The Bulletin of Symbolic Logic
Volume 5, Number 2, June 1999

CHURCH'S PROBLEM REVISITED

ORNA KUPFERMAN AND MOSHE Y. VARDI

Abstract. In program synthesis, we transform a specification into a system that is guaranteed to satisfy the specification. When the system is open, then at each moment it reads input signals and writes output signals, which depend on the input signals and the history of the computation so far. The specification considers all possible input sequences. Thus, if the specification is linear, it should hold in every computation generated by the interaction, and if the specification is branching, it should hold in the tree that embodies all possible input sequences.

Often, the system cannot read all the input signals generated by its environment. For example, in a distributed setting, it might be that each process can read input signals of only part of the underlying processes. Then, we should transform a specification into a system whose output depends only on the readable parts of the input signals and the history of the computation. This is called *synthesis with incomplete information*. In this work we solve the problem of synthesis with incomplete information in its full generality. We consider linear and branching settings with complete and incomplete information. We claim that *alternation* is a suitable and helpful mechanism for coping with incomplete information. Using *alternating tree automata*, we show that incomplete information does not make the synthesis problem more complex, in both the linear and the branching paradigm. In particular, we prove that independently of the presence of incomplete information, the synthesis problems for CTL and CTL* are complete for EXPTIME and 2EXPTIME, respectively.

§1. Introduction. In computer system design, we distinguish between *closed* and *open* systems [13]. 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. In *program synthesis*, we transform a specification into a program that is guaranteed to satisfy the specification. Earlier works on synthesis consider *closed systems*. There, a program that meets the specification can be extracted from a constructive proof that the formula is satisfiable [20, 8]. As argued in [5, 25, 1], such synthesis paradigms are

Received September 4, 1998; revised March 13, 1999.

A preliminary version, titled "Synthesis with incomplete informatio", appeared in the Proceedings of the 2nd International Conference on Temporal Logic.

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

© 1999, Association for Symbolic Logic
1079-8986/99/0502-0004/$2.90

not of much interest when applied to *open systems*, which interact with an environment. Consider for example a scheduler for a printer that serves two users. The scheduler is an open system. Each time unit it reads the input signals $J1$ and $J2$ (a job sent from the first or the second user, respectively), and it writes the output signals $P1$ and $P2$ (print a job of the first or the second user, respectively). The scheduler should be designed so that jobs of the two users are not printed simultaneously, and whenever a user sends a job, the job is printed eventually. Of course, this should hold no matter how the users send jobs.

We can specify the required behaviors of closed and open systems using *temporal logic* [24]. In a temporal logic, we augment a conventional logic with temporal modalities, making it possible to describe an occurrence of events in time. For example, using the temporal modalities *always* and *eventually*, we can specify the behavior "whenever $J1$ holds, $P1$ eventually holds too". Two possible views regarding the nature of time induce two types of temporal logics [19]. In *linear* temporal logics, such as LTL [23], time is treated as if each moment in time has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and we regard them as describing a behavior of a single computation of a system. In *branching* temporal logics, such as CTL and CTL* [9, 10], each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted can be viewed as infinite trees, each describing the behavior of the possible computations of a nondeterministic system.

The requirement for the scheduler can be specified in terms of a an LTL formula ψ. Satisfiability of ψ does not imply that a required scheduler exists. To see this, observe that ψ is satisfied in every structure in which the four signals never hold. In addition, an evidence to ψ's satisfiability is not of much help in extracting a correct scheduler. Indeed, while such an evidence only suggests a scheduler that is guaranteed to satisfy ψ for *some* input sequence, we want a scheduler that satisfies ψ for *all* possible scripts of jobs sent to the printer. We now make this intuition more formal. Given sets I and O of input and output signals, respectively, we can view a program as a *strategy* $P\colon (2^I)^* \to 2^O$ that maps a finite sequence of sets of input signals into a set of output signals. When P interacts with an environment that generates infinite input sequences, it associates with each input sequence an infinite computation over $2^{I\cup O}$. Given an LTL formula ψ over $I \cup O$, *realizability* of ψ is the problem of determining whether there exists a program P all of whose computations satisfy ψ. Synthesis of ψ then amounts to constructing such P [25].

The linear paradigm for realizability and synthesis is closely related to *Church's solvability problem* [4]. There, we are given a relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$ that is definable in the sequential calculus (S1S) and we seek a function

$f: (2^I)^\omega \to (2^O)^\omega$, generated by a strategy, such that for all $x \in (2^I)^\omega$, we have $R(x, f(x))$. We can view the relation R as a linear specification for the program: it defines all the permitted pairs of input and output sequences. A function f as above then maps every possible input sequence into a permitted output sequence, and can be therefore viewed as a correct program. The solutions to Church's problem and the LTL synthesis problem are similar [28, 25], and consist of a reduction to the nonemptiness problem of *tree automata* (an earlier and more complicated solution can be found in [3]).

Though the program P is deterministic, it induces a computation tree. The branches of the tree correspond to external nondeterminism, caused by different possible inputs. Thus, the tree has a fixed branching degree $|2^I|$, and it embodies all the possible inputs (and hence also computations) of P. When we synthesize P from an LTL specification ψ, we require ψ to hold in all the paths of P's computation tree. Consequently, we cannot impose possibility requirements on P [19, 10]. In the scheduler example, while we can require, for instance, that for every infinite sequence of inputs a job of the first user is eventually printed, we cannot require that every finite sequence of inputs *can be* extended so that a job of the first user is eventually printed. In order to express possibility properties, we should specify P using *branching temporal logics*, which enable both universal and existential path quantification [7]. Given a branching specification ψ over $I \cup O$, realizability of ψ is the problem of determining whether there exists a program P whose computation tree satisfies ψ. Synthesis of ψ then amounts to constructing such P.

So far, we considered the case where the specifications (either linear or branching) refer solely to signals in I and O, both are known to P. This is called synthesis with *complete information*. Often, the program does not have complete information about its environment. For example, in a distributed setting, it might be that each process can read input signals of only part of the underlying processes. Let E be the set of input signals that the program can not read. In the scheduler example, we take $E = \{B1, B2\}$, where $B1$ holds when the job sent to the printer by the first user is a paper containing a bug, and similarly for $B2$. Unfortunately, while the scheduler can see whether the users send jobs, it cannot trace bugs in their papers.

Since P cannot read the signals in E, its activity is independent of them. Hence, it can still be viewed as a strategy $P: (2^I)^* \to 2^O$. Nevertheless, the computations of P are now infinite words over $2^{I \cup E \cup O}$. Similarly, embodying all the possible inputs to P, the computation tree induced by P now has a fixed branching degree $|2^{I \cup E}|$ and it is labeled by letters in $2^{I \cup E \cup O}$. Note that different nodes in this tree may have, according P's incomplete information, the same "history of inputs". In the scheduler example, P cannot distinguish between two nodes that the input sequences leading to them differ only in the values of $B1$ and $B2$ in some points along them.

