Semantics and invariance proof method for weakly consistent parallelism

Jade Alglave (MSR-Cambridge, UCL, UK)
Patrick Cousot (NYU, Emer. ENS, PSL)

Dagstuhl Seminar 16471

http://www.dagstuhl.de/16471

Concurrency with Weak Memory Models: Semantics, Languages, Compilation, Verification, Static Analysis, and Synthesis

November 20 – 25 , 2016

Weakly consistent parallel programs

Weakly consistent parallel programs

var $x_1,\ldots,x_m$; // shared variables
P_0; // prelude initializing $x_1,\ldots,x_m$
[P_1 || P_2 || … || P_n]

- P_1, P_2, …, P_n are the processes modifying the shared variables and their local registers R, …
- The execution of a write x := E to a shared variable and the read R := x of a shared variable is <u>not</u> instantaneous (as in sequential consistency)

Example (lb, load buffer)

- Algorithm A:

```
0:{ x = 0; y = 0; }
P0           ║ P1           ;
1:r[] r1 x   ║ 11:r[] r2 y;
2:w[] y 1    ║ 12:w[] x 1 ;
3:           ║ 13:        ;
```

- Specification S_{inv}:

$$\texttt{at 3} \wedge \texttt{at 13} \Rightarrow \neg(\texttt{r1}{=}1 \wedge \texttt{r2}{=}1)$$

Example (Peterson)

- **Algorithm A:**

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                      || P1:
1:w[] F1 true            || 10:w[] F2 true;
2:w[] T 2                || 11:w[] T 1;
3:do {i}                 || 12:do {j}
4:    r[] R1 F2          || 13:    r[] R3 F1;
5:    r[] R2 T           || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1      || 15:while R3 ∧ R4 ≠ 2;
7:skip (* CS1 *)         || 16:skip (* CS2 *)
8:w[] F1 false           || 17:w[] F2 false;
9:                       || 18:
```

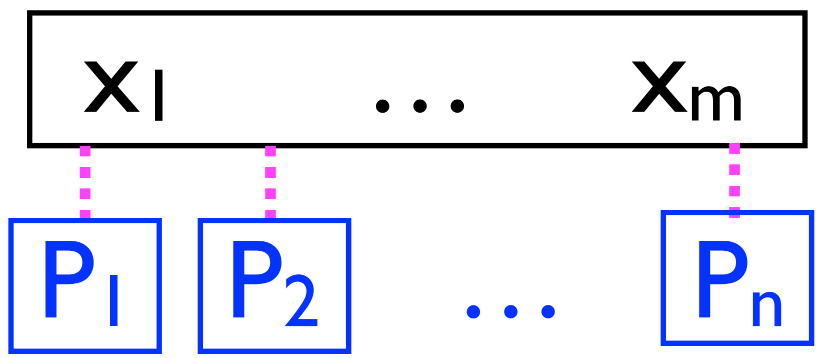
- **Specification S_{inv}:**

1: {true}	10: {true}
...	...
7: {¬at{16}}	16: {¬at{7}}
...	...
9: {true}	18: {true}

G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981. doi: 10.1016/0020-0190(81)90106-X. URL http://dx.doi.org/10.1016/0020-0190(81)90106-X.

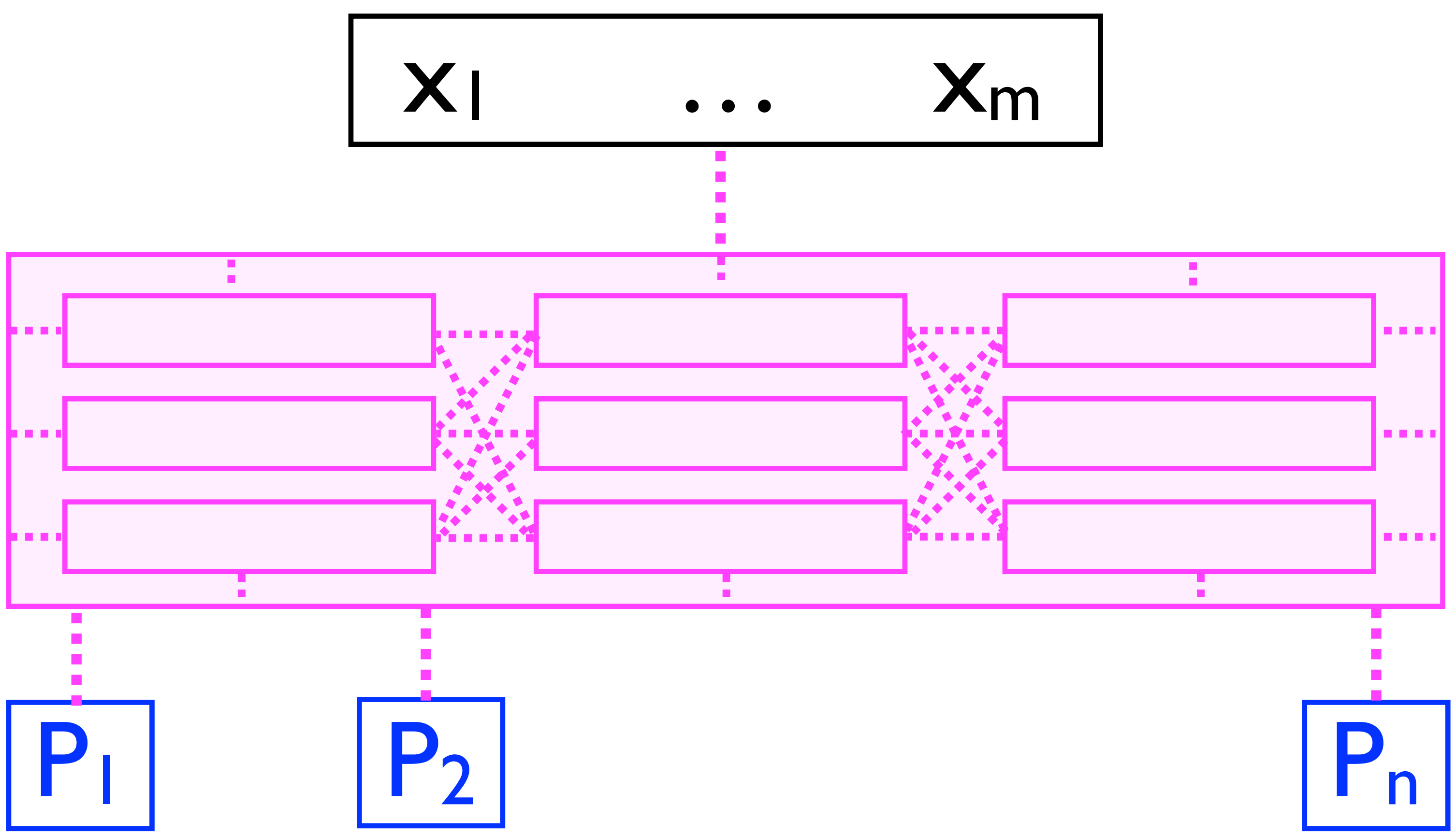
Weak memory/consistency models

- Sequential consistency:



atomic instantaneous communications

- Weak memory models:

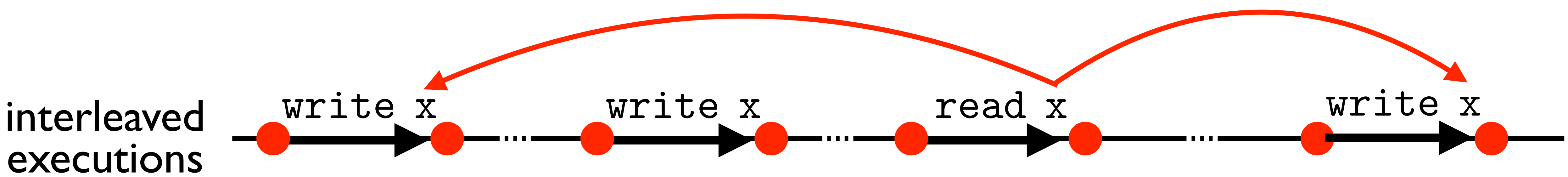


communication network

(anticipations, delays, shuffles…)

Read/write matching

- In the worst case a `read x` can read from any past or future `write x` of any process (including for the reading process)



Example (1b, incorrect)

- ```
  0:{ x = 0; y = 0; }
  P0            || P1         ;
  1:r[] r1 x    || 11:r[] r2 y;
  2:w[] y 1     || 12:w[] x 1 ;
  3:            || 13:        ;
  ```
- at 3 $\wedge$ at 13 $\wedge$ r1$=1$ $\wedge$ r2$=1$
- This erroneous behavior can be observed on TSO machines
  ```

Example: Peterson (incorrect)

- Can read the wrong flags

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                          || P1:
1:w[] F1 true                || 10:w[] F2 true;
2:w[] T 2                rf  || 11:w[] T 1;
3:do                         || 12:do
4:   r[] R1 F2               || 13:   r[] R3 F1;
5:   r[] R2 T                || 14:   r[] R4 T;
6:while R1 ∧ R2 ≠ 1          || 15:while R3 ∧ R4 ≠ 2;
7:skip (* CS1 *)             || 16:skip (* CS2 *)
8:w[] F1 false               || 17:w[] F2 false;
9:                           || 18:
```

at 6 $\wedge$ at 16: $\neg$R1 $\wedge$ R2=1 $\wedge$ $\neg$R3 $\wedge$ R4=2 holds

$\Rightarrow$ both processes simultaneously enter their critical section

Example: Peterson (incorrect)

- Can read the wrong turns

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                        || P1:
1:w[] F1 true              || 10:w[] F2 true;
2:w[] T 2              rf  || 11:w[] T 1;
3:do                       || 12:do
4:    r[] R1 F2            || 13:    r[] R3 F1;
5:    r[] R2 T             || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1        || 15:while R3 ∧ R4 ≠ 2;
7:skip (* CS1 *)           || 16:skip (* CS2 *)
8:w[] F1 false             || 17:w[] F2 false;
9:                         || 18:
```

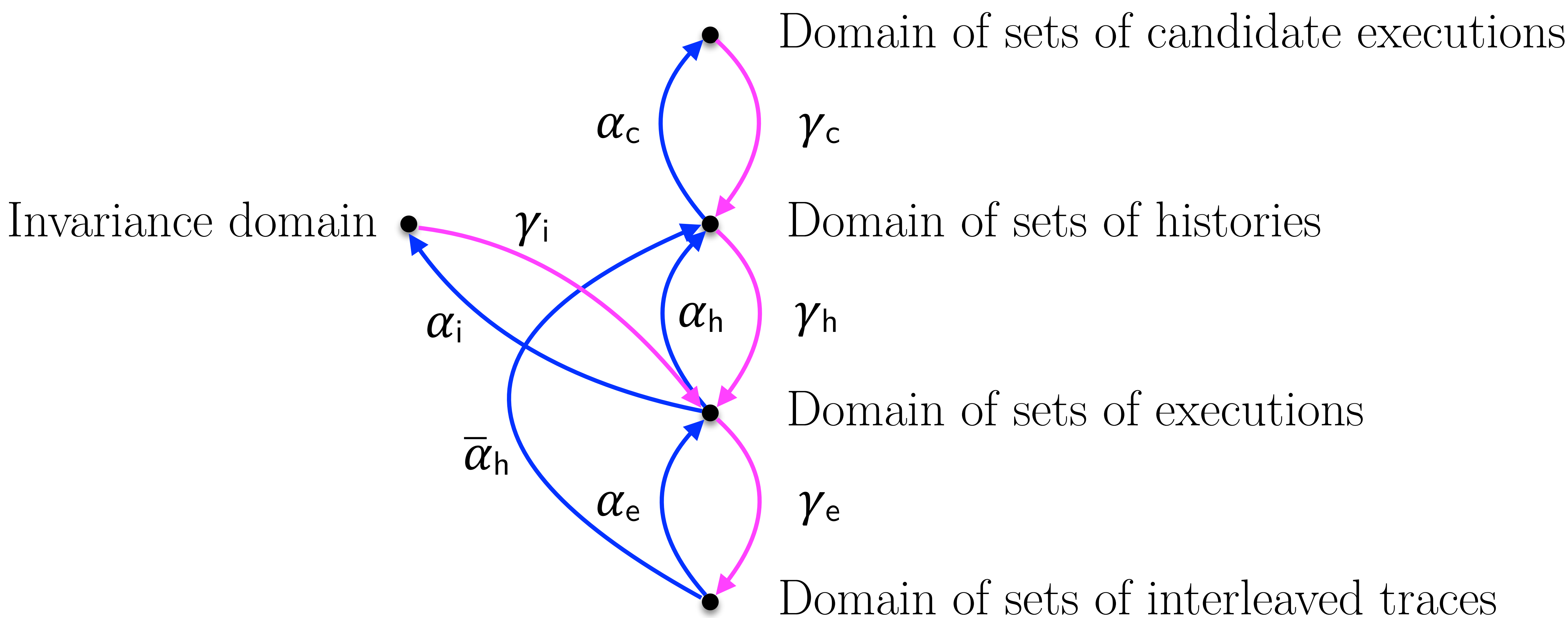
at 6 $\wedge$ at 16: $\neg R1 \wedge R2=1 \wedge \neg R3 \wedge R4=2$ holds

$\Rightarrow$ both processes simultaneously enter their critical section

A hierarchy of semantics of weakly consistent parallelism

Hierarchy of semantics

- Hierarchy of semantic domains:



- Induces a hierarchy of semantics

Sets of interleaved traces

- Traces: maximal finite or infinite sequence of states separated by events generated by computation and communication steps $\longrightarrow$ *global time*

- States: shared memory assigning values to global variables, store buffers, … program point of each process, assignment to local registers

- Events e: P(e) process executed, A(e): labelled action executed, X(e): shared variable involved, V(e): value involved, …

- No restriction on who can read which write on the same shared variable!

