Quint

Protocols as code in the logic of TLA+

Shon Feder, Igor Konnov, Jure Kukovec, Gabriela Moreira, Thomas Pani

TLA+ Community Meeting

April 22, 2023

informal SYSTEMS

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-ti
`|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 restr ction, w liveness is preserved.

In general the local clock may drift from the global time. (It may progress faster, e.g., one second of lock time might take 1.00 real-time). As a result the local clock and the global clock may be measured in different time units. U sually, the message delay in global clock time units. To estimate the correct local timeout precisely, we would need to estimate the clo delay taking into account the clock drift. For simplicity we ignore this, and directly postulate the message delay assumption in terms of local time.

TLA+

```
\* 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"
```

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 only)
- IBC: 02, 03, 04, 18
- (Partial) specs in security audits
- Namada's Proof-of-Stake

informal SYSTEMS

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

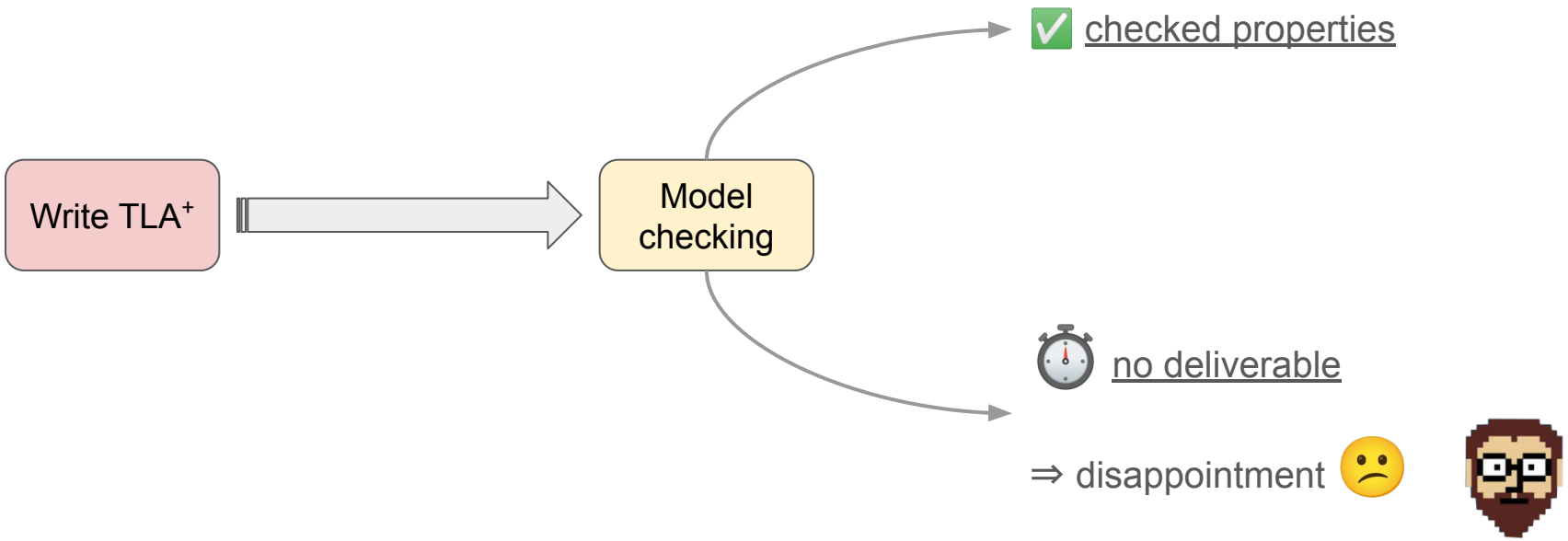
Blockchain engineers and security auditors

- 😕 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

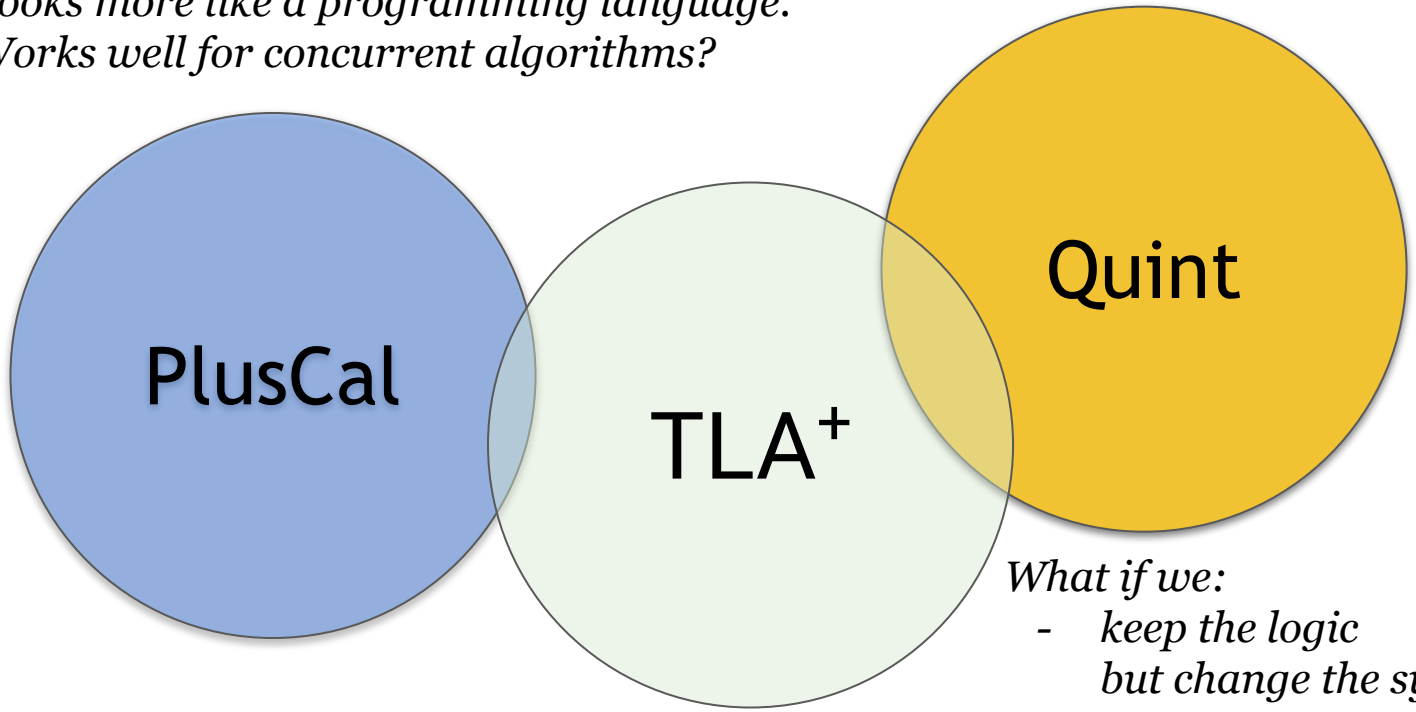
Similar to the experience of:

