Verification of Open Systems

Moshe Y. Vardi*

Rice University, Department of Computer Science, 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 verification 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. *Module checking* is an algorithmic method that checks, given an open system (modeled as a finite structure) and a desired requirement (specified by a temporal-logic formula), whether the open system satisfies the requirement with respect to all environments. In this paper we describe and examine *module checking problem*, and study its computational complexity. Our results show that module checking is computationally harder than model checking.

1 Introduction

Temporal logics, which are modal logics geared towards the description of the temporal ordering of events, have been adopted as a powerful tool for specifying and verifying reactive systems [Pnu81]. One of the most significant developments in this area is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [CE81, QS81, LP85, CES86, VW86a]. This derives its significance both from the fact that many synchronization and communication protocols can be modeled as finite-state systems, as well as from the great ease of use of fully algorithmic methods. Experience has shown that algorithmic verification techniques scale up to industrial-sized designs [CGH$^+$95], and tools based on such techniques are gaining acceptance in industry [BBG$^+$94]

We distinguish here between two types of temporal logics: *universal* and *non-universal*. Both logics describe the computation tree induced by the system. Formulas of universal temporal logics, such as LTL, $\forall$CTL, and $\forall$CTL*, describe requirements that should hold in all the branches of the tree [GL94]. These requirements may be either linear (e.g., in all computations, only finitely many requests are sent) as in LTL or branching (e.g., in all computations we eventually reach a state from which, no matter how we continue, no requests are sent) as in $\forall$CTL. In both cases, the more behaviors the system has, the harder it is for the system to satisfy the requirements. Indeed, universal temporal logics induce the *simulation* order between systems [Mil71, CGB86]. That is, a system M simulates a system M' if and only if all universal temporal logic formulas that are satisfied in M' are satisfied in M as well. On the other hand, formulas of non-universal temporal logics, such as CTL and CTL*, may also impose possibility requirements on the system (e.g., there exists a computation in which only finitely many requests are sent) [EH86]. Here, it is no longer

* Supported in part by NSF grants CCR-9628400 and CCR-9700061 and by a grant from the Intel Corporation.

true that simulation between systems corresponds to agreement on satisfaction of requirements. Indeed, it might be that adding behaviors to the system helps it to satisfy a possibility requirement or, equivalently, that disabling some of its behaviors causes the requirement not to be satisfied.

We also distinguish between two types of systems: *closed* and *open* [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. Thus, while in a closed system all the nondeterministic choices are internal, and resolved by the system, in an open system there are also external nondeterministic choices, which are resolved by the environment [Hoa85]. In order to check whether a closed system satisfies a required property, we translate the system into some formal model, specify the property 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. In order to check whether an open system satisfies a required property, we should check the behavior of the system with respect to any environment, and often there is much uncertainty regarding the environment [FZ88]. In particular, it might be that the environment does not enable all the external nondeterministic choices. To see this, consider a sandwich-dispensing machine that serves, upon request, sandwiches with either ham or cheese. The machine is an open system and an environment for the system is an infinite line of hungry people. Since each person in the line can like either both ham and cheese, or only ham, or only cheese, each person suggests a different disabling of the external nondeterministic choices. Accordingly, there are many different possible environments to consider.

It turned out that model-checking methods are applicable also for verification of open systems with respect to universal temporal-logic formulas [MP92, KV96, KV97]. To see this, consider an execution of an open system in a maximal environment; i.e., an environment that enables all the external nondeterministic choices. The result is a closed system, and it is simulated by any other execution of the system in some environment. Therefore, one can check satisfaction of universal requirements in an open system by model checking the system viewed as a closed system (i.e., all nondeterministic choices are internal). This approach, however, can not be adapted when verifying an open system with respect to non-universal requirements. Here, satisfaction of the requirements with respect to the maximal environment does not imply their satisfaction with respect to all environments. Hence, we should explicitly make sure that all possibility requirements are satisfied, no matter how the environment restricts the system. For example, verifying that the sandwich-dispensing machine described above can always eventually serve ham, we want to make sure that this can happen no matter what the eating habits of the people in line are. Note that while this requirement holds with respect to the maximal environment, it does not hold, for instance, in an environment in which all the people in line do not like ham.

Module checking is suggested in [KV96, KVW97, KV97] as a general method for verification of open systems (we use the terms "open system" and "module" interchangeably). Given a module M and a temporal-logic formula ψ, the module-checking problem asks whether for all possible environments $\mathcal{E}$, the execution of M in $\mathcal{E}$ satisfies ψ. There are two ways to model open systems. In the first approach [KV96, KVW97], we model open systems by transition systems with a partition of the states into two sets. One set contains *system states* and corresponds to states where the system makes a transition. The second set contains *environment states* and corresponds to states where the environment makes a transition. 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 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.

We examine the *complexity* of the module-checking problem for non-universal temporal logics. It turns out that for such logics module checking is much harder than model checking; in fact, module checking as is as hard as satisfiability. Thus, CTL module checking is EXPTIME-complete and CTL* module checking is 2EXPTIME-complete. In both cases the complexity in terms of the size of the module is polynomial.

In the second approach to modeling open systems [KV97], we look at the states of the transition system in more detail. We view these states as assignment of values to variables. These variables are controlled by either the system or by the environment. In this approach we can capture the phenomenon in which the environment the has *incomplete information* about the system; i.e., not all the variables are readable by the environment. Let us explain this issue in greater detail.