Example of interleaved trace for lb

- ```
  0:{ x = 0; y = 0; }
  P0         || P1         ;
  1:r[] r1 x || 11:r[] r2 y;
  2:w[] y 1  || 12:w[] x 1 ;
  3:         || 13:        ;
  ```
- start $\xrightarrow{\texttt{0:}\overbrace{\texttt{x = 0}}^{w^0_{\mathtt{x}}}\texttt{;}\ \overbrace{\texttt{y = 0}}^{w^0_{\mathtt{y}}}} \langle\{\mathtt{x} \leftarrow w^0_{\mathtt{x}}, \mathtt{y} \leftarrow w^0_{\mathtt{y}}\},\ \texttt{1:}\{\mathtt{r1} \leftarrow 0\},\ \texttt{11:}\{\mathtt{r2} \leftarrow 0\}\rangle \xrightarrow{\overbrace{\texttt{1:r[] r1 x}}^{r^1_{\mathtt{x}}}} \langle\{\mathtt{x} \leftarrow w^{12}_{\mathtt{x}}, \mathtt{y} \leftarrow w^0_{\mathtt{y}}\},\ \texttt{2:}\{\mathtt{r1} \leftarrow 1\},\ \texttt{11:}\{\mathtt{r2} \leftarrow 0\}\rangle \xrightarrow{\overbrace{\texttt{11:r[] r2 y}}^{r^{11}_{\mathtt{y}}}} \langle\{\mathtt{x} \leftarrow w^{12}_{\mathtt{x}}, \mathtt{y} \leftarrow w^2_{\mathtt{y}}\},\ \texttt{2:}\{\mathtt{r1} \leftarrow 1\},\ \texttt{12:}\{\mathtt{r2} \leftarrow 1\}\rangle \xrightarrow{\overbrace{\texttt{12:w[] x 1}}^{w^{12}_{\mathtt{x}}}} \langle\{\mathtt{x} \leftarrow w^{12}_{\mathtt{x}}, \mathtt{y} \leftarrow w^2_{\mathtt{y}}\},\ \texttt{2:}\{\mathtt{r1} \leftarrow 1\},\ \texttt{13:}\{\mathtt{r2} \leftarrow 1\}\rangle \xrightarrow{\overbrace{\texttt{2:w[] y 1}}^{w^2_{\mathtt{y}}}} \langle\{\mathtt{x} \leftarrow w^{12}_{\mathtt{x}}, \mathtt{y} \leftarrow w^2_{\mathtt{y}}\},\ \texttt{3:}\{\mathtt{r1} \leftarrow 1\},\ \texttt{13:}\{\mathtt{r2} \leftarrow 1\}\rangle$
  ```

Sets of truly parallel execution traces

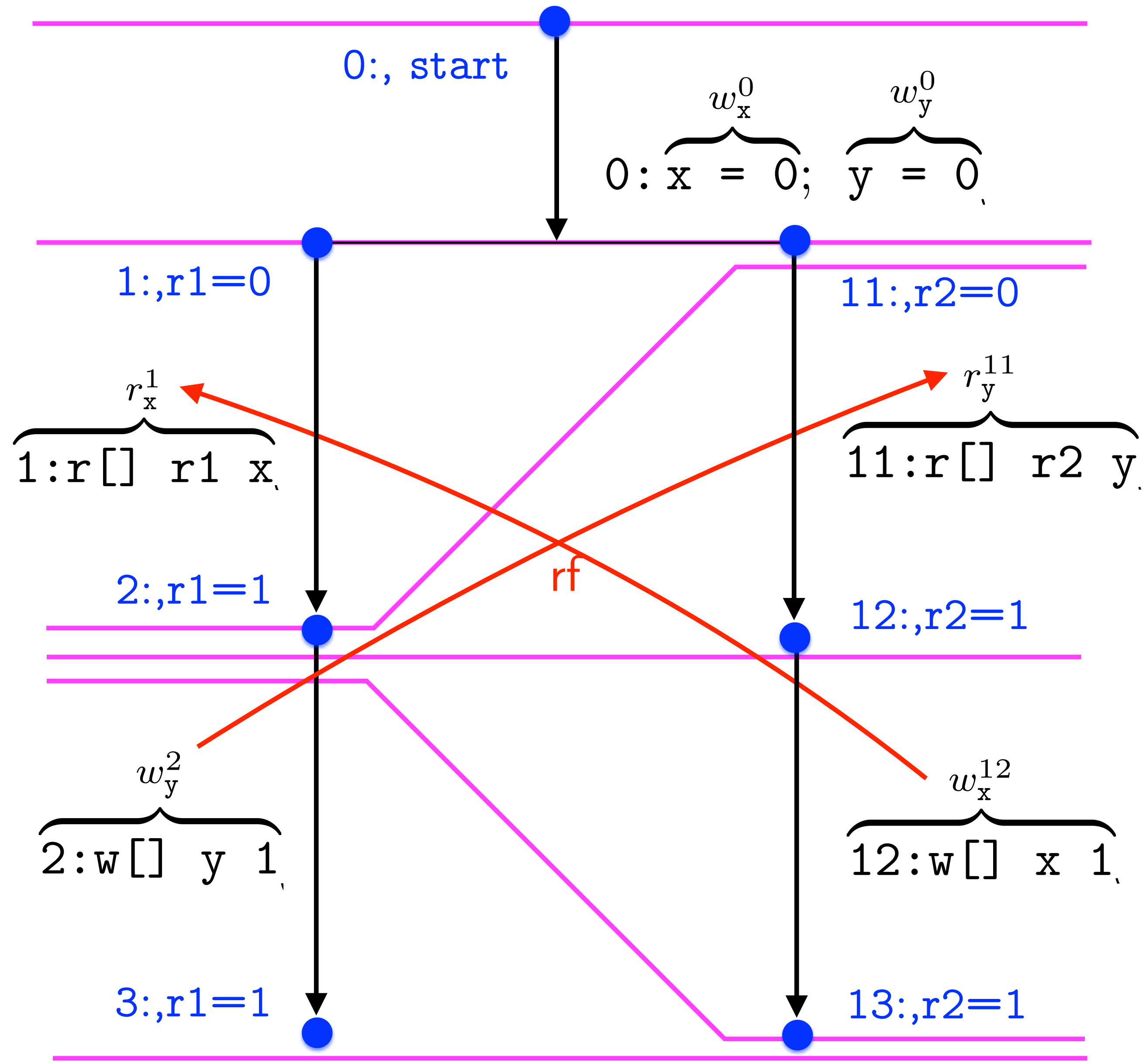
- project traces per process $\longrightarrow$ *local time on computations*
- get rid of shared memory states using a read-from relation rf $\longrightarrow$ *no time on communications*

  $\langle r, w\rangle \in \mathsf{rf} \Longleftrightarrow \tau = \tau_0\langle \nu, \ldots\rangle \xrightarrow{r} \langle \nu', \ldots\rangle\tau_1 \wedge \nu'(\mathsf{X}(r)) = w$

- keep local states on process control points and values of registers
- keep computation progress information using cuts of parallel traces $\longrightarrow$ *global time*

Example of truly parallel execution for lb

```
0:{ x = 0; y = 0; }
P0           || P1          ;
1:r[] r1 x   || 11:r[] r2 y;
2:w[] y 1    || 12:w[] x 1 ;
3:           || 13:        ;
```

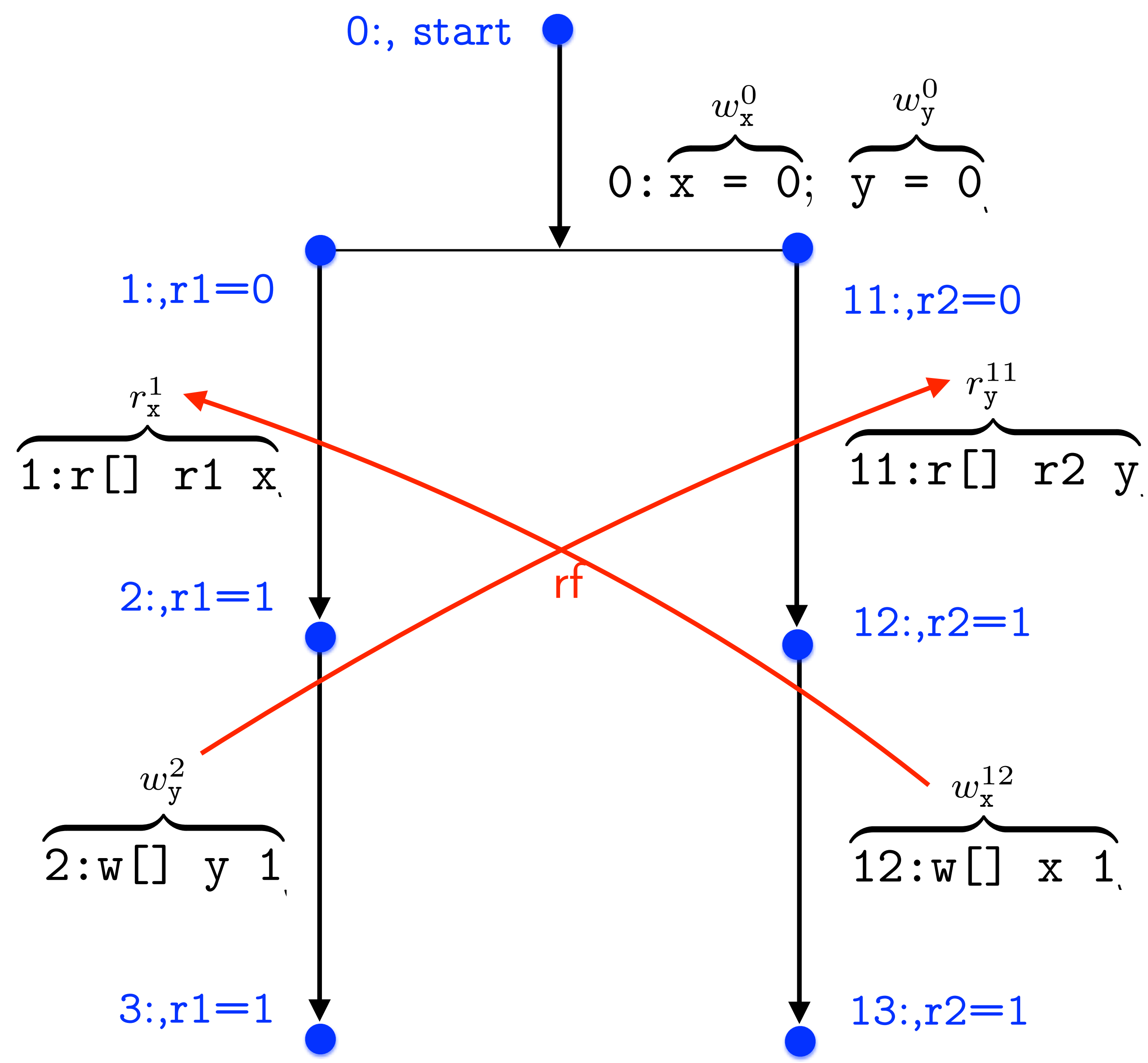


Sets of histories

- Get rid of cuts $\longrightarrow$ *no global time*
- A processor cannot know where the others parallel processors are in their computations

Example of history for lb

