Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight

Ivan Beschastnikh*, Yuriy Brun†, Michael D. Ernst‡, Arvind Krishnamurthy‡
*Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
†School of Computer Science
University of Massachusetts
Amherst, MA, USA
‡Computer Science & Engineering
University of Washington
Seattle, WA, USA
bestchai@cs.ubc.ca, brun@cs.umass.edu, {mernst, arvind}@cs.washington.edu

ABSTRACT

Concurrent systems are notoriously difficult to debug and understand. A common way of gaining insight into system behavior is to inspect execution logs and documentation. Unfortunately, manual inspection of logs is an arduous process, and documentation is often incomplete and out of sync with the implementation.

To provide developers with more insight into concurrent systems, we developed CSight. CSight mines logs of a system's executions to infer a concise and accurate model of that system's behavior, in the form of a communicating finite state machine (CFSM).

Engineers can use the inferred CFSM model to understand complex behavior, detect anomalies, debug, and increase confidence in the correctness of their implementations. CSight's only requirement is that the logged events have vector timestamps. We provide a tool that automatically adds vector timestamps to system logs. Our tool prototypes are available at http://synoptic.googlecode.com/.

This paper presents algorithms for inferring CFSM models from traces of concurrent systems, proves them correct, provides an implementation, and evaluates the implementation in two ways: by running it on logs from three different networked systems and via a user study that focused on bug finding. Our evaluation finds that CSight infers accurate models that can help developers find bugs.

Categories and Subject Descriptors: D.1.3 [Concurrent Programming]: Distributed programming
General Terms: Algorithms, Design, Modeling
Keywords: Model inference, log analysis, concurrency, distributed systems, CSight

1. INTRODUCTION

When a system behaves in an unexpected manner, or when a developer must make changes to legacy code, the developer faces the challenging task of understanding the system's behavior. To help with this task, developers often enable logging and analyze runtime logs. Unfortunately, the size and complexity of logs often exceed a human's ability to navigate and make sense of the captured data.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2756-5/14/05
http://dx.doi.org/10.1145/2568225.2568246

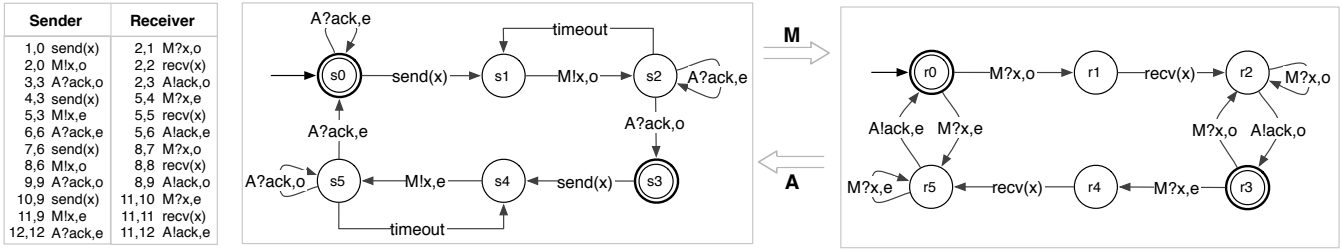
One promising approach to help developers is model inference. The goal of a model-inference algorithm is to convert a log of system executions into a model, typically a finite state machine, that accurately and concisely represents the system that generated the log. Numerous model-inference algorithms and tools exist to help debug, verify, and validate *sequential* programs [11, 42, 43, 10].

Unfortunately, this rich prior body of work is not directly applicable to distributed or concurrent systems. This is because a common assumption made in model inference is that the underlying set of executions is totally ordered — for every pair of events in an execution, one precedes the other. This assumption is crucial to the inference algorithms' correctness, and running them on logs that are not totally ordered results in inaccurate models. Unfortunately, this assumption does not hold for concurrent systems, such as networked systems, for which events at different nodes may occur without happens-before relationships [39]. Additionally, most model-inference algorithms infer finite state machine (FSM) models, which are inappropriate for modeling multi-process implementations.

This paper describes a new model-inference technique and a corresponding tool, called CSight (for "concurrency insight"), which infers a communicating finite state machine (CFSM) [14] model of the processes that generated the log. CSight can be applied to logs of distributed systems, protocol traces, traces of AJAX events in a web-browser, and other concurrent behavior traces. CSight models have multiple developer-oriented uses — for example, developers can inspect, query, and check CSight's models against their own mental model of the system to find bugs. These uses are the focus of our present work. However, we believe that CSight-generated models have numerous other applications, such as model-based testing of concurrent systems, and automated detection of anomalous behavior as systems are exposed to new workloads and environments.

This paper presents the following four contributions:

1. A novel model-inference algorithm to infer a CFSM model from a log of vector-timestamped concurrent executions (Section 3), and a corresponding tool called CSight.
2. A proof that CSight-generated models are accurate as they satisfy several key properties of the input log (Section 4).
3. A tool for automatically augmenting existing process-local logging mechanisms with vector-timestamping logic. This makes it easy to apply CSight to existing systems (Section 5).
4. A two-pronged evaluation of CSight (Section 6):
   - We applied CSight to logs of three systems — the stop-and-wait protocol, the opening/closing handshakes of TCP, and the replication strategy in the Voldemort [56] distributed hash table. CSight was effective for uncovering the true model for each of these systems.



(a) Input log (b.1) Output model (Sender) (b.2) Output model (Receiver)

Figure 1: Example input and output of CSight for the stop-and-wait protocol (SAW) [54]. (a) Example input log with a single trace of two processes running SAW. The two integers at the beginning of each event in the log are the event's vector clock timestamp. (b) The CSight-derived CFSM model of SAW, derived from a log with additional traces (not-shown), consisting of (b.1) the sender process model and (b.2) the receiver process model. In SAW, the sender transmits messages to the receiver using channel `M`, and the receiver replies with acknowledgments through channel `A`. Notation `Q!x` means enqueue message `x` at tail of channel `Q`, and event `Q?x` means dequeue message `x` from the head of channel `Q`. The event `send(x)` is a down-call to send `x` at the sender, and `recv(x)` is an up-call at the receiver indicating that `x` was received. In the SAW implementation, `x` is a variable that stands for an arbitrary message string. CSight, however, treats the logged event instances, like "send(x)" and "timeout", as strings and does not interpret them. The `timeout` event at the sender triggers a message re-transmission after some internal timeout threshold is reached. The "alternating bit" is associated with each message and is appended to a message before it is sent. For example, the first (and every odd) message sent by the sender is represented as `x,o` (`o` for odd) while every even message sent by the sender is encoded as `x,e` (`e` for even).

- We performed a user study with a class of 39 undergraduates to evaluate the efficacy of CFSM models in finding bugs. We found that CFSM models are just as useful in finding implementation bugs as time-space diagrams, a popular alternative for visualizing concurrent executions.

We start with an overview of the CSight algorithm (Section 2).

2. CSIGHT OVERVIEW

CSight's input is a log of observed system execution events, and its output is a model that describes the concurrent system that generated the log. An input log consists of *execution traces* of the system. A trace is a set of events, each of which has a vector timestamp [25, 45]. Vector time is a standard logical clock mechanism that provides a partial order of events in the system. Section 5 describes a tool that automatically adds vector time tracking to existing systems.

Figure 1(a) shows an example input log generated by two processes executing the stop-and-wait protocol [54]. In this protocol, a *sender* process communicates a sequence of messages to a *receiver* process over an unreliable channel. The receiver must acknowledge an outstanding message before the sender moves on to the next message. If a message is delayed or lost, the sender retransmits the message after a timeout.

Figure 1(b) shows CSight's output — a communicating finite state machine [14] (CFSM) model. A CFSM models multiple processes, or threads of execution, each of which is described by a finite state machine (FSM). In the standard CFSM formalism, processes communicate with one another via message passing over reliable FIFO channels. However, unreliable channels can be simulated by replacing each unreliable channel with a lossy "middlebox" FSM that non-deterministically chooses between forwarding and losing a message. The model in Figure 1(b) handles message loss, but the lossy middlebox is not shown in the diagram.

We use the CFSM formalism because it is similar to the widely known FSM formalism. CFSMs are well-established in the formal methods community, and we believe (and empirically verify in Section 6) that a CFSM is intuitive and sufficiently simple for developers to comprehend. For example, a single-process FSM in a CFSM can be inspected and understood without needing to understand the activity of other processes in the system.

CSight infers a CFSM (e.g., Figure 1(b)) by transforming the input log (e.g., Figure 1(a)) through a series of representations and analyses. Figure 2 details this process. The CSight process has three key stages:

Stage 1: temporal property mining. CSight mines temporal properties, or invariants, from the log (steps ① and ② in Figure 2). For example, for the log in Figure 1(a), CSight would mine the property "event instance string `send(x)` always precedes the event instance string `receive(x)`". Section 3.2 describes all the types of properties CSight mines. CSight uses these properties to judge the accuracy of the current model, as it refines the model to accurately describe the log.

