Practical Formal Methods for Distributed Systems

George Pîrlea

October 13, 2021

National University of Singapore

Contents

Contents ii

1 Introduction 1
- 1.1 Distributed Systems Are Buggy . . . 2
  - 1.1.1 Protocols Are Incorrect . . . 2
  - 1.1.2 Implementations Are Buggy . . . 7
- 1.2 Formal Methods Can Help . . . 11
- 1.3 Current Challenges . . . 12
- 1.4 Practical Methods . . . 13
- 1.5 Contributions . . . 14

2 State of the Art 15
- 2.1 Testing and Model Checking . . . 15
  - 2.1.1 Fault-Injection and Random Testing . . . 16
  - 2.1.2 Checking Temporal Logic Specifications . . . 19
  - 2.1.3 Connecting Models to Implementations . . . 21
- 2.2 Interactive Verification . . . 22
  - 2.2.1 Refinement Using Verified System Transformers . . . 22
  - 2.2.2 Compositional Reasoning With Separation Logic . . . 23
  - 2.2.3 Linking Refinement and Separation Logic . . . 23
- 2.3 Semi-Automated Verification . . . 24
  - 2.3.1 Deductive Verification of TLA Specifications . . . 24
  - 2.3.2 Safety and Liveness Proofs for Implementations . . . 24
  - 2.3.3 Finding Invariants by Interactive Generalisation . . . 25
- 2.4 Automated Verification . . . 27
  - 2.4.1 Property-Directed Reachability . . . 27
  - 2.4.2 Incremental Inference of Universal Invariants . . . 29
  - 2.4.3 Data-Driven Invariant Learning . . . 30
  - 2.4.4 Invariants with Quantifier Alternations . . . 32
- 2.5 Gaps in the State of the Art . . . 34

3 Preliminary Work 37

4 Towards Automated Reasoning 39
- 4.1 Guided Invariant Discovery . . . 39
- 4.2 Challenges of EPR Encoding . . . 45

5 Conclusion 49

Bibliography 50

1 Introduction

Computer programming is art, science, and engineering rolled into one [Knu74]. Nowhere is this more apparent than in *systems programming*, where practitioners juggle concerns of architectural elegance, empirical validation, and low-level implementation all at once.

[Knu74]: Knuth (1974), 'Computer programming as an art'

In this work, we are interested in *distributed systems*, which have to cope with the combination of (partial) failure, concurrency, and fickle latency. We seek to answer the question:

> How to design, implement, monitor, and maintain distributed systems such that we can have formal assurance about their correctness?

First, we are concerned with real systems in all their complexity. Toy examples and models will serve only as stepping stones towards building confidence in practical systems.

Second, we want the techniques we use to require as little human effort and ingenuity as possible.[1] Automated techniques are preferable to manual ones.

1: In industry, formal methods have a reputation of requiring extraordinary efforts to verify even simple code [New+15]. To be considered *practical*, formal techniques need to scale to realistic systems with a reasonable amount of effort.

Third, we are interested in the entire system lifecycle: its design, implementation, service, and ongoing maintenance and evolution. Many techniques apply only to abstract models, not concrete code, or offer guarantees about the code without reference to the environment it is executing in, or expect that the code is generated from the specification rather than written separately, or are brittle in the face of changes. We want assurance that touches the full development cycle.[2]

2: Integrating with existing software development workflows is key for industry adoption of formal methods [Rei+20].

Finally, we want *formal* assurance. We should be able to precisely describe our level of confidence and justify it on a mathematical basis.

This last desideratum requires some motivation. Surely guarantees are desirable, but why do we need *formal* assurance?

Table 1.1: Errors found in distributed protocols.

Protocol	Reference	Violation	Counter-example
Chord	[Sto+01; LBK02]	liveness[a]	[Zav12; Zav17]
Pastry	[RD01]	safety	[AMW16; AMW18]
Generalised Paxos	[Lam05]	non-triviality[b]	[SS10]
FaB Paxos	[MA05; MA06]	liveness	[Abr+17]
Multi-Paxos[c]	[CGR07]	safety	[Mic+17]
Zyzzyva	[Kot+07; Kot+09]	safety	[Abr+17]
CRAQ	[TF09]	safety[d]	[Whi20]
JPaxos	[Koń+11]	safety	[Mic+17]
VR Revisited	[LC12]	safety	[Mic+17]
EPaxos	[MAK13]	safety	[Sut20]
EPaxos	[MAK13]	safety	[Whi21]
Raft	[OO14]	liveness	[Hoc14]
Raft	[Ong14]	safety	[AZ15; Ong15]
Raft	[OO14; Ong14]	liveness	[HA20; JHM21]
hBFT	[DPL15]	safety	[SKD19]
Tendermint	[Buc16]	liveness	[CV17]
CAESAR	[Aru+17]	liveness	[Ene+21]
DPaxos	[NAE18]	safety	[Whi+21]
Sync HotStuff	[Abr+19]	safety & liveness	[MC19]
Gasper	[But+20]	safety & liveness	[NTT21]

[a] Eventual reachability is Chord's key correctness property.
[b] Acceptors might accept commands that have not been proposed.
[c] As described in Paxos Made Live.
[d] Client reads might fail due to incorrect garbage collection.

1.1 Distributed Systems Are Buggy

To show the benefit of formal methods, it helps to review a few episodes that illustrate the pitfalls of informal and semi-formal reasoning about distributed protocols and systems.

1.1.1 Protocols Are Incorrect

It is widely accepted that distributed protocols[3] are difficult to reason about. But few appreciate *how often* even domain experts make mistakes. To give the reader some intuition, Table 1.1 presents a (non-exhaustive) list of errors found in the descriptions of notable protocols.

3: Throughout this report, we distinguish *protocols* and *systems*. A protocol is an abstract description of an algorithm, whereas a system is its practical instantiation, consisting of code deployed in a real execution environment.

These protocols were published by experienced researchers, and some came with detailed correctness arguments, but all had errors. We describe the three most instructive ones:

Chord. Chord is a peer-to-peer lookup protocol that implements a distributed hash table [Sto+01]. The protocol maps keys to values by maintaining—without a central authority—a mapping between each key and the node responsible for storing the key's value.

[Sto+01]: Stoica et al. (2001), 'Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications'

Concretely, each node is identified by a hash of length 2^m bits, and nodes organise themselves in a ring topology: each node is succeeded by the node with the next higher hash value (mod 2^m).[4] Keys are assigned to nodes by hashing. The key's value is held by the node with the smallest identifier that is equal to or greater than the key's hash.

4: In practice, nodes keep track of a successor *list* rather than a single successor. The key operating assumption in Chord is that every node has at least one live node in its successor list.

As nodes leave and join the network, they maintain the ring topology and move values around as needed to keep the mapping consistent.[5] Chord's key correctness property is *eventual reachability*—given ample time, as long as no further nodes join or leave, the protocol will eventually repair all disruptions in the ring structure [Zav12]. The protocol has to uphold this property even in the presence of failures, concurrent leaves and joins, and message delays.

5: A leaving node passes responsibility for its keys to its successor, whereas a joining node receives responsibility from its successor. No other changes are necessary.

In a separate paper analysing Chord's properties, the authors identify a set of invariants that must hold for Chord to be correct in the sense of eventual reachability [LBK02].[6]

6: Formally, *eventual reachability* says that in any execution state, if there are no subsequent joins or failures, then eventually the network will become ideal (an ordered ring) and remain ideal.

However, in subsequent work using the Alloy Analyzer modelling tool, Pamela Zave showed that Chord does not satisfy *any* of its claimed invariants [Zav12]. The root issue is that during reconfigurations (due to joins or detected failures), Chord nodes can update their successor lists in ways that remove all live nodes—despite those nodes neither failing nor leaving the network.

In follow-up work, Zave published a corrected version of Chord, model-checked for all instances of up to $n = 9$ nodes and successor lists of length up to $r = 3$ [Zav17].[7] Importantly,

7: r is a parameter. Longer lists let the protocol tolerate more concurrent reconfigurations.

this version has a simple inductive invariant, consisting of only two clauses: (1) every successor list has at least one live entry and (2) there are at least $r+1$ nodes that are not skipped by any node's successor list.[8]

The Chord story shows how semi-formal reasoning about even relatively simple protocols can be very wrong.[9] Moreover, it shows the benefits of *tool-supported formal reasoning*. Zave credits the Alloy Analyzer—which allows model exploration and shows concrete counter-examples—for allowing her both to understand why the claimed invariants did not hold *and* to discover the much simpler invariant behind the corrected version of Chord [Zav17].[10]

Due to formal methods, not only do we have more assurance about Chord's correctness, but we also have a better understanding of the protocol—the simple invariant shows exactly what keeps Chord safe and, in fact, reveals that parts of the original protocol were unnecessary.[11]

Pastry. Pastry is a peer-to-peer overlay network[12] with a built-in distributed hash table [RD01]. Pastry is similar to Chord: it builds a ring based on node identifiers and places each key at the node with the numerically closest identifier.

To route key requests to the appropriate node, Pastry also builds a *routing table* using an application-specific proximity metric.[13] The routing table of a node stores, for every prefix in identifier space, a list of the *closest* nodes with identifiers that match the prefix, where closeness is computed using the proximity metric.[14] Nodes also store a list of neighbours in identifier space and a separate list of neighbours according to the proximity metric.

Using this information, Pastry nodes can perform hierarchical routing, thus minimising the distance messages travel.

A desirable safety property for Pastry is *key consistency*—at any time, there is only one node responsible for any key.[15] To this end, nodes must maintain a consistent view of the ring structure as participants join and leave the network.

8: The second clause guarantees the ring retains enough structure to fix r disruptions.

9: Invariants capture an understanding of the protocol. Chord's invariants did not hold because the protocol behaved differently than its creators thought.

10: This story convinced engineers at Amazon Web Services (AWS) that formal methods can be applied productively to real distributed systems [New+15].

11: After failing, a node had to wait a long time, until all references to it in successor lists were purged, before rejoining. This was shown to be unnecessary.

12: An *overlay network* is built on top of an existing network.

13: Typically, proximity is defined in terms of number of network hops between nodes.

14: For each prefix of length n, the list contains nodes with every possible $n+1$th digit (if such a node is known).

15: To our knowledge, key consistency, a.k.a. *correct delivery*, has not been studied for Chord, but eventual reachability implies eventual key consistency.

However, Pastry's authors empirically observed that their protocol does not guarantee key consistency under high rates of churn [CCR04] or in the presence of asymmetric connectivity [Hae+05] and devised enhancements to correct these deficiencies [CCR04; Hae+05].[16] Concretely, Haeberlen introduced a sub-protocol that manages ownership of keys via the concept of leases—when a node joins, it must explicitly lease a portion of the key space before it can become active and answer lookup requests; similarly, a failed node's lease expires and is negotiated among its neighbours [Hae+05].

16: [Hae+05] notes that [CCR04] does not maintain safety in the presence of path failures, despite being hardened against churn.

Despite these changes being empirically validated,[17] they are incorrect in a rather unfortunate way.

17: "[T]he new version has been successfully run for multiple days without any detected routing inconsistencies." [Hae+05]

The hardened protocol [CCR04] could, when multiple nodes attempt to join concurrently, end up with nodes having inconsistent views of their neighbourhood. The new protocol [Hae+05] does fix the specific issue that lead to inconsistent views, but *still fails* to correctly handle concurrent joins—as Lu showed by modelling both versions in TLA$^+$ [Lu13].[18]

18: The error manifests without any failures—concurrent joins suffice. It is fixed by having each node process only one join at a time.

The lesson here is that, even if designers know that their protocol is incorrect, that does not necessarily mean they will be able to develop a correct fix. Moreover, we see that empirical validation is not sufficient to guarantee correctness.

Once again, *tool-supported formal reasoning* can give us assurance that the changes we make to our protocols are indeed correct (or show us that they are not).[19]

19: This was an incorrect change to an incorrect protocol. Often, however, we see incorrect changes to correct protocols—an example is Zyzzyva, which tried to optimise PBFT by adding speculation, but is unsafe [Abr+17]. Validating optimisations is an important use-case for formal methods in industry [New+15].

Raft. Raft is a distributed consensus protocol, described as more practical and easier to understand than Paxos [OO14]. Its key feature is a strong notion of leadership—log entries only flow from leaders to backups and thus no log transfer is needed during leader election. Moreover, Raft's leader election is very simple and surprisingly efficient [HM20].

Raft's original description included a membership change scheme called *joint consensus*, in which during the transition from the old to the new configuration, agreement requires majorities from *both* the old and the new cluster. This lets Raft

continue serving client requests during the transition, but is complex and subtle [Ong14]. Indeed, the original version of joint consensus was shown to have liveness issues [Hoc14].

Acknowledging the complexity of joint consensus, Ongaro developed a simpler membership change algorithm that only adds or removes one server at a time [Ong14].[20] However, using TLA$^+$, Amos and Zhang showed that this scheme is unsafe if a leader fails during a configuration change and comes back online as a newly-elected leader attempts a different change [AZ15; Ong15].[21] To solve this issue, a new leader should not propose a configuration change until it commits an entry in its term—this ensures that it knows all previously committed configuration changes *and* that no uncommitted conflicting change can commit.[22]

20: To preseve safety, majorities in the old and new clusters should intersect in at least one server.

21: When the old leader restarts, three configurations (one old and two new) are competing. There is no guarantee that they all intersect, so safety is compromised.

22: A new configuration requires a majority of the old configuration to agree, but this is impossible since a majority has already agreed to the new leader's entry.

Once again, we see that simpler protocols are not necessarily easier to reason about—the simple membership change scheme was unsafe. In a similar vein, Raft's simple leader election scheme also had errors. In unusual network conditions, when the cluster is only partially connected, Raft leader election was shown not be live even when a majority of the cluster *can* communicate [HA20]. This bug was observed in production at Cloudflare and led to an outage that lasted 6 hours and 33 minutes [LS20].[23] To avoid this issue, candidates in Raft must run a trial election to test that they can win before forcing the current leader to step down *and* leaders must actively step down if they do not receive heartbeats from a majority of the cluster [HA20; JHM21].

23: The Cloudflare blog post calls the issue a Byzantine failure, but this is incorrect. It was an omission failure (message loss), which Raft should tolerate.

Lessons. We presented errors found in three protocols: Chord, Pastry, and Raft. These show that reasoning about distributed protocols is difficult and that even experts get it wrong. Table 1.1 references many more mistakes. While we do not have enough space to explain them, we hope that the sheer number of errors discovered is enough to convince the reader that *mistakes in distributed protocols are very common*. We also saw that formal reasoning can help, and in Chapter 2 we will see precisely what current tools can and cannot do.

1.1.2 Implementations Are Buggy

It is not only protocols that are incorrect. Implementations have bugs too, and much more often than protocols.

Bugs are common and have impact. In a comprehensive study of six popular cloud systems, Gunawi *et al.* review 21,399 bug reports and perform a deep analysis of 3655 "vital" issues that affected real system deployments [Gun+14]. They find that 45% of bugs impact the reliability[24] of affected systems, 22% impact system performance, and 16% compromise availability.[25] Moreover, about 5% of bugs, while not causing data loss or corruption, produce data inconsistencies.

In the following, we review the major classes of distributed system bugs identified in the literature.

Error handling. Yuan *et al.* manually reviewed 198 randomly selected, user-reported failures[26] of popular distributed systems: Cassandra, HBase, HDFS, Hadoop MapReduce, and Redis [Yua+14]. They find that failures arise especially when services are starting, particularly in long-running clusters, and that 24% of failure scenarios involve nodes being unreachable.

Yuan *et al.* distinguish between catastrophic failures, where most or all users of the system experience an outage or data loss, and non-catastrophic failures, where only a minority of users are affected. Of the 198 sampled failures, 48 are catastrophic and 150 are non-catastrophic.

All but one of the catastrophic failures in the sample were caused by incorrect error-handling.[27] Of these, 43% were caused by trivial mistakes in error-handling code, *e.g.*, ignoring errors or aborting execution due to non-critical exceptions such as I/O errors, and 23% were caused by exception-handling blocks that *always* crash.[28] The remaining 34% were complex bugs where the developers did not anticipate certain error scenarios.

24: *E.g.*, cause data loss, data corruption, or otherwise make the system behave incorrectly.

25: Gunawi *et al.* note that designers prioritise availability over reliability, likely because users evaluate services based on availability and performance metrics rather than correctness, which is harder to quantify.

26: Reported bugs might not be a representative sample of all bugs. Issues that do not lead to outages or data loss are less likely to be detected in the first place.

27: Only 25% of the normal failures were caused by incorrect error-handling. We should not be hasty to generalise.

28: This code was never exercised during testing. A test suite with full line coverage would have detected all these errors.

Out of the 198 failures, 77% are either deterministic or have simple timing dependencies and therefore can be reproduced by unit testing.[29] Non-deterministic bugs arise mostly due to incorrect use of shared-memory concurrency.

29: This suggests that regresion testing can be very effective for distributed systems.

The event sequences that led to these failures are relatively complex, requiring multiple steps from a large space of possibilities.[30] Nonetheless, while the space of possible steps is large, 98% of the surveyed failures manifest with 3 or fewer nodes and 90% require at most 3 steps [Yua+14].

30: These systems have large test suites and are widely used, so common errors had already been eliminated.