An interaction between a system and its environment proceeds through a designated set of input and output variables. In addition, the system often has internal variables, which the environment cannot read. If two states of the system differ only in the values of unreadable variables, then the environment cannot distinguish between them. Similarly, if two computations of the system differ only in the values of unreadable variables along them, then the environment cannot distinguish between them either and thus, its behaviors along these computations are the same. More formally, when we execute a module M with an environment $\mathcal{E}$, and several states in the execution look the same and have the same history according to $\mathcal{E}$'s incomplete information, then the nondeterministic choices done by $\mathcal{E}$ in each of these states coincide. In the sandwich-dispensing machine example, the people in line cannot see whether the ham and the cheese are fresh. Therefore, their choices are independent of this missing information. Given an open system M with a partition of M's variables into readable and unreadable, and a temporal-logic formula ψ, the module-checking problem with incomplete information asks whether the execution of M in $\mathcal{E}$ satisfies ψ, for all environments $\mathcal{E}$ whose nondeterministic choices are independent of the unreadable variables (that is, $\mathcal{E}$ behaves the same in indistinguishable states).

It turns out that the presence of incomplete information makes module checking more complex. The problem of module checking with incomplete information is is EXPTIME-complete and 2EXPTIME-complete for CTL and CTL*, respectively. In both cases, however, the complexity in terms of the size of the module is exponential, making module checking with incomplete information quite intractable.

2 Module Checking

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** 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 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 $L : W \to 2^{AP}$ 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 (with respect to a given program P). The relation $\models$ is inductively defined as follows.

– For all w, we have that $w \models$ **true** and $w \not\models$ **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").
– $F\varphi =$ **true**$U\varphi$ ("eventually").
– $G\varphi = \neg F\neg\varphi$ ("always").
– $A\varphi = \neg E\neg\varphi$ ("for all paths").

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 unique $c \in [k]$ 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 $V : T \rightarrow \Sigma$ 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. 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 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:

- $V(\varepsilon) = w_0$.
- 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 *module checking*. 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, CES86, EL85, 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.

As explained earlier, the distinction between model checking and module checking does not apply to universal temporal logics.

Lemma 1. [KV96, KVW97] *For universal temporal logics, the module-checking problem and the model-checking problem coincide.*

In order to solve the module-checking problem for non-universal logics, we 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, $\delta : Q \times \Sigma \times \mathcal{D} \rightarrow 2^{Q^*}$ 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 $O(|\mathcal{A}|)$.

3 The Complexity of Module Checking

We have already seen that for non-universal 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 2. [KV96]

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

Proof (sketch): 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 does 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 in T_M).
- $Q = W \times \{\top, \vdash, \bot\})$. Thus, every state w of M induces three states $\langle w, \top\rangle$, $\langle w, \vdash\rangle$, and $\langle w, \bot\rangle$ in $\mathcal{A}_M$. Intuitively, when $\mathcal{A}_M$ is in state $\langle w, \bot\rangle$, it can read only the letter $\bot$. 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 $\bot$. 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 $\langle w, \bot\rangle$ 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 $\delta : Q \times \Sigma \times \mathcal{D} \to 2^{Q^*}$ 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 $m \in \{\vdash, \bot\}$, we have
  $$\delta(\langle w, m\rangle, \bot, k) = \langle\langle w_1, \bot\rangle, \langle w_2, \bot\rangle, \ldots, \langle w_k, \bot\rangle\rangle.$$
  - 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
  $$\begin{aligned}\delta(\langle w, m\rangle, L(w), k) = \{ &\langle\langle w_1, \top\rangle, \langle w_2, \vdash\rangle, \ldots, \langle w_k, \vdash\rangle\rangle,\\ &\langle\langle w_1, \vdash\rangle, \langle w_2, \top\rangle, \ldots, \langle w_k, \vdash\rangle\rangle,\\ &\vdots\\ &\langle\langle w_1, \vdash\rangle, \langle w_2, \vdash\rangle, \ldots, \langle w_k, \top\rangle\rangle \}.\end{aligned}$$

    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 for the case $k \neq bd(w)$ or when the input that not meet the restriction imposed by the $\top, \vdash$, and $\bot$ annotations, or 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 $\bot$ 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 tress recognized by $\mathcal{A}_M$, we should treat a node that is labeled by $\bot$ as a node that does not exist. To do this, we define a function f : CTL* formulas $\to$ CTL* formulas such that $f(\xi)$ restricts path quantification to paths that never visit a state labeled with $\bot$. 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, PR89], 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 $M \not\models_r \neg\varphi$. The proof is the same for CTL*. Here, we do a reduction from satisfiability of CTL*, proved to be 2EXPTIME-hard in [VS85]. See [KV96] for more details. □

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]. We now 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 3. [KV96] *The program complexity of CTL and CTL* module checking is PTIME-complete.*

Proof: Since the algorithms given in the proof of Theorem 2 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 $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, 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 α. Intuitively, each tree in $exec(M_{\alpha,\mathbf{x}})$ 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 $\langle T, V \rangle \in exec(M_{\alpha,\mathbf{x}})$ 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 $M_{\alpha,\mathbf{x}} \models_r EF0$ iff the output of α on x is 0. □

