Module Checking

Orna Kupferman
UC Berkeley [*]

Moshe Y. Vardi
Rice University[†]

Pierre Wolper
Université de Liège[‡]

February 22, 1998

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*. 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.

[*]Address: EECS Department, Berkeley, CA 94720-1770, U.S.A. Email: `orna@ic.eecs.berkeley.edu`, URL: `http://www.eecs.berkeley.edu/~orna`. Supported in part by ONR YIP award N00014-95-1-0520, by NSF CAREER award CCR-9501708, by NSF grant CCR-9504469, by AFOSR contract F49620-93-1-0056, by ARO MURI grant DAAH-04-96-1-0341, by ARPA grant NAG2-892, and by SRC contract 95-DC-324.036.

[†]Address: Department of Computer Science, 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.

[‡]Address: Institut Montefiore, B28; B-4000 Liège Sart-Tilman; Belgium. Email: `pw@montefiore.ulg.ac.be`, URL: `http://www.montefiore.ulg.ac.be/~pw`.

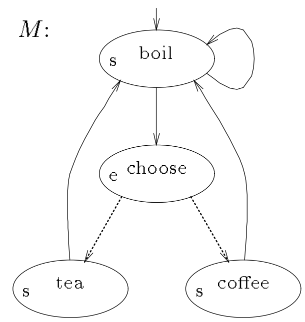
1 Introduction

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 cannot 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 behavior 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[1]. 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 cannot 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 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 be the answer? Unfortunately, existing model-checking tools do not distinguish between closed and open systems. They regard M as a program and answer positively.

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.

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 ψ. For a 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

[1]A similar way for modeling 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.

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 likes both coffee and tea induce trees in which the first *choose* node has two successors[2]. Note that the way we have defined module checking implies that we view the path quantifiers of the branching formula as ranging over the internal choices of the module, whereas the external choices are implicitly universally quantified. We believe that, in many circumstances, this is the natural choice and we justify it further in Section 6. One can generalize from this point of view and consider that the path quantifiers of the logic can be interpreted both over internal and external choices. This then implies that the logic should be extended in order to explicitly specify over which choices path quantifiers should be interpreted. This idea is investigated in [AHK97].

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* 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* model checking can be done in polynomial space [EL85], this is really bad news. We also show that for CTL and CTL*, 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* 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 cannot 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

[2]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.

are PTIME-complete).

2 Preliminaries

The logic *CTL** is a branching temporal logic. A path quantifier, E ("for some path") or A ("for all paths"), can prefix an assertion composed of an arbitrary combination of linear time operators. There are two types of formulas in CTL*: *state formulas*, whose satisfaction is related to a specific state, and *path formulas*, whose satisfaction is related to a specific path. Formally, let AP be a set of atomic proposition names. A CTL* state formula is either:

- **true**, **false**, or p, for $p \in AP$.
- $\neg\varphi$, $\varphi \vee \psi$, or $\varphi \wedge \psi$ where φ and ψ are CTL* state formulas.
- $E\varphi$ or $A\varphi$, where φ is a CTL* path formula.

A CTL* path formula is either:

- A CTL* state formula.
- $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $G\varphi$, $F\varphi$, $X\varphi$, or $\varphi U \psi$, where φ and ψ are CTL* path formulas.

The logic CTL* consists of the set of state formulas generated by the above rules.

The logic *CTL* is a restricted subset of CTL*. In CTL, the temporal operators G, F, X, and U must be immediately preceded by a path quantifier. Formally, it is the subset of CTL* obtained by restricting the path formulas to be $G\varphi$, $F\varphi$, $X\varphi$, $\varphi U \psi$, where φ and ψ are CTL state formulas. Thus, for example, the CTL* formula $\varphi = AGF(p \wedge EXq)$ is not a CTL formula. Adding a path quantifier, say A, before the F temporal operator in φ results in the formula $AGAF(p \wedge EXq)$, which is a CTL formula. The logic $\forall CTL^\star$ is a restricted subset of CTL* that allows only universal path quantification. Thus, it allows only the path quantifier A, which must always be in the scope of an even number of negations. Note that assertions of the form $\neg A\psi$, which is equivalent to $E\neg\psi$, are not possible. Thus, the logic $\forall$CTL* is not closed under negation. The formula φ above is not a $\forall$CTL* formula. Changing the path quantifier E in φ to the path quantifier A results in the formula $AGF(p \wedge AXq)$, which is a $\forall$CTL* formula. The logic $\forall$CTL is defined similarly, as the restricted subset of CTL that allows only universal path quantification. The logics $\exists$CTL* and $\exists$CTL are defined analogously, as the existential fragments of CTL* and CTL, respectively. Note that negating a $\forall$CTL* formula results in an $\exists$CTL* formula. The logic *LTL* is a linear temporal logic. Its syntax does not allow any path quantification.

The semantics of the logic CTL* (and its sub-logics) 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 $L : W \rightarrow 2^{AP}$ maps each state to a set of atomic propositions true in this state. For w and w' with $R(w, w')$, we say that w' is a successor of w and we use $bd(w)$ to denote the number of successors that w has. A *path* of P is an infinite sequence $\pi = w^0, w^1, \ldots$ of states such that for every $i \geq 0$, we have $R(w^i, w^{i+1})$. The suffix $w^i, w^{i+1}, \ldots$ of π is denoted by π^i. We use $w \models \varphi$ to indicate that a state formula φ holds at state w, and we use $\pi \models \varphi$ to indicate that a path formula φ holds at path π (assuming an agreed program P). The relation $\models$ is inductively defined as follows.

