Hardware Verification 2IMF20

Julien Schmaltz

Lecture 06:
Verification of on-chip communication networks

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Living a revolution

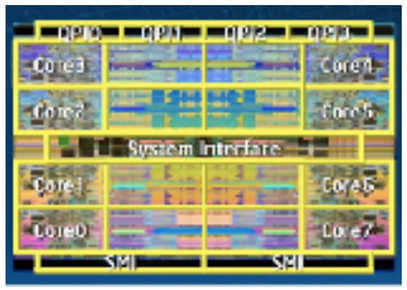
"It is a really exiting time to be in computing right now, because we are [experiencing] a major change. A change that happens maybe once every 20 or 30 years ... a fundamental re-thinking of ... what computation is"

William Dally, DAC keynote 2009

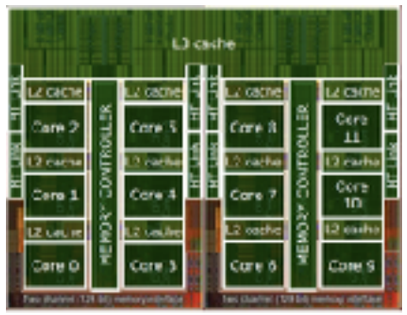
Multi-core shift reality

Growing number of cores

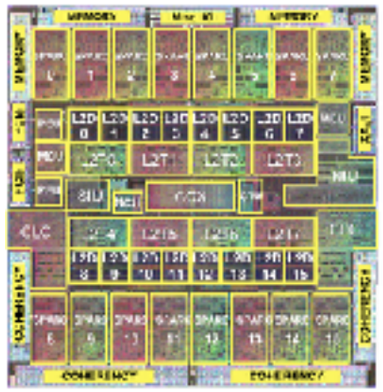
Intel 8 cores
~2.3 Bill. T. on 6.8cm²



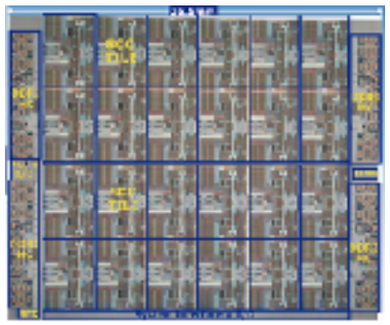
AMD Opteron 12 cores
~1.8 Bill. T. on 2x3.46cm²



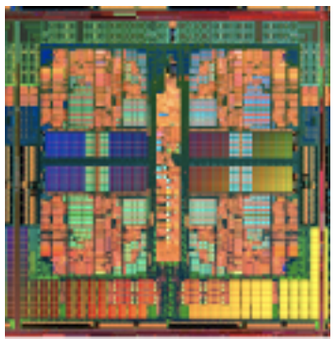
Sun Niagara3 16 cores
~1 Bill. T. on 3.7cm²



Intel SCC 48 cores
~1.3 Bill. T. on 5.6cm²

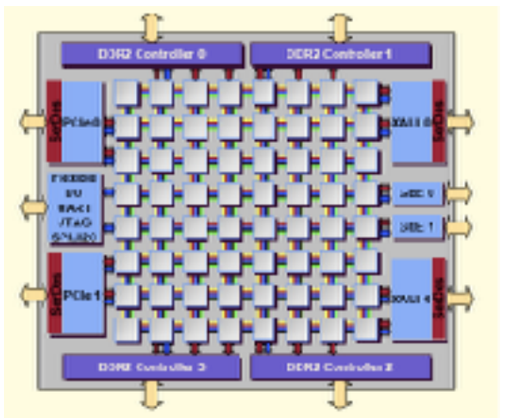


Intel 4 cores
~582 Mio. T. on 2.86cm²

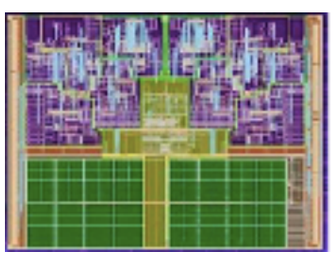


Intel Research 80 cores
~100 Mio. T. on 2.75cm²

Tilera TILEPro64 64 cores

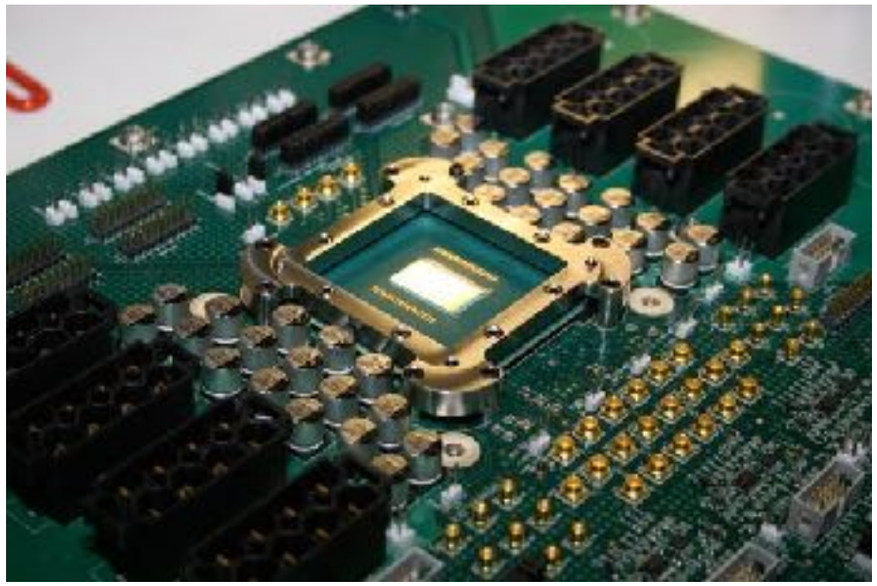


Intel 2 cores
~167 Mio. T. on 1.1cm²



80 Cores Research Chip (Intel)

- Teraflops, 62 Watts
- 100 millions transistors, 275 mm2
- 25% node area for router



- ASCI Red Supercomputer
- Teraflops (Dec. 1996)
- 10, 000 Pentium Pro
- 104 cabinets, 230 m2

It is only the beginning …

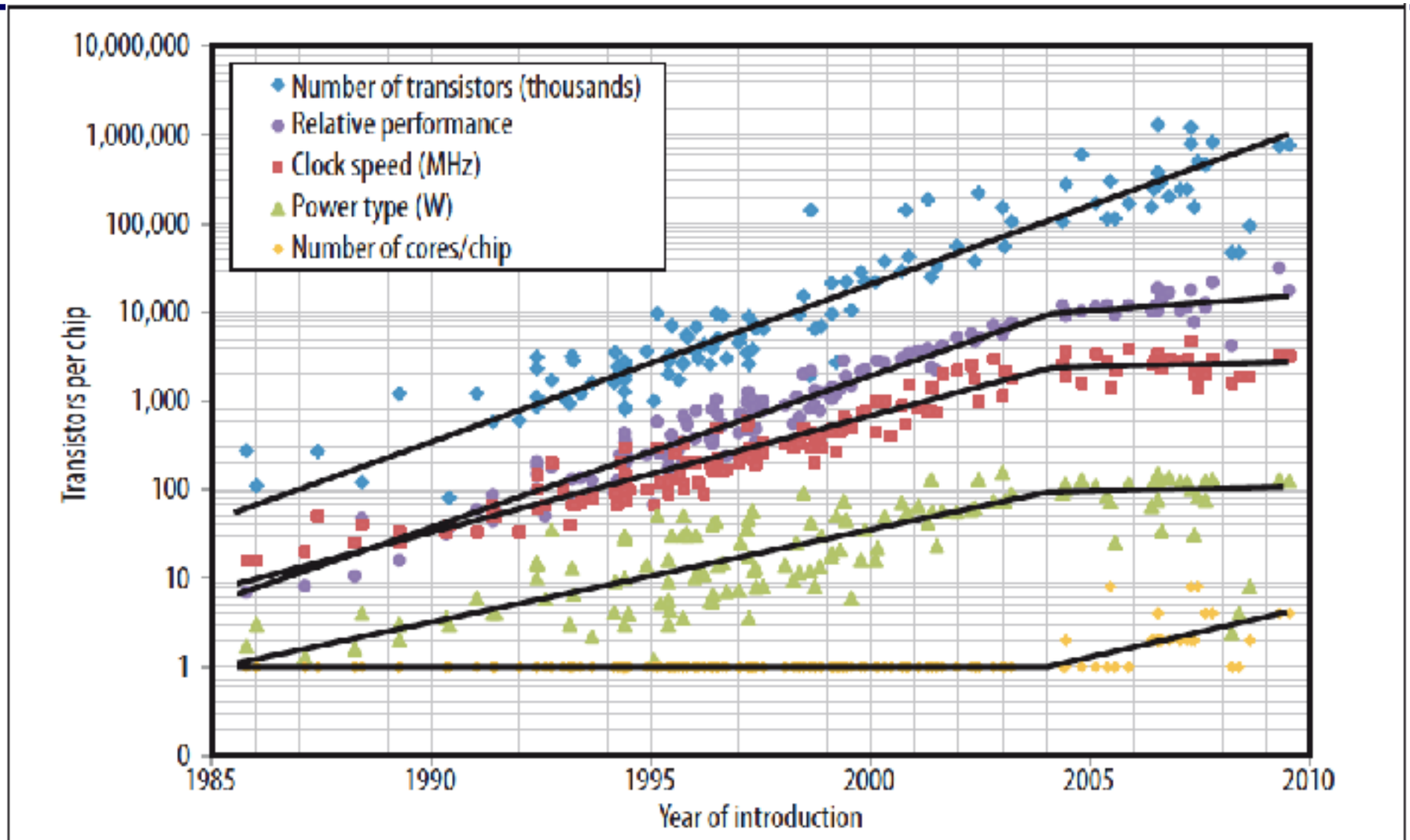


Figure 1. Transistors, frequency, power, performance, and processor cores over time. The original Moore's law projection of increasing transistors per chip remains unabated even as performance has stalled.

Source. *IEEE Computers* 2011

Communications are key

- 80 core research chip
  - –about 25% area for the NoC
  - –about 30% power consumption for the NoC

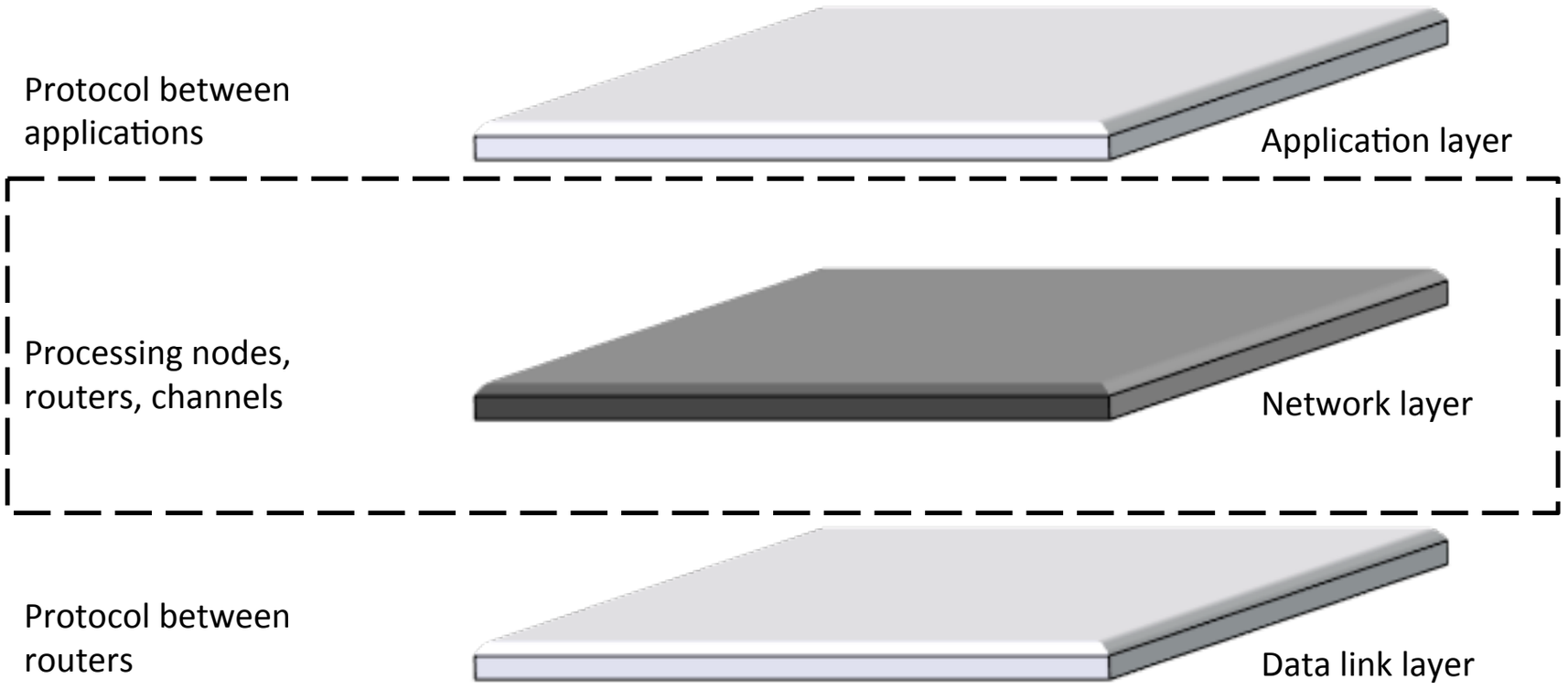
- Communication fabrics key
  - –to performance and efficiency
  - –to functional correctness

Verification Challenges

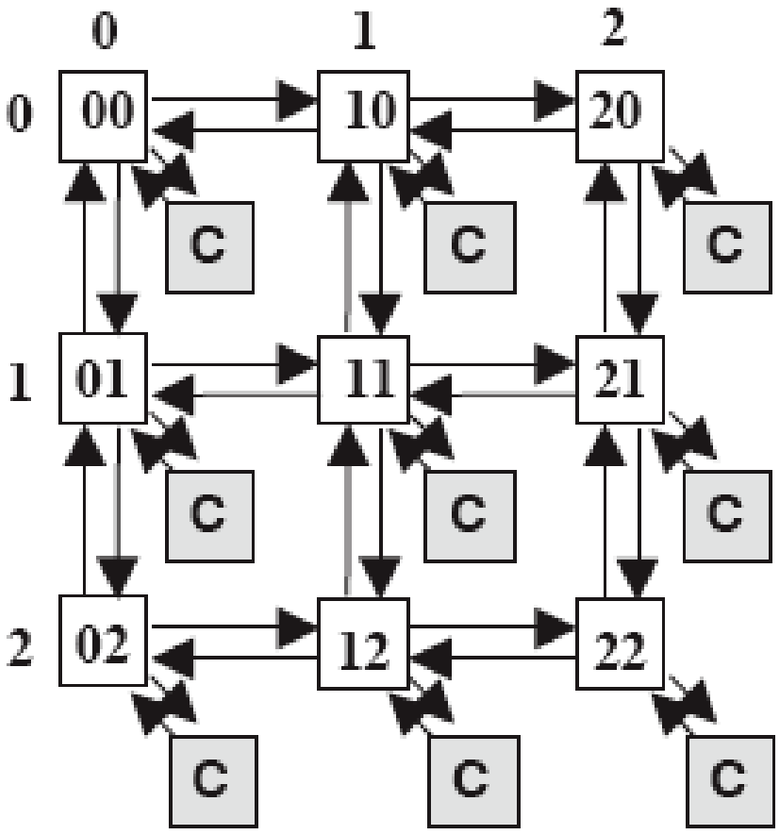
- NoCs are very large systems
  - –Methods must scale up to 100s of agents
  - –Large number of parameters (routing, switching, buffers, etc.)
  - –Regular and irregular structures

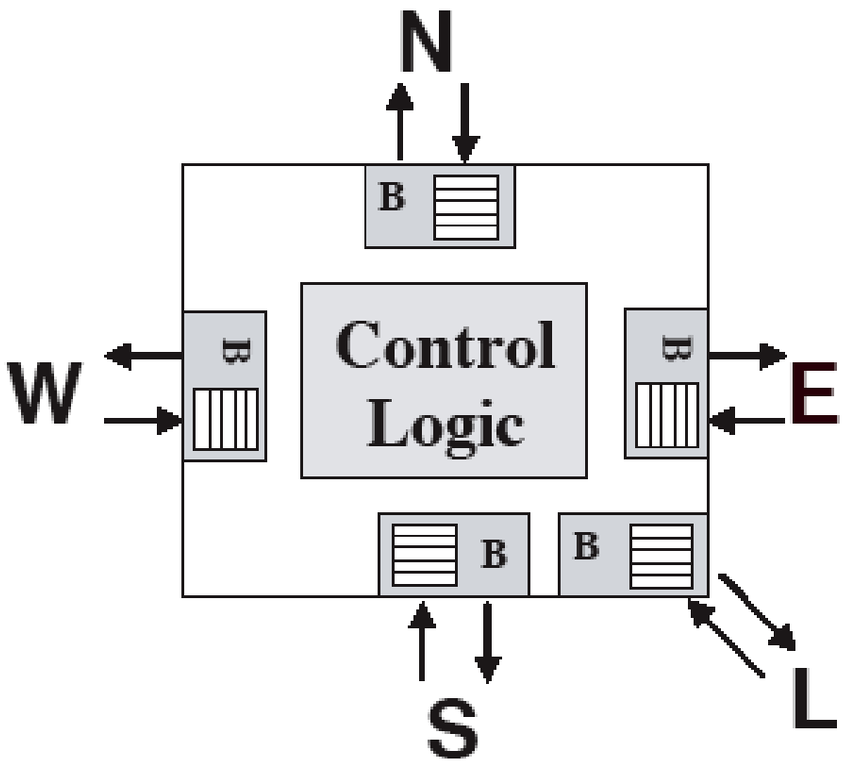
- NoCs must be fault-tolerant
  - –Deep sub-micron effect
  - –Not all routers/processors are working
  - –Static and dynamic fault-models

- NoCs have intricate message dependencies
  - –Mix between interconnect and protocols
  - –e.g. cache coherency or master/slave
  - –Deadlocks can emerge from deadlock-free routing and protocols


Protocol between applications
Application layer
Processing nodes, routers, channels
Network layer
Protocol between routers
Data link layer

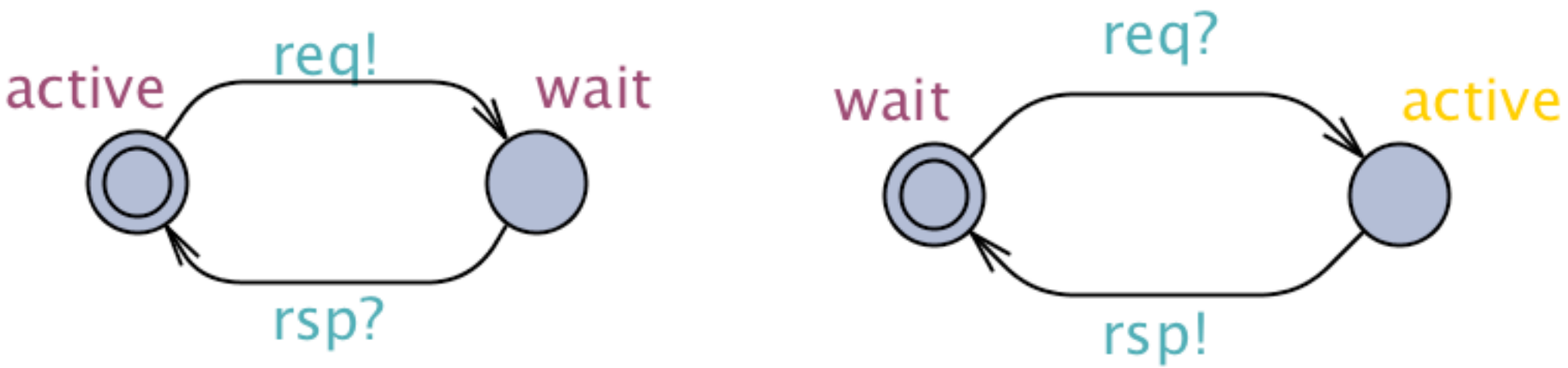
NoC Example: Hermes





Application layer

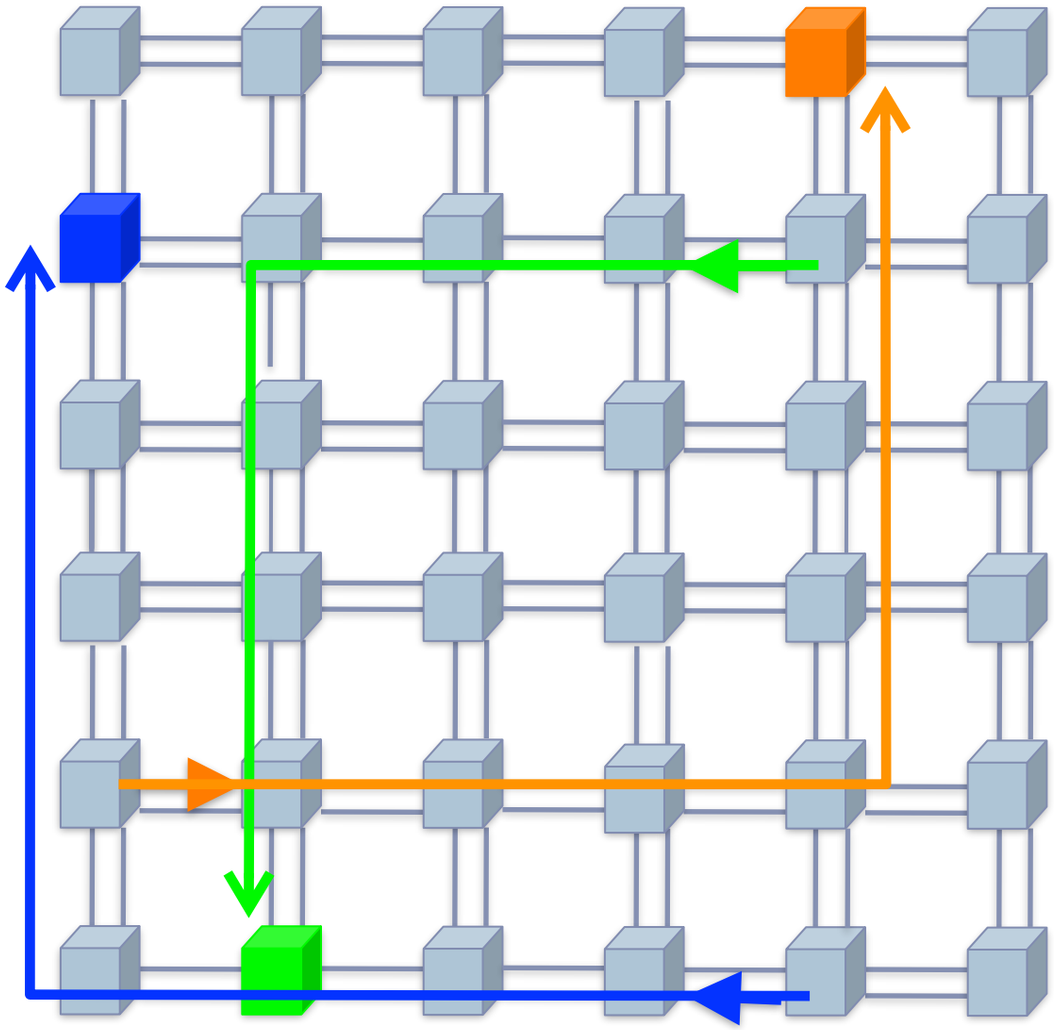
- Masters send requests and wait for responses
- Slaves produce responses when receiving requests
- Deadlock-free protocol



No cyclic message dependencies:

Network layer

- Deterministic simple routing algorithm
- First route to the destination column and then to the correct row
- No cyclic dependencies and thus deadlock-free

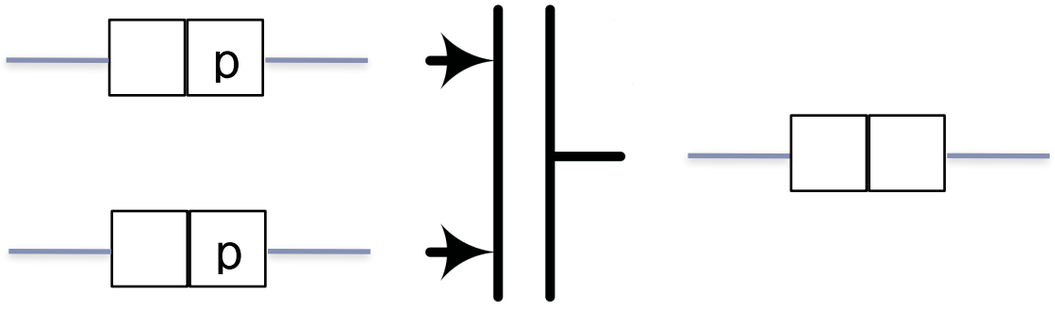


Link layer

- A packet moves if the next channel has free space

- A packet moves if it has sufficiently credits

- A packet is joined with another packet



Deadlock-free?

Deadlock-free application layer

\+ Deadlock-free network layer

\+ Deadlock-free link layer

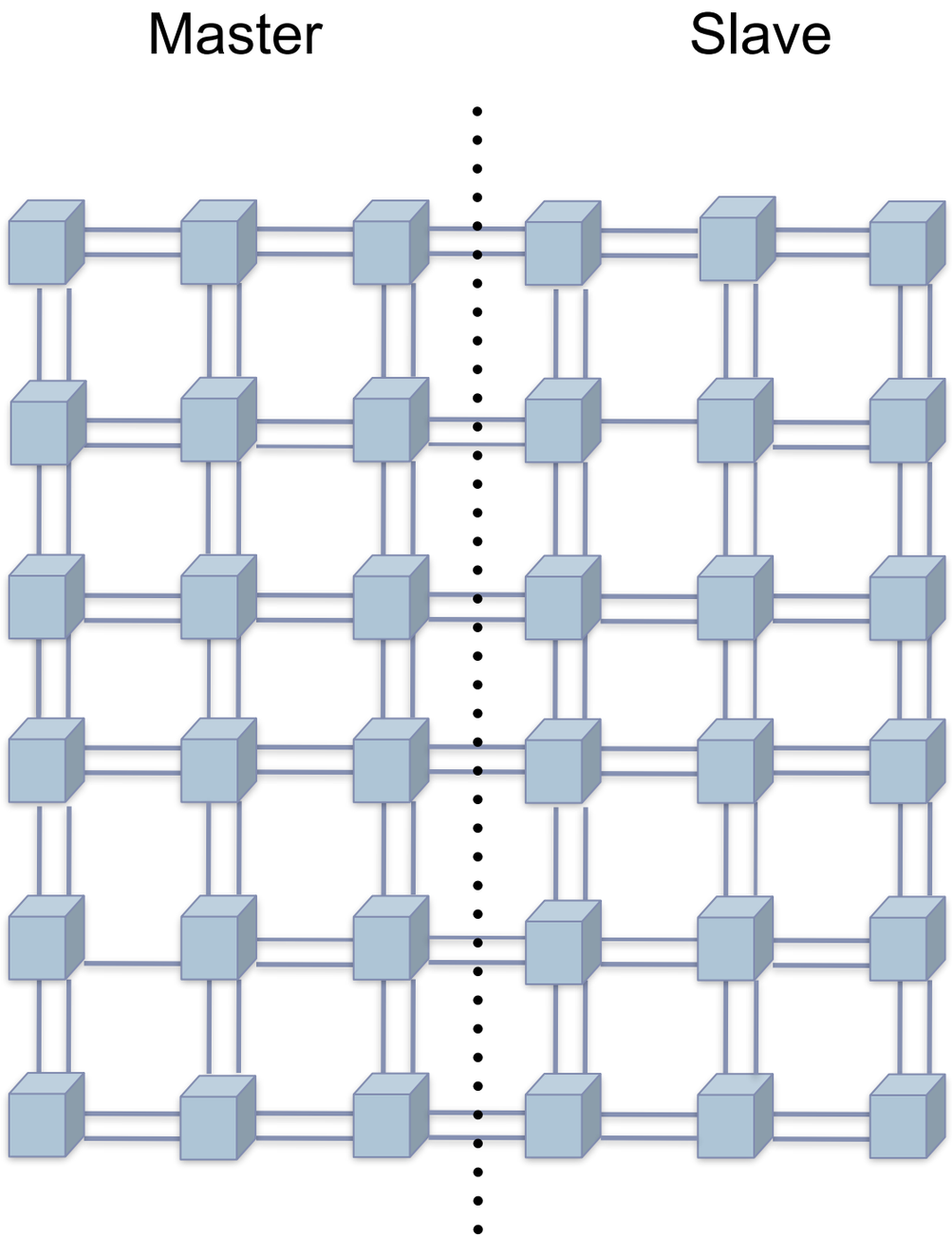
?

=

Deadlock-free system

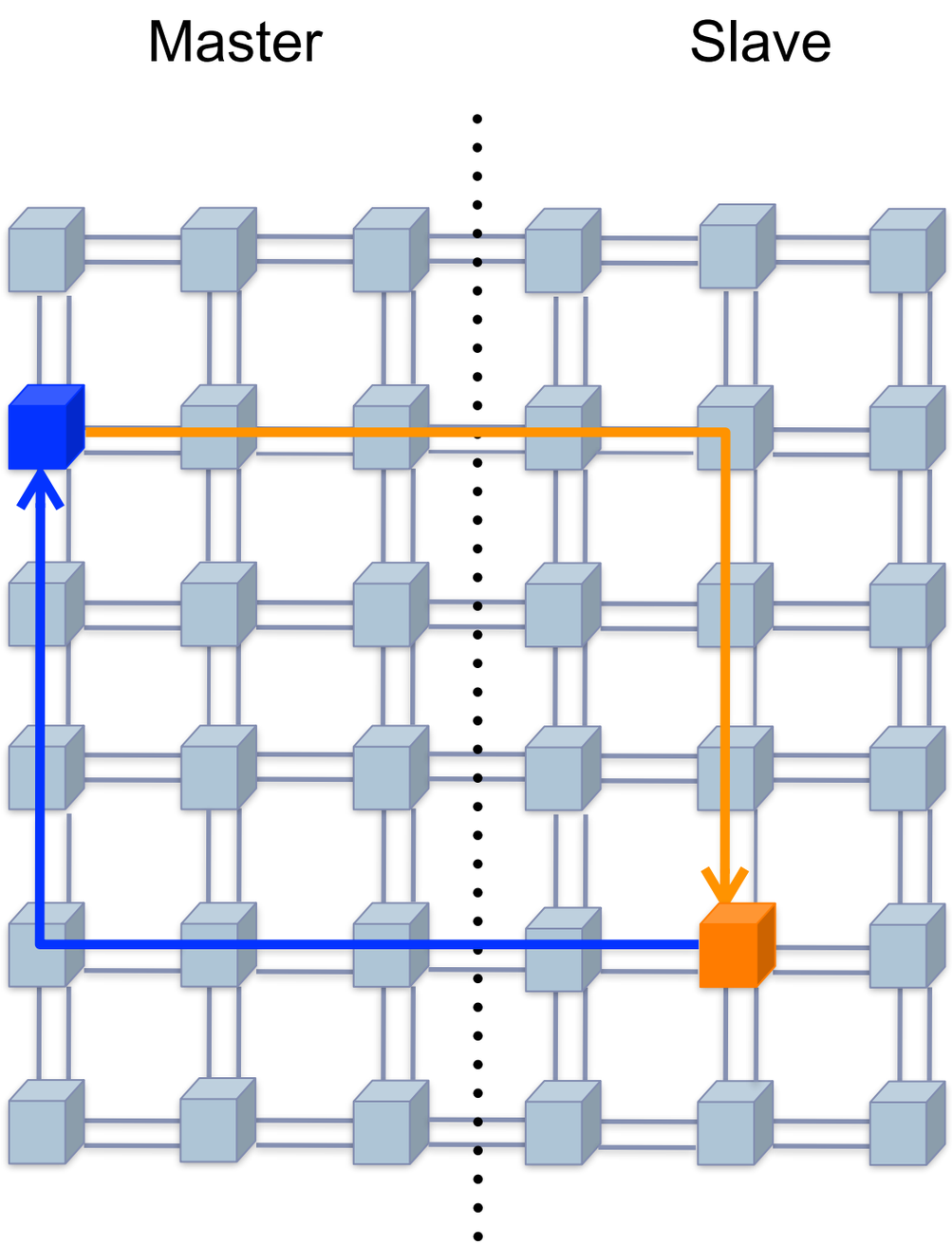
All masters right, slaves left

» Is the system deadlock-free ?



