Synthesis with finite automata

Igor Walukiewicz

CNRS, LaBRI
Université de Bordeaux
351, Cours de la Libération
33 405, Talence cedex, France

2010 Mathematics Subject Classification: 68Q45,68Q17,03D05,68Q60,68Q85

Key words: Synthesis, Church Problem, Distributed Synthesis

Contents

1 Introduction

Synthesis is about constructing a system from a given specification. An example is construction of a circuit realizing a given boolean function. In this setting, the task is easy until one adds some constraints, for example on gate placement. In some other settings the task is algorithmically impossible from the very beginning: there is no algorithm constructing a program realizing the specified I/O function from an arithmetic formula. Yet, some severe restrictions of this problem are decidable, e.g., when considering only formulas of Presburger arithmetic. In this chapter we will be interested in an extension of the first example to infinite, reactive behavior.

We give an overview of some of the most important versions of the synthesis problem. In the first part of this chapter we present a pure formulation of the problem, proposed by Church. It asks to construct a single input/output device subject to a given specification. We then proceed to the richer setting of Ramadge and Wonham, where the objective is to find a controller for a given plant. These classical formulations have been extended in an overwhelming number of ways. From a multitude of possibilities we choose to focus on distributed synthesis: a promising and challenging direction for synthesis.

More than half a century ago, Church [24] formulated a synthesis problem for devices that transform an infinite sequence of input bits into an infinite sequence of output bits. Such a device is required to work "on-line": for each input bit read, it should produce an output bit. Church asked for an algorithm constructing such a device from a given specification. The specification language he considered is monadic second-order logic (MSOL) over the natural numbers with order, $\langle \mathbb{N}, \leqslant \rangle$. In this case, a specification is a formula $\varphi(X, Y)$, where X and Y stand for subsets of $\mathbb{N}$, or equivalently, infinite sequences of bits. So the formula defines a desired relation between the input sequence X and the output sequence Y.

The problem formulated by Church is fundamentally more difficult than decidability of the MSOL theory of natural numbers with order. Observe that the satisfiability of the formula $\forall X.\exists Y.\ \varphi(X, Y)$ is just a necessary condition for the existence of a required device: not only for every input sequence there should exist a good output sequence, but moreover this sequence should be produced "on-line". Indeed, while Büchi showed the decidability of the MSOL theory of $\langle \mathbb{N}, \leqslant \rangle$ in 1960 [15], the solution to the synthesis problem came almost a decade later [14, 55, 56].

Another interesting formulation of a synthesis problem was proposed by Ramadge and Wonham at the end of the 1980's [59, 40, 18]. It is based on concepts used in control theory. The formulation starts with a finite automaton, also called a plant, that defines all possible sequential behaviors of the system. The goal is to find, for a given plant, another finite automaton, called controller, such that the product of the two automata satisfies a given specification. Specifications, are usually MSOL properties of the language of the product. This formulation makes it possible to put restrictions not only on the controlled system, but also on the controller itself. For example, one can require that a controller does not block some actions or that it cannot observe some actions. We will present the basic formulation, as well as its extension permitting the handling of all regular specifications.

In a distributed system one can have multiple processes. The system specifies possible interactions between the processes and the environment, as well as, interactions between the processes themselves. The synthesis problem is to find a controller for each of the

processes such that the overall behavior of the system satisfies a given specification. The question studied most in this framework is a characterization of architectures and specifications for which the synthesis problem is decidable. Up to the present time, we know many undecidable cases, but only a few decidable ones. We will present these results, as well as some research directions.

2 Church problem

The Church problem is to construct a device that transforms an infinite sequence of input bits to an infinite sequence of output bits in a way given by a specification (cf. Figure 1). The device is required to work *on-line*, that is, for every bit read from the input, it should produce one bit on the output before reading the next input bit.

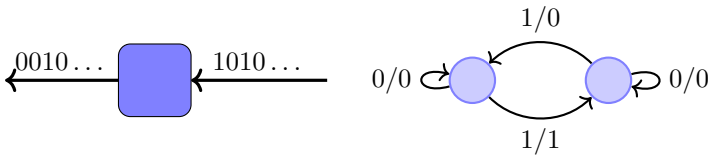


Figure 1. An I/O device, and its realization as an automaton with two states. It is more readable to have devices with input on the right since the first letter of a word is on the left.

Example: Suppose that we want that at every moment the number of 1's produced in the output is half of that read on the input. To be precise, let us say that there should be $\lfloor n/2 \rfloor$ of 1's in the output, where n is the number of 1's in the input. A possible device realizing this specification can simply copy input to the output, changing every second 1 to 0. A finite automaton, or one could say a transducer, doing this is presented in Figure 1; a label $1/0$ on a transition means that when reading 1 on the input the automaton produces 0 on the output.

Of course it is not difficult to find specifications that are not realizable by a finite automaton; consider, for example : "output 1 when the number of 1's on the input is a prime number".

Example: The specification from the first example is very special, because it describes exactly what the output at every time instance should be, as a function of the input that has been read. To see a different type of specification, consider the one saying "Infinitely many 1's on the input if and only if infinitely many 1's on the output". There are many ways to realize this specification. The device from the previous example realizes it, but probably the simplest solution is just to copy the input to the output.

Example: Sometimes specifications are easier to realize than it may seem at first; consider for example: "if infinitely many 1's in the input then finitely many 1's in the output". A simple way to realize this specification is to output only 0's. This makes the conclusion of the implication true, independently of the input. With this simple example we touch

an important difficulty in synthesis, at least in the Church style. If a specification is not sufficiently accurate, then we are very likely to get "strange" solutions.

Example: A variation on the previous example shows an instructive case when the problem has no solution. Consider a specification: "infinitely many 1's in the input if and only if finitely many 1's in the output". Suppose that there is a device realizing this specification. Let us look at its behavior when reading a sequence of 0's on the input. In order to realize the specification, the device should write 1 at some moment on the output. If not, then after reading an infinite sequence of 0's on the input the device would produce an infinite sequence of 0's on the output violating the specification. When the device produces 1 on the output, we put 1 on the input followed by another sequence of 0's. For the same reason as before, when reading this second sequence of 0's the device should at some moment output 1. At that point we put 1 on the input and restart sending 0's. This procedure is guaranteed to confuse our hypothetical device, since we will produce an input with infinitely many 1's on which the device outputs infinitely many 1's, contradicting the specification.

2.1 Specification formalisms

In order to state the Church problem algorithmically, we need to settle on a formal language for describing specifications. In the examples above, specifications were binary relations on infinite sequences over the alphabet $\{0,1\}$, or equivalently, sets of sequences over the alphabet $\{0,1\}^2$. It will be convenient to use two different formalisms for describing such objects: finite automata, and monadic second-order logic (MSOL).

Formally, an *infinite sequence* over an alphabet A is a function $w : \mathbb{N} \to A$. Often we will write it as $w_0 w_1 \cdots$, and call such a sequence an *infinite word*, where $w_i \in A$ are its letters. To describe a solution to the Church problem we will need to consider infinite trees. An *infinite binary labeled tree* over an alphabet A is a function $t : \{0,1\}^* \to A$. So the nodes are finite sequences over a two-letter alphabet, and each node is labeled with a letter from A. The empty word ε is the *root* of the tree, and every word w has the *left successor*, $w0$, and the *right successor*, $w1$.

Automata A (deterministic) *parity automaton* over an alphabet A is presented as a tuple $\langle Q, q^0, e : Q \times A \to Q, \Omega : Q \to \mathbb{N} \rangle$. The last component of the automaton is a function assigning a rank to every state: it is used to define acceptance. A *run* of the automaton over a word w is a sequence of states $q_0, q_1, \ldots$ such that $q_{i+1} = e(q_i, w_i)$. A run is *accepting* if the associated sequence of ranks $\Omega(q_0), \Omega(q_1), \ldots$ satisfies the *parity condition*:

$$\limsup_{i \to \infty} \Omega(q_i) \qquad \text{is even.}$$

In other words, a run is accepting if the largest rank appearing infinitely often in the run is even. Since Q is finite, this largest rank always exists. The *language of the automaton* is the set of words it accepts.

Example: Consider a language "all but finitely many positions are 0's" over an alphabet $A = \{0,1\}$. So the language consists of words with only finitely many occurrences of 1's. This language is recognized by an automaton having the set of states $Q = \{q_0, q_1\}$,

with $\Omega(q_i) = i$, and the transition function: $e(q_i, 0) = q_0$, $e(q_i, 1) = q_1$; for $i = 0, 1$. Since the transition function depends only on a letter that is read, it is not important what the initial state is. The run of the automaton on a word $w \in A^\omega$ reflects this word, in the sense that the $(i+1)$-st element of the run is q_1 if and only if $w_i = 1$. In consequence, the largest rank appearing infinitely often in the run over a word w is 1 if and only if there are infinitely many 1's in the word. So the word is accepted if and only if there are only finitely many occurrences of 1's in the sequence.

A (nondeterministic) *tree parity automaton* is a tuple

$$\langle Q, q^0, \delta : Q \times A \times Q \times Q, \Omega : Q \to \mathbb{N} \rangle \ .$$

The difference with the word case is that, instead of a transition function, we have a more complicated transition relation. A *run* of the automaton on a tree $t : \{0,1\}^* \to A$ is another labeled tree $r : \{0,1\}^* \to Q$ whose root is labeled with the initial state, $r(\varepsilon) = q^0$, and whose labeling respects the transition relation, $(r(w0), r(w1)) \in \delta(r(w), t(w))$. A run is accepting if and only if for every path of the tree the sequence of states appearing on the path satisfies the parity condition given by Ω.

Example: Consider a language "there is a path where all but finitely many positions are 0's" over the alphabet $A = \{0, 1\}$. It is recognized by an automaton having the set of states $Q = \{q_0, q_1, q_2\}$, with $\Omega(q_i) = i$, and the transition relation

$$\delta(q_2, j) = \{(q_2, q_2)\} \qquad \delta(q_i, j) = \{(q_j, q_2), (q_2, q_j)\} \quad i \in \{0,1\}, j \in \{0,1\}.$$

This means that starting from q_2 the automaton accepts every tree. In states q_0 and q_1, the automaton accepts one subtree by sending q_2 there, and sends q_j to the other subtree, where j is the letter read. So a run of the automaton is a tree with states q_0, q_1 on one path, and state q_2 everywhere else. The path with states q_0, q_1 satisfies the parity condition if there are infinitely many 1's on it. Hence, there is an accepting run of the automaton on a tree if and only if there is a path with finitely many 1's in the tree. Notice that nondeterminism is important here to single out such a path.

Monadic second-order logic If A has two letters, say $\{0, 1\}$, then a word $w : \mathbb{N} \to A$ can be identified with a subset of natural numbers: the positions in the word having the symbol 1. This set representation is very convenient for logic. We will write S_w to denote this subset of $\mathbb{N}$. To manipulate such sets we will need very little structure of the natural numbers; we will just take the successor relation as the only predicate:

$$\langle \mathbb{N}, \mathrm{succ} \rangle.$$

Monadic second order logic (MSOL) is an extension of first-order logic with quantification over sets of elements. For the structure $\langle \mathbb{N}, \mathrm{succ} \rangle$, first-order variables $x, y, \ldots$ range over natural numbers, and second-order variables $X, Y, \ldots$ over sets of natural numbers. The formulas are constructed using the binary relation symbol succ, standard Boolean operations, and quantification. Apart from these, formulas of monadic second-order logic admit a membership relation written in an infix notation ($z \in X$), and quantification over second-order variables. The semantics should be clear from the syntax and we omit it. For example, a formula

$$\forall X. \big[(y \in X) \wedge \forall z, z'. \, (z \in X \wedge \mathrm{succ}(z, z')) \Rightarrow (z' \in X) \big] \Rightarrow y' \in X$$

defines the inequality relation $y \leqslant y'$. Literally, the formula says that for every set of numbers X, if y is in X, and X is closed under taking successors, then y' is in X. The use of second-order quantification is crucial here. Using Ehrenfeucht-Fraïssé games, one can show that first-order logic cannot define the order from the successor relation.

We write $\langle \mathbb{N}, \mathrm{succ} \rangle, V \vDash \varphi$ to say that φ holds in the model $\langle \mathbb{N}, \mathrm{succ} \rangle$ under valuation V. Observe that V assigns natural numbers to first-order variables in φ and sets of numbers to set variables in φ. Thanks to identification of infinite sequences with sets of natural numbers, a formula $\varphi(Z)$ with one free second-order variable Z defines a set of sequences. For example,

$$\langle \mathbb{N}, \mathrm{succ} \rangle, V \vDash \exists x.\, \forall y.\, x \leqslant y \Rightarrow \neg Z(y)$$

holds if the valuation of Z, namely $V(Z)$, is a finite set. Thus the formula defines the set of words with finitely many 1's.

If $\varphi(Z)$ is a formula with a unique free second-order variable Z, and if $S \subseteq \mathbb{N}$, then we simply write $\langle \mathbb{N}, \mathrm{succ} \rangle \vDash \varphi(S)$ to say that $\varphi(Z)$ is true under a valuation that maps Z to S, i.e., when $\langle \mathbb{N}, \mathrm{succ} \rangle, V[Z \mapsto S] \vDash \varphi(Z)$. So, such a formula defines the set of sequences w satisfying $\langle \mathbb{N}, \mathrm{succ} \rangle \vDash \varphi(S_w)$.

To describe a solution to the Church problem we will need also monadic second-order logic on the infinite binary tree. Similarly to infinite sequences, this tree can be represented as a relational structure

$$\langle \{0,1\}^*, \mathrm{succ}_0, \mathrm{succ}_1 \rangle,$$

where succ_0 and succ_1 are the left and right successor relations.

The utility of working with subsets of the binary tree will be immediately visible if we observe that a device in the Church formulation of the synthesis problem is a function $f : \{0,1\}^+ \to \{0,1\}$. This means that a device can be identified with a subset of nodes of the binary tree: the set of nodes $w \in \{0,1\}^+$ with $f(w) = 1$. We will write S_f to denote this subset of $\{0,1\}^*$.

Thanks to identification of devices with subsets of the nodes of the binary tree, we can use monadic second-order logic on binary trees to talk about devices. The syntax of the logic is exactly the same as for monadic second-order logic over sequences, but now we have two successor relations. For example a formula

$$\begin{aligned} \mathit{path}(Y) \equiv\ & \varepsilon \in Y \wedge \\ & \forall x, x_0, x_1 (x \in Y \wedge \mathrm{succ}_0(x, x_0) \wedge \mathrm{succ}_1(x, x_1)) \Rightarrow (x_0 \in Y \vee x_1 \in Y) \end{aligned}$$

says that Y is a set containing at least one infinite path starting in the root of the tree.

Monadic second-order logic over sequences and trees is a well studied logic. For us the most important are decidability and regular witness properties. We say that S *is a regular subset of* $\langle \{0,1\}^*, \mathrm{succ}_0, \mathrm{succ}_1 \rangle$ if and only if S considered as a language over $\{0,1\}^*$ is a regular language, i.e., a language recognized by a finite automaton.

