Uniform Distributed Synthesis

Bernd Finkbeiner and Sven Schewe
Universität des Saarlandes

Abstract

We provide a uniform solution to the problem of synthesizing a finite-state distributed system. An instance of the synthesis problem consists of a system architecture and a temporal specification. The architecture is given as a directed graph, where the nodes represent processes (including the environment as a special process) that communicate synchronously through shared variables attached to the edges. The same variable may occur on multiple outgoing edges of a single node, allowing for the broadcast of data. A solution to the synthesis problem is a collection of finite-state programs for the processes in the architecture, such that the joint behavior of the programs satisfies the specification in an unrestricted environment. We define information forks, a comprehensive criterion that characterizes all architectures with an undecidable synthesis problem. The criterion is effective: for a given architecture with n processes and v variables, it can be determined in $O(n^2 \cdot v)$ time whether the synthesis problem is decidable. We give a uniform synthesis algorithm for all decidable cases. Our algorithm works for all ω-regular tree specification languages, including the μ-calculus. The undecidability proof, on the other hand, uses only LTL or, alternatively, CTL as the specification language. Our results therefore hold for the entire range of specification languages from LTL/CTL to the μ-calculus.

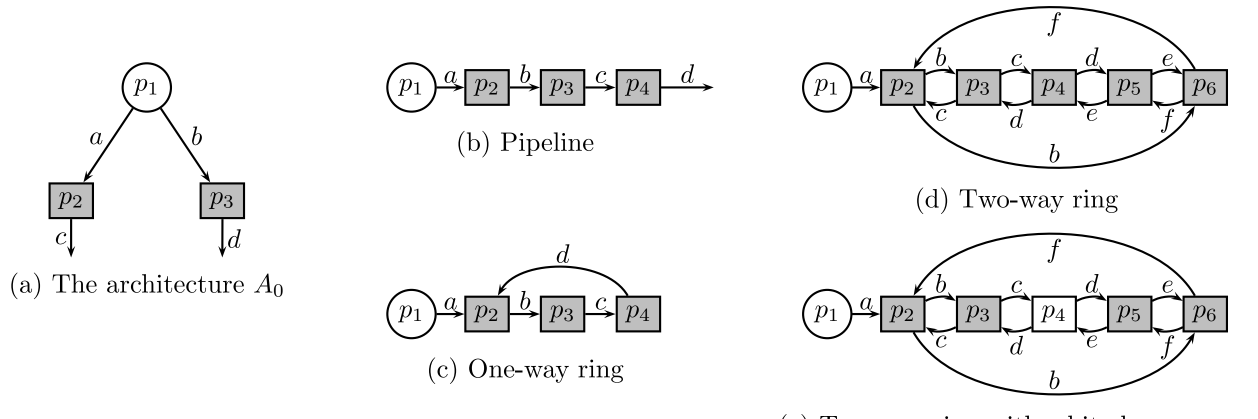
1 Introduction

Synthesis algorithms decide whether a given specification has an implementation. For distributed systems, the specification is usually given as a formula of a temporal logic and an implementation is a collection of finite-state programs that satisfy the formula when composed into the complete system.

The synthesis algorithms in the literature solve various instances of this problem that differ in the choice of the system architecture and the specification logic. *Closed synthesis*, the case of a single-process implementation without any interaction with the environment, was solved for CTL [1] and LTL [11]. *Open synthesis* concerns systems consisting of a single process and an environment and was solved for CTL* [4] as well as the μ-calculus [6]. An automata-based synthesis algorithm for pipeline and ring architectures and CTL* specifications is due to Kupferman and Vardi [7]; Walukiewicz and Mohalik provided an alternative game-based construction [10]. There is also a negative result: Pnueli and Rosner [9] showed that the synthesis problem is undecidable for LTL specifications and the simple architecture A_0, consisting of the environment and two independent system processes.

The question arises whether it is necessary to continue this series of isolated results, one for each architecture and logic. Can we provide a comprehensive criterion to determine if the distributed synthesis problem for a given system architecture and specification logic is decidable? Can the synthesis problem in fact be solved *uniformly*, that is, by a single algorithm for all decidable cases? In this paper, we give a positive answer to both questions.

In the *uniform distributed synthesis* problem, we decide for a given architecture A and a temporal specification φ over a set of boolean variables V whether there exists a finite-state program for each process in A, such that the composition of the programs satisfies φ. The architecture A is given as a directed graph, where the nodes represent processes, including the environment as a special process. The edges of the graph are labeled by variables from V, indicating that data may be transmitted between two processes. The same variable may occur on multiple outgoing edges of a single node, allowing for the broadcast of data. Among the set of system processes, we distinguish two types: a process is black-box if its implementation is unknown and needs to be discovered by the synthesis algorithm. A process is white-box if the implementation is already known and fixed. Figure 1 shows several example architectures, depicting the environment as a circle, black-box processes as filled rectangles, and white-box processes



(a) The architecture A_0

(b) Pipeline

(c) One-way ring

(d) Two-way ring

(e) Two-way ring with white-box process

Figure 1. Distributed architectures

as empty rectangles.

We provide a comprehensive criterion for the decidability of the synthesis problem in a given architecture: the problem is decidable if and only if the architecture does not contain an information fork. Intuitively, an information fork is a situation where two black-box processes receive information from the environment (directly or indirectly) in such a way that they cannot completely deduce the information received by the other process.

With the information fork criterion, it is very simple to determine for a given architecture whether the synthesis problem is decidable. Consider, for example, the 5-process two-way ring of Figure 1d. The synthesis problem is undecidable because of the information fork in the processes p_4 and p_5. The environment p_1 can transmit information through a, b, c to p_4 that remains unknown to p_5, and, vice versa, transmit information through a, b, f to p_5 that remains unknown to p_4. Interestingly, the architecture becomes decidable if we eliminate one of the two processes (resulting in a 4-process two-way ring) or, alternatively, fix one of their implementations, turning the process into a white-box, as shown for p_4 in Figure 1e.

The information fork criterion connects and extends the isolated decidability results in the literature. Pipelines and one-way rings, for example, have decidable synthesis problems [7] because the environment cannot communicate any information to a process without giving the same information to all processes to the left (when depicted as in Figure 1). By allowing for both broadcast and single-process communication, we distinguish the undecidable architecture A_0 in Figure 1a from the decidable architecture that can be obtained by adding variable a to the edge between processes p_1 and p_3 in architecture A_0. By identifying processes as black-box and white-box, we distinguish the decidable architecture in Figure 1e from the undecidable two-way ring in Figure 1d.

