On Distributed Monitoring and Synthesis

Anca Muscholl(✉)

LaBRI, University of Bordeaux, Talence, France
anca@labri.fr

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 Java for example, are based on threads, shared-memory, and locking mechanisms that guard access to common data. More recent paradigms, such as the reactive programming model of Erlang [2] and Scala/Akka [1,3] replace shared memory by asynchronous message passing, where sending a message is non-blocking.

In all these concurrent frameworks, writing reliable software is a big challenge because programmers tend to think about code mostly in a sequential way, and have difficulties in overviewing all possible interleavings of executions by different entities. For the same reason, formal verification and analysis of concurrent programs is very challenging. 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. Formal 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 verification of programs requires as a pre-requisite a clear mathematical model for programs. Usually, verification of sequential programs starts 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 crucial pieces in automata-based verification and synthesis.

Synthesis means to translate a specification into a program that conforms with the specification, and thus can provide solutions that are correct by construction. Synthesis of *reactive systems*, that is of systems that interact with an environment, started as a problem in logics. In the sixties, A. Church asked for an algorithm to construct devices that transform sequences of input bits into sequences of output bits in a way required by a logical formula [9]. Later,

© Springer International Publishing Switzerland 2015
A. Beckmann et al. (Eds.): CiE 2015, LNCS 9136, pp. 54–62, 2015.
DOI: 10.1007/978-3-319-20028-6_6

Ramadge and Wonham proposed the *supervisory control* formulation [33], where a plant and a specification are given; a controller should be designed such that its product with the plant satisfies the specification. Thus, control means restricting the behavior of the plant. Synthesis is the particular case of control where the plant allows for every possible behavior. Rabin's result about the decidability of monadic second-order logic over infinite trees solved Church's question for MSO specifications [32].

When adding concurrency, the landscape of verification and automated synthesis becomes much more complicated. First, there is no canonical model for concurrent systems, simply because there can be very different kinds of interaction between processes. Compare for example multi-threaded shared memory systems and programs with asynchronous function calls. A second serious obstacle for developing automata-based verification techniques for concurrent systems is the lack of a general framework for distributed synthesis, and this even for systems without environment. The question whether a sequential specification can be turned into a distributed implementation over a given distributed architecture was first raised in the context of Petri nets. Ehrenfeucht and Rozenberg introduced the notion of regions to describe how to associate places of nets with states of a transition system [12].

Inspired by Petri nets, Mazurkiewicz proposed in the late seventies the theory of *Mazurkiewicz traces* [27], that we present in the next section. Within this theory, Zielonka's theorem [36] is a prime example for distributed synthesis. Our survey aims at introducing Mazurkiewicz traces and Zielonka's theorem, and describe how this theory can help to verify and design concurrent programs.

2 Mazurkiewicz Traces and Zielonka Automata

Mazurkiewicz traces [27] are one of the simplest formalisms able to describe concurrency. To define the model we fix 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 definition is that two dependent actions are always ordered and cannot be permuted. For instance, in a multi-threaded program all actions belonging to one thread must be ordered according to the program order. The actions of acquiring or releasing the same lock are also ordered, since a thread needs to wait that a lock is released before acquiring it. By contrast, independent actions can be permuted.

A by now classical way to express such dependencies is Lamport's *happens-before* partial order [25]. Mazurkiewicz traces capture this partial order through the dependence relation: from a linear execution $w = a_1 \dots a_n \in \Sigma^*$ a partial order $T(w) = \langle E, \preceq \rangle$ is defined, where:

- $E = \{a_1, \dots, a_n\}$ is the set of *events*, in one-to-one relation with the positions of w,
- $\preceq$ is the reflexive-transitive closure of $\{(a_i, a_j) \mid i < j,\ a_i\ D\ a_j\}$.

Partial orders $T(w)$ as above are called *Mazurkiewicz traces*.

Example 1. As an example consider a concurrent program with threads $T \in \mathcal{T}$ that have read/write access to shared variables $x \in X$. The dependence relation D over the alphabet of actions $\Sigma = \{r(T,x), w(T,x) \mid T \in \mathcal{T}, x \in X\}$ is given by $a\ D\ b$ if

- a, b are actions of the same thread T, or
- a, b access to the same variable $x \in X$ and at least one of them is a write.

This dependence relation simply describes that two actions are independent only if they belong to different threads. Moreover, if they access the same shared variable, then they must be both read actions.

From a language-theoretical viewpoint, traces are almost as attractive as words, and a rich body of results on automata and logics over finite and infinite traces exists, see the handbook [11]. One of the cornerstone results in Mazurkiewicz trace theory is based on a simple notion of finite-state distributed automata, Zielonka automata, that we present in the remaining of the section.

