Automated Synthesis of Distributed Controllers

Anca Muscholl

LaBRI, University of Bordeaux

Abstract. Synthesis is a particularly challenging problem for concurrent programs. At the same time it is a very promising approach, since concurrent programs are difficult to get right, or to analyze with traditional verification techniques. This paper gives an introduction to distributed synthesis in the setting of Mazurkiewicz traces, and its applications to decentralized runtime monitoring.

1 Context

Modern computing systems are increasingly distributed and heterogeneous. Software needs to be able to exploit these advances, providing means for applications to be more performant. Traditional concurrent programming paradigms, as in Java, are based on threads, shared-memory, and locking mechanisms that guard access to common data. More recent paradigms like the reactive programming model of Erlang [4] and Scala [35,36] replace shared memory by asynchronous message passing, where sending a message is non-blocking.

In all these concurrent frameworks, writing reliable software is a serious challenge. Programmers tend to think about code mostly in a sequential way, and it is hard to grasp all possible schedulings of events in a concurrent execution. For similar reasons, verification and analysis of concurrent programs is a difficult task. Testing, which is still the main method for error detection in software, has low coverage for concurrent programs. The reason is that bugs in such programs are difficult to reproduce: they may happen under very specific thread schedules and the likelihood of taking such corner-case schedules is very low. Automated verification, such as model-checking and other traditional exploration techniques, can handle very limited instances of concurrent programs, mostly because of the very large number of possible states and of possible interleavings of executions.

Formal analysis of programs requires as a pre-requisite a clean mathematical model for programs. Verification of sequential programs starts usually with an abstraction step – reducing the value domains of variables to finite domains, viewing conditional branching as non-determinism, etc. Another major simplification consists in disallowing recursion. This leads to a very robust computational model, namely *finite-state automata* and *regular languages*. Regular languages of words (and trees) are particularly well understood notions. The deep connections between logic and automata revealed by the foundational work of Büchi, Rabin, and others, are the main ingredients in automata-based verification.

In program synthesis, the task is to turn a specification into a program that is guaranteed to satisfy it. Synthesis can therefore provide solutions that are correct by construction. It is thus particularly attractive for designing concurrent programs, that are often difficult to get right or to analyze by traditional methods. In distributed synthesis, we are given in addition an architecture, and the task is to turn the specification into a distributed implementation over this architecture.

Distributed synthesis proves to be a real challenge, and there are at least two reasons for this. First, there is no canonical model for concurrent systems, simply because there are very different kinds of interactions between processes. Compare, for example, multi-threaded shared memory Java programs, to Erlang or Scala programs with asynchronous function calls. This issue is connected with another, more fundamental reason: techniques for distributed synthesis are rather rare, and decidability results are conditioned by the right match between the given concurrency model and the kind of questions that we ask.

Mazurkiewicz traces were introduced in the late seventies by A. Mazurkiewicz [31] as a simple model for concurrency inspired by Petri nets. Within this theory, Zielonka's theorem [45] is a prime example of a result on distributed synthesis.

This paper gives a brief introduction to Mazurkiewicz traces and to Zielonka's theorem, and describes how this theory can be used in the verification and the design of concurrent programs. We focus on the synthesis of concurrent programs and its application to decentralized runtime monitoring.

Monitoring is a more lightweight alternative to model-checking and synthesis. The task is to observe the execution of a program in order to detect possible violations of safety requirements. Monitoring is a prerequisite for control, because it can gather information about things that went wrong and about components that require repair actions. In programming, monitoring takes the form of assertions: an invalidation of an assertion is the first sign that something has gone wrong in the system. However, concurrent programs often require assertions concerning several components. A straightforward but impractical way to verify such an assertion at runtime is to synchronize the concerned components and to inquire about their states. A much better way to do this is to write a distributed monitor that deduces the required information by recording and exchanging suitable information using the available communication in the program. Mazurkiewicz trace theory and Zielonka's theorem can provide a general, and yet practical method for synthesizing distributed monitors.

Overview of the paper. Section 2 sets the stage by describing some classical correctness issues for concurrent programs. Section 3 introduces Mazurkiewicz traces, and Section 4 presents some applications to decentralized monitoring.

2 Distributed models: some motivation

Concurrent programming models usually consist of entities, like processes or threads, that evolve in an asynchronous manner and synchronize on joint events,

such as access to shared variables, or communication. We start with some illustrating examples from multi-threaded programming, and with some typical correctness properties. This will allow us to present the type of questions that we want to address.

A multi-threaded program consists of an arbitrary number of concurrently executing threads. We will assume that there is a fixed set $\mathcal{T}$ of threads. There is no global clock, so threads progress asynchronously. Threads can either perform local actions or access the global memory, consisting of shared variables from a fixed set X. Possible actions of a thread $T \in \mathcal{T}$ include reads $r(T, x, v)$ and writes $w(T, x, v)$ on a shared variable $x \in X$ (for some value v) and acquiring $acq(T, L)$, resp. releasing $rel(T, L)$ a lock L. More complex forms of access to the shared memory, such as compare-and-set (CAS), are commonly used in lock-free programming. We will not use CAS in the remaining of this section, but come back to it in Section 3.

