Specifying blockchain protocols with Quint/TLA+ and checking them

Igor Konnov

Principal Research Scientist

informal SYSTEMS

| Links

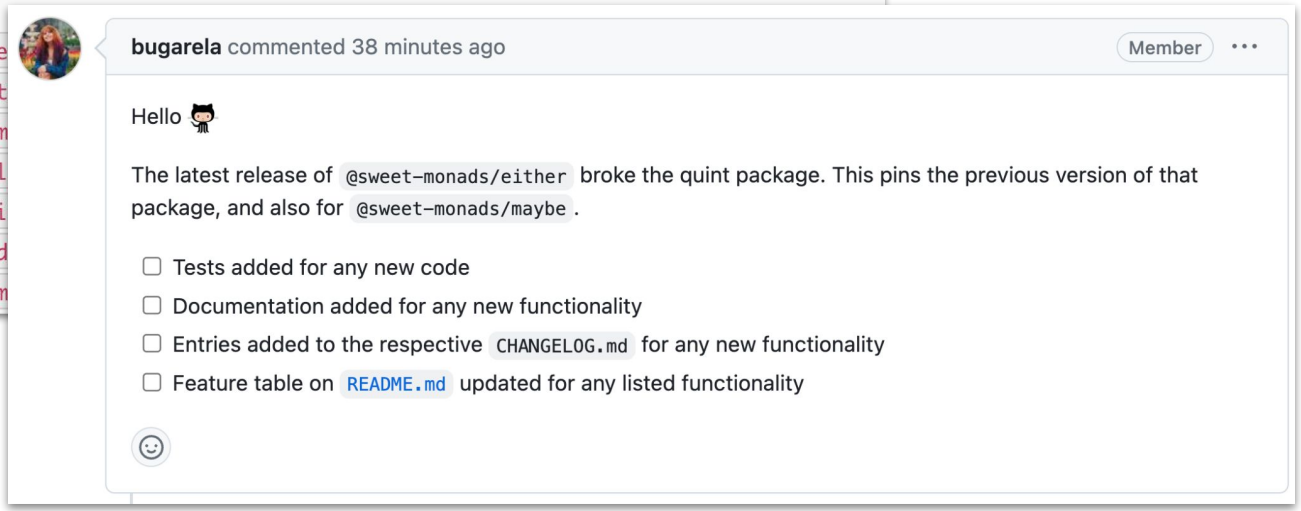
Material: **https://konnov.github.io/vtsa23/**

VSCode session: TODO

| Erratum

```
~ % quint --version
node:internal/modules/cjs/loader:553
  throw e;
  ^

Error: Cannot find module '/opt/homebrew/lib/node_modules/@informalsystems/quint/node_modules/@sweet-monads/either/cjs/index.js'
  at createEsmNotFoundErr (node:inte
  at finalizeEsmResolution (node:int
  at resolveExports (node:internal/m
  at Module._findPath (node:internal
  at Module._resolveFilename (node:i
  at Module._load (node:internal/mod
  at Module.require (node:internal/m
```



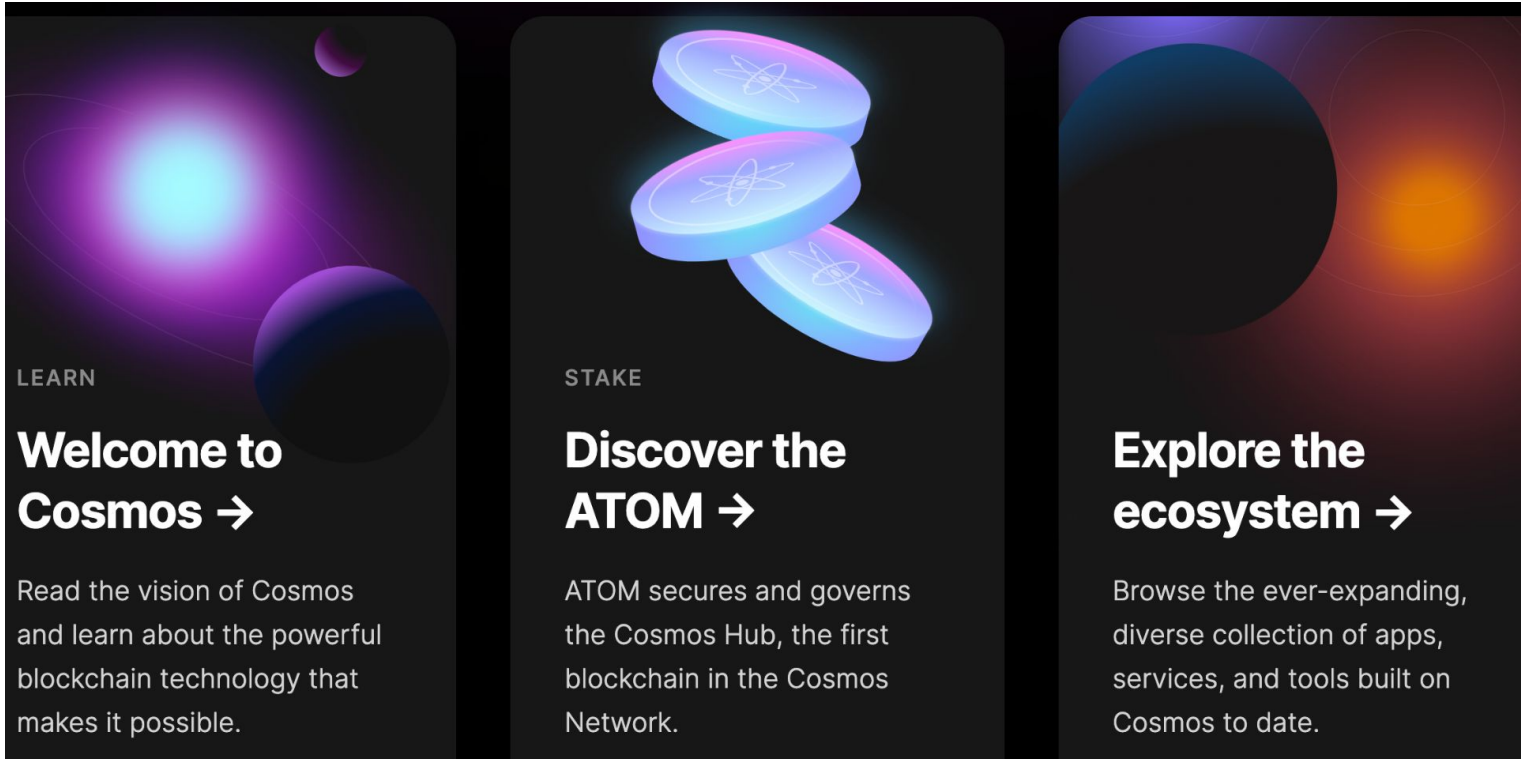
bugarela commented 38 minutes ago — Member

Hello

The latest release of `@sweet-monads/either` broke the quint package. This pins the previous version of that package, and also for `@sweet-monads/maybe`.

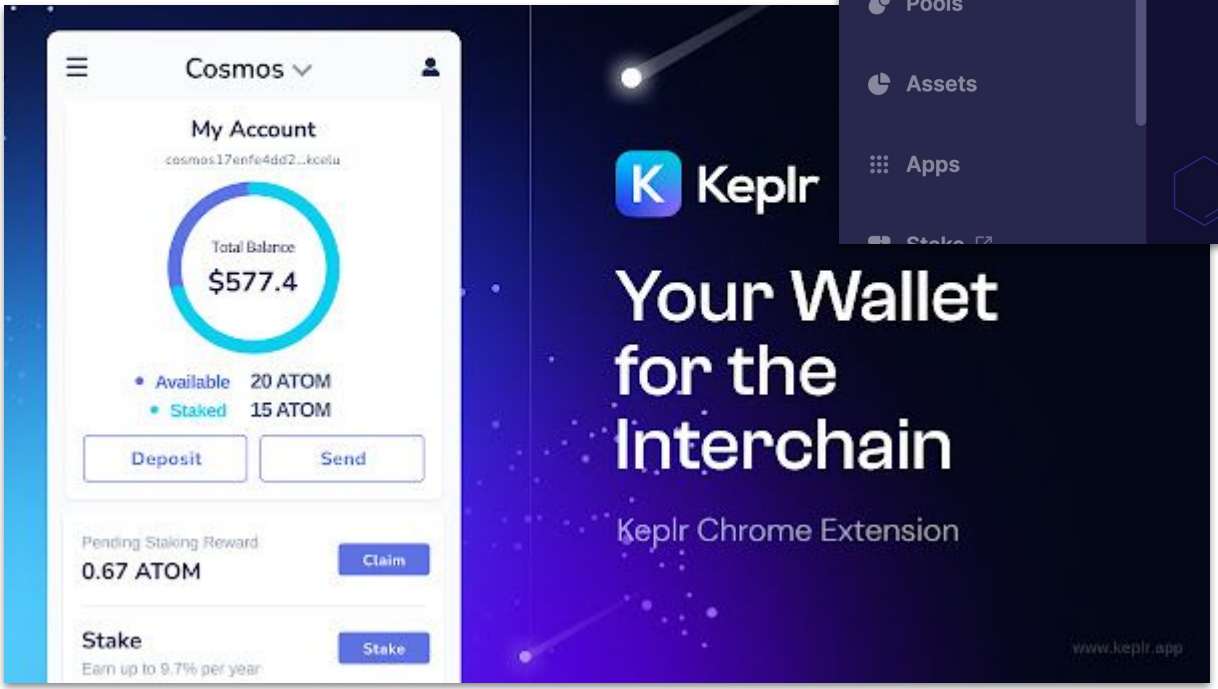
- [ ] Tests added for any new code
- [ ] Documentation added for any new functionality
- [ ] Entries added to the respective `CHANGELOG.md` for any new functionality
- [ ] Feature table on `README.md` updated for any listed functionality

Context
the Cosmos Ecosystem



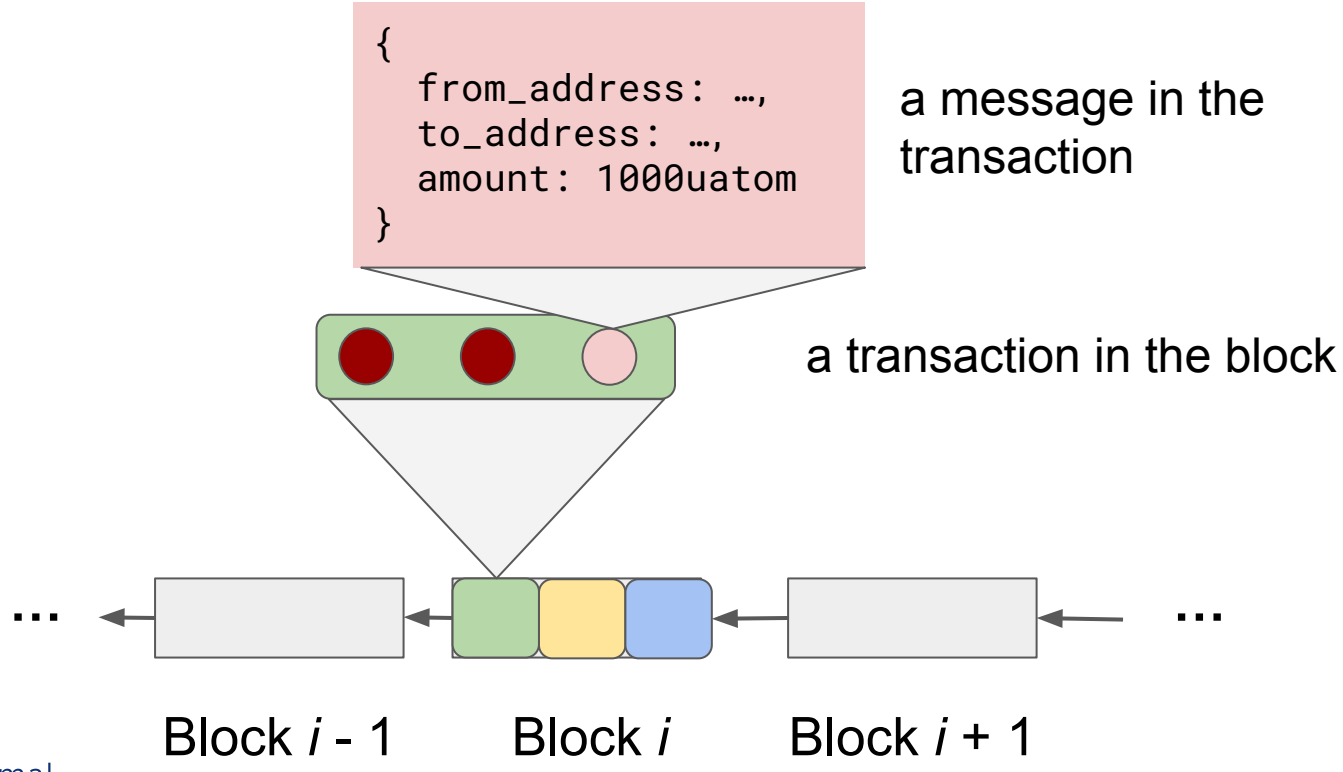
cosmos.network

keplr.app

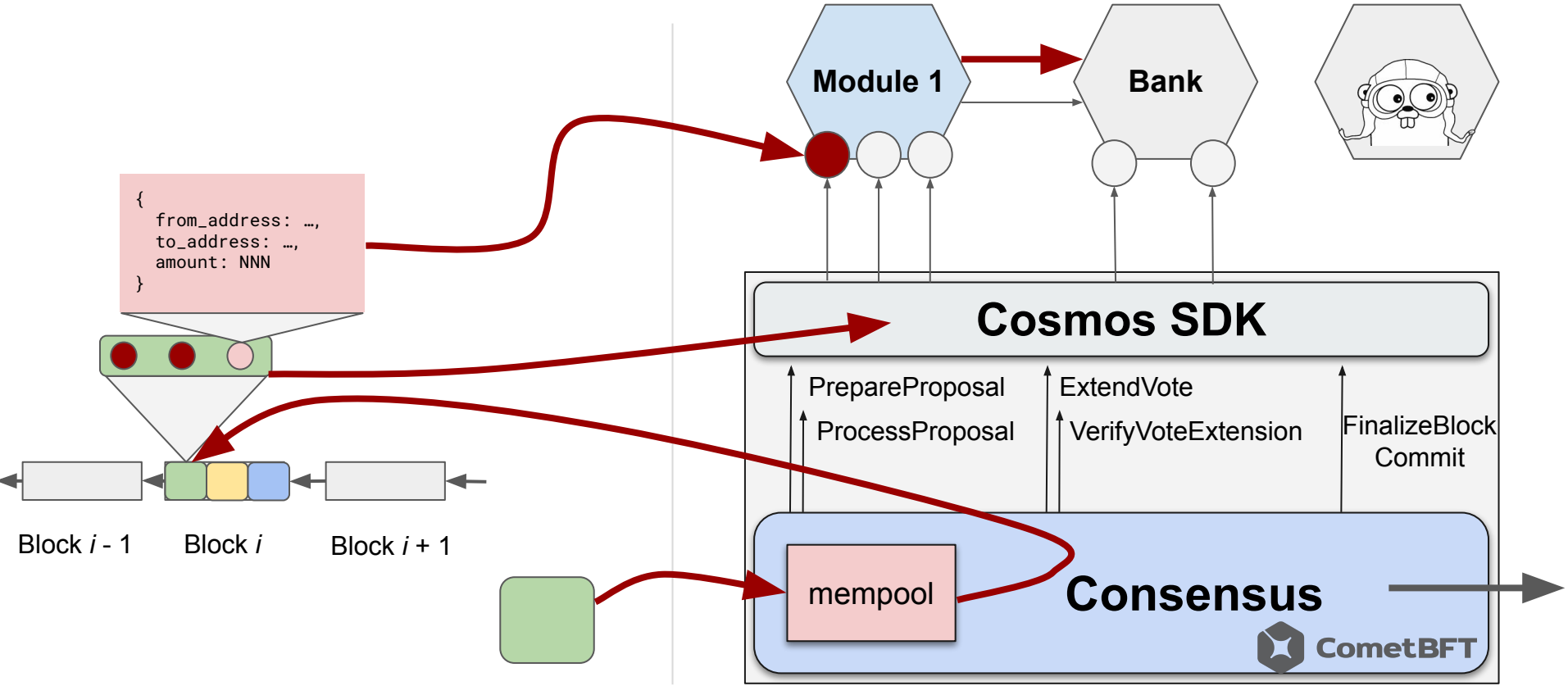


osmosis.zone

Blockchain as a ledger



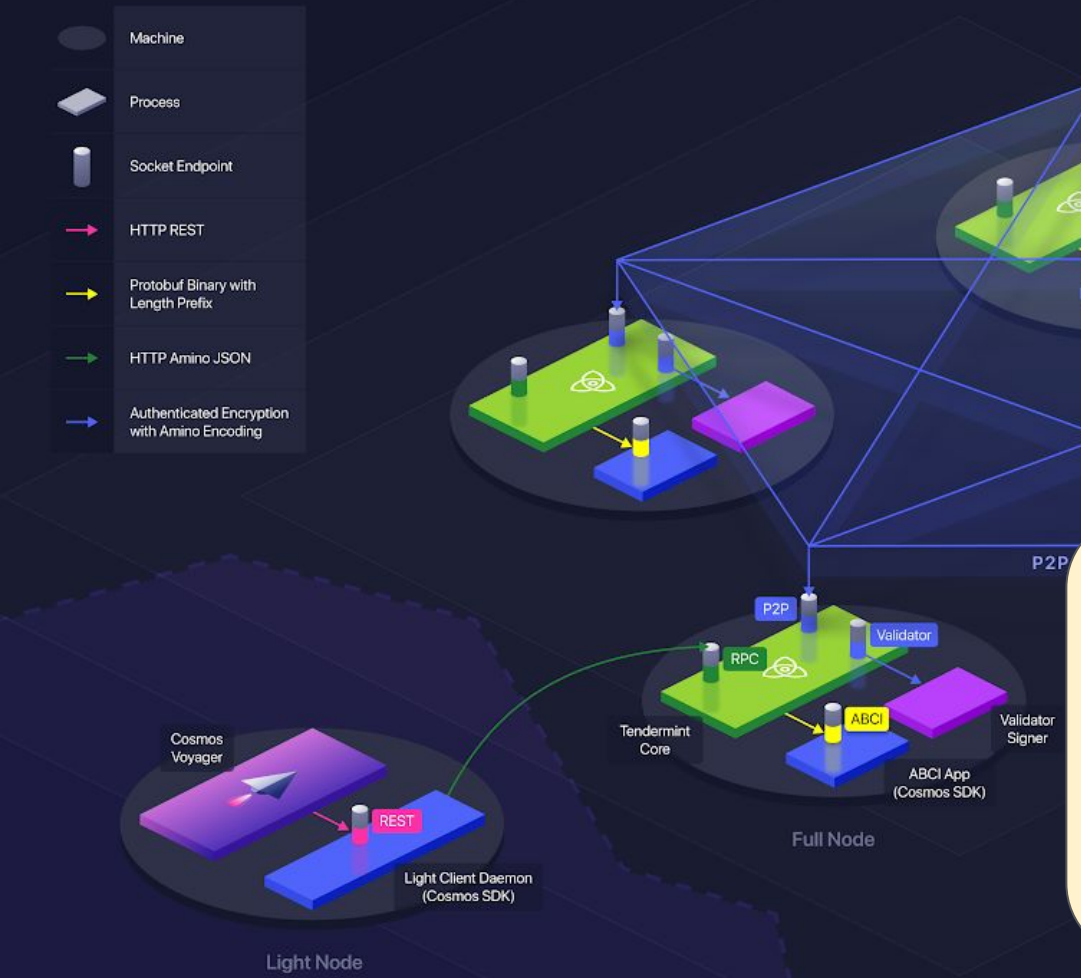
Blockchain validator



Smart contracts and dApps in Cosmos

Cosmos SDK	Golang
CosmWasm	Rust → WebAssembly
Agoric Zoe	Hardened Javascript
Evmos	Solidity

