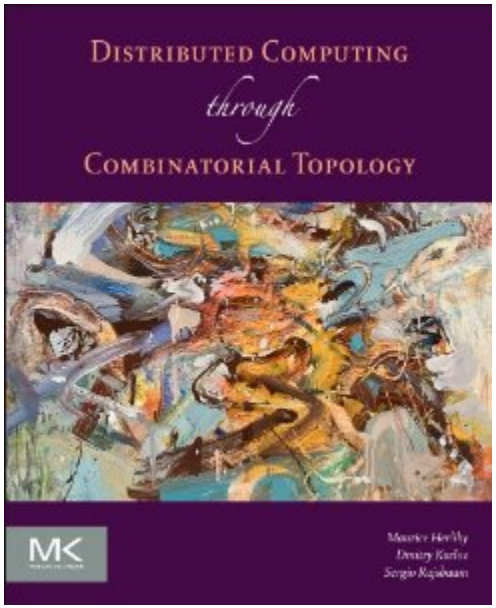
Simulations and Reductions



Companion slides for

Distributed Computing

Through Combinatorial Topology

Maurice Herlihy & Dmitry Kozlov & Sergio Rajsbaum

Reduction in Complexity Theory

SAT is NP-Complete

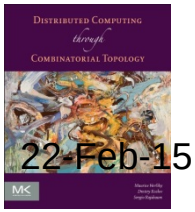
Hard to prove

CLIQUE reduces to *SAT*

Much easier to prove

Therefore *CLIQUE is* NP-Complete

Reduction is powerful



Reduction in Distributed Computing

Task *T* impossible for *n+1* asynchronous processes if any *t* can fail

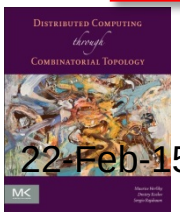
Easy to prove only if *n* = *t*

t+1 processes can "simulate" *n*+1 processes where any *t* can fail

[Borowsky Gafni]

Therefore task *T* impossible for *n+1* asynchronous processes if any *t* can fail

Reduction still powerful?

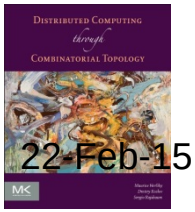


Observations

Reduction often easier than proof from first principles

Existence of reduction is important …

How reduction works? Not so much.



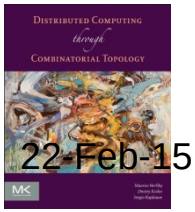
Limitations

Actual reductions often complex, ad-hoc model-specific arguments

Clever but complex

What does it mean for one model to "simulate" another?

Specific examples only



Goal

Define when one model of computation "simulates" another

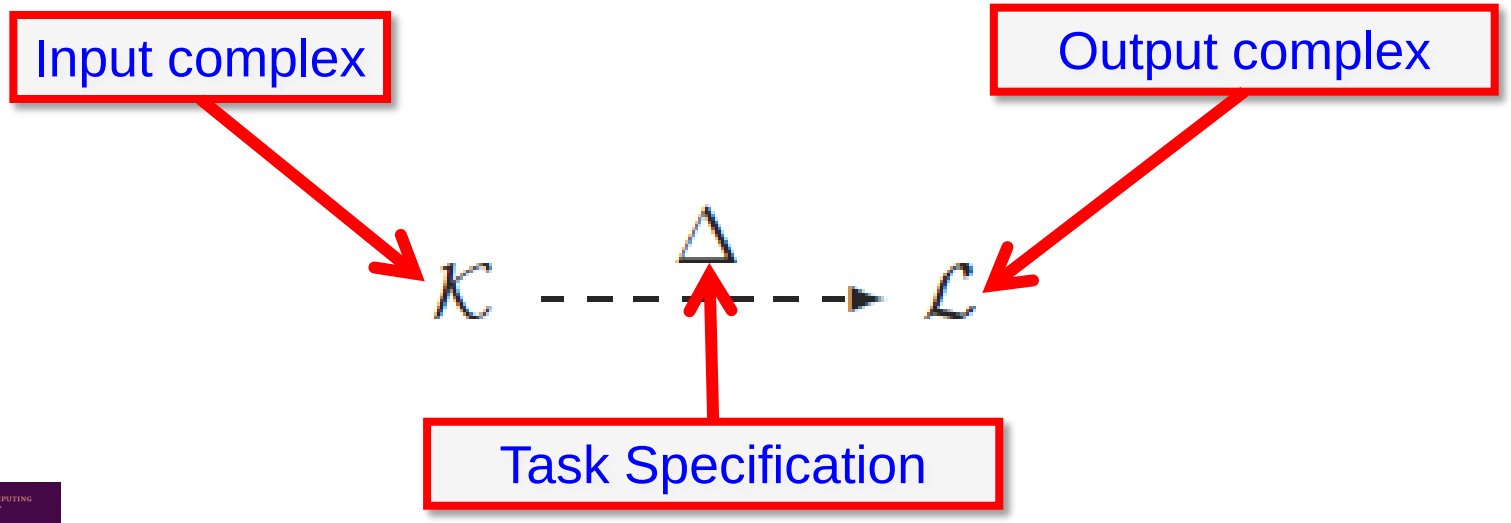
Covers many cases, not all.

