THE UNIVERSITY OF CHICAGO

# ALGORITHMIC, HEURISTIC, AND SYSTEMATIC APPROACHES FOR SOFTWARE MODEL CHECKING OF DISTRIBUTED SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
JEFFREY F. LUKMAN

CHICAGO, ILLINOIS
AUGUST 2020

*To my two ladies: my wife and my daughter*

*"There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relavitely small set of common tasks."*

Leslie Lamport

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

Our success in life is deeply affected by our hard work and by the people that surrounds us through the journey. The quest for my Ph.D. is one of the proofs for this truth. In this section, I want to express my gratitude to special people who usher me to my finish line.

To Haryadi Gunawi. I still have a clear memory of our first Skype call where you convinced me to do research with you and pursue my Ph.D. even though, at that moment, it meant that I need to sacrifice half of my salary with no guarantee that I might end up qualified to be a Ph.D. student. Your message was simple, "While you are young, the best investment that you can do is to invest in yourself. And Ph.D. is one of the best experiences that you can have in life."

Throughout my Ph.D., you pointed out my flaws, you shared your wisdom and experience so that I can learn from it, you coached me to be pragmatic instead of perfectionist, you facilitated my needs, you funded me to many conferences (you even allowed my wife to accompany my trips!), you mentored me to be a better public speaker, a better writer, a better negotiator, a better leader, and a more productive person (automate and delegate!). You taught me to expect the best from myself and plan for the worst for things out of my control. It is my honor to call you my advisor. From the bottom of my heart, I thank you, sir.

To Shan Lu and Tanakorn Leesatapornwongsa. Both of you are more than my Ph.D. committee. You have also been my constant research collaborators in most of my publications. Shan, you specifically taught me how to pay attention to details as the details are what matter. My understanding of those details is what made me an expert. Korn, you taught me how to base all of my decisions on data and facts. My collaboration with you allowed me to start and finish my Ph.D.!

To all the members of the UCARE group and Shan's group: Mingzhe Hao, Huaicheng Li, Cesar Stuardo, Riza Suminto, Huan Ke, Daniar Kurniawan, Meng Wang, Haopeng Liu, and Guangpu Li. Without our friendship, I would not survive the toughest time when we were pursuing submission deadlines. I would not enjoy my time of being a graduate student. I'm proud to call each one of you my friend!

To Papa Hariono and Mama Naniek. Thank you for preparing me and Hueynie for our trip, when we were about to start our adventure in Chicago. Thank you for coming to Chicago when we need you the most. And thank you for allowing Hueynie to accompany me to go through my Ph.D. journey (and the rest of my life).

To Papa Ferry and Mama Ira. Thank you for never giving up even through the hardest time of our family life. Thank you for pointing to me that ”Wisdom is more precious than gold.”. Thank you for always reminding me to not overthink stuff. I am who I am today because of you both.

To Hueynie and Hazel. You are my treasures on earth. I dedicate this Ph.D. for you.

# ABSTRACT

Today, distributed software infrastructures have become a dominant backbone for cloud computing and modern applications. Large-scale distributed systems such as scalable frameworks, storage systems, synchronization, and cluster management services have emerged as the data center operating system. Unfortunately, the reliability of distributed systems is threatened by non-deterministic concurrency bugs as it executes many complicated distributed protocols on thousands of machines with no common clocks and it must face a variety of random hardware failures. Facing the challenge of concurrency bugs, this dissertation proposes efficient approaches to empower software model checking to quickly unearth concurrency bugs based on the study of the comprehensive characteristics of real-world concurrency bugs in distributed systems.

This dissertation makes three main contributions. First, we present TaxDC, the largest and most comprehensive taxonomy of distributed concurrency (DC) bugs. We study the characteristics of 104 real-world distributed concurrency (DC) bugs along several axes of analysis, such as the triggering timing condition and input preconditions, error and failure symptoms, and fix strategies from four widely-deployed cloud-scale distributed systems: Cassandra, Hadoop MapReduce, HBase, and ZooKeeper. The study of DC bugs characteristics provides many motivation and guidelines for DC bugs detection, testing, and tools design.

Second, we present FLYMC, a fast, scalable, and systematic software model checker for testing distributed systems implementations. To overcome the path/state-space explosion problem in testing complex interleavings of messages and faults, FLYMC introduces three powerful algorithms: state symmetry, event independence, and parallel flips. As a result, collectively, these algorithms make our approach on average 16× (up to 78×) faster than other state-of-the-art solutions. FLYMC is integrated with eight distributed systems, successfully reproduced twelve known DC bugs, and found ten new DC bugs which all have been confirmed by developers.

Third, we present HMC, a heuristic software model checker to unearth DC bugs faster. By exploiting the properties of distributed systems and software model checking, HMC introduces

four novel algorithms: blocking state-event, last state-event, miss-prediction step, and prioritized node crash, to prioritize more-likely test scenarios that will reach new corner cases. With HMC, we reproduced five DC bugs in Cassandra on average 6× (up to 15×) faster than FLYMC which we consider as the state-of-the-art of systematic software model checker.

# CHAPTER 1

# INTRODUCTION

*"Do we have to rethink this entire [HBase] root and meta 'huh hah'? There isn't a week going by without some new bugs about* ***races*** *between splitting and assignment [distributed protocols]." — A comment in HB-4397*

Today, beyond single-machine software, distributed software infrastructures have become a dominant backbone for cloud computing and modern applications. Some driving forces for these developments:

- Internet companies (*e.g.*, Google, Amazon, Facebook, Twitter) are handling enormous amounts of data and traffic, forcing them to create new tools to allow them to handle such a scale.
- Businesses want to move rapidly, check assumptions cheaply, and respond quickly to new understandings of the markets by executing fast queries on a huge amount of data.
- CPU clock speeds are hardly improving, but networks are getting faster. Hence, more workloads are executed in a parallel fashion.
- Infrastructure as a Service (IaaS), such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform enables a small group of people to build applications with systems that are distributed across many machines and spread out geographically in many different regions.
- End users and businesses are expecting their cloud services to be highly available. Therefore, system outages become even more unacceptable.

This trend motivates the development of many large-scale distributed systems, such as scalable frameworks [3, 33], storage systems [2, 5, 26, 34], synchronization [6, 23], and cluster management

services [4, 57]. The synergy of all of these distributed systems has emerged as the datacenter operating system. As a result, increasing numbers of developers write complex large-scale distributed systems and billions of end-users rely on the reliability of these systems.

## 1.1 Motivation

### *1.1.1 Concurrency Bugs in Distributed Systems*

Unfortunately, the reliability of distributed systems is severely threatened by non-deterministic concurrency bugs which we refer to as *distributed concurrency (DC) bugs*. Distributed systems execute many complicated distributed protocols (*e.g.*, serving millions of users' requests, operating background tasks such as data replication, data compaction, garbage collection) on hundreds/thousands of machines with no common clocks and even more, it must face a variety of random failures [37, 53]. Furthermore, most developers think and program sequentially, thus, they can easily make mistakes when thinking about multiple components or events of distributed systems executed concurrently.

This combination makes distributed systems prone to DC bugs caused by non-deterministic timing of distributed events, such as message arrivals, node crashes, reboots, local computations, and timeouts. And as hinted by the developers' comments quoted throughout this dissertation, DC bugs manifest non-deterministically. Hence, they are extremely difficult to detect, diagnose and mitigate. Many of them can take weeks/months to resolve.

A large body of *local concurrency (LC) bugs* work has focused on concurrency issues in single-machine multi-threaded software, such as many comprehensive LC bug studies [41, 78], detection tools [77, 79, 91], testing frameworks [64, 93], and auto-fixing approaches [60, 86]. On the other hand, DC bugs have not received the same amount of attention within the research community.

### *1.1.2 Impacts of Concurrency Bugs in the Real World*

As DC bugs are timing-related bugs, they manifest non-deterministically and if they do, DC bugs can cause fatal implications such as operation failures, deadlocks, downtimes, data loss, and inconsistencies. That is, DC bugs can make software as a single point of failure.

In April 2011, Amazon Elastic Compute Cloud (EC2) and Amazon Relational Database Service (RDS) experienced eleven hours of service outage in its US East Region service [15]. In summary, a subset of the Amazon Elastic Block Store (EBS) volumes in a single Availability Zone within the US East Region became unable to service read and write operations. Hence, EC2 and RDS who access data in the EBS fails to execute client requests. Some companies that are affected by these major outages are Reedit, Quora, FourSquare, Hootsuite, parts of the New York Times, ProPublica, and 70 other sites [16]. And after the AWS developers contained the issue, these websites still need to wait over 36 hours to get back to normal service. This incident has violated AWS Service Level Agreement (SLA) at that moment which guaranteed 99.95% availability – equal to a maximum of 4.4 hours of downtimes in a year. It caused a major financial loss and harmed Amazon's reputation as a trusted cloud service provider.

Further investigation found that the incident was triggered by a network misconfiguration which caused a re-mirroring storm in the EBS cluster. Eventually, the re-mirroring storm exposed DC bug in the EBS cluster as a huge amount of network traffic was directed to these EBS nodes. In detail, as multiple messages are received, the EBS node could not handle the race condition of those messages which previously very rare to happen, hence the EBS node crashed. During the worst condition, up to 13% of the EBS cluster died and the rest of the cluster must handle the overwhelmingly high traffic that it is out of what it was designed to handle.

Lastly, after AWS has fully recovered from the outage incident, it still took another couple of weeks for AWS to resolve the DC bug in the EBS node. This incident shows how severe the impact that DC bugs can cause, how important it is to detect DC bugs in distributed systems prior to shipping it into the production site, and how hard it is to test and fix them.

## 1.2 Building More Reliable Distributed Systems

As we have seen the sad reality that our distributed systems are prone to DC bugs, in this section, we propose our approaches to improve distributed systems reliability. First, we introduce TaxDC [68], the largest and most comprehensive taxonomy of distributed concurrency bugs. Next, we present FLYMC [80], a fast, scalable, and systematic software model checker for testing distributed systems implementations. Lastly, we present HMC, a heuristic software model checker to unearth DC bugs faster.

### *1.2.1 TaxDC: A Taxonomy of Distributed Concurrency Bugs*

A lot of studies have closely examined LC bugs in various software systems [28, 41, 76, 78, 85, 89, 98] which have motivated and guided many aspects of building reliable multithreaded applications. Unfortunately, DC bugs have not received the same amount of attention within the research community yet.

There are a few bug studies on large-scale distributed systems that have been conducted [53, 69], but they did not specifically dissect DC bugs. There is also another analysis on non-determinism bugs in MapReduce, but that analysis only discussed five bugs [101]. Therefore, we believe to make progress in combating DC bugs a comprehensive DC bug study is needed.

In this dissertation, we close the gap by performing a large-scale DC bug study. We study 104 real-world DC bugs from four popular open-source distributed systems: Cassandra, HBase, Hadoop MapReduce, and ZooKeeper. We focus our DC bug characteristic study by evaluating what triggers the DC bug to happen, how the DC bug is manifested, and how developers decided to fix those DC bugs.

As a result, our study produces the first complete taxonomy of DC bugs (TaxDC) which we release publicly. TaxDC database contains detailed characteristics of DC bugs that are stored in the form of 2,083 classification labels and 4,528 lines of re-enumerated steps to the bugs that we added manually. We hope that TaxDC will instill various future research technqiues on combating

DC bugs, such as detecting them early before the systems reach into production sites, tools and mechanisms to prevent failures to happen, and automated fixing techniques.

### *1.2.2* FLYMC*: A Fast, Scalable, and Systematic Software Model Checker*

Ideally, bugs should be unearthed in testing, not in deployment [35]. One systematic testing technique that fits the bill is software model checking that runs directly on implementation-level distributed systems [51, 54, 63, 67, 95, 102, 103]. These software model checkers attempt to exercise many possible interleavings of non-deterministic events such as messages and fault timings, hereby pushing the target system into unexplored states and potentially revealing hard-to-detect DC bugs.

One nemesis of software model checkers is the *path-/state-explosion problem*. As a quick illustration, suppose there are 10 concurrent messages (*events*) $\{a, b, .., j\}$, a naive software model checker that explores with a depth-first search (DFS) strategy has to exercise $10!$ (factorial) unique execution paths ($ab..ij$, $ab..ji$, and so on). For this reason, software model checkers have the reputation of being limited to short runs, often just a few events [87].

To tackle this problem, we present FLYMC, a fast, scalable, and systematic software model checker that covers all states relevant to observable events for testing distributed systems implementations. FLYMC achieves scalability by leveraging the internal properties of distributed systems with two reduction algorithms: state symmetry, and event independence; and one prioritization algorithm: parallel flips.

We integrated FLYMC with 8 various distributed systems, and thus, FLYMC successfully reproduced 12 known bugs and detected 10 new bugs which all are confirmed by the developers. In comparison to the state-of-the-art of software model checkers, FLYMC on average is 16× (up to 78×) faster.

### *1.2.3* HMC*: Heuristic Algorithms To Speed Up Software Model Checker*

Reduction algorithms (*e.g.*, DPOR [40, 44]) have shown its power to reduce the number of paths that need to be explored by a checker. Unfortunately, the number of paths left in the to-explore paths oftentimes are still too big to be explored in a limited time budget (*e.g.*, one day). Hence, a checker needs to maximize its ability to discover corner cases by introducing heuristic algorithms that help guide the checker to choose which prefix paths that it should explore next among all to-explore paths.

Prior works, such as MACEMC [63] and PCTCP [83] have introduced random algorithms that are simple to be implemented in a checker, but yet have a reasonable power to direct the checker to reach corner cases. MODIST citeYang+09-Modist and taPCT [84] improves this approach by combining random heuristic with some DPOR and bounded-depth exploration.

But as we explore checker literature, specifically for exploring multithreaded applications [38, 47, 50, 99], there exists an alternative approach that potentially provides a better result than exploring the state space randomly. That is, by staying systematic and exploiting the checker and the target system properties (*e.g.*, queue size, thread interleavings). So, instead of using the naive depth-first search (DFS) approach to explore the to-explore paths, a checker could implement some heuristics to prioritize some prefix paths over the others. The heuristic algorithms goal is to help the checker to explore more-likely prefix paths that will lead the checker to explore states that it has not explored before.

We present HMC, a systematic software model checker to quickly reach corner cases that are empowered with four novel heuristic algorithms: (1) Blocking State-Event, (2) Last State-Event, (3) Miss-Prediction Step, and (4) Prioritized Node Crash. We have integrated HMC to 3 versions of Cassandra, and HMC successfully reproduced 5DC bugs on average 6× (up to 15×) faster than FLYMC which we consider as the state-of-the-art of systematic software model checker. Hence, HMC shows that a checker can be systematic and yet explore corner cases faster.

## 1.3 Summary of Contributions / Overview

Below is the summary of our contributions (and also how the rest of the dissertation is organized).

• **BACKGROUND AND RELATED WORK:** Chapter 2 provides the background on the difference between local concurrency (LC) bugs and distributed concurrency (DC) bugs (Section 2.1), what software model checking is and its challenge (Section 2.2), and existing research ideas to tame the software model checking challenge (Section 2.3).

• **PROBLEM:** We start presenting the first contribution of this dissertation in Chapter 3, which elaborates the taxonomy of DC bugs in several axes of analysis, such as the triggering conditions (Section 3.2), the error and failure symptoms (Section 3.3), and the fix strategies (Section 3.4).

• **SOLUTIONS:** The next two chapters focus on empowering software model checking with reduction algorithms, heuristic algorithms, and various design optimizations to speed up its efficiency in detecting DC bugs. Chapter 4 presents FLYMC, a fast, scalable, and systematic model checker and chapter 5 presents HMC, a software model checker that is equipped with smart heuristic algorithms.

• **CONCLUSIONS AND FUTURE WORK:** Chapter 6 concludes this dissertation by summarizing our work and highlighting the lessons that we learned, and lastly, discussing various directions for future work to further improve distributed systems reliability.

# CHAPTER 2

# BACKGROUND AND RELATED WORK

*"I'm suspecting there is a race condition." — A comment in ZK-1496*

This chapter provides some background on various aspects important to this dissertation. First, Section 2.1 identifies the distinction between concurrency bugs in multi-threaded applications and distributed systems. Second, Section 2.2 explains the concept of software model checking, how it works, and the challenge for software model checking to detect DC bugs. Lastly, Section 2.3 and Section 2.4 will briefly expose how existing software model checkers overcome the challenges and what other techniques exist to combat DC bugs.

## 2.1 The Difference Between DC bugs and LC bugs

When developers hear about concurrency issues, they mostly are going to associate it with single-machine multi-threaded applications where there might exist conflicting accesses to a shared memory location by multiple thread interleavings simultaneously, also known as *local concurrency (LC) bugs*. LC bugs happen due to one of these root causes:

- **Data race:** a situation where two shared memory locations are accessed by different threads simultaneously without proper synchronization and at least one of the accesses involves a write operation.
- **Atomicity violation:** a situation where concurrent threads unexpectedly violate the developers' assumptions of the atomicity of a certain code region.
- **Order Violation:** a situation where concurrent threads violate developers assumptions of two groups of operations to be executed in a certain order.

Well-trained developers will avoid these LC bugs by putting locks and synchronization mechanisms around the block of shared memory accesses, as often as possible use immutable objects and/or thread-safe objects in concurrent operations, etc.. However, in practice, building reliable multi-threads applications are still hard because developers might miss some corner cases. As LC bugs are non-deterministic, hence they are hard to debug, test, and to fix.

In distributed systems, on top of the LC bugs, the systems are also prone to *distributed concurrency (DC) bugs* as a result of multiple nodes' interactions in distributed protocols. Although these nodes don't share memory accesses, the timing of their network communication with other nodes (*e.g.*message arrivals timing) and when data from other nodes' messages will be used in a node's local computation might cause concurrency issues in distributed systems. Furthermore, computer networks might act randomly due to some network partitions, package delays due to nonoptimal package routing, and package drops.

In order to save costs, most of the time distributed systems are run on top of commodity hardware which will fail often. But because these systems serve a large-scale number of client requests, they are expected to be highly available and fault-tolerant. In other words, distributed systems' developers are expected to anticipate failures to arise in any moment of any data operations, such that no data will get corrupted, lost, or end up being inconsistent across the nodes or between the system and the client-side. Therefore, building highly available, fault-tolerant distributed systems is an extremely challenging task [17, 55].

To conclude, DC bugs do not only involve messages and concurrent computations interleavings but it also involves hardware failures timing. Figure 2.1 displays a DC bug in Cassandra that happens because of an untimely order of message arrivals and an untimely node crashes. Suppose only one element occurs, the DC bug will not get manifested.

**CA-12438:**
(1) A client submits Paxos *Write-1 (W1)* to node A with a column in key K's row.
(2) Node A sends W1's prepare messages and propose messages.
(3) All nodes accepted the prepare and propose messages.
(4) Node A sends W1's commit messages.
(5) *Node C crashes before* accepting the commit message.
(6) Nodes A and B accepts the W1's commit messages. At this point, A and B have stored W1 locally.
(7) *Node C reboots*.
(8) Another client submits Paxos *Write-2 (W2)* to A, updating another column in key K's row.
(9) Node A sends W2's prepare messages (then propose and commit messages), accepted by all the nodes. At this point, Paxos nodes incorrectly have *inconsistent data*; A and B store W1-2, but C only stores W2 locally. The read repair did not happen during W2's preparation. Thus, if a client reads K from C, she would get an inconsistent data (missing W1's update).

Figure 2.1: **A DC Bug in Cassandra Paxos.** *The list above summarizes the total order of 48 messages including one crash and one reboot at specific timings.*

## 2.2 Software Model Checking

One technique that is effective to detect DC bugs is software model checking. In this dissertation, we focus on software model checking that is directly integrated into the distributed system implementation. The main goal of a software model checker (in short, **checker**) is to quickly detect DC bugs by controlling all non-deterministic events and determining how the order of all observed events should be explored. Now, let's go through the common design of the checkers.

As shown in Figure 5.3, when a checker explores a path, the path will go through two phases: *(1) path exploration* where it interacts with the target system nodes and *(2) path evaluation* where it evaluates the explored path.

• **PATH EXPLORATION:** In this phase, the checker runs the target workload (*e.g.*, in nodes $A$ and $B$) and intercepts all in-flight messages (*e.g.*, the four concurrent messages $a_1$, $a_2$, $b_1$, and $b_2$ intercepted by the gray "hooks") to control their timings. It then *enables* one message (or other) event at a time (*e.g.*, enable $b_1$). The checker's hooks wait briefly for the target system to reach

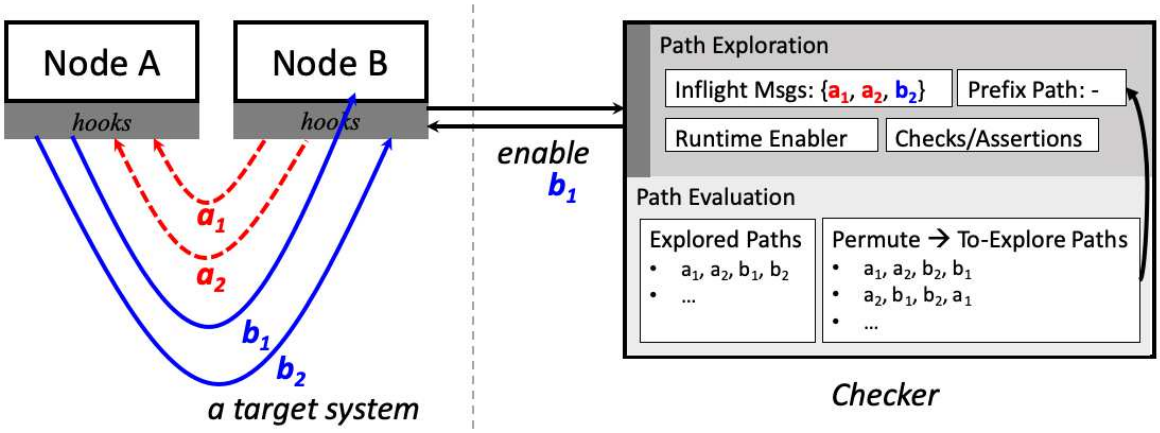



Figure 2.2: **A checker architecture..** *The figure illustrates a typical usage of a distributed system model checker as explained in Section 2.2.*

a quiescent state (after $b_1$ is processed) and wait for nodes to pass the new global states (*e.g.*, $S_1$) to the checker which records it into the *state-event history*. Global states are a combination of all nodes' local states. The testers decide which state variables are important to be observed (*e.g.*, role, leader, ballot number) – a common practice in past checkers. Lastly, the checker runs the assertions to find any safety and liveness violation in the new global state.

After an event is enabled, it is possible that the target system nodes react by sending more message events which will be intercepted again by the checker (*e.g.*$a_3$ generated in response to $b_1$, not shown in the figure). This whole process repeats ($S_0+b_1 \rightarrow S_1$, $S_1+b_2 \rightarrow S_2$, $...+a_3 \rightarrow ...$) until the checker reaches a *termination point* - either when a specification is violated or the workload ends without any violation. This forms an explored path (*e.g.*, $b_1 b_2 ... a_3$). A path implies a unique total ordering of events.

• **PATH EVALUATION:** In this phase, the checker will permute all possible interleavings of the explored path and restart the workload. For example, it might flip $b_2$ before $b_1$, hence generating a new prefix path $b_2 b_1 ...$ within the same workload. Paths can also contain crash/reboot events; for example, a path $b_1 \not{B} B^{\uparrow} b_2 ...$ implies a crash $\not{B}$ and a reboot $B^{\uparrow}$ on node $B$ are injected after $b_1$ is processed but before $b_2$ arrives. The whole path explorations *finish* (has exhausted all state space) when there are no more paths left in the to-explore paths.

