From Church and Prior to PSL

Moshe Y. Vardi*

Rice University, Department of Computer Science, Rice University, Houston, TX 77251-1892, U.S.A., Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/∼vardi

Abstract. One of the surprising developments in the area of program verification is how ideas introduced originally by logicians in the 1950s ended up yielding by 2003 an industrial-standard property-specification language called PSL. This development was enabled by the equally unlikely transformation of the mathematical machinery of automata on infinite words, introduced in the early 1960s for second-order arithmetics, into effective algorithms for model-checking tools. This paper attempts to trace the tangled threads of this development.

1 Thread I: Classical Logic of Time

1.1 Reasoning about Sequential Circuits

The field of hardware verification seems to have been started in a little known 1957 paper by Alonzo Church, 1903–1995, in which he described the use of logic to specify *sequential circuits* [24]. A sequential circuit is a switching circuit whose output depends not only upon its input, but also on what its input has been in the past. A sequential circuit is a particular type of finite-state machine, which became a subject of study in mathematical logic and computer science in the 1950s.

Formally, a sequential circuit $C = (I, O, R, f, g, \mathbf{r}_0)$ consists of a finite set I of Boolean input signals, a finite set O of Boolean output signals, a finite set R of Boolean sequential elements, a transition function $f : 2^I \times 2^R \rightarrow 2^R$, an output function $g : 2^R \rightarrow 2^O$, and an initial state $\mathbf{r}_0 \in 2^R$. (We refer to elements of $I \cup O \cup R$ as *circuit elements*, and assume that I, O, and R are disjoint.) Intuitively, a state of the circuit is a Boolean assignment to the sequential elements. The initial state is $\mathbf{r}_0$. In a state $\mathbf{r} \in 2^R$, the Boolean assignment to the output signals is $g(\mathbf{r})$. When the circuit is in state $\mathbf{r} \in 2^R$ and it reads an input assignment $\mathbf{i} \in 2^I$, it changes its state to $f(\mathbf{i}, \mathbf{r})$.

A *trace* over a set V of Boolean variables is an infinite word over the alphabet 2^V, i.e., an element of $(2^V)^\omega$. A trace of the sequential circuit C is a trace over $I \cup O \cup R$ that satisfies some conditions. Specifically, a sequence $\tau = (\mathbf{i}_0, \mathbf{r}_0, \mathbf{o}_0), (\mathbf{i}_1, \mathbf{r}_1, \mathbf{o}_1), \ldots$, where $\mathbf{i}_j \in 2^I$, $\mathbf{o}_j \in 2^O$, and $\mathbf{r}_j \in 2^R$, is a trace of C if $\mathbf{r}_{j+1} = f(\mathbf{i}_j, \mathbf{r}_j)$ and $\mathbf{o}_j = g(\mathbf{r}_j)$, for $j \geq 0$. Thus, in modern terminology, Church was following the *linear-time* approach [81] (see discussion in Section 2.1). The set of traces of C is denoted by traces(C).

* Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, and ANI-0216467, by BSF grant 9800096, and by a gift from the Intel Corporation. The “Y” in the author’s middle name stands for “Ya’akov”

Church observed that we can associate with an infinite word $w = a_0, a_1, \ldots$ over an alphabet 2^V, a relational structure $M_w = (I\!N, \leq, V)$, with the naturals $I\!N$ as the domain, ordered by $\leq$, and extended by the set V of unary predicates, where $j \in p$, for $p \in V$, precisely when p *holds* (i.e., is assigned 1) in a_i.[1] We refer to such structures as *word structures*. When we refer to the *vocabulary* of such a structure, we refer explicitly only to V, taking $\leq$ for granted.

We can now specify traces using first-order logic (FO) sentences constructed from atomic formulas of the form $x = y$, $x \leq y$, and $p(x)$ for $p \in V = I \cup R \cup O$.[2] For example, the FO sentence

$$(\forall x)(\exists y)(x < y \wedge p(y))$$

says that p holds infinitely often in the trace. In a follow-up paper in 1963 [25], Church considered also specifying traces using *monadic second-order logic* (MSO), where in addition to first-order quantifiers, which range over the elements of $I\!N$, we allow also monadic second-order quantifiers, ranging over subsets of $I\!N$, and atomic formulas of the form $Q(x)$, where Q is a monadic predicate variable. (This logic is also called *S1S*, the "second-order theory of one successor function".) For example, the MSO sentence,

$$\begin{aligned}(\exists P)(\forall x)(\forall y)((((P(x) \wedge y = x+1) \rightarrow (\neg P(y)))\wedge \\ (((\neg P(x)) \wedge y = x+1) \rightarrow P(y)))\wedge \\ (x = 0 \rightarrow P(x)) \wedge (P(x) \rightarrow q(x))),\end{aligned}$$

where $x = 0$ is an abbrevaition for $(\neg(\exists z)(z < x))$ and $y = x + 1$ is an abbreviation for $(y > x \wedge \neg(\exists z)(x < z \wedge z < y))$, says that q holds at every even point on the trace. MSO was introduced in [15, 17, 43, 120].) In effect, Church was proposing to use classical logic (FO or MSO) as a logic of time, by focusing on word structures. The set of models of an FO or MSO sentence φ is denoted by models(φ).

Church posed two problems related to sequential circuits [24]:

- The DECISION problem: Given circuit C and a sentence φ, does φ hold in all traces of C? That is, does traces(C) $\subseteq$ models(φ) hold?
- The SYNTHESIS problem: Given sets I and O of input and output signals, and a sentence φ over the vocabulary $I \cup O$, construct, if possible, a sequential circuit C with input signals I and output signals O such that φ holds in all traces of C. That is, construct C such that traces(C) $\subseteq$ models(φ) holds.

In modern terminology, Church's DECISION problem is precisely the MODEL-CHECKING problem in the linear-time approach (see Section 2.2). This problem did not receive much attention after [24, 25], until the introduction of model checking in the early 1980s. In contrast, the SYNTHESIS problem has remained a subject of ongoing research; see [18, 75, 77, 105, 119]. One reason that the DECISION problem did not remain a subject of study, is the easy observation in [25] that the DECISION problem can be reduced to the VALIDITY problem in the underlying logic (FO or MSO). Given a sequential circuit C, we can easily generate an FO sentence α_C that holds in precisely all structures

[1] We overload notation here and treat p as both a Boolean variable and a predicate.
[2] We overload notation here and treat p as both a circuit element and a predicate symbol.

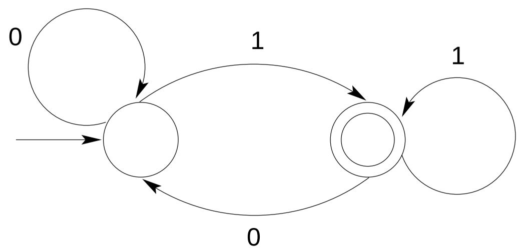
associated with traces of C. Intuitively, the sentence α_C simply has to encode the transition and output functions of C, which are Boolean functions. Then φ holds in all traces of C precisely when $\alpha_C \rightarrow \varphi$ holds in all word structures (of the appropriate vocabulary). Thus, to solve the DECISION problem we need to solve the VALIDITY problem over word structures. As we see next, this problem was solved in 1962.

