Branching vs. Linear Time: Final Showdown*

Moshe Y. Vardi**

Rice University, Department of Computer Science, Houston, TX 77005-1892, USA

December 14, 2002

Abstract. The discussion of the relative merits of linear- versus branching-time frameworks goes back to early 1980s. One of the beliefs dominating this discussion has been that "while specifying is easier in LTL (linear-temporal logic), verification is easier for CTL (branching-temporal logic)". Indeed, the restricted syntax of CTL limits its expressive power and many important behaviors (e.g., strong fairness) can not be specified in CTL. On the other hand, while model checking for CTL can be done in time that is linear in the size of the specification, it takes time that is exponential in the specification for LTL. Because of these arguments, and for historical reasons, the dominant temporal specification language in industrial use is CTL.
In this paper we argue that in spite of the phenomenal success of CTL-based model checking, CTL suffers from several fundamental limitations as a specification language, all stemming from the fact that CTL is a branching-time formalism: the language is unintuitive and hard to use, it does not lend itself to compositional reasoning, and it is fundamentally incompatible with semi-formal verification. These inherent limitations severely impede the functionality of CTL-based model checkers. In contrast, the linear-time framework is expressive and intuitive, supports compositional reasoning and semi-formal verification, and is amenable to combining enumerative and symbolic search methods. While we argue in favor of the linear-time framework, we also we argue that LTL is not expressive enough, and discuss what would be the "ultimate" temporal specification language.

1 Introduction

As indicated in the National Technology Roadmap for Semiconductors[1], the semiconductor industry faces a serious challenge: chip designers are finding it increasingly difficult to keep up with the advances in semiconductor manufacturing. As a result, they are unable to exploit the enormous capacity that this technology provides. The Roadmap suggests that the semiconductor industry will require productivity gains greater than the historical 30% per-year cost reduction. This is referred to as the "design productivity crisis".

Integrated circuits are currently designed through a series of steps that refine a more abstract specification into a more concrete implementation. The process starts at a "behavioral model", such as a program that implements the instruction set architecture of a processor. It ends in a description of the actual geometries of the transistors and wires on the chip. Each refinement step used to synthesize the processor must preserve the germane behavior of the abstract model. As designs grow more complex, it becomes easier to introduce flaws into the design during refinement. Thus, designers use various validation techniques to prove the correctness of the design after each refinement. Unfortunately, these techniques themselves grow more expensive and difficult with design complexity. Indeed, for many designs,

* The original version of this paper appeared in the proceedings of TACAS'01, April 2001. This document continues to evolve in response to comments and discussions. Changes are marked with changebars.

** Supported in part by NSF grants CCR-9700061 and CCR-9988322, and by a grant from the Intel Corporation. URL: `http://www.cs.rice.edu/~vardi`.

[1] `http://public.itrs.net/files/1999_SIA_Roadmap/Home.htm`

the size of the validation team now exceeds that of the design team. As the validation process has begun to exceed half the design project resources, the semiconductor industry has begun to refer to this problem as the "validation crisis".

Formal verification provides a new approach to validating the correct behavior of logic designs. In simulation, the traditional mode of design validation, "confidence" is the result of running a large number of test cases through the design. Formal verification, in contrast, uses mathematical techniques to check the entire state space of the design for conformance to some specified behavior. Thus, while simulation is open-ended and fraught with uncertainty, formal verification is definitive and eliminates uncertainty [27].

One of the most significant recent developments in the area of formal design verification is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [23, 78, 97, 116]. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired property by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this property (see [26]). *Symbolic model checking* [17] has been used to successfully verify a large number of complex designs. This approach uses symbolic data structures, such as *binary decision diagrams* (BDDs), to efficiently represent and manipulate state spaces. Using symbolic model checking, designs containing on the order of 100 to 200 binary latches can be routinely verified automatically.

Model-checking tools have enjoyed a substantial and growing use over the last few years, showing ability to discover subtle flaws that result from extremely improbable events. While until recently these tools were viewed as of academic interest only, they are now routinely used in industrial applications [8, 48]. Companies such as AT&T, Cadence, Fujitsu, HP, IBM, Intel, Motorola, NEC, SGI, Siemens, and Sun are using model checkers increasingly on their own designs to ensure outstanding product quality. Three model-checking tools are widely used in the semiconductor industry: SMV, a tool from Carnegie Mellon University [84], with many industrial incarnations (e.g., IBM's RuleBase [9]); VIS, a tool developed at the University of California, Berkeley [14]; and FormalCheck, a tool developed at Bell Labs [51] and marketed by Cadence.

A key issue in the design of a model-checking tool is the choice of the temporal language used to specify properties, as this language, which we refer to as the *temporal property-specification language*, is one of the primary interfaces to the tool. (The other primary interface is the modeling language, which is typically the hardware description language used by the designers). One of the major aspects of all temporal languages is their underlying model of time. Two possible views regarding the nature of time induce two types of temporal logics [75]. (For a description of earlier discussions of this distinction in the philosophical-logic literature, see [19].) In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and we regard them as describing a behavior of a single computation of a program. In *branching* temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of a nondeterministic program.

In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives as well as the temporal connective G ("always"), F ("eventually"), X ("next"), and U ("until"). The branching temporal logic CTL* augments LTL by the path quantifiers E ("there exists a computation") and A ("for all computations"). The branching temporal logic CTL is a fragment of CTL* in which every temporal connective is preceded by a path quantifier. Finally, the branching temporal logic $\forall$CTL is a fragment of CTL in which only universal path quantification is allowed. (Note that LTL has implicit universal path quantifiers in front of its formulas.)

The discussion of the relative merits of linear versus branching temporal logics in the context of specification and verification goes back to 1980 [91, 75, 36, 10, 93, 40, 38, 22, 113, 114]. As analyzed in [93], linear and branching time logics correspond to two distinct views of time. It is not surprising therefore that LTL and CTL are expressively incomparable [75, 38, 22]. The LTL formula FGp is not expressible in CTL, while the CTL formula $AFAGp$ is not expressible in LTL. On the other hand, CTL seems to be superior to LTL when it comes to algorithmic verification, as we now explain.

Given a transition system M and a linear temporal logic formula φ, the model-checking problem for M and φ is to decide whether φ holds in all the computations of M. When φ is a branching temporal logic formula, the problem is to decide whether φ holds in the computation tree of M. The complexity of model checking for both linear and branching temporal logics is well understood: suppose we are given a transition system of size n and a temporal logic formula of size m. For the branching temporal logic CTL, model-checking algorithms run in time $O(nm)$ [23], while, for the linear temporal logic LTL, model-checking algorithms run in time $n2^{O(m)}$ [78]. Since LTL model checking is PSPACE-complete [103], the latter bound probably cannot be improved.

The difference in the complexity of linear and branching model checking has been viewed as an argument in favor of the branching paradigm. In particular, the computational advantage of CTL model checking over LTL model checking makes CTL a popular choice, leading to efficient model-checking tools for this logic [24]. Through the 1990s, the dominant temporal specification language in industrial use has been CTL. This dominance stemmed from the phenomenal success of SMV, the first symbolic model checker, which is CTL-based, and its follower VIS, also CTL-based, which serve as the basis for many industrial model checkers. (Verification systems that use linear-time formalisms are the above mentioned FormalCheck, Bell Labs's SPIN [55], Intel's Prover, and Cadence SMV.)

