Lean as a Multi-Modal Meta Verifier

Ilya Sergey

ilyasergey.net

joint work with
George Pîrlea, **Vladimir Gladshtein**, **Qiyuan Zhao**, **Yueyang Feng**, **Dipesh Kafle**,
Elad Kinsbruner (Technion) and **Vitaly Kurin** (NUP)

Part I:
Lean as a Multi-Modal Verifier for Distributed Protocols

Distributed Protocols

Define how multiple parties (nodes)
collaborate with each other
to achieve a common goal 🤝

Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

Operating Systems — R. Stockton Gaines, Editor

An Optimal Algorithm for Mutual Exclusion in Computer Networks

Glenn Ricart
National Institutes of Health

Ashok K. Agrawala
University of Maryland

Minister

Anne, are you prepared to commit to this relationship?

Henry

Anne

Two-Phase Commit depiction from Leslie Lamport's video course on TLA+

Distributed Protocols

Some nodes might *fail* 💥

Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki
Barbara H. Liskov

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

There Is More Consensus in Egalitarian Parliaments

Iulian Moraru, David G. Andersen, Michael Kaminsky
Carnegie Mellon University and Intel Labs

In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout, *Stanford U*

https://www.usenix.org/conference/atc14/technical-sessions/pres

Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering

Jialin Li Ellis Michael Naveen Kr. Sharma Adriana Szekeres Dan R. K. Ports
University of Washington
{lijl, emichael, naveenks, aaasz, drkp}@cs.washington.edu

Distributed Protocols

Some nodes might behave *maliciously* 😈

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov

The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus

DAVID MAZIÈRES, Stellar Development Foundation

HotStuff: BFT Consensus with Linearity and Responsiveness

Maofan Yin, Cornell University, VMware Research
Dahlia Malkhi, VMware Research
Michael K. Reiter, UNC-Chapel Hill
Guy Golan Gueta, VMware Research
Ittai Abraham, VMware Research

All You Need is DAG

Eleftherios Kokoris-Kogias
IST Austria and Novi Research

Alexander Spiegelman
Novi Research

Bullshark: DAG BFT Protocols Made Practical

Alexander Spiegelman
sasha.spiegelman@gmail.com

Neil Giridharan
giridhn@berkeley.edu

Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback

Rati Gelashvili, *Novi Research*
Lefteris Kokoris-Kogias, *Novi Research & IST Austria*
Alberto Sonnino, *Novi Research*
Alexander Spiegelman, *Novi Research*
Zhuolun Xiang*, *University of Illinois at Urbana-Champaign*

A Secure Sharding Protocol For Open Blockchains

Loi Luu, National University of Singapore, loiluu@comp.nus.edu.sg
Viswesh Narayanan, National University of Singapore, visweshn@comp.nus.edu.sg
Chaodong Zheng, National University of Singapore, chaodong.zheng@comp.nus.edu.sg
Kunal Baweja, National University of Singapore, bawejaku@comp.nus.edu.sg
Seth Gilbert, National University of Singapore, seth.gilbert@comp.nus.edu.sg
Prateek Saxena, National University of Singapore, prateeks@comp.nus.edu.sg

Verifying Distributed Protocols

Google “Errors found in distributed protocols”
github.com/dranov/protocol-bugs-list

- Safety properties: “Bad thing will not happen.”
- Liveness properties: “Good thing will eventually happen.”

Safety bugs **can** be reliably discovered with conventional black-box **testing**.

Frameworks for Verifying Distributed Protocols

Proving the Correctness of Disk Paxos in Isabelle/HOL

2005

IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Ivy: Safety Verification by Interactive Generalization

Oded Padon
Tel Aviv University, Israel

Kenneth L. McMillan
Microsoft Research, USA

Aurojit Panda
UC Berkeley, USA

Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

Programming and Proving with Distributed Protocols

ILYA SERGEY, University College London, UK
JAMES R. WILCOX, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq *

Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems

Grove: a Separation-Logic Library for Verifying Distributed Systems

, Amin Timany*, Marit Edna Ohlenbusch,

LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs

LONGFEI QIU, Yale University, USA
YOONSEUNG KIM, Yale University, USA
JI-YONG SHIN, Northeastern University, USA
JIEUNG KIM, Inha University, South Korea
WOLF HONORÉ*, Yale University, USA
ZHONG SHAO, Yale University, USA

Compositional Verification of Composite Byzantine Protocols

Qiyuan Zhao
National University of Singapore
Singapore, Singapore
qiyuanz@comp.nus.edu.sg

George Pîrlea
National University of Singapore
Singapore, Singapore
gpirlea@comp.nus.edu.sg

Karolina Grzeszkiewicz
Yale-NUS College
Singapore, Singapore
karolina.grzeszkiewicz@u.yale-nus.edu.sg

Seth Gilbert
National University of Singapore
Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

Ilya Sergey
National University of Singapore
Singapore, Singapore
ilya@nus.edu.sg

Frameworks for Verifying Distributed Protocols

Proving the Correctness of Disk Paxos in Isabelle/HOL

2005

IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Verdi: A Framework for I
Formally Verifying Distr

Programming and Proving with

ILYA SERGEY, University College London, UK
JAMES R. WILCOX, University of Washington, USA
ZACHARY TATLOCK, University of Washington, U

Grove: a Separation-Logic Lib

LiDO: Linearizable Byzantine Distri
Refinement-Based Liveness Proofs

LONGFEI QIU, Yale University, USA
YOONSEUNG KIM, Yale University, USA
JI-YONG SHIN, Northeastern University, USA
JIEUNG KIM, Inha University, South Korea
WOLF HONORÉ*, Yale University, USA
ZHONG SHAO, Yale University, USA

tion by Interactive Generalization

Kenneth L. McMillan
Microsoft Research, USA

Aurojit Panda
UC Berkeley, USA

tine Fault-Tolerant Protocols
wered by Coq *

anised Logic for Modular
out Distributed Systems

Amin Timany*, Marit Edna Ohlenbusch,

of Composite Byzantine Protocols

Qiyuan Zhao
National University of Singapore
Singapore, Singapore
qiyuanz@comp.nus.edu.sg

George Pîrlea
National University of Singapore
Singapore, Singapore
gpirlea@comp.nus.edu.sg

Karolina Grzeszkiewicz
Yale-NUS College
Singapore, Singapore
karolina.grzeszkiewicz@u.yale-nus.edu.sg

Seth Gilbert
National University of Singapore
Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

Ilya Sergey
National University of Singapore
Singapore, Singapore
ilya@nus.edu.sg

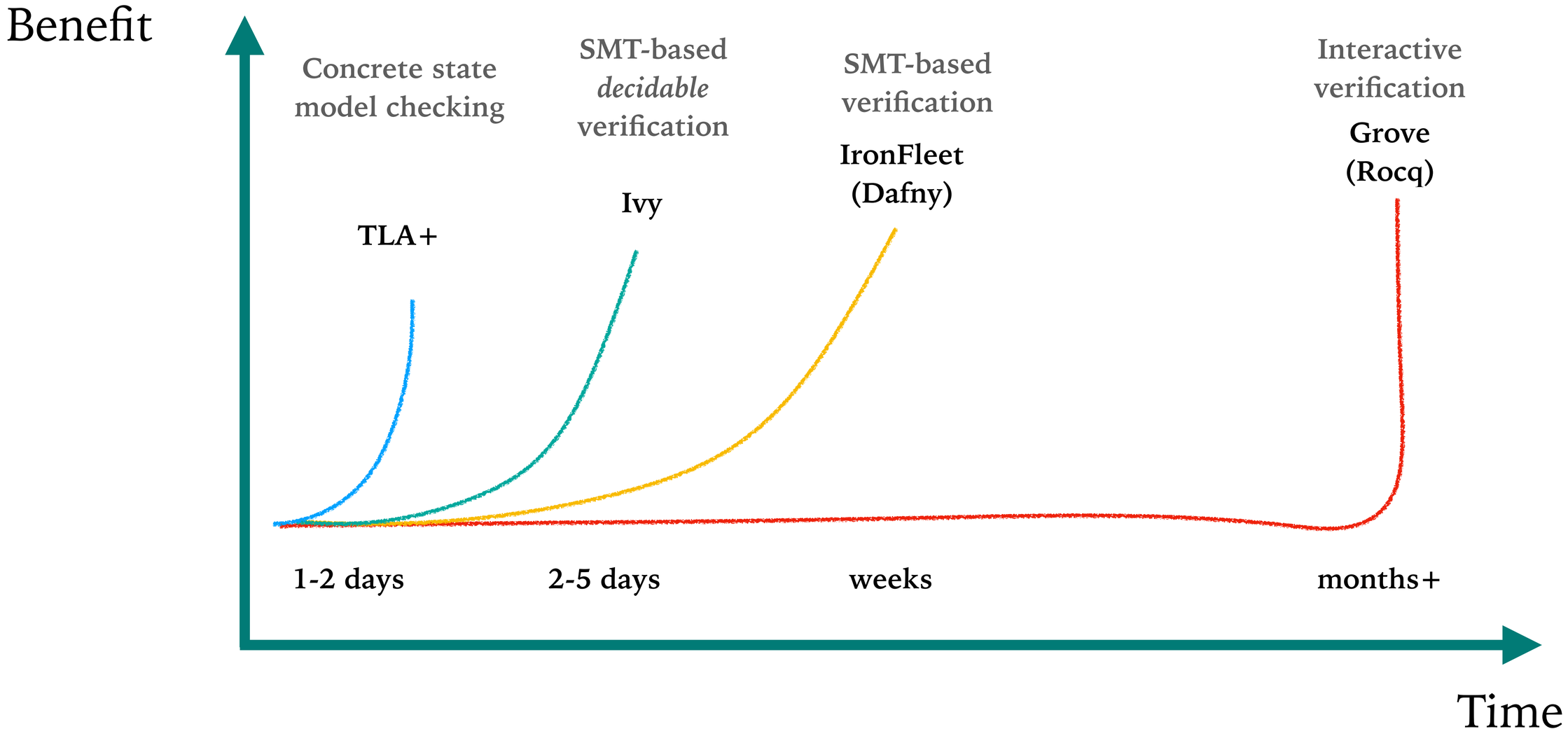
Fast Feedback for Protocol Design

protocol & properties

✅

or counterexample

Approaches for Verifying Distributed Protocols



When Your Tool is Not Sufficient

You either:

- use a **combination** of tools, or

- add an **interactive escape hatch** to your automated tool

Pîrlea's Law

Every verifier expands until it contains an ad hoc, bug-ridden, unusable implementation of half of an interactive theorem prover.

Those verifiers which cannot so expand are replaced by ones which can.

see also:
Greenspun's tenth rule

Lessons from Building an Auto-Active Verifier in Lean

George Pîrlea
National University of Singapore
Singapore
gpirlea@comp.nus.edu.sg

Vladimir Gladshtein
National University of Singapore
Singapore
vgladsht@comp.nus.edu.sg

Qiyuan Zhao
National University of Singapore
Singapore
qiyuanz@comp.nus.edu.sg

Ilya Sergey
National University of Singapore
Singapore
ilya@nus.edu.sg

Veil

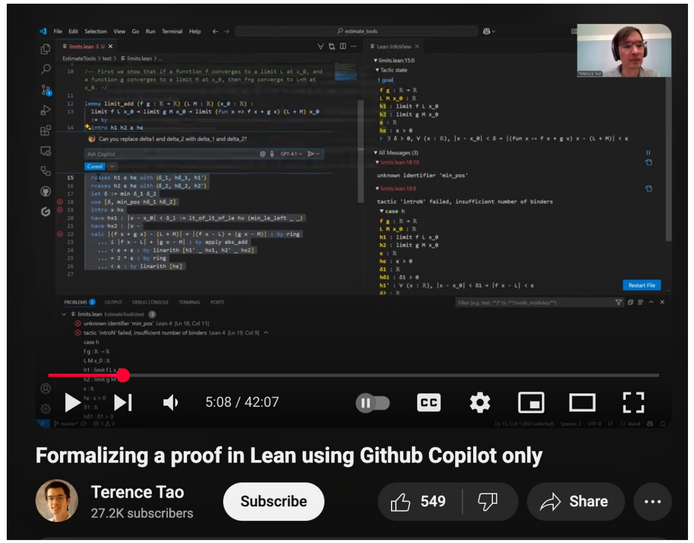
We just built the whole verifier in Lean!

What is Lean

- Lean is a **programming language** and **proof assistant**
- Lean and its tooling are **implemented in Lean**. Lean is very extensible.
- Batteries included: LSP, Parser, Macro System, Type Checker, Tactic Framework, Proof automation, Compiler, Build System, Documentation Authoring Tool.
- Lean has a **small trusted core**, proofs can be exported and independently checked.
- The Lean FRO is a nonprofit dedicated to developing Lean.

