Synthesizing Distributed Systems

Orna Kupferman
Hebrew University*

Moshe Y. Vardi
Rice University†

Abstract

In system synthesis, we transform a specification into a system that is guaranteed to satisfy the specification. When the system is distributed, the goal is to construct the system's underlying processes. Results on multi-player games imply that the synthesis problem for linear specifications is undecidable for general architectures, and is nonelementary decidable for hierarchical architectures, where the processes are linearly ordered and information among them flows in one direction. In this paper we present a significant extension of this result. We handle both linear and branching specifications, and we show that a sufficient condition for decidability of the synthesis problem is a linear or cyclic order among the processes, in which information flows in either one or both directions. We also allow the processes to have internal hidden variables, and we consider communications with and without delay. Many practical applications fall into this class.

1 Introduction

In *system synthesis*, we transform a specification into a system that is guaranteed to satisfy the specification. Early work on synthesis consider *closed systems*. There, a system that meets the specification can be extracted from a constructive proof that the specification is satisfiable [MW80, EC82]. As argued in [ALW89, Dil89, PR89a], such synthesis paradigms are not of much interest when applied to *open systems*, which interact with an environment. While synthesis that is based on satisfiability assumes no environment or a cooperative one, synthesis of open systems should assume a hostile environment, and should generate a system that satisfies the specification no matter how the environment behaves. The work in [ALW89, PR89a] formulated the synthesis problem in terms of a *game* between the system and the environment, and is closely related to *Church's solvability problem* [Chu63]. Given sets I and O of input and output signals, respectively, we can view a system as a *strategy* $P : (2^I)^* \to 2^O$ that maps a finite sequence of sets of input signals (the behavior of the environment so far) into a set of output signals (the reaction of the system to this behavior).

When P interacts with an environment that generates infinite input sequences, it associates with each input sequence an infinite computation over $2^{I \cup O}$. We say that a specification ψ is *realizable* iff there is a strategy all of whose computations satisfy ψ, in case ψ is a linear specification, or a strategy whose induced computation tree satisfies ψ, in case ψ is a branching specification. *Synthesis* of ψ then amounts to constructing such a strategy. Solutions for the realizability and synthesis problems for specifications in the linear temporal logic LTL are presented in [ALW89, PR89a]. The solutions are extended in [PR89b, Var95] to asynchronous systems and in [KV99] to systems with incomplete information and specifications in the branching temporal logic CTL*. Methods developed for synthesis of open systems are applicable also for *supervisory control*, where instead of hostile environments we consider collaborative controllers of nondeterministic systems [RW89].

While the transition to open systems has significantly broaden the scope of synthesis to real-life designs, it is still limited to settings in which the open system consists of a single process. In a more realistic setting, that of a *distributed system*, the input to the synthesis problem consists of both the specification and an *architecture*, which may consist of more than one process and describes the communication channels between the different processes. More formally, we assume a setting with n processes, with process i referring to sets I_i, O_i, and H_i, of input, output, and hidden (internal) signals (input signals may be *external*; i.e., generated by the environment), and we want to construct for each process a strat-

*Work partially supported by BSF grant 9800096. Address: School of Computer Science and Engineering, Jerusalem 91904, Israel. Email: orna@cs.huji.ac.il

†Work partially supported by NSF grants CCR-9700061 and CCR-9988322, BSF grant 9800096, and a grant from the Intel Corporation. Address: Department of Computer Science, Houston, TX 77251-1892, U.S.A. Email: vardi@cs.rice.edu

egy $P_i : (2^{I_i})^* \to 2^{O_i \cup H_i}$ so that the composition of the strategies satisfies the specification. The architecture is given by a set of conditions like $O_2 \cup O_4 \subseteq I_3$ ("the only channels to P_3 are from P_2 to P_4"). The exact definition of the composition of the strategies then depends on assumptions on the communication (e.g., whether communication involves a delay). If, for example, we want to synthesize five dining philosophers [Dij72], we can specify in temporal logic the mutual exclusion and non-starvation requirements for the philosophers, specify a two-way ring with five processes, and ask the synthesis procedure to construct appropriate strategies for the processes. Clearly, a solution for the dining philosophers that refers to a single process is not of much interest.

There are two possible ways to approach the synthesis problem for distributed systems. One approach is to use a synthesis procedure for a single process, and then *decompose* the process according to the given architecture [EC82, MW84]. While this approach has a computational advantage, known decomposition algorithms are not complete in the sense that a specification may be realizable with respect to a given architecture yet the decomposition algorithm would fail [PR90]. Thus, one can view decomposition as a heuristic for the synthesis problem, which is not guaranteed to work. The second approach is to refer to the architecture of the distributed system from the outset and construct the underlying processes directly [PR90].

Results on multi-player games imply that the realizability problem for general distributed systems is undecidable [PR79, PR90] (the results in [PR79] refer to multiple-person alternating Turing machines and are extended in [PR90] to the synthesis setting). Essentially, there is an architecture Ω (in fact, a very simple architecture, consisting of two independent processes P_1 and P_2 that interact with the same environment; that is $I_1 \cap (O_2 \cup H_2) = \emptyset$ and $I_2 \cap (O_1 \cup H_1) = \emptyset$) such that for every deterministic Turing machine M, there is an LTL formula ψ_M such that M halts on the empty tape iff ψ_M is realizable in Ω. The reduction is heavily based on P_1 and P_2 being independent, and it fails, for example, if we assume that P_2 gets its input from P_1 (i.e., $O_1 \subseteq I_2$). Indeed, it is shown in [PR79, PR90] that once we consider *hierarchical architectures*, in which the processes are linearly ordered and information flows in one direction, the realizability problem is nonelementary decidable for specifications in LTL.

The decidability result in [PR90] suffers from two limitations. First, when we synthesize a system from an LTL specification ψ, we require ψ to hold in all the computations of the system. Consequently, we cannot impose possibility requirements on the system (cf. [DTV99]). In the dining-philosophers example, while we can specify in LTL mutual exclusion, we cannot specify deadlock freedom (every finite interaction *can be* extended so that a philosopher eventually eats). In order to express possibility properties, we should specify the system using *branching temporal logic*, which enables both universal and existential path quantification [EH86, Eme90]. Second, and more crucially, the algorithm in [PR90] is not applicable for architectures that are not hierarchical, and real-life designs are rarely based on hierarchical architectures. We do not count the nonelementary complexity as a limitation, as it is accompanied by a matching lower bound and, as we discuss further in Section 6, the worst-case complexity rarely appears in practice.

