State Transition System alternative to Linearizability

Ilya Sergey

joint work (in progress) with
Aleks Nanevski, Anindya Banerjee,
Ruy Ley-Wild and Germán Delbianco

institute imdea software

Concurrency Yak
San Diego, 21 January 2014

Linearizability

Herlihy-Wing:TOPLAS90

- Golden standard for canonical specifications
- A tool for granularity abstraction

Canonical Specifications

$\{ S = xs \}$ `push(x)` $\{ S = x :: xs \}$

$\{ S = xs \}$ `pop()` $\{ \quad res = \text{Nothing} \wedge S = \textbf{Nil}$
$\vee\ \exists \text{x}, \text{xs}.\ res = Just(x) \wedge S = x :: xs \wedge$
$S' = xs \}$

Suitable for sequential case

Canonical Specifications

$\{ S = xs \}$ `push(x)` $\{ S = x :: xs \}$

$\{ S = xs \}$ `pop()` $\{ \quad res = \text{Nothing} \wedge S = \mathbf{Nil}$
$\vee\ \exists \mathrm{x, xs.}\ res = Just(x) \wedge S = x :: xs \wedge$
$S' = xs \}$

Bad for concurrent use:
<u>not</u> *stable* under interference

Stable Concurrent Specifications

$\forall\, P : Elem \to Prop.$

$\{\, P(x) \,\}$ `push(x)` $\{\, \mathbf{true} \,\}$

$\{\, \mathbf{true} \,\}$ `pop()` $\{\, res = Nothing \lor \exists x.\, res = Just(x) \land P(x) \,\}$

Not a canonical spec:
the same one holds for
queues, sets, bags

Svendsen-al:ESOP13

Turon-al:ICFP13

Making things worse

$\forall\, P : Elem \to Prop.$

$\{\, P(x) \,\}$ `push(x)` $\{\, \mathbf{true} \,\}$

$\{\, \mathbf{true} \,\}$ `pop()` $\{\quad res = Nothing$
$\quad \lor\ \exists \mathrm{x}.\ res = Just(x) \land P(x) \,\}$

$\{\, P(\mathrm{x}) \,\}$ `contains(x)` $\{\, \mathrm{res} = ??? \,\}$

Linearizability to the rescue

canonical spec = sequential spec *

$\{ S = xs \}$ `push(x)` $\{ S = x :: xs \}$

$\{ S = xs \}$ `pop()` $\{ \quad res = \mathrm{Nothing} \wedge S = \mathbf{Nil}$
$\vee\ \exists \mathrm{x}, \mathrm{xs}.\ res = \mathit{Just}(x) \wedge S = x :: xs \wedge$
$S' = xs \}$

$\{ S = xs \}$ `contains(x)` $\{ res = (\mathrm{x} \in \mathrm{xs}) \wedge S' = xs \}$

* or *atomic* operations with the sequential spec above

Can we provide a convenient *concurrent* specification for `contains()` without appealing to linearizability?

(probably, it will also be more straightforward to prove)

Reasoning with *hindsight*

O'Hearn-al:PODC10

`contains(x) =` **`true`**	`x` was in the contents of the stack *S* at some moment *before* or *during* the execution of `contains()`
`contains(x) =` **`false`**	`x` was not in the contents *S* at some moment *before* or *during* the execution of `contains()`

Hindsight is a property of a resource's past history

Formalising the idea of
hindsight
for a large class of
concurrent protocols.

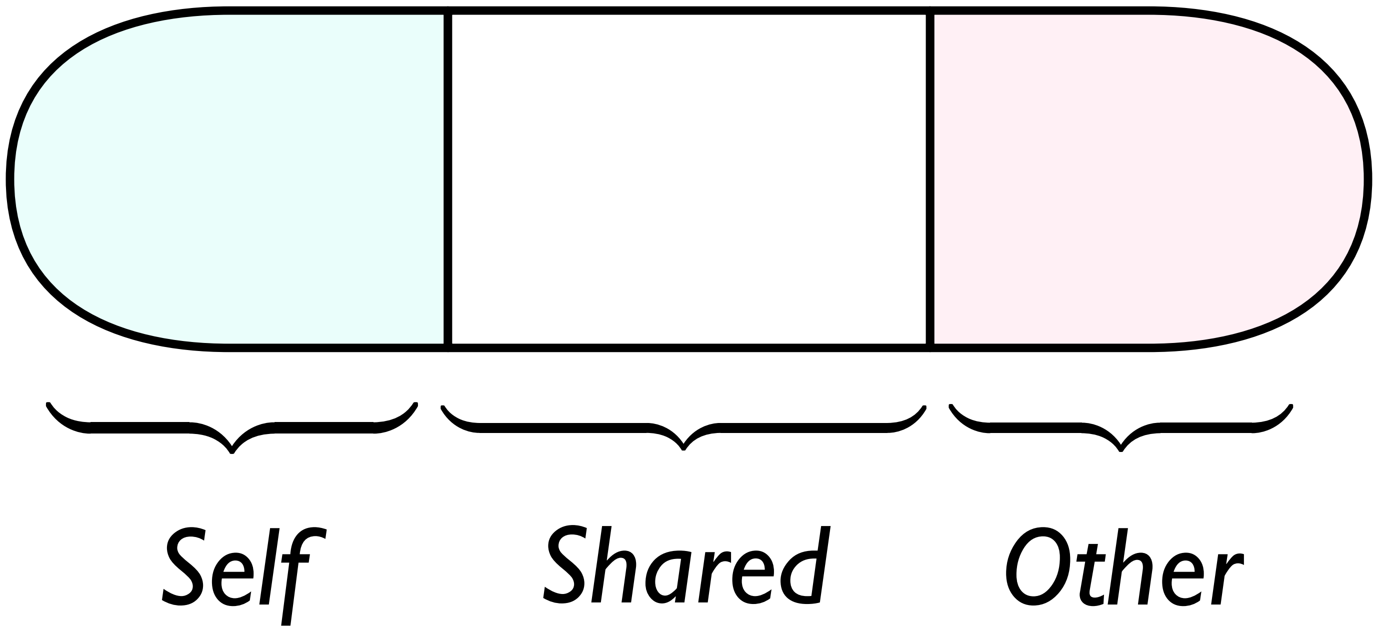
A model for resources with histories

- Resources represented by *State-Transition Systems* (STS)
- Transitions define *Rely/Guarantee* of a resource
- Auxiliaries are *ghost* parts of the resource's state

DinsdaleYoung-al:ECOOP10, O'Hearn-al:PODC10, LeyWild-Nanevski:POPL13, Turon-al:POPL13, Turon-al:ICFP13, Svendsen-al:ESOP13, Svendsen-Birkedal:ESOP14, Nanevski-al:ESOP14, …

