Composing Protocols

BCS, London

Farhad Arbab
Emeritus Professor

Center for Mathematics and Computer Science (CWI), Amsterdam
Leiden Institute of Advanced Computer Science, Leiden University

April 3, 2019

Memories... The way we were

Barbra Streisand

Mem'ries light the corners of my mind
Misty water-colored mem'ries of the way we were
Scattered pictures of the smiles we left behind
Smiles we gave to one another for the way we were
Can it be that it was all so simple then
Or has time rewritten every line
If we had the chance to do it all again,
tell me, would we, could we
Mem'ries may be beautiful and yet
What's too painful to remember
We simply choose to forget
So it's the laughter we will remember
Whenever we remember the way we were
The way we were

Nostalgia of simpler times!

- Apollo mission and moon landing.
- No mobile phone, no Internet, no PC!
- Mass media of print, radio, TV, and cinema bonded people via shared communal experiences.
- We seemed to have matured beyond weaponizing religion ever again.
- Trump as president was conceivable only in dystopian worlds of science fiction!
- Brexit mess was *inconceivable* even in dystopian worlds of science fiction!

The way we program(med)

Sequential

- Using progressively more abstract constructs
  - Machine code and assembly
  - Fortran, Cobol, Algol, PL/I, …
  - Lisp, APL: functional abstraction
  - Rigorous type systems
  - Abstract data types
  - Objects & classes
  - Prolog: logic programming
  - Haskell: monads and monoids
- Higher-level abstractions
  - Simplify expressing intention
  - Facilitate reasoning and proofs
  - Produce more efficient executables (than hand-crafted code)

Concurrent

- As tasks, processes, threads, etc., using primitives like
  - Locks & Mutex — prehistoric
  - Semaphores (Dijkstra) — 1962/1963
  - Monitors (Brinch Hansen & Hoare) — 1973/'74
  - CSP (Hoare) — 1978
  - π-calculus (Milner) — 1973-1980
  - Rendezvous (Ada) — 1980
  - ACP (Bergstra & Klop) — 1982
- Still use 40-50 year-old primitives!
- Lower-level abstractions
  - Complicate expressing intention
  - Hinder reasoning and proofs
  - Need top skills to get efficient executables (by hand-craft optimization)

Agenda

- Affirm that there exists a better way to conceive of and express concurrency protocols using language constructs in higher-levels of abstraction.
- Introduce a concrete programming language that offers such constructs.

Producers and Consumer

- Construct an application consisting of:
  - A Display consumer process
  - A Green producer process
  - A Red producer process
- The Display consumer must display the contents made available alternately by the Green and the Red producers.

Java-like Implementation

- Shared entities

```
private final Semaphore greenSemaphore = new Semaphore(1);
private final Semaphore redSemaphore = new Semaphore(0);
private final Semaphore bufferSemaphore = new Semaphore(1);
private String buffer = EMPTY;
```

- Consumer

```
while (true) {
  sleep (4000);
  bufferSemaphore.acquire();
  if (buffer != EMPTY) {
    println(buffer);
    buffer = EMPTY;
  }
  bufferSemaphore.release();
}
```

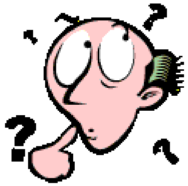
- Producers

```
while (true) {
  sleep (5000);
  greenText = ...
  greenSemaphore.acquire();
  bufferSemaphore.acquire();
  buffer = greenText;
  bufferSemaphore.release();
  redSemaphore.release();
}
```

```
while (true) {
  sleep (3000);
  redText = ...
  redSemaphore.acquire();
  bufferSemaphore.acquire();
  buffer = redText;
  bufferSemaphore.release();
  greenSemaphore.release();
}
```

- Where is green text computed?
- Where is red text computed?
- Where is text printed?
- Where is the protocol?

  - What determines who first?
  - What determines producers alternate?
  - What provides buffer protection?
  - Deadlocks?
  - Live-locks?
  - …



- Protocol becomes
  - Implicit, nebulous, and intangible
  - Difficult to reuse

Process Algebras

- Calculus to contrive expressions of action compositions.
  - Composition operators, e.g.: ., |, +, :=, implied recursion
- Abstract away the clutter of computation details.
- Enable reasoning through rules of an algebra.

Shared names:	g, r, b, d
Consumer:	B := ?b(t) . print(t) . !d("done") . B
Green producer:	G := ?g(k) . genG(t) . !b(t) . ?d(j) . !r(k) . G
Red producer:	R := ?r(k) . genR(t) . !b(t) . ?d(j) . !g(k) . R
Model:	G | R | B | !g("token")

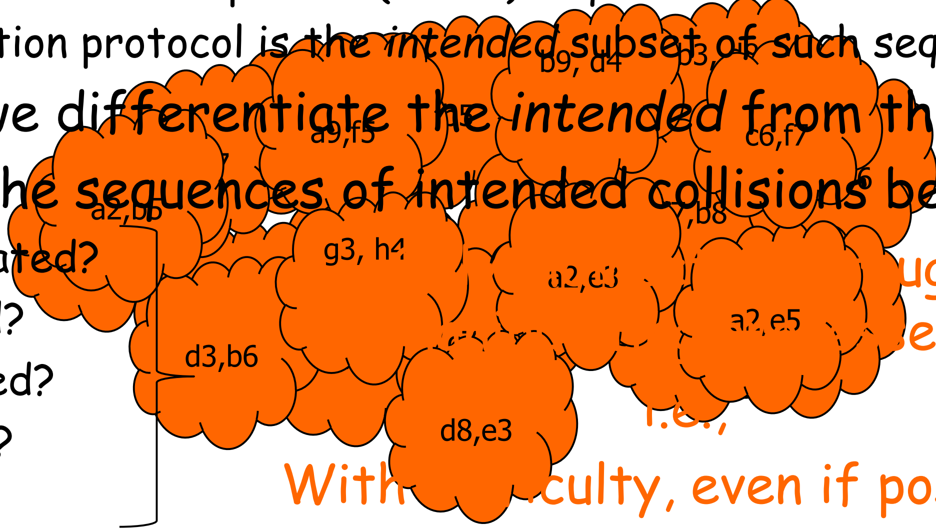
- Composition of actions yields more complex actions!
  - Hence the name "process algebra"!

Duh!

- Where is interaction?

Implicit Interaction

- Interaction (protocol) is implicit in action-based models of concurrency
- Interaction is a by-product of processes executing their actions
  - Action ai of process A collides with action bj of process B
  - Interaction is the specific (timed) sequence of such collisions in a run
  - Interaction protocol is the *intended* subset of such sequences.
- How can we differentiate the *intended* from the *coincidental*?
- How can the sequences of intended collisions be
  - Manipulated?
  - Verified?
  - Debugged?
  - Reused ?
  - ...



ugh

es,

i.e.,

With culty, even if possible!

