Composing Software Systems that are Provably Correct

Ilya Sergey

ilyasergey.net


ECOOP 2008
Paphos, Cyprus

(this picture is actually from ECOOP 2009)

(this picture is from ECOOP 2008)

Research in Programming Languages

- Object-oriented software development
- Models and Modeling
- Language Design
- Parallelism
- Program Logics
- Applications (systems, networking, AI/ML)
- Analysis of Concurrent Programs
- Object-Oriented Programming
- Correctness
- Verification
- Type Systems
- Program Analysis
- Components and APIs
- Garbage Collection
- Array Processing
- Semantics of concurrent programs
- Low-level compiler optimisations
- Parsing
- Resource management
- Compiler optimisations

"Tracking the Flow of Ideas through the Programming Languages Literature", *Greenberg et al.*, SNAPL 2015

Language Design — how to encode *reusable* abstractions

Applications (systems, *etc.*) — how to *combine* them into systems

Correctness — how to *scale* the verification efforts

Language Design — how to encode *reusable* abstractions

Applications (systems, *etc.*) — how to *combine* them into systems

Correctness — how to *scale* the verification efforts

Composition

Compositional Software Verification

Compositional Software Verification

Tomorrow!

Rethinking Compositionality: Composing Proofs From Program Behaviours

Keynote

Track ECOOP 2019 ECOOP Research Papers

When Thu 18 Jul 2019 09:00 - 10:00 at Mancy - Keynote Chair(s): Sophia Drossopoulou

Session Program

Thu 18 Jul

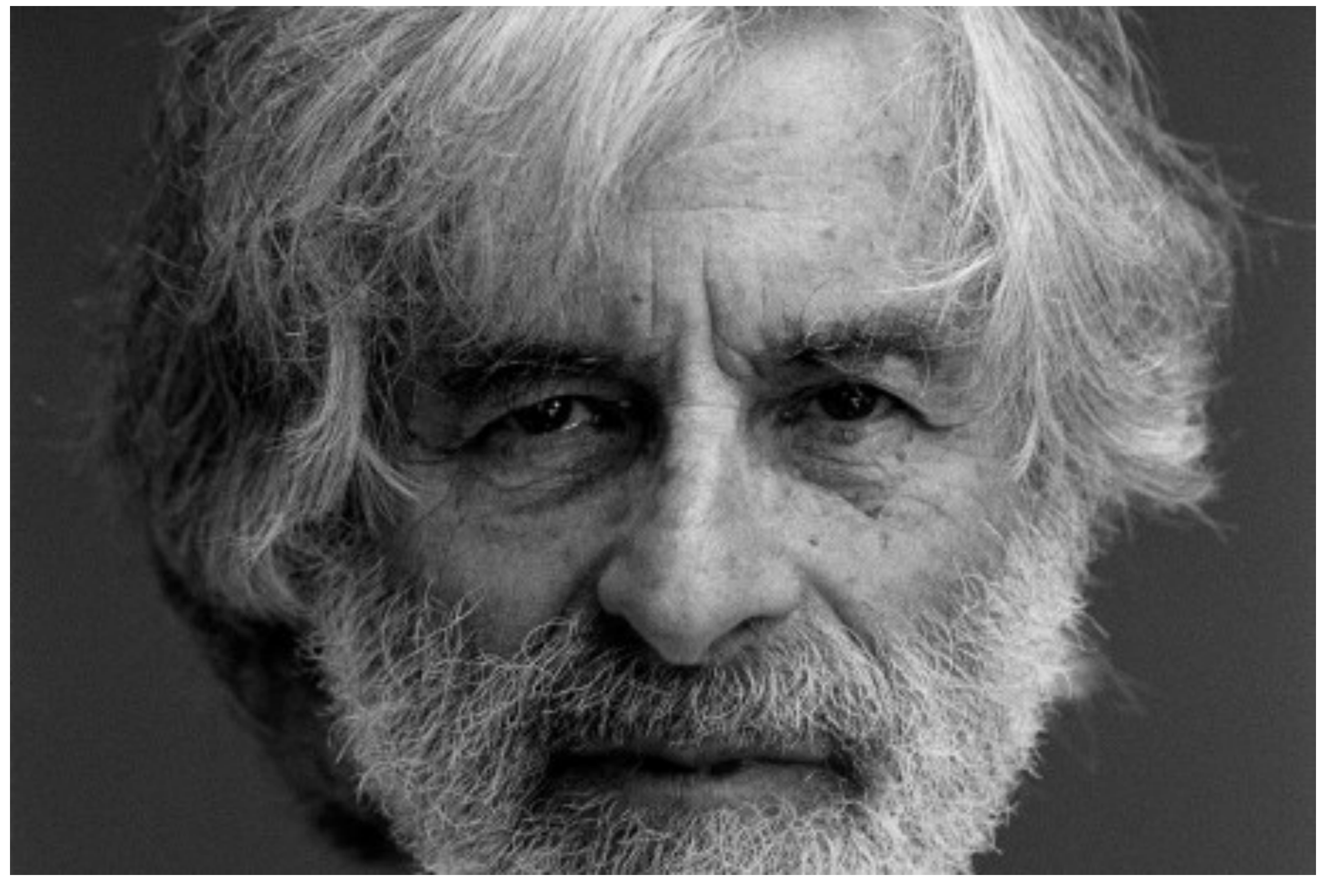
09:00 - 10:00: ECOOP Research Papers - Keynote at Mancy
Chair(s): Sophia Drossopoulou Imperial College London

09:00 - 10:00 ☆ **Rethinking Compositionality: Composing Proofs From Program Behaviours**
Talk Azadeh Farzan University of Toronto
Keynote

Azadeh Farzan Keynote Speaker
University of Toronto

Composition:
A Way to Make Proofs Harder

Leslie Lamport, 1997



In 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build.

It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years.

When distracting *language features* are removed and the underlying *mathematics* is revealed, compositional reasoning is seen to be of little use.

Compositional Software Verification

is

uncovering the *mathematics* inside Programs

This Talk

Composing Proofs about Programs

This Talk

Composing Proofs about
Distributed Programs

Composing Proofs about Programs

Functional

Imperative

Concurrent

Distributed

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional		
Imperative		
Concurrent		
Distributed		

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	
Imperative		
Concurrent		
Distributed		

$$\mathbb{N} \qquad\qquad \times, +, -$$

$$\prod_{i=1}^{n} f(i)$$

$$n! = \prod_{i=1}^{n} i \qquad\qquad k^n = \prod_{i=1}^{n} k$$

$$r(n) = n^n - n!$$

Theorem: $\forall\, n > 1,\ r(n) \in \mathbb{N} \wedge r(n) > 0.$

Corollary: $\forall\, n > 1,\ r(n) \geq 0.$

$\mathbb{N}$ $\times, +, -$ $\prod_{i=1}^{n} f(i)$

`nat` `*, +, -` `product`

$$n! = \prod_{i=1}^{n} i$$

```
fact n = product [1..n]
```

$$k^n = \prod_{i=1}^{n} k$$

```
pow k n = product (replicate n k)
```

$$r(n) = n^n - n!$$

```
r n = pow n n - fact n
```

Theorem: ∀ n > 1, r(n) ∈ $\mathbb{N}$ ∧ r(n) > 0.

Theorem: `∀ n > 1, r n : nat ∧ r n > 0.`

Corollary: ∀ n > 1, r(n) ≥ 0.

Corollary: `∀ n > 1, r n ≥ 0.`

Functional (Declarative) Programming

Programs are *functions* that manipulate with values.

Verification: proving theorems about function compositions.

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative		
Concurrent		
Distributed		

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	
Concurrent		
Distributed		

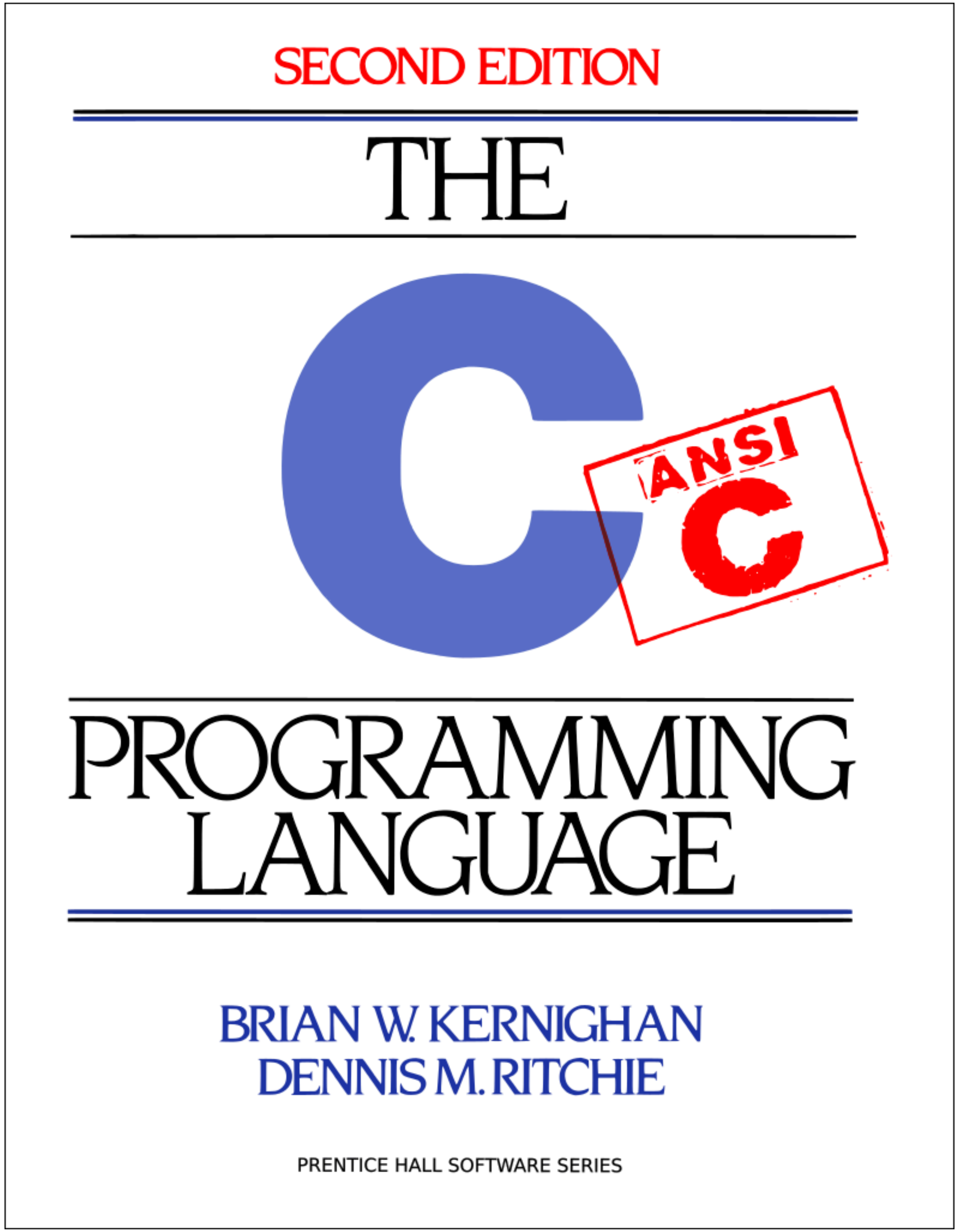
SECOND EDITION

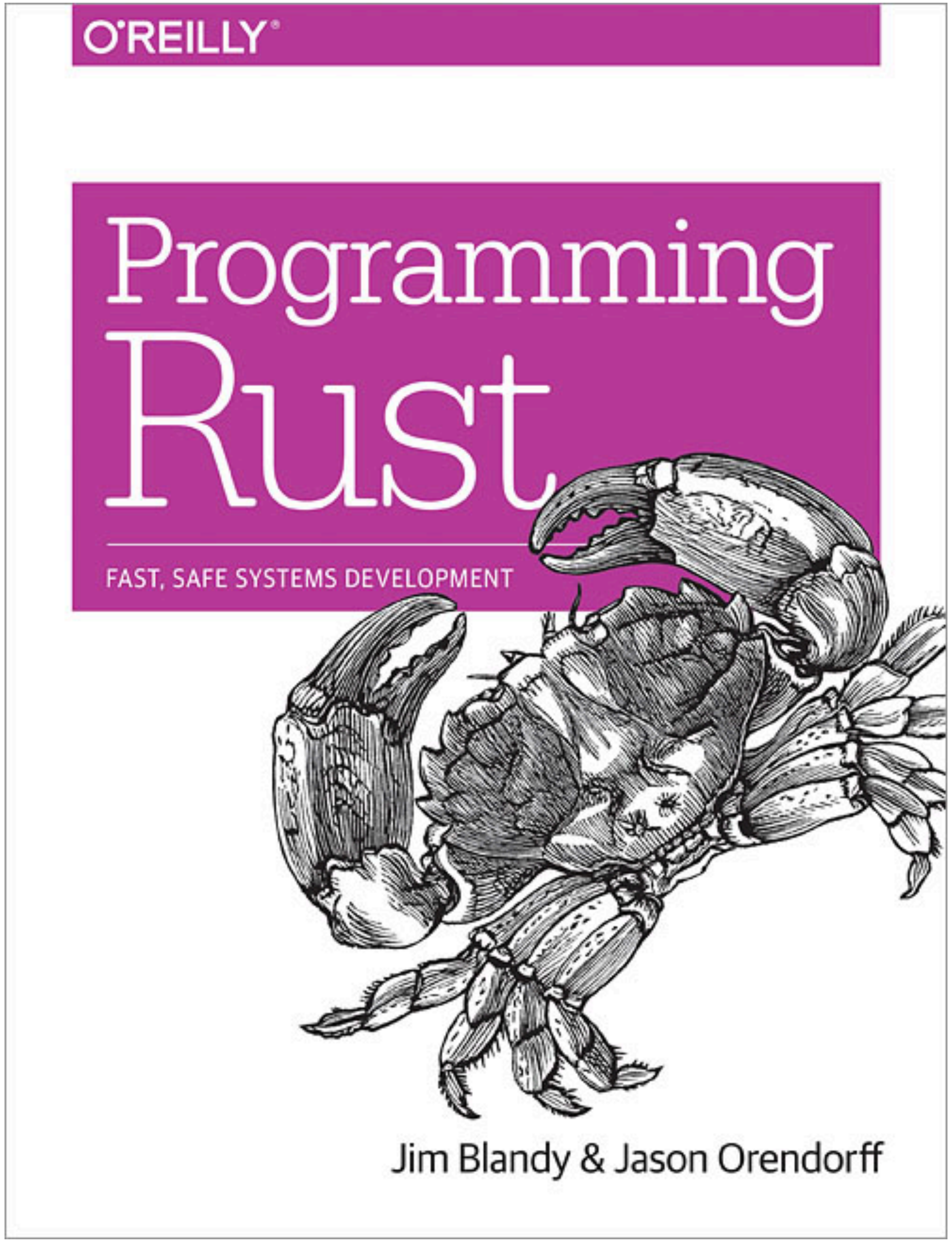
THE C PROGRAMMING LANGUAGE

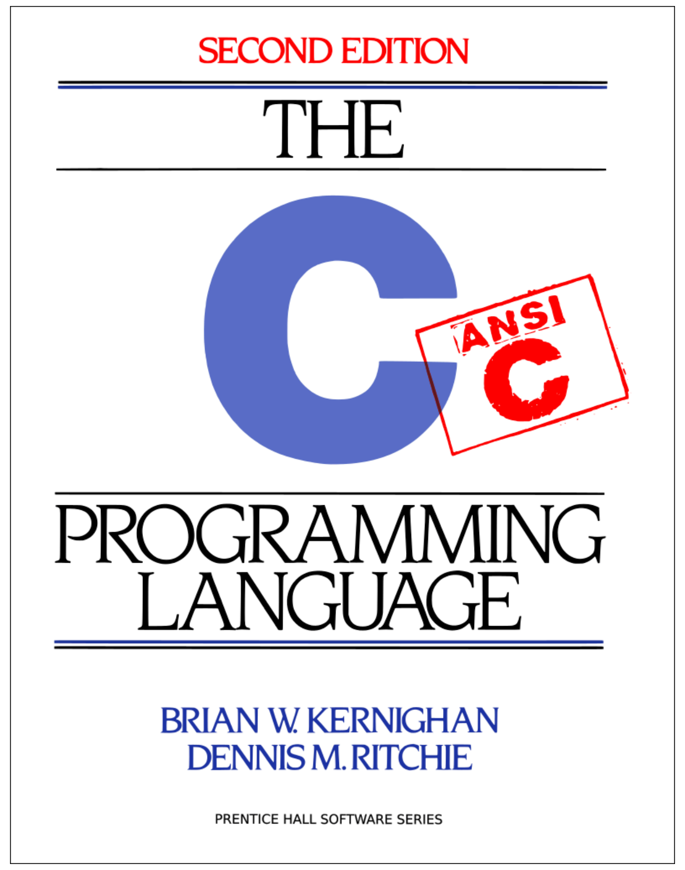
ANSI C

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES


SECOND EDITION
THE
C
ANSI C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES


O'REILLY
Programming
Rust
FAST, SAFE SYSTEMS DEVELOPMENT
Jim Blandy & Jason Orendorff

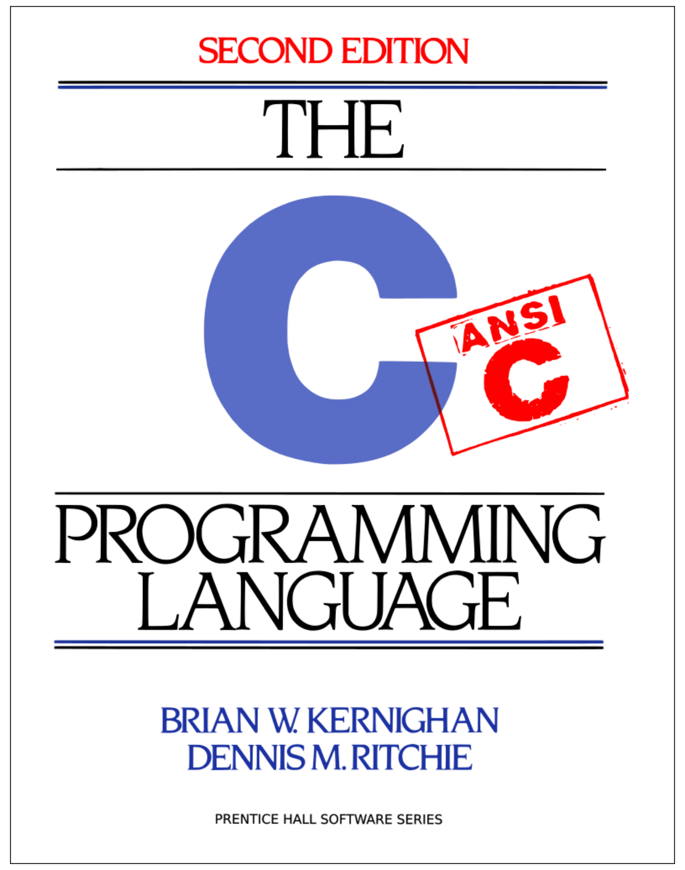



```c
int pow(int k, int n) {
  int result = 1;
  int term = k;
  while (n) {
    if (n & 1) result *= term;
    term *= term;
    n = n >> 1;
  }
  return result;
}
```

```c
int fact(int n) {
  if (n <= 0) return 1;
  int i = 1, f = 1;
  while (i <= n) {
    f = f * i;
    i++;
  }
  return f;
}
```

```c
int pow(int k, int n) {
  int result = 1;
  int term = k;
  while (n) {
    if (n & 1) result *= term;
    term *= term;
    n = n >> 1;
  }
  return result;
}
```

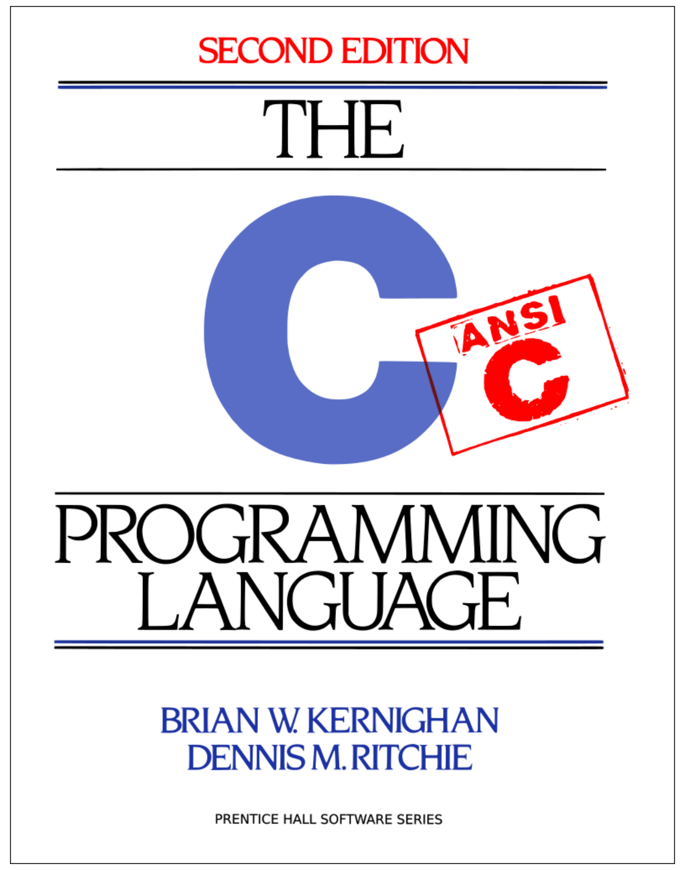
```c
int fact(int n) {
  if (n <= 0) return 1;
  int i = 1, f = 1;
  while (i <= n) {
    f = f * i;
    i++;
  }
  return f;
}
```




```c
int x, y, z;
int r(int n) {
  x = fact(n);
  y = pow(n, n);
  z = y - x;
  return z;
}
```

```
int pow(int k, int n) {
  int result = 1;
  int term = k;
  while (n) {
    if (n & 1) result *= term;
    term *= term;
    n = n >> 1;
  }
  return result;
}
```

```
int fact(int n) {
  if (n <= 0) return 1;
  int i = 1, f = 1;
  while (i <= n) {
    f = f * i;
    i++;
  }
  return f;
}
```




```
int x, y, z;
int r(int n) {
  x = fact(n);
  y = pow(n, n);
  z = y - x;
  return z;
}
```

Sub-programs are no longer mathematical *functions*: their *result* is their *effect* on the state of memory!

```
int x, y, z;
int r(int n) {
  x = fact(n);
  y = pow(n, n);
  z = y - x;
  return z;
}
```

Theorem: $\forall\, n > 1,\ r(n) \in \mathbb{N} \wedge r(n) > 0.$

Morally, still “holds”, but the “proof” requires
unfolding all definitions and *symbolically executing* the code.

Imperative (Systems) Programming

Programs are step-wise *state transformers*.

Verification: proving theorems about *sequences of state modifications*.