Informally, a Zielonka automaton [36] is a finite-state automaton with control distributed over several *processes* that synchronize on shared actions. 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*. Sharing of actions is defined through a fixed distributed action alphabet.

A *distributed action alphabet* on a finite set $\mathbb{P}$ of processes is a pair (Σ, dom), where Σ is a finite set of *actions* and $dom : \Sigma \to (2^{\mathbb{P}} \setminus \emptyset)$ is a *location function*. The location $dom(a)$ of action a comprises all processes that synchronize in order to perform this action. The location induces a natural dependence relation D over Σ by letting $a\ D\ b$ if $dom(a) \cap dom(b) \neq \emptyset$.

Example 2. As an example of distributed alphabet reconsider Example 1. A pair (Σ, dom) corresponding to the dependence relation D defined above can be obtained from the set of processes: $\mathbb{P} = \mathcal{T} \cup \{\langle T, x\rangle \mid T \in \mathcal{T}, x \in X\}$. Informally each thread represents a process, and there is a process for each pair $\langle T, x\rangle$, representing the cached value of x in thread T.

The location function defined below satisfies $a\ D\ b$ iff $\mathrm{dom}(a) \cap \mathrm{dom}(b) \neq \emptyset$:

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

Formally, 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:

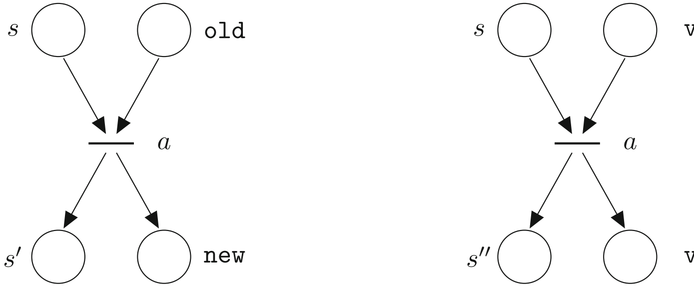
- a finite set S_p of (local) states with an initial state $s_p^{init} \in S_p$, for every process $p \in \mathbb{P}$,
- a partial transition relation $\delta \subseteq \bigcup_{a\in\Sigma} \left(\prod_{p\in dom(a)} S_p \times \{a\} \times \prod_{p\in dom(a)} S_p\right)$.

As usual, an automaton is called deterministic if the transition relation is a (partial) function. 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 the processes having action a in their alphabet can exchange information via the execution of a. The following example illustrates this effect:

Example 3. The CAS 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 was swapped, and `false` otherwise.

We can view the `CAS` instruction 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 a of the form `y = CAS(T,x,old,new)` becomes a synchronization action between P_T and P_x with the following two transitions (represented for convenience as Petri net transitions; places on the left represent states of P_T, and on the right of P_x):



On the left side of the figure we have the case where the value of x is `old`, and on the right half when it is different from `old`. Notice that in state s' the value of y is `true`, whereas in s'', it is `false`.

For convenience, we abbreviate a tuple $(s_p)_{p \in P}$ of local states by s_P.

Notice that a Zielonka automaton can be seen as a usual finite-state automaton with the state set $S = \prod_{p \in \mathbb{P}} S_p$ 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 the 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 permutation of adjacent independent symbols.

3 Distributed Synthesis

A cornerstone result in the theory of Mazurkiewicz traces is a construction that transforms sequential automata into deterministic Zielonka automata, whenever

the language is trace-closed. This important result is one of the rare examples of distributed synthesis, next to Ehrenfeucht and Rozenberg's theory of regions.

Theorem 1 **([36]).** *For a given distributed alphabet* (Σ, dom)*, and a regular trace-closed language* $L \subseteq \Sigma^*$ *over* (Σ, dom)*, a deterministic Zielonka automaton* $\mathcal{A}$ *can be effectively constructed with* $L(\mathcal{A}) = L$*.*

The intricacy of Zielonka's construction is such that there has been a lot of work to simplify it and to improve its complexity, see e.g. [10,16,19,28]. The most recent construction produces deterministic Zielonka automata of size that is exponential only in the number of processes. It was shown in [16] that the construction is optimal 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, and checks whether a state is final in polynomial time as well.*

Besides a theoretical interest of having an algorithm constructing deterministic Zielonka automata, there is also a strong practical motivation, namely to monitor distributed programs or systems at runtime. Of course, monitoring a system offline is also possible, however it can be done only *a posteriori* or by a centralized monitor that requires additional communication. If we want to monitor a distributed system at runtime, we need a decentralized monitor. The idea is simple: we have some trace-closed, regular property ϕ that should be satisfied by every execution of the 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. Assuming that we model our program P by a Zielonka automaton $\mathcal{A}_P$, running monitor M, that is also a Zielonka automaton $\mathcal{A}_M$, amounts to build the usual product automaton on each process between $\mathcal{A}_P$ and $\mathcal{A}_M$.

