11. Petri Nets

Oscar Nierstrasz

J. L. Peterson, *Petri Nets Theory and the Modelling of Systems*, Prentice Hall, 1983.

Roadmap

- Definition:
  - places, transitions, inputs, outputs
  - firing enabled transitions
- Modelling:
  - concurrency and synchronization
- Properties of nets:
  - liveness, boundedness
- Implementing Petri net models:
  - centralized and decentralized schemes

Roadmap

> **Definition:**
>
> —**places, transitions, inputs, outputs**
>
> —**firing enabled transitions**

> Modelling:
>
> —concurrency and synchronization

> Properties of nets:
>
> —liveness, boundedness

> Implementing Petri net models:
>
> —centralized and decentralized schemes

Petri nets: a definition

A Petri net $C = \langle P,T,I,O \rangle$ consists of:

1. A finite set P of *places*
2. A finite set T of *transitions*
3. An *input function* $I: T \to Nat^P$ (maps to bags of places)
4. An *output function* $O: T \to Nat^P$

A marking of C is a mapping $m: P \to Nat$

Example:

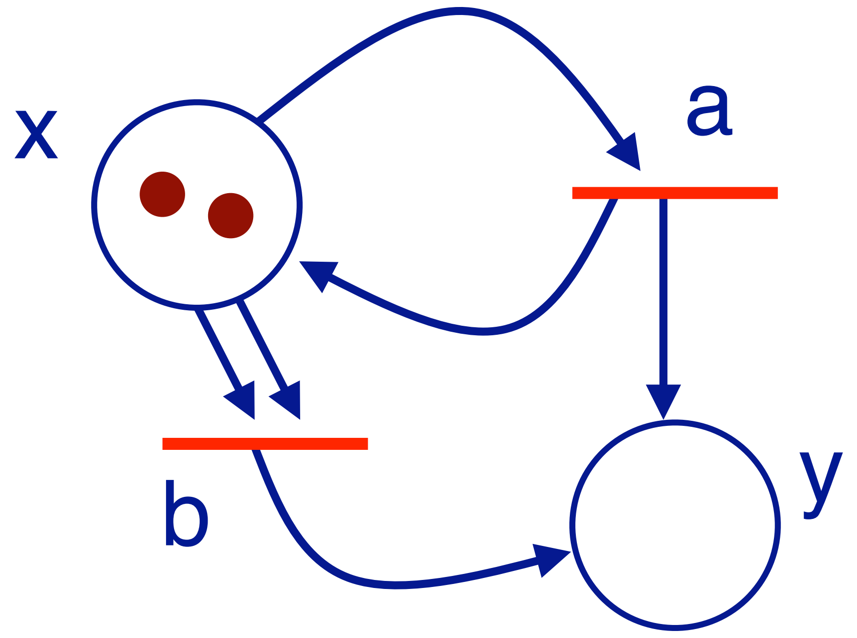
$P = \{ x, y \}$

$T = \{ a, b \}$

$I(a) = \{ x \}$, $I(b) = \{ x, x \}$

$O(a) = \{ x, y \}$, $O(b) = \{ y \}$

$m = \{ x, x \}$



Petri nets (or “place/transition nets”) were invented Carl Adam Petri. They offer a very intuitive graphical formalism for modeling concurrent processes. Formal Petri nets are directed bigraphs, graphs with two kinds of vertices — *places* and *transitions* — together with a *marking* — a function that indicates how many tokens are currently in every given place. Transitions are *enabled* if they have at least one token in every input place. *Firing* an enabled transition leads to a new marking by removing one token from every input and adding one to every output place.

In the diagram we see the same net represented both graphically, and using multisets.

The following classic survey (and subsequent book) offer an excellent introduction:

James L. Peterson. *Petri Nets*. In ACM Computing Surveys 9(3) p. 223—252, September 1977.

http://dx.doi.org/10.1145/356698.356702

http://scgresources.unibe.ch/Literature/CP/Pete77aSurvey.pdf

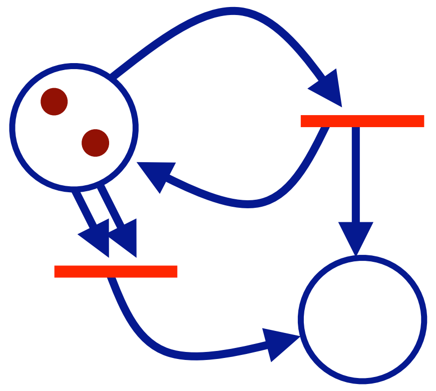
See also:

https://en.wikipedia.org/wiki/Petri_net

Firing transitions

To fire a transition t:

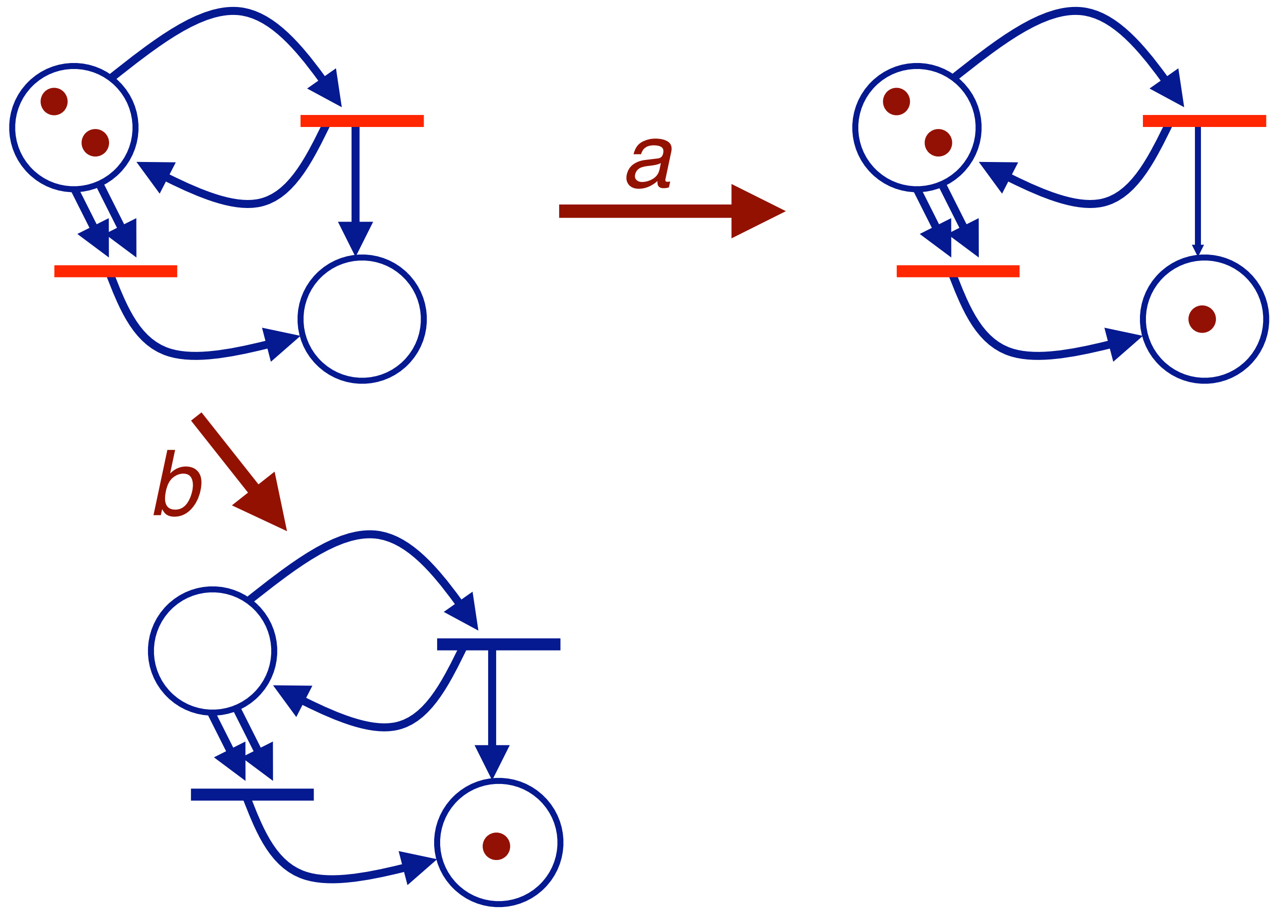
1. t must be enabled: $m \geq I(t)$
2. *consume* inputs and *generate* output: $m' = m - I(t) + O(t)$



Firing transitions

To fire a transition t:

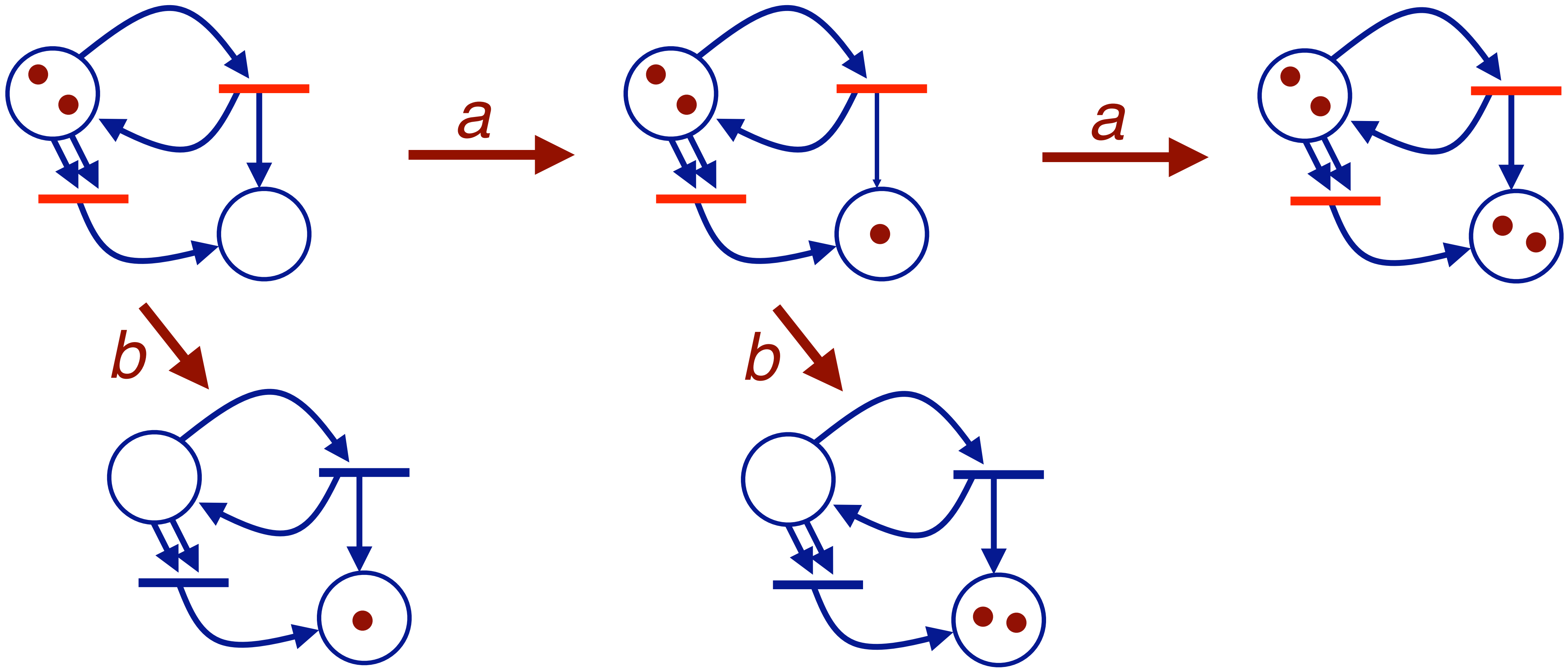
1. t must be enabled: m ≥ I(t)
2. *consume* inputs and *generate* output: m′= m - I(t) + O(t)



Firing transitions

To fire a transition t:

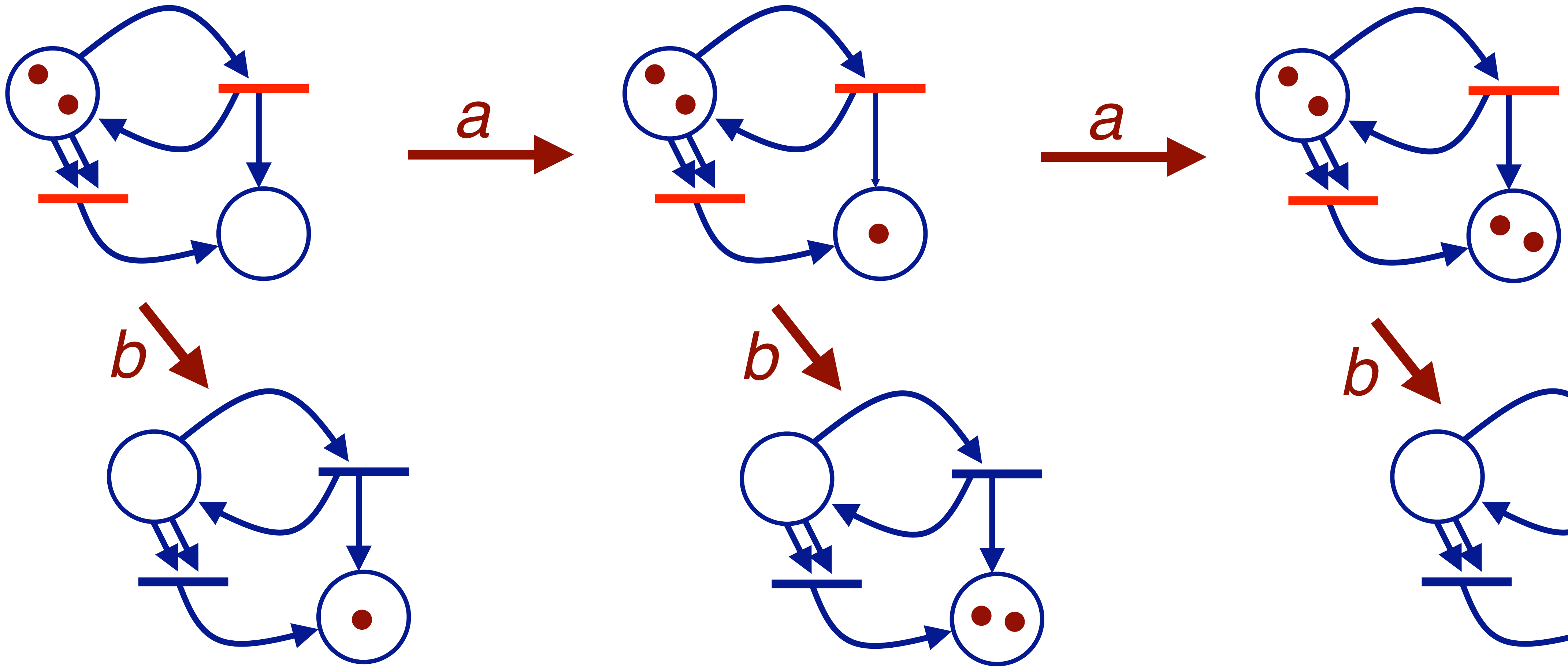
1. t must be enabled: $m \geq I(t)$
2. *consume* inputs and *generate* output: $m' = m - I(t) + O(t)$



Firing transitions

To fire a transition t:

1. t must be enabled: m ≥ I(t)
2. *consume* inputs and *generate* output: m′= m - I(t) + O(t)



This diagram shows how the net of the previous slide can lead to various possible sequences of transitions being fired. Note that the "*language*" of a Petri net consists of the set of possible firing sequences. In this case, the language can be expressed as a regular expression.

Exercise: what is the (regular) language of this net?

Roadmap

> Definition:
  - places, transitions, inputs, outputs
  - firing enabled transitions
> **Modelling:**
  - **concurrency and synchronization**
> Properties of nets:
  - liveness, boundedness
> Implementing Petri net models:
  - centralized and decentralized schemes

Modelling with Petri nets

Petri nets are good for modelling:

> concurrency
> synchronization

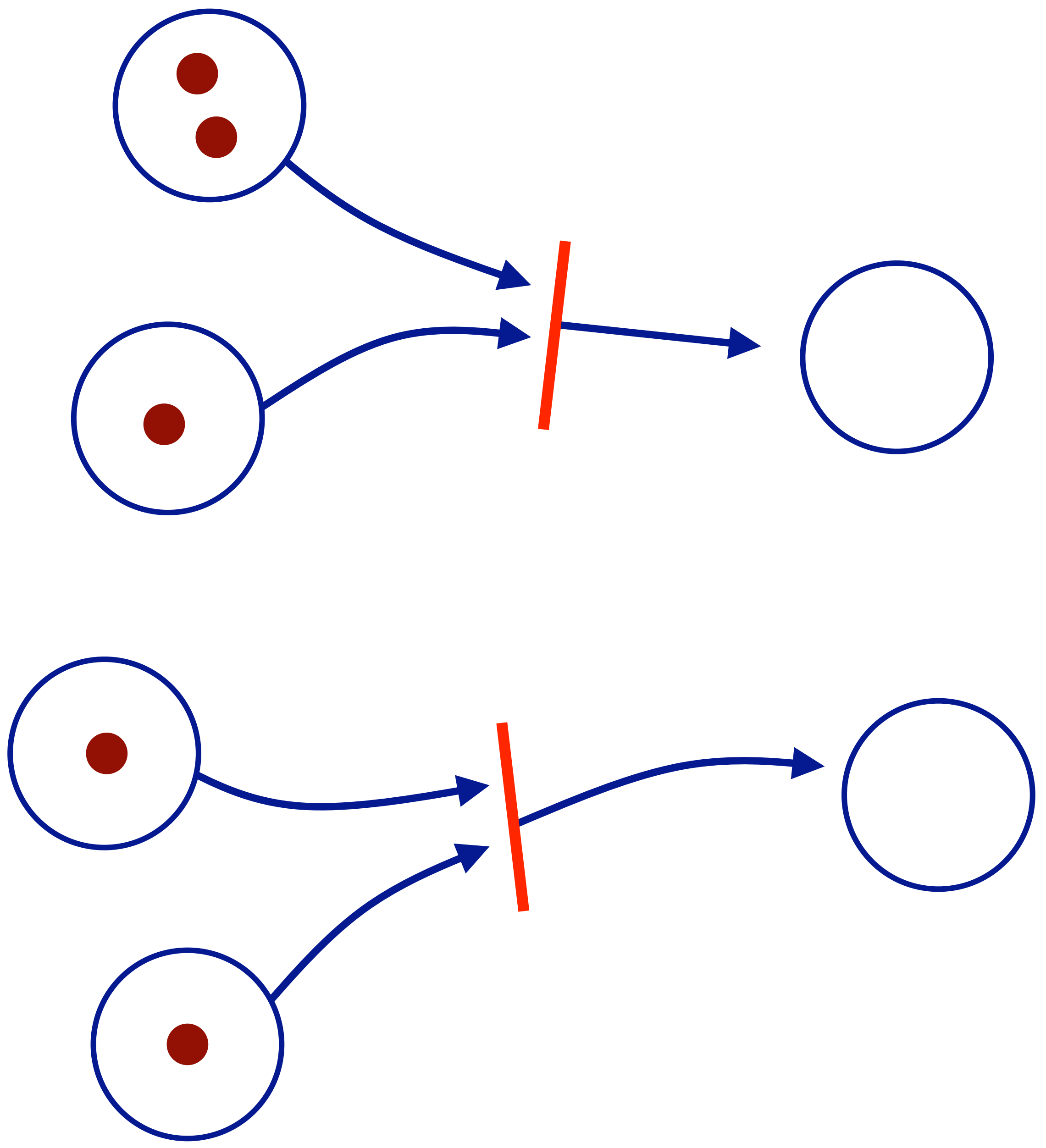
Tokens can represent:

> resource availability
> jobs to perform
> flow of control
> synchronization conditions ...

Petri nets can be used to model a wide range of concurrency problems. Transitions can represent competing processes, and places can represent resources, with tokens (markings) indicating the availability of a resource. But a process may also correspond to a subnet, with places representing the state of a process. Tokens can then represent control flow, or data flow, or synchronization conditions.