Theorem 2.1. *[15, 55] It is decidable if a given MSO formula holds in the sequence model* $\langle \mathbb{N}, \mathrm{succ} \rangle$. *Similarly for the tree model* $\langle \{0,1\}^*, \mathrm{succ}_0, \mathrm{succ}_1 \rangle$. *If a sentence* $\exists Z.\varphi(Z)$ *holds in* $\langle \{0,1\}^*, \mathrm{succ}_0, \mathrm{succ}_1 \rangle$ *then there is a regular set* S *such that* $\varphi(S)$ *holds in this model.*

2.2 Formulation of the problem

The formulation of the Church problem requires saying what is a specification and what are possible devices that we are looking for. A specification is a relation between input and output sequences, that is, a subset of $\{0,1\}^\omega \times \{0,1\}^\omega$. Recall that we have identified sequences $w \in \{0,1\}^\omega$ with subsets $S_w \subseteq \mathbb{N}$. Thanks to this identification, a relation between two sequences can be given by an MSOL formula $\varphi(X,Y)$ with two free set variables:

$$\{(v,w) : \langle \mathbb{N}, \mathrm{succ}\rangle \vDash \varphi(S_v, S_w)\}.$$

The form of devices is not restricted. This concretely means that a device can be an arbitrary function $f : \{0,1\}^+ \to \{0,1\}$. Intuitively, such a function determines the output bit, depending on the sequence of input bits received so far. For an input sequence $v \in \{0,1\}^\omega$ the *output sequence produced by* f is

$$w = \overline{f}(v) \qquad \text{with } w(i) = f(v(0)\cdots v(i)) \text{ for } i = 0,1,\ldots$$

This abstract definition clearly includes every physical device that bases its answers on the history of inputs, and it clearly excludes all kinds of devices with oracles or look-ahead. We say that f *satisfies a specification* given by $\varphi(X,Y)$ if for every input sequence $v \in \{0,1\}^\omega$, the formula $\varphi(S_v, S_{\overline{f}(v)})$ holds in $\langle \mathbb{N}, \mathrm{succ}\rangle$.

We now have all the ingredients to state the problem formally:

Definition 2.1 (Church synthesis problem). For a specification given as an MSOL formula $\varphi(X,Y)$, decide if there exists a device satisfying the specification, and construct one if it exists.

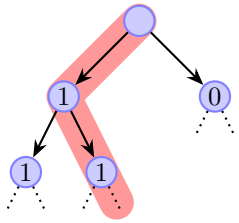
The second part of the formulation may seem rather unrealistic, since it is in general not clear how to construct an arbitrary function $f : \{0,1\}^* \to \{0,1\}$. Fortunately, it turns out that if the answer is positive then there is a device implementable as a finite automaton.

2.3 Solution to the problem

To solve the Church synthesis problem we use MSOL over the binary tree, since this logic is able to talk directly about the existence of a device with the desired properties. An infinite sequence $w \in \{0,1\}^\omega$ determines an infinite path in the tree, namely, the set of nodes that are finite prefixes of w. We denote this path by P_w. Recall that a device f can be encoded as a subset S_f of the binary tree (cf. page 1154). In this presentation, the output sequence $\overline{f}(w)$ is just the sequence of labels on P_w: i-th element of the sequence is 1 when the i-th node on the path, not counting the root, belongs to S_f (cf. Figure 2). Given a specification $\varphi(X,Y)$ as in the definition of the Church problem, it is not difficult to write a formula $\widehat{\varphi}(P,Z)$ over the binary tree such that for every sequence $w \in \{0,1\}^\omega$

> formula $\widehat{\varphi}(P_w, S_f)$ holds in the tree structure $\langle \{0,1\}, \mathrm{succ}_0, \mathrm{succ}_1\rangle$ if and only if formula $\varphi(S_w, S_{\overline{f}(w)})$ holds in the sequence structure $\langle \mathbb{N}, \mathrm{succ}\rangle$.

The later formula says that for the input sequence w, the output sequence $\overline{f}(w)$ satisfies the specification. Device f is a solution to the Church problem if and only if



$w = 01\ldots$
$\overline{f}(w) = 11\ldots$

Figure 2. Device f encoded in a tree

$\varphi(w, \overline{f}(w))$ holds for every infinite sequence w. Since infinite sequences correspond to paths in the binary tree, using the above equivalence, we can express it as the requirement that $\forall Y.\ path(Y) \Rightarrow \widehat{\varphi}(Y, S_f)$ holds in $\langle \{0,1\}^*, \mathrm{succ}_0, \mathrm{succ}_1 \rangle$. In consequence, the solution to the Church problem is equivalent to the satisfiability of the formula $\exists Z. \forall Y.\ path(Y) \Rightarrow \widehat{\varphi}(Y, Z)$ in the tree structure $\langle \{0,1\}^*, \mathrm{succ}_0, \mathrm{succ}_1 \rangle$. Indeed, the formula is satisfiable when there is an appropriate valuation for Z, and this valuation determines a device f. By Theorem 2.1 we can decide if the formula holds in $\langle \{0,1\}^*, \mathrm{succ}_0, \mathrm{succ}_1 \rangle$; and if it does then a regular witness for the meaning of Z can be constructed. This witness gives a device in a form of a finite automaton.

2.4 Synthesis via games

There exists another way of formulating the Church synthesis problem and its solution. One can view a specification as a game between a device and its environment: the environment provides inputs to the device and the device gives outputs. The winning condition is given by the specification, that is, by an MSOL formula. The advantage of this approach is that some automata theory permits us to greatly simplify the class of winning conditions: it is enough to consider parity conditions instead of all MSO properties.

The Church problem can be quite simply reduced to solving games with regular winning conditions. Indeed, it is sufficient to consider a game where repeatedly first Adam chooses a bit and then Eve responds with her choice of a bit (cf. Figure 3). If we have an instance of the Church problem given by an MSOL formula $\varphi(X, Y)$, then we consider the game where the winning plays for Eve are exactly those that satisfy the formula $\varphi(X, Y)$ (identifying plays with sequences of pairs of bits). It follows directly from the definitions that the Church problem given by $\varphi(X, Y)$ has a solution if and only if Eve has a winning strategy in the constructed game.

The above game has a rather complex winning condition. In the rest of this section we will present a reduction to games with parity conditions that have many good properties. We will start with a brief presentation of parity games.

Parity games A *game* is a graph with a partition of nodes between two players, called Eve and Adam, and a labeling defining the winning condition. Formally it is a tuple

$$\mathcal{G} = \langle V, V_E, V_A, T \subseteq V \times V, \Omega : V \to \mathbb{N} \rangle$$

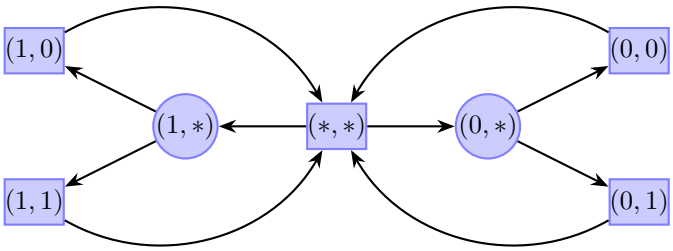


Figure 3. Reduction of the Church synthesis problem to a game. From rectangular vertices Adam chooses an input bit, and from circular vertices Eve responds with an output bit.

where (V_E, V_A) is a partition of the set of vertices V into those of Eve and Adam, T is the transition relation determining what are possible *successors* for each vertex, and Ω defines the *parity winning condition*. Since, unlike for automata, V is not necessarily finite, we should explicitly require that the there are finitely many ranks: the image of V under Ω should be finite.

A *play* between Eve and Adam from some vertex $v \in V = V_E \cup V_A$ proceeds as follows: if $v \in V_E$, then Eve makes a choice of a successor, otherwise Adam chooses a successor. From this successor the same rule applies and a play goes on forever unless one of the parties cannot make a move. The player who cannot make a move loses. The result of an infinite play is an infinite path $v_0 v_1 v_2 \cdots$ This *path is winning* for Eve if it satisfies the parity condition: the biggest rank appearing infinitely often in the sequence $\Omega(v_0), \Omega(v_1), \ldots$ is even. Otherwise Adam is the winner.

In the game presented in Figure 4, the positions of Eve are marked with circles and the positions of Adam with squares. Observe that the unique position with no successors belongs to Adam, so he loses there. Eve wins a play if it passes infinitely often through the position labeled with 2. For instance, if in the unique node for Eve she always chooses to move down then she wins, as 2 is on the loop. Actually Eve can also allow herself to move up, as then Adam has to go back to the position of rank 1 and Eve has once again a chance to move down. So as long as Eve moves infinitely often down, she sees 2 infinitely often and wins.

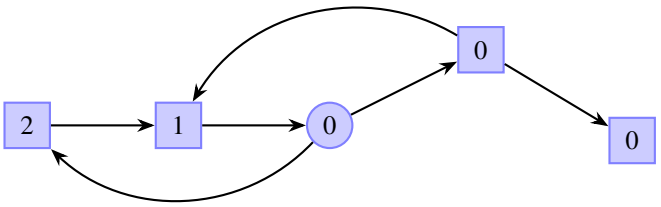


Figure 4. A parity game

A *strategy* for Eve is a function θ assigning, to every sequence of vertices $\vec{v}$ ending in a vertex v from V_E, a vertex $\theta(\vec{v})$ which is a successor of v. A *play respecting* θ is a sequence $v_0 v_1 \cdots$ such that $v_{i+1} = \theta(v_i)$ for all i with $v_i \in V_E$. The *strategy* θ *is winning for Eve* from a vertex v if all the plays starting in v and respecting θ are winning. A *vertex is winning* if there exists a strategy winning from it. The strategies for Adam are defined similarly. A strategy is *positional* if it does not depend on the sequence of vertices that were played until now, but only on the present vertex. So such a strategy can be represented as a function $\theta : V_E \to V$ and identified with a choice of edges in the graph of the game.

The main algorithmic question about parity games is to decide which of the two players has a winning strategy from a given position. In other words, to decide whether a given position is *winning for Eve* or *winning for Adam*. The principal results that we need about parity games are summarized in the following theorem. We refer the reader to [69] for more details.

Theorem 2.2 ([47, 30, 50]). *Every position of a parity game is winning for one of the two players. Moreover, a player has a positional strategy winning from each of his winning vertices. It is algorithmically decidable who is the winner from a given vertex in a finite game.*

Reduction to parity conditions The reduction of the Church problem to games with parity winning conditions is based on the following fundamental theorem relating models of MSO formulas and languages of deterministic parity automata. This is possible thanks to the representation of models of a formula $\varphi(X, Y)$ by infinite sequences over an alphabet $\{0,1\} \times \{0,1\}$.

Theorem 2.3 ([15, 49]). *For every MSOL formula $\varphi(X, Y)$, there is a deterministic parity automaton $\mathcal{A}_\varphi$ whose language is exactly the set of representations of the models satisfying the formula.*

The deterministic parity automaton $\mathcal{A}_\varphi$ given by the theorem provides a quick reduction of the Church synthesis problem to the problem of solving parity games. The automaton is over the alphabet $A = \{0,1\} \times \{0,1\}$. We transform it into a game where Adam guesses the first bit of the pair and Eve the second. The states of the game are $Q \cup (Q \times \{0,1\})$. From a position $q \in Q$ Adam can move to a position (q, b) for $b \in \{0,1\}$. From a position $(q, b) \in Q \times \{0,1\}$ Eve can move to $q' \in Q$ if there is a bit $b' \in \{0,1\}$ such that $q' = \delta(q, (b, b'))$, where δ is the transition function of $\mathcal{A}_\varphi$. The rank of a position is given by the rank function of the automaton: the rank of q as well as that of (q, b) is $\Omega(q)$. Hence, an infinite play is winning for Eve if and only if the chosen sequence of pairs of bits is accepted by $\mathcal{A}_\varphi$. This means that there is a strategy for Eve in this game if and only if the Church synthesis problem for the formula $\varphi(X, Y)$ has a solution.

To sum up, the notion of a parity game together with Theorems 2.2 and 2.3 provide another algorithmic solution to the Church problem.

2.5 Notes

Of course, the work on the centralized synthesis problem has a much more varied past and present than described in this section. For example, Kleene [39] talks about synthesizing automata from regular expressions. Later the synthesis problem was considered in the context of temporal logics [25, 46]. It has been even the motivation for introducing CTL (Computational Tree Logic).

In recent years the Church problem has been studied in a multitude of variants. More expressive specification formalisms have been explored. One possibility is to consider extensions of MSOL with monadic predicates [57, 37]. Another is to consider deterministic pushdown automata on infinite words [72], or even on higher-order collapsible pushdown automata [17, 13]. The problem becomes undecidable if we consider specifications given by nondeterministic pushdown automata with Büchi conditions.

Going a different way, one can consider the game formulation that lends itself to numerous extensions. It is possible to change the rules of the game so that the two players choose at the same time [21]. The games become not determined and naturally suggest the introduction of randomization both to moves of the game as well as to strategies [26, 22]. It is also possible to consider infinite arenas. One example is arenas defined by pushdown automata or their extensions, another is timed games [23, 10], and yet another is to consider vector addition systems or general well-quasi order conditions on the transition graph of the game [1]. Then there is also a vast literature on partial observation games, where players do not have complete information about the state of the game.

3 Control of discrete event systems

The Ramadge and Wonham set-up [59] offers a more detailed view of the synthesis problem than the Church formulation. It introduces new concepts allowing us to track down some challenging aspects of the problem. In particular, the idea of a split between a plant and a controller makes it possible to put additional requirements on the control. After presenting the standard Ramadge and Wonham setting, we discuss an extension to all regular specifications [58, 5]. We show how to formulate the Church problem in this extension.

3.1 Standard specifications

We start with principal concepts and a standard formulation of the problem. For this we do not need more than the basic theory of finite automata on finite words.

A *plant* over an alphabet A is a finite deterministic automaton without accepting states

$$\mathcal{A}_p = \langle Q_p, q_p^0, e_p : Q_p \times A \to Q_p \rangle.$$

Here Q_p is a finite set of states of the plant, q_p^0 is the initial state, and e_p is the transition function. The behavior of the plant is the language of $\mathcal{A}_p$ when all the states are considered accepting. We denote it by $L(\mathcal{A}_p)$. In other words, $L(\mathcal{A}_p)$ is the set of labels of paths in the graph of the automaton starting from the initial state. By definition $L(\mathcal{A}_p)$ is

prefix-closed.

A *controller* for $\mathcal{A}_p$ is another automaton $\mathcal{A}_c$ over the same alphabet as $\mathcal{A}_p$. A *controlled plant* is the product automaton $\mathcal{A}_p \times \mathcal{A}_c$

$$\mathcal{A}_p \times \mathcal{A}_c = \langle Q_p \times Q_c, (q_P^0, q_C^0), e_\times : (Q_p \times Q_c) \times A \to (Q_p \times Q_c) \rangle,$$

with $e_\times((q_p, q_c), a) = (q_p', q_c')$, where $q_p' = e_p(q_p, a)$ and $q_c' = e_c(q_c, a)$. Intuitively the role of the controller is to limit the set of possible behaviors, as only actions present in the controller are possible in the controlled plant.

The reason for having separated a plant and a controller is that we can now put some restrictions on the form of the controller. It is important to remark that these restrictions may not necessarily be visible in the product of a plant and a controller. Hence, being able to talk about the properties of a controller separately is something new and useful.

A first example of a restriction on a controller talks about *uncontrollable actions*. The idea is that a controller should not be able to forbid an uncontrollable action. Formally, the *controllability condition* with respect to a set $A_{\text{unc}} \subseteq A$ of uncontrollable actions is as follows

Controllability : From every state there is a transition on every $a \in A_{\text{unc}}$.

