Formal Specification and Validation of Minimal Routing Algorithms for the 2D Mesh

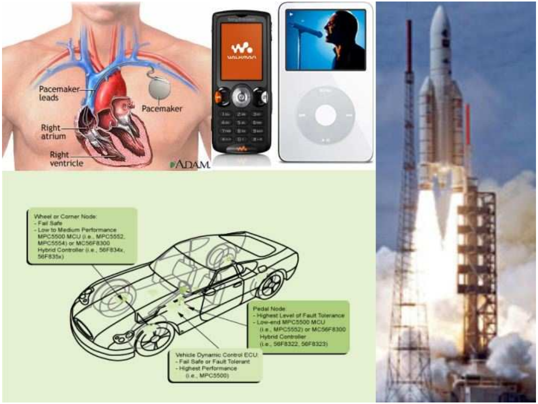
Julien Schmaltz

Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands
julien@cs.ru.nl

Part of this work funded by the EU Marie Curie project TAROT

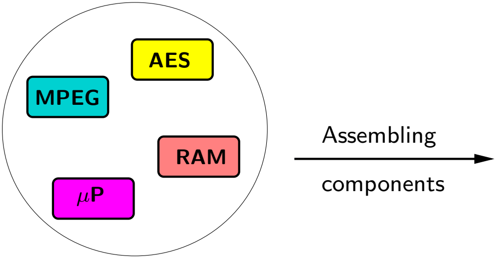
ACL2 2007, Nov. 15-16

Systems on a Chip



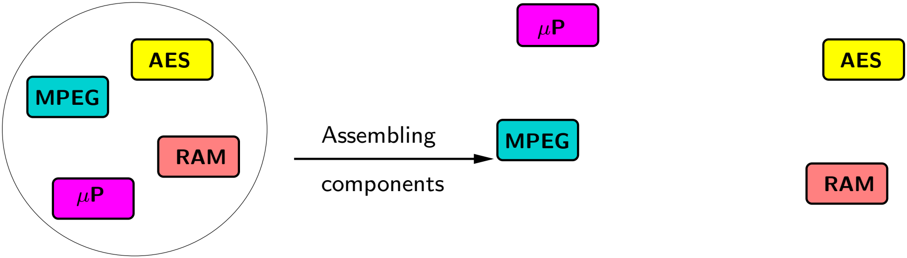
- Everywhere, critical systems
- Ever growing complexity (HW/SW)
- Safety and correctness

Platform-Based Design and Networks on a Chip



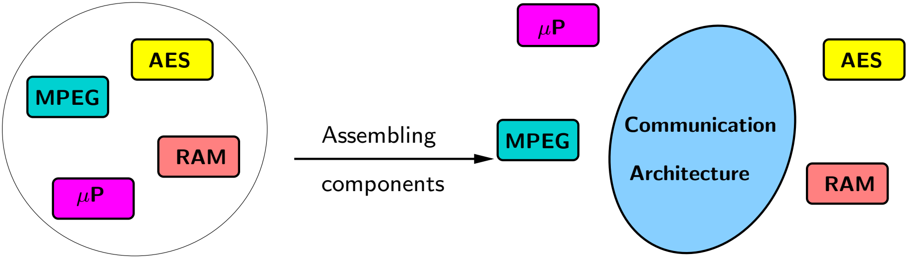
- Re-use of parameterized modules (*Intellectual Properties*)
- High-level of abstraction

Platform-Based Design and Networks on a Chip



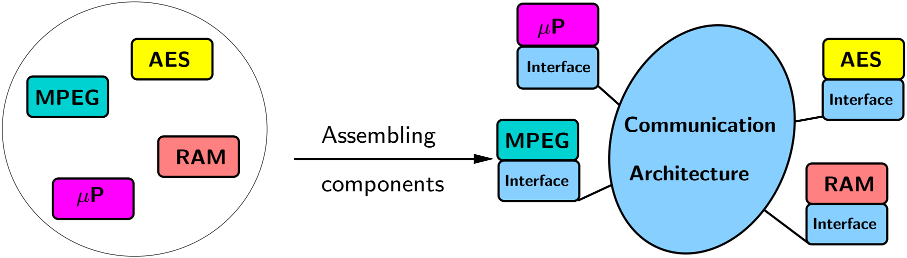
- Re-use of parameterized modules (*Intellectual Properties*)
- High-level of abstraction

Platform-Based Design and Networks on a Chip



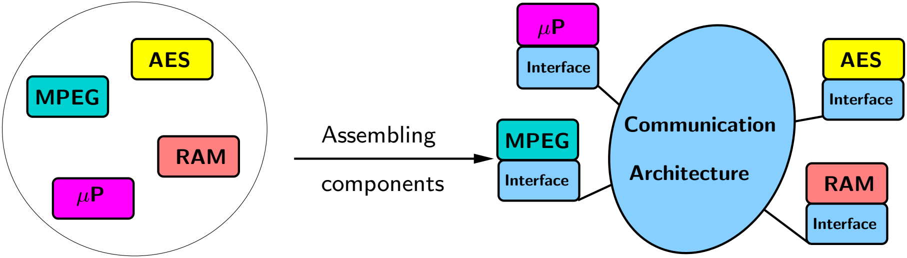
- Re-use of parameterized modules (*Intellectual Properties*)
- High-level of abstraction

Platform-Based Design and Networks on a Chip



- Re-use of parameterized modules (*Intellectual Properties*)
- High-level of abstraction
- *Communication*-centric: from buses to networks