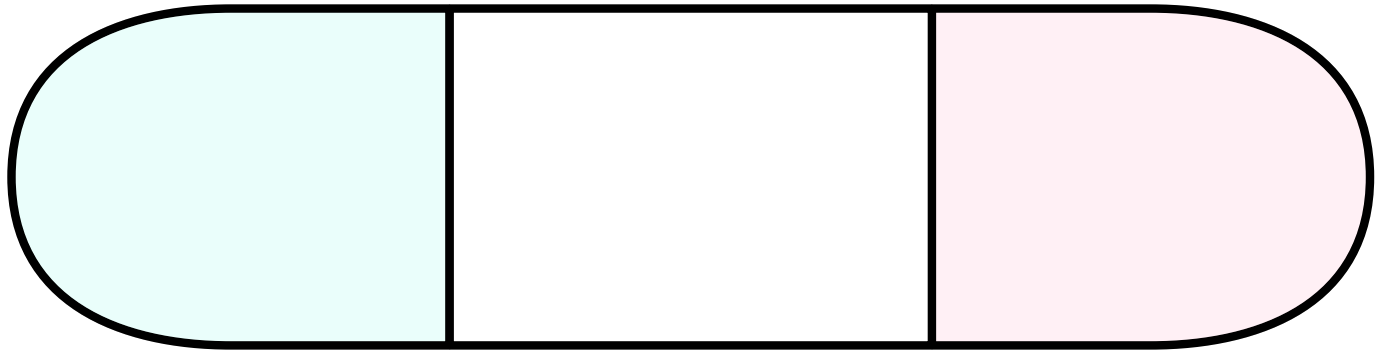
Concurroids — Subjective STSs

Nanevski-al:ESOP14



- *Self* - (possibly ghost) resources owned by <u>me</u>
- *Other* - (possibly ghost) resources owned by <u>all others</u>
- *Shared* - resources owned by <u>the protocol module</u>
- Self and Other are elements of a *Partial Commutative Monoid* (PCM): $(S, \mathbf{0}, \oplus)$.

Specifications with Concurroids

C = 

$$\{\ p\ \}\ c\ \{\ q\ \}\ \underbrace{@\ C}_{\text{defines } \textit{Rely/Guarantee} \text{ and RI}}$$

A model for resources with histories

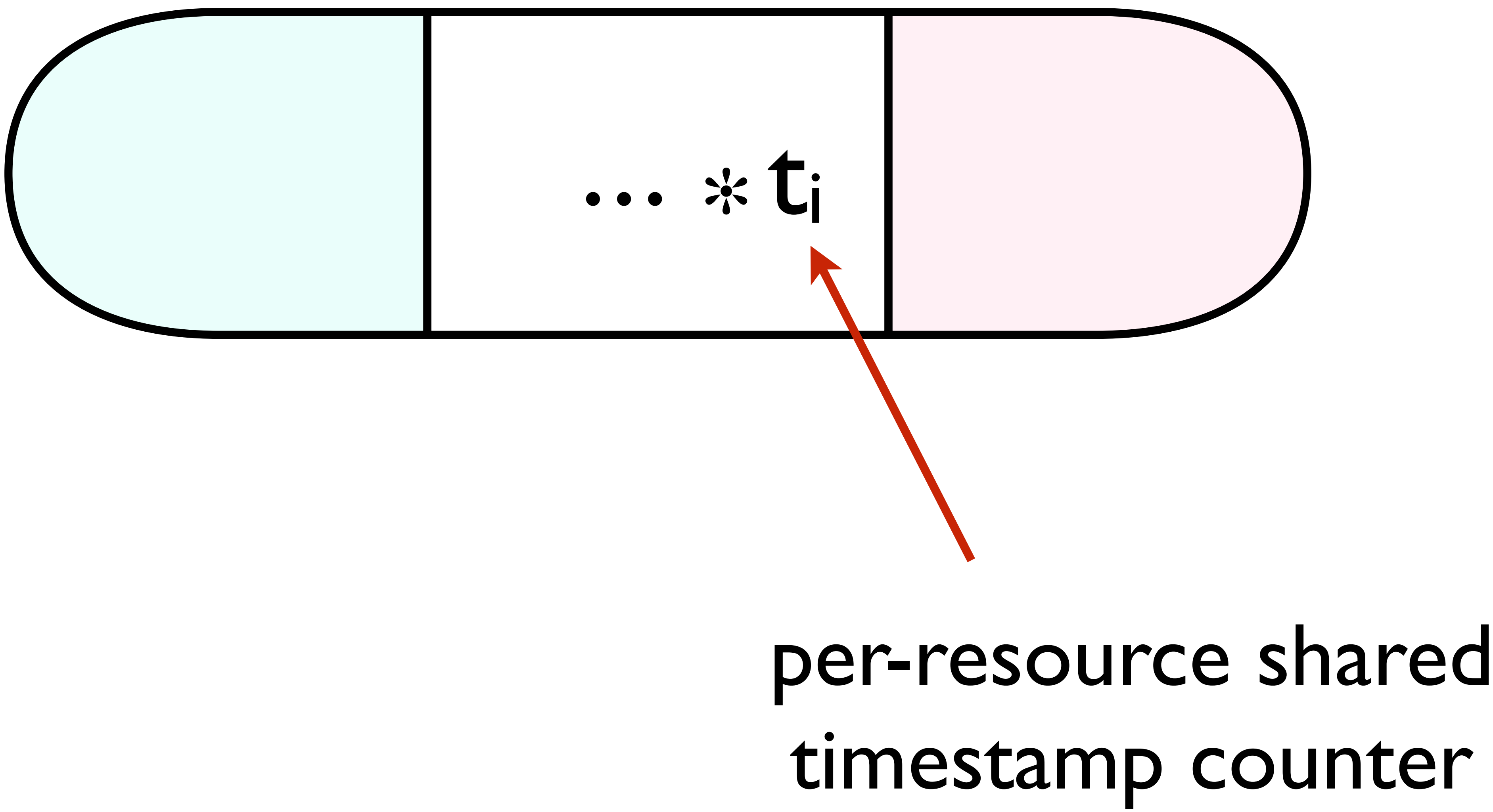
- Resources represented by *State-Transition Systems* (STS)
- Transitions define *Rely/Guarantee* of a resource
- Auxiliaries are *ghost* parts of the resource's state

