Linear vs. Branching Time: A Complexity-Theoretic Perspective

Moshe Y. Vardi*
Rice University

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.

A closer examination of the the issue reveals, however, that the computational superiority of the branching time framework is perhaps illusory. In this talk we will compare the complexity of branching-time verification vs. linear-time verification in many scenarios, and show that linear-time verification is not harder and often is even easier than branching-time verification. This suggests that the tradeoff between branching and linear time is not a simple tradeoff between complexity and expressiveness.

1 Introduction

Temporal logics, which are modal logics geared towards the description of the temporal ordering of events, have been adopted as a powerful tool for specifying and verifying open programs [MP92a, MP95]. Two possible views regarding the nature of time induce two types of temporal logics [Lam80]. 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 operators G ("always"), F ("eventually"), X ("next"), and U ("until"). For example, the LTL formula $\psi = G(\mathit{req} \rightarrow F\mathit{grant})$ refers to the atomic propositions *req* and *grant* and is true in a computation iff every state in the computation in which *req* hold is followed by some state in the future in which *grant* holds. The branching temporal logic CTL* augments LTL by the path quantifiers E ("there exists a computation") and A ("for all computations"). For example, the CTL* formula $\varphi = AG(\mathit{req} \rightarrow EF\mathit{grant})$ is true in a computation tree iff in all its computations, every state in which *req* holds branches to at least one computation in which *grant* eventually holds. The branching temporal logic CTL is a fragement of CTL* in which every temporal operator is preceded by a path quantifier. For example, while φ above is a CTL formula, the CTL* formula $AG(\mathit{req} \rightarrow F\mathit{grant})$ is not a CTL formula. The expressive powers of LTL and CTL are incomparable [Lam80], while CTL* is more expressive than both LTL and CTL [EH86].

The discussion of the relative merits of linear versus branching temporal logics is by now almost 20-years old [Lam80, CE81, QS81, Pnu81, BMP83,

*Address: CS Department, Rice University, Houston TX 77005-1892, U.S.A. Email: `vardi@cs.rice.edu`, URL: `http://www.cs.rice.edu/~vardi`. Supported in part by NSF grants CCR-9628400 and CCR-9700061, and by a grant from the Intel Corporation.

Pnu85b, EH86, Sti87]. One of the most frequent arguments in this discussion relates to the computational complexity of algorithmic verification. The discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* programs [CE81, QS81, LP85, VW86] is one of the most significant developments in the area of formal verification in the last two decades. This discovery derives its significance both from the fact that many synchronization and communication protocols, as well as many digital systems, can be modeled as finite-state systems, and from the great ease of use of fully algorithmic methods [BBG$^+$94]. Finite-state systems can be modeled by transition systems where each state has a bounded description, and hence can be characterized by a fixed number of Boolean atomic propositions. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired behavior by checking whether the labeled transition system satisfies a temporal logic formula that specifies this behavior. When the model checker fails to verify the behavior, it produces a counterexample, which is in essence a trace of an incorrect behavior. Indeed, model checkers have demonstrated their ability to discover very subtle flaws in designs of considerable complexity [CK86].

Given a transition system M and a linear temporal logic formula ψ, the model-checking problem for M and ψ is to decide whether ψ holds in all the computations of M. When ψ is a branching temporal logic formula, the problem is to decide whether ψ 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 $O(nm)$ [CES86], while, for the linear temporal logic LTL, and even for the more expressive CTL*, model-checking algorithms run in time $n2^{O(m)}$ [LP85, EL85]. Since LTL model checking is PSPACE-complete [SC85], the latter bound probably cannot be improved. Because of these results, one of the beliefs dominating the linear-vs.-branching-time discussion has traditionally been that "model checking for CTL is easy, while model checking for LTL is hard". Furthermore, the fact that the complexity of model checking is the same for LTL and CTL* was taken as evidence that "branching is free".