```
0:{ x = 0; y = 0; }
P0           || P1          ;
1:r[] r1 x   || 11:r[] r2 y;
2:w[] y 1    || 12:w[] x 1 ;
3:           || 13:         ;
```



Sets of candidate executions

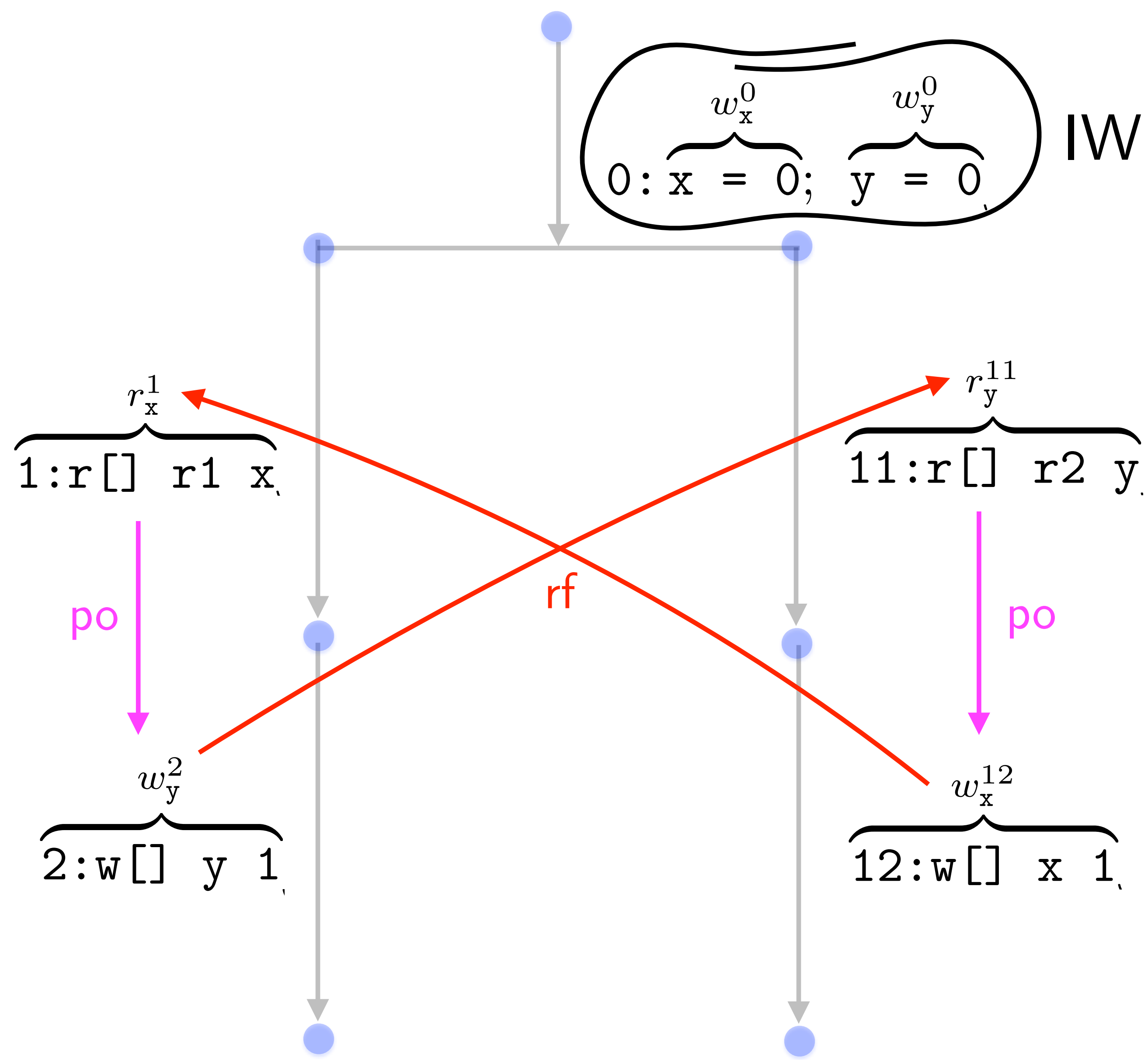
- Keep the set of events
- Keep the read-from relation rf
- Represent process traces $\tau_0 \prod_{i=1}^{n} \tau_i$, rf by
  - the set of initial writes IW in τ_0
  - the program order po

    $\langle e, e' \rangle \in \text{po} \iff \tau_i = \tau_i' \xrightarrow{e} \tau_i'' \xrightarrow{e'} \tau_i'''$

    $\longrightarrow$ *relational on events*
- Get rid of states

Example of candidate execution for `lb`

```
0:{ x = 0; y = 0; }
P0            ║ P1          ;
1:r[] r1 x    ║ 11:r[] r2 y;
2:w[] y 1     ║ 12:w[] x 1  ;
3:            ║ 13:         ;
```



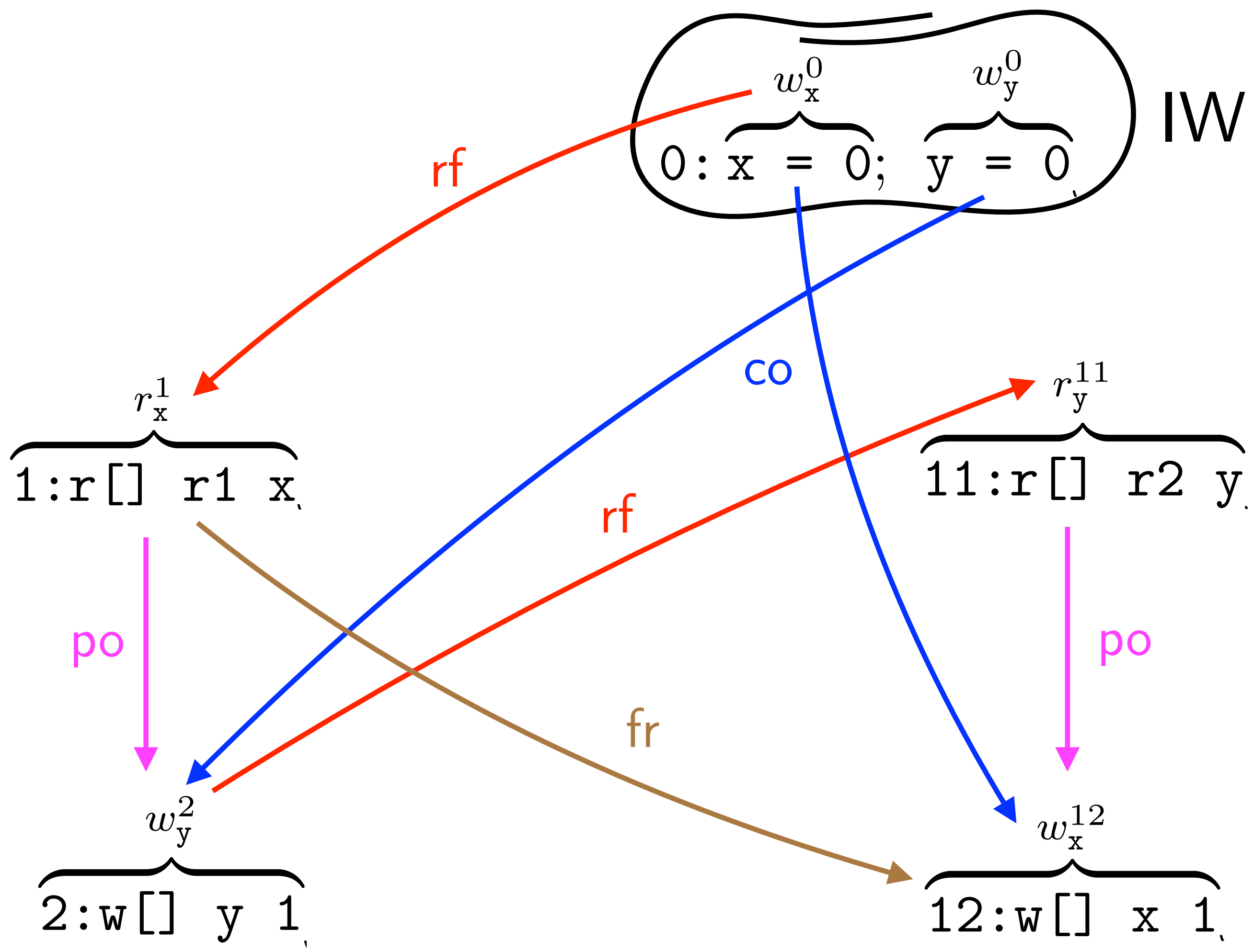
Auxiliary relations

- loc: between events on the same shared variable
- ext: between events on different processes
- coherence order co: between a write and the later ones on the same shared variable
- from-read fr: between a read reading from a write and the later writes to the same shared variable

  $$fr = rf^{-1} \,;\, co$$

Auxiliary relations

```
0:{ x = 0; y = 0; }
P0          || P1           ;
1:r[] r1 x  || 11:r[] r2 y;
2:w[] y 1   || 12:w[] x 1 ;
3:          || 13:          ;
```



co in cat

```
"co.cat"

let fold f =
  let rec fold_rec (es,y) = match es with
  || {} -> y
  || e ++ es -> fold_rec (es, f(e,y))
  end in
  fold_rec

let map f = fun es -> fold (fun (e,y) -> f e ++ y) (es,{})

let rec cross S = match S with
  || {} -> { 0 }
  || S1 ++ S ->
      let yss = cross S in
      fold
        (fun (e1,r) -> map (fun t -> e1 | t) yss | r)
(S1,{}) end

let co0 = loc & (IW * (W\IW))
let makeCo(s) = linearisations(s,co0)
let same-loc-writes = loc & (W*W)
let allCoL = map makeCo (classes (same-loc-writes))
let allCo = cross allCoL

