Asynchronous Games over Tree Architectures

Blaise Genest[1], Hugo Gimbert[2], Anca Muscholl[2], Igor Walukiewicz[2]

[1] IRISA, CNRS, Rennes, France
[2] LaBRI, CNRS/Université Bordeaux, France

Abstract. We consider the distributed control problem in the setting of Zielonka asynchronous automata. Such automata are compositions of finite processes communicating via shared actions and evolving asynchronously. Most importantly, processes participating in a shared action can exchange complete information about their causal past. This gives more power to controllers, and avoids simple pathological undecidable cases as in the setting of Pnueli and Rosner. We show the decidability of the control problem for Zielonka automata over acyclic communication architectures. We provide also a matching lower bound, which is l-fold exponential, l being the height of the architecture tree.

1 Introduction

Synthesis is by now well understood in the case of sequential systems. It is useful for constructing small, yet safe, critical modules. Initially, the *synthesis problem* was stated by Church, who asked for an algorithm to construct devices transforming sequences of input bits into sequences of output bits in a way required by a specification [2]. Later Ramadge and Wonham proposed the *supervisory control* formulation, where a plant and a specification are given, and a controller should be designed such that its product with the plant satisfies the specification [18]. So control means restricting the behavior of the plant. Synthesis is the particular case of control where the plant allows for every possible behavior.

For synthesis of *distributed* systems, a common belief is that the problem is in general undecidable, referring to work by Pnueli and Rosner [17]. They extended Church's formulation to an architecture of *synchronously* communicating processes, that exchange messages through one slot communication channels. Undecidability in this setting comes mainly from *partial information*: specifications permit to control the flow of information about the global state of the system. The only decidable type of architectures is that of pipelines.

The setting we consider here is based on a by now well-established model of distributed computation using shared actions: *Zielonka's asynchronous automata* [20]. Such a device is an asynchronous product of finite-state processes synchronizing on common actions. Asynchronicity means that processes can progress at different speed. Similarly to [6,12] we consider the control problem for such automata. Given a Zielonka automaton (plant), find another Zielonka automaton (controller) such that the product of the two satisfies a given specification. In particular, the controller does not restrict the parallelism of the

system. Moreover, during synchronization the individual processes of the controller can exchange all their information about the global state of the system. This gives more power to the controller than in the Pnueli and Rosner model, thus avoiding simple pathological scenarios leading to undecidability. It is still open whether the control problem for Zielonka automata is decidable.

In this paper we prove decidability of the control problem for reachability objectives on tree architectures. In such architectures every process can communicate with its parent, its children, and with the environment. If a controller exists, our algorithm yields a controller that is a finite state Zielonka automaton exchanging information of *bounded* size. We also provide the first non-trivial lower bound for asynchronous distributed control. It matches the l-fold exponential complexity of our algorithm (l being the height of the architecture tree).

As an example, our decidability result covers client-server architectures where a server communicates with clients, and server and clients have their own interactions with the environment (cf. Figure 1). Our algorithm providing a controller for this architecture runs in exponential time. Moreover, each controller adds polynomially many bits to the state space of the process. Note also that this architecture is undecidable for [17] (each process has inputs), and is neither covered by [6] (the action alphabet is not a co-graph), nor by [12] (there is no bound on the number of actions performed concurrently).

Related work. The setting proposed by Pnueli and Rosner [17] has been thoroughly investigated in past years. By now we understand that, suitably using the interplay between specifications and an architecture, one can get undecidability results for most architectures rather easily. While specifications leading to undecidability are very artificial, no elegant solution to eliminate them exists at present.

The paper [10] gives an automata-theoretic approach to solving pipeline architectures and at the same time extends the decidability results to CTL* specifications and variations of the pipeline architecture, like one-way ring architectures. The synthesis setting is investigated in [11] for local specifications, meaning that each process has its own, linear-time specification. For such specifications, it is shown that an architecture has a decidable synthesis problem if and only if it is a sub-architecture of a pipeline with inputs at both endpoints. The paper [5] proposes information forks as an uniform notion explaining the (un)decidability results in distributed synthesis. In [15] the authors consider distributed synthesis for knowledge-based specifications. The paper [7] studies an interesting case of external specifications and well-connected architectures.

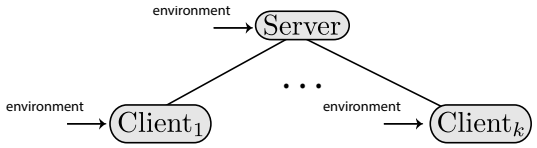


Fig. 1. Server/client architecture

Synthesis for asynchronous systems has been strongly advocated by Pnueli and Rosner in [16]. Their notion of asynchronicity is not exactly the same as ours: it means roughly that system/environment interaction is not turn-based, and processes observe the system only when scheduled. This notion of asynchronicity appears in several subsequent works, such as [19,9] for distributed synthesis.