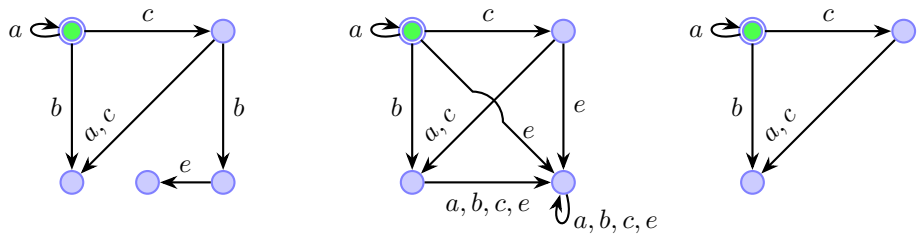


Figure 5. A plant, a controller, and the resulting controlled system. The set of uncontrollable actions is $A_{\text{unc}} = \{e\}$.

Example: Figure 5 shows a plant, a possible controller, and the resulting controlled plant. Assuming that $A_{\text{unc}} = \{e\}$, every state of the controller should have an outgoing transition on e. The control objective is $K = \{a, b, c\}^*$, or in other words, "avoid e". With these assumptions the presented controller is the maximal one. For example, it cannot permit doing the sequence cb from the upper-left state, as this would enable action e in the product.

It is somehow more elegant to formulate the Ramadge and Wonham control problem in terms of languages rather than automata. We call this synthesis problem *centralized* in anticipation of a more general, decentralized, problem that asks to construct several controllers at the same time.

Definition 3.1 (Centralized controller synthesis). Given a set $A_{\text{unc}} \subseteq A$ of uncontrollable actions, and prefix closed languages $P, K \subseteq A^*$, find the largest language $C \subseteq A^*$ with respect to inclusion such that $P \cap C \subseteq K$ and the two following conditions are satisfied:

prefix C is prefix closed;
control if $w \in C$ and $a \in A_{\text{unc}}$ then $wa \in C$.

The formulation asks for the largest controller, since the empty language is always a solution. The prefix closure requirement is equivalent to asking that a language be recognized by a finite automaton with all states accepting.

Theorem 3.1 ([59]). *There always exists the largest controller that is a solution to the centralized control problem. Moreover, this controller is a regular language, and can be constructed algorithmically.*

Proof. Take the minimal complete deterministic automata $\mathcal{A}_p$, $\mathcal{A}_k$ for languages P and K, respectively. Since both languages are prefix closed, both automata have only one non-accepting state that is also a sink state (a state from which all outgoing transitions are self-loops). We denote these states $\bot_p$ and $\bot_k$, respectively. Let us take the product automaton $\mathcal{A}_\times = \mathcal{A}_p \times \mathcal{A}_k$ and do the following:

- Introduce a new state $\top$, and make it recognize all A^*.
- Direct all transitions leading to states of the form $(\bot_p, s_k)$ to $\top$.
- Remove all states (s_p, s_k) from which a state $(s'_p, \bot_k)$ is reachable by a (possibly empty) sequence of uncontrollable actions.

Let $\mathcal{A}_C$ be the result of these operations and let $C = L(\mathcal{A}_c)$. We claim that C constructed this way is the solution announced by the theorem. The first observation gives a useful characterization of the behavior of the controlled plant.

Lemma 3.2. *$w \in P \cap C$ if and only if $w \in P \cap K$ and $wA^*_{\text{unc}} \cap P \subseteq wA^*_{\text{unc}} \cap K$.*

To see why this characterization holds, consider a word $w \in P \cap C$. By construction, after reading this word $\mathcal{A}_c$ ends up in a state (s_p, s_k) with none of its components being a sink state. This means that $w \in P \cap K$. To check the second condition, take $u \in A^*_{\text{unc}}$ and suppose $wu \in P$. This means that for the state (s'_p, s'_k) reached on this word in $\mathcal{A}_c$ the first component is not $\bot_p$. Observe that (s'_p, s'_k) is also the state reached on u from (s_p, s_k) on u. Since (s_p, s_k) is not removed, s'_k is not $\bot_k$. Hence $wu \in K$.

For the other direction of the Lemma 3.2, take $w \in P \cap K$. Automaton $\mathcal{A}_\times$ reading w reaches a state (s_p, s_k) with neither of the two components being a sink state. It is sufficient to check that this state is not removed in the second step of the construction.

Using Lemma 3.2 we can check that $C = L(\mathcal{A}_c)$ is a solution to the problem. The lemma implies that $P \cap C \subseteq K$. By construction C is prefix closed. It remains to check the control condition. For this we take $w \in C$ and $a \in A_{\text{unc}}$. We want to show $wa \in C$. There are two cases.

- If $wa \notin P$ then $wa = vbu$, where v is the longest prefix included in P. We have that in $\mathcal{A}_c$ word v leads to a state (s_p, s_k) with s_p not a sink state. By the construction a transition on b from this state leads to $\top_c$. Hence $vbA^* \subseteq C$, and in particular, $wa \in C$.
- The second case is when $wa \in P$. Then $w \in P \cap C$, so using Lemma 3.2 we get $wa \in K$. Moreover for every $u \in A_{\text{unc}}$ we have that if $wau \in P$ then $wau \in K$. Hence $wa \in C$ again by Lemma 3.2.

Finally, we check that C is the largest solution with respect to language inclusion. For this, consider some other controller C' and take $w \in C'$. We need to show $w \in C$. If $w \in P$ then $w \in P \cap C' \subseteq K$, since C' is supposed to be a controller for the specification K. Analogously, for every $u \in (A_{\mathrm{unc}})^*$, if $wu \in P$ then $wu \in K$, since $wu \in C'$ by the control condition. From Lemma 3.2 we get that $w \in C$. The remaining case is when $w \notin P$. Let v be the longest prefix of w that is in P, so $w = vbu$ for some letter b and word u. From the previous case we have $v \in P \cap C$. Since $vb \notin P$, we have that in $\mathcal{A}_c$ the word vb leads to $\top_c$. This means that $vbA^* \subseteq C$, and in particular $w = vbu \in C$. □

The interesting point of the above theorem is that it guarantees the existence of a maximal controller implementable as a finite automaton. The price to pay is rather severe limits on the form of a specification. In particular, the formulation does not allow expressing deadlock or liveness constraints. For example, the controlled plant from Figure 5 has deadlock states from which no transition is possible. In this example, a controller avoiding deadlocks should permit nothing but a actions.

The simplest approach to handle deadlock is to explicitly require that the resulting controlled plant does not have deadlock states.

Blocking : Every state of the controlled plant has an outgoing transition.

Observe that this time the condition refers to a controlled plant and not on a controller alone. Of course there are many different variants of the blocking condition. We take this one as a representative example.

It is not difficult to solve the control problem with a blocking requirement. We take the automaton $\mathcal{A}_c$ as constructed above. We call a state (p, k) *blocked* if there is no action enabled from it. Clearly, in order to satisfy the blocking condition we should remove all blocked states. We call a state (p, k) *unstable* if there is there is some uncontrollable action enabled in p and not enabled in (p, k). In $\mathcal{A}_c$ there are no unstable states, but once we remove blocked states we can get unstable states. Removing unstable states can produce new blocked or unstable states that should be removed. We repeat this process until there are no states to remove. The automaton we obtain at the end is the solution to the synthesis problem with blocking condition and it is the largest solution with respect to language inclusion.

Another important example concerns *unobservable actions*. These are actions whose execution by a plant should not be visible to a controller. In other words, transition labeled with an unobservable action should not change the state of the controller – it should be a self-loop. Formally, the *observability condition* with respect to a set $A_{\mathrm{uno}} \subseteq A$ of unobservable actions is as follows:

Observability : Every transition on $a \in A_{\mathrm{uno}}$ is a self-loop.

Example: Again consider the plant from Figure 5. Suppose that additionally c is unobservable; that is $A_{\mathrm{uno}} = \{c\}$, and as before $A_{\mathrm{unc}} = \{e\}$. If the specification is to avoid doing the action e then the largest controller for this plant has just one state with actions a, c, e being self-loops on this state. Put differently, the language of the controller permits all the sequences of actions not containing b.

In general, in order to solve the control problem with observability constraints, one first needs to perform a kind of powerset construction on the plant with respect to un-

observable actions. Then the rest of the argument is the same as in the previous cases. While conceptually easy in a centralized setting, the observability condition increases the algorithmic complexity of the problem. The constructions we have presented before were all in polynomial time. In particular, a controller has been always a sub-automaton of the product of the plant and the specification automata. A controller under observability constraints can be exponentially bigger than this product.

3.2 General specifications

In the previous subsection we have seen three types of constraints: controllability, blocking, and observability. One can, of course, very well imagine variations on these properties, as well as completely different properties. For example, liveness properties of the form: some action appears infinitely often on every execution. Or branching properties like: from every state it is possible to reach a reset state. In this subsection we will present a way to handle such extensions.

A specification in the centralized control problem talks about some properties of the product $\mathcal{A}_p \times \mathcal{A}_c$ of the plant and the controller. The most general way to specify such properties would be to talk directly about the product as a graph with labeled edges: states are nodes, and edges are given by the transition function. More precisely, an automaton $\langle Q, q_0, e : Q \times A \to A\rangle$ can be seen as a graph with labeled edges and a distinguished node $\langle Q, q_0, \{R_a\}_{a\in A}\rangle$ where $(q, q') \in R_a$ when $e(q, a) = q'$. To connect this to standard terminology we will call such graphs *transition systems*. Observe that these transition systems are *deterministic*: for every node and label there is at most one outgoing edge with that label.

Example: The transition system view of an automaton encourages formulation of properties not expressible in terms of language inclusion. For example, we may require that from every node one can reach a node where a transition on a reset action r is possible. In terms of properties of the language this means that every word in the language has a prolongation ending with r.

We are looking for a logic capable of describing properties of graphs with labeled edges. This time, though, we cannot just take monadic second-order logic as we have done for sequences, since the logic is undecidable over graphs. Fortunately, there exists a well-determined fragment of the logic that is decidable and can express most of the properties we are interested in.

Mu-calculus with loop testing Mu-calculus [4, 11, 71, 12] is a modal logic with fixpoints. A formula of this logic describes a set of states of a transition system. To define the syntax we fix an alphabet A of actions, that is, labels of edges of a transition system, and a countable set of *variables*, whose meanings will be sets of states of the transition system. The set of formulas of the logic is the smallest set containing variables, the constant $true$, closed under Boolean connectives, and two additional constructs:

modalities if α is a formula and a an action, then $\langle a\rangle\alpha$ is a formula;

fixpoint if α is a formula and X a variable whose all occurrences in α are positive (under even number of negations), then $\mu X.\alpha$ is a formula.

The meaning of a formula in a transition system is a set of states satisfying the formula. Since a formula may have free variables, its meaning depends on the meanings of the free variables. More formally, given a transition system $\mathcal{M} = \langle S, \{R_a\}_{a \in Act} \rangle$ and a valuation $\mathcal{V} : \mathrm{Var} \to \mathcal{P}(S)$ we define the meaning of a formula $[\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}}$ by induction on its structure. The meaning of variables is given by the valuation. The meaning of $true$ is the set of all the states of the transition system. The meaning of Boolean connectives is standard.

The meaning of modalities is determined by transitions. Formula $\langle a \rangle \alpha$ holds in all states from which there is an transition on a to a state satisfying α

$$[\![\langle a \rangle \alpha]\!]^{\mathcal{M}}_{\mathcal{V}} = \{s \in S : \exists s'. R_a(s, s') \wedge s' \in [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}}\}.$$

Finally, the μ construct is interpreted as the least fixpoint of an operator determined by α. A formula $\alpha(X)$ containing a free variable X can be seen as an operator on sets of states mapping a set S' to the semantics of α when X is interpreted as S'; in symbols: $S' \mapsto [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]}$. As all occurrences of X in α are positive, this operator is monotonic. Its least fixpoint is given by

$$[\![\mu X. \alpha]\!]^{\mathcal{M}}_{\mathcal{V}} = \bigcap \{S' \subseteq S : [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}[S'/X]} \subseteq S'\}.$$

We will often write $\mathcal{M}, s, \mathcal{V} \vDash \alpha$ instead of $s \in [\![\alpha]\!]^{\mathcal{M}}_{\mathcal{V}}$. Moreover we will omit $\mathcal{V}$ or $\mathcal{M}$ if it is not important or clear from the context.

Before giving some examples, let us introduce some useful abbreviations. We will write $[a]\alpha$ for $\neg\langle a \rangle \neg \alpha$. This formula holds in a state if α holds in all states reachable from it by transitions on a. We will write $\nu X. \alpha(X)$ for $\neg \mu X. \neg \alpha(\neg X)$. It can be checked that $\nu X. \alpha(X)$ is the greatest fixpoint of the operator defined by $\alpha(X)$.

Example: Formula $\langle a \rangle true$ means 'there is transition labeled by a'. With one fixpoint, we can talk about termination properties of paths in a transition system. The formula $\nu X. \langle a \rangle X$ means that there is an infinite sequence of a transitions. The formula $\mu X. [a] X$ means that all sequences of a transitions are finite. Observe the crucial role of fixpoints in the last two formulas; indeed changing μ to ν in the last formula gives $\nu X. [a] X$ that is always true since $[a] true$ is always true. With two fixpoints, we can write fairness formulas, such as $\nu Y. \mu Z. (\langle a \rangle Z) \vee \langle b \rangle Y$, meaning "there is a path of a's and b's with infinitely many occurrences of b's". A very useful formula $\nu X. (\langle b \rangle true) \wedge \bigwedge_{a \in A} [a] X$ says that action b is possible from every node reachable by a sequence of actions from A. Since A is the set of all actions, it means that action b is possible from every node reachable from the initial node. We write $\mathrm{Everywhere}(\gamma)$ for the same formula with γ replacing $\langle b \rangle true$. So $\mathrm{Everywhere}(\gamma)$ says that in every node reachable from the initial node the formula γ is true.

The relation between MSOL and the mu-calculus is expressed in terms of bisimulation invariance. Two states are bisimilar if the computations from them behave in the same way. More formally, a *bisimulation* is a symmetric relation on states of a transition system such that for every (s_1, s_2) in the relation and letter b the following holds: if there is a transition on b from s_1 to s_1', then there is a transition on b from s_2 to a state in the relation with s_1'. A set of states of a transition system is *bisimulation invariant* if for every pair of states in some bisimulation relation, the two states are either both in the set or both outside the set. For every transition system, a mu-calculus sentence defines a set of states where

the sentence holds. This set is always bisimulation invariant. In short, we can say that every sentence of the mu-calculus defines a bisimulation invariant property. An MSOL formula with one free variable also defines a set of states of a given transition system. This set may not be bisimulation invariant. Mu-calculus can be translated to MSOL in the sense that for every mu-calculus sentence one can construct an MSOL formula with one free variable such that in every transition system the two formulas define the same set of states. The translation is syntax directed, and follows from the fact that least and greatest fixpoints are definable in MSOL. The following characterization shows under which condition the inverse translation is possible.

Theorem 3.3 ([38]). *The mu-calculus is expressively equivalent to the bisimulation-invariant fragment of MSOL: if an MSOL formula $\varphi(x)$ defines a bisimulation-invariant property, then it is equivalent to a mu-calculus sentence.*

We will use this theorem to avoid writing complicated formulas. Formulations of a synthesis problem use specifications of the form "the label of every path is in a regular language K". Since this condition is expressible in MSOL and bisimulation invariant, it is also expressible in the mu-calculus.

