# Module Checking

Orna Kupferman[1] and Moshe Y. Vardi[2]

[1] Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
Email: ok@research.att.com
[2] Rice University, Department of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

**Abstract.** 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. The ability of temporal logics to describe an ongoing interaction of a reactive program with its environment makes them particularly appropriate for the specification of open systems. Nevertheless, model-checking algorithms used for the verification of closed systems are not appropriate for the verification of open systems. Correct model checking of open systems should check the system with respect to arbitrary environments and should take into account uncertainty regarding the environment. This is not the case with current model-checking algorithms and tools. In this paper we introduce and examine the problem of *model checking of open systems* (*module checking*, for short). We show that while module checking and model checking coincide for the linear-time paradigm, module checking is much harder than model checking for the branching-time paradigm. We prove that the problem of module checking is EXPTIME-complete for specifications in CTL and is 2EXPTIME-complete for specifications in CTL$^\star$. This bad news is also carried over when we consider the program-complexity of module checking. As good news, we show that for the commonly-used fragment of CTL (universal, possibly, and always possibly properties), current model-checking tools do work correctly, or can be easily adjusted to work correctly, with respect to both closed and open systems.

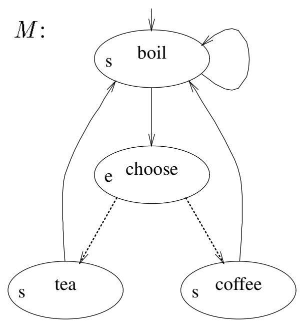
## 1 Introduction

In computer system design, we distinguish between *closed* and *open* systems [HP85]. 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. As an example to closed and open systems, we can think of 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 solely, 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.

Designing correct open systems is not an easy task. The design has to be correct with respect to any environment, and often there is much uncertainty regarding the environment [FZ88]. Therefore, in the context of open systems, formal specification and verification of the design has great importance. Traditional formalisms for specification of systems relate the initial state and the final state of a system [Flo67, Hoa69]. In 1977, Pnueli suggested *temporal logics* as a suitable formalism for reasoning about the correctness of *nonterminating systems* [Pnu77]. The breakthrough that temporal logics brought to the area of specification and verification arises from their ability to describe an *ongoing interaction* of a *reactive module* with its environment [HP85]. This ability makes temporal logics particularly appropriate for the specification of open systems.

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 the interaction of the system with its environment along a single computation. 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 are infinite trees, and they describe the possible interactions of a system with its environment. In both paradigms, we can describe the design in some formal model, specify its required behaviour with a temporal logic formula, and check formally that the model satisfies the formula. Hence the name *model checking* for the verification methods derived from this viewpoint.

We model finite-state closed systems by *programs*. We model finite-state open systems by *reactive programs* (*modules*, for short). A module is simply a program 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[3]. Consider the module $M$ presented on the right. 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 program $P$, then it models the first drink-dispensing machine described above.




To see the difference between the semantics of programs and 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 yes. Is it always possible for the second machine to eventually serve tea? Here, the answer is no. 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 be the answer? Unfortunately, model-checking tools do not distinguish between closed and open systems. They regard $M$ as a program and answer yes.

As discussed in [MP92], 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.

[3] A similar way for modelling open systems is suggested in [LT88, Lar89]. There, Larsen and Thomsen use Modal Transition Systems, where some of the transitions are *admissible* and some are *necessary*, in order to specify processes loosely, allowing a refinement ordering between processes.

We now specify formally the problem of *model checking of open systems* (*module checking*, for short). As with usual model checking, the problem has two inputs. A module $M$ and a temporal logic formula $\psi$. For a module $M$, let $V_M$ denote the unwinding of $M$ into an infinite tree. We say that $M$ satisfies $\psi$ iff $\psi$ 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 $\psi$ 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[4].

We examine the *complexity* of the module-checking problem for linear and branching temporal logics. Recall that for the linear paradigm, the problem of module checking coincides with the problem of model checking. Hence, the known complexity results for LTL model checking remain valid. As we have seen, for the branching paradigm these problems do not coincide. We show that the problem of module checking is much harder. In fact, it is as hard as satisfiability. Thus, CTL module checking is EXPTIME-complete and CTL$^\star$ module checking is 2EXPTIME-complete, both worse than the PSPACE complexity we have for LTL. Keeping in mind that CTL model checking can be done in linear time [CES86] and CTL$^\star$ model checking can be done in polynomial space [EL85], this is really bad news. We also show that for CTL and CTL$^\star$, the program complexity of module checking (i.e., the complexity of this problem in terms of the size of the module, assuming the formula is fixed), is PTIME-complete, worse than the NLOGSPACE complexity we have for LTL. As the program complexity of model checking for both CTL and CTL$^\star$ is NLOGSPACE [BVW94], this is bad news too.