Node changes. In a study reviewing only node-change bugs,[31] Lu *et al.* identified 620 distinct node change bugs and manually inspected 120 chosen at random [Lu+19]. They analysed bugs in Cassandra, HBase, HDFS, Hadoop YARN (the successor to MapReduce), and Zookeeper.

31: Node changes are reconfigurations, node crashes, or restarts. These are notoriously difficult to handle and a common source of errors in both protocols and systems. Indeed, all protocol errors we reviewed in the previous section were node-change errors.

Restarts are the most common trigger for errors, accounting for 68.2% of the surveyed bugs, followed by crashes at 18.7% and graceful shutdowns at 11.6%, respectively. Lu *et al.* find that node additions are rarely a source of errors for the systems they look at—of the 620 bugs analysed, only 9 were caused by starting new nodes.

Similar to Yu *et al.*, they find that 85% of bugs are shallow, requiring no more than 2 events to trigger.

In terms of root causes, Lu *et al.* describe 33.3% of the issues as distributed concurrency bugs, resulting from unexpected interleavings either between threads at a single node or across nodes. Incorrect crash-recovery and restart code accounts for 36.7% of the bugs, while incorrect shutdown handlers are responsible for 13.3% of errors. The remaining 16.7% of bugs are system specific and difficult to clearly categorise.

Importantly, 45% of the analysed node-change bugs print explicit error messages and are thus easy to identify in system logs, whereas 55% are silent.

In prior work, Gao *et al.* specifically analysed bugs in the crash-recovery mechanisms of distributed systems [Gao+18].[32]

32: Guarding systems gainst arbitrary crashes is challenging. The problem has been studied formally for file system implementations [Che+15; Sig+16].

Almost all of the inspected bugs involve 4 or fewer nodes and 87% of crash recovery bugs require a combination of no more than three crashes and one reboot. However, such bugs are difficult to find since they heavily depend on timing—crashes must occur in specific system states to manifest as bugs.

Distributed concurrency. Concurrency is a major source of errors in implementations of distributed systems, as developers have to reason about event interleavings not just at a single node but across the entire cluster.

Leesatapornwongsa *et al.* review 104 distributed concurrency[33] bugs from four cloud systems, noting that such bugs are difficult to find, as they often arise due to unexpected interactions between *multiple* protocols, and in 47% of cases lead to silent failures rather than overt errors [Lee+16].

33: In distributed concurrency (DC), event interleavings arise due to distributed execution and message delays. Contrast with local concurrency (LC), where interleavings arise due to having multiple threads of execution.

They classify the conditions that trigger DC bugs into two categories: (1) *order violations*, where correct behaviour depends on events happening in a specific order which is not observed in practice, and (2) *atomicity violations*, where certain events must execute independently but in practice they interfere with each other.

Interestingly, while local concurrency errors are known to mostly arise due to atomicity violations (roughly 70% of LC bugs) [Lu+08], Leesatapornwongsa *et al.* observe that, in the systems they study, distributed concurrency errors are mostly caused by order violations [Lee+16].[34]

34: Intuitively, a distributed environment presents more opportunities for order violations.

Timeout problems. Timeouts are a common mechanism for detecting and handling failures in distributed systems.

Analysing 156 timeout bugs in 11 cloud systems, Dai *et al.* find that 40% of timeout-related bugs cause system unavailability, 33% cause specific jobs within the system to fail, 26% cause performance degradation, and 2% cause data loss [Dai+18].

In terms of root causes, almost half of timeout bugs arise because a timeout variable is misused, being set incorrectly,

not being updated during execution, being ignored, or being wrongly reused by more than one timeout checking function. A third of problems arise due to missing timeout checking, whereas the rest are caused by incorrect handling of timeouts, unnecessary timeouts, or clock drift.

Partial network partitions. Implementations of fault tolerance techniques are often inadequate, such that network failures, especially if partial,[35] cause serious problems.

Conducting an in-depth study of 51 partial network partitioning failures in 12 cloud systems, Alfatafta *et al.* al found that 74.5% are catastrophic, either crashing the system or making the system violate its guarantees, with data loss and unavailability being the most common occurrences [Alf+20].

Worryingly, 84.3% of the failures are silent, not producing any informative error message. Leader election is the most commonly affected mechanism, followed by reconfiguration and replication. However, 68.6% of the studied bugs required three or fewer events to trigger, and *all* of them manifest with only a single-node partial partition.[36]

Not all of these failures are caused by implementation errors, however. Looking at the 72.5% of errors that were fixed, Alfatafta *et al.* found that the majority, 56.8% required design changes in the protocol, *i.e.*, they were protocol bugs.[37] Nonetheless, 43.2% of the fixed issues required only code changes rather than fundamental redesigns [Alf+20].

File system faults. Disk faults are another source of errors. While most distributed systems replicate data at multiple nodes, implementations often fail to provide fault tolerance against disk read and write errors.

Inspecting eight popular distributed storage systems, Ganesan *et al.* found that file system faults at a single node can lead to cluster-wide data corruption, as errors are propagated to the entire cluster, or unavailability, as errors lead to node crashes [Gan+17].

35: A *complete* partition separates a cluster into disconnected parts, whereas a network experiencing a *partial* partition is still connected, but not in a clique topology (*i.e.*, all-to-all).

36: This might be a side-effect of multiple-node partial partitions being less common and thus less likely to lead to bug reports.

37: We saw a protocol-level error of this kind in the Raft leader election liveness issue, so this is not entirely surprising.

Lessons. Implementations of distributed systems are *very* buggy, with hundreds of critical errors found in mature, production-ready systems. Moreover, these bugs have varied causes and can be difficult to find, reason about, and fix.

One can see, therefore, that it would be useful to have some sort of *formal* assurance about implementations, *e.g.*, at the very least, certainty that certain classes of bugs are not present or that shallow bugs do not exist,[38] and perhaps even assurance that the implementation is entirely bug-free.

38: The fact that, empirically, most *discovered* implementation bugs are shallow makes this very attractive—although it is not clear whether this observation is more a reflection of our (in)ability to find bugs rather than an intrinsic property of distributed systems.

1.2 Formal Methods Can Help

We have already seen that formal reasoning tools can be used to discover and fix errors in protocols. They can also help with implementations, and to great effect.

Fonseca *et al.* studied the correctness of IronFleet [Haw+15], Verdi [Wil+15], and Chapar [LBC16], three formally verified distributed system implementations [Fon+17]. Through a combination of code review and testing, they found 16 bugs in these systems. The key finding, however, is that none of these were in code that was verified.[39] Specifically, 11 bugs were found in the so-called *shim layer* that handles the interaction between the verified implementation and the operating system, *e.g.*, network communication and disk operations. Verdi and Chapar have an unverified shim, written in OCaml, in which Fonseca *et al.* found bugs. IronFleet's shim, on the other hand, is verified, and no bugs were discovered in it.

[Fon+17]: Fonseca et al. (2017), 'An Empirical Study on the Correctness of Formally Verified Distributed Systems'

39: Fonseca *et al.* report spending *more than eight months* searching for bugs. In their words, "this result suggests that these verified distributed systems *correctly* implement the distributed system protocols, which is particularly impressive given the notorious complexity of distributed protocols" [Fon+17].

This shows that formal verification is in fact effective at removing bugs from system implementations.

However, IronFleet had a serious specification error—its Multi-Paxos implementation handled packet duplication incorrectly and thus failed to provide exactly-once semantics for client requests, and Dafny, the verification tool used in IronFleet had two bugs that could manifest as incorrect programs being reported as verified [Fon+17].[40] Similarly,

40: Studies in other application areas confirm this pattern. For instance, Yang *et al.* studied bugs in C compilers. For CompCert, a verified compiler, after about 6 CPU-years of testing, they found an error in its unverified front-end and an error caused by a specification bug [Yan+11].

Verdi [Wil+15], which includes a verified implementation of Raft, did not catch any of the Raft protocol bugs we discussed in Section 1.1.1, since it did not model cluster membership changes or attempt to verify liveness.[41]

41: Nonetheless, the safety bug in the one-at-a-time membership change protocol was discovered via model checking.

The lesson is that so-called "correctness" guarantees should be properly understood—verification certainly does remove bugs, but specification errors and omissions *are* a real threat, and similarly, verification tools with large trusted computing bases (TCBs) should be treated with caution.

1.3 Current Challenges

Despite their benefits, many formal techniques struggle to scale to industrial applications. For instance, the systems Fonseca *et al.* reviewed [Fon+17] required years of expert effort to formally verify and are, as things currently stand, unused and largely unmaintainable.[42] Manual proofs do not scale and even semi-automated reasoning is too expensive.

42: Verdi was developed over 18 months [Woo+16] and IronFleet required approximately 3.7 person-years [Haw+15].

Faced with this reality, industrial actors have focused on *lightweight* methods that do not involve proofs. The most popular such method is model checking, which is widely used in industry [New+15; Dem18; Lam18; Hal20; SZT21]. Model checking tools are easy for developers to learn[43] and can be used to quickly experiment with and validate potential system designs. Moreover, models can be and in practice often are connected to implementations via model-based testing [Bor+21] and conformance monitoring [Hal20].

43: It takes AWS engineers two to three weeks to become productive with TLA$^+$ [New+15].

Such approaches do not completely eliminate bugs, as verification methods do, but require much less manual effort and, significantly, do not demand a large upfront investment—they operate on a *pay-as-you-go* model [AT17; Bor+21], in which the initial cost of adoption is low and increased assurance is obtained by incremental effort.[44]

44: This is in contrast to interactive or semi-automated verification approaches, which typically do not give any assurance until the effort is completed. Such methods are much more difficult to adopt in industry [Rei+20].

However, even lightweight methods have issues. Besides not offering strong guarantees, the most pressing problem not

only for lightweight methods, but indeed for all types of techniques is *software evolution*.[45] As code changes, the specification must change as well, and in the case of interactive and semi-automated verification methods, proofs must be rewritten. This is difficult in an industrial setting, where the developer who changes the code might not have the expertise (or motivation) to update the specification and proofs.

45: Another problem is checking *liveness*, as opposed to safety. Techniques to reason about liveness are under-developed.

1.4 Practical Methods

In terms of validating implementations in a *practical* fashion, three approaches show promise.

In recent work to validate distributed systems at Amazon Web Services, engineers express both the implementation code and the specification in the same general purpose programming language [Bor+21]. In this case, the specification is a simple-to-understand *reference implementation*, which lives in the same source repository as the real system. As the implementation changes, its conformance to the specification is automatically checked via property-based testing on every code commit, as part of the system's regular test suite. While this cannot be used to validate deep properties, it is a cheap way to improve assurance in the system.

[Bor+21]: Bornholt et al. (2021), 'Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3'

A different method, called *continuous formal verification*, can be used to verify (not just validate) implementations [Chu+18]. The idea is to *automatically* connect high assurance proofs written in Coq, that rarely require changes, to system code that changes daily, via symbolic execution. This is used at AWS to verify deep properties (computational indistinguishability) of cryptographic implementations written in C.[46]

[Chu+18]: Chudnov et al. (2018), 'Continuous Formal Verification of Amazon s2n'

46: O'Hearn identifies continuous reasoning, which mirrors the continuous model of software development practiced in industry, as key to scaling formal methods in practice [OHe18].

This, in turn, can be enhanced by increasing the level of automation for the high-assurance proofs. If neither specification nor implementation changes require human effort to update proofs, the process of continuous formal verification

could be completely automated. Moreover, it might be possible to automatically derive changes in the implementation from changes in the specification, and vice-versa.

1.5 Contributions

This report is structured as follows.

- In **Chapter 2**, we survey the state of the art in formal methods for distributed systems and identify open problems, highlighting how formal methods could be used to improve assurance in all system lifecycle stages.

- In **Chapter 3**, we report on our previous work with formal methods and distributed systems, explaining how different methods compare, based on our experience.

- In **Chapter 4**, we present our initial experiments with automated reasoning and invariant inference, techniques we describe in Chapter 2. We use Ivy's graphical interface [Pad+16b] to discover an inductive invariant for the classic two-phase commit protocol, and use mypyvy [Fel+19] to encode a Raft-style version of Multi-Paxos, as described in a recent comparison of the two protocols [HM20]. This gave us a clearer understanding of what needs to be done to enable automatic translation of specifications into decidable logics.

- In **Chapter 5**, we conclude by reviewing the ground we covered and reflecting about future work.

State of the Art 2

In this chapter, we survey existing techniques for building assurance in distributed systems. We cover the full spectrum of methods: testing and model checking, interactive verification, and automated reasoning. We aim to present a detailed overview of the field and identify gaps in the state of the art, both for protocol and for system verification.

2.1 Testing and Model Checking

The most direct way to build confidence that a system does what it should is to test it, *i.e.*, give inputs to the system and check that the outputs match the expectation.

Testing can be very effective for systems in which the space of inputs is small, and exhaustive enumeration of all inputs is therefore possible. For such systems, testing can provide certainty of the system's correctness.[1] However, most systems, and certainly most *interesting* systems, have infinite input spaces,[2] so exhaustive testing is out of the question.

Distributed systems are typically reactive, *i.e.*, they maintain an ongoing interaction with their environment and never terminate, so they are generally not amenable to exhaustive testing. Moreover, for reactive systems, we are not only interested in safety properties, but also liveness properties, which can be more difficult to test.

For protocols, the equivalent of testing is *model checking*. Given a description of the protocol as a state transition system (*i.e.*, a model) and a logical formulation of the properties it must satisfy, a model checker systematically explores the protocol's behaviours[3] and checks that the specified properties hold.

The benefit of model checking over testing comes from models being expressed at higher levels of abstraction than implementations. Abstract models are easy to change and

1: The process of obtaining absolute certainty is called *verification*. Exhaustive testing is a form of verification [GG75].

2: Or, at least, input spaces that are too large to feasibly explore.

3: A behaviour is an execution trace of the protocol.

therefore they are a good tool for exploring and validating different possible system designs.[4] Indeed, companies in industry use modelling and model checking tools when designing new systems, before they start implementing them, to catch errors early [New+15].

4: A change that might take days to code in an implementation can be developed in a matter of hours for a model.

At the same time, however, in a commercial setting at least, it is not sufficient to validate only the *design* of a system without validating its *implementation*. Testing is also necessary.

Moreover, testing can be performed in the environment in which the system actually runs. A common error with models is that they do not precisely describe the environment—the "real world" can exhibit faults which the model does not capture, but which affect the correctness of the system. For this reason, companies employ so-called *chaos engineering*, deliberately introducing random faults into their systems to ensure they behave correctly even in unfavourable execution environments [Net11; Rob+12; Nak15].[5]

5: Notably, Netflix, which introduced the term, performs chaos testing in production [Net11].

2.1.1 Fault-Injection and Random Testing

Experience shows that random testing of distributed systems can be very effective. For instance, Jepsen, a *black-box* testing[6] framework for partition faults has found a remarkably large number of bugs in over 25 production systems [Kin20].[7]

6: Black-box tests interact with the system only via its regular input and output interfaces. Unlike white-box testing, no access to the source code is required.

7: Jepsen introduces "special" partition faults, *e.g.*, producing topologies in which every node can see a majority, but no node sees the *same* majority as others, or where the cluster is split in half, but there is a bridge node which has bidirectional connectivity to both halves. It also supports other types of faults, *e.g.*, pausing process execution or randomly shifting system clocks by small amounts [Kin13].

Fault injection. While Jepsen is arguably the most widely known tool for fault injection, other projects perform similar testing with custom tooling [Has15; She16; Rei16; Cor20].

FoundationDB, for instance, was designed from the ground up to allow deterministic simulation of the entire system. Its simulator can introduce node crashes, reboots, network faults, partitions, disk errors, and clock errors to check that the database provides strong consistency in unfavourable environments under arbitrary workloads [Zho+21]. Most interestingly, FDB's code cooperates with the simulator in

producing rare events via a process called *buggification*—code that can fail (*e.g.*, disk operations) is annotated with the `BUGGIFY` macro such that the failure condition at that point can be artificially triggered by the simulator [Mil21]. This *white-box* approach finds errors faster than black-box tools like Jepsen, but has the disadvantage of requiring more effort to set up and needing some human input (in the form of code annotations) to guide the search.

Lineage-driven fault injection is an alternative approach that is fully automated, and requires no human ingenuity [Alv+16]. It was developed precisely to solve the dual problems of black-box tools wasting resources, on one hand, and white-box approaches requiring human ingenuity, on the other. Rather than inject faults at random or using human-generated heuristics, LDFI learns[8] what mechanisms provide fault tolerance in a correct execution (*e.g.*, redundancy, broadcast), and injects only failures that affect the mechanisms involved in reaching a correct outcome [ARH15; Alv+16; AT17]. Failures that cannot lead to incorrect outcomes are never explored, reducing the search space.

[Alv+16]: Alvaro et al. (2016), 'Automating Failure Testing Research at Internet Scale'

8: In the original paper [ARH15], lineage information for each protocol was *given*, expressed in Datalog. When deployed in production at Netflix, however, lineage was first *learned* by observing real executions via tracing and then used to derive a model of redundancy [Alv+16; AT17].

When is testing effective? Jepsen is so successful because, while the space of possible executions is too large to explore to any significant degree, partition bugs can, with overwhelming probability, be discovered by exploring only a small subset of executions [MN17].