Platform-Based Design and Networks on a Chip

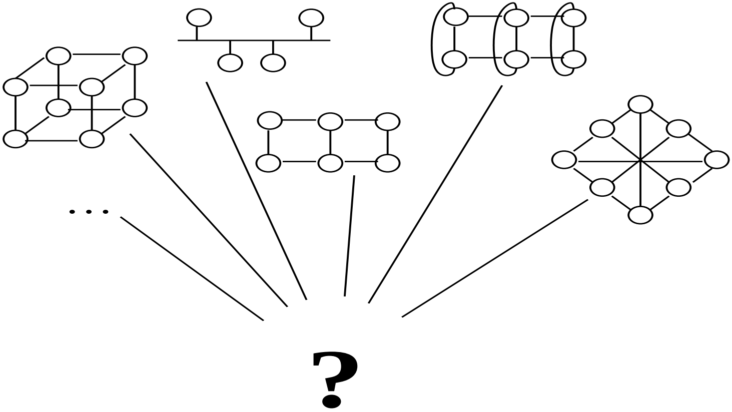


Formal Verification:

- Proof of each component
- Proof of their interconnection

Our Global Objective

Build a *meta-model* of networks: one model for all architectures



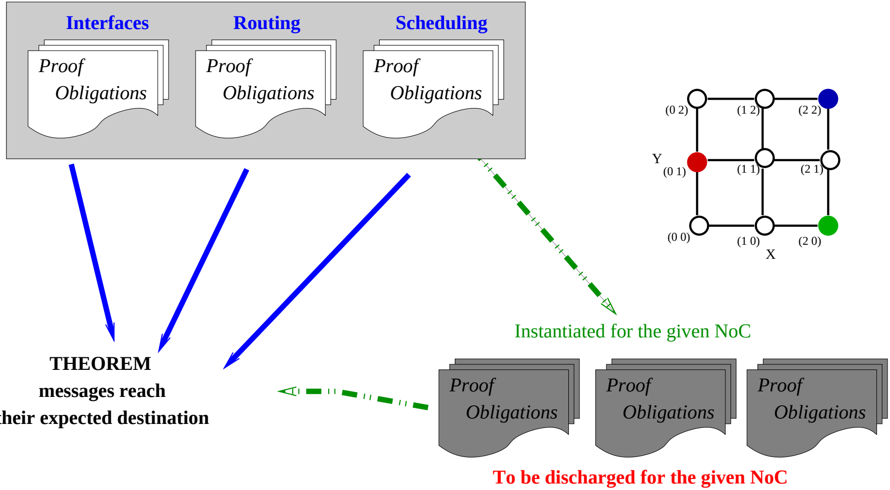
Contribution

A functional formalism for communications: *GeNoC* (Generic Network on Chip)

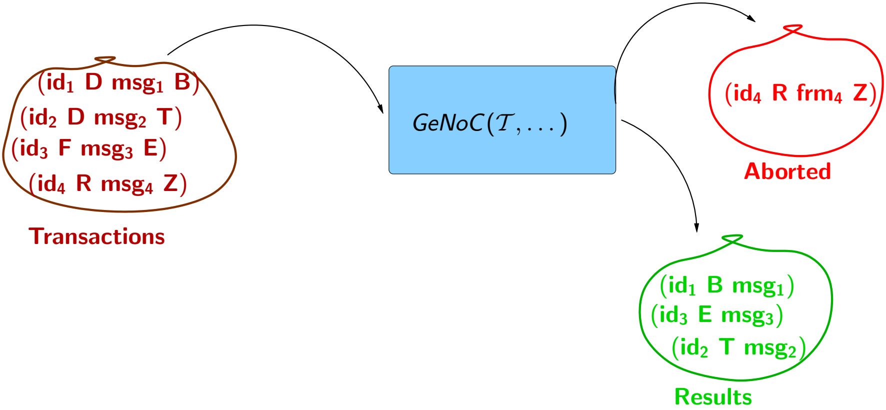
- Identifies the essential constituents and their properties
- Formalizes the interactions between them
- Correctness of the system is a consequence of the essential properties of the constituents
- Mechanized support in ACL2 (see ACL2 2006)
  - Encapsulation allows abstraction
  - Functional instantiation generates proof obligations automatically
- *Minimal routing algorithms for the 2D Mesh*
  - Deterministic case
  - Adaptive algorithm

