Challenges in Model Checking of Fault-tolerant Designs in TLA$^+$

Igor Konnov, Helmut Veith, and Josef Widder

TU Wien (Vienna University of Technology)

Abstract. Although, historically, fault tolerance is connected to safety-critical systems, there has been an increasing interest in fault tolerance in mainstream application such as the cloud. There is a need for formal specification and verification of industrial fault-tolerant designs, since they integrate, in a non-trivial way, the ideas from distributed algorithms, whose correctness is usually based on very subtle mathematical arguments. More and more fault-tolerant designs are formally specified in TLA$^+$. Based on our experience in model checking of fault-tolerant distributed algorithms, we propose a research agenda towards model checking of fault-tolerant designs in TLA$^+$.

1 Introduction

As many modern applications are running in the cloud, the user requests are processed by tens of thousands of commodity computers in data centers operated by, e.g., Amazon, Google, and Facebook. When such a large number of computers is involved, faults become a norm rather than an exception [31]. As demonstrated by the recent outage of the Amazon Elastic Compute Cloud (EC2), which brought down about 70 sites [1], an unlucky sequence of faults and software bugs can make a data center unavailable. To make systems more reliable, we have to deal with two kinds of undesired phenomena: *faults and bugs*.

By a *fault* we understand a situation that is not controllable by a system designer, e.g., a computer crash or reboot, disk corruption, network disconnect, or power outage. A *bug* is a manifestation of an incorrect protocol design or software implementation, e.g., too small timeout, race condition, deadlock, buffer overflow, or memory leak. Historically, faults are dealt with by fault-tolerant distributed algorithms, which are mainly designed for safety-critical systems [21], e.g., plant or vehicle control systems. As the mentioned examples suggest, more and more mainstream IT applications adopt fault-tolerance for economic reasons.

The existing methods to show the correctness — i.e., freedom of bugs — of fault-tolerant systems require deep mathematical background and are labor-intensive, e.g., paper-and-pencil proofs or computer-assisted proofs. In mainstream applications, the implementations and designs change at much shorter

Supported by the Austrian National Research Network S11403 and S11405 (RiSE), and project P27722 (PRAVDA) of the Austrian Science Fund (FWF).

intervals than the ones in safety-critical systems. Hence, the labor-intensive verification methods are of limited use. For these new applications of fault tolerance, we need domain-specific verification methods having a high automation degree.

The de-facto approach to implement fault-tolerant systems is to take a fault-tolerant distributed algorithm from the literature and try to implement its algorithmic idea faithfully. Transferring a distributed algorithm, which is typically given in pseudo code, to a running system can be ambiguous and ad-hoc. The Paxos [23] approach to replicated state machines has been transferred into implementations such as PBFT [7], Chubby [8], Zyzzyva [22], ZAB [18], EPaxos [30], RAFT [33]. As admitted by the specialists who did such implementations, "*Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations — some published in the literature and some not.*" [8]. Even if a fault-tolerant distributed algorithm has been shown to be correct in theory, due to this ad-hoc approach, the correctness of the theoretical algorithm does not imply the correctness of the *implementation*. For instance, the developers of Apache ZooKeeper introduced conceptual bugs to the ZAB protocol while "optimizing" it [27].

Specification frameworks. Anticipating these problems, Lamport introduced TLA in the 80ies (and later TLA$^+$). Its precise semantics replaces the pseudo code that is predominant in distributed algorithm literature. TLA$^+$ by Leslie Lamport [24] and I/O Automata (IOA) by Nancy Lynch [26] are specification and refinement frameworks that are intended to span the design and implementation process. Both were devised when automated verification was out of reach, and were thus designed with manual proofs in mind. Still, industry is starting to adopt these frameworks: TLA$^+$ was used at Intel [5] and Amazon [32], and IOA was used at Oracle [25]. Researchers are using TLA$^+$ and IOA to document fault-tolerant protocols: PBFT [7] is specified in IOA, and DiskPaxos [13], EPaxos [30], and RAFT [33] are specified in TLA$^+$.

