Isabelle/HOL on Fork Prevention in the Coming Ethereum Protocol

Yoichi Hirai
Ethereum Foundation

Berlin, 30 Aug. 2017

What is Ethereum

Ethereum is one instance of a virtual machine, which is

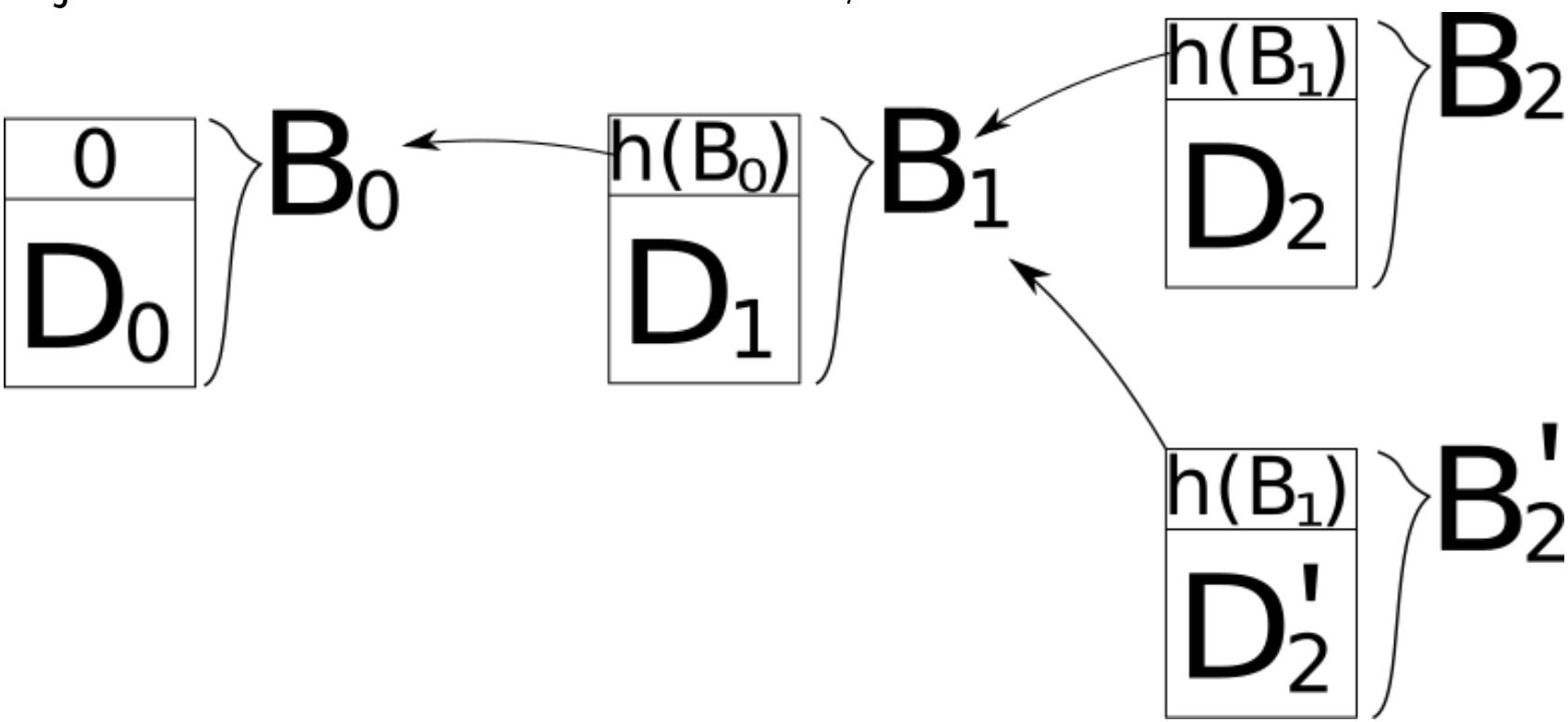
- as powerful as a 20-year-old smart phone
- replicated globally
- with no central parties
- running for 2 years by now

How does Ethereum synchronize execution traces?

Asynchronous communication cannot establish new common knowledge [Chandy & Misra, 1986].

A Blockchain

is just a data strucuture. It is a tree, not a chain.



A small number $h(B_n)$ identifies a sequence $B_0, \ldots B_n$ and moreover, in Ethereum, the execution trace of a virtual machine.