As mentioned above, we do not know whether the control problem in our setting is decidable in general. Two related decidability results are known, both of different flavor than ours. The first one [6] restricts the alphabet of actions: control with reachability condition is decidable for co-graph alphabets. This restriction excludes among others client-server architectures. The second result [12] shows decidability by restricting the plant: roughly speaking, the restriction says that every process can have only bounded missing knowledge about the other processes (unless they diverge). The proof of [12] goes beyond the controller synthesis problem, by coding it into monadic second-order theory of event structures and showing that this theory is decidable when the criterion on the plant holds. Unfortunately, very simple plants have a decidable control problem but undecidable MSO-theory of the associated event structure. Melliès [14] relates game semantics and asynchronous games, played on event structures. More recent work [3] considers finite games on event structures and shows a determinacy result for such games under some restrictions.

Organization of the paper. The next section presents basic definitions. The two consecutive sections present the algorithm and the matching lower bound. The full version of the paper is available at `http://hal.archives-ouvertes.fr/hal-00684223`.

2 Basic definitions and observations

We start by introducing Zielonka automata and state the control problem for such automata. We also give a game-based formulation of the problem.

2.1 Zielonka automata

Zielonka automata are simple parallel finite-state devices. Such an automaton is a parallel composition of several finite automata, called *processes*, synchronizing on shared actions. There is no global clock, so between two synchronizations, two processes can do a different number of actions. Because of this Zielonka automata are also called asynchronous automata.

A *distributed action alphabet* on a finite set $\mathbb{P}$ of processes is a pair (Σ, dom), where Σ is a finite set of *actions* and $dom : \Sigma \to (2^{\mathbb{P}} \setminus \emptyset)$ is a *location function*. The location $dom(a)$ of action $a \in \Sigma$ comprises all processes that need to synchronize in order to perform this action. A (deterministic) *Zielonka automaton* $\mathcal{A} = \langle \{S_p\}_{p\in\mathbb{P}}, s_{in}, \{\delta_a\}_{a\in\Sigma}\rangle$ is given by:

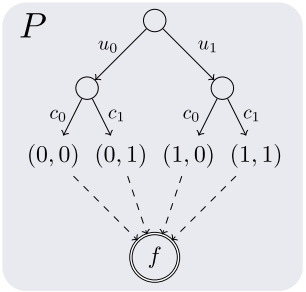
- for every process p a finite set S_p of (local) states,
- the initial state $s_{in} \in \prod_{p\in\mathbb{P}} S_p$,

– for every action $a \in \Sigma$ a partial transition function $\delta_a : \prod_{p \in dom(a)} S_p \stackrel{\cdot}{\to} \prod_{p \in dom(a)} S_p$ on tuples of states of processes in $dom(a)$.

For convenience, we abbreviate a tuple $(s_p)_{p \in P}$ of local states by s_P, where $P \subseteq \mathbb{P}$. We also talk about S_p as the set of *p-states* and of $\prod_{p \in \mathbb{P}} S_p$ as *global states*. Actions from $\Sigma_p = \{a \in \Sigma \mid p \in dom(a)\}$ are called *p-actions*. For $p, q \in \mathbb{P}$, let $\Sigma_{p,q} = \{a \in \Sigma \mid dom(a) = \{p, q\}\}$ be the set of synchronization actions between p and q. We write Σ_p^{loc} instead of $\Sigma_{p,p}$ for the set of *local* actions of p, and $\Sigma_p^{com} = \Sigma_p \setminus \Sigma_p^{loc}$ for the *synchronization actions* of p.

A Zielonka automaton can be seen as a sequential automaton with the state set $S = \prod_{p \in \mathbb{P}} S_p$ and transitions $s \xrightarrow{a} s'$ if $(s_{dom(a)}, s'_{dom(a)}) \in \delta_a$, and $s_{\mathbb{P} \setminus dom(a)} = s'_{\mathbb{P} \setminus dom(a)}$. By $L(\mathcal{A})$ we denote the set of words labeling runs of this sequential automaton that start from the initial state. Notice that $L(\mathcal{A})$ is closed under the congruence $\sim$ generated by $\{ab = ba \mid dom(a) \cap dom(b) = \emptyset\}$, in other words, it is a *trace-closed* language. A (Mazurkiewicz) trace is an equivalence class $[w]_\sim$ for some $w \in \Sigma^*$. The notion of *trace* and the idea of describing concurrency by a fixed independence relation on actions goes back to the late seventies, to Mazurkiewicz [13] (see also [4]).

Consider a Zielonka automaton $\mathcal{A}$ with two processes: $P, \overline{P}$. The local actions of process P are $\{u_0, u_1, c_0, c_1\}$ and those of process $\overline{P}$ are $\{\overline{u_0}, \overline{u_1}, \overline{c_0}, \overline{c_1}\}$. In addition, there is a shared action \$ with $dom(\$) = \{P, \overline{P}\}$. From the initial state, P can reach state (i, j) by executing $u_i c_j$, same for $\overline{P}$ with $\overline{u_k}\,\overline{c_l}$ and state $\overline{(k, l)}$. Action \$ is enabled in every pair $((i, j), \overline{(k, l)})$ satisfying $i = l$ or $k = j$, it leads to the final state. So $L(\mathcal{A}) = \{[u_i c_j \overline{u_k}\,\overline{c_l}\$] \mid i = l \text{ or } k = j\}$.



