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/Universit 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 [19]. 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 [18]. 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* [22]. 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 [6,13] 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).

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 [18] (each process has inputs), and is not covered by [6] (the action alphabet is not a co-graph), nor by [13] (there is no bound on the number of actions performed concurrently).

Related work. The setting proposed by Pnueli and Rosner [18] 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 [11] 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 [12] 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 [16] the authors consider distributed synthesis for knowledge-based specifications. The paper [7] studies an interesting case of external specifications and well-connected architectures.

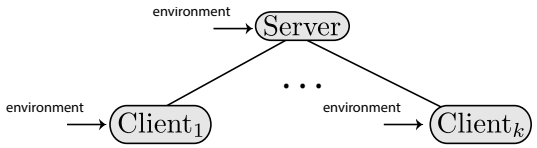


Fig. 1. Server/client architecture

Synthesis for asynchronous systems has been strongly advocated by Pnueli and Rosner in [17]. 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 [20,9] 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 that 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 [13] 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 [13] 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 [15] 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.

2 Basic definitions and observations

Our control problem can be formulated in the same way as the Ramadge and Wonham control problem but using Zielonka automata instead of standard finite automata. We start by presenting Zielonka automata and an associated notion of concurrency. Then we briefly recall the Ramadge and Wonham formulation and our variant of it. Finally, we give a more convenient game-based formulation of the problem.

2.1 Zielonka automata

Zielonka automata are simple parallel devices. Such an automaton is a parallel composition of several finite automata, denoted as *processes*, synchronizing on common 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 denoted as *p-actions*.

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.

This definition has an important consequence. The location mapping *dom* defines in a natural way an independence relation I: two actions $a, b \in \Sigma$ are independent (written as $(a, b) \in I$) if they involve different processes, that is, if $dom(a) \cap dom(b) = \emptyset$. Notice that the order of execution of two independent actions $(a, b) \in I$ in a Zielonka automaton is irrelevant, they can be executed as a, b, or b, a - or even concurrently. More generally, we can consider the congruence $\sim_I$ on Σ^* generated by I, and observe that whenever $u \sim_I v$, the global state reached from the initial state on u and v, respectively, is the same. Hence, $u \in L(\mathcal{A})$ if and only if $v \in L(\mathcal{A})$. Notice also that if $u \sim_I vx$ and $x \in \Sigma^*$ involves no p-action, then the p-state reached on u and v, respectively, is the same.

The idea of describing concurrency by an independence relation on actions goes back to the late seventies, to Mazurkiewicz [14] and Keller [10] (see also [4]). An equivalence class $[w]_I$ of $\sim_I$ is called a Mazurkiewicz *trace*, it can be also viewed as labeled pomset of a special kind. Here, we will often refer to a trace using just a word w instead of writing $[w]_I$. As we have observed $L(\mathcal{A})$ is a sum of such equivalence classes. In other words it is *trace-closed*.

Example 1. Consider the following, very simple, example with processes $1, 2, 3$. Process 1 has local actions a_0, a_1 and synchronization actions $c_{i,j}$ $(i, j = 0, 1)$ shared with process 2. Similarly, process 3 has local actions b_0, b_1 and synchronization actions $d_{i,j}$ $(i, j = 0, 1)$ shared with process 2 (cf. Figure 2 where the symbol $*$ denotes any value 0 or 1). Each process is a finite automaton and the Zielonka automaton is the product of the three components synchronizing on common actions. We have for instance $(a_i, b_j) \in I$ and $(c_{i,j}, d_{k,l}) \notin I$. The final states are the rightmost states of each automaton. The automaton accepts traces of the form $a_i b_j c_{i,k} d_{j,l}$ with $i = l$ or $j = k$.

Since the notion of a trace can be formulated without a reference to an accepting device, it is natural to 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. *[22] 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*

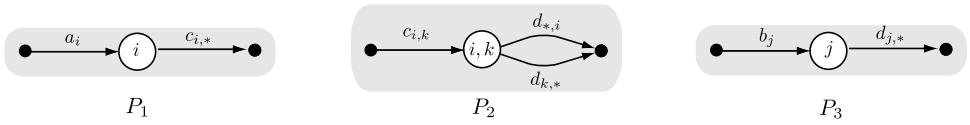


Fig. 2. A Zielonka automaton

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]).

One could try to use Zielonka's theorem directly to solve a distributed control problem. For example, one can start with the Ramadge and Wonham control problem, solve it, and if a solution happened to respect the required independence, then distribute it. Unfortunately, there is no reason for the solution to respect the independence. Even worse, the following, relatively simple, result says that it is algorithmically impossible to approximate a regular language by a language respecting a given independence relation.

Theorem 2. *[21] It is not decidable if, given a distributed alphabet and a regular language $L \subseteq \Sigma^*$, there is a trace-closed language $K \subseteq L$ such that every letter from Σ appears in some word of K.*

The condition on appearance of letters above is not crucial for the above undecidability result. Observe that we need some condition in order to make the problem nontrivial, since by definition the empty language is trace-closed.

2.2 The control problem

We can now formulate our control problem as a variant of the Ramadge and Wonham formulation. We will then provide an equivalent description of the problem in terms of games. While more complicated to state, this description is easier to work with.

Recall that in Ramadge and Wonham's control problem [19] 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 such that the product $P \times C$ satisfies a given specification. Here both the plant and the controller are finite deterministic automata over 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}.

Our control problem can be formulated as follows: 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 the controller is required not to block uncontrollable actions:

from every state of C every uncontrollable action should be possible. The important point is that the controller should have the same distributed structure as the plant. The product of the two automata, that is just the standard product, means that plant and controller are totally synchronized, in particular communications between processes happen at the same time. Hence concurrency in the controlled system is the same as in the plant. The major difference between the controlled system and the plant is that the states carry the additional information computed by the controller.

Example 2. Reconsider the automaton in Figure 2 and assume that $a_i, b_j \in \Sigma^{env}$ are uncontrollable. So the controller needs to propose controllable actions $c_{i,k}$ and $d_{j,l}$, resp., in such a way that all processes reach their final state. In particular, process 2 should not block. At first sight this may seem impossible to guarantee, as it looks like process 1 needs to know what b_j process 3 has received, or process 3 needs to know about the a_i received by process 1. Nevertheless, a controller exists. It consists of P_1 proposing $\{c_{ii}\}$ at state i, process P_3 proposing $\{d_{j,1-j}\}$ at state j and process P_2 proposing all actions. If $i = j$ then P_2 reaches the final state by the transition $d_{k,*}$, else by the transition $d_{*,i}$.

It will be more convenient to work with a game formulation of this problem. Instead of talking about controller we will talk about distributed strategy 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 $\mathit{Plays}(\mathcal{A})$ 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 $\mathit{view}_p(u)$, is the smallest trace $[v]_I$ such that $u \sim_I vy$ and y contains no action from Σ_p. We write $\mathit{Plays}_p(\mathcal{A})$ for the set of plays that are p-views:

$$\mathit{Plays}_p(\mathcal{A}) = \{\mathit{view}_p(u) \mid u \in \mathit{Plays}(\mathcal{A})\}.$$

A *strategy for a process* p is a function $\sigma_p : \mathit{Plays}_p(\mathcal{A}) \to 2^{\Sigma_p^{sys}}$, where $\Sigma_p^{sys} = \{a \in \Sigma^{sys} \mid p \in \mathit{dom}(a)\}$. We require in addition, for every $u \in \mathit{Plays}_p(\mathcal{A})$, 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 $\mathit{Plays}(\mathcal{A}, \sigma)$, is the smallest set containing the empty play ε, and such that for every $u \in \mathit{Plays}(\mathcal{A}, \sigma)$:

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

Intuitively, the definition says that actions of the environment are always possible, whereas actions of the system are possible only if they are allowed by the strategies of all involved processes. As in [13] (and unlike [6]) our strategies are process-based. That is, a controllable action a with $dom(a) = \{p, q\}$ is allowed from (s_p, s_q) if it is proposed by process p in state s_p and by process q in state s_q. Before defining winning strategies, we need to introduce infinite plays that are consistent with a given strategy σ. Such plays can be seen as (infinite) traces associated with infinite, initial runs of $\mathcal{A}$ satisfying the two conditions of the definition of $Plays(\mathcal{A}, \sigma)$. We write $Plays^{\infty}(\mathcal{A}, \sigma)$ for the set of finite or infinite such plays. A play from $Plays^{\infty}(\mathcal{A}, \sigma)$ is also denoted as σ-*play*.

A play $u \in Plays^{\infty}(\mathcal{A}, \sigma)$ is called *maximal*, if there is no action c such that $uc \in Plays^{\infty}(\mathcal{A}, \sigma)$. In particular, u is maximal if $view_p(u)$ is infinite for every process p. Otherwise, if $view_p(u)$ is finite then p cannot have enabled local actions (either controllable or uncontrollable). Moreover there should be no communication possible between any two processes with finite views in u.

In this paper we consider *local reachability* winning conditions. For this, every process has a set of target states $F_p \subseteq S_p$. We 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 $p \in dom(a)$.

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 $u \in Plays^{\infty}(\mathcal{A}, \sigma)$ ends in $\prod_{p\in\mathbb{P}} F_p$ (and is thus finite). Such traces and strategies are called* winning.