Proof of Work

- a block's content D needs to contain a nicely chosen nonce so that the hash $h(D)$ of the block is small enough
- when you find such a nonce, the protocol gives reward in your account (in the state after the block)
- account?

Finding a good nonce is called mining.

I belive nobody has found a better way than brute-forcing.

"Ethereum miners are renting Boeing 747s to ship graphics cards and AMD shares are soaring" (`https://qz.com/1039809`)
(sounds a bit like Moai's).

Proof of Work

If you succeed creating a block

1. the block reward is only spendable in chains that include your block
2. you should send around the block, maybe

If you see a block being sent around

1. if that's the best block you've seen, you should try to mine on the block
   (assumption: absense of complicated motives, other nodes' straightforward behavior)
2. you should broadcast the block you are mining on, maybe

Does it work? Yes, see Bitcoin.
Why? I don't know, honestly.

Evaluating Proof of Work

No good properties distributed-computation-wise.
(A good leader-election protocol can tolerate less than 1/3 Byzantine nodes.)

One Byzantine node can

- be lucky enough to guess the secret keys of everyone.

One (economically) irrational node can

- buy lots of machines to mine quicker than anybody, rewriting the history from any point in the past.

Why this "algorithm"? number of nodes is not reliable.
Instead, Proof-of-Work uses electricity consumption (or luck).
Instead, Proof-of-Stake uses in-protocol deposit.
These designs require numbers that carry values associated to public keys.

Proof-of-Stake

- Replacing GPU and electricity with deposits of in-protocol tokens.
- A difference: in-protocol tokens duplicate as forks happen.
- Solution: provably dishonest behavior is punished on all forks.

Slash those who's active on multiple chains!
is too intimidating.

I Received a Challenge: Original Text

Message types

- commit(HASH, view)
- prepare(HASH, view, view_source),
  -1 $\leq$ view_source $<$ view

Slashing conditions

1. commit(H, v) REQUIRES 2/3 prepare(H, v, vs) for some consistent vs
2. prepare(H, v, vs) REQUIRES 2/3 prepare(H_anc, vs, vs') for some consistent vs', where H_anc is a (v-vs)-degree ancestor of H, UNLESS vs = -1
3. commit(H, v) + prepare(H, w, u) ILLEGAL if u $<$ v $<$ w
4. prepare(X1, v, vs1) + prepare(X2, v, vs2) ILLEGAL unless X1 = X2 and vs1 = vs2

Challenge Continued

Accountable safety argument

(proof path - assume two incompatible values got committed, show 1/3+ SLASHED)

Case 1

2/3 commit(X, v) & 2/3 commit(Y, v)

→ 2/3 prepare(X, v, vs) & 2/3 prepare(Y, v, vs') (1.)

→ 1/3 SLASHED (4.)

Case 2

2/3 commit(Y, v2) & 2/3 commit(X, v1),

Y is NOT a (v2-v1)-degree descendant of X, define Y[i] to be the ancestor of Y in view i

→ 2/3 prepare(Y[v2], v2, vs), vs $<$ v2 (1.)

→ 2/3 prepare(Y[vs], vs, vs') (2.)

→ ...

[continue induction until vs' $<$ v1]

(Two base cases follow.)