In spite of the phenomenal success of CTL-based model checking, CTL suffers from several fundamental limitations as a temporal property-specification language, all stemming from the fact that CTL is a branching-time formalism: the language is unintuitive and hard to use, it does not lend itself to compositional reasoning, and it is fundamentally incompatible with semi-formal verification. In contrast, the linear-time framework is expressive and intuitive, supports compositional reasoning and semi-formal verification, and is amenable to combining enumerative and symbolic search methods. While we argue in favor of the linear-time framework, we also we argue that LTL is not expressive enough, and discuss what would be the ultimate temporal specification language. It is not our intention in this paper to argue which approach, the linear or the branching is the "correct" one. We agree with van Glabbeek who say in [47] that "it is not the task of process theory to find the 'true' semantics of processes, but rather to determine which process semantics is suitable for which applications". Our position here is that the linear-time framework is more suitable for model checking.

We assume familiarity with the syntax and semantics of temporal logic [35, 61, 106].

2 CTL

2.1 Expressiveness

It is important to understand that expressiveness is not merely a theoretical issue; expressiveness is also a usability issue. Verification engineers find CTL unintuitive. The linear framework is simply more natural for verification engineers, who tend to think linearly, e.g., timing diagrams [42] and message-sequence charts [74], rather than "branchingly". IBM's experience with the RuleBase system has been that "nontrivial CTL equations are hard to understand and prone to error" [102] and "CTL is difficult to use for most users and requires a new way of thinking about hardware" [9]. Indeed, IBM has been trying to "linearize" CTL in their RuleBase system [9]. It is simply much harder to reason about computation trees than about linear computations.

As an example, consider the LTL formulas XFp and FXp. Both formulas say the same thing: "p holds sometimes in the strict future". In contrast, consider the CTL formulas $AFAXp$ and $AXAFp$. Are these formulas logically equivalent? Do they assert that

"p holds sometimes in the strict future"? It takes a few minutes of serious pondering to realize that while $AXAFp$ does assert that "p holds sometimes in the strict future", this is not the case for $AFAXp$ (we challenge the reader to figure out the precise meaning of $AFAXp$). The unintuitiveness of CTL significantly reduces the usability of CTL-based formal-verification tools. A perusal of the literature reveals that the vast majority of CTL formulas used in formal verification are actually equivalent to LTL formulas. Thus, the branching nature of CTL is very rarely used in practice. As a consequence, even though LTL and CTL are expressively incomparable from a theoretical point of view, from a practical point of view LTL is more expressive than CTL.

One often hears the claim that expressiveness "is not an issue", since "all users want to verify are simple invariance property of the form AGp". Of course, the reason for that could be the difficulty of expressing in CTL more complicated properties. Industrial experience with linear-time formalism shows that verification engineers often use much more complicated temporal properties, when provided with a language that facilitates the expression of such properties. Further more, even when attempting to verify an invariance property, users often need to express relevant properties, which can be rather complex, of the environment of the unit under verification. We come back to this point later.

Reader who is steeped in the concurrency-theory literature may be somewhat surprised at the assertion that that CTL lacks expressive power. After all, it is known that CTL characterizes *bisimulation*, in the sense that two states in a transition system are bisimilar iff they satisfy exactly the same CTL formulas [15] (see also [53]), and bisimulation is considered to be the finest reasonable notion of equivalence between processes [90, 86]. This result, however, says little about the usefulness of CTL as a property-specification language. For example, it is known that multi-modal logic (known also as "Hennessy-Milner Logic") also characterizes bisimulation, but no one seriously advocate this very weak logic as a property-specification language for reactive systems. Bisimulation is a structural relation, while in the context of model checking what is needed is a way to specify behavioral properties rather than structural properties. Assertions about behavior are best stated in terms of traces rather than in terms of computation trees (recall, for example, the subtle distinction between $AFAXp$ and $AXAFp$).[2] The reader should recall the above quote from [47] that "it is not the task of process theory to find the 'true' semantics of processes, but rather to determine which process semantics is suitable for which applications". As shown in [82] (cf. Proposition 4), linear-time semantics is complete with respect to linear-time properties in the the sense that in order to prove linear-time properties of parallel composition of reactive systems it suffices to consider the linear-time properties of the individual components. Our assertion is that branching-time semantics is simply too fine in the context of model checking.

2.2 Complexity

As we saw earlier, the complexity bounds for CTL model checking are better than those for LTL model checking. We first show that this superiority disappears in the context of open systems.

In computer system design, we distinguish between *closed* and *open* systems. A closed system is a system whose behavior is completely determined by the state of the system. An open system is a system that interacts with its environment and whose behavior depends on this interaction. Such systems are called also *reactive systems* [52]. In closed systems,

[2] It is also worth noting that when modeling systems in terms of *state* transition systems, as is the standard practice in the model-checking literature (cf. [26]), deadlocks have to be modeled explicitly, since in state transition systems we model states and their properties rather than actions. Once deadlocks are modeled explicitly, the two process $a(b+c)$ and $ab+ac$, which are typically considered to be trace equivalent but not bisimilar [86], become trace inequivalent. In fact, many of the different semantics described in [47] coincide once deadlocks are modeled explicitly.

nondeterminism reflect an *internal* choice, while in open systems it can also reflect an *external* choice [54]. Formally, in a closed system, the environment can not modify any of the system variables. In contrast, in an open system, the environment can modify some of the system variables. In reality, the vast majority of interesting systems are open, since they have to interact with an external environment.

We can model finite-state open systems by *open modules*. An open module is simply a module with a partition of the states into two sets. One set contains *system states* and corresponds to locations where the system makes a transition. The second set contains *environment states* and corresponds to locations where the environment makes a transition.

As discussed in [82], when the specification is given in linear temporal logic, there is indeed no need to worry about uncertainty with respect to the environment. Since all the possible interactions of the system with its environment have to satisfy a linear temporal logic specification in order for a program to satisfy the specification, the distinction between internal and external nondeterminism is irrelevant. In contrast, when the specification is given in a branching temporal logic, this distinction is relevant. There is a need to define a different model-checking problem for open systems, and there is a need to adjust current model-checking tools to handle open systems correctly.

We now specify formally the problem of *model checking of open modules* (*module checking*, for short). As with usual model checking, the problem has two inputs: an open module M and a temporal logic formula φ. For an open module M, let V_M denote the unwinding of M into an infinite tree. We say that M satisfies φ iff φ holds in all the trees obtained by pruning from V_M subtrees whose root is a successor of an environment state. The intuition is that each such tree corresponds to a different (and possible) environment. We want φ to hold in every such tree, since, of course, we want the open system to satisfy its specification no matter how the environment behaves.

A *module* $M = \langle W, W_0, R, V\rangle$ consists of a set W of states, a set W_0 of initial states, a total transition relation $R \subseteq W \times W$, and a labeling function $V : W \to 2^{Prop}$ that maps each state to a set of atomic propositions that hold in this state. We model an open system by an *open module* $M = \langle W_s, W_e, W_0, R, V\rangle$, where $\langle W_s \cup W_e, W_0, R, V\rangle$ is a module, W_s is a set of *system states*, and W_e is a set of *environment states*. We use W to denote $W_s \cup W_e$. For each state $w \in W$, let $succ(w)$ be the set of w's R-successors; i.e., $succ(w) = \{w' : R(w, w')\}$. Consider a system state w_s and an environment state w_e. When the current state is w_s, all the states in $succ(w_s)$ are possible next states. In contrast, when the current state is w_e, there is no certainty with respect to the environment transitions and not all the states in $succ(w_e)$ are necessarily possible next states. The only thing guaranteed is that not all the environment transitions are impossible, since the environment can never be blocked. For a state $w \in W$, let $step(w)$ denote the set of the possible sets of w's next successors during an execution. By the above, $step(w_s) = \{succ(w_s)\}$ and $step(w_e)$ contains all the nonempty subsets of $succ(w_e)$.