4 Module Checking with Incomplete Information

We first need to generalize the definition of trees from Section 2. Given a finite set Υ, an *Υ-tree* is a nonempty set $T \subseteq \Upsilon^*$ such that if $s \cdot \upsilon \in T$, where $s \in \Upsilon^*$ and $\upsilon \in \Upsilon$, then also $s \in T$. When Υ is not important or clear from the context, we call T a tree. The elements of T are called *nodes*, and the empty word ϵ is the *root* of T. For every $s \in T$, the nodes $s \cdot \upsilon \in T$ where $\upsilon \in \Upsilon$ are the *children* of s. An Υ-tree T is a *full infinite tree* if $T = \Upsilon^*$. Each node s of T has a *direction* in Υ. The direction of the root is some designated $\upsilon_0 \in \Upsilon$. The direction of a node $s \cdot \upsilon$ is υ. A *path* π of T is a set $\pi \subseteq T$ such that $\epsilon \in \pi$ and for every $s \in \pi$ there exists a unique $\upsilon \in \Upsilon$ such that $s \cdot \upsilon \in \pi$. Given two finite sets Υ and Σ, a *Σ-labeled Υ-tree* is a pair $\langle T, V \rangle$ where T is an Υ-tree and $V : T \to \Sigma$ maps each node of T to a letter in Σ. When Υ and Σ are not important or clear from the context, we call $\langle T, V \rangle$ a labeled tree.

For finite sets X and Y, and a node $s \in (X \times Y)^*$, let $hide_Y(s)$ be the node in X^* obtained from s by replacing each letter $(x \cdot y)$ by the letter x. For example, when $X = Y = \{0, 1\}$, the node 0010 of the $(X \times Y)$-tree on the right corresponds, by $hide_Y$, to the node 01 of the X-tree on the left. Note that the nodes $0011, 0110$, and 0111 of the $(X \times Y)$-tree also correspondto the node 01 of the X-tree.

Let Z be a finite set. For a Z-labeled X-tree $\langle T, V \rangle$, we define the *Y-widening* of $\langle T, V \rangle$, denoted $wide_Y(\langle T, V \rangle)$, as the Z-labeled $(X \times Y)$-tree $\langle T', V' \rangle$ where for every $s \in T$, we have $hide_Y^{-1}(s) \subseteq T'$ and for every $t \in T'$, we have $V'(t) = V(hide_Y(t))$. Note that for every node $t \in T'$, and $x \in X$, the children $t \cdot (x \cdot y)$ of t, for all y, agree on their label in $\langle T', V' \rangle$. Indeed, they are all labeled with $V(hide_Y(t) \cdot x)$.

We now describe a second approach to modeling open systems. We describe an open system by a *module* $M = \langle I, O, H, W, w_0, R, L \rangle$, where

- I, O, and H are sets of input, readable output, and hidden (internal) variables, respectively. We assume that I, O, and H are pairwise disjoint, we use K to denote the variables known to the environment; thus $K = I \cup O$, and we use P to denote all variables; thus $P = K \cup H$.
- W is a set of states, and $w_0 \in W$ is an initial state.
- $R \subseteq W \times W$ is a total transition relation. For $\langle w, w' \rangle \in R$, we say that w' is a successor of w. Requiring R to be total means that every state w has at least one successor.
- $L : W \to 2^P$ maps each state to the set of variables that hold in this state. The intuition is that in every state w, the module reads $L(w) \cap I$ and writes $L(w) \cap (O \cup H)$.

A *computation* of M is a sequence $w_0, w_1, \ldots$ of states, such that for all $i \geq 0$ we have $\langle w_i, w_{i+1}\rangle \in R$. We define the *size* $|M|$ of M as $(|W| * |P|) + |R|$. We assume, without loss of generality, that all the states of M are labeled differently; i.e., there exist no w_1 and w_2 in W for which $L(w_1) = L(w_2)$ (otherwise, we can add variables in H that differentiate states with identical labeling). With each module M we can associate a computation tree $\langle T_M, V_M\rangle$ obtained by pruning M from the initial state. More formally, $\langle T_M, V_M\rangle$ is a 2^P-labeled 2^P-tree (not necessarily with a fixed branching degree). Each node of $\langle T_M, V_M\rangle$ corresponds to a state of M, with the root corresponding to the initial state. A node corresponding to a state w is labeled by $L(w)$ and its children correspond to the successors of w in M. The assumption that the nodes are labeled differently enable us to embody $\langle T_M, V_M\rangle$ in a $(2^P)^*$-tree, with a node with direction v labeled v.

A module M is *closed* iff $I = \emptyset$. Otherwise, it is *open*. Consider an open module M. The module interacts with some environment $\mathcal{E}$ that supplies its inputs. When M is in state w, its ability to move to a certain successor w' of w is conditioned by the behavior of its environment. If, for example, $L(w') \cap I = \sigma$ and the environment does not supply σ to M, then M cannot move to w'. Thus, the environment may disable some of M's transitions. We can think of an environment to M as a *strategy* $\mathcal{E} : (2^K)^* \to \{\top, \bot\}$ that maps a finite history s of a computation (as seen by the environment) to either $\top$, meaning that the environment enables M to execute s, or $\bot$, meaning that the environment does not enable M to execute s. In other words, if M reaches a state w by executing some $s \in (2^K)^*$, and a successor w' of w has $L(w) \cap K = \sigma$, then an interaction of M with $\mathcal{E}$ can proceed from w to w' iff $\mathcal{E}(s \cdot \sigma) = \top$. We say that the tree $\langle (2^K)^*, \mathcal{E}\rangle$ *maintains* the strategy applied by $\mathcal{E}$. We denote by $M \lhd \mathcal{E}$ the execution of M in $\mathcal{E}$; that is, the tree obtained by pruning from the computation tree $\langle T_M, V_M\rangle$ subtrees according to $\mathcal{E}$. Note that $\mathcal{E}$ may disable all the successors of w. We say that a composition $M \lhd \mathcal{E}$ is *deadlock free* iff for every state w, at least one successor of w is enabled. Given M, we can define the *maximal environment* $\mathcal{E}_{max}$ for M. The maximal environment has $\mathcal{E}_{max}(x) = \top$ for all $x \in (2^K)^*$; thus it enables all the transitions of M.