Outline

The GeNoC Approach

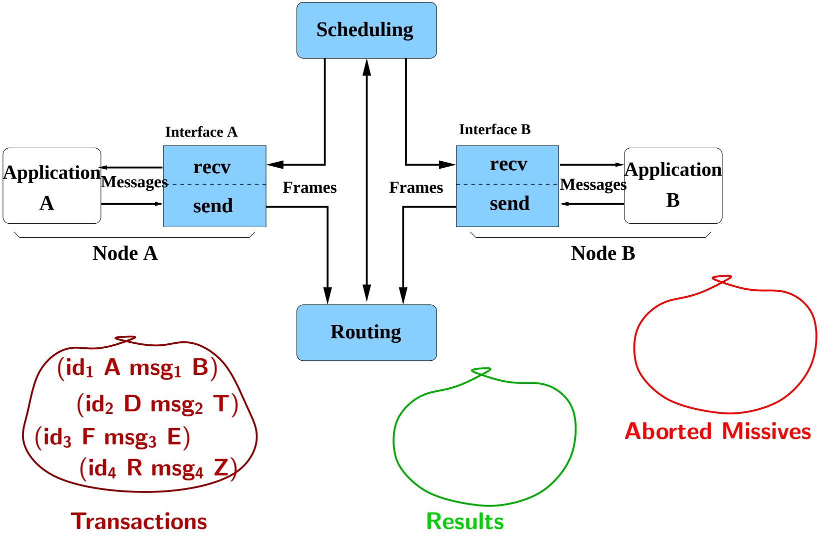


Overview of Function *GeNoC*

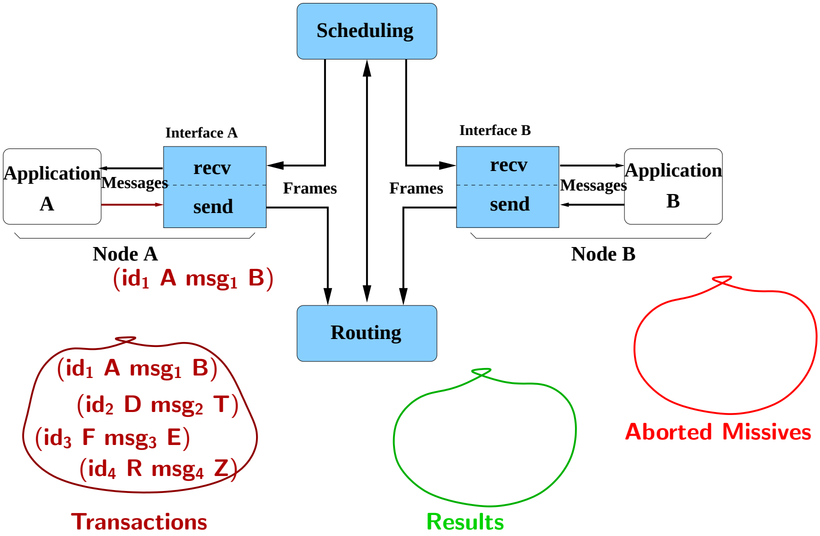


Pending communications are successful or aborted

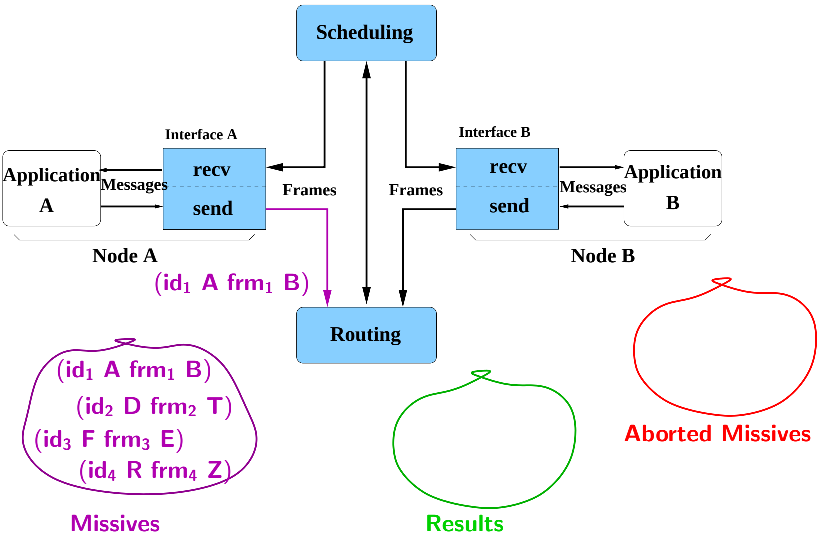
Function *GeNoC*



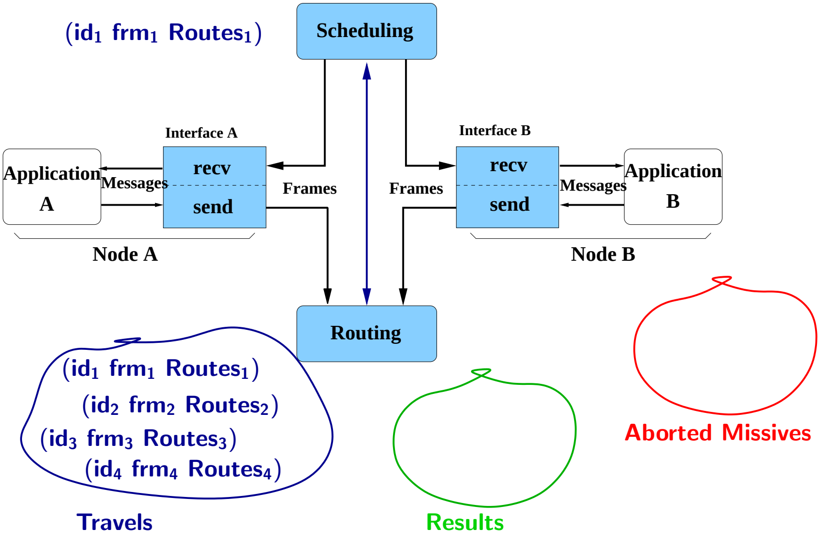
From Transactions to Missives



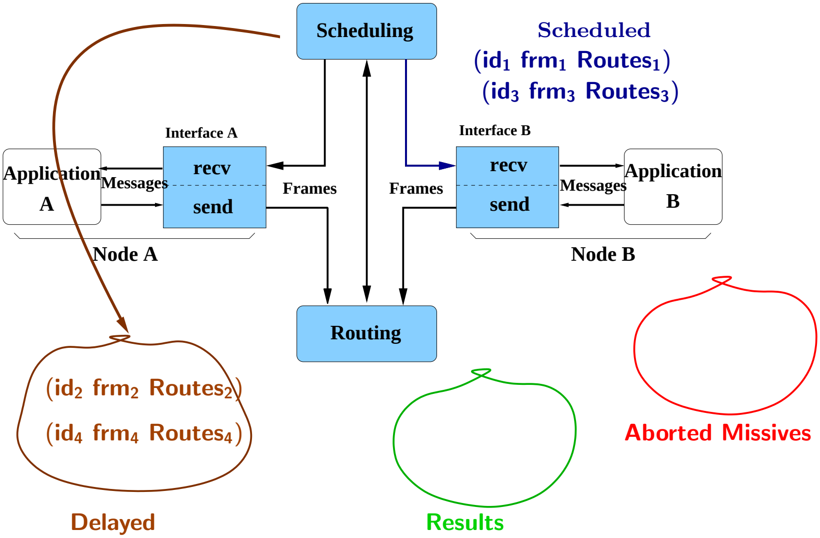
From Transactions to Missives