[MN17]: Majumdar et al. (2017), 'Why is random testing effective for partition tolerance bugs?'

Other kinds of bugs, however, do not have this property, so they are much harder to discover. Nonetheless, as we have seen in Section 1.1.2, most bugs we empirically encounter in distributed systems are shallow, requiring only a few events to occur for the bug to manifest—they have low *bug depth*. This means that, given enough time, random testing will discover these bugs. Moreover, as we have seen, human-provided annotations and lineage reasoning can be used to bias the search in productive directions.

Reduction. Various other techniques improve the performance of random testing by reducing the number of executions that need to be explored. The general idea is to exploit properties such as (1) communication and state symmetry and (2) event independence to show that classes of executions have equivalent behaviours—therefore it suffices to explore only a representative execution from each class. Techniques that exploit these properties are called *symmetry reduction* [MDC06] and *partial order reduction* [Cla+99; FGL05], respectively, and are widely used.

Software model checkers. While these techniques were initially developed for model checking, they have been adapted for use in testing implementations[9] via tools such as MaceMC [Kil+07], MODIST [Yan+09], dBug [SBG10], SAMC [Lee+14], and FlyMC [Luk+19]. These so-called *software model checkers* allow systematic testing of distributed system implementations and have been successful at finding complex bugs—FlyMC, for instance, found a Paxos bug in Cassandra requiring 54 events to occur [Luk+19].

9: Somewhat confusingly, the tools that perform this testing are called *software model checkers* in the systems literature, despite checking software implementations, not models.

A major benefit of software model checkers compared to non-systematic testing is that they can provide *formal* assurance in the form of coverage guarantees—*e.g.*, if the search explores all executions up to depth d and finds no errors, that means that no errors exist up to that bound.

Byzantine testing. Systems that are supposed to withstand Byzantine faults are trickier to test. The problem is that there is no principled way to generate *arbitrary* faults.

Twins, a recent approach for systematic testing of BFT systems, simulates Byzantine behaviour by running multiple instances of *correct* nodes with the same identity. These nodes run in "parallel universes" and thus produce different, potentially conflicting messages. Twins then replays these messages in the same execution—this is equivalent to the node equivocating. While this approach does not emulate *arbitrary* faults, it does catch many errors [Ban+20].

2.1.2 Checking Temporal Logic Specifications

Models take less effort to develop than implementations, and writing a model can help uncover design issues early in the development process. Moreover, especially for distributed systems, having to precisely specify correctness properties helps engineers focus on ensuring the system design works not only on the "happy path" on which no failures occur, but also in the presence of rare events [New+15]. However, all of this can be done with pen and paper, without computers.

Precise, testable system descriptions. The major advantage of computer-assisted modelling is that it produces system descriptions that can be *manipulated automatically*.[10]

An obvious way to manipulate a model is to systematically test it with a model checker. In the presence of a model checker, the model is not only a description of the system's design—it becomes a *tool* for reasoning about different potential designs.[11] In the words of AWS engineers, "a precise, testable description of a system becomes a what-if tool for designs, analogous to how spreadsheets are a what-if tool for financial models" [New+15].

Types of model checkers. Given a description of a protocol and a specification of its expected properties, there are multiple ways to *check* that the protocol meets the specification.

Most common are *explicit state* model checkers like SPIN [Hol97] and TLC [YML99]. These tools, starting from the initial states of the system, explicitly build a state transition graph via breadth-first search[12] and validate that the properties are upheld in all reachable states.[13] This approach is simple to understand and the reduction techniques discussed in the previous section can be directly applied.

An alternative approach is *symbolic* model checking, based on either Binary Decision Diagrams (BDD)[14] [Cla+96] or Boolean Satisfiability (SAT) [McM02]. Rather than represent states

10: The point is *not* obvious. The long-term benefit of formal methods will come not just in the form of "correctness" guarantees, but also from tools built *around* formal descriptions (*e.g.*, for model checking, symbolic execution, invariant inference, automated reasoning, synthesis and repair). Formal methods promise to reduce not only human error, but also human effort.

11: And with a program synthesiser, a model becomes a tool to generate implementations, *etc*.

12: Nested depth-first search is used to detect cycles when checking liveness properties [Hol18].

13: This is called *model checking* because the tool checks that the state graph is a model (in the logic, model-theory sense) of a temporal logic formula.

14: BDDs are a compact representation of boolean formulas.

and the state transition graph explicitly, symbolic approaches represent them as propositional formulas. This is typically more compact than an explicit representation. For this reason, symbolic model checkers can explore state spaces that are several orders of magnitude larger than what is feasible with an explicit state model checker [CG18].

Alloy Analyzer [JSS00] and Apalache [KKT19] are notable symbolic model checkers, used to validate specifications written in Alloy and TLA$^+$, respectively.[15]

15: Ivy and mypyvy, two tools we will discuss in Section 2.4, also implement (bounded) symbolic model checking.

Abstraction. By default, model checkers cannot verify the correctness of *infinite* state transition systems. For systems with infinite state spaces, both explicit and symbolic approaches bound the number of times the transition relation is unfolded.[16] In other words, they explore the state graph up to some finite depth. Nonetheless, perhaps surprisingly, we *can* use model checkers to verify infinite state systems.

16: For systems with *finite* state spaces, there is no need to bound the execution depth—after some time, all states will be explored.

The idea is to reduce, via a sound over-approximation, the check on an infinite state graph to a check on a finite state graph. This process is called *abstraction*.[17] Given a concrete model C, abstraction produces an abstract model A, such that if A satisfies some property P, then C satisfies it as well.[18]

17: With abstraction, the line between model checking, symbolic execution, automated theorem proving, and program analysis via abstract interpretation, becomes *very blurry*. The techniques we discuss in Section 2.4 mix insights from all these fields.

18: The reverse is not true. Since A is an over-approximation, there are properties which C satisfies, but A does not—the abstraction has "spurious counterexamples".

Abstraction is a field of study in its own right, but the major techniques are counterexample-guided abstraction refinement (CEGAR) [Cla+00] and interpolation [McM03].

Industrial use. Model checking has seen tremendous use in industry, particularly for verifying hardware designs [Bee+96; Eir96; Jan+97; PS99]. Interestingly, the adoption of model checking for hardware validation was driven by the same problem that led to the creation of software model checkers for distributed system implementations—testing was inadequate to catch deep errors [Kur18].

[Kur18]: Kurshan (2018), 'Transfer of Model Checking to Industrial Practice'

In the realm of distributed systems, companies like Amazon [New+15], Microsoft [Dem18; Lam18], Dropbox [Lam18], and MongoDB [SZT21] use model checking to validate designs.

2.1.3 Connecting Models to Implementations

Models are used to validate system designs. However, systems are not just designed; they are also implemented. It becomes important, then, to ensure that implementations match the designs they are derived from. This is a *different* problem than checking that the implementation is correct.[19]

For instance, we might have multiple implementations of a protocol, written in different programming languages, and want to ensure that they all implement the *same* protocol. Or maybe our model acts as documentation, and we must check that in remains in sync with the code.[20]

To handle this sort of problem, Gravell *et al.* proposed a programming methodology called *eXtreme Modelling* [Gra+11]. It combines both *model-based testing* [DF93] and *model-based trace checking* [JB83; How+11], with the aim of keeping specifications and implementations in sync. The former technique uses the model to generate tests that can be run against implementations, whereas the latter instruments implementations to capture traces that are then checked against the model.

Conformance monitoring. For distributed systems, model-based trace checking is particularly interesting, and is in fact used in industry [DHS20]. In real deployments, distributed systems collect logs that are used for runtime monitoring and realtime alerts. If a system stops processing requests, for instance, an engineer gets alerted to fix the issue. Trace checking enhances this workflow. System traces obtained from logs are fed into a model checker to make sure that the system behaves as expected [Hal20]. If it does not, an engineer can be alerted.

The benefit of this approach is that it detects not just catastrophic scenarios, or trends that are easily observed via metrics,[21] but also hidden issues which might indicate the existence of an implementation bug—and should therefore be looked at by an engineer.

19: Software model checkers, described previously, can test implementation correctness.

20: AWS describes TLA$^+$ models as "an excellent form of documentation" [New+15].

[DHS20]: Davis et al. (2020), 'Extreme modelling in practice'

[Hal20]: Halterman (2020), *Real-Time Conformance Monitoring with TLC*

21: *E.g.*, performance degradation or increased error rates.

2.2 Interactive Verification

As we have seen, most distributed systems cannot be exhaustively tested or model-checked.[22] For such systems, we can obtain formal correctness guarantees through interactive verification. We can encode a description of the system and its desired properties in an *interactive theorem prover*,[23] and show via a series of deductive steps that the system has the desired properties. This approach applies to both protocols and implementations, but requires a large amount of effort.

Much of the research in this area has, therefore, focused on ways to reduce the effort required to interactively verify distributed systems and their implementations.

22: We treat *abstraction refinement* as a method for automated verification, not for testing. We discuss invariant inference, the automated verification flavour of CEGAR, in Section 2.4.

23: Popular proof assistants include Coq, Isabelle, and Agda.

2.2.1 Refinement Using Verified System Transformers

One approach comes from Verdi, in the form of verified system transformers (VSTs) [Wil+15]. A typical way to prove the correctness of a distributed system is *refinement*—a system is correct if its behaviours[24] are a subset of those of another system that is known to be correct. For example, a state machine replication protocol is linearizable if all input/output traces it produces—even in the presence of faults—could have been produced by the unreplicated state machine.

A verified system transformer is a mechanism to transform a system into another, such that the transformed system refines the original. VSTs are generic—they can be applied to any existing system to add extra capabilities,[25] *e.g.*, the ability to tolerate dropped or reordered packets, or node crashes. The benefit is that the transformer only needs to be *verified* once, and then that proof can be reused in the correctness proofs of multiple systems (derived using the transformer), reducing the overall amount of effort required [Wil+15].[26]

The Verdi framework has been used to verify an implementation of Raft, extracted from Coq to OCaml [Wil+15].

[Wil+15]: Wilcox et al. (2015), 'Verdi: A Framework for Implementing and Formally Verifying Distributed Systems'

24: Behaviours are traces of inputs and outputs observed during the system's execution.

25: This is a form of *vertical* composition. The system is built by composing layers.

26: Verifying the transformers remains challenging, however. The Raft VST, which provides fault-tolerant state machine replication, took 18 months of effort to prove correct [Woo+16]. The challenge lies in establishing inductive invariants.

2.2.2 Compositional Reasoning With Separation Logic

Verified transformers let us reason about systems built via the vertical composition of features by reasoning about the correctness of each feature individually. But many distributed systems do not fit neatly within this pattern.

Some systems are composed of different protocols that interact. For example, a system might use two-phase commit (TPC) to maintain a replicated log, but also have a side-channel protocol for inspecting the state of TPC participants [SWT18]. The two protocols are coupled, so it is hard to express them as verified transformers. Nonetheless, we want to verify them separately and then reason about their composition, rather than having to verify the composed system as a monolith.

[SWT18]: Sergey et al. (2018), 'Programming and Proving with Distributed Protocols'

Composition of proofs is a difficult problem, but in the realm of program verification, separation logic [ORY01; OHe07] has proven a useful technique for compositional reasoning.[27]

27: Leslie Lamport is skeptical of program logics and argues that they are not necessary, as "any proof in mathematics is compositional" [Lam98a].

To provide compositional reasoning for distributed systems, Sergey *et al.* developed *distributed separation logic*, Disel, which adds an explicit notion of protocols to the logic and introduces rules tailored towards protocol reasoning [SWT18]. Later work added support for implementing systems with node-local concurrency, *i.e.*, multi-threading [Kro+20].

2.2.3 Linking Refinement and Separation Logic

Sprenger *et al.* propose Igloo, a methodology that combines the trace-based refinement of Verdi[28] with separation logic reasoning [Spr+20]. To verify a system, we first specify an *abstract model* of the system in terms of a trace property. Then, we build a *protocol model*, that describes communicating nodes, and show that this refines the abstract trace property.[29] We then decompose this into separate models for each node and the environment, and give each node a separation logic specification, which we use to verify its implementation.[30]

28: Trace-based refinement is often called "TLA-style refinement" in the literature, since TLA$^+$ can express refinement relations that can be model-checked by TLC.

29: Up to this point, the methodology is equivalent to Verdi.

30: The decomposition between protocol and node-level verification lets us use any framework to verify implementations, including, *e.g.*, automated verifiers. It also permits implementations in different languages to coexist.

2.3 Semi-Automated Verification

The problem with interactive verification is that it requires significant human effort to write proofs. This burden can be eased by *automated theorem provers*, but for most proofs about distributed systems, ATPs still require human input to guide them.[31] Humans need to provide a high-level proof structure, in which the ATP can fill in the details, and more importantly, they need to discover *inductive invariants* that capture the system's safety properties. This requires deep insight and is challenging even for experienced researchers, taking days for simple systems and months for complex protocols.

31: For instance, SMT solvers can work with undecidable logics, but that means they are not complete. Solvers can automatically prove low-level theorems that are cumbersome but do not require much insight, but need guidance for complex proofs.

2.3.1 Deductive Verification of TLA Specifications

An example of a semi-automated proof system is TLAPS, the TLA$^+$ proof system [Cha+08; Cha+10; Cou+12].

TLAPS lets users write hierarchical proofs for safety properties of TLA$^+$ models, and passes these proofs to backend verifiers to check their validity. Optionally, backends can attempt to fill in any gaps in the proofs. Indeed, a common way to prove theorems is to use the `OBVIOUS` command, which instructs the backend to fill in the entire proof. This typically fails, so the user has to provide the structure of the proof before the solver can handle the proof's cases automatically.

TLAPS has been used to prove the safety of Byzantine Paxos[32] [Lam11], Multi-Paxos [CLS16], and Pastry [AMW18].

32: Byzantine Paxos was proven safe by showing that it refines the regular Paxos protocol.

2.3.2 Safety and Liveness Proofs for Implementations

IronFleet is a methodology for semi-automated verification of both safety and liveness properties of distributed system implementations [Haw+15]. It combines TLA-style refine-

[Haw+15]: Hawblitzel et al. (2015), 'IronFleet: proving practical distributed systems correct'

ment for reasoning about abstract protocols with Hoare-style verification of the code at each node.[33]

IronFleet is implemented in Dafny [Lei10], a language designed for automated theorem proving, and includes an embedding of TLA in Dafny to verify liveness. Notably, Hawblitzel *et al.* develop a set of SMT solver *heuristics* for TLA liveness proofs which allow the solver to automatically prove many low-level goals. Indeed, they report that a single heuristic is "enough to automatically prove 40 fundamental TLA proof rules", and complex rules require only a few manually written lines of proof [Haw+15].

IronFleet was used to produce an implementation of Multi-Paxos verified for both safety and liveness.

33: The Igloo methodology we reviewed earlier is heavily inspired by IronFleet.

2.3.3 Finding Invariants by Interactive Generalisation

TLAPS and IronFleet ease the burden of writing proofs, but insight is still required to discover inductive invariants. To help with this, Padon *et al.* developed Ivy [Pad+16b], a tool that *guides* users towards discovering inductive invariants.

[Pad+16b]: Padon et al. (2016), 'Ivy: safety verification by interactive generalization'

This is more challenging than it sounds. In general, it is undecidable whether a given proposition is an inductive invariant or not. The way invariants[34] are discovered manually is by trial and error—we come up with a candidate, try to prove that it is inductive, and in most cases, we *fail*; we then try another candidate and repeat the process until we succeed. The limiting factor is that proofs take a long time to develop, so we might spend weeks before we realise that our candidate invariant is not inductive. It is a very slow process.

34: For conciseness, in this section, we use the term "invariant" to refer to inductive invariants.

Decidable proofs. Padon's insight was that *if* the proofs can be automated, this process can be greatly sped up, and moreover, the computer can assist the human verifier by showing exactly how the inductiveness proof fails for the

candidate invariant—thereby helping formulate a new candidate. To make this possible, Padon *et al.* introduced RML, a modelling language inspired by Alloy [Jac02], which guarantees that verification conditions for inductiveness checking are *decidable*,[35] and thus the inductiveness proofs can be fully automated [Pad+16b]. When the proof fails, the tool displays a counter-example to induction, and the user must come up with another candidate invariant that does not have this counter-example. This continues until the user arrives at an inductive invariant that establishes the desired property.[36]

35: Specifically, it ensures that verification conditions fall into an *extension* of the Bernays-Schönfinkel-Ramsey fragment of first-order logic known as EPR [PdMB10]. The extension is called extended EPR [Pad+17].

36: Ivy only helps discover universally quantified invariants.

The importance of Ivy. RML is limited in terms of the properties it can express whilst remaining decidable. Nonetheless, Ivy gives us a deep insight about how to fully automate correctness proofs for distributed systems.[37]

If we develop a way to discover inductive invariants without human interaction, then we have a *fully automated* proof system for any protocol expressible in RML. The challenge, then, becomes one of (1) expressing interesting protocols in this restricted fragment of first-order logic[38] and (2) automatically discovering inductive invariants for such protocols.

37: Ivy models can be extracted into C++ code to obtain executable implementations [MP20].

38: Padon explores how to specify complex protocols within this decidable fragment of first-order logic in later work [Pad+17], but his method is manual.