• **CAUSAL DEPENDENCY:** A checker must also track causal dependency by using vector clocks. In the example above, if after $a_1$ is enabled, the checker intercepts $c_1$, then $c_1$ is *causally dependent* on $a_1$. That is, $c_1$ has a happens-before relationship with $a_1$. Later on, if enabling $c_1$ generates $a_3$, then $a_3$ is causally dependent on $c_1$ (and also $a_1$), thus $a_1$ and $a_3$ *cannot* be flipped due to their causal dependency.

• **THE PATH EXPLOSION PROBLEM:** As a checker must permute all the possible interleavings, it faces the *path explosion* problem (or also known as state-space explosion problem). In a naive depth-first-search (DFS) method, $E$ concurrent events will generate $E!$ (factorial) paths to exercise. The more complex the workload (*e.g.*number of nodes, system API calls, messages involved as a consequence of the system API calls, crash and reboot events), the larger number of events that need to be permuted.

## 2.3 Software Model Checking Strategies on Detecting DC Bugs

As mentioned before, the main issue that checkers have to overcome is the path-/state-explosion problem. Table 2.1 summarizes the difference between many state-of-the-art checkers. To address the path-explosion problem, some checkers employ reduction algorithms to skip paths that would lead to "similar" states that have been explored before. Such algorithms must be systematic and generic (*i.e.*, not based on randomness or bug-specific knowledge). Another direction that other checkers take is to direct the path explorations based on random walk or some heuristic algorithms.

Subsequent works, CrystalBall [102] and dBug [95] adopted Dynamic Partial Order Reduction (DPOR) independence [40, 44] in a black-box manner without domain-specific knowledge ("■"✓ and "Ind" ✓in Table 2.1). The $\not{N}$+$N^{\uparrow}$ column in Table 2.1 shows that prior checkers did not interleave crash and reboot timings (to limit the path explosion).

To explain the concept of *independency* in DPOR, let us assume the target system is at state $S_i$ with two concurrent events $a_1$ and $a_2$ to enable. If $a_1a_2$ and $a_2a_1$ orderings would lead to *different* states $S_j$ and $S_k$, then $a_1$ and $a_2$ are *dependent*. Thus, dependent events should be reordered to

| | ■ | □ | **Ind** | **Sym** | **Ran** | **Heu** | $\not{N}$+$N^{\uparrow}$ |
|---|---|---|---|---|---|---|---|
| CrystalBall [102] | ✓ | | ✓ | | | | |
| dBug [95] | ✓ | | ✓ | | | | |
| SAMC [67] | | ✓ | ✓ | ✓ | | | $\leq 3$ |
| MACEMC[63] | ✓ | | | | ✓ | | |
| PCTCP [83] | ✓ | | | | ✓ | | |
| MODIST [103] | ✓ | | ✓ | | ✓ | ✓ | 1 |
| taPCT [84] | ✓ | | ✓ | | ✓ | ✓ | |

Table 2.1: **State-of-the-art DC checkers.** *The table is described in Section 2.3. "■" denotes a black-box approach; "□" a white-box approach; "Ind" independence; "Sym" symmetry; "Ran" random; "Heu" heuristic; "$\not{N}$+$N^{\uparrow}$" number of crashes and reboots injected;*

explore new state transitions. Note that "dependent" here is not the same as "causally dependent"; in model checking terminology, two events are dependent if their different interleavings lead to two different states.

However, if both $a_1a_2$ and $a_2a_1$ orderings would lead to the same state $S_j$, then $a_1$ and $a_2$ are *independent*. In other words, exploring one of the reorderings is sufficient, while the other reordering is *unnecessary/redundant*.

SAMC [67] is the first checker that enhances DPOR's independence by exploiting white-box information ("□"✓ in Table 2.1). SAMC allows testers to write some domain-specific algorithms by providing some guiding principles on what kind of rules can cautiously be implemented. Furthermore, SAMC also employs symmetry-based reduction algorithms to handle path with crash and reboot events ("Sym"✓ in Table 2.1).

Symmetry algorithms [39, 97] that SAMC implemented is another major foundation of reduction algorithms. This line of method exploits the architectural symmetry present in the target system. For example, in a ring of nodes, one can rotate the ring and observe the same overall behavior. As another example, if there are three follower nodes all have equal roles, a checker does not need to crash each of the follower nodes in different paths. The implementation of symmetry requires state abstraction. For example, crashing Node 1 (a follower) at global state $S$ is essentially recorded as crashing a follower at global state $S$. Thus, in another path, if the checker

attempts to crash Node 2 (another follower) at the same global state $S$, the checker will skip such an attempt as "crashing a follower at state $S$" has been done before. In other words, it will create an unnecessary/redundant path.

On the other directions, MACEMC [63], PCTCP [83], and others [10, 13] decided to implement some random algorithms to direct the path explorations ("Ran"✓in Table 2.1). Random algorithms are arguably easier to implement but could actually speed up the checkers substantially in comparison to DFS approach. Lastly, some checkers, such as MODIST [103] and taPCT [84] have combined both randomnesses with DPOR algorithm and some depth-bounded exploration heuristics ("Heu"✓in Table 2.1).

There are also other checkers such as DIR [54] and LMC [51] that mainly address the decoupling of local and global explorations (they are not shown in Table 2.1). Finally, others suggested parallelizing DPOR by distributing the path executions across many worker nodes [96, 104]. We believe that these checkers advancements are orthogonal to the reduction and heuristic algorithms mentioned above.

## 2.4 Other Related Work

**VERIFICATION AND TESTING:** There is a growing body of work on new verifiable programming frameworks for distributed systems (*e.g.*, IronFleet [56], PLang [36], Verdi [100]). Such methods are more formal than checkers, but the developers must write proofs that are typically in the thousands of lines. Another work evaluated three formally verified distributed systems including IronFleet and Verdi and uncovered bugs in their implementations [42] (*e.g.*, due to the interaction between the verified and unverified modules). Compared to verification and testing [52, 61, 62] or bug-finding tools [72, 73], software model checking is often considered to be in "between" [19, 46]; for example, checkers deliver higher coverage than testing/bug-finding tools but lower than verification. However, the development cost is cheaper than verification but higher than testing. Our view is that all the techniques above complement each other and advancing each

of the areas is critical to improve distributed systems reliability.

**SYMBOLIC EXECUTION** Another powerful method to check systems correctness is symbolic execution [24, 25, 48]. In comparison to checkers, symbolic execution tool focuses on automatically generating tests to cover diverse code path flows. Hence, it also faces an explosion problem, specifically the code path explosion problem. To address this issue, there exists a huge body of of work [21, 27, 29, 32, 49, 70, 107] on improving how fast the symbolic executor can find the test scenarios leading to detecting bugs. It might be possible that checkers' developers can adopt some heuristic algorithms from the symbolic executors. Another direction might be to combine symbolic execution with software model checking [22].

**POST-MORTEM DIAGNOSIS:** Post-mortem methods such as record-and-replay [43, 74, 75] and flow reconstruction [92, 108] are popular methods to reverse engineer failures. However, tracing is often done in a coarse-grained way [81, 90, 94], but thousands of messages arrive per second and not all of them are logged, thus not all DC bugs can be reconstructed easily in post-mortem analysis. ZooKeeper developers shared with us that occasionally more than ten of iterations of log changes over a long period of time are required to replay DC-related failures at customer sites.

# CHAPTER 3

# TAXDC: A TAXONOMY OF DISTRIBUTED CONCURRENCY BUGS

*"It definitively doesn't look good that this messages comes so late, but I feel this is a serious issue of the SERIAL/LOCAL _ SERIAL consistency levels since this breaks the basic guarantee they [Paxos protocol] exist to provide, ..." — A comment in CA-12126*

This chapter presents our in-depth analysis of 104 *distributed concurrency* (DC) bugs. The bugs came from four popular distributed systems: Cassandra [2], HBase [5], Hadoop MapReduce [3], and ZooKeeper [6]. We introduce TaxDC, a comprehensive taxonomy of real-world DC bugs across several axes of analysis such as the triggering timing condition and input preconditions, error and failure symptoms, and fix strategies, as shown in detail in Table 3.1. The results of our study are stored in the form of 2,083 classification labels in TaxDC database [1].

As our main contribution, TaxDC will be the first large-scale DC-bug benchmark. In the last five years, bug benchmarks for *local concurrency* (LC) bugs have been released [59, 105], but no large-scale benchmarks exist for DC bugs. Researchers who want to evaluate the effectiveness of existing or new tools in combating DC bugs do not have a benchmark reference. TaxDC provides researchers with more than 100 thoroughly taxonomized DC bugs to choose from. Practitioners can also use TaxDC to check whether their systems have similar bugs. The DC bugs we studied are considerably general, representing bugs in popular types of distributed systems.

As a side contribution, TaxDC can help open up new research directions. In the past, the lack of understanding of real-world DC bugs has hindered researchers to innovate new ways to combat DC bugs. The state of the art focuses on three lines of research: monitoring and postmortem debugging [43, 74, 75, 87], testing and model checking [54, 63, 67, 95, 103], and verifiable language frameworks [36, 100]. We hope our study will not only improve these lines of research, but also inspire new research in bug detection tool design, runtime prevention, and bug fixing.

## 3.1 Methodology

Our study examined bugs from four widely-deployed open-source distributed systems that represent a diverse set of system architectures: Hadoop MapReduce (including Yarn) [3] representing distributed computing frameworks, HBase [5] and Cassandra [2] representing distributed key-value stores (also known as NoSQL systems), and ZooKeeper [6] representing synchronization services. They are all fully complete systems containing many complex concurrent protocols. Throughout this chapter, we will present short examples of DC bugs in these systems. Some detailed examples are illustrated in Figure 3.2 and 3.4.

The development projects of our target systems are all hosted under Apache Software Foundation wherein organized issue repositories (named "JIRA") are maintained. To date, across the four systems, there are over 60,000 issues submitted. One major challenge is that issues pertaining to DC bugs do not always contain plain terms such as "concurrency", "race", "atomicity", etc. Scanning all the issues is a daunting task. Thus, we started our study from an open source cloud bug study (CBS) database [1], which already labels issues related to concurrency bugs. However, the CBS work did not differentiate DC from LC bugs and did not dissect DC bugs further.

From CBS, we first filtered out LC bugs, then exclude DC bugs that do not contain clear description, and finally randomly picked 104 samples from the remaining detailed DC bugs, specifically 19 Cassandra, 30 HBase, 36 Hadoop MapReduce, and 19 ZooKeeper DC bugs, reported in January 2011-2014 (the time range of CBS work). We have seen much fewer clearly explained DC bugs in CBS from Cassandra and ZooKeeper than those from HBase and Hadoop MapReduce, which may be related to the fact that they are different types of distributed systems. For example, ZooKeeper, as a synchronization service, is quite robust as it is built on the assumption of event asynchrony since day one. Cassandra was built on eventual consistency, and thus did not have many complex transactions, until Cassandra adopts Paxos for its Light Weight Transaction (LWT) protocol. Up until today, we still see many DC bugs are reported to each system's JIRA repository.

### *3.1.1 Taxonomy*

| **Triggering** (Section 3.2) |
|---|
| *What is the triggering timing condition?* |
| Message arrives unexpectedly late/early |
| Message arrives unexpectedly in the middle |
| Fault (component failures) at an unexpected state |
| Reboot at an unexpected state |
| *What are the triggering inputs preconditions?* |
| Fault, reboot, timeout, background protocols, and others |
| *What is the triggering scope?* |
| *How many nodes/messages/protocols are involved?* |
| **Errors & Failures** (Section 3.3) |
| *What is the error symptom?* |
| Local memory exceptions |
| Local semantic error messages & exceptions |
| Local hang |
| Local silent errors (inconsistent local states) |
| Global missing messages |
| Global unexpected messages |
| Global silent errors (inconsistent global states) |
| *What is the failure symptom?* |
| Node downtimes, data loss/corruption, operation failures, slowdowns |
| **Fixing** (Section 3.4) |
| *What is the fix strategy?* |
| Fix Timing: add global synchronization |
| Fix Timing: add local synchronization |
| Fix Handling: retry message handling at a later time |
| Fix Handling: ignore a message |
| Fix Handling: accepting a message without new computation logics |
| Fix Handling: others |

Table 3.1: **Taxonomy of DC Bugs.**

We study the characteristics of DC bugs along three key stages: triggering, errors & failures, and fixing (Table 3.1). *Triggering* is the process where software execution states deviate from correct to incorrect under specific conditions. At the end of this process, the manifestation of DC bugs changes from non-deterministic to deterministic. *Errors and failures* are internal and external software misbehaviors. *Fixing* shows how developers correct the bug. We will discuss in detail

these categories in their respective sections.

### 3.1.2 Threats to Validity

For every bug, we first ensure that the developers marked it as a real bug (not a false positive). We also check that the bug description is clear. We then *re-enumerate* the full sequence of operations (the "*steps*") to a clearer and more concise description such as the ones in Figure 3.2. Our study cannot and does not cover DC bugs not fixed by the developers. Even for fixed bugs, we do not cover those that are not described clearly in the bug repositories, a sacrifice we had to make to maintain the accuracy of our results.

Readers should be cautioned not to generalize the statistics we report as each distributed system has unique purpose, design and implementation. For example, we observe 2:1 overall ratio between order and atomicity violations Section 3.2.1, however the individual ratios are different across the four systems (*e.g.* 1:2 in ZooKeeper and 6:1 in MapReduce). Like all empirical studies, our findings have to be interpreted with our methodology in mind.

### 3.1.3 TaxDC Database

We name the product of our study TaxDC database. TaxDC contains in total 2,083 classification labels and 4,528 lines of clear and concise re-description of the bugs (our version, that we manually wrote) including the re-enumeration of the steps, triggering conditions, errors and fixes. We release TaxDC to the public [1]. We believe TaxDC will be a rich "bug benchmark" for researchers who want to tackle distributed concurrency problems. They will have sample bugs to begin with, advance their work, and do not have to repeat our multi-people-year effort.

1. Please check our group website at http://ucare.cs.uchicago.edu

### *3.1.4 Detailed Terminologies*

Below are the detailed terminologies we use in this chapter. We use the term "state" to interchangeably imply *local state* (both in-memory and on-disk per-node state) or *global state* (a collection of local states and outstanding messages). A *protocol* (*e.g.*, read, write, load balancing) creates a chain of events that modify system state. User-facing protocols are referred as *foreground* protocols while those generated by daemons or operators are referred as *background* protocols.

We consider four types of *events*: message, local computation, fault and reboot. The term *fault* represents component failures such as crashes, timeouts, and disk errors. A *timeout* (system-specific) implies a network disconnection or busy peer node. A *crash* usually implies the node experiences a power failure. A *reboot* means the node comes back up.

Throughout this chapter, we present bug examples by abstracting system-specific names. As shown in Figure 3.1, we use capital letters for nodes (*e.g.*, A, B), two small letters for a message between two nodes (*ab* is from A to B). Occasionally, we attach system-specific information in the subscript (*e.g.*, $A_{\text{AppMaster}}$ sends $ab_{\text{taskKill}}$ message to $B_{\text{NodeManager}}$). We use "/" to imply concurrency ($ac/bc$ implies the two messages can arrive at C in different orders, *ac* or *bc* first). A dash, "–", means causal relation of two events (*ab*-*bc* means *ab* causally precedes *bc*). Finally, we use "N*" to represent crash, "N!" reboot, and "N+" local computation at N.

We cite bug examples with clickable hyperlinks (*e.g.*, MR-3274). To keep most examples uniform, we use MapReduce examples whenever possible. We use the following abbreviations for system names: "c/CA" for Cassandra, "h/HB" for HBase, "m/MR" for Hadoop MapReduce, and "z/ZK" for ZooKeeper; and for system-specific components: "AM" for application master, "RM" for resource manager, "NM" for node manager, "RS" for region server, and "ZAB" for ZooKeeper atomic broadcast.

## 3.2 Trigger

DC bugs often have a long triggering process, with many local and global events involved. To better reason about this complicated process, we study them from two perspectives:

1. *Timing conditions* (Section 3.2.1): For every DC bug, we identify the smallest set of concurrent events $E$, so that a specific ordering of $E$ can guarantee the bug manifestation. This is similar to the interleaving condition for LC bugs.

2. *Input preconditions* (Section 3.2.2): In order for those events in $E$ to happen, regardless of the ordering, certain inputs or fault conditions (*e.g.*, node crashes) must occur. This is similar to the input condition for LC bugs.

Understanding the triggering can help the design of testing tools that can proactively trigger DC bugs, bug detection tools that can predict which bugs can be triggered through program analysis, and failure prevention tools that can sabotage the triggering conditions at run time.

### *3.2.1 Timing Conditions*

Most DC bugs are triggered by some *timing conditions* (TC), either by untimely delivery of messages, referred to as *message timing bugs*, or by untimely faults or reboots, referred to as *fault timing bugs*. Rarely DC bugs are triggered by both untimely messages and faults, referred to as *message-fault bugs*. Table 3.2 shows the per-system breakdown and Figure 3.3a (TC) the overall breakdown. Since a few bugs are triggered by more than one type of timing conditions (Section 3.2.3), the sum of numbers in Table 3.2 is slightly larger than the total number of DC bugs.

**Message Timing Bugs.** The timing conditions can be abstracted to two categories:

a. *Order violation* (44% in Table 3.2) means a DC bug manifests whenever a message comes earlier (later) than another event, which is another message or a local computation, but not when the message comes later (earlier).

| | Ordering | Atomicity | Fault | Reboot |
|---|---|---|---|---|
| CA | 4 | 4 | 6 | 5 |
| HB | 13 | 9 | 8 | 1 |
| MR | 25 | 4 | 5 | 3 |
| ZK | 4 | 8 | 7 | 5 |
| All | 46 | 25 | 26 | 14 |

Table 3.2: **#DC bugs triggered by timing conditions (Section 3.2.1).** *The total is more than 104 because some bugs require more than one triggering condition. More specifically, 46 bugs (44%) are caused* only *by ordering violations, 21 bugs (20%)* only *by atomicity violations, and 4 bugs (4%) by multiple timing conditions (as also shown in Figure 3.3a).*

b. *Atomicity violation* (20% in Table 3.2) means a DC bug manifests whenever a message comes in the middle of a set of events, which is a local computation or global communication, but not when the message comes either before or after the events.

LC and DC bugs are similar in that their timing conditions can both be abstracted into the above two types. However, the subjects in these conditions are different: shared memory accesses in LC and message deliveries in DC. The ratio between order violation and atomicity violation bugs are also different: previous study of LC bugs showed that atomicity violations are much more common than order violations in practice [78]; our study of DC bugs shows that this relationship does not apply or even gets reversed in several representative distributed systems.

An order violation can originate from a race between two messages (*message-message race*) at one node. The race can happen between two message arrivals. For example, Figure 3.1a illustrates $ac/bc$ race at node C in MR-3274. Specifically, $\mathrm{B}_{\mathrm{RM}}$ sends to $\mathrm{C}_{\mathrm{NM}}$ a task-init message ($bc_{\mathrm{init}}$), and soon afterwards, $\mathrm{A}_{\mathrm{AM}}$ sends to $\mathrm{C}_{\mathrm{NM}}$ a task-kill preemption message ($ac_{\mathrm{kill}}$), however $ac_{\mathrm{kill}}$ arrives *before* $bc_{\mathrm{init}}$ and thus is incorrectly ignored by C. The bug would not manifest if $ac_{\mathrm{kill}}$ arrives *after* $bc_{\mathrm{init}}$ (Figure 3.1b). Message-message race can also happen between a message arrival and a message sending. For example, the $ab/bc$ race in Figure 3.1c depicts HB-5780. In this bug, $\mathrm{B}_{\mathrm{RS}}$ sends to $\mathrm{C}_{\mathrm{Master}}$ a cluster-join request ($bc_{\mathrm{join}}$) unexpectedly *before* a security-key message ($ab_{\mathrm{key}}$) from $\mathrm{A}_{\mathrm{ZK}}$ arrives at B, causing the initialization to abort.

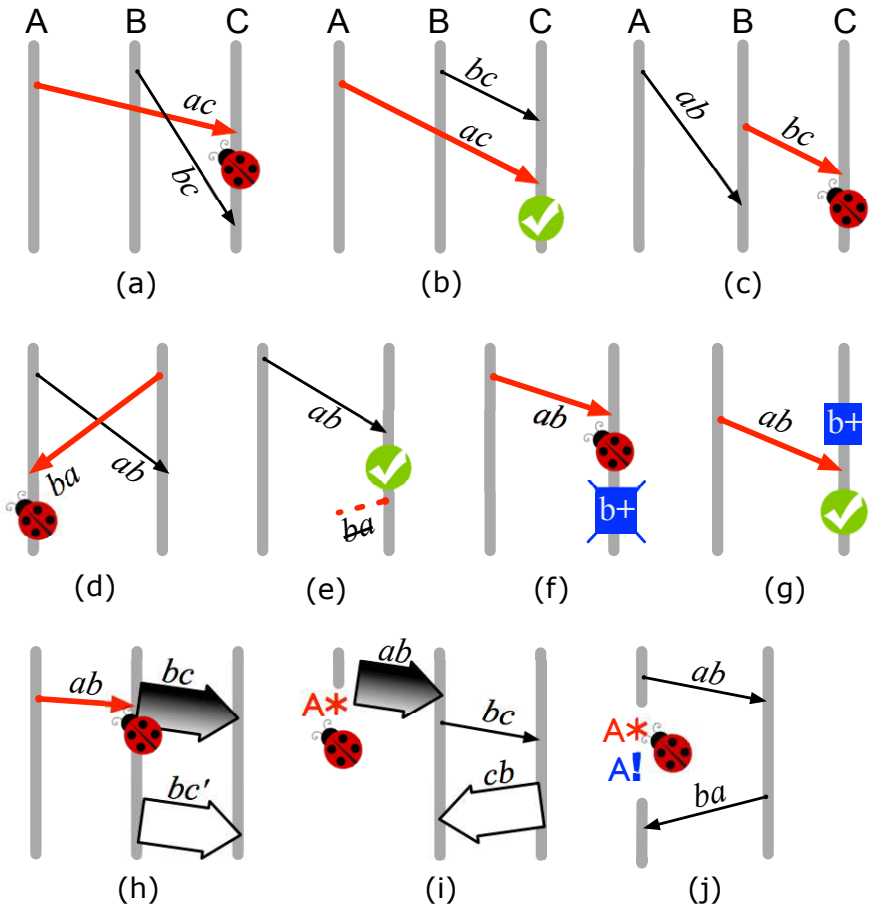



Figure 3.1: **Triggering patterns (Section 3.2.1).** *The three vertical lines represent the timeline of nodes A, B and C. An arrow with* `xy` *label implies a message from X to Y. A square box with label* `x+` *implies a local state-modifying computation at node X. A thick arrow implies a set of messages performing an atomic operation.* `X*` *and* `X!` *implies a crash and reboot at node X respectively (Section 3.1.4). All figures are discussed in Section 3.2.1*

Interestingly, message-message race can also occur concurrently across two nodes. For example, Figure 3.1d illustrates *ab*/*ba* race crisscrossing two nodes A and B in MR-5358. Specifically, $A_{AM}$ sends $ab_{kill}$ to a backup speculative task at $B_{NM}$ because the job has completed, but concurrently the backup task at B sends $ba_{complete}$ to A, creating a double-complete exception at A. If $ab_{kill}$ arrives early at B, *ba* will not exist and the bug will not manifest (Figure 3.1e).

An order violation can also originate from a race between a message and a local computation (*message-compute race*). For example, Figure 3.1f illustrates *ab*/*b*+ race in MR-4157. First, $B_{AM}$ was informed that a task has finished and B plans to close the job and remove its local temporary

files ($b$+). However, just *before* $b$+, $A_{RM}$ sends to B a kill message ($ab$) and hence the files are never removed, eventually creating space issues. To prevent the failure, the kill message has to arrive after the local cleanup (Figure 3.1g).