An *infinite tree* is a set $T \subseteq X^*$ such that if $x \cdot c \in T$ where $x \in X^*$ and $c \in X$, then also $x \in T$, and for all $0 \leq c' < c$, we have that $x \cdot c' \in T$. In addition, if $x \in T$, then $x \cdot 0 \in T$. The elements of T are called *nodes*, and the empty word ϵ is the *root* of T. Given an alphabet Σ, a *Σ-labeled tree* is a pair $\langle T, V\rangle$ where T is a tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ. An open module M can be unwound into an infinite tree $\langle T_M, V_M\rangle$ in a straightforward way. When we examine a specification with respect to M, it should hold not only in $\langle T_M, V_M\rangle$ (which corresponds to a very specific environment that does never restrict the set of its next states), but in all the trees obtained by pruning from $\langle T_M, V_M\rangle$ subtrees whose root is a successor of a node corresponding to an environment state. Let $exec(M)$ denote the set of all these trees. Formally, $\langle T, V\rangle \in exec(M)$ iff the following holds:

- $\epsilon \in T$ and $V(\epsilon) = w_0$.
- For all $x \in T$ with $V(x) = w$, there exists $\{w_0, \ldots, w_n\} \in step(w)$ such that $T \cap X^{|x|+1} = \{x \cdot 0, x \cdot 1, \ldots, x \cdot n\}$ and for all $0 \leq c \leq n$ we have $V(x \cdot c) = w_c$.

Intuitively, each tree in $exec(M)$ corresponds to a different behavior of the environment. Note that a single environment state with more than one successor suffices to make $exec(M)$ infinite.

Given an open module M and a CTL* formula φ, we say that M satisfies φ, denoted $M \models_o \varphi$, if all the trees in $exec(M)$ satisfy φ. The problem of deciding whether M satisfies φ is called *module checking*. We use $M \models \varphi$ to indicate that when we regard M as a closed module (thus refer to all its states as system states), then M satisfies φ. The problem of deciding whether $M \models \varphi$ is the usual model-checking problem. Note that while $M \models_o \varphi$ entails $M \models \varphi$, all that $M \models \varphi$ entails is that $M \not\models_o \neg\varphi$. Indeed, $M \models_o \varphi$ requires all the trees in $exec(M)$ to satisfy φ. On the other hand, $M \models \varphi$ means that the tree $\langle T_M, V_M \rangle$ satisfies φ. Finally, $M \not\models_o \neg\varphi$ only tells us that there exists some tree in $exec(M)$ that satisfies φ. We can define module checking also with respect to linear-time specifications. We say that an open module M satisfies an LTL formula φ iff $M \models_o A\varphi$.

Theorem 1. [62, 71]

(2) *The module-checking problem for LTL is PSPACE-complete.*
(2) *The module-checking problem for CTL is EXPTIME-complete.*
(3) *The module-checking problem for CTL* is 2EXPTIME-complete.*

Thus, module checking for LTL is easier than for CTL (assuming that EXPTIME is different than PSPACE), which is, in turn, easier than for CTL*. In particular, this results shows that branching is not "free", as has been claimed in [40].[3] See [63, 68] for further discussion of open-system verification and [113] for further discussion on the complexity-theoretic comparison between linear time and branching time.

Even in the context of closed systems, the alleged superiority of CTL from the complexity perspective is questionable. The traditional comparison is in terms of worst-case complexity. Since, however, CTL and LTL are expressively incomparable, a comparison in terms of worst-case complexity is not very meaningful. A more meaningful comparison would be with respect to properties that can be expressed in both CTL and LTL. We claim that under such a comparison the superiority of CTL disappears.

For simplicity, we consider systems M with no fairness conditions; i.e., systems in which all the computations are fair. As the "representative" CTL model checker we take the bottom-up labeling procedure of [23]. There, in order to check whether M satisfies φ, we label the states of M by subformulas of φ, starting from the innermost formulas and proceeding such that, when labeling a formula, all its subformulas are already labeled. Labeling subformulas that are atomic propositions, Boolean combinations of other subformulas, or of the form $AX\theta$ or $EX\theta$ is straightforward. Labeling subformulas of the form $A\theta_1 U\theta_2$, $E\theta_1 U\theta_2$, $A\theta_1 \tilde{U}\theta_2$, or $E\theta_1 \tilde{U}\theta_2$ involves a backward reachability test. As the "representative" LTL model checker, we take the automata-based algorithm of [116]. There, in order to check whether M satisfies φ, we construct a Büchi word automaton $\mathcal{A}_{\neg\varphi}$ for $\neg\varphi$ and check whether the intersection of the language of M with that of $\mathcal{A}_{\neg\varphi}$ is nonempty. In practice, the latter check proceeds by checking whether there exists an initial state in the intersection that satisfies CTL formula $EG\mathbf{true}$. For the construction of $\mathcal{A}_{\neg\varphi}$, we follow the algorithms in [46] or [33], which improve [116] by being demand-driven; that is, the state space of $\mathcal{A}_{\neg\varphi}$ is restricted to states that are reachable from the initial state.

The exponential term in the running time of LTL model checking comes from a potential exponential blow-up in the translation of φ into an automaton $\mathcal{A}_{\neg\varphi}$. It is shown, however, in [80] that for LTL formulas that can also be expressed in $\forall$CTL (the universal fragment of CTL) there is Büchi automaton $\mathcal{A}_{\neg\varphi}$ whose size is *linear* in $|\varphi|$. Furthermore, this automaton has a special structure (it is "weak"), which enables the model checker to apply improved algorithms for checking the emptiness of the intersection of M with $\mathcal{A}_{\neg\varphi}$

[3] Note also that while the satisfiability problem for LTL is PSPACE-complete [103], the problem is EXPTIME-complete for CTL [41, 37] and 2EXPTIME-complete for CTL* [115, 39].

[12]. (See also [65, 66] for a through analysis of the relationship between LTL and CTL model checkers.)

Another context in which the alleged superiority of CTL from the complexity perspective disappears is that of hierarchical systems. In that setting, both LTL model checking and CTL model checking are PSPACE-complete, but while LTL model checking is polynomial in the size of the system, CTL model checking is exponential in the size of the system [3]. Similarly, in the context of pushdown systems, for both LTL and CTL model checking is EXPTIME-complete, but it is polynomial in the size of the system for LTL [13] and exponential in the size of the system for CTL [118].

2.3 Compositionality

Model checking is known to suffer from the so-called *state-explosion* problem. In a concurrent setting, the system under consideration is typically the parallel composition of many modules. As a result, the size of the state space of the system is the product of the sizes of the state spaces of the participating modules. This gives rise to state spaces of exceedingly large sizes, which makes model-checking algorithms impractical. This issue is one of the most important ones in the area of computer-aided verification and is the subject of active research (cf. [18]).

Compositional, or *modular*, verification is one possible way to address the state-explosion problem, cf. [28]. In modular verification, one uses proof rules of the following form:

$$\left.\begin{array}{l} M_1 \models \psi_1 \\ M_2 \models \psi_2 \\ C(\psi_1, \psi_2, \psi) \end{array}\right\} M_1 \| M_2 \models \psi$$

Here $M \models \theta$ means that the module M satisfies the formula θ, the symbol "$\|$" denotes parallel composition, and $C(\psi_1, \psi_2, \psi)$ is some logical condition relating ψ_1, ψ_2, and ψ. The advantage of using modular proof rules is that it enables one to apply model checking only to the underlying modules, which have much smaller state spaces.