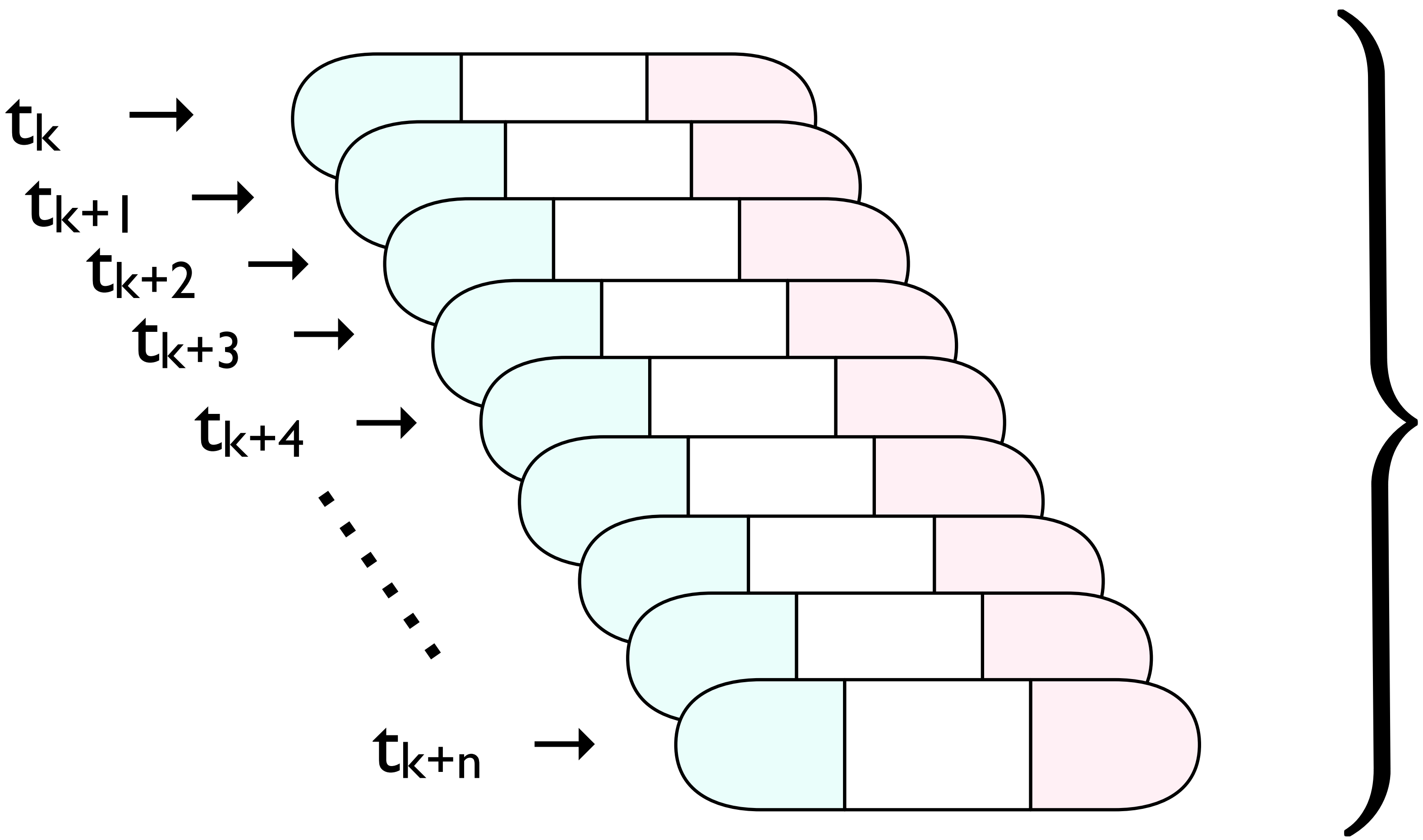
Dinsdale-Young-al:ECOOP10, O'Hearn-al:PODC10, Turon-al:POPL13, Turon-al:ICFP13, Svendsen-al:ESOP13, Svendsen-Birkedal:ESOP14, Nanevski-al:ESOP14, …

A model for resources with histories

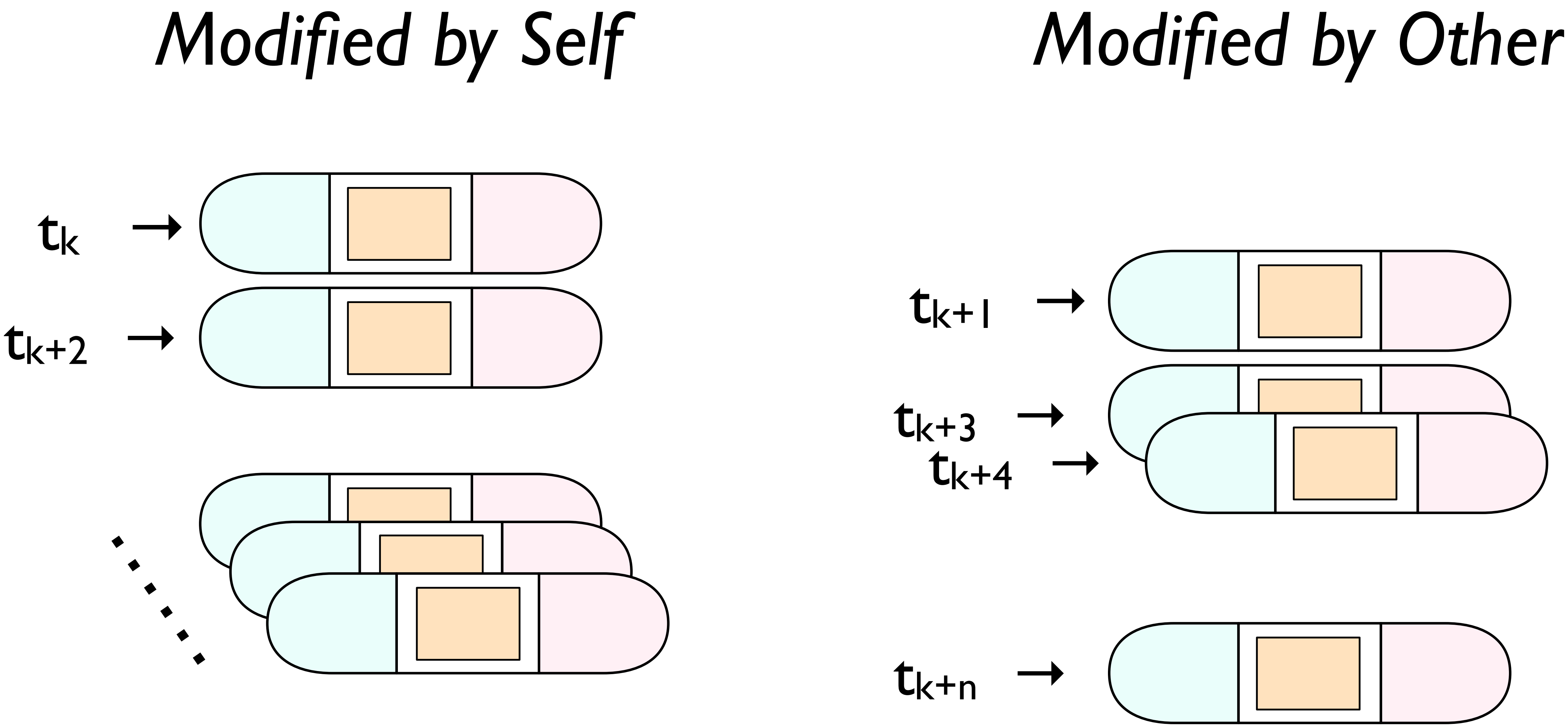
- Resources represented by *State-Transition Systems* (STS)
- Transitions define *Rely/Guarantee* of a resource
- Auxiliaries are *ghost* parts of the resource's state
- *Histories* are a particular case of ghosts