Stage 2: create initial model. CSight uses the log to build a compact model (steps ③ and ④ in Figure 2). This model generalizes all the executions in the log, and it is inaccurate in that it admits executions that violate the properties mined from step one. To build this small model CSight assumes that whenever an event of a particular type executes, the system must be in a unique state associated with events of that type. For example, every time the log has a `send(x)` event, CSight assumes the system was in the exact same `send(x)` state. This creates an overly permissive but highly compact model.

Stage 3: refine the model. CSight gradually changes the initial compact model from stage 2 into a larger model that satisfies all of the mined properties from stage 1 (steps ⑤ and ⑥ in Figure 2). To accomplish this, CSight checks which of the mined properties are not satisfied by the present model. A property is not satisfied if the model allows an execution that violates the property. CSight checks properties with model checking, which either guarantees that the property is satisfied by all modeled executions, or finds a counter-example execution that violates the property. If a counter-example exists, CSight refines the model to eliminate that counter-example using the CEGAR approach [16]. CSight repeats the model-checking and refinement loop until all of the mined properties are satisfied, at which point it outputs the resulting model.

The next section formally describes the CSight process and details each of the steps in Figure 2.

3. FORMAL DESCRIPTION OF CSIGHT

In Section 4, we will prove three important properties about CSight's model-inference process:

- **Inferred model fits the input log.** The final model accepts all the observed traces in the input log (see Theorem 1).
- **Refinement always makes progress.** Every iteration of the refinement process (stage 3 in Section 2) makes progress toward satisfying all of the mined invariants (see Theorem 2).

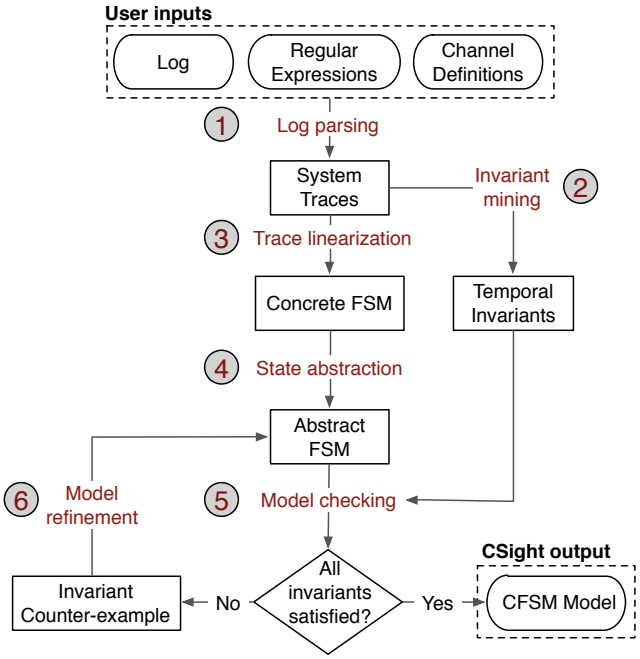


Figure 2: CSight process flow. Section 3 describes the numbered steps.

- **Inferred model satisfies mined invariants.** Every trace accepted by the final model behaves like the observed traces; that is, every invariant that is true of the observed traces is also true of every accepted trace (see Theorem 3).

To enable proving these properties, we must first formalize the problem domain. A reader who wishes to only get an intuitive understanding of the approach can skim this section.

We start by defining CSight's inputs (Section 3.1) and the invariants that CSight mines (Section 3.2). Then, we specify how CSight converts a log into a concrete FSM (Section 3.3) and how CSight abstracts the concrete FSM into the initial abstract FSM (AFSM) (Section 3.4). Finally, we describe how CSight model-checks and refines the AFSM to satisfy the valid invariants, and how the AFSM is converted into a CFSM model (Section 3.5).

First, we describe the notation used in the rest of the paper. Given an n-tuple t, let $|t| = n$, and for all $1 \leq i \leq n$, let $t[i]$ refer to the i^{th} component of t; for $i > n$, let $t[i] = \varepsilon$. We write $t'' = t \cdot t'$ to denote concatenation of two tuples: t'' has length $|t| + |t'|$, and for all $1 \leq i \leq |t|$, $t''[i] = t[i]$ and for all $1 \leq i \leq |t'|]$, $t''[|t| + i] = t'[i]$. We call t a *prefix* of t'' iff $\exists t'$ such that $t'' = t \cdot t'$. The projection function π maps a tuple t and a set S to a tuple t' such that t' is generated from t by deleting all components of t that are not in S. Finally, we use the symbol $\hat{}$ to denote variables that represent abstract objects in the modeling process and to differentiate these from concrete features of the input log. For example, for a logged (concrete) event instance e the corresponding abstract event type is denoted as $\hat{e}$.

3.1 Expected Log Input

To use CSight, the user must provide an input *log*, a set of user-defined regular expressions, and a set of channel definitions (step (1) in Figure 2). The regular expressions determine which log lines are parsed (and which are ignored), what part of a line corresponds to a vector timestamp, and which local, send, or receive event the line represents. The channel definitions are used to associate send and receive events with inter-process channels.

Currently, CSight requires the input log to contain only complete, error-free executions. Lifting this requirement is future work.

The rest of this section formally describes the structure of the input log. We assume that an input *log* L is produced by a system composed of h processes, indexed from 1 to h. The log contains multiple *system traces*, each of which represents a single concurrent execution of the system. A system trace consists of a set of event instances logged by different processes and a *happens-before* relation [39] (a strict partial ordering over event instances). For example, the log in Figure 1(a) contains a single system trace; in this trace the vector timestamps encode the partial order.

Definition 1 (System trace)**.** A *system trace* is the pair $S = \langle T, \prec \rangle$, where $T = \cup T_i$ is the union of a set of process traces (see Def. 2), one per process; and $\prec$ is the happens-before relation. This relation partially orders event instances at different processes and totally orders event instances (see Def. 3) in each process trace T_i according to their positions in the trace. That is, $\forall e_i = \langle \hat{e}, i, k \rangle \in T_i$, $e_i' = \langle \hat{e}', i, k' \rangle \in T_i$, $e_i \prec e_i' \iff k < k'$.

S must satisfy three communication consistency constraints: (1) every sent message is received, (2) only sent messages are received, and (3) sent messages on the same channel must be received in FIFO order. The validity of the communication consistency constraints #1 and #3 depends on the underlying message passing protocol.[1]

We express the above three constraints with a bijection between message send and message receive event instances for every pair of processes. For all i, j, let $S_{ij} \subset T$ be the send events along channel c_{ij}, and let $R_{ij} \subset T$ be the receive events along channel c_{ij}. Then $\forall i, j, \exists$ a bijection $f : S_{ij} \to R_{ij}$, such that:

1. $f(s) = r \implies s \prec r$
2. $\forall s_1, s_2 \in S_{ij}, s_1 \prec s_2 \implies f(s_1) \prec f(s_2)$.

A system trace is composed of multiple process traces, and each process trace is the set of event instances generated by a specific process. We assume that each event instance is given a unique position k in a consecutive order, from 1 to the length of the trace.

Definition 2 (Process trace)**.** For the process i, a *process trace* is a set T_i of event instances, such that $\forall k \in [1, |T_i|]$, $\exists \langle \hat{e}, i, k \rangle \in T_i$, and $\langle \hat{e_1}, i, k_1 \rangle \in T_i$ and $\langle \hat{e_2}, i, k_2 \rangle \in T_i \implies k_1 = k_2$.

The execution of each process generates a sequence of event instances, each of which has an event type from a finite alphabet of process event types. These event types can be local events, message send events, or message receive events.

Definition 3 (Event instance)**.** For a process i, an *event instance* is a triple $e = \langle \hat{e}, i, k \rangle$, where $\hat{e}$ is the event type of e, and $k \geq 1$ is an integer that uniquely indicates the order (position) of the event instance, among all event instances generated by process i. When the value of k is not important, we denote this triplet as e_i.

Finally, we assume a fixed set of channels, each of which is used to connect one sender process to one receiver process. Each of the send and receive event types is associated with a channel.

Definition 4 (Channel)**.** A *channel* c_{ij} is identified by a pair of process indices (i, j), where $i \neq j$ and $i, j \in [1, h]$. Indices i and j denote the channel's *sender* and *receiver* process, respectively.

We use the standard notation `!` to denote send events, `?` to denote receive events, and use labels for channels. For example, in Figure 1, the event `M!x,e` is a send of message `x,e` on channel `M`.

[1] In this paper we focus on TCP, which satisfies both assumptions. Generally, TCP is used by the complex systems that CSight targets. Removing these assumptions is future work (see Section 8).

3.2 Invariant Mining

CSight uses the log to mine a set of temporal invariants — linear temporal logic expressions — that relate events in the log (step (2) in Figure 2). These invariants (Def. 5) are true for all of the observed execution traces. CSight guarantees that the final inferred CFSM model satisfies all the mined invariants.

Definition 5 (Event invariant)**.** Let L be a log, and let $\hat{a_i}$ and $\hat{b_j}$ be two event types whose corresponding event instances, a_i and b_j, appear at least once in some system trace in L. Then, an *event invariant* is a property that relates $\hat{a_i}$ and $\hat{b_j}$ in one of the following three ways.