In this paper we remove both limitations. We consider specifications in the branching temporal logic CTL* (which subsumes LTL), and we handle all architectures in which there is a linear or cyclic order among the processes, in which information flows in either one or both directions. Thus, our architectures can be either chains or rings with both one-way and two-way communication channels. In addition, we allow the processes to have internal hidden variables, and we consider communications with and without delay. We show that the realizability problem stays decidable in all these cases. The solution we present is based on *alternating tree automata*, which separate the logical and algorithmic aspects of the problem: given a specification ψ and an architecture Ω, we construct an automaton $\mathcal{A}_{\Omega,\psi}$ such that ψ is realizable in Ω iff $\mathcal{A}_{\Omega,\psi}$ is not empty. To check realizability, the automaton has to be tested for nonemptiness [EJ88, PR89a, KV98]. The nonemptiness algorithm also synthesizes the processes in Ω that together realize ψ.

We argue that the results in the paper significantly extend the scope of synthesis for distributed systems, as commonly used architecture belong to the class of architectures we handle [Tan87]. Examples of applications of these architectures include various communication protocols in which communication proceeds in layers. For example, the so-called OSI model consists of a seven-layer *protocol stack* (Application, Presentation, Session, Transport, Network, Data link, and Physical layers), where every layer communicates with the layer above it and the layer below it. The environment talks to the top layer and the bottom layer [Man99]. Architectures with two-way communication channels are common in *scientific computations*, say when we iterate in order to solve a differential equa-

tion and each process works on part of the computed domain. Then, it is useful to divide the domain to layers so that in each iteration every layer updates its neighbors with its results from the previous iteration [PTVF92].

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. When $T = \Upsilon^*$, we say that T is *full*. 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*, $dir(x)$ in Υ. The direction of ϵ is υ^0, for some designated $\upsilon^0 \in \Upsilon$, called the *root direction*. The direction of a node $x \cdot \upsilon$ is υ.

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 a Σ-labeled Υ-tree $\langle \Upsilon^*, V\rangle$, we define the *memoryfull* version of $\langle \Upsilon^*, V\rangle$, denoted $mem(\langle \Upsilon^*, V\rangle)$ as the Σ^+-labeled Υ-tree $\langle \Upsilon^*, V'\rangle$ where $V'(\epsilon) = V(\epsilon)$, for $\upsilon \in \Upsilon$ we have $V'(\upsilon) = V(\epsilon) \cdot V(\upsilon)$, and for all $x \in \Upsilon^+$ and $\upsilon \in \Upsilon$ we have $V'(x \cdot \upsilon) = V'(x) \cdot V(\upsilon)$. Thus, the label of a node x in $mem(\langle \Upsilon^*, V\rangle)$ is the word obtained by concatenating the labels of all the prefixes (including ϵ) of x in $\langle \Upsilon^*, V\rangle$.

For a Σ-labeled Υ-tree $\langle \Upsilon^*, V\rangle$, we define the *x-ray* of $\langle \Upsilon^*, V\rangle$, denoted $xray(\langle \Upsilon^*, V\rangle)$, as the $(\Upsilon \times \Sigma)$-labeled Υ-tree $\langle \Upsilon^*, V'\rangle$ in which each node is labeled by both its direction and its labeling in $\langle \Upsilon^*, V\rangle$. Thus, for every $x \in \Upsilon^*$, we have $V'(x) = \langle dir(x), V(x)\rangle$. Essentially, the labels in $xray(\langle \Upsilon^*, V\rangle)$ contain information not only about the surface of $\langle \Upsilon^*, V\rangle$ (its labels) but also about its skeleton (its nodes).

For a Σ-labeled Υ-tree $\langle \Upsilon^*, V\rangle$, we define the *delay* of $\langle \Upsilon^*, V\rangle$, denoted $delay(\langle \Upsilon^*, V\rangle)$, as the Σ-labeled Υ-tree $\langle \Upsilon^*, V'\rangle$ in which $V'(\epsilon) = V(\epsilon)$ and for all $x \in \Upsilon^*$ and $\upsilon \in \Upsilon$, we have $V'(x \cdot \upsilon) = V(\upsilon_0 \cdot x)$, where $\upsilon_0 = dir(\epsilon)$ is the root direction of Υ. Intuitively, the delay of $\langle \Upsilon^*, V\rangle$ describes the label node x would have when the sequence of directions leading to x arrives with a delay, thus the last direction in x is missing and x is prefixed by the root direction.

Consider a set $X \times Y$ of directions. For a node $\tau \in (X \times Y)^*$, let $hide_Y(\tau)$ be the node in X^* obtained from τ by replacing each letter $\langle x, y\rangle$ by the letter x. For example, the node $\langle 0,0\rangle \cdot \langle 1,0\rangle$ of the 4-ary $(\{0,1\} \times \{0,1\})$-tree corresponds, by $hide_{\{0,1\}}$, to the node $0 \cdot 1$ of the $\{0,1\}$-tree. Note that the nodes $\langle 0,0\rangle \cdot \langle 1,1\rangle, \langle 0,1\rangle \cdot \langle 1,0\rangle$, and $\langle 0,1\rangle \cdot \langle 1,1\rangle$ of the 4-ary tree also correspond, by $hide_{\{0,1\}}$, to the node $0 \cdot 1$ of the binary tree. For a Z-labeled X-tree $\langle X^*, V\rangle$, we define the Y-*widening* of $\langle X^*, V\rangle$, denoted $wide_Y(\langle X^*, V\rangle)$, as the Z-labeled $(X \times Y)$-tree $\langle (X \times Y)^*, V'\rangle$ where for every $\tau \in (X \times Y)^*$, we have $V'(\tau) = V(hide_Y(\tau))$. As we explain further in Section 3, nodes τ_1 and τ_2 with $hide_Y(\tau_1) = hide_Y(\tau_2) = \tau$ are indistinguishable in $wide_Y(\langle X^*, V\rangle)$ by someone that does not observe Y. Indeed, for such an observer, both nodes are reached by traversing τ and are labeled by $V(\tau)$.