Can it be that it was all so simple then?

- The interesting* side of concurrency is *interaction*, not action!
- An action is a mere "half-interaction" in a binary interaction.
- An action is an interaction-shard in a multiparty interaction.

Alternative to algebra of interaction-shards?

- Managing interaction becomes more difficult than necessary when done through its shards.
  - Tolerable with not too many shards (simple inte
    parties)
  - Unmanageable otherwise: increasingly the case
- Our failure to take interaction seriously as a first-class concept has made concurrent programming more complex than necessary.
- First-class concept:
  - Explicit construct to capture the concept
  - Composition operators, ideally, forming an algebra.
- Make action the implicit concept!

*As in: intriguing, exciting, challenging, exacting, difficult, arduous, grueling, herculean, laborious, curse!

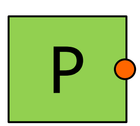
Concurrency by interaction

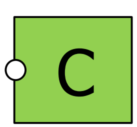
- A concurrent system consists of *actors* that *interact*.
  - An actor may itself contain nested interacting actors.
  - An atomic actor performs a sequential computation.
- Specification of a concurrent system:
  - What does each actor do?
    - Specification of computation.
  - What are the permissible interactions amongst actors?
    - Specification of *interaction protocol* as a constraint on ordering of activities and exchanges of partial results amongst independently running actors.

Interaction centric concurrency (1: actors)

- Specification of a concurrent system in terms of **actors** and their **interaction protocol**.
- **Actors** are black-box environment-agnostic processes:
  - Do not share memory
  - Contain no concurrency primitives (locks, semaphores, etc.)
  - Offer no inter-process methods nor make such calls
  - Do not send/receive targeted messages
  - Communicate exclusively by exchange of values through blocking I/O primitives that they perform only on their own ports:
    - get(p, v) or get(p, v, t)
    - put(p, v) or put(p, v, t)

```
while (true)
  sleep(3000);
  redText = ...;
  put(output, redText);
}
```






```
while (true) {
  sleep(4000);
  get(input, displayText);
  print(displayText);
}
```

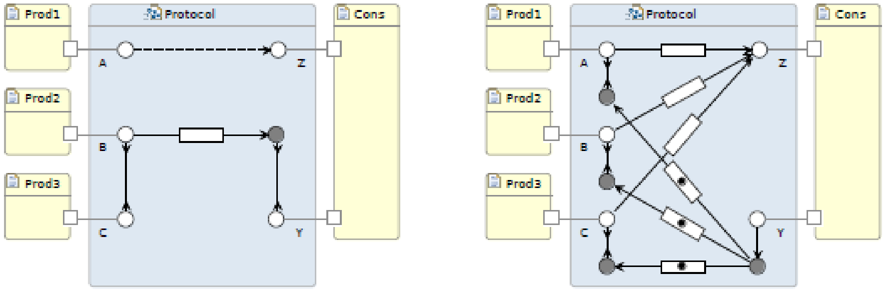
Interaction centric concurrency (2: protocols)

- **Interaction protocols** are connectors that *exogenously constrain* otherwise arbitrary interaction attempts by actors
- Composing same processes with different connectors yields different systems: **exogenous coordination**

- Compositional specification of interaction protocols:
  - Start with a set of primitive interactions as binary constraints
  - Define (constraint) composition operators to combine interactions into more complex interactions

- Farhad Arbab, Ivan Herman, and Per Spilling, "**Interaction Management of a Window Manager in Manifold**," Proceedings of the Fourth International Conference on Computing and Information, IEEE, Toronto, May 1992.
- Marcello Bonsangue, Farhad Arbab, Jaco de Bakker, Jan Rutten, Adriano Secutella, and Gianluigi Zavattaro, "**A Transition System Semantics for the Control-Driven Coordination Language Manifold**," *Theoretical Computer Science*, Elsevier, Vol. 240, No. 1, pp. 3-47, 2000.
- George A. Papadopoulos and Farhad Arbab, "Coordination Models and Languages," **Advances in Computers, Vol. 46**, Academic Press, 1998.

Reo



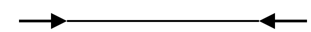
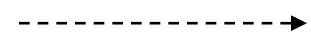
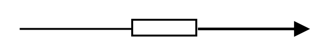
- Reo is a language for compositional construction of interaction protocols.
- Interaction is the only first-class concept in Reo:
  - Explicit constructs representing interaction
  - Composition operators over interaction constructs (set of interactions is closed under composition operators)
- Protocols manifest as a connectors
- In its graphical syntax, connectors are graphs
  - Data items flow through channels represented as edges
  - Boundary nodes permit (components to perform) I/O operations
- Formal semantics given as ABT (and various other formalisms)
- Tool support: draw, animate, verify, compile

- F. Arbab "**Puff, The Magic Protocol**," Formal Modeling: Actors, Open Systems, Biological Systems 2011, SRI International, Menlo Park, California, November 3-4, 2011, Lecture Notes in Computer Science, Springer, vol. 7000, pp. 169-206, 2011.
- Farhad Arbab, "**Reo: A Channel-based Coordination Model for Component Composition**," *Mathematical Structures in Computer Science*, Cambridge University Press, Vol. 14, Issue 3, pp. 329-366, June 2004.

Channels

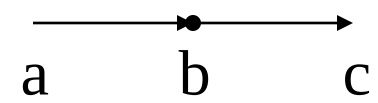
- Atomic connectors in Reo are called *channels*.
- Reo generalizes the common notion of channel.
- A channel is an abstract communication medium with:
  - exactly two ends; and
  - a constraint that relates (the flows of data at) its ends.
- Two types of channel ends
  - Source: data enters into the channel.
  - Sink: data leaves the channel.
- A channel can have two sources or two sinks.
- A channel represents a primitive interaction.