$\hat{a_i} \rightarrow \hat{b_j}$ **:** An event instance of type $\hat{a}$ at host i is **always followed by** an event instance of type $\hat{b}$ at host j. Formally: $\forall \langle T, \prec \rangle \in L, \forall a_i \in T, \exists b_j \in T, a_i \prec b_j$.

$\hat{a_i} \not\rightarrow \hat{b_j}$ **:** An event instance of type $\hat{a}$ at host i is **never followed by** an event instance of type $\hat{b}$ at host j. Formally: $\forall \langle T, \prec \rangle \in L, \forall a_i \in T, \nexists b_j \in T, a_i \prec b_j$.

$\hat{a_i} \leftarrow \hat{b_j}$ **:** An event instance of type $\hat{a}$ at host i **always precedes** an event instance of type $\hat{b}$ at host j. Formally: $\forall \langle T, \prec \rangle \in L, \forall b_j \in T, \exists a_i \in T, a_i \prec b_j$.

For example, one invariant of the stop-and-wait protocol model in Figure 1 is `M?m-0` $\rightarrow$ `A?a-0`. The invariant types and the corresponding mining algorithms are described in more detail in [9]. The above invariant types are also exactly the most frequently observed specification patterns formulated by Dwyer et al. [24] for serial systems.[2] In our experience, these invariants were sufficient for capturing key temporal properties of the systems that produced the logs we considered.

3.3 Deriving a Concrete FSM

In step (3) of Figure 2, CSight creates a concrete FSM that accepts all possible linearizations of the partially ordered system traces in the log. The concrete FSM model closely fits the logged observations, but it is not a concise model. Later, we will show how CSight uses abstraction to make the concrete FSM concise.

3.3.1 Concrete State

To define a concrete FSM we need to introduce notions of state that describe the concrete observations in the log. For this, we will define process states, channel states, and system states.

Definition 6 (Local process state)**.** Each process begins execution in an initial state, q_i^ε, and after executing a sequence s of process i event instances, the process enters state q_i^s. More formally, let L_i be the set of process i traces in a log L, then the set of process i local states is Q_i:

$$Q_i = \{q_i^\varepsilon\} \bigcup \{q_i^s \mid \exists t \in L_i, s \text{ is a prefix of } t\}$$

We call q_i^s a *terminal* state for process i if and only if $s \in L_i$.

Now, we define the global process state and global channel state that together make up the system state.

Definition 7 (Global process state)**.** A *global* process state $q = \langle q_1, \ldots, q_h \rangle$ is a h-tuple that represents the state of all processes in the system. That is, $q \in Q = Q_1 \times \cdots \times Q_h$, with $q_i \in Q_i$ denoting a state at process i.

[2]Scope is constrained to a trace (i.e., global scope). The translation is not one-to-one: $\hat{a} \rightarrow \hat{b}$ is Dwyer et al.'s Existence pattern when $\hat{a}$ is the *start* event that precedes every trace, and is otherwise Dwyer et al.'s Response pattern. Another example is $\forall \hat{b}, \hat{a} \leftarrow \hat{b}$, which is Dwyer et al.'s Universality pattern.

A channel contains all sent messages not yet received. For each channel c_{ij}, a (possibly empty) finite set of *messages* M_{ij} are the only messages that can be *sent* and *received* on c_{ij}.

Definition 8 (Channel state)**.** The *channel state* w_{ij} of a channel c_{ij} is a tuple of variable length whose entries are messages that can be sent and received along c_{ij}. That is, $w_{ij} \in (M_{ij})^*$.

Definition 9 (Global channel state)**.** A *global channel state* w is a set of channel states for all channels in the system. More formally, $w = \{w_{ij} \mid w_{ij} \in (M_{ij})^*\}$. We reuse ε to also stand for a global channel state with all channels empty. Further, we denote the set of all possible global channel states M.

Finally, we represent the *system state* as a pair of global process state and global channel state, $\langle q, w \rangle \in Q \times M$.

3.3.2 Transitions

Next, we specify how a sequence of event instances impacts the state of the system. For this, we define a process transition function, δ_i, which maps a process i state and an event instance to a new state.

Definition 10 (Process transition function)**.** Let E_i be the set of all process i event instances in a log L and let L_i be the set of process i traces in L. Then, the *process transition function* for a process i is $\delta_i : Q_i \times E_i \rightarrow Q_i$, such that

- $\delta_i(q_i^s, e_i) = q_i^{s \cdot e_i} \iff \exists t \in L_i, (s \cdot e_i)$ is a prefix of t.

As an example that illustrates δ_i, assume that states are the natural numbers: $Q_i = \mathbb{N}$, and event instances E_i are totally ordered: $\exists g \colon E_i \rightarrow \mathbb{N}$. Then, we can define $\delta_i(q_i, e_i)$ as the number that is formed by concatenating q_i and $g(e_i)$.

Notice that δ_i has a distinguishing property — two local process states are different if they were generated by two distinct sequences of events. Or, more formally:

1. $e_i \neq f_i \iff \delta_i(q_i, e_i) \neq \delta_i(q_i, f_i)$
2. $q_i = q_i' \iff \delta_i(q_i, e_i) = \delta_i(q_i', e_i)$

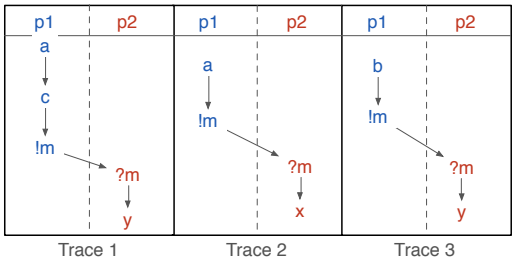
3.3.3 The Concrete FSM

CSight uses the system traces parsed from a log L to construct a *concrete FSM* $\mathcal{F}_L$ (step (3) in Figure 2). This FSM represents the observed, concrete executions of the whole system. Each state in the concrete FSM is a tuple of the individual process states and channel contents (for all processes and channels in the system). A channel's content is computed from the sequence of observed message sends and receives in the trace. Process states, however, must be inferred, and are assumed to be uniquely determined by the process history. In other words, for a specific sequence of events at a process, CSight creates a single unique *anonymous* process state. Figure 3 illustrates how the anonymous process states and the corresponding concrete FSM are derived from a set of input system traces.

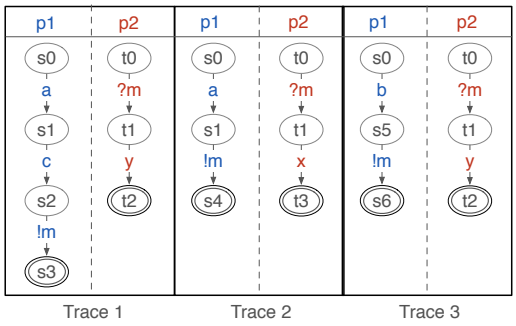
A key property of $\mathcal{F}_L$ is that it accepts all linearizations of all traces in L, as well as all possible traces that are *stitchings* of different concrete traces that share identical concrete states.

Definition 11 (Concrete FSM)**.** Given a log L, a *concrete FSM* $\mathcal{F}_L$ for L is a tuple $\langle S, s_I, E, \Delta, S_T \rangle$.

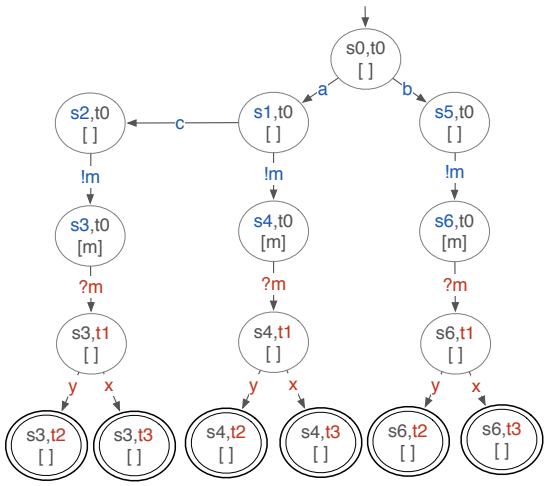
- The states of $\mathcal{F}_L$ are system states: $S = Q \times M$
- There is a unique initial system state: $s_I = \langle [q_1^\varepsilon, \ldots, q_h^\varepsilon], \varepsilon \rangle$
- $E = \{e \mid \exists \langle T, \prec \rangle \in L, e \in T\}$
- The transition function Δ is the composition of the process transition functions and handles communication events, $\Delta : Q \times M \times E \rightarrow Q \times M$, and $\Delta(\langle q, w \rangle, e_i) = \langle q', w' \rangle$, where:



(a) System traces



(b) Traces with anonymous states



(c) Concrete FSM

Figure 3: (a) System traces parsed from an input log with three traces, generated by two processes: p1 and p2. All messages flow from p1 to p2. (b) The traces from (a) with added per-process anonymous states. Note the reuse of states s0, s1, t0, t1, and t2. (c) Concrete FSM for the system traces in (a).

