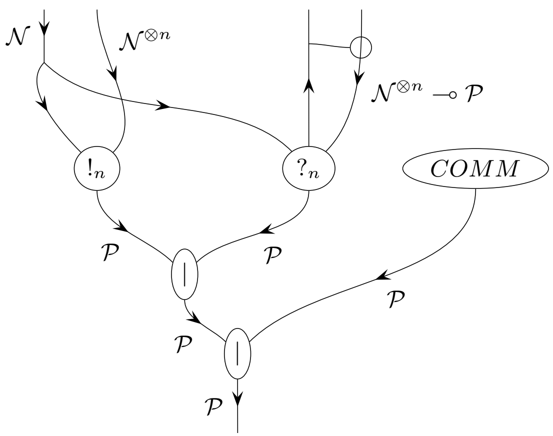
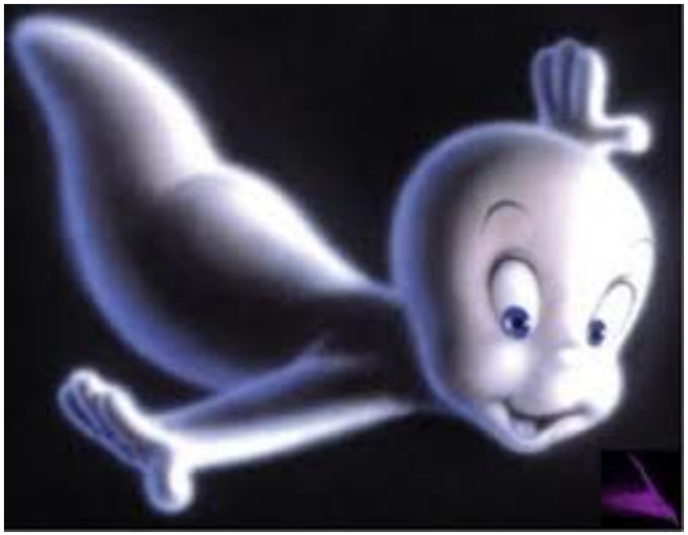
Casper Formalized Pt I

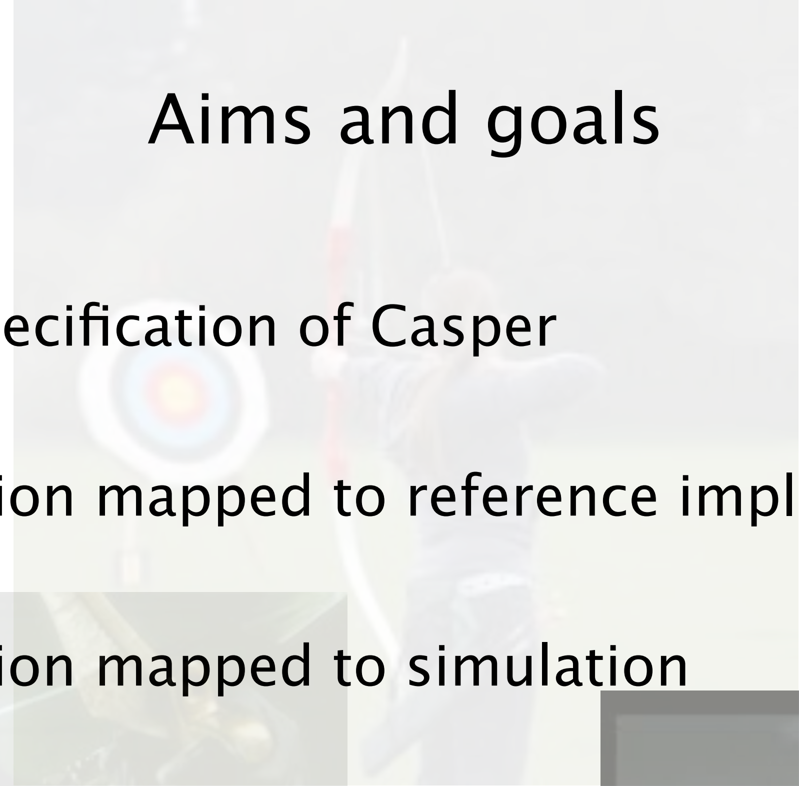
Applying π-calculus to modeling the Casper protocol





