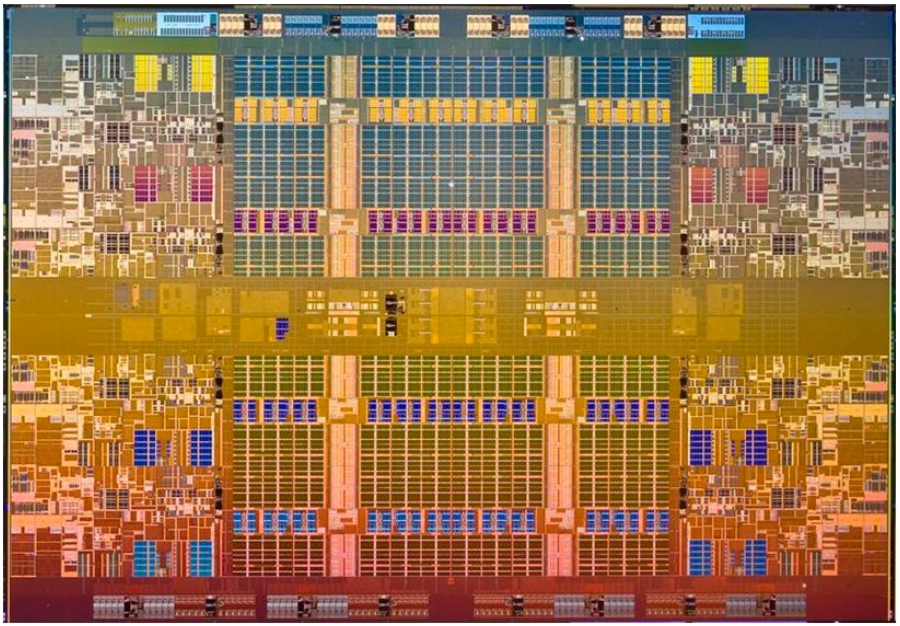
Challenges in verifying communication fabrics

Alexander Gotmanov, Satrajit Chatterjee,
Yuriy Viktorov, Michael Kishinevsky

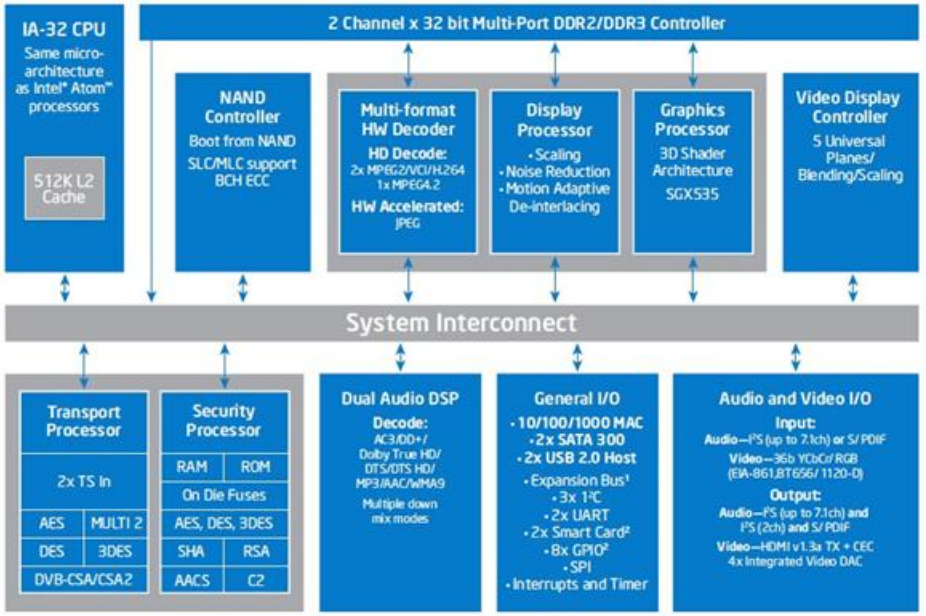
Strategic CAD Labs, Intel

ITP conference August 22, 2011

Communication Fabrics
= logic connecting different agents on a chip



Intel® Xeon® processor 7500. 8 cores, 16 threads, 24 MB LLC, glueless 8 socket systems

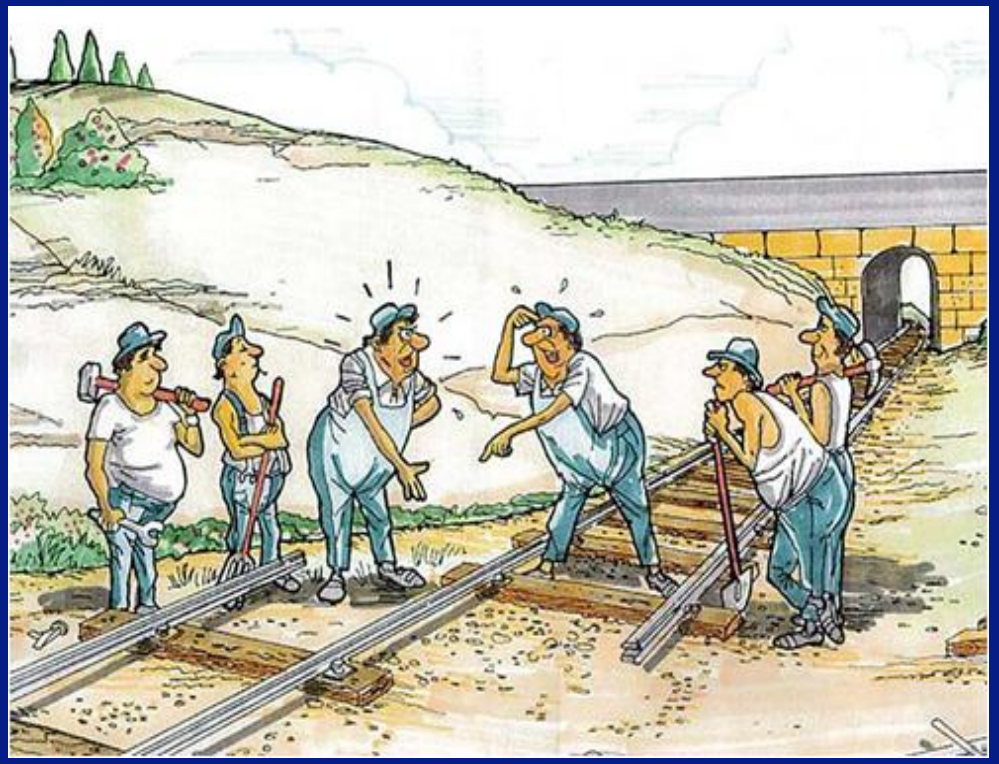


Intel® Atom® processor CE4100

- Many interconnect fabrics talking to each other (NoCs like or ad hoc)
  - Coherent interconnect between cores
  - IO fabrics, memory fabrics
  - Sideband fabrics for power management or debug
- Critical for fast product integration, correctness and quality (performance, power, cost)

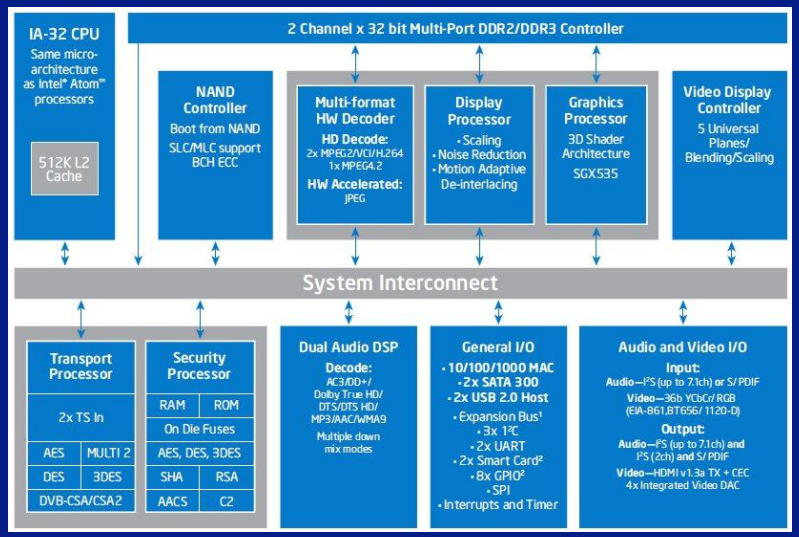
Verification of communication fabrics is complex

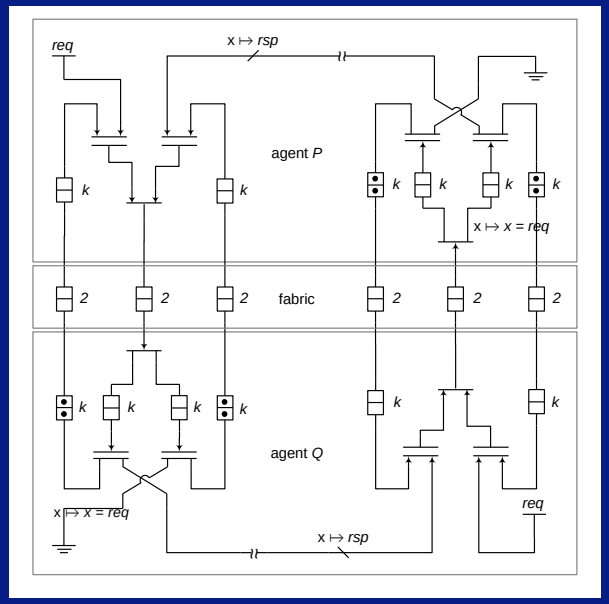
- Tricky, distributed interaction
  - deadlock
  - starvation
  - Ordering
- Problems seen late
  - during integration
  - late RTL or in silicon



Our Approach

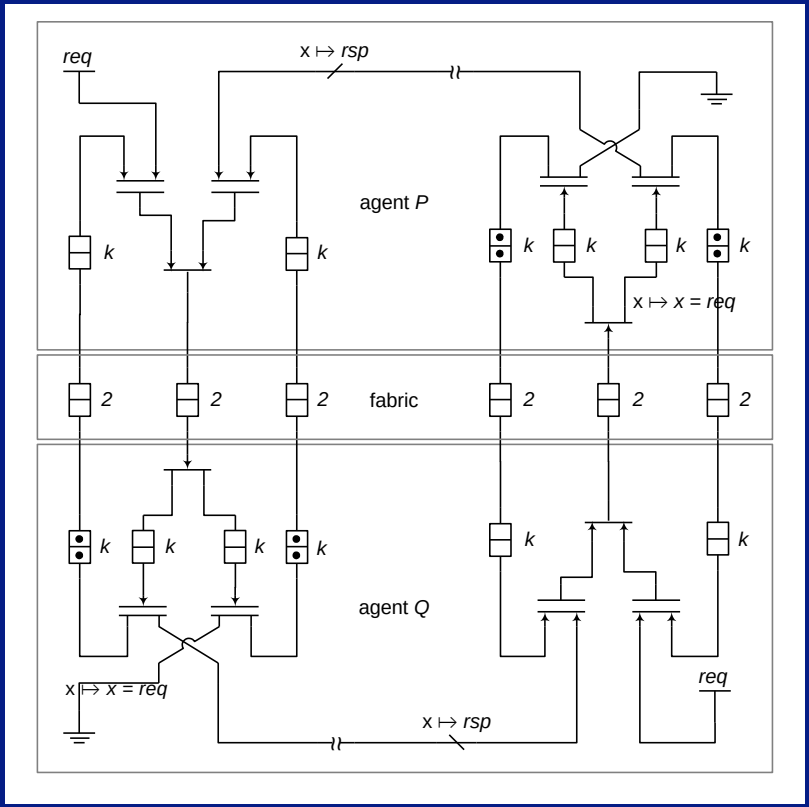
Build early, abstract models of the microarchitecture





Goal: Prove correctness on these models
and influence development of micro-architecture

Two key ideas



1. Capture models at a "high-level" of abstraction

2. Exploit high-level structure for quick automatic proofs

Can prove properties that are otherwise intractable with automatic methods (model checking) or require tedious manual work (theorem proving)

Outline

- How to capture "high-level" structure?
- How to exploit "high-level" structure?