2.2 Alternating 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 nodes are labeled by elements 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 *Rabin* alternating tree automata, $\alpha \subseteq 2^Q \times 2^Q$, and an infinite path ρ satisfies an acceptance condition $\alpha = \{\langle G_1, B_1\rangle, \ldots, \langle G_k, B_k\rangle\}$ iff there exists $1 \leq i \leq k$ for which $inf(\rho) \cap G_i \neq \emptyset$ and $inf(\rho) \cap B_i = \emptyset$. We refer to the number of pairs in α as the *index* of $\mathcal{A}$. 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 $\mathcal{L}(\mathcal{A}) \neq \emptyset$. For an acceptance condition α over Q and a set S, we denote by $\alpha \times S$ the acceptance condition over $Q \times S$ obtained from α by replacing each set F participating in α by the set $F \times S$. For example, if α is the Rabin acceptance condition $\{\langle G, B\rangle\}$, then $\alpha \times S = \{\langle G \times S, B \times S\rangle\}$.

Nondeterministic tree automata can be viewed as a special case of alternating tree automata, where the formulas in $\mathcal{B}^+(\Upsilon \times Q)$ are such that if a formula is rewritten in disjunctive normal form, then for every direction $\upsilon \in \Upsilon$, there is exactly one element of $\{\upsilon\} \times Q$ in each disjunct. While nondeterministic tree automata are not less expressive than alternating tree automata, they are exponentially less succinct:

Theorem 2.1 [MS95] *An alternating Rabin tree automaton with m states and k pairs can be translated to an equivalent nondeterministic Rabin tree automaton with $m^{O(mk)}$ states and $O(mk)$ pairs.*

3 Architectures and the synthesis problem

Given sets I and O of input and output signals, respectively, we can view a process P as a *strategy* $f : (2^I)^* \to 2^O$ that maps a finite sequence of sets of input signals into a set of output signals. We often refer to the strategy f as the 2^O-labeled 2^I-tree $\langle (2^I)^*, f\rangle$. Let i_0 be the root direction of 2^I. When P interacts with an environment that generates infinite input sequences, it associates with each infinite input sequence $i_1, i_2, \ldots$, an infinite computation $\{i_0\} \cup f(\varepsilon), \{i_1\} \cup f(i_1), \{i_2\} \cup f(i_1 \cdot i_2), \ldots$ over $2^{I \cup O}$. The interaction of P with all possible input sequences induces the $(2^{I \cup O})$-labeled 2^I-tree $xray(\langle (2^I)^*, f\rangle)$. The environment may have hidden internal signals, which are not readable by P. Let H denote the set of hidden signals. Then, a strategy for P is still a function $f : (2^I)^* \to 2^O$, but the interaction of P with an outcome of the environment induces an infinite computation over $2^{I \cup O \cup H}$, and its interaction with all possible outcomes induces the $(2^{I \cup O \cup H})$-labeled $(2^{I \cup H})$-tree $xray(wide_{(2^H)}(\langle (2^I)^*, f\rangle))$. Each node in this tree has $2^{|I \cup H|}$ children[1], corresponding to the $2^{|I \cup H|}$ possible assignments to $I \cup H$. Note that since P cannot see the signals in H, and thus cannot distinguish between children that agree on their assignment to signals in I, the tree above is the 2^H-widening of the interaction between P and its environment as seen by P.

In a setting with n processes $P_1, \ldots, P_n$, where process P_i reads I_i, writes O_i, and has hidden internal signals H_i, a strategy for P_i is a function $f_i : (2^{I_i})^* \to 2^{O_i \cup H_i}$. We denote $\bigcup_{1 \leq i \leq n} I_i$ by I, and similarly for O and H. The n processes $P_1, \ldots, P_n$ interact with each other and may also interact with an environment. We denote by O_{env} the output signals of the environment (that is, the external input to the n processes), and denote by H_{env} the hidden signals of the environment.

Different architectures induce different communication channels between the processes. We consider here four classes of architectures (see figure next page). In all classes, each signal can be written by a single process (that is, $O_i \cap O_j = \emptyset$ for all $i \neq j$), but can be read by several processes (that is, possibly $I_i \cap I_j \neq \emptyset$).

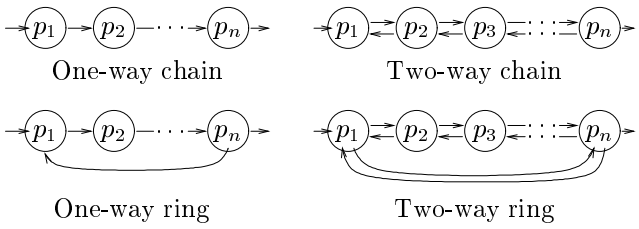
- In a *one-way chain*, P_1 reads from the environment, P_n writes to the environment, and all the other processes read from the process to their left, and write to the process to their right. Formally, $I_1 = O_{env}$, and for all $2 \leq i \leq n$ we have $I_i = O_{i-1}$. Note that P_i cannot read the internal signals of the process to its left and that $I \cup O = I \cup O_n = I_1 \cup O$.
- A *one-way ring* extends a one-way chain by a communication channel from P_n to P_1. Thus, P_1 reads from both P_n and the environment (i.e., $I_1 = O_n \cup O_{env}$), and P_n writes to both P_1 and the environment.
- In a *two-way chain*, P_1 reads from both P_2 and the environment and writes to P_2, P_n reads from P_{n-1} and writes to both P_{n-1} and the environment, and all the other processes read from the

[1] We consider synthesis with respect to *maximal environments*, which provide all possible input sequences. An extension to non-maximal environment is possible, using the same techniques as in [KMTV00].

processes to their left and right, and write to the processes to their left and right. Formally, $I_1 = O_{env} \cup O_2$, for all $2 \leq i \leq n-1$ we have $I_i = O_{i-1} \cup O_{i+1}$, and $I_n = O_{n-1}$.

- A *two-way ring* extends a two-way chain by a communication channel between P_n and P_1. Thus, P_1 reads from P_2, P_n, and the environment (i.e., $I_1 = O_{env} \cup O_2 \cup O_n$), and writes to both P_2 and P_n, and P_n reads from both P_1 and P_{n-1} and writes to both P_1, P_{n-1}, and the environment.

