Title

An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes

Authors

Shah, Amar
Mora, Federico
Seshia, Sanjit A

Publication Date

2024

Peer reviewed

Message Chains for Distributed System Verification

FEDERICO MORA, University of California, Berkeley, USA
ANKUSH DESAI, Amazon Web Services, USA
ELIZABETH POLGREEN, University of Edinburgh, UK
SANJIT A. SESHIA, University of California, Berkeley, USA

Verification of asynchronous distributed programs is challenging due to the need to reason about numerous control paths resulting from the myriad interleaving of messages and failures. In this paper, we propose an automated bookkeeping method based on *message chains*. Message chains reveal structure in asynchronous distributed system executions and can help programmers verify their systems at the message passing level of abstraction. To evaluate our contributions empirically we build a verification prototype for the P programming language that integrates message chains. We use it to verify 16 benchmarks from related work, one new benchmark that exemplifies the kinds of systems our method focuses on, and two industrial benchmarks. We find that message chains are able to simplify existing proofs and our prototype performs comparably to existing work in terms of runtime. We extend our work with support for specification mining and find that message chains provide enough structure to allow existing learning and program synthesis tools to automatically infer meaningful specifications using only execution examples.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; • **Theory of computation** → **Logic and verification**.

Additional Key Words and Phrases: Formal verification, distributed systems, message passing

ACM Reference Format:
Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. 2023. Message Chains for Distributed System Verification. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 300 (October 2023), 27 pages. https://doi.org/10.1145/3622876

1 INTRODUCTION

Programming error-free, reliable distributed systems is hard. Engineers need to reason about numerous control branches stemming from a myriad interleaving of messages, the systems are extremely tricky to debug due to the nested mesh of messages and failures, and it is surprisingly easy to introduce subtle errors while improvising to fill in gaps between high-level system descriptions and their concrete implementations. Yet we are dependent on distributed systems to deliver high-performance computing, fault-tolerant networked services, and global-scale cloud infrastructures. So what can we do to ease the job of the engineer tasked with programming a distributed system? In this paper, we turn to a combination of formal modeling and automated reasoning. Specifically, we propose a new set of tools—based on a novel notion of *message chains*—that distributed system engineers can use to (automatically) reason about the fruits of their labor.

Authors' addresses: Federico Mora, fmora@berkeley.edu, University of California, Berkeley, USA; Ankush Desai, ankushpd@amazon.com, Amazon Web Services, USA; Elizabeth Polgreen, elizabeth.polgreen@ed.ac.uk, University of Edinburgh, UK; Sanjit A. Seshia, sseshia@berkeley.edu, University of California, Berkeley, USA.

2475-1421/2023/10-ART300
https://doi.org/10.1145/3622876

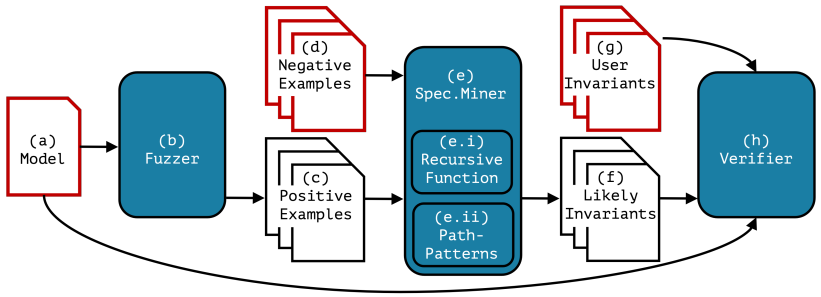


Fig. 1. Contributions overview and workflow. White rectangles are artefacts; blue rectangles are components of the framework. Red outline indicates artefacts are user inputs.

Modeling languages and automated reasoning already play an important role in raising the level of assurance in distributed systems and cloud services. Classic domain-specific examples of the combination include TLA+ [Lamport 2002]; newer examples include Ivy [McMillan and Padon 2020]. These languages and associated automated reasoning tools can be used to guarantee correctness for all executions of a given system, or at least systematically explore the space in search of errors. Unfortunately, these tools do not provide users with language primitives or abstractions to support explicit message passing. To work at the message passing level—where a significant source of implementation errors lie—therefore requires manual modelling and significant user effort.

To remedy this, we propose message chains, an automated method for keeping track of message chains, and a verification framework that uses message chains to give users the vocabulary to reason about the low-level details of explicit message passing while remaining at an intuitive level of abstraction. Informally, message chains are sequences of logically related messages. This idea is related to message sequence charts [ITU-T 2011] from the software engineering community, and choreographic programming languages [Montesi 2014] from the programming languages community. Message sequence charts are partial specifications that describe the flow of messages through distributed systems. Engineers use these charts for testing, design, simulation, and documentation. Choreographic programming languages, on the other hand, are high-level languages that define distributed systems from a global perspective, focusing on communication flows. Both message sequence charts and choreographic programming offer developers a way to reason about explicit message passing but with the context of how messages flow through the system in question. In this paper, we use message chains to bring these same ideas to formal verification.

Fig. 1 shows the complete workflow of our proposed approach. The user first inputs a model (1.a) of their system written in the version of P [Desai et al. 2013, 2018] described in Sec. 5. This model is fuzzed (1.b) to generate positive example message chains (1.c). We formally define message chains in Sec. 3 and we formally define positive (and negative) examples in Sec. 4. The user can optionally add negative examples (1.d) and then run the specification mining framework (1.e) described in Sec. 4, producing likely invariants (1.f). The user can use these likely invariants as their specification, provide their own invariants (1.g), or both. Either way, the model (1.a) and invariants (1.f and 1.g) are given to the verification engine (1.h) described in Sec. 3 which returns either true, if the proof by induction passes, or false otherwise. This process is usually iterative with the user changing their model, invariants, or adding examples manually or through fuzzing. Note that while a successful proof by induction implies there are no bugs, a failed proof by induction does not imply there are bugs. This makes our tool unsuitable for bug finding, like all other proof

```
1  data node
2  type mc := message_chain[node]
3  type sys := system[node]
4
5  function btw(w: node, x: node, y: node): Bool
6  function right(n: node): node
7  function le(x: node, y: node): Bool
8
9  event eNominate := {id: node}
10 machine Node {
11   state Search {
12     on entry do {
13       send right(this), eNominate(this)}
14     on eNominate e do {
15       let curr := e.payload.id in
16       if curr = this then goto Won
17       else if le(this, curr) then
18         send right(this), eNominate(curr)
19       else
20         send right(this), eNominate(this)}}
21   state Won {}}
```

(a) System model (example of Fig. 1.a)

```
22 function in_flight(s: sys, e: mc): Bool :=
23   s.events[e] and e is send
24 function leader(s: sys, l: node): Bool :=
25   s.machines[l].Node_state is Won
26 recursive function participated(e: mc, r: node): Bool :=
27   if e is empty then false
28   else e.source = r or participated(e.history, r)
29
30 induction (s: sys)
31   invariant unique_leader: forall (l: node, n: node)
32     leader(s, l) and leader(s, n) ==> l = n
33   invariant leader_max: forall (l: node, n: node)
34     leader(s, l) ==> le(n, l)
35   invariant participated_means_le_head:
36     forall (e: mc, n: node) let head := e.payload.id in
37       in_flight(s, e) and participated(e, n)
38         ==> le(n, head)
39   invariant not_participated_means_going_to_visit:
40     forall (e: mc, n: node) let head := e.payload.id in
41       in_flight(s, e) and not participated(e, n)
42         ==> btw(head, e.target, n) or e.target = n
```

(b) Specifications (example of Fig. 1.g or Fig. 1.f)

Fig. 2. Ring leader election in the UPVerifier. Definitions of `btw` ("between"), `right`, and `le` ("less than or equal to") functions omitted. `btw` is used to define the ring topology, `right` uses `btw` to determine the next node in the ring given an input node, and `le` is used to determine who the winner should be.

by induction-based tools. But our work can be easily extended to support bug finding algorithms, like bounded model checking.

1.1 Running Example: Ring Leader Election Protocol

To better understand this workflow, consider the following ring leader election protocol inspired by Le Lann [1977], Chang and Roberts [1979]. The protocol consists of a set of nodes, each with a unique label value. The goal of the protocol is for the nodes to collectively discover the node with the greatest label. Operationally, nodes are arranged in a ring: each node can only receive messages from the node on its "left" and can only send messages to the node on its "right." Nodes send messages holding label values and begin their execution by sending their own label to their right. If a node receives a label greater than its own label, it forwards that received label to its right. If the received label is smaller than its own label, the node sends its own label to the right. Finally, if a node ever receives its own label, it declares itself the winner.

We display an implementation of the ring protocol in P-like syntax in Fig. 2a. This code corresponds to user provided "Model" in Fig. 1.a. P programs consist primarily of event and machine declarations. Event declarations define the kind of messages machines exchange and the payloads these messages hold. Machine declarations define the kinds of machines in the system and how they interact with each other. In this figure, we define one kind of event, `eNominate`, which holds an identifier (line 9). We also define one kind of machine, `Node`, which has no fields, and two states: `Search` and `Won` (lines 10-21). When nodes enter the `Search` state, they send their value to the node on their right (lines 12-13). After sending their value, while nodes are still in the `Search` state, nodes react to receiving nominations in one of three ways (lines 14-20). First, when nodes receive a value less than their value, they send their value to the node on their right (lines 19-20). Second, when nodes receive a value greater than their value, they forward this new value to the node on

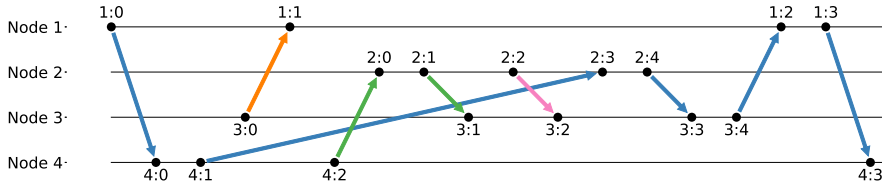


Fig. 3. Lamport diagram for an execution of Fig. 2a instantiated with four nodes. Nodes are arranged in a ring, ordered 1, 4, 2, 3. Events are labeled *n*:*i*, where *n* is the node the event occurs on and *i* is the event number on that node. Colors on arrows between send and receive events indicate message chains.

their right (lines 17-18). Third, when nodes receive their own value, they move to the Won state (line 16). The code blocks that begin with the keyword on are *handlers*.

Note that Fig. 2a makes no mention of the number of nodes in the system. In fact, this definition represents all possible instantiations of the protocol. We describe this in detail in Sec. 3. For now, it suffices to know that the node (lower-case "n") definition (line 1) declares a new uninterpreted sort that represents a pointer to, or label of, Node (upper-case "N") machines. The size of the universe of node determines the number of Node machines, and the verification engine that we define will be free to search for counterexamples over any possible universe of node.

1.2 Message Chains

Fig. 3 shows the Lamport diagram for an execution of the ring leader election protocol on four nodes (the universe of node is the set {1, 2, 3, 4}). Every event in Fig. 3 corresponds to the entry into a handler or the execution of a send instruction in Fig. 2a. In this example, node 4 is to the right of node 1, node 2 is to the right of node 4, node 3 is to the right of node 2, and node 1 is to the right of node 3. The execution begins with node 1 sending its own label value to node 4 (event 1:0, which corresponds to entering the `on entry` handler of Fig. 2a). When node 4 receives the value 1 (event 4:0), it compares 1 to its own value, and decides to send its own value to node 2 (event 4:1). This process goes on until node 4 receives its own value (event 4:3) and declares itself the winner. Fig. 3 also shows four message chains observed during the same execution. The first message chain, in blue, consists of the events 1:0, 4:0, 4:1, 2:3, 2:4, 3:3, 3:4, 1:2, 1:3, and 4:3, along with omitted associated payloads. The remaining message chains are in orange, green, and pink. The message chains in Fig. 3 correspond to the "Positive Examples" in Fig. 1.c: these can be obtained automatically by fuzzing the input model (Fig. 1.a).