Both IOA and TLA$^+$ offer basic tool support for verification: The Tempo toolset [2] for IOA interfaces with the proof assistant PVS [34] and the model checker UPPAAL [6], whereas the TLA$^+$ Toolbox [24] offers the proof assistant TLAPS and the model checker TLC [36]. While development of the Tempo toolset seems to be frozen, there is a constant improvement of TLA$^+$. The only existing tool for automatic verification, the TLC model checker, is used to check *simplified and finite-state* designs [32]: one has to fix the number of components in a system (to a typically very small number), one has to fix the domain of variables, etc. The core algorithm of TLC uses explicit state enumeration.

Our goal is the development of state-of-the-art model checking tools for TLA$^+$. *Due to rising industrial interest in* TLA$^+$, *e.g., [32], and the availability of advanced model checking techniques, we believe in the timeliness of the effort.*

2 Challenges

From a very high-level perspective, we face the following challenges, when designing a model checker for fault-tolerant designs in TLA$^+$:

– *Rich language.* Specifications in TLA$^+$ are considerably more expressive than standard software: TLA$^+$ is untyped, it allows quantification over sets, comparison of cardinalities, and comparison and updates of the states of concurrent components.
– *Parameterized systems.* TLA$^+$ specifications should be checked for an unknown number of components (to verify systems of practical scale, such as in the cloud).
– *Verification beyond toy examples.* The practical examples include RAFT [33], DiskPaxos [13], EPaxos [30], ZAB [18], GFS, Niobe, Chain [14]. The TLA$^+$ specifications of RAFT, DiskPaxos, and EPaxos are available on the Web.

Due to the expressiveness of TLA$^+$ and parameterization of the realistic TLA$^+$ specifications, most of verification problems are undecidable. Model checking research, including our own work [17,19,20], shows that abstraction is a pragmatic approach to tackle practical instances of undecidable problems. To circumvent undecidability, abstraction exploits domain knowledge and features of the problem instances, e.g., counter abstraction [35,17] uses symmetry.

Our own research in the last years was a first step towards this agenda: we introduced new techniques for parameterized model checking of fault-tolerant distributed algorithms [17,19,20]. These techniques apply to algorithms that are parameterized in the number n of identical (symmetric) processes, among which at most t processes are faulty, and whose process code contains threshold guards like "`if received (ping) from at least n-t distinct processes`". Testing whether the number of messages is above the threshold $n - t$ is an algorithmic pattern, which is omnipresent in the literature. For example, in [20] we verified subtle fault-tolerant algorithms that exclusively use this pattern: asynchronous reliable broadcast and Byzantine agreement, non-blocking atomic commit, condition-based consensus, one-step consensus.

Industrial TLA$^+$ designs are instantiations of many algorithmic patterns from distributed algorithms (e.g., threshold guards, leader election, heartbeats). We suggest to exploit such patterns to automatically build abstractions of TLA$^+$ code and thus make the verification problem amenable for model checking. Similar to the collection of driver API rules in SLAM [3], we propose to collect TLA$^+$ patterns from the published TLA$^+$ code of fault-tolerant designs:

Challenge 1: Establishing the common repository of TLA$^+$ *patterns.*

Predicate abstraction [15], CEGAR [10], SLAM [4,3], and BLAST [16], have dramatically changed the landscape of software verification. We aim at transferring these techniques to the TLA$^+$ domain. The key challenge is to construct the predicate abstraction of a system step encoded in TLA$^+$. Software model checkers use logic decision procedures to do this for program statements. In general, to construct predicate abstraction, we need decision procedures for TLA$^+$ [28,29]. More specifically, to verify fault-tolerant designs in TLA$^+$, we need decision procedures for message sets, threshold guards, faults, resilience conditions, etc. [17,11]:

Challenge 2: Adjusting predicate abstraction for TLA$^+$ *specifications.*

Fault-tolerant distributed algorithms and fault-tolerant designs are parameterized in multiple dimensions, for instance: in the number of correct processes, in the number of faulty processes, in the number of input values in consensus, or in the set of potential ballots in Paxos. In [17], we manually combined (and generalized) data abstraction and counter abstraction to deal with message counters, process counters, and resilience conditions. Unfortunately, there is no single parameterized verification technique that fits all features of TLA$^+$ designs. For instance, the fault-tolerant algorithm from [9] requires us to reason both about rotating coordinators and message counting. While our work [17] deals with message counting, and the work on token rings deals with round-robin scheduling [12], there is no obvious way to compose these two techniques.

