AMC: DEEP EXPLORATION OF DISTRIBUTED CONCURRENCY BUGS WITH A HIGHLY SCALABLE MODEL CHECKER

A DISSERTATION SUBMITTED TO THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES IN CANDIDACY FOR THE DEGREE OF MASTER’S

DEPARTMENT OF COMPUTER SCIENCE

BY

JEFFREY F. LUKMAN

CHICAGO, ILLINOIS 2017
To The Almighty God, Hueynie, and my family.
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. vi

LIST OF TABLES ................................................................. vii

ACKNOWLEDGMENTS ............................................................. viii

ABSTRACT ........................................................................... x

1 INTRODUCTION ................................................................. 1

2 BACKGROUND AND MOTIVATION ........................................ 5
   2.1 DMCK Primer ............................................................... 5
   2.2 Deep DC Bugs ............................................................. 7
   2.3 State of the Art ............................................................ 8

3 AMC ................................................................................. 11
   3.1 State Symmetry ........................................................... 11
   3.2 Disjoint-Update Independence ........................................ 14
   3.3 Parallel Flips ............................................................ 16
   3.4 No Zero-Impact-Crash Reordering ................................. 18
      3.4.1 No Crash-after-Discard Ordering .......................... 19
      3.4.2 No Consecutive-Crash Flips ............................... 19

4 PROGRAM ANALYSIS SUPPORT .......................................... 21

5 IMPLEMENTATION AND INTEGRATION ................................. 23

6 EVALUATION ................................................................. 25
   6.1 Methodology ............................................................ 25
   6.2 niDPOR and SAMC .................................................... 28
   6.3 AMC Speed in Finding Bugs ...................................... 28
   6.4 AMC Path Reduction ................................................. 29
   6.5 New Bugs ............................................................. 31

7 DISCUSSIONS ................................................................. 32

8 RELATED WORK ............................................................. 34

9 CONCLUSION AND FUTURE WORK .................................... 35
   9.1 Conclusion .............................................................. 35
   9.2 Future Work ........................................................... 35
      9.2.1 CompleteMC .................................................... 36
      9.2.2 AMC .......................................................... 36
9.2.3 MonitorMC ......................................................... 37

10 APPENDIX ............................................................. 38
   10.1 Bug Descriptions .................................................. 38
   10.2 Reduction Policies Pseudo-Code ............................... 41
   10.3 Program Analysis Pseudo-Code ............................... 45

REFERENCES ............................................................ 52
LIST OF FIGURES

2.1 DMCK. The figure is explained in Section 2.1. ........................................... 6
2.2 Deep DC bugs in Cassandra and ZooKeeper. The bugs are described in §2.2 and in
more detail in §10.1. .................................................................................................. 7

3.1 Communication symmetry. The figure is explained in the “Problem” part of Section 3.1. 12
3.2 State symmetry. The figure is explained in the “Intuition” part of Section 3.1. ............. 12
3.3 Parallel flips. Figures (a) and (b) are explained in the “Problem” segment of §3.3 and figure
(c) in the “Reduction algorithm” segment of §3.3. .................................................... 17

6.1 Scalability of dmck reduction algorithms (niDPOR, SAMC, and AMC). The figure
format is explained in §6.1. The x-axis represents the number of remaining steps to hit the bug
and the y-axis the number of paths explored to hit the bug (the lower the better). The niDPOR
and SAMC lines are discussed in §6.2 and AMC lines in §6.3. ..................................... 26
6.2 Path explosion and reduction. The figure is explained in §6.4. The y-axis represents
the number of generated (to-be-explored) paths over time. The maximum time in the x-axis
is around 3.5 days, after 5000 paths have been explored. AMC reduces the path explosion
problem further by two orders of magnitude from niDPOR and SAMC. ...................... 29
LIST OF TABLES

2.1 **State-of-the-art DMCKs.** The table is described in Section 2.3. “■” denotes black-box approach; “□” white-box approach; “Ind.” independency; “Sym.” symmetry; “X” number of crashes injected; “N↑” number of reboots; “✚✚‖” more generalization.

6.1 **Bug benchmark.** The table lists the bugs to benchmark dmck scalability. “CASS” represents Cassandra, “ZOOK” ZooKeeper, “RAFT” Raft LogCabin [9, 51], and “MAPR” Hadoop MapReduce. “LE” stands for leader election. #S, #C, and #R stand for #Steps, #Crashes, #Reboots.
I always believe that an achievement in life is a result of hard work, the surrounding people that support him, and God who provides all that are needed to reach the goal. In this section, I would like to acknowledge all the people that have supported me to finish my master thesis.

I sincerely would like to thank Professor Haryadi S. Gunawi who is my advisor, one of my role models and the person who opened up the way for me to pursue my highest education. Without his initial action in starting a Computer Science research group in Indonesia, I will not have a chance to taste the best quality of education in University of Chicago, USA. Not only that, all of his advices related on how to work effectively, how to prioritize things in life, how to focus on our goals, and how to raise up our family have shaped me to become the better version of myself.

I also want to thank Professor Shan Lu and Professor Ben Zhao who have become the committee for my master thesis. Their comments and constructive criticisms have greatly improved this master thesis. Furthermore, I am also grateful for my collaboration with Professor Shan Lu for a couple of projects. I always admire her story-telling skill, her approach on raising up questions and finding patterns in existing problems.

Another special person that has guided me so far through my research journey is my senior, Tanakorn Leesatapornwongsa. Because of his foundational work and our intensive discussions, together we might be able to extend his previous work to be a better research work. I would also thank Huan Ke, Cesar Stuardo, Riza Suminto, Dikaimin (Surya University), and Satria Priambada (Bandung Institute Technology), without all of them, this thesis will not be as strong as it is. And for the other UCARE members – Mingzhe Hao, Huaicheng Li, Michael Tong and Shiqin Yan – I want to say thank you for the friendship that we have.

Next, I want to thank my wife, Hueynie. Without her presence, support, encouragement, and prayer along the way, I won’t be able to finish this thesis. I will always remember her willingness to stay awake with me through all of my late night times of work. Also thank you to my father, my mother, papa Hariono, mama Naniek, Natascha, Daniel, Dave, Timothy, Andhika, Hueyna and...
Zion for all of their supports and precious prayers for me.

Last but not least, I want to give all the glory and praise to God who has strengthen me and provided all the resources and the right people around me, so that this thesis can be finished.
ABSTRACT

Reliability and availability are essential for cloud services to succeed. Unfortunately, building flawless cloud systems is hard due to latent distributed concurrency (DC) bugs. Distributed System Model Checking (dmck) helps developers to systematically and quickly expose DC bugs by testing the implementation code. However, one challenge exists for dmck: path explosion challenge. The state-of-the-art dmcks such as MoDIST, dBug, CrystallBall, and SAMC handle this challenge by implementing reduction algorithms. Although these dmcks have shown some early successes, we found that these dmcks are not scalable enough to handle deep DC bug scenarios. We present AMC, an automated, fast and scalable dmck that focuses on finding deep DC bugs. AMC implements four new path reduction algorithms that are generic and backed up by static analysis: state symmetry, disjoint-updates independence, parallel flips and no zero-crash-impact reordering. As a result, AMC outperforms existing dmcks reduction algorithms by one to two orders of magnitude. AMC is integrated with 5 real-world systems (a mix of open-sourced, production, and proprietary systems) and overall has successfully reproduced 9 bugs and found 7 new bugs.
CHAPTER 1
INTRODUCTION

Cloud-scale systems such as distributed computing frameworks [19, 49, 68], storage systems [2, 28, 41, 57], lock services [13, 37], and cluster managers [12, 36, 63] are the backbone engines of modern applications. Their complexities and intricacies however make them hard to get right. One notorious issue is the problem of “distributed concurrency (DC) bugs” [44], which are caused by concurrent distributed events occurring in non-deterministic order. They cause harmful consequences in production systems including failed jobs, node/cluster unavailability, data loss and inconsistency [33, 44].

Many approaches have been proposed to combat DC bugs [11], including testing [56], theorem proving [35, 50, 64], record and replay [26, 45], tracing [48, 54, 58], failure/log diagnosis [55, 65], which all have pros and cons (more in §8). Conversations with developers suggest that they prefer methods that are systematic (not random), but fast in exposing bugs, and able to directly test the implementation code, not the model [1].

One technique that fits the bill is implementation-level software distributed system model checker (or “dmck” for short), which will exercise all possible reorderings of non-deterministic events such as messages and fault timings, hereby pushing the target system into unexplored states and potentially revealing hard-to-find bugs. A few dmcks have been successfully used for production systems [10, 34, 43, 67].

One nemesis of dmck is the path explosion problem; for example, with 10 concurrent events \(ab..ij\), a naive dmck might exercise \(10!\) (factorial) unique paths \((ab..ji, aj..bi, \text{and so on})\). To tame this problem, existing dmcks employ path reduction algorithms such as dynamic partial order reduction (DPOR), [59, 66, 67], dynamic partial inference [34], and symmetry [43] (more in §2.3).

Despite these early successes, existing dmck algorithms are not scalable enough to quickly find deep DC bugs. Some real-world bugs require more than 50 events to be reordered or only surface when multiple crashes and reboots are injected at specific timings. With such depth, the path
explosion problem becomes untameable. Yet, for mature production systems that are generally robust in the “first order” (with common orderings) [1], it is the deep bugs that the developers would like to unearth. In this context, we elaborate our motivations:

First, while most dmcks successfully reproduced some bugs, the papers did not report their depths [40, 59, 66]. Although they successfully reduce the state space, “significant state-space reduction does not automatically translate to proportional increases in bug-finding effectiveness” [34, §5.3]. Thus, dmck scalability needs to be evaluated with deep, complex workloads (e.g., three concurrent Paxos updates, or multiple atomic broadcasts interleaved with two rounds of leader election).

Second, time matters. With shallow bugs/workloads, the dmck can perhaps find the bugs in one or few days, a tolerable testing time. With deeper scenarios, tens to hundreds of thousands of paths need to be exercised. Unfortunately, exercising a path takes time; only 1500-2000 paths can be explored in a day (§2.2). The problem is that many existing works mainly focus on the state-space reduction but not the absolute testing time.

