Minimizing Test Cases For Distributed Systems

Colin Scott
Qualifying Exam Proposal
UC Berkeley Department of Computer Science
October 30th 2014

Abstract

Software bugs are pervasive in distributed systems, and troubleshooting is a tedious, time-consuming task. In this dissertation proposal we argue that it is possible to automatically identify minimal sequences of inputs responsible for triggering test case failures, thereby easing the troubleshooting process. We begin by describing work that has already been completed on identifying minimal sequences without making assumptions about the language or instrumentation of the software under test. This work was applied to one type of distributed system: software-defined networking (SDN) control software. We end by outlining plans to broaden the application of our techniques beyond SDN and provide a stronger theoretical foundation for the mechanics of the techniques.

1 Introduction

Developers of distributed systems face notoriously difficult challenges, such as concurrency, asynchrony, and partial failure. A crucial consequence of these challenges is that software bugs are pervasive in distributed systems.

In an effort to catch these bugs before they are released into production, many commercial vendors employ teams of QA engineers. The QA engineers exercise automated test scenarios that involve sequences of external events—such as failures or message delays—on large testbeds. If they detect an invariant violation, they hand the resulting trace to a developer for analysis.

When faced with symptoms of a failing QA test (*e.g.* inconsistent system state) software developers first need to identify which of the potentially tens of thousands of events[1] in the QA test are relevant to triggering this apparent bug before they can begin to isolate and fix it. This act of "troubleshooting" (which precedes the act of debugging the code) is highly time-consuming, as developers spend hours poring over multigigabyte execution traces. This should come as no surprise since software developers in general spend roughly half (49% according to one study [21]) of their time troubleshooting and debugging, and spend considerable time troubleshooting bugs that are difficult to trigger (the same study found that 70% of the reported concurrency bugs take days to months to fix). Consequently, when asked to describe their ideal tool, most network operators said "automated troubleshooting" [63].

Our aim is to reduce effort spent on troubleshooting distributed systems. Our thesis is the following:

> *It is possible to automatically identify minimal sequences of inputs responsible for triggering distributed systems test case failures.*

By eliminating events from event traces that are not related to the invariant violation at the end of the trace, we produce a "minimal causal sequence" (MCS) of triggering events. After the MCS has been produced, the developer embarks on the debugging process. We claim that the greatly reduced size of the trace makes it easier for the developer to figure out which code path contains the underlying bug, allowing them to focus their effort on the task of fixing the problematic code itself. After the bug

[1] Consider that an hour long QA test emulating event rates observed in production could contain 8.5 network error events per minute [22] and 500 VM migrations per hour [50], for a total of $8.5 \cdot 60 + 500 \approx 1000$ externally triggered events, not counting internal events.

has been fixed, the MCS can serve as a test case to prevent regression, and can help identify redundant bug reports where the MCSes are the same.

Our goal of minimizing traces is in the spirit of delta debugging [61], but our problem is complicated by the distributed nature of control software: our input is not a single file fed to a single point of execution, but an ongoing sequence of events involving multiple actors. We therefore need to carefully control the interleaving of events in the face of asynchrony, concurrency and non-determinism in order to reproduce bugs throughout the minimization process.

In the first part of this dissertation proposal (§4), we present research that has already been completed on minimizing traces without making assumptions about the language or instrumentation of the software [49]. We evaluated this on one particular type of distributed system: software-defined networking (SDN) control software.

The remainder of this dissertation proposal (§5) fleshes out plans for broadening the applicability of our techniques to other kinds of distributed systems besides SDN control software, and providing a stronger theoretical foundation for mechanics of the techniques. In particular, we plan to (i) start with an infeasible but provably correct approach, and (ii) find practical approximations to this approach, many of which will involve leveraging empirical properties of (iii) different classes of distributed software systems (beyond just SDN control software).

We start by providing a formal problem definition (§2) and discussing related work (§3). We end by presenting a timeline for the completion of the proposed research (§6).

2 Problem Definition

A distributed system consists of multiple processes that send each other messages. We represent the state of a distributed system at a particular time as a configuration c, containing the current state of each process (but not any in-flight messages). A distributed system takes a sequence of external network events $E = e_1 \rightarrow e_2 \rightarrow \cdots e_m$ (*e.g.* failures) as inputs, and produces a sequence of configurations $C = c_1, c_2, \ldots, c_n$.

An invariant is a predicate P over configuration state (a safety condition). We say that configuration c violates the invariant if $P(c)$ is false, denoted $\overline{P}(c)$.