We solve the uniform synthesis problem with a single algorithm for all decidable cases. The algorithm consists of a first phase in which the architecture is transformed and a second phase wherein an automata-based construction solves the synthesis problem for the simplified architecture. First, the processes are ordered according to the information they possess about the environment's behavior. Groups of black-box processes with the same level of information can simulate each other, and are therefore collapsed into single processes. Then, we eliminate all white-box processes by replacing the indirect communication through a white-box process by direct edges between black-box processes. As the last simplification step, we eliminate feedback edges in the architecture, i.e., any backwards flow of information from processes with a lower level of information to those with a higher level. The feedback can be predicted by the better-informed process, making the edge in the architecture redundant.

The transformation steps turn any architecture without an information fork into an architecture that satisfies two conditions: the resulting architecture is acyclic and the order on the processes according to the level of information is strict. For this type of architecture, we solve the synthesis problem with an automata-based construction that successively eliminates processes along the information order, starting with the best-informed process.

2 Uniform Distributed Synthesis

In the *uniform distributed synthesis* problem, we 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 set of (possibly empty) sets of hidden variables for each process. We assume that the sets in H are pairwise disjoint from each other as well as from the sets in O. The same variable may occur on two separate edges to indicate broadcasting, but only if the edges originate in the same node.

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. For convenience, we use $O_{(p,p')} = \emptyset$ for $(p, p') \notin E$. An architecture A is called *acyclic* if the graph (P, E) is acyclic. A process p is called *idle* if $O_p = \emptyset$. The set $B = P \setminus W$ contains the black-box processes and the environment; $B^- = \{p \in B \setminus \{p_{env}\} \mid O_p \neq \emptyset\}$ is the set of non-idle black-box processes.

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 is a set of strategies $S = \{s_p | \, p \in B^-\}$ for all non-idle black-box processes.

Let $O_Q = \bigcup_{p \in Q} O_p$ denote the common output of a set $Q \subseteq P^-$ of processes and $I_Q = \bigcup_{p \in Q} I_p \setminus O_Q$ their common input. The *composition* $\bigotimes_{p \in Q} s_p = s_Q : (2^{I_Q})^* \to 2^{O_Q}$ of a set of strategies $\{s_p | p \in Q \subseteq P^-\}$ maps the common input history of the processes in Q to their common output: $s_Q : \varepsilon \mapsto \bigcup_{p \in Q} s_p(\varepsilon)$ and $s_Q : x \cdot v \mapsto \bigcup_{p \in Q} s_p(mem_p(x \cdot v))$, with $mem_p : (2^{I_Q})^* \to (2^{I_p})^*$, $mem_p : \varepsilon \mapsto \varepsilon$ and $mem_p : x \cdot v \mapsto mem_p(x) \cdot ((s_Q(x) \cup v) \cap I_p)$.

We use trees as a representation for strategies and computations. As usual, a (full) *tree* is given as the set Υ^* of all finite words over a given set of directions Υ. We define that every non-empty node $x \cdot \upsilon$, $x \in \Upsilon^*, \upsilon \in \Upsilon$, has the direction $dir(x \cdot \upsilon) = \upsilon$ and the empty word ε has some designated *root-direction* $dir(\varepsilon) = \upsilon_0 \in \Upsilon$. For given finite sets Σ and Υ, a Σ-*labeled* Υ-*tree* is a pair $\langle \Upsilon^*, l \rangle$ with a labeling function $l : \Upsilon^* \to \Sigma$ that maps every node of Υ^* to a letter of Σ. For a set $\Xi \times \Upsilon$ of directions and a node $x \in (\Xi \times \Upsilon)^*$, $hide_\Upsilon(x)$ denotes the node in Ξ^* obtained from x by replacing (ξ, υ) by ξ in each letter of x.

For a Σ-labeled $\Xi \times \Upsilon$-tree $\langle (\Xi \times \Upsilon)^*, l \rangle$, we define the function $xray_\Xi : \langle (\Xi \times \Upsilon)^*, l \rangle \mapsto \langle (\Xi \times \Upsilon)^*, l' \rangle$ with $l'(x) = (\mathrm{pr}_1(dir(x)), l(x))$ that maps Σ-labeled $\Xi \times \Upsilon$-trees to $\Xi \times \Sigma$-labeled $\Xi \times \Upsilon$-trees, adding the Ξ part of the direction of a node to its label.

For a Σ-labeled Ξ-tree $\langle \Xi^*, l \rangle$ we define the Υ-widening of $\langle \Xi^*, l \rangle$, denoted by $wide_\Upsilon(\langle \Xi^*, l \rangle)$, as the Σ-labeled $\Xi \times \Upsilon$-tree $\langle (\Xi \times \Upsilon)^*, l' \rangle$ with $l'(x) = l(hide_\Upsilon(x))$. In $wide_\Upsilon(\langle \Xi^*, l \rangle)$, nodes that are indistinguishable for someone who cannot observe Υ (i.e., nodes x, y with $hide_\Upsilon(x) = hide_\Upsilon(y)$) have the same label.

The specification φ refers to the computation tree, which maps the output history of the environment to the joint output of all processes. The *computation tree* of an implementation S is defined as the 2^V-labeled $2^{O_{env}}$-tree $\langle {2^{O_{env}}}^*, l \rangle = xray_{2^{O_{env}}}(wide_{2^{H_{env}}}(\langle (2^{O_{env} \setminus H_{env}})^*, \bigotimes_{p \in P^-} s_p \rangle))$. An implementation *solves* a triple $(A, \varphi, \{s_w | w \in W\})$ if its computation tree satisfies φ.

Synthesis. A triple $(A, \varphi, \{s_w | w \in W\})$ is *realizable* iff there exists an implementation that solves $(A, \varphi, \{s_w | w \in W\})$.

We call an architecture A *decidable* if there exists an algorithm 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 realizable.