Partial orders are a classical abstraction for reasoning about executions of multi-threaded programs. The computation on each thread is abstracted out by a set of events, and the multi-threaded execution is abstracted in form of a partial order on these events. An early example is Lamport's *happens-before* relation [27], originally described for communicating systems. This relation orders the events on each thread, and the sending of a message before its receive. In multi-threaded programs with shared memory, where locks guard the access to shared variables, the happens-before relation orders two events if they are performed by the same thread or they use the same lock.

A more formal, general definition of the happens-before relation for programs goes as follows. Let Σ be the set of actions in a program. We will assume throughout the paper that Σ is finite. Depending on the problem that we consider, we will assume that there is a binary *conflict* relation $D \subseteq \Sigma \times \Sigma$ between the actions of the program. For example, we will have $a \ D \ b$ if a and b are performed by the same thread. Starting with a linear execution $a_1 \cdots a_n \in \Sigma^*$ of a program, the *happens-before* relation is the partial order on positions defined as the reflexive-transitive closure of the relation $\{i \prec j \mid i < j \text{ and } a_i \ D \ a_j\}$. As we will see in Section 3, if the conflict relation is symmetric, this partial order is a *Mazurkiewicz trace*.

In the remaining of this section we outline two frequently considered correctness issues for concurrent programs, that will be used as examples for decentralized monitoring in Section 4.

2.1 Race detection

Race detection is one of the widely studied problems of concurrent software. Informally, a race occurs whenever there are conflicting accesses to the same variable without proper synchronization. Detecting races is important since executions with races may yield unexpected behaviors, caused by the outcome of the computation depending on the schedule of threads.

In order to define races for multi-threaded programs with lock synchronization we need to introduce the happens-before relation for such programs. Let Σ

be the set of actions in a program, for instance:

$$\Sigma = \{w(T,x), r(T,x), acq(T,L), rel(T,L) \mid T, x, L\}\,.$$

Two actions from Σ are in *conflict* if

- they are performed by the same thread, or
- they acquire or release the same lock.

A *race* occurs in an execution if there are two accesses to the same shared variable such that

- they are *unordered* in the happens-before relation, and
- at least one of them is a write.

Example 1. Figure 1 illustrates a race problem due to locking that is too fine-grained. Two threads have access to a list pointed to by `head`. `Thread 1` adds an element to the head of the list, while `Thread 2` deletes the head element. The two instructions protected by the lock are ordered in the happens-before relation. However, `t1.next = head` and `head = head.next` are unordered. Since the first instruction is a read, and the second a write, this situation is a race condition.

```
type list {int data; list *next}
list *head
```

```
Thread 1

1: t1 = new(list);
2: t1.data = 42;
3: t1.next = head;
4: ack(lock)
5:   head = t1
6: rel(lock)
```

```
Thread 2

7:  t2 = head;
8:  ack(lock)
9:   head = head.next
10: rel(lock)
```

Fig. 1. A race condition in the execution $1, 2, 3, 7, 8, 9$: events 3 and 9 are unordered in the happens-before relation.

2.2 Atomicity

Atomicity, or conflict serializability, is a high-level correctness notion for concurrent programs that has its origins in database transactions. A transaction consists of a block of operations, such as reads and writes to shared memory variables, that is marked as `atomic`. An execution of the transaction system is serial if transactions are scheduled one after the other, without interleaving them. A serial execution reflects the intuition of the programmer, about parts of the code marked as transactions as being executed atomically.

In order to define when a multi-threaded program is conflict-serializable, we need first the notion of equivalent executions. Two executions are *equivalent* if they define the same happens-before relation w.r.t. the following conflict relation: Two actions from $\Sigma = \{w(T,x), r(T,x) \mid T, x\}$ are in conflict if

- they are performed by the same thread, or
- they access to the same variable, and at least one of them is a write.

The above conflict relation has a different purpose than the one used for the race problem: here, we are interested in the values that threads compute. Two executions are considered to be equivalent if all threads end up with the same values of (local and global) variables. Since a write $w(T,x)$ potentially modifies the value of x, its order w.r.t. any other access to x should be preserved. This guarantees that the values of x are the same in two equivalent executions.

A program is called *conflict-serializable* (or *atomic*) if every execution is equivalent to a serial one. As we will explain in Section 3 this means that every execution can be reordered into an equivalent one where no transaction is interrupted.

Example 2. Figure 2 shows a simple program with two threads that is not conflict-serializable. The interleaved execution where `Thread 2` writes after the read and before the write of `Thread 1`, is not equivalent to any serial execution.

```
Thread 1

1: atomic {
2:   read(x);
3:   write(x)
4:   }
```

```
Thread 2

5: atomic {
6:   write(x);
7:   }
```

Fig. 2. A program that is not conflict-serializable: the execution $1, 2, 5, 6, 7, 3, 4$ is not equivalent to any serial execution.

