The Beginning of Model Checking: A Personal Perspective*

E. Allen Emerson[1,2]

1. Department of Computer Sciences
2. Computer Engineering Research Center
The University of Texas at Austin,
Austin TX 78712, USA

Abstract. Model checking provides an automated method for verifying concurrent systems. Correctness specifications are given in temporal logic. The method hinges on an efficient and flexible graph-theoretic reachability algorithm. At the time of its introduction in the early 1980's, the prevailing paradigm for verification was a manual one of proof-theoretic reasoning using formal axioms and inference rules oriented towards sequential programs. The need to encompass concurrent programs, the desire to avoid the difficulties with manual deductive proofs, and the small model theorem for temporal logic motivated the development of model checking.

Keywords: model checking, model-theoretic, synthesis, history, origins

1 Introduction

It has long been known that computer software programs, computer hardware designs, and computer systems in general exhibit errors. Working programmers may devote more than half of their time on testing and debugging in order to increase reliability. A great deal of research effort has been and is devoted to developing improved testing methods. Testing successfully identifies many significant errors. Yet, serious errors still afflict many computer systems including systems that are safety critical, mission critical, or economically vital. The US National Institute of Standards and Technology has estimated that programming errors cost the US economy $60B annually [Ni02].

Given the incomplete coverage of testing, alternative approaches have been sought. The most promising approach depends on the fact that programs and more generally computer systems may be viewed as mathematical objects with behavior that is in principle well-determined. This makes it possible to specify using mathematical logic what constitutes the intended (correct) behavior. Then one can try to give a formal proof or otherwise establish that the program meets

* This work was supported in part by National Science Foundation grants CCR-009-8141 & CCR-020-5483 and funding from Fujitsu Labs of America.
email: emerson@cs.utexas.edu URL: www.cs.utexas.edu/∼emerson/

its specification. This line of study has been active for about four decades now. It is often referred to as *formal methods*.

The *verification problem* is: Given program M and specification h determine whether or not the behavior of M meets the specification h. Formulated in terms of Turing Machines, the verification problem was considered by Turing [Tu36]. Given a Turing Machine M and the specification h that it should eventually halt (say on blank input tape), one has the halting problem which is algorithmically unsolvable. In a later paper [Tu49] Turing argued for the need to give a (manual) proof of termination using ordinals, thereby presaging work by Floyd [Fl67] and others.

The model checking problem is an instance of the verification problem. Model checking provides an automated method for verifying concurrent (nominally) finite state systems that uses an efficient and flexible graph search, to determine whether or not the ongoing behavior described by a temporal property holds of the system's state graph. The method is algorithmic and often efficient because the system is finite state, despite reasoning about infinite behavior. If the answer is *yes* then the system meets its specification. If the answer is *no* then the system violates its specification; in practice, the model checker can usually produce a counterexample for debugging purposes.

At this point it should be emphasized that the verification problem and the model checking problem are mathematical problems. The specification is formulated in mathematical logic. The verification problem is distinct from the pleasantness problem [Di89] which concerns having a specification capturing a system that is truly needed and wanted. The pleasantness problem is inherently pre-formal. Nonetheless, it has been found that carefully writing a formal specification (which may be the conjunction of many sub-specifications) is an excellent way to illuminate the murk associated with the pleasantness problem.

At the time of its introduction in the early 1980's, the prevailing paradigm for verification was a manual one of proof-theoretic reasoning using formal axioms and inference rules oriented towards sequential programs. The need to encompass concurrent programs, and the desire to avoid the difficulties with manual deductive proofs, motivated the development of model checking.

In my experience, constructing proofs was sufficiently difficult that it did seem there ought to be an easier alternative. The alternative was suggested by temporal logic. Temporal logic possessed a nice combination of expressiveness and decidability. It could naturally capture a variety of correctness properties, yet was decidable on account of the "Small" Finite Model Theorem which ensured that any satisfiable formula was true in some finite model that was small. It should be stressed that the Small Finite Model Theorem concerns the satisfiability problem of propositional temporal logic, i.e., truth in *some* state graph. This ultimately lead to model checking, i.e., truth in a *given* state graph.

The origin and development of model checking will be described below. Despite being hampered by state explosion, over the past 25 years model checking has had a substantive impact on program verification efforts. Formal verification

has progressed from discussions of how to manually prove programs correct to the routine, algorithmic, model-theoretic verification of many programs.

The remainder of the paper is organized as follows. Historical background is discussed in section 2 largely related to verification in the Floyd-Hoare paradigm; protocol verification is also considered. Section 3 describes temporal logic. A very general type of temporal logic, the mu-calculus, that defines correctness in terms of fixpoint expressions is described in section 4. The origin of model checking is described in section 5 along with some relevant personal influences on me. A discussion of model checking today is given in section 6. Some concluding remarks are made in section 7.

2 Background of Model Checking

At the time of the introduction of model checking in the early 1980's, axiomatic verification was the prevailing verification paradigm. The orientation of this paradigm was manual proofs of correctness for (deterministic) sequential programs, that nominally started with their input and terminated with their output. The work of Floyd [Fl67] established basic principles for proving *partial correctness*, a type of safety property, as well as *termination* and *total correctness*, forms of liveness properties. Hoare [Ho69] proposed an axiomatic basis for verification of partial correctness using axioms and inference rules in a formal deductive system. An important advantage of Hoare's approach is that it was *compositional* so that the proof a program was obtained from the proofs of its constituent subprograms.

The Floyd-Hoare framework was a tremendous success intellectually. It engendered great interest among researchers. Relevant notions from logic such as soundness and (relative) completeness as well as compositionality were investigated. Proof systems were proposed for new programming languages and constructs. Examples of proofs of correctness were given for small programs.

However, this framework turned out to be of limited use in practice. It did not scale up to "industrial strength" programs, despite its merits. Problems start with the approach being one of manual proof construction. These are formal proofs that can involve the manipulations of extremely long logical formulae. This can be inordinately tedious and error-prone work for a human. In practice, it may be wholly infeasible. Even if strict formal reasoning were used throughout, the plethora of technical detail could be overwhelming. By analogy, consider the task of a human adding 100,000 decimal numbers of 1,000 digits each. This is rudimentary in principle, but likely impossible in practice for any human to perform reliably. Similarly, the manual verification of 100,000 or 10,000 or even 1,000 line programs by hand is not feasible. Transcription errors alone would be prohibitive. Furthermore, substantial ingenuity may also be required on the part of the human to devise suitable assertions for loop invariants.

