

Coordination programming for self-tuning: the challenge of a heterogeneous open environment

Alex Shafarenko
Compiler Technology and Computer
Architecture Group
University of Hertfordshire

A coordination language

- Concept of coordination
 - orchestration of de-contextualised components
 - no knowledge of the overall system, or any other component
 - algorithm dependent solely on self-contained inputs (seen as messages), no external refs!
 - output is also self-contained, “messages” without a destination
 - a coordination program that:
 - coordinates **progress**: “the what” & “the when”
 - coordinates **interfaces**: so that components understand each other (including polymorphism, inheritance, etc.)
 - coordinates **access to shared objects**.

AstraKahn: coordination by streaming

- Some previous exposure: Functional-Reactive, FBP, Linda, etc.
- AstraKahn is a successor (or an off-shoot) of S-Net (see *snet-home.org*)
- S-Net fully implemented, well researched.
- Industrial evaluation:
 - THALES, SAP, Philips.
- 2 EU Framework projects
 - AETHER, ADVANCE; influenced PARAPHRASE
- S-Net has a similar concept of connectivity, but no fixed implementation model.

Means of Compositionality

- Kahn's Streaming
 - components do not interact by sharing data
 - FIFO communication with a (generally) static topology
 - each message a unit of work
 - messages can be used to trigger component actions
 - no explicit memory model (i.e. all memory partitioned into private memories of components)
 - [John Leidel's/Micron talk]
 - nevertheless, “communication” is a metaphor:
 - what gets passed around is “access” to data, which may or may not require real communication
 - hence a shared-memory and a mixed-memory interpretation are possible


Components + streaming not knew

- Kahn in the '70s: Kahn Process Networks (KPN)
 - components are deterministic sequential processes
 - fixed-topology input and output FIFOs
 - blocking reads, nonblocking writes = infinite resources
 - no alternation, one read at a time
 - theoretical proofs of soundness for semantics
 - a nontrivial result, since possibly cyclic topology
 - there exists a functional interpretation of KPN : a graph
 - vertices are stream monotonic functions
 - edges are stream variables
 - a system of eqs, guaranteed to have a solution

Purpose of **Astra**Kahn

- “Modernise” KPNs
 - infinite FIFOs don’t exist (limited resources)
 - (sequential) vertices have algebraic properties w.r.t input streams, ergo potential concurrency
 - add useful nondeterminism for real-time/embedded systems, e.g. DFRs, but as a general mechanism and under control
- Structure KPNs
 - means of hierarchical definition of topology
 - means of vertex refinement
 - vertices themselves can be simple network patterns
 - separate computation from synchronisation, make synchro-patterns analysable
- Coordination agenda
 - mentioned earlier

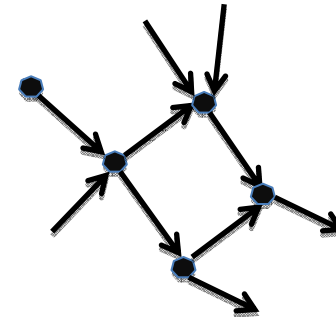
A three layer construction

- 
- Topology and Progress Layer (TPL)
 - connects components
 - controls concurrency and communication
 - provides a “programmable” synch facility
 - Constraint Aggregation Layer
 - does interface reconciliation
 - matches the system configuration to the environment
 - Data and Instrumentation Layer
 - manages shared data
 - provides transaction mechanisms
 - gathers stream statistics

AstraKahn/TPL

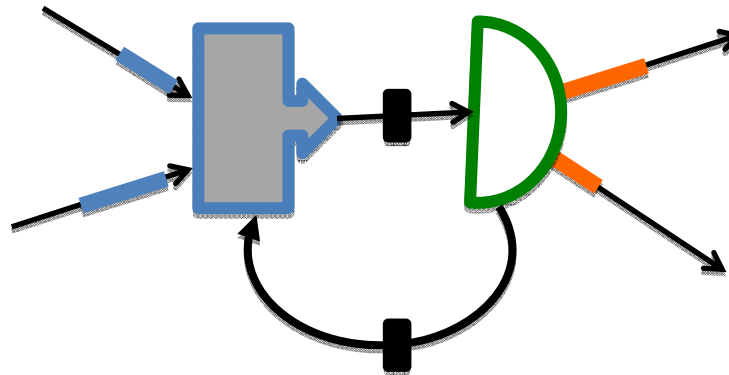
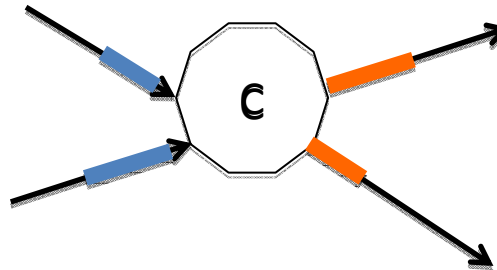
Agenda:

- Classification of vertices
- Synchronisers
- Connectivity
- Concurrency + (self-)tuning