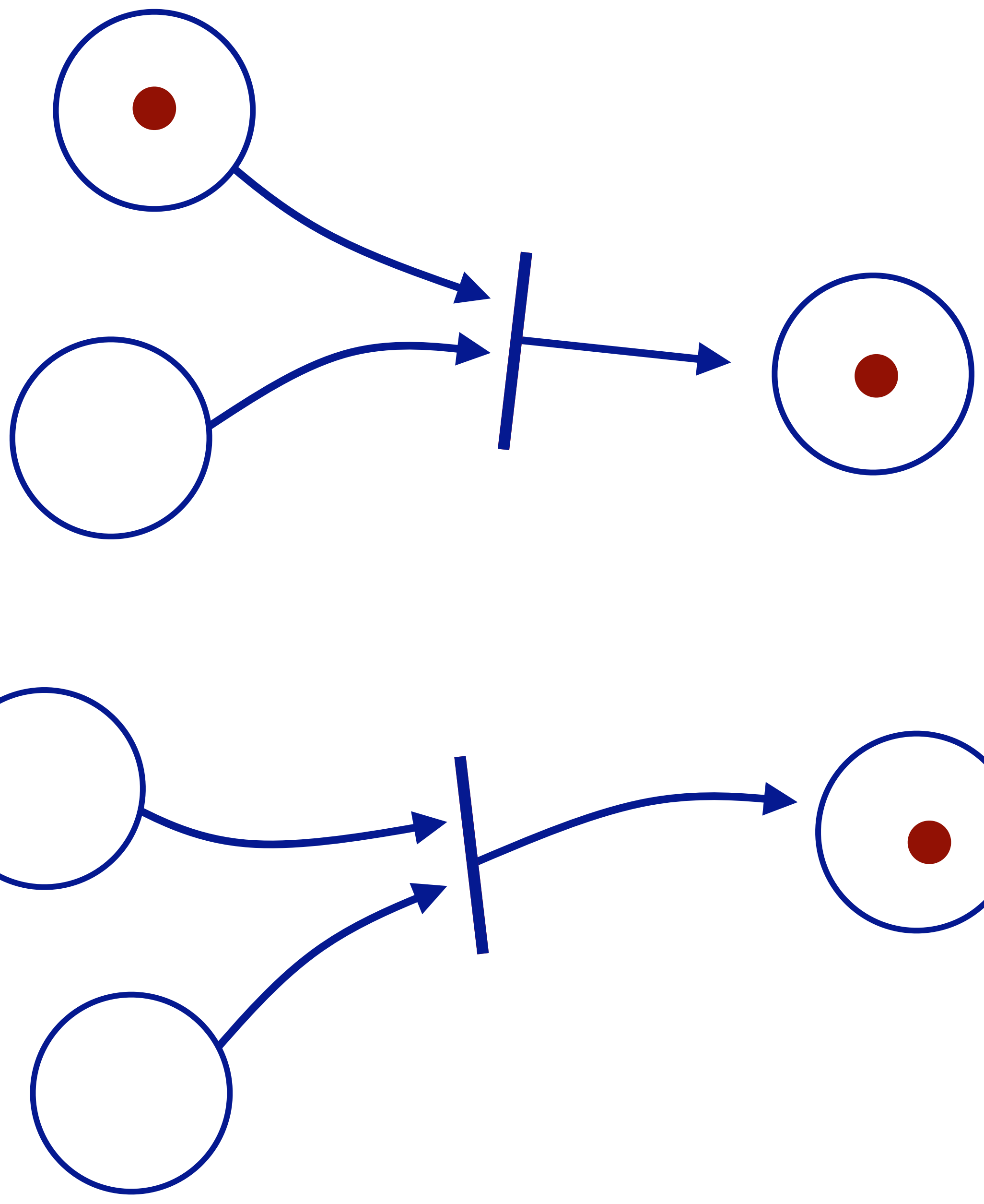
Concurrency

Independent inputs permit "concurrent" firing of transitions



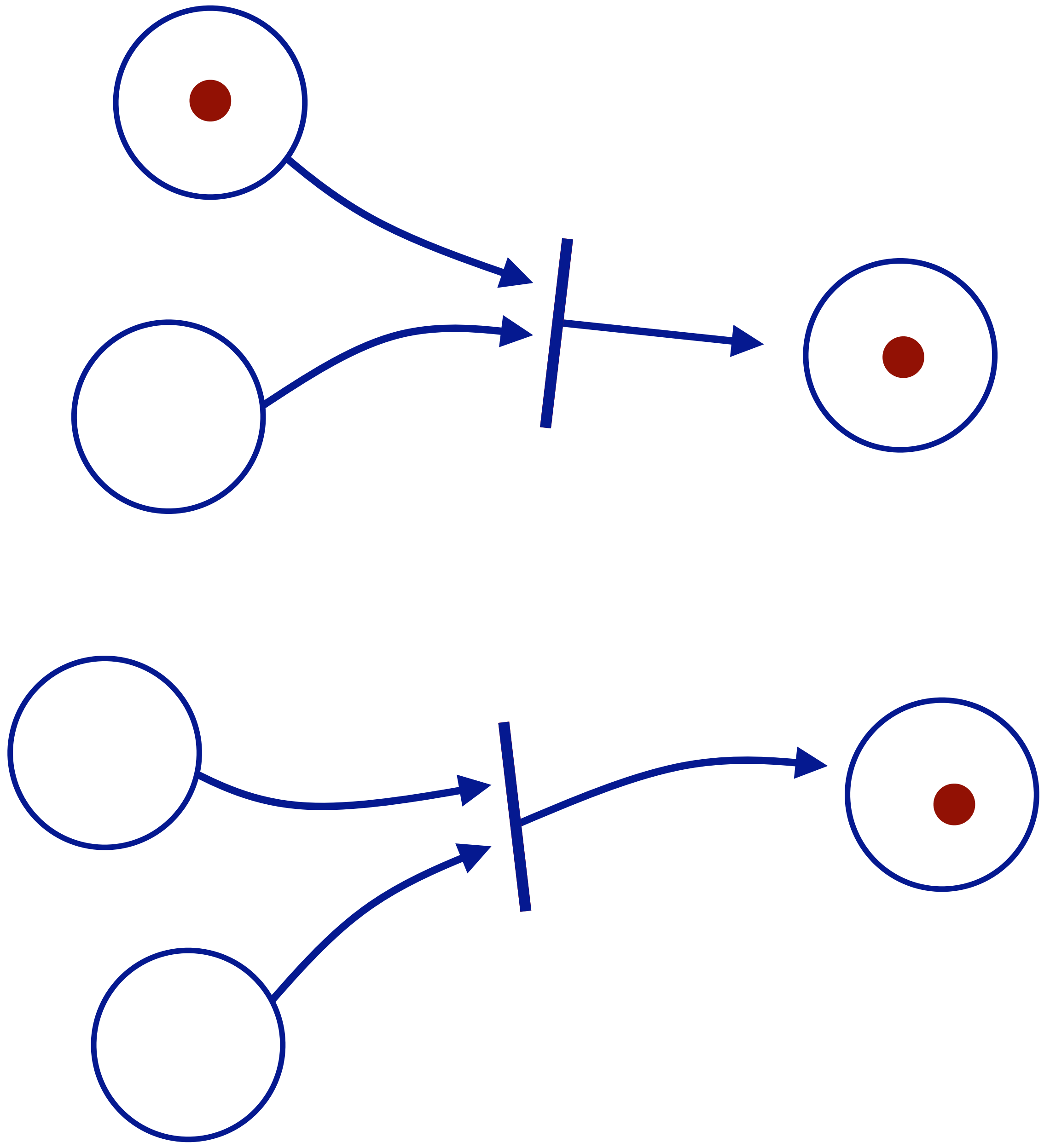
Concurrency

Independent inputs permit "concurrent" firing of transitions



Concurrency

Independent inputs permit "concurrent" firing of transitions

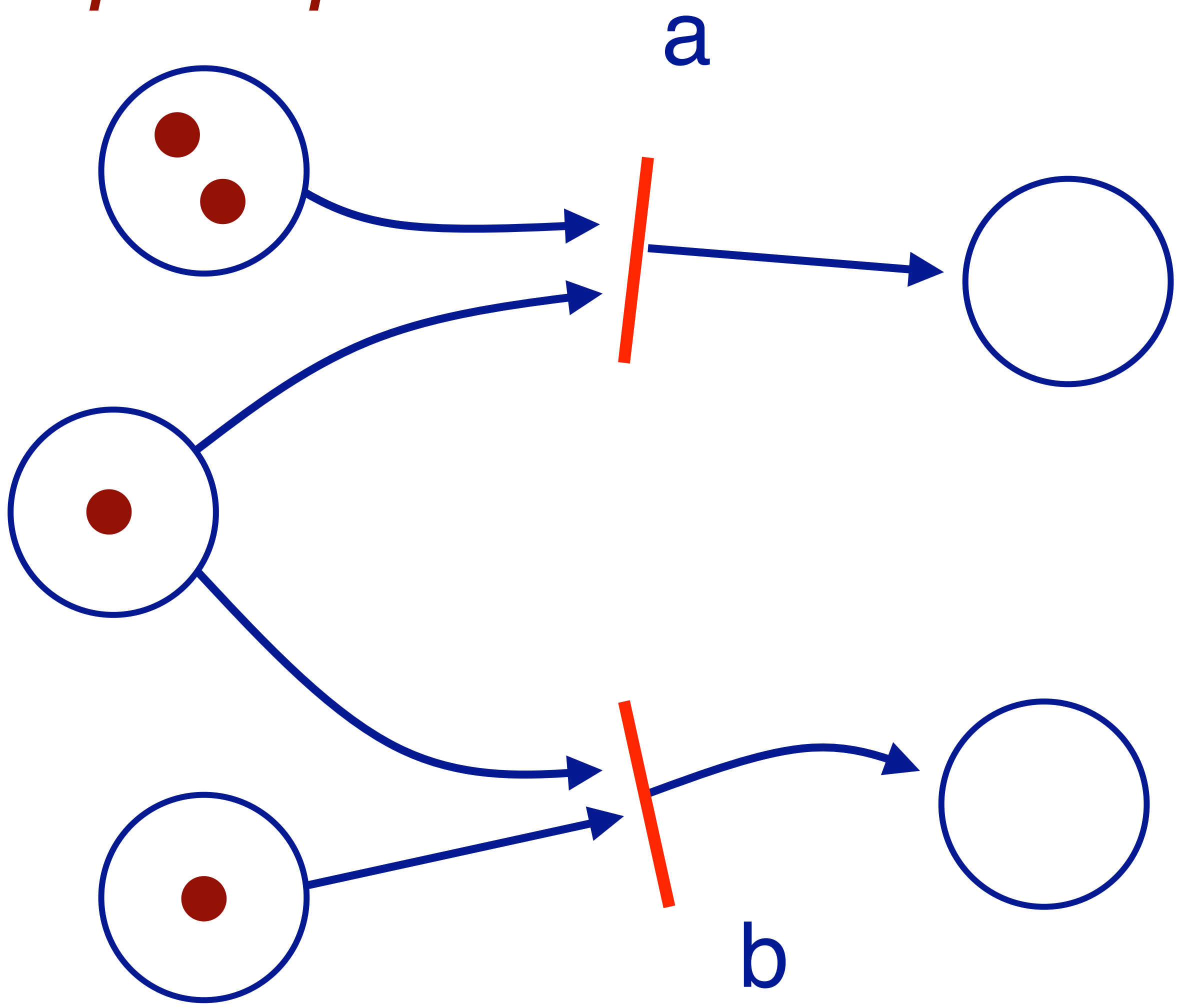


In this example, two independent subnets can fire concurrently, since they share no resources (no common places).

Note that Petri nets have no notion of simultaneous firing of transitions. Instead "concurrent" transitions may fire in an interleaving way.

Conflict

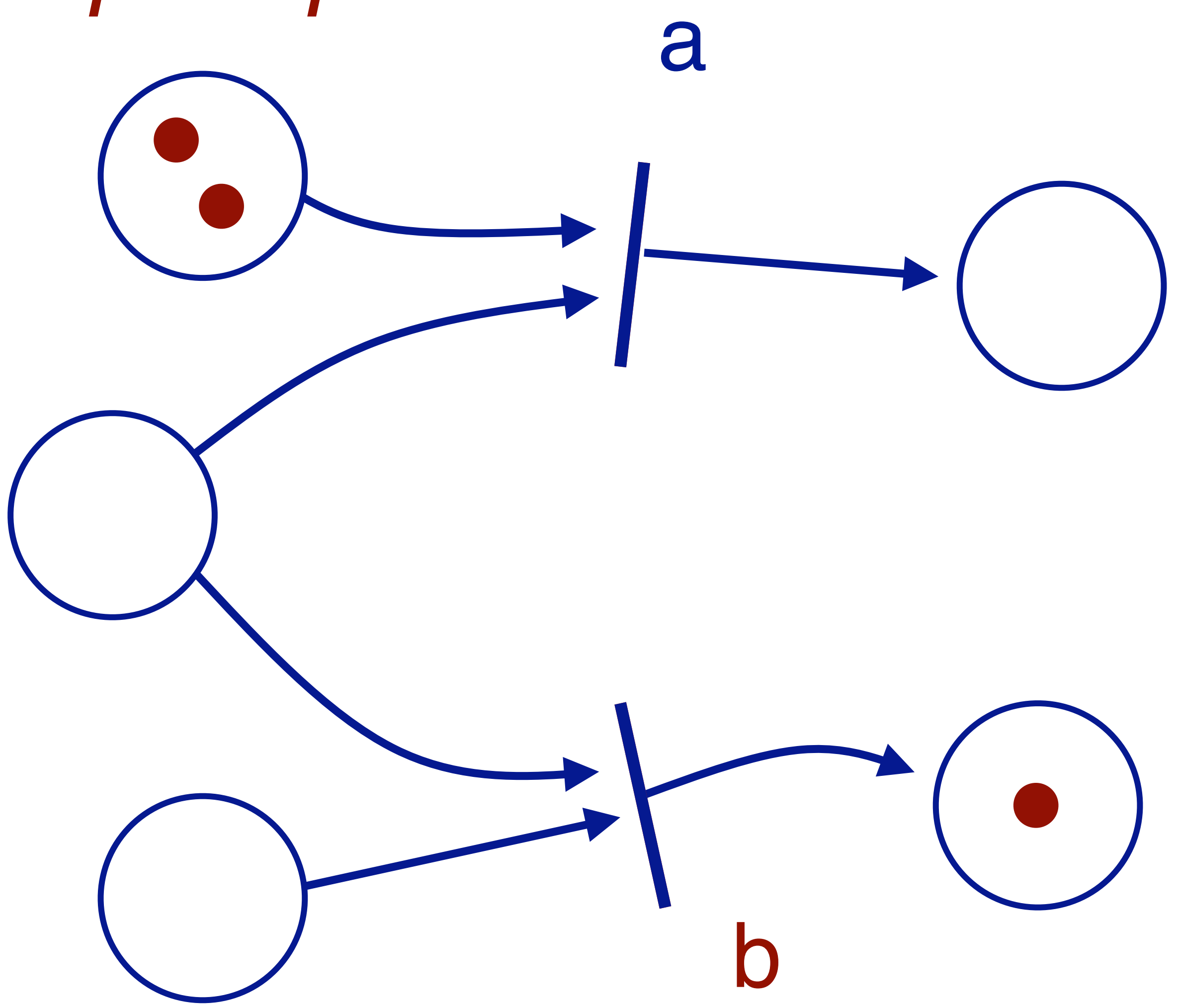
Overlapping inputs put transitions in conflict



Only one of a or b may fire

Conflict

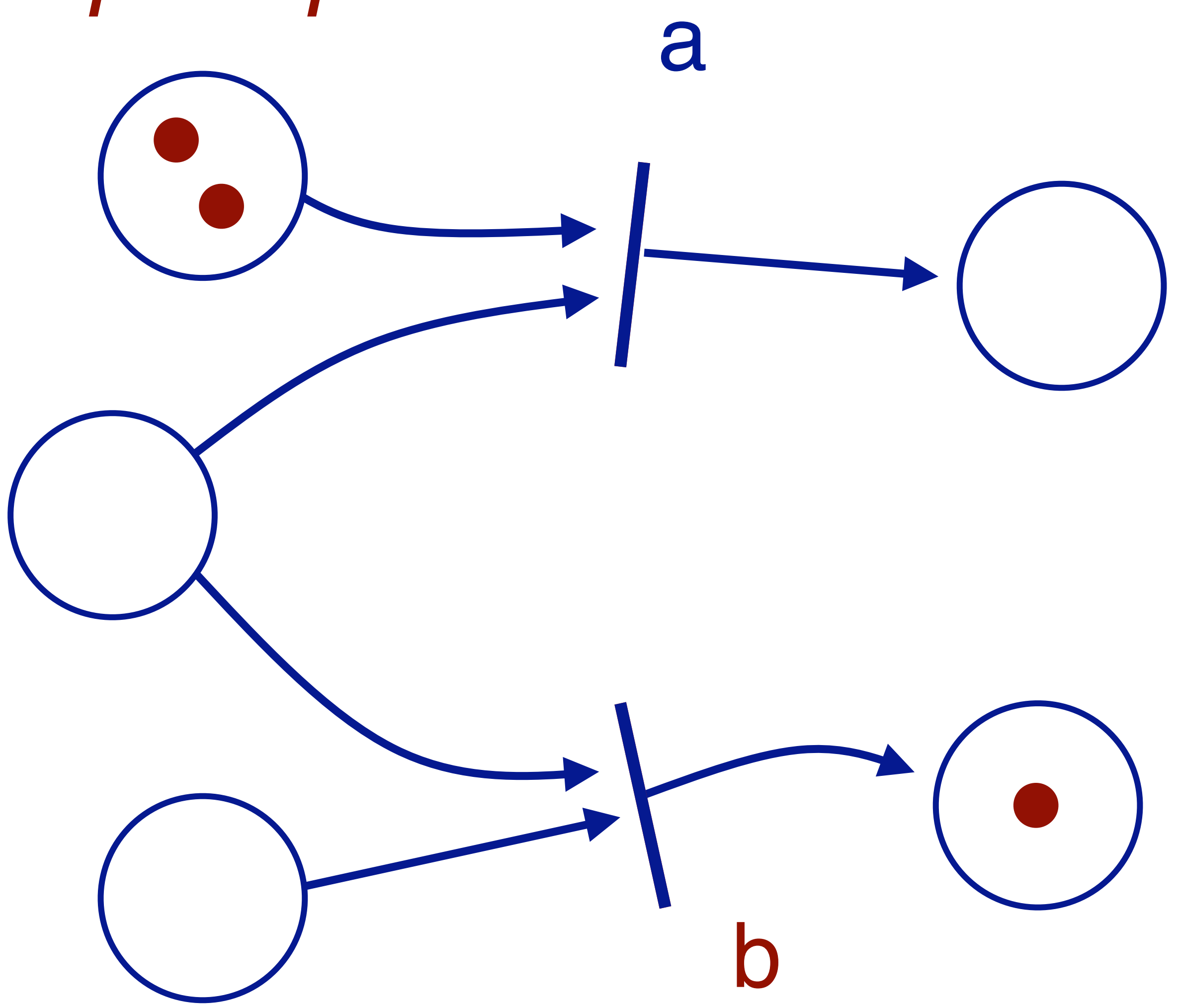
Overlapping inputs put transitions in conflict



Only one of a or b may fire

Conflict

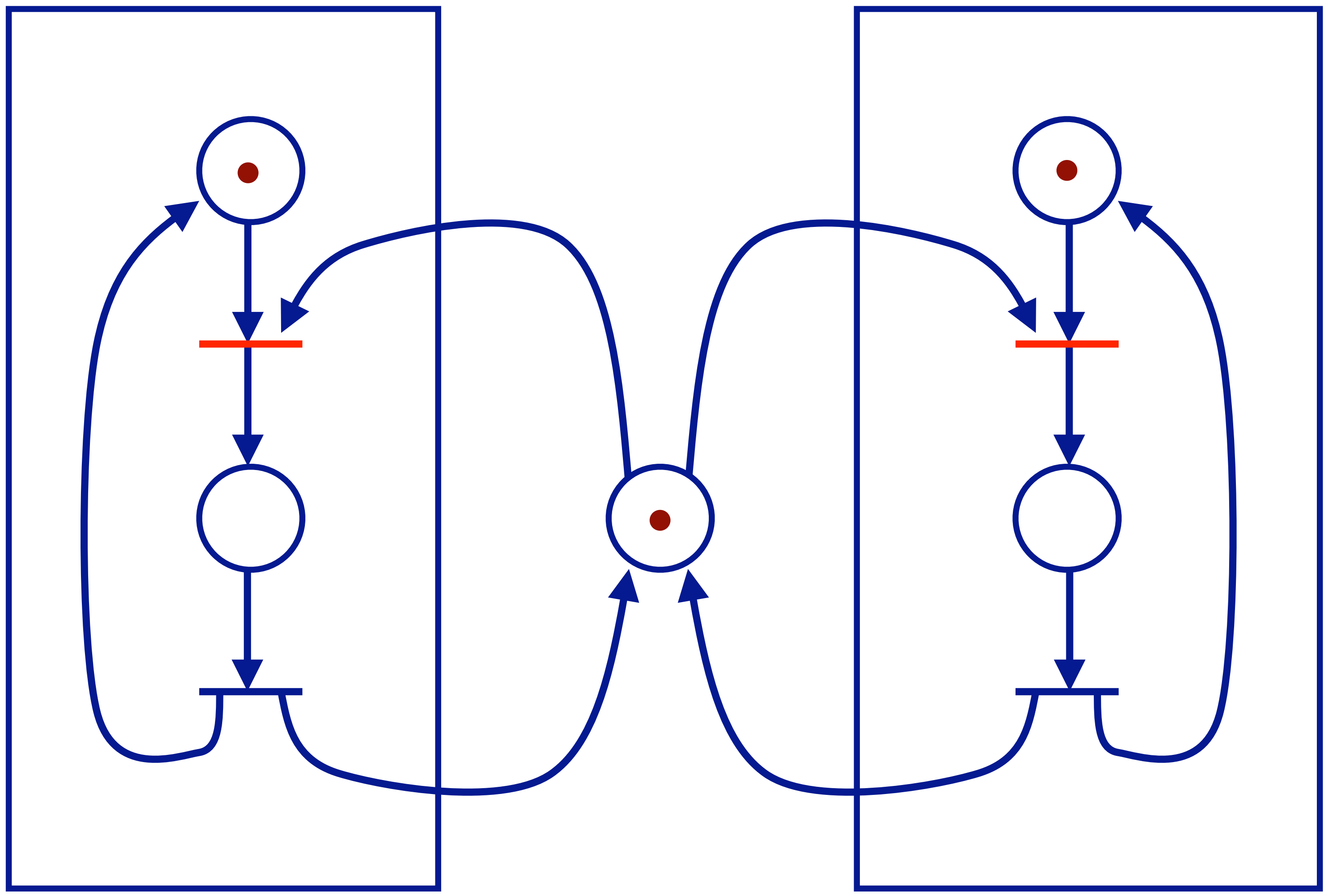
Overlapping inputs put transitions in conflict