An atomicity violation, as defined above, originates when a message arrives in the middle of a supposedly-atomic local computation or global communication. For example, Figure 3.1h illustrates MR-5009. When $B_{NM}$ is in the middle of a commit transaction, transferring task output data ($bc$) to $C_{HDFS}$, $A_{RM}$ sends a kill preemption message ($ab$) to B, preempting the task without resetting commit states on C. The system is never able to finish the commit — when B later reruns the task and tries to commit to C ($bc$'), C throws a double-commit exception. This failure would not happen if the kill message ($ab$) comes before or after the commit transaction ($bc$).

**Fault and Reboot Timing Bugs.** Fault and reboot timing bugs (32% in Table 3.2) manifest when faults and/or reboots occur at specific global states $S_i$; the bugs do not manifest if the faults and reboots happen at different global states $S_j$.

Figure 3.1i illustrates a fault-timing bug in MR-3858. Here, $A_{NM1}$ is sending a task's output to $B_{AM}$ ($ab$) but A crashes in the middle (A*) leaving the output half-sent. The system is then unable to recover from this untimely crash — B detects the fault and reruns the task at $C_{NM2}$ (via $bc$) and later when C re-sends the output ($cb$), B throws an exception. This bug would not manifest, if the crash (A*) happens before/after the output transfer ($ab$).

Figure 3.1j depicts a reboot-timing bug in MR-3186. Here, $A_{RM}$ sends a job ($ab$) to $B_{AM}$ and while B is executing the job, A crashes and reboots (A*, A!) losing all its in-memory job description. Later, B sends a job-commit message ($ba$) but A throws an exception because A does not have the job information. The bug would not manifest if A reboots later: if A is still down when B sends $ba_{commit}$ message, B will realize the crash and cancel the job before A reboots and A will repeat the entire job assignment correctly.

**Message-Fault Bugs.** Four DC bugs are caused by a combination of messages and faults. For example, in Figure 3.2, a message (step 4) arrives in the middle of some atomic operation (step 3-6). This message atomicity violation leads to an error that further requires a fault timing (step 8) to become an externally visible failure.

> **Finding #1:** DC bugs are triggered mostly by *untimely messages* (64% in Table 3.2) and sometimes by *untimely faults/reboots* (32%), and occasionally by a *combination* of both (4%). Among untimely messages, two thirds commit order violations due to message-message or message-computation race on the node they arrive; the others commit atomicity violations.

### *3.2.2 Input Preconditions*

The previous section presents simple timing conditions that can be understood in few simple steps. In practice, many of the conditions happen "deep" in system execution. In other words, the triggering path is caused by complex *input preconditions* (IP) such as faults, reboots, and multiple protocols. Let's use the same example in Figure 3.2. First, a fault and a reboot (step 1-2) and a client request (step 4) must happen to create a path to the message atomicity violation (step 4 interfering with step 3-6). Second, conflicting messages from two different protocols (ZAB and

> **ZK-1264:** (1) *Follower F crashed* in the past, (2) *F reboots* and joins the cluster, (3) Leader L sync data with F and send snapshot, (4) *In the middle of step 3-6*, client updates data with Tx-#15; L forwards the update to F, (5) F applies the update in memory only, due to a concurrent sync, (6) L tells F syncing is finished, (7) Client updates data with Tx-#16; F writes update to disk correctly, (8) *F crashes*, (9) *F reboots* and joins the cluster again, (10) L sync data with F, but this time L sends only "diff" starting with Tx-#17 (11) F permanently *loses data* from Tx-#15, inconsistent with L and other followers!

Figure 3.2: **ZK-1264 bug description.** *This figure shows a DC bug caused by a mix of untimely message arrivals and crash event.*

NodeJoin initiated in step 2 and 4) have to follow specific bug-triggering timing conditions. Even after the atomicity violation (after step 6), the bug is not guaranteed to lead to any error yet (*i.e.*, a benign race). Finally, the follower experiences an untimely fault (step 8), such that after it reboots (step 9), a global replica-inconsistency error will happen (step 11). Put it in a reverse way, before step 8, the global state is $S_{\text{i}}$ and $S_{\text{i}}$+crash→error, and the only way for the system to reach $S_{\text{i}}$ is from complex preconditions such as a fault, a reboot, and some foreground and background protocols.

Statistically, Figure 3.3b (IP-FLT) shows that 63% of DC bugs must have at least one fault. In more detail, Figure 3.3c-e (IP-TO, IP-CR, IP-RB) shows the percentage of issues that require timeouts, crashes and reboots respectively, including how many instances of such faults must be there; the rest is other faults such as disk errors (not shown).

Figure 3.3f (IP-PR) shows how many "protocol initiations" mentioned in the bug description. For example, if the system needs to perform one execution of background protocol and also three concurrent calls to the `write` protocol, then we label it with four protocol initiations. Up to 3 protocol initiations covers three quarters of DC bugs. When we count the number of *unique* protocols involved in all the bugs we study, we record 10 Cassandra, 13 HBase, 10 MapReduce, 6 ZooKeeper unique protocols, or 39 protocols in total. This again highlights the complexity of fully complete systems. Figure 3.3g (IP-B/F) shows our categorization of protocols that are concurrently running into foreground only, background only, and foreground-background (mix) categories. More than three quarters of the bugs involve some background protocols and about a quarter involves a mix of foreground and background protocols.

**Finding #2:** Many DC bugs need *complex input preconditions*, such as faults (63% in Figure 3.3b), multiple protocols (80% in Figure 3.3f), and background protocols (81% in Figure 3.3g) .

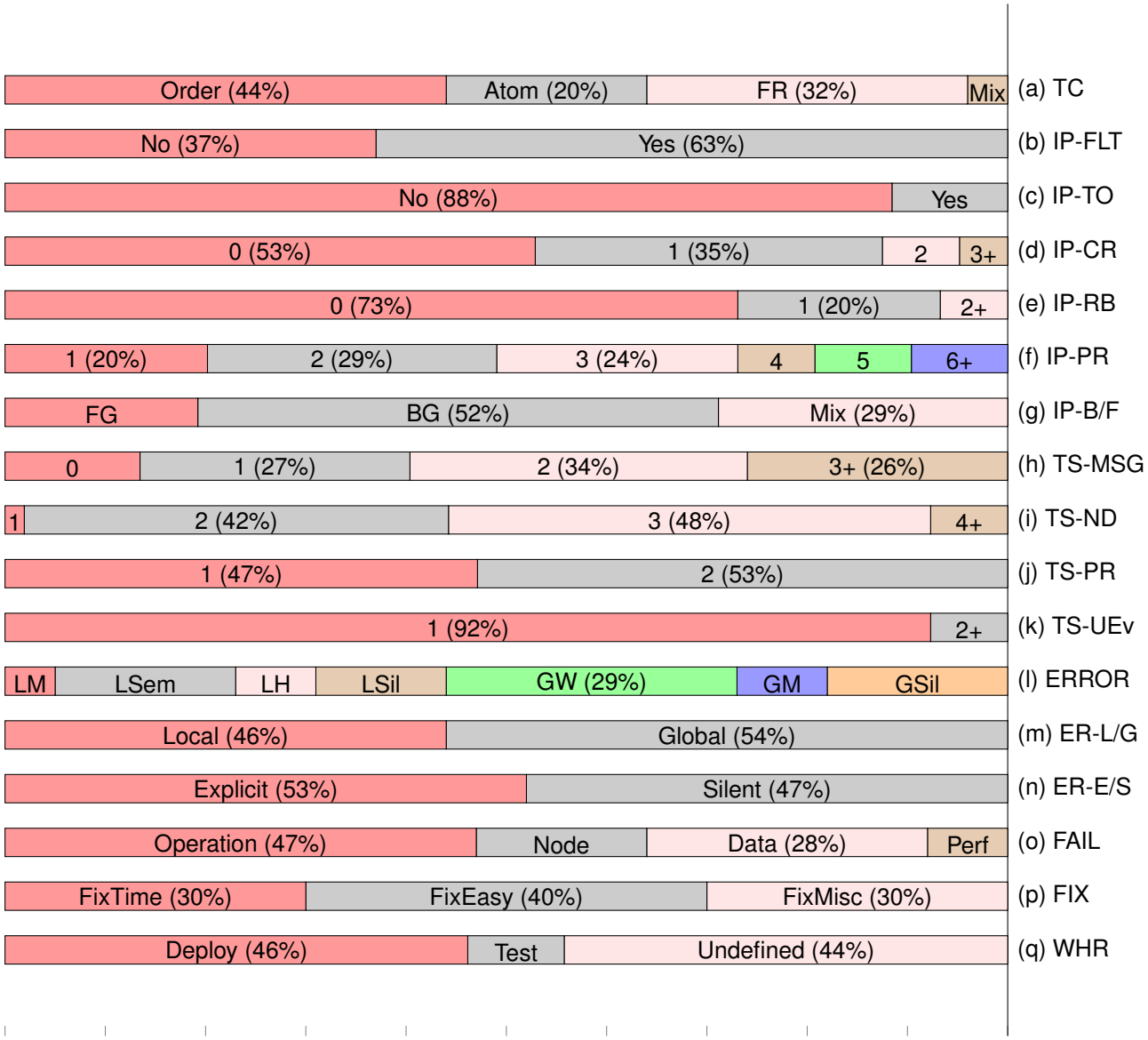



Figure 3.3: **Statistical overview of TaxDC.** *Timing Conditions (TC) is discussed in Section 3.2.1, Input Preconditions (IP) in Section 3.2.2, Triggering Scope (TS) in Section 3.2.3, Errors (ER) in Section 3.3.1, Failures (FAIL) in Section 3.3.2, Fixes (FIX) in Section 3.4, and Where Found (WHR) in Section 3.6.*

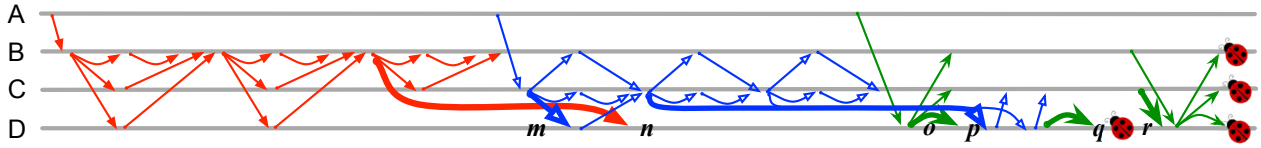



Figure 3.4: **A Cassandra's Paxos bug.** *In CA-6023, three key-value updates (different arrow types) concurrently execute the Paxos protocol on four nodes (we simplify from the actual six nodes). The bug requires three message-message race conditions: (1)* m *arrives before* n*, (2)* o *before* p*, and (3)* q *before* r*, which collectively makes D corrupt the data and propagate the corruption to all replicas after the last broadcast. Note that the bug would not surface if any of the conditions did not happen. It took us one full day to study this bug.*

### *3.2.3 Triggering Scope*

We now analyze the *triggering scope* (TS), which is a complexity measure of DC-bug timing conditions. We use four metrics to measure the scope: message (TS-MSG), node (TS-ND), protocol (TS-PR), and untimely event (TS-UEv) counts (see Figure 3.3h-k). This statistic is important with respect to the scalability of model checking, bug detection and failure diagnostic tools.

Message count implies the minimum number of messages involved in $E$ as defined in the beginning of section 3.2. Figure 3.3h (TS-MSG) shows that one or two triggering messages are the most common, with 7 messages as the maximum. Informally, zero implies fault timing bugs without any message-related races, one implies message-compute race, two implies message-message as in Figure 3.1a, and three implies a scenario such as *ac*/(*ab*-*bc*) race where *ab* and *ac* are concurrent or non-blocking message sending operations.

The node and protocol scopes present how many nodes and protocols are involved within the message scope. Figure 3.3i-j (TS-ND and TS-PR) shows that the scale of node and protocol triggering scope is also small, mostly two or three nodes and one or two protocols.

The untimely events count implies the total number of order violations, atomicity violations, untimely faults and reboots in the triggering timing condition of a bug. Figure 3.3k (TS-UEv) shows that only eight bugs require more than one untimely events. Four of them are message-fault bugs, each requiring one untimely message and one untimely fault to trigger (*e.g.*, step 4 and 8 in Figure 3.2). Three are fault-reboot timing bugs, each requiring one untimely fault and one

untimely reboot. The last one is CA-6023, shown in Figure 3.4, requiring three message-message order violations to happen.

> **Finding #3:** The *timing conditions* of most DC bugs only involve *one to three* messages, nodes, and protocols (>90% in Figure 3.3h-j). Most DC bugs are mostly triggered by only *one* untimely event (92% in Figure 3.3k).

## 3.3 Errors and Failures

### *3.3.1 Error Symptoms*

From the triggering conditions, we then scrutinize the *first* error that happens immediately after. First errors are the pivotal point that bridges the triggering and error-propagation process. Identifying first errors help failure diagnosis get closer to disclosing bug triggering and root causes and help bug detection get closer to accurately predict failures.

We categorize first errors into *local* errors and *global* errors, based on whether they can be observed from the triggering node $N_T$ alone. Here, $N_T$ is the node where triggering ends. It is the receiver node of untimely messages (*e.g.*, node C in Figure 3.1a) or the node with untimely fault (*e.g.*, node A in Figure 3.1i). For each error, we also check whether it is an *explicit* or *silent* error. Table 3.3 and Figure 3.3l (ERR) show the per-system and overall breakdowns respectively. Some MapReduce bugs caused multiple concurrent first errors of different types.

First, DC bugs can manifest into both local explicit errors and local silent errors. The former includes *memory exceptions* such as null-pointer exceptions (5% in Table 3.3) and *semantic errors* such as wrong state-machine transition exceptions thrown by the software (19%). Local silent errors include *hangs*, such as forever waiting for certain states to change or certain messages to arrive which are typically observed implicitly by users (9%), and *local silent* state corruption, such as half-cleaned temporary files (13%).

| | Local Errors | | | | Global Errors | | |
|---|---|---|---|---|---|---|---|
| | Mem | Sem | Hang | Sil | Wrong | Miss | Sil |
| CA | 2 | 0 | 0 | 4 | 3 | 3 | 7 |
| HB | 1 | 2 | 1 | 2 | 15 | 3 | 6 |
| MR | 2 | 13 | 7 | 4 | 14 | 4 | 0 |
| ZK | 0 | 6 | 2 | 5 | 1 | 0 | 5 |
| All | 5 | 21 | 10 | 15 | 33 | 10 | 18 |

Table 3.3: **First error symptoms of DC bugs (Section 3.3.1).** *Some bugs cause multiple concurrent first errors.*

When local error is non-obvious in $N_T$, we analyze if the error is observable in other nodes communicating with $N_T$. Many DC bugs manifest into explicit global errors through *wrong messages* (29% in Table 3.3). Specifically, the communicating node receives an incorrect message from $N_T$, and throws an exception during the message handling. However, a few DC bugs still lead to silent global errors. These include *missing messages*, where $N_T$ never sends a reply that the communicating node is waiting for in the absence of timeout (9%), and *global silent* state corruption such as replica inconsistencies between $N_T$ and the other nodes (16%).

**Finding #4:** *Local* and *global* first errors are about equally common; 46% vs. 54% in Figure 3.3m (ER-L/G). About half of the DC bugs generate *explicit* first errors (53%), including local exceptions and global wrong messages, and the remaining DC bugs lead to *silent* errors (47%), as shown in Figure 3.3n (ER-E/S). Some of them immediately lead to hangs in the triggering node $N_T$ (9%) or a node communicating with $N_T$ (9%).

### 3.3.2 Failure Symptoms

Figure 3.3o (FAIL) shows that errors from DC bugs will eventually lead to a wide range of fatal failures including node downtimes (17%), data loss/corruption/inconsistencies (28%), operation failures (47%), and performance degradation (8%). A node downtime happens when the node

either crashes or hangs (*i.e.*, it may still be heartbeating). It happens to both master/leader nodes and worker/follower nodes in our study. Data-related failures and performance problems are an artifact of incorrect state logic induced from DC bugs. For example, in HBase, concurrent region updates and log splittings can cause data loss. In Cassandra, some dead nodes are incorrectly listed as alive causing unnecessary data movement that degrades performance. Node downtimes and data-related failures could also cause some operations to fail. To avoid double counting, we consider a bug as causing operation failures only when it does not cause node downtimes or data-related failures.

## 3.4 Fixes

We next analyze bug patches to understand developers' fix strategies. In general, we find that DC bugs can be fixed by either disabling the triggering timing or changing the system's handling to that timing (*fix timing* vs. *fix handling*). The former prevents concurrency with extra synchronization and the latter allows concurrency by handling untimely events properly. Since message timing bugs are fixed quite differently from fault timing bugs, we separate them below.

### *3.4.1 Message Timing Bug Fixes*

The left half of Table 3.4 shows that only one fifth of message timing bugs are fixed by disabling the triggering timing, through either global or local synchronization. Only a couple of bugs are fixed through extra *global synchronization*, mainly due to its complexity and communication cost. For example, to prevent a triggering pattern $b+/ab$ in Figure 3.1f, MR-5465's fix *adds* a monitor on $\text{A}_{\text{RM}}$ to wait for $ba_{\text{done}}$ message from $\text{B}_{\text{AM}}$ after B finishes with its local computation ($b+$); the result is $b+$-$ba$-$ab$ global serialization. More often, the buggy timing is disabled through *local synchronization*, such as re-ordering message sending operations within a single node. For example, HB-5780's fix for $ab/bc$ race in Figure 3.1c forces the sending of $bc$ request at B to wait for the receipt of $ab$; the result is $ab$-$bc$ local serialization at B.

| | Fix Timing | | Fix Handling | | | |
|---|---|---|---|---|---|---|
| | Glob | Loc | Ret | Ign | Acc | Misc |
| CA | 0 | 0 | 0 | 1 | 3 | 4 |
| HB | 2 | 7 | 2 | 1 | 7 | 3 |
| MR | 2 | 8 | 2 | 7 | 8 | 3 |
| ZK | 0 | 4 | 0 | 3 | 0 | 1 |
| All | 4 | 19 | 4 | 12 | 18 | 11 |

Table 3.4: **Fix strategies for message timing bugs (Section 3.4.1).** *Some bugs require more than one fix strategy.*

The right half of Table 3.4 shows that fix handling is more popular. Fix handling fortunately can be simple; many fixes do *not* introduce brand-new computation logic into the system, which can be done in three ways. First, the fix can handle the untimely message by simply *retrying* it at a later time (as opposed to ignoring or accepting it incorrectly). For example, to handle $bc/ac$ race in Figure 3.1a, MR-3274 retries the unexpectedly-early $ac_{\text{kill}}$ message at a later time, right after the to-be-killed task starts. Second, the fix can simply *ignore* the message (as opposed to accepting it incorrectly). For example, to handle $ab/ba$ race in Figure 3.1d, MR-5358 simply ignores the unexpectedly-late $ba_{\text{finish}}$ message that arrives after $\text{A}_{\text{AM}}$ sends an $ab_{\text{kill}}$ message. Finally, the patch can simply *accept* the untimely message by *re-using* existing handlers (as opposed to ignoring it or throwing an error). For example, MR-2995's fix changes the node AM to accept an unexpectedly-early expiration message using an existing handler that was originally designed to accept the same message at a later state of AM. MR-5198's fix handles the atomicity violation by using an existing handler and simply cancels the atomicity violating local operation. The rest of the fix-handling cases require new computation logic to fix bugs.

### *3.4.2 Fault/Reboot Timing Bug Fixes*

Table 3.5 summarizes fix strategies for fault/reboot timing bugs. Unlike message timing, only rare bugs can be fixed by controlling the triggering timing either globally or locally (*e.g.*, by controlling the timing of the fault recovery actions). A prime example is an HBase cluster-wide restart

| | Fix Timing | | Fix Handling | | | |
|---|---|---|---|---|---|---|
| | G | L | Detect | | Recover | |
| | | | TO | Msg | Canc | Misc |
| CA | 1 | 0 | 3 | 2 | 4 | 6 |
| HB | 0 | 1 | 3 | 1 | 6 | 1 |
| MR | 2 | 1 | 1 | 1 | 2 | 1 |
| ZK | 0 | 3 | 0 | 1 | 1 | 7 |
| All | 3 | 5 | 7 | 5 | 13 | 15 |

Table 3.5: **Fix strategies for fault/reboot timing bugs (Section 3.4.2).** *Some bugs require more than one fix strategy.*

scenario (HB-3596). Here, as A shuts down earlier, B assumes responsibility of A's regions (via a region-takeover recovery protocol), but soon B shuts down as well with the regions still locked in ZooKeeper and the takeover cannot be resumed after restart. The patch simply adds a delay before a node starts region takeover so that it will likely get forced down before the takeover starts.

For the majority of fault timing bugs, their patches conduct two tasks: (1) detect the local/global state inconsistency caused by the fault and (2) repair/recover the inconsistency. The former is accomplished through timeouts, additional message exchanges, or others (omitted from Table 3.5). The latter can be achieved by simply canceling operations or adding new computation logic.

**Finding #5:** A *small number of fix strategies* have fixed most DC bugs. A few DC bugs are fixed by *disabling* the triggering timing (30% in Figure 3.3p), occasionally through extra messages and mostly through local operation re-orderings. Most DC bugs are fixed by better handling the triggering timing, most of which do not introduce new computation logic — they *ignore* or *delay* messages, *re-use* existing handlers, and *cancel* computation (40%).

## 3.5 Root Causes

It is difficult to know for sure why many DC-bug triggering conditions were not anticipated by the developers (*i.e.*, the root causes). In this section, we postulate some possible and common misbeliefs behind DC bugs.

*"One hop is faster than two hop."* Some bugs manifest under scenario *bc*/(*ba-ac*), similar to Figure 3.1a. Developers may assume that *bc* (one hop) should arrive earlier than *ba-ac* (two hops), but *ac* can arrive earlier and hit a DC bug.

*"No hop is faster than one hop."* Some bugs manifest under scenario *ba*-(*b*+/*ab*), similar to Figure 3.1f. Developers may incorrectly expect *b*+ (local computation with no hop) to always finish before *ab* arrives (one hop).

*"Atomic blocks cannot be broken."* Developers might believe that "atomic" blocks (local or global transactions) can only be broken unintentionally by some faults such as crashes. However, we see a few cases where atomic blocks are broken inadvertently by the system itself, specifically via untimely arrival of kill/preemption messages in the middle of an atomic block. More often, the system does not record this interruption and thus unconsciously leaves state changes half way. Contrary, in fault-induced interruption, some fault recovery protocol typically will handle it.

*"Interactions between multiple protocols seem to be safe."* In common cases, multiple protocols rarely interact, and even when they do, non-deterministic DC bugs might not surface. This can be unwittingly treated as normally safe, but does not mean completely safe.

*"Enough states are maintained."* Untimely events can unexpectedly corrupt system states and when this happens the system does not have enough information to recollect what had happened in the past, as not all event history is logged. We observe that some fixes add new in-memory/on-disk state variables to handle untimely message and fault timings.

*"The message sent earlier will take effect earlier"* For example, we discussed earlier that MR-3274 is triggered by a *bc*/*ac* race, and the software cannot handle $ac_{\text{kill}}$ when it arrives earlier than $bc_{\text{init}}$.

(Figure 3.1a), In fact, $bc_{\text{init}}$ is caused by another message $ab_{\text{init}}$ that leaves node $A_{AM}$ a while before $ac_{\text{kill}}$ does, which may be why developers assume *bc* will arrive before *ac* at C.

> **Finding #6:** Many DC bugs are related with a few common misconceptions that are unique to distributed systems.