Sat Chatterjee, Mike Kishinevsky, Umit Ogras "Quick Formal Modeling of Communication Fabrics to Enable Verification" HLDVT 2010

Sat Chatterjee, Mike Kishinevsky "Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics" CAV 2010

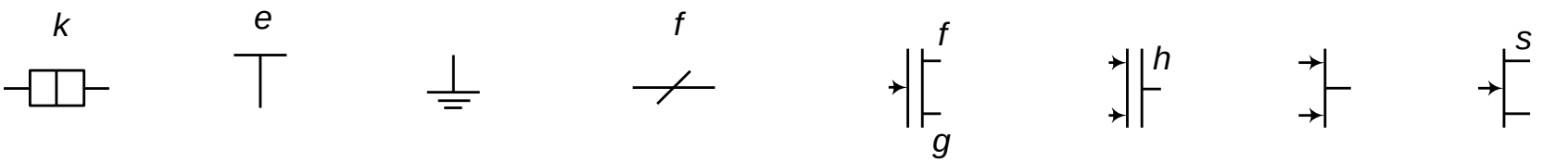
Alexander Gotmanov, Sat Chatterjee, Mike Kishinevsky "Verifying Deadlock-Freedom of Communication Fabrics" VMCAI 2011

ABC system (Alan Mishchenko, Bob Brayton, Sat Chatterjee and recently Niklas Een) : Berkeley Logic Synthesis Group. http://www.eecs.berkeley.edu/~alanmi/abc/

Sayak Ray implemented l2s command for converting liveness to safety using Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science 66(2), 160 – 177 (2002)

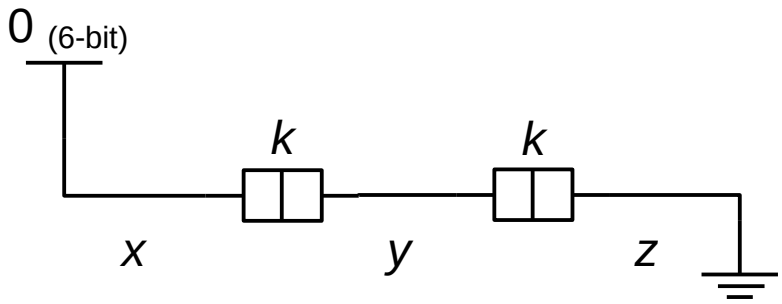
xMAS Compositional Modeling

Kernel primitives



Models (including bigger modules) are networks of primitives

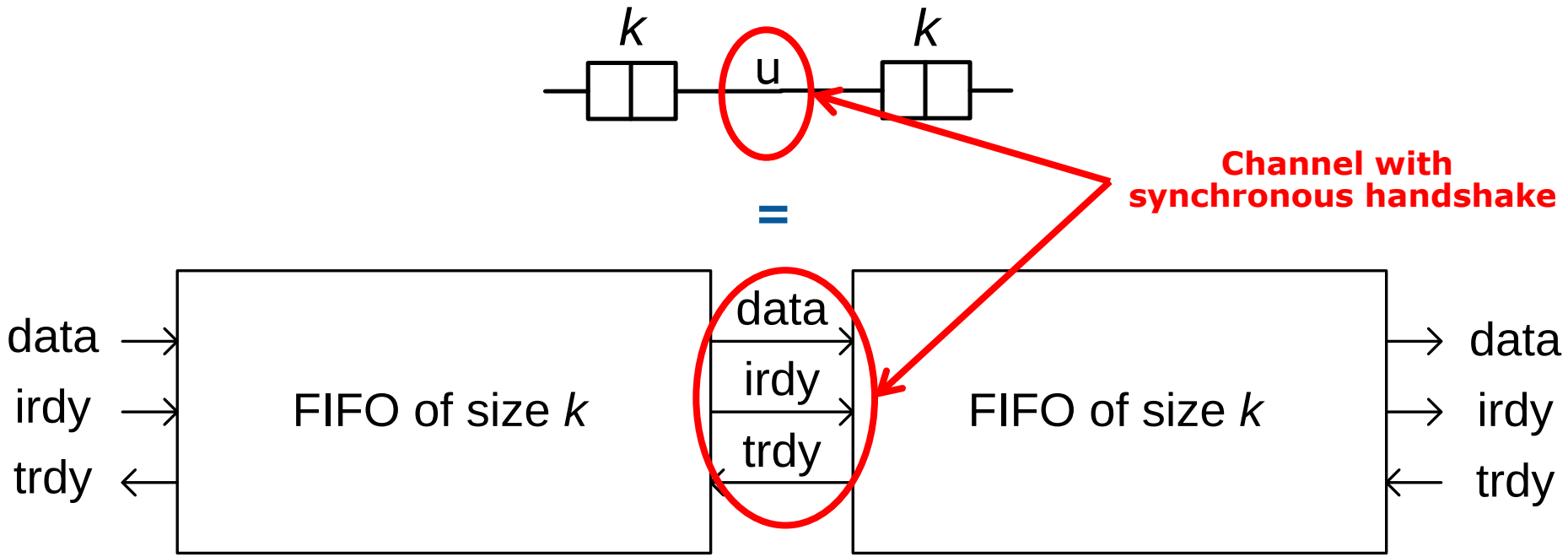
Behavior: sequences of data transfers



Synchronous Semantics

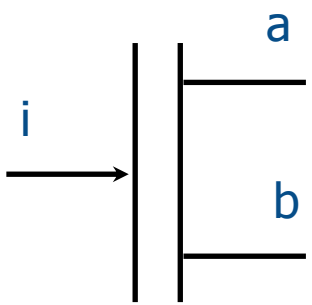
Each primitive is a synchronous module (think single clock Verilog)

Different modules are connected by channels



Transfer: xfer(u) = u.irdy & u.trdy

Synchronous model of a fork



Equations

a.irdy **:=** i.irdy **and** b.trdy

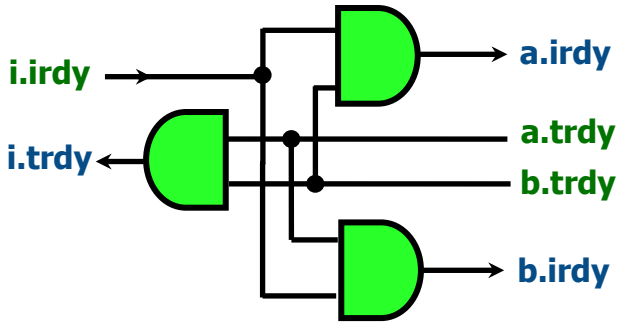
b.irdy **:=** i.irdy **and** a.trdy

i.trdy **:=** a.trdy **and** b.trdy

a.data := i.data

b.data := i.data

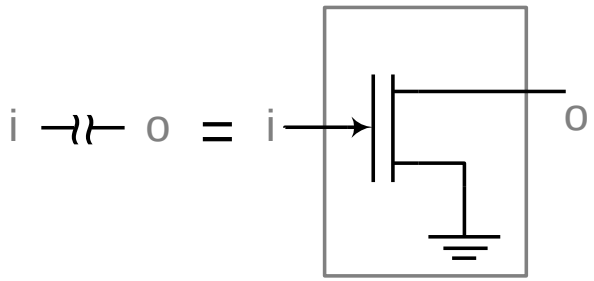
Netlist (control part)



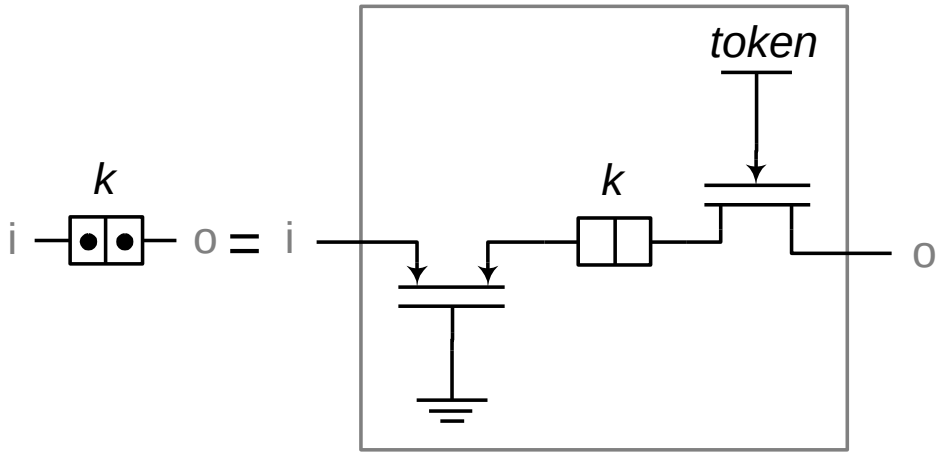
Lazy fork: waits for readiness of both targets and then transfers

xfer(i) = **xfer**(a) = **xfer**(b)

