Learning-Enabled Verification of Distributed Systems with End-to-End Proofs

Federico Mora, University of California, Berkeley

Available at federico.morarocha.ca/quals. Navigate using A S D keys

(use Firefox and adjust size by zooming)

Distributed System Bugs are Common and Costly

> Distributed bugs... are often severe... Not only are these outages widespread and expensive, they can be caused by bugs that were deployed to production months earlier... [For example] [amazon.com] went down because one remote server couldn't display any product information.
>
> - Jacob Gabrielson, Senior Principal Engineer at Amazon Web Services

https://aws.amazon.com/builders-library/challenges-with-distributed-systems/

Proposal: Verification Methodology for Async Distributed Systems

Our collection of tools and techniques will be used to

- Verify message-passing distributed systems
  - for any number of nodes and any execution length
  - starting with asynchronous
- Without abstracting away messages
  - staying close to implementations (inside an existing programming language!)
- While giving meaningful feedback
  - generating end-to-end proofs on success
  - suggesting lemmas on failure
- With a high degree of automation
  - combining automated theorem proving and inductive reasoning

Proposal: Verification Methodology for Async Distributed Systems

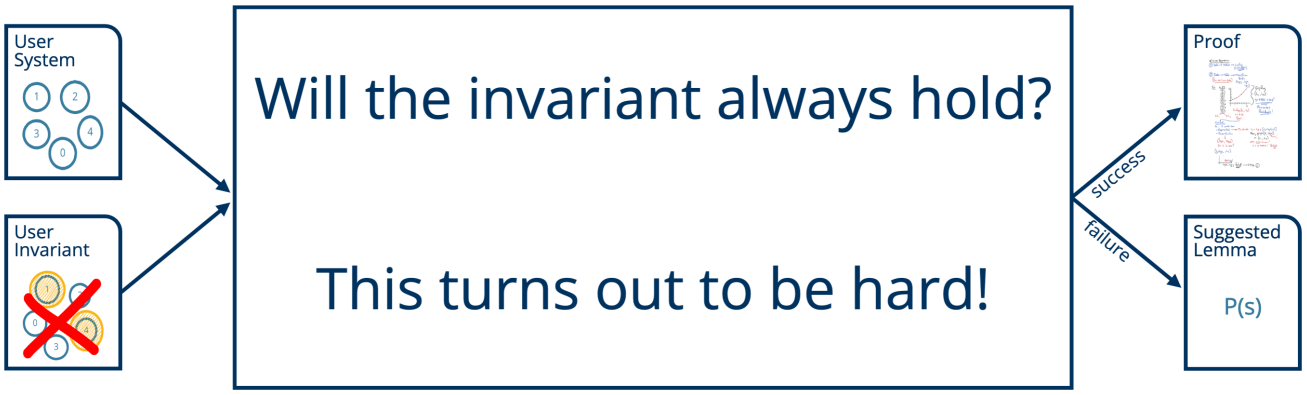


Table of Desired Features

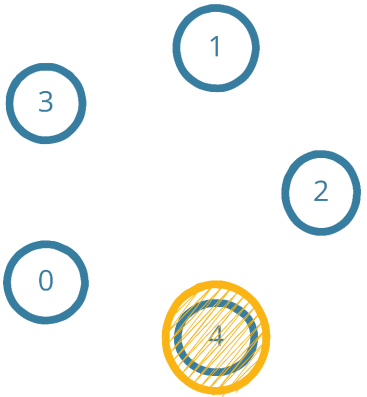
	Domain-Specific	Implementation Level	Modular Verification	High Automation	Proof Generation	Specification Mining
TLA+	✔	✘	✔	✘	✘	✘
Ivy	✔	✘	✔	✔	✘	✘
mypyvy	✔	✘	✘	✔	✘	✘
Coq/Lean/...	✘	✔	✔	✘	✔	✘
Dafny/F*/...	✘	✔	✔	✔	✘	✘
Proposal	✔	✔	✔*	✔	✔*	✔

* denotes future work; Learn More

Background

Asynchronous Systems and Verification
Reasoning About Messages
Coming Up with True Invariants
Future work: Extensions, Proofs, and Cloud Solving

