Project Proposal, Inria 2017

Stewart Grant

1 Overview

A common trend in distributed computing is the mass execution of small homogeneous processes. This pattern is prevalent in big data processing, and micro service based architectures. A key tenant of these processes is minimal statem, A consequence largely due to the complexities of designing massive systems which coordinate stateful behavior. Here we propose a technique for analyzing the stateful behavior of thousands of homogeneous processes, over many executions. Our analysis composes the logs of all processes into a single aggregate state machine. This model is replicated in a runtime, and is used to simulate new executions which are checked for specified safety and liveness properties. We propose two models for constructing aggregate state machines. First a theoretical model which generates no false positives, but which is constrained to minimal systems. Second a pragmatic model for large systems which permits false positives. Our second model is paired with additional novel analysis which orders false positives by their likelihood to be real violations.

2 Intro

Building large scale distributed systems is challenging. Rather than face the headaches of coordinating thousands of machines by hands, developers use frameworks which hide the complexity [6, 24, 19]. These frameworks sacrifice performance for generality and simplicity [18]. Recent work has demonstrated that some tasks benefit greatly from the implementation of custom stateful algorithms [8]. Such performance gains are not feasible for the majority of developers, as the difficulty of checking the correctness of their systems remains high.

Static and dynamic analysis are common approaches for building correct distributed systems. Languages such as TLA+ and COQ [15, 5] are usefully for specifying systems, and use model checkers to ensure that safety and liveness properties are not violated. These checkers require extreme amounts of computation to verify and require the maintenance of a fully flushed out specification, in addition to a complete implementation. Further, the gap between specification, and implementation admits bugs. Dynamic model checking techniques check implementations by systematically exercising a system, or replaying known faults [20, 22]. While these approaches are less costly computationally, they represent a strict under approximation of the systems behavior.

In this work we propose a dynamic analysis technique which simulates executions using logs to reach safety violating states not reached during execution. Our technique uses the logs generated from many executions of a system and composes an aggregate state machine of a single *super* process, which summarizes all logged behavior. A runtime environment replicates these machines, and systematically steps through state transitions. User specified safety and liveness conditions are checked during simulation. Violations, and their corresponding simulated traces are output to the user.

Our state machine aggregation algorithm builds on one fundamental observation - If the logged state of any two processes match exactly, then their states are the same. Our state machines are built by finding all occurrences of matching state from all processes on all executions. The same matching rule is applied to messages, and local events which trigger state transitions. Using exact state matching, distributed states not reached during execution can be observed through simulation, further all violations are real bugs.

Our exact state matching algorithm is the extension of theoretical literature [12]. While correct, their **State Matching** conditions are rarely met in practice, for instance *ip: port* combinations alone fragment an aggregate state machine into a sparse graph closely resembling traces themselves. To apply our analysis in practice we extend our notion of exact state matching to relaxed state matching, where the logged states of processes represent the same state in our model if a subset of their states match. This relaxation of state matching collapses the size of a state machine, but over approximates state transitions, thereby permitting false positives.

We propose an additional analysis procedure to order violations detected using relaxed state matching, by their likelihood to be real violations. Prior to simulation data invariants are collected, for each FSM node, from state

traces on variables which do not match. Operations which define transitions between nodes on non matching variables are approximated using program synthesis. Operations are synthesized using state transitions as input output examples as inputs to the popular Z3 SMT solver. During simulation non matching variables are approximated by applying operations generated by the synthesis. False positive likelihood is evaluated by checking invariant violations generated by the simulated execution. Traces which violate the minimal number of invariants are reported to the user, as they are least likely to have diverged from the systems constrained behavior.

The rest of the paper is arranged as follows. Section 3 Defines our model of a distributed system, and FSM construction. Section 4 describes our system. Sections 5 & 6 outline a proposed evaluation, and timeline.

3 model

In the following section we describe our model of a distributed system, and state machine. We then further extend our state machine model to a relaxed version.

