Hi, I'm Isaac
I'm a senior research scientist at Heliax, and
I'm not here with a new proof technique, or a new tool.
I'm here with a report about our experiences using formal models to make the software infrastructure we build more robust and reliable.
I'm hoping that through our experiences, we can build up a kind of wish-list of things that would help industry make better use for formal verification in distributed programs.

I should stress that these projects have been by a bunch of people, but for various reasons, some of them visa related, only I'm here today.
My expertise is in the distributed algorithms side, so I came into this with relatively little understanding of how todo this kind of formal verification, while several of my colleagues came into this with the reverse: expertise in Coq and Agda, but not so much distributed systems.

BACKGROUND

HELIAX

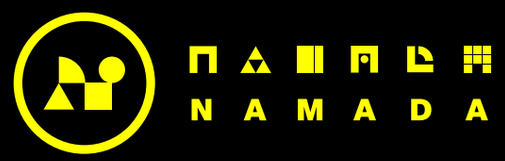
So before I get started, I want to take a minute to talk about what Heliax does.
Heliax is an open source software company working on a couple of projects,
CLICK
anoma
CLICK
And namada.
Both of these are what I would call
CLICK
distributed systems infrastructure.

BACKGROUND

HELIAX

BACKGROUND

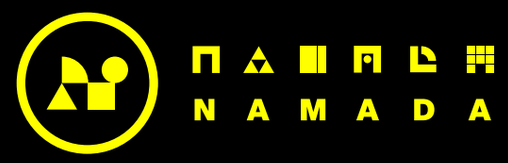
HELIAX



BACKGROUND

HELIAX

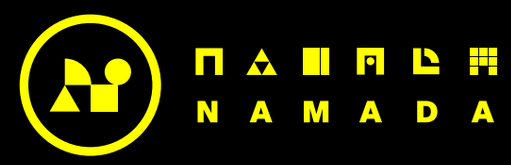
- Distributed Systems Infrastructure



BACKGROUND

HELIAX

- Distributed Systems Infrastructure
- Replicated State Machines (Blockchains)

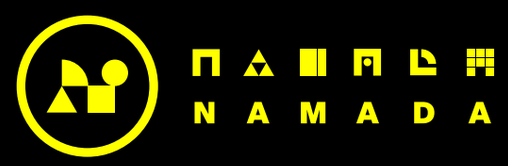


In particular, both of these involve building software to run byzantine fault tolerant replicated state machines, a.k.a. Blockchains.

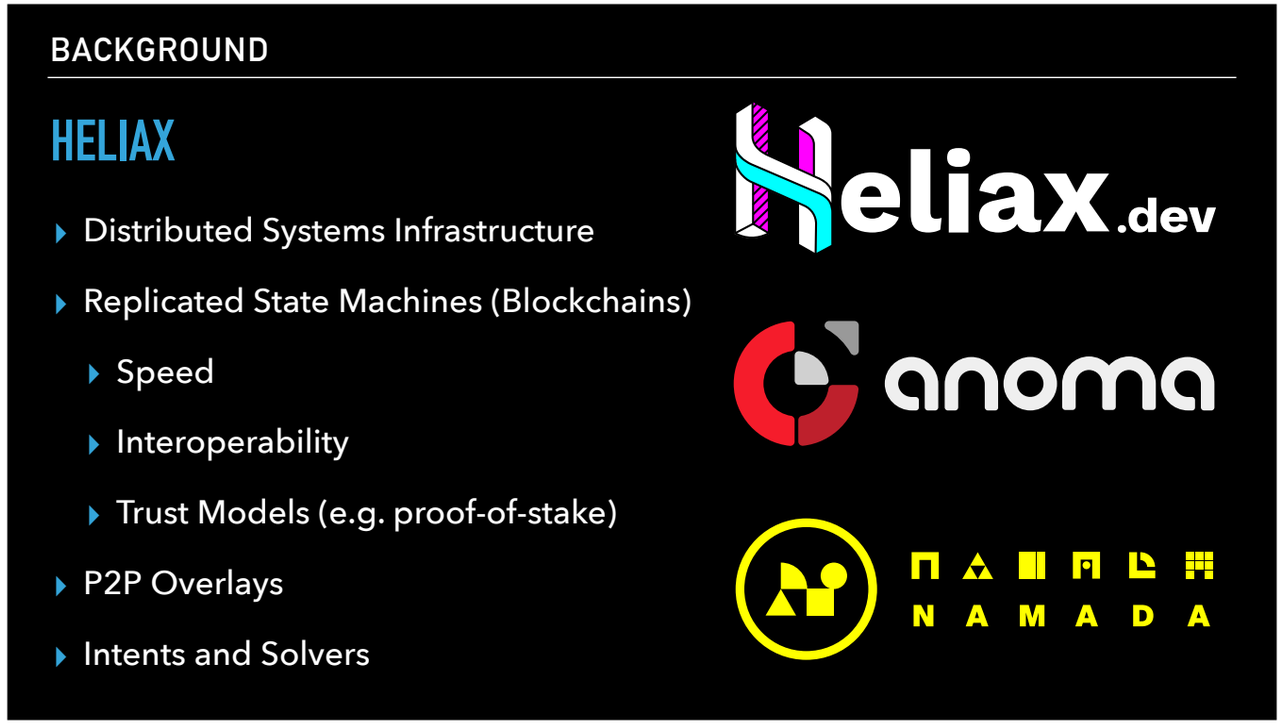
BACKGROUND

HELIAX

- Distributed Systems Infrastructure
- Replicated State Machines (Blockchains)
  - Speed
  - Interoperability
  - Trust Models (e.g. proof-of-stake)



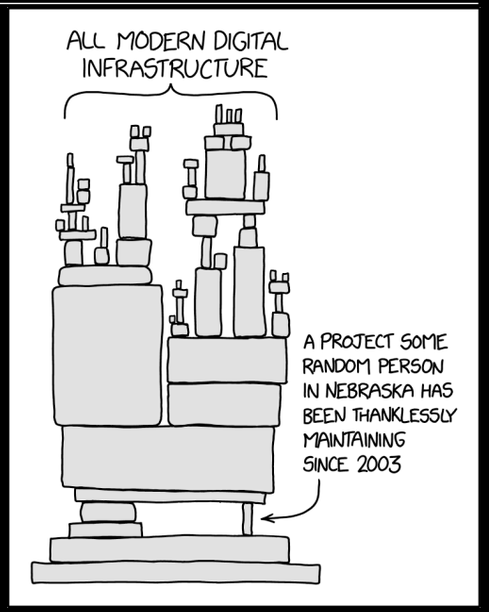
There are several innovations we want to make here, including a lot of stuff to do with speed, so latency and throughput of transaction processing, but a lot of the distributed algorithms stuff comes in when
Accommodating different trust models.
Ultimately, anoma in particular should allow anyone to spin up a replicated state machine with their own trust model, based on authority, or some kind of stake, or whatever, and
We want to allow as much interoperability as possible, so applications can use state across these differently trusted chains.



There are several other distributed aspects of these projects, including P2P overlays, and intents and solvers, which we haven’t done much formal verification work on, so I won’t talk much about them here.

BACKGROUND

SOFTWARE INFRASTRUCTURE



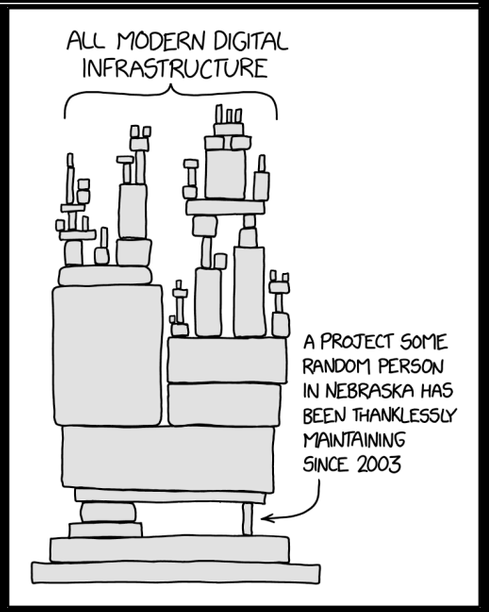
Monroe, Randall. "Dependency," *xkcd*, xkcd.com/2347

OK, so when I call these projects "software infrastructure," what does that mean?
Well, I suppose the most common time I hear "software infrastructure" or "digital infrastructure", it refers to
CLICK

BACKGROUND

SOFTWARE INFRASTRUCTURE

- Libraries used as dependencies



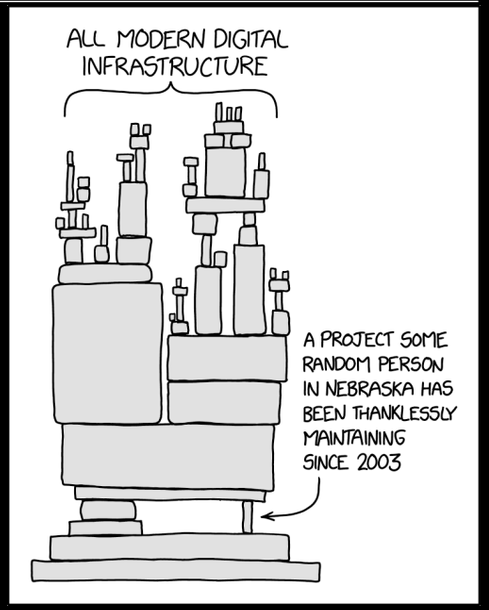
Monroe, Randall. "Dependency," *xkcd*, xkcd.com/2347

Libraries that used as dependencies.
Well, looking at this dependency diagram from xkcd, I guess there's some kind of spectrum, where it gets more and more "infrastructure" as more stuff depends (perhaps transitively) on it.
CLICK

BACKGROUND

SOFTWARE INFRASTRUCTURE

- Libraries used as dependencies *a lot*



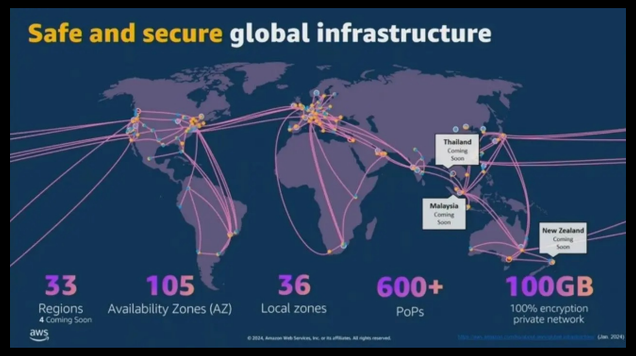
Monroe, Randall. "Dependency," *xkcd*, xkcd.com/2347

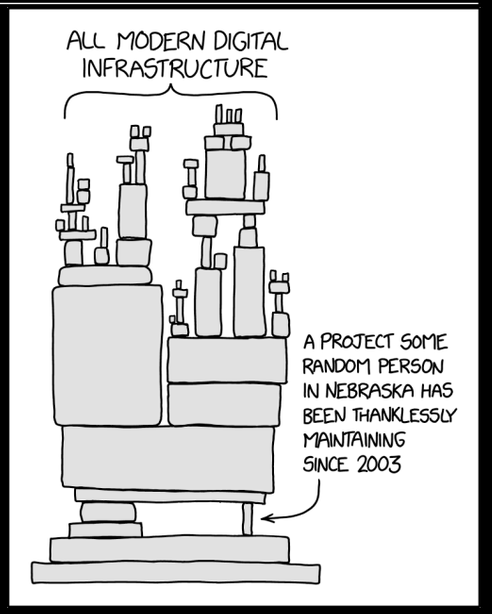
So we'll say libraries that get used as dependencies "a lot" count as infrastructure, like ImageMagick.

BACKGROUND

SOFTWARE INFRASTRUCTURE

- Libraries used as dependencies *a lot*
- Services on which applications depend





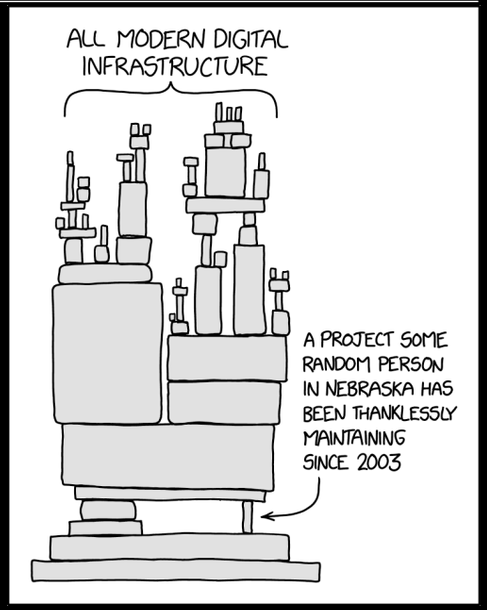
Monroe, Randall. "Dependency," *xkcd*, xkcd.com/2347

But we can also talk about services on which applications depend.
When these services malfunction, lots of otherwise independent applications fail.
So, DNS is infrastructure.
Cloud providers like Google or Microsoft, or, in this case, Amazon, provide infrastructure.

BACKGROUND

SOFTWARE INFRASTRUCTURE

- Libraries used as dependencies *a lot*
- Services on which applications depend
- Widespread ramifications of failure
- Where formal verification is most important



Monroe, Randall. "Dependency," *xkcd*, xkcd.com/2347

What makes infrastructure different from any-old software then is that a bunch of other developers' applications depend on it.

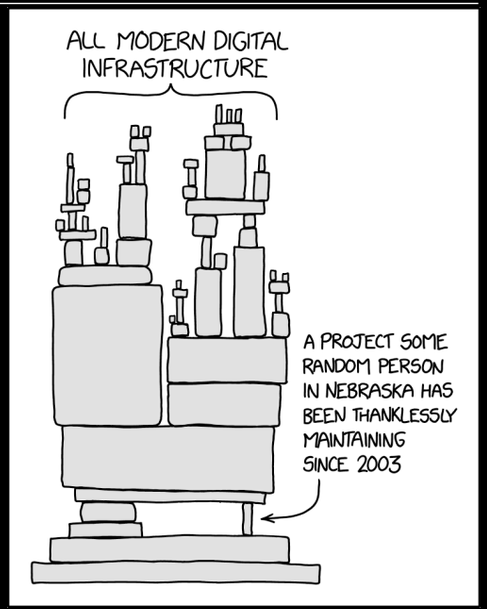
So it seems to me that this is where formal verification is most important.
This is where we want to provide the strongest guarantees, so that other software can build against those guarantees.

BACKGROUND

SOFTWARE INFRASTRUCTURE

- Libraries used as dependencies *a lot*
- Services on which applications depend
- Widespread ramifications of failure
- Where formal verification is most important

anoma



Monroe, Randall. "Dependency," *xkcd*, xkcd.com/2347

So my push for formal verification is built on the assertion that the anoma project is software infrastructure.
We want anyone on the internet to be able to spin up their own relicated state machines with anoma, *and* we want applications to be able to operate reliably over these replicated state machines.
Failures here could affect applications that anoma developers can't forsee, and to the extent that these replicated state machines are supposed to keep reliable, confidential, permanent records, corrupt important data with repercussions that can't be undone.
For example, if we're confidentially tracking ownership of some resources with some zkps, and the anoma software screws up some transfer of ownership, there could be multiple parties trying to own and use a thing, and no way to disentangle what went wrong.

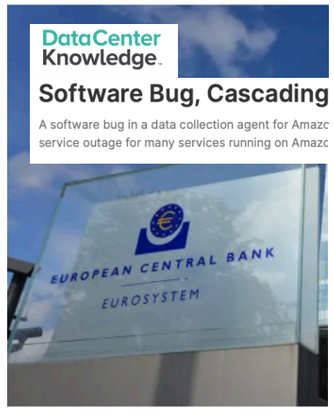
BACKGROUND

SOFTWARE INFRASTRUCTURE FAILURES

Unfortunately, traditional distributed system service infrastructure doesn’t always live up to its lofty goals
CLICK
we’ve come to accept that there our applications are going to fail, data will be leaked and corrupted by faulty infrastructure.

TIPS System Glitch Stalls European Banks

BY **PYMNTS** | AUGUST 10, 2021



Software Bug, Cascading

A software bug in a data collection agent for Amazc
service outage for many services running on Amazc

A glitch in the European Central Bank's payment system la
stranded during the vital hours near the close of business.

The Washington Post
Democracy Dies in Darkness

Major Internet outage along East Coast causes large parts of the Web to crash — again

By Timothy Bella

July 22, 2021 at 2:02 p.m. EDT

Another massive Internet outage along the East Coast struck significant online platforms Thursday, causing the high-traffic websites of companies such as Amazon, Airbnb, FedEx and Delta Air Lines to go dark.

According to the tracking website Downdetector, the websites of UPS, USAA, Home Depot, HBO Max and Costco were also among those affected. The websites of British Airways, GoDaddy, Fidelity, Vanguard and AT&T were among those loading slowly.