Abstraction. Interestingly, both the problem of expressing a system in extended EPR and that of finding an inductive invariant can be understood in terms of abstraction.

Not all constructs can be expressed in EPR, so we must find alternatives that capture in sufficient detail the aspects we want to describe. For example, set cardinalities cannot be expressed in EPR, but are useful to reason about quorums in distributed protocols. However, often the only property we care about is quorum intersection, which *can* be expressed. Abstracting the property lets us automate reasoning.[39]

Similarly, an inductive invariant is an abstraction of the transitive closure of the system's transition relation. The challenge is to find an abstraction that captures sufficient detail to imply the desired safety properties.

39: Reducing from undecidable to decidable logics is in principle similar to abstracting from infinite to finite transition systems.

2.4 Automated Verification

We want to build computer tools that can automatically reason about systems.[40] For this to be possible, we must have precise, formal descriptions of the systems we are interested in. But not all formal descriptions are *suitable* for automated reasoning. Indeed, as hinted already, the key enabler of automated verification is abstraction—which is, in effect, the process of transforming a formal description that is not suitable for automated reasoning into one that is. This comes at the cost of introducing imprecision, which is why we need to be careful when choosing an abstraction to ensure it captures sufficient detail to imply the desired properties.[41]

40: In this section, we focus on safety verification via invariant inference, but other kinds of tools, *e.g.*, program synthesisers, are similar in principle.

41: As Clarke *et al.* put it, "abstraction amounts to removing or simplifying details as well as removing entire components of the original design that are irrelevant to the property under consideration" [Cla+03].

In the previous section, we mentioned two kinds of abstraction that make possible the automated verification of distributed systems—translating specifications into a decidable fragment of first-order logic and automatic invariant inference. To our knowledge, no work has been done to automate the former kind, so in this section we focus exclusively on techniques for *automated inference of inductive invariants*, that is, finding sound over-approximations of the transitive closure of a system's transition relation.

2.4.1 Property-Directed Reachability

An approach that came out of the symbolic model-checking world is *property-directed reachability* (PDR) [Bra11; EMB11]. The idea is, instead of unfolding the transition relation a fixed number of times (thus only being able to model-check correctness), *incrementally* construct an inductive invariant that over-approximates reachability (*i.e.*, the transitive closure of the transition relation) and is strong enough to imply the desired property—hence the name of the approach.

[Bra11]: Bradley (2011), 'SAT-Based Model Checking without Unrolling'
[EMB11]: Een et al. (2011), 'Efficient Implementation of Property Directed Reachability'

Bradley proposed an algorithm called IC3 [Bra11], a specific instance of the PDR approach.[42] The algorithm constructs a sequence of formulas (called *frames*), $F_0, F_1, F_2, \ldots, F_k$, that over-approximate the sets of states reachable in at most

42: The original algorithm was later extended in various ways, as described in [Gur15].

$0, 1, \ldots, k$ steps. Throughout the execution of IC3, each formula F_i is an invariant (expressed as a conjunction of relatively inductive clauses) that implies the property ($F_i \Rightarrow P$) and maintains the approximation ($F_i \wedge T \Rightarrow F'_{i+1}$). The algorithm terminates when the over-approximation reaches a fixpoint, *i.e.*, when $F_i = F_{i+1}$.[43] At that point, F_i is an inductive invariant that implies the desired safety property.

The key part of the IC3 algorithm[44] is the construction of clauses, which is driven by counterexamples, in the style of CEGAR [Cla+03]. Initially, F_1 is set to P, the desired property (assuming no 1-step counterexamples exist). The algorithm tries to expand the over-approximation sequence $F_0, F_1, F_2, \ldots, F_k$, so it checks whether $F_k \wedge T \Rightarrow P'$.[45] If yes, it sets $F_{k+1} = P$. Every formula F_{i+1} is then refined by adding to it clauses c from F_i that are maintained by the transition ($F_i \wedge T \Rightarrow c'$)—this is called *propagation*. If no, then there exists a state in F_k that transitions into a bad state s that does not satisfy P. IC3 then refines the over-approximation sequence by finding the highest i such that $\neg s$ is relatively inductive to F_i[46] and adding to $F_0, \ldots, F_i$[47] an *inductive generalisation* of $\neg s$.[48] If this addition breaks the approximation ($F_i \wedge T \not\Rightarrow F'_{i+1}$), the algorithm recurses on the new counterexample, until all counterexamples up to F_k are eliminated and k can be increased.

For finite state systems, IC3 always terminates and produces an inductive invariant—in the worst case, the conjunction of all bad states, but typically a much more reasonable one.

Universal PDR. A similar approach, PDR$^\forall$ [Kar+17], works for infinite state systems, by generalising the form that invariant clauses take. Concretely, if invariants are conjunctions not of pure propositional formulas as in IC3, but rather conjunctions of *universally quantified* formulas, then it is possible to use roughly the same algorithm for systems with an infinite state space. The difference lies in how one generalises from counterexamples. In PDR$^\forall$, Karbyshev *et al.* employ the model theory notion of a *diagram* to generalise individual

43: For finite state systems, a fixpoint will always be reached. Bradley's approach is for finite state systems, but a related approach, UPDR, is guaranteed to terminate on certain classes of *infinite* systems [Kar+17].

44: IC3 stands for "incremental construction of inductive clauses for indubitable correctness".

45: P' refers to P in the post-state of the transition relation T.

46: Intuitively, F_i is the weakest (containing the largest number of states) over-approximation that can be strengthened to eliminate s. In the worst case, this is F_0, the initial state.

47: All approximations before F_i are stronger, so a clause that is inductive relative to F_i is necessarily inductive relative to them.

48: A generalisation of $\neg s$ is a clause that eliminates s but is weaker than it. To generalise, Bradley removes literals from $\neg s$ until no more literals can be removed without the clause ceasing to be relatively inductive to F_i [Bra11]. A cheaper option is to use the UNSAT core returned by the SAT solver [EMB11].

states into sets of states [Kar+17].[49] Concretely, when PDR$^\forall$ tries to expand its over-approximation sequence by checking whether $F_k \wedge T \Rightarrow P'$, if it identifies a counterexample to safety (the implication does not hold), it generalises the counterexample via its diagram and strengthens previous frames accordingly,[50] in a fashion similar to IC3.

PDR$^\forall$ works on logics that have the *finite model property*, that is, in which satisfiable formulas have models of finite size—this is a precondition for diagrams to be well-defined.[51] While the algorithm is not in general guaranteed to terminate, *if* it terminates, it has the nice property that it either produces a universally quantified inductive invariant that implies safety, or shows that the system admits no such invariant. In this sense, diagram-based abstraction is *precise*. The abstraction introduces spurious counterexamples (that exist in the abstraction, but not in the concrete system) only if $\forall$ formulas are not sufficient to approximate the system. Moreover, PDR$^\forall$ has been shown to always terminate on certain classes of programs, *e.g.*, those manipulating singly-linked lists.[52]

49: The diagram of a state s is an existentially quantified formula that is the conjunction of *all* atoms or negations of atoms that are true in s—it captures all equalities and inequalities among elements in the model, as well as the truth value of all possible predicate applications. Intuitively, a diagram is a full description of the state, from which all facts about it can be derived.

50: The diagram is an existentially quantified formula. Its negation, which gets added as an invariant clause, is therefore universally quantified.

51: Extended EPR has the finite model property.

52: Invariant inference for such programs was previously shown to be decidable [Pad+16a].

2.4.2 Incremental Inference of Universal Invariants

In Ivy, in the usual process, when a counterexample to induction is found, the human verifier has to manually generalise it—retaining only the essential features of the counterexample.[53] Once the user conjectures a generalisation, however, Ivy can strengthen it via a technique called *BMC + Auto Generalize* [Pad+16b]. To do this, Ivy model-checks the conjecture up to some user-chosen execution bound, verifying that it holds in all reachable states up to that bound. If this check passes, Ivy then uses the *minimal UNSAT core* returned by the solver to strengthen the conjecture—retaining only its essential parts. The risk of this approach is that, if the user selects a bound that is too small, the discovered generalisation can be bogus, failing to hold at higher execution depths.

53: We say "in the usual process" because Ivy can in fact generalise counterexamples to induction via diagrams and validate the correctness of the generalisation via BMC [Pad+16b]. We describe the process in Chapter 4.

Automating the Ivy loop. Ma *et al.* observed that this process can be automated [Ma+19].[54] Ivy needs the user to suggest an initial conjecture, but then it does *not* require human assistance—apart from selecting a BMC bound—to strengthen it and use it to prove the system's safety.

The idea behind I4 is that we can obtain the initial conjecture automatically—and not just one clause of the invariant, as in Ivy, but indeed a complete candidate inductive invariant can be obtained automatically. The key insight is that invariants of finite instances of the system tend to be very similar to invariants of infinite instances.[55] We can use an algorithm like IC3 to discover an invariant of the finite instance, and then generalise it via *syntactic templates*. This is exactly what I4 does. It finitises the specification, uses the AVR model checker [GS19; GS20] to come up with an inductive invariant for the finite instance, and generalises the invariant via a set of syntactic heuristics or templates [Ma+19].[56]

The I4 process does not always succeed in producing an inductive invariant that implies the safety property—this can happen either because the finite instance size was too small or because the syntactic template for a clause generalised it incorrectly. To fix this, I4 prunes clauses in an attempt to remove instance-specific clauses from the generalised invariant (the second kind of error), and if that fails, restarts with larger bounds on the domains (the first kind of error).

Ma *et al.* do not directly compare their approach to PDR$^\forall$, which also automatically infers universally quantified inductive invariants, but evaluation in later work shows that, for simple benchmarks, I4 and PDR$^\forall$ behave similarly [GS21a].

[Ma+19]: Ma et al. (2019), 'I4: incremental inference of inductive invariants for verification of distributed protocols'

54: An easy way to automate the Ivy loop is to pick one of the possible conjectures that passes BMC with a large bound. Indeed, Ivy supports this natively, as we describe in Chapter 4. With this approach, however, there is no way to identify that a wrong clause was added.

55: The Ivy *BMC + Auto Generalize* process bounds the execution depth of the system. I4 instead bounds the domains used in the system, *e.g.*, instead of unbounded integers, use $\{0, 1, 2\}$. This gives a finite state space.

56: The syntactic templates are quite simple. They universally quantify every clause in the finite invariant and add a distinctness condition for elements of the same sort.

2.4.3 Data-Driven Invariant Learning

Another, very different, approach that generalises from small instances is DistAI [Yao+21]. I4 picks an instance size and then discovers an inductive invariant for that finite instance and tries to generalise it. DistAI does something different.

[Yao+21]: Yao et al. (2021), 'DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols'

Instead of choosing an instance size, it picks a formula size (of the candidate inductive invariant) and enumerates formulas of that size.[57] This approach *will* eventually find an inductive invariant that implies safety, but it is impractical—it requires too many SMT queries to check the inductiveness of the candidate invariants. To make this search feasible, DistAI introduces two novel techniques, and proceeds as follows.

First, DistAI builds up a set of candidate[58] (not necessarily inductive) invariants of the given size. To do this, it uses a novel data-driven approach. It generates many executions of differently-sized instances of the system, and collects the set of reachable states—this is the data the algorithm operates with.[59] These samples could be used to quickly filter out bad candidate invariants without needing to query the SMT solver. Instead, achieving the same effect but with better performance, candidates are generated directly from the samples using an invariant template. This is done efficiently by dividing the template into subtemplates that exploit the symmetry of the search space to reduce the number of candidates generated. Overall, DistAI generates the set of all invariants *of the given size* that hold over the sample.

The crucial observation is that, if an *inductive* invariant of this size exists for the infinite system, it is necessarily weaker than the conjunction of the clauses in the generated set.[60]

This observation enables the second, similarly novel, part of the algorithm, which monotonically weakens the clauses in the set until their conjunction becomes an inductive invariant that implies safety. For every clause that fails to be invariant, DistAI applies a *minimum weakening*—it (1) removes the clause from the invariant set, and (2) finds all minimally weakened versions of the clause (by adding *one* atom as a disjunction) and adds back to the set all weakened clauses that are not implied by some other clause in the set.[61] Before addition to the set, these weakened clauses can be filtered via counterexamples. This weakening process continues until an inductive invariant that implies safety is found. If none

57: DistAI only enumerates universally quantified invariants. A formula's size is the number of (quantified) variables and literals in the formula, *e.g.*, 2 variables and 2 literals.

58: These invariants, which are universal clauses, are not validated by the SMT solver, so they might be incorrect.

59: Each state stores the values of all predicates, much like the diagrams used by PDR$^{\forall}$.

60: The set contains *all* invariants that hold over the sample. This includes all actual invariants, plus some incorrect ones.

61: The weakening procedure also has third step, that projects the failed clause to higher subtemplates and adds these projections to the set. This is a technical detail that arises due to how candidates are enumerated in the first DistAI phase. The broken clause implied certain other clauses, which were not enumerated, but its removal requires readding these clauses to the set to maintain its completeness.

is found, that implies none of chosen size exists, and the algorithm is restarted with a higher formula size.

DistAI always terminates if the system can be verified using a universally quantified invariant[62] and has been used to automatically verify a number of simple protocols [Yao+21].

62: DistAI never generates undecidable verification conditions, so SMT queries always terminate, and systems have inductive invariants of finite size, which will be eventually found.

2.4.4 Invariants with Quantifier Alternations

A major limitation of the approaches discussed so far is that they only discover invariants that are universally quantified. These suffice to verify simple protocols, but are not strong enough to capture the behaviour of more complex protocols like Paxos [Lam98b] and Raft [OO14]. These have invariants with quantifier alternations, particularly of the $\forall^*\exists^*$ kind. Various techniques have been developed to automatically discover such invariants.

SWISS. A technique similar to DistAI is SWISS [Han+21]. It leverages the same observation—that invariants of distributed protocols tend to be concise (*i.e.*, small)—and searches the space of possible invariants via syntactic templates that define the quantifier structure of the clause.[63] To speed up the search, SWISS exploits the symmetry of the search space and learns from counterexamples, only producing candidates that are not equivalent to previous ones and which do not violate known counterexamples.

[Han+21]: Hance et al. (2021), 'Finding Invariants of Distributed Systems: It's a Small (Enough) World After All'

63: These templates are either provided by the user or hard-coded into the SWISS tool.

There is, however, a significant difference between SWISS and DistAI.[64] Whereas DistAI produced the strongest possible candidate invariant and then gradually weakened until it became inductive, SWISS takes a different approach.

64: SWISS and DistAI were developed concurrently, with SWISS published first.

In its first phase, called Breadth, it generates (non-redundant) invariant clauses that are *relatively inductive* previously discovered ones.[65] This builds an invariant incrementally, but without reference to the desired safety property.

65: This is similar to how IC3 operates, constructing an inductive invariant incrementally [Bra11].

In its second phase, Finisher, SWISS attempts to strengthen[66] the invariant produced by Breadth with a new clause, such

66: Strengthening vs weakening reflects SWISS searching bottom-up and DistAI top-down.

that the strengthened invariant is inductive and implies safety.[67] Finisher can be applied on its own, to attempt to discover a strong enough inductive invariant in one shot, but this is typically too difficult for complex protocols—the incremental, two-phase approach performs better.

Notably, SWISS is the first algorithm to verify Paxos and Flexible Paxos automatically, and Multi-Paxos semi-automatically (with a user-given invariant template) [Han+21].

67: Crucially, SWISS succeeds only if the invariant produced by Breadth is weaker than the inductive invariant that implies safety, *i.e.*, only if the desired inductive invariant can be produced via strengthening. In contrast, DistAI's approach always succeeds.

Quantifier alternations without templates. The problem with SWISS is that it requires the user (or, equivalently, the tool developer) to provide the quantifier structure of the invariants as input via templates. As such, the method is not *fully* automated. Two works address this issue.

The first is FOL-IC3 [Koe+20], which preceded SWISS. It uses the notion of *separators* such that the human verifier does not need to provide explicit invariant templates, but only the number of quantifiers required.[68] If any invariant with the given number of quantifiers (regardless of its alternation structure) exists, FOL-IC3 will find it. This was, in fact, the first approach capable of finding invariants with quantifier alternations, but it is slow, so it did not scale to Paxos.

[Koe+20]: Koenig et al. (2020), 'First-order quantified separators'

68: This still requires human input, therefore, but arguably a kind of input that requires less insight.

An alternative, data-driven approach is IC3PO [GS21a]. This discovers quantifier templates from executions of finite instances of the system. The key insight, which resulted in a technique called *symmetry boosting*, is that symmetries between invariant clauses of finite instances reveal quantifier alternation patterns for invariants of the infinite instance. In further work, Goel *et al.* extended this with *range boosting* to support learning quantified invariants involving total orders (*e.g.*, ballot numbers) from finite instances [GS21b].

[GS21a]: Goel et al. (2021), 'On Symmetry and Quantification'

[GS21b]: Goel et al. (2021), 'Towards an Automatic Proof of Lamport's Paxos'

Given a hierarchical specification of Paxos (in terms of higher-level protocols), this is sufficient to automatically discover invariants for Paxos, Flexible Paxos and Multi-Paxos without any human input [GS21b].