3 Information Forks

As discussed in the introduction, an information fork is a situation where two black-box processes receive information from the environment (directly or indirectly) in such a way that they cannot completely deduce the information received by the other process. Formally, an *information fork* is a tuple (P', V', p, p'), where P' is a subset of the processes, V' is a subset of the variables disjoint from $I_p \cup I_{p'}$, and $p, p' \in B^- \setminus P'$ are two different black-box processes. Such a tuple is an information fork if P' together with the edges that are labeled with at least one variable from V' forms a subgraph rooted in the environment and there exist two nodes $q, q' \in P'$ that have edges to p, p', respectively, such that $O_{(q,p)} \nsubseteq I_{p'}$ and $O_{(q',p')} \nsubseteq I_p$.

For example, the architecture A_0 contains the information fork $(\{p_1\}, \emptyset, p_2, p_3)$. The 5-process two-way ring of Figure 1d contains the information fork (P', V', p, p') with $P' = \{p_1, p_2, p_3, p_6\}$, $V' = \{a, b\}, p = p_4, p' = p_5$.

We now show that the information fork criterion is effectively decidable. Our construction is based on the observation that every architecture that does not contain an information fork can be ordered according to the relative informedness of the processes.

Consider, for a black-box process p, the set $E_p = \{e \in E | O_e \nsubseteq I_p\}$ of edges that carry information invisible to p, and the set $U_p = \{q \in B|$ there is no directed path from p_{env} to q in $(P, E_p)\}$ of processes that are not reachable by such edges. The preorder $\preccurlyeq$ (read: has more or equal information than) is then defined as follows: for two black-box processes $p, p' \in B$, $p \preccurlyeq p' \Leftrightarrow p' \in U_p$.

An architecture A is called *ordered* by a surjective function $f : B \to \mathbb{N}_n$ for some $n \in \mathbb{N}$, if $\{p_{env}\}$ is the preimage of 1 and for all $p, p' \in B$: $f(p) \leqslant f(p')$ iff $p \preccurlyeq p'$. If f is bijective, A is called *strictly ordered* by f. An architecture is called (strictly) ordered if it is (strictly) ordered by some function f.

An architecture is called *idle-free* iff none of its black-box processes are idle. Having no output, idle processes have only the canonical strategy to output $\emptyset$ upon every input-history and can be pruned.

We define the *related idle-free architecture* $A' = \mathit{idlefree}(A)$ to an architecture A, as follows:

- $P' = W \cup B^- \cup \{p_{env}\}$, $W' = W$,
- $E' = E \cap P' \times P'$,
- $O'_e = O_e$ for all $e \in E'$ and
- $H'_p = O_p \setminus \bigcup_{p' \in P'} O_{(p,p')}$ for all $p \in P'$.

An architecture A is called *weakly ordered* iff $\mathit{idlefree}(A)$ is ordered.

Theorem 3.1 An architecture A is weakly ordered iff A does not contain an information fork.

Proof: Suppose A contains an information fork (P', V', p, p'), then $p \not\preccurlyeq p', p' \not\preccurlyeq p$. Hence, $\mathit{idlefree}(A)$ is not ordered. If $A' = \mathit{idlefree}(A)$ is ordered, then $\forall p, p' \in B^- : p \preccurlyeq p' \vee p' \preccurlyeq p$ and (P', V', p, p') is not an information fork. $\square$

Whether a given architecture A contains an information fork can therefore be checked as follows:

1. compute $\mathit{idlefree}(A)$;
2. compute $\preccurlyeq$,
3. check if for each two processes $p, p' \in B$, $p \preccurlyeq p'$ or $p' \preccurlyeq p$.

The algorithm runs in $O(n^2 \cdot v)$ time, where $n = |P|$ is number of processes and $v = |V|$ is the number of variables in the architecture A. As we show in Section 5, architectures that contain an information fork are undecidable. In the following section we show that architectures without information forks are decidable.

4 The Synthesis Algorithm

The synthesis algorithm consists of three phases: in the first phase, the architecture is transformed into a strictly ordered acyclic architecture without white-box processes. In the second phase, an automata-based construction decides whether the simplified architecture is realizable. If so, an implementation is computed in the third phase.

4.1 Architecture Transformations

We apply four transformations: elimination of idle processes, clustering of equally informed processes, elimination of white-box processes, and elimination of feedback edges.

Elimination of idle processes. An implementation $S = \{s_p\}$ solves $(\mathit{idlefree}(A), \varphi, \{s_w | w \in W\})$ iff it solves $(A, \varphi, \{s_w | w \in W\})$.

Clustering of equally informed processes. Black-box processes with the same level of information can simulate each other and are therefore collapsed into a single process. Let the architecture A be ordered by a function $f : B \to \mathbb{N}_n$ and let $g : P \to \mathbb{N}_n \cup W$ with $g \upharpoonleft_B = f$ and $g \upharpoonleft_W = \mathrm{id}_W$.[1] The *quotient architecture* $A' = A/\sim$ (where $\sim$ is the equivalence induced by $\preccurlyeq$) is defined as follows:

- $B' = \mathbb{N}_n$, $W' = W$,
- $E' = \bigcup_{(p,p') \in E} \{(g(p), g(p'))\} \setminus \bigcup_{i \in B'} \{(i, i)\}$,
- $O'_{(i,j)} = \bigcup_{p \in g^{-1}(i), p' \in g^{-1}(j)} O_{(p,p')}$ and
- $H'_i = \bigcup_{p \in g^{-1}(i)} O_p \setminus \bigcup_{j \in P'} O'_{(i,j)}$.

The quotient architecture is strictly ordered by id_n.

Lemma 4.1 Let the architecture A be ordered by a function f. Then $(A, \varphi, \{s_w | w \in W\})$ is realizable iff, for $A' = A/\sim$, $(A', \varphi, \{s_w | w \in W\})$ is realizable.

[1] $\upharpoonleft$ and $\upharpoonright$ denote input- and output-restrictions, respectively.

Proof: Let $W_i = \{w \in W|$ there is no directed path from p_{env} to w in $(P', E') \smallsetminus \{i\}\}$.

Let $S = \{s_p | p \in B^-\}$ solve $(A, \varphi, \{s_w | w \in W\})$ and let $\overline{s}_i = \bigotimes_{p \in W_i \cup f^{-1}(\mathbb{N}_n \smallsetminus \mathbb{N}_{i-1})} s_p$, then $S' = \{s'_i | p \in B'^-\}$ with $s'_i : x \mapsto \overline{s}_i(hide_{2^{I_i} \smallsetminus \overline{I}_i}(x)) \cap O_i$ solves $(A', \varphi, \{s_w | w \in W\})$.

