Synthesis of Asynchronous Systems

Sven Schewe and Bernd Finkbeiner

Universität des Saarlandes, 66123 Saarbrücken, Germany
{schewe|finkbeiner}@cs.uni-sb.de

Abstract. This paper addresses the problem of synthesizing an asynchronous system from a temporal specification. We show that the cost of synthesizing a single-process implementation is the same for synchronous and asynchronous systems (2EXPTIME-complete for CTL* and EXPTIME-complete for the μ-calculus) if we assume a full scheduler (i.e., a scheduler that allows every possible scheduling), and exponentially more expensive for asynchronous systems without this assumption (3EXPTIME-complete for CTL* and 2EXPTIME-complete for the μ-calculus). While multi-process synthesis for synchronous distributed systems is possible for certain architectures (like pipelines and rings), we show that the synthesis of asynchronous distributed systems is decidable if and only if at most one process implementation is unknown.

1 Introduction

Synthesis automatically transforms a specification into an implementation that is guaranteed to satisfy the specification. For *synchronous* systems, the synthesis problem is well-understood. Synthesizing single-process implementations is EXPTIME-complete for the μ-calculus [5, 9], and 2EXPTIME-complete for linear-time temporal logic (LTL) and computation-tree logic (CTL*) [7, 9, 6]. Multi-process synthesis, the problem of finding implementations for the processes in a given distributed architecture, has been solved for pipelines [13], rings [10], and in general for all architectures without information forks (i.e., pairs of processes with incomparable information) [3].

By contrast, the problem of synthesizing *asynchronous* systems has so far received very little attention: the synthesis algorithms in the literature are limited to LTL specifications and single-process implementations. The first solution for asynchronous synthesis with specifications in LTL, but without fairness conditions, is due to Pnueli and Rosner [12]. Anuchitanukul and Manna [1] later showed that fairness conditions can be included in a deductive approach; Vardi [14] provided an automata-based algorithm for the same problem.

The question arises if the lack of synthesis algorithms for asynchronous systems is a coincidence or rather an indication of an inherent hardness of the synthesis problem for asynchronous systems. In this paper, we systematically study the challenges in extending synthesis to the asynchronous case and, in doing so, give a comprehensive answer to this question.

Challenge 1: Synthesizing asynchronous processes for branching-time specifications. We begin by generalizing the synthesis of single-process implementations from linear-time to branching-time specifications. The behavior of an asynchronous process depends on the scheduler: while synchronous processes are aware of each change to their inputs, asynchronous processes may fail to see certain changes (when the writing process is scheduled more often than the reading process) and may see duplicate input values (when the reading process is scheduled multiple times between two writes). For linear-time specifications, asynchronous processes are typically analyzed in combination with a *full* scheduler, which allows every possible scheduling to occur along some path of the computation tree. In our first algorithm, we adapt this setting to branching-time specifications and synthesize an asynchronous process implementation such that the computation tree that results from the combination with a full scheduler satisfies the branching-time specification. The algorithm runs in exponential time for μ-calculus specifications and in double exponential time for CTL*. We thus obtain the result that *under full scheduling, the cost of synthesizing single-process implementations is the same for synchronous and asynchronous systems.*

Challenge 2: Synthesizing scheduler-independent implementations. Dropping the assumption of a full scheduler leads to the problem of synthesizing *scheduler-independent* implementations: we require that the implementation must satisfy the specification for *every* scheduler. For LTL (and, more generally, for universal specifications), the two synthesis problems coincide. For branching-time specifications, scheduler-independent synthesis is the strictly more general problem. Consider the existential specification "there is a path where the output of the process changes in every second step." This specification can trivially be satisfied under the assumption of a full scheduler, but there is no implementation that guarantees this specification for all schedulers. Scheduler-independent synthesis allows us to explicitly state the assumptions on the scheduler as part of the specification. An interesting example for such an assumption is fairness. While synthesis under full scheduling allows us to find implementations that perform correctly on *fair paths* ("there is a *fair scheduling* where the output of the process changes in every second step"), scheduler-independent synthesis allows us to find implementations that perform correctly whenever the scheduling is fair ("if the *scheduler* is *fair* on all paths then there is a path where the output of the process changes in every second step"). In our second algorithm, we synthesize an asynchronous process implementation such that *any* computation tree that results from the combination of the process with some scheduler satisfies the branching-time specification. The algorithm runs in double exponential time for μ-calculus specifications and in triple exponential time for CTL*. We provide matching lower bounds for both logics, obtaining the result that *scheduler-independent synthesis is exponentially harder than synthesis under full scheduling.*

Challenge 3: Synthesizing asynchronous distributed systems. We finally consider the multi-process synthesis problem, where the distributed architecture is given as a directed graph. Each process is either identified as black-box, if its implementation is to be determined by the synthesis algorithm, or as

white-box, if the implementation is already known and fixed. In the synchronous case, the distributed synthesis problem is decidable if and only if the architecture does not contain an information fork [3]. We show that in the asynchronous case, *the distributed synthesis problem is decidable if and only if the architecture contains only a single black-box process.*

Our results thus demonstrate that, except for the case of single-process implementations and full scheduling, the synthesis of asynchronous systems is indeed harder than the synthesis of synchronous systems. Our algorithms solve the distributed synthesis problem for architectures with a single black-box process. Since the synthesis problem is undecidable for all architectures with two or more black-box processes, it is impossible to extend our algorithms to a larger set of architectures.

2 The Synthesis Problem

We study the synthesis problem in the general setting of distributed systems. The *synthesis problem* is to decide for the triple $(A, \varphi, \{s_w | w \in W\})$, consisting of an architecture A, a specification φ, and a set of white-box strategies $\{s_w | w \in W\}$, whether there exists a finite-state program (or *strategy*) for each black-box process in A, such that the joint behavior satisfies φ.

Architectures. An architecture A is a tuple (P, W, p_{env}, E, O, H), where P is a set of processes with a subset $W \subset P$ of white-box processes and a distinguished environment process $p_{env} \in P \setminus W$. (P, E) is a directed graph, $O = \{O_e | e \in E\}$ a set of nonempty sets of (output) variables for every edge and $H = \{H_p | p \in P\}$ a pairwise disjoint set of (possibly empty) sets of hidden variables for every process such that $\bigcup_{e \in E} O_e \cap \bigcup_{p \in P} H_p = \emptyset$ and $e, e' \in E, O_e \cap O_{e'} \neq \emptyset \Rightarrow pr_1(e) = pr_1(e')$ hold (where pr_1 denotes the projection on the first element).