As a consolation for the branching-time paradigm, we show that from a practical point of view, our news is not *that* bad. We show that in the absence of existential quantification, module checking and model checking do coincide. Thus, $\forall$CTL module checking can be done in linear time, and its program complexity is NLOGSPACE. More consolation can be found in "possibly" and "always possibly" properties. These classes of properties are considered an advantage of the branching paradigm. While being easily specified using the CTL formulas $EF\xi$ and $AGEF\xi$, these properties can not be specified in LTL [EH86]. We show that module checking of the formulas $EF\xi$ and $AGEF\xi$ can be done in linear time (though the problems are PTIME-complete).

## 2 Preliminaries

The logic *CTL*$^\star$ combines both branching-time and linear-time operators. Formulas of CTL$^\star$ are defined with respect to a set $AP$ of atomic propositions. A *path quantifier*, either $E$ ("for some path") or $A$ ("for all paths"), can prefix a *path formula* composed of an arbitrary combination of the linear-time operators $F$ ("eventually"), $G$ ("always"), $X$ ("next time"), and $U$ ("until").

The semantics of CTL$^\star$ is defined with respect to a *program* $P = \langle AP, W, R, w_0, L\rangle$, where $AP$ is the set of atomic propositions, $W$ is a set of states, $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $R(w, w')$), $w_0$ is an initial state, and $L : W \to 2^{AP}$ maps each state to a set of atomic propositions true in this state. A *path* of $P$ is an infinite sequence $w_0, w_1, \ldots$ of states such that for every $i \geq 0$, we have $R(w_i, w_{i+1})$. The notation

[4] Readers familiar with game theory can view module checking as solving an *infinite game* between the system and the environment. A correct system is then one that has a winning strategy in this game.

$P \models \varphi$ indicates that the formula $\varphi$ holds at state $w_0$ of the program $P$. A formal definition of the relation $\models$ can be found in [Eme90].

The logic *CTL* is a restricted subset of CTL$^\star$ in which the temporal operators must be immediately preceded by a path quantifier. Thus, for example, the CTL$^\star$ formula $\varphi = AGF(p \wedge EXq)$ is not a CTL formula. Adding a path quantifier, say $A$, before the $F$ temporal operator in $\varphi$ results in the formula $AGAF(p \wedge EXq)$, which is a CTL formula. The logics $\forall$*CTL* and $\forall$*CTL*$^\star$, known as the *universal fragments* of CTL and CTL$^\star$, respectively, allow only universal quantification of path formulas. Thus, all the occurrences of the path quantifier $E$ should be under an odd number of negations. The formula $\varphi$ above is therefore not a $\forall$CTL$^\star$ formula. Changing the path quantifier $E$ in $\varphi$ to the path quantifier $A$ results in the formula $AGF(p \wedge AXq)$, which is a $\forall$CTL$^\star$ formula. The logic *LTL* is a linear temporal logic. Its syntax does not allow any path quantification. Formulas of LTL are interpreted over paths in a program. The notation $P \models \psi$ indicates that the LTL formula $\psi$ holds in all the paths of the program $P$.

A *closed system* is a system whose behavior is completely determined by the state of the system. We model a closed system by a program. An *open system* is a system that interacts with its environment and whose behavior depends on that interaction. We model an open system by a *module* $M = \langle AP, W_s, W_e, R, w_0, L\rangle$, where $AP$, $R$, $w_0$, and $L$ are as in programs, $W_s$ is a set of *system states*, $W_e$ is a set of *environment states*, and we often 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 $w_e$. Whenever a module is in the state $w_s$, all the states in $succ(w_s)$ are possible next states. In contrast, when the module is in state $w_e$, there is no certainty with respect to the environment transitions and not all the states in $succ(w_e)$ are 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 $\epsilon$ is the *root* of $T$. Given an alphabet $\Sigma$, a *$\Sigma$-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 $\Sigma$. A 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. We will sometimes view the trees in $exec(M)$ as $2^{AP}$-labeled trees, taking the label of a node $x$ to be $L(V(x))$. Which interpretation is intended will be clear from the context.

Given a module $M$ and a CTL$^\star$ formula $\psi$, we say that $M$ satisfies $\psi$, denoted $M \models_r \psi$, if all the trees in $exec(M)$ satisfy $\psi$. The problem of deciding whether $M$ satisfies $\psi$ is called *module checking*[5].

[5] A different problem where a specification is checked to be correct with respect to any environment is discussed in [ASSSV94]. There, all the states of the module are system states, and the formula should hold in all compositions that contain the module as a component.