Async System Example



Ring Leader Election Protocol

- Send your own identifier clockwise
- If you receive a value greater than your own, send new value clockwise
- If you receive your own identifier, declare yourself the winner

Log

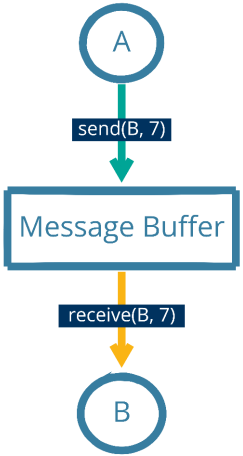
send(to, payload) | receive(from, payload) | win(node)

send(4, 2); send(0, 4); send(0, 4); receive(0, 4) and send(3, 4); send(3, 0); receive(0, 4) and send(3, 4); receive(3, 4) and send(1, 4); send(1, 3); receive(1, 4) and send(2, 4); receive(3, 4) and send(1, 4); receive(2, 4) and send(4, 4); receive(4, 4) and win(4).

Le Lann (1977); Chang and Roberts (1979); Refresh For New Execution

Async Systems, Generally

- **Nodes** are state machines with a unique identifier
- The set of node identifiers N has at least two elements
- Nodes perform atomic actions
- **Messages** are pairs (t, p), where
  - $t \in N$ represents the message target and
  - p is the message **payload**
- There is multiset of messages, called the **message buffer**
- Nodes can only read/write to their own state
- Except that every node n can (for any t and p)
  - $\text{send}(t, p)$ in the message buffer, or
  - $\text{receive}(n, p)$ from the message buffer.
    - If (n, p) is present, then $\text{receive}(n, p)$ returns p
    - Otherwise, $\text{receive}(n, p)$ returns $\varnothing$



Fischer, Lynch, and Paterson (1985); Dwork, Lynch, and Stockmeyer (1988); See Formal Definition of Ring Example

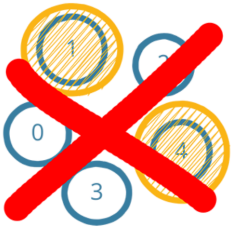
Verification Goal Example

Ring System Assumptions:

- The set of node identifiers is an **unknown** subset of the natural numbers
- Each node knows the identifier of the next "clockwise" node
- Initially, no node is in the winning state and the message buffer is empty

Ring System Specification (Verification Goal):

- There is always at most one node in the winning state



Verification Goal, Generally

- An **invariant** is a predicate over the state of the system
- We want to guarantee that an invariant will hold over every state of every possible execution

- The **state** of the system is the
  - state of every node, and
  - the state of the message buffer
- A system **step** is a pair of states (σ, σ'), where σ' is the result of a node's action on σ
- An **execution** is a sequence of states $\sigma_1 \bullet\bullet\bullet \sigma_n$ such that
  - σ_1 is an initial state, and
  - for each i, (σ_i, σ_{i+1}) is a step of the system

Lynch (1996); Fang, McMillan, Pnueli, and Zuck (2006)

Verification Technique: Proof by Induction

- An invariant I holds for every execution if
  1. Base case: $I(\sigma)$ holds for every initial state σ
  2. Inductive step: $I(\sigma)$ implies $I(\sigma')$ holds for every step (σ, σ')

- An invariant that meets condition (2) is **inductive**
- A counterexample to (2) is a **counterexample to induction**

Verification Challenges Example

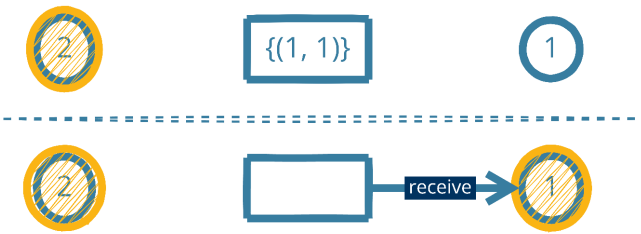
Ring System Specification (Verification Goal):