We are given a log L generated by a centralized QA test orchestrator. The log L contains a sequence of events

$$\tau_L = e_1 \rightarrow i_1 \rightarrow i_2 \rightarrow e_2 \rightarrow \cdots e_m \rightarrow \cdots i_p$$

which includes external events $E_L = e_1, e_2 \cdots e_m$ injected by the test orchestrator, and internal events $I_L = i_1, i_2 \cdots i_p$ triggered by the distributed system (e.g. messages sent between nodes, or state transitions within nodes). The events E_L include timestamps $\{(e_k, t_k)\}$ from the orchestrator's clock.

A replay of log L involves replaying the external events E_L, possibly taking into account the occurrence of internal events I_L as observed by the orchestrator. We denote a replay attempt by $replay(\tau)$. The output of $replay$ is a sequence of configurations $C_R = \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$. Ideally $replay(\tau_L)$ reproduces the original configuration sequence, but this does not always hold.

If the configuration sequence $C_L = c_1, c_2, \ldots, c_n$ associated with the log L violated predicate P (*i.e.* $\exists_{c_i \in C_L}.\overline{P}(c_i)$) then we say $replay(\cdot) = C_R$ reproduces that violation if C_R contains an equivalent faulty configuration (*i.e.* $\exists_{\hat{c}_i \in C_R}.\overline{P}(\hat{c}_i)$).

The goal of our work is, when given a log L that exhibited an invariant violation, to find a small, replayable sequence of events that reproduces that invariant violation. Formally, we define a minimal causal sequence (MCS) to be a sequence τ_M where the external events $E_M \in \tau_M$ are a subsequence of E_L such that $replay(\tau_M)$ reproduces the invariant violation, but for all proper subsequences E_N of E_M there is no sequence τ_N such that $replay(\tau_N)$ reproduces the violation. Note that an MCS is not necessarily *globally* minimal, in that there could be smaller subsequences of E_L that reproduce this violation, but are not a subsequence of this MCS.

Given a log L generated from testing infrastructure, our goal is to find an approximate MCS, so that a human can examine the MCS rather than the full log. In general, finding an MCS is difficult, but the utility of an MCS does not depend on its being exactly minimal, but in it being much smaller than the original QA log.

Finding the MCS involves two tasks: (i) searching through subsequences of E_L, and more importantly, (ii) deciding when to inject external events (i.e. choosing a set of timings) for each subsequence so that, whenever possible, the invariant violation is retriggered.

3 Related Work

Techniques for Selecting Subsequences. Checking random subsequences of E_L would be one viable but inefficient approach to achieving our first task. A better approach is the delta debugging algorithm [61, 62], a divide-and-conquer algorithm for isolating fault-inducing inputs. We use delta debugging[2] to iteratively select subsequences of E_L and $replay$ each subsequence with some timing T.

Techniques for Selecting Timings. Simply exploring subsequences E_S of E_L is insufficient for finding MCSes: the timing of when we inject the external events during $replay$ is crucial for reproducing violations.

The most natural approach to scheduling external events is to maintain the original wall-clock timing intervals between them. If this is able to find all minimization opportunities, *i.e.* reproduce the violation for all subsequences that are a supersequence of some MCS, we say that the inputs are isolated. The original applications of delta debugging [8, 47, 61, 62] make this assumption (where a single input is fed to a single program), as well as QuickCheck's input "shrinking" [13] when applied to blackbox systems like synchronous telecommunications protocols [3].

We tried this approach, but were rarely able to reproduce invariant violations. As our case studies demonstrate [49], this is largely due to the concurrent, asynchronous nature of distributed systems; consider that the network can reorder or delay messages, or that processes may respond to multiple inputs simultaneously. Inputs injected according to wall-clock time are not guaranteed to coincide correctly with the current state of the processes.

We must therefore consider the distributed system's internal events. To deterministically reproduce bugs, we would need visibility into every I/O request and response (*e.g.* clock values or socket reads), as well as all thread scheduling decisions for each process. This information is the starting point for techniques that seek to minimize thread interleavings leading up to race conditions. These approaches involve iteratively feeding a single input (the thread schedule) to a single entity (a deterministic scheduler) [12, 14, 28], or statically analyzing feasible thread schedules [25].

A crucial constraint of these approaches is that they must keep the inputs fixed; that is, behavior must depend uniquely on the thread schedule. Otherwise, the nodes may take a divergent code path. If this occurs some processes might issue a previously unobserved I/O request, and the replayer will not have a recorded response; worse yet, a divergent process might deschedule itself at a different point than it did originally, so that the remainder of the recorded thread schedule is unusable to the replayer.