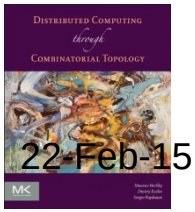
Technique to prove when a simulation exists

No need for explicit construction

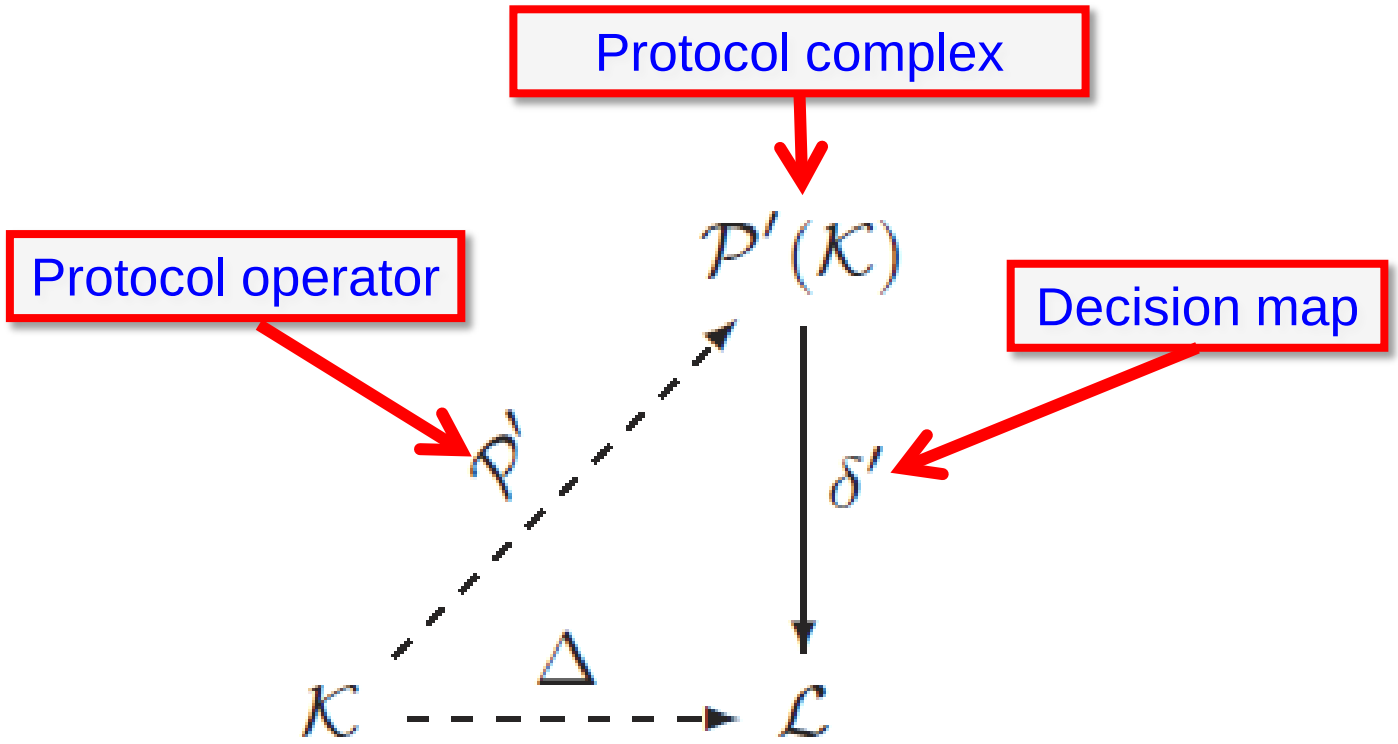
Strong enough to support reduction

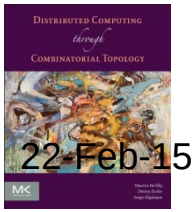
Task Specification



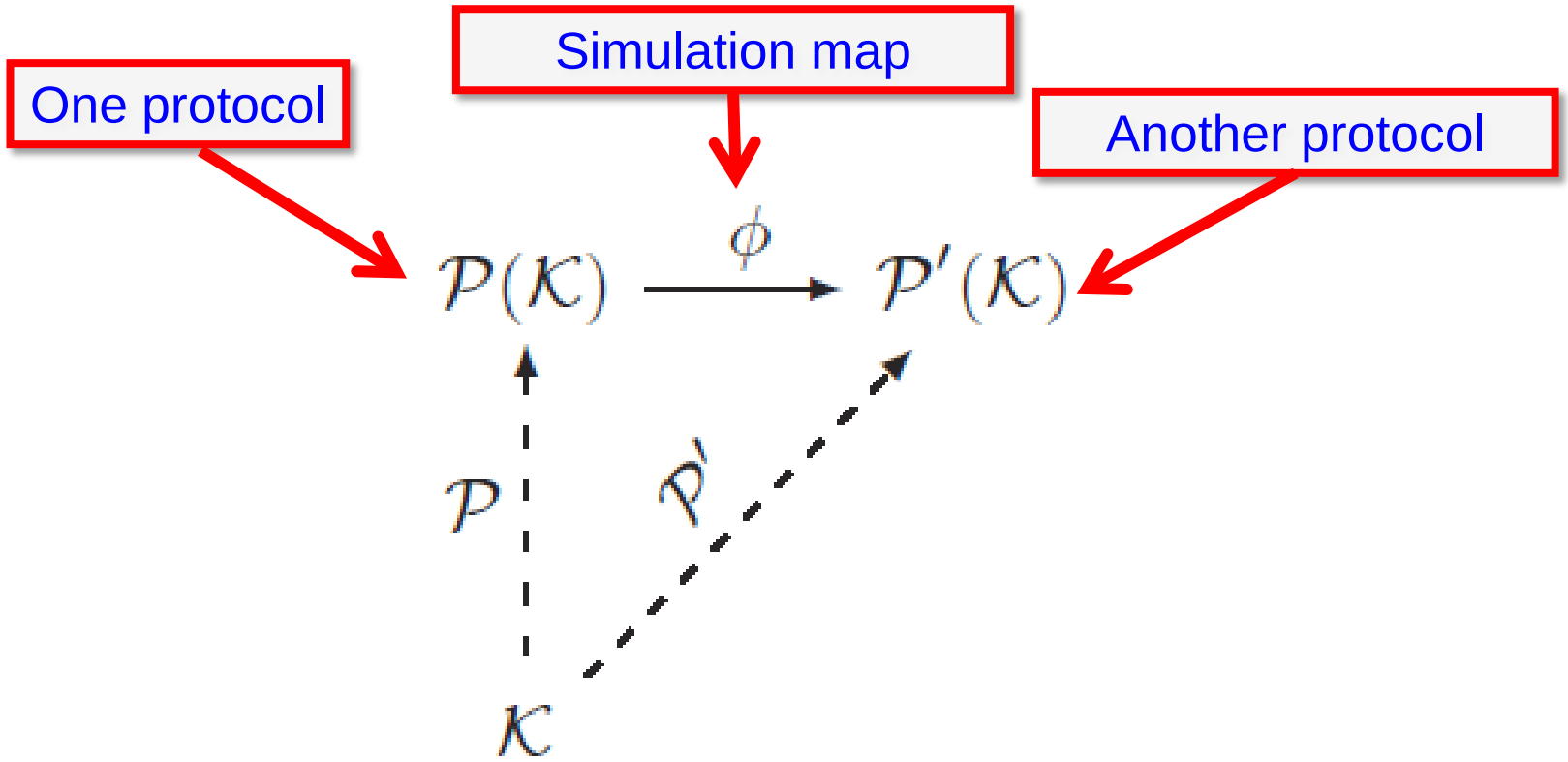


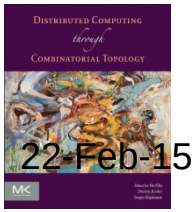
How a Protocol Solves a Task



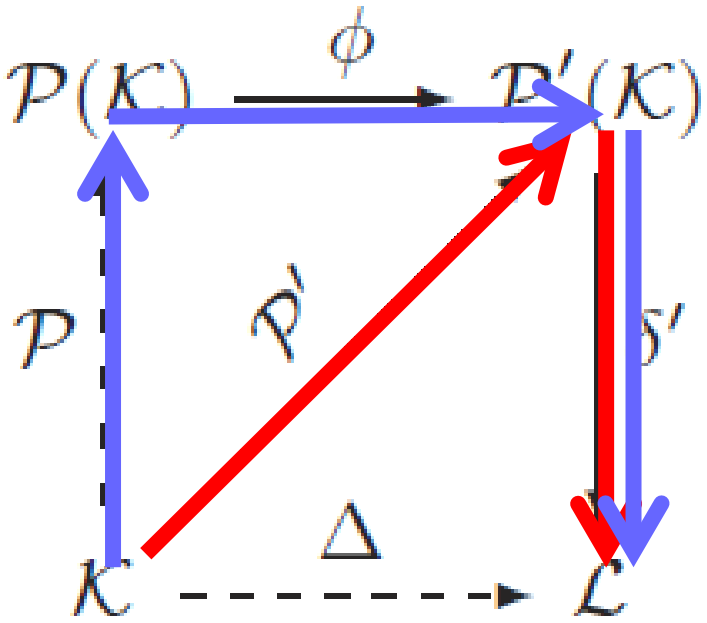


A Simulation

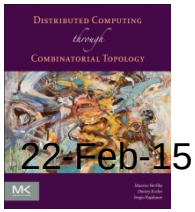




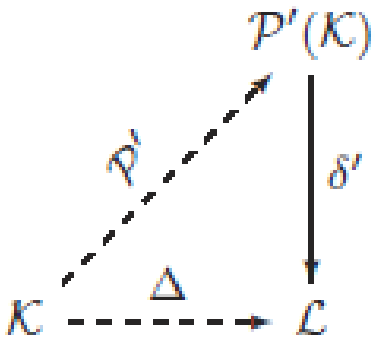
A Reduction



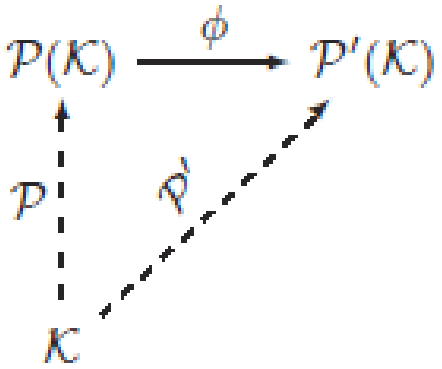
The Diagram *commutes*



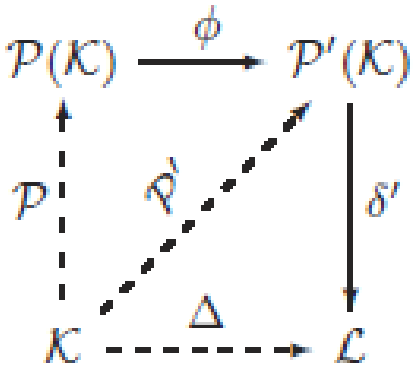
Summary



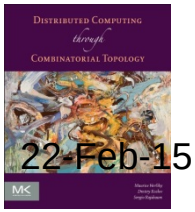
solves



simulates



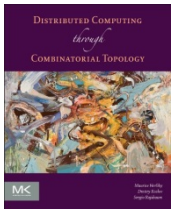
reduces



Strategy

Show that simulation maps *exist*

Construct simulation map *explicitly*