Two examples of common macro modules



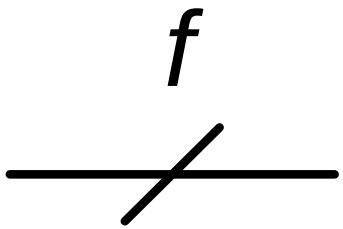
non-deterministic delay



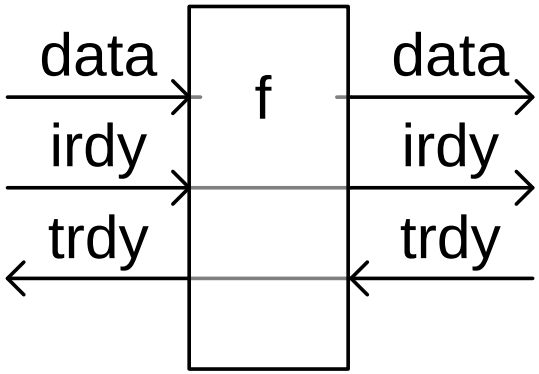
credit logic

xMAS Semantics

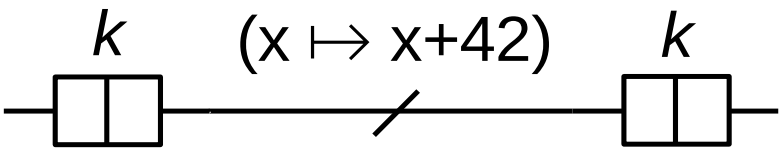
Primitives are parameterized

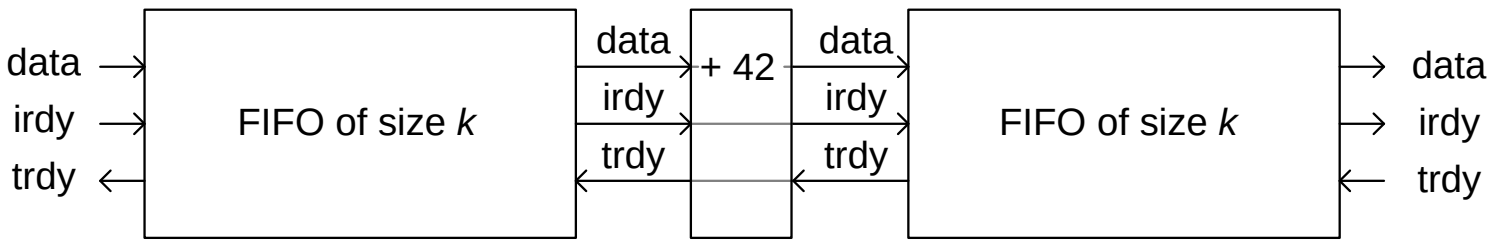


is short-hand for

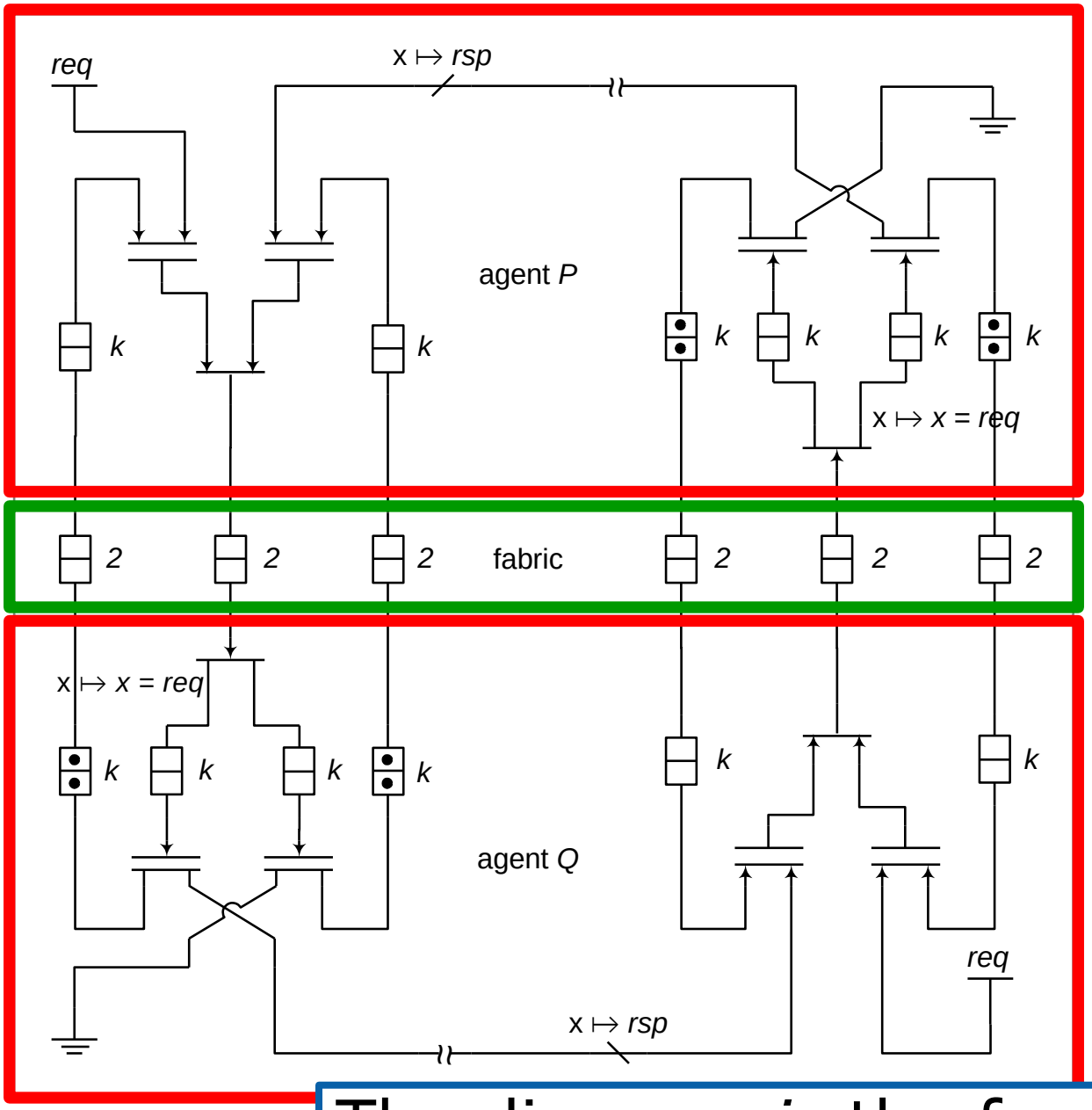


Example





Virtual Channels



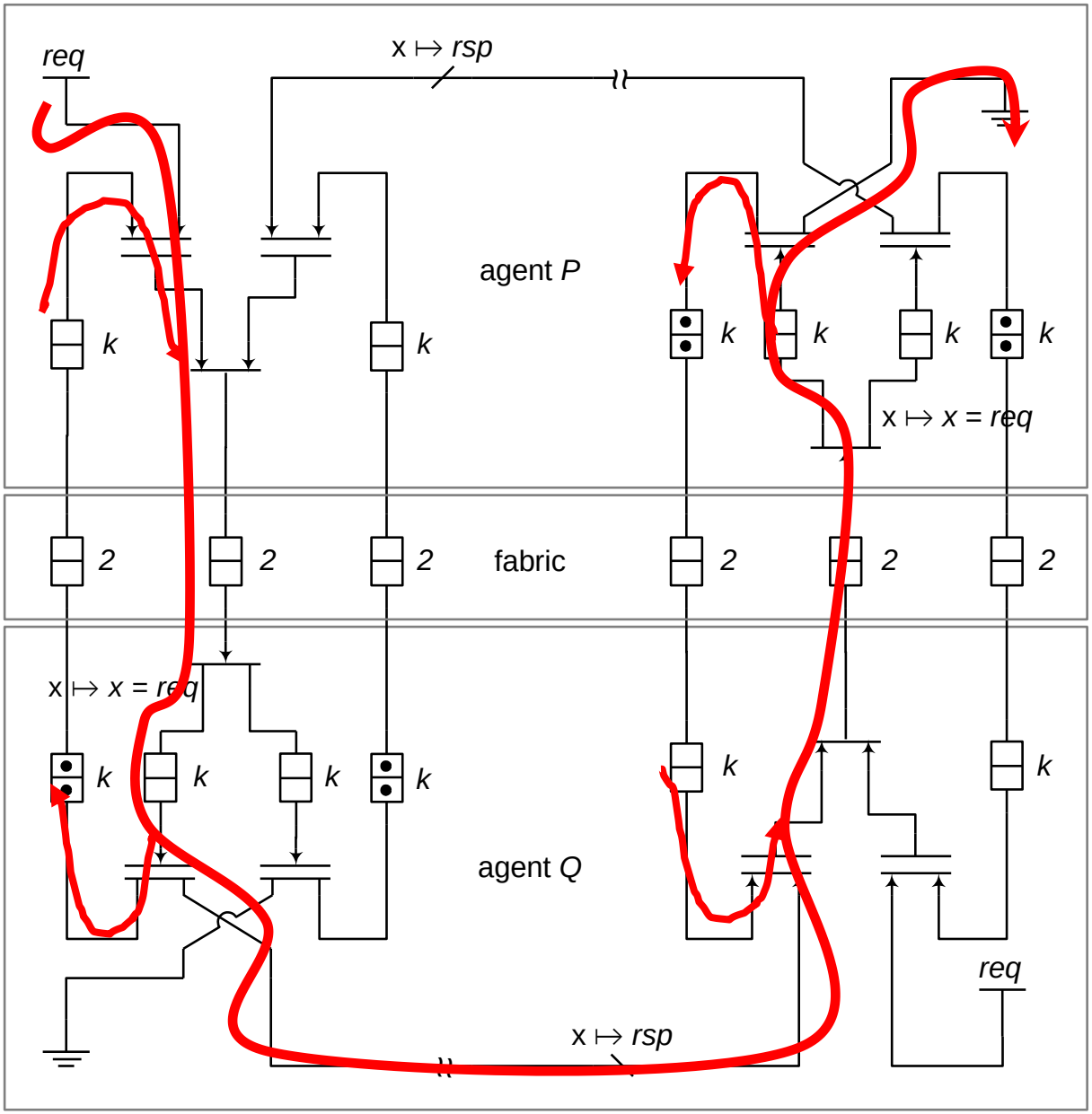
Agent P

Fabric

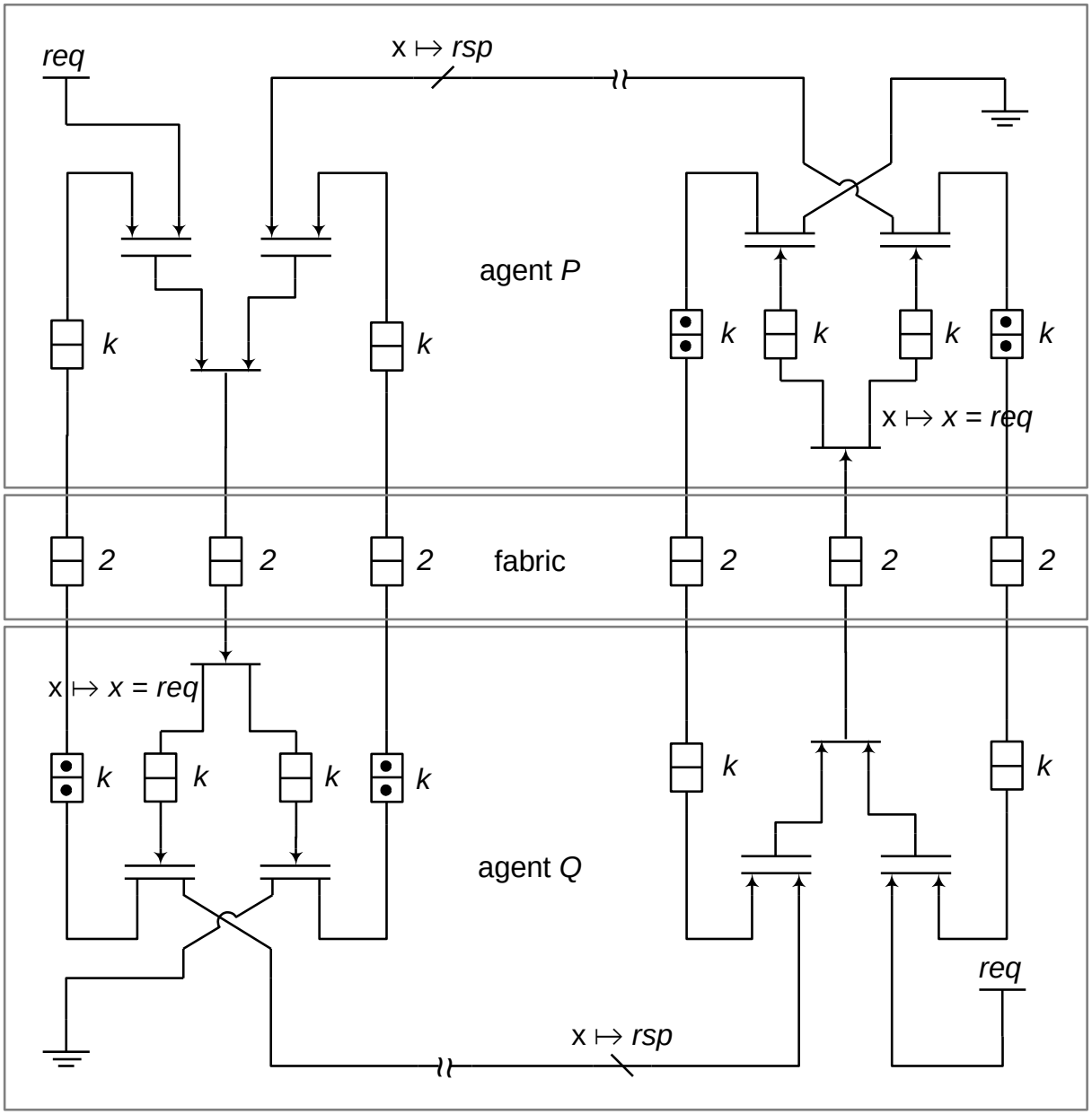
Agent Q

The diagram *is* the formal model!