Because they keep the inputs fixed, these approaches strive for a subtly different goal than ours: minimizing thread context switches rather than input events. At best, these approaches can indirectly minimize input events by truncating individual thread executions. That is, they can cause threads to exit early (thereby shortening the execution trace), but they cannot remove extraneous events from the middle of the trace.

With additional information obtained by program flow analysis [26, 35, 52] however, the inputs no longer need to be fixed. The internal events considered by these program flow reduction techniques are individual instructions executed by the programs (obtained by instrumenting the language runtime), in addition to I/O responses and the thread schedule. With this information they can compute program flow dependencies, and thereby remove input events from anywhere in the trace as long as they can prove that doing so cannot possibly cause the faulty execution path to diverge.

While program flow reduction is able to minimize inputs, these techniques are not able to explore alternate code paths that still trigger the invariant violation. They are also overly conservative in removing inputs (*e.g.* EFF takes the transitive closure of all possible dependencies [35]) causing them to miss opportunities to remove dependencies that actually semantically commute.

Our Contribution. For the first portion of this dissertation proposal (§4) we discuss heuristics we have developed for interleaving internal and external events that we have empirically shown to work well for minimizing event traces. Our approach there is to allow processes to proceed along divergent paths rather than recording all low-level I/O and thread scheduling decisions. This has several advantages. Unlike the other approaches, we can find shorter alternate code paths that still trig-

[2] Domain-specific minimization algorithms also exist [13, 47, 56], but we focus on the general case.

ger the invariant violation. Previous *best-effort* execution minimization techniques [10, 15, 54] also allow alternate code paths, but do not systematically consider concurrency and asynchrony.[3] We also avoid the performance overhead of recording all I/O requests and later replaying them (*e.g.* EFF incurs ˜10x slowdown during replay [35]). Lastly, we avoid the extensive effort required to instrument the control software's language runtime, needed by the other approaches to implement a deterministic thread scheduler, interpose on syscalls, or perform program flow analysis. By avoiding assumptions about the language of the control software, we were able to easily apply our system to five different control platforms written in three different languages.

Unfortunately, these heuristics do not lend themselves to solid theoretical explanations for why they work well. In the latter portion of the proposal (§5) we outline a plan for developing scheduling strategies that are based on sound principles (starting with visibility into the structure of the software under test). As far as we know, the problem statement and approach we pose there—applying model checkers to *certify* whether each subsequence chosen by delta debugging reproduces the original bug—has not appeared in the literature before.

Broadly Related Work. We end this section by discussing techniques in the general area of troubleshooting. We characterize the other troubleshooting approaches as (i) instrumentation (tracing), (ii) bug detection (invariant checking), (iii) replay, (iv) root cause analysis (of device failures), (v) log comprehension (visualization), (vi) program slicing, and (vii) automated debugging (fault localization).

Instrumentation. Unstructured log files collected at each node are the most common form of diagnostic information. The goal of tracing frameworks [4,11,18,24,48] is to produce structured logs that can be easily analyzed, such as DAGs tracking requests passing through the distributed system. These tools help developers to understand how, when, and where the system broke. In contrast, we focus on making it easier for developers to understand what lead the software to violate an invariant in the first place.

Bug Detection. With instrumentation available, it becomes possible to check expectations about the system's state (either offline [37] or online [38]), or about the paths requests take through the system [48]. Within the networking community, this research is primarily focused on verifying routing tables [30–32, 42] or forwarding behavior [64, 65]. We use bug detection techniques (invariant checking) to guide delta debugging's minimization process.

It is also possible to infer performance anomalies by building probabilistic models from collections of traces [4,11]. Our goal is to produce exact minimal causal sequences, and we are primarily focused on correctness instead of performance.

Model checkers [9,44] seek to proactively find safety and liveness violations by analyzing all possible code paths. After identifying a bug with model checking, finding a minimal code path leading to it is straightforward. However, the testing systems we aim to improve do not employ formal methods such as model checking, in part because model checking usually suffers from exponential state explosion when run on large systems.[4] Nonetheless, in §5 we propose the use of model checkers to provide provably minimal MCSes. Crucially, we assume that we are already given a known execution leading to an invariant violation, rather than attempting to find all possible invariant violations without prior knowledge of faulty executions. We believe that the prior knowledge contained in failing test cases can mitigate exponential state explosion, and as far as we know we are the first to use model checkers to minimize existing executions.