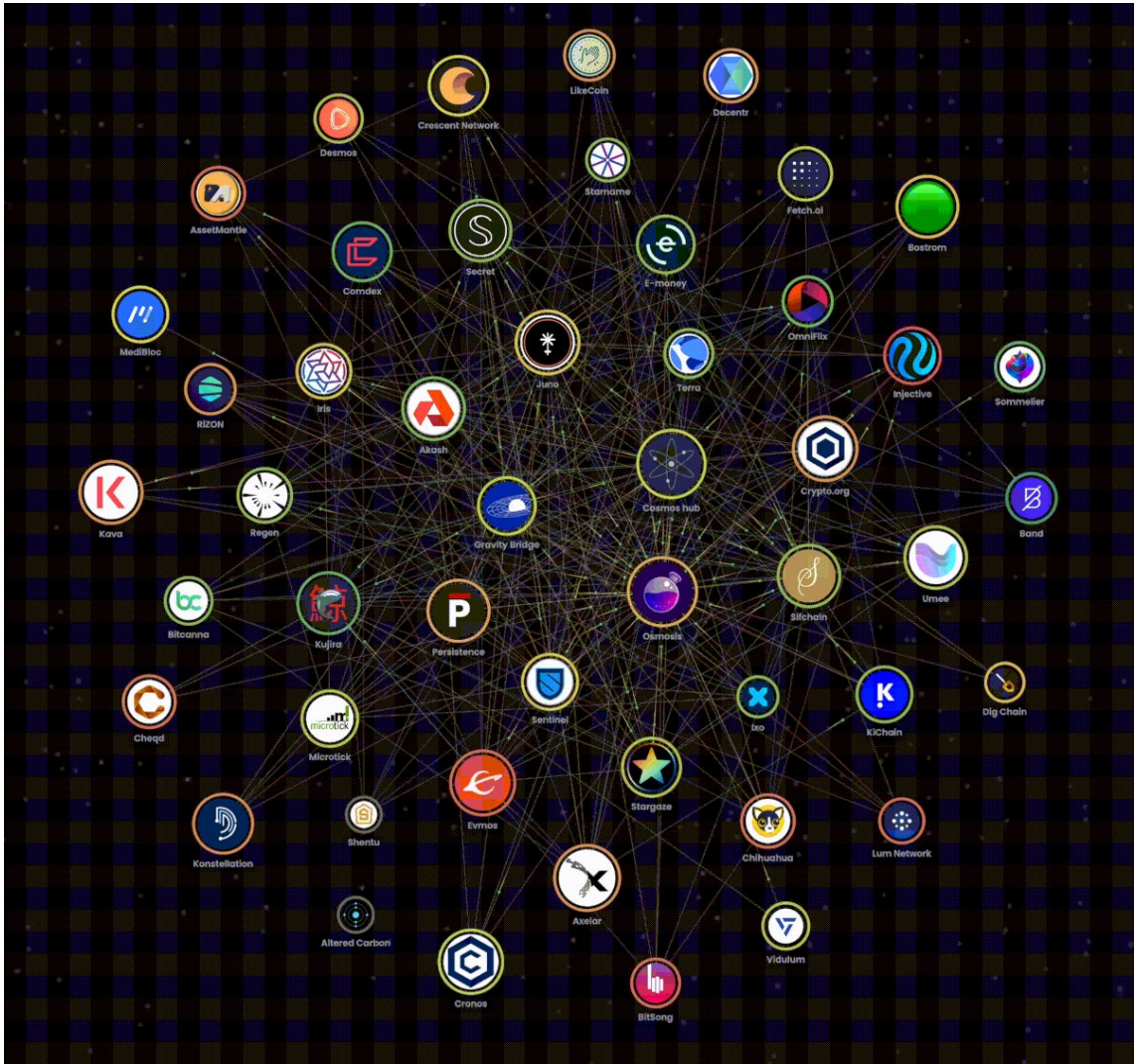
Tendermint Stack
CometBFT



Cosmos Hub:

- **175 validators** running 24/7
- jailed for downtime
- slashed for double signing, etc.
- many more full nodes

https://cosmos.network https://tendermint.com

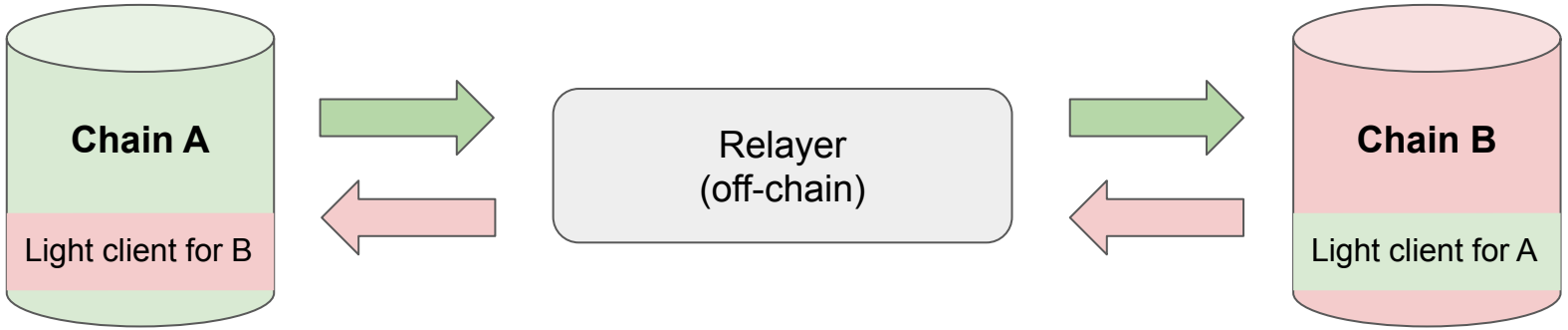


mapofzones.com

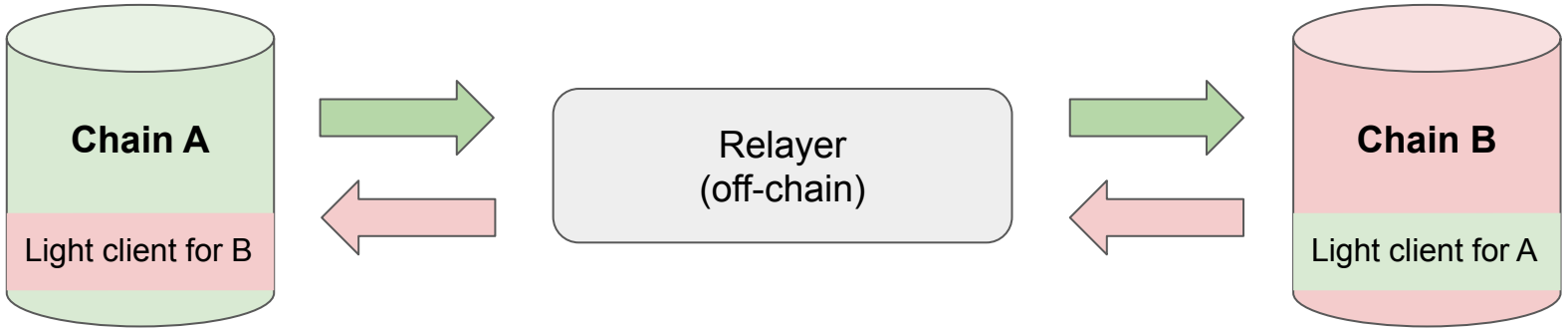
>50 blockchains

Inter-blockchain Communication (IBC)

High-level picture of IBC

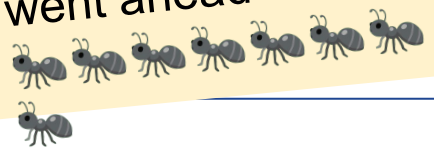


High-level picture of IBC



- ❑ 23 Interchain standards (ICS)
- ❑ English prose + pseudocode
- ❑ TLA$^+$ specifications

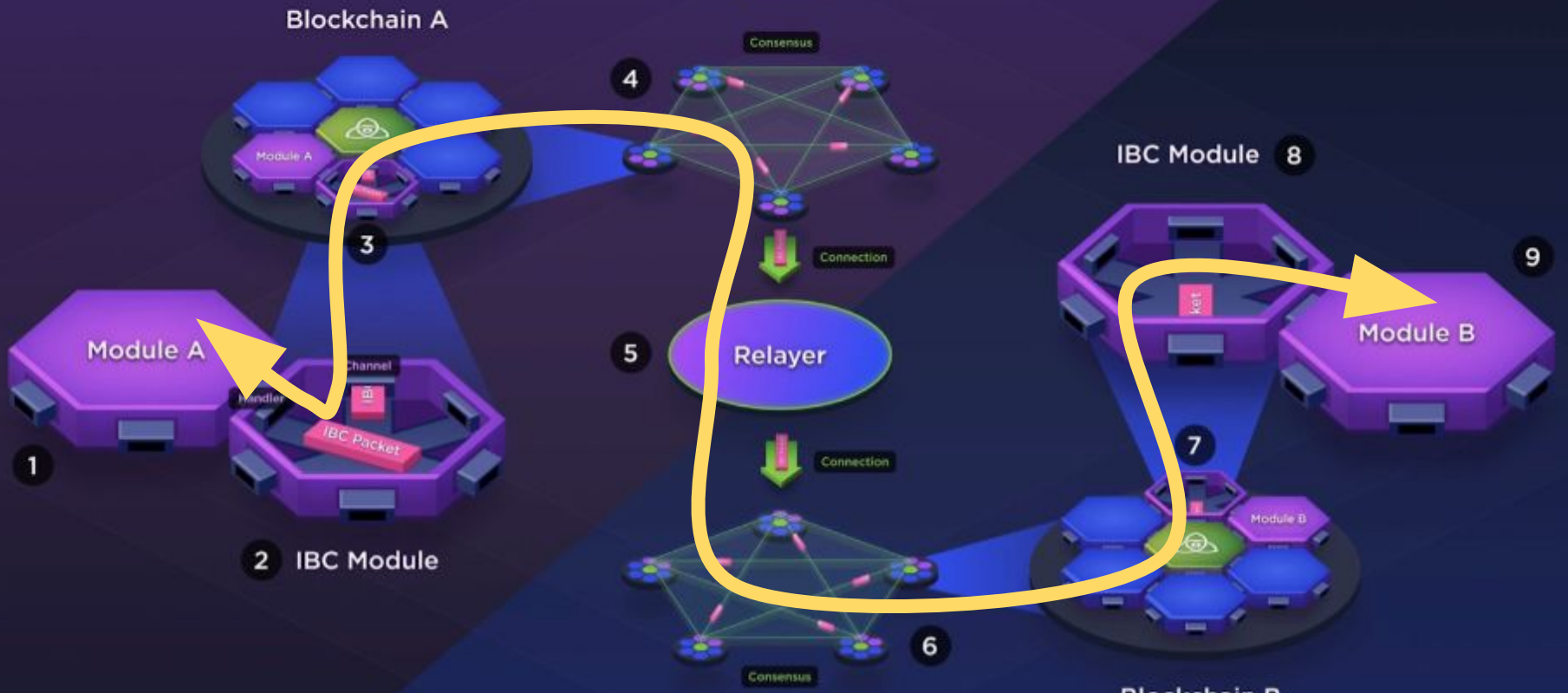
The implementation went ahead



IBC

IBC is the Inter-Blockchain Communication Protocol

A visualization of an IBC packet traveling between two blockchains



CHAIN A

CHAIN B

Specification and model checking

Heavy-weight verification

- Complete functional verification
- Hoare-style proofs

Mid-weight verification

- Complete model checking
- Conformance testing

Lightweight verification

- Static analysis
- Property-based testing

- Model checking
- Stateless exploration
- Bug finding

informal SYSTEMS

TLA+

PODC/DISC style

Proposer-Based Time - Part I

System Model

Time and Clocks

[PBTS-CLOCK-NEWTON.0]

There is a reference Newtonian real-time `t` (UTC).

Every correct validator `V` maintains a synchronized clock `C_V` that ensures:

[PBTS-CLOCK-PRECISION.0]

There exists a system parameter `PRECISION` such that for any two correct validators `V` and `W`, and at any real-time `t`:
`|C_V(t) - C_W(t)| < PRECISION`

Message Delays

We do not want to interfere with the Tendermint timing assumptions. We will postulate a timing restriction, which, if satisfied, ensures liveness is preserved.

In general the local clock may drift from the global time. (It may progress faster, e.g., one second of local clock time may correspond to 2 seconds of real-time). As a result the local clock and the global clock may be measured in different time units. Usually, the message delay is measured in global clock time units. To estimate the correct local timeout precisely, we would need to estimate the clock time duration of a message delay taking into account the clock drift. For simplicity we ignore this, and directly postulate the message delay assumption in terms of local time.

```
\* lines 36-46
\* [PBTS-ALG-NEW-PREVOTE.0]
UponProposalInPrevoteOrCommitAndPrevote(p) ==
  \E v \in ValidValues, t \in Timestamps, vr \in RoundsOrNil:
    /\ step[p] \in {"PREVOTE", "PRECOMMIT"} \* line 36
    /\ LET msg ==
         AsMsg([type |-> "PROPOSAL", src |-> Proposer[round[p]],
                round |-> round[p], proposal |-> Proposal(v, t), validRound |-> vr]) IN
       /\ <<p, msg>> \in receivedTimelyProposal \* updated line 36
       /\ LET PV == { m \in msgsPrevote[round[p]]: m.id = Id(Proposal(v, t)) } IN
         /\ Cardinality(PV) >= THRESHOLD2 \* line 36
         /\ evidence' = PV \union {msg} \union evidence
    /\  IF step[p] = "PREVOTE"
       THEN \* lines 38-41:
         /\ lockedValue' = [lockedValue EXCEPT ![p] = v]
         /\ lockedRound' = [lockedRound EXCEPT ![p] = round[p]]
         /\ BroadcastPrecommit(p, round[p], Id(Proposal(v, t)))
         /\ step' = [step EXCEPT ![p] = "PRECOMMIT"]
       ELSE
         UNCHANGED <<lockedValue, lockedRound, msgsPrecommit, step>>
      \* lines 42-43
    /\ validValue' = [validValue EXCEPT ![p] = v]
    /\ validRound' = [validRound EXCEPT ![p] = round[p]]
    /\ UNCHANGED <<round, decision, msgsPropose, msgsPrevote,
                   localClock, realTime, receivedTimelyProposal, inspectedProposal,
                   beginConsensus, endConsensus, lastBeginConsensus, proposalTime, proposalReceivedTime>>
    /\ action' = "UponProposalInPrevoteOrCommitAndPrevote"
```

informal SYSTEMS

Core Cosmos infrastructure

PODC/DISC style

- Tendermint consensus
- Proposer-based time
- Light client
- FastSync
- ABCI++
- Interchain security (complete)
- Namada's Proof-of-Stake
- CosmosSDK Store
- Celestia

TLA+

- Tendermint consensus:

  (safety + accountability)

- Proposer-based time
- Light client
- FastSync
- Interchain Security (CCV)
- IBC: 02, 03, 04, 18
- (Partial) specs in security audits
- Namada's Proof-of-Stake

TLA$^+$ — a general specification language

Set theory + temporal logic

TLA$^+$ is used in industry

TLA$^+$ tools maintained at Inria
- an interactive proof system (TLAPS)
- a model checker (TLC), state enumeration

Microsoft Research amazon

Markus Kuppe

Stephan Merz

APALACHE: model checker for TLA$^+$
- a symbolic model checker

Quint Lang

Jure Kukovec

Shon Feder

Gabriela Moreira
@bugarela

Igor Konnov
@konnov

Thomas Pani

Jure Kukovec

Thanh-Hai Tran

Marijana Lazić

Josef Widder

WWTF
VIENNA SCIENCE AND TECHNOLOGY FUND

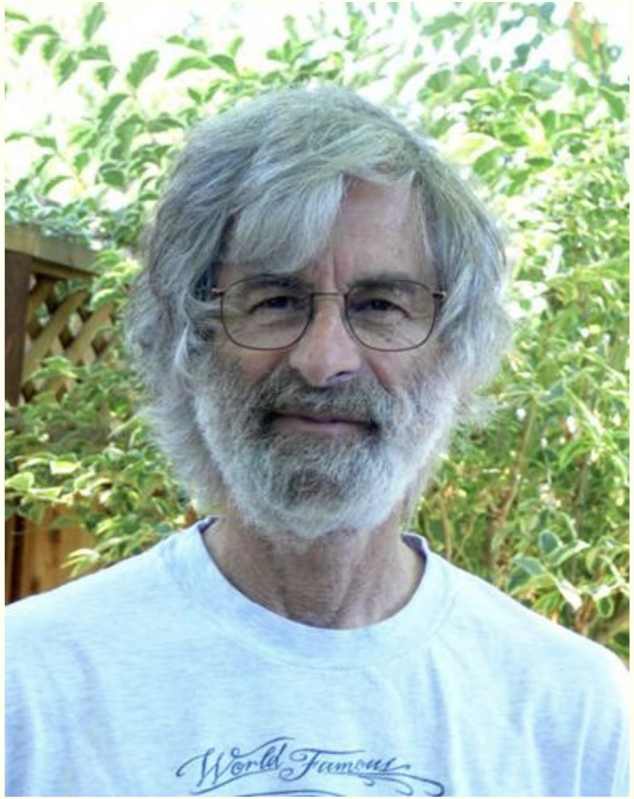
informal SYSTEMS

TU WIEN

wirtschaftsagentur wien | Für die Stadt Wien

lamport.org

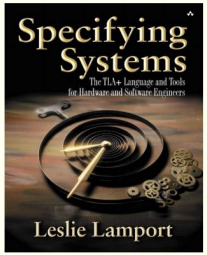
LESLIE LAMPORT'S HOME PAGE



[TLA+ Use at Amazon]

[The TLA Web Page]

[My Collected Works]



The Tools

SANY Syntactic Analyzer [show]

TLC Model Checker [hide]

A model checker and simulator for "executable" TLA+ specifications, which include most specifications written by engineers. It is an explicit-state model checker that can check both safety and liveness properties. For safety checking, TLC achieves an approximately linear speedup on modern large computers using 32 or more cores. It can further speed up model checking by running on a network of computers, and provides easy deployment on cloud systems. TLC was written by Yuan Yu and extended by Markus Kuppe.

Apalache Model Checker [show]

PlusCal Translator [show]

TLAPS Proof System [hide]

A system for mechanically checking proofs written in TLA+, including proofs of the kinds of properties described here. It is being developed at the Microsoft Research-Inria Joint Centre. See the TLAPS home page for more information. The first version of TLAPS was written by Kaustuv Chaudhuri. Improvements have been made by Damien Doligez, Stephan Merz, Hernán Vanzetto, Tomer Libal, Martin Riener, and Denis Cousineau.