Only one of a or b may fire

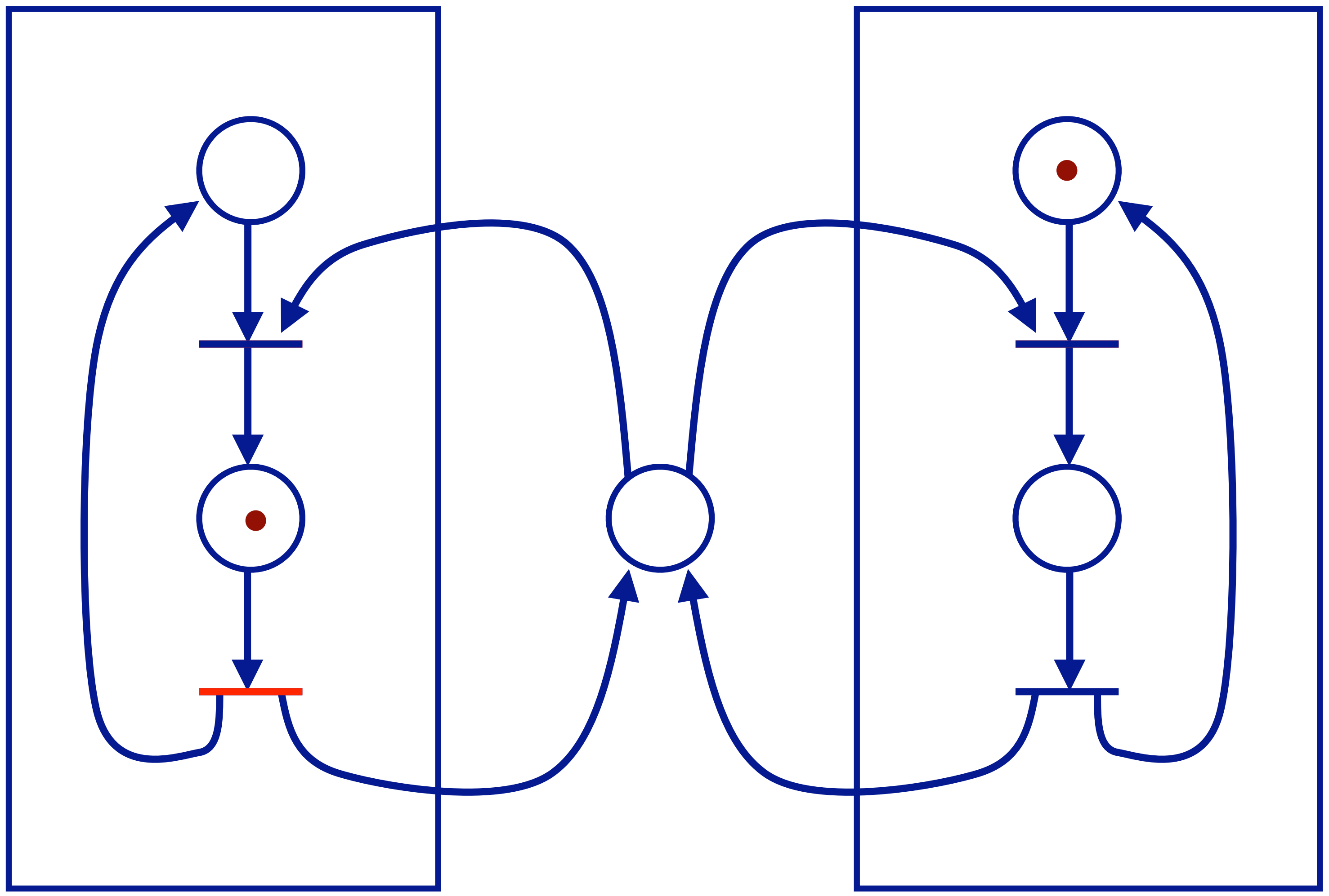
Mutual Exclusion

The two subnets are forced to synchronize



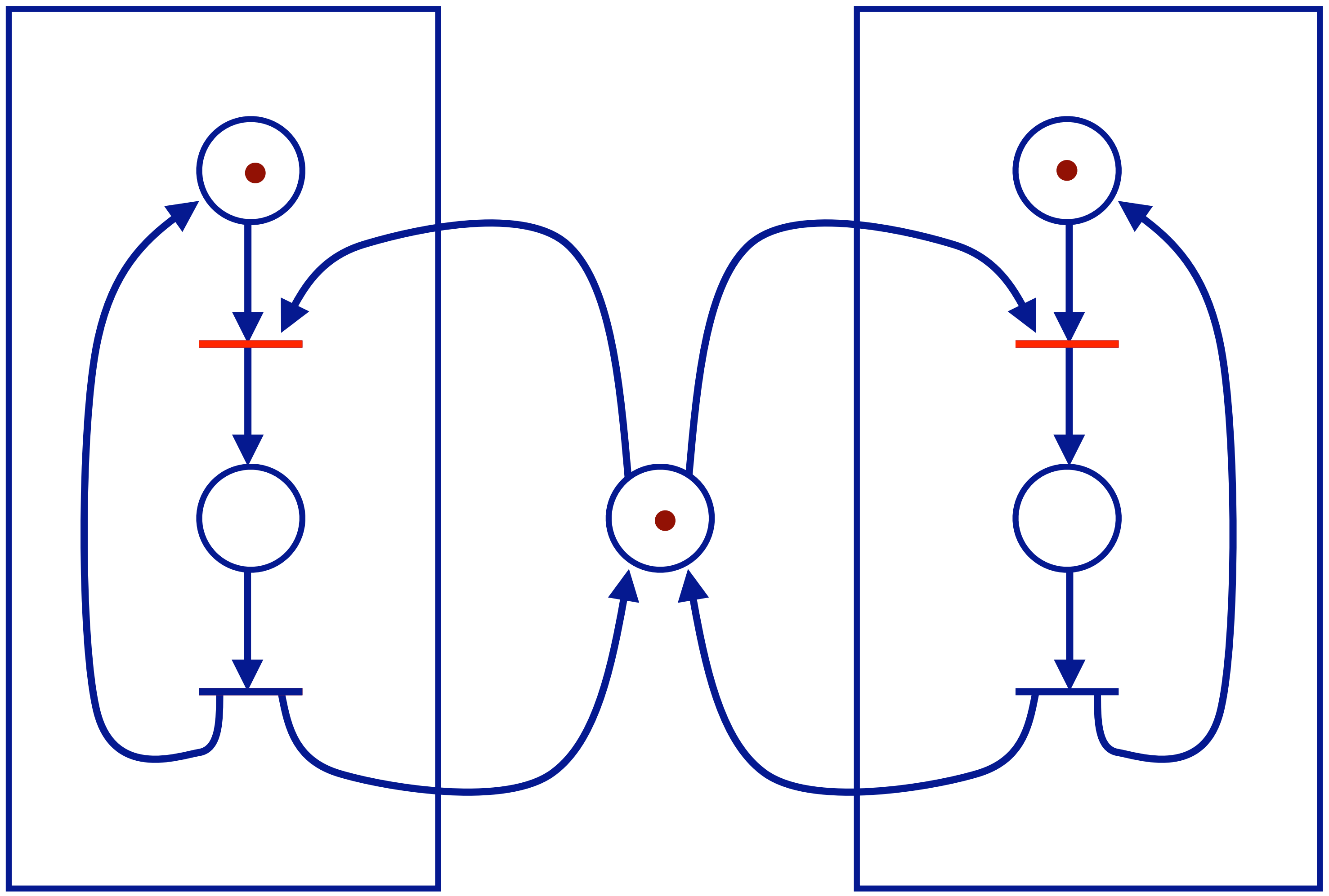
Mutual Exclusion

The two subnets are forced to synchronize



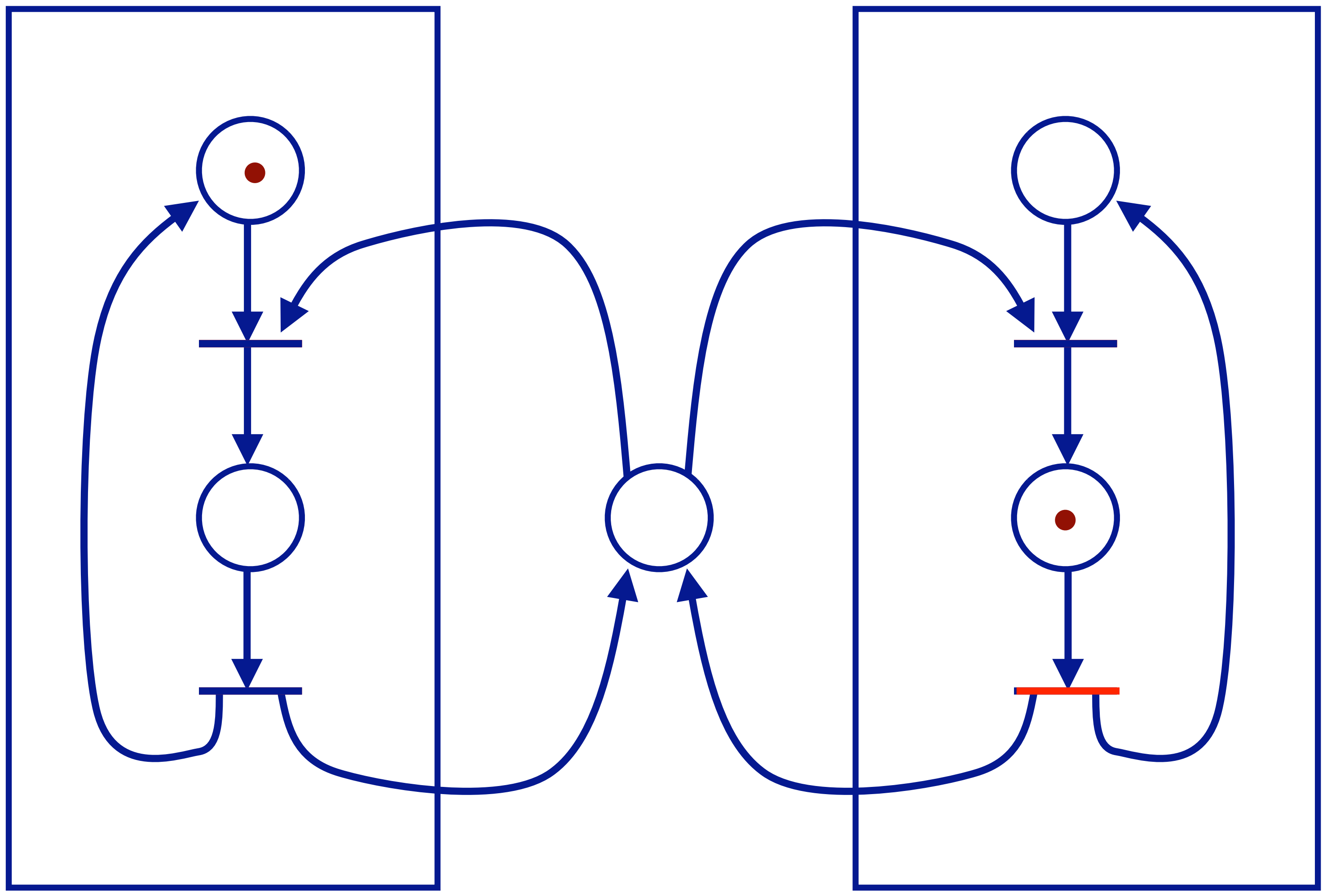
Mutual Exclusion

The two subnets are forced to synchronize



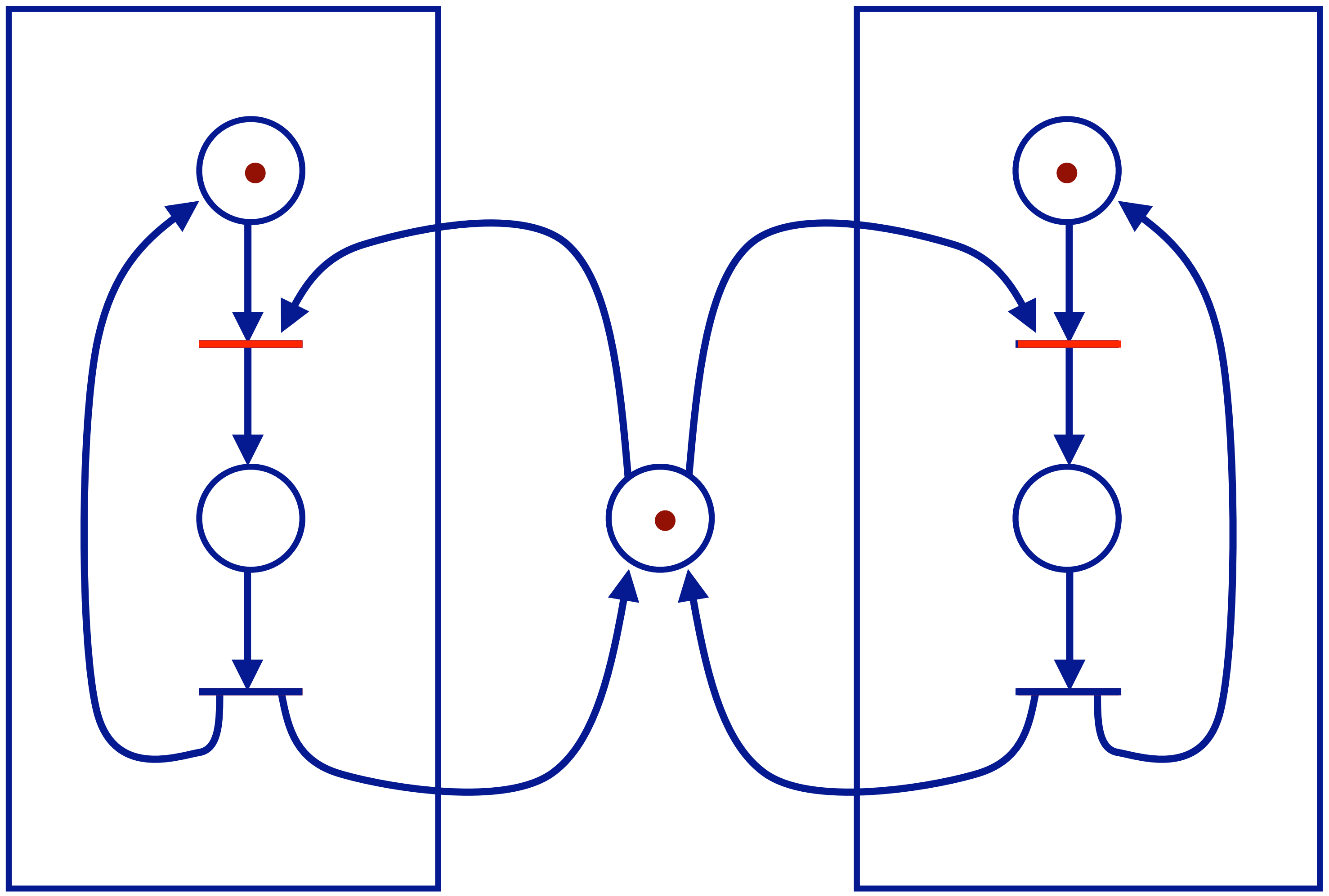
Mutual Exclusion

The two subnets are forced to synchronize



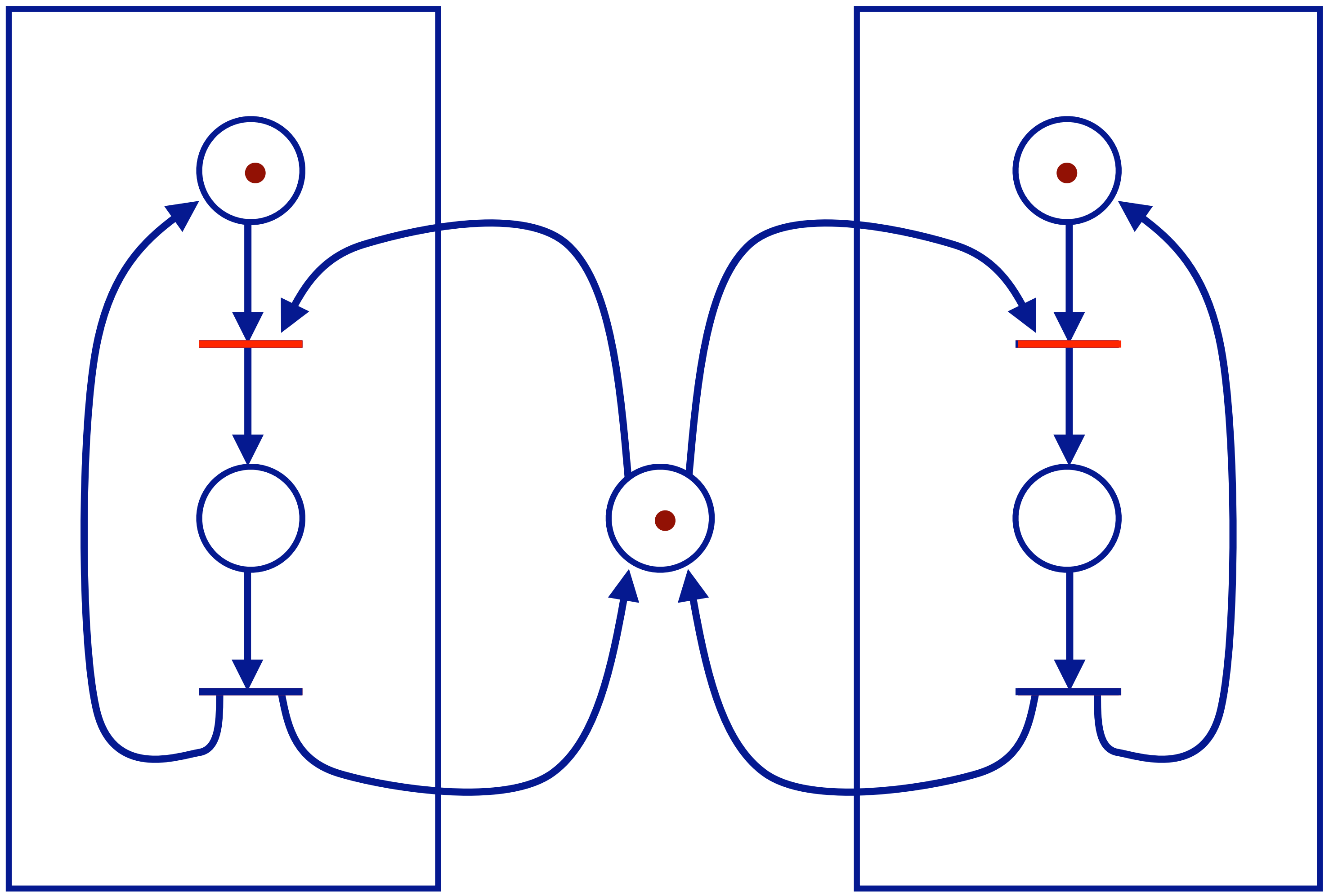
Mutual Exclusion

The two subnets are forced to synchronize



Mutual Exclusion

The two subnets are forced to synchronize

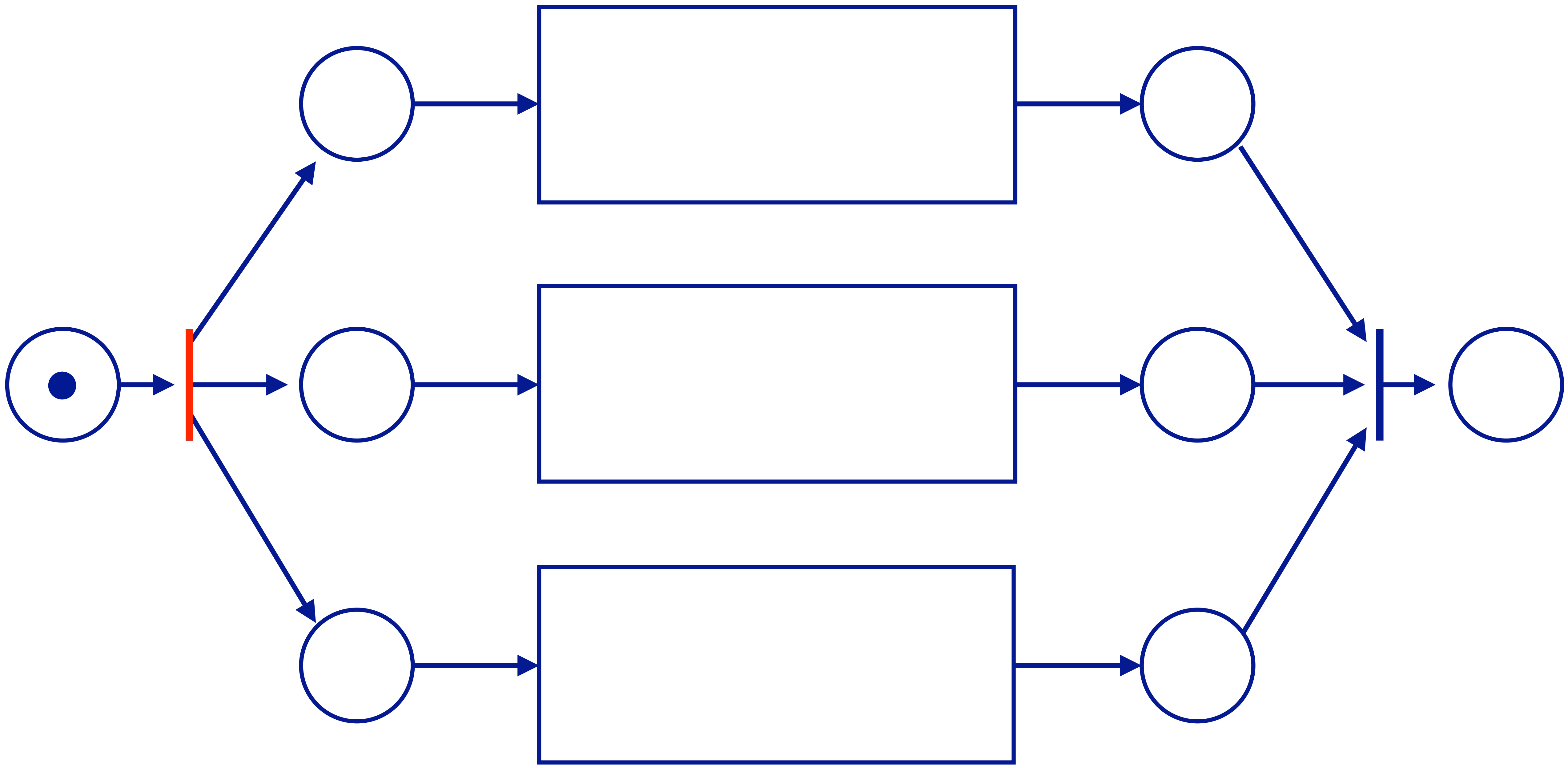


In this diagram we have two subnets that represent two concurrent processes. Each subnet has two transitions — one to enter a "critical section", and one to leave it. The tokens within the subnets represent flow of control. The place connecting the subnets represents a mutual exclusion condition (or semaphore, or lock). A token is present if the resource is available, and is absent if there is already a process in the critical section.

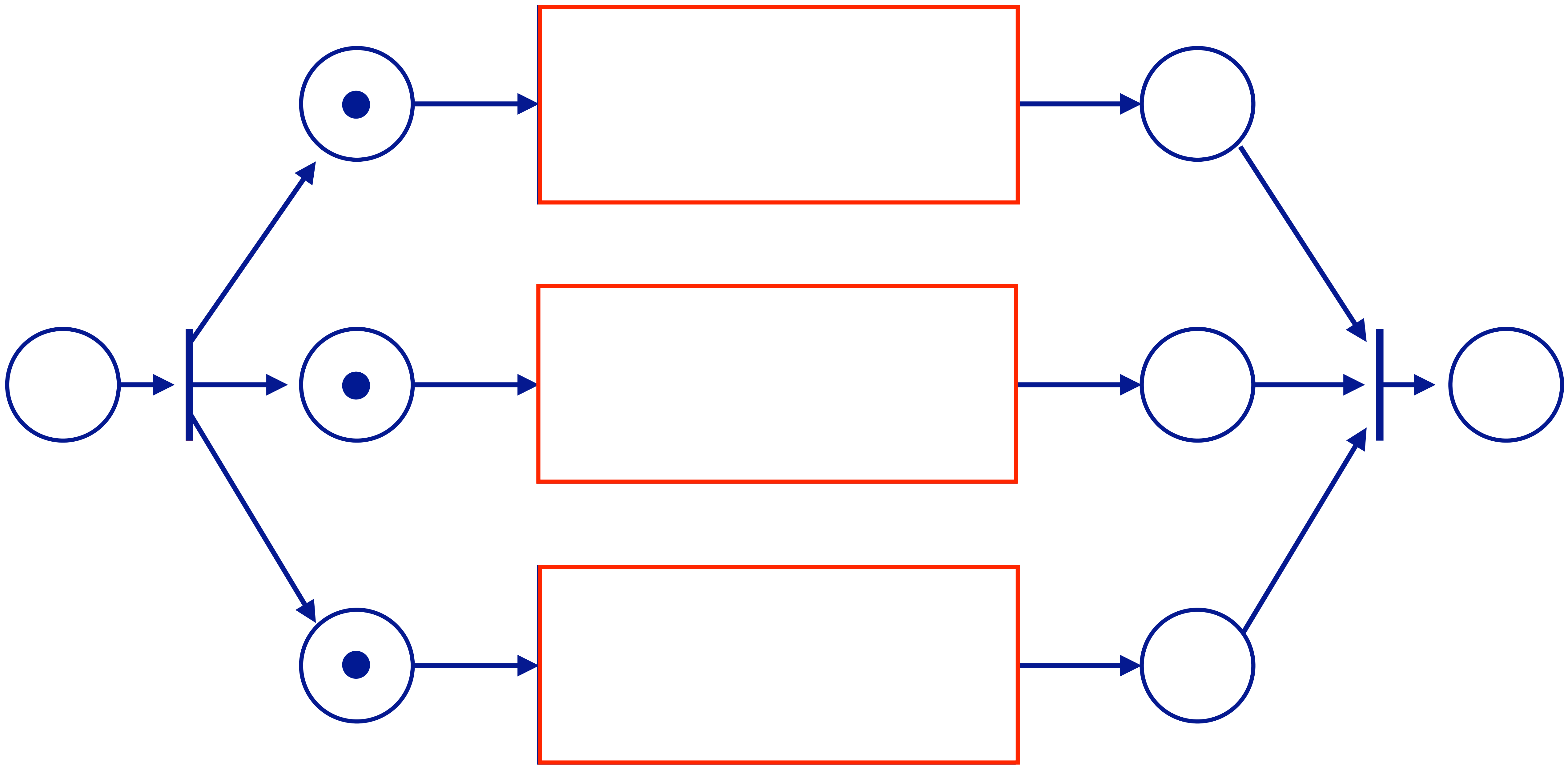
Note how places and tokens represent very different things in this model.