2.5 Gaps in the State of the Art

Our survey is by no means complete, but it nonetheless provides a broad overview of the field. We explained current techniques for building assurance in distributed protocols and their implementations, and placed these techniques in context, showing how they are related to each other and giving a hint of their relative advantages and disadvantages.

In the remainder of this chapter, we aim to highlight some of the ways in which the state of the art falls short. Put differently, we identify areas of potential improvement that future work might choose to address.

The implementation formality gap. There is a large gap between our ability to reason about protocols and our ability to reason about implementations. Two approaches, software model checking and conformance testing and monitoring, seek to address this problem, but fall short in the same way.

Software checkers can provide formal assurance about implementations, in the form of coverage guarantees, but are not complete—they cannot verify systems. On the other hand, protocol models can be verified using abstraction-based techniques,[69] but even if they are, there is no way to carry this assurance over to the implementation, because both testing and monitoring approaches are incomplete.

69: Although in typical practice they are not verified, but only model-checked. The next named paragraph addresses this issue.

There is a *gap* in the level of formal assurance we can obtain for implementations, as compared to protocols. Researchers have closed this gap by extracting implementations from specifications that were either interactively [Wil+15; LBC16], semi-automatically [Haw+15], or automatically verified [MP20]. However, no one is using these extracted implementations in practice and it is unlikely that anyone ever will. This is the case because the extraction approach completely upends the normal software development lifecycle—it expects organisations to abandon existing implementations and technologies

and switch to a correctness-focused rather than a product-focused mode of operation [Rei+20]. Moreover, the first two approaches require unscalable amounts of human effort.

[Rei+20]: Reid et al. (2020), 'Towards making formal methods normal'

A potential solution to the formality gap that might see more industrial adoption is based on abstraction. An Ivy-style decidable specification could be used to automatically infer system invariants that imply the desired safety properties, and these invariants could be checked in the implementation via, for example, symbolic execution.[70] In effect, this would check that both spec and implementation satisfy the same abstraction, and it opens many avenues of potential research. For instance, how can this correspondence be maintained in the face of *software evolution*? Can patches to the specification be automatically derived from implementation changes, or vice-versa?[71] Can parts of the implementation that do not conform to the specification be *repaired* automatically?

70: This not a direct translation, as the way state is represented differs between the implementation and specification, but it might be possible to *learn* the translation scheme.

71: That implementations and specifications are written in different languages is a barrier to maintaining their correspondence. In fact, recent work in industry uses reference implementations, written in the same language as the main codebase, instead of TLA$^+$ models, to ameliorate this issue [Bor+21]. However, this foregoes automatic reasoning about specifications.

Automation-unfriendly specifications. In Section 2.1.2, we speculated that the long-term benefit of formal methods will come from tools built *around* formal descriptions, *e.g.*, tools for symbolic execution, program synthesis and repair, *etc.* And just now, we saw how such tools might address deep gaps in the state of the art. However, this vision faces an obstacle in the fact that most current specifications are written in languages like TLA$^+$, that are not suited for automated reasoning. Moreover, writing specifications such that their safety can be verified automatically via decidable reasoning is *unnatural*—specifications amenable to automated reasoning typically cannot be expressed in the normal "operational" style and require expertise to develop [Pad+17].

It would be a significant advance to be able to automatically rewrite "normal" specifications into specifications suitable for automated reasoning. Data-driven invariant inference tools like I4 [Ma+19], DistAI [Yao+21], and IC3PO [GS21a] provide a clue as to how this might be achieved—after all, as we established in Section 2.3.3, invariant inference and expressing specifications in EPR are both forms of abstraction.

It might be possible, given an operational specification of a protocol, to execute it on small instances and thereby *learn* a specification whose verification conditions fit within EPR. This would open up many different ways of using *existing* specifications, besides model checking them.

Synthesis of new protocols. One consequence of having decidable protocol specifications is that it becomes possible to manipulate them in interesting ways.

For instance, most protocols are not developed from scratch. Rather, they are obtained by adapting existing protocols. An example of this is HotStuff [Yin+19], a variant of PBFT [CL99] that executes a view-change and checkpoint after every commit. It might be possible to automatically derive HotStuff given a specification of PBFT. After all, the protocols are not very different—they satisfy the same specification, but HotStuff obeys additional constraints.[72] Similarly, given a specification of HotStuff and synchrony assumptions, it might be possible to derive Sync HotStuff [Abr+20]. This is enabled by automated reasoning—if we can automatically *check* that a protocol is correct (via invariant inference and decidable reasoning), then we can synthesise new protocols, either from scratch or by adapting existing ones.[73]

72: Many systems papers published in the last decade describe what are, in effect, classic protocols re-engineereed to work under non-standard assumptions.

73: Indeed, a major motivation for formal specifications in industry is to enable engineers to confidently optimise existing protocols [New+15].

Protocol composition. Current approaches struggle with reasoning about the composition of different protocols. Techniques based on separation logic [SWT18; Kro+20; Spr+20] alleviate the problem, but do not fundamentally solve it. Composition has also been studied in the context of automated reasoning, although in the form of *decomposing* protocols into components to enable decidable verification [Tau+18].

[Tau+18]: Taube et al. (2018), 'Modularity for decidability of deductive verification with applications to distributed systems'

It would be interesting to explore how automated techniques like invariant inference could help reasoning about protocol composition, *e.g.*, by generating invariants for a composed system based on those of its components.

3 Preliminary Work

In this chapter, we briefly review our previous work with formal methods and distributed systems.[1] Moreover, we explain how different methods compare, based on our experience.

1: Excepting the work on certified synthesis [Wat+21], all work described here was conducted *before* the author joined NUS.

Mechanising blockchain consensus. In 2018, we presented Toychain, the first formalisation of Nakamoto consensus with a proof of its consistency mechanised in an interactive proof assistant [PS18]. We used the Coq proof assistant to specify the protocol, and define our network model in the same small-step style that Verdi [Wil+15] and Disel [SWT18] use. Concretely, we represent the state of the "world" as an inductive datatype whose constructors are the *events* that can occur, *e.g.*, delivering or dropping a packet, performing a local computation at a node, *etc*. Nodes' message handlers are specified in a pure functional style, so our specification is executable—it can act as a reference implementation.

[PS18]: Pîrlea et al. (2018), 'Mechanising Blockchain Consensus'

We prove a basic form of eventual consistency called *quiescent consistency*, which states that when all messages have been delivered, then protocol participants agree on the current chain. Despite this being a weak property for a relatively simple protocol, it took us over 2 months of effort to discover and establish an inductive invariant that implied it.[2]

2: It would be interesting to verify Toychain in a framework that supports automated invariant inference and decidable proofs, and compare the relative effort.

This involved a process of trial and error, gradually refining the invariant so that is neither too weak (not implying the desired property) nor too strong (not maintained by every transition). The challenge was, that, once we came up with an alternative candidate invariant, it usually took a few days to progress sufficiently in the proof to realise that the invariant is inadequate.

In later work, we relaxed some of the assumptions that the original specification relied upon, and extracted the protocol specification into an executable OCaml program [Pîr19].

[Pîr19]: Pîrlea (2019), 'Toychain: Formally-Verified Blockchain Consensus'

Solver-assisted reasoning. We also experimented with F⋆ [Swa+16], a proof assistant similar to Dafny [Lei10], capable of automatically discharging certain proof obligations by invoking an SMT solver. We developed a formal model of Coco, a blockchain system at Microsoft Research, to increase assurance in the design—we proved that the protocol, under some assumptions, satisfies sequential consistency. Interestingly, while the model was not formally connected to the implementation, the process of designing the model helped us discover a concurrency bug in the implementation that would have compromised safety. F⋆'s automation facilities were useful, as they helped us focus on the high-level proof whilst the SMT solver handled the cumbersome parts.

Informal reasoning. While developing CoSplit [PKS21], a static analysis tool that improves blockchain scalability by parallelising the execution of smart contract transactions across shards, we discovered an error in the design of the blockchain protocol we integrated our tool with. Concretely, when we tried to evaluate CoSplit's impact on transaction throughput, we noticed that the system—even without our modifications—would stall when put under heavy load. High transaction loads produced liveness failures. The root issue was simple and might have been spotted if a formal model of the system existed: the implementation relied on a timeout (*i.e.*, synchrony assumption) instead of correctly implementing PBFT's commit phase. However, it was difficult to pin-point the issue in a large codebase, because our mental model of how the system worked did not match reality.[3]

[PKS21]: Pîrlea et al. (2021), 'Practical smart contract sharding with ownership and commutativity analysis'

3: Having a model that was connected to the code might have helped us identify the root cause faster. But it is unclear whether conformance testing would have discovered the discrepancy between the implementation and the intended design, as the issue relied on timing to manifest.

Synthesising certified implementations. Finally, with the Certified Suslik project [Wat+21], we extended an existing program synthesiser to produce not just programs, but also machine-checked proof certificates. While this work is not directly related to distributed systems, it would be interesting to develop synthesisers for such systems, automatically generating verified implementations from specifications.

[Wat+21]: Watanabe et al. (2021), 'Certifying the synthesis of heap-manipulating programs'

Towards Automated Reasoning

4

In this chapter, we present our initial experiments with automated reasoning and invariant inference.[1] We experimented with two tools for decidable reasoning, Ivy [Pad+16b] and mypyvy [Fel+19]. We used Ivy's graphical interface to discover invariants for two-phase commit and used mypyvy to encode a Raft-style version of Multi-Paxos, as described in a recent comparison of the two protocols [HM20].

1: We also experimented with model-based testing of the popular *etcd* implementation of Raft consensus. That was part of a group project, however, and is therefore not covered in this report, which focuses on individual contributions.

4.1 Guided Invariant Discovery

For our first experiment, we manually translated a TLA$^+$ specification of two-phase commit (TPC) into Ivy. The resulting specification is shown in Figure 4.1.

Our goal was to understand how Ivy's counterexample-guided invariant discovery process operates and compare it to the manual, non-assisted approach. To this end, we first manually came up with an inductive invariant, clause by clause. Afterwards, to compare, we used Ivy's graphical interface, being presented with counterexamples to induction and generalising them into invariant clauses, until the conjunction of the clauses became inductive.[2]

2: In Ivy, the safety property is implicitly one of the clauses of the invariant, so whenever the invariant becomes inductive, safety is established.

The key property in TPC is *consistency*, stated as follows:

```
invariant [consistency]
  forall X:replica, Y:replica.
    ~ (reState(X) = aborted & reState(Y) = committed)
```

This says that replicas are never in conflicting states, *i.e.*, it is impossible for one replica to have committed the transaction if any other replica aborted, or vice-versa. It might still be the case, however, that replicas never make a decision.

This property is invariant, but not inductive. We need to strengthen it with other clauses until it becomes inductive.

```
1  type replica
2  type coordinatorState = {start, done}
3  type replicaState = {working, prepared, aborted,
       committed}
4
5  individual coState : coordinatorState
6  function reState(R:replica) : replicaState
7  function sentPreparedMsg(R:replica) : bool
8  function rcvdPrepareFrom(R:replica) : bool
9  individual sentCommitMsg : bool
10 individual sentAbortMsg : bool
11
12 after init {
13     coState := start;
14     reState(R) := working;
15     sentPreparedMsg(R) := false;
16     rcvdPrepareFrom(R) := false;
17     sentCommitMsg := false;
18     sentAbortMsg := false;
19 }
20 action coRcvPrepared(r:replica) = {
21     require coState = start
22         & sentPreparedMsg(r);
23     rcvdPrepareFrom(r) := true;
24 }
25 action coCommit = {
26     require coState = start
27         & forall R. rcvdPrepareFrom(R);
28     coState := done;
29     sentCommitMsg := true;
30 }
31 action coAbort = {
32     require coState = start;
33     coState := done;
34     sentAbortMsg := true;
35 }
36 action rePrepare(r:replica) = {
37     require reState(r) = working;
38     sentPreparedMsg(r) := true;
39     reState(r) := prepared;
40 }
41 action reAbort(r:replica) = {
42     require reState(r) = working;
43     reState(r) := aborted;
44 }
45 action reRcvCommit(r:replica) = {
46     require sentCommitMsg = true;
47     reState(r) := committed;
48 }
49 action reRcvAbort(r:replica) = {
50     require sentAbortMsg = true;
51     reState(r) := aborted;
52 }
```

Figure 4.1: Ivy description of the TPC protocol. The protocol has a single coordinator, whose state is represented as the singleton (**`individual`**) on line 5. There are multiple replicas, so we use a **`function`** to represent their state, mapping from each replica to its state. Similarly, for each replica, we store whether it sent a "prepared" message to the coordinator (line 7), and for the coordinator, we store whether it received such a message *from* each replica (line 8). We also keep track of whether the coordinator sent the "commit" or "abort" messages—these are implicitly broadcast so are represented as singletons (lines 9 and 10). Every **`action`** describes a transition, which can only trigger if its precondition (**`require`**) is satisfied, and produces an effect on the state. The **`after init`** declaration (line 12) describes the initial state of the system.

```
invariant [send_before_receive]
    rcvdPrepareFrom(R) -> sentPreparedMsg(R)
invariant [decision_mutually_exclusive]
    ~(sentCommitMsg & sentAbortMsg)
invariant [decision_iff_done]
    (sentCommitMsg | sentAbortMsg) <-> coState = done
invariant [commit_implies_sentPrepared]
    sentCommitMsg -> sentPreparedMsg(R)
invariant [sentPrepared_implies_not_working]
    sentPreparedMsg(R) -> ~(reState(R) = working)
invariant [if_prepared_only_coordinator_can_abort]
    sentPreparedMsg(R) & ~sentAbortMsg -> ~(reState(R) = aborted)
invariant [cannot_abort_if_coordinator_committed]
    sentCommitMsg -> ~(reState(R) = aborted)
invariant [cannot_commit_if_coordinator_not_done]
    ~(coState = done) -> ~(reState(R) = committed)
invariant [cannot_commit_if_coordinator_aborted]
    sentAbortMsg -> ~(reState(R) = committed)
```

Figure 4.2: Manually discovered invariant clauses for TPC, that together with consistency, are inductive. In Ivy, identifiers with capital letters (*e.g.*, `R`) are implicitly universally quantified.

Indeed, as soon as we invoke `ivy_check`, the tool tells us that consistency is not inductive.[3] In other words, there exist states that are consistent but which can transition into states that are not—called *counterexamples to induction* (CTIs). The problem is that the property does not sufficiently constrain the components of the protocol's state to establish that no transition into a bad state can happen. To establish safety, we need to strengthen it by adding clauses that constrain the state space sufficiently to eliminate all CTIs.

3: Concretely, we are told that the property is not maintained by the `reAbort`, `reRcvAbort`, and `reRcvCommit` transitions.

Intuition. Intuitively, by adding clauses, we are translating the *operational* description of the protocol, given in terms of its transitions, into a *static* over-approximation of the set of reachable states. We want to encode operational knowledge, of the form "this state can transition into that state", into static knowledge of the form "a state like this cannot be reached".[4] Interestingly, this is how experts tend to think about protocols. An invariant captures an understanding (description) of the system in terms of what states the system disallows. The invariant is inductive when the description is closed under transitions—there is no way to "escape" it.

4: An invariant is an *over-approximation* of the set of reachable states. To aid understanding when talking about over-approximations, it often helps to describe their negations, *i.e.*, the states that *cannot* be reached.

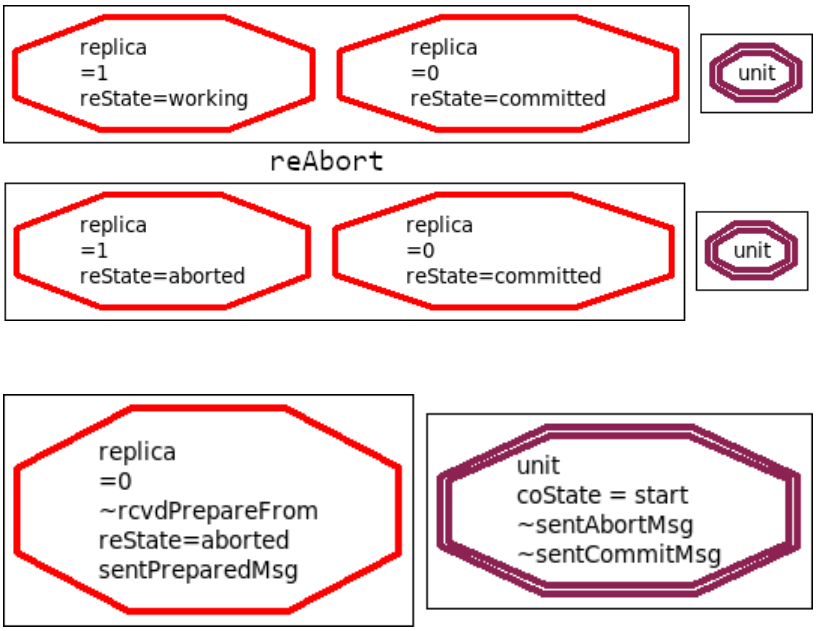

Figure 4.3: The first CTI returned by Ivy for the TPC specification. The top half is the pre-state, whereas the bottom half is the unsafe post-state after taking the `reAbort` transition.

Figure 4.4: A CTI that requires insight to identify the problem. This state is unreachable because the replica cannot abort when it has already prepared the transaction (implied by `sentPreparedMsg`), unless the coordinator aborts—which it has not.