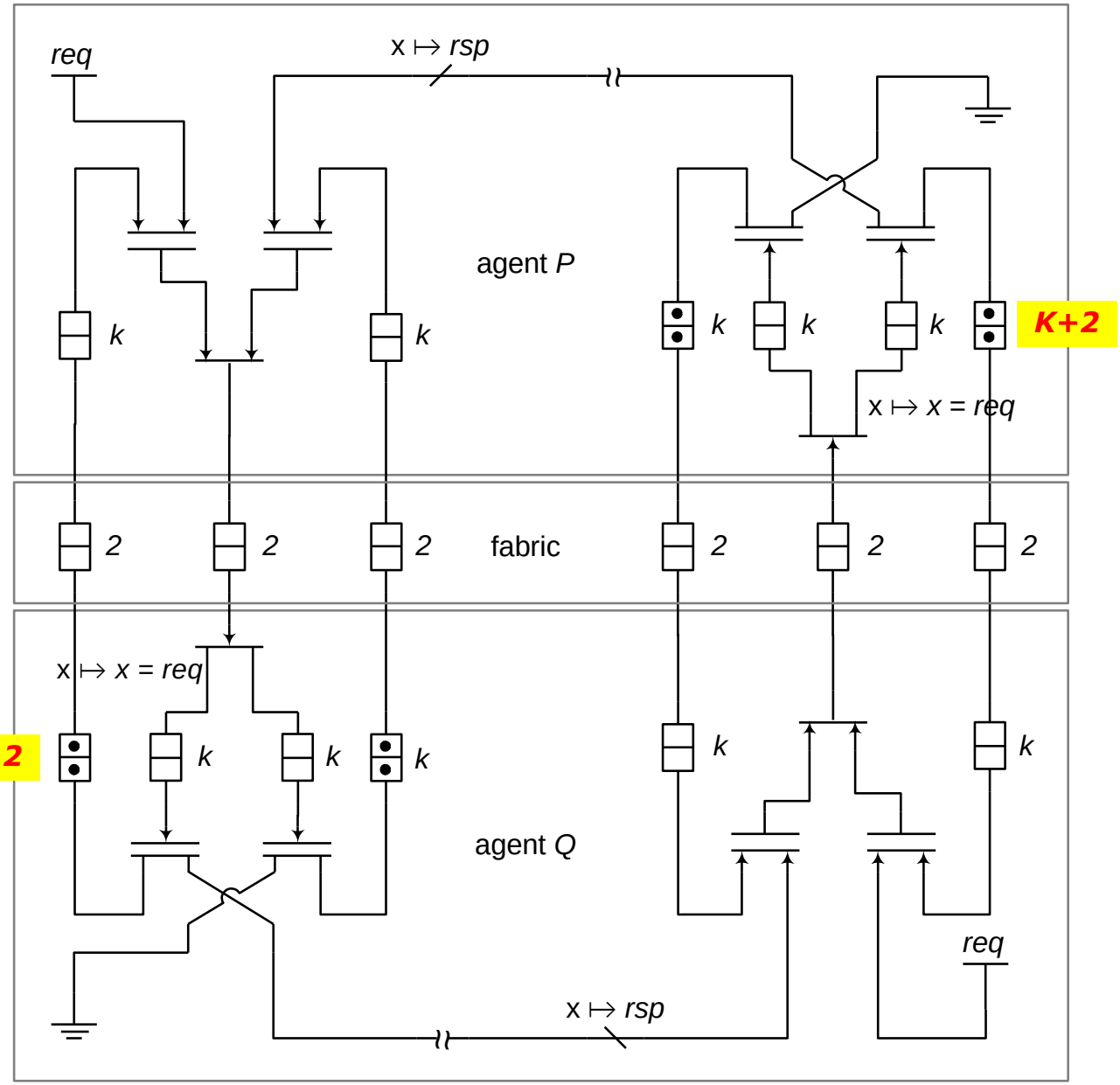
Virtual Channels



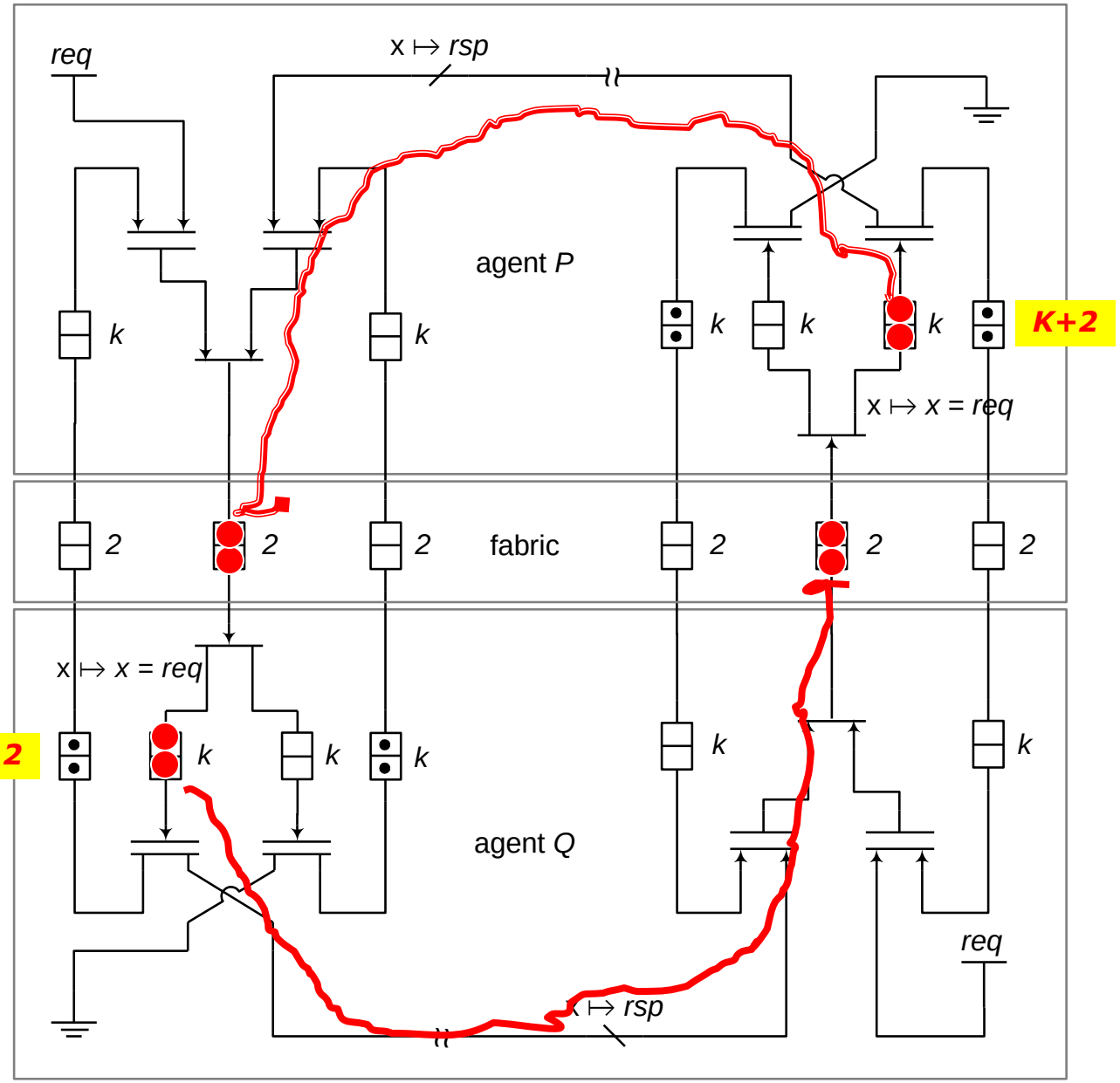
Quiz: Is there a deadlock?



Quiz: Is there a deadlock?



Quiz: Is there a deadlock? Answer



req

rsp

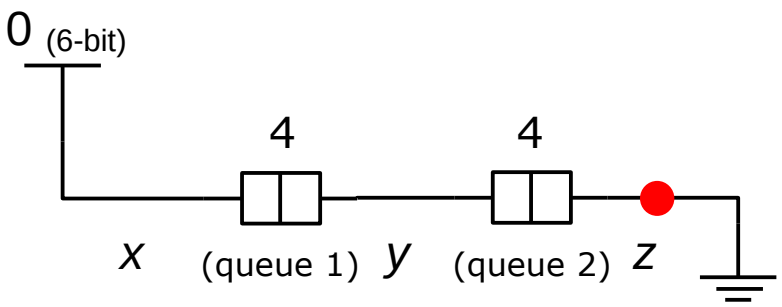
Outline

- How to capture "high-level" structure?
- How to exploit "high-level" structure?
  - Safety properties
  - Liveness properties

A Simple Example

A *channel property* checks that all packets on a channel satisfy some condition

- e.g. all packets received by an agent have correct dest_id



Example of channel property:

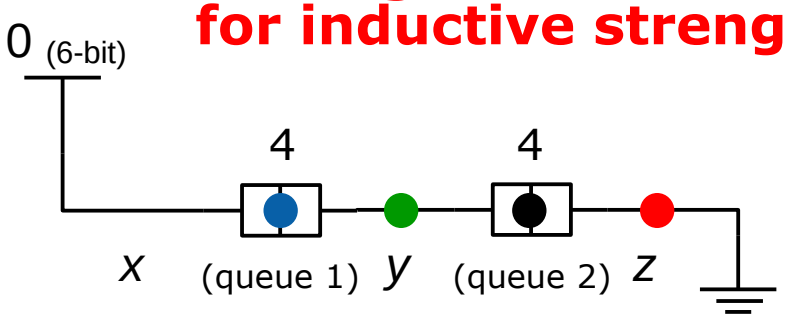
G (*z*.irdy ⇒ (*z*.data = 0))

Although this property is obviously true:

- Interpolation (in ABC) takes 10 mins to prove this!
- Explicit or BDD-based reachability not an option for large models

Property propagation

Use high-level structure to add invariants for inductive strengthening



Target channel property:

G (z.irdy ⇒ (z.data = 0))

Invariant 1: **G** ($used_j$ ⇒ (mem_j = 0))

If location j in queue 2 is in use, it must contain 0

Invariant 2: **G** (y.irdy ⇒ (y.data = 0))

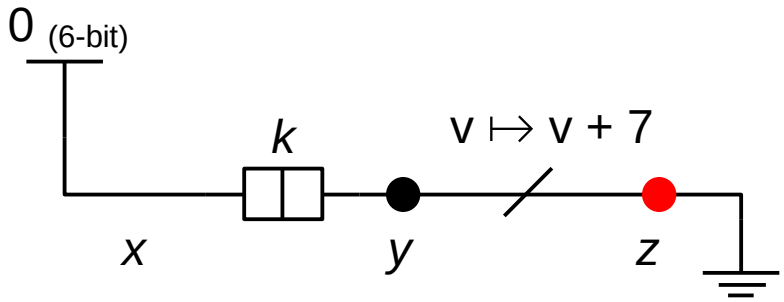
Invariant 3: **G** ($used_j$ ⇒ (mem_j = 0)) (For queue 1)

This set of invariants is inductive!

Automatic: based on property propagation and local invariants of queues

Similar propagation for other primitives

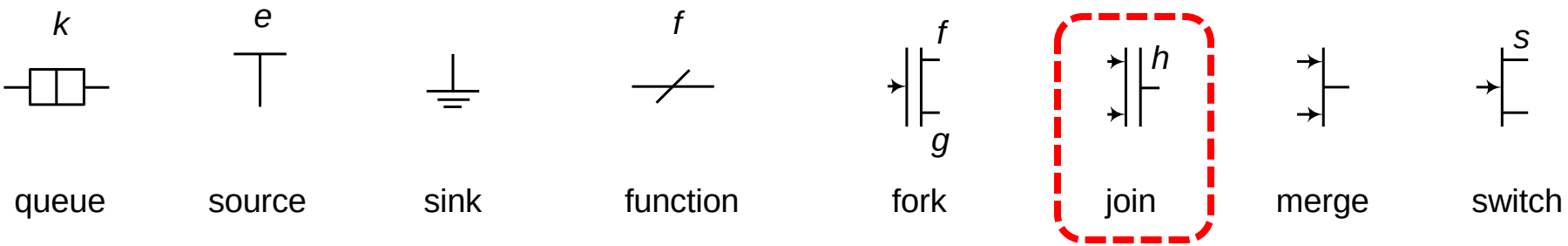
Example: Propagation across function primitive



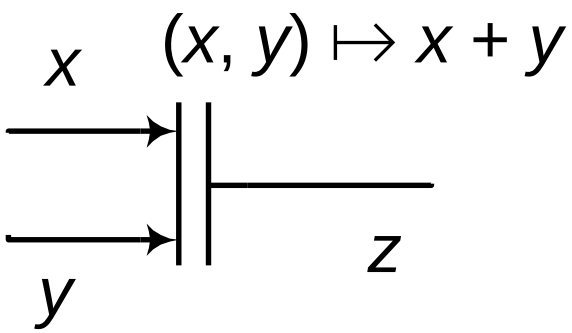
Property: **G** (*z*.irdy ⇒ (*z*.data = 7))

Invariant: **G** (y.irdy ⇒ ((y.data+7)=7))

Careful with Joins

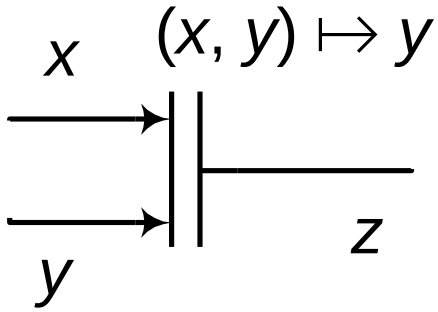


Hard case



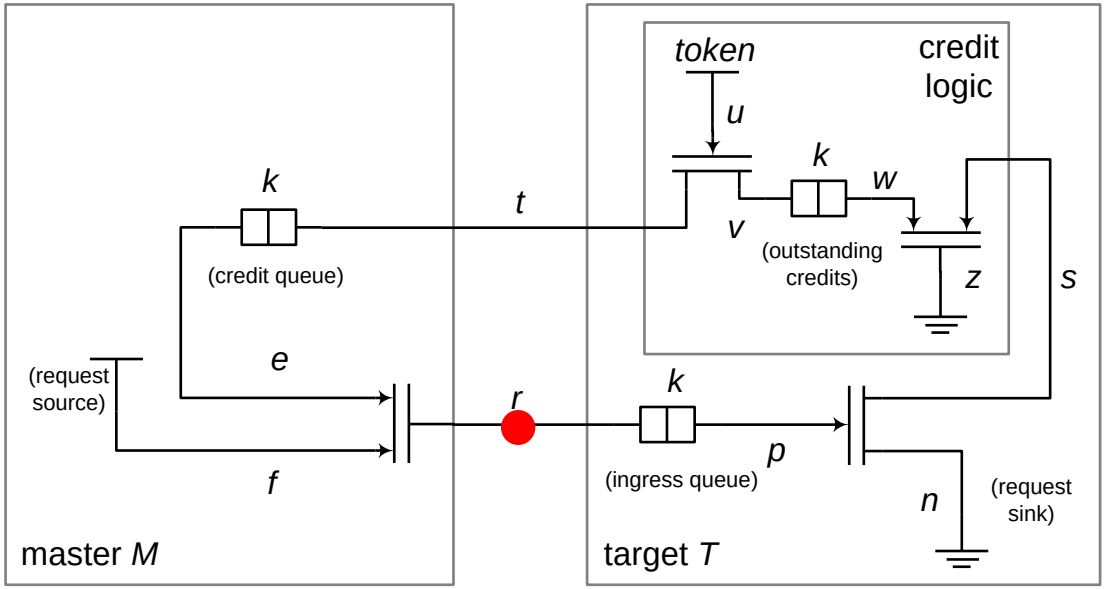
G (*z*.irdy ⇒ (*z*.data = 42))