Replay. Crucial diagnostic information is often missing from traces. Record and replay techniques [20, 36, 59, 60] instead allow users to step through (deterministic) executions and interactively examine the state of the system in exchange for performance overhead. Manually examining long system executions can be tedious, and our goal is to minimize such executions so that developers find it easier to identify the problematic code through replay or other means.

Root Cause Analysis. Without perfect instrumentation, it is often not possible to know exactly what events are occurring (*e.g.* which components have failed) in a distributed system. Root cause analysis [29, 58] seeks to re-

[3] PRES explores alternate code paths in best-effort replay of multi-threaded executions, but does not minimize executions [46].

[4] For example, NICE [9] took 30 hours to model check a network with two switches, two hosts, the NOX MAC-learning control program (98 LoC), and five concurrent messages between the hosts.

construct those unknown events from limited monitoring data. Here we know exactly which events occurred, but seek to identify a minimal sequence of events.

Log Comprehension. Model inference techniques summarize log files in order to make them more easily understandable by humans [5–7, 39, 40]. Model inference does not modify the logs, although it can be applied to our minimized logs to further facilitate the troubleshooting process.

Program Slicing. It is worth mentioning another goal outside the purview of distributed systems, but closely in line with ours: program slicing [55] is a technique for finding the minimal subset of a program that could possibly affect the result of a particular line of code. Our goal is to slice the temporal dimension of an execution rather than the code dimension.

Automated Debugging. Literature following Weisers' original program slicing paper goes so far as to try to automatically locate the exact line(s) of code or state transitions that are responsible for a bug, using statistical data [66], test coverage data [27, 57], constraint solving [41], fault injection [33] and experimentally driven executions of the failing program [16,51]. These approaches aid the debugging process rather than test case simplification, and are therefore complementary.

4 Past Work

In this section we present heuristics we previously developed and applied to one kind of distributed system: software-defined networking (SDN) control software. Here, we begin by briefly describing the behavior of SDN control software.

Overview of SDN Control Software. Network operating systems, the key component of SDN software infrastructure, consist of control software running on a replicated set of servers, each running a controller instance. Controllers coordinate between themselves, and receive input events (*e.g.* link failure notifications) and statistics from switches (either physical or virtual), policy changes via a management interface, and possibly dataplane packets. In response, the controllers issue forwarding instructions to switches. All input events are asynchronous, and individual controllers may fail at any time. The controllers either communicate with each other over the dataplane network, or use a separate dedicated network, and may become partitioned.

Internal Message	Masked Values
OpenFlow messages	xac id, cookie, buffer id, stats
packet_out/in payload	all values except src, dst, data
Log statements	varargs parameters to printf

Table 1: Internal messages and their masked values.

The goal of the network control plane is to configure the switch forwarding entries so as to enforce one or more invariants, such as connectivity (*i.e.* ensuring that a route exists between every endpoint pair), isolation and access control (*i.e.* various limitations on connectivity), and virtualization (*i.e.* ensuring that packets are handled in a manner consistent with the specified virtual network). A bug causes an invariant to be violated. Invariants can be violated because the system was improperly configured (*e.g.* the management system [1] or a human improperly specified their goals), or because there is a bug within the SDN control plane itself. In this paper we focus on troubleshooting bugs in the SDN control plane after it has been given a valid policy configuration.[5]

Heuristics for Accounting for Interleavings. To reproduce an invariant violation found in an execution of SDN control software, we need to inject each input event e only after all other events, including internal events, that precede it in the happens-before relation [34] from the original execution ($\{i \mid i \rightarrow e\}$) have occurred [53].

The internal events we consider are (a) message delivery events, either between controllers (*e.g.* database synchronization messages) or between controllers and switches (*e.g.* OpenFlow messages), and (b) state transitions within controllers (*e.g.* a backup node deciding to become master). Our replay orchestrator obtains visibility into (a) by interposing on all messages within the test environment (described in detail in [49]). It optionally obtains partial visibility into (b) by instrumenting controller software with a simple interposition layer (described in [49]).

Given a subsequence E_S, our goal is to find an execution that obeys the original happens-before relation. We do not control the occurrence of internal events, but we can manipulate when they are delivered through our inter-

[5]This does not preclude us from troubleshooting misspecified policies so long as test invariants [31] are specified separately.