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

The feedback loop



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



What if we:

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

Universal,
solid logic foundation

Quint syntax

Design principles

Least surprise: copy syntax from mainstream languages

Easy to read: keeps the set of ASCII control characters to minimum

Easy to write and parse: a small set of syntactic rules (250 LOC)

(Mostly) compatible with TLA$^+$:

- folds instead of recursive operators
- simplified instances

Command line-first: IDEs change, CLI tools stay

Uniform operators syntax

Two interchangeable forms

```
union(set1, set2)
set1.union(set2)
// works for built-in operators
// as well as for user-defined operators
```

Binding via lambda syntax

```
set.map(x => 2 * x)
set.filter(x => x % 2 == 0)
set.forall(x => x > 0)
1.to(30).exists(x => x == 22)
```

Few special forms

```
e == f, e != f
p or q, p and q, p implies q, p iff q
i > j, i >= j, i < j, i <= j
if (p) e else f
Set(1, 2, 3)
Map("Alice" -> 3, "Bob" -> 5)
{ error: msg, state: state }
("TLA+", 2023)
```

Layered language

Pure and stateful definitions

```
pure val MAX_UINT = 2^256 - 1


pure def sumOverBalances(balances) = {
    balances.keys().fold(0,
      (sum, a) => sum + balances.get(a))
}
var state: Erc20State
val totalSupplyInv = isTotalSupplyCorrect(state)
```

Actions

```
action submit(tx: Transaction): bool = all {
    mempool' = mempool.union(Set(tx)),
    erc20State' = erc20State,
    lastTx' = tx,
}
```

Temporal formulas

```
temporal totalSupplyNeverChanges =
  always(totalSupplyInv)
```

Isolating non-determinism

Data non-determinism

∃ sender ∈ ADDR: …

Control non-determinism

∨ A1
∨ A2
∨ A3

```
action step = {
  nondet sender = oneOf(ADDR)
  nondet amount = oneOf(AMOUNTS)
  any {
    nondet toAddr = oneOf(ADDR)
    fromResult(erc20State.transfer(sender, toAddr, amount)),
    nondet spender = oneOf(ADDR)
    fromResult(erc20State.approve(sender, spender, amount)),
    nondet fromAddr = oneOf(ADDR)
    nondet toAddr = oneOf(ADDR)
    fromResult(erc20State.transferFrom(sender, fromAddr,
                                       toAddr, amount)),
  }
}
```

Types are built-in

```
// type aliases

type Address = str

type Uint = int

// variables must have a type annotation

var mempool: Set[Transaction]

// operators may have a type annotation

pure def isUint(i: int): bool =
  (0 <= i and i <= MAX_UINT)
```

```
// a record type
type Erc20State = {
  // a map of addresses to amounts
  balanceOf: Address -> Uint,
  // the sum of all balances
  totalSupply: Uint,
  // a map of pairs to amounts
  allowance: (Address, Address) -> Uint,
  // the address of the contract creator
  owner: Address,
}
```

Folds instead of recursive operators

Iteration over lists

- always terminates
- len(..) iterations

```
pure def simpleHash(word) =
  word.foldl(0, (i, j) => i + j) % BASE
```

Iteration over sets

- iterates in some order
- always terminates
- size(..) iterations

```
pure def sumOverBalances(balances) = {
    keys(balances)
      .fold(0,
            (sum, a) => sum + balances.get(a))
}
```

General recursion

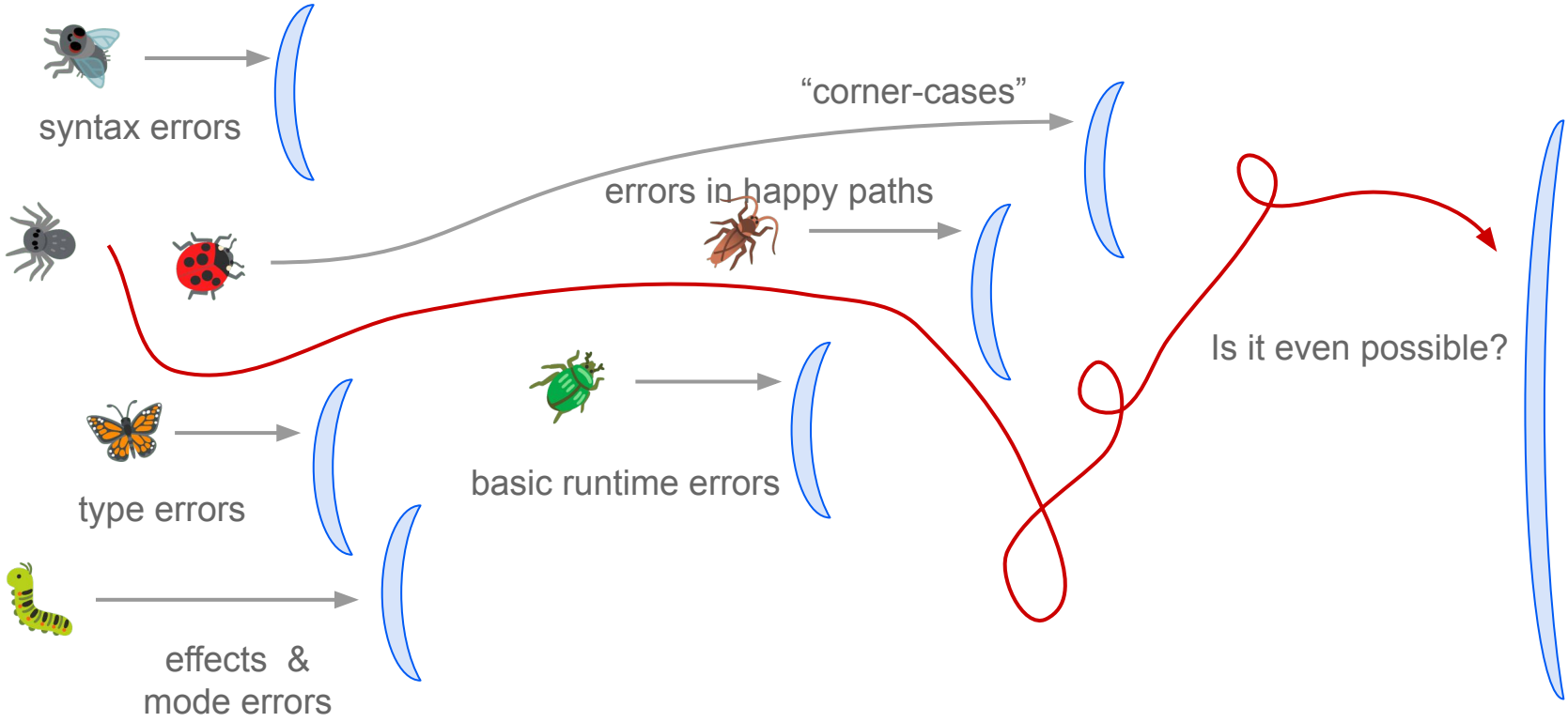
- have a practical example?

