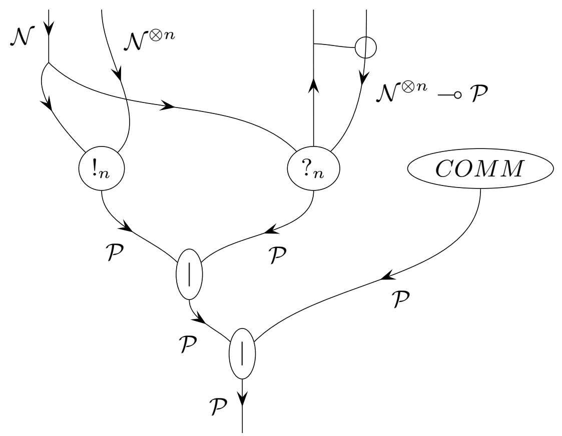
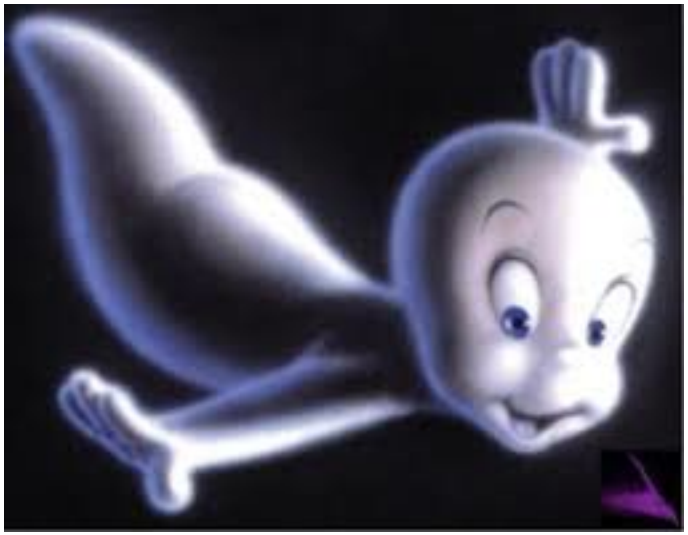
Casper Formalized Pt I

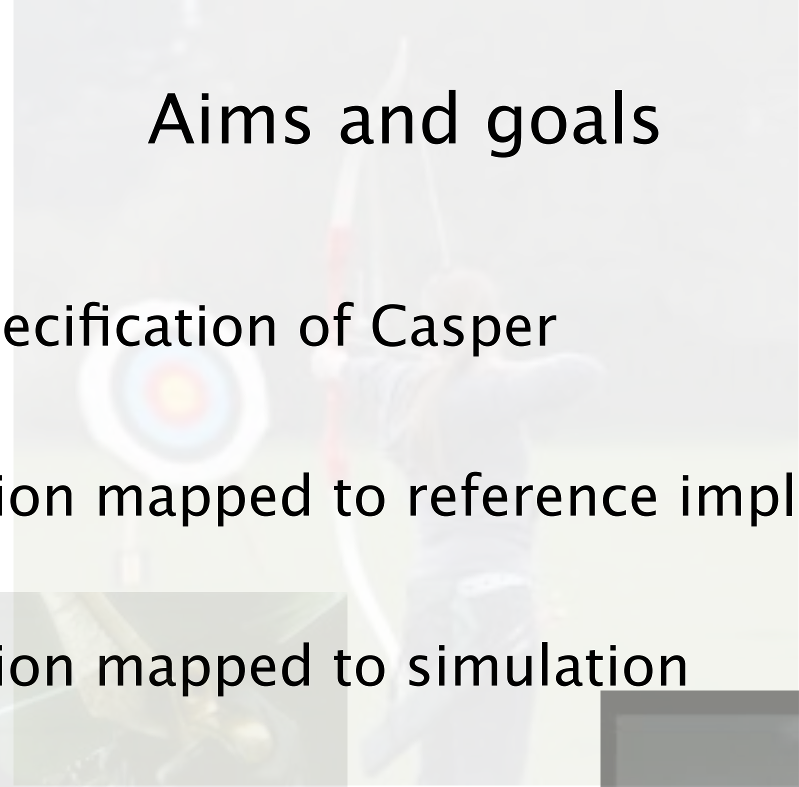
Applying π-calculus to modeling the Casper protocol