TLATeX Pretty-Printer [show]

How TLA+ looks like

```
255     \* lines 14-19, a proposal may be sent later
256     \* @type: $process => Bool;
257     InsertProposal(p) ==
258       LET r == round[p] IN
259       /\ p = Proposer[r]
260       /\ step[p] = "PROPOSE"
261         \* if the proposer is sending a proposal, then there
262         \* by the correct processes for the same round
263       /\ \A m \in msgsPropose[r]: m.src /= p
264       /\ \E v \in ValidValues:
265           LET proposal ==
266               IF validValue[p] /= NilValue
267               THEN validValue[p]
268               ELSE v
269           IN BroadcastProposal(p, round[p], proposal, validRound[p])
270       /\ UNCHANGED <<round, decision, lockedValue, lockedRound,
271                      validValue, step, validRound, msgsPrevote, msgsPrecommit,
272                      evidencePropose, evidencePrevote, evidencePrecommit>>
273       /\ action' = "InsertProposal"
```

lines $34-35+$ lines $61-64$ ($onTimeoutPrevote$)

$UponQuorumOfPrevotesAny(p) \triangleq$
$\land\ step[p] = \text{“PREVOTE”}$ line 34 and 61
$\land\ \exists\, MyEvidence \in \text{SUBSET}\ msgsPrevote[round[p]]:$
find the unique voters in the evidence
$\text{LET}\ Voters \triangleq \{m.src : m \in MyEvidence\}\ \text{IN}$
compare the number of the unique voters against the threshold
$\land\ Cardinality(Voters) \geq THRESHOLD2$ line 34
$\land\ evidencePrevote' = MyEvidence \cup evidencePrevote$
$\land\ BroadcastPrecommit(p, round[p], NilValue)$
$\land\ step' = [step\ \text{EXCEPT}\ ![p] = \text{“PRECOMMIT”}]$
$\land\ \text{UNCHANGED}\ \langle round, decision, lockedValue, lockedRound,$
$validValue, validRound, msgsPropose, msgsPrevote,$
$evidencePropose, evidencePrecommit\rangle$
$\land\ action' = \text{“UponQuorumOfPrevotesAny”}$

Quint Lang

new skin for TLA+

Protocol designers

- ✅ Onboarding is not too hard
- ✅ Happy to see counterexamples
- ✅ Check state invariants with Apalache
- 😕 Longer executions? No time for inductive invariants
- 😕 Apalache is slow
  - ✅ Antipatterns
  - ✅ Randomized symbolic execution
  - ✅ Parallel execution in the cloud

Blockchain engineers and security auditors

- 😕 Onboarding is hard – TLA$^+$ is not a programming language
- 😕 High expectations from tooling
- 😕 Want to write unit tests in Golang/Rust
- 😕 Want to use property-based testing

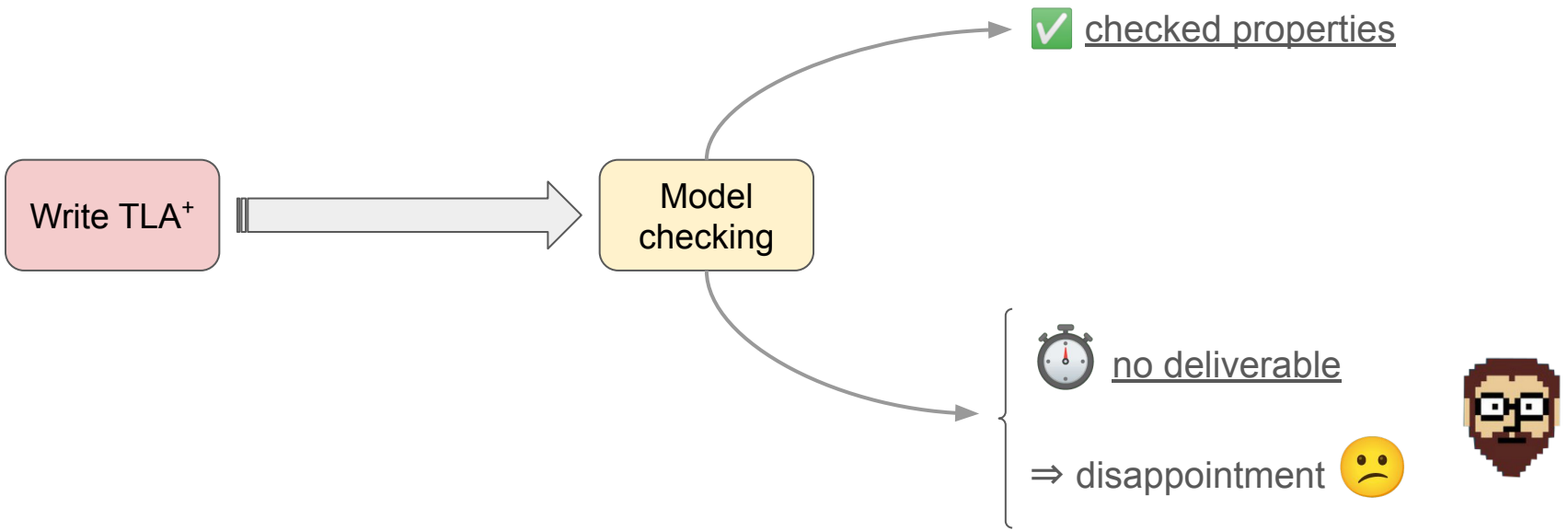
Similar to the experience of:

A. Reid et. al. Towards making formal methods normal: meeting developers where they are (2020)

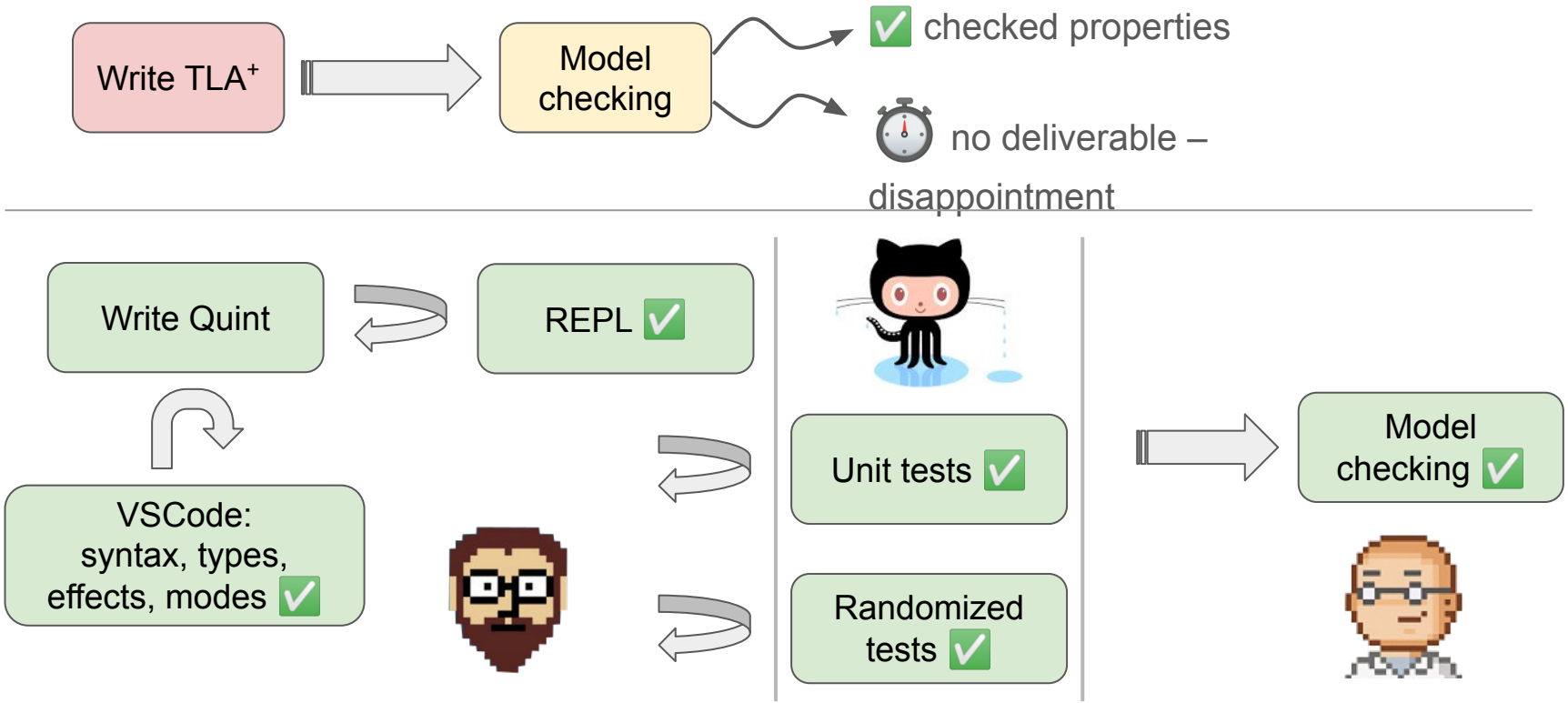
Security auditors

- 😕 Onboarding is hard – TLA$^+$ is not a programming language
- 😕 Lucky, if the customer gives them a markdown spec
- 😕 Inspect 10-40 KLOC of Golang/Rust in several weeks
- 😕 Just want to use fuzzers

The feedback loop

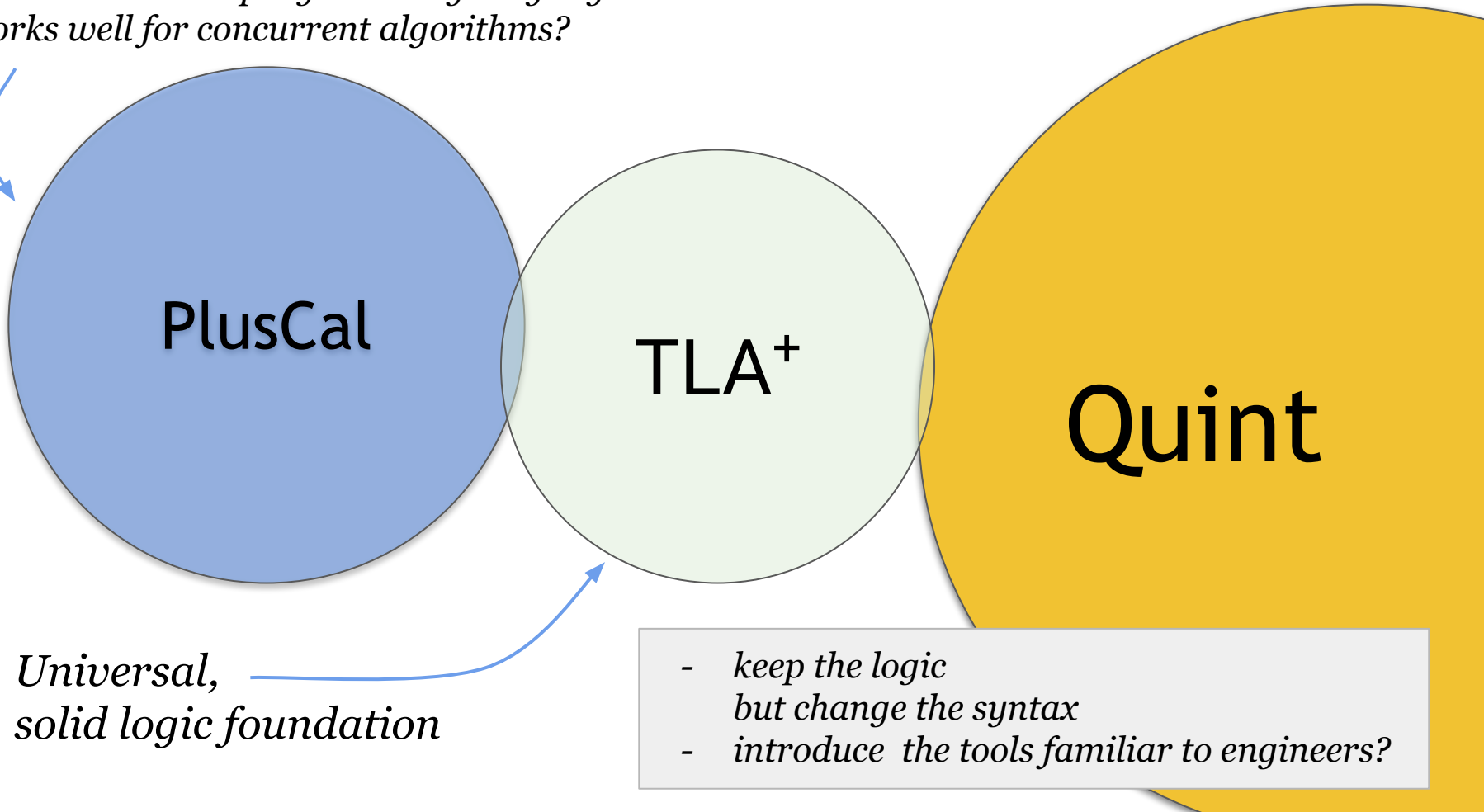


Shortening the feedback loop



[Potential user base]

Looks more like a programming language.
Works well for concurrent algorithms?



Universal,
solid logic foundation

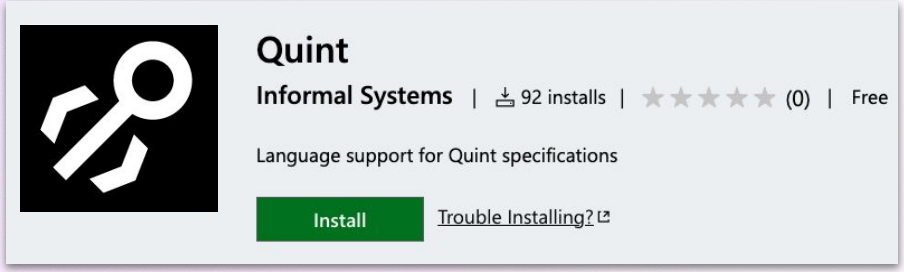
- *keep the logic*
  but change the syntax
- *introduce the tools familiar to engineers?*

| Time Plan

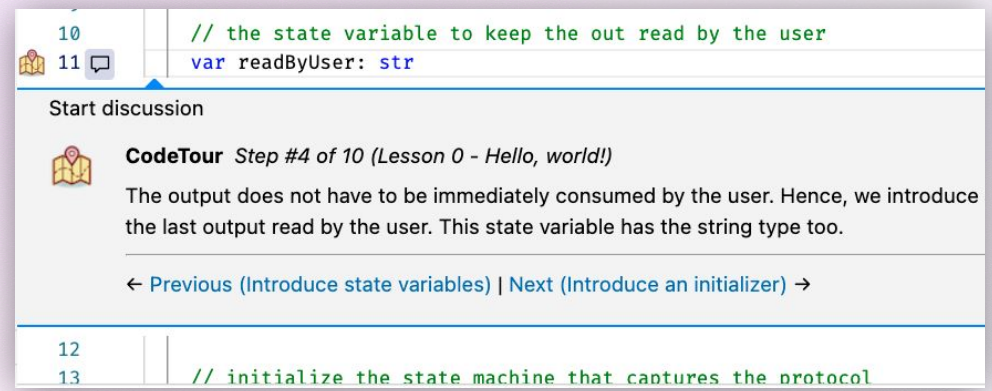
1. Quint by example
2. Quint as a specification language
3. More examples and Apalache
4. Brief introduction to TLA$^+$ via Quint

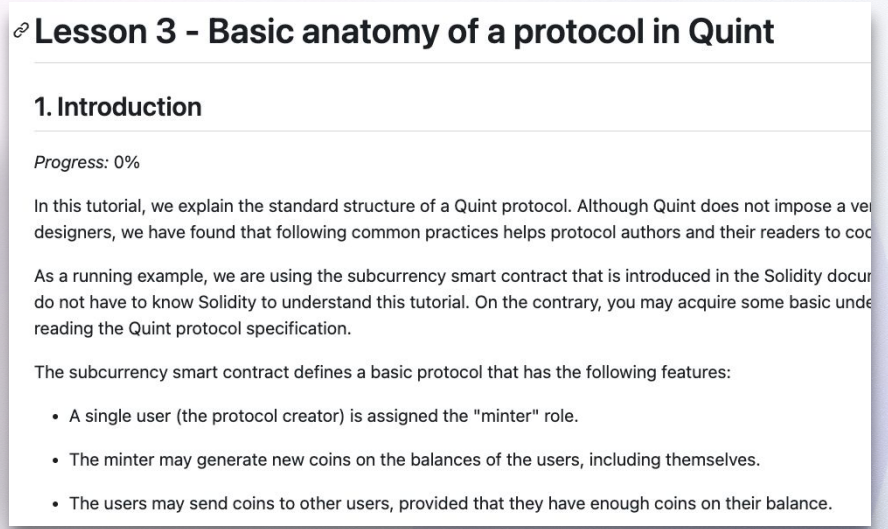
Quint by Example

Install the VSCode Plugin: [github.com/informalsystems/quint]



Read tutorials:





| Quint Cheatsheet

Comments

```
// one line
/* multiple
        lines */
```

Basic types

`bool` – Booleans

`int` – signed big integers

`str` – string literals

`type Name = otherType`
type alias, starts with upper-case

Literals

`false true`

`123  123_000  0x12abcd`

`"Quint"`: str, a string

Int: Set[int] – all integers

Nat: Set[int] – all non-negative integers

Bool = Set(false, true)

Records

`{ name: str, age: int }`
record type

`{ name: "TLA+", age: 33 }`
new record of two fields

`R.name` the field value

`R.with("name", "Quint")`

Sets - core data structure!