Capturing histories with timestamps



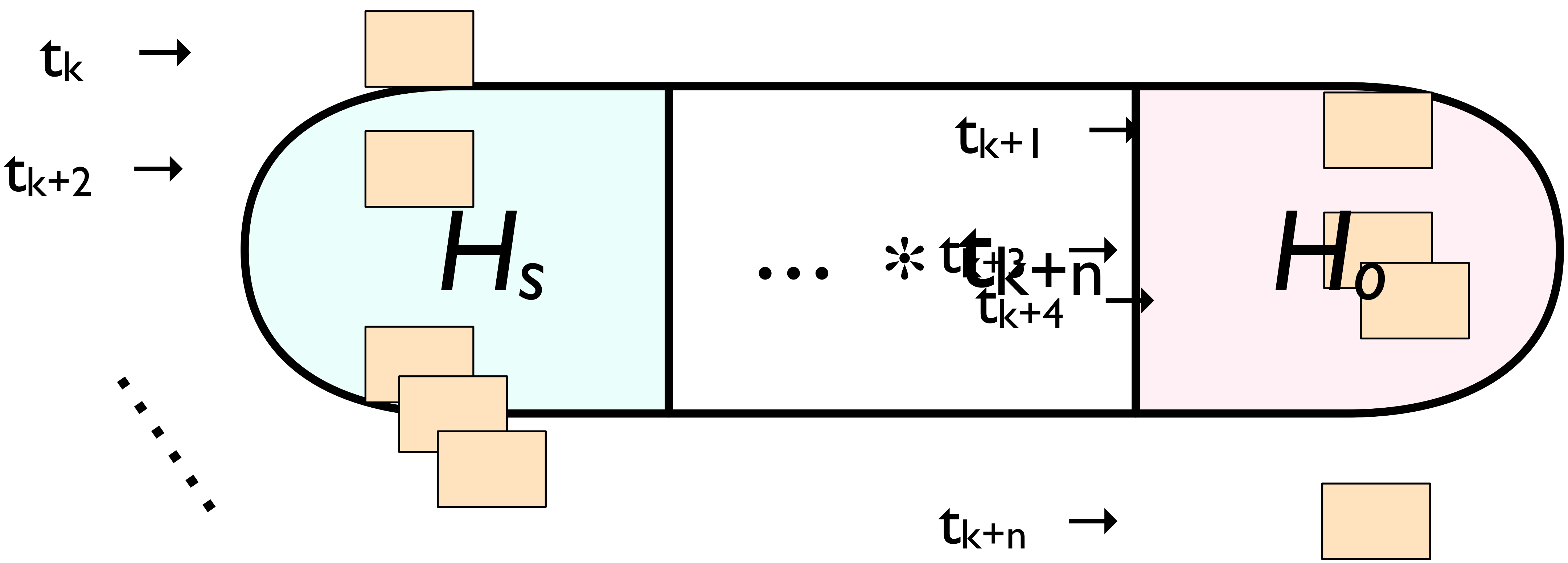


time increased
at every change
in "visible" state



We will record only interesting *projections* of the shared state

Modified by Self *Modified by Other*



- H_s, H_o — *self/other* contributions to the protocol history
- *Timestamped histories* form a PCM ⇒ can be split

Reasoning about pair snapshots

Qadeer-al:TR09, Liang-Feng:PLDI13

Atomically update and increase the version

```
write_x(v) { <x := (x.v, x.s++)> }

write_y(v) { <y := (y.v, y.s++)> }
```

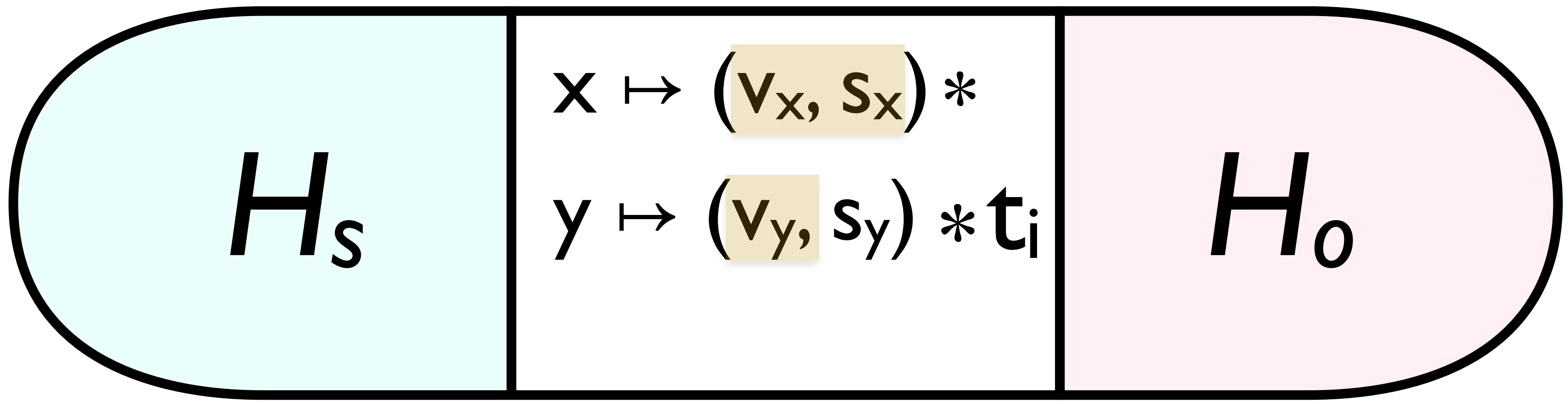
```
letrec read_pair(): (Val, Val) = {
   (v, s) <- <read_x()>;
   (w, _) <- <read_y()>;
   if (s == <read_x()>.s)
   then (v, w);
   else read_pair();
}
```

Atomically read each component

If `x` wasn't changed until this moment, then return a snapshot, else try again.