Often, programs need to satisfy specifications that refer to signals they cannot read. For example, following several events, it was decided to change the specification for the printer scheduler so that if the first user sends to the printer an erroneous paper, then the paper is never printed. The new scheduler needs to satisfy the specification even though it cannot trace bugs. This problem, of synthesis with *incomplete information*, is the subject of this work. Formally, given a specification ψ over the sets I, E, and O of readable input, unreadable input, and output signals, respectively, synthesis with incomplete information amounts to constructing a program $P\colon (2^I)^* \to 2^O$, which is independent of E, and which realizes ψ (that is, if ψ is linear then all the computations of P satisfy ψ, and if ψ is branching then the computation tree of P satisfies ψ).

It is known how to cope with incomplete information in the linear paradigm. In particular, the approach used in [25] can be extended to handle LTL synthesis with incomplete information. Essentially, nondeterminism of the automata can be used to guess the missing information, making sure that no guess violates the specification [31]. Similarly, methods for control and synthesis in other linear paradigms (e.g., when specifying terminating programs by regular languages) have been extended to handle incomplete information [14, 15].

Coping with incomplete information is more difficult in the branching paradigm. The methods used in the linear paradigm are not applicable here. To see why, let us consider first realizability with complete information. There, recall, we are given a branching specification ψ over $2^{I\cup O}$, and we check whether there exists a program P whose computation tree satisfies ψ. In other words, we check whether we can take the I*-exhaustive* tree (i.e., the 2^I-labeled tree that embodies all input sequences) and annotate it with outputs so that the resulted $2^{I\cup O}$-labeled tree satisfies ψ. This problem can be easily solved using tree automata. Essentially, we first construct a tree automaton $\mathcal{A}_\psi$ that accepts exactly all $2^{I\cup O}$-labeled trees that satisfy ψ [6, 33]. Then, we construct a tree automaton $\mathcal{A}_P$ that accepts all the "potential" computation trees (i.e., all $2^{I\cup O}$-labeled trees obtained by annotating the I-exhaustive tree with outputs). Finally, we check that the intersection of the two automata is nonempty. Can we follow the same lines in the presence of incomplete information? Now, that ψ is defined over $I \cup E \cup O$, the automaton $\mathcal{A}_\psi$ should accept all the $2^{I\cup E\cup O}$-labeled trees that satisfy ψ. This causes no difficulty. In addition, since the environment produces signals in both I and E, the automaton $\mathcal{A}_P$ should consider annotations of the $(I\cup E)$-exhaustive tree. Unlike, however, in the case of complete information, here not all annotations induce potential computation trees. Since P is independent of signals in E, an annotation induces a potential computation tree only if every two nodes that have the same history (according to P's incomplete information) are annotated with the same output! This consistency condition is non-regular and cannot be checked by an automaton [30]. It is this need, to restrict the set of candidate computation trees to trees that meet some non-regular condition, that makes incomplete information in the branching paradigm so challenging.[1]

In this paper we solve the problem of synthesis with incomplete information for the branching paradigm, which we show to be a proper extension of the linear paradigm and the complete-information paradigm. We claim that *alternation* is a suitable and helpful mechanism for coping with incomplete information. Using *alternating tree automata*, we show that incomplete information does not make the synthesis problem more complex, in both the linear and the branching paradigm. In fact, as alternating tree automata shift all the combinatorial difficulties of the synthesis problem to the nonemptiness test, the automata-based algorithms that we describe are as simple as the known automata-based algorithms for the satisfiability problem. In particular, we prove that independently of the presence of incomplete information, the synthesis problems for CTL and CTL* are complete for EXPTIME and 2EXPTIME, respectively. These results join the 2EXPTIME-complete bound for LTL synthesis in both settings [25, 29, 31]. Keeping in mind that the satisfiability problems for LTL, CTL, and CTL* are complete for PSPACE, EXPTIME, and 2EXPTIME [7], it follows that while the transition from closed to open systems dramatically increases the complexity of synthesis in the linear paradigm, it does not seem to influence the complexity in the branching paradigm.

§2. Preliminaries.

2.1. Trees and labeled trees. Given a finite set Υ, an Υ-*tree* is a set $T \subseteq \Upsilon^*$ such that if $x \cdot \upsilon \in T$, where $x \in \Upsilon^*$ and $\upsilon \in \Upsilon$, then also $x \in T$. 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 $x \in T$, the nodes $x \cdot \upsilon \in T$ where $\upsilon \in \Upsilon$ are the *children* of x. Each node x of T has a *direction* in Υ. The direction of the root is υ^0, for some designated $\upsilon^0 \in \Upsilon$, called the *root direction*. The direction of a node $x \cdot \upsilon$ is υ. We denote by $\mathrm{dir}(x)$ the direction of node x. An Υ-tree T is a *full infinite tree* if $T = \Upsilon^*$. Unless otherwise mentioned, we consider here full infinite trees. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$ there exists a unique $\upsilon \in \Upsilon$ such that $x \cdot \upsilon \in \pi$. For a path π and $j \geq 0$, let π_j denote the node of length j in π. Each path $\pi \subseteq T$ corresponds to a word $T(\pi) = \mathrm{dir}(\pi_0) \cdot \mathrm{dir}(\pi_1) \cdots$ in Υ^ω.

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

[1]Another way to look at is that while in the linear paradigm it is easy for a nondeterministic automaton to choose a guessed annotation, in the branching paradigm the chosen annotation should specify a condition that cannot be specified in by an automaton [12].

Σ are not important or clear from the context, we call $\langle T, V\rangle$ a labeled tree. Each path π in $\langle T, V\rangle$ corresponds to a word $V(\pi) = V(\pi_0)\cdot V(\pi_1)\cdots$ in Σ^ω. Of special interest to us are labeled trees in which $\Sigma = 2^{AP}$ for some set AP of atomic propositions. We call such labeled trees *computation trees*. For a Σ-labeled Υ-tree $\langle T, V\rangle$, we define the *x-ray* of $\langle T, V\rangle$, denoted $\mathrm{xray}(\langle T, V\rangle)$, as the $(\Upsilon \times \Sigma)$-labeled Υ-tree $\langle T, V'\rangle$ in which each node is labeled by both its direction and its labeling in $\langle T, V\rangle$. Thus, for every $x \in T$, we have $V'(x) = \langle \mathrm{dir}(x), V(x)\rangle$. Essentially, the labels in $\mathrm{xray}(\langle T, V\rangle)$ contain information not only about the surface of $\langle T, V\rangle$ (its labels) but also about its skeleton (its nodes).

Let $\Upsilon_2 = \{0, 1\}$ and $\Upsilon_4 = \{00, 01, 10, 11\}$. When $T = \Upsilon_2^*$, we say that T is a full infinite binary tree (*binary tree*, for short). We refer to the child $x \cdot 0$ of x as the left child and refer to the child $x \cdot 1$ as the right child. When $T = \Upsilon_4^*$, we say that T is a full infinite 4-ary tree (*4-ary tree*, for short). We refer to $x \cdot 00$ as the leftmost child, $x \cdot 01$ is its brother to the right, then comes $x \cdot 10$, and $x \cdot 11$ is the rightmost child (see Figures 1 and 2).

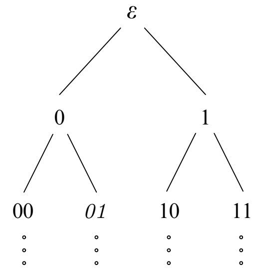

FIGURE 1. A binary tree.

