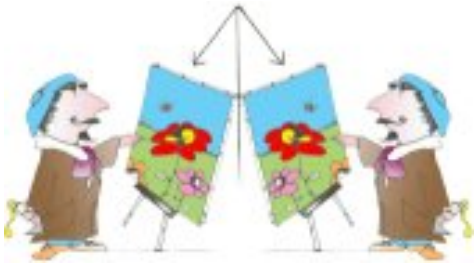


ULB

Model Checking Communicating Processes: Run Graphs, Graph Grammars & MSO

Alexander Heußner

Formal Methods & Verification group Brussels

GT-VMT 2012

Agenda

- formal model of distributed recursive communicating processes
- recall un-/decidability of basic verification questions
- show line of attack via under-approximative approaches

- from semantic "boundedness" to graph grammars
- and back to decidability of MSO model checking

article takes different way:

- try to extend known results beyond reachability
- directly arrive at undecidability
- extend proof-idea of reachability result for subclass of RCPS
- arrive at graph grammars for run graphs
- apply known HRG-treewidth-MSO connection

Agenda

- formal model of distributed recursive communicating processes
- recall un-/decidability of basic verification questions
- show line of attack via under-approximative approaches

- from semantic "boundedness" to graph grammars
- and back to decidability of MSO model checking

article takes different way:

- try to extend known results beyond reachability
- directly arrive at undecidability
- extend proof-idea of reachability result for subclass of RCPS
- arrive at graph grammars for run graphs
- apply known HRG-treewidth-MSO connection

Agenda

- ➪ formal model of distributed recursive communicating processes
- ➪ recall un-/decidability of basic verification questions
- ➪ show line of attack via under-approximative approaches

- ➪ from semantic "boundedness" to graph grammars
- ➪ and back to decidability of MSO model checking

article takes different way:

- ➪ try to extend known results beyond reachability
- ➪ directly arrive at undecidability
- ➪ extend proof-idea of reachability result for subclass of RCPS
- ➪ arrive at graph grammars for run graphs
- ➪ apply known HRG-treewidth-MSO connection

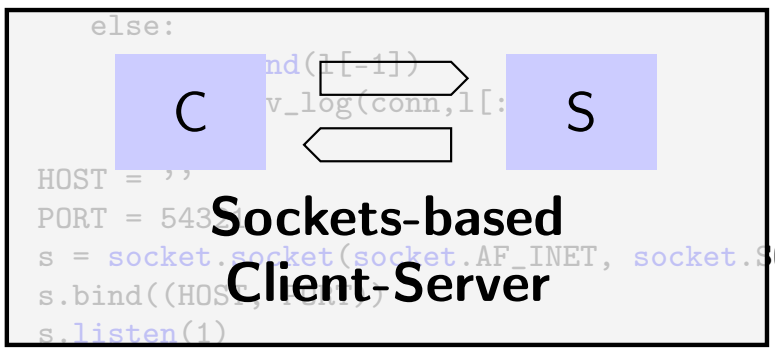
Verification of Distributed Systems

```
# Simple Reverse Echo Server
import socket

def send_rev_log(conn,l):
   if not l:
      return []
   else:
      conn.send(l[-1])
      send_rev_log(conn,l[:-1])

HOST = ''
PORT = 54321
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)

conn, addr = s.accept()

log=[] # local pushdown store

while 1:
   msg = conn.recv(1024)
   if not msg: break

   if msg.startswith("show"):
      log=send_rev_log(conn,log)
   else:
      log.append(msg)
conn.close()
```

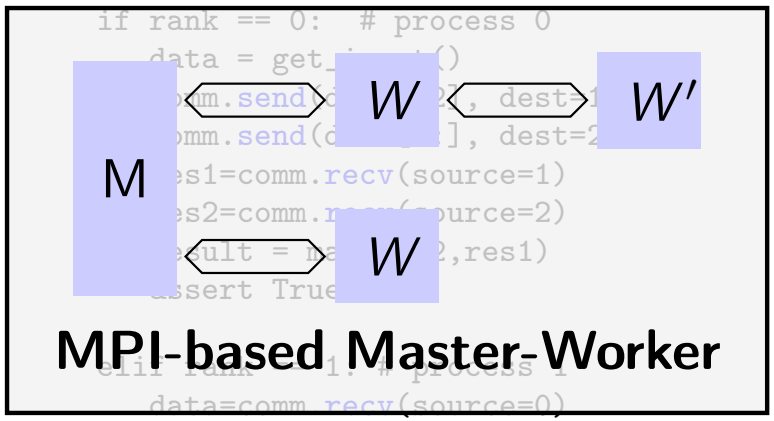
```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
assert comm.Get_size() == 7

if rank == 0:  # process 0
   data = get_input()
   comm.send(data[:2], dest=1)
   comm.send(data[2:], dest=2)
   res1=comm.recv(source=1)
   res2=comm.recv(source=2)
   result = max(res2,res1)
   assert True

elif rank == 1: # process 1
   data=comm.recv(source=0)
   comm.send(data[:1], dest=4)
   comm.send(data[1:], dest=6)
   res1=comm.recv(source=4)
   res2=comm.recv(source=6)

   def gcd(x,y) :
      if y==0: return x
      else: return gcd(y,x%y)

   result = gcd(res1,res2)
   comm.send(result,dest=0)
   assert True
# ...
```

Verification of Distributed Systems

```
# Simple Reverse Echo Server
import socket

def send_rev_log(conn,l):
   if not l:
      return []
   else:
      conn.send(l[-1])
      send_rev_log(conn,l[:-1])

HOST = ''
PORT = 5432
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)

conn, addr = s.accept()

log=[] # local pushdown store

while 1:
   msg = conn.recv(1024)
   if not msg: break

   if msg.startswith("show"):
      log=send_rev_log(conn,log)
   else:
      log.append(msg)
conn.close()
```




```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
assert comm.Get_size() == 7

if rank == 0:  # process 0
   data = get_input()
   comm.send(data[:1], dest=1)
   comm.send(data[1:], dest=2)
   res1=comm.recv(source=1)
   res2=comm.recv(source=2)
   result = gcd(res1,res2)
   assert True
elif rank == 1: # process 1
   data=comm.recv(source=0)
   comm.send(data[:1], dest=4)
   comm.send(data[1:], dest=6)
   res1=comm.recv(source=4)
   res2=comm.recv(source=6)

   def gcd(x,y) :
      if y==0: return x
      else: return gcd(y,x%y)

   result = gcd(res1,res2)
   comm.send(result,dest=0)
   assert True
# ...
```



Verification of Distributed Systems

```
# Simple Reverse Echo Server
import socket

def send_rev_log(conn,l):
   if not l:
      return []
   else:
      conn.send(l[-1])
      send_rev_log(conn,l[:-1])

HOST = ''
PORT = 54321
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)

conn, addr = s.accept()

log=[] # local pushdown store

while 1:
   msg = conn.recv(1024)
   if not msg: break

   if msg.startswith("show"):
      log=send_rev_log(conn,log)
   else:
      log.append(msg)
conn.close()
```

concurrency

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
assert comm.Get_size() == 7

if rank == 0:  # process 0
   data = get_input()
   comm.send(data[:2], dest=1)
   comm.send(data[2:], dest=2)
   res1=comm.recv(source=1)
   res2=comm.recv(source=2)
   result = max(res2,res1)
   assert True

elif rank == 1: # process 1
   data=comm.recv(source=0)
   comm.send(data[:1], dest=4)
   comm.send(data[1:], dest=6)
   res1=comm.recv(source=4)
   res2=comm.recv(source=6)

   def gcd(x,y) :
      if y==0: return x
      else: return gcd(y,x%y)

   result = gcd(res1,res2)
   comm.send(result,dest=0)
   assert True
# ...
```

Verification of Distributed Systems

```
# Simple Reverse Echo Server
import socket

def send_rev_log(conn,l):
   if not l:
      return []
   else:
      conn.send(l[-1])
      send_rev_log(conn,l[:-1])

HOST = ''
PORT = 54321
s = socket.socket(socket.AF_INE   socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)

conn, addr = s.accept()

log=[] # local pushdown store

while 1:
   msg = conn.recv(1024)
   if not msg: break

   if msg.startswith("show"):
      log=send_rev_log(conn,log)
   else:
      log.append(msg)
conn.close()
```

concurrency

asynchronous communication

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
assert comm.Get_size() == 7

if rank == 0:  # process 0
   data = get_input()
   comm.send(data[:2], dest=1)
   comm.send(data[2:], dest=2)
   res1=comm.recv(source=1)
   res2=comm.recv(source=2)
   resul  = max(res2,res1)
   assert True

elif rank == 1: # process 1
   data=comm.recv(source=0)
   comm.send(data[:1], dest=4)
   comm.send(data[1:], dest=6)
   res1=comm.recv(source=4)
   res2=comm.recv(source=6)

   def gcd(x,y) :
      if y==0: return x
      else: return gcd(y,x%y)

   result = gcd(res1,res2)
   comm.send(result,dest=0)
   assert True
# ...
```

Verification of Distributed Systems

```
# Simple Reverse Echo Server
import socket

def send_rev_log(conn,l):
   if not l:
      return []
   else:
      conn.send(l[-1])
      send_rev_log(conn,l[:-1])

HOST = ''
PORT = 54321
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)

conn, addr = s.accept()

log=[] # local pushdown store

while 1:
   msg = conn.recv(1024)
   if not msg: break

   if msg.startswith("show"):
      log=send_rev_log(conn,log)
   else:
      log.append(msg)
conn.close()
```

concurrency

asynchronous communication

recursion

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
assert comm.Get_size() == 7

if rank == 0:  # process 0
   data = get_input()
   comm.send(data[:2], dest=1)
   comm.send(data[2:], dest=2)
   res1=comm.recv(source=1)
   res2=comm.recv(source=2)
   result = max(res2,res1)
   assert True

elif rank == 1: # process 1
   data=comm.recv(source=0)
   comm.send(data[:1], dest=4)
   comm.send(data[1:], dest=6)
   res1=comm.recv(source=4)
   res2=comm.recv(source=6)

   def gcd(x,y) :
      if y==0: return x
      else: return gcd(y,x%y)

   result = gcd(res1,res2)
   comm.send(result,dest=0)
   assert True
# ...
```

Verification of Distributed Systems

```
# Simple Reverse Echo Server
import socket

def send_rev_log(conn,l):
   if not l:
      return []
   else:
      conn.send(l[-1])
      send_rev_log(conn,l[:-1])

HOST = ''
PORT = 54321
s = socket.socket(socket.AF_INE   socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)

conn, addr = s.accept()

log=[] # local pushdown store

while 1:
   msg = conn.recv(1024)
   if not msg: break

   if msg.startswith("show"):
      log=send_rev_log(conn,log)
   else:
      log.append(msg)
conn.close()
```

concurrency

asynchronous communication

recursion

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
assert comm.Get_size() == 7

if rank == 0:  # process 0
   data = get_input()
   comm.send(data[:2], dest=1)
   comm.send(data[2:], dest=2)
   res1=comm.recv(source=1)
   res2=comm.recv(source=2)
   result = max(res2,res1)
   assert True

elif rank == 1: # p
   data=comm.recv(s
   comm.send(data[:1], dest=4)
   comm.send(data[1:], dest=6)
   res1=comm.recv(source=4)
   res2=comm.recv(source=6)

   def gcd(x,y) :
      if y==0: return x
      else: return gcd(y,x%y)

   result = gcd(res1,res2)
   comm.send(result,dest=0)
   assert True
# ...
```

check safety

Verification of Distributed Systems

check specification in formal logic

e.g., the server returns all received messages in reverse order

concurrency

asynchronous communication

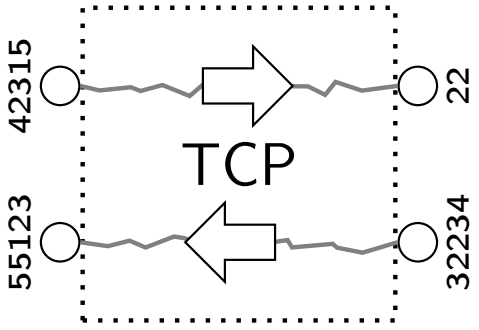
recursion

check safety

```
    else:
        conn.send(l[-1])
        send_rev_log(conn,l[:-1])

HOST = ''
PORT = 54321
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)

conn, addr = s.accept()

log=[] # local pushdown store

while 1:
    msg = conn.recv(1024)
    if not msg: break

    if msg.startswith("show"):
        log=send_rev_log(conn,log)
    else:
        log.append(msg)
conn.close()
```

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
assert comm.Get_size() == 7

if rank == 0:  # process 0
    data = get_input()
    comm.send(data[:2], dest=1)
    comm.send(data[2:], dest=2)
    res1=comm.recv(source=1)
    res2=comm.recv(source=2)
    result = max(res2,res1)
    assert True

elif rank == 1: # p
    data=comm.recv(s
    comm.send(data[:1], dest=4)
    comm.send(data[1:], dest=6)
    res1=comm.recv(source=4)
    res2=comm.recv(source=6)

    def gcd(x,y) :
        if y==0: return x
        else: return gcd(y,x%y)