Conversely, let the implementation $S' = \{s'_i | i \in B'^-\}$ solve $(A', \varphi, \{s_w | w \in W\})$ and let $\bigotimes_{i \in W_i \cup \mathbb{N}_n \smallsetminus \mathbb{N}_{i-1}} s_i = \overline{s}'_i : (2^{\overline{I}_i}) \to 2^{\overline{O}_i}$. Then $S = \{s_p | p \in B^-\}$ with $s_p : x \mapsto \overline{s}'_{f(p)}(hide_{2^{I_p} \smallsetminus \overline{I}_{f(p)}}(x)) \cap O_p$ solves $(A, \varphi, \{s_w | w \in W\})$. □

Elimination of white-box processes. We eliminate a white-box process by attaching it to a process in B that can simulate it. We choose the best-informed process p_0 that provides the white-box with input and add edges from p_0 to ensure that each black-box process still has the same input after the elimination of the white-box process. The strategy s_w of a white-box process w is turned into an equivalent specification φ_w and added to the original specification φ.

Let A be an architecture that is strictly ordered by id_n, let $\{s_w | w \in W\}$ be a given set of white-box strategies and let $\{\varphi_w | w \in W\}$ be equivalent specifications. The *black-box synthesis problem related to* $(A, \varphi, \{s_w | w \in W\})$ is the synthesis problem $(A', \widehat{\varphi}, \emptyset) = black(A, \varphi, \{s_w | w \in W\})$, defined by:

- $P' = B \ (= \mathbb{N}_n)$, $W' = \emptyset$,
- $O'_{(b,b')} = O_{(b,b')} \cup \bigcup_{w \in set(b)} O_{(w,b')}$,
- $E' = \{e \in P' \times P' | O'_e \neq \emptyset\}$,
- $H'_p = O_p \cup \bigcup_{w \in set(b)} O_{(w,b')} \smallsetminus \bigcup_{p' \in P} O'_{(p,p')}$, and
- $\widehat{\varphi} = \bigwedge_{w \in W \smallsetminus set(p_{env})} \varphi_w \wedge (\bigwedge_{w \in set(p_{env})} \varphi_w \to \varphi)$,

where $set : \mathbb{N}_n \to 2^W$, $set : p \mapsto \{w \in W | p = min\{n; \{b \in B|$ there is a directed path from b to w in (P, E) that does not pass any black-box process$\}\}\}$. The architecture A' is again strictly ordered.

Lemma 4.2 Let the architecture A be strictly ordered by id_n. Then $(A, \varphi, \{s_w | w \in W\})$ is realizable iff $black(A, \varphi, \{s_w | w \in W\})$ is realizable.

Proof: Let S be an implementation that solves $(A, \varphi, \{s_w | w \in W\})$. Each black-box process p can simulate the output of all black-box processes $p' \geqslant p$. Since the behavior of the white-box processes in $set(p)$ is determined by that output, p can simulate their output as well.

Conversely, if $S' = \{s'_p\}$ is an implementation solving $black(A, \varphi, \{s_w | w \in W\})$, the restriction of the strategies to the reduced output, $S = \{s_p | p \in B^-\}$ with $s_p = s'_p \upharpoonright_{2^{O_p}}$, solves $(A, \varphi, \{s_w | w \in W\})$. □

Elimination of feedback edges. Edges from processes with a lower level of information to those with a higher level are redundant, because the feedback can be simulated by the better-informed process.

Let A be an architecture that is strictly ordered by f with $W = \emptyset$. The *acyclic architecture related to* A, $A' = acycle(A, f)$, is defined as follows:

- $P' = P$,
- $E' = \{(p, p') \in E | f(p) < f(p')\}$,
- $O'_e = O_e$ and $H'_p = O_p \smallsetminus \bigcup_{p' \in P} O'_{(p,p')}$.

The architecture $acycle(A, f)$ is acyclic and strictly ordered by f.

Lemma 4.3 Let the architecture A be strictly ordered by id_n with $W = \emptyset$ and let $A' = acycle(A, \mathrm{id}_n)$. Then $(A, \varphi, \emptyset)$ is realizable iff $(A', \varphi, \emptyset)$ is realizable.

Proof: Let $S = \{s_i | i \in B^-\}$ solve $(A, \varphi, \emptyset)$, and let $\overline{s}_i = \bigotimes_{j \in \mathbb{N}_n \smallsetminus \mathbb{N}_{i-1}} s_j$, then $S' = \{s'_i | p \in B'^-\}$ with $s'_i : x \mapsto \overline{s}_i(x) \cap O'_i$ solves $(A', \varphi, \emptyset)$.

Conversely, if $S' = \{s'_i | i \in B'^-\}$ solves $(A', \varphi, \emptyset)$, $S = \{s'_i \circ hide_{2^{I_i} \smallsetminus I'_i} | i \in B^-\}$ solves $(A, \varphi, \emptyset)$. □

4.2 Realizability

As the result of the transformation steps we obtain a strictly ordered acyclic architecture without white-box processes. We solve the synthesis problem for the simplified architecture in an automata-based construction. Our construction builds on the algorithm for pipeline architectures by Kupferman and Vardi [7]. We consider synchronous behavior with delay. The adaptation to the case without delay is straightforward [7].

Automata. An alternating automaton $\mathcal{A} = (\Sigma, Q, q_0, \delta, \alpha)$ runs on 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 $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$, 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 $(q_0, l(\varepsilon))$ and where for a node n with a label (q, x) and a set of children $child(n)$, the labels of these children have the following properties:

- for all $m \in child(n)$: the label of m is $(q_m, x \cdot \upsilon_m)$, $q_m \in Q, \upsilon_m \in \Upsilon$ 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 of colors $C \subset \mathbb{N}$. A path is accepted if the highest color appearing infinitely often is even. A *Streett condition* is a set of pairs of subsets of Q, $(G_i, R_i)_{i\in I}$ for some finite index set I, called green and red states. A path is accepted iff for all pairs $(G_i, R_i)_{i\in I}$ an element of G_i or no element of R_i appears infinitely often.

A 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 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.

A nondeterministic automaton is a special alternating automaton, where the image of δ consists only of such formulae that, when rewritten in disjunctive normal form, contain exactly one element of $Q \times \{v\}$ in every disjunct. Note that for nondeterministic automata, every node of a run tree corresponds to a node in the input tree. For nondeterministic automata, δ can also be viewed as a mapping $\delta : Q \times \Sigma \to 2^{\Upsilon \to Q}$. For nondeterministic automata, emptiness can be checked with an *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*.