N&S Conditions

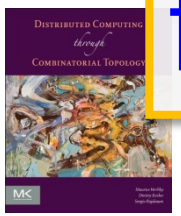
In each model …

$(\mathcal{I},\mathcal{O},\Delta)$ has a protocol iff …

$f: |\text{skel}^t\ \mathcal{I}| \rightarrow |\mathcal{O}|$

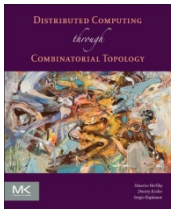
carried by Δ.

for model-specific t



Models that Solve the Same Colorless Tasks

processes	fault-tolerance	model
t+1	wait-free	layered IS
n+1	t-resilient	layered IS
n+1	wait-free	(t+1)-set layered IS
n+1	t-resilient, $2t<n+1$	message-passing
n+1	A-resilient, min core t+1	layered IS adversary
n+1	t-resilient $n+1 > (\dim \mathcal{I} +2)t$	Byzantine



Some Implications

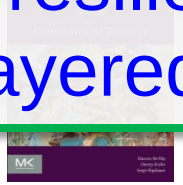
(t+1)-process wait-free can simulate an (n+1)-process wait-free, and vice-versa.

If $2t > n+1$, (n+1)-process t-resilient message-passing can simulate IS, and vice-versa.

Any adversary can simulate any other adversary whose minimum core size is the same or larger.

An adversary with minimum core size k can simulate a wait-free k-set layered IS.

t-resilient Byzantine can simulate t-resilient layered IS if $n + 1 > (\dim(\mathcal{I}) + 2)t$.



BG Simulation

Borowsky-Gafni

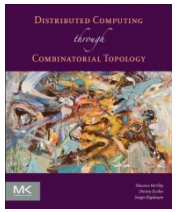
Explicit construction

n+1 processes, adversary $\mathbb{A}$

simulate

m+1 processes, adversary $\mathbb{A}$'

where $\mathbb{A}$, $\mathbb{A}$' have same min core size



Safe Agreement

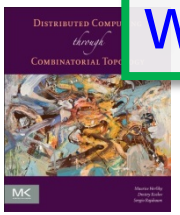
Validity

all processes that decide,
decide some process's input.

Agreement

all processes that decide,
decide the same value

we do *not* require termination!



Propose-Resolve

propose(v)

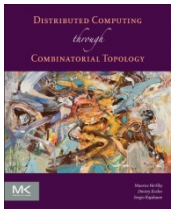
called once when joining protocol

resolve()