As additional notation, we use $V = \bigcup_{e \in E} O_e \cup \bigcup_{p \in P} H_p$ for the set of variables, $I_p = \bigcup_{p' \in P} O_{(p',p)}$ and $O_p = \bigcup_{p' \in P} O_{(p,p')} \cup H_p$ for the input and output, respectively, of a process p, and $P^- = P \setminus \{p_{env}\}$ for the set of system processes. The set $B = P^- \setminus W$ contains the black-box processes; we assume additionally that, for all $b \in B$, the output of a black-box process O_b is not empty (otherwise, the output is known and we turn b into a white-box process).

Implementations. A process p is implemented by a *strategy*, i.e., a function $s_p : (2^{I_p})^* \to 2^{O_p}$. A strategy is *finite-state* if it can be represented by a finite-state automaton. An *implementation* of an architecture consists of strategies $S = \{s_p | \, p \in B\}$ for all black-box processes.

An implementation defines a computation tree. As usual, a *tree* is given as a prefix-closed subset $Y \subseteq \Upsilon^*$ of all finite words over a given set of directions Υ. If Y contains the empty word ε and a successor for every element $(\forall y \in Y \exists \upsilon \in \Upsilon.\, y \cdot \upsilon \in \Upsilon)$, Y is called *total*, and if $Y = \Upsilon^*$, the tree is called *full*. For given finite sets Σ and Υ, a *Σ-labeled Υ-tree* is a pair $\langle Y, l \rangle$ with $Y \subseteq \Upsilon^*$ and a labeling function $l : Y \to \Sigma$ that maps every node of Y to a letter of Σ.

Computations. The computation tree identifies the system state (i.e., the values of the system variables and the currently scheduled processes) for every possible history of input assignments and scheduling decisions. For an implementation S and a set of white-box strategies $\{s_w | w \in W\}$, we define the *computation tree* as the full $2^{V \cup P}$-labeled $(2^{O_{env}} \cup \{\bot\}) \times 2^{P^-}$-tree $\langle((2^{O_{env}} \cup \{\bot\}) \times 2^{P^-})^*, c\rangle$ with the following properties[1]:

- $c(\varepsilon) = \rho_0 \cup \bigcup_{p \in P^-} s_p(\varepsilon)$ and
- $c(x \cdot (\upsilon, \pi)) = \widehat{\upsilon} \cup \pi \cup \bigcup_{p \in P^-} s_p(vis_p(x))$, with
  - $\widehat{\upsilon} = \upsilon \cup \{p_{env}\}$, if $\upsilon \neq \bot$,
  - $\widehat{\upsilon} = c(x) \cap O_{env}$, if $\upsilon = \bot$, and
  - $vis_p : ((2^{O_{env}} \cup \{\bot\}) \times 2^{P^-})^* \to (2^{I_p})^*$, which maps a path in the computation tree to the input history of a process p, i.e.,
    - $vis_p(\varepsilon) = \varepsilon$,
    - $vis_p(x \cdot (\upsilon, \pi)) = vis_p(x)$ if $p \notin \pi$, and
    - $vis_p(x \cdot (\upsilon, \pi)) = vis_p(x) \cdot (c(x) \cap I_p)$ if $p \in \pi$.

Intuitively, $\bot$ denotes that the environment is not scheduled. In this case, the values of its output variables O_{env} remain unchanged.

The Scheduler. In every step, the scheduler makes a (possibly nondeterministic) choice which processes are scheduled. In a *full* scheduler, all choices are possible in each step. In general, some choices may be disabled, and the set of choices may depend on the history of states.

We formalize the scheduler as a function from $((2^{O_{env}} \cup \{\bot\}) \times 2^{P^-})^*$ to the set of potential scheduling decisions $\mathcal{P} = 2^{2^P}$, which consists of the sets of nonempty subsets of the set of processes. We represent the function as a $\mathcal{P}$-labeled Υ_A-tree $\langle {\Upsilon_A}^*, scheduler\rangle$, where $\Upsilon_A = (2^{O_{env}} \cup \{\bot\}) \times 2^{P^-}$ denotes the set of directions, and the label refers to the nondeterministic choice of the scheduler. A scheduler $\langle {\Upsilon_A}^*, scheduler\rangle$ defines a subset $Y_{scheduler} \subseteq {\Upsilon_A}^*$ of reachable nodes of ${\Upsilon_A}^*$. $Y_{scheduler}$ can be defined inductively as the smallest subset of ${\Upsilon_A}^*$ with

- $\varepsilon \in Y_{scheduler}$,
- $\forall y \in Y_{scheduler}.\, P' \in scheduler(\upsilon) \wedge p_{env} \in P'$
  $\Rightarrow \forall O \subseteq O_{env}.\, y \cdot (O, P' \setminus \{p_{env}\}) \in Y_{scheduler}$, and
- $\forall y \in Y_{scheduler}.\, P' \in scheduler(\upsilon) \wedge p_{env} \notin P' \Rightarrow y \cdot (\bot, P') \in Y_{scheduler}$.

The Synthesis Problem. A triple $(A, \varphi, \{s_w | w \in W\})$, consisting of an architecture A, a specification φ, and a set of white-box strategies $\{s_w | w \in W\}$, is called

- *realizable under full scheduling* if there exists an implementation S, such that the computation tree $\langle {\Upsilon_A}^*, c\rangle$ of S satisfies φ; and

[1] For technical convenience, we fix $\rho_0 \subseteq O_{env} \cup P$, called the *fixed-root*, which is comparable to the fixing of a root direction in the synchronous case [7, 8].

– *scheduler-independently realizable* if there exists an implementation S with the computation tree $\langle {\Upsilon_A}^*, c\rangle$ such that, for all schedulers $\langle {\Upsilon_A}^*, \mathit{scheduler}\rangle$, $\langle Y_{\mathit{scheduler}}, c\rangle$ satisfies φ.

We call an architecture A (scheduler-independently) *decidable* if an algorithm exists that decides for all specifications φ and all sets of finite-state white-box strategies $\{s_w | w \in W\}$ if $(A, \varphi, \{s_w | w \in W\})$ is (scheduler-independently) realizable.