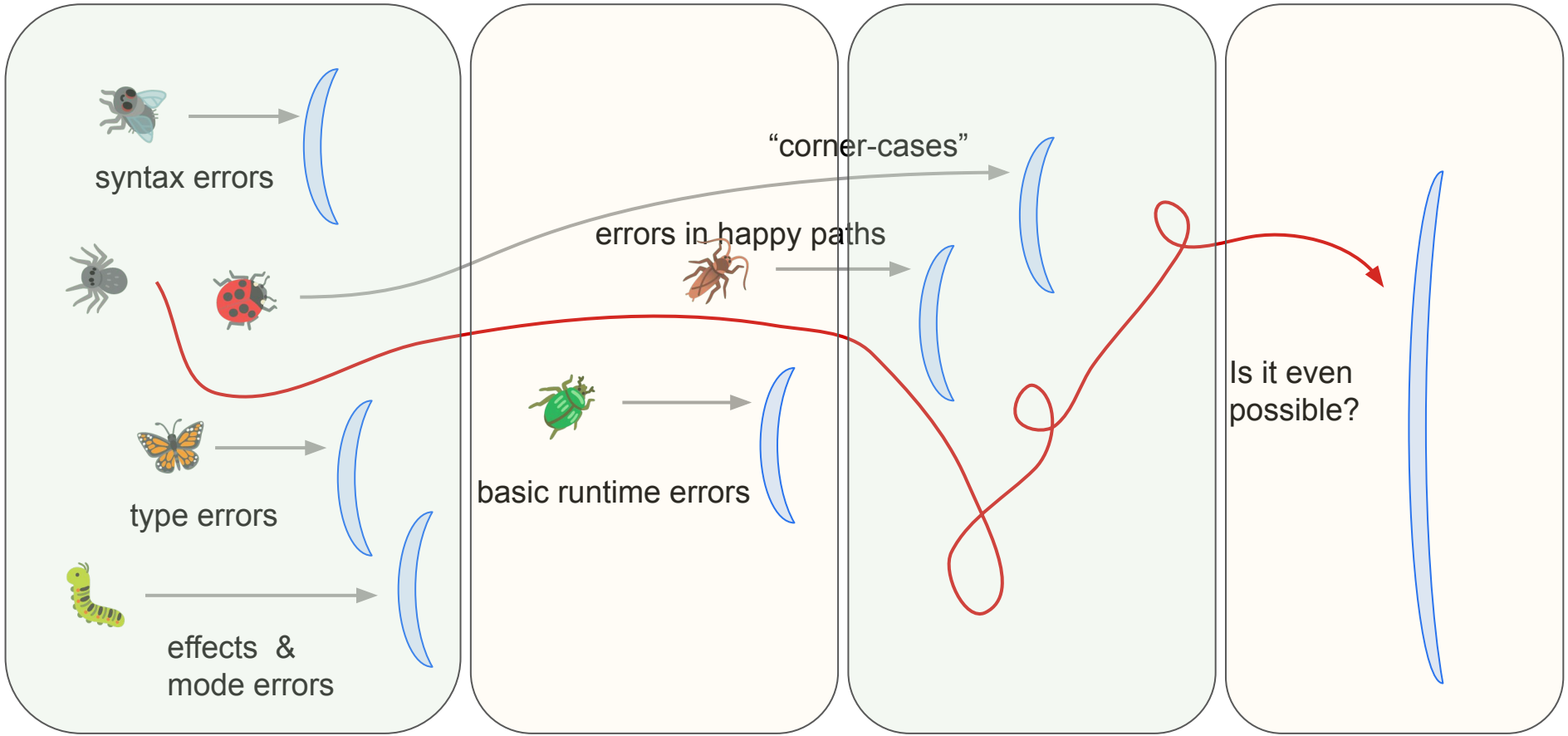
Runs (new)

- sequence of actions
- A.then(B) is A \cdot B of TLA$^+$
- unit tests and property-based
- evidence of liveness

```
run transferFromWhileApproveInFlightTest = {
  all {
    erc20State' = newErc20("alice", 91),
    mempool' = Set(), lastTx' = NoneTx,
  } // alice sets a high approval for bob
  .then(submit(ApproveTx("alice", "bob", 92)))
  // bob immediately initiates his transaction
  then(submit(TransferFromTx("bob", "alice", "eve", 54)))
  // alice changes her mind and lowers her approval to bob
  .then(submit(ApproveTx("alice", "bob", 9)))
  // but the previous approval arrives at the ledger
  .then(commit(ApproveTx("alice", "bob", 92)))
  // bob transfers more than alice wanted to
  .then(commit(TransferFromTx("bob", "alice", "eve", 54)))
}
```