Intuitively, message chains behave like email chains: each message in the system is passed around with the message it was responding to, creating a history consisting of a sequence of messages that caused the current message. Looking at this history is often useful. For example, if you receive a reply to an email, you know that sender received your email and you can tell which email they are responding to by looking at the chain. It does not matter whether that sender also sent other emails to you or other people, the order in which they sent these other emails, or the content of those other emails. For your conversation with the sender, the email chain itself provides sufficient information. We use the same principles to reason about message-passing distributed systems.

We capture this intuitive notion by instrumenting handlers. Specifically, when a handler sends a message without having received a message, our instrumentation begins a new message chain containing only that message. For example, when the `on entry` handler of Fig. 2a is triggered at event 1:0 of Fig. 3, we start a new message chain (in blue) containing a single message from node 1 to node 4 containing the payload 1. When a machine receives a message and reacts in the same handler by sending a message, our instrumentation extends the message chain associated with the

received message. For example, when the `on eNominate` handler of Fig. 2a is triggered at event 4:0, we extend the blue message chain with a message from node 4 to node 2 with payload 4 (event 4:1).

1.3 Verification

Our goal in general is to prove that systems of asynchronously communicating state machines guarantee specified safety properties. The "Verifier" in Fig. 1.h is responsible for this process. It takes a set of properties ("User Invariants" and "Likely Invariants" in Fig. 1.g and Fig. 1.f), and returns true if the conjunction of properties passes a proof by induction. For the ring leader election protocol specifically, we prove that there is only ever a single node that declares itself the winner and that the winning node has the label with the greatest value, no matter how many nodes are included in the system. In general and for the ring leader election protocol specifically, our proofs will rely on message chains.

Textbook proofs of correctness for such a ring leader election protocol often assume synchronous execution of nodes (e.g., Aspnes [2020]). For the synchronous version of this protocol, for every round $k < n$, every node sends the greatest value it has seen so far to the node on its right, and after n rounds the protocol terminates as every node is aware of the greatest value in the ring. Proofs of correctness for the synchronous version of this protocol use induction on the round number. Unfortunately, most realistic implementations are not synchronous and so these textbook proofs of correctness do not apply. In the asynchronous setting, proofs of correctness for this protocol tend to ignore message passing altogether. For example, Koenig et al. [2020] provide a proof in mypyvy that abstracts away individual message buffers into a single, shared, append-only relation.[1] Padon et al. [2016] provide a similar proof in Ivy.

In this paper, we propose a new verification approach based on message chains and use it to prove the correctness of asynchronous distributed systems at the message passing level of abstraction. For example, for the ring leader election protocol, we are able to encode and machine check a proof of correctness that is simple, like the textbook proof, and asynchronous, like the mypyvy proof, all while being at a lower level of abstraction than both, since it captures message passing directly. Specifically, Fig. 2b displays the full verification of the ring leader election protocol in our prototype verification framework, the UPVerifier. This proof, which we revisit in Sec. 3, uses two target invariants and two auxiliary invariants—invariants used to make the target specification inductive. The first invariant captures the desired property that there is only ever a single node that declares itself the winner. The second invariant captures the desired property that the winning node has the label with the greatest value. The third invariant is an auxiliary invariant. It says that the head of every message chain always holds the largest label it has seen so far. The fourth invariant is also an auxiliary invariant. It says that, for every message chain, if there is a node that has not yet participated in the message chain then the message chain is going towards that node. Together, these four invariants capture the textbook proof of correctness over every message chain: for every message chain, the round number in the textbook proof is equal to the length of the message chain.

1.4 Specification Mining

Choosing properties and formally expressing them can be challenging for users, so we employ specification mining techniques to lessen this burden. This process corresponds to the "Spec Miner" in Fig. 1.e and the output of this process corresponds to "Likely Invariants" in Fig. 1.f.

The intuition behind our approach is that message chains reveal enough structure that they can even be used to automatically learn meaningful specifications from only distributed system execution data. That is, given only example message chains (no access to the system definition or

[1] Available at https://github.com/wilcoxjay/mypyvy/blob/pldi20-artifact/examples/fol/ring_id.pyv

specifications), existing learning techniques can discover properties that hold for all executions of a system. For example, in Sec. 4.1 we describe how an existing program synthesis technique can be used to discover the third invariant of Fig. 2b using only example message chains. Without message chains, given only example execution logs, the same off-the-shelf, state-of-the-art tool is unable to learn any meaningful specifications.

This learning from examples problem is usually called specification mining [Ammons et al. 2002] or likely invariant synthesis [Ernst et al. 2001] and has applications beyond verification. For example, specification mining can be used for automated program repair [Demsky et al. 2006], testing [Schuler et al. 2009], system understanding [Beschastnikh et al. 2015], and more. In this paper, we focus on validating the use of message chains for specification mining by showing that learning techniques can discover invariants that appear in our manual verification efforts.

1.5 Roadmap and Contributions

The rest of this paper is organized as follows. We begin by giving the necessary background on SMT and I/O Automata in Sec. 2. We use this background to formally define the verification problem and our encoding in Sec. 3. The problem definition captures systems with any number of machines executing for any number of steps; the encoding captures the notion of message chains and the instrumentation needed to use them. Sec. 3 also formally defines message chain invariants and gives a detailed verification of the running example using them. In Sec. 4 we formally define the specification mining problem and describe two approaches for mining message chain invariants. In Sec. 5 we implement an instance of our verification approach for the P programming language specifically and we call the resulting tool the UPVerifier. We then empirically evaluate our verification and specification mining approaches in Sec. 6. We conclude by surveying related work in Sec. 7. Overall, we make the following contributions.

(1) We define message-passing distributed systems based on I/O Automata. We define the verification problem and build a framework that compiles the problem to satisfiability modulo theories (SMT) [Barrett et al. 2021] queries.
(2) We define the notion of message chains and integrate it into our message-passing distributed systems. This notion is suited to programming languages like Erlang [Telefonaktiebolaget LM Ericsson 2022], Akka [Lightbend, Inc 2022], and P [Desai et al. 2013].
(3) We instantiate our framework for a version of the P language and call the result the UPVerifier. We then replicate verification efforts from related work, and simplify these proofs using message chains. We then verify a correctness property for a system with complex message passing—the onion routing network—and two industrial case studies.
(4) We define the notion of positive and negative message chain examples and describe how to use existing learning and program synthesis techniques to automatically mine specifications from examples only. We evaluate this technique by automatically mining useful specifications for the distributed systems we verified.

2 BACKGROUND

In this section, we give the necessary background on I/O Automata (Sec. 2.1) and SMT (Sec. 2.2).

2.1 I/O Automata Background

We present a version of I/O Automata [Lynch 1996] tailored to our context along with examples focused on our encoding. Let *states* and *actions* be disjoint sets. An I/O Automaton A is a six-tuple

$$(inp(A), int(A), out(A), states(A), start(A), trans(A)),$$

where $inp(A)$ (input actions), $int(A)$ (internal actions), and $out(A)$ (output actions) are disjoint subsets of *actions*, $states(A)$ is a subset of *states*, $start(A)$ is a subset of $states(A)$, and $trans(A)$ is a transition relation over state-action-state triples. Specifically, $trans(A)$ is a relation on $states(A) \times inp(A) \cup int(A) \cup out(A) \times states(A)$. We call individual triples (s, a, s') in $trans(A)$ *steps* of A. Like Lynch, we require I/O Automata to be *input-enabled*. That is, we require that for every state s and action $\alpha \in inp(A)$ there exists a state s' such that $(s, \alpha, s') \in trans(A)$.

Example 2.1 (Simple Universal Buffer I/O Automata). We define an I/O Automaton called the simple universal buffer (*SUB*). Intuitively, *SUB* represents a set of integers that other external automata will be able to add to or remove from. Formally, for a non-empty index set R, let $inp(SUB)$ be $\{put(v)_{i,j} \mid v \in \mathbb{Z};\ i, j \in R\}$; $int(SUB)$ be $\emptyset$; $out(SUB)$ be $\{get(v)_{i,j} \mid v \in \mathbb{Z};\ i, j \in R\}$; $states(SUB)$ be the set of all sets of integers unioned with a special error state, i.e., $2^{\mathbb{Z}} \cup \{\bot\}$; $start(SUB)$ be the singleton set containing the empty set of integers; and $trans(SUB)$ be a relation such that $put(v)_{i,j}$ actions add the integer v to the state, and, if v is in the set, $get(v)_{i,j}$ actions remove the integer v from the state. If v is not in the set, then $get(v)_{i,j}$ actions move *SUB* to the error state. Once in the error state, no actions change the state. Fig. 4a illustrates the interface of *SUB*, where input (output) actions are arrows toward (away from) the automaton.

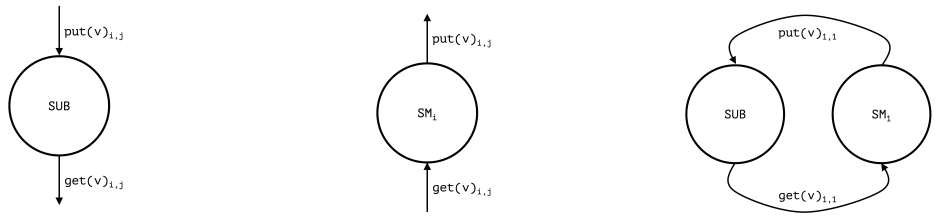
We often define transition relations piecemeal through precondition-effect pairs.

Definition 2.2 (Precondition-Effect Pair). Let A be an I/O Automaton. A precondition-effect pair (p, e) of A consist of a predicate $p(s, \alpha)$ over an input state and an input action, and a function $e(s, \alpha)$ that takes in an input state and an input action and returns a new state. Given a precondition-effect pair (p, e), if $p(s, \alpha)$ is true and $e(s, \alpha) = s'$, then we say that $(s, \alpha, s') \in trans(A)$. As a matter of convention, when the I/O automaton in question has an error state (like $\bot$ for *SUB*), actions that do not satisfy any precondition move the state of the I/O automaton to that error state. This convention makes it easier to define input-enabled I/O automata in terms of precondition-effect pairs. When non-determinism is needed, we extend p and e to take extra, non-deterministic arguments. However, to simplify the presentation, we elide non-deterministic arguments for the remainder of this paper.

For example, the simple universal buffer (*SUB*) has a precondition-effect pair with precondition $p(s, get(v)_{i,j}) := v \in s$ and effect $e(s, get(v)_{i,j}) := s \setminus v$. This precondition asserts that v must be in the set (we cannot get an element which is not present), and this effect removes v from the set s.

Definition 2.3 (I/O Automata Composition). When I/O Automata are *compatible*, they can be *composed* to form new I/O Automata. A finite, non-empty set of I/O Automaton M are compatible if $\forall A, B \in M\ A \neq B \implies int(A) \cap int(B) = \emptyset$ and $\forall A, B \in M\ A \neq B \implies out(A) \cap out(B) = \emptyset$. The composition of machines M for a finite, non-empty index set R and enumeration μ, denoted $M\cdot$, is a new I/O Automata where $inp(M\cdot) = \bigcup_{A \in M} inp(A) \setminus \bigcup_{A \in M} out(A)$; $int(M\cdot) = \bigcup_{A \in M} int(A)$; $out(M\cdot) = \bigcup_{A \in M} out(A)$; $states(M\cdot)$ is the set of all arrays s such that for every $i \in R$, $s[i]$ is a member of $states(\mu(i))$; $start(M\cdot)$ is the set of all arrays s such that for every $i \in R$, $s[i]$ is a member of $start(\mu(i))$; and $trans(M\cdot)$ is the set of triples (s, α, s') such that $\forall i \in R$ if α is in the actions of A then $(s[i], \alpha, s[i]') \in trans(s[i])$ and otherwise $(s[i], \alpha, s[i]) \in trans(s[i])$. Note that our definition of composition is associative.