A Sample of Channels

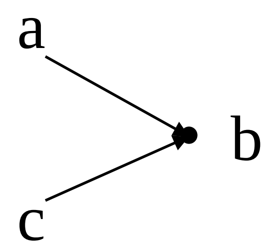
- Synchronous channel 
  - write/take
- Synchronous drain: two sources 
  - write/write
- Synchronous spout: two sinks 
  - take/take
- Lossy synchronous channel 
- Asynchronous FIFO1 channel 
  - write/take

Join

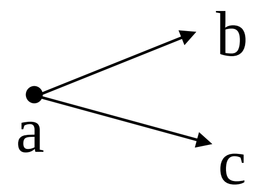
- Mixed node
  - Atomic merge + replication



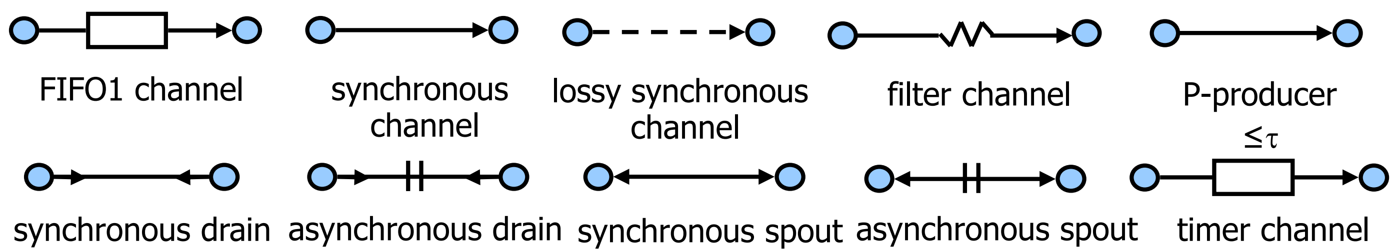
- Sink node
  - Non-deterministic merge

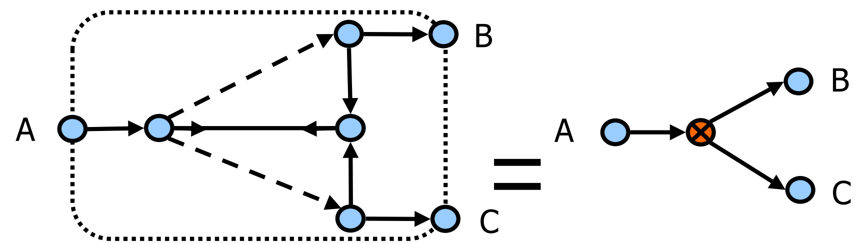


- Source node
  - Atomic replication

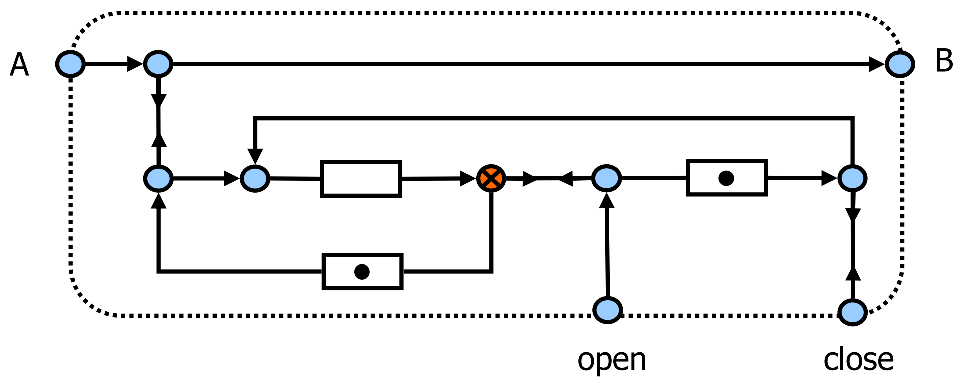


Reo Connectors





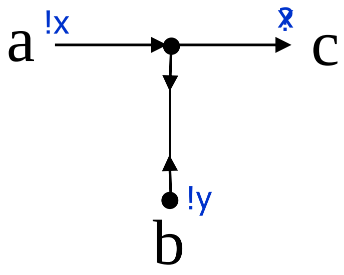
Exclusive choice (deffered XOR)



Valve connector:
controls flow from A to B

Flow regulator

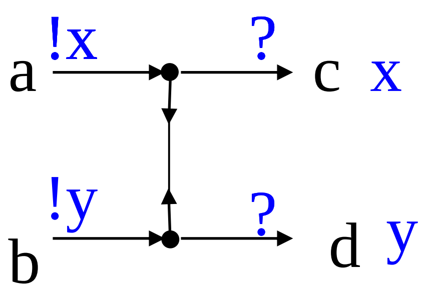
- Write-cue synchronous flow-regulator




```
regulatorwwr(a, b, c) {
  sync(a, m) syncdrain(m, b) sync(m, c)
}
```

Flow Synchronization

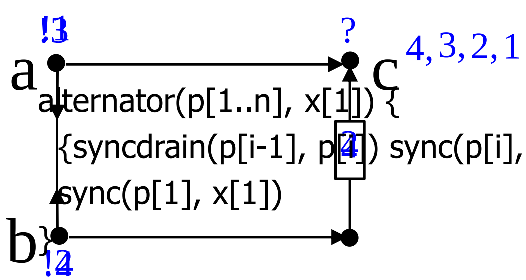
- The write/take operations on the pairs of channel ends a/c and b/d are synchronized.
- Barrier synchronization.




```
barrier(x[1..n], y[1..n]) {
  {sync(x[i], z[i]) sync(z[i], y[i]) | i:<1..n>}
}
```

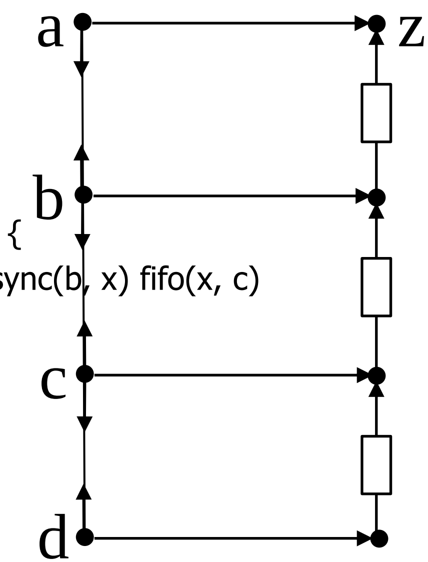
Alternator

- Subsequent takes from c retrieve items from the streams alternating between a and b.
- Items at both a and b must be present in each round before a pair can go through.
- Generalize to n inputs:




```
alternator(p[1..n], x[1]) {
  {syncdrain(p[i-1], p[i]) sync(p[i], x[i]) fifo(x[i], x[i-1]) | i:<2 .. N>}
  sync(p[1], x[1])
}
```