The cause of the outage, the latest major Internet outage this summer, was linked to Akamai Technologies, the global content delivery network based in Cambridge, Mass. Oracle, a cloud service provider, said its outage was the direct result of the Akamai disruption.

n

ts system returns to
crash

t unable to pay for goods and
Europe

ZDNET
/innovation

AWS cloud accidentally deletes customer data

downed the service on Sunday

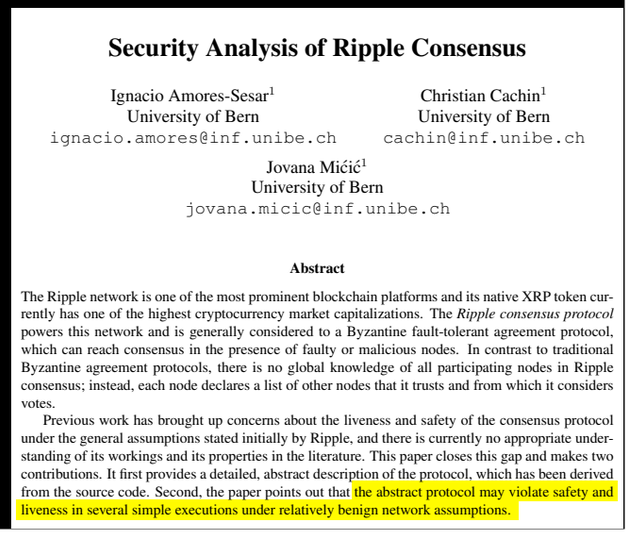
aws / aws-sdk-js

getObject + callback + createReadStream = corrupted file downloads #1628

BACKGROUND

BLOCKCHAIN DISTRIBUTED SYSTEM INFRASTRUCTURE FAILURES

- Consensus fails to guarantee:
  - Agreement
  - Termination



Security Analysis of Ripple Consensus

Ignacio Amores-Sesar[1]
University of Bern
ignacio.amores@inf.unibe.ch

Christian Cachin[1]
University of Bern
cachin@inf.unibe.ch

Jovana Mićić[1]
University of Bern
jovana.micic@inf.unibe.ch

Abstract

The Ripple network is one of the most prominent blockchain platforms and its native XRP token currently has one of the highest cryptocurrency market capitalizations. The *Ripple consensus protocol* powers this network and is generally considered to a Byzantine fault-tolerant agreement protocol, which can reach consensus in the presence of faulty or malicious nodes. In contrast to traditional Byzantine agreement protocols, there is no global knowledge of all participating nodes in Ripple consensus; instead, each node declares a list of other nodes that it trusts and from which it considers votes.

Previous work has brought up concerns about the liveness and safety of the consensus protocol under the general assumptions stated initially by Ripple, and there is currently no appropriate understanding of its workings and its properties in the literature. This paper closes this gap and makes two contributions. It first provides a detailed, abstract description of the protocol, which has been derived from the source code. Second, the paper points out that the abstract protocol may violate safety and liveness in several simple executions under relatively benign network assumptions.

Even though a lot of blockchain claims center around reliability and robustness, in practice the distributed systems involved are sometimes flawed.
For example, just looking at consensus, some relatively ambitious attempts at innovative blockchain consensus have had fundamental flaws that went unnoticed for years.
This paper is perhaps the most damning in a series of analyses that showed the Ripple consensus protocol simply didn’t guarantee agreement or termination under the originally stated assumptions.
Several tweaks were made to the protocol to fix each of these, and each turned out to be inadequate.
The Ripple chain had to be heavily modified, while running users’ applications (which were in turn in some amount of danger), in order to get some well-articulated guarantees.

BACKGROUND

BLOCKCHAIN DISTRIBUTED SYSTEM INFRASTRUCTURE FAILURES

- Consensus fails to guarantee:
  - Agreement
  - Termination
  - Fairness
  - Incentive-Compatibility

Uncle Maker: (Time)Stamping Out The Competition in Ethereum

Aviv Yaish
aviv.yaish@mail.huji.ac.il
The Hebrew University
Jerusalem, Israel

Gilad Stern
gilad.stern@mail.huji.ac.il
The Hebrew University
Jerusalem, Israel

Aviv Zohar
avivz@cs.huji.ac.il
The Hebrew University
Jerusalem, Israel

ABSTRACT

We present and analyze an attack on Ethereum 1's consensus mechanism, which allows miners to obtain higher mining rewards compared to their honest peers. This attack is novel in that it relies on manipulating block timestamps and the difficulty-adjustment algorithm (DAA) to give the miner an advantage whenever block races ensue. We call our attack *Uncle Maker*, as it induces a higher rate of uncle blocks. We describe several variants of the attack. Among these, one that is risk-free for miners.

Our attack differs from past attacks such as Selfish Mining [30], that have been shown to be profitable but were never observed in practice: We analyze data from Ethereum's blockchain and show that some of Ethereum's miners have been *actively* running a variant of this attack for several years without being detected, making

Table 1: A comparison of our attack and previous ones.

	Uncle Maker [this work]	Selfish Mining [30]	Stubborn Mining [57]	Coin Hopping [54]	Energy Equilibria [32, 38]	Stretch & Squeeze [74]
Analyzed on Ethereum	✓	✓	✓	-	-	✓
Does not require block withholding	✓	-	-	✓	✓	✓
Always more profitable than mining honestly	✓	-	-	-	-	-

Even the most popular chains like Ethereum and Bitcoin started out with unsubstantiated claims about important properties.
Bitcoin was supposed to tolerate less than half of participants being byzantine, but the usual 1/3rd bound reared its head when the selfish mining attack was published.

This paper has a whole table here of attacks unaccounted for in the original designs.
Some of them mess with other properties like Fairness, the notion that applications should get some kind of fair access to infrastructure, or
incentive-compatibility, the notion that infrastructure participants optimize their rewards by following the protocol.
What I particularly like about this incentive compatibility attack is that it's actually been running for years, before the protocol designers even worked out it was possible.

There have also been countless attacks on buggy smart contracts, wallets, and such, but I'm most interested in the distributed algorithm stuff.

STORY

ANOMA

- Distributed Systems Infrastructure
- Replicated State Machines (Blockchains)
  - Speed
  - Interoperability
  - Trust Models (e.g. proof-of-stake)
- P2P Overlays
- Intents and Solvers

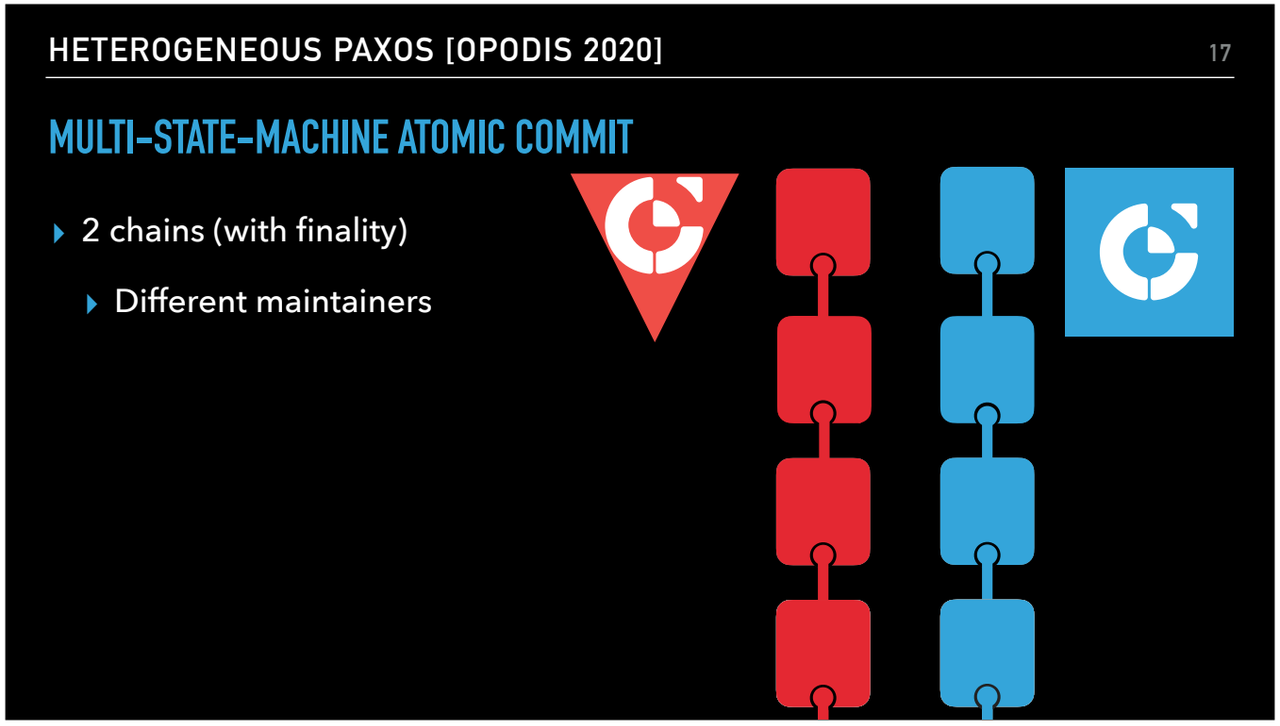
So we start working on this anoma ecosystem, which as I've said, aims to allow people to spin up their own replicated state machines,
And it's pretty clear to me anyway that this counts as infrastructure
So when we go to incorporate recent research into components, I figure we should try to formally express what guarantees they can provide and prove that we actually provide those.

STORY

ANOMA

- Distributed Systems Infrastructure
- Replicated State Machines (Blockchains)
  - Speed
  - Interoperability
  - Trust Models (e.g. proof-of-stake)
- P2P Overlays
- Intents and Solvers

For the sake of example, I want to start with an interoperability feature that builds on Heterogeneous Paxos.
The idea here is to take advantage of the high overlap between the trust models of various state machines, especially proof-of-stake chains, to do cross-state-machine atomic transactions.

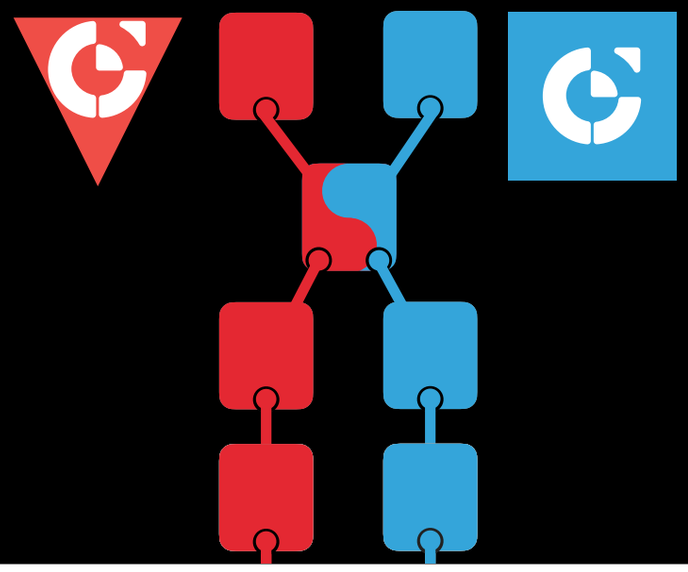


So I'm going to take a minute to explain what heterogeneous Paxos is, and what it's supposed to do.
Heterogeneous Paxos is a paper I published in 2020 with Andrew Myers, Xinwen Wang, and Robbert van Renesse.

The idea is to do multi-state-machine atomic commits.
So, suppose we have 2 blockchains, with some kind of finality, so they're each running a BFT consensus with quorums.
And as is traditional with blockchains, we'll call the batches of transactions they're committing "blocks."
So rather than doing some kind of 2-phase commit for a cross-chain atomic transaction, the thing we'd ideally want to do…

ONE BLOCK ON MULTIPLE CHAINS

- 2 chains (with finality)
  - Different maintainers
- One block for both chains

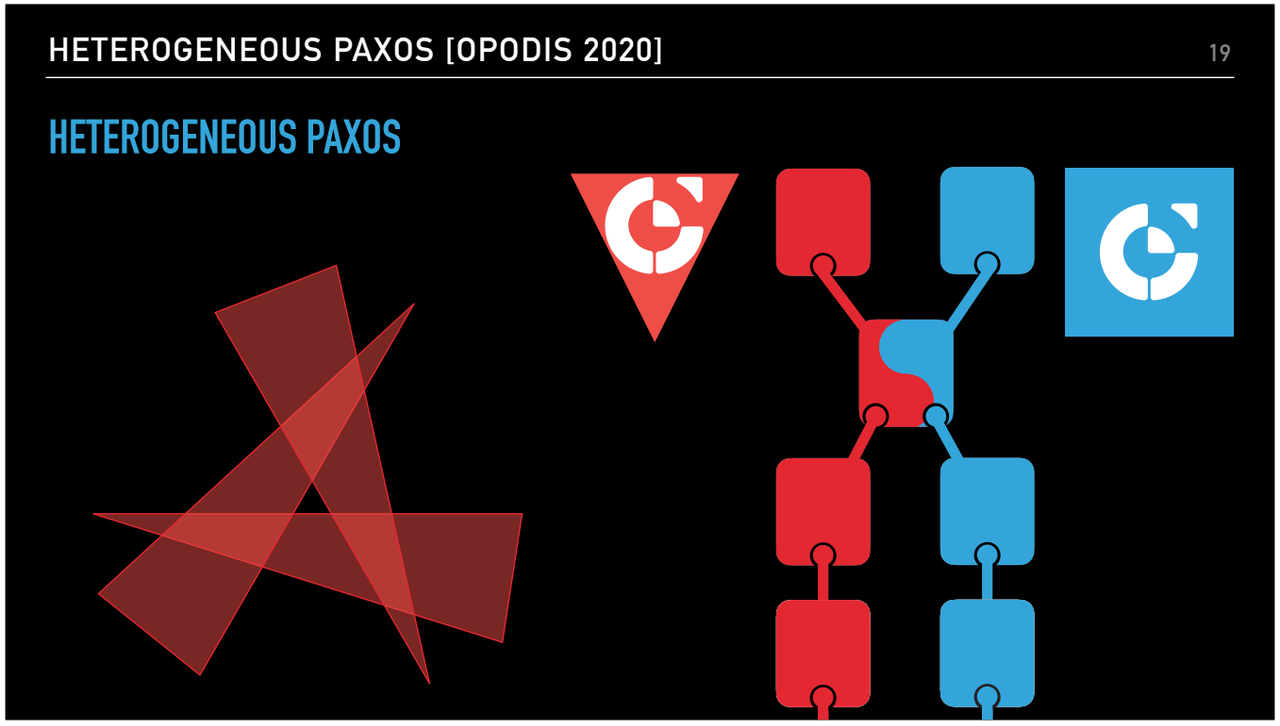


... is atomically commit one block containing transactions for both chains.
This batch could include transactions that represent sort of the “red half” and the “blue half” of a red-blue cross-chain atomic transaction.

In principle, we could use something like Stellar consensus here too.
We're using Heterogeneous Paxos in part because it's designed with a closed system model where we know everyone's safety and liveness requirements: something most chains define, for instance, in terms of stakeholders.
It also has some additional flexibility in complex failure scenarios, but I won’t get into that just now.

So to understand at a high level what guarantees Heterogeneous Paxos is supposed to make, we have to talk about quorums.
To borrow some of Lamport’s terminology, a quorum is a set of participants sufficient to make a learner decide.

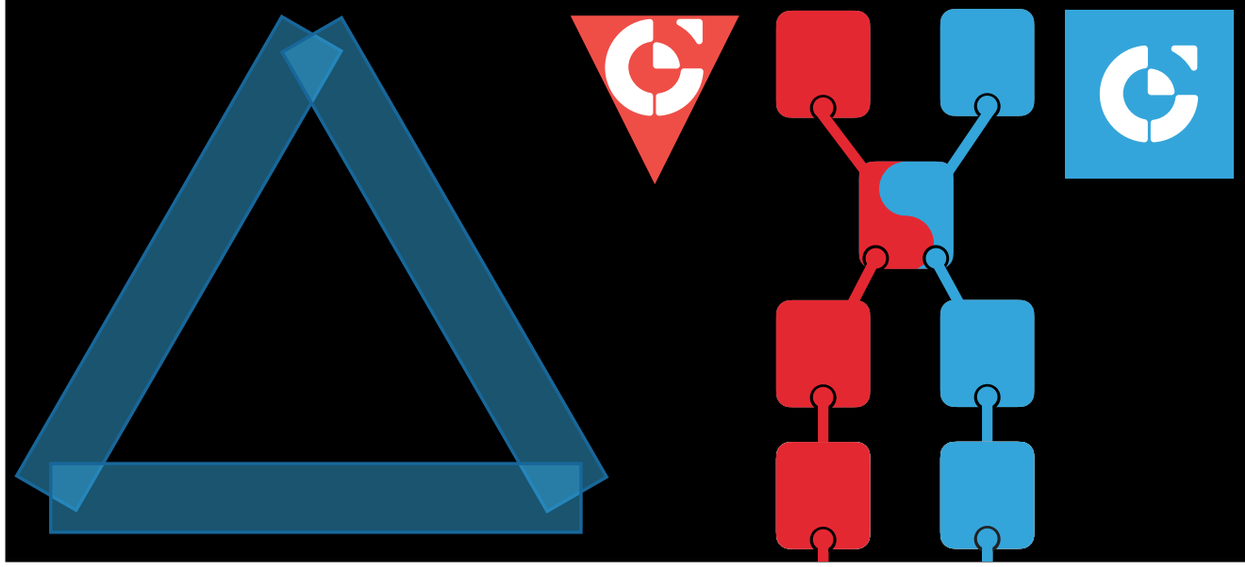
Each of these chains has their own independent trust assumptions.
We refer to that as having different “learners.”
So suppose that for red chain's learners, these quorums...