3 Mazurkiewicz traces and Zielonka's theorem

This section introduces Mazurkiewicz traces [31], one of the simplest formalisms able to describe concurrency. We will see that traces are perfectly suited to describe dependency and the happens-before relation. The notion of conflicting actions and the happens-before relation seen in the previous section are instances of this more abstract approach.

The definition of traces starts with an alphabet of actions Σ and a *dependence relation* $D \subseteq \Sigma \times \Sigma$ on actions, that is reflexive and symmetric. The idea behind this relation is that two dependent actions are always ordered, for instance because the outcome of one action affects the other action. For example, the actions of acquiring or releasing the same lock are ordered, since a thread has to wait for a lock to be released before acquiring it.

Example 3. Coming back to the problems introduced in Sections 2.1 and 2.2, note that the conflict relations defined there are both symmetric. For example, we can define the dependence relation D over the alphabet $\Sigma = \{r(T,x), w(T,x) \mid T \in \mathcal{T}, x \in X\}$ of Section 2.2, by letting $a\ D\ b$ if a, b are in conflict.

While the dependence relation coincides with the conflict relation, the happens-before relation is the *Mazurkiewicz trace* order. A Mazurkiewicz trace is the labelled partial order $T(w) = \langle E, \preceq \rangle$ obtained from a word $w = a_1 \dots a_n \in \Sigma^*$ in the following way:

- $E = \{e_1, \dots, e_n\}$ is the set of *events*, in one-to-one correspondence with the positions of w, where event e_i has label a_i,
- $\preceq$ is the reflexive-transitive closure of $\{(e_i, e_j) \mid i < j,\ a_i\ D\ a_j\}$.

From a language-theoretical viewpoint, traces are almost as attractive as words, and several results from automata and logics generalize from finite and infinite words to traces, see e.g. the handbook [10]. One of the cornerstone results in Mazurkiewicz trace theory is based on an elegant notion of finite-state distributed automata, *Zielonka automata*, that we present in the remaining of the section.

Informally, a Zielonka automaton [45] is a finite-state automaton with control distributed over several *processes* that synchronize on shared actions. Synchronization is modeled through a distributed action alphabet. There is no global clock: for instance between two synchronizations, two processes can do a different number of actions. Because of this, Zielonka automata are also known as *asynchronous automata*.

A *distributed action alphabet* on a finite set $\mathbb{P}$ of processes is a pair (Σ, dom), where Σ is a finite set of *actions* and $dom : \Sigma \to (2^{\mathbb{P}} \setminus \emptyset)$ is a *domain function*. The domain $dom(b)$ of action b comprises all processes that synchronize in order to perform b. The domain function induces a natural dependence relation D over Σ by setting $a\ D\ b$ if $dom(a) \cap dom(b) \neq \emptyset$. The idea behind is that executions of two dependent actions affect at least one common process, so their order matters. By contrast, two *independent actions* a, b, i.e., where $dom(a) \cap dom(b) = \emptyset$, can be executed either as ab or as ba, the order is immaterial.

Example 4. We reconsider Example 3. The dependence relation D defined there can be realized by a distributed alphabet (Σ, dom) on the following set of processes:

$$\mathbb{P} = \mathcal{T} \cup \{\langle T, x \rangle \mid T \in \mathcal{T}, x \in X\}\,.$$

Informally, each thread $T \in \mathcal{T}$ represents a process; in addition, there is a process for each pair $\langle T, x \rangle$. The process $\langle T, x \rangle$ stands for the cached value of x in thread T.

The domain function defined below satisfies $a\ D\ b$ iff $dom(a) \cap dom(b) \neq \emptyset$:

$$dom(a) = \begin{cases} \{T, \langle T, x \rangle\} & \text{if } a = r(T,x) \\ \{T, \langle T', x \rangle \mid T' \in \mathcal{T}\} & \text{if } a = w(T,x)\,. \end{cases}$$

The intuition behind $dom(a)$ is as follows. A read $r(T, x)$ depends both on the internal state of thread T and the cached value of x, and will affect the state of T. A write $w(T, x)$ depends on the internal state of thread T and will affect not only the state of T, but also the cached values of x on other threads using x, since the new value will be written into these caches.

A *Zielonka automaton* $\mathcal{A} = \langle (S_p)_{p\in\mathbb{P}},\ (s_p^{init})_{p\in\mathbb{P}},\ \delta\rangle$ over (Σ, dom) consists of:

- a finite set S_p of (local) states with an initial state $s_p^{init} \in S_p$, for every process $p \in \mathbb{P}$,
- a transition relation $\delta \subseteq \bigcup_{a\in\Sigma} \big(\prod_{p\in dom(a)} S_p \times \{a\} \times \prod_{p\in dom(a)} S_p\big)$.

For convenience, we abbreviate a tuple $(s_p)_{p\in P}$ of local states by s_P. An automaton is called *deterministic* if the transition relation is a partial function.