We use $M \models \psi$ to indicate that when we regard $M$ as a program (thus refer to all its states as system states), then $M$ satisfies $\psi$. The problem of deciding whether $M \models \psi$ is the usual model-checking problem [CE81, QS81]. Let $A \mapsto B$ denote that $A$ implies $B$ but $B$ does not necessarily imply $A$. It is easy to see that

$$M \models_r \psi \mapsto M \models \psi \mapsto M \not\models_r \neg\psi.$$

Indeed, $M \models_r \psi$ requires all the trees in $exec(M)$ to satisfy $\psi$. On the other hand, $M \models \psi$ means that the tree $\langle T_M, V_M \rangle$ satisfies $\psi$. Finally, $M \not\models_r \neg\psi$ only tells us that there exists some tree in $exec(M)$ that satisfies $\psi$.

We can define module checking also with respect to linear-time specifications. We say that a module $M$ satisfies an LTL formula $\psi$ iff $M \models_r A\psi$.

## 3 Module Checking for Branching Temporal Logics

We have already seen that for branching temporal logics, the model checking problem and the module checking problem do not coincide. In this section we study the complexity of CTL and CTL$^\star$ module checking. We show that not only the problems do not coincide but also their complexities do not coincide, and in a very significant manner.

**Theorem 1.**

**(1)** *The module-checking problem for CTL is EXPTIME-complete.*
**(2)** *The module-checking problem for CTL$^\star$ is 2EXPTIME-complete.*

**Proof (sketch):** We start with the upper bounds. Consider a CTL formula $\psi$ and a set $\mathcal{D} \subset \mathbb{N}$ with a maximal element $k$. Let $\mathcal{A}_{\mathcal{D},\neg\psi}$ be a Büchi tree automaton that accepts exactly all the tree models of $\neg\psi$ with branching degrees in $\mathcal{D}$. By [VW86b], such $\mathcal{A}_{\mathcal{D},\neg\psi}$ of size $O(2^{k\cdot|\psi|})$ exists.

Given a module $M = \langle AP, W_s, W_e, R, w_0, L \rangle$, we define a Büchi tree automaton $\mathcal{A}_M$ that accepts the set of all trees in $exec(M)$. Intuitively, $\mathcal{A}_M$ guesses which subtrees of $\langle T_M, V_M \rangle$ are pruned. Formally, $\mathcal{A}_M = \langle 2^{AP}, \mathcal{D}, W, \delta, w_0, W \rangle$ where $\mathcal{D}$ and $\delta$ are as follows.

- $\mathcal{D} = \bigcup_{w \in W_s} \{|succ(w)|\} \cup \bigcup_{w \in W_e} \{1, \ldots, |succ(w)|\}$.
- For every $w \in W$, $\sigma \in 2^{AP}$, and $d \in \mathcal{D}$, we have $\langle w_1, \ldots, w_d \rangle \in \delta(w, \sigma, d)$ iff $L(w) = \sigma$ and $\{w_1, \ldots, w_d\} \in step(w)$.

Since the acceptance condition only requires $\mathcal{A}_M$ not to get stuck (note that $\delta$ is partial), it is easy to see that $\mathcal{L}(\mathcal{A}_M) = exec(M)$. Since for every environment state $w$, the set $step(w)$ considers all possible subsets of $succ(w)$, the size of $\mathcal{A}_M$ is exponential in $\max_{w \in W_e}\{|succ(w)|\}$, thus exponential in the size of $M$.

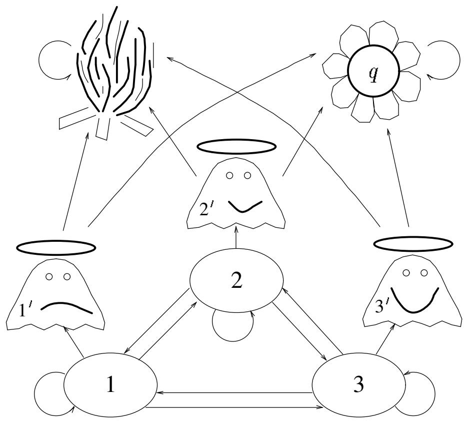
By the definition of satisfaction, we have that $M \models_r \psi$ iff all the trees in $exec(M)$ satisfy $\psi$. In other words, if no tree in $exec(M)$ satisfies $\neg\psi$. This can be checked by testing $\mathcal{L}(\mathcal{A}_M) \cap \mathcal{L}(\mathcal{A}_{\mathcal{D},\neg\psi})$ for emptiness. Equivalently, we have to test $\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi})$ for emptiness. By [VW86b], the nonemptiness problem of Büchi tree automata can be solved in quadratic time, which gives us an algorithm of time complexity $2^{O(|M|+k\cdot|\psi|)}$. We can, however, do better. By [VW86a], the number of states in the automaton $\mathcal{A}_{\mathcal{D},\neg\psi}$ is $2^{O(|\psi|)}$ and is independent of $k$. Also, the automaton $\mathcal{A}_M$ has the same number of states as $M$. The fact that the sizes of these automata are exponential in $k$ and $M$ originates from a special structure where all subsets of a certain tuple in the transition relation are possible tuples too. Therefore, the algorithm in [VW86b] can be implemented to test $\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi})$ for emptiness in time polynomial in $|M| * 2^{|\psi|}$.