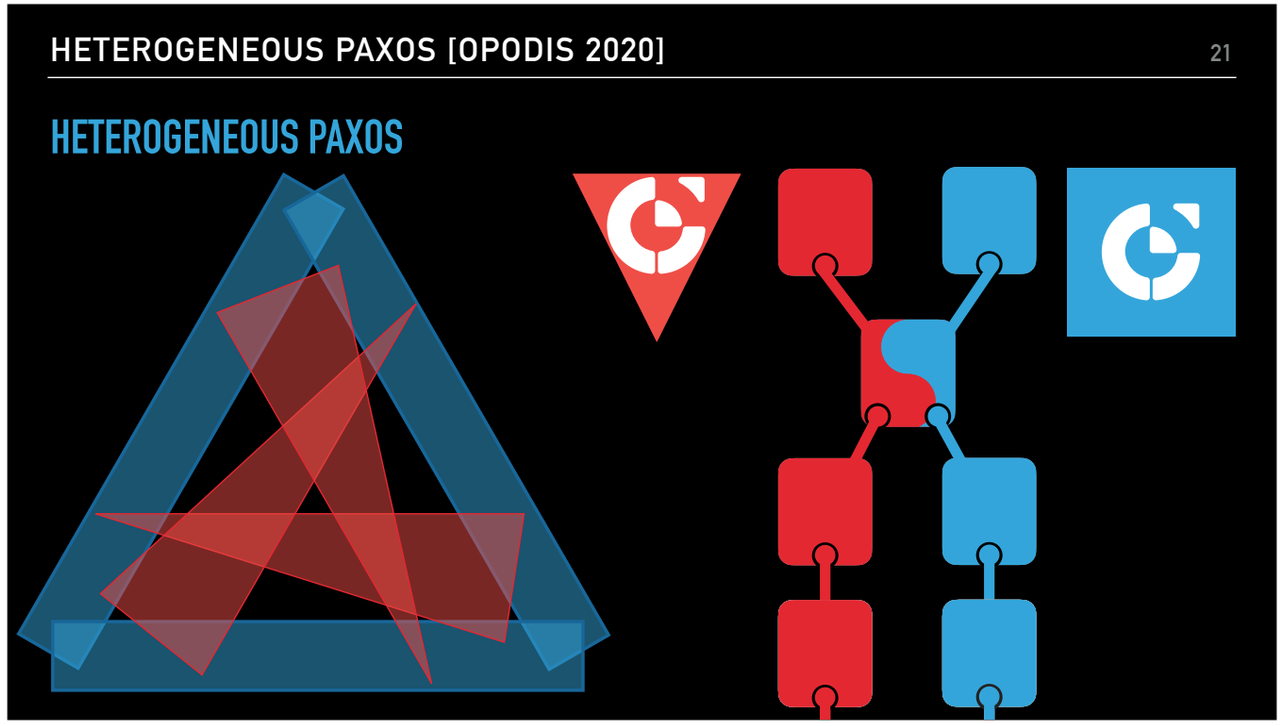
Look like this.
Crucially, they all intersect.
If you ever have two quorums that don't intersect, then it's possible for one person to hear from one quorum, and another person to hear from another quorum, and they could all operate completely separately: there's no way to be sure they agree.

What's more, there needs to be a safe acceptor in that intersection: if they're all unsafe (Byzantine), then they could act one way for the purposes of one quorum, and another way for the purposes of another quorum, and you're screwed.

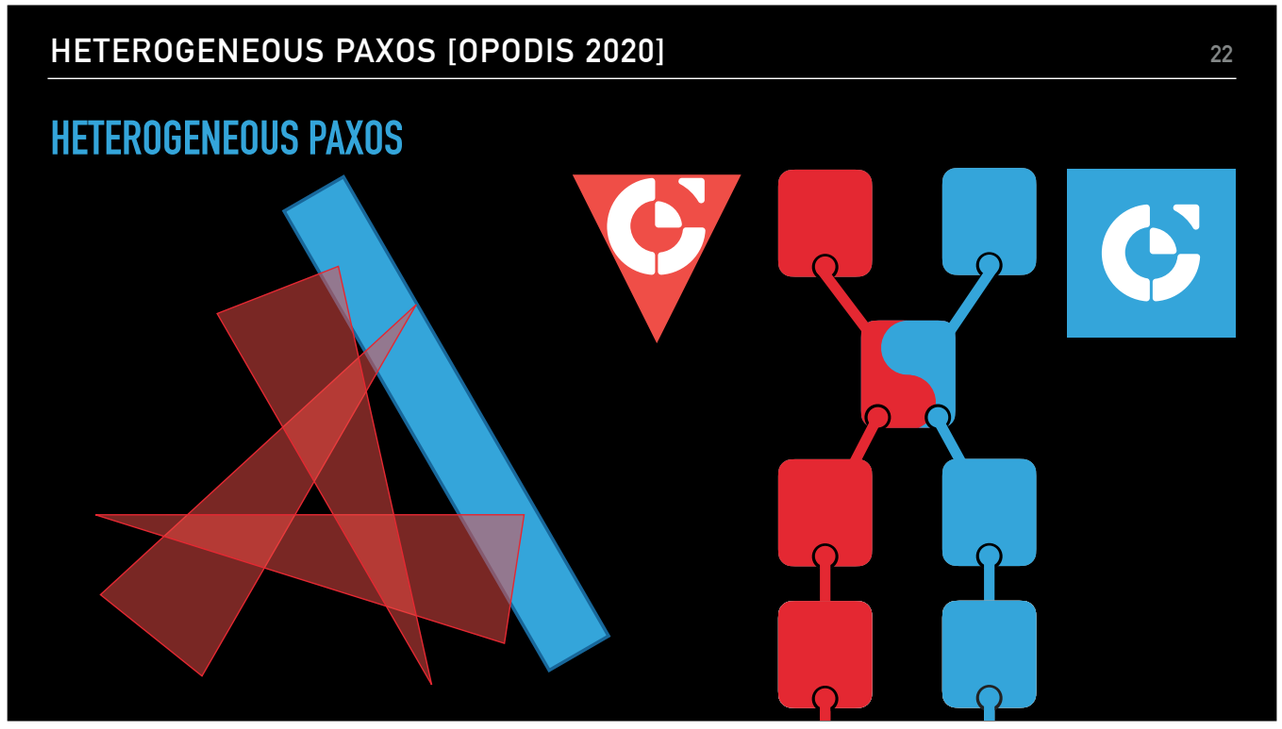
HETEROGENEOUS PAXOS



Let’s assume that Blue similarly has 3 quorums.

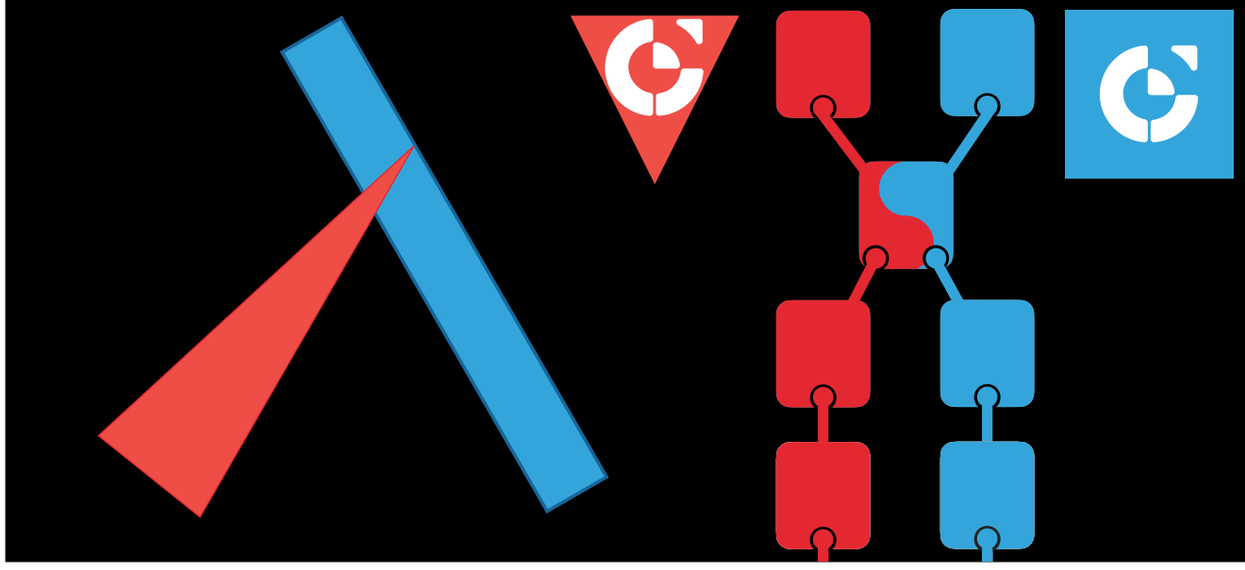


But things really get interesting if each of the Red quorums intersects each of the Blue quorums.
This is the situation when we can, as long as each of those red-blue intersections has a safe acceptor, do Heterogeneous Paxos.
We can commit a block, as far as blue is concerned, CLICK

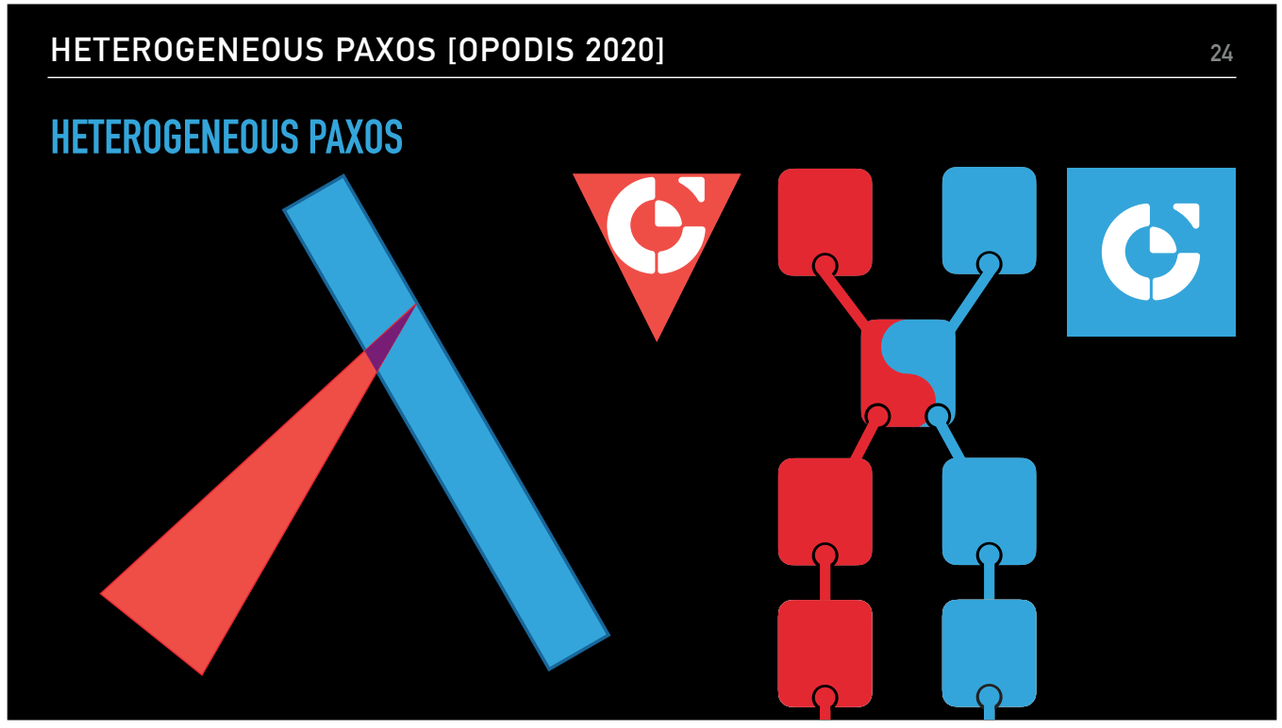


using only a blue quorum, and commit a block, as far as red is concerned, CLICK

HETEROGENEOUS PAXOS



using a red quorum, and red and blue will agree whenever CLICK

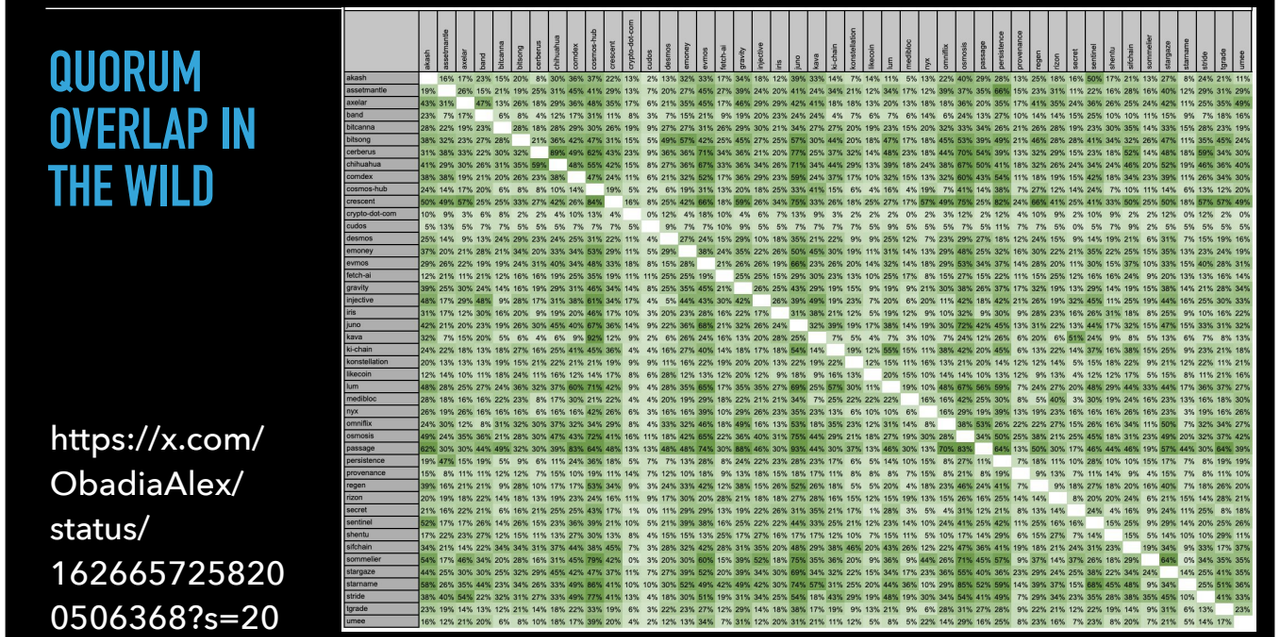


that intersection has a safe acceptor.

PAUSE

QUORUM OVERLAP IN THE WILD

https://x.com/ObadiaAlex/status/1626657258200506368?s=20

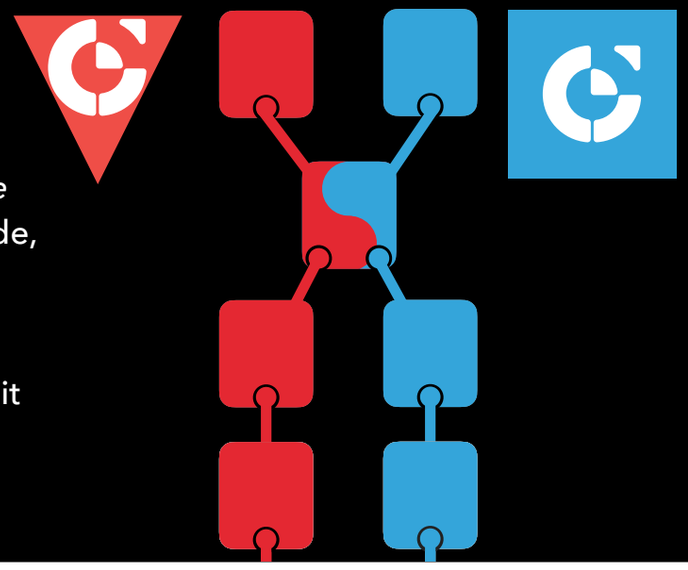


The interesting thing is that there is actually a huge amount of overlap between the validators of different (mostly proof-of-stake) blockchains: often half to two-thirds of the validators of large chains are shared.

We expect many anoma chains will likely have similar overlap.

