Tutorial on the TLA$^+$ language and its support tools: Apalache

Igor Konnov

TLA$^+$ Community Meeting 2023

April 22, 2023

informal SYSTEMS

Apalache

Symbolic model checker for TLA$^+$

`apalache.informal.systems`

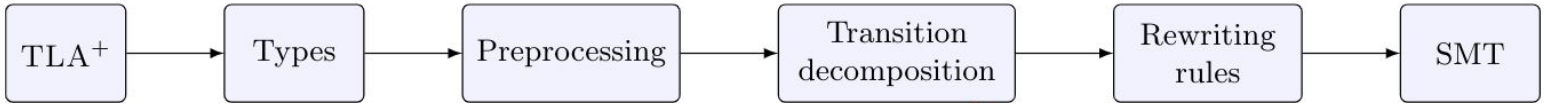
bounded number of integer constants is OK

Symbolic model checker that works under the assumptions of TLC:

- **Fixed and finite constants** (parameters)
- **Finite sets, function domains** and **co-domains**
- TLC's restrictions on formula structure
- Bounded model checking to check safety
- **As few language restrictions as possible**

Technically,

- Quantifier-free formulas in SMT: `QF_UFNIA`
- Unfolding quantified expressions: $\forall x \in S : \; P$ as $\bigwedge_{c \in S} P[c/x]$



informal SYSTEMS

Running example
ERC20 Tokens

github.com/informalsystems/tla-apalache-workshop/tree/main/examples/erc20

ERC20

ethereum.org/en/developers/docs/standards/tokens/erc-20/

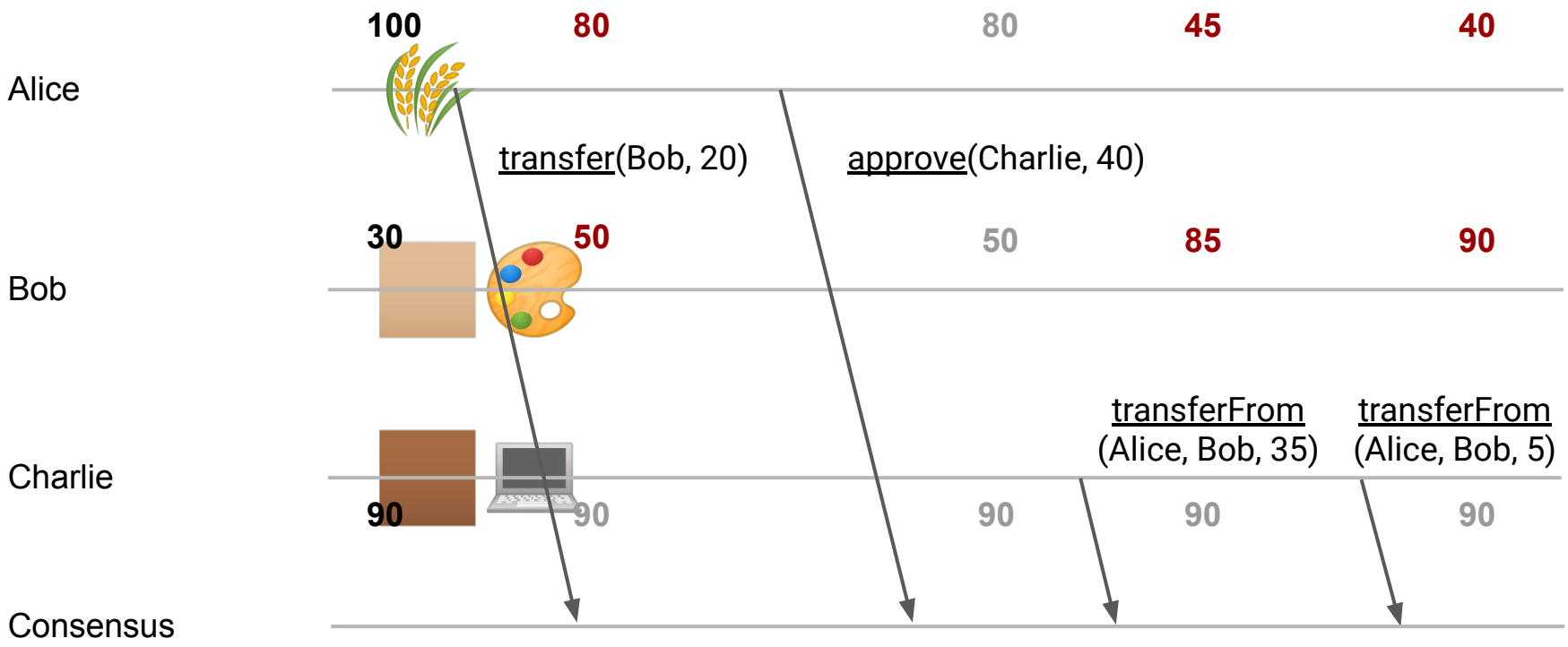
- A foundational Solidity API for virtually all tokens running on Ethereum

- Incorrect use leads to security issues

- Similar interfaces in Cosmos:
  - The bank module in Cosmos SDK – in Golang
  - CW20: Cosmwasm contracts – in Rust
  - ICS20: token transfer between Cosmos blockchains – crossplatform

Can we do anything useful about it with TLA$^+$?