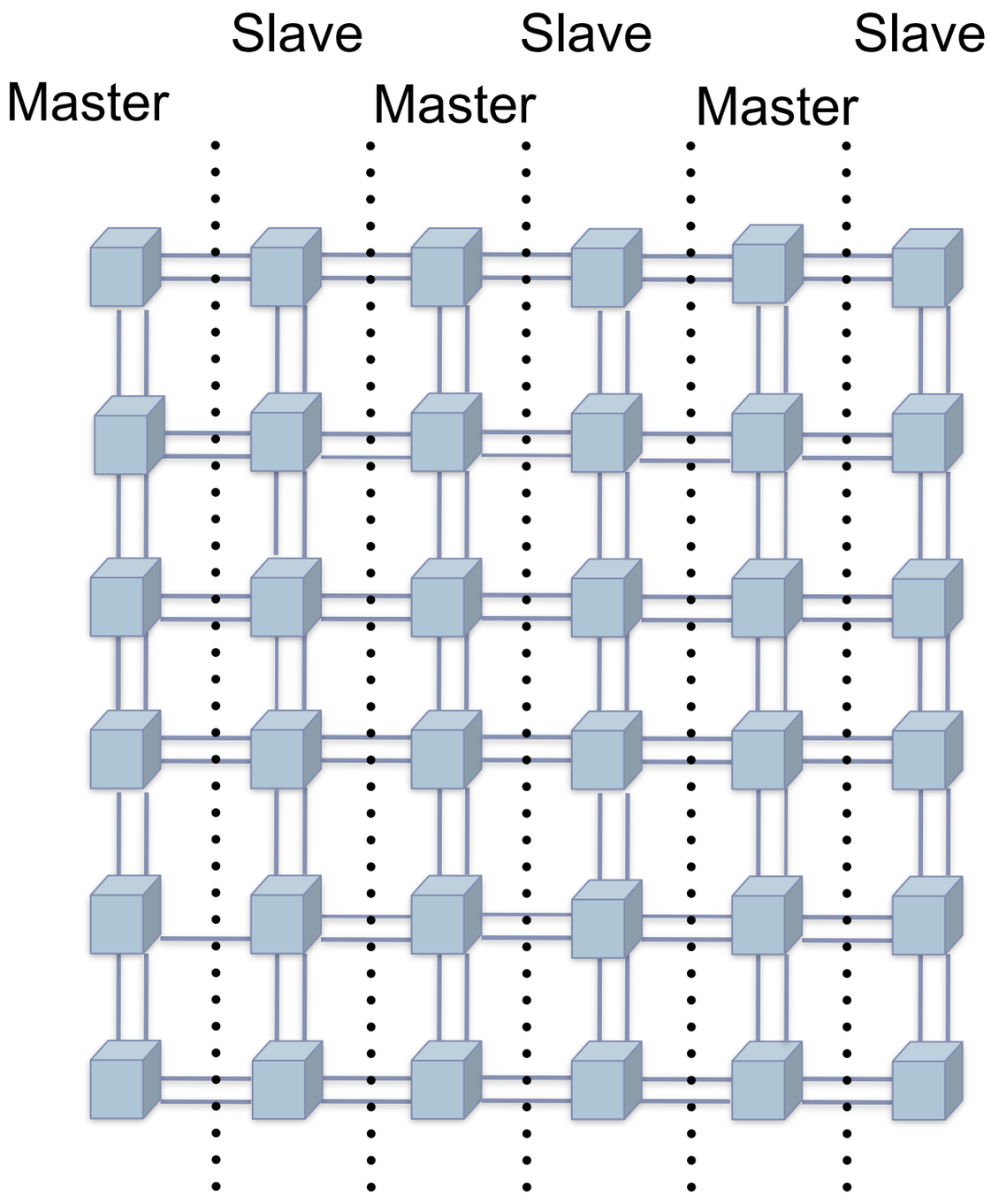
All masters right, slaves left

» Is the system deadlock-free ?
» Yes ! A deadlock would require a response to wait for a request



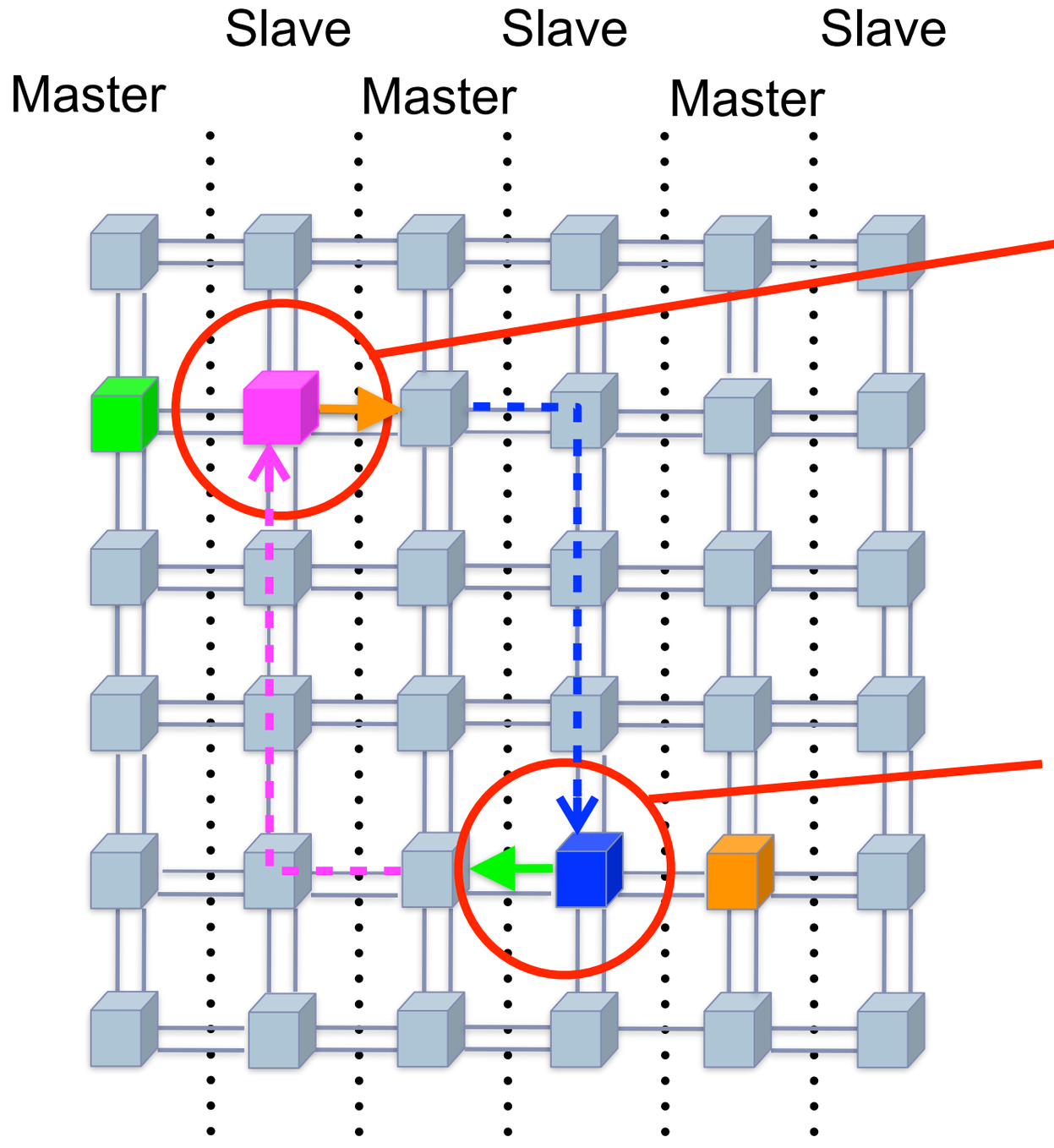
Masters even columns, slaves odd

» Is the system deadlock-free ?



Masters even columns, slaves odd

» Is the system deadlock-free ?
» No if at least four columns, yes otherwise.

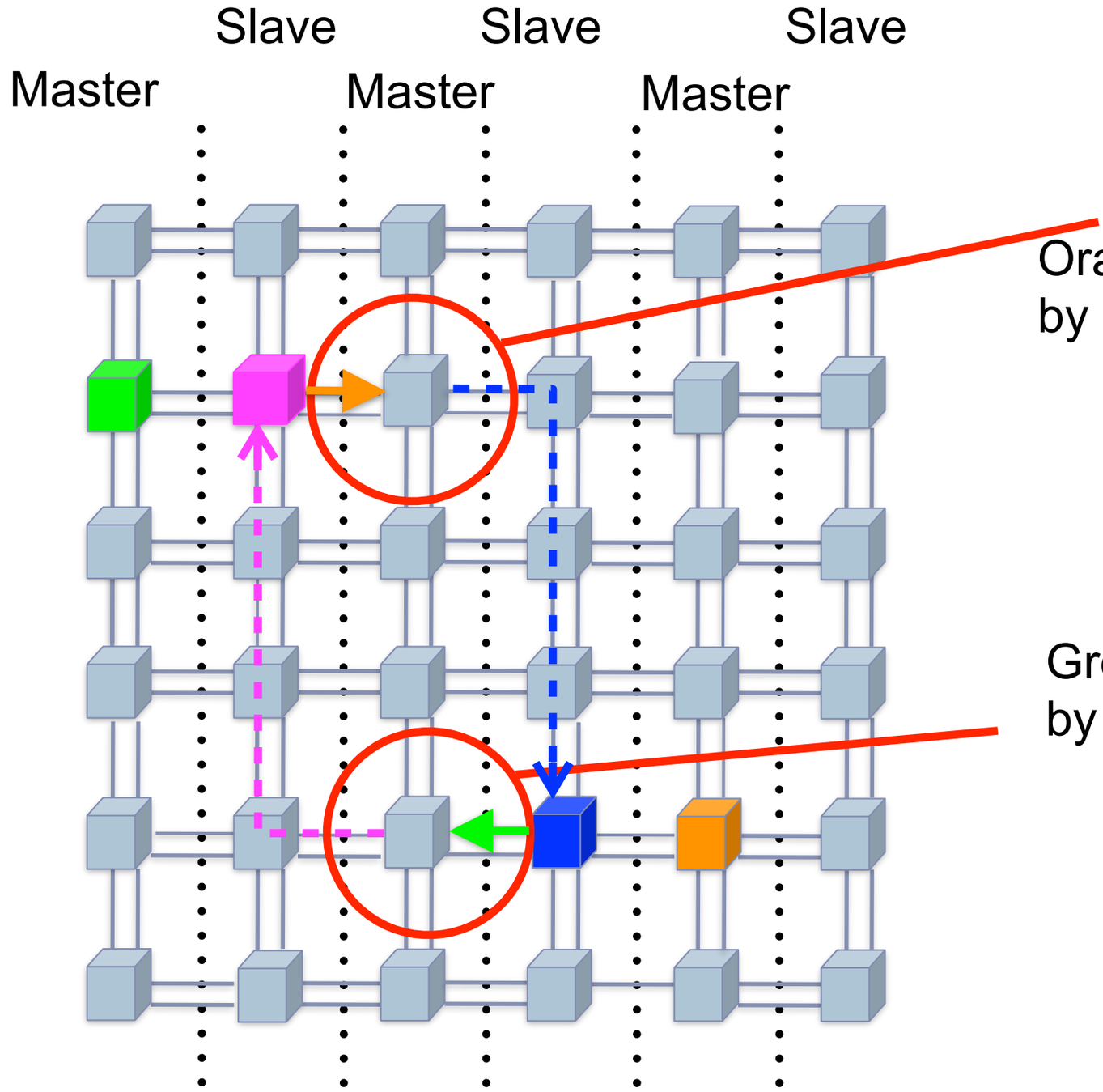


Purple requests are transformed into orange responses

Blue requests are transformed into green responses

Masters even columns, slaves odd

» Is the system deadlock-free ?
» No if at least four columns, yes otherwise.



Orange responses blocked by blue requests

Green responses blocked by purple requests

Deadlock-free application layer

\+ Deadlock-free network layer

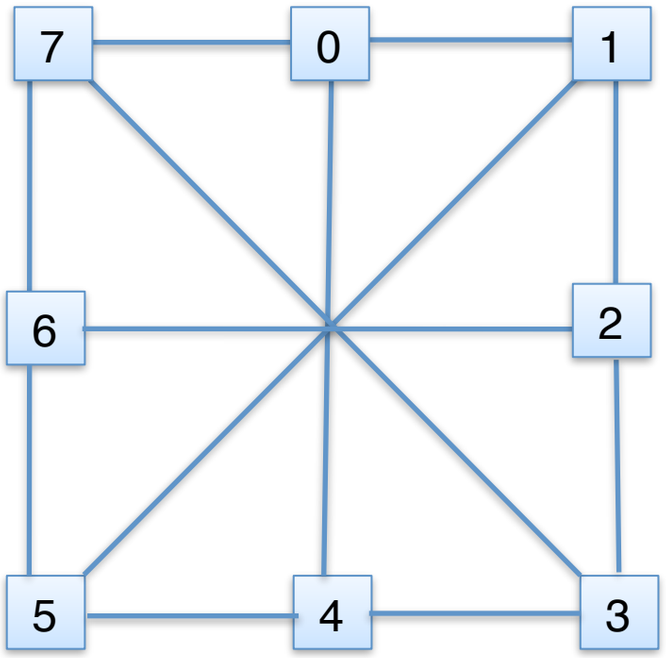
\+ Deadlock-free link layer

?

=

Deadlock-free system

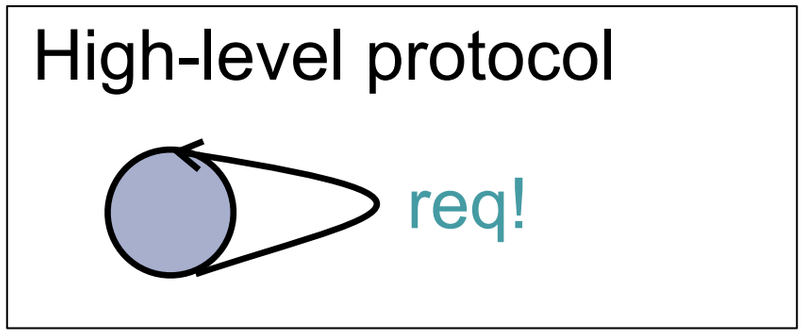
NoC Example (2) - Spidergon



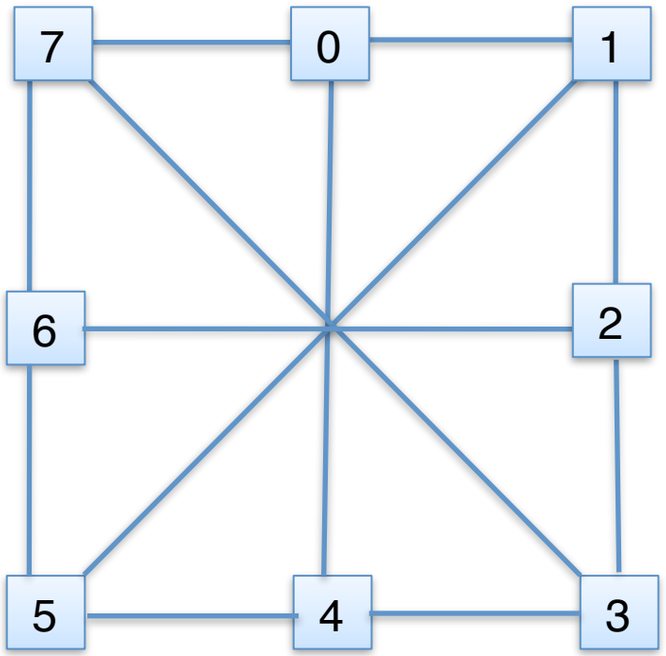
- Design by STMicroelectronics

Application layer

» Nodes send requests



Network layer



Routing logic

RelAd = (dest - current) mod 4 * N
if RelAd = 0 **then**
    stop

elseif 0 < RelAd <= N **then**
    go clockwise

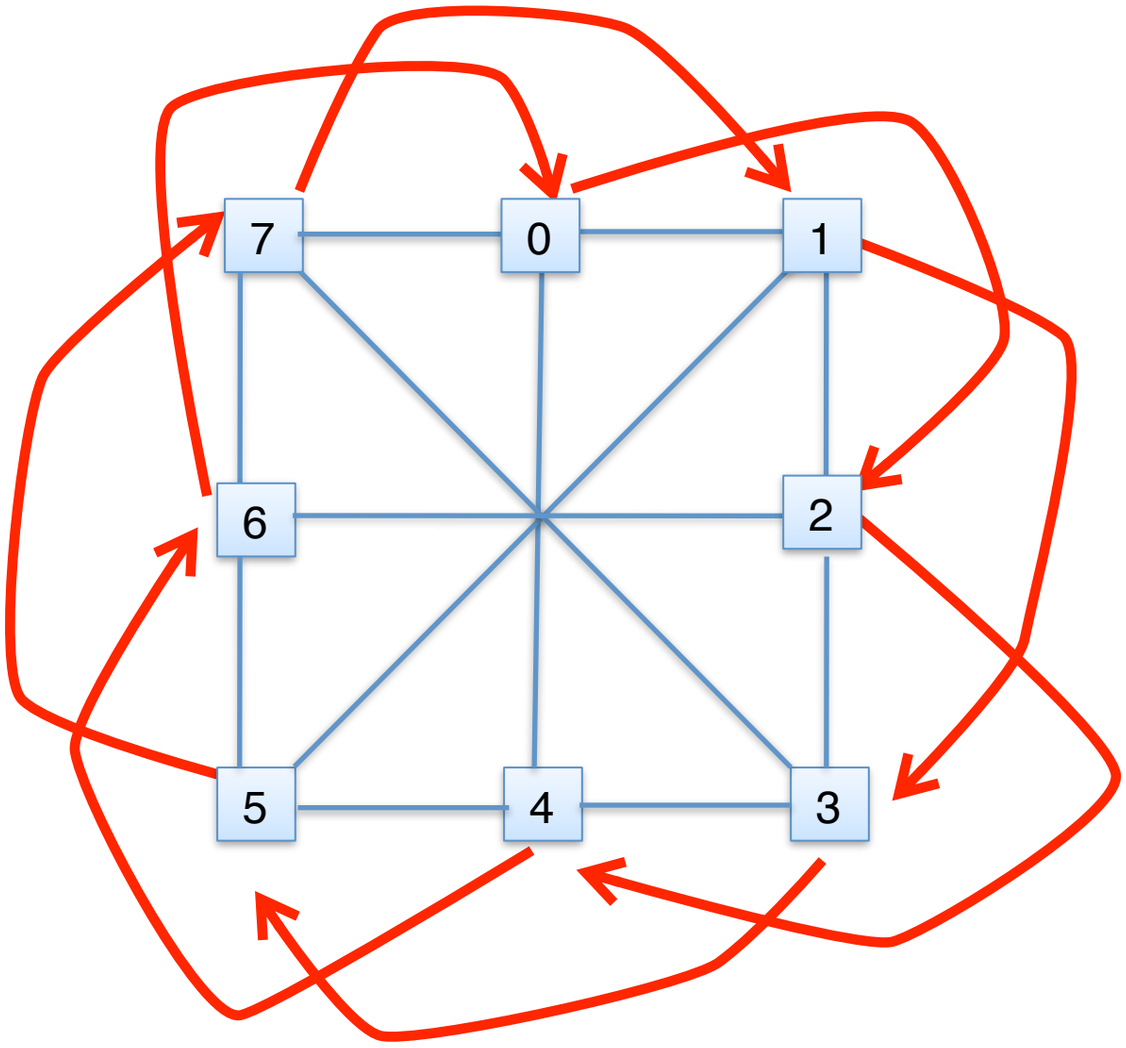
elseif 3*N <= RelAd <= 4*N **then**
    go counter clockwise

else
    go across

endif

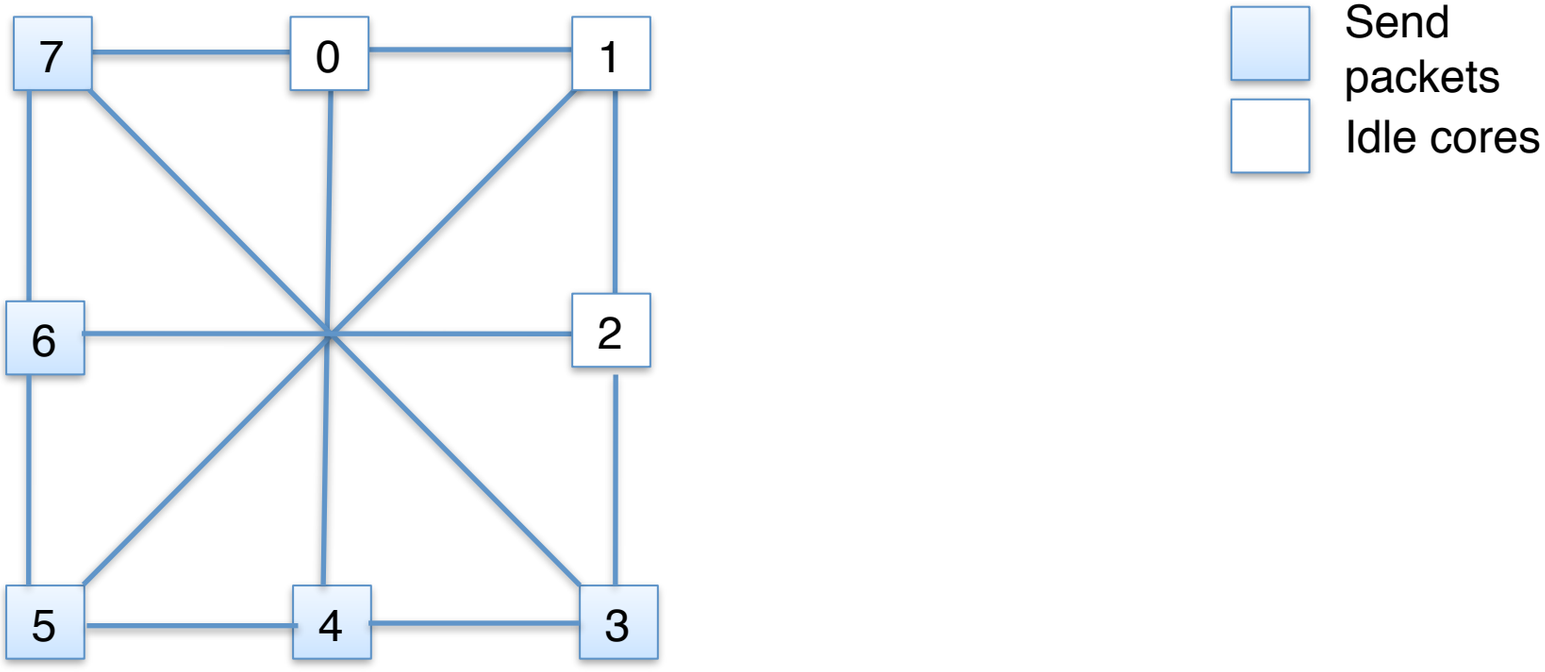
The network has a deadlock !

» For instance, we can have a cycle of packets



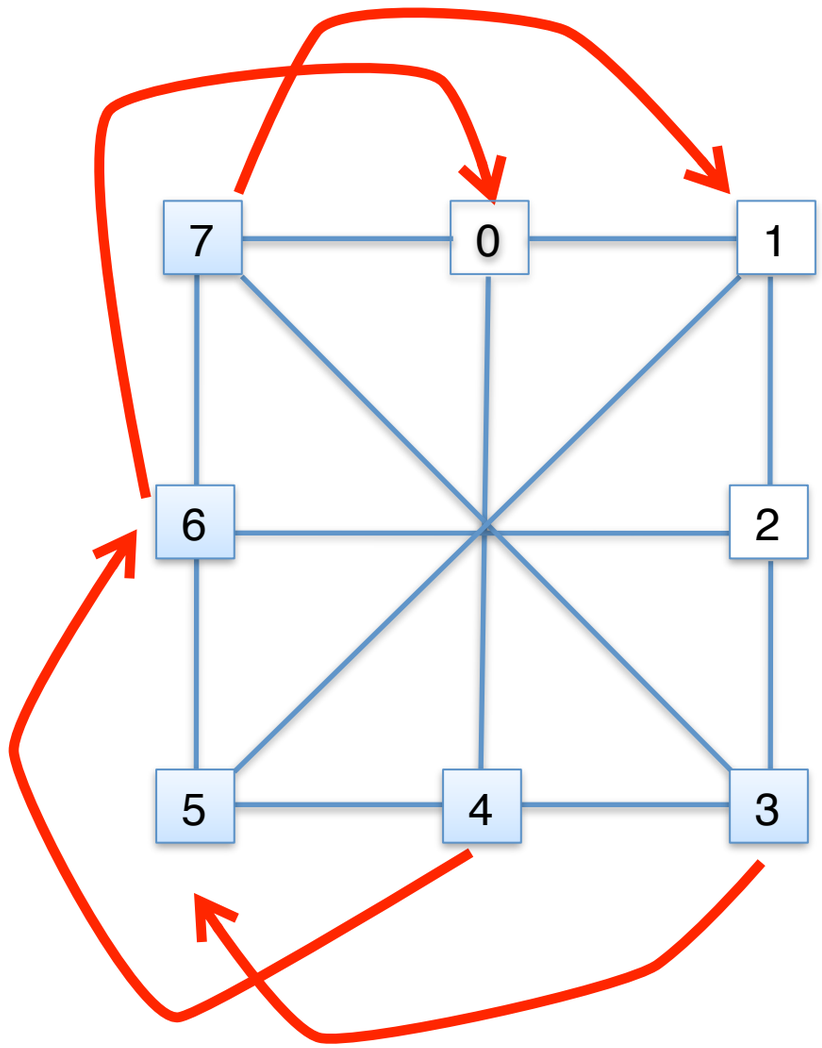
Dividing in two networks

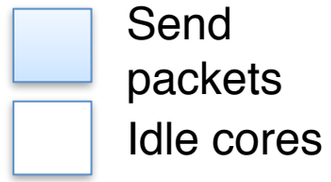
» Is the system deadlock-free ?



Dividing in two networks

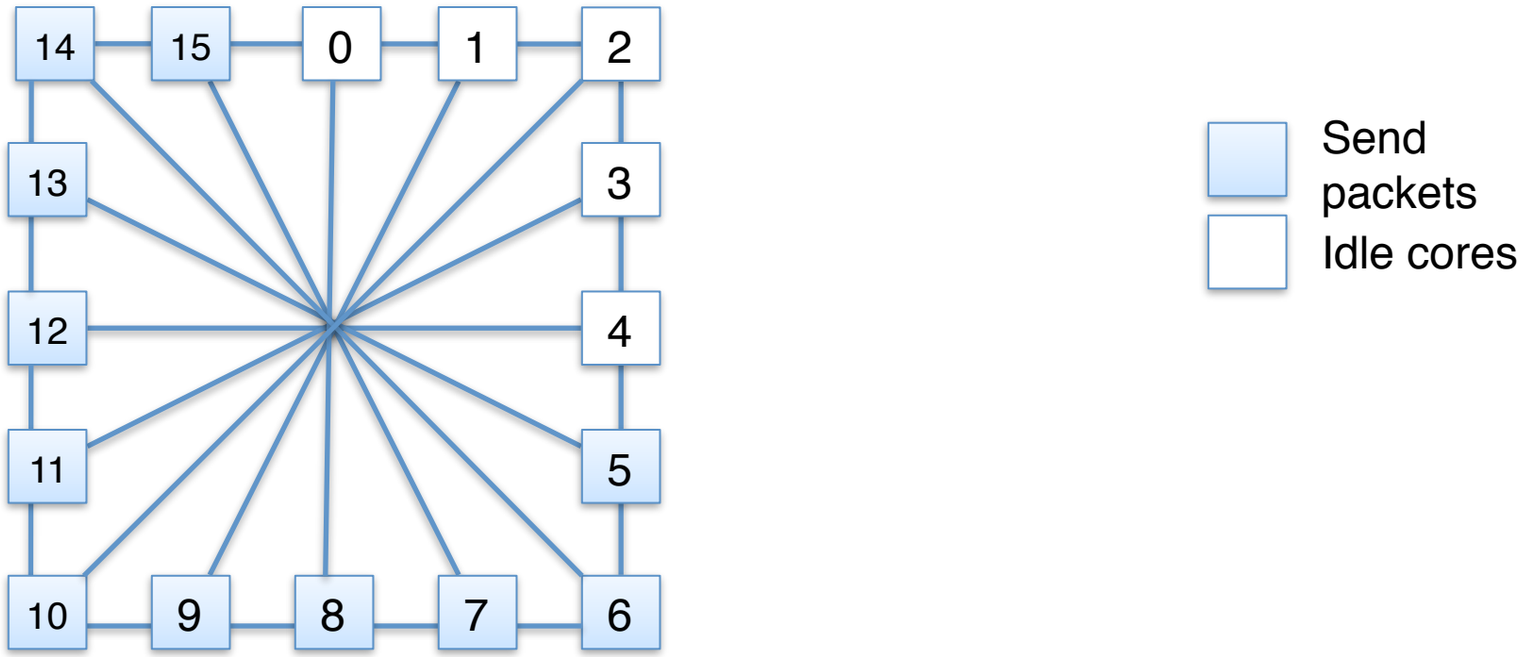
» Is the system deadlock-free ?
» Yes ! None of the dependencies in the right upper quarter occur.





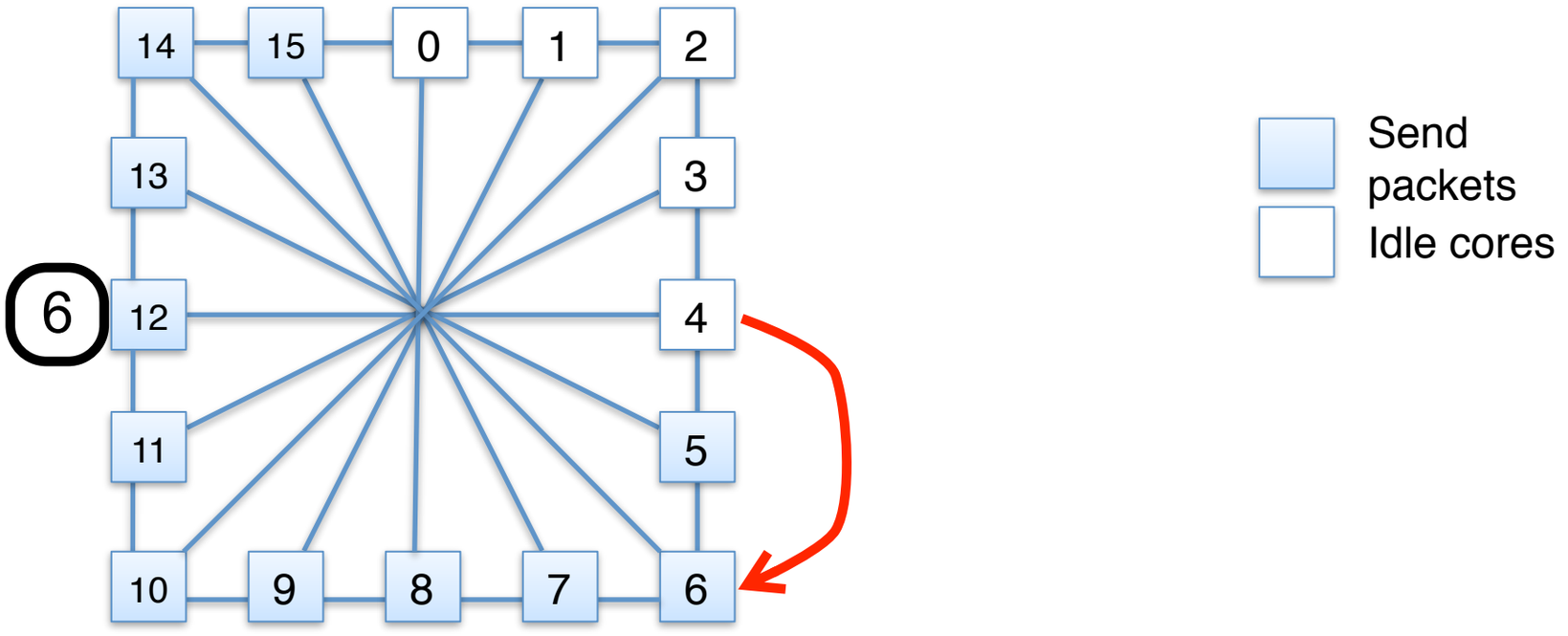
Slaves in 1st quarter only?

» Is the system deadlock-free ?



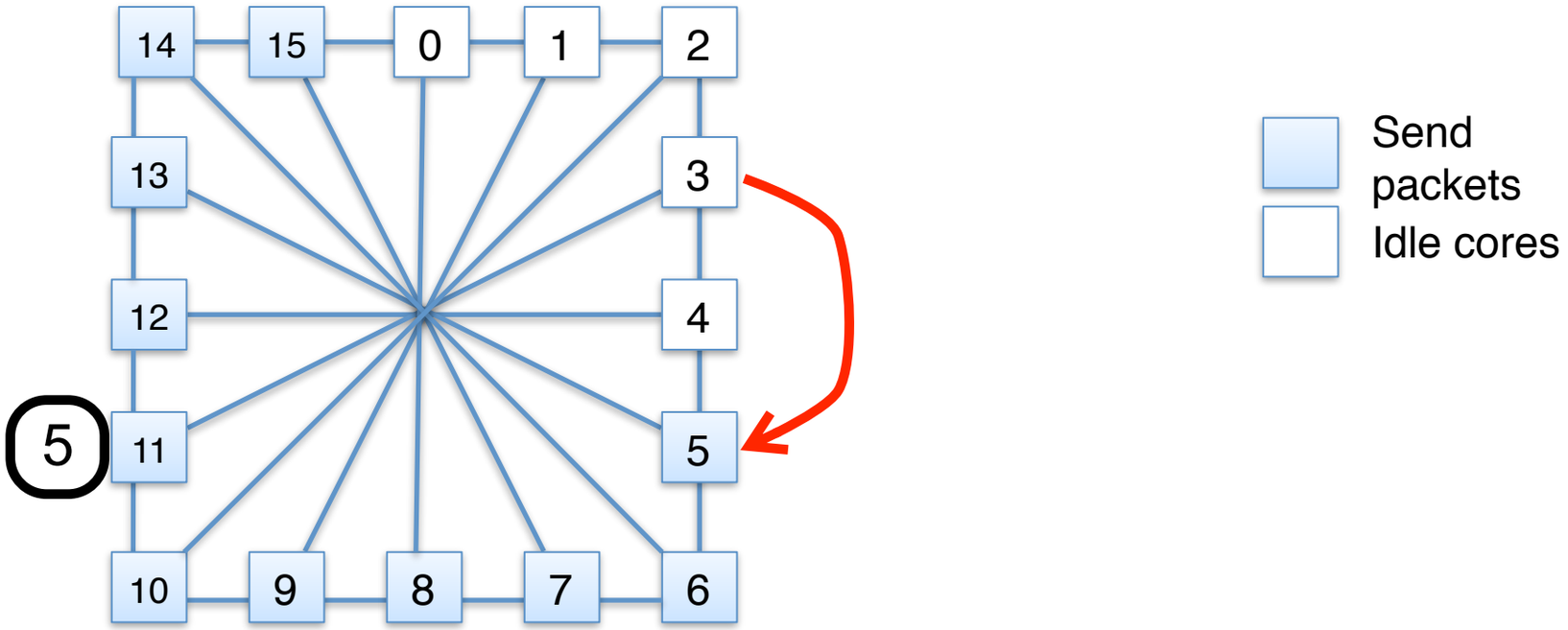
Slaves in 1st quarter only?