Declarative
~~Imperative~~ Programming

functions (state → state)

Programs are ~~step-wise *state transformers*.~~

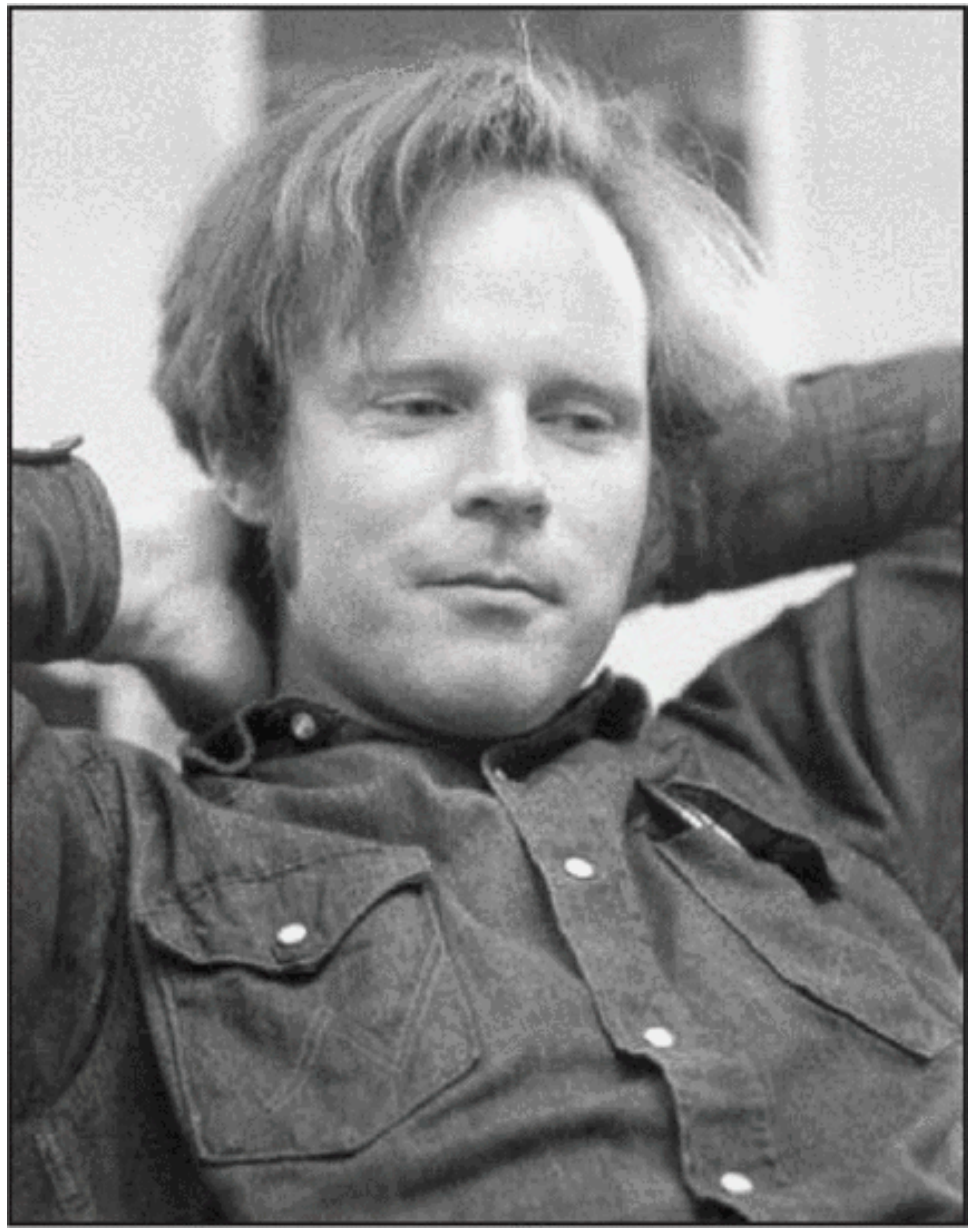
net effects of

Verification: proving theorems about *sequences of state modifications*.

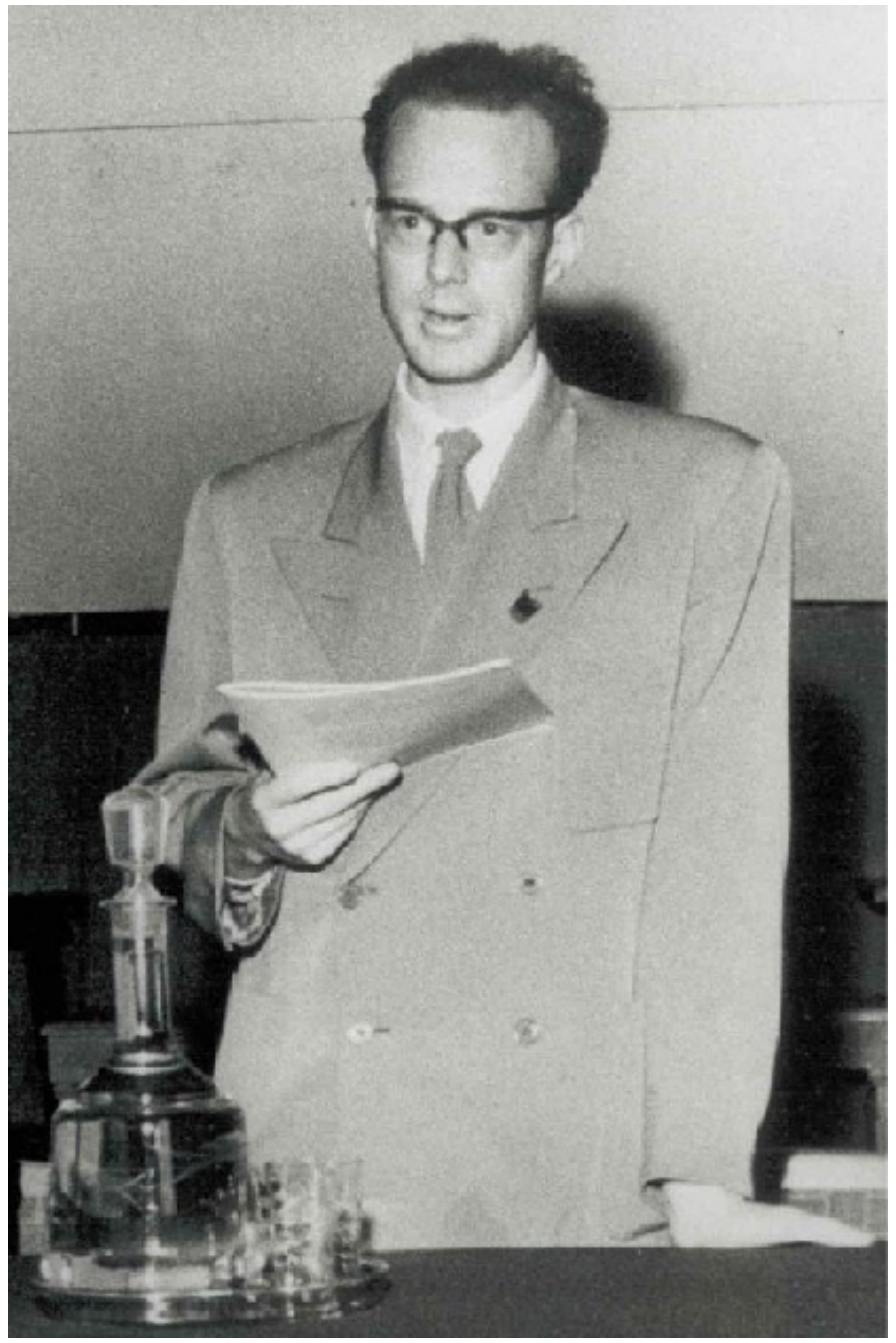
Program Logics

(aka *Type Theories for Imperative Programming*)

Program Logics



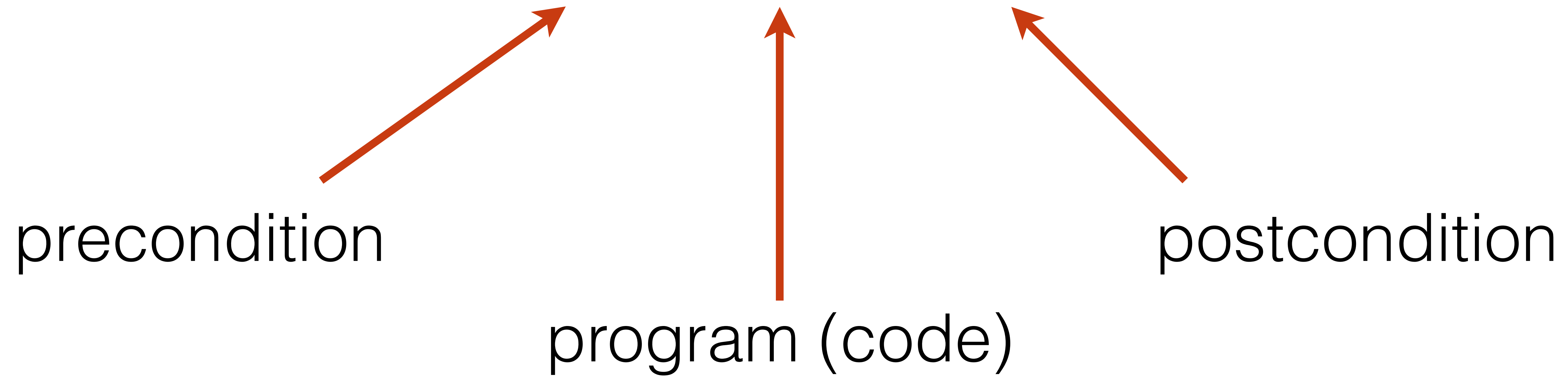
Robert W. Floyd (1967)



Tony Hoare (1969)

Program Logics

$$\{\, P \,\}\ c\ \{\, Q \,\}$$



If the initial state satisfies P, then
the program c is *safe* to run and its *final* state satisfies Q.

Example: $\{\,\mathrm{True}\,\}$ `x := 3; x := 5` $\{\, x = 5 \,\}$

Program Logics

$$\{ P[e/x] \}\ x := e\ \{ P \} \quad \text{(Assign)}$$

$$\frac{\{ I \wedge e \}\ c\ \{ I \}}{\{ I \}\ \textbf{while}\ e\ \textbf{do}\ c\ \{ I \wedge \neg e \}} \quad \text{(While)}$$

$$\frac{\{ P \}\ c_1\ \{ Q \} \qquad \{ Q \}\ c_2\ \{ R \}}{\{ P \}\ c_1; c_2\ \{ R \}} \quad \text{(Seq)}$$

$$\frac{\{ P \wedge e \}\ c_1\ \{ Q \} \qquad \{ P \wedge \neg e \}\ c_2\ \{ Q \}}{\{ P \}\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{ Q \}} \quad \text{(Cond)}$$

$$\frac{P \Rightarrow P_1 \qquad \{ P_1 \}\ c\ \{ Q_1 \} \qquad Q_1 \Rightarrow Q}{\{ P \}\ c\ \{ Q \}} \quad \text{(Conseq)}$$

Program Logics in Action

```
int x, y, z;
int r(int n) {
  x = fact(n);
  y = pow(n, n);
  z = y - x;
  return z;
}
```

Program Logics in Action

{ n ≥ 1 } `fact(n)` { res = n! }

{ n ≥ 1 } `pow(k, n)` { res = k^n }

```
int x, y, z;
int r(int n) {
```

{ n > 1 }

```
x = fact(n);
```

(Assign & Seq) { n > 1 ∧ x = n! }

```
y = pow(n, n);
```

(Assign & Seq) { n > 1 ∧ x = n! ∧ y = n^n }

```
z = y - x;

return z; }
```

Program Logics in Action

{ n ≥ 1 } `fact(n)` { res = n! }

{ n ≥ 1 } `pow(k, n)` { res = k^n }

```
int x, y, z;
int r(int n) {
```

{ n > 1 }

```
  x = fact(n);
```

{ n > 1 ∧ x = n! }

```
  y = pow(n, n);
```

{ n > 1 ∧ x = n! ∧ y = n^n }

```
  z = y - x;
```

(Assign & Seq) { n > 1 ∧ x = n! ∧ y = n^n ∧ z = n^n - n! }

```
  return z; }
```

Program Logics in Action

{ n ≥ 1 } *fact*(n) { res = n! }   { n ≥ 1 } *pow*(k, n) { res = k^n }

```
int x, y, z;
int r(int n) {
```

{ n > 1 }

```
  x = fact(n);
```

{ n > 1 ∧ x = n! }

```
  y = pow(n, n);
```

{ n > 1 ∧ x = n! ∧ y = n^n }

```
  z = y - x;
```

(Conseq)   { n > 1 ∧ res = n^n - n! }

```
return z; }
```

Program Logics in Action

$\{ n \geq 1 \}$ ***fact***`(n)` $\{ res = n! \}$

$\{ n \geq 1 \}$ ***pow***`(k, n)` $\{ res = k^n \}$

$\{ \texttt{n} > 1 \}$

r`(n)`

$\{ \texttt{n} > 1 \land \texttt{res} = \texttt{n}^{\texttt{n}} - \texttt{n}! \}$

(Conseq) $\{ \texttt{res} \in \mathbb{N} \land \texttt{res} > 0 \}$

Program Logics in Action

{ n ≥ 1 } `fact(n)` { res = n! }

{ n ≥ 1 } `pow(k, n)` { res = k^n }

{ `n` > 1 }

`r(n)`

{ `res` ∈ ℕ ∧ `res > 0` }

Theorem: ∀ n > 1, *r*(n) ∈ ℕ ∧ *r*(n) > 0.

Program Logics

Imperative programming → declarative programming (math)

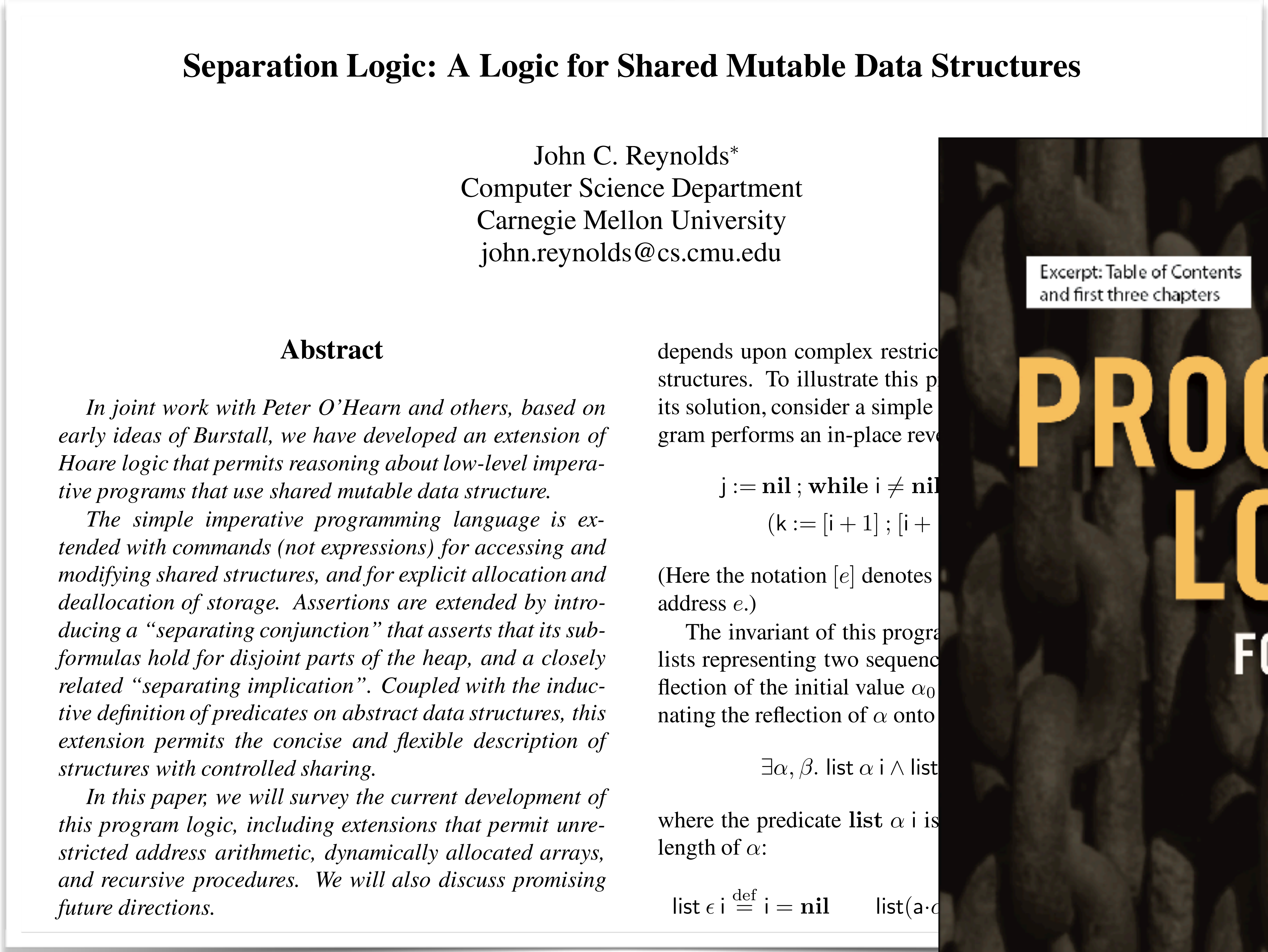
Hide implementation details.

“Saying *what* a program does without saying *how* it does it.”

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	Program Logics
Concurrent		
Distributed		

Program Logics in Academia and Industry



Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds*
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.

depends upon complex restric
structures. To illustrate this p
its solution, consider a simple
gram performs an in-place reve

j := **nil** ; **while** i ≠ **nil**
(k := [i + 1] ; [i +

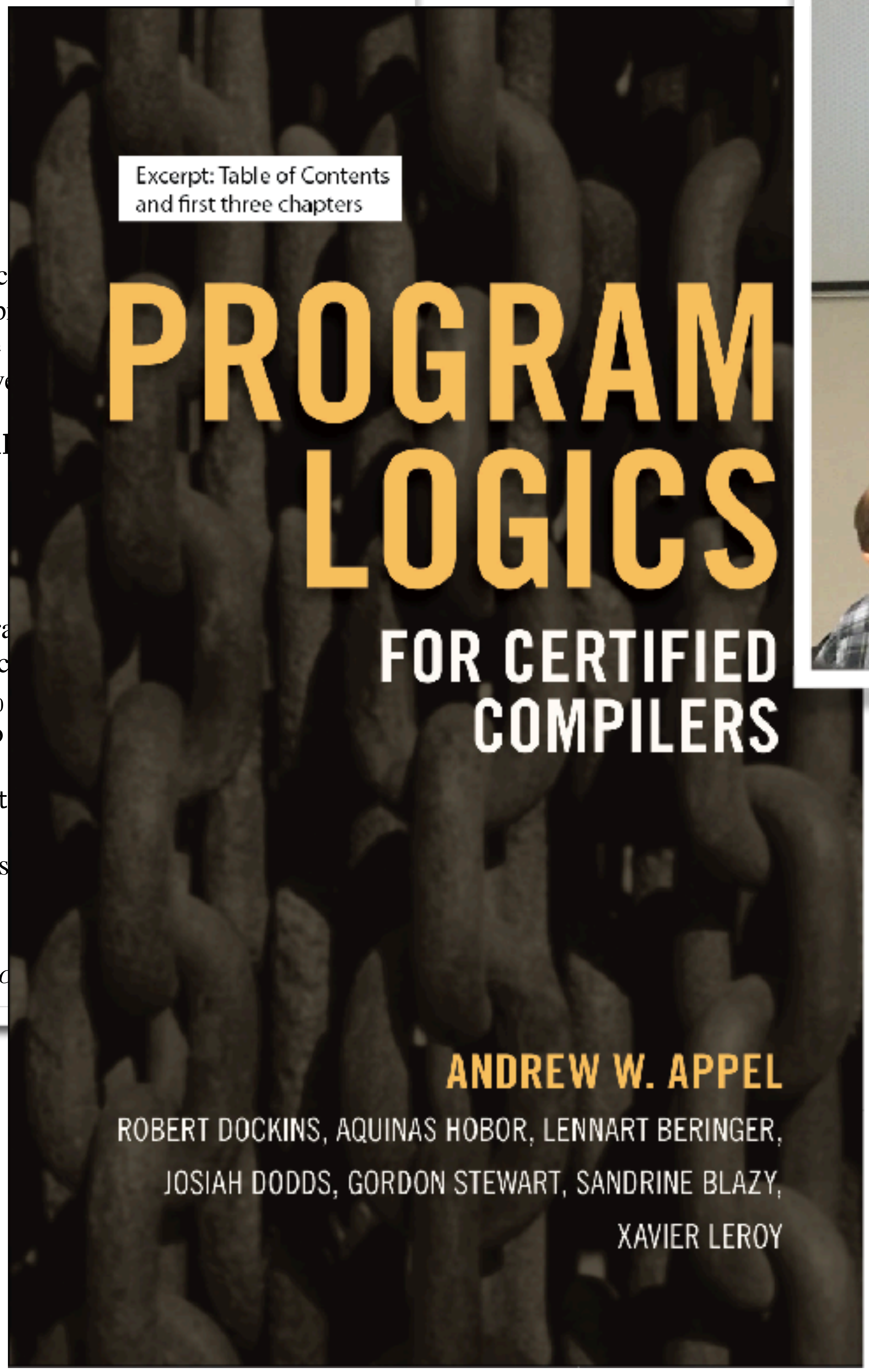
(Here the notation $[e]$ denotes
address e.)

The invariant of this progra
lists representing two sequenc
flection of the initial value α_0
nating the reflection of α onto

$\exists \alpha, \beta.$ list α i $\wedge$ list

where the predicate **list** α i is
length of α:

list ϵ i $\stackrel{\text{def}}{=}$ i = **nil**   list(a·



Code
Infer
TAKIPI

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	Program Logics
Concurrent		
Distributed		

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	Program Logics
Concurrent	interference	
Distributed		

Abstract Specifications of a Stack

$\{ S = xs \}$ **`push x`** $\{ S' = x :: xs \}$

$\{ S = xs \}$ **`pop()`** $\{ \ res = \bot \land S = Nil \lor \exists x, xs'.\ res = x \land xs = x :: xs' \land S' = xs' \}$

Suitable for sequential programming

Abstract Specifications of a Stack

$\{ S = xs \}$ **`push x`** $\{ S' = x :: xs \}$

$\{ S = xs \}$ **`pop()`** $\{ \; res = \bot \wedge S = Nil$
$\vee \; \exists x, xs'. \, res = x \wedge$
$xs = x :: xs' \wedge S' = xs' \}$

Breaks composition in the presence of thread interference.