Recall that in Section 2, we modeled open systems using system and environment states, and only transitions from environment states may be disabled. Here, the interaction of the system with its environment is more explicit, and transitions are disabled by the environment assigning values to the system's input variables.

The hiding and widening operators enable us to refer to the interaction of M with $\mathcal{E}$ as seen by both M and $\mathcal{E}$. As we shall see below, this interaction looks different from the two points of views. First, clearly, the labels of the computation tree of M, as seen by $\mathcal{E}$, do not contain variables in H. Consequently, $\mathcal{E}$ thinks that $\langle T_M, V_M\rangle$ is a 2^K-tree, rather than a 2^P-tree. Indeed, $\mathcal{E}$ cannot distinguish between two nodes that differ only in the values of variables in H in their labels. Accordingly, a branch of $\langle T_M, V_M\rangle$ into two such nodes is viewed by $\mathcal{E}$ as a single transition. This incomplete information of $\mathcal{E}$ is reflected in its strategy, which is independent of H. Thus, successors of a state that agree on the labeling of the readable variables are either all enabled or all disabled. Formally, if $\langle (2^K)^*, \mathcal{E}\rangle$ is the $\{\top, \bot\}$-labeled 2^K-tree that maintains the strategy applied by $\mathcal{E}$, then the $\{\top, \bot\}$-labeled 2^P-tree $wide_{(2^H)}(\langle (2^K)^*, \mathcal{E}\rangle)$ maintains the "full" strategy for $\mathcal{E}$, as seen by someone that sees both K and H.

Another way to see the effect of incomplete information is to associate with each environment $\mathcal{E}$ a tree obtained from $\langle T_M, V_M\rangle$ by pruning some of its subtrees. A subtree with root $s \in T_M$ is pruned iff $K'(hide_{(2^H)}(s)) = \bot$. Every two nodes s_1 and s_2 that are indistinguishable according to $\mathcal{E}$'s incomplete information have $hide_{(2^H)}(s_1) = hide_{(2^H)}(s_2)$. Hence, either both subtrees with roots s_1 and s_2 are pruned or both are not pruned. Note that once $\mathcal{E}(x) = \bot$ for some $s \in (2^K)^*$, we can assume that $\mathcal{E}(s \cdot t)$ for all $t \in (2^K)^*$ is also $\bot$. Indeed, once the environment disables the transition to a certain node s, it actually disables the transitions to all the nodes in the subtree with root s. Note also that $M \lhd \mathcal{E}$ is deadlock free iff for every $s \in T_M$ with $\mathcal{E}(hide_{(2^H)}(s)) = \top$, at least one direction $\upsilon \in 2^P$ has $s \cdot \upsilon \in T_M$ and $\mathcal{E}(hide_{(2^H)}(s \cdot \upsilon)) = \top$.

5 The Complexity of Module Checking with Incomplete Information

The *module-checking with incomplete information* problem is defined as follows. Let M be a module, and let ψ be a temporal-logic formula over the set P of M's variables. Does $M \lhd \mathcal{E}$ satisfy ψ for every environment $\mathcal{E}$ for which $M \lhd \mathcal{E}$ is deadlock free? When the answer to the module-checking question is positive, we say that M *reactively satisfies* ψ, denoted $M \models_r \psi$. Note that when $H = \emptyset$, i.e., there are no hidden variables, then we get the module-checking problem, which was studied in Section 3.

Even with incomplete information, the distinction between model checking and module checking does not apply to universal temporal logics.

Lemma 4. [KV97] *For universal temporal logics, the module-checking with incomplete information problem and the model-checking problem coincide.*

Dealing incomplete information for non-universal logics is complicated. The solution we suggest is based on alternating tree automata and is outlined below. In Sections 5.1 and 5.2, we define alternating tree automata and describe the solutions in detail. We start by recalling the solution to the module-checking problem. Given M and ψ, we proceed as follows.

A1. Define a nondeterministic tree automaton $\mathcal{A}_M$ that accepts all the 2^P-labeled trees that correspond to compositions of M with some $\mathcal{E}$ for which $M \lhd \mathcal{E}$ is deadlock free. Thus, each tree accepted by $\mathcal{A}_M$ is obtained from $\langle T_M, V_M \rangle$ by pruning some of its subtrees.
A2. Define a nondeterministic tree automaton $\mathcal{A}_{\neg\psi}$ that accepts all the 2^P-labeled trees that do not satisfy ψ.
A3. $M \models_r \psi$ iff no composition $M \lhd \mathcal{E}$ satisfies $\neg\psi$, thus iff the intersection of $\mathcal{A}_M$ and $\mathcal{A}_{\neg\psi}$ is empty.