ERC20 as diagrams (happy paths)



Step 1

Specifying the contract
without specifying a state machine

informal SYSTEMS

Preambule

```
---------- MODULE erc20 ----------
EXTENDS Integers, Apalache, erc20_typedefs
CONSTANTS
  \* @type: Set(ADDR);
  AllAddresses,
  \* @type: Int;
  MAX_UINT
```

```
\* The special address 0x0 in EVM
ZERO_ADDRESS ≜ "0_OF_ADDR"
ASSUME(ZERO_ADDRESS ∈ AllAddresses)
\* The predicate that we use
\* to check for overflows.
\* In EVM, MAX_UINT = 2^256 - 1
isUint(i) ≜ 0 ≤ i ∧ i ≤ MAX_UINT
```

Let's talk about types later

Contract constructor

```
\* @type: (ADDR, Int) => $state;
newErc20(sender, initialSupply) ≜ [
  balanceOf ↦ [ a ∈ AllAddresses ↦
    IF a ≠ sender THEN 0 ELSE initialSupply
  ],
  totalSupply ↦ initialSupply,
  allowance ↦ [ a, b ∈ AllAddresses ↦ 0],
  owner ↦ sender
]
```

Specifying transfer

```
\* @type: ($state, ADDR, ADDR, Int) => $result;
transfer(state, fromAddr, toAddr, amount) ≜
  LET fromBalance ≜ state.balanceOf[fromAddr] IN
  LET err ≜
    CASE ¬(fromAddr ≠ ZERO_ADDRESS) →
        "ERC20: transfer from the zero address"
      □ ¬(toAddr ≠ ZERO_ADDRESS) →
        "ERC20: transfer to the zero address"
      □ ¬(fromBalance ≥ amount) →
        "ERC20: transfer amount exceeds balance"
      □ OTHER → ""
  IN
  ...
```

```
...
IF err ≠ "" THEN Error(err)
ELSE
  LET newBalances ≜
      IF fromAddr = toAddr
      THEN state.balanceOf
      ELSE [ state.balanceOf EXCEPT
          ![fromAddr] = fromBalance - amount,
          ![toAddr] = @ + amount ]
  IN
  Ok(TRUE,
      [ state EXCEPT !.balanceOf = newBalances ])
```

overflows?

Step 2

Simple state machine to model check

informal SYSTEMS

State machine to model check

```
---------- MODULE erc20_tests ----------
AllAddresses ≜ { "0_OF_ADDR", "alice_OF_ADDR",
                 "bob_OF_ADDR", "eve_OF_ADDR" }
MAX_UINT ≜ 2^256 - 1
AMOUNTS ≜ 0..MAX_UINT
INSTANCE erc20
VARIABLE \* @type: $state;
  state
Init ≜
  ∃ sender ∈ AllAddresses \ { ZERO_ADDRESS }:
   ∃ initialSupply ∈ AMOUNTS:
    state = newErc20(sender, initialSupply)
```

```
\* invoke only 'transfer' with various inputs
Next1 ≜
  ∃ sender, toAddr ∈ AllAddresses:
   ∃ amount ∈ AMOUNTS:
    fromResult(transfer(state,
      sender, toAddr, amount))

\* @type: $result => Bool;
fromResult(result) ≜
  ∧ VariantTag(result) = "Ok"
  ∧ VariantGetUnsafe("Ok", result).returnedTrue
  ∧ state' = VariantGetUnsafe("Ok", result).state
```

Running Apalache

Check for 10 steps:

zeroAddressInv ≜

state.balanceOf[ZERO_ADDRESS] = 0

Bounded model checking

Input: Init, Next and Inv

0. $Init \wedge \neg Inv$
1. $(Init \cdot Next) \wedge \neg Inv'$
2. $(Init \cdot Next \cdot Next) \wedge \neg Inv'$

…

k. $(Init \cdot Next \cdot \ldots \cdot Next) \wedge \neg Inv'$

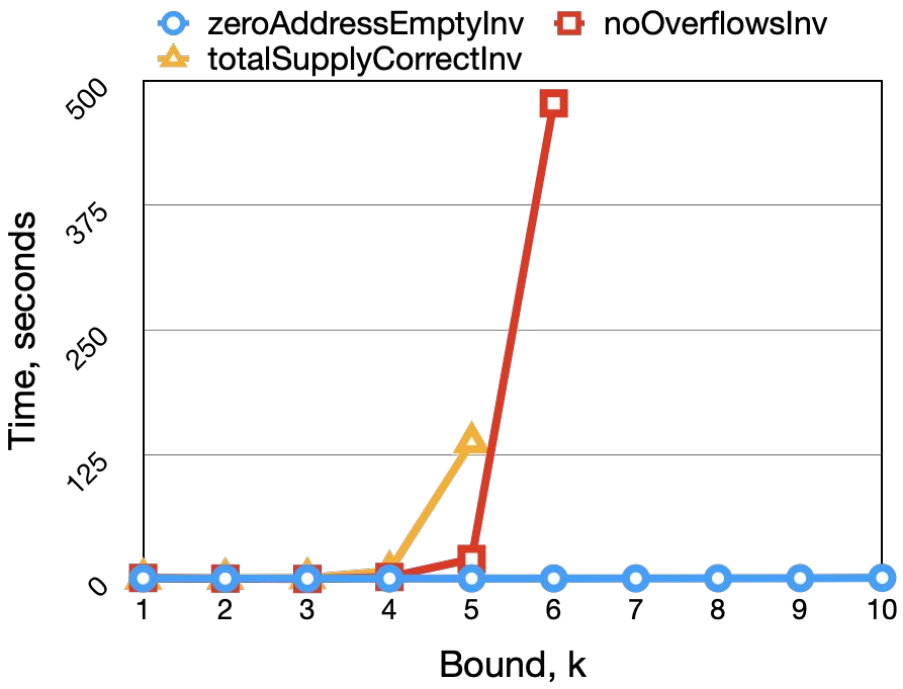
k is the bound

Backend:

