSP*, pages 75–87, 2000.

[31] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. pages 97–107, Jan. 1985.

[32] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[33] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[34] K. L. McMillan. Applying SAT methods in unbounded symbolic Model Checking. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 250–264, 2002.

[35] K. L. McMillan. Interpolation and SAT-based Model Checking. In *Computer-Aided Verification (CAV'03)*, LNCS 2725, pages 1–13, 2003.

[36] D. Peled. Combining partial order reductions with on-the-fly Model-Checking. In *Computer Aided Verification (CAV'94)*, LNCS 818, pages 377–390, 1994.

[37] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Comput. Sci.*, 13:45–60, 1981.

[38] A. Pnueli. Verification engineering: A future profession (A. M. Turing Award Lecture). In *PODC*, page 7, 1997.

[39] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1982.

[40] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS 1954, pages 108–125, 2000.

[41] A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.

[42] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

[43] A. Valmari. A stubborn attack on the state explosion problem. In *Computer-Aided Verification (CAV'90)*, LNCS 531, 1990.

[44] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings, Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, Massachusetts, USA*, pages 332–344, 1986.

[45] P. Wolper. Temporal logic can be more expressive. 56:72–99, 1983.