$\{ S = Nil \}$

```
y := pop();
```

$\{ y = ??? \}$

$\{ S = Nil \}$

```
                    push 1;
y := pop();   ||
                    push 2;
```

$\{ y \in \{\bot\} \cup \{1, 2\} \}$

$\{ S = Nil \}$

```
y := pop();   ||   push 1;   ||   push 3;
                   push 2;
```

$\{ y \in \{\bot\} \cup \{1, 2, 3\} \}$

No proof reuse
(not thread-modular)

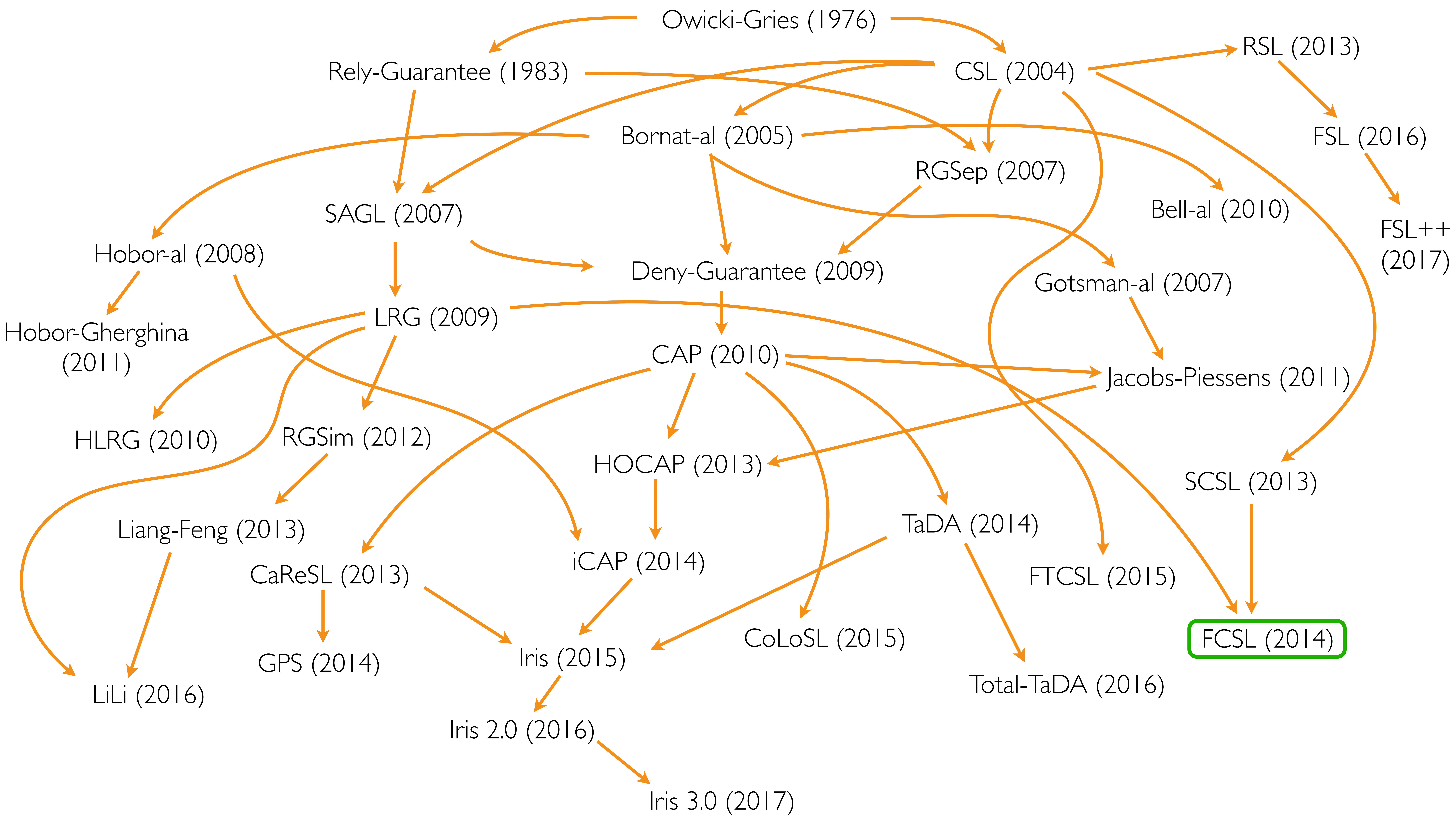
A reusable specification for pop?

$\{ S = Nil \}$

```
y := pop();
```

$\{ y = ??? \}$

Program Logics for Concurrency


Owicki-Gries (1976)
Rely-Guarantee (1983)
CSL (2004)
RSL (2013)
Bornat-al (2005)
FSL (2016)
RGSep (2007)
SAGL (2007)
Bell-al (2010)
FSL++ (2017)
Hobor-al (2008)
Deny-Guarantee (2009)
Gotsman-al (2007)
LRG (2009)
Hobor-Gherghina (2011)
CAP (2010)
Jacobs-Piessens (2011)
HLRG (2010)
RGSim (2012)
HOCAP (2013)
SCSL (2013)
Liang-Feng (2013)
TaDA (2014)
iCAP (2014)
CaReSL (2013)
FTCSL (2015)
CoLoSL (2015)
FCSL (2014)
GPS (2014)
Iris (2015)
Total-TaDA (2016)
LiLi (2016)
Iris 2.0 (2016)
Iris 3.0 (2017)

FCSL: Fine-grained Concurrent Separation Logic

Nanevski, Ley-Wild, Sergey, Delbianco [ESOP'14]

The main idea of FCSL

Capture the effect of *self*,
abstract over the effect of *others*.

(aka *Subjective specifications*)

Subjective stack specifications

Sergey, Nanevski, Banerjee [ESOP'15]

- H_s — "ghost history" of my **pushes/pops** to the stack
- H_o — "ghost history" of **pushes/pops** by all other threads

$\{ H_s = \varnothing \}$

```
y := pop();
```

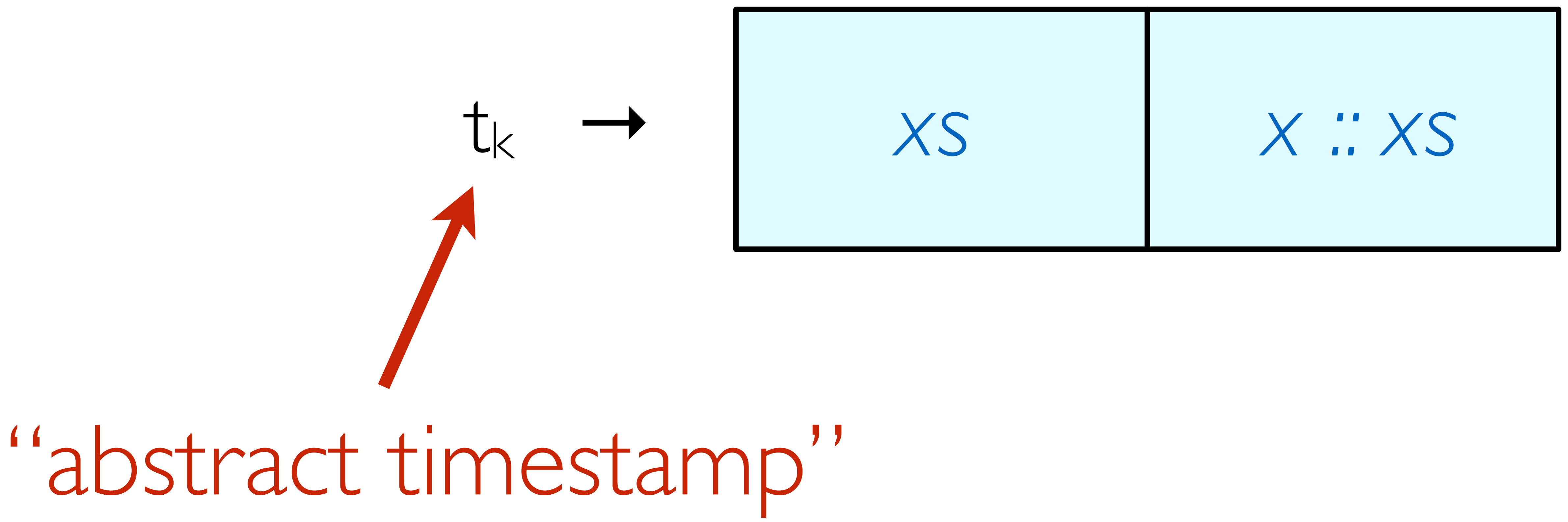
$\{ y = \bot \lor y = v, \text{ where } v \in \underbrace{\mathit{pushed}}(H_o) \}$

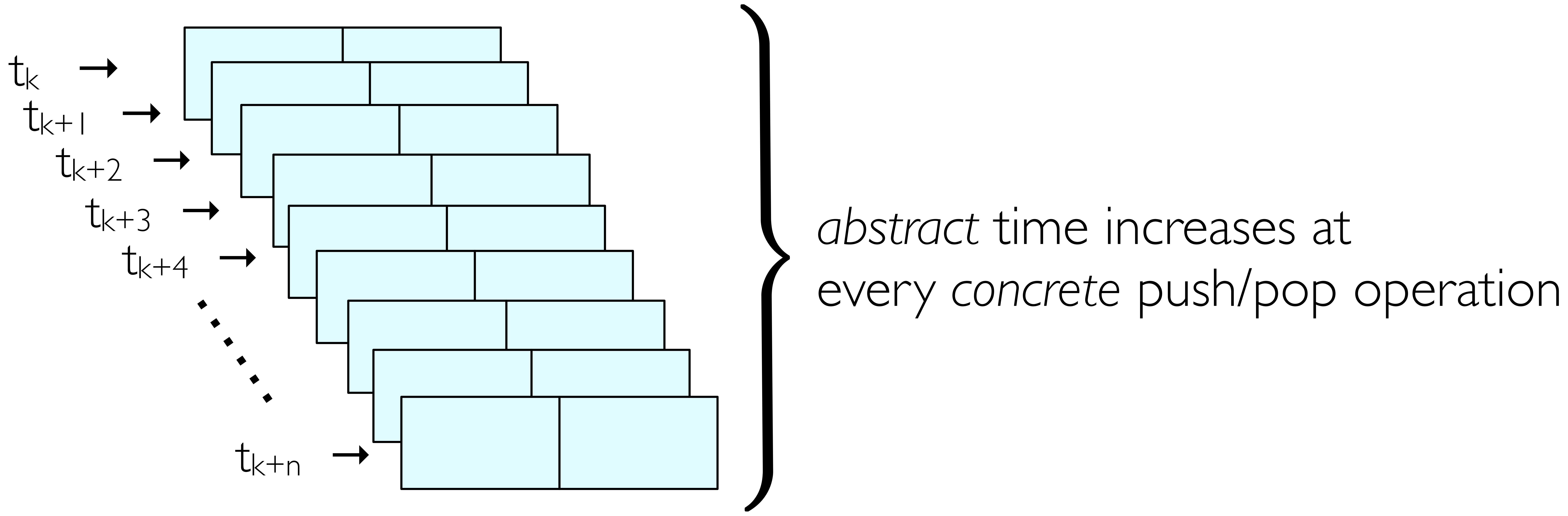
what I popped depends
on what the others have pushed

Atomic stack specifications

$\{\, S = xs \,\}$ **`push x`** $\{\, S' = x :: xs \,\}$

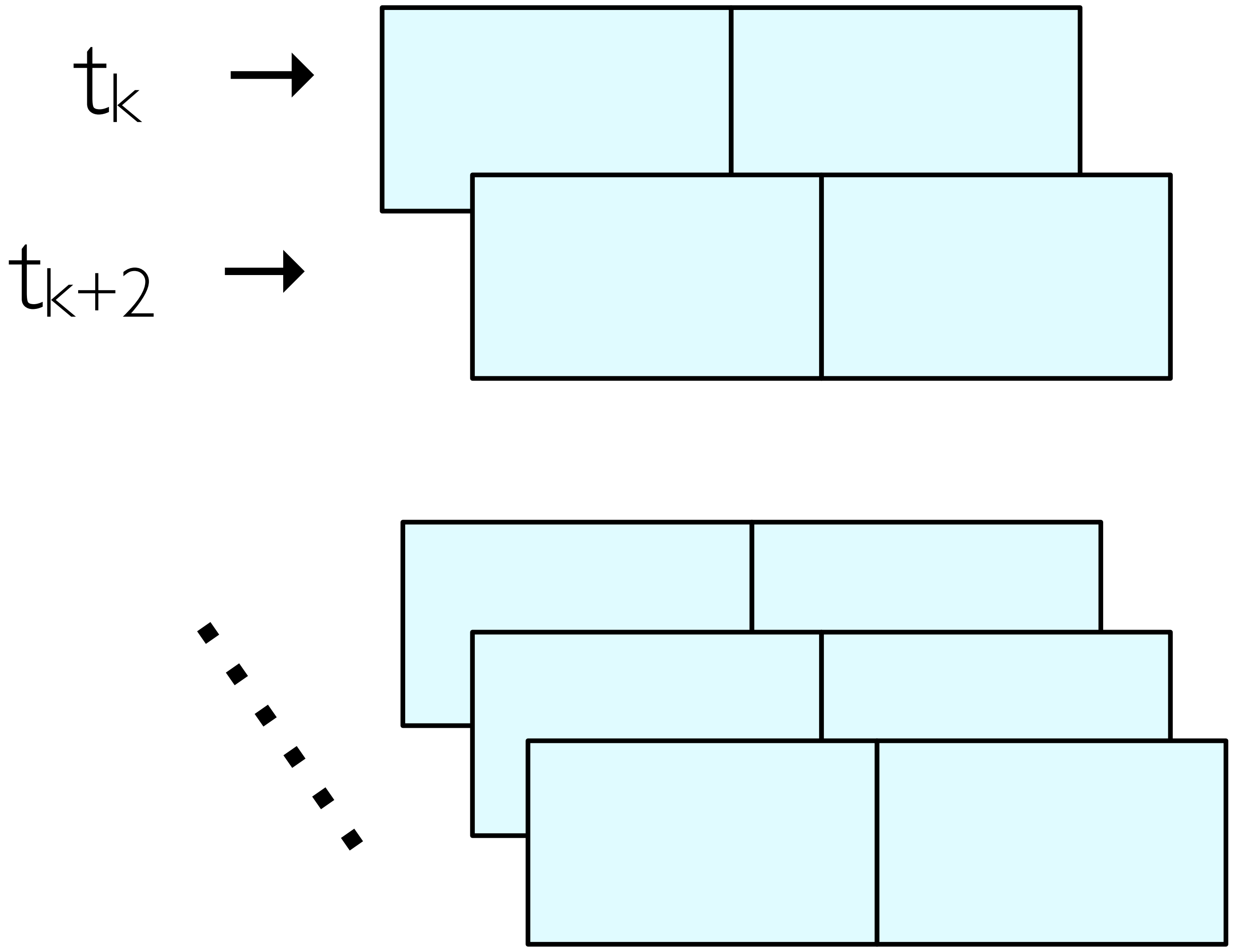
Atomic stack specifications



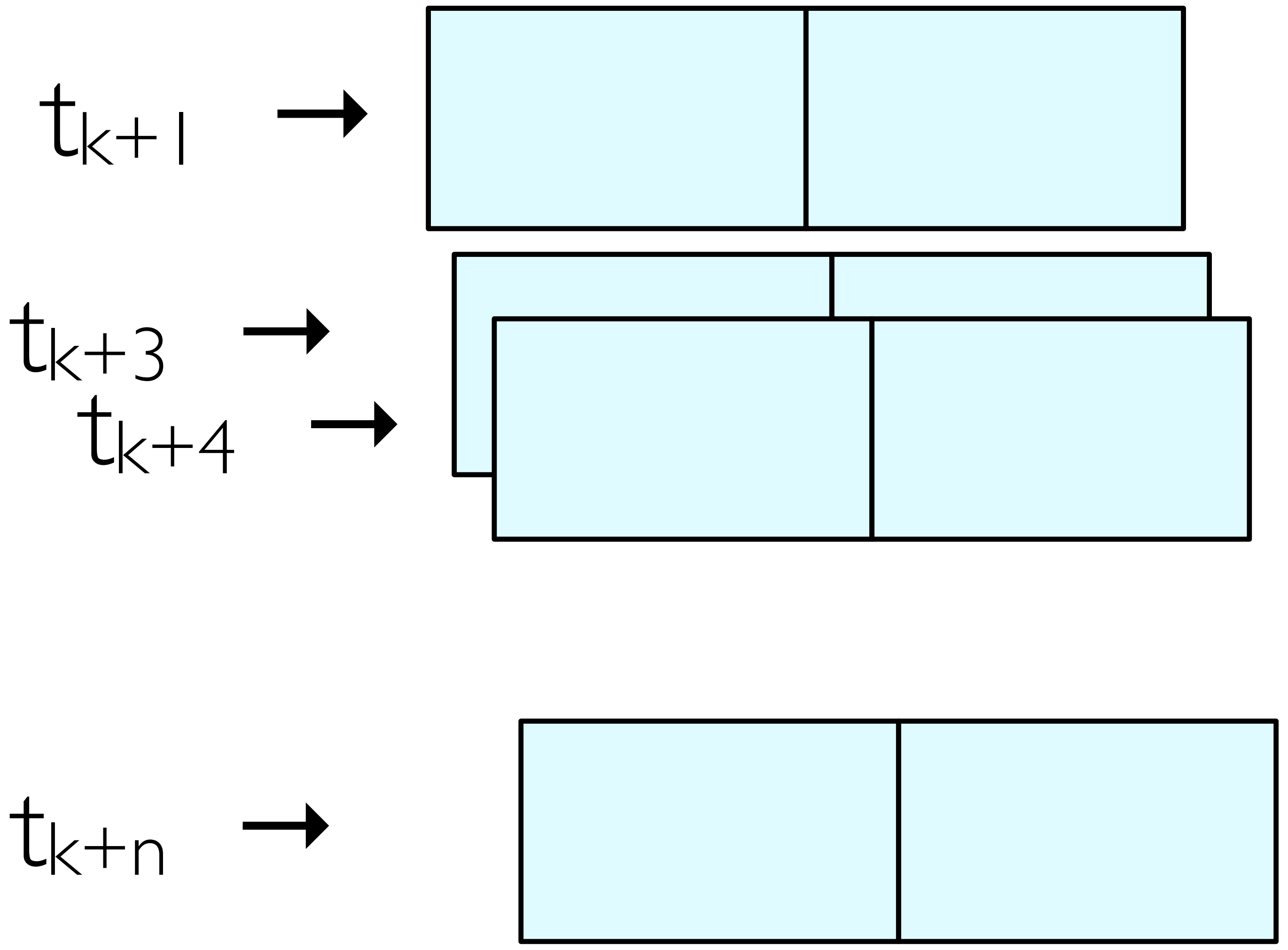


t_k
t_{k+1}
t_{k+2}
t_{k+3}
t_{k+4}
t_{k+n}
abstract time increases at
every *concrete* push/pop operation

Changes by *this* thread



Changes by *other* threads



Abstract Concurrent Stack Specification

self-contribution is a single entry

timestamp t allocated during the call

$$\forall H, \; \{ H_s = \emptyset \wedge H \subseteq H_o \}$$

`push x`

$$C_{stack} \vdash \{ \exists t, xs.\; H_s = t \mapsto (xs, x{::}xs) \wedge H \subseteq H_o \wedge H < t \}$$

$\{ S = Nil \}$

```
y := pop();  ||  push 1;  ||  push 3;
                 push 2;
```

$\{ H_s = \emptyset \}$

```
push 1;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \}$

```
push 2;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \oplus t_2 \mapsto (ys, \mathbf{2}::ys) \}$

$\{ H_s = \varnothing \}$

```
push 1;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \}$

```
push 2;
```

$\{ H_s = t_l \mapsto (xs, \mathbf{1}::xs) \oplus t_2 \mapsto (ys, \mathbf{2}::ys) \}$

$\{ H_s = \emptyset \}$

```
push 1;
```

$\{ H_s = t_1 \mapsto (xs, \mathbf{1}::xs) \}$

```
push 2;
```

$\{ H_s = t_1 \mapsto (xs, \mathbf{1}::xs) \oplus t_2 \mapsto (ys, \mathbf{2}::ys) \}$

$\|$

$\{ H_s = \emptyset \}$

```
push 3;
```

$\{ H_s = t_3 \mapsto (zs, \mathbf{3}::zs) \}$

$\{ H_s = \varnothing \}$

```
y := pop();
```

$\{ y \in \{\bot\} \cup \underline{\textit{pushed}}(H_o) \}$

||

$\{ H_s = \varnothing \}$

```
push 1;

push 2;
```

$\{ H_s = t_1 \mapsto (xs, \mathbf{1}{::}xs) \oplus t_2 \mapsto (ys, \mathbf{2}{::}ys) \}$

||

$\{ H_s = \varnothing \}$

```
push 3;
```

$\{ H_s = t_3 \mapsto (zs, \mathbf{3}{::}zs) \}$

$\{ H_s = \emptyset \}$

```
y := pop();
```

$\{ y \in \{\bot\} \cup \underline{pushed}(H_o) \}$

||

$\{ H_s = \emptyset \}$

```
push 1;
push 2;
```

$\{ H_s = t_1 \mapsto (xs, \underline{\mathbf{1}}::xs) \oplus t_2 \mapsto (ys, \underline{\mathbf{2}}::ys) \}$

||

$\{ H_s = \emptyset \}$

```
push 3;
```

$\{ H_s = t_3 \mapsto (zs, \underline{\mathbf{3}}::zs) \}$

$\{ H_s = \emptyset \}$

```
y := pop();
```

$\{ y \in \{\bot\} \cup \{\mathbf{1}, \mathbf{2}, \mathbf{3}\} \}$

||

$\{ H_s = \emptyset \}$

```
push 1;
push 2;
```

$\{ H_s = t_1 \mapsto (xs, \underline{\mathbf{1}}{::}xs) \oplus t_2 \mapsto (ys, \underline{\mathbf{2}}{::}ys) \}$

||

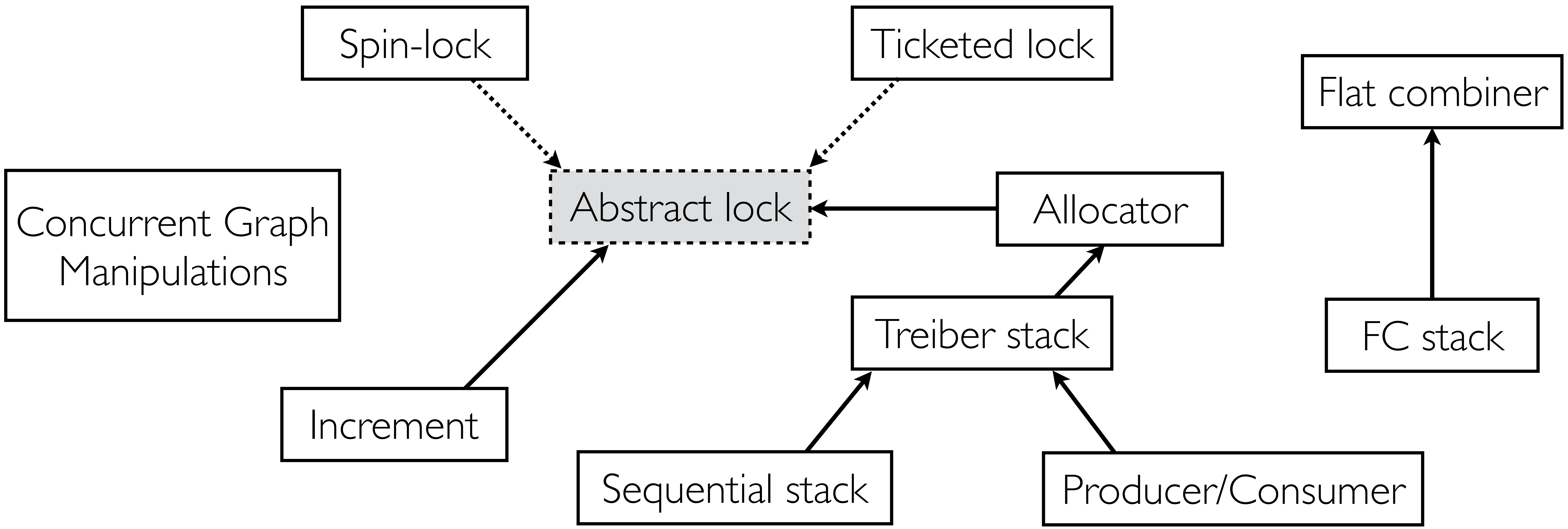
$\{ H_s = \emptyset \}$