Finally, existing dmcks are not backed by program analysis, thus their algorithm implementations must be simple and safe (i.e., do not accidentally skip unique paths that would lead to unexplored states). Thus, program analysis is needed to support more advanced and scalable algorithms.

The motivations above raise a new urgency for more scalable algorithms. To this end, we present AMC, an automated, fast, and scalable dmck. The key to AMC’s scalability is in its four novel reduction algorithms as summarized below. The intuitions behind our algorithms were the results of analyzing thousands of paths generated from deep workloads.

(1) State symmetry: Commonly in distributed 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 the messages, irrespective of the node IDs. Thus, path reorderings that would result in symmetrical state transitions can be eliminated.
(2) **Disjoint updates:** While state symmetry significantly omits symmetrical paths, many events must still be permuted within the non-symmetrical paths. From local-concurrency (LC) model checking literature [17, 30], one effective reduction technique is “disjoint-update independence.” Applying this to distributed systems, we observed that many concurrent messages update disjoint sets of variables, hence no need to be reordered.

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

(4) **No zero-impact-crash reordering:** The three techniques above are mainly about message reordering, but some DC bugs require crash injections to surface. Reducing the number of unnecessary crashes in dmck is relatively a new problem without many solutions. We found opportunities to eliminate zero-impact crashes at certain timings such as crash after a “discard” message (§2.1) or flips of consecutive crashes.

All of our methods above are based on sound model checking principles such as DPOR [24] and symmetry [15], thus our algorithms are systematic, not random or bug specific. Throughout the paper, we will describe the challenges in safely implementing the algorithms above and how our simple program analysis addresses the challenges and automatically assists the construction of the algorithm. Collectively, our algorithms can find DC bugs one to two orders of magnitude faster than state-of-the-art algorithms used in modern dmcks such as MoDIST [67], dBug [59], CrystalBall [66], and SAMC [43].

We built AMC in 10,175 LOC and integrated it to 1 academic and 4 production systems (Cassandra [41], ZooKeeper [37], Hadoop [57], Raft LogCabin [51], and a proprietary system “X” of a large technology company). Within these systems, we model checked 5 protocol implementations: Paxos, leader election, atomic broadcast, snapshot and task management protocols. AMC successfully reproduced 9 old bugs and found 7 new critical DC bugs, all confirmed by the developers.
In next sections, we present background and extended motivation (§2), AMC algorithms, program analysis (§3-4), AMC implementation and evaluation (§5-6), discussion, related work, and conclusion (§7-9.1).
CHAPTER 2
BACKGROUND AND MOTIVATION

2.1 DMCK Primer

This section describes how software/implementation-level distributed system model checker (dmck) works, as illustrated in Figure 2.1, along with related terms.

**DC bugs:** A goal of dmck is to catch distributed concurrency (DC) bugs, caused by distributed events that can occur in non-deterministic order. An event can be a message arrival, crash, or reboot. Dmck controls the timings of such events by inserting “hooks”. For example, the four concurrent messages \( a_1 \), \( a_2 \), \( b_1 \), and \( b_2 \) in Figure 2.1, are all being intercepted and waiting to be enabled by the dmck server. DC bugs are different from local concurrency (LC) bugs, which happen locally within a node due to thread interleaving. However, as each node in a distributed system can run multiple threads, it is possible that a bug is induced by both DC and LC (more later in §7). Our work only focuses on DC.

**Workload/input:** With a dmck, we can test some target protocols (e.g., leader election, Paxos) with some workloads (e.g., three concurrent key-value updates or an update while scaling out the cluster). In Figure 2.1, the workload generates the four concurrent messages. In addition, the tester can tell the dmck the maximum number of crashes/reboots to be injected. The dmck will permute the crash/reboot events in space (crashing different nodes) and time (crashing at different timings).

**Steps (enabled events):** After intercepting all in-flight messages to a wait queue, the dmck then enables one event at a time (which we also call a “step”). An enabled event can make the system generates new events, which will be intercepted again by the dmck. For example, enabling \( b_1 \) to node \( B \) might generate a response \( a_3 \) to node \( A \) (not shown). The dmck then waits briefly for the system to quiesce and then enable the next event. This whole process repeats until the termination point – when a specification is violated (e.g., inconsistent commits across the nodes) or the workload ends without any violation. A message label (e.g., \( b_1 \)) implies the hash value of...
the message content as well as the source and destination nodes.

**Paths and the explosion problem:** A unique sequence of steps forms a *path* (*e.g.*, $a_1a_2b_1b_2...$). Given a previously exercised path, the dmck will permute the possible reorderings and restart the workload; for example, in the next run, it will *flip* $b_2$ before $b_1$, hence exercising a new path $a_1a_2b_2b_1$ under the same workload. Paths can also contain crash/reboot events; for example, a path $a_1\mathcal{A}A\mathcal{T}a_2b_1b_2$ implies a crash $\mathcal{A}$ and a reboot $\mathcal{T}$ on node $A$ are injected after $a_1$ is processed but before $a_2$ arrives. As dmck must permute all the possible reorderings, dmck faces the *path explosion* problem. In a naive method, $E$ concurrent events will generate $E!$ (factorial) paths to exercise. The more complex the workload, the larger number of events must be permuted.

**Independency-based reduction:** To tame the path explosion problem, dmck can 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).

One core foundation of reduction is the concept of dependency and independency in *dynamic partial order reduction (DPOR)* [24, 29]. 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 explore new state transitions. 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, exercising one of the reorderings is sufficient, while the other reordering is unneccessary/redundant.

A simple example of independency is “*discard messages.*” Let us imagine that we know both messages $a_1$ and $a_2$ will be discarded by node $A$ when the system is at $S_i$ (*e.g.*, the messages are
old votes or lower ballot numbers). In this case, both $a_1b_2$ and $a_2a_1$ would lead to the same state $S_i$, thus one of the reorderings can be skipped.

Symmetry-based reduction: Another major foundation of reduction algorithms is symmetry [22, 61], which exploits the topological symmetry present in the target system (e.g., when model checking a uniform set of multi-processors or ring of nodes).

Causal dependency: Dmck must also track causal dependency by using vector clocks. In the example above, if after $a_1$ is enabled, the dmck intercepts $c_1$, then $c_1$ is causally dependent on $a_1$. 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. To not confuse “causally dependent” with the “(in)dependent” model checking term, we will use “reliant/conditional” to represent the former.

### 2.2 Deep DC Bugs

Figure 2.2 illustrates two real-world deep DC bugs that cause a permanent inconsistency across the nodes. The first one, bug# Cassandra-6023 in Cassandra Paxos [4], requires three concurrent key-value updates, resulting in 54 events (steps) to hit the bug. The bug is also “deep” because two important flips must happen: the prepare message with ballot number 2 must happen exactly before the commit with ballot 1 and then the prepare with ballot 3 must happen exactly before the propose with ballot 2. Note that these two flips must happen within all the possible flips of the 54 events. The second one, bug# ZooKeeper-335 in ZooKeeper [5], requires 46 steps (from 3 crashes and 3 reboots, along with two incoming transactions), hence a complex concurrency between the
Table 2.1: **State-of-the-art DMCKs.** The table is described in Section 2.3. “■” denotes black-box approach; “□” white-box approach; “Ind.” independency; “Sym.” symmetry; “N” number of crashes injected; “N↑” number of reboots; “+A” program analysis support; and “+G” more generalization.

<table>
<thead>
<tr>
<th></th>
<th>■</th>
<th>□</th>
<th>Ind.</th>
<th>Sym.</th>
<th>X</th>
<th>N↑</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACE MC [40]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>CrystalBall [66]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>dBug [59]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>MoDIST [67]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>SAMC [43]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AMC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

ZooKeeper atomic broadcast (ZAB) and leader election (LE) protocols.

We show later in the evaluation that existing dmcks are not scalable enough to find such deep bugs. Scalability significantly matters for dmck because it takes 40-60 seconds to just exercise one path. This is because software/implementation-level model checking, especially for stateful distributed systems, requires many time-consuming tasks (back-and-forth communications between the target system and the dmck server, wait time to quiesce, crashing/rebooting the nodes restarting the whole system from a clean state, etc.). Thus, roughly in a day, a dmck can only exercise 1500-2000 paths.

### 2.3 State of the Art

Table 2.1 summarizes the state of the art. First, MACE MC [40] employs a combination of depth first search (DFS) and random walk, biased with weighted (prioritized) events. The approach is not systematic (due to the random walk) and enforces testers to manually label event weights.

Subsequent works, CrystalBall [66], MoDIST [67], and dBug [59], began to apply the concept of DPOR’s independency, specifically: “a message to be processed by a given node is independent of other concurrent messages destined to other nodes [hence, not interleaved]” which we will refer as the “**node-independent DPOR**” rule. Using the example in Figure 2.1, while DFS will create 4! paths, this rule only generates, $a_1a_2b_1b_2$, $a_1a_2b_2b_1$, $a_2a_1b_1b_2$, and $a_2a_1b_2b_1$, giving a 6X path reduction, as $a_1a_2$ are independent of $b_1b_2$. This rule can be implemented in a black-box
manner (“■” in Table 2.1), without access to the target system’s code.

SAMC [43] improves upon the node-independent DPOR rule by using semantic/white-box (□) information (e.g., message processing code) to define more events that are actually independent. SAMC suggests that one can know \textit{apriori} how a message will be processed by analyzing the if-else statements in the target code. For example, given a \textit{discard code pattern} such as “\texttt{if (msg.vote < state.myLargestVote) \{ /*discard*/ \}}”, then one can identify discard messages and mark them as independent (§2.1). In addition, SAMC also employs crash and reboot symmetry (e.g., crashing a follower node $A$ is the same as crashing another follower node $B$, if they are at the same state $S_i$).

There are other dmcks not shown in Table 2.1 such as dynamic-interface reduction (DIR) [34] and LMC [31], but they are more about decoupling local and global explorations as they target LC and DC (e.g., different thread interleavings often lead to the same global events that do not have to be repeated). Thus, these works are orthogonal to our work wherein we focus on pruning message reordering. Another recent work suggested “\textit{parallel DPOR}” [60]. In our opinion, parallelizing dmck only helps a little because parallelism (scaling out nodes) is linear, while the path explosion problem is exponential.