Vlad Zamfir, L.G. Meredith

Aims and goals

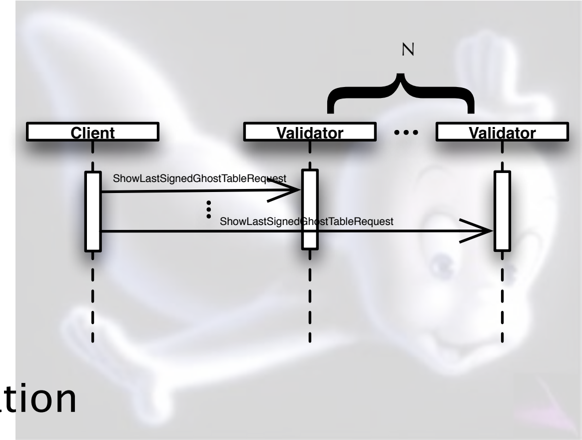


Formal specification of Casper

Specification mapped to reference implementation

Specification mapped to simulation

Methodology



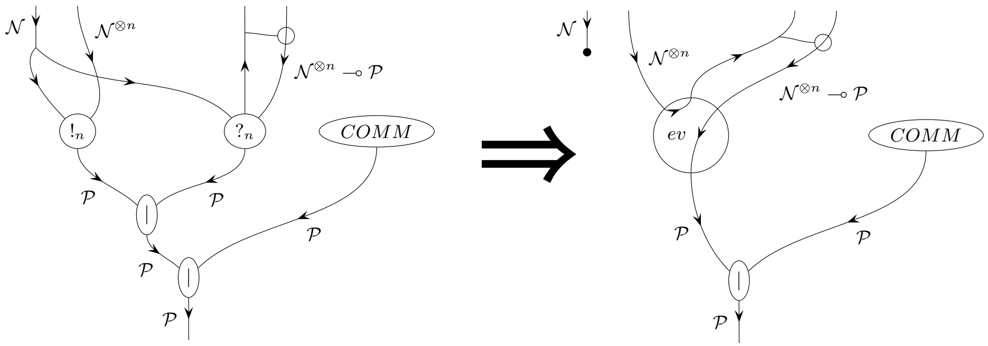
Bootstrap Casper spec using interaction diagrams