Alloy modelling

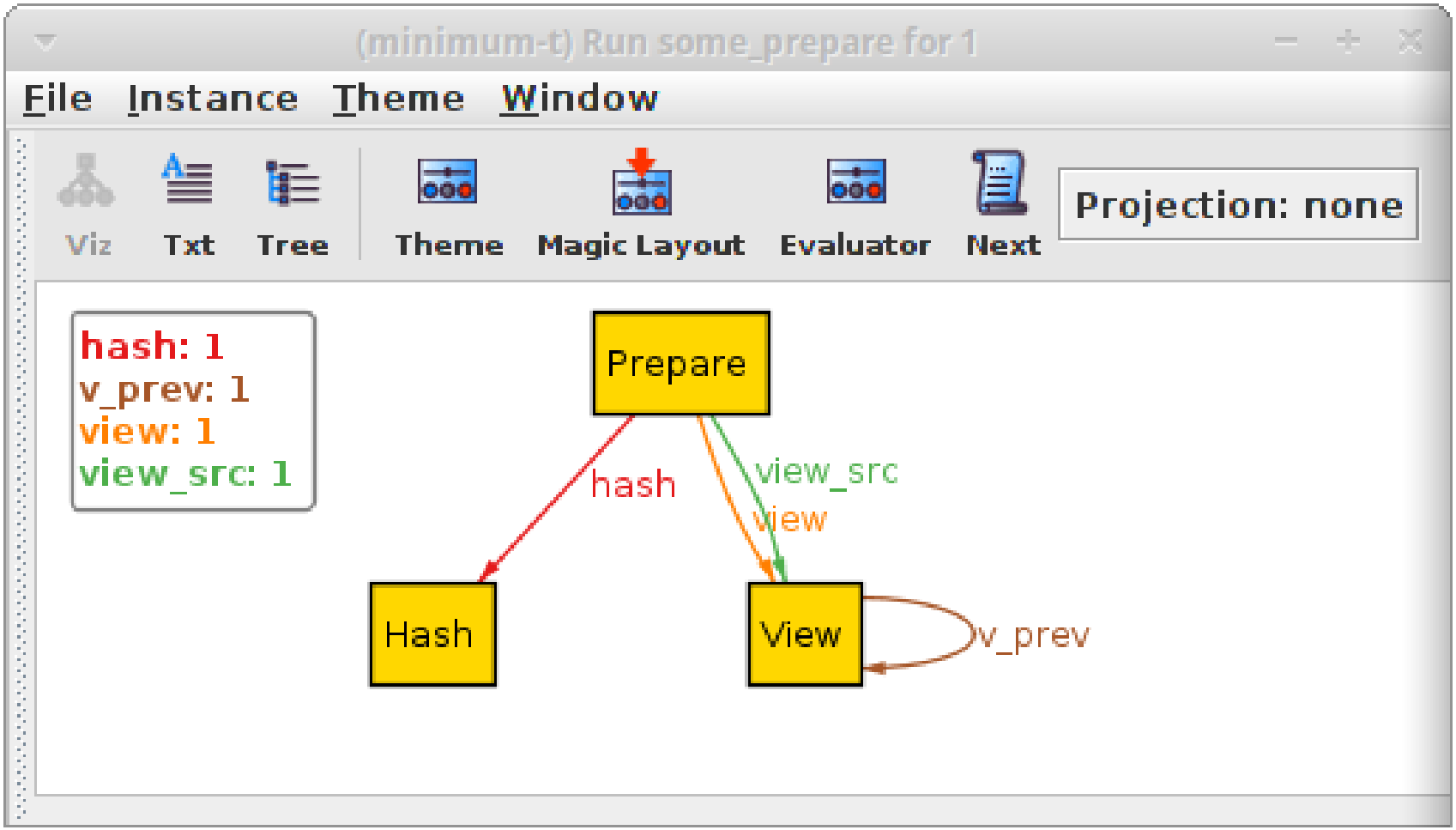
Being unsure what the definitions meant, I turned to Alloy.
Alloy `http://alloy.mit.edu/alloy/` is a prototyping-tool.

You can type in definitions, assumptiions and a conjecture in relational algebra (a bit more expressible than FOL).

Alloy tries to find a counterexample. No guarantees of false-negatives.

Alloy example

```
sig View { v_prev: lone View }

sig Hash { h_prev: lone Hash }

fact { no x : Hash | x in x.(^h_prev) }

sig Prepare { hash : Hash,
              view : View,
              view_src : View }

fact{ all p : Prepare | p.view_src in (p.view.(^v_prev))}

pred some_prepare { some Prepare }

run some_prepare for 3
```


(minimum-t) Run some_prepare for 1
File Instance Theme Window
Viz
Txt
Tree
Theme
Magic Layout
Evaluator
Next
Projection: none
hash: 1
v_prev: 1
view: 1
view_src: 1
Prepare
hash
view_src
view
Hash
View
v_prev

Somehow I can code the Slashing Conditions

Text: commit(H, v) REQUIRES 2/3 prepare(H, v, vs) for some consistent vs

```
// Slashing condition [i]
pred slashed1 (s : Node) {
   some c : Commit |
      s in c.c_sender &&
      (#{n : Node | some p : Prepare |
            p.p_sender = n
         && p.p_hash = c.c_hash})

     . mul[ 3]
     < mul[ #{n : Node}, 2 ]
}
```