1.2 Reasoning about Words

Church's DECISION problem was essentially solved in 1962 by Julius Richard Büchi, 1924–1984, who showed that the VALIDITY problem over word structures is decidable [16]. Actually, Büchi showed the decidability of the dual problem, which is the SATISFIABILITY problem for MSO over word structures. Büchi's approach consisted of extending the *automata-theoretic approach*, which was introduced a few years earlier [15, 43, 120] for finite word structures, to (infinite) word structures. To that end, Büchi extended automata theory to automata on infinite words.

A *nondeterministic Büchi automaton on words* (NBW) $A = (\Sigma, S, S_0, \rho, F)$ consists of a finite input alphabet Σ, a finite state set S, an initial state set $S_0 \subseteq S$, a transition relation $\rho \subseteq S \times \Sigma \times S$, and an accepting state set $F \subseteq S$. An NBW runs over an infinite input word $w = a_0, a_1, \ldots \in \Sigma^\omega$. A *run* of A on w is an infinite sequence $r = s_0, s_1, \ldots$ of states in S such that $s_0 \in S_0$, and $(s_i, a_i, s_{i+1}) \in \rho$, for $i \geq 0$. The run r is *accepting* if F is visited by r infinitely often; that is, $s_i \in F$ for infinitely many i's. The word w is *accepted* by A if A has an accepting run on w. The *language* of A, denoted $L(A)$, is the set of words accepted by A.

Example 1. We describe graphically an NBW that accepts all words over the alphabet $\{0, 1\}$ that contain infinitely many occurrences of 1. The arrow on the left designates the initial state, and the circle on the right designates an accepting state.



The class of languages accepted by NBWs forms the class of ω-*regular* languages, which are defined in terms of regular expressions augmented with the ω-power operator (e^ω denotes an infinitary iteration of e) [16].

The paradigmatic idea of the automata-theoretic approach is that we can compile high-level logical specifications into an equivalent low-level finite-state formalism.

Theorem 1. [16] *Given an MSO sentence φ over a vocabulary V, one can construct an NBW A_φ with alphabet 2^V such that a word w in $(2^V)^\omega$ is accepted by A_φ iff φ holds in the associated word structure M_w.*

The theorem says that models(φ) $= L(A_\varphi)$. Thus, the class of languages defined by MSO sentences is precisely the class of ω-regular languages. This result was inspired by an analogous earlier theorem for MSO over finite words [15, 43, 120], which showed that MSO over finite words defines precisely the class of regular languages.

To decide whether sentence φ is satisfiable, that is, whether models(φ) $\neq \emptyset$, we need to check that $L(A_\varphi) \neq \emptyset$. This turns out to be an easy problem. Let $A = (\Sigma, S, S_0, \rho, F)$ be an NBW. Construct a directed graph $G_A = (S, E_A)$, with S as the set of nodes, and $E_A = \{(s,t) : (s,a,t) \in \rho \text{ for some } a \in \Sigma\}$. The following lemma is implicit in [16] and more explicit in [121].

Lemma 1. $L(A) \neq \emptyset$ *iff there are states $s_0 \in S^0$ and $t \in F$ such that in G_A there is a path from s_0 to t and a path from t to itself.*

We thus obtain an algorithm for the SATISFIABILITY problem of MSO over word structures: given an MSO sentence φ, construct the NBW A_φ and check whether $L(A) \neq \emptyset$. Since the DECISION problem can be reduced to the SATISFIABILITY problem, this also solves the DECISION problem.

Neither Büchi nor Church analyzed the complexity of the DECISION problem. This had to wait until 1974. Define the function $exp(k,n)$ inductively as follows: $exp(0,n) = n$ and $exp(k+1,n) = 2^{exp(k,n)}$. We say that a problem is *nonelementary* if it can not be solved by an algorithm whose running time is bounded by $exp(k,n)$ for some fixed $k \geq 0$; that is, the running time cannot be bounded by a tower of exponentials of a fixed height. It is not too difficult to observe that the construction of the automaton A_φ in [16] is nonelementary. It was shown in [87, 113] that the SATISFIABILITY problem for MSO is nonelementary. In fact, the problem is already nonelementary for FO over finite words [113].

2 Thread II: Temporal Logic

2.1 From Aristotle to Kamp

The history of time in logic goes back to ancient times.[3] Aristotle pondered how to interpret sentences such as “Tomorrow there will be a sea fight,” or “Tomorrow there will not be a sea fight.” Medieval philosophers also pondered the issue of time.[4] By the Renaissance period, philosophical interest in the logic of time seems to have waned.

[3] For a detailed history of temporal logic from ancient times to the modern period, see [91].

[4] For example, William of Ockham, 1288–1348, wrote (rather obscurely for the modern reader): “Wherefore the difference between present tense propositions and past and future tense propositions is that the predicate in a present tense proposition stands in the same way as the subject, unless something added to it stops this; but in a past tense and a future tense proposition it varies, for the predicate does not merely stand for those things concerning which it is truly predicated in the past and future tense propositions, because in order for such a proposition to be true, it is not sufficient that that thing of which the predicate is truly predicated (whether by

There were some stirrings of interest in the 19th century, by Boole and Peirce. Peirce wrote:

> "Time has usually been considered by logicians to be what is called 'extra-logical' matter. I have never shared this opinion. But I have thought that logic had not yet reached the state of development at which the introduction of temporal modifications of its forms would not result in great confusion; and I am much of that way of thinking yet."

There were also some stirrings of interest in the first half of the 20th century, but the birth of modern temporal logic is unquestionably credited to Arthur Norman Prior, 1914-1969. Prior was a philosopher, who was interested in theological and ethical issues. His own religious path was somewhat convoluted; he was born a Methodist, converted to Presbytarianism, became an atheist, and ended up an agnostic. In 1949, he published a book titled "Logic and The Basis of Ethics". He was particularly interested in the conflict between the assumption of *free will* ("the future is to some extent, even if it is only a very small extent, something we can make for ourselves"), *foredestination* ("of what will be, it has now been the case that it will be"), and *foreknowledge* ("there is a deity who infallibly knows the entire future"). He was also interested in modal logic [102]. This confluence of interests led Prior to the development of *temporal logic*. [5] His wife, Mary Prior, recalled after his death:

> "I remember his waking me one night [in 1953], coming and sitting on my bed, . . ., and saying he thought one could make a formalised tense logic."

Prior lectured on his new work when he was the John Locke Lecturer at the University of Oxford in 1955–6, and published his book "Time and Modality" in 1957 [100].[6] In this book, he presented a temporal logic that is propositional logic extended with two temporal connectives, F and P, corresponding to "sometime in the future" and "sometime in the past". A crucial feature of this logic is that it has an implicit notion of "now", which is treated as an *indexical*, that is, it depends on the context of utterance for its meaning. Both future and past are defined with respect to this implicit "now".

It is interesting to note that the *linear* vs. *branching* time dichotomy, which has been a subject of some controversy in the computer science literature since 1980 (see [126]), has been present from the very beginning of temporal-logic development. In Prior's early work on temporal logic, he assumed that time was linear. In 1958, he received a letter from Saul Kripke,[7] who wrote

a verb in the present tense or in the future tense) is that which the subject denotes, although it is required that the very same predicate is truly predicated of that which the subject denotes, by means of what is asserted by such a proposition."

[5] An earlier term was *tense logic*; the term *temporal logic* was introduced in [90]. The technical distinction between the two terms seems fuzzy.