Map interaction diagrams to initial π-calculus specification

Refine specification

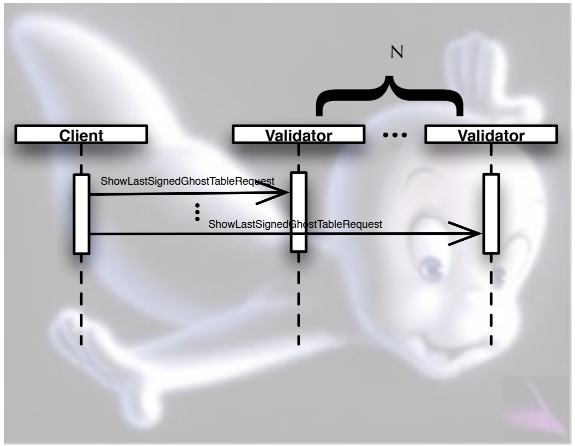
Map refined specification to SpecialK

Map refined specification to Stochastic π-machine

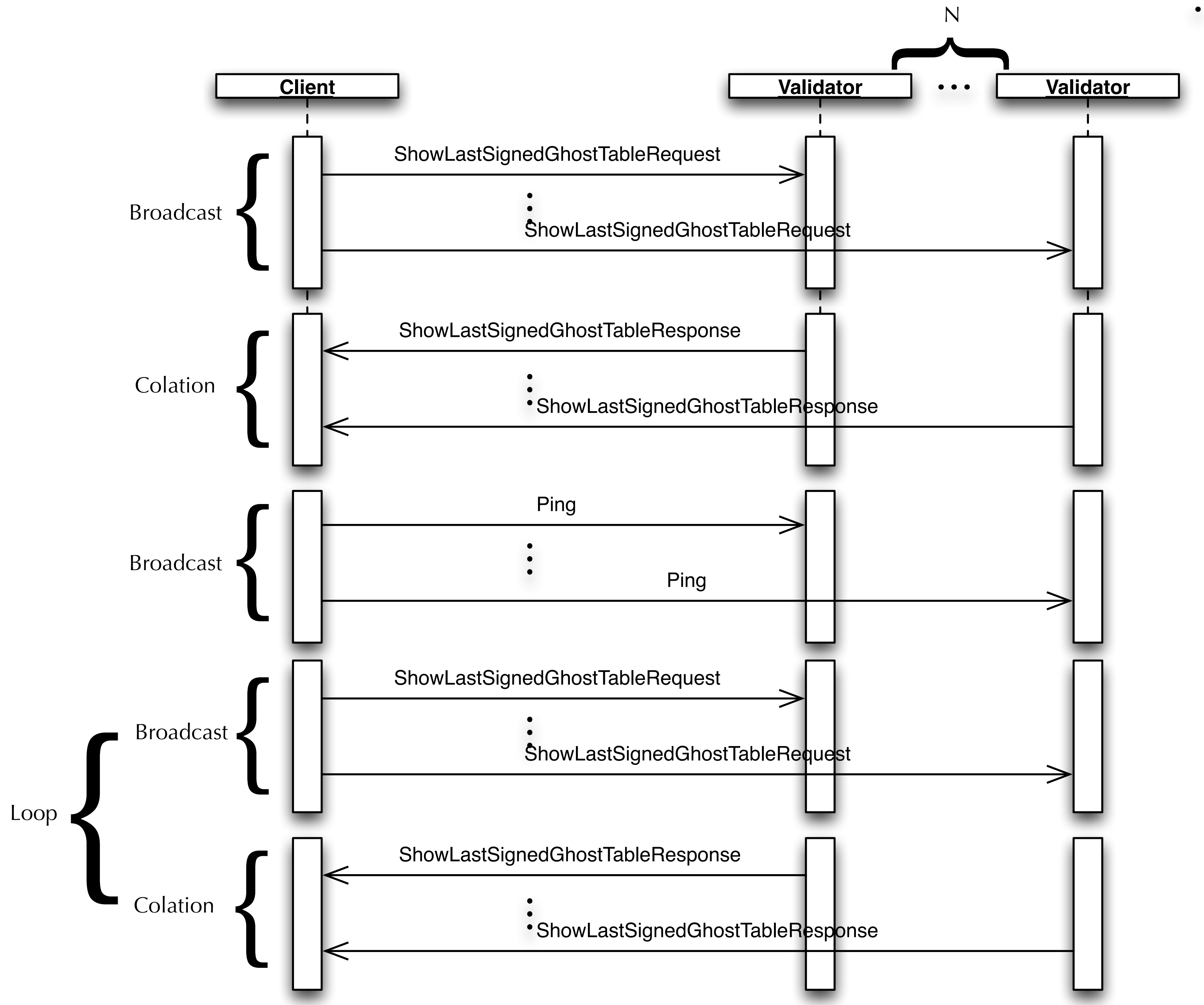


Casper in π-calculus

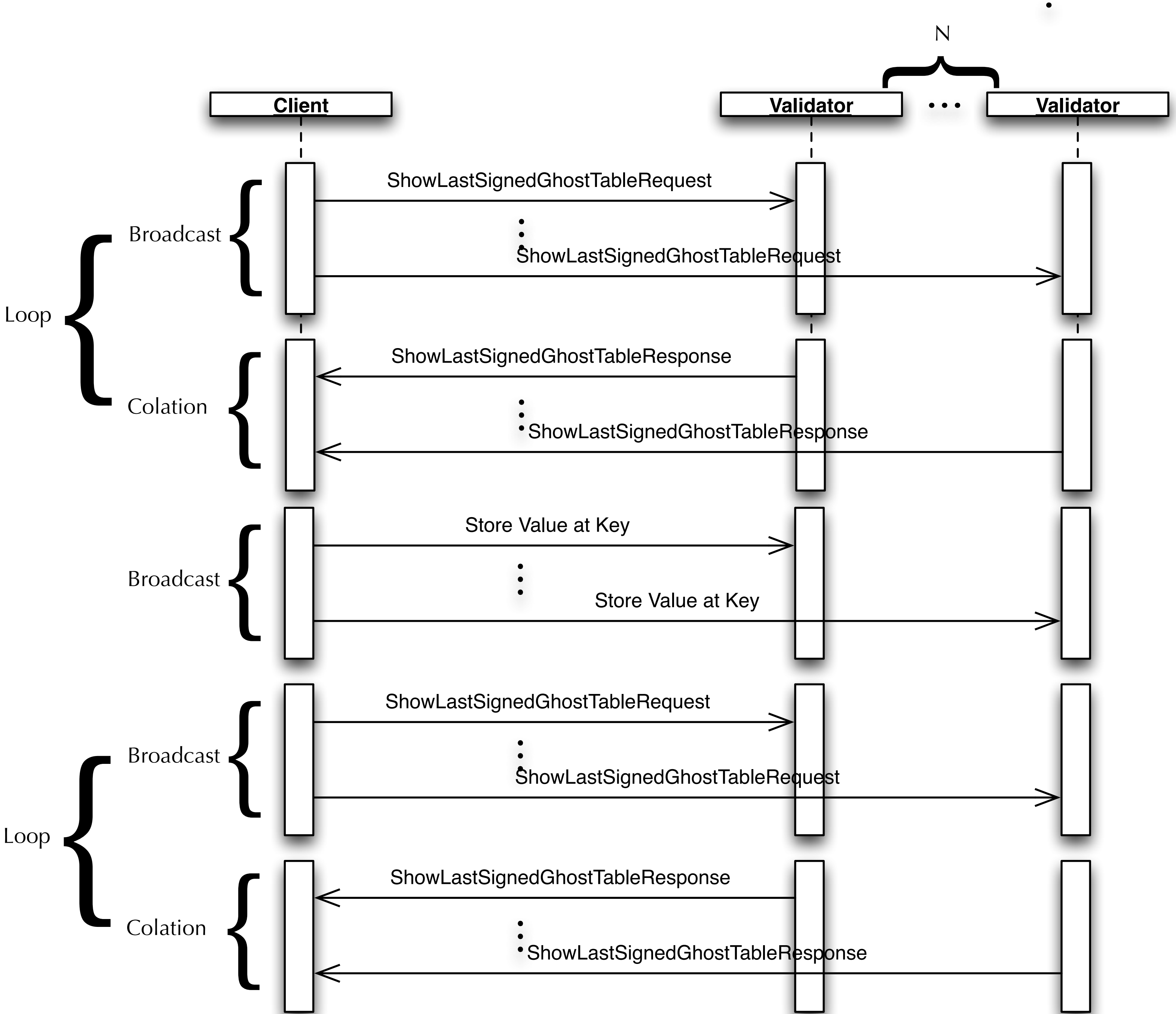
Casper in interaction diagrams