SMT
(Microsoft z3)

Checking other invariants

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




```
totalSupplyInv ≜ state.totalSupply =
  sumOverBalances(state.balanceOf)
```

```
noOverflowInv ≜
  ∧ isUint(state.totalSupply)
  ∧ ∀ a ∈ DOMAIN state.balanceOf: isUint(state.balanceOf[a])
  ∧ ∀ p ∈ DOMAIN state.allowance: isUint(state.allowance[p])
```

Step 3

Understanding the underlying techniques

informal SYSTEMS

Translation to SMT

Mimic the semantics implemented by TLC (explicit model checker)

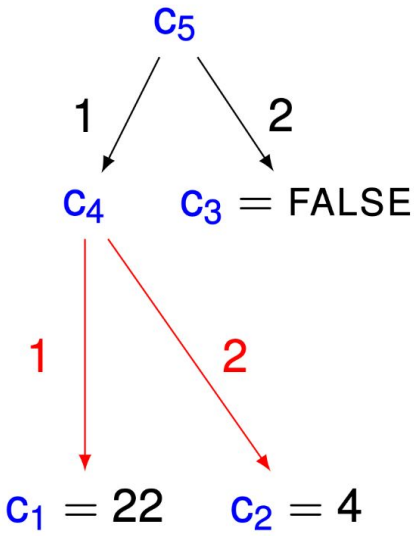
Compute layout of data structures, constrain contents with SMT

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`

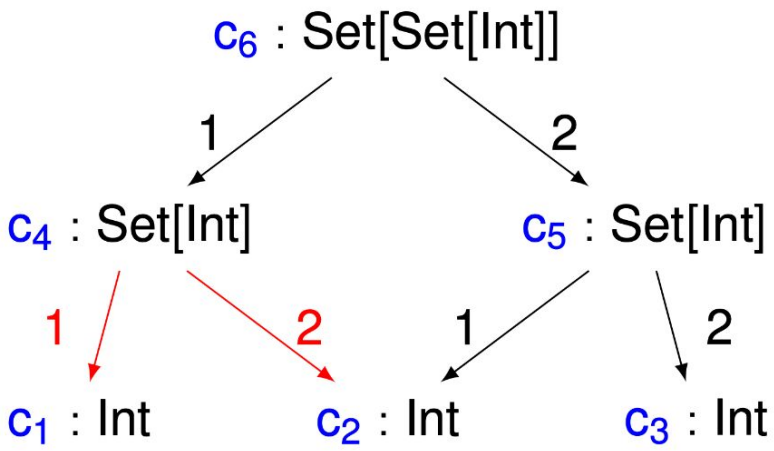
Boolean `sort Bool`

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 $\{\{1\},\{2\}\}$:

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

[OOPSLA'19]

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 encoding

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

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

  Symbolic Model Checking for TLA+ Made Faster [TACAS'23]

  (See the talk by Rodrigo on Thursday!)

- Working faster on classical fault-tolerant algorithms

Step 4

Inductive reasoning

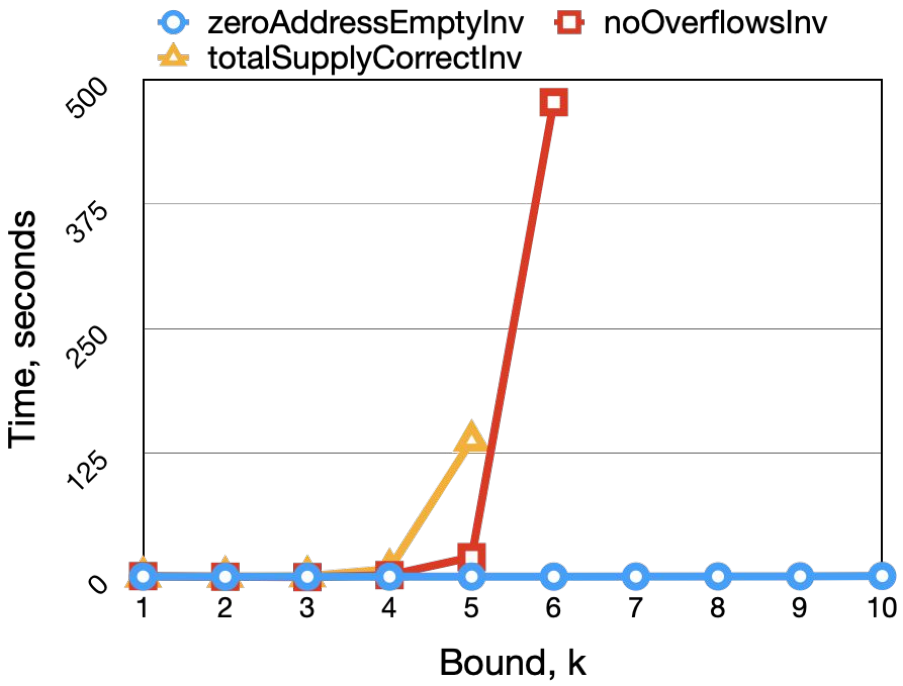
informal SYSTEMS

What value of *k* is convincing?