The proof is similar for CTL$^\star$. Here, following [ES84, EJ88], we have that $\mathcal{A}_{\mathcal{D},\neg\psi}$ is a Rabin tree automaton with $2^{k\cdot 2^{|\psi|}}$ states and $2^{|\psi|}$ pairs. By [EJ88, PR89], and again, using the restricted structures of the automata $\mathcal{A}_{\mathcal{D},\neg\psi}$ and $\mathcal{A}_M$, checking the emptiness of $\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi})$ can then be done in time $|M|^{O(|\psi|)} * 2^{2^{O(|\psi|)}}$.

It remains to prove the lower bounds. To get an EXPTIME lower bound for CTL, we reduce CTL satisfiability, proved to be EXPTIME-complete in [FL79], to CTL module checking. Given a CTL formula $\psi$, we construct a module $M$ and a CTL formula $\varphi$ such that the size of $M$ is quadratic in the length of $\psi$, the length of $\varphi$ is linear in the length of $\psi$, and $\psi$ is satisfiable iff $M \not\models_r \neg\varphi$.

Consider a CTL formula $\psi$. For simplicity, let us first assume that $\psi$ has a single atomic proposition $q$. Let $n$ be the number of existential quantifiers in $\psi$ plus 1. By the sufficient branching-degree property of CTL, $\psi$ is satisfiable iff there exists a $\{\emptyset, \{q\}\}$-labeled tree of branching degree $n$ that satisfies $\psi$ [Eme90]. Let $P_n$ be a clique with $n$ states. By the above, $\psi$ is satisfiable iff there exists a possibility to label an unwinding of $P_n$ such that the resulted $\{\emptyset, \{q\}\}$-labeled tree satisfies $\psi$. This simple idea, due to [Kup95], is the key to our reduction. We define a module $M_n$ such that each tree in $exec(M_n)$ corresponds to a $\{\emptyset, \{q\}\}$-labeling of $\langle T_{P_n}, V_{P_n}\rangle$. We then define $\varphi$ such that there exists a tree satisfying $\varphi$ in $exec(M_n)$ iff there exists a $\{\emptyset, \{q\}\}$-labeling of $\langle T_{P_n}, V_{P_n}\rangle$ that satisfies $\psi$. It follows that $\psi$ is satisfiable iff $M \not\models_r \neg\varphi$. Let $[n] = \{1, \ldots, n\}$, $[n]' = \{1', \ldots, n'\}$, and let $M_n = \langle AP, W_s, W_e, R, w, L\rangle$, where,

- $AP = \{\textit{ghost}, q\}$.
- $W_s = [n]$.
- $W_e = [n]' \cup \{\textit{heaven}, \textit{hell}\}$.
- $R = \{\langle i,j\rangle : i,j \in [n]\} \cup \{\langle i,i'\rangle : i \in [n]\} \cup ([n]' \times \{\textit{heaven}, \textit{hell}\}) \cup \{\langle \textit{heaven}, \textit{heaven}\rangle\} \cup \{\langle \textit{hell}, \textit{hell}\rangle\}$.
- $w = 1$.
- For all $i \in [n]$, we have $L(i) = \emptyset$ and $L(i') = \{\textit{ghost}\}$. Also, $L(\textit{heaven}) = \{q\}$ and $L(\textit{hell}) = \emptyset$.