Easy case



Most joins like this in practice

Non-blocking by Example: Credit Logic

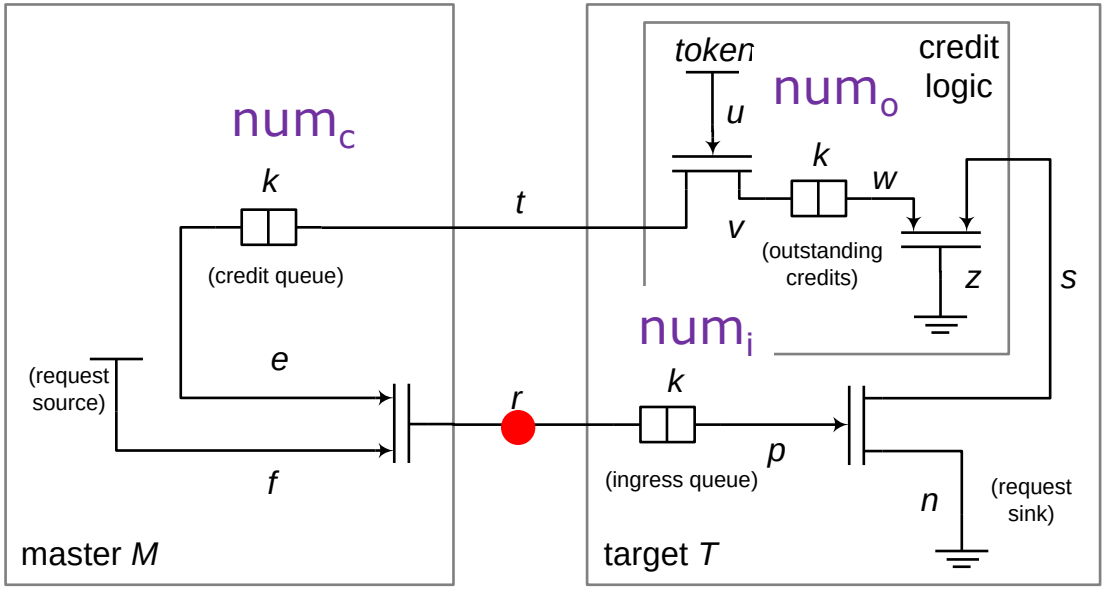


Non-blocking property on channel *r*: **G** (*r*.irdy ⇒ *r*.trdy)

Intuition: If a packet is in *r* there is room in the ingress queue (useful to reason about liveness)

Hard property to prove: Not inductive!

Strengthening non-blocking properties using global invariants



Non-blocking property on channel *r*: **G** (*r*.irdy ⇒ *r*.trdy)

(Inductive) Invariant: **G** (num_i + num_c= num_o)

How to find $num_i + num_c = num_o$ automatically?

Introduce λ variables to count transfers on each channel

$$\lambda_e = \lambda_f = \lambda_r$$

$$\lambda_u = \lambda_t = \lambda_v$$

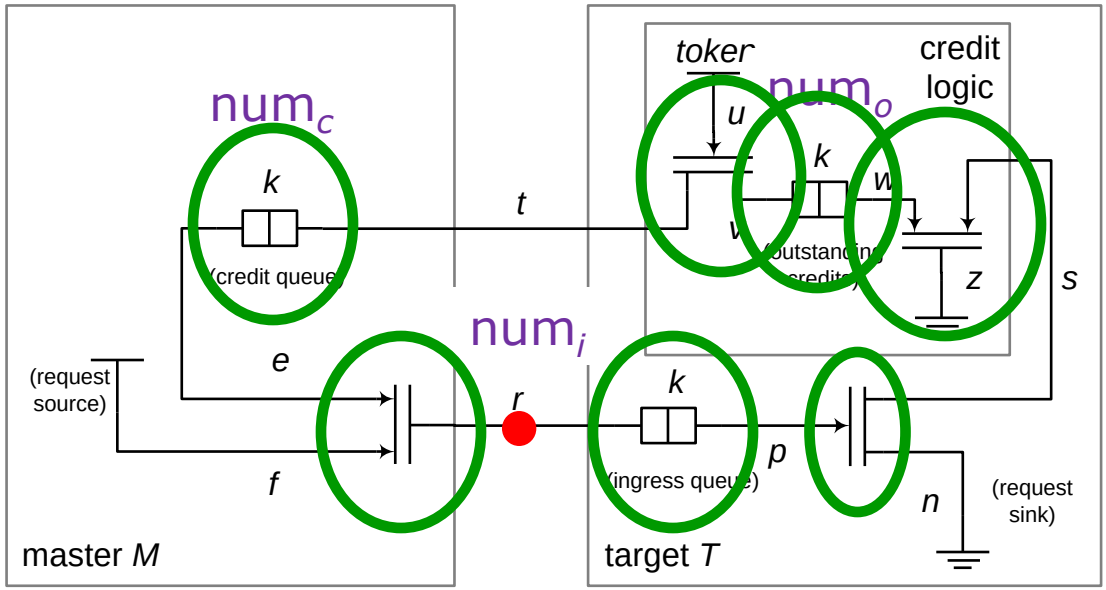
$$\lambda_s = \lambda_w = \lambda_z.$$

$$\lambda_p = \lambda_n = \lambda_s.$$

$$\lambda_r = \mathrm{num}_i + \lambda_p$$

$$\lambda_t = \mathrm{num}_c + \lambda_e$$

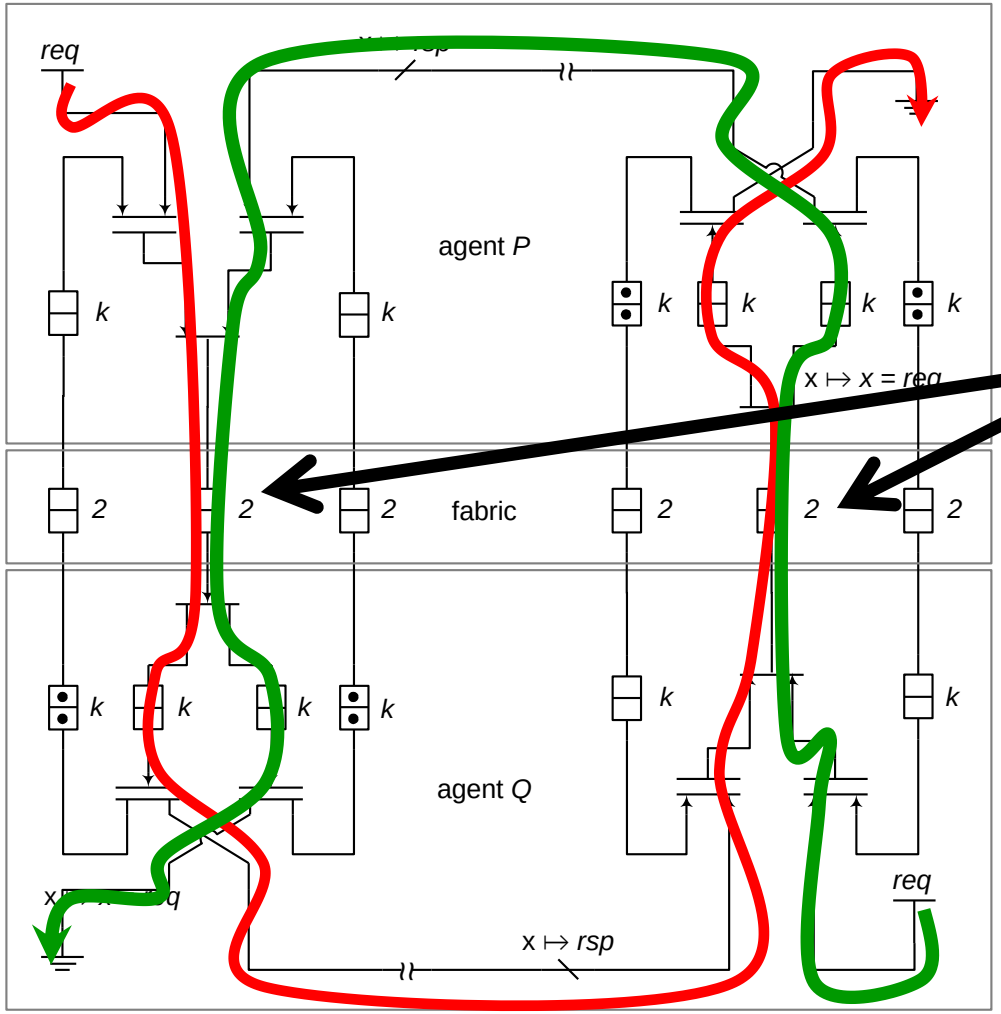
$$\lambda_v = \mathrm{num}_o + \lambda_w$$



Eliminate λs using modified Gaussian Elimination

$$\mathrm{num}_c + \mathrm{num}_i = \mathrm{num}_o$$

Need more precise analysis



- Have to track different *flows*

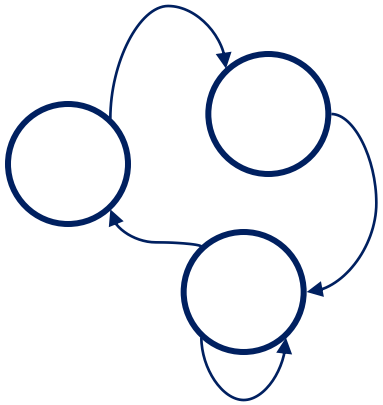
Need to track num_{red} and num_{green} separately for these two queues.