Example 2.4 (I/O Automata Composition). We define a class of I/O automaton called simple machine (SM_i) and then compose an instance of the class with *SUB* from Ex. 2.1. Intuitively, SM_i automaton receive integers from *SUB*, keep track of the most recent integer received, and send integers to *SUB* that are greater than the most recent integer received. Formally, for a non-empty



(a) Simple Universal Buffer (b) Simple Machine (c) *SUB* and SM_1 Composition

Fig. 4. Visualizations of I/O automata for examples 2.1 and 2.4. Irrelevant actions omitted.

index set R, let $inp(SM_i)$ be $\{get(v)_{i,j} \mid v \in \mathbb{Z};\ j \in R\}$; $int(SM_i)$ be $\emptyset$; $out(SM_i)$ be $\{put(v)_{i,j} \mid v \in \mathbb{Z};\ j \in R\}$; $states(SM_i)$ be the integers unioned with an error state $\mathbb{Z} \cup \{\bot\}$; $start(SM_i)$ be the singleton set containing the integer zero; and $trans(SM_i)$ be a relation such that $get(v)_{i,j}$ sets the state to v, and $put(v)_{i,j}$ can occur if v is greater than the current state. Fig. 4b illustrates the interface of SM_i. The automaton SM_1 is compatible with *SUB* and so $SM_1 \cdot SUB$ is also an I/O automaton. Fig. 4c illustrates this composition with irrelevant actions omitted. Note that $SM_1 \cdot SUB$ is compatible with every instance SM_j iff $j \neq 1$. Therefore, we could further compose simple machine instances.

Definition 2.5 (I/O Automata Executions). Let A be an I/O Automaton. An *execution* of A is a finite sequence of alternating states and actions $s_0, \alpha_1, s_1, ..., \alpha_k, s_k$ such that s_0 belongs to $start(A)$, every s_i belongs to $states(A)$, and every triple $(s_i, \alpha_{i+1}, s_{i+1})$ belongs to $trans(A)$. We use $exec(A)$ to refer to the set of executions of A. We say that a state s is *reachable* by A if there exists an execution $s_0, \alpha_1, s_1, ..., \alpha_k, s_k \in exec(A)$ with $s_k = s$.

For example, the sequence $\emptyset, put(7)_{i,j}, \{7\}, get(7)_{i,j}, \emptyset$ is an execution of *SUB* from Ex. 2.1. On the other hand, the sequence $\emptyset, get(7)_{i,j}, \emptyset$ is not an execution of *SUB*.

Definition 2.6 (I/O Automata Verification). An *invariant assertion* (invariant) is a predicate p over states. We say that an invariant holds of an I/O Automaton A if for all reachable states s, $p(s)$ holds. A *counterexample* is a reachable state s such that $p(s)$ does not hold. An *inductive invariant* is an invariant p such that for every step $(s, \alpha, s') \in trans(A)$ if $p(s)$ holds then $p(s')$ holds. A *counterexample to induction* is a step $(s, \alpha, s') \in trans(A)$ such that $p(s)$ holds but $p(s')$ does not. A *proof by induction* for an invariant p is a proof that p is inductive and that $p(s)$ holds for every state $s \in start(A)$. A *verification* of an I/O Automata is a successful proof by induction.

For example, consider $SM_1 \cdot SUB$ from Ex. 2.4 and the invariant assertion which states that every element in the set maintained by the *SUB* component is greater than zero. That is, the invariant is $p([s_{SM_1}, s_{SUB}]) := \forall_{i \in s_{SUB}}\ i > 0$. A counterexample to induction for this I/O Automaton and invariant is the step $([-1, \{\}], put_0, [-1, \{0\}])$. Note that the pre-state $[-1, \{\}]$ is not reachable and that the invariant does indeed always hold. Nevertheless, the proof by induction fails.

2.2 Satisfiability Modulo Theories (SMT) Background

We assume a working knowledge of SMT and the standard logical theories over integers, arrays, and sequences. For more details, we refer the reader to the SMT-LIB standard [Barrett et al. 2016] and works on the specific theories (e.g., [Barrett et al. 2021; Bjørner et al. 2012]). In the remainder of the paper, we refer to the SMT-LIB standard presentation of many-sorted first-order logic with algebraic data types built-in, along with integer, array, and sequence theories, as SMT.

Fig. 5 shows the abstract syntax of untyped SMT terms and values. An SMT *formula* is a well-typed term of sort `Bool` with no free variables. An SMT interpretation I is a mapping on signatures

```
t ::= v                          | x                          | f t
    | c(s1:= t1, ...., sn:= tn)  | t.s                        | t is c
    | t1[t2]                     | t1[t2 -> t3]               | concat(t1, ..., tn)
    | forall (x)  t              | exists (x)  t              | |t|
    | let x := t1 in t2          | if t1 then t2 else t3

v ::= true                       | false                      | integer
    | c(s1:= v1, ..., sn:= vn)   | v*
    | const v                    | v1[v2 -> v3]
```

Fig. 5. Untyped SMT-LIB term and value abstract syntax. `integer` represents all concrete integer values and v^* represents all concrete, finite sequence values. The syntax $t.s$ stands for $s(t)$, t_1`[`t_2`]` stands for `select`(t_1, t_2), t_1`[`t_2 `->` t_3`]` stands for `store`(t_1, t_2, t_3), and $|t|$ stands for `length`(t).

and we require that every interpretation obeys the semantics of the SMT-LIB standard and the respective theory definitions. An Interpretation I *satisfies* a formula t if t evaluates to true under the interpretation I. In this case, we write $I \models t$ and call I a *model* of t. When every model of t_1 is also a model of t_2, we re-use notation and write $t_1 \models t_2$. We say that a formula is *satisfiable* (*unsatisfiable*) if there (does not) exists an interpretation that satisfies it.

We deviate from the SMT-LIB standard slightly in the vocabulary we use related to algebraic data types. The following clarifies the connection. If a sort t has a non-empty set of constructors, then we call t an *algebraic data type*. If t has a single constructor c and every selector s of c is such that it is not the case that t appears in the sort of s, then we call t a *record*. When t is a record with constructor c, we call every selector s of c a *field* of t. If t is an algebraic data type and every constructor of t has no associated selectors, then we call t an *enum*. When t is an enum, we call every constructor of t a *variant* of t and often treat the enum as a set.

We also deviate from the SMT-LIB standard in that we support ad-hoc polymorphism for function definitions (macros). This allows for a cleaner presentation when frequently using sort parameters.

2.3 I/O Automata In SMT

Our encoding of I/O Automata into SMT is straightforward: each element of the six-tuple defining I/O Automata maps to an SMT construct. Specifically, for an I/O Automaton A, we encode the set of actions, *actions*, as a sort a, and we encode $inp(A)$, $int(A)$, and $out(A)$, as predicates on a. We encode the state space, *states*, as a sort s and we encode $states(A)$, as a predicate on s. Similarly, we encode $states(A)$, as a predicate on s. Finally, we encode $trans(A)$, as predicates on triples (s, a, s). When transitions are defined by precondition-effect pairs, we encode the precondition as a predicate over an input state and an input action, and the effect as a function that takes in an input state and an input action and returns a new state. For our encoding, we require that there are a finite number of precondition-effect pairs per query.

3 VERIFICATION APPROACH

In this section, we formulate the problem of verifying asynchronously communicating state machines, we describe our verification workflow, and we define *message chain invariants*. All of this is built on top of Folds Automata (first-order logic distributed systems automata), a short-hand for parametric compositions of I/O automata with restrictions that

(1) make modelling distributed systems easy (see Sec. 5),
(2) make encoding these systems into SMT straightforward (see Sec. 3.3), and

(3) enable our verification workflow by integrating message chains.

3.1 Folds Automaton: Encoding Distributed Systems as I/O Automata

At a high-level, Folds Automata represent compositions of many state machines (similar to instances of SM_i of Ex. 2.4) with a single universal buffer (similar to *SUB* of Ex. 2.1). Our main contribution, and one of the key differences between Folds Automata and the I/O automata in examples 2.1 and 2.4, is that, instead of communicating by exchanging integers, Folds Automata communicate with algebraic data types representing chains of messages.

Definition 3.1 (Message Chains). `MC` is an algebraic data type with two sort parameters and two constructors. The sort parameters are `E`, representing the type of messages, and `R`, representing the set of machine identifiers. The two constructors are `empty` and `send`. The constructor `empty` has no selectors. The constructor `send` has four selectors: `source`, `target`, `payload`, and `history`. The `source` selector represents the machine that sent the current message and is of sort `R`. The `target` selector represents the destination of the current message and is of sort `R`. The `payload` selector represents the contents of the current message and is of sort `E`. The `history` selector represents the tail of the message chain and is of sort `MC[E, R]`. A message chain is an instance of the `MC` algebraic data type. The following functional pseudo-code snippet captures this definition succinctly.

```
1  data [E, R] MC := | empty
2       | send {source: R, target: R, payload: E, history: MC[E, R]}
```

For example, `send(2, 3, eNominate(4), send(4, 2, eNominate(4), empty))` is the green message chain in Fig. 3. Note that message chains are essentially lists of triples. For convenience, we sometimes treat message chains as sequences of triples instead. This is most relevant in Sec. 4.

Definition 3.2 (Folds Automata). A Folds Automaton A is a triple (E, R, M_i), where E is a sort representing messages, R is a sort representing machine identifiers, and M_i is a class of I/O Automata that satisfies the following conditions.

(1) For every $i \in R$, M_i is an I/O automata.
(2) If $i, j \in R$ $i \neq j$ then M_i and M_j are compatible.
(3) $inp(M_i)$ is $\{get(v)_{i,j} \mid v \in 2^{MC};\ j \in R\}$.
(4) $out(M_i)$ is $\{put(v)_{i,j} \mid v \in 2^{MC};\ j \in R\}$.
(5) $states(M_i)$ is an error state $\perp$, or a record with two sets of message chains, $input_i$ and $output_i$.
(6) $start(M_i)$ must only include non-error states where $input_i$ and $output_i$ are empty.
(7) $trans(M_i)$ is a relation such that $get(v)_{i,j}$ updates $input_i$ to be $input_i \cup v$, and $put(v)_{i,j}$ updates $output_i$ to be $output_i \setminus v$, if v was in $output_i$ (otherwise M_i moves to the error state $\perp$).
(8) No internal action adds to $input_i$.
(9) If an internal action removes from $input_i$, then it removes exactly one message chain. We call this message chain the *received* message chain.
(10) Internal actions can only depend on the received message chain.
(11) No internal action removes from $output_i$.
(12) Every message chain that is added to $output_i$ is of the form `send`(e, h), where e is a message and h is either the received message chain, if the internal action received a message chain, or `empty` if the internal action did not receive a message chain. We call e the current message in the message chain and we call h the history of the message chain.
(13) No internal action can depend on the history of the received message chain.
(14) Internal actions can only add to $output_i$ after any local state changes; internal actions can only make local state changes after any $input_i$ removal.

Intuitively, the first seven conditions (1-7) define network communication: machines send and receive sets of message chains and have input and output buffers that are sets of message chains. The next four conditions (8-11) ensure that machines use their own input and output buffers correctly. Condition 12 ensures that message chains are constructed correctly. Condition 13 ensures that message chains do not affect the behavior of the system we are modeling. That is, message chains behave like ghost variables. Finally, condition 14 ensures that, even though machines take turns stepping, our system is equivalent to an asynchronous composition. This follows from the same argument put forth by Padon [2018], which is in turn due to Lipton [1975].