- Improve the model checker,

  get stuck at larger *k*

- **Alternative**: inductive reasoning



Checking invariants in one step

```
initArbitrary ≜
  ∃ owner ∈ AllAddresses \ { ZERO_ADDRESS }:
   ∃ balances ∈ [ AllAddresses → Int ]:
    ∃ allowances ∈
     [ AllAddresses × AllAddresses → Int ]:
     ∧ state = [ balanceOf ↦ balances,
        totalSupply ↦ sumOverBalances(balances),
        allowance ↦ allowances,
        owner ↦ owner ]
     ∧ isValid

isValid ≜
 totalSupplyInv ∧ zeroAddressInv ∧ noOverflowInv
```

```
$ apalache-mc check \
    --init=initArbitrary \
    --next=Next1 --inv=isValid \
    --length=1 erc20_tests.tla

$ ✅ in 2 seconds

$ apalache-mc check \
    --init=Init \
    --next=Next1 --inv=isValid \
    --length=0 erc20_tests.tla

$ ✅ in 2 seconds
```

Checking the postcondition

```
transferPrePost ≜
  ∃ sender, toAddr ∈ AllAddresses:
   LET ob ≜ state.balanceOf IN
   LET nb ≜ state'.balanceOf IN
    ∧ ∨ sender = toAddr ∧ nb = ob
      ∨ ∧ sender ≠ toAddr
        ∧ nb[sender] ≤ ob[sender]
        ∧ nb[toAddr] ≥ ob[toAddr]
        ∧ nb[toAddr] - ob[toAddr] = ob[sender] - nb[sender]
        ∧ ∀ a ∈ AllAddresses:
            a ∉ { sender, toAddr } ⇒ nb[a] = ob[a]
    ∧ state'.allowance = state.allowance
    ∧ state'.owner = state.owner
    ∧ state'.totalSupply = state.totalSupply
```

```
$ apalache-mc check \
    --init=initArbitrary \
    --next=Next1 \
    --inv=transferPrePost \
    --length=2 erc20_tests.tla
```

$ ✅ in 2 seconds

Step 5

Specifying approve and transferFrom

informal SYSTEMS

Specifying approve

```
\* @type: ($state, ADDR, ADDR, Int) => $result;
approve(state, sender, spender, amount) ≜
  LET err ≜
    CASE ¬(sender ≠ ZERO_ADDRESS) →
        "ERC20: transfer from the zero address"
      □ ¬(spender ≠ ZERO_ADDRESS) →
        "ERC20: transfer to the zero address"
      □ OTHER → ""
  IN
  ...
```

```
...
IF err ≠ ""
THEN Error(err)
ELSE
  \* sender == spender seems to be allowed
  Ok(TRUE,
      [ state EXCEPT
          !.allowance[sender, spender] = amount ])
```

Specifying transferFrom

```
\* @type: ($state, ADDR, ADDR, ADDR, Int) => $result;
transferFrom(state, sender, fromAddr, toAddr, amount) ≜
  LET currentAllowance ≜
    state.allowance[fromAddr, sender] IN
  LET err ≜
    CASE ¬(currentAllowance ≥ amount) →
        "ERC20: insufficient allowance"
      □ ¬(fromAddr ≠ ZERO_ADDRESS) →
        "ERC20: approve from the zero address"
      □ ¬(toAddr ≠ ZERO_ADDRESS) →
        "ERC20: approve to the zero address"
      □ OTHER → ""
  IN
  ...
```

Since `transfer` is not an action, composition is easy

```
...
LET updatedState ≜
  IF currentAllowance = MAX_UINT
  THEN state
  ELSE [ state EXCEPT
    !.allowance[fromAddr, sender] = @ - amount
  ]
IN
IF err ≠ ""
THEN Error(err)
ELSE transfer(updatedState,
              fromAddr, toAddr, amount)
```

Checking the inductive invariant

```
$ apalache-mc check --init=initArbitrary --next=Next --inv=isValid \
    --length=1 erc20_tests.tla

$ ✅ in 3 seconds

$ apalache-mc check --init=Init --next=Next --inv=isValid \
    --length=0 erc20_tests.tla

$ ✅ in 2 seconds
```

Step 7

Type definitions

informal SYSTEMS

Type definitions

```
---------- MODULE erc20_typedefs ------------------
EXTENDS Variants
(* Type definitions for ERC20.
   ADDR is an uninterpreted type representing
   an account address.
 // A state of an ERC20 contract/token
 @typeAlias: state = {
   balanceOf: ADDR -> Int,
   totalSupply: Int,
   allowance: <<ADDR, ADDR>> -> Int,
   owner: ADDR
 };
 *)
```

```
(* The result of applying an ERC20 method
   @typeAlias: result =
      Error(Str)
    | Ok({ returnedTrue: Bool, state: $state }); *)
erc20_typedefs ≜ TRUE
\* A convenience operator for constructing an Error
\* @type: Str => $result;
Error(msg) ≜ Variant("Error", msg)
\* A convenience operator for constructing an Ok result
\* @type: (Bool, $state) => $result;
Ok(returnedTrue, state) ≜
   Variant("Ok",
    [ returnedTrue ↦ returnedTrue, state ↦ state ])
```

Type checker Snowcat

- Damas & Milner type inference + row types (no inductive types)
- Resolving type imprecision between function-like types
- May require type annotations for records, tuples, functions, and sequences