    result = gcd(res1,res2)
    comm.send(result,dest=0)
    assert True
# ...
```

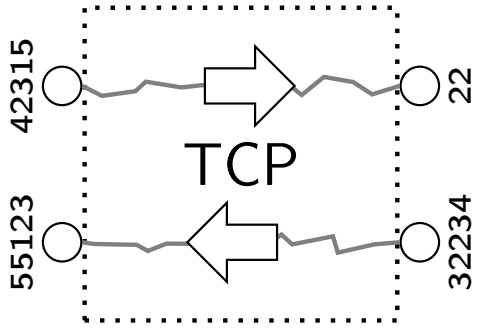
Formal Model : RCPS

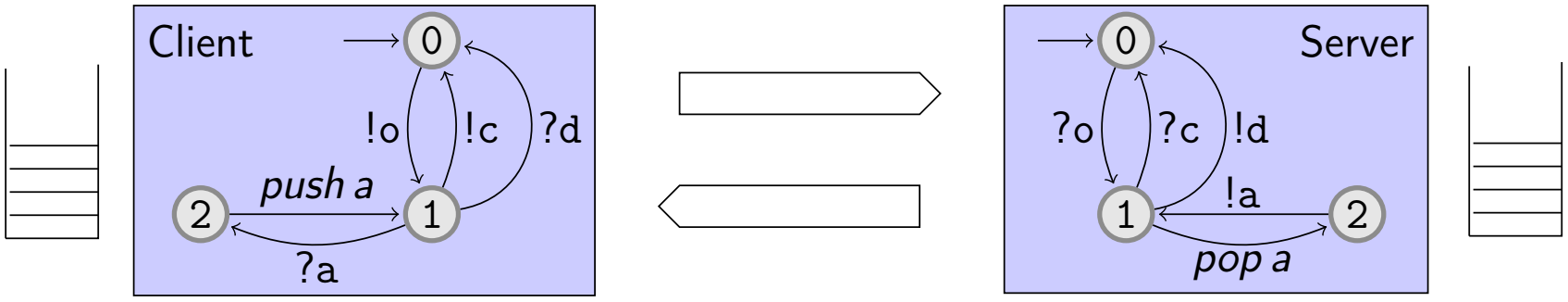


Formalize by Recursive Communicating ProcesseS

- reliable, asynchronous, unbounded channels $\longmapsto$ fifo queues
- network $\longmapsto$ (communication) graph / architecture
- recursion $\longmapsto$ pushdown stack

Formal Model : RCPS

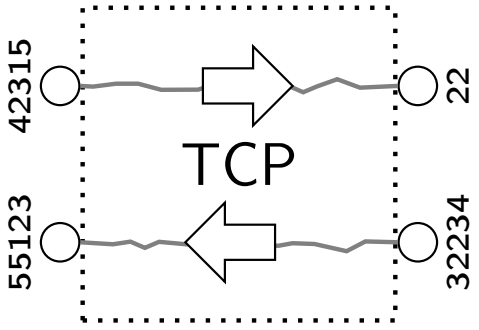


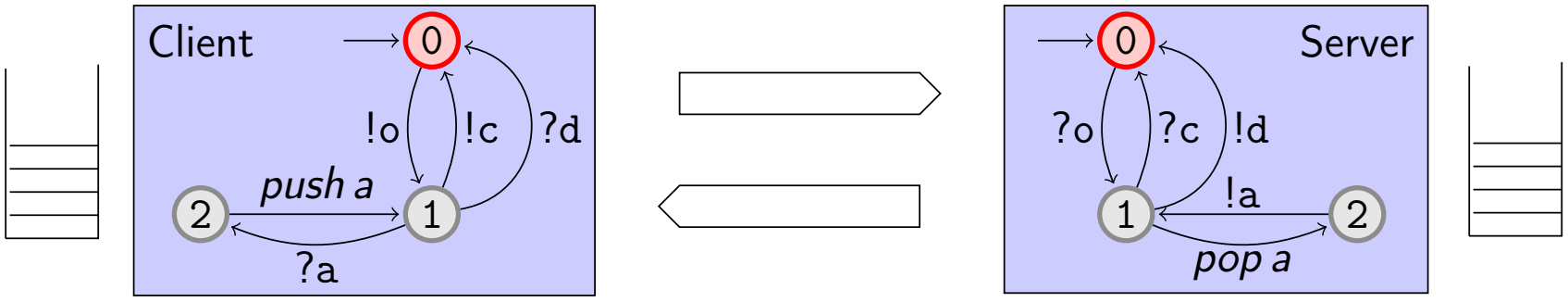


Formalize by Recursive Communicating ProcesseS

- reliable, asynchronous, unbounded channels $\longmapsto$ fifo queues
- network $\longmapsto$ (communication) graph / architecture
- recursion $\longmapsto$ pushdown stack

Formal Model : RCPS

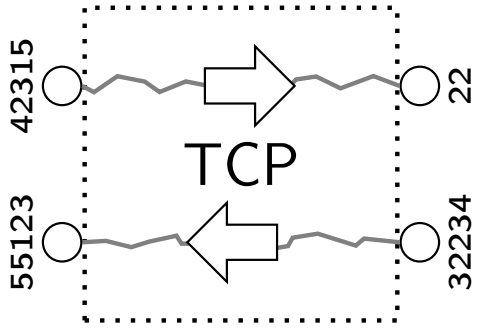


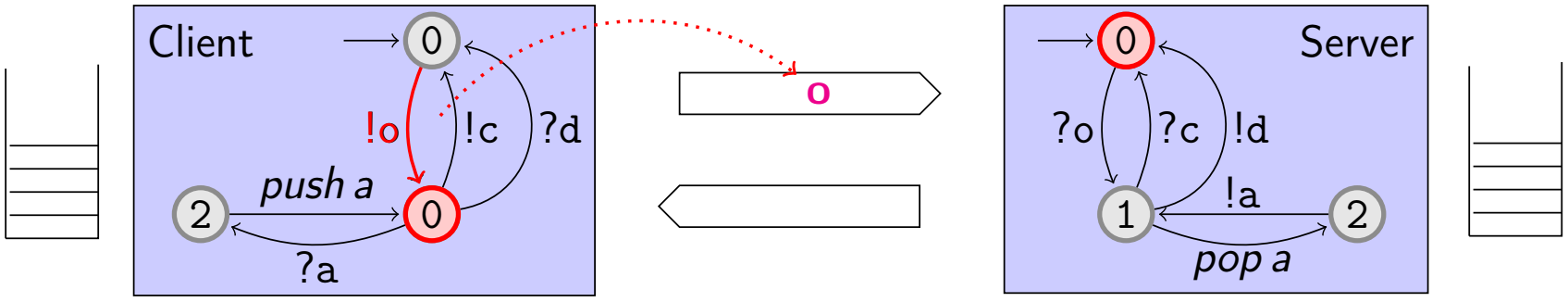


Formalize by Recursive Communicating ProcesseS

- reliable, asynchronous, unbounded channels $\longmapsto$ fifo queues
- network $\longmapsto$ (communication) graph / architecture
- recursion $\longmapsto$ pushdown stack

Formal Model : RCPS

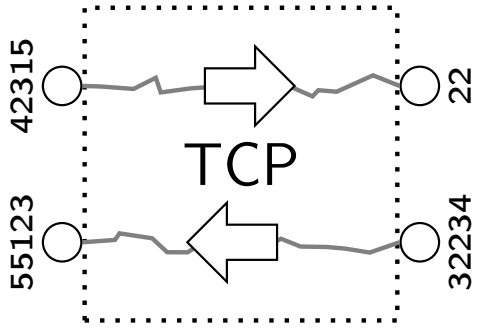


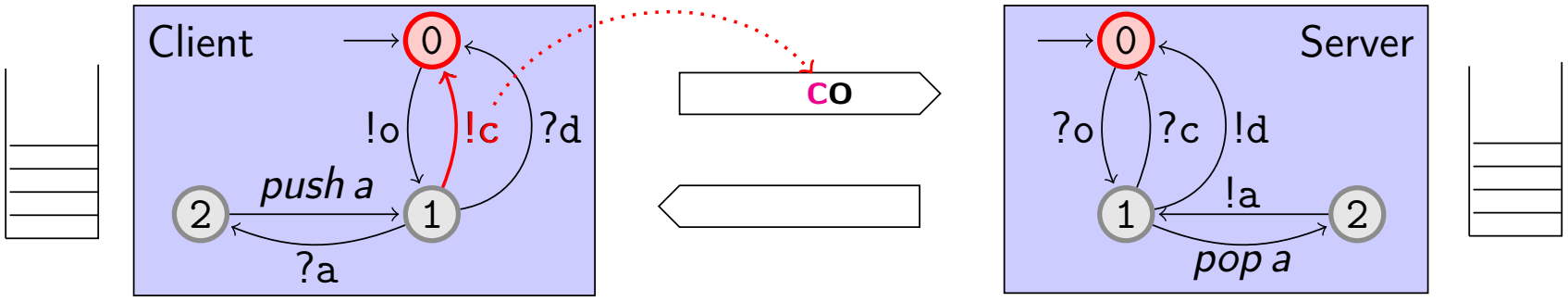


Formalize by Recursive Communicating ProcesseS

- reliable, asynchronous, unbounded channels $\longmapsto$ fifo queues
- network $\longmapsto$ (communication) graph / architecture
- recursion $\longmapsto$ pushdown stack

Formal Model : RCPS

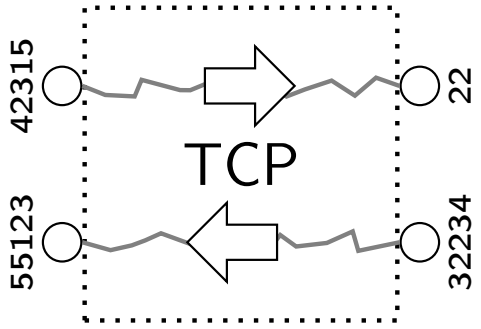


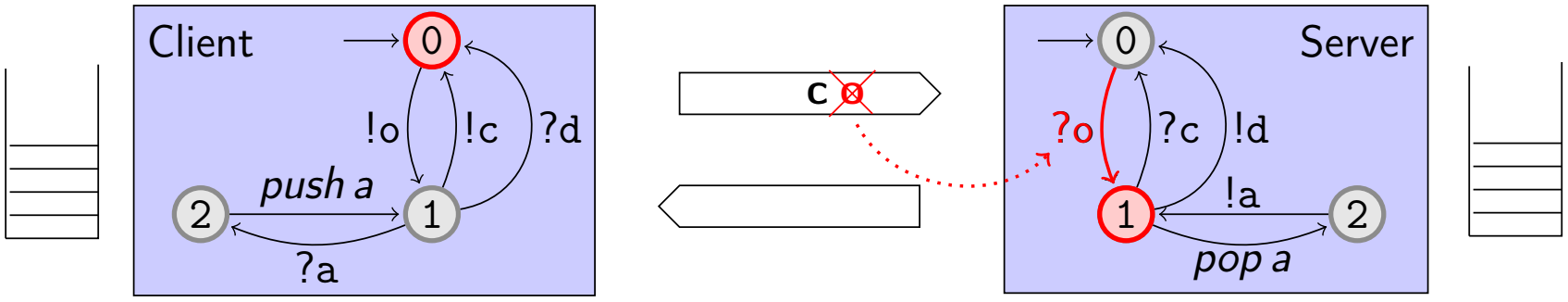


Formalize by Recursive Communicating ProcesseS

- reliable, asynchronous, unbounded channels $\longmapsto$ fifo queues
- network $\longmapsto$ (communication) graph / architecture
- recursion $\longmapsto$ pushdown stack

Formal Model : RCPS

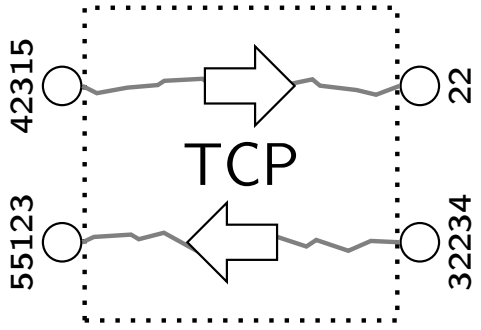


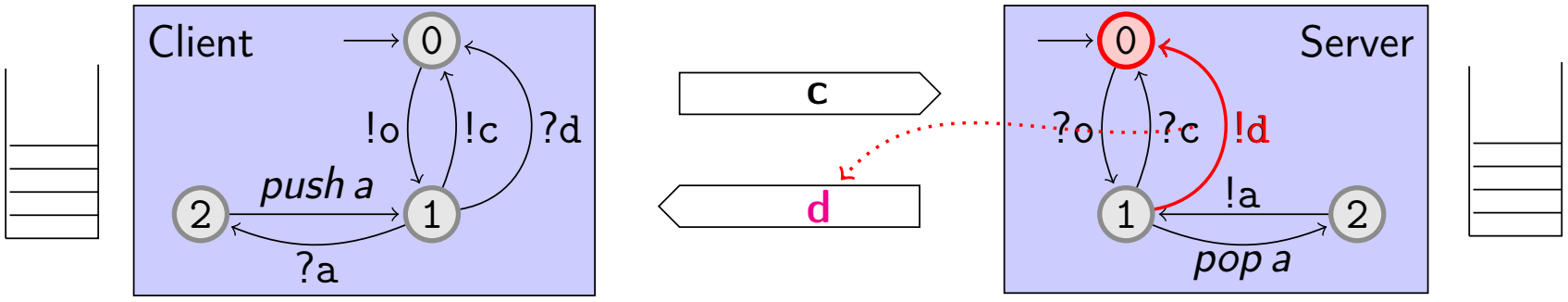


Formalize by Recursive Communicating ProcesseS

- reliable, asynchronous, unbounded channels $\longmapsto$ fifo queues
- network $\longmapsto$ (communication) graph / architecture
- recursion $\longmapsto$ pushdown stack

Formal Model : RCPS





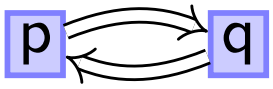
Formalize by Recursive Communicating ProcesseS

- reliable, asynchronous, unbounded channels $\longmapsto$ fifo queues
- network $\longmapsto$ (communication) graph / architecture
- recursion $\longmapsto$ pushdown stack

Reachability Verification

- reachability undecidable for finite-state RCPS ↯

  already 

  & 

  are Turing powerfull

  - need to avoid cyclic communication: impractical ☹

- and for simple non-cyclic architectures with pushdowns ↯

  already for

  : PCP/intersect. of cf langs

  - avoid any communication ? ☹ ☹ ☹

Reachability Verification

- reachability undecidable for finite-state RCPS ↯

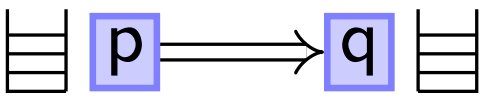
  already 

  & 

  are Turing powerfull

  ↺ need to avoid cyclic communication: impractical ☹

- and for simple non-cyclic architectures with pushdowns ↯

  already for 

  : PCP/intersect. of cf langs

  ↺ avoid any communication ? ☹ ☹ ☹

Under-Approximation

Idea:

Restrict focus only on subset of possible runs that have a certain form (under-approximate behaviour).

- ➪ try to find restriction on runs that have
  - (i) good theoretical properties (decidable reachability etc.)
  - (ii) is "applicable" in practical domain
  - (iii) can decide (syntactically) if restriction holds
  - (iv) algorithms can be implemented

Under-Approximation

Idea:

Restrict focus only on subset of possible runs that have a certain form (under-approximate behaviour).

➪ try to find restriction on runs that have

(i) good theoretical properties (decidable reachability etc.)

(ii) is "applicable" in practical domain

(iii) can decide (syntactically) if restriction holds

(iv) algorithms can be implemented