One can attempt to partially automate the process of proof construction using an interactive theorem prover. This can relieve much of the clerical burden.

However, human ingenuity is still required for invariants and various lemmas. Theorem provers may also require an expert operator to be used effectively.

Moreover, the proof-theoretic framework is one-sided. It focuses on providing a way to (syntactically) prove correct programs that are genuinely (semantically) correct. If one falters or fails in the laborious process of constructing a proof of a program, what then? Perhaps the program is really correct but one has not been clever enough to prove it so. On the other hand, if the program is really incorrect, the proof systems do not cater for proving incorrectness. Since in practice programs contain bugs in the overwhelming majority of the cases, the inability to identify errors is a serious drawback of the proof-theoretic approach.

It seemed there ought to be a better way. It would be suggested by temporal logic as discussed below.

Remark. We mention that the term *verification* is sometimes used in a specific sense meaning to establish correctness, while the term *refutation* (or *falsification*) is used meaning to detect an error. More generally, *verification* refers to the two-sided process of determining whether the system is correct or erroneous.

Lastly, we should also mention in this section the important and useful area of protocol validation. Network protocols are commonly finite state. This makes it possible to do simple graph reachability analysis to determine if a *bad* state is accessible (cf. [vB78], [Su78]). What was lacking here was a flexible and expressive means to specify a richer class of properties.

3 Temporal Logic

Modal and temporal logics provided key inspiration for model checking. Originally developed by philosophers, modal logic deals with different *modalities* of truth, distinguishing between P being true in the present circumstances, *possibly* P holding under some circumstances, and *necessarily* P holding under all circumstances. When the circumstances are points in time, we have a modal tense logic or *temporal logic*. Basic temporal modalities include *sometimes* P and *always* P.

Several writers including Prior [Pr67] and Burstall [Bu74] suggested that temporal logic might be useful in reasoning about computer programs. For instance, Prior suggested that it could be used to describe the "workings of a digital computer". But it was the seminal paper of Pnueli [Pn77] that made the critical suggestion of using temporal logic for reasoning about ongoing concurrent programs which are often characterized as *reactive systems*.

Reactive systems typically exhibit ideally nonterminating behavior so that they do not conform to the Floyd-Hoare paradigm. They are also typically nondeterministic so that their non-repeatable behavior was not amenable to testing. Their semantics can be given as infinite sequences of computation states (*paths*) or as computation *trees*. Examples of reactive systems include microprocessors, operating systems, banking networks, communication protocols, on-board avionics systems, automotive electronics, and many modern medical devices.

Pnueli 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). Besides the basic temporal operators applied to propositional arguments, LTL permitted formulae to be built up by forming nestings and boolean combinations of subformulae. For example, $G\neg(C_1 \wedge C_2)$ expresses mutual exclusion of critical sections C_1 and C_2; formula $G(T_1 \Rightarrow (T_1\ U\ C_1)$ specifies that if process 1 is in its trying region it remains there until it eventually enters its critical section.

The advantages of such a logic include a high degree of expressiveness permitting the ready capture of a wide range of correctness properties of concurrent programs, and a great deal of flexibility. Pnueli focussed on a proof-theoretic approach, giving a proof in a deductive system for temporal logic of a small example program. Pnueli does sketch a decision procedure for truth over finite state graphs. However, the complexity would be nonelementary, growing faster than any fixed composition of exponential functions, as it entails a reduction to S1S, the monadic Second order theory of 1 Successor, (or SOLLO; see below). In his second paper [Pn79] on temporal logic the focus is again on the proof-theoretic approach.

I would claim that temporal logic has been a crucial factor in the success of model checking. We have one logical framework with a few basic temporal operators permitting the expression of limitless specifications. The connection with natural language is often significant as well. Temporal logic made it possible, by and large, to express the correctness properties that needed to be expressed. Without that ability, there would be no reason to use model checking. Alternative temporal formalisms in some cases may be used as they can be more expressive or succinct than temporal logic. But historically it was temporal logic that was the driving force.

These alternative temporal formalisms include: (finite state) automata (on infinite strings) which accept infinite inputs by infinitely often entering a designated set of automaton states [Bu62]. An expressively equivalent but less succinct formalism is that of ω-regular expressions; for example, $ab^\star c^\omega$ denotes strings of the form: one a, 0 or more bs, and infinitely many copies of c; and a property not expressible in LTL $(true\ P)^\omega$ ensuring that at every even moment P holds. FOLLO (First Order Language of Linear Order) which allows quantification over individual times, for example, $\forall i \geq 0\ Q(i)$; and SOLLO (Second Order Language of Linear Order) which also allows quantification over sets of times corresponding to monadic predicates such as $\exists Q(Q(0)\wedge\forall i \geq 0(Q(i) \Rightarrow Q(i{+}1))$.[1] These alternatives are sometimes used for reasons of familiarity, expressiveness or succinctness. LTL is expressively equivalent to FOLLO, but FOLLO can be nonelementarily more succinct. This succinctness is generally found to offer no significant practical advantage. Moreover, model checking is intractably (nonelementarily) hard for FOLLO. Similarly, SOLLO is expressively equivalent to $\omega-$regular expressions but nonelementarily more succinct. See [Em90] for further discussion.

[1] Technically, the latter abbreviates $\exists Q(Q(0) \wedge \forall i, j \geq 0(i < j \wedge \neg\exists k(i < k < j)) \Rightarrow (Q(i) \Rightarrow Q(j))$.

Temporal logic comes in two broad styles. A *linear time* LTL assertion h is interpreted with respect to a single path. When interpreted over a program there is an implicit universal quantifications over all paths of the program. An assertion of a *branching time* logic is interpreted over computation trees. The universal A (for all futures) and existential E (for some future) *path quantifiers* are important in this context. We can distinguish between AFP (along all futures, P eventually holds and is thus inevitable)) and EFP (along some future, P eventually holds and is thus possible).