```
push 3;
```

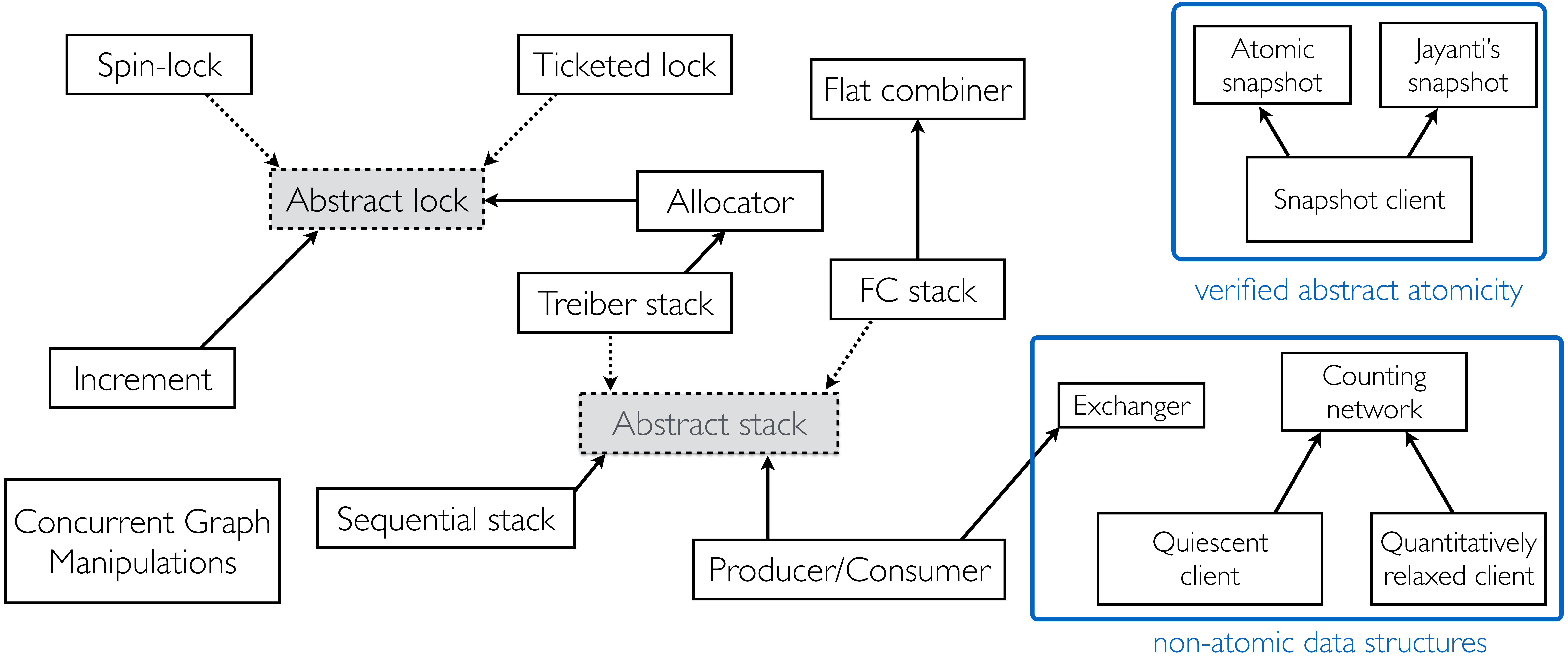
$\{ H_s = t_3 \mapsto (zs, \underline{\mathbf{3}}{::}zs) \}$

Composing Verified Concurrent Libraries

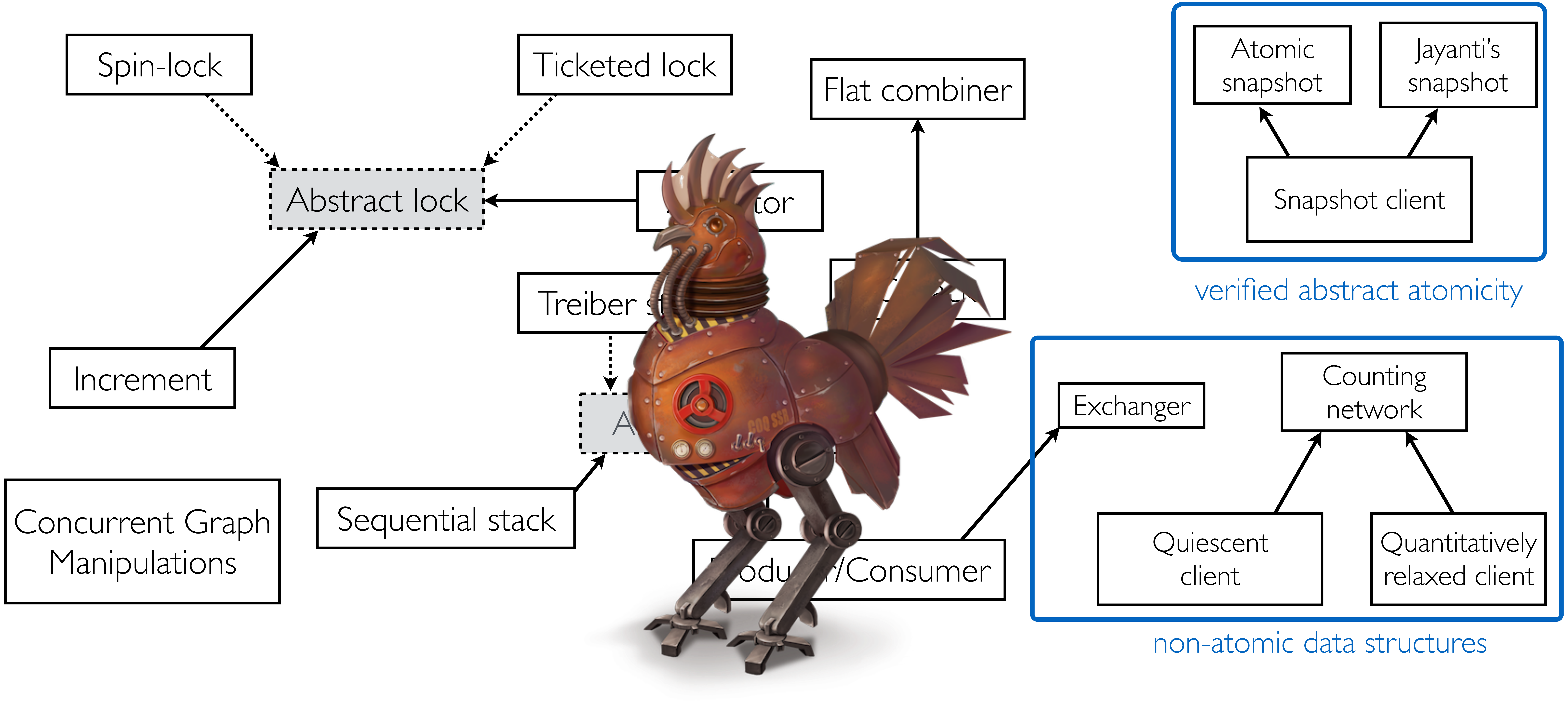
Sergey, Nanevski, Banerjee, Delbianco [PLDI'15, OOPSLA'16, ECOOP'17]



Composing Verified Concurrent Libraries



Composing Verified Concurrent Libraries



Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	Program Logics
Concurrent	interference	
Distributed		

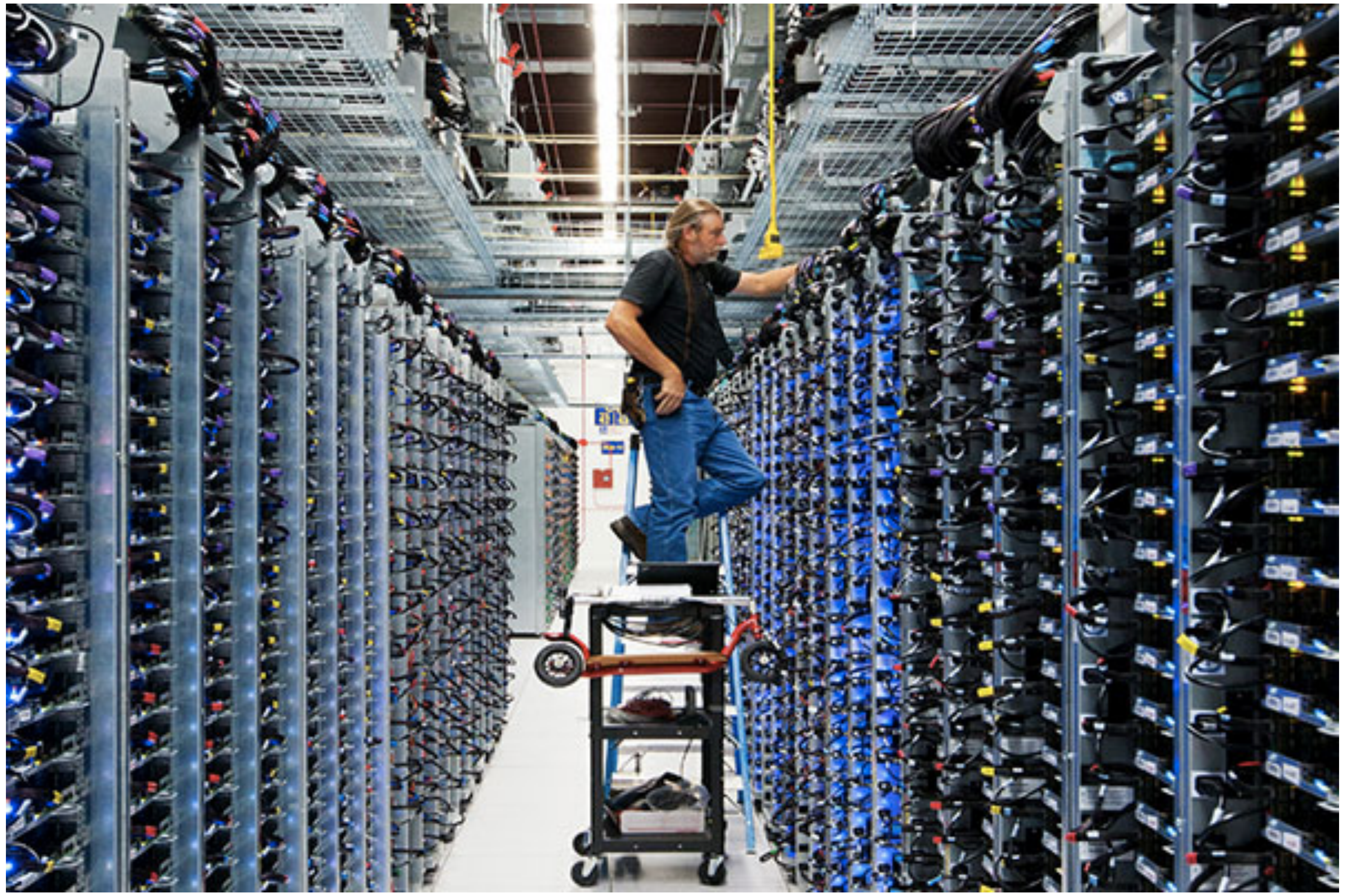
Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	Program Logics
Concurrent	interference	Program Logics + Subjectivity
Distributed		

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	Program Logics
Concurrent	interference	Program Logics + Subjectivity
Distributed	asynchronous message delivery unbounded delays lack of synchronisation network faults and partitions	

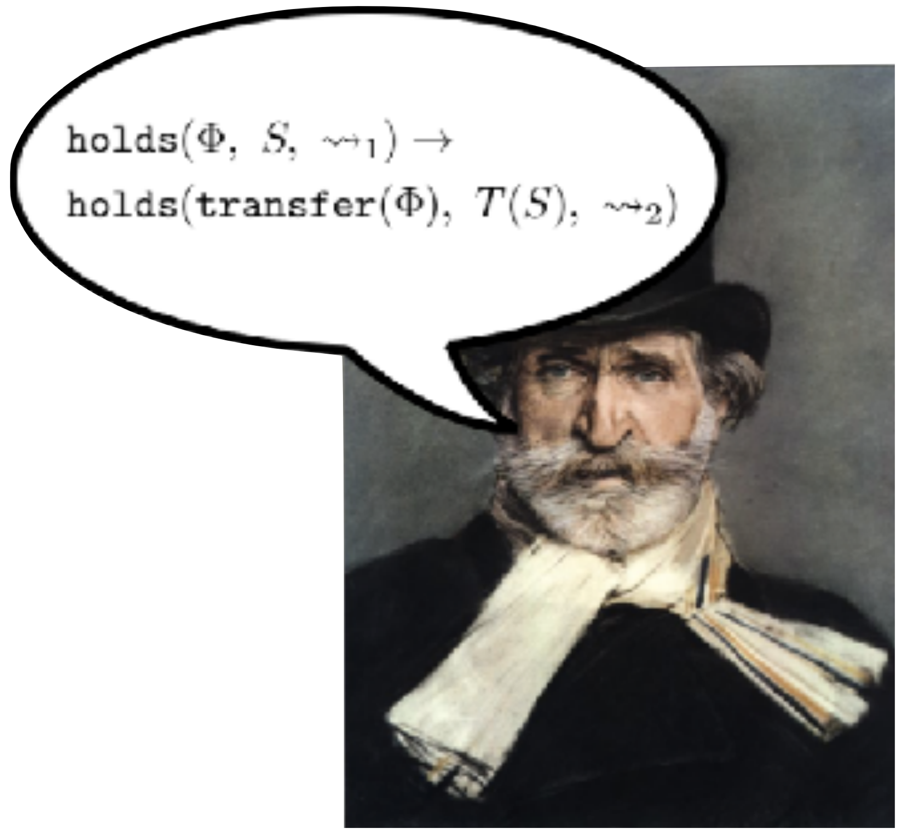
Distributed Systems

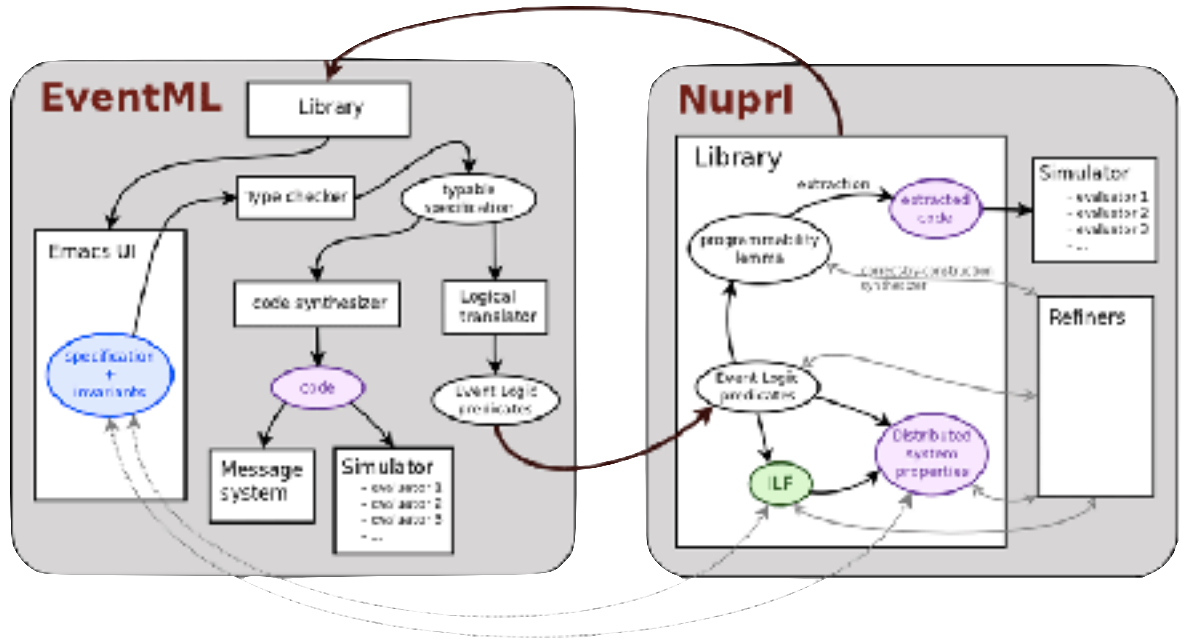


Distributed *Infrastructure*

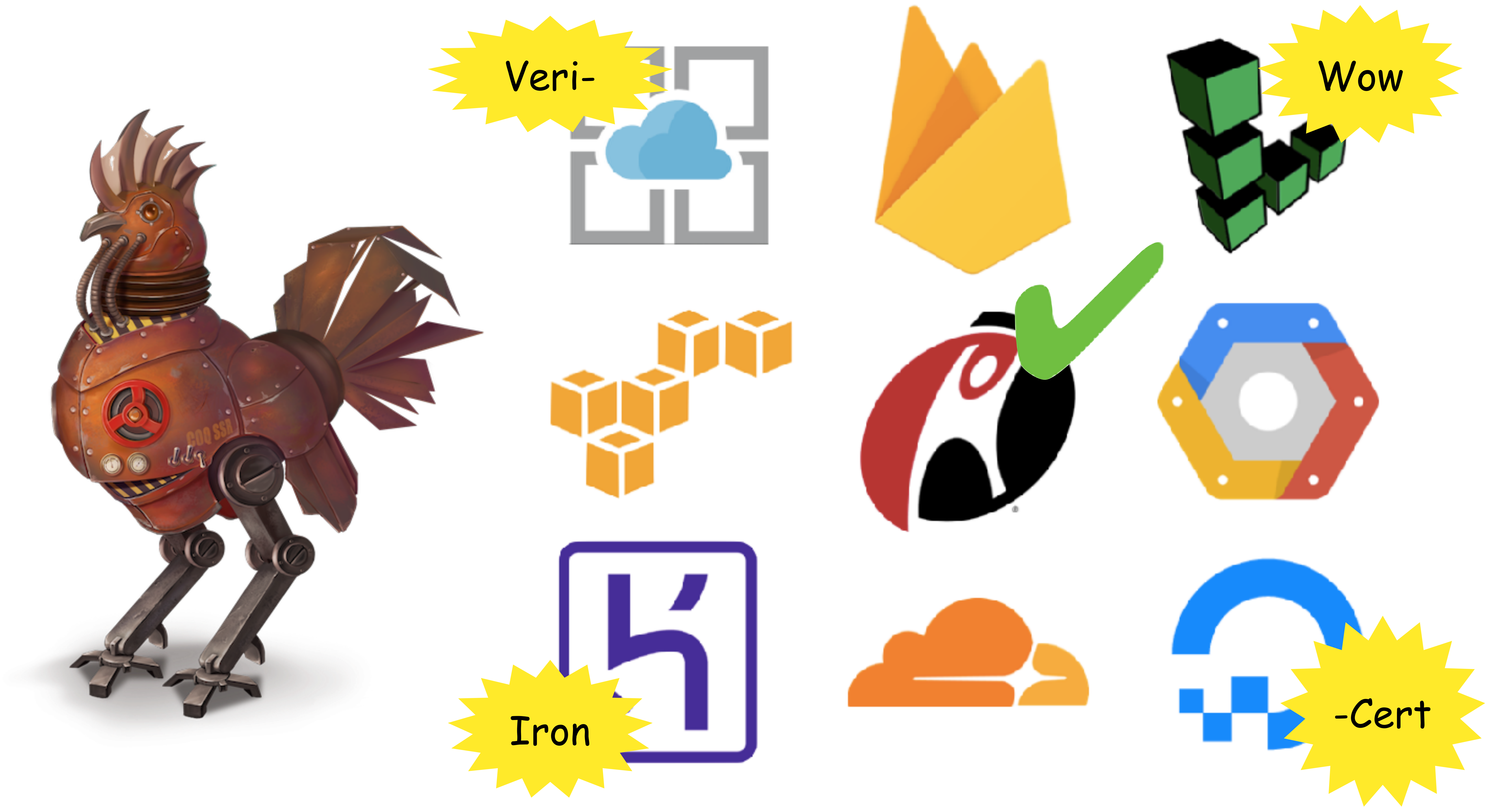
Distributed *Applications*

Verified Distributed Systems





Verified Distributed *Infrastructure*



Verified Distributed *Applications*

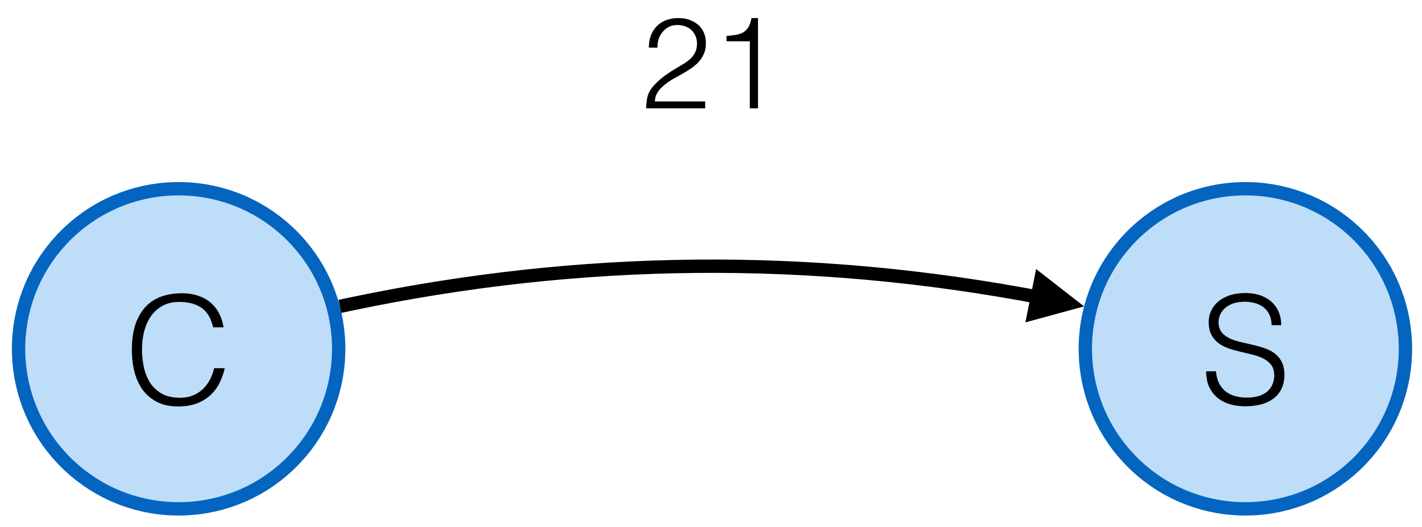
Verified Distributed *Applications*

Challenge: **Horizontal Compositionality**

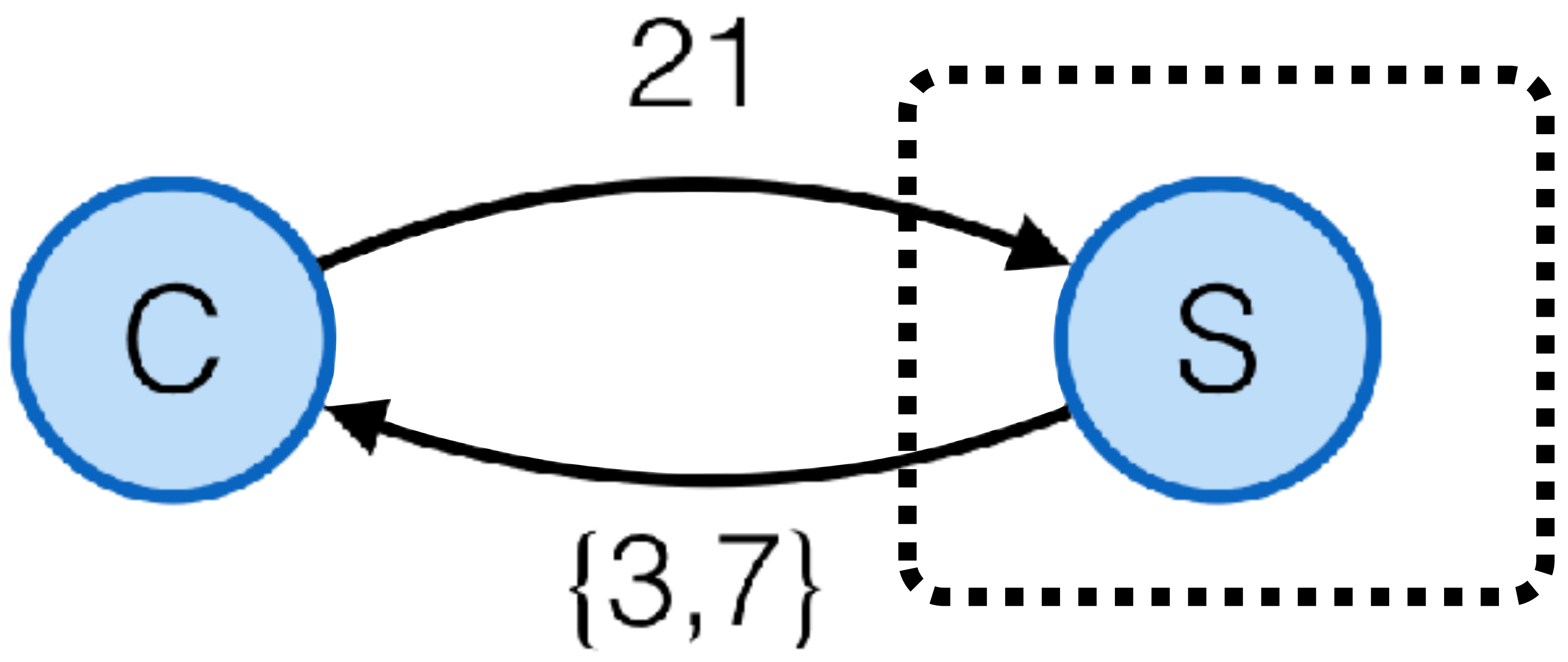
Verify *applications* ***without re-verifying*** the *services*.

One node's **client** is another node's **server**.

Cloud Compute



Cloud Compute



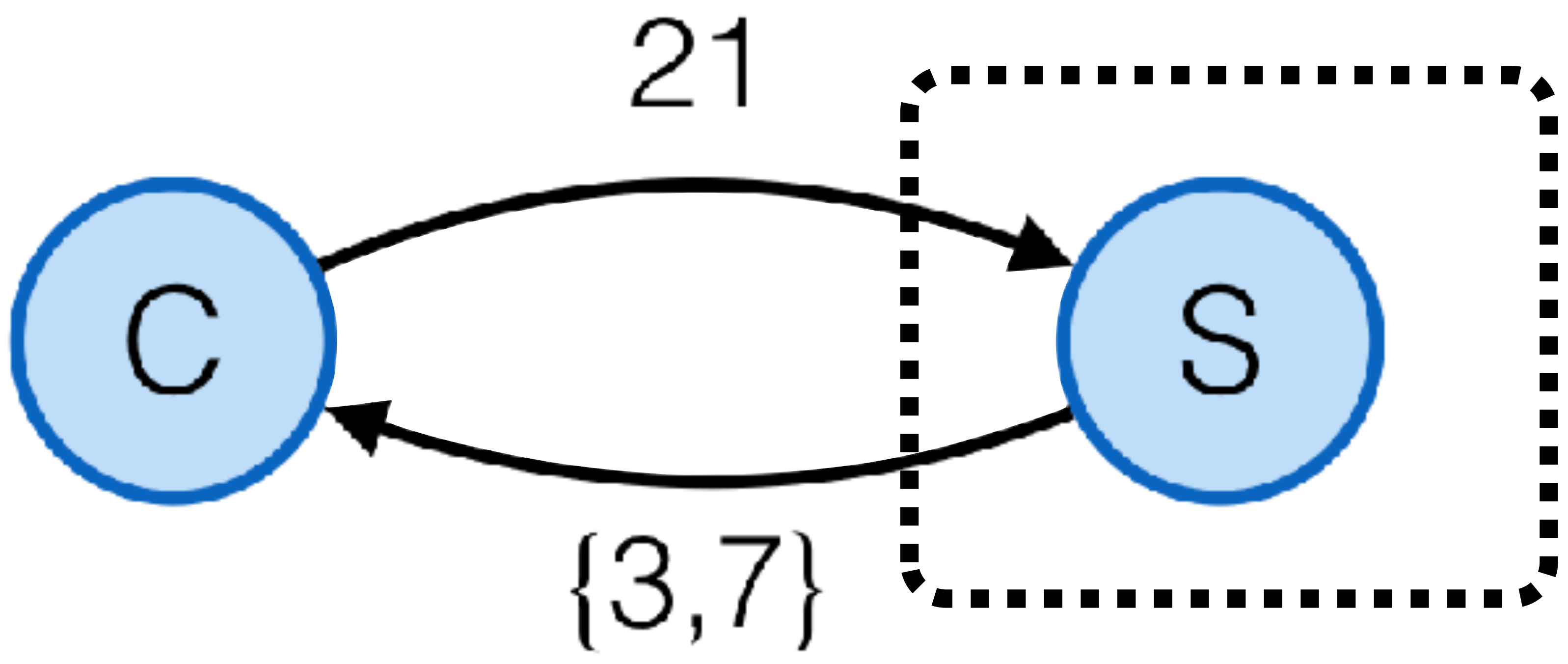
Cloud Compute: Server