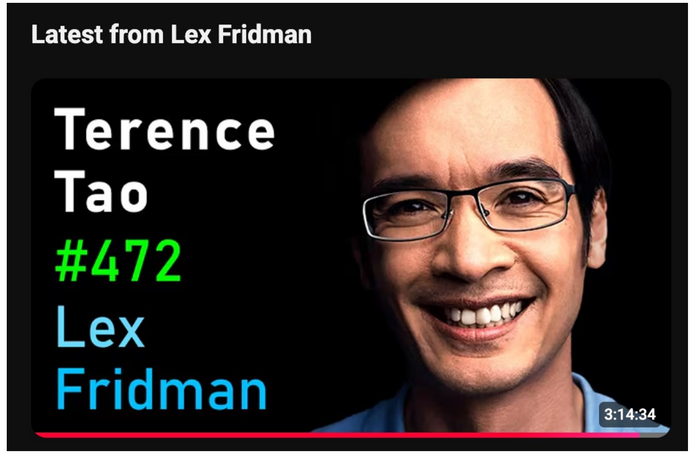
Lean is Taking Mathematics by Storm

*"**Lean enables large-scale collaboration** by allowing mathematicians to break down complex proofs into smaller, verifiable components. This formalization process ensures the correctness of proofs and facilitates contributions from a broader community. **With Lean, we are beginning to see how AI can accelerate the formalization of mathematics, opening up new possibilities for research.**" — Terence Tao*



Fermat's Last Theorem – Kevin Buzzard

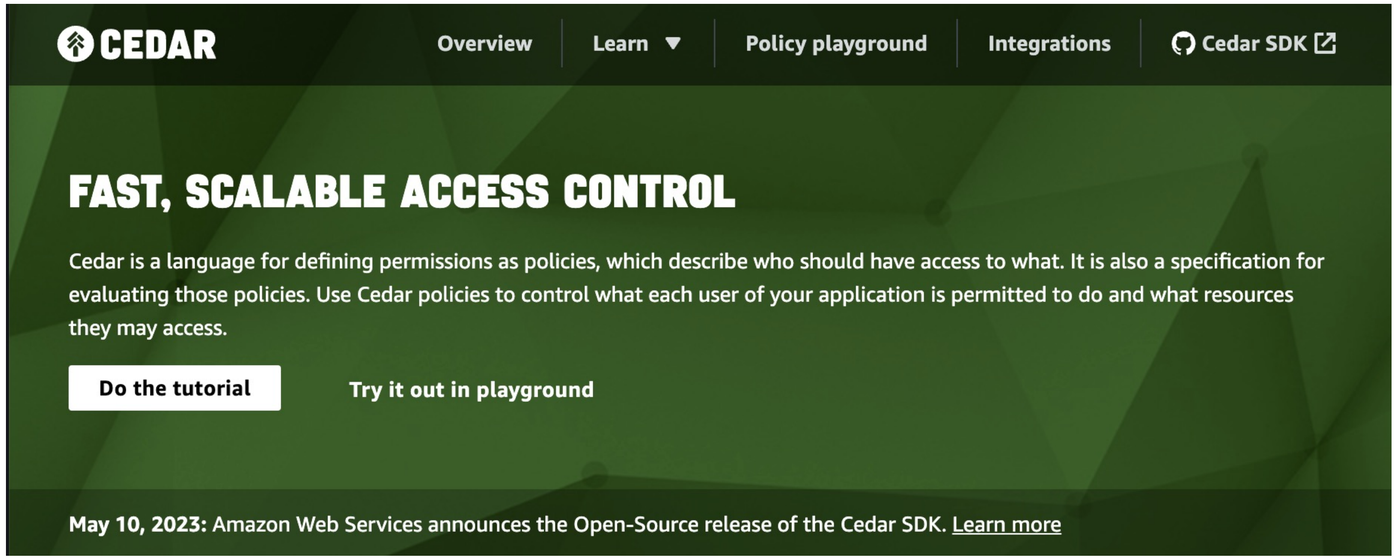
Carleson's Theorem – Floris van Doorn



slide credit: Leo de Moura

Cedar

https://www.cedarpolicy.com/

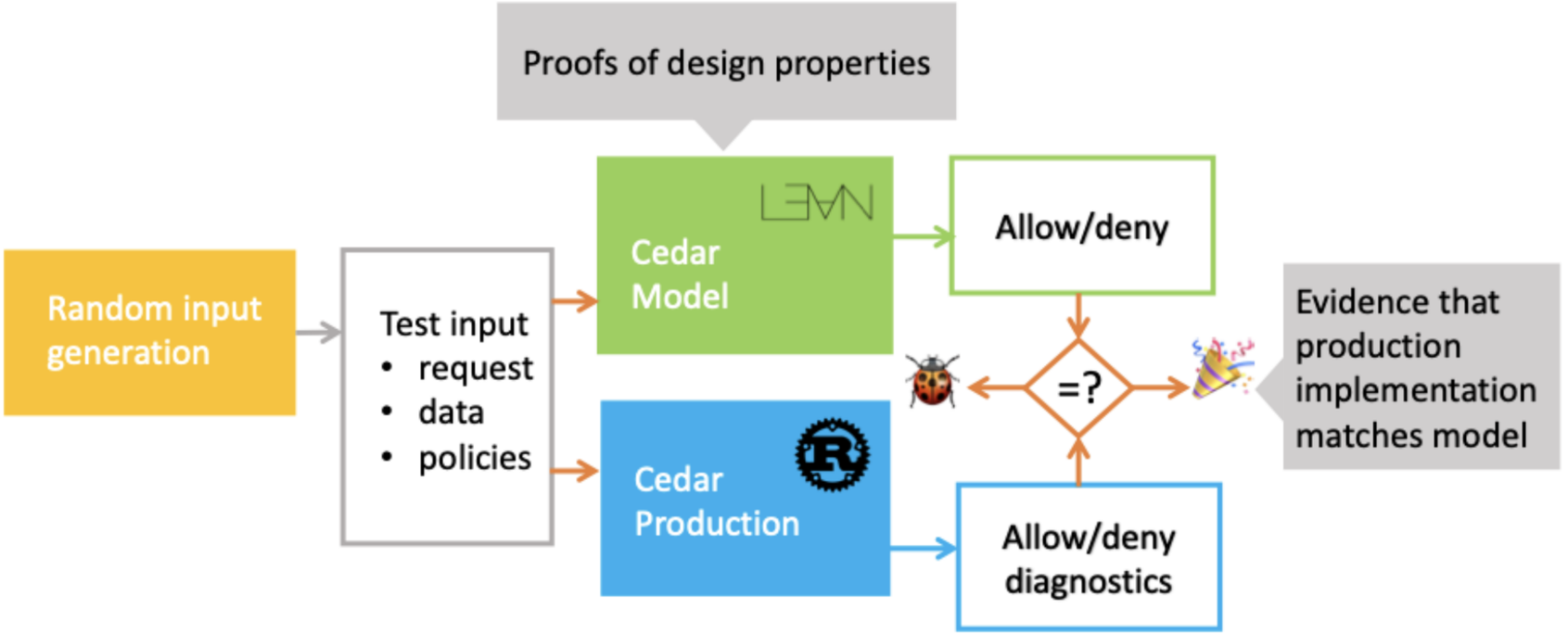


https://github.com/cedar-policy/cedar-spec

```
def isAuthorized (req : Request) (entities : Entities) (policies : Policies) : Response :=
  let forbids := satisfiedPolicies .forbid policies req entities
  let permits := satisfiedPolicies .permit policies req entities
  let erroringPolicies := errorPolicies policies req entities
  if forbids.isEmpty && !permits.isEmpty
  then { decision := .allow, determiningPolicies := permits, erroringPolicies }
  else { decision := .deny,  determiningPolicies := forbids, erroringPolicies }
```


slide credit: Leo de Moura

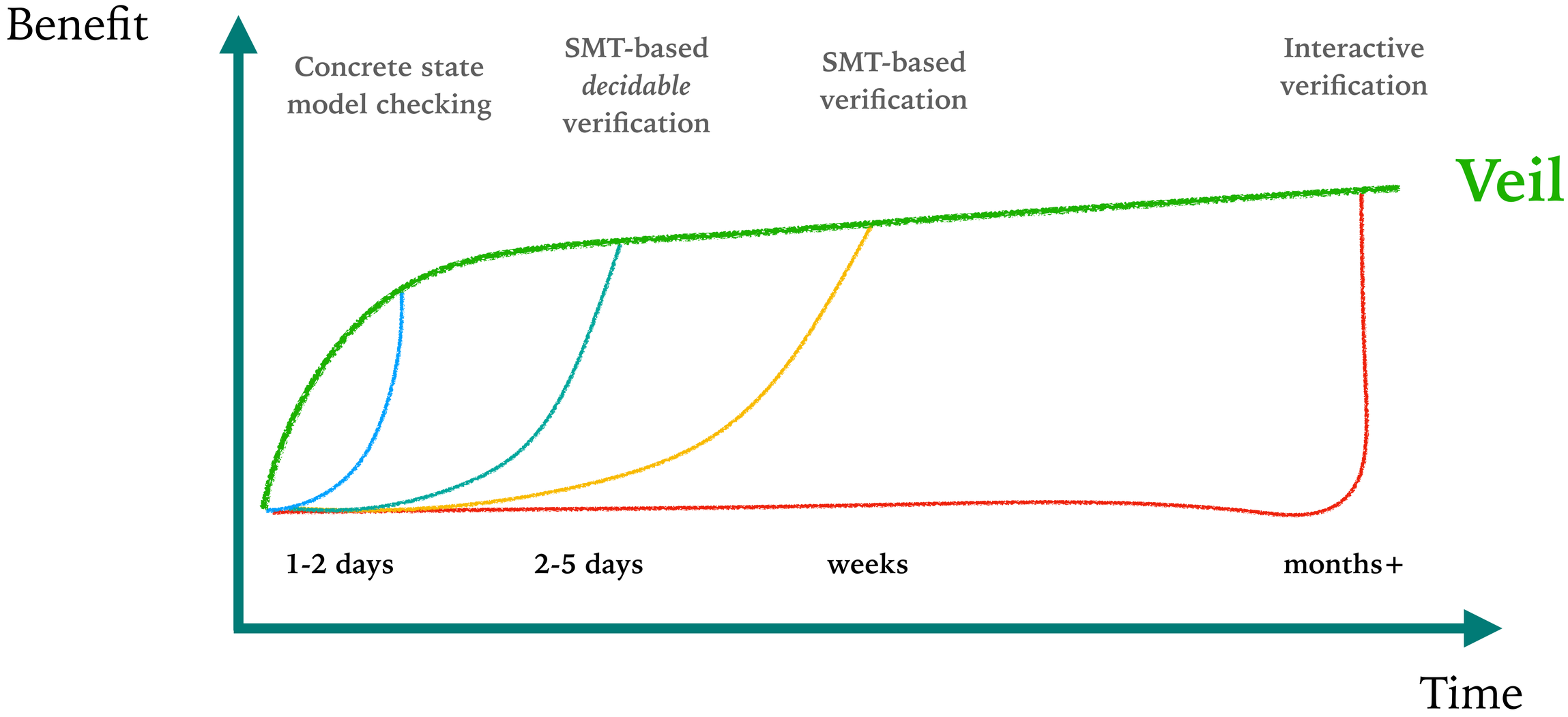
Cedar



*"**Lean is the core verification technology behind Cedar**, the open-source authorization language that powers cloud services like Amazon Verified Permissions and AWS Verified Access. Our team rigorously formalizes and verifies core components of Cedar using Lean's proof assistant, and we leverage **Lean's lightning-fast runtime** to continuously test our production Rust code against the Lean formalization. Lean's efficiency, extensive libraries, and vibrant community **enable us to develop and maintain Cedar at scale**, while ensuring the key correctness and security properties that our users depend on." — Emina Torlak, Senior Principal Applied Scientist, AWS*