Quint tools


syntax errors
“corner-cases”
errors in happy paths
Is it even possible?
basic runtime errors
type errors
effects &
mode errors



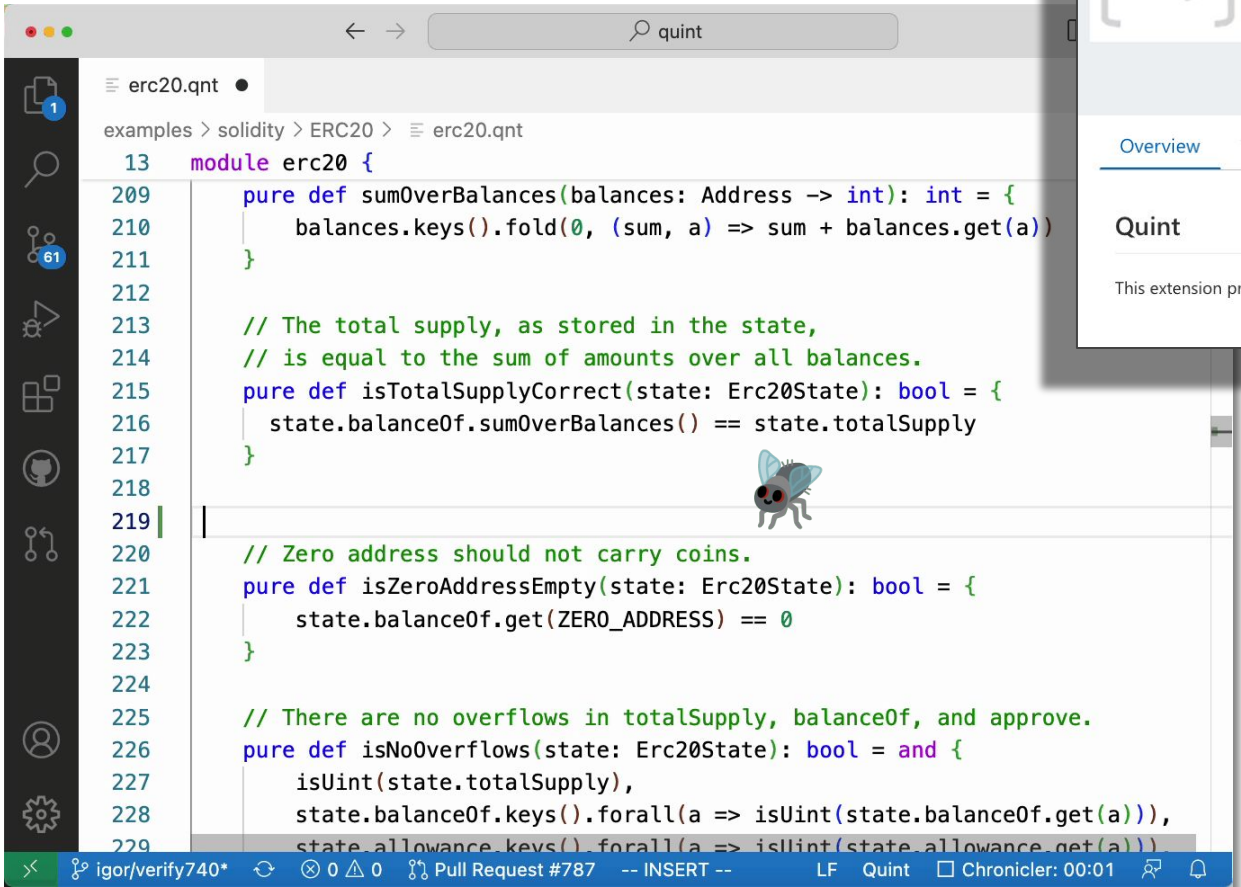
VSCode: syntax, types, effects, modes

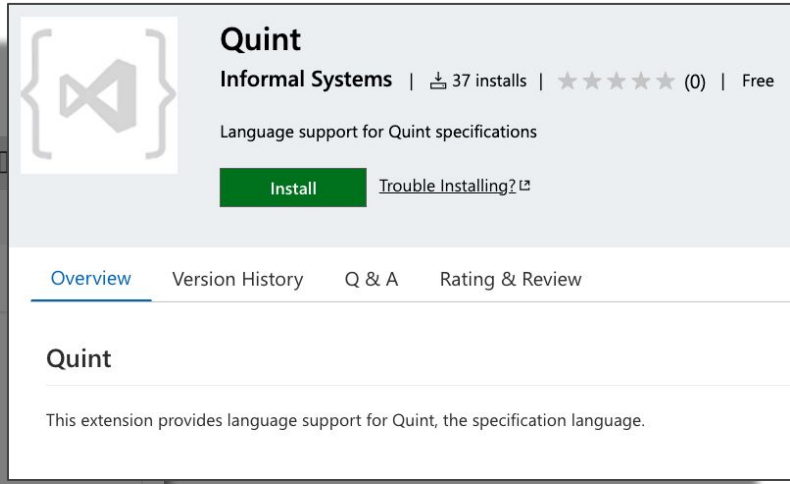
REPL: interactive debugging

Unit tests, randomized tests

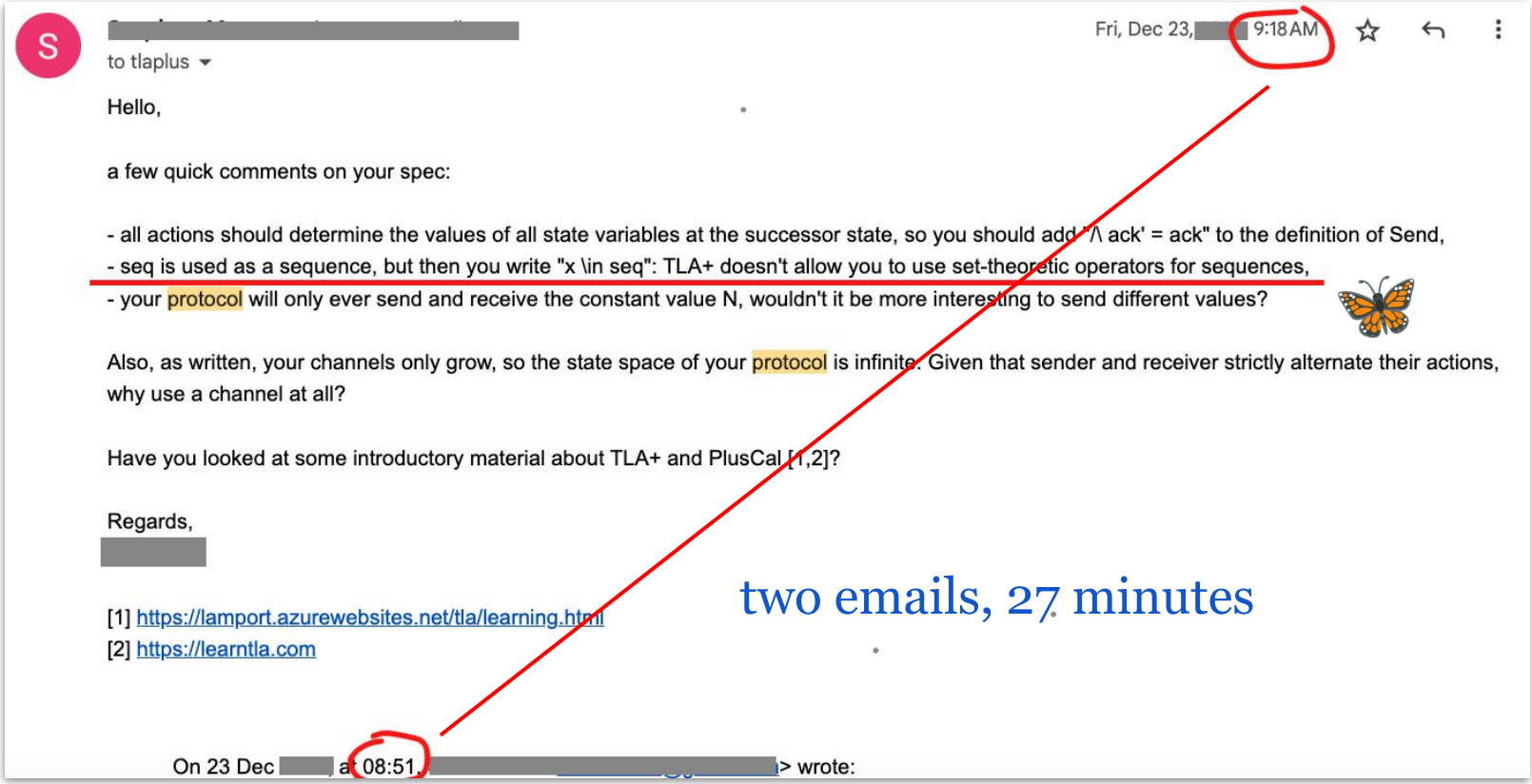
Model checking

Parser and VSCode plugin





Type errors



S

[redacted]

Fri, Dec 23, [redacted] 9:18 AM

to tlaplus

Hello,

a few quick comments on your spec:

- all actions should determine the values of all state variables at the successor state, so you should add "/\ ack' = ack" to the definition of Send,