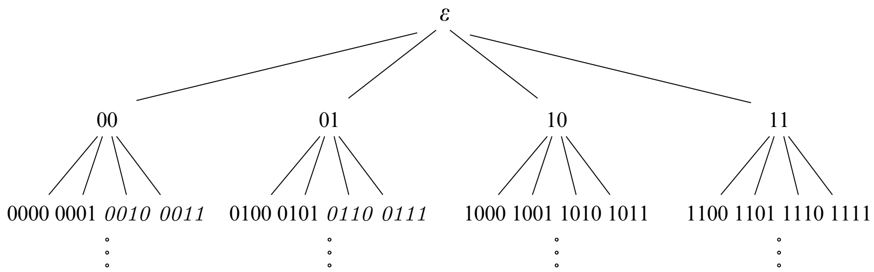

FIGURE 2. A 4-ary tree.

Consider a node $x \in \Upsilon_4^*$. Recall that each letter in Υ_4 is a pair c_1c_2 of letters in Υ_2. Given x, let $\mathrm{hide}(x)$ be the node in Υ_2^* obtained from x by replacing each letter c_1c_2 by the letter c_1. For example, the node 0010 of the 4-ary tree in the figure corresponds, by hide, to the node 01 of the binary tree.

Note that the nodes 0011, 0110, and 0111 of the 4-ary tree also correspond, by hide, to the node 01 of the binary tree. We extend the hiding operator to paths $\pi \subset \Upsilon_4^*$ in the straightforward way. That is, the path hide$(\pi) \subset \Upsilon_2^\omega$ is obtained from π by replacing each node $x \in \pi$ by the node hide(x).

Let Z be a finite set. For a Z-labeled Υ_2-tree $\langle \Upsilon_2^*, V\rangle$, we define the *widening* of $\langle \Upsilon_2^*, V\rangle$, denoted wide$(\langle T, V\rangle)$, as the Z-labeled Υ_4-tree $\langle \Upsilon_4^*, V'\rangle$ where for every $x \in \Upsilon_4^*$, we have $V'(x) = V(\text{hide}(x))$. Note that for every node $x \in \Upsilon_4^*$, the children $x \cdot 00$ and $x \cdot 01$ of x agree on their label in $\langle \Upsilon_4^*, V'\rangle$. Indeed, they are both labeled with $V(\text{hide}(x) \cdot 0)$. Similarly, the children $x \cdot 10$ and $x \cdot 11$ are both labeled with $V(\text{hide}(x) \cdot 1)$. The essence of widening is that for every path π in $\langle \Upsilon_2^*, V\rangle$ and for every path $\rho \in \text{hide}^{-1}(\pi)$, the path ρ exists in $\langle \Upsilon_4^*, V'\rangle$ and $V(\pi) = V'(\rho)$. In other words, for every two paths ρ_1 and ρ_2 in $\langle \Upsilon_4^*, V'\rangle$ such that hide$(\rho_1) =$ hide$(\rho_2) = \pi$, we have $V'(\rho_1) = V'(\rho_2) = V(\pi)$.

Given finite sets X and Y, we generalize the operators hide and wide to Z-labeled $(X \times Y)$-trees by parameterizing them with the set Y as follows (the operators defined above correspond to the special case where $X = Y = \Upsilon_2$). The operator hide$_Y\colon (X \times Y)^* \to X^*$ replaces each pair $\langle x, y\rangle$ by the letter x. Accordingly, the operator wide$_Y$ maps Z-labeled X-trees to Z-labeled $(X \times Y)$-trees. Formally,

$$\text{wide}_Y(\langle X^*, V\rangle) = \langle (X \times Y)^*, V'\rangle,$$

where for every node $w \in (X \times Y)^*$, we have $V'(w) = V(\text{hide}_Y(w))$.

2.2. Temporal logic. The logic LTL is a linear temporal logic. Formulas of LTL are built from a set AP of atomic proposition using the usual Boolean operators and the temporal operators X ("next time"), G ("always"), and U ("until"). We present here a positive normal form in which negation may be applied only to atomic propositions. Given a set AP, an LTL formula is defined as follows:

- ***true***, ***false***, p, or $\neg p$, for $p \in AP$.
- $\psi \vee \varphi$, $\psi \wedge \varphi$, $X\psi$, $G\psi$, or $\psi U \varphi$, where ψ and φ are LTL formulas.

We define the semantics of LTL with respect to a path π of a computation tree $\langle T, V\rangle$ and a position of π. We use $\pi, j \models \psi$ to indicate that an LTL formula ψ holds in the j's position of the path π. The relation $\models$ is inductively defined as follows (assuming an agreed tree $\langle T, V\rangle$:

- For all π and j, we have that $\pi, j \models$ ***true*** and $\pi, j \not\models$ ***false***.
- For an atomic proposition $p \in AP$, we have $\pi, j \models p$ if and only if $p \in \pi_j$ and $\pi, j \models \neg p$ if and only if $p \notin V(\pi_j)$.
- $\pi, j \models \psi \vee \varphi$ if and only if $\pi, j \models \psi$ or $\pi, j \models \varphi$.
- $\pi, j \models \psi \wedge \varphi$ if and only if $\pi, j \models \psi$ and $\pi, j \models \varphi$.
- $\pi, j \models X\psi$ if and only if $\pi, j + 1 \models \psi$.
- $\pi, j \models G\psi$ if and only if for every $k \geq j$, we have $\pi, k \models \psi$.

- $\pi, j \models \psi U \varphi$ if and only if there exists $k \geq j$ such that $\pi, k \models \varphi$ and $\pi, i \models \psi$ for all $j \leq i < k$.

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 (again, in a positive normal form) is either:

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

A CTL* path formula is either:

- A CTL* state formula.
- $\psi \vee \varphi$, $\psi \wedge \varphi$, $X\psi$, $G\psi$, or $\psi U \varphi$, where ψ and φ 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 X, G, and U must be immediately preceded by a path quantifier. Formally, CTL is the subset of CTL* obtained by restricting the path formulas to be $X\psi$, $G\psi$, or $\psi U \varphi$, where ψ and φ are CTL state formulas.

We denote the size of a CTL* formula ψ by $|\psi|$. The *closure* $\mathrm{cl}(\psi)$ of a CTL* formula ψ is the set of all state subformulas of ψ (including ψ but excluding ***true*** and ***false***). For example,

$$\mathrm{cl}(E(pU(AXq))) = \{E(pU(AXq)), p, AXq, q\}.$$

It is easy to see that the size of $\mathrm{cl}(\psi)$ is linear in the size of ψ.

We define the semantics of CTL* (and its sublanguage CTL) with respect to nodes in computation trees. We use $x \models \varphi$ to indicate that a state formula φ holds at node x (assuming an agreed computation tree $\langle T, V \rangle$). The relation $\models$ is inductively defined as follows (the relation $\pi, j \models \psi$ for a path formula ψ that is not a state formula is the same as for ψ in LTL).