Our goal in this overview paper is to show that the belief that CTL is superior to LTL from the computational-complexity perspective is rather simplistic. We will show that in certain verfication scenarios CTL does not dominate LTL computationaly. As a simple example of what we mean by "verification scenario" consider a designer who just completed a design M and is trying to verify it against a specification φ. Since the design is rather "fresh", the designer fully expects the verification to fail and is using the model checker as a sophisticated debugger. What should the designer conclude if, surprisingly, the model checker succeeded in verifying M against φ? A plausible conclusion is that the specification φ is flawed. For example, if φ is a *valid* formula, then every design satisfies φ. Thus, the designer may prudently want to check that φ is *not* valid before concluding that the design M is correct. This argument suggests that a model checker ought to include also the functionality of validity checking. Note, however, that while the validity problem for LTL is PSPACE-complete [SC85], the validity problem for CTL is EXPTIME-complete [FL79, EH85], and the validity problem for CTL* is 2EXPTIME-complete [VS85, EJ88].

In the rest of this paper we examine four additional verification scenarios: modular verification [KV95], verification of open systems [KV96, KVW98], verification of concurrent systems [BVW94, Kup95], and automata-theoretic verification [KV98]. In each case we show that CTL does not dominate LTL computationally. In the concluding section of the paper we discuss the implication of these results on the linear-vs.-branching-time discussion. We assume familiarity with the syntax and semantics of temporal logic [Eme90, KT90, Sti92].

2 Modular Model Checking

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 even linear-time 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. [BCM+92]).

Modular verification is one possible way to address the state-explosion problem, cf. [CLM89]. 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 [MC81, Lam83, Jon83], 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 [Pnu85a], 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 [GL94]. 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 [GL94], this is not equivalent to M satisfying $\varphi \to \psi$.) We call this *branching modular model checking*. Furthermore, it is argued in [GL94], as well as in [DDGJ89, Jos89, GL94, DGG93], 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 [GL94] is on using $\forall$CTL, the universal fragment of CTL, for both the assumption and the guarantee. In this paper, we focus on the branching modular model-checking problem, where assumptions and guarantees are in both $\forall$CTL and in the more expressive $\forall$CTL*, the universal fragment of CTL*.

A *module* $M = \langle AP, W, R, W_0, L \rangle$ consists of a set AP of atomic propositions, a set W of states, a transition relation $R \subseteq W \times W$, a set W_0 of initial states, and a labeling function $L : W \to 2^{AP}$ that maps each state to a set of atomic propositions that hold in this state. A *computation* of a module is a sequence of states, $\pi = w_0, w_1, \ldots$ such that for every $i \geq 0$, we have that $\langle w_i, w_{i+1} \rangle \in R$. We use $M, w \models \varphi$ to denote that the state w in M satisfies the formula φ. A module M satisfies a formula φ if φ holds in *all* initial states of M. The *size* of M is the sum of the number of states in W and the number of transitions in R.

Let M and M' 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 $\langle AP'', W'', R'', W_{0''}, L'' \rangle$, where $AP'' = AP \cup AP'$, $W'' = \{\langle w, w' \rangle : L(w) \cap AP' = L'(w') \cap AP\}$, $W_{0''} = (W_0 \times W_{0'}) \cap W''$, $R'' = \{\langle \langle w, w' \rangle, \langle s, s' \rangle \rangle : \langle w, s \rangle \in R$ and $\langle w', s' \rangle \in R'\}$, and $L''(\langle w, w' \rangle) =$

$L(w) \cup L'(w')$ 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 hold, which we called the *branching modular model-checking problem*.

Theorem 2.1 [KV95]

(1) *The branching modular model-checking problem for* $\forall$*CTL is PSPACE-complete.*

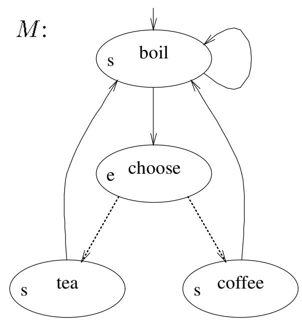
(2) *The branching modular model-checking problem for* $\forall$*CTL** *is EXPSPACE-complete.*

Thus, in the context of modular model checking, $\forall$CTL has the same computational complexity as LTL, while $\forall$CTL* is exponentially harder.

3 Verification 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* [HP85]. As an example of closed and open systems, consider two drink-dispensing machines. One machine, which is a closed system, repeatedly boils water, makes an *internal nondeterministic* choice, and serves either coffee or tea. The second machine, which is an open system, repeatedly boils water, asks the environment to choose between coffee and tea, and *deterministically* serves a drink according to the *external* choice [Hoa85]. Both machines induce the same infinite tree of possible executions. Nevertheless, while the behavior of the first machine is determined by internal choices the behavior of the second machine is determined also by external choices, made by its environment. 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.