- seq is used as a sequence, but then you write "x \in seq": TLA+ doesn't allow you to use set-theoretic operators for sequences,
- your protocol will only ever send and receive the constant value N, wouldn't it be more interesting to send different values?

Also, as written, your channels only grow, so the state space of your protocol is infinite. Given that sender and receiver strictly alternate their actions, why use a channel at all?

Have you looked at some introductory material about TLA+ and PlusCal [1,2]?

Regards,

[redacted]

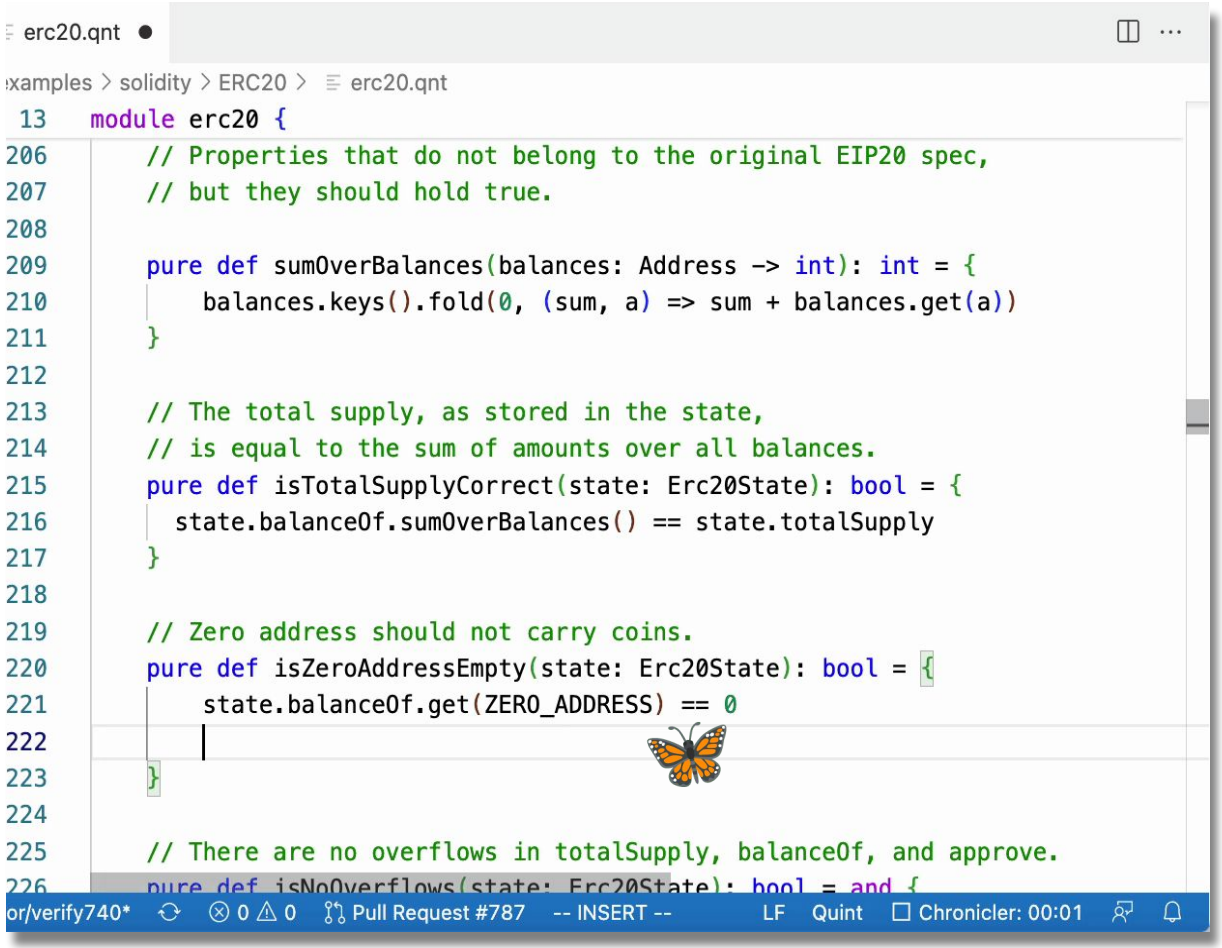
[1] https://lamport.azurewebsites.net/tla/learning.html
[2] https://learntla.com