The reduction of the module-checking problem to the emptiness problem for tree automata implies, by the finite-model property of tree automata [Eme85], that defining reactive satisfaction with respect to only *finite-state* environments is equivalent to the current definition.

In the presence of incomplete information, not all possible pruning of $\langle T_M, V_M \rangle$ correspond to compositions of M with some $\mathcal{E}$. In order to correspond to such a composition, a tree should be *consistent in its pruning*. A tree is consistent in its pruning iff for every two nodes that the paths leading to them differ only in values of variables in H (i.e., every two nodes that have the same history according to $\mathcal{E}$'s incomplete information), either both nodes are pruned or both nodes are not pruned. Intuitively, hiding variables from the environment makes it easier for M to reactively satisfy a requirement: out of all the pruning of $\langle T_M, V_M \rangle$ that should satisfy the requirement in the case of complete information, only these that are consistent should satisfy the requirement in the presence of incomplete information. Unfortunately, the consistency condition is non-regular, and cannot be checked by an automaton. In order to circumvent this difficulty, we employ alternating tree automata. We solve the module-checking problem with incomplete information as follows.

B1. Define an alternating tree automaton $\mathcal{A}_{M,\neg\psi}$ that accepts a $\{\top, \bot\}$-labeled 2^K-tree iff it corresponds to a strategy $\langle (2^K)^*, \mathcal{E} \rangle$ such that $M \lhd \mathcal{E}$ is deadlock free and does not satisfy ψ.
B2. $M \models_r \psi$ iff all deadlock free compositions of M with $\mathcal{E}$ that is independent of H satisfy ψ, thus iff no strategy induces a computation tree that does not satisfy ψ, thus iff $\mathcal{A}_{M,\neg\psi}$ is empty.

We now turn to a detailed description of the solution of the module-checking problem with incomplete information, and the complexity results it entails. For that, we first define formally alternating tree automata.

5.1 Alternating Tree Automata

Alternating tree automata generalize nondeterministic tree automata and were first introduced in [MS87]. An alternating tree automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$ runs on full Σ-labeled Υ-trees (for an agreed set Υ of directions). It consists of a finite set Q of states, an initial state $q_0 \in Q$, a transition function δ, and an acceptance condition α (a condition that defines a subset of Q^ω).

For a set Υ of directions, let $\mathcal{B}^+(\Upsilon \times Q)$ be the set of positive Boolean formulas over $\Upsilon \times Q$; i.e., Boolean formulas built from elements in $\Upsilon \times Q$ using $\wedge$ and $\vee$, where we also allow the formulas **true** and **false** and, as usual, $\wedge$ has precedence over $\vee$. The transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(\Upsilon \times Q)$ maps a state and an input letter to a formula that suggests a new configuration for the automaton. For example, when $\Upsilon = \{0, 1\}$, having

$$\delta(q, \sigma) = (0, q_1) \wedge (0, q_2) \vee (0, q_2) \wedge (1, q_2) \wedge (1, q_3)$$

means that when the automaton is in state q and reads the letter σ, it can either send two copies, in states q_1 and q_2, to direction 0 of the tree, or send a copy in state q_2 to direction 0 and two copies, in states q_2 and q_3, to direction 1. Thus, unlike nondeterministic tree automata, here the transition function may require the automaton to send several copies to the same direction or allow it not to send copies to all directions.

A *run of an alternating automaton* $\mathcal{A}$ on an input Σ-labeled Υ-tree $\langle T, V \rangle$ is a tree $\langle T_r, r \rangle$ in which the root is labeled by q_0 and every other node is labeled by an element of $\Upsilon^* \times Q$. Each node of T_r corresponds to a node of T. A node in T_r, labeled by (x, q), describes a copy of the automaton that reads the node x of T and visits the state q. Note that many nodes of T_r can correspond to the same node of T; in contrast, in a run of a nondeterministic automaton on $\langle T, V \rangle$ there is a one-to-one correspondence between the nodes of the run and the nodes of the tree. The labels of a node and its children have to satisfy the transition function. For example, if $\langle T, V \rangle$ is a $\{0, 1\}$-tree with $V(\epsilon) = a$ and $\delta(q_0, a) = ((0, q_1) \vee (0, q_2)) \wedge ((0, q_3) \vee (1, q_2))$, then the nodes of $\langle T_r, r \rangle$ at level 1 include the label $(0, q_1)$ or $(0, q_2)$, and include the label $(0, q_3)$ or $(1, q_2)$. Each infinite path ρ in $\langle T_r, r \rangle$ is labeled by a word $r(\rho)$ in Q^ω. Let $inf(\rho)$ denote the set of states in Q that appear in $r(\rho)$ infinitely often. A run $\langle T_r, r \rangle$ is accepting iff all its infinite paths satisfy the acceptance condition. In *Büchi* alternating tree automata, $\alpha \subseteq Q$, and an infinite path ρ satisfies α iff $inf(\rho) \cap \alpha \neq \emptyset$. As with nondeterministic automata, an automaton accepts a tree iff there exists an accepting run on it. We denote by $\mathcal{L}(\mathcal{A})$ the language of the automaton $\mathcal{A}$; i.e., the set of all labeled trees that $\mathcal{A}$ accepts. We say that an automaton is *nonempty* iff $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

We define the *size* $|\mathcal{A}|$ of an alternating automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$ as $|Q| + |\alpha| + |\delta|$, where $|Q|$ and $|\alpha|$ are the respective cardinalities of the sets Q and α, and where $|\delta|$ is the sum of the lengths of the satisfiable (i.e., not **false**) formulas that appear as $\delta(q, \sigma)$ for some q and σ.