`Set[T]` – type: set with elements of type T

`Set(1, 2, 3)` – new set, contains its arguments

`1.to(4)` – new set: Set(1, 2, 3, 4)

`1.in(S)` – true, if the argument is in *S*

`S.contains(1)` – the same

`S.subseteq(T)` – true, if all elements of *S* are in *T*

`S.union(T)` – new set: elements in *S* or in *T*

`S.intersect(T)` – new set: elements both in *S* and in *T*

`S.exclude(T)` – new set: elements in *S* but not in *T*

`S.map(x => 2 * x)` – new set: elements of S are transformed by expression

`S.filter(x => x > 0)` – new set: leaves the elements of S that satisfy condition

`S.exists(x => x > 10)` – true, if some element of *S* satisfies condition

`S.forall(x => x <= 10)` – true, if all elements of *S* satisfy condition

`size(S)` – the number of elements in *S*, unless *S* is infinite (Int or Nat)

`isFinite(S)` – true, if *S* is finite

`Set(1, 2).powerset()` all subsets:

Maps - key/value bindings

`a -> b` – type: binds keys of type *a* to values of type *b*

`Map(1 -> 2, 3 -> 6)` – binds keys 1, 3 to values 2, 6

`S.mapBy(x => 2 * x)` – binds keys in *S* to expressions

`M.keys()` – the set of keys

`M.get(key)` – get the value bound to *key*

`M.set(k, v)` – copy of *M*: but binds *k* to *v*, if *k* has a value

`M.put(key, v)` – copy of *M*: but (re-)binds *k* to *v*

`M.setBy(k, (old => old + 1))`
as *M*.set(*k*, *v*) but *v* is computed via anonymous operator with *old* == *M*.get(*k*)

`S.setOfMaps(T)` – new set: contains **all maps** that bind elements of *S* to elements of *T*

`Set((1, 2), (3, 6)).setToMap()`
new map: bind the first elements of tuples to the second elements

Tuples

`(str, int, bool)`
tuple type

`("Quint", 2023, true)`
new tuple

`T._1  T._2  T._3`
get tuple elements

`tuples(S1, S2, S3)`

Lists - use Set, if you can

`List[T]` – type: list with elements of type T

`[1, 2, 3]` – new list, contains its arguments in order

`List(1, 2, 3)` – the same

`range(start, end)` – new list [*start*, *start* + 1, …, *end* – 1]

`length(L)` – the number of elements in the list *L*

`L[i]` – ith element, if `0 <= i < length(L)`

`L.concat(K)` – new list: start with elements of *L*, continue with elements of *K*

`L.append(x)` – new list: just L.concat([x])

`L.replaceAt(i, x)` – *L*'s copy but the ith element is set to *x*

`L.slice(s, e)` – new list: [*L*[*s*], …, *L*[*e* – 1]]

`L.select(x > 5)` – new list: leaves the elements of *L* that satisfy condition

`L.foldl(i, (s, x) => x + s)`
go over elements of *L* *in order*, apply expression, continue with the result; i is the initial value of *s*

`head(L)` – the element L[0]

`tail(L)` – new list: all elements of *L* but the head

informal SYSTEMS

| Examples

- The examples used in these lectures:

  https://github.com/informalsystems/quint-sandbox/

- Quint examples:

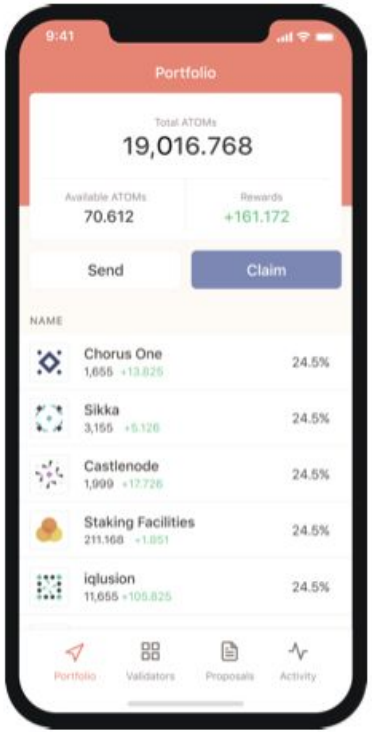
  https://github.com/informalsystems/quint/tree/main/examples

IG

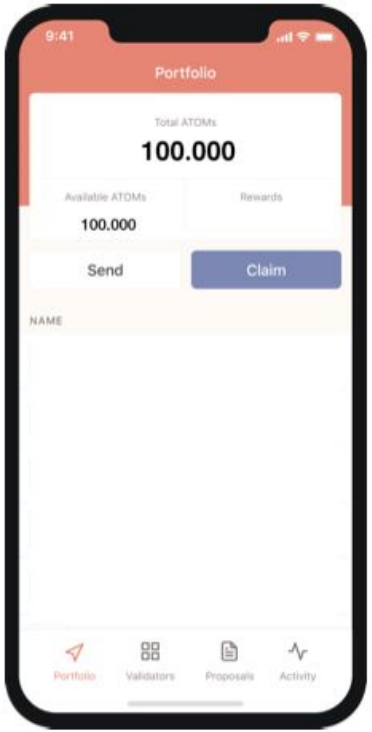
Can you generate a Solidity contract that allows the user to deposit N Ether and get 2 * N Ether back?

Certainly! Below is an example of a Solidity contract that allows users to deposit Ether and receive twice the amount they deposited when they withdraw.

| Crypto transfers



COSMOS



| Simple Ponzi Contract

- The contract author is the first investor
- The next investor transfers ≥110% of the investment
- **The previous investor** gets the whole new amount

- If you think, we made it up:

github.com/djokicx/sol-simple-ponzi/blob/master/contracts/SimplePonzi.sol

The original contract in Solidity

SimplePonzi.sol 1

```
1   pragma solidity ^0.5.0;
2
3   contract SimplePonzi {
4       // if no one ever tops the bid of 'currentInvestor',
5       // they will be the one to lose the investment
6       address payable currentInvestor;
7       // the investment they stand to lose
8       uint public currentInvestment = 0;
9
10      // payable callback function; gets executed by directly
11      // sending ether to the contract
12      function () external payable {
13          // new investments must be 10% greater than current, current * 1.1 (expressed in fraction)
14          uint minimumInvestment = (currentInvestment * 11) / 10;
15          require(msg.value > minimumInvestment, "Investment must meet the minimum requirement of 0.001 ether.");
16
17          // document new investor
18          address payable previousInvestor = currentInvestor;
19          currentInvestor = msg.sender;
20          currentInvestment = msg.value;
21
22          // payout previous investor
23
24          // .transfer would allow anyone to block our contract
25          // send forwards only 2300 gas stipend, so we are safe from re-entrancy attack
26          // detering the attackers by imposing a monetary penalty for attempting a malicious tx
27          // developers get penalized for their errors (after all, security is paramount)
28          previousInvestor.send(msg.value);
29      }
30  }
```

Contract spec in Quint

```
module simplePonzi {
  // addresses are string literals
  type Addr = str

  // a state of the EVM that is observed/modified by the contract
  type EvmState = {
    // the account balances for every address
    balances: Addr -> int
  }

  // a state of the contract
  type PonziState = {
   // the address of the last investor, initially of the contract
    currentInvestor: Addr,
    // the investment made by the current investor
    currentInvestment: int,
  }

  // create a new instance of the contract
  pure def newPonzi(owner: Addr): PonziState = {
    currentInvestor: owner, currentInvestment: 0
  }

  // The result of applying a method.
  // If error != "", then the states are not modified,
  // and error contains the error message.
  // Otherwise, evm and ponzi contain the new states.
  type Result = {
    error: str,
    evm: EvmState,
    ponzi: PonziState,
  }
```

```
  // Receive an investment and distribute the rewards (to the previous investor).
  pure def receive(evm: EvmState,
      ponzi: PonziState, investor: Addr, amount: int): Result = {
    if (amount > evm.balances.get(investor)) {
      {
        error: "Insufficient funds",
        evm: evm,
        ponzi: ponzi,
      }
    } else if (amount < 11 * ponzi.currentInvestment / 10) {
      {
        error: "New investment must be 110% of the last one",
        evm: evm,
        ponzi: ponzi,
      }
    } else {
      pure val newBalances =
        evm.balances
          .setBy(investor, old => old - amount)
          .setBy(ponzi.currentInvestor, old => old + amount)
      {
        evm: { balances: newBalances },
        ponzi: { currentInvestor: investor, currentInvestment: amount },
        error: "",
      }
    }
  }
}
```

There is more...

- ❑ Specifying a state machine for testing
- ❑ Specifying expected properties
- ❑ Execute the spec
- ❑ Interactive session

```
73  // An instance of simplePonzi intended for testing.
74  // In this module, we wire our system and test a state machine.
75  module simplePonziTest {
76    pure val addr = Set("alice", "bob", "charlie", "eve")
77
78    import simplePonzi.*
79
80    var evmState: EvmState
81    var ponziState: PonziState
82
83    // initialize the state machine
84    action init = all {
85      // every account has 10000 tokens initially
86      evmState' = { balances: addr.mapBy(a => 10000) },
87      // alice creates a single Ponzi contract
88      ponziState' = newPonzi("alice"),
89    }
90
91    // When investor is sending amount tokens.
92    // This action succeeds and updates the state of the state machine,
93    // only if no error is returned.
94    action onReceive(investor: Addr, amount: int): bool = {
95      val result = receive(evmState, ponziState, investor, amount)
96      all {
97        result.error == "",
98        evmState' = result.evm,
99        ponziState' = result.ponzi,
100     }
101   }
102
```

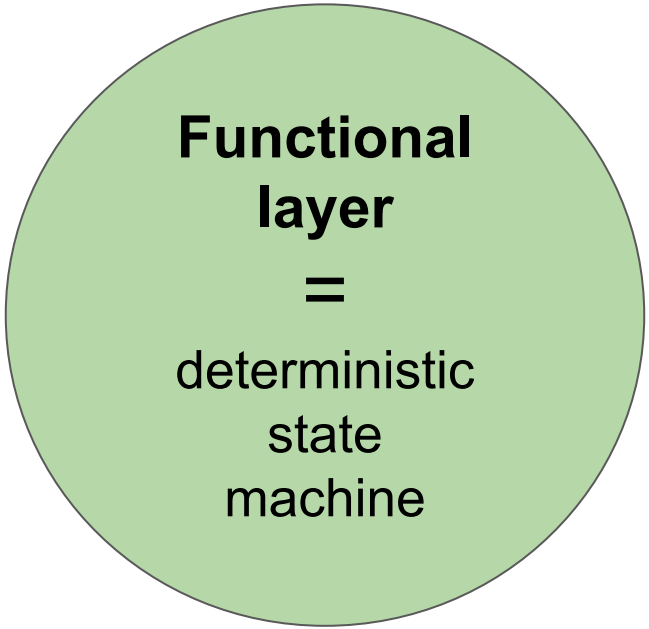
informal SYSTEMS

let’s see Ponzi in VSCode...

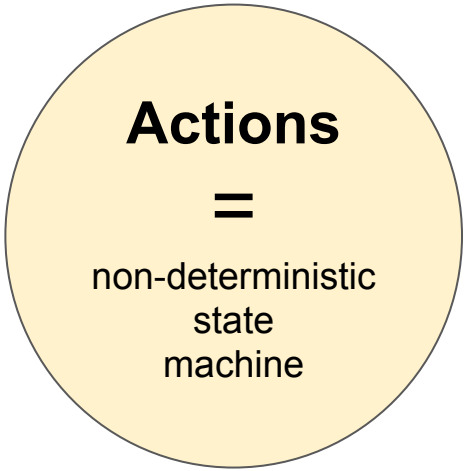
A warm-up exercise

Add a gas fee of 200 tokens on every transaction

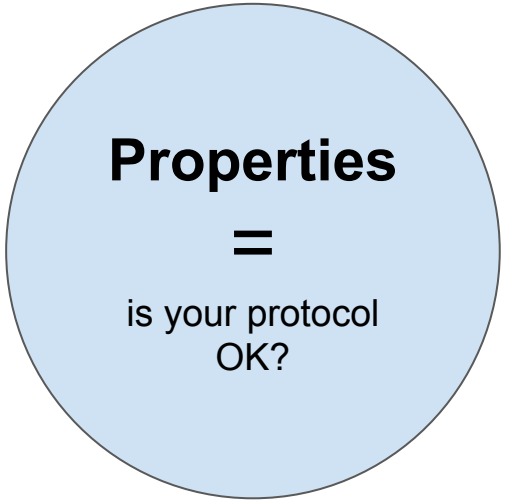
```
// When investor is sending amount tokens.
// This action succeeds and updates the state of the state machine,
// only if no error is returned.
action onReceive(investor: Addr, amount: int): bool = {
  val result = receive(evmState, ponziState, investor, amount)
  all {
    result.error == "",
    evmState' = result.evm,
    ponziState' = result.ponzi,
  }
}
```



concepts from programming languages



little stuff to learn: similar to testing



the great value of Quint!

| Random simulator

- oneOf(*S*) randomly selects a set element
- special syntax form: nondet *x* = oneOf(S)

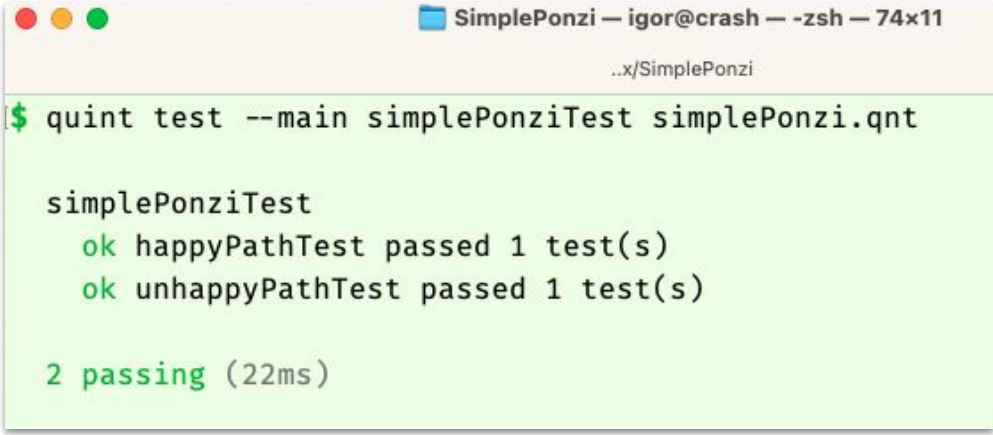
- any { A_1, ..., A_n } randomly selects an action
- produce runs up to --max-steps

- checks state invariants

```
action step =
  any {
    nondet sender = oneOf(ADDR)
    nondet amount = oneOf(AMOUNTS)
    any { // transfer
      nondet toAddr = oneOf(ADDR)
      submit(TransferTx(sender, toAddr, amount)),
      // approve and transferFrom
      ...
    },
    all {
      mempool != Set(),
      nondet tx = oneOf(mempool)
        commit(tx)
    }
  }
```

Tests

```
112  // A simple test to confirm our intuition.
113  // Run in by hand in REPL, or with `quint test`:
114  //
115  // $ quint test --main=simplePonziTest simplePonzi.qnt
116  run happyPathTest = {
117    init
118      .then(onReceive("bob", 100))
119      .then(onReceive("eve", 110))
120      .then(onReceive("alice", 121))
121      .then(all {
122        assert(ponziState.currentInvestor == "alice"),
123        assert(ponziState.currentInvestment == 121),
124        assert(evmState.balances.get("alice") == 9979),
125        assert(evmState.balances.get("eve") == 10011),
126        evmState' = evmState,
127        ponziState' = ponziState,
128      })
129  }
130
131  // a simple test to see that smaller investments are not allowed
132  run unhappyPathTest = {
133    init
134      .then(onReceive("bob", 100))
135      .then(onReceive("eve", 105))
136      .fail()
137  }
```

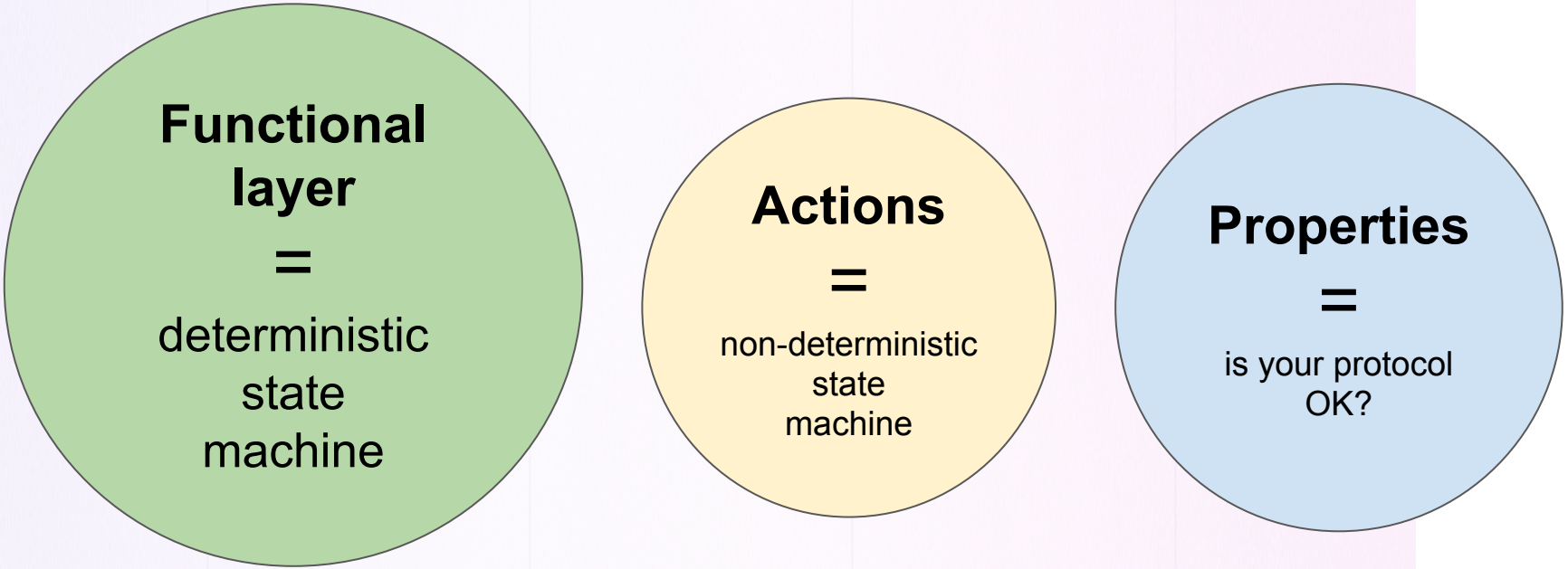


| Intermediate Summary

Done: An interactive introduction to Quint

Next: A structured introduction to the language

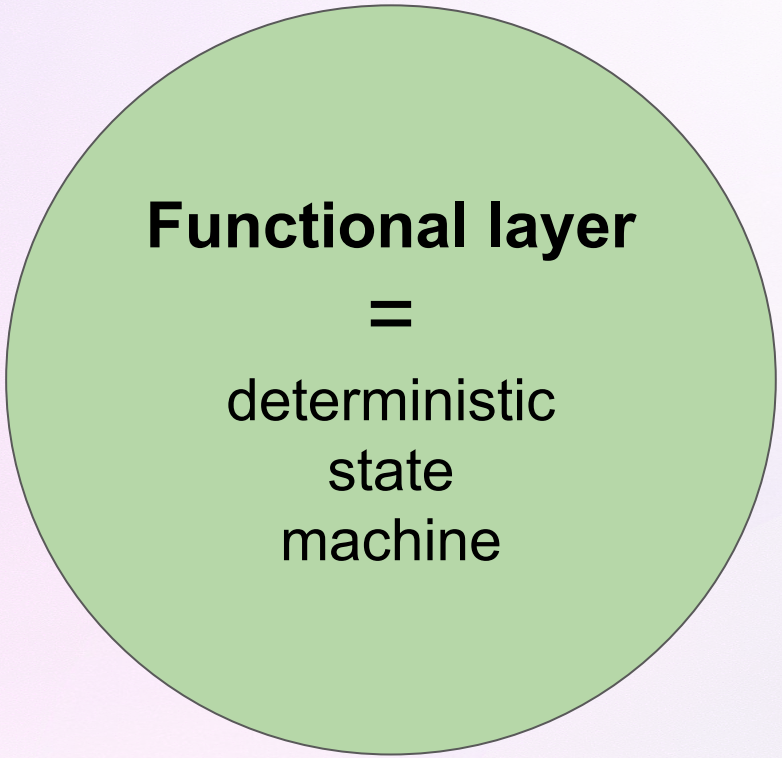
The language of Quint



concepts from programming languages

little stuff to learn: similar to testing

the great value of Quint!