- For all x, we have that $x \models$ ***true*** and $x \not\models$ ***false***.
- For an atomic proposition $p \in AP$, we have $x \models p$ if and only if $p \in V(x)$ and $x \models \neg p$ if and only if $p \notin V(x)$.
- $x \models \psi \vee \varphi$ if and only if $x \models \psi$ or $x \models \varphi$.
- $x \models \psi \wedge \varphi$ if and only if $x \models \psi$ and $x \models \varphi$.
- $x \models E\psi$ if and only if there exists a path π such that $\pi_j = x$ and $\pi, j \models \psi$.
- $x \models A\psi$ if and only if for all paths π with $\pi_j = x$, we have $\pi, j \models \psi$.
- $\pi, j \models \varphi$ for a state formula φ if and only if $\pi_j \models \varphi$.

§3. The problem. Consider a program P that interacts with its environment. Let I and E be the sets of input signals, readable and unreadable by

P, respectively, and let O be the set of P's output signals. We can view P as a *strategy* $P\colon (2^I)^* \to 2^O$. Indeed, P maps each finite sequence of sets of readable input signals into a set of output signals. Note that the information of P on its input is incomplete and it does not depend on the unreadable signals in E. We assume the following interaction between P and its environment. The interaction starts by P outputting $P(\varepsilon)$. The environment replies with some $\langle i_1, e_1\rangle \in 2^I \times 2^E$, to which P replies with $P(i_1)$. Interaction then continues step by step, with an output $P(i_1 \cdot i_2 \cdots i_j)$ corresponding to a sequence $\langle i_1, e_1\rangle \cdot \langle i_2, e_2\rangle \cdots \langle i_j, e_j\rangle$ of inputs. Thus, P associates with each infinite input sequence $\langle i_1, e_1\rangle \cdot \langle i_2, e_2\rangle \cdots$, an infinite computation

$$[P(\varepsilon)] \cdot [i_1 \cup e_1 \cup P(i_1)] \cdot [i_2 \cup e_2 \cup P(i_1 \cdot i_2)] \cdots$$

over $2^{I\cup E\cup O}$. We note that our choice of P starting the interaction is for technical convenience only.

Though the program P is deterministic, it induces a computation tree. The branches of the tree correspond to external nondeterminism, caused by different possible inputs. Thus, P is a $2^{I\cup E\cup O}$-labeled $2^{I\cup E}$-tree, where each node with direction $i \cup e \in I \cup E$ is labeled by $i \cup e \cup o$, where o is the set of output signals that P assigns to the sequence of readable inputs leading to the node. Formally, we obtain the computation tree of P by two transformations on the 2^O-labeled tree $\langle (2^I)^*, P\rangle$, which represents P. First, while $\langle (2^I)^*, P\rangle$ ignores the signals in E and the extra external nondeterminism induced by them, the computation tree of P, which embodies all possible computations, takes them into account. For that, we define the 2^O-labeled tree

$$\langle (2^{I\cup E})^*, P'\rangle = \mathrm{wide}_{(2^E)}(\langle (2^I)^*, P\rangle).$$

By the definition of the wide operator, each two nodes in $(2^{I\cup E})^*$ that correspond, according to P's incomplete information, to the same input sequence are labeled by P' with the same output. Now, as the signals in I and E are represented in $\langle (2^{I\cup E})^*, P'\rangle$ only in its nodes and not in its labels, we define the computation tree of P as

$$\langle (2^{I\cup E})^*, P''\rangle = \mathrm{xray}(\langle (2^{I\cup E})^*, P'\rangle).$$

Note that, as I, E, and O are disjoint, we refer to $\mathrm{wide}_{(2^E)}(\langle (2^I)^*, P\rangle)$ as a $2^{I\cup E}$-tree, rather than a $(2^I \times 2^E)$-tree. Similarly, $\mathrm{xray}(\langle (2^{I\cup E})^*, P'\rangle)$ is a $2^{I\cup E\cup O}$-labeled tree, rather than a $(2^{I\cup E} \times 2^O)$-labeled tree.

Given a CTL* formula ψ over the sets $I \cup E \cup O$ of atomic propositions, the problem of branching realizability with incomplete information is to determine whether there is a program P whose computation tree satisfies ψ. The synthesis problem requires the construction of such P.

It is easy to see that problem of branching synthesis with incomplete information is a proper extension of the problem of branching synthesis with complete information. We claim that it is also a proper extension of the linear synthesis problem with incomplete (and hence also complete) information [25, 27, 31]. For that, we show that synthesis for an LTL formula φ can be reduced to synthesis for the CTL* formula $A\varphi$. Moreover, φ and $A\varphi$ are realized by the same programs. Consider a program P. Assume first that P realizes $A\varphi$. Then, clearly, P associates with every input sequence a computation that satisfies φ, and thus it realizes φ as well. The second direction seems less immediate, but it follows easily from the fact that P is a (deterministic) strategy. Indeed, if P does not realize $A\varphi$, then there must be an input sequence with which P does no associate a computation satisfying φ.

In the next section we solve the general synthesis problem and we show that generalization comes at no cost. In more detail, we show that CTL* synthesis with incomplete information can be done in 2EXPTIME, matching the 2EXPTIME lower bound for LTL synthesis with complete information.

§4. The solution. Church's solvability problem and the LTL synthesis problem are solved, in [28, 25], using nondeterministic tree automata. Both solutions follow the same lines: we can translate the linear specification for the program into a word automaton, we can determinize the automaton and extend it to a tree automaton, and we can check the nonemptiness of this tree automaton when restricted to trees that embody all possible inputs. Such a method cannot be easily extended to handle synthesis with incomplete information in the branching paradigm. Here, we need to check the nonemptiness of a tree automaton when restricted not only to trees that embody all possible inputs, but also to trees in which the labels of the output signals are consistent. This condition is non-regular and cannot be checked by an automaton. In order to solve this difficulty, we need the framework of alternating tree automata.

4.1. Alternating tree automata. *Alternating tree automata* run on labeled trees. They generalize nondeterministic tree automata and were first introduced in [21]. For simplicity, we refer first to automata over infinite binary trees. Consider a nondeterministic tree automaton $\mathcal{A}$ with a set Q of states and transition function δ. The transition function δ maps an automaton state $q \in Q$ and an input letter $\sigma \in \Sigma$ to a set of pairs of states. Each such pair suggests a nondeterministic choice for the automaton's next configuration. When the automaton is in a state q and is reading a node x labeled by a letter σ, it proceeds by first choosing a pair $\langle q_1, q_2 \rangle \in \delta(q, \sigma)$ and then splitting into two copies. One copy enters the state q_1 and proceeds to the node $x \cdot 0$ (the left child of x), and the other copy enters the state q_2 and proceeds to the node $x \cdot 1$ (the right child of x).