One widely used branching time logic is CTL (Computation Tree Logic) (cf. [CE81]). 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 derives from [EC80]. There we defined the precursor branching time logic CTF which has path quantifiers $\forall fullpath$ and $\exists fullpath$, and is very similar to CTL. In CTF we could write $\forall fullpath \exists stateP$ as well as $\exists fullpath \exists stateP$ These would be rendered in CTL as AFP and EFP, respectively. The streamlined notation was derived from [BMP81]. We also defined a modal mu-calculus FPF, and then showed how to translate CTF into FPF. The heart of the translation was characterizing the temporal modalities such as AFP and EFP as fixpoints. Once we had the fixpoint characterizations of these temporal operators, we were close to having model checking.

CTL and LTL are of incomparable expressive power. CTL can assert the existence of behaviors, e.g., $AGEFstart$ asserts that it is always possible to re-initialize a circuit. LTL can assert certain more complex behaviors along a computation, such as $GFen \Rightarrow Fex$ relating to fairness. (It turns out this formula is not expressible in CTL, but it is in "FairCTL" [EL87]) The branching time logic CTL* [EH86] provides a uniform framework that subsumes both LTL and CTL, but at the higher cost of deciding satisfiability. There has been an ongoing debate as to whether linear time logic or branching time logic is better for program reasoning (cf. [La80], [EH86], [Va01]).

Remark. The formal semantics of temporal logic formulae are defined with respect to a *(Kripke) structure* $M = (S, S_0, R, L)$ where S is a set of states, S_0 comprises the initial states, $R \subseteq S \times S$ is a total binary relation, and L is a labelling of states with atomic facts (propositions) true there. An LTL formula h such as FP is defined over path $x = t_0, t_1, t_2 \ldots$ through M by the rule $M, x \models FP$ iff $\exists i \geq 0 \ P \in L(t_i)$. Similarly a CTL formula f such as EGP holds of a state t_0, denoted $M, t_0 \models EGP$, iff there exists a path $x = t_0, t_1, t_2, \ldots$ in M such that $\forall i \geq 0 \ P \in L(t_i)$. For LTL h, we define $M \models h$ iff for all paths x starting in S_0, $M, x \models h$. For CTL formula f we define $M \models f$ iff for each $s \in S_0$, $M, s \models f$. A structure is also known as a *state graph* or *state transition graph* or *transition system.* See [Em90] for details.

4 The Mu-calculus

The mu-calculus may be viewed as a particular but very general temporal logic. Some formulations go back to the work of de Bakker and Scott [deBS69]; we deal specifically with the (propositional or) modal mu-calculus (cf. [EC80], [Ko83]). The mu-calculus provides operators for defining correctness properties using recursive definitions plus least fixpoint and greatest fixpoint operators. Least fixpoints correspond to well-founded or terminating recursion, and are used to capture liveness or progress properties asserting that something does happen. Greatest fixpoints permit infinite recursion. They can be used to capture safety or invariance properties. The mu-calculus is very expressive and very flexible. It has been referred to as a "Theory of Everything".

The formulae of the mu-calculus are built up from atomic proposition constants $P, Q, \ldots$, atomic proposition variables $Y, Z, \ldots$, propositional connectives $\vee, \wedge, \neg$, and the least fixpoint operator, μ as well as the greatest fixpoint operator, ν. Each fixpoint formula such as $\mu Z.\tau(Z)$ should be syntactically monotone meaning Z occurs under an even number of negations, and similarly for ν.

The mu-calculus is interpreted with respect to a structure $M = (S, R, L)$. The power set of S, 2^S, may be viewed as the complete lattice $(2^S, S, \emptyset, \subseteq, \cup, \cap)$. Intuitively, each (closed) formula may be identified with the set of states of S where it is true. Thus, *false* which corresponds to the empty set is the bottom element, *true* which corresponds to S is the top element, and implication $(\forall s \in S[P(s) \Rightarrow Q(s)])$ which corresponds to simple set-theoretic containment $(P \subseteq Q)$ provides the partial ordering on the lattice. An open formula $\tau(Z)$ defines a mapping from $2^S \rightarrow 2^S$ whose value varies as Z varies. A given $\tau : 2^S \rightarrow 2^S$ is *monotone* provided that $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$.

Tarski-Knaster Theorem. (cf. [Ta55], [Kn28])
Let $\tau : 2^S \rightarrow 2^S$ be a monotone functional. Then

(a) $\mu Y.\tau(Y) = \cap \{Y : \tau(Y) = Y\} = \cap \{Y : \tau(Y) \subseteq Y\}$,
(b) $\nu Y.\tau(Y) = \cup \{Y : \tau(Y) = Y\} = \cup \{Y : \tau(Y) \supseteq Y\}$,
(c) $\mu Y.\tau(Y) = \cup_i \tau^i(\mathit{false})$ where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$, and
(d) $\nu Y.\tau(Y) = \cap_i \tau^i(\mathit{true})$ where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$.

Consider the CTL property AFP. Note that it is a fixed point or fixpoint of the functional $\tau(Z) = P \vee AXZ$. That is, as the value of the input Z varies, the value of the output $\tau(Z)$ varies, and we have $AFP = \tau(AFP) = P \vee AXAFP$. It can be shown that AFP is the least fixpoint of $\tau(Z)$, meaning the set of states associated with AFP is a subset of the set of states associated with Z, for any fixpoint $Z = \tau(Z)$. This might be denoted $\mu Z.Z = \tau(Z)$. More succinctly, we normally write just $\mu Z.\tau(Z)$. In this case we have $AFP = \mu Z.P \vee AXZ$.

We can get some intuition for the the mu-calculus by noting the following fixpoint characterizations for CTL properties:

$EFP = \mu Z.P \vee EXZ$
$AGP = \nu Z.P \wedge AXZ$
$AFP = \mu Z.P \vee AXZ$
$EGP = \nu Z.P \wedge EXZ$
$A(P\ U\ Q) = \mu Z.Q \vee (P \wedge AXZ)$
$E(P\ U\ Q) = \mu Z.Q \vee (P \wedge EXZ)$

For all these properties, as we see, the fixpoint characterizations are simple and plausible. It is not too difficult to give rigorous proofs of their correctness (cf. [EC80], [EL86]). We emphasize that the mu-calculus is a rich and powerful formalism; its formulae are really representations of alternating finite state automata on infinite trees [EJ91]. Since even such basic automata as deterministic finite state automata on finite strings can form quite complex "cans of worms", we should not be so surprised that it is possible to write down highly inscrutable mu-calculus formulae for which there is no readily apparent intuition regarding their intended meaning. The mu-calculus has also been referred to as the "assembly language of program logics" reflecting both its comprehensiveness and potentially intricate syntax. On the other hand, many mu-calculus characterizations of correctness properties are elegant due to its simple underlying mathematical organization.