with co from allCo
```

Example of specification of weakly consistent parallelism in the semantic hierarchy: sequential consistency

Sequential consistency

- Interleaved semantics: a read can only read from the last past write

- lb:

$$\text{start} \xrightarrow{\texttt{0:}\,\overbrace{\texttt{x = 0}}^{w_{\mathtt{x}}^{0}}\texttt{;}\ \overbrace{\texttt{y = 0}}^{w_{\mathtt{y}}^{0}}} \langle\{\mathtt{x} \leftarrow w_{\mathtt{x}}^{0}, \mathtt{y} \leftarrow w_{\mathtt{y}}^{0}\},\ \mathtt{1}{:}\{\mathtt{r1} \leftarrow 0\},\ \mathtt{11}{:}\{\mathtt{r2} \leftarrow$$

$$0\}\rangle \xrightarrow{\overbrace{\texttt{1:r[] r1 x}}^{r_{\mathtt{x}}^{12}}} \langle\{\mathtt{x} \leftarrow \cancel{w_{\mathtt{x}}^{12}}, \mathtt{y} \leftarrow w_{\mathtt{y}}^{0}\},\ \mathtt{2}{:}\{\mathtt{r1} \leftarrow 1\},\ \mathtt{11}{:}\{\mathtt{r2} \leftarrow 0\}\rangle \xrightarrow{\overbrace{\texttt{11:r[] r2 y}}^{r_{\mathtt{y}}^{11}}}$$

$$\langle\{\mathtt{x} \leftarrow w_{\mathtt{x}}^{12}, \mathtt{y} \leftarrow \cancel{w_{\mathtt{y}}^{2}}\},\ \mathtt{2}{:}\{\mathtt{r1} \leftarrow 1\},\ \mathtt{12}{:}\{\mathtt{r2} \leftarrow 1\}\rangle \xrightarrow{\overbrace{\texttt{12:w[] x 1}}^{w_{\mathtt{x}}^{12}}} \langle\{\mathtt{x} \leftarrow w_{\mathtt{x}}^{12}, \mathtt{y} \leftarrow$$

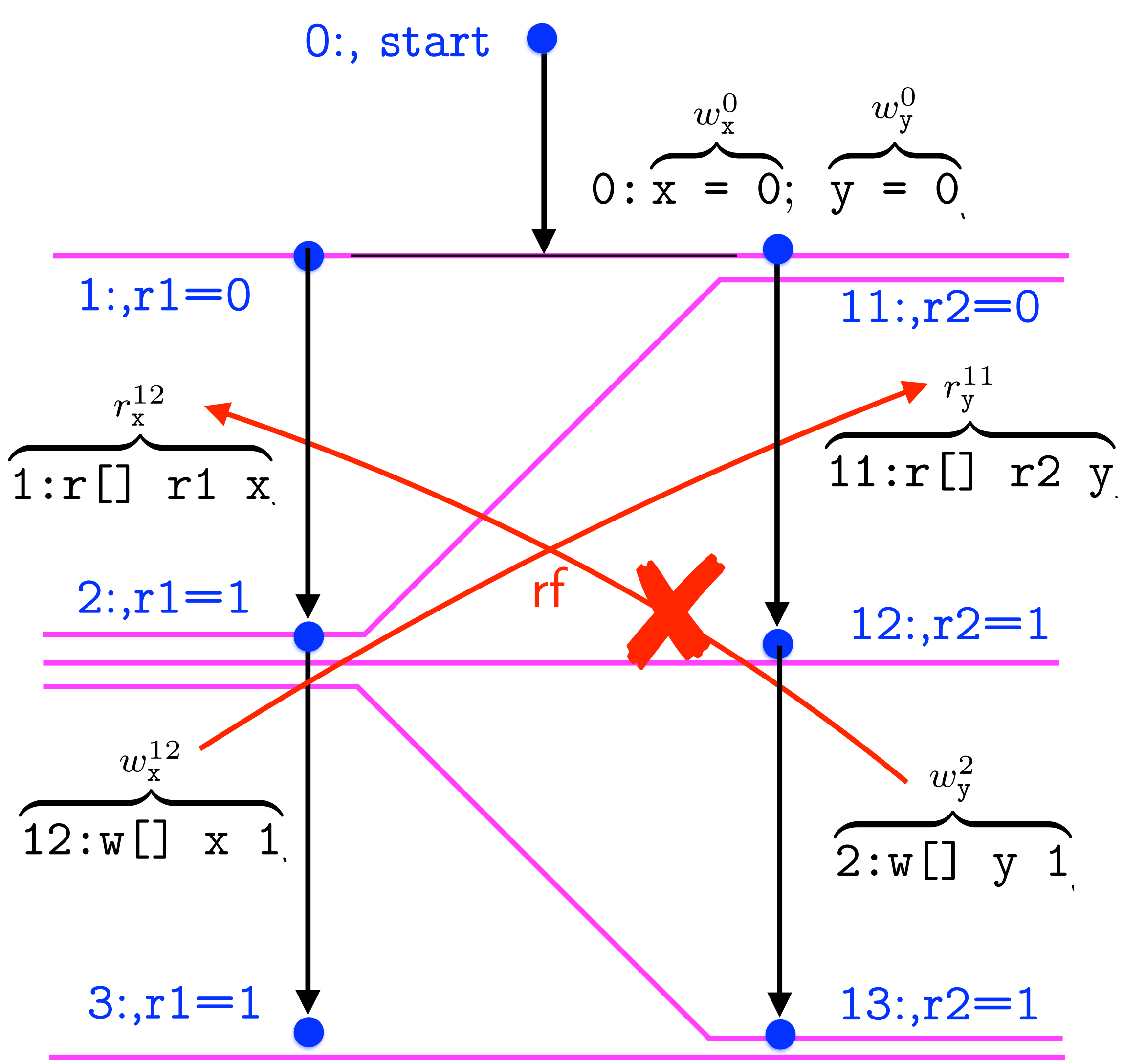
$$w_{\mathtt{y}}^{2}\},\ \mathtt{2}{:}\{\mathtt{r1} \leftarrow 1\},\ \mathtt{13}{:}\{\mathtt{r2} \leftarrow 1\}\rangle \xrightarrow{\overbrace{\texttt{2:w[] y 1}}^{w_{\mathtt{y}}^{2}}} \langle\{\mathtt{x} \leftarrow w_{\mathtt{x}}^{12}, \mathtt{y} \leftarrow w_{\mathtt{y}}^{2}\},\ \mathtt{3}{:}\{\mathtt{r1} \leftarrow 1\},$$

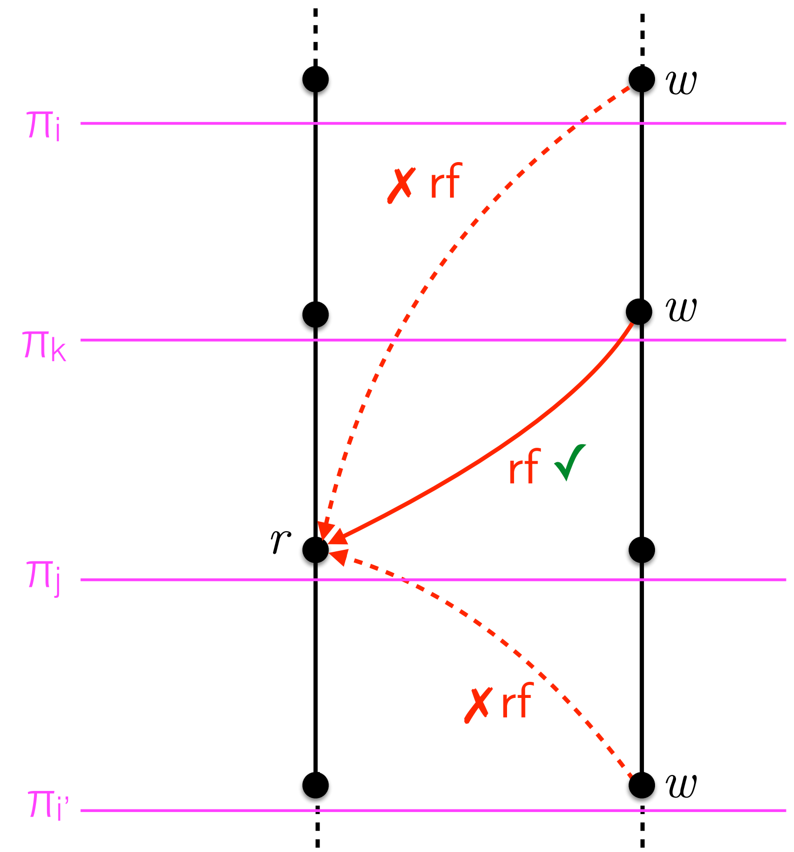
$$\mathtt{11}{:}\{\mathtt{r2} \leftarrow 1\}\rangle$$

Example: sequential consistency for lb

- Parallel executions with cuts: a read can read only the last write before its cut

- lb:

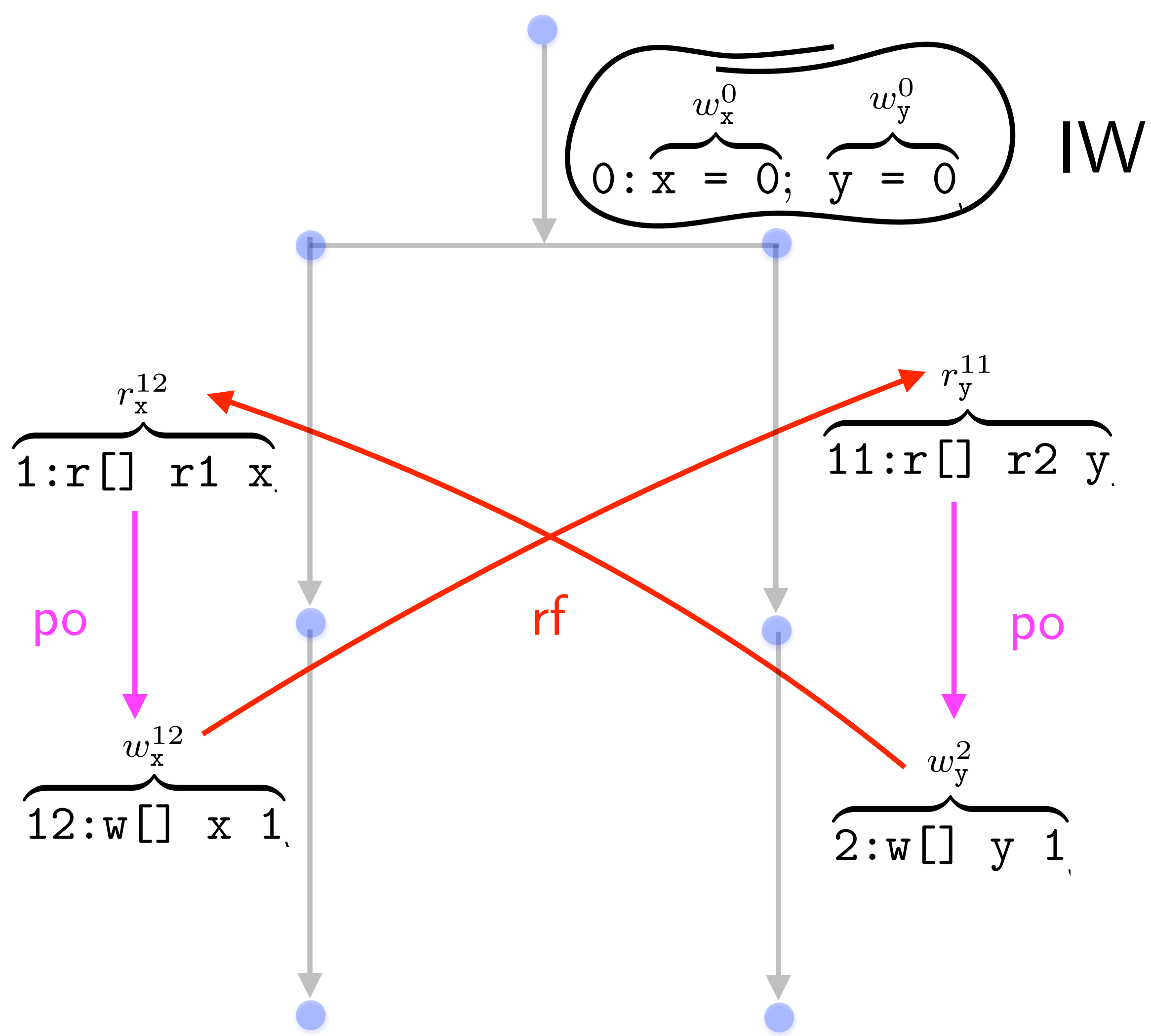




Example: sequential consistency for lb

- Parallel histories: abstract to candidate execution and check it is allowed

- Candidate executions: irreflexive po ; rf ; po; rf



Analytic semantics of weakly consistent parallelism

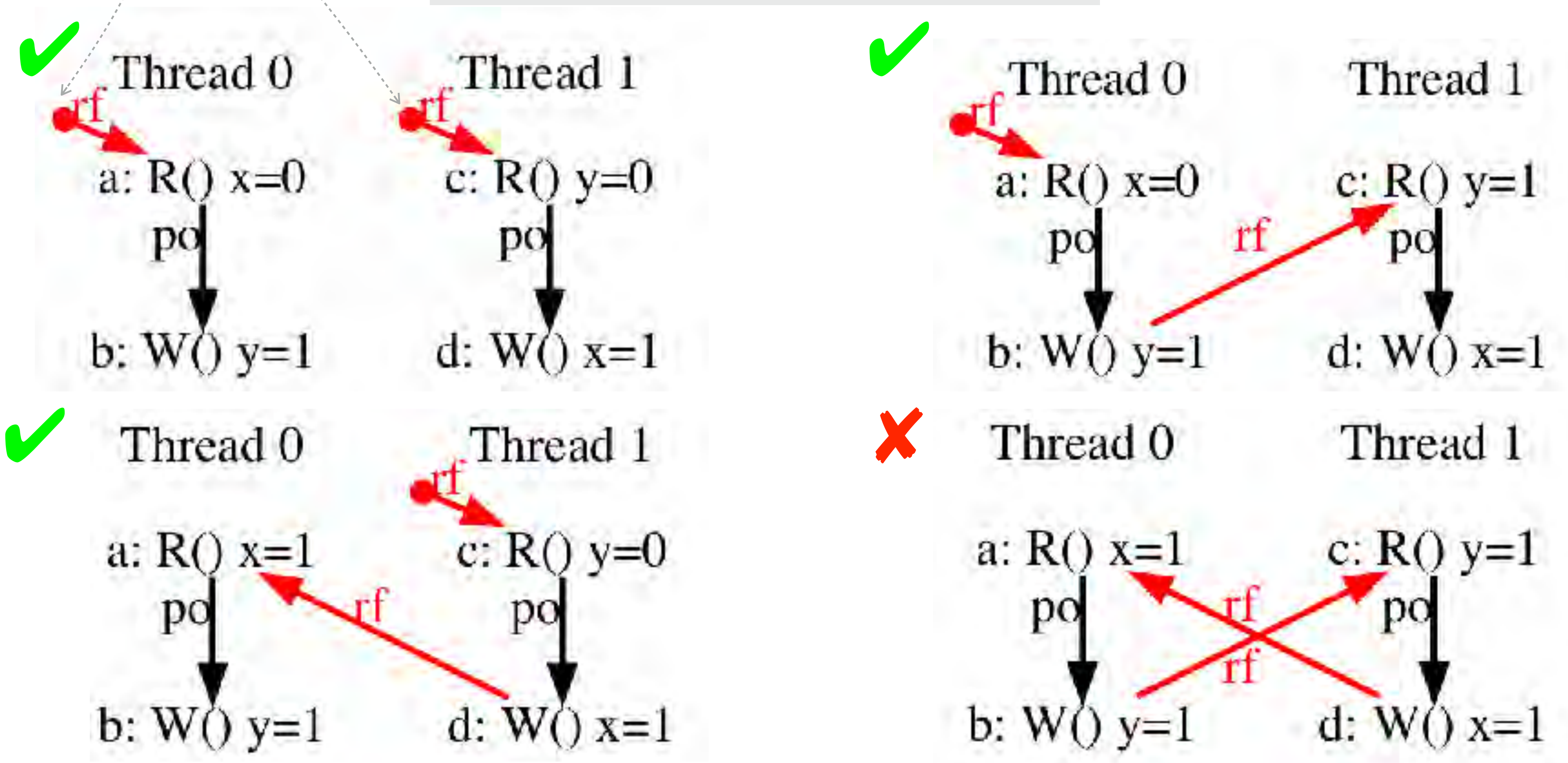
Analytic semantics

- Anarchic semantics: all possible executions with cuts/histories with no restriction on rf (any read can read any value from any write to the same shared variable)
- Communication consistency: requirements on rf specified on an abstraction to a candidate execution
- Analytic semantics: all executions with cuts/histories which rf satisfies the consistency requirements

Example of anarchic semantics: LB

```
{ x = 0; y = 0; }
 P0          | P1          ;
 r[] r1 x | r[] r2 y ;
 w[] y 1   | w[] x 1   ;
```

read from initial write

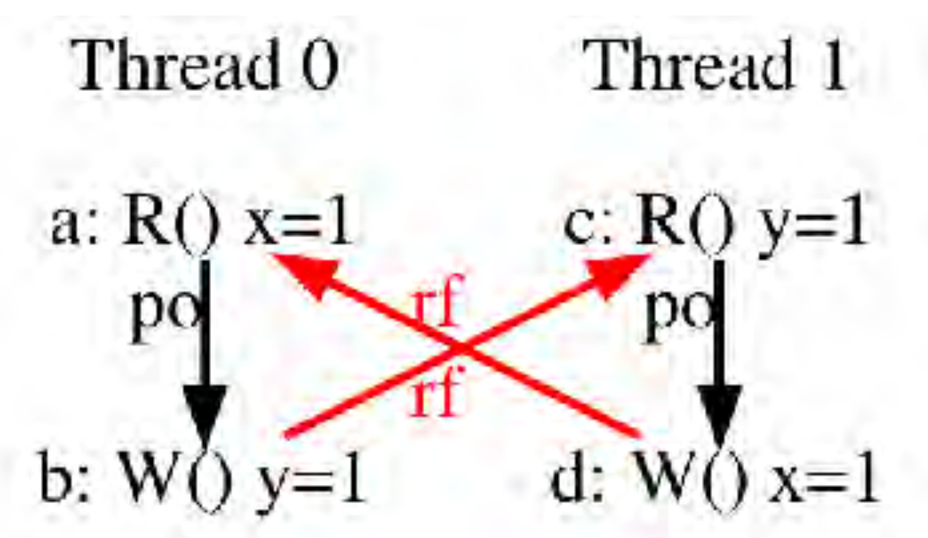


J. Alglave and L. Maranget. herd7. virginia.cs.ucl.ac.uk/herd, 31 Aug. 2015.

Example of communication specification in the `cat` language for LB