Our synthesis algorithm consists of a series of automata transformations. Before we discuss the construction in detail, we give an overview of the algorithm.

Overview. Let A be an acyclic architecture with $W = \emptyset$ and $P = \mathbb{N}_n$, strictly ordered by id_n. We define $\widehat{O}_i = \bigcup_{i\leq j\leq n} O_j$ for all $i \in P$ and $\widehat{I}_i = I_i \cup O_{i-1}$ for all $i \in P \smallsetminus \{p_{env}\}$.

The input to our algorithm is a μ-calculus specification φ. We translate φ into an equivalent alternating parity automaton $\mathcal{A}_\varphi$ and construct the following automata:

- the alternating parity automaton $\mathcal{A}_1 = \mathit{cover}_{2^{O_1}}(\mathcal{A}_\varphi)$, accepting the $2^{\widehat{O}_2}$-labeled 2^{O_1}-trees that solve the (not distributed and delay-free) synthesis problem for an enriched input;
- the alternating parity automaton $\mathcal{B}_1 = \mathit{wait}(\mathcal{A}_1)$, accepting the $2^{\widehat{O}_2}$-labeled $2^{\widehat{I}_2}$-trees that solve the (not distributed) synthesis problem for an enriched input;
- for $2 \leq i \leq n$:
  - the alternating parity automaton $\mathcal{A}_i = \mathit{narrow}_{2^{I_i}}(\mathcal{B}_{i-1})$, accepting the $2^{\widehat{O}_i}$-labeled 2^{I_i}-trees that solve the (not distributed) synthesis problem for the remaining processes $\{i, .., n\}$;
  - the nondeterministic parity automaton $\mathcal{N}_i = \mathit{ndet}(\mathcal{A}_i)$, equivalent to $\mathcal{A}_i$;
  - the alternating parity automaton $\mathcal{B}_i = \mathit{change}_{2^{O_i}, 2^{I_{i+1} \smallsetminus O_i}}(\mathcal{N}_i)$, accepting those $2^{\widehat{O}_{i+1}}$-labeled $2^{\widehat{I}_{i+1}}$-trees that solve the (not distributed) synthesis problem for the remaining processes $\{i+1, .., n\}$ for an enriched input.

$(A, \varphi, \emptyset)$ is realizable iff $\mathcal{N}_n$ is not empty.

Automata Constructions. The first step is to turn the specification into an equivalent alternating automaton. For μ-calculus specifications, the automaton has $O(n^2)$ states and $O(n)$ colors (n being the size of the specification).

Theorem 4.4 [6] Given a μ-calculus specification φ over a set V of atomic propositions and a finite set Υ, we can construct an alternating parity automaton $\mathcal{A}$ that accepts an 2^V-labeled Υ-tree iff it satisfies φ.

We are only interested in those trees where the label of every node is in accordance with its direction. The following automata transformation assures this and deletes the now redundant information from the labels. The state space of the resulting automaton is linear in the state space of the original automaton, while the set of colors remains unchanged.

Theorem 4.5 [5] Given an alternating parity automaton $\mathcal{A}$ over $\Upsilon \times \Sigma$-labeled Υ-trees, we can construct an alternating parity automaton $\mathcal{A}'$ over Σ-labeled Υ-trees, such that $\mathcal{A}'$ accepts $\langle \Upsilon^*, l\rangle$ iff $\mathcal{A}$ accepts $\mathit{xray}_\Upsilon(\langle \Upsilon^*, l\rangle)$. This automaton is denoted by $\mathit{cover}(\mathcal{A})$.

Since we consider communication with delay, the output of the processes must not depend on the last decision of the environment: i.e., all children of a node must be labeled equally. This is assured by the following transformation. The state space of the resulting automaton is linear in the state space of the original automaton, while the set of colors remains unchanged.

For a Σ-labeled Υ-tree $\langle\Upsilon^*, l\rangle$, we define the function *delay* : $\langle\Upsilon^*, l\rangle \mapsto \langle\Upsilon^*, l'\rangle$ with $l'(\varepsilon) = l(\varepsilon)$ and $l'(x\cdot\upsilon) = l(\upsilon_0 \cdot x)$ that maps Σ-labeled Υ-trees to Σ-labeled Υ-trees.

Theorem 4.6 [7] Given an alternating parity automaton $\mathcal{A}$ over Σ-labeled Υ-trees, we can construct an alternating parity automaton $\mathcal{A}'$ over Σ-labeled Υ-trees, such that $\mathcal{A}'$ accepts $\langle\Upsilon^*, l\rangle$ iff $\mathcal{A}$ accepts $delay(\langle\Upsilon^*, l\rangle)$. This automaton is denoted by $wait(\mathcal{A})$.

The following transformation ensures that the output of a process depends only on its input. The state-space and the set of colors remain unchanged.

Theorem 4.7 [5] Given an alternating parity automaton $\mathcal{A}$ over Σ-labeled $\Xi \times \Upsilon$-trees, we can construct an alternating parity automaton $\mathcal{A}'$ over Σ-labeled Ξ-trees, such that $\mathcal{A}'$ accepts $\langle\Xi^*, l\rangle$ iff $\mathcal{A}$ accepts $wide_\Upsilon(\langle\Xi^*, l\rangle)$. This automaton is denoted by $narrow_\Xi(\mathcal{A})$.

As the last transformation requires a nondeterministic automaton as input, we have to translate the alternating automaton into an equivalent nondeterministic automaton. This translation is done in two steps: in the first step, an alternating parity automaton $\mathcal{A}$ is turned into a nondeterministic Streett automaton $\mathcal{N}_S$ using a method by Muller and Schupp [8]. If $\mathcal{A}$ has n states and c colors, then $\mathcal{N}_S$ has $n^{O(c\cdot n)}$ states and $O(c\cdot n)$ pairs. In the second step, we turn the nondeterministic Streett automaton $\mathcal{N}_S$ into a nondeterministic parity automaton $\mathcal{N}$. If $\mathcal{N}_S$ has m states and p pairs, then $\mathcal{N}$ has $p^{O(p)} \cdot m$ states and $O(p)$ colors.