NB: The rectangles are there just to indicate subnets; they are not part of the Petri net formalism.

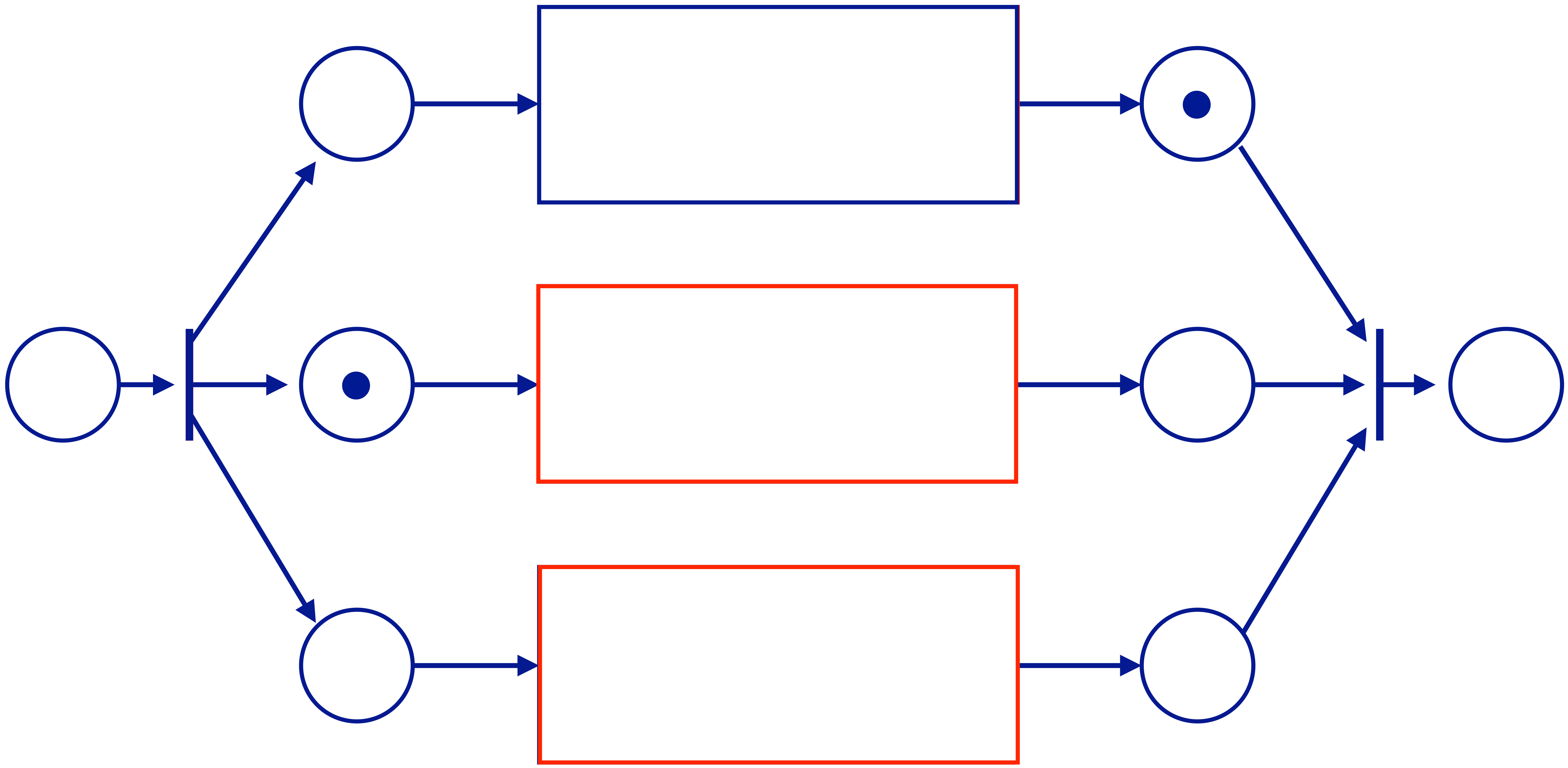
Fork and Join



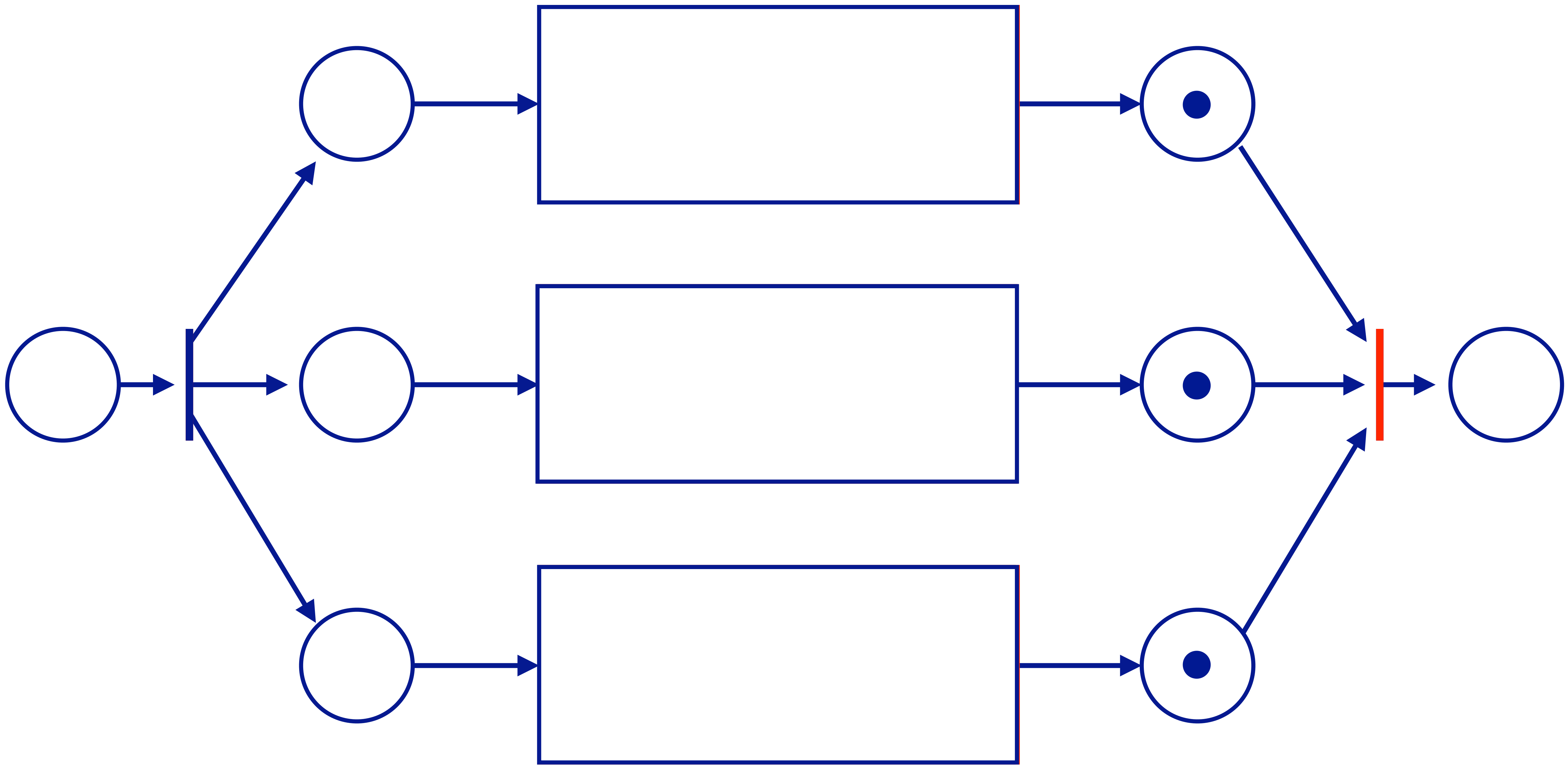
Fork and Join



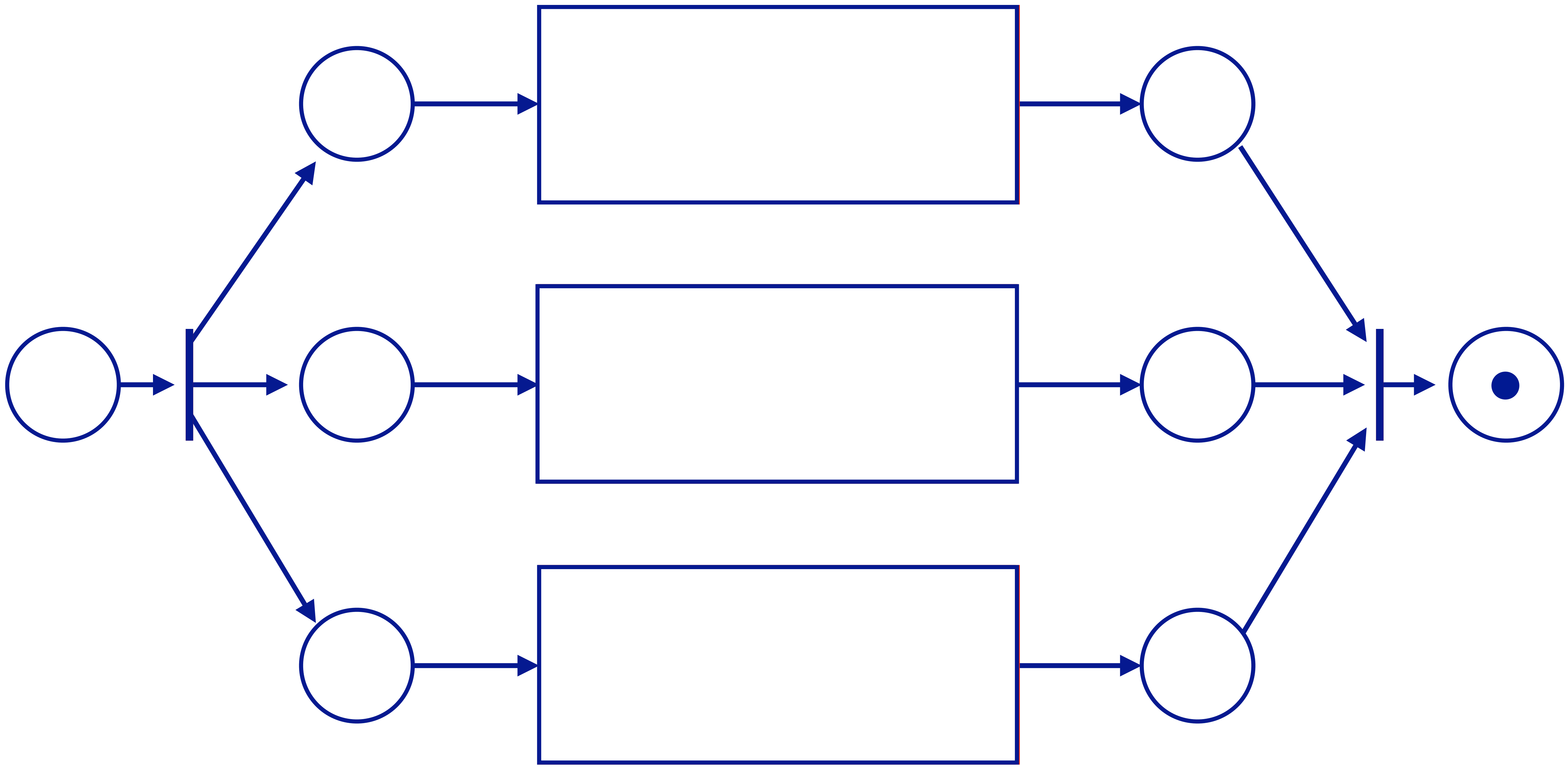
Fork and Join



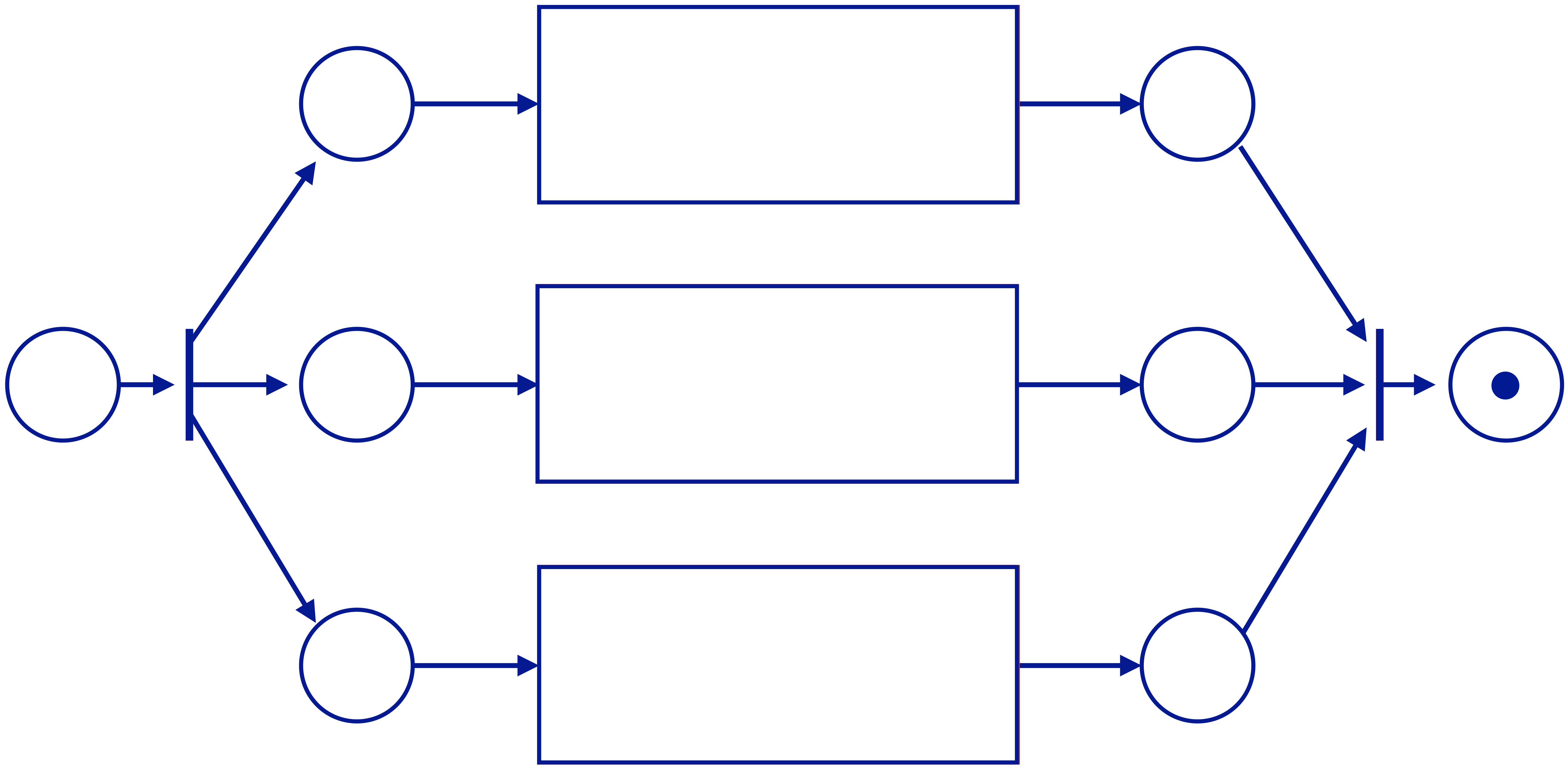
Fork and Join



Fork and Join



Fork and Join

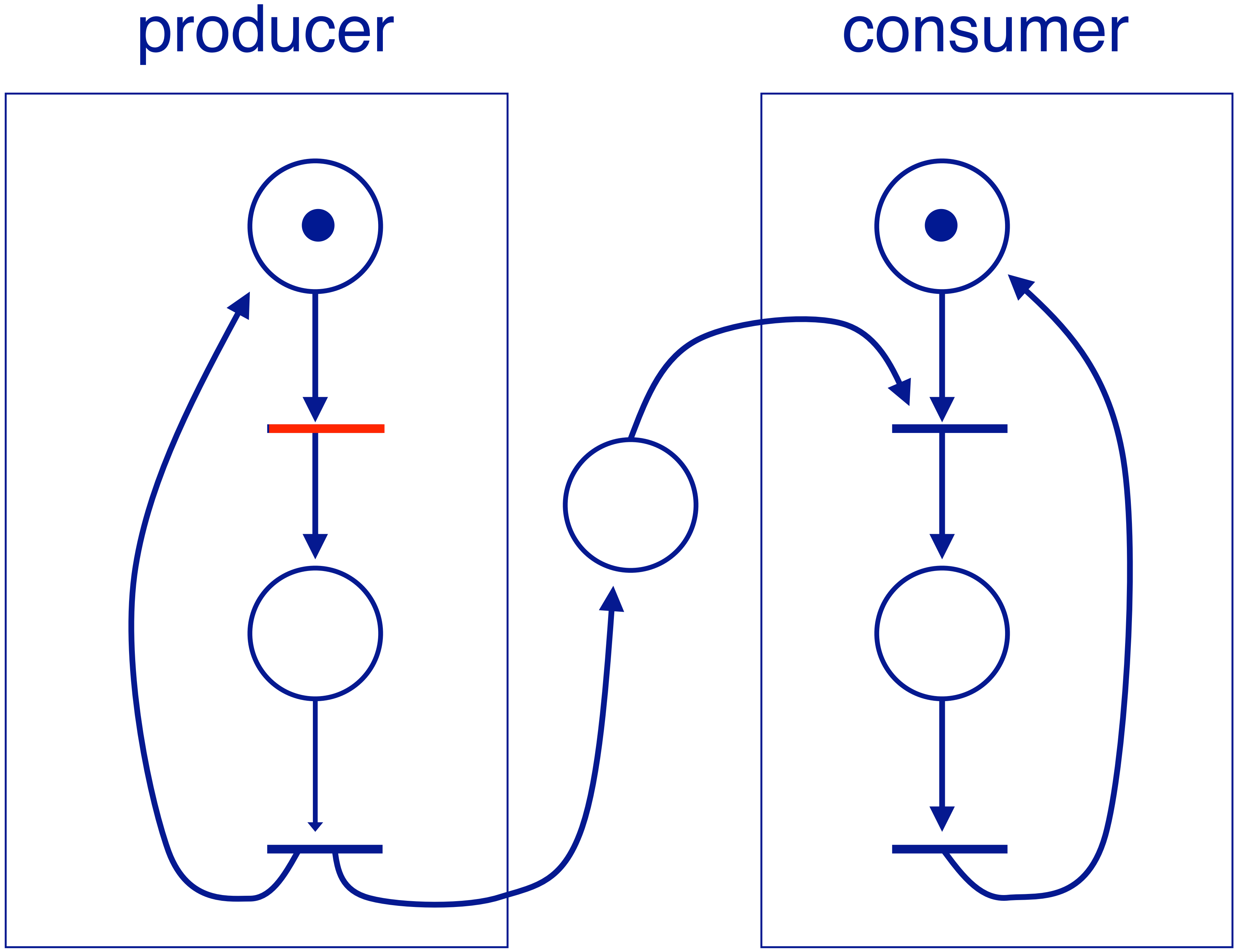


Here the rectangles again represent subnets, but we do not show what is in them.