- $q_i' = \delta_i(q_i, e_i)$, and $q_j' = q_j$ if $j \neq i$.
- $w_{ij}' = \begin{cases} w_{ij} \cdot m & e_i = c_{ij}!m \\ tail & e_i = c_{ij}?m,\ w_{ij} = m \cdot tail \\ \text{undefined} & e_i = c_{ij}?m,\ w_{ij}[1] \neq m \\ w_{ij} & \text{otherwise} \end{cases}$

- The terminal states have empty channels, and each process is in a terminal state that was derived by executing all of the event instances for that process in some system trace in L. $S_T = \{\langle q, \varepsilon\rangle \mid \forall i, q_i \text{ is terminal}\}$

3.3.4 *Validating the Mined Invariants*

Although the invariants CSight mines (Section 3.2) are true of the input traces, in some cases it is not always possible to construct a concrete FSM model that satisfies the invariants and accepts all of the input traces. This situation is caused by the incompleteness of the log.

Our technical report [8] describes an *invariant validation* step that removes such invariants.

3.4 Abstracting a Concrete FSM

A concrete FSM is not concise — it is a DAG whose longest path is as long as the longest execution — and it is not sufficiently abstract. CSight generates a more concise *abstract FSM* model from the concrete FSM, using a process we call *state abstraction* (step (4) in Figure 2). As a reminder, we use the symbol ˆ to denote variables that represent abstract objects in the modeling process and to differentiate these from concrete features of the input log.

The concrete FSM $\mathcal{F}_L$ accepts all possible linearized sequences of event *instances* from executions in a log L. Let $\hat{\mathbf{P}}$ represent a partitioning of states in $\mathcal{F}_L$, that is a partitioning of $Q \times M$ (we use **bold font** to denote a set of sets)[3]. CSight's aim can now be defined as deriving an *abstract* FSM (AFSM) $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$ whose states are partitions in $\hat{\mathbf{P}}$ and that accepts sequences of *events* — rather than *event instances*. Transitions between states (partitions) in $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$ are generated through existential abstraction: there is a transition from $\hat{P}_1 \in \hat{\mathbf{P}}$ to $\hat{P}_2 \in \hat{\mathbf{P}}$ on event type $\hat{e}$ iff there are concrete states s_1 and s_2 such that $s_1 \in \hat{P}_1, s_2 \in \hat{P}_2$, and there is a transition from s_1 to s_2 on some event instance e, corresponding to $\hat{e}$. Note that $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$, like the concrete FSM $\mathcal{F}_L$, accepts all of the linearized event instance sequences. We now formally define $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$.

Definition 12 (Abstract FSM (AFSM))**.** For a log L, let $\mathcal{F}_L = \langle S, s_I, E, \Delta, S_T\rangle$ be the concrete FSM for L, and let $\hat{\mathbf{P}}$ be a partitioning of S. Then, an *abstraction of* $\mathcal{F}_L$, or an *abstract FSM* of L, is an FSM $\hat{\mathcal{A}}_L(\hat{\mathbf{P}}) = \langle \hat{\mathbf{P}}, \hat{\mathbf{P}}_I, \hat{E}, \hat{\Delta}, \hat{\mathbf{P}}_T\rangle$, where

- Initial states in $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$ are those partitions that contain the initial concrete state: $\hat{\mathbf{P}}_I = \{\hat{P} \in \hat{\mathbf{P}} \mid s_I \in \hat{P}\}$.
- Transitions in $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$ are event types that correspond to event instances in the concrete FSM: $\hat{E} = \{\hat{e} \mid \exists e, e \in E\}$.
- A transition between two partitions in $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$ exists if and only if there is a corresponding concrete transition between some two concrete states in the two partitions: $\hat{\Delta}(\hat{P}, \hat{e}) = \hat{P}' \iff \exists q \in \hat{P}, q' \in \hat{P}', e \in E, \Delta(q, e) = q'$.
- Terminal states in $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$ are those partitions that contain a terminal concrete state: $\hat{\mathbf{P}}_T = \{\hat{P} \in \hat{\mathbf{P}} \mid S_T \cap \hat{P} \neq \emptyset\}$.

An important feature of an AFSM is that it generalizes observed system states. A partition contains a finite number of observed system states, but through loops with transitions that modify channel state, an AFSM can generate arbitrarily long channel contents, leading to an arbitrarily large number of system states. We may not have observed these system states, but an AFSM model generalizes to predict that they are feasible.

CSight uses a first-k-in-channels partitioning strategy for generating an initial AFSM for a concrete FSM. This partitioning assigns two system states to the same partition if and only if the first-k message sequences in the channel states of the two states are identical. For example, suppose a system has two channels, c_{12} and c_{21}, and there are three concrete states: s_1, s_2, and s_3. Let $s.channels$ denote the channel contents for state s and suppose that $s_1.channels = \{c_{12} : [], c_{21} : [m]\}$, $s_2.channels = \{c_{12} : [], c_{21} : [m, m]\}$, and $s_3.channels = \{c_{12} : [l], c_{21} : [m]\}$. Then the first-1 contents of s_1 and s_2 are $\{c_{12} : [], c_{21} : [m]\}$ (the second m in $s_2.c_{21}$ is

[3]We use ˆ for $\hat{\mathbf{P}}$ even though it is a partitioning of states in the *concrete* FSM because the partitions in this set represent states in the *abstract* FSM. This set links the concrete and abstract FSMs.

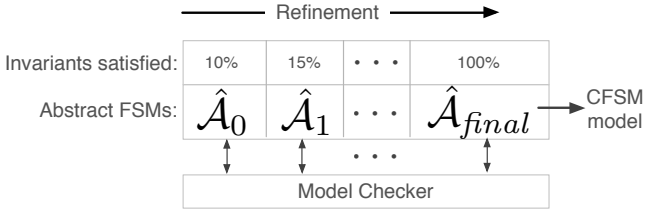


Figure 4: A high-level summary of the CSight refinement process (steps (5) and (6) in Figure 2). CSight starts with an initial abstract FSM, $\hat{\mathcal{A}}_0$, which is refined to a final abstract FSM, $\hat{\mathcal{A}}_{final}$. This final abstract FSM satisfies all of the mined invariants, and is converted into a CFSM that is returned as output.

not included), while the first-1 contents of s_3 are $\{c_{12} : [l], c_{21} : [m]\}$. Therefore, in a first-1 partitioning strategy, s_1 and s_2 would map to the same partition, while s_3 would map to a different partition.

Definition 13 (First-k partitioning)**.** Let S be a set of system states. $\hat{\mathbf{P}}_k$ is a first-k partitioning of S if $\forall \hat{P} \in \hat{\mathbf{P}}_k$, $\langle q, w\rangle, \langle q', w'\rangle \in \hat{P} \iff \forall c_{ij}, \forall g \in [1,k], w_{ij}[g] = w'_{ij}[g]$.

Finally, a CFSM is a set of per-process FSMs that communicate over FIFO channels. Channels are reliable, unidirectional, and have a single sender and receiver process. The alphabet of a process FSM includes process-local events, and inter-process communication (message *send* and message *receive*) events. We use the notation $c!m$ for a *send* event of message m on channel c, and the notation $c?m$ for a *receive* event of message m on channel c. Each channel has its own set of valid messages.

Definition 14 (Communicating FSM (CFSM))**.** A CFSM of h processes is a tuple of h process FSMs, $\langle F_1, F_2, \ldots, F_h\rangle$, and a set message sets $\mathcal{M}$ to indicate the message types that can be sent by one process and received at another. For each process i, its FSM is $\hat{F}_i = \langle \hat{Q}_i, \hat{I}_i, \hat{E}_i, \hat{\Delta}_i, \hat{T}_i\rangle$ and the message set M_i contains message types that can be sent from process i and received by some other process j, for all j. That is, $M_i = \cup_j M_{ij}$, where $c_{ij}?m \in \hat{E}_j \iff \exists i, m \in M_{ij}$ and $c_{ij}!m \in \hat{E}_i \iff \exists j, m \in M_{ij}$.

3.5 Model-Checking and Refining an AFSM

Figure 4 overviews, at a high level, the CSight model-checking and refinement process and Figure 5 lists an outline of the complete CSight algorithm. CSight uses the McScM [33] model checker to check if an invariant holds in the AFSM (step (5) in Figure 2). As McScM model-checks CFSMs and not AFSMs, to use McScM, CSight converts an AFSM into a CFSM. Further, as McScM reasons about state (un-)reachability, CSight encodes a temporal invariant in terms of states that can only be reached if the sequence of executed events violates the invariant. Both the conversion from an AFSM to a CFSM and the encoding are described in [8].