A key observation, see [87, 76, 57, 105, 92], is that in modular verification the specification should include two parts. One part describes the desired behavior of the module. The other part describes the assumed behavior of the system within which the module is interacting. This is called the *assume-guarantee* paradigm, as the specification describes what behavior the module is *guaranteed* to exhibit, *assuming* that the system behaves in the promised way.

For the linear temporal paradigm, an assume-guarantee specification is a pair $\langle \varphi, \psi \rangle$, where both φ and ψ are linear temporal logic formulas. The meaning of such a pair is that all the computations of the module are guaranteed to satisfy ψ, assuming that all the computations of the environment satisfy φ. As observed in [92], in this case the assume-guarantee pair $\langle \varphi, \psi \rangle$ can be combined to a single linear temporal logic formula $\varphi \to \psi$. Thus, model checking a module with respect to assume-guarantee specifications in which both the assumed and the guaranteed behaviors are linear temporal logic formulas is essentially the same as model checking the module with respect to linear temporal logic formulas.

The situation is different for the branching temporal paradigm, where assumptions are taken to apply to the computation tree of the system within which the module is interacting [49]. In this framework, a module M satisfies an assume-guarantee pair $\langle \varphi, \psi \rangle$ iff whenever M is part of a system satisfying φ, the system also satisfies ψ. (As is shown in [49], this is not equivalent to M satisfying $\varphi \to \psi$.) We call this *branching modular model checking*. Furthermore, it is argued in [49], as well as in [30, 58, 49, 31], that in the context of modular verification it is advantageous to use only *universal* branching temporal logic, i.e., branching temporal logic without existential path quantifiers. In a universal branching temporal logic one can state properties of all computations of a program, but one cannot state that certain computations exist. Consequently, universal branching temporal logic formulas have the helpful property that once they are satisfied in a module, they are satisfied also in every system that contains this module. The focus in [49] is on using ∀CTL, the universal fragment of CTL, for both the assumption and the guarantee. We now focus on the branching modular model-checking problem, where assumptions and guarantees are in both ∀CTL and in the more expressive ∀CTL*, the universal fragment of CTL*.

Let $M = (W, W_0, R, V)$ and $M' = (W', W'_0, R', V')$ be two modules with sets AP and AP' of atomic propositions. The *composition* of M and M', denoted $M \| M'$, is a module that has exactly these behaviors which are joint to M and M'. We define $M \| M'$ to be the module $\langle W'', W_{0''}, R, V'' \rangle$ over the set $AP'' = AP \cup AP'$ of atomic propositions, where $W'' = (W \times W') \cap \{\langle w, w' \rangle : V(w) \cap AP' = V'(w') \cap AP\}$, $W_{0''} = (W_0 \times W_{0'}) \cap W''$, $R'' = \{\langle \langle w, w' \rangle, \langle s, s' \rangle \rangle : \langle w, s \rangle \in R$ and $\langle w', s' \rangle \in R'\}$, and $V''(\langle w, w' \rangle) = V(w) \cup V'(w')$ for $\langle w, w' \rangle \in W''$.

In modular verification, one uses assertions of the form $\langle \varphi \rangle M \langle \psi \rangle$ to specify that whenever M is part of a system satisfying the universal branching temporal logic formula φ, the system satisfies the universal branching temporal logic formula ψ too. Formally, $\langle \varphi \rangle M \langle \psi \rangle$ holds if $M \| M' \models \psi$ for all M' such that $M \| M' \models \varphi$. Here φ is an assumption on the behavior of the system and ψ is the guarantee on the behavior of the module. Assume-guarantee assertions are used in modular proof rules of the following form:

$$\left.\begin{array}{l} \langle \varphi_1 \rangle M_1 \langle \psi_1 \rangle \\ \langle \mathbf{true} \rangle M_1 \langle \varphi_1 \rangle \\ \langle \varphi_2 \rangle M_2 \langle \psi_2 \rangle \\ \langle \mathbf{true} \rangle M_2 \langle \varphi_2 \rangle \end{array}\right\} \langle \mathbf{true} \rangle M_1 \| M_2 \langle \psi_1 \wedge \psi_2 \rangle$$

Thus, a key step in modular verification is checking that assume-guarantee assertions of the form $\langle \varphi \rangle M \langle \psi \rangle$ hold, which we called the *branching modular model-checking problem*.

Theorem 2. [70]

(1) *The branching modular model-checking problem for ∀CTL is PSPACE-complete.*
(2) *The branching modular model-checking problem for ∀CTL* is EXPSPACE-complete.*

Thus, in the context of modular model checking, ∀CTL has the same computational complexity as LTL, while ∀CTL* is exponentially harder. The fact that the complexity for ∀CTL is the same as the complexity for LTL is, however, somewhat misleading. ∀CTL is simply not expressive enough to express assumptions that are strong enough to prove the desired guarantee. This motivated Josko to consider modular verification with guarantees in CTL and assumptions in LTL. Unfortunately, it is shown in [70] that the EXPSPACE lower bound above applies even for that setting.

Another approach to modular verification for ∀CTL is proposed in [49], where the following inference rule is proposed:

$$\left.\begin{array}{l} M_1 \preceq A_1 \\ A_1 || M_2 \preceq A_2 \\ M_1 || A_2 \models \varphi \end{array}\right\} M_1 || M_2 \models \varphi$$

Here A_1 and A_2 are modules that serve as assumptions, and $\preceq$ is the *simulation* refinement relation [85]. In other words, if M_1 guarantees the assumption A_1, M_2 under the assumption A_1 guarantees the assumption A_2, and M_1 under the assumption A_2 guarantees φ, then we know that $M_1 || M_2$, under no assumption, guarantees φ. The advantage of this rule is that both the $\preceq$ and $\models$ relation can be evaluated in polynomial time. Unfortunately, the simulation relation is much finer than the trace-containment relation (which is the refinement relation in the linear-time framework). This makes it exceedingly difficult to come up with the assumptions A_1 and A_2 above.

What do CTL users do in practice? In practice, they use the following rule:

$$\left.\begin{array}{l} M_2 \preceq A_2 \\ M_1 || A_2 \models \varphi \end{array}\right\} M_1 || M_2 \models \varphi$$

That is, instead of checking that $M_1||M_2 \models \varphi$, one checks that $M_1||A_2 \models \varphi$, where A_2 is an abstraction of M_2. As CTL model checkers usually do not support the test $M_2 \preceq A_2$, users often rely on their "intuition", which is typically a "linear intuition" rather than "branching intuition".[4] In other words, a typical way users overcome the limited expressive power of CTL is by "escaping" outside the tool; they build the "stub" A_2 in a hardware description language. Unfortunately, since stubs themselves could be incorrect, this practice is unsafe. (Users often check that the abstraction A_2 satisfies some CTL properties, such as $AGEFp$, but this is not sufficient to establish that $M_2 \preceq A_2$.)

In summary, CTL is not adequate for modular verification, which explains why recent attempts to augment SMV with assume-guarantee reasoning are based on linear time reasoning [83].

2.4 Semi-Formal Verification

Because of the state-explosion problem, it is unrealistic to expect formal-verification tools to handle full systems or even large components. At the same time, simulation-based dynamic validation, while being able to handle large designs, covers only a small fraction of the design space, due to resource constraints. Thus, it has become clear that future verification tools need to combine formal and informal verification [120]. The combined approach is called *semi-formal verification* (cf. [44]). Such a combination, however, is rather problematic for CTL-based tools. CTL specifications and model-checking algorithms are in terms of computation trees; in fact, it is known that there are CTL formulas, e.g., $AFAXp$, whose failure cannot be witnessed by a linear counterexample [22].[5] In contrast, dynamic validation is fundamentally linear, as simulation generates individual computations. Thus, there is an inherent "impedance mismatch" between the two approaches. This explains why current approaches to semi-formal verification are limited to invariances, i.e., properties of the form AGp. While many design errors can be discovered by model checking invariances, modular verification of even simple invariances often requires rather complicated assumptions on the environment in which the component under verification operates. Current semi-formal approaches, however, cannot handle general assumptions. Thus, the restriction of semi-formal verification to invariances is quite severe, limiting the possibility of integrating CTL-based model checking in traditional validation environments.

