Interactive Verification of Distributed Protocols Using Decidable Logic

Sharon Shoham(✉)

Tel Aviv University, Tel Aviv, Israel
sharon.shoham@gmail.com

1 Extended Abstract

Distributed systems are becoming more and more pervasive in our lives, making their correctness crucial. Unfortunately, distributed systems are notoriously hard to get right and verify. Due to the infinite state space (e.g., unbounded number of nodes and messages) and the complexity of the protocols used, verification of such systems is both undecidable and hard in practice.

Numerous works have considered the problem of *automatically* verifying distributed and parameterized systems, e.g., [1,9,10,17,18,20,23,24,26,38]. Full automation is extremely appealing. Unfortunately, automatic techniques are bound to fail in some cases due to the undecidability of the problem. Some impose restrictions on the verified systems (e.g., [26]), some may diverge (e.g., [24]) and some may report false alarms (e.g., [2]). Moreover, such techniques often suffer from scalability issues and from an unpredictable performance. As a result, most efforts towards verifying real-world systems use relatively little automation [19,25,31].

In contrast, *deductive verification* approaches let a user annotate the verified system with inductive invariants and pre/post specifications, and reduce the verification problem to the problem of proving the validity of the corresponding *verification conditions*. Tools for doing so vary in their expressiveness and level of automation. Some (e.g., [6,12,13,22,33,34]) check the verification conditions by decision procedures, but are limited in their expressivity. Others (e.g., [29]) use undecidable logics and semi-decision procedures, e.g., as provided by satisfiability modulo theories (SMT) solvers (e.g., Z3 [11], CVC4 [4], OpenSMT2 [21], Yices [14]), or by first-order solvers (e.g., Vampire [40], iProver [27]). Tools based on semi-decision procedures might fail to discharge the verification conditions either by non-terminating or by yielding inconclusive answers. Similarly to automatic verification approaches, they also suffer from an unpredictable performance: They might work well on some programs, but diverge when a small change is performed. This is sometimes referred to as a *butterfly effect* [30]. When this happens, it is often extremely difficult to discover, let alone remedy, the root cause of failure in the complex chain of reasoning produced by the algorithm.

Proof assistants such as Coq [5] and Isabelle/HOL [35] offer great expressivity, but require the user to write the proof (possibly exploiting various tactics), while mechanically validating every step in the proof. Verification using

A. Podelski (Ed.): SAS 2018, LNCS 11002, pp. 77–85, 2018.
https://doi.org/10.1007/978-3-319-99725-4_7

proof assistants is extremely labor intensive and requires tremendous efforts even by expert users (e.g., approx. 10 lines of proof were required per line of code in [42,43]). Thus, it is hard to deploy this method to verify complicated systems.

In summary, all of these approaches either (i) handle limited classes of systems, (ii) employ sound but incomplete automatic reasoning which reports too many false alarms, (iii) use semi-algorithms that tend to be fragile, unpredictable and often diverge, or (iv) require too much manual effort, relying on expertise in logic and verification.

Approach. We propose to overcome the shortcomings of existing approaches by using an interactive verification methodology that divides the verification problem into tasks that are well suited for automation and can be solved by decision procedures, and tasks that are best done by a human, and finds a suitable mode of interaction between the human and the machine.

This methodology is based on the conjecture that users usually have high level knowledge of the functionality of the code and interactions between different parts of the program. On the other hand, algorithmic techniques can be effective in reasoning about corner cases missed by the user. The key to success is to exploit these fortes when defining the roles of the user and the automated analysis, and to provide the suitable interface between them.

> *"One thing all programmers have in common is that they enjoy working with machines; so let's keep them in the loop. Some tasks are best done by machine, while others are best done by human insight; and a properly designed system will find the right balance."* — D. Knuth

We argue that letting a user convey her intuition to an automated analysis, and making sure that automation is restricted to decidable problems, will make the verification process more efficient and predictable, and will allow to balance between automation and expressivity.

An attempt at applying this methodology is implemented in Ivy [37]. In this work, we developed an interactive procedure for verifying safety properties of distributed protocols, where the verification conditions are expressed using decidable logic, allowing to check their validity completely automatically with decision procedures, and where the user's creativity guides the construction of the proper annotations. This is achieved by graphically displaying states that violate the verification conditions and letting the user select the relevant parts of the state according to which the annotations (inductive invariants) are updated. We elaborate on this approach in the sequel. We start with some background.