3 Single-Process Synthesis under Full Scheduling

In this section, we show that under the assumption of full scheduling, the cost of synthesizing single-process implementations is the same for synchronous and asynchronous systems. We develop an automata-theoretic synthesis algorithm for asynchronous systems with a single black-box process. The algorithm runs in EXPTIME in the length of a CTL or μ-calculus specification and in 2EXPTIME in the length of a CTL* specification.

3.1 Preliminaries: Tree Automata

An *alternating automaton* $\mathcal{A} = (\Sigma, Q, q_0, \delta, \alpha)$ runs on full Σ-labeled Υ-trees (for a predefined finite set Υ of directions). Q denotes a finite set of states, $q_0 \in Q$ denotes a designated initial state, δ denotes a transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon_\varepsilon)$, where Υ_ε denotes $\Upsilon \cup \{\varepsilon\}$, and α is an acceptance condition.

A *run tree* on a given Σ-labeled Υ-tree $\langle \Upsilon^*, l\rangle$ is a $Q \times \Upsilon^*$-labeled tree where the root is labeled with (q_0, ε) and where for a node n with a label (q, x) and a set $\mathit{child}(n)$ of children, the labels of these children have the following properties:

– for all children $m \in \mathit{child}(n)$ of n, the label of m is $(q_m, x \cdot \upsilon_m)$ for some $q_m \in Q$ and $\upsilon_m \in \Upsilon_\varepsilon$ such that (q_m, υ_m) is an atom of $\delta(q, l(x))$, and
– the set of atoms defined by the children of n satisfies $\delta(q, l(x))$.

A run tree is *accepting* if all its paths fulfill the acceptance condition. A *parity condition* is a function α from Q to a finite set $C \subset \mathbb{N}$ of colors. A path is accepted if the highest color appearing infinitely often is even.

A full Σ-labeled Υ-tree is accepted if it has an accepting run tree. The set of trees accepted by an alternating automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton called is empty, if its language is empty. The acceptance of a tree can also be viewed as the outcome of a game, where player *accept* chooses, for every pair $(q, \sigma) \in Q \times \Sigma$, a set of atoms of $\delta(q, \sigma)$, satisfying $\delta(q, \sigma)$, and player *reject* chooses one of these atoms, which is executed. The input tree is accepted iff player *accept* has a strategy enforcing a path fulfilling α.

If $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$, i.e., if there are no ε-transitions, the alternating automaton is called *ε-free*. A *nondeterministic* automaton is a special ε-free alternating automaton, where the image of δ consists only of such formulas that, when rewritten in disjunctive normal form, contain exactly one element of

$Q \times \{\upsilon\}$ for all $\upsilon \in \Upsilon$ in every disjunct. For nondeterministic automata, every node of a run tree corresponds to a node in the input tree. The emptiness of a nondeterministic automaton can be checked with the *emptiness game*, where player *accept* also chooses the letter of the input alphabet. A nondeterministic automaton is empty iff the emptiness game is won by *reject*.

Symmetric alternating automata are a variant of alternating automata that run on total Σ-labeled trees. For a symmetric alternating automaton $\mathcal{S} = (\Sigma, Q, q_0, \delta, \alpha)$, Q, q_0, and α are defined as before. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \{\Box, \Diamond, \varepsilon\})$ now maps a state and an input letter to a positive boolean combination over atoms that refer to *all* ($\Box$) successor nodes, *some* ($\Diamond$) successor node or the current (ε) node.

3.2 Overview

The algorithm assumes an architecture A with a single black-box process b. It starts by representing a specification φ as a symmetric alternating parity automaton $\mathcal{A}_\varphi$, which is transformed into a nondeterministic automaton $\mathcal{N}_\varphi$ that accepts a tree $\langle (2^{I_b})^*, s_b \rangle$ iff s_b is an implementation of φ. The solution of the emptiness game for $\mathcal{N}_\varphi$ then provides such an implementation.

The following are the main steps of the algorithm:

- **From formulas to automata.** We first construct the symmetric alternating parity automaton $\mathcal{A}_\varphi$, with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{M}_\varphi$ (Lemma 1).
  In this section, $\mathcal{A}_\varphi$ is only used for $2^{V \cup P}$-labeled Υ_A-trees, i.e., for trees with the shape and labeling of the computation trees.
- **From computation trees to strategy trees.** We then construct the alternating parity automaton $\mathcal{S}_\varphi$ that accepts a strategy tree $\langle (2^{I_p})^*, s_p \rangle$ iff its computation tree is accepted by $\mathcal{A}_\varphi$ (Lemma 2).
- **Nondeterminization.** In a third step, we construct a nondeterministic parity automaton $\mathcal{N}_\varphi$, with $\mathcal{L}(\mathcal{N}_\varphi) = \mathcal{L}(\mathcal{S}_\varphi)$ (Lemmata 3 and 4).
- **Strategy construction.** Finally, we construct a strategy for the black-box process such that the induced computation tree is a model of φ (or demonstrate that no such strategy exists) by solving the emptiness game for $\mathcal{N}_\varphi$ (Lemma 5).

We now describe the automata transformations of the construction in more detail.

3.3 From Formulas to Automata

The symmetric alternating parity automaton $\mathcal{A}_\varphi$ can be built from a temporal or fixed point specification φ using standard constructions.

Lemma 1. *Given a μ-calculus specification φ, we can construct a symmetric alternating automaton $\mathcal{A}_\varphi$ with $O(|\varphi|)$ states such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{M}_\varphi$ [8, 9].*
Given a CTL specification φ, we can construct a symmetric alternating automaton $\mathcal{A}_\varphi$ with $2^{O(|\varphi|)}$ states such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{M}_\varphi$ [8].* □

$\mathcal{A}_\varphi$ can be transformed into an ordinary alternating automaton over full Υ-trees by replacing each occurrence of $(q, \Box)$ and $(q, \Diamond)$ in the transition function by $\bigwedge_{\upsilon \in \Upsilon}(q, \upsilon)$ and $\bigvee_{\upsilon \in \Upsilon}(q, \upsilon)$, respectively.

3.4 From Computation Trees to Strategy Trees

The central automata transformation for asynchronous synthesis is the transformation of an alternating parity automaton $\mathcal{A}_\varphi$ recognizing a set of computation trees into an alternating parity automaton $\mathcal{S}_\varphi$ that accepts those implementations whose computation trees are accepted by $\mathcal{A}_\varphi$.