Let $\mathcal{B}^+(\Upsilon_2 \times Q)$ be the set of positive Boolean formulas over $\Upsilon_2 \times Q$; i.e., Boolean formulas built from elements in $\Upsilon_2 \times Q$ using $\wedge$ and $\vee$, where we also allow the formulas ***true*** and ***false*** and, as usual, $\wedge$ has precedence over $\vee$. For a set $S \subseteq \Upsilon_2 \times Q$ and a formula $\theta \in \mathcal{B}^+(\Upsilon_2 \times Q)$, we say that S

satisfies θ if and only if assigning ***true*** to elements in S and assigning ***false*** to elements in $(\Upsilon_2 \times Q) \setminus S$ makes θ true. We can represent δ using $\mathcal{B}^+(\Upsilon_2 \times Q)$. For example,

$$\delta(q, \sigma) = \{\langle q_1, q_2\rangle, \langle q_3, q_1\rangle\}$$

can be written as

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

In nondeterministic tree automata, each conjunction in δ has exactly one element associated with each direction. In alternating automata on binary trees, $\delta(q, \sigma)$ can be an arbitrary formula from $\mathcal{B}^+(\Upsilon_2 \times Q)$. We can have, for instance, a transition

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

The above transition illustrates that several copies may go to the same direction and that the automaton is not required to send copies to all the directions. Formally, a finite alternating automaton on infinite binary trees is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \alpha\rangle$ where Σ is the input alphabet, Q is a finite set of states, $\delta\colon Q \times \Sigma \to \mathcal{B}^+(\Upsilon_2 \times Q)$ is a transition function, $q_0 \in Q$ is an initial state, and α specifies the acceptance condition (a condition that defines a subset of Q^ω).

A *run of an alternating automaton* $\mathcal{A}$ on an input binary labeled tree $\langle T, V\rangle$ is a tree $\langle T_r, r\rangle$ (to be formally defined shortly) in which the root is labeled by q_0 and every other node is labeled by an element of $\Upsilon_2^* \times Q$. Unlike T, in which each node has exactly two children, the tree T_r may have nodes with many children and may also have *leaves* (nodes with no children). Thus, $T_r \subset \mathbb{N}^*$ and a path in T_r may be either finite, in which case it contains a leaf, or infinite. 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. Formally, $\langle T_r, r\rangle$ is a Σ_r-labeled tree where $\Sigma_r = \Upsilon_2^* \times Q$ and $\langle T_r, r\rangle$ satisfies the following:

1. $\varepsilon \in T_r$ and $r(\varepsilon) = (\varepsilon, q_0)$.
2. Let $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, V(x)) = \theta$. Then there is a (possibly empty) set

$$S = \{(c_0, q_0), (c_1, q_1), \ldots, (c_{n-1}, q_{n-1})\} \subseteq \Upsilon_2 \times Q,$$

such that the following hold:
   - S satisfies θ, and
   - for all $0 \le i < n$, we have $y \cdot i \in T_r$ and $r(y \cdot i) = (x \cdot c_i, q_i)$.

For example, if $\langle T, V\rangle$ is a binary tree with $V(\varepsilon) = a$ 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 $(1, q_2)$. Note that if $\theta = \textbf{\textit{true}}$, then y need not have children. This is the reason why T_r may have leaves. Also, since there exists no set S as required for $\theta = \textbf{\textit{false}}$, we cannot have a run that takes a transition with $\theta = \textbf{\textit{false}}$. Each infinite path ρ in $\langle T_r, r\rangle$ is labeled by a word 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 if and only if all its infinite paths satisfy the acceptance condition. In alternating *Rabin* tree automata, $\alpha \subseteq 2^Q \times 2^Q$, and an infinite path ρ satisfies an acceptance condition

$$\alpha = \{\langle G_1, B_1\rangle, \dots , \langle G_m, B_m\rangle\}$$

if and only if there exists $1 \leq i \leq m$ for which $\inf(\rho) \cap G_i \neq \emptyset$ and $\inf(\rho) \cap B_i = \emptyset$. In alternating *Büchi* tree automata, $\alpha \subseteq Q$, and an infinite path ρ satisfies α if and only if $\inf(\rho) \cap \alpha \neq \emptyset$.

For an acceptance condition α over the state space Q and a given set S, the *adjustment* of α to the state space $Q \times S$, denoted $\alpha \times S$, is obtained from α by replacing each set F participating in α by the set $F \times S$.

As with nondeterministic automata, an automaton accepts a tree if and only if 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 $\mathcal{A}$ is *nonempty* if and only if $\mathcal{L}(\mathcal{A}) \neq \emptyset$. We say that $\mathcal{A}$ is a *nondeterministic* tree automaton if and only if all the transitions of $\mathcal{A}$ have only disjunctively related atoms sent to the same direction; i.e., if the transitions are written in DNF, then every disjunct contains at most one atom of the form $(0, q)$ and one atom of the form $(1, q)$.

Alternating tree automata over Υ-trees, for an arbitrary finite set Υ, are defined similarly, with a transition function $\delta\colon Q \times \Sigma \to \mathcal{B}^+(\Upsilon \times Q)$. Accordingly, a run of an alternating automaton over an Υ-tree is a Σ_r-labeled tree with $\Sigma_r = \Upsilon^* \times Q$. In particular, alternating word automata can be viewed as special case of alternating tree automata, running over Υ-trees with a singleton Υ. The transition function of an alternating word automaton is $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$. That is, we omit the unique direction from the atoms in δ.

We define the *size* $|\mathcal{A}|$ of an alternating automaton $\mathcal{A} = \langle\Sigma, Q, \delta, q_0, \alpha\rangle$ 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 $q \in Q$ and $\sigma \in \Sigma$ (the restriction to satisfiable formulas is to avoid an unnecessary $|Q| \cdot |\Sigma|$ minimal size for δ). Note that $\mathcal{A}$ can be stored in space $O(|\mathcal{A}|)$.

4.2. Automata-theoretic constructions. The first step in our solution is to translate the input CTL* formula into an alternating tree automaton. Here, alternation enables us to define automata that are exponentially smaller than automata with no alternations.

THEOREM 4.1 ([2]). *Given a* CTL* *formula* ψ *over a set* AP *of atomic propositions and a set* Υ *of directions, there exists an alternating Rabin tree automaton* $\mathcal{A}_{\Upsilon,\psi}$ *over* 2^{AP}*-labeled* Υ*-trees, with* $2^{O(|\psi|)}$ *states and two pairs, such that* $\mathcal{L}(\mathcal{A}_{\Upsilon,\psi})$ *is exactly the set of trees satisfying* ψ.

The computation tree of P is a $2^{I\cup E\cup O}$-labeled $2^{I\cup E}$-tree. Given an automaton $\mathcal{A}$, which accepts a set of such computation trees, we construct, in Theorem 4.2 below, an automaton $\mathcal{A}'$ that accepts the projection of the trees in $\mathcal{L}(\mathcal{A})$ on the 2^O element of their labels.

THEOREM 4.2. *Given an alternating tree automaton* $\mathcal{A}$ *over* $(\Upsilon \times \Sigma)$*-labeled* Υ*-trees, we can construct an alternating tree automaton* $\mathcal{A}'$ *over* Σ*-labeled* Υ*-trees such that*

(1) $\mathcal{A}'$ *accepts a labeled tree* $\langle \Upsilon^*, V\rangle$ *if and only if* $\mathcal{A}$ *accepts* $\mathrm{xray}(\langle \Upsilon^*, V\rangle)$.
(2) $\mathcal{A}'$ *and* $\mathcal{A}$ *have the same acceptance condition*.
(3) $|\mathcal{A}'| = O(|\mathcal{A}|)$.