To give the semantics of Folds Automata, we first give the definition of the Universal Buffer, which mediates machine communication by holding and distributing message chains.

Definition 3.3 (Universal Buffer). for a given non-empty index set R, the Universal Buffer UB, is an I/O automata such that $inp(UB)$ is $\{put(v)_{i,j} \mid v \in 2^{MC};\ i, j \in R\}$; $out(UB)$ is $\{get(v)_{i,j} \mid v \in 2^{MC};\ i, j \in R\}$; $int(UB)$ is the empty set; $states(UB)$ is $2^{MC} \cup \{\bot\}$; $start(UB)$ is the singleton set containing the empty set; and $trans(SUB)$ is a relation such that $put(v)_{i,j}$ updates the state s to $s \cup v$ and $get(v)_{i,j}$ updates the state s to $s \setminus v$, if s contained v (otherwise SUB moves to the error state $\bot$).

A Folds Automaton (E, R, M_i) is equivalent to the I/O Automata composition of $\{M_i \mid i \in R\} \cup \{UB\}$. All definitions that apply to this composed I/O Automata can be lifted to Folds Automata. For example, an *invariant assertion* on a Folds Automaton F is a predicate p over states of the I/O Automata corresponding to F. Similarly, a *proof by induction* for an invariant p of a Folds Automaton F is a proof that p is inductive and that $p(s)$ holds for every starting state s of F.

We wish to verify Folds Automata for all possible values of R. For example, we want to verify that the ring leader election protocol is correct no matter how many nodes participate in the election. To do this, we define parametric Folds Automata.

Definition 3.4 (Parametric Folds Automata). A *parametric Folds Automaton* is a function $F(R) := (E, R, M_i)$, where E is a sort representing messages, R is the parameter, and M_i is a class of I/O Automata satisfying the Folds Automaton conditions. We call every output of F an *instance* of F.

3.2 Problem Definition: Parametric Verification by Induction

Given a parametric Folds Automaton F and an invariant p, the verification problem is to verify every possible instance of F. Specifically, the verification problem is to check

(1) $\forall R \, \forall s \in start(F(R))\ p(s)$; and
(2) $\forall R \, \forall (s, \alpha, s') \in trans(F(R))\ p(s) \implies p(s')$.

In our setting, given our encoding into I/O automata, error states encode the malfunction of the runtime itself—something that we do not wish to reason about. For example, an action $get(v)_{i,j}$ when v is not in the set maintained by the Universal Buffer will result in the Universal Buffer transitioning to the error state $\bot$ and the message chain v "magically" appearing in the input buffer of the j^{th} machine. Therefore, to avoid nonsensical counterexamples deriving from these transitions, we implicitly assume that all invariants are such that $inv(s) = inv(s) \vee error(s)$, where $error$ is a predicate that returns true iff any machine is currently in an error state. When input programs do not specify an initial set of states, we take it to be equal to the conjunction of invariants.

3.3 Encoding and Complexity

Encoding Folds Automata is exactly like encoding I/O automata. Encoding parametric Folds Automata requires only a small change. Specifically, we encode the state space of a parametric Folds Automaton as an extensional array whose index sort is a parameter and whose element

sort is the sort representing states of the individual I/O Automata in the composition. The problem of verification by induction can then be encoded as an SMT query using an uninterpreted sort for the parameter of the parametric Folds Automaton. Satisfying interpretations represent counterexamples, while unsatisfiability proofs represent proofs of correctness.

When invariants contain quantifiers, the verification queries are as expressive as first-order logic and the problem is undecidable. Note that there are fragments of first-order logic that are decidable (e.g., EPR [Lewis 1980]) but we support logical theories that are outside of these fragments (e.g., sequences for EPR). When no invariant contains quantifiers, the verification by induction problem can be posed as a quantifier-free SMT query. Unfortunately, it is a long-standing open problem to determine the decidability of quantifier-free queries over sequences with length constraints [Day et al. 2023] and so the decidability of our verification by induction problem is also unknown. Without quantifiers and without sequences, since the theories of equality, linear integer arithmetic, arrays, and algebraic data types are strongly polite and decidable [Bonacina et al. 2019; Sheng et al. 2020], their combination and our verification problem is decidable.

3.4 Message Chain Invariants

Our problem formulation implicitly supports a novel kind of invariant, *message chain invariants*, that developers can use along with their usual invariants to verify their systems. Intuitively, message chain invariants describe the way that messages flow through a distributed system. Formally, we define a message chain invariant to be an invariant of the from

$$\forall c_1, ..., c_n \in MC \bigwedge_{i}^{n} alive(s, c_n) \implies r(c_1, ..., c_n),$$

where s is the current state of the system, *alive* is a predicate that returns true iff the message chain c_i is not empty and is held by the universal buffer or any machine's buffers, and r is a predicate over message chains. We call r the message chain invariant since the rest of the predicate is fixed.

Consider the four message chains in Fig. 3. One message chain invariant that is true for all four message chains is that the payload at the head of every message chain holds the largest label value seen so far by that message chain. This turns out to be true for all message chains in the system, and is a crucial fact used in the verification of the ring leader election protocol.

Since predicates in our language can be recursive and solvers directly support these functions [Suter et al. 2011], message chain invariants over a single message chain ($n = 1$) can theoretically express any recursively enumerable language over sequences of messages. When properties are over more than a single message chain ($n > 1$), these invariants can express properties akin to hyperproperties for traces [Clarkson and Schneider 2010]. However, message chains invariants cannot express properties about the state of individual machines.

3.5 Ring Leader Election Protocol Verification Example

To better understand message chain invariants and our end-to-end workflow, consider the P implementation and verification by induction query for the ring leader election protocol displayed in Fig. 2. This verification by induction query has two target invariants. The first invariant, `unique_leader`, states that there is only ever one node that declares itself as the winner. The second invariant, `leader_max`, states that this winning node has the greatest label. Note that neither of these two invariants are message chain invariants: we support message chain invariants, non message chain invariants, and combinations thereof.

Proposition 3.5 (Ring). *The protocol introduced in Sec. 1.1 and implemented in Fig. 2a is correct.*

Proof. We define the semantics of P programs in terms of Folds Automata in Sec. 5. Given these semantics, the verification query in Fig. 2 without the last two invariants, is not inductive. This can be explained by two counterexamples. Consider a two-node system—nodes labeled 1 and 2—and the following two message chains

(1) c := `send(2, 1, eNominate(1), send(1, 2, eNominate(1), empty))` and
(2) c' := `send(1, 1, eNominate(1), empty)`.

For the first counterexample, suppose node 2 has declared itself the winner and $input_1$ contains the message chain c. This system will satisfy both target invariants before taking a step but will falsify `unique_leader` when node 1 receives c. For the second counterexample, suppose node 2 has declared itself the winner and $input_1$ contains the message chain c'. Again, this system will satisfy both invariants before taking a step but will falsify `unique_leader` when node 1 receives c'.

To complete the proof by induction, the user can introduce two auxiliary message chain invariants—the last two invariants in Fig. 2. The first auxiliary invariant blocks the first counter-example by asserting that the head of every message chain always holds the largest label it has seen so far. Formally, this is the message chain invariant

$$r(c) := \forall n\ participated(c, n) ==> n.id \leq c.id,$$

where $participated(c, n)$ is a helper function that is true iff the node n was the source or target of a message in the message chain c. For the counterexample two-node system above, the head of c cannot hold the label 1 since the node labeled 2 has participated in c, thus our added auxiliary invariant r blocks the first counter-example.

The second auxiliary invariant blocks the second counterexample above by asserting that, for every message chain, if there is a node that has not yet participated in the message chain then the message chain is going towards that node. Formally, this is the message chain invariant

$$r'(c) := \forall n\ \neg participated(c, n) ==> (between(c.id, c.target, n) \vee c.target = n),$$

where $between(c.id, c.target, n)$ is a helper function that is true iff the target of the message chain c is between the node labeled $c.id$ and the node labeled n in the ring. For the counterexample two node system above, the node labeled 2 has not participated in the message chain c' so the target of c' should be 2 (not 1), thus our added auxiliary invariant r' blocks the second counterexample.

Together, these two auxiliary invariants block all counterexamples to induction for all system instances—not just for the two-node system above. The UPVerifier, our prototype implementation for the P programming language defined in in Sec. 5, using Z3 [de Moura and Bjørner 2008], will accept this proof by induction in less than a tenth of a second (see Sec. 6). This proof is like doing the textbook proof of correctness described in Sec. 1 on every message chain at once: for every message chain, the round number in the textbook proof is equal to the length of the message chain. Yet our proof captures the nitty-gritty details of individual machine message passing. □

4 SPECIFICATION MINING

Up until now, we have assumed that users provide system models and specifications. This is the same requirement imposed by related work. However, in practice, specifications are not always obvious and formally expressing them is not always easy. In this section, we describe an approach to mining specifications—message chain invariants—using existing learning techniques.

The inputs to a specification mining tool are a set of pairs. The first element of each pair is a message chain and the second element is a Boolean indicating if the message chain can appear in an execution of the system. We call these pairs *examples*, and we say that an example is a *positive* (*negative*) example if the Boolean is true (false). For example, for the ring leader election protocol,

(1) (`Send(1, 4, eNominate(1), Empty)`, `true`) is a positive example and

(2) `(Send(1, 4, eNominate(4), Empty)), false)` is a negative example.

Given a set of positive and negative examples, the output of a specification mining tool is a function that evaluates to true for the positive examples and false for the negative examples. Intuitively, this function represents a system specification that is *likely* to hold, but is not guaranteed to hold. Note that allowing negative examples makes this definition slightly more general than the related definitions of Ernst et al. [2001] and Ammons et al. [2002]. Positive examples can be generated easily by fuzzing the target system. Negative examples, on the other hand, are harder to generate automatically. In our setting, we take negative examples to be hints from the user: they are examples that the user believes can never occur during an execution.

4.1 Specification Mining Using Recursive Function Synthesis

When message chains are represented as algebraic data types, our specification mining problem matches the programming-by-example problem [Halbert 1984] for functional, recursive programs. In this section, we describe how to use an existing program synthesis technique in this space, Burst [Miltner et al. 2022], to mine message chain invariants.

Burst is a synthesis tool based on bottom-up enumeration. The Burst algorithm, by design, will produce syntactically short functions. Burst begins by synthesizing a recursive function that satisfies the specification (positive and negative examples) assuming that undefined recursive calls behave exactly as required to satisfy the specification (referred to as "angelic" semantics). It then checks whether the synthesized recursive function satisfies the specification under the actual semantics, and, if not, strengthens the specification based on the existing assumptions. This process repeats until a recursive function is found that satisfies the specification and has no undefined recursive calls. However, the synthesizer may have to backtrack some of the specification strengthening if the specification is made unsatisfiable by any added assumptions.

At a high-level, to synthesize message chain invariants using Burst, we give it queries consisting of the algebraic data type definition of message chains; a function signature (maps a message chain to a Boolean), examples (message chains that occur and do not occur); and a basic library of helper functions. This library includes functions for computing logical operators (like conjunction and negation) and simple functions for reasoning about message chains (like a function that takes a message chain and returns the innermost message if one exists). Burst then returns recursive functions that evaluate to true on all positive examples and false on all negative examples.

4.2 Specification Mining Using Extended Non-Erasing Pattern Learning

When message chains are represented as sequences, our specification mining problem matches the problem of learning formal languages from examples. In this section, we describe how to use non-erasing pattern languages to mine message chain invariants using only positive examples. We first give the necessary background on non-erasing pattern learning and then provide an extension to tailor it to our distributed systems domain.