Model-checking an invariant produces one of three cases:
(1) The invariant holds in the model. There is nothing more for CSight to do for this invariant.
(2) The invariant does not hold and McScM finds and reports a counter-example CFSM execution. A *CFSM execution* is a sequence of events that abides by CFSM semantics (e.g., a process can only receive a message if that message is at the head of the channel). A *counter-example* CFSM execution is a sequence of events that violates the invariant. In this case, CSight refines the AFSM to eliminate the counter-example (Section 3.5.1) and then re-model-checks this invariant in case another counter-example for it exists.
(3) McScM fails to terminate within a user-defined threshold (defaulted to 5 minutes). CSight then attempts to check a different

```
1  function CSight(Log L, k):
2    let F_L = concrete FSM for L
3    let Â = AFSM for F_L with first-k partitioning P̂_k
4    let Ĉ_Â = AFSMtoCFSM(Â)
5    let Invs = MineInvariants(L)
6    foreach Inv ∈ Invs:
7      while (Ĉ_Â violates Inv):        // Call to model checker.
8        let p̂ = counter-example path for Inv in Ĉ_Â
9        // Translate events path p̂ in Ĉ_Â into Ŝ, a list of
10       // sets of paths in Â, |Ŝ| = h
11       let Ŝ = CFSMPathToAFSMPath(p, Ĉ_Â)
12       Â = Refine(Â, Ŝ)
13       Ĉ_Â = AFSMtoCFSM(Â)
14   return Ĉ_Â
```

Figure 5: An outline of the CSight algorithm. The `AFSMtoCFSM` and `CFSMPathToAFSMPath` procedures are detailed in [8], `MineInvariants` is described in [9]. The CSight implementation also handles model checker timeouts.

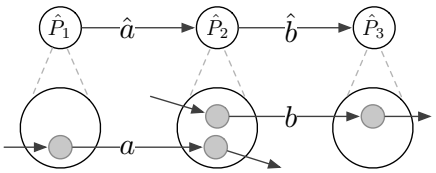


Figure 6: An AFSM path, $[\hat{a}, \hat{b}]$, that can be eliminated by refining (splitting) the abstract state $\hat{P}_2$, separating the two concrete states that generate the abstract path.

invariant first before coming back to the invariant that timed out.[4]

If CSight resolves all invariants, it outputs the model. But, if McScM repeatedly times out on every invariant, then CSight terminates and does not output a model. We did not observe the latter case in our experiments.

3.5.1 *Refining an AFSM to Satisfy Invariants*

CSight uses counter-example guided abstraction refinement (CEGAR) [16] to eliminate a counter-example for an invariant (step (6) in Figure 2). Figure 6 illustrates how the concrete states from the log may generate a counter-example partitions path in the AFSM.

The McScM-generated invariant counter-example is a CFSM execution. Refining CFSM states to eliminate a counter-example is challenging because a single state in the concrete FSM can map to multiple states in the CFSM. Instead, CSight performs partition refinement in the AFSM. It does this by mapping the McScM-generated *CFSM counter-example* into an *AFSM counter-example* [8].

Note that the AFSM counter-example is a list of *sets* of AFSM paths, one set of AFSM paths for each process in the system. This is because the process-specific event subsequence of a CFSM execution maps to potentially multiple paths in the AFSM.

Once the AFSM counter-example is generated, CSight eliminates the CFSM counter-example by transforming the AFSM into a more concrete (or less abstract) AFSM. It does so using partition refinement (`Refine` in Figure 7). Given an AFSM counter-example, `Refine` identifies the set of partitions that stitch concrete observations, as in partition $\hat{P}_2$ in Figure 6. It then refines all partitions in a set that is smallest across all processes and returns the refined AFSM.

[4] After refining the model to satisfy one invariant, a different, previously difficult-to-check invariant may become trivial to model-check.

```
1  function Refine(AFSM Â, AFSM Event Paths Ŝ):
2    let ⟨P̂, P̂_I, Ê, Δ̂, P̂_T⟩ = Â
3    // min is an index into Ŝ, denoting a set of process paths
4    // requiring the fewest number of refinements to eliminate.
5    let min = 0
6    foreach i ∈ [1,...,h]:
7      foreach ŝ ∈ Ŝ[i]:
8        let Π̂ = state sequence for ŝ in Â
9        // Find stitching partitions, e.g., P̂_2 in Fig. 6, by
10       // traversing Π̂ and recording partitions that can be
11       // refined to eliminate Π̂ from Â.
12       let Stitch_ŝ = {P̂ | P̂ is a stitching state in Π̂}
13       if Stitch_ŝ is empty:
14         next i
15     // Set of stitching partitions shared by those paths in Â
16     // that correspond to strings in Ŝ[i]
17     let Stitch_i = ∩_{ŝ∈Ŝ[i]} Stitch_ŝ
18     // min is an index that tracks the smallest Stitch_i
19     if min = 0 or |Stitch_i| < |Stitch_min|:
20       min = i
21   let P̂' = P̂ with all partitions in Stitch_min refined to
22            eliminate all paths in Ŝ[min]
23   // Derive P̂'_I, Δ̂', and P̂'_T from P̂' as in Def. 12.
24   let AFSM Â' = ⟨P̂', P̂'_I, Ê, Δ̂', P̂'_T⟩
25   return Â'
```

Figure 7: `Refine` removes an invariant counter-example from an AFSM. It refines one of the sets of process paths in $\hat{S}$, selecting the one that require the fewest refinements. Note that $\hat{S}$ is a list of sets of paths, one set per process.

A refined AFSM is more concrete — closer to the the concrete FSM. The refined AFSM has more partition states, and each partition state contains fewer concrete system states.

Definition 15 (AFSM Refinement)**.** An AFSM $\hat{\mathcal{A}}_L(\hat{\mathbf{Q}})$ is a *refinement* of AFSM $\hat{\mathcal{A}}_L(\hat{\mathbf{P}})$ if $\forall \hat{Q} \in \hat{\mathbf{Q}}, \exists \hat{P} \in \hat{\mathbf{P}}, \hat{Q} \subseteq \hat{P}$, and

$$\bigcup_{\hat{Q} \in \hat{\mathbf{Q}}} \hat{Q} = \bigcup_{\hat{P} \in \hat{\mathbf{P}}} \hat{P}$$

Next, in Section 4, we prove three key properties of the CSight process.

4. FORMAL ANALYSIS

We now use the formalisms defined in the previous section to prove three results about the CSight model inference process: (1) the inferred model fits the input log, (2) the inferred model satisfies mined invariants, and (3) refinement always makes progress.

We begin with an observation: The concrete FSM $\mathcal{F}_L$ satisfies all mined, validated invariants[5]. This is true by construction.

Observation 1 (Concrete FSM satisfies mined, validated invariants)**.** Let L be a log, and let *Invs* be the set of invariants that are valid in $\mathcal{F}_L$. Then, $\forall Inv \in Invs$, $s \in Lang(\mathcal{F}_L)$, s satisfies Inv.

[5]Invariant validation is described in Section 3.3.4 and further detailed in [8]

Theorem 1 (Inferred model fits the input log)**.** *For all logs L and integers k, `CSight`(L,k) returns a CFSM model that accepts all traces in L.*

Proof of Theorem 1. Let $\hat{\mathcal{C}}_{\hat{\mathcal{A}}}$ be the model returned by `CSight`(L,k). By construction, $\hat{\mathcal{A}}$ accepts all of the traces in $\hat{\mathcal{C}}_{\hat{\mathcal{A}}}$. Furthermore, $\hat{\mathcal{A}}$ is an abstract FSM for $\mathcal{F}_L$ — the concrete FSM of log L.

All abstract FSMs must at least accept the traces in the concrete FSM. Therefore, $\hat{\mathcal{A}}$ accepts all of the traces in $\mathcal{F}_L$. Since, $\mathcal{F}_L$ accepts all of the traces in L by construction, $\hat{\mathcal{A}}$ accepts the traces as well, and therefore $\hat{\mathcal{C}}_{\hat{\mathcal{A}}}$ must also accept them. □

A key property of the `Refine` procedure in Figure 7 is that it eliminates a counter-example path from a CFSM. We prove this next.

Theorem 2 (Refinement progress: Refinement eliminates counter-examples)**.** *Let $\hat{p}$ be a CFSM counter-example path for invariant Inv in $\hat{\mathcal{C}}_{\hat{\mathcal{A}}}$, let $\hat{S} =$ `CFSMPathToAFSMPath`$(\hat{p}, \hat{\mathcal{C}}_{\hat{\mathcal{A}}})$, and let $\hat{\mathcal{A}}' =$ `Refine`$(\hat{\mathcal{A}}, \hat{S})$. Then, $\hat{p}$ is not a counter-example to Inv in $\hat{\mathcal{C}}_{\hat{\mathcal{A}}'}$. That is, $\hat{p}$ is not a valid execution of $\hat{\mathcal{C}}_{\hat{\mathcal{A}}'}$.*

Proof of Theorem 2. Proof by contradiction. Assume that $\hat{p}$ is a sequence of events that is a valid execution of $\hat{\mathcal{C}}_{\hat{\mathcal{A}}'}$ and that $\hat{p}$ violates *Inv*. Consider an invocation of `Refine`$(\hat{\mathcal{A}}, \hat{S})$. `Refine` (Figure 7) uses a non-zero *min* value to compute $\hat{\mathbf{P}}'$ on line 19 and returns a CFSM $\hat{\mathcal{C}}_{\hat{\mathcal{A}}'} = \langle \hat{F}_i \rangle_{i=1}^{h}$.