Decidable reasoning in Ivy. Ivy is a verification system based on decidable logic. Decidability greatly improves the predictability of proof automation, resulting in a more practical verification approach. Furthermore, it facilitates an interactive process, where the user may modify the invariants used for verification based on counterexamples.

Ivy supports several decidable fragments of first-order logic, of which [37] uses the **E**ffectively **PR**opositional fragment (EPR). EPR [39] is a fragment of first-order logic where the vocabulary is restricted to constant and relation symbols,[1] and the quantifier prefix is restricted to $\exists^*\forall^*$ in prenex normal form.[2] Satisfiability of EPR formulas is decidable [32] and supported by existing SMT solvers such as Z3 [11] and first-order logic provers such as iProver [27]. Moreover, EPR has the *finite-model property*, which means that every satisfiable formula has a finite model.

EPR is a relatively weak logic, but, perhaps surprisingly, it turns out to be suitable for modeling and verifying interesting systems, including software defined networks [3], heap manipulating programs [16,22], and, as we do in this work, distributed protocols [36,37,41]. We refer the interested reader to the aforementioned works for more details on modeling systems and their properties using EPR.

Safety verification. Safety properties specify bad behaviors that should never be encountered in any run of a system. An example of a bad behavior is the election of more than one leader in a leader election protocol. Safety properties are essential requirements that, when violated, might incur catastrophical outcomes.

One of the most useful techniques for proving safety of infinite-state systems already advocated by Floyd [15] is based on *inductive invariants*. Inductive invariants are an adaptation of the mathematical concept of "induction hypothesis" to the domain of programs. Technically, an inductive invariant I is a property of the system that (i) holds initially (initiation), (ii) implies the safety property (safety), and (iii) is preserved by every step of the system, namely if the system makes a step from any configuration that satisfies I, it reaches a configuration that satisfies I as well (consecution). If an inductive invariant exists, the system is safe. Thus, safety verification reduces to inferring inductive invariants. Similarly to mathematical proofs by induction, the most challenging and creative task in deductive verification of safety properties is coming up with the inductive invariants.

Example 1. As a concrete example, consider a simple distributed protocol for leader election in a ring [8]. The protocol assumes a ring of unbounded size. Every node has a unique ID with a total order on the IDs. Thus, electing a leader can be done by a decentralized extrema-finding protocol. The protocol works by sending messages in the ring in one direction: Each node announces its ID to its immediate neighbor. A node only forwards messages with higher ID than its own ID. When a node receives a message with its own ID, it declares itself as a leader. The safety property of interest here is that no more than one leader is elected. To verify the protocol means to verify that this property holds in every instance of nodes that run the protocol.

[1] It is straightforward to extend EPR to allow *stratified* function symbols, i.e., function symbols that do not create cycles among sorts (e.g., if there is a function symbol from sort A to sort B, then no function symbol from sort B to sort A is allowed).

[2] In particular, EPR does not allow the use of arithmetic operations.

In this example, the safety property itself is not inductive. For example, in a configuration where one leader is already elected but there is a pending message to some other node with its own ID, the property will be violated in the next step, hence violating the consecution requirement. Such a configuration is not reachable from the initial configuration of the protocol (where no leader is elected and no message is pending), but the safety property itself is not strong enough to exclude it. In order to exclude the counterexample to induction and make the candidate invariant inductive, it needs to be strengthened by adding (1) a conjecture saying that a message can reach a node with the same ID only if this ID is maximal — this conjecture will exclude the scenario described above, (2) a conjecture saying that the leader has the highest ID, and (3) a conjecture saying that messages cannot bypass nodes with higher IDs.