4.2.1 Non-Erasing Pattern Languages Background. We present patterns and pattern languages [Angluin 1980], but with definitions tailored to SMT-LIB. Let Σ be a zero-ary sort with finite cardinality greater than two, which represents a finite alphabet of size equal to the cardinality of the sort. Let $X = \{x_1, x_2, \ldots\}$ be a countable set of symbols disjoint from Σ, which we refer to as pattern variables. A sequence over Σ is an n-ary concatenation of symbols where every $t_1, \ldots, t_n$ is an element of Σ, and the set of non-zero finite sequences over Σ is denoted Σ^+.

A *pattern* p is a sequence over $\Sigma \cup X$, i.e., an n-ary concatenation of symbols $t_1, \ldots, t_n$ where every t_i is an element in $\Sigma \cup X$. We use juxtaposition to represent concatenation, so write $p := t_1 \ldots t_n$. We use $Var(p)$ to denote the pattern variables that occur in p. Patterns define languages in the

following way. For a pattern p, call $e(\omega') := p = \omega' \wedge \bigwedge_{x \in Var(p)} |x| \geq 1$ the *characteristic formula* of p, where the pattern variables in p are free variables, $|x|$ denotes the length of x, and ω' is a given sequence in Σ^+. The language of a pattern p, $L(p)$, is the largest set of sequences such that for every $\omega \in L(p)$, $e(\omega)$ is satisfiable. When m is a model of $e(\omega)$ for a particular p, we say that m *justifies* $\omega \in L(p)$. For example, for $\Sigma := \{f, g, h\}$ and $X := \{x_1, x_2, ...\}$ the pattern $p := x_1x_2x_3x_2x_1$ defines a pattern language $L(p)$ that contains the sequence *fghgf* and the sequence *hfggggfh* but does not contain *fghfg*, and the model $\{x_1 := f, x_2 := g, x_3 := h\}$ justifies that $fghgf \in L(p)$. We use this notion of justification to, at a high level, learn the most strict but justified semantic constraints on patterns over a useful concept class. The characteristic formula constrains the model to map each variable in $Var(p)$ to a concatenation of *at least one* symbol in ω. We do not use erasing patterns, where variables can map to the empty sequence, in this work.

Angluin [1980] defines *ℓ-minl* and Shinohara [1982] presents an algorithm to compute it. The *ℓ-minl* algorithm takes in a set of sequences in Σ^+ and returns the longest pattern that best describes the examples. We treat the *ℓ-minl* algorithm as a black box but rely on the following theorems.

Theorem 4.1 (Angluin [1980]). *Let Σ be a zero-ary sort with finite cardinality greater than two, let $\Omega \subseteq \Sigma^+$ be a set of finite sequences over Σ and let p be the output of the ℓ-minl algorithm. Then p is the longest pattern such that $\Omega \subseteq L(p)$ and there is no pattern q with $\Omega \subseteq L(q)$ and $L(q) \subsetneq L(p)$.*

Theorem 4.2 (Angluin [1980], Shinohara [1982]). *The ℓ-minl algorithm can be computed by a deterministic polynomial-time Turing machine using an oracle in NP. Specifically, for a given pattern p and set of sequences $\Omega \subseteq \Sigma^+$, the NP oracle checks $\Omega \subseteq L(p)$ by checking $\omega \in L(p)$ at most $|\Omega|$ times. This membership check is NP-Complete for erasing and non-erasing patterns [Jiang et al. 1994].*

4.2.2 Extended Non-Erasing Pattern Learning. Given only positive input message chain examples, the *ℓ-minl* algorithm can quickly learn a non-erasing pattern that likely holds for all message chains in the target system. Note that most similar learning algorithms, like those for learning regular languages, require both positive and negative examples. Unfortunately, non-erasing patterns on their own are not specific enough for our domain. This is in part due to the fact that no non-erasing pattern containing at least one variable cannot represent a finite language, and many distributed systems only exhibit bounded message chains. For example, suppose you have a simple client-server system where clients send requests to servers and servers respond to the client. Every message chain in this system has length at most two, but no non-erasing pattern can represent this language.

To fix this, we extend the *ℓ-minl* algorithm with an enumerative approach that learns a conjunction of constraints. These constraints are useful for the distributed systems domain and yet guarantee that the enumerative algorithm terminates. We call the resulting pattern a path-pattern and denote it $p[l]$, where p is the base *ℓ-minl* pattern and l is a set of added constraints. When adding constraints, the goal is to be as specific as possible without being too specific. We formalize the notion of *too specific* in the following definition. Given a set of sequences Ω and a pattern p with characteristic formula e, we say that a path-pattern $p[l]$ is *too specific* iff $\bigvee_{\omega \in \Omega} e(\omega) \not\models l$. That is, if every model that can be used to justify the training set is preserved by l, l is not too specific.

Consider the following two examples. First, let $\Sigma := \{f, g\}$, let $\Omega := \{fg, gf, ff\}$, and the corresponding pattern $p := x_1x_2$. In this case, the constraint $l := |x_1| = 1 \wedge |x_2| = 1$ is better than the constraint $l' := true$ because it gives us more information about the variables in the pattern. It is more specific. Second, keep Σ and p as before, but take Ω to be $\{fg, gff, gggg\}$ and consider the three different constraints $l := |x_1| = 1$, $l' := |x_2| = 1$, and $l'' := true$. While it is the case that $\Omega \subseteq L(p[l]) = L(p[l']) = L(p[l''])$, the first two constraints are too specific while the last constraint is not. That is, they impose constraints on the roles the variables in the pattern play that are not supported by evidence: if these path-patterns represented machine indexes, the first path-pattern would say that x_1 always represents a single machine, the second path-pattern would

say that x_2 always represents a single machine, and the last path-pattern would make no such unfounded judgments. Therefore, in this case, we would consider l'' to be the best choice. We now describe our procedure for learning two kinds of constraints over *ℓ-minl* patterns: length constraints and membership constraints.

Learning Lengths The procedure to discover length constraints uses the following proposition.

Proposition 4.3 (Lengths). *Let Ω be a set of sequences, $p[l]$ be a pattern such that $\Omega \subseteq L(p[l])$ and $p[l]$ is not too specific, $x \in Var(p)$ be a variable, $x_1, x_2 \notin Var(p)$ be two fresh variables, and $p' \coloneqq p\langle x_1x_2/x\rangle$ be p but with all occurrences of x substituted with x_1x_2. If $\Omega \cap L(p'[l]) = \emptyset$, then*

(1) $p[l \wedge |x| = 1]$ is more specific than $p[l]$ and
(2) $p[l \wedge |x| = 1]$ is not too specific.

Proof. (1) Let e_l be the characteristic formula of $p[l]$. The characteristic formula of $p[l \wedge |e| = 1]$ is then $e_l \wedge |x| = 1$ and $e_l \wedge |x| = 1 \models e_l$ holds by the semantics of conjunction. (2) Now suppose for contradiction that $p[l \wedge |x| = 1]$ is too specific. That is, suppose that

$$\bigvee_{s \in S} e_l(\omega) \not\models l \wedge |x| = 1.$$

Then there exists a model m of $\bigvee_{\omega \in \Omega} e_l(\omega)$ that is not a model of $l \wedge |x| = 1$ and that interprets x as the sequence z such that $e_l(\omega)$ evaluates to true for some sequence $\omega \in \Omega$. Since $m \models l$ ($p[l]$ is not too specific by assumption) we have that $m \not\models |x| = 1$ and therefore that z is of length at least two. Let $e_{p'}$ be the characteristic formula of $p'[l]$ and let m' be the model that is like m in every way but extended to interpret x_1 as the first element of z and x_2 as the rest of z. The contradiction is that we have $m' \models e_{p'}(\omega)$ but, by the antecedent of the implication, $\omega \notin L(p'[l])$. □

The procedure itself starts with the set of constraints $\bigwedge_{x \in Var(x)} |x| = 1$. For every $x \in Var(p)$, we check $\Omega \cap L(p'[l]) = \emptyset$, where p' is constructed as in Prop. 4.3. If $\Omega \cap L(p'[l]) = \emptyset$ holds for x, then we keep $|x| = 1$ in the conjunction and we move to the next variable. If it does not hold, then we remove $|x| = 1$ from the conjunction and we move to the next variable.

Proposition 4.4 (Effective Procedure for Discovering Lengths). *Let S be a set of sequences over Σ, p be a pattern with $\Omega \subseteq L(p)$, and l be the output of the above procedure. We claim that $p[l]$ is not too specific and that for every set of constraints l' of the same form that is not too specific, $p[l]$ is more specific than $p[l']$. Furthermore, the above procedure runs in $O(|p|)$ time using an oracle in NP.*

Proof. We use the same oracle as the *ℓ-minl* algorithm in Thm. 4.2 but instead of checking $\Omega \subseteq L(p)$ we check $\Omega \cap L(p'[l]) = \emptyset$, where p' is constructed as in Prop. 4.3. We call this oracle once for every variable that appears in p, which is at most the size of p. Correctness follows directly from Prop. 4.3 and two facts. First, if $|x| = 1$ is too specific on its own, then so is every conjunction that contains $|x| = 1$. Second, the most specific set of constraints is the largest set. □

Discovering Membership Discovering membership constraints follows a similar process. To make things more tractable, we assume that users supply a list of pairwise disjoint subsets of Σ of interest. The following proposition is key.

Proposition 4.5 (Membership). *Let Ω be a set of sequences over Σ, $p[l]$ be a pattern such that $\Omega \subseteq L(p[l])$ and $p[l]$ is not too specific, $x \in Var(p)$ be a variable, $x_1, x_2 \notin Var(p)$ be two fresh variables, and σ be a strict subset of Σ. If for every $c \in \Sigma \setminus \sigma$ it is the case that $\Omega \cap L(p\langle x_1cx_2/x\rangle\{x_1, x_2\}[l]) = \emptyset$, then (1) $p[l \wedge x \in \sigma^+]$ is more specific than $p[l]$ and (2) $p[l \wedge x \in \sigma^+]$ is not too specific.*

Proof. Similar to Prop. 4.3, suppose for contradiction that $p[l \wedge x \in \sigma^+]$ is too specific. That is, suppose that $\bigvee_{s \in S} e_l(s) \not\models l \wedge x \in \sigma^+$. Then there exists a model m of $\bigvee_{s \in S} e_l(s)$ that is not a model

of $l \wedge x \in \sigma^+$ and that interprets x as the sequence z such that $e_l(s)$ evaluates to true for some sequence $s \in S$. Since $m \models l$ ($p[l]$ is not too specific by assumption) we have that $m \not\models x \in \sigma^+$ and therefore that z is a sequence of the form $y_1 c y_2$, where y_1 and y_2 are fresh, erasing variables and $c \in \Sigma \setminus \sigma$. Let $e_{p'}$ be the characteristic formula of $p\langle x_1 c x_2/x\rangle\{x_1, x_2\}[l]$ for the same c and let m' be the model that is like m in every way but extended to interpret x_1 as the first element of y_1 and x_2 as y_2. The contradiction is that we have $m' \models e_{p'}(s)$ but, by the antecedent of the implication, $s \notin L(p\langle x_1 c x_2/x\rangle\{x_1, x_2\}[l])$. □

The procedure to discover membership constraints is again a simple loop. We start with the set of constraints $\bigwedge_\sigma \bigwedge_{x \in Var(x)} x \in \sigma^+$. For every $x \in Var(p)$ and input subset σ, we check $\Omega \cap L(p'[l]) = \emptyset$ for every $c \in \Sigma \setminus \sigma$, where p' is constructed as in Prop. 4.5. If $\Omega \cap L(p'[l]) = \emptyset$ holds for x for every c, then we keep $x \in \sigma^+$ in the conjunction and we move to the next variable. If $\Omega \cap L(p'[l]) = \emptyset$ does not hold for x for every c, then we remove $x \in \sigma^+$ from the conjunction and we move to the next variable. The proof that this is an effective procedure is the exact same as that of Prop. 4.4 but note that we also depend on the number and size of input sets σ.