may be called multiple times

returns v if protocol resolved

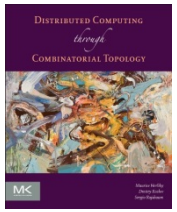
returns ⊥ if protocol still unresolved



Propose

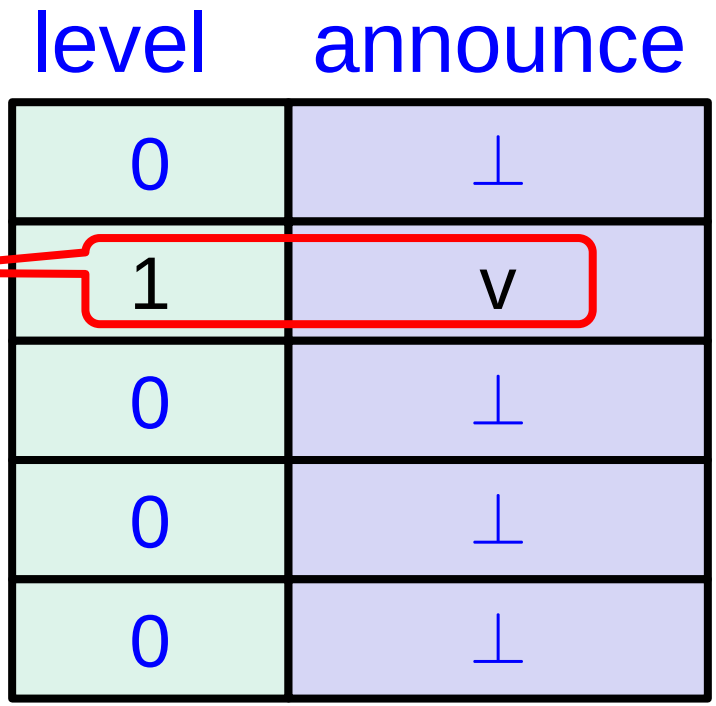
level	announce
0	⊥
0	⊥
0	⊥
0	⊥
0	⊥

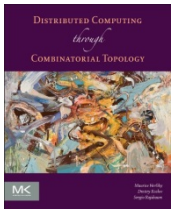
$n+1$



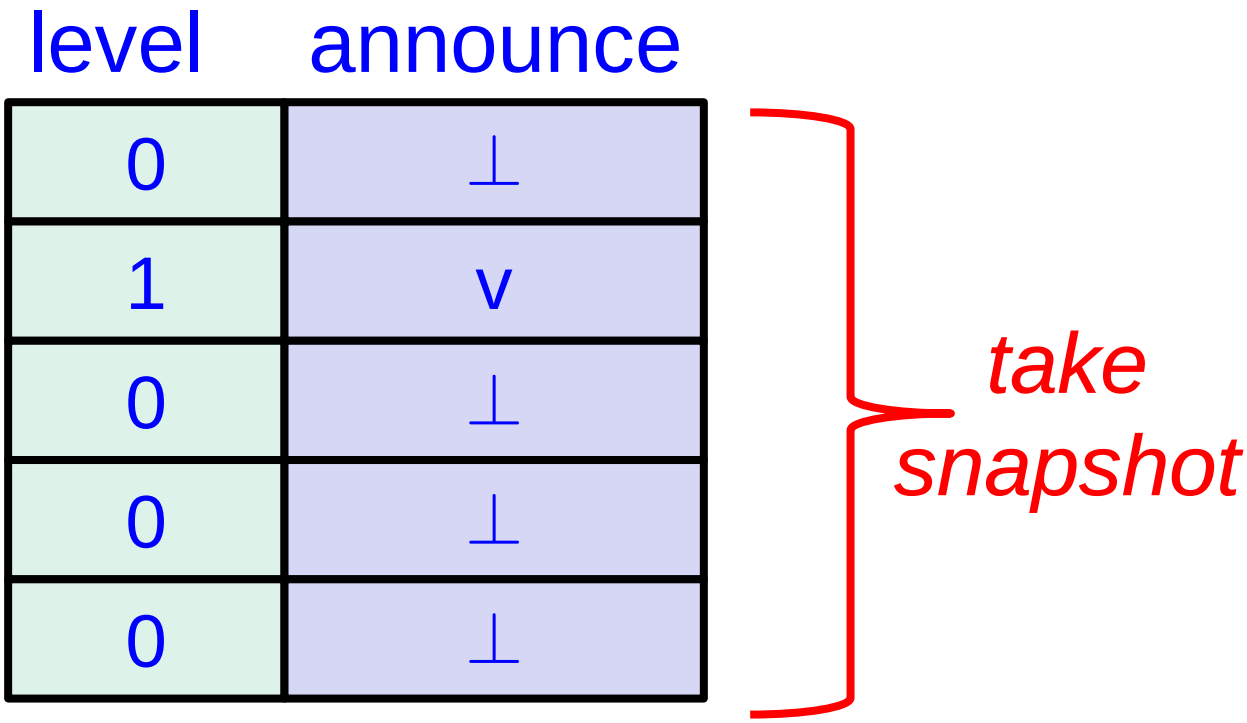
Propose: Unsafe Zone

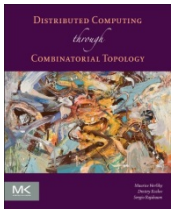
announce value with level 1



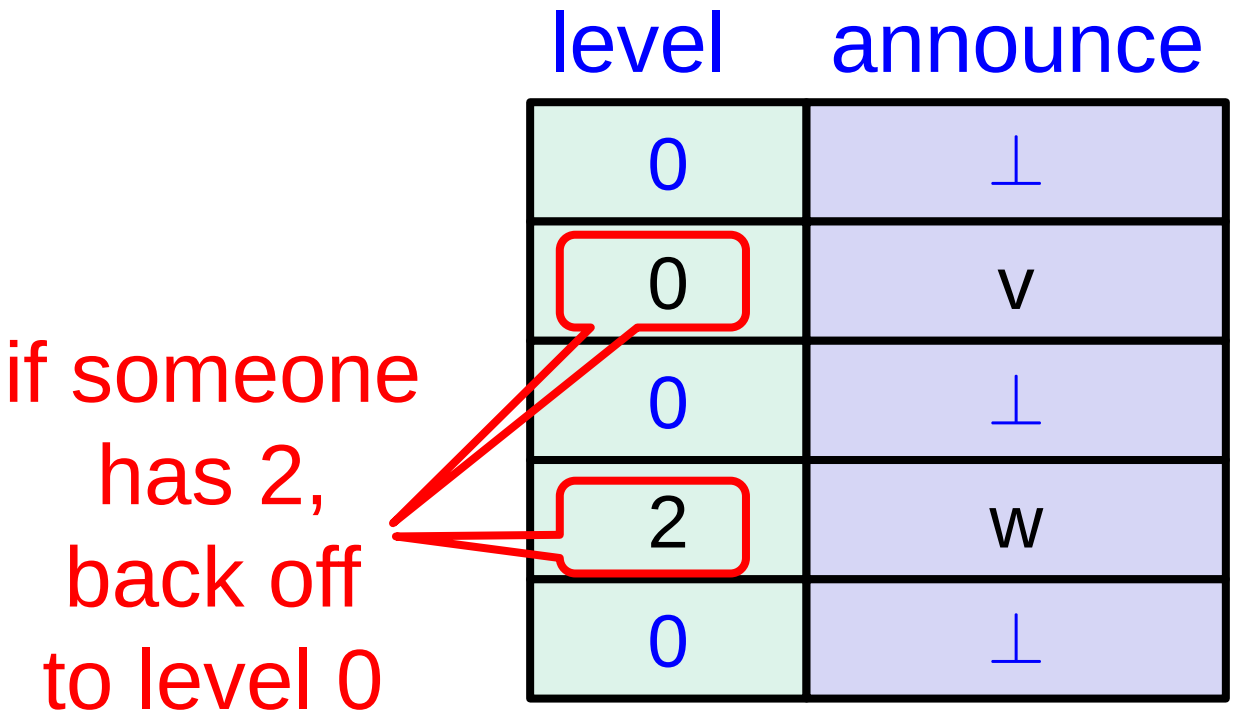


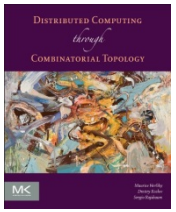
Propose: Unsafe Zone



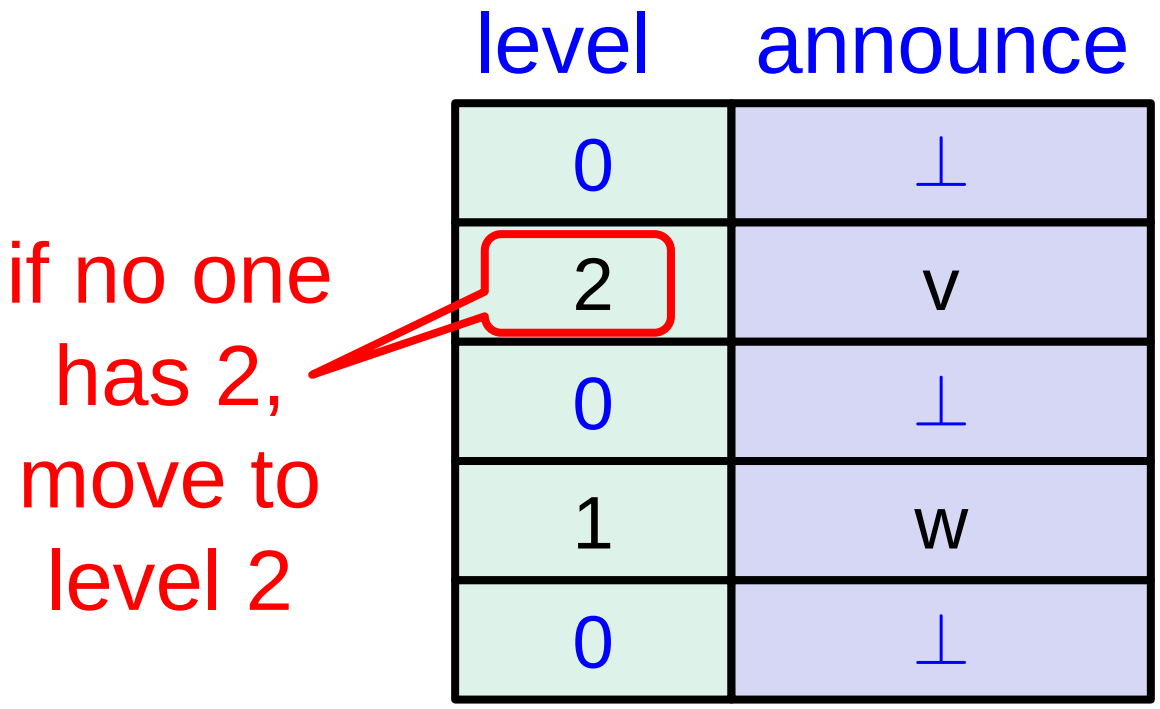


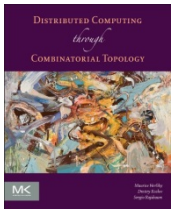
Propose: Safe Zone





Propose: Safe Zone

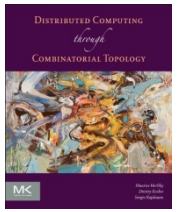




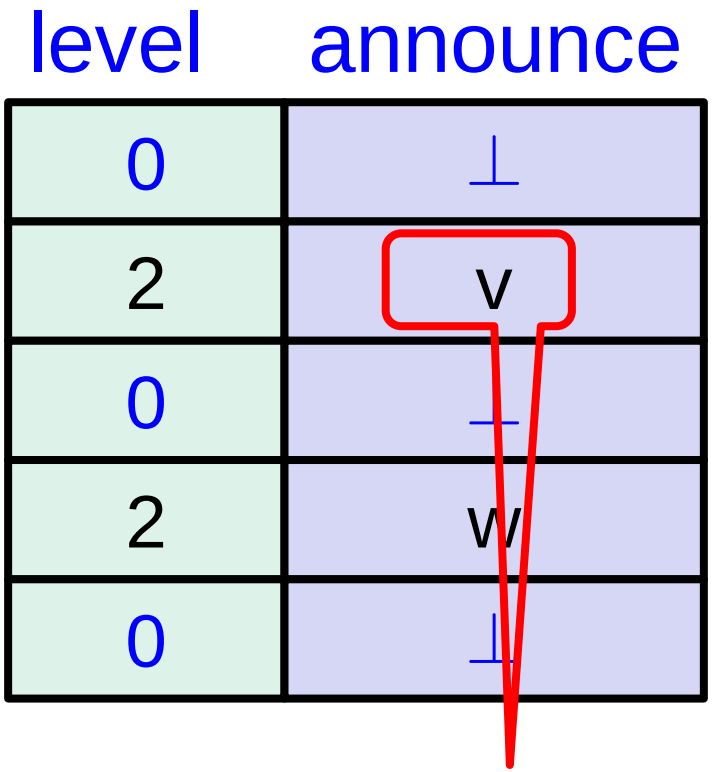
Resolve

level	announce
0	⊥
2	v
0	⊥
1	w
0	⊥

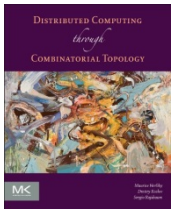
if anyone has 1, return ⊥



Resolve



return value at least index with 2

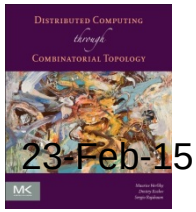


Propose

```
method propose(input: value)
 announce[i] := input
 level[i] := 1
 snap = snapshot(level)
 if (∃ j | level[j] = 2)
   then
     level[i] := 0
   else
     level[i] := 2
```

Resolve

```
method resolve(): value