[6] Due to the arcane infix notation of the time, the book may not be too accessible to modern readers, who may have difficulties parsing formulas such as $CKMpMqAMKpMqMKqMp$.

[7] Kripke was a high-school student, not quite 18, in Omaha, Nebraska. Kripke's interest in modal logic was inspired by a paper by Prior on this subject [103]. Prior turned out to be the referee of Kripke's first paper [74].

> "In an indetermined system, we perhaps should not regard time as a linear series, as you have done. Given the present moment, there are several possibilities for what the next moment may be like – and for each possible next moment, there are several possibilities for the moment after that. Thus the situation takes the form, not of a linear sequence, but of a 'tree'."

Prior immediately saw the merit of Kripke's suggestion: "the determinist sees time as a line, and the indeterminist sees times as a system of forking paths." He went on to develop two theories of branching time, which he called "Ockhamist" and "Peircean". (Prior did not use path quantifiers; those were introduced later, in the 1980s. See Section 3.2.)

While the introduction of branching time seems quite reasonable in the context of trying to formalize free will, it is far from being simple philosophically. Prior argued that the nature of the course of time is branching, while the nature of a course of events is linear [101]. In contrast, it was argued in [90] that the nature of time is linear, but the nature of the course of events is branching: "We have 'branching *in* time,' not 'branching *of* time'."[8]

During the 1960s, the development of temporal logic continued through both the linear-time approach and the branching-time approach. There was little connection, however, between research on temporal logic and research on classical logics, as described in Section 1. That changed in 1968, when Johan Anthony Willem (Hans) Kamp tied together the two threads in his doctoral dissertation.

Theorem 2. [70] *Linear temporal logic with past and binary temporal connectives ("strict until" and "strict since") has precisely the expressive power of FO over the ordered naturals (with monadic vocabularies).*

It should be noted that Kamp's Theorem is actually more general and asserts expressive equivalence of FO and temporal logic over all "Dedekind-closed orders". The introduction of binary temporal connectives by Kamp was necessary for reaching the expressive power of FO; *unary* linear temporal logic, which has only unary temporal connectives, is weaker than FO [51]. The theorem refers to FO formulas with one free variable, which are satisfied at an element of a structure, analogously to temporal logic formulas, which are satisfied at a point of time.

It should be noted that one direction of Kamp's Theorem, the translation from temporal logic to FO, is quite straightforward; the hard direction is the translation from FO to temporal logic. Both directions are algorithmically effective; translating from temporal logic to FO involves a linear blowup, but translation in the other direction involves a nonelementary blowup.

If we focus on FO sentences rather than FO formulas, then they define sets of traces (a sentence φ defines models(φ)). A characterization of of the expressiveness of FO sentences over the naturals, in terms of their ability to define sets of traces, was obtained in 1979.

[8] One is reminded of St. Augustin, who said in his *Confessions*: "What, then, is time? If no one asks me, I know; but if I wish to explain it to some who should ask me, I do not know."

Theorem 3. [118] *FO sentences over naturals have the expressive power of $*$-free ω-regular expressions.*

Recall that MSO defines the class of ω-regular languages. It was already shown in [44] that FO over the naturals is weaker expressively than MSO over the naturals. Theorem 3 was inspired by an analogous theorem in [86] for finite words.

2.2 The Temporal Logic of Programs

There were some early observations that temporal logic can be applied to programs. Prior stated: "There are practical gains to be had from this study too, for example, in the representation of time-delay in computer circuits" [101]. Also, a discussion of the application of temporal logic to processes, which are defined as "programmed sequences of states, deterministic or stochastic" appeared in [90].

The "big bang" for the application of temporal logic to program correctness occurred with Amir Pnueli's 1977 paper [93]. In this paper, Pnueli, inspired by [90], advocated using future linear temporal logic (LTL) as a logic for the specification of non-terminating programs.

LTL is a temporal logic with two temporal connectives, "next" and "until".[9] In LTL, formulas are constructed from a set $Prop$ of atomic propositions using the usual Boolean connectives as well as the unary temporal connective X ("next"), and the binary temporal connective U ("until"). Additional unary temporal connectives F ("eventually"), and G ("always") can be defined in terms of U. Note that all temporal connectives refer to the future here, in contrast to Kamp's "strict since" operator, which refers to the past. Thus, LTL is a *future temporal logic*. For extensions with past temporal connectives, see [83, 84, 123].

LTL is interpreted over traces over the set $Prop$ of atomic propositions. For a trace τ and a point $i \in I\!N$, the notation $\tau, i \models \varphi$ indicates that the formula φ holds at the point i of the trace τ. Thus, the point i is the implicit "now" with respect to which the formula is interpreted. We have that

- $\tau, i \models p$ if p holds at $\tau(i)$,
- $\tau, i \models X\varphi$ if $\tau, i+1 \models \varphi$, and
- $\tau, i \models \varphi U \psi$ if for some $j \geq i$, we have $\tau, j \models \psi$ and for all k, $i \leq k < j$, we have $\tau, k \models \varphi$.

The temporal connectives F and G can be defined in terms of the temporal connective U; $F\varphi$ is defined as $\mathbf{true}\ U\varphi$, and $G\varphi$ is defined as $\neg F \neg \varphi$. We say that τ *satisfies* a formula φ, denoted $\tau \models \varphi$, iff $\tau, 0 \models \varphi$. We denote by models(φ) the set of traces satisfying φ.

As an example, the LTL formula $G(\mathit{request} \rightarrow F\ \mathit{grant})$, which refers to the atomic propositions *request* and *grant*, is true in a trace precisely when every state in the trace in which *request* holds is followed by some state in the (non-strict) future in which

[9] Unlike Kamp's "strict until" ("p strict until q" requires q to hold in the strict future), Pnueli's "until" is not strict ("p until q" can be satisfied by q holding now), which is why the "next" connective is required.

grant holds. Also, the LTL formula $G(\textit{request} \rightarrow (\textit{request}\ U\ \textit{grant}))$ is true in a trace precisely if, whenever *request* holds in a state of the trace, it holds until a state in which *grant* holds is reached.

The focus on satisfaction at 0, called *initial semantics*, is motivated by the desire to specify computations at their starting point. It enables an alternative version of Kamp's Theorem, which does not require past temporal connectives, but focuses on initial semantics.

Theorem 4. [56] *LTL has precisely the expressive power of FO over the ordered naturals (with monadic vocabularies) with respect to initial semantics.*

As we saw earlier, FO has the expressive power of star-free ω-regular expressions over the naturals. Thus, LTL has the expressive power of star-free ω-regular expressions (see [95]), and is strictly weaker than MSO. An interesting outcome of the above theorem is that it lead to the following assertion regarding LTL [88]: "The corollary due to Meyer – I have to get in my controversial remark – is that that [Theorem 4] makes it theoretically uninteresting." Developments since 1980 have proven this assertion to be overly pessimistic on the merits of LTL.

