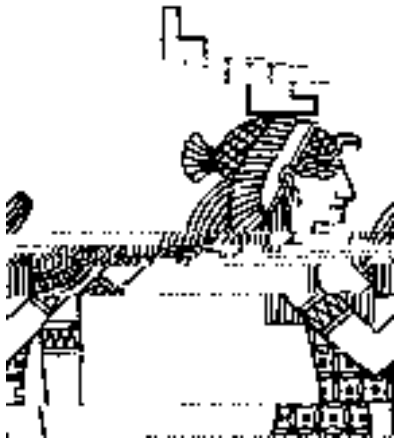
H o r

Jason Hickey
Cornell University
Based on work by: Mark Hayden,
Karl Crary, Robbert van Renesse,
Brad Glade, and others

What is Horus?

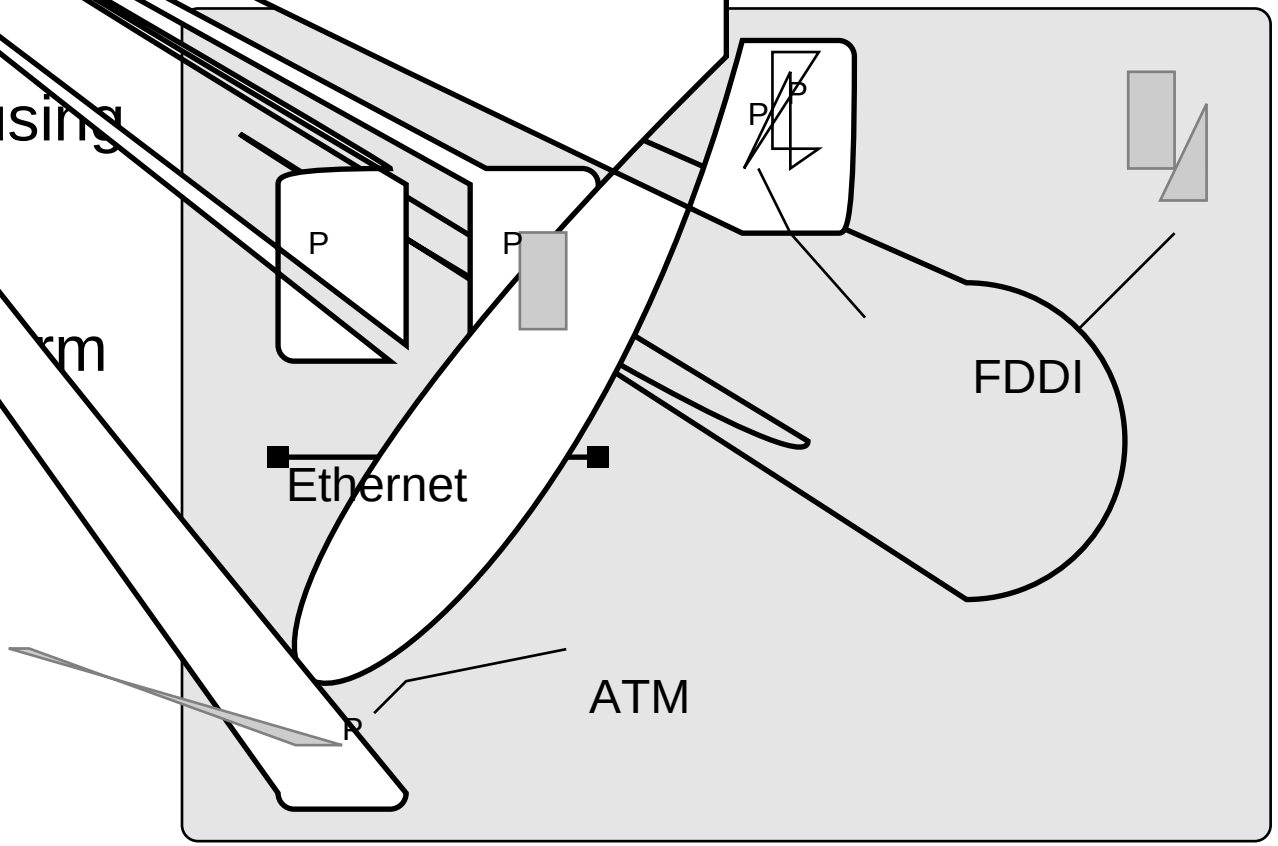


- A framework/toolkit for building distributed programs
  - High performance
  - Fault tolerant
  - Secure
- Very popular
- Ported to several platforms
  - Unix, Mach, NT