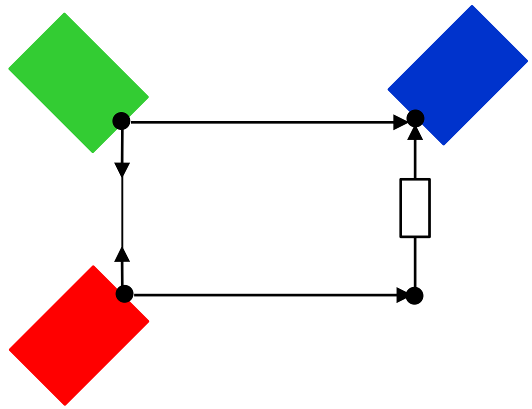
```
alternator(a, b, c) {
  syncdrain(a, b) sync(b, x) fifo(x, c)
  sync(a, c)
}
```



Alternating Producers

- We can use the alternator circuit to impose the protocol on the green and red producers of our example
  - From outside
  - Without their knowledge

```
while (true) {
  sleep(5000);
  greenText = ...;
  put(output, greenText);
}
```

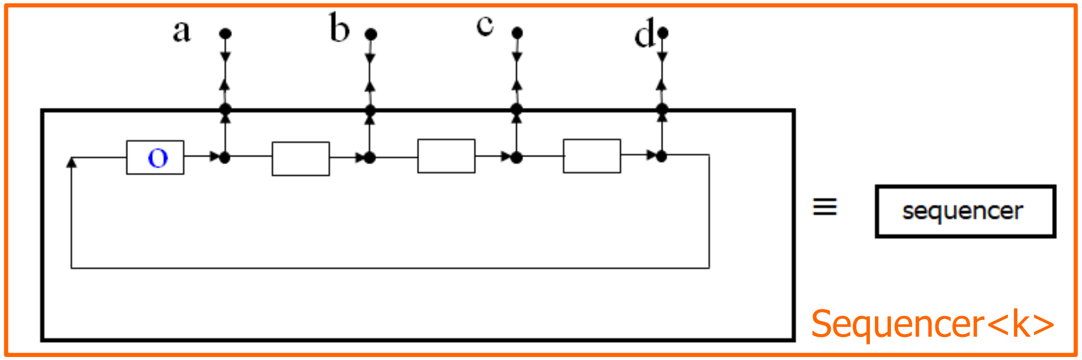


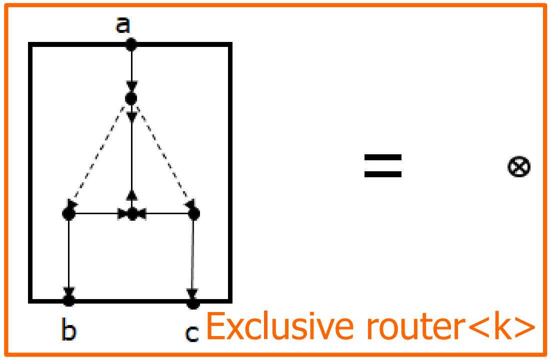
```
while (true) {
  sleep(4000);
  get(input, displayText);
  print(displayText);
}
```

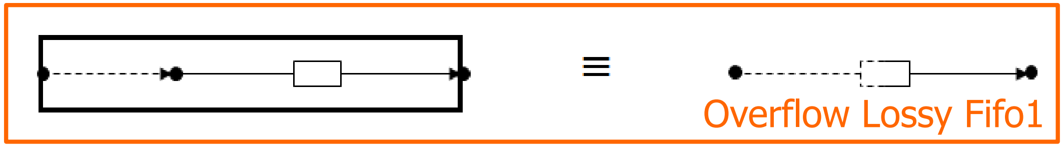
```
while (true)
  sleep(3000);
  redText = ...;
  put(output, redText);
}
```

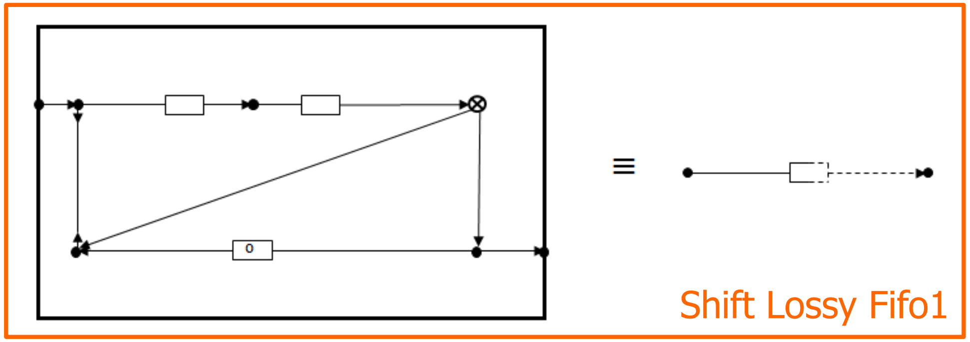
```
main() {
  green(a) red(b) blue(c) alternator(a, b, c)
}
```