FORMALIZATION OVERVIEW

- Formalized trust model: *Learner graph*
- Agreement: If 2 learners *require agreement*, and they both decide, they decide the same value
- Termination: If a learner has a quorum of correct participants, it eventually decides

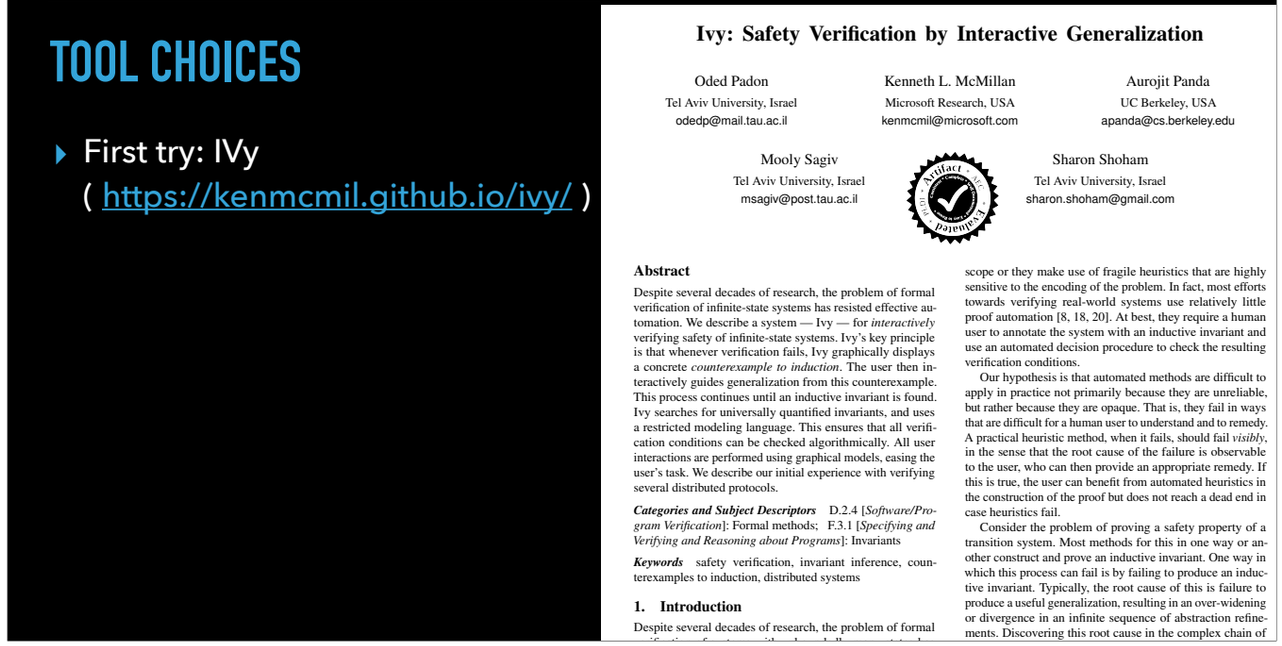


So, in the formal version of this, we develop a model for expressing this multi-learner trust, which we call a learner graph.
And this lets us express the precise conditions, the set of failures under which each pair learners must agree.
So, the safety condition of Paxos becomes: if 2 learners require agreement, and they both reach a decision, they decide on the same value.
Likewise, we can express each learner's liveness requirements, which is just "they have a quorum of correct participants,"
So the liveness condition becomes "if a learner has a correct quorum, they eventually reach a decision"

STORY

TOOL CHOICES

- First try: IVy
  (https://kenmcmil.github.io/ivy/)



Ivy: Safety Verification by Interactive Generalization

Oded Padon
Tel Aviv University, Israel
odedp@mail.tau.ac.il

Kenneth L. McMillan
Microsoft Research, USA
kenmcmil@microsoft.com

Aurojit Panda
UC Berkeley, USA
apanda@cs.berkeley.edu

Mooly Sagiv
Tel Aviv University, Israel
msagiv@post.tau.ac.il

Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com

Abstract

Despite several decades of research, the problem of formal verification of infinite-state systems has resisted effective automation. We describe a system — Ivy — for *interactively* verifying safety of infinite-state systems. Ivy's key principle is that whenever verification fails, Ivy graphically displays a concrete *counterexample to induction*. The user then interactively guides generalization from this counterexample. This process continues until an inductive invariant is found. Ivy searches for universally quantified invariants, and uses a restricted modeling language. This ensures that all verification conditions can be checked algorithmically. All user interactions are performed using graphical models, easing the user's task. We describe our initial experience with verifying several distributed protocols.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Invariants

Keywords safety verification, invariant inference, counterexamples to induction, distributed systems

1. Introduction

Despite several decades of research, the problem of formal

scope or they make use of fragile heuristics that are highly sensitive to the encoding of the problem. In fact, most efforts towards verifying real-world systems use relatively little proof automation [8, 18, 20]. At best, they require a human user to annotate the system with an inductive invariant and use an automated decision procedure to check the resulting verification conditions.

Our hypothesis is that automated methods are difficult to apply in practice not primarily because they are unreliable, but rather because they are opaque. That is, they fail in ways that are difficult for a human user to understand and to remedy. A practical heuristic method, when it fails, should fail *visibly*, in the sense that the root cause of the failure is observable to the user, who can then provide an appropriate remedy. If this is true, the user can benefit from automated heuristics in the construction of the proof but does not reach a dead end in case heuristics fail.

Consider the problem of proving a safety property of a transition system. Most methods for this in one way or another construct and prove an inductive invariant. One way in which this process can fail is by failing to produce an inductive invariant. Typically, the root cause of this is failure to produce a useful generalization, resulting in an over-widening or divergence in an infinite sequence of abstraction refinements. Discovering this root cause in the complex chain of

So, Aleks and I set out to find some tools to formalize Heterogeneous Paxos and prove these properties.
Our first try was in IVy, which, if you're not familiar with it, you can find here.

STORY

TOOL CHOICES

- First try: IVy
  (https://kenmcmil.github.io/ivy/)

Ivy: Safety Verification by Interactive Generalization

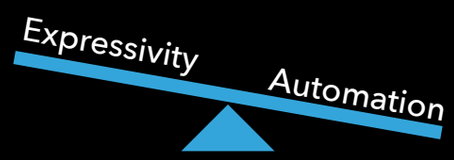
Oded Padon
Tel Aviv University, Israel
odedp@mail.tau.ac.il

Kenneth L. McMillan
Microsoft Research, USA
kenmcmil@microsoft.com

Aurojit Panda
UC Berkeley, USA
apanda@cs.berkeley.edu

Mooly Sagiv
Tel Aviv University, Israel
msagiv@post.tau.ac.il

Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com



In general, there is a trade-off in these kinds of tools between expressivity of the tool, and how easily it can automatically check stuff.
We started in Ivy because the automation is great.

STORY

TOOL CHOICES

- First try: IVy
  (https://kenmcmil.github.io/ivy/)
  - Protocol Steps in EPR
  - Invariants
    - Automatic, complete counterexample generation

Ivy: Safety Verification by Interactive Generalization

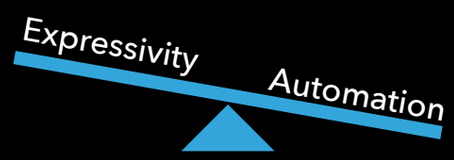
Oded Padon
Tel Aviv University, Israel
odedp@mail.tau.ac.il

Kenneth L. McMillan
Microsoft Research, USA
kenmcmil@microsoft.com

Aurojit Panda
UC Berkeley, USA
apanda@cs.berkeley.edu

Mooly Sagiv
Tel Aviv University, Israel
msagiv@post.tau.ac.il

Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com



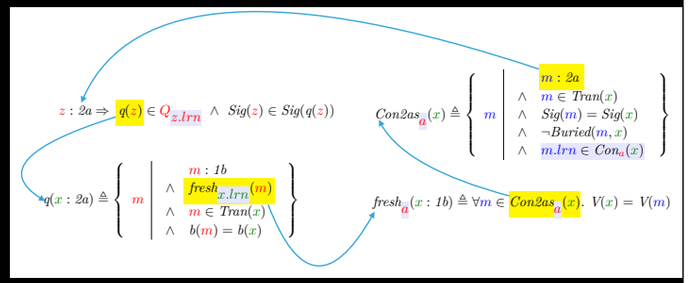
The idea in IVy is that you can specify allowed protocol steps in Effectively Propositional Relations, a decidable logic.
Then you specify an invariant, and IVy checks if the steps preserve the invariant.
If they don’t, you get a counterexample: an allowed before state and a disallowed after state.

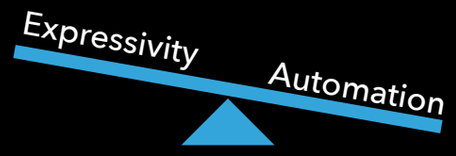
This was really cool, and for a lot of protocols Ivy is a great choice, but that expressivity trade-off turned out to be a problem for Heterogenous paxos.

STORY

TOOL CHOICES

- First try: IVy
  (https://kenmcmil.github.io/ivy/)
  - Protocol Steps in EPR
  - Invariants
    - Automatic, complete counterexample generation



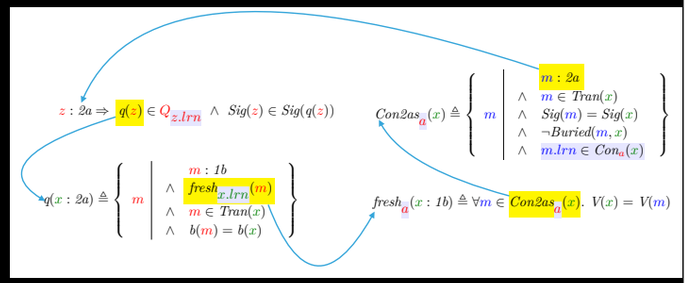


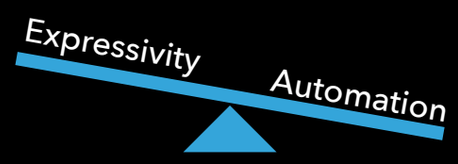
For example, there are some complex mutually recursive functions in the definitions in Heterogeneous Paxos, and they don’t fit in EPR.

STORY

TOOL CHOICES

- First try: IVy
  (https://kenmcmil.github.io/ivy/)
  - Protocol Steps in EPR
  - Invariants
    - Automatic, complete counterexample generation





- Simpler protocol: safe
  - Not live, easy to miss that (hard to express as invariant)

So we ended up actually developing a conceptually simpler protocol, based on existing formulations of byzantized Paxos,
And we could prove that safe.

But it turned out that we had cut something important, and we had produced a protocol that could always take steps, but was never guaranteed to reach a decision.
We had broken liveness, which is not intuitive to express in these check-able invariants.

STORY

TOOL CHOICES

WISH LIST

- Expressive
  - Liveness properties
  - Recursive functions

So we went looking for another tool, now with the beginnings of a wish list.
We wanted something with a different expressiveness / automation trade/off,
 that could express our complicated protocol, with complex recursive functions, and
Ideally something that could reason explicitly about liveness, so we didn’t make that mistake again.

So we went looking for another tool, now with the beginnings of a wish list.
We wanted something that could express our complicated protocol, with complex recursive functions, and
Ideally something that could reason explicitly about liveness, so we didn’t make that mistake again.

It looked like the popular choice was TLA+ and the TLA+ proof system, so that’s what we tried next.

There's a lot to like about TLA+.
Lamport's introductory material for how to model stuff in TLA+ is really nice.
There's less on PlusCal, although it's not hard to pick up.
Some of those same recursive functions, however, uh, we gave up on trying to state those in PlusCal, so we ended up writing our initial spec in TLA+.

The proof system, on the other hand, is, well Aleks once described it as "Like Coq, but with absolutely no convenience features."
It can do a lot, although it can be quite tedious.

There's a lot to like about TLA+.
Lamport's introductory material for how to model stuff in TLA+ is really nice.
There's less on PlusCal, although it's not hard to pick up.
Some of those same recursive functions, however, uh, we gave up on trying to state those in PlusCal, so we ended up writing our initial spec in TLA+.

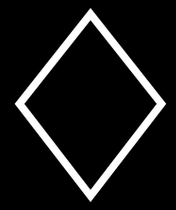
The proof system, on the other hand, is, well Aleks once described it as "Like Coq, but with absolutely no convenience features."
It can do a lot, although it can be quite tedious.

Nevertheless, we did eventually produce a proof of safety, which is great.

STORY

LIVENESS IS HARD

- Not a lot of proofs use “eventually” operator



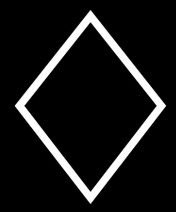
Proof System

Unfortunately, it turns out that proving liveness is way harder.
There aren’t nearly so many examples to go on using the eventually operator.
I mean, it exists in the logic and the proof system, but it’s not easy to reason about, and there aren’t a lot of libraries or examples or texts to lean on.

STORY

LIVENESS IS HARD

- Not a lot of proofs use “eventually” operator
- 13 consecutive synchronous periods
  - All messages sent in one arrive in the next



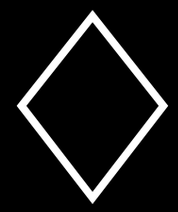
Proof System

The pen-and-paper liveness proof relies on a period of synchrony, with 13 consecutive periods, in which all messages sent in one arrive in the next.

STORY

LIVENESS IS HARD

- Not a lot of proofs use “eventually” operator
- 12 consecutive synchronous periods
  - All messages sent in one arrive in the next
  - As opposed to Lamport’s “final ballot” assumption



Proof System

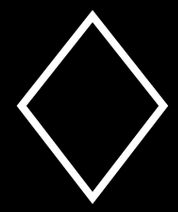
By way of contrast, Lamport’s liveness proof for Paxos relies on a “final ballot” assumption, asserting there will be some final ballot, and so long as it exists, a decision is always possible.

We don’t have that.
In the worst case, Heterogeneous Paxos requires 3 consecutive “final ballots.”

STORY

LIVENESS IS HARD

- Not a lot of proofs use “eventually” operator
- 12 consecutive synchronous periods
  - All messages sent in one arrive in the next
  - As opposed to Lamport’s “final ballot” assumption
- Model Checking: State Explosion



Proof System

The solution it seems a lot of people arrive at is to model check some example cases.
Unfortunately, we got a state explosion almost immediately.
The messages structure of Heterogeneous Paxos is potentially complex, so beyond a handful of participants, and a single learner with I think one quorum, it just wasn’t practical to check them all.

STORY

COMMUNITY KNOWLEDGE

This is one place where there seems to be some knowledge in the community, hopefully in this room, that just isn’t obvious to, say, a team of industrial researchers with separate backgrounds in formal tools and distributed protocols.

STORY

COMMUNITY KNOWLEDGE

- Liveness to Safety Transformations
  - Which?
  - Correct? Automated?
- Ghost Variables
  - Countdown
  - Maintain model correctness?

For example, there are several ways of transforming liveness properties into safety properties, but exactly which should you use, and which produce a safety property I can have any intuition reasoning about?
How can I be sure I got it right?
There are some tools out there with automated safety to liveness transformations, including I think apalache, which we haven’t tried yet.
This kind of thing is, of course, not the kind of stuff covered in, for example, Lamport’s specifying systems book.

STORY

COMMUNITY KNOWLEDGE

- Liveness to Safety Transformations
  - Which?
  - Correct? Automated?
- Ghost Variables
  - Countdown
  - Maintain model correctness?

I was talking to Giuliano, and we started talking about what kind of ghost variables we could introduce so that our liveness properties might become an invariant.
We might, for instance, have a countdown variable that “starts” at some unknown point after GST, and every step thereafter has to decrement the countdown, and then the crucial invariant is that if the countdown is below zero, a decision exists.
Of course then you also have to prove that the ghost variables don’t affect the actors in the model, and the proof about the invariant is itself rather complex, but it could work.
We also have to prove that some kind of step is always enabled, which is another invariant.
But it can be done.
My point is just that there are some things that may seem obvious in the research community, but aren’t sufficiently easy to pick up in industry, and that alone may be holding back some serious impact.