5.2 Solving the Problem of Module-Checking with Incomplete Information

Theorem 5. [KV97] *Given a module M and a CTL formula ψ over the sets I, O, and H, of M's variables, there exists an alternating Büchi tree automaton $\mathcal{A}_{M,\psi}$ over $\{\top, \bot\}$-labeled $2^{I \cup O}$-trees, of size $O(|M| * |\psi|)$, such that $\mathcal{L}(\mathcal{A}_{M,\psi})$ is exactly the set of strategies $\mathcal{E}$ such that $M \lhd \mathcal{E}$ is deadlock free and satisfies ψ.*

Proof (sketch): Let $M = \langle I, O, H, W, w_0, R, L \rangle$, and let $K = I \cup O$. For $w \in W$ and $\upsilon \in 2^K$, we define $s(w, \upsilon) = \{w' \mid \langle w, w' \rangle \in R \text{ and } L(w') \cap K = \upsilon\}$ and $d(w) = \{\upsilon \mid s(w, \upsilon) \neq \emptyset\}$.

That is, $s(w, \upsilon)$ contains all the successors of w that agree in their readable variables with υ. Each such successor corresponds to a node in $\langle T_M, V_M\rangle$ with a direction in $hide^{-1}_{(2^H)}(\upsilon)$. Accordingly, $d(w)$ contains all directions υ for which nodes corresponding to w in $\langle T_M, V_M\rangle$ have at least one successor with a direction in $hide^{-1}_{(2^H)}(\upsilon)$.

Essentially, the automaton $\mathcal{A}_{M,\psi}$ is similar to the product alternating tree automaton obtained in the alternating-automata theoretic framework for CTL model checking [BVW94]. There, as there is a single computation tree with respect to which the formula is checked, the automaton obtained is a 1-letter automaton. Here, as there are many computation trees to check, we get a 2-letter automaton: each $\{\top, \bot\}$-labeled tree induces a different computation tree, and $\mathcal{A}_{M,\psi}$ considers them all. In addition, it checks that the composition of the strategy in the input with M is deadlock free. We assume that ψ is given in a positive normal form, thus negations are applied only to atomic propositions. We define $\mathcal{A}_{M,\psi} = \langle\{\top, \bot\}, Q, q_0, \delta, \alpha\rangle$, where

- $Q = (W \times (cl(\psi) \cup \{p_\top\}) \times \{\forall, \exists\}) \cup \{q_0\}$, where $cl(\psi)$ denotes the set of ψ's subformulas. Intuitively, when the automaton is in state $\langle w, \varphi, \forall\rangle$, it accepts all strategies for which w is either pruned or satisfies φ, where $\varphi = p_\top$ is satisfied iff the root of the strategy is labeled $\top$. When the automaton is in state $\langle w, \varphi, \exists\rangle$, it accepts all strategies for which w is not pruned and it satisfies φ. We call $\forall$ and $\exists$ the *mode* of the state. While the states in $W \times \{p_\top\} \times \{\forall, \exists\}$ check that the composition of M with the strategy in the input is deadlock free, the states in $W \times cl(\psi) \times \{\forall, \exists\}$ check that this composition satisfies ψ. The initial state q_0 sends copies to check both the deadlock freeness of the composition and the satisfaction of ψ.
- The transition function $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+(2^K \times Q)$ is defined as follows (with $m \in \{\exists, \forall\}$).
  - $\delta(q_0, \bot) =$ **false**, and $\delta(q_0, \top) = \delta(\langle w_0, p_\top, \exists\rangle, \top) \wedge \delta(\langle w_0, \psi, \exists\rangle, \top)$.
  - For all w and φ, we have $\delta(\langle w, \varphi, \forall\rangle, \bot) =$ **true** and $\delta(\langle w, \varphi, \exists\rangle, \bot) =$ **false**.
  - $\delta(\langle w, p_\top, m\rangle, \top) =$
  $$(\bigvee_{\upsilon \in 2^K} \bigvee_{w' \in s(w,\upsilon)} (\upsilon, \langle w', p_\top, \exists\rangle)) \wedge (\bigwedge_{\upsilon \in 2^K} \bigwedge_{w' \in s(w,\upsilon)} (\upsilon, \langle w', p_\top, \forall\rangle)).$$
  - $\delta(\langle w, p, m\rangle, \top) =$ **true** if $p \in L(w)$, and $\delta(\langle w, p, m\rangle, \top) =$ **false** if $p \notin L(w)$.
  - $\delta(\langle w, \neg p, m\rangle, \top) =$ **true** if $p \notin L(w)$, and $\delta(\langle w, \neg p, m\rangle, \top) =$ **false** if $p \in L(w)$.
  - $\delta(\langle w, \varphi_1 \wedge \varphi_2, m\rangle, \top) = \delta(\langle w, \varphi_1, m\rangle, \top) \wedge \delta(\langle w, \varphi_2, m\rangle, \top)$.
  - $\delta(\langle w, \varphi_1 \vee \varphi_2, m\rangle, \top) = \delta(\langle w, \varphi_1, m\rangle, \top) \vee \delta(\langle w, \varphi_2, m\rangle, \top)$.
  - $\delta(\langle w, AX\varphi, m\rangle, \top) = \bigwedge_{\upsilon \in 2^K} \bigwedge_{w' \in s(w,\upsilon)} (\upsilon, \langle w', \varphi, \forall\rangle)$.
  - $\delta(\langle w, EX\varphi, m\rangle, \top) = \bigvee_{\upsilon \in 2^K} \bigvee_{w' \in s(w,\upsilon)} (\upsilon, \langle w', \varphi, \exists\rangle)$.
  - $\delta(\langle w, A\varphi_1 U \varphi_2, m\rangle, \top) =$
  $$\delta(\langle w, \varphi_2, m\rangle, \top) \vee (\delta(\langle w, \varphi_1, m\rangle, \top) \wedge \bigwedge_{\upsilon \in 2^K} \bigwedge_{w' \in s(w,\upsilon)} (\upsilon, \langle w', A\varphi_1 U \varphi_2, \forall\rangle)).$$
  - $\delta(\langle w, E\varphi_1 U \varphi_2, m\rangle, \top) =$
  $$\delta(\langle w, \varphi_2, m\rangle, \top) \vee (\delta(\langle w, \varphi_1, m\rangle, \top) \wedge \bigvee_{\upsilon \in 2^K} \bigvee_{w' \in s(w,\upsilon)} (\upsilon, \langle w', E\varphi_1 U \varphi_2, \exists\rangle)).$$
  - $\delta(\langle w, AG\varphi, m\rangle, \top) = \delta(\langle w, \varphi, m\rangle, \top) \wedge \bigwedge_{\upsilon \in 2^K} \bigwedge_{w' \in s(w,\upsilon)} (\upsilon, \langle w', AG\varphi, \forall\rangle)$.
  - $\delta(\langle w, EG\varphi, m\rangle, \top) = \delta(\langle w, \varphi, m\rangle, \top) \wedge \bigvee_{\upsilon \in 2^K} \bigvee_{w' \in s(w,\upsilon)} (\upsilon, \langle w', EG\varphi, \exists\rangle)$.