As the notion of a trace can be formulated without a reference to an accepting device, one can ask if the model of Zielonka automata is powerful enough. Zielonka's theorem says that this is indeed the case, hence these automata are a right model for the simple view of concurrency captured by Mazurkiewicz traces.

Theorem 1. *[20] Let $dom : \Sigma \to (2^{\mathcal{P}} \setminus \{\emptyset\})$ be a distribution of letters. If a language $L \subseteq \Sigma^*$ is regular and trace-closed then there is a deterministic Zielonka automaton accepting L (of size exponential in the number of processes and polynomial in the size of the minimal automaton for L, see [8]).*

2.2 The control problem

In Ramadge and Wonham's control setting [18] we are given an alphabet Σ of actions partitioned into system and environment actions: $\Sigma^{sys} \cup \Sigma^{env} = \Sigma$. Given a plant P we are asked to find a controller C over Σ such that the product $P \times C$ satisfies a given specification (the product being the standard product of the two automata). Both the plant and the controller are finite, deterministic automata over the same alphabet Σ. Additionally, the controller is required not

to block environment actions, which in technical terms means that from every state of the controller there should be a transition on every action from Σ^{env}.

The definition of our problem is the same with the difference that we take Zielonka automata instead of finite automata. Given a distributed alphabet (Σ, dom) as above, and a Zielonka automaton P, find a Zielonka automaton C over the same distributed alphabet such that $P \times C$ satisfies a given specification. Additionally it is required that from every state of C there is a transition for every action from Σ^{env}. The important point here is that the controller has the same distributed structure as the plant. Hence concurrency in the controlled system is the same as in the plant. Observe that in the controlled system $P \times C$ the states carry the additional information computed by the controller.

Example: Reconsider the automaton on page 4, and assume that $u_i, \overline{u_k} \in \Sigma^{env}$ are the uncontrollable actions $(i, k \in \{0, 1\})$. So the controller needs to propose controllable actions c_j and $\overline{c_k}$, resp., in such a way that both P and $\overline{P}$ reach their final states $f, \overline{f}$ by executing the shared action \$. At first sight this may seem impossible to guarantee, as it looks like process P needs to know what $\overline{u_k}$ process $\overline{P}$ has received, or vice-versa. Nevertheless, such a controller exists. It consists of P allowing after u_i only action c_i, and $\overline{P}$ allowing after $\overline{u_k}$ only action $\overline{c_{1-k}}$. Regardless if the environment chooses $i = j$ or $i \neq j$, the action \$ is enabled in state $((i, i), (j, 1 - j))$, so both $P, \overline{P}$ can reach their final states.

It will be more convenient to work with a game formulation of this problem, as in [6,12]. Instead of talking about controllers we will talk about distributed strategies in a game between *system* and *environment*. A plant defines a game arena, with plays corresponding to initial runs of $\mathcal{A}$. Since $\mathcal{A}$ is deterministic, we can view a play as a word from $L(\mathcal{A})$ – or a trace, since $L(\mathcal{A})$ is trace-closed. Let $Plays(\mathcal{A})$ denote the set of traces associated with words from $L(\mathcal{A})$.

A strategy for the system will be a collection of individual strategies for each process. The important notion here is the view each process has about the global state of the system. Intuitively this is the part of the current play that the process could see or learn about from other processes during a communication with them. Formally, the *p-view* of a play u, denoted $view_p(u)$, is the smallest trace $[v]$ such that $u \sim vy$ and y contains no action from Σ_p. We write $Plays_p(\mathcal{A})$ for the set of plays that are p-views: $Plays_p(\mathcal{A}) = \{view_p(u) \mid u \in Plays(\mathcal{A})\}$.

A *strategy for a process* p is a function $\sigma_p : Plays_p(\mathcal{A}) \to 2^{\Sigma_p^{sys}}$, where $\Sigma_p^{sys} = \{a \in \Sigma^{sys} \mid p \in dom(a)\}$. We require in addition, for every $u \in Plays_p(\mathcal{A})$, that $\sigma_p(u)$ is a subset of the actions that are possible in the p-state reached on u. A *strategy* is a family of strategies $\{\sigma_p\}_{p \in \mathbb{P}}$, one for each process.

The set of plays respecting a strategy $\sigma = \{\sigma_p\}_{p \in \mathbb{P}}$, denoted $Plays(\mathcal{A}, \sigma)$, is the smallest set containing the empty play ε, and such that for every $u \in Plays(\mathcal{A}, \sigma)$:

1. if $a \in \Sigma^{env}$ and $ua \in Plays(\mathcal{A})$ then ua is in $Plays(\mathcal{A}, \sigma)$;
2. if $a \in \Sigma^{sys}$ and $ua \in Plays(\mathcal{A})$ then $ua \in Plays(\mathcal{A}, \sigma)$ provided that $a \in \sigma_p(view_p(u))$ *for all* $p \in dom(a)$.