- There is always at most one node in the winning state

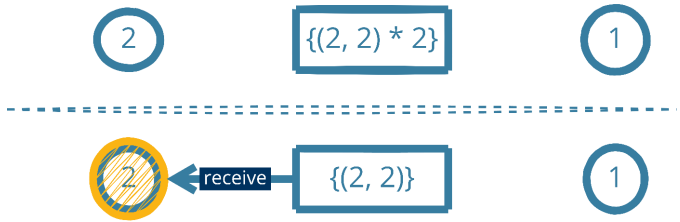
Ring System Specification (Take 2):

- There is always at most one node in the winning state, and
- ~~if a node won, then there is no message in the buffer whose target and payload are equal~~

Counterexample to Induction (Failure):



Counterexample to Induction (Take 2):



Verification Challenges, Generally

Reasoning about "stray" messages.
- Some move to a more abstract model, e.g., mypyvy's shared memory
- Some perform painstaking verification using general purpose proof assistants, e.g., IronFleet

Coming up with true invariants (let alone inductive ones).
- Most require an existing property, e.g., DistAI's auxiliary invariant synthesis

Getting solvers to terminate in a reasonable amount of time.
- Some argue for decision procedures that use doubly exponential time e.g., Ivy's EPR
- Some use hand-crafted heuristics and finite bounds as a best effort, e.g., FStar's fuel

Independently checking and sharing verification results.
- Most do not generate proofs (and have bugs), e.g., Dafny
- Most do not support sharing, so e.g., expensive sub-proofs cannot be offloaded to servers

Message Chains

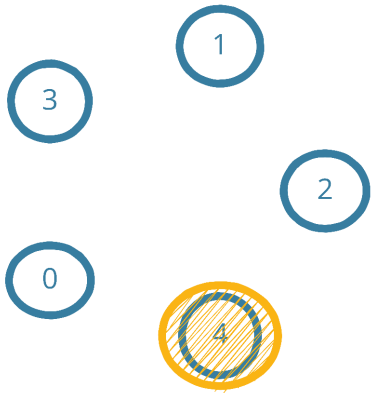
Asynchronous Systems and Verification
Reasoning About Messages
Coming Up with True Invariants
Future work: Extensions, Proofs, and Cloud Solving

Note: Message Chains are our contribution and a prototype implementation inside the P programming language is available

Message Chains Proposal Overview

- We propose **message chains**
  - an organization of messages into sequences
  - that is well-aligned with existing
    - design standards and
    - programming languages
- Users can write invariants over message chains
  - making formal verification of message passing systems easier
- Later, we describe how to automatically learn some message chain invariants

Message Chains Example and Intuitive Definition



Ring Leader Election Protocol

- Send your own identifier clockwise
  - starts a message chain
- If you receive a value greater than your own, send new value clockwise
  - extends a message chain
- If you receive your own identifier, declare yourself the winner

Unique Message Chains

source $\rightarrow^{\text{payload}}$ target

- $0 \rightarrow^{0} 3;$
- $1 \rightarrow^{1} 2;$
- $2 \rightarrow^{2} 4;$
- $3 \rightarrow^{3} 1;\ 1 \rightarrow^{3} 2;\ 2 \rightarrow^{3} 4;$
- $4 \rightarrow^{4} 0;\ 0 \rightarrow^{4} 3;\ 3 \rightarrow^{4} 1;\ 1 \rightarrow^{4} 2;\ 2 \rightarrow^{4} 4;$

Refresh For New Execution; See Formal Definition

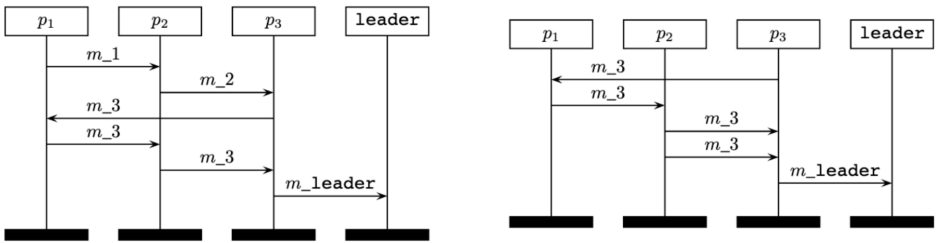
Me

- If you receive and then send in the same action, then extend the message chain.
- If you send without first receiving, then start a new message chain.

e clockwise

er

Developers Rarely Think At The Level Of Individual Messages



i) One positive and one negative scenario for the leader election protocol

- "A Message Sequence Chart... describes the message flow between instances. One Message Sequence Chart describes a partial behaviour of a system."