We 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. Consider the open module M presented below. It has three system states (*boil*, *tea*, and *coffee*), and it has one environment state (*choose*). It models the second drink-dispensing machine described above. When M is in the system state *boil*, we know exactly what its possible next states are. It can either stay in the state *boil* or move to the state *choose*. In contrast, when M is in the environment state *choose*, there is no certainty with respect to the environment and we can not be sure that both *tea* and *coffee* are possible next states. For example, it might be that for some users of the machine, coffee is not a desirable option. If we ignore the partition of M's states to system and environment states and regard it as a (standard) module P, then it models the first drink-dispensing machine described above.



To see the difference between the semantics of modules and open modules, let us consider two questions. Is it always possible for the first machine to eventually serve tea? This is equivalent to asking whether P satisfies the CTL formula $AGEF$*tea*, and the answer is positive. Is it always possible for the second machine to eventually serve tea? Here, the answer is negative. Indeed, if the environment always choose coffee, the second machine will never serve tea. Suppose now that we check with current model-checking tools whether it is always possible for the second machine to eventually serve tea, what will the answer be? Unfortunately, model-checking tools do not distinguish between closed and open systems and they will say that M does satisfy $AGEF$*tea*.

As discussed in [MP92b], 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 M to satisfy the specification, the program P and the module M satisfy exactly the same linear temporal logic formulas. From the example above we learn that when the specification is given in branching temporal logic, we do need to take into account the uncertainty about the environment. 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 ψ. For an open module M, let V_M denote the unwinding of M into an infinite tree. We say that M satisfies ψ iff ψ 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 ψ to hold in every such tree, since, of course, we want the open system to satisfy its specification no matter how the environment behaves. For example, an environment for the second drink-dispensing machine is an infinite line of thirsty people waiting for their drinks. Since each person in the line can either like both coffee and tea, or like only coffee, or like only tea, there are many different possible environments to consider. Each environment induces a different tree. For example, an environment in which all the people in line do not like tea, induces a tree that has the left subtree of all its *choose* nodes pruned. Similarly, environments in which the first person in line like both coffee and tea induce trees in which the first *choose* node has two successors.

Formally, we model an open system by an *open module* $M = \langle AP, W_s, W_e, R, W_0, L\rangle$, where $\langle AP, W_s \cup W_e, R, W_0, L\rangle$ 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 $T \subseteq \mathbb{N}^*$ such that if $x \cdot c \in T$ where $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$, 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 $T \cap \mathbb{N}^{|x|+1} = \{x \cdot 0, x \cdot 1, \ldots, x \cdot n\}$ 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 behaviour 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 ψ, we say that M satisfies ψ, denoted $M \models_o \psi$, if all the trees in $exec(M)$ satisfy ψ. The problem of deciding whether M satisfies ψ is called *module checking*. We use $M \models \psi$ to indicate that when we regard M as a module (thus refer to all its states as system states), then M satisfies ψ. The problem of deciding whether $M \models \psi$ is the usual model-checking problem. Note that while $M \models_o \psi$ entails $M \models \psi$, all that $M \models \psi$ entails is that $M \not\models_o \neg\psi$. Indeed, $M \models_o \psi$ requires all the trees in $exec(M)$ to satisfy ψ. On the other hand, $M \models \psi$ means that the tree $\langle T_M, V_M \rangle$ satisfies ψ. Finally, $M \not\models_o \neg\psi$ only tells us that there exists some tree in $exec(M)$ that satisfies ψ. We can define module checking also with respect to linear-time specifications. We say that a module M satisfies an LTL formula ψ iff $M \models_o A\psi$.

Theorem 3.1 [KV96, KVW98]

(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*.

When analyzing the complexity of model checking, a distinction should be made between complexity in the size of the input structure and complexity in the size of the input formula; it is the complexity in size of the structure that is typically the computational bottleneck [LP85]. Let us now consider the *program complexity* [VW86] of module checking; i.e., the complexity of this problem in terms of the size of the input module, assuming the formula is fixed. It is known that the program complexity of LTL, CTL, and CTL* model checking is NLOGSPACE [VW86, BVW94]. This is very significant since it implies that if the system to be checked is obtained as the product of the components of a concurrent program (as is usually the case), the space required is polynomial in the size of these components rather than of the order of the exponentially larger composition.