In summary, *ℓ-minl* gives us the *best* pattern from a syntactic perspective (Thm. 4.1) and we refine this pattern to get the *best* path-pattern from a semantic perspective. At a high-level, to synthesize message chain invariants using this algorithm, we give it queries consisting of positive examples generated by fuzzing, and it returns path-patterns that capture these examples.

4.3 Combining Specification Mining Approaches Into One Framework

We combine the recursive function synthesis approach and the extended non-erasing pattern learning approach into one framework and, for each target distributed system of interest, we give the framework many different queries. The queries all follow the same structure described above, but we categorize and clean the input data before calling the individual tools.

For categorization, we tag example message chains with the last instruction relevant to them, and we group message chains by this tag. For example, for the ring leader election protocol in Fig. 2a, there are four groups: message chains that just started at line 13; message chains that have reached the leader at line 16; and message chains that have just been extended at lines 18 and 20. For cleaning, we take each group of message chains and we generate new sets of message chains that each focus on a different aspect. Specifically, we generate a set that removes all payload information from message chains, leaving only the sequence of nodes visited by each message chain (the source of each message along with the final target); and we generate a set that removes all source and target information, leaving only the sequence of payload values exchanged.

For the ring leader election protocol, this categorization and cleaning process generates 12 queries (4 groups, 3 versions of queries each). We give all twelve queries to both specification mining methods (using algebraic data type and sequence encoding as appropriate) and we return 24 suggested invariants to the user. The input queries to Burst contain three negative examples generated by hand and 8 positive examples generated automatically by fuzzing. The three negative examples represent three obviously impossible message chain: one where a node sends a value that is not its own and it did not receive; one where a node sends a value to a node that is not on its "right;" and one where a node receives its own value and instead of declaring itself the winner and stopping, it extends the message chain by sending a new message.

5 THE UPVERIFIER

To demonstrate the expressive power of our problem formulation, we implement a verification framework accepting programs written in a version of the P programming language—a large, functional subset that treats message buffers as sets of message chains instead of queues of messages.

We call our implementation the UPVerifier, for unbounded verification of P programs. The UPVerifier implementation is available[2] as an OCaml project consisting of approximately 5000 lines of code. This code also includes fuzzing infrastructure to generate positive example message chains. We implement the extended *ℓ-minl* algorithm as a Python program of approximately 250 lines that uses Z3 for the NP oracle that checks sequence membership. The Python code is available with the OCaml code. We use Burst out of the box for synthesis of recursive functions.

P is a natural fit for our approach because of its domain-specific structure but our approach generalizes to standard message-passing distributed system languages. In particular, the most important aspect of P to support message chains, the notion of a *handler*, is shared across many programming languages. For example, Erlang uses the *receive* keyword and Akka uses the method *receiveMessage* on *Behaviors* for a similar notion.

We support only the core features of P that are specific to distributed systems. For example, we support the definition of machines and messages, but we do not support foreign function calling or modules. We also assume that sequential code blocks—like the body of event handlers—is written in the syntax of Fig. 5 and uses the associated SMT-LIB semantics.

At a high level, our compiler encodes P programs into parametric Folds Automata. We use P event declarations to create a message sort and P machine declarations to create as a class of I/O Automata, the two required components for an Folds Automaton. The sort of messages (E) is given by an algebraic data type with e constructors, where e is the number of events in the input P program. The i^{th} constructor of E holds a single record representing the payload type of the i^{th} event declaration in the input P program.

The state space of P machines follows a similar encoding. It is represented by an algebraic data type M with $m+1$ constructors, where m is the number of machine kind declarations in the input P program. The i^{th} constructor of M holds a single record representing the fields of the i^{th} machine kind declaration in the input P program. Specifically, the i^{th} record holds the variables of the P machines, a state indicator, an entry flag, a "this" reference, and the required $input_i$ and $output_i$ buffers. The state indicator is an enum whose variants are the set of states named by the P machines. The entry flag is a Boolean. The last constructor of M is $\perp$ and has no selectors.

The internal actions of the resulting family of I/O Automata is $\{e_{j,i} \mid e \in N \; j \in R\} \cup \epsilon_i$, where N is the set of event names. We will use the action $e_{j,i}$ to trigger that the current machine, i, receives a message of kind e, from some other machine, j. We will use ϵ_i to trigger P entry handlers: the first code block that executes after a machine enters a given state. The transitions are defined by precondition-effect pairs derived from handlers as follows. For a handler of event e within P machine state declaration l for a machine kind K, let m be a message chain and let s be the machine instance at index i, the precondition for the action $e_{j,i}$ asserts (1) $s.this = i$, (2) $s \in K$, (3) $s.state = l$, (4) $\neg s.entry$, (5) $m \in s.input_i$, and (6) $m.current \in e$. In other words, the precondition for the action $e_{j,i}$ asserts the i^{th} machine (1) is the correct target, (2) is of the correct kind, (3) is in the correct state with (4) its entry flag is set to false, and (5) is actually receiving a message chain (5) whose head is of the correct kind. For an entry handler, the precondition for the action ϵ_i asserts (1) $s.this = i$, (2) $s \in K$, (3) $s.state = l$, and (4) $s.entry$. In other words, the precondition for the action ϵ_i asserts that the i^{th} machine (1) is the correct machine, (2) is of the correct kind, (3) is in the correct state, and (4) its entry flag is set to true. We also enforce that every entry handler effect sets the entry flag to false and that every event handler that receives a message chain e removes e from $input_i$.

The effects for the actions $e_{j,i}$ and ϵ_i are given by the corresponding handler blocks (e event handlers and entry handlers). We extend the language of Fig. 5 with some syntactic sugar. First, we provide a `send` keyword that represents a procedure that takes the index of the target machine

[2] Available at https://github.com/FedericoAureliano/upverifier

Table 1. Comparison on benchmarks from Koenig et al. [2020]. First two columns are verification times, next three count invariants, last two flag the use of quantifier alternations. Asterisk free columns refer to a direct translation of the mypyvy verification without the use of message chain; #Inv* and ∃∀?* refer to an improved proof that uses message chains. The UPVerifier is faster and requires fewer quantifier alternations.

Benchmark	mypyvy	UPVerifier	# Inv M	# Inv P	# Inv P*	∃∀?	∃∀?*
ring-id	0.365s	**0.138s**	4	4	4	N	N
toy-consensus-forall	0.380s	**0.064s**	**4**	5		N	
consensus-wo-decide	0.338s	**0.072s**	**5**	8		N	
sharded-kv	0.407s	**0.045s**	5	5		N	
learning-switch	0.410s	**0.048s**	6	6		N	
consensus-forall	0.566s	**0.120s**	**7**	10		N	
lockserv	0.406s	**0.049s**	9	9		N	
ticket	0.402s	**0.068s**	14	**9**		N	
firewall	0.364s	**0.033s**	2	2	**1**	Y	**N**
sharded-kv-no-lost-keys	**0.328s**	0.589s	2	2		Y	
client-server-ae	0.323s	**0.037s**	**2**	3		Y	
toy-consensus-epr	0.392s	**0.106s**	4	4		Y	
client-server-db-ae	0.403s	**0.060s**	5	8	**4**	Y	**N**
ring-id-not-dead	0.479s	**0.194s**	6	5	**5**	Y	Y
consensus-epr	0.557s	**0.088s**	**7**	8		Y	
hybrid-reliable-broadcast	0.676s	**0.489s**	8	**5**		Y	

`t`, the payload of the message `p`, and (implicitly) the message chain that triggered the handler `h`, and adds `send(this, t, p, h)` to $output_i$. Second, we provide a `goto` keyword that represents a procedure that takes a state label and sets the state value to that label and the entry flag to true.

We make a few simplifications in the implementation that manifest in our evaluation benchmarks. First, instead of the state of the system being an array, we represent the state of the system as a record with two selectors: `events` and `machines`. The first, `events`, represents the state of the universal buffer. The second, `machines`, represents the states of all other machines. Second, instead of two buffers per machine, we collapse all buffers into the `events` selector.

6 EMPIRICAL EVALUATION

We aim to answer the following research questions. (RQ1) Is the UPVerifier expressive enough to replicate existing proofs from other systems? (RQ2) How does the performance of the UPVerifier compare to the state-of-the-art? (RQ3) Do message chains help simplify proofs? (RQ4) Can we automatically mine meaningful message chains specifications using only examples? (RQ5) Can we use the UPVerifier to verify industrial distributed systems?

6.1 RQ1 and RQ2: Baseline Comparison of Expressive Power and Performance

To evaluate the expressive power of the UPVerifier, we take verified benchmarks from Koenig et al. [2020] written in mypyvy, and re-verify them using the UPVerifier. This set of benchmarks contains consensus protocols like the ring election protocol, standard systems like sharded key-value store services, and more complicated systems, like the hybrid reliable broadcast originally described by Widder and Schmid [2007] and modeled by Berkovits et al. [2019].

The UPVerifier and mypyvy make different modeling choices and so the translation process is not always straightforward. For example, while the UPVerifier intrinsically encodes implementation details like machine kinds and message buffers, mypyvy users would need to manually model these details. This encourages mypyvy users to write proofs at higher levels of abstraction and

means that the corresponding UPVerifier proof must explicitly link the implementation details to the abstract proof. For example, for the client-server system (described in detail in Sec. 6.2.1), the mypyvy model uses three logical relations to keep track of all messages that have been sent and received so far in the system. These relations abstract the notion of a message buffer—we can infer that a message is in the buffer if it has been sent but not yet received. We use the same logical relations in the UPVerifier version of the proof but also maintain the implementation level detail of message buffers by adding one auxiliary invariant that amounts to saying that these abstract relations are append-only versions of the concrete message buffer in the system.

On the other hand, when mypyvy models do include low-level detail, translation into the UPVerifier can actually simplify the proof since these details are frequently handled by the programming model itself. For example, the "ticket" benchmark models threads as state machines with three states. The UPVerifier version of the proof encodes this directly by defining one machine kind called "Thread" which has three states. The mypyvy model defines three relations on threads, each acting as a flag to indicate when a thread is in the given state. This complicates their proof because auxiliary invariants must be added to ensure that no thread can be in more than one state at a time, a property that is tautological in the UPVerifier model.

Table 1 summarizes the data required to answer RQ1. Each row of the table corresponds to a benchmark. The first column contains the name of the benchmark in question, and the columns labeled "# Inv M" and "# Inv P" display the number of invariants used in the mypyvy and UPVerifier proofs, respectively. While in general, the number of invariants is not always indicative of the complexity of a proof, in our context, where a one-to-one translation was made whenever possible, the number of invariants gives an idea of the extra proof considerations required.

Overall, we find that the UPVerifier is comparably expressive to mypyvy and that it can successfully verify existing benchmarks. When our programming model requires extra proof considerations, we are able to address these considerations with few auxiliary invariants. In terms of RQ1, we answer that yes, *the UPVerifier is expressive enough to replicate proofs from other systems.*

To evaluate the performance of the UPVerifier, we revisit Table 1. The second column shows the wall-clock time taken by mypyvy to verify each benchmark; the third column shows the wall-clock time taken by the UPVerifier to verify the corresponding benchmark. All verification times are collected on a 2.3 GHz Quad-Core Intel Core i7 CPU with 32 GB of RAM. With the exception of the benchmark "sharded-kv-no-lost-keys" the UPVerifier is faster on every single case. Thus, for RQ2, we answer that *the UPVerifier performs comparably to mypyvy in terms of runtime*. Note that, as a sanity check, we introduced bugs into all benchmarks and attempted to verify the buggy version. No proof by induction succeeded, as expected.