Input Type	Implementation
Switch failure/recovery	TCP teardown
Controller failure/recovery	`SIGKILL`
Link failure/recovery	`ofp_port_status`
Controller partition	`iptables`
Dataplane packet injection	Network namespaces
Dataplane packet drop	Dataplane interposition
Dataplane packet delay	Dataplane interposition
Host migration	`ofp_port_status`
Control message delay	Controlplane interposition
Non-deterministic TCAMs	Modified switches

Table 2: Input types currently supported by our troubleshooting system.

procedure PEEK($input\ subsequence$)
$inferred \leftarrow [\,]$
for e_i *in subsequence*
$checkpoint\ system$
$inject\ e_i$
$\Delta \leftarrow |e_{i+1}.time - e_i.time| + \epsilon$
$record\ events\ for\ \Delta\ seconds$
$matched \leftarrow original\ events\ \&\ recorded\ events$
$inferred \leftarrow inferred + [e_i] + matched$
$restore\ checkpoint$
return $inferred$

Figure 1: PEEK determines which internal events from the original sequence occur for a given subsequence.

position layer,[6] and we also decide when to inject the external events E_S. The key challenges in choosing a schedule stem from the fact that the original execution has been modified: internal events may differ syntactically, some expected internal events may no longer occur, and new internal events may occur that were not observed at all in the original execution.

Functional Equivalence. Internal events may differ syntactically (*e.g.* sequence numbers of control packets may all differ) when replaying a subsequence of the original log. We observe that many internal events are *functionally equivalent*, in the sense that they have the same effect on the state of the system with respect to triggering the invariant violation. For example, `flow_mod` messages may cause switches to make the same change to their forwarding behavior even if their transaction ids differ.

We apply this observation by defining masks over semantically extraneous fields of internal events.[7] We show the fields we mask in Table 1. Note that these masks only need to be specified once, and can later be applied programmatically.

We then consider an internal event i' observed in $replay$ equivalent (in the sense of inheriting all of its happens-before relations) to an internal event i from the original log if and only if all unmasked fields have the same value and i occurs between i''s preceding and succeeding inputs in the happens-before relation.

Handling Absent Internal Events. Some internal events from the original log that "happen before" some external input may be absent when replaying a subsequence. For instance, if we prune a link failure, the corresponding notification message will not arise.

To avoid waiting forever we infer the presence of internal events before we $replay$ each subsequence. Our algorithm (called PEEK()) for inferring the presence of internal events is depicted in Figure 1. The algorithm injects each input, records a checkpoint[8] of the network and the control software's state, allows the system to proceed up until the following input (plus a small time ϵ), records the observed events, and matches the recorded events with the functionally equivalent internal events observed in the original trace.[9]

Handling New Internal Events. The last possible induced change is the occurrence of new internal events that were not observed in the original log. New events present multiple possibilities for where we should inject the next input. Consider the following case: if i_2 and i_3 are internal events observed during $replay$ that are both in the same equivalence class as a single event i_1 from the original run, we could inject the next input after i_2 or after i_3.

In the general case it is always possible to construct two state machines that lead to differing outcomes: one that only leads to the invariant violation when we inject the next input *before* a new internal event, and another only when we inject *after* a new internal event. In other words, to be guaranteed to traverse any state transition suffix that

[6] In this way we totally order messages. Without interposition on process scheduling however, the system may still be concurrent.

[7] One consequence of applying masks is that bugs involving masked fields are outside the purview of our approach.

[8] We discuss the implementation details of checkpointing in [49].

[9] In the case that, due to non-determinism, an internal event occurs during PEEK() but does not occur during $replay$, we time out on internal events after ϵ seconds of their expected occurrence.

leads to the violation, we must recursively branch, trying both possibilities for every new internal event. This implies an exponential worst case number of possibilities to be explored.

Exponential search over these possibilities is not a practical option. Our heuristic is to proceed normally if there are new internal events, always injecting the next input when its last expected predecessor either occurs or times out. This ensures that we always find state transition suffixes that contain a subsequence of the (equivalent) original internal events, but leaves open the possibility of finding divergent suffixes that lead to the invariant violation.

Recap. We combine these heuristics to replay each subsequence chosen by delta debugging: we compute functional equivalency for all internal events intercepted by our test orchestrator's interposition layer, we invoke PEEK() to infer absent internal events, and with these inferred causal dependencies we $replay$ the input subsequence, waiting to inject each input until each of its (functionally equivalent) predecessors have occurred while allowing new internal events through the interposition layer immediately.

Results. We end by briefly describing our evaluation of these techniques. The evaluation was carried out on our troubleshooting system STS, which incorporates our heuristics. For a full evaluation, see [49].