Vlad Zamfir, L.G. Meredith

Aims and goals

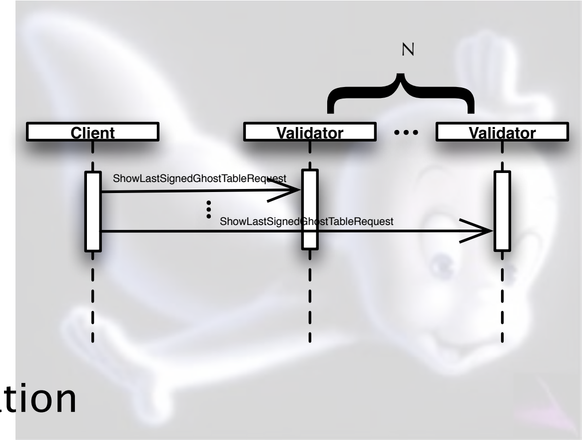


Formal specification of Casper

Specification mapped to reference implementation

Specification mapped to simulation

Methodology



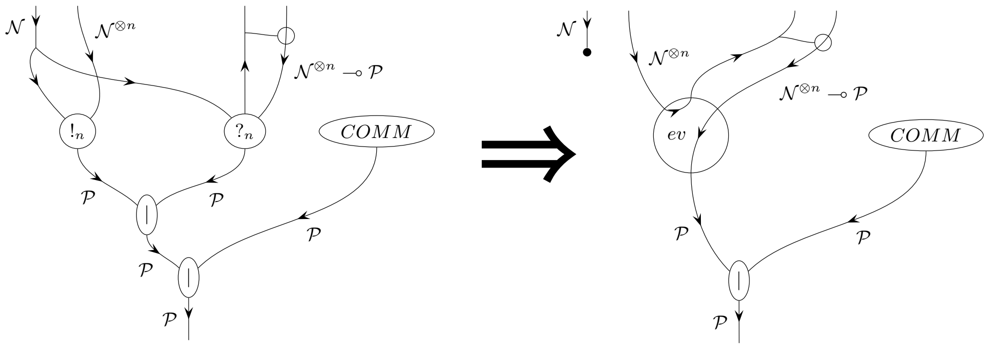
Bootstrap Casper spec using interaction diagrams