» Is the system deadlock-free ?
» No !



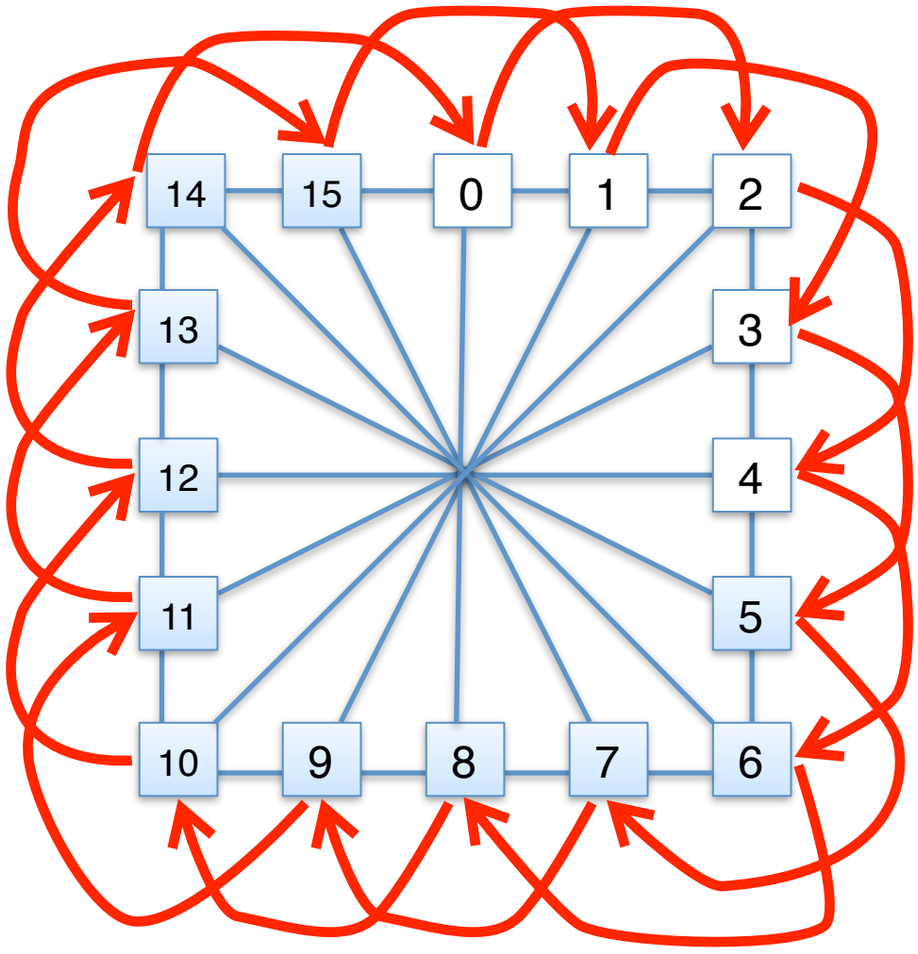
Slaves in 1st quarter only?

» Is the system deadlock-free ?
» No !



Slaves in 1st quarter only?

» Is the system deadlock-free ?
» No !



Send packets
Idle cores

Deadlock-free application layer

\+ Network layer with deadlocks

\+ Deadlock-free link layer

$\stackrel{?}{=}$

Deadlock-free system

Confusing …

» We need tools to (quickly) check for deadlocks
  – taking details of all three layers into account
  – in large systems

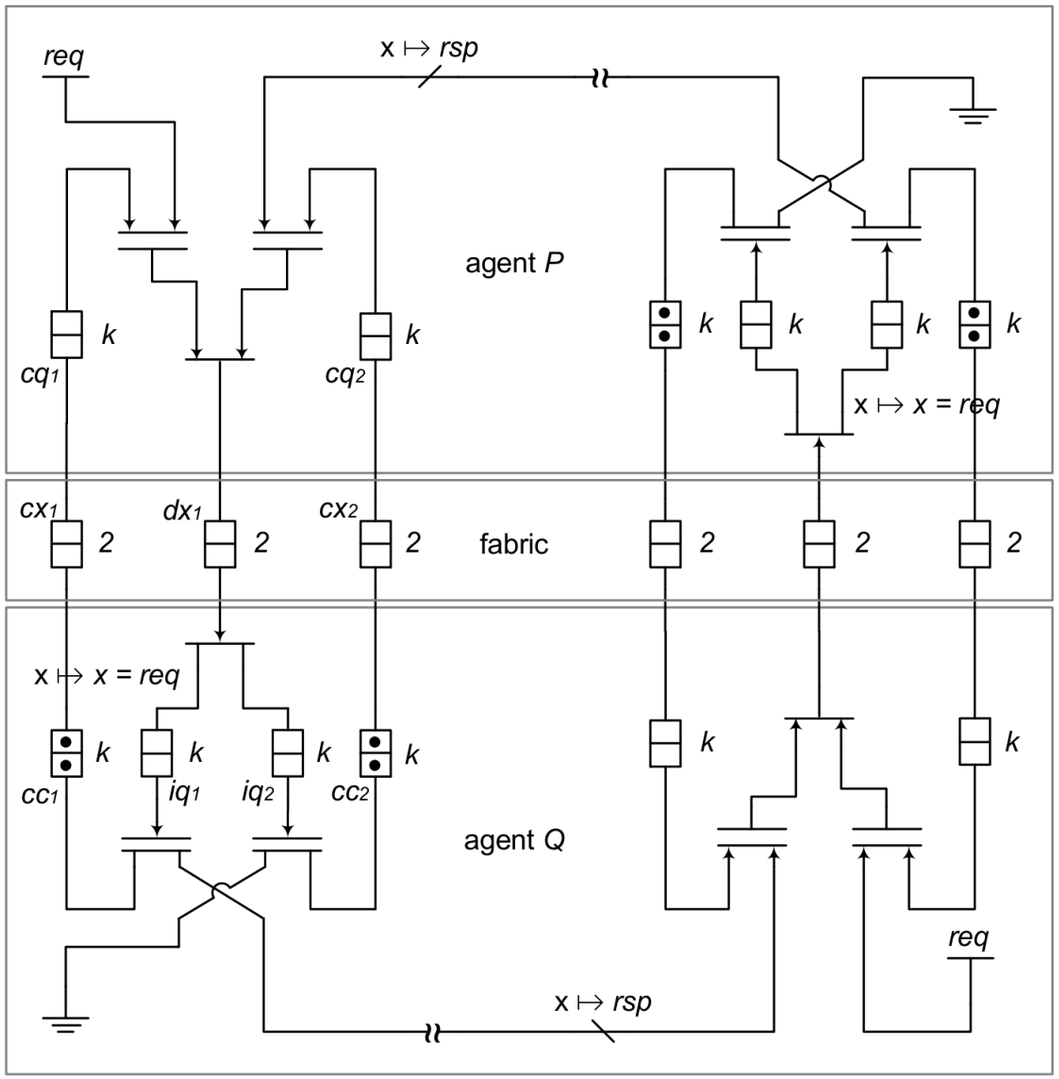
Outline

» Intel's micro-architectural description language
  – xMAS language
  – Capturing high-level structure and message dependencies
  – Extended to “MaDL” at TU/e.
    – Micro-architectural Description Language

» Deadlock verification for MaDL
  – Definition of deadlocks
  – Labelled dependency graph
  – Feasible logically closed subgraph

» Conclusion and future work

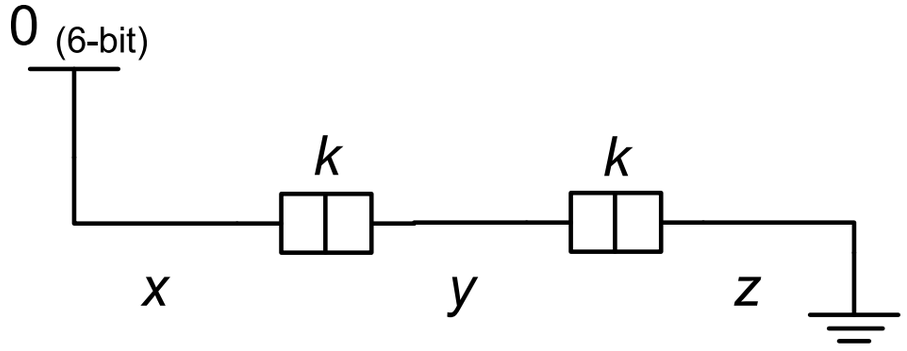
Intel's abstraction for networks



- High-level of abstraction
- Exploit high-level structure

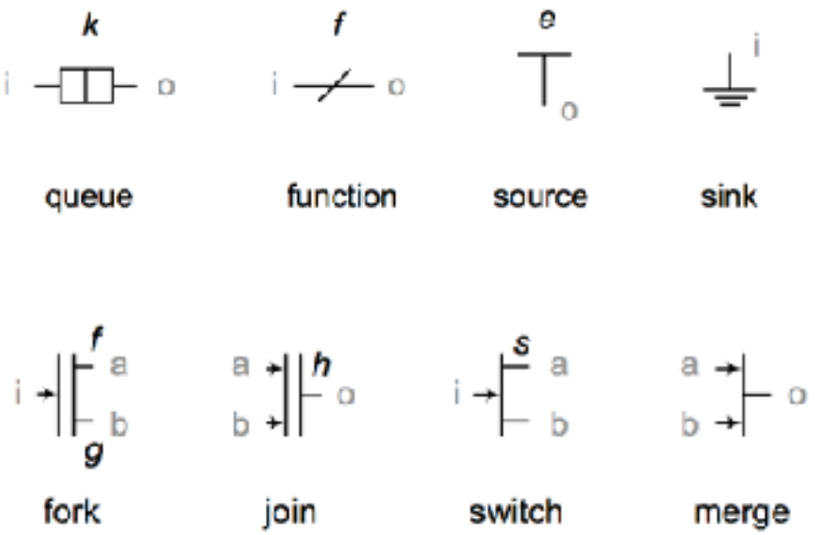
Automatic proofs using invariant generation and hardware model-checking

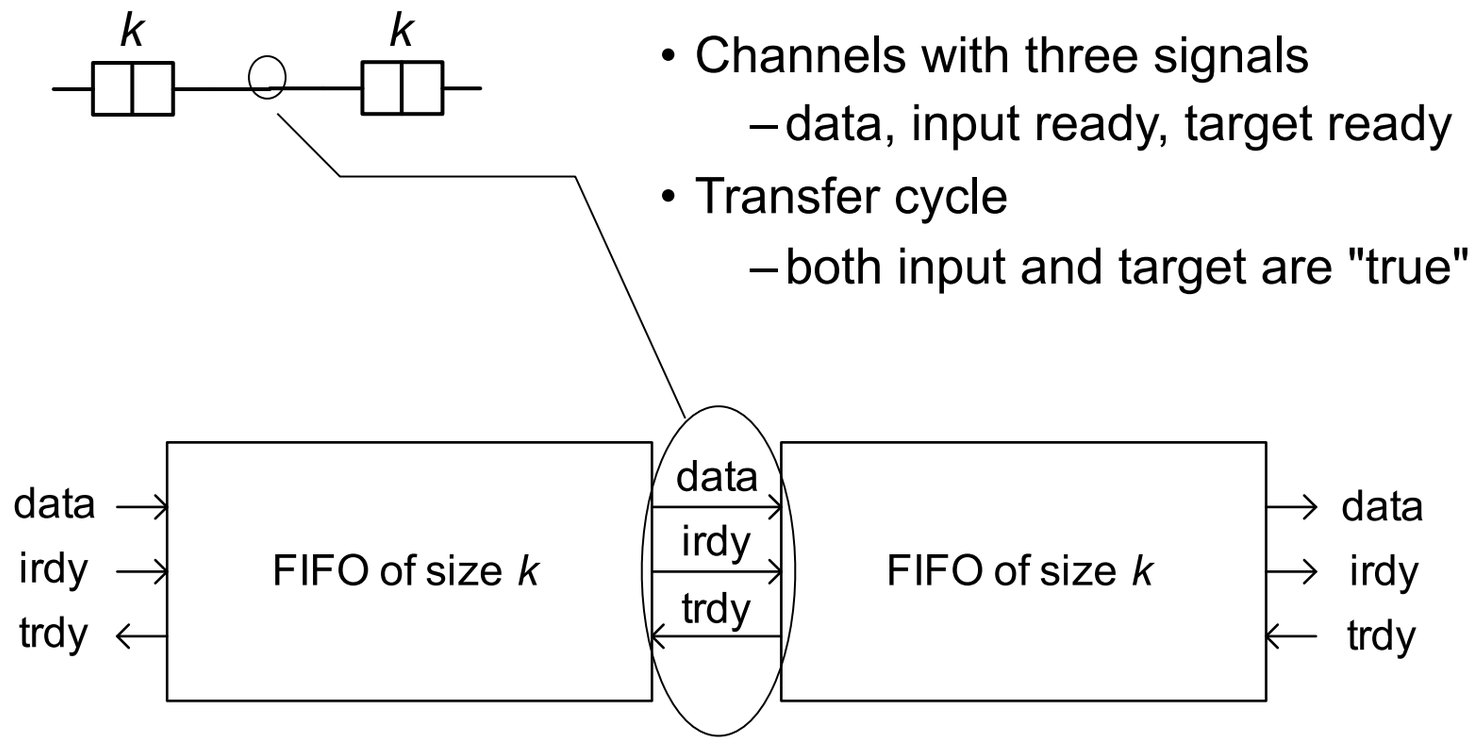
xMAS
Executable Micro-Architectural Specification



- Fair sinks and sometimes sources
- Diagram is formal model
- Friendly to microarchitects

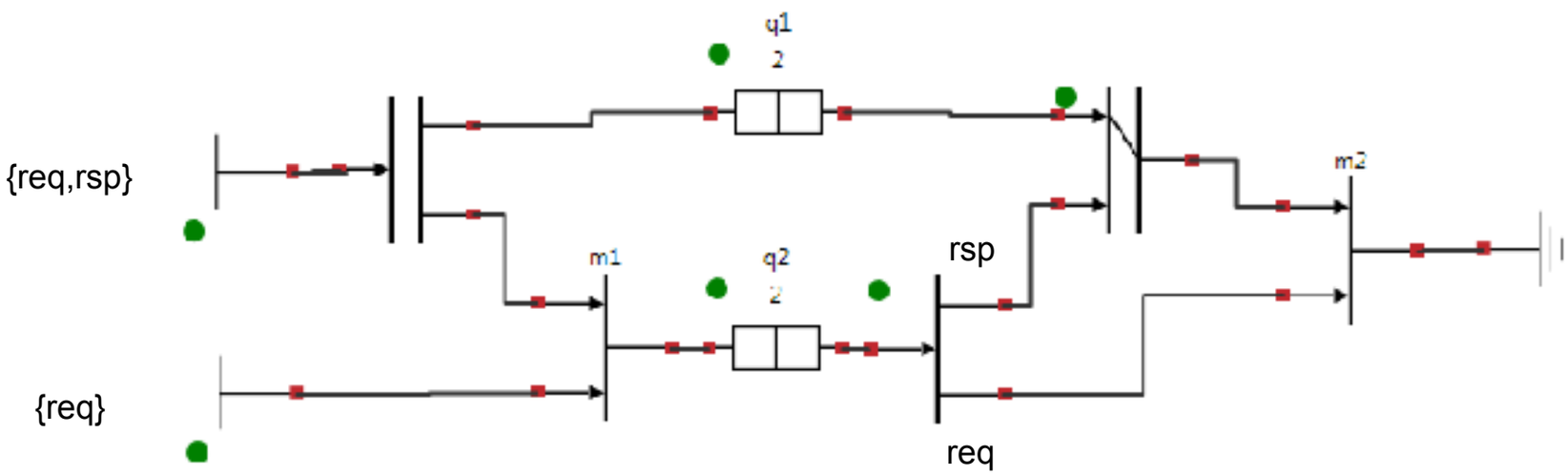
MaDL (1)





- Channels with three signals
  - data, input ready, target ready
- Transfer cycle
  - both input and target are "true"

Notion of a “dead” channel: F (c.irdy & G ! c.trdy)

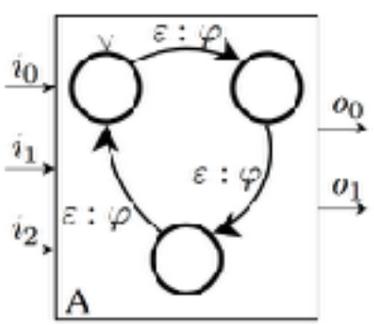


MaDL (2)

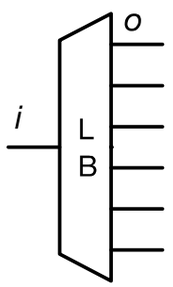
textual input

```
param int QSIZE;
const pkt;
chan src := Source(pkt);
chan one, two := Fork(src);
chan long := Queue(QSIZE,Queue(QSIZE, one));
chan short := Queue(QSIZE, two);
Sink(CtrlJoin(long,short));
```

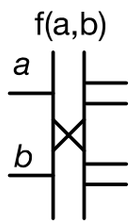
additional complex primitives



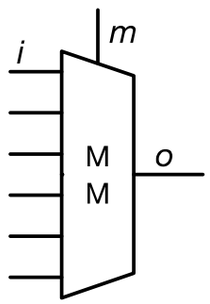
IO-FSM



LoadBalancer



Joitch



Multi-Match

And more: for-loops, if-then-else, uses, …

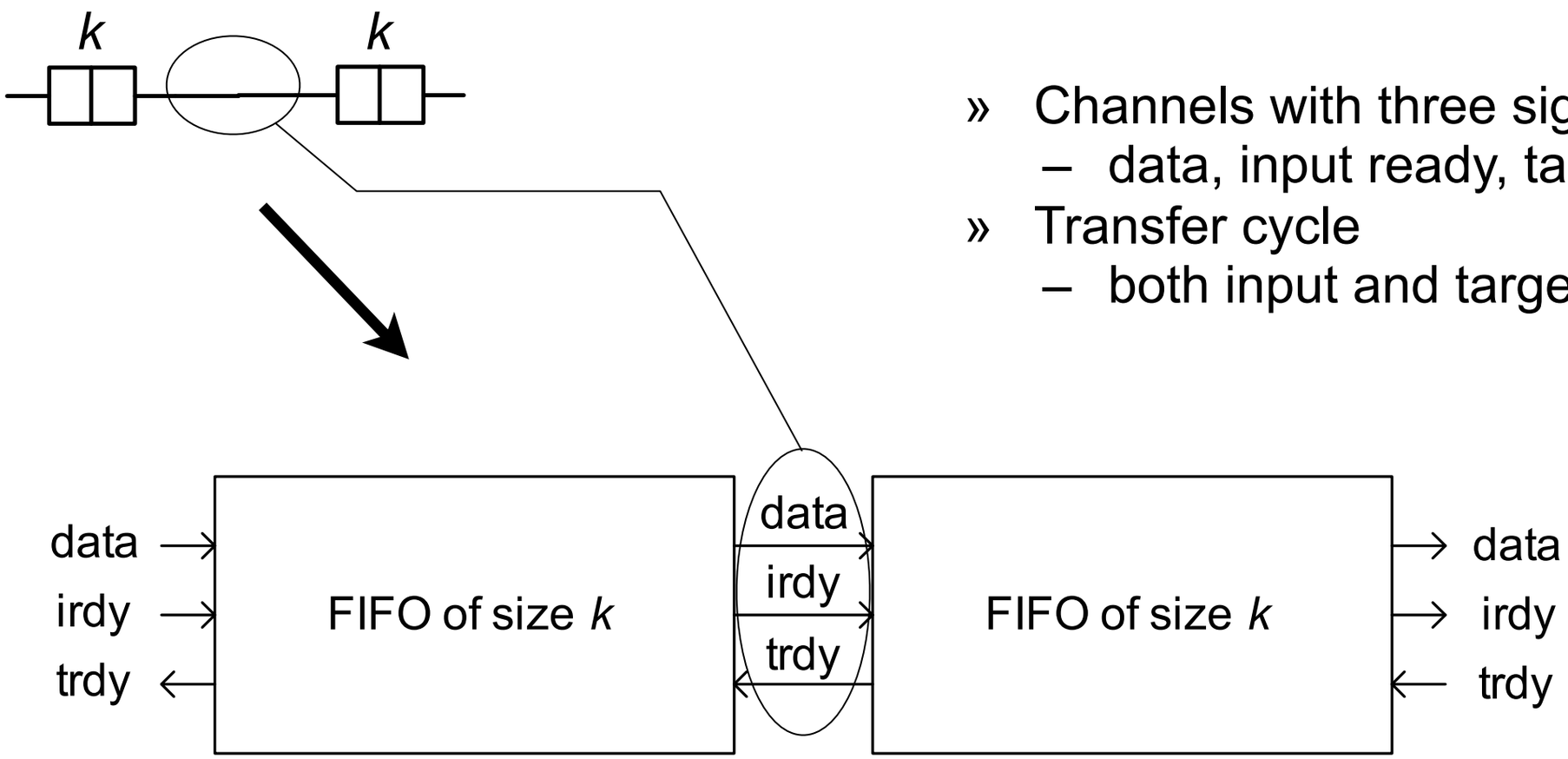
predicates and functions

```
pred P (p : pkt_t, q : pkt_t ) {
    p.id == q.id
};
function F (p : pkt_t) : pkt_t {
    fldA = p.fldB;
    fldB = p.fldA;
};
```

structured recursive data types

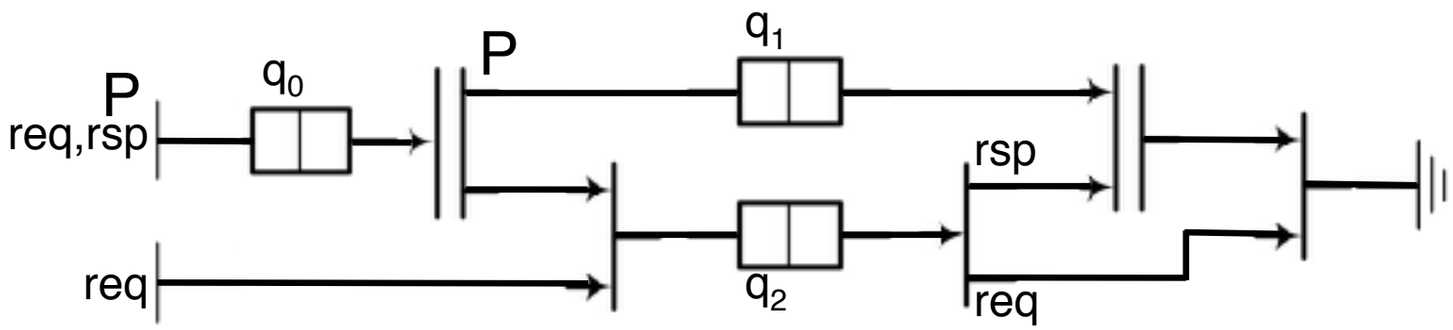
```
struct pkt0_t {
    fldA : [2:0];
    fldB : [3:0];
};
union pkt1_t {
    optionA : pktA_t;
    optionB : pktB_t;
};
enum pkt2_t { optionA; optionB; };

const optionA;
```

Composing modules via channels



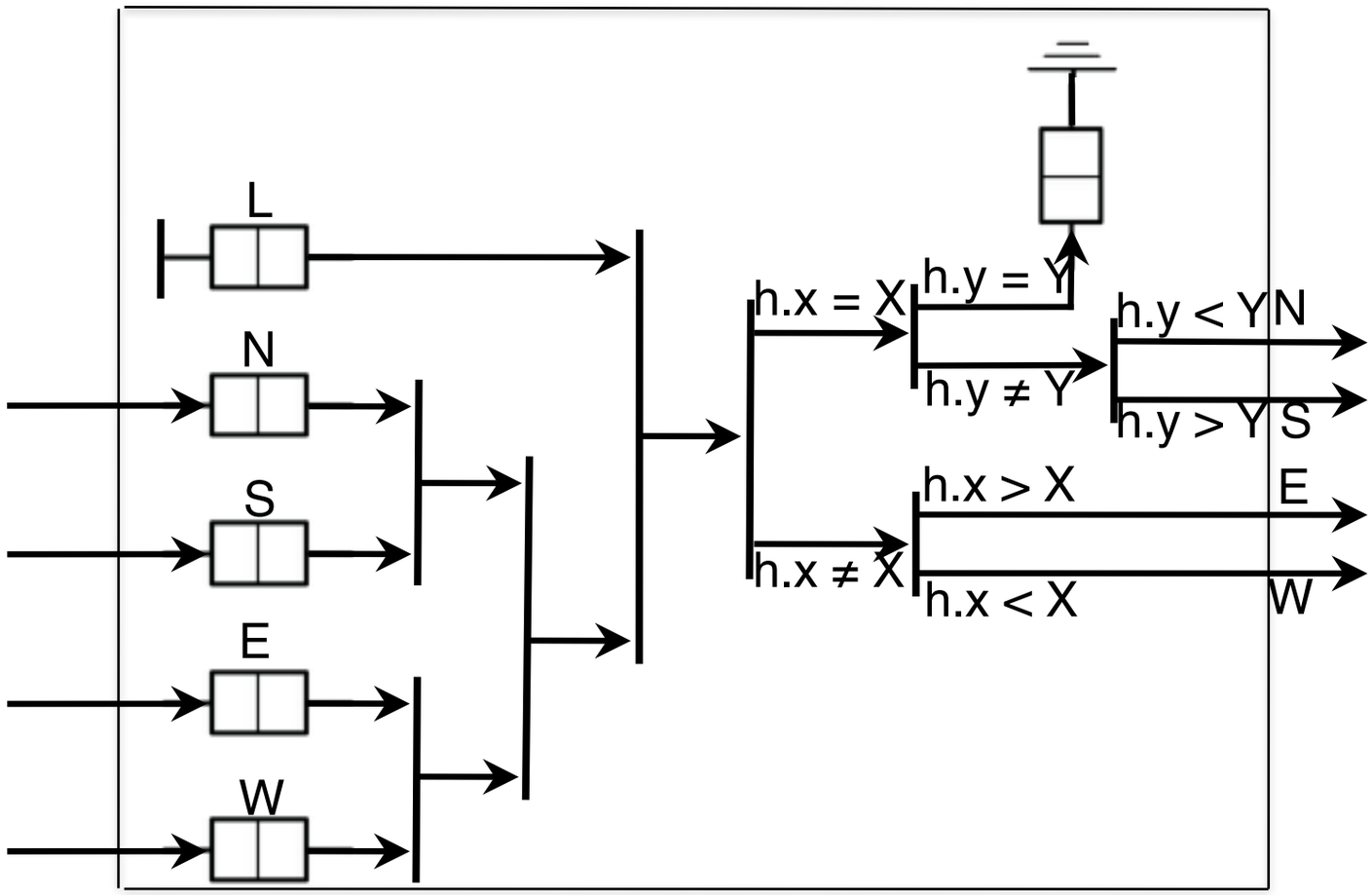
- Channels with three signals
  - data, input ready, target ready
- Transfer cycle
  - both input and target are "true"

MaDL example

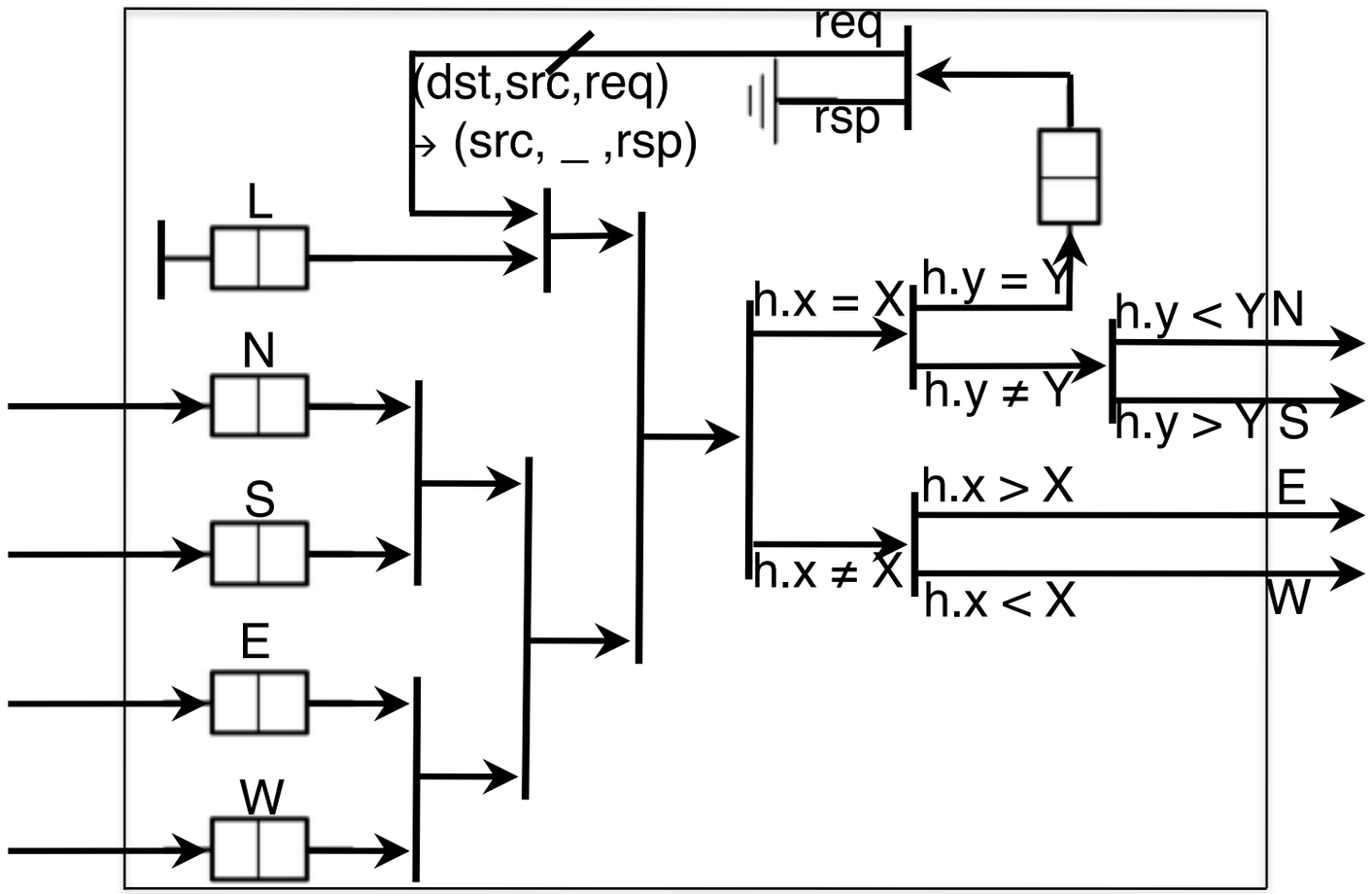


» Two sources
  – one for requests
  – one for responses

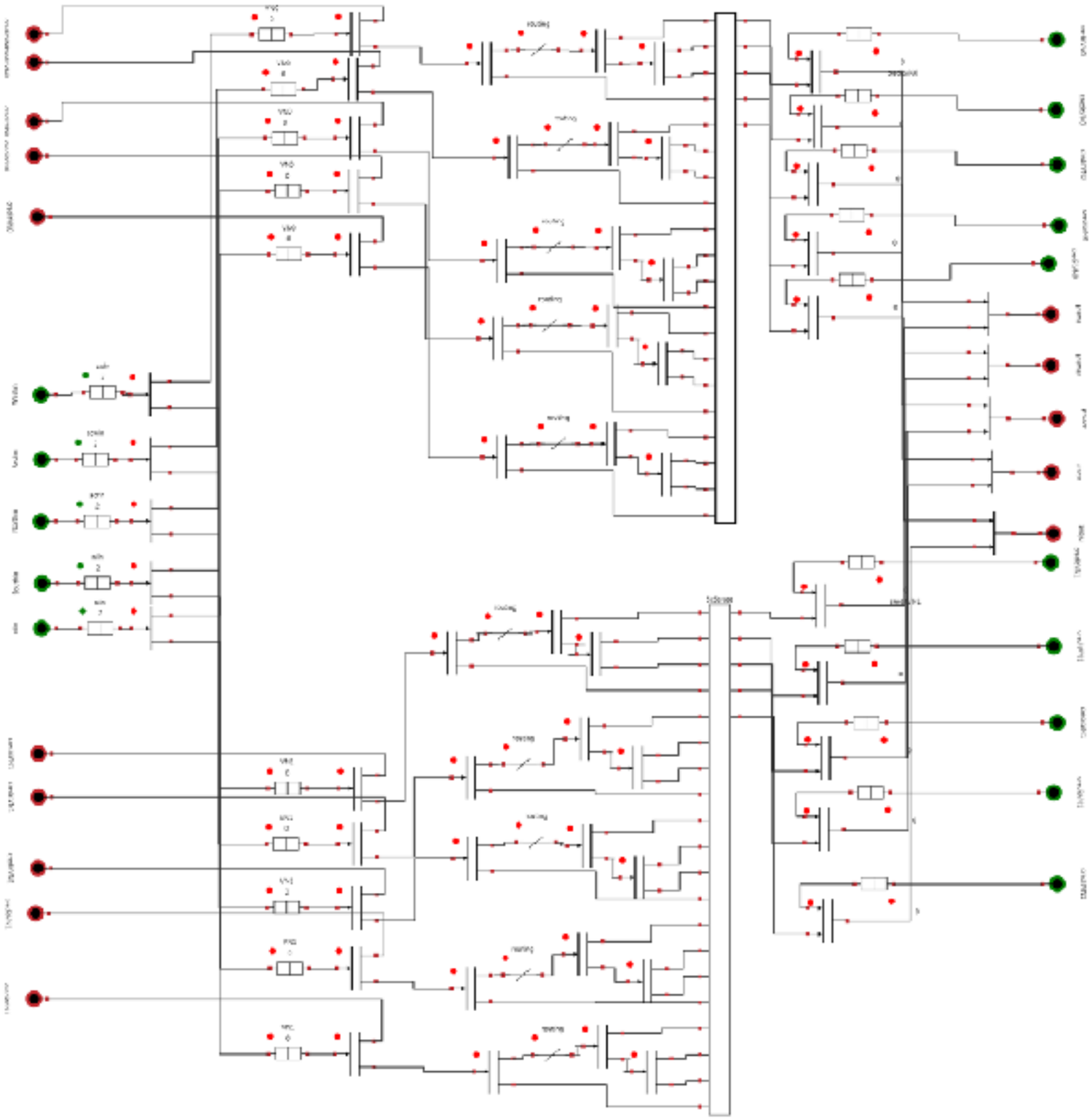
Processing node for XY routing in a 2D-mesh