The reactive module $M_3$

That is, the system states of $M_n$ induce the clique $P_n$. In addition, each system state has a ghost: an environment state with two successors, one labeled with $q$ and one not labeled with $q$. Intuitively, the ability of the ghost $i'$ to take an environment transition to heaven in $M_n$, corresponds to the ability of a node associated with the state $i$ in $\langle T_{P_n}, V_{P_n}\rangle$ to be labeled with $q$. Thus, each tree in $exec(M_n)$ indeed corresponds to a $\{\emptyset, \{q\}\}$-labeling of $\langle T_{P_n}, V_{P_n}\rangle$. We now have to define $\varphi$ such that whenever the formula $\psi$ refers to $q$, the formula $\varphi$ will refer to $EXEXq$. Indeed, since *heaven* is the only state labeled with $q$, then a system state satisfies $EXEXq$ iff the transition of its ghost to heaven is enabled. In addition, path quantification in $\varphi$ should be restricted to computations of $P_n$. That is, to paths that never meet a ghost. To do this, we define a function $f$ : CTL$^\star$ formulas $\rightarrow$ CTL$^\star$ formulas such that $f(\xi)$ restricts path quantification to paths that never visit a state labeled with $ghost$. We define $f$ inductively as follows.

- $f(q) = q$.
- $f(\neg\xi) = \neg f(\xi)$.
- $f(\xi_1 \vee \xi_2) = f(\xi_1) \vee f(\xi_2)$.
- $f(E\xi) = E((G\neg \textit{ghost}) \wedge f(\xi))$.
- $f(A\xi) = A((F\textit{ghost}) \vee f(\xi))$.
- $f(X\xi) = Xf(\xi)$.

– $f(\xi_1 U \xi_2) = f(\xi_1) U f(\xi_2)$.

For example, $f(EqU(AFp)) = E((G\neg \mathit{ghost}) \wedge (qU(A((F\mathit{ghost}) \vee Fq))))$. We can now define $\varphi$ as $f(\psi)$ with $EXEXq$ replacing $q$. Note that we first apply $f$ and only then do the replacement. When $\psi$ is a CTL formula, the formula $f(\psi)$ is not necessarily a CTL formula. Still, it has a restricted syntax: its path formulas have either a single linear-time operator or two linear-time operators connected by a Boolean operator. By [BG94], formulas of this syntax have a linear translation to CTL.

When $\psi$ has more than one atomic proposition, the reduction is very similar. Then, for $\psi$ over $\{q_1, \ldots, q_m\}$, we have $m$ heavens, one for each atomic proposition, and we associate with each system state $m$ ghosts, again, one for each atomic propositions. We can now replace a proposition $q_i$ in $\psi$ with $EXEXq_i$ in $\varphi$. The obtained module has $n + nm + m + 1$ states and it has $n^2 + 3nm + m + 1$ transitions.

The proof is the same for CTL$^\star$. Here, we do a reduction from satisfiability of CTL$^\star$, proved to be 2EXPTIME-hard in [VS85]. □

We note that the problem of CTL module checking is EXPTIME-complete (and the one for CTL$^\star$ is 2EXPTIME-complete) even when we restrict ourselves to modules in which all states are environment states. To see this, note we could have defined $M_n$ as the clique $P_n$, adding a transition from each state to heaven. We could then force each node of a tree in $exec(M_n)$ to have as children at least its $n$ successors in $P_n$ (this can be enforced by the formula, having $[n]$ as atomic propositions, and having formulas like $AG(1 \rightarrow EX2 \wedge EX3)$ conjuncted with the original formula), and replace $q$ in $\psi$ with $EXq$ in $\varphi$. The price of using only environment states is that now the length of $\varphi$ is quadratic in the length of $\psi$.

Moreover, module checking for CTL is EXPTIME-complete even for modules of a fixed size. To see this, note that the size of $M_n$ depends on the number of atomic propositions in $\psi$ and on the minimum branching degree of models of $\psi$. Proving that the satisfiability problem for CTL is EXPTIME-hard, Fisher and Lander reduce acceptance of a word $x$ by a linear-space alternating Turing machine to satisfiability of a CTL formula $\psi_x$ [FL79]. A somewhat different reduction, which considers a fixed Turing machine that accepts an EXPTIME-complete problem, results in $\psi_x$ of length polynomial in $|x|$, but with a fixed number of atomic propositions, which, if satisfiable, has models with branching degree 2. Such $\psi_x$ induces, for all $x$, modules of a fixed size.

## 4 The Program Complexity of Module Checking

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]. In this section we consider the *program complexity* [VW86a] 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$^\star$ model checking is NLOGSPACE [VW86a, 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$^\star$, module checking is much harder than model checking. We now claim that when we consider program complexity, module checking is still harder.

**Theorem 2.** *The program complexity of CTL and CTL$^\star$ module checking is PTIME-complete.*

**Proof (sketch):** Since the algorithms given in the proof of Theorem 1 are polynomial in the size of the module, membership in PTIME is immediate.