- For all w, we have that $w \models \mathbf{true}$ and $w \not\models \mathbf{false}$.
- For an atomic proposition $p \in AP$, we have $w \models p$ iff $p \in L(w)$
- $w \models \neg\varphi$ iff $w \not\models \varphi$.
- $w \models \varphi \vee \psi$ iff $w \models \varphi$ or $w \models \psi$.
- $w \models E\varphi$ iff there exists a path $\pi = w_0, w_1, \ldots$ such that $w_0 = w$ and $\pi \models \varphi$.
- $\pi \models \varphi$ for a state formula φ iff $w^0 \models \varphi$.
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$.
- $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ or $w \models \psi$.
- $\pi \models X\varphi$ iff $\pi^1 \models \varphi$.
- $\pi \models \varphi U \psi$ iff there exists $j \geq 0$ such that $\pi^j \models \psi$ and for all $0 \leq i < j$, we have $\pi^i \models \varphi$.

The semantics above considers the Boolean operators $\neg$ ("negation") and $\vee$ ("or"), the temporal operators X ("next") and U ("until"), and the path quantifier A. The other operators are superfluous and can be viewed as the following abbreviations.

- $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$ ("and").
- $F\varphi = \mathbf{true} U \varphi$ ("eventually").
- $G\varphi = \neg F \neg\varphi$ ("always").
- $A\varphi = \neg E \neg\varphi$ ("for all paths").

Formulas of LTL are interpreted over paths in a program. The notation $P \models \varphi$ indicates that the LTL formula φ 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 $W_s \cup W_e$.

We assume that the states in M are ordered. For each state $w \in W$, let $succ(w)$ be an ordered tuple of w's R-successors; i.e., $succ(w) = \langle w_1, \ldots, w_{bd(w)} \rangle$, where for all $1 \leq i \leq bd(w)$, we have $R(w, w_i)$, and the w_i's are ordered. 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, since we consider environments that cannot block the system, is that not all the transitions from w_e are disabled. For a state $w \in W$, let $step(w)$ denote the set of the possible (ordered) 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 sub-tuples of $succ(w_e)$.

For $k \in \mathbb{N}$, let $[k]$ denote the set $\{1, 2, \ldots, k\}$. An *infinite tree* with branching degrees bounded by k is a nonempty set $T \subseteq [k]^*$ such that if $x \cdot c \in T$ where $x \in [k]^*$ and $c \in [k]$, then also $x \in T$, and for all $1 \leq c' < c$, we have that $x \cdot c' \in T$. In addition, if $x \in T$, then $x \cdot 1 \in T$. The elements of T are called *nodes*, and the empty word ε is the *root* of T. For every node $x \in T$, we denote by $d(x)$ the branching degree of x; that is, the number of $c \in [k]$ for which $x \cdot c$ in T. A *path* of T is a set $\pi \subseteq T$ such that $\varepsilon \in T$ and for all $x \in \pi$, there exists a single $c \in [k]$ such that $x \cdot c \in \pi$. 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. A module M can be unwound into an infinite tree $\langle T_M, V_M \rangle$ in a straightforward way[3]. When we examine a specification with respect to M, the specification should hold not only in $\langle T_M, V_M \rangle$ (which corresponds to a very specific environment that never restricts 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:

- $V(\varepsilon) = w_0$.
- For all $x \in T$ with $V(x) = w$, there exists $\langle w_1, \ldots, w_n \rangle \in step(w)$ such that $T \cap (\{x\} \times \mathbb{N}) = \{x \cdot 1, x \cdot 2, \ldots, x \cdot n\}$ and for all $1 \leq c \leq n$ we have $V(x \cdot c) = w_c$.

Intuitively, each tree in $exec(M)$ corresponds to a different behavior of the environment. 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* formula ψ, we say that M satisfies ψ, denoted $M \models_r \psi$, if all the trees in $exec(M)$ satisfy ψ. The problem of deciding whether M satisfies ψ is called

[3] Since we assume that the states of M are ordered, there is indeed only a single such tree. Since temporal-logic formulas cannot distinguish between trees obtained by different orders, we do not lose generality by considering a particular order.

module checking[4]. 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 ψ. The problem of deciding whether $M \models \psi$ is the usual model-checking problem [CE81, QS81]. It is easy to see that while $M \models_r \psi$ implies that $M \models \psi$, the other direction is not necessarily true. Also, while $M \models \psi$ implies that $M \not\models_r \neg\psi$, the other direction is not true as well. Indeed, $M \models_r \psi$ 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, $M \not\models_r \neg\psi$ 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 $M \models_r A\psi$.