In summary, we highlight three issues: (1) Most of existing dmcks are based on simple, black-box reduction algorithms (“■” in Table 2.1) and do not leverage both independency and symmetry. (2) As a result, they are not scalable to find deep DC bugs. For example, with only the node-independent DPOR rule, 6 bugs in our benchmark cannot be reached even after 5 days. Moreover, the $\mathcal{N}$(crash) and $\mathcal{N}^\uparrow$(reboot) columns of Table 2.1 also suggest that prior dmcks were not evaluated with crash and reboot injections (to limit the path explosion), however almost 50% of DC bugs require at least one crash to surface, 12% require two or more crashes, and 25% require at least one reboot [44, Figure 3]. (3) Finally, the reduction algorithms are entirely written manually by the tester and not backed by program analysis (hence the simplicity of the algorithms). Because of this reason, SAMC, which opens up the concept of white-box reductions, only introduces
“cautious” reduction algorithms. On the other hand, AMC, as backed by program analysis (“$^+$” in Table 2.1), is able to employ more advanced reduction algorithms that generalize (“$^+$”) SAMC’s independency- and symmetry-based reductions.
CHAPTER 3
AMC

As mentioned in previous chapter, the major foundations of fast model checking are independency and symmetry, but they must be applied uniquely to the target domain, in our case, distributed systems. Thus, to find more independency and symmetry relationships in distributed systems, we executed existing algorithms (e.g., MODIST, SAMC) with complex bug benchmarks in different real-world systems (more later in §6.3), and analyzed thousands of paths that are generated to hit the bugs.

As a result, we found deeper and more interesting independent and symmetrical natures of distributed systems that can be exploited in new reduction algorithms, which led us to the development of AMC. Within AMC, we introduce four novel reduction algorithms: state symmetry (§3.1), disjoint-update independence (§3.2), parallel flips (§3.3), and no zero-impact-crash reordering (§3.4). We will show how our algorithms generalize many other algorithms in existing dmcks, hence the increased capability to remove more unnecessary reorderings. However, more generalizations and more reductions require more cautions to make sure unique orderings are not accidentally skipped. For this, AMC is powered with a static analysis (which we will explain in detail later in Section 5).

Below we describe each of the AMC algorithms in the following format: (a) the specific path explosion problem being solved, (b) the intuition for reduction, (c) the reduction algorithm in a high-level description, and (d) the challenges to make the algorithm and the implementation safely remove truly redundant reorderings. The pseudo-code of our algorithms can be found in §10.2.

3.1 State Symmetry

- **Problem:** Let us imagine a simple communication in Figure 3.1a where message \( k \) triggers \( l \) and \( x \) triggers \( y \). Figures 3.1b and 3.1c show two possible reordered paths \( klx \) and \( xykl \). While the
Figure 3.1: **Communication symmetry.** *The figure is explained in the “Problem” part of Section 3.1.*

![Diagram of communication symmetry](image)

Figure 3.2: **State symmetry.** *The figure is explained in the “Intuition” part of Section 3.1.*

![Diagram of state symmetry](image)

paths seem to be different, their communication structures in Figures 3.1b-c hint a possibility for symmetrical reduction.

A method to implement symmetrical reduction is to abstract the system property [15, 16, 61]. We initially attempted to abstract only the communication structure, specifically by abstracting the sender and destination node IDs to a canonical receiving order; for example in Figure 3.1b, as node B is the first to receive, its node ID is abstracted to node 1, and similarly in Figure 3.1c, as node A is the first to receive, its node ID is abstracted to node 1, hence establishing a symmetry.

Unfortunately, this approach is not effective because most messages carry a unique content. For example, in Figure 3.1, message k might carry ballot number 1 while x ballot number 2. Thus, while the arrows look symmetrical, only abstracting the messages does not lead to a massive reduction.

- **Intuition:** Fortunately, in many distributed 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 Figure 3.2a, which rep-
resents the first phase of a (simplified) Paxos implementation, given two concurrent updates from nodes $A$ and $B$. Node $A$ broadcasts its prepare messages, $a_1$ to itself and $b_1$ to node $B$ (bold lines), with “1” represents a ballot number 1. Similarly, node $B$ broadcasts $b_2$ to itself and $a_2$ to node $A$ (dashed lines), both carrying ballot number 2.

If we compare the two communication structures in Figure 3.2a and 3.2b, they are not symmetrical as in our previous example (Figure 3.1). However, let us analyze the global state transitions in the middle table in Figure 3.2. In this Paxos example, every node only accepts a higher ballot and discards a lower new incoming ballot, hence the nodes’ prepare status monotonically increases (e.g., from 1 to 2). In the left ordering $b_1a_2b_2a_1$, the state transition of node $A$ is 00222 and $B$ is 01122. In the ordering on the right, $a_1b_2a_2b_1$, the state transition is symmetrical (mirrored), with 01122 for $A$ and 00222 for $B$. To sum up, while communication structures of two paths are not symmetrical, their states transitions can be symmetrical, hence our state-symmetry algorithm.

- **Reduction algorithm:** We keep a history of $\{\text{absState}+\text{absEv}\}$ that have been exercised in the past; absState represents an abstracted global state (alphabetically ordered) and absEv implies an abstracted event that is enabled when the system is at absState.

  Using the example in Figure 3.2a, the first event will generate $\{00+1\}$ where 00 represents the abstracted state of nodes $A$ and $B$ (i.e., just the highest ballot number received, excluding the node IDs) and 1 represents the abstracted $a_1$ message (i.e., source and destination node IDs are removed). Subsequently, we record $\{01+2\}$, $\{12+2\}$, and $\{22+1\}$ to the history. Important to note that state 12 is from the alphabetically sorted state 21; symmetry implementation requires alphabetical/numerical sorting [43, §3.3.3].

  With this history, the second ordering $a_1b_2a_2b_1$ in Figure 3.2b will be marked symmetrical. For example, when $a_1$ is to be enabled (abstracted to +1) when the system is at abstract state 00, a historical match $\{00+1\}$ will be found. Similarly, for $b_2$ (abstracted to +2) when the system is at state 01, a match $\{01+2\}$ will be found.

  As another benefit, our state symmetry generalizes SAMC’s symmetry algorithm for crash and
reboot injections [43, §3.3.3–3.3.4]. Now, we do not need to have separate algorithms. More specifically, a crash is abstracted to a crash event targeted to a particular target node; for example, \{12+2\} implies a crash injected at the node with ballot number 2.

**Challenges:** 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 to and excluded from the abstracted information? For example, if the protocol turns out to be processing the sender IDs of the messages, then excluding sender IDs from the abstracted event is not safe, as this can incorrectly skip unique event reorderings.

To support this, our static analysis outputs a list of message variables that state transitions depend on. For example, for Cassandra Paxos, neither the sender nor destination (node) IDs are used by the protocol, hence are excluded from the abstracted information.

### 3.2 Disjoint-Update Independence

**Problem:** While state symmetry significantly omits symmetrical paths, there are still many events that need to be reordered within the non-symmetrical paths. For example, in Figure 3.1, we need to flip \( k-y \) (such that \( y \) happens before \( k \)) and \( l-x \), and in Figure 3.2, \( a_1-a_2 \) and \( b_1-b_2 \). To generalize the problem, if four messages \( a_1...a_4 \) are concurrent to node \( A \), the permutation will lead to \( 4! \) times more paths. In other words, all concurrent messages to every node are still reordered, hence more reductions are needed.

**Intuition:** From the literature of local-concurrency (LC) model checking, we learn that one of the most effective reduction techniques is “disjoint-update independence” [17, 30]. For example, if two concurrent threads (code blocks) do not access the same memory location, then they are disjoint, hence independent (*i.e.*, the corresponding instructions do not have to be interleaved, as no data race will ensue).

Similarly, in distributed systems, we observed that many concurrent messages (to a destination
node) can update different variables, and thus we introduce disjoint-update independence to AMC. For example, in 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.

- **Reduction algorithm:** For every message \( n_i \) to a node \( N \), our static analysis builds the live \( \text{readSet} \) and \( \text{updateSet} \), a set of to-be-read and -updated variables, given \( N \)'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 information, two messages \( n_i \) and \( n_j \) to a node \( N \) are marked independent if \( n_i \)'s \( \text{readSet} \) and \( \text{updateSet} \) do not overlap with \( n_j \)'s \( \text{updateSet} \), and vice versa.

As another benefit, disjoint-update independence generalizes SAMC's discard-based algorithm [43, §3.3.1]. SAMC reduces message-message reordering by skipping reorderings of discard messages (§2.1). For example, if \( a_1 \) and \( a_2 \) will be discarded by node \( A \), they are not reordered (as the state transition remains the same). SAMC however only prevents the flip of two messages when both are discard messages. But as we generalize a discard message as an empty \( \text{updateSet} \), such a message automatically does not conflict with any other messages, hence not need to be reordered. One might wonder why discard messages are not just removed from the paths, which we will explain at the end of this section.

- **Challenges:** While disjoint-update independence seems to be straightforward, two challenges must be addressed as we target distributed systems.

  First, as we target stateful distributed systems, two messages, \( n_i \) and \( n_j \), 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 log is updated but potentially in different orders. Thus, our static analysis also traces file APIs.

  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 $a_1$ will make $x=1$ and $a_2 y=2$. 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. $A$ is at state $\{x=0, y=0\}$
2. $A$ receives $a_1$, hence $x=1$
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 above scenario shows $a_1$ (2) is enabled before $a_2$ (4). If we (incorrectly) declare them as independent, $a_2 - a_1$ ordering will be skipped, thus we will never see $\{(y=2, x=0)\}$ broadcasted to $B$. If node $B$ has a logic such as “if $(y==2 \&\& x==0)$ panic(),” then we will miss this $a_2 - a_1$ induced bug.

For this reason, in addition to readSet and updateSet, we keep track the “sendSet,” the variables that are sent out after a message is processed. In the example above, 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.