We assume that the strategies $\{s_w | w \in W\}$ of the white-box processes are given as a family of *strategy automata* $\{\mathcal{S}_w = (I_w, S_w, s_0^w, d_w, o_w) | w \in W\}$. A strategy automaton is a Moore machine, where

- a state $s \in S_w$ represents the quotient class of equivalent positions in the strategy tree $\langle (2^{I_w})^*, s_w \rangle$;
- the initial state s_0^w represents the quotient class including the root of the strategy tree;
- $o_w : S_w \to 2^{O_w}$ maps each state of $\mathcal{S}_w$ to an output label of the process w;
- upon reading an input $\upsilon \in 2^{I_w} \cup \{\varepsilon\}$, the automaton proceeds to the state $s' = d_w(s, \upsilon)$, and returns the label $o_w(s')$ of this state.
  An ε-transition does not change the state ($d_w(s, \varepsilon) = s$).

We construct the automaton $\mathcal{S}_\varphi$ accepting the strategy trees of the black-box process by simulating $\mathcal{A}_\varphi$ on the (implicitly defined) computation tree.

Given a strategy s_b of the black-box process b and a set of strategies $\{s_w | w \in W\}$ for the white-box processes, represented by strategy automata $\{\mathcal{S}_w | w \in W\}$, the computation tree $\langle ((2^{O_{env}} \cup \{\bot\}) \times 2^{P^-})^*, c \rangle$ can be constructed by setting $c : \varepsilon \mapsto \bigcup_{w \in W} \mathcal{S}_w(\varepsilon) \cup s_b(\varepsilon) \cup \rho_0$, and $c : x \cdot (\upsilon, \pi) \mapsto \bigcup_{w \in \pi \cap W} \mathcal{S}_w(c(x) \cap (I_p)) \cup \bigcup_{w \in W \setminus \pi} \mathcal{S}_w(\varepsilon) \cup \widehat{\upsilon} \cup \pi \cup s_b(vis_b(x))$, where vis_b and $\widehat{\upsilon}$ are taken from the definition of computation trees.

To obtain $\mathcal{S}_\varphi$, we add the output of the environment and of the white-box processes to the states of the automaton.

Lemma 2. *Let $A = (P, W, p_{env}, E, O, H)$ be a given architecture with $B = \{b\}$, let $\{s_w | w \in W\}$ be a given set of white-box strategies represented as strategy automata $\{\mathcal{S}_w = (I_w, S_w, s_0^w, d_w, o_w) | w \in W\}$, and let $\mathcal{A}_\varphi = (2^{V \cup P}, Q, q_0, \delta, \alpha)$ be an alternating parity automaton running on $2^{V \cup P}$-labeled $(2^{O_{env}} \cup \{\bot\}) \times 2^{P^-}$-trees. Then we can, for $S = \bigotimes_{w \in W} S_w$, construct an alternating parity automaton $\mathcal{S} = (2^{O_p}, Q \times S \times 2^{P \cup O_{env}}, q_0', \delta', \alpha')$ running on 2^{O_p}-labeled 2^{I_p}-trees that accepts a strategy tree $\langle (2^{I_p})^*, s_p \rangle$ if its computation tree is accepted by $\mathcal{A}$.*

Proof. We simulate the behavior of $\mathcal{A}_\varphi$ on the computation tree $\langle {\Upsilon_A}^*, c \rangle$ that is defined by the strategy tree $\langle (2^{I_b})^*, s_b \rangle$. First, we set q_0' to $(q_0, \{s_0^w\}_{w \in W}, \rho_0)$. For $\Sigma = 2^{V \cup P}$, $\Upsilon = \Upsilon_A = (2^{O_{env}} \cup \{\bot\}) \times 2^{P^-}$ and $\delta : Q \times \Sigma \to$

$Q \times \Upsilon_\varepsilon$ with $\delta : (q, \sigma) \mapsto b^n_{(q,\sigma)}(q_i, \upsilon_i)_{i \in \mathbb{N}_n}$ that assigns positive boolean combinations of states and directions to each state and input letter, we define $\delta' : Q \times S \times 2^{O_{env} \cup P} \times 2^{O_b} \to Q \times S \times 2^{I_p}_\varepsilon$ as $\delta' : (q, s, \sigma_{env}, \sigma_b) \mapsto b^n_{(q, \bigcup_{w \in W} o_w(s) \cup \sigma_{env} \cup \sigma_b)}(q_i, f(s, \sigma_{env}, \sigma_b, \upsilon_i), g(s, \sigma_{env}, \sigma_b, \upsilon_i))_{i \in \mathbb{N}_n}$, where f and g are auxiliary functions.

The function f preserves the correct states of the strategy automata $\mathcal{S}_w$ in S and memorizes the values of the variables controlled by the environment ($O_{env} \cup P$). The former is done by applying the correct transition functions to the states of the strategy automata, the latter by storing the correct environment output. $f : \{s^w\}_{w \in W} \times \sigma_{env} \times \sigma_b \times \upsilon \mapsto (\{s^{w\prime}\}_{w \in W}, \sigma'_{env})$ has the following properties:

- $\forall w \in W : w \notin \upsilon \Rightarrow s^{w\prime} = s^w$,
- $\forall w \in W : w \in \upsilon \Rightarrow s^{w\prime} = d_w(s^w, (\bigcup_{w \in W} o_w(s^w) \cup \sigma_{env} \cup \sigma_b) \cap I_w)$;
- $\upsilon = \varepsilon \Rightarrow \sigma'_{env} = \sigma_{env}$,
- $\upsilon \neq \varepsilon \Rightarrow \sigma'_{env} \cap P^- = \upsilon \cap P^-$ and
  - $\bot \in \upsilon \Rightarrow \sigma'_{env} \cap O_{env} = \sigma_{env} \cap O_{env}, p_{env} \notin \sigma'_{env}$ and
  - $\bot \notin \upsilon \neq \varepsilon \Rightarrow \sigma'_{env} \cap O_{env} = \upsilon \cap O_{env}, p_{env} \in \sigma'_{env}$.

The function g maps the label and direction of the computation tree to a direction in the strategy tree of b; if b is scheduled, $g(\upsilon)$ is the I_b part of the label, if b is not scheduled (including the ε-transition), the position in the strategy tree remains unchanged. $g : S \times 2^{O_{env} \cup P} \times 2^{O_b} \times \Upsilon_\varepsilon \to 2^{I_b}_\varepsilon$ has the following properties:

- $g : \upsilon \mapsto \varepsilon$ if $b \notin \upsilon$ and
- $g : \upsilon \mapsto (\bigcup_{w \in W} o_w(s^w) \cup \sigma_{env} \cup \sigma_b) \cap I_b$ if $b \in \upsilon$.