Pair snapshot concurroid

F_{ps} =



- $H_s, H_o = \{\ t_k \mapsto (v_x, v_y, s_x), \ldots\ \}$
- $H = H_s \uplus H_o$
- Additional coherence constraint:

  $(H(t) = (v_x, v_y, s_x) \wedge H'(t') = (v'_x, v'_y, s_x)) \Rightarrow v_x = v'_x$

- Transitions (R/G) are *writes* with *versions incrementation*

Pair snapshot specification

$H' = H'_s \cup H'_o$

$\{ H_s = \varnothing \}$ `write_x(v)` $\{ \exists t, v_y, s_x.\ H'(t) = (\text{-}, v_y, s_x)$
$\wedge H'_s = [t{+}1 \mapsto (v, v_y, s_x{+}1)] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `write_y(v)` $\{ \exists t, v_x, s_x.\ H'(t) = (v_x, \text{-}, s_x)$
$\wedge H'_s = [t{+}1 \mapsto (v_x, v, s_x)] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `read_pair()` $\{ \exists t, v_x, v_y, s_x.\ \boxed{H'(t) = (v_x, v_y, s_x)} \wedge \boxed{H'_s = \varnothing}$
$\wedge \boxed{res = (v_x, v_y)} \}@F_{ps}$

The proof is trivial, by *coherence requirement* and *Rely*

Stacks specification

$H' = H'_s \cup H'_o$

$\{ H_s = \varnothing \}$ `push(x)` $\{ \exists t, xs.\ H'(t) = xs \wedge H'_s = [t+1 \mapsto (x::xs)] \}@C_{stack}$

$\{ H_s = \varnothing \}$ `pop()` $\{ \textbf{if } (res = Just(x))$
$\textbf{then } \exists t, xs.\ H'(t) = x::xs \wedge H'_s = [t+1 \mapsto xs]$
$\textbf{else } \exists t.\ H'_s = \varnothing \wedge H'(t) = \textit{Nil} \}@C_{stack}$

$\{ H_s = \varnothing \}$ `contains(x)` $\{ H'_s = H_s \wedge$
$\textbf{if } (res) \textbf{ then } \exists t, xs.\ H'(t) = xs \wedge \boxed{x \in xs}$
$\textbf{else } \exists t.\ H(t) = xs \wedge \boxed{x \notin xs} \}@C_{stack}$

What about granularity abstraction?

(for the sake of Hoare-style reasoning simplification)

Granularity abstraction via linearizability

- If and ADT c_1 is linearizable *wrt* to c_2, we can replace c_1 by c_2 for the sake of simpler reasoning (Vafeiadis:PhD08, Liang-Feng:PLDI13)
- Alternatively, if c_1 is *contextual refinement* of c_2, its clients can reason as about c_2 (Filipović-al:TCS10, Turon-al:ICFP13)
- Both *linearizability* and *CR* are relations on *program modules*
- Logics for them are inherently *relational*

Why don't us relate
state-transition systems instead?

(which is, presumably, easier than relating *programs*)

$$\{\, p \,\}\ c\ \{\, q \,\}\ \underbrace{@\, F}_{}$$

defines *Rely/Guarantee*

"fine-grained" concurroid

Refinement function:

$$\Phi : F \rightarrow C$$

Abadi-Lamport:LICS88

$$\frac{\{\, p \,\}\ c\ \{\, q \,\}\ @\ F}{\{\, \Phi(p) \,\}\ \underline{\text{refine}}_{\Phi}\ (c)\ \{\, \Phi(q) \,\}\ @\ C}$$

simple "coarse-grained" concurroid

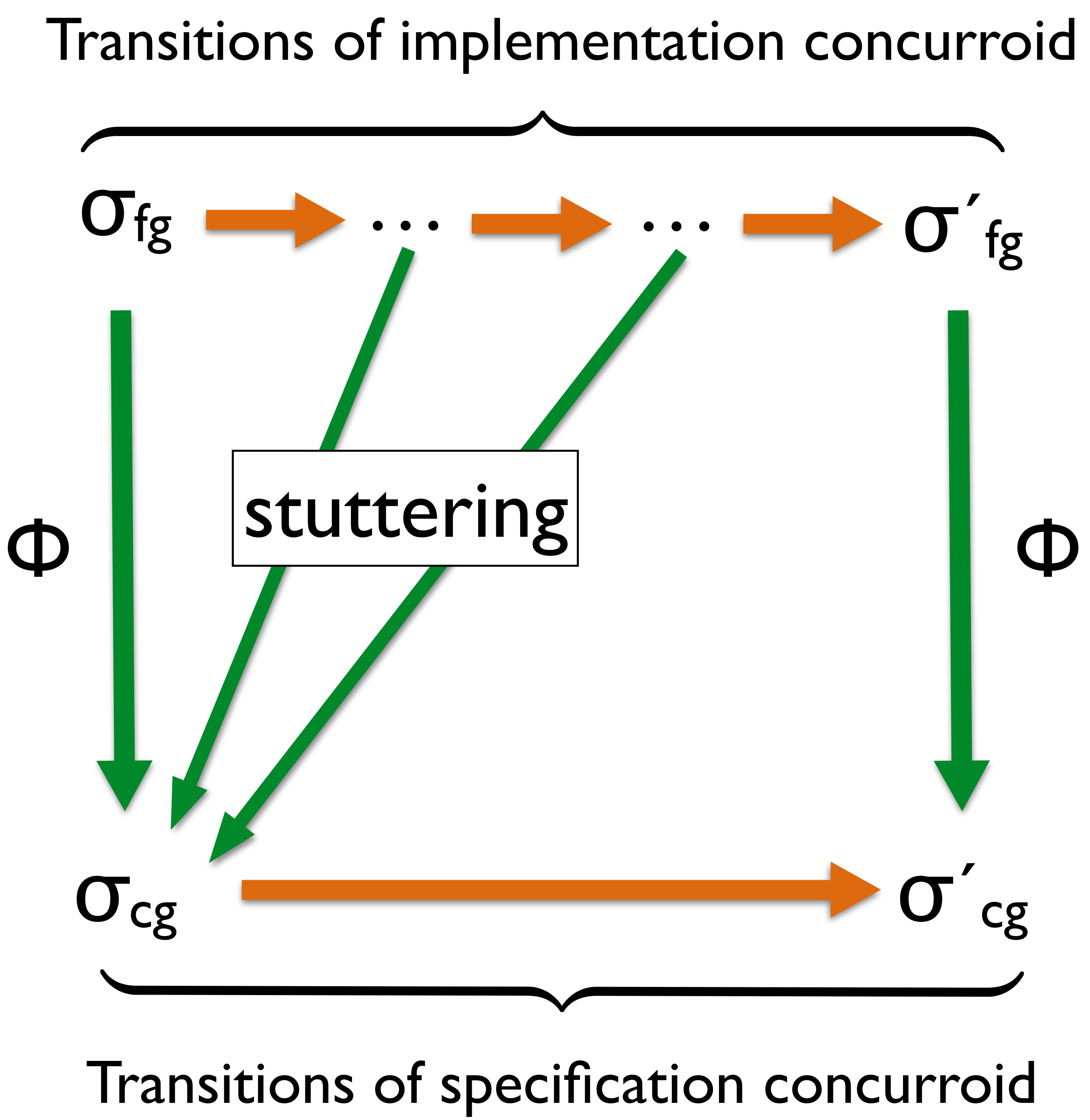
Refinement function:

$$\Phi : F \to C$$

A state in implementation concurroid

$$\sigma_{fg}$$

Establishing Refinement



Refinement for pair snapshots

Pair spec we used to have

$H' = H'_s \cup H'_o$

$\{ H_s = \varnothing \}$ `write_x(v)` $\{ \exists t, v_y, s_x.\ H'(t) = (-, v_y, s_x)$
$\wedge\ H'_s = [t{+}1 \mapsto (v, v_y, s_x{+}1)] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `write_y(v)` $\{ \exists t, v_x, s_x.\ H'(t) = (v_x, -, s_x)$
$\wedge\ H'_s = [t{+}1 \mapsto (v_x, v, s_x)] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `read_pair()` $\{ \exists t, v_x, v_y, s_x.\ H'(t) = (v_x, v_y, s_x) \wedge H'_s = H_s$
$\wedge\ res = (v_x, v_y) \}@F_{ps}$