Under-Approximation

Idea:

Restrict focus only on subset of possible runs that have a certain form (under-approximate behaviour).

➪ try to find restriction on runs that have

(i) good theoretical properties (decidable reachability etc.)

(ii) is "applicable" in practical domain

(iii) can decide (syntactically) if restriction holds

(iv) algorithms can be implemented

Under-Approximation

Idea:

Restrict focus only on subset of possible runs that have a certain form (under-approximate behaviour).

➪ try to find restriction on runs that have

(i) good theoretical properties (decidable reachability etc.)

(ii) is "applicable" in practical domain

(iii) can decide (syntactically) if restriction holds

(iv) algorithms can be implemented

Under-Approximation

Idea:

Restrict focus only on subset of possible runs that have a certain form (under-approximate behaviour).

➪ try to find restriction on runs that have

(i) good theoretical properties (decidable reachability etc.)

(ii) is "applicable" in practical domain

(iii) can decide (syntactically) if restriction holds

(iv) algorithms can be implemented

Under-Approximation

Idea:

Restrict focus only on subset of possible runs that have a certain form (under-approximate behaviour).

➪ try to find restriction on runs that have

(i) good theoretical properties (decidable reachability etc.)

(ii) is "applicable" in practical domain

(iii) can decide (syntactically) if restriction holds

(iv) algorithms can be implemented

Under-Approximation

Idea:

Restrict focus only on subset of possible runs that have a certain form (under-approximate behaviour).

➪ try to find restriction on runs that have

(i) good theoretical properties (decidable reachability etc.)

(ii) is "applicable" in practical domain

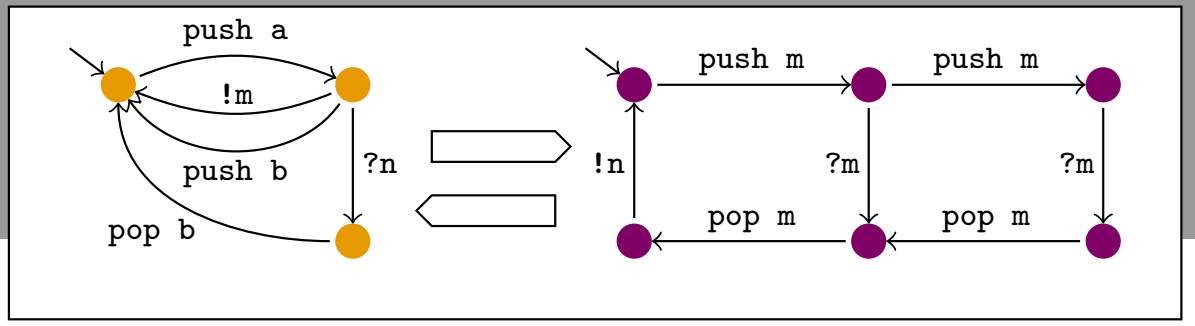
(iii) can decide (syntactically) if restriction holds

(iv) algorithms can be implemented

Runs of RCPS

- interleaving semantics (run = altern. seq. of states/actions)
- event-based/partial-order semantics (run graph)

Runs of RCPS

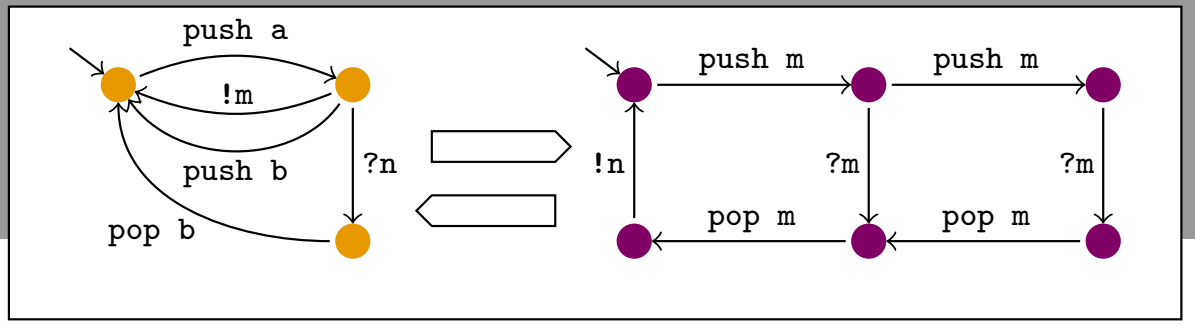


- interleaving semantics (run = altern. seq. of states/actions)

$$s_0, \mathit{push}\ a, s_2, !m, s_3, \mathit{push}\ b, s_4, \mathit{push}\ m, s_5, ?m, s_6, \mathit{pop}\ b, s_7, \ldots$$
$$\ldots, s_7, \mathit{pop}\ m, s_8, !n, s_9, ?n, s_{10}, \mathit{push}\ m, s_{11}, \mathit{pop}\ a, s_{12}$$

- event-based/partial-order semantics (run graph)

Runs of RCPS

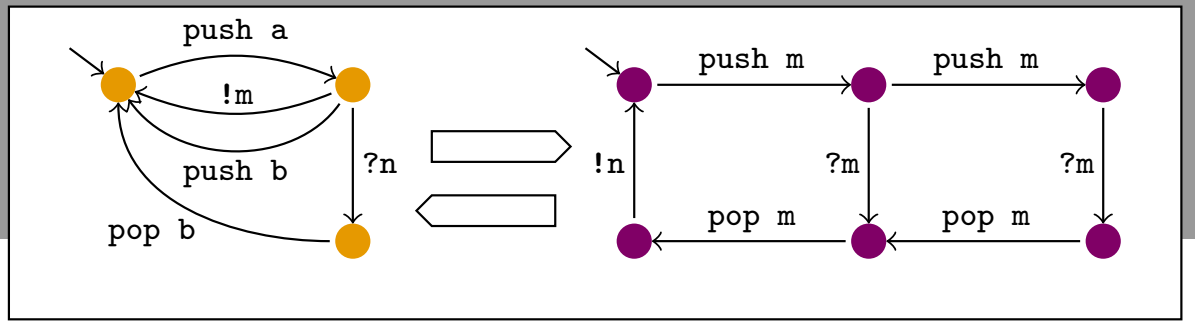


- interleaving semantics (run = altern. seq. of states/actions)

s_0, *push a*, s_2, *!m*, s_3, *push b*, s_4, *push m*, s_5, *?m*, s_6, *pop b*, s_7, ...
..., s_7, *pop m*, s_8, *!n*, s_9, *?n*, s_{10}, *push m*, s_{11}, *pop a*, s_{12}

- event-based/partial-order semantics (run graph)

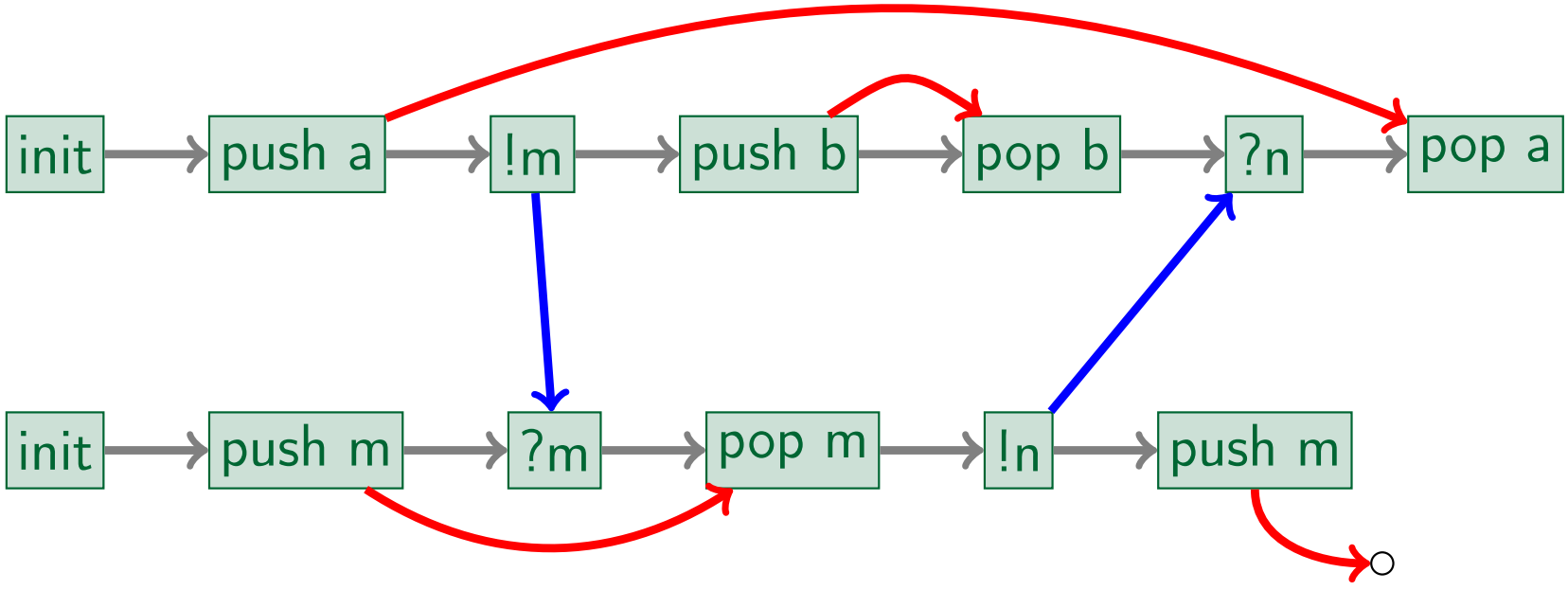
Runs of RCPS



- interleaving semantics (run = altern. seq. of states/actions)

s_0, *push a*, s_2, *!m*, s_3, *push b*, s_4, *push m*, s_5, *?m*, s_6, *pop b*, s_7, ...
..., s_7, *pop m*, s_8, *!n*, s_9, *?n*, s_{10}, *push m*, s_{11}, *pop a*, s_{12}

- event-based/partial-order semantics (run graph)



Boundedness Conditions: Interleaving

- “ressource”-boundedness
  - atomic communication
  - untangle pushdown and communication actions
  - architecture: tree-like, acyclic,. . .
  - mixed: half-duplex,. . .

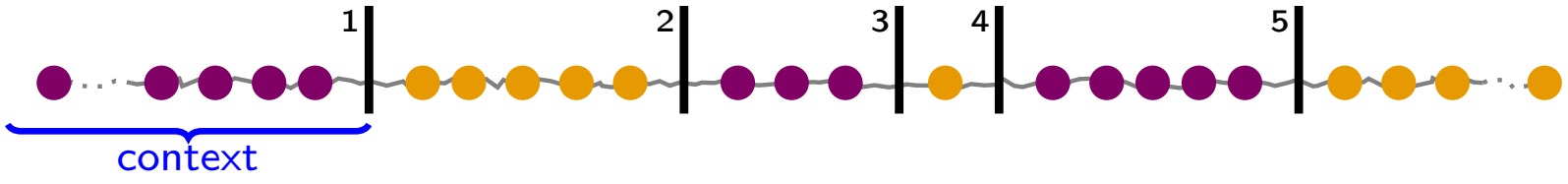
- family of phase-boundedness restrictions:
  e.g., contexts [Qadeer/Rehof’05]

- reachability decidable in $\mathcal{O}(\text{ExpTime} / \text{ExpSpace})$
- rarely decidability results beyond reachability

Boundedness Conditions: Interleaving

- “ressource”-boundedness
  - atomic communication
  - untangle pushdown and communication actions
  - architecture: tree-like, acyclic,. . .