  Product code is written in C
  - expressive
  - efficient

e/

- Dis buted proc ses using multica
- Processes rm



Horus process model

Applications get services by pushing

Goals

- Want td_ake sure key services act as advertised
- Properties include both functionality and security
- Verify properties incrementally (a.k.a. “hardening”)
-

Example: Virtual synchrony

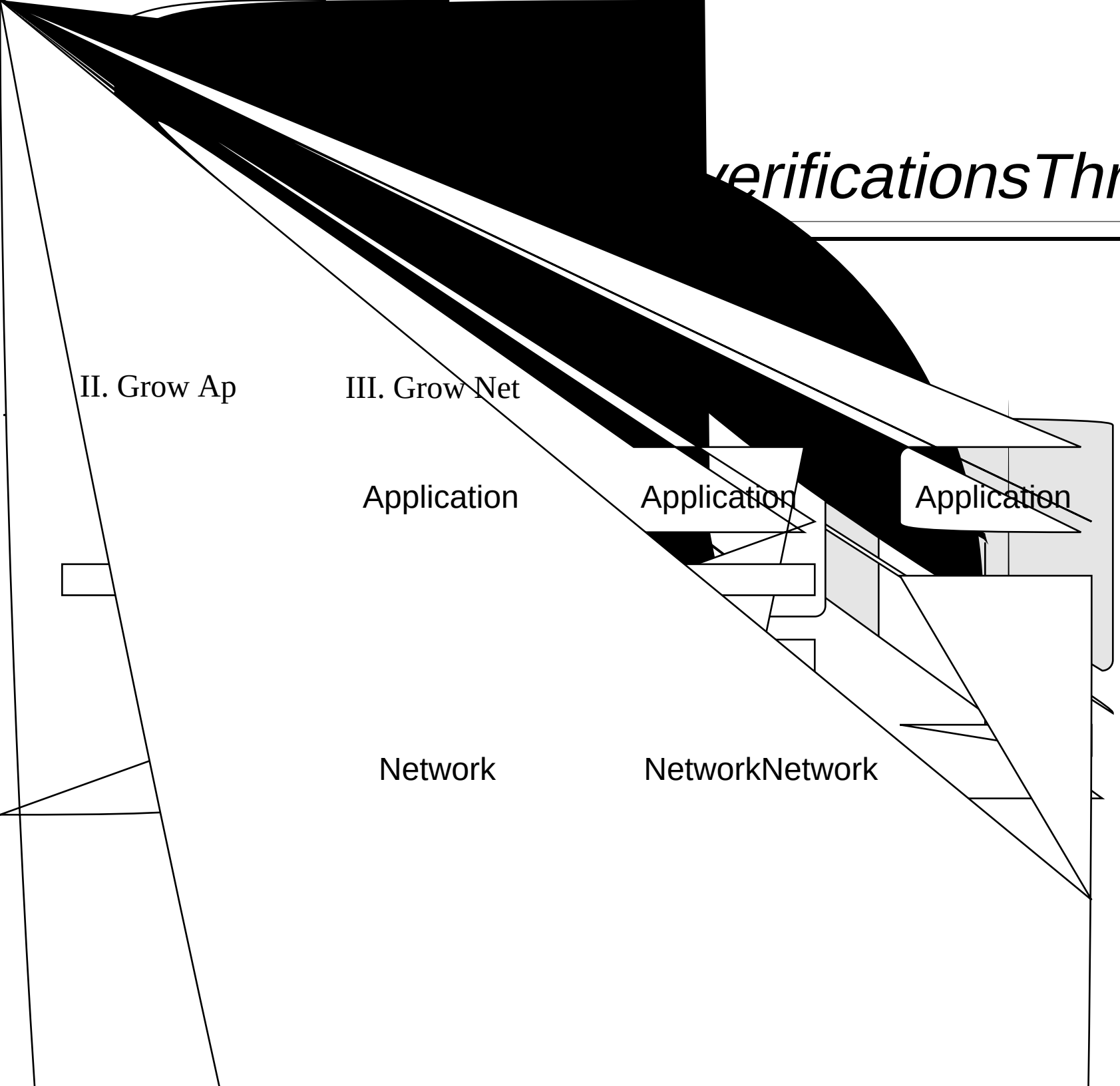
- Manages group message ordering
- Messages can't arrive from dead processes

- Can’t specify or validate all properties
  - Some properties are “obvious”
  - Validate the key features
  - Assume basic properties
    - Virtual Synchrony