STORY

MODELING HETEROGENEOUS PAXOS

- TLA+ Model
- TLAPS Safety Proof
- No liveness proof …

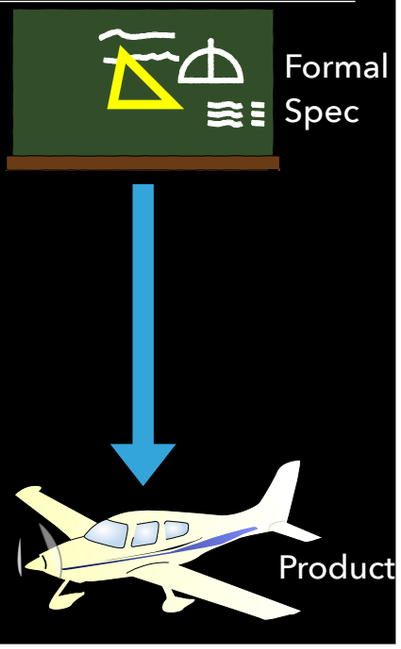
WISH LIST

- Expressive
  - Liveness properties
  - Recursive functions

So anyway, there we are, with our TLA+ model, and our safety proof, and while we’re rethinking our tooling, we went to re-formulate the protocol itself, hoping to make the protocol more efficient…

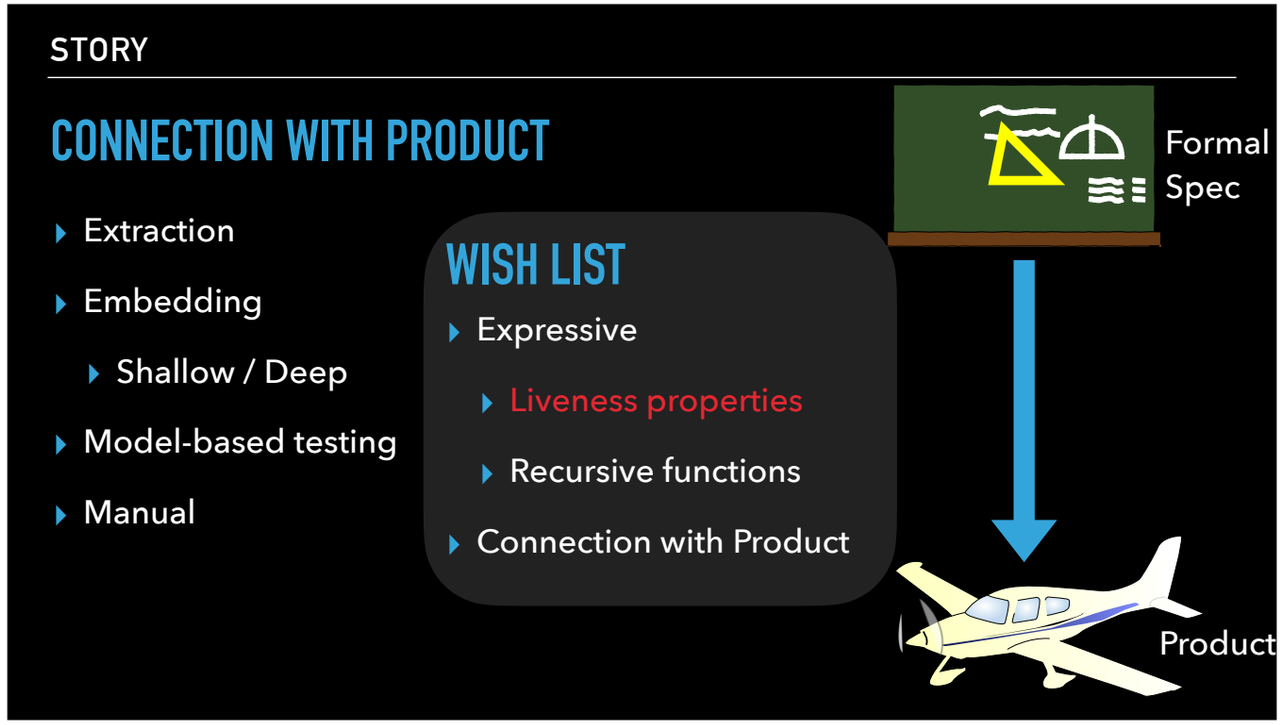
STORY

CONNECTION WITH PRODUCT



And while we’re thinking about efficiency and possibly choosing different formal tools, we figure as long as we’re nailing down our optimized protocol, we’d like to connect it to the final implementation, the product.
After all, the whole point is supposed to be that some kind of formal spec informs the product.
Ideally, guarantees we’d made about the formal spec would somehow translate to guarantees we’d make about the product.

So we’ve got to add that to our wishlist.



This is where people talk about extracting code directly from the model to form the product, or part of it.
Alternatively, you could embed the product code in your model somewhere, with various notions of shallow or deep embeddings.
Still other techniques involve automatically deriving test cases from the model, to test stuff about the product.

- less complete, but better than nothing.

Finally, one might talk about “Manual” connection, which basically means getting humans to examine both the model and the code, and check that the two correspond in some meaningful way.

So how will anoma production software be written?
Does our model reflect that?

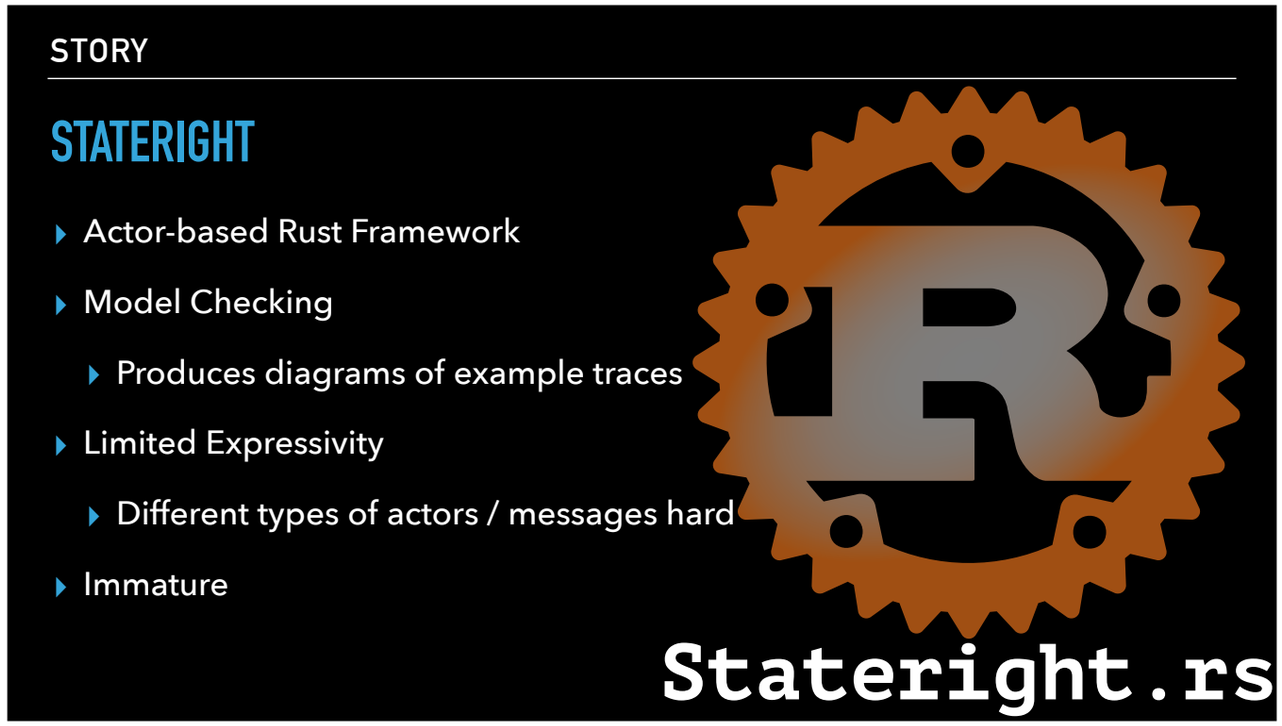
Well, in practice, if you're coding a distributed algorithm right now, you're writing code for each actor.
And of course you can indeed reason about actors in TLA+,
Although it doesn't talk about those natively, so you don't automatically get that, for instance, all decisions an actor needs to make can be made locally.

The PlusCal language, which translates into TLA+, essentially is coding for actors directly, but you actually lose some expressiveness.
If I recall correctly, some of the recursive functions we wanted to encode had to be done directly in TLA+.

I'd note that we did eventually come up with a simplification of heterogeneous Paxos comfortably expressible in PlusCal, but that's not really the point right now.

Anyway, as far as production-level coding is concerned, languages like Elixir or Erlang talk about actors fairly directly, and it’s pretty common to code them in rust or languages like that.

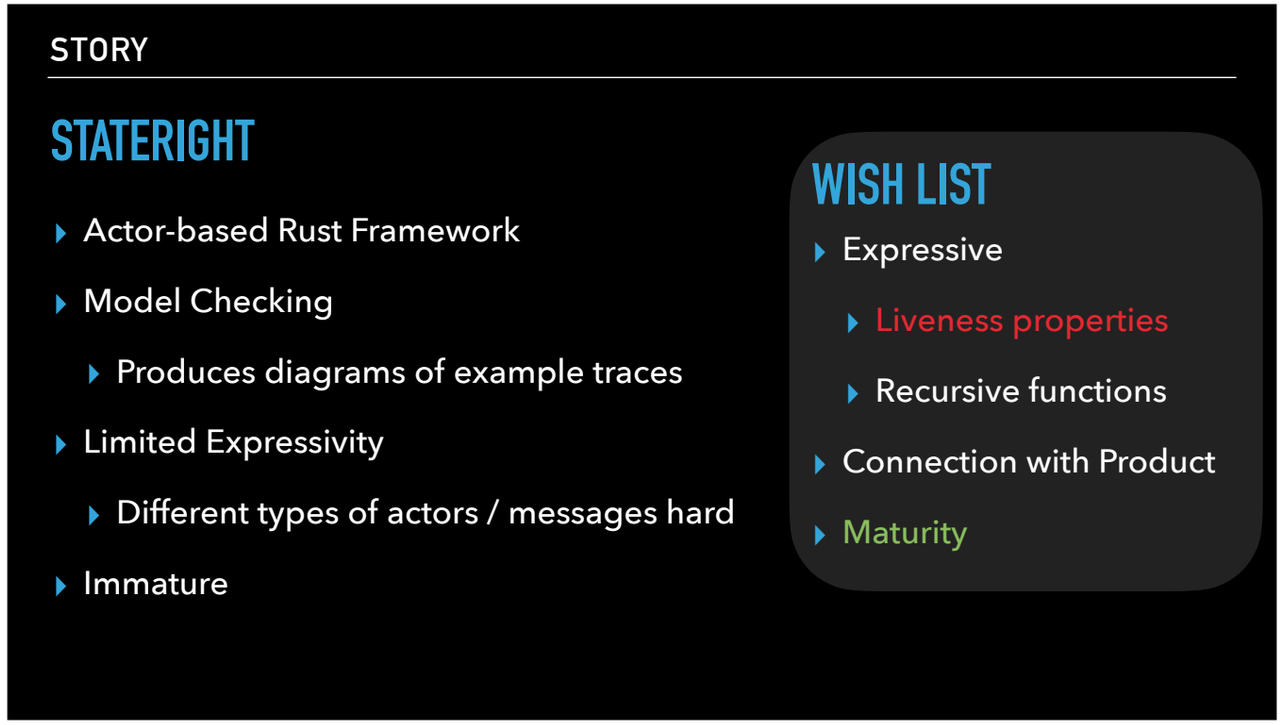
So we went looking for an actor-based framework that would let us formally reason about engineered production code.



So that’s when our eye fell on StateRight.

Stateright is an actor-based framework in rust.
You can write your production code, and also model-check it.
Even if you can’t exhaustively model check it, it produces diagrams of example traces, with an inspection interface that lets you click around to see everyone’s states at various points in the trace, which is pretty slick.
We had faced a state explosion problem before, but we thought maybe we could invest time into making the state space less explodey if we had a tool we liked enough.

We did encounter some problems with expressivity, however.
For instance, the system is parameterized with a message type and an actor type, so if different actors want to send different types of messages to each other, everything has to be catalogued in big old switch statements.
There was some other boiler plate stuff to get out of the way, and a few bugs in actually running it as a distributed system.
In the end, we left Stateright behind because it wasn’t mature enough: it doesn’t have the polish, the tools, or the community to support industrial use.



Ultimately, that adds another element to our wish-list: maturity.
It’s hard to build an industrial product on a framework that’s going to change, or that no one knows how to use properly.
The exception would be if you’re willing to take ownership of it yourself, and maintain or improve it yourself.

STORY

LOTS OF TOOLS

WISH LIST

- Expressive
  - Liveness properties
  - Recursive functions
- Connection with Product
- Maturity

Modeling	Proving	Connecting to Product	Notes
TLA+	Model Checking: - TLC or - Apalache Theorem Proving: - TLA+ Proof System	Model Based Testing: - Modelator	-relatively well established track record -kind of old -no native types, but maybe Snowcat can help -TLAPS is annoying to use, and hard to prove liveness - Heterogeneous Paxos Proofs
Quint	Model Checking with Apalache and maybe other stuff via compiling to TLA+	maybe via compiling to TLA+	In development at Informal Systems
Petri Nets: - CPN IDE - GreatSPN	Model Checking	1. might be able to use some of the SML code 2. MBT/CPN: model based testing	Node transitions written in simplified SML/NJ
StateRight	Model Checking	Model is Product? or some of the code is re-used?	IIRC, we thought it was “not ready” because of limits in type support
Dafny	Theorem Proving with help from model checker	Model is Product	- relatively well established - IronFleet - not sure of byzantine example - I don’t know much about it
Coq	Theorem Proving	Model is Product: - Verdi Maybe prove stuff about Rust code with RustBelt and Iris	- well established for not-distributed systems - nothing in Coq is meant to run fast.
Idris2	Theorem Proving	Model is Product	- research code - never been used in a distributed system
Agda	Theorem Proving	Model is Product?	- well established for not-distributed systems - I don’t know of any distributed systems Agda projects
Ivy	Theorem Proving with help from a model checker		- research code - EPR part was easy to use - bridging not-EPR to EPR part was kind of tricky. - encoding properties in EPR tricky - no “native” liveness reasoning
Isabelle/HOL	Theorem Proving	code generation?	- well established - Anthony describes it as easiest for proving theorems
Lean	Theorem Proving	Model is Product?	- well established - more popular for “pure” math - not aware of distributed systems projects in Lean
Maude	Theorem Proving (it would call it “formal reasoning”)	???	- rewriting logic - over 20 years old - I don’t know much about Maude

We actually ended up looking at a lot of tools to various degrees, and I can’t say we found any one that was clearly the way to go.

STORY

LOTS OF TOOLS

WISH LIST

- Expressive
  - Liveness properties
  - Recursive functions
- Connection with Product
- Maturity

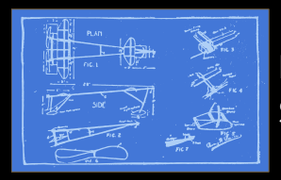
Modeling	Proving	Connecting to Product	Notes
TLA+	Model Checking: - TLC or - Apalache Theorem Proving: - TLA+ Proof System	Model Based Testing: - Modelator	-relatively well established track record -kind of old -no native types, but maybe Snowcat can help -TLAPS is annoying to use, and hard to prove liveness - Heterogeneous Paxos Proofs
Quint	Model Checking with Apalache and maybe other stuff via compiling to TLA+	maybe via compiling to TLA+	In development at Informal Systems
Petri Nets: - CPN IDE - GreatSPN	Model Checking	1. might be able to use some of the SML code 2. MBT/CPN: model based testing	Node transitions written in simplified SML/NJ
StateRight	Model Checking	Model is Product? or some of the code is re-used?	IIRC, we thought it was “not ready” because of limits in type support
Dafny	Theorem Proving with help from model checker	Model is Product	- relatively well established - IronFleet - not sure of byzantine example - I don’t know much about it
Coq	Theorem Proving	Model is Product: - Verdi Maybe prove stuff about Rust code with RustBelt and Iris	- well established for not-distributed systems - nothing in Coq is meant to run fast.
Idris2	Theorem Proving	Model is Product	- research code - never been used in a distributed system
Agda	Theorem Proving	Model is Product?	- well established for not-distributed systems - I don’t know of any distributed systems Agda projects
Ivy	Theorem Proving with help from a model checker		- research code - EPR part was easy to use - bridging not-EPR to EPR part was kind of tricky. - encoding properties in EPR tricky - no “native” liveness reasoning
Isabelle/HOL	Theorem Proving	code generation?	- well established - Anthony describes it as easiest for proving theorems
Lean	Theorem Proving	Model is Product?	- well established - more popular for “pure” math - not aware of distributed systems projects in Lean
Maude	Theorem Proving (it would call it “formal reasoning”)	???	- rewriting logic - over 20 years old - I don’t know much about Maude

Ultimately, I think connection with product was the most difficult to satisfy.
When you prove stuff about code, and then extract it to become the product, you risk making product development extremely brittle: every change can require changing the proofs.
In contrast, proofs about abstract models can be elegant, but systems for connecting them with products are largely immature and/or incomplete.

In the end, we opted for a manual connection, to maintain flexibility in both modeling and production.
To formalize their relationship, and try to maintain solid checks, we developed a specification system.

STORY

COMMUNICATION WITH ENGINEERS

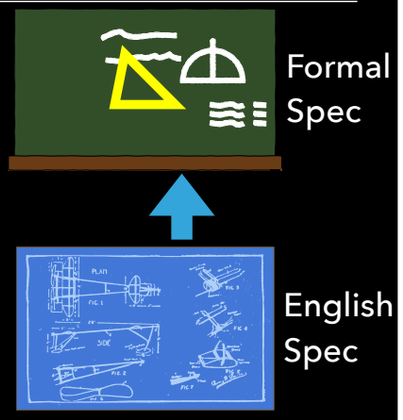


English
Spec

There would be an English Spec, which explains in detail how the system is supposed to work.

STORY

COMMUNICATION WITH ENGINEERS

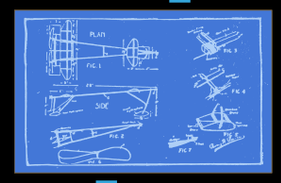


From this we would build a formal spec, which would then prove that the system works properly.