As already mentioned, we do not know if this control problem is decidable in general. In this paper we put one restriction on possible communications between processes. First, 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 $a \in \Sigma^{env}$. So the first restriction says that we allow only binary synchronizations. It makes the technical reasoning much simpler. The second restriction reflects the fact that each process is modeled with its own, local environment.

Definition 2. *A distributed alphabet (Σ, dom) with unary and binary actions defines 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.

3 The upper bound for acyclic communication graphs

We fix in this section a distributed alphabet (Σ, dom). According to Definition 2 the alphabet determines a communication graph $\mathcal{CG}$. We assume that $\mathcal{CG}$ is acyclic and has at least one edge. This allows us to choose a leaf $r \in \mathbb{P}$ in $\mathcal{CG}$, with $\{q, r\}$ an edge in $\mathcal{CG}$. Throughout this section, r denotes this fixed leaf process and q its parent process. Starting from a control problem with input

$\mathcal{A}$, $(F_p)_{p\in\mathbb{P}}$ we define below a control problem over the smaller (acyclic) graph $\mathcal{CG}' = \mathcal{CG}_{\mathbb{P}\setminus\{r\}}$. The construction will be an exponential-time reduction from the control problem over $\mathcal{CG}$ to a control problem over $\mathcal{CG}'$. 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 $\mathcal{CG}$ architecture.

The main idea of the reduction is simple: process q simulates the behavior of process r. The reason why a simulation can work is that after each synchronization between q and r, the views of both processes are identical, and between two such synchronizations r evolves locally. But the construction is more delicate than this simple description suggests, and needs some preliminary considerations about winning strategies.

We start with a lemma showing how to restrict the winning strategies. For $p, p' \in \mathbb{P}$ let $\Sigma_{p,p'} = \{a \in \Sigma \mid dom(a) = \{p, p'\}\}$. So $\Sigma_{p,p'}$ is the set of synchronization actions between p and p'. Moreover $\Sigma_{p,p}$ is just the set of local actions of p. We write Σ_p^{loc} instead of $\Sigma_{p,p}$ and $\Sigma_p^{com} = \Sigma_p \setminus \Sigma_p^{loc}$. Recall that in the lemma below r is the fixed leaf process, and q its parent.

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

1. *If an uncontrolable action is possible from a state s_r of process r then for every play u with $state_r(u) = s_r$ we have $\sigma_r(view_r(u)) = \emptyset$.*
2. *For every process p and $X = \sigma_p(view_p(u))$, we have either $X = \{a\}$ for some $a \in \Sigma_p^{loc}$ or $X \subseteq \Sigma_p^{com}$.*
3. *Let $X = \sigma_q(view_q(u))$ with $X \subseteq \Sigma_q^{com}$. Then either $X \subseteq \Sigma_{q,r}$ or $X \subseteq \Sigma_q^{com} \setminus \Sigma_{q,r}$ holds.*

Proof. The first item is immediate, since uncontrollable actions are alwyas possible. For the second item we modify σ into σ' as follows. If $\sigma_p(u)$ contains some local action, then we choose one, say a, and put $\sigma'_p(u) = \{a\}$. We do this for every process p and show that the resulting strategy σ' is winning. Suppose that $v \in Plays(\mathcal{A}, \sigma')$ is maximal, but not winning. Clearly v is a σ-play, but not a maximal one, since σ is winning. Thus, there is $vc \in Plays(\mathcal{A}, \sigma)$ for some processes $p \neq p'$ and some $c \in \Sigma_{p,p'}$. By definition of σ' it means that either $\sigma_p(view_p(v))$ or $\sigma_{p'}(view_{p'}(v))$ contains some local action, say $a \in \sigma_p(view_p(v))$ and $\sigma'_p(view_p(v)) = \{a\}$. But then va is a σ'-play, a contradiction with the maximality of v.

For the last item we can assume that σ_q and σ_r always propose either a local action or a set of communication actions. Now given a winning strategy σ we will produce a winning strategy σ' satisfying the condition of the lemma, by modifying only σ_r.

Assume that $u \in Plays_q(\mathcal{A}, \sigma)$ with $s_q = state_q(u)$, and $\sigma_q(u) = B \cup C$, where $B \subseteq \Sigma_{q,r}$ and $C \subseteq \Sigma_q^{com} \setminus \Sigma_{q,r}$ with both B, C non-empty. We define σ'_q by cases:

$$\sigma'_q(u) = \begin{cases} C & \text{there exists } (s_r, A) \in Sync_r^{\sigma}(u) \text{ with } (s_r, A) \bowtie (s_q, B) = \emptyset, \\ B & \text{otherwise.} \end{cases}$$

The idea behind the definition above is simple: if there is a possible local future for r that makes synchronization with q impossible (first case), then q's strategy can as well propose only communication with other processes than r – since such communication leads to winning as well. If not, q's strategy can offer only communication with r, since this choice will never block.

We show now that σ' is winning. Assume by contradiction that v is a maximal σ'-play, but not winning. It is then a σ-play, but not a maximal one. So there must be some $a \in \Sigma_q^{com}$ such that $va \in \mathit{Plays}(\mathcal{A}, \sigma)$. In particular, q's state after v is not final. Let $u = \mathit{view}_q(v)$, $s_q = \mathit{state}_q(u)$, and $\sigma_q(u) = B \cup C$ with $B \subseteq \Sigma_{q,r}$ and $C \subseteq \Sigma_q^{com} \setminus \Sigma_{q,r}$. We have two cases.

Suppose $\sigma'_q(u) = C$, so we are in the first case of the above above. Thus there exists $(s_r, A) \in \mathit{Sync}_r^\sigma(u)$ such that $(s_r, A) \bowtie (s_q, B) = \emptyset$. By definition of Sync_r^σ we find $x \in (\Sigma_r^{loc})^*$ such that $u' = ux$ is a σ-play and $\sigma_r(\mathit{view}_r(u')) = A$. Since $u = \mathit{view}_q(u')$, we have $\sigma_q(\mathit{view}_q(u')) = B \cup C$. This means that no communication between q and r is possible after u'. No local action of q is possible after u' since $u = \mathit{view}_q(v)$, and we have assumed that v is a maximal σ'-play. Finally, by the choice of x, no local action of process r is possible from u'. To obtain a contradiction it suffices to show that u' can be extended to a maximal σ-play by adding a sequence of actions w of processes other than q and r. This will do as $\mathit{state}_q(u')$ is not accepting by assumption, and we will get a maximal σ-play that is not winning. To find the desired w observe that $v \sim uwy$ where $w \in (\Sigma \setminus (\Sigma_q \cup \Sigma_r))^*$ and $y \in \Sigma_r^*$. So y represents the actions of r after the last action of q in v, and w represents the actions of other processes. Taking $v' = uwx$ we observe that $v' \sim u'w$ and that v' is a maximal σ-play. So we have found the desired w.

The second case is when $\sigma'_q(u) = B$. This means that for all $(s_r, A) \in \mathit{Sync}_\ell^\sigma(u)$, we have $(s_r, A) \bowtie (s_r, B) \neq \emptyset$. Since v is a maximal σ'-play, no local action of r is possible. This means that $(s_r, A) := (\mathit{state}_r(v), \sigma_r(\mathit{view}_r(v))) \in \mathit{Sync}_r^\sigma(u)$. But then $(s_r, A) \bowtie (s_q, B) \neq \emptyset$. Since $\sigma'_q(u) = B$ there is some possible communication between q and r after v, so v is not maximal w.r.t. σ'. □

The following definition associates with a strategy σ and the leaf process r all the outcomes of local plays of r such that r is either waiting for a synchronization with q or is in a final (hence blocking) state. For an initial run u of $\mathcal{A}$ we denote by $\mathit{state}_p(u)$ the p-state reached by $\mathcal{A}$ on u.

Definition 3. *Given a strategy σ and a σ-play u, let $\mathit{Sync}_r^\sigma(u) \subseteq S_r \times \mathcal{P}(\Sigma_{q,r})$ be the set:*

$$\begin{aligned}\mathit{Sync}_r^\sigma(u) = \{(s_r, A) \mid\ & \exists x \in (\Sigma_r^{loc})^*\,.\ ux \text{ is a } \sigma\text{-play,} \\ & \mathit{state}_r(ux) = s_r,\ \sigma_r(\mathit{view}_r(ux)) = A \subseteq \Sigma_{q,r}, \text{ and} \\ & s_r \text{ final or } A \neq \emptyset\}\,.\end{aligned}$$

Observe that if σ allows r to reach a final state s_r from u without communication, then $(s_r, \emptyset) \in \mathit{Sync}_r^\sigma(u)$. This is so, since final states are assumed to be blocking.

For the game reduction we need to precalculate all possible sets Sync_r^σ. These sets will be actually of the special form described below.

Definition 4. *Let s_r be a state of r. We say that $T \subseteq S_r \times \mathcal{P}(\Sigma_{qr})$ is an* admissible plan *in s_r if there is a play u with $state_r(u) = s_r$, and a strategy σ such that (i) $T = Sync_r^\sigma(u)$, (ii) every σ-play of r from s_r reaches a final state or a state where σ proposes some communication action, and (iii) one of the following holds:*

- *$A \neq \emptyset$ for every $(t_r, A) \in T$, or*
- *$t_r \in F_r$ and $A = \emptyset$ for every $(t_r, A) \in T$.*

In the second case T is called a final plan.