We have seen that for CTL and CTL*, module checking is much harder than model checking. We now claim that when we consider program complexity, module checking is still harder (assuming that PTIME is different than NLOGSPACE).

Theorem 3.2 *The program complexity of CTL and CTL* module checking is PTIME-complete.*

4 Verification of Concurrent Systems

As noted earlier, a common cause of the *state-explosion* problem is that fact that the system under consideration is typically the concurrent 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 even algorithms with linear running time impractical. This difficulty, however, is hidden in the standard way of reporting the complexity of model checking, which is as a function of the size of the state space of the system under consideration. A better way to report this complexity is perhaps in term of the size of the state spaces of the underlying modules.

In a concurrent setting, we assume that have have k modules $M_1, \ldots, M_k$ and a formula ψ. We have to verify that $M_1 \| \ldots \| M_k \models \psi$. We define the complexity of model checking here with respect to the sizes of the components M_i's and the length of the formula ψ being checked. Analogously, the program complexity of model checking for concurrent systems is defined in terms of the size of the components of the concurrent system.

Theorem 4.1 [BVW94, Kup95]

(1) *The complexity of LTL, CTL and CTL* model checking for concurrent systems is PSPACE-complete.*

(2) *The program complexity of LTL, CTL and CTL* model checking for concurrent programs is PSPACE-complete.*

Thus, in a concurrent setting model checking of LTL, CTL, and CTL* have the same complexity, as the complexity is completely dominated by the "explosion" of the concurrent state space.

5 Automata-Theoretic Verification

In design verification one checks that an *implementation* meets its *specification*. So far we used different formalisms to describe implementations and specification; we used an automata-theoretic formalism to describe implementations and we used a logical formalism to describe specifications. It is sometimes more natural to express both implementations and specifications using the same formalism. Both the specification and the implementation describe the possible behaviors of the system, but the implementation is more concrete than the specification, or, equivalently, the specification is more abstract than the implementation (cf. [AL91]). This basic notion of verification suggests a top-down method for design development. Starting with a highly abstract specification, we can construct a sequence of "behavior descriptions". Each description refers to its predecessor as a specification, so it is less abstract than its predecessor. The last description contains no abstractions, and constitutes the final implementation. Hence the name *hierarchical refinement* for this methodology (cf. [LS84, LT87, Kur94]).

Here we use the automata-theoretic approach and describe both specifications and implementations as modules [Kur94]. We distinguish between two approaches to implementation of specifications. The first approach is *trace-based implementation*, where we require every computation of the implementation to *correlate* to some computation of the specification. The second approach is *tree-based implementation*, where we require every computation tree embodied in the implementation to correlate to some computation tree embodied in the specification. The exact notion of correct implementation then depends on how we interpret correlation. We can, for example, interpret correlation as identity. Then, correct trace-based implementation is one in which every computation is also a computation of the specification, and correct tree-based implementation is one in which every embodied computation tree is also embodied in the specification. Numerous interpretations of correlation are suggested and studied in the literature (see [Hen85, Mil89, AL91]). Here, we consider a very simple definition of correlation and interpret it as equivalence with respect to the variables joint to the implementation and the specification, as the implementation is typically defined over a wider set of variables, reflecting the fact that it is more concrete than the specification.

The tree-based approach is stronger in the following sense. If $\mathcal{I}$ is a correct tree-based implementation of the specification $\mathcal{S}$, then $\mathcal{I}$ is also a correct trace-based implementation of $\mathcal{S}$. As shown by Milner [Mil80], the opposite direction is not true. The two approaches to implementation are strongly related to the linear- versus branching-time dichotomy in temporal logic [Pnu85b]. The temporal-logic analogy to the strength of the tree-based approach is the expressiveness superiority of $\forall$CTL* over LTL [CD88]. Indeed, while a correct trace-based implementation is guaranteed to satisfy all the LTL formulas satisfied in the specification, a correct tree-based implementation is guaranteed to satisfy all the $\forall$CTL* formulas satisfied in the specification [GL94].

Here we consider and compare the complexity of the problem of determining whether $\mathcal{I}$ is a correct trace-based implementation of $\mathcal{S}$, and the problem of determining whether $\mathcal{I}$ is a correct tree-based implementation of $\mathcal{S}$. The different levels of abstraction in the implementation and the specification are reflected in their sizes. The implementation is typically much larger than the specification and it is its size that is the computational bottleneck. Therefore, of particular interest to us is the *implementation complexity* of these problems; i.e., their complexity in terms of $\mathcal{I}$, assuming $\mathcal{S}$ is fixed. This is analogous to the notion of program complexity discussed earlier.