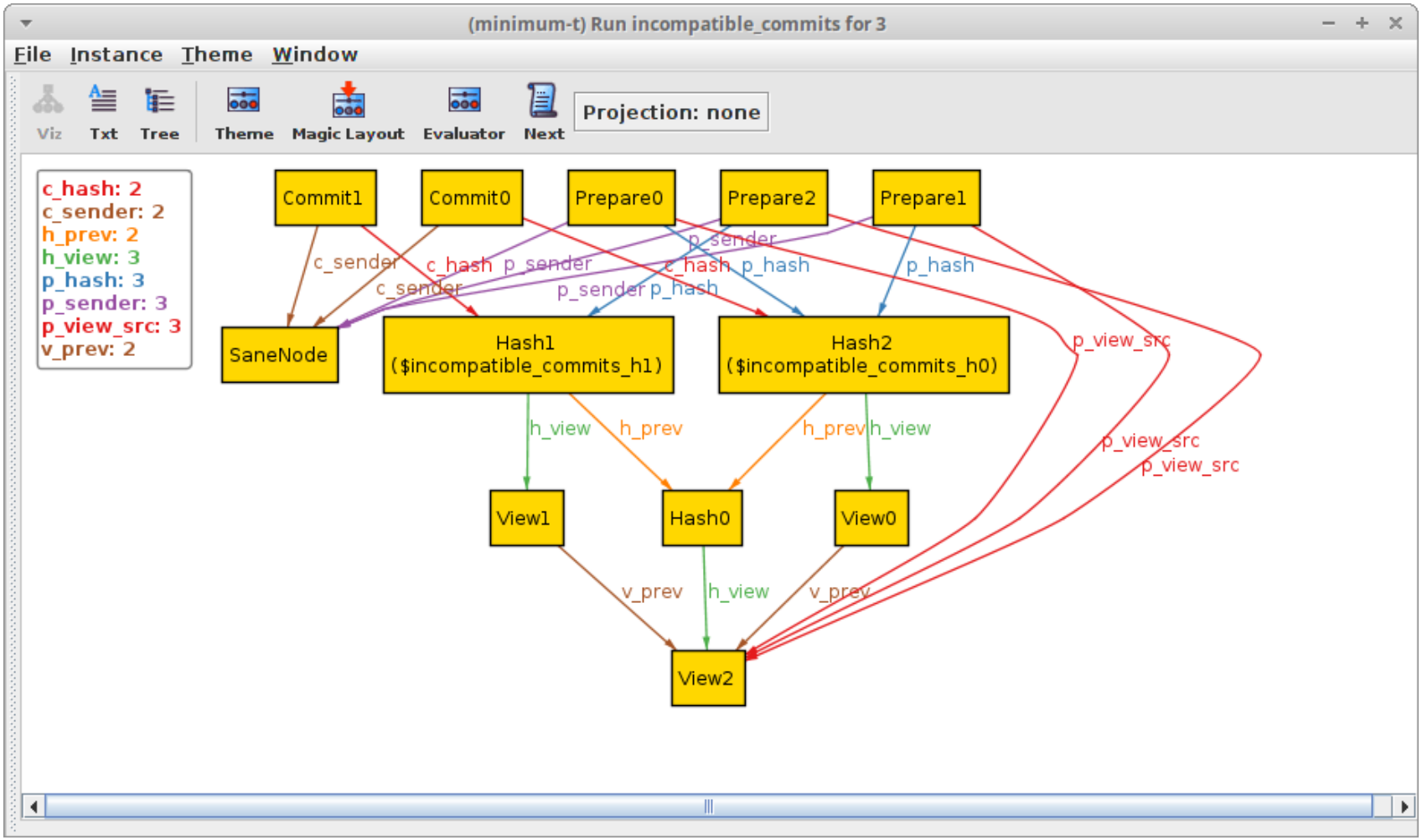
etc.