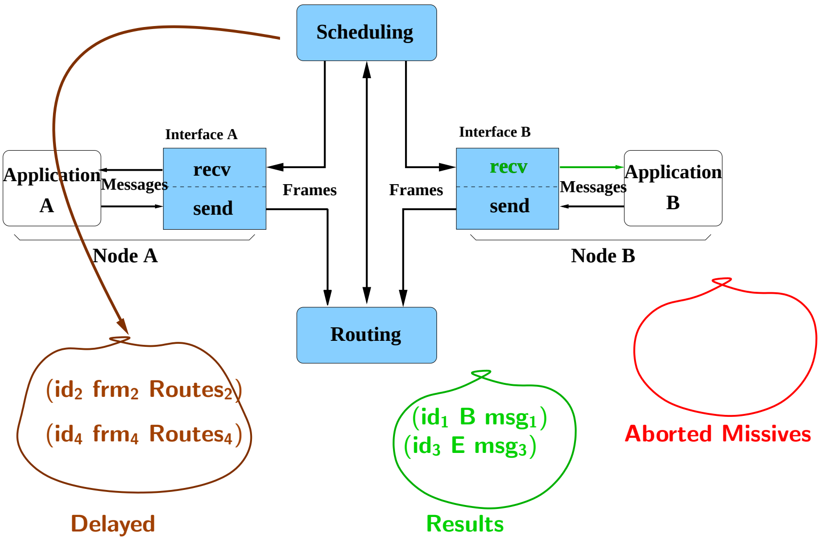
Routing Algorithm



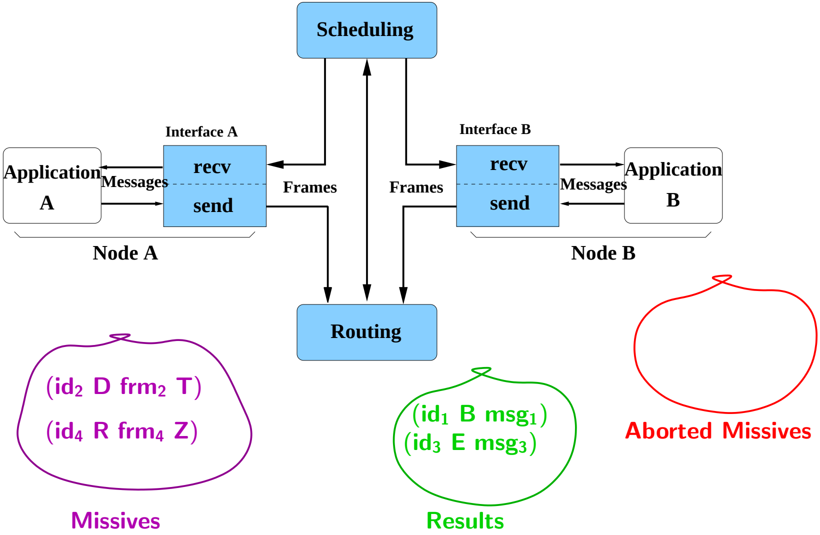
Scheduling Policy



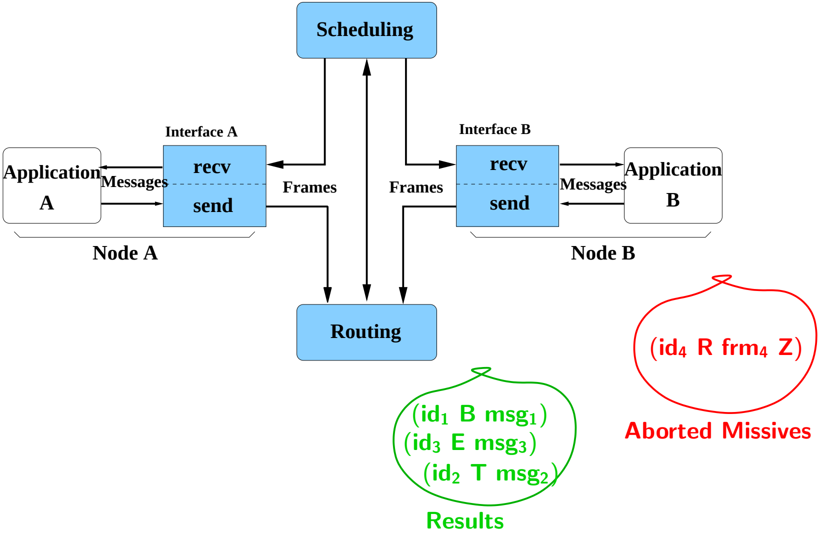
Results


Scheduling
Interface A
recv
send
Application
A
Messages
Frames
Node A
Interface B
recv
send
Application
B
Messages
Frames
Node B
Routing
(id$_2$ frm$_2$ Routes$_2$)
(id$_4$ frm$_4$ Routes$_4$)
Delayed
(id$_1$ B msg$_1$)
(id$_3$ E msg$_3$)
Results
Aborted Missives