It is not difficult to see that we can compute the set of all admissible plans. In the above definition we do not ask that σ is winning in the global game, but just that it can locally bring r to one of the situations described by T. So verifying if T is an admisible plan simply amounts to solve a 2-players reachability game on process r against the (local) environment.

Lemma 2 below allows to deduce that the sets $Sync_r^\sigma$ are admissible plans whenever σ is winning. For $(s_r, A), (s_q, B)$ with $s_q \in S_q, s_r \in S_r$, $A, B \subseteq \Sigma_{q,r}$ let $(s_r, A) \bowtie (s_q, B) := \{a \in A \cap B \mid \delta_a(s_q, s_r) \text{ is defined}\}$. So $(s_r, A) \bowtie (s_q, B)$ contains all actions belonging to both A and B, that are enabled in the state (s_q, s_r).

Lemma 2. *If σ is a winning strategy satisfying Lemma 1 then for every σ-play u in $\mathcal{A}$ we have:*

1. *if there is some σ-play uy with $y \in (\Sigma \setminus \Sigma_r)^*$ and $state_q(uy) \in F_q$ then $Sync_r^\sigma(u)$ is a final plan;*
2. *if there is some σ-play uy with $y \in (\Sigma \setminus \Sigma_r)^*$, $s_q = state_q(uy)$, $\sigma_q(uy) = B \subseteq \Sigma_{q,r}$, and $B \neq \emptyset$ then for every $(t_r, A) \in Sync_r^\sigma(u)$ we have $(s_q, B) \bowtie (t_r, A) \neq \emptyset$.*

In particular, $Sync_r^\sigma(u)$ is always an admissible plan.

Proof. Take y as in the statement of the lemma and suppose $state_q(uy) \in F_q$. Take $(t_r, A) \in Sync_r^\sigma(u)$. By definition this means that there is $x \in (\Sigma_r^{loc})^*$ such that ux is a σ-play, $state_r(ux) = t_r$, and $\sigma_r(view_r(ux)) = A$ with $A \subseteq \Sigma_{q,r}$. Observe that uyx is also a σ-play. Hence t_r should be final because after uyx process r can do at most communication with q, but this is impossible since q is in a final state. Since t_r is final, it cannot propose an action, hence $A = \emptyset$. This shows the first item of the lemma.

For the second item of the lemma take y, s_q, B, and (t_r, A) as in the assumption. Once again we get $x \in (\Sigma_r^{loc})^*$ such that ux is a σ-play, $state_r(ux) = t_r$, and $\sigma_r(view_r(ux)) = A$ with $A \subseteq \Sigma_{q,r}$. Once again uyx is a σ-play. We have that s_q is not final since $B \neq \emptyset$. As σ is winning, the play uyx can be extended by an action of q. But the only such action that is possible is a communication between q and r. Since A and B are the communication sets proposed by σ_r and σ_q, respectively, we must have $(s_q, B) \bowtie (t_r, A) \neq \emptyset$. □

The new plant $\mathcal{A}'$. We are now ready to define the reduced plant $\mathcal{A}'$ that is the result of eliminating process r. Let $\mathbb{P}' = \mathbb{P} \setminus \{r\}$. We have $\mathcal{A}' = \langle \{S'_p\}_{p \in \mathbb{P}'}, s'_{in}, \{\delta'_a\}_{a \in \Sigma'} \rangle$ where the components will be defined below.

The states of process q in $\mathcal{A}'$ are of one of the following types:

$$\langle s_q, s_r \rangle\,, \quad \langle s_q, T \rangle\,, \quad \langle s_q, T, B \rangle\,,$$

where $s_q \in S_q, s_r \in S_r$, $T \subseteq S_r \times \mathcal{P}(\Sigma_{q,r})$ is an admissible plan, $B \subseteq \Sigma_{q,r}$. The new initial state for q is $\langle (s_{in})_q, (s_{in})_r \rangle$.

For every $p \neq q$, we let $S'_p = S_p$ and $F'_p = F_p$. The local winning condition for q becomes $F'_q = F_q \times F_r \cup \{\langle s_q, T \rangle \mid s_q \in F_q, \text{ and } T \text{ is a final plan}\}$.

The set of actions Σ' is $\Sigma \setminus \Sigma_r$, plus additional local q-actions that we introduce below. All transitions δ_a with $dom(a) \cap \{q, r\} = \emptyset$ are as in $\mathcal{A}$. Regarding q we have the following transitions:

1. If not in a final state then process q chooses an admissible plan:

   $$\langle s_q, s_r \rangle \xrightarrow{ch(T)} \langle s_q, T \rangle,$$

   where T is an admissible plan in s_r, and $\langle s_q, s_r \rangle \notin F_q \times F_r$.
2. Local action of q:

   $$\langle s_q, T \rangle \xrightarrow{a} \langle s'_q, T \rangle, \quad \text{if } s_q \xrightarrow{a} s'_q \text{ in } \mathcal{A}\,.$$

3. Synchronization between q and $p \neq r$:

   $$(\langle s_q, T \rangle, s_p) \xrightarrow{b} (\langle s'_q, T \rangle, s'_p), \quad \text{if } (s_q, s_p) \xrightarrow{b} (s'_q, s'_p)\,.$$

4. Synchronization between q and r. Process q declares the communication actions with r:

   $$\langle s_q, T \rangle \xrightarrow{ch(B)} \langle s_q, T, B \rangle, \qquad \text{if } \ B \subseteq \Sigma_{q,r}$$

   when s_q is not final, T is not a final plan, and for every $(t_r, A) \in T$ we have $(t_r, A) \bowtie (s_q, B) \neq \emptyset$.
   Then the environment can choose the target state of r and a synchronization action $a \in \Sigma_{q,r}$:

   $$\langle s_q, T, B \rangle \xrightarrow{(a,t_r)} \langle s'_q, s'_r \rangle \qquad \text{if } (s_q, t_r) \xrightarrow{a} (s'_q, s'_r) \text{ in } \mathcal{A}$$

   for every (a, t_r) such that $(t_r, A) \in T$ for some A, and $a \in A \cap B$. Notice that the complicated name of the action (a, t_r) is needed to ensure that the transition is deterministic.

To summarize the new actions of process q in plant $\mathcal{A}'$ are:

- $ch(T) \in \Sigma^{sys}$, for every admissible plan T,
- $ch(B) \in \Sigma^{sys}$, for each $B \subseteq \Sigma_{q,r}$,
- $(a, t_r) \in \Sigma^{env}$ for each $a \in \Sigma_{q,r}, t_r \in S_r$.

The proof showing that this construction is correct provides a translation from winning strategies in $\mathcal{A}$ to winning strategies in $\mathcal{A}'$, and back. To this purpose we rely on a translation from plays in $\mathcal{A}$ to plays in $\mathcal{A}'$. A (finite or infinite) play u in $\mathcal{A}$ is a trace that will be convenient to view as a word of the form

$$u = y_0 x_0 a_1 \ \cdots \ a_i y_i x_i \ a_{i+1} \dots$$

where for $i \in \mathbb{N}$ we have that: $a_i \in \Sigma_{q,r}$ is communication between q and r; $x_i \in (\Sigma_r^{loc})^*$ is a sequence of local actions of r; and $y_i \in (\Sigma \setminus \Sigma_r)^*$ is a sequence of actions of other processes than r. Note that x_i, y_i are concurrent, for each i. We will write $u|_{a_i}$ for the prefix of u ending in a_i. Similarly $u|_{y_i}$ for the prefix ending with y_i; analogously for x_i.

Fix a strategy σ in $\mathcal{A}$. With a word u as above we will associate the word

$$\chi(u) = ch(T_0) y_0 \, ch(B_0)(a_1, t_r^1) \ \cdots \ (a_i, t_r^i) \, ch(T_i) \ y_i \, ch(B_i)(a_{i+1}, t_r^{i+1}) \dots$$

where for every $i = 0, 1, \dots$:

- $T_i = Sync_r^\sigma(u|_{a_i})$ and $T_0 = Sync_r^\sigma(\varepsilon)$;
- $B_i = \sigma_q(view_q(u|_{y_i}))$;
- $t_r^i = state_r(u|_{x_i})$.

We then construct a strategy that plays $\chi(u)$ in $\mathcal{A}'$ instead of u in $\mathcal{A}$. In Figure 3 we have pictorially represented which parts of u determine which parts of $\chi(u)$.

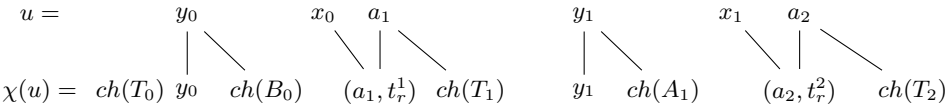


Fig. 3. Definition of $\chi(u)$

The next lemma follows directly from the definition of the reduction from $\mathcal{A}$ to $\mathcal{A}'$.

Lemma 3. *If u ends in a letter from $\Sigma_{q,r}$ then we have the following*

- $state_q(\chi(u)) = \langle state_q(u), state_r(u) \rangle$.
- $state_p(\chi(u)y) = state_p(uy)$ *for every* $p \neq q$ *and* $y \in (\Sigma \setminus \Sigma_{q,r})^*$.
- $state_q(\chi(u) \, ch(T) y) = \langle state_q(uy), T \rangle$ *for every* $y \in (\Sigma \setminus \Sigma_{q,r})^*$.
- $state_q(\chi(u) \, ch(T) y \, ch(B)) = \langle state_q(uy), T, B \rangle$ *for every* $y \in (\Sigma \setminus \Sigma_{q,r})^*$.