I can also define a fork with 2/3 non-slashed nodes

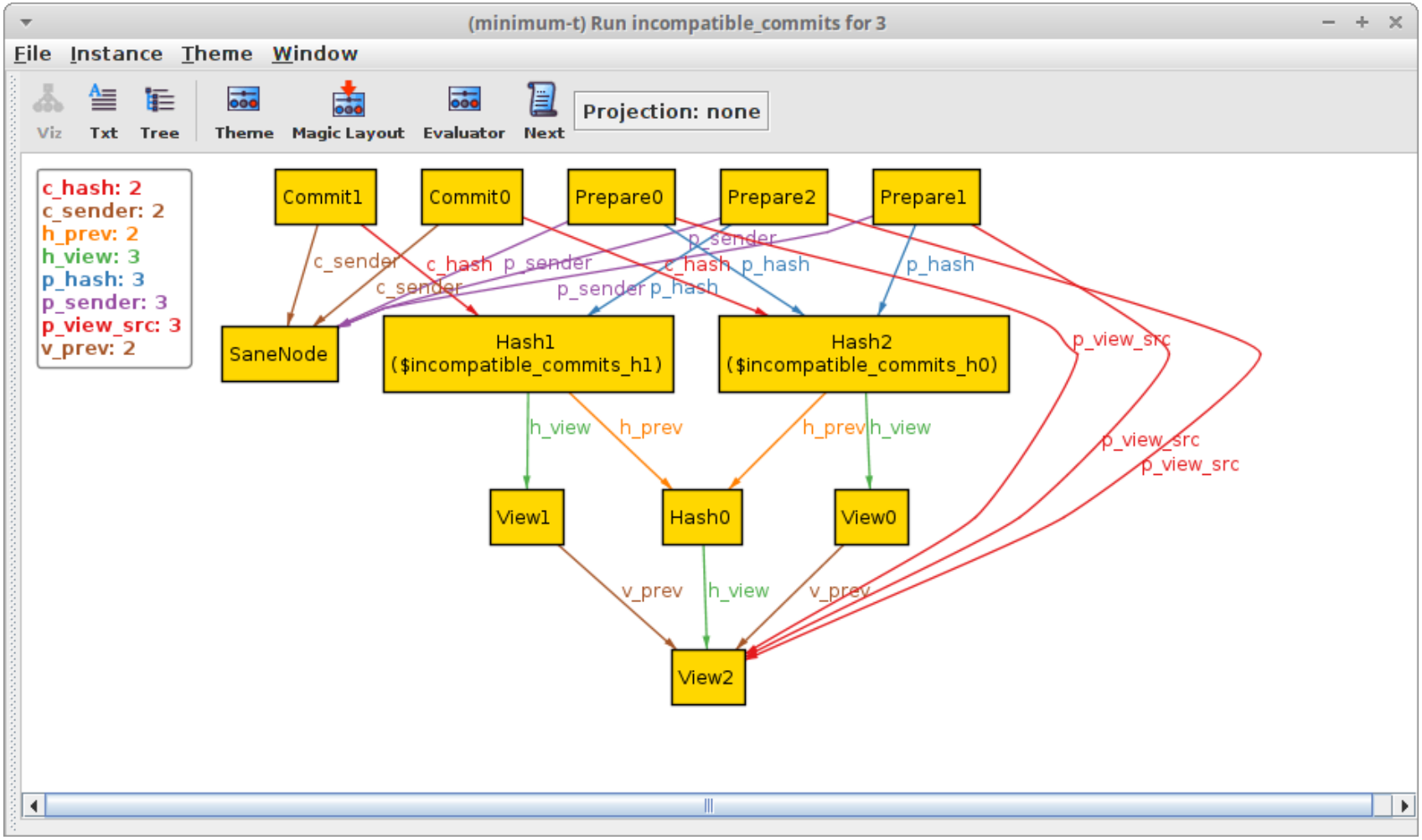
```
pred incompatible_commits {
    some Node &&
    some h0, h1 : Hash |
     (not h0 in h1.(*h_prev)) &&
     (not h1 in h0.(*h_prev)) &&
     (#{n0 : Node |
          some c0 : Commit |
          c0.c_sender = n0 && c0.c_hash = h0}
     ).mul[3] >= (#Node).mul[2] &&
     (#{n1 : Node |
          some c1 : Commit |
          c1.c_sender = n1 && c1.c_hash = h1}
     ).mul[3] >= (#Node).mul[2] &&
     (#SlashedNode).mul[3] < (#Node)
}
```

Now Alloy, go find incompatible commits.

Fork?