   snap = snapshot(level)
   if (∃ j | level[j] = 1)
      then
         return ⊥
      else
         return announce[j]
            for min {j : level[j] = 2}
```



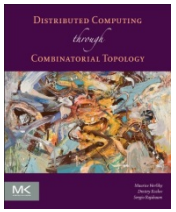
What it does

if no one halts in unsafe region (level 1) …

then all resolve same input

if someone halts in unsafe region …

never resolves



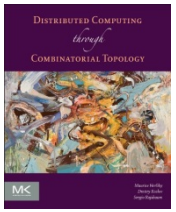
BG Simulation

There are t+1 processes …

who do a wait-free simulation of

a t-resilient (n+1)-process protocol

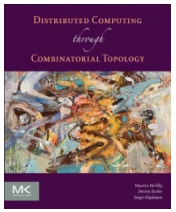
transforms between t-resilient and wait-free



BG Simulation

Use safe agreement …

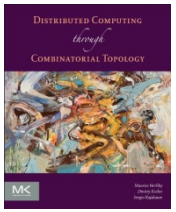
to agree on simulated snapshots



BG Simulation

Each simulating process participates in…

multiple *simultaneous* safe agreements

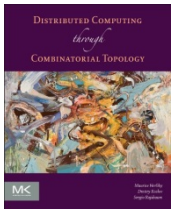


BG Simulation

If one process fails in unsafe region …

it blocks one simulated snapshot …

one simulated crash

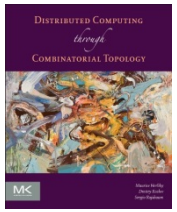


BG Simulation

If t out of t+1 halt in unsafe region …

simulates t out of n+1 failures …

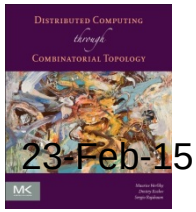
remaining process simulates n+1-t survivors