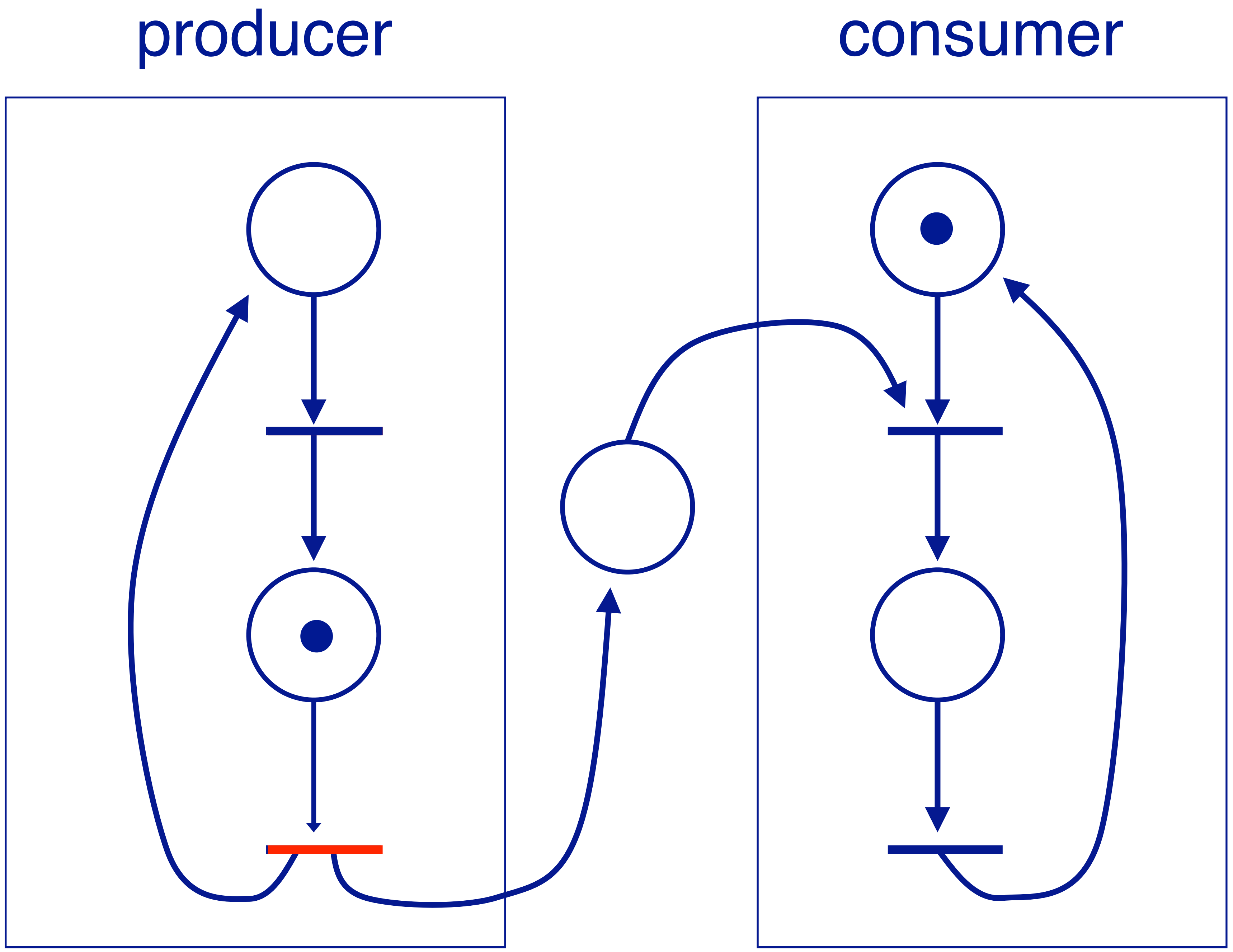
Note how the first transition ("fork") spawns several threads of control, each thread being represented by a token. The final transition ("join") waits for all threads to complete, and then fires, replacing the multiple threads by a single one.

What assumptions need to hold over the subnets for this to work?