Pair spec we used to have

$H' = H'_s \cup H'_o$

$\{ H_s = \varnothing \}$ `write_x(v)` $\{ \exists t, v_y, s_x.\, H'(t) = (\text{-}, v_y, \boxed{s_x})$
$\wedge\, H'_s = [t{+}1 \mapsto (v, v_y, \boxed{s_x{+}1})] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `write_y(v)` $\{ \exists t, v_x, s_x.\, H'(t) = (v_x, \text{-}, \boxed{s_x})$
$\wedge\, H'_s = [t{+}1 \mapsto (v_x, v, \boxed{s_x})] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `read_pair()` $\{ \exists t, v_x, v_y, s_x.\, H'(t) = (v_x, v_y, \boxed{s_x}) \wedge H'_s = H_s$
$\wedge\, res = (v_x, v_y) \}@F_{ps}$

Pair spec we used to have

$H' = H'_s \cup H'_o$

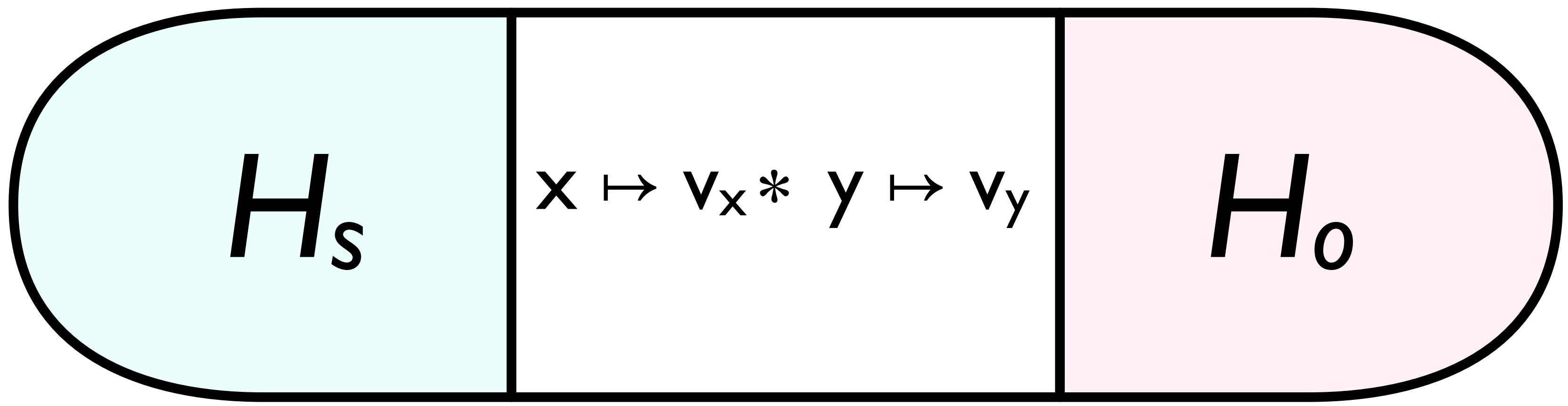
$\{ H_s = \varnothing \}$ `write_x(v)` $\{ \exists t, v_y, s_x.\ H'\boxed{(t)} = (-, v_y, s_x)$
$\wedge\ H'_s = \boxed{[t+1} \mapsto (v, v_y, s_x+1)] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `write_y(v)` $\{ \exists t, v_x, s_x.\ H'\boxed{(t)} = (v_x, -, s_x)$
$\wedge\ H'_s = \boxed{[t+1} \mapsto (v_x, v, s_x)] \}@F_{ps}$

$\{ H_s = \varnothing \}$ `read_pair()` $\{ \exists t, v_x, v_y, s_x.\ H'\boxed{(t)} = (v_x, v_y, s_x) \wedge H'_s = H_s$
$\wedge\ res = (v_x, v_y) \}@F_{ps}$

Coarse-grained Pair concurroid

C_{ps} =



- No timestamps, no value versions
- H_s, H_o = { (v_x, v_y), … } — multi-sets
- $H = H_s \cup H_o$
- Transitions (R/G) are just atomic *writes*
- Φ: $F_{ps} \to C_{ps}$ erases versions and timestamps

Pair spec we have now

$H' = H'_s \cup H'_o$

$\{ H_s = \varnothing \}$ `write_x(v)` $\{ \exists v_y.\ (\text{-}, v_y) \in H'$
$\wedge\ H'_s = \{(v, v_y)\}]\ \}@C_{ps}$

$\{ H_s = \varnothing \}$ `write_y(v)` $\{ \exists v_x, s_x.\ H'(t) = (v_x, \text{-})$
$\wedge\ H'_s = \{(v_x, v)\}\ \}@C_{ps}$

$\{ H_s = \varnothing \}$ `read_pair()` $\{ \exists v_x, v_y.\ (v_x, v_y) \in H' \wedge H'_s = H_s$
$\wedge\ res = (v_x, v_y)\ \}@C_{ps}$

Meeting some old friends

CSL Resource Rule

O'Hearn:TCS07

$$\frac{\Gamma, \boxed{r : I} \vdash \{\, p \,\}\ c\ \{\, q \,\}}{\Gamma \vdash \{\, p \boxed{* \ I} \}\ \underline{\text{resource}}\ r\ \underline{\text{in}}\ c\ \{\, q \boxed{* \ I} \}}$$