On 23 Dec [redacted] at 08:51, [redacted] wrote:

two emails, 27 minutes

Type checker

Type feedback in 1s



Type checker mirroring Apalache

- Damas & Milner type inference + row types
- No inductive types, no subtyping, no ad-hoc polymorphism

  records, tuples, maps, and lists have distinct operators

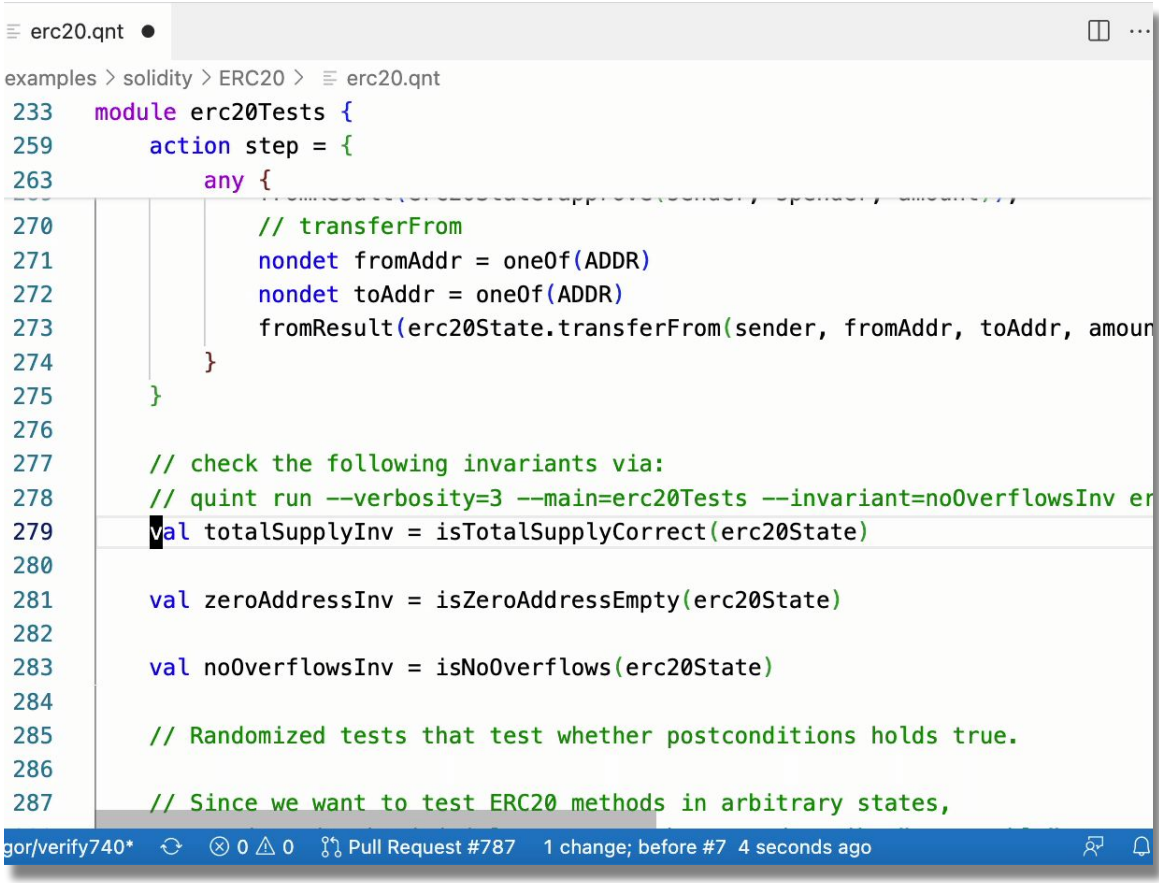
Int	Bool	Str
UNINTERPRETED	Set[a]	List[a]
a -> b	(a, b, c)	
{ f1: a, f2: b, f3: c }	Tag1(a) | Tag2(b) | Tag3(c)	(a, b, c) => d

work-in-progress 🚧

Mode checker 🐛

- States and actions not allowed in pure def
- Actions not allowed in def and val
- Ever tried to write in TLA+?