```
while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from
```

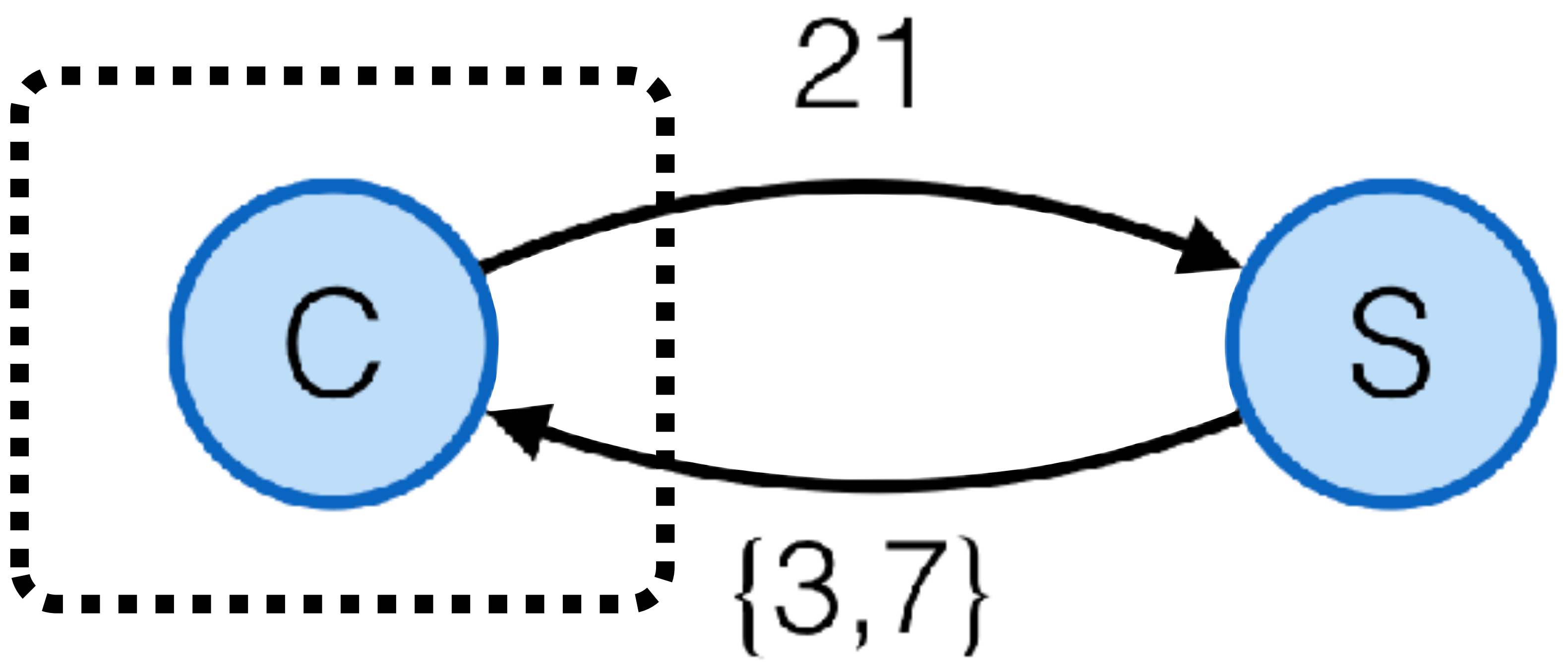
Traditional specification:
  messages from server have correct factors

Proved by finding an invariant of the system

Cloud Compute: Server



Cloud Compute: Client



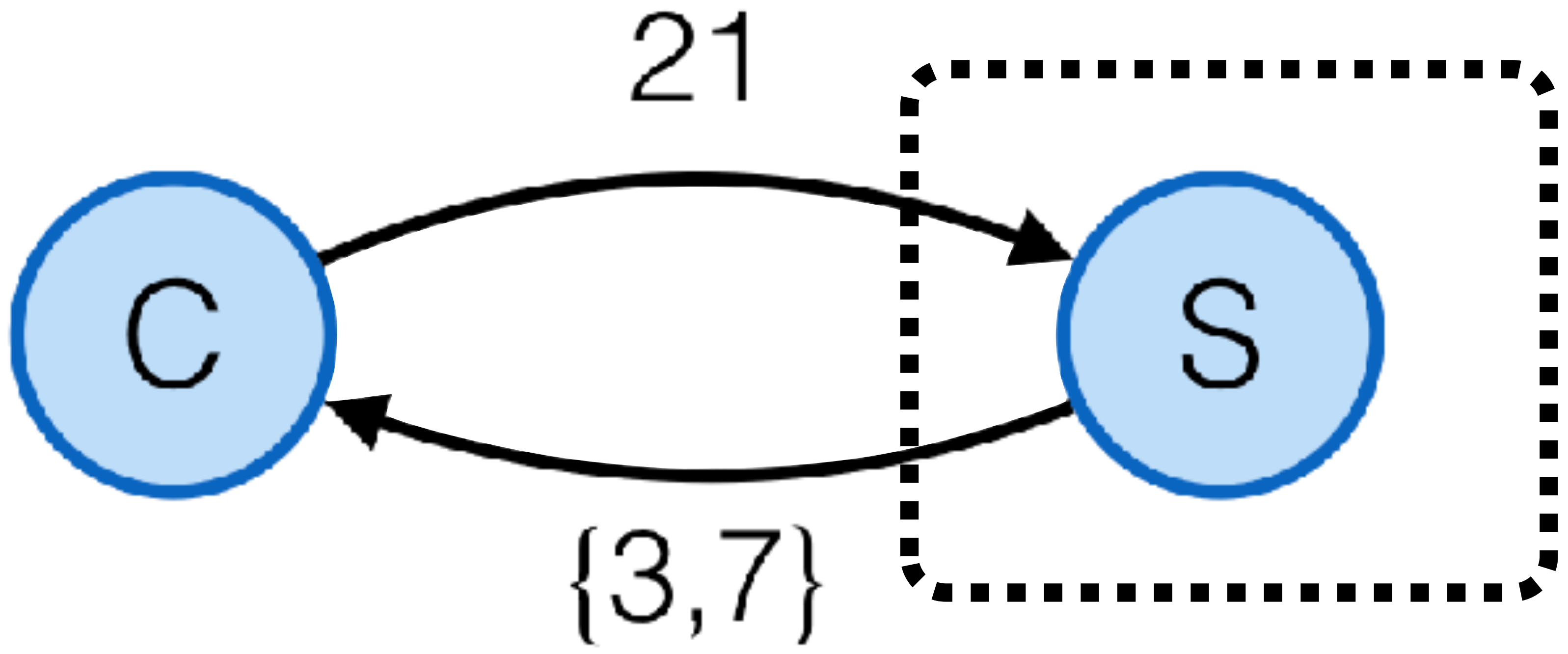
Cloud Compute: Client

```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

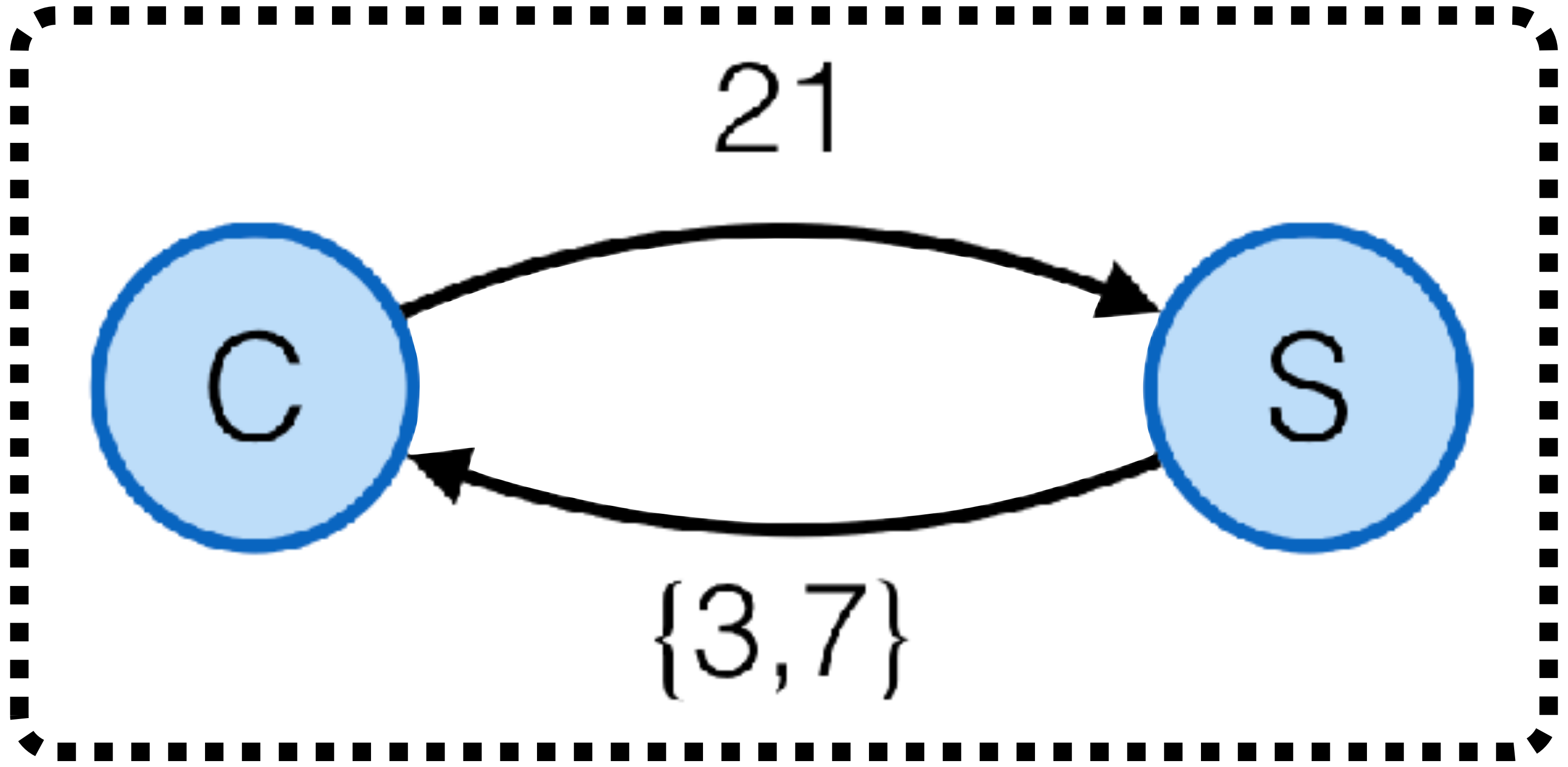
Start over with clients in system?

Idea: use protocol to describe client interface

Protocols



Protocols



A protocol is an **interface** among nodes

Enables compositional verification

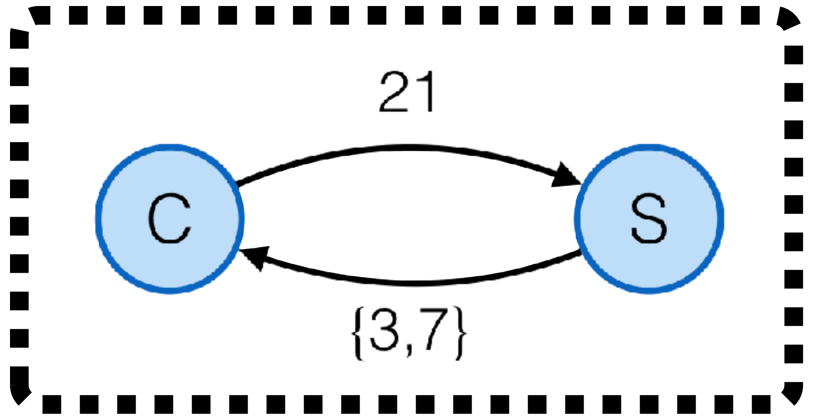
Distributed Systems

implement

Interaction Protocols

Compositional Reasoning
about **Distributed Systems**
requires
a **Protocol-Aware**
Program Logic

Cloud Compute Protocol



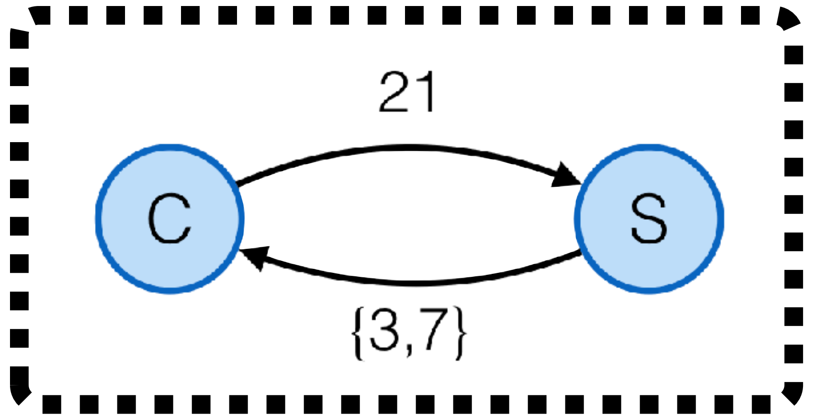
Messages:

State:

Transitions:

- Sends: precondition and effect
- Receives: effect

Cloud Compute Protocol



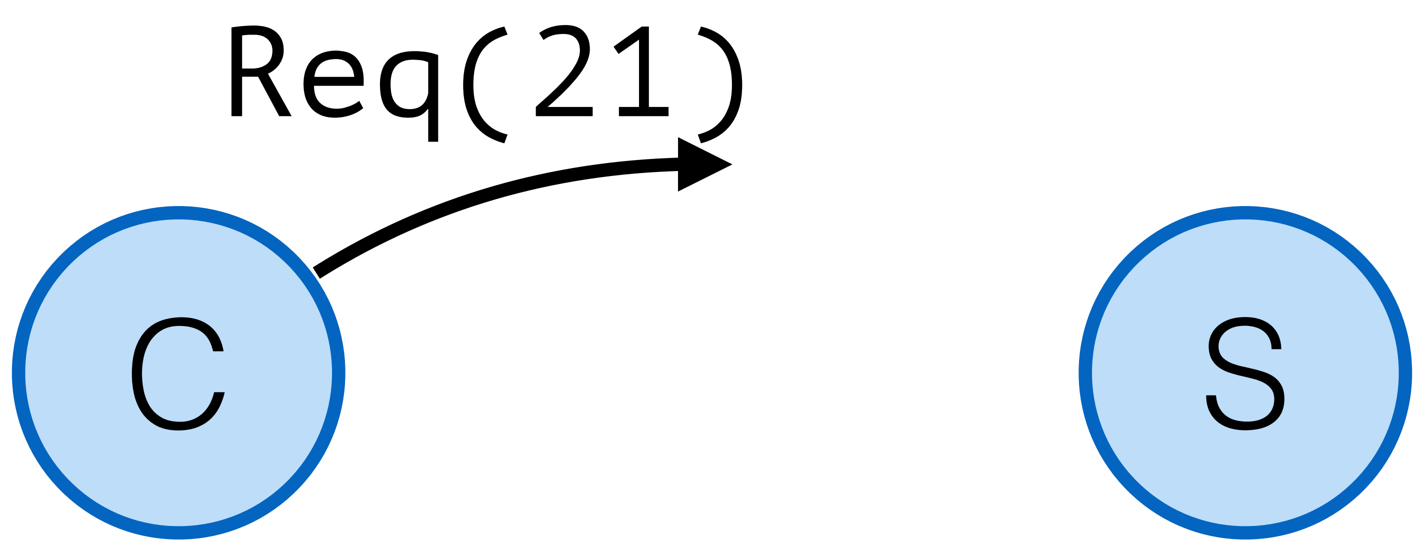
Messages: `Req(n) | Resp(n,s)`

State: `outstanding: Set<Msg>`

Transitions:

Sends:	Req	Resp
Receives:	Req	Resp

Cloud Compute

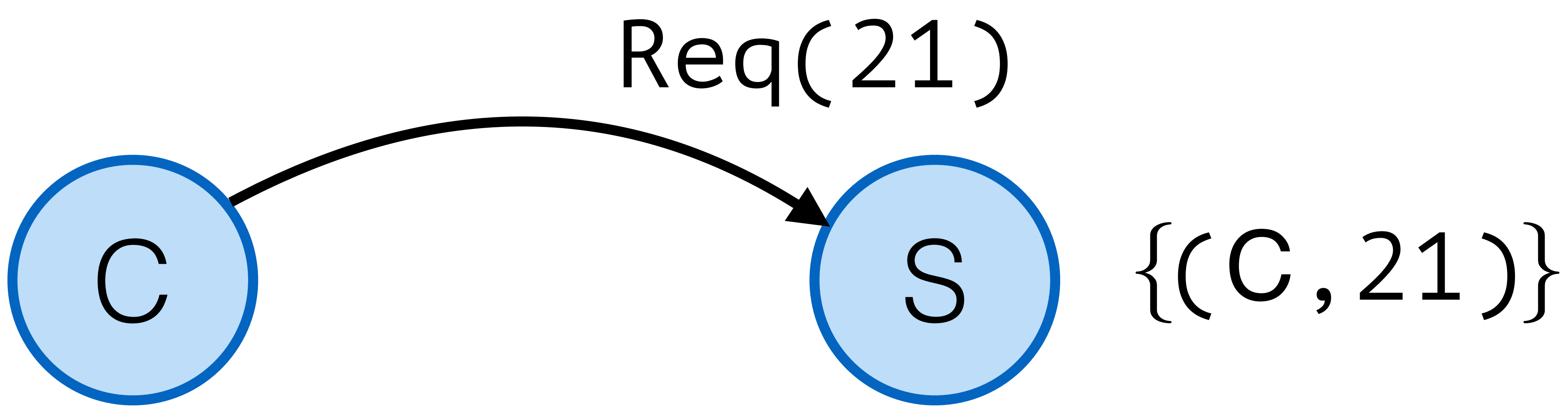


Send **`Req(n)`**

Precondition: none

Effect: none

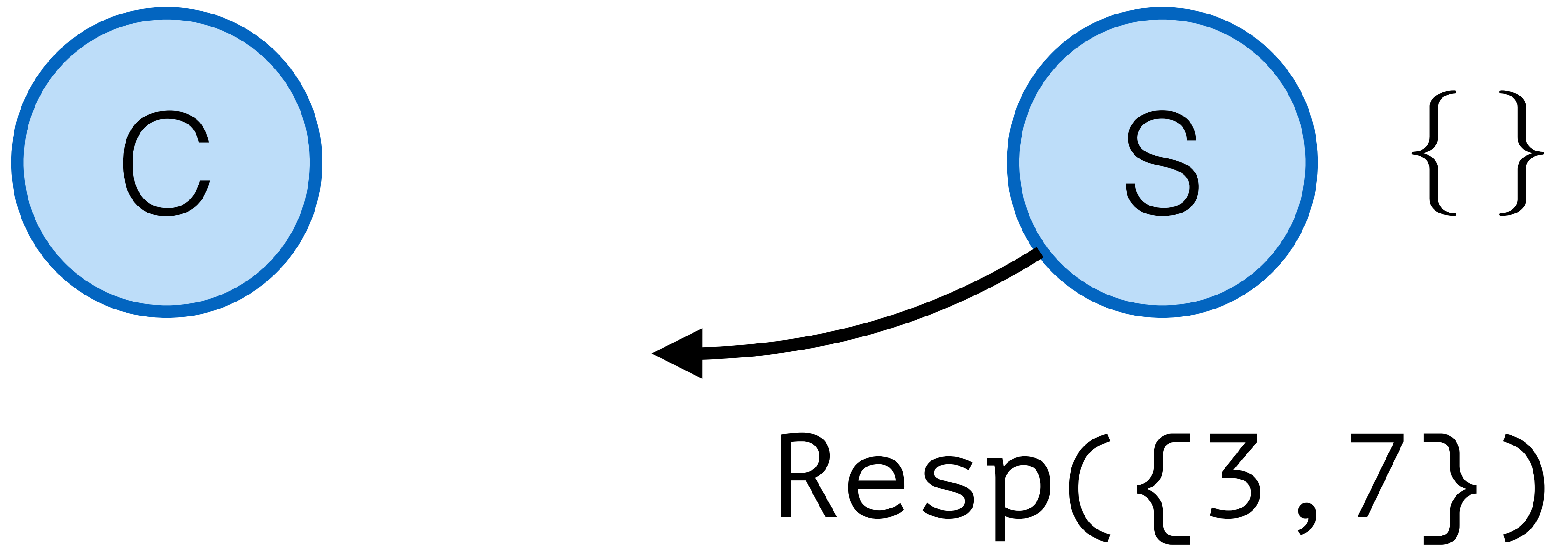
Cloud Compute



Receive `Req(n)`

Effect: add `(from, n)` to `out`

Cloud Compute



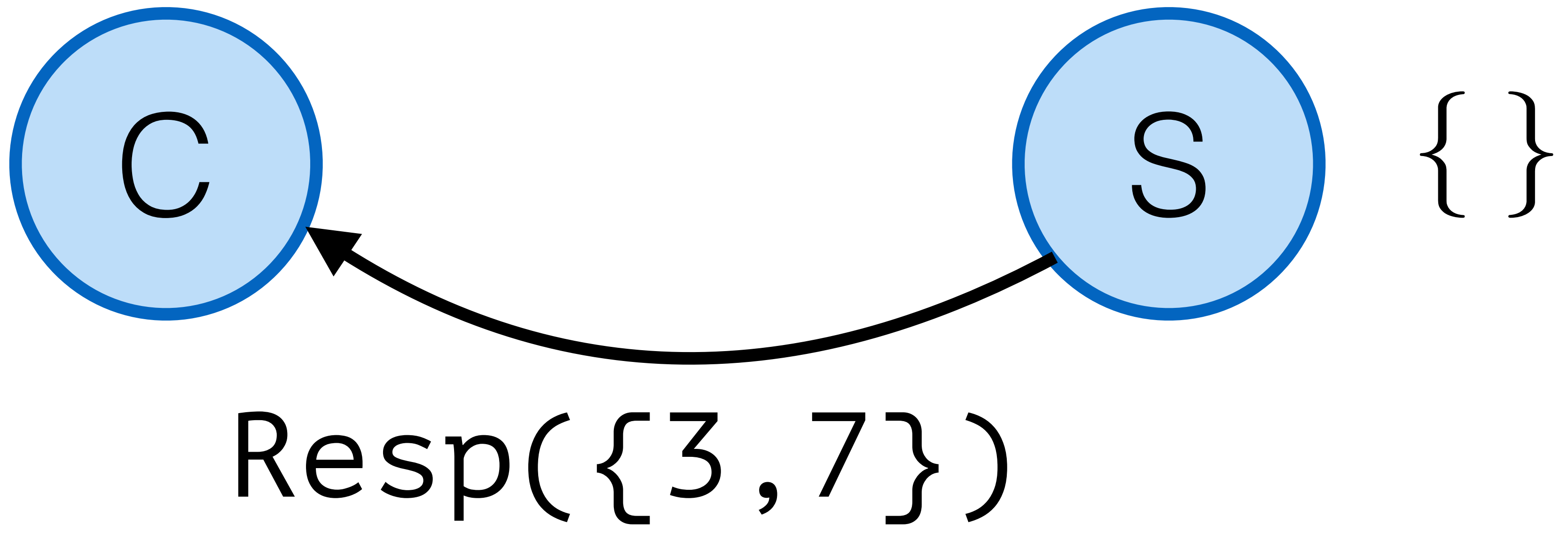
Send `Resp(n,l)`

Requires: `l == factors(n)`

`(n,to)` in `out`

Effect: removes `(n,to)` from `out`

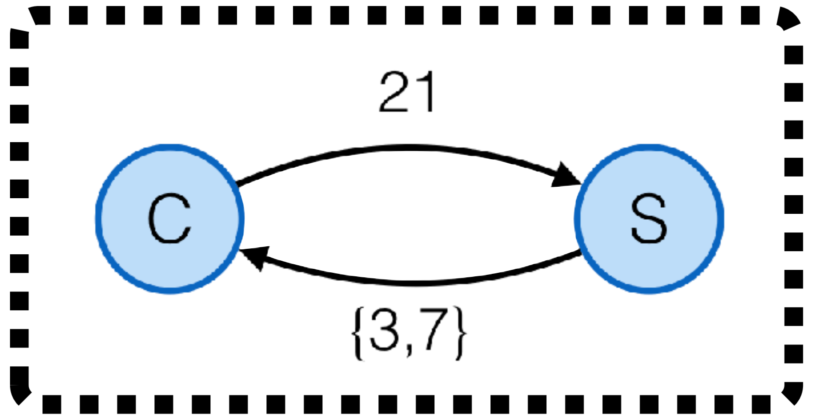
Cloud Compute



Recv `Resp(n,l)`

Effect: none

Cloud Compute Protocol



Messages: `Req(n) | Resp(n,s)`

State: `outstanding: Set<Msg>`

Transitions:

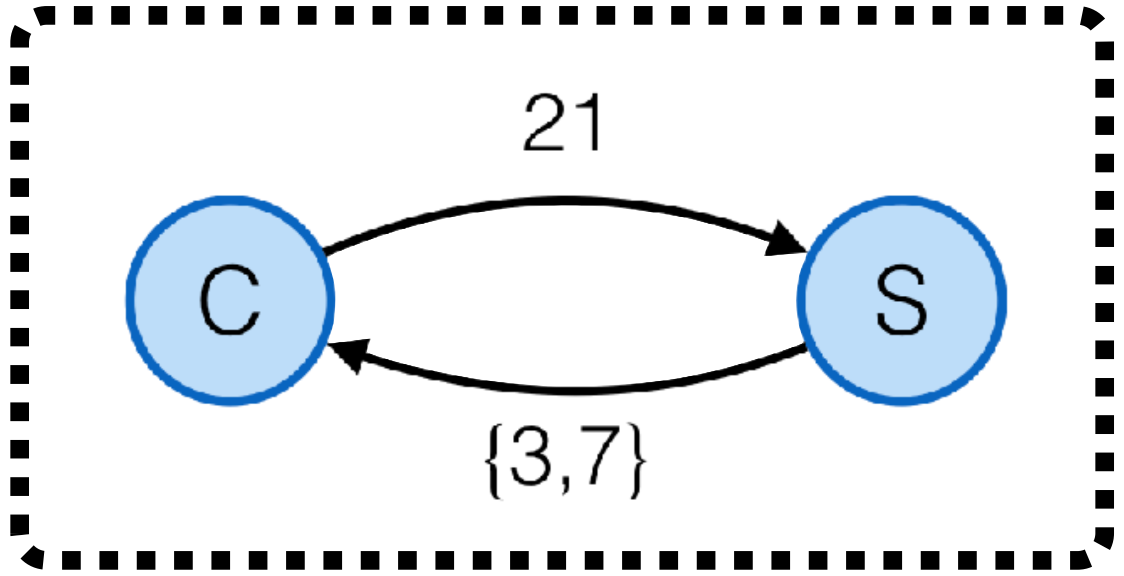
Sends: Req Resp

Receives: Req Resp

Disel: Distributed Separation Logic

Sergey, Wilcox, Tatlock [POPL'18]

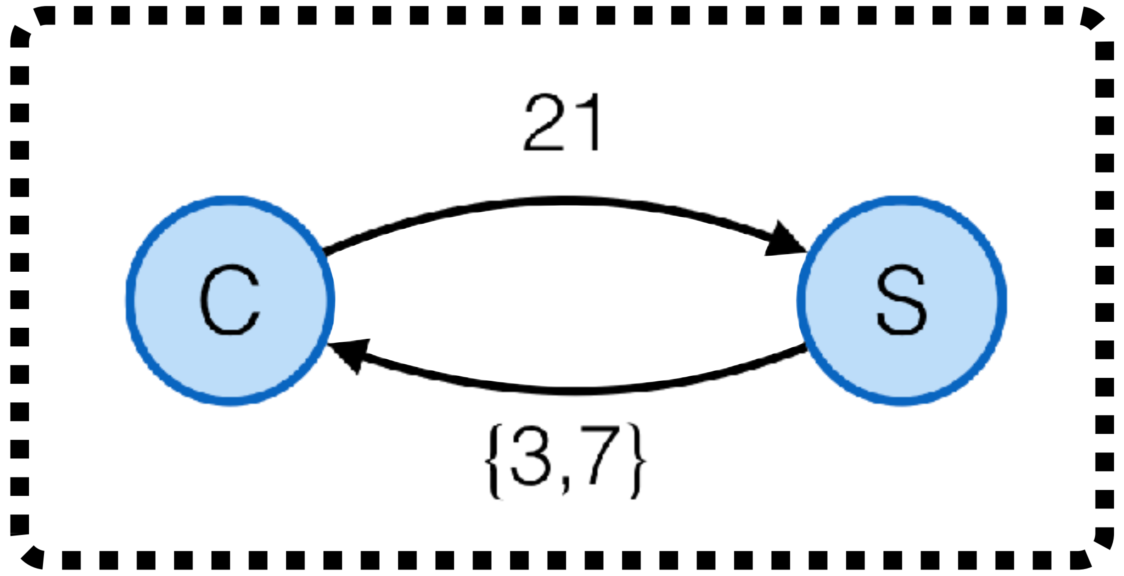
A Protocol-Aware program logic for distributed programs.



$\vdash_{n} \{\, P \,\}\ c\ \{\, Q \,\}$

protocol model

per-node implementation

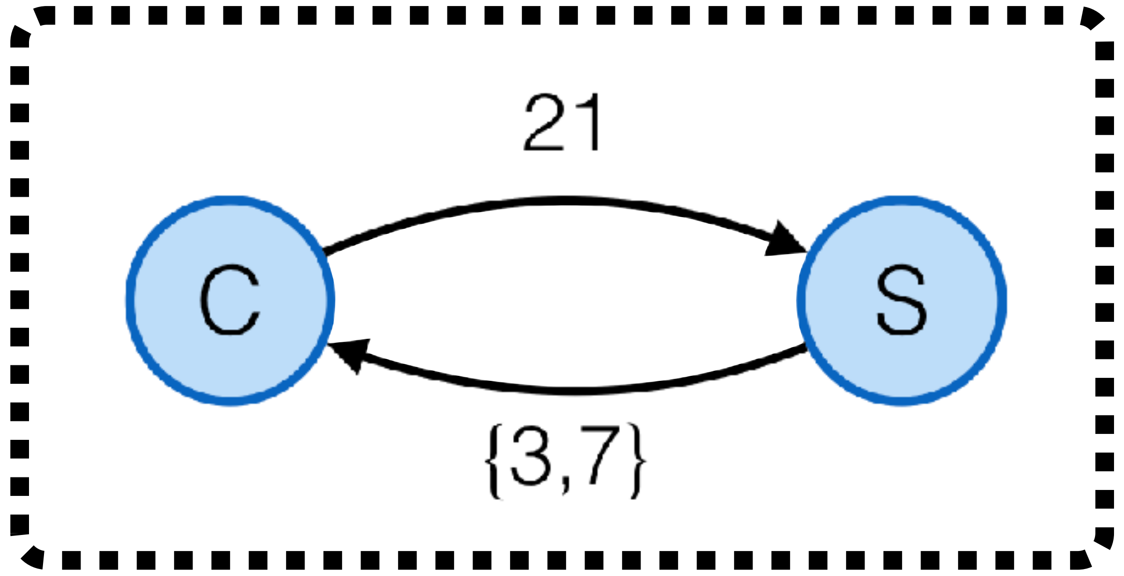


$$\vdash_{n} \{ P \}\ c\ \{ Q \}$$

If the initial state satisfies P,
then the code c is safe to run and the final state satisfies Q.

protocol model

per-node implementation



$$\vdash_{n} \{\, P \,\}\ c\ \{\, Q \,\}$$

If the initial state satisfies P,

then the code c *run by n*

does not violate the protocol on the left of $\vdash$,

and the final state satisfies Q.

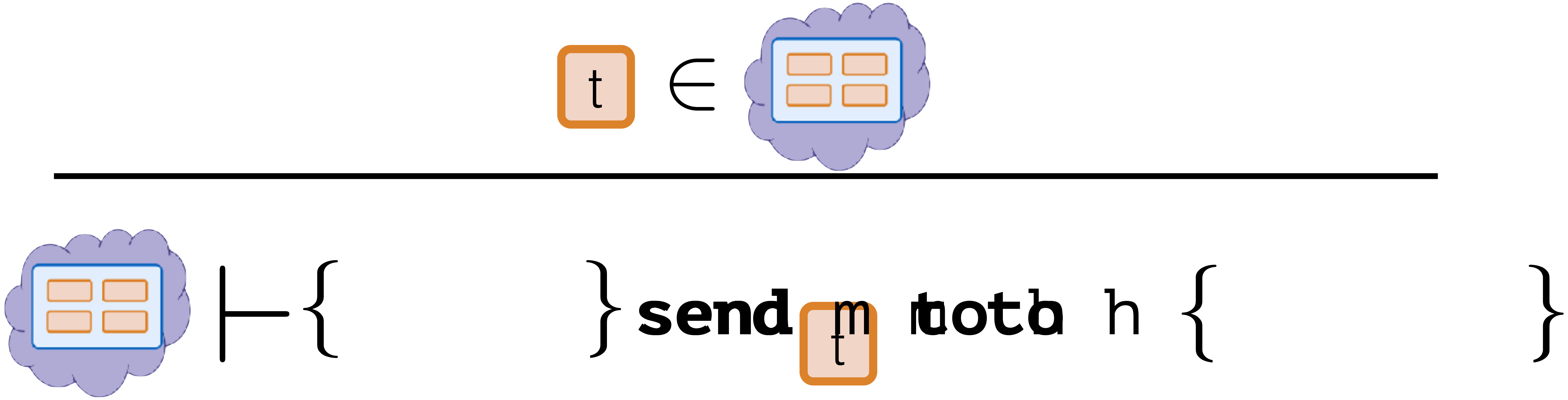
Disel by Example

Cloud Compute: Server

```
while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from
```

Precondition on **send** requires correct factors

Cloud Compute: Server




```
while true:
   (from, n) <- recv Req
   send Resp(n, factors(n)) to from
```

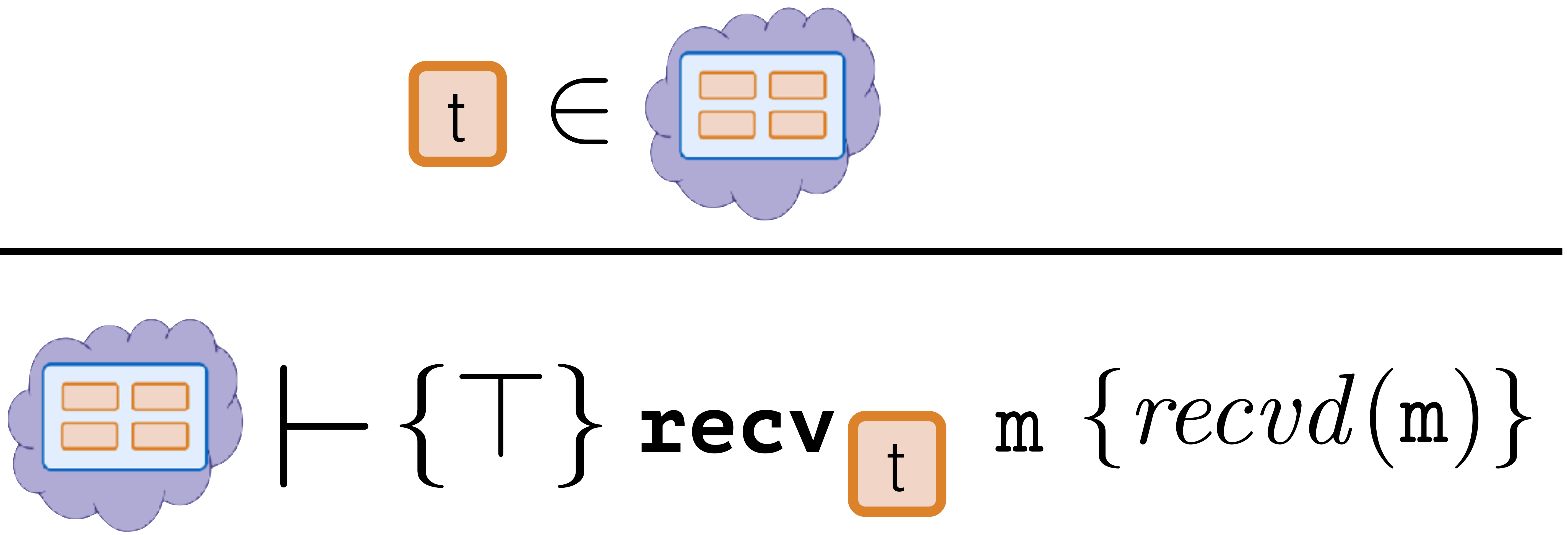
Precondition on **send** requires correct factors

Cloud Compute: Client

```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

recv doesn't ensure correct factors

Cloud Compute: Client

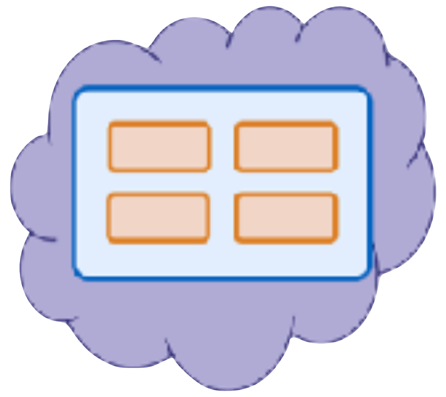



```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

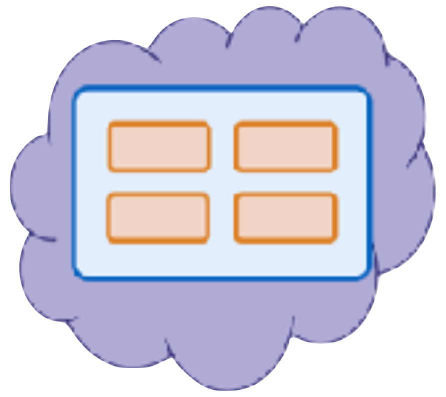
recv doesn't ensure correct factors

Protocol Invariants

Property of the **Protocol**, proven ***independently*** of a program *c*

 $\vdash \{P\}\ c\ \{Q\} \qquad I$ inductive

---

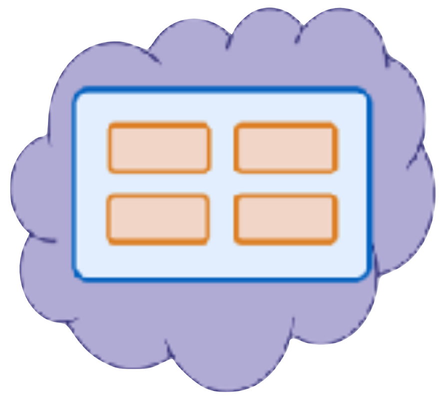
 $' \quad \vdash \{P \wedge I\}\ c\ \{Q \wedge I\}$