Processing node with requests and responses



A more complex processing nodes with virtual channels and credits



Outline

- Intel's micro-architectural description language
  - xMAS language
  - Capturing high-level structure and message dependencies
  - Extended to “MaDL” at TU/e.
    - Micro-architectural Description Language

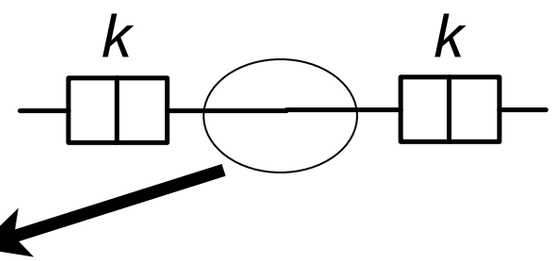
- **Deadlock verification for MaDL**
  - **Definition of deadlocks**
  - **Labelled dependency graph**
  - **Feasible logically closed subgraph**

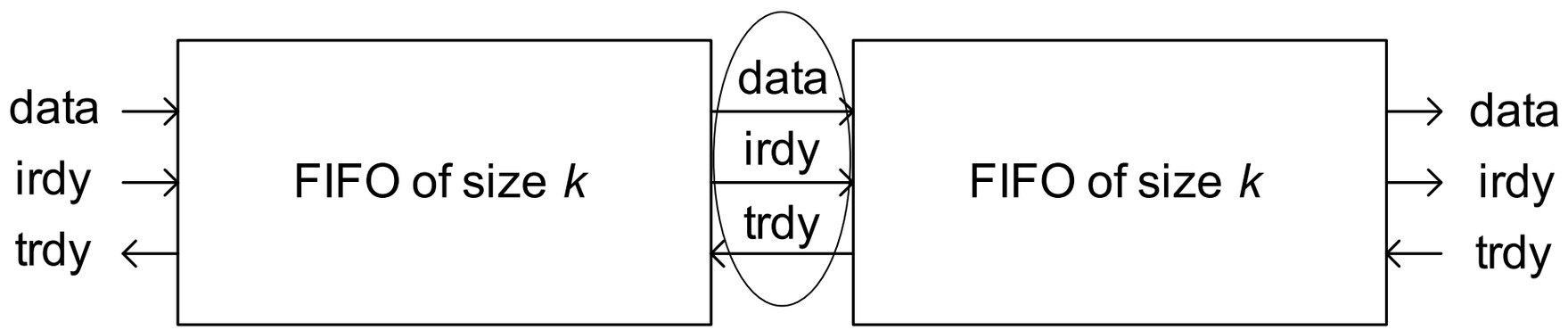
- Conclusion and future work

Formal definition of "deadlock" in MaDL

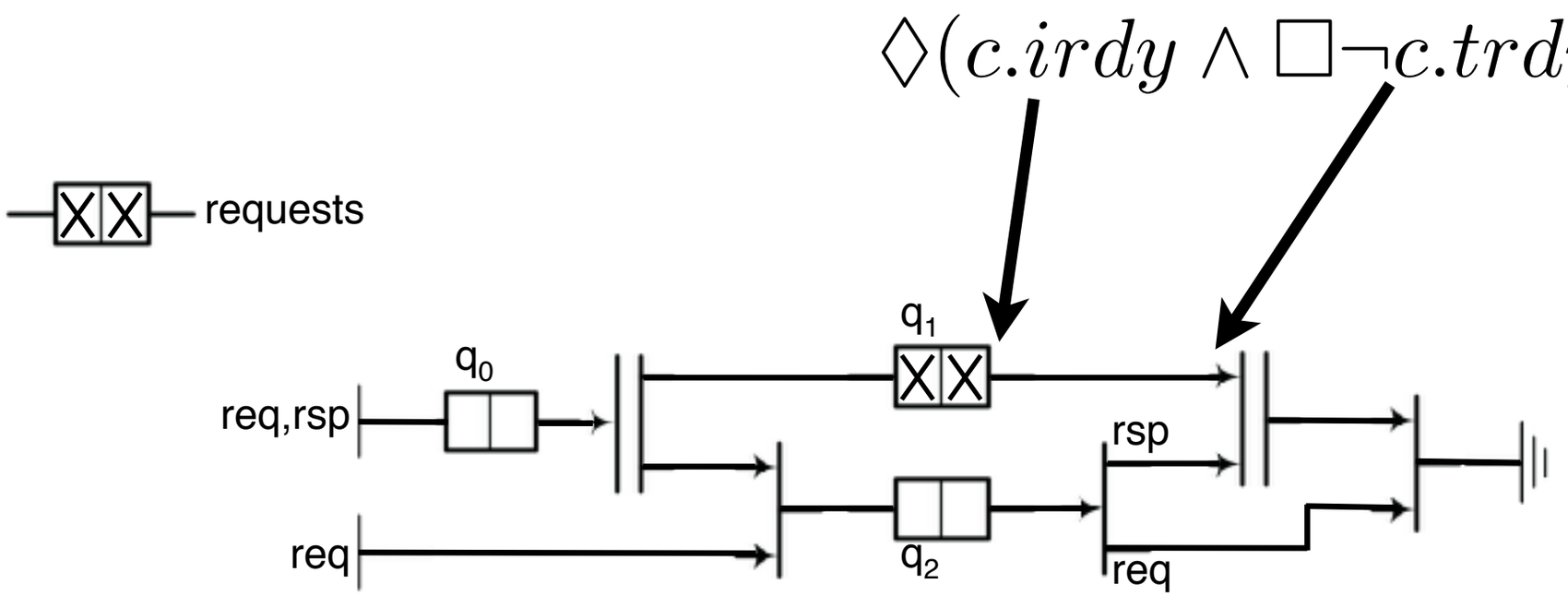
- Intuition is a "dead" channel
- Formal definition based on Linear Temporal Logic
  - –Predicate logic
  - –Temporal operators "eventually" ($\Diamond$) and "globally" ($\Box$)
- Channel *c* is dead iff

$$\Diamond(c.irdy \wedge \Box\neg c.trdy)$$



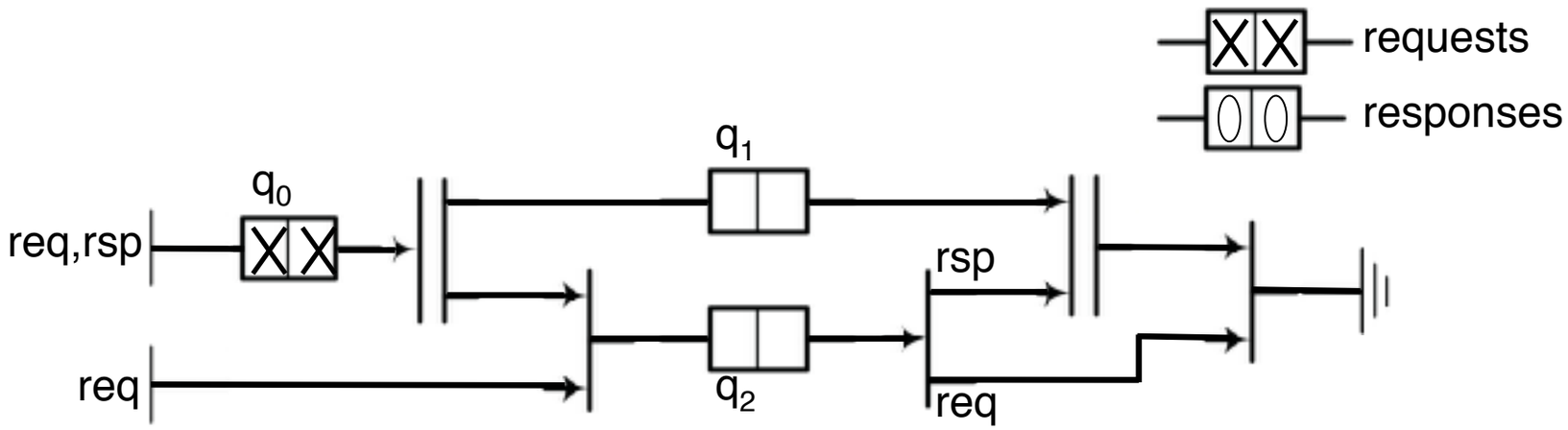


MaDL example



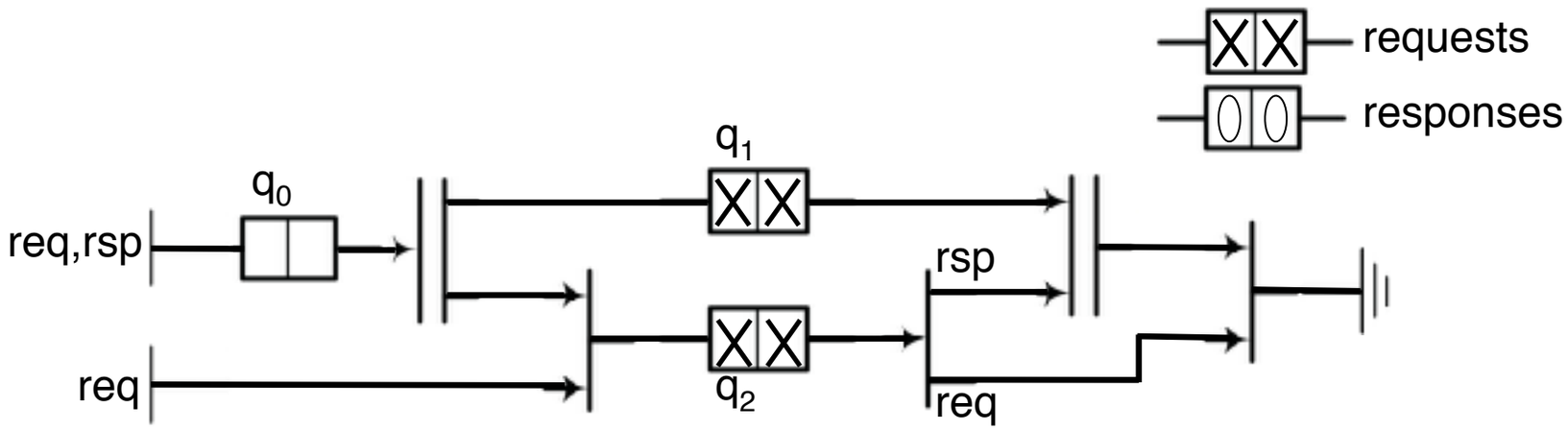
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk

MaDL example



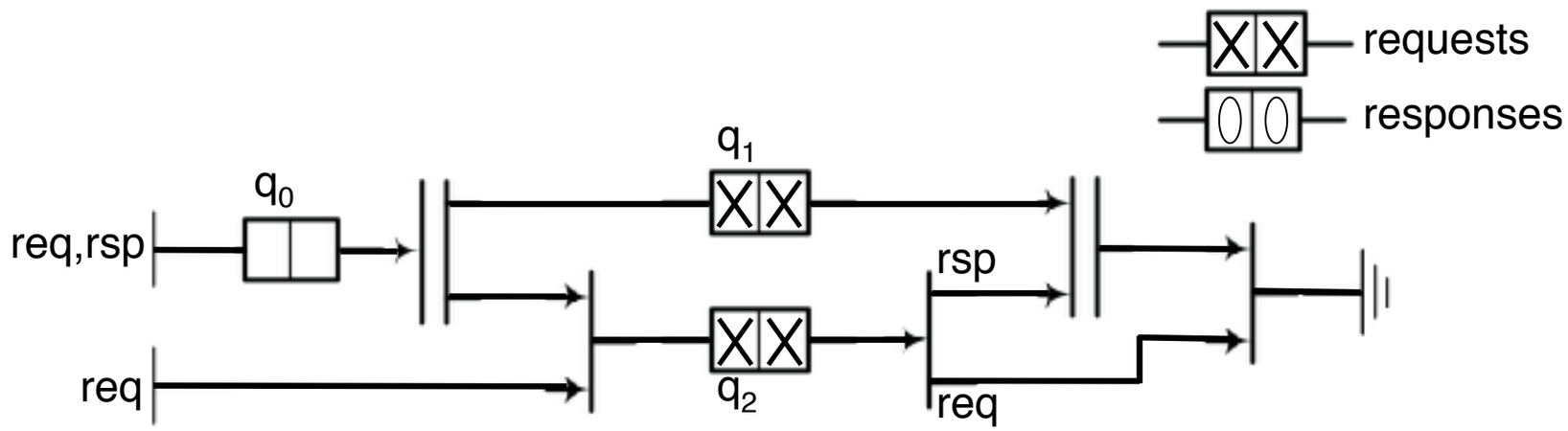
- Inject two requests in q0

MaDL example



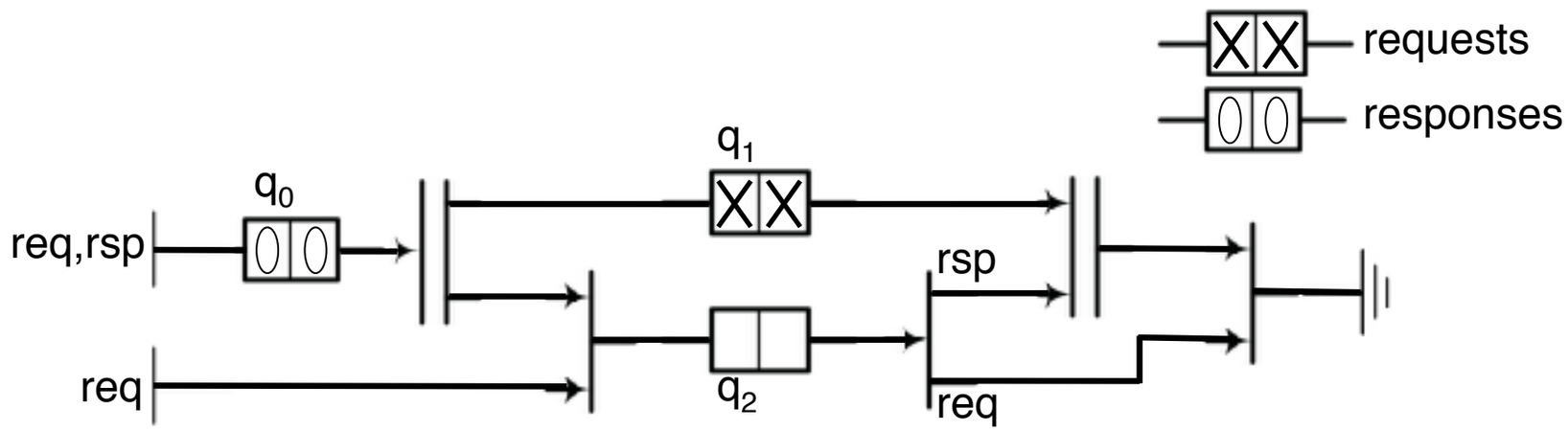
- Inject two requests in q0
- Fork creates two copies

MaDL example



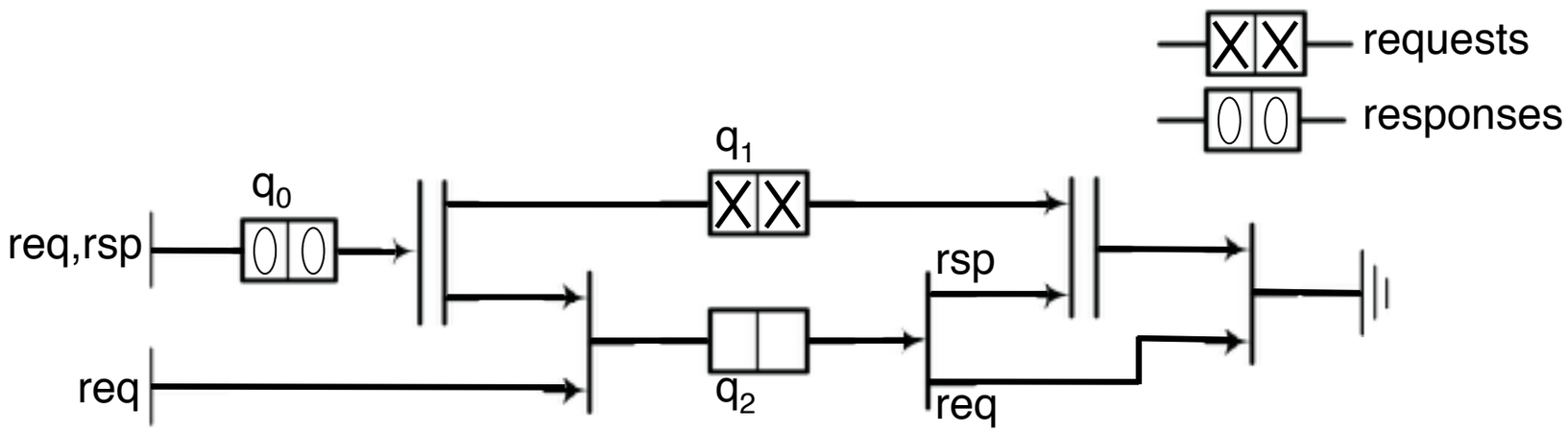
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk

MaDL example



- Inject two requests in q0
- Fork creates two copies
- One pair is sunk
- Inject two responses in q0

MaDL example



- Inject two requests in q0
- Fork creates two copies
- One pair is sunk
- Inject two responses in q0
- q2 is permanently idle for responses, q1 is permanently blocking

We have a deadlock without a circular wait !

General approach for deadlock detection in MaDL networks

- Define deadlock equations for all components
  - –Equations capture the reason why a component is idle or blocking

- Build a labelled waiting graph for each queue
  - –Labels correspond to the equations
  - –Graph captures the topology, i.e., the dependencies between the MaDL components

- Search for a feasible logically closed subgraph
  - –Corresponds to a deadlock situation
  - –Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - –A first step generates invariants to rule out false deadlocks
  - –Invariants are rather weak and simple - false deadlocks are in theory still possible

General approach for deadlock detection in xMAS networks

- Define deadlock equations for all components
  - Equations capture the reason why a component is idle or blocking

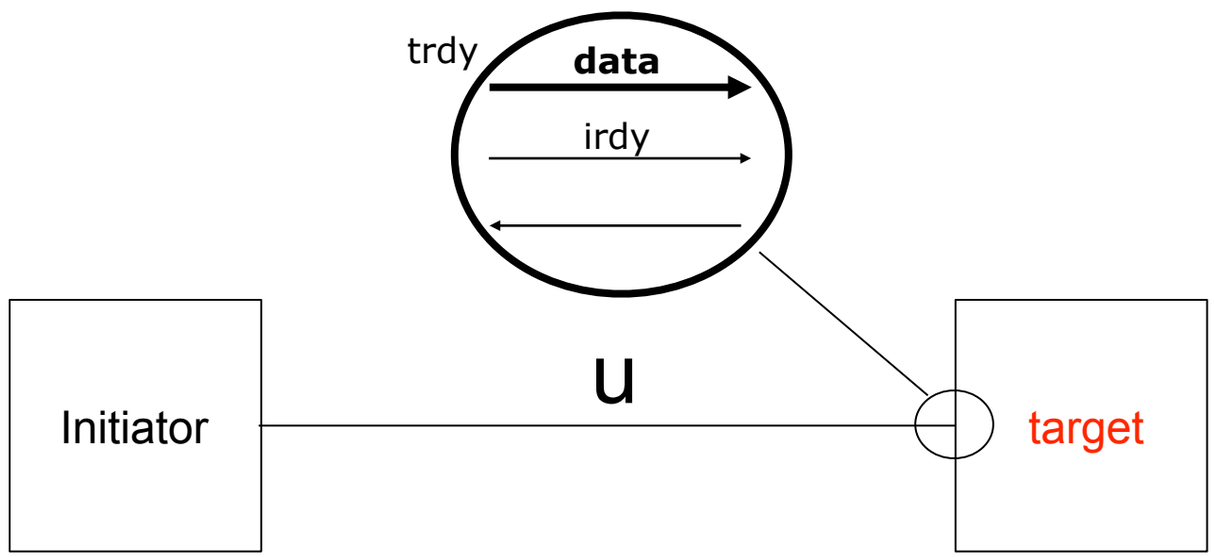
- Build a labelled waiting graph for each queue
  - Labels correspond to the equations
  - Graph captures the topology, i.e., the dependencies between the MaDL components

- Search for a feasible logically closed subgraph
  - Corresponds to a deadlock situation
  - Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - A first step generates invariants to rule out false deadlocks
  - Invariants are rather weak and simple - false deadlocks are in theory still possible

Deadlock equations for a channel

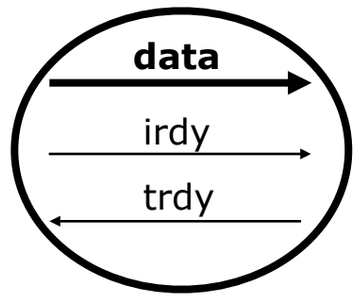
- Depends on the target component connected to the channel
- We look at the input port of the target component



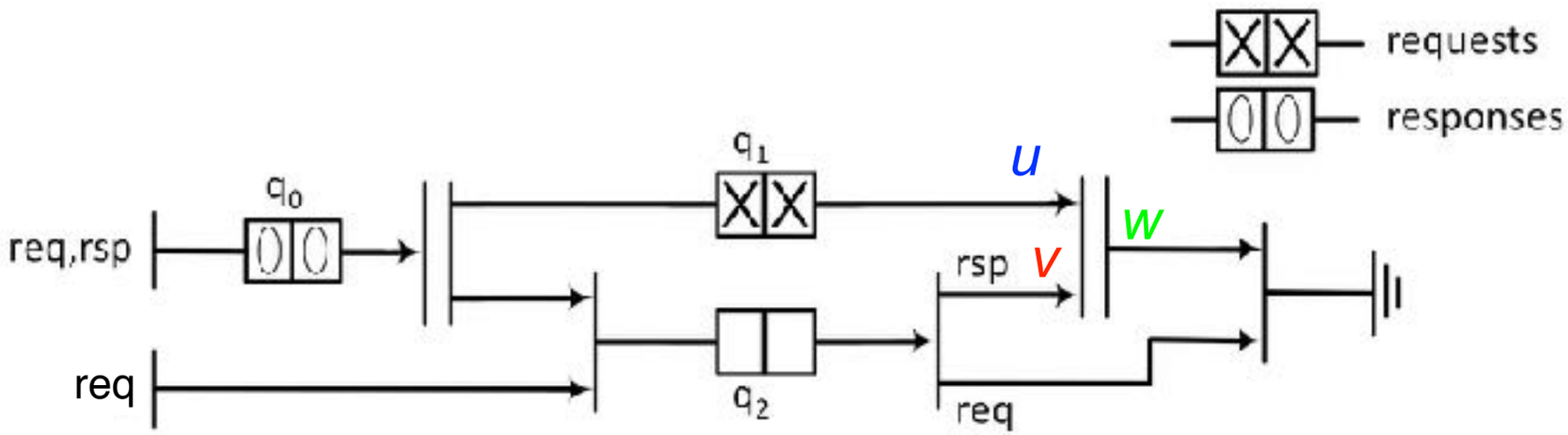
Deadlock equations for a join

- 2 cases
  - –output is blocked
  - –the other input is idle

We need to know when a channel is idle !

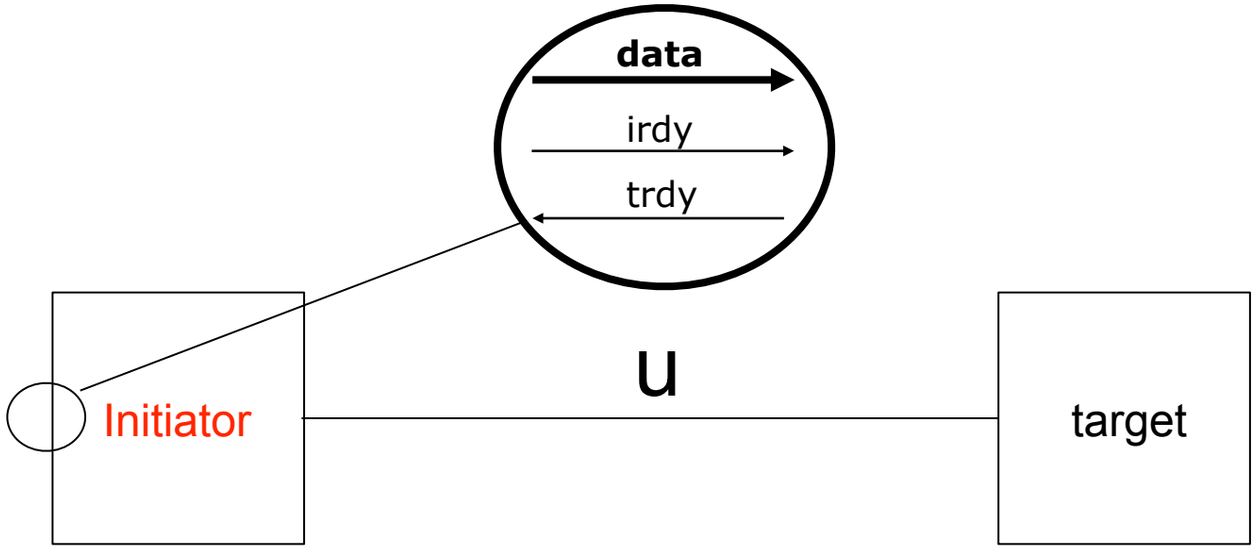


- **Block**(u) = **Idle**(v) + **Block**(w)



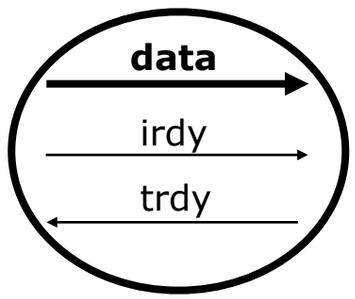
Idle equations for a channel

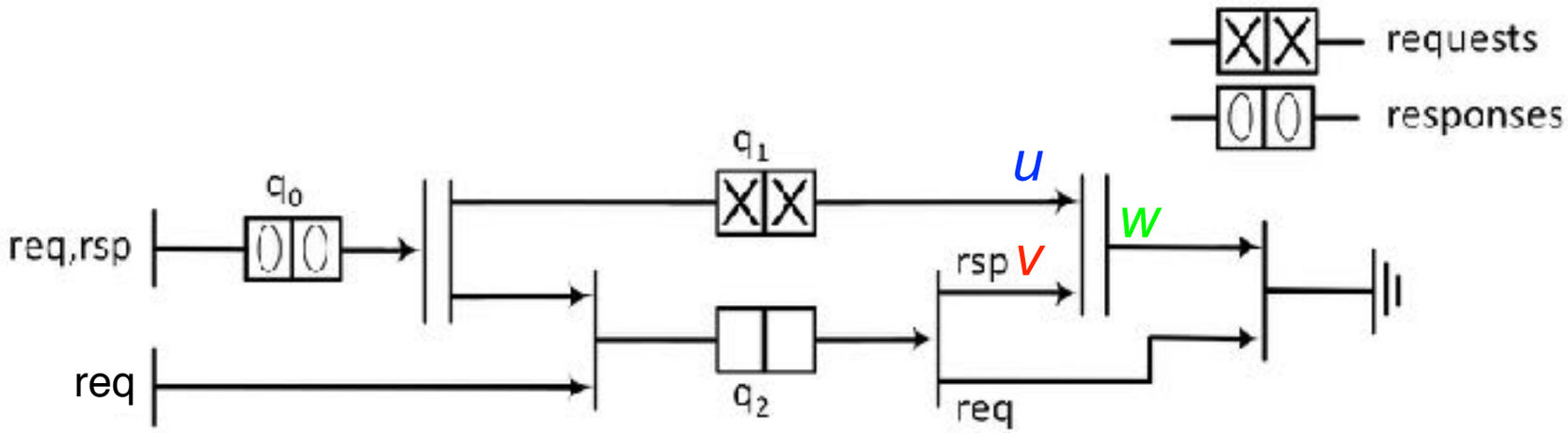
- Depends on the initiator component connected to the channel
- We are looking at the input port of the initiator



Idle equations for a join

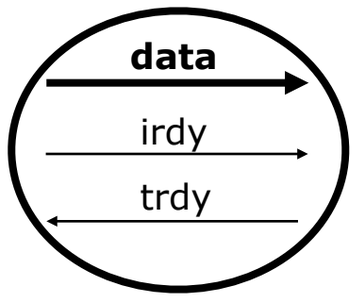
- A join is idle if one of the input channels is idle
- **Idle**(w) = **Idle**(u) + **Idle**(v)

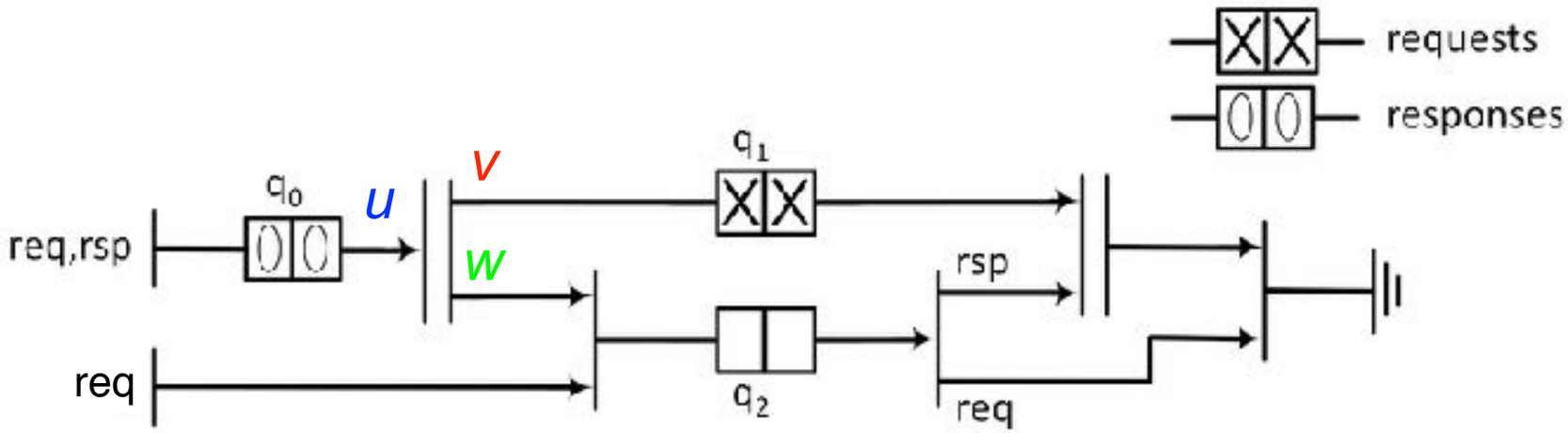




Idle equations for a fork

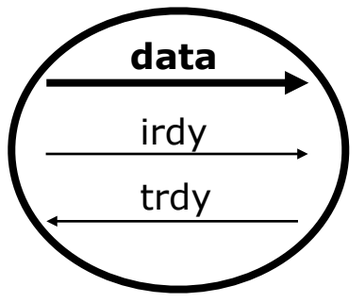
- A fork output is idle if the input is idle or the other output is blocked
- **Idle**(w) = **Idle**(u) + **Block**(v)