Pnueli also discussed the analog of Church's DECISION problem: given a finite-state program P and an LTL formula φ, decide if φ holds in all traces of P. Just like Church, Pnueli observed that this problem can be solved by reduction to MSO. Rather than focus on sequential circuits, Pnueli focused on programs, modeled as (labeled) *transition systems* [71]. A transition system $M = (W, W_0, R, V)$ consists of a set W of states that the system can be in, a set $W_0 \subseteq W$ of initial states, a transition relation $R \subseteq W^2$ that indicates the allowable state transitions of the system, and an assignment $V : W \rightarrow 2^{Prop}$ of truth values to the atomic propositions in each state of the system. (A transition system is essentially a Kripke structure [10].) A *path* in M that *starts at* u is a possible infinite behavior of the system starting at u, i.e., it is an infinite sequence $u_0, u_1 \ldots$ of states in W such that $u_0 = u$, and $(u_i, u_{i+1}) \in R$ for all $i \geq 0$. The sequence $V(u_0), V(u_1) \ldots$ is a *trace* of M that *starts at* u. It is the sequence of truth assignments visited by the path. The *language* of M, denoted $L(M)$, consists of all traces of M that start at a state in W_0. Note that $L(M)$ is a language of infinite words over the alphabet 2^{Prop}. The language $L(M)$ can be viewed as an abstract description of the system M, describing all possible traces. We say that M *satisfies* an LTL formula φ if all traces in $L(M)$ satisfy φ, that is, if $L(M) \subseteq \text{models}(\varphi)$. When W is finite, we have a finite-state system, and can apply algorithmic techniques.

What about the complexity of LTL reasoning? Recall from Section 1 that satisfiability of FO over trace structures is nonelementary. In contrast, it was shown in [60, 61, 108–110, 132, 133] that LTL SATISFIABILITY is elementary; in fact, it is PSPACE-complete. It was also shown that the DECISION problem for LTL with respect to finite transition systems is PSPACE-complete [108–110]. The basic technique for proving these elementary upper bounds is the *tableau* technique, which was adapted from *dynamic logics* [98] (see Section 3.1). Thus, even though FO and LTL are expressively equivalent, they have dramatically different computational properties, as LTL reasoning is in PSPACE, while FO reasoning is nonelementary.

The second "big bang" in the application of temporal logic to program correctness was the introduction of *model checking* by Edmund Melson Clarke and Ernest Allen

Emerson [28] and by Jean-Pierre Queille and Joseph Sifakis [104]. The two papers used two different branching-time logics. Clarke and Emerson used CTL (inspired by the branching-time logic UB of [9]), which extends LTL with existential and universal path quantifiers E and A. Queille and Sifakis used a logic introduced by Leslie Lamport [81], which extends propositional logic with the temporal connectives POT (which corresponds to the CTL operator EF) and $INEV$ (which corresponds to the CTL operator AF). The focus in both papers was on model checking, which is essentially what Church called the DECISION problem: does a given finite-state program, viewed as a finite transition system, satisfy its given temporal specification. In particular, Clarke and Emerson showed that model checking transition systems of size m with respect to formulas of size n can be done in time polynomial in m and n. This was refined later to $O(mn)$ (even in the presence of *fairness* constraints, which restrict attention to certain infinite paths in the underlying transition system) [29, 30]. We drop the term "DECISION problem" from now on, and replace it with the term "MODEL-CHECKING problem".[10]

It should be noted that the linear complexity of model checking refers to the size of the transition system, rather than the size of the program that gave rise to that system. For sequential circuits, transition-system size is essentially exponential in the size of the description of the circuit (say, in some Hardware Description Language). This is referred to as the "state-explosion problem" [31]. In spite of the state-explosion problem, in the first few years after the publication of the first model-checking papers in 1981-2, Clarke and his students demonstrated that model checking is a highly successful technique for automated program verification [13, 33]. By the late 1980s, automated verification had become a recognized research area. Also by the late 1980s, *symbolic* model checking was developed [19, 20], and the SMV tool, developed at CMU by Kenneth Laughlin McMillan [85], was starting to have an industrial impact. See [27] for more details.

The detailed complexity analysis in [29] inspired a similar detailed analysis of linear time model checking. It was shown in [82] that model checking transition systems of size m with respect to LTL formulas of size n can be done in time $m2^{O(n)}$. (This again was shown using a tableau-based technique.) While the bound here is exponential in n, the argument was that n is typically rather small, and therefore an exponential bound is acceptable.

2.3 Back to Automata

Since LTL can be translated to FO, and FO can be translated to NBW, it is clear that LTL can be translated to NBW. Going through FO, however, would incur, in general, a nonelementary blowup. In 1983, Pierre Wolper, Aravinda Prasad Sistla, and I showed that this nonelementary blowup can be avoided.

Theorem 5. [130, 134] *Given an LTL formula φ of size n, one can construct an NBW A_φ of size $2^{O(n)}$ such that a trace σ satisfies φ if and only if σ is accepted by A_φ.*

[10] The model-checking problem is analogous to database query evaluation, where we check the truth of a logical formula, representing a query, with respect to a database, viewed as a finite relational structure. Interestingly, the study of the complexity of database query evaluation started about the same time as that of model checking [122].

It now follows that we can obtain a PSPACE algorithm for LTL SATISFIABILITY: given an LTL formula φ, we construct A_φ and check that $A_\varphi \neq \emptyset$ using the graph-theoretic approach described earlier. We can avoid using exponential space, by constructing the automaton *on the fly* [130, 134].

What about model checking? We know that a transition system M satisfies an LTL formula φ if $L(M) \subseteq$ models(φ). It was then observed in [129] that the following are equivalent:

– M satisfies φ
– $L(M) \subseteq$ models(φ)
– $L(M) \subseteq L(A_\varphi)$
– $L(M) \cap ((2^{Prop})^\omega - L(A_\varphi)) = \emptyset$
– $L(M) \cap L(A_{\neg\varphi}) = \emptyset$
– $L(M \times A_{\neg\varphi}) = \emptyset$

Thus, rather than complementing A_φ using an exponential complementation construction [16, 76, 112], we complement the LTL property using logical negation. It is easy to see that we can now get the same bound as in [82]: model checking programs of size m with respect to LTL formulas of size n can be done in time $m2^{O(n)}$. Thus, the optimal bounds for LTL satisfiability and model checking can be obtained without resorting to ad-hoc tableau-based techniques; the key is the exponential translation of LTL to NBW.

One may wonder whether this theory is practical. Reduction to practice took over a decade of further research, which saw the development of

– an optimized search algorithm for explicit-state model checking [36, 37],
– a symbolic, BDD-based[11] algorithm for NBW nonemptiness [19, 20, 49],
– symbolic algorithms for LTL to NBW translation [19, 20, 32], and
– an optimized explicit algorithm for LTL to NBW translation [58].

By 1995, there were two model-checking tools that implemented LTL model checking via the automata-theoretic approach: Spin [68] is an explicit-state LTL model checker, and Cadence's SMV is a symbolic LTL model checker.[12] See [127] for a description of algorithmic developments since the mid 1990s. Additional tools today are *VIS* [12], *NuSMV* [26], and *SPOT* [38].

It should be noted that Robert Kurshan developed the automata-theoretic approach independently, also going back to the 1980s [1, 2, 78]. In his approach (as also in [106, 134]), one uses automata to represent both the system and its specification [79].[13] The first implementation of COSPAN, a model-checking tool that is based on this approach [62], also goes back to the 1980s; see [80].

[11] To be precise, one should use the acronym ROBDD, for Reduced Ordered Binary Decision Diagrams [14].
[12] Cadence's SMV is also a CTL model checker. See
`www.cadence.com/webforms/cbl_software/index.aspx`.
[13] The connection to automata is somewhat difficult to discern in the early papers [1, 2].