We summarize the results of our evaluation in Table 3. The table summarizes case studies we ran on five open source SDN control platforms: ONOS [45] (Java), POX [43] (Python), NOX [23] (C++), Pyretic [19] (Python), and Floodlight [17] (Java), and debugged these with the help of STS. The case studies include seven real bugs we found ourselves with fuzz testing, as well as a demonstration of the boundaries of where STS works well and where it does not on previously known and synthetic bugs that span a range of bug types encountered in practice.

We note that with the exception of Delicate Timer Interleaving and ONOS Database Locking, STS was able to significantly reduce input traces. The MCS WI column, showing the MCS sizes we produced when ignoring internal events entirely, indicates that our techniques for interleaving events are often crucial. In one case however—Reactive Routing Trigger—non-determinism was particularly acute, and STS's interposition on internal events actually made minimization worse due to timeouts on inferred internal events that did not occur after PEEK(). In this case we found better results by simply turning off interposition on internal events. For all of the other case studies, either non-determinism was not problematic, or we were able to counteract it by replaying multiple times per subsequence and adding instrumentation.

The cases where STS was most useful were those where a developer would have started from the end of the trace and worked backwards, but the actual root cause lies many events in the past (as in Memory Corruption). This requires many re-iterations through the code and logs using standard debugging tools (*e.g.* source level debuggers), and is highly tedious on human timescales. In contrast, it was easy to step through a small event trace and manually identify the code paths responsible for a failure.

Bugs that depend on fine-grained thread-interleaving or timers inside of the controller are the worst-case for STS. This is not surprising, as they do not directly depend on the input events from the network, and we do not directly control the internal scheduling and timing of the controllers. The fact that STS has a difficult time reducing these traces is itself indication to the developer that fine-grained non-determinism is at play.

5 Future Work

The scheduling heuristics we developed in the previous section have several shortcomings. Most importantly, treating the software as a blackbox disallows us from showing formal properties of the MCSes we produce. That is, we cannot explain why our outputs are good approximations to truly minimal sequences, nor can we explain how the heuristics helped. Second, although we do not believe that the techniques are specific to SDN control software, they currently pertain only to that specific domain. Here we outline plans to address both of the these shortcomings. At a high level, we plan to (i) start with an infeasible but provably correct approach, and (ii) find practical approximations to this approach, many of which will involve leveraging empirical properties of (iii) different classes of distributed software systems (beyond just SDN control software).

Showing Formal Properties of MCSes. If we were to repeatedly $replay$ a fixed subsequence of external events

	Bug Name	Topology	Runtime (s)	Input Size	MCS Size	MCS WI	MCS Helpful?
Newly Found	Pyretic Loop	3 switch mesh	266.2	36	1	2	Yes
	POX Premature PacketIn	4 switch mesh	249.1	102	2	NR	Yes
	POX In-Flight Blackhole	2 switch mesh	1478.9	27	11	NR	Yes
	POX Migration Blackhole	4 switch mesh	1796.0	29	3	NR	Yes
	NOX Discovery Loop	4 switch mesh	4990.9	150	18	NR	Indirectly
	Floodlight Loop	3 switch mesh	27930.6	117	13	NR	Yes
	ONOS Database Locking	2 switch mesh	N/A	1	1	1	N/A
Known	Floodlight Failover	2 switch mesh	-	202	2	-	Yes
	ONOS Master Election	2 switch mesh	2746.0	20	2	2	Yes
	POX Load Balancer	3 switch mesh	2396.7	106	24 (N+1)	26	Yes
Synthetic	Delicate Timer Interleaving	3 switch mesh	N/A	39	NR	NR	No
	Reactive Routing Trigger	3 switch mesh	525.2	40	7	2	Indirectly
	Overlapping Flow Entries	2 switch mesh	115.4	27	2	3	Yes
	Null Pointer	20 switch FatTree	157.4	62	2	2	Yes
	Multithreaded Race Condition	10 switch mesh	36967.5	1596	2	2	Indirectly
	Memory Leak	2 switch mesh	15022.6	719	32 (M+2)	33	Indirectly
	Memory Corruption	4 switch mesh	145.7	341	2	2	Yes

Table 3: Overview of Case Studies. 'WI' denotes 'Without Interposition', and 'NR' denotes 'Not Replayable'.