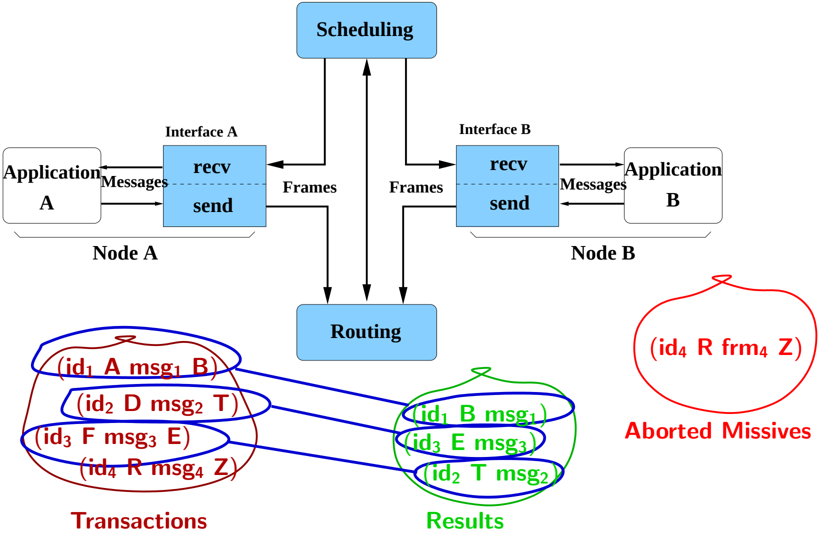
Aborted Missives



Aborted Missives



Correctness Criterion



Termination

Function *GeNoC* is recursive:

- Must be proved to terminate
- Associate to every node a *finite* number of attempts

Proof Obligations

- Interfaces
  - The composition $recv \circ send$ is an identity
- Routing $(id\ A\ frm\ B) \mapsto (id\ frm\ Routes)$
  - Missive/Travel matching
    - Same frame and identifier
    - Routes effectively go from the correct origin to the correct destination
- Scheduling
  - Mutual exclusion between *Scheduled* and *Delayed*
  - No addition of new identifiers
  - Preserve frames and route correctness

Proof of the Theorem

- Routing correctness + preserved by scheduling
  - $\rightarrow$ right destination
- No modification on frames
  - $\rightarrow$ every result is obtained by $recv \circ send$
- Interfaces correctness
  - $\rightarrow$ received message = sent message
- Mutual exclusion between *Scheduled* and *Delayed* + no new identifiers
  - $\rightarrow$ cut the proof in two parts

Outline

GeNoC Routing in ACL2

- Generic modules using encapsulation
  - Proof obligations as encapsulated constraints
  - Main constraint expressed by function CorrectRoutesp
- Compliance checked via functional-instantiation
  - Instances of proof obligations automatically generated
  - Prove 't' with functional-instantiation hint

Route Validity

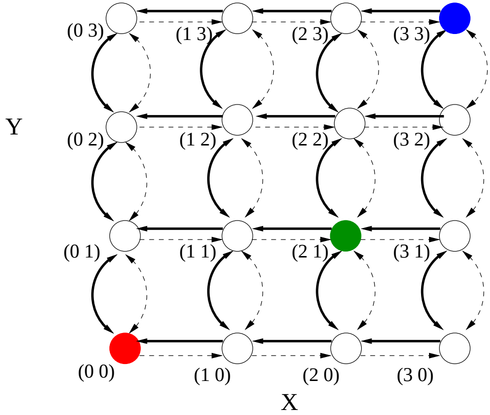
```
( defun ValidRoutep (r m)
  (and (equal (car r) (OrgM m)) ;; 1st  = origin
       (equal (car (last r)) (DestM m)) ;; last = destination
       (<= 2 (len r)))) ;; visit at least 2 nodes

( defun CheckRoutes (routes m NodeSet)
  (if (endp routes)
      t
    (let ((r (car routes)))
      (and (ValidRoutep r m)
           (subsetp r NodeSet) ;; use valid nodes only
           (CheckRoutes (cdr routes) m NodeSet)))))
```

Route Correctness

```
(defun CorrectRoutesp (V M NodeSet)
  (if (endp V)
      (if (endp M)
          t ;; len(M) = len(V)
        nil)
    (let* ((tr (car V))
           (msv (car M))
           (routes (RoutesV tr)))
      (and (CheckRoutes routes msv NodeSet)
           (equal (IdV tr) (IdM msv)) ;; same id
           (equal (FrmV tr) (FrmM msv)) ;; same frame
           (CorrectRoutesp (cdr V)
                           (cdr M) NodeSet)))))
```