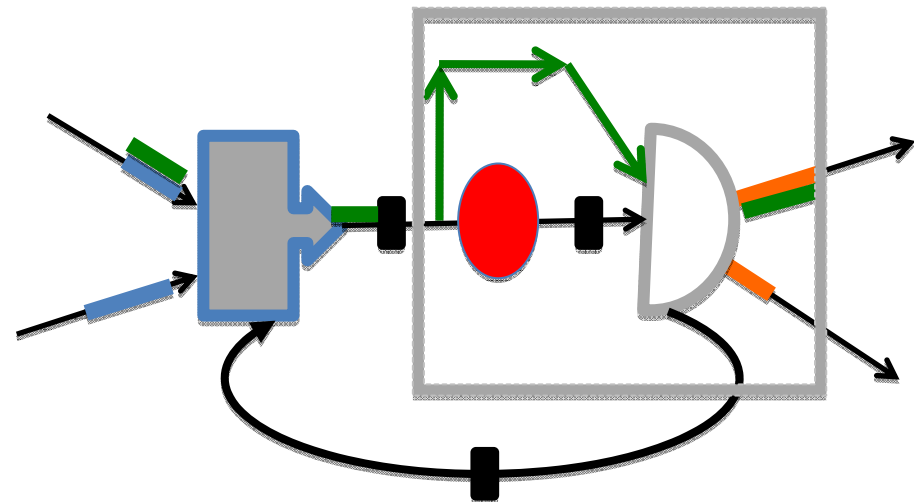
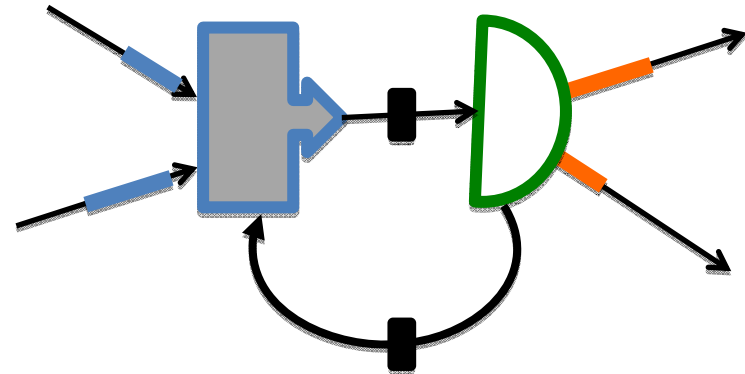
A single Kahn vertex

- Prefix monotonicity
 $\Rightarrow \exists$ prefix
- Refinement:
synchroniser plus
box
- **Synch** has a state,
joins messages into
one, is programmed
in AstraKahn
- **Box** maps one msg
onto output streams,
has no state, is
programmed in a
box language
 - since stateless,
need not >1 input
chan



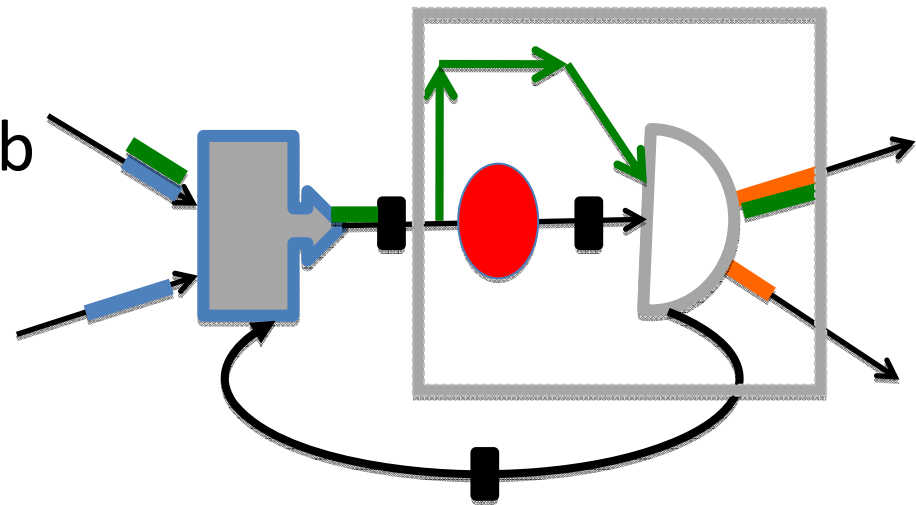
Further refinement

- Use an interface coordination layer (CAL) to refine the vertex down to a **SISO, purely functional core** and a **splitter shell**
- Splitting based on type and (crucially) subtyping
- Take care of inheritance in addition to encapsulation
- encapsulation, inheritance + subtyping = OOP, but now requiring an ab initio construction.
 - hierarchies due to the network decomposition, NOT the nature of messages!



Data Analytics interpretation

- Components written in std languages [Cray,IBM]
 - red blobs is the “algorithm”, available globally.
 - since stateless
 - a message carries a job
 - a channel is a trigger
 - a splitter is a router
- red blobs gets instantiated near the blue data site
 - data movement is minimised



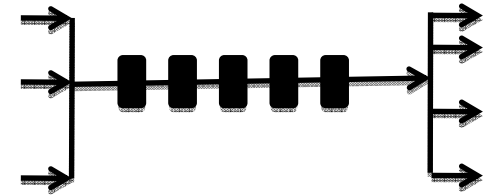
A single channel

- A FIFO of limited size
- Concept of pressure
 - pressure $p = \#$ of msg in channel
 - $p \leq p_{cr}$
 - blocking write (ext. of KPN)
 - negative pressure
 - $p = \#$ of msg the consumer promises to consume **immediately**
 - **Remember: a channel is a trigger** not a communication medium between stationary processes...



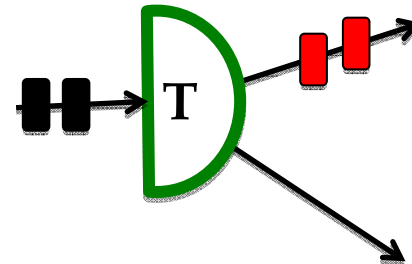
Multiple Input Multiple Output

- Many inputs merged nondeterministically into a single FIFO
 - don't want n/d then don't use multiple inputs!
- The output is copied to multiple destinations
- Any consumer can cause critical pressure
- Negative pressure = max over the consumers' pressures