Client <-> Validator interactions

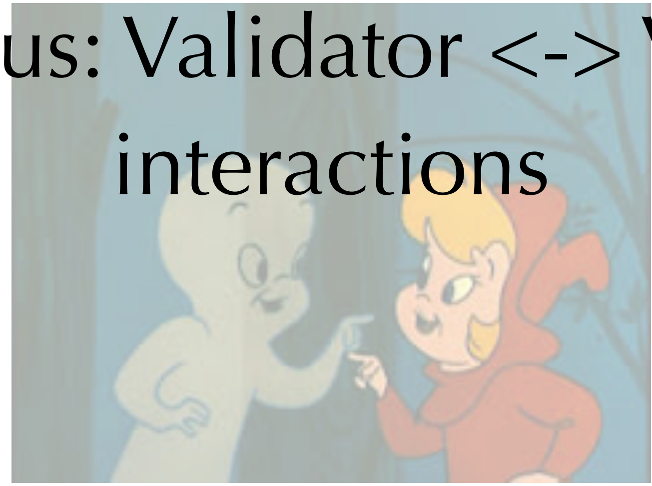


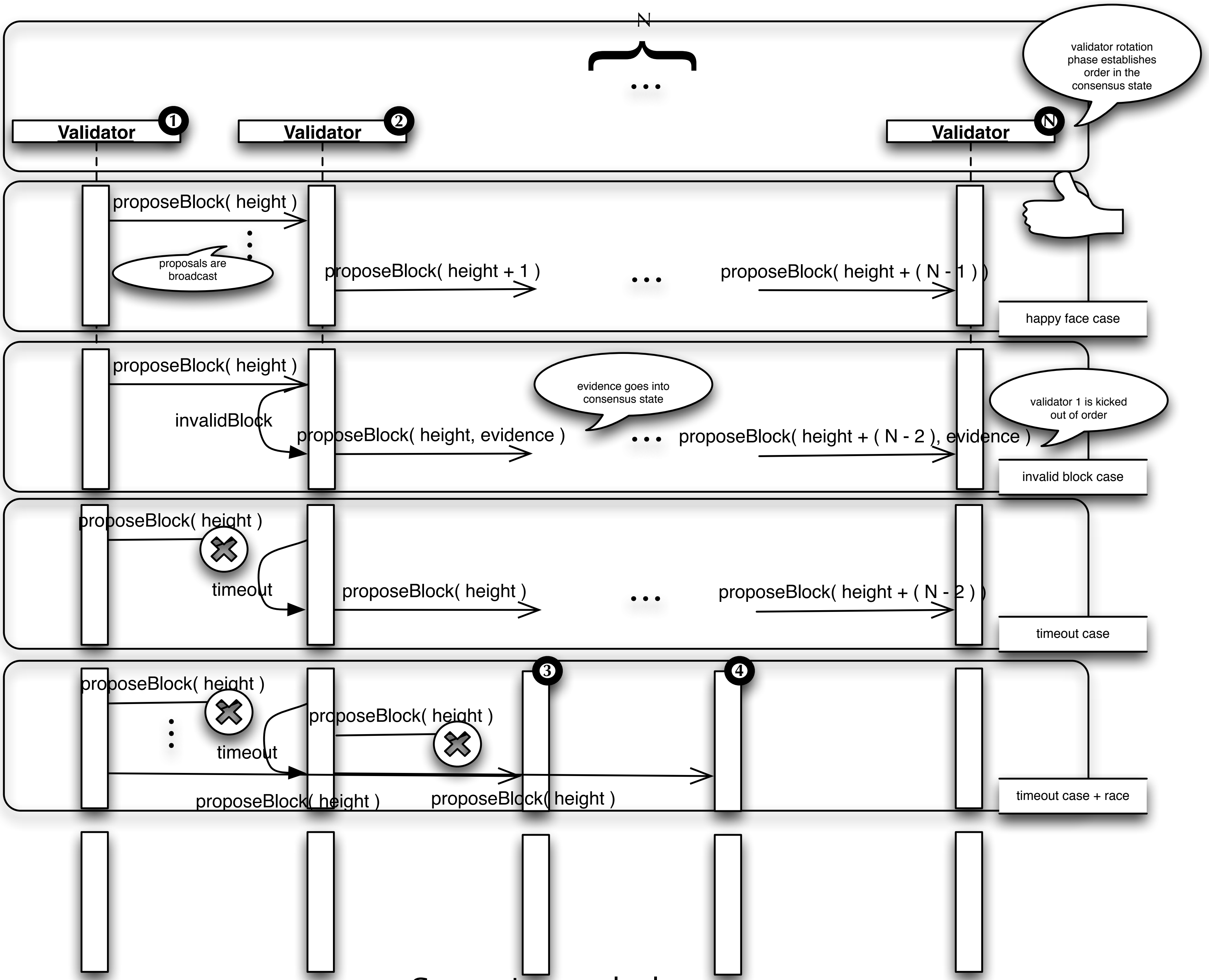
Casper in π-calculus



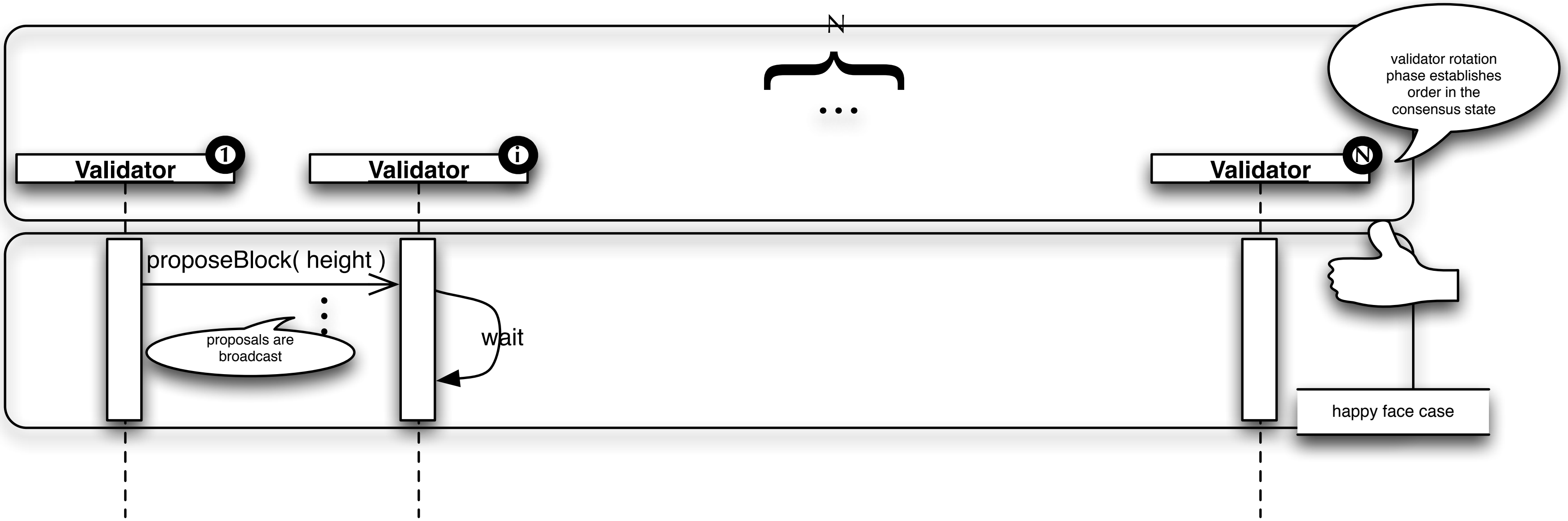
Casper in π-calculus

Consensus: Validator <-> Validator interactions





Casper in π-calculus

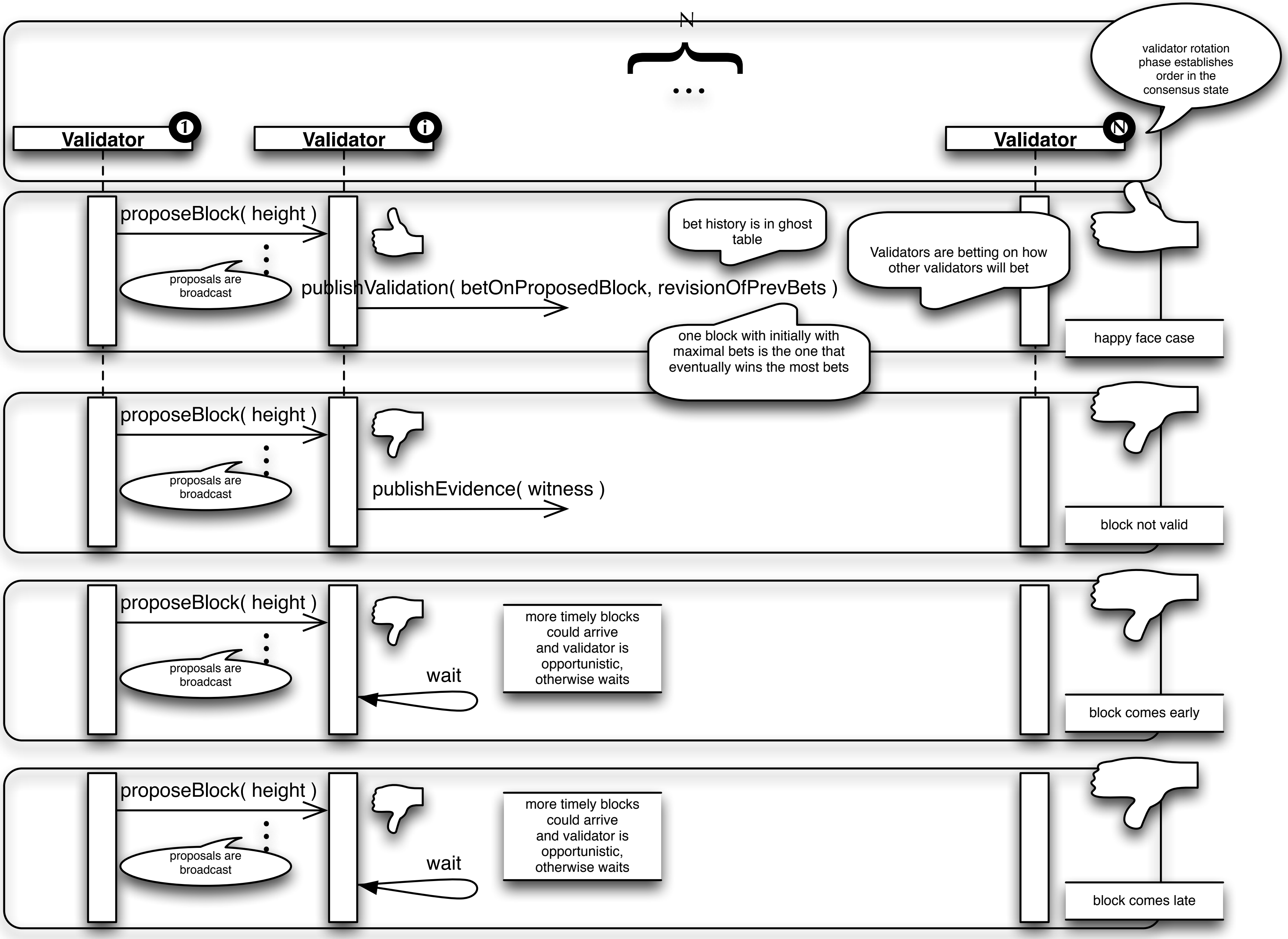


For validator i, v(i), if the proposal is from v(current), then v(i) can publish validation straightaway, otherwise wait for the proposer == v(current), then publish validation.

Blocks contain validations of other blocks.

Publishing a validation is placing a bet that the validated block is the block that will be finalized at that height.

Validations also contain new bets on non-finalized previous blocks.