### 3.3 Parallel Flips

- **Problem:** While our prior methods reduce message-message reordering to every node, in aggregate many flips must still be done across all the nodes. The problem is that in existing dmcks, to create a new reordered path from a prior path, only one pair of events can be flipped at a time. For example, in Figure 3.3a, two concurrent messages $a_1$ and $a_2$ are in flight to node $A$ and four messages $b_1...b_4$ to node $B$. Figure 3.3b illustrates how existing approaches flip one dependent pair of events at a time; for example, after exercising path (1), $a_1 a_2 b_1 b_2 b_3 b_4$, 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 happen before $a_1$). In the standard approach above, it will take $4!$ reordering (of the four messages to $B$) before we have the chance to flip $a_2$ before $a_1$.

**Intuition:** We observed such pattern when we analyzed our bug benchmarks. For example, as shown earlier in Figure 2.2a, a prepare message with ballot #2 must arrive before the commit message with ballot #1, but there are 8 earlier in-flight messages to other nodes that must be flipped. Worse, after that, a prepare message #3 must arrive before a propose message #2, while 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 independent pairs of events are allowed. While we created this algorithm based on our experience with Cassandra bugs, we found that parallel flips do *not* delay AMC in finding the other bugs in our benchmarks. Thus, parallel flips are not bug driven.

**Reduction algorithm:** For every new path, we will flip a pair of independent messages to every node. At most $#N$ parallel flips are made for a new path, where $#N$ is the number of nodes. For example, in Figure 3.3c, after exercising path (1) $a_1a_2b_1b_4$, in path (2) we will make both $a_2a_1$ and $b_4b_3$ flips at the same time. This is permissible because the in-flight messages to node $A$ are independent of those to node $B$ (per the “node-independent DPOR” rule in §2.3). Furthermore, in path (3), we revert back to single flips ($b_4b_2$) as no parallel flips are possible (permutation of $a_1a_2$ has completed).

More precisely, parallel flips are only allowed if none of the events within the flips are reliant
on one another. For example, let us consider $a_1a_2b_1b_2$ at the first case in the figure on the right, where $b_1$ is reliant on $a_1$, and $b_2$ on $a_2$. If we carelessly make the two flips $a_2a_1b_2b_1$, it is possible that $b_1$ will never happen because the different ordering $a_2a_1$ that node $A$ receives, as shown at the second case in the figure. In this case, the path $a_2a_1b_2b_1$ will “hang” and waste some execution time. Thus, correct parallel flips require the analysis of reliance (causal dependency).

- **Challenges:** Let us imagine a path $a_1a_2b_1b_2...$, and in a subsequent path, $a_2a_1b_2b_1...(from making parallel flips). The question is then whether $a_1a_2$ should be performed again. The answer is that if a flip $yx$ creates new reliant messages concurrent to existing messages to the same node, then the pair $xy$ needs to be repeated again in subsequent paths. For example, if $a_1a_2$ ordering will create $b_6$ and $a_2a_1$ will generate $b_7$(while $b_1$ and $b_2$ are outstanding), then $a_1a_2$ (hence $b_6$) must be repeated again in subsequent paths. This is because $b_6$ must still be reordered with other concurrent events to the same node (e.g., $..b_1b_6b_2.., ..b_6b_1b_2..$), which can only happen if $a_1a_2$ is performed.

### 3.4 No Zero-Impact-Crash Reordering

- **Problem:** Our three previous techniques are mainly about message reordering, but now we address crash reordering. DC bugs also linger in crash recovery paths. The TaxDC paper reports that almost 50% of DC bugs can only be revealed with at least one crash injected [44, Figure 3]. When crashes are injected, taming the path explosion problem becomes more challenging as more events need to be reordered (imagine $..a_1a_2\bar{A}.., ..a_1\bar{A}.., ..a_2\bar{A}..$, where “$\bar{A}$” denotes crashing of node $A$). Not to mention if multiple crashes need to be injected.

Reducing the number of unnecessary crash injections in dmck is relatively a new problem without many solutions in the field. All dmcks that we are aware of [34, 40, 59, 66, 67] did not optimize the crash-injection feature. SAMC introduced crash-injection symmetry [43, §3.3.3] (generalized in §3.1) and crash-message independence [43, §3.3.2]. In this work, we found more
opportunities for crash reordering reduction.

3.4.1 No Crash-after-Discard Ordering

- **Intuition/algorithm:** As crash events are reordered (injected at different timings), we noticed that a crash can be ordered after a discard message, which is unnecessary. For example, let us consider a path ...$a_1A...$, where $a_1$ is a message that will be discarded by node $A$, and after that, node $A$ will be crashed. This ordering $a_1A$ is unnecessary because $a_1$ will not create any state changes before crash $A$ is injected. Thus, given a crash $N$, if the last message $n_i$ to that node will be discarded, then the crash $N$ should not be injected after $n_i$.

The rule can be generalized from discard to “no IO” messages. That is, even if $n_i$ is not discarded but rather modifies $N$’s memory state, and as long as there is no network or disk IOs triggered by $n_i$ at node $N$, then the crash $N$ is not necessary. However, in our experience this generalization boils down to only discard messages; that is, only discard messages appear as instances of no-IO messages. Most of other messages will create some responses; for example, message $a_1$ triggers new messages $b_1$ and $c_1$ to other nodes, hence injecting crash $A$ after $a_1$ is necessary as the crash might trigger new recovery messages (e.g., $b_2$ and $c_2$) that would form interesting interleavings with the earlier in-flight messages (e.g., $a_1A...b_1c_2b_2c_1$).

3.4.2 No Consecutive-Crash Flips

- **Intuition/algorithm:** TaxDC reports that about 12% of DC bugs can only be revealed with at least two crash injections. When we injected multiple crashes, we noticed that many paths contain different orders of consecutive crash injections (e.g., ..$A\overline{B}..$, ..$B\overline{A}..$). Moreover, when more than two crashes are injected, the possible permutation exacerbates the path explosion problem. To address this, AMC does not flip consecutive crashes. For example, if a prior path already exercised $A\overline{B}$ when the system at state $S_i$, then we will skip $B\overline{A}$ at $S_i$. 

19
**Challenges:** In our simple rule above, if $AB$ was already exercised, $BA$ will be skipped. However, let us imagine the following unique case if $B$ happens first:

1. Crash node $B$.
2. $A$ detects the crash and logs the status ($A_{log}$).
3. Crash node $A$.

Now, let us consider that $AB$ already exercised and we decide to skip $BA$. In this case, we would never see the scenario above (i.e., the path $BA_{log}A$). This is unsafe because a dmck should traverse all unique state transitions. Therefore, our algorithm only skips a concurrent-crash flip where neither of the crashes lead to a “persistent” impact (disk writes) to the other node.

To obtain such information, our static analysis parses crash recovery paths and builds the “isPersist” predicate. For example, for not flipping $AB$, the $isPersist$ of node $A$ under $B$ condition must be false, and vice versa. That is, $isPersist$ of $A$ is false if crashing node $B$ does not lead to disk writes in node $A$. Similar to our discussion in the algorithm subsection of Section 3.2, the $isPersist$ predicate is not static but rather depends on the live state of the system when a crash is injected (e.g., $isPersist$ of $A$ for $B$ can be true when the system is at state $S_i$, but false when the system is at state $S_j$).
AMC is supported with a generic program analysis that automatically builds `readSet`, `updateSet`, `sendSet` (§3.2), and `diskSet` (§3.4).

**Annotation (input):** To use this static analysis, 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 [53, 67]. Annotating message class names (e.g., “MessageIn” in Cassandra) is relatively simple. Crash handling paths are typically in the catch blocks of failed network IOs, for example:

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

In addition, our program analysis also maintains a dictionary of disk IO library calls. On average, the annotation is only 19 LOC per target system.

**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 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) ...

Thus, with such output, we can track that the relationship of two concurrent messages $n_i$ to $n_j$ to node $N$. For example for disjoint updates (§3.2), if both messages are type A, they conflict and...
must be reordered. However if $n_i$ is of type A while $n_j$ is of type B, they exhibit disjoint updates and do not need to be reordered.

To create such output given the input annotation, our program analysis performs basic data- and control-flow analysis in 1799 LOC. The pseudo-code of our program analysis can be found in §10.3.
CHAPTER 5
IMPLEMENTATION AND INTEGRATION

We now describe several implementation details.

- **Complexity:** AMC is implemented in 10,175 LOC in Java, which includes all the features mentioned in Section 2.1, but the core algorithms (§3) are only 2420 LOC. To the best of our knowledge, only a few advanced dmcks are open sourced (SAMC [43], Namazu [10], and dBug [59]), while the rest are proprietary (MODIST [67], DIR [34]) or only work for restricted high-level languages (MACEMC [40], LMC [31]).

  We built AMC upon SAMC as the most recent dmck, but we only borrow 2838 LOC and rebuilt the rest, for the following reasons. First, SAMC does not use vector clock, thus causally dependent events are connected manually. AMC automatically builds the vector clocks. Second, SAMC only targets Java-based systems (e.g., using AspectJ [3] for hooks). AMC uses a more general file-based signals to enable portable communication between the target system (e.g., in Java or C++) and the dmck server. Note that they all can run on just one machine (where nodes are processes). Third, the data structures to capture the target system’s states and events are built on rigid classes. We generalize them into extensible key-value lists.

- **Other features:** AMC also comes with other features such as prefix-based evaluation (more in §6.1), deterministic replay of paths that hit the bugs, log analysis scripts for analyzing tens/hundreds thousands of generated paths (to help future researchers invent new algorithms), and unit tests for checking AMC algorithms. The last one is quite important as dmck itself is a complex system with many engineering challenges. The unit tests double check that AMC does not accidentally skip unique reorderings. In fact, with such testing, we found a bug that makes SAMC skips some flips of events that are actually dependent (more in §6.2).

- **Systems integrated:** We have integrated AMC to 5 systems: Cassandra [41], ZooKeeper [37], Hadoop MapReduce/Yarn [19, 63], Raft LogCabin [51], and a production system “X” of a large
company [anonymized]. Collectively, they are written in Java and C++. The hooks to a target system on average are 147 LOC and the workload setups on average are 75 LOC.

Within these systems, we specifically model check 5 unique protocol implementations: Cassandra Paxos, ZooKeeper leader election and atomic broadcast, Hadoop cluster management task assignment, Raft leader election and snapshot, and “X” leader election.