Manual invariant discovery. In line with this intuition, we manually construct the invariant in Figure 4.2, clause by clause, gradually adding more structure to our over-approximation of the set of reachable states, until `ivy_check` reports that the conjunction of all clauses is inductive.

Interestingly, these clauses are logical implications that express the necessary relationships between different portions of the state, as imposed by the protocol's transitions. For instance, the clause on lines 1–2 says that, if the coordinator received a "prepare" message from some replica R, then it must be the case that R has sent such a message.[5] Intuitively, the clause expresses a necessary condition, `sentPreparedMsg(R)`, that must hold before the transition that makes `rcvdPrepareFrom(R)` true can be executed. Indeed, this is exactly what lines 22–23 in Figure 4.1 express. Similarly, the equivalence on line 6 expresses that whenever a decision is made, the coordinator enters the `done` state in the same transition. We obtain all invariant clauses by inspecting the protocol's transitions and making direct inferences about the relationship between state components. Because TPC is such a simple protocol, relatively little creativity is required.

5: In Ivy, identifiers with capital letters are implicitly universally quantified.

```
1  invariant [cannot_commit_while_working]
2      ~(reState(R) = committed & reState(S) = working)
3  invariant [cannot_commit_if_coordinator_aborted]
4      ~(reState(R) = committed & sentAbortMsg)
5  invariant [cannot_commit_if_coordinator_is_not_done]
6      ~(reState(R) = committed & coState = start)
7  invariant [cannot_abort_after_coordinator_done]
8      ~(reState(R) = aborted & sentCommitMsg & coState = done)
9  invariant [cannot_abort_if_prepared_via_sentCommit_unless_coordinator_aborts]
10     ~(reState(R) = aborted & ~sentAbortMsg & sentCommitMsg)
11 invariant [cannot_both_abort_and_commit]
12     ~(sentAbortMsg & sentCommitMsg)
13 invariant [decision_implies_done]
14     ~((sentCommitMsg | sentAbortMsg) & coState = start)
15 invariant [cannot_abort_if_prepared_via_rcvdPr_unless_coordinator_aborts]
16     ~(rcvdPrepareFrom(R) & reState(R) = aborted & ~sentAbortMsg)
17 invariant [cannot_abort_if_prepared_via_sentPr_unless_coordinator_aborts]
18     ~(sentPreparedMsg(R) & reState(R) = aborted & ~sentAbortMsg)
19 invariant [commit_implies_prepared_implies_not_working]
20     ~(reState(R) = working & sentCommitMsg)
21 invariant [cannot_receive_preparedMsg_if_not_sent]
22     ~(rcvdPrepareFrom(R) & ~sentPreparedMsg(R))
23 invariant [cannot_be_working_if_prepared_via_sentPreparedMsg]
24     ~(sentPreparedMsg(R) & reState(R) = working)
```

Figure 4.5: Invariant clauses for TPC discovered using Ivy's counterexample-guided process.

Guided invariant discovery. The alternative option with Ivy is to use its counterexample-guided process to discover invariants. Given a candidate invariant, initially consisting of just the safety property, Ivy finds a CTI and presents it graphically to the user, as in Figure 4.3. This shows a pre-state allowed by the current invariant, that can transition into an unsafe post-state. The problem in this case is that replica 0 committed even though replica 1 has not yet prepared the transaction. In fact, this is not allowed by the protocol. To eliminate this CTI, we strengthen our invariant with the first clause (lines 1–2) in Figure 4.5, that capture the fact that this state is not be reachable in valid TPC executions.

We continue this process, eliminating CTIs one by one until our invariant becomes inductive.[6] However, the process does require insight. Particularly, when presented with a CTI like the one in Figure 4.4, we must identify which parts of the state exhibit the problem and which are incidental to it. If we make

6: Following Ivy's process, we obtain a more complex invariant, containing 12 clauses, compared to the manually obtained invariant, composed of 9 clauses.

a mistake in this, our conjectured clause might be wrong, *i.e.*, not actually invariant. If we accept the conjecture anyway, we will never be able to prove the safety property—we will simply be presented with CTI after CTI until we realise one of our previous conjectures was wrong.

To guard against this possibility, Ivy lets us perform bounded model checking (BMC) to validate our conjectures up to some transition depth that we choose. For simple protocols like two-phase commit, a small bound is sufficient to show that a conjectured clause is not invariant. However, this is not necessarily the case for more complex protocols, where counterexamples might manifest only at a depth that is infeasible for BMC to reach.

Automated invariant guessing. If we are confident that, for our protocol, counterexamples to candidate clauses are shallow (as they are in TPC), then we can use Ivy to automate invariant discovery in a fashion similar to I4 [Ma+19].[7] Concretely, when we are presented with a CTI, we can use Ivy's *Minimize* function, which gives us the minimal generalisation (*i.e.*, subset of the state) of the CTI that is k-invariant, for a depth k that we choose, and add the result as our chosen invariant clause—repeating the process until we obtain an inductive invariant. In effect, *Minimize* tries all possible generalisations[8] of the CTI, checks whether they are invariant up to depth k, and selects the smallest generalisation that is k-invariant. Like I4 and PDR$^\forall$, however, this process can only find invariant clauses that are universally quantified. Even with a large BMC depth k, it would not find all invariants needed to show the safety of, for example, Paxos.

A potentially interesting experiment would be to compare the effectiveness of this Ivy approach, I4, and PDR$^\forall$. The IC3PO paper [GS21a] compares I4 and PDR$^\forall$ and empirically observes similar effectiveness. It would not be surprising if this approach, which existed before both PDR$^\forall$ and I4,[9] performs in line with the later approaches.

7: In I4, the domains (types) were made finite, but with BMC we bound the depth to which the transition relation is unrolled.

8: The actual process is more efficient, and is based on diagrams, the same notion later used by PDR$^\forall$ [Kar+17].

9: We stress that the *Minimize* loop is not automated in the Ivy implementation as such, but all the ingredients are present.

```
mutable function current_term(node): term
mutable function log_term(node, idx): term
mutable function log_value(node, idx): value
mutable function last_index(node): idx
mutable relation request_vote(node, term, idx)
mutable relation append_entries(node, node, term, idx, term, term, value, idx)
```

Figure 4.6: Selected parts of the state for our mypyvy encoding of Multi-Paxos.

4.2 Challenges of EPR Encoding

For our second experiment, we wanted to better understand what challenges arise when trying to encode a complex protocol into extended EPR, and more specifically, we wanted to see what might be required to perform an automated translation from a language like TLA$^+$ into a decidable logic. This is ongoing work, so we only give a high-level description.

Our case study is a variant of Multi-Paxos, described using the style and abstractions of Raft [HM20].

Instead of Ivy, we use another tool, mypyvy,[10] whose specification language is closer to TLA$^+$. Transitions in mypyvy are specified as two-state logical formulas formulas (in first-order logic), as opposed to Ivy's more imperative style.

10: In the course of this project, we identified a number of bugs in mypyvy, which we reported to the maintainers. We also developed a Visual Studio Code syntax highlighting extension for both Ivy and mypyvy.

Encoding the specification. To encode Multi-Paxos into mypyvy, we use the English language description of the protocol as a reference.

First, we identify the different components of the protocol state and what their types are. Each node in Multi-Paxos stores its current term, a log of entries, and various indices, *e.g.*, the highest committed index in the log. Moreover, nodes can send messages to each other, containing, for instance, node IDs, terms, indices, and values. We define these types as first-order logic sorts, as shown in the first 4 lines of Figure 4.7. Then, we use these sorts to specify the various components of the state, partially shown in Figure 4.6.

```
sort node
sort term
sort idx
sort value
sort quorum
```

Figure 4.7: Sorts used in our encoding of Multi-Paxos. `quorum` is an artificial sort, which does not exist in the usual description of Paxos, but which we introduce as part of our encoding of the protocol into first-order logic.

To see what challenges automated translation might run into, we translate the state components into first-order logic as

```
1 immutable relation member(node, quorum)
2 axiom [intersection] forall Q1, Q2. exists N.
3     member(N, Q1) & member(N, Q2)
```

Figure 4.8: Axiomatisation of quorums in FOL, capturing the intersection property.

```
1 transition commit(n: node, maj_idx: idx, q: quorum)
2     modifies commit_index
3     & is_leader(n)
4     & !index_le(maj_idx, commit_index(n))
5     & index_le(maj_idx, last_index(n))
6     & (forall N. member(N, q) ->
7         index_le(maj_idx, match_index(n, N)) | N = n)
8     & set_commit_index(n, maj_idx)
```

Figure 4.9: The `commit` action in Multi-Paxos. A leader (line 3) can advance its commit index to `maj_idx` (line 8), an index greater than its current commit index (line 4), if it knows that a quorum of nodes (line 6) have the same content in their logs up to `maj_idx` (line 7).

directly as possible. Concretely, we encode each component of a node's state as a *function* from the node[11] to the state component's value for that particular node. For example, the current term of a node (line 1) is a function that takes a node as argument and returns that node's term.

11: Or, equivalently, the node's identifier.

To encode logs, which are mappings from indices to pairs consisting of a term and a value, we split the pair into two separate components, `log_term` and `log_value`, and encode them as *functions* that take a node and index as argument and return the respective term or value (lines 2–3 in Figure 4.6).

We encode the set of sent messages ("message soup") as four different *relations*, one for each kind of message, as shown in lines 5–6 of Figure 4.6.[12]

12: In a typical TLA$^+$ spec, the message soup contains all kinds of messages in one set. This could be represented as a single relation whose components are the superset of all message fields (with some default value for unused fields), but splitting the soup is more principled.

Axiomatising orders and quorums. Not everything can be expressed in first-order logic, so to encode the protocol's transitions, we need some insight. Specifically, FOL cannot express all forms of either integer arithmetic or quantification over sets. Nonetheless, Multi-Paxos uses integer addition (to increment indices), comparison (for terms and indices), and set cardinalities (to determine if a quorum was reached).

To express these constructs in FOL, we need to axiomatise them. Concretely, to express comparisons between terms, for instance, we introduce a new relation `index_le`[13] and axiomatise it as a total order that is reflexive, transitive,

13: Corresponding to the $\leq$ operator. We can define other comparison operators based on this, *e.g.*, $x > y \equiv \neg(x \leq y)$.

and antisymmetric. We then use this order to define incrementation.[14] We encode sets of nodes as a new sort, `quorum`, introduce a new relation, `member`, to express set membership, and axiomatise the notion of majority used by Multi-Paxos as the quorum intersection property, as shown in Figure 4.8.

14: A number b is the successor of the number a *iff* $b > a$ and $\forall n, b > n \Rightarrow n \leq a$.

This suffices to encode the protocol's state and all its transitions into extended EPR, as shown in Figure 4.9.

Why is axiomatisation necessary? In a sense, it is not surprising that we can encode the protocol in a decidable logic. After all, the protocol's transitions are all decidable—it must be the case than given a state, you can compute (*i.e.*, decide) the next state, otherwise the protocol is ill-specified. Why did we need to axiomatise things, then?[15] It is a key question.

15: To stress the point: we have not yet encoded any properties of the protocol, or tried to prove anything. We just specified its transitions, yet still needed to introduce an axiomatisation. Why?

The answer is that real systems operate on finite data. Our EPR specification, however, has potentially infinite sorts. We can always compare finite numbers, but a comparison between infinite numbers might never terminate. Similarly, we cannot express set cardinalities because sets might be infinite, so computing their cardinality might never terminate.

When we axiomatise, in effect, we are adding an *oracle* to our specification, a magic function[16] that terminates on any input, even if it is infinite. This solves the decidability problem. However, our oracle is only useful if we can constrain its output—which we do by adding axioms.[17] If we can describe the output of the oracle in extended EPR, then our transitions are decidable even for infinite (unbounded) inputs.

16: Or, in general, relation.

17: An oracle that returns arbitrary output is not helpful.

Quantifier alternations. EPR formulas can only contain constant symbols and relations, and must start with a $\exists^*\forall^*$ quantifier prefix in prenex normal form.

Our specification, however, uses functions and includes $\forall^*\exists^*$ formulas such as the intersection axiom in Figure 4.8. Verification conditions are nonetheless decidable because we exhibit only a restricted form of quantifier alternations.

```
1 invariant [committed_implies_quorum_voted]
2     forall i:idx, et:term, ev:value.
3         (exists n:node. committed(n, i, et, ev)) ->
4         (exists Q:quorum. forall n:node. member(n, Q) -> voted(n, i, et, ev))
```

Figure 4.10: An invariant like this one—which says that if any node committed a value at some index i in some term (line 3), then a majority of nodes have in their logs at index i, the same value and term (line 4)—is necessary to show safety of Multi-Paxos. However, this introduces cycles in our stratification graph.

Sort stratification. We can see that our quantifier alternations are of a restricted form by drawing the *stratification graph* for our specification, as shown in Figure 4.11. The nodes of the graph are the *sorts* in our specification, and we draw a directed $\forall\exists$ edge from sort A to sort B:

- **Quantifier edges**—for every $\exists b : B$ that occurs within the scope of a $\forall a : A$, and
- **Function edges**—if A is one of the components of the domain of a function and B is its codomain.

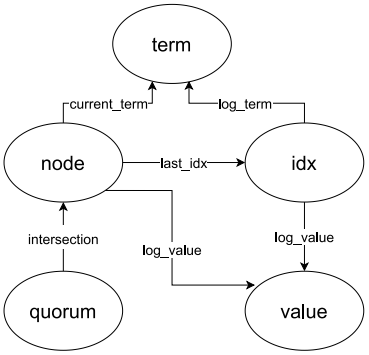

Figure 4.11: Stratification graph for our specification. Multiple functions or properties might correspond to a given edge, but only one is named on the edge.

For example, the intersection axiom in Figure 4.8 produces an edge between `quorum` and `node` because `exists N:node` occurs within the scope of `forall Q1:quorum`. Similarly, `log_value (node, idx):value` produces two edges, one from `node` to `value`, and another from `idx` to `value`.

If we go through our specification and draw the edges accordingly, we obtain the graph in Figure 4.11. Because this graph has *no cycles*, even though our specification includes functions and $\forall^*\exists^*$ quantifier alternations, verification conditions are still decidable—they fall within *extended* EPR.[18]

EPR verification. However, we have not yet stated any properties. We just specified transitions. Sadly, we cannot automatically verify the safety of this specification as it is written.[19] Invariants that are required to show that Multi-Paxos is safe, such as the one in Figure 4.10, introduce cycles in our stratification graph, making verification undecidable.

The solution is to rewrite the spec to use relations instead of functions, such that edges in the graph come only from invariants and axioms, not from the state description [Pad+17]. However, it is unclear whether this can be automated.[20]

18: The extension permits the use of functions and quantifier alternations, as long as the stratification graph has no cycles.

19: Recall that we translated the specification in the most direct way, similar to how an automated tool might do it.

[Pad+17]: Padon et al. (2017), 'Paxos made EPR'

20: We leave the question to future work.

5 Conclusion

In this chapter, we review the material we covered and briefly reflect on potential avenues for future research.

The problem. We began by showing that it is common for protocols, even those designed by experts, to have errors. Numerous examples attest to this. Moreover, implementations are very buggy indeed, with even popular, well-tested systems exhibiting hundreds of critical bugs. The problem is that distributed protocols and systems are difficult to reason about and it is easy to make mistakes.

The state of the art. To guard against human errors, it is helpful to develop computer-encoded *formal specifications*. The key is that, once we have a formal description of a system, it becomes possible to automatically manipulate it in many ways.[1] We can systematically test it with a model checker, or connect it at runtime with an implementation, and check for conformance between the two. But testing is not complete, so if we want higher levels of assurance, we need to employ methods for verification. However, proving systems correct using interactive methods requires both insight and significant effort. Semi-automated methods use solvers to ease the burden of writing proofs, but still require insight to discover inductive invariants. However, tools can assist in discovering invariants. Impressively, automated reasoning tools can both discover invariants and prove safety properties without human interaction, although with limitations.

1: We reiterate our belief that, in the long-term, the benefit of formal methods will also come from tools built *around* formal descriptions (for synthesis and repair, invariant inference, symbolic execution, *etc.*), not strictly from "correctness" guarantees.

The future. We envision that formal techniques will continue to evolve. The level of automation will increase, specifications and their proofs will be more tightly connected with implementations, with the connection automatically maintained as both change, and new tools like for program synthesis and repair will improve developer productivity.

Bibliography

[Abr+17] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. *arXiv:1712.01367 [cs]*, (December 2017). arXiv: 1712.01367. Retrieved 09/06/2021 from (cited on pages 2, 5).

[Abr+19] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2019. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. Technical report 270. Retrieved 09/16/2021 from (cited on page 2).

[Abr+20] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. en. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, (May 2020), 106–118. DOI: 10.1109/SP40000.2020.00044. Retrieved 10/12/2020 from (cited on page 36).

[Alf+20] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. en, 19 (cited on page 10).

[Alv+16] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. en. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, Santa Clara CA USA, (October 2016), 17–28. DOI: 10.1145/2987550.2987555. Retrieved 03/26/2021 from (cited on page 17).

[ARH15] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, (May 2015), 331–346. DOI: 10.1145/2723372.2723711. Retrieved 09/27/2021 from (cited on page 17).