Plays from $Plays(\mathcal{A}, \sigma)$ are called σ-plays and we write $Plays_p(\mathcal{A}, \sigma)$ for the set $Plays(\mathcal{A}, \sigma) \cap Plays_p(\mathcal{A})$. The above definition says that actions of the environ-

ment are always possible, whereas actions of the system are possible only if they are allowed by the strategies of all involved processes.

Our winning conditions in this paper are *local reachability* conditions: every process has a set of target states $F_p \subseteq S_p$. We also assume that states in F_p are *blocking*, that is, they have no outgoing transitions. This means that if $(s_{dom(a)}, s'_{dom(a)}) \in \delta_a$ then $s_p \notin F_p$ for all $p \in dom(a)$. For defining winning strategies, we need to consider also infinite σ-plays. By $Plays^{\infty}(\mathcal{A}, \sigma)$ we denote the set of finite or infinite σ-plays in $\mathcal{A}$. Such plays are defined as finite ones, replacing u in the definition of $Plays(\mathcal{A}, \sigma)$ by a possibly infinite, initial run of $\mathcal{A}$. A play $u \in Plays^{\infty}(\mathcal{A}, \sigma)$ is *maximal*, if there is no action c such that the trace uc is a σ-play (note that uc is defined only if no process in $dom(c)$ is scheduled infinitely often in u).

Definition 1. *The* control problem *for a plant $\mathcal{A}$ and a local reachability condition $(F_p)_{p\in\mathbb{P}}$ is to determine if there is a strategy $\sigma = (\sigma_p)_{p\in\mathbb{P}}$ such that every maximal trace $u \in Plays^{\infty}(\mathcal{A}, \sigma)$ is finite and ends in $\prod_{p\in\mathbb{P}} F_p$. Such traces and strategies are called* winning.

3 The upper bound for acyclic communication graphs

We impose two simplifying assumptions on the distributed alphabet (Σ, dom). The first one is that all actions are at most binary: $|dom(a)| \leq 2$, for every $a \in \Sigma$. The second requires that all uncontrollable actions are local: $|dom(a)| = 1$, for every $a \in \Sigma^{env}$. The first restriction makes the technical reasoning much simpler. The second restriction reflects the fact that each process is modeled with its own, local environment.

Since actions are at most binary, we can define an undirected graph $\mathcal{CG}$ with node set $\mathbb{P}$ and edges $\{p, q\}$ if there exists $a \in \Sigma$ with $dom(a) = \{p, q\}$, $p \neq q$. Such a graph is called *communication graph*. We assume throughout this section that $\mathcal{CG}$ is *acyclic* and has at least one edge. This allows us to choose a leaf $\ell \in \mathbb{P}$ in $\mathcal{CG}$, with $\{r, \ell\}$ an edge in $\mathcal{CG}$. So, in this section ℓ denotes this fixed leaf process and r its parent process. Starting from a control problem with input $\mathcal{A}$, $(F_p)_{p\in\mathbb{P}}$ we will reduce it to a control problem over the smaller (acyclic) graph $\mathcal{CG}' = \mathcal{CG}_{\mathbb{P}\setminus\{\ell\}}$. The reduction will work in exponential-time. If we represent $\mathcal{CG}$ as a tree of depth l then applying this construction iteratively we will get an l-fold exponential algorithm to solve the control problem for the $\mathcal{CG}$ architecture.

The main idea is that process r can simulate the behavior of process ℓ. Indeed, after each synchronization between r and ℓ, the views of both processes are identical, and until the next synchronization (or termination) ℓ evolves locally. The way r simulates ℓ is by "guessing" the future local evolution of ℓ until the next synchronizations (or termination) in a summarized form. Correctness is ensured by letting the environment challenge the guesses.

In order to proceed this way, we first show that winning control strategies can be assumed to satisfy a "separation" property concerning the synchronizations of process r (cf. 2nd item of Lemma 1):

Lemma 1. *If there exists a winning strategy for controlling $\mathcal{A}$, then there is one, say σ, such that for every $u \in Plays(\mathcal{A}, \sigma)$ the following holds:*

1. *For every process p and $A = \sigma_p(view_p(u))$, we have either $A \subseteq \Sigma_p^{com}$ or $A = \{a\}$ for some $a \in \Sigma_p^{loc}$.*
2. *Let $A = \sigma_r(view_r(u))$ with $A \subseteq \Sigma_r^{com}$. Then either $A \subseteq \Sigma_{r,\ell}$ or $A \subseteq \Sigma_r^{com} \setminus \Sigma_{r,\ell}$ holds.*

It is important to note that the 2nd item of Lemma 1 only holds when final states are blocking. To see this, consider the client-server example in Fig. 1 and assume that environment can either put a client directly to a final state, or oblige him to synchronize with the server before going to the final state. Suppose that all states of the server are final. In this case, the server's strategy must propose synchronization with *all* clients at the same time in order to guarantee that all clients can reach their final states.