We model specifications and implementations by modules. The modulels are defined over the sets $AP_\mathcal{I}$ and $AP_\mathcal{S}$ of *atomic propositions* used in the implementation and specification, respectively. Thus, the alphabets of the systems are $2^{AP_\mathcal{I}}$ and $2^{AP_\mathcal{S}}$. Recall that usually the implementation has more variables than the specification. Hence, $AP_{\mathcal{I}} \supseteq AP_{\mathcal{S}}$. We therefore interpret correlation as equivalence with respect to $AP_{\mathcal{S}}$. In other words, associating computations and computation trees of the implementation with these of the specification, we first projet them on $AP_{\mathcal{S}}$.

Within this framework, correct trace-based implementation corresponds to *language containment* and correct tree-based implementation corresponds to *simulation* [Mil71]. Since simulation can be checked in polynomial time, whereas the trace containment problem is PSPACE-complete, it seems that the tree-based approach is more efficient than the trace-based approach. This is reminiscent of the computational advantage of branching-time model checking over linear-time model checking. Once, however, we want our implementations and specifications to describe behaviors that satisfy both *liveness* and *safety* properties, modules are too weak. Then, we need the framework of *fair modules*. We consider *unconditional* fairness (also known as *impartiality*) [LPS81]. Within this framework, correct trace-based implementation corresponds to *fair containment* and correct tree-based implementation corresponds to *fair simulation* [BBLS92, ASB+94, GL94]. Hence, it is the complexity of these problems that should be examined when we compare the trace-based and the tree-based approaches.

A *fair module* $M = \langle AP, W, R, W_0, L, \alpha\rangle$ consists of a module $\langle AP, W, R, W_0, L\rangle$ and a fairness condition α. An unconditionl fairness condition α is simply a subset of W. Recall that a computation of S is a sequence $\pi = w_0, w_1, w_2, \ldots$ of states such that for every $i \geq 0$ we have $R(w_i, w_{i+1})$. In order to determine whether a computation is *fair*, we refer to the set $Inf(\pi)$ of states that π visits infinitely often. We say that π is fair if $Inf(\pi) \cap \alpha \neq \emptyset$. It is easy to see is that fair modules essentially constitute a notational variant of Büchi automata [Tho90].

Trace-based and tree-based implementations are formalized in terms of (fair) containment and (fair) simulation between an implementation $\mathcal{I}$ and a specification $\mathcal{S}$. Recall that $\mathcal{I}$ and $\mathcal{S}$ are given as fair modules over the alphabets $2^{AP_{\mathcal{I}}}$ and $2^{AP_{\mathcal{S}}}$ respectively, with $AP_{\mathcal{I}} \supseteq AP_{\mathcal{S}}$. For technical convenience, we assume that $AP_{\mathcal{I}} = AP_{\mathcal{S}}$; thus, the implementation and the specification are defined over the same alphabet. By taking, for each $\sigma \in 2^{AP_{\mathcal{I}}}$, the letter $\sigma \cap AP_{\mathcal{S}}$ instead the letter σ, our results are valid also for the case $AP_{\mathcal{I}} \supset AP_{\mathcal{S}}$.

For a state w, a w-computation is a computation $w_0, w_1, w_2, \ldots$ with $w_0 = w$. We use $\mathcal{L}(S^w)$ (resp., $\mathcal{L}_f(S^w)$) to denote the set of all words $\sigma_0 \cdot \sigma_1 \cdots \in \Sigma^\omega$ for which there exists a (resp., fair) w-computation $w_0, w_1, \ldots$ in S with $L(w_i) = \sigma_i$ for all $i \geq 0$. The language $\mathcal{L}(S)$ (resp., $\mathcal{L}_f(S)$) of S is then defined as $\bigcup_{w \in W_0} \mathcal{L}(S^w)$ (resp., $\bigcup_{w \in W_0} \mathcal{L}_f(S^w)$). Given two fair modules M and M', trace-based implementation can now be characterized in terms of language *containment*, i.e, $\mathcal{L}(S) \subseteq \mathcal{L}(S')$, and *fair containment*, i.e., $\mathcal{L}_f(S) \subseteq \mathcal{L}_f(S')$. This gives rise to the *containment probem* (decide if $\mathcal{L}(S) \subseteq \mathcal{L}(S')$) and the *fair containment problem* (decide if $\mathcal{L}_f(S) \subseteq \mathcal{L}_f(S')$).