PROOF. Let $\mathcal{A} = \langle \Upsilon\times\Sigma, Q, \delta, q_0, \alpha\rangle$. Then, $\mathcal{A}' = \langle \Sigma, Q\times\Upsilon, \delta', \langle q_0, \upsilon^0\rangle, \alpha\times\Upsilon\rangle$, where for every $q \in Q$, $\upsilon \in \Upsilon$, and $\sigma \in \Sigma$, the transition $\delta'(\langle q, \upsilon\rangle, \sigma)$ is obtained from $\delta(q, \langle \upsilon, \sigma\rangle)$ by replacing each atom (υ', q') by the atom $(\upsilon', \langle q', \upsilon'\rangle)$.

That is, a state $\langle q, \upsilon\rangle$ in $\mathcal{A}'$ corresponds to a state q in $\mathcal{A}$ that reads only nodes with direction υ. Thus, the Υ-element in the states of $\mathcal{A}'$ provides the "missing" Υ-element of the alphabet of $\mathcal{A}$. It is easy to see that $\mathcal{A}'$ indeed accepts exactly all the trees whose x-rays are accepted by $\mathcal{A}$. ⊣

Consider a $2^{I\cup E\cup O}$-labeled $2^{I\cup E}$-tree. Recall that we cannot define an automaton that accepts the language of $\mathcal{A}$ when restricted to consistent trees. Instead, we define, in Theorem 4.3 below, an automaton $\mathcal{A}'$ that accepts a $2^{I\cup E\cup O}$-labeled 2^O-tree if and only if its 2^E-widening is accepted by $\mathcal{A}$. Each such 2^E-widening is guaranteed to be consistent! Thus, $\mathcal{A}'$ is nonempty if and only if $\mathcal{A}$ when restricted to consistent trees is nonempty.

THEOREM 4.3. *Let* X, Y, *and* Z *be finite sets. Given an alternating tree automaton* $\mathcal{A}$ *over* Z*-labeled* $(X \times Y)$*-trees, we can construct an alternating tree automaton* $\mathcal{A}'$ *over* Z*-labeled* X*-trees such that*

(1) $\mathcal{A}'$ *accepts a labeled tree* $\langle X^*, V\rangle$ *if and only if* $\mathcal{A}$ *accepts* $\mathrm{wide}_Y(\langle X^*, V\rangle)$.
(2) $\mathcal{A}'$ *and* $\mathcal{A}$ *have the same acceptance condition*.
(3) $|\mathcal{A}'| = O(|\mathcal{A}|)$.

PROOF. Let $\mathcal{A} = \langle Z, Q, \delta, q_0, \alpha\rangle$. Then, $\mathcal{A}' = \langle Z, Q, \delta', q_0, \alpha\rangle$, where for every $q \in Q$ and $z \in Z$, the transition $\delta'(q, z)$ is obtained from $\delta(q, z)$ by replacing each atom $(\langle x, y\rangle, q')$ by the atom (x, q'). We prove the correctness of our construction. Thus, we prove that for every Z-labeled X-tree $\langle X^*, V\rangle$, we have

$$\langle X^*, V\rangle \in \mathcal{L}(\mathcal{A}') \qquad \text{if and only if} \qquad \text{wide}_Y(\langle X^*, V\rangle) \in \mathcal{L}(\mathcal{A}).$$

Assume first that $\text{wide}_Y(\langle X^*, V\rangle) \in \mathcal{L}(\mathcal{A})$. We prove that $\langle X^*, V\rangle \in \mathcal{L}(\mathcal{A}')$. Let $\langle T_r, r\rangle$ be an accepting run of $\mathcal{A}$ on $\text{wide}_Y(\langle X^*, V\rangle)$. By the definition of a run, we have that $r\colon T_r \to ((X \times Y)^* \times Q)$ maps a node of the run-tree to a pair consisting of a node in $(X \times Y)^*$ and a state of $\mathcal{A}$ that reads this node. Consider the labeled tree $\langle T_r, r'\rangle$ where for every $w \in T_r$ with $r(w) = \langle u, q\rangle$ we have $r'(w) = \langle \text{hide}(u), q\rangle$. It is easy to see that $\langle T_r, r'\rangle$ is an accepting run of $\mathcal{A}'$ on $\langle X^*, V\rangle$.

Assume now that $\langle X^*, V\rangle \in \mathcal{L}(\mathcal{A}')$. We prove that $\text{wide}_Y(\langle X^*, V\rangle) \in \mathcal{L}(\mathcal{A})$. Since $\text{hide}^{-1}(u)$, for $u \in X^*$, is not a singleton, this direction is harder. We first define an alternating tree automaton $\mathcal{A}''$. The automaton $\mathcal{A}''$ is identical to the automaton $\mathcal{A}'$, only that it annotates the states with a direction from Y that maintains the information that gets lost when we obtain δ' from δ. Formally,

$$\mathcal{A}'' = \langle Z, Q \times Y, \delta'', \langle q_0, y^0\rangle, \alpha \times Y\rangle,$$

where y^0 is the root direction of Y, and for every $q \in Q$, $y \in Y$, and $z \in Z$, the transition $\delta''(\langle q, y\rangle, z)$ is obtained from $\delta(q, z)$ by replacing each atom $(\langle x, y'\rangle, q')$ by the atom $(x, \langle q', y'\rangle)$.

It is easy to see that $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A}')$. We now prove that if $\langle X^*, V\rangle \in \mathcal{L}(\mathcal{A}'')$, then $\text{wide}_Y(\langle X^*, V\rangle) \in \mathcal{L}(\mathcal{A})$. Let $\langle T_r, r''\rangle$ be an accepting run of $\mathcal{A}''$ on $\langle X^*, V\rangle$. By the definition of a run, we have that $r'' : T_r \to (X^* \times (Q \times Y))$ maps a node of the run to a pair consisting of a node in X^* and a state of $\mathcal{A}''$ that reads this node. Consider the labeled tree $\langle T_r, r\rangle$ with $r : T_r \to ((X \times Y)^* \times Q)$, where

- $r(\varepsilon) = r''(\varepsilon)$, and
- for every $wc \in T_r$ with $r(w) = \langle u, q'\rangle$ and $r''(wc) = \langle ux, \langle q, y\rangle\rangle$, we have $r(wc) = \langle u\langle x, y\rangle, q\rangle$.

It is easy to see that $\langle T_r, r\rangle$ is an accepting run of $\mathcal{A}$ on $\text{wide}_Y(\langle X^*, V\rangle)$. ⊣

Given an alternating tree automaton $\mathcal{A}$, let *cover*$(\mathcal{A})$ and *narrow*$_Y(\mathcal{A})$ denote the corresponding automata $\mathcal{A}'$ constructed in Theorems 4.2 and 4.3 (for a set Y of directions), respectively.

4.3. Automata-theoretic solution. We can now use Theorems 4.1, 4.2, and 4.3, to solve the general synthesis problem.

THEOREM 4.4. *For every* CTL* *formula* ψ *over the sets* I, E, *and* O *of readable input, unreadable input, and output signals,* ψ *is realizable if and only if* $\text{narrow}_{(2^E)}(\text{cover}(\mathcal{A}_{(2^{I\cup E}),\psi}))$ *is nonempty.*

PROOF. Let $\mathcal{A} = \mathcal{A}_{(2^{I\cup E}),\psi}$. Recall that $\mathcal{A}$ is an alternating tree automaton over $2^{I\cup E\cup O}$-labeled $2^{I\cup E}$-trees. Hence, the automaton $\text{cover}(\mathcal{A})$ runs over

2^O-labeled $2^{I \cup E}$-trees and the automaton $\text{narrow}_{(2^E)}(\text{cover}(\mathcal{A}))$ runs over 2^O-labeled 2^I-trees.