6.2 RQ3: Simplification Power

To evaluate the simplification power of message chains, we take four benchmarks from related work and attempt to use message chains to improve the proofs. We then explore a new benchmark that is difficult for existing tools but is trivial in our framework. The four benchmarks from existing work, "client-server-db-ae," "firewall," "ring-id," and "ring-id-not-dead," are those in Table 1 that display message chain that are longer than three messages.

6.2.1 Client-Server-Database. The first system is a client-server system inspired by Feldman et al. [2017]. This system involves three kinds of machines: clients, servers, and databases. Clients send requests to servers which then consult a database. Once the server receives an answer from the database, it forwards the response to the original client. Feldman et al. verify that whenever a client in the client-server-database system receives a response, the same client had sent a matching request. This property is like the no-forge property of Griffin et al. [2020] (which states that no

machine "forges" a message) and requires significant machinery to state and verify. Namely, for a message-passing model without message chains, this proof requires three ghost variables, which track every request and response ever sent by the system, three invariants to ensure that the concrete message buffers are abstracted by these ghost variables, and four invariants of the form ∀∃ that state that for every response, there existed a request that "matches" that response.

Stating and verifying this no-forge property using message chain invariants, on the other hand, is easy and requires no modeling tricks: message chains provide the vocabulary that would otherwise need to be integrated into the system model manually. Specifically, the message chain-based uses four invariants. The first invariant uses the path-pattern on machine indexes $x_1x_2x_3x_2x_1[x_1 \in C \wedge x_2 \in S \wedge x_3 \in D]$, where $\Sigma \coloneqq C \cup S \cup D$, C is the set of client machine indexes, S is the set of server machine indexes, and D is the set of database machine indexes. This path-pattern captures the desired no-forge property and the invariant states that every message chain is a prefix of that path-pattern. The second invariant, states that for every message chain *mc*, `mc.message.val = mc.history.message.val`. The third invariant states that the payload of the first message in every message chain is the original client. Finally, the fourth invariant again uses a path-pattern but this time on sequences of message kinds $qqpp[]$, where q represents an *eRequest* message, p represents an *eResponse* message and $\Sigma \coloneqq \{q, p\}$. Together, these four invariants are inductive for all system instances and message interleavings.

6.2.2 Firewall. The second models a set of nodes communicating through a firewall, where the firewall blocks traffic coming from outside a network unless the traffic is in response to an internal request. In the firewall case, the property asserts that every message received by an internal node was either sent by another internal node or was sent as a response to an internal node from an external node. Like the client-server-database system, the mypyvy encoding defines abstract relations to keep track of what messages have been sent out, and then when a message is received, the encoding asserts that there exists some message in the abstract relation with the desired property. This specification is a ∀∃ properties that establishes the provenance of node interactions. To make the encoding inductive, auxiliary invariants are needed.

Using message chains and the UPVerifier we can replace the ∀∃ properties with a single message chain invariant that does not contain existential quantifiers. Specifically, the solution to the existential part of the property is given by the first element of every message chain. Again, no abstraction is needed to verify this system at the message-passing level.

6.2.3 Ring Leader Election Protocol. The final two systems from related work, "ring-id" and "ring-id-not-dead," are variants of the ring election protocol described in Sec. 1 and verified in Sec. 3.5, where "ring-id-not-dead" includes an extra goal invariant specifying that the system is never "stuck." As previously described, we use message chain invariants to verify both systems using a proof that is simple and asynchronous, all while being at a lower level of abstraction than proofs from related work, since it captures message passing directly.

6.2.4 The Onion Routing Network. Finally, we detail a new verification that is difficult for related work but easy for our framework: the Onion Routing (Tor) network. Tor is a distributed network comprised of relay nodes used to anonymize TCP-based web traffic. TCP-based web traffic is normally not anonymous because anyone in the network can view the source and destination of packets. In the Tor network, clients build a path through network nodes to a target server and then send their requests through this path. Each step of the path is encrypted on top of the previous steps of the path, making a layered structure (like an onion). When a node receives a packet, it decrypts the message (peels one layer of the onion) to determine the node it should forward the packet to. In this way, each node in the path knows the identity of its successor and predecessor, but no other

nodes. To send a response back to a client, servers follow a similar process in reverse. By scrambling network activity, Tor makes it difficult for observers to trace client-server communication.

We modeled the Tor network using three machine kinds: `Client`, `Node`, and `Server`. `Client` machines build a path to `Server` machines through some sequence of `Node` machines and then back to themselves through a second sequence of `Node` machines. Message payloads hold a `todo` variable that corresponds to the path left to traverse. When a `Node` machine receives a message, it "peels" an element of the `todo` variable, and forwards the message to the next machine in line. `Server` machines behave similarly but could also do some processing to serve the given request.

There are interesting statistical and cryptographic properties to verify for the Tor network, but we focus on basic correctness: every complete message chain should start at a client, go through some non-zero number of intermediate nodes, reach the desired server, return through some non-zero number of intermediate nodes, and end at the same client that began the interaction. To verify this property, we introduced a single ghost variable to messages, `done`, and added code to maintain it such that `concat(done, todo)` stays constant throughout every message chain. Intuitively, whenever a machine peels an element off of `todo`, it now adds it onto `done`. From there it is easy to prove that `concat(done, todo)` always has the shape defined by the target specification and that every message chain is a prefix of `concat(done, todo)`. Since this proof used interpreted sequence functions heavily (e.g., prefix and concat), we chose to represent the message chain using the SMT-LIB theory of sequences instead of algebraic data types, which would have required us to manually define these functions. Related work would have a much harder time verifying the same property of the same system at the individual message passing level of abstraction.

For RQ3 we find that *message chains can simplify proofs when the system involves communication.*

6.3 RQ4: Specification Mining

We evaluate our specification mining framework on the four unique systems from the benchmarks that display message chain of length longer than three messages because these are the hardest cases. The other benchmarks from related work are either simplified versions of the four (e.g., 'client-server-ae' is like 'client-server–db-ae' but without a database) or they do not display very complicated message passing (e.g., 'ticket' which we briefly describe in Sec. 6.1). We generate examples and create specification mining queries as described in Sec. 4. All specification mining runs terminated in less than a second.

For "ring," we are able to automatically discover the first auxiliary invariant of Prop. 3.5, and the correct, but less immediately useful fact that the suffix of message chains leading to a victory is a list of nodes from left to right, ending in the leader. For "client-server-db-ae," we are able to to automatically discover the first (target) and fourth (auxiliary) invariants described in the verification in Sec. 6.2.1; and that every message chain in the client-server-db system begins with a Client—a fact that is very close to the third invariant of the same verification. For "firewall" we are able to discover the specification that every message arriving at an internal node must have originated from an internal node; that the firewall blocks messages originating from external nodes; that blocked message chains are of length at most one; and that complete message chains are of length at least one. For "tor," we are able to mine the target invariant described in Sec. 6.2.4.

In total, this means six out of 24 invariants used in the proofs were mined. Removing the mined specifications from their respective proofs makes each of the proofs fail: these six specifications are essential. The mined specifications that do not appear in the hand written proofs may still be useful for other proofs or other applications, like testing, debugging, and documentation, but a thorough evaluation of the effectiveness for other applications is needed to draw any conclusions about them. In conclusion, for RQ4, we find that *we can learn meaningful specifications using only message chain examples, but a rigorous evaluation, especially for other applications, is required.*

6.4 RQ5: Industrial Distributed Systems

To evaluate the UPVerifier on industrial benchmarks, we conduct two case studies on verification problems provided to us by our industrial collaborators.

6.4.1 GlobalClock. The first industrial case study models and verifies a system like the ClockBound [Amazon.com, Inc. 2023] protocol, which is a mechanism for generating timestamps that are guaranteed to be within some bound of "true time." Here, "true time" is just the time of a reference machine, but when this reference machine is reliable, the protocol can be used to evaluate distributed consistency by comparing the timestamps of events.

The protocol offers an API that clients can query. When one requests a time from the system, one receives a time range $T_i = (E_i, L_i)$, where E_i corresponds to the earliest time guaranteed to have passed and L_i represents the latest time guaranteed to have not yet passed. Internally, the API consists of many systems with local clocks communicating with a corresponding global reference clock. When requesting time from a local clock, the local clock will send a query to their corresponding global clock who then responds with a "true time" that the local clocks use to return a lower and upper bound on time.

Our industrial partners have target specifications for this system. Specifically, that (1) for two time requests T_1 and T_2, if T_1 and T_2 happened on the same node and T_2 happened after T_1 then $E_2 \geq E_1$; (2) for two time requests T_1 and T_2, if T_2 happened after T_1 then $L_2 > E_1$; and (3) for two time requests T_1 and T_2, if $L_1 < E_2$ then T_1 happened before T_2.

The model and proof is 150 lines of code and takes less than two seconds to verify. We used ten auxiliary invariants, including one message chain invariant. The message chains in this system are up to five messages long and we use a message chain invariant to ensure that every local clock only ever participates in one message chain at a time. This is a powerful invariant because it blocks spurious counterexamples to induction stemming from rogue messages targeting local clocks. This is an example of a message chain invariant over multiple message chains, as described in Sec. 3.4.

6.4.2 Two-Phase Commit Multi-Version Concurrency Control. The second industrial case study is a distributed transaction commit protocol that implements multi-version concurrency control (MVCC) [Bernstein and Goodman 1983] using a two-phase commit (2PC) [Gray and Lamport 2006] protocol for agreeing on transactions. The general idea of MVCC is to allow non-blocking reads and writes by maintaining multiple versions of the data in question. Whenever a write occurs, a new version of the data is created. Reads can observe the most recent version of the data and do not need to wait for concurrent writes to terminate. The main issue is when two writes occur simultaneously. In this case, the system must agree on what writes to commit and in what order. 2PC is used to resolve this issue. Whenever a write transaction is initiated, it must broadcast a commit request to all participants. If all participants agree, then the transaction can be committed—the write can take place—and the transaction announces its success.

Our industrial partners have target specifications for this system. Specifically, (1) we only read the latest committed transactions; (2) for two read transactions T_1 and T_2, if T_1 happened before T_2 then the version read by T_1 must be earlier or equal to the version read by T_2; and (3) if a transaction announces success then all participants must have agreed to commit the corresponding write. This model and proof is 250 lines of code and takes approximately 12 minutes to verify.

The 2PC portion of this protocol is particularly interesting because it demonstrates a common phenomenon in distributed systems where an action depends on multiple concurrent messages. That is, in 2PC, coordinators require all participants to vote "yes" before telling them all to commit (and abort if any vote "no"). In terms of message chains, this means that many message chains end at the coordinator and only the message chain corresponding to the last "yes" vote or the first

"no" vote is extended. An interesting aspect of our approach is that, since it is a non-deterministic choice of which message chain is extended (in the asynchronous setting the order that votes arrive in is non-deterministic), message chain invariants reason about all of them at once.

For example, it may be surprising but, a single message chain invariant is sufficient to capture important implementation details like that the coordinator in 2PC will never send a commit message if any participant voted "no". Specifically, the message chain invariant is (in pseudocode): `c.head is commit ==> c.tail.head is vote(``yes'')`. This works because the verification engine is free to pick any of the votes as the prefix that is extended, therefore the verification only succeeds if all of the possible prefixes satisfy the invariant. The same principle applies for different thresholds (e.g., quorums).