Note that in all the four classes, and for all i and j with $i < j$, the process P_i has complete information about the input to P_j, thus P_i can simulate P_j and have complete information also about its output[2]. This means, for example, that in a two-way chain, we could give up the channel from P_2 to P_1, letting P_1 compute the information along this channel, and similarly for the other right-to-left channels. While this would not change the answer to the realizability question, it may significantly increase the sizes of the synthesized processes.



For all the architectures, we define the *composition* of strategies $f_1, \ldots, f_n$ as a function $f : (2^{O_{env}})^* \to 2^{O \cup H}$ that describes the joint behavior of the processes on an infinite sequence of external input signals. The exact definition of a composition depends on the particular architecture as well as on assumptions on the communication (e.g., whether communication involves a delay). We define several compositions in Section 5. In [PR90], Pnueli and Rosner study one-way channels (called "hierarchical architectures" there) where communication involves no delay. In this setting, compositions are defined as follows. For the strategy $\langle (2^{I_i})^*, f_i \rangle$, let $\langle (2^{I_i})^*, f_i' \rangle = mem(\langle (2^{I_i})^*, f_i \rangle)$. Recall that in a one-way chain, $O_{env} = I_1$. Then, $f : (2^{O_{env}})^* \to 2^{O \cup H}$ is such that for every $\sigma \in (2^{O_{env}})^*$, we have

$$f(\sigma) = f_1(\sigma) \cup f_2(f_1'(\sigma)) \cup f_3(f_2'(f_1'(\sigma))) \cup \ldots \cup f_n(f_{n-1}'(\cdots(f_2'(f_1'(\sigma)))\cdots)).$$

Intuitively, for all i, the output of P_i (and, consequently, the contribution of f_i to f), depends on the history of the outputs of P_{i-1}, namely the memoryfull version of f_{i-1}, which by itself depends on the memoryfull version of f_{i-2}, and so on.

The composition f induces the *computation tree* of $P_1, \ldots, P_n$, which is the $(2^{I \cup O \cup H \cup H_{env}})$-labeled $(2^{O_{env} \cup H_{env}})$-tree $xray(wide_{(2^{H_{env}})}(\langle (2^{O_{env}})^*, f \rangle))$. The transition from the composition to the computation tree involves two transformations. First, while the composition f corresponds to the composition as seen by the processes, and thus ignores the signals in H_{env} and the nondeterminism induced by them, the computation tree corresponds to the composition as seen by someone that sees all signals, which involves a $2^{H_{env}}$-widening. In addition, as the signals in O_{env} and H_{env} are represented in the widening of the composition only in its nodes and not in its labels, we employ *xray* and obtain a tree whose labels refer to all signals.

Given a CTL* formula ψ over $I \cup O \cup H \cup H_{env}$, and an architecture Ω with processes $P_1, \ldots, P_n$, we say that ψ is *realizable* in Ω iff there are strategies for $P_1, \ldots, P_n$ whose composition induces a computation tree that satisfies ψ. The *synthesis problem* is then to construct these strategies. The synthesis problem for one-way chains with complete information is introduced and solved in [PR90] for specifications in the linear temporal logic LTL (which is a strict subset of CTL*). The synthesis problem for CTL* for an architecture with a single process with incomplete information is introduced and solved in [KV99]. In this paper, we solve the synthesis problem for CTL* for the four classes of architectures introduced above. Our solution is based on automata on infinite trees. For our purposes, the crucial feature of CTL* is the following translation of CTL* formulas to alternating Rabin tree automata.

Theorem 3.1 [KVW00] *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.*

4 Useful automata constructions

Let X, Y, and Z be finite sets, and let z_0 be the root direction of Z. For an $(X \times Y)$-labeled Z-tree $\langle Z^*, f \rangle$,

[2] Indeed P_j, for $j > i$, generates also hidden signals, but these signals are generated by a strategy that is known to P_i, since our framework assumes that the processes are collaborative, while the environment is adversarial.

we say that $\langle Z^*, f\rangle$ is a *composition* of an X-labeled Z-tree $\langle Z^*, f_X\rangle$, where $mem(\langle Z^*, f_X\rangle) = \langle Z^*, f'_X\rangle$, and a Y-labeled X-tree $\langle X^*, f_Y\rangle$ iff for every z_1 and z_2 in Z and for every $\sigma \in Z^*$, we have

- $f(\epsilon) = f_X(\epsilon) \cup f_Y(\epsilon)$.
- $f(z_1) = f_X(z_0) \cup f_Y(f'_X(\epsilon))$.
- $f(\sigma \cdot z_1 \cdot z_2) = f_X(z_0 \cdot \sigma \cdot z_1) \cup f_Y(f'_X(z_0 \cdot \sigma))$.

We then say that $f = f_X + f_Y$. For a set $\mathcal{T}$ of $(X \times Y)$-labeled Z-trees, the set $shape_X(\mathcal{T})$ consists of all Y-labeled X-trees $\langle X^*, f_Y\rangle$ for which there exists an X-labeled Z-tree $\langle Z^*, f_X\rangle$ such that the $(X \times Y)$-labeled Z-tree $\langle Z^*, f_X + f_Y\rangle$ is in $\mathcal{T}$.

Theorem 4.1 *Let X, Y, and Z be finite sets. Given a nondeterministic tree automaton $\mathcal{A}$ over $(X \times Y)$-labeled Z-trees, we can construct an alternating tree automaton $\mathcal{A}'$ over Y-labeled X-trees such that $\mathcal{L}(\mathcal{A}') = shape_X(\mathcal{L}(\mathcal{A}))$ and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

Proof: Let $\mathcal{A} = \langle X \times Y, Q, q_0, \delta, \alpha\rangle$. Then, $\mathcal{A}' = \langle Y, Q, q_0, \delta', \alpha\rangle$, where for every $q \in Q$ and $y \in Y$, we have

$$\delta'(q, y) = \bigvee_{\substack{x \in X,\\ \langle s_1, s_2, \ldots, s_{|Z|}\rangle \in \delta(q, \langle x, y\rangle)}} (x, s_1) \wedge (x, s_2) \wedge \ldots \wedge (x, s_{|Z|}).$$