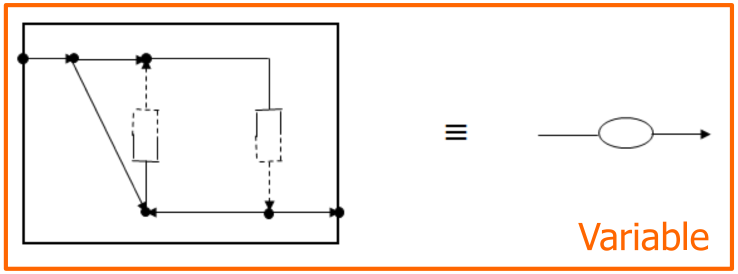
Library


a
b
c
d
o
≡
sequencer
Sequencer<k>


a
b
c
=
⊗
Exclusive router<k>

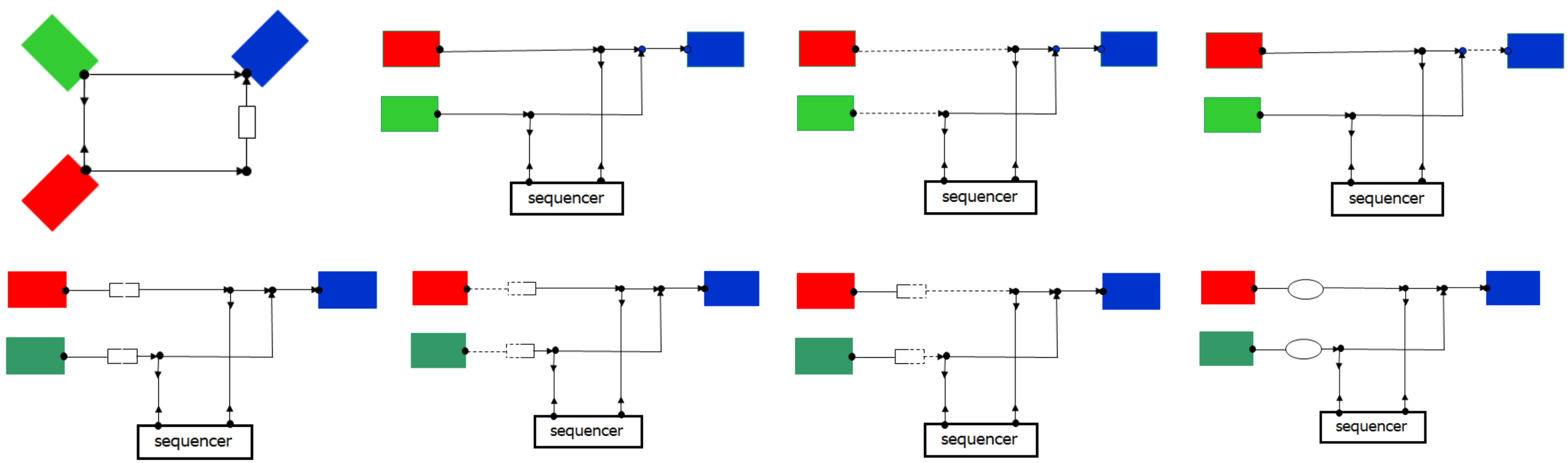

≡
Overflow Lossy Fifo1


0
≡
Shift Lossy Fifo1

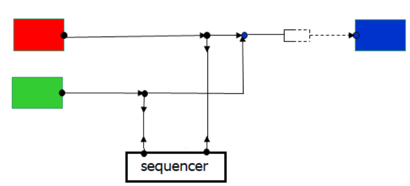

≡
Variable

Possible intended alternatives

- At least 8 different sensible protocol alternatives:



- None of which has the bug/feature of the Java code!

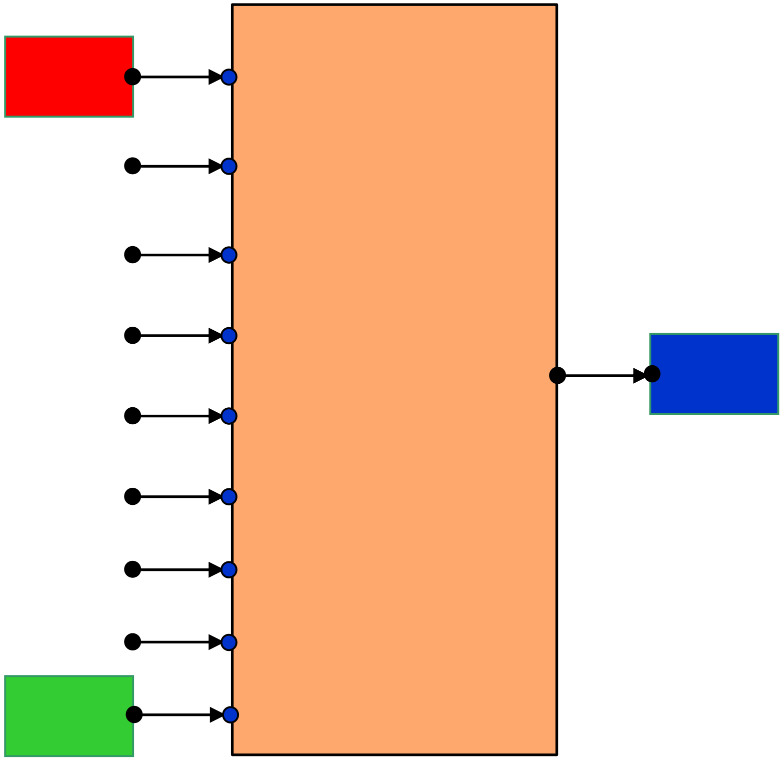


Scaling up

- Scale up?

```
main() {
  green(a[1]) … red(a[n]) blue(b)
  connector(a[1..n], b)
}

connector(a[1..n], b) {
  seqc(x[n])
  {sync(a[i], x[i]) sync(x[i], m) | i : <1 ..n>}
  sync(m, b)
}
```

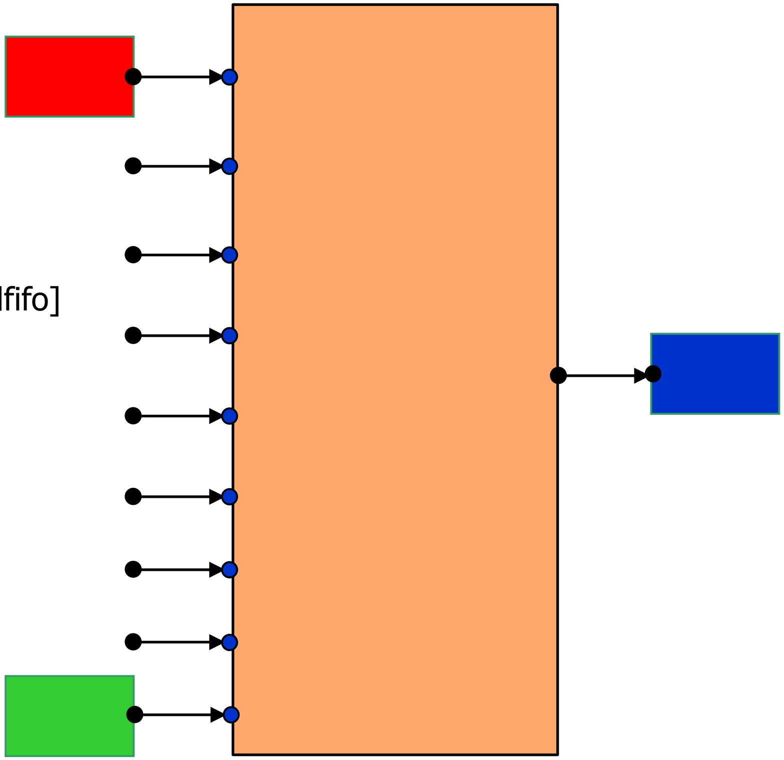


Scale and combine

- Mix and match?

```
main() {
  green(a[1]) … red(a[n]) blue(b)
  ileg = [sync, lossysync, fifo, sync, variable, …, shiftlossyfifo, ovflfifo]
  connector<ileg[1..n], sync>(a[1..n], b)
}

Connector<ileg[1..n](?, !), oleg(?, !)> (a[1..n], b) {
  seqc(x[n])
  {ileg[i](a[i], x[i]) sync(x[i], m) | for i : <1 ..n>}
  oleg(m, b)
}
```