Lemma 1 implies that the behavior of process ℓ can be divided in phases consisting of a local game ending in states where the strategy proposes communications with r and no local actions. This allows to define summaries of results of local plays of the leaf process ℓ. We denote by $state_\ell(v)$ the ℓ-component of the state reached on $v \in \Sigma^*$ from the initial state. Given a strategy $\sigma = (\sigma_p)_{p \in \mathbb{P}}$ and a play $u \in Plays_\ell(\mathcal{A}, \sigma)$, we define:

$$Sync_\ell^\sigma(u) = \{(t_\ell, A) \mid \exists x \in (\Sigma_\ell^{loc})^* \,.\, ux \text{ is a } \sigma\text{-play}, state_\ell(ux) = t_\ell, \\ \sigma_\ell(ux) = A \subseteq \Sigma_{r,\ell}, \text{ and } A = \emptyset \text{ iff } t_\ell \text{ is final}\}.$$

Since our winning conditions are local reachability conditions, we can show that it suffices to consider memoryless local strategies for process ℓ until the next synchronization with r (or until termination). Moreover, since final states are blocking, either all possible local plays from a given ℓ-state ultimately require synchronization with r, or they all terminate in a final state of ℓ (mixing the two situations would result in a process blocked on communication).

Lemma 2. *If there exists a winning strategy for controlling $\mathcal{A}$, then there is one, say $\sigma = (\sigma_p)_{p \in \mathbb{P}}$, such that for all plays $u \in Plays_\ell(\mathcal{A}, \sigma)$ the following hold:*

1. *Either $Sync_\ell^\sigma(u) \subseteq (S_\ell \setminus F_\ell) \times (2^{\Sigma_{r,\ell}} \setminus \{\emptyset\})$ or $Sync_\ell^\sigma(u) \subseteq F_\ell \times \{\emptyset\}$.*
2. *If uy is a σ-play with $y \in (\Sigma \setminus \Sigma_\ell)^*$, $\sigma_r(view_r(uy)) = B \subseteq \Sigma_{r,\ell}$ and $B \neq \emptyset$, then for every $(t_\ell, A) \in Sync_\ell^\sigma(u)$ some action from $A \cap B$ is enabled in $(state_r(uy), t_\ell)$.*
3. *There is a memoryless local strategy $\tau : S_\ell \to (\Sigma_\ell^{sys} \cap \Sigma_\ell^{loc})$ to reach from $state_\ell(u)$ the set of local states $\{t_\ell \mid (t_\ell, A) \in Sync_\ell^\sigma(u)$ for some $A\}$.*

The second item of the lemma says that every evolution of r should be compatible with every evolution of ℓ. The memoryless strategy from the third item proposes local actions of ℓ based only on the current state of ℓ and not on the history of the play. This strategy is used in a game on the transition graph of process ℓ. The third item of the lemma follows from the fact that 2-player games with reachability objectives admit memoryless winning strategies.

Definition 2. *An* admissible plan T from $s_\ell \in S_\ell$ *for process* ℓ *is either a subset of* $(S_\ell \setminus F_\ell) \times (2^{\Sigma_{r,\ell}} \setminus \{\emptyset\})$ *or a subset of* $F_\ell \times \{\emptyset\}$*, such that there exists a memoryless local strategy* $\tau : S_\ell \to (\Sigma_\ell^{sys} \cap \Sigma_\ell^{loc})$ *to reach from* s_ℓ *the set* $\{t_\ell \mid (t_\ell, A) \in T \text{ for some } A\}$*. An admissible plan* T *is* final *if* $T \subseteq F_\ell \times \{\emptyset\}$*.*

So Lemma 2 states that if there exists a winning strategy then there is one, say σ such that $Sync_\ell^\sigma(u)$ is an admissible plan for every σ-play u. Note also that we can check in polynomial time whether T as above is an admissible plan.

We are now ready to define informally the reduced plant $\mathcal{A}'$ on the process set $\mathbb{P}' = \mathbb{P} \setminus \{\ell\}$, that is the result of eliminating process ℓ. The only part that changes in $\mathcal{A}$ concerns process r, who now simulates former processes r and ℓ. The new process r starts in state $\langle s_{in,r}, s_{in,\ell}\rangle$. It will get into a state from $S_r \times S_\ell$ every time it simulates a synchronization between the former r and ℓ. Between these synchronizations its behaviour is as follows.

- From a state of the form $\langle s_r, s_\ell\rangle$, process r can do a controllable action $ch(T)$, for every admissible plan T from s_ℓ, and go to state $\langle s_r, T\rangle$.
- From a state of the form $\langle s_r, T\rangle$ process r can behave as the former r: it can either do a local action (controllable or not) or a shared action with some $p \neq \ell$, that updates the S_r-component to some $\langle s'_r, T\rangle$.
- From a state $\langle s_r, T\rangle$ process r can also do a controllable action $ch(B)$ for some $B \subseteq \Sigma_{r,\ell}$ and go to state $\langle s_r, T, B\rangle$; from $\langle s_r, T, B\rangle$ there are new, uncontrollable actions of the form (a, t_ℓ) where $(t_\ell, A) \in T$ and $a \in A \cap B$ such that $(s_r, t_\ell) \xrightarrow{a} (s'_r, t'_\ell)$ in $\mathcal{A}$. This case simulates r choosing a set of synchronization actions with ℓ, and the synchronization itself. For correctness of this step it is important that B is chosen such that for every $(t_\ell, A) \in T$ there is some $a \in A \cap B$ enabled in (s_r, t_ℓ).