1. **Bare minimum**

2. Advanced data structures

| Data: String literals

- “Hello, world!”
- “Server 1”
- Names and distinct constants of type `str`
- The only available operators are equality and inequality:

s1 == s2	s1 != s2
“A” == “A”	“A” != “B”

| Data: Booleans

Values and names of type `bool: false, true`

Operators:

not(*b*)	*p* == *q*	*p* != *q*
p and *q*	and { p_1, …, p_k }	
p or *q*	or { p_1, …, p_k }	
p implies *q*	*p* iff *q*	

| Data: Big Integers

- Values and names of type `int`: `0`, `1`, `-1`, `2`, `-2`, `3`, `-3`, …,

  `340282366920938463463374607431768211459`, `123_000`, `0x12abcd`

- No fixed bit width, like in math

Operators:	`i^j`				
	`-i`				
`i * j`	`i / j`	`i % j`			
`i + j`	`i - j`				
`i == j`	`i != j`	`i < j`	`i <= j`	`i > j`	`i >= j`

| Data: Tuples

Constructor:

```
("Quint", 2023, true)
// of type (str, int, bool)
```

Operators:

```
// access a tuple element via index
t._1        t._2    t._3    …       t._50
// the set of tuples (x_1, …, x_n) over S_1, …, S_n
tuples(S_1, …, S_n)
```

Data: Records

Constructor:

```
{
  name: “Quint”,
  version: 14
}
// of type { name: str, age: int }
```

Operators:

```
// access a record field by name
r.name          r.version

// make a new record ‘fr’ that is like ‘r’ except
// the mentioned fields are updated
{ …r, version: 20 }
```

| Control: if-else

Example:

```
if (x > y) x else y
```

Syntax:

```
if (expr) expr else expr
```

| Control: Pure values and operators

Example:

```
pure val N = 42

pure def max(x, y) =
  if (x > y) x else y
```

Syntax:

```
pure val name = expr
pure val name: type = expr
pure def name(name_1, …, name_n) = expr
pure def name(name_1: type_1, …, name_n: type_n): type =
expr
```

| Application forms: Normal and UCFS

Two interchangeable forms:

$$f(x_1, x_2, \ldots, x_n)$$

$$x_1.f(x_2, \ldots, x_n)$$

Example:

```
max(3, 4)
```

```
3.max(4)
```

| Control: Higher-order operators

Example:

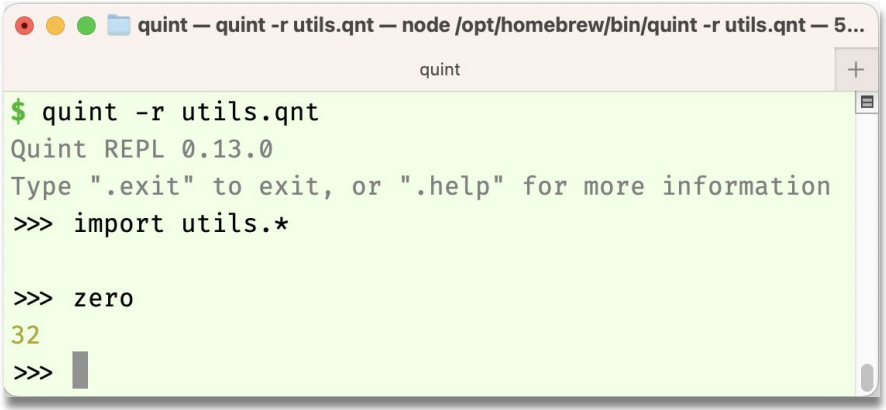
```
pure def double(x) = x + x

pure def f(g, x) = g(x)

pure val k = f(double, 11)
// Q: what is the value of k?
```

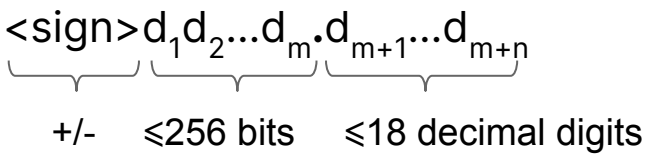
| Control: Modules

```
// file: units.qnt
module units {
  pure def fahrenheit(celsius) = {
    celsius * 9 / 5 + 32
  }
}
// file: utils.qnt
module utils {
  import units.* from “units”
  pure val zero = fahrenheit(0)
}
```



| Example: Cosmos Decimals

Representing fixed point arithmetic with integers:



https://github.com/informalsystems/quint-sandbox/tree/main/decimal

https://github.com/cosmos/cosmos-sdk/blob/v0.46.4/types/decimal.go

| Checking decimals with a state machine

```
var opcode: str
var opArg1: Dec
var opArg2: Dec
var opResult: Dec

action init = all {
    opcode' = "nop",
    opArg1' = toDec(0),
    opArg2' = toDec(0),
    opResult' = toDec(0),
}
```

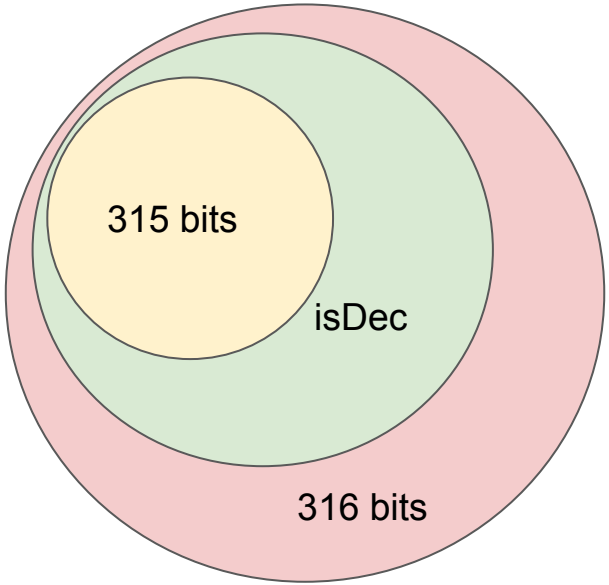
```
// apply a binary operator
action applyBinary(name: str, f: (Dec, Dec) => Dec): bool = {
    nondet whole1 = (-2^256 + 1).to(2^256 - 1).oneOf()
    nondet frac1 = (-10^18 + 1).to(10^18 - 1).oneOf()
    nondet whole2 = (-2^256 + 1).to(2^256 - 1).oneOf()
    nondet frac2 = (-10^18 + 1).to(10^18 - 1).oneOf()
    pure val d1: Dec = { error: false, value: whole1 * ONE + frac1 }
    pure val d2: Dec = { error: false, value: whole2 * ONE + frac2 }
    all {
        opcode' = name,
        opArg1' = d1,
        opArg2' = d2,
        opResult' = f(d1, d2),
    }
}
```

| Checking decimals

Checking properties:

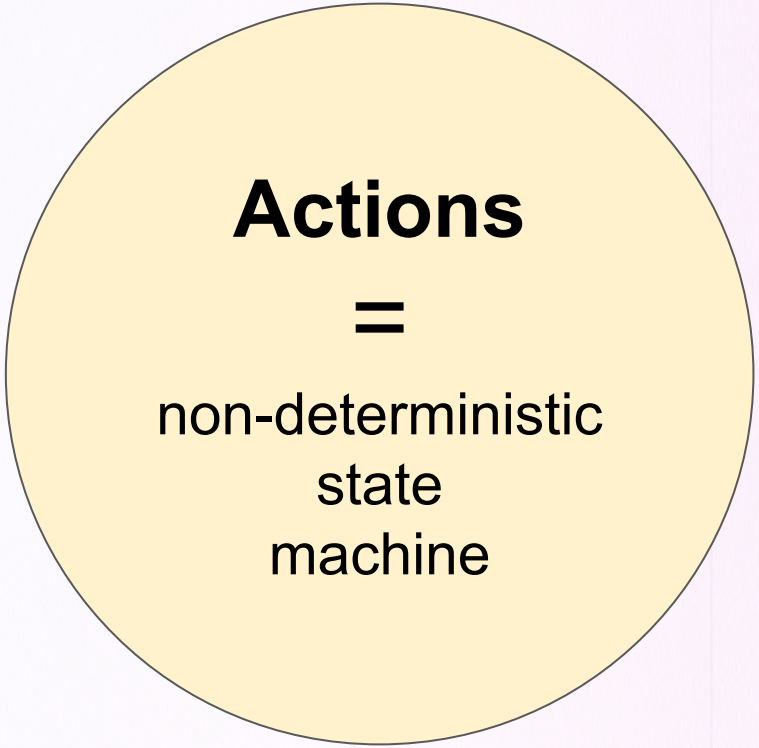
- isDecWhenNoError ✅ in v0.45.1 ❌ in v0.46.4
- noErrorWhenIsDec ❌ in v0.45.1 ✅ in v0.46.4
- isDecWhenNoError for all Dec constructors

- Identify more properties and check?

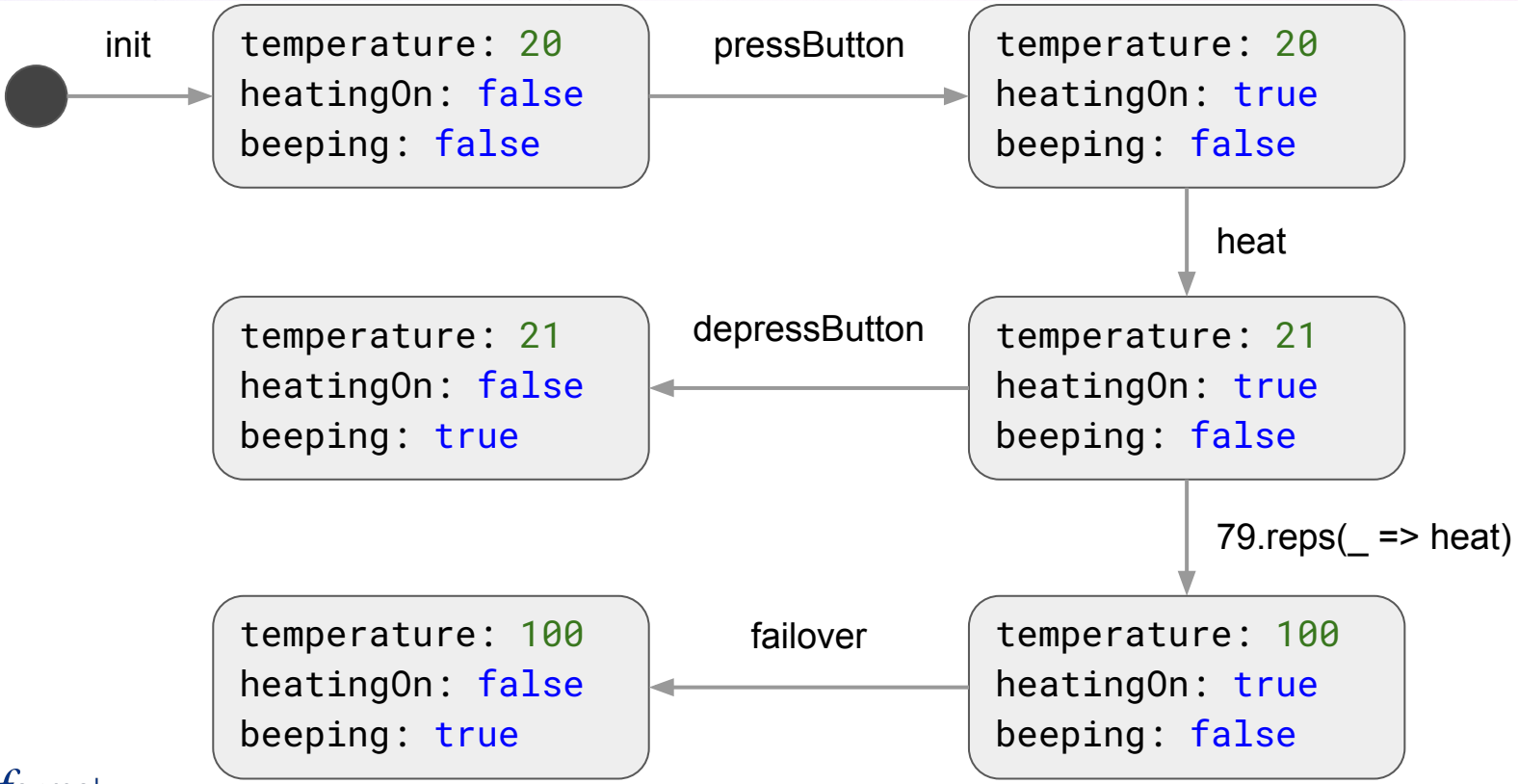


let’s switch to VSCode...

we will come back to this example later

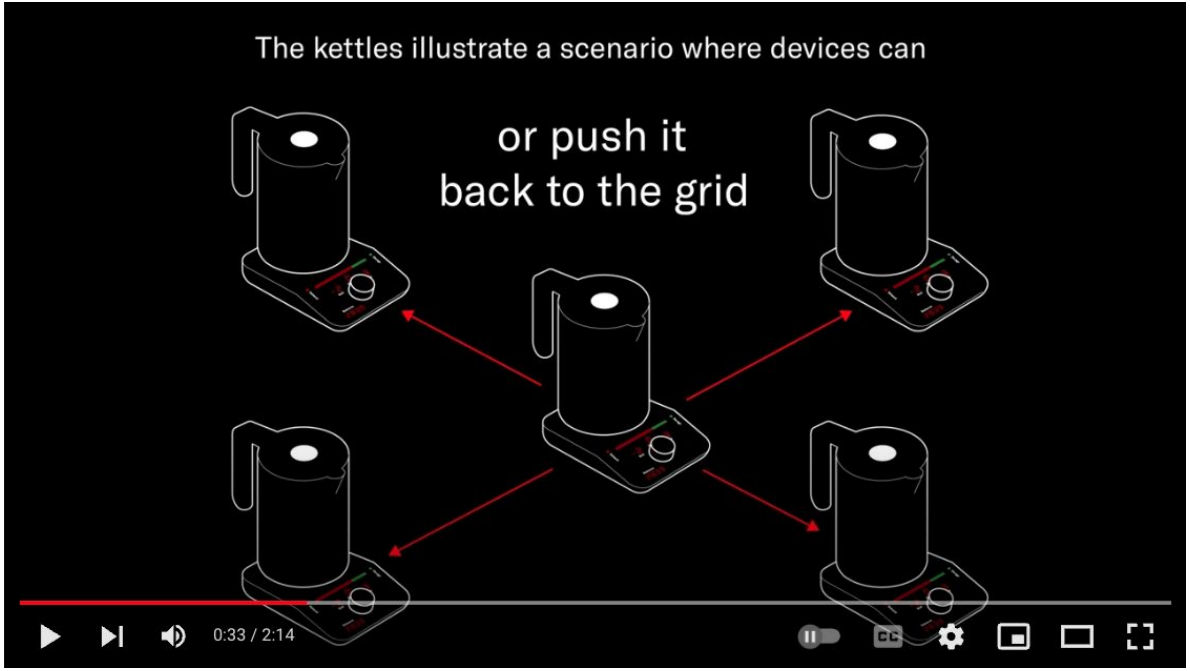

Actions
=
non-deterministic
state
machine

The state machine of a kettle



Blockchains?

Search for “**Karma Kettle**” by Larissa Pschetz on YouTube



Source: https://www.youtube.com/watch?v=Q_eLfcbSsr0

State variables

```
var temperature: int
var heatingOn: bool
var beeping: bool
```

Syntax: `var <name>: <type>`

- State variables must be assigned a variable in every state
- Similar to global variables in programming languages

Operators on actions

```
// delayed assignment: x == e in a successor state
x' = e

// true iff all actions evaluate to true
// (the assignments propagate)
all { A_1, …, A_n }

// true iff one of the actions evaluates to true
// (the assignments propagate)
any { A_1, …, A_n }

// non-deterministically pick one element of
// a non-empty set S, then evaluate expr
nondet x = oneOf(S)
expr
```

| User-defined actions

Example:

```
action heat = all {
  heatingOn,
  temperature < 100,
  temperature' = temperature + 1,
  heatingOn' = true,
  beeping' = false,
}
```

Syntax:

```
pure val name = expr
pure val name: type = expr
pure def name(name_1, …, name_n) = expr
pure def name(name_1: type_1, …, name_n: type_n): type =
expr
```

Stateful definitions

Example:

```
var temperature: int

def isColder(t) = temperature < t

// isColder is not pure, as it uses ‘temperature’
```

Syntax:

```
val name = expr
val name: type = expr
def name(name_1, …, name_n) = expr
def name(name_1: type_1, …, name_n: type_n): type = expr
```

| Effects

Every operator has an effect: the state variables it reads and modifies

```
var temperature: int

// reads(t) modifies()
def isColder(t) = temperature < t

action failover = all {
  // reads(heatingOn, temperature)
  // modifies(heatingOn, beeping, temperature)
  heatingOn,
  temperature >= 100,
  heatingOn' = false,
  beeping' = true,
  temperature' = temperature,
}
```

| Effect checker

- Every state variable may be modified at most once
- Arguments of all and any must modify the same set of variables

```
action heatOrOffNotOk = any {
  // modifies(temperature)
  all { heatingOn, temperature' = temperature + 1 },
  // modifies(heatingOn)
  all { heatingOn = false },
}

action heatOrOff = any {
  // modifies(heatingOn, temperature)
  all { heatingOn, temperature' = temperature + 1, heatingOn' = true },
  // modifies(heatingOn, temperature)
  all { heatingOn = false, temperature' = temperature },
}
```

Two special actions

A specification of a state machine normally has two special actions:

- An initializer (`init`): modifies all state variables, reads none
- A transition (`step`): modifies all state variables, may read some

```
action init = all {
  temperature' = 20,
  heatingOn' = false,
  beeping' = false,
}
```

```
action step = any {
  pressButton,
  heat,
  depressButton,
  failover,
}
```

| Remember non-determinism

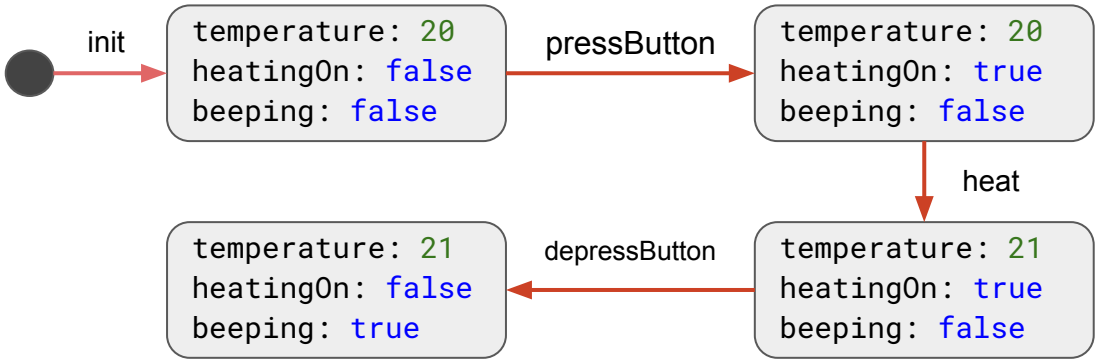
We can have multiple initial states:

```
action initNondet = all {
  nondet temp = (-40).to(40).oneOf()
  temperature' = temp,
  heatingOn' = false,
  beeping' = false,
}
```

Runs

The happy and unhappy paths. Think of tests!