- See CAV10 paper for
  - Flow detection
  - invariants per flow

Outline

- How to capture “high-level” structure?
- How to exploit “high-level” structure?
  - Safety properties
  - Liveness properties

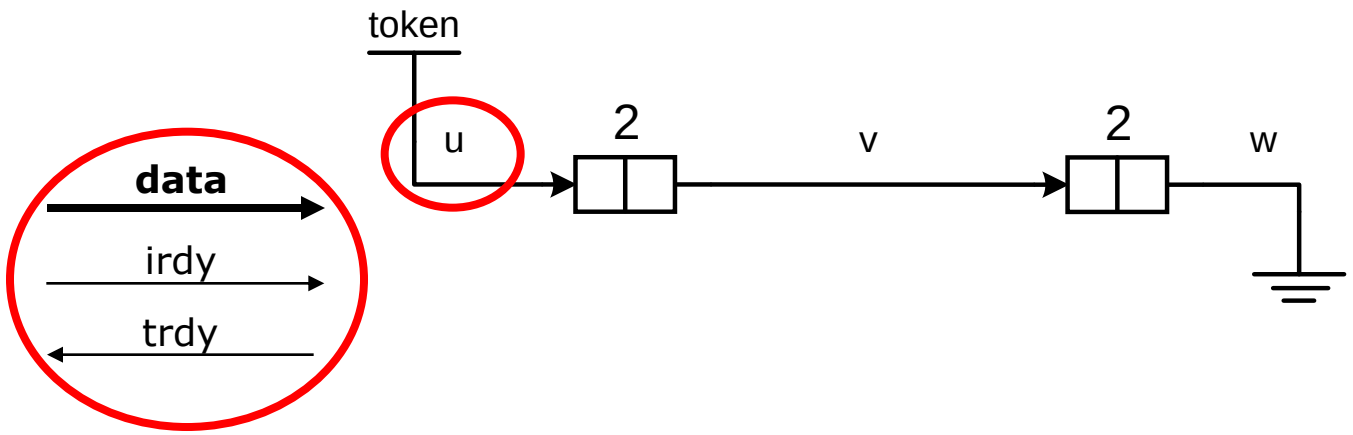
Linear Temporal Logic (LTL)



LTL is a formal language to reason about executions of a state machine in time

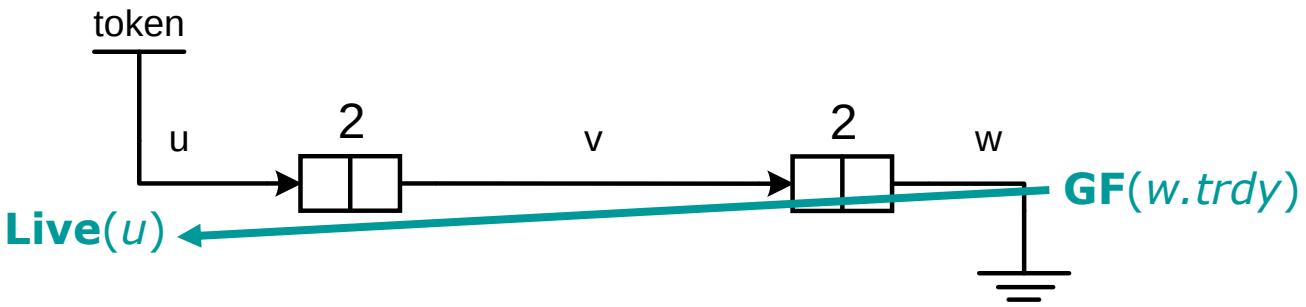
- Predicates
  - *(x < 7), (y-2 == z)*
- Propositional connectives
  - “·”, “+”, “→”, “¬”, …
- Temporal operators
  - **F** = “eventually”
  - **G** = “always”

Deadlock in xMAS



- Deadlock in xMAS is defined for a channel (hence local)
- **Dead**(u) = **F**($u.irdy$ · **G**¬$u.trdy$) "eventually a packet arrives at input of u, but output of u always blocked"
- **Live**(u) = ¬**Dead**(u) = **G**($u.irdy \rightarrow$ **F**$u.trdy$)

Fairness Constraints

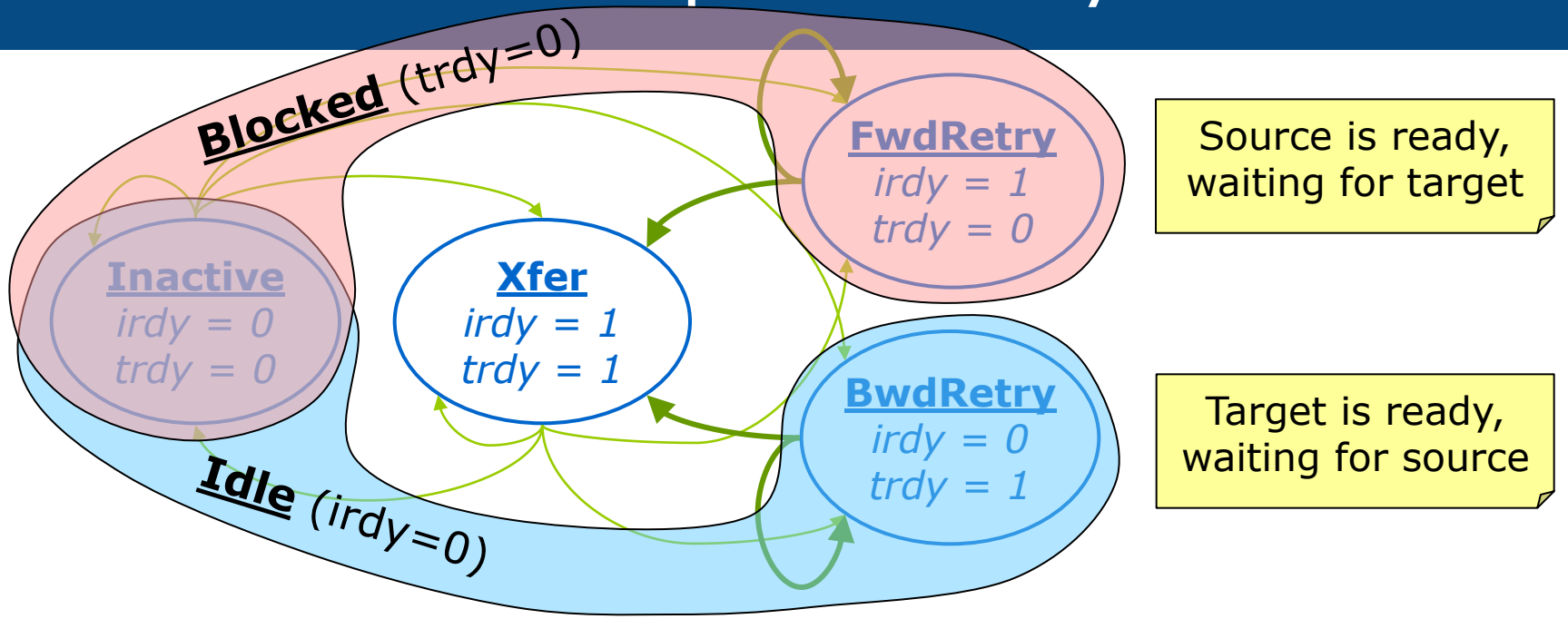


- Fairness constraint:

- Execution of xMAS model is fair if it satisfies all fairness constraints

- **Fair** = conjunction of all fairness constraints

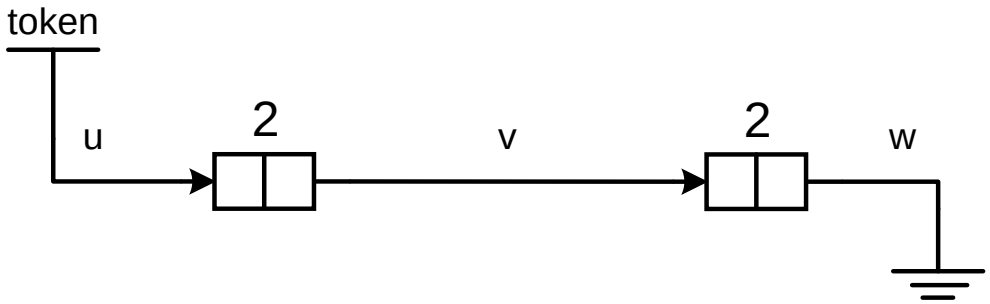
Channel persistency



- Every channel in xMAS model is persistent
  - Common xMAS primitives are designed to ensure this property
- Persistency guarantees that a handshake is never missed
  - Rules out some nasty livelock scenarios
- In practice, many micro-architectures are persistent

A retry state either persists or ends in transfer.

Idle and Block Stuck-at Conditions



Idle(u) = **FG**($\neg u.irdy$) "u (eventually) stuck at idle"

Block(u) = **FG**($\neg u.trdy$) "u (eventually) stuck at blocked"

For a **persistent** channel

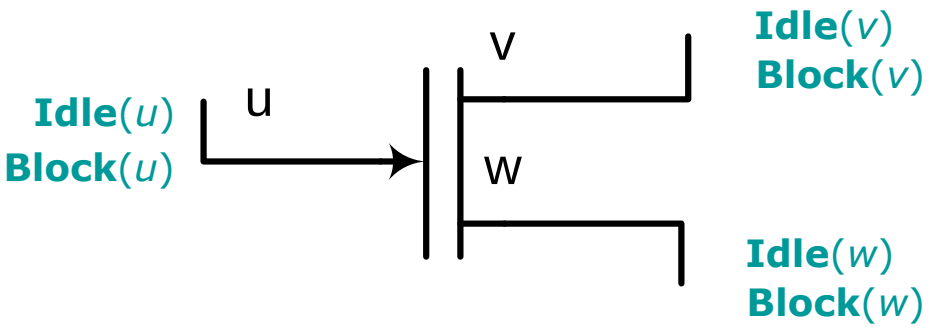
Dead(u) = $\neg$**Idle**(u) · **Block**(u)

"u is in deadlock iff u stuck at blocked, but not stuck at idle"

Fair = $\neg$**Block**(w)

"sink is fair iff w not stuck at blocked"

Equations for fork



$$\begin{cases} u.trdy = v.trdy \cdot w.trdy \\ v.irdy = u.irdy \cdot w.trdy \\ w.irdy = u.irdy \cdot v.trdy \end{cases}$$

Block(*u*) = **Block**(v) + **Block**(w)

"*u* stuck at blocked iff *v* or *w* stuck at blocked"

Idle(*v*) = **Idle**(*u*) + **Block**(w)

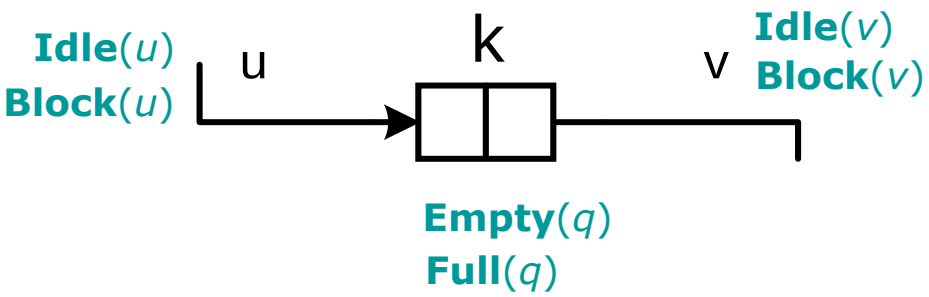
"*v* stuck at idle iff *u* stuck at idle or *w* stuck at blocked"

Idle(*w*) = **Idle**(*u*) + **Block**(v)

"*w* stuck at idle iff *u* stuck at idle or *v* stuck at blocked"

Equations captures relations between **Idle** and **Block** conditions on inputs and outputs of a fork

Equations for queue



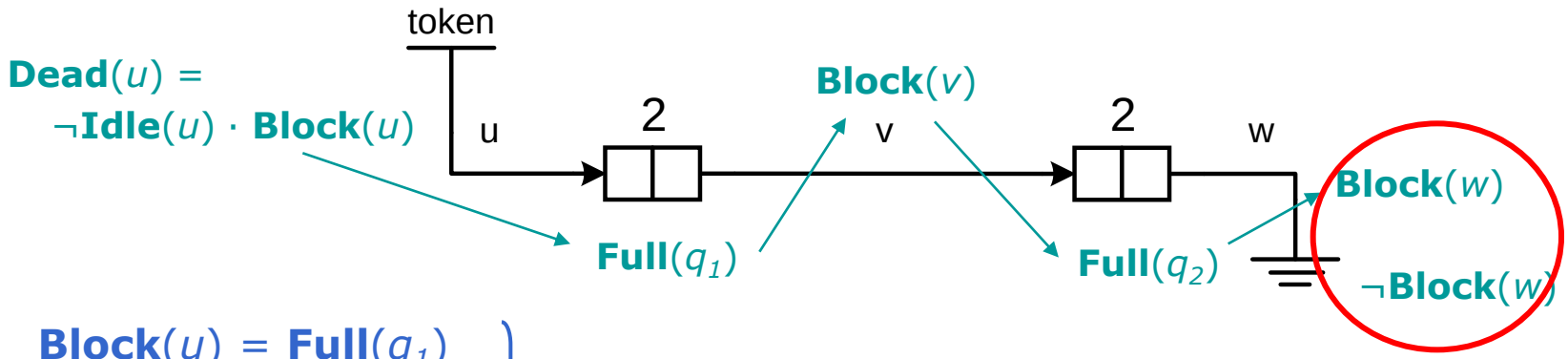
$$\begin{cases} u.trdy = (q.num < k) \\ v.irdy = (q.num > 0) \\ \dots \end{cases}$$

Block(u) = **Full**(q) = **FG**($q.num=k$) "q (eventually) stuck at full"

Idle(v) = **Empty**(q) = **FG**($q.num=0$) "q (eventually) stuck at empty"

Other equations: $\neg$**Block**(v) $\rightarrow$ $\neg$**Full**(q), **Empty**(q) $\rightarrow$ **Idle**(u),
$\neg$**Idle**(u) $\cdot$ **Block**(v) $\rightarrow$ **Full**(q), etc.