```
irreflexive (po | rf)+
```

Rejects only the anarchic execution:

✗ 

J. Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7, 2010.

J. Alglave, P. Cousot, and L. Maranget. Syntax and semantics of the `cat` language. *HSA Foundation*, Version 1.1:38 p., 16 Oct 2015b. URL `http://www.hsafoundation.com/?ddownload=5382`.

Examples of architecture specification

- SC (sequential consistency):

```
let co = (IW*W) & loc
let fr = (rf^-1;co)
acyclic po | rf | co | fr as sc
```

- TSO:

```
let co = (IW*W) & loc
let fr = (rf^-1;co)
let po-loc = po & loc
acyclic po-loc | rf | co | fr as scpv
let ppo = po \ (W*R)
let rfe = rf & ext
acyclic ppo | rfe | co | fr as tso
```

- For lb:

```
acyclic (po | rf) as lb
```

sc $\Rightarrow$ lb, tso $\not\Rightarrow$ lb

Fence specification:

- In `Lisa`:

```
{ x = 0; y = 0; }
 P0          | P1          ;
 r[] r1 x    | r[] r2 y    ;
 f[dep]      | f[lw]       ;
 w[] y 1     | w[] x 1     ;
```

- Implementation with dependencies and fences in TSO:

```
{ x = 0; y = 0; }
 P0                | P1         ;
 r[] r1 x          | r[] r2 y   ;
 r2 = xor r1 r1    | mfence     ;
 r3 = r2+1         |
 w[] y r3          | w[] x r3   ;
```

data dependency

`cat`

- Handles one history at a time
- For each execution relies on:
  - the set E (_) of events of the execution (partitionned into initial writes `IW`, writes `W`, read `R`, fences `F`, `...`
  - the program order `po` of events per process
  - the read-from relation `rf` per variable
- Has predefined relations `loc`, `ext`, ...
- Can define new relations e.g. `*`, `;`, |, &, \, `+`, `^-1`, ...
- Accepts/eliminates the execution by defining relations r and checking `irreflexive r`, `acyclic r`, `empty r`, `not empty r`

ARM in cat

```
let fr = rf^-1;co
acyclic po-loc | rf | co | fr as scpv

let deps = addr | data
let rdw = po-loc & (fre;rfe)
let detour = po-loc & (coe ; rfe)

let ii0 = deps | rfi | rdw
let ic0 = 0
let ci0 = ctrlcfence(ISB) | detour
let cc0 = deps | ctrl | (addr;po)

let rec ii = ii0 | ci | (ic;ci) | (ii;ii)
and ic = ic0 | ii | cc | (ic;cc) | (ii;ic)
and ci = ci0 | (ci;ii) | (cc;ci)
and cc = cc0 | ci | (ci;ic) | (cc;cc)

let ppo = ii & R*R | ic & R*W

let dmb = fencerel(DMB)
let dsb = fencerel(DSB)
let fences = dmb|dsb
let A-cumul = rfe;fences

let hb = ppo | fences | rfe
acyclic hb as no-thin-air

let prop-base = (fences | A-cumul);hb*
let prop = (prop-base & W*W)| (com*; prop-base*; fences; hb*)

irreflexive fre;prop;hb* as observation
acyclic co | prop as propagation
```


Invariance proof method for weakly consistent parallelism

Difficulties

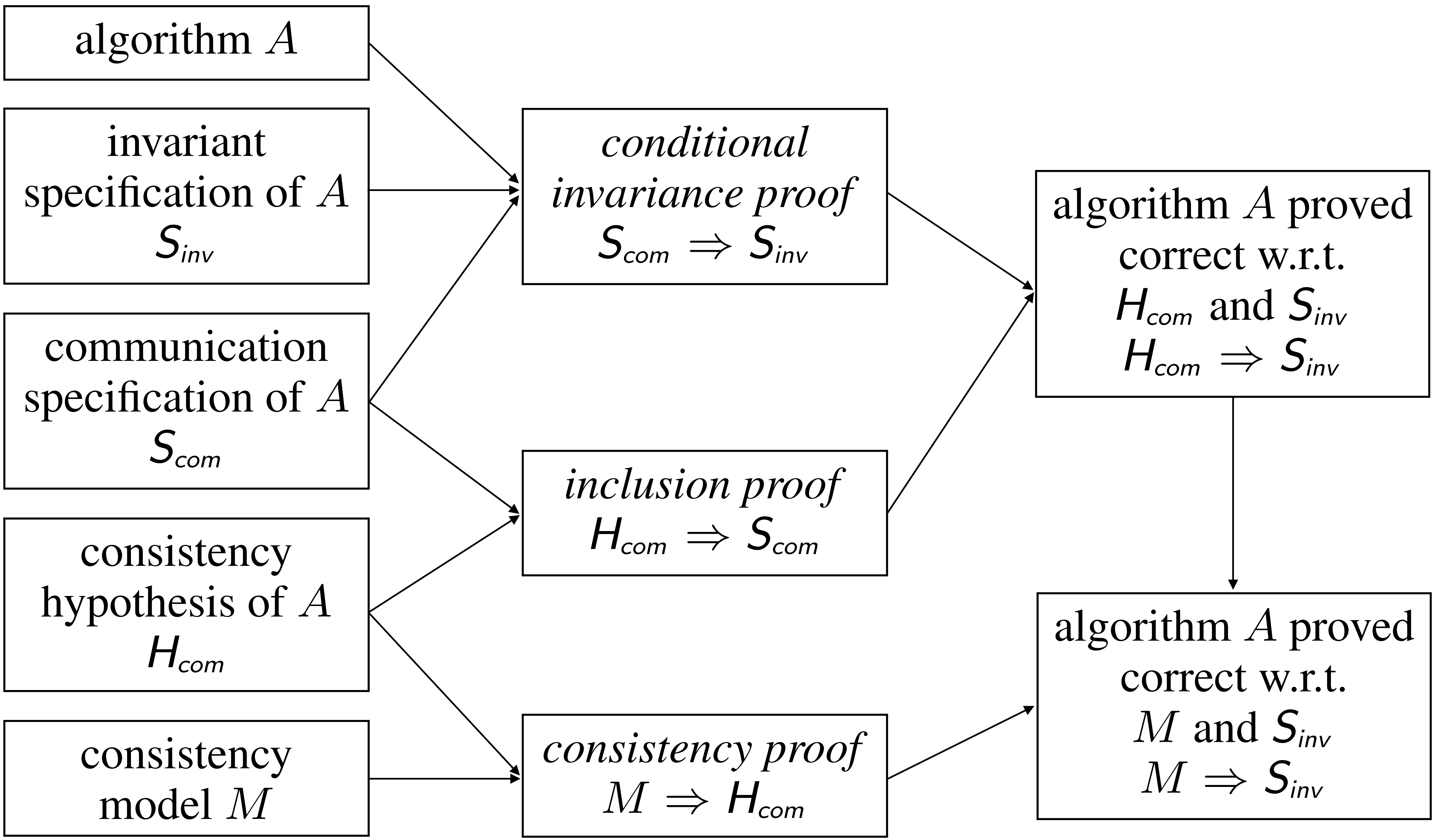
- There is no longer a notion of instantaneous value of the shared variables:
  - ⇒ pythia variables (denoting values of variables when read)
  - ⇒ communications rf (keeping track of which writes events the pythia variables take there values from)
  - ⇒ stamps (keeping track of events to distinguish different instruction executions)

Difficulties

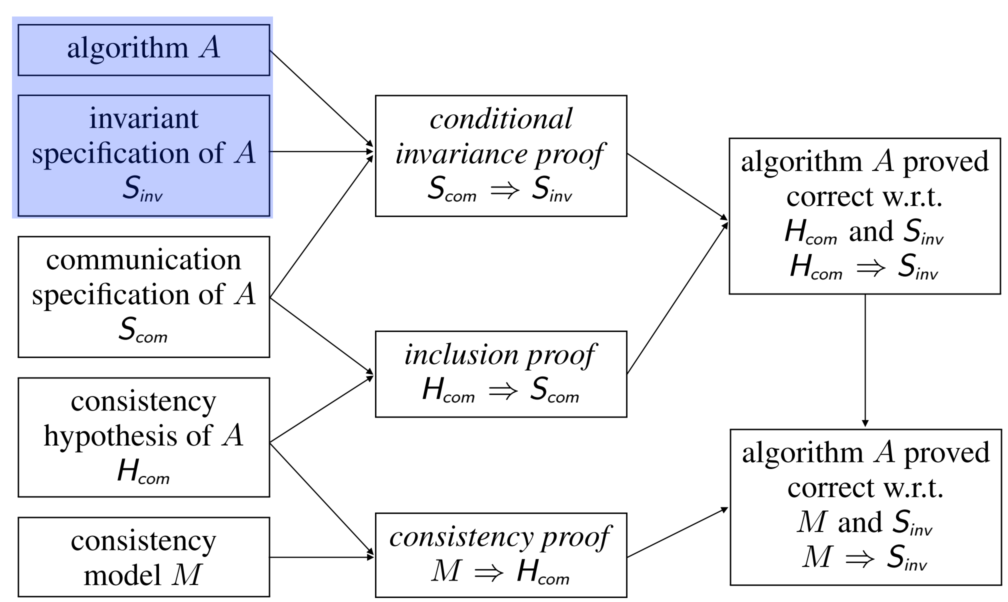
- We have to make hypotheses on how communications do happen:
  $\Rightarrow$ communication specification S_{com}

- We have to show that the communication specification is correctly implemented on an architecture:
  $\Rightarrow$ a way to mix invariant S_{com} and `cat` specifications

Methodology



Invariant


algorithm A
invariant specification of A S_{inv}
communication specification of A S_{com}
consistency hypothesis of A H_{com}
consistency model M
conditional invariance proof $S_{com} \Rightarrow S_{inv}$
inclusion proof $H_{com} \Rightarrow S_{com}$
consistency proof $M \Rightarrow H_{com}$
algorithm A proved correct w.r.t. H_{com} and S_{inv} $H_{com} \Rightarrow S_{inv}$
algorithm A proved correct w.r.t. M and S_{inv} $M \Rightarrow S_{inv}$

Pythia variables

- Unique name given to communicated values during execution (using stamps)

0:{ w F1 false; w F2 false; w T 0; }

P0:

1:w[] F1 true
2:w[] T 2
3:repeat {i}
4:    r[] R1 F2 {$\leadsto \mathtt{F2}_{4}^{i}$}
5:    r[] R2 T {$\leadsto \mathtt{T}_{5}^{i}$}
6:until $\neg$R1 $\vee$ R2 = 1 {i_{end}}
7:skip (* CS1 *)
8:w[] F1 false
9:

$\|$

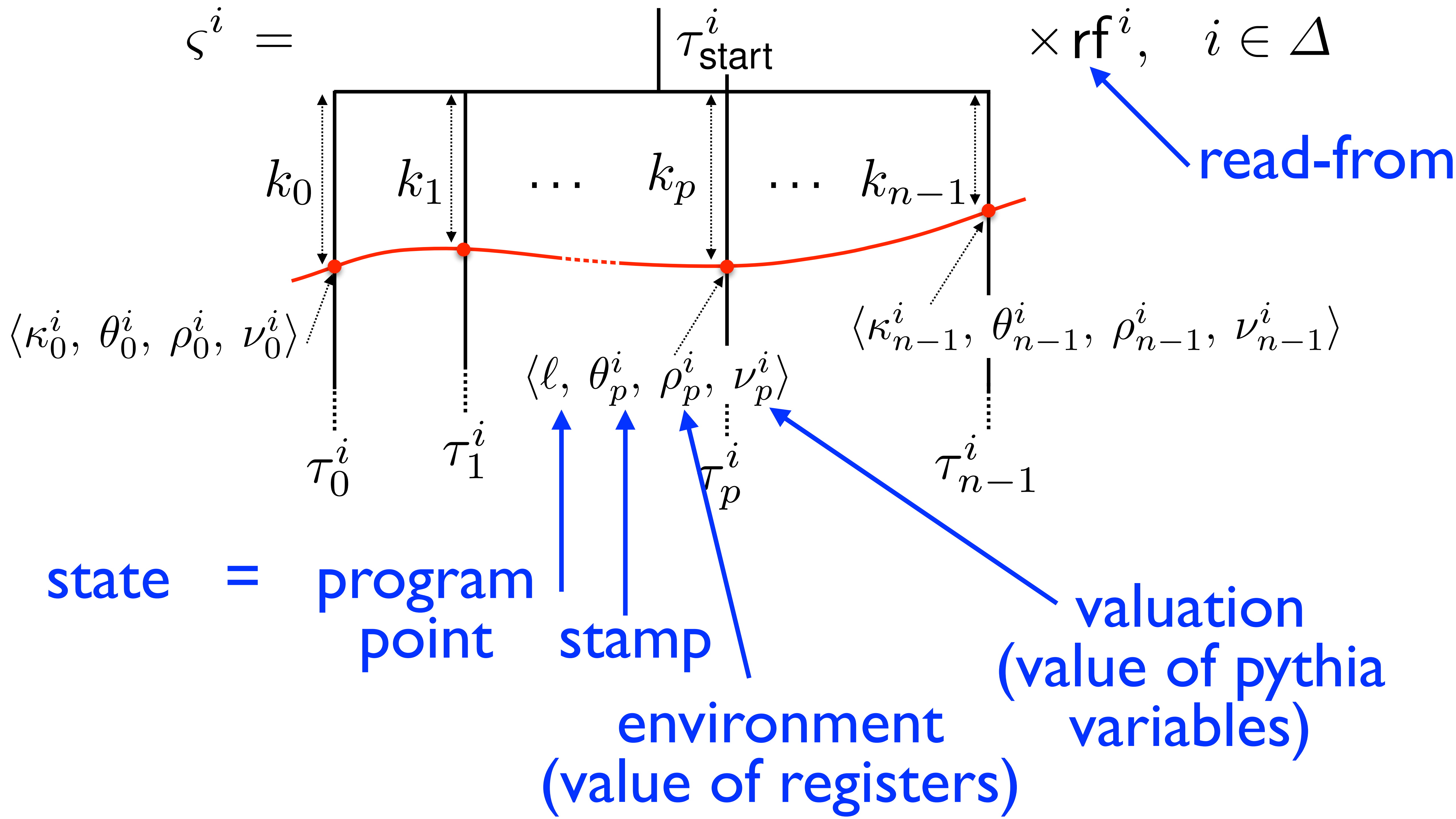
P1:

10:w[] F2 true;
11:w[] T 1;
12:repeat {j}
13:    r[] R3 F1; {$\leadsto \mathtt{F1}_{13}^{j}$}
14:    r[] R4 T; {$\leadsto \mathtt{T}_{14}^{j}$}
15:until $\neg$R3 $\vee$ R4 = 2; {j_{end}}
16:skip (* CS2 *)
17:w[] F2 false;
18:

Stamp: label,counter

Pythia variables

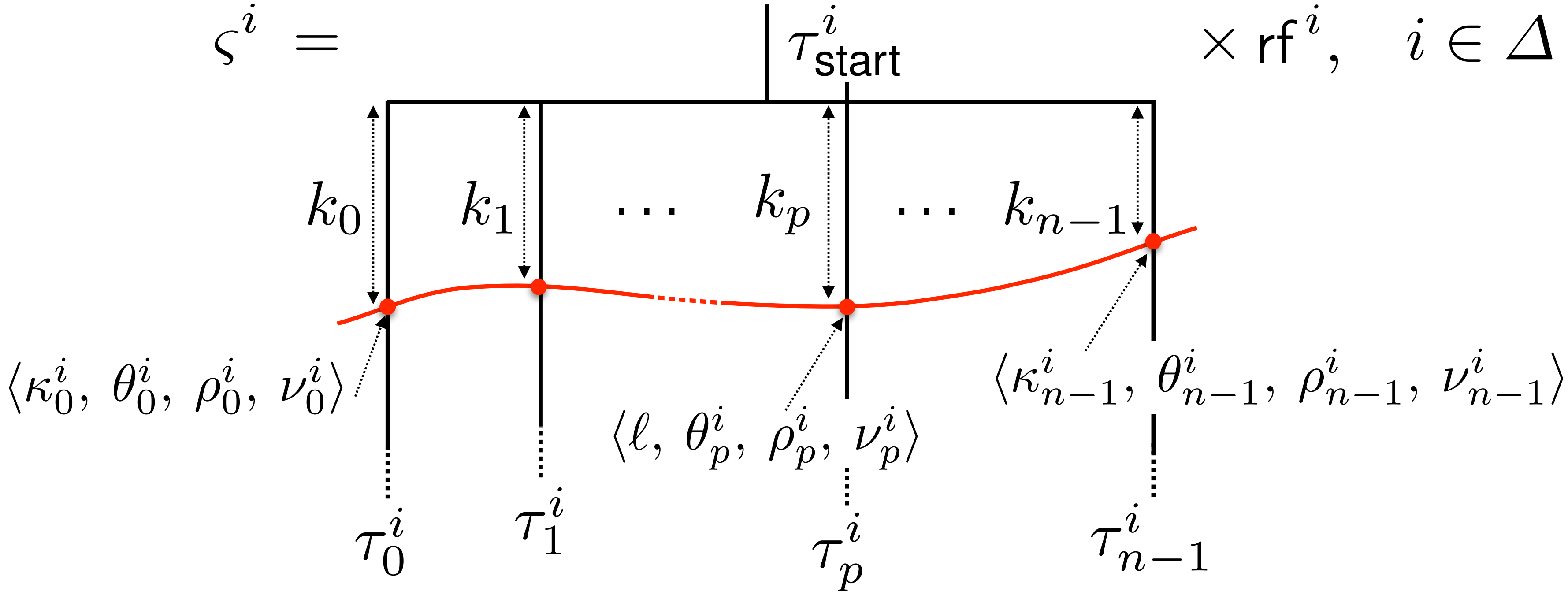
Invariance abstraction



Invariance abstraction

$$\pi^i = \varsigma^i \times \Gamma^i$$

$$\varsigma^i = \quad \ldots \quad \times\, \mathsf{rf}^i, \quad i \in \Delta$$



$$\alpha_a(\{\pi^i \mid i \in \Delta\}) \triangleq \prod_{p \in \mathbb{P}\mathfrak{i}} \prod_{\ell \in \mathbb{L}(p)} \bigcup_{i \in \Delta} \{\langle \kappa^i_{0,k_0}, \theta^i_{0,k_0}, \rho^i_{0,k_0}, \nu^i_{0,k_0}, \ldots,$$
$$\nu^i_{p-1,k_{p-1}}, \theta^i_{p,k_p}, \rho^i_{p,k_p}, \nu^i_{p,k_p}, \kappa^i_{p+1,k_{p+1}}, \ldots, \kappa^i_{n-1,k_{n-1}}, \theta^i_{n-1,k_{n-1}},$$
$$\rho^i_{n-1,k_{n-1}}, \nu^i_{n-1,k_{n-1}}, \mathsf{rf}^i\rangle \mid \forall q \in [0, n[\setminus \{p\} \,.\, \underline{\tau^i_q}_{k_q} =$$
$$\mathfrak{s}\langle \kappa^i_{q,k_q},\ \theta^i_{q,k_q},\ \rho^i_{q,k_q},\ \nu^i_{q,k_q} \rangle \wedge \underline{\tau^i_p}_{k_p} = \mathfrak{s}\langle \ell,\ \theta^i_{p,k_p},\ \rho^i_{p,k_p},\ \nu^i_{p,k_p} \rangle\}.$$

Invariant

- An invariant $S_{inv}(p)$ at point p of process P_i is a statement relating
  - the program points $p_1,\ldots,p_{i-1}, p_{i+1},\ldots, p_m$ of the other processes
  - the pythia variables (forbidden to mention of shared variables)
  - the local registers of all processes
  - the communications (rf)

which always holds when at the cut where execution reaches point p of process Pi and the other processes are at $p_1,\ldots,p_{i-1}, p_{i+1},\ldots, p_m$

Example (Peterson)

