Who is afraid of Model Checking Distributed Algorithms?[⋆]

Igor Konnov, Helmut Veith, and Josef Widder

TU Wien
Formal Methods in Systems Engineering Group

Abstract. While distributed algorithms is a highly active area, and the correctness of distributed algorithms is usually based on very subtle mathematical arguments, there have been very limited efforts to achieve automated verification of distributed algorithms. In this note we discuss the major technical obstacles and methodological challenges. Our hope is that the collection of issues collected in this position paper will isolate the most urgent and open research questions to brave future researchers.

1 Introduction

Distributed algorithms are both important and notoriously difficult. It is therefore quite surprising that there have been quite limited efforts for model checking of distributed algorithms. Only some distributed algorithms have been formally verified in the literature. Typically, these papers have addressed specific algorithms in a fixed computation environment, for instance, [Lam11], [Mül98], [WEG$^+$04], [Hen05], or [TS11]. However, model checking of distributed algorithms has not been approached in a structured way.

A major obstacle is the *methodological gap* between the theory of distributed computing and the practice of designing and verifying the correctness of reliable distributed systems. In this position paper we discuss the two major reasons for this gap: *the formalization problem* and the *verification problem.*

Regarding formalization, we note that distributed computing models are traditionally represented in natural language, and algorithms are described in pseudo code. The classical approach to distributed algorithms is thus not very formal, and it is not always clear under which circumstances a given distributed algorithm actually is correct.

Even in the presence of a formal model, automated verification is difficult for several reasons: First, the assumptions on timing behavior and fault-tolerance result in systems with large state spaces and large amounts of non-determinism. Second, many distributed algorithms are parameterized not only in the number of participating processes (the system size) but also in the assumed upper bound on the number of faults.

⋆ Supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF), and by the Vienna Science and Technology Fund (WWTF) grant PROSEED.

In the rest of this paper, we shall explore the two problem areas in more detail. Our observations are based on recent work by the authors on abstraction-based model checking of fault-tolerant threshold-based algorithms [JKS+]. It is characteristic of the situation that our work was only possible with distributed algorithms experts on the team.

2 The Formalization problem

In the literature, the vast majority of distributed algorithms is described in pseudo code, for instance, [ST87,ADGFT06,WBG+12]. The intended semantics of the pseudo code is folklore knowledge among the distributed computing community. Researchers who have been working in this community have intuitive understanding of keywords like "send", "receive", or "broadcast". For instance, inside the community it is understood that there is a semantical difference between "send to all" and "broadcast" in the context of fault tolerance. Moreover, the constraints on the environment are given in a rather informal way. For instance, in the authenticated Byzantine model [DLS88], it is assumed that faulty processes may behave arbitrarily. At the same time, it is assumed that there is some authentication service. In conclusion, it is thus assumed that faulty processes send any messages they like, *except* ones that look like messages sent by correct processes. However, inferring this kind of information about the behavior of faulty processes is a very intricate task.

At the bottom line, a close familiarity with the distributed algorithms community is required to adequately model a distributed algorithm in preparation of formal verification. When the essential conditions are hidden between the lines of a research paper, then one cannot be sure to actually verify the algorithm actually intended by the original authors. With the current state of the art, we are thus forced to do *verification of a moving target.*

We conclude that there is need for a versatile specification language which can express distributed algorithms along with their environment. Such a language should be natural for distributed algorithms researchers, but provide unambiguous and clear semantics. Since distributed algorithms come with a wide range of different assumptions, the language has to be easily configurable to these situations. Unfortunately, most checkers do not provide sufficiently expressive languages for this task. Thus, it is hard for researchers from the distributed computing community to use these tools out of the box. Although distributed algorithms are usually presented in a very compact form, the "language primitives" (of pseudo code) are used without consideration of implementation issues and computational complexity. For instance sets and operations on sets are often used as they ease presentation of concepts to readers, although fixed size vectors would be sufficient to express the algorithm and more efficient to implement. Besides, it is not unusual to assume that any local computation on a node can be completed within one step. Another example is the handling of messages. For instance, how a process stores the messages that have been received in the past is usually not explained in detail. At the same time, quite complex operations are performed on this information.

3 The Verification problem

Let us now suppose for the sake of argument that the Formalization Problem is solved for a certain class of distributed algorithms. We will now address the major issues that have to be considered in order to solve the verification problem.

Degrees of concurrency. While verification of concurrent executions is a classic problem considered in model checking, the degrees of concurrency that are actually considered in the model checking literature are quite simple. For instance, there is a lot of work on verification of executions with either purely asynchronous interleaving or purely synchronous executions as found in hardware. However, due to well-known impossibility results [FLP85], the distributed computing community was forced to study many different types of executions [DDS87,DLS88] that lie between these extremal cases. They are expressed by non-trivial fairness conditions. For instance, one often assumes partially synchronous processes, that is, in each run there is some integer Φ such that between two steps of some process, any other *correct* process takes at most Φ steps. In our verification efforts, we have to address these complex interleavings.