Casper in π-calculus

To do list

State transition function

Bonding/unbonding

Rejection of validators based on evaluation of evidence

Distribution of transaction fees and bonds

Execution, re-ordering, and finalizing of txns

Assume fn, current : time -> Z[N] where N is the number of validators. At end / beginning of bonding period current is recalculated.

current = current_1 || current_2 || ... || current_M

to get parallelism speed up opportunities

To do list

Block consists of
* ghost entries (prev hash, next hash)
* reorg transitions
* new txns

```
valid( block ) ==
  stateX( block.ghostEntries._1 ) == block.ghostEntries._2
&& stateX( block.reorgX._1 ) == block.reorgX._2
```

Blocks and ghost table structure

block ::= (entry*, signature)

entry ::= ghost | reorg | txn

ghost ::= (prev, (block | validation | evidence), post)

evidence ::= (address, bet*)

reorg ::= (prev, txn*, post)

txn ::= (prev, (receiver, data, signature), post)

validation ::= ((hash(block), bet)*, signature)

state ::= (cmgrState, appState)

cmgrState ::= (ghostTable, validator*, evidenceChecker, txnFees)

Blocks and ghost table structure

appState ::= state of a deterministic automaton

history : ghostTable -> txn*

stateFn : (appState, txn) -> appState

appState
==
((initialState) \ history) ((txn, state) => stateFn(txn, state))

stateHash ::= hash((hash(cmgr), hash(appState))

prev ::= stateHash

post ::= stateHash

ghostTable ::= height -> (block ->(bet, validator)*)*

Blocks and ghost table structure

validator ::= (address, deposit)

evidenceChecker ::= (ghostTable, address) -> boolean
| evidence -> boolean

txnFees ::= height -> double

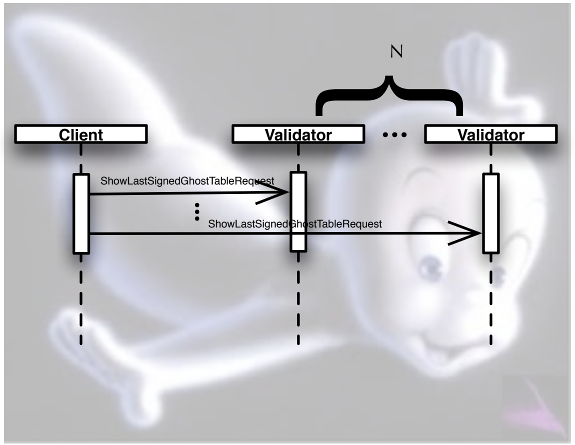
To do list

Assume fn, current : time -> Z[N] where N is the number of validators. At end / beginning of bonding period current is recalculated.

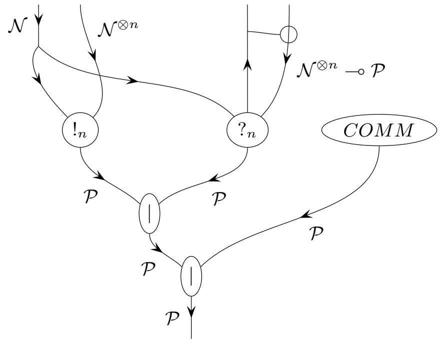
current = current_1 || current_2 || ... || current_M

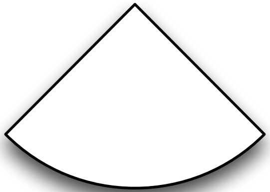
to get parallelism speed up opportunities

How to turn interaction diagrams into π-calculus specs



$\Rightarrow$





programming model

Claimant

Verifier

Relying party

allow Abed to check claim C

```
for(
    e1 <- chan?( C2V )( allowCheck( ) );
```