In [EL86] we introduced the idea of model checking for the mu-calculus instead of testing satisfiability. We catered for efficient model checking in fragments of the the mu-calculus. This provides a basis for practical (symbolic) model checking algorithms. We gave an algorithm essentially of complexity n^d, where d is the alternation depth reflecting the number of significantly nested least and greatest fixpoint operators. We showed that common logics such as CTL, LTL, and CTL* were of low alternation depth $d = 1$ or $d = 2$. We also provided succinct fixpoint characterizations for various natural fair scheduling criteria. A symbolic fair cycle detection method, known as the "Emerson-Lei" algorithm, is comprised of a simple fixpoint characterization plus the Tarski-Knaster theorem. It is widely used in practice even though it has worst case quadratic cost. Empirically, it usually outperforms alternatives.

5 The Origin of Model Checking

There were several influences in my personal background that facilitated the development of model checking. In 1975 Zohar Manna gave a talk at the University of Texas on fixpoints and the Tarski-Knaster Theorem. I was familiar with Dijkstra's book [Di76] extending the Floyd-Hoare framework with *wlp* the *weakest liberal precondition* for partial correctness and *wp* the *weakest precondition* for total correctness. It turns out that *wlp* and *wp* may be viewed as modal operators, for which Dijkstra implicitly gave fixpoint characterizations, although Dijkstra did not favor this viewpoint. Basu and Yeh [BY75] at Texas gave fixpoint characterizations of weakest preconditions for while loops. Ed Clarke [Cl79] gave similar fixpoint characterizations for both *wp* and *wlp* for a variety of control structures.

I will now describe how model checking originated at Harvard University. In prior work [EC80] we gave fixpoint characterizations for the main modalities of a logic that was essentially CTL. These would ultimately provide the first key ingredient of model checking.

Incidentally, [EC80] is a paper that could very well not have appeared. Somehow the courier service delivering the hard-copies of the submission to Amsterdam for the program chair at CWI (Dutch for "Center for Mathematics and Computer Science") sent the package in bill-the-recipient mode. Fortunately, CWI was gracious and accepted the package. All that remained to undo this small misfortune was to get an overseas bank draft to reimburse them.

The next work, entitled "Design and Synthesis of Synchronization Skeletons using Branching Time Logic", was devoted to program synthesis and model checking. I suggested to Ed Clarke that we present the paper, which would be known as [CE81], at the IBM Logics of Programs workshop, since he had an invitation to participate.

Both the idea and the term *model checking* were introduced by Clarke and Emerson in [CE81]. Intuitively, this is a method to establish that a given program meets a given specification where:

- The program defines a finite state graph M.
- M is searched for elaborate *patterns* to determine if the specification f holds.
- Pattern specification is *flexible*.
- The method is *efficient* in the sizes of M and, ideally, f.
- The method is *algorithmic*.
- The method is *practical*.

The conception of model checking was inspired by program synthesis. I was interested in verification, but struck by the difficulties associated with manual proof-theoretic verification as noted above. It seemed that it might be possible to avoid verification altogether and mechanically synthesize a correct program directly from its CTL specification. The idea was to exploit the small model property possessed by certain decidable temporal logics: any satisfiable formula must have a "small" finite model of size that is a function of the formula size. The synthesis method would be *sound*: if the input specification was satisfiable, it built a finite global state graph that was a model of the specification, from which individual processes could be extracted The synthesis method should also be *complete*: If the specification was unsatisfiable, it should say so.

Initially, it seemed to me technically problematic to develop a sound and complete synthesis method for CTL. However, it could always be ensured that an alleged synthesis method was at least sound. This was clear because *given any finite* ***model*** *M and CTL specification f one can algorithmically* ***check*** *that M is a genuine model of f by evaluating (verifying) the basic temporal modalities over M based on the Tarski-Knaster theorem.* This was the second key ingredient of model checking. Composite temporal formulae comprised of nested subformulae and boolean combinations of subformulae could be verified by recursive descent. Thus, fixpoint characterizations, the Tarski-Knaster theorem, and recursion yielded *model checking*.

Thus, we obtained the model checking framework. A model checker could be quite useful in practice, given the prevalence of finite state concurrent systems. The temporal logic CTL had the flexibility and expressiveness to capture many important correctness properties. In addition the CTL model checking algorithm was of reasonable efficiency, polynomial in the structure and specification sizes. Incidentally, in later years we sometimes referred to *temporal logic model checking.*

The crucial roles of abstraction, synchronization skeletons, and finite state spaces were discussed in [CE81]:

> The synchronization skeleton is an abstraction where detail irrelevant to synchronization is suppressed. *Most solutions to synchronization problems are in fact given as synchronization skeletons.*
>
> Because synchronization skeletons are in general finite state ... *propositional temporal logic can be used to specify their properties.*
>
> The finite model property ensures that any program whose synchronization properties can be expressed in propositional temporal logic *can be realized by a finite state machine.*

Conclusions of [CE81] included the following prognostications, which seem to have been on target:

> [Program Synthesis] may in the long run be quite practical. Much additional research will be needed, however, to make it feasible in practice. ... We believe that practical [model checking] tools could be developed in the near future.

To sum up, [CE81] made several contributions. It introduced model checking, giving an algorithm of quadratic complexity $O(|f||M|^2)$. It introduced the logic CTL. It gave an algorithmic method for concurrent program synthesis (that was both sound and complete). It argued that most concurrent systems can be abstracted to finite state synchronization skeletons. It described a method for efficiently model checking basic fairness using strongly connected components. An NP-hardness result was established for checking certain assertions in a richer logic than CTL. A prototype (and non-robust) model checking tool BMoC was developed, primarily by a Harvard undergraduate, to permit verification of synchronization protocols.

A later paper [CES86] improved the complexity of CTL model checking to linear $O(|f||M|)$. It showed how to efficiently model check relative to unconditional and weak fairness. The EMC model checking tool was described, and a version of the alternating bit protocol verified. A general framework for efficiently model checking fairness properties was given in [EL87], along with a reduction showing that CTL* model checking could be done as efficiently as LTL model checking.