```
0: { w F1 false; w F2 false; w T 0; }
```

$\{F1=false \;\wedge\; F2=false \;\wedge\; T=0\}$ }

Thread 1:

1: $\{R1=0 \;\wedge\; R2=0\}$
w[] F1 true
2: $\{R1=0 \;\wedge\; R2=0\}$
w[] T 2
3: $\{R1=0 \;\wedge\; R2=0\}$
do $\{i\}$
4: $\{(i=0 \;\wedge\; R1=0 \;\wedge\; R2=0) \vee (i>0 \;\wedge\; R1=F2_4^{i-1} \;\wedge\; R2=T_5^{i-1})\}$
r[] R1 F2 $\{\leadsto F2_4^i\}$
5: $\{R1=F2_4^i \wedge (i=0 \;\wedge\; R2=0) \vee (i>0 \;\wedge\; R2=T_5^{i-1})\}$
r[] R2 T $\{\leadsto T_5^i\}$
6: $\{R1=F2_4^i \wedge R2=T_5^i\}$
while R1 $\wedge$ R2$\neq$1 $\{i_{end}\}$
7: $\{\neg F2_4^{i_{end}} \vee T_5^{i_{end}}=1\}$
skip (* CS1 *)
8: $\{\neg F2_4^{i_{end}} \vee T_5^{i_{end}}=1\}$
w[] F1 false
9: $\{\neg F2_4^{i_{end}} \vee T_5^{i_{end}}=1\}$

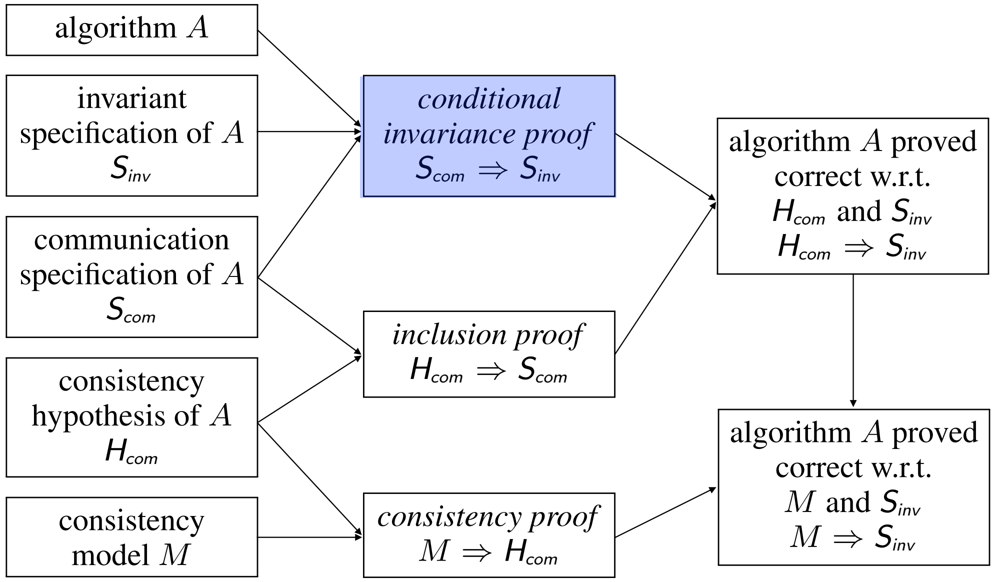
||

Thread 2:

10: $\{R3=0 \;\wedge\; R4=0\}$
w[] F2 true;
11: $\{R3=0 \;\wedge\; R4=0\}$
w[] T 1;
12: $\{R3=0 \;\wedge\; R4=0\}$
do $\{j\}$
13: $\{(j=0 \;\wedge\; R3=0 \;\wedge\; R4=0) \vee (j>0 \;\wedge\; R3=F1_{13}^{j-1} \;\wedge\; R4=T_{14}^{j-1})\}$
r[] R3 F1 $\{\leadsto F1_{13}^j\}$;
14: $\{R3=F1_{13}^j \wedge (j=0 \;\wedge\; R4=0) \vee (j>0 \;\wedge\; R4=T_{14}^{j-1})\}$
r[] R4 T; $\{\leadsto T_{14}^j\}$
15: $\{R3=F1_{13}^j \;\wedge\; R4=T_{14}^j)\}$
while R3 $\wedge$ R4$\neq$2 $\{j_{end}\}$;
16: $\{\neg F1_{13}^{j_{end}} \vee T_{14}^{j_{end}}=2\}$
skip (* CS2 *)
17: $\{\neg F1_{13}^{j_{end}} \vee T_{14}^{j_{end}}=2\}$
w[] F2 false;
18: $\{\neg F1_{13}^{j_{end}} \vee T_{14}^{j_{end}}=2\}$

(these invariants are for the anarchic semantics, so all communications are possible, no constraints on rf)

Invariance proof



Verification conditions

- Sequential proof
- Absence of interference proof
- Communication proof

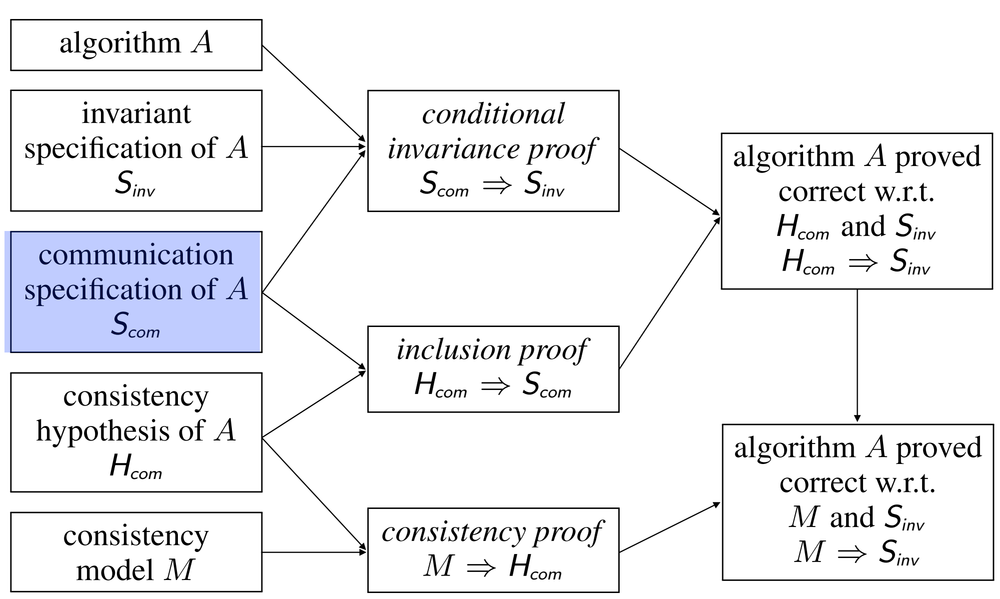
Examples:

- $\{$ P(R, ..., rf) $\wedge \langle w(\mathrm{x}, v), r(\theta, \mathrm{x})\rangle \in$ rf $\}$
  `read x R {` $\rightsquigarrow$ x_θ `}` .... *communication*
  $\{$ P[R $\leftarrow \mathrm{x}_\theta$, $\mathrm{x}_\theta \leftarrow v$, ..., rf]$\}$
- { P } `fence` { P } (fences are markers in the execution)
- { P } `write R x` { P } (a write has no local effect)

Communication proof

- The communications rf must be checked to be well-formed (none allowed by H_{cm} should miss, see later)
- If $\langle w(\mathrm{P}, \mathrm{p}, \theta, \mathtt{x}, \upsilon), r(\mathrm{P'}, \mathrm{p'}, \theta', \mathtt{x}, \mathtt{x}_{\theta'})\rangle \in$ rf then:
  - The read instruction of at point p' process P' must read from an *initial* or a *reachable* write
  - A read event (for a given stamp θ') must read from a *unique write event* with the same variable $\mathtt{x}$
  - The *value assigned* to the read pythia variable $\mathtt{x}_{\theta'}$ must be that of υ the matching write

Communication specification S_{com}



Communication specification

- The algorithm A is often incorrect for the anarchic semantics
- The allowable communications are specified by a communication specification S_{com} (i.e. an invariant constraining the allowed communications rf)
- This communication specification can often be calculated from the anarchic invariant and the inductive invariant S_{ind}

Example (Peterson)

$$\begin{array}{l} \quad \texttt{at 7} \wedge \texttt{at 16} \\ \Rightarrow (\neg \texttt{F2}_4^{i_{\text{end}}} \vee \texttt{T}_5^{i_{\text{end}}} = 1) \wedge (\neg \texttt{F1}_{13}^{j_{\text{end}}} \vee \texttt{T}_{14}^{j_{\text{end}}} = 2)\} \\ \qquad\qquad\qquad\qquad \text{(i.e. the invariants at lines } \texttt{7:} \text{ and } \texttt{16:} \text{ hold)} \\ \Rightarrow \neg \mathcal{S}_{com} \quad \text{(since by taking } i = i_{\text{end}} \text{ and } j = j_{\text{end}}\text{, we have} \\ \qquad\qquad (\texttt{F2}_4^i = \mathsf{false} \vee \texttt{T}_5^i = 1) \wedge (\texttt{F1}_{13}^j = \mathsf{false} \vee \texttt{T}_{14}^j = 2)\text{)} \end{array}$$

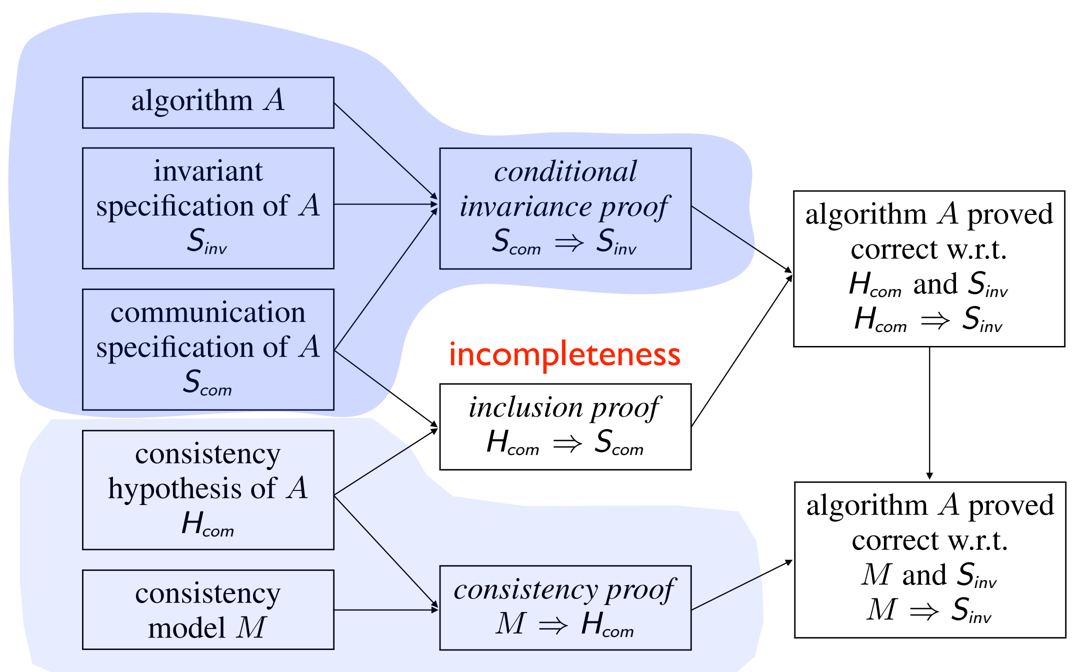
so that Peterson has been proved correct under the hypothesis that the communication specification $\mathcal{S}_{com}$ holds:

$$\begin{array}{rl} \mathcal{S}_{com} \triangleq \neg[\exists i, j.[\mathfrak{rf}\langle \texttt{F2}_4^i,\ \langle \texttt{0:},\ \mathsf{false}\rangle\rangle \vee \mathfrak{rf}\langle \texttt{F2}_4^i,\ \langle \texttt{17:},\ \mathsf{false}\rangle\rangle \\ \vee\ \mathfrak{rf}\langle \texttt{T}_5^i,\ \langle \texttt{11:},\ 1\rangle\rangle] \wedge [\mathfrak{rf}\langle \texttt{F1}_{13}^j,\ \langle \texttt{0:},\ \mathsf{false}\rangle\rangle \\ \vee\ \mathfrak{rf}\langle \texttt{F1}_{13}^j,\ \langle \texttt{8:},\ \mathsf{false}\rangle\rangle \vee \mathfrak{rf}\langle \texttt{T}_{14}^j,\ \langle \texttt{2:},\ 2\rangle\rangle]] \end{array}$$

(preventing the incorrect case)

Soundness and completeness

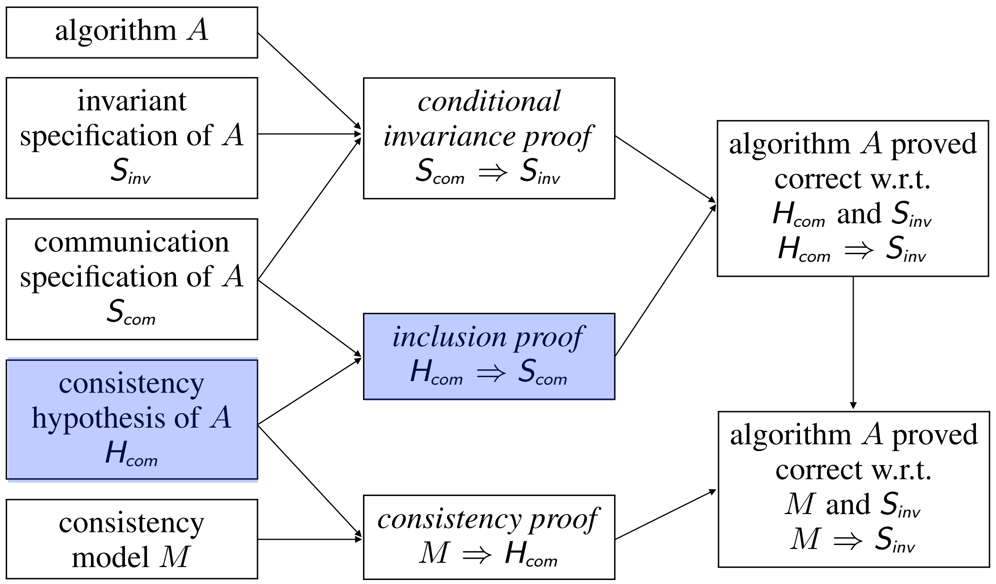
- The invariance proof method is derived from the truly parallel semantics with cuts by calculational design $\Rightarrow$ soundness and (relative) completeness



- A consistency specification H_{com} may be less expressive than S_{com} $\Rightarrow$ *incompleteness* (*)

(*) e.g. hardware cannot restrict a read to input from writes writing odd numbers.

Consistency hypothesis and inclusion proof



Consistency hypothesis

- The communication specification S_{com} is useful to reason on invariance, but not on machine architecture
- We express S_{com} as a consistency hypothesis H_{com} expressed in the `cat` language
- H_{com} is derived from S_{com} by calculations design while doing the inclusion proof

Inclusion proof

- Inclusion proof: $\neg S_{com} \Rightarrow \neg H_{com}$

- Calculational design of H_{com}:

  - Calculate all possible execution scenarios violating S_{com}

$$S_{com} \triangleq \neg[\exists i,j.[\mathfrak{rf}\langle \mathtt{F2}^{i}_{4},\ \langle 0\!:,\ \mathsf{false}\rangle\rangle \vee \mathfrak{rf}\langle \mathtt{F2}^{i}_{4},\ \langle 17\!:,\ \mathsf{false}\rangle\rangle \vee \mathfrak{rf}\langle \mathtt{T}^{i}_{5},\ \langle 11\!:,\ 1\rangle\rangle] \wedge [\mathfrak{rf}\langle \mathtt{F1}^{j}_{13},\ \langle 0\!:,\ \mathsf{false}\rangle\rangle \vee \mathfrak{rf}\langle \mathtt{F1}^{j}_{13},\ \langle 8\!:,\ \mathsf{false}\rangle\rangle \vee \mathfrak{rf}\langle \mathtt{T}^{j}_{14},\ \langle 2\!:,\ 2\rangle\rangle]]$$

  - Prevent each of them by a `cat` specification

  - H_{com} is their conjunction

Example: Peterson

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                        || P1:
1:w[] F1 true              || 10:w[] F2 true;
2:w[] T 2                  || 11:w[] T 1;
3:do                       || 12:do
4:   r[] R1 F2             || 13:   r[] R3 F1;
5:   r[] R2 T              || 14:   r[] R4 T;
6:while R1 ∧ R2 ≠ 1        || 15:while R3 ∧ R4 ≠ 2;
7:   f[p0] (* CS1 *)       || 16:   f[p1] (* CS2 *)
8:w[] F1 false             || 17:w[] F2 false;
```