Note that in our construction this blow-up is consumed by the blow-up of the previous automata transformation; the resulting state-space is still of size $n^{O(c\cdot n)}$ and the resulting automaton has $O(c\cdot n)$ colors.

Theorem 4.8 [8] Given an alternating parity automaton $\mathcal{A}$, we can construct a nondeterministic Streett automaton $\mathcal{N}_S$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{N}_S)$.

Theorem 4.9 Given a nondeterministic Streett automaton $\mathcal{N}_S$, we can construct a nondeterministic parity automaton $\mathcal{N}$ with $\mathcal{L}(\mathcal{N}_S) = \mathcal{L}(\mathcal{N})$.

Construction: For $\mathcal{N}_S = (\Sigma, Q, q_0, \delta, (G_i, R_i)_{i\in I})$ running on Υ-trees, we define $\mathcal{N} = (\Sigma, Q', q_0', \delta', \alpha)$ in the following way:

- $Q' = Q \times perm(I) \times I \times I$, where $perm(I)$ denotes the set of permutations of the elements of I,
- $(q_\upsilon, \pi_\upsilon, r_\upsilon, g_\upsilon)_{\upsilon\in\Upsilon} \in \delta'((q, \pi, r, g), \sigma)$ iff
  - $(q_\upsilon)_{\upsilon\in\Upsilon} \in \delta(q, \sigma)$,
  - $\pi_\upsilon = (p_1, p_2, \ldots, p_k)$ is obtained from $\pi = (j_1, j_2, \ldots, j_k)$ by shifting all numbers j_l with $q_\upsilon \in G_{j_l}$ to the left,
  - r_υ is the greatest number such that $q \in R_{j_{r_\upsilon}}$ (or 0 if q is in no red set) and
  - g_υ is the greatest number such that $q \in G_{j_{g_\upsilon}}$ (or 0 if q is in no green set);
- $q_0' = (q_0, \pi_0, r_0, g_0)$ for some arbitrary (π_0, r_0, g_0),
- $\alpha : (q, \pi, r, g) \mapsto 2g$ iff $g \leqslant r$ and
- $\alpha : (q, \pi, r, g) \mapsto 2r - 1$ otherwise.

Proof: The construction uses the memoryless determinacy of Streett games with index appearance record [2, 8]. The states (q, π, r, g) consist of some state q of $\mathcal{N}_S$, the memory (index appearance record) π, and numbers r and g, memorizing the rightmost position of an index in the index appearance record of the *previous* state, such that $q \in R_{\pi(r)}$ and $q \in G_{\pi(g)}$, respectively. We call these numbers *maximal previous red (green) position.*

If both players play memoryless in the acceptance game, the game will end in a circle with the following properties: a subset of, say n, indices will be continually shifted to the left in the index appearance record, while all other indices will remain unchanged on the right. The game is won by player *accept* (green) iff none of the unchanged indices is ever red in the circle.

The highest value of the maximal previous green position in the circle is $g = n$, while the highest value of the maximal previous red position $r \leqslant n$ iff the game is won by green. Hence, the maximal color of the circle is odd ($2r_{max} - 1 > 2n$) if player *reject* (red) wins and even ($2g_{max} = 2n$) if player *accept* (green) wins. ☐

Corollary 4.10 Given an alternating parity automaton $\mathcal{A}$, we can construct an equivalent nondeterministic parity automaton $\mathcal{N}$. This automaton is denoted by $ndet(\mathcal{A})$.

The last transformation turns a nondeterministic automaton $\mathcal{N}$, accepting strategy trees of two processes p_1 and p_2 with perfect knowledge, into an alternating automaton $\mathcal{A}$, accepting strategy trees for p_2 with limited knowledge. Process p_2 is aware of a subset of the environment actions and the output of p_1, delayed by one turn.

For a Σ-labeled Υ-tree $\langle\Upsilon^*, l\rangle$, we define the *memoryful version* of $\langle\Upsilon^*, l\rangle$, denoted by $mem(\langle\Upsilon^*, l\rangle)$, as the Σ^+-labeled Υ-tree $\langle \Upsilon^*, l' \rangle$ with $l'(\varepsilon) = l(\varepsilon)$ and $\forall (x, \upsilon) \in \Upsilon^* \times \Upsilon : l'(x \cdot \upsilon) = l'(x) \cdot l(x \cdot \upsilon)$.

For a Σ-labeled $\Xi \times \Upsilon$-tree $\langle (\Xi \times \Upsilon)^*, l_\Sigma \rangle$ and a Ξ-labeled $\Upsilon \times \Theta$-tree $\langle (\Upsilon \times \Theta)^*, l_\Xi \rangle$ we define their *composition*, denoted by $\langle (\Xi \times \Upsilon)^*, l_\Sigma \rangle \oplus \langle (\Upsilon \times \Theta)^*, l_\Xi \rangle$, as a $\Xi \times \Sigma$-labeled $\Upsilon \times \Theta$-tree $\langle (\Upsilon \times \Theta)^*, l \rangle$, with the following properties for $\langle (\Upsilon \times \Theta)^*, l^d_\Xi \rangle = \mathit{delay}(\langle (\Upsilon \times \Theta)^*, l_\Xi \rangle)$, $\langle (\Upsilon \times \Theta)^*, l_{mem} \rangle = \mathit{mem}(\mathit{xray}_\Upsilon(\langle (\Upsilon \times \Theta)^*, l^d_\Xi \rangle))$ and for all $x \in (\Upsilon \times \Theta)^*, \upsilon \in \Upsilon \times \Theta$:

- $l(\varepsilon) = l^d_\Xi(\varepsilon) \cup l_\Sigma(\varepsilon)$
- $l(x \cdot \upsilon) = l^d_\Xi(x \cdot \upsilon) \cup l_\Sigma(l_{mem}(x))$

For a set $\mathcal{T}$ of $\Xi \times \Sigma$-labeled $\Upsilon \times \Theta$-trees we define $\mathit{shape}_{\Xi,\Upsilon}(\mathcal{T})$ as the set of Σ-labeled $\Xi \times \Upsilon$-trees $\langle (\Xi \times \Upsilon)^*, l_\Sigma \rangle$ for which there is an Ξ-labeled $\Upsilon \times \Theta$-tree $\langle (\Upsilon \times \Theta)^*, l_\Xi \rangle$ with $\langle (\Xi \times \Upsilon)^*, l_\Sigma \rangle \oplus \langle (\Upsilon \times \Theta)^*, l_\Xi \rangle \in \mathcal{T}$.