Verification conditions. We express protocols using a transition relation formula, denoted $Tr(\mathcal{V}, \mathcal{V}')$, where $\mathcal{V}$ is the vocabulary $\mathcal{V}$ used to describe the protocol's state, and $\mathcal{V}'$ is its copy used to represent the post-state of a transition. Initial state conditions, safety properties and inductive invariants are also specified via formulas, $Init(\mathcal{V})$, $P(\mathcal{V})$ and $Inv(\mathcal{V})$, respectively, over $\mathcal{V}$. Checking whether Inv satisfies initiation, consecution and safety, then corresponds to checking the validity of the following verification conditions:

initiation	$Init(\mathcal{V}) \rightarrow Inv(\mathcal{V})$
safety	$Inv(\mathcal{V}) \rightarrow P(\mathcal{V})$
consecution	$Inv(\mathcal{V}) \wedge Tr(\mathcal{V}, \mathcal{V}') \rightarrow Inv(\mathcal{V}')$

which in turn corresponds to checking the unsatisfiability of the following formulas that encode violations of the requirements:

violation of initiation	$Init(\mathcal{V}) \wedge \neg Inv(\mathcal{V})$
violation of safety	$Inv(\mathcal{V}) \wedge \neg P(\mathcal{V})$
violation of consecution	$Inv(\mathcal{V}) \wedge Tr(\mathcal{V}, \mathcal{V}') \wedge \neg Inv(\mathcal{V}')$

If Tr, $Init$ and $\neg P$ are EPR formulas and Inv is universally quantified, then these formulas fall into the decidable EPR fragment. Indeed, this is the case in the leader election example. If one of the formulas is satisfiable (i.e., the corresponding requirement does not hold), then a *finite* satisfying model exists (due to EPR's finite model property). For example, if consecution is violated, then a finite *counterexample to induction* is found – a state that satisfies Inv but has an outgoing transition to a state that violates it.

Inference of universally quantified inductive invariants via interactive generalization. In [37], we propose an interactive technique for inferring inductive invariants in the form of universally quantified formulas that is able to discover the inductive invariant of Example 1. The approach, implemented in Ivy, is based on iterative strengthening.

Iterative strengthening starts from a candidate inductive invariant, e.g., the safety property, and strengthens it iteratively until it becomes inductive.

Strengthening in Ivy is based on counterexamples to induction: a counterexample to induction s is excluded by conjoining the candidate invariant with a new conjecture that generalizes s into a set of excluded states. Generalization is crucial for the success of the approach. First, the conjecture obtained by generalization must not exclude any reachable state (otherwise, it would not be an invariant). In addition, it needs to be provable by induction with the given language of inductive invariants (otherwise, no further strengthening would turn the invariant into one that is also inductive). Finding a good generalization is extremely difficult to automate, and is a key reason for failure of many automatic techniques.

Therefore, Ivy uses an interactive generalization process, where the user controls the generalization, but is assisted by the tool. Ivy interacts with the user based on a graphical representation of concrete counterexamples to induction, taking advantage of the finite-model property of EPR formulas, as well as of the model-theoretic notion of a *diagram*.

The *diagram* [7] of a finite state (first-order structure) s, denoted $Diag(s)$, is an existentially quantified cube (conjunction of literals) that is satisfied by a state s' if and only if s' contains s as a substructure (where a substructure of s' is a structure obtained by restricting the domain of s' to some subset and projecting all interpretations to the remaining elements in the domain). As such, the negation of the diagram of s is a universally quantified clause that "excludes" s as well as any structure that contains it as a substructure, providing a natural generalization scheme. Additional generalization can be obtained by omitting from $Diag(s)$ some of the literals describing s (equivalently, omitting some "features" from s). These observations were used in [24] as part of an automatic invariant inference algorithm.

Ivy uses the diagram as a means to alternate between counterexamples to induction (which are natural for the user to look at) and universally quantified clauses that exclude them. Namely, when the consecution check fails, the user is presented with a *minimal* finite counterexample to induction, displayed graphically. The user responds by determining whether the counterexample to induction is reachable. If it is, then the inductive invariant is too strong and needs to weakened. If it is not reachable, the invariant can be strengthened to exclude it. In the latter case, the user hides some of the features of the counterexample to induction (e.g., the interpretation of some relation symbol) that she judges to be irrelevant to unreachability (i.e., such that the state remains unreachable with any valuation of these features). In this way, she uses her intuition to focus on the part of the state that really needs to be excluded. The feature selection is performed via a graphical interface. Ivy then computes the diagram of the generalized state, and transforms it into a universally quantified clause (conjecture) that excludes the generalized state and all the states that extend it. It offers the user several additional checks, such as bounded model checking to help verify that the new conjecture does not exclude any reachable state, and additional generalization via interpolation based on the bounded model checking check. All of these checks are implemented using decision procedures (relying on EPR's

decidability). In this way, the user controls the generalization process, and is assisted by predictable automation.