2.4 Enhancing Expressiveness

Can the development of LTL model checking [82, 129] be viewed as a satisfactory solution to Church's DECISION problem? Almost, but not quite, since, as we observed earlier, LTL is not as expressive as MSO, which means that LTL is expressively weaker than NBW. Why do we need the expressive power of NBWs? First, note that once we add fairness to transitions systems (sse [29, 30]), they can be viewed as variants of NBWs. Second, there are good reasons to expect the specification language to be as expressive as the underlying model of programs [94]. Thus, achieving the expressive power of NBWs, which we refer to as *ω-regularity*, is a desirable goal. This motivated efforts since the early 1980s to extend LTL.

The first attempt along this line was made by Wolper [132, 133], who defined ETL (for *Extended Temporal Logic*), which is LTL extended with grammar operators. He showed that ETL is more expressive than LTL, while its SATISFIABILITY problem can still be solved in exponential time (and even PSPACE [108–110]). Then, Sistla, Wolper and I showed how to extend LTL with automata connectives, reaching ω-regularity, without losing the PSPACE upper bound for the SATISFIABILITY problem [130, 134]. Actually, three syntactical variations, denoted ETL_f, ETL_l, and ETL_r were shown to be expressively equivalent and have these properties [130, 134].

Two other ways to achieve ω-regularity were discovered in the 1980s. The first is to enhance LTL with monadic second-order quantifiers as in MSO, which yields a logic, QPTL, with a nonelementary SATISFIABILITY problem [111, 112]. The second is to enhance LTL with least and greatest fixpoints [6, 124], which yields a logic, μLTL, that achieves ω-regularity, and has a PSPACE upper bound on its SATISFIABILITY and MODEL-CHECKING problems [124]. For example, the (not too readable) formula

$$(\nu P)(\mu Q)(P \wedge X(p \vee Q)),$$

where ν and μ denote greatest and least fixpoint operators, respectively, is equivalent to the LTL formula GFp, which says that p holds infinitely often.

3 Thread III: Dynamic and Branching-Time Logics

3.1 Dynamic Logics

In 1976, a year before Pnueli proposed using LTL to specify programs, Vaughan Ronald Pratt proposed using *dynamic logic*, an extension of modal logic, to specify programs [96].[14] In modal logic $\Box\varphi$ means that φ holds in all worlds that are possible with respect to the current world [10]. Thus, $\Box\varphi$ can be taken to mean that φ holds after an execution of a program step, taking the transition relation of the program to be the possibility relation of a Kripke structure. Pratt proposed the addition of dynamic modalities $[e]\varphi$, where e is a program, which asserts that φ holds in all states reachable by an execution of the program e. Dynamic logic can then be viewed as an extension of Hoare logic, since $\psi \rightarrow [e]\varphi$ corresponds to the Hoare triple $\{\psi\}e\{\varphi\}$ (see [3]). See [64] for an extensive coverage of dynamic logic.

[14] See discussion of precursor and related developments, such as [21, 34, 50, 107], in [64].

In 1977, a propositional version of Pratt's dynamic logic, called PDL, was proposed, in which programs are regular expressions over atomic programs [52, 53]. It was shown there that the SATISFIABILITY problem for PDL is in NEXPTIME and EXPTIME-hard. Pratt then proved an EXPTIME upper bound, adapting tableau techniques from modal logic [97, 98]. (We saw earlier that Wolper then adapted these techniques to linear-time logic.)

Pratt's dynamic logic was designed for terminating programs, while Pnueli was interested in nonterminating programs. This motivated various extensions of dynamic logic to nonterminating programs [67, 115, 114, 116]. Nevertheless, these logics are much less natural for the specification of ongoing behavior than temporal logic. They inspired, however, the introduction of the (*modal*) μ-*calculus* by Dexter Kozen [72, 73]. The μ-calculus is an extension of modal logic with least and greatest fixpoints. It subsumes expressively essentially all dynamic and temporal logics [11]. Kozen's paper was inspired by previous papers that showed the usefulness of fixpoints in characterizing correctness properties of programs [45, 92] (see also [99]). In turn, the μ-calculus inspired the introduction of μLTL, mentioned earlier. The μ-calculus also played an important role in the development of symbolic model checking [19, 20, 49].

3.2 Branching-Time Logics

Dynamic logic provided a branching-time approach to reasoning about programs, in contrast to Pnueli's linear-time approach. Lamport was the first to study the dichotomy between linear and branching time in the context of program correctness [81]. This was followed by the introduction of the branching-time logic UB, which extends unary LTL (LTL without the temporal connective "until") with the existential and universal path quantifiers, E and A [9]. Path quantifiers enable us to quantify over different future behavior of the system. By adapting Pratt's tableau-based method for PDL to UB, it was shown that its SATISFIABILITY problem is in EXPTIME [9]. Clarke and Emerson then added the temporal conncetive "until" to UB and obtained CTL [28]. (They did not focus on the SATISFIABILITY problem for CTL, but, as we saw earlier, on its MODEL-CHECKING problem; the SATISFIABILITY problem was shown later to be solvable in EXPTIME [47].) Finally, it was shown that LTL and CTL have incomparable expressive power, leading to the introduction of the branching-time logic CTL*, which unifies LTL and CTL [46, 48].

The key feature of branching-time logics in the 1980s was the introduction of explicit path quantifiers in [9]. This was an idea that was not discovered by Prior and his followers in the 1960s and 1970s. Most likely, Prior would have found CTL* satisfactory for his philosophical applications and would have seen no need to introduce the "Ockhamist" and "Peircean" approaches.

3.3 Combining Dynamic and Temporal Logics

By the early 1980s it became clear that temporal logics and dynamic logics provide two distinct perspectives for specifying programs: the first is *state* based, while the second is *action* based. Various efforts have been made to combine the two approaches. These include the introduction of *Process Logic* [63] (branching time), *Yet Another*

Process Logic [128] (branching time), *Regular Process Logic* [66] (linear time), *Dynamic LTL* [59] (linear time), and *RCTL* [8] (branching time), which ultimately evolved into *Sugar* [7]. RCTL/Sugar is unique among these logics in that it did not attempt to borrow the action-based part of dynamic logic. It is a state-based branching-time logic with no notion of actions. Rather, what it borrowed from dynamic logic was the use of regular-expression-based dynamic modalities. Unlike dynamic logic, which uses regular expressions over program statements, RCTL/Sugar uses regular expressions over state predicates, analogously to the automata of ETL [130, 134], which run over sequences of formulas.

4 Thread IV: From LTL to ForSpec and PSL

In the late 1990s and early 2000s, model checking was having an increasing industrial impact. That led to the development of two industrial temporal logics based on LTL: *ForSpec*, developed by Intel, and *PSL*, developed by an industrial standards committee.

4.1 From LTL to ForSpec

Intel's involvement with model checking started in 1990, when Kurshan, spending a sabbatical year in Israel, conducted a successful feasibility study at the Intel Design Center (IDC) in Haifa, using COSPAN, which at that point was a prototype tool; see [80]. In 1992, IDC started a pilot project using SMV. By 1995, model checking was used by several design projects at Intel, using an internally developed model checker based on SMV. Intel users have found CTL to be lacking in expressive power and the Design Technology group at Intel developed its own specification language, FSL. The FSL language was a linear-time logic, and it was model checked using the automata-theoretic approach, but its design was rather ad-hoc, and its expressive power was unclear; see [54].