Independently, a similar method was developed in France by Sifakis and his student [QS82]. Programs were interpreted over transition systems. A branching

time logic with operators POT (EF) and $INEV$ (AF) and their duals was used; omitted were the X (*next-time*) and U (*until*) operators available in CTL. Interestingly, there was no emphasis on the role of finiteness, no complexity analysis, and no proof of termination. However, the central role of fixpoint computation was identified. (The follow-on paper [FSS83] does emphasize the importance of finiteness.) A tool CESAR is described and the verification of an alternating bit protocol discussed.

Remark. The study of program synthesis together with analysis of programs (or verification) has a long history. In 1956 Kleene [Kl56] proposes (i) the *synthesis problem*: from a given regular expression h, construct an equivalent finite state automaton M; and (ii) the *analysis problem*: given a finite automaton M construct an equivalent regular expression h, i.e., in other words the strongest specification h such that M verifies h. Strictly speaking, Kleene dealt with machinery suitable for reasoning about ongoing but finite behavior; however, the results generalize to infinite behavior.

6 Model Checking Today

The fundamental accomplishment of model checking is enabling broad scale formal verification. Twenty five years ago our community mostly just talked about verification. Today we do verification: many industrial-strength systems have been verified using model checking. More are being verified on a routine basis. Formal verification is becoming a staple of CS and EE education. At the same time there is ever growing research interest in model checking.

How and why did this come about? I would argue that it is due to the following. In [CE81] a feasible framework including a usefully expressive branching time temporal logic (CTL) and a reasonably efficient model checking algorithm was introduced. Its utility was clear for small examples. Plainly one could model check many interesting programs because they could be represented at a meaningful level of abstraction by a finite state system with dozens to hundreds or thousands of states. Protocols provide many examples of this. Yet there was doubtless a need to be able to handle larger programs. This garnered the attention of a sizable and still growing number of researchers in both academia and industry.

The most serious (and obvious) drawback of model checking is the *state explosion problem*. The size of the global state graph can be (at least) exponential in the size of the program text. A concurrent program with k processes can have a state graph of size $exp(k)$. For instance in a banking network with 100 automatic teller machines each controlled by a finite state machine with 10 states, we can have 10^{100} global states. Systems with infinite state spaces, in general, cannot be handled.

To reduce the state explosion problem, methods based on abstraction, symbolic representation, and compositional reasoning are used. These are discussed in more detail subsequently.

Today, model checkers are able to verify protocols with millions of states and many hardware circuits with 10^{50} or more states. Even some systems with an infinite number of states can be amenable to model checking, if we have a suitable finite representation of infinite sets of states in terms of symbolic constraints.

Model checking has made verification commonplace in many industrial settings where applications are safety critical or economically vital. These include hardware and CAD companies such as IBM, Intel, Cadence, and Synopsys, software companies such as Microsoft, and government agencies such as NASA.

Today there are many logics and methods for model checking. CTL and the mu-calculus with associated fixpoint-based model checking algorithms are still in widespread use. There is also linear temporal logic LTL model checking. It was considered in a tableaux-theoretic approach by Lichtenstein and Pnueli [LP85]. More generally, LTL model checking can be done through reduction to automaton nonemptiness as shown by Vardi and Wolper [VW86], and independently by Kurshan (cf. [Ku94]). The automata-theoretic approach readily generalizes to a broader class of linear formalisms than just LTL. Interestingly, it is often implemented on top of a mu-calculus or (fair) CTL model checking algorithm (cf. [EL86], [EL87]), where linear temporal model checking over one structure is transformed to checking fairness over another structure.

A crucial factor is the formulation and application of abstractions. Given original system M an abstraction is obtained by suppressing detail yielding a simpler and likely smaller system $\overline{M}$ that is, ideally, equivalent to M for purposes of verification. The precise nature of the abstraction and the correspondence between M and $\overline{M}$ can vary considerably.

An *exact* abstraction guarantees that M is correct if and only if $\overline{M}$ is correct. A *bisimulation* is a technical concept [Pa81] associating M and $\overline{M}$ that guarantees an exact abstraction, and such that the original and the abstraction cannot be distinguished by any "reasonable" temporal logic.

Systems M comprised of multiple, interchangeable subcomponents typically exhibit symmetry which may be thought of as a form of redundancy. This can be abstracted out by identifying symmetric states to get abstraction $\overline{M}$ that is bisimilar to M. The symmetry abstraction can be exponentially smaller than the original, yielding a dramatic speedup in model checking. A resource controller with 150 homogeneous processes and a global state graph M of size about 10^{47} states[2] can be model checked over the abstract $\overline{M}$ in a few tens of minutes [ES97].

A related problem is *parameterized* model checking. Given a family of n similar processes, establish correctness for systems of all sizes n. (Note that collectively this amounts to an infinite state program.) While in general an undecidable problem, various restricted cases can be solved. In [EN96] we developed a mathematical theory for a restricted but still useful framework. In [EN98] we showed how to use this theory to verify parameterized correctness of the Soci-

[2] It should be emphasized that the original state graph of size 10^{47} is not and cannot be constructed by the model checker. A smaller abstract graph representing the essential information is built instead.

ety of Automotive Engineers SAE-J1850 automotive bus protocol. This solved a problem relating to the use of multiple embedded Motorola micro-controllers in Ford automobiles.

A *conservative* abstraction ensures that correctness of $\overline{M}$ implies correctness of M. An abstraction obtained from M by partitioning and clustering states in the natural way will be conservative. A *simulation* from M to $\overline{M}$ yields a conservative abstraction, preserving correctness in $\overline{M}$ to M. On the other hand, if $\overline{M}$ is incorrect the error may be bogus and the abstraction too coarse. Repeatedly refining it as needed and as resources permit typically leads to determination of correctness vs. incorrectness for M (cf. [Ku94]).

For hardware verification a basic aid is symbolic representation of the program's states and state transitions using data structures called Reduced Ordered Binary Decision Diagrams (ROBDDs) [Br86] often called BDDs for short (cf. [Le59], [Ak78]). A BDD is essentially an acyclic deterministic finite state automaton. Given a set P of system states, each a string of bits $b_1 b_2 \ldots b_n$, the BDD for P accepts exactly those states in P. Note that a BDD with a polynomial number of nodes may have an exponential number of paths. In this way, a BDD may represent a vastly larger set of states.