From σ in $\mathcal{A}$ to σ' in $\mathcal{A}'$. We are now ready to define σ' from a winning strategy σ. We assume that σ satisfies the property stated in Lemma 1. We will define σ' only for certain plays and then show that this is sufficient.

Consider u' such that $u' = \chi(u)$ for some σ-play u ending in a letter from $\Sigma_{q,r}$. We have:

– If $state_q(u') \notin F_q$ then $\sigma'_q(view_q(u')) = \{ch(T)\}$ where $T = Sync^{\sigma}_r(u)$.
– For every process $p \neq q$ we put $\sigma'_p(view_p(u'\, ch(T)y)) = \sigma_p(view_p(uy))$ for $y \in (\Sigma \setminus \Sigma_{q,r})^*$.
– For $y \in (\Sigma \setminus \Sigma_{q,r})^*$ and $B = \sigma_q(view_q(uy))$ we define

$$\sigma'_q(view_q(u'\, ch(T)y)) = \begin{cases} B & \text{if } B \cap \Sigma_{q,r} = \emptyset \\ \{ch(B)\} & \text{if } B \subseteq \Sigma_{q,r} \end{cases}$$

– $\sigma'_q(view_q(u'\, ch(T)y\, ch(B))) = \emptyset$.

Observe that in the last case the strategy proposes no move as there are only moves of the environment from a position reached on a play of this form.

The next lemma states the correctness of the construction.

Lemma 4. *If σ is a winning strategy for $\mathcal{A}, (F_p)_{p\in\mathbb{P}}$ then σ' is a winning strategy for $\mathcal{A}', (F'_p)_{p\in\mathbb{P}'}$.*

Proof. We will show inductively that for every σ'-play u' ending in a letter of the form (a', t'_r) there is a σ-play u such that $u' = \chi(u)$. Then we will show that every maximal σ'-play is winning.

We start with the induction step, later we will explain how to do the induction base. Let us take $u' = \chi(u)$ as in the induction hypothesis. By Lemma 3 we have $state_q(u') = \langle state_q(u), state_r(u)\rangle$.

Consider a possible, σ'-compatible, extension of u' till the next letter (a, t_r). It is of the form $u'\, ch(T)y\, ch(B)(a, t_r)$ where $y \in (\Sigma \setminus \Sigma_r)^*$. We will show that it is of the form $\chi(uyxa)$ for some $x \in (\Sigma^{loc}_r)^*$, and that $uyxa$ is a σ-play.

– By definition of the automaton $\mathcal{A}'$ and the strategy σ' we have $\sigma'(u') = \{ch(T)\}$ with $T = Sync^{\sigma}_r(u)$.
– Since σ' is the same as σ on actions from $\Sigma \setminus \Sigma_r$, we get that uy is a σ-play.
– Concerning $ch(B)$, by the definition of σ' we have that $B = \sigma_q(view_q(uy))$. Then by the definition of $\mathcal{A}'$ we get some A such that $(t_r, A) \in T$, and $a \in (t_r, A) \bowtie (s_q, B)$ with $s_q = state_q(uy)$. As $T = Sync^{\sigma}_r(u)$ we can find $x \in (\Sigma^{loc}_r)^*$ such that ux is a σ-play, $state_r(ux) = t_r$ and $\sigma_r(ux) = A$. We get that $uyxa$ is a σ-play with $\chi(uyxa) = u'\, ch(T)y\, ch(B)(a, t_r)$, and we are done.

The induction base is exactly the same as the induction step taking u' and u to be the empty sequence.

To finish the lemma we need to show that every maximal σ'-play is winning. For this we examine all possible situations where such a play can end. We consider plays u' and u as at the beginning of the lemma.

If u' itself is maximal then $state_q(u')$ is final because otherwise $ch(T)$ would be possible. Hence, by Lemma 3 $state_q(u)$ and $state_r(u)$ are final. Since σ and σ' are the same on processes other than q and r, no action a with $dom(a) \cap \{q, r\} = \emptyset$ is possible from u. It follows that u is a maximal σ-play. Since σ is winning, $state_p(u)$ is final for every process p. By Lemma 3, u' is winning too.

Suppose now that $u'\,ch(T)y$ is maximal for some $y \in (\Sigma \setminus \Sigma_r)^*$. By the same reasoning as above there is no σ-play extending uy by an action from $\Sigma \setminus \Sigma_r$. We have two cases

- If $state_q(uy)$ is final then T is a final plan by Lemma 2. So there is $x \in (\Sigma_r^{loc})^*$ such that $state_r(uyx)$ is final. Then uyx is a maximal σ-play. Since σ is winning, after uyx all processes are in the final state. By Lemma 3, $u'\,ch(T)y$ is winning too.
- If $state_q(uy)$ is not final then $\sigma(uy) \subseteq \Sigma_{q,r} \neq \emptyset$ since σ is assumed to satisfy Lemma 1, and communication with other processes than r is not possible. By Lemma 2 T cannot be final and action $ch(B)$ for $B = \sigma(uy)$ is possible according to σ'. A contradiction.

A play of the form $u'ch(T)y\,ch(B)$ cannot be maximal since some local actions of the form (a, t_r) are always possible. This covers all the cases and completes the proof. □

From σ' in $\mathcal{A}'$ to σ in $\mathcal{A}$. From a strategy $\sigma' = (\sigma'_p)_{p\in\mathbb{P}'}$ for $\mathcal{A}'$ we define a strategy $\sigma = (\sigma_p)_{p\in\mathbb{P}}$ for $\mathcal{A}$. We assume that σ' satisfies Lemma 1. We consider u ending in an action from $\Sigma_{q,r}$ such that $\chi(u)$ is a σ'-play. First, for every $p \neq q, r$ and every $y \in (\Sigma \setminus \Sigma_r)^*$ we set

$$\sigma_p(view_p(uy)) = \sigma'_p(view_p(\chi(u)y)).$$

If $state_q(\chi(u))$ is not final then $\sigma'(\chi(u)) = \{ch(T)\}$ for some admissible plan T in state $state_r(\chi(u))$. This means that $T = Sync_r^\rho(u)$ for some strategy ρ. In this case:

- for every $x \in (\Sigma_r^{loc})^*$ we set $\sigma_r(ux) = \rho_r(ux)$;
- for every $y \in (\Sigma \setminus \Sigma_r)^*$ we consider $X = \sigma'_q(view_q(\chi(u)\,ch(T)y))$ and set

$$\sigma_q(view_q(uy)) = \begin{cases} B & \text{if } X = \{ch(B)\} \\ X & \text{otherwise} \end{cases}$$

Lemma 5. *If σ' is a winning strategy for $\mathcal{A}', (F'_p)_{p\in\mathbb{P}'}$ then σ is a winning strategy for $\mathcal{A}, (F_p)_{p\in\mathbb{P}}$.*

Proof. Suppose that u is σ-play ending in an action from $\Sigma_{q,r}$ and such that $\chi(u)$ is a σ'-play. We first show that for every extension of u to a σ-play $uyxa$ with $y \in (\Sigma \setminus \Sigma_r)^*$, $x \in (\Sigma_r^{loc})^*$, and $a \in \Sigma_{q,r}$, its image $\chi(uyxa)$ is a σ'-play. Then we will show that every maximal σ-play is winning.

Take $uyxa$. By Lemma 3 $state_q(\chi(u))$ is not final, so we have $\sigma'(\chi(u)) = \{ch(T)\}$. Then $T = Sync_r^\sigma(u)$ by the definition of σ. Again directly from the definition we have that $\chi(u)\,ch(T)y$ is a σ'-play. By definition of σ we have then that $\chi(u)\,ch(T)y\,ch(B)$ is a σ'-play for $B = \sigma_q(view_q(uy))$. Finally, we need to see why (a, t_r) with $t_r = state_r(ux)$ is possible. Since $T = Sync_r^\sigma(u)$ we get

that $(t_r, \sigma_r(view_r(ux))) \in T$. Then $a \in \sigma_r(view_r(ux)) \cap B$, and in consequence $\chi(u)\, ch(T) y\, ch(B)(a, t_r)$ is possible by Lemma 3 and the definition of $\mathcal{A}'$.

It remains to verify that every maximal σ-play is winning. Consider a maximal σ-play uyx where u ends in an action from $\Sigma_{q,r}$, $x \in (\Sigma_r^{loc})^*$, and $y \in (\Sigma \setminus \Sigma_r)^*$ (this includes the cases when x, or y are empty). We look at $\chi(u)$ and consider two situations:

- If no $ch(T)$ is possible from $\chi(u)$ then $state_q(\chi(u))$ is final. This means that x is empty and $state_q(u)$ and $state_r(u)$ are both final. It is then clear that $\chi(u)y$ is a maximal σ'-play. Since σ' is winning, every process is in a final state. So uy is a winning play in $\mathcal{A}$.
- If $\chi(u)\, ch(T)$ is a σ'-play for some T then again we have two cases:
  - If $s_r = state_r(uyx)$ is final then $(s_r, \emptyset) \in T$ by the definition of σ. As T is an admissible plan, T is final. After $\chi(u)y$ no action other than $ch(B)$ is possible. But $ch(B)$ is not possible either since T is final. Hence $\chi(u)y$ is a maximal σ'-play. So all the states reached on $\chi(u)y$ are final. By Lemma 3 we deduce the same for uyx, hence uyx is winning.
  - If s_r is not final then $\sigma_r(view_r(uyx)) = A \subseteq \Sigma_{q,r}$ for $A \neq \emptyset$ (local actions of r are not possible, since uyx is maximal). Hence $(s_r, A) \in T$, and T is not final. This means that $s_q = state_q(\chi(u)\, ch(T)y)$ is not final. So it is possible to extend the σ'-play with an action of the form $ch(B)$. But by the definition of $\mathcal{A}'$ we have $(s_q, B) \bowtie (s_r, A) \neq \emptyset$. Hence uyx can be extended by a communication between q and r on a letter from $B \cap A$; a contradiction.