In order to solve the module-checking problem, we are going to use *nondeterministic tree automata*. Tree automata run on Σ-labeled trees. A Büchi tree automaton is $\mathcal{A} = \langle \Sigma, \mathcal{D}, Q, q_0, \delta, F \rangle$, where Σ is an alphabet, $\mathcal{D}$ is a finite set of branching degrees (positive integers), Q is a set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \times \mathcal{D} \to 2^{Q^*}$ is a transition function satisfying $\delta(q, \sigma, d) \in Q^d$, for every $q \in Q$, $\sigma \in \Sigma$, and $d \in \mathcal{D}$, and $F \subseteq Q$ is an acceptance condition.

A run of $\mathcal{A}$ on an input Σ-labeled tree $\langle T, V \rangle$ with branching degrees in $\mathcal{D}$ is a Q-labeled tree $\langle T, r \rangle$ such that $r(\varepsilon) = q_0$ and for every $x \in T$, we have that $\langle r(x \cdot 1), r(x \cdot 2), \ldots, r(x \cdot d) \rangle \in \delta(r(x), V(x), d(x))$. If, for instance, $r(1 \cdot 1) = q$, $V(1 \cdot 1) = \sigma$, $d(1 \cdot 1) = 2$, and $\delta(q, \sigma, 2) = \{\langle q_1, q_2 \rangle, \langle q_4, q_5 \rangle\}$, then either $r(1 \cdot 1 \cdot 1) = q_1$ and $r(1 \cdot 1 \cdot 2) = q_2$, or $r(1 \cdot 1 \cdot 1) = q_4$ and $r(1 \cdot 1 \cdot 2) = q_5$. Given a run $\langle T, r \rangle$ and a path $\pi \subseteq T$, we define

$$Inf(r|\pi) = \{q \in Q \ : \ \text{for infinitely many } x \in \pi, \text{ we have } r(x) = q\}.$$

That is, $Inf(r|\pi)$ is the set of states that r visits infinitely often along π. A run $\langle T, r \rangle$ is *accepting* iff for all paths $\pi \subseteq T$, we have $Inf(r|\pi) \cap F \neq \emptyset$. Namely, along all the paths of T, the run visits states from F infinitely often. An automaton $\mathcal{A}$ accepts $\langle T, V \rangle$ iff there exists an accepting run $\langle T, r \rangle$ of $\mathcal{A}$ on $\langle T, V \rangle$. We use $\mathcal{L}(\mathcal{A})$ to denote the language of the automaton $\mathcal{A}$; i.e., the set of all trees accepted by $\mathcal{A}$. In addition to Büchi tree automata, we also refer to Rabin tree automata. There, $F \subseteq 2^Q \times 2^Q$, and a run is accepting iff for every path $\pi \subseteq T$, there exists a pair $\langle G, B \rangle \in F$ such that $Inf(r|\pi) \cap G \neq \emptyset$ and $Inf(r|\pi) \cap B = \emptyset$.

The *size* of an automaton $\mathcal{A}$, denoted $|\mathcal{A}|$, is defined as $|Q| + |\delta| + |F|$, where $|\delta|$ is the sum of the lengths of tuples that appear in the transitions in δ, and $|F|$ is the sum of the sizes of the sets appearing in F (a single set in the case $\mathcal{A}$ is a Büchi automaton, and $2m$ sets in the case $\mathcal{A}$ is a Rabin automaton with m pairs). Note that $|\mathcal{A}|$ is independent of the sizes of Σ and $\mathcal{D}$. Note also that $\mathcal{A}$ can be stored in space $O(|\mathcal{A}|)$.

[4] A different problem where a specification is checked to be correct with respect to any environment is discussed in [ASSS94]. There, the formula may refer to atomic propositions not in the module and it should hold in all compositions that contain the module as a component.

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* module checking. We show that the difference between the model-checking and the module-checking problems reflects in their complexities, and in a very significant manner.

Theorem 3.1

(1) *The module-checking problem for CTL is EXPTIME-complete.*

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

Proof: We start with the upper bounds. Given M and ψ, we define two tree automata. Essentially, the first automaton accepts the set of trees in $exec(M)$ and the second automaton accepts the set of trees that do not satisfy ψ. Thus, $M \models_r \psi$ iff the intersection of the automata is empty.

Recall that each tree in $exec(M)$ is obtained from $\langle T_M, V_M\rangle$ by pruning some of its subtrees. The tree $\langle T_M, V_M\rangle$ is a 2^{AP}-labeled tree. We can think of a tree $\langle T, V\rangle \in exec(M)$ as the $(2^{AP} \cup \{\perp\})$-labeled tree obtained from $\langle T_M, V_M\rangle$ by replacing the labels of nodes pruned in $\langle T, V\rangle$ by $\perp$. Doing so, all the trees in $exec(M)$ have the same shape (they all coincide with T_M), and they differ only in their labeling. Accordingly, we can think of an environment to $\langle T_M, V_M\rangle$ as a strategy for placing $\perp$'s in $\langle T_M, V_M\rangle$: placing a $\perp$ in a certain node corresponds to the environment disabling the transition to that node. Since we consider environments that do not "block" the system, at least one successor of each node is not labeled with $\perp$. Also, once the environment places a $\perp$ in a certain node x, it should keep placing $\perp$'s in all the nodes of the subtree that has x as its root. Indeed, all the nodes to this subtree are disabled. The first automaton, $\mathcal{A}_M$, accepts all the $(2^{AP} \cup \{\perp\})$-labeled tree obtained from $\langle T_M, V_M\rangle$ by such a "legal" placement of $\perp$'s. Formally, given a module $M = \langle AP, W_s, W_e, R, w_0, L\rangle$, we define $\mathcal{A}_M = \langle 2^{AP} \cup \{\perp\}, \mathcal{D}, Q, q_0, \delta, Q\rangle$, where