Ivy was successfully used to infer invariants for several distributed protocols which are beyond reach of automatic verification algorithms, demonstrating the effectiveness of EPR and the interaction based on counterexamples to induction. Moreover, under the assumption that the user identifies the "correct" features, we are able to bound the complexity of the approach by means of the size of a target invariant.

We note that while the interactive generalization technique is restricted to generating universally quantified inductive invariants, Ivy's graphical interface is useful also in cases where the inductive invariant is more complex. In such cases, Ivy provides counterexamples to induction, and updating the inductive invariant to eliminate them is done entirely by the user. This approach has also proven itself most effective, e.g., in verifying the Paxos consensus protocol [28] and several of its variants [36].

Conclusion. We propose a verification methodology that aims to balance between automation, expressivity and predictability by properly dividing the verification task between the human and the machine. Ivy realizes this methodology by letting the tool check inductiveness of a given candidate inductive invariant using decidable logic, and letting the user update the inductive invariant based on graphically displayed counterexamples to induction. For universally quantified inductive invariants, the latter is also done interactively via a process of interactive generalization. It is left to future work to investigate these ideas with respect to other logics, other inference algorithms (more sophisticated than iterative strengthening), and other interaction modes.

Acknowledgement. This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, and the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259.

References

1. Abdulla, P.A., Haziza, F., Holík, L.: Parameterized verification through view abstraction. STTT **18**(5), 495–516 (2016)
2. Alpernas, K., Manevich, R., Panda, A., Sagiv, M., Shenker, S., Shoham, S., Velner, Y.: Abstract interpretation of stateful networks. In: Static Analysis Symposium (SAS) (2018)
3. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: towards verifying controller programs in software-defined networks. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, UK, 9–11 June 2014, pp. 282–293 (2014)

4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004)
6. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 167–182. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_14
7. Chang, C., Keisler, H.: Model Theory. Studies in Logic and the Foundations of Mathematics. Elsevier Science, New York (1990)
8. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (1979)
9. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_55
10. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 61–68 (2013)
11. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
12. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 161–181. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_10
13. Dragoi, C., Henzinger, T.A., Zufferey, D.: Psync: a partially synchronous language for fault-tolerant distributed algorithms. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 400–415 (2016)
14. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV 2014), vol. 8559. LNCS, pp. 737–744. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-319-08867-9_49
15. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of Symposium on Applied Mathematics, vol. 32 (1967)
16. Frumkin, A., Feldman, Y.M.Y., Lhoták, O., Padon, O., Sagiv, M., Shoham, S.: Property directed reachability for proving absence of concurrent modification errors. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 209–227. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_12
17. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
18. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based verification of parameterized systems. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering (2016, to appear)
19. Hawblitzel, C., et al.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP, pp. 1–17 (2015)

20. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. In: Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014, pp. 39–52 (2014)
21. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: an SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_35
22. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 756–772. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_53
23. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_55
24. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. J. ACM, **64**(1):7:1–7:33 (2017)
25. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system Kernel. Commun. ACM **53**(6), 107–115 (2010)
26. Konnov, I.V., Lazic, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 719–734 (2017)
27. Korovin, K.: iProver - an instantiation-based theorem prover for first-order logic (system description). In: Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, 12–15 August 2008, Proceedings, pp. 292–298 (2008)
28. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
29. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
30. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 361–381. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_20
31. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
32. Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Comput. Syst. Sci. **21**(3), 317–353 (1980)
33. Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using STRAND. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 43–59. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_8
34. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, 20–22 June 2001, pp. 221–231 (2001)

35. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer, Heidelberg (2002)
36. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Program. Lang. PACMPL, **1**(OOPSLA), 108:1–108:31 (2017)
37. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, 13–17 June 2016, pp. 614–630 (2016)
38. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, 2–6 April 2001, Proceedings, pp. 82–97 (2001)
39. Ramsey, F.: On a problem in formal logic. Proc. London Math. Soc. **30**, 264–286 (1930)
40. Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, 13–19 July 2013, Proceedings. LNCS, vol. 8044. Springer, Heidelberg (2013)
41. Taube, M., et al.: Modularity for decidability of deductive verification with applications to distributed systems. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, 18–22 June 2018, pp. 662–677 (2018)
42. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 357–368 (2015)
43. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, 20–22 January 2016, pp. 154–165 (2016)