In 1997, Intel's Design Technology group at IDC embarked on the development of a second-generation model-checking technology. The goal was to develop a model-checking engine from scratch, as well as a new specification language. A BDD-based model checker was released in 1999 [55], and a SAT-based model checker was released in 2000 [35].

I got involved in the design of the second-generation specification language in 1997. That language, ForSpec, was released in 2000 [5]. The first issue to be decided was whether the language should be linear or branching. This led to an in-depth examination of this issue [126], and the decision was to pursue a linear-time language. An obvious candidate was LTL; we saw that by the mid 1990s there were both explicit-state and symbolic model checkers for LTL, so there was no question of feasibility. I had numerous conversations with Limor Fix, Michael Hadash, Yonit Kesten, and Moshe Sananes on this issue. The conclusion was that LTL is not expressive enough for industrial usage. In particular, many properties that are expressible in FSL are not expressible in LTL. Thus, it turned out that the theoretical considerations regarding the expressiveness of LTL, i.e., its lack of ω-regularity, had practical significance. I offered two extensions of LTL; as we saw earlier both ETL and μLTL achieve ω-regularity and have the same complexity as LTL. Neither of these proposals was accepted, due to the perceived difficulty of usage of such logics by Intel validation engineers, who typically have only basic familiarity with automata theory and logic.

These conversations continued in 1998, now with Avner Landver. Avner also argued that Intel validation engineers would not be receptive to the automata-based formalism of ETL. Being familiar with RCTL/Sugar and its dynamic modalities [7, 8], he asked me about regular expressions, and my answer was that regular expressions are equivalent to automata [69], so the automata of ETL_f, which extends LTL with automata on *finite* words, can be replaced by regular expressions over state predicates. This lead to the development of *RELTL*, which is LTL augmented by the dynamic regular modalities of dynamic logic (interpreted linearly, as in ETL). Instead of the dynamic-logic notation $[e]\varphi$, ForSpec uses the more readable (to engineers) $(e\ \mathsf{triggers}\ \varphi)$, where e is a regular expression over state predicates (e.g., $(p \vee q)^*$, $(p \wedge q)$), and φ is a formula. Semantically, $\tau, i \models (e\ \mathsf{triggers}\ \varphi)$ if, for all $j \geq i$, if $\tau[i,j]$ (that is, the finite word $\tau(i), \ldots, \tau(j)$) "matches" e (in the intuitive formal sense), then $\tau, j \models \varphi$; see [22]. Using the ω-regularity of ETL_f, it is now easy to show that RELTL also achieves ω-regularity [5].

While the addition of dynamic modalities to LTL is sufficient to achieve ω-regularity, we decided to also offer direct support to two specification modes often used by verification engineers at Intel: *clocks* and *resets*. Both clocks and resets are features that are needed to address the fact that modern semiconductor designs consist of interacting parallel modules. While clocks and resets have a simple underlying intuition, defining their semantics formally is quite nontrivial. ForSpec is essentially RELTL, augmented with features corresponding to clocks and resets, as we now explain.

Today's semiconductor designs are still dominated by synchronous circuits. In synchronous circuits, clock signals synchronize the sequential logic, providing the designer with a simple operational model. While the asynchronous approach holds the promise of greater speed (see [23]), designing asynchronous circuits is significantly harder than designing synchronous circuits. Current design methodology attempts to strike a compromise between the two approaches by using multiple clocks. This results in architectures that are globally asynchronous but locally synchronous. The temporal-logic literature mostly ignores the issue of explicitly supporting clocks. ForSpec supports multiple clocks via the notion of *current clock*. Specifically, ForSpec has a construct $\mathsf{change\_on}\ c\ \varphi$, which states that the temporal formula φ is to be evaluated with respect to the clock c; that is, the formula φ is to be evaluated in the trace defined by the high phases of the clock c. The key feature of clocks in ForSpec is that each subformula may advance according to a different clock [5].

Another feature of modern designs' consisting of interacting parallel modules is the fact that a process running on one module can be reset by a signal coming from another module. As noted in [117], reset control has long been a critical aspect of embedded control design. ForSpec directly supports reset signals. The formula $\mathsf{accept\_on}\ a\ \varphi$ states that the property φ should be checked only until the arrival of the reset signal a, at which point the check is considered to have *succeeded*. In contrast, $\mathsf{reject\_on}\ r\ \varphi$ states that the property φ should be checked only until the arrival of the reset signal r, at which point the check is considered to have *failed*. The key feature of resets in ForSpec is that each subformula may be reset (positively or negatively) by a different reset signal; for a longer discussion see [5].

ForSpec is an industrial property-specification language that supports hardware-oriented constructs as well as uniform semantics for formal and dynamic validation, while at the same time it has a well understood expressiveness (ω-regularity) and computational complexity (SATISFIABILITY and MODEL-CHECKING problems have the same complexity for ForSpec as for LTL) [5]. The design effort strove to find an acceptable compromise, with trade-offs clarified by theory, between conflicting demands, such as expressiveness, usability, and implementability. Clocks and resets, both important to hardware designers, have a clear intuitive semantics, but formalizing this semantics is nontrivial. The rigorous semantics, however, not only enabled mechanical verification of various theorems about the language, but also served as a reference document for the implementors. The implementation of model checking for ForSpec followed the automata-theoretic approach, using *alternating* automata as advocated in [125] (see [57]).

4.2 From ForSpec to PSL

In 2000, the Electronic Design Automation Association instituted a standardization body called *Accellera*.[15] Accellera's mission is to drive worldwide development and use of standards required by systems, semiconductor and design tools companies. Accellera decided that the development of a standard specification language is a requirement for formal verification to become an industrial reality (see [80]). Since the focus was on specifying properties of designs rather than designs themselves, the chosen term was "property specification language" (PSL). The PSL standard committee solicited industrial contributions and received four language contributions: *CBV*, from Motorola, ForSpec, from Intel, *Temporal **e***, from Verisity [89], and Sugar, from IBM.

The committee's discussions were quite fierce.[16] Ultimately, it became clear that while technical considerations play an important role, industrial committees' decisions are ultimately made for business considerations. In that contention, IBM had the upper hand, and Accellera chose Sugar as the base language for PSL in 2003. At the same time, the technical merits of ForSpec were accepted and PSL adopted all the main features of ForSpec. In essence, PSL (the current version 1.1) is LTL, extended with dynamic modalities (referred to as the *regular layer*), clocks, and resets (called *aborts*). PSL did inherit the syntax of Sugar, and does include a branching-time extension as an acknowledgment to Sugar.[17]

There was some evolution of PSL with respect to ForSpec. After some debate on the proper way to define resets [4], ForSpec's approach was essentially accepted after some reformulation [41]. ForSpec's fundamental approach to clocks, which is semantic, was accepted, but modified in some important details [42]. In addition to the dynamic modalities, borrowed from dynamic logic, PSL also has weak dynamic modalities [40],

[15] See `http://www.accellera.org/`.
[16] See `http://www.eda-stds.org/vfv/`.
[17] See [39] and language reference manual at `http://www.eda.org/vfv/docs/PSL-v1.1.pdf` and

which are reminiscent of "looping" modalities in dynamic logic [67, 65]. Today PSL 1.1 is an IEEE Standard 1850–2005, and continues to be refined by the IEEE P1850 PSL Working Group.[18]

Practical use of ForSpec and PSL has shown that the regular layer (that is, the dynamic modalities), is highly popular with verification engineers. Another standardized property specification language, called *SVA* (for SystemVerilog Assertions), is based, in essence, on that regular layer [131].

