Formalizing Minimal Causal Sequences

Colin Scott Aurojit Panda Scott Shenker

1 Overview

In this document we attempt to formalize some of the questions left open by STS [7], a system for finding minimal causal sequences of inputs from buggy executions of software-defined networks.

In terms of problem statements, the closest work to ours is log reduction based on program flow analysis [6, 8, 11], which seeks to remove irrelevant events from deterministic replay logs (consisting of syscall return values, asynchronous signals, and a thread schedule) by analyzing the software to infer which events trigger code paths that do not effect whether the bug occurs. The crucial difference in our problem statement is that we treat the control software as a blackbox, and can only observe which messages the software sends in response to input events.

We start in §2 by formulating our own problem statement in detail.

2 Problem Statement

We represent the forwarding state of the network at a particular time as a configuration c, which contains all the forwarding entries in the network as well as the liveness of the various network elements. The control software is a system that takes a sequence of external network events $E = e_1, e_2, \ldots, e_m$ (such as link failures) as input, and produces a sequence of network configurations $C = c_1, c_2, \ldots, c_n$. Note that the network configuration c does not include the internal state of the control software.

An invariant is a predicate P over forwarding state (a safety condition, such as having no loops or blackholes). We say that a configuration c violates the invariant if $P(c)$ does not hold, denoted as $\overline{P}(c)$.

2.1 Log Input

We are given a log L of a system execution generated by a centralized QA test orchestrator. The log L contains external events $E_L = e_1, e_2, \ldots, e_m$, and timestamps $T_L = \{(e_k, t_k)\}$ of when the external events occurred, recorded from the test orchestrator's clock. A replay of log L involves replaying the external events along with a particular timing T, which need not be identical to the original timings T_L captured in the log. We denote a replay attempt by $replay(E_L, T)$. The output of $replay$ is a sequence of forwarding state configurations $C_R = \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$. In the ideal case

$replay(E_L, T_L)$ reproduces the same sequence of network configurations as occurred in the original execution, but as we discuss later this does not always hold.

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

2.2 Internal Events

As stated, the log (E_L, T_L) does not include information about events that are internal to the control software, including (a) message delivery events, either between controllers (*e.g.* database synchronization messages) or between controllers and switches (*e.g.* OpenFlow commands), and (b) state transitions within controllers (*e.g.* a backup node deciding to become master).

The interleaving of internal events and input events is important for ensuring that $replay$ reproduces the original invariant violation. In particular, if the following (conservative) condition holds [9], $replay(E_L, \cdot)$ will be guaranteed to reproduce the original violation:

Condition 2.1 *Each input e is injected only after all other events, including internal events, that precede it in the happens-before [5] relation ($\{i \mid i \rightarrow e\}$) from the original execution have occurred.*

Note that $replay(E_L, T_L)$ does not necessarily meet condition 2.1, since the control software may behave differently during replay due to non-determinism,[1] and using the original timings might cause the inputs to interleave improperly with internal events.

While it is not practically feasible for us to to observe and record all state transitions within control software, we can feasibly record message delivery events. We can therefore augment the original log (E_L, T_L) with a schedule $\tau_L = e_1 \rightarrow i_1 \rightarrow \ldots e_2 \rightarrow \ldots$, where each i is a message delivery event observed in the original execution.[2]

With this information, we schedule input events and message delivery events during $replay$ so that we enforce an order of events that is consistent with τ_L, *i.e.* we ensure that if $a \rightarrow b \in \tau_L$ then $a \rightarrow b$ during replay. While this does not imply that Condition 2.1 holds (since we do not observe all internal events), enforcing an order consistent with τ_L is a necessary condition for ensuring that Condition 2.1 holds.

2.3 Minimizing Inputs

The goal of our work is, when given a log (E_L, T_L, τ_L) that exhibited an invariant violation, to find a small sequence of events that reproduces that invariant violation. Formally, we define a minimal causal sequence (MCS) to be a subsequence E_M of E_L and a timing T_M such that $replay(E_M, T_M)$ reproduces the invariant violation,

[1] As we explain later, the control software may also behave differently as a result of pruning inputs from E_L.

[2] We further assume that we can arbitrarily reorder or drop message delivery events (through interposition) during $replay$.

but for all proper subsequences E_N of E_M there is no timing T s.t. $replay(E_N, T)$ reproduces the violation. That is, an MCS is a sequence and timing of external events that reproduces the violation, where one cannot find a subsequence of the MCS that reproduces the violation. Note that an MCS is not necessarily *globally* minimal, in that there could be smaller subsequences that reproduce this violation, but are not a subsequence of this MCS.

We employ delta debugging [10] to iteratively compute subsequences $E_S \subseteq E_L$, and then replay each E_S and check for an invariant violation. The responsibility of delta debugging is to efficiently search through the space of possible subsequences; the remaining task is to pick replay timings T such that, whenever delta debugging chooses a subsequence E_S that is a superset of some MCS, then $replay(E_S, T)$ reproduces the invariant. This is crucial to ensuring that delta debugging's final output is minimal. Since we do not know which subsequences are a superset of some MCS, our task is to always pick a timing T that could reproduce the invariant violation.

2.4 Selecting Timings

Assume each E_S is such that all logically dependent events occur, *e.g.* links go down before coming up, and hosts migrate along a coherent sequence of locations. Each subsequence of events E_S corresponds with timings for those events $T_S \subseteq T_L$.