  { y' = x + 1: x \in S }



Bogus unchanged and missing x’ = e 🐛

lemmy commented on Nov 3, 2021 • edited ▾ Member ⋯

With lemmy/raft.tla@ `17a4f67` , smoke testing reliably finds the bogus UNCHANGED! On average, simulation takes 10k states, 26 traces, less than a second, and traces of 21 steps to find the UNCHANGED (dataset ~300 runs). The important part was to slightly increase the number of initial states to >=512. It appears as if a subset of initial states make it either impossible or extremely unlikely to find the bogus UNCHANGED. In other words, smoke testing works surprisingly well even for large specs/state spaces. :-)

☺

This is what the effects checker is looking for
no model checker is needed

Effects checker

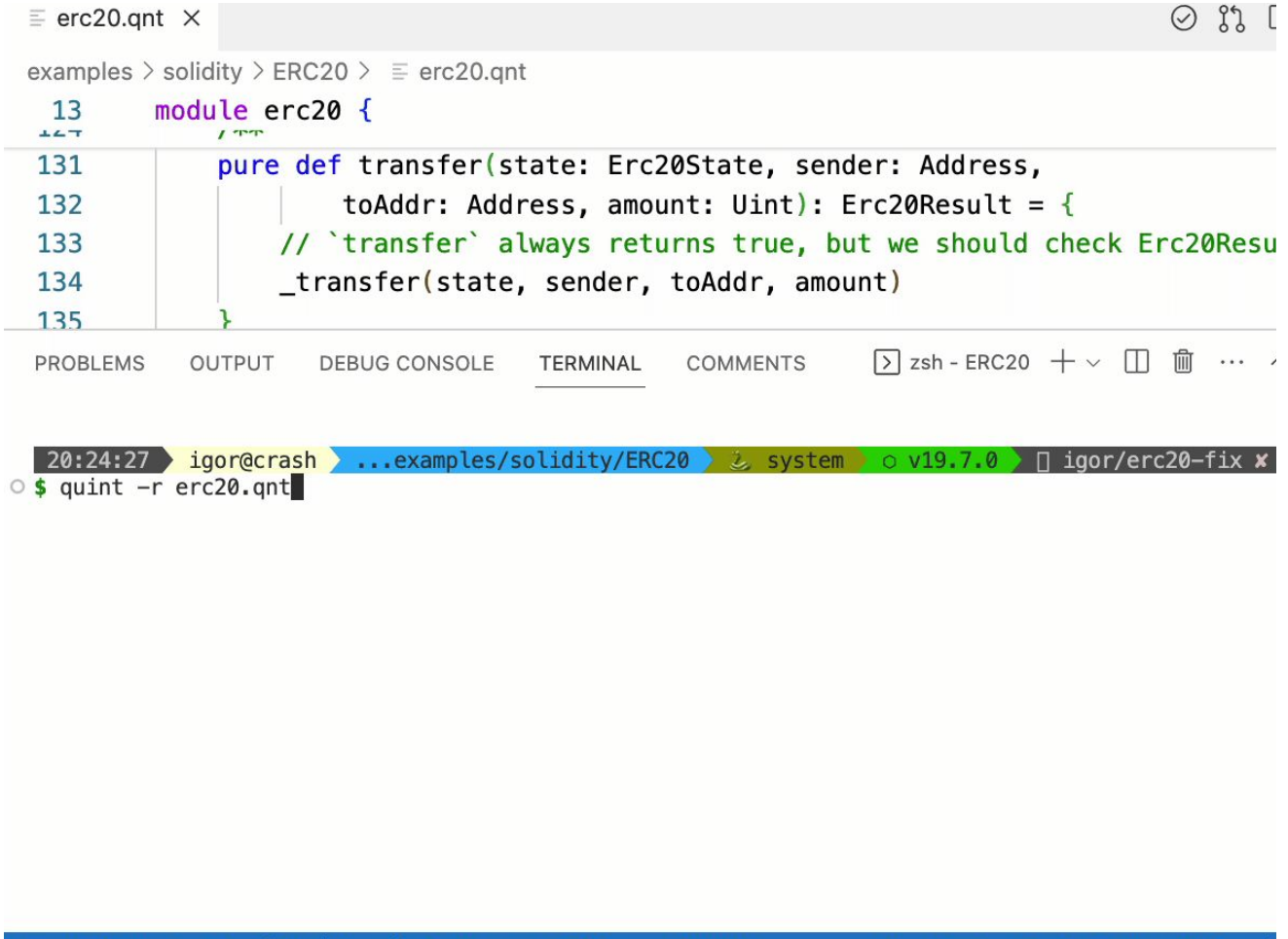
erc20.qnt ●

examples > solidity > ERC20 > erc20.qnt

```
351     module mempool {
412         action submit(tx: Transaction): bool = all {
416         }
417
418         // an auxilliary action that assigns variables from a method execution res
419         action fromResult(tx: Transaction, r: Erc20Result): bool = all {
420             val status = if (r.returnedTrue and r.error != "") r.error else "succe
421             lastTx' = tx.with("status", status),
422             erc20State' = r.state,
423         }
424
425         // commit a transaction from the memory pool
426         action commit(tx: Transaction): bool = all {
427             mempool' = mempool.exclude(Set(tx)),
428             any {
429                 all {
430                     tx.kind == "transfer",
431                     fromResult(tx, transfer(erc20State, tx.sender, tx.toAddr, tx.a
432                 },
433                 all {
434                     tx.kind == "approve",
435                     fromResult(tx, approve(erc20State, tx.sender, tx.spender, tx.a
```