For $\hat{p}$ to be a valid execution in $\hat{\mathcal{C}}_{\hat{\mathcal{A}}'}$, the sub-sequence of process i events $\hat{p}_i$, $\hat{p}_i = \pi(\hat{p}, \hat{E}_i)$, must be a valid execution in $\hat{F}_i$, for all i. The procedure `AFSMtoCFSM` (line 13 in Figure 5) constructs $\hat{F}_i$ to accept $\hat{p}_i$ iff there is a complete path $\hat{s}$ in $\hat{\mathcal{A}}'$, such that $\hat{p}_i = \pi(\hat{s}, \hat{E}_i)$. However, any such $\hat{s}$ must also be in $\widehat{Stitch}_{min}$, which is used to compute $\hat{\mathbf{P}}'$ in Figure 7. Therefore, after refining $\widehat{Stitch}_{min}$, $\hat{s}$ can no longer be a valid path in $\hat{\mathcal{A}}'$. Contradiction. □

Now, we prove that the `CSight` procedure in Figure 5 returns a CFSM model that satisfies all of the mined, validated invariants.

Theorem 3 (Inferred model satisfies mined invariants.)**.** *For a given log L, if `CSight` outputs a CFSM model then this model satisfies all of the mined, validated event invariants from $\mathcal{F}_L$.*

Proof of Theorem 3. For a log L with a total of n event instances, CSight can refine the initial abstract FSM for L, $\hat{\mathcal{A}}_L(\hat{\mathbf{P}}_k)$, at most $n-1$ times. This is because after $n-1$ refinements, each partition in the abstract FSM must map to exactly one concrete state, and a singleton partition cannot be refined further.

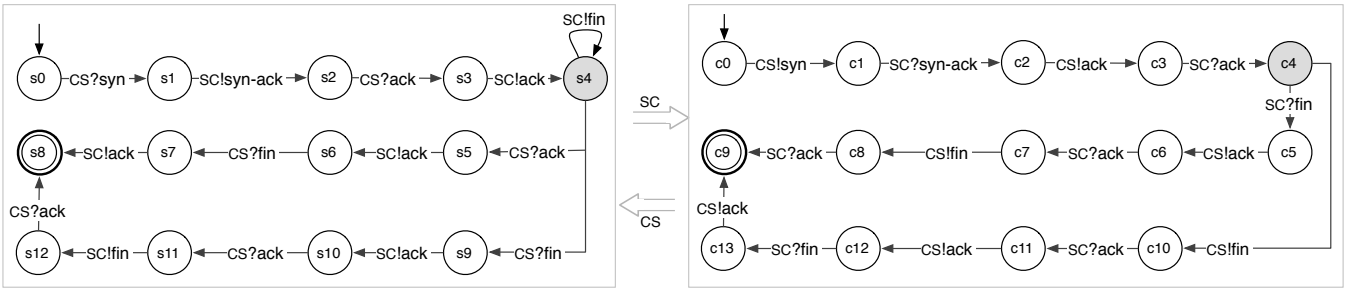
Let $\hat{\mathcal{A}}$ be the abstract FSM after $n-1$ refinements of $\hat{\mathcal{A}}_L(\hat{\mathbf{P}}_k)$. Because $\hat{\mathcal{A}}$ maps each event instance to a unique partition, it is indistinguishable from $\mathcal{F}_L$, the concrete FSM it abstracts. Therefore, $Lang(\hat{\mathcal{A}}) = Lang(\mathcal{F}_L)$. By Observation 1, $\mathcal{F}_L$ satisfies all validated invariants, therefore so does $\hat{\mathcal{A}}$.

Since CSight does not terminate until all the validated invariants are satisfied in the abstract FSM, it either returns $\hat{\mathcal{A}}$ after $n-1$ refinements, or it returns a smaller (and more abstract) $\hat{\mathcal{A}}'$. In both cases, the returned AFSM satisfies all of the validated invariants. □

5. LOGGING VECTOR TIMESTAMPS

CSight requires its input logs to be annotated with vector timestamps. Vector time is a logical clock mechanism to partially order events in a concurrent system [25, 45]. To ease using CSight with concurrent systems, we have built ShiVector[6], a tool to automatically

[6]https://bitbucket.org/bestchai/shivector/



(a) TCP server **(b) TCP client**

Figure 8: The inferred state machines for the (a) TCP server and (b) TCP client. The server communicates with the client over the `SC` channel, and the client communicates with the server over the `CS` channel. Shaded states represent the `connection-established` states.

compute and insert vector timestamps into Java system logs. ShiVector does not require modification of the system's source code, only recompilation with the ShiVector library added to the classpath.

ShiVector uses aspects (via AspectJ) to intercept socket-based network communication to piggy-back process vector timestamps on top of existing application-level protocols.[7] ShiVector also intercepts logging events to augment each logged event with a vector timestamp at the time that the event was logged. Thus, ShiVector automatically tracks and injects vector timestamps into concurrent systems' logs.

6. EXPERIMENTAL EVALUATION

The CSight prototype uses McScM for model checking and graphviz for model visualization. In all experiments we used a value of $k = 1$ for the first-k-in-channels partitioning strategy. We evaluated CSight on three sets of logs, produced by: a simulator of the stop-and-wait protocol; the TCP stack of OS X; and Voldemort [56], an open-source project that implements a distributed hash table [22] and is used in data centers at companies like LinkedIn. We also carried out a user study with a group of undergraduate students to evaluate the efficacy of CFSM models in finding bugs.

6.1 Stop-and-Wait Protocol

We applied CSight to traces from a simulator of the stop-and-wait protocol described in Section 2. We derived a diverse set of traces by varying message delays to produce different message interleavings. CSight mined a total of 66 valid invariants. The model CSight derived (Figure 1(b)) is identical to the true model of the stop-and-wait protocol. This experiment was a sanity check to verify that CSight performed as expected on this well-understood protocol, when faced with concurrency-induced non-determinism in interleavings.

6.2 TCP

The TCP protocol uses a three-phase opening handshake to establish a bi-directional communication channel between two end-points. It tears down and cleans up the connection using a four-phase closing handshake. The TCP state machine is complicated by the fact that packet delays and packet losses cause the end-points to timeout and re-transmit certain packets, which may in turn induce new messages. Our goal was to model common-case TCP behavior, so we did not explore these protocol corner cases.

We used netcat and dummynet [51] to generate TCP packet flow. We captured packets using tcpdump and then semi-manually annotated the log to include vector timestamps. A subset of the traces that involve only the the opening and closing TCP handshakes were fed into CSight.

[7]We plan to extend ShiVector to interpose on other kinds of inter-process communication.

For the captured TCP log, CSight identified 149 valid invariants, some of which are not true of the complete protocol (e.g., because the input traces did not contain certain packet retransmissions). The CSight-derived CFSM model is shown in Figure 8. The shaded states `s4` and `c4` represent the server and client `connection-established` states, which is reached when the two end-points have successfully set up the bi-directional channel. Transitions up to these two states model the opening handshake, while transitions after these states model the closing handshake. The closing handshake is split into a server-initiated tear-down sequence (middle row of states) and a client-initiated tear-down (bottom-most row of states).

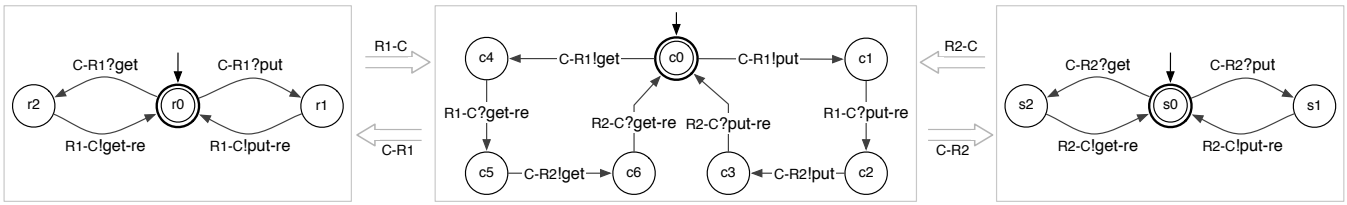
The derived model is accurate except for the self-loop on state `s4` in Figure 8(a). This loop appears because `s4` represents both the `connection-established` state and the state after the server has initiated the closing handshake. This loop appears to contradict the $\texttt{SC!fin} \not\rightarrow \texttt{SC!fin}$ invariant, which *is* mined by CSight and is valid. However, the model checker only considers counter-examples that terminate. Note that if the loop at `s4` is traversed twice then the client will be unable to consume both server `fin` packet copies and will enter an unspecified reception error state and therefore not terminate. Such unspecified reception states are typically undesirable, as they can be confusing. The McScM model checker can be used to detect these states and further refinement will eliminate them. Implementing this elimination remains as part of future work.

Figure 8 also illustrates a key feature of CSight — the user decides (by specifying a set of line-matching regular expressions) what information in the log is relevant to the modeling task. Thus, for each use of CSight, the user decides on the trade-off between comprehensibility of the model (e.g., model size) and the amount of information lost/retained by the modeling process. For example, the TCP model in Figure 8 is simple to understand, but it omits TCP sequence numbers, data packets, and other details that were present in the input log.