Map interaction diagrams to initial π-calculus specification

Refine specification

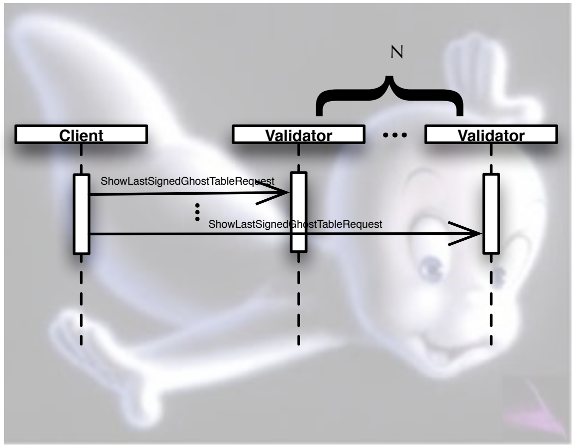
Map refined specification to SpecialK

Map refined specification to Stochastic π-machine

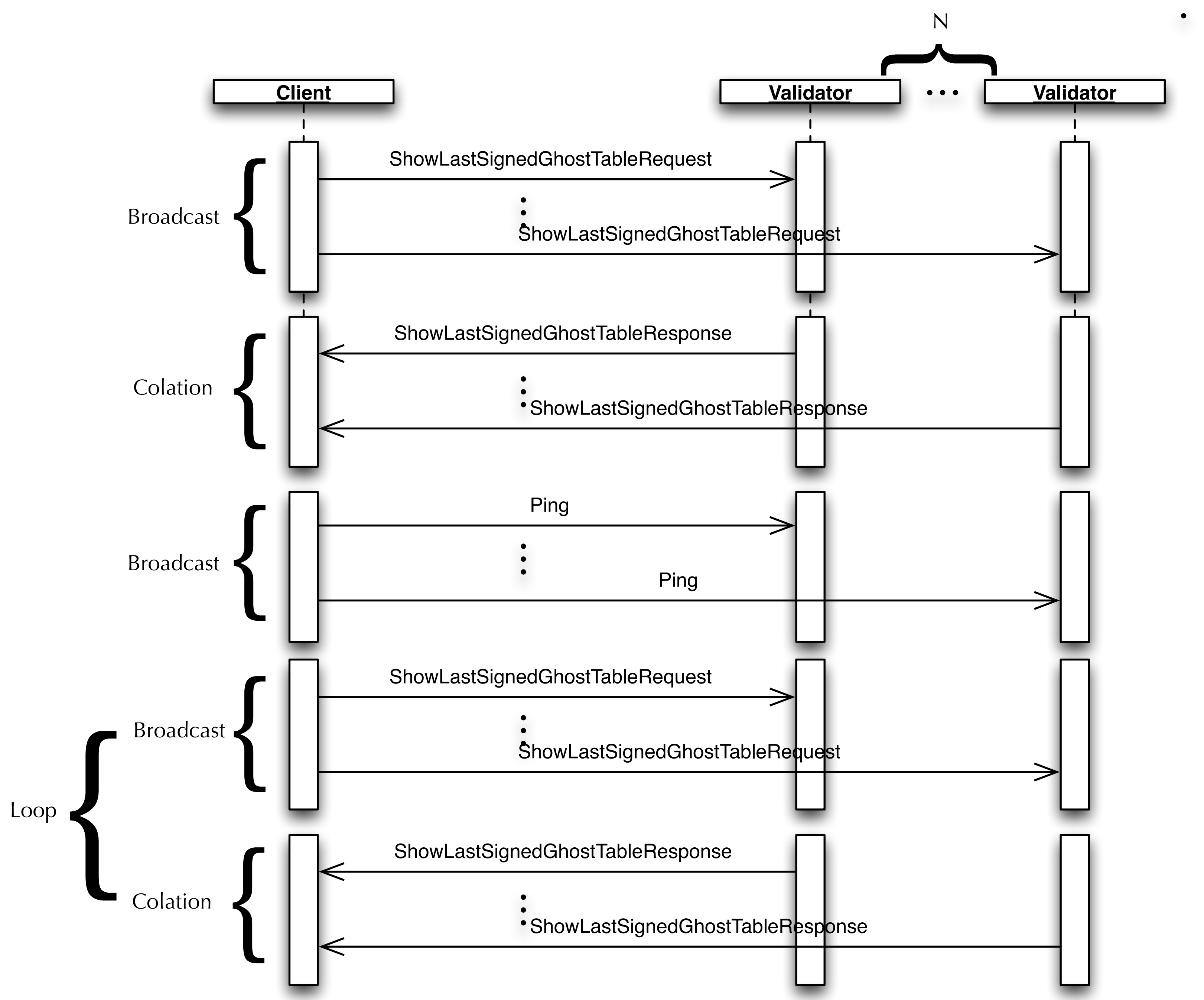