Symbolic model checking [B+90] is based on the original CTL logic and fixpoint based model checking algorithm [CE81] plus BDDs to represent sets of states and transitions. It is routinely able to verify hardware designs modeled with 100 to 300 or more state variables and having about 10^{30} to 10^{90} or more global states. This corresponds to a large enough chunk of real estate on a chip to be extremely valuable. Larger components are often amenable to verification through decomposition and compositional reasoning.

BDDs tend to blow up in size for large designs. Conventional BDDs have topped out for systems with a few hundred state variables. SAT-based bounded model checking is an alternative approach [B+99]. The SAT approach can accommodate larger designs than the BDD approach. However it only explores for "close" errors at depth bounded by k where typically k ranges from a few tens to hundreds of steps. In general it cannot find "deep" errors and provide verification of correctness.

Remark. It should be emphasized that not all systems with, say, 10^{90} states can be handled, since there are relatively few succinct representations and they are insufficient to cover all such astronomically large systems. The pertinent fact is that the method works routinely on the large systems encountered in practice. On the other hand, there are some relatively small hardware systems for which BDDs are too big, while a conventional explicit state representation is workable.

In software model checking, Microsoft has routinely verified device drivers with 100,000 lines of code. The task is made easier by the fact that drivers are more or less sequential code. Therefore state explosion is less of an issue. However, software is usually more difficult to verify than hardware. It typically has less of a regular organization. It may involve significant use of recursion, and complex, dynamic data structures on the heap. It can also be extremely large.

A remaining significant factor in ameliorating state explosion is the exponential growth in computer power, speed and especially memory size, expressed in Moore's law which has obtained over the past quarter century. For example, in 1981 the IBM PC had less than 1M (random access) memory while today many PC's have 1G or more memory. Such a 1000-fold or larger increase in memory permits significantly larger programs to be handled.

There are numerous model checking tools. They typically include a modeling language for representing the program corresponding to the structure M, a specification logic such as CTL or LTL for capturing correctness properties f, a model checking algorithm that is often fixpoint based, and perhaps special data structures such as BDDs for symbolic model checking or for incrementally building the state graph for explicit state model checking. Some of these are academic tools, others are industrial internal tools, and some are for sale by CAD vendors.

7 Conclusions and Future Trends

The fundamental accomplishment of model checking is enabling broad scale formal verification. Twenty five years ago our community mostly just talked about verification. Today we do verification: many industrial-strength systems have been verified using model checking. More are being verified on a routine basis. Model checking thus has produced an era of feasible, automatic, model-theoretic verification. It should be emphasized that a model checker decides correctness: yes or no. Thus it caters for both verification and refutation of correctness properties. Since most programs do contain errors, an important strength of model checkers is that they can readily provide a counter-example for most errors.

Model checking realizes in small part the Dream of Leibniz (1646 – 1716) to permit calculation of the truth status of formalized assertions. The Dream of Leibniz was comprised of two parts: *lingua characteristica universalis*, a language in which all knowledge could be formally expressed; and *calculus ratiocinator*, a method of calculating the truth value of such an assertion. Leibniz's original Dream was unworkable because its scope was too vast and the level of available precision too low.

Model checking is feasible because its domain is both well-defined and much more narrow. Temporal logic is also precisely defined while limited in expressive power especially in comparison to formalisms such as First Order Arithmetic plus Monadic Predicates; yet, temporal logic and related formalisms seem ideally suited to describing synchronization and coordination behavior of concurrent systems; the restriction to finite state systems means that model checking procedures are in principle algorithmic and in practice efficient for many systems of considerable size. It is in just these coordination aspects that errors are most prone to occur for concurrent programs.

Beyond the Dream of Leibniz, model checking also validates in small part the seemingly naive expectation during the early days of computing that large problems could be solved by brute force computation including the technique of exhaustive search. Model checking algorithms are typically efficient in the

size of the state graph. The difficulty is that the state graph can be immense. Abstraction can be helpful because it in effect replaces the original state graph by a much smaller one verifying the same formulae.

Model checking has been an enabling technology facilitating cross disciplinary work. Problems from diverse areas distinct from formal methods can with some frequency be handled with existing model checking tools and possibly without a deep understanding of model checking per se.

It is worth mentioning some of the applications of model checking elsewhere. These include understanding and analyzing legal contracts, which are after all prescriptions for behavior [Da00]; analyzing processes in living organisms for systems biology [H+06]; e-business processes such as accounting and workflow systems [Wa+00]. Model checking has also been employed for tasks in artificial intelligence such as planning [GT99]. Conversely, techniques from artificial intelligence related to SAT-based planning [KS92] are relevant to (bounded) model checking.

In the present formal methods research milieu, the ideal paper contributes both new theory and practical experimental evidence of the merits of the theory. A significant benefit of this tight coupling is that it promotes the development of concretely applicable theory. On the other hand, such a synchronous organization may risk slowing down the rate of advance on the theoretical track as well as on the practice track. It will be interesting to see if as the field develops in the future it might adopt something of the more specialized organization of older disciplines. For instance, one has the broad divisions of theoretical physics and experimental physics, which are more loosely coupled.

Model checking would benefit from future theoretical advances. This is especially important in view of the fact that many model checking methods are in principle algorithmic but of high theoretical, worst case complexity. Their good performance in practice has a heuristic character and is not well-understood on a mathematical basis. Many efficiency enhancement techniques produce an unpredictable gain in efficiency. To gain a better theoretical understanding of when good efficiency obtains would be a very desirable goal for the future.

It is thus especially important to obtain convincing empirical documentation of a verification method's effectiveness and efficiency. One way to do this might be to establish a broad set of benchmarks. A major obstacle, however, is that proprietary hardware designs and software code are virtually never available in an open fashion because they are patented, or trade secrets, etc. This means that, to the extent that success at verifying industrial systems is the yardstick of merit, we have lost the critical standard of repeatability normally associated with experimental sciences.

Various interesting remarks have been made concerning model checking. Edsger W. Dijkstra commented to me that it was an "acceptable crutch" if one was going to do after-the-fact verification. When I had the pleasure of meeting Saul Kripke and explaining model checking over Kripke structures to him, he commented that he never thought of that. Daniel Jackson has remarked that model checking has "saved the reputation" of formal methods [Ja97].