Outline

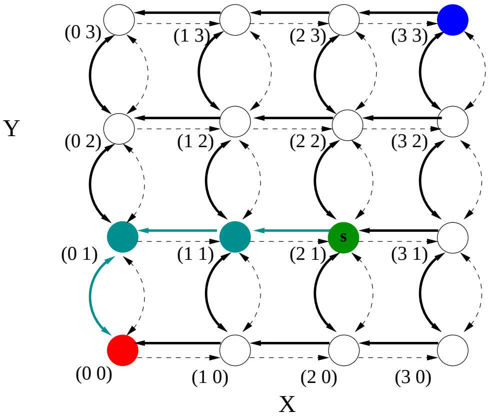
DoubleY Channel Routing: Overview



1 x-channel, 2 y-channel

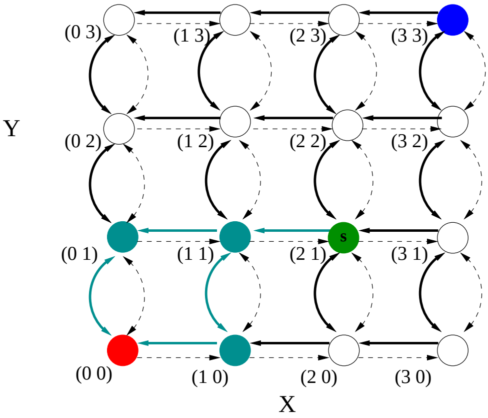
2 subnetworks

DoubleY Channel Routing: Overview



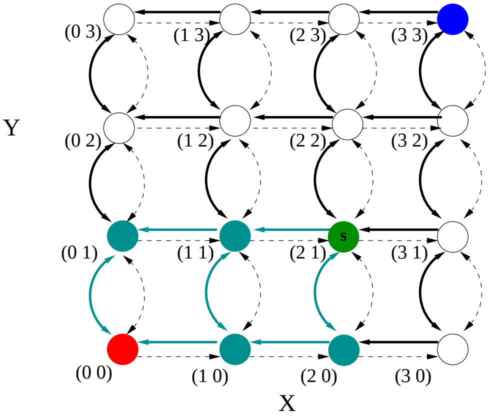
1 x-channel, 2 y-channel
2 subnetworks

DoubleY Channel Routing: Overview



1 x-channel, 2 y-channel
2 subnetworks

DoubleY Channel Routing: Overview

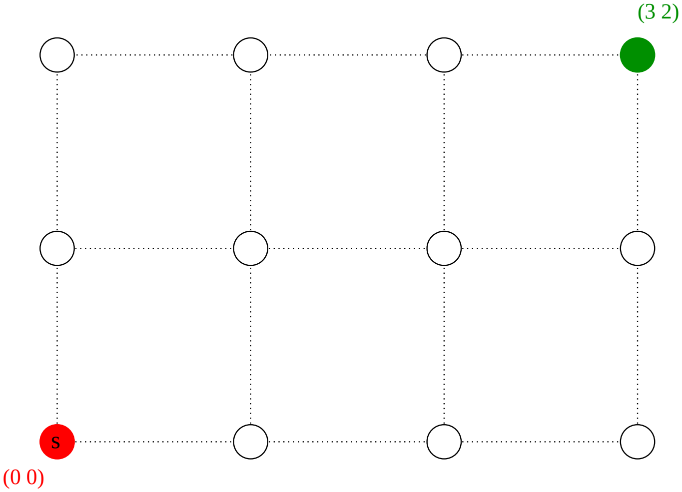


1 x-channel, 2 y-channel
2 subnetworks

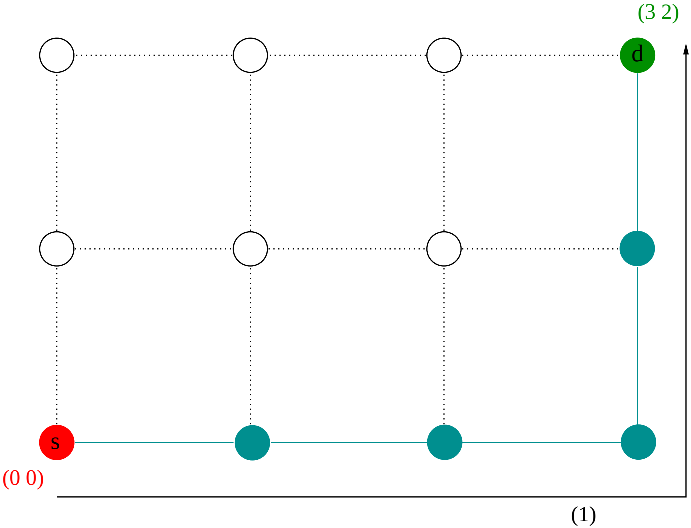
DoubleY Channel Algorithm: Principles

- Compute all possible minimal paths between a source and a destination
- Alternative application of XY and YX algorithms
  - Reuse previous proof efforts