Bollig et al. 2010; International Telecommunication Union (ITU) Recommendation Z.120

Developers Implement Message Passing with Event Handlers

Erlang

```
receive
  pong -> io:format("received pong~n", [])
end,
```

Akka

```
def receive = {
  case "pong" => log.info("received pong")
  case _      => log.info("received unknown message")
}
```

P

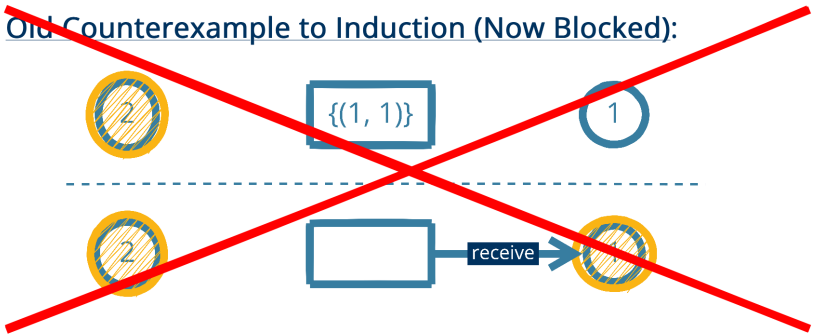
```
on ePong do {
  print "received pong";
}
```

Ring Leader Election Protocol Verification

System Specification:

- There is always at most one node in the winning state
- For every message chain **c** in the current state of the system
  - the payload at the head of **c** is the largest source label that appears in **c**
  - if a node **n** has not yet participated in **c**, then **n** lies between the first and last nodes of **c**

Old Counterexample to Induction (Now Blocked):

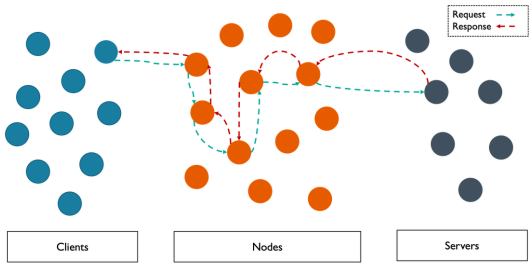


See Full Verification in P

What Other Systems Does This Work For?

Example: Onion Routing Protocol

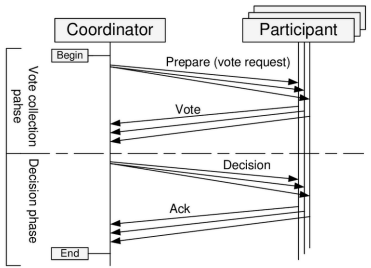
If a client receives a response, then the client must have sent a matching request.



Every message chain follows the path:
Client Node⁺ Server Node⁺ Client

Example: Two-Phase Commit

All participants agree to either commit or abort.



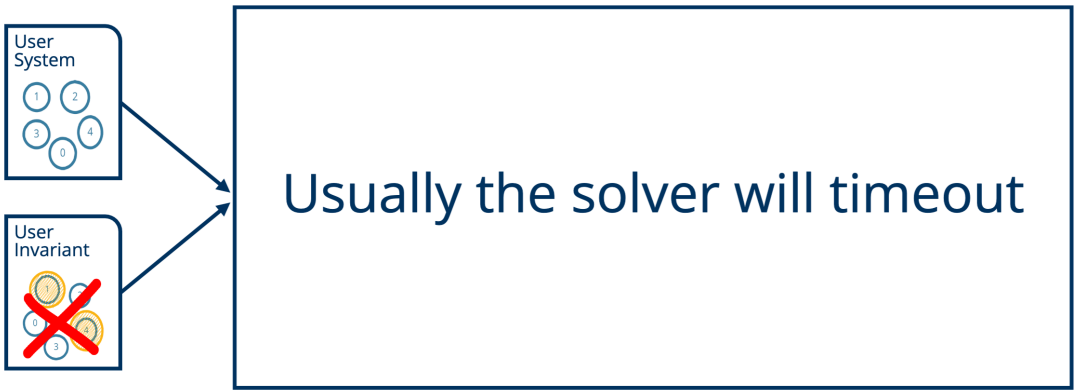
Prepare → Vote → Decision → Ack
But what happens between **Vote → Decision**?