Going in the opposite direction, we introduce a useful construct that does not preserve bisimulation invariance. We consider a *loop testing predicate* $\circlearrowleft_a$. This predicate holds in a state if there is a transition on a that is a self-loop: $s \models \circlearrowleft_a$ if $(s, s) \in R_a$. This construction allows to express observability conditions, and at the same time does not increase the complexity of the satisfiability problem for the mu-calculus.

Theorem 3.4 ([5]). *The satisfiability problem for the modal mu-calculus with loop testing predicates is decidable in* EXPTIME.

Generalized specifications The first advantage of the mu-calculus with loop testing is that it is expressive enough to express control, blocking and observability constraints, as well as their many possible variations.

- Control; every accessible state has outgoing transition on every uncontrollable action: $\mathrm{Everywhere}(\bigwedge_{a \in Aunc} \langle a \rangle true)$.
- Blocking; every accessible state has at least one outgoing transition: $\mathrm{Everywhere}(\bigvee_{a \in A} \langle a \rangle true)$
- Observability; for every accessible state all actions on unobservable events are self-loops: $\mathrm{Everywhere}(\bigwedge_{\alpha \in A_{\mathrm{uno}}} \circlearrowleft_a)$

Note that for the last of the above properties we really need a loop testing predicate; the property is not expressible in the standard mu-calculus, since it is not bisimulation invariant.

Let us see some more examples of new conditions we can express in the mu-calculus. Suppose that A contains two actions c_1, c_2 and we want to say that at each moment at most one of the two is controllable: $\mathrm{Everywhere}(\langle c_1 \rangle true \vee \langle c_2 \rangle true)$.

For another example take the alphabet $A = \{a, f\}$. Suppose the failure action f is uncontrollable, and we want to say that action a becomes uncontrollable after f occurs: $\nu X.([a]X \wedge \langle f \rangle \mathrm{Everywhere}(\langle a \rangle true))$.

Finally, our example from the beginning of the section "from every state a reset action is reachable" is expressible by $\mathrm{Everywhere}\big(\mu X.\ (\langle r\rangle \mathit{true}) \vee \bigvee_{a\in A}\langle a\rangle X\big)$.

These examples justify the following definition.

Definition 3.2 (Generalized centralized controller synthesis). Given an automaton $\mathcal{A}_p$ and formulas α, β of the mu-calculus with loop testing, decide if there is an automaton $\mathcal{A}_c$ such that $\mathcal{A}_c \vDash \beta$ and $\mathcal{A}_p \times \mathcal{A}_c \vDash \alpha$.

Observe that we can use β to state controllability or observability constraints. The blocking constraint can be expressed using α. On the other hand, in this approach we have no way to express maximality of a controller. In its original formulation, the maximality constraint was introduced to avoid trivial solutions. Here we can avoid trivial solutions using specifications. The price to pay for the richer specification language is that we cannot expect to always have a maximal controller. For example, if the specification α says that every sequence of b actions should be finite, then we can have controllers permitting longer and longer sequences of b actions, but there is no maximal controller for this specification since there is no bound on the length of these sequences.

Theorem 3.5 ([5]). *The generalized controller synthesis problem is decidable.*

It turns out that it is not a restriction to require that a controller is a finite automaton. It can be shown that whenever a potentially infinite controller exists then there exists a finite one too.

The proof of the theorem uses an operation called *division* [3, 5]. It can be shown that for a transition system P and a formula α of the μ-calculus with loop predicates there is a formula α/P of the same logic such that for every transition system C:

$$P \times C \vDash \alpha \quad \text{if and only if} \quad C \vDash \alpha/P.$$

With the help of this operation, we have that

$$\mathcal{A}_c \vDash \beta \wedge (\alpha/P) \quad \text{if and only if} \quad \mathcal{A}_c \vDash \beta \text{ and } \mathcal{A}_c \times P \vDash \alpha.$$

This means that the synthesis problem is reduced to checking satisfiability of the formula $\beta \wedge (\alpha/P)$ of the mu-calculus with loop predicates. By Theorem 3.4 the satisfiability problem is decidable, and, in case the answer is positive, a finite automaton can be effectively constructed.

As a final remark about the generalized Ramadge and Wonham problem, we sketch the encoding of the Church problem. Recall that Church problem is given by an MSOL formula $\varphi(X, Y)$ specifying the relation between input and output sequences of bits. To make the distinction between input and output explicit we take two alphabets A_{in} and A_{out} of input and output bits. For the plant we take a two-state automaton $\mathcal{A}_p$ accepting the language $(A_{\mathrm{in}} \cdot A_{\mathrm{out}})^*$; that is, all the words with interleaved input and output bits. We declare all the letters of A_{in} uncontrollable. This way the controller cannot influence what letters appear in the input. The constraint β that we put on controllers is the conjunction of the controllability condition together with the requirement that there should be no deadlock states in a controller. Finally, the specification α for the controlled plant says that all the labels of all the infinite paths considered as infinite sequences of pairs of bits

satisfy the Church specification $\varphi(X, Y)$. Since this is a bisimulation-invariant property, it can be written as a mu-calculus formula. If $\mathcal{A}_c$ is a solution to a problem formulated this way, we can directly transform $\mathcal{A}_p \times \mathcal{A}_c$ to a device in the Church sense satisfying the specification.

3.3 Notes

The Ramadge and Wonham formulation of the synthesis problem has been intensively investigated [8, 40, 18, 73]. The problem for generalized specifications has been extended to nondeterministic automata [6]. Some extensions not covered by this line of research concern the case when a plant is a pushdown automaton. In this case there are some non-regular specifications talking about stack properties for which the problem is decidable [16, 9, 63].

There are also other formulations of the synthesis problem with devices given beforehand. For example, one can ask to construct a system from a given set of I/O devices, that is, finite-state transducers, subject to some precise restrictions for composing them [42, 61]. Other types of general devices and composition methods are considered in the field of web-services orchestration [2, 7].

4 Distributed synthesis: synchronous architectures

Recall that the Church synthesis problem requires constructing a device that interacts with an environment by reading input signals and sending output signals. We have represented this schematically in Figure 1 as a box with an ingoing arrow for the input and an outgoing arrow for the output.

The distributed synthesis problem formulated by Pnueli and Rosner [54] asks to construct several devices that communicate with the environment and between themselves. This is represented as a graph, with boxes being place-holders for the devices to construct, and edges being communication channels (see Figure 6). As in the Church problem, in every box we put an input/output automaton that reads a letter from every input channel going into the box, and outputs a letter to every output channel going out of the box. The behavior of the whole system is totally synchronous: in one cycle every device reads its input letter and then produces its output letter. Observe that the output of one device can be the input of another device. In this case the letter output by the first device is read in the same cycle by the second device. This kind of semantics is, of course, problematic if there are loops in the architecture graph. For this reason we will restrict our discussion to architectures without loops. Adding loops complicates the semantics, but does not add new insights to the problem, at least from the perspective of this chapter.

After formulating the problem precisely, we will show that the problem is undecidable for essentially all architectures except pipelines. Our objective in this section is to present a selection of results highlighting phenomena that can appear in the distributed synthesis problem. Concerning undecidability, we will discuss a couple of representative architectures in detail. Concerning the pipeline architecture, we will not only present the

decidability proof, but also a detailed sketch of the nonelementary lower bound for the synthesis problem. Even though this architecture is of limited use in itself, the tools used in the analysis may be of broader interest.

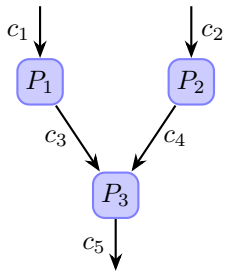


Figure 6. A distributed architecture

Formally, an *architecture* over a channel alphabet A is a tuple

$$\langle A, P, C, \mathrm{src} : C \to \mathcal{P}(P \cup \{\epsilon\}), \mathrm{tgt} : C \to \mathcal{P}(P \cup \{\epsilon\})\rangle$$

where P is the set of processes, nodes in the graph; C is the set of channels, edges in the graph; and src, tgt are the functions defining the incidence relation. Here ϵ is a special label standing for the environment. So if $\mathrm{src}(c) = \epsilon$ then c goes from the environment, or in other words, c is an input channel for the system. Similarly, if $\mathrm{tgt}(c) = \epsilon$ then c is an output channel. As it will be clear from the semantics below, channels c with $\mathrm{src}(c) = \mathrm{tgt}(c) = \epsilon$ do not make much sense, since they do not influence the behavior of the architecture. We will write $\mathrm{In}_p = \mathrm{tgt}^{-1}(p)$ for the set of channels that arrive at the process p, similarly we set $\mathrm{Out}_p = \mathrm{src}^{-1}(p)$. Going back to the architecture from Figure 6, we have $P = \{p_1, p_2, p_3\}$, and $C = \{c_1, \ldots, c_5\}$. For the edges, we have, for example, $\mathrm{src}(c_1) = \epsilon$, and $\mathrm{tgt}(c_1) = p_1$.

At every moment each channel contains one letter. So the content of the channels is described by a function $\chi : C \to A$. We write $\mathrm{In}_p(\chi)$ for a restriction of χ to In_p, and analogously for $\mathrm{Out}_p(\chi)$. A device for a process $p \in P$ is a function $f_p : (A^{\mathrm{In}_p})^* \to A^{\mathrm{Out}_p}$. Given a device for each process a *behavior of a system* is a sequence: $\chi_0, \chi_1, \ldots$ such that for every $i = 0, 1, \ldots$ and every $p \in P$ we have

$$\mathrm{Out}_p(\chi_i) = f_p(\mathrm{In}_p(\chi_0) \cdots \mathrm{In}_p(\chi_i)).$$

This means that the output of p in cycle i depends on the contents of all its input channels during all the previous cycles, including cycle i. Observe that a system may have many behaviors, as the contents of the channels coming from the environment is not constrained. Let us go back to the example in Figure 6. We take f_1 to be the identity function. We then take $f_2(\vec{\chi}) = 1$ if and only if $\vec{\chi}$ contains a 1; this means that f_2 will emit constantly 1 after the first appearance of 1 on the input. For f_3 we take the Boolean conjunction: a function such that $f_3(\vec{\chi}) = 1$ if and only if $\vec{\chi} \in (\{0,1\}^2)^*$ ends in the letter $(1,1)$. Representing a channel contents χ as a vector of five bits, the following is a possible behavior

$$(1,0,1,0,0), (0,1,0,1,0), (1,0,1,1,1), \ldots$$

In the second cycle device f_2 receives 1 on its input, so from that moment the value of its

output channel, c_4, will be always 1. Hence, from that moment the architecture will copy the contents of the channel c_1 to c_5.

Distributed synthesis problem for a fixed architecture asks if for a given specification there exist devices such that when put into the boxes the behavior of the resulting system satisfies the specification.

Definition 4.1 (Distributed synthesis problem). Given an architecture $\langle A, P, C, \mathrm{src}, \mathrm{tgt}\rangle$ with p processes and k channels, and a specification in a form of a regular language K of infinite trees over A^k, decide if there exist devices $f_1, \ldots, f_p$ such that the tree of all the behaviors of the resulting system is in K.

In this formulation we have permitted branching regular specifications since this is the most general case we treat in this chapter. Undecidability results from the next subsection hold also when we take much weaker specifications as in the formulation of the original Ramadge and Wonham problem, namely when we require that all finite prefixes of all the behaviors are in a given language of finite words.

4.1 Undecidability: global and local specifications

It turns out that for most architectures the synthesis problem is undecidable. We will first discuss a general undecidability result that uses the power of specifications to talk about all the processes at the same time. This leads to a notion of local specification that is a conjunction of requirements on each process separately. We show that somehow surprisingly this does not help substantially. The class of decidable architectures for this kind of specifications does not increase substantially.

Consider an architecture presented in Figure 7 consisting of two independent processes, each having its own input and its own output.

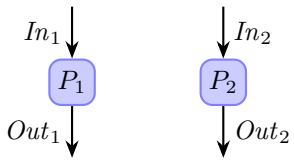


Figure 7. A simple undecidable architecture.

Theorem 4.1. *[54] The synthesis problem for the architecture from Figure 7 is undecidable.*

Proof. We reduce the halting problem for deterministic Turing machines. Given a Turing machine we construct a specification that is realizable if and only if the run of the machine from the empty configuration is infinite.

We fix a deterministic Turing machine M. We assume some encoding of configurations of M by infinite words; say that there is a blank symbol to make the encoding infinite. Let A_M be the alphabet used to write configurations. For two infinite words

$v, w \in (A_M)^\omega$ we write $v \vdash_M w$ to say that w is a successor configuration of v in a computation of M; in particular v must be a configuration too.

The alphabet of the architecture will contain the alphabet of configurations, A_M, together with two special letters $\#$ and \$. The specification will only consider what happens when the input to the two processes is of the form $\#^i\$^\omega$, namely a sequence of $\#$ symbols followed by infinitely many \$ symbols. The first requirement is that each process on the input $\#^i\$^\omega$ produces an output $\#^i v$ with $v \in (A_M)^\omega$. We will also require that on the input $\$^\omega$, namely when there is no $\#$ symbol, the word v on the output is the encoding of the initial configuration of M.

The remaining two requirements will talk about behavior of the two processes at the same time. The first is pictorially expressed by:

$$\text{if} \quad \begin{array}{ll} \mathrm{In}_1\colon & \# \quad \# \; \$ \; \$ \\ \mathrm{Out}_1\colon & \# \quad \cdots \quad \# \; a_1 \; a_2 \quad \cdots \\ \mathrm{In}_2\colon & \# \quad \# \; \$ \; \$ \\ \mathrm{Out}_2\colon & \# \quad \# \; b_1 \; b_2 \end{array} \qquad \text{then } a_i = b_i \text{ for all } i \tag{4.1}$$

where we have represented channel contents at consecutive cycles as vertical tuples. The specification says that if the first \$ sign arrives at the same time in In_1 and in In_2 then the outputs of the two processes should be the same, namely, $a_i = b_i$ for all i.

The second requirement is schematically represented by:

$$\text{if} \quad \begin{array}{ll} \mathrm{In}_1\colon & \# \quad \# \; \$ \; \$ \; \$ \\ \mathrm{Out}_1\colon & \# \quad \cdots \quad \# \; a_1 \; a_2 \; a_3 \quad \cdots \\ \mathrm{In}_2\colon & \# \quad \# \; \# \; \$ \; \$ \\ \mathrm{Out}_2\colon & \# \quad \# \; \# \; b_1 \; b_2 \end{array} \qquad \text{then } (a_1 a_2 \dots) \vdash_M (b_1 b_2 \dots) \tag{4.2}$$

It says that when in the second input the first \$ sign arrives one cycle later than in the first input, the word $(b_1 b_2 \dots)$ should represent the successor configuration of the configuration represented by the word $(a_1 a_2 \dots)$.

We claim that this specification is realizable if and only if the run of M from the initial configuration is infinite. Suppose that we have devices f and g that realize the specification. The crucial observation is that the output in Out_1 is independent from the input in In_2. This means that on the input $\#^i\$^\omega$ device f outputs the same $\#^i v$ independently of what is the input to the other device. Let v_i denote the word that is output by f when reading $\#^i\$^\omega$. Similarly w_i for g.