REPL

- Interactive learning
- Step-by-step debugging



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)
    }
  }
```

Finding counterexamples

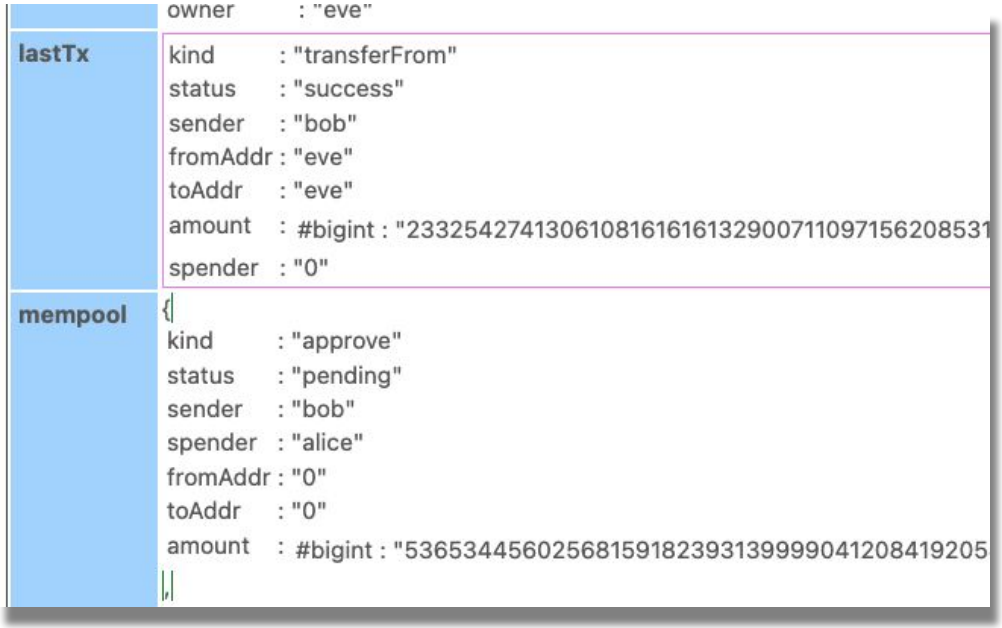
```
[Frame 1]
 q::stepAndInvariant() ⇒ true
 └─ q::step() ⇒ true
    └─ step() ⇒ true
       └─ submit({ kind: "approve", status: "pending", sender: "0"
, spender: "alice", fromAddr: "0", toAddr: "0", amount: 9499149855
9526274836123202226267546379050703071640247512155136690770454366517 }) ⇒ true
          └─ ApproveTx("0", "alice", 94991498559526274836123202226
26754637905070307164024751215513669077045436651 7) ⇒ { kind: "appr
ove", status: "pending", sender: "0", spender: "alice", fromAddr:
"0", toAddr: "0", amount: 9499149855952627483612320222626754637905
07030716402475121551366907704543665 17 }
```

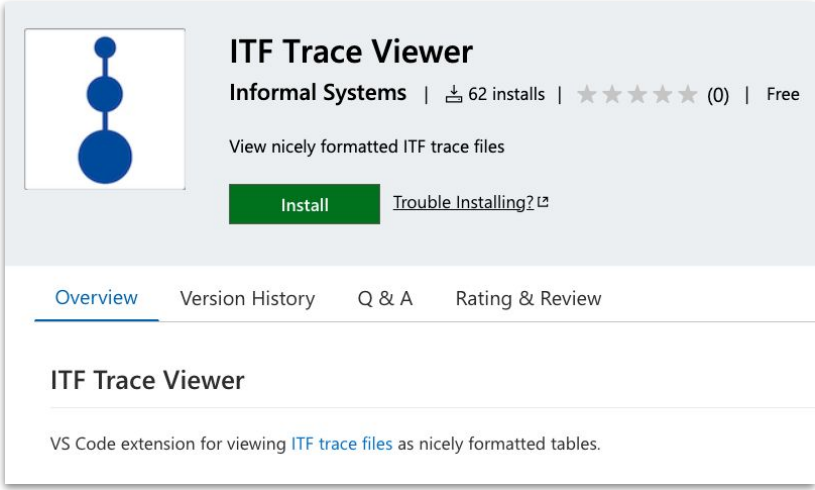
```
[State 16]
 erc20State: { balanceOf: Map("0" → 0, "alice" → 0, "bob" → 0,
"eve" → 791721037759798107063077463563162357503204582894890020713
73698387586366560739), totalSupply: 791721037759798107063077463563
16235750320458289489002071373698387586366560739, allowance: Map(("
0", "0") → 0, ("0", "alice") → 0, ("0", "bob") → 0, ("0", "eve"
```

```
$ quint run \
  --invariant=noTransferFromWhileApproveInFlight \
  --verbosity=3 \
  erc20.qnt
```

$ ❌ in **10.853 s** ± 8.359 s

Trace viewer





by Hernan Vanzetto @ Informal

Testing framework

```
$ quint test --main=erc20Tests erc20.qnt

  erc20Tests
    ok transferTest passed 10000 test(s)

  1 passing (895ms)
```

- Unit tests and PBT tests
- The standard *unit UX
- Easy to use with continuous integration

```
$ quint test --main=mempool erc20.qnt

  mempool
    ok transferFromWhileApproveInFlightTest passed 1 test(s)

  1 passing (133ms)
```

Model checker – Apalache

```
$ quint verify \
  --invariant=noTransferFromWhileApproveInFlight \
  erc20.qnt
```

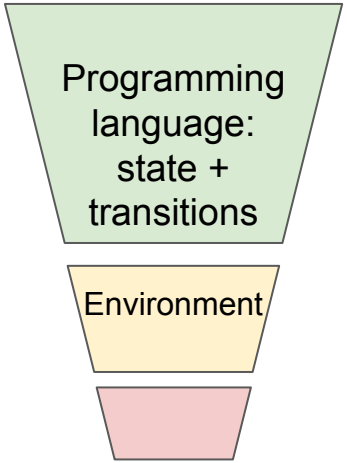
...next week(s) 😀

- Finishing the integration
- Quint IR and Apalache IR are quite similar
- Subtle differences: Quint is lambda-centric, indices start with 0, etc.

Quint workflow

Engineers

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

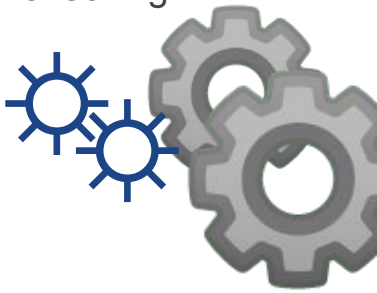


Invariants
temporal properties

Auditors

Protocol designers

- Symbolic simulation
- Model checking



Where we are

- Finishing the integration between Apalache and Quint
- Talking to the first users
- Fixing irregularities in Quint
- Open source
- Contributions are welcome

github.com/informalsystems/quint

Quint

build passing VSCode v0.2.0 npm v0.7.0

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.

This is how typical Quint code looks:

```
// `validateBalance` should only be called upon gen
pure def validateBalance(ctx: BankCtx, addr: Addr):
  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

Used by 1

@informalsystems / quint

Contributors 9

Languages

- TypeScript 93.7%
- ANTLR 1.7%
- JavaScript 1.0%
- Python 1.0%
- Shell 0.9%
- Vim Script 0.4%
- Other 1.3%

As one engineer said

TLA+: OK, it works for consensus

But it will not work for my problem

Quint: OK, it works for consensus and smart contracts

But it will not work for my problem