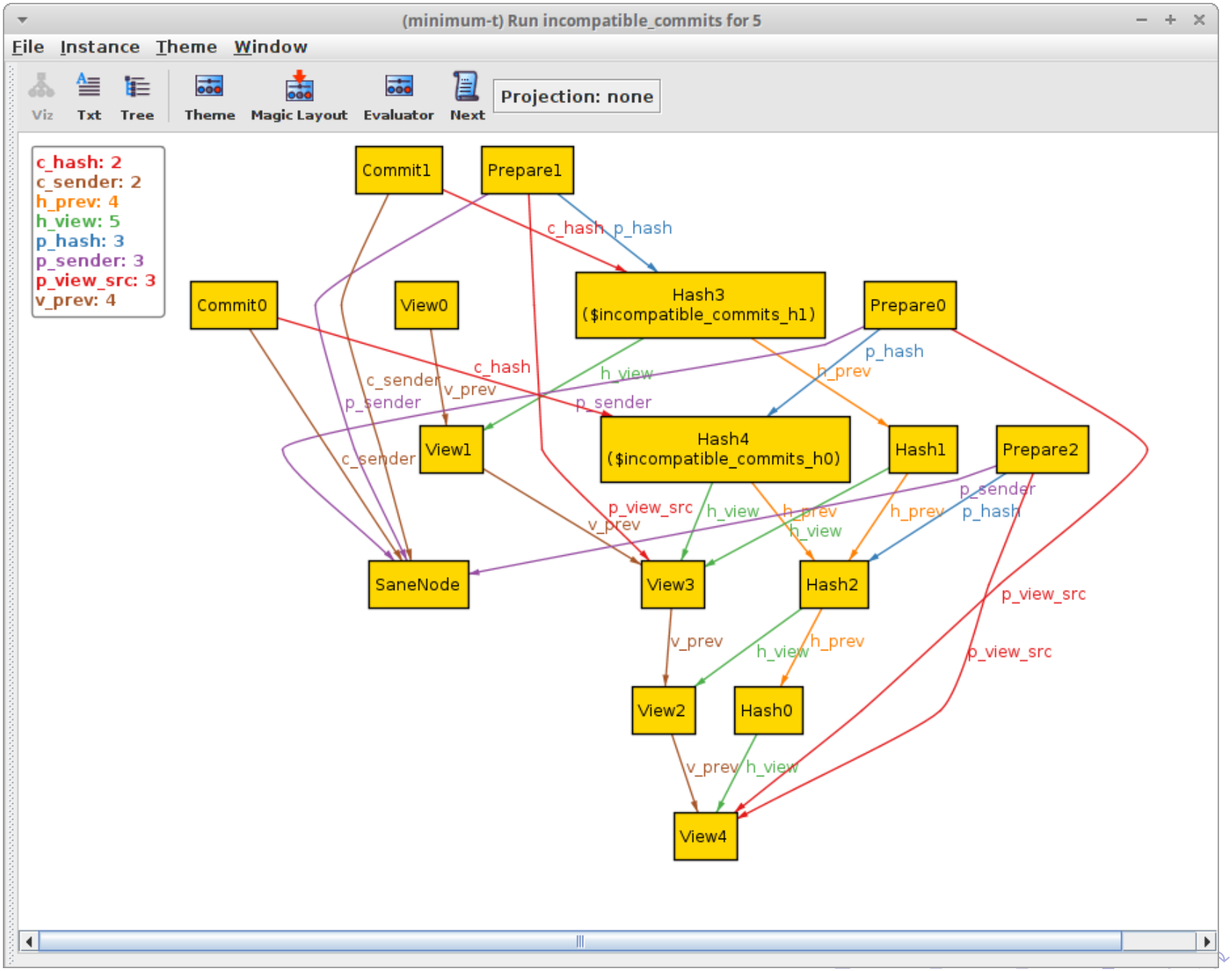

(minimum-t) Run incompatible_commits for 3
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projection: none
c_hash: 2
c_sender: 2
h_prev: 2
h_view: 3
p_hash: 3
p_sender: 3
p_view_src: 3
v_prev: 2
Commit1
Commit0
Prepare0
Prepare2
Prepare1
SaneNode
Hash1
($incompatible_commits_h1)
Hash2
($incompatible_commits_h0)
View1
Hash0
View0
View2
c_sender
c_hash
p_sender
p_hash
h_view
h_prev
v_prev
p_view_src

Fork?



Mistake: I forgot to specify Views have a total order.

Another Fork?