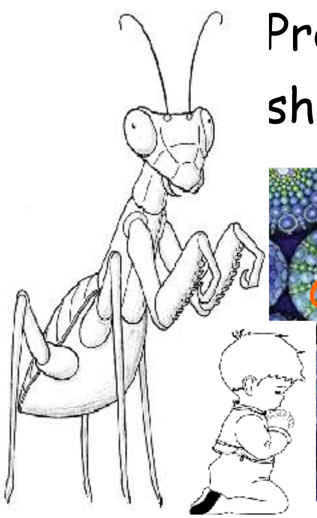
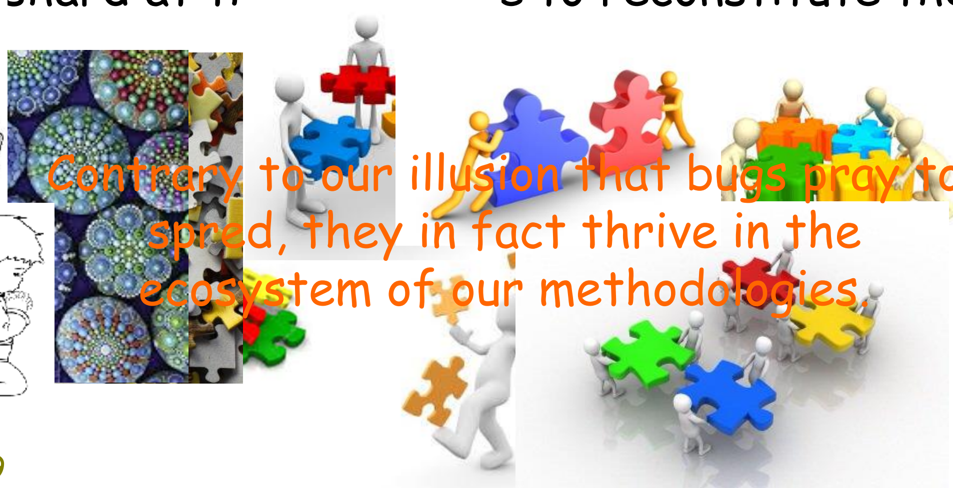


Protocol programming example

- A 1-out-of-n protocol:
  - Output 1 item out of items from n input ports & repeat.

Protocol programming

- Action-centric programming of an interaction:
  - Smash interaction on the solid rock of **action**
  - Let each process/thread pick up some interaction-shards.
  - Pray that:
    - No shards go missing or get lost
    - Processes will independently pick up and flip over just the right shard at the right time to reconstitute the original interaction.


Contrary to our illusion that bugs pray to spred, they in fact thrive in the ecosystem of our methodologies.

Protocol programming

- Interaction-centric programming of an interaction:
  - Separation of concerns.
    - Nature: ultimate machinery employing separation of concerns.

Nature manifests magnificently complex forms and behaviors by bridling simple unintelligent actions of independent actors, ignorant of those emerging patterns, with superimposition of easy constraints.

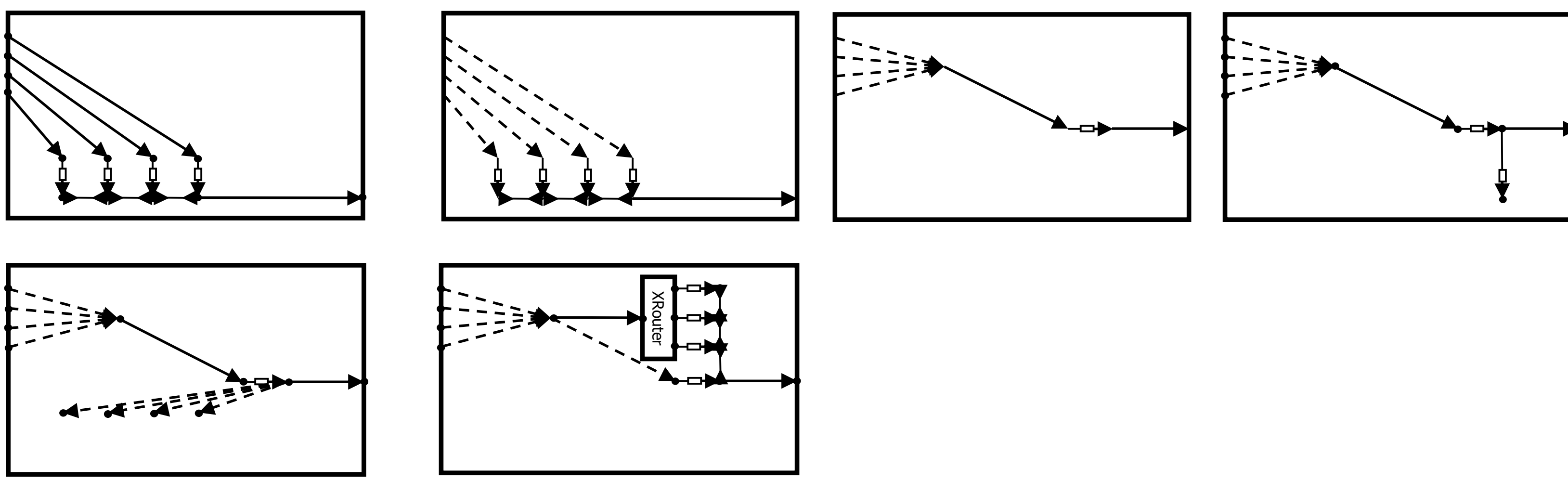
  - Consider the protocol as manifestation of a constraint.
  - Decompose the constraint into simple, down to easy constraints.
  - Superimpose constraints through mathematical composition of relations.

A 1-out-of-n protocol

- Output 1 item out of items from n input ports & repeat.
  - Which item?
    - Any one?
    - The first/last arriving one?
    - A specific one? Which one? In temporal order? In structural order?
  - How to handle excess input from the same source in a cycle?
    - Delay it for next cycle?
    - Lose it?
  - When should output become available?
    - As soon as available?
    - At the end of a cycle?
  - When does a cycle end?
    - After one input from each source?
    - Once the output is taken?
- Generalize 1-out-of-n to k-out-of-n