While containment and fair containment refer to each word in $\mathcal{L}(S)$ and $\mathcal{L}_f(S)$ independently, *simulation* refers also to the branching structure of the modules under consideration. Let $M = \langle AP, W, R, W_0, L, \alpha\rangle$ and $M' = \langle AP, W', R', W_0', L', \alpha'\rangle$ be fair modules. Let w and w' be states in W and W', respectively. A relation $H \subseteq W \times W'$ is a *simulation relation* from $\langle M, w\rangle$ to $\langle M', w'\rangle$ iff the following conditions hold [Mil71]:

(1) $H(w, w')$.

(2) For all t and t' with $H(t, t')$, we have $L(t) = L'(t')$.

(3) For all t and t' with $H(t, t')$ and for all $s \in W$ such that $R(t, s)$, there exists $s' \in W'$ such that $R'(t', s')$ and $H(s, s')$.

To account for fairness, it is convenient to extend H to relate also infinite computations of M and M'. For two computations $\pi = w_0, w_1, \ldots$ in M, and $\pi' = w_0', w_1', \ldots$ in M', we say that $H(\pi, \pi')$ holds iff $H(w_i, w_i')$ holds for all $i \geq 0$. For a pair $\langle w, w'\rangle \in W \times W'$, we say that $\langle w, w'\rangle$ is *good in* H iff for every fair w-computation π in M, there exists a fair w'-computation π' in M', such that $H(\pi, \pi')$. The relation H is a *fair simulation relation* from $\langle S, w\rangle$ to $\langle S', w'\rangle$ iff the following conditions hold [GL94]:

(1) $H(w, w')$.

(2) For all t and t' with $H(t, t')$, we have $L(t) = L'(t')$.

(3) For all t and t' with $H(t, t')$, the pair $\langle t, t' \rangle$ is good in H.

A (fair) simulation relation H is a *(fair) simulation from M to M'* if for every $w \in W_0$ there exists $w' \in W_0'$ such that $H(w, w')$. If there exists a (fair) simulation from M to M', we say that M *(fairly) simulates* M' and we write $M \leq M'$ (resp., $M \leq_f M'$). Intuitively, it means that the transition system S' has more behaviors than the transition system S. This gives rise to the *simulation probem* (decide if $M \leq M'$) and the *fair simulation problem* (decide if $M \leq_f M'$). It is easy to see that simulation implies containment and fair simulation implies fair containment. The opposite, however, is not true in general.

Without fairness, the tree-based approach is easier than the trace-based approach (assuming that PSPACE is different than PTIME).

Theorem 5.1

(1) [MS72, KV98] *The containment problem is PSPACE-complete.*

(2) [Mil80, BGS92, KV98] *The simulation problem is PTIME-complete.*

One we introduce fairness, however, the advantage of the tree-based approach disappears.

Theorem 5.2

(1) [Wol82] *The fair containment problem is PSPACE-complete.*

(2) [KV98] *The fair simulation problem is PSPACE-complete.*

Recall that our main concern is the complexity in terms of the (much larger) implementation.

Theorem 5.3 [KV98]

(1) *The implementation complexity of containment and fair containment is NLOGSPACE-complete.*

(2) *The implementation complexity of simulation and fair simulation is PTIME-complete*

Thus, with regard to implementation complexity, the trace-based approach is actually easier than the tree-based approach.

6 Concluding Remarks

We examine have four verification scenarios: modular verification, verification of open systems, verification of concurrent systems, and automata-theoretic verification. In each case we showed that CTL does not dominate LTL computationally. We believe that this evidence demonstrates convingly that the assertion "model checking for CTL is easy, while model checking for LTL is hard" is rather simplistic. Furthermore, as in many of these scenarios the complexity of verifying CTL* properties is markedly higher than that for LTL properties, it is quite clear that "branching is not free".