## 3.6 Other Statistics

We now present other quantitative findings not included in previous discussions. We attempted to measure the complexity of DC bugs using four metrics: (a) the number of "re-enumerated steps" as informally defined in Section 3.1.2, (b) the patch LOC including new test suites for the corresponding bug, (c) the time to resolve (TTR), and (d) the number of discussion comments between the bug submitter and developers. The 25th percentile, median, and 75th percentile for the four metrics are (a) 7, 9, and 11 steps, (b) 44, 172, and 776 LOC, (c) 4, 14, and 48 days to resolve, (d) 12, 18, and 33 comments.

In terms of where the bugs were found, Figure 3.3r (WHR) highlights that 46% were found in deployment and 10% from failed unit tests. The rest, 44%, are not defined (could be manually found or from deployment). Some DC bugs were reported from large-scale deployments such as executions of thousands of tasks on hundreds of machines.

## 3.7 Summary

In this chapter, we have described our thorough study of DC bugs and present TaxDC, the comprehensive taxonomy of DC bugs. We study 104 real-world DC bugs from four popular distributed systems, such as, Cassandra, Hadoop MapReduce, HBase and ZooKeeper, and we categorize them into 3 different key stages: triggering, errors & failures, and fixing. Here we summarize some findings on the intricacies of DC bugs:

• Throughout the development of our target systems, new DC bugs continue to surface. Although these systems are popular, there is a lack of effective testing, verification, and analysis tools to detect DC bugs prior to deployment.

• Real-world DC bugs are hard to find because many of them linger in complex concurrent executions of *multiple* protocols. Complete systems contain many background and operational protocols beyond user-facing foreground protocols. Their concurrent interactions can be deadly.

• 63% of DC bugs surface in the presence of hardware faults such as machine crashes (and reboots), network delay and partition (timeouts), and disk errors. As faults happen, recovery protocols create more non-deterministic events concurrent with ongoing operations.

• 47% of DC bugs lead to silent failures and hence are hard to debug in production and reproduce offline.

Nevertheless, through a careful and detailed study of each bug, our results also bring fresh and positive insights:

• More than 60% of DC bugs are triggered by a *single* untimely message delivery that commits order violation or atomicity violation, with regard to other messages or computation. This finding motivates DC bug detection to focus on timing-specification inference and violation detection; it provides simple program-invariant and failure-predictor templates for DC bug detection, failure diagnosis, and run-time prevention.

• 92% of DC bugs are triggered only by a *single* untimely event. This simple but powerful finding motivates future DC bug detection, failure diagnosis and run-time prevention approaches to find, focus on, and rectify the "single culprit".

• 53% of DC bugs lead to *explicit* local or global errors. This finding motivates inferring timing specifications based on local correctness specifications, in the form of error checking already provided by developers.

• Most DC bugs are fixed through a small set of strategies. 30% are fixed by prohibiting the trig-

gering timing and another 40% by simply delaying or ignoring the untimely message, or accepting it without introducing new handling logic. This finding implies unique research opportunities for automated in-production fixing for DC bugs.

We hope our analyses in this chapter will commence more interdisciplinary actions from diverse researchers and practitioners in the areas of concurrency, fault-tolerance and distributed systems to combat DC bugs together.

# CHAPTER 4

# FLYMC: A FAST, SCALABLE, AND SYSTEMATIC SOFTWARE MODEL CHECKER

*"We have already found and fix many cases ... however it seems exist many other cases." — A comment in HB-6147*

As discussed in Section 2.2, one nemesis of checkers in detecting DC bugs is the *path explosion problem*. To tame this problem, checkers employ path *reduction algorithms*. For example, MoDist [103] and some others [95, 102] adapted the popular concept of Dynamic Partial Order Reduction (DPOR) [40, 44], for example "a message to be processed by a given node is *independent* of other concurrent messages destined to other nodes [hence, *not* need to be interleaved]." SAMC [67] also extended DPOR further. As a result, reductions significantly improve upon a naive DFS method, as shown by the "mDPOR" and "SAMC" bars on `Raft-1` in Figure 4.1.

Despite these early successes, we found that the path explosion problem is still untamed under more *complex* workloads. For example, under two or three concurrent Raft updates (`Raft-2` and `-3` workloads in Figure 4.1), the number of paths to explore still increases significantly in MoDist and SAMC. Not to mention a much more complex workload such as Paxos [66] where the path explosion is larger (*e.g.*, `Paxos-1` to `-3` workloads in Figure 4.1).

To sum up, existing checkers fail to scale under more complex distributed workloads. Yet in reality, some real-world bugs are still hidden behind complex interleavings (Section 4.5). For example, the Paxos bug in Cassandra in Figure 4.2 can only surface under a workload with three concurrent updates with 54 events in total. This kind of bugs will take weeks to surface with existing checkers, wasting testing compute resources and delaying bug finding and fixes. For all the reasons above, to find DC bugs, some checkers mix their algorithms with *random* walks [103] or *manual* checkpoints [54], hoping to faster reach "interesting" interleavings that would lead to DC bugs. However, concurrency testing process becomes unsystematic – the random and manual

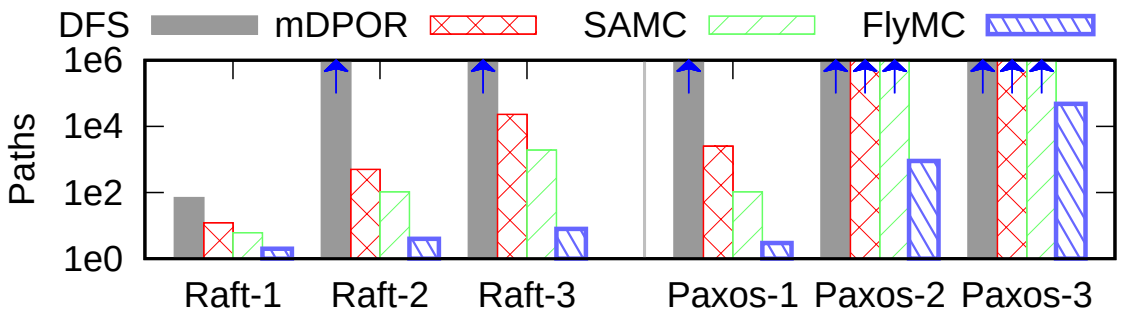



Figure 4.1: **Checkers scalability.** *The x-axis represents the tested protocols (Raft or Paxos) with 1 to 3 concurrent updates. The log-scaled y-axis represents the number of paths to exhaust the search space (i.e., the path explosion). Compared to our checker,* FLYMC*, current checkers do not scale well under more complex workloads. "mDPOR" stands for* MODIST*'s DPOR rule.* **"↑"** *indicates incomplete path exploration.*

approaches lead to poorer coverage than a systematic coverage of all states relevant to observable events. Thus, a conventional view believes that testing with model checking can only exhaustively search a small space and randomized approaches non-exhaustively search a large space. We show that is field of research, it is feasible to stay systematic and be fast.

In this chapter, we present FLYMC, a fast, scalable, and systematic software/stateless model checker that covers all states relevant to observable events for testing distributed systems implementations. FLYMC achieves scalability by highly leveraging the internal properties of distributed systems with two reduction and one prioritization algorithms. The reduction algorithms reduce unnecessary interleavings (redundant paths) that would lead to the same states already explored before while the prioritization algorithm prioritizes interleavings that would reach corner cases faster. We illustrate the FLYMC's algorithms as follows:

*(1) Communication and state symmetry:* Common in cloud systems, many nodes have the same role (*e.g.*, follower nodes, data nodes). The state transitions of such symmetrical nodes usually depend solely on the order and content of messages, irrespective of the node IDs/addresses. Thus, FLYMC reduces different paths that represent the same symmetrical communication or state transition into a single path.

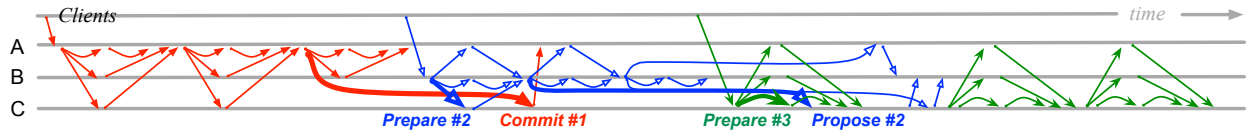



Figure 4.2: **A complex DC bug in Cassandra Paxos (CA-6023).** *As shown in Figure 3.4, this bug requires three concurrent Paxos updates and only surfaces with the two flips (the prepare message with ballot-2 must be enabled before the commit with ballot-1 and the prepare with ballot-3 before the propose with ballot-2) happening within all the possible flips of the 54 events, resulting in data inconsistency.*

*(2) Events independency:* While state symmetry significantly omits symmetrical paths, many events must still be permuted within the non-symmetrical paths. FLYMC is able to identify a large number of event independencies that can be leveraged to alleviate a wasteful reordering. For example, FLYMC automatically marks concurrent messages that update disjoint sets of variables as independent. FLYMC can also find independency among crash-related events.

*(3) Parallel flips:* While the methods above reduce message interleavings to every node, in aggregate many flips (reordering of events) must still be done across all the nodes. The problem is that in existing checkers, only one pair of events is flipped (reordered) at a time. To speed this up, parallel flips perform simultaneous reorderings of concurrent messages across different nodes to quickly reach hard-to-reach corner cases.

Finally, not only path reduction but wall-clock speed also matters. Existing checkers must wait a non-negligible amount of time in between every pair of enabled events for two purposes: (1) to prevent concurrency issue within itself and (2) to wait for new updated states from the target system. The wait time is reasonable under simple workloads, but significantly hurts the aggregate testing time of complex workloads. In FLYMC, we optimize this design with local ordering enforcement and state transition caching.

Collectively, the algorithms make FLYMC faster 16× on average (up to 78×) than other state-of-the-art systematic and random-based approaches, and the design optimizations improve it to 28× (up to 158×). FLYMC is integrated with 8 widely-used systems, the largest number of integration that we are aware of. Within all of these systems, we model checked 10 protocol imple-

mentations (Paxos, Raft, etc.), successfully reproduced 12 old bugs, and found 10 new critical DC bugs, all confirmed by the developers and all were done in a systematic way *without* random walks or manual checkpoints.

In the following sections, we will present the highly scalable checker algorithms (Section 4.1), the checker design that is backed with static analysis (Section 4.2), the checker's optimizations (Section 4.3), and lastly, the implementations and the detailed evaluations of the checker's effectiveness (Section 4.4-4.5).

## 4.1 FLYMC Algorithms

By incorporating the properties of distributed systems, we equip FLYMC with two reduction algorithms: communication and state symmetry (Section 4.1.1) and events independency (Section 4.1.2); and one prioritization algorithm: parallel flips (Section 4.1.3). The two reduction algorithms reduce unnecessary interleavings (redundant paths) that would lead to the same states that have been explored before. While the prioritization algorithm prioritizes interleavings that would reach corner cases faster.

Throughout this section, we describe each of the algorithms in the following format: (a) the specific path explosion problem being solved, (b) the intuition for the reduction or prioritization, (c) the algorithm in a high-level description, and (d) a comparison to existing solutions. Later in Section 4.2, we discuss the intricacies of implementing these algorithms correctly and how our static analyses support can help developers in this regard.

### *4.1.1 Communication and State Symmetry*

• **PROBLEM:** Let us imagine a simple communication in Figure 4.3a where message $k$ triggers $l$, $x$ triggers $y$, and $k$ and $x$ are messages of the same type (*e.g.*, a write request). Figures 4.3b and 4.3c show two possible reordered paths $klxy$ and $xykl$. While these paths seem to be different, their communication structures in Figures 4.3b-c hint at a possibility for symmetrical reduction.

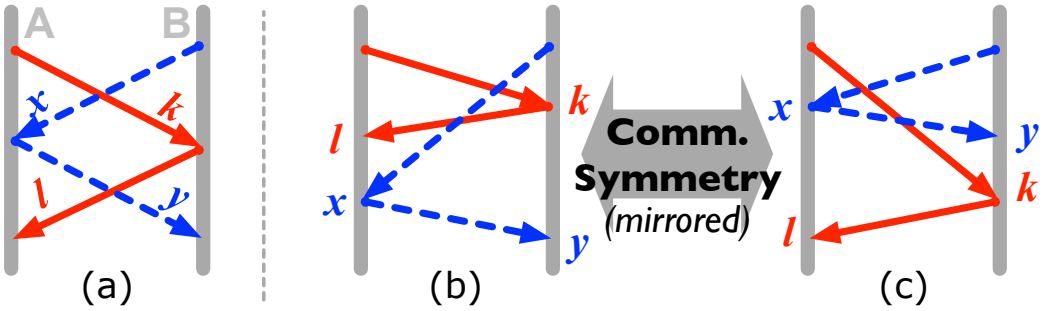



Figure 4.3: **Communication symmetry.** *The figure is explained in the "Problem" part of Section 4.1.1.*

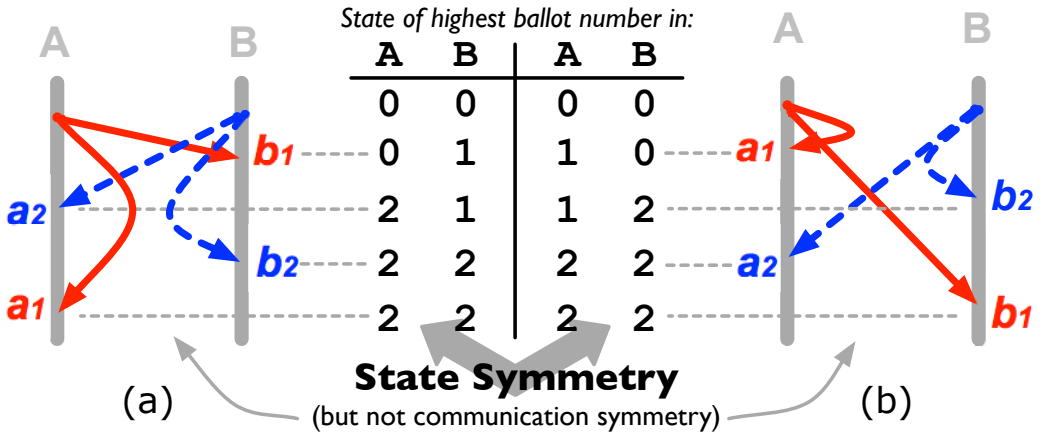



Figure 4.4: **State symmetry.** *The figure is explained in the "Intuition" part of Section 4.1.1.*

A method to implement the symmetrical reduction in LC literature is to *abstract* the system property [30, 31, 97]. Applying this to distributed systems, we initially attempted to abstract only the communication structure, specifically by abstracting the sender and destination node IDs (*e.g.*, IP addresses) to a canonical receiving order; for example in Figure 4.3b, as node $B$ is the first to receive, its node ID is abstracted to node "1" (*e.g.*, $k_{A\to B}$ becomes $k_{2\to 1}$). Similarly in Figure 4.3c, as node $A$ is the first to receive, its node ID is abstracted to node "1" (*e.g.*, $x_{B\to A}$ becomes $x_{2\to 1}$), hence the two figures exhibit a communication symmetry as $k$ and $x$ are messages of the same type from node "2" to "1".

Unfortunately, this approach is not always effective because most messages carry a unique content. For example, in Paxos, messages $k$ and $x$ carry different ballot numbers, hence cannot be

treated the same. Thus, while the communication structures (the arrows) in Figures 4.3b and 4.3c look symmetrical, abstracting only the messages does *not* lead to a massive reduction.

• **INTUITION:** Fortunately, in many cloud systems, many nodes have the same role (*e.g.*, follower nodes, data nodes) although their node IDs are different. Furthermore, the state transitions of such symmetrical nodes usually depend solely on the order and content of the messages, irrespective of the sending/receiving node IDs.

To illustrate this, let us consider the two communication structures in Figures 4.4a-b, which represent the first phase of a (much simplified) Paxos implementation with two concurrent updates (solid and dashed lines). Node $A$ broadcasts its prepare messages (the solid lines), $a_1$ to itself and $b_1$ to node $B$, with "$_1$" representing a ballot number 1. Similarly, node $B$ broadcasts $b_2$ to itself and $a_2$ to node $A$ with ballot number 2 (dashed lines).

If we compare the two communication structures in Figures 4.4a-b, they are *not* symmetrical, unlike the previous example in Figure 4.3. But let's analyze the state transition of every node, such as the highest ballot number the node has received, as shown in the middle table of Figure 4.4. In this Paxos example, every node only accepts a higher ballot and discards a new lower one, hence the node prepare status monotonically increases (*e.g.*, `1` to `2`). In the left ordering, $b_1a_2b_2a_1$ in Figure 4.4a, node $A$'s state transition is `00222` and $B$'s is `01122`. In the ordering on the right, $a_1b_2a_2b_1$, the *state transition is symmetrical* (mirrored), `01122` in $A$ and `00222` in $B$.

To sum up, while the two paths do not exhibit communication symmetry (Figures 4.4a-b), their *state transitions are symmetrical* (the middle table). Thus, *state symmetry* can be effective for path pruning (*e.g.*, if $b_1a_2b_2a_1$ is already explored, then $a_1b_2a_2b_1$ is redundant).

• **ALGORITHM:** To implement symmetry, first, we keep a history of state-event transitions (Section 2.2) that have been exercised in the past, in the following format: `Si`+$e_j$→`Sj` where "`S`" denotes the global state (*i.e.*, collection of per-node states) and $e_j$ is the next enabled event when the system state is at `Si` that transitions the global state to `Sj`. For example, in Figure 4.4a, after the first enabled event $b_1$, we record `00`+$1$→`01`.

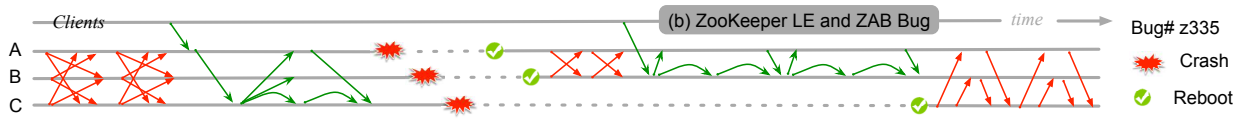



Figure 4.5: **A ZooKeeper bug with complex timings of multiple crashes (ZK-335).** *The bug is referenced in Section 4.1.2 and Section 4.5.1. This bug requires 46 events including 3 crash and 3 reboot events, along with two incoming transactions, a complex concurrency between the ZooKeeper atomic broadcast (ZAB) and leader election (LE) protocols.*

In addition, we keep a history of $\{\texttt{absState+}absEv\}$ transitions where `absState` denotes the abstracted global state (in alphabetical order) that excludes the node IDs for symmetrical nodes such as datanodes (and similarly $absEv$ for events). Using the example in Figure 4.4a, the first event will generate $\{\texttt{00+}\mathit{1}\}$ where `00` represents the abstracted state of datanodes $A$ and $B$ (with just the highest ballot numbers, excluding the node IDs) and *1* represents the abstracted $a_1$ message (with the source and destination node IDs removed). Subsequently, we record $\{\texttt{01+}\mathit{2}\}$, $\{\texttt{12+}\mathit{2}\}$, and $\{\texttt{22+}\mathit{1}\}$ to the history. Important to note that state `12` is from the alphabetically ordered state `21`; that is, symmetry implementation requires alphabetical/numerical sorting.

With this history, the second ordering $a_1b_2a_2b_1$ in Figure 4.4b will be marked symmetrical; when $a_1$ is to be enabled (abstracted to *+1*) when the system is at state `00`, a historical match $\{\texttt{00+}\mathit{1}\}$ will be found. Similarly, for $b_2$ (abstracted to *+2*) when the system is at state `01`, a match $\{\texttt{01+}\mathit{2}\}$ will be found. One caveat is that state symmetry works less effective in earlier paths as the history is still being built up, but after a few initial paths, the "cache hit rate" increases significantly (more in Section 4.3).

• **COMPARISON:** In classical (stateful) model checking, symmetry is commonly used, *e.g.*, for symmetrical processors [30, 97]. In distributed checkers, we found none that employs symmetry [51, 54, 63, 67, 95, 102, 103], except SAMC [67]. However, SAMC only uses symmetry for reducing unnecessary crash timings, but not for concurrent messages. FLYMC's symmetry is more powerful as it also generalizes for crash timings. More specifically, a crash is abstracted as a crash event targeted to a particular node; for example, $\{\texttt{12+}\hat{\mathit{2}}\}$ implies a crash injected at the node with ballot number 2 (regardless of the datanode IDs).

### *4.1.2 Events Independency*

• **PROBLEM:** While state symmetry significantly omits symmetrical paths, there are many other events to reorder within the non-symmetrical paths. For example, if four messages $a_1...a_4$ of different types are concurrent to node $A$, the permutation will lead to $4!$ times more paths. As some concurrent messages to every node must still be reordered, more reduction is needed.

• **INTUITION:** In this context FLYMC adapts the concept of DPOR's "independency" (aka. commutativity) as mentioned in the introduction. In DPOR, two events $e_1$ and $e_2$ are independent if $S_i$+$e_1$+$e_2$→$S_j$ and $S_i$+$e_2$+$e_1$→$S_j$. That is, if $e_1e_2$ or $e_2e_1$ result in the same global state transition from $S_i$ to $S_j$, the pair of events $e_1$ and $e_2$ do not have to be flipped when the system is at $S_i$, hence reducing the number of paths to explore. An example of independency in distributed systems is when many concurrent messages (to a destination node) update different variables. For example, in some distributed systems such as ZooKeeper, the atomic broadcast protocol might be running concurrently with the leader election protocol (because of a crashed node), but some of the messages in these two protocols do not update the same variables (when the system is at a specific state $S_i$), hence they are unnecessary to be flipped.

• **ALGORITHM:** While the concept of DPOR/independency arose from stateful model checkers (with known state transitions) [18, 40, 44, 88], adapting it to stateless distributed checkers is not straightforward – how can a checker has a *prior* knowledge that $S_i$+$e_1e_2$ and +$e_2e_1$ would lead to the same future state $S_j$ *before* exercising the events? For this, chelps developers identify *disjoint updates* ahead of time with the static analyses (more details in Section 4.2.1).

Essentially, for every message $n_i$ to a node $N$, our static analyses builds the live `readSet` and `updateSet`, a set of to-be-read and -updated variables, within the flow of processing $n_i$ at $N$'s current state. That is, our approach incorporates the fact that $n_i$'s read and update sets can change as node $N$ transitions across different states. Given such an information, two messages $n_i$ and $n_j$ to a node $N$ are marked independent if $n_i$'s `readSet` and `updateSet` do not overlap with $n_j$'s `updateSet` at the current state $S_i$, and vice versa. In addition, if the `updateSets` of two messages

intersect completely and all the variables in the sets are in/decremented by one (*e.g.*, a common acknowledgment increment "`ack++`" in distributed systems), then the two messages are marked independent/commutative.

Not only reducing unnecessary message interleavings, a scalable checker must reduce unnecessary crash injections at different timings. 50% of DC bugs can only surface with at least one crash injection and 12% require at least two crash events at specific timings [68] (*e.g.*, the ZooKeeper bug in Figure 4.5), which exacerbates further the path explosion problem (imagine different fault timings such as $..a_1a_2\not{A}..$, $..a_1\not{A}..$, $..a_2\not{A}..$, where "$\not{A}$" denotes crashing of node $A$). Thus, another uniqueness of FLYMC's adaptation of independency is building the sets above for crash events. For example, if a follower node is crashed ($\not{B}$) and the leader node $A$ reacts by reducing the live-nodes count (*e.g.*, `liveNodes--`), then $\not{B}$'s `updateSet` will include $A$'s `liveNodes` variable.