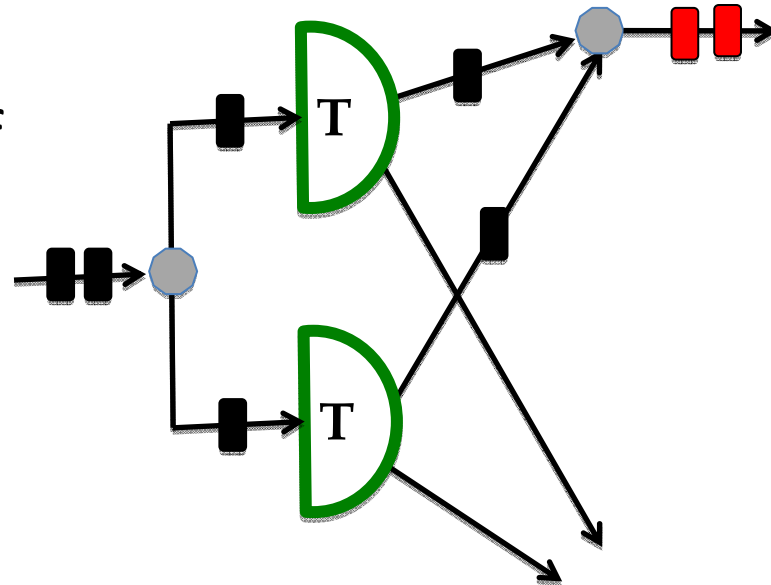
Boxes: transductor (“map”)

- Simplest box category
- no more than 1 output message per input message per channel
- box is not run if the output pressure is critical on any of its output chans; this increases input pressure
 - “back-propagation” of pressure
- single step semantics
- Concurrency



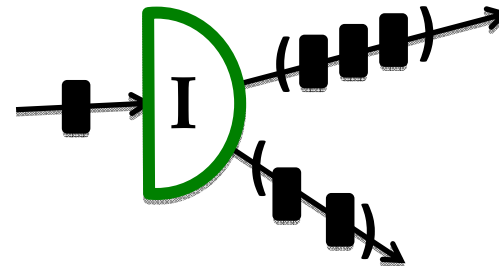
Proliferation

- Simplest box category
- no more than 1 output message per input message per channel
- box is not run if the output pressure is critical on any of its output chans; this increases input pressure
 - “back-propagation” of pressure
- single step semantics
- Concurrency



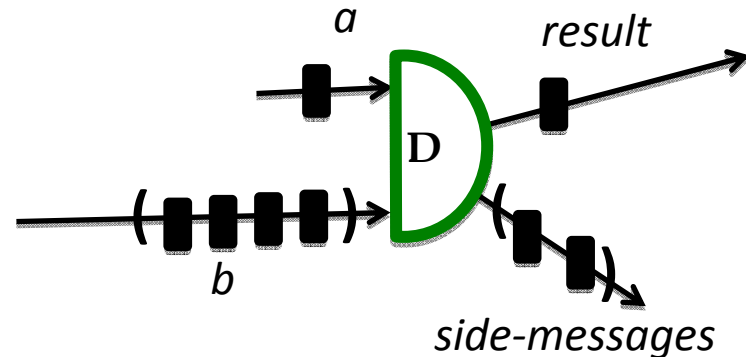
Boxes: inductor

- one input msg => a seq of output msgs per chan
- “brackets” demarcate seqs, without taking space in FIFOs
- behaviour under pressure
- boxes do not see, nor produce brackets