  Consider, for example, a transition from the state $\langle w, AX\varphi, \exists\rangle$. First, if the transition to w is disabled (that is, the automaton reads $\bot$), then, as the current mode is existential, the run is rejecting. If the transition to w is enabled, then w's successors that are enabled should satisfy φ. The state w may have several successors that agree on some labeling $\upsilon \in 2^K$ and differ only on the labeling of variables in H. These successors are indistinguishable by the environment, and the automaton sends them all to the same direction υ. This guarantees that either all these successors are enabled by the strategy (in case the letter to be read in direction υ is $\top$) or all are disabled (in case the letter in direction υ is $\bot$). In addition, since the requirement to satisfy φ concerns only successors of w that are enabled, the mode of the new states is universal. The copies of $\mathcal{A}_{M,\psi}$ that check the composition with the strategy to be deadlock free guarantee that at least one successor of w is enabled. Note that as the transition relation R is total, the conjunctions and disjunctions in δ cannot be empty.

- $\alpha = W \times G(\psi) \times \{\exists, \forall\}$, where $G(\psi)$ is the set of all formulas of the form $AG\varphi$ or $EG\varphi$ in $cl(\psi)$. Thus, while the automaton cannot get trapped in states associated with "Until-formulas" (then, the eventuality of the until is not satisfied), it may get trapped in states associated with "Always-formulas" (then, the safety requirement is never violated).

We now consider the size of $\mathcal{A}_{M,\neg\psi}$. Clearly, $|Q| = O(|W| * |\psi|)$. Also, as the transition associated with a state $\langle w, \varphi, m\rangle$ depends on the successors of w, we have that $|\delta| = O(|R| * |\psi|)$. Finally, $|\alpha| \leq |Q|$, and we are done. □

Extending the alternating automata described in [BVW94] to handle incomplete information is possible thanks to the special structure of the automata, which alternate between universal and existential modes. This structure (the "hesitation condition", as called in [BVW94]) exists also in automata associated with CTL* formulas, and imply the following analogous theorem.

Theorem 6. [KV97] *Given a module M and a CTL* formula ψ over the sets I, O, and H, of M's variables, there exists an alternating Rabin tree automaton $\mathcal{A}_{M,\psi}$ over $\{\top, \bot\}$-labeled $2^{I\cup O}$-trees, with $|W| * 2^{O(|\psi|)}$ states and two pairs, such that $\mathcal{L}(\mathcal{A}_{M,\psi})$ is exactly the set of strategies $\mathcal{E}$ such that $M \lhd \mathcal{E}$ is deadlock free and satisfies ψ.*

We now consider the complexity bounds that follow from our algorithm.