E_S to a blackbox distributed system, it is possible that it would not to produce the same output on each execution, even if we were to apply the heuristics outlined in §4. The issue is that we do not have full visibility into internal events nor control over the internal scheduling decisions of the distributed system. Our approach in [49] was to replay each subsequence multiple times, but this does not provide any guarantees on the minimality of the MCSes we produce, since it's possible that a subsequence E_S that we thought did not reproduced the invariant violation could have done so if the distributed system had interleaved its internal events in some other way.

Suppose we want to show how close our outputs are to minimal. One approach would be to use a model checker to *certify* whether each subsequence E_S chosen by delta debugging does or does not reproduce the original violation, thereby producing a provably minimal MCS. The model checker could accomplish this by systematically exploring all possible interleavings of internal events given the fixed sequence of external events E_S.[10]

Working Around Impossibility. In some cases, it is not possible for the model checker to certify a given subsequence E_S. For example, suppose that the distributed system does not terminate (or more specifically, suppose its state space is infinite and non-recurring). In that case, model checking is not guaranteed to terminate. Crucially, if the model checker does not find a way to trigger the original violation in finite time, that does not imply that the violation cannot be triggered, and we therefore lose our guarantees on minimality.

In distributed systems, non-termination means that an algorithm is not guaranteed to stop sending messages. The formal term for this property is 'non-quiescence'. In [2] Aguilera et al. prove that all failure detectors—an important algorithmic component used by many distributed systems—are non-quiescent.

As a fallback to non-quiescence, we can resort to bounded model checking, where the model checker only explores the event interleavings for a given subsequence E_S up to a certain number of execution steps. This approaches weakens the soundness (minimality) guarantees provided by our approach in favor of the ability to operate on real distributed systems implementations.

In some cases, we can also leverage the following observation to work around non-quiescence: given a failure detector component, it is possible to implement many other distributed algorithms in a quiescent manner [2]. Building on this observation, one might mark (non-quiescent) failure detector algorithms as 'trusted-components', such that the model checker takes control of when the failure detector reports failures and recoveries

[10]Note that the internal events are not fixed. That is, the schedule chosen by the model checker affects which internal events are subsequently triggered by the software under test.

to the application, and systematically explores when these failure/recovery reports are triggered in order to certify the remaining (quiescent) components of the distributed system.

Making Certification Practical. Although we only use the model checker to find specific invariant violations rather than asking it to find all invariant violations, it may nonetheless need to explore an intractable number of event interleavings. This would make it impractical for use on real distributed software.

We believe that we can ameliorate computational intractability by leveraging the prior knowledge contained in the original failing test case. That is, by making the $replay$ function stateful, we can develop heuristics that lead the model checker to quickly find interleavings that trigger the original violation, so that it is only forced to enumerate all interleavings for subsequence E_S that do not trigger the original violation.

As a concrete example, if we lead the model checker to first explore interleavings that have small edit distances from the original execution, we hypothesize that its probability of finding the same invariant violation within the first few explored scheduled will increase dramatically versus randomly choosing schedules to explore. If the model checker can find violation-triggering schedules in $O(N)$ time rather than $O(2^N)$ time (where N denotes the number of events to schedule), the asymptotic complexity of the overall minimization process can be reduced. We will evaluate these heuristics empirically, by measuring the number of interleavings that need to be explored for event traces where we know the MCS *a priori*. Often, we will develop these heuristics by examining the behavior of specific distributed systems, as we describe next.

Applying to Other Distributed Systems. In [49] we only applied our techniques to SDN control software. We do not however believe that our techniques are specific to SDN. We have already begun applying our techniques to other kinds of distributed systems, including distributed databases and consensus protocols. One contribution of our work will be to tailor our minimization strategies so that they leverage the properties of those systems (e.g. atomic transactions in the domain of distributed databases) or their empirical behavior (e.g. a tendency to trigger violations along certain schedules) in order to make the minimization process more practical.

6 Timeline

Before closing, we present a timeline for completion of the proposed research:

- Complete: [SIGCOMM 2014] Presented work on identifying minimal sequences without making assumptions about the language or instrumentation of the software under test [49].
- Planned: [SOSP 2015] Work on (i) finding an infeasible but provably correct approach, and (ii) finding practical approximations to this approach, many of which will involve leveraging empirical properties of (iii) different classes of distributed software systems (beyond just SDN control software).
- Graduation: Winter 2015 / Spring 2016.

References

[1] OpenStack Neutron. http://tinyurl.com/qj8ebuc.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. International Workshop on Distributed Algorithms '97.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. Erlang '06.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. OSDI '04.

[5] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring Models of Concurrent Systems from Logs of their Behavior with CSight. ICSE '14.