Obviously, for $\alpha' : (q, s, o) \mapsto \alpha(q)$, a strategy tree is accepted by $\mathcal{S}$ iff its computation tree is accepted by $\mathcal{A}$. □

3.5 Nondeterminization

To check $\mathcal{S}_\varphi$ for emptiness, we first eliminate the ε-transitions and then construct an equivalent nondeterministic parity automaton.

Lemma 3. *[15, 9] Given an alternating parity automaton $\mathcal{S}$ with n states, we can construct an ε-free alternating parity automaton $\mathcal{S}'$ with at most n^2 states.* □

Lemma 4. *[11, 3] Given an alternating parity automaton $\mathcal{S}_\varphi$ with n states, we can construct a nondeterministic parity automaton $\mathcal{N}_\varphi$ with $n^{O(n^2)}$ states and $O(n^2)$ colors.* □

3.6 Strategy Construction

In the last step, we obtain a strategy by solving the emptiness game of the resulting nondeterministic parity automaton $\mathcal{N}_\varphi$.

Lemma 5. *Given a nondeterministic automaton $\mathcal{N} = (\Sigma, Q, q_0, \delta, \alpha)$ running on Σ-labeled Υ-trees with n states and c colors, we can construct a regular tree accepted by $\mathcal{N}$ or show that the language of $\mathcal{N}$ is empty in time $n^{O(c)}$.*

Proof. The nonemptiness problem can be reduced to a parity game with at most $n + n^{|\Upsilon|}$ states and c colors: Player *accept* owns the states Q and chooses a label $\sigma \in \Sigma$ and a conjunction $\bigwedge_{\upsilon \in \Upsilon}(q_\upsilon, \upsilon)$ satisfying $\delta(q, \sigma)$. Player *reject* owns these conjunctions and can move from a state $\bigwedge_{\upsilon \in \Upsilon}(q_\upsilon, \upsilon)$ to a state q_υ by choosing a direction $\upsilon \in \Upsilon$. The colors of the states of player *accept* are defined by the coloring function α, while all states of player *reject* are colored by the minimum color in the mapping of α. This parity game can be solved in time $n^{O(c)}$ [4].

$\mathcal{N}$ is empty iff player *reject* has a winning strategy, and the Σ-projection of a memoryless winning strategy for player *accept* defines a regular tree. □

3.7 Complexity

The construction described in Section 3.2 provides EXPTIME and 2EXPTIME upper bounds for the synthesis problems under full scheduling in case of μ-calculus and CTL* specifications, respectively. Matching lower bounds can be inferred from the known lower bounds for the synthesis problems for CTL and CTL* in synchronous systems by applying linear specification transformations.

Theorem 1. *The distributed synthesis problem under full scheduling for architectures with a single black-box process is EXPTIME-complete for specifications in CTL and the μ-calculus and 2EXPTIME-complete for CTL* specifications.*

Proof. The upper bounds follow from the construction suggested in Section 3.2 together with the Lemmata 1 through 5.

We establish the lower bound for the special case of architectures without white-box processes. We reduce the synthesis problem for this case from the synthesis problem for the synchronous setting, which is EXPTIME-hard for CTL and 2EXPTIME-hard for CTL* [8]. The reduction is by a linear transformation of each CTL or CTL* formula φ_{sync} that reasons only over the specification variables V, to a CTL or CTL* specification φ_{async}, respectively, such that φ_{async} is realizable iff φ_{sync} if realizable in the synchronous setting.

For CTL specifications, we replace every occurrence of $A\varphi U\psi$, $E\varphi U\psi$, $AX\psi$ and $EX\psi$ by $A\varphi U(\psi \vee \neg b \vee \neg p_{env})$, $E(\varphi \wedge b \wedge p_{env})U(\psi \wedge b \wedge p_{env})$, $AX(b \wedge p_{env} \rightarrow \psi)$ and $EX(b \wedge p_{env} \wedge \psi)$, respectively.

For CTL* specifications, we replace every occurrence of $A\pi$ by $A(G(b \wedge p_{env}) \rightarrow \pi)$ and every occurrence of $E\pi$ by $E(G(b \wedge p_{env}) \wedge \pi)$.

A strategy s_b for the black-box process is obviously a realization for the transformed specification iff s_b realizes the original specification in the synchronous setting.

The EXPTIME lower bound for CTL implies the EXPTIME lower bound for the μ-calculus, and the EXPTIME upper bound for the μ-calculus establishes a matching upper bound for CTL. □

4 Synthesis of Scheduler-Independent Implementations

We now present an algorithm for *scheduler-independent* synthesis, where we only consider implementations that satisfy the specification for *all* schedulers. Scheduler-independent synthesis can also be used to find implementations that satisfy their specification if the scheduler satisfies assumptions that are explicitly stated in the specification.

4.1 Overview

We again begin with an overview over the main steps of the construction. The algorithm runs in 2EXPTIME and 3EXPTIME in the length of a μ-calculus and CTL* specification, respectively.

- **From formulas to symmetric automata.** We first construct the symmetric alternating parity automaton $\mathcal{A}_\varphi$, with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{M}_\varphi$ (Lemma 1).
- **Considering a scheduler.** We then construct the alternating parity automaton $\mathcal{B}_\varphi$ that accepts the product $\langle {\Upsilon_A}^*, \mathit{scheduler} \times c\rangle$ of a scheduler $\langle {\Upsilon_A}^*, \mathit{scheduler}\rangle$ and a computation tree $\langle {\Upsilon_A}^*, c\rangle$ iff $\langle Y_{\mathit{scheduler}}, c\rangle$ is accepted by $\mathcal{A}_\varphi$ (Lemma 6).
- **Quantification over all schedulers.** In a third step, we construct an alternating automaton $\mathcal{C}_\varphi$ that accepts a computation tree $\langle {\Upsilon_A}^*, c\rangle$ iff, for all schedulers $\langle {\Upsilon_A}^*, \mathit{scheduler}\rangle$, the product $\langle {\Upsilon_A}^*, \mathit{scheduler} \times c\rangle$ of the computation tree and the scheduler is accepted by $\mathcal{B}_\varphi$ (Lemmata 3 and 7).
- **Strategy construction.** Finally, we construct a strategy for the black-box process such that the induced computation tree is accepted by $\mathcal{C}_\varphi$ or demonstrate that no such strategy exists (Lemmata 2 through 5).