Specification Mining

Asynchronous Systems and Verification
Reasoning About Messages
Coming Up with True Invariants
Future work: Extensions, Proofs, and Cloud Solving

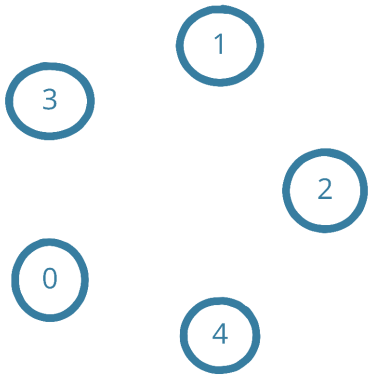
Note: Message chain invariant learning algorithms are our contribution and prototype implementations are available

What Happens When Verification Fails?



Leino and Wüstholz (2014)

Specification Mining: Can We Learn From Executions?



For every message chain c, the head payload is the largest source label that appears in c.

Algorithm

- Send your own identifier left
- If you receive a value greater than your own, send received value left
- If you receive your own identifier, declare yourself the winner

Unique Message Chains

source $\rightarrow^{\text{payload}}$ target

- $0 \rightarrow^{0} 3;$
- $1 \rightarrow^{1} 2;$
- $2 \rightarrow^{2} 4;$
- $3 \rightarrow^{3} 1;\ 1 \rightarrow^{3} 2;$
- $4 \rightarrow^{4} 0;\ 0 \rightarrow^{4} 3;\ 3 \rightarrow^{4} 1;\ 1 \rightarrow^{4} 2;\ 2 \rightarrow^{4} 4;$

Specification Mining, Generally

"... a machine learning approach to discovering formal specifications" that code *likely* obeys.

- Input: observed program executions
- Output: suggested invariant

The goal is **not** to reverse engineer the code from observations (or to discover a model).

In our setting, we are interested in mining **message chain** invariants:

- predicates that hold for every message chain that appears during the execution of a system

Ernst (2000); Ammons, Bodík, and Larus (2002); Biermann and Feldman (1972)

Non-Erasing Pattern Learning Example

- $\Sigma = \{C_1, C_2, C_3, S_1, S_2, S_3\}$
- $V = \{X, Y, Z\}$

- Input strings (message chains sampled from a client-server system):
  - $C_1 S_1 C_1$
  - $C_2 S_1 C_2$
  - $C_1 S_2 C_1$

- Angluin's output pattern: XYX

- Strings in the language of the pattern but that are kind of disappointing:
  - $S_1 S_1 S_1$
  - $C_2 C_2 S_1 C_2 C_2$
  - $C_1 S_2 C_2 S_3 C_3 S_2 C_1$

Angluin (1980); Learn More

Justified Constraint Examples

- Input strings (message chains sampled from a client-server system):
  - $C_1 S_1 C_1$
  - $C_2 S_1 C_2$
  - $C_1 S_2 C_1$
- Output pattern: $XYX \wedge |X| = 1 \wedge |Y| = 1 \wedge X \in C^+ \wedge Y \in S^+$