slide credit: Leo de Moura

The Future of Verifying Distributed Protocols



Veil in Action

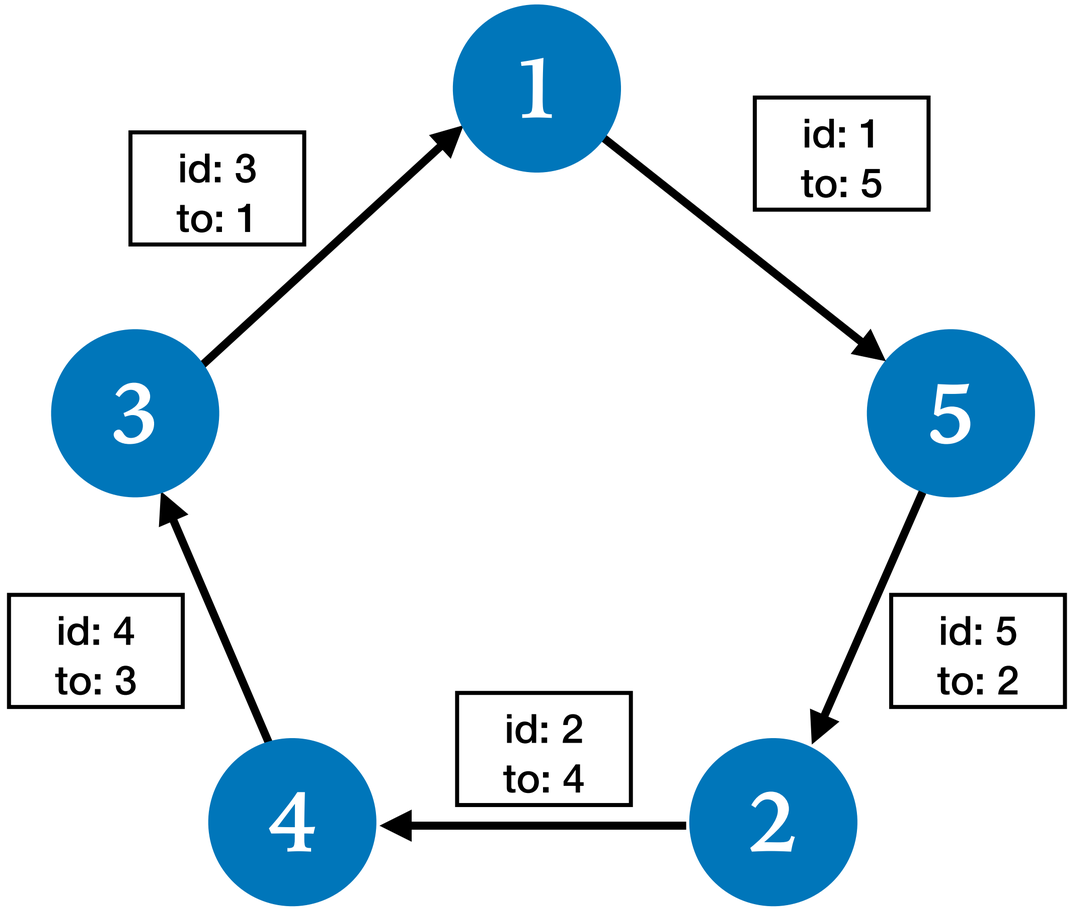
Repository: github.com/verse-lab/veil

Branch **leanlang-demo**

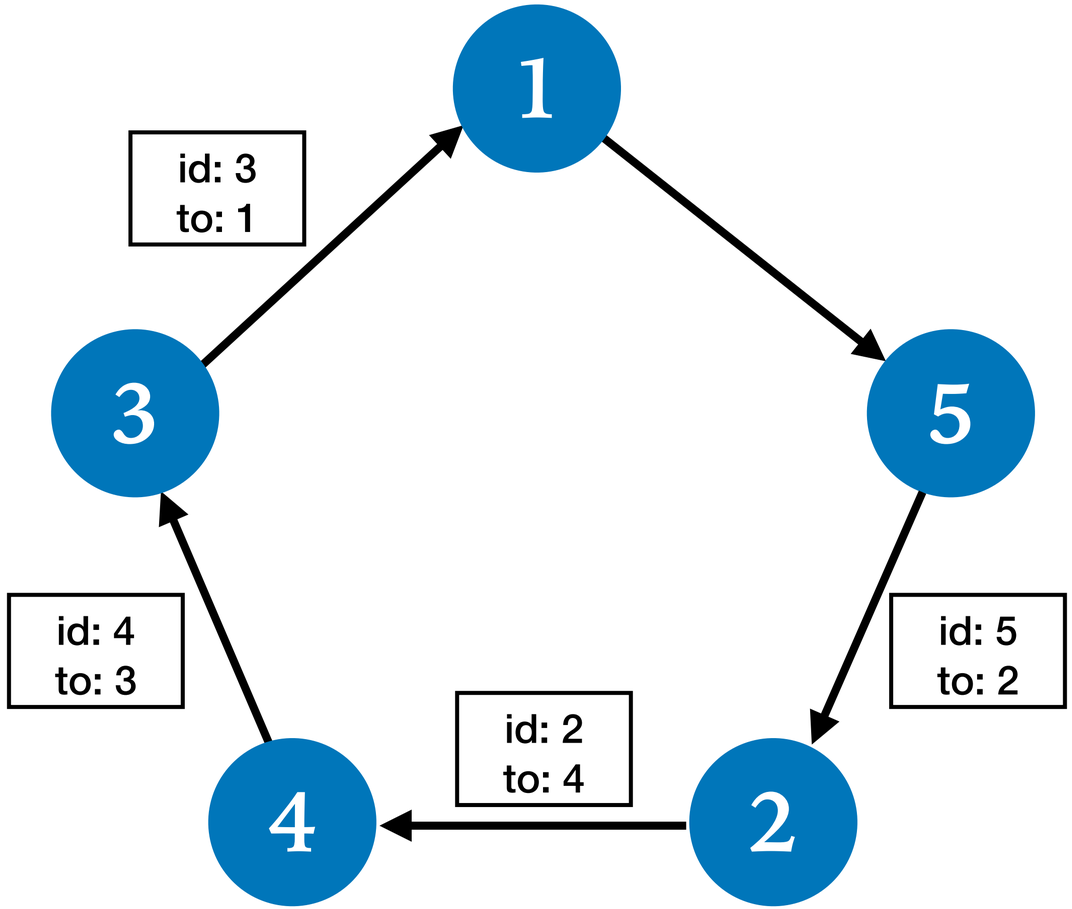
```
lake exe cache get; lake build
```

Leader Election in a Ring

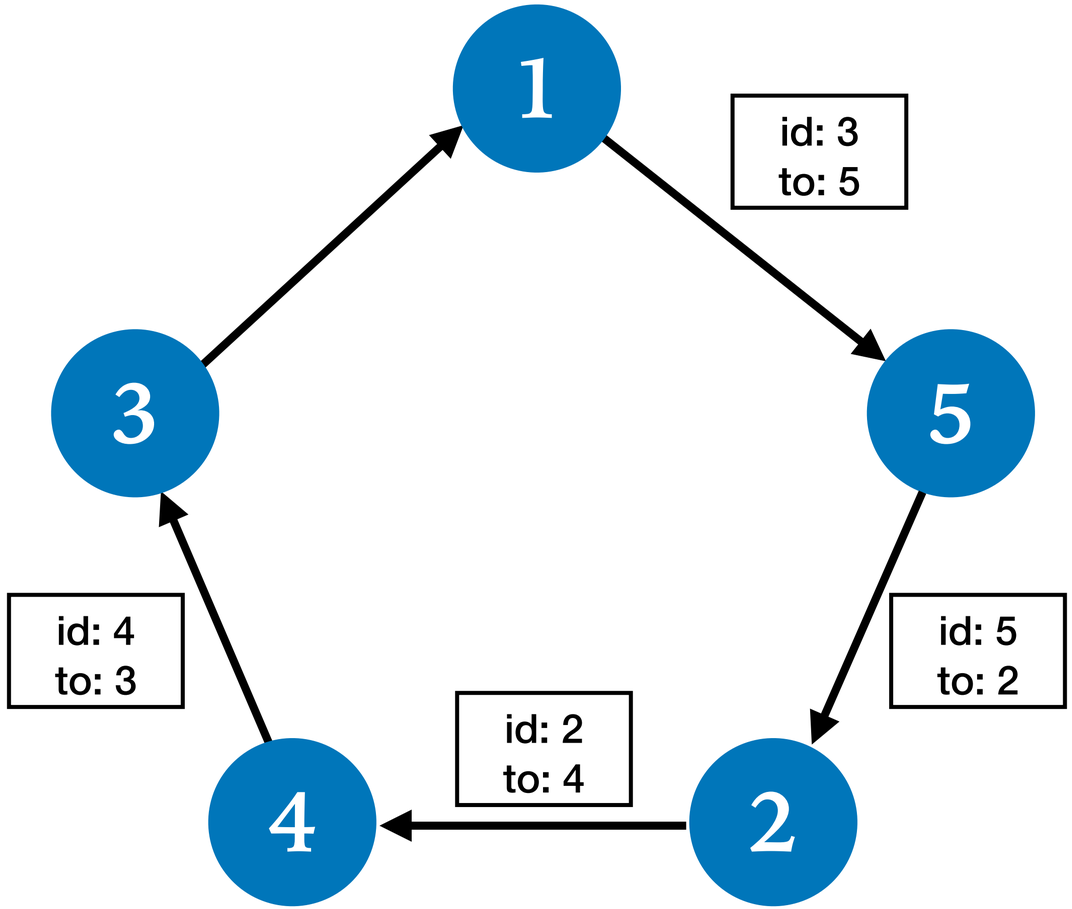
Ring Leader Election



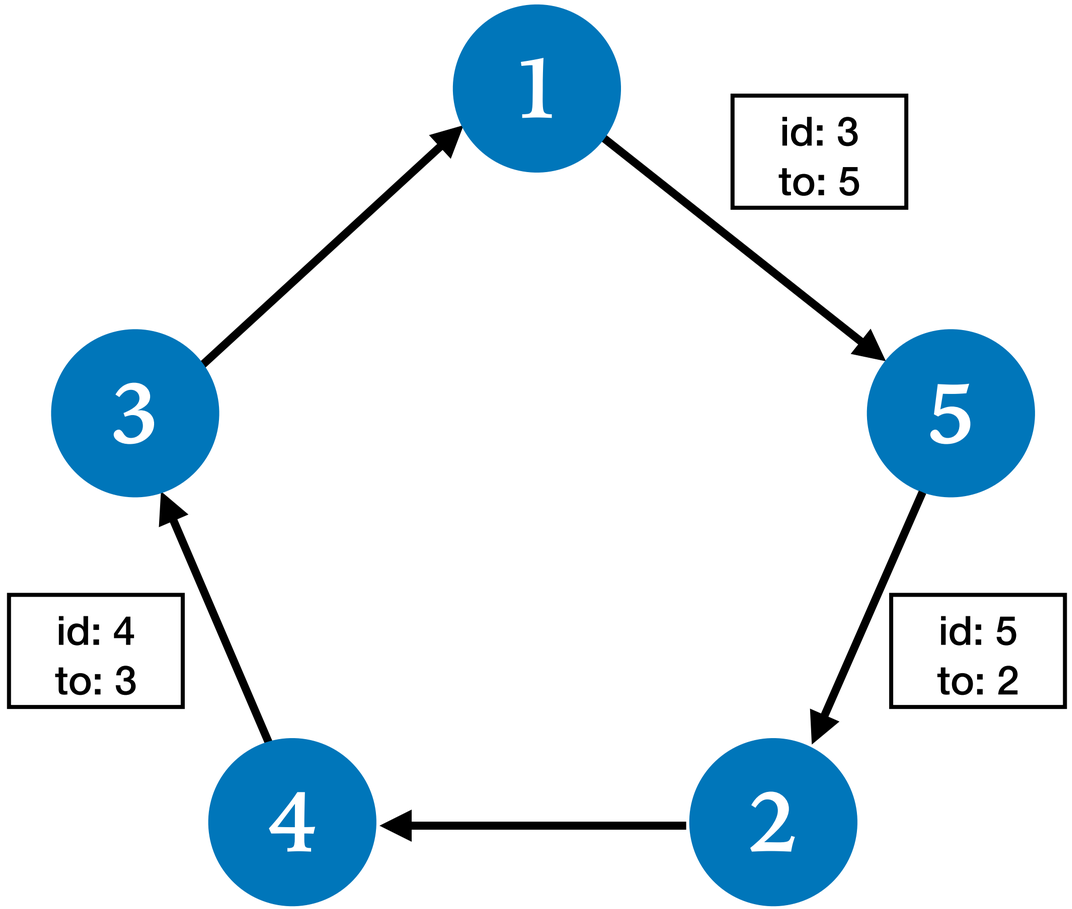
Ring Leader Election



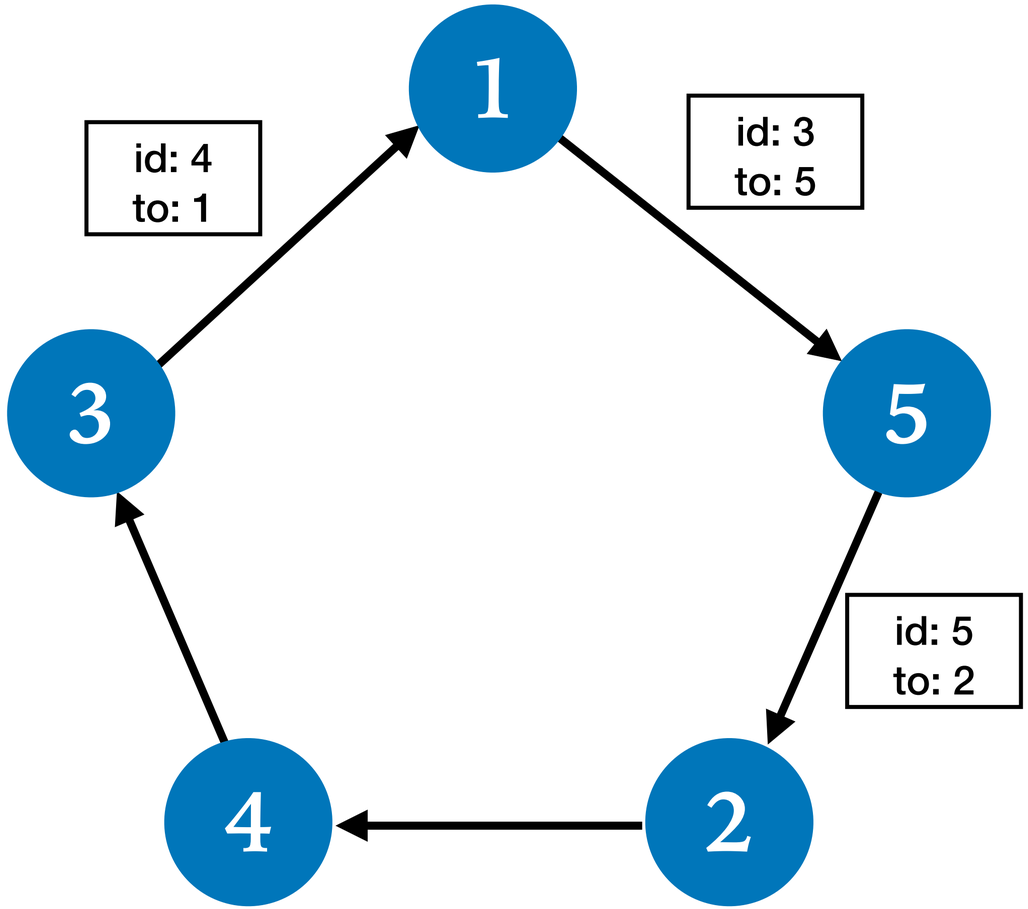
Ring Leader Election



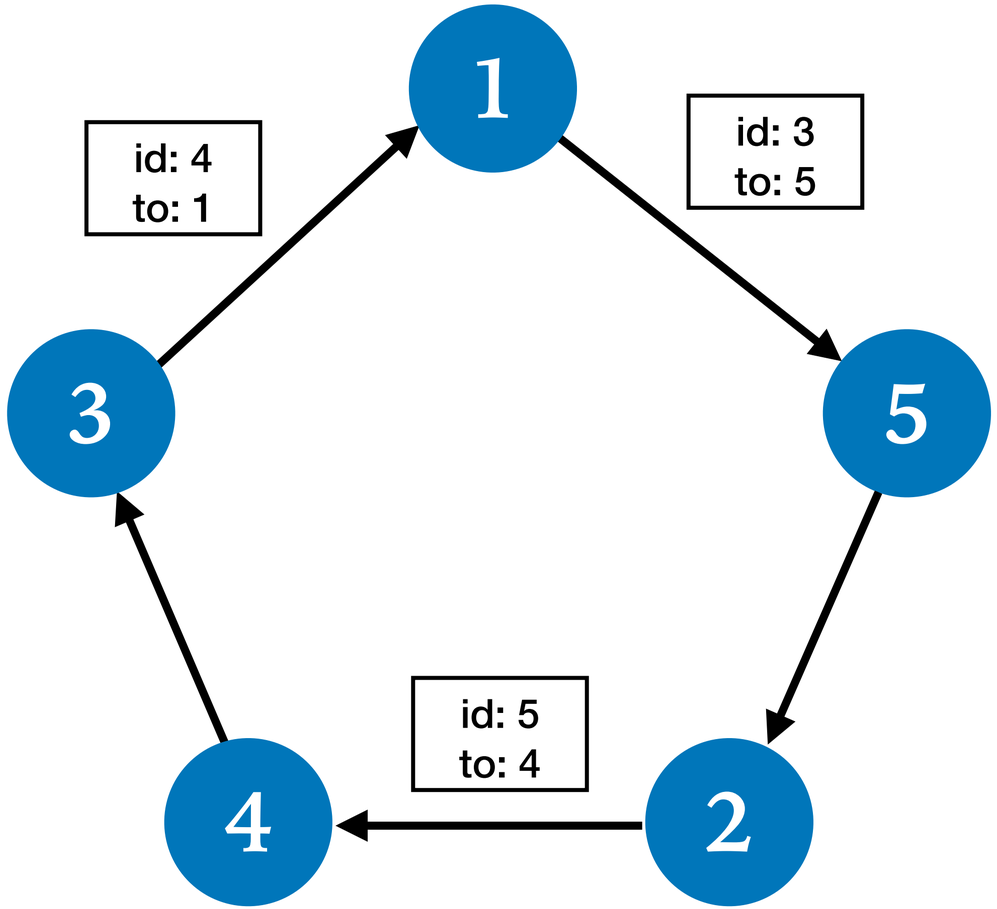
Ring Leader Election



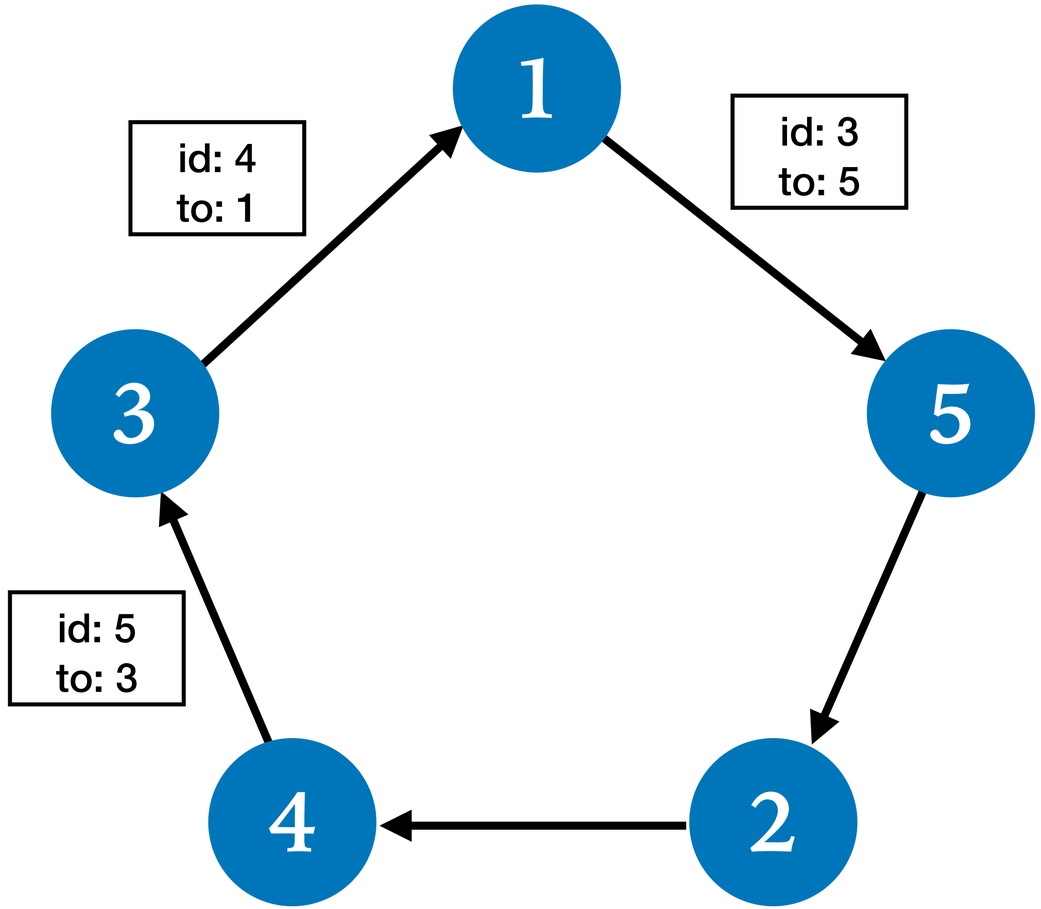
Ring Leader Election



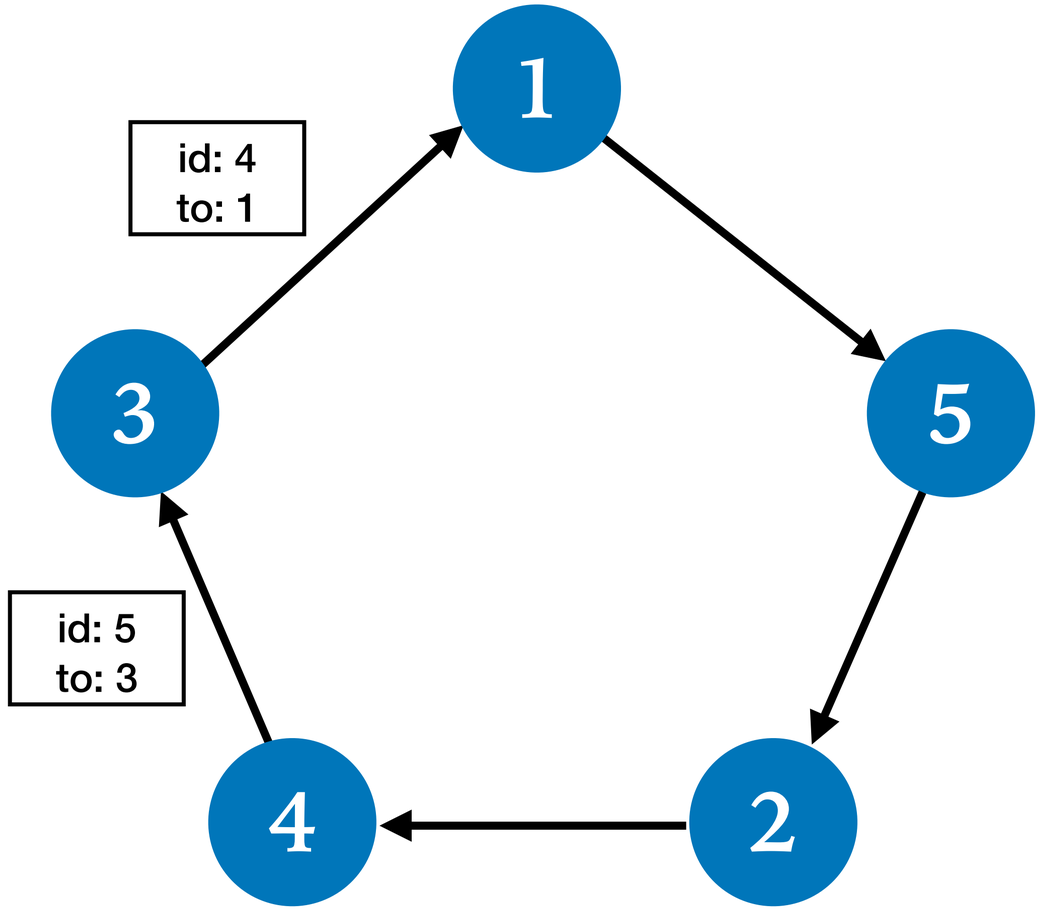
Ring Leader Election



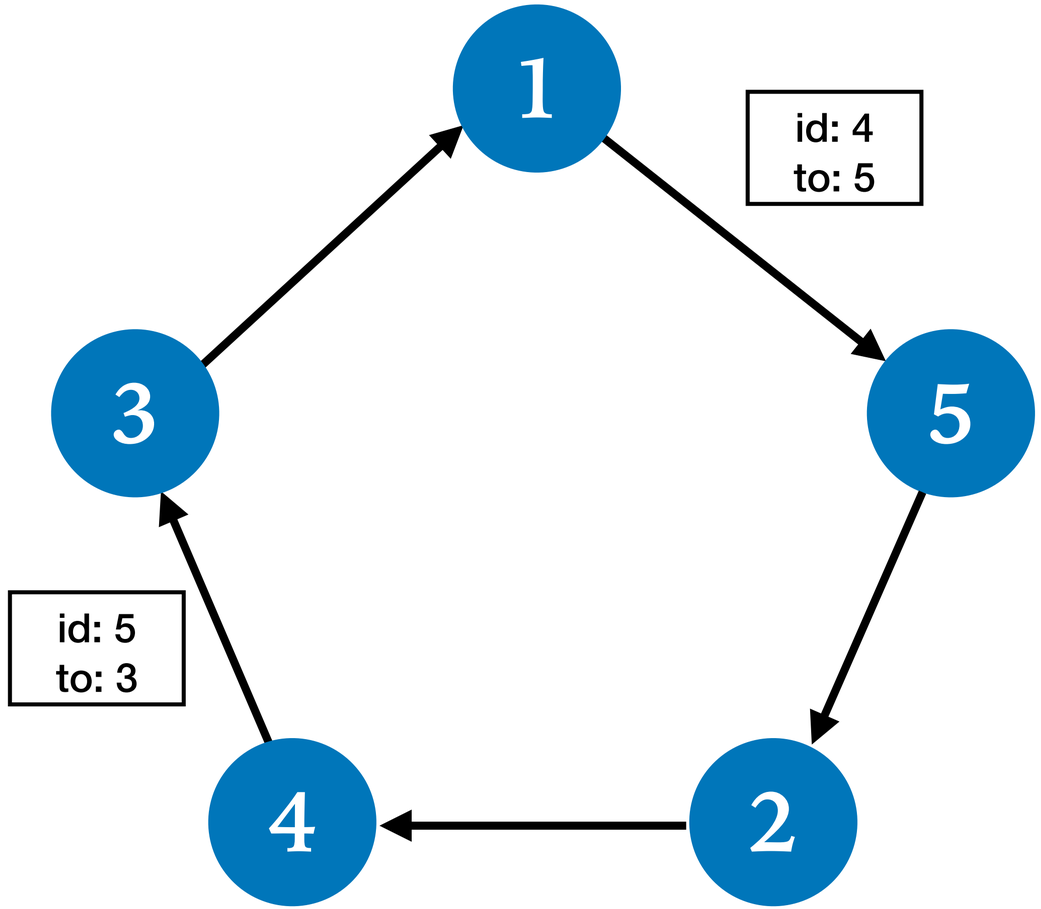
Ring Leader Election



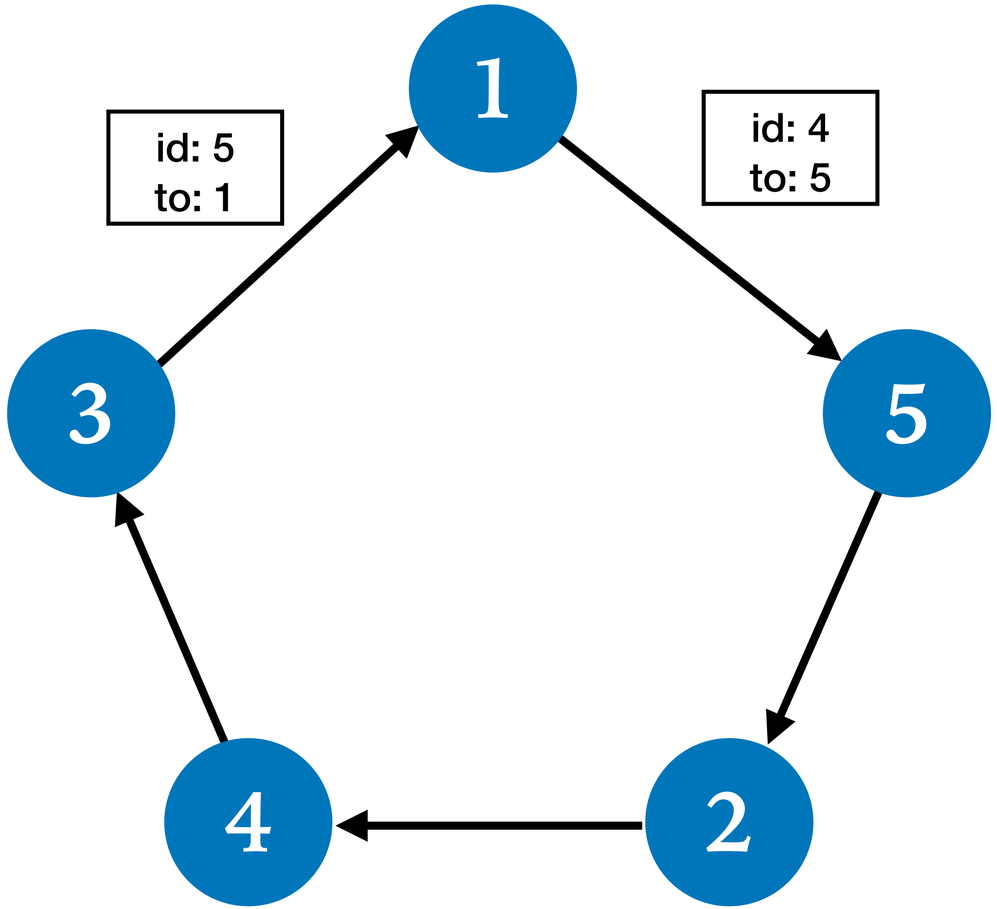
Ring Leader Election



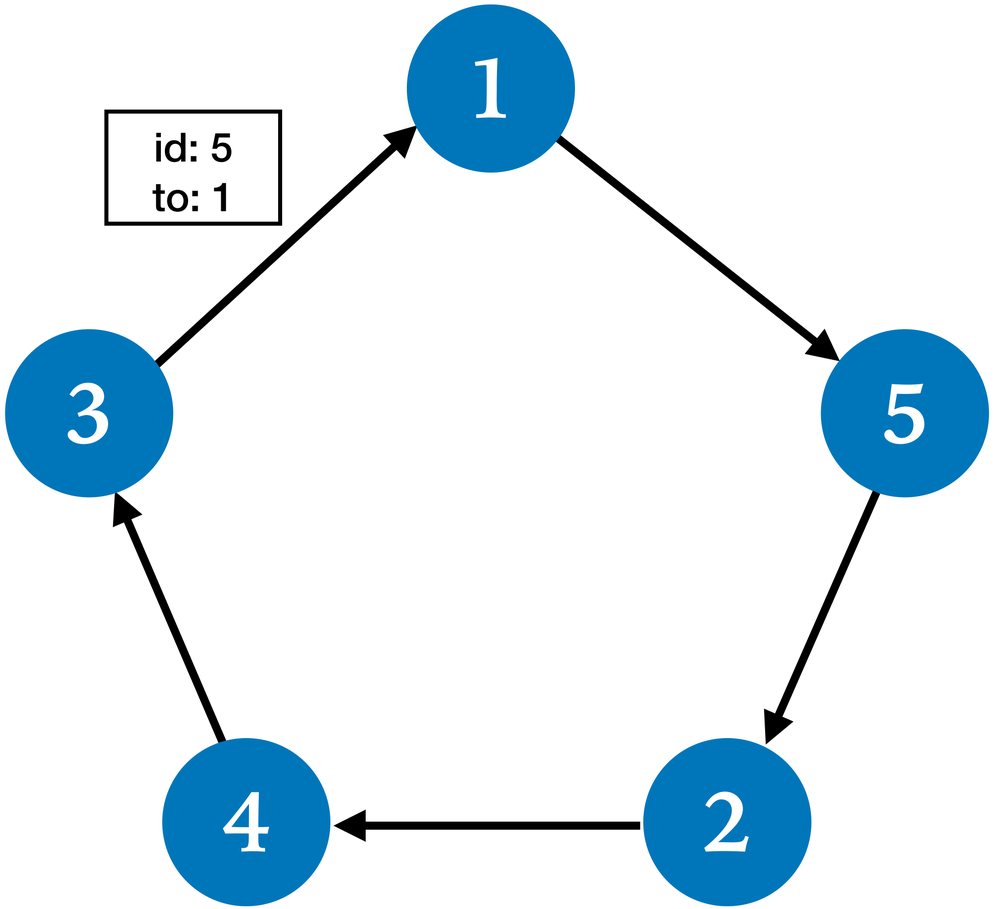
Ring Leader Election



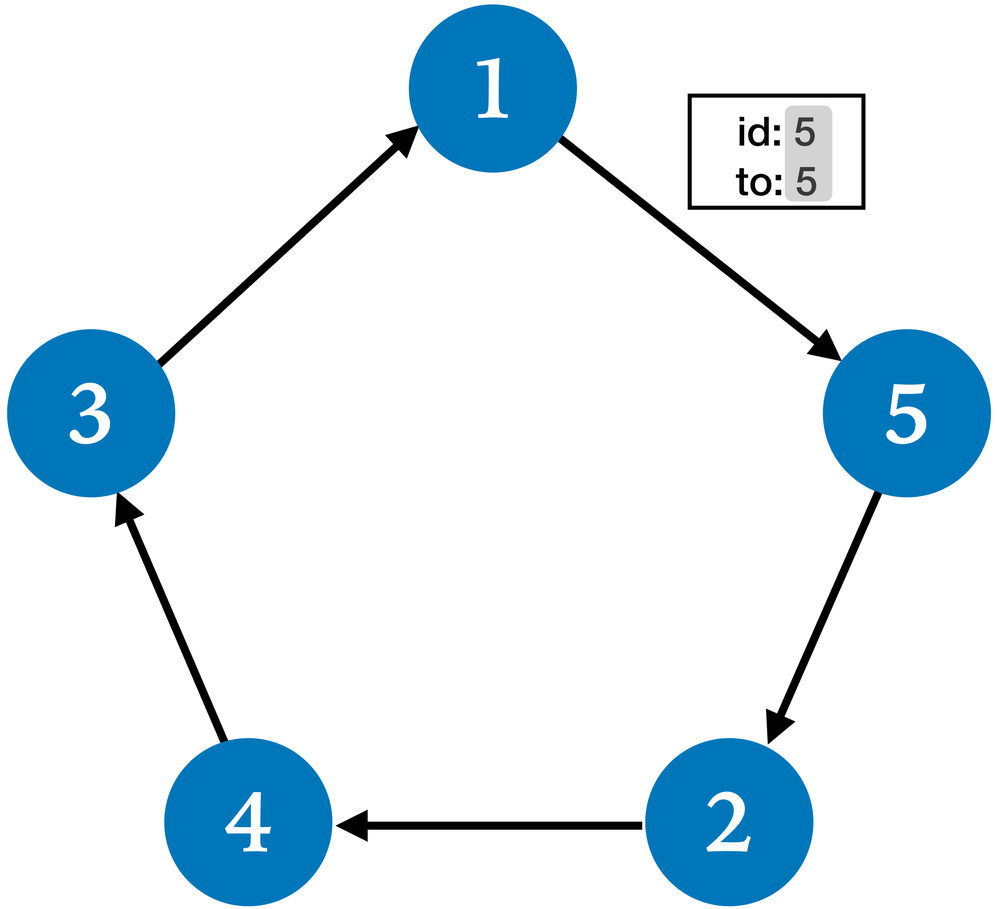
Ring Leader Election



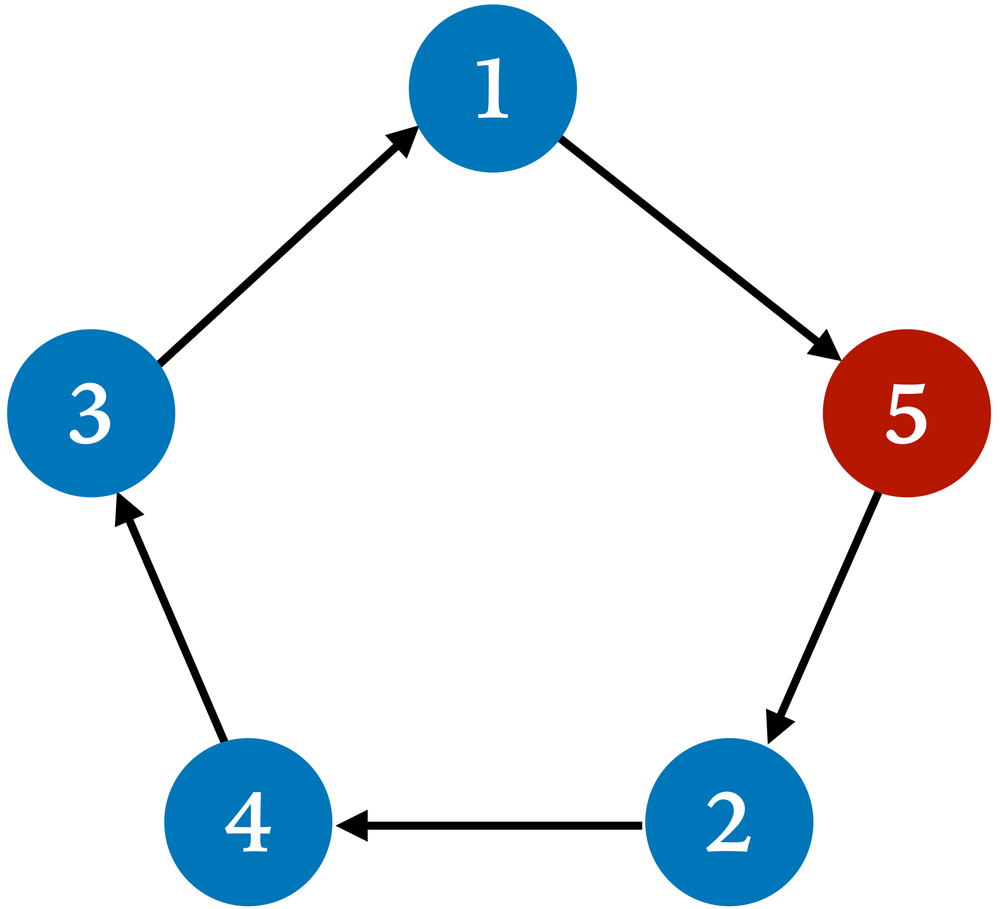
Ring Leader Election



Ring Leader Election

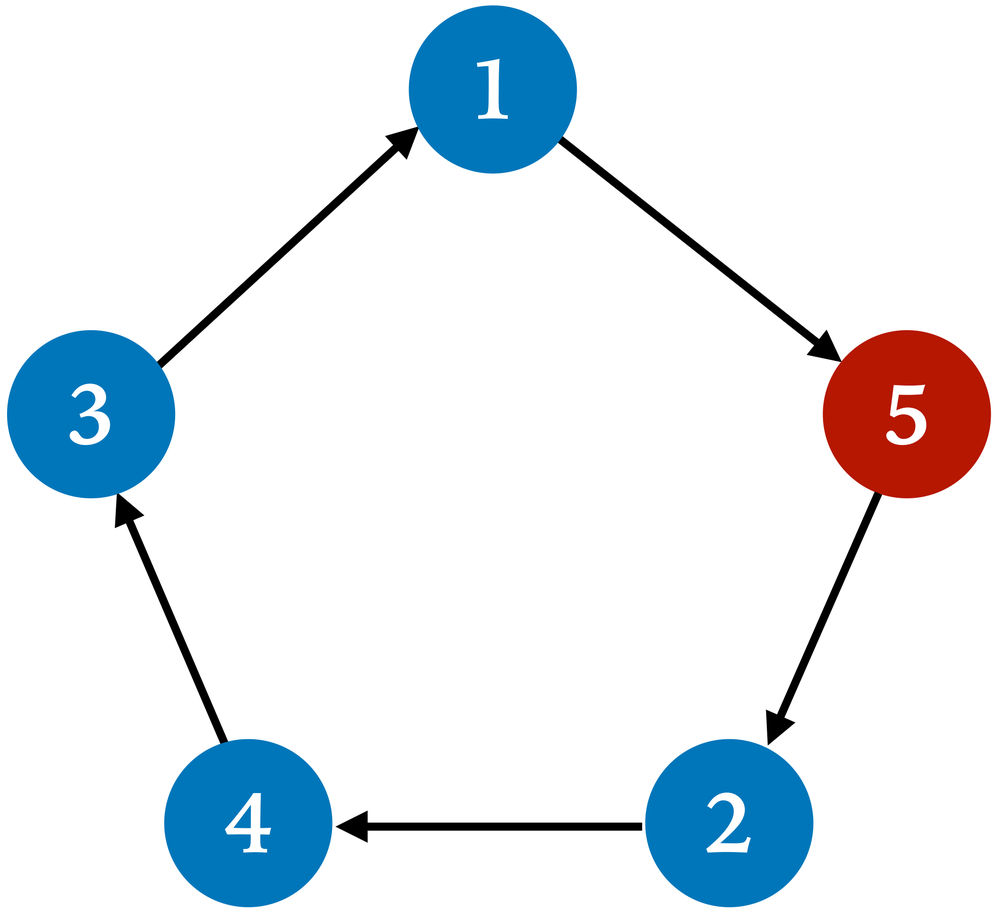


Ring Leader Election



leader elected