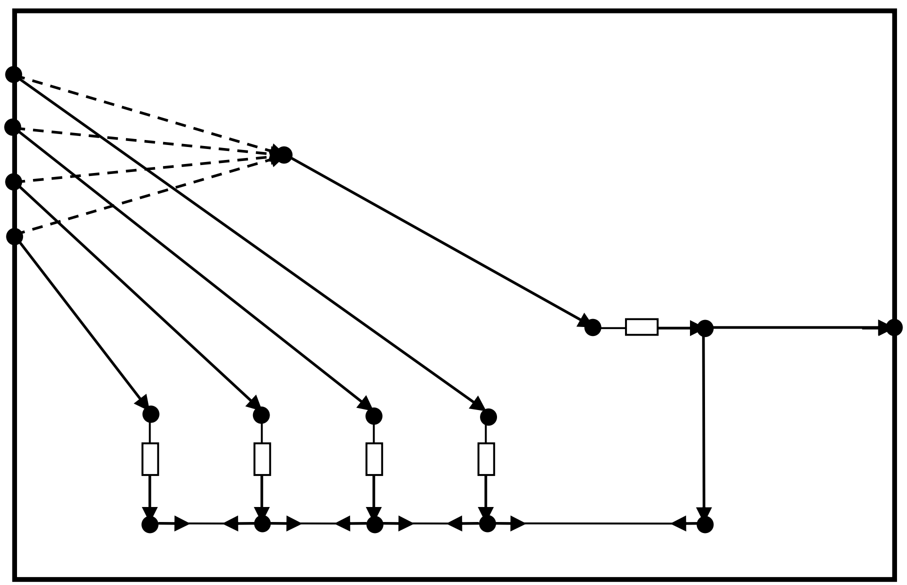
Interaction programming

- Decompose a protocol into simpler protocols.
- Compose the original protocol by superimposition of simpler protocols.
- Some simple sub-protocols for k-out-of-n:



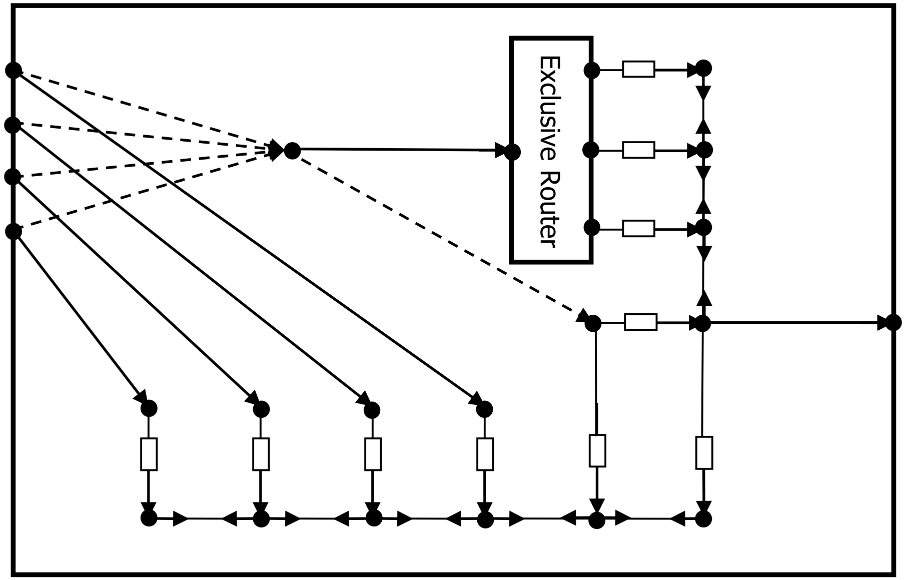
Sparing Delayed 1st Out of *n*

- Outputs only the first of the *n* arriving inputs in each cycle.
- Output is delayed until the end of each cycle.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are spared for the next cycle.



Sparing Prompt *k* Out of *n*

- Outputs only the first of the *n* arriving inputs in each cycle.
- Output is possible promptly after the first k input values arrive.
- Cycle ends after:
  - A value arrives on each input node, and
  - A value is taken from the output node.
- Extra input values of a node are spared for the next cycle.



Semantics

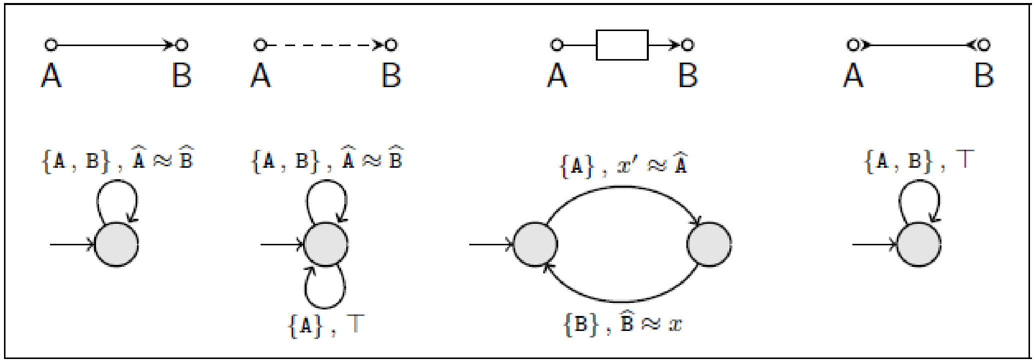
- Reo allows:
  - Arbitrary user-defined channels as primitives.
  - Arbitrary mix of synchrony and asynchrony.
  - Relational constraints between input and output.
- Reo is more expressive than, e.g., dataflow models, Kahn networks, workflow models, stream processing models, Petri nets, and synchronous languages.
- Formal semantics:
  - Coalgebraic semantics based on timed-data streams.
  - Constraint automata.
  - SOS semantics (in Maude).
  - Constraint propagation (connector coloring scheme).
  - First order predicate logic, Intuitionistic linear logic

- Sung-Shik T.Q. Jongmans and Farhad Arbab, "**Overview of Thirty Semantic Formalisms for Reo**," *Scientific Annals of Computer Science*, vol. 12, Issue 1, pp. 201-251, 2012.

Constraint automata

- Finite-state automata where a transition has a pair of constraints as its label:
  - (Synchronization-constraint, Data-constraint)
- Introduced to capture operational semantics of Reo