We prove hardness in PTIME by reducing the Monotonic Circuit Value Problem (MCV), proved to be PTIME-hard in [Gol77], to module checking of the CTL formula $EFp$. In the MCV problem, we are given a monotonic Boolean circuit $\alpha$ (i.e., a circuit constructed solely of AND gates and OR gates), and a vector $\langle x_1, \ldots, x_n \rangle$ of Boolean input values. The problem is to determine whether the output of $\alpha$ on $\langle x_1, \ldots, x_n \rangle$ is 1.

Let us denote a monotonic circuit by a tuple $\alpha = \langle G_\forall, G_\exists, G_{in}, g_{out}, T \rangle$, where $G_\forall$ is the set of AND gates, $G_\exists$ is the set of OR gates, $G_{in}$ is the set of input gates (identified as $g_1, \ldots, g_n$), $g_{out} \in G_\forall \cup G_\exists \cup G_{in}$ is the output gate, and $T \subset G \times G$ denotes the acyclic dependencies in $\alpha$, that is $\langle g, g' \rangle \in T$ iff the output of gate $g'$ is an input of gate $g$.

Given a monotonic circuit $\alpha = \langle G_\forall, G_\exists, G_{in}, g_{out}, T \rangle$ and an input vector $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, we construct a module $M_{\alpha,\mathbf{x}} = \langle \{0, 1\}, G_\forall, G_\exists \cup G_{in}, R, g_{out}, L \rangle$, where

- $R = T \cup \{\langle g, g \rangle : g \in G_{in}\}$.
- For $g \in G_\forall \cup G_\exists$, we have $L(g) = 1$. For $g_i \in G_{in}$, we have $L(g_i) = x_i$.

Clearly, the size of $M_{\alpha,\mathbf{x}}$ is linear in the size of $\alpha$. Intuitively, each tree in $exec(M_{\alpha,\mathbf{x}})$ corresponds to a decision of $\alpha$ as to how to satisfy its OR gates (we satisfy an OR gate by satisfying any nonempty subset of its inputs). It is therefore easy to see that $M_{\alpha,\mathbf{x}} \models_r EF0$ iff there exists no $V \in exec(M_{\alpha,\mathbf{x}})$ such that $V \models AG1$, which holds iff the output of $\alpha$ on $x$ is 0. □

Recall that for a CTL formula $\psi$, checking that a module $M$ satisfies $\psi$ reduces to testing emptiness of the automaton $\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi}$. Checking nonemptiness of a Büchi tree automaton can be reduced to calculating a $\mu$-calculus expression of alternation depth 2 [Rab69, VW86b]. As such, it can be implemented, using symbolic methods, in tools that handle fixed-point calculations (e.g., SMV [BCM+90, McM93]).

## 5 Pragmatics

How bad is our news? In this section we show that from a pragmatic point of view, it is not *that* bad. We show that in the absence of existential quantification, module checking and model checking coincide, and that in the case where there is only a limited use of existential quantification, module checking can still be done in linear time.

### 5.1 Module Checking for Universal Temporal Logics

**Lemma 3.** *For universal branching temporal logics, the module checking problem and the model checking problem coincide.*

**Proof:** Given a module $M$ and a $\forall$CTL$^\star$ formula $\psi$, we prove that $M \models_r \psi$ iff $M \models \psi$. Assume first that $M \models_r \psi$. Then, all trees in $exec(M)$ satisfy $\psi$. Thus, in particular, $\langle T_M, V_M \rangle$ satisfies $\psi$ and $M \models \psi$. Assume now that $M \models \psi$. The relation $\{\langle w, w \rangle : w \in W\}$ is a simulation relation between any tree in $exec(M)$ and $M$. Therefore, by [GL94], all trees in $exec(M)$ satisfy $\psi$, and $M \models_r \psi$. □

Theorem 4 now follows from the known complexity results for $\forall$CTL and $\forall$CTL$^\star$ model checking [CES86, SC85, BVW94].

**Theorem 4.**

**(1)** *The module-checking problem for $\forall$CTL is in linear time.*
**(2)** *The module-checking problem for $\forall$CTL$^\star$ is PSPACE-complete.*
**(3)** *The program complexity of module checking for $\forall$CTL and $\forall$CTL$^\star$ is NLOGSPACE-complete.*

It follows from the above theorem that the module-checking problem for LTL is PSPACE-complete and its program complexity is NLOGSPACE-complete.

### 5.2 Module Checking of "Possibly" and "Always Possibly" Properties

We have seen that, for each fixed CTL formula $\psi$, checking that a module $M$ satisfies $\psi$ can be checked in time polynomial in the size of $M$. Sometimes, we can do even better. Some CTL formulas have a special structure that enables us to module-check them in time linear in the size of $M$. In this section we show that "possibly" and "always possibly" properties, by far the most popular properties specified in CTL and not specifiable in ∀CTL, induce such formulas.