Theorem 7. [KV97] *The module-checking problem with incomplete information is EXPTIME-complete for CTL and is 2EXPTIME-complete for CTL*.*

Proof (sketch): The lower bounds follows from the known bounds for module checking with complete information [KV96]. For the upper bounds, in Theorems 5 and 6 we reduced the problem $M \models_r \psi$ to the problem of checking the nonemptiness of the automaton $\mathcal{A}_{M,\neg\psi}$. When ψ is a CTL formula, $\mathcal{A}_{M,\neg\psi}$ is an alternating Büchi automaton of size $O(|M| * |\psi|)$. By [VW86b, MS95], checking the nonemptiness of $\mathcal{A}_{M,\neg\psi}$ is then exponential in the sizes of M and ψ. When ψ is a CTL* formula, the automaton $\mathcal{A}_{M,\neg\psi}$ is an alternating Rabin automaton, with $|W| * 2^{O(|\psi|)}$ states and two pairs. Accordingly, by [EJ88, MS95], checking the nonemptiness of $\mathcal{A}_{M,\neg\psi}$ is exponential in $|W|$ and double exponential in $|\psi|$. □

As the module-checking problem for CTL is already EXPTIME-hard for environments with complete information, it might seem as if incomplete information can be handled at no cost. This is, however, not true. By Theorem 3, the program complexity of CTL module checking with complete information is PTIME-complete. On the other hand, the time complexity of the algorithm we present here is exponential in the size of the both the formula and the system. Can we do better? In Theorem 8 below, we answer this question negatively. To see why, consider a module M with hidden variables. When M interacts with an environment $\mathcal{E}$, the module seen by $\mathcal{E}$ is different from M. Indeed, every state of the module seen by $\mathcal{E}$ corresponds to a set of states of M. Therefore, coping with incomplete information involves some subset construction, which blows-up the state space exponentially. In our algorithm, the subset construction hides in the emptiness test of $\mathcal{A}_{M,\neg\psi}$.

Theorem 8. [KV97] *The program complexity of CTL module checking with incomplete information is EXPTIME-complete.*

Proof (sketch): The upper bound follows from Theorem 7. For the lower bound, we do a reduction from the outcome problem for two-players games with incomplete information, proved to be EXPTIME-hard in [Rei84]. A two-player game with incomplete information consists of an AND-OR graph with an initial state and a set of designated states. Each of the states in the graph is labeled by readable and unreadable observations. The game is played between two players, called the OR-player and the AND-player. The two players generate together a path in the graph. The path starts at the initial state. Whenever the game is at an OR-state, the OR-player determines the next state. Whenever the game is at an AND-state, the AND-player determines the next state. The outcome problem is to determine whether the OR-player has a strategy that depends only on the readable observations (that is, a strategy that maps finite sequences of sets of readable observations to a set of known observations) such that following this strategy guarantees that, no matter how the AND-player plays, the path eventually visits one of the designated states.

Given an AND-OR graph G as above, we define a module M_G such that M_G reactively satisfies a fixed CTL formula φ iff the OR-player has no strategy as above. The environments of M_G correspond to strategies for the OR-player. Each environment suggests a pruning of $\langle T_{M_G}, V_{M_G} \rangle$ such that the set of paths in the pruned tree corresponds to a set of paths that the OR-player can force the game into, no matter how the AND-player plays. The module M_G is very similar to G, and the formula φ requires the existence of a computation that never visits a designated state. The formal definition of M_G and φ involves some technical complications required in order to make sure that the environment disables only transitions from OR-states. $\square$

6 Discussion

The discussion of the relative merits of linear versus branching temporal logics is almost as early as these paradigms [Lam80]. 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 computational complexity of CTL model checking have compensated for its lack of expressive power and branching-time model-checking tools can handle systems with extremely large state spaces [BCM+90, McM93, CGL93].

If we examine this issue more closely, however, we find that the computational superiority of CTL over LTL is not that clear. For example, 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 discussed in this paper questions the computational superiority of the branching-time paradigm further.

Our conclusion is that the debate about the relative merit of the linear and branching paradigms will not be settled by technical arguments such as expressive power or computational complexity. Rather, the discussion should focus on the attractiveness of the approaches to practitioners who practice computer-aided verification in realistic settings. We believe that this discussion will end up with the conclusion that both approaches have their merits and computer-aided verification tools should therefore combine the two approaches rather than "religiously" adhere to one or the other.

References

[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.

[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.

[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.

[CGB86] E.M. Clarke, O. Grumberg, and M.C. Browne. Reasoning about networks with many identical finite-state processes. In *Proc. 5th ACM Symposium on Principles of Distributed Computing*, pages 240–248, Calgary, Alberta, August 1986.

[CGH+95] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[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.

[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*, pages 368–377, White Plains, October 1988.

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

[Eme85] E.A. Emerson. Automata, tableaux, and temporal logics. In *Proc. Workshop on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 79–87. Springer-Verlag, 1985.

[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. *J. of Computer and Systems Sciences*, 18:194–211, 1979.

[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.

[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.

[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.

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

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

[KV97] O. Kupferman and M.Y. Vardi. Module checking revisited. In *Computer Aided Verification, Proc. 9th Int. Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 1997.

[KVW97] O. Kupferman, M.Y. Vardi, and P. Wolper. Module checking. submitted for publication, 1997.

[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.

[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.

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

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.

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

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54,:267–276, 1987.

[MS95] D.E. Muller and P.E. Schupp. Simulating aternating tree automata by nondeterministic automata: New results and new proofs of theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141:69–107, 1995.

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

[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, January 1989.

[Pra80] V.R. Pratt. A near-optimal method for reasoning about action. *J. on 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.

[Rei84] J.H. Reif. The complexity of two-player games of incomplete information. *J. on Computer and System Sciences*, 29:274–301, 1984.

[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.