For some of these systems, for our evaluation of reproducing old bugs, we must port AMC to different target versions, for a total of 6 versions. Integrating AMC to a completely new system roughly takes 2 months, and roughly around 2 weeks to an older version of the same system.
CHAPTER 6
EVALUATION

We now evaluate AMC with the following questions: Can AMC reach deep DC bugs? Can AMC find bugs faster than state-of-the-art approaches? Can AMC unearth new bugs?

6.1 Methodology

<table>
<thead>
<tr>
<th>Alias Bug</th>
<th>Real Bug</th>
<th>#S</th>
<th>#C</th>
<th>#R</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASS-1</td>
<td>Cassandra-6013</td>
<td>30</td>
<td>–</td>
<td>–</td>
<td>Paxos</td>
</tr>
<tr>
<td>CASS-2</td>
<td>Cassandra-6023</td>
<td>54</td>
<td>–</td>
<td>–</td>
<td>Paxos</td>
</tr>
<tr>
<td>CASS-3</td>
<td>Cassandra-5925</td>
<td>15</td>
<td>–</td>
<td>–</td>
<td>Paxos</td>
</tr>
<tr>
<td>ZOOK-1</td>
<td>ZooKeeper-335</td>
<td>46</td>
<td>3</td>
<td>3</td>
<td>ZLE, ZAB</td>
</tr>
<tr>
<td>ZOOK-2</td>
<td>ZooKeeper-790</td>
<td>39</td>
<td>1</td>
<td>1</td>
<td>ZLE</td>
</tr>
<tr>
<td>ZOOK-3</td>
<td>ZooKeeper-1419</td>
<td>41</td>
<td>3</td>
<td>3</td>
<td>ZLE</td>
</tr>
<tr>
<td>ZOOK-4</td>
<td>ZooKeeper-1492</td>
<td>24</td>
<td>1</td>
<td>–</td>
<td>ZLE</td>
</tr>
<tr>
<td>RAFT-1</td>
<td>Raft-174</td>
<td>21</td>
<td>2</td>
<td>2</td>
<td>RaftLE, Snapshot</td>
</tr>
<tr>
<td>MAPR-1</td>
<td>MapReduce-5505</td>
<td>36</td>
<td>1</td>
<td>1</td>
<td>Task Assignment</td>
</tr>
</tbody>
</table>

Table 6.1: Bug benchmark. The table lists the bugs to benchmark dmck scalability. “CASS” represents Cassandra, “ZOOK” ZooKeeper, “RAFT” Raft LogCabin [9, 51], and “MAPR” Hadoop MapReduce. “LE” stands for leader election. #S, #C, and #R stand for #Steps, #Crashes, #Reboots.

We first describe our evaluation methodology.

**Bug benchmark:** Table 6.1 shows the bug benchmark that we use, including the number of steps, crashes, and reboots to hit the bugs. Most dmck papers did not report the evaluated bug depth [40, 59, 67], but it is important to pick deep DC bugs for scalability evaluation. The bugs are described in more detail in Appendix 10.1.

**Other techniques compared:** We evaluate AMC against (1) the node-independent DPOR reduction (“niDPOR” for short) used by many prior dmcks [59, 66, 67] (§2.3) and (2) SAMC algorithms [43, §3.3]. We did not compare with random/prioritization exploration (e.g., MACEMC’s random walk and weighted paths), as prioritization is not systematic and randomness is also not effective in finding deep DC bugs [43, §5.1]. We also did not compare with DIR [34] and LMC [31], as they are about LC+DC reductions (§2.3).
**Speedup metrics:** Our primary metrics of evaluation are the *number of paths to hit the bugs* and *the number of paths to explore* (the lower the better). Other prior dmcks used “the number of unique states explored” as a metric (the higher the better) [34, 59, 67], which is appropriate if only independency algorithms are used. But if symmetry algorithms are adopted, such a metric is no longer accurate as multiple “unique” states can actually be symmetrical (higher does not imply better).

![Graphs showing scalability of dmck reduction algorithms](image)

Figure 6.1: **Scalability of dmck reduction algorithms (niDPOR, SAMC, and AMC).** The figure format is explained in §6.1. The x-axis represents the number of remaining steps to hit the bug and the y-axis the number of paths explored to hit the bug (the lower the better). The niDPOR and SAMC lines are discussed in §6.2 and AMC lines in §6.3.

**Speedup/scalability presentation:** Figure 6.1 contrasts the scalability of niDPOR, SAMC, and AMC.

The x-axis of Figure 6.1 represents the number of *remaining steps* to hit the bug (with “path
prefix”). The maximum value in the $x$-axis represents the total number of steps to hit the bug without any prefix (same as the “#Steps” in Table 6.1). For example, for reproducing CASS-2 (Figure 6.1b), we need a total of 54 steps.

Path prefix is needed because some algorithms do not scale and cannot reach the bugs for days. Thus, we control our dmck server to execute in order some of the earlier steps (i.e., the path prefix) and let the rest be reordered. For example, in Figure 6.1b, with $x=10$, we control the first 44 initial steps and let the dmck reorder the last 10 steps to hit to the bug. We set up the path prefix based on the bug reports. Note that we use path prefix merely for evaluation purposes. Ultimately, dmck algorithms should scale without any prefix at all (no prior knowledge of the bugs).

The $y$-axis shows the number of paths explored until the bug is reached. For instance, in Figure 6.1a, at $x=26$, niDPOR must explore 505 paths to hit the bug, but SAMC and AMC are able to hit the bug in 55 and 18 paths respectively.

Exercising a path can take 40-60 seconds (§2.2). Thus, we stop the experiments when niDPOR or SAMC does not hit the bug after 10,000 paths (in 5 days). Thus, the solid lines in Figure 6.1 represent completed experiments, while the dashed lines imply that the bugs were not hit after 10,000 paths. Furthermore, due to the long experiment duration, we did not complete every $x$ value.

To summarize, the nature of the lines in Figure 6.1 is as follows. First, the more #steps to reorder (higher $x$ value) leads to more #paths to explore (higher $y$ value). Second, the lines with $y=1$ (only 1 path to hit the bug) within $x<X$ range is because the order of the remaining $X$ steps do not matter for the bug to manifest. Finally, the lines that “dip” down are because the dmck found other alternative similar paths to hit the bug besides the paths we deconstruct from the bug reports.

**Scale of evaluation:** All experiments ran on Emulab d430 machines (two 2.4GHz @ 8-core E5-2630 v3 Haswell with 64GB DRAM and 1TB SATA disk) [7] and on Chameleon CHI@UC cluster machines (two 2.30GHz @ 12-core E5-2670 v3 Haswell with 128GB DRAM and 2TB hard
drives) [6]. Overall, our extensive evaluation exercised over 110,000 paths and used approximately over 5000 full machine days.

### 6.2 niDPOR and SAMC

We first discuss the performance of niDPOR and SAMC. First, niDPOR (used in MoDIST [67], dBug [59], and CrystalBall [66]) is only scalable for ZOOK-2 and ZOOK-3 (Figures 6.1e-f) because the buggy scenario is similar with the first execution path that the DMCK produces, therefore it only needs a couple of event reorderings to catch the bug. However, niDPOR is not scalable for the rest of the bugs (the vertical spikes in the other figures). This is mainly because of the depth (> 20 steps) and crash/reboot inclusions, and deep messages reordering (e.g., as in 2 messages race in 2.2a).

SAMC [43], a more recent dmck, as expected, is scalable for most of the bugs, but not for CASS-2 and ZOOK-1 (Figures 6.1b and 6.1d), due to its limitations as discussed throughout Section 3. Bug CASS-2 was not used in SAMC’s evaluation, but ZOOK-1 was used and reached. However, we found an incorrect implementation in the SAMC’s manual causal-dependency tracker (§5) that accidentally skipped some unique reorderings (confirmed by the authors). When this problem is fixed, SAMC is not scalable enough for ZOOK-1, as there are more paths to explore. However, as an important note, this problem did not nullify SAMC’s advancement over niDPOR; as shown in Figure 6.1, as we fix the problem in both niDPOR and SAMC implementations, SAMC still outperforms niDPOR by one to two orders of magnitude (consistent with the claim in the SAMC paper).

### 6.3 AMC Speed in Finding Bugs

We now discuss AMC scalability in Figure 6.1 with the following observations.

First, although SAMC is scalable enough for most of the bugs, AMC, as it generalizes SAMC
algorithms, is more efficient. For example, for all of the bugs that can be reached by SAMC (within 10,000 paths before 5 days), AMC can reach them even faster by 1-6X (CASS-1, CASS-3, ZOOK-2, ZOOK-3, and ZOOK-4 in Figures 6.1a, 6.1c, 6.1e, 6.1f, and 6.1g).

Second, for CASS-2 and ZOOK-1, the two deepest bugs in our benchmark, AMC is much more scalable than SAMC. For example, in CASS-2 (Figure 6.1b), at x=33 steps, SAMC already “explodes” to more than 10,000 paths, while the maximum number of steps is 54. AMC on the other hand can hit the bug in 8894 paths without any prefix (x=54). Later in Section 6.4, we show in more detail the effectiveness of each of AMC algorithms for this workload. Similarly in ZOOK-1 (Figure 6.1d), SAMC also already “explodes” at x=38, but the maximum number of steps is 46. AMC on the other hand only needs to exercise 129 paths without any prefix (x=46). For this bug, the no zero-crash-impact and disjoint-update independence algorithms are the most effective because there are many discard messages that don’t need to be reordered with the crash events and furthermore the ZooKeeper Leader Election messages are disjoint with the Atomic Broadcast messages.

6.4 AMC Path Reduction

Figure 6.2: Path explosion and reduction. The figure is explained in §6.4. The y-axis represents the number of generated (to-be-explored) paths over time. The maximum time in the x-axis is around 3.5 days, after 5000 paths have been explored. AMC reduces the path explosion problem further by two orders of magnitude from niDPOR and SAMC.

Figure 6.2a depicts the number of generated, to-be-exercised paths (y-axis, in logscale) over time given the already exercised paths (x-axis), within the workload of bug CASS-2 (i.e., three concurrent Paxos updates, as depicted earlier in Figure 2.2a). The “explosion” in the beginning
shows the many possible permutations after the first path is exercised. Suppose the first path contains 14 concurrent events, the dmck will automatically generate 14! new paths ($8 \times 10^{10}$ paths). However, not all of them need to be exercised, as removed by the reduction algorithms. After the initial explosion, as more paths are exercised over time, the dmck will see new events which will again increase the number of paths to be exercised.