6.3 Voldemort Distributed Hash Table

Voldemort [56] implements a distributed hash table with a client API that has two main methods. `put(k,v)` associates the value `v` with the key `k`, and `get(k)` retrieves the current value associated with the key `k`. Voldemort is a *distributed system* as it provides scalability by partitioning the key space across multiple machines and achieves fault tolerance by replicating keys and values across multiple machines.

The Voldemort project has an extensive test suite. We used a subset of integration tests that exercise the synchronous replication protocol to generate a log of replication messages in a system with one client and two replicas. We logged messages generated by client calls to the synchronized versions of `put` and `get` and captured just



(a) Voldemort Replica 1 **(b) Voldemort client** **(c) Voldemort Replica 2**

Figure 9: (a,c) replica models and (b) client model for a three-node Voldemort cluster. "re" stands for "response".

the messages between the client and the two replicas.[8] Since Voldemort does not implement vector timestamps, we used ShiVector (described in Section 5) to produce a vector timestamped log.

CSight mined 112 valid invariants and generated the model in Figure 9. This model contains a client FSM and two replica FSMs. As expected, the replica FSMs are identical. Synchronized Voldemort operations are serialized in a specific order, so the flow of messages for `put` as well as for `get` is identical — the client first executes the operation at replica 1 and then at replica 2.

We manually inspected Voldemort's replication code to confirm that the model in Figure 9 is accurate. This model provides a high-level overview of how replication messages flow in the system. However, as with the TCP protocol, the model also abstract away and omits numerous details, such as what happens when replicas fail. One feature of CSight is that it allows a developer to focus on those aspects of system behavior that are important to them. For example, by not exercising irrelevant system behavior in the first place, or by running CSight with regular expressions that ignore certain logged behavior in the log parsing stage of the process.

6.4 User Study

To determine whether CFSM models are useful in finding bugs, we designed and ran a user study. It is typical for developers to interpret distributed executions in the form of time-space diagrams. Therefore, we compared the efficacy of CFSM models against time-space diagrams in bug-finding tasks. Throughout the study and in the oral feedback session that followed, participants were not told the purpose of the study.

The study focused on two concurrent systems: the stop-and-wait protocol and Voldemort. We introduced a bug into each system — the sender in the stop-and-wait protocol failed to re-transmit packets on timeout, and the synchronized Voldemort client sent requests to all replicas concurrently, instead of blocking on acknowledgment from each replica. In our experience, both bugs are representative of real bugs faced by distributed system developers. The root cause of the stop-and-wait bug is not taking the right action on an event; the root cause of the Voldemort bug is performing an action at the wrong time. For each buggy system, we generated: (1) a set of representative time-space execution diagrams (8 for the stop-and-wait protocol and 6 for Voldemort), and (2) a CFSM model. Overall, we created four artifacts for the study — *tspace-saw* and *cfsm-saw* (time-space diagrams and CFSM model of the buggy stop-and-wait protocol), and *tspace-vol* and *cfsm-vol* (artifacts for buggy Voldemort).

The study consisted of an in-class, web-based assignment[9] in an undergraduate Introduction to Software Engineering class at the University of Massachusetts, Amherst. The 39 students who completed the assignment had, on average, 4.2 years of programming experience, and 76% of them had never taken a networks course.

[8]The replicas also communicate with each other to maintain key availability, but we excluded this communication from the log.

[9]Available online: http://www.bestchai.net/papers/icse2014_csight_eval/

We considered two factors: the model factor (time-space diagrams vs. CFSM models), and the task factor (stop-and-wait vs. Voldemort). To account for learning effects, we used a within-participants mixed design across all 39 participants. We randomly assigned each student to one of four possible study sequences: ⟨*tspace-saw*, *cfsm-vol*⟩, ⟨*tspace-vol*, *cfsm-saw*⟩, ⟨*cfsm-saw*, *tspace-vol*⟩, ⟨*cfsm-vol*, *tspace-saw*⟩. We considered the two bugs to be independent — finding one bug in one artifact does not help in finding the other bug in a different artifact.

Each task consisted of two steps:

1. Each student completed a mini-tutorial on the appropriate diagram (time-space and CFSM, respectively). To verify their understanding, at the end of each tutorial, each student had to answer two basic questions correctly. They could resubmit their answers until both were correct.[10]

2. Each student was given a *correct*, written description of the system in English, along with either a set of time-space diagrams or a CFSM model. The student was asked to respond to a single open-ended question. For time-space diagrams (*tspace-saw*, *tspace-vol*) we asked: "List all of the time-space diagrams above that you think do not conform to the description of the system above. What made you choose these diagrams?" For the CFSM models (*cfsm-saw*, *cfsm-vol*) we asked: "How does the observed model differ from the intended system description?"

6.4.1 Results

Students found bugs approximately as well with the CFSM model as they did with the time-space diagrams that are developers' current preferred visualization. For the stop-and-wait protocol, students who were shown the CFSM were 72% successful in getting the right answer, while those who were shown the 8 time-space diagrams were 61% successful. For Voldemort, the success rate with the CFSM was 72% and the success rate with the 6 time-space diagrams was 86%.

Students found CFSMs just as useful (for completing the task) as the small collection of time-space diagrams. These results indicate that CFSM models can be used to effectively find bugs. Moreover, in practice, developers have to inspect neither 8 nor 6 executions of the system, but hundreds or thousands. The task of manually finding the anomalous time-space diagrams would be infeasible. The CFSM models, though, will remain roughly of the same size and complexity regardless of the number of executions. Therefore, we believe that our results on the utility of CFSMs over time-space diagrams are conservative, and CFSMs would perform even better in practice. Our study finds that they already perform as well as time-space diagrams tightly focused on the buggy behavior.

Students' oral feedback on the assignment reveals why many of them preferred the CFSM model. According to many students,

[10]The average time to complete either tutorial was 5 minutes. Students made, on average, 1.8 attempts to answer the tutorial questions correctly. These measures were similar when the time-space and the CFSM tutorials were considered separately.

time-space diagrams were difficult to follow, especially for long executions:

> "I found the time-space diagrams confusing. It was hard to tell what was happening when. The [CFSM] models were simpler and made it clear what state a system was in and I could keep track of that state." – student 1

> "Time-space diagrams were easy to follow for small time segments. For longer time segments, you got lost. The other [CFSM] models were better for longer time." – student 2

Students could compare a CFSM model to their own mental model, or the model they would draw after reading the system description:

> "I read the description and tried to recreate the [CFSM] model myself first. Then, I compared what I drew to the given diagram and found mistakes. Understanding those mistakes helped me understand the system a lot better." – student 3

Finally, students mentioned that CFSM models were easier to use because they did not explicitly model time:

> "For the [CFSM] models, I assumed no time delay in the network messages. It was harder to do in the time-space diagrams because you cannot ignore the delay there. In CFSMs, you can ignore time at first, and then allow for it." – student 4

7. RELATED WORK

CFSMs can be inferred from manually-labeled message sequence charts [13]. In contrast, CSight automates the inference by relying on mined invariants. CFSMs inferred from executions can demonstrate system properties, such as absence of deadlocks and unspecified receptions [15, 55]. We found that CFSMs also provide a concise representation of complex concurrent system logs. Another potentially useful inferred model of concurrent systems is *class*-level specifications in the form of symbolic message sequence charts [38]. CSight can use these to merge identical process FSMs, such as the replica models in Figure 9(a,c).

In addition to inferring models, concurrent system logs can be used to detect anomalies [35, 44, 57], identify performance bugs [52, 53], and mine temporal system properties [9, 17, 59]. Our focus is on concurrency and on extracting a model that can aid understanding of more general system behavior. Our own work on mining temporal invariants of concurrent systems [9] is complementary to CSight, which needs to mine such invariants from execution logs. Probabilistic analysis of logs (measuring number, frequency, and regularity of event occurrences) can help discover concurrent system execution patterns, which can help the developer better understand and improve the design of concurrent systems [17]. Logs can also be used to mine message sequence charts [37], which describe the process interactions within a concurrent system.

The problem of finding a short sequence of refinements to produce a small abstract FSM that satisfies all of the valid invariants is NP-hard [16], and CSight's design finds an approximate solution. For logs of serial systems (totally ordered logs), the problem of automata inference from positive examples of executions is computable [12] but NP-complete [30, 5], and the FSA cannot be approximated by any polynomial-time algorithm [48]. Unlike CSight, prior work on model inference from totally ordered logs either excluded concurrency or captured a particular interleaving of concurrent events [2, 10, 7, 11, 18, 28, 43, 46, 49]. Tomte [1] and Synoptic [10] take a similar approach to CSight, using CEGAR [16] to refine models. Synoptic mines temporal invariants and infers FSM models from a log of sequential executions, while Tomte infers scalarset Mealy machines. CSight mirrors Synoptic's inference procedure but uses a different modeling formalism and algorithms, and works for concurrent systems that log concurrency as a partial order. We believe that modeling concurrency explicitly is crucial to understanding a concurrent system's behavior.