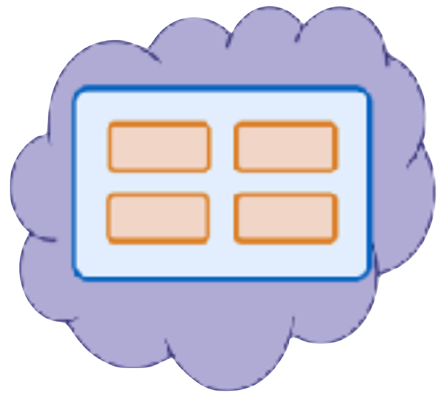
Protocol where every state satisfies I

Cloud Compute: Client

Inductive Invariant I ⇒

“Responses contain correct factors”

t $\in$ 

 $' \vdash \{\top\}\ \mathbf{recv}_{t}\ \mathtt{m}\ \{recvd(\mathtt{m})\}$

```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

Now **recv** ensures correct factors

Cloud Compute: More Clients

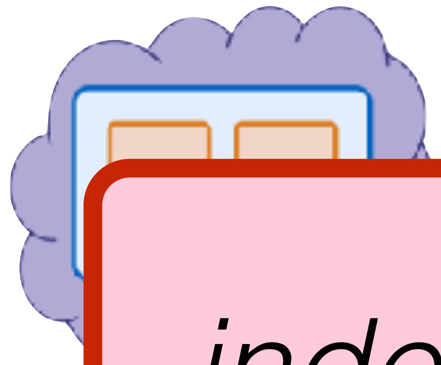
```
send Req(21) to server₁
send Req(35) to server₂
(_, ans₁) <- recv Resp
(_, ans₂) <- recv Resp
assert ans₁ ∪ ans₂ == {3, 5, 7}
```

Compute Instance 1

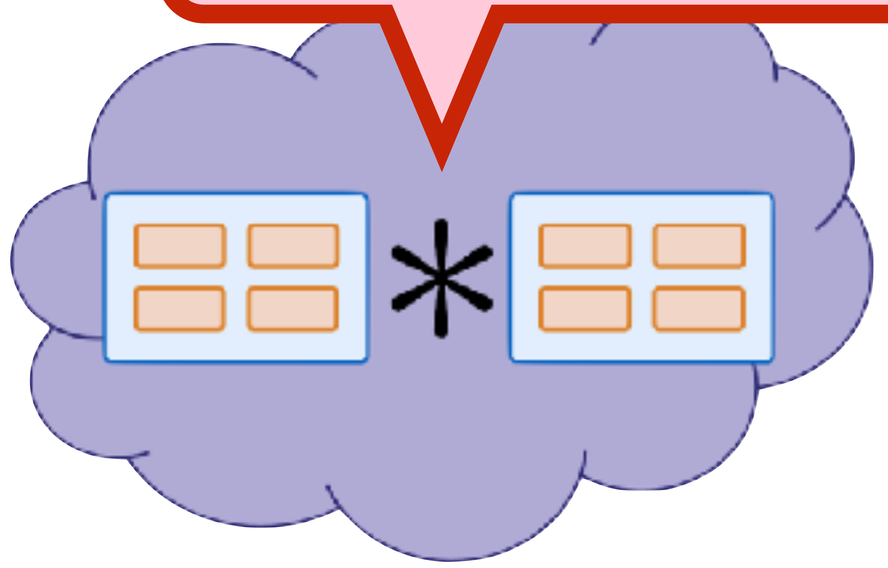
Compute Instance 2

Same protocol enables verification

Horizontal Composition: Frame Rule

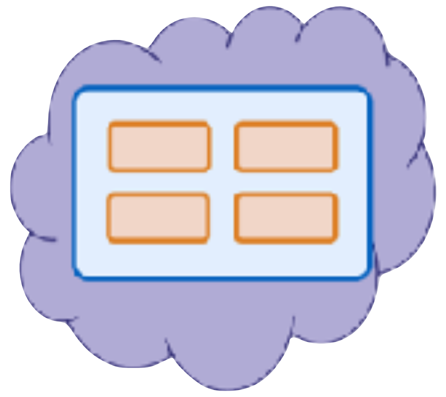
 $\vdash \{P\}\ c\ \{Q\} \qquad R \text{ stable}$

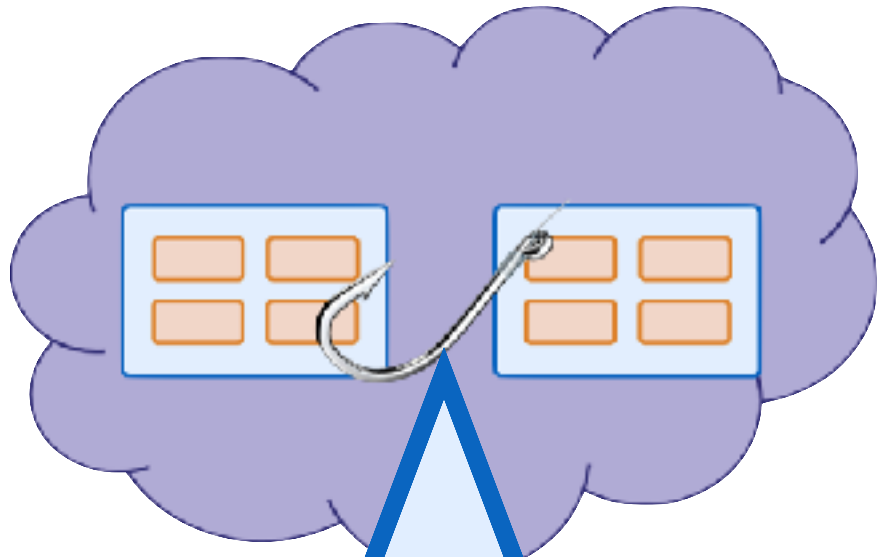
independent protocols

 $\vdash \{P * R\}\ c\ \{Q * R\}$

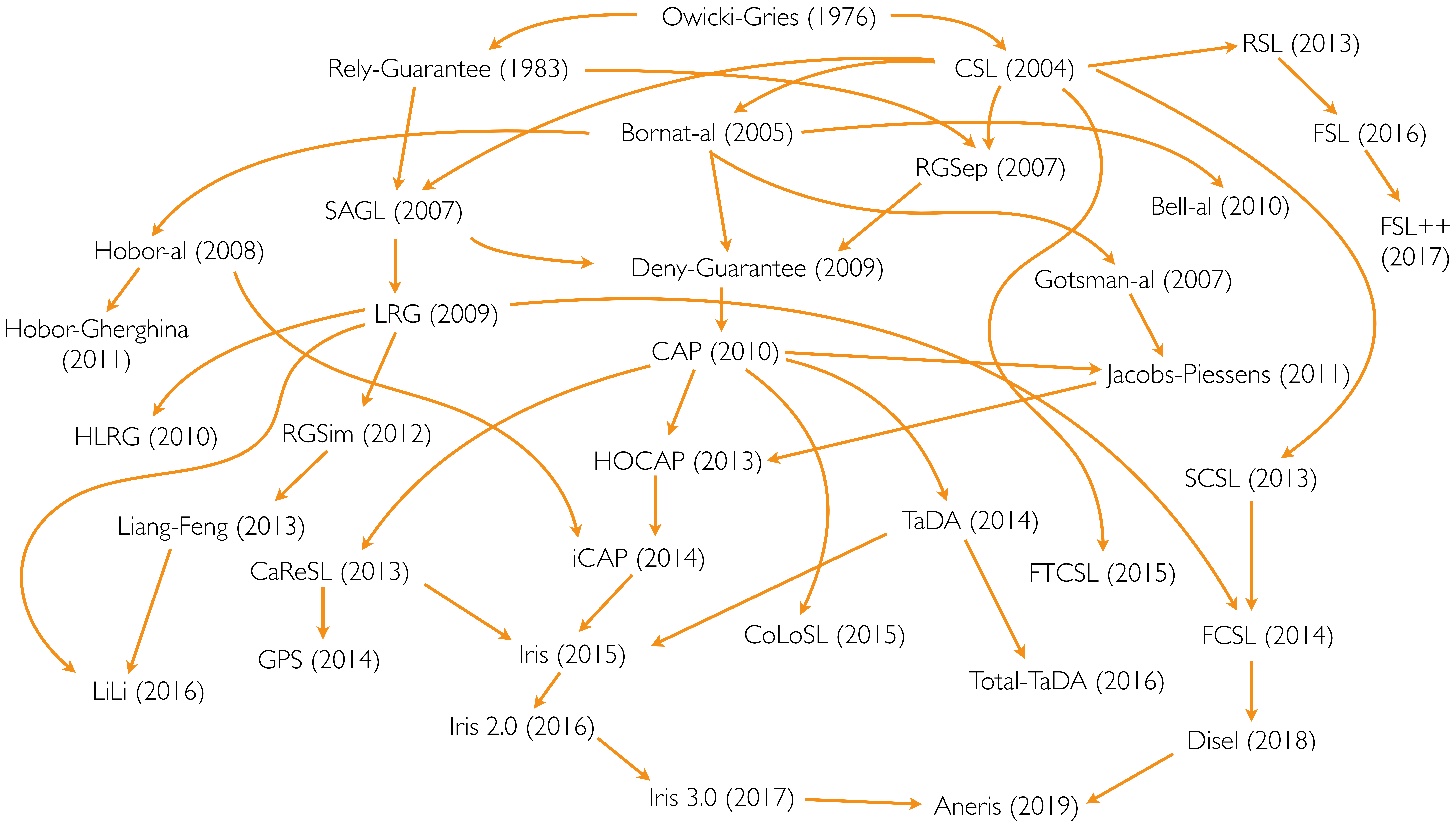
Reuse invariants from component protocols

Horizontal Composition: Frame Rule with Hooks

 $\vdash \{P\}\ c\ \{Q\} \qquad R \text{ stable}$

 $\vdash \{P * R\}\ c\ \{Q * R\}$

Allows one protocol to restrict another



Owicki-Gries (1976)
Rely-Guarantee (1983)
CSL (2004)
RSL (2013)
Bornat-al (2005)
FSL (2016)
RGSep (2007)
SAGL (2007)
Bell-al (2010)
FSL++ (2017)
Hobor-al (2008)
Deny-Guarantee (2009)
Gotsman-al (2007)
Hobor-Gherghina (2011)
LRG (2009)
CAP (2010)
Jacobs-Piessens (2011)
HLRG (2010)
RGSim (2012)
HOCAP (2013)
SCSL (2013)
Liang-Feng (2013)
TaDA (2014)
CaReSL (2013)
iCAP (2014)
FTCSL (2015)
GPS (2014)
Iris (2015)
CoLoSL (2015)
FCSL (2014)
LiLi (2016)
Total-TaDA (2016)
Iris 2.0 (2016)
Disel (2018)
Iris 3.0 (2017)
Aneris (2019)

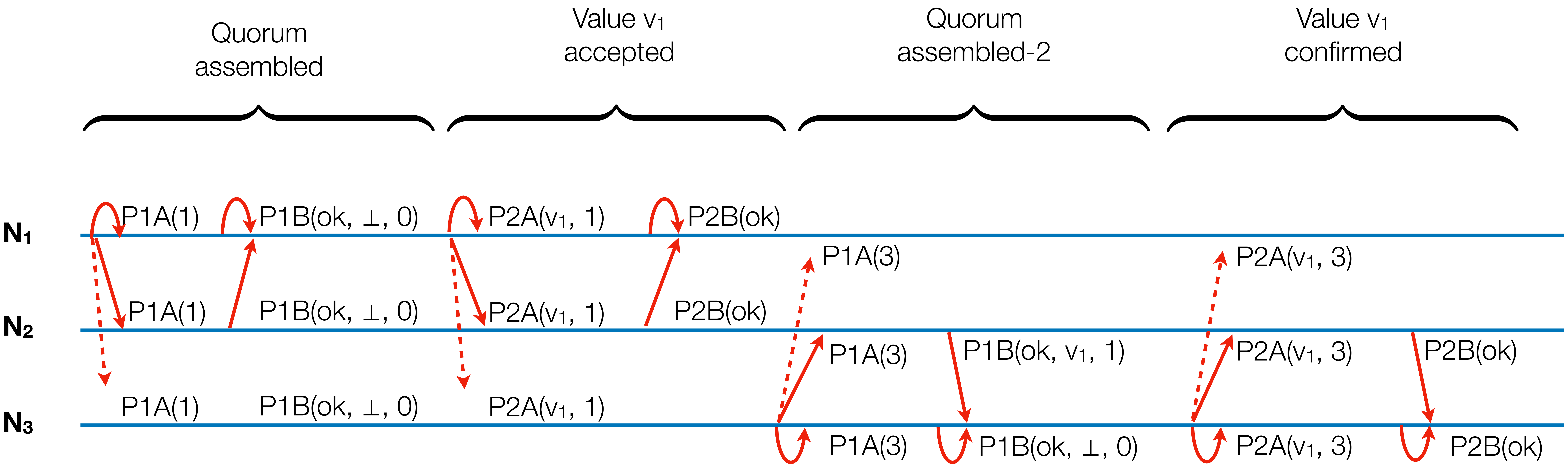
Disel: Case Studies and Applications

- Cloud Calculator + Variations
- Two Phase Commit: Invariants, Clients (Replicated Logging)
- Paxos Consensus Protocol and its clients
- Extraction and trusted shim implementation (verified systems can be *run*)

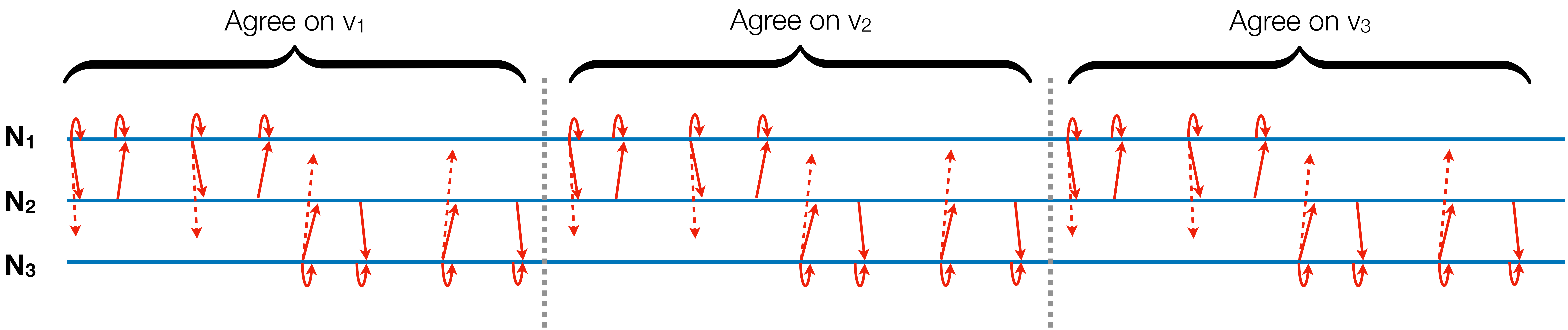
Paxos Consensus Protocol

- A practical *fault-tolerant distributed consensus algorithm,* allowing the *majority* of distributed parties have to **agree on a single value**
- Invented by Leslie Lamport in 1990, published in 1998
- Nowadays used everywhere: Google (Bigtable, Chubby), IBM, Microsoft
- Requires *multiple rounds* of interaction

Single Run of Paxos for Agreeing on $\mathbf{v}_1$



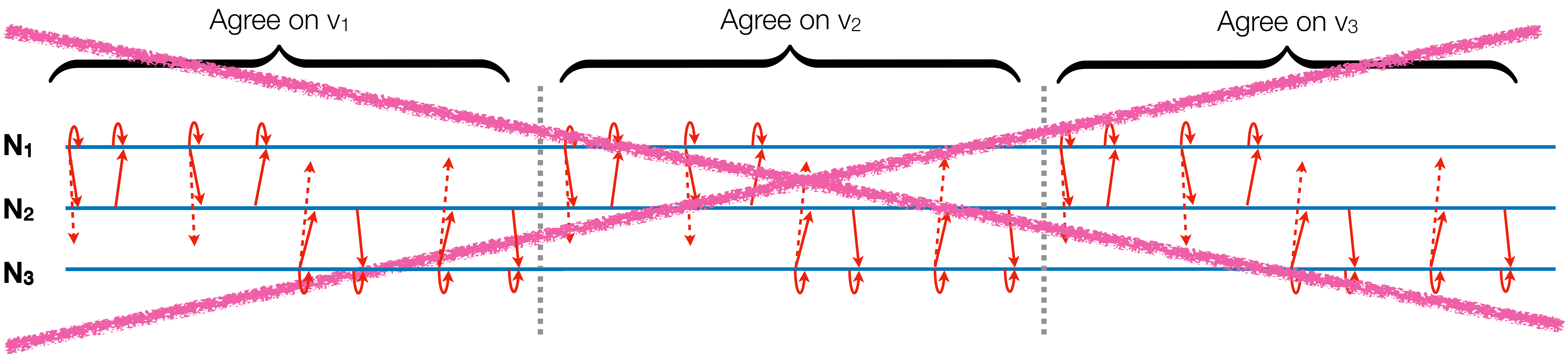
From Single-Run Paxos to Multi-Paxos



From the perspective of N_1:

$\{\ log = []\ \}$ **N₁**`.run_paxos(v1);` $\{\ log = [v_1]\ \}$ **N₁**`.run_paxos(v2);` $\{\ log = [v_1; v_2]\ \}$ **N₁**`.run_paxos(v3);` $\{\ log = [v_1; v_2; v_2]\ \}$

From Single-Run Paxos to Multi-Paxos



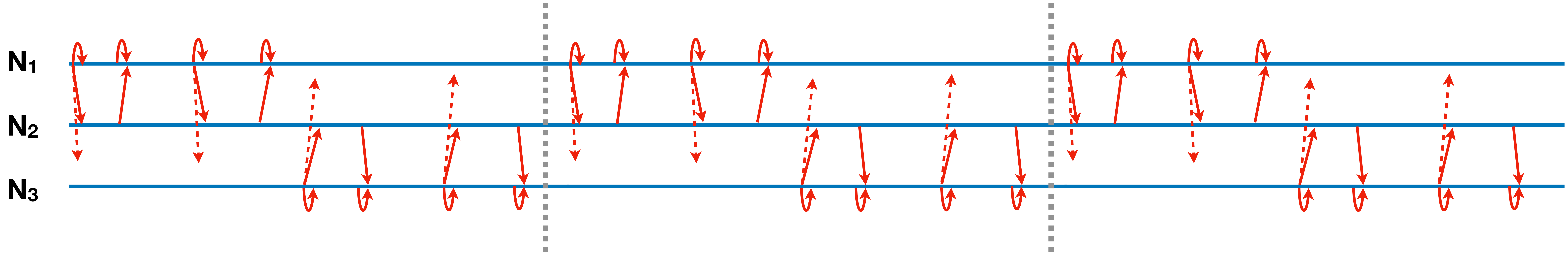
$\{\ log = [\,]\ \}$

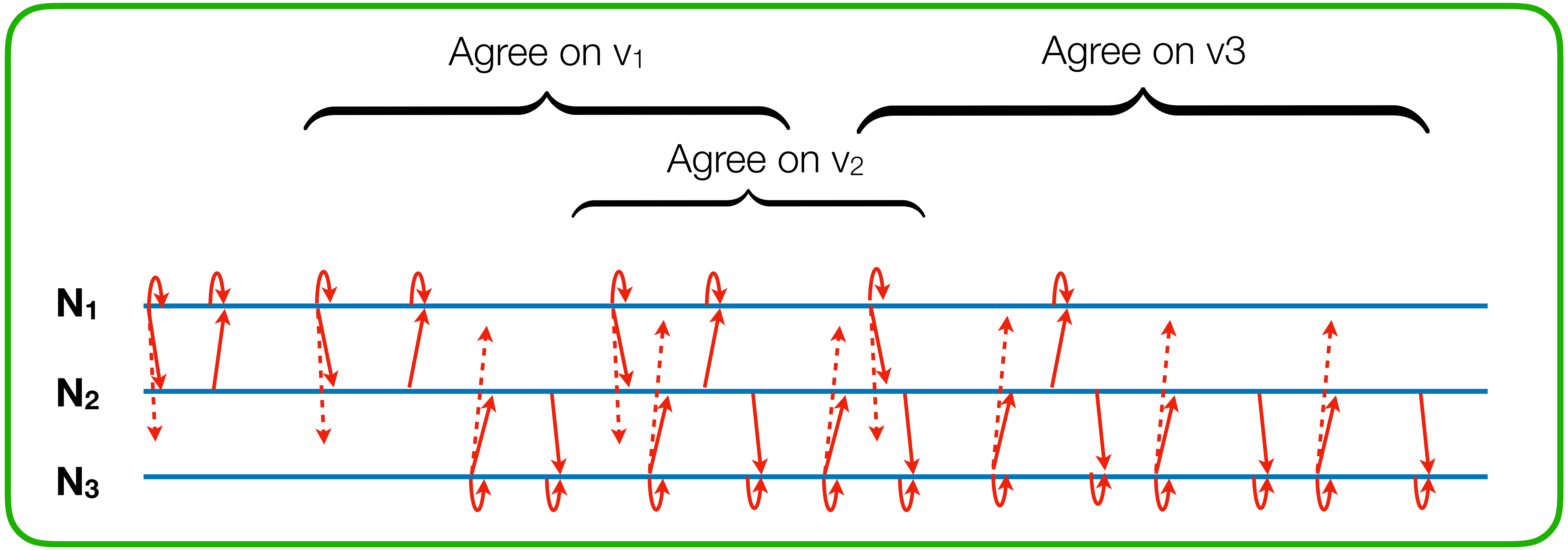
This is not how Multi-Paxos works!

```
N1.run_paxos(v1);

N1.run_paxos(v2);

N1.run_paxos(v3);
```

$\{\ log = [v_1; v_2; v_2]\ \}$

From Single-Run Paxos to Multi-Paxos





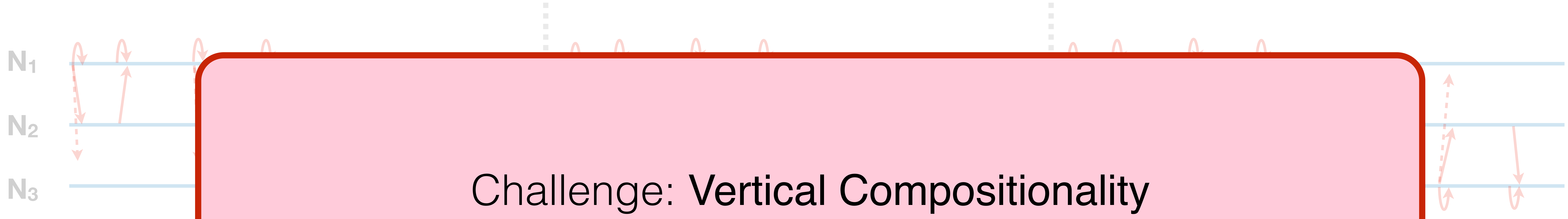
Multi-Paxos

$\{ \ log = [\,] \ \}$

```
N1.run_paxos(v1);

N1.run_paxos(v2);