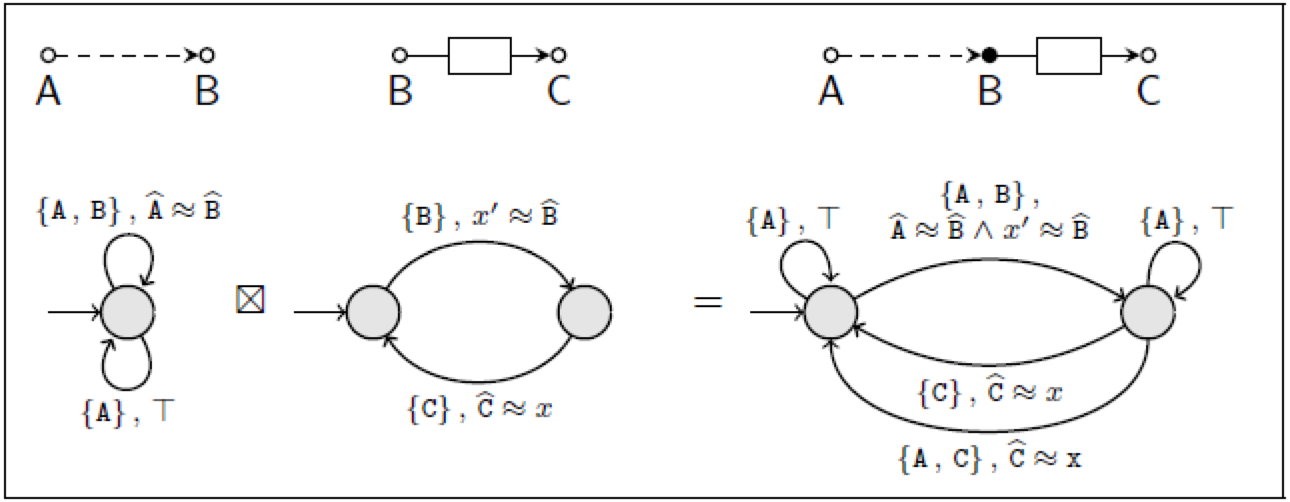
CA of typical Reo primitives:



- F. Arbab, C. Baier, J.J.M.M. Rutten, and M. Sirjani, "**Modeling Component Connectors in Reo by Constraint Automata**," Proc. International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003), CONCUR 2003, Marseille, France, September 2003, Electronic Notes in Theoretical Computer Science, 97.22, Elsevier Science, July 2004.
- C. Baier, M. Sirjani, F. Arbab, and J.J.M.M. Rutten, "**Modeling Component Connectors in Reo by Constraint Automata**," Science of Computer Programming, Elsevier, Vol. 61, Issue 2, pp. 75-113, July 2006.
- F. Arbab, C. Baier, F.S. de Boer, and J.J.M.M. Rutten, "**Models and Temporal Logical Specifications for Timed Component Connectors**," International Journal on Software and Systems Modeling, pp. 59-82, Vol. 6, No. 1, March 2007, Springer.

CA of a connector

- The *CA* semantics of a connector is composed from the *CA* of its constituents via a synchronous product operator.



Vereofy Model Checker

- Symbolic model checker for Reo:
  - Based on constraint automata
  - Developed at the University of Dresden
  - LTL and CTL-like logic for property specification
- Modal formulae
  - Branching time temporal logic:
    - AG[EX[*true*]]
    - check for deadlocks
  - Linear temporal logics:
    - **G**(*request* → **F** (*reject* ∪ *sendFormOut*))
    - check that admissible states *reject* or *sendFormOut* are reached
- http://www.vereofy.de

Executable code generation

- Reo makes interaction explicit and tangible, allowing
  - Specification
  - Composition
  - Analysis
  - Verification
  - Reuse

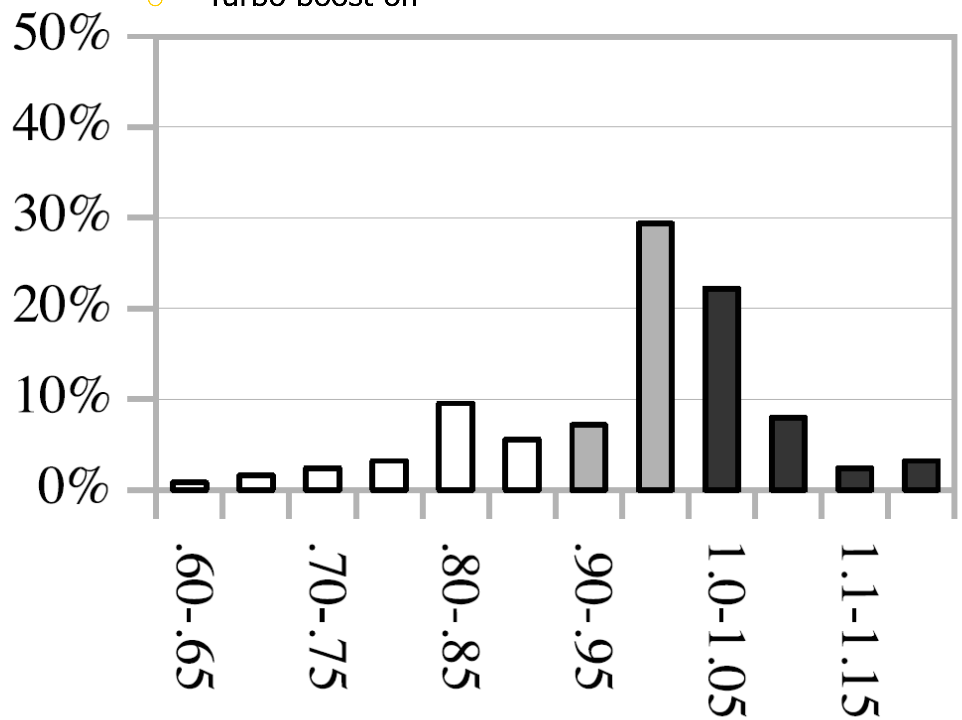
  Of interaction protocols
- Efficient executable code directly from Reo models?
  - Performance comparable to hand-crafted optimized code.
  - Choreography of Web services
  - Coordinated composition of distributed components
  - Concurrent applications on multi-core platforms
- Use Constraint Automata

NASA benchmarks

- Java version of NASA Parallel Benchmarks (NPB)
  - 84 full programs
  - Reo circuits reused for same protocols in different cases
  - Each case ran 5 times
- In 37% of cases generated code no worse than 10% slower
- In 38% of cases generated code is up to 20% faster
- In 25% of cases generated code is between 10% to 40% slower

Optimization opportunities!

- 24-cores, 2 Intel E5-2690V3 processors in 2 sockets
- Static clock frequency
  - Hyper-threading off
  - Turbo boost off



- Sung-Shik T.Q. Jongmans "**Automata-theoretic protocol programming**," PhD thesis, Leiden University, 2016, http://hdl.handle.net/1887/38223.

What are you doing the rest of your life?

- As the exponentially complex aspect of concurrency, interaction protocols become simpler to construct, validate, compose, and reuse as first-class entities.
- Interaction-centric programming needs programming constructs for:
  - Explicit formal representation
  - Direct composition

- Reo is a simple, rich, versatile, and surprisingly expressive language for compositional construction of pure interaction protocols.
  - Treats interaction as (the only) first-class concept.
  - Free combination of synchrony, exclusion, and asynchrony.
  - Offers direct composition and verbatim reuse of protocols.

http://reo.project.cwi.nl