Simplify Horus layer model

- Verify one layer at a time

verificationsThree choices



Formalism

- What is a behavior of the system?
- Choices:

  Try to come up with a model that makes the entire system a function call

  - Use type theory to describe properties of the function
  - A naive translation is not feasible
  - But—any behavior will eventually be functional

Formalism proposal

- No need for an imperative logic
  - Each layer is independent, opaque
  - Behavior depends only on the messages sent & received

  A system **behavior** is a sequence of events

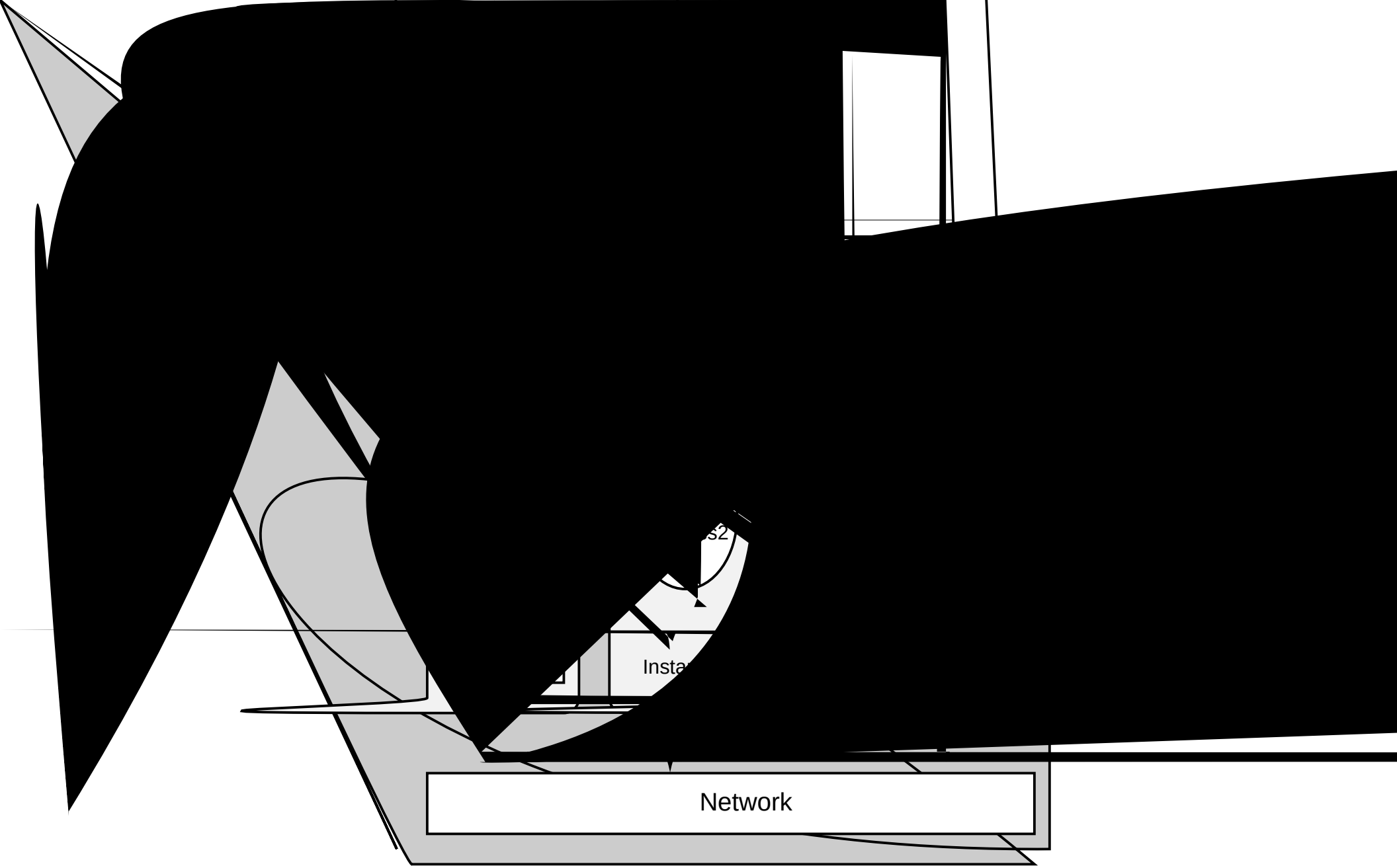
  Every event is a message transmission or reception (redundant)