Before explaining the semantics of Zielonka automata, let us comment the idea behind the transitions and illustrate it through an example. The reader may be more familiar with synchronous products of finite automata, where a joint action means that every automaton having this action in its alphabet executes it according to its transition relation. Joint transitions in Zielonka automata follow a *rendez-vous* paradigm, meaning that processes having action b in their alphabet can exchange information via the execution of b: a transition on b depends on the states of all processes executing b. The following example illustrates this effect:

Example 5. The `CAS` (compare-and-swap) operation is available as atomic operation in the JAVA package java.util.concurrent.atomic, and supported by many architectures. It takes as parameters the thread identifier T, the variable name x, and two values, `old` and `new`. The effect of the instruction `y = CAS(T,x,old,new)` is conditional: the value of x is replaced by `new` if it is equal to `old`, otherwise it does not change. The method returns `true` if the value changed, and `false` otherwise.

A `CAS` instruction can be seen as a synchronization between two processes: P_T associated with the thread T, and P_x associated with the variable x. The states of P_T are valuations of the local variables of T. The states of P_x are the values x can take. An instruction of the form `y = CAS(T,x,old,new)` becomes a synchronization action between P_T and P_x with the two transitions of Figure 3 (represented for convenience as Petri net transitions).

A Zielonka automaton can be seen as a usual finite-state automaton, whose set of states $S = \prod_{p\in\mathbb{P}} S_p$ is given by the global states, and transitions $s \xrightarrow{a} s'$ if $(s_{dom(a)}, a, s'_{dom(a)}) \in \delta$, and $s_{\mathbb{P}\setminus dom(a)} = s'_{\mathbb{P}\setminus dom(a)}$. Thus states of this automaton are tuples of states of the processes of the Zielonka automaton. As a language acceptor, a Zielonka automaton $\mathcal{A}$ accepts a *trace-closed language* $L(\mathcal{A})$, that is, a language closed under commuting adjacent independent symbols. Formally, a language L is trace-closed when $uabv \in L$ if and only if $ubav \in L$, for all $u, v \in \Sigma^*$ and all independent actions a, b.

A cornerstone result in the theory of Mazurkiewicz traces is a construction transforming a sequential automaton into an equivalent deterministic Zielonka

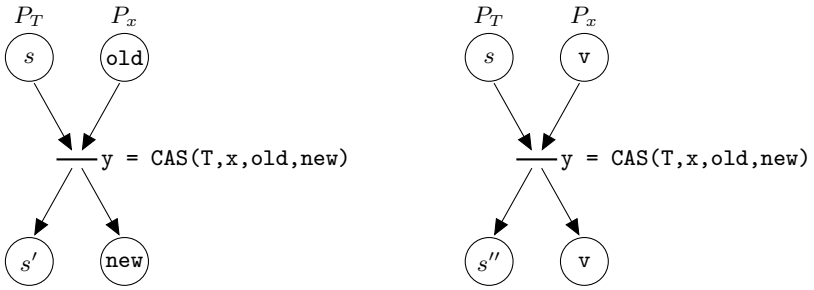


Fig. 3. `CAS` as transitions of a Zielonka automaton. On the left side of the figure we have the case when the value of x is `old`; on the right side `v` is different from `old`. Notice that in state s' the value of y is `true`, whereas in s'', it is `false`.

automaton. This beautiful result is one of the rare examples of distributed synthesis with broader scope.

Theorem 1. *[45] Given a distributed alphabet (Σ, dom), and a regular trace-closed language $L \subseteq \Sigma^*$ over (Σ, dom). A deterministic Zielonka automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$ can be effectively constructed.*

The only assumption of the theorem above is that the language of the automaton is trace-closed, but this is unavoidable. Moreover, trace closure can be checked easily, e.g. on the minimal DFA of the given language.

The construction behind Theorem 1 is technically involved, but also very fascinating. The crux is to show how to put together distributed information using additional memory that is *finite*[1]. Many researchers contributed to simplify the construction and to improve its complexity, see [8,33,19,16] and references therein. The most recent construction [16] produces deterministic Zielonka automata of size that is exponential in the number of processes (and polynomial in the size of a DFA for L). The exponential dependence on the number of processes is necessary, modulo a technical assumption (that is actually required for monitoring).

Theorem 2 ([16]). *There is an algorithm that takes as input a distributed alphabet (Σ, dom) over n processes and a DFA $\mathcal{A}$ accepting a trace-closed language over (Σ, dom), and computes an equivalent deterministic Zielonka automaton $\mathcal{B}$ with at most $4^{n^4} \cdot |\mathcal{A}|^{n^2}$ states per process. Moreover, the algorithm computes the transitions of $\mathcal{B}$ on-the-fly in polynomial time.*

4 Distributed monitoring

The construction of deterministic Zielonka automata opens interesting perspectives for monitoring concurrent programs. In order to monitor a concurrent program at runtime, the monitor has to be distributed (or decentralized). This

[1] Vector clocks [30] are a similar notion in distributed computing, but they do not require a finite domain of values.

means that there is a local monitor on each thread, and these local monitors can exchange information. The exchange can be implemented by allowing local monitors to initiate extra communication, or, more conservatively, by using the available communication in the program in order to share monitoring-relevant information. We follow the latter setting here, since adding communication can reduce the concurrency, and it is very difficult to quantify how much performance is lost by adding communication.