[AT17] Peter Alvaro and Severine Tymon. 2017. Abstracting the Geniuses Away from Failure Testing. en. *Communications of the ACM*, 27 (cited on pages 12, 17).

[AZ15] Brandon Amos and Huanchen Zhang. 2015. 15-812 Term Paper: Specifying and proving cluster membership for the Raft distributed consensus algorithm. en. Technical report, 46 (cited on pages 2, 6).

[Aru+17] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. ISSN: 2158-3927. (June 2017), 49–60. DOI: 10.1109/DSN.2017.35 (cited on page 2).

[AMW16] Noran Azmy, Stephan Merz, and Christoph Weidenbach. 2016. A Rigorous Correctness Proof for Pastry. en. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Volume 9675. Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors. Series Title: Lecture Notes in Computer Science. Springer International Publishing, Cham, 86–101. DOI: 10.1007/978-3-319-33600-8_5. Retrieved 09/07/2021 from (cited on page 2).

[AMW18] Noran Azmy, Stephan Merz, and Christoph Weidenbach. 2018. A machine-checked correctness proof for Pastry. en. *Science of Computer Programming*, 158, (June 2018), 64–80. DOI: 10.1016/j.scico.2017.08.003. Retrieved 09/07/2021 from (cited on pages 2, 24).

[Ban+20] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. 2020. Twins: White-Glove Approach for BFT Testing. en. *arXiv:2004.10617 [cs]*, (April 2020). arXiv: 2004.10617. Retrieved 09/25/2021 from (cited on page 18).

[Bee+96] Ilan Beer, Shoham Ben-david, Cindy Eisner, Avner Landver, and Avner L. 1996. RuleBase: an Industry-Oriented Formal Verification Tool. In *In 33rd Design Automation Conference*, 655–660 (cited on page 20).

[Bor+21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, and Serdar Tasiran. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. en, 15 (cited on pages 12, 13, 35).

[Bra11] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. en. In *Verification, Model Checking, and Abstract Interpretation*. Volume 6538. Ranjit Jhala and David Schmidt, editors. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 70–87. DOI: 10.1007/978-3-642-18275-4_7. Retrieved 06/08/2021 from (cited on pages 27, 28, 32).

[Buc16] Ethan Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis. University of Guelph, Guelph, Ontario, Canada, (June 2016) (cited on page 2).

[But+20] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X. Zhang. 2020. Combining GHOST and Casper. *arXiv:2003.03052 [cs]*, (May 2020). arXiv: 2003.03052. Retrieved 09/16/2021 from (cited on page 2).

[CV17] Christian Cachin and Marko Vukolić. 2017. Blockchain Consensus Protocols in the Wild. *arXiv:1707.01873 [cs]*, (July 2017). arXiv: 1707.01873. Retrieved 09/16/2021 from (cited on page 2).

[CCR04] M. Castro, M. Costa, and A. Rowstron. 2004. Performance and dependability of structured peer-to-peer overlays. en. In *International Conference on Dependable Systems and Networks, 2004*. IEEE, Florence, Italy, 9–18. DOI: 10.1109/DSN.2004.1311872. Retrieved 09/13/2021 from (cited on page 5).

[CL99] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. en. In (February 1999), 14 (cited on page 36).

[CG18] Sagar Chaki and Arie Gurfinkel. 2018. BDD-Based Symbolic Model Checking. en. In *Handbook of Model Checking*. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. Springer International Publishing, Cham, 219–245. DOI: 10.1007/978-3-319-10575-8_8. Retrieved 09/28/2021 from (cited on page 20).

[CLS16] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. en. In *FM 2016: Formal Methods* (Lecture Notes in Computer Science). John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors. Springer International Publishing, Cham, 119–136. DOI: 10.1007/978-3-319-48989-6_8 (cited on page 24).

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. en. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing - PODC '07*. ACM Press, Portland, Oregon, USA, 398–407. DOI: 10.1145/1281100.1281103. Retrieved 01/19/2021 from (cited on page 2).

[Cha+10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. Verifying Safety Properties with the TLA+ Proof System. en. In *Automated Reasoning* (Lecture Notes in Computer Science). Jürgen Giesl and Reiner Hähnle, editors. Springer, Berlin, Heidelberg, 142–148. DOI: 10.1007/978-3-642-14203-1_12 (cited on page 24).

[Cha+08] Kaustuv C. Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2008. A TLA+ Proof System. *arXiv:0811.1914 [cs]*, (November 2008). arXiv: 0811.1914. Retrieved 10/01/2021 from (cited on page 24).

[Che+15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, (October 2015), 18–37. Retrieved 07/27/2021 from (cited on page 8).

[Chu+18] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. en. In *Computer Aided Verification* (Lecture Notes in Computer Science). Hana Chockler and Georg Weissenbacher, editors. Springer International Publishing, Cham, 430–446. DOI: 10.1007/978-3-319-96142-2_26 (cited on page 13).

[Cla+96] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. 1996. Symbolic model checking. en. In *Computer Aided Verification* (Lecture Notes in Computer Science). Rajeev Alur and Thomas A. Henzinger, editors. Springer, Berlin, Heidelberg, 419–422. DOI: 10.1007/3-540-61474-5_93 (cited on page 19).

[Cla+99] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. 1999. State space reduction using partial order techniques. en. *International Journal on Software Tools for Technology Transfer (STTT)*, 2, 3, (November 1999), 279–287. DOI: 10.1007/s100090050035. Retrieved 09/25/2021 from (cited on page 18).

[Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. en. In *Computer Aided Verification* (Lecture Notes in Computer Science). E. Allen Emerson and Aravinda Prasad Sistla, editors. Springer, Berlin, Heidelberg, 154–169. DOI: 10.1007/10722167_15 (cited on page 20).

[Cla+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50, 5, (September 2003), 752–794. DOI: 10.1145/876638.876643. Retrieved 10/04/2021 from (cited on pages 27, 28).

[Cor20] Jay Corbett. 2020. Randomized Testing of Cloud Spanner. (August 2020). Retrieved 09/25/2021 from https://medium.com/@jcorbett_26889/randomized-testing-of-cloud-spanner-5286f1eaba75 (cited on page 16).

[Cou+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. 2012. TLA+ Proofs. en. In *FM 2012: Formal Methods* (Lecture Notes in Computer Science). Dimitra Giannakopoulou and

Dominique Méry, editors. Springer, Berlin, Heidelberg, 147–154. DOI: 10.1007/978-3-642-32759-9_14 (cited on page 24).

[Dai+18] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding Real-World Timeout Problems in Cloud Server Systems. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. (April 2018), 1–11. DOI: 10.1109/IC2E.2018.00022 (cited on page 9).

[DHS20] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. Extreme modelling in practice. en. *Proceedings of the VLDB Endowment*, 13, 9, (May 2020), 1346–1358. DOI: 10.14778/3397230.3397233. Retrieved 03/30/2021 from (cited on page 21).

[Dem18] Murat Demirbas. 2018. TLA+ Specifications of the Consistency Guarantees Provided by Cosmos DB. en-US. (November 2018). Retrieved 09/29/2021 from https://www.microsoft.com/en-us/research/video/tla-specifications-of-the-consistency-guarantees-provided-by-cosmos-db/ (cited on pages 12, 20).

[DF93] Jeremy Dick and Alain Faivre. 1993. Automating the generation and sequencing of test cases from model-based specifications. en. In *FME '93: Industrial-Strength Formal Methods* (Lecture Notes in Computer Science). James C. P. Woodcock and Peter G. Larsen, editors. Springer, Berlin, Heidelberg, 268–284. DOI: 10.1007/BFb0024651 (cited on page 21).

[DPL15] Sisi Duan, Sean Peisert, and Karl N. Levitt. 2015. hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost. *IEEE Transactions on Dependable and Secure Computing*, 12, 1, (January 2015), 58–70. Conference Name: IEEE Transactions on Dependable and Secure Computing. DOI: 10.1109/TDSC.2014.2312331 (cited on page 2).

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. 2011. Efficient Implementation of Property Directed Reachability. en, 10 (cited on pages 27, 28).

[Eir96] A.T. Eiriksson. 1996. Integrating formal verification methods with a conventional project design flow. In *33rd Design Automation Conference Proceedings, 1996*. ISSN: 0738-100X. (June 1996), 666–671. DOI: 10.1109/DAC.1996.545658 (cited on page 20).

[Ene+21] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient replication via timestamp stability. en. In *Proceedings of the Sixteenth European Conference on Computer Systems*. ACM, Online Event United Kingdom, (April 2021), 178–193. DOI: 10.1145/3447786.3456236. Retrieved 09/16/2021 from (cited on page 2).

[Fel+19] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. 2019. Inferring Inductive Invariants from Phase Structures. Technical report. arXiv: 1905.07739. (May 2019). Retrieved 06/15/2021 from (cited on pages 14, 39).

[FGL05] Cormac Flanagan, Patrice Godefroid, and Bell Laboratories. 2005. Dynamic Partial-Order Reduction for Model Checking Software. en. In 12 (cited on page 18).

[Fon+17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM Press, 328–343. Retrieved 03/01/2019 from (cited on pages 11, 12).

[Gan+17] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. en, 19 (cited on page 10).

[Gao+18] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. en. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista FL USA, (October 2018), 539–550. DOI: 10.1145/3236024.3236030. Retrieved 08/31/2021 from (cited on page 8).

[GS19] Aman Goel and Karem Sakallah. 2019. Model Checking of Verilog RTL Using IC3 with Syntax-Guided Abstraction. en. In *NASA Formal Methods* (Lecture Notes in Computer Science). Julia M. Badger and Kristin Yvonne Rozier, editors. Springer International Publishing, Cham, 166–185. DOI: 10.1007/978-3-030-20652-9_11 (cited on page 30).

[GS20] Aman Goel and Karem Sakallah. 2020. AVR: Abstractly Verifying Reachability. en. In *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science). Armin Biere and David Parker, editors. Springer International Publishing, Cham, 413–422. DOI: 10.1007/978-3-030-45190-5_23 (cited on page 30).

[GS21a] Aman Goel and Karem A. Sakallah. 2021. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. *arXiv:2103.14831 [cs]*, 12673, 131–150. arXiv: 2103.14831. DOI: 10.1007/978-3-030-76384-8_9. Retrieved 08/06/2021 from (cited on pages 30, 33, 35, 44).

[GS21b] Aman Goel and Karem A. Sakallah. 2021. Towards an Automatic Proof of Lamport's Paxos. *arXiv:2108.08796 [cs]*, (August 2021). arXiv: 2108.08796. Retrieved 08/25/2021 from (cited on page 33).

[GG75] John B. Goodenough and Susan L. Gerhart. 1975. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*. Association for Computing Machinery, New York, NY, USA, (April 1975), 493–510. DOI: 10.1145/800027.808473. Retrieved 09/23/2021 from (cited on page 15).

[Gra+11] A. Gravell, Y. Howard, J. C. Augusto, C. Ferreira, and S. Gruner. 2011. Concurrent Development of Model and Implementation. *arXiv:1111.2826 [cs]*, (November 2011). arXiv: 1111.2826. Retrieved 03/31/2021 from (cited on page 21).

[Gun+14] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (SOCC '14). Association for Computing Machinery, New York, NY, USA, (November 2014), 1–14. DOI: 10.1145/2670979.2670986. Retrieved 09/29/2020 from (cited on page 7).

[Gur15] Arie Gurfinkel. IC3, PDR, and Friends. Fifth Summer School on Formal Techniques, (2015). Retrieved 08/06/2021 from (cited on page 27).

[Hae+05] Andreas Haeberlen, Jeff Hoye, Alan Mislove, and Peter Druschel. 2005. Consistent Key Mapping in Structured Overlays. Technical report TR05-456. Rice University, Houston, Texas, (August 2005), 6 (cited on page 5).

[Hal20] Jordan Halterman. 2020. Real-Time Conformance Monitoring with TLC: A Post-Mortem. Reddit Post. (March 2020). Retrieved 09/29/2021 from www.reddit.com/r/tlaplus/comments/fg4741/realtime_conformance_monitoring_with_tlc_a/ (cited on pages 12, 21).

[Han+21] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. en. In 17 (cited on pages 32, 33).

[Has15] Mazdak Hashemi. 2015. How we break things at Twitter: failure testing. (December 2015). Retrieved 09/27/2021 from https://blog.twitter.com/engineering/en_us/a/2015/how-we-break-things-at-twitter-failure-testing (cited on page 16).

[Haw+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*. ACM Press. Retrieved 04/08/2019 from (cited on pages 11, 12, 24, 25, 34).

[Hoc14] Ezra Hoch. 2014. Configuration changes. (February 2014). Retrieved 09/09/2021 from https://groups.google.com/g/raft-dev/c/xux5HRxH3Ic/m/mz_PDK-qMJgJ (cited on pages 2, 6).

[Hol97] Gerard J. Holzmann. 1997. The model checker SPIN. en. *IEEE Transactions on Software Engineering*, 23, 5, (May 1997), 279–295. DOI: 10.1109/32.588521. Retrieved 09/28/2021 from (cited on page 19).

[Hol18] Gerard J. Holzmann. 2018. Explicit-State Model Checking. en. In *Handbook of Model Checking*. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. Springer International Publishing, Cham, 153–171. DOI: 10.1007/978-3-319-10575-8_5. Retrieved 09/28/2021 from (cited on page 19).

[HA20] Heidi Howard and Ittai Abraham. 2020. Raft does not Guarantee Liveness in the face of Network Faults. en. (December 2020). Retrieved 05/07/2021 from https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/ (cited on pages 2, 6).

[HM20] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: Have we reached consensus on distributed consensus? en. *arXiv:2004.05074 [cs]*, (April 2020). arXiv: 2004.05074. Retrieved 04/28/2020 from (cited on pages 5, 14, 39, 45).

[How+11] Yvonne Howard, Stefan Gruner, Andrew M Gravell, Carla Ferreira, and Juan Carlos Augusto. 2011. Model-Based Trace-Checking. en. Technical report. (November 2011), 14 (cited on page 21).

[Jac02] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11, 2, (April 2002), 256–290. DOI: 10.1145/505145.505149. Retrieved 10/01/2021 from (cited on page 26).

[JSS00] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. 2000. Alcoa: the Alloy constraint analyzer. Technical report. Journal Abbreviation: Proceedings - International Conference on Software Engineering Pages: 733 Publication Title: Proceedings - International Conference on Software Engineering. (February 2000). DOI: 10.1109/ICSE.2000.870482 (cited on page 20).

[Jan+97] Jae-young Jang, Shaz Qadeer, Matt Kaufmann, and Carl Pixley. 1997. Formal verification of FIRE: A case study. In *In DAC*, 173–177 (cited on page 20).

[JB83] C. Jard and G. v. Bochmann. 1983. An approach to testing specifications. *ACM SIGPLAN Notices*, 18, 8, (March 1983), 53–59. DOI: 10.1145/1006142.1006159. Retrieved 03/31/2021 from (cited on page 21).

[JHM21] Chris Jensen, Heidi Howard, and Richard Mortier. 2021. Examining Raft's behaviour during partial network failures. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems* (HAOC '21). Association for Computing Machinery, New York, NY, USA, (April 2021), 11–17. DOI: 10.1145/3447851.3458739. Retrieved 09/08/2021 from (cited on pages 2, 6).

[Kar+17] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2017. Property-Directed Inference of Universal Invariants or Proving Their Absence. en. *Journal of the ACM*, 30 (cited on pages 28, 29, 44).

[Kil+07] Charles Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007. Mace: Language Support for Building Distributed Systems. en, 10 (cited on page 18).

[Kin13] Kyle Kingsbury. 2013. Jepsen: A framework for distributed systems verification, with fault injection. (2013). Retrieved 04/10/2019 from https://github.com/jepsen-io/jepsen (cited on page 16).

[Kin20] Kyle Kingsbury. 2020. Jepsen Analyses. (2020). Retrieved 09/25/2021 from https://jepsen.io/analyses (cited on page 16).

[Knu74] Donald E. Knuth. 1974. Computer programming as an art. *Communications of the ACM*, 17, 12, (December 1974), 667–673. DOI: 10.1145/361604.361612. Retrieved 09/03/2021 from (cited on page 1).

[Koe+20] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. 2020. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020). Association for Computing Machinery, New York, NY, USA, (June 2020), 703–717. DOI: 10.1145/3385412.3386018. Retrieved 06/08/2021 from (cited on page 33).

[Koń+11] Jan Kończak, Nuno Filipe de Sousa Santos, Tomasz Żurkowski, Paweł T. Wojciechowski, and André Schiper. 2011. JPaxos: State machine replication based on the Paxos protocol. Technical report EPFL-REPORT-167765 (cited on page 2).

[KKT19] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ model checking made symbolic. en. *Proceedings of the ACM on Programming Languages*, 3, OOPSLA, (October 2019), 1–30. 00001. DOI: 10.1145/3360549. Retrieved 11/29/2019 from (cited on page 20).

[Kot+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative byzantine fault tolerance. In *In Symposium on Operating Systems Principles (SOSP* (cited on page 2).

[Kot+09] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine fault tolerance. en. *ACM Transactions on Computer Systems*, 27, 4, 1–39. DOI: 10.1145/1658357.1658358. Retrieved 09/07/2021 from (cited on page 2).