  Message $\times$ MessageType $\times$ Wire#

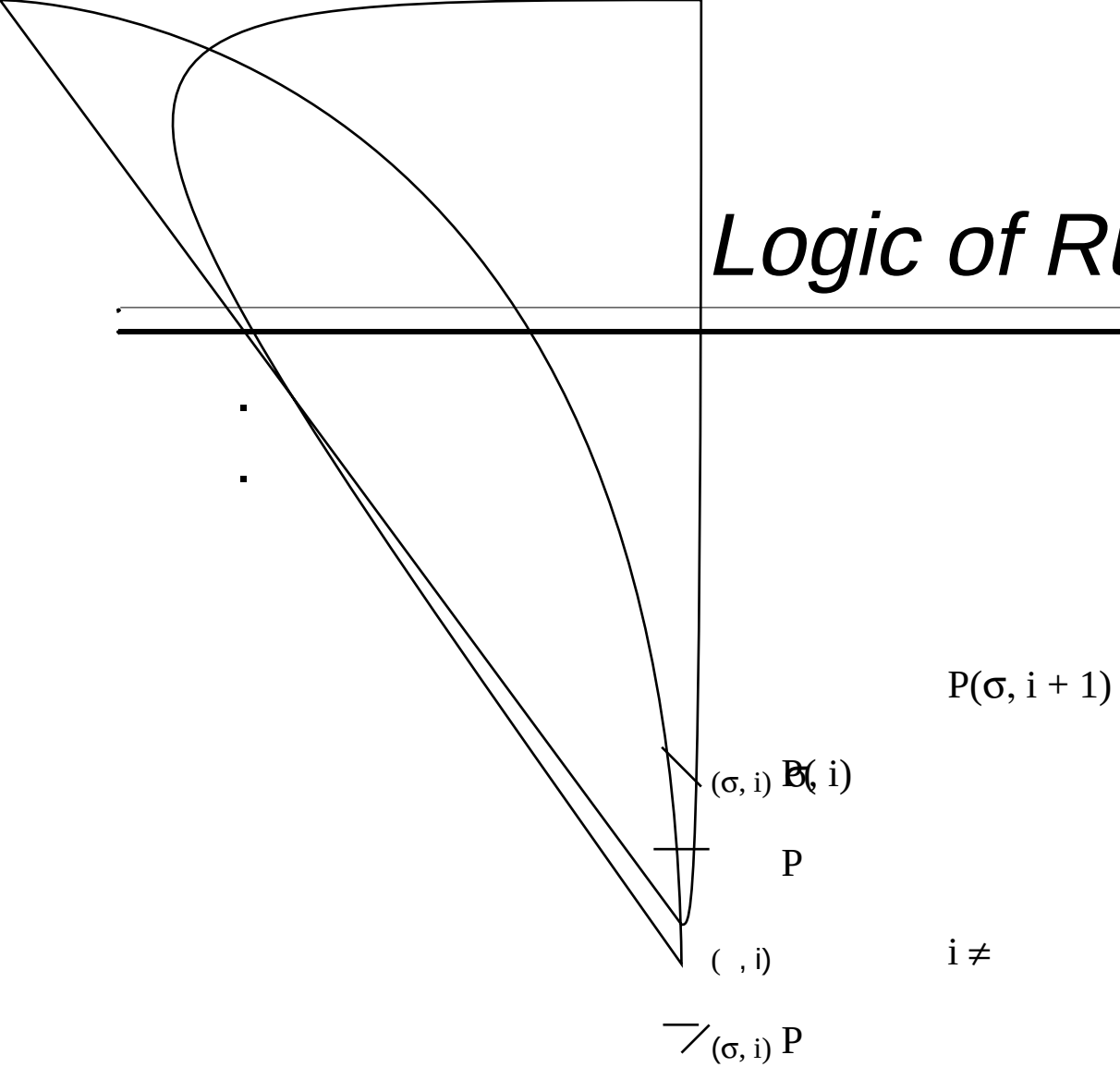
  A ___ i rinite sequence of events, and a curre G time

