Automated Synthesis: a Distributed Viewpoint

Anca Muscholl

LaBRI, University of Bordeaux, France

Abstract

Distributed algorithms are inherently hard to get right, and a major challenge is to come up with automated techniques for error detection and recovery. The talk will survey recent results on the synthesis of distributed monitors and controllers.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.1.3 Concurrent Programming

Keywords and phrases Synthesis, distributed programs

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.3

Category Invited Paper

1 Overview

Distributed applications represent a significant part of our everyday life. Typically, our personal data are stored on remote distributed servers, data management relies on remote applications, data-intensive computations are performed on computer clusters, etc. Since distributed applications are deployed at increasingly large scale, they have to be reliable and robust, satisfying stringent correctness criteria. But distributed programs are hard to get right, and errors can be very subtle[1].

Formal methods, and in particular model-checking, can produce rigorous, automated reliability proofs for hardware and software systems. The area has always had special interest in distributed applications, for two reasons. First, distributed programs are error prone, because programmers have to consider all possible effects induced by different schedulings of events. Second, testing, which is widely used for certifying sequential programs, tends to have low coverage in the distributed setting, because bugs are usually difficult to reproduce: they may happen under very specific thread schedules, and the likelihood of taking such corner-case schedules may be very low. As a consequence, automated verification techniques represent a crucial support in the development of reliable distributed applications.

Many recent advances in formal methods are motivated by a substantial increase in deploying distributed applications like robot-assisted systems, cloud-based services, etc. However, formal verification of distributed programs is still extremely challenging. The first challenge is *scalability*: automated verification, such as model-checking and other traditional exploration techniques, can only handle small instances of concurrent programs, mostly because of the very large number of possible states and of the asynchronicity of concurrent executions. The second challenge is *parametrization*: distributed protocols are usually designed to work for an arbitrary number of concurrent processes, which means that verification is required for an arbitrary number of processes. The third challenge is related to

[1] A well-known quotation of Leslie Lamport says "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

contextual assumptions: distributed programs are often designed under specific assumptions about synchronicity, failure behaviors, etc., which are often either difficult to model or even hard to formalize at an informal level.

Runtime verification. In traditional *model-checking*, distributed programs and algorithms are modeled as a set of communicating finite-state processes, and the correctness of all possible executions is specified in some temporal logic. If at all doable, model-checking usually requires finite abstractions and clever heuristics, like partial-order reduction [9, 17, 21], in order to cope with the scalability problem. *Runtime verification* is an appealing alternative method to traditional exhaustive exploration, situated somewhere between testing and model-checking (see e.g. the surveys [12, 10] and the dedicated conference RV running for more than 15 years). Runtime verification is about monitoring program executions against formal specifications, and is a support for error detection. But designing distributed monitors from a given specification is significantly more difficult than for sequential programs, since for sake of feasibility, let alone efficiency, the monitoring information has to be computed by a distributed algorithm using only the communication means provided by the execution of the monitored program. The challenge here is to come up with algorithms constructing efficient monitors on various communication architectures, and with a reasonable computational overhead.

Synthesis. In *reactive synthesis* the goal is to automatically derive from some given specification a reactive program, that is, a program interacting continuously with its environment. The notion of reactive synthesis goes back to work by Church in the 60s [1], and it has given birth to a beautiful and rich theory of automata, logics and games of infinite duration (see e.g. [11, 20]). The games paradigm captures the idea of interaction between a program and its environment, and computing a winning strategy amounts to construct a program that behaves correctly in any possible situation. Nowadays the efforts to translate the theory into practical algorithms are considerable: a growing number of synthesis tools are available, and there is the dedicated annual tool competition Syntcomp@CAV.

The reactive synthesis of distributed programs is a particularly attractive problem because distributed programs are notoriously hard to get right. The problem has been considered already in the early 90s, starting with a model proposed by Pnueli and Rosner [18]: processes communicate via shared variables synchronously, according to a fixed communication architecture. Each process has a partial view about the global state, since its knowledge is limited to its input variables. The partial knowledge of processes has as consequence that the distributed synthesis problem is decidable only for very restricted communication architectures, without so-called information forks [5], basically for pipelines only [14].

2 Going distributed

The main distinguishing feature of runtime verification is that it is performed at runtime, by continuously checking program executions against formal specifications. This opens the possibility to use it as support for error diagnosis, and also to deploy correction mechanisms when an illegal behavior of the program is detected.

The traditional runtime verification approach is to construct a monitor from a given property. The monitor is then used to check e.g. the current execution of the system. In other words, the monitor reads the finite trace incrementally and it is supposed to notify if an error occurs. While in model-checking all executions of the system are considered (often with emphasis on infinite executions) runtime verification deals with a single execution at a time and does not require complete knowledge about the program or system. In other terms, runtime verification can be applied on black-box systems for which no model is available. It also represents a lightweight method regarding complexity, since from a theoretical viewpoint monitoring single traces simply corresponds to the word problem. The main issue in runtime monitoring is the complexity of the monitor, i.e., its memory and computation time requirements, as a monitor runs in parallel with the system and should not slow it down too much.

Designing distributed monitors from a given specification is far more challenging than for sequential programs, as the monitoring information has to be computed by a distributed algorithm. A straightforward, but impractical, way to monitor a distributed program is to synchronize the relevant components and to inquire about their states. A much better way to do this is to build local monitors that deduce the required information by recording and exchanging suitable information using mainly the communication means provided by the execution of the monitored program. So the main point about distributed monitoring is to avoid adding synchronization in the program, since this usually impacts negatively on the overall performance [19].