The automaton $\mathcal{A}$ accepts a strategy tree for p_2 iff there is a (not necessarily forgetful) strategy for p_1 such that the composition of the two strategies is accepted by $\mathcal{N}$. The key to achieve this is to guess such a strategy for p_1 nondeterministically. The state-space and the coloring function remain unchanged.

Theorem 4.11 Given a nondeterministic parity automaton $\mathcal{N}$ over $\Xi \times \Sigma$-labeled $\Upsilon \times \Theta$-trees, we can construct an alternating parity automaton $\mathcal{A}$ over Σ-labeled $\Xi \times \Upsilon$-trees, such that $\mathcal{L}(\mathcal{A}) = \mathit{shape}_{\Xi,\Upsilon}(\mathcal{L}(\mathcal{N}))$. This automaton is denoted by $\mathit{change}_{\Xi,\Upsilon}(\mathcal{N})$.

Proof: For $\mathcal{N} = (\Xi \times \Sigma, Q, q_0, \delta, \alpha)$ we set $\mathcal{A} = (\Sigma, Q, q_0, \delta', \alpha)$ with $\delta'(q, \sigma) \mapsto \bigvee_{\xi \in \Xi, f \in \delta(q,(\xi,\sigma))} \bigwedge_{\upsilon \in \Upsilon, \vartheta \in \Theta} (f(\upsilon, \theta), (\xi, \upsilon))$.

When $\mathcal{A}$ reads the root σ_0 of the input tree $\langle (\Xi \times \Upsilon)^*, l_\Sigma \rangle$, it guesses a $\xi_0 \in \Xi$, which is our guess for $l_\Xi(\varepsilon)$. We proceed in accordance with $\delta(q_0, (\xi_0, \sigma_0))$. By the definition of δ', each copy of $\mathcal{A}$ that is sent to a state q into a direction (υ, θ) is sent to state q into the direction (ξ, υ). In the acceptance game for the input tree, the environment now chooses a pair of a state q and a node x (i.e., it chooses a direction and the node is evaluated accordingly). $\square$

This construction is a generalization of Kupferman and Vardi's transformation [7], where only the special case $\Upsilon = \emptyset$ is considered. It makes use of the fact that in case of nondeterministic tree automata only one copy is sent in every direction.

4.3 Synthesis

The triple $(A, \varphi, \emptyset)$ is realizable iff $\mathcal{N}_n$ is not empty. The construction involves one transformation of an alternating parity automaton to a nondeterministic parity automaton for each $i \in \{2, \ldots, n\}$, and therefore takes $(n-1)$-exponential time.

Theorem 4.12 *The distributed synthesis problem for a weakly-ordered acyclic architecture A, where $\mathit{idlefree}(A)/\sim$ has n black-box processes, and a specification given as a μ-calculus formula can be solved in n-exponential time.*

Proof: The actual emptiness test or the synthesis of a strategy for process n can be done in time polynomial in the state-space and exponential in the number of colors. More precisely, if $\mathcal{N}_n$ has m states and c colors, a strategy (or the proof of emptiness) can be found in $m^{O(c)}$ time [3]. The overall complexity is not altered by this step.

If $(A, \varphi, \emptyset)$ is realizable, it is easy to deduce a partial function t_n, mapping the state-space of $\mathcal{N}_n$ to $2^{\widehat{O}_n}$, from a winning strategy for *accept* in the emptiness game of $\mathcal{N}_n$.

We obtain strategies for processes $2, \ldots, n-1$ from the combined strategy s for all processes in $\{2, \ldots, n\}$. For a process $i \in \{2, \ldots, n\}$, the resulting strategy is the projection of the combined strategy to its respective output: $s_i = s \restriction_{2^{O_i}}$ (which depends only on the input of process i).

To compute the combined strategy for the processes $i, \ldots, n$ from the combined strategy for the processes $i+1, \ldots, n$, one can simply take the partial function t_{i+1}, build a safety automaton from this function running on $2^{\widehat{O}_{i+1}}$-labeled $2^{I_{i+1}}$-trees, intersect it with $\mathcal{N}_i$, and solve the resulting parity game. While the number of colors is decreasing, the number of states is the number of states of $\mathcal{N}_i$ multiplied with the size of the domain of the partial function t_{i+1}. Hence, the overall size of the state-space is quasi linear in the state-space of $\mathcal{N}_n$ and the overall complexity remains $(n-1)$-exponential.

A solution for the original problem is obtained from the strategies in the simplified problem by simply applying output restrictions, as shown in the proofs in Section 4. $\square$

5 Undecidable Architectures

The algorithm from Section 4 solves the synthesis problem for all architectures without information forks. In this section we show that the occurrence of an information fork is a sufficient condition for the undecidability of an architecture and hence establish the completeness of our approach.

Rosner and Pnueli [9] showed undecidability for the architecture A_0 and LTL using a reduction from the halting problem. In the proof of Lemma 5.1 we give a new reduction that applies to both CTL and LTL. Theorem 5.3 further extends the result to all architectures that contain an information fork.

Lemma 5.1 The distributed synthesis problem for CTL and LTL specifications is undecidable for the architecture A_0.

Proof: For a given deterministic Turing machine M, we define a specification φ_M that is realizable iff M halts on the empty input tape.

In the architecture A_0 (see Figure 1a), the environment p_1 communicates independently with two system processes p_2 and p_3 through their input variables a and b, respectively. In a first step, we define a specification ψ_M that has exactly one (not necessarily finite-state) implementation in which the environment p_1 can prompt processes p_2 and p_3 to output the entire computation of M (i.e., a series of successive configurations) on the hidden variables c and d, respectively, by sending a *start* command through the input variables a and b, respectively. Further *start* commands have no effect.

A configuration C is output as follows: it starts with the (possibly empty) sequence of tape symbols left of the read/write head, followed by first the internal state of M, and then the sequence of tape symbols from the position of the read/write head up to the first *blank*-sign.

Let $\perp$ denote the terminal state of the Turing machine and let $C \vdash C'$ denote that C' is the configuration succeeding C.