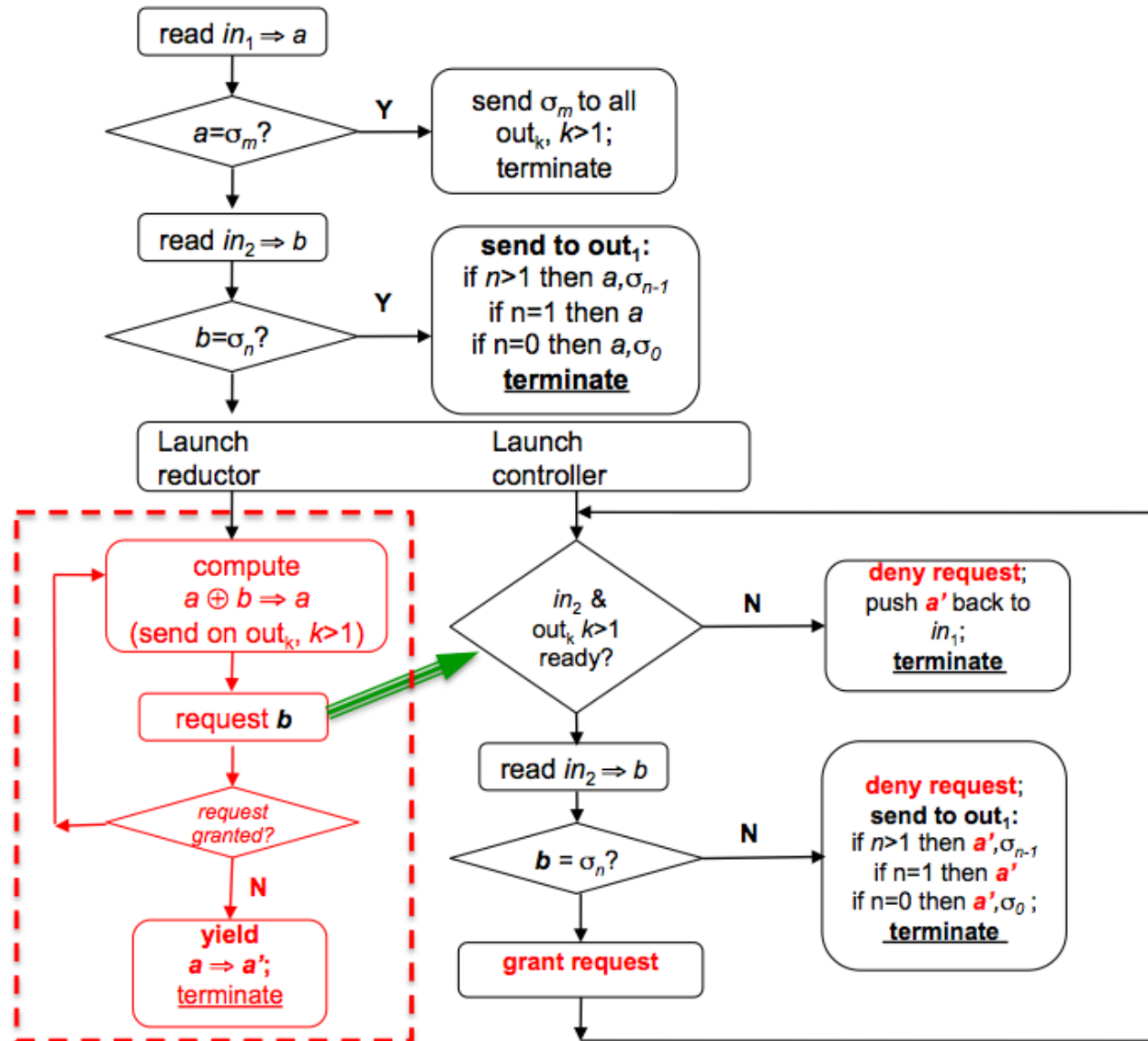


Boxes: reductor

- Dyadic reductor
based on most general
 $\oplus : a \rightarrow b \rightarrow a$
 $a \oplus b_1 \oplus b_2 \oplus b_3 \dots \oplus b_n$
- single-step semantics
under pressure
(animation)
- Concurrency
classification:
 - 2DO
 - 2DU

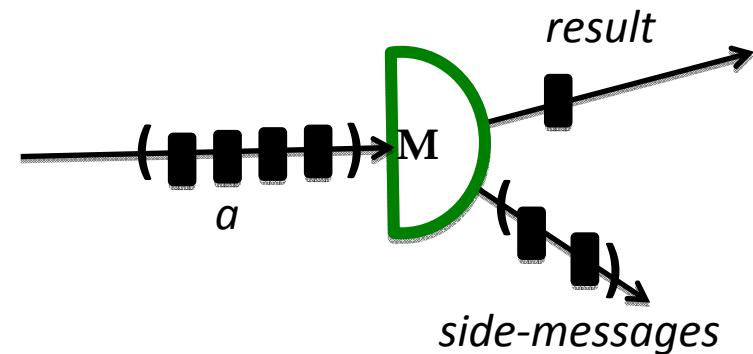


Dydadic reductor protocol



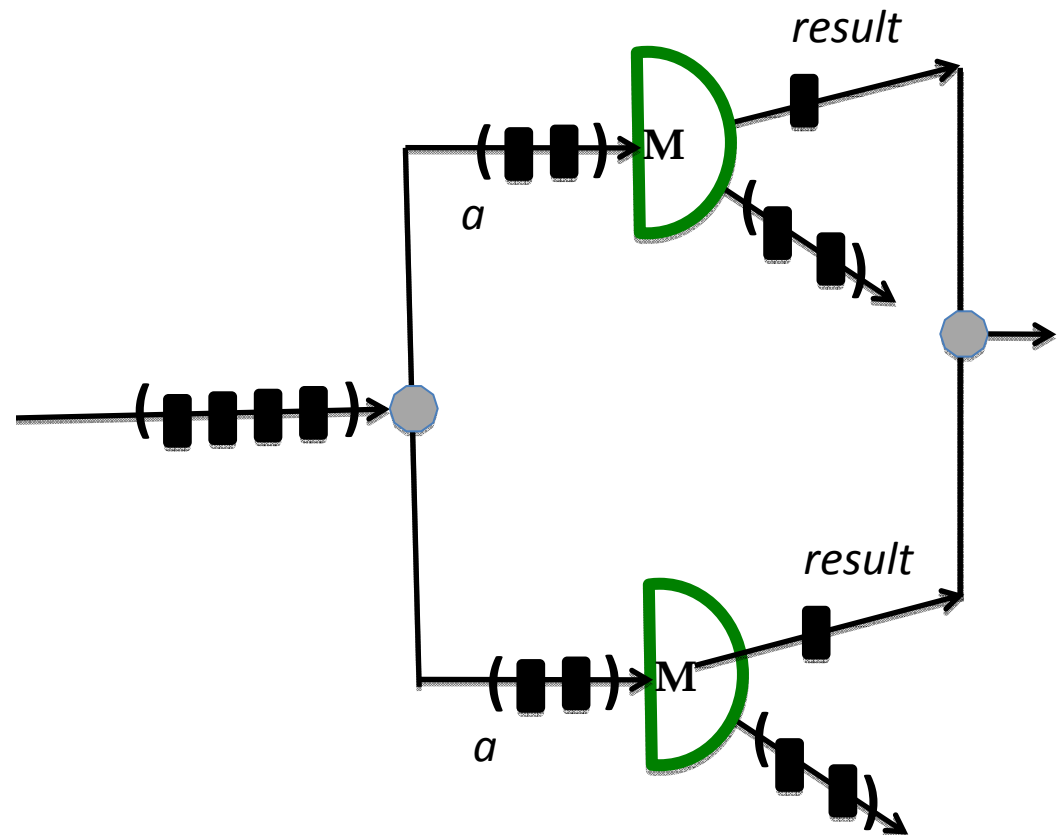
Monadic reductor

- same as dyadic except the operator is $a \rightarrow a \rightarrow a$
- richer classification
 - 2MO
 - 2MS
 - 2MU