For the proof in one direction suppose that the computation of M from the initial state is infinite. For f, we can take the device outputting the i-th configuration of M as v_i. We take same device for g. Clearly this strategy realizes the specification.

For the proof in the other direction, the first part of the specification tells us that v_0 is the initial configuration of M. The requirement (4.1) tells us that $v_i = w_i$ for all i. The requirement (4.2) enforces $v_i \vdash_M w_{i+1}$ for all i. This way we have:

$$v_0 \vdash_M w_1 = v_1 \vdash_M w_2 = v_2 \vdash_M w_3 \dots$$

So the sequence $v_1, v_2, \dots$ is an infinite computation of M. Hence, a specification is realizable if an only if there is an infinite computation of M on the empty input. □

Looking at the proof one is tempted to "blame" the specification for undecidability. Indeed, the specification links behaviors of the two processes while they have no means to communicate between themselves. One can say that the specification is *global*, i.e., describes the behavior of the system from the outside; while the visibility of each process

is *local*, i.e., it sees only its input and output channels.

This observation suggest to consider only *local specifications* [44]: specifications that are conjunctions of requirements on input and output channels of each process. Restricting to local specifications remedies our immediate trouble because it makes the synthesis problem decidable for the architecture in Figure 7. Indeed, in this case we just need to solve two independent instances of Church synthesis.

Surprisingly, the restriction to local specifications does not enlarge the class of decidable architectures substantially.

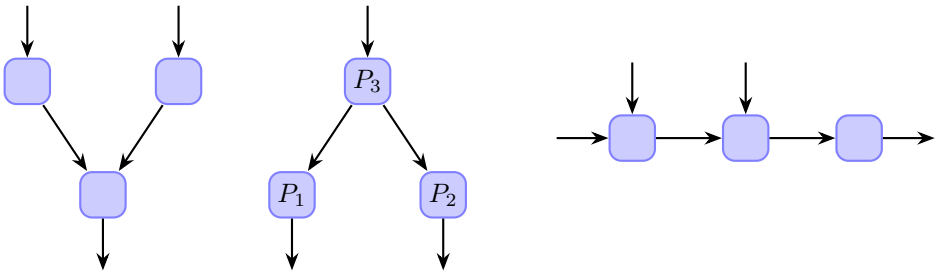


Figure 8. Undecidable architectures for local specifications. The argument for the architecture in the middle of the figure relies on a specification "fixing the input".

Theorem 4.2 ([44]). *The synthesis problem for local specifications is undecidable for the architectures presented in Figure 8.*

The consequence of this theorem is that the synthesis problem is undecidable for all architectures in which we can find one of the patterns from Figure 8. This theorem leaves us with not much more than the pipeline architecture that we will discuss in the next section.

Let us explain the reasons for the undecidability results from Theorem 4.2. For the first architecture of Figure 8 it is not difficult to imagine that a local specification on the process at the bottom can simulate a global specification on the two processes above it. This gives undecidability since these two processes form exactly undecidable architecture from Figure 7.

The reasons for undecidability for the two other architectures are a bit different. We will briefly describe the argument for the second architecture, the one for the third being similar.

Consider the following specification. For every i: on the input of the form $\#^i\$v$, the output should be $\#^i v$. This means that after reading $\$$ on the input the process should output the first letter of v. In the next cycle the same first letter should appear on the input and the process outputs the second letter of v, etc. This strange specification fixes one arbitrary v in a sense that after reading $\#^i\$$ only the fixed v can be send to the input without violating the specification.

As in the proof of Theorem 4.1, we want to write a specification which is realizable if and only if a given deterministic Turing machine has an infinite run on the empty word. For this we impose the strange specification from the previous paragraph on both P_2 and

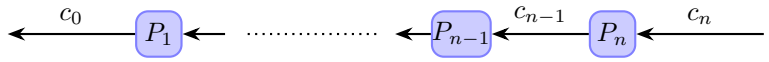


Figure 9. A pipeline architecture. The input on c_n is processed in sequence by $P_n, \ldots, P_1$ and the results is output on c_0.

P_3. We will be interested in inputs to P_1 of two forms: $\#^i(eq)\$^\omega$ or $\#^i(succ)\$^\omega$; where (eq) and $(succ)$ are new symbols. On an input $\#^i(eq)\$^\omega$ process P_1 should send $\#^i\$v$ both to P_2 and P_3. The only restriction on v is that if $i = 0$ then v should be encoding of the initial configuration of M. On an input $\#^i(succ)\$^\omega$ process P_1 should send $\#^i\$v$ to P_2 and $\#^{i+1}\$w$ to P_3, with a condition that v and w are successive configurations of our fixed Turing machine M. This property can be checked using a finite automaton since v and w are produced letter by letter in parallel. Now, because of the specifications we have imposed on P_2 and P_3, for every i there is only one v_i such that $\#^i\$v_i$ can be an input to P_2. Similarly w_i for P_3. Because of the conditions on P_1 we have that: v_0 is the initial configuration, $v_i = w_i$ as well as w_{i+1} is the successor configuration of v_i; for all $i = 0, 1, \ldots$. Hence, the specification is realizable if and only if the computation of M from the initial configuration is infinite.

4.2 How to solve pipeline

A pipeline is a sequence of processes, each reading the output of the preceding one (cf. Figure 9). For unrestricted specifications, the undecidability results from the previous subsection leave pipelines as essentially the only candidates for architectures with decidable distributed synthesis problem. We give a decidability proof, and examine the computational complexity of the problem.

Theorem 4.3 ([54]). *For every pipeline architecture the synthesis problem is decidable.*

Before presenting the proof let us mention that for local specifications the problem is decidable also for a pipeline with additional input at process P_1 [44, 43]. Observe that Theorem 4.2 implies that the problem is undecidable if we add an input at some process P_i for $n > i > 1$.

Proof. Consider a pipeline as in Figure 9. Suppose that for $z = n, \ldots, 1$, we have a device $f_z : A^* \to A$ for the process P_z. A behavior of the pipeline with these devices is a sequence $\chi_0, \chi_1, \ldots$ of channel contents $\chi_i : \{c_n, \ldots, c_0\} \to A$. The constraints coming from the architecture tell us that the output on the channel c_{z-1} is the result of applying device f_z to the input on the channel c_z:

$$\chi_i(c_{z-1}) = f_z(\chi_0(c_z) \ldots \chi_i(c_z)), \quad \text{for } i = 0, 1 \ldots \text{ and } z = 1, \ldots, n.$$

In particular, once devices are fixed, the input on channel c_n determines the behavior of the pipeline. For $w \in A^\omega$, let $\chi_0^w, \chi_1^w, \ldots$ denote the behavior of the pipeline on the input w, namely a unique sequence as above satisfying: $\chi_i^w(c_n) = w_i$ for $i = 0, 1, \ldots$.

The solution we will present uses a reformulation of the notion of the behavior of the pipeline. We will need an operation of composition: for functions $h_1 : A_1^* \to A_2$ and

$h_2 : A_2^* \to A_3$, the composition $comp(h_1, h_2)$ is a function $A_1^* \to (A_2 \times A_3)$ such that for all $v \in A_1^*$

$$comp(h_1, h_2)(v) = (h_1(v), h_2(\overline{h}_1(v))) \tag{4.3}$$

where $\overline{h}_1(v)$ is the sequence of results of h_1 on prefixes of v, namely the sequence $h_1(v_0)h_1(v_0v_1)\dots h_1(v_0\dots v_i)$ where $v = v_0\dots v_i$.

Using the operation of composition we can put together the devices, starting from the rightmost one

$$g_1 = f_1 \quad \text{and} \quad g_i = comp(f_i, g_{i-1}) \text{ for } i = 2, \dots, n\ . \tag{4.4}$$

It may be useful at this point to recall the types of objects in these formulas, $f_i : A^* \to A$, and $g_i : A^* \to A^i$. In particular $g_n : A^* \to A^n$. We get that g_n describes the semantics of the pipeline.

Lemma 4.4. *For every infinite sequence $w \in A^\omega$, and its prefix $w_1 \cdots w_i$ we have $g_n(w_1 \cdots w_i) = (\chi_i^w(c_{n-1}), \dots, \chi_i^w(c_0))$.*

Recall that the semantics of a distributed system is a tree of all its behaviors. Using the above lemma we can consider that the semantics of the pipeline is rather given as a function $h : A^* \to A^n$. In particular, we can assume that the specification is given as a regular language L_n of functions $h : A^* \to A^n$. Indeed, it is straightforward to translate an MSOL specification on tree of behaviors into MSOL specification on such functions. Hence, the pipeline problem can be stated as: given by a regular language L of functions $h : A^* \to A^n$, decide if there exist devices $f_n, \dots, f_1$ such that g_n as defined in (4.4) is in L.

To solve this problem we consider an automaton construction that allows us to deal with *comp* operation used in (4.4) to define the semantics of the pipeline. Given L, a set of functions $h : A_1^* \to (A_2 \times A_3)$, we define the set

$$shape(L) = \{g : A_2^* \to A_3 : \exists f : A_1^* \to A_2.\ comp(f, g) \in L\}\ .$$

So $shape(L)$ is the set of all g for which it is possible to find f such that the result of the composition of the two is in L.

Theorem 4.5 ([41]). *If L is a regular tree language of functions $h : A_1^* \to (A_2 \times A_3)$, then $shape(L)$ is a regular tree language of functions $h' : A_2^* \to A_3$. The parity automaton for $shape(L)$ can be effectively constructed from the parity automaton for L.*

The construction stated in the theorem is based on the equivalence of non-deterministic and alternating tree automata. This theorem gives us a tool to solve the pipeline problem. Taking our specification L we define a sequence of languages:

$$L_n = L, \qquad L_i = shape(L_{i+1}) \quad \text{for } i = n-1, \dots, 1.$$

Let us verify that the pipeline problem has a solution if and only if L_1 is not empty.

If $f_n, \dots, f_1$ is a solution to the pipeline problem then we take the functions g_i as defined in (4.4). Since the pipeline with devices $f_n, \dots, f_1$ satisfies the specification, we have that $g_n \in L_n$. By definition $g_n = comp(f_n, g_{n-1})$, hence $g_{n-1} \in shape(L_n) = L_{n-1}$. By a straightforward induction $g_i \in L_i$ for all $i = n, \dots, 1$.

For the opposite direction, suppose that $L_1 \neq \emptyset$. Take $f_1 : A^* \to A$ from L_1. Since $L_1 = \mathit{shape}(L_2)$, by definition there exists $f_2 : A^* \to A$ such that $\mathit{comp}(f_2, f_1) \in L_2$. Let us use h_2 to denote $\mathit{comp}(f_2, f_1)$. By induction on i we show that there exists $f_i : A^* \to A$ such that $h_i = \mathit{comp}(f_i, h_{i+1}) \in L_i$. Hence $h_n \in L_n$. Function h_n describes the semantics of the pipeline as in (4.4). We get that $f_n, \ldots, f_1$ is a solution to the pipeline problem.

□

4.3 A lower bound for pipeline architecture

Even though the synthesis problem for pipeline is decidable, it turns out to be algorithmically difficult. The complexity grows by one exponential with every new element of the pipeline. We will show that every algorithm for solving the synthesis problem for a pipeline of n elements needs order of $\mathit{Tower}_{n-2}(k)$ time, where k is the size of the specification given as a finite automaton. We denote by $\mathit{Tower}_n(k)$ the tower of exponentials function, namely, $\mathit{Tower}_0(k) = k$ and $\mathit{Tower}_{i+1}(k) = 2^{\mathit{Tower}_i(k)}$.

Theorem 4.6 ([54]). *The complexity of the synthesis problem for pipeline architectures is nonelementary in the number of components.*

This subsection is devoted to a rather detailed sketch of this result since the lower bounds stated in the literature refer to results on multi-player Turing machines [53]. The argument presented below gives an opportunity to show some peculiar specifications one can write in this framework. Among others, we will see once again strange specifications fixing the input we have used to show undecidability of architectures from Figure 8. The specifications we present below can be made local in a sense of [44]. So the lower bound applies also to local specifications. It is a challenging problem to find an interesting subclass of specifications for which the pipeline synthesis problem has lower complexity.

The proof of the lower bound will use similar tools as the proof from [66] of the nonelementary complexity of the satisfiability problem for first-order logic over $\langle \mathbb{N}, \leqslant \rangle$. We will simulate an alternation of quantifiers of the form $\forall_{x_1} \exists_{x_2 > x_1} \forall_{x_3 > x_2} \ldots$, where variables range over positions in an infinite word. Universal quantifiers will be simulated by the input from the environment, existential quantifiers by guessing. The nesting of quantifiers will be simulated by visibility restrictions.

Let us fix n. We will be interested in counters counting to $\mathit{Tower}_n(n)$. We suppose that we have alphabets $\Sigma^k = \{a^k, b^k, \vdash^k, \dashv^k\}$ for $k = 1, \ldots, n$. Additionally we will have a blank symbol B.

Definition 4.2. A 1-*counter* is a sequence of n-letters from Σ^1 prefixed with $\vdash^1$ and finished with $\dashv^1$. Such a sequence represents a number between 0 and $2^n - 1$ by interpreting a^1 as 0 and b^1 as 1, and assuming that the most significant bit is on the right.

A k-*counter*, for $k > 1$, is a sequence of the form $\vdash^k \; c_0 \sigma_0 \ldots c_i \sigma_i \; \dashv^k$, where all c_j are $(k-1)$-counters and all $\sigma_j \in \Sigma^k$. Moreover we require that c_0 represents 0; c_{j+1} represents the successor of a number represented by c_j, for all $j = 0, \ldots, i-1$; and c_i represents the maximal possible value (that is $\mathit{Tower}_k(n) - 1$). The value of the

counter is given by the sequence $\sigma_0 \dots \sigma_i$ interpreted as a binary number with the most the significant bit to the right (as before we consider that a^k stands for 0 and b^k for 1).

In what follows we will construct a specification forcing a controller for a pipeline architecture to output an $(n+1)$-counter. As soon as we achieve this, it will be then easy to modify the construction so that this word is not an $(n+1)$-counter but a computation of $Tower_n(n)$-space bounded Turing machine. The architecture will have $n+2$ processes.

We will start by describing what we mean by forcing a controller to output a word, then we will describe, a quite complicated, specification that forces this word to be an n-counter.

Fixing a word. Suppose that we ask that the letter output by process P_{-1} should appear on its input in the next cycle. In other words, in the first cycle the input to P_{-1} should be the blank symbol B, and the output some letter a_0. Then in a cycle i the input should be a_{i-1}, the letter on the output from the cycle $i-1$, and the output some letter a_i. This curious requirement implies that in some sense process P_{-1} controls its input. We have already used this kind of specifications to prove undecidability for architectures from Figure 8.

We use this requirement in the following context. Suppose that for all processes $P_0, \dots, P_{n-1}$ we just demand that they copy their input to their output. We do not put any requirement on process P_n. The accumulated effect of these requirements is that the processes have to agree on the word that they will output: they should output this word independently on what is the input to P_n. This situation is schematically presented in Figure 10. For a fixed infinite word w, process P_{-1} outputs w, and the other processes output Bw. In particular process P_n disregards its input u.