What can we conclude from these results regarding the relative merits of the linear and branching time approaches? Our conclusion is that the debate cannot be settled by reference to complexity-based arguments, since no approach dominates the other from the worst-case-computational-complexity perspective. At the same time, the debate also cannot be settled by reference to expressiveness-based arguments, since it is known that LTL and CTL are expressively incomparable [Lam80]. Thus, it seems that the debate cannot be settled by reference to *theoretical* arguments solely. Rather, it is time that *pragmatic* arguments be brought to bear on this issue. Given the growing use of model-checking tools in industry [Goe97, Kur97], the debate might end up being decided by the marketplace rather than by the research community.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[ASB+94] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A.L. Sangiovanni-Vincentelli. Equivalences for fair kripke structures. In *Proc. 21st Int. Colloquium on Automata, Languages and Programming*, Jerusalem, Israel, July 1994.

[BBG+94] 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.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proc. 4th Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 260–273, Montreal, June 1992. Springer-Verlag.

[BCM+92] 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.

[BGS92] J. Balcazar, J. Gabarro, and M. Santha. Deciding bisimilarity is P-complete. *Formal Aspects of Computing*, 4(6):638–648, 1992.

[BMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[BVW94] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. L. Dill, editor, *Computer Aided Verification, Proc. 6th Int. Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, June 1994. Springer-Verlag, Berlin.

[CD88] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *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.

[CE81] 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-Verlag, 1981.

[CES86] 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.

[CK86] E.M. Clarke and R.P. Kurshan. Computer aided verification. *IEEE Spectrum*, 33:61–67, 1986.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In R. Parikh, editor, *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, 1989.

[DDGJ89] 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.

[DGG93] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proc. 5th Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 479–490. Springer-Verlag, June 1993.

[EH85] 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.

[EH86] 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.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 368–377, White Plains, October 1988.

[EL85] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, January 1985.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, pages 997–1072, 1990.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[Goe97] R. Goering. Model checking expands verification's scope. *Electronic Engineering Today*, February 1997.

[Hen85] M. Hennessy. *Algebraic theory of Processes*. MIT Press, Cambridge, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HP85] 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.

[Jon83] 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.

[Jos89] B. Josko. Verifying the correctness of AADL modules using model chekcing. 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.

[KT90] D. Kozen and L. Tiuryn. Logics of programs. *Handbook of Theoretical Computer Science*, pages 789–840, 1990.

[Kup95] O. Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, The Technion, 1995.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[Kur97] R.P. Kurshan. Formal verification in a commercial setting. *The Verification Times*, 1997.

[KV95] O. Kupferman and M.Y. Vardi. On the complexity of branching modular model checking. In *Proc. 6th Conferance on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 408–422, Philadelphia, August 1995. Springer-Verlag.

[KV96] O. Kupferman and M.Y. Vardi. Module checking. In *Computer Aided Verification, Proc. 8th Int. Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 1996.

[KV98] O. Kupferman and M.Y. Vardi. Verification of fair transition systems. *Chicago Journal of Theoretical Computer Science*, 1998(2), March 1998.

[KVW98] O. Kupferman, M.Y. Vardi, and P. Wolper. Module checking. to appear, 1998.

[Lam80] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systenms*, 5:190–222, 1983.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LPS81] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness – the ethics of concurrent termination. In *Proc. 8th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, July 1981.

[LS84] S.S. Lam and A.U. Shankar. Protocol verification via projection. *IEEE Trans. on Software Engineering*, 10:325–342, 1984.

[LT87] N. A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[MC81] B. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Engineering*, 7:417–426, 1981.

[Mil71] 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.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[Mil89] R. Milner. *Communication and Concurrecny*. Prentice-Hall, Englewood Clifs, 1989.

[MP92a] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.

[MP92b] Z. Manna and A. Pnueli. Temporal specification and verification of reactive modules. 1992.

[MP95] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer-Verlag, New York, 1995.

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu85a] 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.

[Pnu85b] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Int. Colloquium on Automata, Languages and Programming*, pages 15–32. Lecture Notes in Computer Science, Springer-Verlag, 1985.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, 32:733–749, 1985.

[Sti87] C. Stirling. Comparing linear and branching time temporal logics. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398, pages 1–20. Lecture Notes in Computer Science, Springer-Verlag, 1987.

[Sti92] C. Stirling. Modal and temporal logics. *Handbook of Logic in Computer Science, Vol. 2*, pages 477–563, 1992.

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of Theoretical Computer Science*, pages 165–191, 1990.

[VS85] 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.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[Wol82] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications*. PhD thesis, Stanford University, 1982.