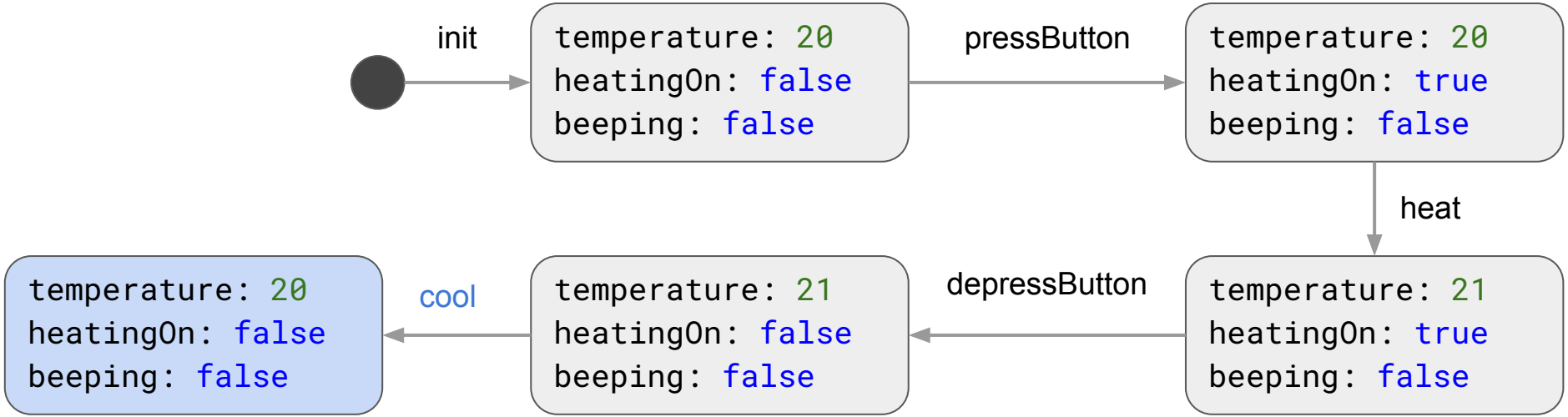
```
// depressing a button should work
run depressTest = {
  init
    .then(pressButton)
    .then(heat)
    .then(depressButton)
}
```



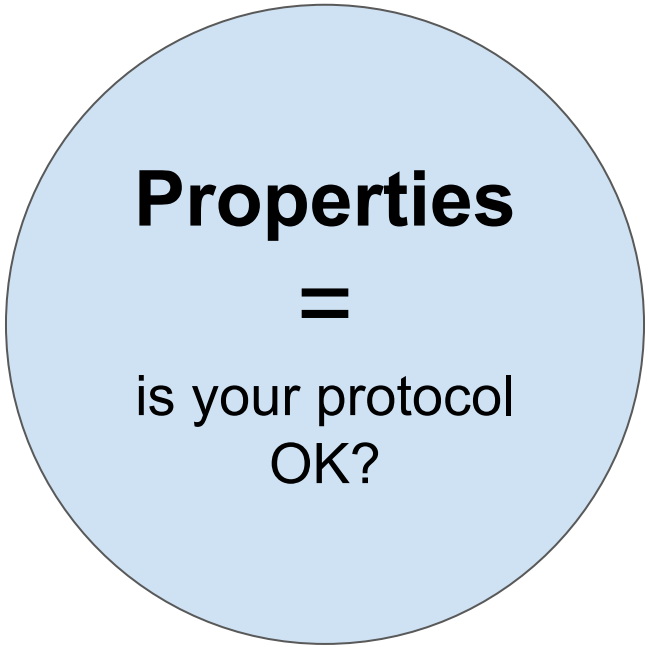
| Engineers love tests!

| Cool down exercise

Implement an action ‘cool’ that reduces the temperature



let’s see the kettle example in VSCode...


Properties
=
is your protocol
OK?

State invariants

```
// if no error is reported,
// then the result is a proper a decimal
def isDecWhenNoError =
    not(opResult.error) implies isDec(opResult.value)

// if the result is a proper decimal, then no error is reported
// (unless there is a division by zero)
def noErrorWhenIsDec =
    isDec(opResult.value) implies or {
        not(opResult.error),
        opcode == "quo" and opArg2.value == 0,
        opcode == "quoTruncate" and opArg2.value == 0,
    }
```

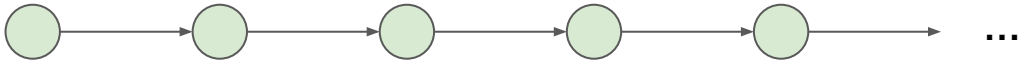
| Action invariants

Check an invariant for two consecutive states:

```
temporal noCooldownInv =
    next(temperature) >= temperature
```

| Temporal properties 🚧

```
temporal noOverheat =
  always(temperature <= 100)
```



```
temporal eventuallyOff =
  eventually(not(heatingOn))
```

```
temporal onThenOff =
  always(heatingOn implies eventually(not(heatingOn)))
```

| Advanced temporal operators 🚧

Following TLA$^+$, Quint offers advanced temporal operators:

- `A.orKeep(vars)`
- `A.mustChange(vars)`
- `enabled(A)`
- `A.weakFair(vars)`
- `A.strongFair(vars)`

You can write them in your specifications, but there is no tool support

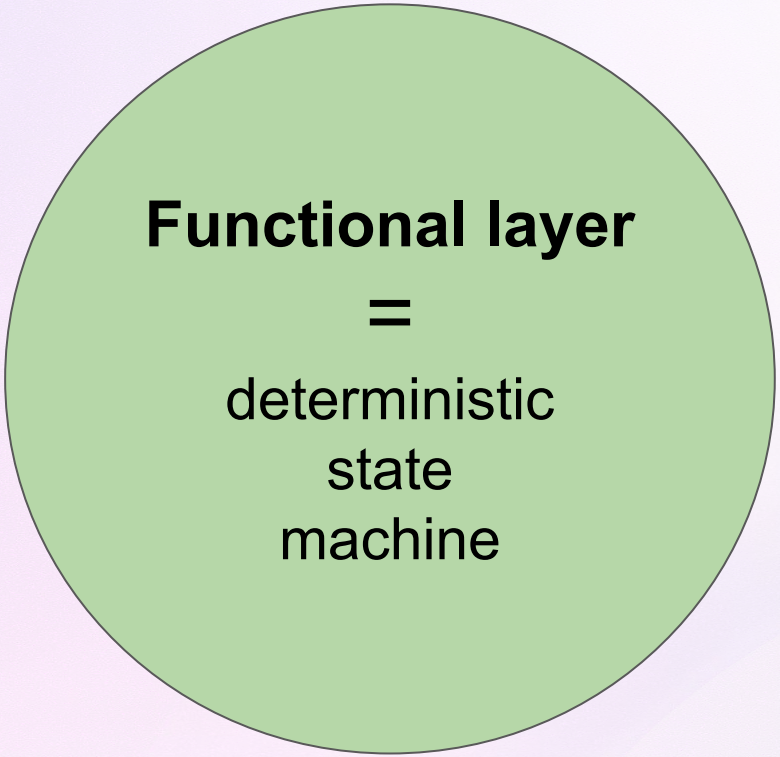
| Intermediate Summary

Done:

- A simple functional layer
- Actions
- Temporal operators

Many verification tools have comparable input languages

Next: We will see more powerful language primitives



1. Bare minimum

2. Advanced data structures

Appetizer: Distributed Consensus

| Fault-tolerant distributed systems

Distributed

logically and geographically

Fault-tolerant

individual machines may crash and even act malicious

Safe and live

e.g., no double spending

every transaction is eventually committed

| Properties of Distributed Consensus

A distributed algorithm for N replicas

every replica proposes a value $w \in V$

Termination

every correct replica eventually decides on a value $v \in V$

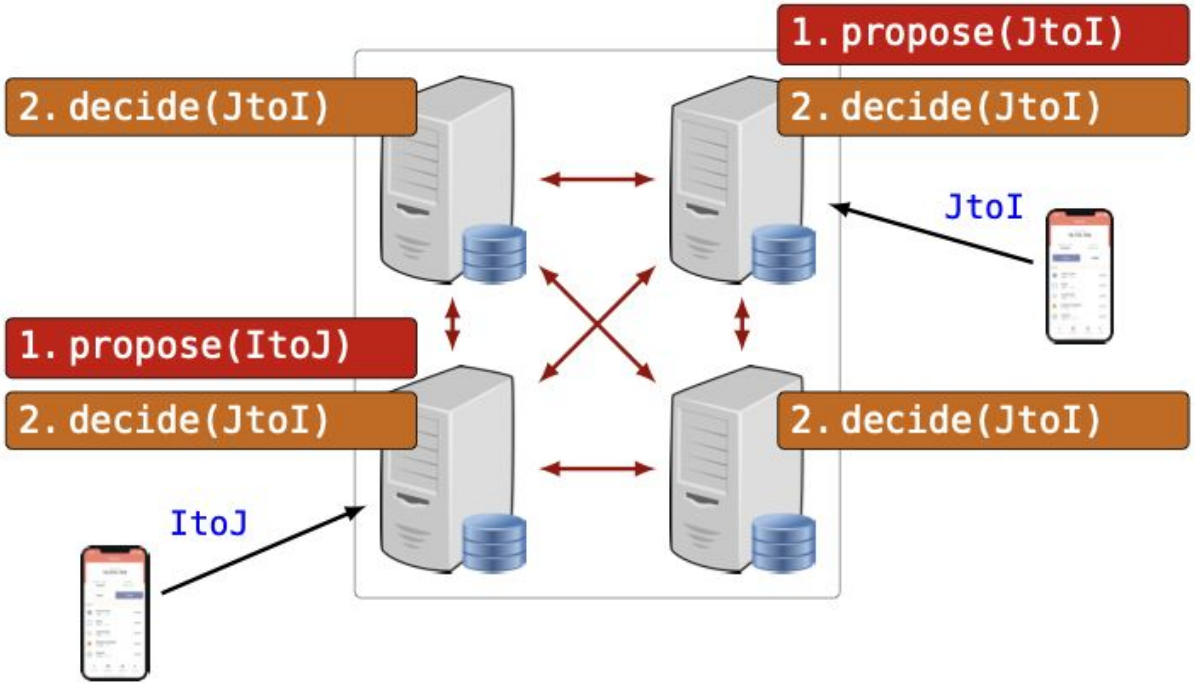
Agreement

if a replica decides on v, no replica decides on $V \setminus \{v\}$

Validity

if a replica decides on v, the value v was proposed earlier

| How is consensus useful?



| Asynchrony

r_1 sends/receives on Monday/Thursday, computes on Friday

r_2 sends/receives/computes once a month

r_3 went for a two-month vacation

r_4 left job without notice

r_1 uses DHL, r_2 uses LA POSTE, r_3 uses Post

| Consensus and (partial) asynchrony

Various processor speeds

Various message delays, unbounded but finite

Consensus is not solvable [Fischer, Lynch, Paterson, 1985]

Practical consensus algorithms:

- termination is the engineering problem, **Paxos**
- or restrict asynchrony, **DLS88, Tendermint**
- or prove almost-sure termination **Ben-Or**

| BFT Consensus

What if some replicas lie?

This is **Byzantine** behavior [Lamport, Shostak, Pease, 1982]

More than two-thirds must be correct: $n > 3t$

e.g., Tendermint

CometBFT

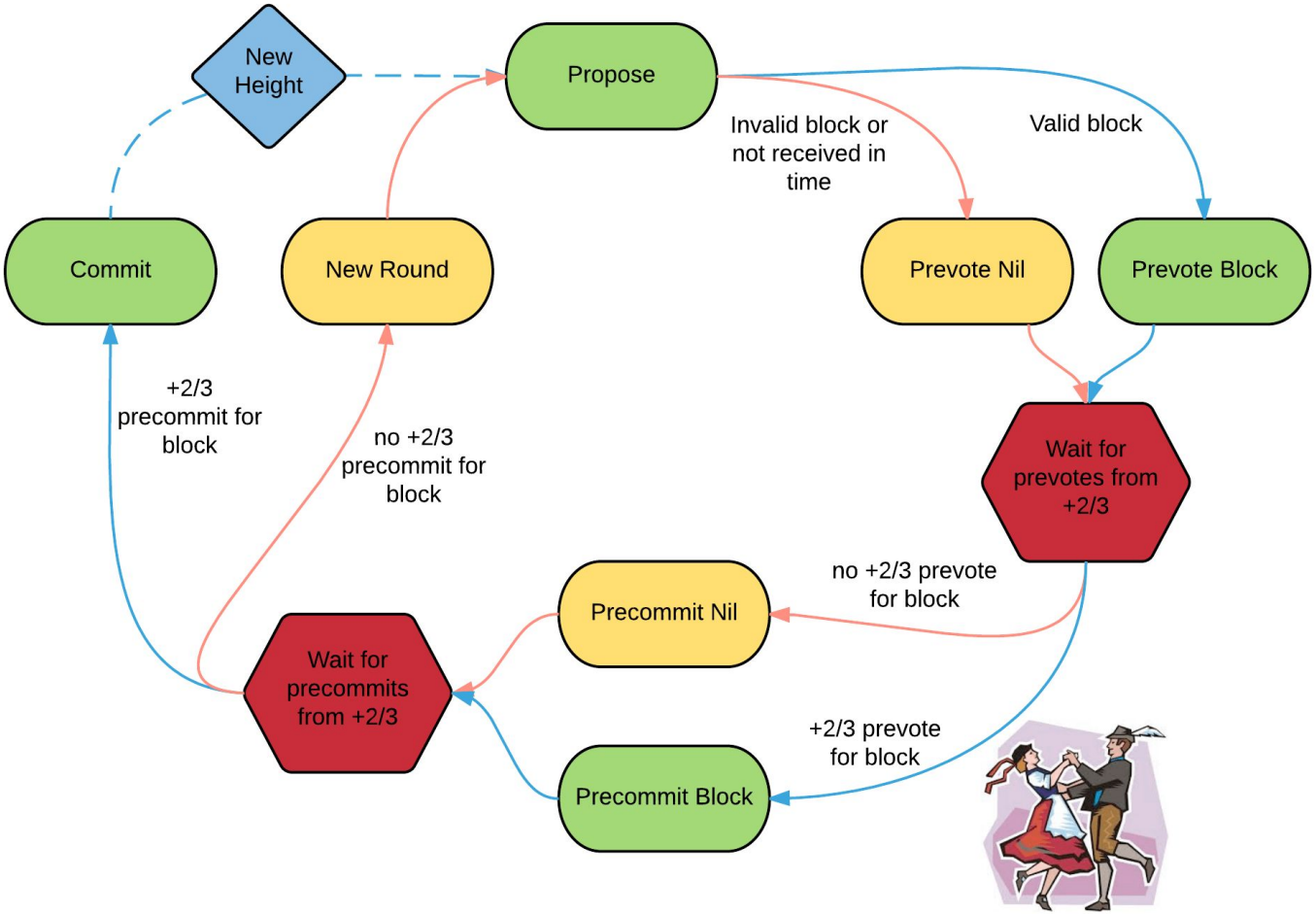
| Partial synchrony

Liveness is only guaranteed under:

Gossip communication

If a correct process *p* sends some message *m* at time *t*, all correct processes will receive m before max(*t*, GST) + Δ. Furthermore, if a correct process *p* receives some message *m* at time *t*, all correct processes will receive *m* before max(*t*, GST) + Δ.

The latest gossip on BFT Consensus (2018)


New Height
Propose
Invalid block or not received in time
Valid block
Commit
New Round
Prevote Nil
Prevote Block
+2/3 precommit for block
no +2/3 precommit for block
Wait for prevotes from +2/3
no +2/3 prevote for block
Precommit Nil
Wait for precommits from +2/3
+2/3 prevote for block
Precommit Block

Algorithm 1 Tendermint consensus algorithm

1: **Initialization:**
2: $h_p := 0$
3: $round_p := 0$
4: $step_p \in \{propose, prevote, precommit\}$
5: $decision_p[] := nil$
6: $lockedValue_p := nil$
7: $lockedRound_p := -1$
8: $validValue_p := nil$
9: $validRound_p := -1$
10: **upon** start **do** $StartRound(0)$
11: **Function** $StartRound(round)$:
12: $round_p \leftarrow round$
13: $step_p \leftarrow propose$
14: **if** proposer$(h_p, round_p) = p$ **then**
15: **if** $validValue_p \neq nil$ **then**
16: $proposal \leftarrow validValue_p$
17: **else**
18: $proposal \leftarrow getValue()$
19: **broadcast** $\langle$PROPOSAL$, h_p, round_p, proposal, validRound_p\rangle$
20: **else**
21: **schedule** $OnTimeoutPropose(h_p, round_p)$ to be executed **after** $timeoutPropose(round_p)$

22: **upon** $\langle$PROPOSAL$, h_p, round_p, v, -1\rangle$ **from** proposer$(h_p, round_p)$ **while** $step_p = propose$ **do**
23: **if** $valid(v) \wedge (lockedRound_p = -1 \vee lockedValue_p = v)$ **then**
24: **broadcast** $\langle$PREVOTE$, h_p, round_p, id(v)\rangle$
25: **else**
26: **broadcast** $\langle$PREVOTE$, h_p, round_p, nil\rangle$
27: $step_p \leftarrow prevote$

28: **upon** $\langle$PROPOSAL$, h_p, round_p, v, vr\rangle$ **from** proposer$(h_p, round_p)$ **AND** $2f + 1$ $\langle$PREVOTE$, h_p, vr, id(v)\rangle$ **while** $step_p = propose \wedge (vr \geq 0 \wedge vr < round_p)$ **do**
29: **if** $valid(v) \wedge (lockedRound_p \leq vr \vee lockedValue_p = v)$ **then**
30: **broadcast** $\langle$PREVOTE$, h_p, round_p, id(v)\rangle$
31: **else**
32: **broadcast** $\langle$PREVOTE$, h_p, round_p, nil\rangle$
33: $step_p \leftarrow prevote$

The latest gossip on BFT Consensus (2018)

Pen & paper math proofs

```
307   \* lines 34-35 + lines 61-64 (onTimeoutPrevote)
308   UponQuorumOfPrevotesAny(p) ==
309     /\ step[p] = "PREVOTE" \* line 34 and 61
310     /\ \E MyEvidence \in SUBSET msgsPrevote[round[p]]:
311         \* find the unique voters in the evidence
312         LET Voters == { m.src: m \in MyEvidence } IN
313         \* compare the number of the unique voters against the threshold
314         /\ Cardinality(Voters) >= THRESHOLD2 \* line 34
315         /\ evidencePrevote' = MyEvidence \union evidencePrevote
316         /\ BroadcastPrecommit(p, round[p], NilValue)
317         /\ step' = [step EXCEPT ![p] = "PRECOMMIT"]
318         /\ UNCHANGED <<round, decision, lockedValue, lockedRound,
319                        validValue, validRound, msgsPropose, msgsPrevote,
320                        evidencePropose, evidencePrecommit>>
321         /\ action' = "UponQuorumOfPrevotesAny"
322
323   \* lines 36-46
324   UponProposalInPrevoteOrCommitAndPrevote(p) ==
325     \E v \in ValidValues, vr \in RoundsOrNil:
326       /\ step[p] \in {"PREVOTE", "PRECOMMIT"} \* line 36
327       /\ LET msg == [
328             type       |-> "PROPOSAL",
329             src        |-> Proposer[round[p]],
330             round      |-> round[p],
331             proposal   |-> v,
332             validRound |-> vr
333           ]
334           IN
335            /\ msg \in msgsPropose[round[p]] \* line 36
336            /\ LET PV == { m \in msgsPrevote[round[p]]: m.id = Id(v) } IN
337              /\ Cardinality(PV) >= THRESHOLD2 \* line 36
338              /\ evidencePrevote' = PV \union evidencePrevote
```

TLA$^+$ specification (2019-2020)

Bounded model checking
for n ∈ 4..7, f ∈ 0..2, 3-4 rounds

Proving inductive invariant
for n ∈ 4..7, f ∈ 0..2, 3-4 rounds

apalache.informal.systems

Formally Verifying the Tendermint Blockchain Protocol

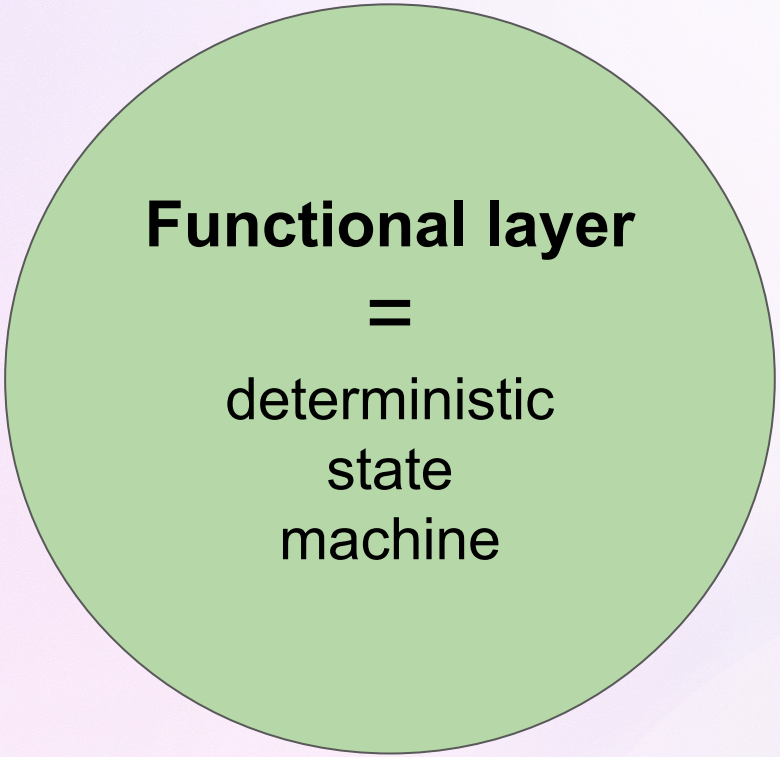
TUESDAY, JULY 20, 2021

CRYPTOGRAPHY, FORMAL METHODS

, Giuliano Losa

Parameterized proof with Ivy (2021)

Distributed protocols enable components such as blockchain validator nodes, cloud servers, or IoT devices to coordinate and cooperate toward a common goal. However, in such a diverse environment, a lot of things can go wrong: hardware can fail, software can be buggy, network links can be unreliable, attackers may compromise components, and so on. Due to those difficulties, distributed protocols present a fascinating target for formal verification.