□

Together, Lemmas 4 and 5 show Theorem 3.

Theorem 3. *Let r be the fixed leaf process with $\mathbb{P}' = \mathbb{P} \setminus \{r\}$ and q its parent. Then the system has a winning strategy for $\mathcal{A}, (F_p)_{p\in\mathbb{P}}$ iff it has one for $\mathcal{A}', (F'_p)_{p\in\mathbb{P}'}$. All the components of $\mathcal{A}'$ are identical to those of $\mathcal{A}$, apart that for the process q. The size of q in $\mathcal{A}'$ is $\mathcal{O}(M_q 2^{M_r|\Sigma_{qr}|})$, where M_q and M_r are the sizes of processes q and r 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 $\mathcal{P}(\Sigma_{r,\ell})$. 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.

Theorem 4. *The control problem for distributed alphabets with acyclic communication graph is decidable. There is an algorithm for solving the problem (and computing a finite-state controller, if it exists) whose working time is bounded by a tower of exponentials of height equal to half of the diameter of the graph.*

Our reduction algorithm can be actually used to compute a (finite-state) distributed controller:

Corollary 1. *There is an algorithm which solves the control problem for distributed alphabets whose communication graph is acyclic and if the answer is positive, the algorithm outputs a controller satisfying the following property: For every process p and every state s of the controller $\mathcal{A}_c$, the set of actions allowed for process p in state s is the set of all uncontrollable local actions plus:*

- *either a unique controllable local action,*
- *or a set of controllable actions shared with a unique neighbour q of p.*

4 The lower bound

We show in this section that in the simplest non-trivial case of acyclic communication graphs, consisting of a line of three processes, the control problem is already EXPTIME-complete. In the general case the complexity of the control problem grows as a tower of exponentials function with respect to the size of the diameter of the communication graph.

4.1 Height one

Proposition 1. *The control problem for the communication graph* $1 \text{——} 2 \text{——} 3$ *is* EXPTIME*-complete.*

Proof. The EXPTIME upper bound follows from Theorem 3, as the height of the tree is 1. So the reduction is applied twice from process 2, first simulating process 1, then simulating process 3. Finally, a reachability game is solved on an exponential size arena.

For the lower bound we simulate an alternating polynomial space Turing machine M on input w. We assume that M has a unique accepting, blocking configuration (say with blank tape, head leftmost). The goal now is to let processes $1, 3$ guess an accepting computation tree of M on w. The environment will be able to choose a branch in this tree and challenge each proposed configuration. Process 2 will be used to validate tests initiated by the environment. If a test reveals an inconsistency, process 2 blocks and the environment wins. To summarize the idea of the construction: processes 1 and 3 generate sequences of configurations (encoded by local actions), separated by action \$ and $\overline{\$}$, respectively, shared with process 2. Both start with the initial configuration of M on w. Transitions from existential states are chosen by the plant, and those from universal ones by the environment. At a given time, process 1 has generated

the same number of configurations is process 3, or process 3 is about generating one configuration more. In the first case, the environment can check that it is the same configuration; and in the second, it can check that it is the successor configuration. In this way, 1 and 3 need to generate the same branch of the run tree.

A computation of M with space bound n is a sequence $C_0 \vdash C_1 \vdash \cdots \vdash C_N$, where each configuration C_i is encoded as a word from $\Gamma^*(Q \times \Gamma)\Gamma^*$ of length n. Since M is alternating, its acceptance is expressed by the existence of a tree of accepting computations.

Processes 1 starts by generating the initial configuration on w, followed by a synchronization symbol \$ with process 2. After this, process 1 generates a sequence of configurations separated by \$. When generating a configuration, process 1 remembers M's state q and the symbol A under the head. All transitions so far are controllable. After generating \$ process 1 goes into a state where the outgoing transitions are labeled by M's transitions on (q, A) (if the configuration was not blocking). These transitions are controllable if q is existential, and uncontrollable if q is universal. The transition chosen, either by the plant or the environment, is stored in the state up to the next synchronization symbol. Finally, if the current configuration is final then process 1 synchronizes with 2 on $\$_F$ (instead of \$) and goes into an accepting state.

The description is similar for process 3, with $\overline{\Gamma}, \overline{Q}, \overline{\$}, \overline{\$}_F$ instead of $\Gamma, Q, \$, \$_F$. Finally, process 2 has two main states, *eq* and *succ*, with transitions $eq \xrightarrow{\overline{\$}} succ$ and $succ \xrightarrow{\$} eq$. From state *eq* it can go to an accepting state after reading $\overline{\$}_F$ followed by $\$_F$.

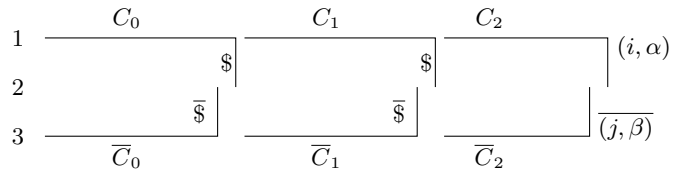


Fig. 4. Environment chooses positions i, j in $C_P, \overline{C}_P$ with $P = 2$. System wins iff $\alpha = \beta$ or $i \neq j$.

The environment can initiate 2 kinds of tests: equality and successor test. The equality test checks that $C_P = \overline{C}_P$ and the successor test checks that $C_P \vdash \overline{C}_{P+1}$.

For the equality test, the environment can choose a position i within C_P and a position j in $\overline{C}_P$. Formally, for each (controllable) outgoing transition $s \xrightarrow{\alpha}$ of process 1 with $\alpha \in \Gamma \cup (Q \times \Gamma)$ there is a transition $s \xrightarrow{(\downarrow,\alpha)} (\downarrow, i, \alpha)$ with $(\downarrow, \alpha)$ uncontrollable. The target state $(\downarrow, i, \alpha)$ records the tape position i (known from s) and the tape symbol α. In state $(\downarrow, i, \alpha)$ process 1 synchronizes with 2 on action $(\downarrow, i, \alpha)$, and then stops (accepting). The same for process 3 with uncontrollable actions $\overline{(\downarrow, \beta)}$, and synchronization action $\overline{(\downarrow, j, \beta)}$.

From state *eq* process 2 can perform a synchronization $\overline{(\downarrow, j, \beta)}$ with process 3 and then one with process 1 on any $(\downarrow, i, \alpha)$, provided $i \neq j$ or $\alpha = \beta$, and then accept. This is the case where the environment has chosen positions on both lines 1 and 3 (see Figure 4). If the environment has chosen a test transition in C_P but not in $\overline{C}_P$ (or vice-versa), process 2 will accept (and stop), too.The successor test is similar.

The successor test is similar, it consists in choosing a position within C_P and one within $\overline{C}_{P+1}$. The information checked by process 2 includes the symbols $\alpha_-, \alpha, \alpha_+$ of C_P at positions $i-1, i, i+1$ resp., so process 1 goes on transition $\overline{(\searrow, \alpha)}$ into a state of the form $(i, \alpha, \alpha_-, \alpha_+)$. In state $\bar{t}$ process 2 can perform a synchronization on $(\searrow, i, \alpha, \alpha_-, \alpha_+)$ with process 1, and then one with process 3 on $\overline{(\searrow, j, \beta)}$, provided $i \neq j$ or the symbols $\alpha_-, \alpha, \alpha_+$ are inconsistent with the new middle symbol β according to M's transition relation.

The reader may notice that we need to guarantee that the universal transitions chosen by the environment are the same, for processes 1 and 3. This can be enforced by communicating the transitions with actions $\$, \overline{\$}$ to process 2, who is in charge of checking. Moreover, note that the action alphabet above is not constant, in particular it depends on n. This can be fixed by replacing each action of type $(\downarrow, i, \alpha)$ (or alike) by a sequence of synchronization actions where i is transmitted bitwise. By alternating the bits transmitted by 1 and 3, respectively, process 2 can still compare indices i, j.