Monadic reductor: proliferation

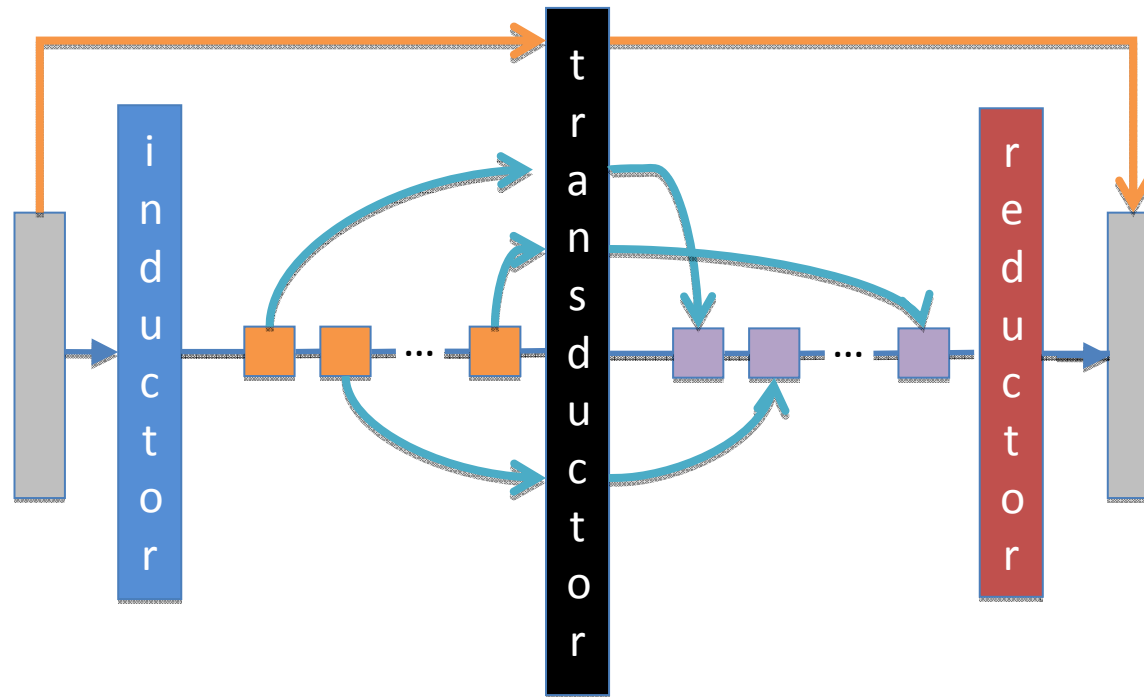
- same as dyadic except the operator is $a \rightarrow a \rightarrow a$
- richer classification
 - 2MO
 - 2MS
 - 2MU



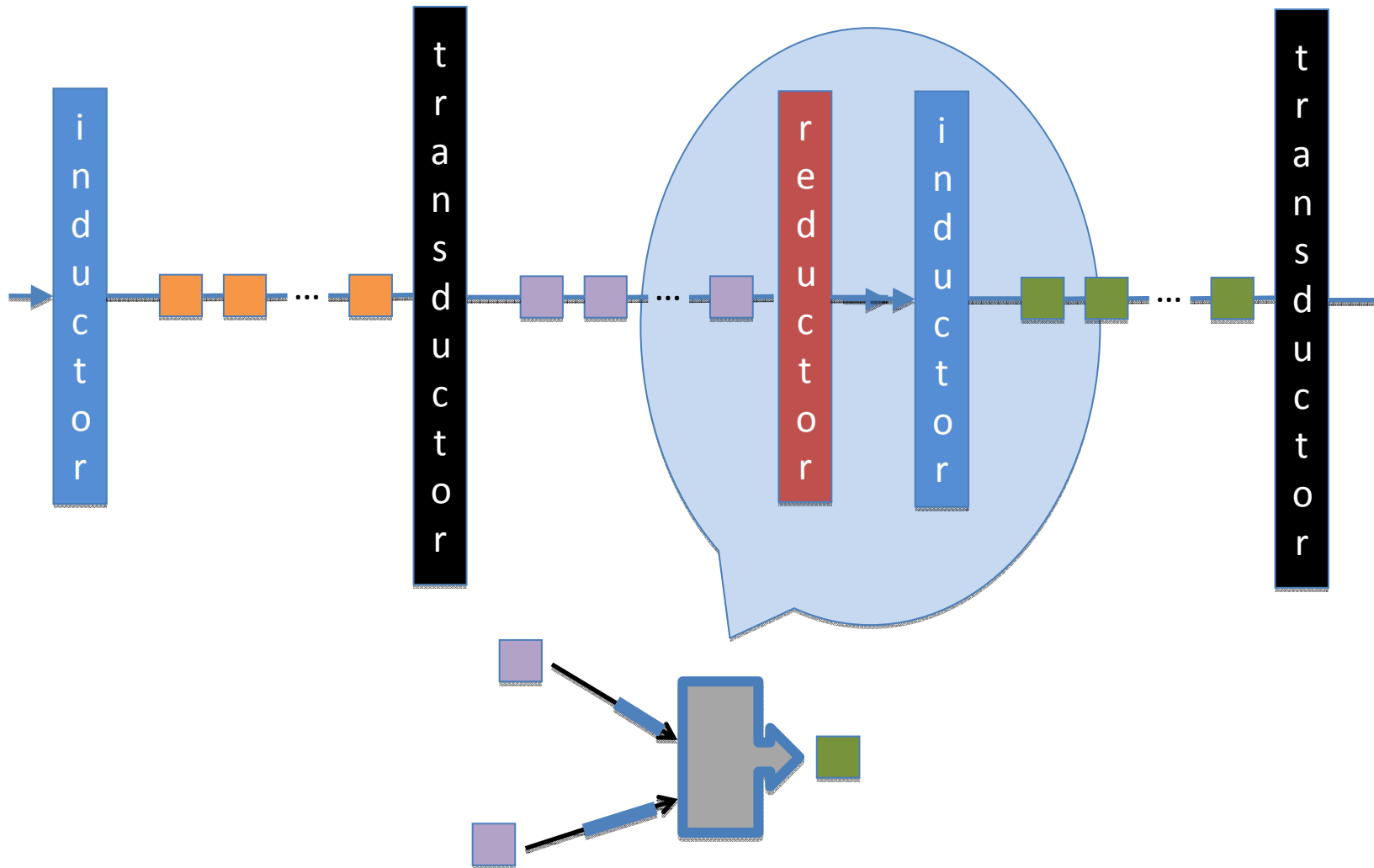
Morphisms

- Proliferation happens under high supply and demand
 - Positive pressure on the input: many messages available
 - Negative pressure on the output: many result messages can be accepted immediately.
- What if there is only one record?
 - The box may allow data parallelism
 - Input message split into many, results combined
- A morphism is an automatic replacement of a transducer by a simple net.