Liveness proof



Block(u) = **Full**(q_1)

Full(q_1) → **Block**(v)

Full(q_2) → **Block**(w)

DeadEq
(true for every execution)

If u is in a deadlock, then w should be stuck at blocked.

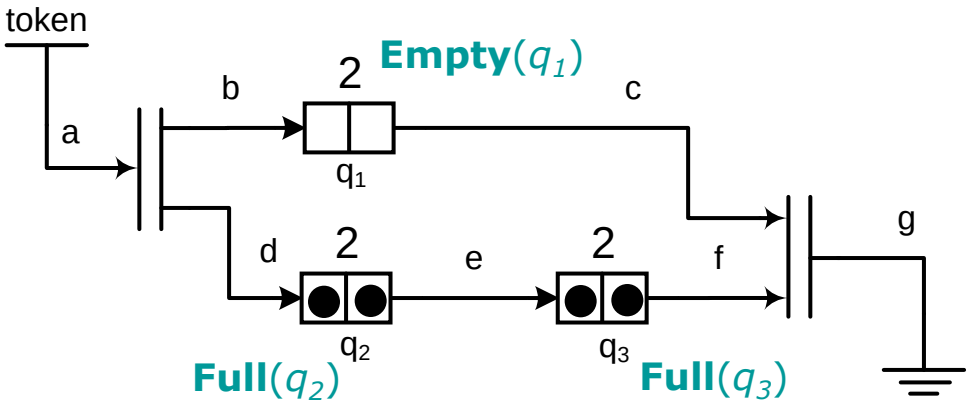
Fair = ¬**Block**(w)
(true for all fair executions)

Dead(u) · **DeadEq** · **Fair** = 0 in propositional logic

There is no fair execution of the model, where channel u is dead.

The above equation captures structural deadlocks
(reachability from initial state is ignored)

Adding invariants / 1



Dead(a)
Fair = ¬**Block**(g)

Dead(a) · **DeadEq** · **Fair** has a solution

- Solution is unreachable execution with q_1 stuck at empty and q_2, q_3 stuck at full
- The execution contradicts flow invariant (automatically generated):

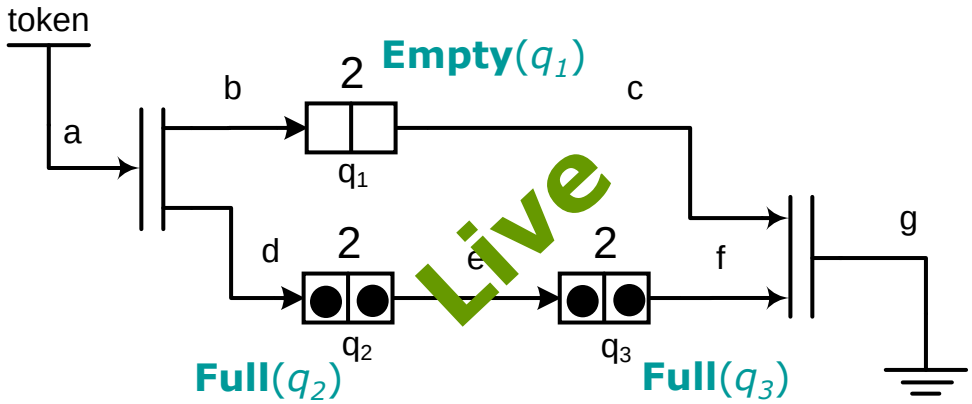
$$q_1.num = q_2.num + q_3.num$$

Deadlock equations – about executions

Flow invariants – about instantaneous state

?

Adding invariants / 2



Dead(a)
Fair = ¬**Block**(g)

- Add equations connecting **Idle**, **Block**, **Empty**, **Full**, etc. with instantaneous state of the model (*q.num*, *irdy*, *trdy*, etc.):

Empty(q) → ($q.num = 0$)
Full(q) → ($q.num = k$)
Idle(u) → ($u.irdy = 0$)
…

InfEq (eventually true for every execution)

- Add *any* instantaneous invariants (**Inv**), solve

Dead(u) · **DeadEq** · **Fair** · **Inv** · **InfEq** for propositional satisfiability

UNSAT

Method

Dead(u)

As ¬**Idle** · **Block**

+

DeadEq
(per primitive)

LTL theorems, describing how **Idle** and **Block** propagate through primitives

+

Fair

As ¬**Block**

- **UNSAT** = Liveness proof
- **SAT** = Counterexample

+

InfEq

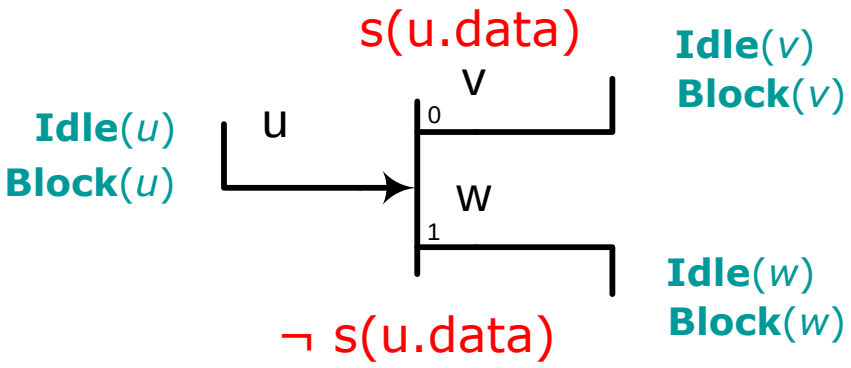
Connect **Idle**, **Block**, etc. with instantaneous state

+

Inv

Automatically generated instantaneous invariants

Data dependencies / 1



$$\begin{cases} u.trdy = u.irdy \cdot (s(u.data) \cdot v.trdy + \\ \qquad + \neg\, s(u.data) \cdot w.trdy) \\ v.irdy = u.irdy \cdot \quad s(u.data) \\ w.irdy = u.irdy \cdot \neg\, s(u.data) \end{cases}$$

Idle(v) depends on what kind of data is coming from input channel u

Idle(v) = **Idle**(u) + "all packets are going to the bottom branch"

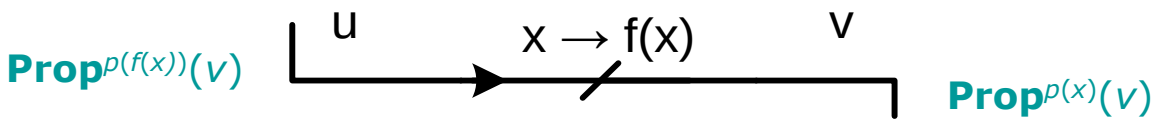
Prop$^{p(x)}$(u) = **FG**($u.irdy \rightarrow p(u.data)$) "$u$ (eventually) stuck at $p(x)$"

Idle(v) = **Idle**(u) + **Prop**$^{\neg s(x)}$(u)

"v stuck at idle, iff u stuck at idle or u stuck at $\neg s(x)$"

- **Prop** conditions in addition to **Idle** and **Block**, etc.
- Add equations to compute **Prop** conditions

Data dependencies / 2



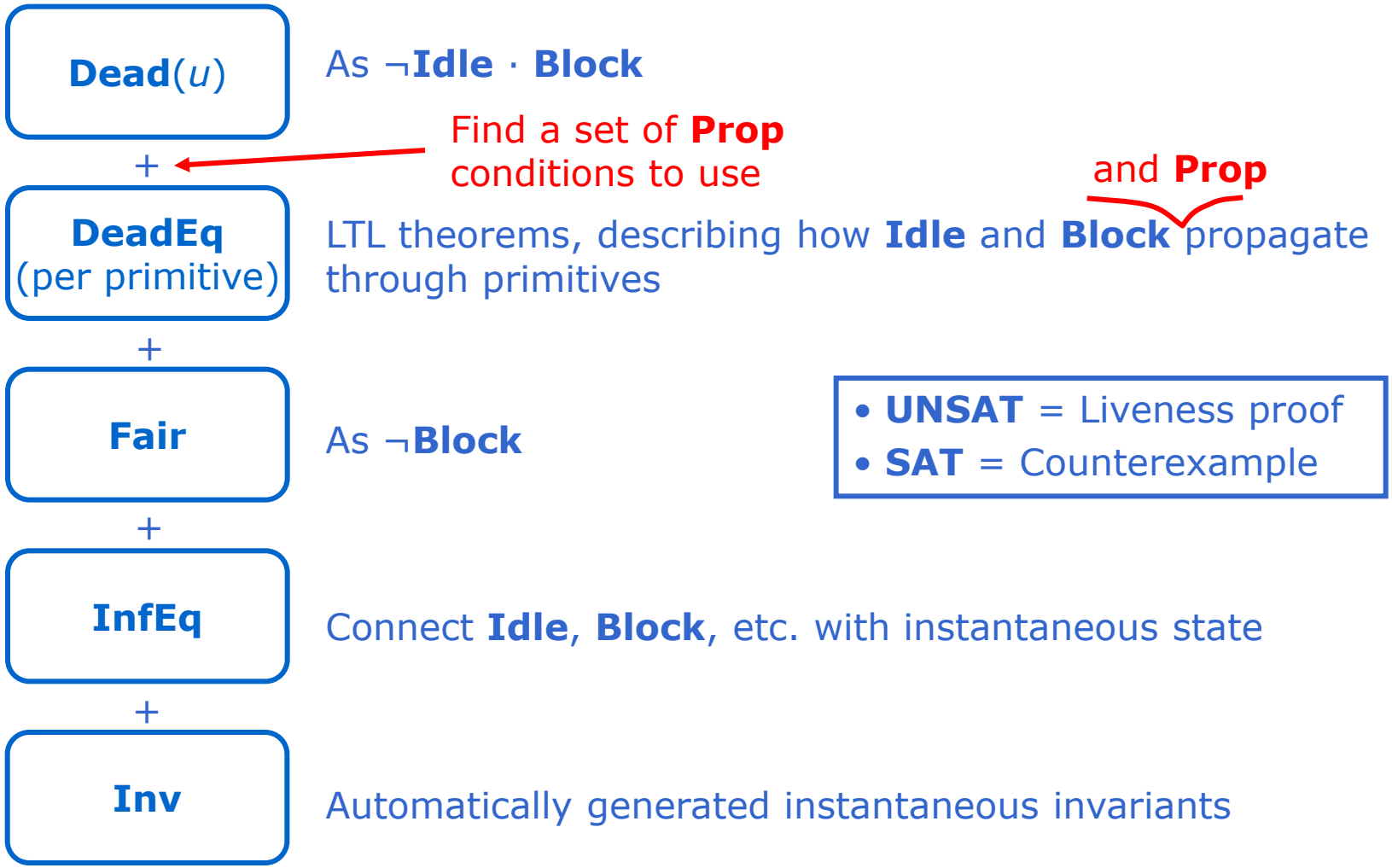
Prop$^{p(x)}(v)$ = **Prop**$^{p(f(x))}(u)$

"v stuck at $p(x)$ iff u stuck at $p(f(x))$"

Number of all possible **Prop** conditions is
(number of channels) x (average number of predicates per channel).

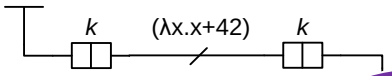
Linear from size of the model.
May be exponential from width of data on channels, but explosion is avoidable in practice

Accounting for Data dependencies



Modeling and Verification Flow

(from architects)



Formal Analysis
Invariant generation
Deadlock equations
etc.