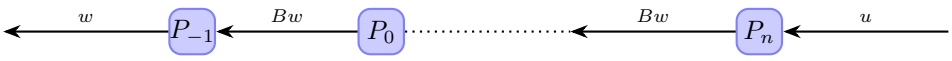


Figure 10. Fixing a word

Marking question and answer positions. Being able to fix a word gives us a great deal of control. Our objective is to force this word to be an n-counter. In what follows we will forget about process P_{-1} whose unique role is to fix a word as described above.

Of course the construction cannot ignore the input completely. The actual mechanics will be slightly more complicated since we will allow processes to output letters decorated by pointers. We will have two sets of pointers:

$$\text{Questions} = \{\uparrow_0, \dots, \uparrow_n\} \cup \{\uparrow^s_0, \dots, \uparrow^s_n\} \qquad \text{and} \qquad \text{Answers} = \{\downarrow_0, \dots, \downarrow_n\}.$$

We have two kinds of question pointers and one kind of answer pointers. One should think of pointers as accents: they come at the same time as a letter. So the alphabet of the pipeline is really:

$$\big(\{B\} \cup \bigcup_{i=1\dots,n} \Sigma_i\big) \times \big(\text{Questions} \cup \text{Answers}\big)$$

We will consider only inputs that are sequences of blanks possibly decorated with pointers. The only interesting input sequences will be those that have $n+1$ question pointers, starting from index n and going down to 0, followed by $(n+1)$ answer pointers with the same order of indices. The pointers will be marking beginnings of counters in the sense that a k pointer will mark the beginning of a k counter. So, in the majority of cases, only symbols $\vdash_k$ will have pointers attached.

We will now describe formally the requirements on appearances of pointers. These requirements intend to simulate the quantifier alternation. As we will see, for an n-counter we will need n quantifier alternations. The descriptions of encodings of 1-counters and 2-counters, presented later, give an example of how these requirements work.

A schema of the desired behavior of pointers is presented in Figure 11. We require that when the pointer $\uparrow_j$, or $\uparrow_j^s$ (for $j = 0, \ldots, n$) appears at the input of a process P_i, for $i = n, \ldots, n-j+1$, then the process should immediately copy it to the output. The process P_{n-j} is forbidden to copy the pointer so the other processes are not aware of its position. In particular $\uparrow_0$ is not copied, so only P_n knows its placement. The only interesting case will be when all the pointers are placed in the n-counter pointed by $\uparrow_n$. After this, process P_0 can mark one of the following n-counter with $\downarrow_n$. Hence $\downarrow_n$ should mark an occurrence of $\vdash^n$ symbol.

When P_0 emits $\downarrow_n$ all other processes should be informed about it, but this information has to flow against the sense of the arrows. We will use once again the "fixing the input" trick to accomplish this. When P_0 emits $\downarrow_n$, we require in the next cycle $\downarrow_n$ appears also on the input of P_n and that it is copied immediately by all other processes. The specification is instantly satisfied if it is not the case that $\downarrow_n$ appears on the input of P_0 one cycle after it appeared on the output of P_0. Observe that P_0 emits $\downarrow_n$ before it learns that there is one on the input. With this mechanism we "discard" all the inputs but the one that points to the position following the one chosen by P_0. Next process P_1 emits $\downarrow_{n-1}$ at the beginning of some $(n-1)$-counter inside the n-counter pointed by $\downarrow_n$. All process are informed about this by the same mechanism as before. This procedure continues till P_n emits $\downarrow_0$. At that moment some properties of the structure of pointers, as described in the following paragraphs, will be checked. A schema of this desired behavior is presented in Figure 11. For clarity we do not show pointers $\downarrow_i$ used to inform other processes about the answer. From now on we assume the behavior of the pipeline as described above.

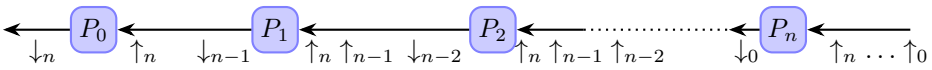
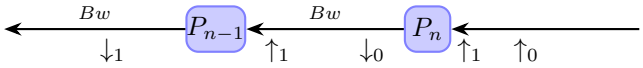


Figure 11. The behavior of question and answer pointers

Specifications eq_1 and $struct_1$. This first part of the specification will force process P_{n-1} to place $\downarrow_1$ pointer at the beginning of a 1-counter with the same value as the 1-counter pointed by $\uparrow_1$. For this specification we are interested only in processes P_n and P_{n-1}. The following picture describes their behavior according to the general mechanics described above.

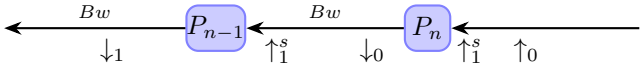


We will say that $\uparrow_1$ and $\uparrow_0$ are *well-placed* if $\uparrow_1$ arrives at the beginning of a 1-counter and $\uparrow_0$ follows at the distance at most n from $\uparrow_1$. By the distance we mean the number of letters in between the two pointers in our fixed word, or equivalently, the number of cycles between appearances of the two pointers.

The specification eq_1 requires that if $\uparrow_1$ and $\uparrow_0$ are well placed then $\uparrow_0$ points to the same position in the counter pointed by $\uparrow_1$ as the position pointed by $\downarrow_0$ in the counter pointed by $\downarrow_1$. Moreover the bits pointed by $\uparrow_0$ and $\downarrow_0$ should be the same.

We claim that if there is a strategy satisfying eq_1, and the general constraints on mechanics described above are satisfied, then after receiving $\uparrow_1$ pointer marking a 1-counter, process P_{n-1} should at some later moment emit $\downarrow_1$ marking a 1-counter with the same value. Indeed, placing $\downarrow_1$ by P_{n-1} depends only on the position of $\uparrow_1$, as P_{n-1} should emit $\downarrow_1$ before seeing $\downarrow_0$. Now, P_{n-1} should put the pointer in such a way that P_n will be able to put its pointer $\downarrow_0$, no matter when $\uparrow_0$ pointer arrives on the input. As P_{n-1} does not know the position of $\uparrow_0$, the only way it can make it possible for P_n to satisfy this condition is to put $\downarrow_1$ at the sequence of n-letters that is the same as that after $\uparrow_1$.

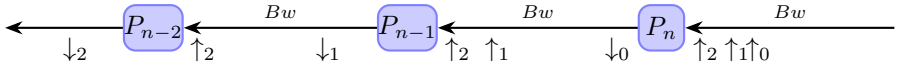
The specification $struct_1$ will force the output word w to be of the form $c_1v_1c_2v_2\ldots$ where c_i are 2-counters and v_i are some, words of size $\leqslant n$ over the alphabet $\Sigma_2 \cup \cdots \cup \Sigma_n$. In principle, these will be the parts used for bigger counters. The specification will be very similar to eq_1. One difference will be that $\uparrow_1$ pointer is replaced by $\uparrow_1^s$ pointer in order to signal that a modified behavior is required.



The other changes to eq_1 are as follows. We demand that $\downarrow_1$ points at the 1-counter immediately following the 1-counter pointed by $\uparrow_1^s$. If the value of $\uparrow_1^s$ counter is not maximal (not a sequence of n letters b^1) then the bits pointed by $\uparrow_0$ and $\downarrow_0$ should satisfy the dependencies required by the successor relation. Moreover, there must be precisely one letter from Σ_2 in between the two counters. If the value is maximal then we can have up to n letters before the next 2-counter, and the value of the counter pointed by $\downarrow_1$ should be 0 (the sequence of n letters a^1).

To sum up, the specification $struct_1$ enforces that each 1-counter in w is followed by a 1-counter representing the successor number, and after the maximal number is reached the counter restarts at 0. This forces the fixed word to be a sequence of 2-counters.

Specifications eq_2 and $struct_2$.T Before giving an inductive construction we examine the constructions for 2-counters. We want to write a specification that permits the input to chose a 2-counter, after which the only way for controllers to win will be to choose another 2-counter with the same value. For this we will use pointers with indices 0, 1, 2. In the picture below we present the relevant part of the behavior of the pipeline that satisfies our general requirements.



To give an intuition we repeat the description of general mechanics in this particular case. On the input we have three pointers arriving in the order $\uparrow_2, \uparrow_1, \uparrow_0$. When $\uparrow_2$ arrives both process P_n and P_{n-1} have to copy it to their respective output channels, while P_{n-2} does not copy the pointer. When $\uparrow_1$ arrives, only P_n copies the pointer. The pointer $\uparrow_0$ is not copied. At some later moment, process P_{n-2} should output a $\downarrow_2$ pointer. If one cycle later $\downarrow_2$ pointer does not appear on the input of P_n then the specification is instantaneously satisfied. So the only interesting case is when $\downarrow_2$ appears at the right moment and it is copied by all the processes. In consequence, P_n and P_{n-1} get to know that $\downarrow_2$ has been emitted. At some later point P_{n-1} outputs a pointer $\downarrow_1$, and P_n is informed about it by the same mechanism. Finally, P_n outputs $\downarrow_0$.

We will require that $struct_1$ holds so we can think that w is a sequence of 2-counters with some other letters in between. Similarly to the previous case, we will say that $\uparrow_2, \uparrow_1$, are well-placed if $\uparrow_2$ is at the beginning of a 2-counter (at the symbol $\vdash_2$ followed by the 1-counter representing 0) and $\uparrow_1$ is inside this 2-counter (i.e., before the next letter $\dashv_2$). We will also talk about well-placed $\downarrow_2, \downarrow_1$.

The specification eq_2 requires four things:

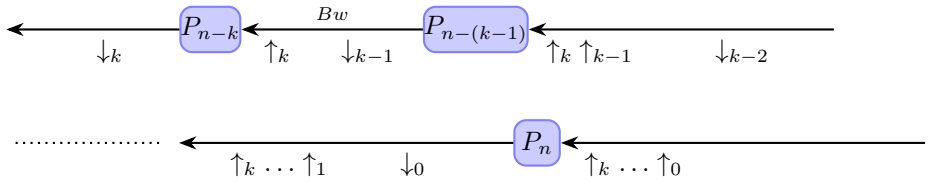
- specification $struct_1$ should be satisfied;
- the policy of pointer placement as described above should be followed;
- if $\uparrow_2, \uparrow_1$ are well-placed then so should be $\downarrow_2, \downarrow_1$.
- the specification eq_1 should hold, and the letter from Σ_2 just after 1-counter pointed by $\uparrow_1$ should be the same as the letter after one counter pointed by $\downarrow_1$.

We claim that if there is a strategy satisfying eq_2 then in a reply to a $\uparrow_2$ pointer placed at the beginning of some 2-counter the strategy must put $\downarrow_2$ at the beginning of some other 2-counter with the same value. First, as $struct_1$ is satisfied, we know that w is a sequence of 2-counters separated by some control letters. The mechanics of putting pointers is set so that P_{n-2} has to place $\downarrow_2$ pointer only knowing the placement of $\uparrow_2$ pointer. It should do this so that P_{n-1} has then a chance to put $\downarrow_1$ pointer without violating the specification. As $\downarrow_1$ must be well-placed, this means that the 1-counter pointed by $\downarrow_1$ should be inside the 2-counter pointed by $\downarrow_2$. Because eq_1 must hold, the values of the counters pointed by $\uparrow_1$ and $\downarrow_1$ should be the same. This means that the position of the $\downarrow_1$ pointer is uniquely determined when the position of $\downarrow_2$ is chosen. Because P_{n-2} does not know the position of $\downarrow_1$, the only way to satisfy the specification is to choose a 2-counter that has the same value as the one pointed by $\uparrow_2$.

Let us briefly describe a specification $struct_2$ that will force the output to be of the form $c_1 v_1 c_2 v_2 \ldots$, where c_i are 3-counters and v_i are some words of size $\leqslant n-1$. It is very similar to eq_2 but for the following modifications. Pointer $\uparrow_2$ is replaced by $\uparrow_2^s$ to signal that a different type of check is need. We additionally require that $\downarrow_2$ pointer should point to the 2-counter immediately after the one pointed by $\uparrow_2^s$. We still require that eq_1 condition holds, but now the letters from Σ_2 just after 1-counters pointed by $\uparrow_1$ and $\downarrow_1$ should not be compared for equality but rather should follow the rules for successor. There is one exception, when the counter pointed by $\uparrow_2^s$ is maximal (consists only of b_2's) then the counter pointed by $\downarrow_2$ should represent 0 (consist only of a_2's). These conditions mean that $\downarrow_2$ should point to the counter just following the one pointed by $\uparrow_2^s$. Pointers $\uparrow_1$ and $\downarrow_1$ should point at the same positions in respective counters, and the bits at these positions should respect the successor rules. So the value of a 2-counter pointed by $\uparrow_2^s$ is the successor of the value of the counter pointed by $\downarrow_2$.

Specifications eq_k and $struct_k$.T We can now present a generalization of eq_2 to arbitrary $k \leqslant n$. We want to enforce that $\downarrow_k$ pointer is put at the beginning of a k-counter with the same value as the k-counter pointed by $\uparrow_k$. The construction is by induction so we assume that we already have specifications eq_{k-1} and $struct_{k-1}$.

The behavior implied by the general rules of pointer placement is depicted below:



As in the previous cases we say that $\uparrow_k$, $\uparrow_{k-1}$ are well-placed when: (i) $\uparrow_k$ points at the beginning of some k-counter, and (ii) $\uparrow_{k-1}$ points inside this k-counter. The condition (i) means that $\uparrow_k$ marks the symbol $\vdash_k$, and the value of the $k-1$ counter that follows is 0. This is easily verified as it amounts to checking that there are no b^{k-1} letters before a letter from Σ^k. The condition (ii) amounts to checking that $\uparrow_{k-1}$ is before the next Σ^{k+1} letter. We will also talk about well-placed $\downarrow_k$ and $\downarrow_{k-1}$.

The specification eq_k requires four things:

- the specification $struct_{k-1}$ should be satisfied;
- the policy of pointer placement should be followed;
- if $\uparrow_k$, $\uparrow_{k-1}$ are well-placed then $\downarrow_k$, $\downarrow_{k-1}$ should be too.
- specification eq_{k-1} should be satisfied, and the letter from Σ_k appearing after $(k-1)$-counter pointed by $\uparrow_{k-1}$ should be the same as the letter appearing after $(k-1)$-counter pointed by $\downarrow_{k-1}$.

Since $struct_{k-1}$, holds we know that the fixed word is a sequence of k-counters. The placement of $\uparrow_k$ depends only on $\downarrow_k$. The later should be placed in such a way that process $P_{n-(k-1)}$ can place $\downarrow_{k-1}$ pointer without violating the specification. The specification asks that positions of $\downarrow_{k-1}$ and $\uparrow_{k-1}$ in their respective counters should be the same. The specification also says that the bits of the k-counters at these positions should be the same. Since process P_{n-k} does not know the position of $\uparrow_{k-1}$ pointer, the only way for him to permit satisfaction of the specification is to put $\downarrow_k$ pointer at the counter whose value is equal to the one pointed by $\uparrow_k$ pointer.

The specification $struct_k$ should say that the fixed word is of the form $c_1 v_1 c_2 v_2 \ldots$, where c_i are $k+1$-counters and v_i are some words of size $\leqslant n-1$. This specification is a modification of eq_k specification in the same way as $struct_2$ is that of eq_2. Pointer $\uparrow_k^s$ is used in place of $\uparrow_k$. It is required that $\downarrow_k$ points to the k-counter immediately following the one pointed by $\uparrow_k^s$. Finally, the dependencies of bits should follow the rules for successor.