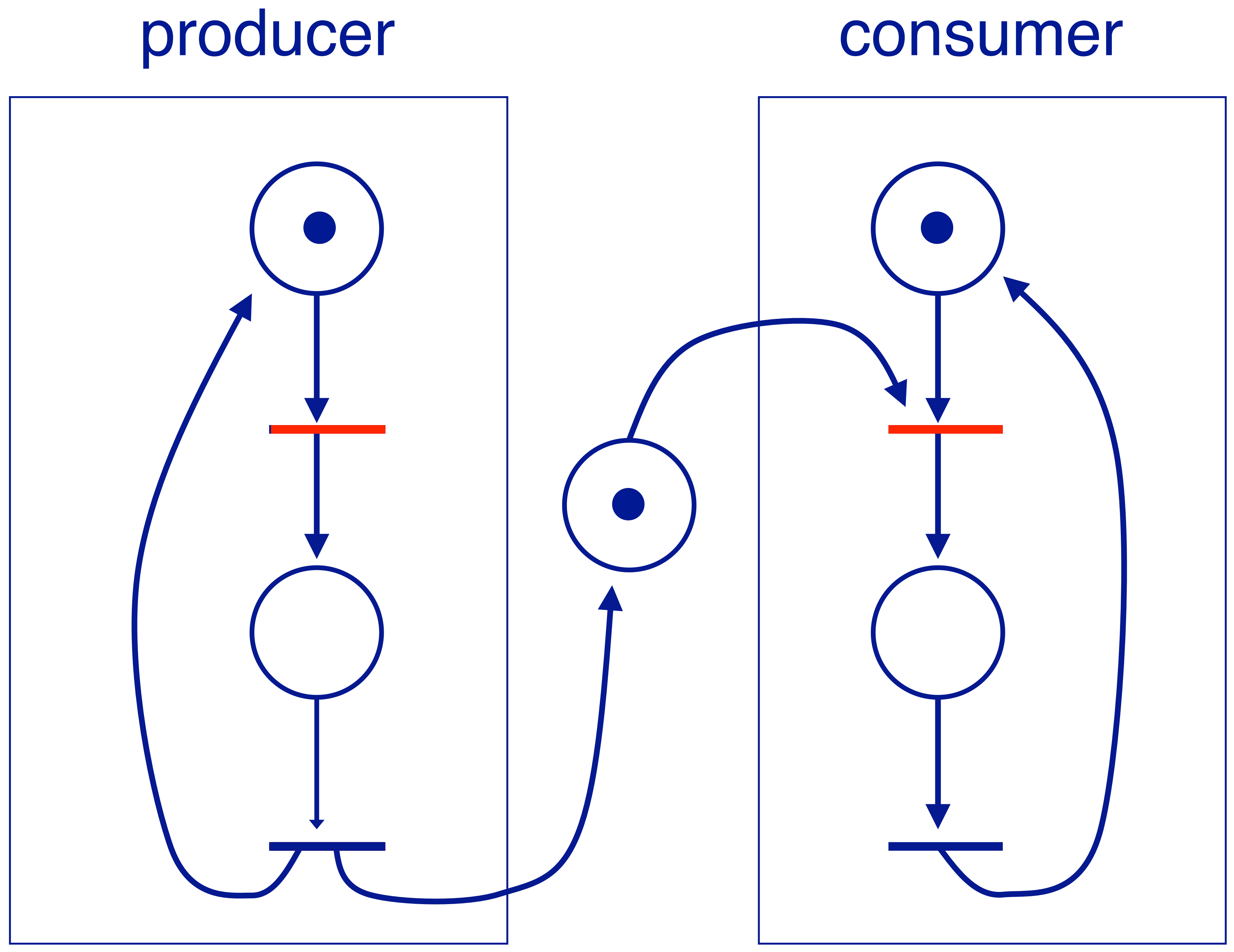
Producers and Consumers



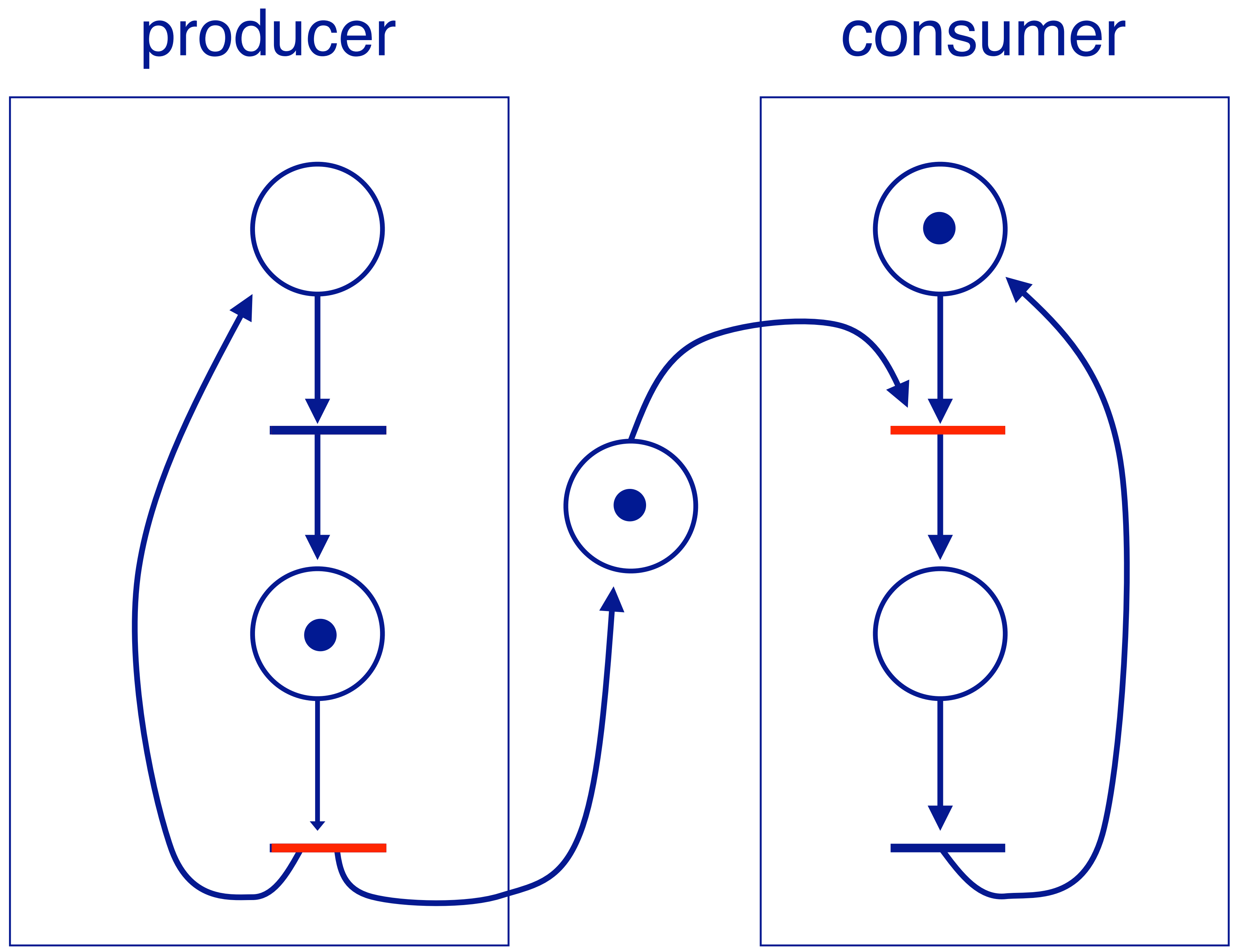
Producers and Consumers



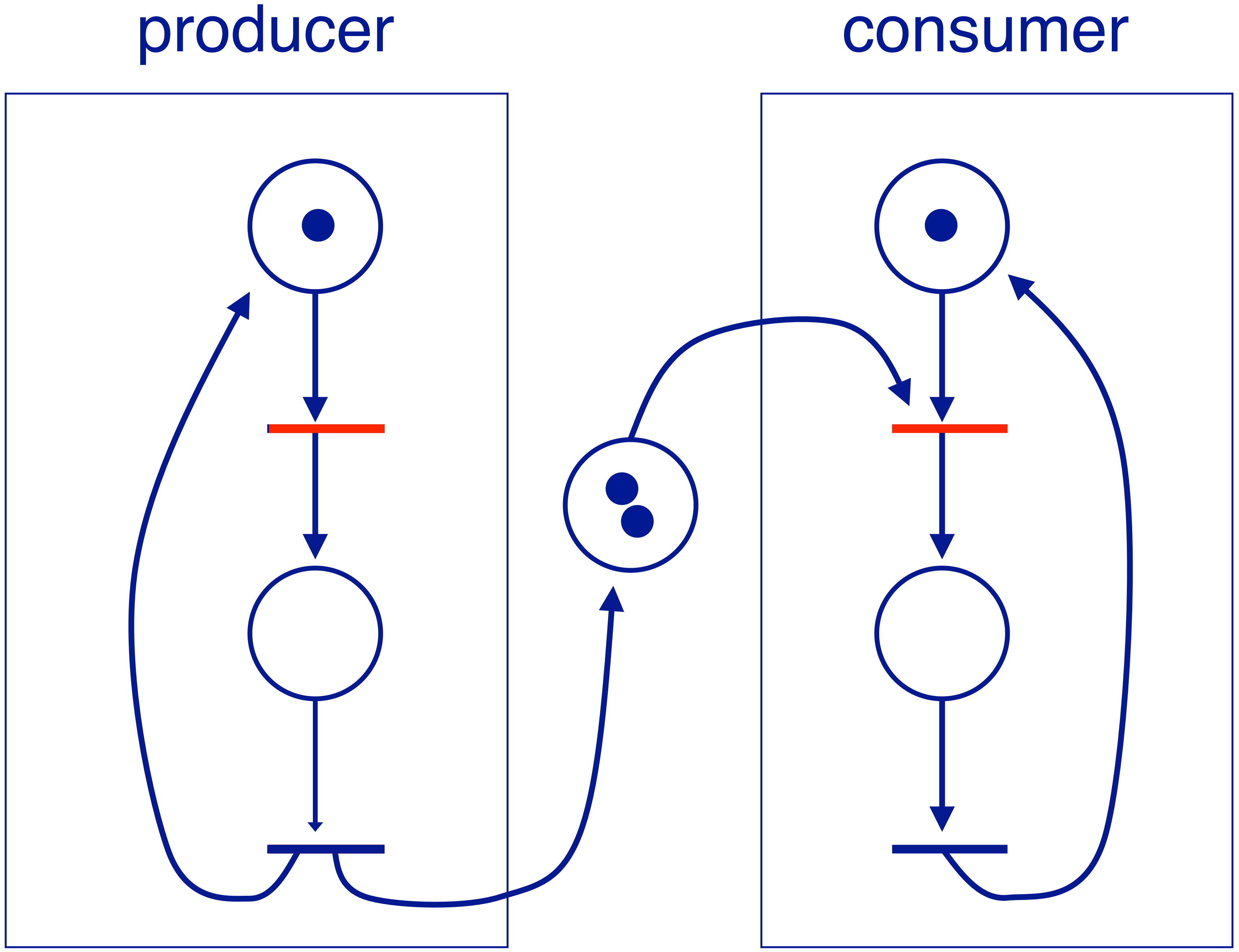
Producers and Consumers



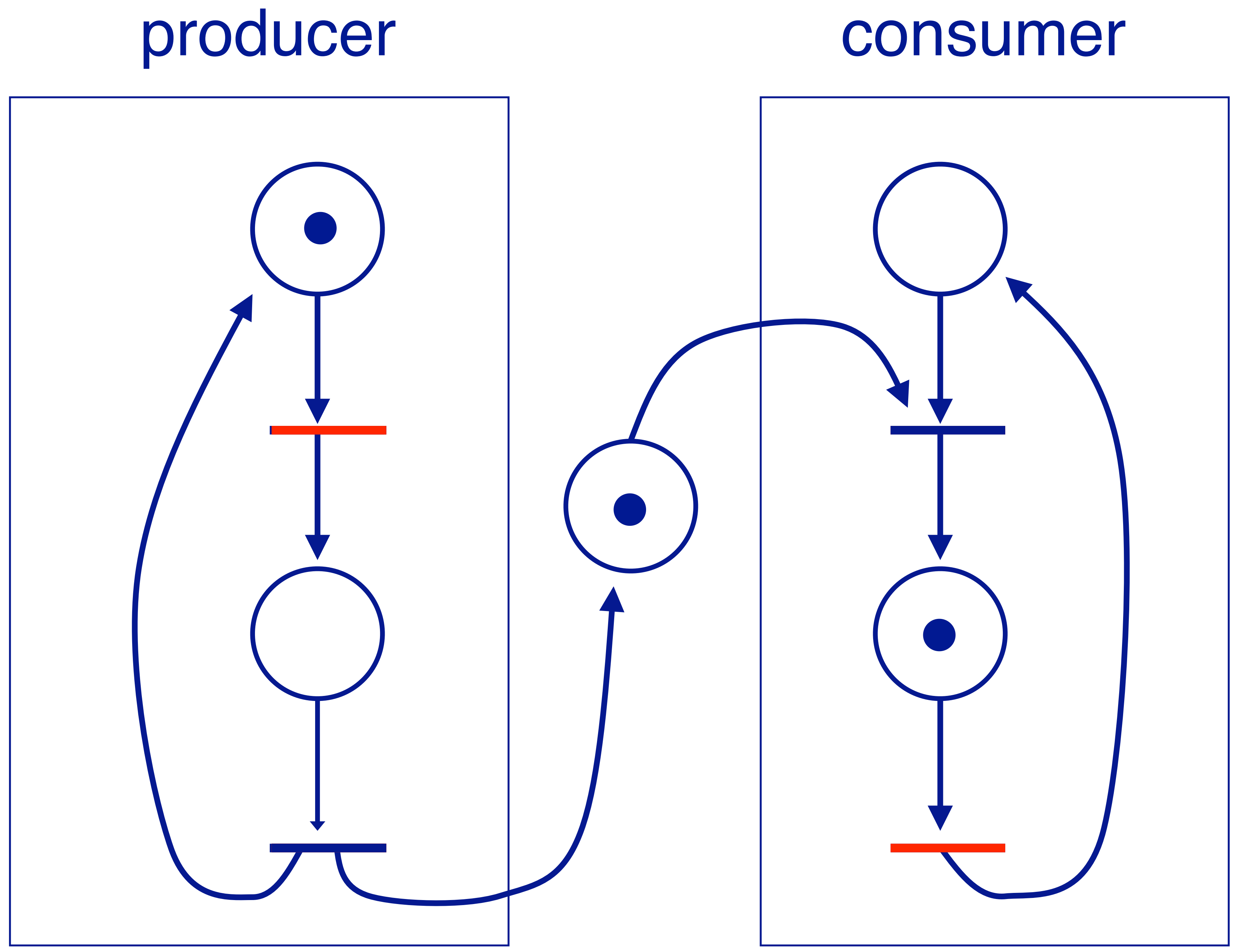
Producers and Consumers



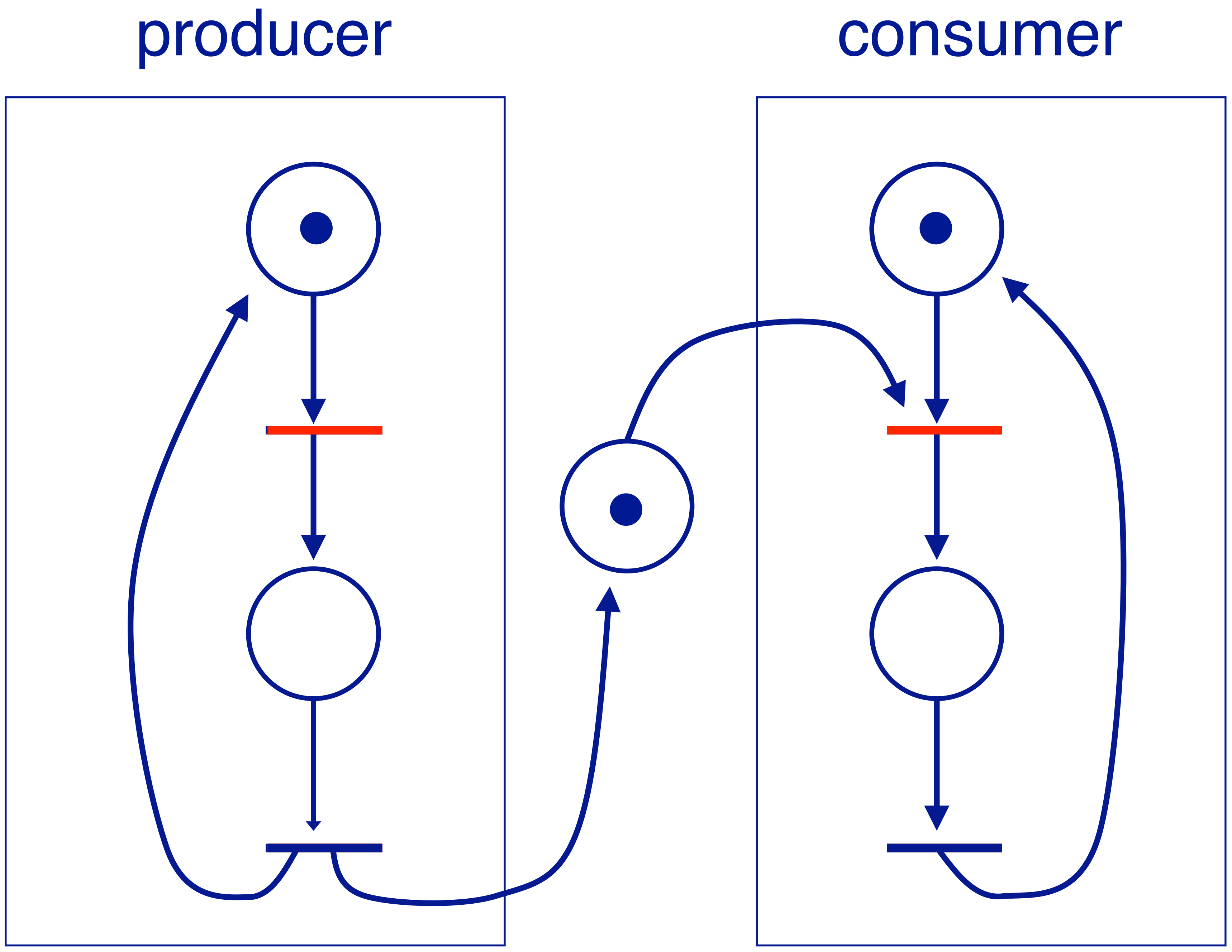
Producers and Consumers



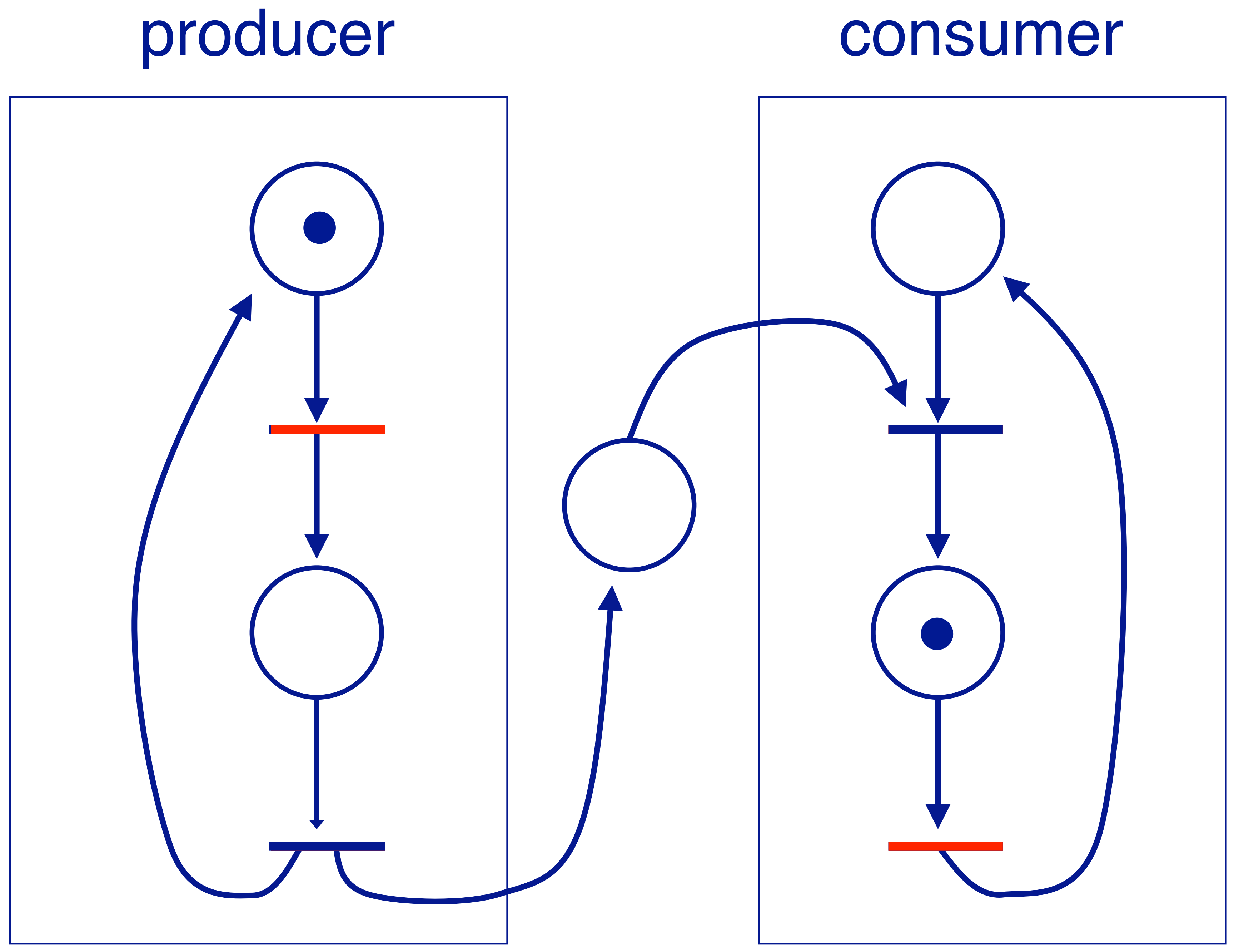
Producers and Consumers



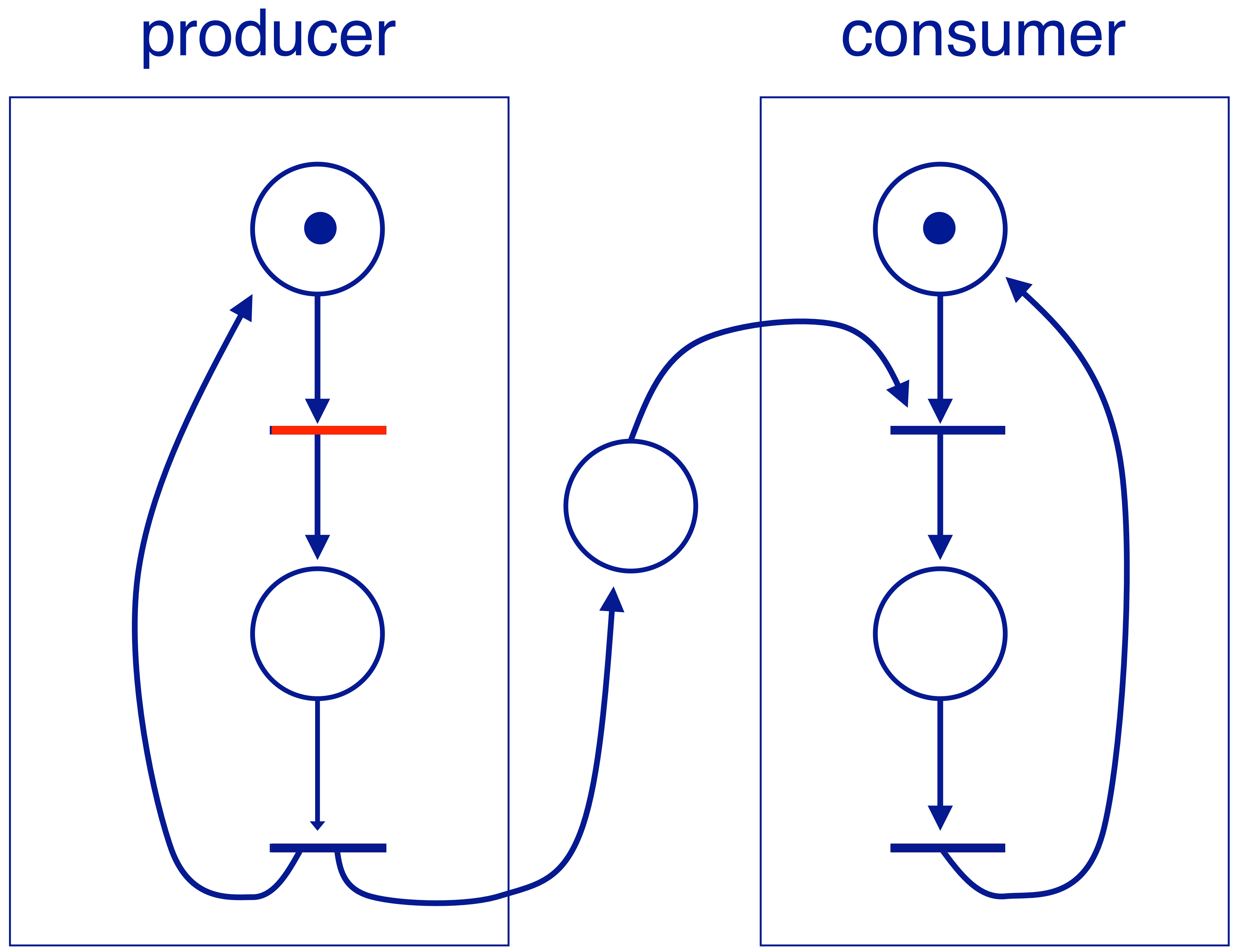
Producers and Consumers



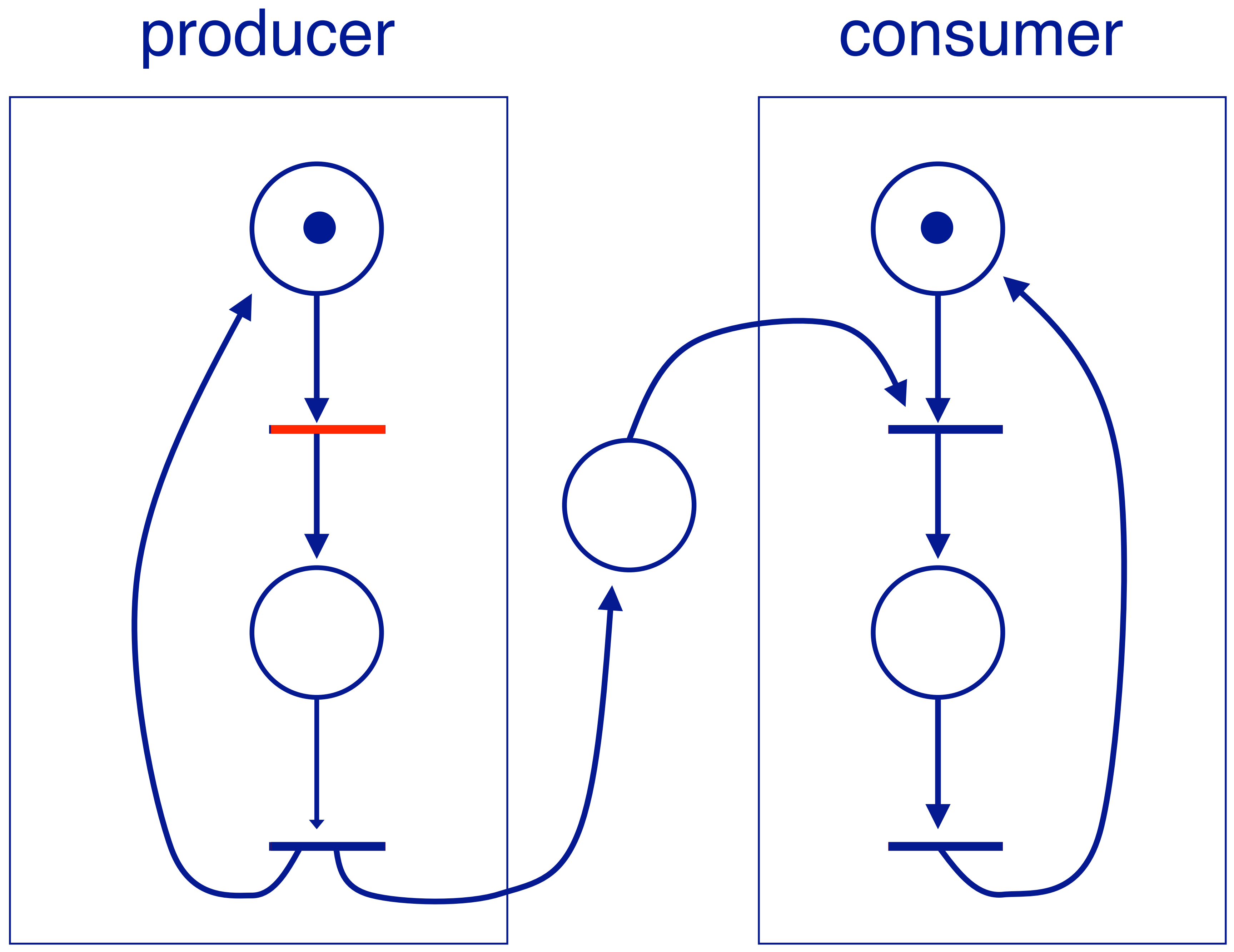
Producers and Consumers



Producers and Consumers

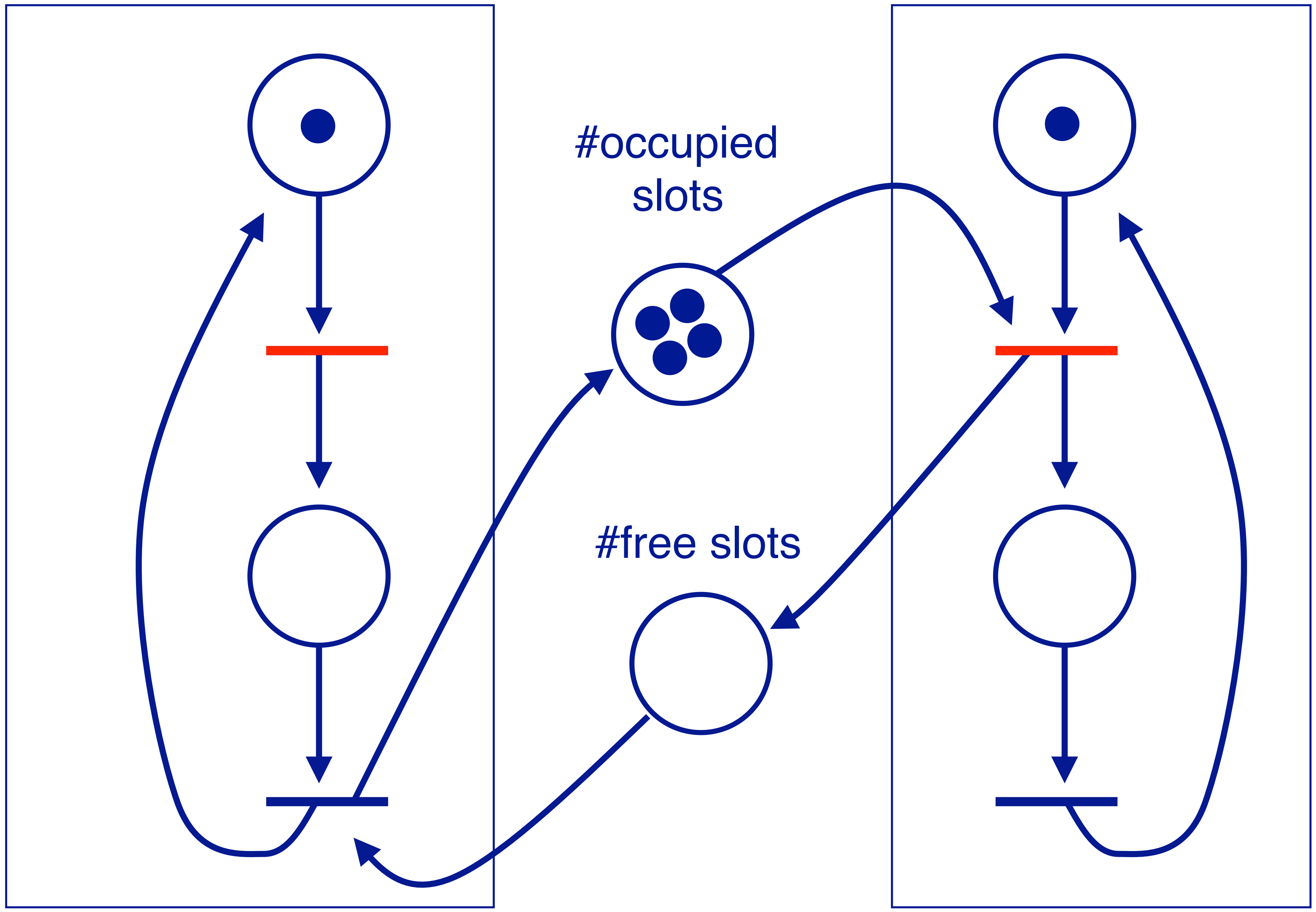


Producers and Consumers

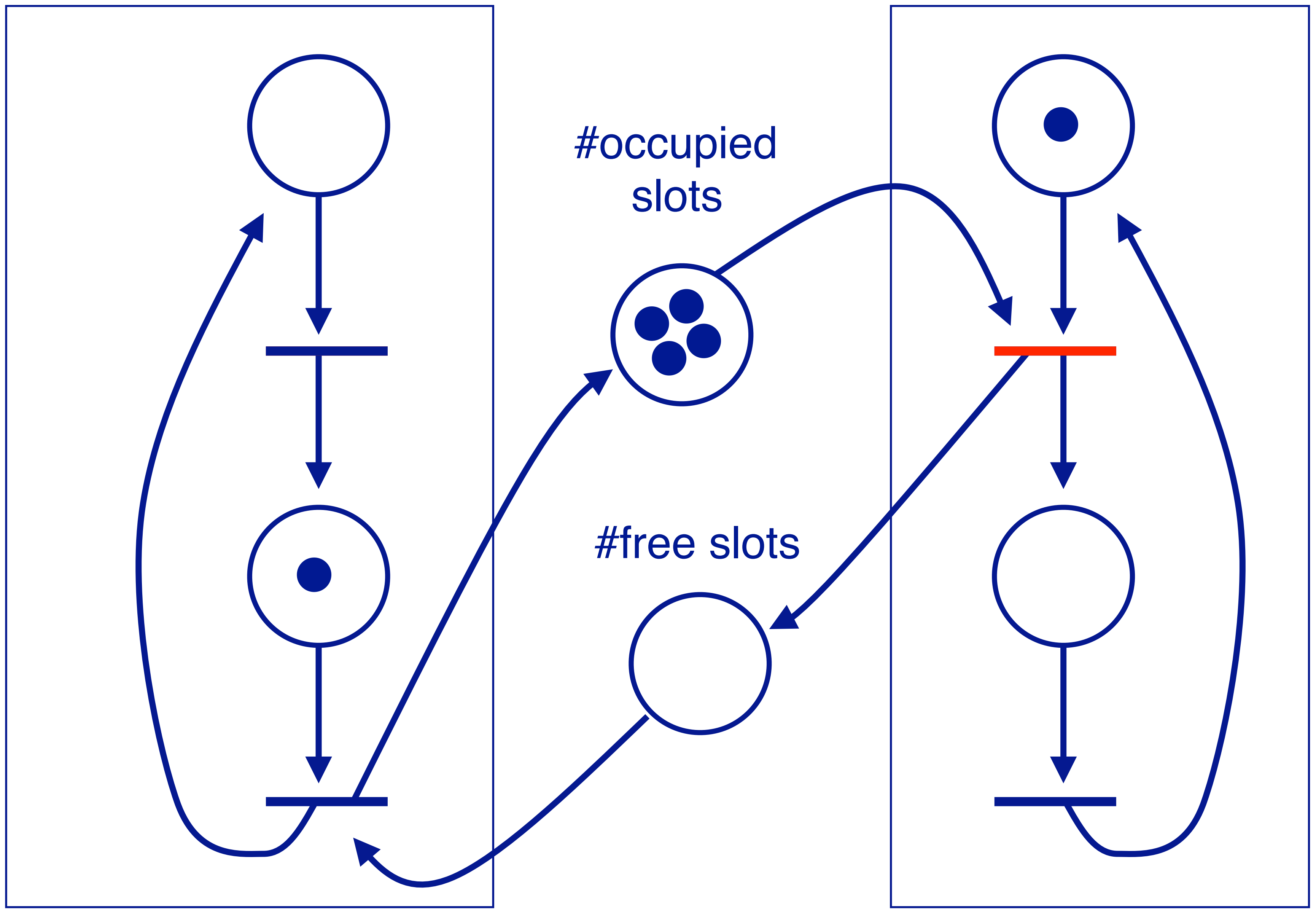


Here we again have subnets representing communicating processes, with tokens inside the subnets representing flow of control. The place connecting them, however, this time represents a buffer, with the tokens it contains representing payloads (messages) produced by one subnet and consumed by the other.