Consider the CTL formula $EFsend$. The formula states that it is possible for the system to eventually send a request. We call properties of this form *possibly properties*. Consider now the CTL formula $AGEFsend$. The formula states that in all computations, it is always possible for the system to eventually send a request. We call properties of this form *always possibly properties*. It is easy to see that possibly and always possibly properties can not be specified in linear temporal logics, nor in universal branching logics [EH86].

**Theorem 5.** *Module checking of possibly and always possibly properties can be done in linear running time.*

**Proof (sketch):** We describe an efficient algorithm that module-checks these properties. For simplicity, we assume that system and environment states are labeled with atomic propositions $s$ and $e$, respectively. Consider a module $M = \langle AP, W_s, W_e, R, w_0, L \rangle$ and a propositional assertion $\xi$. By definition, $M \models_r EF\xi$ iff there exists no tree $\langle T, V \rangle \in exec(M)$ all of whose nodes satisfy $\neg\xi$. We say that a state $w \in W$ is *safe* iff such a tree $\langle T, V \rangle$ can not have $w$ as its root. We check that $M \models_r EF\xi$ by checking that $w_0$ is safe. In order to be safe, a state $w$ should satisfy one of the following:

1. $w \models \xi$,
2. $w$ is a system state that has a safe successor, or
3. $w$ is an environment state all of whose successors are safe.

Consider the monotone function $f : 2^W \to 2^W$ where $f(y) = \xi \vee (s \wedge EXy) \vee (e \wedge AXy)$. It can be shown that $w$ is safe iff $w$ is in the least fixed-point of $f$. Therefore, we have that $w$ is safe iff $w \models \mu y.\xi \vee (s \wedge EXy) \vee (e \wedge AXy)$. Hence,

$$M \models_r EF\xi \Leftrightarrow M \models \mu y.\xi \vee (s \wedge EXy) \vee (e \wedge AXy).$$

Now, $M \models_r AGEF\xi$ iff there exists no tree $\langle T, V \rangle \in exec(M)$ such that $\langle T, V \rangle$ has a subtree $\langle T', V' \rangle$ all of whose nodes satisfy $\neg\xi$. We can therefore check that $M \models_r AGEF\xi$ by checking that all the reachable states in $M$ are safe. Hence,

$$M \models_r AGEF\xi \Leftrightarrow M \models \nu z.[\mu y.\xi \vee (s \wedge EXy) \vee (e \wedge AXy)] \wedge AXz.$$

So, we reduced module checking of possibly and always possibly properties to model checking of an alternation-free $\mu$-calculus formula. As the latter can be done in linear running time [Cle93], we are done.

□

Again, as our algorithms involve at most two simple fixed-point computations, they can be easily implemented symbolically.

What about the space complexity of checking these properties? Is there a nondeterministic algorithm that can check always possibly properties in logarithmic space? As the formula we used proving Theorem 2 is $EF\xi$, the answer for possibly properties is no. Unsurprisingly, this is also the answer for the more complicated always possibly properties, as we claim in the theorem below.

**Theorem 6.** *Module checking of possibly and always possibly properties is PTIME-complete.*

**Proof (sketch):** Membership in PTIME follows from Theorem 5. To prove hardness in PTIME, we do the same reduction we did for CTL. For $EF\xi$, we need no change. For $AGEF\xi$ we do the following change. Instead a self loop, each state associated with an input gate now has a transition to the initial state $g_{out}$. Let us call the resulted module $M'_{\alpha,\mathbf{x}}$. It is easy to see that $M'_{\alpha,\mathbf{x}} \models_r AGEF0$ iff there exists no $V \in exec(M'_{\alpha,\mathbf{x}})$ such that $V \models EFAG1$, which holds iff the output of $\alpha$ on $x$ is 0. $\square$

## 6 Discussion

The discussion of the relative merits of linear versus branching temporal logics is almost as early as these paradigms [Lam80]. We mainly refer here to the linear temporal logic LTL and the branching temporal logic CTL. One of the beliefs dominating this discussion has been "while specifying is easier in LTL, model checking is easier for CTL". 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 $O(|P| * |\psi|)$ [CES86], it takes time $O(|P| * 2^{|\psi|})$ for LTL [LP85]. Since LTL model checking is PSPACE-complete [SC85], the latter bound probably cannot be improved. The attractive complexity of CTL model checking have compensated for its lack of expressive power and branching-time model-checking tools that can handle systems with more than $10^{120}$ states [Bro86, McM93, CGL93] are incorporated into industrial development of new designs [BBG$^+$94].