- $\mathcal{D} = \bigcup_{w \in W}\{bd(w)\}$. That is, $\mathcal{D}$ contains all the branching degrees in M (and hence also all branching degrees in T_M).
- $Q = W \times \{\top, \vdash, \perp\})$. Thus, every state w of M induces three states $\langle w, \top\rangle$, $\langle w, \vdash\rangle$, and $\langle w, \perp\rangle$ in $\mathcal{A}_M$. Intuitively, when $\mathcal{A}_M$ is in state $\langle w, \perp\rangle$, it can read only the letter $\perp$. When $\mathcal{A}_M$ is in state $\langle w, \top\rangle$, it can read only letters in 2^{AP}. Finally, when $\mathcal{A}_M$ is in state $\langle w, \vdash\rangle$, it can read both letters in 2^{AP} and the letter $\perp$. Thus, while a state $\langle w, \vdash\rangle$ leaves it for the environment to decide whether the transition to w is enabled, a state $\langle w, \top\rangle$ requires the environment to enable the transition to w, and a state $\langle w, \perp\rangle$ requires the environment to disable the transition to w. The three types of states help us to make sure that the environment enables all transitions from system states, enables at least

one transition from each environment state, and disables transitions from states that the transition to them have already been disabled.

- $q_0 = \langle w_0, \top \rangle$.
- The transition function $\delta : Q \times \Sigma \times \mathcal{D} \rightarrow 2^{Q^*}$ is defined for $w \in W$ and $k = bd(w)$ as follows. Let $succ(w) = \langle w_1, \ldots, w_k \rangle$.
  - For $w \in W_s \cup W_e$ and $m \in \{\vdash, \perp\}$, we have

$$\delta(\langle w, m \rangle, \perp, k) = \langle \langle w_1, \perp \rangle, \langle w_2, \perp \rangle, \ldots, \langle w_k, \perp \rangle \rangle.$$

  - For $w \in W_s$ and $m \in \{\top, \vdash\}$, we have

$$\delta(\langle w, m \rangle, L(w), k) = \langle \langle w_1, \top \rangle, \langle w_2, \top \rangle, \ldots, \langle w_k, \top \rangle \rangle.$$

  - For $w \in W_e$ and $m \in \{\top, \vdash\}$, we have

$$\delta(\langle w, m \rangle, L(w), k) = \{ \begin{array}{l} \langle \langle w_1, \top \rangle, \langle w_2, \vdash \rangle, \ldots, \langle w_k, \vdash \rangle \rangle, \\ \langle \langle w_1, \vdash \rangle, \langle w_2, \top \rangle, \ldots, \langle w_k, \vdash \rangle \rangle, \\ \vdots \\ \langle \langle w_1, \vdash \rangle, \langle w_2, \vdash \rangle, \ldots, \langle w_k, \top \rangle \rangle \ \}. \end{array}$$

    That is, $\delta(\langle w, m \rangle, L(w), k)$ contains k k-tuples. When the automaton proceeds according to the ith tuple, the environment can disable the transitions to all w's successors, except the transition to w_i, which must be enabled.

  Note that δ is not defined in the case where $k \neq bd(w)$ or when the input does not meet the restriction imposed by the $\top, \vdash$, and $\perp$ annotations, or by the labeling of w.

Let k be the maximal branching degree in M. It is easy to see that $|Q| \leq 3 \cdot |W|$ and $|\delta| \leq k \cdot |R|$. Thus, assuming that $|W| \leq |R|$, the size of $\mathcal{A}_M$ is bounded by $O(k \cdot |R|)$.

Recall that a node of $\langle T, V \rangle \in \mathcal{L}(\mathcal{A}_M)$ that is labeled $\perp$ stands for a node that actually does not exist in the corresponding pruning of $\langle T_M, V_M \rangle$. Accordingly, if we interpret CTL* formulas over the trees obtained by pruning subtrees of $\langle T_M, V_M \rangle$ by means of the trees recognized by $\mathcal{A}_M$, we should treat a node that is labeled by $\perp$ as a node that does not exist. To do this, we define a function f : CTL* formulas $\rightarrow$ CTL* formulas such that $f(\xi)$ restricts path quantification to paths that never visit a state labeled with $\perp$. We define f inductively as follows.

- $f(q) = q$.
- $f(\neg \xi) = \neg f(\xi)$.
- $f(\xi_1 \vee \xi_2) = f(\xi_1) \vee f(\xi_2)$.

- $f(E\xi) = E((G\neg\bot) \wedge f(\xi))$.
- $f(A\xi) = A((F\bot) \vee f(\xi))$.
- $f(X\xi) = Xf(\xi)$.
- $f(\xi_1 U \xi_2) = f(\xi_1) U f(\xi_2)$.