Safety Property to Verify



There is *at most one leader*.

Demo 1

(Examples/Tutorial/RingNat.lean in the repository)

Intermezzo: Die Hard 3

In *Die Hard 3*, McClane (Bruce Willis) and Zeus (Samuel L. Jackson) must defuse a bomb at a park fountain by placing exactly 4 gallons of water on a scale, using only a 5-gallon jug and a 3-gallon jug with no other measuring tools.

https://www.youtube.com/watch?v=2vdF6NASMiE

Symbolic Reasoning

Symbolic Reasoning

- Given a system **S** and a safety property **P**, we want to *prove* that all reachable states of **S** satisfy **P**
  - **Symbolic testing**: sound, but not complete → fully automated
  - **Symbolic verification**: sound and complete → semi-automated
    - Requires *inductive invariants* to be provided

Symbolic Testing

- Symbolic bounded model checking (BMC)
  - $\exists\ s_0\ s_1\ \ldots\ s_k.\ \mathit{Init}\ s_0 \wedge \mathit{Tr}\ s_0\ s_1 \wedge \ldots \wedge \mathit{Tr}\ s_{k-1}\ s_k \wedge P\ s_k$
  - $\forall\ s_0\ s_1\ \ldots\ s_k.\ \mathit{Init}\ s_0 \wedge \mathit{Tr}\ s_0\ s_1 \wedge \ldots \wedge \mathit{Tr}\ s_{k-1}\ s_k \Rightarrow P\ s_k$

Symbolic Verification

- Inductive invariant that over-approximates the set of reachable states
  - `∀ s. Init s ⇒ Inv s` **Initiation**
  - `∀ s s'. Inv s ∧ Tr s s' ⇒ Inv s'` **Consecution**
  - `∀ s. Inv s ⇒ Safety s` **Safety**
- Corollary: all reachable states satisfy `Inv` and thus `Safety`

Abstraction

- Full implementation details are often a hindrance to symbolic reasoning
  - It's helpful to *hide* implementation details behind *abstractions*
- Key idea: operate on an *abstract* state space that is easier to describe and symbolically reason about
  - E.g., rather than talk about natural numbers, talk about uninterpreted sorts with a total order
  - Come up with domain-specific axiomatisations

Decidable Abstractions

- SMT solvers use First-Order Logic for symbolic reasoning
- Some well-studied fragments of FOL are *decidable*
  - These fragments are expressive enough to encode complex distributed protocols (see the entire line of work on Ivy by *Padon et al.*)
- Veil comes with a small library of decidable abstractions for common design patterns in distributed protocols