Note also that configurations $C_P, \overline{C}_P$ are generated in parallel, and so are C_P and $\overline{C}_{P+1}$. This is crucial for the correctness. □

Lemma 6. *The control problem defined in Proposition 1 has a winning strategy if and only if M accepts w.*

Proof. We assume that there is a winning strategy in the control game. Let us consider a maximal winning play without tests, where process 1 generates $C_0\$C_1\$\cdots C_N\$_F$ and process 3 generates $\overline{C}_0\overline{\$}\overline{C}_1\overline{\$}\cdots\overline{C}_{N'}\overline{\$}_F$. By construction, each of the C_p and $\overline{C}_q$ are configurations of length n, $C_0 = \overline{C}_0$ is the initial configuration of M on w, and $C_N = \overline{C}_{N'}$ is the accepting configuration. Suppose by contradiction that $C_0, \ldots, C_N$ is not a run of M. Assume first that $C_p = \overline{C}_p$ for all $0 \leq p < P$, but $C_{P-1} \not\vdash \overline{C}_P$. In this case the environment could have chosen the first position i where $\overline{C}_P$ does not correspond to a successor of C_{P-1}, and process 2 would have rejected after the synchronization $(\searrow, i, \alpha, \alpha_-, \alpha_+)$ followed by $\overline{(\searrow, i, \beta)}$, contradicting the fact that the strategy is winning. The second case is where $C_p = \overline{C}_p$ for all $0 \leq p < P$, but $C_P \neq \overline{C}_P$. Then the environment could have chosen the first position i where C_P and $\overline{C}_P$ differ, and process 2 would have rejected after the synchronization $\overline{(\downarrow, i, \beta)}$ followed by $(\downarrow, i, \alpha)$ with $\alpha \neq \beta$, again a contradiction. This means that $C_0 \vdash C_1 \vdash \cdots C_N$. Moreover, $C_N = \overline{C}_N$ is final since process 1 is in a final state (thus also $N = N'$).

For the converse, we assume that M accepts w. Let the strategy of processes 1 and 3 consist of generating an accepting run tree of M on w. For existential configurations, say that both 1 and 3 choose the first winning transition among all possibilities. Every maximal play without environment test corresponds to

an accepting run $C_0 \vdash C_1 \vdash \cdots C_N$, hence the play reaches a final state on every process. Every maximal play with test is of one of the following forms: (1) $C_0\overline{C}_0\overline{\$}\$\cdots C_{P-1}\overline{C}_{P-1}\overline{\$}\$xy$, where x and y are prefixes of C_P and $\overline{C}_P$, followed by $\downarrow$-actions, or (2) $C_0\overline{C}_0\$\$\cdots\overline{C}_{P-1}\overline{\$}xy$, where x is prefix of C_{P-1} and y a prefix of $\overline{C}_P$, followed by $\searrow$-actions. In both cases, the environment's challenge fails, since $C_P = \overline{C}_P$ and $C_{P-1} \vdash \overline{C}_P$. □

4.2 Lower bound: general case

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 $\mathit{Tower}(2, l)$ (with $\mathit{Tower}(n, l) = 2^{\mathit{Tower}(n,l-1)}$ and $\mathit{Tower}(n, 1) = n$). Then an easy adaptation of the construction will allow to code computations of Turing machines with the same space bound as the capabilities of counters.

We fix n and will be first interested to define n-counters. 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. Following this intuition we write $(1 - c)$ to denote b if $c = a$ and vice versa.

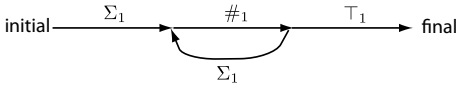
An $(l + 1)$-*counter* is a word

$$x_0u_0x_1u_1 \cdots x_{k-1}u_{k-1}\#_{l+1} \tag{1}$$

where $k = \mathit{Tower}(2, l)$ and for every i, letter $x_i \in \Sigma_{l+1}$ 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 $\#_{l+1}$ will be convenient in the construction that follows. An *iterated* $(l + 1)$-*counter* is a nonempty sequence of $(l + 1)$-counters.

For every l we will define a plant $\mathcal{C}^l$ such that the winning strategy for the system in $\mathcal{C}^l$ will need to produce an iterated l-counter.

For $l = 1$ this is very easy, we have only one process in $\mathcal{C}^1$ and all transitions are controllable.



This automaton can repeatedly produce a 1-counter and eventually go to the accepting state. The letter on which it goes to accepting state will be not important, so we put $\top_1$. Recall that our acceptance condition is that all processes reach a final state from which no actions are possible.

Suppose that we have already constructed $\mathcal{C}^l$. We want now to define $\mathcal{C}^{l+1}$, a plant producing an iterated $(l+1)$-counter, i.e., a sequence of l-counters with values $0, 1, \ldots, (\mathit{Tower}(2, l) - 1), 0, 1, \ldots$. We assume that the communication graph of $\mathcal{C}^l$ has the distinguished root process r_l. Process r_l is in charge of generating an iterated l-counter. From $\mathcal{C}^l$ we will construct two plants $\mathcal{D}^l$ and $\overline{\mathcal{D}}^l$, over disjoint sets of processes. The plant $\mathcal{D}^l$ is obtained by adding a new root process r_{l+1}

that communicates with r_l, similarly for the plant $\overline{\mathcal{D}}^l$ with root process $\overline{r_{l+1}}$. The plant $\mathcal{C}^{l+1}$ will be the composition of $\mathcal{D}^l$ and $\overline{\mathcal{D}}^l$ with a new *verifier process* that we name $\mathcal{V}_{l+1}$. The root process of the communication graph of $\mathcal{C}^{l+1}$ will be r_{l+1}. The schema of the construction is presented in Figure 5. Process r_{l+1}, as well as $\overline{r_{l+1}}$, are in charge of generating an iterated $(l+1)$-counter. That they behave indeed this way is guaranteed by a construction similar to the one of Proposition 1, with the help of the verifier $\mathcal{V}_{l+1}$: the environment gets a chance of challenging each l-counter of the sequence of r_{l+1} (and similarly for $\overline{r_{l+1}}$). These challenges correspond to two types of tests, equality and successor. If there is an error in one of these sequences then the environment can place a challenge and win. Conversely, if there is no error no challenge of the environment can be successful; this means then that the sequences of l-counters have correct values $0, 1, \ldots, (\mathit{Tower}(2, l) - 1), 0, 1, \ldots$.

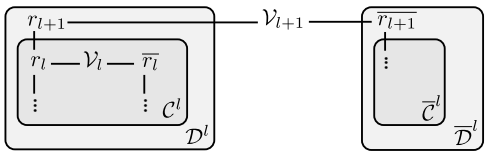


Fig. 5. Architecture of the plant $\mathcal{C}^{l+1}$

Construction of $\mathcal{D}^l$. The construction of the automaton of the new root r_{l+1} is presented in Figure 6.

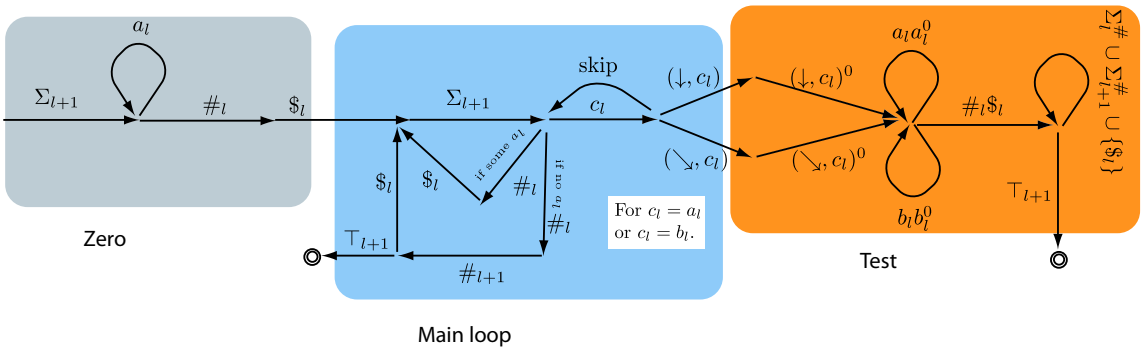


Fig. 6. Automaton for process r_{l+1}

We start by modifying the automaton for process r_l, given by $\mathcal{C}^l$. Actions of r_l from $\Sigma_l^{\#}$, that were previously local for r_l, become shared actions with r_{l+1}.

Process r_{l+1} has new local actions $\Sigma^{\#}_{l+1}$ and an action $\$_l$, shared with process $\mathcal{V}_{l+1}$. The action $\$_l$ is executed after each l-counter, that is, after each $\#_l$.

The automaton for r_{l+1} has two main tasks: it "copies" the sequence of l-counters generated by r_l (actually only the projection onto Σ_l) and it interacts with $\mathcal{V}_{l+1}$ towards the verification of this sequence. This automaton is composed of three parts that synchronize with r_l, forcing it to behave in some specific way. The first part called "zero" enforces that r_l starts with an l-counter with value 0 (otherwise r_{l+1} would block). When we read $\#_l$ we know that the first l-counter has ended and the control is passed to the second, main part of r_{l+1}.

The main part of r_{l+1} gives a possibility for the environment to enter into a test part. That is, after each transition on $c_l \in \Sigma_l$ (that is a_l or b_l) the environment chooses between action *skip* (that continues the main part) or a test action from $\{(\downarrow, c_l), (\searrow, c_l)\}$ that leads into the test part. The main part also outputs a local action $\#_{l+1}$ when needed, i.e., whenever the last seen l-counter was maximal. (Technically it means that there has been no a_l since the last $\#_l$.) The transition on $\#_{l+1}$ gives a possibility to go to the accepting state.

The test part of r_{l+1} simply receives the Σ_l-actions of r_l and sends them to process $\mathcal{V}_{l+1}$ (cf. loop $a_l a_l^0$ and $b_l b_l^0$). It does so until it receives $\#_l$ signaling the end of the counter. Then it sends $\$_l$ to process $\mathcal{V}_{l+1}$ to inform it that the counter has finished. After this r_{l+1} enters in a state where it can do any controllable action. From this state at any moment it can enter the accepting state on a dummy letter $\top_{l+1}$.