3.1 System Model

Execution: An execution of a distributed program is defined as a set of n processes $P_1, P_2, \dots P_n$ all of which execute the same source code, with potentially different configurations.
Process State: The state s of any process P during an execution is the set of m variables v where $s = \{v_1, v_2, \dots v_m\}$, including the program counter. All processes share a unique initial state s_0.
Event: The set of events E is a finite alphabet of events which can be generated by any process. Our model restricts events to 3 general types *Sending, Receiving, and Local*.
Event State: Event state C (channel state) is a set of 1 or more variable values associated with an event. The state of a sending or receiving event, is the set of all transmitted variables. The event state of a local event, is the prior state of the process P.
Trace: A trace T of process P_i is the sequence of k (State, Event) pairs $T_i = (s_0 : e_0), (s_1, : e_1), \dots, (s_k, e_k)$.
Trace Matrix: A trace matrix M, is an n,m matrix in which index i,j is the trace generated by process j on execution i.

3.2 FSM Model

State Matching $\forall i, j\ s_i = s_j \iff \forall v \in s, v_{i,k} = v_{j,k}$.
Event Matching $\forall i, j\ e_i = e_j \iff \forall v \in s, v_{i,k} = v_{j,k} \wedge$ *eventType* $e_i ==$ *eventType* e_j.
Node A unique node n exists for all sets of matching states.
Edge A directed edge is defined as the triple (s_i, e, s_j). An edge exists between two nodes $n_k, n_l \iff \exists$ trace T which contains $(s_i : e_i), (s_{i+1} : e_{i+1})$. Two edges match if their states, and event match.

3.3 Relaxed FSM

Relaxed State Matching states s_i, s_j match $\iff \exists rs \in s$ where rs_i matches rs_j and $|s| - |rs| \geq k$.
Relaxed Event Matching events e_i, e_j match $\iff \exists re \in c$ where re_i matches re_j and $|e| - |re| \geq l$.

4 System

The following section describes our analysis system. The section is broken up into 3 section. First the translation of distributed logs to aggregated FSM is detailed. Second state invariant inverence is discussed, and SMT guided operation synthesis. Finally we discuss our execution engine and fault detectors. An example of our analysis, in the form of the dining philosophers algorithm is detailed in Figures 1, 2, 3, & 4, which are referenced throughout this section.

4.1 FSM Generation

Prior to execution, a system must be instrumented to log its state, and the state of sent and received messages. For this purpose we make use of the Dinv runtime (*citation pending*). Dinv logs the state of individual processes during execution. Using automatic instrumentation, all in scope variables are written to a key-value store at the entrance and exit of each function. Upon executing a sending, receiving or local event, the contents of the key value store are persisted to disk (or aggregated to a central source) with a corresponding vector timestamp. Each write to disk corresponds to a trace pair (s_i, e_i). Post execution the logs of one or more executions are aggregated together for FSM generation. Figure 1 is a simplified buggy version of the dining philosophers. Figure 2 are example traces of this program.

Logging State: Defining a distributed system by the state of each process, and the state of messages, is a common distributed model for many algorithms, and capturing all process, and message state is the fundamental

backbone of many distributed algorithms [16, 17]. Further, the reduction of distributed executions to communication events is a common method for reducing complexity while retaining essential information [9, 2, 11, 14, 10].
FMS Graph Construction: FSM nodes are built by matching states. User configuration determines how matching is performed. By default **State Matching** is used to match states as it guarantees correctness [12]. Figure 3 is an example of exact state matching performed on the traces in Figure 2. Users may also specify a subset of named variables for **Relaxed state matching**. Matching states are processed in linear time by hashing and mapping variable states. Similarly edges are constructed using **Event Matching**. As above relaxed event matching is performed on a subset of user defined variables.

4.2 Invariant Detection, and operation inference

FSM's generated using exact state matching are a strict under representation of a systems behavior, a fundamental consequence of dynamic analysis. Therefore, all safety, and liveness violations are true violations. In practice exact state matching results in a massive FSM with few inferred paths to traverse (as the likelihood of variables such as buffers, id's and ports matching is low). Relaxed state matching generates a smaller FSM. However, matching on a subset of state allows false positives in both safety, and liveness detection. Here we present a novel technique for identifying safety and liveness conditions on relaxed state matching which orders violations by their likelihood of being false positives. This technique uses simulated values for variables which do not match, and invariant violations as a heuristic measure of divergence from the real system.

Each node n is composed of a set of matching states, $n = s_0, s_1, \ldots, s_n$. Some subset of variables in each matching state $s = v_0, v_1, \ldots, v_m$ match, while another subset of variables do not. The subset of variables which do not match form a sub trace which profiles their behavior. We use Daikon to detect data invariants which held on each sub trace during execution [7].