Consider first the case where $q = q_0$ and $\mathcal{A}'$ reads the root of the input tree $\langle X^*, f_Y\rangle$. The letter y read at the root is $f_Y(\epsilon)$. Since in $f_X + f_Y$ the root is labeled $\langle f_X(\epsilon), f_Y(\epsilon)\rangle$, we proceed according to $\delta(q_0, \langle x, y\rangle)$ for some x which is our guess for $f_X(\epsilon)$. By the definition of δ', each copy of $\mathcal{A}$ that is sent to direction $z \in Z$ and visits state s induces a copy of $\mathcal{A}'$ that is sent to direction x and visits the state s. Since the choice of x is joint to all $z \in Z$, all the copies of $\mathcal{A}'$ induced as above are going to read the same letter, which is our guess for $f_Y(f_X(\epsilon))$. Consider now a copy of $\mathcal{A}$ that reads a node $z \in Z$ and visits state s. Recall that the automaton $\mathcal{A}'$ then has a copy that reads the node $f_X(\epsilon)$, visits the state s, and the letter y read by this copy (and all the other copies that read the node $f_X(\epsilon)$) is our guess for $f_Y(f_X(\epsilon))$. Since in $f_X + f_Y$ the node z is labeled $\langle f_X(z_0), f_Y(f_X(\epsilon))\rangle$, we proceed according to $\delta(s, \langle x, y\rangle)$, for some x which is our guess for $f_X(z_0)$. Each copy of $\mathcal{A}$ that is sent to direction $z' \in Z$ and visits state s' then induces a copy of $\mathcal{A}'$ that is sent to direction x and visits the state s'. All these copies are going to read the same letter, which is our guess for $f_Y(f'_X(z_0))$. The same idea repeats in further levels: a copy of $\mathcal{A}$ that reads a node $\sigma \cdot z_1 \cdot z_2 \in Z^*$ and visits state s is associated with a copy of $\mathcal{A}'$ that reads the node $f'_X(z_0 \cdot \sigma)$ and visits the state s. The letter y read by this copy (and all the other copies that read the node $f'_X(z_0 \cdot \sigma)$) is our guess for $f_Y(f'_X(z_0 \cdot \sigma))$. Since in $f_X + f_Y$ the node $\sigma \cdot z_1 \cdot z_2$ is labeled $\langle f_X(z_0 \cdot \sigma \cdot z_1), f_Y(f'_X(z_0 \cdot \sigma))\rangle$, we proceed according to $\delta(s, \langle x, y\rangle)$ for some x which is our guess for $f_X(z_0 \cdot \sigma \cdot z_1)$. All the copies sent to direction x are going to read the same letter, which is our guess for $f_Y(f'_X(z_0 \cdot \sigma \cdot z_1))$. □

Given a nondeterministic tree automaton $\mathcal{A}$, let $shape_X(\mathcal{A})$ denote the corresponding automaton $\mathcal{A}'$ constructed in Theorem 4.1. Note that while $shape_X(\mathcal{A})$ returns an alternating tree automaton, it is defined for a nondeterministic tree automaton $\mathcal{A}$. Thus, successive applications of *shape* require an intermediate application of the exponential alternation-removal procedure in Theorem 2.1.

The construction described in Theorem 4.1 will help us to solve the realizability problem by successively reducing the number of processes in the architectures. The two constructions below will handle the external input to the system and the incomplete information, and they are presented in [KV99], where they are used for the synthesis of a single process with incomplete information.

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 $\mathcal{A}'$ accepts a labeled tree $\langle \Upsilon^*, V\rangle$ iff $\mathcal{A}$ accepts $xray(\langle \Upsilon^*, V\rangle)$, and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

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 $\mathcal{A}'$ accepts a Z-labeled tree $\langle X^*, V\rangle$ iff $\mathcal{A}$ accepts the Z-labeled tree $wide_Y(\langle X^*, V\rangle)$, and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

Finally, since we want our algorithm to be applicable also for settings in which communication involves a delay, we need a construction that handles such a delay.

Theorem 4.4 *Given an alternating tree automaton $\mathcal{A}$ over Σ-labeled Υ-trees, we can construct an alternating tree automaton $\mathcal{A}'$ over Σ-labeled Υ-trees such that $\mathcal{A}'$ accepts a labeled tree $\langle \Upsilon^*, V\rangle$ iff $\mathcal{A}$ accepts $delay(\langle \Upsilon^*, V\rangle)$, and the automata $\mathcal{A}'$ and $\mathcal{A}$ have the same size and index.*

Given an alternating tree automaton $\mathcal{A}$, let $cover(\mathcal{A})$, $narrow_Y(\mathcal{A})$, and $wait(\mathcal{A})$ denote the corresponding automata $\mathcal{A}'$ constructed in Theorems 4.2, 4.3 (for a set Y of directions), and 4.4, respectively.

5 Solving the synthesis problem

In this section we study the synthesis problem for the architectures described in Section 3. We show that for all the four classes, the problem is decidable, with a nonelementary complexity. Thus, given a CTL* formula ψ, a class $\mathcal{C}$ (one-way chain, two-way chain, one-way ring, or two-way ring), and an integer n, the complexity of constructing n strategies for n processes in an architecture of class $\mathcal{C}$ that satisfies ψ is n-$exp(|\psi|)$.[3]

One-way chain We assume that communication involves a delay. Thus, the input to P_{i+1} at time t is the output of P_i (or the environment, when $i = 0$) at time $t-1$. Accordingly, we define the *composition* f of $f_1, \ldots, f_n$ as follows. For a string $\sigma = z_0 \cdot z_1 \cdots z_k$ and $i \geq 0$, let $z_0 \cdot z_1 \cdots z_{k-i}$ be either the prefix of length $k-i+1$ of σ, in case $k - i \geq 0$, or ϵ, in case $k - i + 1 \leq 0$. Also, let z_0 be the root direction of 2^{I_1}. Then, $f : (2^{I_1})^* \to 2^{O \cup H}$ is defined as follows.

- $f(\epsilon) = f_1(\epsilon) \cup \cdots \cup f_n(\epsilon)$.
- For $\sigma \in (2^{I_1})^*$ with $\sigma = z_1 \cdots z_k$, we have $f(\sigma) = f_1(z_0 \cdot z_1 \cdots z_{k-1}) \cup f_2(f'_1(z_0 \cdot z_1 \cdots z_{k-2})) \cup \cdots \cup f_n(f'_{n-1}(z_0 \cdot z_1 \cdots z_{k-n}))$.