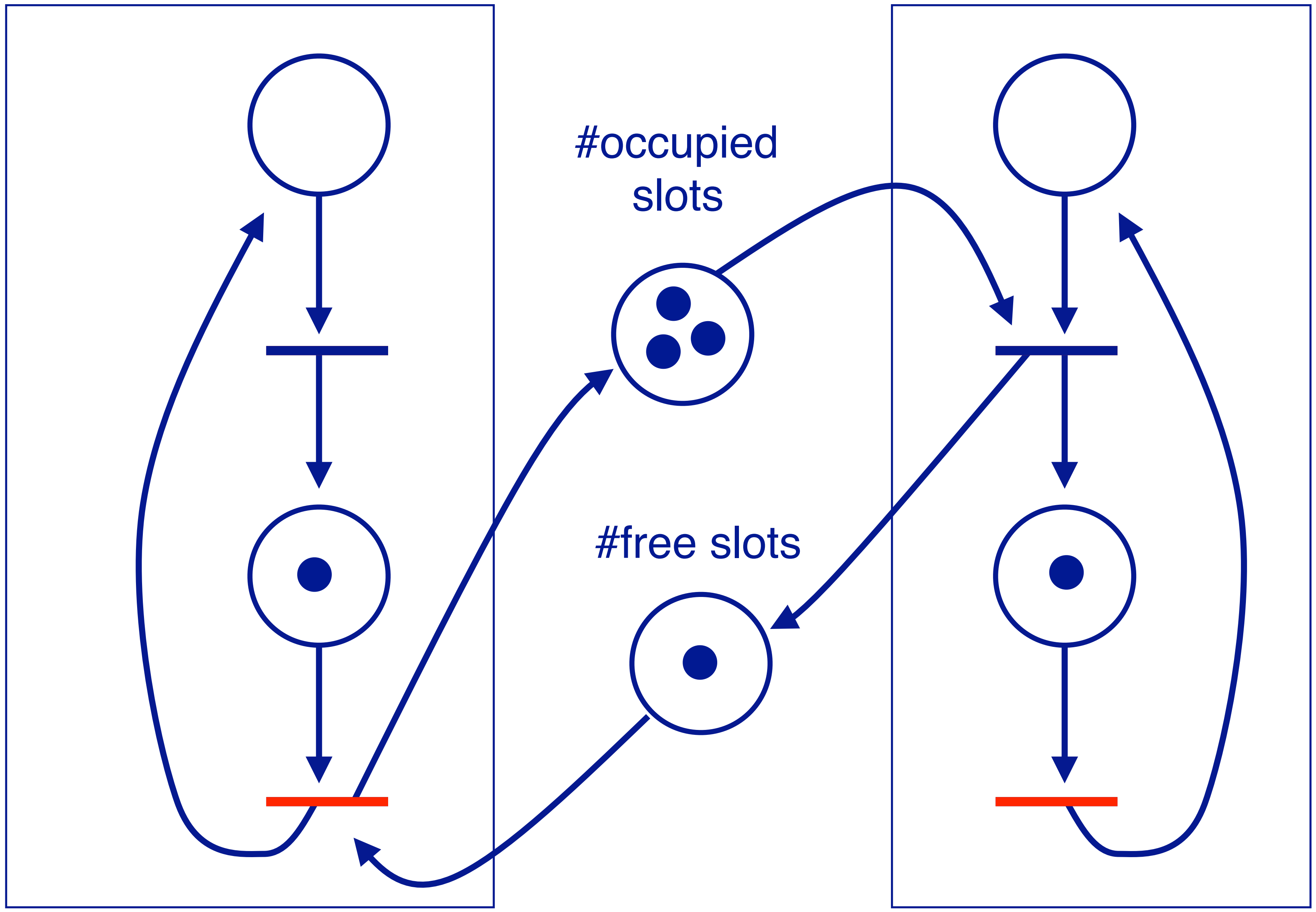
Bounded Buffers



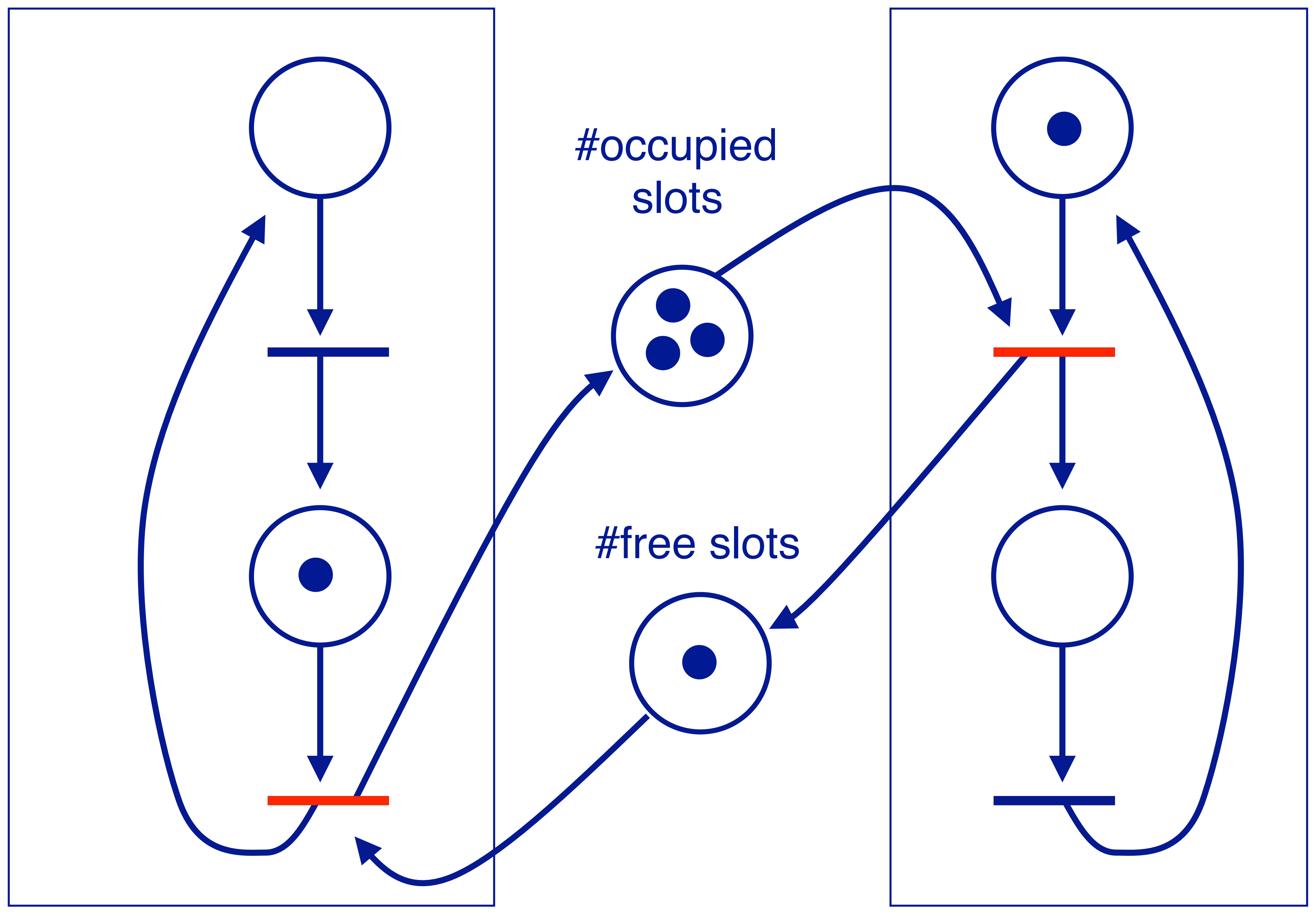
Bounded Buffers



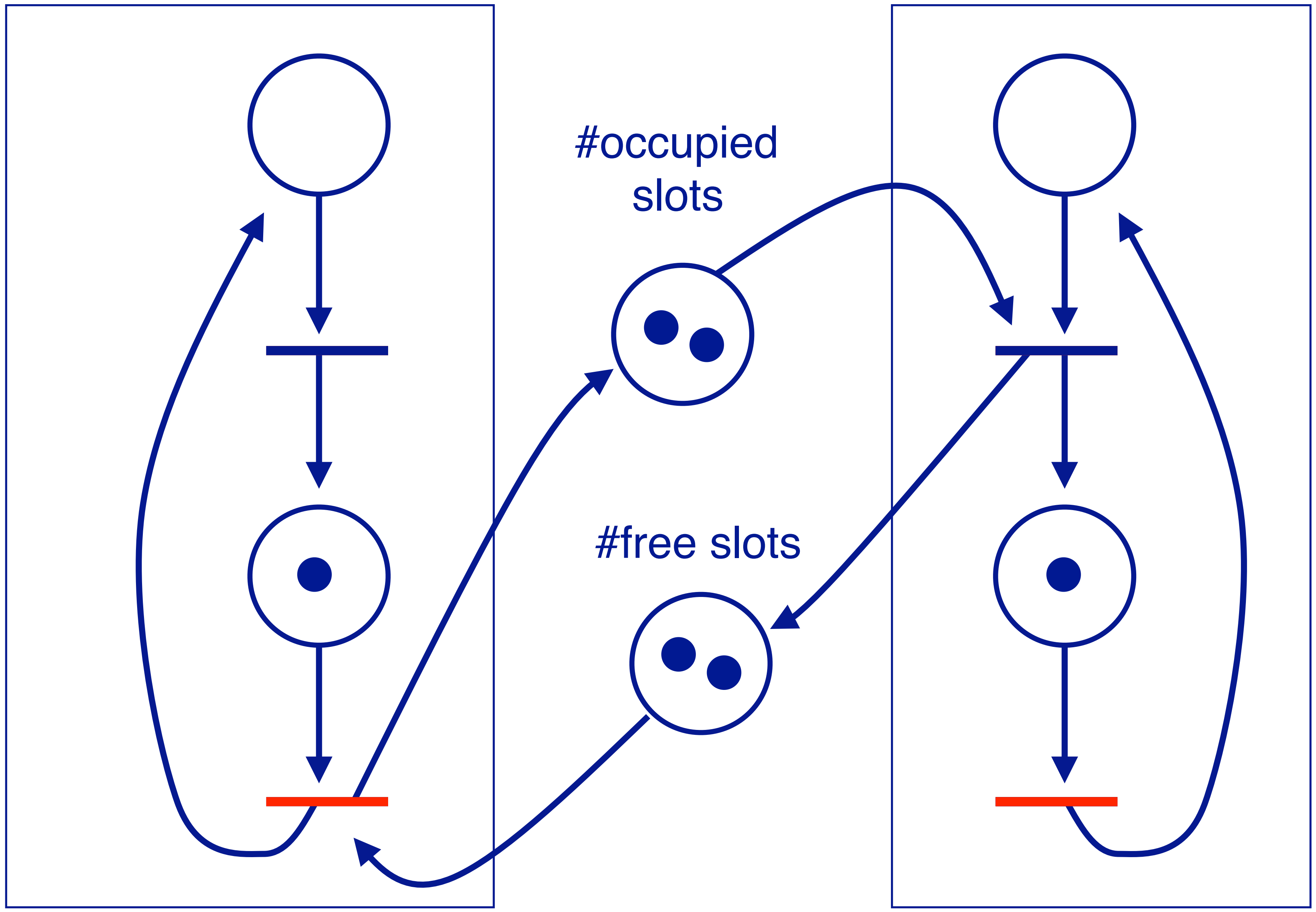
Bounded Buffers



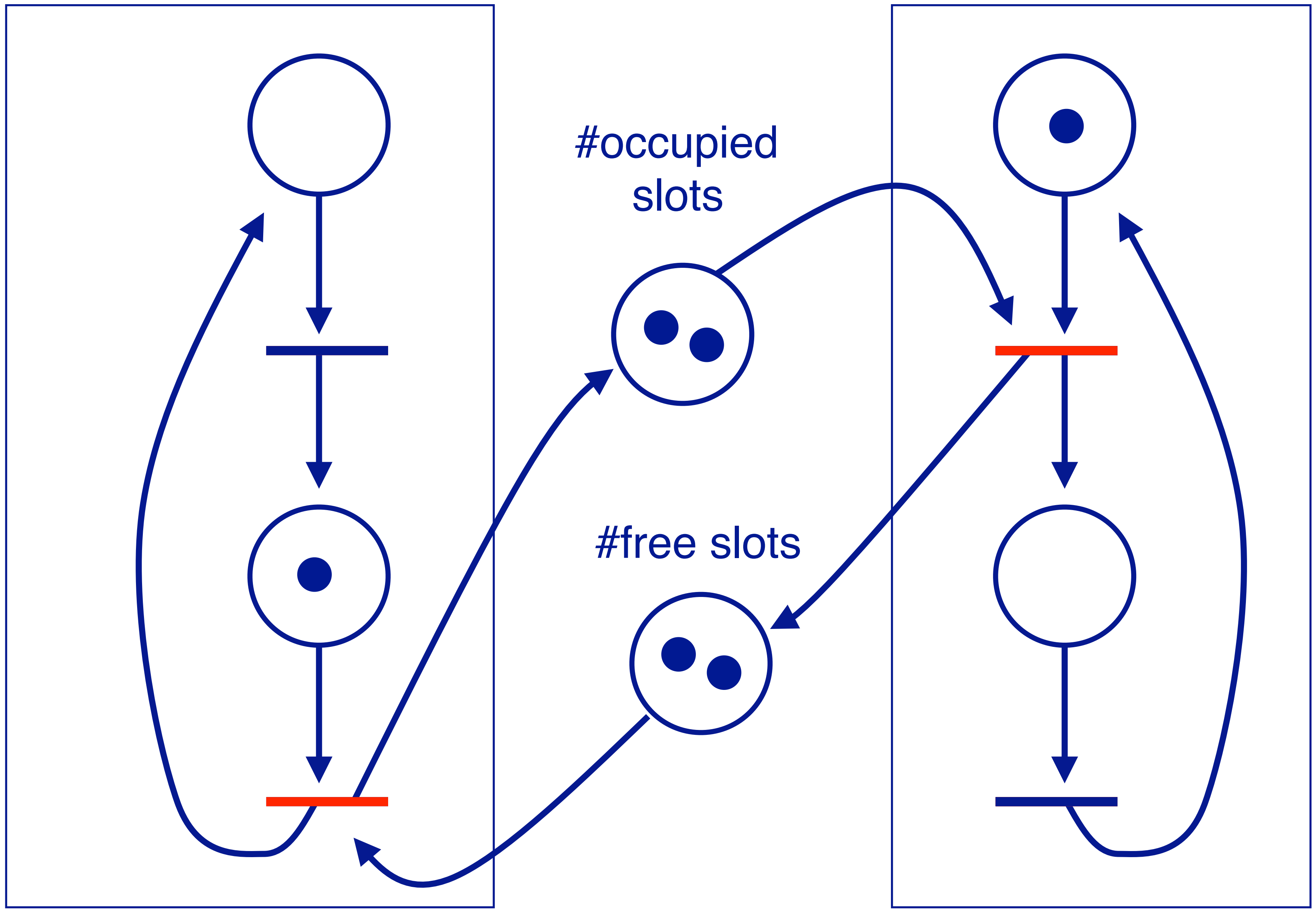
Bounded Buffers



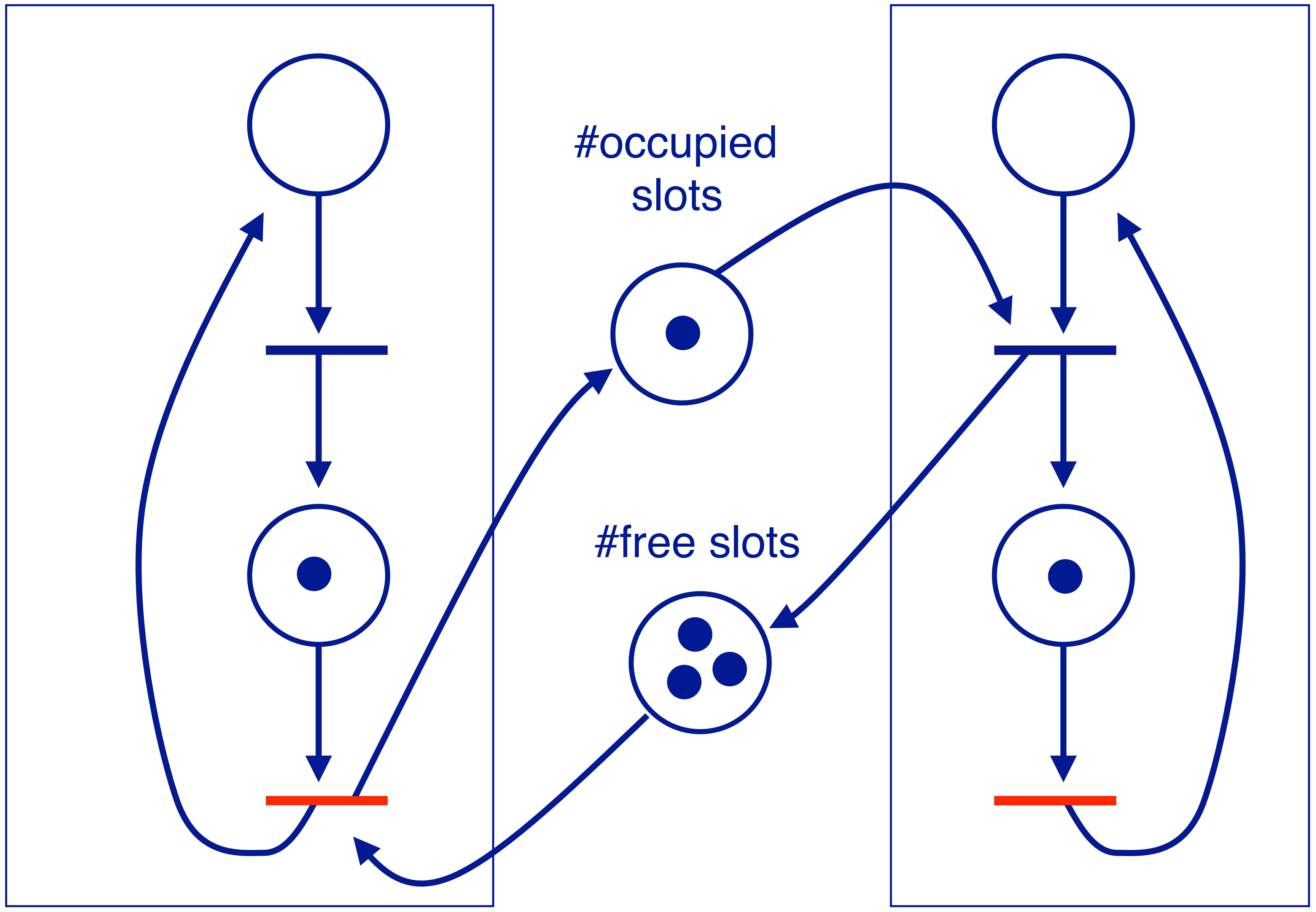
Bounded Buffers



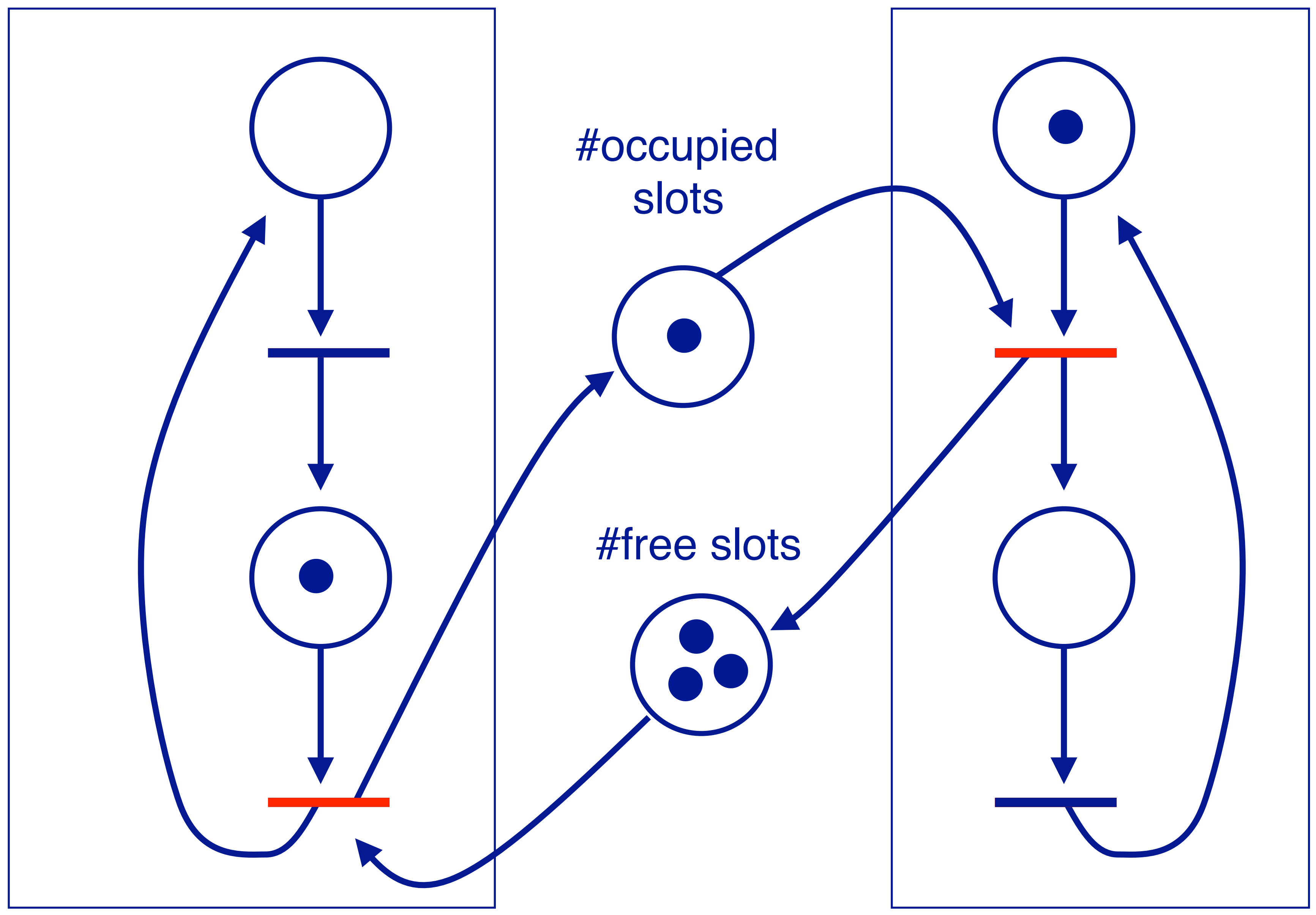
Bounded Buffers



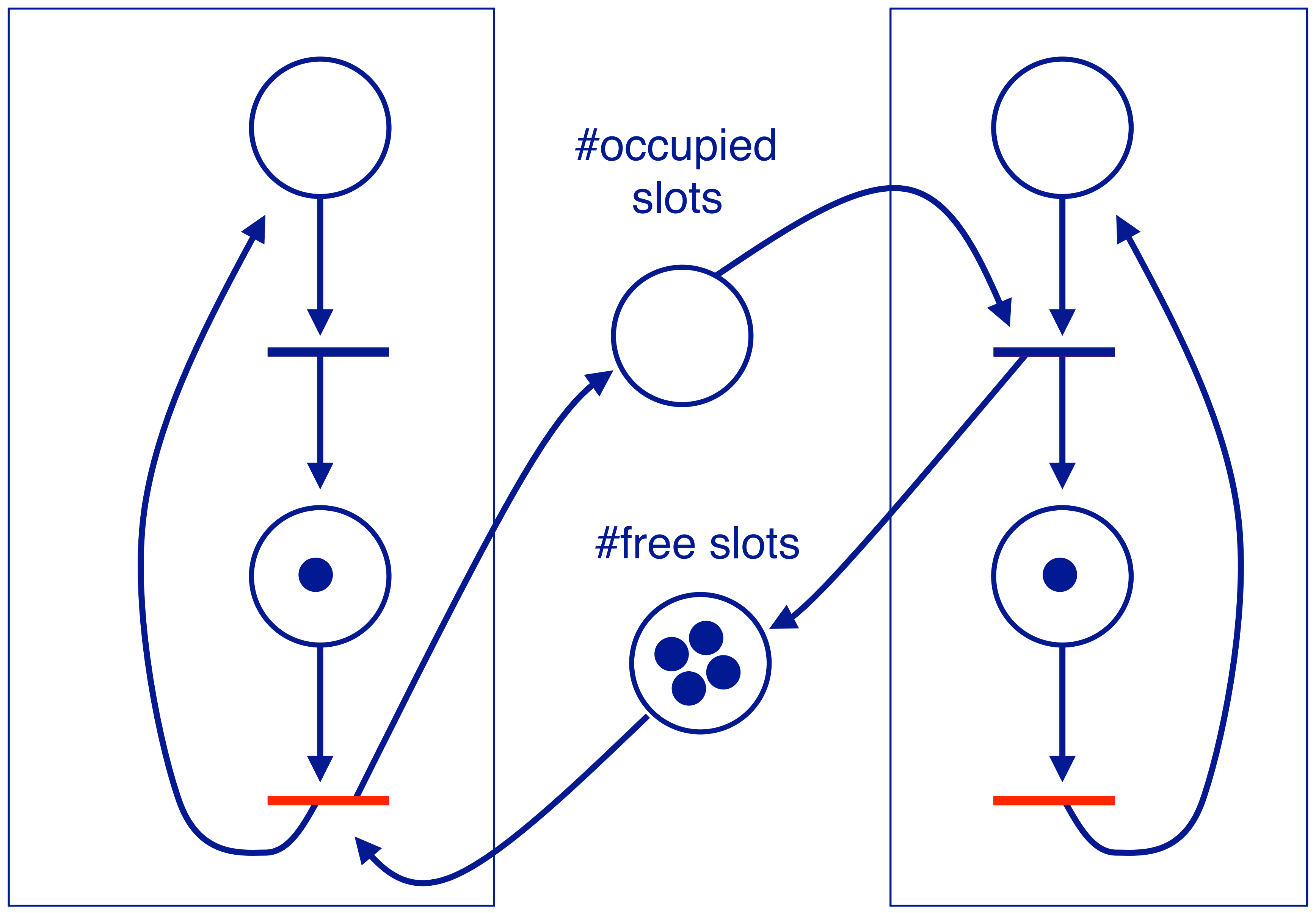
Bounded Buffers



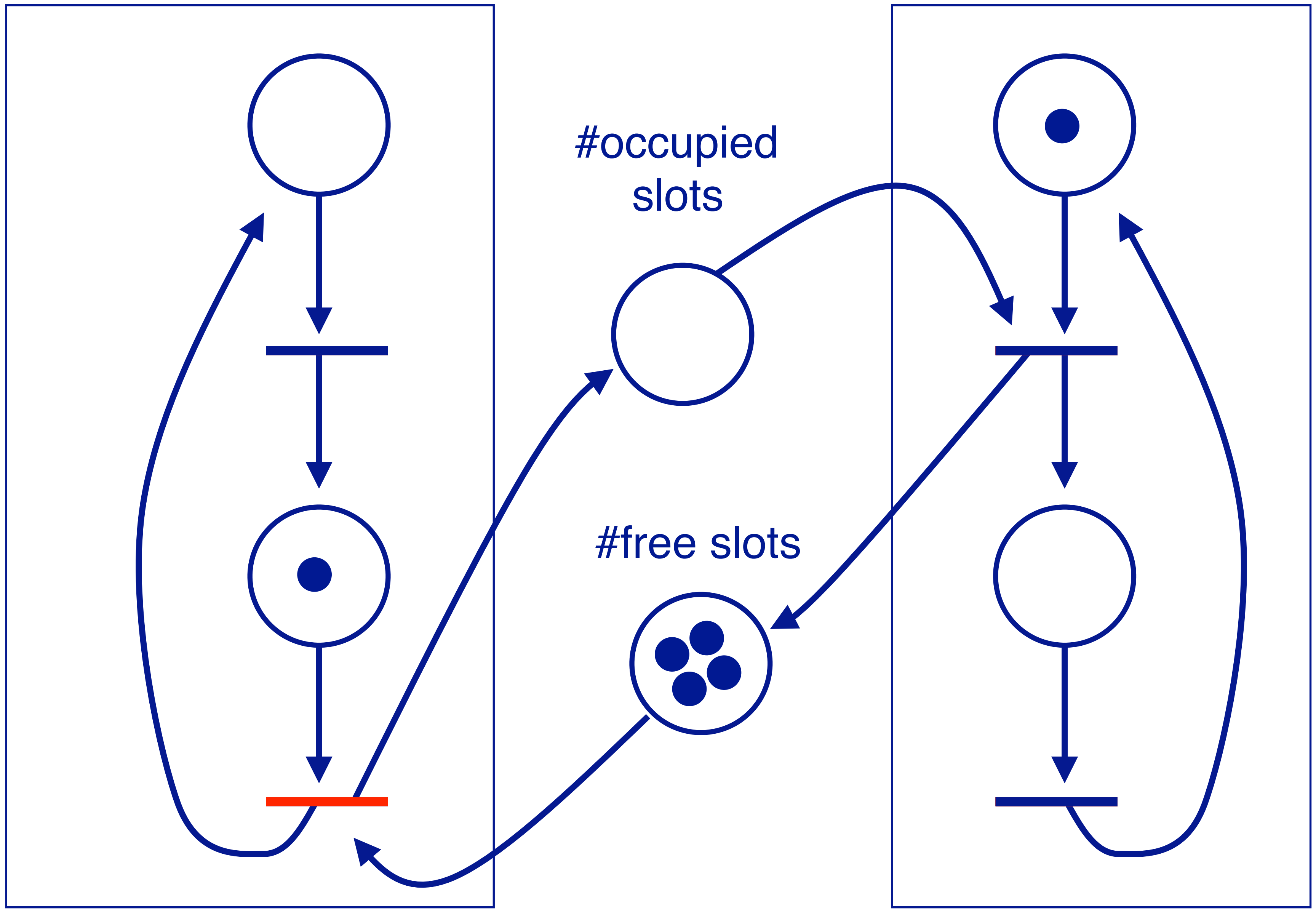
Bounded Buffers



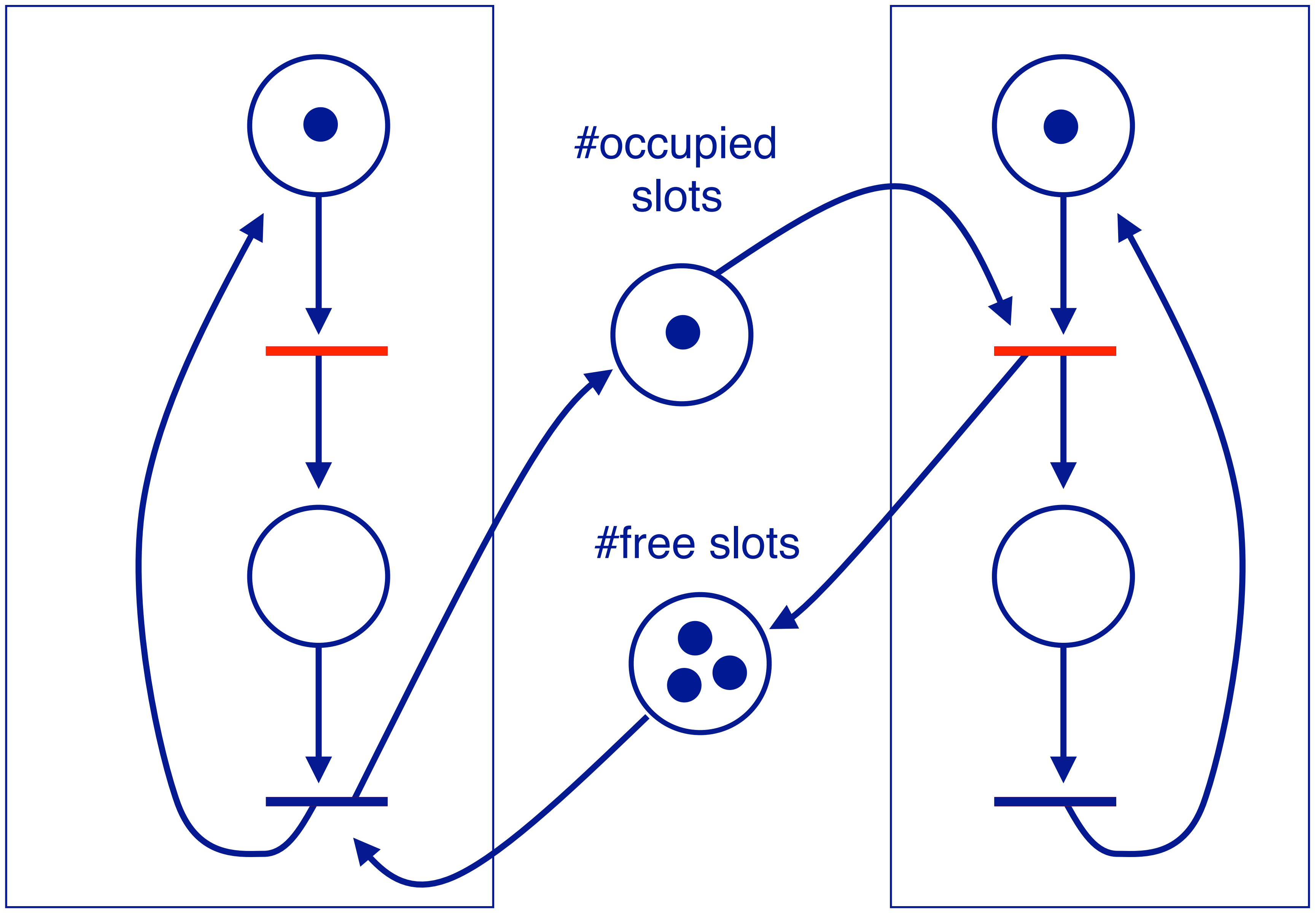
Bounded Buffers



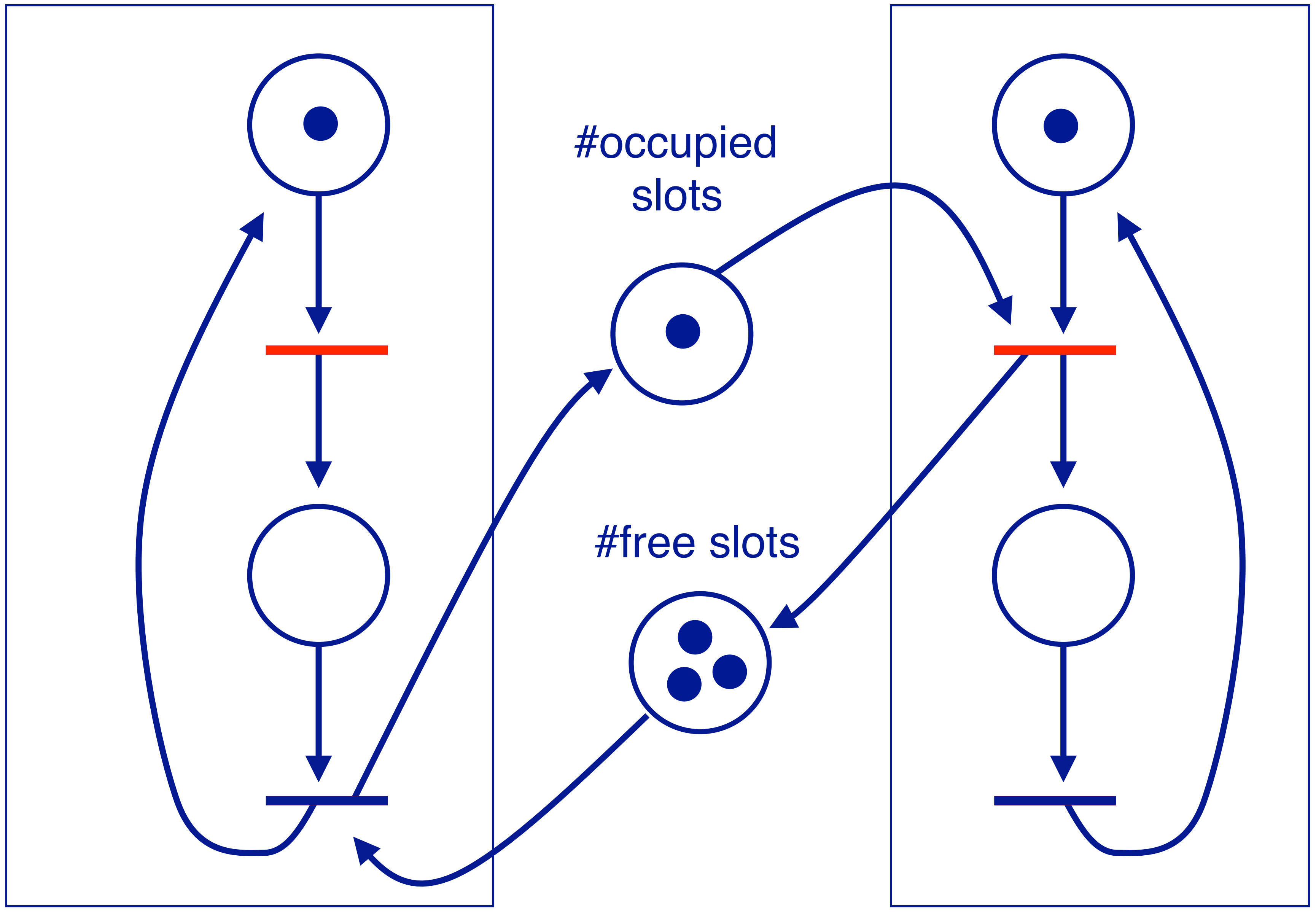
Bounded Buffers



Bounded Buffers



Bounded Buffers



While the previous example made use of an unbounded buffer, this example shows how a bounded buffer can be modeled. Tokens in the middle represent either the number of occupied slots or the available ones. The producer needs at least one free slot to produce an output, and the consumer needs at least one full slot to consume an input.

In this way the producer cannot get ahead of the consumer.

Roadmap

> Definition:
  - places, transitions, inputs, outputs
  - firing enabled transitions
> Modelling:
  - concurrency and synchronization
> **Properties of nets:**
  - **liveness, boundedness**
> Implementing Petri net models:
  - centralized and decentralized schemes

Reachability and Boundedness

Reachability:

> The reachability set R(C,m) of a net C is the set of all markings m′ *reachable from initial marking m*.

Boundedness:

> A net C with initial marking m is safe if places always hold *at most 1 token*.

> A marked net is (k-)bounded if places *never hold more than k tokens*.

> A marked net is conservative if the number of tokens is *constant*.