Apart from detecting violations of safety properties at runtime, the information gathered by such monitors can be also used to recover from an unsafe state. Of course, this can be done only at runtime, and not offline, by inspecting sequential executions a posteriori.

Our general approach for this kind of distributed monitoring is simple: we have some trace-closed, regular property ϕ that should be satisfied by every execution of a given program or system. To detect possible violations of ϕ at runtime, we construct a monitor for ϕ and run it in parallel with the program. Consider the scenario where the program P is modeled by a Zielonka automaton $\mathcal{A}_P$. If a monitor is also a Zielonka automaton $\mathcal{A}_M$, then running the monitor M in parallel to P amounts to build the usual product automaton between $\mathcal{A}_P$ and $\mathcal{A}_M$ process-wise.

Interestingly, many properties one is interested to monitor on concurrent programs can be expressed in terms of the happens-before relation between specific events, as the following example illustrates.

Example 6. Consider the *race detection problem* from Section 2.1. A race occurs when two conflicting accesses to the same variable are unordered in the happens-before relation. Therefore, a violation of the "no-race" property is monitored by looking for two *unordered* accesses to the same variable, at least one of them being a write.

Monitoring a violation of *atomicity* (recall Section 2.2) is done by checking for every transaction on some thread T, that no action c of some thread $T' \neq T$ happened after the beginning $a = beg(T)$ of the transaction on T (cf. instruction 1 of Example 2) and before its matching end $b = end(T)$ (cf. instruction 4). In other words, the monitor looks for events c on $T' \neq T$ satisfying $a \prec c \prec b$ in the happens-before relation.

Determining the partial ordering between specific events is closely related to the kernel of all available constructions behind Zielonka's theorem. This is known as the *gossip automaton* [33], and the name reflects its rôle: it computes what a process knows about the knowledge of other processes. Using *finite-state* gossiping, processes can put together information that is distributed in the system, hence reconstruct the execution of the given DFA.

The gossip automaton is already responsible for the exponential complexity of Zielonka automata, in all available constructions. A natural question is whether the construction of the gossip automaton can be avoided, or at least simplified. Perhaps unsurprisingly, the theorem below shows that gossiping is not needed when the communication structure is acyclic.

The *communication graph* of a distributed alphabet (Σ, dom) with unary or binary action domains is the undirected graph where vertices are the processes, and edges relate processes $p \neq q$ if there is some action $a \in \Sigma$ such that $dom(a) = \{p, q\}$.

Theorem 3 ([23]). *Let (Σ, dom) be a distributed alphabet with acyclic communication graph. Every regular, trace-closed language L over Σ can be accepted by a deterministic Zielonka automaton with $O(s^2)$ states per process, where s is the size of the minimal DFA for L.*

The theorem above can be useful to monitor programs with acyclic communication if we can start from a small DFA for the trace-closed language L representing the monitoring property. However, in some cases the DFA is necessarily large because it needs to take into account many interleavings. For example, monitoring for some unordered occurrences of b and c, requires a DFA to remember *sets* of actions. In this case it is more efficient to start with a description of L by partial orders. We discuss a solution for this setting in Section 4.1 below.

We need to emphasize that using Zielonka automata for monitoring properties in practice does not depend only on the efficiency of the constructions from the above theorems. In addition to determinism, further properties are desirable when building monitoring automata. The first requirement is that a violation of the property to monitor should be detectable locally. The reason for this is that a thread that knows about the failure can start some recovery actions or inform other threads about the failure. Zielonka automata satisfying this property are called *locally rejecting* [16]. More formally, each process p has a subset of states $R_p \subseteq S_p$; an execution leads a process p into a state from R_p if and only if p knows already that the execution cannot be extended to a trace in $L(\mathcal{A})$. The second important requirement is that the monitoring automaton $\mathcal{A}_M$ should not block the monitored system. This can be achieved by asking that in every global state of $\mathcal{A}_M$ such that no process is in a rejecting state, every action is enabled. A related discussion on desirable properties of Zielonka automata and on implementating the construction of [16] is reported in [2].

4.1 Gossip in trees

In this section we describe a setting where we can compute efficiently the happens-before relation for selected actions of a concurrent program. We will first illustrate the idea on the simple example of Section 2.2. The program there has two threads, T_1 and T_2, and one shared variable x. For clarity we add actions $beg(T_i), end(T_i)$ denoting the begin/end of the atomic section on T_i. Thus, the alphabet of actions is:

$$\Sigma = \{beg(T_1), end(T_1), w(T_1, x), r(T_1, x), beg(T_2), end(T_2), w(T_2, x)\}\,.$$

The dependence relation D includes all pairs of actions of the same thread, as well as the pairs $(r(T_1, x), w(T_2, x))$ and $(w(T_1, x), w(T_2, x))$. Following Example 4, the Zielonka automaton has processes $\mathbb{P} = \{T_1, T_2, \langle T_1, x\rangle, \langle T_2, x\rangle\}$ and the

domains of actions are:

$$dom(beg(T_i)) = dom(end(T_i)) = \{T_i\},$$
$$dom(r(T_i, x)) = \{T_i, \langle T_i, x\rangle\}$$
$$dom(w(T_i, x)) = \{T_i, \langle T_i, x\rangle, \langle T_{3-i}, x\rangle\}$$

Note that we can represent these four processes as a (line) tree in which the domain of each action spans a connected part of the tree, see Figure 4.