Finally, accepting states of r in $\mathcal{A}'$ are $F_r \times F_\ell$, and $\langle s_r, T\rangle$ for $s_r \in F_r$ and T a final plan. The proof showing that this construction is correct provides a reduction from the control game on $\mathcal{A}$ to the control game on $\mathcal{A}'$.

Theorem 2. *Let* ℓ *be the fixed leaf process with* $\mathbb{P}' = \mathbb{P} \setminus \{\ell\}$ *and* r *its parent. Then the system has a winning strategy for* $\mathcal{A}, (F_p)_{p\in\mathbb{P}}$ *iff it has one for* $\mathcal{A}', (F'_p)_{p\in\mathbb{P}'}$*. The size of* $\mathcal{A}'$ *is* $|\mathcal{A}| + \mathcal{O}(M_r 2^{M_\ell |\Sigma_{r\ell}|})$*, where* M_r *and* M_ℓ *are the sizes of processes* r *and* ℓ *in* $\mathcal{A}$*, respectively.*

Remark 1. Note that the bound on $|\mathcal{A}'|$ is better than $|\mathcal{A}| + \mathcal{O}(M_r 2^{M_\ell 2^{|\Sigma_{r\ell}|}})$ obtained by simply counting all possible states in the description above. The reason is that we can restrict admissible plans to be (partial) functions from S_ℓ into $2^{\Sigma_{r,\ell}}$. That is, we do not need to consider different sets of communication actions for the same state in S_ℓ.

Let us reconsider the example from Figure 1 of a server with k clients. Applying our reduction k times we reduce out all the clients and obtain the single process plant whose size is $M_s 2^{(M_1+\cdots+M_k)c}$ where M_s is the size of the server, M_i is the size of client i, and c is the maximal number of communication actions between a client and the server. Our first main result also follows by applying the above reduction iteratively.

Theorem 3. *The control problem for distributed plants with acyclic communication graph is decidable. There is an algorithm for solving the problem (and computing a finite-state controller, if it exists) whose running time is bounded by a tower of exponentials of height equal to half of the diameter of the graph.*

4 The lower bound

Our main objective now is to show how using a communication architecture of diameter l one can code a counter able to represent numbers of size $Tower(2, l)$ (with $Tower(n, l) = 2^{Tower(n,l-1)}$ and $Tower(n, 1) = n$). Then an easy adaptation of the construction will allow to encode computations of Turing machines with the same space bound as the capabilities of the counters.

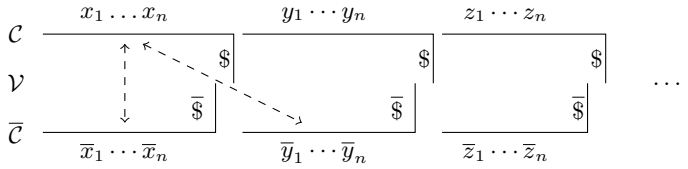


Fig. 2. Shape of a trace with 3 processes. Dashed lines show two types of tests.

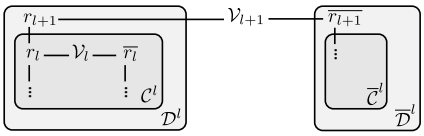
Let us first explain the mechanism we will use. Consider a trace of the shape presented in Figure 2. There are three processes $\mathcal{C}, \overline{\mathcal{C}}$ and $\mathcal{V}$. Process $\mathcal{C}$ repeatedly generates a sequence of n local actions and then synchronizes on action \$ with the verifier process $\mathcal{V}$. Process $\overline{\mathcal{C}}$ does the same. The alphabets of $\mathcal{C}$ and $\overline{\mathcal{C}}$ are of course disjoint. The verifier process $\mathcal{V}$ always synchronizes first with $\overline{\mathcal{C}}$ and then with $\mathcal{C}$. Observe that the actions $\overline{y}_1 \cdots \overline{y}_n$ are concurrent to both $x_1 \cdots x_n$ and $y_1 \cdots y_n$, but they are before z_1. Suppose that we allow the environment to stop this generation process at any moment. Say it stops $\mathcal{C}$ at some x_i, and $\overline{\mathcal{C}}$ at $\overline{x}_i$. We can then set the processes in such a way that they are forced to communicate x_i and $\overline{x}_i$ to $\mathcal{V}$; who can verify if they are correct. The other possibility is that the environment stops $\mathcal{C}$ at x_i and $\overline{\mathcal{C}}$ at $\overline{y}_i$ forcing the comparison of x_i with $\overline{y}_i$. This way we obtain a mechanism allowing to compare position by position the sequence $x_1 \cdots x_n$ both with $\overline{x}_1 \cdots \overline{x}_n$ and with $\overline{y}_1 \cdots \overline{y}_n$. Observe that $\mathcal{V}$ knows which of the two cases he deals with, since the comparison with the latter sequence happens after some $\overline{\$}$ and before the next \$. Now, we can use sequences of n letters to encode numbers from 0 to $2^n - 1$. Then this mechanism permits us to verify if $x_1 \cdots x_n$ represents the same number as $\overline{x}_1 \cdots \overline{x}_n$ and the predecessor of $\overline{y}_1 \cdots \overline{y}_n$. Applying the same reasoning to $y_1 \cdots y_n$ we can test that it represents the same number as $\overline{y}_1 \cdots \overline{y}_n$ and the predecessor of $\overline{z}_1 \cdots \overline{z}_n$. If some test fails, the environment wins. If the environment does not stop $\mathcal{C}$ and $\overline{\mathcal{C}}$ at the same position, or stops only one of them, the system wins. So this way we force the processes $\mathcal{C}$ and $\overline{\mathcal{C}}$ to cycle through representations of numbers from 0 to