Unjustified Constraint Example

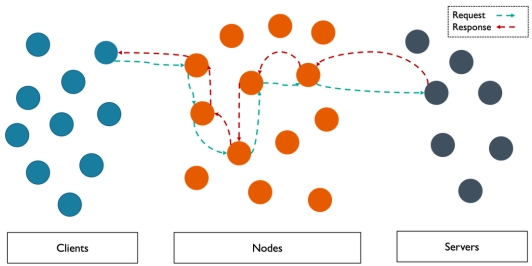
- Input strings:
  - $\mathbf{abcd}$
  - $\mathbf{dcb}$
  - $\mathbf{dc}$
- Output pattern: $\mathbf{XY}$
- Unjustified constraint (BAD): $|\mathbf{X}| = 1$
  - e.g., the assignment $\{\mathbf{X} \mapsto \mathbf{abc}, \mathbf{Y} \mapsto \mathbf{d}\}$ can *justify* $\mathbf{abcd} \in \mathbf{L(XY)}$
- Note that $\mathbf{L(XY)} = \mathbf{L(XY} \wedge |\mathbf{X}| = 1)$
  - We are adding *semantic* constraints to a *syntactic* notion

Non-Erasing Patterns with Justified Constraints, Generally

- Let S be an input set of strings and p be a learned pattern
- The set of *justifications* $J(S, p)$ is the set of assignments μ such that $\exists\, s \in S\ \mu(p) = s$
- A semantic constraint is *justified* if it does not block any justification
  - In other words, a constraint f is justified iff $\forall\, \mu \in J(S, p)\ \mu \vDash f$
- We consider two domain-specific constraint classes
  - Single machines: $|x| = 1$, where $x \in V$
  - Machine kinds: $x \in \Gamma^{+}$, where $\Gamma \subseteq \Sigma$

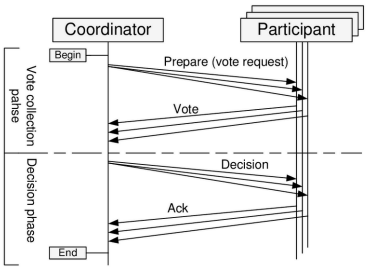
What Can Path-Patterns Learn?

Example: Onion Routing Protocol



Client Node⁺ Server Node⁺ Client
where the first and last client are the same.

Example: Two-Phase Commit



Prepare → Vote → Decision → Ack
where the three coordinator nodes are the same, but the participants before and after the decision may be different.

Learning Even More With Negative Examples

- Positive message chain examples are easy to get
  - Sample by fuzzing the target system
- Negative examples are harder
  - We *should* prove that a message chain can never happen

- PBE tools can synthesize predicates from positive **and** negative examples
- Assuming users can provide negative examples, PBE tools can suggest invariants
- Smyth and Burst, with less than 10 examples, can learn (among other things)
  - For every message chain `c`, the head payload is the largest source label that appears in `c`

Lubin et al. (2020); Miltner et al. (2022)

Limitations and Future Work

Asynchronous Systems and Verification
Reasoning About Messages
Coming Up with True Invariants
Extensions, Proofs, and Cloud Solving

Challenges Revisited

- Reasoning about messages
  - Message chains help but
    - they do not remove the limitations of asynchrony
    - they do not yet support modular reasoning

Extensions

- Coming up with true invariants
  - Path pattern learning helps but captures only one class of invariants
  - Programming-by-example helps but requires negative examples
- Independently checking verification results

Proofs

- Getting solvers to terminate

Cloud Solving

Extensions: Partial Synchrony

- Let Δ be the max message delivery time
- Let Φ be the max difference in node computation speeds
- In the asynchronous setting, no Δ or Φ exist
  - our setting so far
  - "there is no consensus protocol resilient to even one fail-stop fault"
- In the synchronous setting, Δ and Φ exist and are known
- In the partially synchronous setting, Δ and Φ exist but are not known
- What kinds of proofs can message chains help with in the partially synchronous setting?

Dwork and Lynch (1988)

Extensions: Modular Verification

- Decompose complex systems and reason about them in isolation
- Usually through assume-guarantee reasoning

- P supports modular testing
- Can message chains play a role in modular verification?
  - e.g., can we define sub-chains and chains of sub-chains?
  - would that help simplify proofs or improve specification mining?