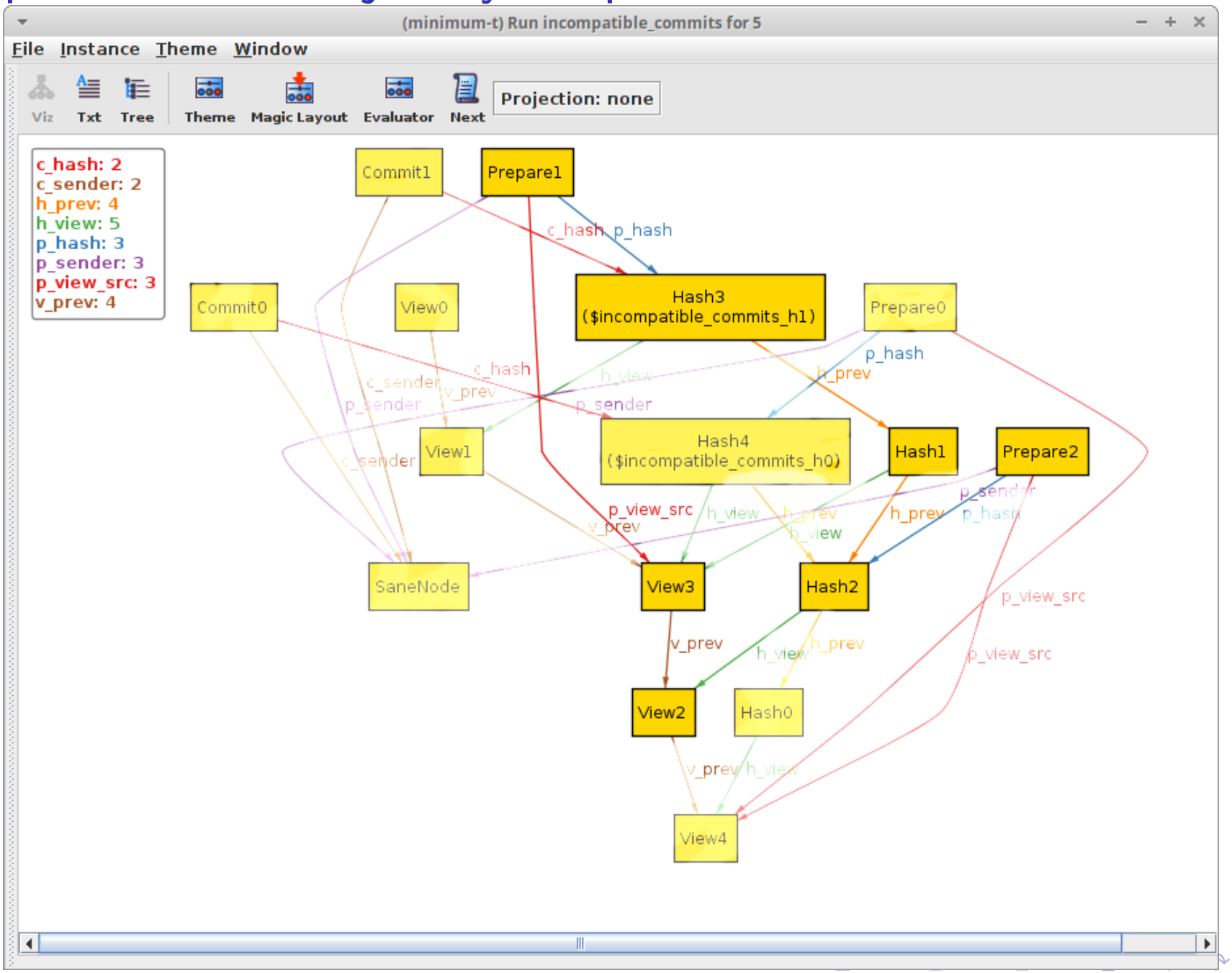


Another Fork? Mistake

"prepare(H, v, vs) REQUIRES 2/3 prepare(H_anc, vs, vs') for some consistent vs', where H_anc is a (v-vs)-degree ancestor of H, UNLESS vs = -1"

```
fact {
    all p : Prepare |
        (p.p_sender in SaneNode && some p.p_view_src.v_prev)
        some h_anc : Hash | some v_src : View |
        h_anc in p.p_hash.(^h_prev) &&
+       h_anc.h_view = p.p_view_src &&
         (#{n : Node | some p' : Prepare | p'.p_sender = n
                && p'.p_hash = h_anc && p'.p_view_src = v_src
          . mul[ 3]  >= (#{n : Node}).mul[ 2 ]
}
```

Prepare2 does not justify Prepare1



Isabelle/HOL

When I was confident I turned to Isabelle/HOL.

Turned out followable in Isabelle/HOL (1,800 lines).