The highest dashed line in Figure 6.2a reflects the number of generated paths by a naive depth-first-search (DFS) algorithm without any reduction algorithm. The dashed line is an estimation, as we can only collect the hashes of $10^9$ paths given a 64GB DRAM (as depicted by the solid line before the dashed line).

The top region (“Reduced by niDPOR”) in Figure 6.2a depicts the number of paths that are reduced by niDPOR, roughly 3 orders of magnitude reduction from the DFS algorithm ($2 \times 10^{10}$ to $1.5 \times 10^7$), hence its popularity in prior dmcks [59, 66, 67]. Next, the middle region (“Reduced by SAMC”) shows that SAMC reduces the explosion further down to $10^7$ paths, roughly 19 machine years (given 1 minute/path, §2.2), hence the need for more reductions. This also suggests that paralleling dmck across multiple machines is not enough to tame the exponential path explosion.

Figures 6.2b-d depict AMC’s reductions, specifically by state symmetry, disjoint update, and parallel flip algorithms, respectively, each reduces the paths by roughly an order of magnitude. Crash-related reduction (§3.4) is not needed here because the workload does not require any crash to be injected.

Figure 6.2e shows that collectively (all algorithms combined) AMC provides two orders of magnitude of path reduction, to only $10^5$ remaining paths to explore (50 machine days). Note that a path can be reduced by multiple algorithms, hence the reason why the aggregate reduction is not three orders of magnitude.
6.5 New Bugs

To evaluate the effectiveness of AMC, we integrated AMC to (1) a production system “X” of a large company and (2) the latest version of ZooKeeper (v3.5.3). We chose these two systems as the developers were interested in using AMC. Note that both systems are mature production systems and all their single-pass unit tests did not find any DC bug. For system X, AMC successfully discovered 5 new confirmed critical DC bugs. Due to the proprietary nature of the system, we apologize we cannot share further details.

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. AMC successfully discovered 2 new bugs. The first bug reveals that the developers’ prior fix to an old DC bug was not robust enough, that there is another reordering that make an old bug surface. The second bug was reported to surface once every 500 unit test cycles. With AMC, we help the developers pinpoint the exact sequence of steps that make the bug reproducible deterministically. Due to this success, we have an ongoing communication with Cloudera’s Apache Kudu [8] developers to explore the possibility of model checking the Kudu’s Raft protocol.
CHAPTER 7
DISCUSSIONS

This chapter provides some discussions related to dmck and bug finding.

- **Why only crash injection and message reordering are exercised, but not timeouts/network partitioning?** As discussed in existing literature [44, §8.2], ideally all types of events (disk delays, local thread schedules, messages, crashes, reboot, and timeouts/network partitioning) must be controlled by dmck. However, this will create a larger path explosion problem, hence more possible future work to be done for creating a more complete dmck. We chose message reordering and crash/reboot injections as they represent a major reason for DC bugs that have been reported in production systems [44]. However, we also learned that in our target systems, network partitioning which will trigger timeouts exercises the same code paths as in crash handling scenarios.

- **Why is bug finding the focus of the evaluation?** In model checking research, arguably there are two objectives: state-space reduction and/or fast bug finding. Our evaluation focuses on the latter, although to achieve that, the former needs to happen as well. Our conversations with developers suggest that full system verification is still hard to achieve, but on the other hand they would like to unearth as many bugs as they can. In fact a recent work stated that “[their] experiences with Demeter on finding safety bugs are mixed, as significant state-space reduction does not translate automatically to proportional increases in bug-finding effectiveness” [34, §5.3]. In another recent work [20], technical leaders and senior managers of a large-scale cloud provider emphasized that “one of the most critical problems today is how to improve testing coverage so that bugs can be uncovered during testing and not in production.” Our take on these statements is that we need to reorder more events that matter (e.g., crash/reboot injections) that can reveal more bugs, while at the same time taming the explosion problem from the new types of events that are permuted.

- **Why do the implementations of formally-verified protocol designs have bugs?** In reality, distributed systems are complex, although they are built from formally-verified concepts such as
Paxos [18, 42], Raft [51], or Zookeeper Atomic Broadcast [37, 52]. We found that companies often build their own distributed systems from ground up, for many reasons including proprietary or patenting. In some cases, they “tweaked” the systems to make it fast, hence modified the actual design. An example, in the ZooKeeper case, while it is robust, the developers added the re-configurable ZooKeeper feature (§6.5) for elasticity purposes. Furthermore, what is verified is only the design of the protocol written in proofs. When writing the actual implementation, there can be human errors in translating the proven design.

• *Are the number of old and new bugs relatively reasonable?* The number of old bugs reproduced and new bugs found in our work is relatively “on par” with existing work [31, 34, 40, 59, 66, 67].

Note that we are targeting real-world production systems that are already heavily tested, while some of the earlier works only target “academic” distributed systems (*e.g.*, written in Mace language) [31, 40, 66]. More importantly finding more new bugs involve three matters: (a) more complex input workload, (b) fast path exploration and reduction, (c) more specifications to catch more violations. Dmck literature including our work focus on (b). Nevertheless, the developers of system “X” (§6.5) and ZooKeeper are already satisfied with what AMC can deliver. Given more time, and (a) more complex workloads are set up and (c) more specifications are written from the help of the developers, potentially more bugs can be found.
CHAPTER 8
RELATED WORK

We now briefly discuss other related approaches to distributed systems verification and testing.

There is a growing body of work on new verifiable programming frameworks/languages for distributed systems (e.g., IronFleet [35], PLang [21], Verdi [64]). Such methods are more formal than dmck, but developers must write proofs, which are typically in thousands of lines of code. A more recent work evaluates three formally verified distributed systems including IronFleet and Verdi and uncovered bugs in their implementations [25].

Post-mortem methods such as record-and-replay [27, 46] and flow reconstructions [55, 70] are popular methods to reverse engineer failures in customer sites. However, for DC bugs, these methods require fine-grained logging. ZooKeeper developers shared to us that they sometimes performed more than ten iterations of log changes over a long period of time to be able to replay the DC-related failures happening at the customer sites. The issue is that 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.

In addition to classical distributed systems (e.g., Paxos), more applications today are becoming distributed. A prime example is OpenFlow applications. A recent work, NICE [14], shows that dmck is an effective method for testing distributed OpenFlow applications.

Besides dmcks, there exists fault-injection testing frameworks for distributed systems (e.g., FATE [32], SETSUDO [38], OpenStack fault-injector [39]). This set of works advocates the need for multiple-fault injections, but they did not systematically permute non-deterministic choices such as message and crash timings.

Exacerbating the problem, cloud systems are becoming larger and geographically distributed [47, 62, 69]. We believe cloud systems will observe more failures and message reorderings, and therefore our work and future dmck advancements with the inclusion of failures will play an important role in increasing the reliability of future cloud systems.
CHAPTER 9
CONCLUSION AND FUTURE WORK

9.1 Conclusion

Uncovering bugs during testing (and not in production) is critical, as emphasized by senior technical leaders and managers [20]. In this thesis, we have presented AMC, a highly scalable software model checker that can explore deep bugs one to two orders of magnitude faster than state-of-the-art approaches by employing 4 new generic reduction policies: state symmetry, disjoint updates, parallel flips, and no zero-crash-impact reordering. With this reduction policies, AMC has successfully reproduced 9 real-world bugs in Cassandra, ZooKeeper, Raft, and Hadoop MapReduce; and it has also detected 7 new bugs in ZooKeeper and in a production system “X” of a large company.

Beyond AMC success in finding some new bugs, we believe our work triggers the discussion of other important research questions:

1. How can we fully control the timings of all necessary events, such as messages, crashes, reboots, disk delays, timeouts, and also local thread schedules?
2. How can the dmck learn about new reduction policies by itself?
3. How can we improve the dmck to help developers easily understand the root cause of the detected bug?

We hope that our current progress with AMC can lead to a further breakthrough in distributed system model checking area.

9.2 Future Work

In this work we have built a number of reduction policies that applied static analysis and explored more generic patterns of symmetry and events pair of independence that we called as State Sym-
metry, Disjoint Update Independence, Parallel Flips, and No Zero-Crash-Impacts Reordering in order explore deep bugs. There are three follow up questions to ask. First, how can we now incorporate more types of event such as timeouts and local thread events effectively? Second, how can the dmck learn new reduction policies by itself? Third, how can we explain the faulty scenario to the developers easily? We will address this questions in this section.

9.2.1 CompleteMC

While AMC is successful in reordering events that it controls, specifically, message, disk writing, crash, and reboot timings, future challenges in dmck research still remain. As elaborated in TaxDC [44], we also need to control the timing of local thread schedules and timeouts to cover all possibilities of DC bugs. But up until today, there is no dmck that is able to control all of the necessary events, including AMC.

To address this, we propose “CompleteMC”, an extension of AMC that controls all necessary events to explore all possibilities of DC bugs. Two additional types of events that need to be handled on top of AMC are timeouts and local thread schedules. In order to control timeouts, we need to add virtual clock feature, as implemented by MODIST[67]. Next, we also need to determine which local events are critical to be reordered with the other outstanding events. Once we add these two types of events under the dmck’s control, we need to come up with more various reduction policies to tame the state space explosions.

9.2.2 AMC

The biggest problem for dmck will always be the state space explosion. Every time we introduce a new event into the dmck, the numbers of paths that need to be explored will grow exponentially. Up until today, researchers go through all the dmcks path executions result and try to extract out reduction policies to tame the state space explosion. Our question is, can the dmck learn the reduction policies by itself advantaging machine learning technique?
To address this question, we propose “AMC” that stands for Automatic Model Checking. The main goal for this AMC is to learn reduction policies by itself while it goes through all the order of events that it has explored. As Daikon [23] has used simple machine learning technique to determine the invariants of a system by running a the code with some test suites, we might also utilize every executed paths to detect target system reactions’ pattern against each event that each node will execute. Therefore, the dmck should be able to construct a knowledge of reduction policies.

### 9.2.3 MonitorMC

Once we have caught a bug using the dmck, the next challenge that exists for the system developers is to correctly understand the root cause of the bug. Otherwise, the bug might not be fully solved and instead only the bug symptoms occurrence possibilities that has been reduced.