• **COMPARISON:** Prior checkers adopted DPOR's independency, but did not do so in a full extent, hence are not scalable under complex interleavings. For example, MODIST [103, Section 3.6], CrystalBall [102, Section 2.2] and dBug [95, Section 2] only adopted DPOR with the following rule: "a message to be processed by a given node is independent of other concurrent messages destined to other nodes," but because they are *black-box* checkers that do not analyze the target source code, they cannot find more independencies. On the other hand, being a *white-box* checker, FLYMCexploits access to source code in today's DevOps-based cloud development where developers are testers and vice versa [71].

SAMC is another example of a white-box checker, but because its algorithms are entirely written manually (no static analyses support) SAMC only introduces cautious and rigid reduction algorithms less powerful than FLYMC's (note that in FLYMC's the content of the sets mentioned above will be automatically constructed specifically for each target system). For example, FLYMC generalizes a discarded message ([67, Section 3.3.1]) as an empty `updateSet`, such a message automatically does not conflict with any other messages, hence not need to be reordered. As another example, a crash that does not lead to new messages (*e.g.*, a quorum is still maintained after

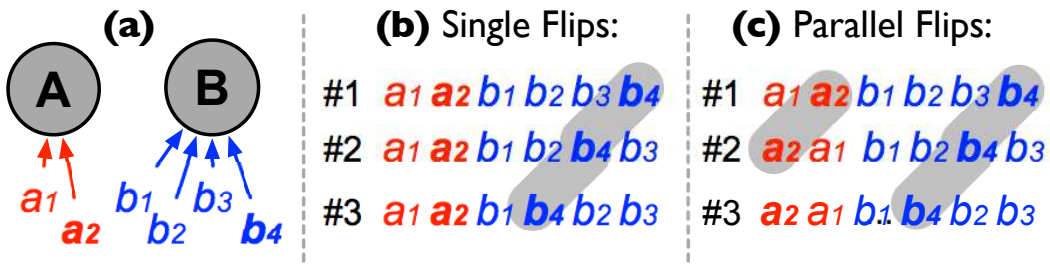



Figure 4.6: **Parallel flips.** *Figures (a+b) and (c) are explained in the "Problem" and "Algorithm" segments of Section 4.1.3, respectively.*

crashing a follower node $\not{B}$) will not be interleaved with all the outstanding messages [67, Section 3.3.2] as FLYMC automatically identifies that the crash event $\not{B}$'s `updateSet` (*e.g.*, `liveNodes` in the leader node) does not conflict with `updateSets` of in-flight messages.

### *4.1.3 Parallel Flips*

• **PROBLEM:** While our previous method reduces message reordering to every node, in aggregate many flips must still be done across all the nodes. The problem is that in existing checkers, to create a new reordered path, only one pair of events is flipped at a time. For example, in Figure 4.6a, two concurrent messages $a_1$ and $a_2$ are in transit to node $A$ and four messages $b_1 ... b_4$ to node $B$. Figure 4.6b illustrates how existing approaches flip only one non-independent pair of events at a time; for example, after path #1 $a_1a_2b_1b_2b_3b_4$, the next path #2 is created by sliding $b_4$ before $b_3$, then a subsequent path #3 with $b_4$ before $b_2$, and so on. Now, let us suppose that a bug is induced by $a_2a_1$ ordering (*i.e.*, $a_2$ must be enabled before $a_1$). In the standard approach above, it will take $4!$ reorderings (of the four messages to $B$) before we have the chance to flip $a_2$ before $a_1$.

• **INTUITION:** We observed such pattern when analyzing our bug benchmarks. For example, to hit the Paxos bug in Figure 4.2, `Prepare#2` message must be enabled before `Commit#1`, both to node $C$, but there are 8 earlier in-flight messages to other nodes that must be reordered. Worse after that, `Prepare#3` message must arrive before `Propose#2`, but there are 5 earlier messages to flip. Thus, the bug-inducing flips are not exercised early.

This problem motivates us to introduce *parallel flips*. That is, rather than making one flip at a time, parallel flips of pairs of events are allowed. Parallel flips also bode well with a typical developers' view that mature cloud systems are generally robust in the "first order" (under common interleavings) [14] but simultaneous "uncommon" interleavings across all the nodes may find bugs faster.

• **ALGORITHM:** For the next to-explore path, we flip a pair of two messages in *every* node, hence $N$ simultaneous flips *across* all the $N$ nodes (but we do not perform multiple flips within a node). For example, in Figure 4.6c, after executing path #1 $a_1 a_2 b_1 ... b_4$, in path #2 we make *both* $a_2 a_1$ and $b_4 b_3$ flips. This is permissible because the in-flight messages to node $A$ are independent of those to node $B$ (per our DPOR adoption in Section 4.1.2). If no parallel flips are possible, we revert back to single flips (*e.g.*, only $b_4 b_2$ flip in path #3).

We emphasize that parallel flips is a prioritization algorithm rather than a reduction algorithm. That is, this algorithm helps developers to unearth bugs faster but does not soundly reduce the state space, i.e., it may miss covering some unique states. Later when evaluating coverage completeness (Section 4.5.3), parallel flips are not included.

• **COMPARISON:** We are not aware of any distributed checkers that employ an algorithm such as parallel flips. However, in the multithreaded checkers [47] and the software testing literature [20, 49], we found that our approach is in spirit similar to "genetic algorithms" and "branch flipping" where multiple branch constraints are flipped simultaneously to cover more corner cases faster.

## 4.2 FLYMC Static Analyses and Design Challenges

There are several challenges in applying FLYMC algorithms correctly in the context of distributed systems. First, we describe FLYMC static analyses that automatically extract the knowledge about the target system (Section 4.2.1). Next, we describe the challenges in applying the FLYMC algorithms (Section 4.2.2).

### *4.2.1 Static Analyses Support*

While FLYMC's algorithms are generic, the details (*e.g.*, the `if-else` predicates for reduction) are specific to a target system. Furthermore, the required predicates can become quite complex, which makes it harder for developers to derive them manually. For this reason, we provide static analyses support in FLYMC, which automatically builds the required predicates from simple annotations provided by the developers. For example, the static analyses automatically build `readSet`, `updateSet`, `sendSet` and `diskSet` (Section 4.1, Section 4.2.2) containing variable names specific to the target system implementation. Below we describe the input and output formats.

• **INPUT (ANNOTATION):** To use FLYMC's static analyses, developers only need to annotate a few data structures: (a) node states, (b) messages, and (c) crash handling paths. Annotating node states that matter (*e.g.*, ballot, key, value) is a common practice [87, 103], for example:

```
public class Commit {
  ...
  @FlyMCNodeState
  public final UUID ballot;
  ...
}
```

Annotating message class declarations such as "`MessageIn`" in Cassandra is relatively simple (note that we only need to annotate the class declarations, but not every instantiation, hence a light annotation). Crash handling paths are typically in the `catch` blocks of failed network IOs, for example:

```
try{
    ...
    binaryOutput.writeRecord(quorumPacket,...);
    ...
} catch { @crashHandlingPath ... }
```

In addition, our static analyses also maintains a dictionary of disk IO library calls. On average, the annotation is only 19 LOC per target system that we have studied.

• **OUTPUT:** The output of the analysis is all the variable sets mentioned above, along with the symbolic paths. For example, for Cassandra, the analysis outputs are as follows:

```
(A) if (m.type == "PROP" && m.ballot > n.ballot)
      updateSet = n.key, n.value, n.ballot
      readSet   = n.ballot
(B) if (m.type == "PROP_RESP" && m.resp == true &&
      n.proposeCounter < majority)
     updateSet = n.proposeCounter
     readSet   = --
(C) ...
```

With this output, we can track the relationship of every two concurrent messages $n_i$ to $n_j$ to node $N$. For example, if both messages are type `A`, they conflict with each other and must be reordered. However, if $n_i$ is of type `A` while $n_j$ is of type `B`, they exhibit disjoint updates and therefore, do not need to be reordered (but they are independent only at specific states that satisfy the `if`-predicates above).

To create such an output given the input annotation, our static analyses performs basic data- and control-flow analyses. Our static analyses do not cover multi-variable correlation and pointer/heap analysis (as not needed in our target cloud systems so far; *e.g.*, a message simply arrives, gets processed, and then is deallocated).

### *4.2.2 Design Challenges*

• **STATE SYMMETRY:** In reality, not only one variable (*e.g.*, ballot number) is included in the abstracted state, which then raises the question of which variables should be included/excluded in the abstracted information. For example, if the protocol processes the sender IDs (node addresses)

of the messages, then excluding sender IDs from the abstracted event is not safe, as this can incorrectly skip unique event reorderings. Thus, for state symmetry, our static analyses outputs a list of message variables that state transitions depend on, hence cannot be abstracted (excluded). For example, for Cassandra Paxos, neither the sender nor destination datanode IDs are used by the protocol, hence can be safely excluded from the abstracted information.

• **EVENT INDEPENDENCE:** We address two challenges in implementing event independence.

First, as we target storage-backed distributed systems, two messages, $n_i$ and $n_j$ to node $N$, might modify two different variables that perhaps will eventually be logged to the same on-disk file. It is not safe to consider them independent as the same file is updated but potentially in different orders. Thus, the two disjoint messages are truly independent only if they are not logged to the same file, which our static analyses tracks (Section 4.2.1).

The second challenge is similar but more subtle. In distributed settings, reordering of messages to one node cannot be seen as a local impact only, as an arriving message can trigger *new* messages. This non-local impact must be put into consideration. For example, let us consider two messages $a_1$ and $a_2$ concurrently arrive at node $A$ whose local state is {`x=0,y=0`}. Now, let us suppose $a_1$ makes `x=1` and $a_2$ makes `y=2`. Here, the two messages seem to be disjoint. However, if after processing each message, node $A$ *sends* its state {`x,y`} to other nodes (*e.g.*, $B$), then the two messages are actually *not* independent. Making them independent would lead to an unsafe reduction. Let us consider the following sequence:

1) $A$'s state is `x=0,y=0`

2) $A$ receives $a_1$, so `x=1`

3) $A$ sends `x=1,y=0` to $B$

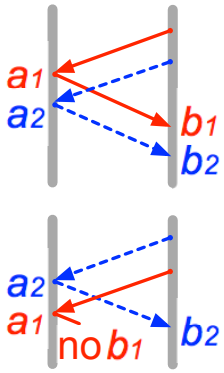
4) $A$ receives $a_2$, hence `y=2`

5) $A$ sends `x=1,y=2` to $B$

The example scenario shows $a_1$ is enabled before $a_2$, (2) before (4). If we (incorrectly) declare them as independent, $a_2a_1$ ordering will be skipped, therefore we will *never* see {`y=2,x=0`} sent

to $B$. If node $B$ has a logic such as "`if (y==2 && x==0) panic()`", then we will *miss* this $a_2a_1$-induced bug. For this reason, in addition to `readSet` and `updateSet`, we keep track of the `sendSet` (Section 4.2.1), the variables that are sent out after a message is processed. In the example, because $a_1$'s and $a_2$'s `sendSets` overlap with their `updateSets` (*i.e.*, `x,y`), $a_1$ and $a_2$ are not independent.

• **PARALLEL FLIPS:** We only allow parallel flips if none of the events within the flips are causally-dependent on one another (*i.e.*, exhibit a happens-before relationship). For example, let us consider $a_1a_2b_1b_2$ in the first case on the right figure, where $b_1$ is causally-dependent on (happens after) $a_1$, and $b_2$ on $a_2$. If we carelessly make the two flips to $a_2a_1b_2b_1$, it is possible that $b_1$ will *never* happen (as shown in the lower figure) because the new ordering $a_2a_1$ that node $A$ receives does not generate $b_1$. In this case, this new path will make `c` *hang*. Thus for correctness, `c`'s parallel flips are backed with happens-before analysis via vector clocks.




Parallel flips do not provide a sound reduction of state space in general, and it is tricky to identify conditions where they might. Thus, we treat parallel flips as a prioritization heuristic to reach corner cases faster, but it is based on a well-grounded intuition (Section 4.1.3) as opposed to randomness or manual checkpoints.

## 4.3 FLYMC Design Optimizations

In wall-clock time, an execution of *one* path can take 8-40 seconds [67, 95]. Since one of the checker's goals is to quickly unearth bugs, thus, per-path wall-clock speed matters. Below we describe our solutions to the bottlenecks.

• **LOCAL ORDERING ENFORCEMENT:** In a path execution, stand-alone checkers that intercept events at the application layer with "hooks" (*e.g.*, dBug [95], SAMC [67], FLYMC) must wait a non-negligible amount of time (*e.g.*, 300 ms) before enabling the next event, for two purposes: to

prevent concurrency issues within itself and to wait for new updated state from the target system.

To illustrate the former, consider two concurrent incoming messages $a_1$ and $a_2$ to node $A$, and the checker's server decides to enable $a_1$ then $a_2$. If the wait time is removed between the two actions, the probability that node $A$ *accidentally* processes $a_2$ before $a_1$ increases. This is because `enable(a1)` and `enable(a2)` actions themselves are *concurrent* messages from the checker's server to node $A$ whose timings are not controlled. Thus, although $a_1 a_2$ is "enforced" at the checker server side, when the enabling actions arrive in node $A$, there is a small probability that $a_2$ runs first before $a_1$. This wait time is too expensive for such rare cases. To remove it, we enhance `c`'s interposition mechanism at the target system side to enforce local action ordering; for example, enabling $a_2$ includes information about the previous enabled event, $a_1$, such that at node $A$'s side, $a_2$ waits for $a_1$, if needed.

**STATE-EVENT CACHING:** Although the wait time has been removed, FLYMC must perform "history tracking" for collecting past state-event transitions `Si+`$e_j$`→Sj`. Here, after enabling $e_j$ and before enabling the next event $e_k$, cmust collect `Sj` from the target system, another expensive round-trip time. To optimize this, we use the state-event history as a cache. That is, if $e_j$ is to be enabled at `Si` and the state transition `Si+`$e_j$ already exists in the history, then no wait is needed between enabling $e_j$ and $e_k$. In this case, FLYMC will automatically change its view state of the target system to `Sj`. This strategy is highly effective; the "cache hit rate" reaches 90% quickly after 35 paths explored.

In summary, a checker itself is a complex system with many opportunities for optimization. Our optimizations have delivered further speed-ups to quickly detect DC bugs (Section 4.5.1).

## 4.4 Implementation and Integration

FLYMC is implemented in around 10 KLOC which includes the fault injection, deterministic replay, interposition hooks, path execution/history management, state caching/snapshotting, etc.. The three core algorithms described throughout Section 4.1 are however only 2420 LOC and the

hooks to a target system are only 147 LOC on average. The static analyses support is written in 1799 LOC in Eclipse AST Parser for Java programs which covers many of our target systems including Cassandra, ZooKeeper, and Hadoop. For LogCabin Raft and Kudu Raft (in C++) and Spark (in Scala), we manually build the sets (Section 4.2.1). We leave porting to other language front-end parsers as future work.

FLYMC has been integrated with 8 popular systems: Cassandra [65], Ethereum Blockchain [9], Hadoop [3], Kudu [11], Raft LogCabin [12, 82], Spark [106], ZooKeeper [58], and a 2-year old production system "$X$" of a large company (to the best of our knowledge, the largest checker integration compared to prior works). Within these systems, we model checked 10 unique protocol implementations, such as Cassandra Paxos, ZooKeeper leader election and atomic broadcast, Hadoop task management, Kudu Raft, LogCabin Raft leader election and snapshot, Spark core, Ethereum fast synchronization, and "$X$" leader election.

## 4.5 Evaluation

We now evaluate FLYMC in terms of speed in reproducing DC bugs (Section 4.5.1), scalability (Section 4.5.2), coverage completeness (Section 4.5.3), per-algorithm effectiveness (Section 4.5.4), and effectiveness in finding new bugs (Section 4.5.5).

**BUG BENCHMARKS:** A popular way to evaluate a checker is how fast it can reproduce a DC bug given the corresponding workload. Table 4.1 shows our bug benchmarks, including the number of events to reproduce the bugs (*i.e.*, the bug "depth"), crashes, and reboots. Most papers did not report bug depths [63, 95, 103], although it is important for scalability evaluation.

**TECHNIQUES COMPARED:** We have exhaustively compared FLYMC against six existing solutions as listed in Table 4.2,: a purely random technique (`Rand`), two systematic techniques (`m-DP` and `SAMC`), and three hybrid systematic + random + bounded techniques (`b-DP`, `r-DP`, `br-DP`). The last category highlights how current approaches incorporate random and bounded-depth exploration to reach bugs faster.

| BugName | Issue# | #Ev | #Cr | #Rb | Protocols |
|---|---|---|---|---|---|
| CASS-1 | CA-6023 | 54 | – | – | Paxos |
| CASS-2 | CA-6013 | 30 | – | – | Paxos |
| CASS-3 | CA-5925 | 15 | – | – | Paxos |
| ZOOK-1 | ZK-335 | 46 | 3 | 3 | LE, AB |
| ZOOK-2 | ZK-790 | 39 | 1 | 1 | LE |
| ZOOK-3 | ZK-1419 | 41 | 3 | 3 | LE |
| ZOOK-4 | ZK-1492 | 24 | 1 | – | LE |
| SPRK-1 | SP-19263 | 42 | – | – | Spark Core |
| SPRK-2 | SP-15262 | 23 | – | – | Spark Core |
| MAPR-1 | MR-5505 | 36 | 1 | 1 | TA |
| RAFT-1 | RA-174 | 21 | 2 | 2 | LE, Snapshot |
| ETHM-1 | ET-15138 | 12 | 1 | 1 | Fast Sync |

Table 4.1: **Bug benchmarks (complex DC bugs).** *The table lists DC bugs used to benchmark checkers scalability. In the first column: "CASS" represents Cassandra, "ZOOK" ZooKeeper, "SPRK" Spark, "MAPR" Hadoop MapReduce, "RAFT" Raft LogCabin, and "ETHM" Ethereum BlockChain. For the Protocols column: "LE" stands for leader election, "AB" atomic broadcast, and "TA" Task Assignment. "#Ev", "#Cr" and "#Rb" stands for #Events, #Crashes and #Reboots that interleave to reach the bugs.*

We do not compare with DEMETER [54] and LMC [51] as they mainly reduce local thread interleavings in the context of reducing global interleavings (Section 2.3). We also do not show the results of depth-first-search (as used in MACEMC [103]) as it is extremely unscalable [103, Fig.

| Label | Technique Description |
|---|---|
| | *Systematic exploration techniques:* |
| m-DP | MODIST' systematic DPOR reduction rule [103, Section 3.6] as discussed in Section 4.1 (comparison segments). Note that this reduction is also used in other checkers such as dBug [95, Section 2] and CrystalBall [102, Section 2.2]. |
| SAMC | SAMC reduction algorithms [67, Section 3.3] as discussed in the comparison segments of Section 4.1. |
| | *Hybrid systematic/random/bounded techniques:* |
| b-DP | MODIST' bounded+DPOR rule [103, Section 3.6] – run DPOR evaluation up until certain depth (*i.e.*, #events). |
| r-DP | MODIST's random+DPOR rule [103, Section 4.5] – execute random path on every 50 paths and then use DPOR to evaluate the path. |
| br-DP | Combination of the last two approaches above (bounded+random+DPOR). |
| | *Random techniques:* |
| Rand | A purely random exploration. |

Table 4.2: **Techniques comparison.** *The table lists all the techniques compared against* FLYMC.

9].

**PERFORMANCE METRICS:** The primary metrics of our evaluation are the numbers of **(1)** explored paths to hit a bug, **(2)** total wall-clock time to hit a bug, **(3)** explored paths to exhaust the entire state space of a given workload, and lastly, **(4)** unique global states covered over time. Another metric other prior checkers use is "the number of unique states explored up to a certain time" (the higher the better) [54, 95, 103] which is appropriate if only independence algorithms are used. But if symmetry algorithms are employed, such a metric is no longer accurate as multiple "unique" states can actually be symmetrical (*i.e.*, higher does not imply better).

**EVALUATION SCALE:** Our extensive evaluation exercised over 200,000 paths (across all compared techniques) and used more than 130 machine days. We use Emulab "d430" machines [8] and Chameleon "compute_haswell" machines [7].

### *4.5.1 Speed*

Figure 4.7a (in log scale) shows the *number of paths* explored to hit each of the bugs in Table 4.1 across different methods listed in Table 4.2. Figure 4.7b shows the *wall-clock time*. For readability, in each bar group, we put FLYMC bar in the middle (striped blue), systematic approaches on the left (`m-DP` and `SAMC` in patterned bars), hybrid and random on the right (`br-DP`, `r-DP`, `b-DP`, and `Rand` in solid colors). The horizontal blue markers are the height of FLYMC bars.

This evaluation method reflects a checker's speed in helping developers to reproduce hidden DC bugs. So, suppose the users supply a workload that non-deterministically (occasionally) fails, the checker then should find the buggy interleaving(s) such that the developers can easily (and deterministically) replay them. Note that some methods fail to find the buggy paths after exploring 10,000 paths (marked with ↑ in Figure 4.7a). We stop at 10,000 paths to prioritize other evaluations. From the figure, we make the following observations:

**(a)** Within the systematic group, MODIST' DPOR (`m-DP`) is not effective for 5 of the bugs (`CASS-1`, `ZOOK-1`, `SPRK-1`, `MAPR-1`, and `RAFT-1`), which is due to the limitations of black-box meth-

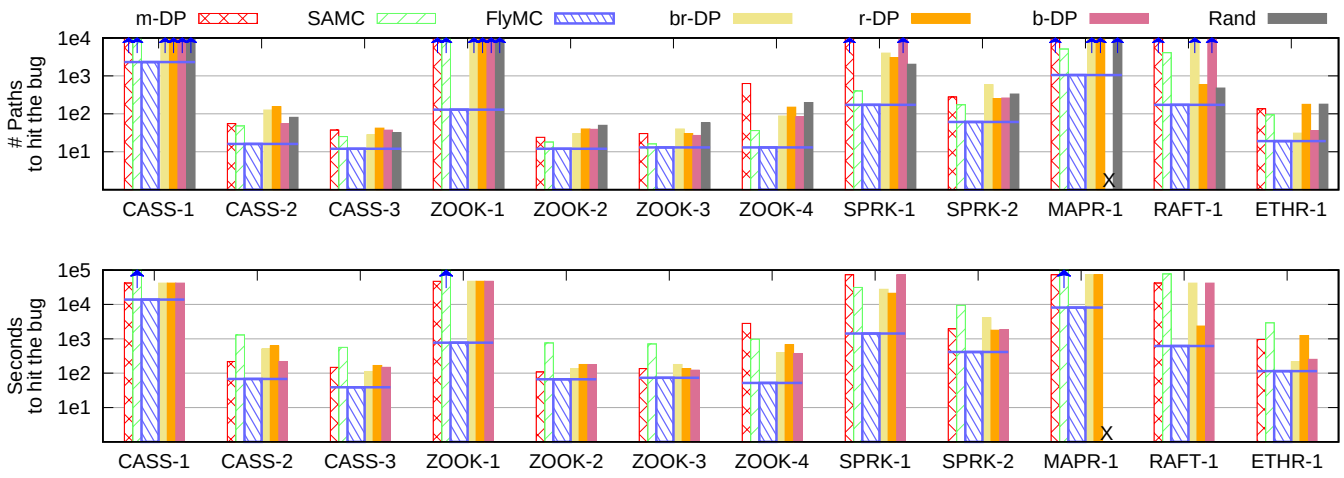



Figure 4.7: **FLYMC speed.** *The top and bottom figures show the number of paths to explore (in log scale) and the wall-clock time, respectively, to find the buggy paths that make the bugs surface, as explained in Section 4.5.1. For the legend labels, please see Table 4.2.* **"↑"** *implies that the bug is not reached after 10,000 paths.* `Rand` *numbers are averaged from five tries.*

ods in pruning redundant paths.