Casper in π-calculus

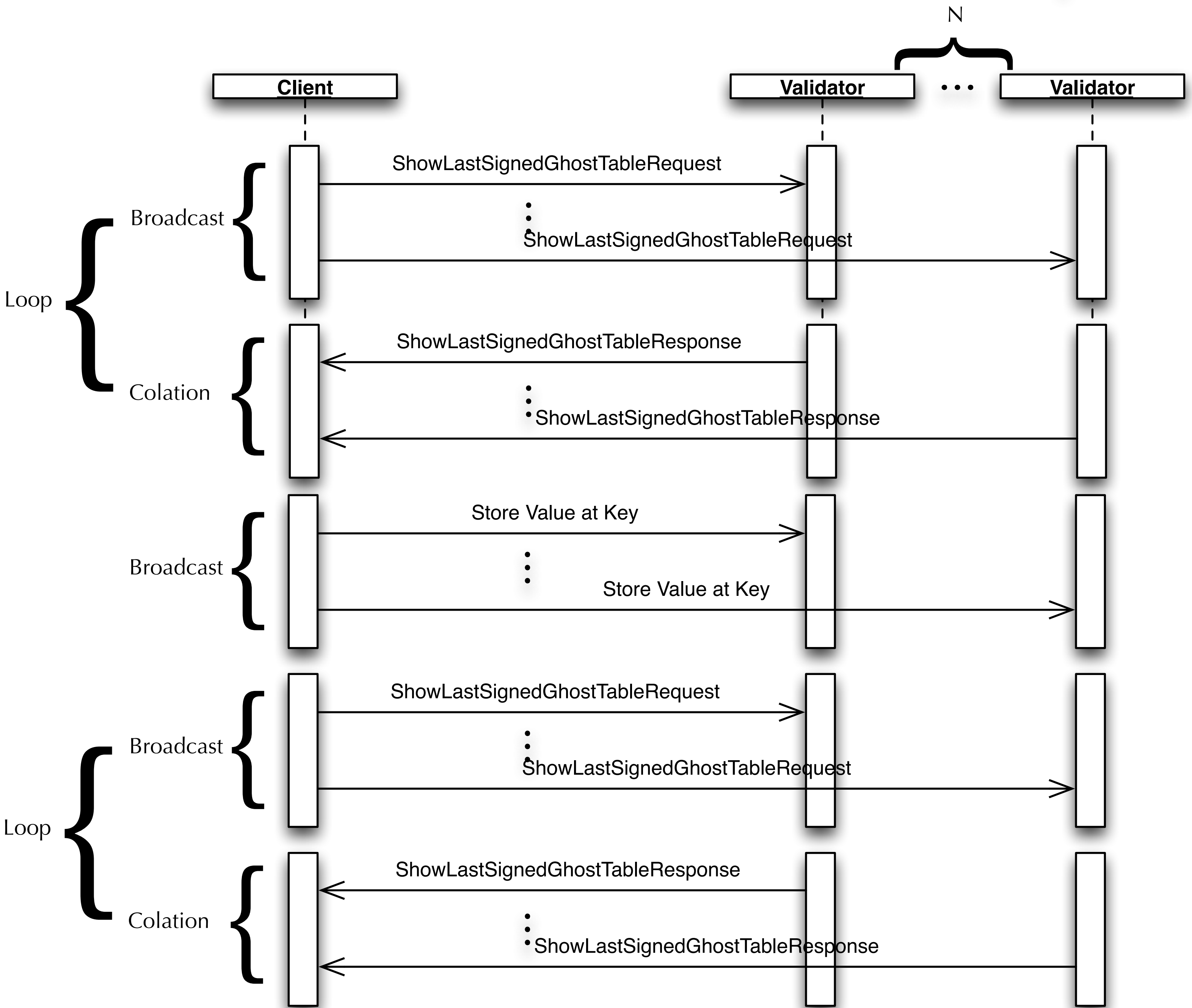
Casper in interaction diagrams



Client <-> Validator interactions

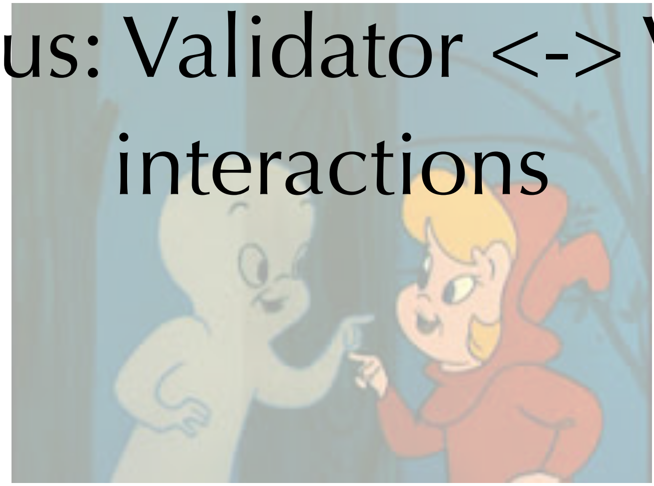


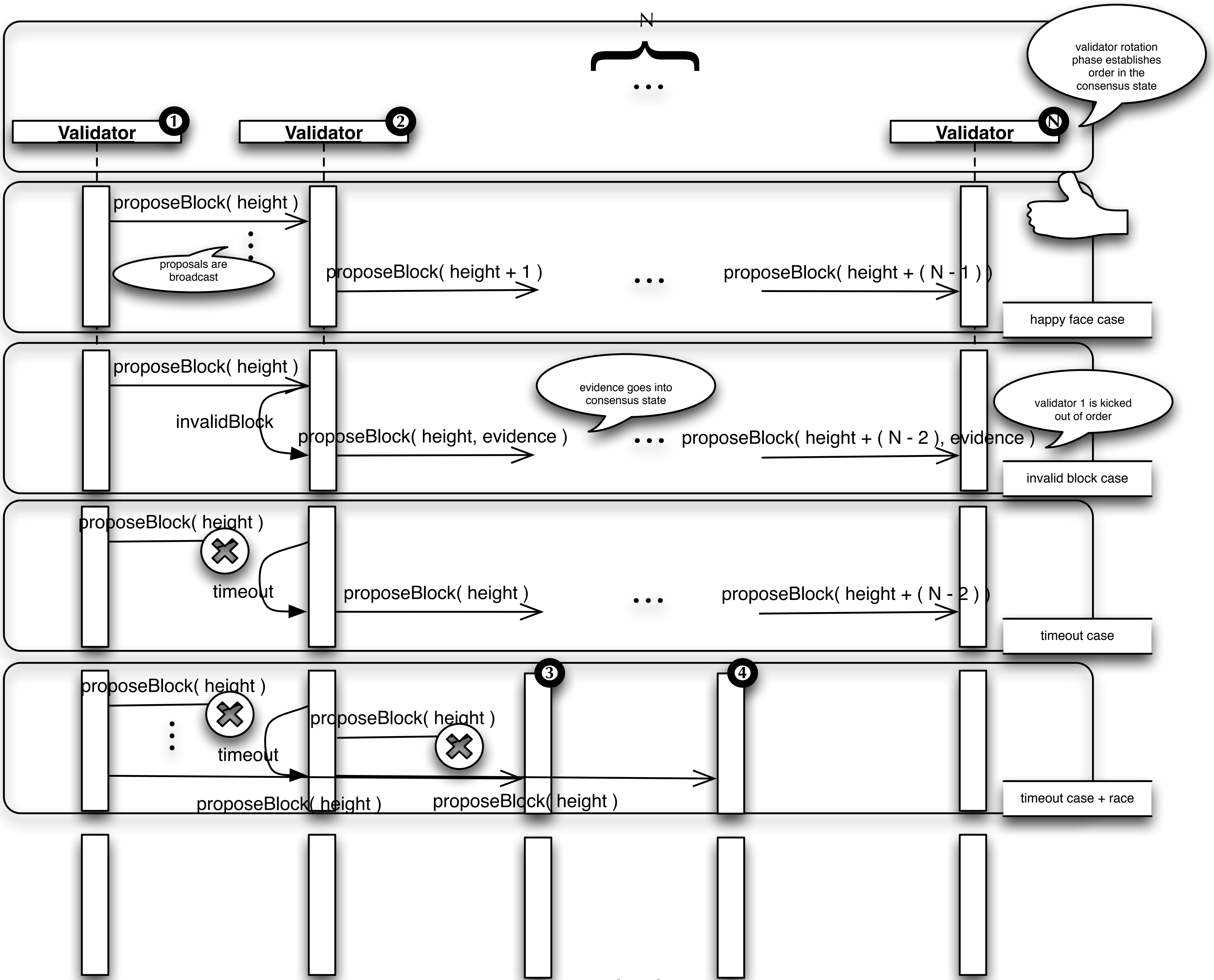
Casper in π-calculus