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

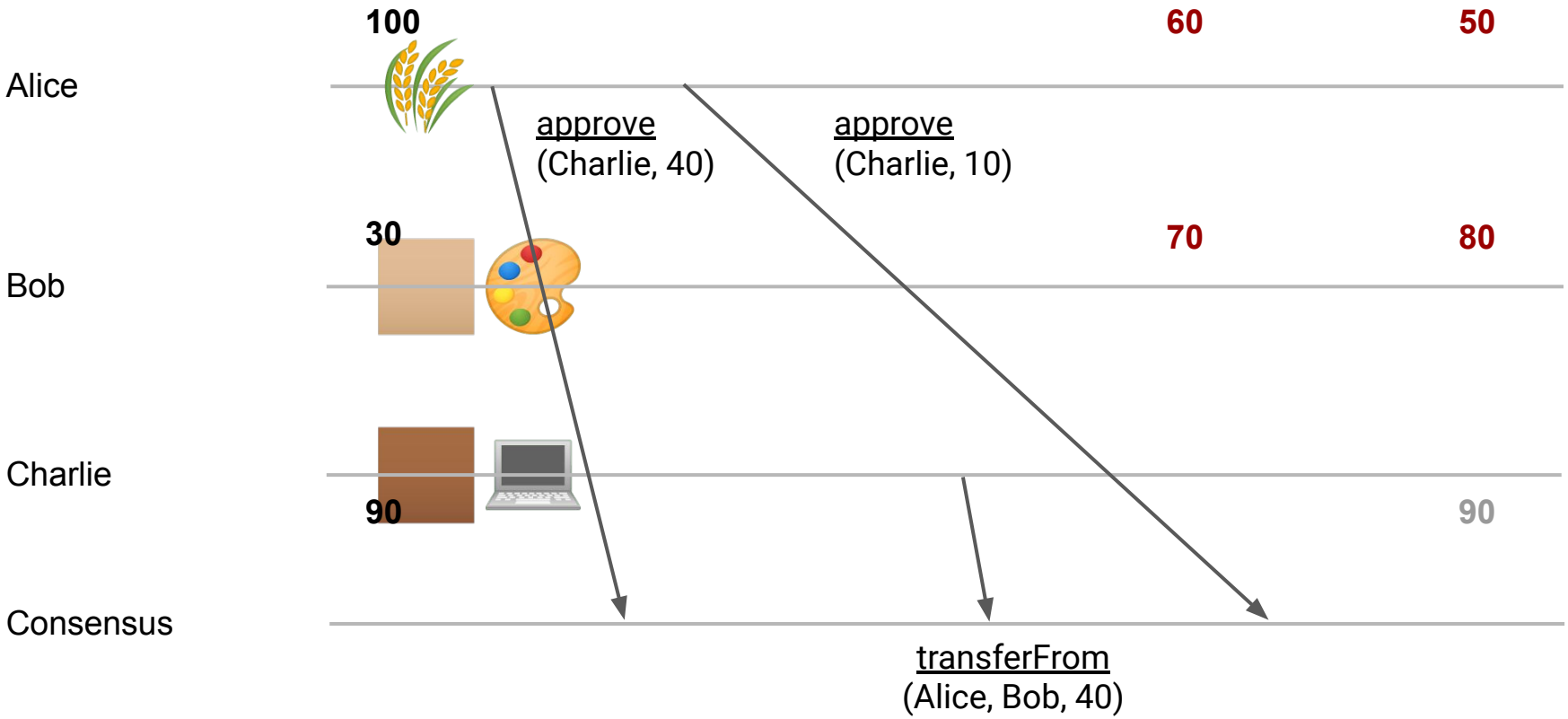
apalache.informal.systems/docs/tutorials/snowcat-tutorial.html