3 Linear Time

Our conclusion from the previous section is that CTL-based model checking, while phenomenally successful over the last 20 years, suffers from some inherent limitations that severely impede its functionality. As we show now, the linear-time approach does not suffer from these limitations.

3.1 The Linear-Time Framework

LTL is interpreted over *computations*, which can be viewed as infinite sequences of truth assignments to the atomic propositions: i.e., a computation is a function $\pi : N \rightarrow 2^{Prop}$ that

[4] Note that linear-time refinement is defined in terms of trace containment, which is a behavioral relation, while branching-time refinement is defined in terms of simulation, which is a state-based relation. Thus, constructing an abstraction A_2 such that $M_2 \preceq A_2$ requires a very deep understanding of the environment M_2.

[5] One of the advertised advantages of model checking is that when the model checker returns a negative answer, that answer is accompanied by a counterexample [25]. Note, however, that validation engineers are usually interested in linear counterexamples, but there are CTL formulas whose failure cannot be witnessed by a linear counterexample. In general, CTL-based model checkers do always accompany a negative answer by a counterexample. A similar comment applies to positive witnesses [69].

assigns truth values to the elements of a set $Prop$ of atomic propositions at each time instant (natural number). For a computation π and a point $i \in N$, the notation $\pi, i \models \varphi$ indicates that a formula φ holds at the point i of the computation π. For example, $\pi, i \models X\varphi$ iff $\pi, i+1 \models \varphi$. We say that π *satisfies* a formula φ, denoted $\pi \models \varphi$, iff $\pi, 0 \models \varphi$.

Designs can be described in a variety of formal description formalisms. Regardless of the formalism used, a *finite-state design* can be abstractly viewed as a *labeled transition system*, i.e., as a module $M = (W, W_0, R, V)$, where W is the finite sets of states that the system can be in, $W_0 \subseteq W$ is the set of initial states of the system, $R \subseteq W^2$ is a total transition relation that indicates the allowable state transitions of the system, and $V : W \rightarrow 2^{Prop}$ assigns truth values to the atomic propositions in each state of the system. 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 $u_i \; R \; u_{i+1}$ for all $i \geq 0$.[6] The sequence $V(u_0), V(u_1) \ldots$ is a *computation* of M that *starts at* u. The *language* of M, denoted $L(M)$, consists of all computations of M that start at a state in W_0. We say that M *satisfies* an LTL formula φ if all computations of $L(M)$ satisfy φ.

The *verification problem* for LTL is to check whether a transition system P satisfies an LTL formula φ. The verification problem for LTL can be solved in time $O(|P| \cdot 2^{|\varphi|})$ [78]. In other words, there is a model-checking algorithm for LTL whose running time is *linear* in the size of the program and *exponential* in the size of the specification. This is acceptable since the size of the specification is typically significantly smaller than the size of the program.

The dominant approach today to LTL model checking is the *automata-theoretic approach* [116] (see also [112]). The key idea underlying the automata-theoretic approach is that, given an LTL formula φ, it is possible to construct a finite-state automaton A_φ that accepts all computations that satisfy φ. The type of finite automata on infinite words we consider is the one defined by Büchi [16] (c.f. [108]). A *Büchi automaton* is a tuple $A = (\Sigma, S, S_0, \rho, F)$, where Σ is a finite alphabet, S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function, and $F \subseteq S$ is a set of accepting states. A *run* of A over an infinite word $w = a_1 a_2 \cdots$, is a sequence $s_0 s_1 \cdots$, where $s_0 \in S_0$ and $s_i \in \rho(s_{i-1}, a_i)$ for all $i \geq 1$. A run $s_0, s_1, \ldots$ is *accepting* if there is some designated state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many i's such that $s_i = s$. The infinite word w is *accepted* by A if there is an accepting run of A over w. The *language* of infinite words accepted by A is denoted $L(A)$. The following fact establishes the correspondence between LTL and Büchi automata: Given an LTL formula φ, one can build a Büchi automaton $A_\varphi = (\Sigma, S, S_0, \rho, , F)$, where $\Sigma = 2^{Prop}$ and $|S| \leq 2^{O(|\varphi|)}$, such that $L(A_\varphi)$ is exactly the set of computations satisfying the formula φ [117].

This correspondence enables the reduction of the verification problem to an automata-theoretic problem as follows [116]. Suppose that we are given a system M and an LTL formula φ: (1) construct the automaton $A_{\neg\varphi}$ that corresponds to the *negation* of the formula φ, (2) take the product of the system M and the automaton $A_{\neg\varphi}$ to obtain an automaton $A_{M,\varphi}$, and (3) check that the automaton $A_{M,\varphi}$ is nonempty, i.e., that it accepts *some* input. If it does not, then the design is correct. If it does, then the design is incorrect and the accepted input is an incorrect computation. The incorrect computation is presented to the user as a finite trace, possibly followed by a cycle.

The linear-time framework is not limited to using LTL as a specification language. There are those who prefer to use automata on infinite words as a specification formalism [117]; in fact, this is the approach of FormalCheck [72]. In this approach, we are given a design represented as a finite transition system M and a property represented by a Büchi (or a related variant) automaton P. The design is correct if all computations in $L(M)$ are accepted by P, i.e., $L(M) \subseteq L(P)$. This approach is called the *language-containment*

[6] It is important to consider infinite paths, since we are interested in ongoing computations. Deadlock and termination can be modeled explicitly via sink state.

approach. To verify M with respect to P we: (1) construct the automaton P^c that *complements* P, (2) take the product of the system M and the automaton P^c to obtain an automaton $A_{M,P}$, and (3) check that the automaton $A_{M,P}$ is nonempty. As before, the design is correct iff $A_{M,P}$ is empty.

3.2 Advantages

The advantages of the linear-time framework are:

- Expressiveness: The linear framework is more natural for verification engineers. In the linear framework both designs and properties are represented as finite-state machines (we saw that even LTL formulas can be viewed as finite-state machines); thus verification engineers employ the same conceptual model when thinking about the implementation and the specification [77].
- Compositionality: The linear framework supports the assume-guarantee methodology. An assumption on the environment is simply expressed as a property E. Thus, instead of checking that $L(M) \subseteq L(P)$, we check that $L(M) \cap L(E) \subseteq L(P)$ [92]. Furthermore, we can add assumptions incrementally. Given assumptions $E_1, \ldots, E_k$, one needs to check that $L(M) \cap L(E_1) \cap \ldots \cap L(E_k) \subseteq L(P)$. The linear formalism is strong enough to express very general assumptions, as it can describe arbitrary finite-state machines, nondeterminism, and fairness. In fact, it is known that to prove linear-time properties of the parallel composition $M||E_1|| \ldots ||E_k$, it suffices to consider the linear-time properties of the components $M, E_1, \ldots, E_k$ [81].
- Semi-formal verification: As we saw, in the linear framework language containment is reduced to language emptiness, i.e., a search for a single computation satisfying some conditions. But this is precisely the same principle underlying dynamic validation. Thus, the linear framework offers support for search procedures that can be varied continuously from dynamic validation to full formal verification. This means that techniques for semi-formal verification can be applied not only to invariances but to much more general properties and can also accommodate assumptions on the environment [67]. In particular, linear-time properties can be compiled into "checkers" of simulation traces, facilitating the integration of formal verification with a traditional validation environment [67]. Such checkers can also be run as run-time monitors, which can issue an error message during a run in which a safety property is violated [20].
- Property-specific abstraction: Abstraction is a powerful technique for combating state explosion. An abstraction suppresses information from a concrete state-space by mapping it into a smaller, abstract state-space. As we saw, language containment is reduced to checking emptiness of a system $A_{M,\varphi}$ (or $A_{M,P}$) that combines the design with the complement of the property. Thus, one can search for abstractions that are tailored to the specific property being checked, resulting in more dramatic state-space reductions [43].
- Combined methods: Nonemptiness of automata can be tested enumeratively [29] or symbolically [109]. Recent work has shown that for invariances enumerative and symbolic methods can be combined [101]. Since in the linear framework model checking of general safety properties can be reduced to invariance checking of the composite system $A_{M,\varphi}$ (or $A_{M,P}$) [67], the enumerative-symbolic approach can be applied to a large class of properties and can also handle assumptions.
- Uniformity: The linear framework offers a uniform treatment of model checking, abstraction, and refinement [1], as all are expressed as language containment. For example, to show that a design P_1 is a refinement a design P_2, we have to check that $L(P_1) \subseteq L(P_2)$. Similarly, one abstracts a design M by generating a design M' that has more behaviors than M, i.e., $L(M) \subseteq L(M')$. Thus, an implementor can focus on an efficient implementation of the language-containment test. This means that a linear-time model checker can also be used to check for *sequential equivalence* of finite-state

machines [56]. Furthermore, the automata-theoretic approach can be easily adapted to perform quantitative timing analysis, which computes minimum and maximum delays over a selected subset of system executions [21].

– Bounded Model Checking: In linear-time model checking one searches for a counterexample trace, finite or infinite, which falsifies the desired temporal property. In bounded model checking, the search is restricted to a trace of a bounded length, in which the bound is selected before the search. The motivating idea is that many errors can be found in traces of relatively small length (say, less than 40 cycles). The restriction to bounded-length traces enables a reduction to propositional satisfiability (SAT). It was recently shown that SAT-based model checking can often significantly outperform BDD-based model checkers [11]. As bounded model checking is essentially a search for counterexample traces of bounded length, its fits naturally within the linear-time framework, but does not fit the branching rime framework.

3.3 Beyond LTL

Since the proposal by Pnueli [91] to apply LTL to the specification and verification of concurrent programs, the adequacy of LTL has been widely studied. One of the conclusions of this research is that LTL is not expressive enough for the task. The first to complain about the expressive power of LTL was Wolper [119] who observed that LTL cannot express certain ω-regular events (in fact, LTL expresses precisely the star-free ω-regular events [107]). As was shown later [79], this makes LTL inadequate for *compositional* verification, since LTL is not expressive enough to express assumptions about the environment in modular verification. It is now recognized that a linear temporal property logic has to be expressive enough to specify all ω-regular properties [117]. What then should be the "ultimate" temporal property-specification language?

Several extensions to LTL have been proposed with the goal of obtaining full ω-regularity:

– Vardi and Wolper proposed ETL, the extension of LTL with temporal connectives that correspond to ω-automata [119, 117]), ETL essentially combines two perspective on hardware specification, the operational perspective (finite-state machines) with the behavioral perspective (temporal connectives). Experience has shown that both perspectives are useful in hardware specification.
– Banieqbal and Barringer proposed extending LTL with fixpoint operators [6] (see also [111]), yielding a linear μ-calculus (cf. [60]), and
– Sistla, Vardi, and Wolper proposed QPTL, the extension of LTL with quantification over propositional variables [104].

It is not clear, however, that any of these approaches provides an adequate solution from a pragmatic perspective: implementing full ETL requires a complementation construction for Büchi automata, which is still a topic under research [64]; fixpoint calculi are notoriously difficult for users, and are best thought as an intermediate language; and full QPTL has a nonelementary time complexity [104].

Another problem with these solutions is the lack of temporal connectives to describe past events. While such connectives are present in works on temporal logic by philosophers (e.g., [96, 89]), they have been purged by many computer scientists, who were motivated by a strive for minimality, following the observation in [45] that in applications with infinite future but finite past, past connectives do not add expressive power. Somewhat later, however, arguments were made for the restoration of the past in temporal logic. The first argument is that while past temporal connectives do not add any expressive power the price for eliminating them can be high. Many natural statements in program specification are much easier to express using past connectives [98]. In fact, the best known procedure to eliminate past connectives in LTL may cause a significant blow-up of the considered formulas [79].

A more important motivation for the restoration of the past is again the use of temporal logic in modular verification. In global verification one uses temporal formulas that refer to locations in the program text [92]. This is absolutely *verboten* in modular verification, since in specifying a module one can refer only to its external behavior. Since we cannot refer to program location we have instead to refer to the history of the computation, and we can do that very easily with past connectives [7].

We can summarize the above arguments for the extension of LTL with a quote by Pnueli [92]: "In order to perform compositional specification and verification, it is *convenient* to use the past operators but *necessary* to have the full power of ETL."

3.4 A Pragmatic Proposal

The design of a temporal property-specification language in an industrial setting is not a mere theoretical exercise. Such an effort was recently undertaken by a formal-verification group at Intel. In designing such a language one has to balance competing needs:

- Expressiveness: The logic has to be expressive enough to cover most properties likely to be used by verification engineers. This should include not only properties of the unit under verification but also relevant properties of the unit's environment.
- Usability: The logic should be easy to understand and to use for verification engineers. At the same time, it is important that the logic has rigorous formal semantics to ensure correct compilation and optimization and enable formal reasoning.
- Closure: The logic should enable the expression of complex properties from simpler one. This enables maintaining libraries of properties and property templates. Thus, the logic should be closed under all of its logical connectives, both Boolean and temporal.
- History: An industrial tool is not developed in a vacuum. At Intel, there was already a community of model-checking users, who were used to a certain temporal property-specification language. While the new language was not expected to be fully backward compatible, the users demanded an easy migration path.
- Implementability: The design of the language went hand-in-hand with the design of the model-checking tool [4]. In considering various language features, their importance had to be balanced against the difficulty of ensuring that the implementation can handle these features.

The effort at Intel culminated with the design of FTL, a new temporal property specification language [5]. FTL is the temporal logic underlying *ForSpec*, which is Intel's new formal specification language . A model checker with ForSpec as its formal specification has been deployed at Intel since 2000. ForSpec is now also part of Synopsys's Open Vera framework (`www.open-vera.com`). The key features of FTL are as follows:

- FTL is a linear temporal logic, with a limited form of past connectives, and with the full expressive power of ω-regular languages,
- it is based on a rich set of logical and arithmetical operations on bit vectors to describe state properties,
- it enables the user to define temporal connectives over time windows,
- it enables the user to define regular events, which are regular sequences of Boolean events, and then relate such events via special connectives, adapted from *propositional dynamic logic* (PDL) [41],
- it contains constructs that enable the users to model multiple clock and reset signals, which is useful in the verification of hardware design.