Casper in π-calculus

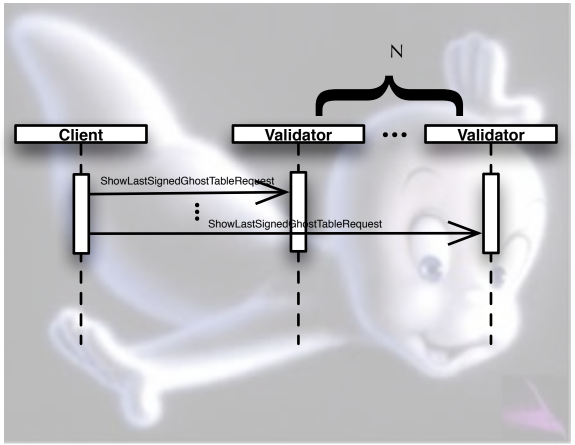
Consensus: Validator <-> Validator interactions



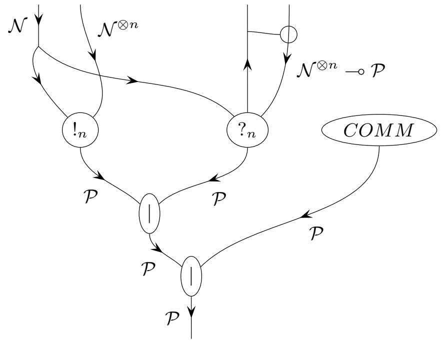


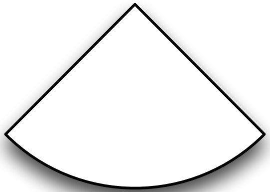
Casper in π-calculus

How to turn interaction diagrams into π-calculus specs



$\Rightarrow$





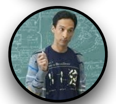
programming model

```
for(
    e1 <- chan?( C2V )( allowCheck( ) ) );
    e2 <- chan?( R2V )( verify( ) )
) {
        chan!( V2R )( confirm( ) )
}
```