Verify Troy's claim C

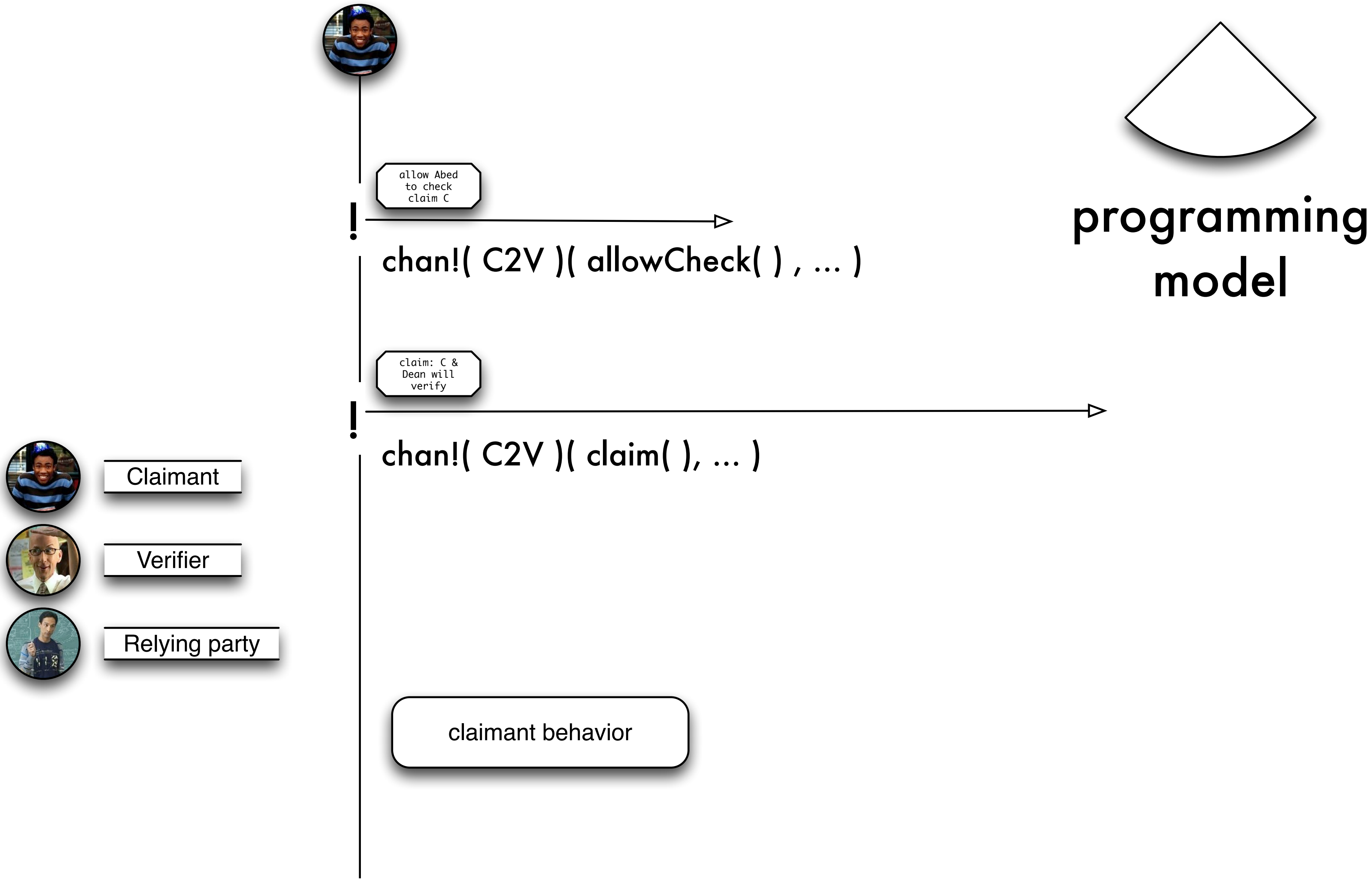
```
    e2 <- chan?( R2V )( verify( ) )
) {
```

confirm Troy's claim C

```
        chan!( V2R )( confirm( ) )
}
```

verifier behavior

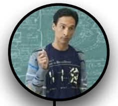
Just walk this line to calculate the specification of the verifier's behavior and derive code



and this line to calculate the specification of the claimant's behavior and derive code

Casper in π-calculus

programming model



claim: C & Dean will verify

```
for(
    e1 <- chan?( C2R )( claim( ) ) )
) {
```

Verify Troy's claim C

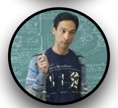
```
    chan!( C2R )( verify( ), ... )
```

confirm Troy's claim C

```
    for(
        e1 <- chan?( C2R )( trudat( ) ) )
    ) {
        ...
    }
}
```


Claimant


Verifier

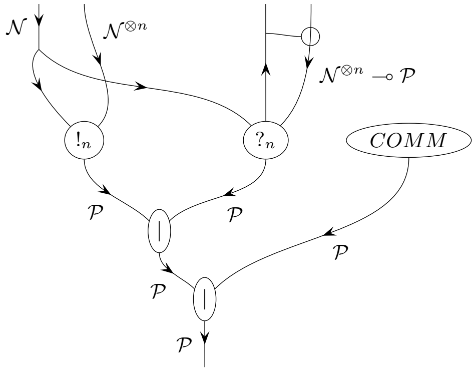

Relying party

relying party behavior

and this line to calculate the specification of the relying party's behavior and derive code

Casper in π-calculus

How to turn π-calculus specs into scala code



$\Rightarrow$ Scala

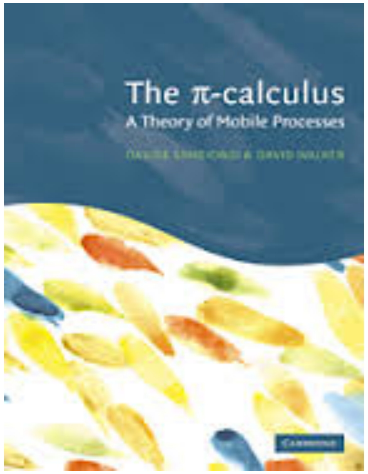
P,Q ::= 0	`{ }`
a![v1, …, vn]	`[| a |]( m ) ![ [| v1 |]( m ), …, [| vn |]( m ) ]`
a?(x1, …, xn)P	`for( [ x1, …, xn ] <- [| a |]( m ) ){ [| P |]( m )( x1, …, xn ) }`
P | Q	`spawn{ [| P |]( m ) };spawn{ [| Q |]( m ) }`
(new a)P	`{ val q = new Queue(); [| P |]( m[ a <- q ] ) }`
(def X(x1, …, xn) = P)[v1, …, vn]	`object X { def apply( x1, …, xn ) = { [| P |]( m ) ( x1, …, xn ) } }`
X[v1, …, vn]	`X( [| v1 |]( m ), …, [| vn |]( m ) )`

```
for( [ x1, …, xn ] <- [| a |]( m ) ){
    [| P |]( m )( x1, …, xn )
}
```