A very successful example for the automatic generation of distributed monitors has its roots in the theory of Mazurkiewicz traces [15]. Within this theory, Zielonka's theorem [22] is a prime example of synthesis of distributed, finite-state monitors. A Zielonka automaton is in essence the parallel composition of finite-state processes that synchronize over shared actions. Many researchers contributed to simplify the construction and to improve its complexity. The most recent construction [7] produces deterministic distributed monitors of size that is exponential in the number of processes (and polynomial in the size of a DFA for the monitoring property). It is very challenging to try to adapt Zielonka's construction to models involving other types of synchronization. Generally speaking, constructing distributed monitors for specific architectures, and under specific conditions, like robustness under failures, is an important open problem.

Reactive synthesis. Reactive synthesis lays the ground for error recovery, once errors have been detected through monitoring. It can support for example the design of controllers that are in charge of taking appropriate recovery steps, depending on the failure cause. Zielonka automata turned out to be a promising alternative for reactive synthesis as well. The crucial difference compared to the model of Pnueli-Rosner is that the synchronization of processes in a Zielonka controller entails an exchange of information about the local knowledge. So although information is still partial, it is in some sense complete, according to the communication architecture. The decidability status of reactive distributed synthesis in this model is still open, but the problem is known to be decidable when the communication structure is acyclic [8, 16]. This result, together with some decidability results for restricted communication [6, 13], makes the Zielonka version of distributed synthesis more attractive than the one of Pnueli-Rosner.

Recently, Petri games were proposed as another formulation of the distributed reactive synthesis. These games are played on Petri nets, with places that are either controllable (belonging to the system) or uncontrollable (belonging to the environment). As for Zielonka automata, the decidability status of the synthesis problem is open. The synthesis problem was shown to be decidable whenever there is a single environment token, and a bounded number of system tokens [4]. Petri net games are an interesting alternative to Zielonka automata, but the precise relation between the two models remains to be explored.

Negotiation diagrams [2], a concurrency model inspired by workflow nets, can offer another fruitful setting for the synthesis problem. A negotiation diagram describes a distributed system with a fixed set of sequential processes. The diagram is composed of "atomic negotiations", each one involving some subset of processes. An atomic negotiation starts when all its participants are ready to engage in it, and concludes with the selection of one out of a fixed set of possible outcomes; for each participant process, the outcome determines which atomic negotiations the process is willing to engage in at the next step. In essence, deterministic negotiations are stateless Zielonka automata, and it turns out that their analysis is algorithmically much easier in some cases. For example, sound diagrams have polynomial-time algorithms for Mazurkiewicz-invariant static analysis problems [3]. Negotiations are very likely to be an attractive setting for distributed synthesis as well.

References

1 Alonzo Church. Logic, arithmetics, and automata. *Proceedings of the International Congress of Mathematicians*, 1962.

2 Javier Esparza and Jörg Desel. On negotiation as concurrency primitive. In *Proc. CONCUR'13*, volume 8052 of *LNCS*, pages 440–454. Springer, 2013.

3 Javier Esparza, Anca Muscholl, and Igor Walukiewicz. Static analysis of deterministic negotiations. In *Proc. LICS'17*, pages 1–12. IEEE Computer Society, 2017.

4 Bernd Finkbeiner and Ernst-Ruediger Olderog. Petri games: Synthesis of distributed systems with causal memory. In *GandALF'14*, EPTCS, pages 217–230, 2014.

5 Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *LICS'05*, pages 321–330. IEEE Computer Society, 2005.

6 Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 275–286. Springer, 2004.

7 Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. Optimal Zielonka-type construction of deterministic asynchronous automata. In *ICALP'10*, volume 6199 of *LNCS*, pages 52–63. Springer, 2010.

8 Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. Asynchronous games over tree architectures. In *ICALP'13*, volume 7966 of *LNCS*, pages 275–286. Springer, 2013.

9 Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

10 Klaus Havelund and Giles Reger. Runtime verification logics a language design perspective. In *Models, Algorithms, Logics and Tools – Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday.*, volume 10460 of *LNCS*. Springer, 2017.

11 Orna Kupferman and Moshe Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245–263, 1999.

12 Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

13 P. Madhusudan, P. S. Thiagarajan, and S. Yang. The MSO theory of connectedly communicating processes. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 201–212. Springer, 2005.

14 P. Madhusudan and P.S. Thiagarajan. Distributed control and synthesis for local specifications. In *ICALP'01*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001.

15 Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

16 Anca Muscholl and Igor Walukiewicz. Distributed synthesis for acyclic architectures. In *FSTTCS'14*, volume 29 of *LIPIcs*, pages 639–651. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014.

17 Doron A. Peled. All from one, one for all: on model checking using representatives. In *CAV'93*, LNCS, pages 409–423. Springer, 1993.

18 Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS'90*, pages 746–757. IEEE Computer Society, 1990.

19 Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In *International Conference on Software Engineering (ICSE'04))*, pages 418–427. IEEE Computer Society, 2004.

20 Wolfgang Thomas. Church's problem and a tour through automata theory. In *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Springer, 2008.

21 Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, number 483 in LNCS, pages 491–515, 1991.

22 W. Zielonka. Notes on finite asynchronous automata. *RAIRO–Theoretical Informatics and Applications*, 21:99–135, 1987.