  - mixed: half-duplex,. . .

- family of phase-boundedness restrictions:
  e.g., contexts [Qadeer/Rehof’05]



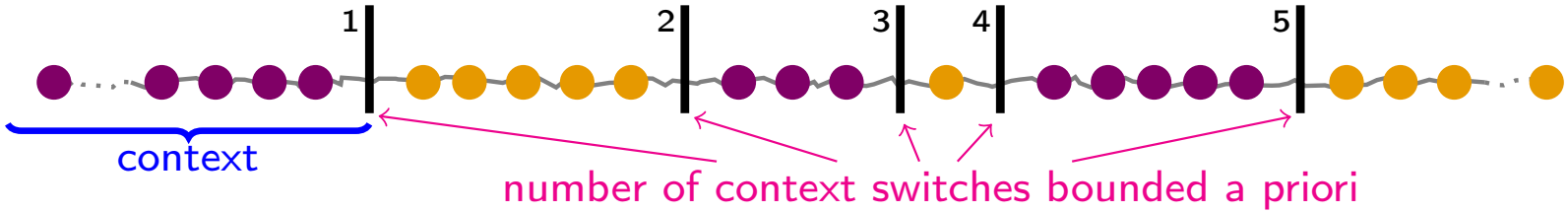
☺ reachability decible in ∼(2)ExpTime / find “shallow” errors

☹ rarely decidability results beyond reachability. . .

Boundedness Conditions: Interleaving

- “ressource”-boundedness
  - atomic communication
  - untangle pushdown and communication actions
  - architecture: tree-like, acyclic,. . .
  - mixed: half-duplex,. . .

- family of phase-boundedness restrictions:
  e.g., contexts [Qadeer/Rehof’05]



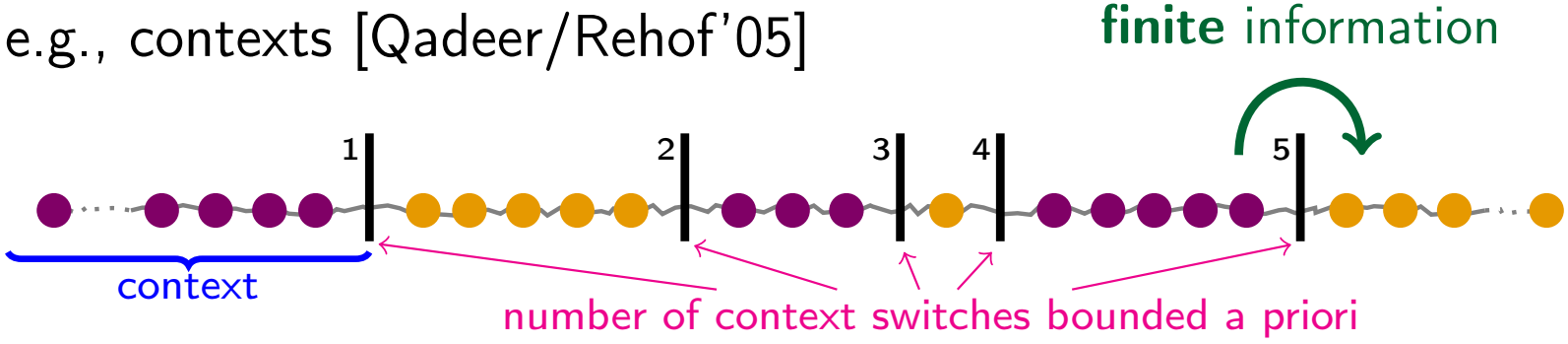
☺ reachability decible in ∼(2)ExpTime / find “shallow” errors

☹ rarely decidability results beyond reachability. . .

Boundedness Conditions: Interleaving

- “ressource”-boundedness
  - atomic communication
  - untangle pushdown and communication actions
  - architecture: tree-like, acyclic,. . .
  - mixed: half-duplex,. . .

- family of phase-boundedness restrictions:
  e.g., contexts [Qadeer/Rehof’05]



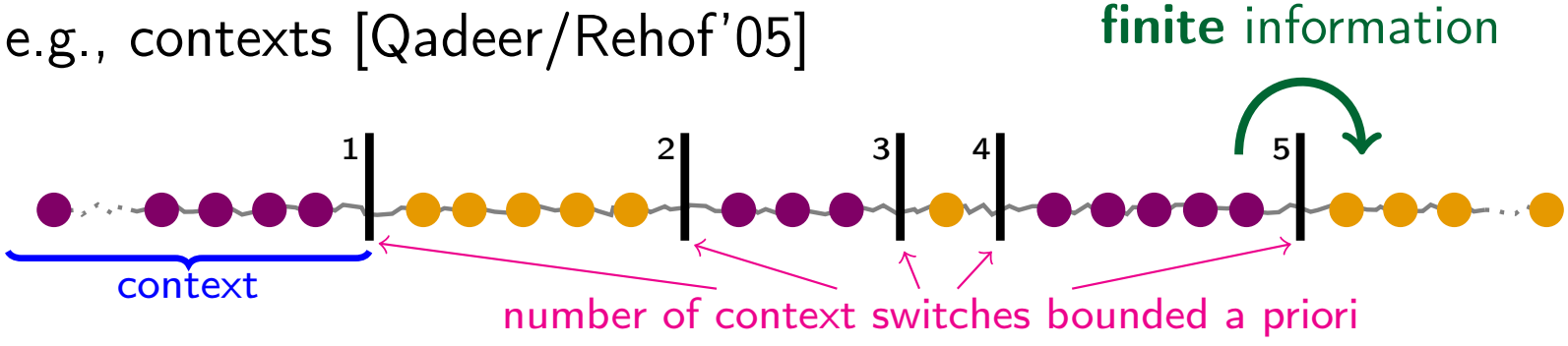
☺ reachability decible in ∼(2)ExpTime / find “shallow” errors

☹ rarely decidability results beyond reachability. . .

Boundedness Conditions: Interleaving

- “ressource”-boundedness
  - atomic communication
  - untangle pushdown and communication actions
  - architecture: tree-like, acyclic,...
  - mixed: half-duplex,...

- family of phase-boundedness restrictions:
  e.g., contexts [Qadeer/Rehof’05]



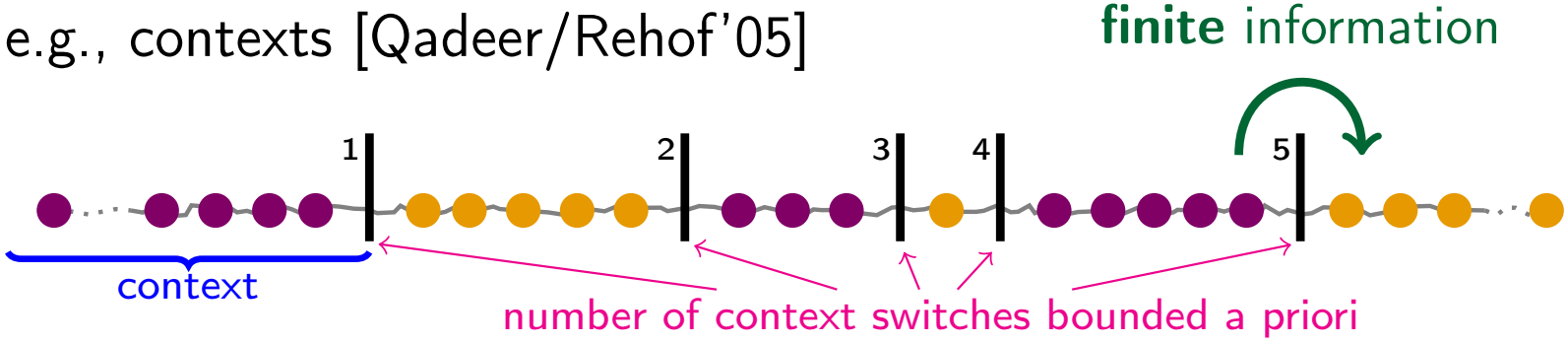
☺ reachability decible in ∼(2)ExpTime / find “shallow” errors

☹ rarely decidability results beyond reachability...

Boundedness Conditions: Interleaving

- “ressource”-boundedness
  - atomic communication
  - untangle pushdown and communication actions
  - architecture: tree-like, acyclic,. . .
  - mixed: half-duplex,. . .

- family of phase-boundedness restrictions:
  e.g., contexts [Qadeer/Rehof’05]



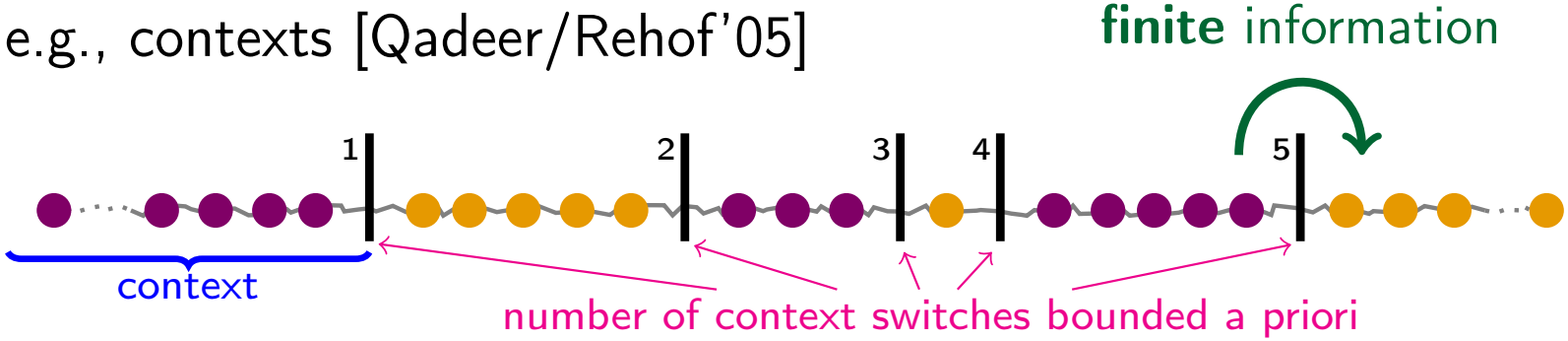
☺ reachability decible in ∼(2)ExpTime / find “shallow” errors

☹ rarely decidability results beyond reachability. . .

Boundedness Conditions: Interleaving

- “ressource”-boundedness
  - atomic communication
  - untangle pushdown and communication actions
  - architecture: tree-like, acyclic,. . .
  - mixed: half-duplex,. . .

- family of phase-boundedness restrictions:
  e.g., contexts [Qadeer/Rehof’05]



☺ reachability decible in ∼(2)ExpTime / find “shallow” errors

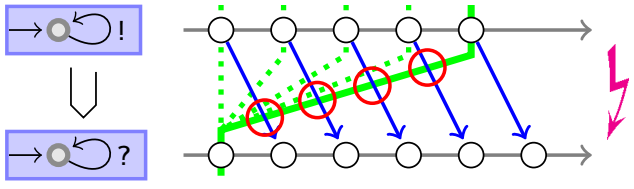
☹ rarely decidability results beyond reachability. . .

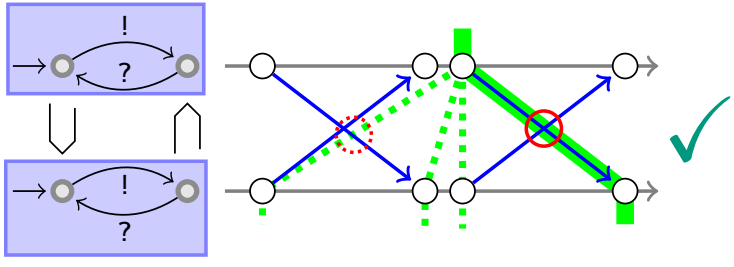
Boundedness: Partial-Order Case

- bounded MSC (= bounded finite-state RCPS)
  - fix size of channels a priori: finite system ✓
  - semantically fix channel usage:
    for all prefixes of any run the number of unreceived messages on each channel is less than or equal to b (for fix $b \in \mathbb{N}$)
  - allows to decompose MSC into "building blocks"
  - LTL/MSO model checking decidable ✓

Boundedness: Partial-Order Case

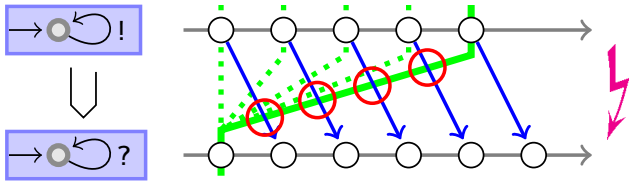
- ➪ bounded MSC (= bounded finite-state RCPS)
  - fix size of channels a priori: finite system ✓
  - semantically fix channel usage:
    for all prefixes of any run the number of unreceived messages on each channel is less than or equal to b (for fix $b \in \mathbb{N}$)
  - allows to decompose MSC into "building blocks"
  - LTL/MSO model checking decidable ✓

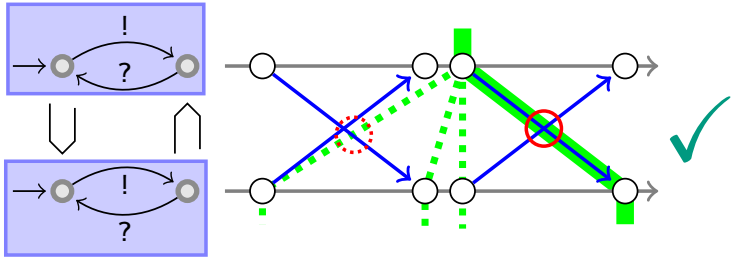




Boundedness: Partial-Order Case

- bounded MSC (= bounded finite-state RCPS)
  - fix size of channels a priori: finite system ✓
  - semantically fix channel usage:
    for all prefixes of any run the number of unreceived messages on each channel is less than or equal to b (for fix $b \in \mathbb{N}$)