Fig. 4. A tree of processes $\mathcal{T}$. The dashed part is the domain of the read action, whereas the dotted parts are the domains of the two writes.

The Mazurkiewicz trace in Figure 5, represented as a partial order, shows a violation of conflict serializability: event c at step 6 satisfies $a \prec c \prec b$, where a represents step 1, and b step 4. The happens-before relation can be computed piecewise by a Zielonka automaton, by exchanging information via the synchronization actions. Figure 6 illustrates how processes update their knowledge about the partial order. Note how the two partial orders represented by thick lines, are combined together with the action $w(T_2, x)$ of step 6, in order to produce the partial orders of processes $\langle T_1, x\rangle$, $\langle T_2, x\rangle$ and T_2 in the last column. Thus, after step 6 these processes know that action $w(T_2, x)$ happened after $beg(T_1)$. Executing then steps 3 and 4 will inform process T_1 about the violation of conflict serializability.

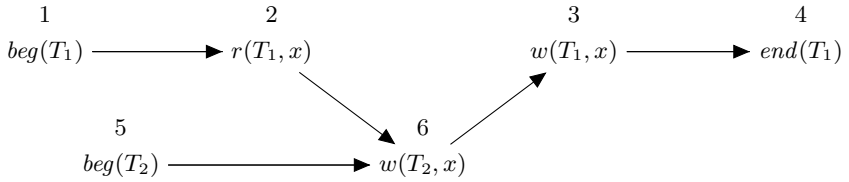


Fig. 5. Violation of conflict serializability (partial order).

Gossiping in trees of processes works more generally as follows. We call a distributed alphabet (Σ, dom) over process set $\mathbb{P}$ *tree-like*, if there exists some tree $\mathcal{T}$ with $\mathbb{P}$ as set of vertices, such that the domain of each action is a *connected* part of $\mathcal{T}$.

Note that the tree $\mathcal{T}$ is uniquely defined by action domains in the special case where the actions have at most binary domains. Otherwise, there can be

several trees as above, but we will assume that the distributed alphabet comes with a suitable tree representation.

We are also given a set of monitoring actions $\Gamma \subseteq \Sigma$. The task is to compute for each process $p \in \mathbb{P}$ the *happens-before relation w.r.t.* Γ, in other words the happens-before relation between the most recent occurrences of actions from Γ that are known to process p. This information is a DAG where the set of nodes is a subset of Γ. Figure 6 below gives an example of such a computation.

Theorem 4. *Given a tree-like distributed alphabet (Σ, dom) with tree $\mathcal{T}$, and a set $\Gamma \subseteq \Sigma$ of actions. The happens-before relation w.r.t. Γ can be computed by a Zielonka automaton where every process p maintains two DAGs of size $|\Gamma| + out(p)$, with $out(p)$ the out-degree of p in $\mathcal{T}$. Each update of the DAGs can be done in linear time in their size.*

Theorem 4 provides a rather simple way of reconstructing the happens-before relation with *finite* additional memory, and in a distributed way. Each synchronization action b will update the DAGs maintained by the processes executing b, by combining these DAGs and selecting the most recent knowledge about actions of Γ. As an example, suppose that processes p and q are neighbors in the tree, say, q is the father of p. As long as there is no synchronization involving both p and q, process p has the most recent knowledge about occurrences of Γ-actions belonging to processes in the subtree of p. As soon as some action synchronizes p and q, process q will be able to include p's knowledge regarding these actions, in its own knowledge.

5 Related work

This brief overview aimed at presenting the motivation behind distributed synthesis and how Mazurkiewicz trace theory can be used for monitoring concurrent programs. To conclude we point out some further related results.

Our discussion turned around distributed synthesis in a simple case where the program evolves without external actions from an environment. Synthesis of open systems, i.e., systems with an environment, is a more complex problem. Synthesis started as a problem in logics, with Church's problem asking for an algorithm to construct devices that transform sequences of input bits into sequences of output bits, so that the interaction conforms to a given logical formula [7]. Later, Ramadge and Wonham proposed the *supervisory control* formulation [42], where a plant and a specification are given; a controller should be designed such that its product with the plant satisfies the specification, no matter what the environment does. Synthesis is a particular case of control where the plant allows for every possible behavior. Rabin's result on the decidability of monadic second-order logic over infinite trees answered Church's question for MSO specifications [41].