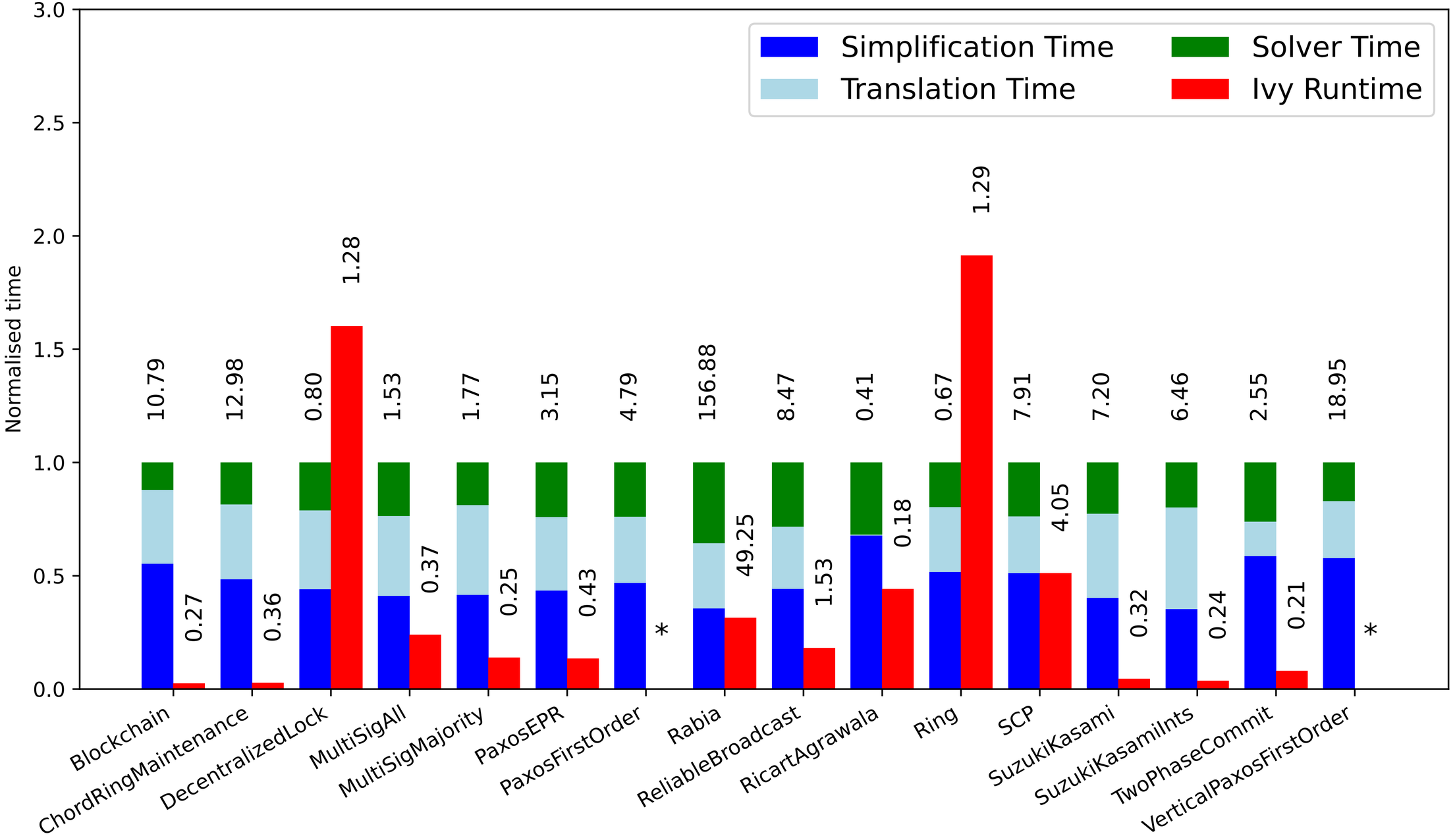
Demo 2

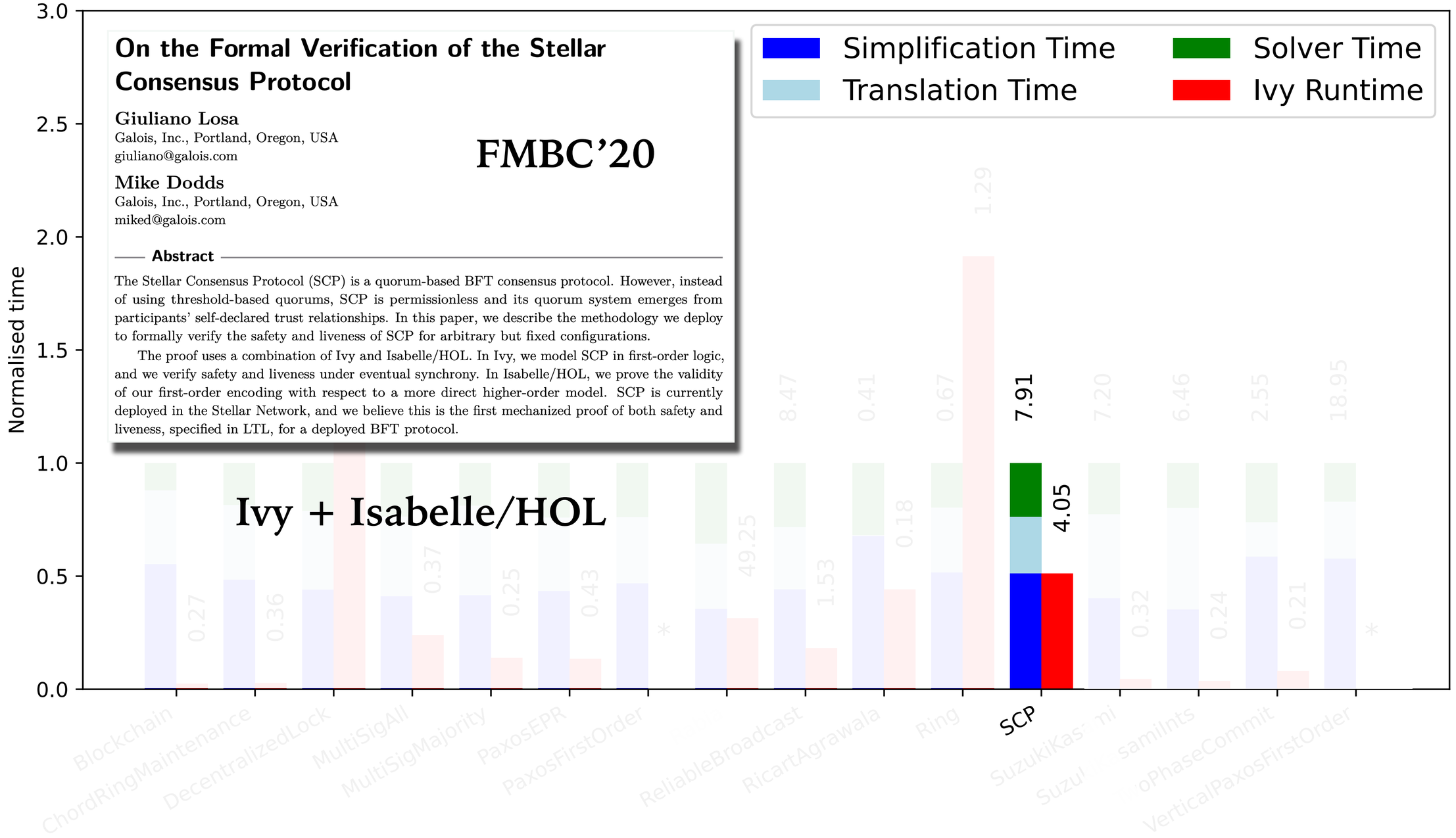
(Examples/Tutorial/RingDec.lean in the repository)

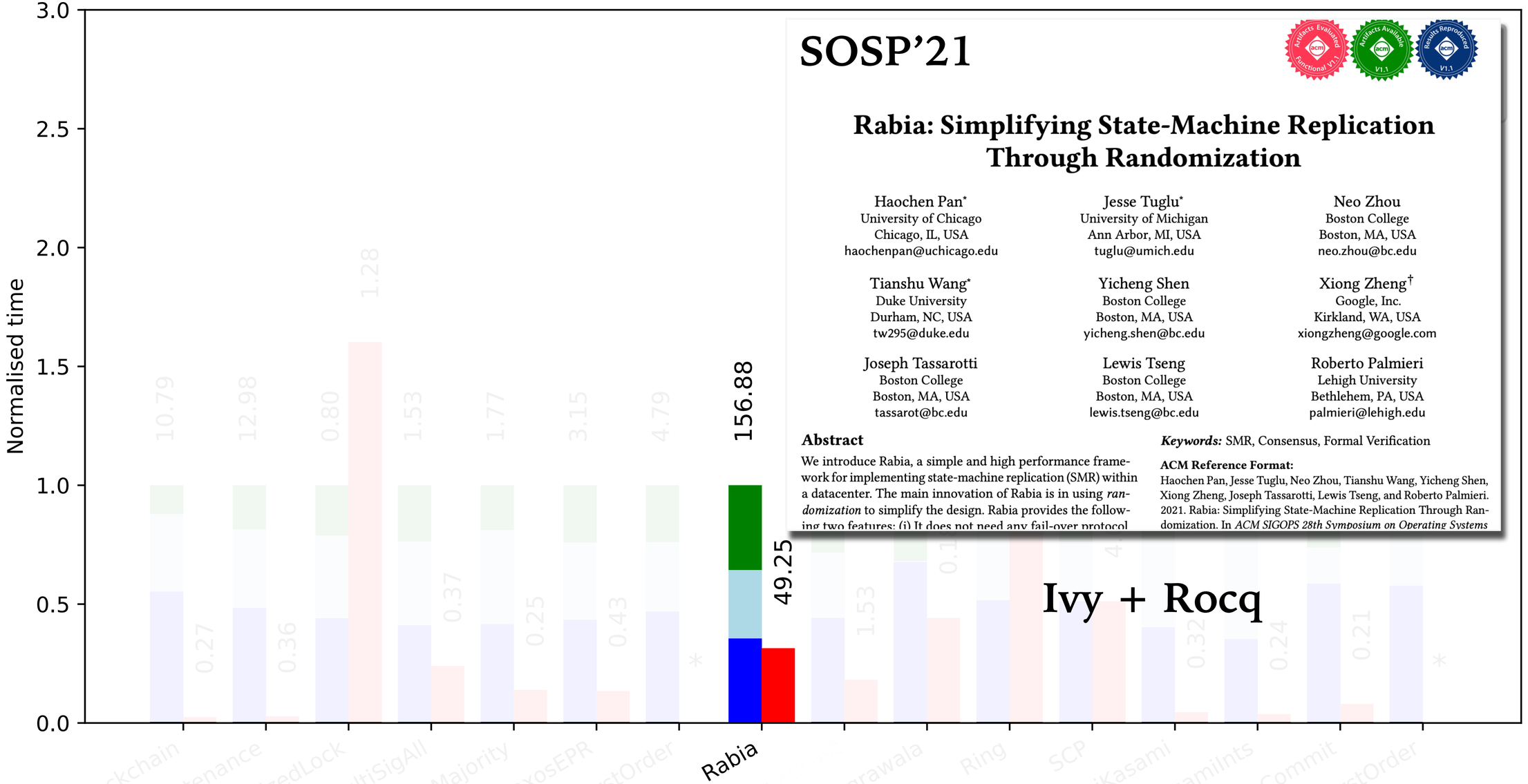
Summary of the Demo

- A protocol state in Veil consists of *state* and *actions*—ordinary Lean types
- TLC-style concrete model allows for fast bug discovery, but is *unsound*
- With FOL-friendly encoding, reachability can be *symbolically tested* using BMC
- Invariants are verified to *hold under all actions*, via SMT or interactively

Case Studies


Simplification Time
Translation Time
Solver Time
Ivy Runtime
Normalised time
3.0
2.5
2.0
1.5
1.0
0.5
0.0
10.79
0.27
12.98
0.36
0.80
1.28
1.53
0.37
1.77
0.25
3.15
0.43
4.79
*
156.88
49.25
8.47
1.53
0.41
0.18
0.67
1.29
7.91
4.05
7.20
0.32
6.46
0.24
2.55
0.21
18.95
*
Blockchain
ChordRingMaintenance
DecentralizedLock
MultiSigAll
MultiSigMajority
PaxosEPR
PaxosFirstOrder
Rabia
ReliableBroadcast
RicartAgrawala
Ring
SCP
SuzukiKasami
SuzukiKasamiInts
TwoPhaseCommit
VerticalPaxosFirstOrder


On the Formal Verification of the Stellar Consensus Protocol
Giuliano Losa
Galois, Inc., Portland, Oregon, USA
giuliano@galois.com
Mike Dodds
Galois, Inc., Portland, Oregon, USA
miked@galois.com
FMBC'20
Abstract
The Stellar Consensus Protocol (SCP) is a quorum-based BFT consensus protocol. However, instead of using threshold-based quorums, SCP is permissionless and its quorum system emerges from participants' self-declared trust relationships. In this paper, we describe the methodology we deploy to formally verify the safety and liveness of SCP for arbitrary but fixed configurations.
The proof uses a combination of Ivy and Isabelle/HOL. In Ivy, we model SCP in first-order logic, and we verify safety and liveness under eventual synchrony. In Isabelle/HOL, we prove the validity of our first-order encoding with respect to a more direct higher-order model. SCP is currently deployed in the Stellar Network, and we believe this is the first mechanized proof of both safety and liveness, specified in LTL, for a deployed BFT protocol.
Ivy + Isabelle/HOL
Simplification Time
Translation Time
Solver Time
Ivy Runtime
Normalised time
3.0
2.5
2.0
1.5
1.0
0.5
0.0
7.91
4.05
SCP
Blockchain
ChordRingMaintenance
DecentralizedLock
MultiSigAll
MultiSigMajority
PaxosEPR
PaxosFirstOrder
ReliableBroadcast
RicartAgrawala
Ring
SuzukiKas
Suzul
samiInts
oPhaseCommit
VerticalPaxosFirstOrder



SOSP'21

Artifacts Evaluated – Functional V1.1 · Artifacts Available V1.1 · Results Reproduced V1.1

Rabia: Simplifying State-Machine Replication Through Randomization

Haochen Pan*
University of Chicago
Chicago, IL, USA
haochenpan@uchicago.edu

Jesse Tuglu*
University of Michigan
Ann Arbor, MI, USA
tuglu@umich.edu

Neo Zhou
Boston College
Boston, MA, USA
neo.zhou@bc.edu

Tianshu Wang*
Duke University
Durham, NC, USA
tw295@duke.edu

Yicheng Shen
Boston College
Boston, MA, USA
yicheng.shen@bc.edu

Xiong Zheng†
Google, Inc.
Kirkland, WA, USA
xiongzheng@google.com

Joseph Tassarotti
Boston College
Boston, MA, USA
tassarot@bc.edu

Lewis Tseng
Boston College
Boston, MA, USA
lewis.tseng@bc.edu

Roberto Palmieri
Lehigh University
Bethlehem, PA, USA
palmieri@lehigh.edu

Abstract

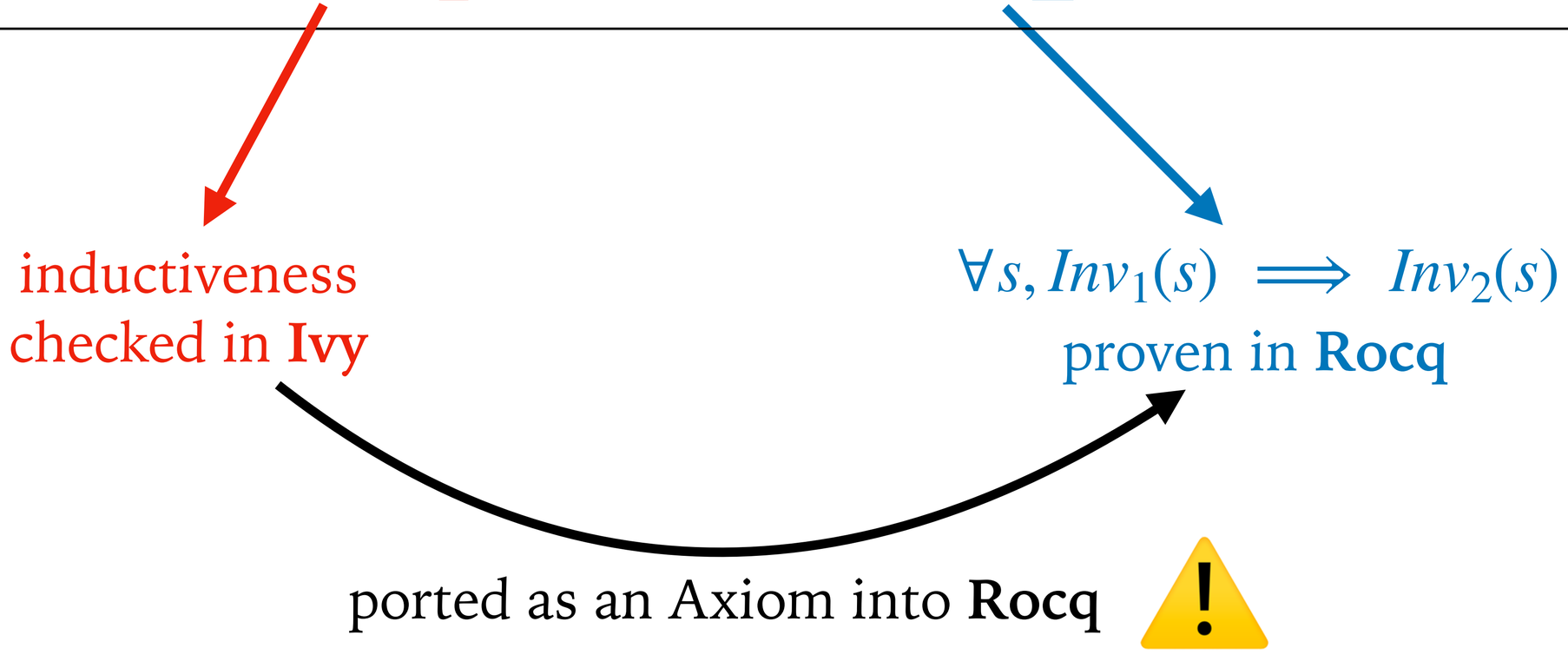
We introduce Rabia, a simple and high performance framework for implementing state-machine replication (SMR) within a datacenter. The main innovation of Rabia is in using *randomization* to simplify the design. Rabia provides the following two features: (i) It does not need any fail-over protocol

Keywords: SMR, Consensus, Formal Verification

ACM Reference Format:
Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying State-Machine Replication Through Randomization. In *ACM SIGOPS 28th Symposium on Operating Systems*

Case Study: The Rabia Protocol

Claim: $Inv_1 \wedge Inv_2$ is an invariant.