  - allows to decompose MSC into "building blocks"
  - LTL/MSO model checking decidable ✓

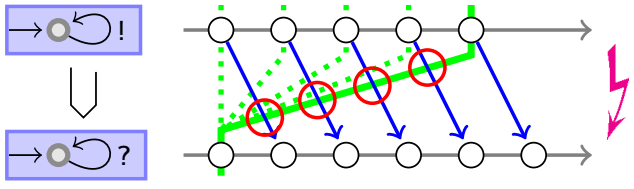


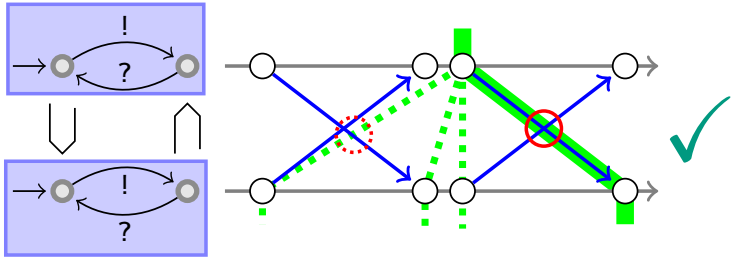


Boundedness: Partial-Order Case

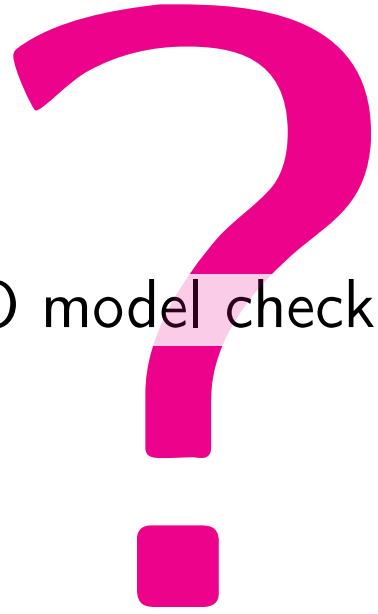
➪ bounded MSC (= bounded finite-state RCPS)

- fix size of channels a priori: finite system ✓
- semantically fix channel usage:
  for all prefixes of any run the number of unreceived messages on each channel is less than or equal to b (for fix $b \in \mathbb{N}$)
- allows to decompose MSC into "building blocks"
- LTL/MSO model checking decidable ✓

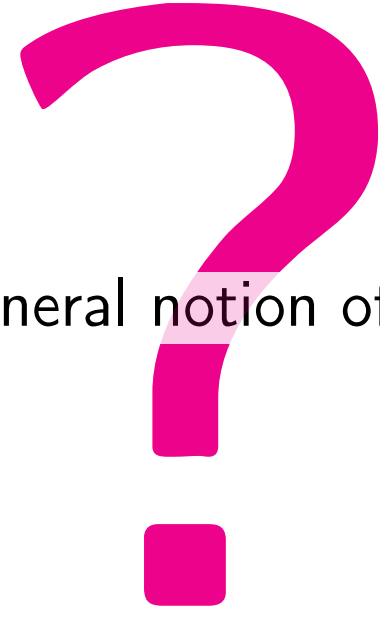




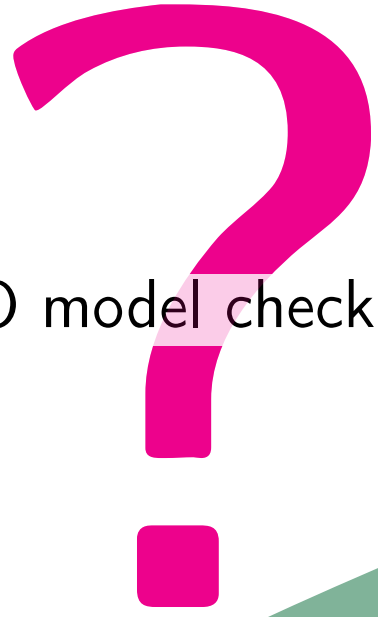
How to extend MSO model checking ideas to RCPS ?



Derive more general notion of boundedness ?



How to extend MSO model checking ideas to RCPS ?



goal: proof one, get one for free !

Derive more general notion of boundedness ?

Again: Undecidability for finite RCPS

Theorem [Seese '75] :

Let $\mathcal{C}_G$ be a class of graphs such that for every integer $k > 1$ there is a graph $G \in \mathcal{C}_G$ such that G has the $k \times k$ grid as induced subgraph[1]. Then MSO is undecidable on $\mathcal{C}_G$.

- MSO undecidable on rungraphs of finite-state RCPS as one can embedd the $k \times k$ grid already on

,e.g., for $k = 3$

$$G_{3\times 3} = \begin{array}{ccc} 1 & \to 2 & \to 3 \\ \downarrow & \downarrow & \downarrow \\ 4 & \to 5 & \to 6 \\ \downarrow & \downarrow & \downarrow \\ 7 & \to 8 & \to 9 \end{array}$$

- ressembles Turing completeness proof [Brand/Zafiropoulo '83]

[1] induced subgraph can be extended to minor+induced subgraph.

Again: Undecidability for finite RCPS

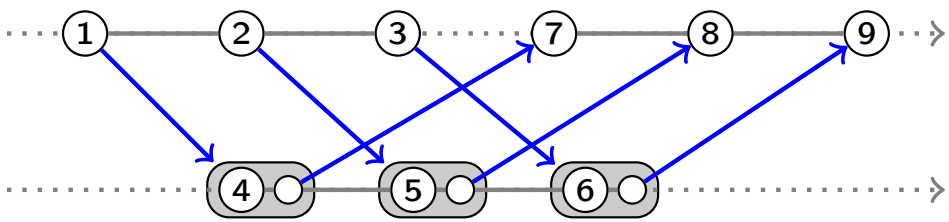
Theorem [Seese '75] :

Let $\mathcal{C}_G$ be a class of graphs such that for every integer $k > 1$ there is a graph $G \in \mathcal{C}_G$ such that G has the $k \times k$ grid as induced subgraph[1]. Then MSO is undecidable on $\mathcal{C}_G$.

- MSO undecidable on rungraphs of finite-state RCPS as one can embedd the $k \times k$ grid already on 

,e.g., for $k = 3$

$$G_{3\times 3} = \begin{array}{ccccc} 1 & \rightarrow & 2 & \rightarrow & 3 \\ \downarrow & & \downarrow & & \downarrow \\ 4 & \rightarrow & 5 & \rightarrow & 6 \\ \downarrow & & \downarrow & & \downarrow \\ 7 & \rightarrow & 8 & \rightarrow & 9 \end{array}$$



- ressembles Turing completeness proof [Brand/Zafiropoulo '83]

[1] induced subgraph can be extended to minor+induced subgraph.

Again: Undecidability for finite RCPS

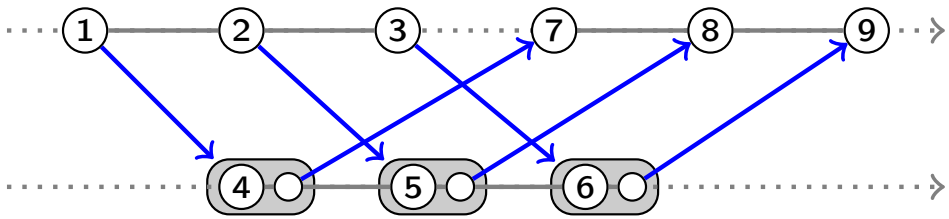
Theorem [Seese '75] :

Let $\mathcal{C}_G$ be a class of graphs such that for every integer $k > 1$ there is a graph $G \in \mathcal{C}_G$ such that G has the $k \times k$ grid as induced subgraph[1]. Then MSO is undecidable on $\mathcal{C}_G$.

- MSO undecidable on rungraphs of finite-state RCPS as one can embedd the $k \times k$ grid already on 

,e.g., for $k = 3$

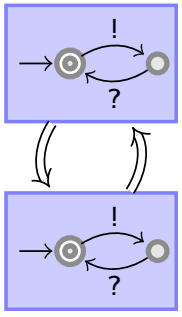
$$G_{3\times 3} = \begin{array}{ccccc} 1 & \rightarrow & 2 & \rightarrow & 3 \\ \downarrow & & \downarrow & & \downarrow \\ 4 & \rightarrow & 5 & \rightarrow & 6 \\ \downarrow & & \downarrow & & \downarrow \\ 7 & \rightarrow & 8 & \rightarrow & 9 \end{array}$$



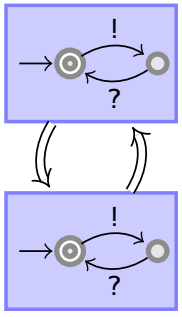
- ressembles Turing completeness proof [Brand/Zafiropoulo '83]

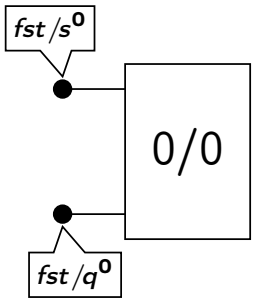
[1] induced subgraph can be extended to minor+induced subgraph.

Rungraph of bounded finite-state RCPS

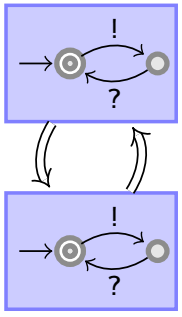


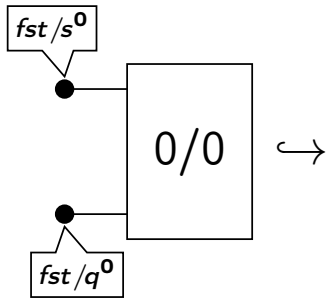
Rungraph of bounded finite-state RCPS



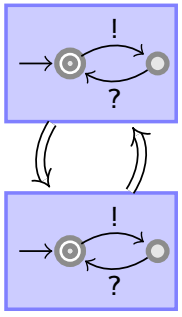


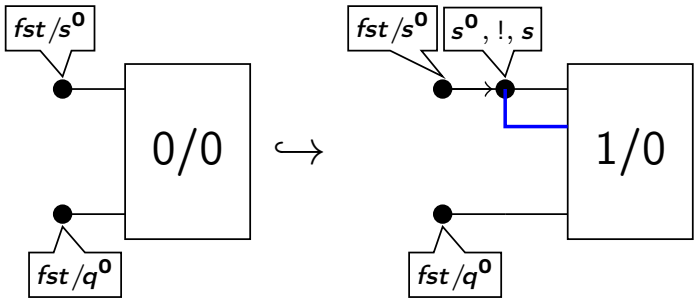
Rungraph of bounded finite-state RCPS



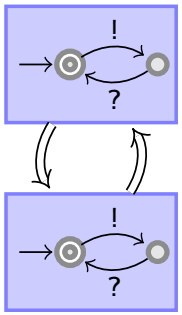