  A _____ is a predicate on runs

  - (N ® Event) $\times$ N
  - Eve G *stream*


Network

Logic of Runs

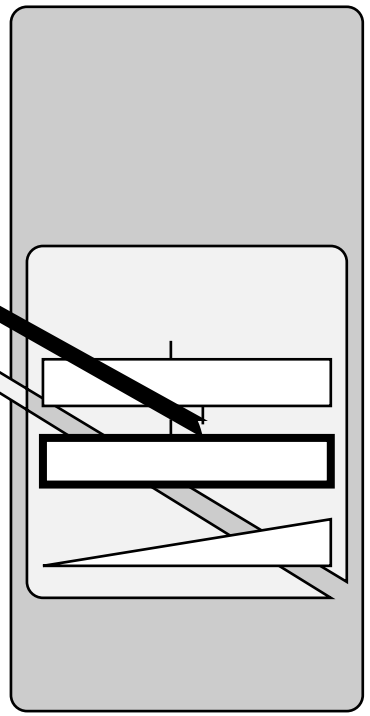


Temporal Logic of Actions

Additional predicates

Operator	Coding
$\text{Enabled}_{\langle s, i \rangle}$	

- Primary goal: virtual synchrony
- Stability layer records who has
- received messages
  Once every process in a group has received a message, it no
- Keep a matrix of message receptions



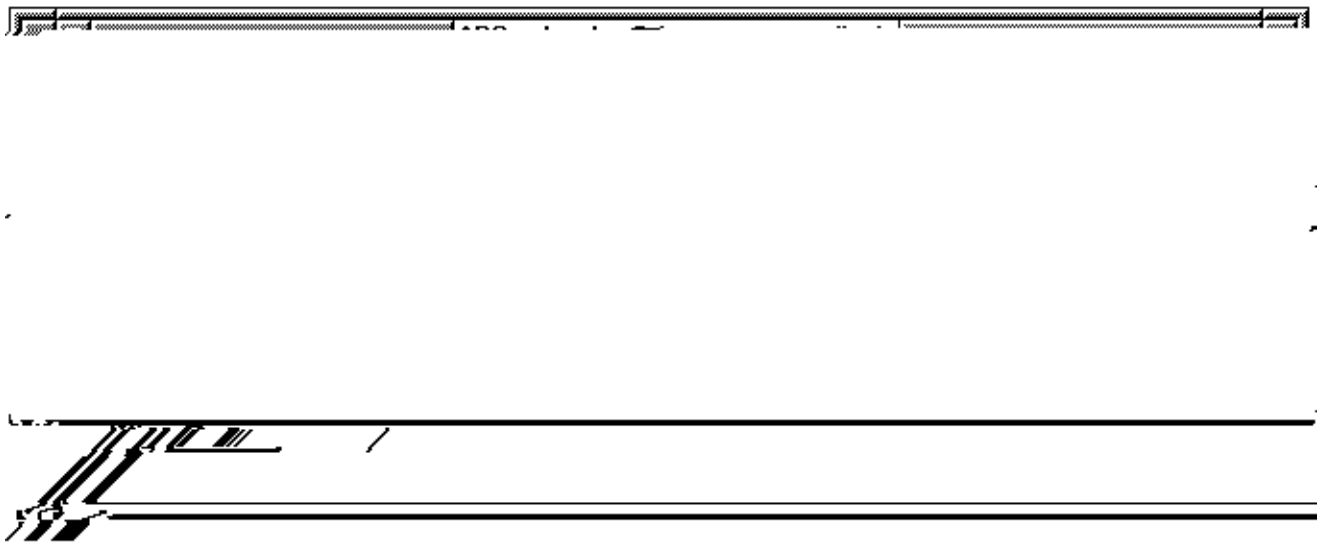
  - are received in order

```
let flushing  = ref true
and nmembers  = ref 0
and my_rank   = ref 0
and ncasts    = ref [||] (* vect (rank) *)
and acks      = ref [||] (* matrix (from,to) *)
and stable    = ref t)|] (* vect (rank) *)
in

(* Incorporate new acks from myself tor from other member *)
let got_acks from new =
  let improved = ref false in
  (* For each mem update ack entry and find new stability *)
  for i = 0 to !nmembers - 1 do
    !acks.(i).(from) <- max (!acks.(i).(from)) (new.(i)) ;
    new.(i) <- !acks.(i).(from);   (* bad orderingd n[

    for j = 0 to !nmembers -1 do
      new.(i) <- min (new.(i)) (!acks.(i).(j))
    done ;

    if tnew.(i) > !stable.(i) then begin
     (* stability has changed *[
     improved := true ;
     !stable.(i) <- new.(i)
    end
  done ;

  (* If tstability improved, deliver new stability event *)
  if t!improved then begin
    upnm (upDef name UpStable[upExtend (StableVect (copy_vect !stable))])
  end
```

Stability points

- Program is imperative
  - Doesn’t have to be, but it emulates with real code
  - Relatively high level imperative operations
- Can simulate with a denotational coding