Edges connecting nodes are state transitions with an associated operation on the state. In the case of **State Matching** transitions between state, are equivalent to an assignment statement (IE if node n and n' are constructed from exactly matching states, for any transition e between states $\forall i,\ v_i == v'_i$. **Relaxed State Matching** is more complicated, because many values may exist for any variable v in state n. A transition between two states may not be valid in the case of **Relaxed State Matching**, as the

```
const (
    sleep = 0
    hungry = 1
    eating = 2
)

var (
    state                int
    outstandingRequests  int
    acksReceived         int
)

func wake() {
    state = hungry
    outstandingRequests = 0
    acksReceived = 0
}

func eat() {
    state = eating
    sleep()
}

func sleep() {
    state = sleep
    outstandingRequests = 0
    acksReceived = 0
}

func sendReq() {
    send(req)
    outstandingRequest++
}

func recReq() {
    if state == hungry ||
       state == sleep {
        reply(ok)
    }
}

func recAck() {
    acksReceived++
    if acksRecived >= 2 {
        eat()
    }
}
```

Figure 1: Psudo code for dining philosophers for a buggy version of the dining philosophers. In this program philosophers must request permission to eat, and do so once they have received acknowledgments from at least two other philosophers. This specification has no requirement on which philosophers Ack requests, or how many times a single philosopher ac ks a request. Further this implementation does not guarantee fairness.