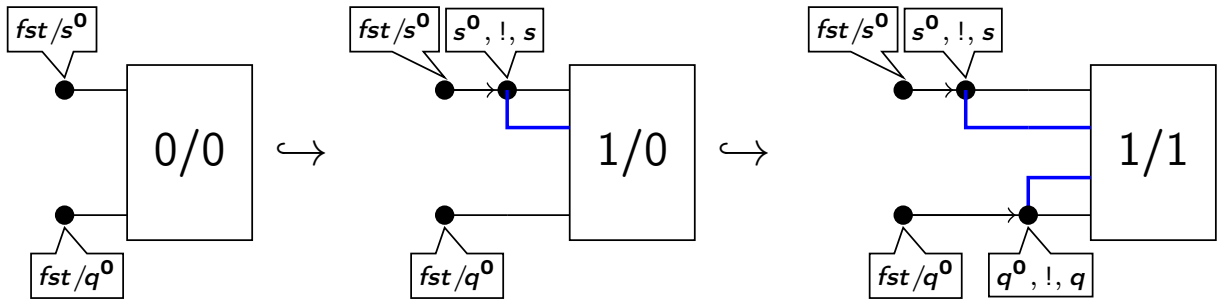
Rungraph of bounded finite-state RCPS



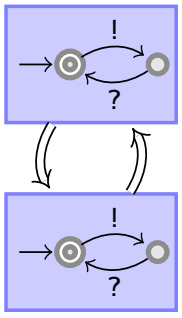


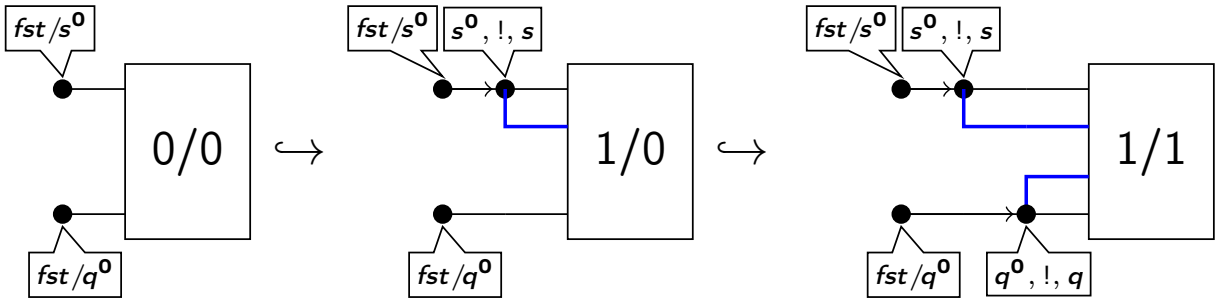
Rungraph of bounded finite-state RCPS

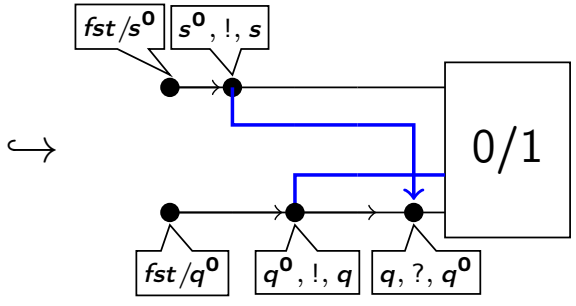




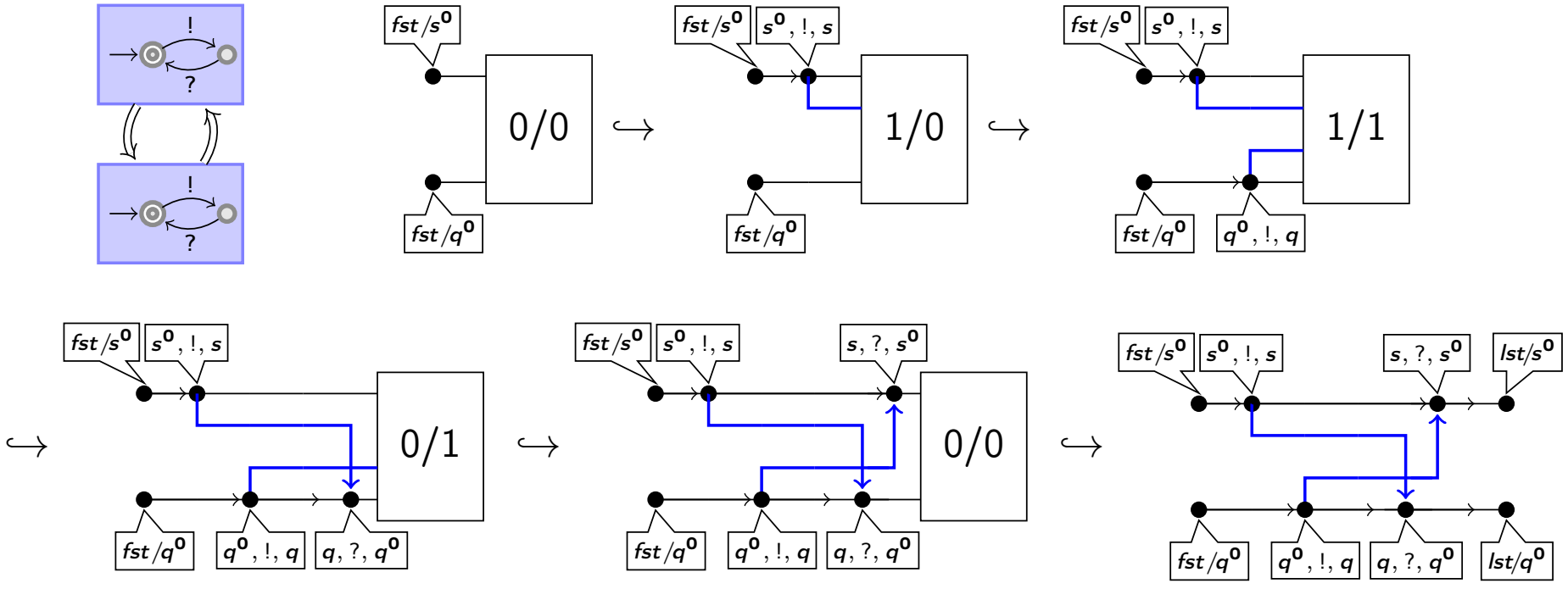
Rungraph of bounded finite-state RCPS





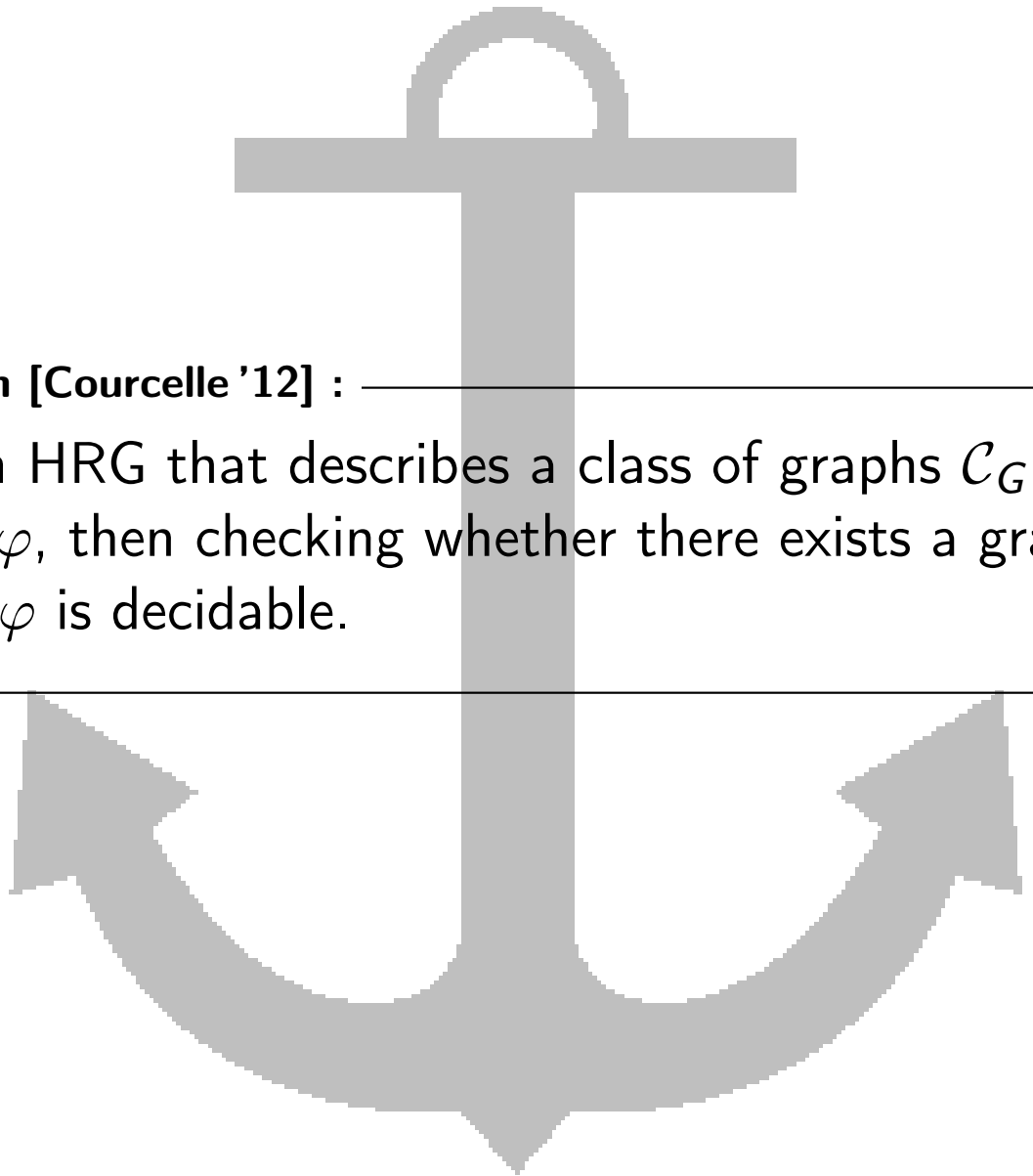


Rungraph of bounded finite-state RCPS



➪ reminds of derivations of (context-free) graph grammars ☺

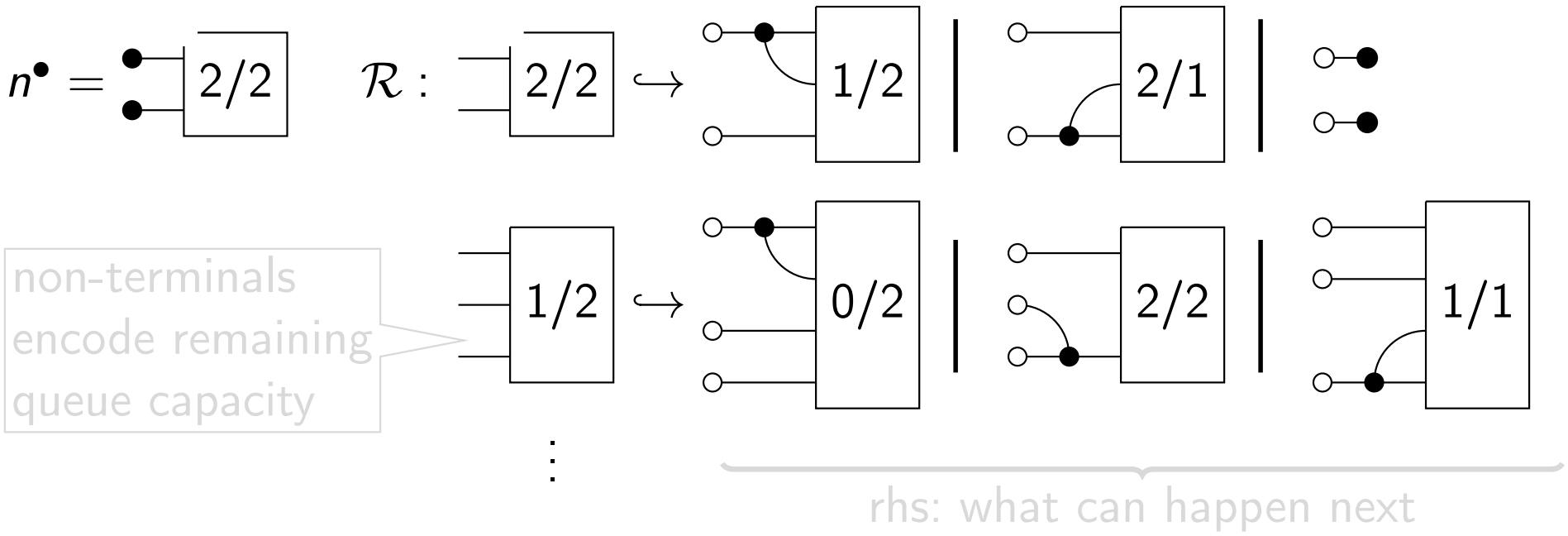
From HRG to MSO Decidability



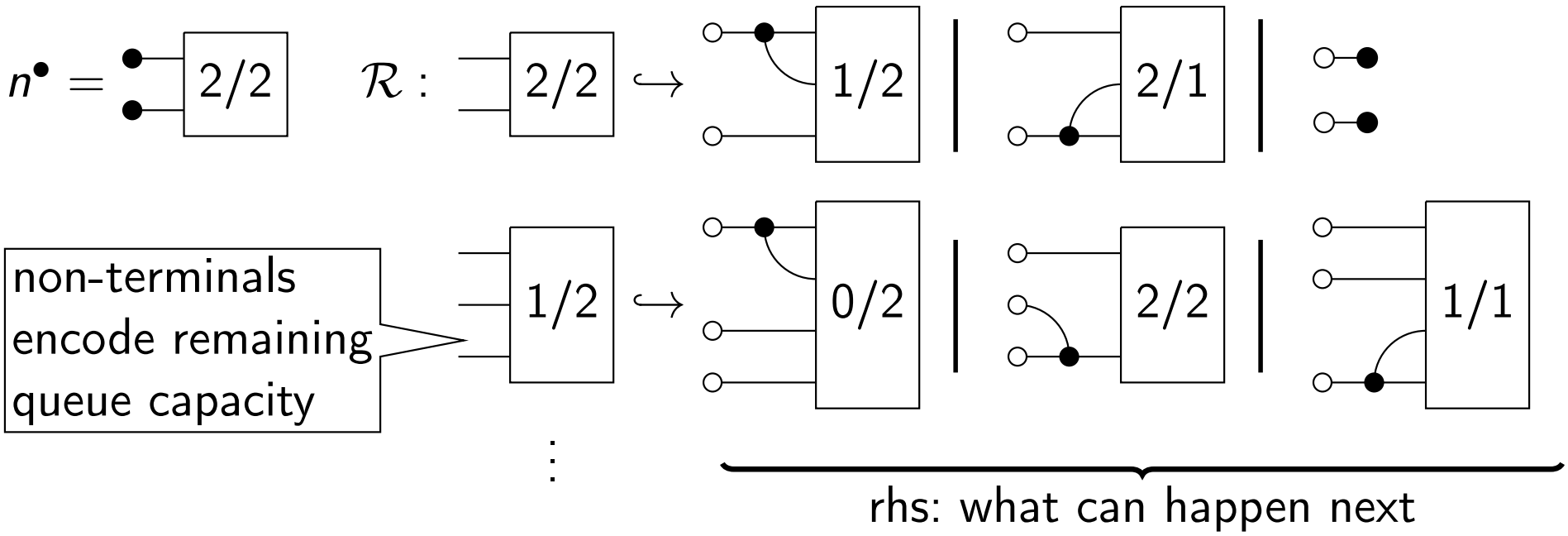
Theorem [Courcelle '12] :

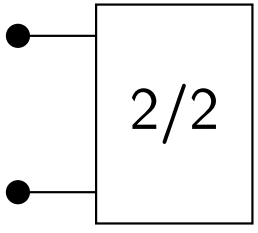
Given an HRG that describes a class of graphs $\mathcal{C}_G$, and an MSO formula φ, then checking whether there exists a graph in $\mathcal{C}_G$ that satisfies φ is decidable.

A HRG for bounded finite RCPS

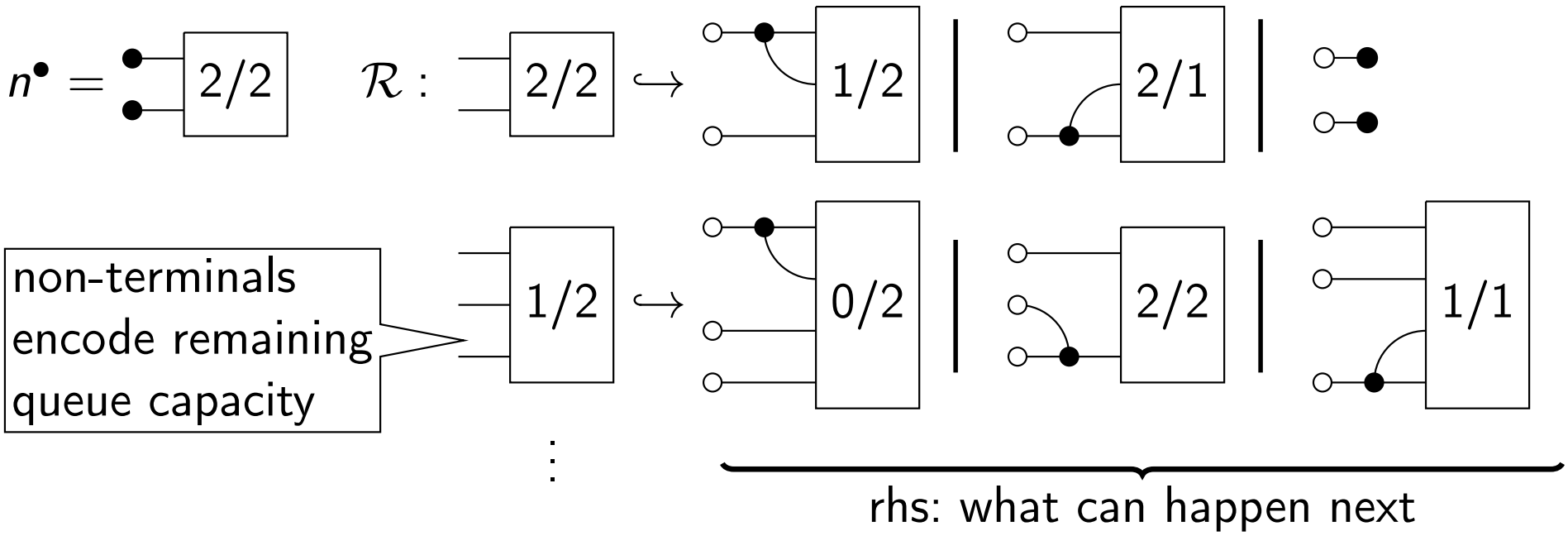


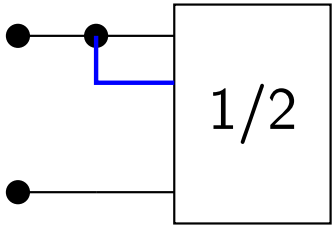
A HRG for bounded finite RCPS



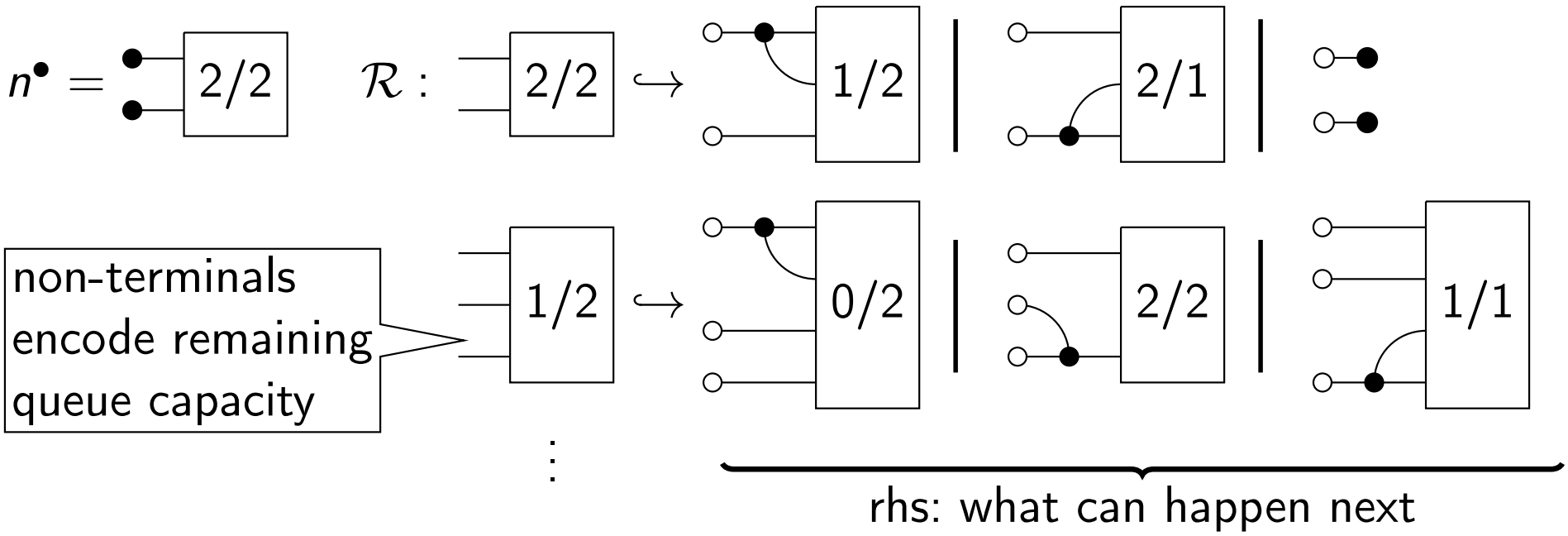


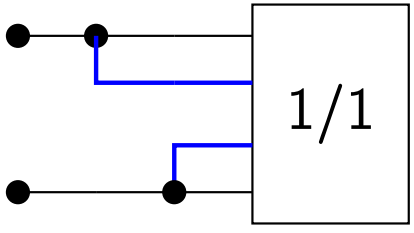
A HRG for bounded finite RCPS



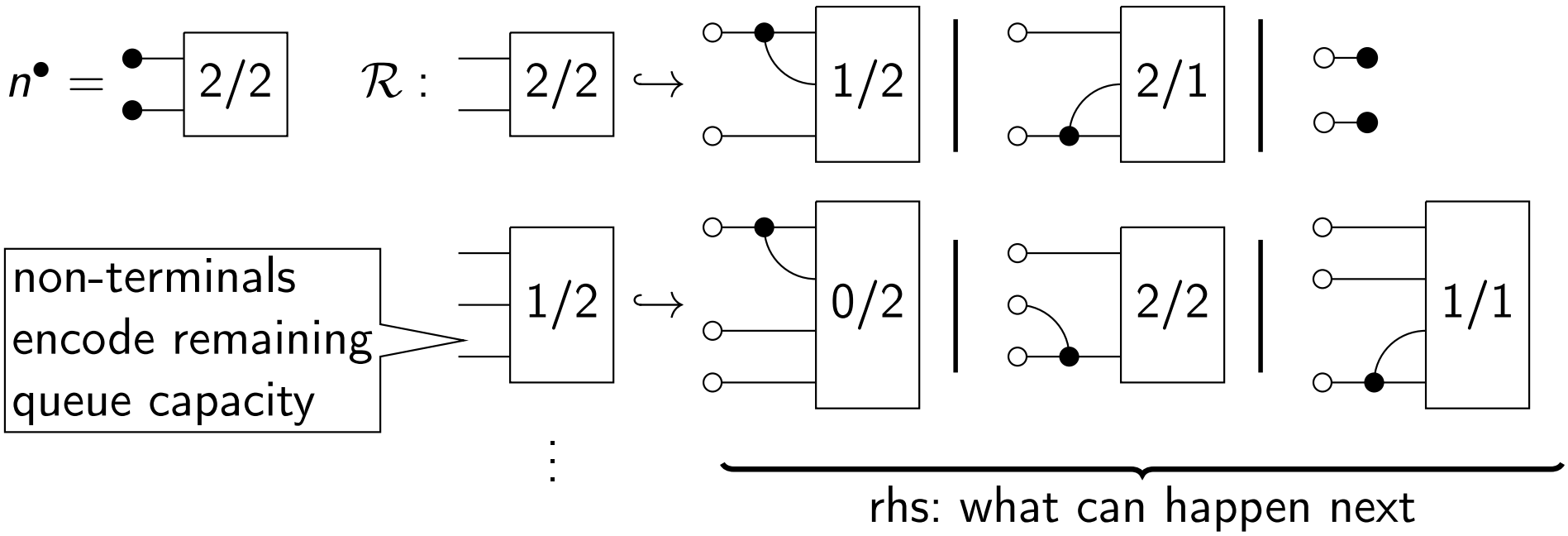


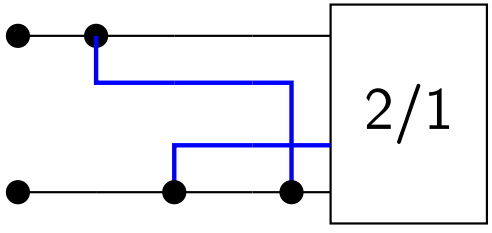
A HRG for bounded finite RCPS



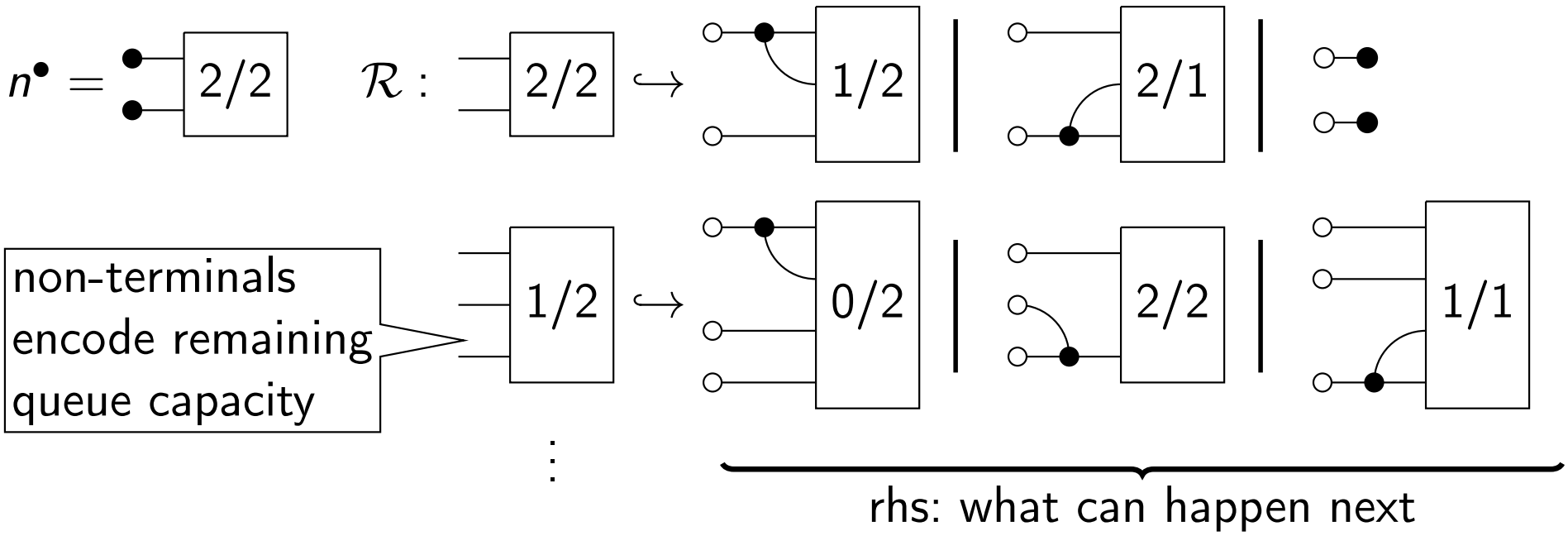


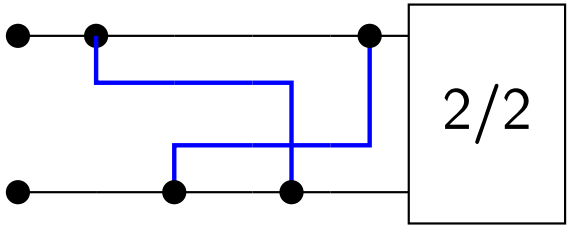
A HRG for bounded finite RCPS



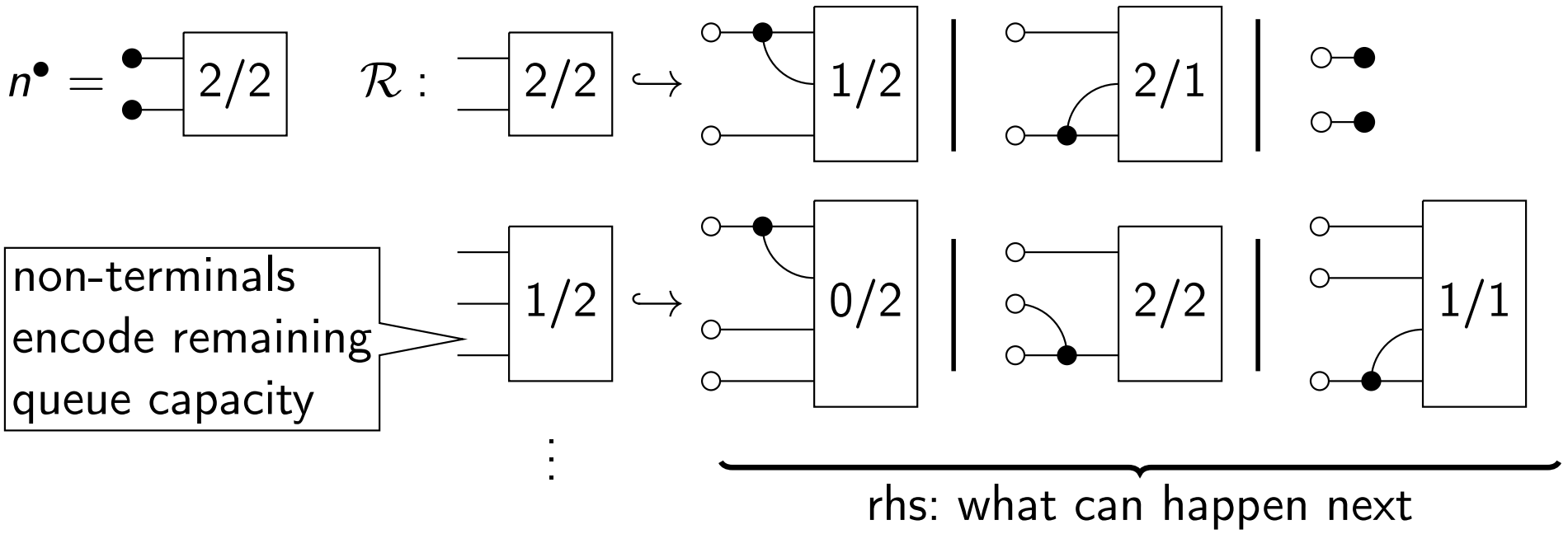


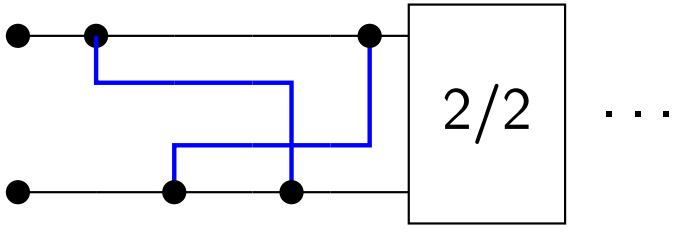
A HRG for bounded finite RCPS



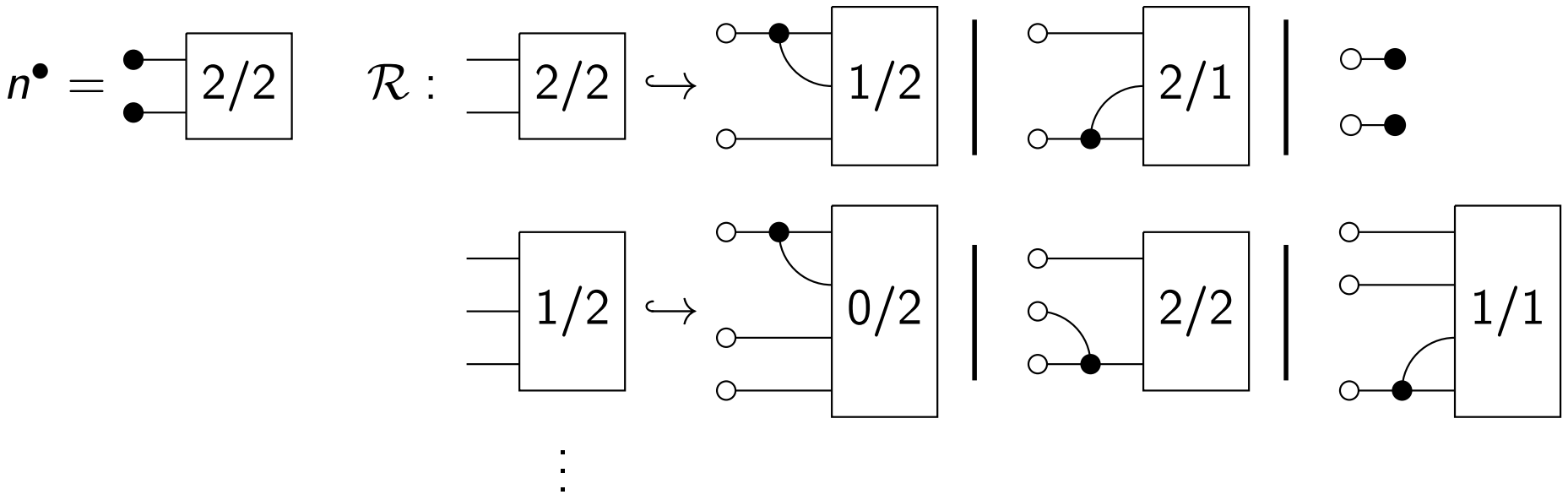


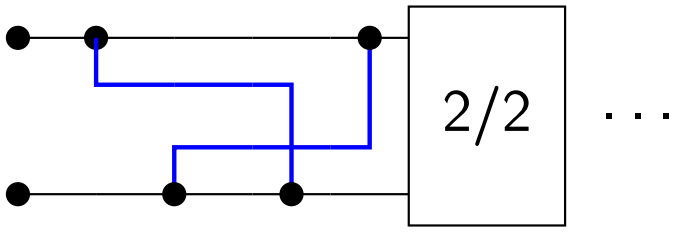
A HRG for bounded finite RCPS





A HRG for bounded finite RCPS





- HRG generates rungraphs of 2-bounded finite RCPS of 2 processes
- generalize: p processes and bound k
- rulewidth of run graphs of bounded finite RCPS is in $\mathcal{O}(p \cdot k)$

From HRG to MSO Decidability

Proposition:

The MSO model checking problem is decidable for rungraphs of bounded finite-state RCPS.

➪ already known [Madhusudan'03] [Genest et al.'07],
but here new "style" of proof ☺

➪ by connection of HRG rulewidth with treewidth:

Corollary:

MSC of bounded finite-state RCPS have bounded treewidth.

➪ bounded treewidth more general notion of semantic boundedness ?

From HRG to MSO Decidability

Proposition:

The MSO model checking problem is decidable for rungraphs of bounded finite-state RCPS.

➪ already known [Madhusudan'03] [Genest et al.'07],
but here new "style" of proof ☺

➪ by connection of HRG rulewidth with treewidth:

Corollary:

MSC of bounded finite-state RCPS have bounded treewidth.

➪ bounded treewidth more general notion of semantic boundedness ?

From HRG to MSO Decidability

Proposition:

The MSO model checking problem is decidable for rungraphs of bounded finite-state RCPS.

➪ already known [Madhusudan'03] [Genest et al.'07],
but here new "style" of proof ☺

➪ by connection of HRG rulewidth with treewidth:

Corollary:

MSC of bounded finite-state RCPS have bounded treewidth.

➪ bounded treewidth more general notion of semantic boundedness ?

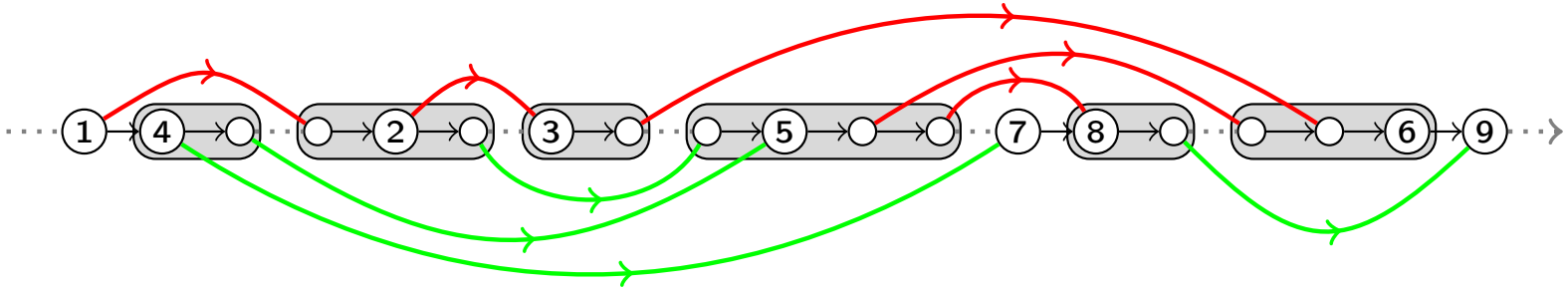
Undecidability for General RCPS

- reachability (and MSO) undecidable for RCPS ↯

  idea: infinite-grid-as-minor argument !

  - look at 2PDS (can reduce RCPS to this case)

$$G_{3\times 3} = \begin{array}{ccccc} 1 & \rightarrow & 2 & \rightarrow & 3 \\ \downarrow & & \downarrow & & \downarrow \\ 4 & \rightarrow & 5 & \rightarrow & 6 \\ \downarrow & & \downarrow & & \downarrow \\ 7 & \rightarrow & 8 & \rightarrow & 9 \end{array}$$



Ressource Bound $\mapsto$ Grammar Bound

Theorem [Heußner et al. 2010]:

Reachability is decidable for RCPS iff they are well-queueing, eager, and imbalanced. (even ExpTime-complete!)

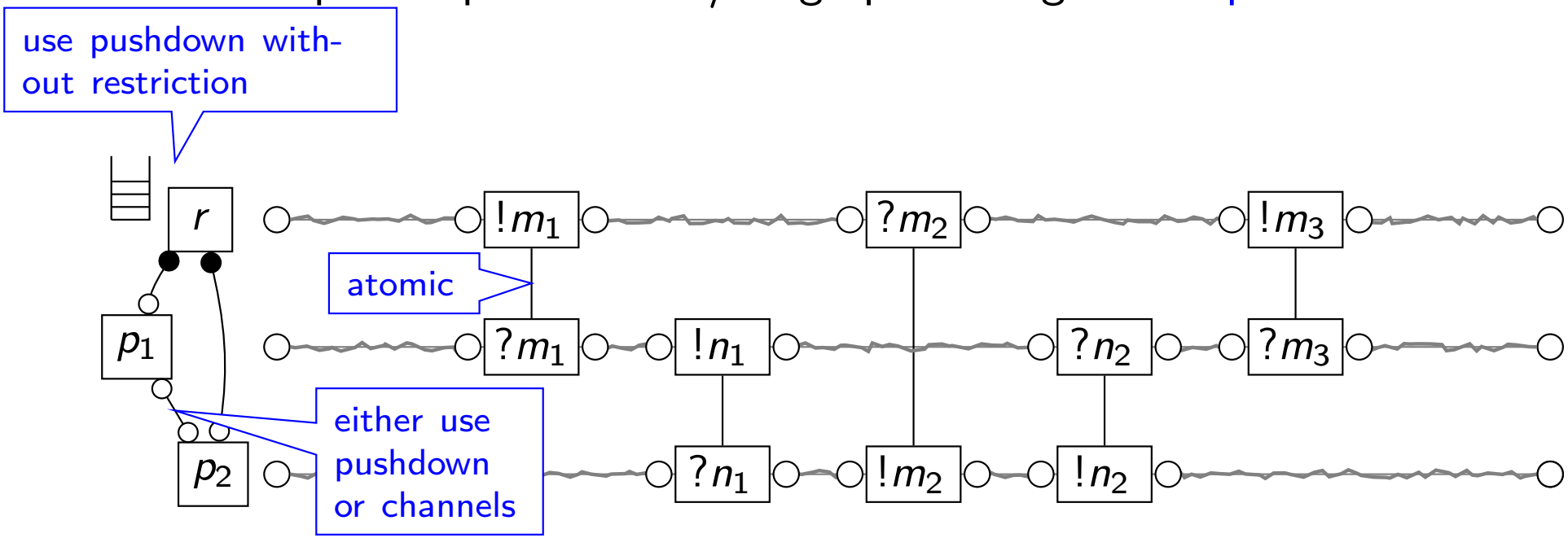
- original proof:
  commutative reordering of interleaving allows 1-stack simulation