Nuprl version of stability



Imperative program constructs

A program is a function State $\rightarrow$ Value

State has a fixed numberables (for)

State v:STJ teVars TypeOf(v)

STateVar {0...sizeof(STate)}

"hidden" application

Variable

improved λstate.state $\equiv$ state.state 5

Expressions compute a value fr the sTJıte

Addition: i + j state.state i + state j

Assignment: x y state. if n = x (i.i) then y state else state n,

Sequence: s1; s2 state. s2 ((s1 state).1)

Program types

- Automatically

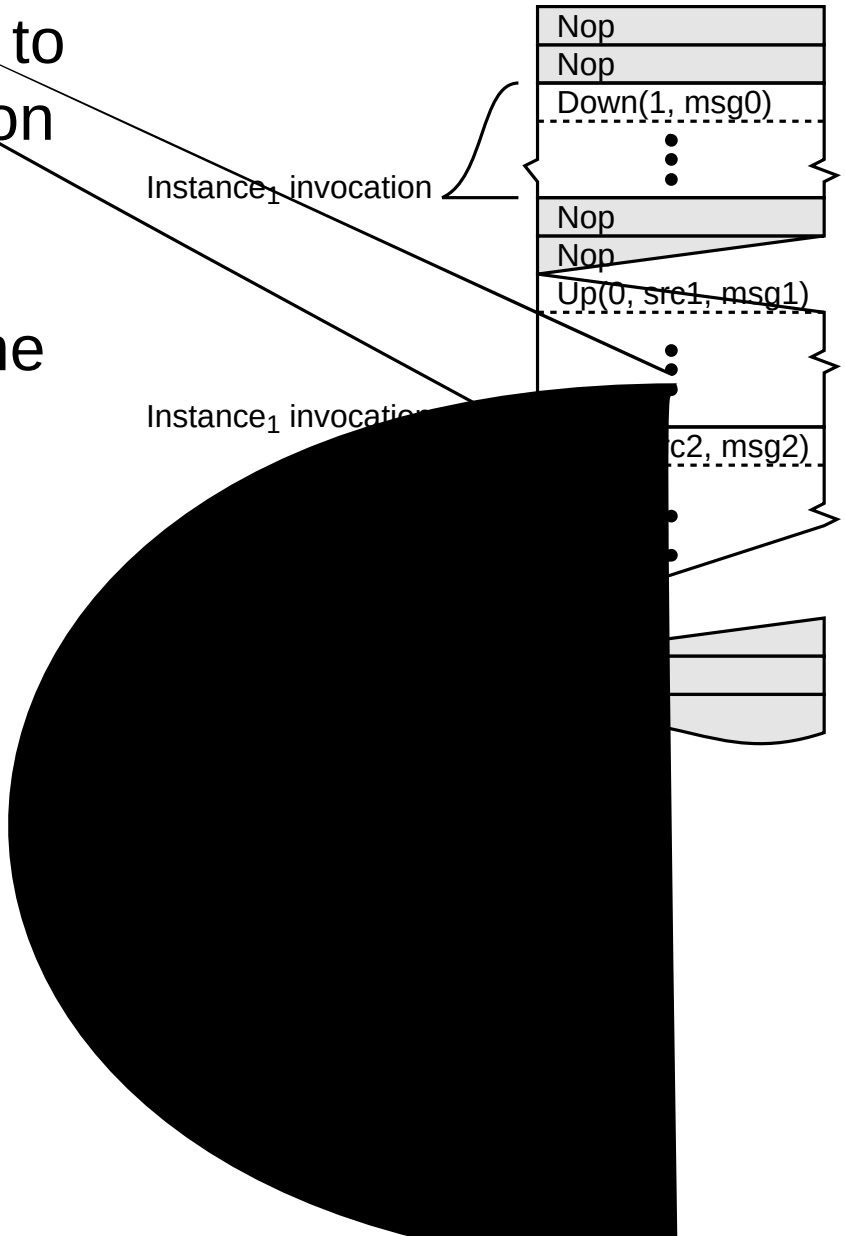
  Enumerations, Unions,

Pattern matching

- Generate patterns
- Reductions
- case construct

How do we verify the programs

- Restrict the possible Runs to get a decidable predicate on possible behaviors
- Immediately follow each reception by a layer with the messages it generates
- Add a Nop to get infinite histories



Difficult to verify layers without including their states in the history

New Run: $(N \in \text{Event} \times \text{State}^n) \times N$

Valid Histories *H* are the runs $(\sigma, i) \in \mathit{Run}$ where:

The states of $\sigma(0)$ are the initial states of the layer instances

for each j where $j \neq 0$, if message m of $\sigma(j)$ is a reception by instance k:

if $j \neq 0$, the states of $\sigma(j)$ are the states of $\sigma(j - 1)$

if σ_k is the state of layer instance k, and the program L, begun in state σ_k computes messages $m_0, \ldots, m_{k-1}$ and a new state σ

Verification predicate

Given layer implementation L:(Event $\times$ State) $\rightarrow$ (Event list

P (R) specifies the user

P_N(R) specifies the network

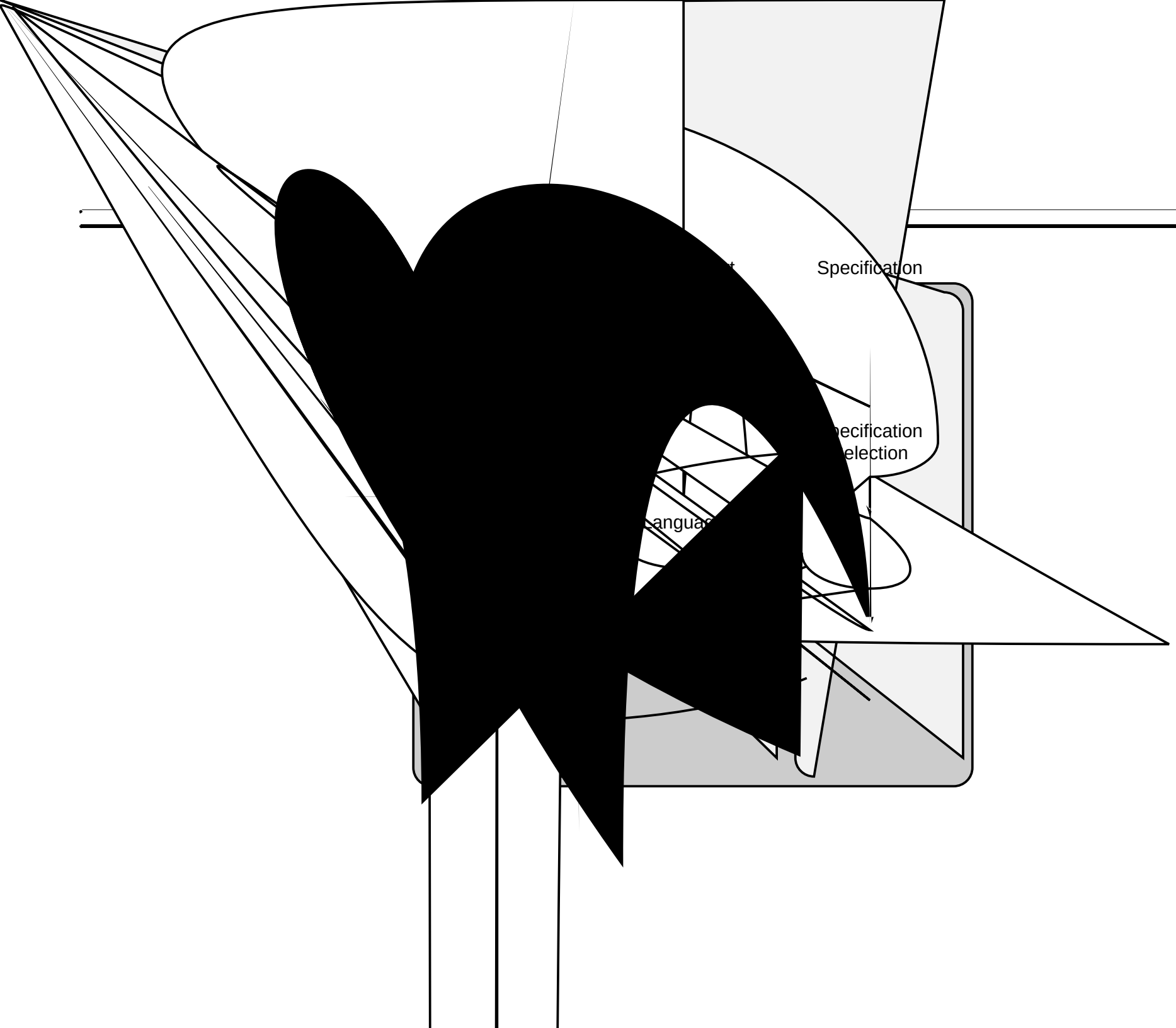
$P_{L(R, i)}$ specifies the layer

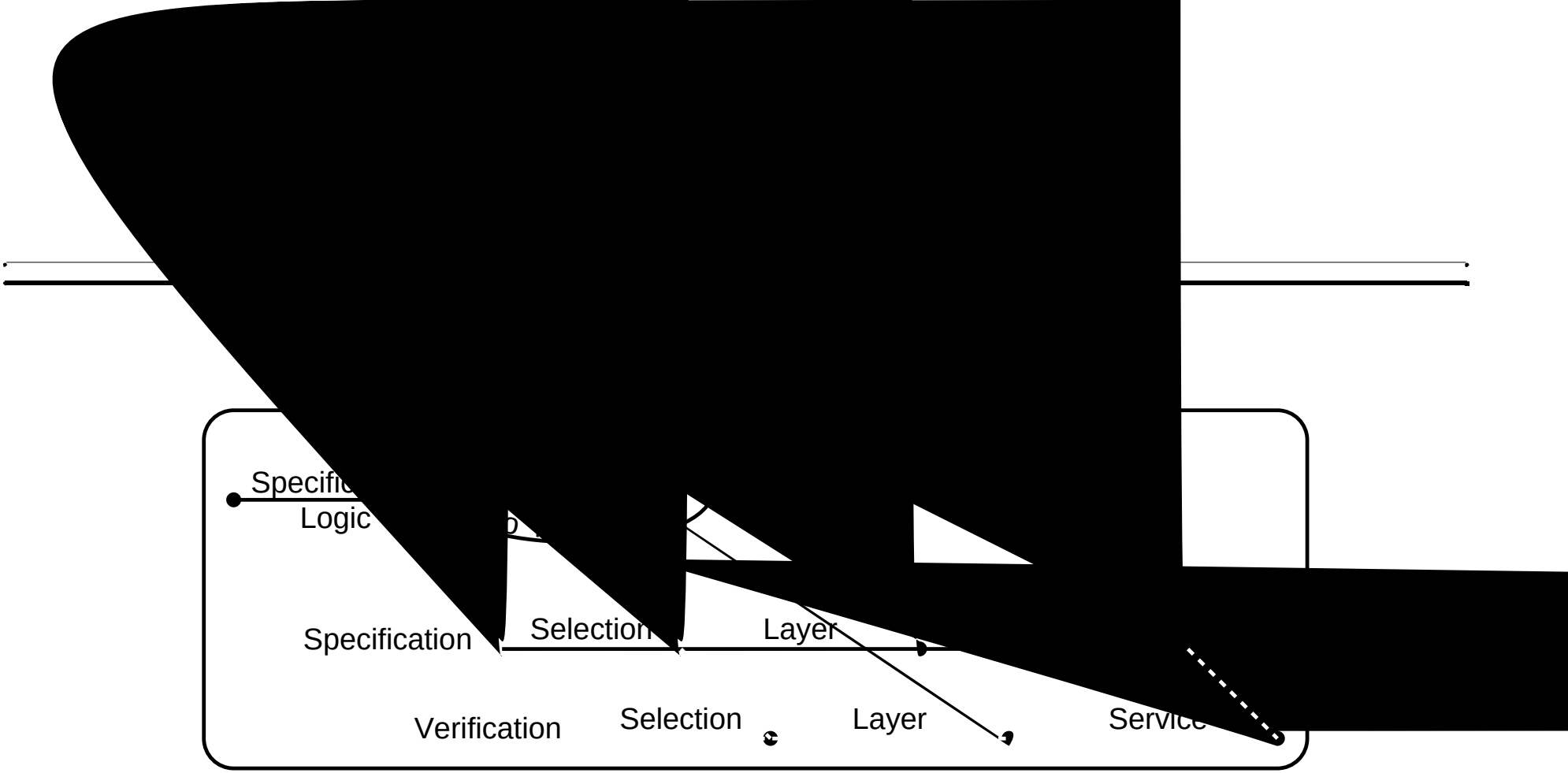
Spec is the desired property of the layer

$\forall n{:}N.\forall R{:}Run$

Example thm

- *TasksA temporal logic of Ru*
- *Provide support for a formal*
- *Provide support for Horus da*
- *Specify the behavior of the us*
- *Specify the services of each la*
- *Specify the properties of the in*
- *Prove critical properties in the t*


Specification


Specific
Logic
Specification
Selection
Layer
Verification
Selection
Layer
Service