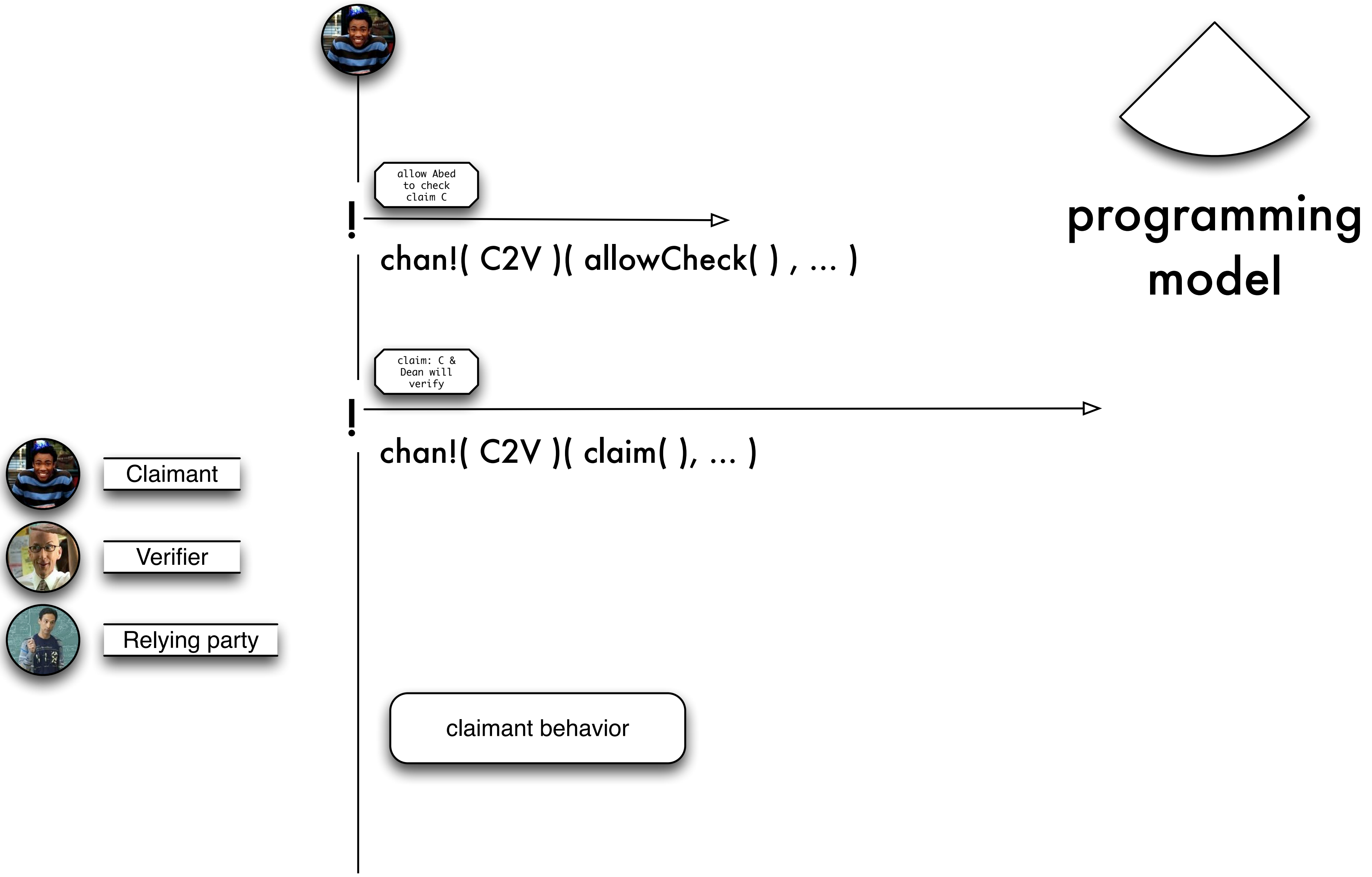

verifier behavior

 Claimant

 Verifier

 Relying party

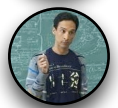
Just walk this line to calculate the specification of the verifier's behavior and derive code



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

Casper in π-calculus

programming model




```
for(
    e1 <- chan?( C2R )( claim( ) ) )
) {
    chan!( C2R )( verify( ), ... )
    for(
        e1 <- chan?( C2R )( trudat( ) ) )
    ) {
        ...
    }
}
```