- reinterpret in partial-order/rungraph setting: decomposition

Ressource Bound $\mapsto$ Grammar Bound

Theorem [Heußner et al. 2010]:

Reachability is decidable for RCPS iff they are well-queueing, eager, and imbalanced. (even ExpTime-complete!)

- ➪ original proof:
  commutative reordering of interleaving allows 1-stack simulation
- ➪ reinterpret in partial-order/rungraph setting: decomposition

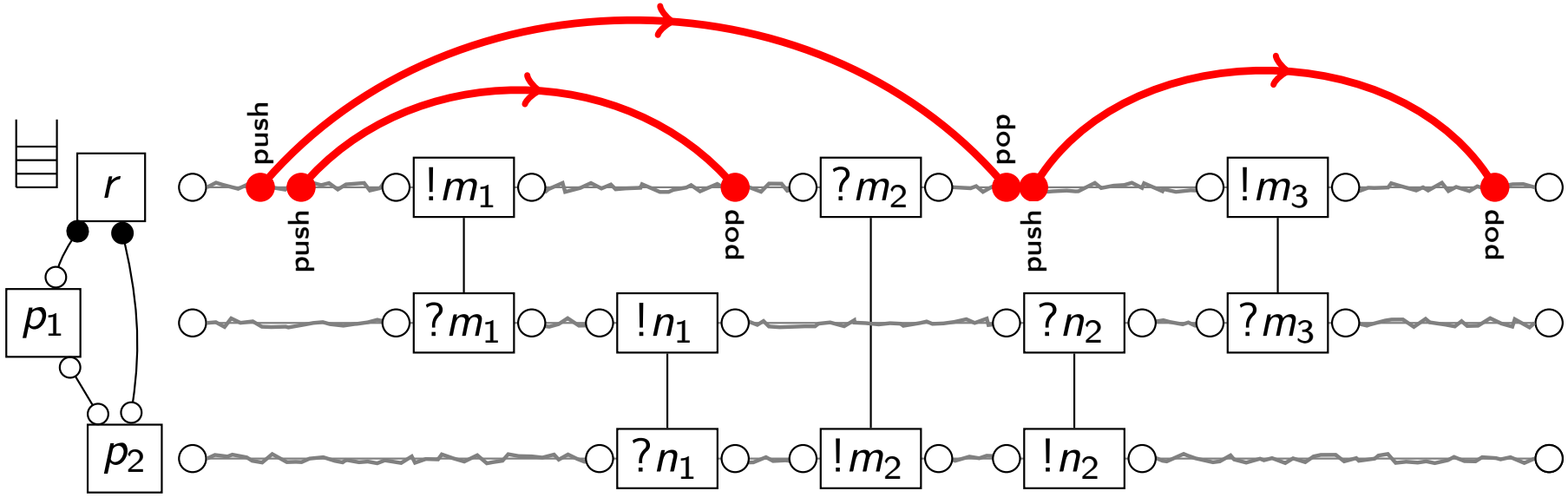


Ressource Bound $\mapsto$ Grammar Bound

Theorem [Heußner et al. 2010]:

Reachability is decidable for RCPS iff they are well-queueing, eager, and imbalanced. (even ExpTime-complete!)

- original proof:
  commutative reordering of interleaving allows 1-stack simulation
- reinterpret in partial-order/rungraph setting: decomposition

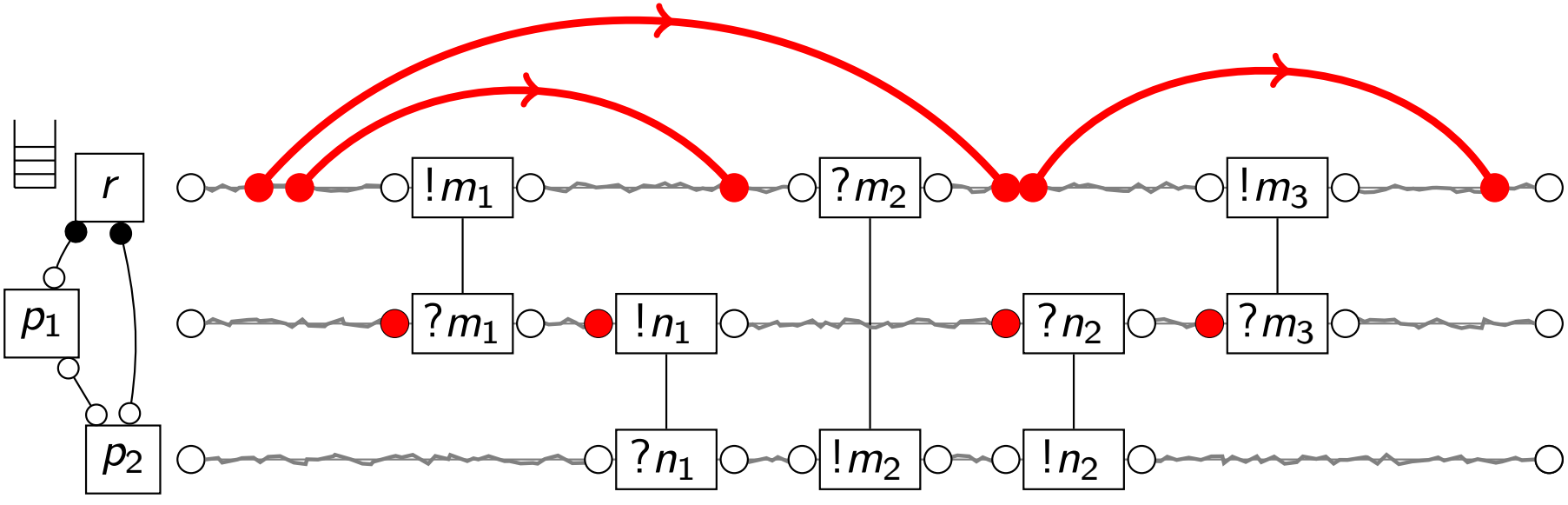


Ressource Bound $\mapsto$ Grammar Bound

Theorem [Heußner et al. 2010]:

Reachability is decidable for RCPS iff they are well-queueing, eager, and imbalanced. (even ExpTime-complete!)

- ➪ original proof:
  commutative reordering of interleaving allows 1-stack simulation
- ➪ reinterpret in partial-order/rungraph setting: decomposition

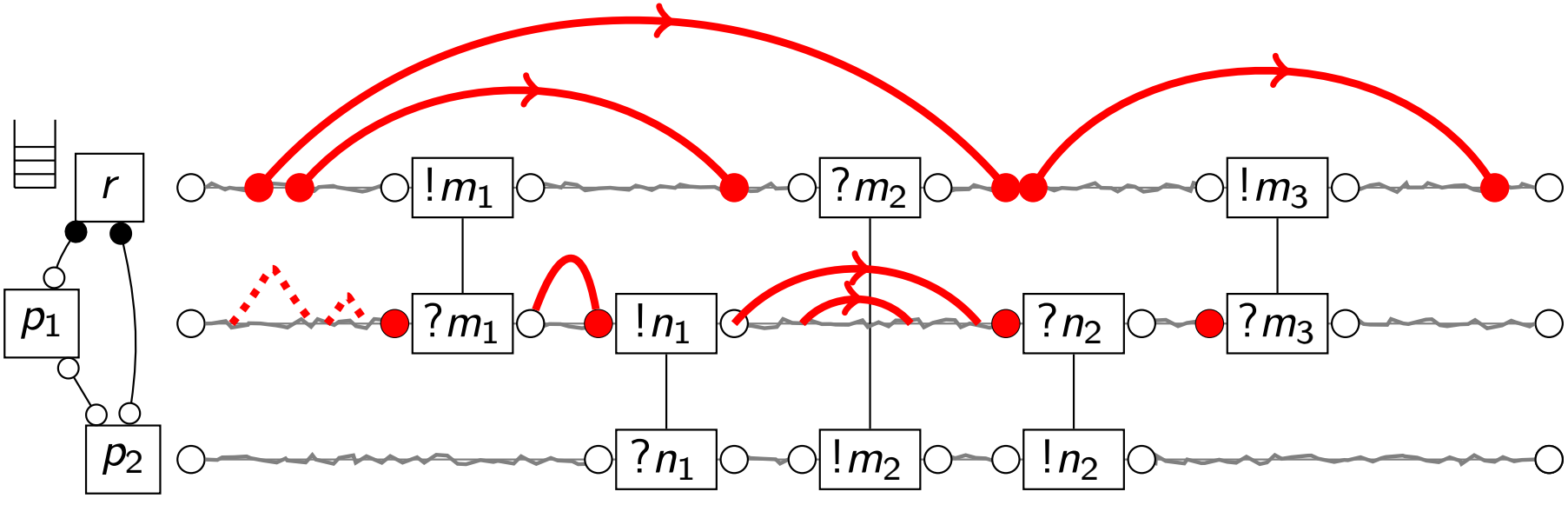


Ressource Bound $\mapsto$ Grammar Bound

Theorem [Heußner et al. 2010]:

Reachability is decidable for RCPS iff they are well-queueing, eager, and imbalanced. (even ExpTime-complete!)

- ➪ original proof:
  commutative reordering of interleaving allows 1-stack simulation
- ➪ reinterpret in partial-order/rungraph setting: decomposition

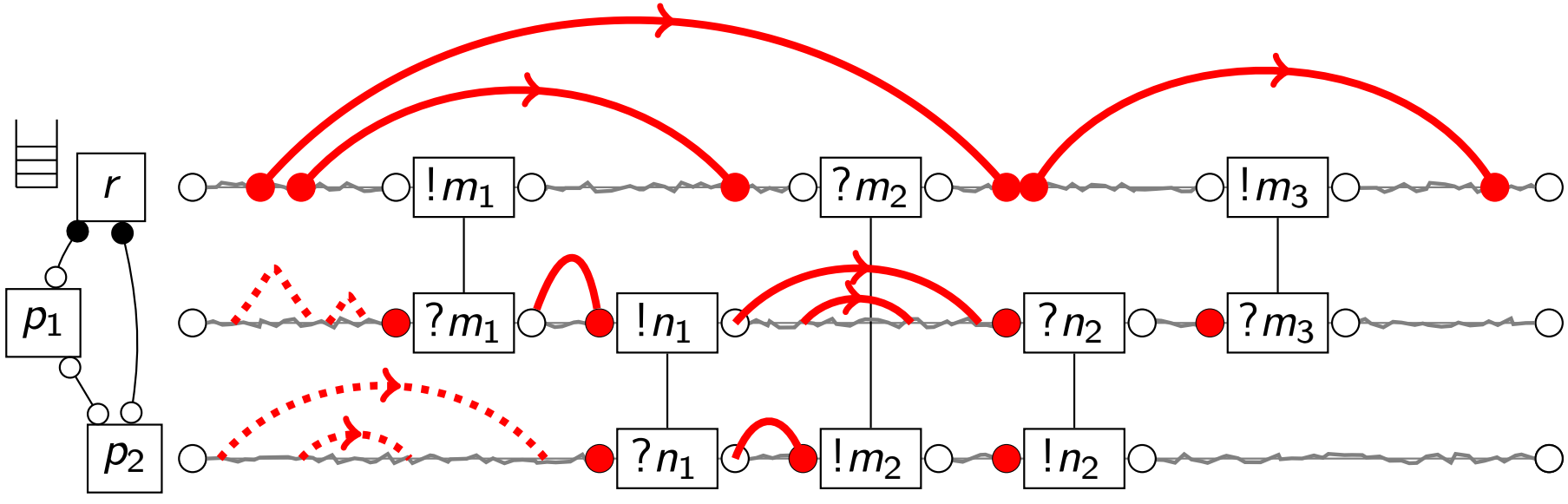


Ressource Bound $\mapsto$ Grammar Bound

Theorem [Heußner et al. 2010]:

Reachability is decidable for RCPS iff they are well-queueing, eager, and imbalanced. (even ExpTime-complete!)

- ➪ original proof:
  commutative reordering of interleaving allows 1-stack simulation
- ➪ reinterpret in partial-order/rungraph setting: decomposition

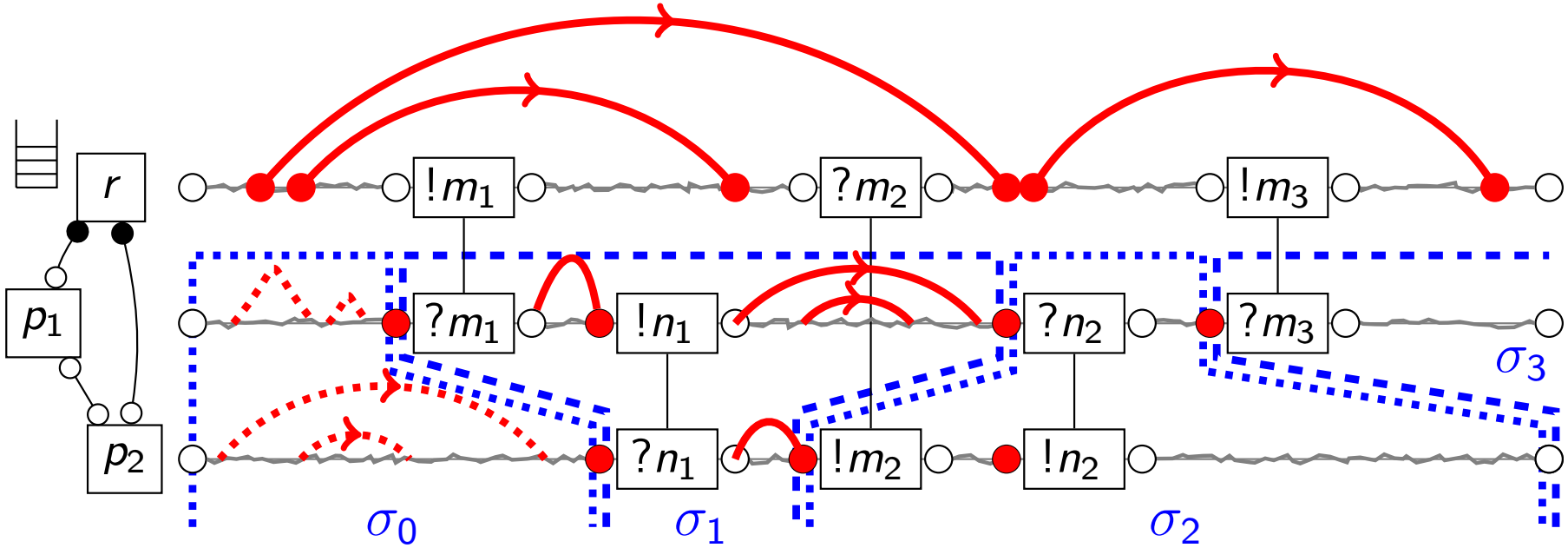


Ressource Bound $\mapsto$ Grammar Bound

Theorem [Heußner et al. 2010]:

Reachability is decidable for RCPS iff they are well-queueing, eager, and imbalanced. (even ExpTime-complete!)

- ➪ original proof:
  commutative reordering of interleaving allows 1-stack simulation
- ➪ reinterpret in partial-order/rungraph setting: decomposition



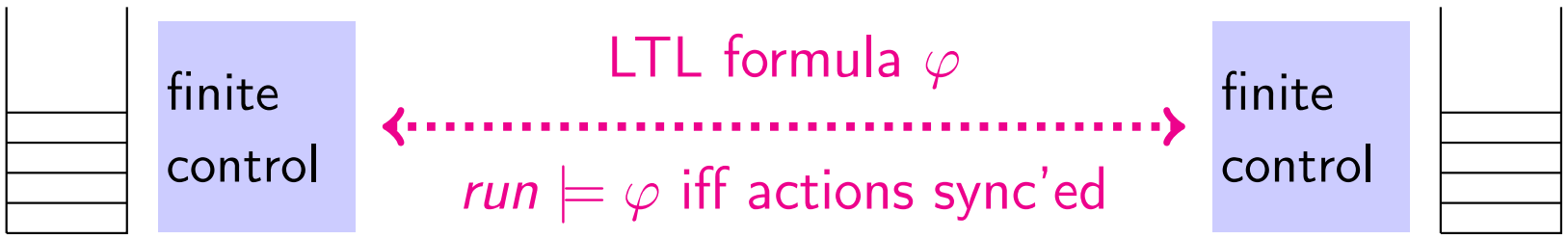
Bounded Ressources and LTL-MC

➪ however extending this result leads directly to:

Theorem:

The LTL model checking question is undecidable for RQCPS that are well-queueing, eager and imbalanced. This already holds for a weak fragment of action LTL.

➪ couple independent events by LTL interpreted on interleaving ↯



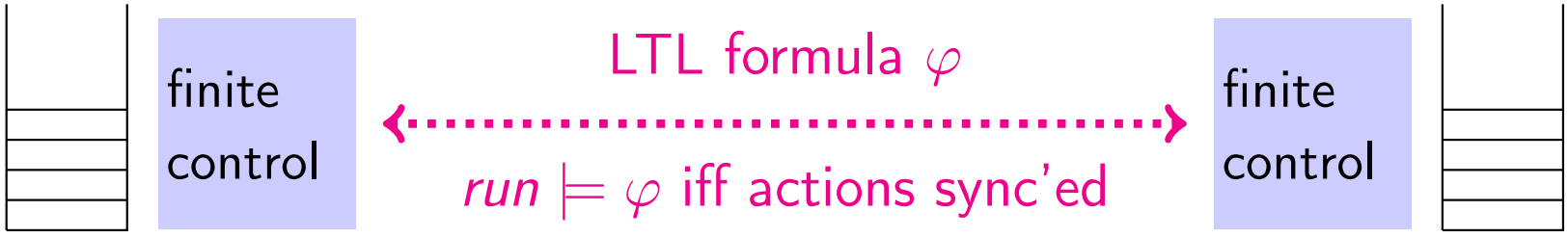
Bounded Ressources and LTL-MC

➪ however extending this result leads directly to:

Theorem:

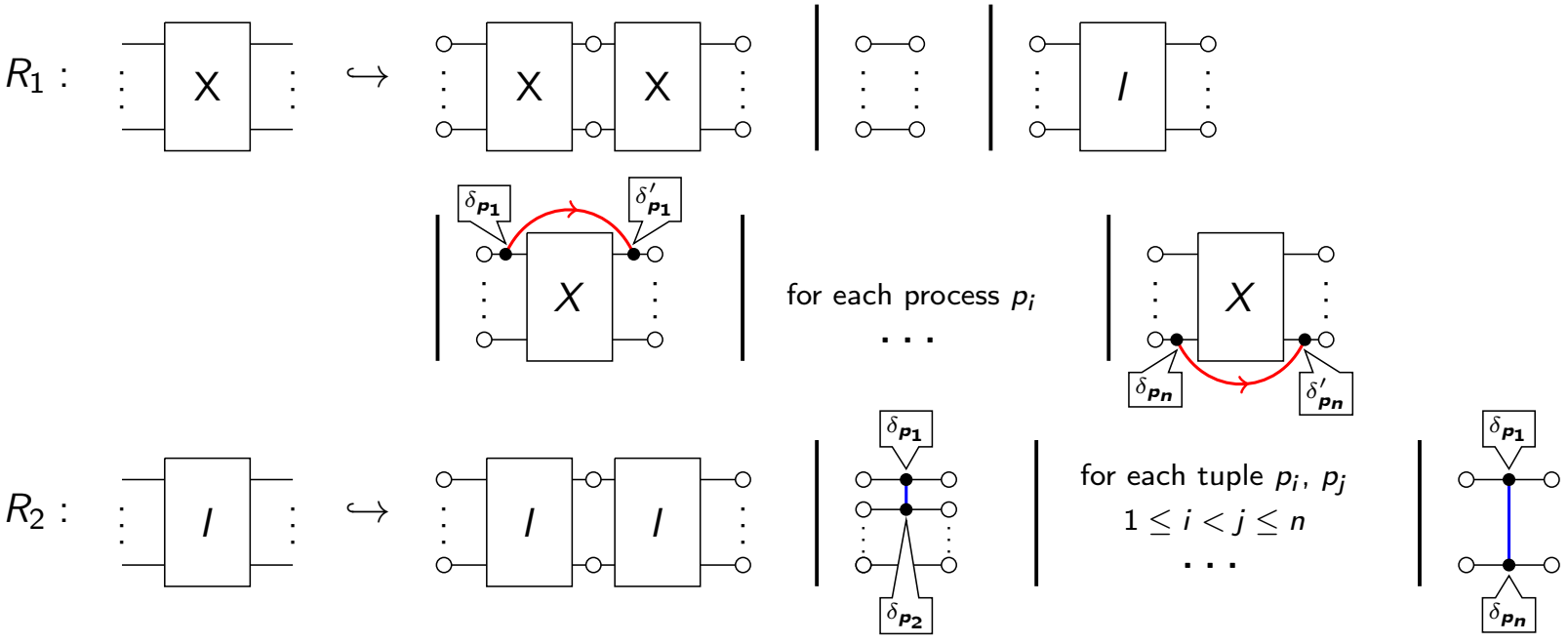
The LTL model checking question is undecidable for RQCPS that are well-queueing, eager and imbalanced. This already holds for a weak fragment of action LTL.

➪ couple independent events by LTL interpreted on interleaving ↯



➪ leave interleaving semantics behind, look at rungraphs !

HRG for 1-stack reducible eager iRCPS



- guess "correct" labelling (transition rule $\delta \in \Delta_p$ f.e. process p)
- rulewidth bounded by $\mathcal{O}(\#\text{procs.})$
- correctness of labelling "local": express by MSO formula φ_{label}
- model check in addition: $\varphi' \equiv \varphi \wedge \varphi_{\text{label}}$

MSO Model Checking RCPS

Theorem:

The MSO model checking question for rungraphs of RCPS that are eager, imbalanced and for which all runs can be reordered such that they are "1-stack reducible" is decidable.

- ➪ cannot decide if for a given RCPS all runs can be reordered into a 1-stack one ↯