**(b)** `SAMC` is faster than `m-DP` up to 25×. However, for two of the bugs (`CASS-1` and `ZOOK-1`) SAMC cannot reach them within 10,000 paths and for the other two cases (`MAPR-1` and `RAFT-1`) `SAMC` is relatively slow. Again, this happens because `SAMC` does not have any static analysis support. Instead, developers need to manually analyze and implement their own reduction algorithms by following the SAMC principles. Therefore, in practice, SAMC might miss some potential reductions. Furthermore, it mainly focuses on reducing unnecessary crash timings, hence does not scale for workloads with many concurrent messages such as in CA-6023 (Figure 4.2).

**(c)** Bounded DPOR (`b-DP`) approximately has the same speed as random-hybrid ones. Interestingly, for bug `MAPR-1`, the exploration completes, *but* the bug was not found ("**X**" in the figure). This shows a weakness of bounding the number of events to flip. Note that with bounded+random (`br-DP`) the randomness might shuffle the critical events first.

**(d)** Random is random. `Rand` is the slowest method for 3 of the bugs, but it is faster than `m-DP` and `SAMC` in 4 and 1 other cases, respectively. In the latter cases, the degree of concurrency is low (*e.g.*, to enable an event, random only needs to pick 1 out of 3 outstanding events), hence

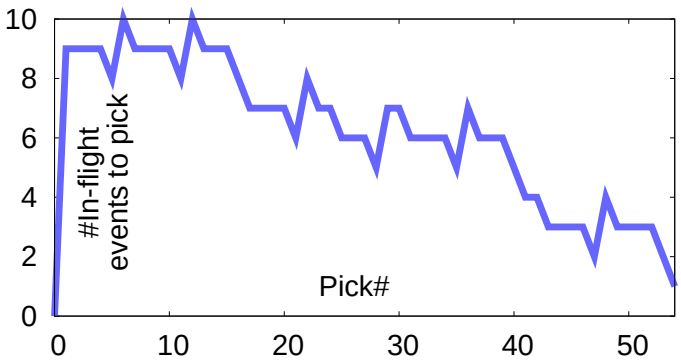



Figure 4.8: **Many choices make random techniques ineffective.** *For model checking complex protocols such as Paxos* `CASS-1`*, the figure shows how many inflight messages (y-axis) that can be chosen for every to-enable event (x-axis) within a path execution. For example, for pick #10 (x=10), there are 9 events to choose from (y=9). The figure shows that there are up to 10 choices when making a pick, hence random techniques are not effective for finding bugs in "deep" complex protocols and workloads.*

the probability that the "interesting" event is picked is high. However, in the former cases (more complex concurrency), random is not effective as there are too many choices and it blindly re-orders non-interesting interleavings. For example, for `CASS-1`, Figure 4.8 shows how many inflight messages (y-axis) that can be picked (up to 10 choices) for every to-enable event (*i.e.*, for every pick) within a path execution (x-axis). This highlights how complex workloads/interleavings make random-hybrid techniques (`r-DP` and `br-DP`) not fast enough.

**(e)** Finally, FLYMC is the fastest among all methods. In our bug benchmark, we have not found any other checker that wins over FLYMC. For the most complex bug, `CASS-1`, FLYMC can find the buggy path in less than 2500 paths. In overall, FLYMC is faster *at least* by 16× on average and up to 78× ("at least" because of the non-finished explorations, labeled with "↑" in Figure 4.7a). Sometimes "significant state-space reduction does not automatically translate to proportional increases in bug-finding effectiveness" [54, Section 5.3], however, we believe our results show that it is possible to stay systematic and increase bug-finding effectiveness with more advanced reduction and prioritization strategies.

Now we discuss the wall-clock speed in Figure 4.7b. As discussed in Section 4.3, for example in `CASS-1`, the per-path execution time is around 40 seconds in SAMC, 6 seconds in total in

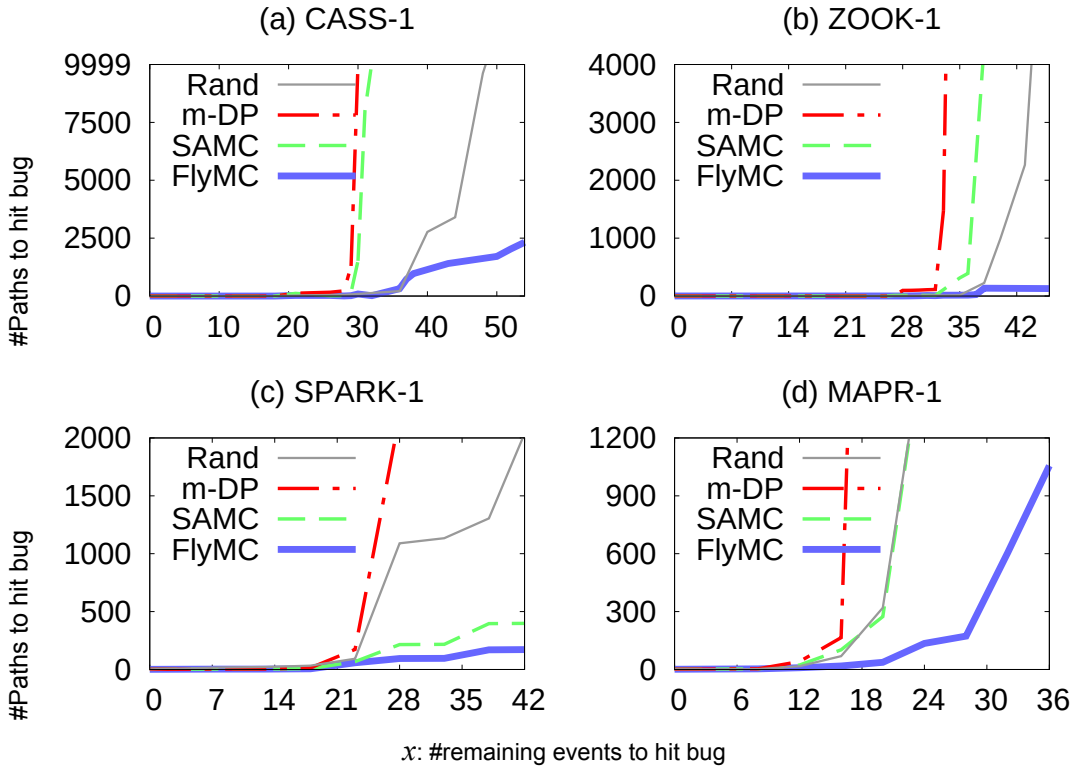



Figure 4.9: **FLYMC scalability.** *(As explained in Section 4.5.2).*

FLYMC, and 2 seconds (plus initialization time) in MODIST. Overall, per our design optimizations (Section 4.3), FLYMC is now 28× faster on average (up to 158×) compared to all methods. We do not show `Rand` in Figure 4.7b because we are comparing specific design implementations.

### *4.5.2 Scalability*

To analyze why non-FLYMC algorithms cannot or are slow to hit some of the deep bugs above, we plot a different type of graph in Figure 4.9. Here, the *$x$-axis* represents the number of *remaining events* to hit the bug. For this, we control the "*path prefix*," *i.e.*, an initial subset of the buggy path. The *maximum* value in the $x$-axis represents the *total* number of events to hit the bug without any prefix (as in the "#E" column in Table 4.1). For example, for reproducing `CASS-1` (Figure 4.9a), the workload generates a total of 54 events. Controlling path prefix means that the checker executes in deterministic order some of the earlier events (the prefix) and let the rest be reordered.

For example, in Figure 4.9a, with $x$=30, we first enable the first 24 initial events and then let the checker reorder the remaining 30 events. Ultimately, a checker should scale without any prefix at all (*i.e.*, no prior knowledge of the bugs).

The *y-axis* shows the number of paths explored until the bug is reached given the remaining events. For instance, in Figure 4.9a, at $x$=26, MODIST's `m-DP` must explore 163 paths to hit the bug, but `SAMC` and FLYMC are able to hit the bug in 55 and 27 paths respectively. With more remaining events to reorder (higher $x$), then more paths need to-be explored (higher $y$), *i.e.*, the bigger the path explosion problem will be.

Essentially, the graphs in the figure show how FLYMC is more scalable than other approaches. For example, in `CASS-1` (Figure 4.9a), at $x$=32, `SAMC` already explodes to more than 10,000 paths. On the other hand, FLYMC can find the buggy path in less than 2500 paths without any prefix (at $x$=54).

For `ZOOK-1`, our SAMC exploration results are different compared to the one reported in the SAMC's paper [67]. Upon our conversation with the SAMC developers, there are two reasons for this difference. First, SAMC includes all the initialization events to reach the initial state $S_0$ (*e.g.*, initial leader election to reach stable cluster), which FLYMC ignored. Second, in our checker, we implemented a proper vector clock, while SAMC did not, which causes our checker to detect more concurrent chain of events. Therefore, in our experiment, SAMC is no longer able to hit the bug under 10,000 paths. However, it does not take away SAMC's conclusion that SAMC is still an order magnitude faster than MODIST's DPOR algorithm.

### *4.5.3 Coverage*

For this evaluation, parallel flips algorithm is disabled because it is a prioritization algorithm. Hence, it does not affect all of the coverage evaluation.

**(a) State coverage:** We have mainly measured FLYMC's speed in finding buggy paths in prior sections. Another form of evaluation is the speed to cover unique protocol states over the explored

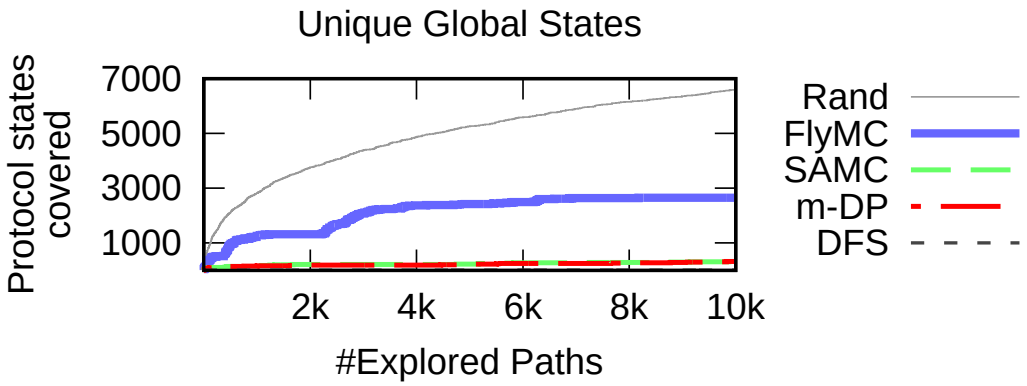



Figure 4.10: **State coverage.** *The figure shows the number of protocol states (y-axis) covered over explored paths (x-axis), as explained in Section 4.5.3a. A unique protocol state is stored as a hash value of a global state $S$ ($S$ is described in Section 4.1.1).*

paths. Figure 4.10 (similar to the format of Figure 10 in [103]) shows the Cassandra Paxos protocol states covered (in $y$-axis) under a 3-update Paxos workload in `CASS-1` within the first 10,000 explored paths (in $x$-axis). We make the following observations.

First, DFS is the worst among all (flat line). SAMC and mDPOR are faster but the growth rate is small. Random is the fastest, and for this reason, checkers sometimes mix their algorithms with random walks (see [103, Figure 10]), but unfortunately reduce their systematicity. So, random is fast in state coverage, but its non-systematic nature does not guarantee a buggy path to be found (*e.g.*, random fails to reach three bugs in Figure 4.7a).

Second, on the other hand, FLYMC does not sacrifice systematicity and is only 3× slower than random. Being *both* fast and systematic is feasible. The figure also shows that coverage growth rate reduces over time (*i.e.*, more paths to explore but they do not always lead to new unseen states). This last observation sheds a light on future work – how to identify which paths more likely to reach new states. The journey of advancing checkers should continue.

**(b) Complete coverage:** Another question is whether the entire state space in a given workload can be covered, *i.e.*, there are no more new unique states to explore. We performed this experiment for Cassandra Paxos and Kudu Raft workloads as shown earlier in Figure 4.1 on page 39, which

we now elaborate.

FLYMC *successfully exhausts* the state space for all the workloads, with one to three concurrent key-value updates in Kudu Raft and Cassandra Paxos (`Raft-1` to `-3` and `Paxos-1` to `-3`) within a reasonable time budget, as shown in Figure 4.1. The most complex one, `Paxos-3`, requires FLYMC to exhaust around 50,000 paths (1 machine week). `Raft-3` is a much simpler case than Paxos as Raft only allows one leader node (per table/partition) to coordinate concurrent updates; for example, three coordinators $A$, $B$, $C$ in Figure 4.2 (on page 40) updating the same key/partition is not allowed in Kudu Raft. Given this simplicity, Raft only needs two rounds, unlike Paxos' three rounds. For this reason, Raft search space is much smaller than Paxos.

MODIST's DPOR and SAMC cannot finish the exploration under 10,000 paths. They do not scale well under these workloads for the following reasons.

MODIST's *inter*-node-independence DPOR algorithm focuses on taming the $N$-induced explosion (*e.g.*, checking 3 nodes will not explode much compared to 2 nodes). However, under a more complex workload where the number of concurrent messages to *each* node increases, this inter-node-independence DPOR algorithm does not scale (for example, `Raft/Paxos-2/3` in Figure 4.1). What is needed is the *intra*-node message independence.

SAMC implements an intra-node message independence. For example, in a single Paxos update (`Paxos-1`), the `ack++` received by the coordinator in each round of the Paxos stages are considered commutative/independent. Hence, SAMC is more scalable than MODIST. However, SAMC's other algorithms (*e.g.*, crash independence and symmetry) do not work in no-crash workloads.

Under two concurrent updates ("`Paxos-2`" in Figure 4.1), the path exploration explodes significantly in all the checkers. This is because in a single update, the three Paxos rounds (prepare, propose, commit) are serialized, but under two updates, each round of the first update can interleave with any round of the second update.

**(c) Systematic coverage:** We use the same sense of "systematic" that is used with concurrent programs [45, 99], where it refers to exploring the state spaces of concurrent processes. Our

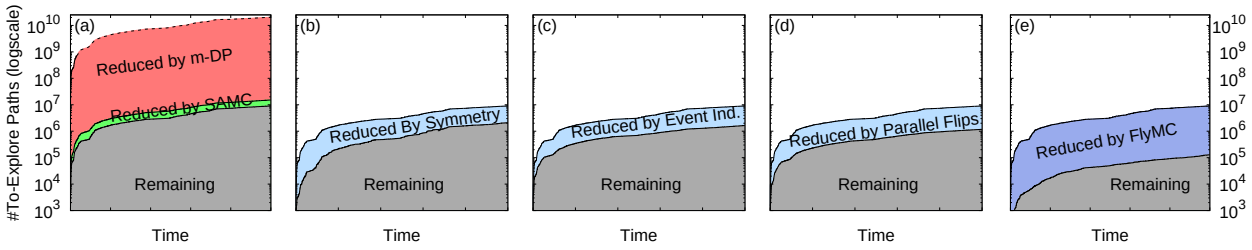



Figure 4.11: **Path explosion and reduction.** *The figure is explained in Section 4.5.4a. The y-axis represents the to-explore paths over time. Figure (e) shows that* FLYMC *reduces the path explosion problem by two orders of magnitude from* MODIST*'s DPOR and SAMC.*

FLYMC reduction algorithms are systematic in that they cover all states relevant to observable events, *i.e.*, the intercepted messages in distributed systems. These algorithms do not skip any interleavings that would lead to new unique states. We want to emphasize that this systematic property follows in principle from correct identification of communication and state symmetry and event independence, which is supported by FLYMC's static analyses (which we assume are correct). As experimental evidence, we collected all unique global states (compressed to hash values) explored by depth-first search (DFS) and MODIST's DPOR for `Raft-1`, `-2`, and `Paxos-1` examples. When compared with the explored states in FLYMC, we found that FLYMC had not missed any unique states. Note that we could not compare more complex workloads since the non-FLYMC techniques take too long to complete.

### *4.5.4 Per-Algorithm Effectiveness*

In this section, we evaluate the effectiveness of each individual algorithm by showing the reduced paths over time and the ratio of reduced paths per algorithm.

**(a) Reduced paths over time:** Figure 4.11 highlights the combined power of FLYMC algorithms, using `CASS-1` as a specific example. Figure 4.11a (in log scale) depicts the number of to-explore paths over time. The path explosion in the beginning, just after $x$=0, shows the many possible interleavings generated after the first path is exercised. Let us imagine that the first path contains 14 all concurrent events, a DFS checker would generate 14! new paths. However, not all of them

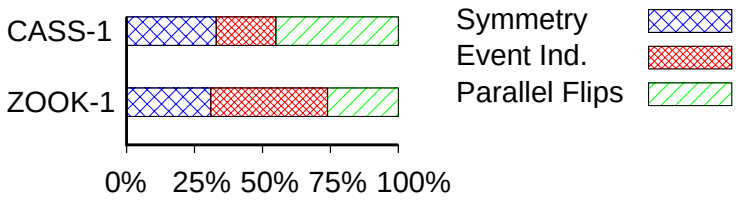



Figure 4.12: **% of removed and deprioritized paths by each algorithm.** *The symmetry and event independence areas represent % of reduced paths, while the parallel-flips are represents % of deprioritized paths. The figure is explained further in Section 4.5.4b.*

need to be exercised, because they are removed by the individual reduction algorithms.

The highest line in Figure 4.11a reflects the number of generated paths by a naive depth-first-search (DFS) algorithm without any reduction algorithm. The top region in Figure 4.11a depicts the number of paths reduced by MoDist's DPOR algorithm, roughly 3 orders of magnitude reduction from DFS, hence its popular usage in other checkers [95, 102]. Next, the middle region shows that SAMC slightly reduces the explosion (SAMC is not highly effective for this bug, as explained in Section 4.5.1-4.5.2).

Figures 4.11b-d depict the individual reductions by state symmetry, event independence, and parallel flips, each reduces the explosion by almost an order of magnitude. Ultimately, Figure 4.11e shows that all FlyMC algorithms *collectively* provide two orders of magnitude of reduction in the `CASS-1` Paxos workload.

**(b) Ratio of reduced paths per algorithm:** We plot Figure 4.12 to show the effectiveness of the individual FlyMC algorithms. Here, the $x$-axis represents the ratio of paths removed (by symmetry and event independence) and deprioritized (by parallel flips) from all the paths. This figure focuses on displaying two bugs from our benchmark with the most complex workloads. The graph essentially shows how all FlyMC algorithms successfully complement each other. We can also see that for different workloads, certain algorithms are more effective than the others.

For example, parallel flips are effective for `CASS-1` (45%) because this workload (three Paxos updates) generates a high degree of concurrency (*e.g.*, up to 9 outstanding events at a given time)

and the important flips are far from the end of the queue, which parallel flips address (as illustrated earlier in Figure 4.6). Symmetry also works best in `CASS-1` (33%) as the workload exercises replication-based protocols involving multiple worker/follower nodes, which are automatically considered symmetrical in FLYMC. Our event independence algorithm is effective in `ZOOK-1` (43%) as the messages in this workload update different sets of variables (*e.g.*, leader election messages and snapshot messages that touch different sets of variables) and, as this bug requires three crashes to surface, reducing unnecessary crashes is effective.

### *4.5.5 New Bugs*

Finally, our last evaluation tests whether FLYMC can find new bugs. For this, we integrated FLYMC with (1) a recent stable version of Cassandra and (2) ZooKeeper. (3) a 2-year old proprietary system; the proprietary system is a production system that supports five other cloud services within the company (akin to how ZooKeeper supports HBase, Yarn, and other cloud systems). We found 10 new bugs in total, all confirmed by the developers.

For Cassandra, we successfully discovered 2 new bugs that require 2 and 3 concurrent Paxos updates. The first bug requires around 39 messages sent across 3-nodes cluster. Figure 4.13 summarizes the complex interleavings needed to hit this bug. The second bug involves 2 concurrent messages, one crash, and one reboot at specific timings. We have communicated these two bugs to the developers and they have confirmed that those two bugs are real issues.

For ZooKeeper, we model check its "reconfiguration" feature, which allows ZooKeeper cluster to elastically grow and shrink while serving foreground requests without any downtime, hence a complex feature. FLYMC successfully discovered 3 new bugs. The first bug reveals that the developers' prior fix to an old DC bug was not robust enough, that there is another interleaving that makes the old bug surface. The second bug was reported to appear once every 500 unit test cycles. With FLYMC, we help the developers pinpoint the exact buggy path to reproduce the bug. The third bug is about two threads in a single node entered a deadlock due to a specific incoming

**CA-12126 :**
(1) Client submits Paxos **Write-1** to **node A**.
(2) Node A sends prepare messages and nodes A, B, C accept the prepare messages.
(3) Node A sends propose messages to all nodes.
(4) Node A accepts the propose message and *saves the value to "inProgress"*.
(5) *Before* node A's propose messages reach node B and C, Client submits **Write-2** to **node B**.
(6) Node B sends prepare messages and all nodes accepted the prepare messages.
(7) Node B receives nodes B & C's prepare response messages *before* node A's prepare response message, hence node B *did not* read the inProgress value from node A.
(8) *Write-2 fails*, because the IF condition is not satisfied.
(9) Client submits **Write-3** to **node C**.
(10) Node A's propose messages reach nodes B and C and got rejected because the proposals have a smaller ballot number.
(11) Node A receives nodes B & C's propose response messages, and since the propose was rejected, node A retries with a higher ballot number.
(12) However, *node A reached timeout*.
(13) Node C sends prepare messages to all nodes.
(14) Node C receives node A's prepare response message *before* the other two responses, hence node C *notices* node A's inProgress value.
(15) Node C recommits node A's inProgress value.
(16) Node C sends propose messages and all nodes accept the propose messages.
(17) Node C sends W1's commit messages and all nodes accept the commit messages.
(18) Because of this, Node C cannot continue W3.
(19) Client saw that CAS Write-1 & -3 timed out, CAS Write-2 was rejected, but the server saved CAS Write-1 data (*inconsistency between client and server*).

Figure 4.13: **A new DC bug in Cassandra Paxos.** *The list above summarizes the total order of the 39 messages to hit the bug.*

message timing and a local thread operation that was managing the node quorum.

For the proprietary system, FLYMC successfully discovered 5 new critical bugs that have significant impacts including unavailability (*e.g.*, no leader is chosen) and data inconsistency. The bug depths range from 9 to 30 events.

## 4.6 Summary

In this chapter, we present FLYMC, a fast, scalable, and systematic software model checker that covers all states relevant to observable events for testing distributed systems implementations.

FLYMC achieves scalability by leveraging the internal properties of distributed systems. Overall, FLYMC introduces three powerful algorithms: (1) communication and state symmetry, (2) event independence, and (3) parallel flips.

Overall, there are four contributions out of this chapter:

1. Highly scalable checker algorithms that provide systematic state coverage for given workloads. (Section 4.1).
2. A checker design that is backed with static analysis help developers extract information from the target system and use it to write the system-specific parts of the algorithms (Section 4.2).
3. Additional optimizations that improve the checker's wall-clock speed in exploring paths (Section 4.3).
4. A comprehensive integration with challenging applications, and detailed evaluations that demonstrate the checker's effectiveness. (Section 4.4-4.5).

More exciting challenges are on the horizon as no checkers to date completely control the timing of *all* non-deterministic events, such as messages, crashes, timeouts, local-thread schedules, as well as disk I/Os [68]. Hence, more research on scalable exploration algorithms are needed.