BG Simulation Code

```
shared mem: array[0..R][0..m] of value
shared agree: array[0..R][0..m] of SafeAgree

local pc: array[0..m] of int := {0,...,0}
```

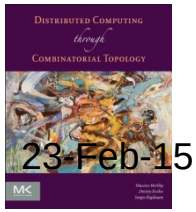


BG Simulation Code

```
shared mem: array[0..R][0..m] of value
shared agree: array[0..R][0..m] of SafeAgree

local pc: array[0..m] of int := {0,...,0}
```

shared simulated R × m memory

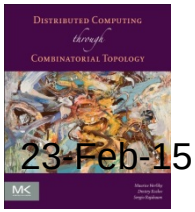


BG Simulation Code

```
shared mem: array[0..R][0..m] of value
shared agree: array[0..R][0..m] of SafeAgree

local pc: array[0..m] of int := {0,...,0}
```

shared safe agreement object
one per memory location

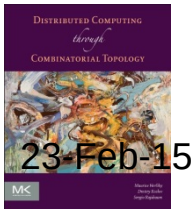


BG Simulation Code

```
shared mem: array[0..R][0..m] of value
shared agree: array[0..R][0..m] of SafeAgree

local pc: array[0..m] of int := {0,...,0}
```

program counters,
one per simulated process



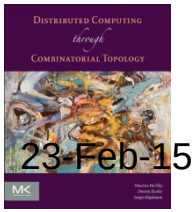
BG Simulation Code

```
method run(input: value): state
  for j := 0 to m do
      agree[0][j].propose(input)
```

input value $\rightarrow$ final state

set as many inputs as possible to mine

(OK because colorless tasks)