Summing up Our encoding of long computations of a Turing machine uses the technique of "fixing an input word" that is done by process P_{-1} in Figure 10. Then with $struct_{n-1}$ we can force the fixed word to be of the form $c_1 v_1 c_2 v_2 \ldots$ where $c_1, c_2, \ldots$ are n-counters. An n-counter is a sequence $\vdash^n c'_0 \sigma_0 \ldots c'_i \sigma_i \dashv^n$ where $c'_0, \ldots, c'_i$ are all the $n-1$ counters listed in the increasing order. The sequence $\sigma_0, \ldots, \sigma_i$ can then encode a configuration of a Turing machine of size $Tower_n(n)$. We can subsequently modify

$struct_n$ so that it does not force the fixed word to be $(n+1)$-counter, but rather a sequence $d_1\$d_2\$d_3\ldots$ where $d_1, d_2, \ldots$ are successive configurations of size $Tower_n(n)$ of some given Turing machine. This way for a pipeline with $n+2$ processes we can write a specification that is realizable if and only if the given $Tower_n(n)$-space bounded Turing machine accepts a given input.

4.4 Notes

The idea of synthesizing a distributed system is of course very attractive. Since [25] it has reappeared in many contexts [48, 19, 65]. One can encode distributed synthesis problem of Pnueli and Rosner into Ramadge and Wonham setting using visibility restrictions [60, 74, 5]. Unfortunately, this does not give distinctively new decidable classes [58, 70, 5, 68, 6].

The decidability proof for pipelines presented here is based on [41]. Generalization to architectures with loops requires to consider a semantics with a delay: a process reads its input in one cycle and reacts with an output in the following cycle. This complicates notation considerably but does not add anything substantially new to the problem. In op. cit. it is shown that the synthesis problem is decidable for doubly flanked pipelines. An extension of the model with broadcast has been also studied [32, 62].

The notion of local specifications has been introduced and studied in [44, 43]. Some more decidability results have been obtained by further restricting specifications to talk only about external inputs and outputs [34, 67].

One promising attempt to get a decidable framework of distributed synthesis is to change the way information is distributed in the system. In the setting presented in this chapter, every controller sees only its inputs and its outputs. In order to deduce some information about the global state of the system a controller can use only his knowledge about the architecture and the initial state of the system. In particular, controllers are not permitted to pass additional information during communication. It is clear though that when we allow some transfer of information during communication, we give more power to controllers.

Pushing the idea of sharing information to the limit, we obtain a model where two processes involved in a communication share all the information they have about the global state of the system [33]. This point of view is not as unrealistic as it may seem at the first glance. It is rooted in the theory of traces that studies finite communicating automata with this kind of information transfer. A fundamental result of Zielonka [75, 29] implies that in fact there is a bound on the size of additional information that needs to be transferred during communication. In our terms, the theory of traces considers the case of distributed synthesis for closed systems, i.e., systems without environment. For the distributed synthesis with environment, decidability results for some special cases are known [33, 45, 52, 20, 36, 51, 31]. Moreover, similarly to Zielonka's Theorem, these results give a bound on additional information that needs to be transferred. The decidability of the general case is open. Interestingly, the general case can be formulated as an extension of the Ramadge and Wonham setting from words, that is linear orders, to special partial orders called Mazurkiewicz traces. We describe this approach in the next section.

5 Distributed synthesis: Zielonka automata

The synthesis problem for synchronous architectures from the previous section is not constrained enough. We have seen that suitably using the interplay between specifications and an architecture, one can get undecidability results for most architectures. Yet the kinds of specifications that lead to these results are rather artificial, like: using a constraint linking two disconnected parts of the system; or using an output channel to single out one input of an unbounded length. These observations motivate a search for other formulations of the distributed synthesis problem that would eliminate some of these undesirable phenomena, and would be decidable for some larger classes of systems.

A Zielonka automaton is a very simple parallel device. It is a parallel composition of several finite automata synchronizing on common actions. Every component has its own alphabet of actions, but these alphabets may have letters in common. A Zielonka automaton accepts a regular language respecting the parallelism implied by the distribution of actions over component automata: if a is followed by b and the two letters do not share a component, i.e. do not appear together in an alphabet of some component, then it can be as well that b is followed by a. Languages of this kind are called *trace languages*. The theory of trace languages offers many results and tools. In particular, many fundamental results of the theory of regular languages have their equivalent trace versions [29].

In this section we present an adaptation of the Ramadge and Wonham formulation of the control problem to Zielonka automata. We obtain this way a setting for distributed synthesis since the devices we construct are distributed by design. In this formulation specifications cannot constraint the flow of information between the components. In consequence, we avoid many pathological behaviors of the Pnueli-Rosner formulation from the previous section. Still, as we will see, the setting is far from being trivial. There are more architectures for which the problem is known to be decidable. It is even possible that the synthesis problem for Zielonka automata is decidable for all architectures.

5.1 Zielonka automata and Zielonka's Theorem

Take $\mathcal{P}$ a finite set of processes, these are names for the components of a Zielonka automaton. An alphabet A is distributed over these components. It means that there is a function $dom : A \to (2^{\mathcal{P}} \setminus \{\emptyset\})$ assigning to each letter a set of processes the letter uses. A *Zielonka automaton* for this distribution is a tuple:

$$\mathcal{A} = \langle (S_p)_{p\in\mathcal{P}}, (\delta_a)_{a\in A}, s^0, F\rangle$$

where S_p is a finite set of states for each process $p \in \mathcal{P}$; we will denote by S the product $\prod_{p\in P} S_p$. We can think of a Zielonka automaton as a constrained product of automata: each over its set of states S_p. The set S is the set of all possible states of this product; they are called *global states*. State $s^0 \in S$ is the initial (global) state; and $F \subseteq S$ is the set of finial states. The crucial part is the definition of the transition relation. We have that $\delta_a \subseteq (\prod_{p\in dom(A)} S_p)^2$ namely that it acts only on the components assigned to the letter a. The transition relation $\delta \subseteq S \times A \times S$ on global states is then given by: $(s, a, s') \in \delta$ if $((s_p)_{p\in dom(a)}, a, (s'_p)_{p\in dom(a)}) \in \delta_a$ and $s'_p = s_p$ for $p \notin dom(a)$. The automaton is *deterministic* if δ_a is a function for all $a \in A$.

The language of the automaton $\mathcal{A}$ is the language of the finite automaton $\langle S, \delta, s^0, F\rangle$ where the components are defined as above. This language has a particular property: if two letters $a, b \in A$ have disjoint process domains, and a word $vabw$ is in the language then $vbaw$ is too. We can thus define an independence relation on letters $(a, b) \in I$ if $dom(a) \cap dom(b) = \emptyset$, and say that $L \subseteq A^*$ is *I-closed* if whenever $vabw \in L$ then $vbaw \in L$ for all words v, w and all $(a, b) \in I$. So languages accepted by Zielonka automata over a fixed distributed alphabet are I-closed.

Zielonka's theorem says that the converse is true. Namely, if a language is I-closed for some I induced by a distribution $dom : A \to (2^{\mathcal{P}} \setminus \{\emptyset\})$ then there is a Zielonka automaton accepting this language.

Theorem 5.1. *[75] Let $dom : A \to (2^{\mathcal{P}} \setminus \{\emptyset\})$ be a distribution of letters, let I be the induced independence relation. If a language $L \subseteq A^*$ is regular and I-closed then there is a deterministic Zielonka automaton accepting L.*

This theorem gives us a tool to implement a regular language on a simple distributed device. Even though the theorem has been proved more than 20 years ago, there is still continuing effort to simplify the proof and improve the complexity of the translation. To give an idea of the complications involved let us look at an example from [27].

Example: Let $\mathcal{P} = \{1, \ldots, n\}$ be the set of processes. The letters of the alphabet are pairs of processes, two letters are dependent if they have a process in common. Formally, the distributed alphabet A consists of two element subsets of $\mathcal{P}$, and the distribution is $dom(\{p, q\}) = \{p, q\}$.

The language $Path_n$ is the set of words $a_1 \cdots a_k$ such that every two consecutive letters have a process in common: $a_i \cap a_{i+1} \neq \emptyset$ for $i = 1, \ldots, k-1$. Observe that a deterministic sequential automaton recognizing this language simply needs to remember the last letter it has read. So it has less than $|\mathcal{P}|^2$ states. Zielonka's theorem guarantees that there is a deterministic Zielonka automaton for the language. Even for $n = 4$ it is not clear how to construct such an automaton with less than a hundred of states. In general we know how to construct a Zielonka automaton that is polynomial in the size of a given sequential automaton and simply exponential in the number of processes [35]. There are no non-trivial lower bounds known for the size of deterministic Zielonka automata for the languages $Path_n$. A lower bound is known under additional assumption of automata being locally-rejecting, meaning that a word is rejected if and only if the run of the automaton passes through some local state designated as rejecting. Locally-rejecting deterministic automaton for $Path_n$ must have at least $2^{n/4}$ states [35].

One could try to use Zielonka's Theorem directly to solve a distributed synthesis problem. For example, one can start with the Church synthesis problem, solve it, and if the solution happenes to respect the required independence, then one could 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 usually algorithmically impossible to approximate a regular language by a language respecting a given independence relation.

Theorem 5.2. *[65] It is not decidable if given an independence relation I and a regular language $L \subseteq A^*$ there is an I-closed language included in L such that every letter from A appears in some word of that language.*

The condition on appearance of letters is not crucial here. Observe that we need some condition in order to make the problem non-trivial, since by definition the empty language is I-closed.

The above theorem suggest that we will not be able to obtain decidability just by restricting the class of possible implementations to Zielonka automata. We need also to restrict specifications. It is quite natural to ask that specifications should be I-closed as well. This is the direction we will present below. Instead of extending Church formulation we will immediately look at a, more general, Ramadge and Wonham formulation.

5.2 Control of Zielonka automata

The starting point of the Ramadge and Wonham formulation was a notion of a plant that is just a finite automaton with all its states accepting. In the distributed case we first fix a set of processes $\mathbb{P}$ and a distribution of actions over processes $dom : A \to (2^{\mathbb{P}} \setminus \emptyset)$. With these fixed, a plant is just any Zielonka automaton with all states accepting. Similarly, a controller is also such an automaton. A controlled plant is modeled by a product of the two Zielonka automata. The control condition for a set of uncontrollable actions $A_{\mathrm{unc}} \subseteq A$ is formulated as in the case of finite automata: in every global state of the controller every uncontrollable action should be possible.

Once again it would be more convenient to formulate the synthesis problem in terms of languages instead of automata. For this we need to understand what kind of languages are recognized by deterministic Zielonka automata with all the states accepting. We call such languages *implementable*. The characterization of implementable languages is based on Zielonka's theorem. One needs to observe that languages of this kind satisfy an additional property called *forward diamond*: if $wa, wb \in L$ and aIb then $wab \in L$.

Proposition 5.3. *[64, 28] A regular language over a distributed alphabet is implementable if and only if it is prefix-closed, I-closed, and satisfies the forward diamond condition.*

Observe that I in the above proposition is determined by the distributed alphabet.

Definition 5.1 (Decentralized control problem). Fix a distribution of actions over processes $dom : A \to (2^{\mathbb{P}} \setminus \emptyset)$ and a set of uncontrollable actions $A_{\mathrm{unc}} \subseteq A$. Given languages P, K implementable with respect to distribution dom, find the biggest with respect to set inclusion implementable language C such that $P \cap C \subseteq K$ and two conditions are satisfied:

implementable C is implementable;
control if $w \in C$ and $a \in A_{\mathrm{unc}}$ then $wa \in C$.

Observe that the definition is very similar to the centralized case; the difference being that the prefix closure requirement is replaced by implementability requirement. This is not surprising as prefix closure is indeed a characterization of languages of finite automata with all states accepting.

The distributed aspect in this definition is hidden in the notion of implementability since it is equivalent to being the language of a Zielonka automaton. The intersection

$P \cap C$ in the definition translates to the product of two Zielonka automata, the one of a plant and the one of a controller. Recall that a Zielonka automaton is a distributed device, so the plant is a set of processes communicating via rendez-vous. Controller can be then seen as a set of local controllers: one for each process. In the product of the plant and the controller at the rendez-vous these controllers can exchange their information. So controller cannot add communication to the plant, but it can use existing communication to transfer information between its parts. The important point is that the specification cannot limit the information they can exchange. In consequence, controllers have more power in this formulation than in Pnueli and Rosner setting.

We will show that with a simple restriction on uncontrollable actions decentralized control problem can be reduced to centralized case. We say that an action b is *local* if $dom(b)$ is a singleton.

Theorem 5.4. *Let $dom : A \to (2^{\mathbb{P}} \setminus \emptyset)$ be distributed alphabet, and suppose that every uncontrollable action is local. Every decentralized control problem over such an alphabet has a unique maximal solution. Moreover this solution is an implementable language and can be computed algorithmically.*

Proof. For the proof of the theorem we will show that a controller C constructed in the centralized case is implementable. We know that C is prefix-closed. From the characterization in Lemma 3.2 on page 1161, and using implementability of P and K, it is not difficult to show that C is I-closed. It remains to check the forward diamond property.

Suppose that $wa, wb \in C$ with aIb. We need to show that $wab \in C$. For this we will use the characterization from Lemma 3.2. First observe that if $wa \in P$ then $wb \in P$ by forward diamond property of P. Hence we consider two cases.

The first case is when $wa, wb \notin P$. Hence automaton $\mathcal{A}_c$ reading wa reaches the state $\top_c$. This means that $waA^* \subseteq C$. In particular $wab \in C$.

The second case is when $wa, wb \in P$. By Lemma 3.2 we get that $wa \in P \cap K$ and $wa(A_{\mathrm{unc}})^* \cap P \subseteq wa(A_{\mathrm{unc}})^* \cap K$; and similarly for wb. To show $wab \in C$ we need to show two things:

- $wab \in P \cap K$. This follows from forward diamond properties of P and K.
- $wab(A_{\mathrm{unc}})^* \cap P \subseteq wab(A_{\mathrm{unc}})^* \cap K$. Take a word $u \in (A_{\mathrm{unc}})^*$ and suppose that $wabu \in P$. We show that $wabu \in K$. Since all uncontrollable letters are local, we can split u into three parts: u_a the subsequence of letters located on processes from dom_a; u_b similarly but for dom_b; u_r the subsequence of the remaining letters. Observe that the three words are independent and moreover u_r is independent from a and b. By I-closure property of P we have that $wu_rau_abu_b \in P$ and $wu_rbu_bau_a \in P$. We get $wu_rau_a \in K$ since wau_au_r is in P and hence also in K. For the same reason $wu_rbu_b \in K$. By induction on the length of u_a and u_b, from I-diamond property we get $wu_rabu_au_b \in K$. Then by I-closure we get the desired $wabu_ru_au_b \in K$.

□

This positive result is not that satisfactory. The controllers it gives will often have deadlocks. For a simple example consider an alphabet $A = \{a, b, c\}$ with aIb and c dependent on both a and b. One can imagine that a is on executed on one process, b

on the other, and c on both. Suppose that all actions are controllable. Let $P = A^*$ and $K = (a + b)c^* + (ab)$. The two languages are implementable. Of course the maximal controller is just K. But K has deadlocks since when a and b happen concurrently no c action is possible. The maximal controller without deadlocks given by the procedure in the sequential setting is $C = (a + b)c^*$. But this controller is not implementable since it does not satisfy forward diamond property. An implementable controller needs to decide to permit only a or only b actions. So for example, $C_a = ac^*$ or $C_b = bc^*$ are reasonable controllers for the problem, but there is no implementable controller containing the two at the same time.