The specification $\psi_M = \psi_{p_2} \wedge \psi_{p_3}$ is constructed as the conjunction of the assertions ψ_{p_2} and ψ_{p_3}, where ψ_{p_2} is defined as follows:

- Initially, p_2 outputs $\perp$ symbols, until the first *start* symbol is received. Then, p_2 outputs the initial configuration of M and the second configuration of M, followed by a sequence of legal configurations of M.
- If p_2 and p_3 output C and C' respectively (starting concurrently) and $C \vdash C'$ holds, then the configurations C_{new} and C'_{new}, output next by p_2 and p_3, respectively, have to satisfy $C_{new} \vdash C'_{new}$.
  Note that their output starts concurrently iff the head was not at the end of the tape (i.e. over the *blank*) in C', and C'_{new} is output with a delay of exactly one symbol otherwise.

ψ_{p_3} is the corresponding assertion, where the roles of p_2 and p_3 are swapped.

A simple inductive argument shows that ψ_M has only the canonical implemention, where both processes output the computation of M:

Assume there is an implementation, where both processes always output the first i configurations following the canonical implementation, but one process (w.l.o.g. p_2) fails to output the $i+1^{th}$ configuration. If p_1 sends the *start* command on b exactly n steps after sending the *start* command on a, where n is the number of steps needed to output the $i-1^{th}$ configuration, then p_3 writes the $i-1^{th}$ configuration at the same time as p_2 outputs the i^{th} configuration. Hence, p_2 is forced to output the $i+1^{th}$ configuration correctly as well.

Consequently, the specification φ_M, ensuring that

- ψ_M holds and
- p_2 and p_3 always eventually output $\perp$,

has a (finite state) implementation iff M halts on the empty input tape. The specification ψ_M can easily be expressed in both CTL and LTL. □

The argument that the canonical implementation is the only possible implementation relies on the fact that p_2 is oblivious of b and p_3 is oblivious of a. In the architecture A_0, a and b are hidden because the environment communicates with both processes separately and neither process is aware of the other's output. We generalize the argument to all architectures with an information fork by first showing that the architecture A_0 remains undecidable if the output becomes visible and, in a second step, by allowing for indirect communication between the environment and the two processes.

Lemma 5.2 The distributed synthesis problem for CTL and LTL specifications is undecidable for the architecture with $P = \{p_{env}, p, p'\}$, $E = \{(p_{env}, p), (p_{env}, p'), (p, p'), (p', p)\}$, $V = \{i, i', h, h'\}$, $O_{p_{env}} = \{i, i'\}$, $I_p = \{i, j\}$, $I_q = \{i', j'\}$, $O_p = \{j\}$, $O_{p'} = \{j'\}$ (architecture A_0 plus communication between p_2 and p_3).

Proof: We introduce a perfect encryption function for each process output (for example XOR) and enlarge the input by the key. In this setting, we can state the specification as in the proof of Lemma 5.1, with the difference that the decrypted version of the output has to fulfill the output-requirements. The processes are oblivious of its decrypted meaning, even though they may read each other's encrypted output. □

For the undecidability of an architecture it already suffices if the environment can pass separate information to two different processes. This extends the class of undecidable architectures further to those containing an information fork.

Theorem 5.3 The distributed synthesis problem for LTL and for CTL specifications is undecidable for all architectures that contain an information fork.

Proof: If (P', V', p, p') is an information fork, it is possible to specify that some output pair (a, b) of the environment is communicated through V' to all processes in P', but only output a is communicated to process p (through $O_{(q,p)} \setminus I_{p'}$) and only output b is communicated to process p' (through $O_{(q',p')} \setminus I_p$, $q, q' \in P'$ as in the definition of information forks). Undecidability therefore follows as in Lemma 5.2. $\square$

Corollary 5.4 The algorithm from Section 4 solves the synthesis problem for all decidable architectures.

6 Conclusions

The invention of model checking in the 1980s has brought formal methods to industrial practice. Hardware and many communication protocols can be modeled as finite-state automata and their automatic analysis makes formal verification economically feasible. A major drawback of model checking methods is that they require the *complete* design to be known before they can be applied. It is, however, crucial to find design errors early, before much effort has gone into the implementation.

Our results make *incomplete* designs accessible to automated analysis. As soon as enough components have been implemented to make the architecture decidable, we can automatically complete the design by deriving an implementation for the remaining processes. If synthesis fails, the unrealizability of the specification demonstrates an error in the existing partial design.

Will it be possible to completely automatize the construction of distributed systems? The results of this paper mark the limits of system synthesis, because our algorithm is already applicable to all decidable architectures. Automated program construction is still likely to work in many practical applications. An example is the system maintenance phase, which dominates the life-time cost of most systems today. Since in every maintenance cycle only a few components are modified, nearly all components remain white-box and the architecture is likely to be decidable.

Semi-algorithms for undecidable architectures are a promising area of future research: if a finite-state solution exists, it can be found by a simple enumeration of the process strategies. Our results show that it is not necessary to enumerate the strategies of all processes. Since enumerating the strategies of a black-box process turns that process white-box, it suffices to consider a sufficient subset of the processes, such that all information forks are eliminated from the architecture.

Acknowledgments. We thank Andreas Podelski for comments on an early draft of this paper. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

References

[1] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[2] Y. Gurevich and L. Harrington. Automata, trees and games. In *Proc. STOC'82*, pages 60–65, 1982.

[3] M. Jurdziński. Small progress measures for solving parity games. In *Proc. STACS'00*, volume 1770 of *LNCS*, pages 290–301. Springer-Verlag, 2000.

[4] O. Kupferman and M. Y. Vardi. Synthesis with incomplete information. In *Proc. ICTL'97*, 1997.

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

[6] O. Kupferman and M. Y. Vardi. μ-calculus synthesis. In *Proc. MFCS'00*, volume 1893 of *LNCS*, pages 497–507. Springer-Verlag, 2000.

[7] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Proc. LICS'01*, July 2001.

[8] D. E. Muller and P. Schupp. Alternating automata on infinite objects, determinancy and Rabin's theorem. In *Automata on Infinite Words*, volume 192 of *LNCS*, pages 100–107. Springer-Verlag, 1985.

[9] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. FOCS'90*, pages 746–757, 1990.

[10] I. Walukiewicz and S. Mohalik. Distributed games. In *Proc. FSTTCS'03*, volume 2914 of *LNCS*, pages 338–351, 2003.

[11] P. Wolper. *Synthesis of Communicating Processes from Temporal-Logic Specifications.* PhD thesis, Stanford University, 1982.