Consider a CTL* formula ψ over $I \cup O \cup H \cup H_{env}$. Recall that in a one-way chain, we have $I \cup O = I_1 \cup O$. In order to solve the realizability problem, we build the following tree automata.

- $\mathcal{A}_\psi$: an alternating Rabin tree automaton that accepts a $(2^{I_1 \cup O \cup H \cup H_{env}})$-labeled $(2^{I_1 \cup H_{env}})$-tree $\langle (2^{I_1 \cup H_{env}})^*, f \rangle$ iff it satisfies ψ [see Theorem 3.1].
- $\mathcal{A}_0$: the alternating Rabin tree automaton $wait(\mathcal{A}_\psi)$. Thus, $\mathcal{A}_0$ accepts a $(2^{I_1 \cup O \cup H \cup H_{env}})$-labeled $(2^{I_1 \cup H_{env}})$-tree $\langle (2^{I_1 \cup H_{env}})^*, f \rangle$ iff $delay(\langle (2^{I_1 \cup H_{env}})^*, f \rangle)$ satisfies ψ [see Theorem 4.4].
- $\mathcal{A}'_0$: the alternating Rabin tree automaton $cover(\mathcal{A}_0)$. Thus, $\mathcal{A}'_0$ accepts a $(2^{O \cup H})$-labeled $(2^{I_1 \cup H_{env}})$-tree $\langle (2^{I_1 \cup H_{env}})^*, f \rangle$ iff $delay(xray(\langle (2^{I_1 \cup H_{env}})^*, f \rangle))$ satisfies ψ [see Theorem 4.2].
- $\mathcal{A}''_0$: the alternating Rabin tree automaton $narrow_{(2^{H_{env}})}(\mathcal{A}'_0)$. Thus, $\mathcal{A}''_0$ accepts a $(2^{O \cup H})$-labeled 2^{I_1}-tree $\langle (2^{I_1})^*, f \rangle$ iff $delay(xray(wide_{(2^{H_{env}})}(\langle (2^{I_1})^*, f \rangle)))$ satisfies ψ [see Theorem 4.3].
- For $1 \leq i \leq n-1$,
  - $\mathcal{A}_i$: a nondeterministic Rabin tree automaton equivalent to $\mathcal{A}''_{i-1}$ [see Theorem 2.1]. Note that the automaton $\mathcal{A}_i$ runs on $(2^{O_i \cup H_i \cup O_{i+1} \cup H_{i+1} \cup \cdots \cup O_n \cup H_n})$-labeled $2^{O_{i-1}}$-trees, where we take $O_0 = I_1$.
  - $\mathcal{A}'_i$: the alternating Rabin automaton $shape_{(2^{O_i \cup H_i})}(\mathcal{A}_i)$. Thus, $\mathcal{A}'_i$ runs on $(2^{O_{i+1} \cup H_{i+1} \cup \cdots \cup O_n \cup H_n})$-labeled $(2^{O_i \cup H_i})$-trees and it accepts a tree $\langle (2^{O_i \cup H_i})^*, f \rangle$ iff there is a $(2^{O_i \cup H_i})$-labeled $2^{O_{i-1}}$-tree $\langle (2^{O_{i-1}})^*, f' \rangle$ such that $\langle (2^{O_{i-1}})^*, f + f' \rangle$ is accepted by $\mathcal{A}_i$ [see Theorem 4.1].
  - $\mathcal{A}''_i$: the alternating Rabin automaton $narrow_{(2^{H_i})}(\mathcal{A}'_i)$. Thus, $\mathcal{A}''_i$ accepts a $(2^{O_{i+1} \cup H_{i+1} \cup \cdots \cup O_n \cup H_n})$-labeled 2^{O_i}-tree $\langle (2^{O_i})^*, f \rangle$ iff $wide_{(2^{H_i})}(\langle (2^{O_i})^*, f \rangle)$ is accepted by $\mathcal{A}'_i$ [see Theorem 4.3].

Intuitively, in each iteration $1 \leq i \leq n$, we assume that the strategies of $P_1, \ldots, P_{i-1}$ are given (they are encapsulated in the transition function of $\mathcal{A}_i$) and the automaton $\mathcal{A}_i$ accepts all the compositions of $P_i, \ldots P_n$ that together with the given strategies satisfy ψ. Thus, the transition from $\mathcal{A}_i$ to $\mathcal{A}_{i+1}$ involves an encapsulation of the possible strategies of P_i (and how they affect the behavior required from $P_{i+1}, \ldots, P_n$ in order to satisfy ψ) into the transition function of $\mathcal{A}_i$.

Lemma 5.1 *ψ is realizable iff $\mathcal{A}'_{n-1}$ is not empty.*

The construction of $\mathcal{A}_i$ goes via i iterations. Each iteration involves two automata transformations. One transformation (*narrow*) gets and returns an alternating tree automaton. The other transformation (*shape*) gets a nondeterministic tree automaton and return an alternating tree automaton. While all the transformations involve no blow-up in the size of the automata, the fact that *shape* handles nondeterministic automata requires the application of an additional transformation, namely the translation of an alternating tree automaton to a nondeterministic one. This transformation involves an exponential blow-up, leading to an overall nonelementary blow-up.

[3] n-$exp(k)$ is a stack of n exponents with k on the top; i.e., 1-$exp(k) = 2^{O(k)}$, and $(i+1)$-$exp(k) = 2^{i\text{-}exp(k)}$.

Theorem 5.2 *The synthesis problem for CTL* and one-way chains is nonelementary decidable.*

Proof: It follows from the constructions described in Section 4 that the size of $\mathcal{A}''_{n-1}$ is $(n-1)$-$exp(|\psi|)$. The nonemptiness problem for $\mathcal{A}''_{n-1}$ can then be solved in time n-$exp(|\psi|)$ [MS95, KV98]. Lemma 5.1 then implies that the realizability problem for ψ can be solved in time n-$exp(|\psi|)$. The nonemptiness algorithm can be extended to produce a witness for the automaton being nonempty (in fact, a witness that is a *memoryless strategy* [Tho95]). A witness for the nonemptiness of $\mathcal{A}''_{n-1}$ induces a strategy f_n for P_n. In order to get a strategy for P_{n-1}, we combine $\mathcal{A}''_{n-2}$ with f_n and get an automaton that is guaranteed to be nonempty and whose witness induces a strategy f_{n-1} for P_{n-1}. We continue similarly until strategies for all processes are synthesized. □

A matching nonelementary lower bound is proved (for LTL formulas) in [PR90] (cf. [PR79]). This lower bounds applies also to the other architecture.