$4 \xrightarrow{fr} 10 \xrightarrow{po} 13 \xrightarrow{fr} 1 \xrightarrow{po} 4$

case 1: 0:F2,0:F1

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                        || P1:
1:w[] F1 true              || 10:w[] F2 true;
2:w[] T 2                  || 11:w[] T 1;
3:do                       || 12:do
4:   r[] R1 F2             || 13:   r[] R3 F1;
5:   r[] R2 T              || 14:   r[] R4 T;
6:while R1 ∧ R2 ≠ 1        || 15:while R3 ∧ R4 ≠ 2;
7:   f[p0] (* CS1 *)       || 16:   f[p1] (* CS2 *)
8:w[] F1 false             || 17:w[] F2 false;
```

$14 \xrightarrow{fr} 11 \xrightarrow{po} 14$

case 2a: 0:F2,1:F1 ($2 \xrightarrow{co} 11$)

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                        || P1:
1:w[] F1 true              || 10:w[] F2 true;
2:w[] T 2                  || 11:w[] T 1;
3:do                       || 12:do
4:   r[] R1 F2             || 13:   r[] R3 F1;
5:   r[] R2 T              || 14:   r[] R4 T;
6:while R1 ∧ R2 ≠ 1        || 15:while R3 ∧ R4 ≠ 2;
7:   f[p0] (* CS1 *)       || 16:   f[p1] (* CS2 *)
8:w[] F1 false             || 17:w[] F2 false;
```

$11 \xrightarrow{co} 2 \xrightarrow{po} 4 \xrightarrow{fr} 10 \xrightarrow{po} 11$

case 2b: 0:F2,1:F1 ($11 \xrightarrow{co} 2$)

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                        || P1:
1:w[] F1 true              || 10:w[] F2 true;
2:w[] T 2                  || 11:w[] T 1;
3:do                       || 12:do
4:   r[] R1 F2             || 13:   r[] R3 F1;
5:   r[] R2 T              || 14:   r[] R4 T;
6:while R1 ∧ R2 ≠ 1        || 15:while R3 ∧ R4 ≠ 2;
7:   f[p0] (* CS1 *)       || 16:   f[p1] (* CS2 *)
8:w[] F1 false             || 17:w[] F2 false;
```

$2 \xrightarrow{co} 11 \xrightarrow{po} 13 \xrightarrow{fr} 1 \xrightarrow{po} 2$

case 3a: 10:F2,0:F1 ($2 \xrightarrow{co} 11$)

Example: Peterson

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                      || P1:
1:w[] F1 true            || 10:w[] F2 true;
2:w[] T 2                || 11:w[] T 1;
3:do                     || 12:do
4:    r[] R1 F2          || 13:    r[] R3 F1;
5:    r[] R2 T           || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1      || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)    || 16:    f[p1] (* CS2 *)
8:w[] F1 false           || 17:w[] F2 false;
```

$5 \xrightarrow{\texttt{fr}} 2 \xrightarrow{\texttt{po}} 5$

case 3b: `10:F2,0:F1` ($11 \xrightarrow{\texttt{co}} 2$)