claim: C & Dean will verify

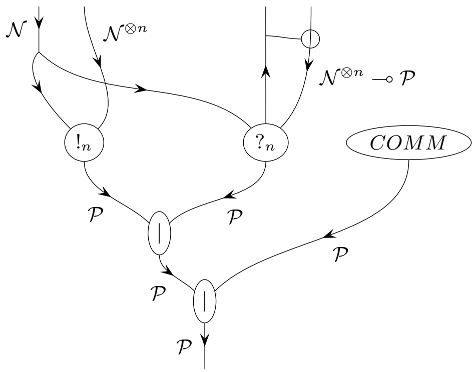
Verify Troy's claim C

confirm Troy's claim C

relying party behavior

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

How to turn π-calculus specs into scala code

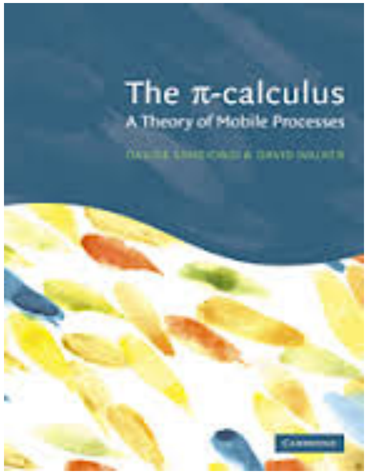


$\Rightarrow$ Scala

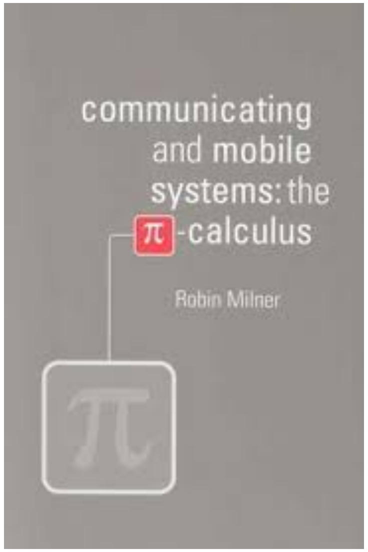
P,Q ::= 0	`{ }`
a![v1, …, vn]	`[| a |]( m ) ![ [| v1 |]( m ), …, [| vn |]( m ) ]`
a?(x1, …, xn)P	`for( [ x1, …, xn ] <- [| a |]( m ) ){ [| P |]( m )( x1, …, xn ) }`
P | Q	`spawn{ [| P |]( m ) };spawn{ [| Q |]( m ) }`
(new a)P	`{ val q = new Queue(); [| P |]( m[ a <- q ] ) }`
(def X(x1, …, xn) = P)[v1,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

```
[| - |]( - ) : ( 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

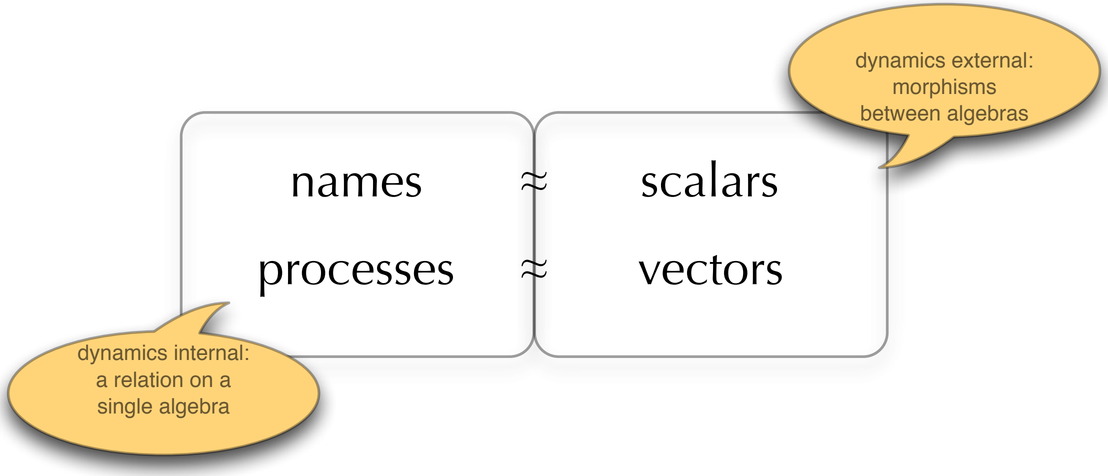
computer science		universal algebra		category theory
a grammar	≈	generators	≈	a free functor
a structural equivalence	≈	relations	≈	an algebra of a monad
an operational semantics	≈	no analog	≈	2-cell

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

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

$$\frac{P \;\text{->}\; P'}{P \mid Q \;\text{->}\; P' \mid Q}$$

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

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

Important properties of computational calculi

$$
\begin{tikzcd}
 & P \arrow[dl] \arrow[dr] & \\
P' \arrow[dr] & & P'' \arrow[dl] \\
 & Q &
\end{tikzcd}
$$

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