There is no reason, however, to think that FTL is the final word on temporal property-specification languages. First, one would not expect to have an "ultimate" temporal property-specification language any more than one would expect to have an "ultimate" programming language. There are also, in effect, two competing languages. FormalCheck uses a built-in library of automata on infinite words as its property-specification language [73], while Cadence SMV[7] uses LTL with universal quantification over propositional variables. Our hope is that the publication of this paper and of [5], concomitant with the release of an FTL-based tool to Intel users, would result in a dialog on the subject of property-specification logic between the research community, tool developers, and tools users. It is time, we believe to close the debate on the linear-time vs. branching time issue, and open a debate on linear-time languages. For example, while, as noted above, QPTL has a nonelementary time complexity, this is a worst-case bound. It is worth pointing out that the model-checking tool MONA [34, 59] successfully verifies many specifications given in a second-order logic that has a nonelementary time complexity. There are also many who argue in favor of an interval-based linear-time logics, cf. [88, 50, 99].[8] Thus, there are at least three distinct approaches within the linear-time "camp".

4 Discussion

Does the discussion above imply that 20 years of research into CTL-based model checking have led to a dead end? To the contrary! The key algorithms underlying symbolic model checking for CTL are efficient graph-theoretic reachability and fair-reachability procedures (cf. [100]). The essence of the language-containment approach is that verification of very general linear-time properties can be reduced to reachability or fair-reachability analysis, an analysis that is at the heart of CTL-based model-checking engines. Thus, a linear-time model checker can be built on top of a CTL model checker, as in Cadence SMV, leveraging two decades of science and technology in CTL-based model checking.

It should also be stated clearly that our criticism of branching time is in the specific context of property-specification languages for model checking. There are contexts in which the branching-time framework is the natural one. For example, when it comes to the synthesis of reactive systems, one has to consider a branching-time framework, since all possible strategies by the environment need to be considered [94, 95][9]. Even when the goal is a simple reachability goal, one is quickly driven towards using branching-time specification languages, cf. [32]. CTL is probably too weak for such multi-agent settings and more expressive specification languages are more appropriate, cf. [2, 110].

Some argue that even if one uses mainly linear-time specifications, CTL is needed for *vacuity checking*. Suppose that the model M satisfies the LTL property $G(p \rightarrow Fq)$. To ensure that the property did not pass vacuously, because p never holds, one should check that M satisfies the CTL property EFp. As shown, however, in [69], vacuity checking for linear-time properties can be done and has to be done within the linear-time framework. Nevertheless, we do believe that even in the context of model checking, branching time has its place. In model checking one checks whether a transition system M satisfies a temporal formula φ. The transition system M is obtained either by compilation from an actual design, typically expressed using a *hardware description language* such as VHDL or Verilog, or is constructed manually by the user using a modeling language, such as SMV's SML [84]. In the latter case, the user often wishes to "play" with M, in order to ensure that M is a good model of the system under consideration. Using CTL, one can express properties such as $AGEFp$, which are structural rather than behavioral. A CTL-based model checker enables the user to "play" with M by checking its structural properties and ensure that the model does not have *too few* behaviors. It is important to note, however, that even for this application CTL is not powerful enough. To express the fact that a state

[7] `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`
[8] An extensive bibliography on interval temporal logics can be found at `http://www.cms.dmu.ac.uk/~cau/itlhomepage/`.
[9] But note the specification there is expressed in a linear-time formalism.

satisfying p can be reached even in the presence of an adversarial environment, one has to use a stronger logic, such as the μ-calculus [71].

Acknowledgment: I'd like to thank Orna Kupferman, my close collaborator over the last few years, for numerous discussions on linear vs. branching time. Also, my collaboration with members the formal-verification team in the Intel Design Center, Israel, during the 1997-2000 period, in particular, with Roy Armoni, Limor Fix, Ranan Fraer, Gila Kamhi, Yonit Kesten, Avner Landver, Sela Mador-Haim and Andreas Tiemeyer, gave me invaluables insights into formal verification in an industrial setting. I'd also like to thank Rajeev Alur, Ben Moszkowski, and Javier Esparza for helping me with some references.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. on Foundations of Computer Science*, pages 100–109, Florida, October 1997.
3. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proc. 18 Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 175–188. Springer-Verlag, 1998.
4. R. Armoni, L. Fix, A. Flaisher, R. Gerth, T. Kanza, A. Landver, S. Mador-Haim, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec compiler. Submitted, 2001.
5. R. Armoni, L. Fix, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, A. Tiemeyer, E. Singerman, M.Y. Vardi, and Y. Zbar. The ForSpec temporal language: A new temporal property-specification language. In *Proc. 8th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, Lecture Notes in Computer Science 2280, pages 296–311. Springer-Verlag, 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-Verlag, 1987.
7. H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a framework. In S.D. Brookes et al., editor, *Seminar in Concurrency*, Lecture Notes in Computer Science, Vol. 197, pages 35–61. Springer-Verlag, Berlin/New York, 1985.
8. I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193, Stanford, June 1994.
9. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Computer Aided Verification, Proc. 10th International Conference*, volume 1427 of *Lecture Notes in Computer Science*, pages 184–194. Springer-Verlag, 1998.
10. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
11. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Design Automation Conference*, pages 317–320. IEEE Computer Society, 1999.
12. R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *Computer Aided Verification, Proc. 11th International Conference*, volume 1633 of *Lecture Notes in Computer Science*, pages 222–235. Springer-Verlag, 1999.
13. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, July 1997. Springer-Verlag.
14. 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 *Computer Aided Verification, Proc. 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.
15. M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

16. J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.
17. 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 Symp. on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.
18. 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, June 1992.
19. J. P. Burgess. Logic and time. *J. Symbolic Logic*, 44:566–582, 1979.
20. W. Cai and S.J. Turner. An approach to the run-time monitoring of parallel programs. *The Computer JournaL*, 37(4):333–345, 1994.
21. S. Campos, E.M. Clarke, and O. Grumberg. Selective quantitative analysis and interval model checking: Verifying different facets of a system. In *Computer-Aided Verification, Proc. 8th Int'l Conf.*, volume 1102 of *Lecture Notes in Computer Science*, pages 257–268. Springer-Verlag, Berlin, 1996.
22. E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proc. Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer-Verlag, 1988.
23. 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 Languages and Systems*, 8(2):244–263, January 1986.
24. E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School)*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer-Verlag, 1993.
25. E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automation Conference*, pages 427–432. IEEE Computer Society, 1995.
26. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
27. E.M. Clarke and R.P. Kurshan. Computer aided verification. *IEEE Spectrum*, 33:61–67, 1986.
28. E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In R. Parikh, editor, *Proc. 4th IEEE Symp. on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, 1989.
29. 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.
30. W. Damm, G. Döhmen, V. Gerstner, and B. Josko. Modular verification of Petri nets: the temporal logic approach. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (Proceedings of REX Workshop)*, volume 430 of *Lecture Notes in Computer Science*, pages 180–207, Mook, The Netherlands, May/June 1989. Springer-Verlag.
31. D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 479–490. Springer-Verlag, June 1993.
32. M. Daniele, P. Traverso, and M.Y. Vardi. Strong cyclic planning revisited. In S. Biundo and M. Fox, editors, *5th European Conference on Planning*, pages 34–46, 1999.
33. N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification, Proc. 11th International Conference*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1999.
34. J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: new techniques for WS1S and WS2S. In *Computer Aided Verification, Proc. 10th International Conference*, volume 1427 of *Lecture Notes in Computer Science*, pages 516–520. Springer-Verlag, Berlin, 1998.
35. E.A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, MIT press, 1990.
36. E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th InternationalColloq. on Automata, Languages and Programming*, pages 169–181, 1980.
37. E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30:1–24, 1985.