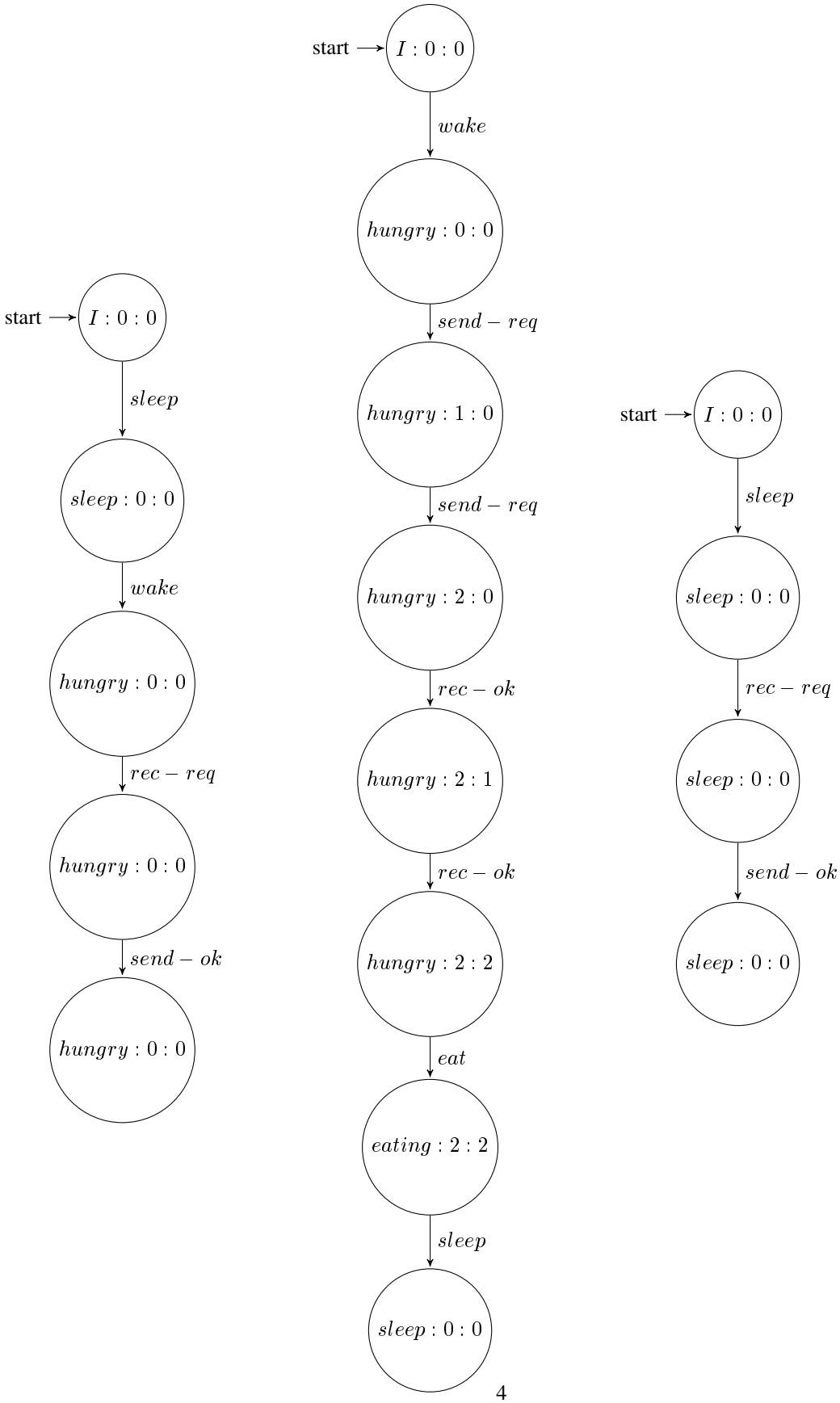

Figure 2: Dining philosopher traces collected from execution of Figure 1. Each process maintains 3 state variables, [$state$,$OustandingRequests$,$AcksReceived$]. Each vertex of these traces is an example of a single state s, and each edge an example of e, Together they for state event pairs (s, e). Sent and received messages are not shown explicitly in these traces, However the rec and $send$ prefix on events denote received and sent messages respectively. Note that from these traces no safety conditions are violated.

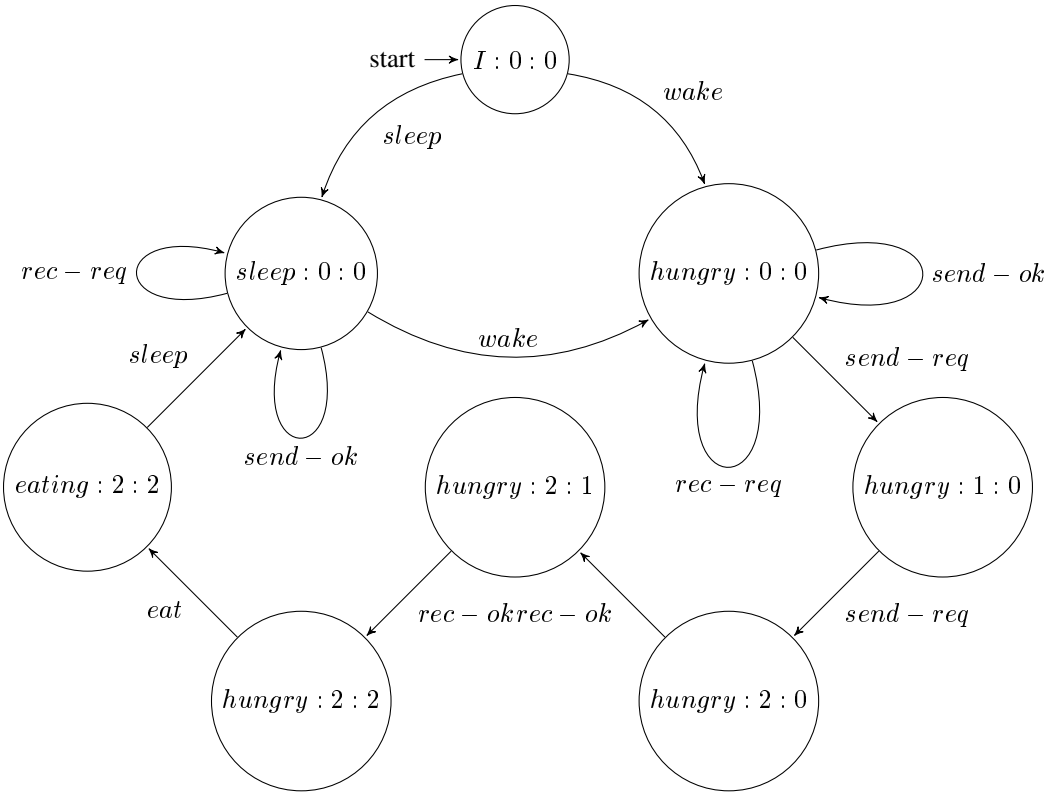

Figure 3: Aggregate state machine from 3 traces, using exact state matching. The states $sleep : 0 : 0$ and $hungry : 0 : 0$ aggregate the most state transitions as they are visited by more than one trace. $Hungry :> 0 :> 0$ states are spread out, due to their unique states. During runtime this state machine is replicated up to n times, and interacts with its replicas to reach unobserved states. Note that safety conditions can be violated with this state machine, as any node in $sleep : 0 : 0$ or $hungry : 0 : 0$ may issue *send - ok* events an unbounded number of times.