[6] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. ESEC/FSE '11.

[7] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of their Behavior.

[8] M. Burger and A. Zeller. Minimizing Reproduction of Software Failures. ISSTA '11.

[9] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. NSDI '12.

[10] K.-h. Chang, V. Bertacco, and I. L. Markov. Simulation-Based Bug Trace Minimization with BMC-Based Refinement. IEEE TCAD '07.

[11] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. DSN '02.

[12] J. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. SIGSOFT '02.

[13] K. Claessen and J. Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. ICFP '00.

[14] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding Race Conditions in Erlang with QuickCheck and PULSE. ICFP '09.

[15] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. ICSE '07.

[16] H. Cleve and A. Zeller. Locating Causes of Program Failures. ICSE '05.

[17] Floodlight Controller. http://tinyurl.com/ntjxa6l.

[18] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. NSDI '07.

[19] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. ICFP '11.

[20] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging For Distributed Applications. ATC '06.

[21] P. Godefroid and N. Nagappan. Concurrency at Microsoft - An Exploratory Survey. CAV '08.

[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network, Sec. 3.4. SIGCOMM '09.

[23] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System For Networks. CCR '08.

[24] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. NSDI '14.

[25] J. Huang and C. Zhang. An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs. SAS '11.

[26] J. Huang and C. Zhang. LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction. OOPSLA '12.

[27] J. A. Jones and M. J. Harrold and J. Stasko. Visualization of Test Information To Assist Fault Localization. ICSE '02.

[28] N. Jalbert and K. Sen. A Trace Simplification Technique for Effective Debugging of Concurrent Programs. FSE '10.

[29] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed Diagnosis in Enterprise Networks. SIGCOMM '09.

[30] P. Kazemian, M. Change, H. Zheng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. NSDI '13.

[31] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. NSDI '12.

[32] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. NSDI '13.

[33] L. Zhang and L. Zhang and S. Khurshid. Injecting Mechanical Faults to Localize Developer Faults for Evolving Software. OOPSLA '13.

[34] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. CACM '78.

[35] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward Generating Reducible Replay Logs. PLDI '11.

[36] C.-C. Lin, V. Jalaparti, M. Caesar, and J. Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. ATC '13.

[37] X. Liu. WiDs Checker: Combating Bugs in Distributed Systems. NSDI '07.

[38] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: Debugging Deployed Distributed Systems. NSDI '08.

[39] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. ICSE '08.

[40] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining Invariants from Console Logs for System Problem Detection. ATC '10.

[41] M. Jose and R. Majmudar. Cause Clue Causes: Error Localization Using Maximum Satisfiability. PLDI '11.

[42] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. SIGCOMM '11.

[43] J. Mccauley. POX: A Python-based OpenFlow Controller. http://www.noxrepo.org/pox/about-pox/.

[44] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. SOSP '08.

[45] ON.Lab. Open Networking Operating System. http://onlab.us/tools.html.

[46] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. SOSP '09.

[47] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. PLDI '12.

[48] P. Reynolds, C. Killian, J. L. Winer, J. C. Mogul, M. A. Shah, and A. Vadhat. Pip: Detecting the Unexpected in Distributed Systems. NSDI '06.

[49] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. SIGCOMM '14.

[50] V. Soundararajan and K. Govil. Challenges in Building Scalable Virtualized Datacenter Management. OSR '10.

[51] W. Sumner and X. Zhang. Comparative Causality: Explaining the Differences Between Executions. ICSE '13.

[52] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction. ISSTA '07.

[53] G. Tel. *Introduction to Distributed Algorithms*. Thm. 2.21. Cambridge University Press, 2000.

[54] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. SOSP '07.

[55] M. Weiser. Program Slicing. ICSE '81.

[56] A. Whitaker, R. Cox, and S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. SOSP '04.

[57] J. Xuan and M. Monperrus. Test Case Purification for Improving Fault Localization. FSE '14.

[58] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. A Survey of Fault Localization Techniques in Computer Networks. Science of Computer Programming '04.

[59] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. ASPLOS '10.

[60] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. EuroSys '10.

[61] A. Zeller. Yesterday, my program worked. Today, it does not. Why? ESEC/FSE '99.

[62] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE TSE '02.

[63] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Technical Report TR12-HPNG-061012, Stanford University '12.

[64] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. CoNEXT '12.

[65] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. NSDI '14.

[66] S. Zhang and C. Zhang. Software Bug Localization with Markov Logic. ICSE '14.