Existing work, such as ShiViz [11] proposed to visualize the target system logs based on the happens-before relationships. ShiViz achieved this by adding a layer of logical vector clock counter inside the cluster’s log system. Although their approach shows promising result, we think it will not work well for deep bug scenarios because there will be too many events that are recorded and visualized that might cause confusion for the developers. Furthermore, ShiViz requires developers to have the buggy scenario in the first place before any bug scenario can be visualized.

In order to help developers to easily understand the dmck caught bug, we propose “MonitorMC” which will be built on top of CompleteMC and AMC. When a dmck catches a bug, MonitorMC will collect the complete order of events that lead to the bug. Since CompleteMC already intercept all critical events from the target system, MonitorMC can automatically build the happens-before relationship, including the precise order of timing for each event execution and therefore, it can directly visualize the bug scenario that is needed to help developers easily understand their complicated and deep bugs.
CHAPTER 10
APPENDIX

10.1 Bug Descriptions

This section will focus on elaborating the DC bugs that AMC has successfully reproduced.

- CASS-1 (Cassandra-6013): (1) Node A sent first client request’s prepare (value = X) to node A, B, C and all nodes promised to accept it; (2) Node A sent propose (value = X) to node A, and accepted its propose, therefore node A inProgress = X. (3) Node B sent second client request’s prepare (value = Y) to node A, B, C; (4) All nodes promised to accept it, but because node A replied that its state inProgress=X, therefore node B updated the client request value from Y to X; (5) Node B sent propose (value = X) to node A, B, C and all nodes accepted node B’s propose. (6) Node A sent propose (value = X) to node B and C, but node B and C rejected it due to expired ballot number; (7) Node A told the client that its request failed to be saved. (8) Node B sent commit (value = X) to Node A, B, C; (9) Node B told the client that its request succeed, but the value that was saved among all the nodes is value X from first client request. Fix: update the check condition.

- CASS-2 (Cassandra-6023): (1) Client submitted 3 client requests almost simultaneously. (2) Node A sent prepare and propose for the first client request (value = X) to node A, B, C and all nodes accepted it; (3) Node A sent commit to node A, B and both committed the first client request. (4) Node B started to send prepare for the second client request (value = Y) and all nodes promised to accept it. At this moment, node A, B inProgress=null, while node B inProgress=X. (5) Node A sent commit to node C and node C committed first client request. But still, node C inProgress=X. (6) Node B sent propose to node A, B and both accepted the value. Their inProgress=Y; (7) Node B committed its inProgress=Y to disk. (8) Node C started to send prepare for third client request with different key to node A, B, C and all nodes accepted it. While node B inProgress=null, node A inProgress=Y and node C inProgress=X. (9) Node B sent propose to node C, which was rejected due to ballot number; (10) Node B sent commit to node A and C and they committed their
inProgress to disk. (11) Node C sent propose that contain inProgress=X and all nodes accepted it and committed it, causing the second client request to be overwritten. Fix: update check condition and message structure.

- CASS-3 (Cassandra-5925): (1) Node A sent prepare to node A, B and node A, B promised to accept it, therefore (2) Node A sent propose to node A, B, C and all nodes accepted it; (3) Node A sent commit to node A, B, C where node A and B successfully committed, but node C crashed because its inProgress ballot=null. Fix: always initialize ballot.

- ZOOK-1 (ZooKeeper-335): (1) Client wrote data X to node C who is the Leader; (2) Node C sent the client request to node A, B, C and all nodes accepted the proposal; (3) Node C wrote the data to disk; (4) But node A, B crashed before they wrote the data to disk. At this point A, B were in epoch 1, while node C has 2:X; (5) Node C lost his quorum and also crashed. (6) Node A, B rebooted and elected node B as Leader; (7) Client updated the data to Y to node B; (8) Node A, B wrote the data to disk as 2:Y. (9) Node C rebooted and sync with node B, where node B replied DIFF to node C; (10) But since node C was already in epoch 2:X, node C ignored node B DIFF and the cluster moved forward having permanent inconsistent data in epoch 2. Fix: add additional check condition and update logic.

- ZOOK-2 (ZooKeeper-790): (1) Node A, B, C agreed to elect node C as Leader; (2) Node C crashed before node C sent his confirmation to node A, B. (3) Node A, B reached timeout and re-done leader election, finally elected node B as Leader. (4) When node C rebooted, based on its stored state in disk, node C still saw itself as Leader. Therefore for some time, there existed two leaders in the cluster. Fix: add conditional check condition.

- ZOOK-3 (ZooKeeper-1419): (1) Node C crashed in the beginning. (2) Node A, B elected node B as Leader and their epoch=1. (3) Node A, B crashed. (4) Node B, C rebooted. (5) Node C would not accept node B as its Leader although its epoch is higher, because its nodeID is bigger; while node B would not be accepted as Leader because its ID is smaller than node C, causing the whole cluster unavailable. Fix: rewrite the check condition.
• ZOOK-4 (ZooKeeper-1492): (1) The cluster has following config: node A, B were set as PARTICIPANT, while node C as OBSERVER. (2) Node B has been elected as Leader; (3) Then node A who was the Follower, crashed. (4) Node B would not redo Leader Election and instead still saw itself as a Leader with valid quorum (although it was not), causing client to fail to connect with the cluster. *Fix:* add additional check condition.

• RAFT-1 (Raft-174): (1) Node C was crashed; (2) Node A, B elected node B as Leader; (3) Admin triggered snapshot for the cluster. (4) Node C rebooted and tried to sync with Leader; (5) In the middle of snapshot sending, node C crashed again. (6) Node C rebooted again, but this time it would keep crashing, because node B tried to continue sending the snapshot to node C, while node C forgot its first unfinished snapshot process. *Fix:* Leader will start over the snapshot process from the beginning.

• MAPR-1 (MapReduce-5505): (1) AM reported that it has finished its job to RM and reported to client; (2) RM crashed before saving AM report to disk and unregister AM. (3) RM rebooted; (4) RM saw that the job was not done and ran another AM, causing split brain condition. *Fix:* add additional check condition.

Next, we describe the new bugs scenario that AMC found in ZooKeeper.

• ZOOK-5 (ZooKeeper-2855): (1) There existed 3 nodes cluster with around 200MB of data. (2) Admin added a new node; (3) Node D directly synchronized and received SNAP from node C (Leader); (4) In the middle of the SNAP sending, the admin triggered reconfig to add node D to the cluster; (5) Node C (Leader) sent PROPOSAL and UPTODATE to nodes A, B, C, D and all nodes accepted the changes. (6) Afterwards, node C sent UPTODATE to node D. (7) When node D later crashed and rebooted, it would always failed to reboot because its config file was not appropriately updated, due to step 5 before step 6. *Fix:* add additional check condition.

• ZOOK-6 (ZooKeeper-2865): (1) The cluster initially had 2 nodes and the admin added 3 new
nodes into the cluster. (2) Admin triggered reconfig to add node C, D, E to the node A, B cluster which also changed port. (3) Node A, B, C, D accepted node B (Leader) reconfig request and all of them have changed their conf file; (4) But node E has not received the proposal from node B before, so node E still has the old config. (5) While the node A, B, C, D already moved to the next config and able to communicate to each other, node E was isolated because node E still tried to communicate to node B (previous Leader) in the closed port.

Fix: clarify the bug documentation, so that admin does not trigger reconfig too early.

We cannot describe new bugs that AMC has found in system “X” due to proprietary reasons.

In conclusion, these bugs happened because (1) untimely messages arrived in uncheck conditions, (2) unexpected failure timing was not handled in recovery protocol, or even (3) wrong timing of admin operation. Therefore, the developers fixed these issues by adding or fixing the conditional checking in the event handler or in the recovery protocol, update the logic of the code or lastly clarifying their documentation so that admin / client use the system accordingly.

10.2 Reduction Policies Pseudo-Code

This section explains the high level reduction policies’ algorithm. As a side note, Disjoint Update and No Zero-Crash-Impact policies are dependent on the AMC program analysis, therefore their algorithms only need to plugin the results from the program analysis. On the other hand, State Symmetry and Parallel Flips are generic reduction policies that do not need program analysis outputs.
Algorithm 1: State Symmetry Reduction Policy

Function isSymmetry (possibleOrders)
1. initGlobalState = null;
2. removePaths = [];
3. for path in possibleOrders do
4.     endGlobalState = propagateState(initGlobalState, path);
5.     if endGlobalState != null and globalStateDB.contains(endGlobalState) then
6.         removePaths.add(path);
7.     end
8. end
9. for removePath in removePaths do
10.    possibleOrders.remove(removePath);
11. end
12. return possibleOrders;

Function propagateState (initGlobalState, path)
13. if initGlobalState == null then
14.     currentGlobalState = generalInitialState;
15. else
16.     currentGlobalState = initGlobalState;
17. end
18. for event in path do
19.     currentGlobalState = eventEffectDB.getNextState(currentGlobalState, event);
20.     if currentGlobalState == null then
21.         return null;
22.     end
23. end
24. return currentGlobalState;

Algorithm 2: Disjoint Update Reduction Policy

Function isDisjointUpdate (event1, event2)
1. /* plugin the program analysis result here, for instance, */
2. if isDiscardMessage(event1) or isDiscardMessage(event2) then
3.     return true;
4. end
5. if event1.type == "PROPOSE" and event2.type == "PROPOSE_RESPONSE" then
6.     return true;
7. end
8. if event1.type == "PROPOSE_RESPONSE" and event2.type == "PROPOSE" then
9.     return true;
10. end
11. /* and so on. */
12. return false;
Algorithm 3: Parallel Flips Reduction Policy - Part 1

1 Function `canBeParalleled (concEventsPaths)`
2     uniqueChains = getUniqueChains(concEventsPaths);
3     manyEventsInANode = 0;
4     parallelPaths = [];
5     for uniqueChain in uniqueChains do
6         if uniqueChain.size() > 1 then
7             manyEventsInANode++;
8         end
9     end
10    if manyEventsInANode > 1 then
11        parallelPaths = findParallelPaths(uniqueChains, concEventsPaths);
12    else
13        parallelPaths = concEventsPaths;
14    end
15    return parallelPaths;