4.2 Considering a Scheduler

To check whether a symmetric alternating automaton $\mathcal{A}$ accepts $\langle Y_{\mathit{scheduler}}, c\rangle$ for a given scheduler $\langle {\Upsilon_A}^*, \mathit{scheduler}\rangle$ and a given a computation tree $\langle {\Upsilon_A}^*, c\rangle$, we use the scheduler to determine which successors of a node $y \in Y_{\mathit{scheduler}}$ are contained in $Y_{\mathit{scheduler}}$. For universal atoms $(q, \Box)$, the copies are sent only to these successors, and for existential atoms, the copy is sent to one of them.

Lemma 6. *Given a symmetric alternating automaton $\mathcal{A} = (2^{V\cup P}, Q, q_0, \delta, \alpha)$, running on total $2^{V\cup P}$-labeled trees, we can construct an alternating automaton $\mathcal{B} = (\mathcal{P}\times 2^{V\cup P}, Q, q_0, \delta', \alpha)$ running on full $\mathcal{P}\times 2^{V\cup P}$-labeled Υ_A-trees that accepts a tree $\langle {\Upsilon_A}^*, \mathit{scheduler} \times c\rangle$ iff $\langle Y_{\mathit{scheduler}}, c\rangle$ is accepted by $\mathcal{A}$.*

Proof. $\mathcal{B}$ simply uses the first element of the label of each node in $\langle {\Upsilon_A}^*, scheduler \times c\rangle$ it traverses to determine which successors, according to the definition of $Y_{scheduler}$, really exist.

$\delta'(q';\pi,\sigma)$ can be constructed from $\delta(q';\sigma)$ by replacing, for all $q \in Q$,

- each occurrence of $(q,\Box)$ and $(q,\Diamond)$ by $(q,(\bot,\pi))$ if $p_{env} \notin \pi$,
- each occurrence of $(q,\Box)$ by $\bigwedge_{O\subseteq O_{env}}(q,(O,\pi))$ if $p_{env} \in \pi$, and
- each occurrence of $(q,\Diamond)$ by $\bigvee_{O\subseteq O_{env}}(q,(O,\pi))$ if $p_{env} \in \pi$. □

4.3 Quantification over all Schedulers

We are only interested in implementations that realize the specification for all schedulers.

For a $\Sigma\times\Xi$-labeled Υ-tree $\langle Y,l\rangle$, we denote the *Ξ-projection* $proj_\Xi : \langle Y,l\rangle \mapsto \langle Y,l_\Xi\rangle$ with $l(y) = (\sigma,\xi) \Rightarrow l_\Sigma : y \mapsto \xi$ that maps $\Sigma\times\Xi$-labeled Υ-trees to Ξ-labeled Υ-trees.

Lemma 7. *[2] Given an ε-free alternating automaton $\mathcal{B}$ running on $\Sigma\times\Xi$-labeled Υ-trees, we can construct an ε-free alternating automaton $\mathcal{C}$ that accepts a Ξ-labeled Υ tree $\langle\Upsilon^*,l\rangle$ iff $\mathcal{B}$ accepts all $\Sigma\times\Xi$-labeled Υ-trees $\langle\Upsilon^*,l'\rangle$ with $\langle\Upsilon^*,l\rangle = proj_\Xi(\langle\Upsilon^*,l'\rangle)$. The number of states of $\mathcal{C}$ is exponential in the number of states of $\mathcal{B}$, and the number of colors of $\mathcal{C}$ is quadratic in the number of states of $\mathcal{B}$.* □

4.4 Complexity

The construction provides 2EXPTIME and 3EXPTIME upper bounds in the length of a specification for the scheduler-independent synthesis problems of μ-calculus and CTL* specifications, respectively.

The 3EXPTIME hardness of scheduler-independent realizability checking for CTL* can be obtained by a reduction from the CTL* synthesis problem for synchronous systems in reactive environments [6].

Theorem 2. *The scheduler-independent realizability and synthesis problem is 2EXPTIME-complete for μ-calculus specifications and in 3EXPTIME-complete for specifications in CTL*.*

Proof. The upper bounds follow from the construction suggested in this section. To establish the lower bound for CTL* specifications, we transform a CTL* specification φ, which reasons only over the communication variables V, into a CTL* specification ψ_φ, which is scheduler-independent realizable iff φ is realizable in a synchronous setting with a reactive environment [6]. The environment is called reactive if it can disable a subset (but not all) of its responses in each turn. A full $2^{O_b\cup I_b}$-labeled 2^{I_b}-tree $\langle(2^{I_b})^*,l\rangle$ is a realization of φ in a reactive environment iff every total subtree of $\langle(2^{I_b})^*,l\rangle$ is a model of φ and the 2^{I_b}-projection of $\langle(2^{I_b})^*,l\rangle$ is a tree, where every node is labeled with its direction $(proj_{2^{I_b}}(\langle(2^{I_b})^*,l\rangle) = \langle(2^{I_b})^*, dir\rangle)$.

Our transformation puts three assumptions on the scheduler: First, we assume that the environment is always scheduled ($\alpha_1 = AG\, p_{env}$). Then, we assume that the process b is scheduled initially and, once it is not scheduled is never scheduled again ($\alpha_2 = b \wedge A\, b\, U\, G \neg b$). And last, we assume that if b is scheduled, then there is a path where b as always scheduled ($\alpha_3 = AG\,(b \rightarrow EG\, b)$).

For architectures without white-box processes and $O_{env} = I_b$, there is a natural bijection between total 2^{I_b}-trees and schedulers that fulfills these assumptions: We simply map a total 2^{I_b}-tree Y to the scheduler $\langle((2^{I_b} \cup \{\bot\}) \times 2^{\{b\}})^*, \mathit{scheduler}_Y\rangle$ that always schedules the environment and b is scheduled iff the 2^{I_b} projection of the input sequence is in Y:

$$b \in \mathit{scheduler}_Y(y) \Leftrightarrow y \in (2^{I_b} \times \{b\})^* \wedge \mathit{proj}_{2^{I_b}}(y) \in Y.$$

The restriction of the scheduler tree of $\mathit{scheduler}_Y$ to those nodes where b is scheduled results in a tree that is isomorphic to Y. Consequently, if we transform a CTL* specification φ to a specification φ' by replacing all quantifications over all paths/some path by quantifications over all paths/some path, where b is constantly scheduled, an implementation $\langle(2^{I_b})^*, s_b\rangle$ realizes φ for a given total tree Y in the synchronous setting iff it realizes φ' for the scheduler $\mathit{scheduler}_Y$. $\langle(2^{I_b})^*, s_b\rangle$ is therefore a realization of φ in a synchronous setting with a reactive environment iff it is a realization of φ for all schedulers which satisfy the assumptions α_1, α_2 and α_3. This is equivalent to realizing $\psi_\varphi = (\alpha_1 \wedge \alpha_2 \wedge \alpha_3) \rightarrow \varphi'$ for all schedulers.