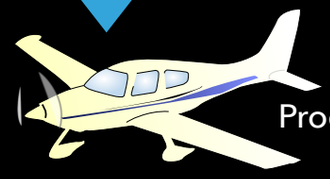
STORY

COMMUNICATION WITH ENGINEERS

Formal Spec



English Spec

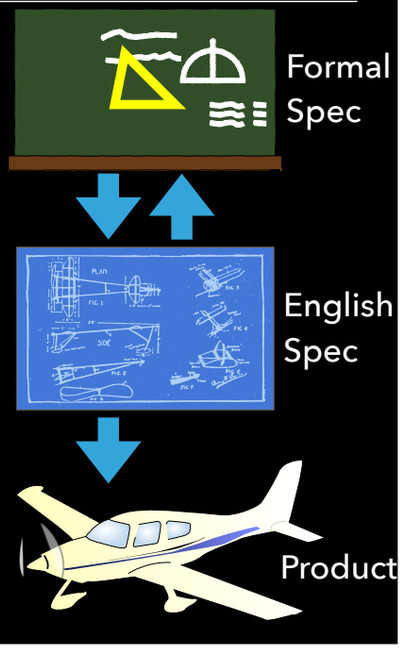


Product

Engineers would also build a product, which would actually implement the system.

STORY

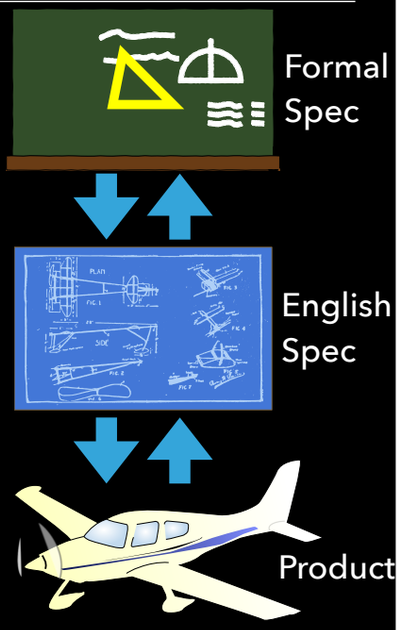
COMMUNICATION WITH ENGINEERS



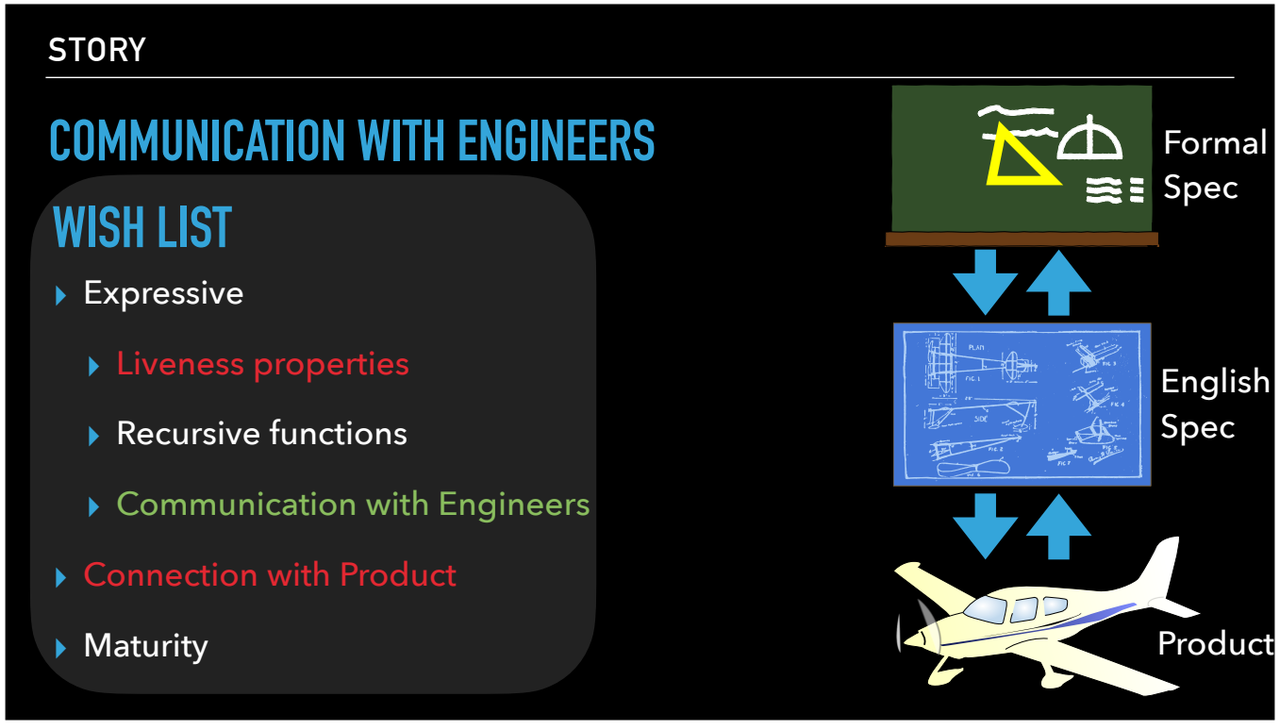
Of course, in practice, working on the formal system may prompt corrections to the English spec, and

STORY

COMMUNICATION WITH ENGINEERS



Working on the product may prompt optimizations and explanations in the English spec.

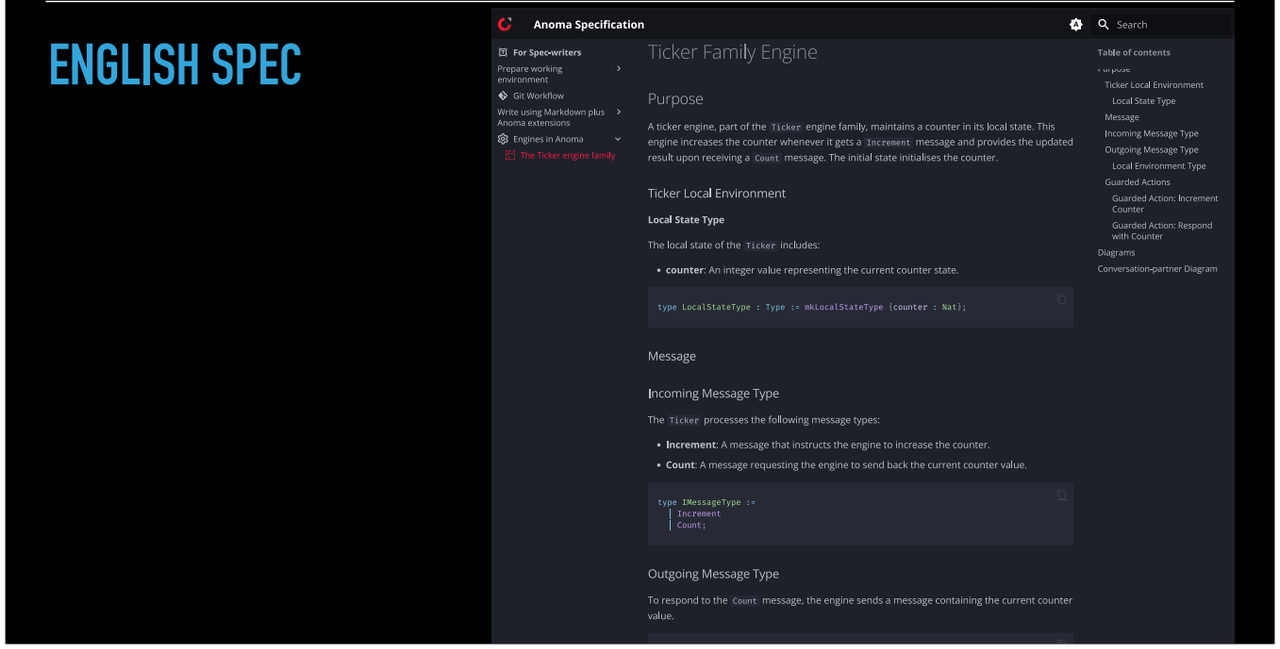


This does mean we have to add something to our wish list: we want engineers to be able to examine the formal spec, and see if and how it corresponds to the English spec, and even the product.

So, how do we build an English spec that carries this weight? That ensures a correspondence with the formal spec, and communicates with the engineers?

CURRENT STATE

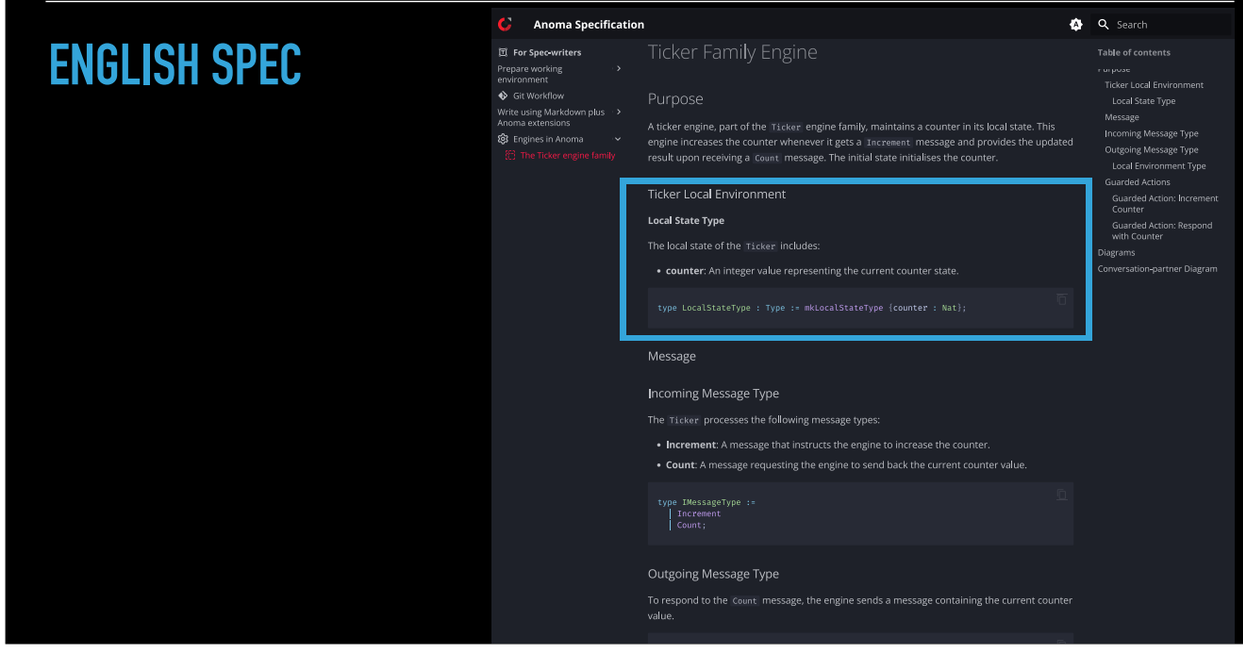
ENGLISH SPEC



Well, our format, at the moment, looks something like this.
We have a page for each type of actor.

CURRENT STATE

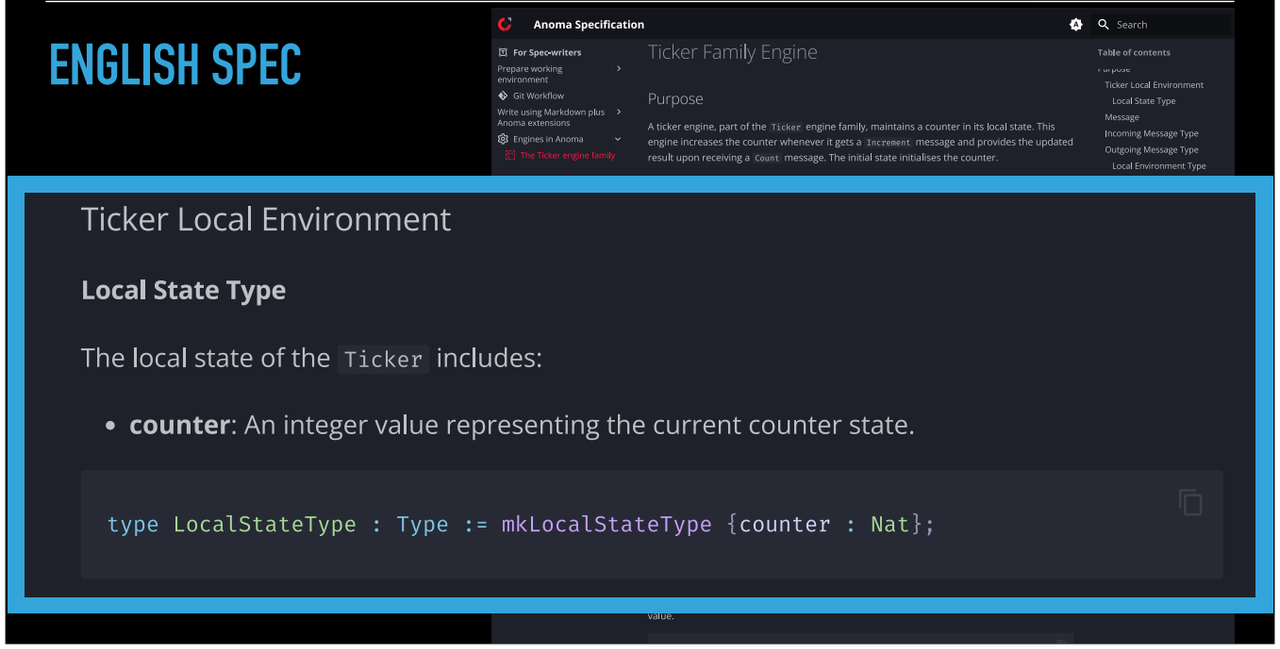
ENGLISH SPEC



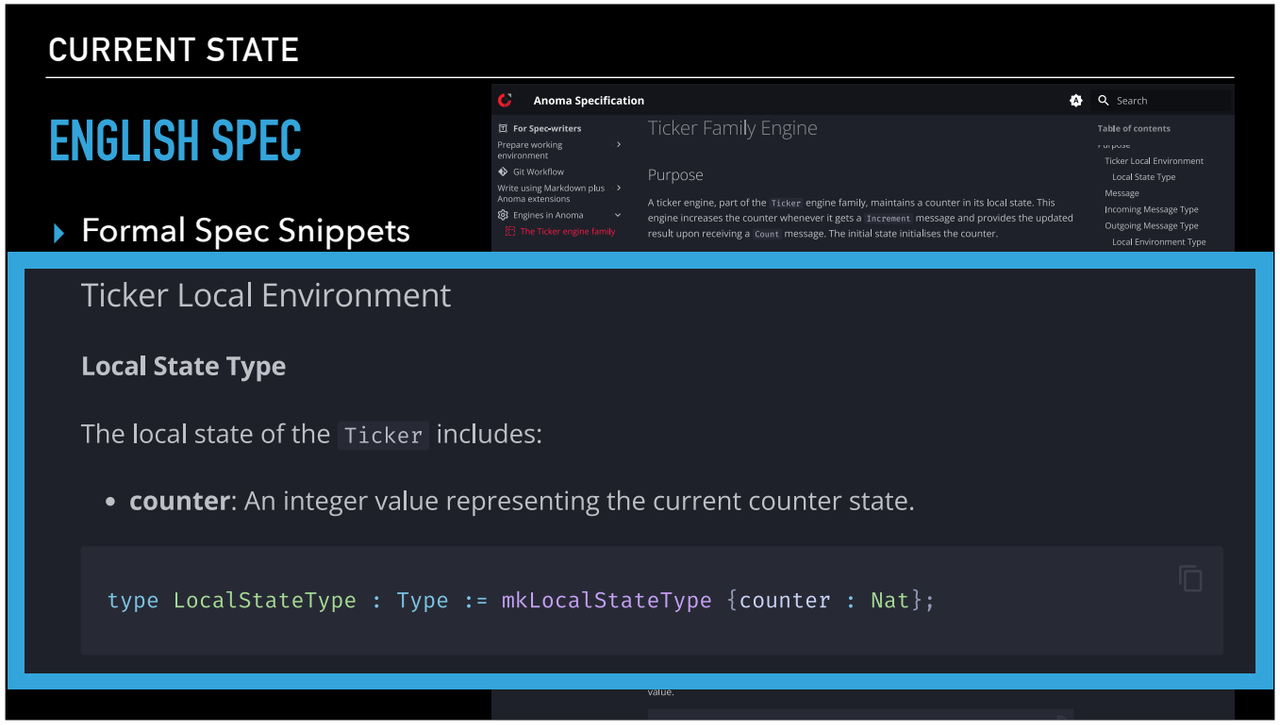
Each actor page specifies things like, what local state does the actor keep?