With appropriate simple modifications (skipping the "wait construction" and redefining the "shape construction" to ignore the delay), the method described above can handle one-way channels in which communication involves no delay (the definition of composition then coincides with the one of [PR90]). As we describe below, the method can also be extended to handle the other classes of architectures described in Section 3. The differences among the architectures influence the sets of labels and directions of the trees over which the automata are defined (for example, in a one-way ring $\mathcal{A}_\psi$ runs on $(2^{O_{env} \cup O_n})$-trees, and in a two-way ring, it runs on $(2^{O_{env} \cup O_2 \cup O_n})$-trees), influence the definition of composition, and accordingly influence the definition of $shape_X(\mathcal{T})$ and the "shape construction" that handles. For all the architectures, however, the idea is similar: a successive reduction in the number of processes, where in each step we omit a process and encapsulate its possible strategies into the transition function of intermediate automata.

One-way ring. Recall that in a one-way ring, the process P_1 reads signals from both P_n and the environment. We suggest two alternative modifications to the method presented for one-way chains. The first is rather simple: all the intermediate automata we construct maintain (in their alphabet) the input that P_1 reads from P_n. Then, in the last automaton, which corresponds to P_n's strategy, we close the ring by requiring the output of P_n to agree with the maintained input. The second approach is cleaner (and it also has a computational advantage), yet it requires a more substantial modification. The idea is to start with P_1 and proceed in both directions, encapsulating two processes in each iteration. The two directions meet at the automaton $\mathcal{A}_{\frac{n}{2}}$, whose nonemptiness witnesses a strategy for $P_{\frac{n}{2}}$ that satisfies the tasks inherited to $P_{\frac{n}{2}}$ by both the processes to his left and these to his right.

Two-way chain. The two-way chain architecture is much richer than that of a one-way chain. Since the difficulties imposed by incomplete information are orthogonal and are handled by the narrow construction, we describe here the solution for systems with complete information, thus $H_{env} \cup H = \emptyset$. In a two-way chain, the process P_i reads both O_{i-1} and O_{i+1}, so its strategy is a function $f_i : (2^{O_{i-1} \cup O_{i+1}})^* \to 2^{O_i}$. Accordingly, while in the case of a one-way chain the reduction of the process P_i involves a transition from an automaton that runs on $(2^{O_i \cup O_{i+1} \cup \cdots \cup O_n})$-labeled $2^{O_{i-1}}$-trees to an automaton that runs on $(2^{O_{i+1} \cup \cdots \cup O_n})$-labeled 2^{O_i}-trees, here the reduction of P_i should involve a transition from an automaton that runs on $(2^{O_i \cup O_{i+1} \cup \cdots \cup O_n})$-labeled $(2^{O_{i-1} \cup O_{i+1}})$-trees to an automaton that runs on $(2^{O_{i+1} \cup \cdots \cup O_n})$-labeled $(2^{O_i \cup O_{i+2}})$-trees. In order to see the modifications that are therefore needed in the shape construction, let us first redefine the predicate *shape* and the composition operator it involves.

Let X_{i-1}, X_i, X_{i+1}, X_{i+2}, and X be finite sets, and let z_0 and z'_0 be the root directions of X_{i-1} and X_{i+1} respectively. For our application, X_j stands for 2^{O_j}, and X stands for $2^{O_{i+3} \cup \ldots \cup O_n}$. For an $(X_i \times X_{i+1} \times X_{i+2} \times X)$-labeled $(X_{i-1} \times X_{i+1})$-tree $\langle (X_{i-1} \times X_{i+1})^*, f \rangle$, we say that $\langle (X_{i-1} \times X_{i+1})^*, f \rangle$ is a *composition* of an X_i-labeled $(X_{i-1} \times X_{i+1})$-tree $\langle (X_{i-1} \times X_{i+1})^*, f_1 \rangle$ and an $(X_{i+1} \times X_{i+2} \times X)$-labeled $(X_i \times X_{i+2})$-tree $\langle (X_i \times X_{i+2})^*, f_2 \rangle$ iff for every $\langle z_1, z'_1 \rangle$ and $\langle z_2, z'_2 \rangle$ in $X_{i-1} \times X_{i+1}$ and for every $\sigma \in (X_{i-1} \times X_{i+1})^*$, we have ($f'$ and f'_1 are the memoryfull versions of f and f'):

- $f(\epsilon) = \langle f_1(\epsilon), f_2(\epsilon) \rangle$.
- $f(\langle z_1, z'_1 \rangle) = \langle f_1(\langle z_0, z'_0 \rangle), f_2(f'_1(\epsilon)) \rangle$.
- $f(\sigma \cdot \langle z_1, z'_1 \rangle \cdot \langle z_1, z'_1 \rangle) = \langle f_1(\langle z_0, z'_0 \rangle \cdot \sigma \cdot \langle z_1, z'_1 \rangle), f_2(f'_1(\langle z_0, z'_0 \rangle \cdot \sigma) \oplus f'(\langle z_0, z'_0 \rangle \cdot \sigma)_{|X_{i+2}}) \rangle$, where $\oplus$ is bitwise concatenation (e.g., $y_1 \cdot y_2 \oplus y_3 \cdot y_4 = \langle y_1, y_3 \rangle \cdot \langle y_2, y_4 \rangle$) and $\tau_{|X_{i+2}}$ is the projection of τ on X_{i+2}.

We then say that $f = f_1 + f_1$. Intuitively, f determines its X_i-element according to f_1 and determines the $(X_{i+1} \times X_{i+2} \times X)$-element by applying

f_2 on an interleaving of an application of f'_1, which gives the X_i element and an application of f' on a strict prefix of the input, which returns an element in $X_i \times X_{i+1} \times X_{i+2} \times X$ and is then projected on X_{i+2}. In addition, since we assume that communication involves a delay, f ignores the last letters in a sequence and refers instead to the root directions.