- ➪ but covers important practical examples ☺
  (e.g., master-worker protocols etc.)
- ➪ semantic boundedness versus syntactic description ☹

MSO Model Checking RCPS

Theorem:

The MSO model checking question for rungraphs of RCPS that are eager, imbalanced and for which all runs can be reordered such that they are "1-stack reducible" is decidable.

- ➪ cannot decide if for a given RCPS all runs can be reordered into a 1-stack one ↯
- ➪ but covers important practical examples ☺
  (e.g., master-worker protocols etc.)
- ➪ semantic boundedness versus syntactic description ☹

MSO Model Checking RCPS

Theorem:

The MSO model checking question for rungraphs of RCPS that are eager, imbalanced and for which all runs can be reordered such that they are "1-stack reducible" is decidable.

- ➪ cannot decide if for a given RCPS all runs can be reordered into a 1-stack one ↯
- ➪ but covers important practical examples ☺
  (e.g., master-worker protocols etc.)
- ➪ semantic boundedness versus syntactic description ☹

Treewidth Boundedness

Conjecture:

Bounded treewidth of the underlying rungraphs is the most general form of semantic boundedness condition for which MSO model checking is decidable.

➪ can generalize other boundedness conditions
(e.g., bounded phases for RCPS ✓ , lock graphs ✓ , etc.)

? how to derive algorithms that are usable in practice

? how to derive syntactic/decidable restrictions

? what about other graph boundedness measures
(cliquewidth, vertex cover, bounded pathwidth,. . .)

Treewidth Boundedness

Conjecture:

Bounded treewidth of the underlying rungraphs is the most general form of semantic boundedness condition for which MSO model checking is decidable.

- ➪ can generalize other boundedness conditions
  (e.g., bounded phases for RCPS ✓ , lock graphs ✓ , etc.)
- ? how to derive algorithms that are usable in practice
- ? how to derive syntactic/decidable restrictions
- ? what about other graph boundedness measures
  (cliquewidth, vertex cover, bounded pathwidth,. . .)

Treewidth Boundedness

Conjecture:

Bounded treewidth of the underlying rungraphs is the most general form of semantic boundedness condition for which MSO model checking is decidable.

- ➪ can generalize other boundedness conditions
  (e.g., bounded phases for RCPS ✓ , lock graphs ✓ , etc.)
- ? how to derive algorithms that are usable in practice
- ? how to derive syntactic/decidable restrictions
- ? what about other graph boundedness measures
  (cliquewidth, vertex cover, bounded pathwidth,. . .)

Treewidth Boundedness

Conjecture:

Bounded treewidth of the underlying rungraphs is the most general form of semantic boundedness condition for which MSO model checking is decidable.

- ➪ can generalize other boundedness conditions
  (e.g., bounded phases for RCPS ✓ , lock graphs ✓ , etc.)
- ? how to derive algorithms that are usable in practice
- ? how to derive syntactic/decidable restrictions
- ? what about other graph boundedness measures
  (cliquewidth, vertex cover, bounded pathwidth,. . .)

Related Works

- [Madhusudan et al.] LTL/MSO model checking for MSC-graphs
- [Parlato et al.'08] use tree-decomposition to derive new model checking algorithms (bounded phases...)
- MSO-definable temporal logics [Kuske/Gastin'10] and verification on partial orders of RCPS [Bollig et al.'11]
- partial-order versions of LTL (no consensus)
- novel boundedness conditions for asynchronous systems [LaTorre/Napoli'11] [Bouajjani/Emmi'12] etc.
- `Autowrite` tool [Durand/Courcelle]
- graph-rewriting based transition systems

Final Summary

- leave behind interleaving semantics
- focus rungraphs
- describe generation of rungraphs by graph grammars
- do model checking on rungraphs
- known connection HRG-treewidth-MSO allows to derive decidability of model checking