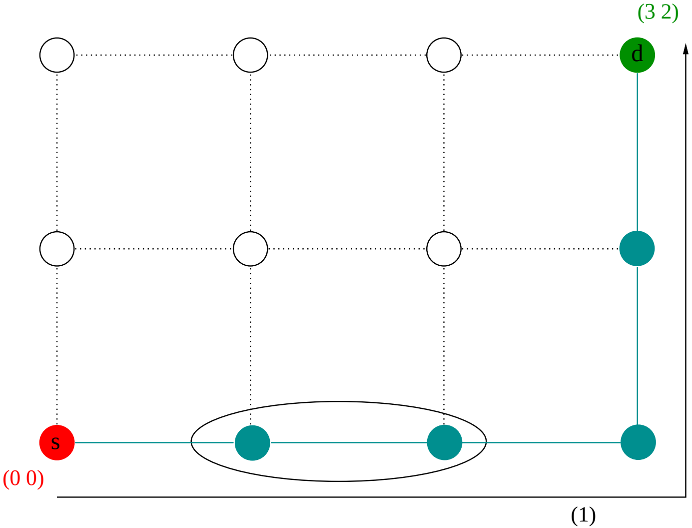
DoubleY Channel Algorithm: Example



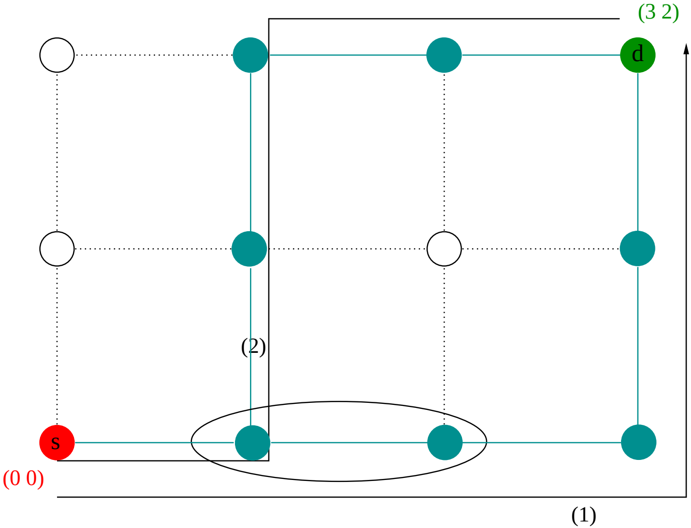
DoubleY Channel Algorithm: Example



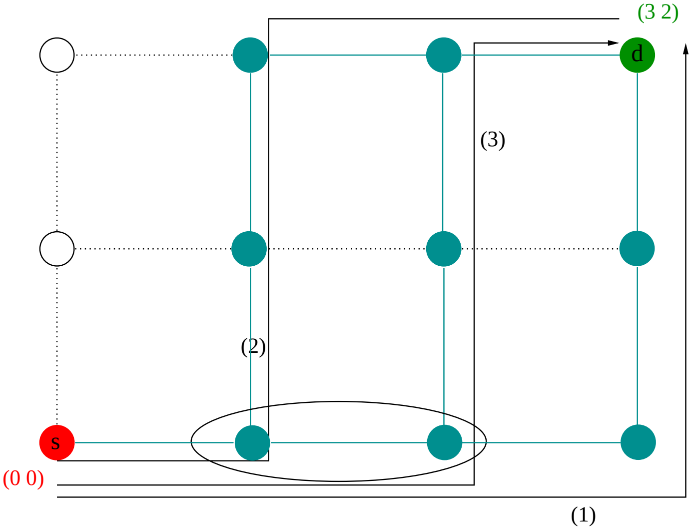
DoubleY Channel Algorithm: Example



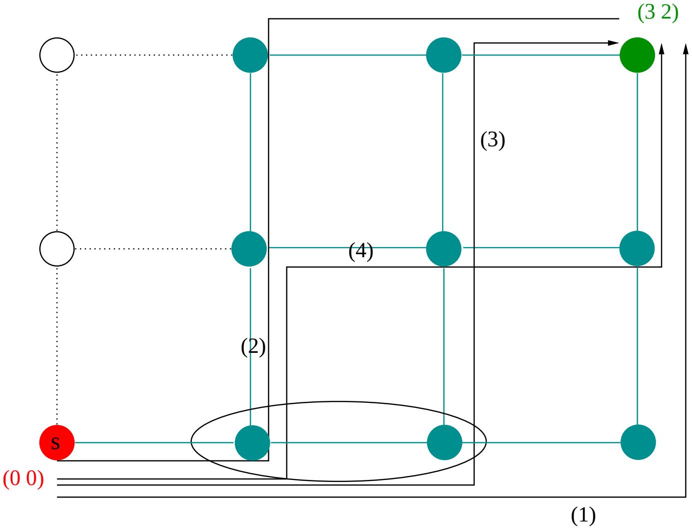
DoubleY Channel Algorithm: Example



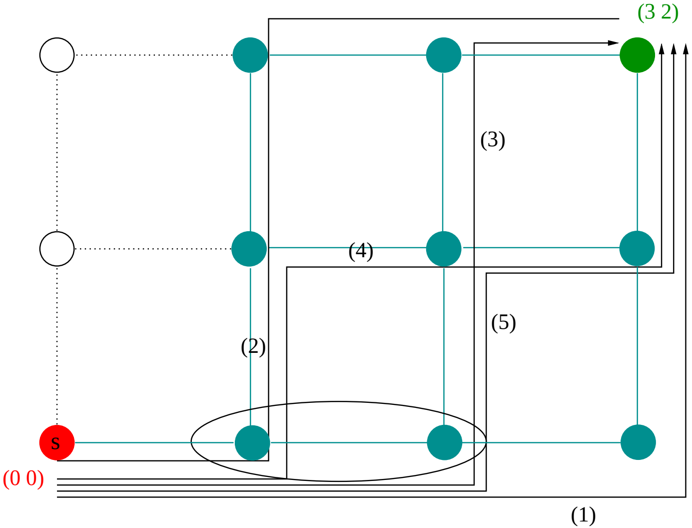
DoubleY Channel Algorithm: Example



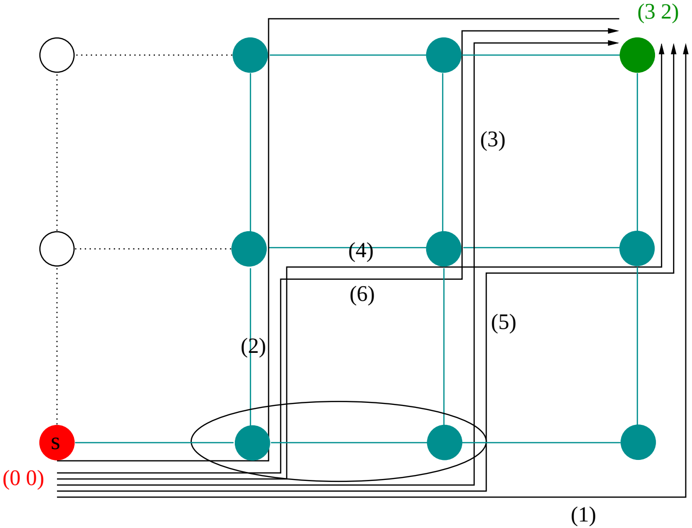
DoubleY Channel Algorithm: Example



DoubleY Channel Algorithm: Example

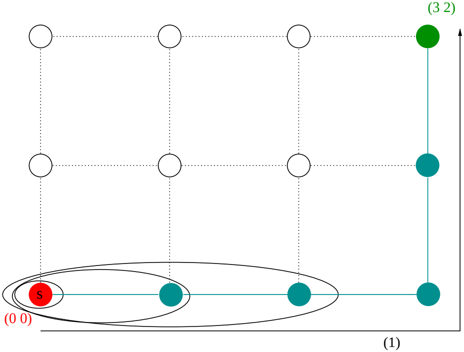


DoubleY Channel Algorithm: Example



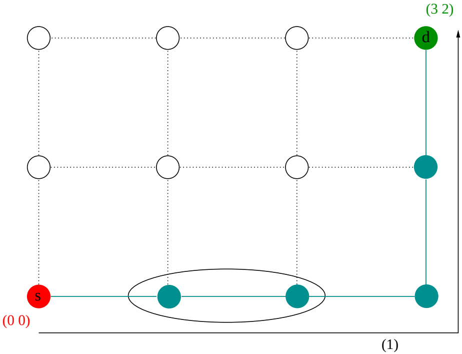
Prefixes

- Partial routes with nodes without common coordinate with destination
- For a route *r* and destination d, prefixes = (`extract-prefixes` r d)