FCSL Generalized Resource Rule

Nanevski-al:ESOP14

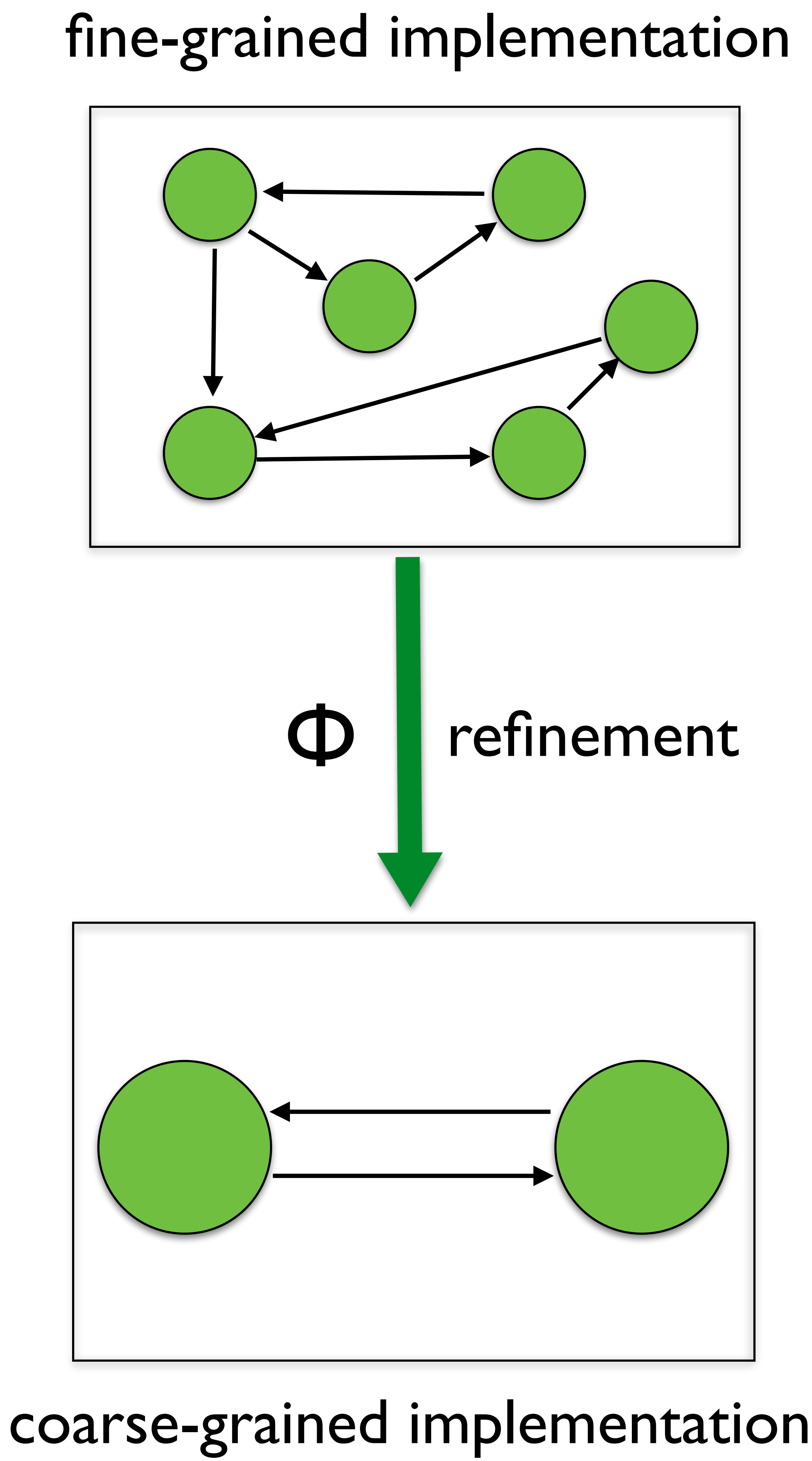
$$\frac{\vdash \{\, \mathit{priv} \mapsto^{s} h * p \,\}\ c\ \{\, \mathit{priv} \mapsto^{s} h' * q \,\}\ @\ (P \rtimes U) \boxed{\rtimes V}}{\vdash \{ \boxed{\Phi(g,h)} * (\Psi(g) -\!\!* \ p) \}\ \underline{\mathrm{hide}}_{\Psi,g}(c)\ \{\, \exists g'. \boxed{\Phi(g,h)} * (\Psi(g) -\!\!* \ q) \,\}\ @\ P \rtimes U}$$

where Φ is defined as Ψ-based *refinement*

Scoped *resource allocation* is a particular case of *refinement*!