Case Study: The Rabia Protocol

- Moving facts between the provers introduced a discrepancy

Ivy

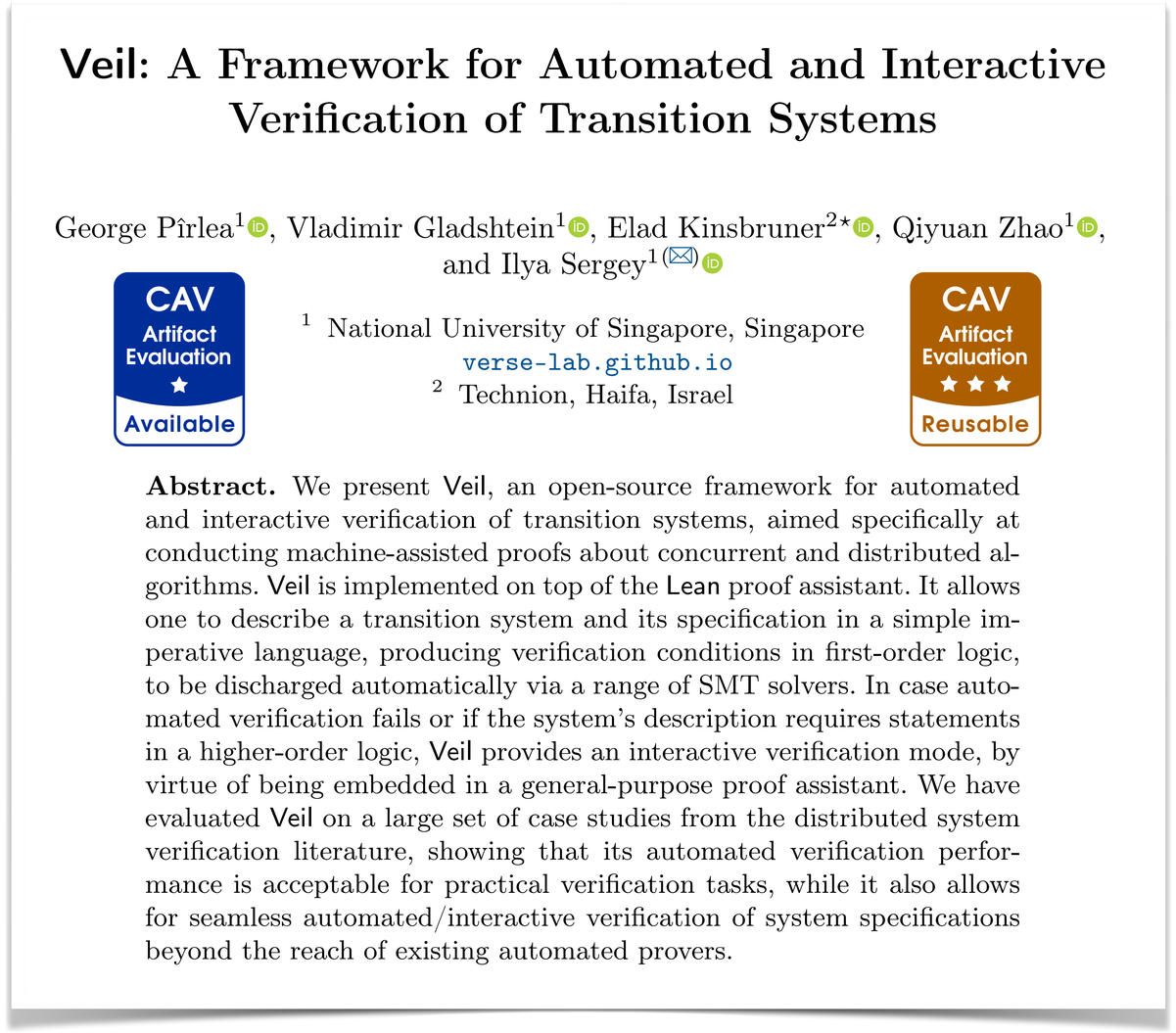
```
specification {
  conjecture [started_pred] (ind_defs.started(Psucc) & succ(P, Psucc) -> ind_defs.started(Psucc))
}
```

Rocq

```
Definition started_pred σ :=
  (∀ P, started σ (succ P) → started σ P).
Axiom started_pred_invariant : invariant started_pred.
```

- In **Veil**, when proving Inv_2 , the proof of Inv_1 (from `#check_invariants`) is reused; this allowed us to catch and fix this discrepancy.

- **Veil** is a Lean library for *automated/interactive* verification of distributed protocols
- Supports both concrete and symbolic model checking, as well and deductive proofs
- Acceptable performance for FOL via SMT, *seamless integration* with HO Lean specs



Veil: A Framework for Automated and Interactive Verification of Transition Systems

George Pîrlea[1], Vladimir Gladshtein[1], Elad Kinsbruner[2*], Qiyuan Zhao[1], and Ilya Sergey[1]

CAV Artifact Evaluation ★ Available

[1] National University of Singapore, Singapore
verse-lab.github.io
[2] Technion, Haifa, Israel

CAV Artifact Evaluation ★★★ Reusable

Abstract. We present Veil, an open-source framework for automated and interactive verification of transition systems, aimed specifically at conducting machine-assisted proofs about concurrent and distributed algorithms. Veil is implemented on top of the Lean proof assistant. It allows one to describe a transition system and its specification in a simple imperative language, producing verification conditions in first-order logic, to be discharged automatically via a range of SMT solvers. In case automated verification fails or if the system's description requires statements in a higher-order logic, Veil provides an interactive verification mode, by virtue of being embedded in a general-purpose proof assistant. We have evaluated Veil on a large set of case studies from the distributed system verification literature, showing that its automated verification performance is acceptable for practical verification tasks, while it also allows for seamless automated/interactive verification of system specifications beyond the reach of existing automated provers.

github.com/verse-lab/veil

Under the Hood:
Generating Verification Conditions

System Specification

initial state

```
after_init {
  leader N := False
  pending M N := False
}
```

actions

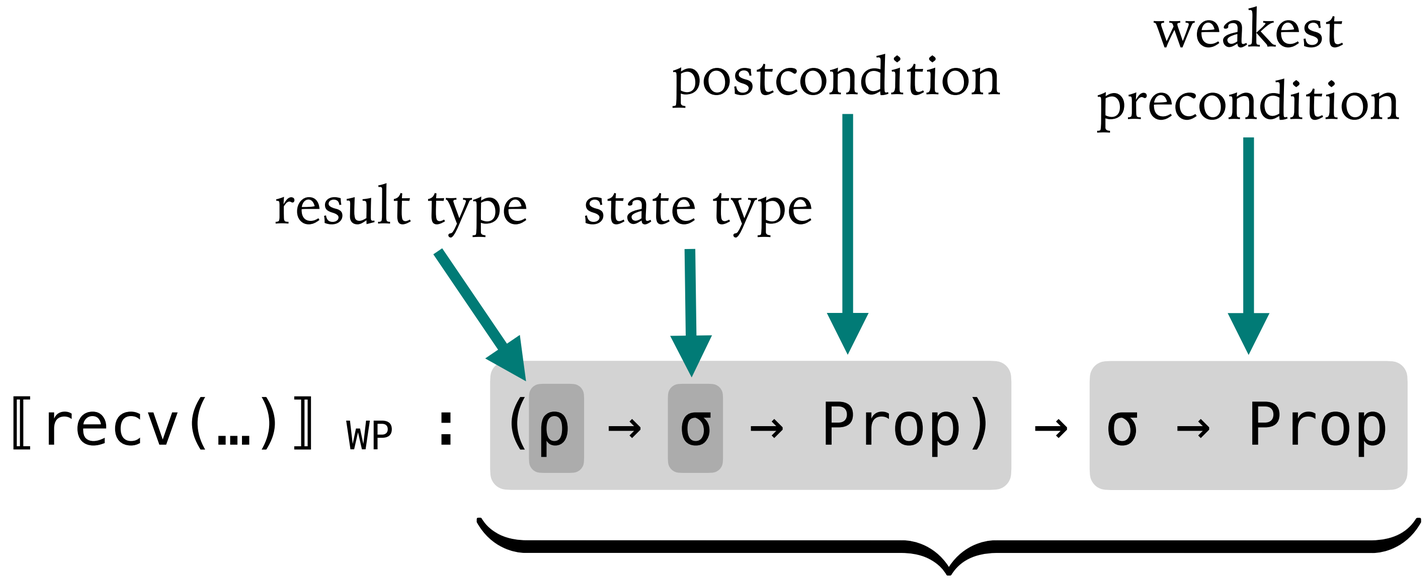
```
action send (n next : node) = {
  require n ≠ next ∧ ∀ Z,
          ((Z ≠ n ∧ Z ≠ next) → btw n next Z)
  pending n next := True
}

action recv (id n next : node) = {
  require isNext n next
  require pending id n
  pending id n := *
  if (id = n) then
    leader n := True
  else
    if (le n id) then
      pending id next := True
}
```

safety properties

```
safety [single_leader] leader L1 ∧ leader L2 → L1 = L2

invariant [leader_greatest] leader L → le N L
invariant [self_msg_only_if_greatest] pending L L → le N L
invariant [no_bypass] pending S D ∧ btw S N D → le N S
```

Semantics of Actions

```
action recv (id n next : node) = {
  require isNext n next
  require pending id n
  pending id n := *
  if (id = n) then
    leader n := True
  else
    if (le n id) then
      pending id next := True
}
```



Verification Conditions for Safety

```
safety [single_leader] leader L1 ∧ leader L2 → L1 = L2
```

```
invariant [leader_greatest] leader L → le N L
invariant [self_msg_only_if_greatest] pending L L → le N L
invariant [no_bypass] pending S D ∧ btw S N D → le N S
```

```
⟦act⟧ WP : (ρ → σ → Prop) → σ → Prop
```

```
after_init {
  leader N := False
  pending M N := False
}
```

```
Inv = λ (r: ρ) (s: σ). Inv s
```

1. ∀ s. ⟦after_init⟧ WP **Inv** s
2. ∀ act s. **Inv** s ⇒ ⟦act⟧ WP **Inv** s
3. ∀ s. Inv s ⇒ Safety s

Generating Weakest Preconditions Automatically

$$[\![\texttt{act}]\!]_{\texttt{WP}} : \underbrace{(\rho \to \sigma \to \texttt{Prop}) \to \sigma \to \texttt{Prop}}$$

weakest precondition transformer (WP σ ρ)

```
def WP.pure (r : ρ) : WP σ ρ := fun s post => post r s

def WP.bind (c : WP σ ρ) (next : ρ → WP σ ρ) : WP σ ρ :=
                        fun s post => c (fun r s' => next r post s') s
```

a.k.a. a Dijkstra monad

Verifying higher-order programs with the Dijkstra monad, Swamy et al., PLDI'13

Generating Weakest Preconditions Automatically

```
def WP.pure (r : ρ) : WP σ ρ := fun s post => post r s

def WP.bind (c : WP σ ρ) (next : ρ → WP σ ρ) : WP σ ρ :=
                              fun s post => c (fun r s' => next r post s') s
```

```
action recv (id n next : node) = {
  require isNext n next
  require pending id n
  pending id n := *
  if (id = n) then
    leader n := True
  else
    if (le n id) then
      pending id next := True
}
```

Generating Weakest Preconditions Automatically

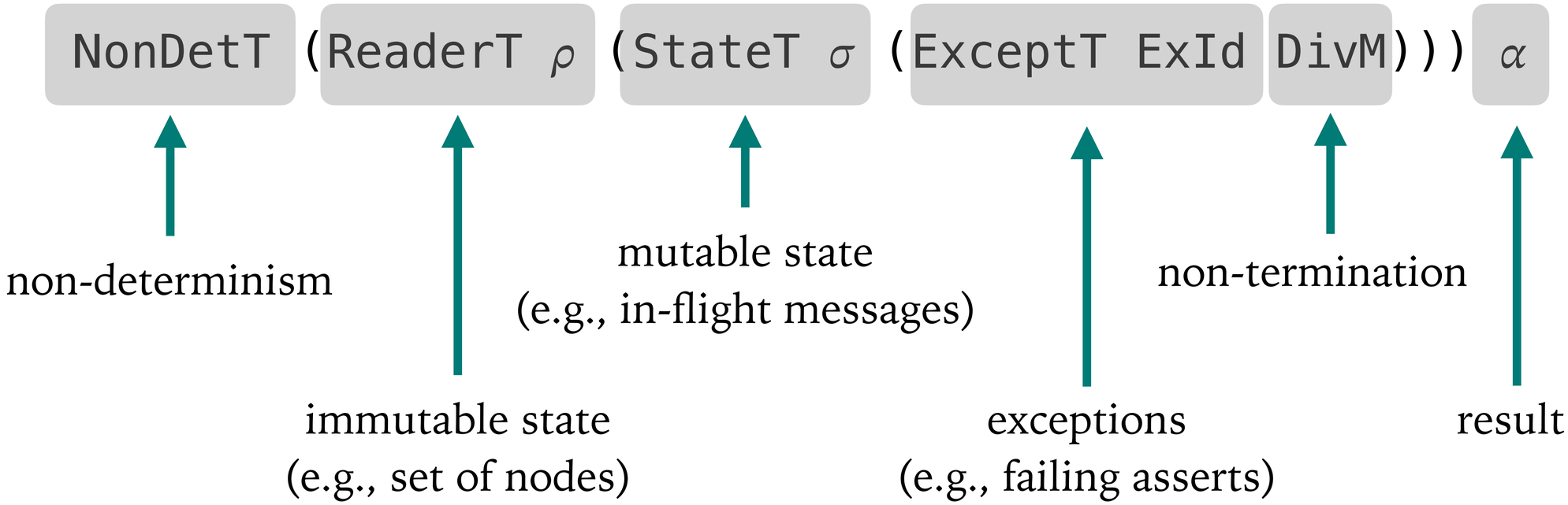
```
def WP.pure (r : ρ) : WP σ ρ := fun s post => post r s

def WP.bind (c : WP σ ρ) (next : ρ → WP σ ρ) : WP σ ρ :=
                                  fun s post => c (fun r s' => next r post s') s
```

```
action recv (id n next : node) = do
  require isNext n next
  require pending id n

  let newPending ← nondet
  pending := newPending

  if (id = n) then
    leader n := True
  else
    if (le n id) then
      pending id next := True
```

```
def nondet (τ : Type) : WP σ τ := fun s post => ∀ (t : τ), post
```

A Computation Monad for Veil Language

```
def VeilM (ρ σ α : Type) :=
  NonDetT (ReaderT ρ (StateT σ (ExceptT ExId DivM))) α
```



Veil's Meta-Theory in a Nutshell

- Runtime semantics of actions is a composition of *monadic effects*:
  state (mutable and immutable), non-determinism, exceptions, divergence
- Each effect comes with a *WP (Dijkstra) monad transformer* for VC generation
- *Soundness* of VCgen for each effect is derived automatically and modularly
- Some effects come with additional parameters:
  e.g., *angelic v. demonic* nondeterminism, treatment of *divergence*

The end of

Part I:

Lean as a Multi-Modal Verifier for Distributed Protocols

Part II:

Lean as a Multi-Modal *Meta* Verifier

Zoo of Automated Verifiers

P∗rust→∗i

How do I implement a verifier for a language X?