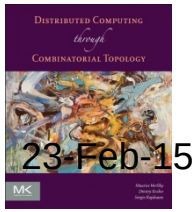
BG Simulation Code

```
do forever
  for j := 0 to m do
    r := pc[j]
    v := agree[r][j].resolve()
    …
```

simulate Q_j

program counter

agree on prior round's snapshot



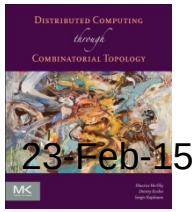
BG Simulation Code

```
do forever
  …
  if v ≠ ⊥ then
    mem[r][j] := v
    if pc[j] = R then
      return v
```

if snapshot resolved …

write snapshot to memory

if simulated state is final, return it



BG Simulation Code

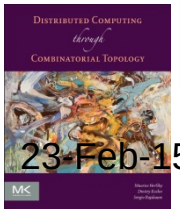
```
do forever
  …
  if survivor set present then
    view := values in snapshot(mem[r])
    agree[r+1][j].propose(view)
    pc[j] := pc[j] + 1
```

if survivor set reached this round…

take a snapshot

propose snapshot to write for next round

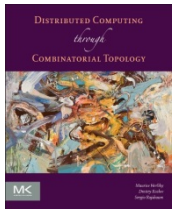
advance program counter



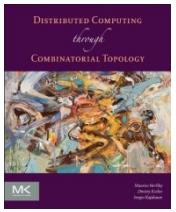
Two Styles of Colorless Simulation

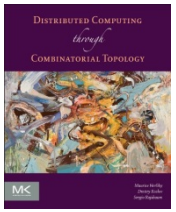
Combinatorial: simulation map exists

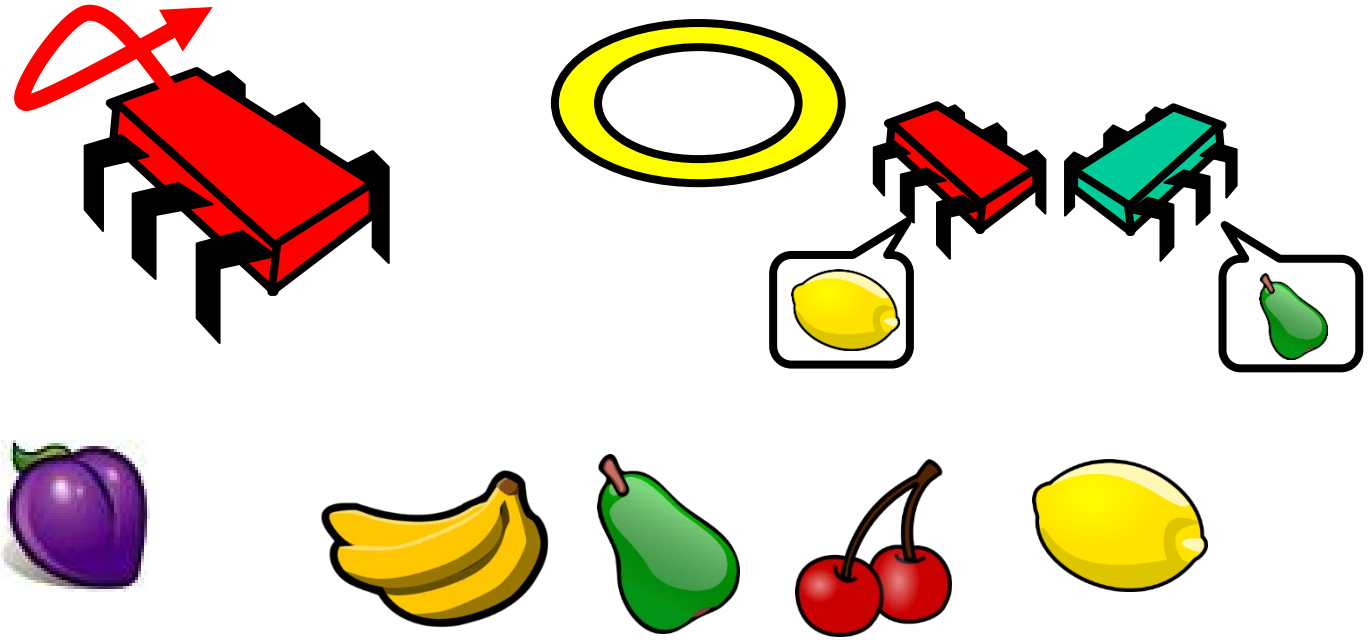
Operational: construct simulation explicitly

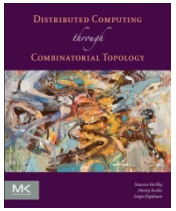


The Simulation








Distributed Computing
through
Combinatorial Topology