In summary, model checking today provides automatic verification that is applicable to a broad range of sizable systems including many that are industrial strength. At the same time the verification problem is not solved. We still have quite a way to go.

Grand Challenge for Hardware. Hardware designs with a few hundreds to thousands of state variables can be model checked in some fashion; but not an entire microprocessor. It would be a Grand Challenge to verify an entire microprocessor with one hundred thousand state variables.

Grand Challenge for Software. Software device drivers have been shown amenable to software model checking. These are mostly sequential software with up to one hundred thousand lines of code. Of course, there is software with millions of lines of code. Windows Vista contains somewhat over 50 million lines of code, and entails concurrency as well. It would be a Grand Challenge to to verify software with millions to tens of millions lines of code.

In the future, we should see progress on the key topics of limiting state explosion, improved abstractions, better symbolic representations, broader parameterized reasoning techniques, and the development of temporal formalisms specialized to particular application domains (cf. [IEEE05]). It seems likely that parallel and distributed model checking will grow in importance (cf. [HvdP06]). In any case, I expect that model checking will become yet more useful.

References

[Ak78] Sheldon B. Akers, "Binary Decision Diagrams", IEEE Trans. on Computers, C-27(6):509-516, June 1978.

[AENT01] Nina Amla, E. Allen Emerson, Kedar S. Namjoshi, Richard J. Trefler: "Assume-Guarantee Based Compositional Reasoning for Synchronous Timing Diagrams", *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001: 465-479.

[B+90] J. Birch, E. Clarke, K. MacMillan, D. Dill, L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond", *Logic in Computer Science*, LICS 1990, 428-439.

[B+99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs", *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 1999, 193-207.

[BMP81] Mordechai Ben-Ari, Zohar Manna, Amir Pnueli: "The Temporal Logic of Branching Time", *Principles of Programming Languages*, POPL 1981: 164-176.

[Br86] Randall Bryant: "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Computers*, 35(8): 677-691 (1986).

[BY75] Basu, S. K., and R. T. Yeh, "Strong Verification of Programs", *IEEE Trans. on Software Engineering*, vol. SE-1, no. 3, 339-345, (1975).

[Bu62] J. R. Buchi, "On a Decision Method in Restricted Second Order Arithmetic", *Proc. of Int'l. Congress on Logic Method, and Philosophy of Science*, 1960, Stanford Univ. Press, pp. 1-12, 1962.

[Bu74] Rod M. Burstall: "Program Proving as Hand Simulation with a Little Induction", *IFIP Congress*, 1974: 308-312.

[CE81] E. M. Clarke and E. A. Emerson, "The Design and Synthesis of Synchronization Skeletons Using Temporal Logic", *Proceedings of the Workshop on Logics of Programs*, IBM Watson Research Center, Yorktown Heights, New York, Springer-Verlag Lecture Notes in Computer Science, #131, pp. 52–71, May 1981.

[CES86] 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. and Sys.*, (8)2, pp. 244-263, 1986.

[Cl79] E. M. Clarke, "Program Invariants as Fixpoints", *Computing*, 21(4), Dec. 1979, 273-294.

[Da00] A Daskalopulu, "Model Checking Contractual Protocols", in *Legal Knowledge and Information Systems*, Breuker, Leenes, and Winkels (eds), JURIX 2000: The 13th Annual Conference, IOS Press, pp. 35-47, 2000.

[DEG06] Jyotirmoy Deshmukh, E. Allen Emerson, and Prateek Gupta, "Automatic Verification of Parameterized Data Structures", *Tools and Algorithms for Construction and Analysis of Systems*, TACAS 2006. pp. 27-41.

[deBS69] deBakker, J.W. and Scott, D. "A Theory of Programs", Unpublished manuscript, 1969.

[Di76] Edsger W. Dijkstra, *Discipline of Programming*, Prentice-Hall, 1976

[Di89] Edsger W. Dijkstra, "In Reply to Comments", EWD1058, 1989.

[EC80] E. Allen Emerson, Edmund M. Clarke: "Characterizing Correctness Properties of Parallel Programs Using Fixpoints", *Int'l Conf. on Automata, Languages, and Programming*, ICALP 1980: 169-181.

[EH86] E. Allen Emerson, Joseph Y. Halpern: "'Sometimes' and 'Not Never' revisited: on branching versus linear time temporal logic", *J. ACM* 33(1): 151-178 (1986).

[EJ91] E. Allen Emerson and Charanjit S. Jutla, "Tree Automata, Mu-calculus, and Determinacy", FOCS 1991, pp 368-377.

[EL86] E. Allen Emerson, Chin-Laung Lei: "Efficient Model Checking in Fragments of the Propositional Mu-Calculus", *Logic in Computer Science*, LICS 1986: 267-278.

[EL87] E. Allen Emerson, Chin-Laung Lei: "Modalities for Model Checking: Branching Time Strikes Back", *Sci. of Comp. Prog.*, 8(3), 275-306, (1987).

[Em90] E. Allen Emerson, "Temporal and Modal Logic", *Handbook of Theoretical Computer Science, vol. B*, North-Holland, 1990.

[EN96] E. Allen Emerson, Kedar S. Namjoshi: *Automatic Verification of Parameterized Synchronous Systems*, *Computer Aided Verification*, CAV 1996: 87-98.

[EN98] E. Allen Emerson, Kedar S. Namjoshi: "Verification of a Parameterized Bus Arbitration Protocol", *Computer Aided Verification*, CAV 1998: 452-463.

[ES97] E. Allen Emerson, A. Prasad Sistla: "Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach", *ACM Trans. Program. Lang. Syst.*, 19(4): 617-638 (1997).

[FG99] Fausto Giunchiglia and Paolo Traverso, "Planning as Model Checking", *5th European Conference on Planning, ECP'99,* pp. 1-20, 1999.

[Fl67] Floyd, R. W., "Assigning meanings to programs", in *Proceedings of a Symposium in Applied Mathematics*, Vol. 19, Mathematical Aspects of Computer Science, Schwartz, J. T. (ed.), 19-32, 1967.