5.3 Notes

It is not known at present if the decentralized control synthesis problem is decidable. One direction to approach this problem could be to study the decidability of the MSOL theory of event structures generated by plants. Using the same ideas as in the sequential case, it is possible to encode the decentralized control problem into satisfiability problem of an MSOL formula over the event structure determined by the plant. Unfortunately, there are very simple plants with undecidable MSOL theory of the generated event structure but with decidable decentralized control problem. There exist though an interesting case when the MSOL theory is decidable [45]. The idea is that for every process of the automaton there should be a fixed bound on the number of actions other processes can do in parallel with this process. Under this restriction there is an MSOL definable encoding of the event structure of the language of a given automaton into the full binary tree. On the other hand, it is easy to see that MSOL theory of the event structure is undecidable if for every n it contains traces of the form $xu^n v^n y$ with u independent form v, and y dependent on both u and v. The conjecture due to Thiagarajan is that this is the only forbidden pattern, namely the even structure of a trace language without such a pattern has a decidable MSOL theory.

Other decidable cases of the decentralized control problem refer to the notion of a communication graph. This is a graph where nodes are processes and edges are possible communication channels between them, or more precisely, there is an edge between two processes if there is an action involving both of them. So the communication graph is determined by the distribution of actions over processes. Decentralized control problem is decidable if the communication graph is a co-graph [33]. It is also decidable when all actions involve at most two processes and the communication graph is a tree [51], or in other words when every process can communicate only with its parent and with its children. The case when the communication graph is a cycle of more than 4 processes is open.

References

[1] P. A. Abdulla, A. Bouajjani, and J. d'Orso. Deciding monotonic games. In M. Baaz and J. A. Makowsky, editors, *Proc. 17th Workshop on Computer Science Logic*, volume 2725 of

Lecture Notes in Comput. Sci., pages 1–14. Springer, 2003. 1159

[2] G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004. 1167

[3] H. R. Andersen. Partial model checking. In *Proc. 10th IEEE Symp. on Logic in Computer Science*, pages 398–407. IEEE Computer Society, 1995. 1166

[4] A. Arnold and D. Niwiński. *Rudiments of μ-calculus*. Elsevier, 2001. 1163

[5] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoret. Comput. Sci.*, 303(1):7–34, 2003. 1159, 1165, 1166, 1180

[6] A. Arnold and I. Walukiewicz. Nondeterministic controllers of nondeterministic processes. In J. Flum, E. Grädel, and T. Wilke, editors, *Logic and Automata*, volume 2 of *Texts in Logic and Games*, pages 29–52. Amsterdam University Press, 2007. 1167, 1180

[7] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of web services with messaging. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, and B. C. Ooi, editors, *Proc. of the 31st Int. Conf. on Very Large Data Bases*, pages 613–624. ACM, 2005. 1167

[8] A. Bergeron. A unified approach to control problems in discrete event processes. *RAIRO Theor. Inform. Appl.*, 27:555–573, 1993. 1167

[9] A. Bouquet, O. Serre, and I. Walukiewicz. Pushdown games with the unboundedness and regular conditions. In P. K. Pandya and J. Radhakrishnan, editors, *Proc. 23rd Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *Lecture Notes in Comput. Sci.*, pages 88–99. Springer, 2003. 1167

[10] P. Bouyer, T. Brihaye, and F. Chevalier. O-minimal hybrid reachability games. *Log. Methods Comput. Sci.*, 6(1:1), Jan. 2010. 1159

[11] J. Bradfield and C. Stirling. Modal mu-calculi. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *The handbook of modal logic*, pages 721–756. Elesvier, 2006. 1163

[12] J. Bradfield and I. Walukiewicz. The mu-calculus and model-checking. In H. V. E. Clarke, T. Henzinger, editor, *Handbook of model checking*, pages 871–919. Springer-Verlag, 2015. 1163

[13] C. Broadbent, A. Carayol, L. Ong, and O. Serre. Recursion schemes and logical reflection. In *Proc. 25th IEEE Symp. on Logic in Computer Science*, pages 120–129. IEEE Computer Society, 2010. 1159

[14] J. Büchi and L. Landweber. Solving sequential conditions by finite state strategies. *Trans. Amer. Math. Soc.*, 138:367–378, 1969. 1150

[15] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Int. Congr. for Logic, Methodology and Philosophy of Science*, pages 1–11, 1962. 1150, 1154, 1158

[16] T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a Sigma-3 winning condition. In J. C. Bradfield, editor, *Proc. 16th Workshop on Computer Science Logic*, volume 2471 of *Lecture Notes in Comput. Sci.*, pages 322–336. Springer, 2002. 1167

[17] A. Carayol, M. Hague, A. Meyer, C.-H. L. Ong, and O. Serre. Winning regions of higher-order pushdown games. In *Proc. 23rd IEEE Symp. on Logic in Computer Science*, pages 193–204. IEEE Computer Society, 2008. 1159

[18] C. G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Kluwer Acad. Publ., 1999. 1150, 1167

[19] I. Castellani, M. Mukund, and P. S. Thiagarajan. Synthesizing distributed transition systems from global specification. In C. P. Rangan, V. Raman, and R. Ramanujam, editors, *Proc. 19th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Comput. Sci.*, pages 219–231. Springer, 1999. 1180

[20] T. Chatain, P. Gastin, and N. Sznajder. Natural specifications yield decidability for distributed synthesis of asynchronous systems. In M. Nielsen, A. Kucera, P. B. Miltersen, C. Palamidessi, P. Tuma, and F. D. Valencia, editors, *Proc. 35th Conf. on Current Trends in Theory and Practice of Informatics (SOFSEM)*, volume 5404 of *Lecture Notes in Comput. Sci.*, pages 141–152. Springer, 2009. 1180

[21] K. Chatterjee, L. de Alfaro, and T. Henzinger. Qualitative concurrent parity games. *ACM Transactions on Computational Logic (TOCL)*, 2011. 1159

[22] K. Chatterjee and T. A. Henzinger. A survey of stochastic ω-regular games. *J. Comput. System Sci.*, 78(2):394–413, 2012. 1159

[23] K. Chatterjee, T. A. Henzinger, and V. Prabhu. Timed games: Complexity and robustness. *Log. Methods Comput. Sci.*, 7(4), 2011. 1159

[24] A. Church. Applications of recursive arithmetic to the problem of cricuit synthesis. In *Summaries of the summer institute of symbolic logic*, volume I, pages 3–50. Cornell Univ., Ithaca, N.Y., 1957. 1150

[25] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Comput. Sci.*, pages 52–71. Springer, 1981. 1159, 1180

[26] A. Condon. The complexity of stochastic games. *Inform. Comput.*, 96(2):203–224, 1992. 1159

[27] R. Cori, Y. Métivier, and W. Zielonka. Asynchronous mappings and asynchronous cellular automata. *Inform. Comput.*, 106:159–202, 1993. 1182

[28] V. Diekert and A. Muscholl. On distributed monitoring of asynchronous systems. In C. L. Ong and R. J. G. B. de Queiroz, editors, *Proc. 19th Int. Workshop on Logic, Language, Information and Computation, WoLLIC 2012*, volume 7456 of *Lecture Notes in Comput. Sci.*, pages 70–84. Springer, 2012. 1183

[29] V. Diekert and G. Rozenberg, editors. *The book of traces*. World Scientific, Singapore, 1995. 1180, 1181

[30] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd Ann. Symp. Found. Comput. Sci.*, pages 368–377. IEEE Computer Society, 1991. 1158

[31] B. Finkbeiner and E. Olderog. Petri games: Synthesis of distributed systems with causal memory. *Inf. Comput.*, 253:181–203, 2017. 1180

[32] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proc. 20th IEEE Symp. on Logic in Computer Science*, pages 321–330. IEEE Computer Society, 2005. 1180

[33] P. Gastin, B. Lerman, and M. Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In K. Lodaya and M. Mahajan, editors, *Proc. 24th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Comput. Sci.*, pages 275–286. Springer, 2004. 1180, 1185

[34] P. Gastin, N. Sznajder, and M. Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3):215–237, June 2009. 1180

[35] B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Optimal Zielonka-type construction of deterministic asynchronous automata. In S. Abramsky, C. Gavoille, C. Kirchner, F. M. auf der Heide, and P. G. Spirakis, editors, *Proc. 37th Int. Conf. on Automata, Languages, and Programming (ICALP) Part II*, volume 6199 of *Lecture Notes in Comput. Sci.*, pages 52–63. Springer, 2010. 1182

[36] B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Asynchronous games over tree architectures. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *Proc. 40th Int. Conf. on Automata, Languages, and Programming (ICALP) Part II*, volume 7966 of *Lecture Notes in Comput. Sci.*, pages 275–286. Springer, 2013. 1180

[37] P. Hänsch, M. Slaats, and W. Thomas. Parametrized regular infinite games and higher-order pushdown strategies. In M. Kutylowski, W. Charatonik, and M. Gebala, editors, *Fundamentals of Computation Theory, 17th Int. Symp.: FCT '09*, volume 5699 of *Lecture Notes in Comput. Sci.*, pages 181–192. Springer, 2009. 1159

[38] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In U. Montanari and V. Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory*, volume 1119 of *Lecture Notes in Comput. Sci.*, pages 263–277. Springer, 1996. 1165

[39] S. C. Kleene. Representation of events in nerve nets and finite automata. In J. McCarthy and C. Shannon, editors, *Automata studies*. Princeton Univ. Press, 1956. 1159

[40] R. Kumar and V. K. Garg. *Modeling and control of logical discrete event systems*. Kluwer Acad. Publ., 1995. 1150, 1167

[41] O. Kupferman and M. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, pages 389–398. IEEE Computer Society, 2001. 1173, 1180

[42] Y. Lustig and M. Y. Vardi. Synthesis from component libraries. In L. de Alfaro, editor, *Proc. 12th Int. Conf. on Foundations of Software Science and Computation Structures*, volume 5504 of *Lecture Notes in Comput. Sci.*, pages 395–409. Springer, 2009. 1167

[43] P. Madhusudan. *Control and synthesis of open reactive systems*. PhD thesis, University of Madras, 2001. 1172, 1180

[44] P. Madhusudan and P. Thiagarajan. Distributed control and synthesis for local specifications. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Proc. 28th Int. Conf. on Automata, Languages, and Programming (ICALP)*, volume 2076 of *Lecture Notes in Comput. Sci.*, pages 396–407. Springer, 2001. 1171, 1172, 1174, 1180

[45] P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In R. Ramanujam and S. Sen, editors, *Proc. 25th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Comput. Sci.*, pages 201–212. Springer, 2005. 1180, 1185

[46] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 6(1):68–93, 1984. 1159

[47] D. Martin. Borel determinacy. *Ann. Math.*, 102:363–371, 1975. 1158

[48] R. Morin. Decompositions of asynchronous systems. In D. Sangiorgi and R. de Simone, editors, *Proc. 9th Int. Conf. on Concurrency Theory*, volume 1466 of *Lecture Notes in Comput. Sci.*, pages 549–564. Springer, 1998. 1180

[49] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In *5th Symposium on Computation Theory*, volume 208 of *Lecture Notes in Comput. Sci.*, pages 157–168. Springer, 1984. 1158

[50] A. W. Mostowski. Games with forbidden positions. Technical Report 78, University of Gdansk, 1991. 1158

[51] A. Muscholl and I. Walukiewicz. Distributed synthesis for acyclic architectures. In V. Raman and S. P. Suresh, editors, *Proc. 34th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 29 of *LIPIcs*, pages 639–651. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. 1180, 1185

[52] A. Muscholl, I. Walukiewicz, and M. Zeitoun. A look at the control of asynchronous automata. In *Perspectives in Concurrency Theory*, pages 356–371. Univ. Press, Hyderabad, 2009. 1180

[53] G. L. Peterson and J. H. Reif. Multi-person alternation. In *Proc. 20th Ann. Symp. Found. Comput. Sci.*, pages 348–363. IEEE Computer Society, 1979. 1174

[54] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st Ann. Symp. Found. Comput. Sci.*, pages 746–757. IEEE Computer Society, 1990. 1167, 1169, 1172, 1174

[55] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–23, 1969. 1150, 1154

[56] M. O. Rabin. *Automata on infinite objects and Church's problem*. Amer. Math. Soc., Providence, RI, 1972. 1150

[57] A. Rabinovich. The Church synthesis problem with parameters. *Log. Methods Comput. Sci.*, 3:1–24, 2007. 1159

[58] P. J. G. Ramadge. Some tractable supervisory control problems for discrete-event systems modeled by Büchi automata. *IEEE Trans. Automat. Control*, 34(1):10–19, 1989. 1159, 1180

[59] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(2):81–98, 1989. 1150, 1159, 1161

[60] K. Rudie and W. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Trans. Automat. Control*, 37(11):1692–1708, 1992. 1180

[61] S. Salvati and I. Walukiewicz. Evaluation is MSOL-compatible. In A. Seth and N. K. Vishnoi, editors, *Proc. 33rd Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 24 of *LIPIcs*, pages 103–114. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. 1167

[62] S. Schewe. *Synthesis of Distributed Systems*. PhD thesis, Universität des Saarlandes, 2008. 1180

[63] O. Serre. Games with winning conditions of high Borel complexity. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Proc. 31st Int. Conf. on Automata, Languages, and Programming (ICALP)*, volume 3142 of *Lecture Notes in Comput. Sci.*, pages 1150–1162. Springer, 2004. 1167

[64] A. Stefanescu. *Automatic synthesis of distributed transition systems*. PhD thesis, Universität Stuttgart, 2006. 1183

[65] A. Stefanescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In R. M. Amadio and D. Lugiez, editors, *Proc. 14th Int. Conf. on Concurrency Theory*, volume 2761 of *Lecture Notes in Comput. Sci.*, pages 27–41, 2003. 1180, 1182

[66] L. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Department of Electrical Engineering MIT, 1974. 1174

[67] N. Sznajder. *Synthèse de systèmes distribués ouverts*. PhD thesis, École Normale Supérieure de Cachan, 2009. 1180

[68] J. G. Thistle. Undecidability in decentralized supervision. *Systems Control Lett.*, 54(5):503–509, 2005. 1180

[69] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal language theory*, volume III, pages 389–455. Springer, 1997. 1158

[70] S. Tripakis. Undecidable problems in decentralized observation and control for regular languages. *Inform. Process. Lett.*, 90(1):21–28, 2004. 1180

[71] M. Y. Vardi and T. Wilke. Automata: From logics to algorithms. In J. Flum, E. Grädel, and T. Wilke, editors, *Logic and automata*, volume 2 of *Texts in Logic and Games*. Amsterdam University Press, 2007. 1163

[72] I. Walukiewicz. Pushdown processes: Games and model checking. *Inform. Comput.*, 164(2):234–263, 2001. 1159

[73] W. Wonham. Supervisory control of discrete-event systems. Technical Report ECE 1636F/1637S 2009-10, University of Toronto, 2010. 1167

[74] T.-S. Yoo and S. Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. *Discrete Event Dynamic Systems*, 12(3):335–377, 2002. 1180

[75] W. Zielonka. Notes on finite asynchronous automata. *RAIRO Theor. Inform. Appl.*, 21:99–135, 1987. 1180, 1182