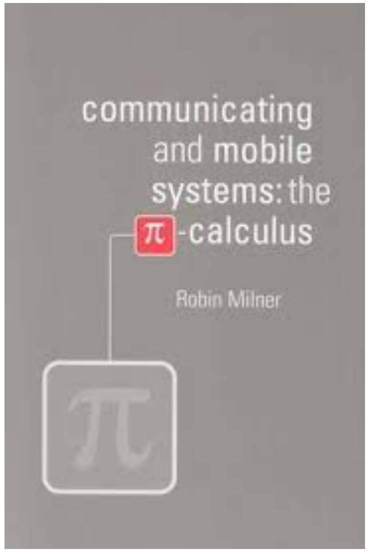
```
object X {
   def apply( x1, …, xn ) = {
      [| P |]( m ) ( x1, …, xn )
   }
}
```

`[| - |]( - ) : ( π-calculus, Map[Symbol,Queue] ) -> Scala`

Casper in π-calculus



applied π-calculus



How to specify a computational calculus

Modern presentations of computational calculi generalize generators and relations style presentations of universal algebra

They are typically given in terms of

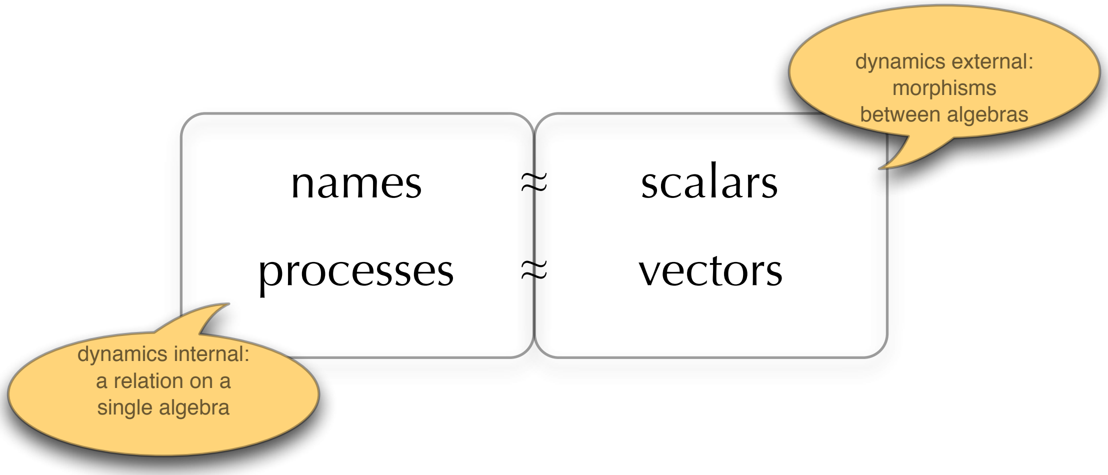
a grammar	≈	generators	≈	a free functor
a structural equivalence	≈	relations	≈	an algebra of a monad
an operational semantics	≈	no analog	≈	2-cell
computer science		universal algebra		category theory

bi-directional rewrites

uni-directional rewrites

How to specify a computational calculus

Process calculi, like vector spaces, are two sorted algebraic structures



A morphism from one process calculus to another ***preserves*** computational dynamics

A morphism from one vector space to another ***is*** computational dynamics

Syntax

```
P,Q  ::=  0

          a![ t ]Q

          a?( t )P

           P | Q

          (new a)P

          ( def X( t ) = P )[ u ]

           X[ t ]
```

```
t,u   ::=   val | var |  fn( t1, ..., tN )

val   ::=   bool  |  int  |  string  |  double | ...

var   ::=   _   |  X | Y | Z | ...

fn    ::=  x | y | z | ...
```

Structural congruence

The *structural congruence* of processes, noted $\equiv$, is the least congruence containing α-equivalence, $\equiv_\alpha$, making $(P, |, 0)$ into commutative monoids and satisfying

$$(\mathsf{new}\ x)(\mathsf{new}\ x)P \equiv (\mathsf{new}\ x)P$$

$$(\mathsf{new}\ x)(\mathsf{new}\ y)P \equiv (\mathsf{new}\ y)(\mathsf{new}\ x)P$$

$$((\mathsf{new}\ x)P) \mid Q \equiv (\mathsf{new}\ x)(P \mid Q)$$

Reduction rules

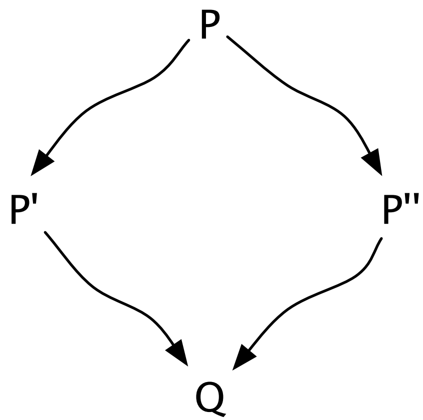
$$\frac{\text{unify(t, u, s)}}{\text{a?(t)P | a![u]Q -> Ps | Qs}}$$

$$\frac{\text{P -> P'}}{\text{P | Q -> P' | Q}}$$

$$\frac{\text{P -> P'}}{\text{(new a)P -> (new a)P'}}$$

$$\frac{P \equiv P' \quad \text{P' -> Q'} \quad Q' \equiv Q}{\text{P -> Q}}$$

Important properties of computational calculi



Confluence

lambda calculus is confluent

π–calculus is **not** confluent

a![u1]Q1 | a?(t)P | a![u2]Q2

typical race condition

Casper in π–calculus