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

Input complex

Task Specification

Output complex
How a Protocol Solves a Task

Protocol operator

Protocol complex

Decision map
A Simulation

One protocol

Simulation map

Another protocol

$P(K) \xrightarrow{\phi} P'(K)$
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 ...

\[ f: |\text{ske}l^t\mathcal{I}| \rightarrow |\mathcal{O}| \]

carried by $\Delta$.

for model-specific $t$
# Models that Solve the Same Colorless Tasks

<table>
<thead>
<tr>
<th>processes</th>
<th>fault-tolerance</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t+1$</td>
<td>wait-free</td>
<td>layered IS</td>
</tr>
<tr>
<td>$n+1$</td>
<td>$t$-resilient</td>
<td>layered IS</td>
</tr>
<tr>
<td>$n+1$</td>
<td>wait-free</td>
<td>$(t+1)$-set layered IS</td>
</tr>
<tr>
<td>$n+1$</td>
<td>$t$-resilient, $2t&lt;n+1$</td>
<td>message-passing</td>
</tr>
<tr>
<td>$n+1$</td>
<td>$A$-resilient, min core $t+1$</td>
<td>layered IS adversary</td>
</tr>
<tr>
<td>$n+1$</td>
<td>$t$-resilient $n+1 &gt; (\dim \mathcal{I} +2)t$</td>
<td>Byzantine</td>
</tr>
</tbody>
</table>
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 $n + 1 > (\dim(I) + 2)t$. 
Explicit construction

$n+1$ processes, adversary $A$

simulate

$m+1$ processes, adversary $A'$

where $A$, $A'$ 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

\textbf{propose}(v)
- called once when joining protocol
- returns $v$ if protocol resolved
- returns $\perp$ if protocol still unresolved

\textbf{resolve}()
- may be called multiple times
- returns $v$ if protocol resolved
- returns $\perp$ if protocol still unresolved
Propose

<table>
<thead>
<tr>
<th>level</th>
<th>announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
</tbody>
</table>

$n+1$
Propose: Unsafe Zone

<table>
<thead>
<tr>
<th>level</th>
<th>announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\bot</td>
</tr>
<tr>
<td>1</td>
<td>v</td>
</tr>
<tr>
<td>0</td>
<td>\bot</td>
</tr>
<tr>
<td>0</td>
<td>\bot</td>
</tr>
<tr>
<td>0</td>
<td>\bot</td>
</tr>
</tbody>
</table>
Propose: Unsafe Zone

<table>
<thead>
<tr>
<th>level</th>
<th>announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
</tbody>
</table>

take snapshot
Propose: Safe Zone

<table>
<thead>
<tr>
<th>level</th>
<th>announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>0</td>
<td>$v$</td>
</tr>
<tr>
<td>0</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>2</td>
<td>$w$</td>
</tr>
<tr>
<td>0</td>
<td>$\downarrow$</td>
</tr>
</tbody>
</table>

if someone has 2, back off to level 0
Propose: Safe Zone

<table>
<thead>
<tr>
<th>level</th>
<th>announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>↓</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>0</td>
<td>↓</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
</tr>
<tr>
<td>0</td>
<td>↓</td>
</tr>
</tbody>
</table>

if no one has 2, move to level 2
Resolve

<table>
<thead>
<tr>
<th>level</th>
<th>announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
</tbody>
</table>

if anyone has 1, return ⊥
Resolve

<table>
<thead>
<tr>
<th>level</th>
<th>announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>w</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
</tr>
</tbody>
</table>

return value at least index with 2
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
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
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 $R \times m$ memory
shared mem: array[0..R][0..m] of value

**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
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
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)
do forever

for j := 0 to m do

r := pc[j]

v := agree[r][j].resolve()

...
BG Simulation Code

do forever
  ...
  if v ≠ \bot 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
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
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “Distributed Computing through Combinatorial Topology” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.