# CHAPTER 5

# HMC: HEURISTIC ALGORITHMS TO SPEED UP SOFTWARE MODEL CHECKER

*"This has become quite messy, we didn't foresee some of this during design, sigh." — A comment in MR-4819*

We have seen in Chapter 4, how FLYMC reduction algorithms, such as event independence and state symmetry, can help control the path-explosion problem in checkers. But if we look under the hood, the numbers of paths that need to be explored are still huge with respect to the complexity of the workload that is evaluated. In other words, it is impractical to cover all possibilities of a distributed system given a limited amount of time. Often times, in this situation, a checker is re-purposed to detect as many errors as possible before it runs out of time budget or computation resources. Therefore, arise the needs of heuristic algorithms to help checkers to quickly reach the corner cases and potentially, DC bugs, as soon as possible.

Software model checkers that only apply reduction algorithms, in general, will try to reduce the numbers of redundant prefix paths that it needs to explore. But once those prefix paths are placed in the to-explore paths, the checker will evaluate each prefix path in a naive First-In-First-Out (FIFO) fashion. To understand the inefficiency of this approach, let's observe Figure 5.1.

In this figure, we have a tree with multiple tree nodes. Each tree node has an ID that represents the ID of a prefix path stored in the to-explore path, *e.g.*tree node with the number 1 in it is equivalent to `Path-1`. For this illustration, the exact order of events that a path/tree node represents is insignificant. What matters is that the children of a parent node will only be added into the to-explore paths once the parent node is executed (*e.g.*, `Path-2` and `Path-3` will only be added to the to-explore paths once the `Path-1` is executed).

Suppose, a bug will be detected after `Path-7` is executed and the checker evaluates each prefix path in the to-explore paths in FIFO fashion (*i.e.*, following the dashed blue arrows), then the

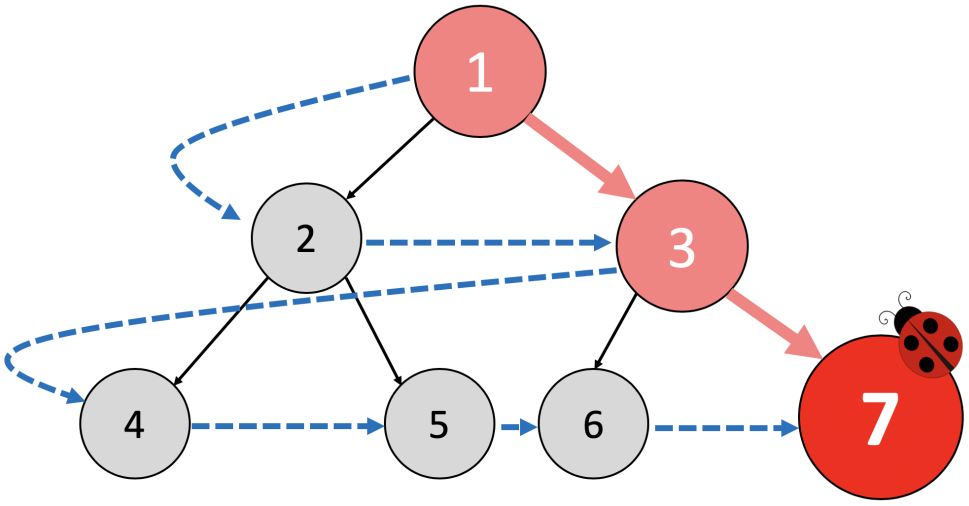



Figure 5.1: **To-Explore Paths Potential Improvements.** *Each node represents an interesting to-explore prefix path, except* `Path-1` *which is the first path explored. The children nodes of a parent node are added to the To-Explore Paths once the parent node is explored. Systematic software model checkers without any heuristic algorithms will explore each node with naive FIFO mechanism (following the dashed blue arrows). The red nodes represent the minimum necessary prefix paths to be explored to detect the DC bug found in the red node with a bug attached.*

checker will only detect the bug after it executes seven paths. But, as shown by the red nodes, the necessary prefix paths that need to be evaluated by the checker to detect the DC bug (represented by `Path-7`) are only `Path-1` and `Path-3` which then lead us to `Path-7`. That is, instead of executing seven paths, the checker really only needs to execute three paths to detect the bug.

We found similar patterns shown in Figure 5.1 in two complex Cassandra bugs that FLYMC has successfully detected. In Figure 5.2, we can see that FLYMC detects CA-6023 and CA-12438 after exploring 2299 paths and 985 paths, respectively. In further investigation, we discovered that the checker potentially can detect each bug in 6 path explorations (following the order of the red nodes shown in Figure 5.2). In other words, there is a big room of improvements for checkers to speed up its corner cases coverage if it knows how to distinguish which prefix paths are more-likely to lead to corner cases by replacing the naive FIFO approach with smarter heuristic algorithms.

Prior checkers for distributed systems, such as MACEMC [63] and PCTCP [83] have introduced heuristic algorithms that are based on random walks. The benefit of this approach is that the heuristic algorithm implementation is simple, yet it provides a reasonable improvement on how fast

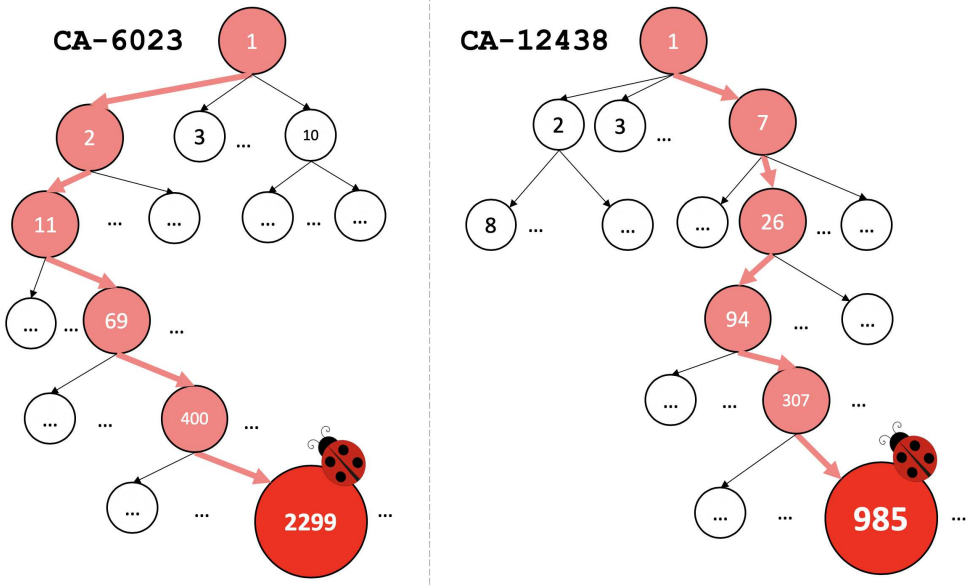



Figure 5.2: **FLYMC To-Explore Evaluation.** *This figure follows Figure 5.1's format. It shows how* FLYMC *evaluated its to-explore paths to detect CA-6023 and CA-12438.*

the checkers can quickly cover corner cases. Furthermore, other checkers, such as MODIST [103] and taPCT [84] have tried to combine reduction algorithms (*e.g.*, DPOR [40, 44]) with some depth-bounded random algorithms.

However, based on the checker's community for multithreaded applications, we could learn that smart heuristic algorithms might beat random heuristics on efficiency [38, 47, 50, 99]. Their smart heuristic algorithms are developed based on the *checker property-specific* (*e.g.*, blocking statements for deadlock, queue size) or by the *target system structure* (*e.g.*, branch counting, thread interleavings). Although their solution is powerful to check multithreaded applications, adapting those algorithms in checkers for distributed systems won't be straightforward as distributed systems have unique properties compared to multithreaded applications.

For this chapter, our goal is to develop a systematic checker to quickly and efficiently explore corner cases by introducing smart heuristic algorithms. The question that we ask ourselves is, how can the checker smartly weigh which prefix paths among all to-explore paths are more-likely to reach corner cases? In other words, what kind of properties of the checker or distributed systems can be exploited to prioritize more-likely buggy prefix paths?

We present HMC, a systematic software model checker that is empowered with novel heuristic algorithms to reach corner cases faster. HMC covers all states relevant to observable events for testing distributed systems implementations. HMC achieves its fast coverage without mixing its algorithms with any *random* algorithms [83, 84, 103], but only by prioritizing the to-explore paths based on some internal properties of distributed systems and the model checker. We illustrate 4 HMC's algorithms as follows:

*(1) Blocking State-Event:* In general, corner cases can be represented with global states and events that are uncovered by previously explored test scenarios. Thus, HMC prioritizes to execute paths with the most uncovered state-event pair among all to-explore paths as these paths have a higher likelihood of reaching new global states.

*(2) Last State-Event:* As the covered state-event pairs knowledge is growing as more paths get explored, the Blocking State-Event prediction eventually may predict all events of the prefix path. But, existing events in the last queue may potentially still lead the prefix path to an uncovered state-event. Hence, HMC will not only focus on predicting the state transition of all events in the prefix path, but also its runtime events queue.

*(3) Miss-Prediction Step:* As the methods above focused on collecting most frequent blocking state-event pair prefix paths at the front of the to-explore paths queue, the checker needs a method to sort prefix paths in each group. Hence, Miss-Prediction Step prioritizes the prefix path with the most higher chance to cover the most number of global states transitions.

*(4) Prioritized Node Crash:* While previous methods help the checker to efficiently detect DC bugs that only involve message events, unfortunately, they are not good enough to guide workloads that involve crashes and reboots. To solve this issue, Prioritized Node Crash allows testers to define the sequence of crashes that she wants to focus on (*e.g.*, crash `node-1`, then crash `node-2`).

Collectively, the algorithms make HMC on average 6× (up to 15×) faster than FLYMC which is the state-of-the-art of systematic software model checker. In total, HMC has successfully reproduced 5 bugs in Cassandra *without* any random algorithms.

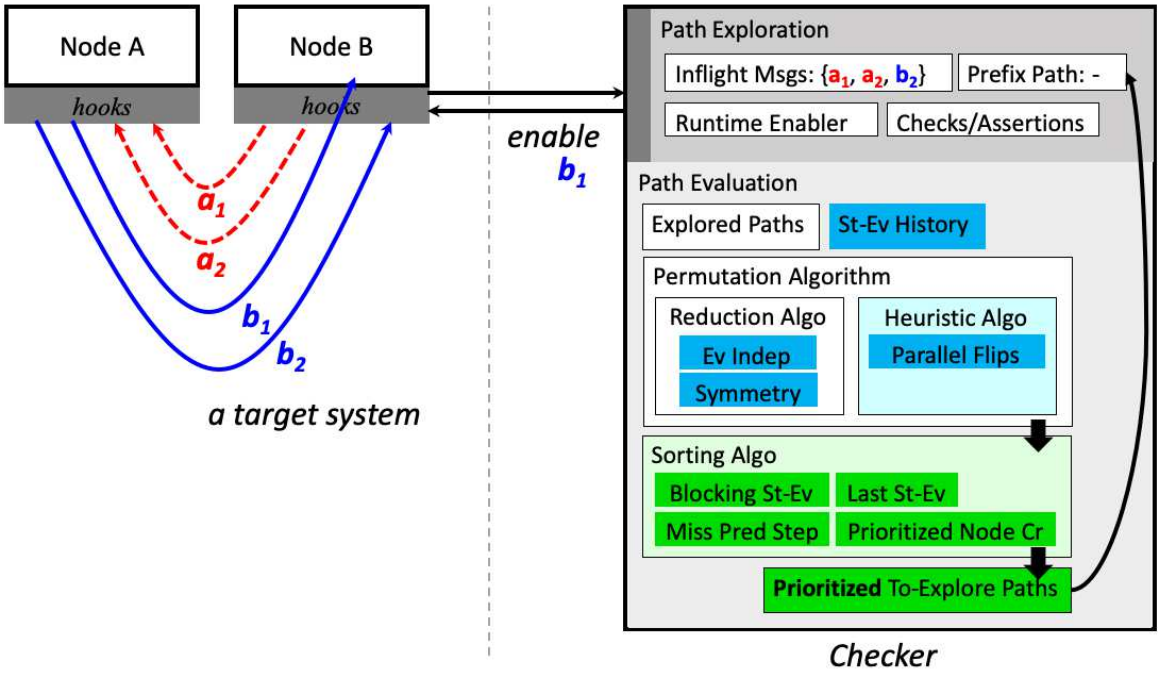



Figure 5.3: **HMC architecture.** *The green-colored boxes represents* HMC *improvements on top of the* FLYMC *improvements (blue-colored boxes) – explained in Section 5.1.*

## 5.1 HMC Architecture

In principle, HMC architecture follows the similar main concept of the other checkers that we have discussed in chapter 2.

In order to implement the new HMC heuristic algorithms, we introduced a new component in the path evaluation phase that we referred to as the *sorting* algorithm. The input for this sorting algorithm is all prefix paths in to-explore paths that are generated by the permutation stage. Next, HMC will weigh each prefix path based on HMC's heuristic algorithms. As a result, the prefix paths will be sorted in descending order based on its score, such that the prioritized to-explore paths will poll the prefix path with the highest score when the checker starts the next path exploration.

With this sorting algorithm, the checker will have an alternative besides the naive First-In First-Out (FIFO) approach to decide which prefix path that it will explore next. Therefore, HMC can wisely consider the more-likely prefix path that reaches any corner cases faster.

## 5.2 HMC Algorithms

By considering the properties of the distributed systems and the runtime checker, HMC introduces four novel heuristic algorithms: blocking state-event (Section 5.2.1), last state-event (Section 5.2.2), miss-prediction step (Section 5.2.3), and prioritized node crash (Section 5.2.4). These algorithms empower HMC to reach corner cases faster by prioritizing paths that more-likely discover new global states that have not been seen before.

The foundation of all HMC algorithms is path prediction based on the abstract state-event history as shown in Figure 5.4. First, during the path exploration phase, the checker records the state-event transition pair to the state-event history. Along the process, HMC builds the abstract state-event history based on that state-event history where each state-event will exclude node-specific properties (*e.g.*, node ID, event sender ID, event receiver ID).

Next, after the checker generates the next prefix paths of the explored path in the permutation phase and places it into the to-explore paths, all of the to-explore paths will be evaluated by the sorting algorithm. In this sorting algorithm, for each prefix path prediction, the checker will prepare initial global states that will be called as predicted global states (*e.g.*, $gs_0$).

Suppose in Figure 5.4, all message events ($a$, $b$, $c$) will go to node-0. The checker will take one event at a time from the current prefix path (*e.g.*, it starts with an event $a$) and try to find the next state transition for node-0 based on the abstract state-event history knowledge. If the current abstract state-event pair can be found, then it is a successful step prediction, then the predicted global states will be updated (*e.g.*, abstract $s_0 + a$ and abstract $s1 + c$ are found). Otherwise, the prediction will stop and the prefix path prediction is flagged as incomplete (*e.g.*, the abstract $s_2 + b$ failed to be found, hence the prefix path prediction failed). If the state prediction keeps succeeding until all events are predicted,then we refer this prefix path as *fully predicted prefix path*.

What the checker will do next with the prediction result is where the HMC algorithms will come in and attribute some scores to the prefix path. In general, the score from each algorithm will be multiplied with a configured multiplier for each algorithm. That is, the multiplier configuration

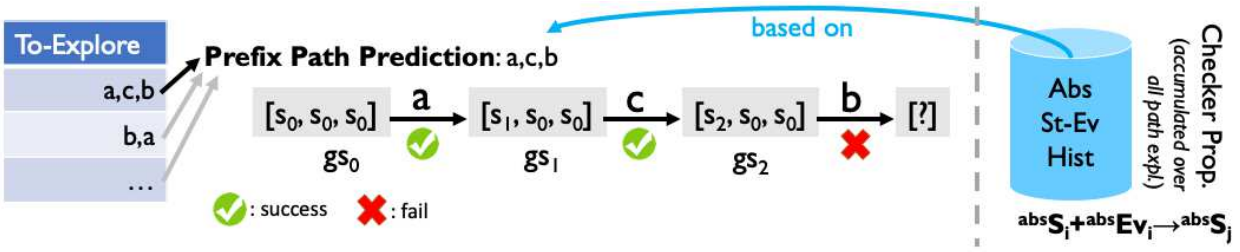



Figure 5.4: **HMC Algorithms Foundation.** *All prefix paths in the to-explore paths will be predicted based on the Abstract State-Event History (consists of a set of abstract $s_i + e_i \rightarrow s_j$) that are accumulated over all path explorations. If the current state + event combination is found, then the predicted global state progressed. Otherwise, path prediction stops. In this example, event $a$ & $c$ are successfully predicted, but event $b$ does not.*

determines which algorithm will be weighed more than the others. The final result of this sorting algorithm then is a descending-score prioritized to-explore path, where the highest score prefix path will become the next path that will be explored.

To emphasize, the main goal of a checker is to quickly detect DC bugs, if any exists. But, in reality, the checker will have no clue which paths will violate any safety or liveness properties until the checker executes it. So, the feasible goal is not to find which prefix paths are buggy, but to find which prefix paths will reach uncovered global states. This goal will be the principle for the HMC algorithms in this section.

### *5.2.1 Blocking State-Event*

• **INTUITION:** Our first intuition for this algorithm is, if a prefix path is fully predicted, then we can conclude that the prefix path will not reach new global states. As a consequence, that prefix path should be deprioritized. Conversely, if the checker fails to fully predict a prefix path because the abstract state-pair history has not recorded the current predicted global states against the current to-be-predicted event, then the particular prefix path potentially reaches a new global states space that the checker has not discovered before. Hence, the checker should prioritize the prefix path.

Furthermore, when the checker reaches the event which it fails to predict based on the current predicted global states, then the checker has found the next state-event pair that is interesting to

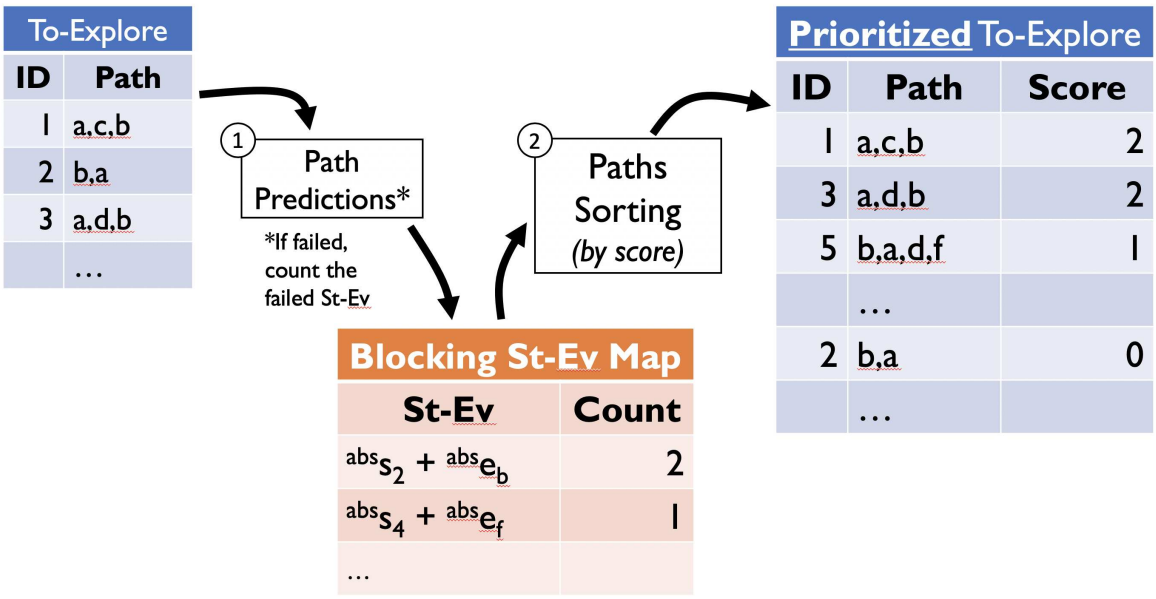



Figure 5.5: **Blocking State-Event Algorithm.** *This algorithm is described further at Section 5.2.1 – Algorithm.*

be explored. We refer to this state-event pair as a blocking state-event pair. Once, the particular prefix path that contains the blocking state-event pair is executed, then potentially new global states have been discovered and a new abstract state-event pair (and potentially more than one pair) will be added to the abstract state-event history. An efficient checker should focus on optimally discovering unique abstract state-event pairs over time.

We introduce the Blocking State-Event to guide the checker to advance its abstract state-event pairs as fast as possible. Its main goal is to prioritize prefix paths with most frequent blocking state-event pair, so that those prefix paths can quickly progress to new global states and quickly discover another blocking state-event pair (if it still failed to be fully predicted), or get deprioritized (if it finally gets fully predicted) and eventually the checker can explore other prefix paths.

• **ALGORITHM:** To illustrate this algorithm, let's look at Figure 5.5. At the beginning of the sorting algorithm, the checker prepares an empty blocking state-event map which will be used to record the blocking state-event pairs and to count its frequency of appearances. Next, in step 1, the checker will predict each prefix path in the to-explore paths. The checker prepares the initial predicted global states (*e.g.*, $[S_0,S_0,S_0]$ shown in Figure 5.4). Then, the checker will go through

each event in the prefix path, to predict the transition of each local state (or global states with respect to a crash event) against the current event by checking the abstract state-event history as shown in Figure 5.4.

If the current predicted state-event pair is not found in the abstract state-event history, then the current prefix path prediction failed and the current predicted state-event pair is counted up in the blocking state-event map. Otherwise, the predicted global states will be updated with the appropriate state effect. This process is repeated until the prefix path is fully predicted or it stops due to blocking state-event conditions above.

Lastly, in step 2 of Figure 5.5, after all prefix paths are evaluated by the path predictions process, the checker will assign the score to each prefix path. Specifically, the counter for each blocking state-event pair will be multiplied by the configured Blocking State-Event multiplier and then assigned to each prefix path accordingly. By its final score, all prefix paths will be sorted and the checker will have its prioritized to-explore paths.

In the next path exploration, this process will be repeated again as the checker generates more prefix paths to its to-explore paths, and its abstract state-event history also gets updated. Therefore, the blocking state-event map will be emptied and the blocking state-event counting will be repeated again. To avoid unnecessary computation, the checker does not need to re-predict fully predicted prefix paths.

### *5.2.2 Last State-Event*

• **INTUITION:** After a prefix path is fully predicted, the checker path prediction will know the final global states for that prefix path following the Blocking State-Event algorithm (Section 5.2.1). But, without the Last State-Event algorithm, the checker will determine the prefix path score mainly on whether the end global states are fully predicted or not and ignoring the possibilities of the next transition based on outstanding events in the last queue. This might cause the checker to deprioritize possible prefix paths that still can lead us to some corner cases as if it exercises any

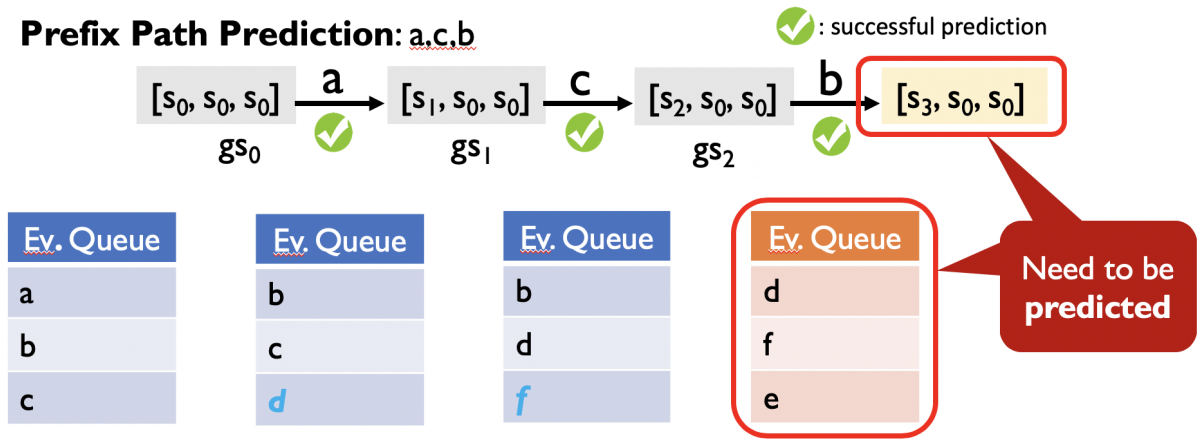



Figure 5.6: **Last State-Event Intuition.** *This figure is described further at Section 5.2.2 – Intuition.*

event in the last queue. That is, if the checker keeps track of the events queue transition through the prefix path prediction, the checker will have a complete perspective of the prefix path possibilities to reach corner cases.