Synthesis without environment. The problem of distributed synthesis without environment was first raised in the context of Petri nets. The task there is to

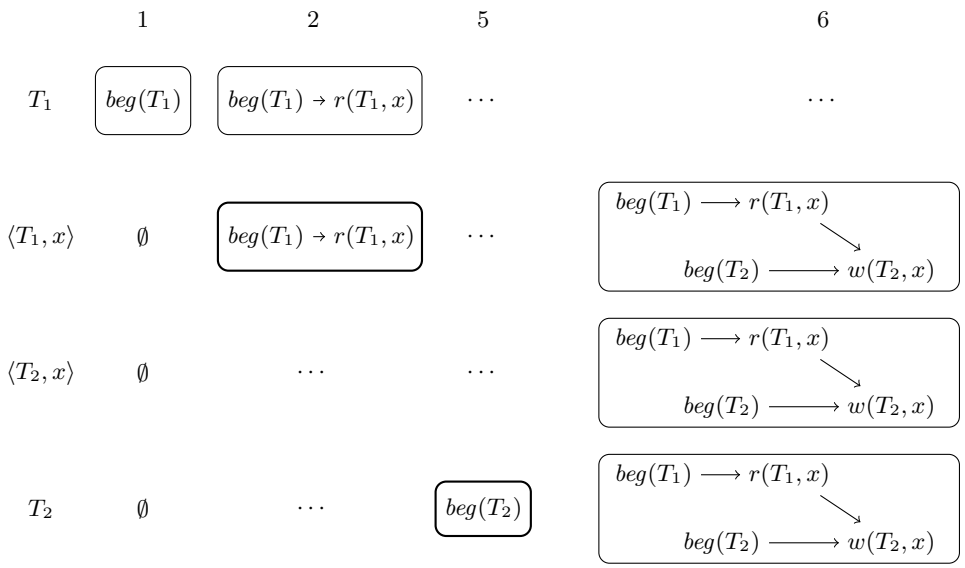


Fig. 6. Computing the partial order. Numbers in the first line stand for the program lines in Example 2. Dots mean that the information does not change.

decide whether an automaton, viewed as a graph, is isomorphic to the marking graph of a net. Ehrenfeucht and Rozenberg introduced the notion of regions, that determines how to decompose a graph for obtaining a net [11].

Zielonka's algorithm has been applied to solve the synthesis problem for models that go beyond Mazurkiewicz traces. One example is synthesis of communicating automata from graphical specifications known as *message sequence charts*. Communicating automata are distributed finite-state automata communicating over point-to-point FIFO channels. As such, the model is Turing powerful. However, if the communication channels are bounded, there is a tight link between execution sequences of the communicating automaton and Mazurkiewicz traces [22]. Actually we can handle even the case where the assumption about bounded channels is relaxed by asking that they are bounded for *at least one* scheduling of message receptions [18]. Producer-consumer behaviors are captured by this relaxed requirement.

Multiply nested words with various kinds of bounds on stack access [40,25,26], are an attractive model for concurrent programs with recursion, because of decidability properties and expressiveness. In [5] the model is extended to nested Mazurkiewicz traces and Zielonka's construction is lifted to this setting.

For runtime monitoring, a similar approach as ours is advocated in [43], that proposes an epistemic temporal logic for describing safety properties. A distributed implementation of a monitor for such properties is obtained, based on a variant of vector clocks.

Synthesis with environment. One way to lift Church's problem to the distributed setting was proposed by Pnueli and Rosner [39]. They showed that synthesis is decidable for very restricted architectures, namely pipelines. General undecidability was already known from the framework on multi-player games [38]. Subsequently, [28] showed that pipelines are essentially the only decidable case. Some interesting extensions of Pnueli and Rosner's setting have been considered in [24,13,15].

Alternatively, a distributed formulation of Church's problem can be formulated in Ramadge and Wonham's supervisory control setting. This problem, when plants and controllers are Zielonka automata, has been considered in [14,29,17,34]. In this formulation, local controllers exchange information when synchronizing on shared actions. In this way, the arguments for undecidability based on hidden information, as in [39,38], do not apply. Decidability of distributed control for Zielonka automata is still open. It is known though that this formulation admits more decidable architectures: the control problem for local parity specifications is decidable over acyclic architectures [34], thus in cases where Pnueli and Rosner's model is undecidable.

Yet another distributed version of Ramadge and Wonham's problem is considered in [9], this time for Petri nets. The problem is to compute local controllers that guarantee basic properties like e.g. deadlock avoidance. The limitation of this approach is that the algorithm may fail to find local controllers, although they exist.

Game semantics and asynchronous games played on event structures are introduced in [32]. Such games are investigated in [21] from a game-theoretical viewpoint, showing a Borel determinacy result under some restrictions.

Verification. As we already mentioned, automated verification of concurrent systems encounters major problems due to state explosion. One particularly efficient technique able to addresses this problem is known as *partial order reduction* (POR) [20,37,44]. It consists of restricting the exploration of the state space by avoiding the execution of similar, or equivalent runs. The notion of equivalence of runs used by POR is based on Mazurkiewicz traces. The efficiency of POR methods depends of course on the precise equivalence notion between executions. Recent variants, such as dynamic POR, work without storing explored states explicitly and aim at improving the precision by computing additional information about (non)-equivalent executions [1].

There are other contexts in verification where analysis gets more efficient using equivalences based on Mazurkiewicz traces. One such setting is counter-example generation based on partial (Mazurkiewicz) traces instead of linear executions [6]. Previous work connecting concurrency issues and Mazurkiewicz trace theory concerns atomicity violations [12], non-linearizability and sequential inconsistency [3].