Unbounded data types. Distributed algorithms are usually designed with idealized data structures, in particular unbounded integer variables and unbounded arrays. In contrast to real life programming languages, the basic data types of distributed algorithms are thus not bounded. For instance, many distributed algorithms proceed in rounds that are numbered increasingly, and the information that is exchanged in these rounds has to be stored. As often the number of rounds required for termination is not bounded, arrays of unbounded size and unbounded integers are crucial for the correctness of these distributed algorithms [DLS88]. From the verification point of view, from the very beginning, we have to use decision procedures and abstraction to verify the algorithms. This limits the direct use of tools like SPIN [Hol03] whose input language provides primitives with semantics close to those used in distributed algorithms, but supports only finite data structures.

Parameterization in multiple parameters. Most distributed algorithms are designed to work in systems with an arbitrary number n of processes. Thus, model checking of distributed algorithms is essentially parameterized model checking. An algorithm is correct if the systems $P(n) \parallel P(n) \parallel \cdots \parallel P(n)$ composed of n processes $P(n)$ are correct for all n. Note that the number of processes present in the system is often used in the algorithm description itself, i.e., the algorithm "knows" this number. Thus, $P(n)$ and $P(n+1)$ are not semantically equivalent. For fault tolerant distributed algorithms the situation becomes even worse: Not only the system size, but also the assumed number t of faulty processes is a parameter. Hence we have to deal with systems of the form $P(n, t) \parallel P(n, t) \parallel \cdots \parallel P(n, t)$, i.e., with multiple parameters.

A big open problem is the identification of cut-offs, i.e., of reductions which enable to infer the correctness of the *algorithm* from the correctness of some *small* system instances.

Contrast to concurrent programs. Although verification of concurrent programs is a hot topic, the goals and problems of distributed algorithm design are different to those of concurrent algorithms. The central topics in model checking of concurrent systems is the absence of design or implementation faults. For instance, one wants to detect race conditions, cf. [HJM04]. At the same time, race conditions are not of much interest in distributed algorithms as processes are considered to run on different computers in a distributed system. In contrast, distributed algorithms usually try to make distributed systems more resilient against partial failure of the system. These failures are usually considered to be outside of the control of the designer, for instance, faults that are due to adverse environments. In such system, establishing global properties within loosely coupled processes is the central goal. Thus, sophisticated model checking and testing algorithms developed for concurrent programs are usually tackling questions that are not central to distributed algorithms.

Continuous time. Many distributed algorithms — most notably for clock synchronization [ST87,WS07] — are given in computational models that are based on continuous time. While some model checkers support continuous time, they can only handle relatively simple systems which are not able to handle the other modeling questions discussed in this section.

Deployment. Although distributed algorithms are designed in a parameterized way, concrete implementations are likely to have bounds on data structures, network bandwidth etc. It is a nontrivial question how to verify the correctness of such an implementation against the abstract algorithm. Evidently, expertise from both communities is needed to address this question.

4 Conclusions

In this note we presented our view on the most pressing questions regarding model checking of distributed algorithms. We discussed the issues with respect to two main areas, namely, the Formalization Problem and the Verification Problem, and find that there is plenty of exciting future work for researchers in distributed algorithms and formal verification.

References

ADGFT06. Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Consensus with Byzantine failures and little system synchrony. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 147–155, Washington, DC, USA, 2006. IEEE Computer Society.

DDS87. Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

DLS88. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

FLP85. Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

Hen05. Martijn Hendriks. Model checking the time to reach agreement. In *FORMATS*, volume 3829 of *LNCS*, pages 98–111, 2005.

HJM04. Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.

Hol03. Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

JKS$^+$. Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Abstraction-based model checking of fault-tolerant threshold-based algorithms. (manuscript in preparation).

Lam11. Leslie Lamport. Byzantizing paxos by refinement. In *DISC*, 2011. `http://research.microsoft.com/en-us/um/people/lamport/pubs/web-byzpaxos.pdf`.

Mül98. Olaf Müller. I/O automata and beyond: Temporal logic and abstraction in isabelle. In *TPHOLs*, pages 331–348, 1998. `http://dx.doi.org/10.1007/BFb0055145`.

ST87. T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

TS11. Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5–6):341–358, 2011.

WBG$^+$12. Josef Widder, Martin Biely, Günther Gridling, Bettina Weiss, and Jean-Paul Blanquart. Consensus in the presence of mortal byzantine faulty processes. *Distributed Computing*, 24(6):299–321, 2012.

WEG$^+$04. Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy A. Lynch. Using simulated execution in verifying distributed algorithms. *Int. J. Softw. Tools Technol. Transf.*, 6:67–76, July 2004.

WS07. Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Distributed Computing*, 20(2):115–140, August 2007.