It is worth noting that the properties one would like to monitor on distributed programs can be often expressed in terms of the partial order between specific events. To illustrate this, consider as an example the *race detection problem* for multi-threaded programs. Informally, a race occurs whenever there are conflicting accesses to the same shared variable without proper lock synchronization. Detecting races is important since executions with races may yield non-deterministic, unexpected behaviors. Two accesses to the same variable are called *conflicting*, if at least one of them is a write. A *race* is given by two conflicting accesses that are unordered in the happens-before relation. This relation is a dependence relation in terms of Mazurkiewicz traces, that orders the events of each thread and lock access operations for each lock. So a violation of the "no-race" safety property consists in monitoring for two unordered occurrences of such conflicting accesses.

The construction of a deterministic Zielonka automata for properties asking for the partial ordering between specific events is in fact very close to the critical part of all available proofs of Zielonka's theorem. This critical part is known as the *gossip automaton* [28], and the name reflects already its rôle: it computes what a process knows about the knowledge of other processes.

In general, the gossip automaton is already responsible for the exponential complexity of the Zielonka construction. Thus, an important practical question is whether the construction of the gossip automaton can be avoided, or at least simplified. As the theorem below shows, gossiping is not needed when the communication structure is hierarchical.

A distributed alphabet (Σ, dom) is called *acyclic* if all actions have unary or binary domains, and the following graph $G(\Sigma, \mathrm{dom})$ (called *communication graph*) is acyclic: the set of nodes of $G(\Sigma, \mathrm{dom})$ is the set $\mathbb{P}$ of processes and the set of edges is $\{(p, q) \mid \exists a \in \Sigma : \mathrm{dom}(a) = \{p, q\}\}$.

Theorem 3 ([22]). *Let (Σ, dom) be a distributed alphabet whose communication graph is acyclic. Then every regular, trace-closed language L over Σ can be recognized by a deterministic Zielonka automaton with $O(s^2)$ states per process, where s is the size of the minimal DFA for L.*

We need to stress that the practical use of Zielonka automata for e.g. monitoring properties does not depend exclusively on the efficiency of the constructions from the above theorems. Further properties are required for a monitoring automaton $\mathcal{A}_M$ besides determinism. A first requirement is that violations of the property to monitor should be detectable locally, i.e., by at least one thread. The reason is that local detection enables a thread to start some recovery actions, like rollback of a transaction and a new try. A Zielonka automaton $\mathcal{A}$ with this property is called *locally rejecting* [16]. More formally, each process p has a subset of states $R_p \subseteq S_p$, and an execution leads a process p into a state from R_p if and only if the causal past of p cannot be extended to a trace in $L(\mathcal{A})$. A second requirement is that the monitoring automaton should not block the monitored system $\mathcal{A}_P$. This can be achieved by asking that in every global state of $\mathcal{A}_M$ such that no process is a rejecting state, every action is enabled. A related discussion of desirable properties of Zielonka automata and on an implementation of the construction of [16] is reported in [5] (see also [34]).

4 Related Work

This brief overview aimed at presenting the motivation behind distributed synthesis and how Mazurkiewicz trace theory can be useful in this respect. In the following we point out some related results.

Synthesis. Zielonka's algorithm has been applied for solving 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 strong link between execution sequences of the communicating automaton and Mazurkiewicz traces [21]. Actually we can even handle even the case where the assumption about bounded channels is relaxed by asking that they are are bounded for *at least one* scheduling of message receptions [18]. Producer-consumer behaviors are captured by this second setting.

Multiply nested words with various bounds on stacks [23,24,31] are an attractive model for concurrent programs with recursion, because of their decidability properties and expressiveness. In [7] the model is extended to nested Mazurkiewicz traces and Zielonka's construction is lifted to this setting.

We do not survey here recent results on synthesis of open systems and control for Zielonka automata. The interested reader is referred to [14,15,17,26,29].

Verification. As we already mentioned, automated verification of concurrent systems encounters major problems due to state explosion. One particularly efficient technique able to addresses these problems is known as *partial order reduction* (POR) [20,30,35]. 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 the model of *Mazurkiewicz traces.* The efficiency of POR methods depends of course on the precise equivalence notion between executions. More recent methods such as dynamic POR work without storing explored states explicitly and aim at improving the precision by computing additional information about (non)-equivalent executions [4].