38. 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.
39. E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 328–337, White Plains, October 1988.
40. E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pages 84–96, New Orleans, January 1985.
41. M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.
42. K. Fisler. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language, and Information*, 8:323–361, 1999.
43. K. Fisler and M.Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In G. Gopalakrishnan and P. Windley, editors, *Proc. Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD)*, number 1522 in Lecture Notes in Computer Science, pages 115–132. Springer-Verlag, 1998.
44. R. Fraer, G. Kamhi, L. Fix, and M.Y. Vardi. Evaluating semi-exhausting verification techniques for bug hunting. In *Proc. 1st Int'l Workshop on Symbolic Model Checking (SMC'99)*, Electronic Notes in Theoretical Computer Science, pages 11–22. Elsevier, 1999.
45. 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, January 1980.
46. 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, August 1995.
47. R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.
48. R. Goering. Model checking expands verification's scope. *Electronic Engineering Today*, February 1997.
49. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.
50. J.Y. Halpern, Manna Z, and B. Moszkowski. A hardware semantics based on temporal intervals. In *Proc. 10th Int'l Colloq. on Automata, Languages and Programming*, volume 154, pages 278–291. Springer-Verlag, 1983.
51. R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *Computer Aided Verification, Proc. 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, 1996.
52. D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer-Verlag, 1985.
53. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of ACM*, 32:137–161, 1985.
54. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
55. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
56. S.Y. Huang and K.T. Cheng. *Formal Equivalence Checking and Design Debugging*. Kluwer Academic publishers, 1998.
57. C.B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing 83: Proc. IFIP 9th World Congress*, pages 321–332. IFIP, North-Holland, 1983.
58. B. Josko. Verifying the correctness of AADL modules using model checking. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (Proceedings of REX Workshop)*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400, Mook, The Netherlands, May/June 1989. Springer-Verlag.
59. N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, Lecture Notes in Computer Science, 1998.
60. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

61. D. Kozen and L. Tiuryn. Logics of programs. *Handbook of Theoretical Computer Science*, pages 789–840, 1990.
62. O. Kupferman and M.Y. Vardi. Module checking. In *Computer Aided Verification, Proc. 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 1996.
63. O. Kupferman and M.Y. Vardi. Module checking revisited. In *Computer Aided Verification, Proc. 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 1997.
64. O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. In *Proc. 5th Israeli Symp. on Theory of Computing and Systems*, pages 147–158. IEEE Computer Society Press, 1997.
65. O. Kupferman and M.Y. Vardi. Freedom, weakness, and determinism: from linear-time to branching-time. In *Proc. 13th IEEE Symp. on Logic in Computer Science*, pages 81–92, June 1998.
66. O. Kupferman and M.Y. Vardi. Relating linear and branching model checking. In *IFIP Working Conference on Programming Concepts and Methods*, pages 304 – 326, New York, June 1998. Chapman & Hall.
67. O. Kupferman and M.Y. Vardi. Model checking of safety properties. In *Computer Aided Verification, Proc. 11th International Conference*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer-Verlag, 1999.
68. O. Kupferman and M.Y. Vardi. Robust satisfaction. In *Proc. 10th Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 383–398. Springer-Verlag, August 1999.
69. O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1999.
70. O. Kupferman and M.Y. Vardi. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems*, 22:87–128, 2000.
71. O. Kupferman, M.Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164:322–344, 2001.
72. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.
73. R.P. Kurshan. *FormalCheck User's Manual*. Cadence Design, Inc., 1998.
74. P. Ladkin and S. Leue. What do message sequence charts means? In R.L. Tenney, P.D. Amer, and M.Ü. Uyar, editors, *Proc. 6th Int'l Conf. on Formal Description Techniques*. North Holland, 1994.
75. 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, January 1980.
76. L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systenms*, 5:190–222, 1983.
77. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.
78. 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, New Orleans, January 1985.
79. 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, Brooklyn, June 1985. Springer-Verlag.
80. M. Maidl. *Using Model Checking for System Verification*. PhD thesis, Ludwig-Maximilians-Universität München, 2000.
81. Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear time, branching time, and partial order in logics and models for concurrency*, volume 345 of *Lecture Notes in Computer Science*, pages 201–284. Springer-Verlag, 1989.
82. Z. Manna and A. Pnueli. Temporal specification and verification of reactive modules. Technical report, Weizmann Institute, 1992.
83. K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proc. 10th Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, 1998.
84. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

85. R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society, September 1971.
86. R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Clifs, 1989.
87. B. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Engineering*, 7:417–426, 1981.
88. B. Moszkowski. A temporal logic for multi-level reasoning about hardware. In *Proc. 6th Int'l Symp. on Computer Hardware Description Languages*, pages 79–90. North-Holland, 1983.
89. A. Urquhart N. Rescher. *Temporal Logic*. Springer-Verlag, 1971.
90. D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI Conf. on Theoretical Computer Science*, Lecture Notes in Computer Science, Vol. 104. Springer-Verlag, Berlin/New York, 1981.
91. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science*, pages 46–57, 1977.
92. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 123–144. Springer-Verlag, 1985.
93. A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th International Colloquium on Automata, Languages and Programming*, pages 15–32. Lecture Notes in Computer Science, Springer-Verlag, 1985.
94. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, Austin, January 1989.
95. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th International Colloquium on Automata, Languages and Programming*, volume 372, pages 652–671. Lecture Notes in Computer Science, Springer-Verlag, July 1989.
96. A. Prior. *Past, Present, and Future*. Oxford Univ. Press, 1951.
97. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.
98. W.P. DeRoever R. Koymans, J. Vytopil. Real-time programming and asynchronous message passing. In *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, pages 187–197, 1983.
99. Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon, and G. Kutty. Interval logics and their decision procedures, part i: An interval logic. *Theoretical Computer Science*, 166(1–2):1–47, 1996.
100. K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, Lecture Notes in Computer Science 1954, pages 143–160. Springer-Verlag, 2000.
101. K. Ravi and F. Somenzi. High-density reachability analysis. In *Proc. International Conference on Computer-Aided Design*, pages 154–158, California, 1995.
102. T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development*, 41(4:5), 1997.
103. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, 32:733–749, 1985.
104. 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.
105. E.W. Stark. *Foundations of theory of specifications for distributed systems*. PhD thesis, M.I.T., 1984.
106. C. Stirling. Modal and temporal logics. *Handbook of Logic in Computer Science*, 2:477–563, 1992.
107. W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Computation*, 48:261–283, 1981.
108. W. Thomas. Automata on infinite objects. *Handbook of Theoretical Computer Science*, pages 165–191, 1990.
109. H.J. Touati, R.K. Brayton, and R. Kurshan. Testing language containment for ω-automata using BDD's. *Information and Computation*, 118(1):101–109, April 1995.
110. W. van der Hoek and M. Woolridge. Tractable multiagent planning for epistemic goals. In *Proc. 1st Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS-02)*, 2002.

111. M.Y. Vardi. A temporal fixpoint calculus. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 250–259, San Diego, January 1988.
112. M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, Berlin, 1996.
113. M.Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In *Proc. 13th IEEE Sym.. on Logic in Computer Science*, pages 394–405, 1998.
114. M.Y. Vardi. Sometimes and not never re-revisited: on branching vs. linear time. In D. Sangiorgi and R. de Simone, editors, *Proc. 9th Int'l Conf. on Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 1–17, 1998.
115. M.Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc 17th ACM Symp. on Theory of Computing*, pages 240–251, 1985.
116. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.
117. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.
118. I. Walukiewicz. Model checking ctl properties of pushdown systems. In *Proc. 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*, pages 127–138, New Delhi, India, December 2000. Springer-Verlag.
119. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.
120. J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. In *Computer Aided Verification, Proc. 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 376–387. Springer-Verlag, 1997.