In summary, we give an affirmative answer to RQ5: *the UPVerifier can be used to verify industrial benchmarks and that message chains play an important role.*

7 RELATED WORK

Ivy [McMillan and Padon 2020] is a synchronous reactive programming language and framework for designing, testing, and verifying distributed algorithms. Ivy supports a multitude of features, including compilation to executable code and verification through model checking and induction, all while ensuring that every query to an SMT solver belongs to a decidable fragment of first-order logic. Mypyvy [Feldman et al. 2019] is inspired by Ivy but contributes a methodology for inferring inductive invariants through lightweight user guidance. Specifically, users provide mypyvy with a system model and an automaton that describes the high-level "phases" of the system in question; mypyvy then uses the automaton to decompose the invariant inference problem, and return a safety guarantee based on a set of inferred phase invariants. QuickSilver [Jaber et al. 2020a,b] presents a framework for modeling and verifying distributed systems based on sound abstractions of complex components, and then verifying the system assuming the complex components are verified separately. IronFleet [Hawblitzel et al. 2015] is a methodology for verifying distributed system implementations that was validated through a large-scale case study of a practical, real distributed system. The UPVerifier differs in that it is domain-specific and supports verification at the message passing level of abstraction using message chains.

I4 [Ma et al. 2019] is a property-directed reachability (PDR) based approach to verifying distributed systems. I4 learns an inductive invariant over a bounded number of steps of the protocol and then uses this to infer a general inductive invariant for the infinite distributed protocol. Koenig et al. [2020] extend the PDR algorithm using first-order quantified separators. SWISS [Hance et al. 2021] and DistAI [Yao et al. 2021] tackle the same problem but with approaches that are not based on PDR. Inductive invariant synthesis (what they do) and specification mining (what we do) are two different problems. Inductive invariant synthesis is when you have a target property (something you would like to guarantee) and you want to find auxiliary invariants to make your proof by induction pass. Specification mining is when you have a system that you can observe and you want to come up with properties that likely hold. Furthermore, none of these tools are tailored to the message passing level of abstraction.

Jeppu et al. [2020] propose a program synthesis-based technique for automatically learning models from distributed system execution traces. This work follows a long line of related techniques starting in the 1970s [Biermann and Feldman 1972]. Many modern techniques in this space are based on Evidence-Driven State Merge (EDSM) [Lang et al. 1998]. Our work is complementary in that we aim to mine specifications instead of models. In that sense, our work is more similar to likely invariant synthesis techniques, like Daikon [Ernst et al. 2001]. However, we also provide a verification framework and our invariants are tailored to the distributed systems domain.

Schwarz and Mattern [1994] define a notion of causal histories that has some features in common with our notion of message chains. For a given event e, the causal history of e is the set of events that *happened-before* [Lamport 1978] e. Message chains are a focused and ordered version of causal histories. That is, the chain relation in a message chain induces a sub-relation of happened-before but the inverse is not true. See Fig. 3 for a visual comparison (e.g., event 2:1 happens before event 2:2 but the two events are not related in terms of message chains). Some of our key insights are that thread order is not necessarily important in the context of asynchronous message-passing verification, and that a lightweight but useful abstraction (message chains) that ignores thread order can be extracted from the syntax of message-passing programming languages.

The most related work is that of Talupur and Tuttle [2008] who observe that message sequence charts can be used as a proof artifact. The authors formalize the notion of *message flows*, which in our framework are akin to a sequence of actions, and integrate Message Flows into the CMP proof method [McMillan 2001]. In follow-up work [Sethi et al. 2014], the authors note that the "key limitation of our approach is that the invariants have to be derived manually by inspecting counterexamples." Our work addresses this limitation. Other than the use of specification mining, message chains differ from message flows in three big ways: expressive power, verification flexibility, and programming language integration. In terms of expressive power, message flows can only express fixed length sequences, so they cannot capture properties like those we use for the ring leader election protocol or the onion routing protocol, among others. In terms of verification flexibility, due to how they compile flows to the CMP verification method, their users can only specify a disjunction of possible flows, whereas our users can also express conjunctions and negations. Finally, we show how to use these ideas in real programming languages.

ACKNOWLEDGMENTS

We would like to thank Annamira O'Toole for help modelling and verifying the onion routing protocol. We would like to thank Adwait Godbole, Gabriel Matute, Hazem Torfah, and the anonymous reviewers for their feedback and comments. This work was supported in part by the Qualcomm Innovation Fellowship, NSF grant 1837132, DARPA contract FA8750-20-C-0156, an Amazon Research Award, summer internships at Amazon Web Services, Toyota under the iCyPhy center, and by Intel under the Scalable Assurance program.

REFERENCES

Amazon.com, Inc. 2023. Clock-Bound. https://github.com/aws/clock-bound.

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. *SIGPLAN Not.* 37, 1 (jan 2002), 4–16. https://doi.org/10.1145/565816.503275

Dana Angluin. 1980. Finding patterns common to a set of strings. *J. Comput. System Sci.* 21, 1 (1980), 46–62. https://doi.org/10.1016/0022-0000(80)90041-0

James Aspnes. 2020. Notes on theory of distributed systems. *arXiv preprint arXiv:2001.04235* (2020).

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability* (second ed.), Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). IOS Press, Chapter 33, 1267–1329. https://doi.org/10.3233/FAIA201017

Idan Berkovits, Marijana Lazić, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 245–266. https://doi.org/10.1007/978-3-030-25543-5_15

Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. 8, 4 (dec 1983), 465–483. https://doi.org/10.1145/319996.319998

Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2015. Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms. *IEEE Transactions on Software Engineering* 41, 4 (2015), 408–428. https://doi.org/10.1109/TSE.2014.2369047

A. W. Biermann and J. A. Feldman. 1972. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Trans. Comput.* C-21, 6 (1972), 592–597. https://doi.org/10.1109/TC.1972.5009015

Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. 2012. An SMT-LIB format for sequences and regular expressions. *SMT* 12 (2012), 76–86.

Maria Paola Bonacina, Pascal Fontaine, Christophe Ringeissen, and Cesare Tinelli. 2019. *Theory Combination: Beyond Equality Sharing.* Springer International Publishing, Cham, 57–89. https://doi.org/10.1007/978-3-030-22102-7_3

Ernest Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (may 1979), 281–283. https://doi.org/10.1145/359104.359108

Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (sep 2010), 1157–1210.

Joel D. Day, Vijay Ganesh, Nathan Grewal, and Florin Manea. 2023. On the Expressive Power of String Constraints. *Proc. ACM Program. Lang.* 7, POPL, Article 10 (jan 2023), 31 pages. https://doi.org/10.1145/3571203

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. 2006. Inference and Enforcement of Data Structure Consistency Specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine, USA) *(ISSTA '06)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1146238.1146266

Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-Driven Programming. *SIGPLAN Not.* 48, 6 (jun 2013), 321–332. https://doi.org/10.1145/2499370.2462184

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional Programming and Testing of Dynamic Distributed Systems. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 159 (oct 2018), 30 pages. https://doi.org/10.1145/3276529

M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123. https://doi.org/10.1109/32.908957

Yotam M. Y. Feldman, Oded Padon, Neil Immerman, Mooly Sagiv, and Sharon Shoham. 2017. Bounded Quantifier Instantiation for Checking Inductive Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–95. https://doi.org/10.1007/978-3-662-54577-5_5

Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. 2019. Inferring Inductive Invariants from Phase Structures. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 405–425. https://doi.org/10.1007/978-3-030-25543-5_23

Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Trans. Database Syst.* 31, 1 (mar 2006), 133–160. https://doi.org/10.1145/1132863.1132867

Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. 2020. TLC: Temporal Logic of Distributed Components. *Proc. ACM Program. Lang.* 4, ICFP, Article 123 (aug 2020), 30 pages. https://doi.org/10.1145/3409005

Daniel Conrad Halbert. 1984. *Programming by example.* University of California, Berkeley.

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 115–131. https://www.usenix.org/conference/nsdi21/presentation/hance

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

ITU-T. 2011. *Message Sequence Chart (MSC).* Recommendation Z.120. International Telecommunication Union.

Nouraldin Jaber, Swen Jacobs, Christopher Wagner, Milind Kulkarni, and Roopsha Samanta. 2020a. Parameterized Verification of Systems with Global Synchronization and Guards. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 299–323. https://doi.org/10.1007/978-3-030-53288-8_15

Nouraldin Jaber, Christopher Wagner, Swen Jacobs, Milind Kulkarni, and Roopsha Samanta. 2020b. Parameterized Reasoning for Distributed Systems with Consensus. *CoRR* abs/2004.04613 (2020). arXiv:2004.04613 https://arxiv.org/abs/2004.04613

Natasha Yogananda Jeppu, Thomas Melham, Daniel Kroening, and John O'Leary. 2020. Learning Concise Models from Long Execution Traces. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference* (Virtual Event, USA) *(DAC '20)*. IEEE Press, Article 92, 6 pages.

Tao Jiang, Efim Kinber, Arto Salomaa, Kai Salomaa, and Sheng Yu. 1994. Pattern languages with and without erasing. *International Journal of Computer Mathematics* 50, 3-4 (1994), 147–163.

Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. 2020. First-Order Quantified Separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*.

Association for Computing Machinery, New York, NY, USA, 703–717. https://doi.org/10.1145/3385412.3386018

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., USA.

Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. 1998. Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Grammatical Inference*, Vasant Honavar and Giora Slutzki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/BFb0054059

Gérard Le Lann. 1977. Distributed Systems-Towards a Formal Approach.. In *IFIP congress*, Vol. 7. 155–160.

Harry R. Lewis. 1980. Complexity results for classes of quantificational formulas. *J. Comput. System Sci.* 21, 3 (1980), 317–353. https://doi.org/10.1016/0022-0000(80)90027-6

Lightbend, Inc. 2022. *Akka.* https://akka.io/

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (dec 1975), 717–721. https://doi.org/10.1145/361227.361234

Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 370–384. https://doi.org/10.1145/3341301.3359651

K. L. McMillan. 2001. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Correct Hardware Design and Verification Methods*, Tiziana Margaria and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–195. https://doi.org/10.1007/3-540-44798-9_17

Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-Modal Verification Tool for Distributed Algorithms. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 190–202. https://doi.org/10.1007/978-3-030-53291-8_12

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

Fabrizio Montesi. 2014. *Choreographic programming.* IT-Universitetet i København.

Oded Padon. 2018. *Deductive Verification of Distributed Protocols in First-Order Logic.* Ph.D. Dissertation. Tel Aviv University.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. *SIGPLAN Not.* 51, 6 (jun 2016), 614–630. https://doi.org/10.1145/2980983.2908118

David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient Mutation Testing by Checking Invariant Violations. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/1572272.1572282

Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Comput.* 7, 3 (1994), 149–174. https://doi.org/10.1007/BF02277859

Divjyot Sethi, Muralidhar Talupur, and Sharad Malik. 2014. Using Flow Specifications of Parameterized Cache Coherence Protocols for Verifying Deadlock Freedom. In *Automated Technology for Verification and Analysis*, Franck Cassez and Jean-François Raskin (Eds.). Springer International Publishing, Cham, 330–347. https://doi.org/10.1007/978-3-319-11936-6_24

Ying Sheng, Yoni Zohar, Christophe Ringeissen, Jane Lange, Pascal Fontaine, and Clark Barrett. 2020. Politeness for the Theory of Algebraic Datatypes. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 238–255. https://doi.org/10.1007/978-3-030-51074-9_14

Takeshi Shinohara. 1982. Polynomial time inference of pattern languages and its applications. In *Proc. the 7th IBM Symposium on Mathematical Foundations of Computer Science.* 191–209.

Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 298–315. https://doi.org/10.1007/978-3-642-23702-7_23

Murali Talupur and Mark R. Tuttle. 2008. Going with the Flow: Parameterized Verification Using Message Flows. In *2008 Formal Methods in Computer-Aided Design.* 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.14

Telefonaktiebolaget LM Ericsson. 2022. *Erlang.* https://www.erlang.org/doc/index.html

Josef Widder and Ulrich Schmid. 2007. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Distributed Computing* 20, 2 (2007), 115–140.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 405–421. https://www.usenix.org/conference/osdi21/presentation/yao

Received 2023-04-14; accepted 2023-08-27