C++

compile

exec-utable

run

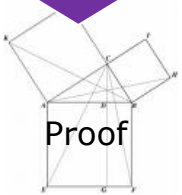
random test-bench.v

model.v + SVA

abc

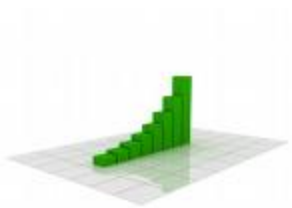
Model Checker

To make this easier or to prove without it



xMAS API (Library)

simulate (RTL Simulator)



Experimental results: on-die IO fabric

- Size of xMAS model: 775 primitives; 131 queues
- Proving liveness of all channels of interest
  - Generate 271 global invariants - 2 sec
  - Generate 4361 deadlock equations – 7 sec
  - Solving SAT formulae with ABC (fraig!) - 88 sec
  - Total time to prove liveness – 97 sec
- Paranoid mode:
  - Proving that all invariants and other safety properties hold (17K+ properties)
  - 1-step induction in ABC – 16 hours

  Other methods:
  - PDR* and interpolation does not converge in a month
  - BMC – saturates at 16 frames in a week

* Aaron Bradley's VMCAI-2011 algorithm implemented with improvements in ABC by Alan Mishchenko, Niklas Een and Bob Brayton – FMCAD 2011

Experimental results: finding a bug

Experiment	NPrims (NQueues)	Time for ABC (BMC)	Time for our method
2 Master/Slave agents (no credit logic)	16 (2)	6s	<1s
Simple fabric: agent + router (unfair sink)	57 (20)	4s	2s
Simple fabric: agent + router (broken credits logic)	57 (20)	8s	2s
Simple fabric: agent + router, message ordering (unfair sink)	75 (24)	1hr	3s
Simple fabric: agent + router, message ordering (broken ordering logic)	75 (24)	4hr	3s

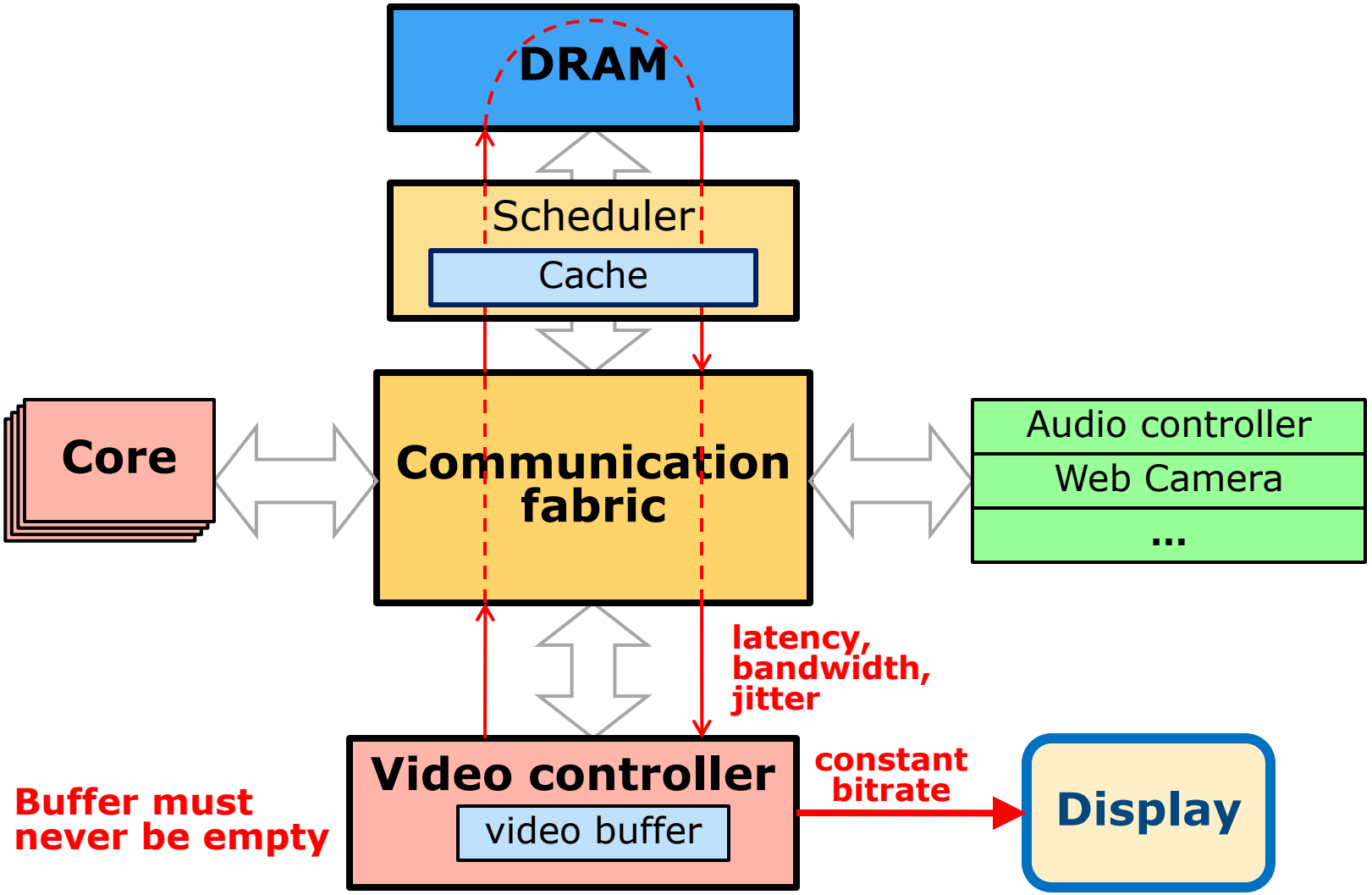
Deadlocks found with our method can be unreachable (due to over-approximation)

In experiments, all found deadlocks were reachable.

Outline

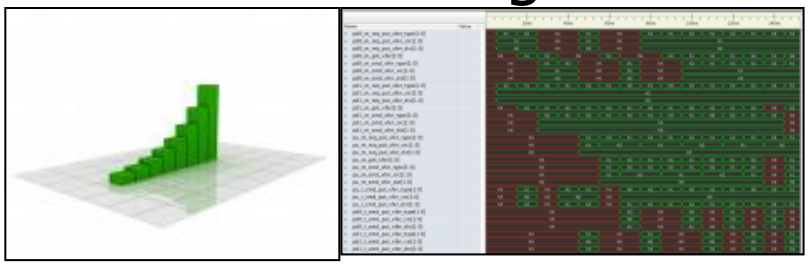
- How to capture “high-level” structure?
- How to exploit “high-level” structure?
  - Safety properties
  - Liveness properties
  - Quality of Service

QoS in System on Chip

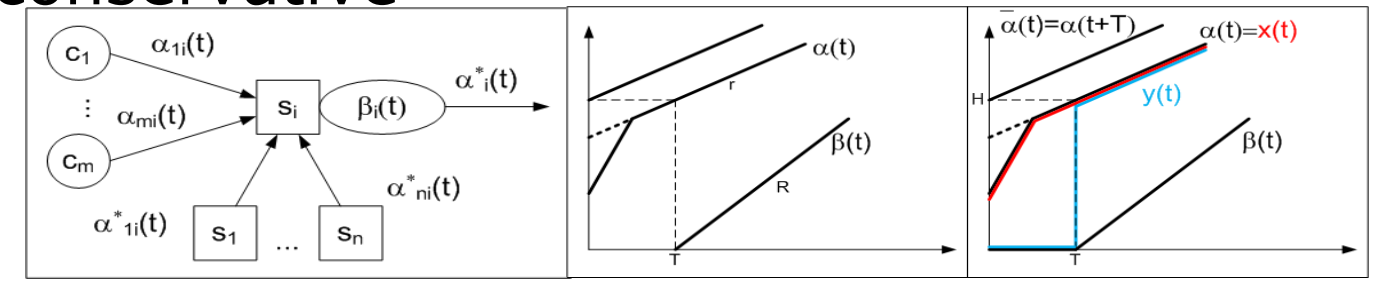


Alternatives for validating QoS

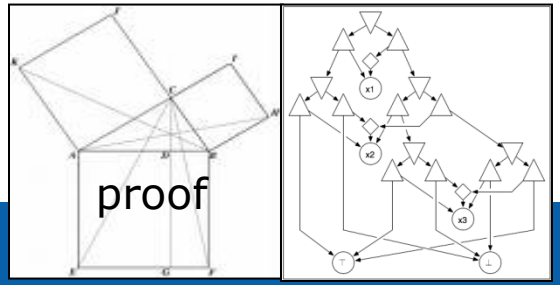
- Simulation – no guarantees



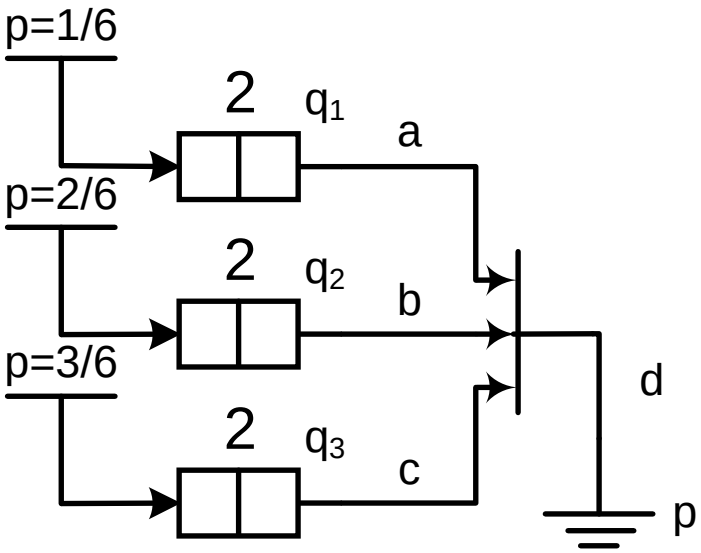
- Analytical methods (e.g. Network Calculus) – too conservative



- Formal verification – does not scale



Toy QoS example



QoS Metrics

- Transaction end-to-end latencies
  - and phases of transactions
- Bandwidth of channels
- Queue utilization
- Worst case and average

Refine xMAS primitives:

arbitration algorithm,
traffic shapers (source rates, sink rates), etc.

Other challenges: micro-architectural verification

- Handling non-restricted joins
- Inductive proofs for QoS
- Ordering
- Protocol verification (e.g. cache coherency) and uarch verification are currently separate
  - Protocol verification abstracts resources
  - Uarch verification abstracts data
  - Can we do them together on the same model?
  - Or is it better to keep a screwdriver and a hammer separate?
  - Protocol flows must be taken into account
- Need fast methods for large models (hence inductive automatic proofs)

Other challenges: links to RTL

- Micro-architectural verification often falls between the cracks
  - Architects are not trained and rely exclusively on intuition (which surprisingly often works, except when it does not) and performance models
  - Validators are responsible for validating RTL, where proving liveness and complex safety properties is intractable
- Generate assertions for RTL
  - Easy on standard interfaces, but not internal to uarchitecture
- RTL and assertion generation from HLM
  - Different classes of fabrics require specialized generators

Other challenges: exploration and optimization

- Even more important than verification
  - We can ship a product without *formally* verifying it
  - We cannot ship a product without deciding on micro-architectural features
- Selection of micro-architectures
- Sizing resources (queues, credits, links,…)
- But this is another talk

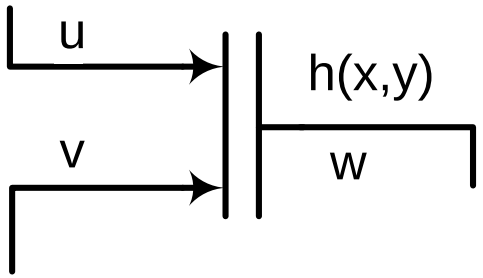
Q&A

State of the art

- ABC (liveness-to-safety translation)
  - Reduces liveness problem to equivalent safety problem by circuit transformation
    - Doubles number of flops
  - BMC can reveal "shallow" deadlocks
  - Induction, interpolation, etc.
  - Does not scale well to xMAS models with 10s of queues
- NuSMV (LTL model checkers)
  - Consistently worse than ABC
- Theorem proving
  - Requires expertise and manual work

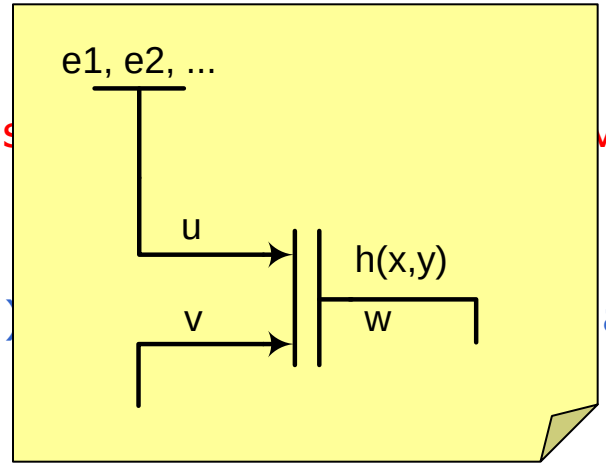
Non-restricted join is a problem

A = "$p(h(x,y))$ holds for any pair of aligned transfers on u and v"



Prop$^{p(z)}(w)$

- Join is non-restricted if h(x,y) depends on
- "*A*" cannot be precisely expressed in terms variables
- Can handle non-restricted join, if $p(h(x,y))$ and y
- Can handle "non-deterministic functions"



- Can over approximate output of non-restricted join with non-deterministic function

Simulation of toy QoS example