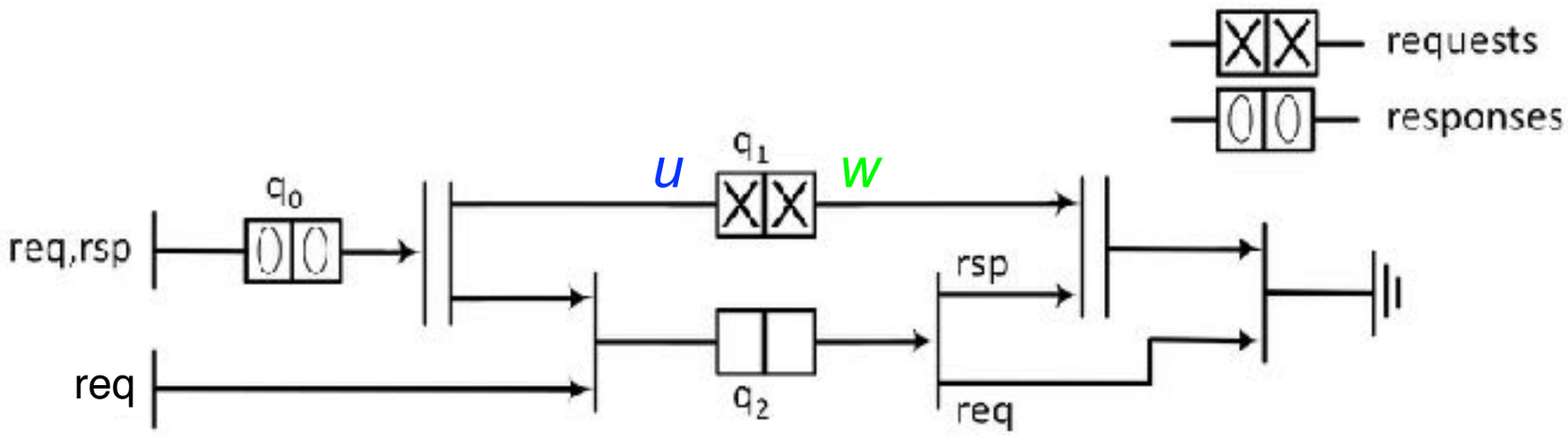


Idle equations for a queue

- A queue is idle if it is empty and its input channel is idle
- This is for one message type which might be blocked by another type



- **Idle**(w) = **Empty**(q) . **Idle**(u) + **Block**(w')
  - –where w' is a message with a type different from w



Our quest for "dead" queues

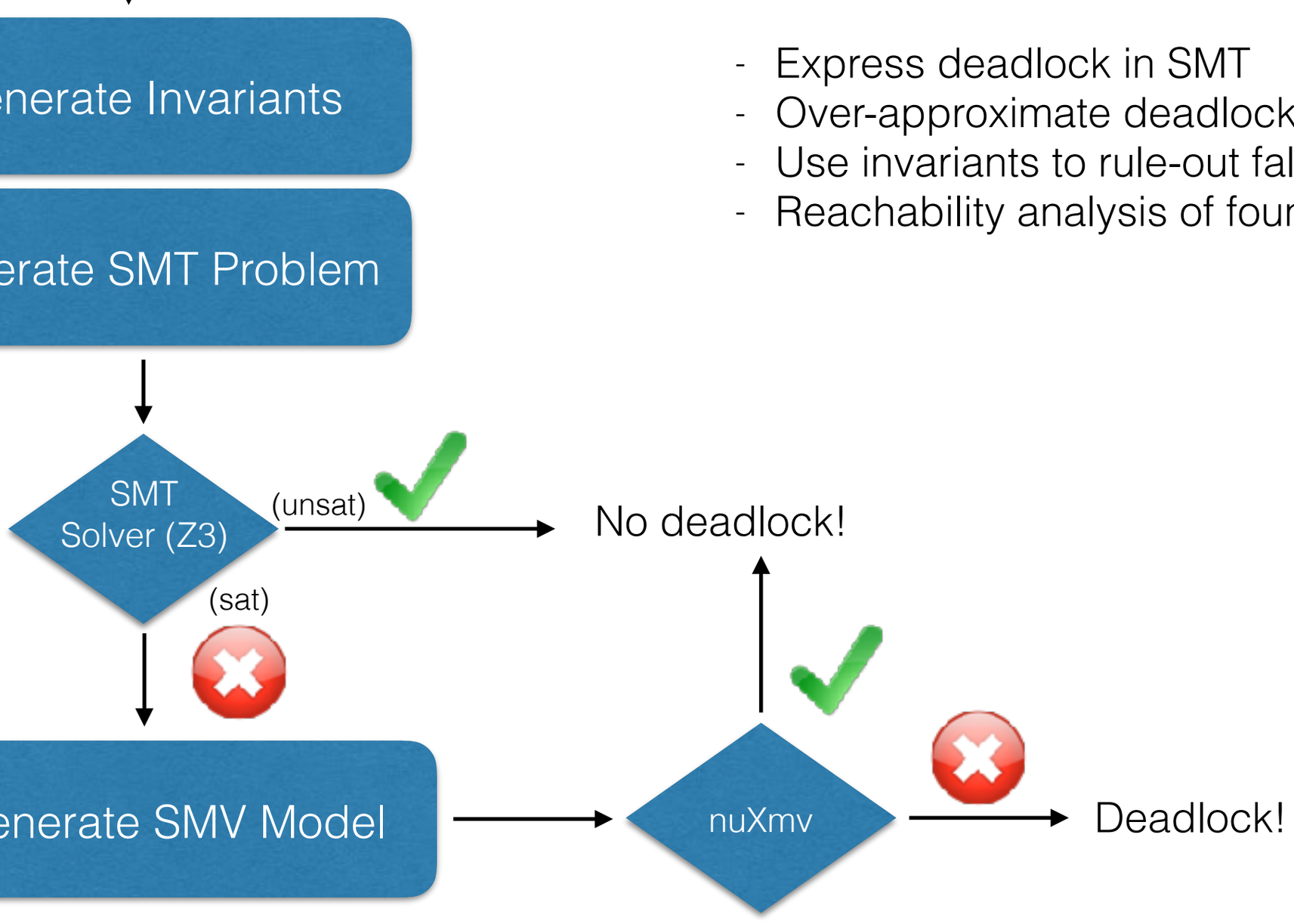
- Definition of a deadlock
  - –F (u.irdy => G~u.trdy)

- We look for a "dead" queue
  - –with a message in it (u.irdy)
  - –output blocked (G~u.trdy)

- Over approximation
  - –configuration not always reachable
  - –we may output false deadlocks

MaDL Verification: Overview

Generate Invariants



Basic idea:

- Express deadlock in SMT
- Over-approximate deadlocks
- Use invariants to rule-out false deadlocks
- Reachability analysis of found deadlocks

General approach for deadlock detection in MaDL networks

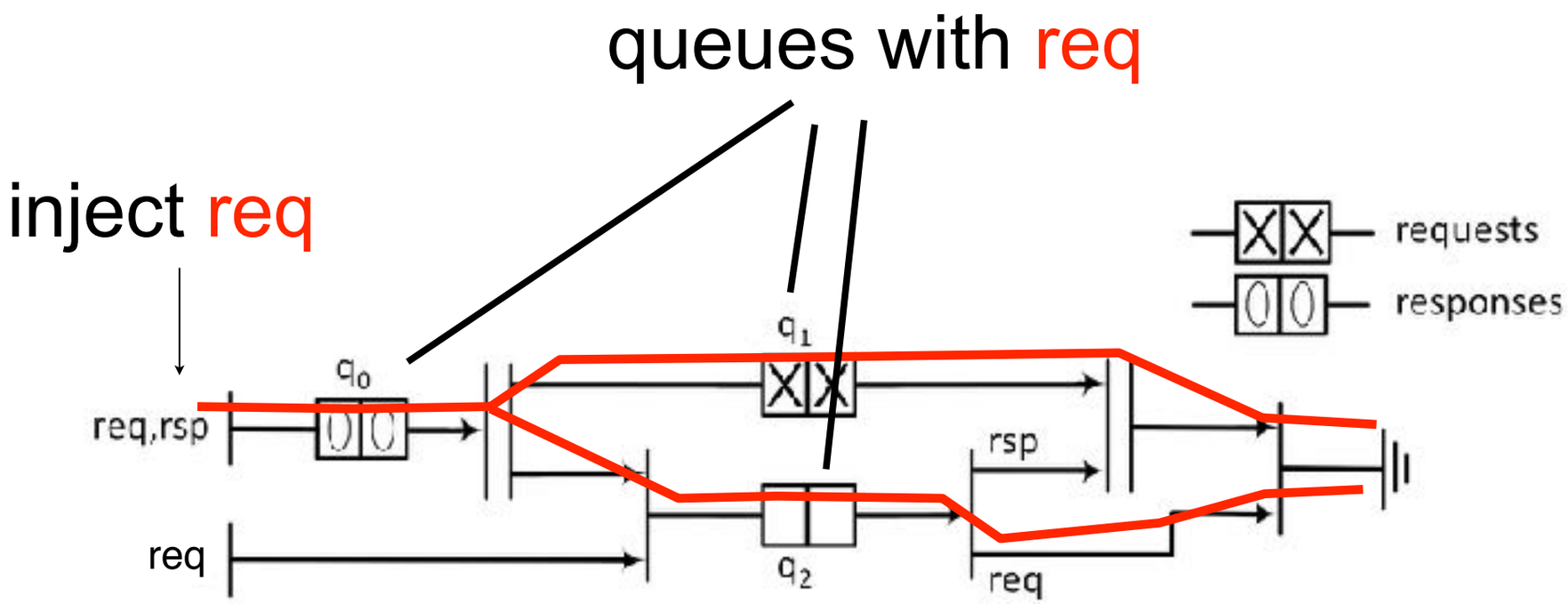
- Define deadlock equations for all components
  - –Equations capture the reason why a component is idle or blocking

- Build a labelled waiting graph for each queue
  - –Labels correspond to the equations
  - –Graph captures the topology, i.e., the dependencies between the MaDL components

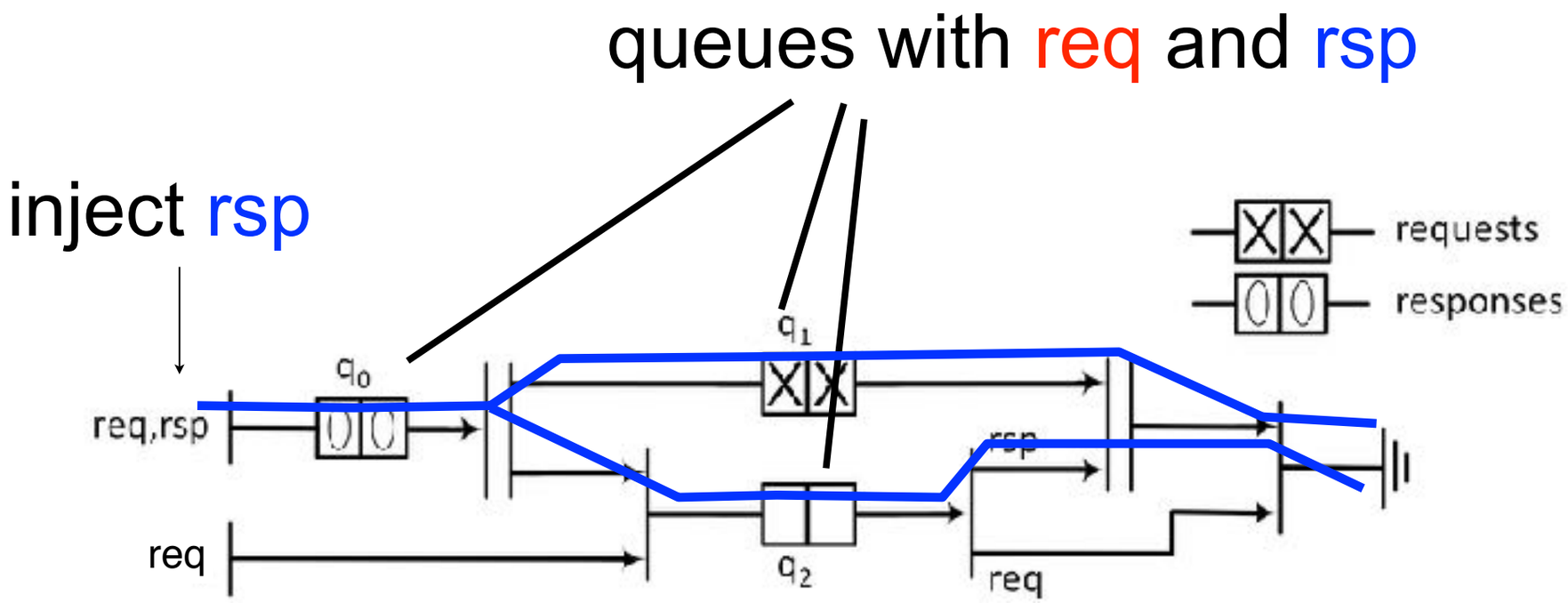
- Search for a feasible logically closed subgraph
  - –Corresponds to a deadlock situation
  - –Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - –A first step generates invariants to rule out false deadlocks
  - –Invariants are rather weak and simple - false deadlocks are in theory still possible

Step 1 / simulation - req



Step 1 / simulation - rsp



General approach for deadlock detection in MaDL networks

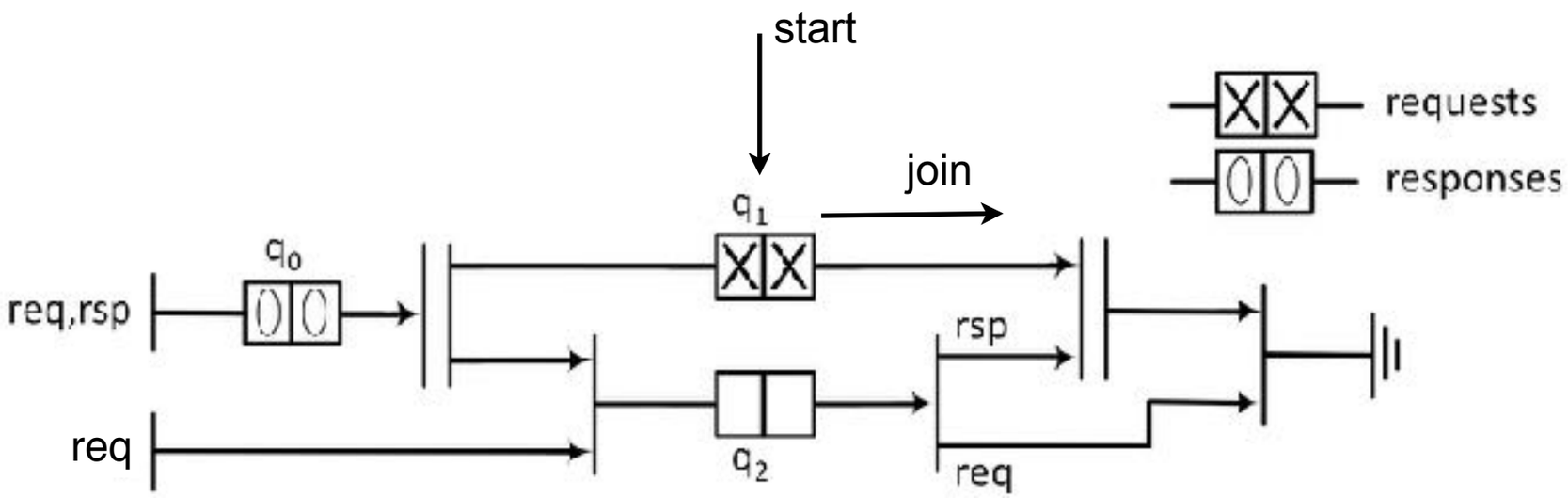
- Define deadlock equations for all components
  - –Equations capture the reason why a component is idle or blocking

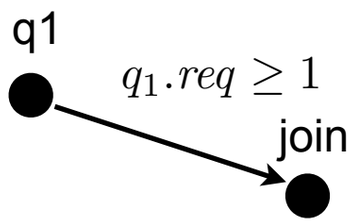
- Build a labelled waiting graph for each queue
  - –Labels correspond to the equations
  - –Graph captures the topology, i.e., the dependencies between the MaDL components

- Search for a feasible logically closed subgraph
  - –Corresponds to a deadlock situation
  - –Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - –A first step generates invariants to rule out false deadlocks
  - –Invariants are rather weak and simple - false deadlocks are in theory still possible

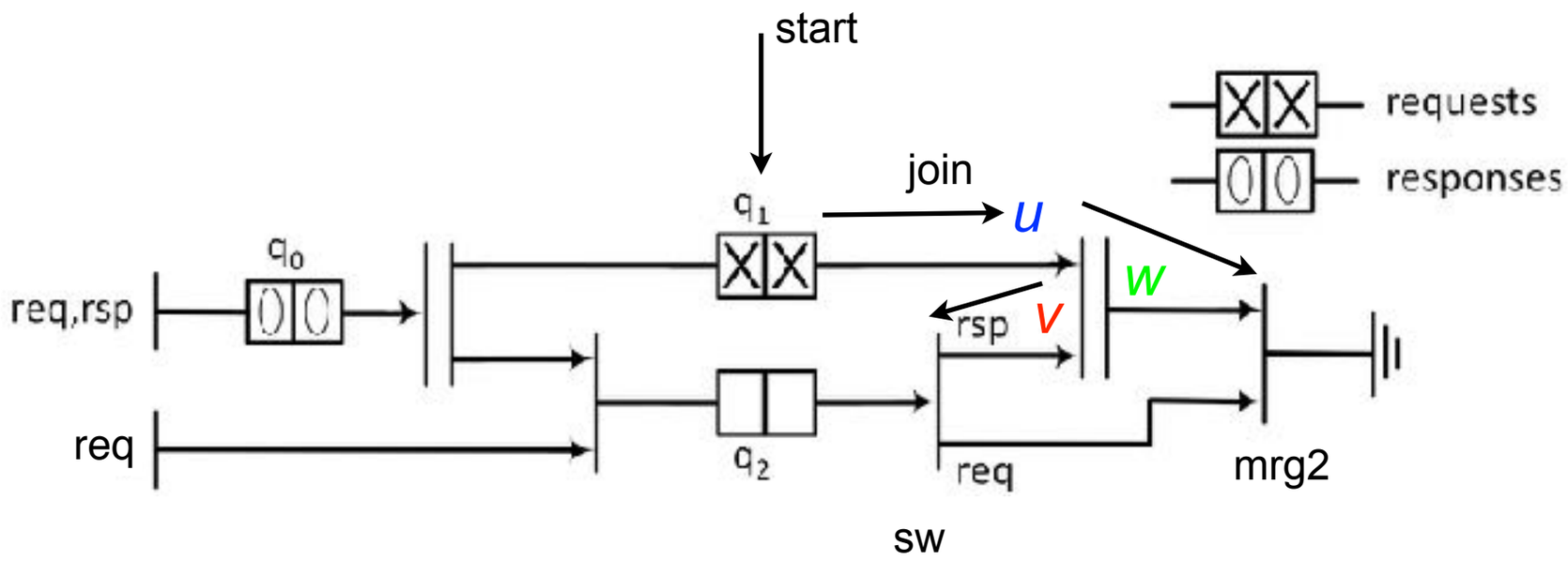
Step 2 / labelled dependency graph (1)



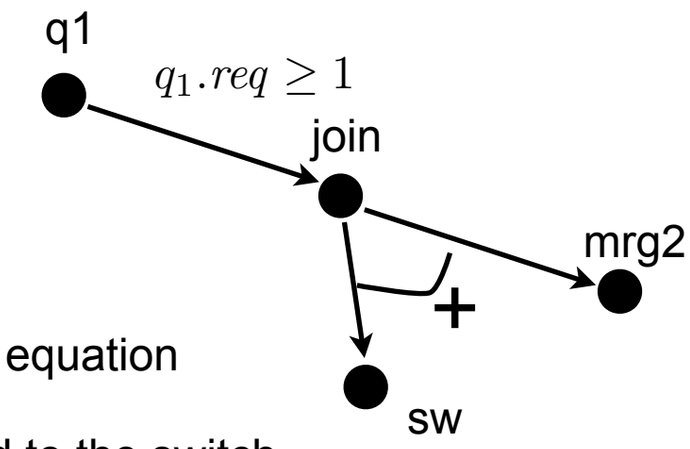


start with a message in q1 and visit the join

Step 2 / labelled dependency graph (2)



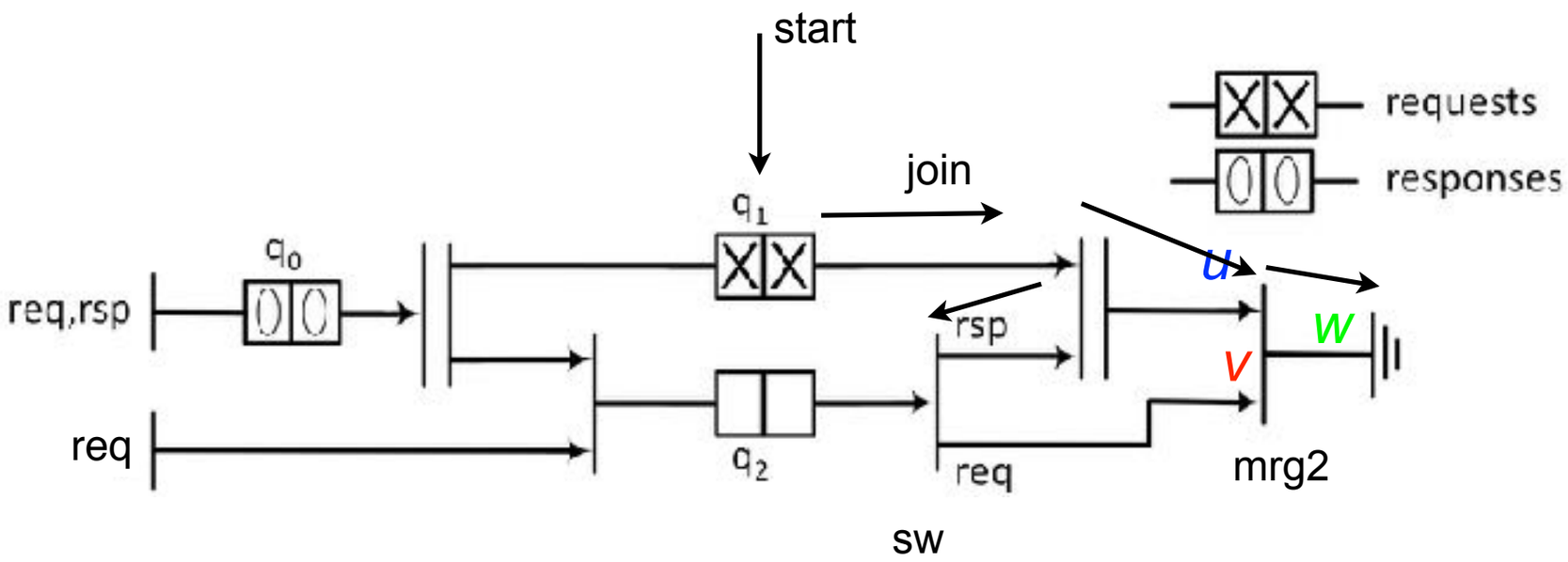
Block(u) = **Idle**(v) + **Block**(w)



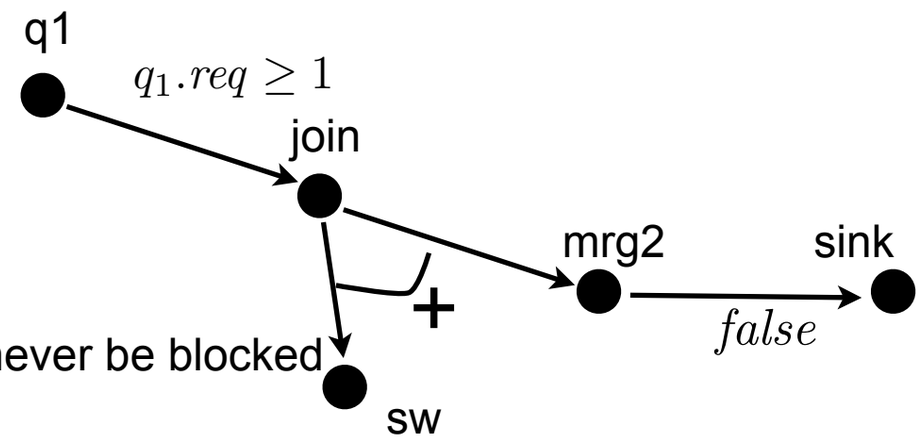
analyse the join according to its deadlock equation

we go forward to the merge and backward to the switch

Step 2 / labelled dependency graph (2)



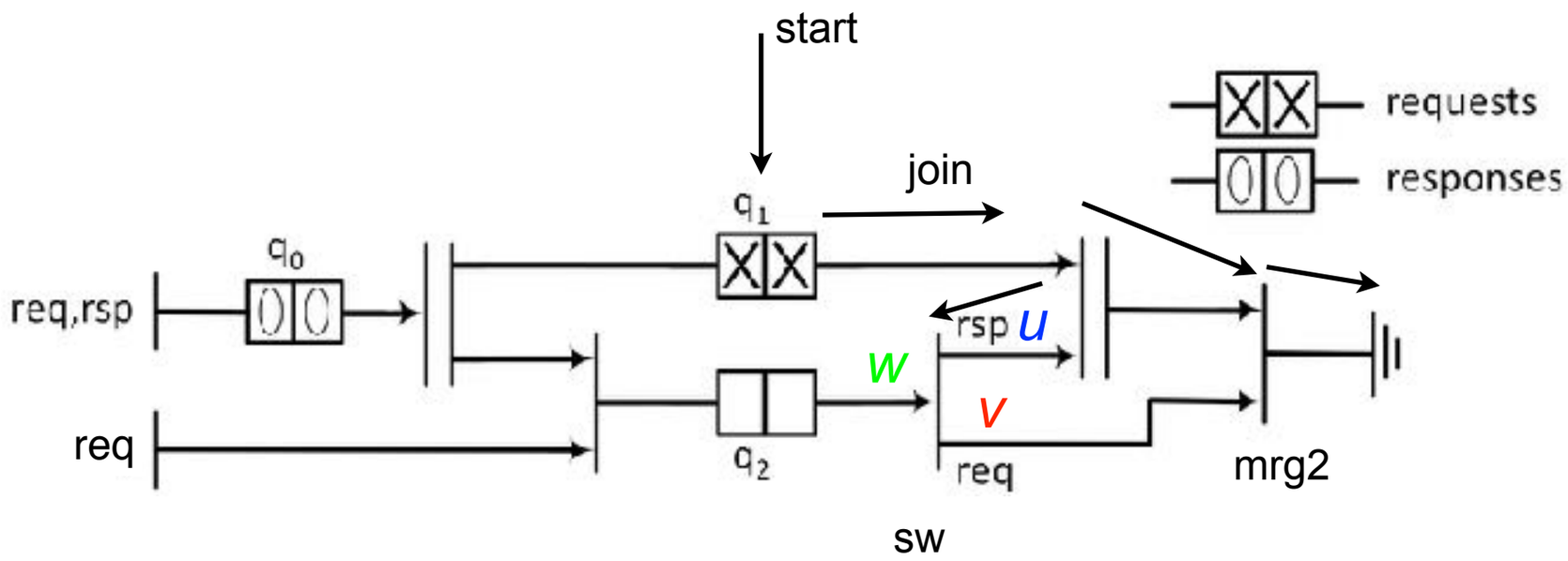
Block(u) = **Block**(w)



forwards to the switch - then the sink can never be blocked

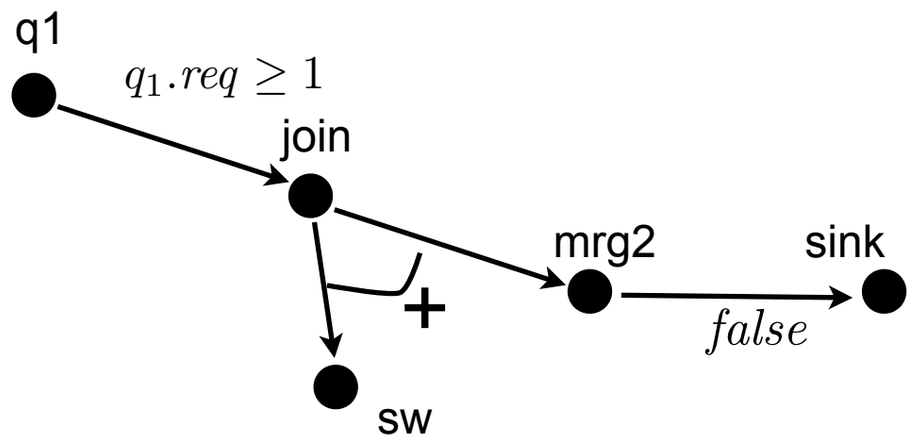
we assume fair sinks

Step 2 / labelled dependency graph (2)

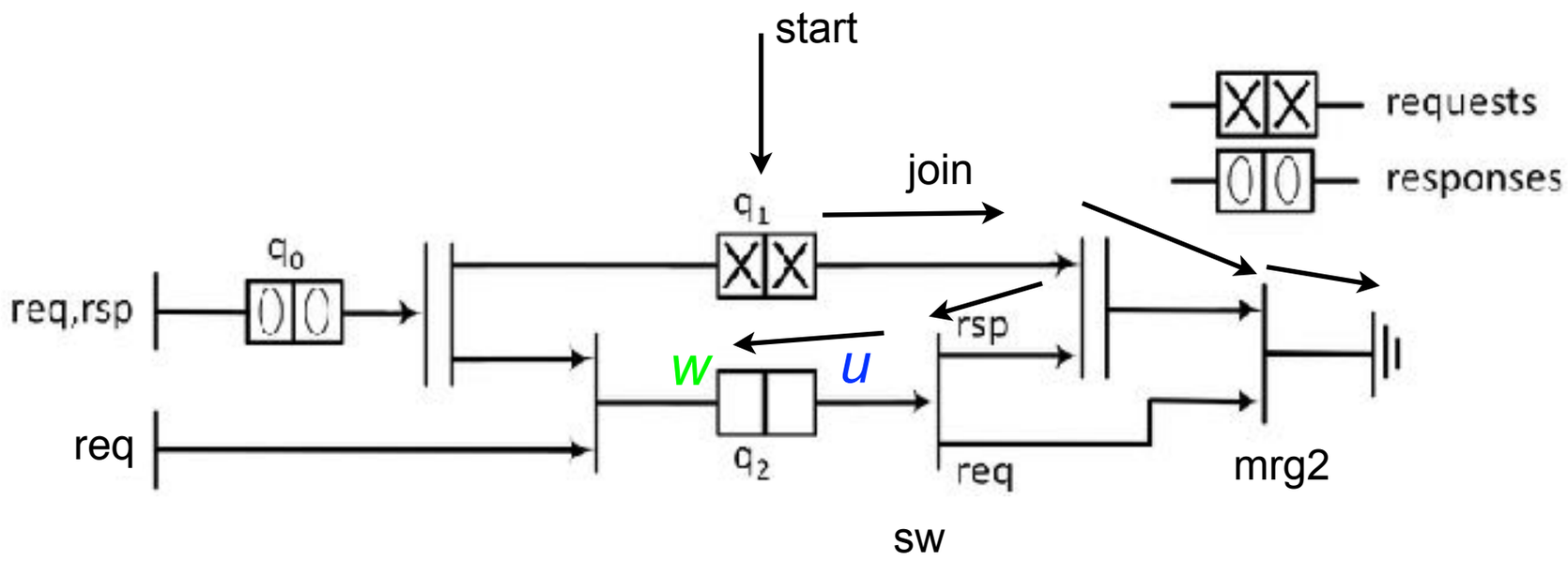


Idle(u) = **Idle**(w)

backwards to the switch



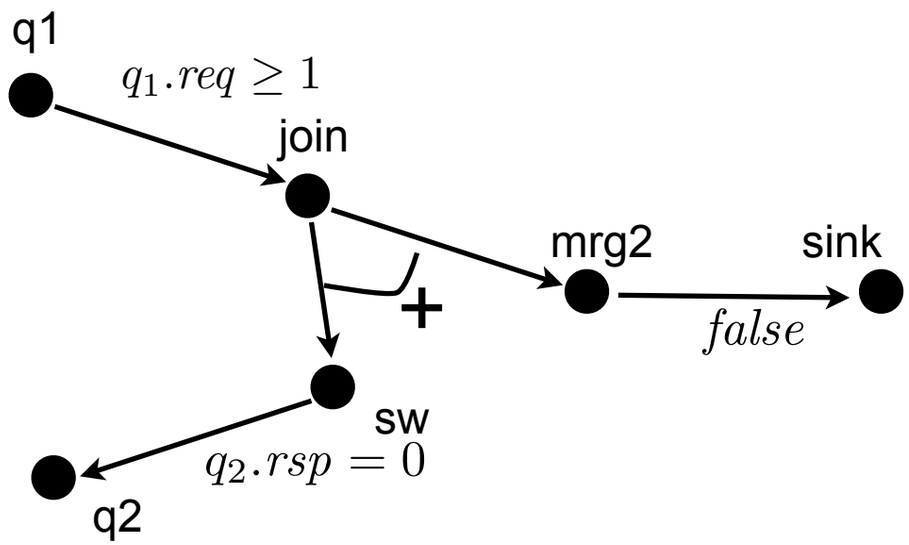
Step 2 / labelled dependency graph (2)