Plant $\overline{\mathcal{D}}^l$. This one is constructed in almost the same way as $\mathcal{D}^l$. Most importantly all processes (and actions) in $\overline{\mathcal{D}}^l$ are made disjoint from $\mathcal{D}^l$. We will write $\overline{a}$ for the letter of $\overline{\mathcal{D}}^l$ corresponding to a in $\mathcal{D}^l$.

The other difference between $\mathcal{D}^l$ and $\overline{\mathcal{D}}^l$ is that in the latter every transition $\overline{(\searrow, c)}$ is changed into $\overline{(\searrow, 1-c)}$ if since the last $\$_l$ there have been only $\overline{l_l}$. This is done to accommodate for the carry needed for the successor test. Recall that $(1-c)$ stands for a if c is b and vice versa.

Process $\mathcal{V}_{l+1}$. This process will have two main states *eq* and *succ*, the first one being initial. From *eq* there is a transition on $\overline{\$}_l$ to *succ*, and from *succ* there is a transition on $\$_l$ back to *eq*. Moreover from *eq* it is possible to go to the accepting state.

Additionally, from *eq* there is a transition on $\overline{(\downarrow, c)^0}$ to the state (eq, c) for every $c \in \Sigma_l$. Similar to the construction of Proposition 1, process $\mathcal{V}_{l+1}$ should accept if either the two bits from Σ_l challenged by the environment are compatible with the test, or their positions are unequal. So, from state (eq, c) on letter $(\downarrow, 1-c)^0$ there is a transition to a state called *neqtest*; on all other letters there is a transition to a looping state (see also Figure 7). Similarly from *succ*, but now with $(\searrow, c)$ letters, and the order of reading from the components reversed.

From state *neqtest* process $\mathcal{V}_{l+1}$ verifies that the sequence of actions Σ_l^0 initiated by r_{l+1} has not the same *length* as the sequence over $\overline{\Sigma}_l^0$ initiated by $\overline{r_{l+1}}$ (up to the moment where $\$_l^0$ and $\overline{\$}_l^0$ are executed). This is done simply by interleaving the two sequences of actions a_l^0, b_l^0, shared with r_{l+1} and $\overline{r_{l+1}}$,

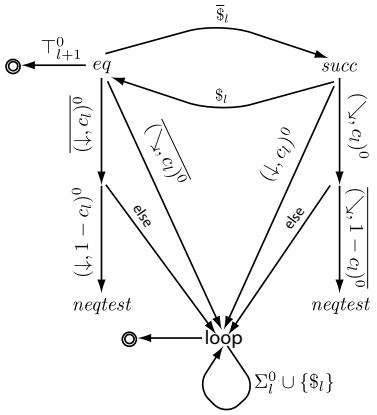


Fig. 7. Process $\mathcal{V}_{l+1}$.

respectively. Notice that the symbols a^0_l, b^0_l by themselves are not important, one could as well replace them by a single symbol. If this is the case, then process $\mathcal{V}_{l+1}$ gets to an accepting state, otherwise it rejects. In state *loop* process $\mathcal{V}_{l+1}$ can perform any controllable action and then enter the accepting state.

Putting together $\mathcal{C}^{l+1}$. The plant $\mathcal{C}^{l+1}$ is the composition of $\mathcal{D}^l$, $\overline{\mathcal{D}}^l$ and the new process $\mathcal{V}_{l+1}$. The actions of $\mathcal{C}^{l+1}$ are the ones of $\mathcal{C}^l$, plus $X \cup \overline{X}$ where X consists of:

- $\Sigma^{\#}_{l+1} \subseteq \Sigma^{sys}$ with domain $\{r_{l+1}\}$,
- $\Sigma^{\#}_{l} \subseteq \Sigma^{sys}$ with domain $\{r_l, r_{l+1}\}$,
- *skip* $\in \Sigma^{env}$ and $(\downarrow, c), (\searrow, c) \in \Sigma^{env}$ with domain $\{r_{l+1}\}$ $(c \in \Sigma_l)$,
- c^0, $\$_l$, $(\downarrow, c)^0$, and $(\searrow, c)^0$, all in Σ^{sys} with domain $\{r_{l+1}, \mathcal{V}_{l+1}\}$ $(c \in \Sigma_l)$.

The set $\overline{X}$ is defined similarly, by replacing every action c by $\overline{c}$, and r_l, r_{l+1} by $\overline{r_l}, \overline{r_{l+1}}$ in the domain of the action.

First we show that the system can indeed win every control instance $\mathcal{C}^l$. Moreover he can win and produce at the same time any iterated l-counter.

Lemma 7. *For every level l and every iterated l-counter $\mathfrak{c}$ there is a winning strategy σ in $\mathcal{C}^l$ such that for every σ-play the projection of this play on $\bigcup_{i=1,\ldots,l} \Sigma^{\#}_i$ is $\mathfrak{c}$.*

Proof. The proof is by induction on l. For $l = 1$ this is obvious since there are no environment moves and all possible behaviours leading to the accepting state are iterated 1-counters.

Let us consider level $l + 1$. Recall that $\mathcal{C}^{l+1}$ is constructed from $\mathcal{C}^l$, $\overline{\mathcal{C}}^l$, and three new processes: r_{l+1}, $\overline{r_{l+1}}$, $\mathcal{V}_{l+1}$. Fix an iterated $(l + 1)$-counter $\mathfrak{c}$. Observe that the projection of $\mathfrak{c}$ on the alphabet of l-counters, namely $\bigcup_{i=1,\ldots,l} \Sigma^{\#}_i$, is

an iterated l-counter. By induction we have a winning strategy producing this counter in $\mathcal{C}^l$. We play this winning strategy in the $\mathcal{C}^l$ and $\overline{\mathcal{C}}^l$ parts of $\mathcal{C}^{l+1}$. It remains to say what the new processes should do.

Process r_{l+1} should just produce $\mathfrak{c}$. By induction assumption we know that the letters this process reads from r_l are the projection of $\mathfrak{c}$ on the alphabet of the l-counter; and it is so no matter if there are environment questions in $\mathcal{C}^l$ or not. So process r_{l+1} has to just fill in missing Σ_{l+1} letters. If the environment asks no questions to r_{l+1} then at the end of $\mathfrak{c}$, this process will do $\#_{l+1}$, then $\top_{l+1}$ and enter the accepting state. Analogously for $\overline{r_{l+1}}$. At the same time process $\mathcal{V}_{l+1}$ will be at state *eq* and it can enter the accepting state, too, since it can count how many $\$_l$ symbols he has received.

Let us suppose now that the environment chooses a question action in r_{l+1} or $\overline{r_{l+1}}$. Let i be the index of an l-counter u_i within $\mathfrak{c}$ at which the first question is asked. We will consider two cases: (i) the question is asked in $\overline{r_{l+1}}$, (ii) the question is asked in r_{l+1} but not in $\overline{r_{l+1}}$.

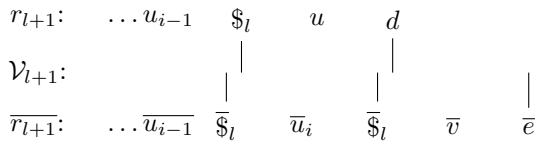
If a question is asked in $\overline{r_{l+1}}$ then the play has the following form:

$$\begin{array}{lccccc} r_{l+1}\colon & \ldots u_{i-1} & \$_l & u & & d \\ & & | & & & | \\ \mathcal{V}_{l+1}\colon & & | & & & | \\ \overline{r_{l+1}}\colon & \ldots \overline{u_{i-1}} & \overline{\$}_l & \overline{v} & & \overline{e} \end{array}$$

with $u, \overline{v}$ being prefixes of u_i; e being a question, and d a synchronization action of r_{l+1} with $\mathcal{V}_{l+1}$. So d can be a question or $\$_l$. Observe that after reading $\overline{\$_l}\$_l$ process $\mathcal{V}_{l+1}$ is in the state *eq*. It means that if the sequence $\overline{e}d$ is not $\overline{(\downarrow, c)}(\downarrow, 1-c)$ for some $c \in \Sigma_l$ then $\mathcal{V}_{l+1}$ enters state *loop*. From there it can calculate how many inputs from r_{l+1} and $\overline{r_{l+1}}$ it is going to receive. It receives them and then enters the accepting state. If $\overline{e}d$ is $\overline{(\downarrow, c)}(\downarrow, 1-c)$ then $\mathcal{V}_{l+1}$ enters state *neqtest*. Since r_{l+1} and $\overline{r_{l+1}}$ output the same iterated counter it must be that the questions are placed in different positions of the two counters. But then $\mathcal{V}_{l+1}$ will receive from the two processes a different number of Σ_l letters. Hence it will enter eventually into the accepting state also in this case.

Process $\overline{r_{l+1}}$ after receiving a question moves to a test component where it transmits the remaining part of the l-counter to $\mathcal{V}_{l+1}$ followed by $\overline{\$_l}$. Then it enters into the loop state of the test copy and can continue to generate $\mathfrak{c}$ since it can do any transition in this state. As for process r_{l+1}, if d is a question, then it does the same thing as $\overline{r_{l+1}}$. If d is $\$_l$ then r_{l+1} can continue to produce $\mathfrak{c}$, and both $\mathcal{V}_{l+1}$ and $\overline{r_{l+1}}$ can simulate their behaviour as if no question has occurred. If the environment asks a question to r_{l+1} at some moment, it too will enter into accepting state and continue to produce $\mathfrak{c}$.