As explained above, $replay(E_S, T_S)$ does not necessarily reproduce the invariant violation (even if E_S is a superset of an MCS), since it may violate Condition 2.1. Rather than replaying with T_S, we attempt to produce a schedule τ_S such that all events in τ_L are arranged in τ_S such that:

If $a, b \in \tau_S$ and $a, b \in \tau_L$ then $a \rightarrow b$ in τ_S if and only if $a \rightarrow b$ in τ_L.

This just says that τ_S follows the happens-before order reflected in the original log.[3]

Constructing τ_S involves three issues: coping with syntactic differences in internal events across replay runs (§3), handling internal events from the original execution that may not occur after pruning (dealt with in [7], section 4.2), and dealing with new internal events that were not observed at all in the original execution (§4).

3 Functional Equivalence

By choosing a subsequence of inputs to replay, *i.e.* pruning some of the original inputs, we may cause the messages sent by the control software to differ syntactically from those in the original run. For example, consider the sequence numbers of control messages: if we prune an input at the beginning of the execution, then the control software's sequence number counter may increment one less time than it did in the original execution, and all control packets sent by it will therefore carry a different sequence number than they did in the original trace. These syntactic differences complicate our goal of ensuring an event ordering that is consistent with τ_L.

[3]But not necessarily the full happens-before order from the original execution, since we do not have visibility into all internal events.

We observe that often these altered 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 (despite syntactic differences).

Formally, ...

4 Coping With Unexpected Events

Another possible change induced by pruning inputs is the occurrence of new internal events that were not observed in the original log. New events are an indication that the control software's state machine has diverged from the path it took in the original run. New events therefore present multiple possibilities for where we should inject the subsequent 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 subsequent input after i_2 or after i_3.

CS: Measurement question: what if we just immediately gave up on any subsequences that exhibited new events? How much minimization would we achieve?

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 existing suffix that leads to the invariant violation, we must recursively branch, trying both possibilities for every new internal event. This implies an exponential number of possibilities to be explored in the worst case.[4]

4.1 Modeling the Control Software

If we assume a model of the control software's state machine, we can potentially avoid exponential blowup. Assume we have a predicate $\Phi(\tau_P, a, b)$ that, given a prefix τ_P of events scheduled so far, and a pair of events a, b (either external or internal), returns true whenever $\tau_P||a \rightarrow b$ leads the software's state machine to a state s s.t. some buggy state $\hat{s}$ is reachable from s along a sequence of labeled state transitions α, where the labels in α that correspond to external events are a subsequence of the remaining external inputs $E_S \setminus \{e \in \tau_P\}$. **CS: May want to remove the requirement that α be a subsequence of external events. Alternatively, have Φ tell us whenever injecting a before b would lead us off the original path (or an coarse-grained version of the original path)** Here, a buggy state refers to a state that produces a network configuration c that violates a given invariant.

For all prefixes $\tau_P||a \rightarrow b$ of τ_L, if τ_L is a superset of an MCS, then $\Phi(\tau_P, a, b)$ returns true.

For any new internal event a and pending external event b, if $\Phi(\tau_P, a, b)$ returns true, then we know that we can allow a through before b. In this way we can interleave new internal events with our schedule of expected events τ_S, and be guaranteed to leave

[4] In our system [7], exponential search over these possibilities is not a practical option. The heuristic our system uses when waiting for expected internal events 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.

open the possibility of finding a divergent suffix through the state machine that leads to an invariant violation.

4.2 Obtaining a model

Two ideas for obtaining Φ:

Extrapolate from complete model. There was a paper in PLDI [2] where the authors wrote their own controller and verified its entire state space. We could issue queries on the state space to compute Φ, and then extrapolate Φ for other controllers assuming their state machines are sufficiently similar.

Infer partial models. In general we want to continue assuming that the control software is a black box. Even with that constraint, we could infer a partial model of the control software by observing its outputs across many executions, *i.e.* build a model from logs. We might use Synoptic [4] to compute the model.

5 Coping with Non-determinism

Completely coping with non-determinism involves two issues:

1. Obtaining visibility into internal state transitions.
2. Having control over internal state transitions during replay.

Let's assume that we have deterministic replay logging enabled during runtime. This gives us visibility.

Let's further assume that the OS under the control software provides us with primitives for enforcing determinism, as in dOS [3] or Determinator [1].

With these primitives, how would we completely cope with non-determinism?

6 What Systems This Works Well On

The characteristics of Φ would help us understand what other systems besides SDN control software our technique would work well on.

Two hypotheses for existing constraints on Φ: quiescence, and centralization. Possibly also: single system image (RSMs).

7 Partial Program Flow Analysis

Suppose we ran program flow analysis on our mock network (to predict internal events it will trigger), but not on the control software? This would still allow us to remain agnostic to the language of the control software, but it might help us get better replay success.

References

[1] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. OSDI '10.

[2] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. PLDI '13.

[3] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. OSDI '10.

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

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

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

[7] C. Scott, A. Wundsam, B. Raghavan, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting SDN Control Software with Minimal Causal Sequences. Technical Report TR-DOWNLOAD, UC Berkeley '14.

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

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

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

[11] X. Zhang, S. Tallam, and R. Gupta. Dynamic Slicing Long Running Programs Through Execution Fast Forwarding. SIGSOFT '06.