For example, $f(EqU(AFp)) = E((G\neg\bot)\wedge(qU(A((F\bot)\vee Fq))))$. When ψ 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 [KG96], formulas of this syntax have a linear translation to CTL.

Given ψ, let $\mathcal{A}_{\mathcal{D},\neg\psi}$ be a Büchi tree automaton that accepts exactly all the tree models of $f(\neg\psi)$ with branching degrees in $\mathcal{D}$. By [VW86b], such $\mathcal{A}_{\mathcal{D},\neg\psi}$ of size $2^{k\cdot O(|\psi|)}$ exists.

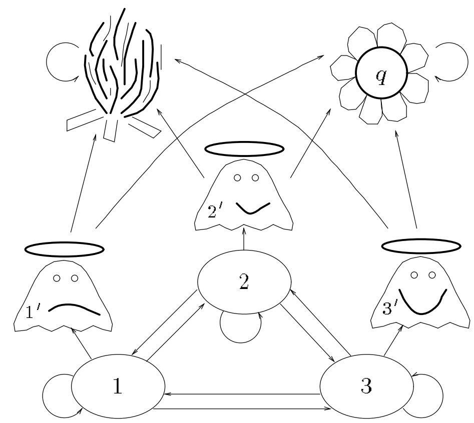
By the definition of satisfaction, we have that $M \models_r \psi$ iff all the trees in $exec(M)$ satisfy ψ. In other words, if no tree in $exec(M)$ satisfies $\neg\psi$. Recall that the automaton $\mathcal{A}_M$ accepts a $(2^{AP} \cup \{\bot\})$-labeled tree iff it corresponds to a "legal" pruning of $\langle T_M, V_M\rangle$ by the environment, with a pruned node being labeled by $\bot$. Also, the automaton $\mathcal{A}_{\mathcal{D},\neg\psi}$ accepts a $(2^{AP} \cup \{\bot\})$-labeled tree iff it does not satisfy ψ, with path quantification ranging only over paths that never meet a node labeled with $\bot$. Hence, checking whether $M \models_r \psi$ can be reduced to 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 $O(|R|^2 \cdot 2^{k\cdot O(|\psi|)})$.

The proof is similar for CTL*. Here, following [ES84, EJ88], we have that $\mathcal{A}_{\mathcal{D},\neg\psi}$ is a Rabin tree automaton with $2^{k\cdot 2^{O(|\psi|)}}$ states and $2^{O(|\psi|)}$ pairs. By [EJ88, PR89a], checking the emptiness of $\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\mathcal{D},\neg\psi})$ can then be done in time $(k\cdot|R|)^{2^{O(|\psi|)}} \cdot 2^{k\cdot 2^{O(|\psi|)}}$.

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, Pra80], to CTL module checking. Given a CTL formula ψ, we construct a module M and a CTL formula φ such that the size of M is quadratic in the length of ψ, the length of φ is linear in the length of ψ, and ψ is satisfiable iff $M \not\models_r \neg\varphi$.

Consider a CTL formula ψ. For simplicity, let us first assume that ψ has a single atomic proposition q. Let n be the number of existential quantifiers in ψ plus 1. By the sufficient branching-degree property of CTL, ψ is satisfiable iff there exists a $\{\emptyset,\{q\}\}$-labeled tree of branching degree n that satisfies ψ [Eme90]. Let P_n be a clique with n states. By the above, ψ is satisfiable iff there exists a possibility to label an unwinding of P_n such that the resulted $\{\emptyset,\{q\}\}$-labeled tree satisfies ψ. This simple idea, due to [Kup97], 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 φ such that there exists a tree satisfying φ in $exec(M_n)$ iff there exists a $\{\emptyset,\{q\}\}$-labeling of $\langle T_{P_n}, V_{P_n}\rangle$ that satisfies ψ. It follows that ψ 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,

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



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 φ such that whenever the formula ψ refers to q, the formula φ 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 φ should be restricted to computations of P_n. That is, to paths that never meet a ghost. To do this, we again use the function f : CTL* formulas $\rightarrow$ CTL* formulas defined above, this time $f(\xi)$ restricts path quantification to paths that never visit a state labeled with *ghost*. For example, $f(EqU(AFp)) = E((G\neg ghost) \wedge (qU(A((F ghost) \vee Fq))))$. We can now define φ as $f(\psi)$ with $EXEXq$ replacing q. Note that we first apply f and only then do the replacement.

When ψ has more than one atomic proposition, the reduction is very similar. Then, for ψ over $\{q_1, \ldots, q_m\}$, we have m heavens, one for each atomic proposition. Accordingly, we replace a proposition q_i in ψ with $EXEXq_i$ in φ. The obtained module has $2n + m + 1$ states and it has $n^2 + n(m+2) + m + 1$ transitions.

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

We note that the problem of CTL module checking is EXPTIME-complete even for the following restricted versions of the problem.

- The modules have only environment states. To see this, we define M_n as the clique P_n, adding a transition from each state to heaven (with no ghosts involved). We 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 ψ with

EXq in φ. The price of using only environment states is that now the length of φ is quadratic in the length of ψ.