Morphism



Override



Synchronisers

- Purpose: to accumulate sufficient prefix from all input channels; to form an output msg(s) and send it/them to appropriate output channels
- A synch is a combination of an FSM and a path functional implemented as call-up storage.
 - **not** a stack-machine, Turing machine, etc.
 - call-up storage is **not** analysed in transitions
 - only purpose of call-up storage is to **form output msgs**

Synch Example: zipper

```
synch zip2(a:0,b:0 | c:0)
{
    store ma:a, mb:b;

start:  on a do
        ma:= this
        goto s1;
    on b do
        mb:= this
        goto s2;

s1:     on b send (ma,this) => c        goto start;
s2:     on a send (mb,this) => c        goto start;
}
```

Synchronisers can deal with brackets

```
synch listMerge (a:d, b:d | c:d+1)
```

```
{
```

```
start:
```

```
    on a.@k & k=d send this => c goto alt;
```

```
    on a.else send this => c goto start;
```

```
alt:
```

```
    on b.@k & k=d send @k+1 => c goto start;
```

```
    on b.else send this => c goto alt;
```

```
}
```

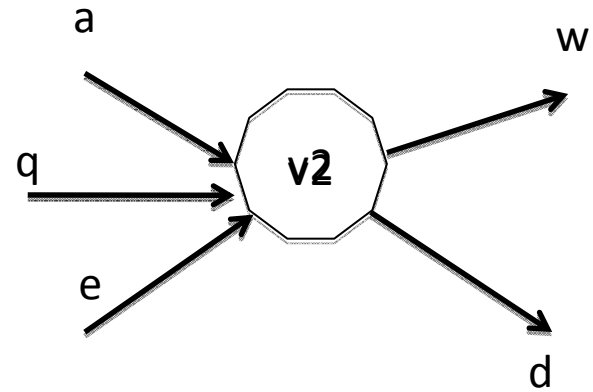
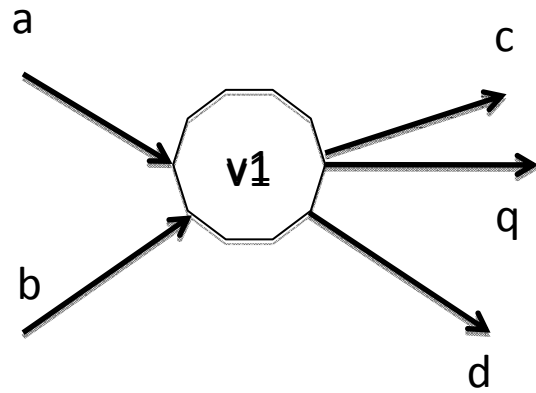
Synchs are sources of pressure

- can block input channel
- by analysing the transition diagram can apply negative pressure to input channels
 - when msgs on those are required for progress
 - by observing the queue on other inputs dependent on that one
- synchs are basis for machine learning and observation-led adaptation