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                      || P1:
1:w[] F1 true            || 10:w[] F2 true;
2:w[] T 2                || 11:w[] T 1;
3:do                     || 12:do
4:    r[] R1 F2          || 13:    r[] R3 F1;
5:    r[] R2 T           || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1      || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)    || 16:    f[p1] (* CS2 *)
8:w[] F1 false           || 17:w[] F2 false;
```

$14 \xrightarrow{\texttt{fr}} 11 \xrightarrow{\texttt{po}} 14$

case 4a: `10:F2,1:F1` ($2 \xrightarrow{\texttt{co}} 11$)

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                      || P1:
1:w[] F1 true            || 10:w[] F2 true;
2:w[] T 2                || 11:w[] T 1;
3:do                     || 12:do
4:    r[] R1 F2          || 13:    r[] R3 F1;
5:    r[] R2 T           || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1      || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)    || 16:    f[p1] (* CS2 *)
8:w[] F1 false           || 17:w[] F2 false;
```

$5 \xrightarrow{\texttt{fr}} 2 \xrightarrow{\texttt{po}} 5$

case 4b: `10:F2,1:F1` ($11 \xrightarrow{\texttt{co}} 2$)

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                      || P1:
1:w[] F1 true            || 10:w[] F2 true;
2:w[] T 2                || 11:w[] T 1;
3:do                     || 12:do
4:    r[] R1 F2          || 13:    r[] R3 F1;
5:    r[] R2 T           || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1      || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)    || 16:    f[p1] (* CS2 *)
8:w[] F1 false           || 17:w[] F2 false;
```

$4 \xrightarrow{\texttt{po}} 8 \xrightarrow{\texttt{rf}} 13 \xrightarrow{\texttt{po}} 17 \xrightarrow{\texttt{rf}} 4$

case 5: `17:F2,8:F1`

Example: Peterson

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                      || P1:
1:w[] F1 true            || 10:w[] F2 true;
2:w[] T 2                || 11:w[] T 1;
3:do                     || 12:do
4:    r[] R1 F2          || 13:    r[] R3 F1;
5:    r[] R2 T           || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1      || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)    || 16:    f[p1] (* CS2 *)
8:w[] F1 false           || 17:w[] F2 false;
```

7 —po→ 8 —rf→ 13 —po→ 17 —cut→ 7

case 6: 8:F1

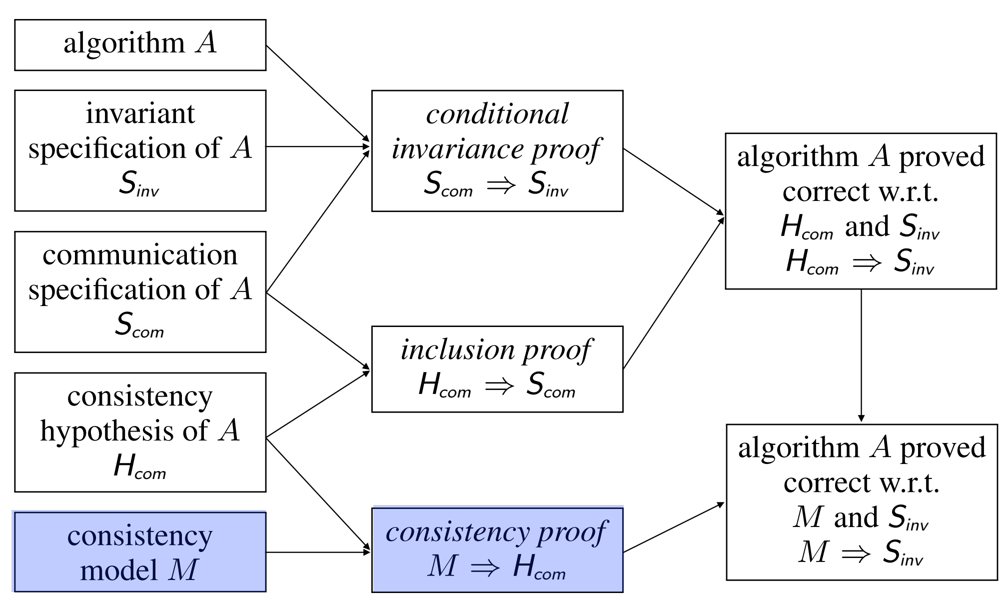
```
0:{ w F1 false; w F2 false; w T 0; }
P0:                      || P1:
1:w[] F1 true            || 10:w[] F2 true;
2:w[] T 2                || 11:w[] T 1;
3:do                     || 12:do
4:    r[] R1 F2          || 13:    r[] R3 F1;
5:    r[] R2 T           || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1      || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)    || 16:    f[p1] (* CS2 *)
8:w[] F1 false           || 17:w[] F2 false;
```

16 —po→ 17 —rf→ 4 —po→ 7 —cut→ 16

case 7: 17:F2

- the cut relation can be expressed in cat using tags on fence markers f[p0] and f[p1]
- H_{com} is
  irreflexive fr;po;fr;po
  irreflexive fr;po
  irreflexive co;po;fr;po
  irreflexive po;rf;po;rf
  irreflexive po;rf;po;*cut*

Consistency model and proof



Example: Peterson in SC

- H_{com} is
  ```
  irreflexive fr;po;fr;po
  irreflexive fr;po
  irreflexive co;po;fr;po
  irreflexive po;rf;po;rf
  ```
  irreflexive po;rf;po;cut

- Sequential consistency in cat:
  ```
  let fr = (rf^-1 ; co)
  acyclic po | rf | co | fr as sc
  ```

- Forbid all first 4 cases

Example: Peterson in SC

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                         || P1:
1:w[] F1 true               || 10:w[] F2 true;
2:w[] T 2                   || 11:w[] T 1;
3:do                        || 12:do
4:    r[] R1 F2             || 13:    r[] R3 F1;
5:    r[] R2 T              || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1         || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)       || 16:    f[p1] (* CS2 *)
8:w[] F1 false              || 17:w[] F2 false;
```

$7 \xrightarrow{\text{po}} 8 \xrightarrow{\text{rf}} 13 \xrightarrow{\text{po}} 17 \xrightarrow{\text{cut}} 7$

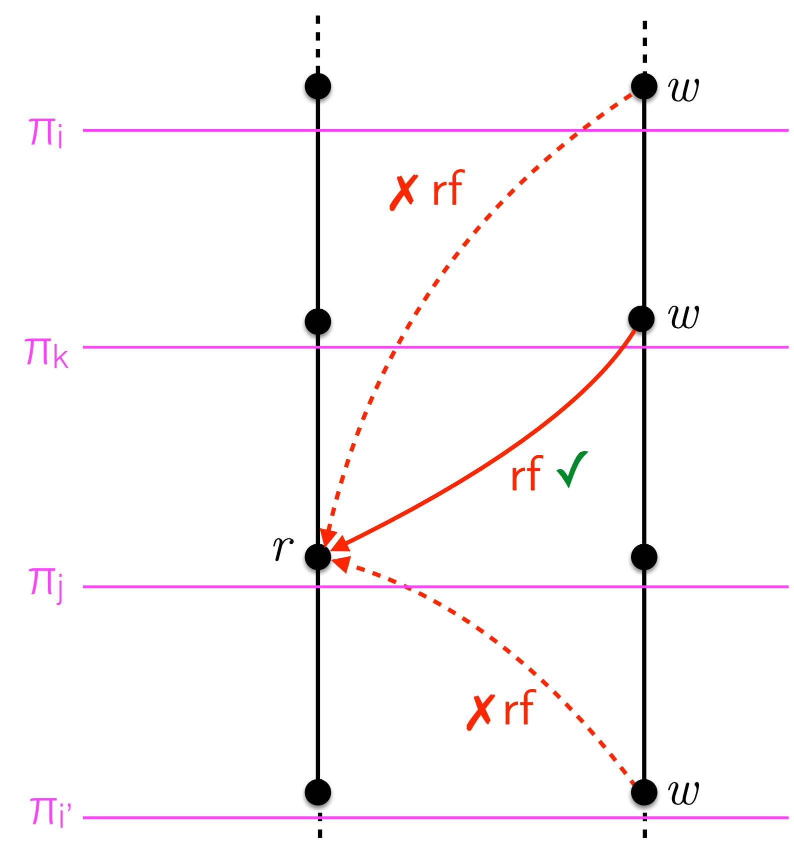
case 6: 8:F1

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                         || P1:
1:w[] F1 true               || 10:w[] F2 true;
2:w[] T 2                   || 11:w[] T 1;
3:do                        || 12:do
4:    r[] R1 F2             || 13:    r[] R3 F1;
5:    r[] R2 T              || 14:    r[] R4 T;
6:while R1 ∧ R2 ≠ 1         || 15:while R3 ∧ R4 ≠ 2;
7:    f[p0] (* CS1 *)       || 16:    f[p1] (* CS2 *)
8:w[] F1 false              || 17:w[] F2 false;
```

$16 \xrightarrow{\text{po}} 17 \xrightarrow{\text{rf}} 4 \xrightarrow{\text{po}} 7 \xrightarrow{\text{cut}} 16$

case 7: 17:F2

- The last case follows from the truly parallel execution trace semantics with cuts for sequential consistency



Example: Peterson in TSO

- H_{com} is <u>not</u> forbidden by TSO:

```
let fr = (rf^-1;co)
let po-loc = po & loc
acyclic po-loc | rf | co | fr as scpv
let ppo = po \ (W*R)
let rfe = rf & ext
acyclic ppo | rfe | co | fr as tso
```

- For example the case 1,

  $\langle w_1, r_4 \rangle \in$ fr ; po ; fr ; po

  is not forbidden by TSO since $\langle w, r \rangle$ pairs on different variables are excluded from ppo.

Implementation with (weak) cat fences

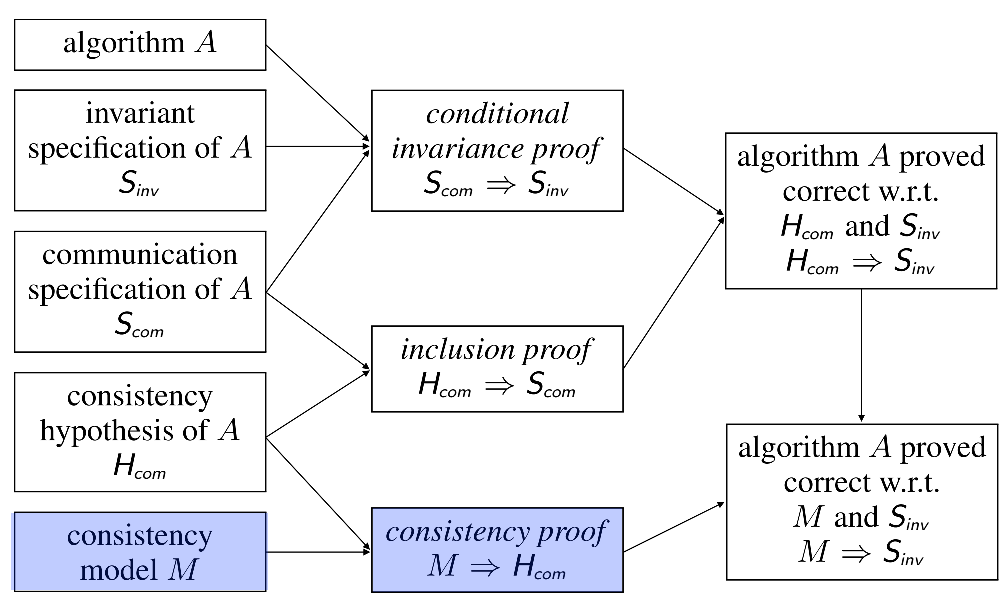
Implementation with fences

```
0:{ F1 = 0; F2 = 0; T = 0; }
1:   w[] F1 1              | 10:   w[] F2 1          ;
2:   w[] T 2               | 11:   w[] T 1           ;
3:   do                    | 12:   do                ;
       f[fhw]              |         f[fhw]          ;
4:     r[] r1 F2           | 13: r[] r3 F1           ;
5:     r[] r2 T            | 14: r[] r4 T            ;
6:   while r1 ∧ r2 ≠ 1     | 15: while r3 ∧ r4 ≠ 1   ;
7:   (* CS1 *)             | 16: (* CS2 *)           ;
       f[fhw]              |         f[fhw]          ;
8:   w[] F1 0              | L17: w[] F2 0           ;
```

let fhw = (po & (_ * F)) ; po
let fre = (rf^-1;co) & ext
irreflexive fhw;fre; fhw;fre
...

- Invariance proof unchanged (`fence` = `skip`)
- Proved to imply the previous fenceless `cat` specification
- so S_{com} unchanged

consistency proof



Example: Peterson

- The proof is valid for the virtual machine defined by the cat specification Peterson
- Porting the algorithm to a different machine M' just need refencing (and redoing the proof $M' \Rightarrow H_{cm}$)
- On machine architecture stronger fences have to be used:
  - SC: `fhw` = no fence
  - TSO: `fhw` = `mfence`
  - ARM: `fhw` = `dbm` | `dsb`

Conclusion

Algorithm design methodology

1. Design the algorithm A and its specification S in the sequential consistency model of parallelism
2. Consider the anarchic semantics of algorithm A
3. Add communication specifications S_{com} to restrict anarchic communications and ensure the correctness of A with respect to specification S
4. Do the invariance proof under WCM with S_{com}
5. Infer H_{cm} in `cat` from S_{com}
6. Prove that the machine memory model M in `cat` implies H_{cm}

Conclusion

- Modern machines have complex memory models

$\Rightarrow$ portability has a price (refencing)

$\Rightarrow$ debugging is very hard/quasi-impossible

$\Rightarrow$ proofs are much harder than with sequential consistency (but still feasible?, mechanically?)

$\Rightarrow$ static analysis parameterized by a WCM will be a challenge

The End, Thank You