For a set $\mathcal{T}$ of $(X_i \times X_{i+1} \times X_{i+2} \times X)$-labeled $(X_{i-1} \times X_{i+1})$-trees, the set $shape_{X_i \times X_{i+2}}(\mathcal{T})$ consists of all $(X_{i+1} \times X_{i+2} \times X)$-labeled $(X_i \times X_{i+2})$-trees $\langle (X_i \times X_{i+2})^*, f_2 \rangle$ for which there exists an X_i-labeled $(X_{i-1} \times X_{i+1})$-tree $\langle (X_{i-1} \times X_{i+1})^*, f_1 \rangle$ such that $\langle (X_{i-1} \times X_{i+1})^*, f_1 + f_2 \rangle$ is in $\mathcal{T}$.

The shape construction in Theorem 4.1 can be modified to handle the definition of shape above. Essentially, while in the current construction the automaton $\mathcal{A}'$ guesses in each transition a direction x to proceed with, in the new construction $\mathcal{A}'$ needs to guess two elements, corresponding to both X_i and X_{i+2}, and it should remember the X_{i+2} element for the projection described above.

Two-way ring. The solution for two-way rings is based on the modified shape construction described for two-way chains and the "two-direction reasoning" described for one-way rings.

The important common property of the four classes we handle is the fact that there are no two processes both reading input from the envirponmrnt. Consequently, the processes can be linearly ordered according to the signals they know. More architectures fall in this category. For example, it is possible to replace a single processes in a chain by a group of processes that share the same knowladge, and adjust the synthesis algorithms accordingly. An exact characterization of architectures for which the synthesis problem is decidable is an open problem.

6 Discussion

One of the most significant developments in the area of system verification over the last decade is the development of algorithmic methods for verifying temporal specifications of finite-state systems [CGP99]. 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. A frequent criticism against this approach, however, is that verification is done after significant resources have already been invested in the development of the program. Since systems typically contain errors, verification simply becomes part of the development process. The critics argue that the desired goal is to use the specification in the system development process in order to guarantee the design of correct systems. This is exactly what synthesis algorithms do. Despite this criticism, synthesis tools are not as popular in the industry as verification tools. There are several reasons for that: the scope of synthesis algorithms has been quite limited, their complexity is high, and they do not always produce practical systems, where practicality is measured in a variety of ways, such as optimality (say, number of latches required for implementing the system in hardware, or number of messages needed to be passed between the underlying processes), testability (the ability to test hardware without access to all the internal variables), and the like.

In this paper, we significantly extended the scope of synthesis to include many practical applications. We claim that the high complexity of the problem is not really a serious objection to the potential usefulness of synthesis. First, we note that experience with verification shows that nonelementary algorithms can nevertheless be practical, since the worst-case complexity does not arise often. For example, while the model-checking problem for specifications in second-order logic has nonelementary complexity, the model-checking tool MONA [EKM98, Kla98] successfully verifies many specifications given in second-order logic. Second, we argue that synthesis is not harder than verification. This may sound as a wishful thinking, as it contradicts the known fact that while verification is easy (linear in the size of the model and at most exponential in the size of the specification), synthesis is hard (nonelementary). There is, however, something misleading in this fact: while the complexity of synthesis is given in terms of the specification, the complexity of verification is given with respect to both the specification and the (much bigger) system. In particular, in a distributed setting, it is shown in [Ros92] that there are LTL specifications ψ_n, of length $O(n)$, and architectures with k processes such that the smallest strategy that realizes ψ_n in the given architecture has $k\text{-}exp(n)$ states. What is the complexity of verifying whether a system satisfies ψ_n? Even if verification is linear in the size of the system, it would be nonelementary in n for correct systems, just as the synthesis problem, since such systems necessarily have at least $k\text{-}exp(n)$ states!

In summary, we believe that the real challenge that synthesis algorithms and tools face in the coming years is mostly not that dealing with computational complexity, but rather that of making automatically synthesized systems more practically useful.

References

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th ICALP*, LNCS 372, pp. 1–17, 1989.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Chu63] A. Church. Logic, arithmetics, and automata. In *Proc. International Congress of Mathematicians, 1962*, pp. 23–35. institut Mittag-Leffler, 1963.

[Dij72] E.W. Dijksta. *Hierarchical ordering of sequential processes, Operating systems techniques.* Academic Press, 1972.

[Dil89] D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits.* MIT Press, 1989.

[DTV99] M. Daniele, P. Traverso, and M.Y. Vardi. Strong cyclic planning revisited. In S. Biundo and M. Fox, editors, *5th European Conference on Planning*, pp. 34–46, 1999.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th FOCS*, pp. 328–337, 1988.

[EKM98] J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: new techniques for WS1S and WS2S. In *Proc 10th CAV*, LNCS 1427, pp. 516–520, 1998.

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

[Kla98] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Proc CSL '97*, LNCS, 1997.

[KMTV00] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc. 11th CONCUR*, LNCS 1877, pp. 92–107, 2000.

[KV98] O. Kupferman and M.Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proc. 30th STOC*, pp. 224–233, 1998.

[KV99] O. Kupferman and M.Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245 – 263, June 1999.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Man99] Microsoft LAN Manager. The protocol stack. http://www.rit.edu/~trb5541/p2_stack.html, 1999.

[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 alternating tree automata by nondeterministic automata: New results and new proofs of theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141:69–107, 1995.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM TOPLAS*, 6(1):68–93, January 1984.

[PR79] G.L. Peterson and J.H. Reif. Multiple-person alternation. In *Proc. 20st IEEE Symp. on Foundation of Computer Science*, pp. 348–363, 1979.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th POPL*, pp. 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th ICALP*, LNCS 372, pp. 652–671, 1989.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st FOCS*, pp. 746–757, 1990.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical recipes in C.* Cambridge University Press, 1992.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems.* PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *IEEE Transactions on Control Theory*, 77:81–98, 1989.

[Tan87] A.S. Tanenboum. *Operating systems, design and implementation.* Prentice-Hall International Editors, New Jersy, 1987.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In E.W. Mayr and C. Puech, editors, *Proc. 12th TACAS*, LNCS 900, pp. 1–13, 1995.

[Var95] M.Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. *Proc 7th CAV*, LNCS 939, pp. 267–292, 1995.