There are many other contexts in verification where analysis can be made more efficient using equivalences based on Mazurkiewicz traces. One such example is counter-example generation based on partial (Mazurkiewicz) traces instead of linear traces, as done in [8]. Another example is the detection of concurrency bugs such as atomicity violations [13], non-linearizability and sequential inconsistency [6].

References

1. Akka. http://akka.io/
2. Erlang programming language. http://www.erlang.org/
3. Scala programming language. http://www.scala-lang.org/
4. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM (2014)
5. Akshay, S., Dinca, I., Genest, B., Stefanescu, A.: Implementing realistic asynchronous automata. In: FSTTCS 2013, LIPIcs, pp. 213–224. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
6. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. In: LICS 1996, pp. 219–228. IEEE (1996)
7. Bollig, B., Grindei, M.-L., Habermehl, P.: Realizability of concurrent recursive programs. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 410–424. Springer, Heidelberg (2009)

8. Černý, P., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Tarrach, T.: Efficient synthesis for concurrency by semantics-preserving transformations. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 951–967. Springer, Heidelberg (2013)
9. Church, A.: Logic, arithmetics, and automata. In: Proceedings of the International Congress of Mathematicians, pp. 23–35 (1962)
10. Cori, R., Métivier, Y., Zielonka, W.: Asynchronous mappings and asynchronous cellular automata. Inf. Comput. **106**, 159–202 (1993)
11. Diekert, V., Rozenberg, G. (eds.): The Book of Traces. World Scientific, Singapore (1995)
12. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures: parts i and ii. Acta Informatica **27**(4), 315–368 (1989)
13. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008)
14. Gastin, P., Lerman, B., Zeitoun, M.: Distributed games with causal memory are decidable for series-parallel systems. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 275–286. Springer, Heidelberg (2004)
15. Gastin, P., Sznajder, N.: Fair synthesis for asynchronous distributed systems. ACM Trans. Comput. Log. **14**(2), 9 (2013)
16. Genest, B., Gimbert, H., Muscholl, A., Walukiewicz, I.: Optimal Zielonka-Type construction of deterministic asynchronous automata. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 52–63. Springer, Heidelberg (2010)
17. Genest, B., Gimbert, H., Muscholl, A., Walukiewicz, I.: Asynchronous games over tree architectures. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part II. LNCS, vol. 7966, pp. 275–286. Springer, Heidelberg (2013)
18. Genest, B., Kuske, D., Muscholl, A.: A Kleene theorem and model checking algorithms for existentially bounded communicating automata. Inf. Comput. **204**(6), 920–956 (2006)
19. Genest, B., Muscholl, A.: Constructing exponential-size deterministic zielonka automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 565–576. Springer, Heidelberg (2006)
20. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. Form. Meth. Syst. Des. **2**(2), 149–164 (1993)
21. Henriksen, J.G., Mukund, M., Kumar, K.N., Sohoni, M., Thiagarajan, P.S.: A theory of regular MSC languages. Inf. Comput. **202**(1), 1–38 (2005)
22. Krishna, S., Muscholl, A.: A quadratic construction for Zielonka automata with acyclic communication structure. Theor. Comput. Sci. **503**, 109–114 (2013)
23. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS 2007, pp. 161–170. IEEE (2007)
24. La Torre, S., Parlato, G.: Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width. In: FSTTCS 2012, LIPIcs, pp. 173–184. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Oper. Syst. **21**(7), 558–565 (1978)
26. Madhusudan, P., Thiagarajan, P.S., Yang, S.: The MSO theory of connectedly communicating processes. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 201–212. Springer, Heidelberg (2005)
27. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. DAIMI report PB 78, Aarhus University, Aarhus (1977)

28. Mukund, M., Sohoni, M.A.: Keeping track of the latest gossip in a distributed system. Distrib. Comput. **10**(3), 137–148 (1997)
29. Muscholl, A., Walukiewicz, I.: Distributed synthesis for acyclic architectures. In: FSTTCS 2014, LIPIcs, pp. 639–651. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014)
30. Peled, D.A.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
31. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
32. Rabin, M.O.: Automata on Infinite Objects and Church's Problem. American Mathematical Society, Providence (1972)
33. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**(2), 81–98 (1989)
34. Stefanescu, A.: Automatic synthesis of distributed transition systems. Ph.D. thesis, Universität Stuttgart (2006)
35. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
36. Zielonka, W.: Notes on finite asynchronous automata. RAIRO-Theor. Inf. Appl. **21**, 99–135 (1987)