Exploring the zoo of STS simulations

Lynch-Vaandrager:InfComp95

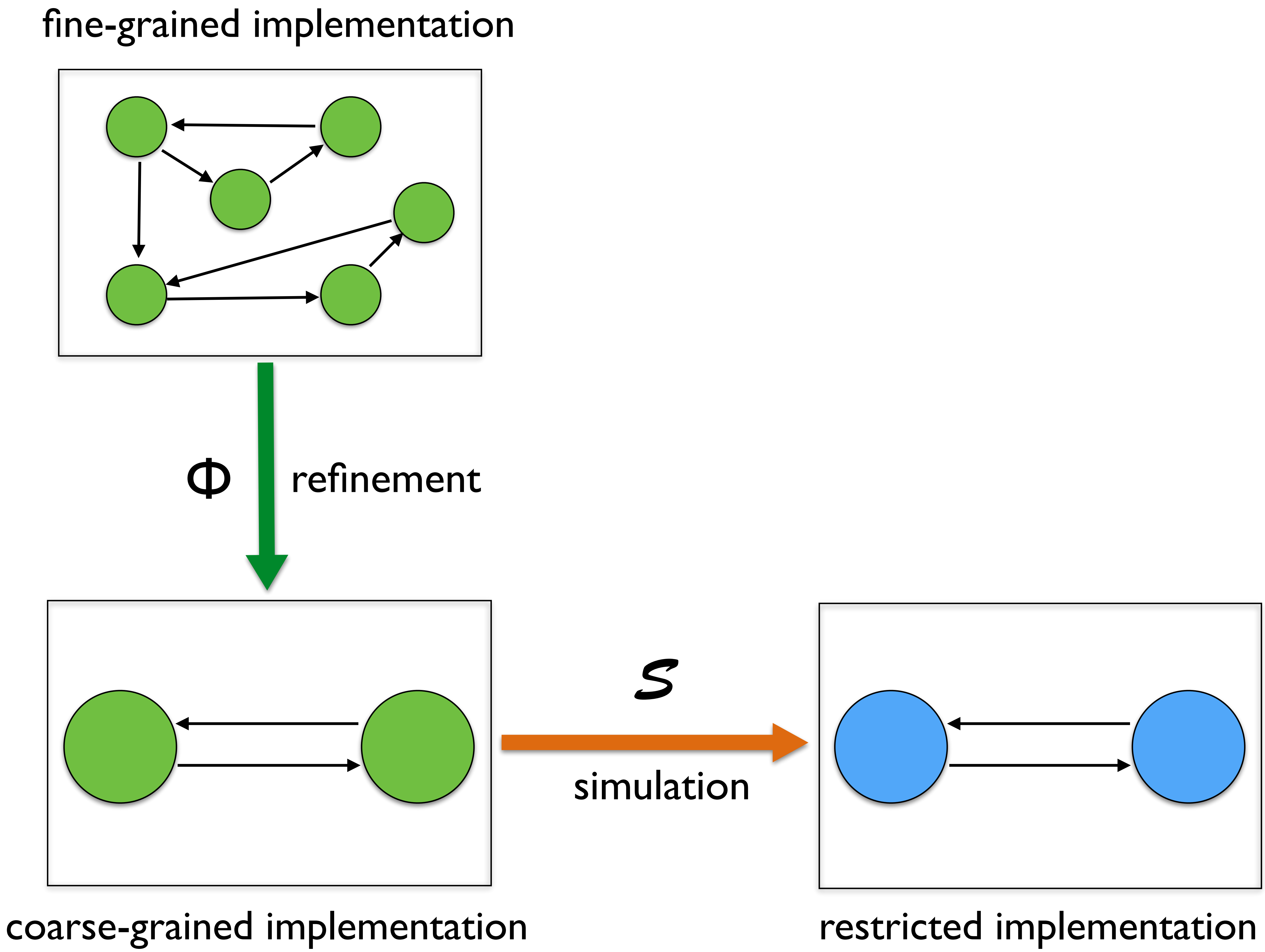

fine-grained implementation
Φ
refinement
coarse-grained implementation

Restricted Stacks

$\forall\, P : Elem \to Prop.$

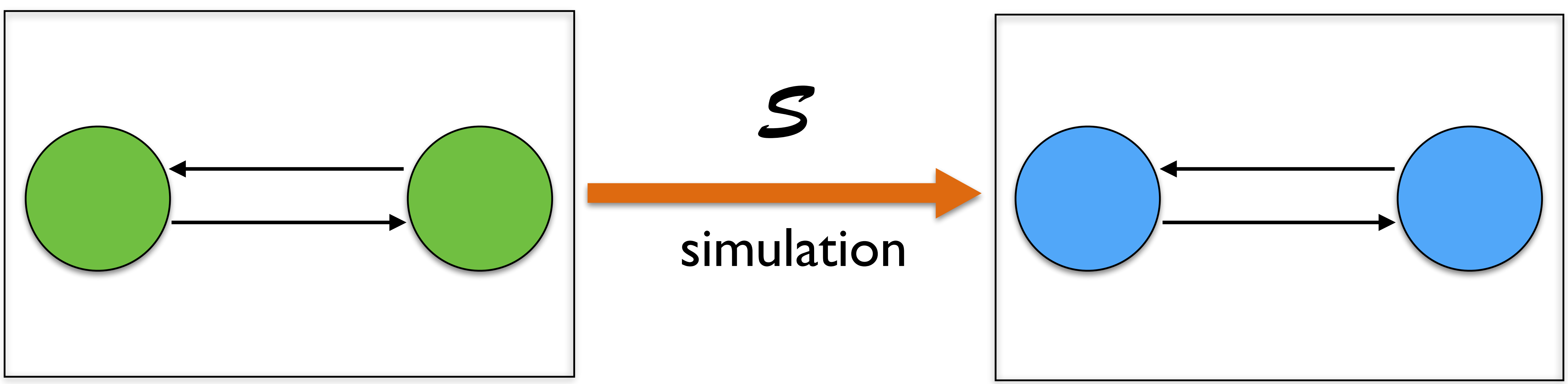
$\{\, P(x) \,\}$ `push(x)` $\{\, \mathbf{true} \,\}$

$\{\, \mathbf{true} \,\}$ `pop()` $\{\ \ res = Nothing$
$\quad \lor\ \exists x.\ res = Just(x) \land P(x) \,\}$



fine-grained implementation
Φ
refinement
S
simulation
coarse-grained implementation
restricted implementation

$$\frac{\{\ p\ \}\ c\ \{\ q\ \}\ @\ C}{\{\ \mathcal{S}(p)\ \}\ \underline{\text{simulate}}_{\mathcal{S}}\ (c)\ \{\ \mathcal{S}(q)\ \}\ @\ C_R}$$



coarse-grained implementation

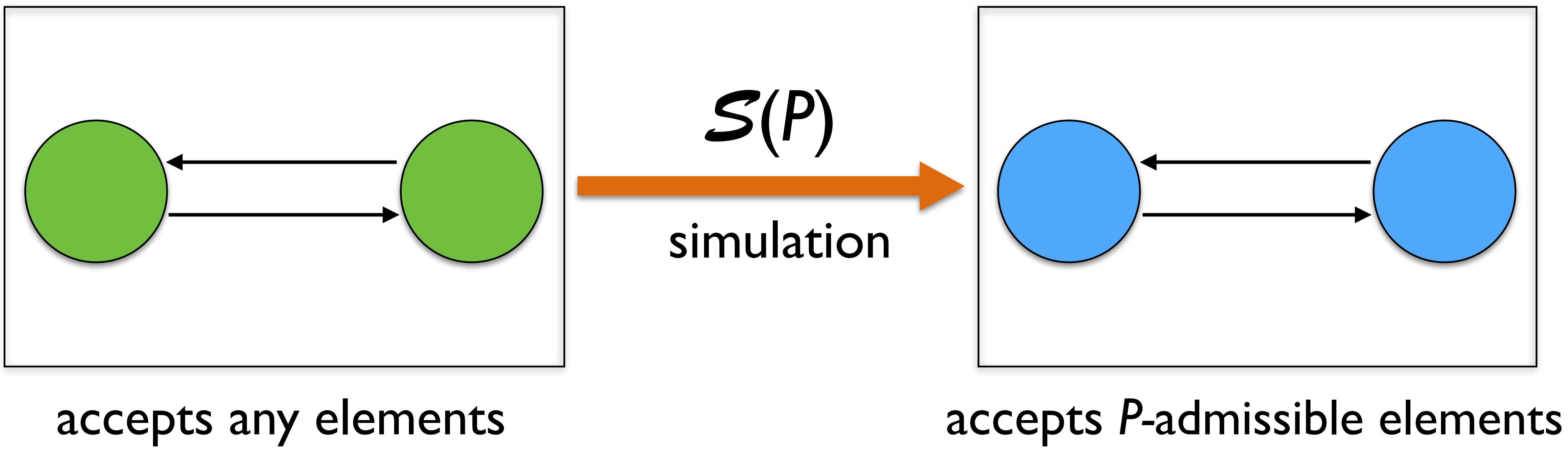
restricted implementation

Restricted stacks

$\forall\, P : \mathit{Elem} \to \mathit{Prop}.$

$\{\, P(x) \,\}$ `push(x)` $\{\, \mathbf{true} \,\}$

$\{\, \mathbf{true} \,\}$ `pop()` $\{\quad \mathit{res} = \mathit{Nothing}$
$\quad \lor\ \exists x.\ \mathit{res} = \mathit{Just}(x) \land P(x) \,\}$



To take away

- We suggest an alternative to linearizability as the only way

  to provide canonical specifications and establish granularity abstraction;
- Histories-as-resource give "canonical" concurrent specs;
- Granularity abstraction could be established via

  STS simulation techniques (hopefully).

Some open questions:

- What is use for other simulation (backwards, FB, BF)?
- So far we didn't need prophecy variables? Can we avoid them at all?
- Can we define the notion of "atomicity" in terms of STS and simulations?

Thanks!