CURRENT STATE

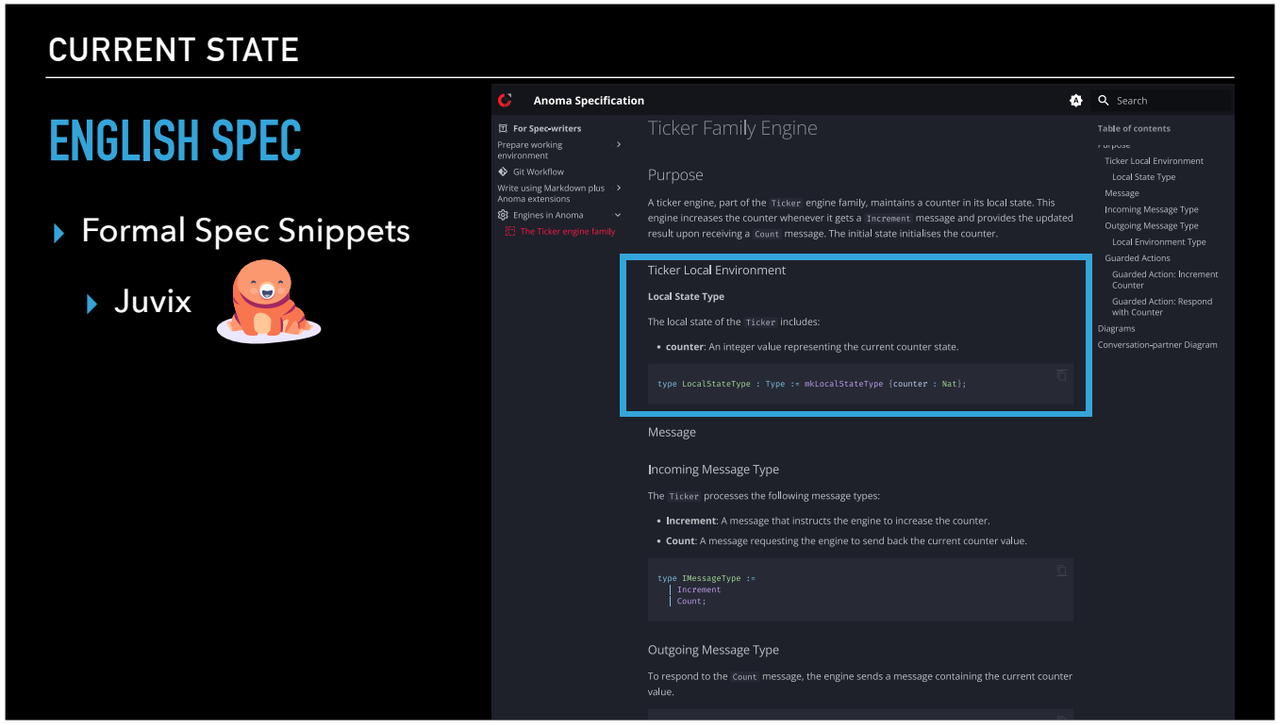
ENGLISH SPEC



Which includes an English language description, and then a …

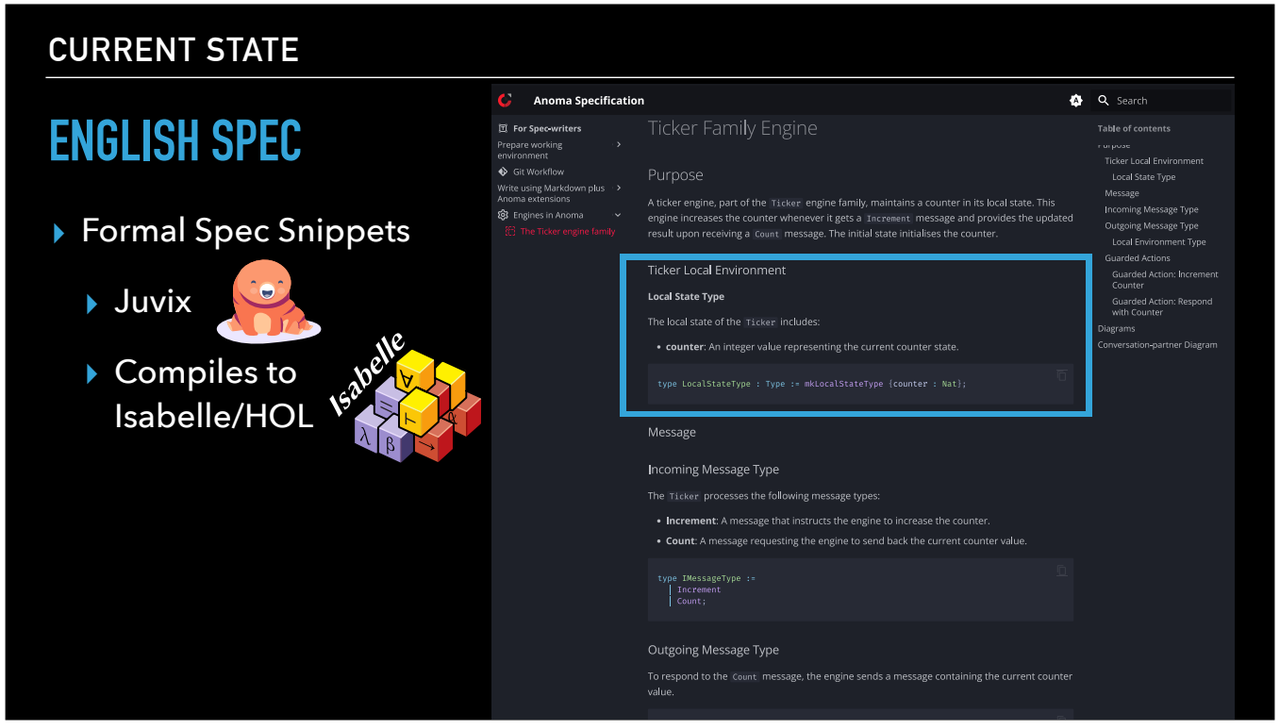


Snippet of formal spec, in this case detailing the local state that this “Ticker” engine keeps, which is a Nat.

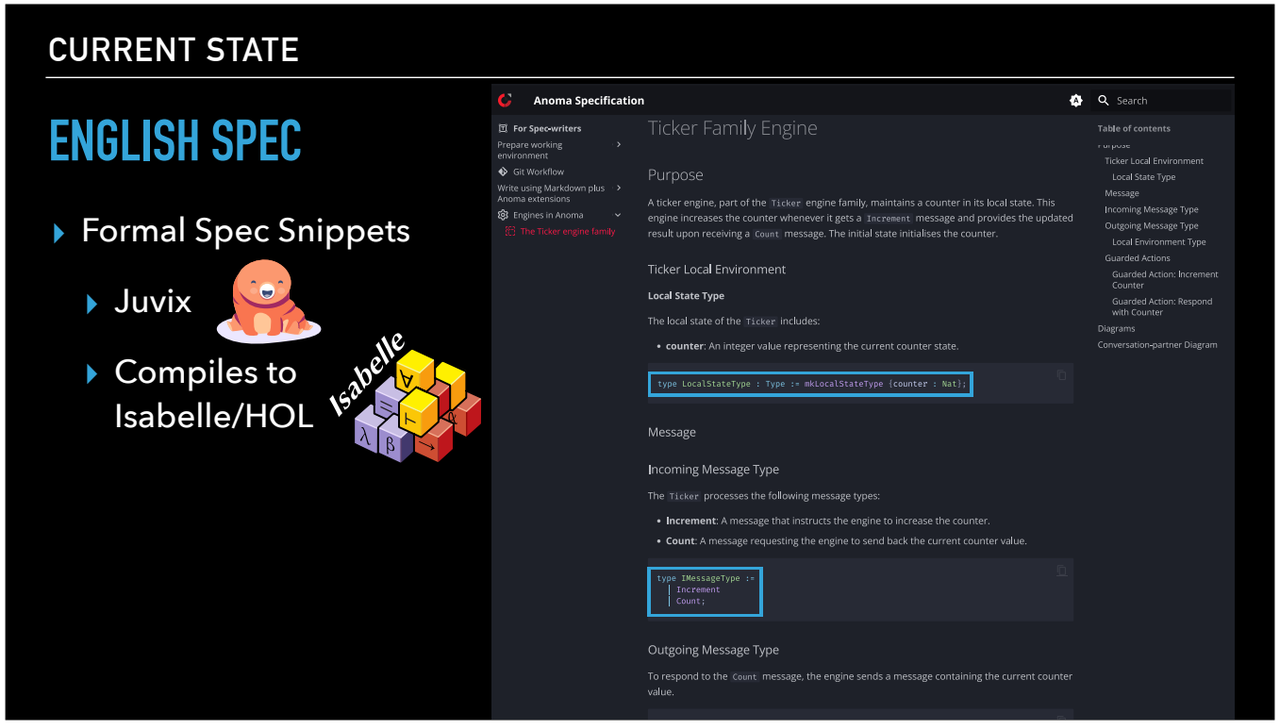


These snippets are written in Juvix.
I mentioned a moment ago that it's hard to work with immature tools unless you're willing to take them on in-house.
Well, Juvix is our in-house functional programming language, with great ambitions.
It lets us write some very ML-like types and declarative functions in this spec, which our engineers like.
One of the ways we're trying to mature it *is* using it here.

That said, we can't prove things in Juvix, at least not yet, so we set out to compile these snippets directly into an existing formal verification language.
For this, we need something fairly general, and well-typed.
We could have chosen Agda or Coq or Lean, and in principle it doesn't matter too much, since this format already dictates how we're formalizing things anyway.



But we chose Isabelle/HOL, because of the mature general-purpose provers, we liked its proof automation, and the language used for locales.



So, these snippets here at the top of the page

CURRENT STATE

ENGLISH SPEC

- Formal Spec Snippets
  - Juvix
  - Compiles to Isabelle/HOL


```
type LocalStateType : Type := mkLocalStateType {counter : Nat};
```

```
type IMessageType :=
  | Increment
  | Count;
```

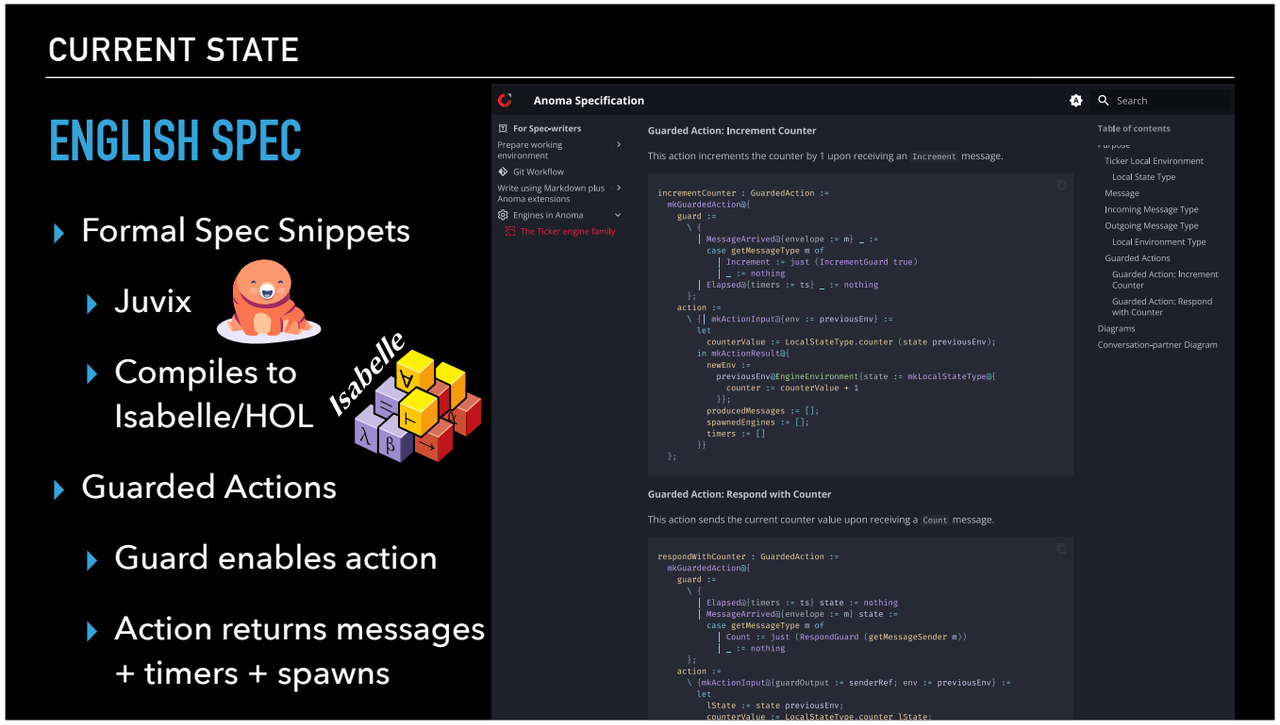
Ticker.thy

```
theory Ticker
imports Main
begin

record LocalStateType =
  counter :: nat

datatype IMessageType
  = Increment |
  Count
```

Literally compile to this Isabelle code.
As you can see, the translation is pretty direct, and that’s intentional.
So the idea here is to make it as easy as possible to see that the formal spec corresponds to the English spec, and so Isabelle proofs about the formal spec apply.

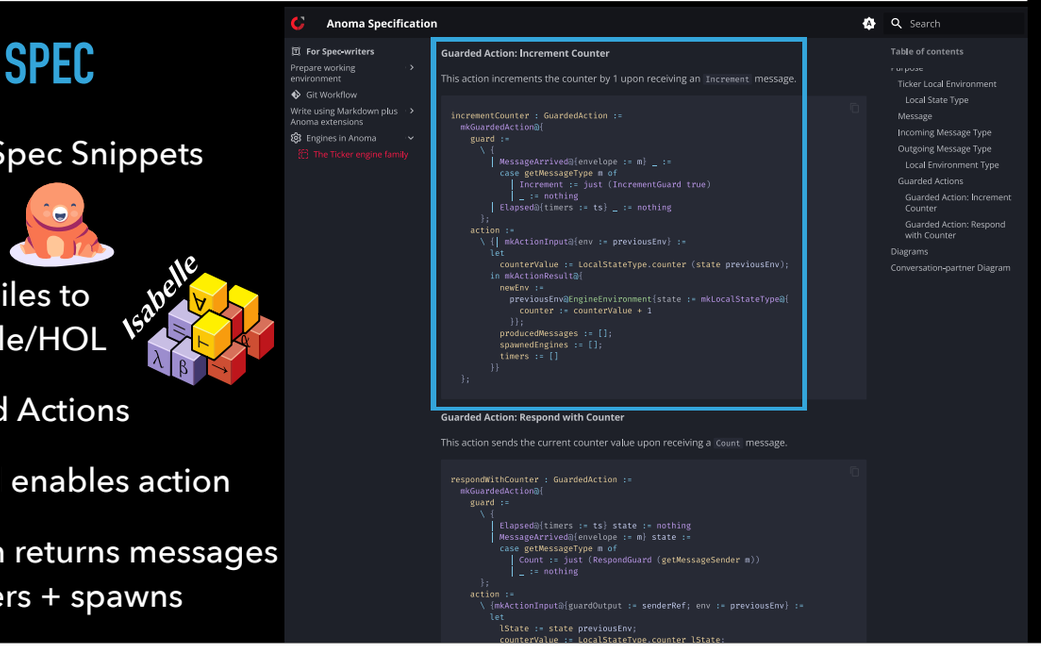


A little further down the same page, we see the “meat” of the spec, as it were.
We chose to write the actual state transitions actors can make as guarded actions, rather than say, a single function to be called each time any message is received.
This separates some concerns: it lets us think in terms of which state transitions are allowed at any given moment, without having to specify how the scheduler decides between them.