Messages and Validators

```
datatype hash = Hash int
type_synonym view = int
datatype message =
  Commit "hash * view"
| Prepare "hash * view * view"
datatype validator = Validator int
type_synonym signed_message = "validator * message"
```

Situation

record situation =
  Validators :: "validator set"
  Messages :: "signed_message set"
  PrevHash :: "hash ⇒ hash option"

Ancestors

```
fun nth_ancestor :: "situation ⇒ nat ⇒ hash ⇒ hash option"
where
  "nth_ancestor _ 0 h = Some h"
| "nth_ancestor s (Suc n) h =
   (case PrevHash s h of
      None ⇒ None
    | Some h' ⇒ nth_ancestor s n h')"
```

definition is_descendant_or_self :: "situation ⇒ hash ⇒ hash ⇒ bool"
where
"is_descendant_or_self s x y = (∃ n. nth_ancestor s n x = Some y)"

definition not_on_same_chain :: "situation ⇒ hash ⇒ hash ⇒ bool"
where
"not_on_same_chain s x y = ((¬ is_descendant_or_self s x y) ∧ (¬ is_descendant_or_self s y x))"

definition two_thirds :: "situation ⇒ (validator ⇒ bool) ⇒ bool"
where
"two_thirds s f =
(2 * card (Validators s) ≤
3 * card ({n. n ∈ Validators s ∧ f n}))"

definition prepared :: "situation ⇒ hash ⇒ view ⇒ view ⇒ bool"
where
"prepared s h v vs =
(two_thirds_sent_message s (Prepare (h, v, vs)))"

definition committed :: "situation ⇒ hash ⇒ bool"
where
"committed s h =
(∃ v. two_thirds_sent_message s (Commit (h, v)))"

[i] A validator is slashed when it has sent a commit message of a hash that is not prepared yet.

definition slashed_one :: "situation ⇒ validator ⇒ bool"
where
"slashed_one s n =
 (n ∈ Validators s ∧
   (∃ h v.
     ((n, Commit (h, v)) ∈ Messages s ∧
   (¬ (∃ vs. -1 ≤ vs ∧ vs ¡ v ∧ prepared s h v vs)))))"

[ii] A validator is slashed when it has sent a prepare message whose view src is not -1 but has no supporting preparation in the view src.

definition slashed_two :: "situation ⇒ validator ⇒ bool"
where
"slashed_two s n =
  (n ∈ Validators s ∧
    (∃ h v vs.
      ((n, Prepare (h, v, vs)) ∈ Messages s ∧
      vs ≠ -1 ∧
      (¬ (∃ h_anc vs'.
          -1 ≤ vs' ∧ vs' ¡ vs ∧
          Some h_anc = nth_ancestor s (nat (v - vs)) h ∧
          prepared s h_anc vs vs')))))"

[iii] A validator is slashed when it has sent a commit message and a prepare message containing view numbers in a specific constellation.

definition slashed_three :: "situation ⇒ validator ⇒ bool"
where
"slashed_three s n =
  (n ∈ Validators s ∧
    (∃ x y v w u.
      (n, Commit (x, v)) ∈ Messages s ∧
      (n, Prepare (y, w, u)) ∈ Messages s ∧
      u ¡ v ∧ v ¡ w))"

[iv] A validator is slashed when it has signed two different Prepare messages at the same view.

definition slashed_four :: "situation ⇒ validator ⇒ bool"
where
"slashed_four s n =
  (n ∈ Validators s ∧
    (∃ x1 x2 v vs1 vs2.
      (n, Prepare (x1, v, vs1)) ∈ Messages s ∧
      (n, Prepare (x2, v, vs2)) ∈ Messages s ∧
      (x1 ≠ x2 ∨ vs1 ≠ vs2)))"

A validator is slashed when at least one of the above conditions [i]–[iv] hold.

definition slashed :: "situation ⇒ validator ⇒ bool"
where
"slashed s n = (slashed_one s n ∨
slashed_two s n ∨
slashed_three s n ∨
slashed_four s n)"

definition one_third_slashed :: "situation ⇒ bool"
where
"one_third_slashed s = one_third s (slashed s)"

Conclusion

The statement of accountable safety is simple. If a situation has a finite number of validators (but not zero), if two hashes x and y are committed in the situation, but if the two hashes are not on the same chain, at least one-third of the validators are slashed in the situation.

lemma accountable_safety :
"situation_has_finitely_many_validators s $\Longrightarrow$
committed s x $\Longrightarrow$ committed s y $\Longrightarrow$
not_on_same_chain s x y $\Longrightarrow$ one_third_slashed s"

After that

- the result was expanded to allow the validator set change
- The “Casper contract” is not verified according to these slashing conditions.

Other formal verification projects

- eth-isabelle: an executable specification of Ethereum Virtual Machine in Lem: available for Isabelle/HOL too.
- a compiler on top of eth-isabelle by @mrsmkl
- some ongoing work on EVM1.5
- KEVM by Grigore Rosu and his team