$2^n - 1$. Building on this idea we can encode alternating polynomial space Turing machines, and show that the control problem for this three process architecture (with diameter 2) is EXPTIME-hard. The algorithm from the previous section provides the matching upper bound.

After this explanation let us introduce general counters. We start with their alphabets. Let $\Sigma_i = \{a_i, b_i\}$ for $i = 1, \ldots, n$. We will think of a_i as 0 and b_i as 1, mnemonically: 0 is round and 1 is tall. Let $\Sigma_i^{\#} = \Sigma_i \cup \{\#_i\}$ be the alphabet extended with an end marker.

A 1-counter is just a letter from Σ_1 followed by $\#_1$. The value of a_1 is 0, and the one of b_1 is 1. An $(l+1)$-*counter* is a word $x_0 u_0 x_1 u_1 \cdots x_{k-1} u_{k-1} \#_{l+1}$ where $k = \mathit{Tower}(2, l)$, and for every $i < k$ we have: $x_i \in \Sigma_{l+1}$ and u_i is an l-counter with value i. The value of the above $(l+1)$-counter is $\sum_{i=0,\ldots,k} x_i 2^i$. The end marker $\#_{l+1}$ is there for convenience. An *iterated* $(l+1)$-*counter* is a nonempty sequence of $(l+1)$-counters (we do not require that the values of consecutive $(l+1)$-counters are consecutive).

Suppose that we have already constructed a plant $\mathcal{C}^l$ with root process r_l, such that every winning strategy in $\mathcal{C}^l$ needs to produce an iterated l-counter on r_l. We now define $\mathcal{C}^{l+1}$, a plant where every winning strategy needs to produce an iterated $(l+1)$-counter on its root process r_{l+1}. Recall that such a counter is a sequence of l-counters with values $0, 1, \ldots, (\mathit{Tower}(2, l)-1), 0, 1, \ldots$ The plant $\mathcal{C}^{l+1}$ is made of two copies of $\mathcal{C}^l$, that we name $\mathcal{D}^l$ and $\overline{\mathcal{D}^l}$. We add three processes: $r_{l+1}, \overline{r_{l+1}}, \mathcal{V}_{l+1}$. The root r_{l+1} of $\mathcal{C}^{l+1}$ communicates with $\mathcal{V}_{l+1}$ and with the root r_l of $\mathcal{D}^l$, while $\overline{r_{l+1}}$ communicates with $\mathcal{V}_{l+1}$ and with the root of $\overline{\mathcal{D}^l}$.



In order to force $\mathcal{C}^{l+1}$ to generate an $(l+1)$-counter, we allow the environment to compare using $\mathcal{V}_{l+1}$ the sequence generated by r_{l+1} and the sequence generated by $\overline{r_{l+1}}$. The mechanism is similar to the example above. After each letter of Σ_l, we add an uncontrollable action that triggers the comparison between the current letters of Σ_l on r_{l+1} and on $\overline{r_{l+1}}$. This may correspond to two types of tests: equality or successor. For equality, $\mathcal{V}_{l+1}$ enters a losing state if (1) the symbols from r_{l+1} and from $\overline{r_{l+1}}$ are different; and (2) the number of remaining letters of Σ_l before $\#_l$ is the same on both r_{l+1} and $\overline{r_{l+1}}$. The latter test ensures that the environment has put the challenge at the same positions of the two counters. The case for successor is similar, accounting for a possible carry. In any other case (for instance, if the test was issued on one process only instead of both r_{l+1} and $\overline{r_{l+1}}$), the test leads to a winning configuration.