CURRENT STATE

ENGLISH SPEC

- Formal Spec Snippets
  - Juvix
  - Compiles to Isabelle/HOL
- Guarded Actions
  - Guard enables action
  - Action returns messages + timers + spawns



This Guarded action, for instance…

Increments a counter in state.
We can see that the code

CURRENT STATE

ENGLISH SPEC

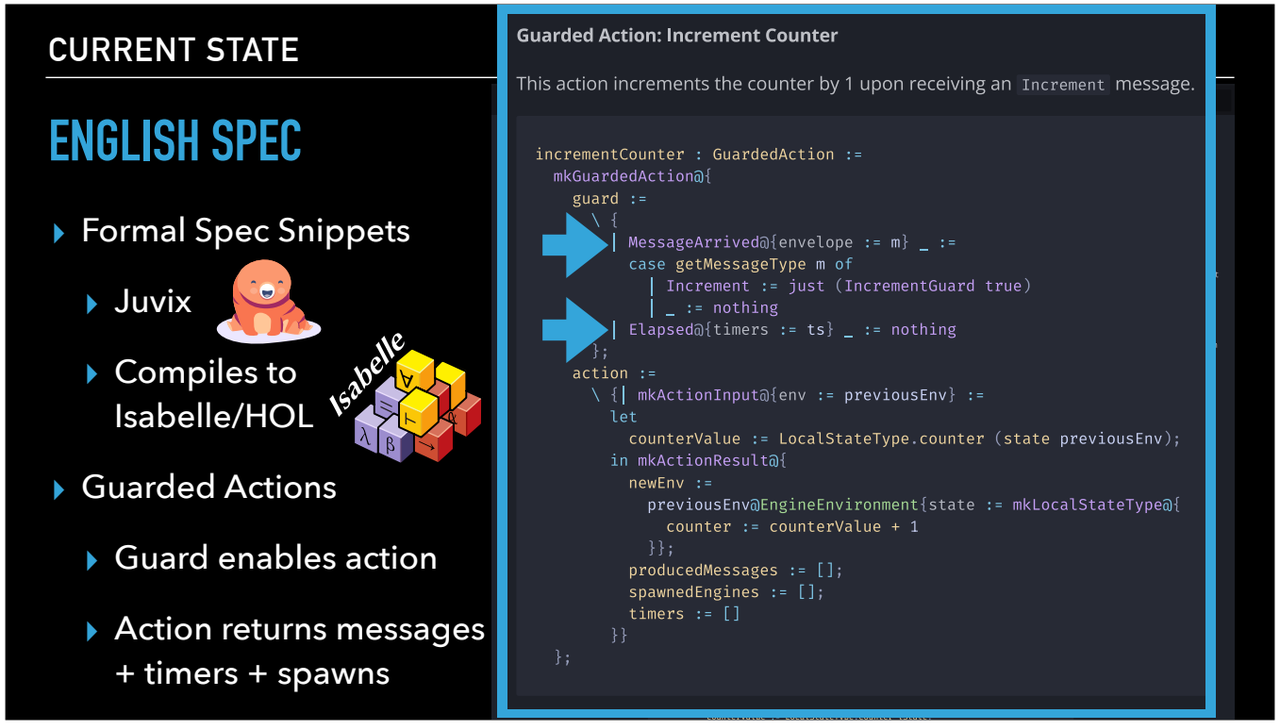
- Formal Spec Snippets
  - Juvix
  - Compiles to Isabelle/HOL
- Guarded Actions
  - Guard enables action
  - Action returns messages + timers + spawns

Guarded Action: Increment Counter

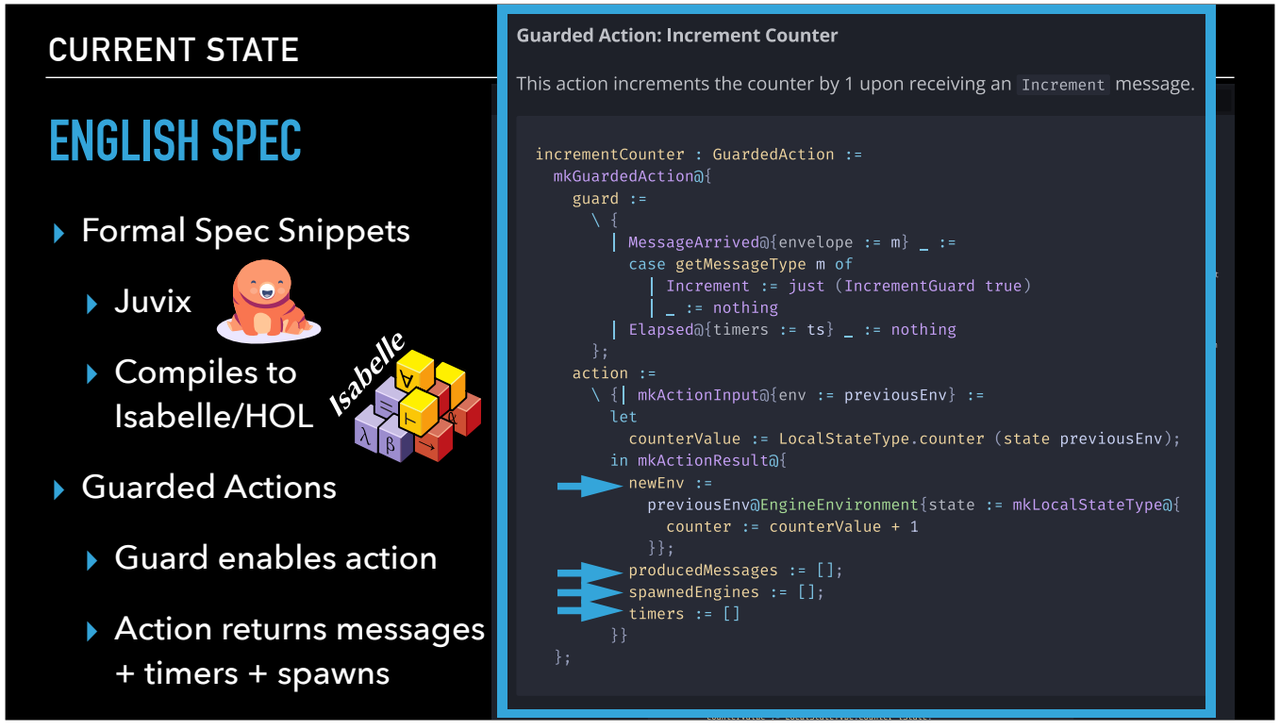
This action increments the counter by 1 upon receiving an `Increment` message.

```
incrementCounter : GuardedAction :=
  mkGuardedAction@{
    guard :=
      \ {
        | MessageArrived@{envelope := m} _ :=
          case getMessageType m of
            | Increment := just (IncrementGuard true)
            | _ := nothing
        | Elapsed@{timers := ts} _ := nothing
      };
    action :=
      \ {| mkActionInput@{env := previousEnv} :=
        let
          counterValue := LocalStateType.counter (state previousEnv);
        in mkActionResult@{
          newEnv :=
            previousEnv@EngineEnvironment{state := mkLocalStateType@{
              counter := counterValue + 1
            }};
          producedMessages := [];
          spawnedEngines := [];
          timers := []
        }}
  };
```

Is broken down into a guard an an action.
The guard can be triggered…



In response to a message arriving or a timer going off,
And if it returns a value that isn’t “nothing,”



Then the action returns a new local state, a set of messages produced, actors spawned, and timers initiated.

I should point out that this structure for actors and their guarded actions is something we arrived at ourselves.
There are a lot of formalizations of actors out there, with various features.
Mealy machines, for instance, are really simple, sending one message each time they receive one message, if I recall correctly.

So the potentially guarded actions with timers and spawning is a design choice that we hope is minimal while still expressing what we need.

CURRENT STATE

SWITCHING TOOLS

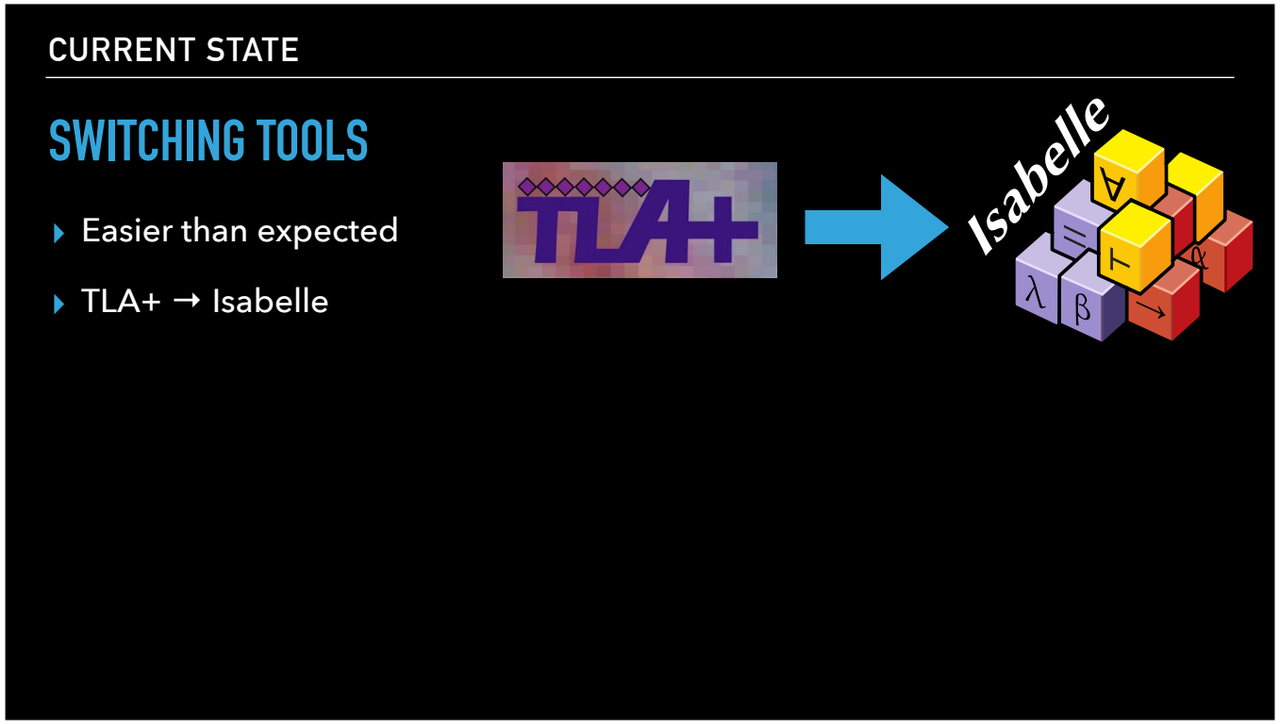
I was actually pretty worried about this spec plan, in that we had a decent experience with TLA+, and starting again in another language seemed hard.

CURRENT STATE

SWITCHING TOOLS

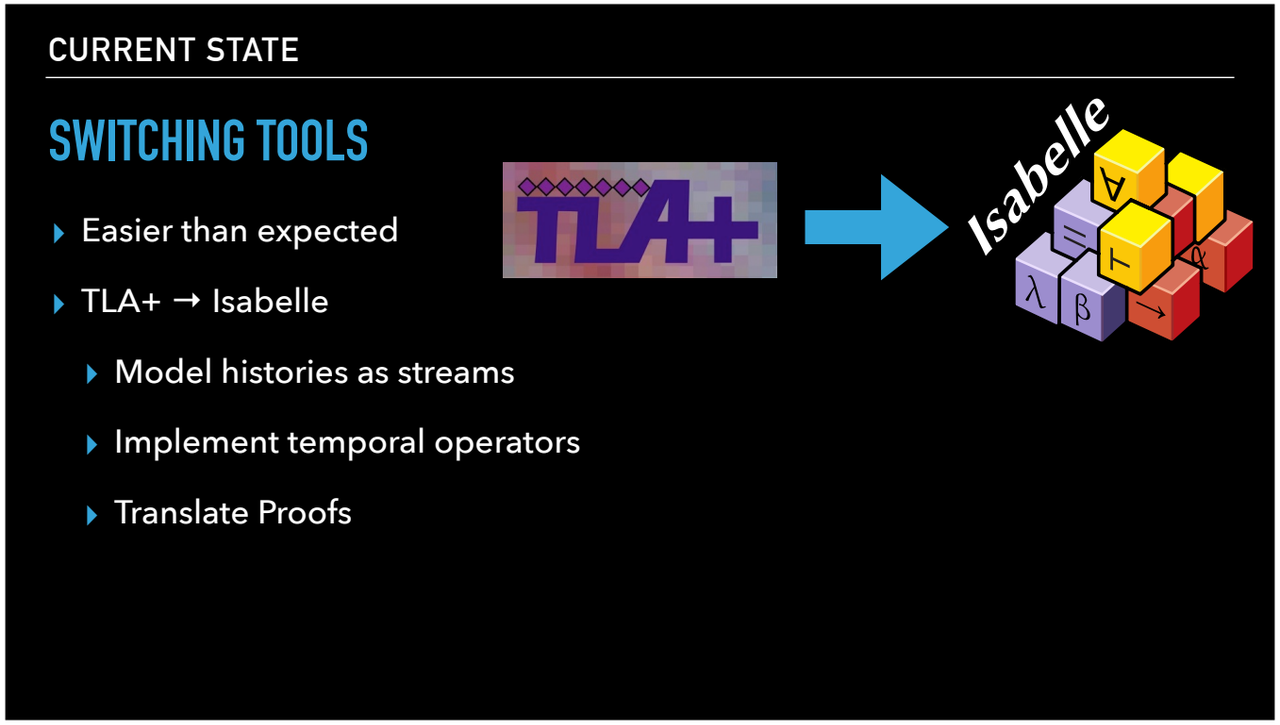
- Easier than expected

But the good news is that switching may be a lot easier than it seems.



At some point Anthony said “I bet it wouldn’t be that hard to translate the models and proofs from TLA+ directly into an Isabelle theory”
And indeed he was able to do it in, if I recall correctly, under a week.

So, TLA+ has several temporal operators built into its metatheory, which Isabelle does not.
The translation process requires that one pick a modeling semantics and then translate whatever temporal operators are needed into that semantics.



Anthony chose to use streams to model the history and temporal operators became predicates over streams.
TLA is also a theory of sets while Isabelle is a theory of predicates, where sets are defined as wrappers around their characteristic function.
This does change the flavor of some things, but in the end, translating proofs isn’t nearly as bad as I’d feared.

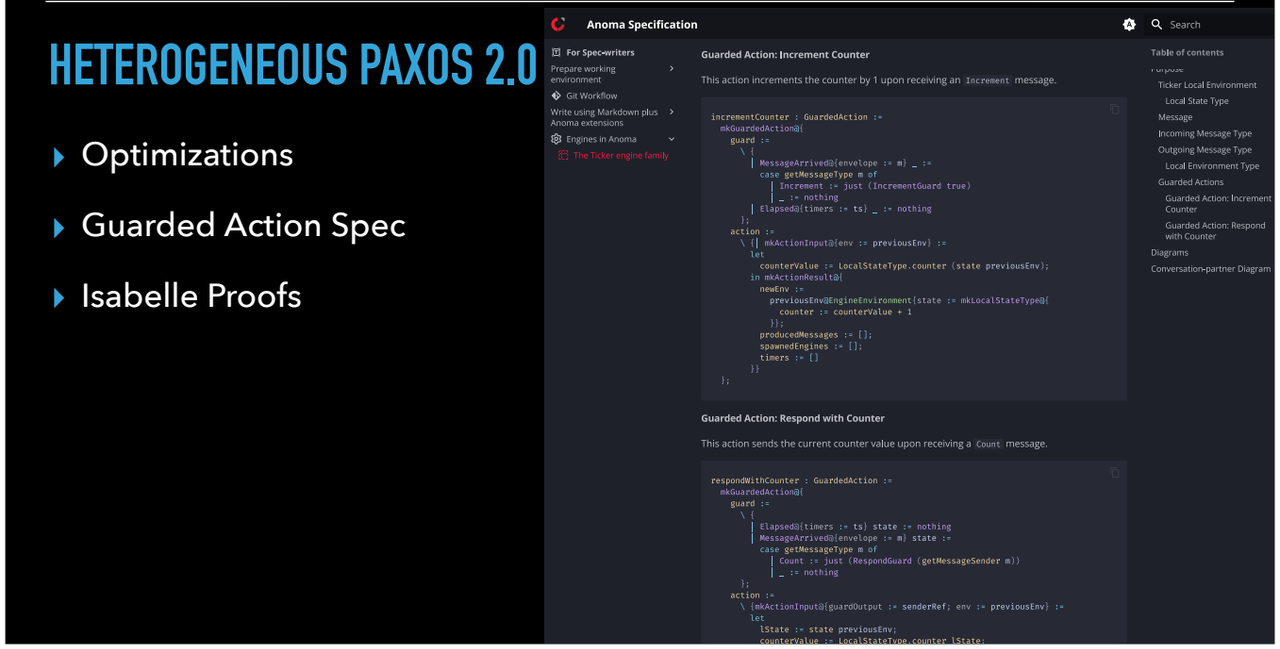
Anthony did point out that TLA itself can be far more intuitive, as one has to think of the model in Isabelle while it's just a formal logic in TLA.

In any case, this particular experience makes us feel more flexible about tool choice: moving work from one to another might not be so difficult

FUTURE

HETEROGENEOUS PAXOS 2.0

- Optimizations
- Guarded Action Spec
- Isabelle Proofs



We’ve got several formal methods projects in early stages, but one of them, which we’re calling Heterogeneous Paxos 2.0, takes into account a bunch of formalizations, and the next challenge is to specify it in the spec format we’ve been discussing, and prove that works properly in Isabelle.
Here’s hoping liveness goes well.

FUTURE

CONCLUSIONS

Community Knowledge

WISH LIST

- Expressive
  - Liveness properties
  - Recursive functions
  - Communication with Engineers
- Connection with Product
- Maturity

And that’s a summary our our experiences so far at Heliax.
I think we’ve learned a lot struggling with this stuff, but there are a couple of things I’m hoping folks here can take home.
One is that there is some knowledge in the research community about how to do stuff, with one example being choosing liveness to safety transformations, that isn’t necessarily obvious if you know distributed protocols but not formal verification, or formal verification but not distributed protocols.
Outreach and onboarding material could make a big difference for industry impact.

Another is a wish list of features we found ourselves wanting in tools, including
Expressiveness, featuring the ability to reason explicitly about liveness properties, using protocols with full-power recursive functions, and ideally the ability for engineers to read the formal spec and understand how this might correspond to product.
Separately, formal connection to the product, either through extraction, or embedding, or model-based testing, or other means of connecting the spec to production code.
And finally, maturity matters for industrial use: research code is fun, but there is a certain amount of polish, debugging, and community support that are really helpful.

Of these, we found liveness reasoning and connection to product the most challenging.
We certainly haven’t explored the full breadth of tools available, but we haven’t found particularly satisfying solutions for either yet.

Thanks!