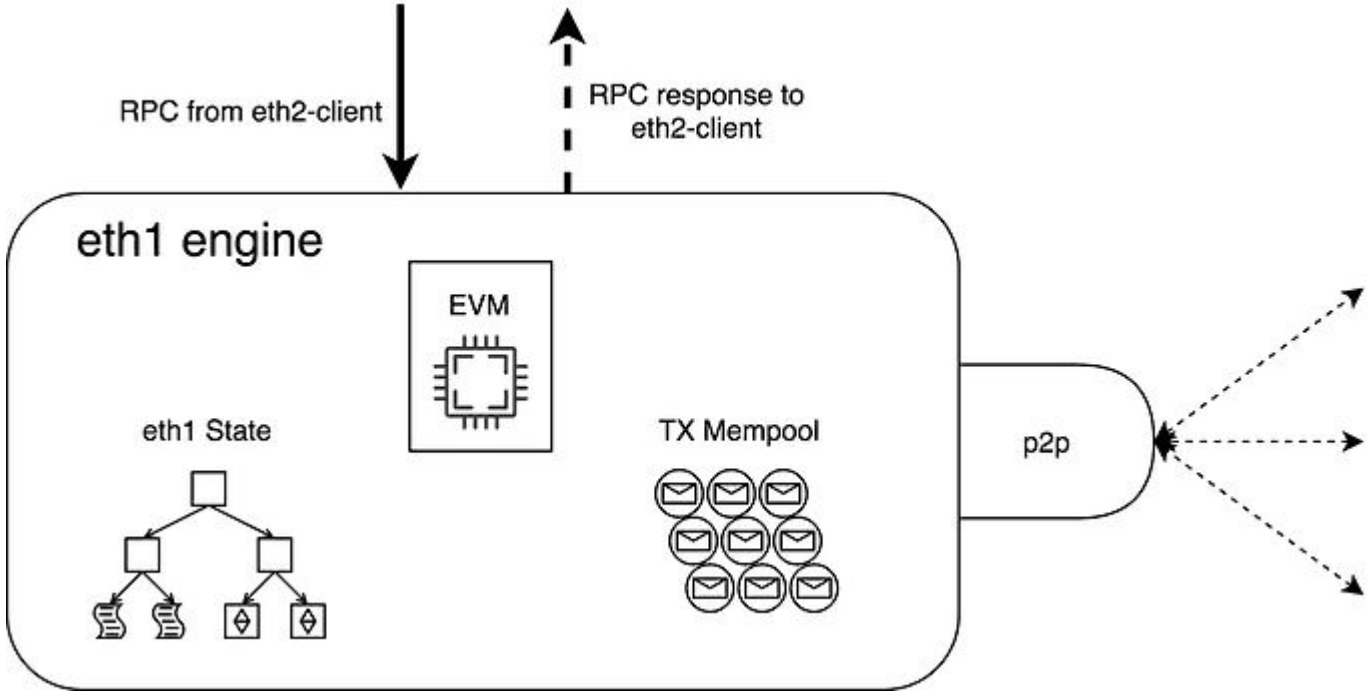
Step 6

Specifying mempool

informal SYSTEMS

Revisit the diagram: an unhappy path





Source: ethereum.org/en/developers/docs/networking-layer/

docs.openzeppelin.com/contracts/4.x/api/token/erc20#IERC20

`approve(address spender, uint256 amount) → bool` external #

Sets `amount` as the allowance of `spender` over the caller's tokens.

Returns a boolean value indicating whether the operation succeeded.

> ⚠ IMPORTANT
>
> Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards: https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

3sGgpQ8H commented on Nov 29, 2016 • edited

Attack vector on ERC20 API (approve/transferFrom methods) and suggested improvements: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/ (commenting enabled)

👍 5 ❤️ 5

States & tx submission

```
------ MODULE erc20_mempool ----------
... \* skipping the constant definitions
INSTANCE erc20

VARIABLES
  \* @type: $state;
  contractState,
  \* @type: Set($tx);
  mempool,
  \* @type: $tx;
  lastTx,
  \* @type: Str;
  lastTxStatus
```

```
submit(tx) ≜
  ∧ mempool' = mempool ∪ { tx }
  ∧ lastTx' = tx
  ∧ lastTxStatus' = "pending"
  ∧ UNCHANGED contractState
```

Committing

```
\* @type: $tx => Bool;
commit(tx) ≜
  ∧ mempool' = mempool \ { tx }
  ∧ ∨ ∧ VariantTag(tx) = "TransferTx"
      ∧ LET ttx ≜
          VariantGetUnsafe("TransferTx", tx) IN
        LET res ≜
          transfer(contractState, ttx.sender,
                   ttx.toAddr, ttx.amount) IN
        fromResult(tx, res)
    ∨ ∧ VariantTag(tx) = "ApproveTx"
      ∧ … \* similar to TransferTx
    ∨ ∧ VariantTag(tx) = "TransferFromTx"
      ∧ … \* similar to TransferTx
```

```
\* boilerplate to propagate the result
\* @type: ($tx, $result) => Bool;
fromResult(tx, result) ≜
  ∧ lastTx' = tx
  ∧ IF VariantTag(result) ≠ "Error"
    THEN
      ∧ lastTxStatus' = "success"
      ∧ contractState' =
          VariantGetUnsafe("Ok", result).state
    ELSE
      ∧ lastTxStatus' =
          VariantGetUnsafe("Error", result)
      ∧ UNCHANGED contractState
```

Exactly the definition from erc20!

Initialization + Transitions

```
Init ≜
  ∧ ∃ sender ∈ AllAddresses \ { ZERO_ADDRESS }:
      ∃ initialSupply ∈ AMOUNTS:
       contractState = newErc20(sender, initialSupply)
  ∧ mempool = {}
  ∧ lastTx = NoneTx
  ∧ lastTxStatus = ""
```

```
Next ≜
  \* execute the contract methods
  ∃ sender ∈ AllAddresses:
    ∃ amount ∈ AMOUNTS:
      ∨ ∃ toAddr ∈ AllAddresses:
          submit(TransferTx(sender, toAddr, amount))
      ∨ ∃ spender ∈ AllAddresses:
          submit(ApproveTx(sender, spender, amount))
      ∨ ∃ fromAddr, toAddr ∈ AllAddresses:
          submit(TransferFromTx(sender, fromAddr,
                                toAddr, amount))
      ∨ ∃ tx ∈ mempool:
          commit(tx)
```

Expectation as an invariant

```
noTransferFromWhileApproveInFlight ≜
 LET Violation ≜
  ∧ lastTxStatus = "success"
  ∧ VariantTag(lastTx) = "TransferFromTx"
  ∧ LET ltx ≜ VariantGetUnsafe("TransferFromTx", lastTx) IN
    ∧ ltx.amount > 0
    ∧ ∃ tx ∈ mempool:
      ∧ VariantTag(tx) = "ApproveTx"
      ∧ LET atx ≜ VariantGetUnsafe("ApproveTx", tx) IN
        ∧ atx.sender = ltx.fromAddr ∧ atx.spender = ltx.sender
        ∧ atx.amount < ltx.amount ∧ atx.amount > 0
 IN
 ¬Violation
```