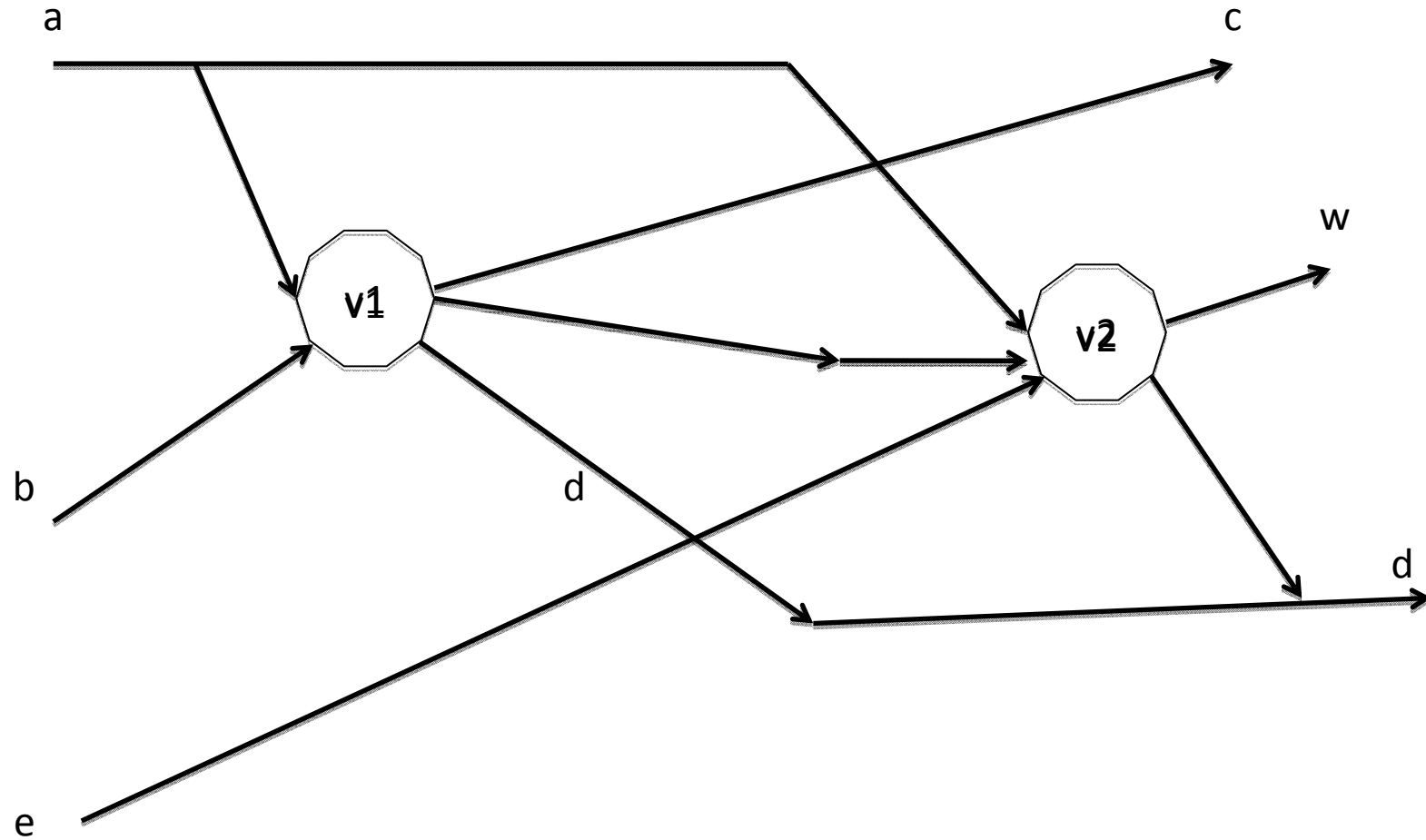
Wiring: box labelling

- Boxes are declared based on their category and the number of output channels, e.g. 2DU, 3T, etc.
- Input and output channels are numbered, not named
- wiring is, by contrast, based on names
- remember: synchs already use named channels
- AK naming construct (naming parentheses) for boxes:
 - e.g. instantiate a 3DU box R:
 <a, b | R | c, d, e>

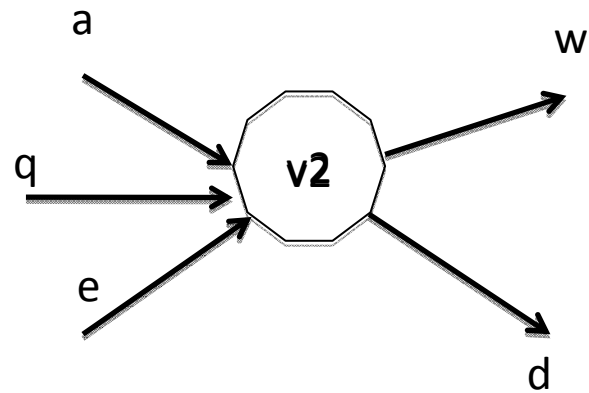
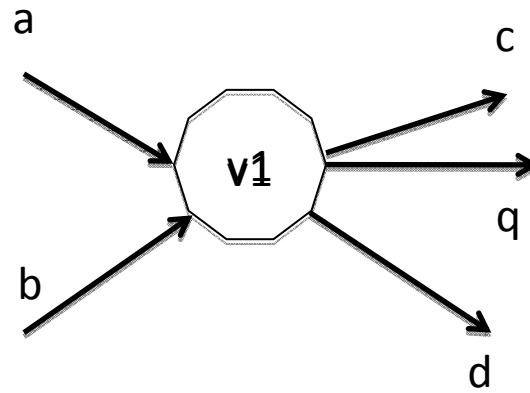
serial composition