aggregate state is an over approximation of the systems observed behavior.

To reduce false positives when applying **Relaxed State Matching** the values of unmatched variables are simulated during runtime. If the trace produced by the runtime violates invariants detected from real executions, the probability of a violation being a false positive increases. One contribution of this research is to establish the quality of this heuristic for reducing false positives.

To simulate variable values, we synthesize operations using Z3, and input output examples from traces [23]. The state transitions inferred by Z3 are applied to variables during runtime. In cases where no operations could be inferred within a given timeout, a value is deterministicly chosen for relaxed variable. Synthesis of large programs is intractable. However, synthesis performs well when operating on constrained data, and operations [21, 13]. A key insight of this work is that state transitions used in distributed protocols are often simplistic, consisting of linear operations on integers, and assignments to state variables. Synthesis used in our analysis limits itself to *boolean*,*string*,and *integer* types, and only synthesizes relations on operators in the set $\{+, -, *, /, \%, ==\}$. In cases where no simple algorithmic fit can be made, a static assignment from a logged state is assigned for the state transition. Figure 4 is an FSM built from relaxed state matching, with state only matched on the variable $state = \{Hungry, Sleep, Eating, \perp\}$. Nodes are labeled with invariants, while edges are labeled with synthesized operations.

4.3 Runtime

Our runtime executes replicated versions of the inferred state machine. Each machine can send and receive messages to any other machine, and execute local events. The number of state machines to execute is a user defined parameter. More machines increase runtime, but may detect subtle bugs reliant on complicated multi machine state. Algorithm 1 overview our runtime engine.

Initially all machines are set to an empty initial state $\perp$. All valid events are generated at the beginning of the main runtime loop. A valid event is the set of all outgoing edges of all the current state of all nodes. Each event is added to an event queue and is applied systematically. Three kinds of events can be applied at runtime.

- *Local Event* A local event transitions a single node from one state to another.
- *Send Event* A send event generates a message, which is placed on an outstanding message list

Data: $Replication, Depth, Model, Conditions$
Result: Condition Violating Traces
while $Depth \geq 0$ **do**
  $eventQueue \leftarrow GenEvents(Model, Replication)$
  while $\neg eventQuene.Empty()$ **do**
    $ApplyEvent(eventQueue.Pop())$
    $CheckSafty(Conditions)$
    $CheckLiveness(Conditions)$
    $Recurse(Replication, Depth --, Model, Conditions)$
  end
end

Algorithm 1: Runtime Algorithm

- *Receive Event* A receive event consumes a message. Two conditions exist for receiving messages, droppable, and undroppable. Droppable message can be consumed by any machine in any state, if no transition exists for the message, the state of the received machine does not change. Undroppable messages are only delivered to machines which are in a state with a transition corresponding to the message being received [22].

Messages are consumed by only a single node. As our search is performed systematically, all permutations of message delivery are eventually explored.

While executing a trace of variables is maintained. Variables are updated during each state transition. If the state transitioned into is exactly matched, runtime variables are assigned the value of the state. Otherwise the operations determined by constraint solving are applied to the variables.

The runtime environment has a holistic view of the system, and is therefore able to check safety predicates at all times. Predicates defined by a user specification, such as only one machine may enter the critical section at a time, or at least one node has a token, are checked on a per variable basis.

As of writing this document no specification language has been chosen to specify safety and liveness properties. A common way to specify safety properties are as regular expressions on events, and states [11], as well as **Pos/Def** global predicates. Liveness properties are typically encoded as LTL or CTL formula. As the proposed analysis technique generates traces, and has full observation of generated execution, all specifications should theoretically be valid input. For the purposes of this project checkers will be designed to be plugable to support, a variety of specification languages.