Acknowledgments. Very special thanks to Jérôme Leroux, Gabriele Puppis and Igor Walukiewicz for numerous comments on previous versions of this paper.

References

1. P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *POPL'14*, pages 373–384. ACM, 2014.
2. S. Akshay, I. Dinca, B. Genest, and A. Stefanescu. Implementing realistic asynchronous automata. In *FSTTCS'13*, LIPIcs, pages 213–224. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
3. R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *LICS'96*, pages 219–228. IEEE, 1996.
4. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
5. B. Bollig, M.-L. Grindei, and P. Habermehl. Realizability of concurrent recursive programs. In *FoSSaCS'09*, LNCS, pages 410–424. Springer, 2009.
6. P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV'13*, LNCS, pages 951–967. Springer, 2013.
7. A. Church. Logic, arithmetics, and automata. *Proceedings of the International Congress of Mathematicians*, 1962.
8. R. Cori, Y. Métivier, and W. Zielonka. Asynchronous mappings and asynchronous cellular automata. *Information and Computation*, 106:159–202, 1993.
9. P. Darondeau and L. Ricker. Distributed control of discrete-event systems: A first step. *Transactions on Petri Nets and Other Models of Concurrency*, 6:24–45, 2012.
10. V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.
11. A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures: Parts i and ii. *Acta Informatica*, 27(4):315–368, 1989.
12. A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *CAV'08*, LNCS, pages 52–65. Springer, 2008.
13. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *LICS'05*, pages 321–330. IEEE, 2005.
14. P. Gastin, B. Lerman, and M. Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *FSTTCS'04*, LNCS, pages 275–286. Springer, 2004.
15. P. Gastin and N. Sznajder. Fair synthesis for asynchronous distributed systems. *ACM Transactions on Computational Logic*, 14(2):9, 2013.
16. B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Optimal Zielonka-type construction of deterministic asynchronous automata. In *ICALP'10*, LNCS, pages 52–63. Springer, 2010.
17. B. Genest, H. Gimbert, A. Muscholl, and I. Walukiewicz. Asynchronous games over tree architectures. In *ICALP'13*, LNCS, pages 275–286. Springer, 2013.
18. B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Inf. Comput.*, 204(6):920–956, 2006.
19. B. Genest and A. Muscholl. Constructing Exponential-Size Deterministic Zielonka Automata. In *ICALP'06*, LNCS, pages 565–576. Springer, 2006.
20. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.
21. J. Gutierrez and G. Winskel. On the determinacy of concurrent games on event structures with infinite winning sets. *J. Comput. Syst. Sci.*, 80(6):1119–1137, 2014.

22. J. G. Henriksen, M. Mukund, K. N. Kumar, M. Sohoni, and P. S. Thiagarajan. A Theory of Regular MSC Languages. *Inf. Comput.*, 202(1):1–38, 2005.
23. S. Krishna and A. Muscholl. A quadratic construction for Zielonka automata with acyclic communication structure. *Theoretical Computer Science*, 503:109–114, 2013.
24. O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *LICS'01*, pages 389–398. IEEE, 2001.
25. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS'07*, pages 161–170. IEEE, 2007.
26. S. La Torre and G. Parlato. Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width. In *FSTTCS'12*, LIPIcs, pages 173–184. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
27. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Operating Systems*, 21(7):558–565, 1978.
28. P. Madhusudan and P. Thiagarajan. Distributed control and synthesis for local specifications. In *ICALP'01*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001.
29. P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In *FSTTCS'05*, LNCS, pages 201–212. Springer, 2005.
30. F. Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.
31. A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.
32. P.-A. Melliès. Asynchronous games 2: The true concurrency of innocence. *TCS*, 358(2-3):200–228, 2006.
33. M. Mukund and M. A. Sohoni. Keeping track of the latest gossip in a distributed system. *Distributed Computing*, 10(3):137–148, 1997.
34. A. Muscholl and I. Walukiewicz. Distributed synthesis for acyclic architectures. In *FSTTCS'14*, LIPIcs, pages 639–651. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.
35. M. Odersky and T. Rompf. Unifying functional and object-oriented programming with Scala. *Communications of the ACM*, 57(4):76–86, 2014.
36. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2010.
37. D. A. Peled. All from one, one for all: on model checking using representatives. In *CAV'93*, LNCS, pages 409–423. Springer, 1993.
38. G. L. Peterson and J. H. Reif. Multi-person alternation. In *FOCS'79*, pages 348–363. IEEE, 1979.
39. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS'90*. IEEE, 1990.
40. S. Qadeer and J. Rehof. Context-bounded model-checking of concurrent software. In *TACAS'05*, LNCS, pages 93–107. Springer, 2005.
41. M. O. Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, Providence, RI, 1972.
42. P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(2):81–98, 1989.
43. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Decentralized runtime analysis of multithreaded applications. In *International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE, 2006.
44. A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, number 483 in LNCS, pages 491–515, 1991.
45. W. Zielonka. Notes on finite asynchronous automata. *RAIRO–Theoretical Informatics and Applications*, 21:99–135, 1987.