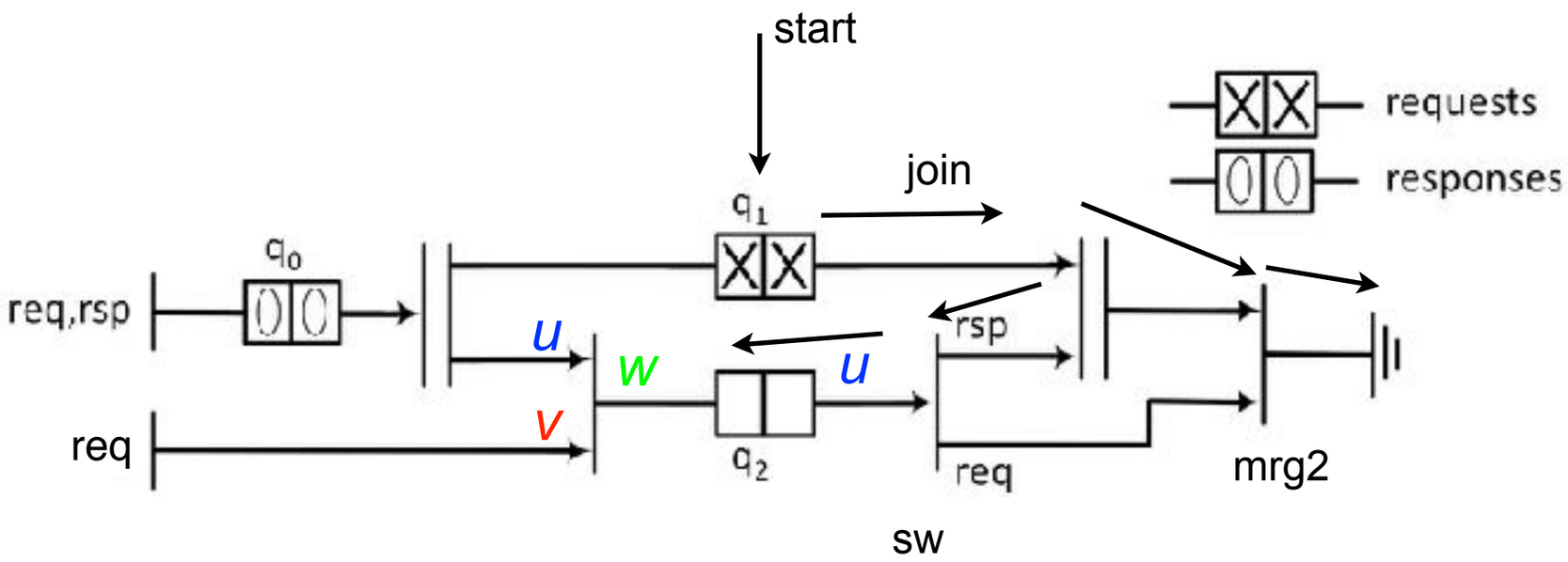
Idle(u) = **Idle**(w) . **Empty**(q2)

backwards to the queue

note that we forgot the **Block**(w') case



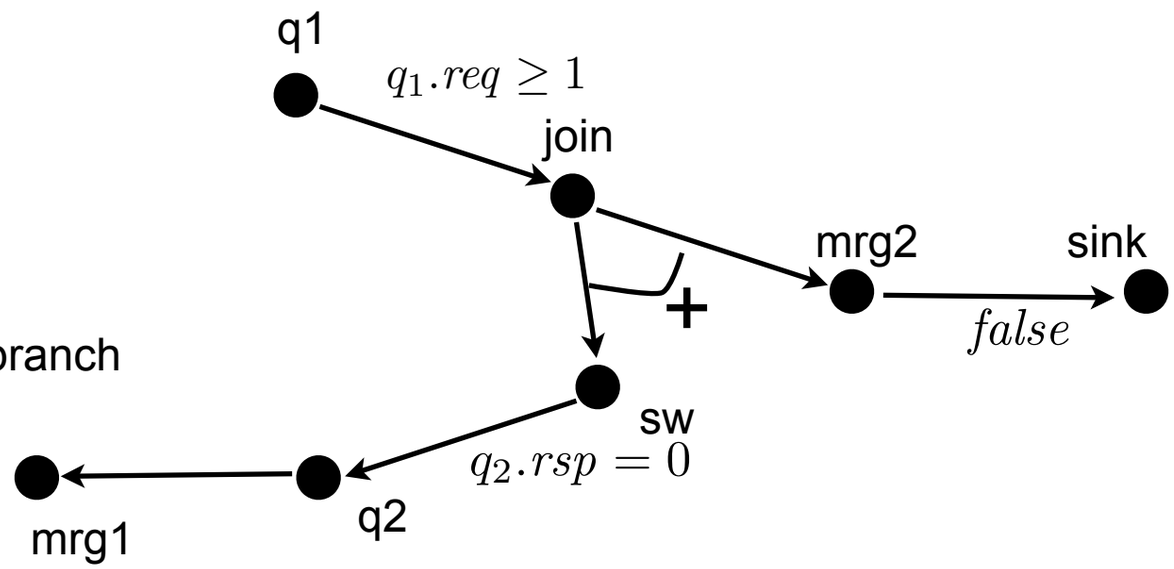
Step 2 / labelled dependency graph (2)



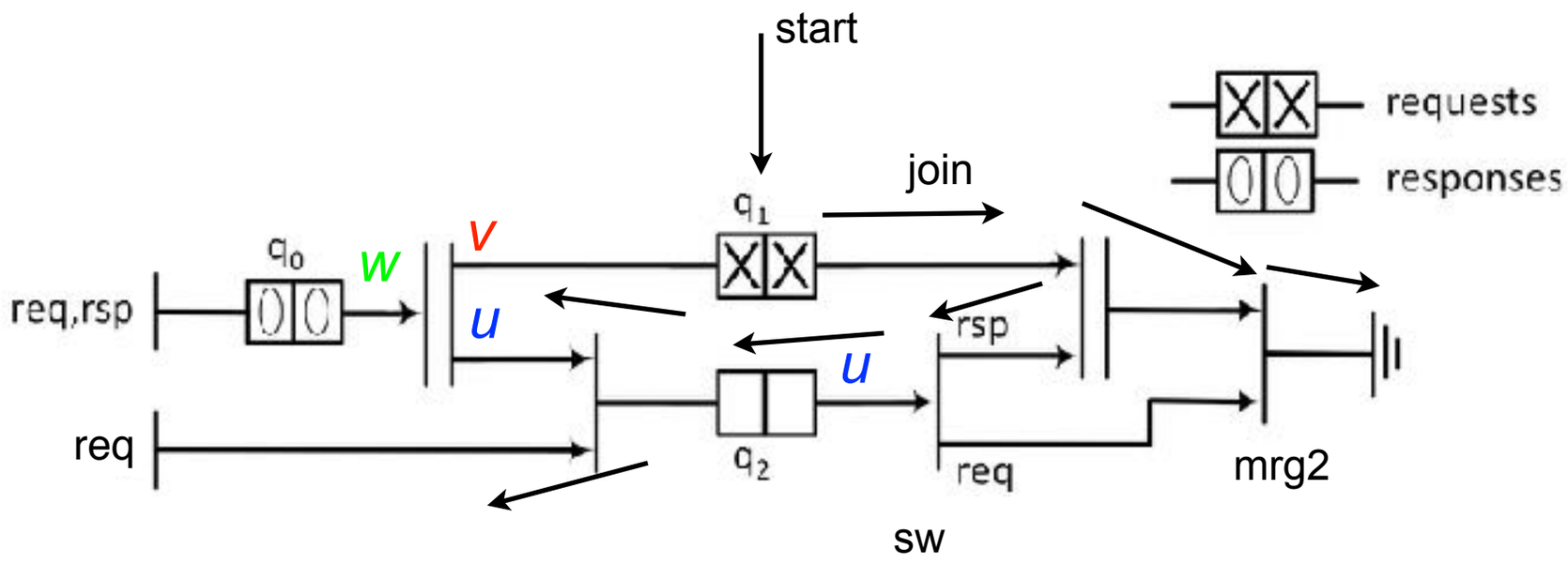
Idle(w) = **Idle**(u) . **Idle**(v)

backwards to the merge and branch

note branching is bad for us



Step 2 / labelled dependency graph (2)

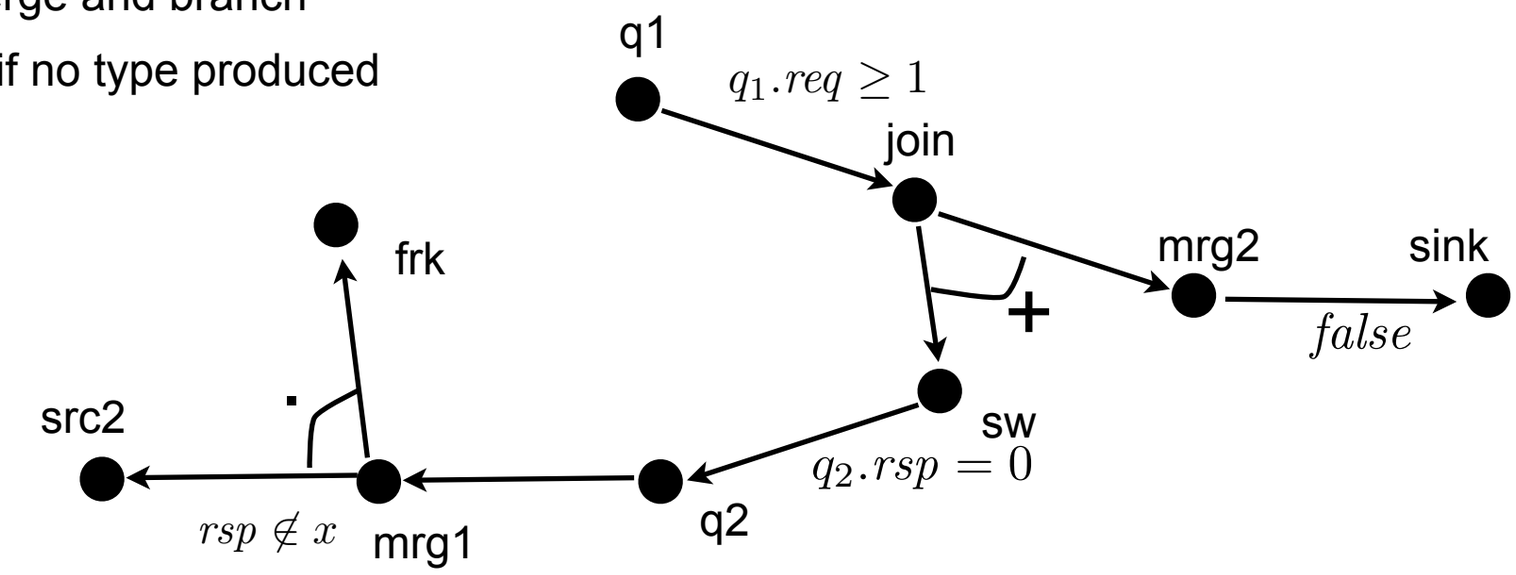


Idle(u) = **Block**(v) + **Idle**(w)

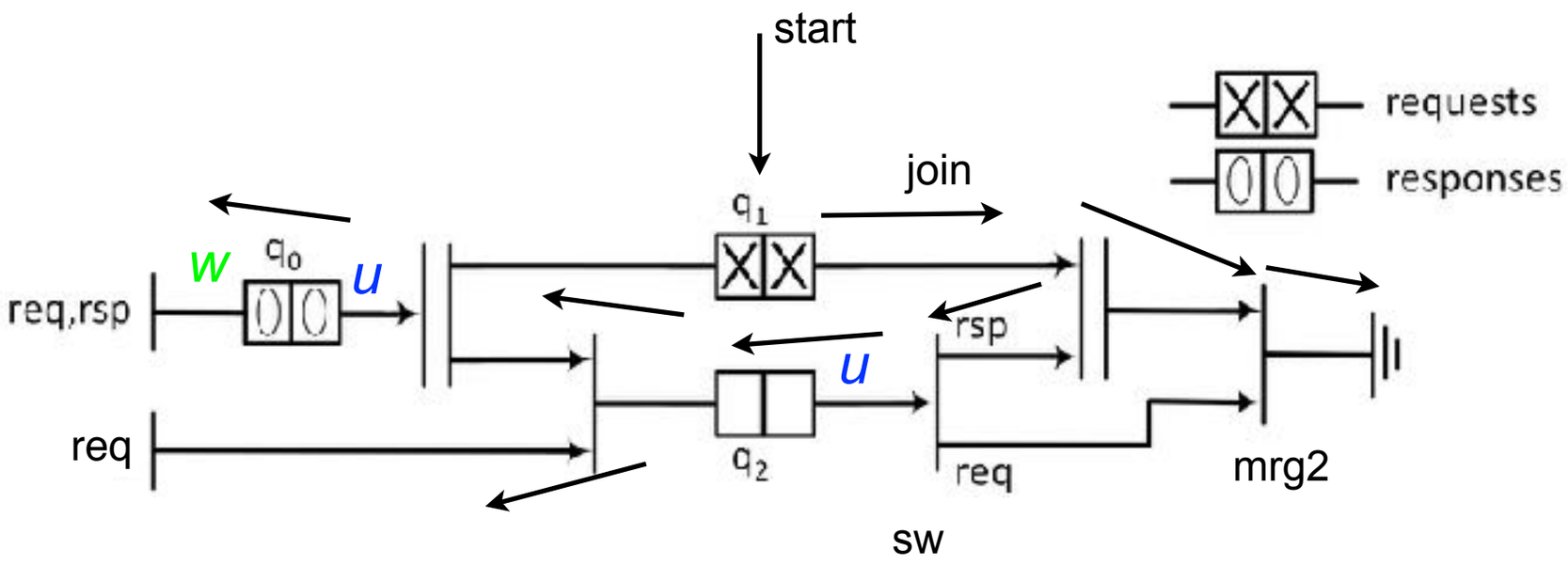
backwards to the merge and branch

to the source - idle if no type produced

to the fork

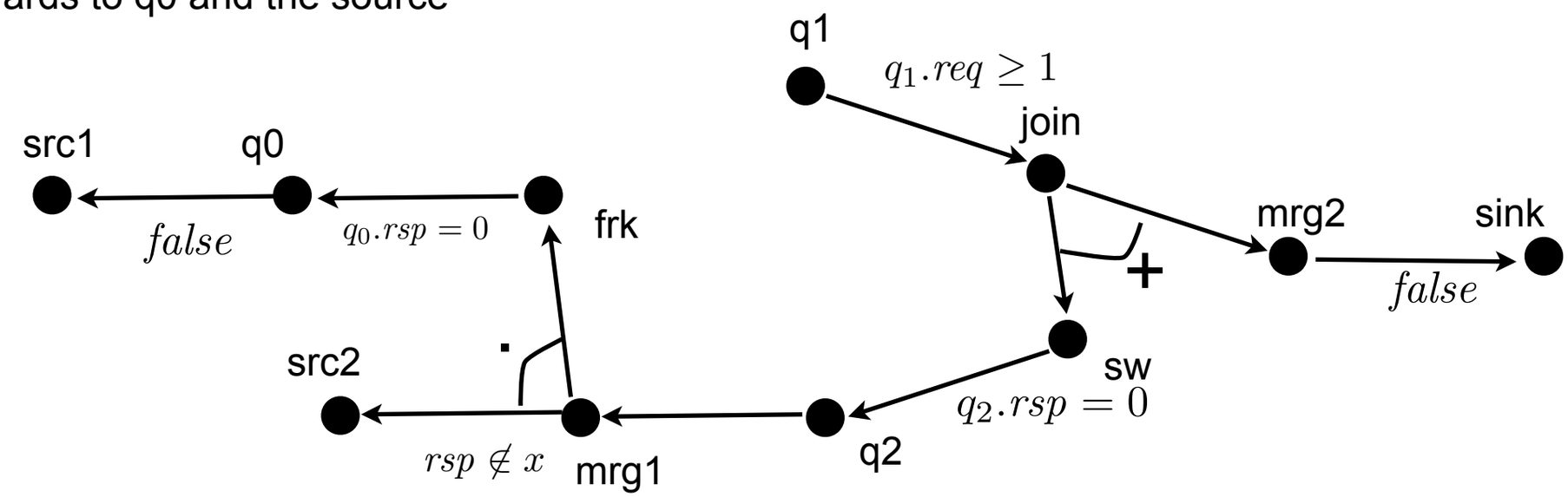


Step 2 / labelled dependency graph (2)

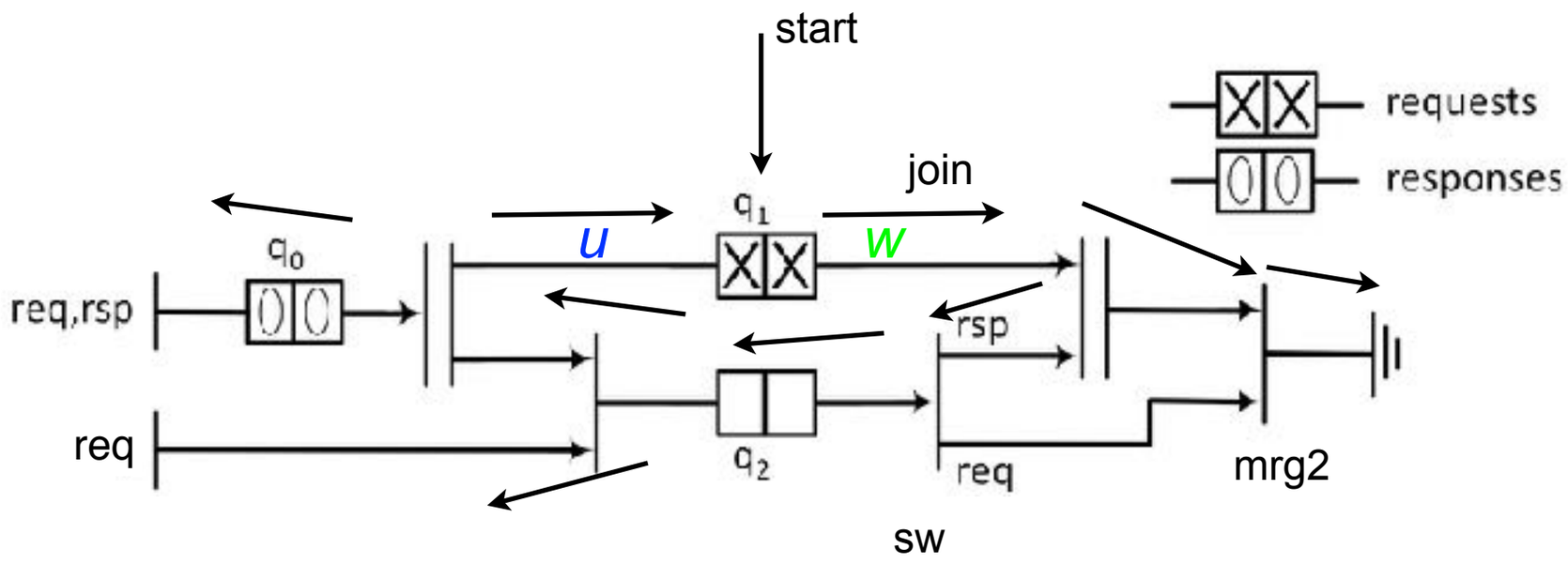


Idle(u) = **Idle**(w) . **Empty**(q0)

backwards to q0 and the source

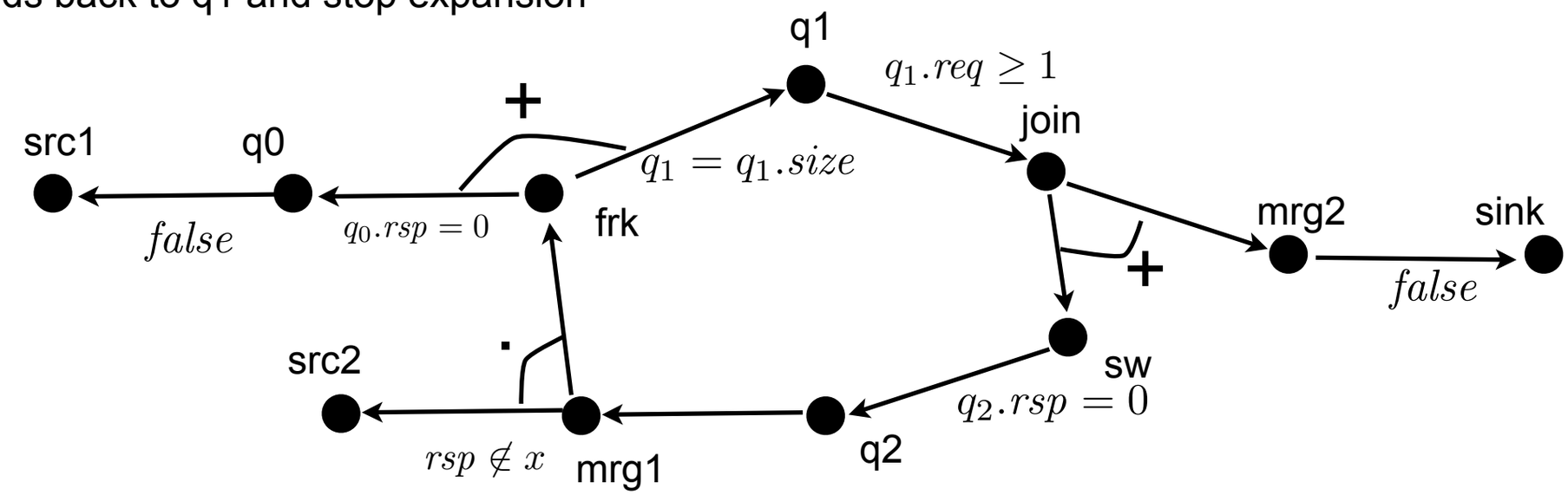


Step 2 / labelled dependency graph (2)



Block(u) = **Block**(w) . **Full**(q1)

forwards back to q1 and stop expansion



General approach for deadlock detection in MaDL networks

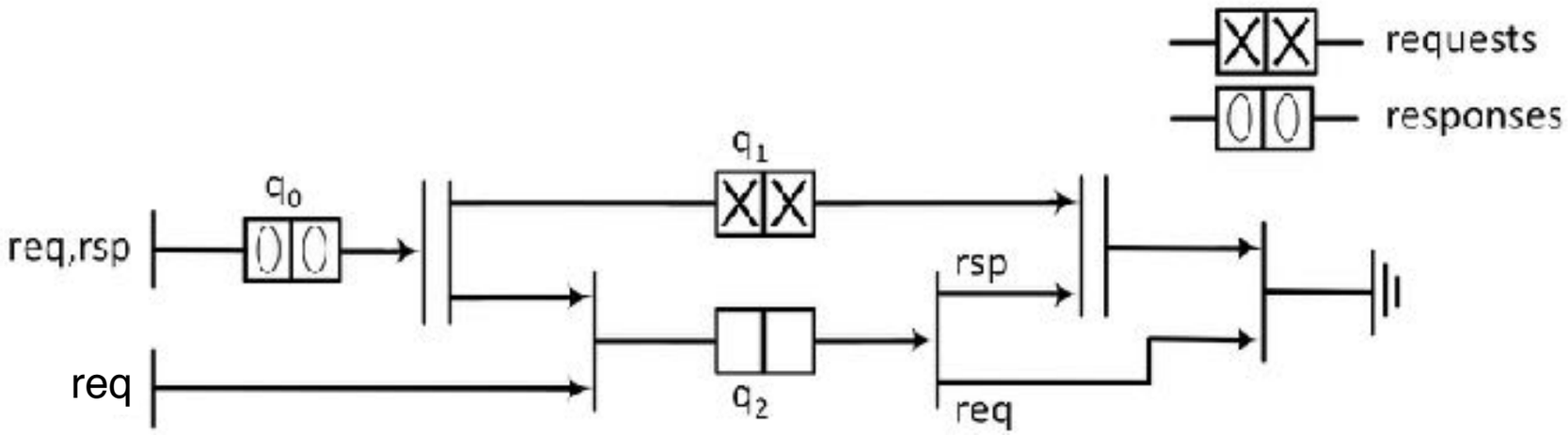
- Define deadlock equations for all components
  - –Equations capture the reason why a component is idle or blocking

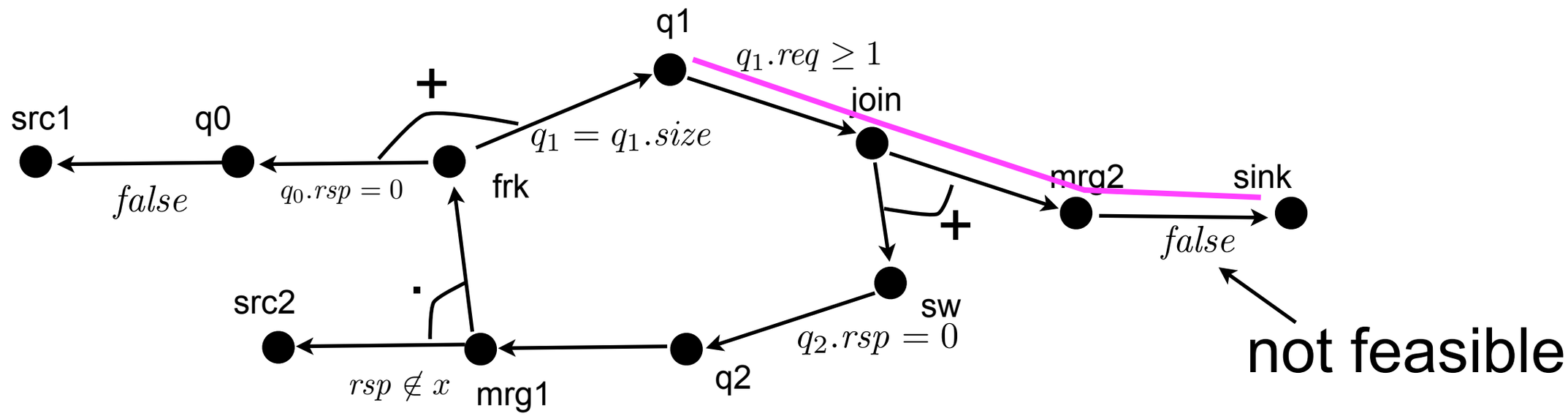
- Build a labelled waiting graph for each queue
  - –Labels correspond to the equations
  - –Graph captures the topology, i.e., the dependencies between the MaDL components

- Search for a feasible logically closed subgraph
  - –Corresponds to a deadlock situation
  - –Feasibility checked using Linear Programming

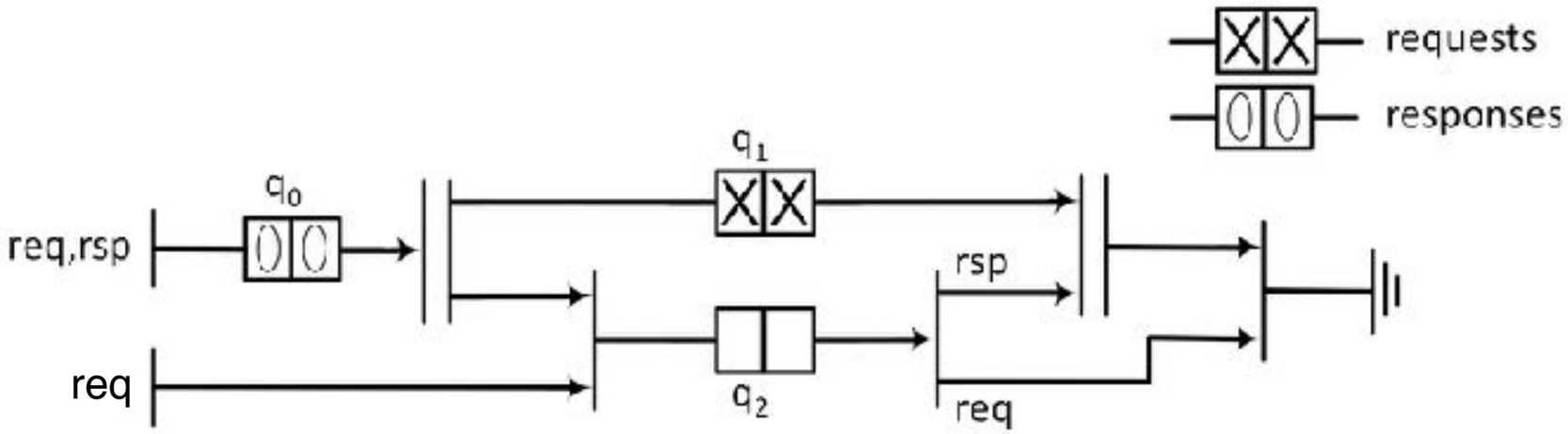
- This approach may output unreachable deadlocks
  - –A first step generates invariants to rule out false deadlocks
  - –Invariants are rather weak and simple - false deadlocks are in theory still possible

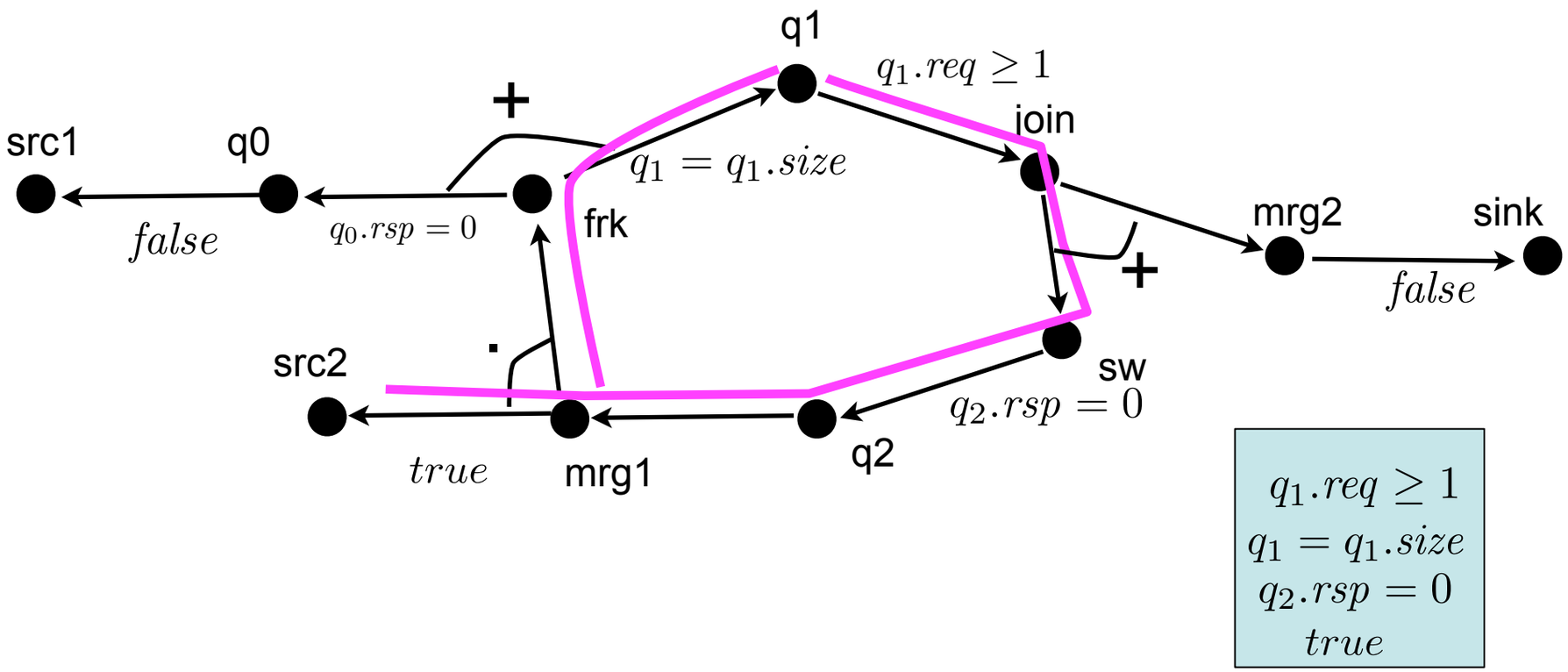
Step 2 / logically closed subgraph 1





Step 2 / logically closed subgraph 2

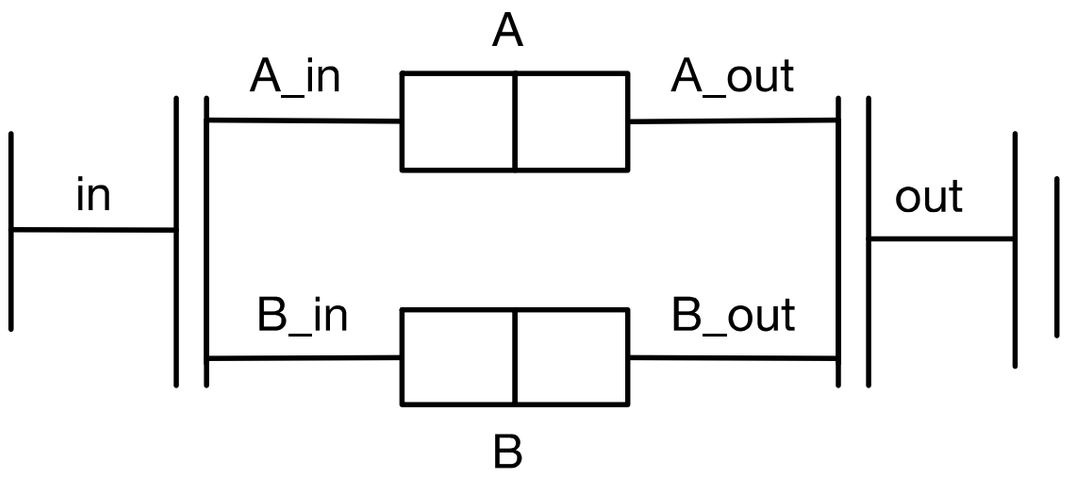




Invariant Generation (1)