Sources

- Choice at nodes without common coordinate with destination
- For a route *r* and destination *d*, sources = (`GetSources r d`)



ACL2 Function dy1

```
( defun dy1 (sources d flg prefixes)
  (declare (xargs :measure
                  (dist (car sources) d)))
  (if (or (endp sources)
          (not (CloserListp sources d))
          (not (2d-mesh-nodesetp sources))
          (not (coordinatep d)))
      nil
```

ACL2 Function dy1

```
( defun dy1 (sources d flg prefixes)
 ...
    (let* ((prefix (car prefixes))
           (s (car sources))
           (s_x (car s))
           (s_y (cadr s))
           (d_x (car d))
           (d_y (cadr d)))
```

ACL2 Function dy1

```
( defun dy1 (sources d flg prefixes)
 ...
       (cond
        ((or (equal s_x d_x) (equal s_y d_y))
        ;; if one coordinate has been reached,
        ;; we stop
         nil)
        (t
         (let
          ((routes
            (cond
```

ACL2 Function dy1

```
( defun dy1 (sources d flg prefixes)
 ...
              (flg
              ;; last was yx, next is xy
              (let ((suffix (xy-routing s d)))
               (cons
                (append prefix suffix)
                (dy1 (getSources suffix d)
                     d nil ;; next is YX
                     (append-l-all
                     prefix
                     (extract-prefixes suffix d))))))
              (t ;; last was xy-routing, next is yx
```

ACL2 Function dy1

```
( defun dy1 (sources d flg prefixes)
 ...
            ;; similar to xy
       )))))
       (append routes
               (dy1 (cdr sources)
                    d flg prefixes)))))))))
```

Validation of dy1

Main property: `CorrectRoutesp` for all routes between *s* and *d*

- Routes are subsets of NodeSet
- All routes start with *s* and end with *d*

  → Main invariant: prefixes and sources remember *s*

```
(defun inv (prefixes sources s)
  (if (endp prefixes)
      (inv-sources sources s)
    (inv-prefixes prefixes s)))
```

Proof Effort Overview

- 1840 lines of code
  - `dy` includes definition and validation of `yx-routing`
  - Almost copy&paste from `xy-routing`
- Around 70sec. on Intel Dual Core 2400 with 2GB of memory

	`defun`	`defthm`	size	time
Mesh NodeSet	8	6	120	0.55
`xy-routing`	7	44	520	3.8
`dy`	21	84	1200	67.6

Table: Data for the Double Y Algorithm

Conclusion

- Application of GeNoC to an adaptive routing algorithm
  - Reuse of previous work on XY routing
  - Following “The Method”, proof done in few weeks
- Limitations
  - Compute all possible paths for minimal routing algorithms only
  - Find an algorithm to compute all these paths
- Checking valid instances of encapsulate events
  - No built-in procedure in ACL2
  - Trick of proving 't' by functional-instantiation
  - Systematic and nicer method
    make-event/defspec.lisp by Ray and Kaufmann (ACL2 v3.2.1)

Future Work

- Non-minimal routing algorithms
- Static deadlocks, livelocks, starvation ...
- Dynamic deadlocks: interaction between protocols and interconnect
- Refinement method to reach RTL designs
- Explicit notion of time
- . . .

Learn more during rump session !

Invariants

First node of prefixes must be the original source:

```
(defun inv-prefixes (prefixes s)
  (if (endp prefixes)
      t
    (and (equal (caar prefixes) s)
         (inv-prefixes (cdr prefixes) s))))
```

... first node of sources as well !

```
(defun inv-sources (sources s)
  (if (endp sources)
      t
    (and (equal (car sources) s)
         (inv-sources (cdr sources) s))))
```

Formal Definition

From a list of **transactions,** $\mathcal{T}$, the set of nodes *NodeSet* and a list of attempt numbers *att*, function *GeNoC* produces:

- The list $\mathcal{R}$ of **results**
- The list $\mathcal{A}$ for **aborted missives**

$$
\begin{aligned}
GeNoC: \quad & \mathcal{D}_{\mathcal{T}} \times GenNodeSet \times AttLst \rightarrow \mathcal{D}_{\mathcal{R}} \times \mathcal{D}_{\mathcal{M}} \\
& (\mathcal{T}, NodeSet, att) \mapsto (\mathcal{R}, \mathcal{A})
\end{aligned}
$$

Correctness Theorem

$\forall res \in \mathcal{R},$

$$\exists! trans \in \mathcal{T}, \begin{cases} & Id_{\mathcal{R}}(res) = Id_{\mathcal{T}}(trans) \\ \wedge & Msg_{\mathcal{R}}(res) = Msg_{\mathcal{T}}(trans) \\ \wedge & Dest_{\mathcal{R}}(res) = Dest_{\mathcal{T}}(trans) \end{cases}$$

For any result *res*, there exists a unique transaction *trans* such that *trans* and *res* have the same identifier, message, and destination.