- The modules are of a fixed size. To see this, note that the size of M_n depends on the number of atomic propositions in ψ and on the minimum branching degree of models of ψ. 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 ψ_x [FL79]. In an alternating Turing machine (with a binary branching degree), each configuration has two possible successor configurations. Some of the configurations are classified as *universal* and some are classified as *existential*. A computation of an alternating Turing machine is a tree labeled with configurations. A node that is labeled by a universal configuration should have two successors. A node that is labeled by an existential configuration may have only one successor. The computation is accepting iff all the paths in the tree eventually reach an accepting configuration. A somewhat different reduction, which considers a fixed linear-space Turing machine that decides an EXPTIME-complete problem, results in ψ_x of length polynomial in $|x|$, but with a fixed number of atomic propositions, which, if satisfiable, has models with branching degree 2. Such ψ_x induces, for all x, modules of a fixed size.

- The formulas are in $\exists$CTL, the existential fragment of CTL. To see this, consider again a fixed linear-space alternating Turing machine T that decides an EXPTIME-complete problem. The formula ψ_x above encodes all the possible accepting computations of T on an input word x. Alternation between universal and existential configurations in T is handled by alternation between universal and existential path quantification in ψ_x. Another way to handle alternation in T is by alternation between system and environment states in the module.

  To see this, assume that T has a tape of length n. Since T is fixed, then by encoding each of its configurations by a sequence of n states, we can define a module M of size linear in n such that M embodies all the possible computations of T on x (and more trees, that are not legal computations). Thus, the tree $\langle T_M, V_M \rangle$ embodies the execution of T on x when we require all the configurations of T, and not only the universal ones, to have two successors in the computation. An environment to M prunes subtrees that correspond to branches from T's existential configurations. Thus, a state in M that encodes a last node in an existential configuration of T is an environment state. On the other hand, since we do not want the environment to prune subtrees that branch from T's universal states, and since neither do we want it to prune a subtree whose root is not the first node in a sequence of nodes that encode a configuration, all other states in M are system states. It follows that each tree in $exec(M)$ corresponds to a computation of T on x. We define a $\forall$CTL formula φ such that T accepts x iff there exists a tree in $exec(M)$ that satisfies φ (hence, T accepts x iff M does not satisfy φ, which is an $\exists$CTL formula). As in [FL79], the formula φ requires that each branch in the tree encodes a sequence of configurations that starts at the initial configuration and ends at an accepting configuration.

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* 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 when we measure the complexity of the module-checking problem in terms of both the program and the formula, then module checking of CTL and CTL* formulas is much harder than their model checking. We now claim that when we consider program complexity, module checking is still harder.

Theorem 4.1 *The program complexity of CTL and CTL* module checking is PTIME-complete with respect to logspace reductions.*

Proof: Since the algorithms given in the proof of Theorem 3.1 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 α (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 α 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 α, 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 $\vec{x} = \langle x_1, \ldots, x_n \rangle$, we construct a module $M_{\alpha,\vec{x}} = \langle \{0,1\}, G_\forall, G_\exists \cup G_{in}, R, g_{out}, L \rangle$, 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 $M_{\alpha,\vec{x}}$ is linear in the size of α. Intuitively, each tree in $exec(M_{\alpha,\vec{x}})$ corresponds to a decision of α 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 there exists a tree $\langle T, V\rangle \in exec(M_{\alpha,\vec{x}})$ such that $\langle T, V\rangle \models AG1$ iff the output of α on x is 1. Hence, by the definition of module checking, we have that $M_{\alpha,\vec{x}} \models_r EF0$ iff the output of α on x is 0. $\square$

Recall that for a CTL formula ψ, checking that a module M satisfies ψ 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 μ-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 5.1 *For universal branching temporal logics, the module-checking problem and the model-checking problem coincide.*

Proof: Given a module M and a $\forall$CTL* formula ψ, 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 ψ. Thus, in particular, $\langle T_M, V_M\rangle$ satisfies ψ and $M \models \psi$. Assume now that $M \models \psi$. Every tree in $exec(M)$ has less behaviors than the tree $\langle T_M, V_M\rangle$. Thus, formally, the tree $\langle T_M, V_M\rangle$ is simulated by all trees in $exec(M)$. Indeed, for every tree $\langle T, V\rangle \in exec(M)$, the relation that maps each node x of T to the node x of $\langle T_M, V_M\rangle$ is a simulation relation between $\langle T, V\rangle$ and $\langle T_M, V_M\rangle$. Therefore, by [GL94], all trees in $exec(M)$ satisfy ψ, and $M \models_r \psi$. $\square$

In particular, it follows from Lemma 5.1 that module checking and model checking coincides for linear temporal logics. Theorem 5.2 now follows from the known complexity results for LTL, $\forall$CTL, and $\forall$CTL* model checking [SC85, CES86, BVW94, VW94].

Theorem 5.2

(1) *The module-checking problem for $\forall$CTL is in linear time.*

(2) *The module-checking problem for LTL and $\forall$CTL* is PSPACE-complete.*

(3) *The program complexity of module checking for LTL, $\forall CTL$, and $\forall CTL^\star$ is NLOGSPACE-complete.*

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

We have seen that, for each fixed CTL formula ψ, checking that a module M satisfies ψ 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 $\forall$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 cannot be specified in linear temporal logics, nor in universal branching logics [EH86].