- Implement a front-end (parser, type checker)
- Verification condition (VC) generator for deductive proofs
  - Direct compilation into SMT-LIB (Boogie, F*, Verus)
  - Using an intermediate verification language (Dafny, Viper, Prusti)
  - Bespoke domain-specific tactics (VST, Iris, RefinedC, RefinedRust)
- Can I combine automated and interactive proofs?
- Can I also run my code and test it?
  - (Mostly) unverified extraction to another language (Rocq, Dafny, F*)
  - Third-party compiler (Viper-based tools)
- Can I have symbolic execution, such as KLEE?
- Can I extend my prototype for a language feature Y?
  - Exceptions, non-determinism, resource consumptions, probabilistic choice, etc.
- Can I prove that my VCgen and SymEx are correct wrt. runtime?

How do I implement a verifier for a language X?

- Implement a front-end (parser, type checker)

deep/shallow embedding

- Verification condition (VC) generator for deductive proofs
  - Direct compilation into SMT-LIB (Boogie, F*, Verus)
  - Using an intermediate verification language (Dafny, Viper, Prusti)
  - Bespoke domain-specific tactics (VST, Iris, RefinedC, RefinedRust)
- Can I combine automated and interactive proofs?
- Can I also run my code and test it?
  - (Mostly) unverified extraction to another language (Rocq, Dafny, F*)
  - Third-party compiler (Viper-based tools)
- Can I have symbolic execution, such as KLEE?

multi-modal verification

- Can I extend my prototype for a language feature Y?
  - Exceptions, non-determinism, resource consumptions, probabilistic choice, etc.

extensible

- Can I prove that my VCgen and SymEx are correct wrt. runtime?

foundational

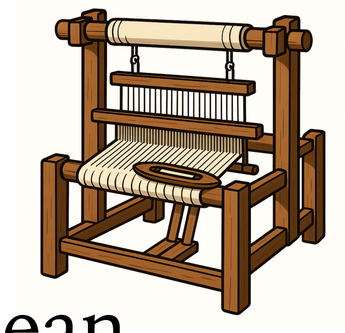
What if we use Lean as a Meta-Verifier?

Lean as a Meta-Verifier

- Great for deep/shallow programming language embedding
  - Metaprogramming for syntactic extension
  - Dependently typed system for lightweight validation
- Multi-modal verification
  - Native interactive proof mode
  - Versatile automation (**aesop**, **grind**)
  - Interaction with third-party provers (**lean-auto**, **LeanSMT**)
  - Compiles and runs natively
- Extensibility?
- Foundational embeddings “for free”?

Let’s generalise the Veil embedding!

Loom



- A library to build verifiers for shallowly embedded PL in Lean
- Multi-modal verification
  - Language-agnostic generation of SMT-friendly VCs
  - Using interactive mode/**aesop**/**grind** when SMT automation fails
  - Language-agnostic symbolic execution
  - Runtime support by running Lean natively
- Extensible
  - Novel theoretical foundation: *monad transformer algebras* for composing multiple effects
  - Easily extensible with additional effects (e.g., resource tracking, probabilistic choice)
- Foundational verification
  - *Sound* VC generators in Loom are generated automatically from the effect composition
  - Loom-based verifiers are *correct by construction*

Loom

owly embedded PL in Lean

utoma

ebras fo

urce tracking, probabilistic choice)

atically from the effect composition

Foundational Multi-Modal Program Verifiers

VLADIMIR GLADSHTEIN, National University of Singapore, Singapore
GEORGE PÎRLEA, National University of Singapore, Singapore
QIYUAN ZHAO, National University of Singapore, Singapore
VITALY KURIN*, Neapolis University Pafos, Cyprus
ILYA SERGEY, National University of Singapore, Singapore

Multi-modal program verification is a process of validating code against its specification using both dynamic and symbolic techniques, and proving its correctness by a combination of automated and interactive machine-assisted tools. In order to be trustworthy, such verification tools must themselves come with formal *soundness* proofs, establishing that *any* program verified in them against a certain specification does not violate the specification's statement when executed. Verification tools that are proven sound in a general-purpose proof assistant with a small trusted core are commonly referred to as *foundational*.

We present a framework that facilitates and streamlines construction of program verifiers that are both foundational and multi-modal. Our approach adopts the well-known idea of *monadic shallow embedding* of an *executable* program semantics into the programming language of a theorem prover based on higher-order logic, in our case, the Lean proof assistant. We provide a library of monad transformers for such semantics, encoding a variety of computational effects, including state, divergence, exceptions, and non-determinism. The key theoretical innovation of our work are *monad transformer algebras* that enable *automated derivation* of the respective sound verification condition generators. We show that proofs of the resulting verification conditions enjoy automation using off-the-shelf SMT solvers and allow for an interactive proof mode when automation fails. To demonstrate versatility of our framework, we instantiated it to embed two foundational multi-modal verifiers into Lean for reasoning about (1) distributed protocol safety and (2) Dafny-style specifications of imperative programs, and used them to mechanically verify a number of non-trivial case studies.

Shipping with Lean ≥4.31

Loom instance I: Veil

Examples > Tutorial > ∀ RingDec.lean

```
3   veil module RingDec
4
5   type node
6   instantiate tot : TotalOrder node
7   instantiate btwn : Between node
8
9   open Between TotalOrder
10
11  relation leader : node → Bool
12  relation pending : node → node → Bool
13
14  #gen_state
15
16  after_init {
17    leader N := false
18    pending M N := false
19  }
20
21  ghost relation isNext (n : node) (next : node) :=
22    ∀ Z, n ≠ next ∧ ((Z ≠ n ∧ Z ≠ next) → btw n next Z)
23
24  action send (n next : node) {
25    require isNext n next
26    pending n next := true
27  }
28
29  action recv (sender n next : node) {
30    require isNext n next
31    require pending sender n
32    pending sender n := false
33    if (sender = n) then
34      leader n := true
35    else
36      if (le n sender) then
37        pending sender next := true
38  }
39
40  safety [single_leader] leader N ∧ leader M → N = M
41  invariant [leader_greatest] leader L → le N L
42  invariant [self_msg_greatest] pending L L → le N L
43  -- invariant [drop_smaller] pending S D ∧ btw S N D → le N S
44
45  #time #gen_spec
46
47  #check_invariants
48
49  end RingDec
```

▼RingDec.lean:47:17

▼HTML Display

Show JSON

Filter by status (12 VCs): All (12) | ✅ Proven (11) | ❌ Disproven (1)

⚠ Trusting SMT solver for 8 goals. `set_option veil.smt.trust false` to enable proof reconstruction.

Initialization must establish the invariant:

✅ doesNotThrow	30ms
✅ single_leader	67ms
✅ leader_greatest	74ms
✅ self_msg_greatest	57ms

The following set of actions must preserve the invariant:

▼ send

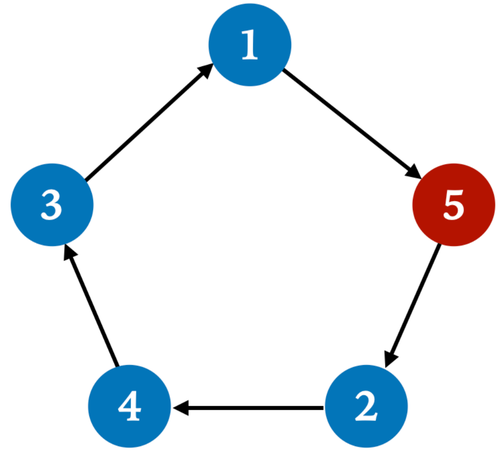
✅ doesNotThrow	33ms
✅ single_leader	76ms
✅ leader_greatest	71ms
✅ self_msg_greatest	90ms

▼ recv

✅ doesNotThrow		40ms
✅ single_leader		115ms
✅ leader_greatest		96ms
▶ ❌ self_msg_greatest	TR	94ms+218ms

Restart File

▼Messages (⊗ 1 ⚠ 1)



There is *at most one leader*.

Loom instance II: Velvet

- A **Dafny/Verus**-style verifier in Lean
- Implemented as a shallow embedding
- Tailored for *intrinsic* verification
  - Can support extrinsic proofs, too
- Features:
  - non-determinism (:| operator)
  - non-termination
- Partial/total correctness proofs
- Immediately executable
  - comes with QuickCheck-style PBT generator

```
method insertionSort
  (mut arr: arrInt) return (u: Unit)
  require 1 ≤ size arr
  ensures forall i j, 0 ≤ i ∧ i ≤ j ∧ j < size arr → arrNew[i] ≤ arrNew[j]
  ensures toMultiset arr = toMultiset arrNew
  do
    let arr₀ := arr
    let arr_size := size arr
    let mut n := 1
    while n ≠ size arr
    invariant size arr = arr_size
    invariant 1 ≤ n ∧ n ≤ size arr
    invariant forall i j, 0 ≤ i ∧ i < j ∧ j <= n - 1 → arr[i] ≤ arr[j]
    invariant toMultiset arr = toMultiset arr₀
    -- decreasing size arr - n
    do
      let mut mind := n
      while mind ≠ 0
      invariant size arr = arr_size
      invariant mind ≤ n
      invariant forall i j, 0 ≤ i ∧ i < j ∧ j ≤ n ∧ j ≠ mind → arr[i] ≤ arr[j]
      invariant toMultiset arr = toMultiset arr₀
      -- decreasing mind
      do
        if arr[mind] < arr[mind - 1] then
          swap arr[mind - 1] arr[mind]
        mind := mind - 1
      n := n + 1 -- try commenting this out for termination
    return
prove_correct insertionSort by
  dsimp [insertionSort]
  loom_solve!
```

Demo 3

Repository: github.com/verse-lab/velvet

Branch **leanlang-demo**

```
lake exe cache get; lake build
```

Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean

Vladimir Gladshtein[1], Vitaly Kurin[2*], Yueyang Feng[1], Dipesh Kafle[1], George Pîrlea[1], Qiyuan Zhao[1], and Ilya Sergey[1] (✉)

[1] VERSE lab, National University of Singapore, Singapore
verse-lab.org
[2] Neapolis University Pafos, Cyprus

Abstract. We present Velvet—a Dafny-style verifier for imperative programs embedded in the Lean proof assistant. Like Dafny, Velvet supports reasoning about effectful programs featuring mutable state, loops, and non-determinism. Unlike Dafny, Velvet seamlessly combines automated SMT-based proofs with the interactive proof mode of the Lean proof assistant, in which it is embedded, thus enabling *multi-modal* proofs. Implemented as a Lean library, Velvet enjoys interaction with the rest of the Lean ecosystem, and in particular, with its automation tactics and rich library of mathematical theories. In this paper, we give a tour of Velvet's features, outline the techniques underlying its implementation, and evaluate its performance and expressivity in comparison with Dafny.

Certified Program Synthesis with a Multi-Modal Verifier

Yueyang Feng*
National University of Singapore
Singapore
yueyangfeng@u.nus.edu

Dipesh Kafle*
National University of Singapore
Singapore
dipesh@u.nus.edu

Vladimir Gladshtein
National University of Singapore
Singapore
vovaglad@u.nus.edu

Vitaly Kurin
Neapolis University Pafos
Cyprus

George Pîrlea
National University of Singapore
Singapore

Qiyuan Zhao
National University of Singapore
Singapore

WybeCoder: Verified Imperative Code Generation

Fabian Gloeckle*[1,2], Mantas Bakšys*[1,3], Darius Feher[1,4], Kunhao Zheng[1,5], Amaury Hayat[2,6], Sean B. Holden[3], Gabriel Synnaeve[1], Peter O'Hearn[1,4]

*Equal contribution

[1]FAIR at Meta [2]CERMICS, ENPC, Institut Polytechnique de Paris, CNRS [3]Computer Lab, University of Cambridge
[4]University College London [5]Miles, LAMSADE, Université Paris-Dauphine-PSL [6]Korea Institute for Advanced Study

Paper | Code | Trajectory Viewer

Abstract

Recent progress in large language models (LLMs) has advanced automatic code generation and formal theorem proving, yet software verification has not seen the same improvement. To address this gap, we propose **WybeCoder**, an agentic code verification framework that enables *prove-as-you-generate* development where code, invariants, and proofs co-evolve. It builds on a recent framework that combines automatic verification condition generation and SMT solvers with interactive proofs in Lean. To enable systematic evaluation, we translate two benchmarks for functional verification in Lean, Verina and Clever, to equivalent imperative code specifications. On complex algorithms such as Heapsort, we observe consistent performance improvements by scaling our approach, synthesizing dozens of valid invariants and dispatching dozens of subgoals, resulting in hundreds of lines of verified code, overcoming plateaus reported in previous works. Our best system solves **74%** of Verina tasks and **62%** of Clever tasks at moderate compute budgets, significantly surpassing previous evaluations and paving a path to automated construction of large-scale datasets of verified imperative code.

Reading List

- *Veil: A Framework for Automated and Interactive Verification of Transition Systems* (CAV'25)
- *Foundational Multi-Modal Program Verifiers* (POPL'26)
- *Lessons from Building an Auto-Active Verifier in Lean* (Dafny'26)
- *Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean* (CAV'26)
- *Certified Program Synthesis with a Multi-Modal Verifier* (ASE'26)
- *Proof Assistant as Platform: Building the Next Generation of Verifiers*, PhD Thesis
  George Pîrlea, 2026
  https://pirlea.net/papers/thesis-draft.pdf
- *Proofs and Intuitions*, Research Blog, https://proofsandintuitions.net

To Take Away

- Lean is a *great IR for verification* that enables *multi-modal verification*
- Program semantics can be embedded shallowly as *compositions of effects*
- This way, we can combine *testing* with *automated* and *interactive* proofs
- *Verifiers-as-libraries*: build your next verifier in Lean!

Stay tuned!

verse-lab.org

Thanks!