Assume first that ψ is realizable. Then, by Theorem 4.1, there exists a program $P \colon (2^I)^* \to 2^O$ such that

$$\text{xray}(\text{wide}_{(2^E)}(\langle (2^I)^*, P \rangle)) \in \mathcal{L}(\mathcal{A}).$$

Hence, by Theorem 4.2,

$$\text{wide}_{(2^E)}(\langle (2^I)^*, P \rangle) \in \mathcal{L}(\text{cover}(\mathcal{A})).$$

Hence, by Theorem 4.3,

$$\langle (2^I)^*, P \rangle \in \mathcal{L}(\text{narrow}_{(2^E)}(\text{cover}(\mathcal{A}))),$$

which is therefore nonempty.

Assume now that $\text{narrow}_{(2^E)}(\text{cover}(\mathcal{A}))$ is nonempty. Then, there exists a tree

$$\langle (2^I)^*, P \rangle \in \mathcal{L}(\text{narrow}_{(2^E)}(\text{cover}(\mathcal{A}))).$$

Hence, by Theorems 4.2 and 4.3,

$$\text{xray}(\text{wide}_{(2^E)}(\langle (2^I)^*, P \rangle)) \in \mathcal{L}(\mathcal{A}).$$

Hence, by Theorem 4.1, P realizes ψ. ⊣

We say that P is a *finite-state program* if and only if there exists a deterministic finite-state word automaton $\mathcal{U}_P$ over the alphabet 2^I, such that the set of states of $\mathcal{U}_P$ is $Q \times 2^O$ for some finite set Q, and for every word $x \in (2^I)^*$, the run of $\mathcal{U}_P$ on x terminates in a state in $Q \times \{P(x)\}$. Thus, finite-state programs are generated by finite-state automata. We sometimes refer also to finite state strategies, for arbitrary Σ-labeled Υ-trees. It is shown in [25] that for every realizable LTL specification we can synthesize a finite-state program. In the theorem below, we show that this holds also for the general synthesis problem.

THEOREM 4.5. *Given an alternating tree automaton $\mathcal{A}$ over Σ-labeled Υ-trees, the following are equivalent*:

1. *$\mathcal{A}$ is nonempty.*
2. *There exists a finite-state strategy $f \colon \Upsilon^* \to \Sigma$ such that $\langle \Upsilon^*, f \rangle \in \mathcal{L}(\mathcal{A})$.*

Furthermore, the nonemptiness algorithm for $\mathcal{A}$ can be extended, within the same complexity bounds, to produce a finite-state strategy.

PROOF. Rabin proved the theorem for nondeterministic Rabin tree automata [28]. Muller and Schupp translated alternating tree automata to nondeterministic tree automata with the same acceptance condition [22]. ⊣

THEOREM 4.6. *The synthesis problem for* LTL *and* CTL*, *with either complete or incomplete information, is* 2EXPTIME-*complete.*

Proof. We reduced the synthesis problem for a CTL* formula ψ to the nonemptiness problem of an alternating Rabin tree automaton with exponentially many states and linearly many pairs. The upper bounds then follow from the known translation of alternating Rabin tree automata to nondeterministic Rabin tree automata [22] and the complexity of the nonemptiness problem for the latters [11, 25]. The lower bounds follow from the known lower bounds to the realizability problem for LTL and the satisfiability problems for CTL* [32, 25, 29]. ⊣

Theorem 4.6 shows that, from a complexity-theoretic perspective, the problem of synthesis with incomplete information is not harder than the problem of synthesis with complete information. To understand the actual cost of incomplete information, recall that in the presence of incomplete information, the alternating automaton whose emptiness is needed to be checked goes through an additional transformation (narrow). This transformation increases the "universality" of the automaton. That is, more copies of the automaton are now sent to each of the fewer directions. These two facts, the increased "universality" and the decrease in the number of directions, may have opposing effects on the complexity of the translation from an alternating automaton to a nondeterministic one, which is part of the nonemptiness procedure. Thus, it is hard to predict what the actual impact is on the running time of the algorithm.

4.4. Handling CTL specifications more efficiently. As CTL is a subset of CTL*, the algorithm described in Theorem 4.4 suggests a solution for the synthesis problem of CTL. Nevertheless, though CTL is simpler than CTL* (in particular, when ψ is a CTL formula, the automaton $\mathcal{A}_{\Upsilon,\psi}$ described in Theorem 4.1 is an alternating Büchi tree automaton with $O(|\psi|)$ states [2]), the time required for executing our algorithm for a CTL formula is doubly exponential in the length of the formula, just as the one for CTL*. To see this, note that the number of states in $cover(\mathcal{A}_{\Upsilon,\psi})$ depends not only on the number of states in $\mathcal{A}_{\Upsilon,\psi}$, but also on the size of Υ. Since $cover(\mathcal{A}_{\Upsilon,\psi})$ is an alternating tree automaton, checking it (or its narrowing) for nonemptiness is exponential in the number of its states, and therefore also in Υ. Hence, as the size of Υ is exponential in the number of input signals, independent of the type of ψ, we end-up with a doubly-exponential-time algorithm. In [18] we described an exponential-time algorithm for solving the synthesis problem for CTL. The idea is to minimize the use of alternation: employ it when it is crucial (checking that candidate trees are consistent), and give it up when it is a luxury (checking that candidate trees are I-exhaustive). For that, we need to change the order in which things are checked. That is, we first check consistency, and then check I-exhaustiveness.

Theorem 4.7. *The synthesis problem for* CTL, *with either complete or incomplete information*, *is* EXPTIME-*complete.*

§5. Discussion and future work. We suggested a framework for the synthesis problem in its full generality. The difficulties caused by incomplete information require the use of alternating tree automata. While the synthesis problem for open systems is different and more general than the satisfiability problem, alternating tree automata suggest very similar solutions to these problems. Both solutions are based on the translation of a temporal logic formula ψ into an alternating tree automaton $\mathcal{A}_\psi$, and on the nonemptiness test for $\mathcal{A}_\psi$. For the synthesis problem, the automaton $\mathcal{A}_\psi$ needs to pass a simple transformation ($cover$) before the nonemptiness test. In the presence of incomplete information, it needs to pass an additional simple transformation ($narrow$). Our framework handles, as a special case, the classical LTL synthesis problem and suggest a simpler solution for this case. In particular, it avoids the determinization procedure required in [25]. Typically, alternating tree automata shift all the combinatorial difficulties of the synthesis problem, in both the linear and the branching framework, to the nonemptiness test.

The setting discussed in this paper assumes a fully synchronous model, where at each time unit both the system and the environment react. In [26], the authors study the synthesis of asynchronous systems that satisfy linear-time specifications. An extension to asynchronous systems with incomplete information is described in [31]. We expect our techniques to be extensible to synthesis of asynchronous systems with incomplete information that satisfy branching time specifications. The setting discussed in this paper also assumes a maximal environment, in which all possible input sequences are enabled. As discussed in [16, 17], often we are interested in a more general setting, in which the specification has to be satisfied no matter how the environment disables some of the inputs. This problem, of *robust synthesis*, is more complex than the one discussed here, and is the subject of further research.