Theorem 5.3 *Module checking of possibly and always possibly properties can be done in linear running time.*

Proof: 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 ξ. 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 every tree $\langle T, V\rangle \in exec(M)$ that has a node labeled w is guaranteed to satisfy $EF\xi$. In other words, w is safe iff w cannot be a node in a tree all of whose nodes satisfy $\neg\xi$. We check that $M \models_r EF\xi$ by checking that w_0 is safe. With every safe state w we can associate a finite integer $level(w)$ such that for all trees in $exec(M)$, the length of the shortest path from a node labeled w to a node that satisfies ξ is at most $level(w)$. Formally, we define the level of each state by initially assigning the level 0 to states that satisfy ξ and assigning the level ∞ to all other states. We then proceed iteratively. In the ith iteration we assign the level i to system states that have a successor in level $i-1$ and to environment states all of whose successors are in level at most $i-1$. Clearly, the maximal finite level that each state may get is $|W|$, which also bounds the number of iterations required.

Consider the monotone function $f : 2^W \rightarrow 2^W$ where a state w is in $f(y)$ iff one of the following holds:

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

We prove that w is safe iff w is in the least fixed-point of f. For that, we use the Tarski-Knaster characterization of least fixed-points and prove that

$$w \text{ is safe iff } w \in \bigcap\{y : f(y) = y\}.$$

Assume first that $w \in \bigcap\{y : f(y) = y\}$. Let S be the set of all safe states. 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.

Hence, $f(S) = S$, which implies, by the assumption, that w is in S and is therefore safe.

Assume now that w is safe. Recall that if w is safe then $level(w) = i$ for some finite i. We prove that for all finite i, if $level(w) = i$, then $w \in \bigcap\{y : f(y) = y\}$. The proof proceeds by induction on i. Consider first the case where $level(w) = 0$. Let y be any set of states for which $f(y) = y$. Since $w \models \xi$, then $w \in f(y)$. Hence, as $f(y) = y$, we have that $w \in y$, and we are done. Assume now that the claim holds for all levels $j \leq i$ and consider a state w with $level(w) = i + 1$. Let y be any set for which $f(y) = y$. We distinguish between two cases. First, if w is a system state, then, by the definition of its level, there exists a state w' such that w' is a successor of w and $level(w') = i$. Then, by the induction hypothesis, $w' \in \bigcap\{z : f(z) = z\}$. In particular, $w' \in y$. Therefore, $w \models (s \wedge EXy)$ and thus, by the definition of f, we have that $w \in f(y)$. Hence, $f(y) = y$, we have that $w \in y$, and we are done. Second, if w is an environment state then, by the definition of its level, all its successor states w' have $level(w') \leq i$, implying, by the induction hypothesis, that $w' \in \bigcap\{z : f(z) = z\}$. So, all the successors of w are in y. Therefore, $w \models (e \wedge AXy)$ and thus, by the definition of f, we have that $w \in f(y)$. Hence, $w \in y$, and we are done.

Using μ-calculus formalism [Koz83], 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 μ-calculus formula. As the latter can be done in linear running time [Cle93], we are done. $\square$

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 4.1 is $EF\xi$, the answer for possibly properties is no, unless NLOGSPACE = PTIME. Unsurprisingly, this is also the answer for the more complicated always possibly properties, as we claim in the theorem below.

Theorem 5.4 *Module checking of possibly and always possibly properties is PTIME-complete.*

Proof: Membership in PTIME follows from Theorem 5.3. 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 resulting module $M'_{\alpha,\vec{x}}$. It is easy to see that there exists a tree $\langle T, V\rangle \in exec(M'_{\alpha,\vec{x}})$ such that $\langle T, V\rangle \models EFAG1$ iff the output of α on x is 1. Hence, $M'_{\alpha,\vec{x}} \models_r AGEF0$ iff the output of α on x is 1. □

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 [McM93, CGL93] are incorporated into industrial development of new designs [BBG$^+$94].

If we examine the history of these issues more closely, we find that things are not that simple. On one hand, the inability of LTL to quantify computations existentially is considered by many a serious drawback [EH86]. In addition, the introduction of fair-CTL [CES86] and of many other extensions to CTL [Lon93, BBG$^+$94, KG96], 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.

The distinction between closed an open systems questions the computational superiority of the branching-time paradigm further. Indeed, the traditional belief of "CTL is easier than LTL" compares these logics with respect to an ill defined setting: the systems are assumed to be nondeterministic, but the nature of the nondeterminism (internal or external) is never considered. One might be tempted to answer that model checking deals with complete systems, not with modules, and hence that it is checking closed systems and that all nondeterminism is internal. This, however, is a fallacy.

To see this, let us think about the origin of the nondeterminism in the programs we do model check. First, there are no good reasons to write inherently nondeterministic programs. Furthermore, this is in fact impossible since, even if one can introduce nondeterministic constructs in a programming language, any implementation of this language will make the choices one way or another and hence be deterministic. Similarly, the nondeterminism that comes from the modeling of concurrency is also not inherent. Indeed, enough knowledge about the environment in which a concurrent program is run (the machine, the operating system,...) can, at least in theory, remove all uncertainty about its execution. So, does this mean that a closed system is always deterministic and hence always has a single linear execution? It does, but only if the system is modeled at a sufficient level of detail. This being almost always impossible, one uses nondeterminism as a representation of the lack of knowledge one has about parts of the system or its implementation: if we do not know what the system will do, we assume it can do anything. If we do not know how processes are scheduled, we assume that scheduling is nondeterministic. But, this is *external* nondeterminism: the choices are viewed as being made by an external agent of which the system has no detailed knowledge and does not control. The same is true of the "models of the environment" that are often added to systems in order to close them: not knowing what the environment will do, we model it as being nondeterministic; it makes unpredictable and uncontrollable choices.