In order to verify the properties of practical fault-tolerant designs, one has to to decompose the parameterized verification problem into several subproblems to be addressed with dedicated parameterized model checking techniques:

Challenge 3: Compositional methods to integrate the spectrum of parameterized model checking techniques.

References

1. Amazon: Summary of the Amazon EC2 and Amazon RDS service disruption in the US East region (2011), `http://aws.amazon.com/message/65648/`
2. Archer, M., Lim, H., Lynch, N.A., Mitra, S., Umeno, S.: Specifying and proving properties of timed I/O Automata using Tempo. Design Autom. for Emb. Sys. 12(1–2), 139–170 (2008)
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. CACM 54(7), 68–76 (2011)
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: ACM SIGPLAN Notices. vol. 37, pp. 1–3. ACM (2002)
5. Batson, B., Lamport, L.: High-level specifications: Lessons from industry. In: Formal methods for components and objects. pp. 242–261. Springer (2003)
6. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing UPPAAL over 15 years. Softw., Pract. Exper. 41(2), 133–142 (2011)
7. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OSDI. vol. 99, pp. 173–186 (1999)
8. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: PODC. pp. 398–407. ACM (2007)
9. Charron-Bost, B., Schiper, A.: The heard-of model: computing in distributed systems with benign faults. Distributed Computing 22(1), 49–71 (2009)
10. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
11. Dragoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: VMCAI. LNCS, vol. 8318, pp. 161–181 (2014)
12. Emerson, E., Namjoshi, K.: Reasoning about rings. In: POPL. pp. 85–94 (1995)
13. Gafni, E., Lamport, L.: Disk paxos. Distributed Computing 16(1), 1–20 (2003)
14. Geambasu, R., Birrell, A., MacCormick, J.: Experiences with formal specification of fault-tolerant file systems. In: DSN. pp. 96–101. IEEE (2008)

15. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. LNCS, vol. 1254, pp. 72–83 (1997)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70 (2002)
17. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD. pp. 201–209 (2013)
18. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: DSN. pp. 245–256. IEEE (2011)
19. Konnov, I., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In: CONCUR. LNCS, vol. 8704, pp. 125–140 (2014)
20. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In: CAV (2015), (accepted, available at: `http://forsyte.at/download/kvw-cav15.pdf`)
21. Kopetz, H., Grünsteidl, G.: TTP – a protocol for fault-tolerant real-time systems. IEEE Computer 27(1), 14–23 (1994)
22. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. In: ACM SIGOPS OSR. vol. 41, pp. 45–58. ACM (2007)
23. Lamport, L.: The part-time parliament. ACM TOCS 16(2), 133–169 (1998)
24. Lamport, L.: Specifying systems: The TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
25. Lesani, M., Luchangco, V., Moir, M.: A framework for formally verifying software transactional memory algorithms. In: CONCUR. LNCS, vol. 7454, pp. 516–530 (2012)
26. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
27. Medeiros, A.: Zookeeper's atomic broadcast protocol: Theory and practice. Tech. rep. (2012)
28. Merz, S., Vanzetto, H.: Automatic verification of TLA + proof obligations with SMT solvers. In: LPAR. pp. 289–303 (2012)
29. Merz, S., Vanzetto, H.: Harnessing SMT solvers for TLA+ proofs. ECEASST 53 (2012)
30. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in egalitarian parliaments. In: ACM SOSP. pp. 358–372. ACM (2013)
31. Netflix: 5 lessons we have learned using AWS. (2010), `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`
32. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. CACM 58(4), 66–73 (2015)
33. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: USENIX. pp. 305–319 (2014)
34. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: CADE-11. pp. 748–752 (1992)
35. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0,1,\infty)$- counter abstraction. In: CAV, LNCS, vol. 2404, pp. 93–111 (2002)
36. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Correct Hardware Design and Verification Methods, pp. 54–66. Springer (1999)