Flow invariants:

- Automatically generated
- Linear equations over the # of packets in channels
  - Gaussian elimination
  - Equalities of # of packet between queues
  - NumX = #In - #Out



Linear equations:

- #in = #A_in = #B_in
- #A = #A_in - #A_out
- #B = #B_in - #B_out
- #A_out = #B_out = #out

After Gaussian elimination:

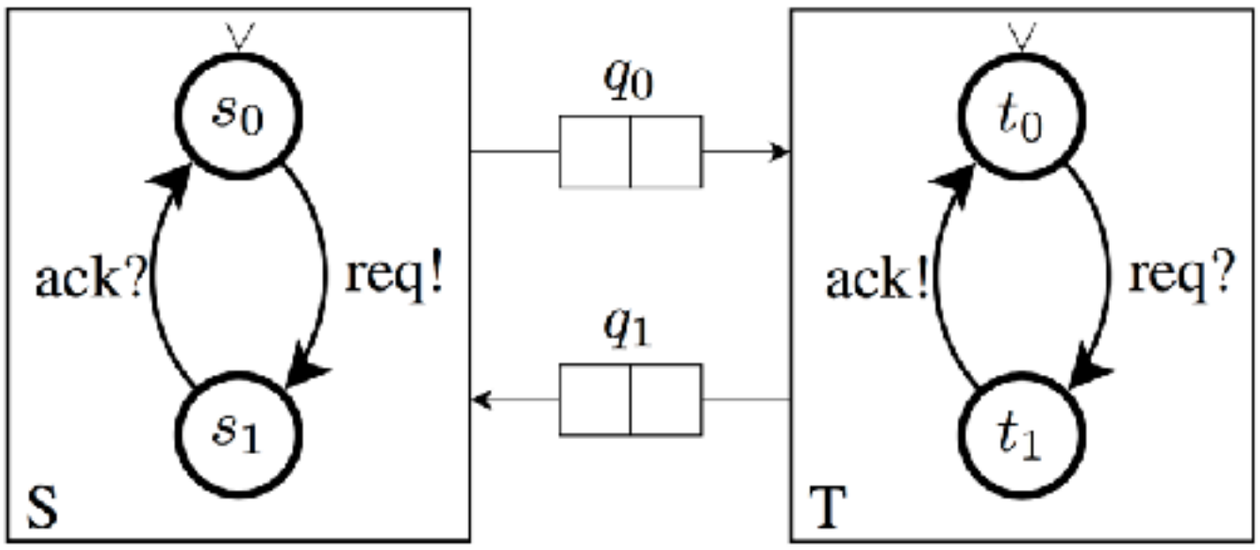
- #A = #B

[Verbeek et al. 2016]

Invariant Generation Highlight (2)

Cross-layer invariants:

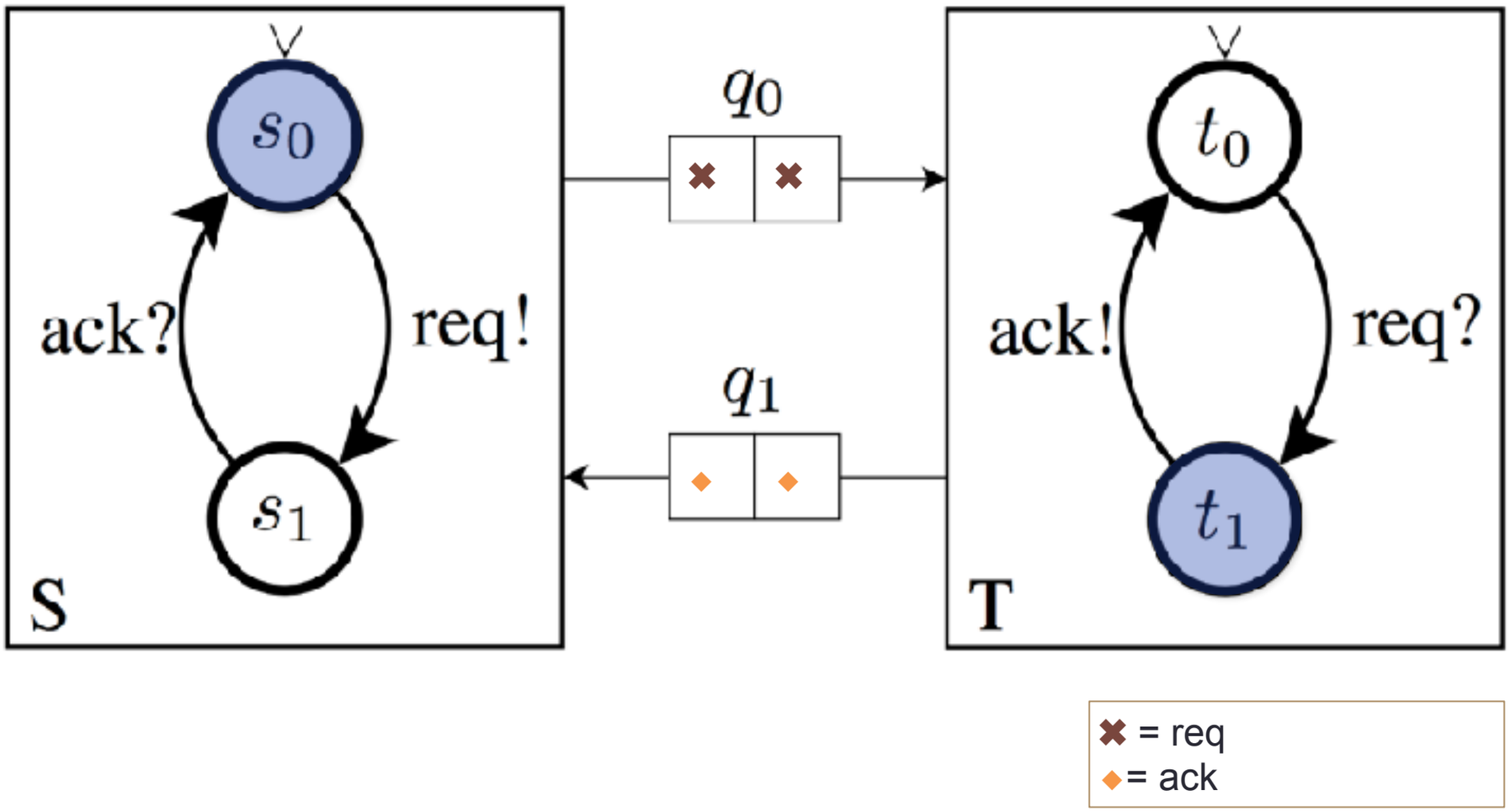
- Automatically generated
- Linear equations over:
  - (1) # transitions and being in a given state
  - (2) # of “events” on a channel and # transitions



$$T.t_0 - S.s_0 = \#q_0.req + \#q_1.ack$$

[Verbeek et al. 2016]

Invariant Generation Highlight (2)



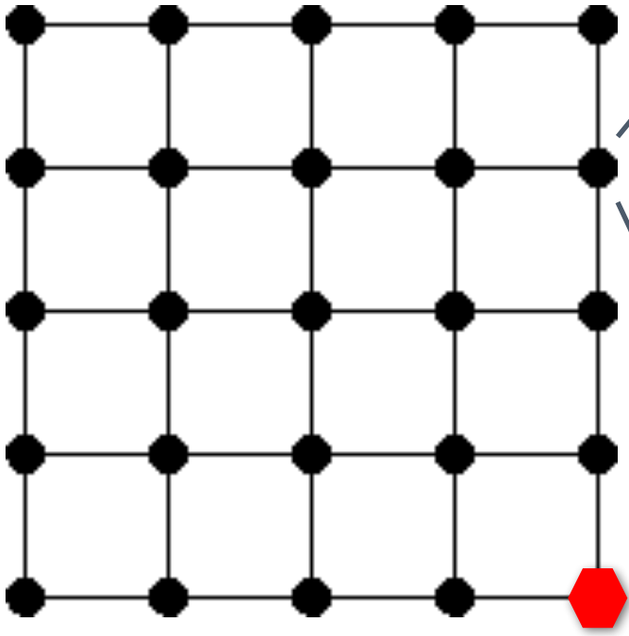
$$T.t_0 - S.s_0 = \#q_0.req + \#q_1.ack$$

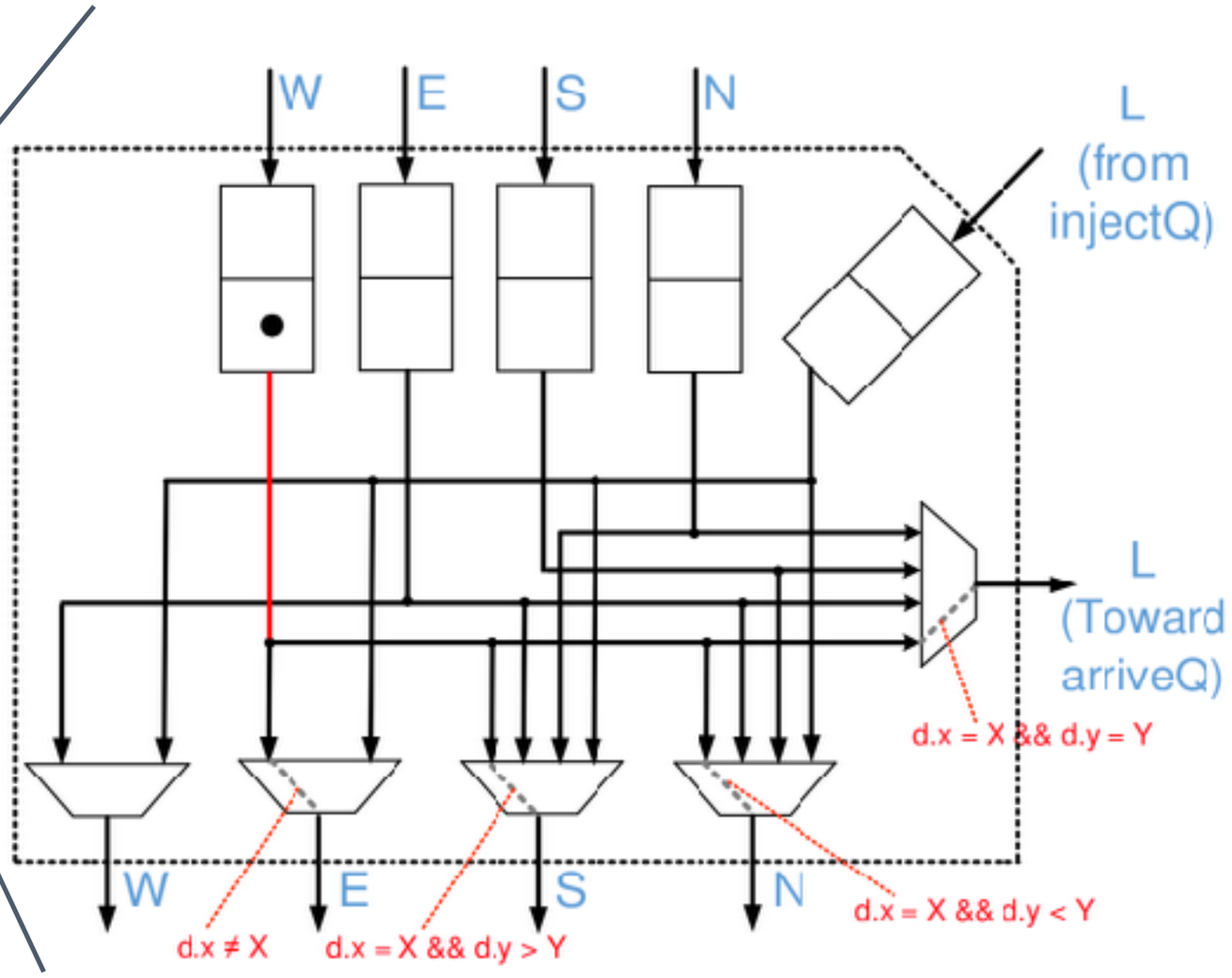
Experimental results

[Verbeek et al. 2016]

Case-study: 2D Mesh - XY routing - MI protocol

Interconnect:

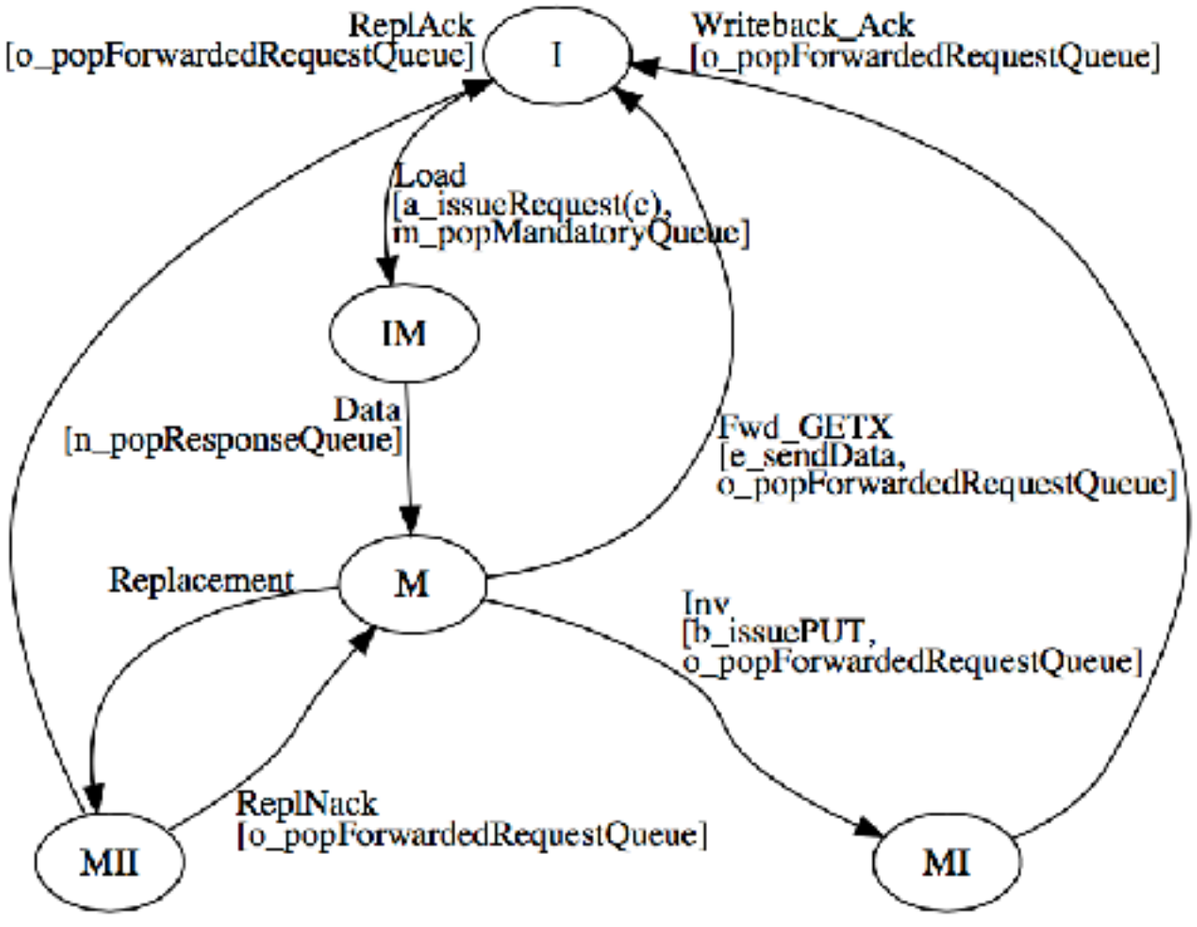




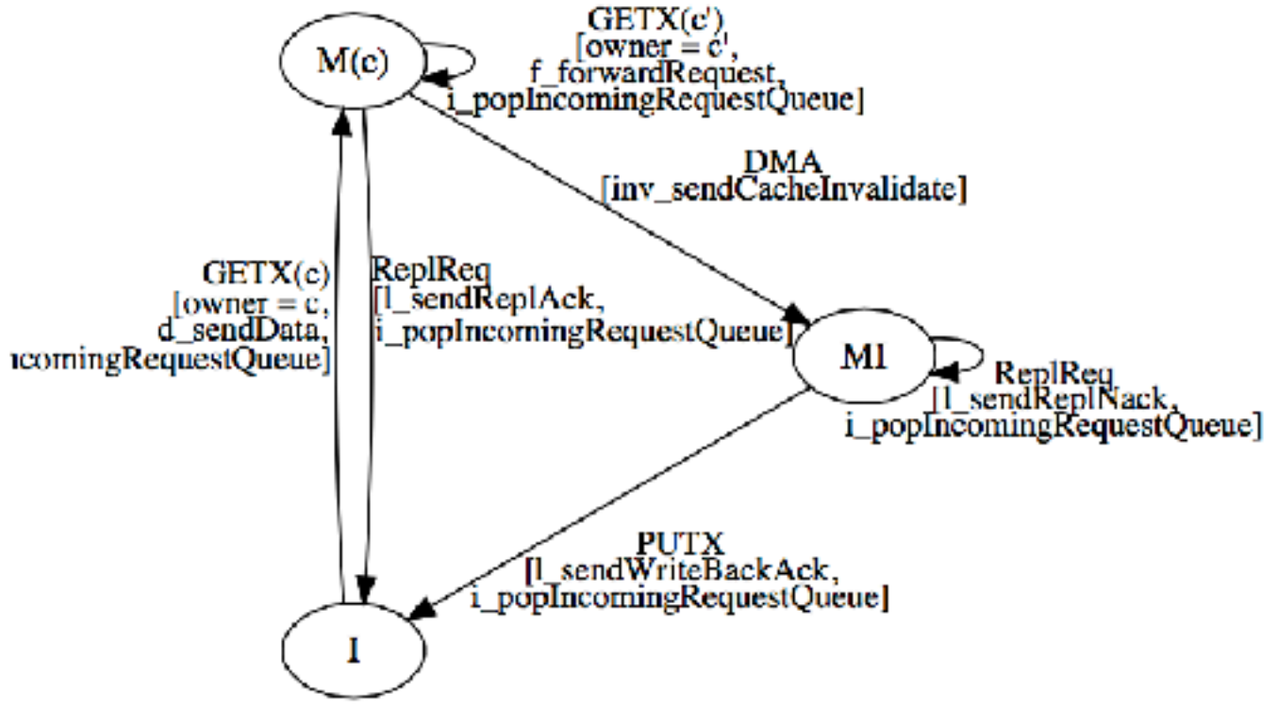
[Verbeek et al. 2016]

Case-study: 2D Mesh - XY routing - MI protocol

Protocol:



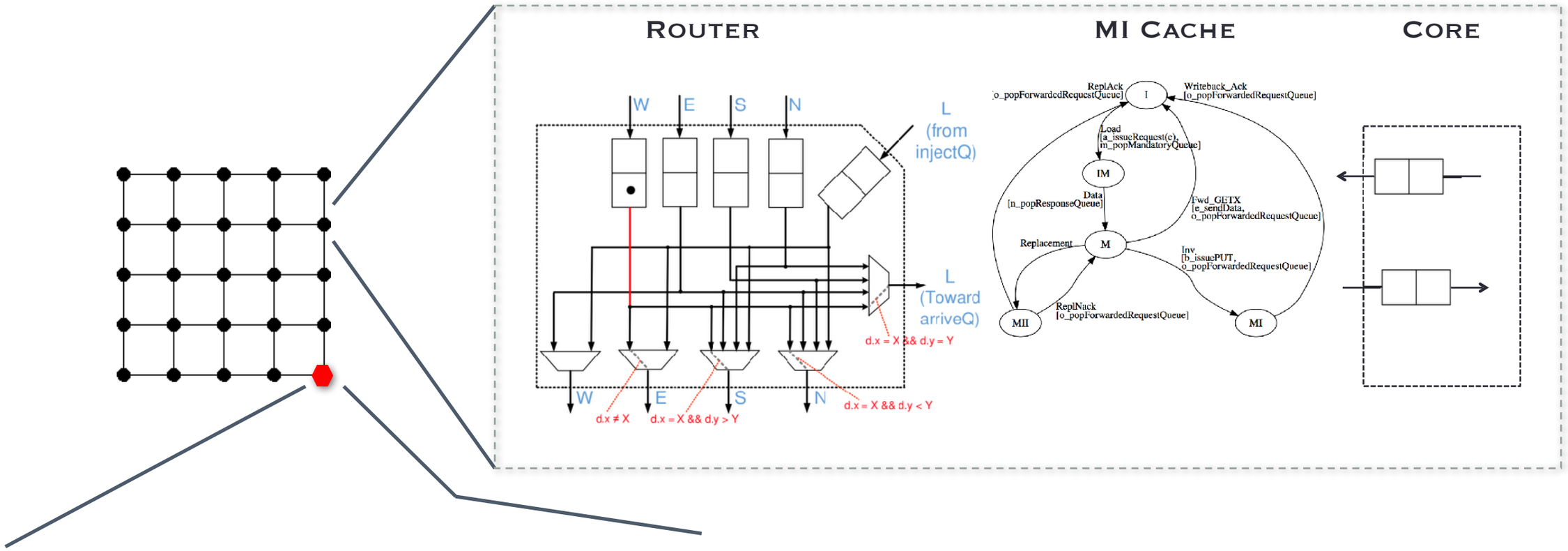
L2 caches

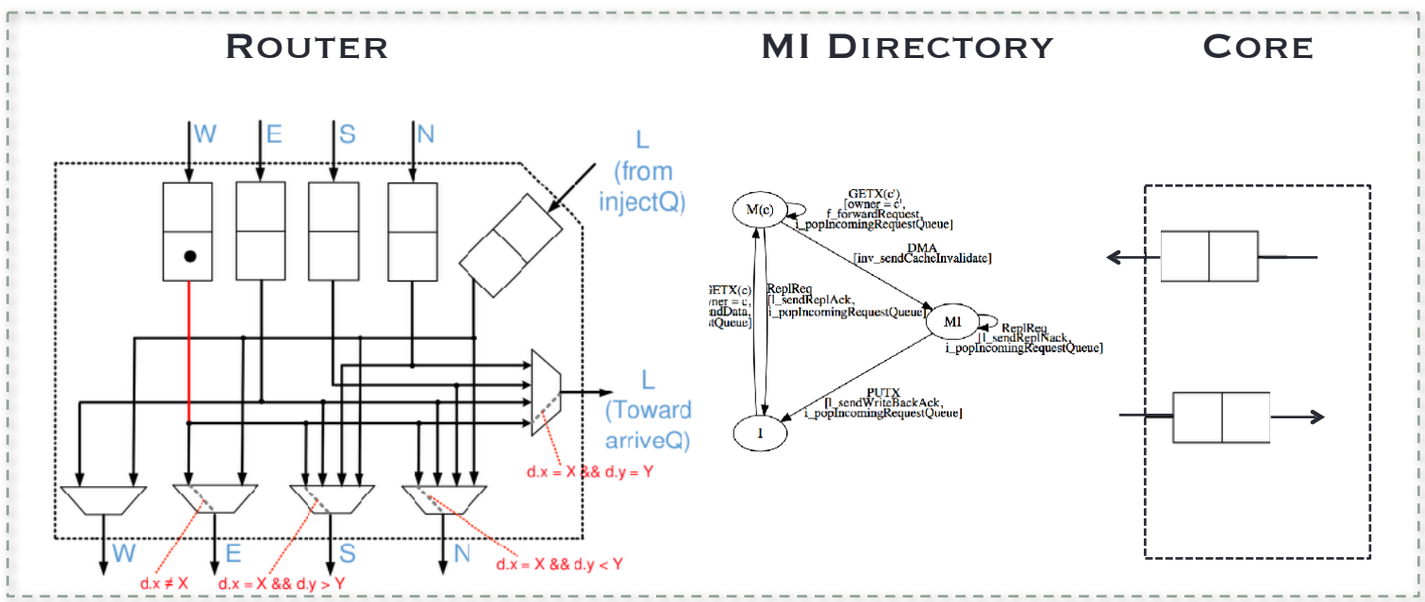


Directory

[Verbeek et al. 2016]

Case-study: 2D Mesh - XY routing - MI protocol





[Verbeek et al. 2016]

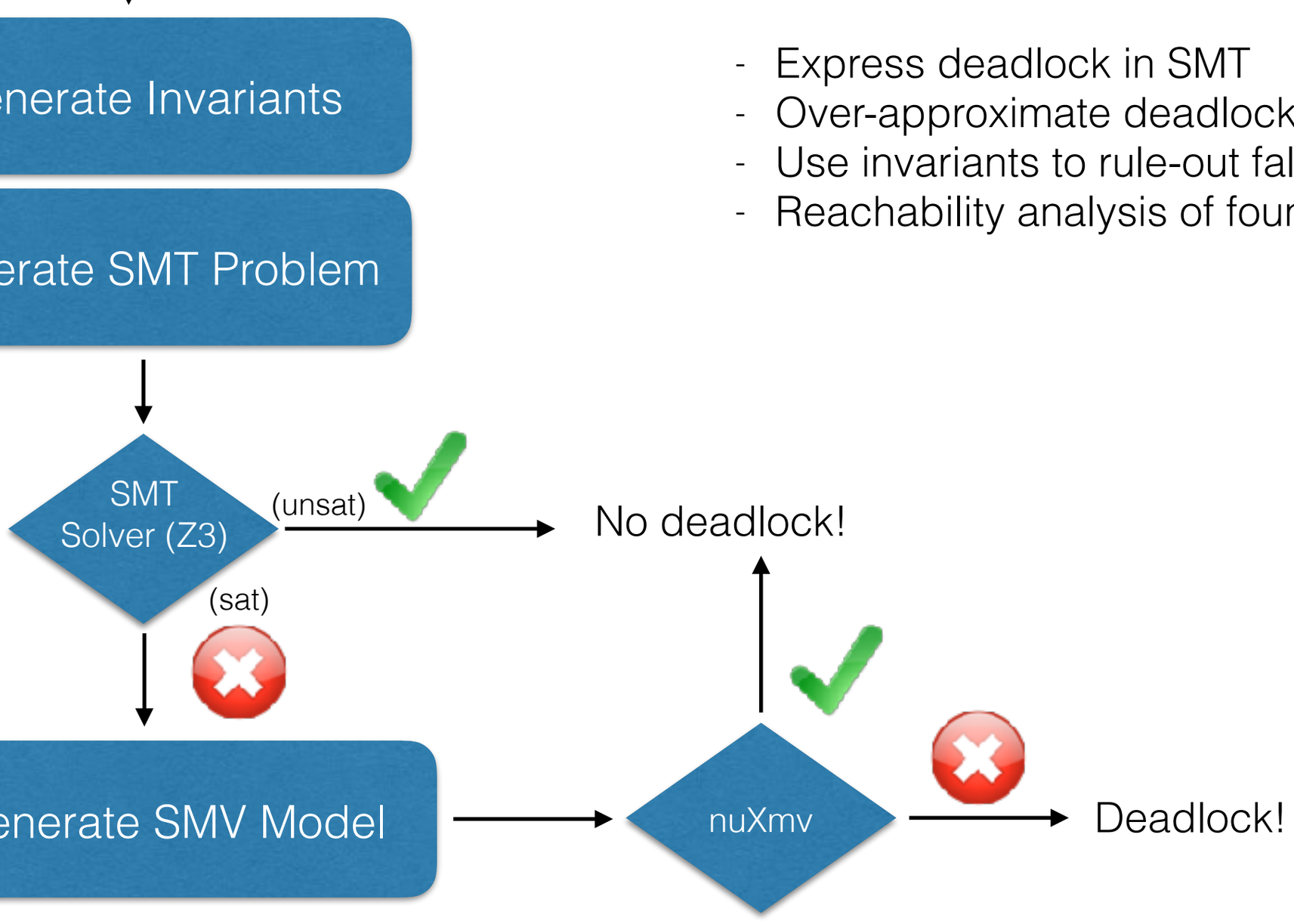
Case-study: Experimental results

Size	DL	DLF	#primitives	#queues	#automata
2×2	1.7s	1.3s	100	24	4
3×3	23s	16s	225	54	9
4×4	3m52s	2m41s	400	96	16
5×5	33m18	23m5s	625	150	25

Reachability analysis

MaDL Verification: Overview

Generate Invariants

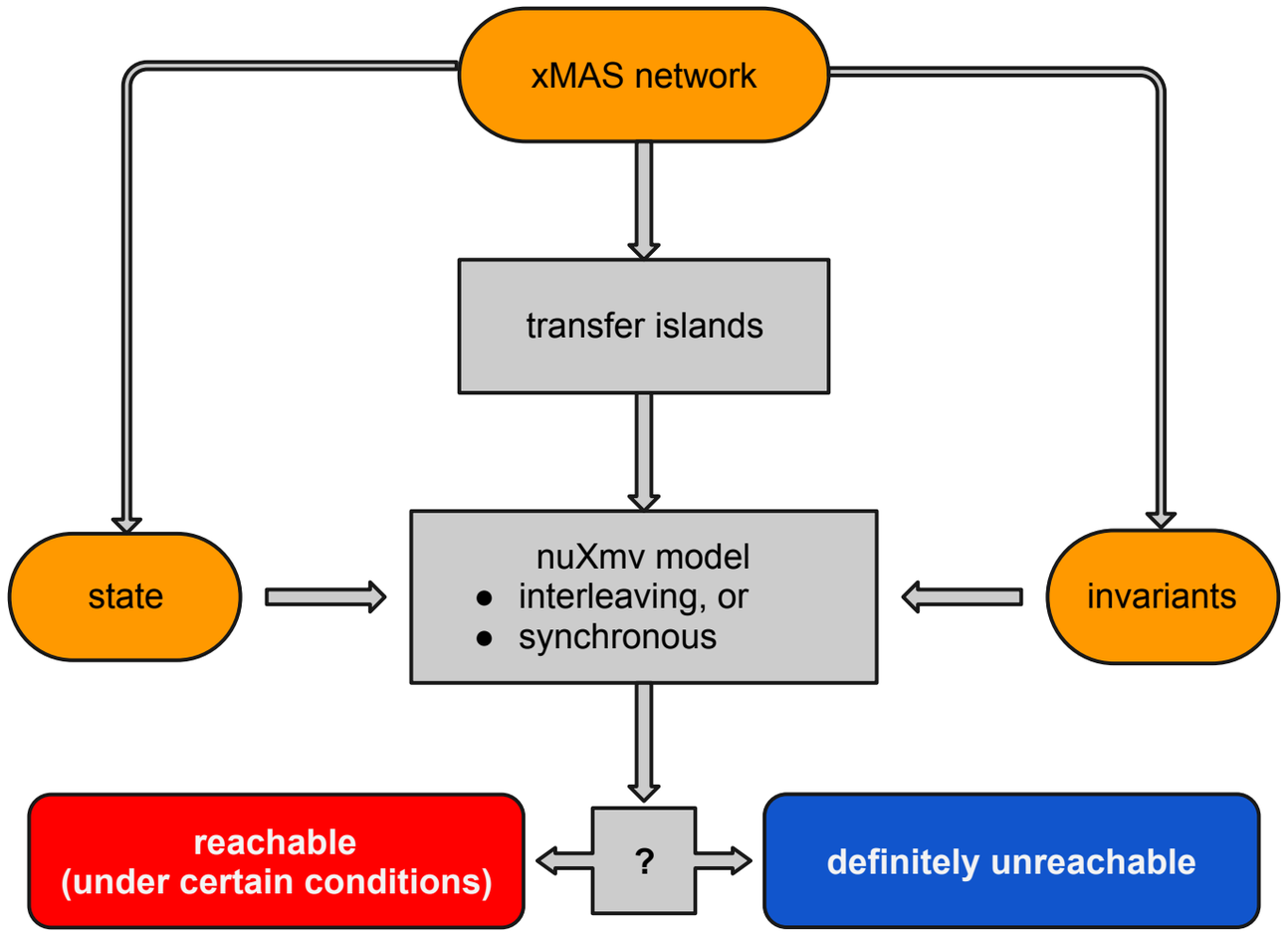


Basic idea:

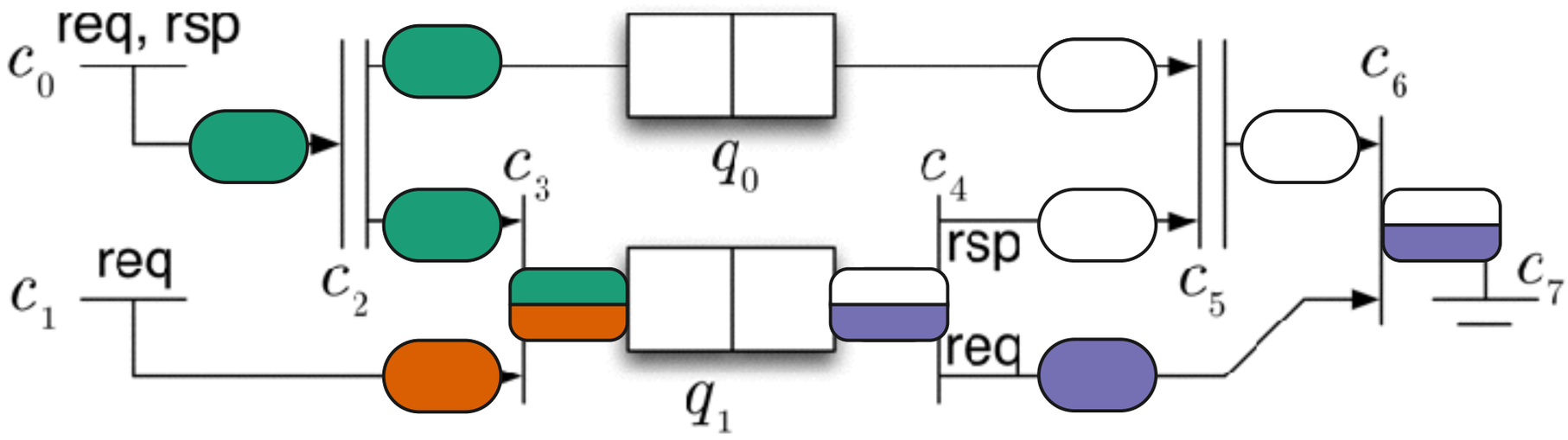
- Express deadlock in SMT
- Over-approximate deadlocks
- Use invariants to rule-out false deadlocks
- Reachability analysis of found deadlocks