16 Function `getUniqueChains (paths)`
17    chainSet = [];
18    for path in paths do
19        uniqueChain = getUniqueChain(path);
20        for node in uniqueChain do
21            chainSet.add(uniqueChain[node]);
22        end
23    end
24    return chainSet;

25 Function `getUniqueChain (path)`
26    eventChain = hashmap();
27    for event in path do
28        eventChain[event.recv].append(path);
29    end
30    return eventChain;
**Algorithm 4:** Parallel Flips Reduction Policy - Part 2

1. **Function** `findParallelPaths(uniqueChains, paths)`
   
   ```python
   rawParallelPaths = getParallelPaths(uniqueChains, paths);
   finalParallelPaths = getParallelPaths(uniqueChains, rawParallelPaths.reverse());
   return finalParallelPaths;
   ```

2. **Function** `getParallelPaths(uniqueChains, paths)`
   
   ```python
   checklist = getCheckList(pairs);
   parallelPaths = [];
   for path in paths do
       chain = getUniqueChain(path);
       hasNewChain = false;
       for node, nodeChain in chain do
           if !checklist[nodeChain] then
               checklist[nodeChain] = true;
               hasNewChain = true;
           end
       end
       if hasNewChain then
           parallelPaths.append(path);
       end
   end
   return parallelPaths;
   ```

3. **Function** `getCheckList(chains)`
   
   ```python
   checklist = hashmap();
   for chain in chains do
       checklist[chain] = false;
   end
   return checklist;
   ```

**Algorithm 5:** No Zero-Crash-Impact Reordering Reduction Policy

1. **Function** `isZeroImpactCrash(event1, event2)`
   
   ```python
   /* plugin program analysis result, for instance, */
   if isDiscardMessage(event1) and isCrash(event2) then
       return true;
   end
   if isDiscardMessage(event2) and isCrash(event1) then
       return true;
   end
   /* and so on.. */
   return false;
   ```
10.3 Program Analysis Pseudo-Code

The AMC program analysis annotates the node states, message types, disk writing proposition and also the recovery protocol entry. From these entry points, the program analysis collects and constructs the AST trees for all methods in the system (Algorithm 6, Algorithm 7). Next it builds the call graph by connecting the method invocation with the method declaration in Algorithm 8. With a complete call graph of the target system and the various message types, AMC program analysis can identify corresponding message handlers to be the entry points and trace their successors to determine the updateSet, readSet, sendSet, and diskSet in Algorithm 9. Finally, AMC analyzes the disjoint update message pairs and the zero-impact-crash possibilities with Algorithm 11 and Algorithm 12.
Algorithm 6: Collect Methods and Corresponding Children

Function `collectMethodsAndChildren` (`messageTypes`)

1. totalMethods = [], totalAssigns = [], totalIfs = [], totalElses = [], totalInvocations = [], totalMessageCreates = [];
2. for class in `collectAllClasses()` do
3.   totalMethods.append(method);
4.   for method in class.getMethods() do
5.     totalMethods.append(method);
6.     for child in method.getBody() do
7.       if child.type == assignment then
8.         totalAssigns.append(child);
9.       else if child.type == ifStatement then
10.      totalIfs.append(child);
11.     else if child.type == elseStatement then
12.      totalElses.append(child);
13.     else if child.type == methodInvocation then
14.      totalInvocations.append(child);
15.     else if child.type == classInstanceCreation then
16.       for argument in child.getArguments() do
17.         if messageTypes.contains(argument) then
18.           totalMessageCreates.append(child);
19.       end
20.     end
21.   end
22.   constructASTTree(method, child);
23. end
24. return totalMethods, totalAssigns, totalIfs, totalElses, totalInvocations, totalMessageCreates;

Function `constructASTTree` (`ASTNode parent`, `ASTNode child`)

27. parent.getChildren().append(child);
28. child.getParents().append(parent);
Algorithm 7: Construct AST Method Trees

Function constructMethodsTree (totalMethods, totalAssigns, totalIfs, totalElses, totalInvocations, totalMessageCreates)
    childrenCollections = [];
    for method in totalMethods do
        if method.getChildren() != null then
            for child in method.getChildren() do
                parseChild(child);
                parentSize = child.getParents().size();
                if parentSize > 1 then
                    resetParentRelationship(child, parentSize);
                end
            end
            resetChildrenRelationship(method);
        end
    end
    return totalMethods;

Function parseChild (child)
    for innerChild in child.getBody() do
        if innerChild.getType() == assignment then
            extractChild = totalAssigns.get(totalAssigns.index(innerChild));
        else if innerChild.getType() == ifStatement then
            extractChild = totalIfs.get(totalIfs.index(innerChild));
        else if innerChild.getType() == elseStatement then
            extractChild = totalElses.get(totalElses.index(innerChild));
        else if innerChild.getType() == methodInvocation then
            extractChild = totalInvocations.get(totalInvocations.index(innerChild));
        else if innerChild.getType() == ClassInstanceCreation then
            for argument in child.getArguments() do
                if messageTypes.contains(argument) then
                    extractChild = totalMessageCreates.get(totalMessageCreates.index(innerChild));
                end
            end
        end
        constructASTTree(child, extractChild);
    end

Function resetParentRelationship(child, parentSize)
    parent = child.getParents(parentSize-1);
    child.getParents().clear(); child.getParents().append(parent);

Function resetChildrenRelationship (method)
    childrenCollections.append(method);
    for child in method.getChildren() do
        if !childrenCollections.contains(child) then
            resetChildrenRelationship(child);
        else
            child.clear();
        end
    end
Algorithm 8: Connect AST Trees

Function connectASTTrees(totalMethods)

for method in totalMethods do
    if method.getChildren() != null then
        connectASTTrees(method);
    end
end
return totalMethods;

Function connectASTTrees(method)

for child in method.getChildren() do
    if child.getType() == assignment then
        connectASTTrees(child);
    end
    else if child.getType() == ifStatement then
        connectASTTrees(child);
    end
    else if child.getType() == elseStatement then
        connectASTTrees(child);
    end
    else if child.getType() == methodInvocation then
        for class in allClasses do
            if child.getClass() == class then
                for decl in classMethods do
                    if d then
                        e
                    end
                    cl == child
                    child.getChildren().append(decl);
                    decl.getParents().append(child);
                end
            end
        end
    else if child.getType() == ClassInstanceCreation then
        connectASTTrees(child);
    end
end
Algorithm 9: Determine the updateSet, readSet, sendSet, diskSet - Part 1

Function `determineAllSets (messageTypes, totalInvocations, diskWriting)`

messageTypeAllSets = hashmap();
initializeStateDiskWritingProperty();

for message in messageTypes do
  readExpressions = [], updateSet = [], readSet = [], sendSet = [], diskSet = [];
  isMessage = false, isMessageHandler = false;
  for invocation in totalInvocations do
    for argument in child.getArguments() do
      if argument == message then
        isMessage = true;
      end
    end
    if isMessage then
      for argument in child.getArguments() do
        if argument.getType() == classInstanceCreation then
          isMessageHandler = true;
          messageHandler = argument.getClassName();
        end
      end
      if isMessageHandler then
        for method in messageHandler.getMethods() do
          parseUpdateVariables(method);
        end
        parseReadVariables();
      end
    end
  end
  messageTypeAllSets[message] = updateSet, readSet, sendSet, diskSet;
end
return messageTypeAllSets;

Function `parseUpdateVariables (method)`

for child in method.getChildren() do
  if child.getType() == assignment then
    readExpressions.append(child.getRightHandSide());
    for state in nodeStates do
      if state == child.getLeftHandSide() then
        updateSet.append(state);
      end
    end
  else if child.getType() == ifStatement then
    readExpressions.append(child.getExpression());
  else if child.getType() == diskWriting then
    parseDiskVariables(child);
  else if child.getType() == classInstanceCreation then
    parseSentVariables(child);
  end
  end
parseUpdateVariables(child);
end
Algorithm 10: Determine The updateSet, readSet, sendSet, diskSet - Part 2

Function parseSentVariables (child)
    for argument in child getArguments() do
        for state in nodeStates do
            if argument == state then
                sendSet.append(state);
            end
        end
    end

Function parseDiskVariables (child)
    for parent in child.getParents() do
        for argument in parent getArguments() do
            for state in nodeStates do
                if state == argument then
                    state.isDiskWriting = true;
                    diskSet.append(state);
                end
            end
        end
    end

parseDiskVariables(parent);

Function parseReadVariables()
    for message in messageTypes do
        for exp in readExpressions do
            for state in nodeStates do
                if state == exp.state then
                    readSet.append(state);
                end
            end
        end
    end
Algorithm 11: Determine Disjoint Update Independence Rules

Function disjointUpdateIndependence(messageType)
1. isOverlap = false, isDiscard = false;
2. discardMessages = [];
3. disjointUpdatePairs = hashmap();
4. for message1 in messageTypes do
5.  if message1.getUpdateSet().size() == 0 then
6.    discardMessages.append(message1);
7.  end
8.  for message2 in messageTypes do
9.    if checkOverlap(message1, message2) or checkOverlap(message2, message1)
10.       then
11.       disjointUpdatePairs[message1] = message2;
12.     end
13.   end
14. end

Function checkOverlap(message1, message2)
15. for update1 in message1.getUpdateSet() do
16.   for update2 in message2.getUpdateSet() do
17.     if update1 == update2 then
18.       isOverlap = true;
19.     end
20.   else if update1.isDiskWriting() and update2.isDiskWriting() then
21.     isOverlap = true;
22.   end
23. end
24. if message2.getReadSet().contains(update1) then
25.   isOverlap = true;
26. end
27. if message2.getSendSet().contains(update1) then
28.   isOverlap = true;
29. end
30. return isOverlap;

Algorithm 12: Determine No Zero-Impact-Crash Reordering Rules

Function isNoZeroImpactCrashReordering(recoveryProtocol, diskWriting)
1. for child in recoveryProtocol.getBody() do
2.   if child.getT ype() == catchClause then
3.     if child.getBody().contains(diskWriting) then
4.       return true;
5.     end
6.   end
7. end
8. return false;
REFERENCES

[1] Personal Communication with ZooKeeper Developers.