Liveness conditions are checked using techniques developed by [12]. If a periodic pattern is found in a dis-

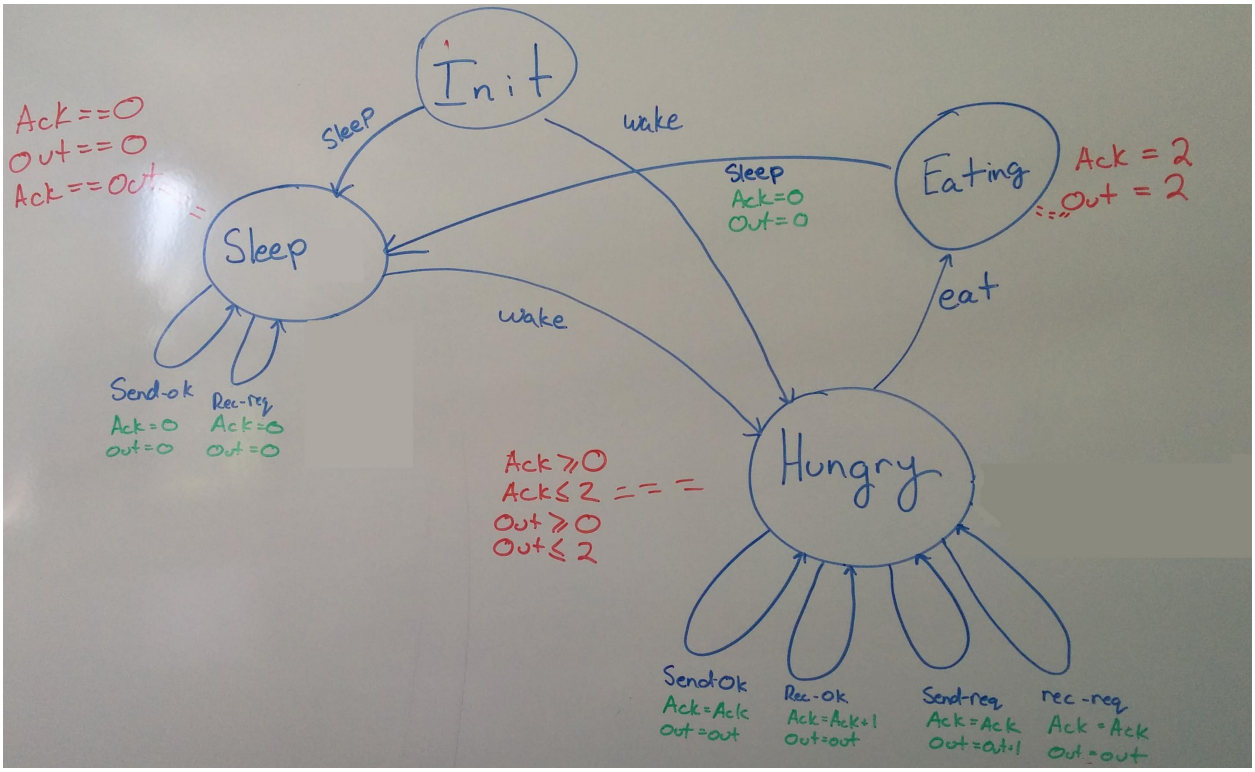

Figure 4: The dining philosophers FSM build using relaxed state matching. Only the $state$ variable is matched on, while $OutstandingRequests$ (out), and $AcksReceived$ (Ack) are not matched on. Constraints in red Are invariants inferred on nodes of the FSM. Operations in green are transition operations inferred using SMT guided program synthesis.

tributed execution, ie if at two points in an execution, the state of the system is exactly the same, the events between the two states can periodically execute forever (given non random liveness guarantees). If exact states are revisited during runtime, and a liveness condition is not met over the interval between matching states, a liveness violation is reported.

4.4 Violations & false positive reduction

The result of simulating is a set of traces on which a safety or liveness violation occurred. Traces contain vector time stamps for simulated events, which are formatted to be visualized by ShiViz, to aid in debugging [1]. Traces generated using **State Matching** and **Event Matching** are guarantees bugs, the traces of which can be traversed to identify the root cause of the bug.