Since φ' can be obtained from φ by replacing each occurrence of $A\pi$ and $E\pi$ in φ by $A(Gb \rightarrow \pi)$ and $E(Gb \wedge \pi)$, respectively, the length of ψ_φ is linear in the length of φ. The 3EXPTIME hardness of scheduler-independent realizability checking for CTL* specifications therefore follows from the 3EXPTIME hardness of realizability checking for CTL* specifications in a synchronous setting with a reactive environment [6]. The 2EXPTIME hardness for realizability checking for the μ-calculus is a direct implication. □

4.5 Synthesis with Explicit Assumptions on the Scheduler

We close the discussion of scheduler-independent synthesis with the remark that this type of synthesis can also be used to find implementations that satisfy a specification φ as long as the scheduler satisfies an explicitly stated *assumption* α: we simply weaken the specification to $\varphi' = \alpha \rightarrow \varphi$.

The assumption α might, for example, specifically specify a round-robin scheduler. The most common assumption on schedulers, however, is *fairness*: A scheduling is considered *impartial* towards a process p if p is scheduled infinitely often, *just* if p is infinitely often disabled or scheduled, and *compassionate* if p being enabled infinitely often implies that p is scheduled infinitely often. The enabledness $\mathit{enabled}(p)$ of a process $p \in P^-$ can be expressed using new hidden variables for the processes. Quantifying over all fair schedulers for a specification φ is equivalent to quantifying over all schedulers for a modified specification φ' that is satisfied both if φ is satisfied or if the scheduler is not fair. With the fairness condition expressed as a path formula (for example, justice is expressed by

$\pi_p = GF\neg enabled(p) \vee GFp)$, we obtain the fairness constraint $\pi = \bigwedge_{p \in P^-} \pi_p$. The modified specification φ' is the disjunction $\varphi' = \neg A\pi \rightarrow \varphi$.

Synthesis under full scheduling and scheduler-independent synthesis thus give us two different approaches to deal with fairness assumptions. While synthesis under full scheduling allows us to require that a property hold for all fair *schedules* (by replacing all occurrences of $A\psi$ and $E\psi$ in CTL* specifications by $A(\pi \rightarrow \psi)$ and $E(\pi \wedge \psi)$, respectively), scheduler-independent synthesis allows us to require that a property hold for all fair *schedulers*.

5 Multi-Process Synthesis

The algorithms from Sections 3 and 4 solve the synthesis problem for all architectures with a single black-box process. We now show that for all architectures with more than one black-box process, the synthesis problem is undecidable. Our synthesis algorithms thus cover all decidable asynchronous architectures.

The following theorem states the undecidability result for synthesis under full scheduling; the undecidability of scheduler-independent synthesis follows as a corollary.

Theorem 3. *The synthesis problem is undecidable for all architectures with at least two black-box processes and CTL or LTL specifications.*

Proof. We prove undecidability with a reduction from Post's Correspondence Problem (PCP). For a given alphabet A, an instance of PCP consists of an indexed set of pairs of words $(u_i, v_i), u_i, v_i \in A^+, i \in I = \{1, \ldots, n\}$, over an alphabet A. A solution of PCP is a sequence of indices $i_1, i_2, \ldots, i_m \in I^+$ such that $u_{i_1} \cdot u_{i_2} \cdot \ldots \cdot u_{i_m} = v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_m}$.

We consider architectures that have at least two different black-box processes p and q, where both p and q have at least one binary output or hidden variable. The basic idea of the reduction is to let process p compute the sequence of indices $i_1, i_2, \ldots, i_m$ and to let q produce the corresponding word $u_{i_1} \cdot u_{i_2} \cdot \ldots \cdot u_{i_m} = v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_m}$. To check that the word produced by q corresponds to the sequence produced by p, we consider two different schedulings, one in which p produces the indices along the u-words and one in which p produces the indices along the v-words.

We ensure that the two processes always see the constant input 0 along both paths and must therefore produce the same output on both paths. For the white-box processes we fix strategies that map any input history to 0. For all black-box processes except p and q (if any) and the environment, we specify that their output variables are globally set to 0. Let the formulas for this requirement, which are in both LTL and CTL, be denoted by γ_b and γ_{env}, respectively.

Each index produced by process p is preceded and followed by the constant 0, and terminated by the special symbol $\perp$: $0, i_1, 0,\ 0, i_2, 0,\ \ldots,\ 0, i_m, 0,\ \perp$. To each letter l produced by process q, we add a flag f_u indicating if this particular letter is the first letter of the u-word in the sequence, and

a flag f_v, if it is the first letter of the v-word. Each letter is again preceded and followed by the constant 0, and the sequence is terminated by $\bot$: $0, l_1, f_{u1}, f_{v1}, 0,\ 0, l_2, f_{u2}, f_{v2}, 0,\ \ldots,\ 0, l_k, f_{uk}, f_{vk}, 0,\ \bot$.

We assume that the encodings of the indices and letters with flags have equal length N. Each encoding of $0, i, 0$ and $0, l, w_i, w_j, 0$ starts with a sequence (say, 0111) that will occur nowhere else in any sequence encoding some sequence of indices or letters with flags, which allows us to identify where the output of an index or letter with flags starts.