Meyer (1992); Abadi and Lamport (1995); Desai (2020)

Proofs: Independently Checking Certificates

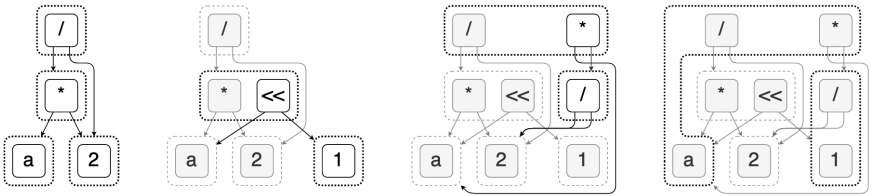
Table 1: Summary of bugs found.

	Verifier		Compiler	Other	Total
	Snd.	Prec.			
Verifier Testing	0	8	1	1	10
Compiler Testing	2	0	18	0	20
Other	1	0	0	0	1
Total	3	8	19	1	31
Confirmed	3	8	19	1	31
Fixed	0	0	7	1	8

- Modern, state-of-the-art automated verification tools, like Dafny, have bugs.
- So do SMT solvers, which are used under the hood.
- Users should not "have to rely on the correctness of a large and frequently-changing code base."

Irfan et al. (2022); Blotsky et al. (2018); Barrett, de Moura, and Fontaine (2015)

Proofs: E-Graphs to the Rescue



- E-graphs are a data structure for representing and rewriting terms
- You can extract proofs from e-graphs
- Usually input and output terms are supposed to be in the same language
  - e.g., when optimizing programs
- But can we use e-graphs as a proof generating compiler?
  - Yes! At least for some pairs of languages (which ones?)
  - But you cannot just extract the smallest term anymore

Willsey et al. (2021); Flatt et al. (2022)

Cloud Solving: Solvers and Selection

- Different SMT solvers have different performance profiles
  - No one solver dominates all others
- Especially apparent with the emergence of cloud solvers
  - Dispatching to the cloud may take longer than solving locally
  - But may be worth it!
- Can we use e-graphs to implement proof-producing, efficient, cloud solvers?
- Can we use solver selection techniques to decide when to go to the cloud (and what solver to use)?

Mora et al. 2021; Pimpalkhare et al. 2021; https://www.amazon.science/blog/automated-reasonings-scientific-frontiers

Learning-Enabled Verification with End-to-End Proofs

- Modular verification
  - verifying one distributed system
  - decomposed into multiple, simpler queries
- Where some queries are solved in the cloud
  - automatically selecting solver and platform per query
- With end-to-end proofs
  - verification encodings produce proofs
  - solvers produce proofs
  - proofs are stitched together and returned

Table of Desired Features, Revisited

	Domain-Specific	Implementation Level	Modular Verification	High Automation	Proof Generation	Specification Mining
TLA+	✔	✘	✔	✘	✘	✘
Ivy	✔	✘	✔	✔	✘	✘
mypyvy	✔	✘	✘	✔	✘	✘
Coq/Lean/...	✘	✔	✔	✘	✔	✘
Dafny/F*/...	✘	✔	✔	✔	✘	✘
Proposal	✔	✔	✔*	✔	✔*	✔

* denotes future work; Learn More

Timeline

- Message Chains for Asynchronous Verification
  - April 2023
    - submit to SOSP
- Learning Invariants using Message Chain Examples
  - May 2023
    - submit to FMCAD
- Modular Message Chains and Partially Synchronous Systems
  - September 2023 to December 2023
    - formalize modular message chains
    - encode partial synchrony in our framework
  - January 2024 to March 2024
    - challenge: verify one system per week of 294-234
  - ~April 2024
    - submit to SOSP or similar
- SMT Solver and Platform Selection: When and How to Cloud
  - March 2023 to September 2023
    - proof-of-concept implementation
      - e-graph-based eager SMT solving in the cloud
      - algorithm selection for when to go to the cloud
  - January 2024
    - submit to CAV
- Proof-Producing Verification Encodings with Learned Rewrites
  - ~May 2024 to December 2024
    - combine previous projects
  - January 2025
    - submit to CAV
- Write thesis: January 2025 to May 2025
- Graduation: May 2025