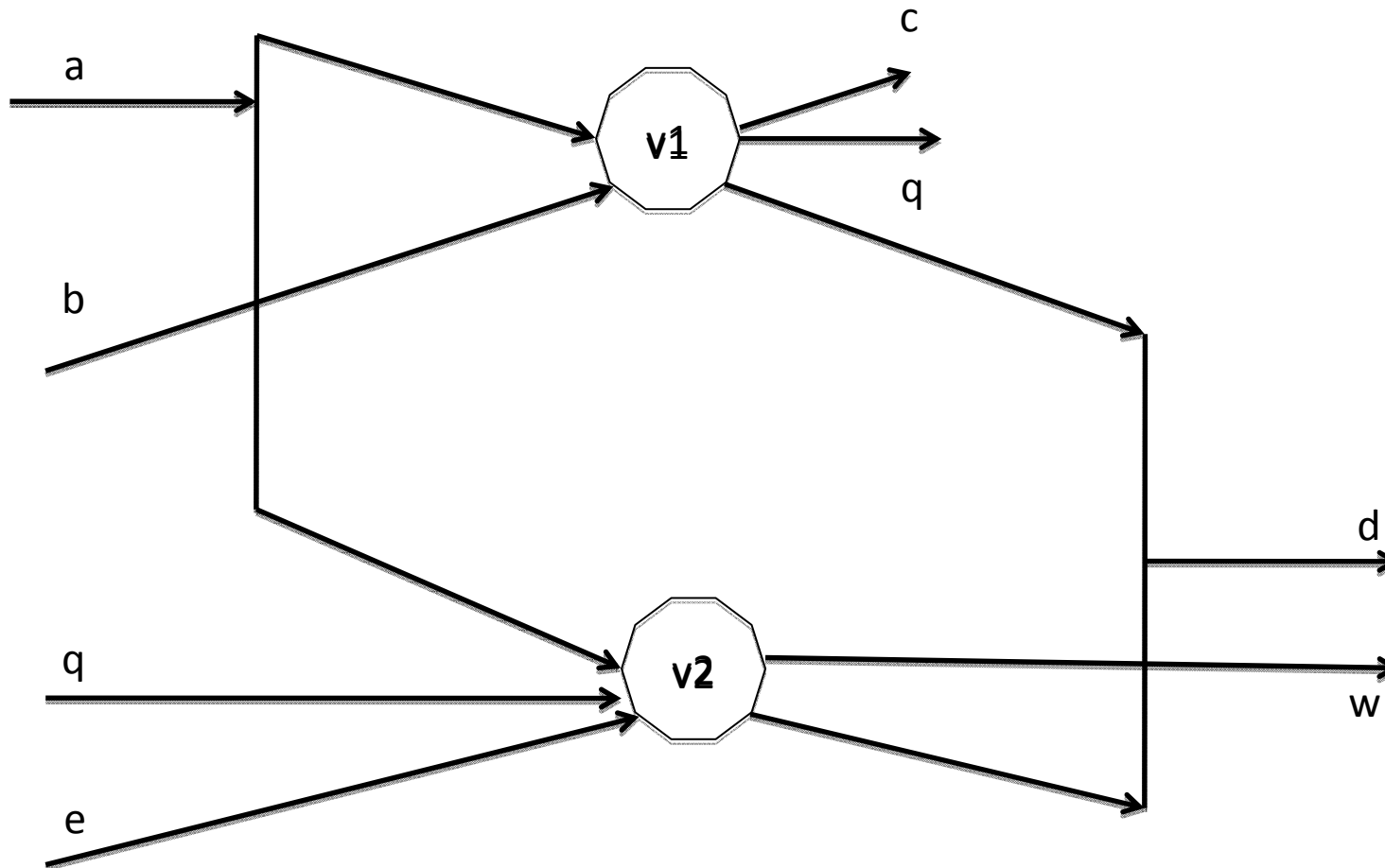
serial composition



parallel composition

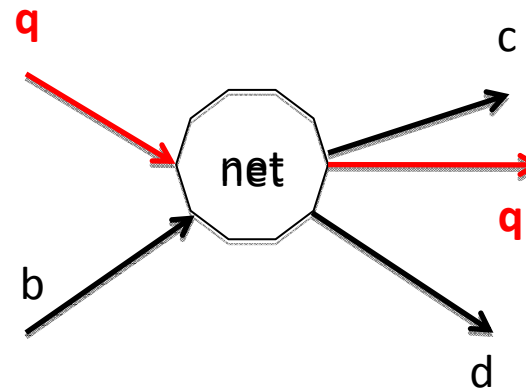


parallel composition



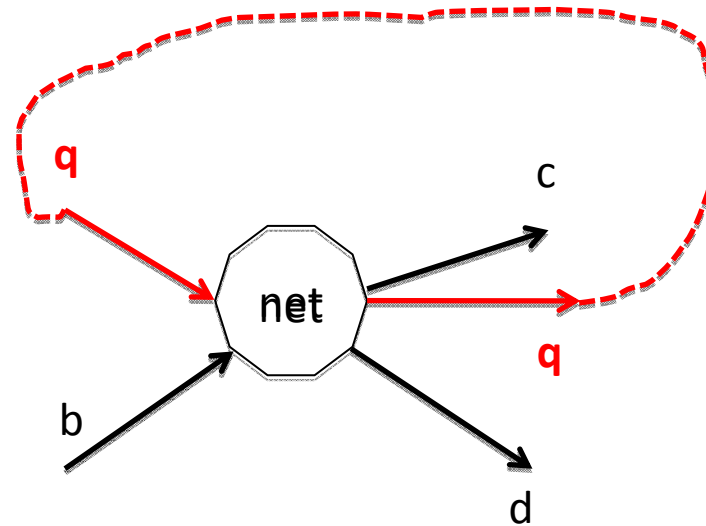
Feed-back

- Consider a net with a pair of channels named identically



Feed-back

- the feed-back channel
 - depressurised
 - critical pressure = infinity, but out-of-memory situation is possible
 - pressure transfer mechanism, e.g. transfer pressure from q to b
 - can be negative as well

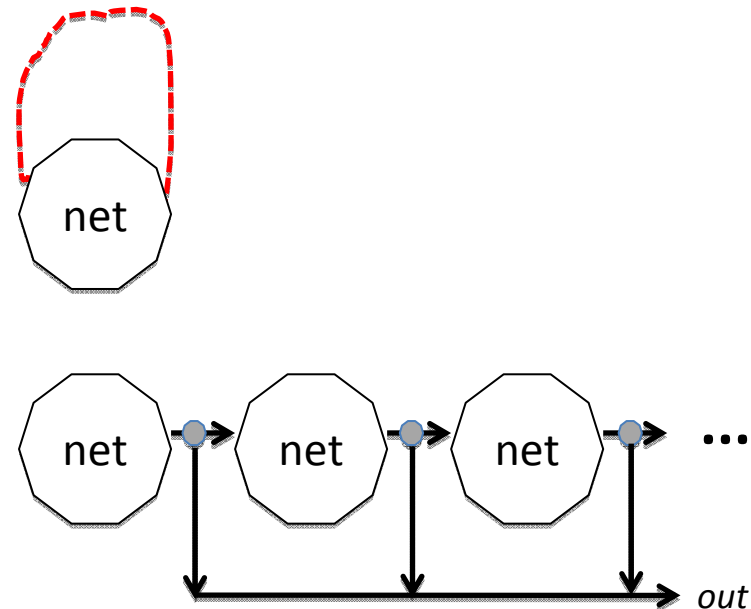


Wiring is generic: can wire any topology, incl. cyclic, using \cdot , \parallel , \backslash

$$\left((v_1^{[0]} \parallel v_2^{[0]} \parallel \dots \parallel v_{k_0}^{[0]}) \bullet \bullet (v_1^{[1]} \parallel v_2^{[1]} \parallel \dots \parallel v_{k_1}^{[1]}) \bullet \bullet \dots \bullet \bullet (v_1^{[d]} \parallel v_2^{[d]} \parallel \dots \parallel v_{k_d}^{[d]}) \right) \backslash$$

Asterisk (hence *AstraKahn*)

- Feedback depressurises the channel
- Suppresses progress control by self-regulation
- Instead of feedback:
 - unroll the cycle
 - introduce feed-forward
 - output based on a fixed point
- The fixed point
 - defined as a path via synchronisers
 - where the message is guaranteed
 - not to change
 - not to cause a change of state
- Reverse fixed point to reclaim the head of the chain
- **Pressure** unfolds the chain



Conclusions

- Full map-reduce style concurrency
- Coordination without tuning due to the pressure mechanism
- Flexible synchronisation, incl. useful nondeterminism
- Separation of concerns through coordination

Future

- implementation as a library
 - All elements of the TPL implemented as API
 - synchro-language implemented as a (micro-) compiler
- “proper” implementation
 - CAL worked on by a PhD student
 - box-language interface with C will be attempted first