Form this it follows that, at the level of abstraction at which model checking is usually done, there are no properly closed systems. Indeed, using nondeterminism to model absence of knowledge about system components amounts to introducing external nondeterminism and hence to describing a module, not a closed system. So, what is needed is module checking, and is even just linear-time module checking, unless we can identify a source of internal nondeterminism.

Suppose that nondeterminism is used not to model a lack of knowledge about a part of the system over which one has no control, but to model the fact that some implementation choices have not yet been made. For instance, imagine there is a scheduler that will be implemented by the system designer but that does not yet exist. A sound design approach could require that the system be model checked with a simple nondeterministic representation of the scheduler before it is actually implemented. The nondeterminism used in the scheduler representation is then internal: the choices are under the control of the system; the implementation of the scheduler can contain any desired choice policy. Furthermore, in this context branching-time model checking becomes quite natural. One can for instance check the existence of paths that do make the implementation of a reasonable scheduling policy feasible. But, what is required is module checking: the branching formula has to be satisfied whatever external choices are made.

Our conclusion thus is that branching-time model checking should essentially always be treated as branching-time module checking. Moreover, the use of existential path quantifiers is especially meaningful when one is checking the possibility of implementing a part of the system that has so far been modeled nondeterministically. In other words, when one is checking for the possibility of synthesizing part of the system. Quite interestingly, the reactive module synthesis problem studied in [PR89a, PR89b] or the realizability problem of [ALW89] can be described as CTL* module checking problems. A similar link also exists with the supervisory control problem studied in the context of control theory [Ant95].

Once one adopts this view that what we need is module checking rather than model checking, the usual argument that, from a complexity point of view, CTL is easier than LTL, topples. Indeed, the situation is then similar to the one existing for validity checking. In both problems, CTL is harder than LTL, and the universal fragment of CTL is easier than its existential one [KV95]. Finally, we note that the additional difficulties that need to be faced when we move from verification of closed systems to verification of open systems, do not arise when we consider verification by *bisimulation* [Mil71]. To see this, consider two modules M and M'. When we label the states of the modules by their system/environment classification, then M and M' are bisimilar iff every tree in $exec(M)$ is bisimulated by a tree in $exec(M')$, and vice versa. Thus, we can check bisimilarity between two modules in linear time, exactly as we check bisimilarity between programs. As in [BCG88], two modules are bisimilar iff they agree on satisfaction (in the module-checking sense) of all CTL* formulas.

Our results are summarized in the table below. All the complexities in the table denote tight bounds.

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

References

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symposium on Foundations of Computer Science*, Florida, October 1997.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372, pages 1–17. Lecture Notes in Computer Science, Springer-Verlag, July 1989.

[Ant95] M. Antoniotti. *Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the Control-D system.* PhD thesis, New York University, New York, 1995.

	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, Pra80]
CTL*	PSPACE [EL85]	2EXPTIME	NLOGSPACE [BVW94]	PTIME	2EXPTIME [EJ88, VS85]
$\forall$CTL	linear-time [CES86]	linear-time	NLOGSPACE [BVW94]	NLOGSPACE	PSPACE [KV95]
$\exists$CTL	linear-time [CES86]	EXPTIME	NLOGSPACE [BVW94]	PTIME	NPTIME [KV95]
$EF\xi$ $AGEF\xi$	linear-time [CES86]	linear-time	NLOGSPACE [BVW94]	PTIME	NPTIME [GJ79]

[ASSS94] 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 Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193, Stanford, June 1994.

[BCG88] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[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 *Proc. 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[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)*, volume 803 of *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 μ-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 *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 368–377, 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*, North Holywood, 1985. Western Periodicals Company.

[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 *Proc. 16th ACM Symposium on Theory of Computing*, Washington, April 1984.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal 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*, volume 331 of *Lecture Notes in Computer Science*, 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.

[KG96] O. Kupferman and O. Grumberg. Buy one, get one free!!! *Journal of Logic and Computation*, 6(4):523–539, 1996.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kup97] O. Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. *Journal of Logic and Computation*, 7:1–14, 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.

[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.

[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 *Proc. 12th 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 *Proc. 3th Symposium on Logic in Computer Science*, Edinburgh, 1988.

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

[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.

[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.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372, pages 652–671. Lecture Notes in Computer Science, Springer-Verlag, July 1989.

[Pra80] V.R. Pratt. A near-optimal method for reasoning about action. *Journal of Computer and System Sciences*, 20(2):231–254, 1980.

[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.

[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. *Journal ACM*, 32:733–749, 1985.

[Var95] M.Y. Vardi. On the complexity of modular model checking. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, pages 101–111, 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 *Proc. 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.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.