5 Contemplation

The evolution of ideas, from Church and Prior to PSL, seems to be an amazing development. It reminds me of the medieval period, when building a cathedral spanned more than a mason's lifetime. Many masons spend their whole lives working on a cathedral, never seeing it to completion. We are fortunate to see the completion of this particular "cathedral". Just like the medieval masons, our contributions are often smaller than we'd like to consider them, but even small contributions can have a major impact. Unlike the medieval cathedrals, the scientific cathedral has no architect; the construction is driven by a complex process, whose outcome is unpredictable. Much that has been discovered is forgotten and has to be rediscovered. It is hard to fathom what our particular "cathedral" will look like in 50 years.

Acknowledgments: I am grateful to E. Clarke, A. Emerson, R. Goldblatt, A. Pnueli, P. Sistla, P. Wolper for helping me trace the many threads of this story, to D. Fisman, C. Eisner, J. Halpern, D. Harel and T. Wilke for their many useful comments on earlier drafts of this paper, and to S. Nain, K. Rozier, and D. Tabakov for proofreading earlier drafts. I'd also like to thank K. Rozier for her help with graphics.

References

1. S. Aggarwal and R.P. Kurshan. Automated implementation from formal specification. In *Proc. 4th Int'l Workshop on Protocol Specification, Testing and Verification*, pages 127–136. North-Holland, 1984.
2. S. Aggarwal, R.P. Kurshan, and D. Sharma. A language for the specification and analysis of protocols. In *Proc. 3rd Int'l Workshop on Protocol Specification, Testing, and Verification*, pages 35–50. North-Holland, 1983.
3. K. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 2006.
4. R. Armoni, D. Bustan, O. Kupferman, and M.Y. Vardi. Resets vs. aborts in linear temporal logic. In *Proc. 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 65 – 80. Springer, 2003.
5. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th Int. Conf. on Tools and Algorithms*

[18] See `http://www.eda.org/ieee-1850/`.

for the Construction and Analysis of Systems, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211. Springer, 2002.
6. B. Banieqbal and H. Barringer. Temporal logic with fixed points. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1987.
7. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer, 2001.
8. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 184–194. Springer, 1998.
9. M. Ben-Ari, Z. Manna, and A. Pnueli. The logic of nexttime. In *Proc. 8th ACM Symp. on Principles of Programming Languages*, pages 164–176, 1981.
10. P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2002.
11. J. Bradfield and C. Stirling. PDL and modal μ-calculus. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*. Elsevier, 2006.
12. R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, T. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *Proc 8th Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.
13. M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computing*, C-35:1035–1044, 1986.
14. R.E. Bryant. Graph-based algorithms for Boolean-function manipulation. *IEEE Transactions on Computing*, C-35(8):677–691, 1986.
15. J.R. Büchi. Weak second-order arithmetic and finite automata. *Zeit. Math. Logik und Grundl. Math.*, 6:66–92, 1960.
16. J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congress on Logic, Method, and Philosophy of Science. 1960*, pages 1–12. Stanford University Press, 1962.
17. J.R. Büchi, C.C. Elgot, and J.B. Wright. The non-existence of certain algorithms for finite automata theory (abstract). *Notices Amer. Math. Soc.*, 5:98, 1958.
18. J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.
19. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. on Logic in Computer Science*, pages 428–439, 1990.
20. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
21. R.M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing 74*, pages 308–312, Stockholm, Sweden, 1974. International Federation for Information Processing, North-Holland.
22. D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Conf. on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2005.
23. S.M. Nowick C.H. van Berkel, M.B. Josephs. Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, 1999.
24. A. Church. Applicaton of recursive arithmetics to the problem of circuit synthesis. In *Summaries of Talks Presented at The Summer Institute for Symbolic Logic*, pages 3–50. Communications Research Division, Institute for Defense Analysis, 1957.

25. A. Church. Logic, arithmetics, and automata. In *Proc. Int. Congress of Mathematicians, 1962*, pages 23–35. Institut Mittag-Leffler, 1963.
26. A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. 14th Int'l Conf. on Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 359–364. Springer, 2002.
27. E.M. Clarke. The birth of model checking. *This Volume*, 2007.
28. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
29. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 117–126, 1983.
30. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languagues and Systems*, 8(2):244–263, 1986.
31. E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model-checking algorithms. In *Proc. 16th ACM Symp. on Principles of Distributed Computing*, pages 294–303, 1987.
32. E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In *Proc 6th Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science, pages 415 – 427. Springer, 1994.
33. E.M. Clarke and B. Mishra. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269–291, 1985.
34. R.L. Constable. On the theory of programming logics. In *Proc. 9th ACM Symp. on Theory of Computing*, pages 269–285, 1977.
35. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 436–453. Springer, 2001.
36. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proc 2nd Int. Conf. on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer, 1990.
37. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
38. A. Duret-Lutz and Denis Poitrenaud. SPOT: An extensible model checking library using transition-based generalized büchi automata. In *Proc. 12th Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 76–83. IEEE Computer Society, 2004.
39. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
40. C. Eisner, D. Fisman, and J. Havlicek. A topological characterization of weakness. In *Proc. 24th ACM Symp. on Principles of Distributed Computing*, pages 1–8, 2005.
41. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. 15th Int'l Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2003.
42. C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout. The definition of a temporal clock operator. In *Proc. 30th Int'l Colloquium on Automata, Languages*

and Programming, volume 2719 of *Lecture Notes in Computer Science*, pages 857–870. Springer, 2003.
43. C. Elgot. Decision problems of finite-automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–51, 1961.
44. C.C. Elgot and J. Wright. Quantifier elimination in a problem of logical design. *Michigan Math. J.*, 6:65–69, 1959.
45. E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th Int. Colloq. on Automata, Languages, and Programming*, pages 169–181, 1980.
46. E.A. Emerson and J.Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 127–140, 1983.
47. E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and Systems Science*, 30:1–24, 1985.
48. E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.
49. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 267–278, 1986.
50. E. Engeler. Algorithmic properties of structures. *Math. Syst. Theory*, 1:183–195, 1967.
51. K. Etessami, M.Y. Vardi, and T. Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002.
52. M.J. Fischer and R.E. Ladner. Propositional modal logic of programs (extended abstract). In *Proc. 9th ACM Symp. on Theory of Computing*, pages 286–294, 1977.
53. M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Science*, 18:194–211, 1979.
54. L. Fix. Fifteen years of formal property verification at intel. *This Volume*, 2007.
55. L. Fix and G. Kamhi. Adaptive variable reordering for symbolic model checking. In *Proc. ACM/IEEE Int'l Conf. on Computer Aided Design*, pages 359–365, 1998.
56. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, pages 163–173, 1980.
57. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.
58. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.
59. T. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Proc. 14th Int. Colloq. on Automata, Languages, and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 269–279. Springer, 1987.
60. JY. Halpern and J.H. Reif. The propositional dynamic logic of deterministic, well-structured programs (extended abstract). In *Proc. 22nd IEEE Symp. on Foundations of Computer Science*, pages 322–334, 1981.
61. J.Y. Halpern and J.H. Reif. The propositional dynamic logic of deterministic, well-structured programs. *Theor. Comput. Sci.*, 27:127–165, 1983.
62. R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *Proc 8th Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 1996.
63. D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness. *J. Comput. Syst. Sci.*, 25(2):144–170, 1982.

64. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
65. D. Harel and D. Peleg. More on looping vs. repeating in dynamic logic. *Inf. Process. Lett.*, 20(2):87–90, 1985.
66. D. Harel and D. Peleg. Process logic with regular formulas. *Theoreti. Comp. Sci.*, 38(2–3):307–322, 1985.
67. D. Harel and R. Sherman. Looping vs. repeating in dynamic logic. *Inf. Comput.*, 55(1–3):175–192, 1982.
68. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
69. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
70. J.A.W. Kamp. *Tense Logic and the Theory of Order*. PhD thesis, UCLA, 1968.
71. R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, 1976.
72. D. Kozen. Results on the propositional μ-calculus. In *Proc. 9th Colloquium on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 1982.
73. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
74. S. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24:1–14, 1959.
75. O. Kupferman, N. Piterman, and M.Y. Vardi. Safraless compositional synthesis. In *Proc 18th Int. Conf. on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006.
76. O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic*, 2(2):408–429, 2001.
77. O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Proc. 46th IEEE Symp. on Foundations of Computer Science*, pages 531–540, 2005.
78. R.P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, and Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453. Springer, 1990.
79. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.
80. R.P. Kurshan. Verification technology transfer. *This Volume*, 2007.
81. L. Lamport. "Sometimes" is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, pages 174–185, 1980.
82. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, 1985.
83. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.
84. N. Markey. Temporal logic with past is exponentially more succinct. *EATCS Bulletin*, 79:122–128, 2003.
85. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
86. R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Pres, 1971.
87. A. R. Meyer. Weak monadic second order theory of successor is not elementary recursive. In *Proc. Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer, 1975.
88. A.R. Meyer. Ten thousand and one logics of programming". Technical report, MIT, 1980. MIT-LCS-TM-150.

89. M.J. Morley. Semantics of temporal ***e***. In T. F. Melham and F.G. Moller, editors, Banff'99 *Higher Order Workshop (Formal Methods in Computation)*. University of Glasgow, Department of Computing Science Technical Report, 1999.
90. A. Urquhart N. Rescher. *Temporal Logic*. Springer, 1971.
91. P. Øhrstrøm and P.F.V. Hasle. *Temporal Logic: from Ancient Times to Artificial Intelligence*. Studies in Linguistics and Philosophy, vol. 57. Kluwer, 1995.
92. D. Park. Finiteness is μ-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.
93. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.
94. A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Int. Colloq. on Automata, Languages, and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 1985.
95. A. Pnueli and L. Zuck. In and out of temporal logic. In *Proc. 8th IEEE Symp. on Logic in Computer Science*, pages 124–135, 1993.
96. V.R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th IEEE Symp. on Foundations of Computer Science*, pages 109–121, 1976.
97. V.R. Pratt. A practical decision method for propositional dynamic logic: Preliminary report. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 326–337, 1978.
98. V.R. Pratt. A near-optimal method for reasoning about action. *Journal of Computer and Systems Science*, 20(2):231–254, 1980.
99. V.R. Pratt. A decidable μ-calculus: preliminary report. In *Proc. 22nd IEEE Symp. on Foundations of Computer Science*, pages 421–427, 1981.
100. A. Prior. *Time and Modality*. Oxford University Press, 1957.
101. A. Prior. *Past, Present, and Future*. Clarendon Press, 1967.
102. A.N. Prior. Modality de dicto and modality de re. *Theoria*, 18:174–180, 1952.
103. A.N. Prior. Modality and quantification in s5. *J. Symbolic Logic*, 21:60–62, 1956.
104. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
105. M.O. Rabin. Automata on infinite objects and Church's problem. *Amer. Mathematical Society*, 1972.
106. K. Sabnani, P. Wolper, and A. Lapone. An algorithmic technique for protocol verification. In *Proc. Globecom '85*, 1985.
107. A. Salwicki. Algorithmic logic: a tool for investigations of programs. In R.E. Butts and J. Hintikka, editors, *Logic Foundations of Mathematics and Computability Theory*, pages 281–295. Reidel, 1977.
108. A.P. Sistla. *Theoretical issues in the design of distributed and concurrent systems*. PhD thesis, Harvard University, 1983.
109. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. In *Proc. 14th Annual ACM Symposium on Theory of Computing*, pages 159–168, 1982.
110. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.
111. A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. In *Proc. 12th Int. Colloq. on Automata, Languages, and Programming*, volume 194, pages 465–474. Springer, 1985.
112. A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.
113. L.J. Stockmeyer. *The complexity of decision procedures in Automata Theory and Logic*. PhD thesis, MIT, 1974. Project MAC Technical Report TR-133.
114. R.S. Street. Propositional dynamic logic of looping and converse. In *Proc. 13th ACM Symp. on Theory of Computing*, pages 375–383, 1981.

115. R.S. Streett. *A propositional dynamic logic for reasoning about program divergence*. PhD thesis, M.Sc. Thesis, MIT, 1980.
116. R.S. Streett. Propositional dynamic logic of looping and converse. *Information and Control*, 54:121–141, 1982.
117. A comparison of reset control methods: Application note 11. `http://www.summitmicro.com/tech_support/notes/note11.htm`, Summit Microelectronics, Inc., 1999.
118. W. Thomas. Star-free regular sets of ω-sequences. *Information and Control*, 42(2):148–156, 1979.
119. W. Thomas. On the synthesis of strategies in infinite games. In E.W. Mayr and C. Puech, editors, *Proc. 12th Symp. on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.
120. B. Trakhtenbrot. The synthesis of logical nets whose operators are described in terms of one-place predicate calculus. *Doklady Akad. Nauk SSSR*, 118(4):646–649, 1958.
121. B.A. Trakhtenbrot and Y.M. Barzdin. *Finite Automata*. North Holland, 1973.
122. M.Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 137–146, 1982.
123. M.Y. Vardi. A temporal fixpoint calculus. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 250–259, 1988.
124. M.Y. Vardi. Unified verification theory. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Temporal Logic in Specification*, volume 398, pages 202–212. Springer, 1989.
125. M.Y. Vardi. Nontraditional applications of automata theory. In *Proc. 11th Symp. on Theoretical Aspects of Computer Science*, volume 789 of *Lecture Notes in Computer Science*, pages 575–597. Springer, 1994.
126. M.Y. Vardi. Branching vs. linear time: Final showdown. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001.
127. M.Y. Vardi. Automata-theoretic model checking revisited. In *Proc. 7th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2007.
128. M.Y. Vardi and P. Wolper. Yet another process logic. In *Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 501–512. Springer, 1984.
129. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 332–344, 1986.
130. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
131. S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.
132. P. Wolper. Temporal logic can be more expressive. In *Proc. 22nd IEEE Symp. on Foundations of Computer Science*, pages 340–348, 1981.
133. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.
134. P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 185–194, 1983.