LTL. We set $\varphi_u = \alpha_1 \rightarrow (\gamma_1 \wedge (\alpha_2 \rightarrow \gamma_2))$ for the following path assumptions α_1, α_2 and guarantees γ_1, γ_2:

- $\alpha_{\mathbf{1}}$: globally, the concurrent scheduling of p and q is succeeded by a sequence of $N-1$ times where only p is scheduled, which is succeeded by a finite sequence where q is not scheduled, which is succeeded by a further concurrent scheduling of p and q;
- $\gamma_{\mathbf{1}}$: globally, the concurrent scheduling of p and q initializes the output of an index $i \in I \cup \{\bot\}$;
- $\alpha_{\mathbf{2}}$: globally, an output sequence of an index $i \in I$ that is started by a concurrent scheduling of p and q is succeeded by $|u_i| \cdot N - 1$ (where $|u_i|$ denotes the length of the word u_i) positions in which only q is scheduled, which is succeeded by a concurrent scheduling of p and q; an output sequence of $\bot$ by p that is started by a concurrent scheduling of p and q is succeeded by $N-1$ positions in which only q is scheduled;
- $\gamma_{\mathbf{2}}$: the concurrent scheduling of p and q initialize the output of an index $i \in I$ followed by the output of u_i, until the concurrent scheduling of p and q initialize the emission of $\bot$ by p, followed by the emission of $\bot$ by q.

If φ_v is defined correspondingly and $\gamma_\bot$ denotes the guarantee that $\bot$ is not immediately emitted, then $\psi = \gamma_b \wedge \gamma_\bot \wedge p \wedge q \wedge \gamma_{env} \rightarrow (\varphi_u \wedge \varphi_v)$ is realizable iff the correspondence problem has a solution.

CTL. For $\varphi_u = E\varphi U \varphi_\bot$, where $\varphi_\bot$ denotes that p and q would start to emit $\bot$, φ is the conjunction of the following assertions:

- if p would (if continously scheduled alone) start to emit u_i, q would start to emit j and the output variables of p and q are set to 0 then $i = j$ and p and q are both scheduled concurrently;
- if p would not start to emit a word u_i and (q would start to output an index or the output variables of p are not set to 0) then only p is scheduled;
- if neither p nor q would start to emit a word or an index, respectively, or the output variables of q are not set to 0 then only q is scheduled;
- the output variables of the environment are all set to 0.

If φ_v is defined correspondingly and γ_0 denotes that the output variables of p and q are set to 0, then $\psi = \gamma_b \wedge \gamma_0 \wedge \varphi_u \wedge \varphi_v \wedge \neg\varphi_\bot$ is realizable iff the correspondence problem has a solution. □

The undecidability of scheduler-independent synthesis follows because for LTL, realizability under full scheduling and scheduler-independent realizability coincide: Since LTL is a trace language, $\langle {\Upsilon_A}^*, l\rangle \models \varphi$ implies $\langle Y_{scheduler}, l\rangle \models \varphi$ for every total tree $Y_{scheduler} \subseteq {\Upsilon_A}^*$ and every LTL specification φ.

The assumption $\alpha = AG \bigwedge_{P' \subseteq P} EX(\bigwedge_{p \in P'} p \wedge \bigwedge_{p \in P \smallsetminus P'} \neg p)$ of a full scheduler can be expressed in CTL, and realizability of a CTL specification φ under full scheduling coincides with the scheduler-independent realizability of $\alpha \rightarrow \varphi$.

Corollary 1. *The distributed scheduler-independent synthesis problem is undecidable for all architectures with at least two black-box processes and CTL or LTL specifications.* □

6 Conclusions

The first synthesis algorithms for synchronous and asynchronous systems were introduced almost simultaneously in the late 1980's for trace languages. In the synchronous paradigm, synthesis has received great attention ever since, while, in the asynchronous setting, results have been few and far between. In the introduction, we raised the question whether this is due to an inherent hardness of the problem.

The results of this paper show that the cost of synthesizing asynchronous systems depends on the treatment of the scheduler. Synthesizing asynchronous systems is computationally no more expensive than synthesizing synchronous systems when using the most commonly used semantics, which presumes a full scheduler. Asynchronous synthesis without assumptions on the scheduler, on the other hand, is exponentially harder.

The undecidability of the multi-process synthesis problem underlines that the synthesis of asynchronous systems is indeed more difficult than the synthesis of synchronous systems: while it is possible to solve the distributed synthesis problem for several synchronous architectures with multiple black-box processes (like pipelines and rings), distributed synthesis for asynchronous systems is only decidable if the architecture contains at most one black-box process.

However, we consider the solution of the distributed synthesis problem, even when restricted to only one black-box process, a significant step forward. Model checking (which can be seen as the special case of the distributed synthesis problem where all processes are white-box) has brought formal methods to industrial practice in the test and verification phase. Distributed synthesis allows the application of formal methods in the much earlier design phase. An incompletely implemented system defines an architecture with a single black-box process (representing the unfinished part of the system) in addition to the completed white-box processes. By checking the realizability of the specification for this architecture, we can recognize design errors as soon as they are introduced into the implementation.

References

1. A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. In *Proc. CAV*, pages 156–168. Springer-Verlag, June 1994.
2. B. Finkbeiner and S. Schewe. Semi-automatic distributed synthesis. In *Proc. ATVA*, pages 263–277. Springer-Verlag, October 2005.
3. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proc. LICS*, pages 321–330. IEEE Computer Society Press, June 2005.
4. M. Jurdziński. Small progress measures for solving parity games. In *Proc. STACS*, pages 290–301. Springer-Verlag, 2000.
5. D. Kozen and R. J. Parikh. A decision procedure for the propositional μ-calculus. In *Proc. Logic of Programs*, pages 313–325. Springer-Verlag, 1983.
6. O. Kupferman, P. Madhusudan, P. Thiagarajan, and M. Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc. 11th Int. Conf. on Concurrency Theory*, pages 92–107. Springer-Verlag, 2000.
7. O. Kupferman and M. Y. Vardi. Synthesis with incomplete informatio. In *Proc. ICTL*, pages 91–106, Manchester, July 1997.
8. O. Kupferman and M. Y. Vardi. Church's problem revisited. *The bulletin of Symbolic Logic*, 5(2):245–263, June 1999.
9. O. Kupferman and M. Y. Vardi. μ-calculus synthesis. In *Proc. MFCS*, pages 497–507. Springer-Verlag, 2000.
10. O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Proc. LICS*, pages 389–398. IEEE Computer Society Press, July 2001.
11. D. E. Muller and P. E. Schupp. Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theor. Comput. Sci.*, 141(1-2):69–107, 1995.
12. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Automata, Languages and Programming, 16th International Colloquium*, pages 652–671. Springer-Verlag, 1989.
13. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. FOCS*, pages 746–757. IEEE Computer Society Press, 1990.
14. M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. CAV*, pages 267–278. Springer-Verlag, July 1995.
15. T. Wilke. CTL$^+$ is exponentially more succinct than CTL. In *Proc. FSTTCS*, pages 110–121. Springer-Verlag, Dec. 1999.