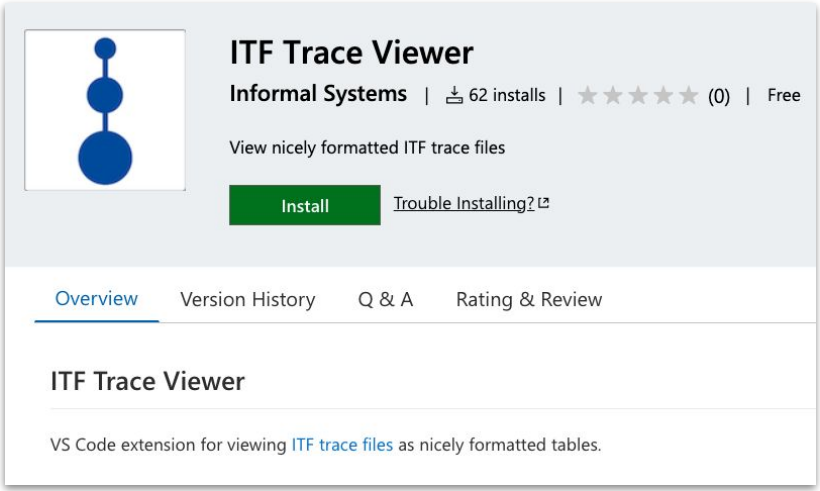
```
$ apalache-mc check \
--inv=noTransferFromWhileApproveInFlight \
  erc20_mempool.tla
```

$ ❌ in 3 seconds

Counterexample

```
mempool
{
  tag   : "ApproveTx"
  value : amount  : 2
          sender  : "bob_OF_ADDR"
          spender : "alice_OF_ADDR"
,
  tag   : "TransferFromTx"
  value : amount   : 3
          fromAddr : "bob_OF_ADDR"
          sender   : "alice_OF_ADDR"
          toAddr   : "alice_OF_ADDR"
}

#          5
lastTx     tag   : "TransferFromTx"
           value : amount   : 3
                   fromAddr : "bob_OF_ADDR"
                   sender   : "alice_OF_ADDR"
                   toAddr   : "alice_OF_ADDR"
lastTxStatus "success"
mempool    {
             tag   : "ApproveTx"
             value : amount  : 2
                     sender  : "bob_OF_ADDR"
                     spender : "alice_OF_ADDR"
             }
```



by Hernan Vanzetto @ Informal

ITF: Informal Trace Format

- TLA$^+$ traces in JSON
- Every engineer can parse JSON
- Extremely simple format
- Designed for tool integration


```
{
  "#meta": {
    "format": "ITF",
    "varTypes": {
      ..
    }
  },
  "vars": [ "lastTx", "mempool", ... ],
  "states": [ {
    "#meta": { "index": 0 },
    "lastTxStatus": "",
    "mempool": { "#set": [ ] },
    "contractState": {
      "allowance": {
        "#map": [
          [
            { "#tup": [ "alice_OF_ADDR", "0_OF_ADDR" ] },
            0
          ],
          ...
```


apalache.informal.systems/docs/adr/015adr-trace.html

Step 7

Discussion

informal SYSTEMS

Functional operator composition

```
transferFrom(state,
  sender, fromAddr, toAddr, amount) ≜
  ...
  IF err ≠ ""
  THEN Error(err)
  ELSE transfer(updatedState,
    fromAddr, toAddr, amount)
```

- We rarely see it in TLA$^+$ specs

- ERC20 is closer to the original contract – easier to match the code
- Composition $F(G(x))$ does not require the \cdot operator: $A \cdot B$
- Easier to reuse in other specs, as we have seen in the mempool example

- The logic of TLA$^+$ supports this kind of modeling too

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

apalache.informal.systems

Apalache

The Symbolic Model Checker for TLA+

View the Project on GitHub
informalsystems/apalache

Download ZIP File	Download TAR Ball	View On GitHub

Tweets from @ApalacheTLA

Apalache: symb...
@A... · Sep 7, 2022

You were asking for precise type checking of @tlaplus records in Apalache, a lot. Since yesterday (v0.29.0), it is the default. We will keep the old records for backwards compatibility till October 30, 2022. How to transition to new records and variants: apalache.informal.systems/docs/HOWTOs/ho...

4

Apalache: symbolic model checker for TLA+

Features • Installation • Manual • Releases • Chat • Contribute

Apalache translates TLA+ into the logic supported by SMT solvers such as Microsoft Z3. Apalache can check inductive invariants (for fixed or bounded parameters) and check safety of bounded executions (bounded model checking). To see the list of supported TLA+ constructs, check the supported features. In general, Apalache runs under the same assumptions as TLC.

To learn more about TLA+, visit Leslie Lamport's page on TLA+ and see his video course. Also, check out TLA+ language manual for engineers.

Tutorials

- Type checking TLA+ with Snowcat
- Extended version of the Apalache tutorial

Talks

- Informal Systems Tutorial: TLA+ Basics
- Extended version of the Apalache tutorial. TLA+ tutorial at DISC 2021 (October 2021).
- How TLA+ and Apalache Helped Us to Design the Tendermint Light

Model-based testing

hanoi.tla × test_hanoi.py 1

model > hanoi.tla > {} hanoi

Udit Gulati, 2 months ago | 1 author (Udit Gulati)