[Kro+20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. en. In *Programming Languages and Systems*. Volume 12075. Peter Müller, editor. Series Title: Lecture Notes in Computer Science. Springer International Publishing, Cham, 336–365. DOI: 10.1007/978-3-030-44914-8_13. Retrieved 03/08/2021 from (cited on pages 23, 36).

[Kur18] Robert P. Kurshan. 2018. Transfer of Model Checking to Industrial Practice. en. In *Handbook of Model Checking*. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. Springer International Publishing, Cham, 763–793. DOI: 10.1007/978-3-319-10575-8_23. Retrieved 09/29/2021 from (cited on page 20).

[Lam98a] Leslie Lamport. 1998. Composition: A Way to Make Proofs Harder. en. In *Compositionality: The Significant Difference*. Volume 1536. Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 402–423. DOI: 10.1007/3-540-49213-5_15. Retrieved 12/31/2019 from (cited on page 23).

[Lam98b] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16, 2, 133–169. 02327. DOI: 10.1145/279227.279229 (cited on page 32).

[Lam05] Leslie Lamport. 2005. Generalized Consensus and Paxos. Technical report MSR-TR-2005-33. Microsoft Research, (March 2005) (cited on page 2).

[Lam11] Leslie Lamport. 2011. Byzantizing Paxos by Refinement. en. In *Distributed Computing*. Volume 6950. David Peleg, editor. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 211–224. DOI: 10.1007/978-3-642-24100-0_22. Retrieved 11/06/2020 from (cited on page 24).

[Lam18] Leslie Lamport. 2018. Industrial Use of TLA+. (December 2018). Retrieved 09/29/2021 from http://lamport.azurewebsites.net/tla/industrial-use.html (cited on pages 12, 20).

[Lee+14] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. en, 17. 00074 (cited on page 18).

[Lee+16] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. en. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Atlanta Georgia USA, (March 2016), 517–530. DOI: 10.1145/2872362.2872374. Retrieved 09/06/2021 from (cited on page 9).

[Lei10] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. en. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Volume 6355. Edmund M. Clarke and Andrei Voronkov, editors. Springer, 348–370. Retrieved 04/16/2019 from (cited on pages 25, 38).

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '16). ACM, 357–370. Retrieved 04/10/2019 from (cited on pages 11, 34).

[LS20] Tom Lianza and Chris Snook. 2020. A Byzantine failure in the real world. en. (November 2020). Retrieved 09/16/2021 from https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/ (cited on page 6).

[LBK02] David Liben-Nowell, Hari Balakrishnan, and David Karger. 2002. Analysis of the evolution of peer-to-peer systems. In *In ACM Conf. on Principles of Distributed Computing (PODC*, 233–242 (cited on pages 2, 3).

[LC12] Barbara Liskov and James Cowling. 2012. Viewstamped Replication Revisited. en. Technical report MIT-CSAIL-TR-2012-021. (July 2012), 16 (cited on page 2).

[Lu+19] Jie Lu, Liu Chen, Lian Li, and Xiaobing Feng. 2019. Understanding Node Change Bugs for Distributed Systems. en. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Hangzhou, China, (February 2019), 399–410. DOI: 10.1109/SANER.2019.8668027. Retrieved 08/31/2021 from (cited on page 8).

[Lu+08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS XIII). Association for Computing Machinery, New York, NY, USA, (March 2008), 329–339. DOI: 10.1145/1346281.1346323. Retrieved 09/19/2021 from (cited on page 9).

[Lu13] Tianxiang Lu. 2013. *Formal verification of the Pastry protocol*. PhD thesis. Saarland University (cited on page 5).

[Luk+19] Jeffrey F. Lukman, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, Haryadi S. Gunawi, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, and Feng Ye. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. en. In *Proceedings of the Fourteenth EuroSys Conference 2019 CD-ROM on ZZZ - EuroSys '19*. 00001. ACM Press, Dresden, Germany, 1–16. DOI: 10/gf9vng. Retrieved 10/14/2019 from (cited on page 18).

[Ma+19] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (SOSP '19). Association for Computing Machinery, New York, NY, USA, (October 2019), 370–384. DOI: 10.1145/3341301.3359651. Retrieved 03/08/2021 from (cited on pages 30, 35, 44).

[MN17] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? en. *Proceedings of the ACM on Programming Languages*, 2, POPL, (December 2017), 1–24. 00005. DOI: 10/gf9vnf. Retrieved 10/14/2019 from (cited on page 17).

[MA05] J.-P. Martin and L. Alvisi. 2005. Fast Byzantine Consensus. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*. ISSN: 2158-3927. (June 2005), 402–411. DOI: 10.1109/DSN.2005.48 (cited on page 2).

[MA06] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. English. *IEEE Transactions on Dependable and Secure Computing*, 3, 3, (September 2006), 202–215. Num Pages: 202-215 Place: Washington, United States Publisher: IEEE Computer Society. DOI: http://dx.doi.org.libproxy1.nus.edu.sg/10.1109/TDSC.2006.35 (cited on page 2).

[McM03] K. L. McMillan. 2003. Interpolation and SAT-Based Model Checking. en. In *Computer Aided Verification*. Volume 2725. Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Warren A. Hunt, and Fabio Somenzi, editors. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–13. DOI: 10.1007/978-3-540-45069-6_1. Retrieved 06/15/2021 from (cited on page 20).

[McM02] Ken L. McMillan. 2002. Applying SAT Methods in Unbounded Symbolic Model Checking. en. In *Computer Aided Verification*. Volume 2404. Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Ed Brinksma, and Kim Guldstrand Larsen, editors. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 250–264. DOI: 10.1007/3-540-45657-0_19. Retrieved 06/14/2021 from (cited on page 19).

[MP20] Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. en. In *Computer Aided Verification*. Volume 12225. Shuvendu K. Lahiri and Chao Wang, editors. Series Title: Lecture Notes in Computer Science. Springer International Publishing, Cham, 190–202. DOI: 10.1007/978-3-030-53291-8_12. Retrieved 03/08/2021 from (cited on pages 26, 34).

[Mic+17] Ellis Michael, Dan R K Ports, Naveen Kr Sharma, and Adriana Szekeres. 2017. Recovering Shared Objects Without Stable Storage. en, (August 2017), 27 (cited on page 2).

[Mil21] Alex Miller. 2021. BUGGIFY - Testing Distributed Systems with Deterministic Simulation. (2021). Retrieved 09/25/2021 from https://transactional.blog/simulation/buggify.html (cited on page 17).

[MDC06] Alice Miller, Alastair Donaldson, and Muffy Calder. 2006. Symmetry in temporal logic model checking. en. *ACM Computing Surveys*, 38, 3, (September 2006), 8. DOI: 10.1145/1132960.1132962. Retrieved 09/25/2021 from (cited on page 18).

[MC19] Atsuki Momose and Jason Paul Cruz. 2019. Force-Locking Attack on Sync Hotstuff. Technical report 1484. Retrieved 09/16/2021 from (cited on page 2).

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (SOSP '13). Association for Computing Machinery, New York, NY, USA, (November 2013), 358–372. DOI: 10.1145/2517349.2517350 (cited on page 2).

[Nak15] Heather Nakama. 2015. Inside Azure Search: Chaos Engineering. en. (July 2015). Retrieved 09/24/2021 from https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/ (cited on page 16).

[NAE18] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications. In *Proceedings of the 2018 International Conference on Management of Data* (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, (May 2018), 1221–1236. DOI: 10.1145/3183713.3196928. Retrieved 09/07/2021 from (cited on page 2).

[Net11] Netflix Technology Blog. 2011. The Netflix Simian Army. en. (July 2011). Retrieved 09/24/2021 from https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (cited on page 16).

[NTT21] Joachim Neu, Ertem Nusret Tas, and David Tse. 2021. Ebb-and-Flow Protocols: A Resolution of the Availability-Finality Dilemma. *arXiv:2009.04987 [cs]*, (February 2021). arXiv: 2009.04987. Retrieved 09/16/2021 from (cited on page 2).

[New+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. en. *Communications of the ACM*, 58, 4, (March 2015), 66–73 (cited on pages 1, 4, 5, 12, 16, 19–21, 36).

[OHe18] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. en. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Oxford United Kingdom, (July 2018), 13–25. DOI: 10.1145/3209108.3209109. Retrieved 10/08/2021 from (cited on page 13).

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. en. In *Computer Science Logic*. Volume 2142. Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Laurent Fribourg, editors. 00000. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–19. DOI: 10.1007/3-540-44802-0_1. Retrieved 06/02/2019 from (cited on page 23).

[OHe07] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. en. *Theoretical Computer Science*, 375, 1-3, (May 2007), 271–307. 00532. DOI: 10/crkgwk. Retrieved 06/02/2019 from (cited on page 23).

[Ong14] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. PhD thesis. Stanford University, (August 2014) (cited on pages 2, 6).

[Ong15] Diego Ongaro. 2015. Bug in single-server membership changes. (July 2015). Retrieved 09/01/2021 from https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J (cited on pages 2, 6).

[OO14] Diego Ongaro and John K. Ousterhout. 2014. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*. 00416, 305–319 (cited on pages 2, 5, 32).

[Pad+16a] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. 2016. Decidability of inferring inductive invariants. *ACM SIGPLAN Notices*, 51, 1, (January 2016), 217–231. DOI: 10.1145/2914770.2837640. Retrieved 06/08/2021 from (cited on page 29).

[Pad+17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. en. In *Proceedings of the ACM on Programming Languages*. Volume 1. 00023. (October 2017), 1–31. DOI: 10/gfzhts. Retrieved 04/16/2019 from (cited on pages 26, 35, 48).

[Pad+16b] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*. ACM Press, 614–630. Retrieved 04/16/2019 from (cited on pages 14, 25, 26, 29, 39).

[Pîr19] George Pîrlea. 2019. *Toychain: Formally-Verified Blockchain Consensus*. PhD thesis. University College London, (April 2019). Retrieved 05/06/2019 from (cited on page 37). 00000.

[PKS21] George Pîrlea, Amrit Kumar, and Ilya Sergey. 2021. Practical smart contract sharding with ownership and commutativity analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2021). Association for Computing Machinery, New York, NY, USA, (June 2021), 1327–1341. DOI: 10.1145/3453483.3454112. Retrieved 07/29/2021 from (cited on page 38).

[PS18] George Pîrlea and Ilya Sergey. 2018. Mechanising Blockchain Consensus. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. DOI: 10.1145/3167086 (cited on page 37).

[PdMB10] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. 2010. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. en. *Journal of Automated Reasoning*, 44, 4, (April 2010), 401–424. DOI: 10.1007/s10817-009-9161-6. Retrieved 06/07/2021 from (cited on page 26).

[PS99] Carl Pixley and Vigyan Singhal. 1999. Model checking: a hardware design perspective. en. *International Journal on Software Tools for Technology Transfer (STTT)*, 2, 3, (November 1999), 288–306. DOI: 10.1007/s100090050036. Retrieved 09/29/2021 from (cited on page 20).

[Rei+20] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. 2020. Towards making formal methods normal: meeting developers where they are. *arXiv:2010.16345 [cs]*, (October 2020). arXiv: 2010.16345. Retrieved 11/02/2020 from (cited on pages 1, 12, 35).

[Rei16] Emily Reinhold. 2016. Rewriting Uber Engineering: The Opportunities Microservices Provide. en-US. (April 2016). Retrieved 09/27/2021 from https://eng.uber.com/building-tincup-microservice-implementation/ (cited on page 16).

[Rob+12] Jesse Robins, Kripa Krishnan, John Allspaw, and Tom Limoncelli. 2012. Resilience Engineering: Learning to Embrace Failure. *ACM Queue*, 10, 9, (September 2012). Retrieved 09/24/2021 from (cited on page 16).

[RD01] Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. en. In *Middleware 2001* (Lecture Notes in Computer Science). Rachid Guerraoui, editor. Springer, Berlin, Heidelberg, 329–350. DOI: 10.1007/3-540-45518-3_18 (cited on pages 2, 4).

[SZT21] William Schultz, Siyuan Zhou, and Stavros Tripakis. 2021. Design and Verification of a Logless Dynamic Reconfiguration Protocol in MongoDB Replication. en. *arXiv:2102.11960 [cs]*, (February 2021). arXiv: 2102.11960. Retrieved 03/26/2021 from (cited on pages 12, 20).

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (January 2018). Retrieved 02/03/2018 from (cited on pages 23, 36, 37).

[She16] Arjun Shenoy. 2016. A Deep Dive into Simoorg, our Open Source Failure Induction Framework. en. (March 2016). Retrieved 09/27/2021 from https://engineering.linkedin.com/blog/2016/03/deep-dive-Simoorg-open-source-failure-induction-framework (cited on page 16).

[SKD19] Nibesh Shrestha, Mohan Kumar, and SiSi Duan. 2019. Revisiting hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost. *arXiv:1902.08505 [cs]*, (April 2019). arXiv: 1902.08505. Retrieved 09/16/2021 from (cited on page 2).

[Sig+16] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. en, 16 (cited on page 8).

[SBG10] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. en, 9 (cited on page 18).

[Spr+20] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David Basin. 2020. Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification. en. *arXiv:2010.04749 [cs]*, (October 2020). arXiv: 2010.04749. DOI: 10.1145/3428220. Retrieved 10/20/2020 from (cited on pages 23, 36).

[Sto+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. en, (August 2001), 12 (cited on pages 2, 3).

[Sut20] Pierre Sutra. 2020. On the correctness of Egalitarian Paxos. en. *Information Processing Letters*, 156, (April 2020), 105901. DOI: 10.1016/j.ipl.2019.105901 (cited on page 2).

[SS10] Pierre Sutra and Marc Shapiro. 2010. Fast Genuine Generalized Consensus. en. Technical report. (corrected August 2010). Section 6.3. (February 2010), 62 (cited on page 2).

[Swa+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. en. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, St. Petersburg FL USA, (January 2016), 256–270. DOI: 10.1145/2837614.2837655. Retrieved 10/09/2021 from (cited on page 38).

[Tau+18] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. en. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Philadelphia PA USA, (June 2018), 662–677. DOI: 10.1145/3192366.3192414. Retrieved 06/03/2021 from (cited on page 36).

[TF09] Jeff Terrace and Michael J. Freedman. 2009. Object Storage on {CRAQ}: High-Throughput Chain Replication for Read-Mostly Workloads. en. In Retrieved 09/09/2021 from (cited on page 2).

[Wat+21] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages*, 5, ICFP, (August 2021), 84:1–84:29. DOI: 10.1145/3473589. Retrieved 08/23/2021 from (cited on pages 37, 38).

[Whi20] Michael Whittaker. CRAQ Bug. original-date: 2020-06-13T18:44:33Z, (June 2020). Retrieved 09/09/2021 from (cited on page 2).

[Whi21] Michael Whittaker. 2021. EPaxos Dependency Set Compaction Bug. original-date: 2018-11-03T04:31:20Z. (September 2021). Retrieved 09/16/2021 from https://github.com/mwhittaker/bipartisan_paxos/blob/master/epaxos_bugs/epaxos_dependency_bug.pdf (cited on page 2).

[Whi+21] Michael Whittaker, Joseph M Hellerstein, Neil Giridharan, Adriana Szekeres, Heidi Howard, and Faisal Nawab. 2021. Matchmaker Paxos: A Reconfigurable Consensus Protocol. en. *Journal of Systems Research*, 22 (cited on page 2).

[Wil+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '15). ACM, 357–368. Retrieved 04/09/2019 from (cited on pages 11, 12, 22, 34, 37).

[Woo+16] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. 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* (CPP 2016). Association for Computing Machinery, New York, NY, USA, (January 2016), 154–165. DOI: 10.1145/2854065.2854081. Retrieved 03/07/2021 from (cited on pages 12, 22).

[Yan+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. en, 16. 00216 (cited on page 18).

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. en, 12. 00559 (cited on page 11).

[Yao+21] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. en. In 17 (cited on pages 30, 32, 35).

[Yin+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. en. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, Toronto ON Canada, (July 2019), 347–356. DOI: 10.1145/3293611.3331591. Retrieved 10/30/2020 from (cited on page 36).

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. en. In *Correct Hardware Design and Verification Methods*. Volume 1703. Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Laurence Pierre, and Thomas Kropf, editors. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 54–66. DOI: 10.1007/3-540-48153-2_6. Retrieved 03/08/2021 from (cited on page 19).

[Yua+14] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. en. In 249–265. Retrieved 09/17/2021 from (cited on pages 7, 8).

[Zav12] Pamela Zave. 2012. Using lightweight modeling to understand chord. en. *ACM SIGCOMM Computer Communication Review*, 42, 2, (March 2012), 49–57. DOI: 10.1145/2185376.2185383. Retrieved 09/06/2021 from (cited on pages 2, 3).

[Zav17] Pamela Zave. 2017. Reasoning About Identifier Spaces: How to Make Chord Correct. *IEEE Transactions on Software Engineering*, 43, 12, (December 2017), 1144–1156. Conference Name: IEEE Transactions on Software Engineering. DOI: 10.1109/TSE.2017.2655056 (cited on pages 2–4).

[Zho+21] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. en. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, (June 2021), 2653–2666. DOI: 10.1145/3448016.3457559. Retrieved 09/27/2021 from (cited on page 16).