System models can also be inferred from developer-written specifications [4, 19, 21, 29, 31, 36]. Compared to CSight, these techniques require significantly more manual effort from the developer and are not suitable for legacy systems.

CSight relies on the McScM model checker [33, 32], which represents a scalability bottleneck. Future work will use the more efficient Spin model checker [34].

Other distributed system debugging approaches focus on data (e.g., [3, 6, 26]), as opposed to event execution sequences. Runtime monitoring approaches can detect execution sequence or data property violations for error reporting [50, 27, 41, 20] or self-adaptation [40, 23, 58, 47]. CSight models can be used at runtime to verify that executions conform to the expected model.

8. DISCUSSION

The vector timestamp tracking and logging by ShiVector (Section 5) may affect system performance. However, as is typical for production systems, ShiVector can be enabled for a small fraction of the requests. If the collected traces are representative of typical system behavior, then CSight's model, too, will be representative.

CSight's model construction cannot directly eliminate two kinds of error states [14] — unspecified reception and deadlock. Unspecified reception occurs when a process enters a state with a message m at the head of its channel, but has no reachable future state that receives m. A deadlock occurs when no process can send a message and at least one process cannot reach a terminating state. CSight does not check if these error states are reachable in the final model. CSight can be extended with such a check (e.g., by using the McScM model checker), but the check would be computationally expensive.

CSight mines three types of invariants and ensures that the final model satisfies the valid invariants. For the invariants to be accurate, the log must have executions representative of all possible executions of the modeled system. While we found these invariant types to be sufficient to infer interesting models in practice, more extensive invariants can lead to more expressive models.

CSight works for system traces that satisfy certain communication constraints (Def. 1). For example, CSight cannot model unclean termination and assumes that each execution terminates with empty channels. However, in practice, system logs may not satisfy these constraints. In our future work we will extend CSight to handle terminal states with non-empty channels.

9. CONCLUSION

Concurrent systems are hard to implement, debug, and verify. To help with these tasks, we developed CSight, a tool that uses a partially ordered log of events to infer a concise and accurate communicating finite state machine model of the system. CSight's accuracy comes from its use of mined temporal properties that relate events in the log. Our tool prototypes are available for download at http://synoptic.googlecode.com/. By automatically mining a system model from logs, CSight has the potential to ease system understanding, debugging, and maintenance tasks.

10. ACKNOWLEDGMENTS

We thank Jenny Abrahamson for her work on ShiVector and Roykrong Sukkerd for her contributions to the project. This material is based upon work supported by the United States Air Force under Contract No. FA8750-12-C-0174 and by IARPA under Contract No. N66001-13-1-2006.

11. REFERENCES

[1] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *the International Symposium on Formal Methods (FM)*, Paris, France, 2012.

[2] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *the Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Dubrovnik, Croatia, 2007.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. *Operating Systems Review (OSR)*, 37(5):74–89, October 2003.

[4] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastin Uchitel. Learning Operational Requirements from Goal Models. In *the International Conference on Software Engineering (ICSE)*, Vancouver, BC, Canada, 2009.

[5] Dana Angluin. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences*, 21(1):46 – 62, 1980.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, 2004.

[7] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Unifying FSM-Inference Algorithms through Declarative Specification. In *the International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, 2013.

[8] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring Models of Networked Systems from Logs of their Behavior with CSight. Technical report, http://hdl.handle.net/2429/46122, Univ. of British Columbia, 2014.

[9] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. Mining Temporal Invariants from Partially Ordered Logs. *Operating Systems Review (OSR)*, 45(3):39–46, December 2011.

[10] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *the Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, 2011.

[11] Alan W. Biermann and Jerome A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers (TC)*, 21(6):592–597, June 1972.

[12] Lenore Blum and Manuel Blum. Toward a Mathematical Theory of Inductive Inference. *Information and Control*, 28(2):125–155, 1975.

[13] Benedikt Bollig, Joost P. Katoen, Carsten Kern, and Martin Leucker. Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering (TSE)*, 36(3):390–408, May 2010.

[14] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[15] Xiao J. Chen and Hasan Ural. Construction of Deadlock-free Designs of Communication Protocols from Observations. *The Computer Journal*, 45(2):162–173, January 2002.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[17] Jonathan E. Cook, Zhidian Du, Chongbing Liu, and Alexander L. Wolf. Discovering models of behavior for concurrent workflows. *Computers in Industry*, 53(3):297–319, 2004.

[18] Jonathan E. Cook and Alexander L. Wolf. Discovering Models of Software Processes from Event-Based Data. *Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.

[19] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis. In *the International Symposium on the Foundations of Software Engineering (FSE)*, Portland, OR, USA, 2006.

[20] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. Live Debugging of Distributed Systems. In *the International Conference on Compiler Construction (CC)*, Boston, MA, USA, 2009.

[21] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Validation of Contracts Using Enabledness Preserving Finite State Abstractions. In *the International Conference on Software Engineering (ICSE)*, Vancouver, BC, Canada, 2009.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *the Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, USA, 2007.

[23] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *the International Symposium on Software Testing and Analysis*, pages 233–243, Portland, ME, USA, July 2006.

[24] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *the International Conference on Software Engineering (ICSE)*, Los Angeles, CA, USA, 1999.

[25] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *the Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[26] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007.

[27] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007.

[28] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining Behavior Models from User-intensive Web Applications. In *the International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014.

[29] Dimitra Giannakopoulou and Jeff Magee. Fluent Model Checking for Event-Based Systems. In *the International Symposium on the Foundations of Software Engineering (FSE)*, Helsinki, Finland, 2003.

[30] E. Mark Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[31] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. *Formal Methods in Software and Systems Modeling*, 3393, 2005.

[32] Alexander Heussner, Tristan Gall, and Grégoire Sutre. Extrapolation-Based Path Invariants for Abstraction Refinement of Fifo Systems. In *the International SPIN Workshop on Model Checking of Software*, Grenoble, France, 2009.

[33] Alexander Heussner, Tristan Le Gall, and Grégoire Sutre. McScM: A General Framework for the Verification of Communicating Machines. In *the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Tallinn, Estonia, 2012.

[34] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, May 1997.

[35] Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In *the International Conference on Autonomic Computing (ICAC)*, Seattle, WA, USA, 2005.

[36] Ivo Krka, Yuriy Brun, George Edwards, and Nenad Medvidovic. Synthesizing Partial Component-Level Behavior Models from System Specifications. In *the International Symposium on the Foundations of Software Engineering (FSE)*, Amsterdam, The Netherlands, 2009.

[37] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Mining Message Sequence Graphs. In *the International Conference on Software Engineering (ICSE)*, pages 91–100, Honolulu, HI, USA, 2011.

[38] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Inferring Class Level Specifications for Distributed Systems. In *the International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, 2012.

[39] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[40] Lee Lin and Michael D. Ernst. Improving the Adaptability of Multi-mode Systems via Program Steering. In *the International Symposium on Software Testing and Analysis*, pages 206–216, Boston, MA, USA, 2004.

[41] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, 2008.

[42] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic Steering of Behavioral Model Inference. In *the Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Amsterdam, The Netherlands, 2009.

[43] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *the International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008.

[44] Jian G. Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining Invariants from Console Logs for System Problem Detection. In *the USENIX Annual Technical Conference (ATC)*, Boston, MA, USA, 2010.

[45] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Gers, France, 1989.

[46] Tony Ohmann, Kevin Thai, Ivan Beschastnikh, and Yuriy Brun. Mining Precise Performance-Aware Behavioral Models from Existing Instrumentation. In *the International Conference on Software Engineering New Ideas and Emerging Results (ICSE NIER) track*, Hyderabad, India, 2014.

[47] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *the Symposium on Operating Systems Principles (SOSP)*, pages 87–102, Big Sky, MT, USA, 2009.

[48] Leonard Pitt and Manfred K. Warmuth. The Minimum Consistent DFA Problem Cannot be Approximated Within any Polynomial. *Journal of the ACM*, 40(1):95–142, 1993.

[49] Steven P. Reiss and Manos Renieris. Encoding Program Executions. In *the International Conference on Software Engineering (ICSE)*, Toronto, ON, Canada, 2001.

[50] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Networked Systems Design and Implementation (NSDI)*, San Jose, CA, USA, 2006.

[51] Luigi Rizzo. Dummynet: a Simple Approach to the Evaluation of Network Protocols. *Computer Communication Review (CCR)*, 27(1):31–41, January 1997.

[52] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Fows. In *Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, 2011.

[53] Jaspal Subhlok and Qiang Xu. Automatic Construction of Coordinated Performance Skeletons. In *the International Parallel & Distributed Processing Symposium (IPDPS)*, Miami, FL, USA, 2008.

[54] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall, 5th edition, 2010.

[55] Hasan Ural and Hüsnü Yenigün. *Towards Design Recovery from Observations*, volume 3235, chapter 9, pages 133–149. Springer Berlin Heidelberg, 2004.

[56] Voldemort. http://project-voldemort.com, 2013.

[57] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *the Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, 2009.

[58] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: predicting and preventing inconsistencies in deployed distributed systems. In *Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, 2009.

[59] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *the International Conference on Software Engineering (ICSE)*, Shanghai, China, 2006.