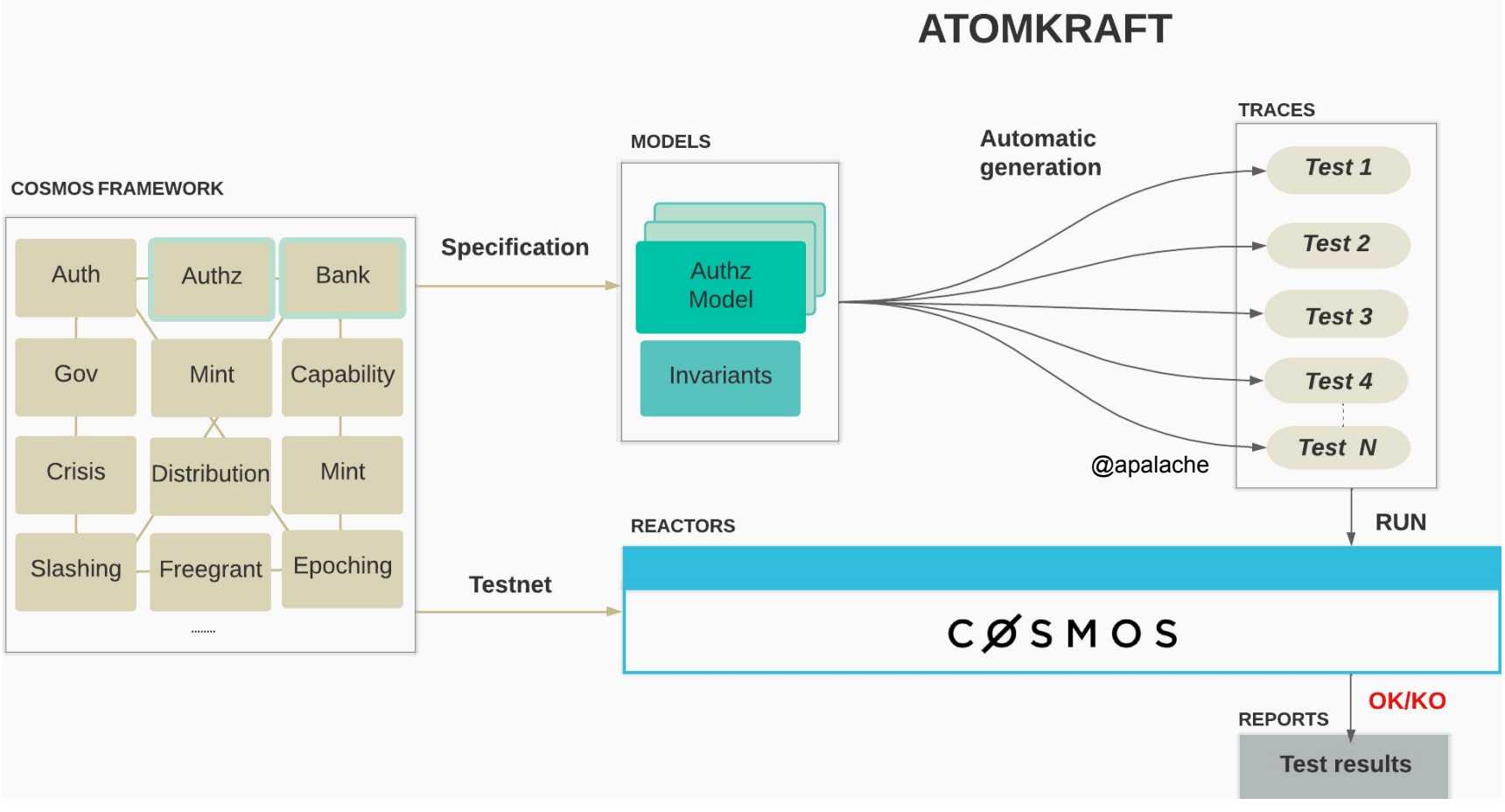
```
1  ---- MODULE hanoi ----
2  EXTENDS Apalache, Integers, Sequences, SequencesExt
3
4  VARIABLES
5      \* @type: Seq(Seq(Int));
6      hanoi,
7      \* @type: {tag: Str, source: Int, target: Int};
8      action
9
10 N_DISC == 3
```

Udit Gulati, 2 months ago • check if model file exists

PROBLEMS 1 OUTPUT TERMINAL GITLENS DEBUG CONSOLE bash

```
modelator/examples/hanoi on ⎇ dev [!?] is 📦 v0.1.0 via 🐍 v3.10.10
○ at 21:29:30 ❯
```

Automatic generation/execution of test suites



Appendix
Symbolic simulation

informal SYSTEMS

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? ⏱️

Init									
A1									
A1	\/ A2	\/ A3	\/ A4						
A1	\/ A2	\/ A3	\/ A4	\/ A5	\/ A6				
A1	\/ A2	\/ A3	\/ A4	\/ A5	\/ A6	\/ A7	\/ A8	\/ A9	\/ A10
A1	\/ A2	\/ A3	\/ A4	\/ A5	\/ A6	\/ A7	\/ A8	\/ A9	\/ A10
A1	\/ A2	\/ A3	\/ A4	\/ A5	\/ A6	\/ A7	\/ A8	\/ A9	\/ A10

Randomized symbolic execution

Init									
A1									
A1	∨ A2	∨ A3	∨ A4						
A1	∨ A2	∨ A3	∨ A4	∨ A5	∨ A6				
A1	∨ A2	∨ A3	∨ A4	∨ A5	∨ A6	∨ A7	∨ A8	∨ A9	∨ A10
A1	∨ A2	∨ A3	∨ A4	∨ A5	∨ A6	∨ A7	∨ A8	∨ A9	∨ A10
A1	∨ A2	∨ A3	∨ A4	∨ A5	∨ A6	∨ A7	∨ A8	∨ A9	∨ A10

¬Invariant

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?**