[FSS83] Jean-Claude Fernandez, J. Ph. Schwartz, Joseph Sifakis: "An Example of Specification and Verification in Cesar", *The Analysis of Concurrent Systems*, 1983: 199-210.

[GT99] F. Giunchiglia, P. Traverso, "Planning as Model Checking", *Proceedings of the Fifth European Workshop on Planning*, (ECP'99), Springer LNAI, 1999.

[H+06] J. Heath, M. Kwiatowska, G. Norman, D. Parker, O. Tymchysyn. "Probabilistic Model Checking of Complex Biological Pathways", in *Proc. Comp. Methods in Systems Biology*, (CSMB'06), C. Priami (ed), Lecture Notes in Bioinformatics vol. 4210, pp. 32-47, Springer, Oct. 2006.

[HvdP06] B. R. Havercourt and J. van der Pol, *5th Inter. Workshop on Parallel and Distributed Methods in verifiCation* PDMC, Springer LNCS no. 4346, 2006.

[Ho69] C. A. R. Hoare: "An Axiomatic Basis for Computer Programming", *Commun. ACM* 12(10): 576-580 (1969)

[Ho96] Gerard J. Holzmann: "On-The-Fly Model Checking", *ACM Comput. Surv.*, 28(4es): 120 (1996)

[IEEE05] IEEE-P1850-2005 Standard for Property Specification Language (PSL).

[Ja97] Daniel Jackson, "Mini-tutorial on Model Checking", *Third IEEE Intl. Symp. on Requirements Engineering*, Annapolis, Maryland, January 6-10, 1997.

[JPZ06] Marcin Jurdenski, Mike Paterson, and Uri Zwick, "A Deterministic Subexponential Algorithm for Parity Games", *ACM-SIAM Symp. on Algorthms for Discrete Systems*, 117-123, Jan. 2006.

[Ko83] Dexter Kozen: "Results on the Propositional Mu-Calculus", *Theor. Comput. Sci.*, 27: 333-354 : 1983.

[Kl56] Stephen C. Kleene, "Representation of Events in Nerve Nets and Finite Automata", in *Automata Studies*, eds. J. McCarthy and C. Shannon, pp. 3-42, Princeton Univ. Press, 1956.

[Kn28] B. Knaster (1928). "Un théorème sur les fonctions d'ensembles". *Ann. Soc. Polon. Math.* 6: 133Ž013134.

[KS92] Henry Kautz and Bart Selman, "Planning as Satisfiability", *Proceedings European Conference on Artificial Intelligence*, ECAI 1992.

[Ku94] Robert P. Kurshan, *Computer Aided Verification of Coordinating Processes: An Automata-theoretic Approach*, Princeton Univ. Press, 1994.

[La80] Leslie Lamport, "'Sometimes' is Sometimes 'Not Never' "- On the Temporal Logic of Programs, *Principles of Programming Languages*, POPL 1980: 174-185.

[Le59] C.Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs", *Bell Systems Technical Journal*, 38:985-999, 1959.

[LP85] Orna Lichtenstein and Amir Pnueli, "Checking that Finite State Programs meet their Linear Specification", *Principles of Programming Languages*, POPL 1985, 97-107.

[Lo+94] David E. Long, Anca Browne, Edmund M. Clarke, Somesh Jha, and Will Marero, "An improved Algorithm for the Evaluation of Fixpoint Expressions", *Computer Aided Verification*, CAV 1994, pp. 338-350.

[NASA97] *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, vol. II, A Practioners Companion*, [NASA-GB-01-97], 1997, 245 pp.

[NK00] Kedar S. Namjoshi, Robert P. Kurshan: "Syntactic Program Transformations for Automatic Abstraction". *Computer Aided Verification*, CAV 2000: 435-449.

[Ni02] National Institute of Standards and Technology, US Department of Commerce, "Software Errors Cost U.S. Economy $59.5 Billion Annually", NIST News Release, June 28, 2002. (www.nist.gov/public_affairs/releases/n02-10.htm).

[Pa69] David Park, "Fixpoint induction and proofs of program properties", in: B. Meltzer, D. Michie (Eds.), *Machine Intelligence*, Vol. 5, Edinburgh University Press, Edinburgh, Scotland, 1969.

[Pa81] David Park, "Concurrency and Automata on Infinite Sequences", *Theoretical Computer Science*, pp. 167-183, 1981.

[Pn77] Amir Pnueli, "The Temporal Logic of Programs", *Foundations of Computer Science*, FOCS, pp. 46-57, 1977.

[Pn79] Amir Pnueli, "The Temporal Semantics of Concurrent Programs", *Semantics of Concurrent Computation*, 1979: 1-20.

[Pr67] Arthur Prior, *Past, Present, and Future*, Oxford University Press, 1967.

[QS82] Jean-Pierre Queille, Joseph Sifakis, " Specification and verification of concurrent systems in CESAR", *Symposium on Programming*, Springer LNCS #137 1982: 337-351.

[Su78] Carl A. Sunshine, "Survey of protocol definition and verification techniques", *ACM SIGCOMM Computer Communication Review*, Volume 8, Issue 3 (July 1978) Pages: 35 - 41.

[Ta55] Alfred Tarski, "A lattice-theoretical fixpoint theorem and its applications", *Pac. J. Math.*, 5 (1955), 285-309.

[Tu36] Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungproblem", *Proc. London Math. Society*, 2 (42), 230-265, 1936. '" A Correction", *ibid*, 43, 544-546.

[Tu49] Alan M. Turing, "Checking a Large Routine", Paper for the EDSAC Inaugural Conference, 24 June 1949. Typescript published in *Report of a Conference on High Speed Automatic Calculating Machines*, pp 67-69.

[Va01] Moshe Y. Vardi, "Branching vs. Linear Time: Final Showdown", *Tools and Algorithmd for Construction and Analysis of Systems*, TACAS 2001: 1-22.

[VW86] Moshe Y. Vardi and Pierre Wolper, "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)", *Logic in Computer Science*, LICS 1986: 332-344.

[vB78] Gregor von Bochmann, "Finite State Description of Communication Protocols", *Computer Networks*, v. 2, pp. 361-372, 1978.

[Wa+00] W. Wang, Z. Hidvegi, A. Bailey, and A. Whinston, "E-Process Design and Assurance Using Model Checking", *IEEE Computer*, v. 33, no. 10, pp. 48-53, Oct. 2000.