1. Bare minimum

2. Advanced data structures

Sets

- The **core data structure**
- Type `Set[T]`, no restrictions on T
- The most common operators:

```
Set(e_1, …, e_k)        // the set { e_1, …, e_k }
S.contains(e), e.in(S)  // e ∈ S
S.subseteq(T)           // S ⊆ T
S.union(T)              // S ∪ T
S.intersect(T)          // S ∩ T
S.exclude(T)            // S \ T
```

| Finite and Infinite Sets

Two operators designed for finite sets:

```
isFinite(S)    // true, if the class is finite

size(S)        // set cardinality
```

Three built-in sets:

```
Int, Nat       // infinite sets of integers and naturals

Bool == Set(false, true)
```

| Sets – Examples

```
pure val Correct = Set(“p1”, “p2”, “p3”) // correct validators
pure val Faulty = Set(“p4”)              // faulty validators
pure val Proc = Correct.union(Faulty)
var evidencePrevote: Set[Pre_t]
action UponProposalInProposeAndPrevote(p, v, vr) = all {
  …
  msg.in(msgsPropose.get(round.get(p))),
  size(PV) >= THRESHOLD2,
  evidencePrevote' = PV.union(evidencePrevote),
```

Set quantifiers & comprehensions

```
S.exists(x => P)        // ∃x∈S: P

S.forall(x => P)        // ∀x∈S: P

S.filter(x => P)        // {x∈S: P}

S.mapBy(x => e)         // {e: x∈S}
```

| Quantifiers & comprehensions – Examples

```
val AmnesiaImpliesEquivocation =
  Faulty.exists(p => AmnesiaBy(p))
    implies Faulty.exists(EquivocationBy)

val PV = msgsPrevote.get(vr).filter(m => m.id == Id(v))
  …
val Voters = Evidence.map(m => m.src)
val Validity = Corr.forall(p =>
  decision.get(p).in(ValidValues.union(Set(NilValue))))
```

| Folding sets

A way to iterate over sets:

```
S.fold(start, (a, x) => f)
```

Similar to a loop in imperative languages:

```
let result = start

for (let x of S) {

  result = f(result, x)

}
```

| Folding Sets – Example

```
val noLeftoversInv = {
  val sumRewards = addr.fold(0,
    (s, a) => s + ponziState.rewards.get(a)
  )
  sumRewards == evmState.balances.get("contract")
}
```

| Folding Sets – Example

```
val noLeftoversInv = {
  val sumRewards = addr.fold(0,
    (s, a) => s + ponziState.rewards.get(a)
  )
  sumRewards == evmState.balances.get("contract")
}
```

There is a catch:
The operator passed to `fold` must be commutative
`f(x, y) == f(y, x)`

| Anonymous operators or lambdas

As you have seen in the examples:

(x_1, ..., x_n) ⇒ e

Pass them to set operators and user-defined HO operators

You cannot return them

| Powersets and unions

```
powerset(S): Set[T] => Set[Set[T]]

  // the set of the subsets of S

flatten(S): Set[Set[T]] => Set[T]

  // the union of the elements of S


// Q: flatten(powerset(S)) == ?
```

| Final piece: chooseSome

For a non-empty set S, **deterministically** choose one element of S

```
chooseSome(S)
```

Property: if `S == T`, then `chooseSome(S) == chooseSome(T)`

Quint simulator: not implemented yet

Apalache: implements a non-deterministic version

| Maps

- The **second most used** data structure (after sets)
- Type `a -> b`, no restrictions on `a` and `b`

```
Map(k_1 -> e_1, …, k_n -> e_n) // maps k_i to e_i
keys(M)                        // the domain of M
S.mapBy(k => e)                // maps k to e (may refer to k)
M.put(k, v)                    // add/update a pair k -> v
M.get(k), M.set(k, v)          // retrieve and update
M.setBy(k, (old => e))         // update a pair k -> e[old/k]
S.setOfMaps(T)                 // all maps from S to T
```

let’s look at Tendermint in VSCode...

| Lists

- The **least used** data structure – as opposed to programming!
- Type List[T], no restrictions on T

```
[ e_1, …, e_n ]          // the list of e_1, …, e_n
range(start, end)        // the list [start, …, end - 1]
length(L)                // the number of elements
L[i]              // the ith element, 0 <= i < length(L)
L.concat(K)              // concatenating two lists
L.append(x)              // L.concat([x])
L.replaceAt(i, x)        // copy of L, but set to x at i
// more in the cheat sheet
```

Folding over lists

```
// similar to fold and reduce in many FP languages
list.foldl(start, (a, x) => f)
// example
pure def listForall(list: List[a], pred: a => bool): bool = {
  list.foldl(true, (b, elem) => {
   if (pred(elem)) b else false })
}
```

In contrast to fold, there is no catch :-)

| Intermediate Summary

Done: Essentially, the whole language of Quint

Very few verification tools have comparable input languages

Next: We will see the relationship with TLA$^+$

Formal semantics?

| Transpiling to TLA+

Many Quint operators have one-to-one correspondence with TLA+

Quint	TLA+	Quint	TLA+
`S.union(T)`	S \union T, S ∪ T	S.filter(x => P)	{ x \in S: P }
`powerset(S)`	SUBSET S	i > j	i > j
`p and q`	p ∧ q	p implies q	p => q
`always p`	[]p	eventually p	<>p
`if (p)` e1 `else` e2	IF p THEN e1 ELSE e2	x == y x != y not(p)	x = y x /= y ~p

| TLA+ functions

TLA+ has only math functions S → T, used as:

- **a function** of finite or infinite domains
- **a record**: a function Keys → T, where Keys ⊂ STRING
- **a tuple**: a function 1..n → T
- **a sequence**: essentially, a tuple but it has more operators

| Transpiling to TLA⁺ functions

Four data structures are translated as TLA⁺ functions:

- a map of type a → b
- a record of type { key_1: T_1, ..., key_n: T_n }
- a tuple of type (T_1, ..., T_n)
- a list of type List[T] is transpiled as a sequence (add +1 to the index!)

| Block disjunctions and conjunctions

Quint	TLA+	Quint	TLA+
and { A_1, ..., A_n }	⋀ A_1 ⋀ ... ⋀ A_n	or { A_1, ..., A_n }	⋁ A_1 ⋁ ... ⋁ A_n
all { A_1, ..., A_n }	⋀ A_1 ⋀ ... ⋀ A_n	any { A_1, ..., A_n }	⋁ A_1 ⋁ ... ⋁ A_n

| Assignments and non-determinism

Quint	TLA+
x' = e	x' = e
nondet x = oneOf(S) expr	∃ x ∈ S : expr

The Quint simulator is more restrictive:
It requires S to be non-empty

| User-defined operators

Quint	TLA+
(pure def|def|action|temporal) f(x_1, ..., x_n) = e	f(x_1, …, x_n) ≜ e
(pure def|def|action|temporal) f(x_1, ..., x_n) = e1 e2	LET f(x_1, …, x_n) ≜ e1 IN e2

| Back to TLA+? (1)

Complication #1

- TLA+ is untyped!
- Solution: Restrict to type-annotated specifications
- See: Apalache Snowcat

```
{ “A”, { “B”, {{ “C”, {}}, { “D”, {}}}}
```

| Back to TLA+? (2)

Complication #2

- TLA+ has RECURSIVE operators
- Solution: Use FoldSet and FoldSeq instead
- See: github.com/tlaplus/CommunityModules

| Back to TLA+? (3)

Complication #3

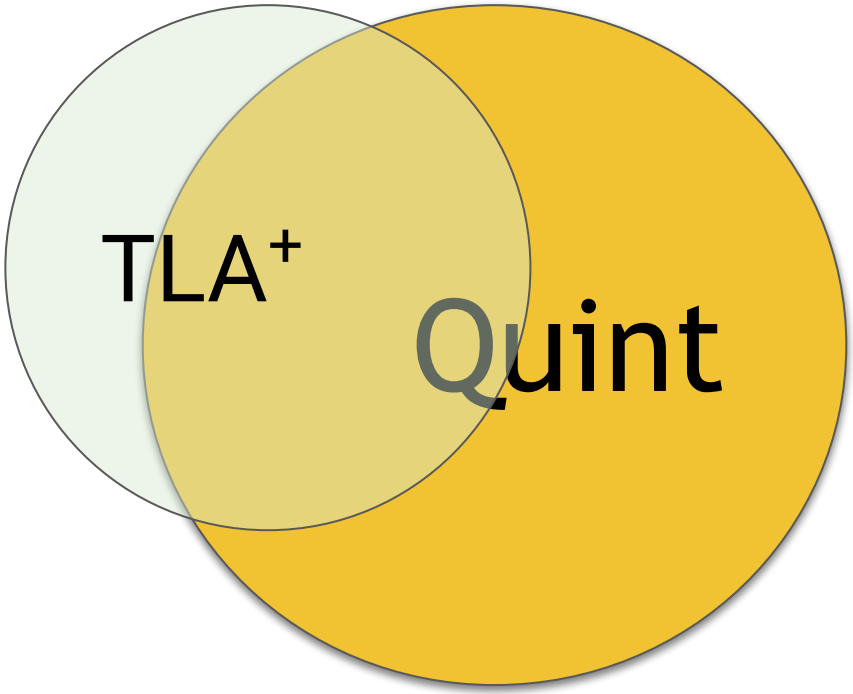
- TLA+ does not have assignments!

```
VARIABLES x, y

Init == y = 4 /\ x * x = 25

Next == y' = y + 1 /\ (x * x)' = y
```

- Solution: use expressions `x' = e`, if you can


TLA+
Quint

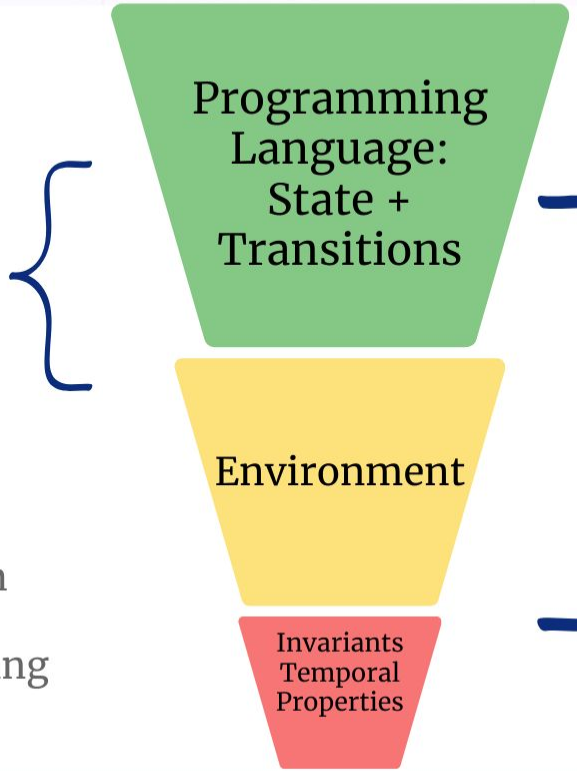
| Intermediate Summary

Done: Now you know the secret

- Quint is essentially another syntax for the logic of TLA$^+$
- People tell us it's another language

Next: We will talk about Apalache and symbolic model checking

Quint



Engineers

- Execution (TypeScript)
- Testing & Fuzzing
- Random simulation
- Interactive debugging

Auditors

Protocol designers

- Symbolic simulation
- Model checking

Protocol prototypes → *Correctness*

Apalache

Translation to SMT

APALACHE: model checker for TLA$^+$

- a symbolic model checker
- parameterized verification




Jure Kukovec Thanh-Hai Tran Marijana Lazić Josef Widder

TLA+ Model Checking Made Symbolic (OOPSLA'19)

Mimic the semantics implemented by TLC – explicit model checker

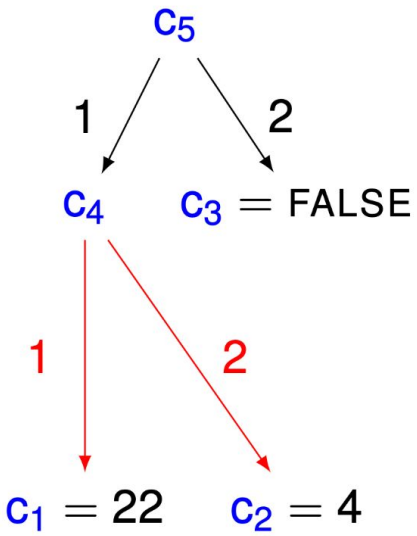
Compute layout of data structures, constrain contents with SMT

Define operational semantics via reduction rules – for bounded data structures

Trade efficiency for expressivity

Static picture of TLA$^+$ values and relations between them

Arena:



SMT:

integer `sort Int`

Boolean `sort Bool`

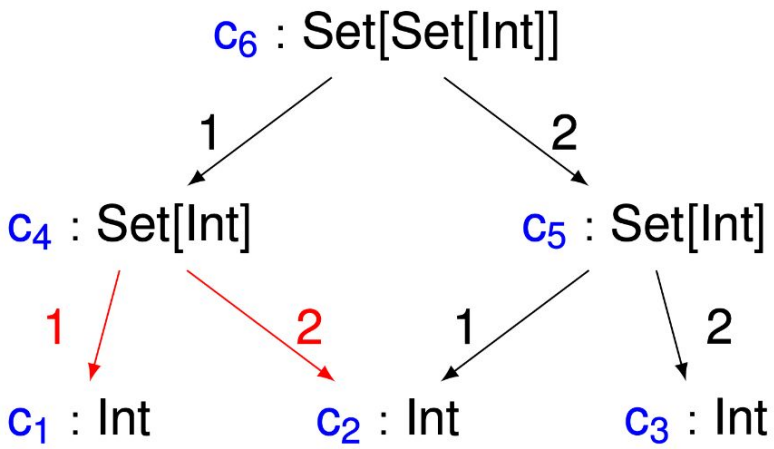
name, e.g., `"abc"`, uninterpreted sort

finite set:

- a constant c of uninterpreted sort set_τ
- propositional constants for members

  $in_{\langle c_1, c\rangle}, \ldots, in_{\langle c_n, c\rangle}$

Arenas for sets: { { 1, 2 }, { 2, 3} }



SMT defines the contents, e.g., to get $\{\{1\},\{2\}\}$:

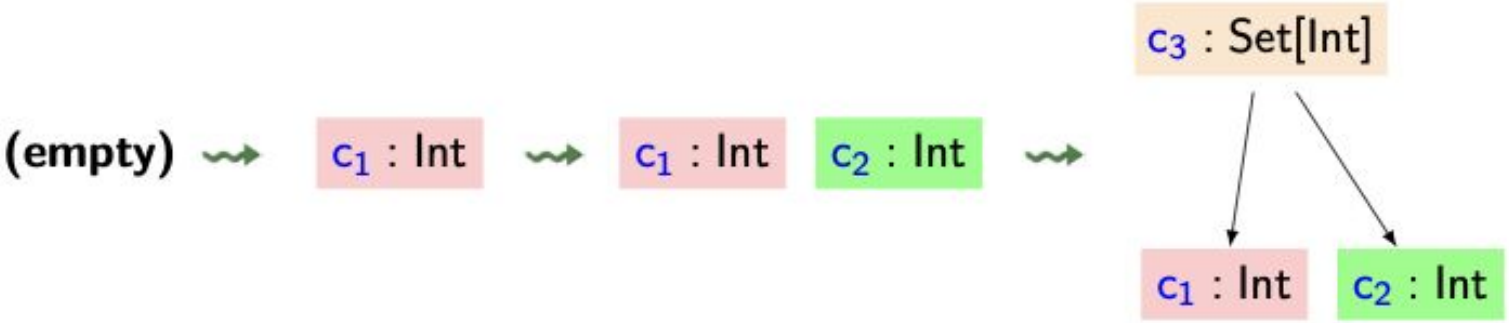
$$in_{\langle c_1,c_4\rangle} \wedge \neg in_{\langle c_2,c_4\rangle} \wedge in_{\langle c_2,c_5\rangle} \wedge \neg in_{\langle c_3,c_5\rangle}$$

Arenas

- Directed acyclic graphs
- Nodes represent symbolic values of TLA^+ expressions
- Edges represent **potential membership**

Rewriting the set construction

Corresponding arena

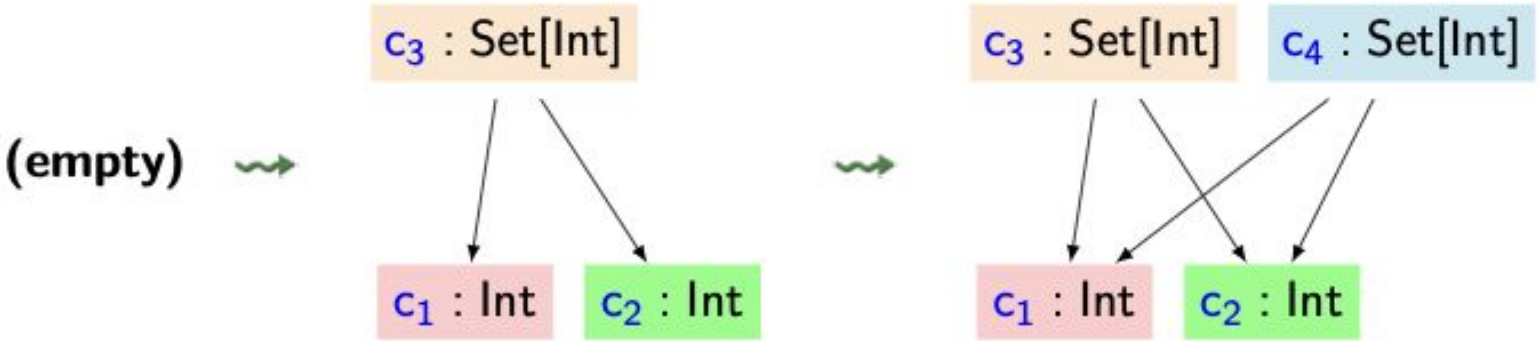


Rewriting and arenas

Rewriting the set filtering

$$\{x \in \{1, 2\} : p(x)\} \rightsquigarrow \{x \in c_3 : p(x)\} \rightsquigarrow c_4$$