As shown in Figure 5.6, the Last Event-State algorithm will focus on predicting the events queue transition for all events in a prefix path prediction, so that after predicting the last event in the prefix path, the checker has the last global state of the prefix path and the last event queue where each event can be executed next. Hence, it can predict each event in the event queue against the appropriate node state in the global states and follow the Blocking State-Event algorithm in determining whether all pairs can be fully predicted or not.

To predict the event queue transition, the Last Event-State will build the *causal new events database* on every event execution during the path exploration phase. The causal new events database will store the $S_i{+}e_j{\rightarrow}\{e_k, e_l, ...\}$ for every event. But, in contrast to the abstract state-event history, the causal new events cannot be abstracted as it needs to be aware of the node sender and receiver for each message event. This is also true for the other new messages that occur due to a crash event.

• **ALGORITHM:** Once the causal new events database is correctly developed over the path exploration phase, then predicting the events queue over the path prediction becomes trivial. The checker

starts with preparing the initial global states and the initial events in the events queue based on the workload. During the path prediction, besides predicting the state transition, the checker should also update the event queue based on the causal new events database.

One last issue that's left is if the checker could not continue predicting the events queue during the path prediction, because the causal new events database is more rigid in comparison to the abstract state-event history. For this case, we decide to not flag the prefix path as a fully predicted path when this last event state algorithm is activated. Hence, at the end of the next path exploration, the checker will retry to predict the prefix path.

On the other hand, when the last events queue is correctly predicted, then the last predicted global states will be predicted against each event in the last events queue. That is, those state-event pairs will be considered in the Blocking State-Event scoring process (Section 5.2.1).

### *5.2.3 Miss-Prediction Step*

• **INTUITION:** Up until now, the Blocking State-Event and Last State-Event algorithms target to prioritize paths with the most frequent blocking state-event pair. To help the checker decide which paths that should be executed first among each group of paths with the same blocking state-event pair, we introduce the Miss-Prediction Step algorithm.

Miss-Prediction Step algorithm prioritizes prefix paths with the largest gap between when the miss-prediction step happens and the length of the whole prefix path. This is simply due to the fact that when more events left in the prefix path to explore after the blocking state-event pair is executed, then there is a bigger chance for the particular prefix path to reach various different global states transitions. That is, it is possible for a prefix path to explore more than one new global states per prefix path exploration.

• **ALGORITHM:** To implement Miss-Prediction Step algorithm, during the path prediction, the checker records when the miss-prediction step happens and compares that with the total events in the prefix path. Lastly, it multiplies the difference with the Miss-Prediction Step multiplier.

### *5.2.4 Prioritized Node Crash*

• **INTUITION:** As we start running our benchmark with all previous algorithms, we observe that they were very effective for workloads that have no crashes. But, when we start exploring workloads that involve crash event, the previous three heuristic algorithms focus on exploring a specific node crash area only. Further analysis shows that this happens because a path will most likely be dominated by message events, hence the blocking state-event pairs will be dominated by some message events or with a specific node crashes that were explored in the first path exploration. Hence, if the node crashes happen in the other node ID, although it is possible that the state-event pair is unique, it is not frequently seen among all other to-explore paths which lead to a very low priority to be explored. To solve this issue, we introduce the Prioritized Node Crash algorithm. This algorithm allows tester to define the sequence of node crashes that she wants to explore in current exploration. For example, a tester can define to direct checker to specify the node crash sequence to be $\not{A}\not{B}$, then all prefix paths that have this subsequence in its path, will be prioritized over the other prefix paths that don't have it.

With this algorithm, testers can run multiple explorations in different machines where each exploration will focus on exploring a specific sequence of node crashes very effectively. Based on TaxDC, testing 3-nodes cluster with at most 2 crashes will be enough to cover the majority of DC bugs that involve fault timing issues (Figure 3.3d).

• **ALGORITHM:** First, tester needs to specify sequence of crashes that will be prioritized for the exploration. During the path prediction, when the checker found matching subsequence of crashes in the prefix path, then the particular path will get score configured for this Prioritized Node Crash algorithm. In default, this algorithm will have the highest multplier, hence path exploration with expected subsequence will raise up high in the to-explore paths.

## 5.3 Evaluation

We now evaluate HMC by presenting experimental results that answer the following questions: (1) How fast is HMC in detecting DC bugs compared to the state-of-the-art systematic checker, FLYMC? (2) How many unique global states that HMC can discover in comparison to other approaches? To answer these questions, we integrated HMC to some versions of Cassandra system: v2.0.0, v2.0.15, and v3.7.

• **DC BUG BENCHMARKS:** Table 5.1 shows the DC bug benchmarks that we use to evaluate how fast a checker can reproduce a known bug given the corresponding workload. The table consists of information about the bug depth (or the number of events needed to hit the bug), the number of crashes, and the number of reboots involved. All of the DC bugs that detected are related to Cassandra Paxos (or also known as the Light Weight Transaction) protocol.

• **TECHNIQUES COMPARED:** We compared HMC with FLYMC as we consider FLYMC to be the state-of-the-art of the systematic software model checker for distributed systems. And as HMC is developed on top of FLYMC, we will see a direct impact of all HMC's algorithms on improving the checker's ability to quickly explore new corner cases.

## 5.4 Speed in Detecting Known Bugs

This section evaluates the speed of HMC vs. FLYMC (representing the state-of-the-art systematic software model checker) in detecting known DC bugs. In total, we successfully reproduced 5 DC bugs in Cassandra. Figure 5.7 illustrates some DC bugs that HMC reproduced.

Table 5.1 shows the result of our comparison. Here, we make several conclusions about our explorations. First, we have demonstrated that HMC algorithms can really help to speed up systematic software model checkers to detect DC bugs. We do not need to apply any randomness to guide the checker. In other words, although the checker explores the to-explore paths not in a FIFO mechanism, HMC allows the checker to stay systematic but yet smarter in deciding which paths

**CASS-2:**
**(1)** A client submits Paxos *Write-1 (W1)* to node A with a column in key K's row.
**(2)** Node A sends W1's prepare messages and propose messages.
**(3)** All nodes accepted the prepare and propose messages.
**(4)** Node A sends W1's commit messages.
**(5)** *Node C crashes before* accepting the commit message.
**(6)** Nodes A and B accepts the W1's commit messages. At this point, A and B have stored W1 locally.
**(7)** *Node C reboots*.
**(8)** Another client submits Paxos *Write-2 (W2)* to A, updating another column in key K's row.
**(9)** Node A sends W2's prepare messages (then propose and commit messages), accepted by all the nodes. At this point, Paxos nodes incorrectly have *inconsistent data*; A and B store W1-2, but C only stores W2 locally. The read repair did not happen during W2's preparation. Thus, if a client reads K from C, she would get an inconsistent data (missing W1's update).

**CASS-3:**
**(1)** Client submits Paxos **Write-1** to **node A**.
**(2)** Node A sends prepare messages and nodes A, B, C accept the prepare messages.
**(3)** Node A sends propose messages to all nodes.
**(4)** Node A accepts the propose message and *saves the value to "inProgress"*.
**(5)** *Before* node A's propose messages reach node B and C, Client submits **Write-2** to **node B**.
**(6)** Node B sends prepare messages and all nodes accepted the prepare messages.
**(7)** Node B receives nodes B & C's prepare response messages *before* node A's prepare response message, hence node B *did not* read the inProgress value from node A.
**(8)** *Write-2 fails*, because the IF condition is not satisfied.
**(9)** Client submits **Write-3** to **node C**.
**(10)** Node A's propose messages reach nodes B and C and got rejected because the proposals have a smaller ballot number.
**(11)** Node A receives nodes B & C's propose response messages, and since the propose was rejected, node A retries with a higher ballot number.
**(12)** However, *node A reached timeout*.
**(13)** Node C sends prepare messages to all nodes.
**(14)** Node C receives node A's prepare response message *before* the other two responses, hence node C *notices* node A's inProgress value.
**(15)** Node C recommits node A's inProgress value.
**(16)** Node C sends propose messages and all nodes accept the propose messages.
**(17)** Node C sends W1's commit messages and all nodes accept the commit messages.
**(18)** Because of this, Node C cannot continue W3.
**(19)** Client saw that CAS Write-1 & -3 timed out, CAS Write-2 was rejected, but the server saved CAS Write-1 data (*inconsistency between client and server*).

Figure 5.7: CASS-2 **and** CASS-3 **Scenarios.** *The list above summarizes the order of events that need to be executed at specific timings to reproduce* CASS-2 *and* CASS-3.

| BugName | Issue | #Dp | #Cr | #Rb | #Executions FLYMC | #Executions HMC | HMC vs. FLYMC |
|---|---|---|---|---|---|---|---|
| CASS-1 | CA-6023 | 54 | – | – | 2299 | 358 | 6.4 |
| CASS-2 | CA-12438 | 48 | 1 | 1 | 985 | 64 | 15.4 |
| CASS-3 | CA-12126 | 39 | – | – | 11 | 3 | 3.7 |
| CASS-4 | CA-6013 | 30 | – | – | 11 | 2 | 5.5 |
| CASS-5 | CA-5925 | 15 | – | – | 2 | 2 | 1 |

Table 5.1: **DC bugs benchmarks.** *The table lists DC bugs used to benchmark the checkers scalability. "#Dp" refers to the bug depth (number of events to hit the bug), "#Cr" refers to the number of crashes, "#Rb" refers to the number of reboots.*

to explore.

Second, HMC is one to two orders of magnitude faster compared to FLYMC. On average, HMC is 6× (or up to 15×) faster than FLYMC. In all DC bugs explorations, HMC is always better or as fast as FLYMC. In more detail, for CASS-1, CASS-3, CASS-4, and CASS-5, our HMC evaluation applies the Blocking State-Event, the Last State-Event, and the Miss-Prediction Step. As we explore CASS-2, we apply the additional Prioritized Node Crash where we configure one of the follower nodes to crash. Applying the Prioritized Node Crash in the other bugs won't help as those bugs have no crash event involved.

## 5.5 Unique Global States Coverage

For this section, we evaluate how fast the checkers cover unique protocol states over the explored paths. Figure 5.8 shows the Cassandra Paxos protocol states covered in $y$-axis under a 3-update Paxos workload in CASS-1 within 350 explored paths (in $x$-axis). Here are our observations.

First, HMC covers about 4× more unique states until each checker covers 350 paths. HMC achieves this result without introducing any random algorithms, hence it stays systematic in discovering new global states (*i.e.*, corner cases). Second, while, FLYMC line is flattening at the end, HMC still has an increasing (although with less degree), which shows that it potentially can discover even more unique states in a short period of time.

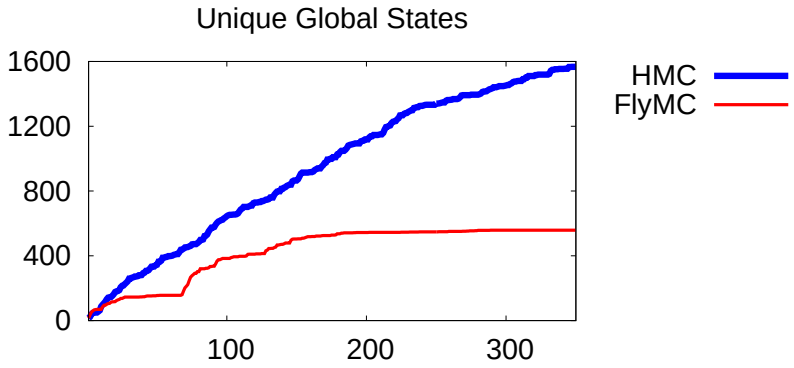



Figure 5.8: **State coverage.** *The figure shows the number of protocol global states (y-axis) that has been covered over the number of explored paths (x-axis) during exploring* `CASS-1`.

## 5.6 Summary

As distributed systems arise to be the backbone of cloud services, they are threatened by DC bugs that might linger in the dark. A small time budget often times increases the chance for these DC bugs to slip into the production site, causing severe implications to the cloud services. To address this issue, software model checkers cannot only be covering a large space of states, they need to discover as many unique states as possible in a given time budget to boost the developers' confidence in the quality of the code that they write.

In this chapter, we present HMC which introduces four novel heuristic algorithms to quickly reach corner cases by exploiting the checker and distributed system properties. Our evaluation shows that HMC has produced a significant improvement in terms of uncovering unique global states fast. We hope that HMC's work can trigger future research in software model checking.

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

*"This issue, and the other related issues ... makes me very nervous about all the state combinations distributed between [ZooKeeper and many HBase components]. After this is done, do you think we can come up with a simpler design? I do not have any particular idea, so just spitballing here." — A comment in HB-6060*

This dissertation's main goal is to improve distributed systems reliability by combating distributed concurrency (DC) bugs using software model checking technique. To achieve this goal, first, we conduct a bug study on DC bugs to understand the nature of DC bugs. This knowledge helps us in advancing software model checking to quickly detect DC bugs by introducing more powerful reduction algorithms to have better control over the path-explosion problem and smart heuristic algorithms to guide the software model checker to reach corner cases fast. This chapter concludes this dissertation work and future work in combating DC bugs with software model checking technique.

## 6.1 Conclusion

DC bugs impact the reliability of real-world distributed systems. Even with all redundancy and fault-recovery mechanisms deployed in today's systems, DC bugs make the software the single point of failure. To address this issue, this dissertation is mainly divided into two parts: understanding the taxonomy of DC bugs and empowering software model checkers to quickly detect DC bugs.

First, we have conducted an in-depth study on DC bugs which focus to narrow the gap between the theory and practice of distributed systems. We evaluated hundreds of real-world DC bugs to understand the triggering conditions, bug manifestations, and the bug fixes to provide guidance

on combating DC bugs. Furthermore, we have also presented the patterns, the root causes, and misconceptions that developers and the research community have in regards to DC bugs.

Next, as a solution, we introduced FLYMC which consists of three powerful algorithms to empower software model checking that is integrated into the implementation level of distributed systems. The algorithms allow the software model checker to test distributed systems under more complex workloads than other existing state-of-the-art model checkers. FLYMC's algorithms speed up the checker's ability in detecting DC bugs on average 16× (up to 78×) than other state-of-the-art checkers. It has been integrated with 8 distributed systems, successfully reproduced 12 known bugs, and found 10 new DC bugs which all are confirmed by developers. That is, we speed up the model checker with some systematic approaches and *without* any random walk or manual checkpointing techniques.

Lastly, we presented HMC, software model checker that is empowered with four novel heuristic algorithms that are based on the checker and the distributed systems properties. These heuristics allow the checker to quickly reaches corner case without applying any randomness. And as a result, HMC has done further improvement on top of FLYMC by detecting 5 Cassandra bugs on average 6× (up to 15×) faster.

## 6.2 Future Work

In terms of future work, our vision is to see software model checking to be adopted by the industry and to see it help developers detecting hidden DC bugs in their systems. To achieve this vision, here are several directions that need to be explored further.

### *6.2.1 Automatic Workload Generator*

One of the first issues that the checker experience in detecting new DC bugs is its dependency on developers/testers to define the workloads that need to be checked. Second, if we want to combine all workloads' possibility, the combinations might be infinite. However, as we have learned in the

TaxDC, in order to detect more DC bugs, we found that at most 2 protocols need to be exercised together. Both foreground and background protocols or two background protocols need to be triggered concurrently as those are the complicated interactions that developers hardly test out. We also need to consider up to 2 crashes to detect majority of DC bugs.

We advocate that future checkers need to smartly generate interesting combinations of workloads given a set of a system API (the foreground protocols) and configurable or timely operations (background protocols). Checkers also need to combine the workloads of some protocols with iterative crash injection (*i.e.*, the number of injections increase based on some threshold or the number of unique states that it has discovered). With this automatic workload generator, we believe more scenarios that are prone to DC bugs can be automatically explored. In other words, the checker will have less dependency on the developers to start its exploration.

### *6.2.2 CompleteMC*

Even though FLYMC has further control of the path explosion problem in software model checking and HMC has introduced heuristics to guide checkers to quickly reach corner cases, both checkers still have not introduced various triggering events, such as timeouts, disk I/O accesses, and local computations timing. Hence, there might be even more DC bugs that have not been detected by checkers. However, once these types of events are introduced to the checkers, the path explosion problems will get worst again. In other words, more advanced reduction and heuristic algorithms will be needed for this type of CompleteMC.

### *6.2.3 FastMC*

Besides the dimension of path explorations, the checker also has the issue in the dimension of how fast it can explore a path. That is, the checker itself needs to be optimized in many aspects of how it conducts path executions and path evaluations. However, requiring developers to rewrite their applications using some specific formal language equipped with the software model checker is not

feasible because many distributed systems are already widely used but have the necessity to be checked. Hence, optimizing stand-alone checkers and improving the method to integrate it with existing systems are key to seeing software model checkers to be utilized widely in the industry.

### *6.2.4 Domain-Specific Specifications*

Lastly, to unleash the real power of software model checking, a checker is very much dependent on the system specifications to judge the manifestation of DC bugs. This is a plague to many tools. As mentioned in TaxDC, DC bugs can manifest itself as an explicit error or as a silent error. Deploying generic ”textbook” specifications (*e.g.*, “only on leader exists”) do not help as developers sometimes have their own specifications (*e.g.*, ZooKeeper allows two leaders at a single point in time). Developers also bemoan the hard-to-debug fail-silent problems and prefer to see easier-to-debug fail-stop bugs. In short, no matter how sophisticated the checkers are, they are ineffective to detect DC bugs without accurate specifications.

For future checkers, we need to empower developers to express its domain-specific specifications or inference of local specifications that can show early errors or symptoms of DC bugs. Only then, checkers will be able to guide developers better in developing reliable distributed systems.

# REFERENCES


[1] http://ucare.cs.uchicago.edu/projects/cbs/.

[2] Apache Cassandra. http://cassandra.apache.org.

[3] Apache Hadoop. http://hadoop.apache.org.

[4] Apache Hadoop NextGen MapReduce (YARN). https://hadoop.apache.org/docs/current/.

[5] Apache HBase. http://hbase.apache.org.

[6] Apache ZooKeeper. http://zookeeper.apache.org.

[7] Chameleon Haswell Website. https://bit.ly/2KrnE4L.

[8] Emulab d430 Website. https://wiki.emulab.net/wiki/d430.

[9] Ethereum. https://www.ethereum.org.

[10] Jepsen. http://jepsen.io/.

[11] Kudu. https://kudu.apache.org/.

[12] Logcabin. https://github.com/logcabin/logcabin.

[13] Namazu. http://osrg.github.io/namazu/.

[14] Personal Communication with ZooKeeper Developers (Michael Han, Patrick Hunt, and Alex Shraer).

[15] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/.

[16] Why Amazon's cloud Titanic went down. https://money.cnn.com/2011/04/22/technology/amazon_ec2_cloud_outage/index.htm.

[17] Daniel Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, March 2009.

[18] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal Dynamic Partial Order Reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[19] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging Distributed Systems: Challenges and Options for Validation and Debugging. In *Communications of the ACM (CACM)*, 2016.

[20] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013.

[21] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the 2011 EuroSys Conference (EuroSys)*, 2011.

[22] J. R. Burch, E. M. Clarke, K L. McMillan, D L. Dill, and L J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[23] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[24] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[25] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[27] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[28] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[29] Maria Christakis, Peter Muller, and Valentin Wustholz. Guiding Dynamic Symbolic Execution toward Unverified Program Executions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[30] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *10th International Conference on Computer Aided Verification (CAV)*, 1998.

[31] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 1994.

[32] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying Systems Rules Using Rule-Directed Symbolic Execution. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2007.

[35] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[36] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. In *Proceedings of the ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[37] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[38] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed Explicit Model Checking with HSF-SPIN. In *International SPIN Workshop on Model Checking of Software (SPIN)*, 2001.

[39] E. Allen Emerson, Somesh Jha, and Doron Peled. Combining Partial Order and Symmetry Reductions. In *The 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1997.

[40] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[41] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A Study of the Internal and External Effects of Concurrency Bugs. In *DSN*, 2010.

[42] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[43] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[44] Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. volume 1032, 1996.

[45] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997.

[46] Patrice Godefroid. Between Testing and Verification: Software Model Checking via Systematic Testing (Talk). In *Haifa Verification Conference (HVC)*, 2015.

[47] Patrice Godefroid and Sarfraz Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2002.

[48] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[49] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. In *Communications of the ACM (CACM)*, 2012.

[50] Alex Groce and Willem Visser. Model Checking Java Programs using Structural Heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[51] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[52] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[53] 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. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[54] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[55] James Hamilton. On Designing and Deploying Internet-Scale Services. In *Proceedings of the 21st Large Installation System Administration Conference (LISA)*, 2007.

[56] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[57] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[58] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[59] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: A concurrency bug benchmark suite. In *The 3rd USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2011.

[60] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated Atomicity-Violation Fixing. In *PLDI*, 2011.

[61] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. SETSUDO : Perturbation-based Testing Framework for Scalable Distributed Systems. In *Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[62] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On Fault Resilience of OpenStack. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[63] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[64] Pramod V. Koppol and Kuo-Chung Tai. An incremental approach to structural testing of concurrent software. In *ISSTA*, 1996.

[65] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[66] Leslie Lamport. The part-time parliament (paxos). *ACM Transactions on Computer Systems*, 16(2), May 1998.

[67] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[68] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[69] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A Characteristic Study on Failures of Production Distributed Data-Parallel Programs. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013.

[70] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 28th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[71] Thomas A. Limoncelli and Doug Hughe. LISA '11 Theme – DevOps: New Challenges, Proven Values. *USENIX ;login: Magazine*, 36(4), August 2011.

[72] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[73] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[74] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[75] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[76] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[77] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[78] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 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)*, 2008.

[79] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[80] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[81] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[82] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[83] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized Testing of Distributed Systems with Probabilistic Guarantees. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.

[84] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace Aware Random Testing for Distributed Systems. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019.

[85] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *ASPLOS*, 2011.

[86] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[87] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[88] Cesar Rodriguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based Partial Order Reduction. In *Proceedings of the 26th International Conference on Concurrency Theory (CONCUR'15)*, 2015.

[89] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, 2010.

[90] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[91] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM TOCS*, 1997.

[92] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing Faulty Executions of Distributed Systems. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[93] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, 2008.

[94] Benjamin H. Sigelman, Luiz Andr Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[95] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV)*, 2010.

[96] Jiri Simsa, Randy Bryant, Garth A. Gibson, and Jason Hickey. Scalable Dynamic Partial Order Reduction. In *The 3rd International Conference on Runtime Verification (RV)*, 2012.

[97] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 2010.

[98] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *FTCS*, 1992.

[99] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[100] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. Verdi: A framework for formally verifying distributed system implementations. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[101] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in MapReduce Considered Harmful? An Empirical Study on Non-commutative Aggregators in MapReduce Programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[102] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[103] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[104] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software*. In *International SPIN Workshop on Model Checking of Software (SPIN)*, 2007.

[105] Jie Yu. A collection of concurrency bugs. https://github.com/jieyu/concurrency-bugs.

[106] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *The 2nd Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[107] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 2010 EuroSys Conference (EuroSys)*, 2010.

[108] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-Intrusive Failure Reproduction of Distributed Systems using the Event Chaining Approach. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.