The challenge with this schema is to keep r_{l+1} and $\overline{r_{l+1}}$ synchronized in a sense that either (i) the two should be generating the same l-counter, or (ii) r_{l+1} should be generating the consecutive counter with respect to the one generated by $\overline{r_{l+1}}$. For this, a similar communication mechanism based on \$ symbols as in the example above is used. An action $\$_l$ shared by r_{l+1} and $\mathcal{V}_{l+1}$ is executed after each l-counter, that is after each $\#_l$ shared between r_{l+1} and r_l. Similarly with

action $\overline{\$_l}$ shared by $\overline{r_{l+1}}$ and $\mathcal{V}_{l+1}$. Process $\mathcal{V}_{l+1}$ switches between state *eq* and state *succ* when receiving $\overline{\$_l}$, and back when receiving $\$_l$ so it knows whether $\overline{r_{l+1}}$ is generating the same l-counter as r_{l+1}, or the next one. As $\overline{r_{l+1}}$ does not synchronize (unless there is a challenge) with $\mathcal{V}_{l+1}$ between two $\overline{\$_l}$, it does not know whether r_{l+1} has already started producing the same l-counter or whether it is still producing the previous one. Another important point about the flow of knowledge is that while r_l is informed when r_{l+1} is being challenged (as it synchronizes frequently with r_{l+1}, and could thus be willing to cheat to produce a different l-counter), $\overline{r_l}$ does not know that r_{l+1} is being challenged, and thus cheating on r_l would be caught by verifier $\mathcal{V}_l$.

Proposition 1. *For every l, the system has a winning strategy in $\mathcal{C}^l$. For every such winning strategy σ, if we consider the unique σ-play without challenges then its projection on $\bigcup_{i=1,\dots,l} \Sigma_i^{\#}$ is an iterated l-counter.*

Proposition 1 is the basis for encoding Turing machines, with $\mathcal{C}^l$ ensuring that the space bound is equal to $Tower(n, l)$.

Theorem 4. *Let $l > 0$. There is an acyclic architecture of diameter $(4l-2)$ and with $3(2^l-1)$ processes such that the space complexity of the control problem for it is $\Omega(Tower(n, l))$-complete.*

5 Conclusions

Distributed synthesis is a difficult and at the same time promising problem, since distributed systems are intrinsically complex to construct. We have considered here an asynchronous, shared-memory model. Already Pnueli and Rosner in [16] strongly argue in favour of asynchronous distributed synthesis. The choice of transmitting additional information while synchronizing is a consequence of the model we have adopted. We think that it is interesting from a practical point of view, since it is already used in multithreaded computing (e.g., CAS primitive) and it offers more decidable settings (e.g., client-server architecture).

Under some restrictions we have shown that the resulting control problem is decidable. The assumption about uncontrollable actions being local represents the most common situation where each process comes with its own environment (e.g., a client). The assumption on binary synchronizations simplifies the definition of architecture graph and is common in distributed algorithms. The most important restriction is that on architectures being a tree. Tree architectures are quite rich and allow to model hierarchical situations, like server/clients (recall that such cases are undecidable in the setting of Pnueli and Rosner). Nevertheless, it would be very interesting to know whether the problem is still decidable e.g. for ring architectures. Such an extension would require new proof ideas. A more immediate task is to consider more general winning conditions. A further interesting research direction is the synthesis of open, concurrent recursive programs, as considered e.g. in [1].

Our non-elementary lower bound result is somehow surprising. Since we have full information sharing, all the complexity is hidden in the uncertainty about actions performed in parallel by other processes.

References

1. B. Bollig, M.-L. Grindei, and P. Habermehl. Realizability of concurrent recursive programs. In *FOSSACS*, volume 5504 of *LNCS*, pages 410–424, 2009.
2. A. Church. Logic, arithmetics, and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35, 1962.
3. P. Clairambault, J. Gutierrez, and G. Winskel. The winning ways of concurrent games. In *LICS*, pages 235–244. IEEE, 2012.
4. V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, 1995.
5. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *LICS*, pages 321–330. IEEE, 2005.
6. P. Gastin, B. Lerman, and M. Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *FSTTCS*, volume 3328 of *LNCS*, pages 275–286, 2004.
7. P. Gastin, N. Sznajder, and M. Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3):215–237, 2009.
8. B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Optimal Zielonka-type construction of deterministic asynchronous automata. In *ICALP*, volume 6199 of *LNCS*, 2010.
9. G. Katz, D. Peled, and S. Schewe. Synthesis of distributed control through knowledge accumulation. In *CAV*, volume 6806 of *LNCS*, pages 510–525. 2011.
10. O. Kupferman and M. Vardi. Synthesizing distributed systems. In *LICS*, 2001.
11. P. Madhusudan and P. Thiagarajan. Distributed control and synthesis for local specifications. In *ICALP*, volume 2076 of *LNCS*, pages 396–407, 2001.
12. P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In *FSTTCS*, volume 3821 of *LNCS*, 2005.
13. A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.
14. P.-A. Melliès. Asynchronous games 2: The true concurrency of innocence. *TCS*, 358(2-3):200–228, 2006.
15. R. V. D. Meyden and T. Wilke. Synthesis of distributed systems from knowledge-based specifications. In *CONCUR, volume 3653 of LNCS*, pages 562–576, 2005.
16. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP*, volume 372, pages 652–671, 1989.
17. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757, 1990.
18. P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(2):81–98, 1989.
19. S. Schewe and B. Finkbeiner. Synthesis of asynchronous systems. In *LOPSTR*, number 4407 in LNCS, pages 127–142. 2006.
20. W. Zielonka. Notes on finite asynchronous automata. *RAIRO–Theoretical Informatics and Applications*, 21:99–135, 1987.