N1.run_paxos(v3);
```

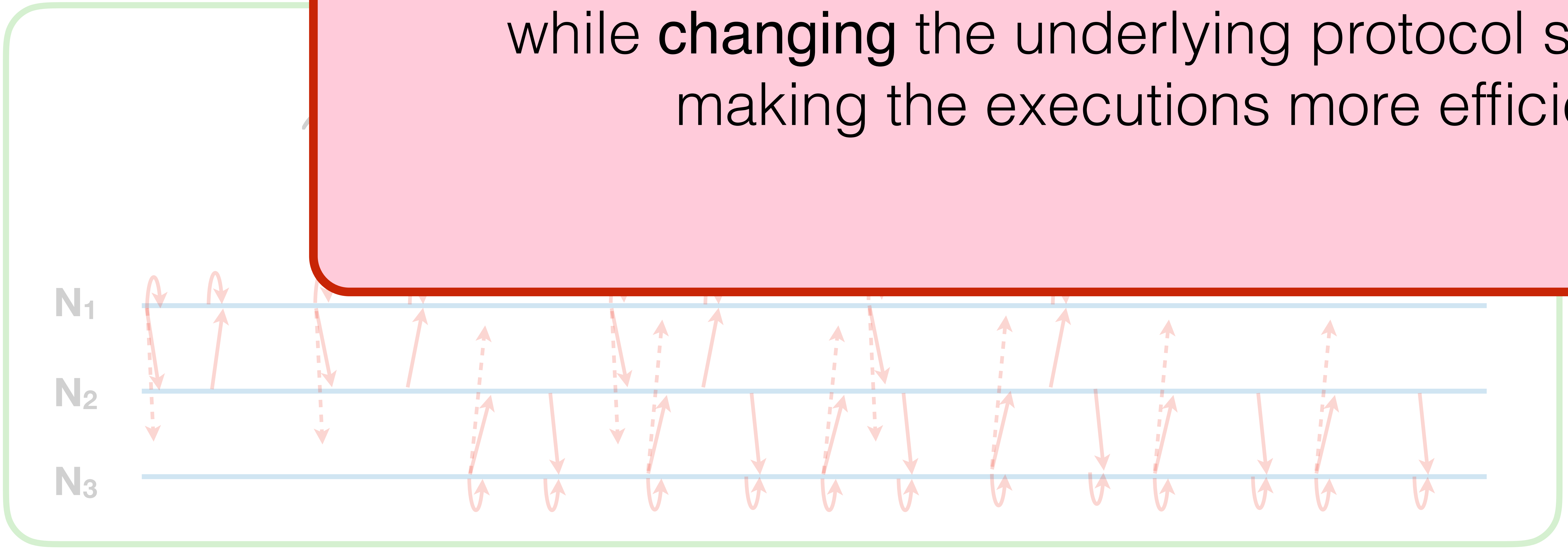
$\{ \ log = [v_1; v_2; v_2] \ \}$

From Single-Run Paxos to Multi-Paxos



Challenge: **Vertical Compositionality**

Need to **preserve** logical specifications
while **changing** the underlying protocol semantics,
making the executions more efficient.



Multi-Paxos

[] }

…os(v1);

…os(v2);

N1.run_paxos(v3);

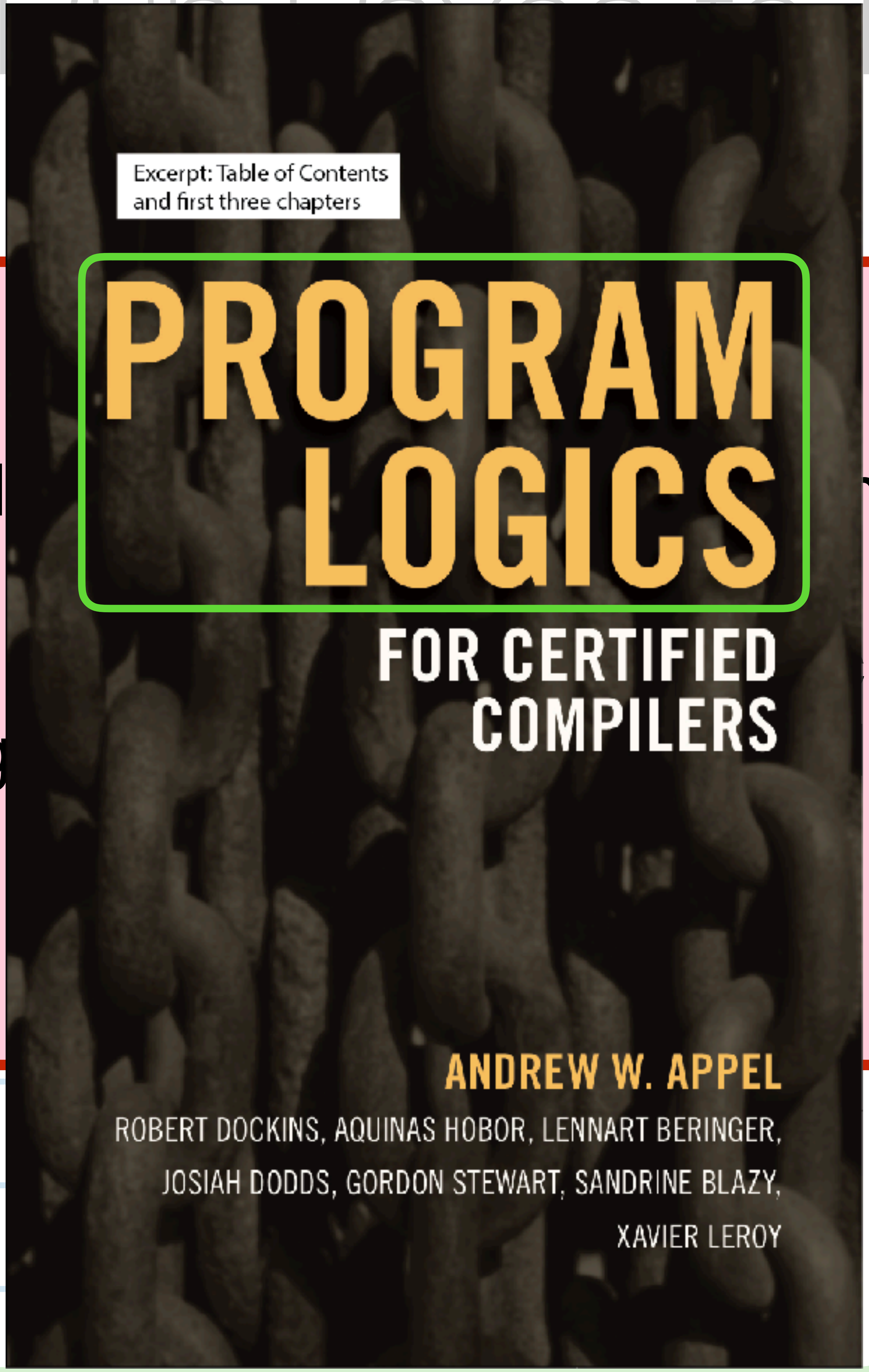
{ *log* = [v_1; v_2; v_2] }

From Single- … Multi-Paxos

N1
N2
N3

Chal … ality

Need … ations
while chang … semantics,



[] }

…os(v_1);

…os(v_2);

N_1.run_paxos(v_3);

$\{ log = [v_1; v_2; v_2] \}$

N1
N2
N3

Multi-Paxos

From Single- … Multi-Paxos

N1

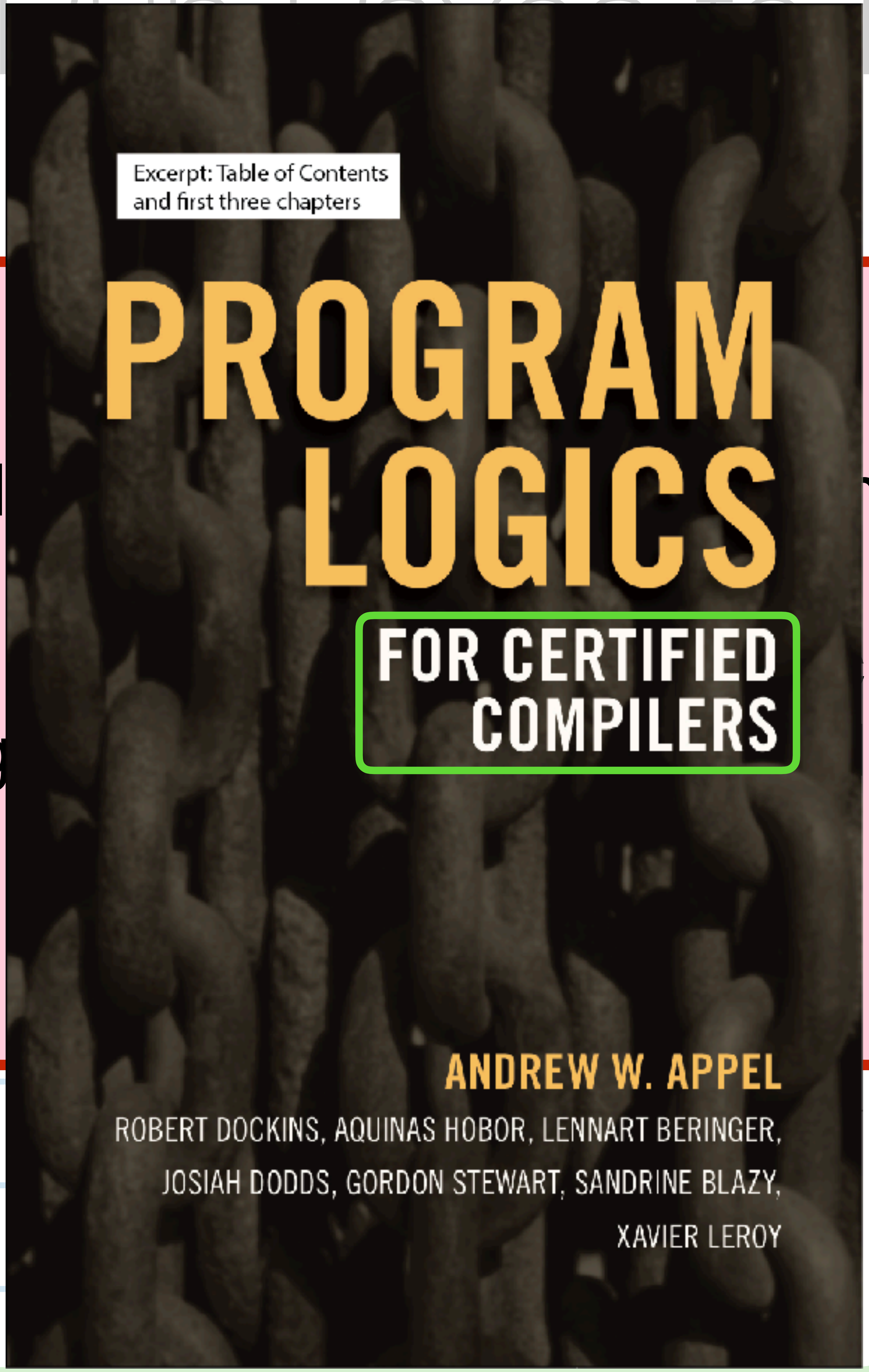
N2

N3

Chal … ality

Need … ations

while chang … semantics,



[] }

```
…os(v1);
…os(v2);
N1.run_paxos(v3);
```

$\{ \, log = [v_1; v_2; v_2] \, \}$

N1

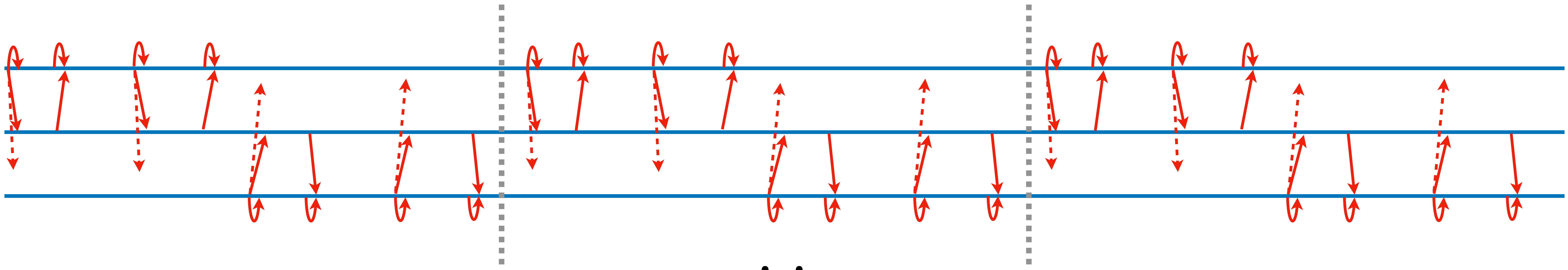
N2

N3

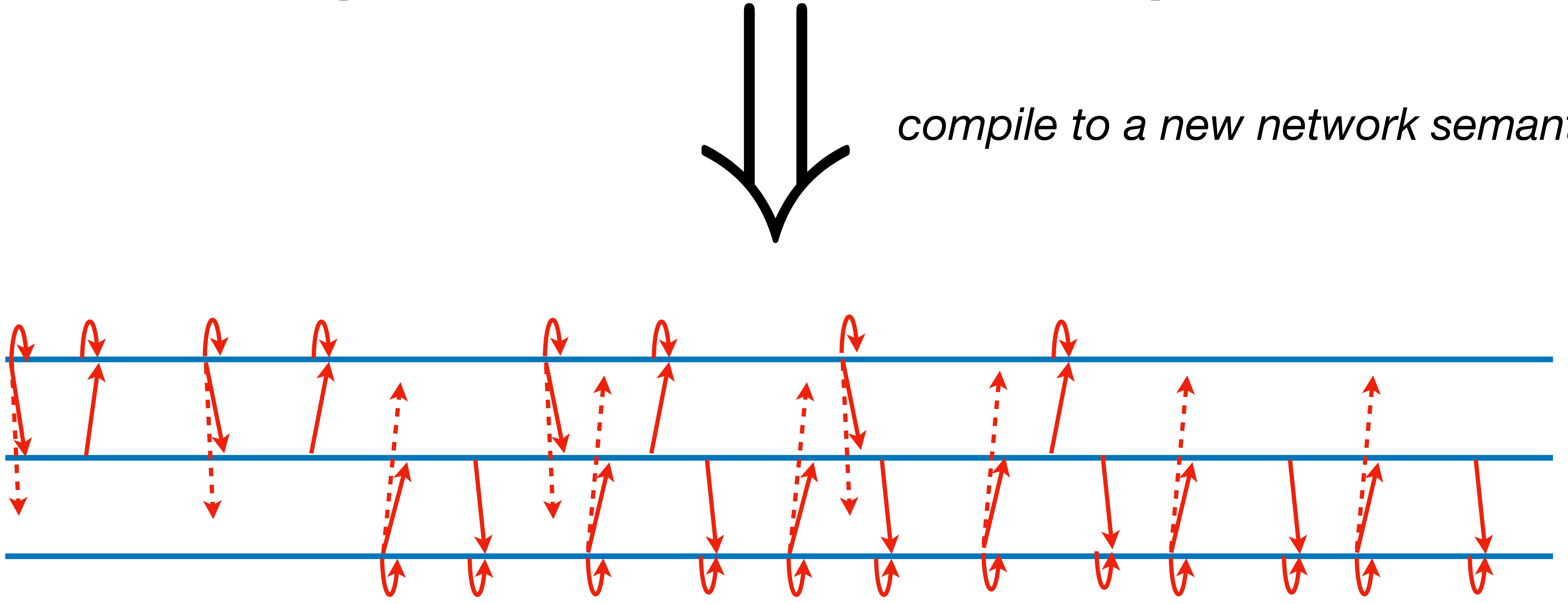
Multi-Paxos

“Compiling” Single-Run Paxos to Multi-Paxos

García-Pérez, Gotsman, Meshman, Sergey [ESOP’18]

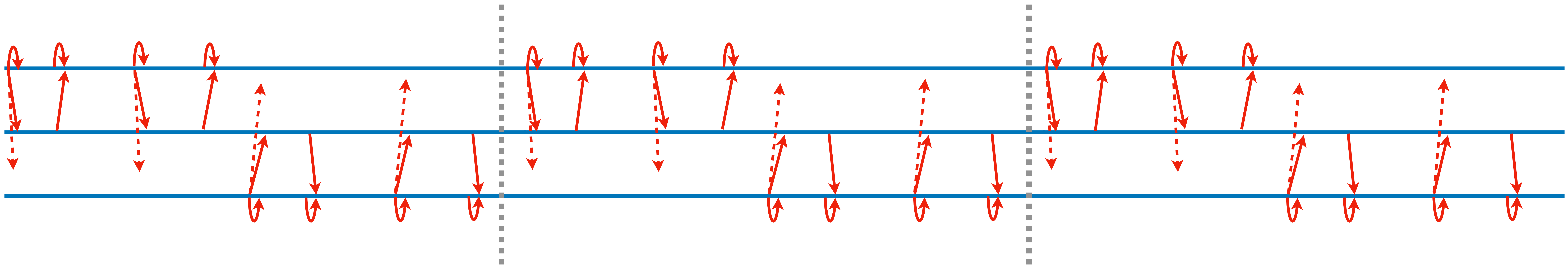


compile to a new network semantics

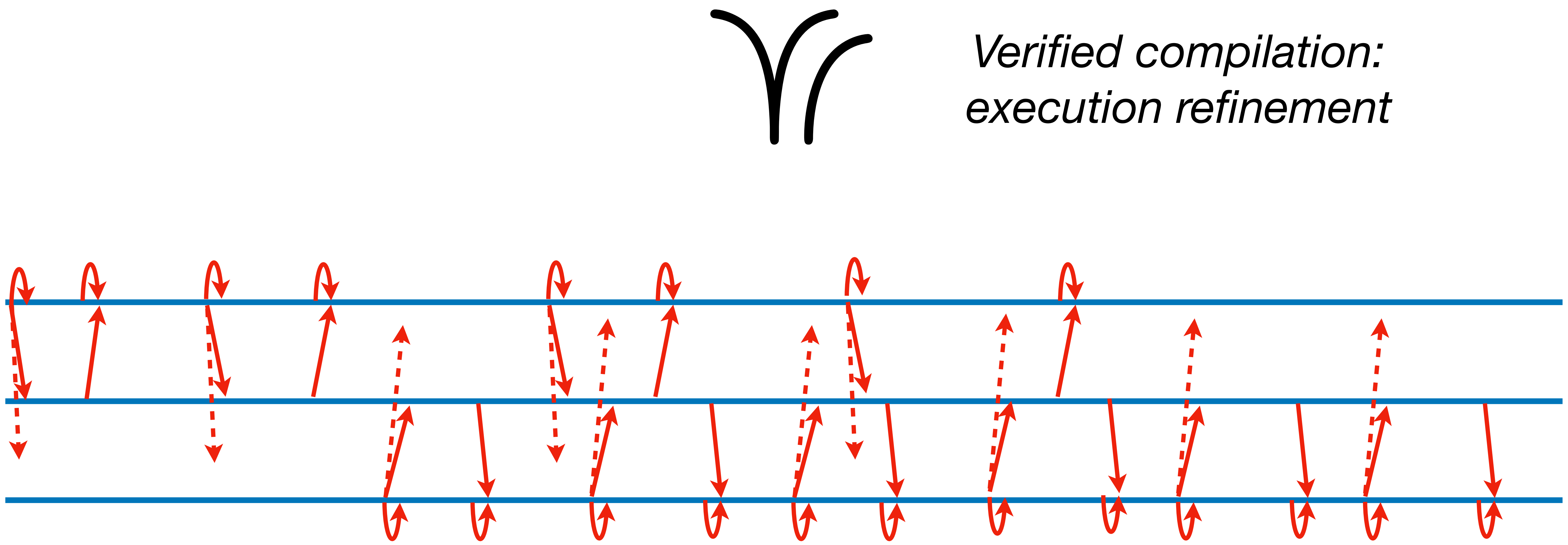


“Executing Paxos on Different Architecture”

“Compiling” Single-Run Paxos to Multi-Paxos

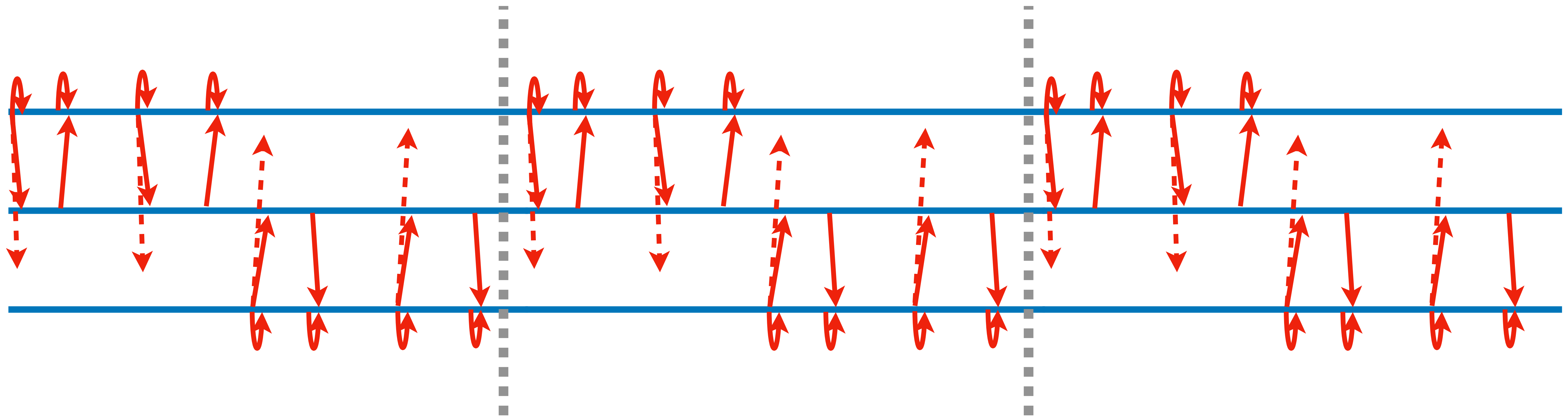


Verified compilation:
execution refinement



“Executing Paxos on Different Architecture”

“Compiling” Single-Run Paxos to Multi-Paxos



$\models$

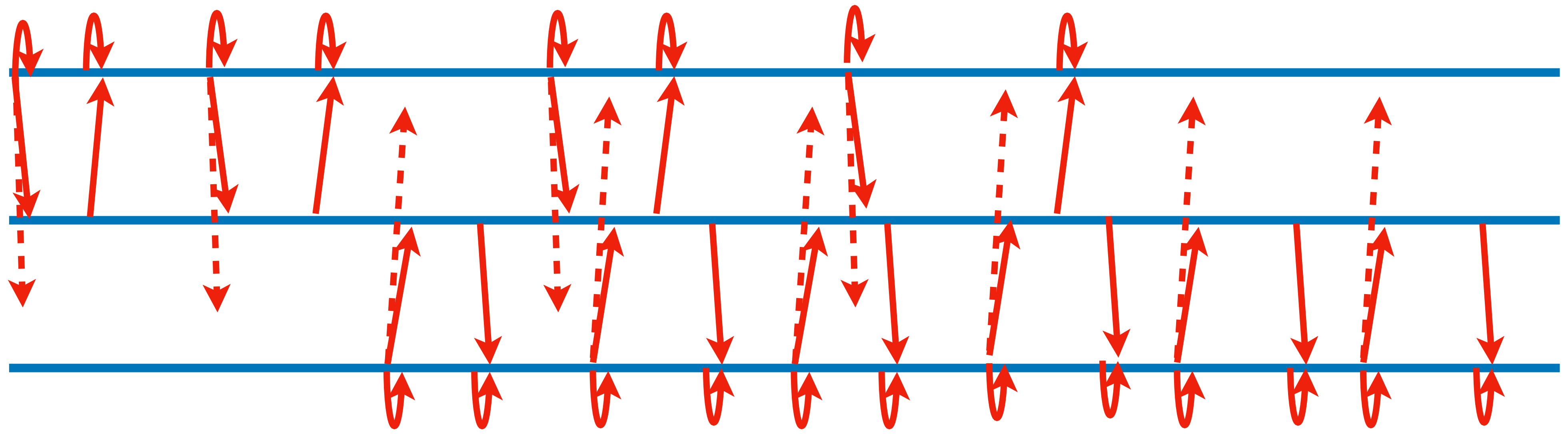
$\{\ log = []\ \}$

```
N₁.run_paxos(v₁);
N₁.run_paxos(v₂);
N₁.run_paxos(v₃);
```

$\{\ log = [v_1; v_2; v_2]\ \}$

$\curlyvee$

$\Downarrow$



$\boxed{\models}$

$\{\ log = []\ \}$

```
N₁.run_paxos(v₁);
N₁.run_paxos(v₂);
N₁.run_paxos(v₃);
```

$\{\ log = [v_1; v_2; v_2]\ \}$

Composing Proofs about Programs

Paradigm	Challenges	Tools
Functional	higher-order functions	Types and Semantics
Imperative	state	Program Logics
Concurrent	interference	Program Logics + Subjectivity
Distributed	asynchronous message delivery unbounded delays lack of synchronisation network faults and partitions	Program Logics + Protocols + Compiler optimisations

Research in Programming Languages

- Object-oriented software development
- Models and Modeling
- Language Design
- Parallelism
- Program Logics
- Applications (systems, networking, AI/ML)
- Analysis of Concurrent Programs
- Object-Oriented Programming
- Correctness
- Verification
- Type Systems
- Program Analysis
- Components and APIs
- Garbage Collection
- Array Processing
- Semantics of concurrent programs
- Low-level compiler optimisations
- Parsing
- Resource management
- Compiler optimisations

In Conclusion

Reusable and *scalable* verification efforts require proofs to *compose* along with the programs.

Search for composition unveils *mathematics behind a program*.

PL research field provides *the tools* to do so.

Looking Ahead

- Compositional deductive program synthesis

  Polikarpova, Sergey [POPL'19]

- Layered verification of blockchain consensus implementations

  Pirlea, Gopinathan, Sergey [CPP'18, CoqPL'19]

- Abstract specifications for Scilla smart contracts

  Sergey, Nagaraj, Johannsen, Kumar, Trunov, Chan [WIP'19]

Acknowledgements

Dave Clarke

Aleks Nanevski

Olivier Danvy

My amazing collaborators:

Nadia Polikarpova, Dominique Devriese, Aquinas Hobor, Jan Midtgaard, Peter O'Hearn, Matt Might, David Van Horn, Simon Peyton Jones, Dimitrios Vytiniotis, Nikos Gorogiannis, Elvira Albert, Albert Rubio, Amrit Kumar, Vaivaswatha Nagaraj, Jacob Johannsen, Anton Trunov, Álvaro Garcia Pérez, Prateek Saxena, Anindya Banerjee, Zach Tatlock, Germán Delbianco, David Darais, Anton Podkopaev, Kristoffer Just Andersen, George Pîrlea, Kiran Gopinathan, and James R. Wilcox.

Thank you!