[Wouda and Schmaltz 2015]

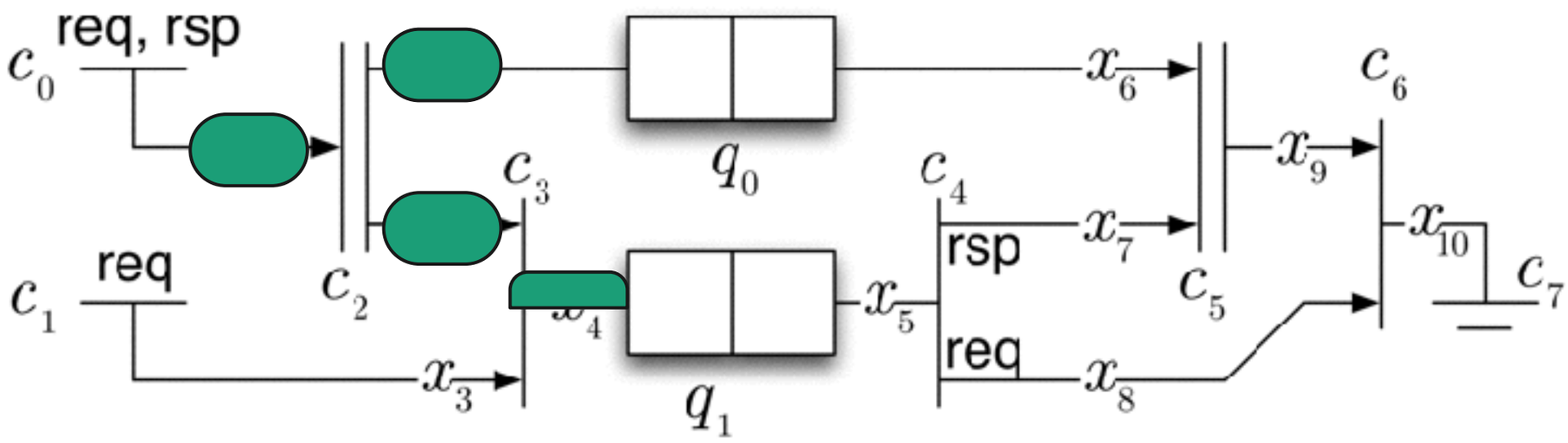
Reachability checking flow



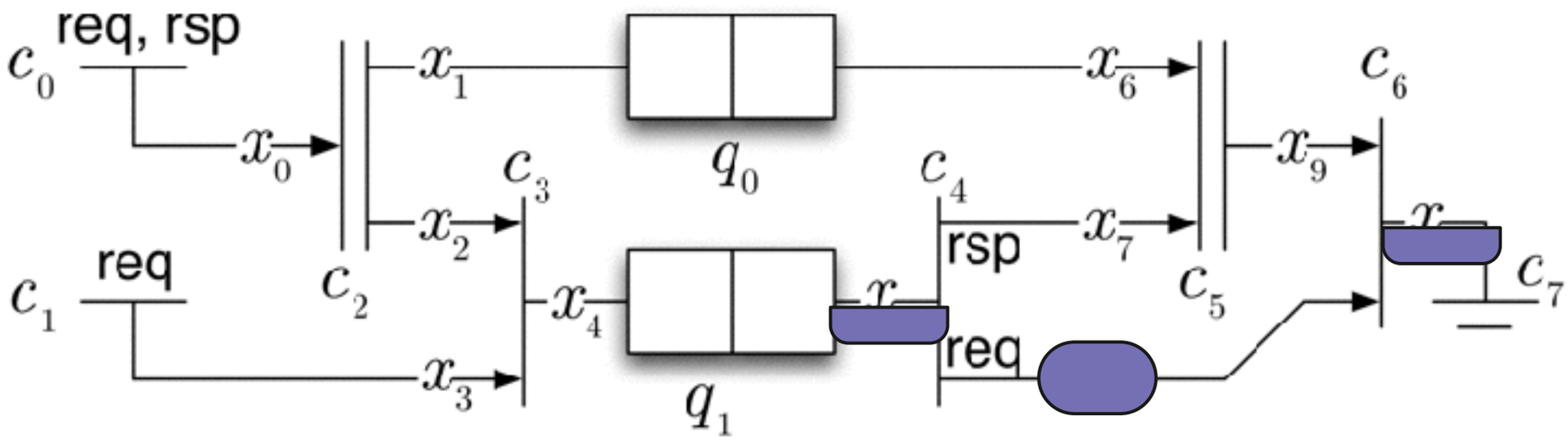
Transfer islands



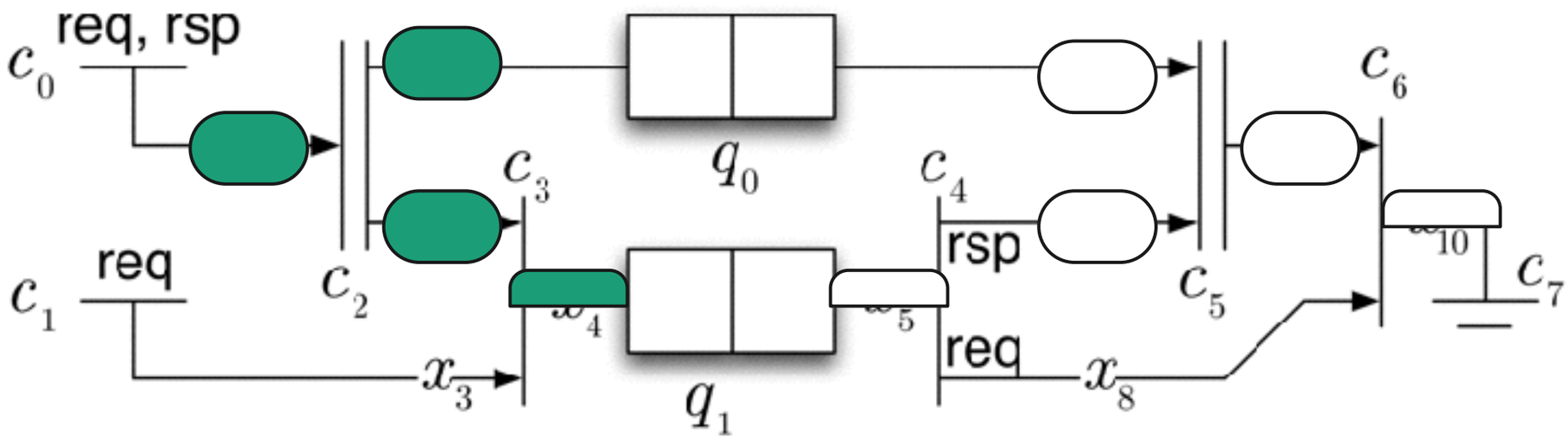
Enabled island



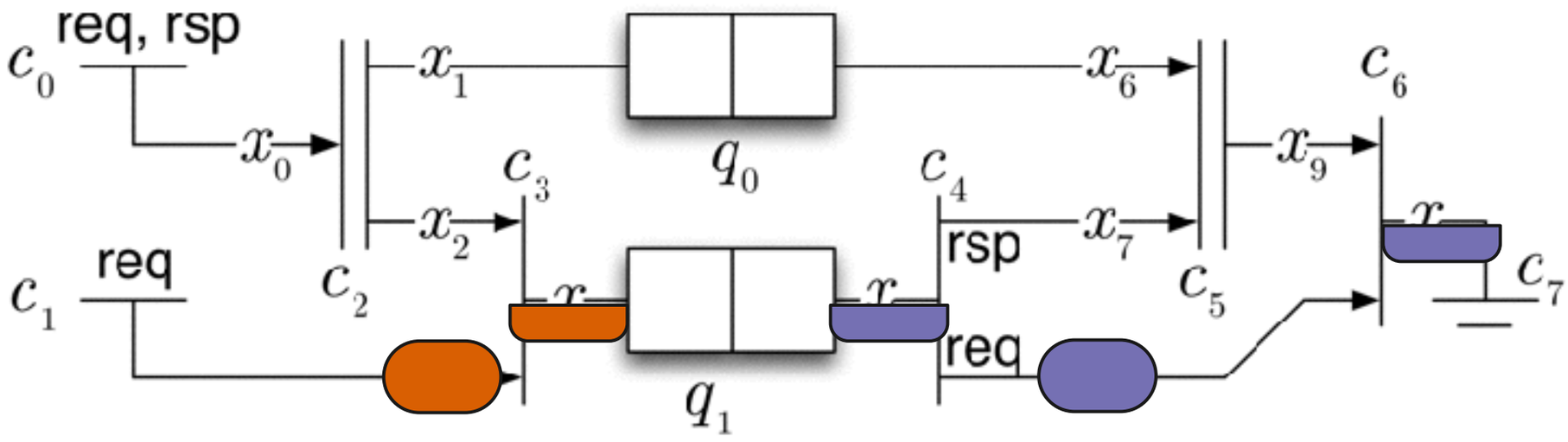
Enabled island



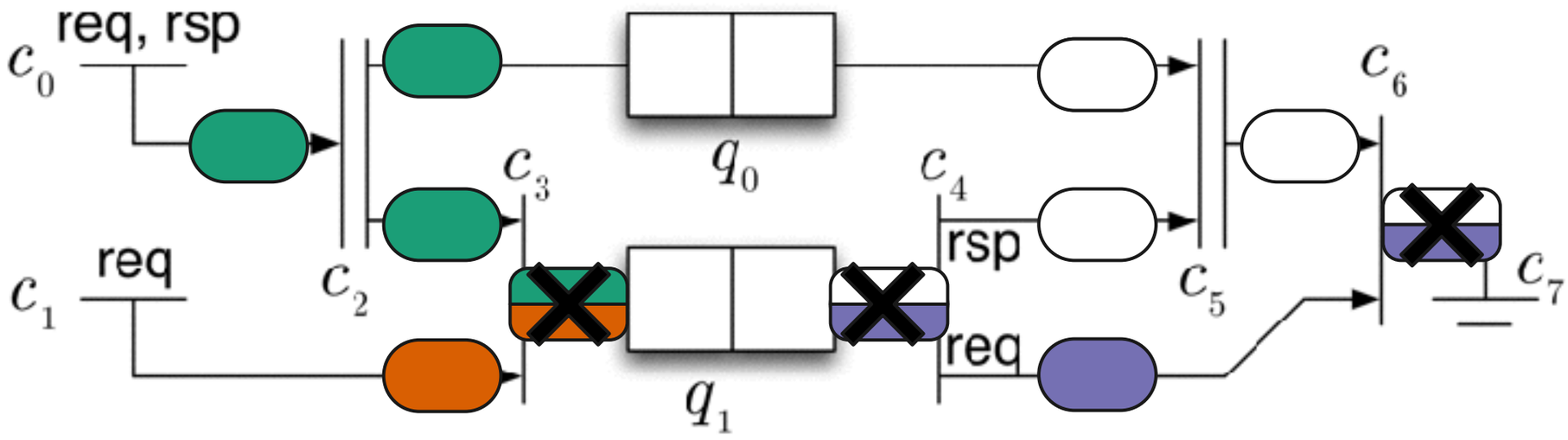
Combination of islands



Combination of islands



Conflicts



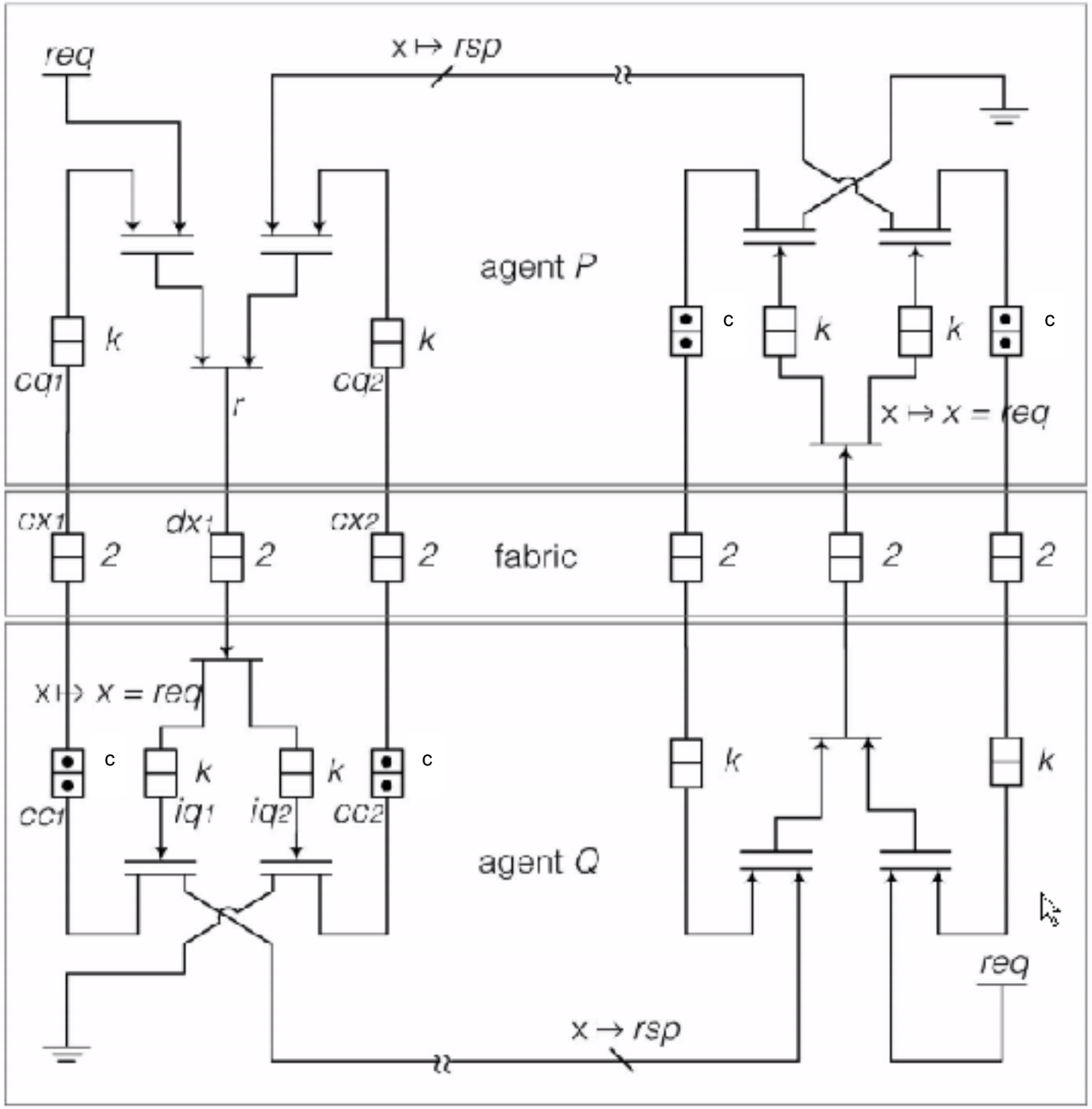
Different semantics

interleaving

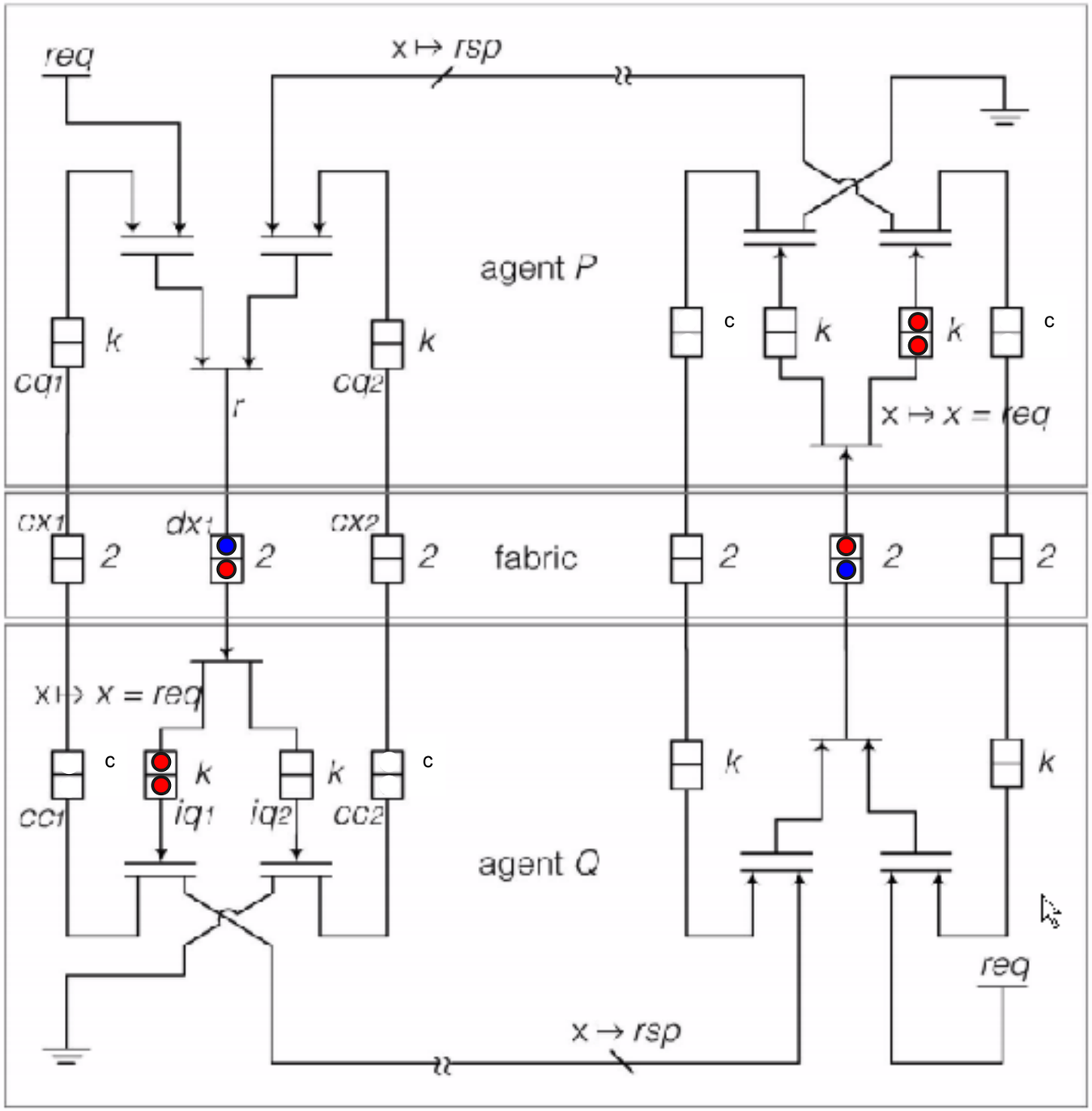
asynchronous

synchronous

Two Agents Example



Deadlock found by SMT

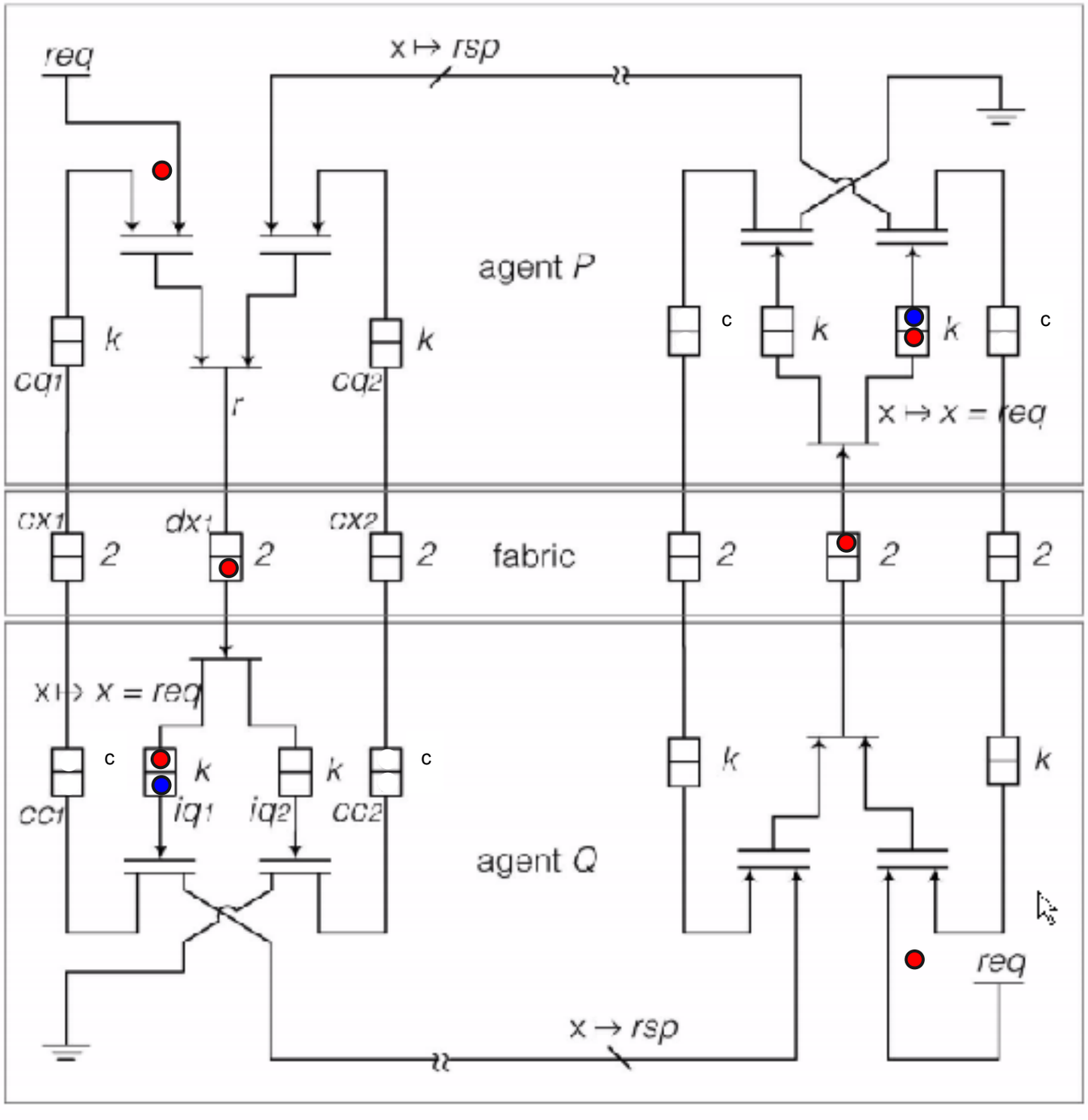


"However if credit counters are sized incorrectly to provide more credits (say k + 1) than the capacity k of the ingress queues, the system deadlocks ..."

-- Chatterjee et al. VMCAI 2011

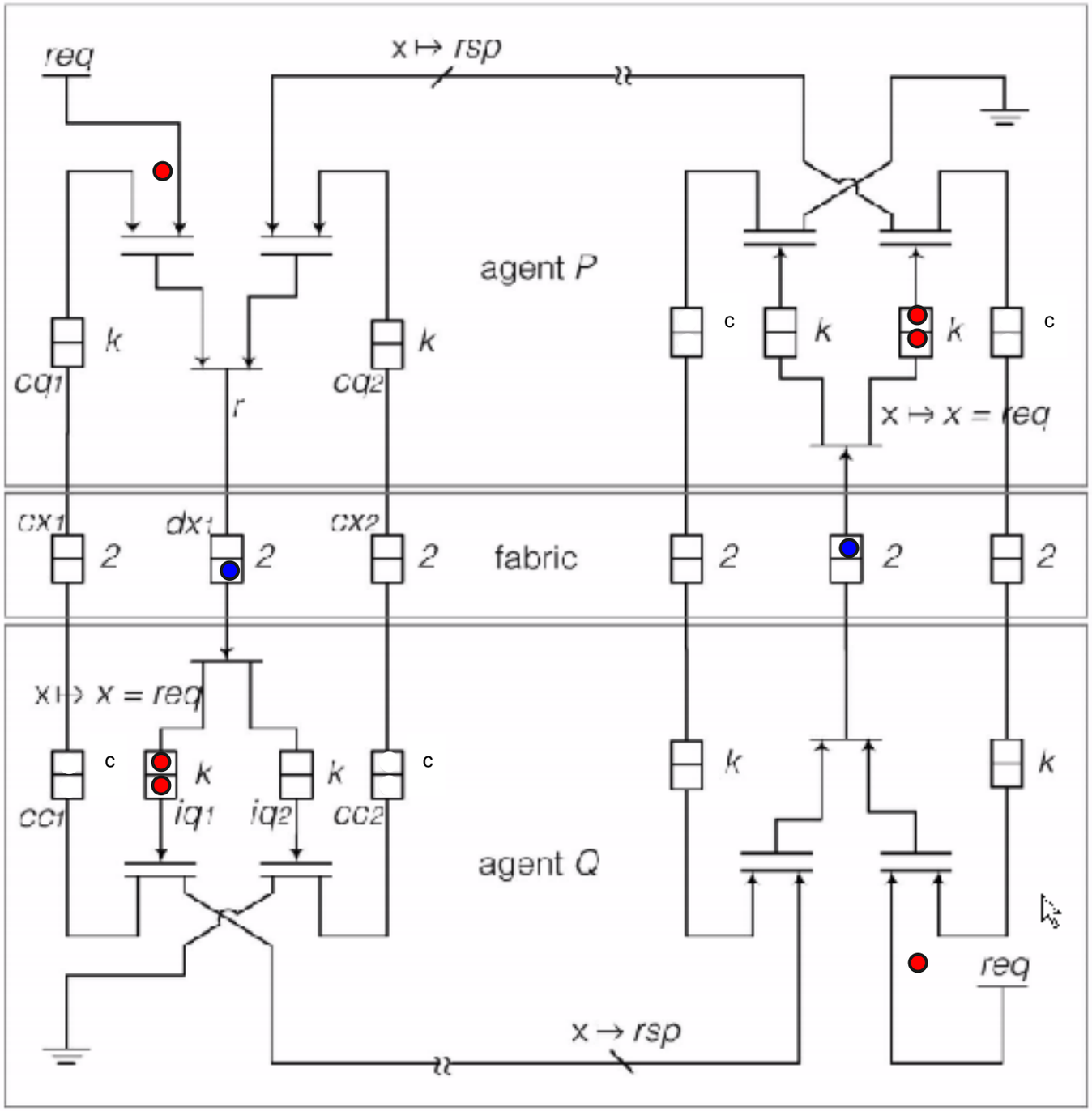
Incorrect!

Let's rewind one cycle



No credit available.
(3 requests)

Let’s move one cycle forward



1 credit back.

Some experimental results

- Two agents example
- IC3 or BDD
- optional invariants
- synchronous or interleaving

c = 2; k = 2	unreachable deadlock
2; 3	**unreachable deadlock**
4; 4	unreachable deadlock
4; 5	**unreachable deadlock**
4; 6	reachable deadlock

synchronous/IC3+INV ~3s

state space: ~2^{40} (about 10^{12})

Outline

» Intel's micro-architectural description language
  – xMAS language
  – Capturing high-level structure and message dependencies
  – Extended to “MaDL” at TU/e.
    – Micro-architectural Description Language

» Deadlock verification for MaDL
  – Definition of deadlocks
  – Labelled dependency graph
  – Feasible logically closed subgraph

» **Conclusion and future work**

Research questions and axes

Q1: How can we formalise the generic aspects of specific domains?

Q2: How can we define specification languages with built-in support for formal analyses?

Q3: How do we relate the formally proven correct specification to the actual design?

Foundations

Formalise domain-specific theories

Abstractions

Focus on aspects and properties

Language to express and analyse

Approximations

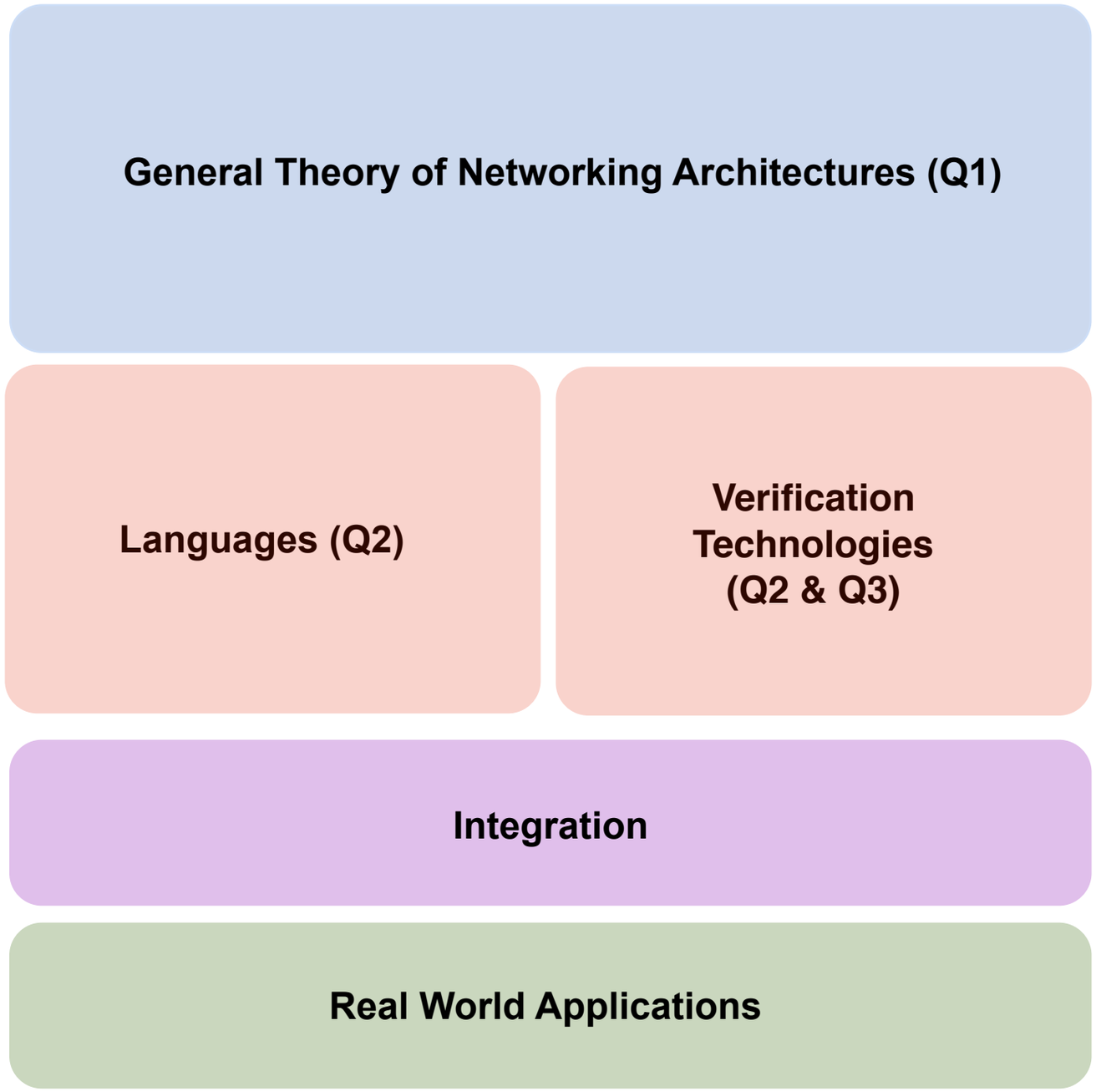
Ignore details to scale-up

Research questions

- Q1: How can we formalise the generic aspects of specific domains?

- Q2: How can we define specification languages with built-in support for formal analyses?

- Q3: How do we relate the formal proven correct specification to the actual design?



GeNoC theory with Productivity Theorem

Deadlock-free routing theory & algorithms

MaDL language & algorithms

Cooperation with industrial partners (Intel, ARM)

and academic partners (FBK, UC Irvine, Tsinghua, …)

Today's focus

General Theory of Networking Architectures (Q1)

GeNoC theory with Productivity Theorem

Deadlock-free routing theory & algorithms

Languages (Q2)

Verification Technologies (Q2 & Q3)

MaDL language & algorithms

Integration

Cooperation with industrial partners (Intel, ARM)

Real World Applications

and academic partners (FBK, UC Irvine, Tsinghua, …)

MaDL: Conclusions & Future Work

- Conclusions
  - Adequate DSL for prototyping network architectures.
  - Being applied to some industrial case-studies
- Future work
  - Equivalence relations between MaDL and RTL
  - Performance (throughput & latency) evaluation
  - Extend to entire systems

MaDL: Current activities

- Master theses
  - Perry: analysis for ring networks
  - Ruud: Quality-of-Services
- PhD theses
  - Alexander: equivalences, pre-orders, link with RTL
- Valorisation
  - Precuneus Solutions B.V.

MaDL Whiteboard DEMO

THANKS!