REFERENCES

[1] M. Abadi, L. Lamport, and P. Wolper, *Realizable and unrealizable concurrent program specifications*, ***Proceedings of the 16th international colloquium on automata, languages and programming***, Lecture Notes in Computer Science, no. 372, Springer-Verlag, Berlin, July 1989, pp. 1–17.

[2] O. Bernholtz, M. Y. Vardi, and P. Wolper, *An automata-theoretic approach to branching-time model checking*, ***Computer aided verification, Proceedings of the 6th international conference*** (D. L. Dill, editor), Lecture Notes in Computer Science, no. 818, Springer-Verlag, Berlin, June 1994, pp. 142–155.

[3] J. R. Büchi and L. HG. Landweber, *Solving sequential conditions by finite-state strategies*, ***Transactions of the American Mathematical Society***, vol. 138 (1969), pp. 295–311.

[4] A. Church, *Logic, arithmetics, and automata*, ***Proceedings of the international congress of mathematicians, 1962***, Institut Mittag-Leffler, 1963, pp. 23–35.

[5] D. L. Dill, ***Trace theory for automatic hierarchical verification of speed independent circuits***, MIT Press, 1989.

[6] A. E. EMERSON and A. P. SISTLA, *Deciding full branching time logics*, ***Information and Control***, vol. 61 (1984), no. 3, pp. 175–201.

[7] E. A. EMERSON, *Temporal and modal logic*, ***Handbook of theoretical computer science***, 1990, pp. 997–1072.

[8] E. A. EMERSON and E. M. CLARKE, *Using branching time logic to synthesize synchronization skeletons*, ***Science of Computer Programming***, vol. 2 (1982), pp. 241–266.

[9] E. A. EMERSON and J. Y. HALPERN, *Decision procedures and expressiveness in the temporal logic of branching time*, ***Journal of Computer and System Sciences***, vol. 30 (1985), pp. 1–24.

[10] ———, *Sometimes and not never revisited: On branching versus linear time*, ***Journal of the Association for Computing Machinery***, vol. 33 (1986), no. 1, pp. 151–178.

[11] E. A. EMERSON and C. JUTLA, *The complexity of tree automata and logics of programs*, ***Proceedings of the 29th IEEE symposium on foundations of computer science***, White Plains, October 1988, pp. 368–377.

[12] Y. GUREVICH and S. SHELAH, *Rabin's uniformization problem*, this JOURNAL, vol. 48 (1983), pp. 1105–1119.

[13] D. HAREL and A. PNUELI, *On the development of reactive systems*, ***Logics and models of concurrent systems*** (K. Apt, editor), NATO Advanced Summer Institutes, no. F-13, Springer-Verlag, 1985, pp. 477–498.

[14] R. KUMAR and V. K. GARG, ***Modeling and control of logical discrete event systems***, Kluwer Academic Publishers, 1995.

[15] R. KUMAR and M. A. SHAYMAN, *Supervisory control of nondeterministic systems under partial observation and decentralization*, ***Siam journal of control and optimization***, 1995.

[16] O. KUPFERMAN and M. Y. VARDI, *Module checking*, ***Computer aided verification, Proceedings of the 8th international conference***, Lecture Notes in Computer Science, no. 1102, Springer-Verlag, 1996, pp. 75–86.

[17] ———, *Module checking revisited*, ***Computer aided verification, Proceedings of the 9th international conference***, Lecture Notes in Computer Science, no. 1254, Springer-Verlag, 1997, pp. 36–47.

[18] ———, *Synthesis with incomplete informatio*, ***2nd international conference on temporal logic***, Kluwer Academic Publishers, Manchester, July 1997, pp. 91–106.

[19] L. LAMPORT, *Sometimes is sometimes "not never"—on the temporal logic of programs*, ***Proceedings of the 7th ACM symposium on principles of programming languages***, January 1980, pp. 174–185.

[20] Z. MANNA and R. WALDINGER, *A deductive approach to program synthesis*, ***Association for Computing Machinery Transactions on Programming Languages and Systems***, vol. 2 (1980), no. 1, pp. 90–121.

[21] D. E. MULLER and P. E. SCHUPP, *Alternating automata on infinite trees*, ***Theoretical Computer Science***, vol. 54 (1987), pp. 267–276.

[22] ———, *Simulating alternating tree automata by nondeterministic automata: new results and new proofs of theorems of Rabin, McNaughton and Safra*, ***Theoretical Computer Science***, vol. 141 (1995), pp. 69–107.

[23] A. PNUELI, *The temporal logic of programs*, ***Proceedings of the 18th IEEE symposium on foundation of computer science***, 1977, pp. 46–57.

[24] ———, *The temporal semantics of concurrent programs*, ***Theoretical Computer Science***, vol. 13 (1981), pp. 45–60.

[25] A. PNUELI and R. ROSNER, *On the synthesis of a reactive module*, ***Proceedings of the 16th ACM symposium on principles of programming languages***, Austin, January 1989.

[26] ———, *On the synthesis of an asynchronous reactive module*, ***Proceedings of the 16th international colloquium on automata, languages and programming***, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, July 1989, pp. 652–671.

[27] ———, *Distributed reactive systems are hard to synthesize*, ***Proceedings of the 31st IEEE symposium on foundation of computer science***, 1990, pp. 746–757.

[28] M. O. RABIN, *Weakly definable relations and special automata*, ***Proceedings of symposia in mathematics, logic and foundations of set theory***, North-Holland, 1970, pp. 1–23.

[29] R. ROSNER, *Modular synthesis of reactive systems*, ***Ph.D. thesis***, Weizmann Institute of Science, Rehovot, Israel, 1992.

[30] J. W. THATCHER, *Tree automata: an informal survey*, ***Currents in the theory of computing*** (A. V. Aho, editor), Prentice-Hall, Englewood Cliffs, 1973, pp. 143–172.

[31] M. Y. VARDI, *An automata-theoretic approach to fair realizability and synthesis*, ***Computer aided verification, Proceedings of the 7th international conference*** (P. Wolper, editor), Lecture Notes in Computer Science, no. 939, Springer-Verlag, Berlin, 1995, pp. 267–292.

[32] M. Y. VARDI and L. STOCKMEYER, *Improved upper and lower bounds for modal logics of programs*, ***Proceedings of the 17th acm symposium on theory of computing***, 1985, pp. 240–251.

[33] M. Y. VARDI and P. WOLPER, *Automata-theoretic techniques for modal logics of programs*, ***Journal of Computer and System Science***, vol. 32 (1986), no. 2, pp. 182–221.

THE INSTITUTE OF COMPUTER SCIENCE
HEBREW UNIVERSITY
JERUSALEM 91904, ISRAEL
E-mail: orna@cs.huji.ac.il

DEPARTMENT OF COMPUTER SCIENCE
RICE UNIVERSITY
HOUSTON, TEXAS 77005-1892, USA
E-mail: vardi@cs.rice.edu