Corresponding arena



SMT constraints

Generated constraints

$$en\langle c_4, 1, c_1\rangle \Leftrightarrow en\langle c_3, 1, c_1\rangle \wedge c_5 = \text{true}$$

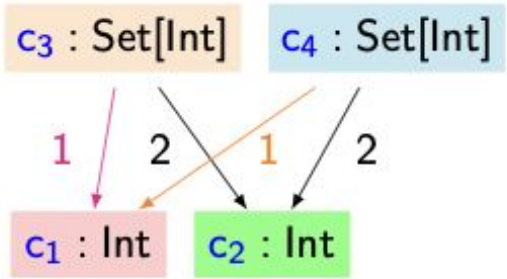
$$en\langle c_4, 2, c_2\rangle \Leftrightarrow en\langle c_3, 2, c_2\rangle \wedge c_6 = \text{true}$$

Set filtering

$\{x \in \{\,1\,, 2\,\} : p(x)\}$

$p(1) \rightsquigarrow c_5 : \text{Bool}$

$p(2) \rightsquigarrow c_6 : \text{Bool}$

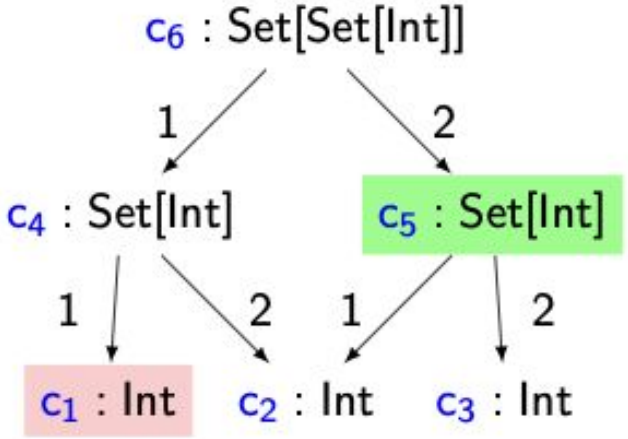


Arena	SMT encoding
Node	Uninterpreted constant
Edge	Unique Boolean constant

Nested sets

Nested sets

{ {1, 2}, {2, 3} }



SMT constraints

$$en\langle c_6, 1, c_4\rangle \wedge en\langle c_6, 1, c_5\rangle$$
$$\wedge\ en\langle c_5, 1, c_2\rangle \wedge en\langle c_5, 2, c_3\rangle$$
$$\wedge\ en\langle c_4, 1, c_1\rangle \wedge en\langle c_4, 1, c_2\rangle$$
$$\wedge\ c_1 = 1 \wedge c_2 = 2 \wedge c_3 = 3$$

What about integers?

TLA+

```
isUint(i) ≜ 0 ≤ i ∧ i ≤ MAX_UINT
isUint(state.totalSupply)
```

SMT

```
(and (<= 0 c_i) (<= c_i c_MAX_UINT))
```

TLA+

```
\* @type: (ADDR -> Int) => Int;
sumOverBalances(balances) ≜
  LET \* @type: (Int, ADDR) => Int;
    Add(sum, addr) ≜ sum + balances[addr]
  IN
  ApaFoldSet(Add, 0, DOMAIN balances)
```

SMT (after removing duplicates)

```
(= c_0 0)
(= c_1 (+ c_0 (ite in_1_S 1 0)))
...
(= c_n (+ c_{n-1} (ite in_n_S 1 0)))
```

Alternative SMT encoding

- Using SMT arrays for TLA$^+$ sets and functions: QF_AUFNIA

- Rodrigo Otoni, IK, J. Kukovec, P. Eugster, N. Sharygina

- Working faster on classical fault-tolerant algorithms

Symbolic Model Checking for TLA+ Made Faster (TACAS'23)

Our SMT encodings are not the only one

Several SMT encodings in TLAPS by Stephan Merz, Hernán Vanzetto:

- Encoding TLA$^+$ into unsorted and many-sorted first-order logic, 2018
- Harnessing SMT Solvers for TLA+ Proofs, 2012

Heavy use of quantifiers

Bounded model checking

Bounded model checking

Input: Init, Next and Inv

0. Init ∧ ¬Inv

1. (Init · Next) ∧ ¬Inv'

2. (Init · Next · Next) ∧ ¬Inv'

…

k. (Init · Next · … · Next) ∧ ¬Inv'

Backend:

SMT
(Microsoft z3)

k is the bound

| The TLA+ spec of Tendermint

Get the specification from:

https://github.com/informalsystems/cometbft/tree/main/spec/light-client/accountability

Run:

```
$ $APALACHE_HOME/bin/apalache-mc check --inv=Agreement\
 --cinit=ConstInit MC_n4_f2.tla
$ $APALACHE_HOME/bin/apalache-mc check --inv=Agreement\
 --cinit=ConstInit MC_n4_f1.tla
```

Bounded model checking: **Agreement**

Init

UponProposalInPropose

UponProposalInPropose

UponProposalInPrevoteOrCommitAndPrevote

UponProposalInPrecommitNoDecision

UponProposalInPrevoteOrCommitAndPrevote

UponProposalInPrecommitNoDecision

decisions = { v0, v1 }

2 correct, 2 faulty:

1 CPU ⇒ 2 min 52 sec

32 CPUs/cloud ⇒ 43 sec

3 correct, 2 faulty

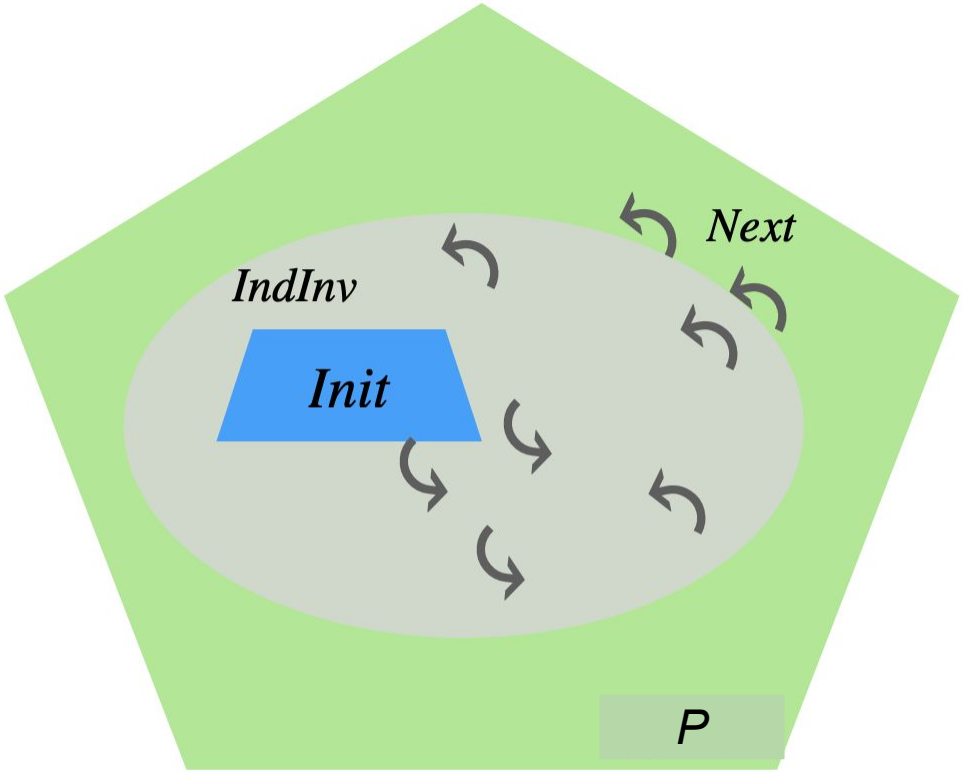
1 CPU ⇒ 6 min 25 sec

32 CPUs/cloud ⇒ 1 min

Inductive invariants

Find a predicate *IndInv* over states:

1. $Init \Rightarrow IndInv$
2. $IndInv \wedge Next \Rightarrow IndInv'$
3. $IndInv \Rightarrow P$



Shallow queries of length 0 and 1 in Apalache!

Randomized symbolic execution

Bounded model checking: **Agreement**

Init

UponProposalInPropose

UponProposalInPropose

UponProposalInPrevoteOrCommitAndPrevote

UponProposalInPrecommitNoDecision

UponProposalInPrevoteOrCommitAndPrevote

UponProposalInPrecommitNoDecision

decisions = { v0, v1 }

2 correct, 2 faulty:

1 CPU ⇒ 2 min 52 sec

32 CPUs/cloud ⇒ 43 sec

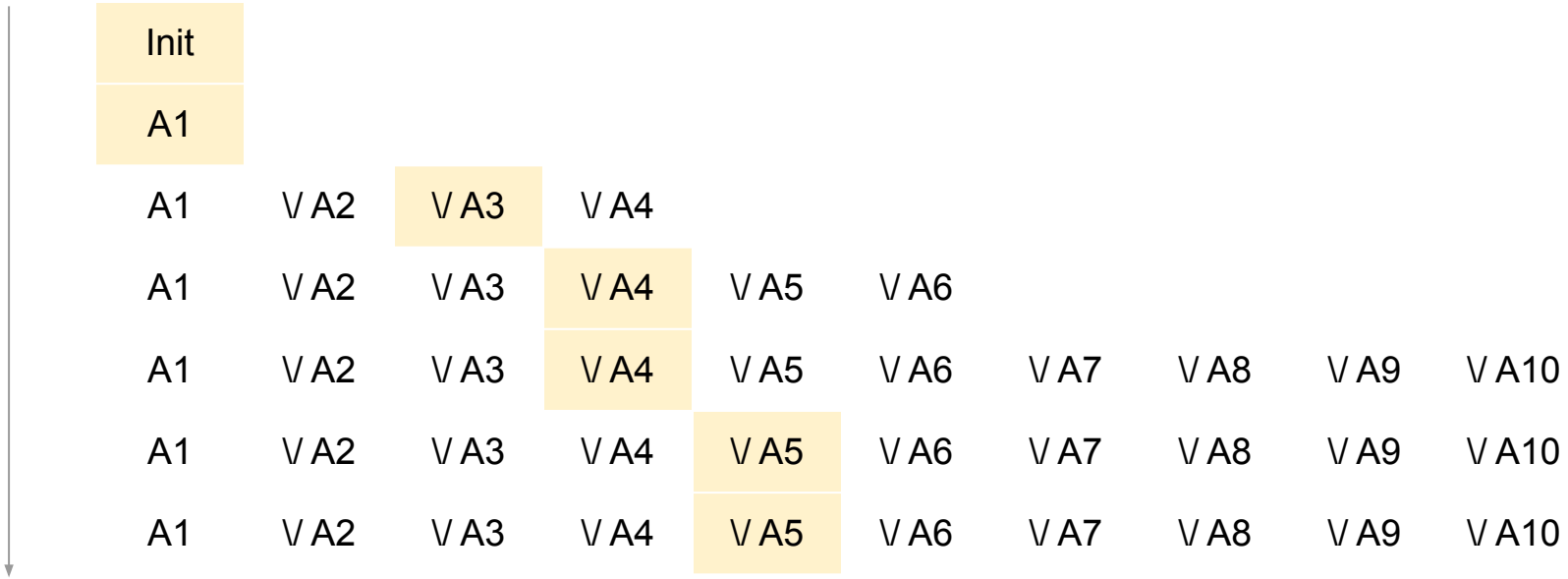
3 correct, 2 faulty

1 CPU ⇒ 6 min 25 sec

32 CPUs/cloud ⇒ 1 min

Why slow down? ⏱️

Randomized symbolic execution



¬Invariant

How to run

```
$ $APALACHE_HOME/bin/apalache-mc simulate --inv=Agreement\
  --cinit=ConstInit MC_n4_f2.tla
```

Is it better?

- Run 20 experiments for Agreement on n = 4, f = 2 with hyperfine:

  31.340 s ± 24.897 s vs **172 s** in non-randomized

- In practice, it finds violations faster

- (No large scale experiments though)

- **What guarantees do we have?**

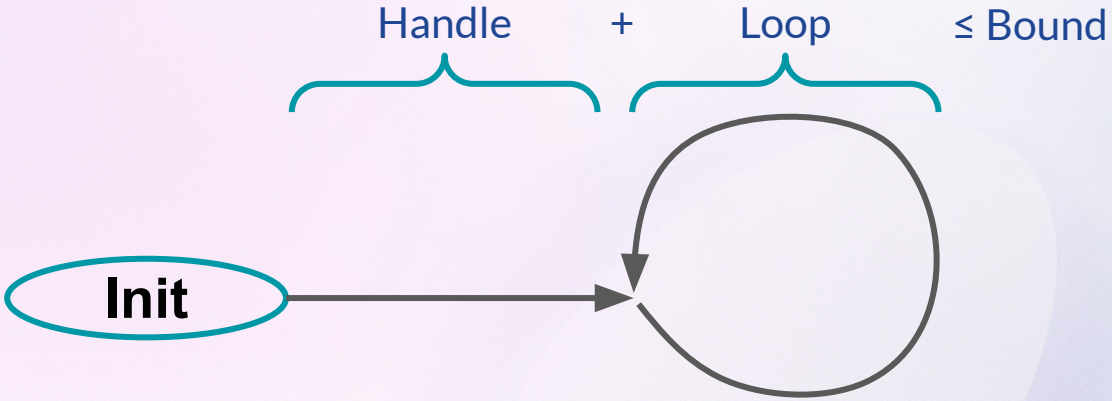
Bounded Model Checking & Liveness

Apalache: Symbolic bounded model checker
Reasons about traces of finite length

"There is no invariant violation in the first 50 steps"

What does this mean for lassos?
Bound on the length of the lasso: Handle + loop

"There is no counterexample lasso of size at most 50"



Biere et al.: Linear Encodings of Bounded LTL Model Checking

Logical Methods in Computer Science
Vol. 2 (5:5) 2006, pp. 1–64
www.lmcs-online.org

Submitted Feb. 16, 2006
Published Nov. 15, 2006

LINEAR ENCODINGS OF BOUNDED LTL MODEL CHECKING

ARMIN BIERE [a], KEIJO HELJANKO [b], TOMMI JUNTTILA [c], TIMO LATVALA [d], AND VIKTOR SCHUPPAN [e]

[a] Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstrasse 69, A-4040 Linz, Austria
e-mail address: biere@jku.at

[b,c] Laboratory for Theoretical Computer Science, Helsinki University of Technology, P.O. Box 5400, FI-02015 TKK, Finland
e-mail address: {Keijo.Heljanko,Tommi.Junttila}@tkk.fi

[d] Department of Computer Science, University of Illinois at Urbana-Champaign, 201 Goodwin Ave., Urbana, IL 61801-2302, USA
e-mail address: tlatvala@uiuc.edu

[e] Computer Systems Institute, ETH Zentrum, CH-8092 Zürich, Switzerland
e-mail address: vschuppan@acm.org

ABSTRACT. We consider the problem of bounded model checking (BMC) for linear temporal logic (LTL). We present several efficient encodings that have size linear in the bound.

| Summary

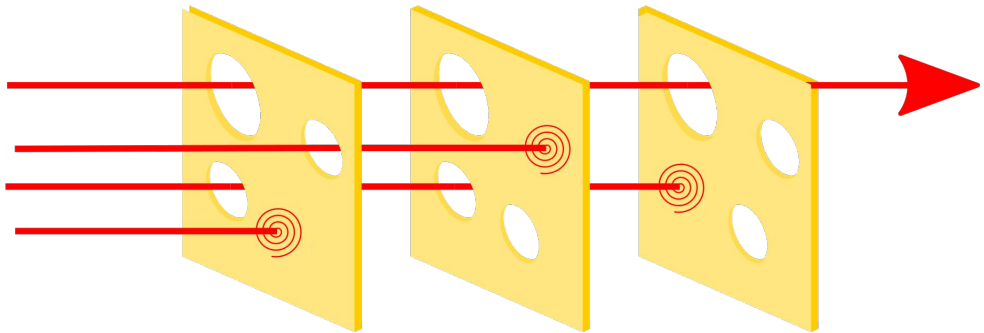
We need multiple tools for one language

Simple tools and complicated tools

This started with TLA⁺ tooling: **TLC**, **TLAPS**, **Apalache**

We've added simpler tools on top of that in Quint

Will you add more?



Source: https://en.wikipedia.org/wiki/Swiss_cheese_model

Where we are

- Finishing the missing features
- Talking to the first users
- Fixing irregularities in Quint
- **Open source**
- Contributions are welcome


Jure Kukovec


Shon Feder


Gabriela Moreira
@bugarela


Igor Konnov
@konnov


Thomas Pani

github.com/informalsystems/quint

build passing | VSCode v0.7.0 | npm v0.13.0

Quint is a modern specification language that is a particularly good fit for distributed systems and blockchain protocols. It combines the robust theoretical basis of the Temporal Logic of Actions (TLA) with state-of-the-art static analysis and development tooling.

If you are impatient, here is a 15 minute intro to Quint at Gateway to Cosmos 2023.

This is how typical Quint code looks:

```
// `validateBalance` should only be called upon g i
pure def validateBalance(ctx: BankCtx, addr: Addr): bo
  ctx.accounts.contains(addr),
  val coins = getAllBalances(ctx, addr)
  coins.keys().forall(denom => coins.get(denom) > 0),
}
```

If you would like to see the same code in TLA+, here is how it looks:

@informalsystems / **quint**

Contributors 14

+ 3 contributors

Languages

- TypeScript 71.8%
- Bluespec 12.5%
- Java 11.9%
- ANTLR 1.1%
- JavaScript 0.9%
- Shell 0.5%
- Other 1.3%

| Revisiting mjrty

Technical Report No. ICSCA-CMP-32
Grant No. MCS-8202943
Contract No. N00014-81-K-0634; NR 049-500

MJRTY - A FAST MAJORITY VOTE ALGORITHM

Robert S. Boyer and J Strother Moore
Institute for Computing Science and Comp
The University of Texas at Austin
Austin, TX 78712

February, 1981

Here is a bloodless way the chairman can simulate the pairing phase. He visits each delegate in turn, keeping in mind a current candidate CAND and a count K, which is initialized to 0. Upon visiting each delegate, the chairman first determines whether K is 0; if it is, the chairman selects the delegate's candidate as the new value of CAND and sets K to 1. Otherwise, the chairman asks the delegate whether his candidate is CAND. If so, then K is incremented by 1. If not, then K is decremented by 1. The chairman then proceeds to the next delegate. When all the delegates have been processed, CAND is in the majority if a majority exists.

let’s look at mjrty.qnt in VSCode...