Traces generated using **Relaxed State Matching** and **Relaxed Event Matching** may be false positives. These traces are approximately ordered by their likelihood to be false positives. Each simulated trace (containing simulated variable values at runtime) is analyzed by Daikon. Using Daikons Invariant Difference tool the count of invariant violations generated by the simulated execution can be determined. Traces which violate the minimum count of invariants are reported to the user first, as they are least likely to have diverged from the systems real behavior.

5 evaluation

The analysis technique proposed in this work should be demonstrated to scale to thousands of processes over multiple executions. Further it should be shown that the time to replay executions is orders of magnitude lower than executing the system itself. The speedup boasted by Modist is 200x because they sped up timeouts. Because the majority of computation is skipped our numbers should be much lower. The cost of generating models may be high, this should be shown to be an amortized cost over many profiled executions. Finally, effort should be minimal for the end user. Given that Dinv can already generate logging statements for all in scope variables, logging should be automatic. Users should only have to capture network functions & specify properties they want to check on their systems.

Candidate systems for checking are as follows

- Glow Map-Reduce in golang [3]. Individual programs could be show to be correct, or the framework itself could be checked
- Gleam A general Map/Reduce, DAG execution system [4], Similar to the project above, but more general
- ExCamera [8]. A custom logger (Java) could be written to capture the state of this algorithm
- TensorFlow programs require stateful interaction. An evaluation of TensorFlow would involve analyzing the framework itself. Further investigation is needed to understand their consistency mechanism. If it is the same timely flow as Naiad, it has well defined predicates.

6 Timeline

- **June 6 - 15** Read and get familiar with associated work
- **June 16 - 22** Brainstorm new ideas
- **June 23 - 28** Build Proposal and small proof of concepts for each component
- **June 29 - July 10** Implement log capturing system and build exact matching FSM
- **July 11 - July 17** Implement relaxed matching FSM (Test on simplified Dining philosopher)
- **July 18 - July 28** Build simulation runtime (generating, and scheduling events, and building traces)
- **July 29 - Aug 10** Add simulated variables to runtime and invariant divergence checking
- **Aug 11 - 17** Apply technique to multiple executions of ETCD raft
- **Aug 18 - 25** Apply analysis to Hadoop / Excamera / Naiaid
- **Aug 26 - 31** Write up results and documentation for source code

References

[1] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst. Shedding light on distributed system executions.

[2] O. Babaoglu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *Journal of Parallel and Distributed Computing*, 28(2):173 – 185, 1995.

[3] chrislusf. Glow. https://github.com/chrislusf/glow, 2016.

[4] chrislusf. Gleam. https://github.com/chrislusf/gleam, 2017.

[5] P. Corbineau. A declarative language for the coq proof assistant. In *Proceedings of the 2007 International Conference on Types for Proofs and Programs*, TYPES'07, pages 69–84, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[8] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[9] E. Fromentin, C. Jard, G.-V. Jourdan, and M. Raynal. On-the-fly analysis of distributed computations. *Inf. Process. Lett.*, 54(5):267–274, June 1995.

[10] E. Fromentin and M. Raynal. Shared global states in distributed computations. *Journal of Computer and System Sciences*, 55(3):522 – 528, 1997.

[11] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson. On the fly testing of regular patterns in distributed computations. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, volume 2, pages 73–76, Aug 1994.

[12] V. K. Garg, A. Agarwal, and V. Ogale. Modeling, analyzing and slicing periodic distributed computations. *Inf. Comput.*, 234:26–43, Feb. 2014.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. January 2011.

[14] M. Hurfin, M. Mizuno, M. Singhal, and M. Raynal. Efficient distributed detection of conjunctions of local predicates. *IEEE Trans. Softw. Eng.*, 24(8):664–677, Aug. 1998.

[15] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and verifying systems with tla+. page 4548, Saint-Emilion, France, September 2002. Association for Computing Machinery, Inc.

[16] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[17] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423 – 434, 1993.

[18] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.

[19] D. G. Murray, F. Mcsherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system.

[20] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing Faulty Executions of Distributed Systems. In *NSDI*, 2016.

[21] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM.

[22] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[23] D. Yurichev. Quick introduction into sat/smt solvers and symbolic execution. https://yurichev.com/writings/SAT_SMT_draft-EN.pdf.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.