If the first counter with a question is in r_{l+1} but not in $\overline{r_{l+1}}$ then the play has the form:



where u is a prefix of u_i, $\overline{v}$ a prefix of u_{i+1}, d is a question, and $\overline{e}$ a synchronization of $\overline{r_{l+1}}$ with $\mathcal{V}_{l+1}$. Observe that after reading $\$_l\overline{\$_l}$ process $\mathcal{V}_{l+1}$ is in state *succ*. As before our first goal is to show that $\mathcal{V}_{l+1}$ gets to an accepting state. If the sequence $d\overline{e}$ is not $(\searrow, c_l)\overline{(\searrow, 1-c_l)}$ then we reason as in the previous case. Otherwise $\mathcal{V}_{l+1}$ gets to state *neqtest*. As before we can deduce that the two questions are asked at different positions of the respective counters. Which means that $\mathcal{V}_{l+1}$ will receive a different number of Σ_l letters from r_{l+1} and $\overline{r_{l+1}}$ so it will get to state *loop*. The rest of the argument is exactly the same as in the previous case. □

We will show that in order to win in $\mathcal{C}^l$ the system has no other choice than to generate an iterated l-counter. Before this we present a general useful lemma:

Lemma 8. *Consider a plant $\mathcal{C}$ consisting of two plants $\mathcal{C}_1$ and $\mathcal{C}_2$ over process set $\mathbb{P}_1$ and $\mathbb{P}_2$, respectively. We assume that there exist $r_1 \in \mathbb{P}_1$ and $r_2 \in \mathbb{P}_2$ such that each action a in $\mathcal{C}$ is such that either $dom(a) \subseteq \mathbb{P}_1$ or $dom(a) \subseteq \mathbb{P}_2$, or $dom(a) \subseteq \{r_1, r_2\}$. Then every winning strategy in $\mathcal{C}$ gives a winning strategy in $\mathcal{C}_1$.*

Proof. Just fix the behaviour of the environment in $\mathcal{C}_2$ and play the strategy in $\mathcal{C}$. □

With this at hand we can now prove the main lemma.

Lemma 9. *If σ is a winning strategy in $\mathcal{C}^{l+1}$ and x is a σ-play with no question then the projection of x on $\bigcup_{i=1,\ldots,l+1} \Sigma_i^{\#}$ is an iterated $(l+1)$-counter.*

Proof. By the construction of $\mathcal{C}^{l+1}$, if there is no question during a σ-play, then the play is uniquely determined by the strategy. We will show that this unique play is an iterated $(l+1)$-counter.

By applying Lemma 8 twice we obtain from σ a winning strategy in $\mathcal{C}^l$. By induction assumption the projection of x on $\bigcup_{i=1,\ldots,l} \Sigma_i^{\#}$ is an iterated l-counter. Thus, between every two consecutive $\$_l$ we have a letter from Σ_{l+1}, followed by an l-counter and $\#_l$ (as long as we stay in the main part). The same holds for the $\overline{r_{l+1}}$ part. It remains to show that the sequence $u_0, u_1, \ldots$ of these l-counters represents the values $0, 1, \ldots$ modulo $Tower(2, l)$, and the same for the sequence $\overline{u_0}, \overline{u_1}, \ldots$

Assume that this is not the case and let i be the index where the first error occurs. We will construct a play winning for the environment.

Let us first assume that the value of $\overline{u_i}$ is correct but the one of u_i is not. Let k be the first position where the error occurs in the u_i counter. After the

k-th letter of u_i is transmitted to r_{l+1} the environment can execute action $(\downarrow, c)$. Similarly, in process $\overline{r_{l+1}}$ after the k-th letter the environment can execute $\overline{(\downarrow, 1-c)}$. Notice that these two questions are concurrent and happen after the letters of the corresponding counters are generated. Process $\mathcal{V}_{l+1}$ goes to *neqtest* since it receives $(\downarrow, c)$, and $\overline{(\downarrow, 1-c)}$. On the other levels the environment does not choose test actions. By induction, processes r_l and $\overline{r_l}$ will continue to generate iterated l-counters, since there are no questions in $\mathcal{C}^l$ and $\overline{\mathcal{C}^l}$. As the environment has chosen the same position k in both counters, process $\mathcal{V}_{l+1}$ will receive the same number of letters from r_{l+1} and $\overline{r_{l+1}}$ thus entering into a rejecting state. This contradicts the assumption that the strategy in $\mathcal{C}^{l+1}$ was winning.

The second case is where the value of u_i equals $i \pmod{\mathit{Tower}(2,l)}$, but the one of $\overline{u_{i+1}}$ is different from $(i+1) \pmod{\mathit{Tower}(2,l)}$. Let k be the position of the first error. In this case the environment can execute actions $\overline{(\searrow, c)}$, and $(\searrow, c)$ or $(\searrow, 1-c)$, depending on whether or not there is some a_l before position k in u_i. As in the case above, these two questions are concurrent because process $\mathcal{V}_{l+1}$ first synchronizes with $\overline{r_{l+1}}$ and then with r_{l+1}. The same argument as above shows that in this case we could find a play consistent with σ and winning for the environment. □

Putting Lemmas 7 and 9 together we obtain:

Proposition 2. *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 questions then its projection on $\bigcup_{i=1,\ldots,l} \Sigma_i^{\#}$ is an iterated l-counter.*

Theorem 5. *Let $l > 0$. There is an acyclic architecture of diameter $2l+1$ and with $(2^{l+3}-3)$ processes such that the space complexity of the control problem for it is $\Omega(\mathit{Tower}(n,l))$-complete.*

Proof. First observe that the plant $\mathcal{C}^l$ has $(2^{l+2}-3)$ processes and diameter $2l-1$. It is straightforward to make the l-counter count till $\mathit{Tower}(n,l)$ and not to $\mathit{Tower}(2,l)$ as we have done in the above construction. For this it is enough to make the 1-counter count to n instead of just to 2.

We will simulate space bounded Turing machines. Take a machine M and a word w of length n. We want to reduce the problem of deciding if w is accepted by M to the problem of deciding if the system has a winning strategy for a plant $\mathcal{C}(M,w)$ of size polynomial in the sizes of M and w.

A $\mathit{Tower}(n,l)$ size configuration can be encoded by an $(l+1)$-counter. In an iterated $(l+1)$-counter we can encode a sequence of such configurations. The plant $\mathcal{C}(M,w)$ is obtained by a rather straightforward modification of the construction of $\mathcal{C}^{l+1}$. Instead of ensuring that the value of the first counter is 0, it needs to ensure that it represents the initial configuration. Instead of ensuring that the two successive counters represent two successive numbers, it needs to ensure that they represent two successive configurations. Using Proposition 2, the problem of deciding if a $\mathit{Tower}(n,l)$-space bounded Turing machine M accepts w

is polynomially reducible to the problem of deciding if the system has a winning strategy in the so obtained $\mathcal{C}(M, w)$. The size of $\mathcal{C}(M, w)$ is exponential in l and polynomial in M, w, n. The game can be constructed in the time proportional to its size. □

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 a simple, yet powerful model based on synchronization using shared memory – as used in multithreaded programs or by hardware primitives such as compare-and-swap. Under some restrictions we have shown that the resulting control problem is decidable. Since every process is allowed to interact with the environment, our tree architectures are quite rich and allow to model hierarchical situations, like server/clients. Such cases are undecidable in the setting of Pnueli and Rosner.

Already Pnueli and Rosner in [17] 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. It is also interesting theoretically, since it allows to avoid simple (and unrealistic) reasons for undecidability. Our lower bound result is somehow surprising. Since we have full information sharing, all the complexity must be hidden in the uncertainty about other processes peforming in parallel.

Important problems remain open, in particular the decidability without the acyclic restriction. A more immediate task is to consider non-blocking winning conditions and Büchi specifications. A further interesting research venue is synthesis of open, concurrent recursive programs, as considered e.g. in [1].

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. R. M. Keller. Parallel program schemata and maximal parallelism I. Fundamental results. *Journal of the Association of Computing Machinery*, 20(3):514–537, 1973.
11. O. Kupferman and M. Vardi. Synthesizing distributed systems. In *LICS*, 2001.
12. P. Madhusudan and P. Thiagarajan. Distributed control and synthesis for local specifications. In *ICALP*, volume 2076 of *LNCS*, pages 396–407, 2001.
13. P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In *FSTTCS*, volume 3821 of *LNCS*, 2005.
14. A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.
15. P.-A. Melliès. Asynchronous games 2: The true concurrency of innocence. *TCS*, 358(2-3):200–228, 2006.
16. 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.
17. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP*, volume 372, pages 652–671, 1989.
18. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757, 1990.
19. P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(2):81–98, 1989.
20. S. Schewe and B. Finkbeiner. Synthesis of asynchronous systems. In *LOPSTR*, number 4407 in LNCS, pages 127–142. 2006.
21. A. Stefanescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *CONCUR*, number 2761 in LNCS, pages 27–41, 2003.
22. W. Zielonka. Notes on finite asynchronous automata. *RAIRO–Theoretical Informatics and Applications*, 21:99–135, 1987.