If we examine the history of this discussion more closely, we found that things are not that simple. On the one hand, the inability of LTL to quantify computations existentially is considered by many a serious drawback. In addition, the introduction of fair-CTL [CES86] and of many other extensions to CTL [Lon93, BBG$^+$94, BG94], have made CTL a basis for specification languages that maintain the efficiency of CTL model checking and yet overcome many of its expressiveness limitations. On the other hand, the computational superiority of CTL is also not that clear. For example, comparing the complexities of CTL and LTL model checking for concurrent programs, both are in PSPACE [VW86a, BVW94]. As shown in [Var95, KV95], the advantage that CTL enjoys over LTL disappears also when the complexity of modular verification is considered.

In this work we questioned the computational superiority of the branching-time paradigm further. We showed that when reasoning about open systems, the complexity of CTL model checking is actually higher than that of LTL. Our results are summarized in the table below. All the complexities in the table denote tight bounds.

**Acknowledgments**. We are grateful to Martin Abadi and Pierre Wolper for fruitful discussions on the verification of reactive systems.

## References

[ASSSV94] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional CTL model checking. In *Proc. 6th Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 324–337, Stanford, CA, June 1994. Springer-Verlag.

[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 Workshop on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193, Stanford, June 1994.

| | model checking | module checking | program complexity of model checking | program complexity of module checking | satisfiability |
|---|---|---|---|---|---|
| LTL | PSPACE [SC85] | PSPACE | NLOGSPACE [VW86b] | NLOGSPACE | PSPACE [SC85] |
| CTL | linear-time [CES86] | EXPTIME | NLOGSPACE [BVW94] | PTIME | EXPTIME [FL79] |
| CTL$^\star$ | PSPACE [EL85] | 2EXPTIME | NLOGSPACE [BVW94] | PTIME | 2EXPTIME [EJ88, VS85] |
| $\forall$CTL | linear-time [CES86] | linear-time | NLOGSPACE [BVW94] | NLOGSPACE | PSPACE [KV95] |
| $EF\xi$ $AGEF\xi$ | linear-time [CES86] | linear-time | NLOGSPACE [BVW94] | PTIME | NPTIME [GJ79] |


[BCM$^+$90] 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 *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[BG94] O. Bernholtz and O. Grumberg. Buy one, get one free !!! In *Proceedings of the First International Conference on Temporal Logic*, volume 827 of *Lecture Notes in Artificial Intelligence*, pages 210–224, Bonn, July 1994. Springer-Verlag.

[Bro86] M.C. Browne. An improved algorithm for the automatic verification of finite state systems using temporal logic. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 260–266, Cambridge, June 1986.

[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.

[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.

[CGL93] 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)*, Lecture Notes in Computer Science, pages 124–175. Springer-Verlag, 1993.

[Cle93] R. Cleaveland. A linear-time model-checking algorithm for the alternation-free modal $\mu$-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[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 *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, October 1988.

[EL85] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, Hawaii, 1985.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of theoretical computer science*, pages 997–1072, 1990.

[ES84] E.A. Emerson and A. P. Sistla. Deciding branching time logic. In *Proceedings of the 16th ACM Symposium on Theory of Computing*, Washington, April 1984.

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

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proceedings Symposium on Applied Mathematics*, volume 19, 1967.

[FZ88] M.J. Fischer and L.D. Zuck. Reasoning about uncertainty in fault-tolerant distributed systems. In M. Joseph, editor, *Proc. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 142–158. Springer-Verlag, 1988.

[GJ79] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. Freeman and Co., San Francisco, 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.

[Gol77] L.M. Goldschlager. The monotone and planar circuit value problems are log space complete for p. *SIGACT News*, 9(2):25–29, 1977.

[Hoa69] C.A.R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[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.

[Kup95] O. Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. In *Computer Aided Verification, Proc. 7th Int. Workshop*, pages 325–338, Liege, July 1995.

[KV95] O. Kupferman and M.Y. Vardi. On the complexity of branching modular model checking. In *Proc. 6th Conferance on Concurrency Theory*, pages 408–422, Philadelphia, August 1995.

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

[Lar89] K.G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407, pages 232–246, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[Lon93] D.E. Long. *Model checking, abstraction and compositional verification*. PhD thesis, Carnegie-Mellon University, Pittsburgh, 1993.

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

[LT88] K.G. Larsen and G.B. Thomsen. A modal process logic. In *Proceedings of the 3th Symposium on Logic in Computer Science*, Edinburgh, 1988.

[McM93] K.L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[MP92] Z. Manna and A. Pnueli. Temporal specification and verification of reactive modules. 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, Austin, Januery 1989.

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

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

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

[Var95] M.Y. Vardi. On the complexity of modular model checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, June 1995.

[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.

[VW86a] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June

1986.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.