Note that each marking *m* represents a possible state of the net. As we have seen in the various examples, the reachability set may be finite or infinite.

A safe net is clearly k-bounded (k=1).

Are conservative nets necessarily k-bounded? Is the reverse true?

Which of these have finite reachability sets?

Liveness and Deadlock

Liveness:

> A transition is deadlocked if it can *never fire.*

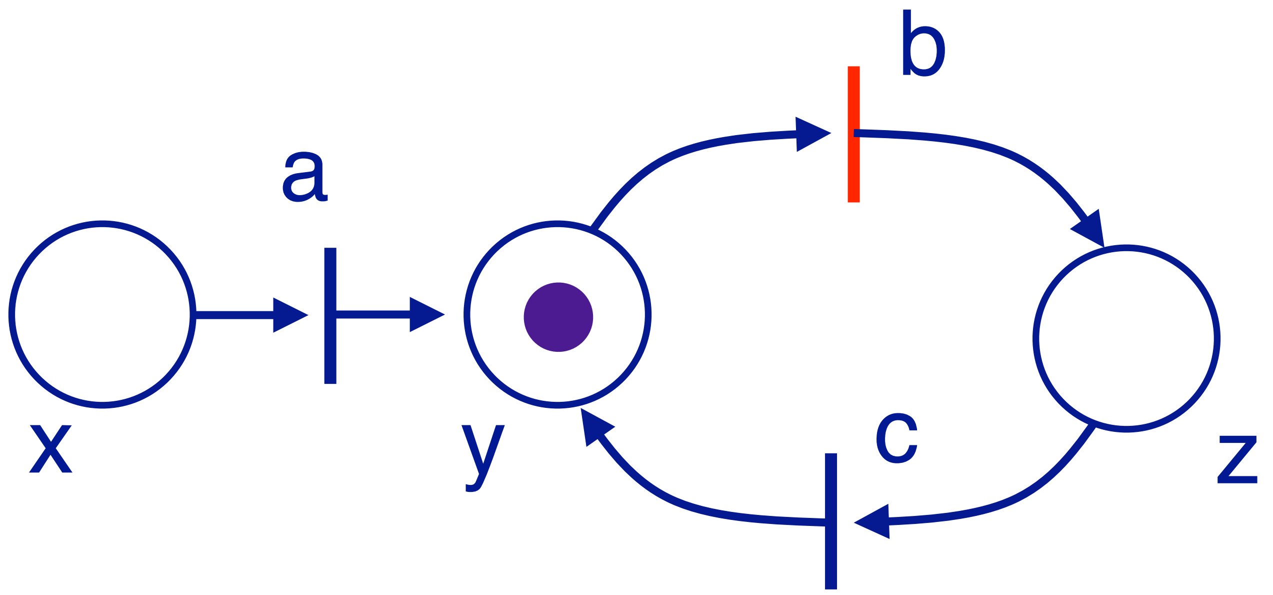
> A transition is live if it can *never deadlock.*

This net is both *safe and conservative*.

Transition a is *deadlocked*.

Transitions b and c are *live*.

The reachability set is {{y}, {z}}.



Are the examples we have seen bounded? Are they live?

Note that liveness is a very strong condition, since it states that it is always possible for the transition to become enabled again. However there is no guarantee that it *will* fire, only that it might.

Go back through all the examples and for each net explain whether or not is it is bounded (safe, conservative, k-bounded) or live.

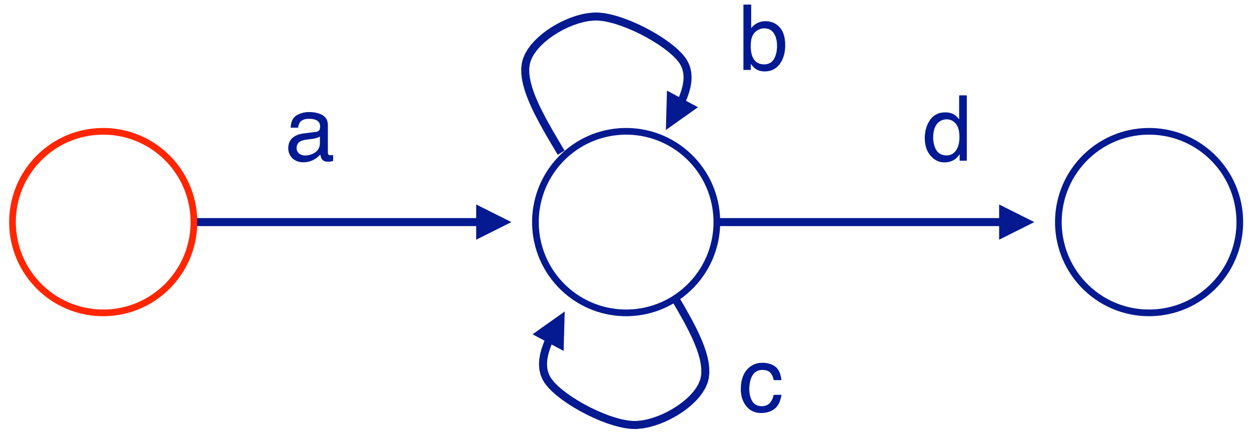
Related Models

Finite State Processes

> Equivalent to *regular expressions*

> Can be modelled by *one-token conservative nets*

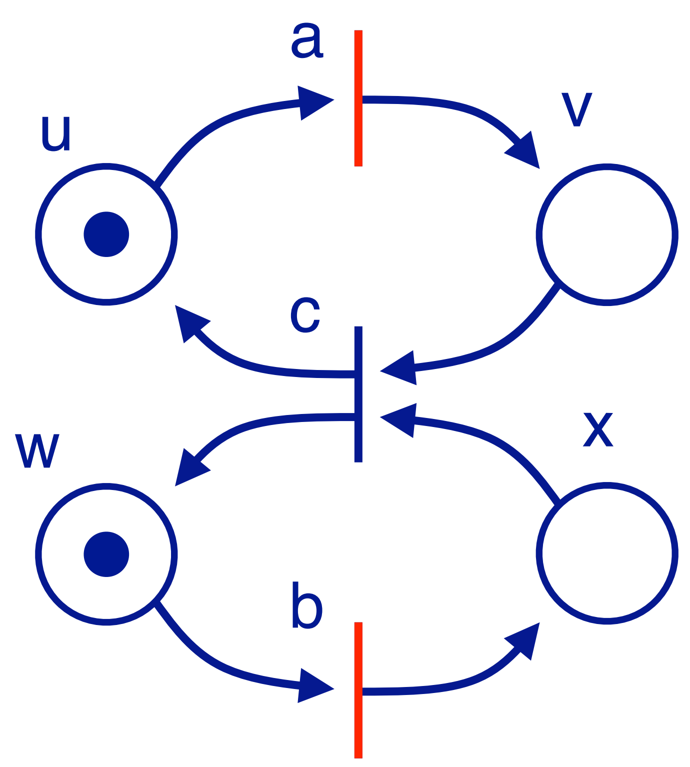
The FSA for: a(b|c)*d

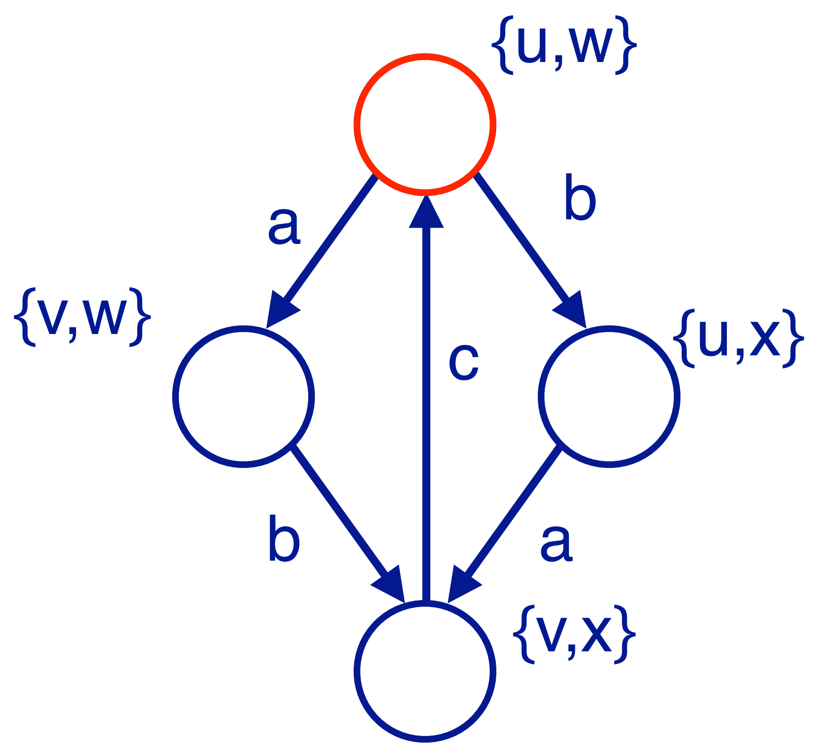


Finite state processes (FSPs) can easily be modeled by Petri nets simply by adding a single net transition between every pair of connected states. A single token then models the current state.

Finite State Nets

Some Petri nets can be modelled by FSPs

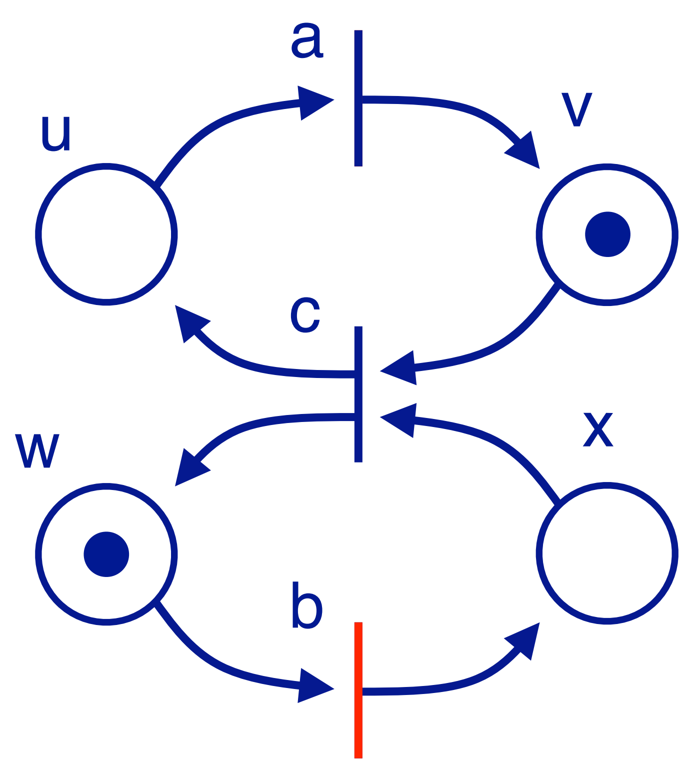


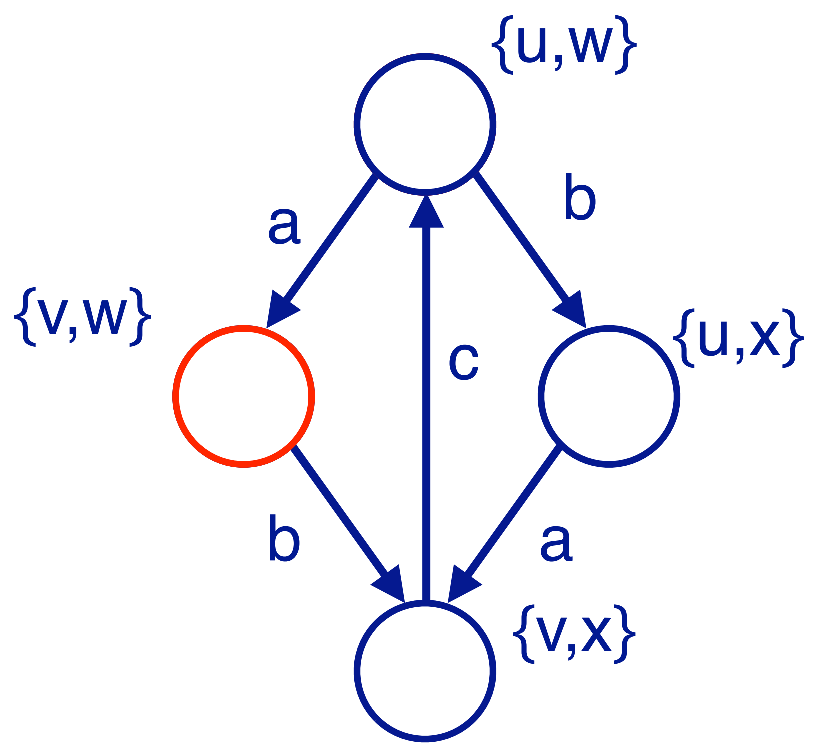


Precisely which nets can (cannot) be modelled by FSPs?

Finite State Nets

Some Petri nets can be modelled by FSPs

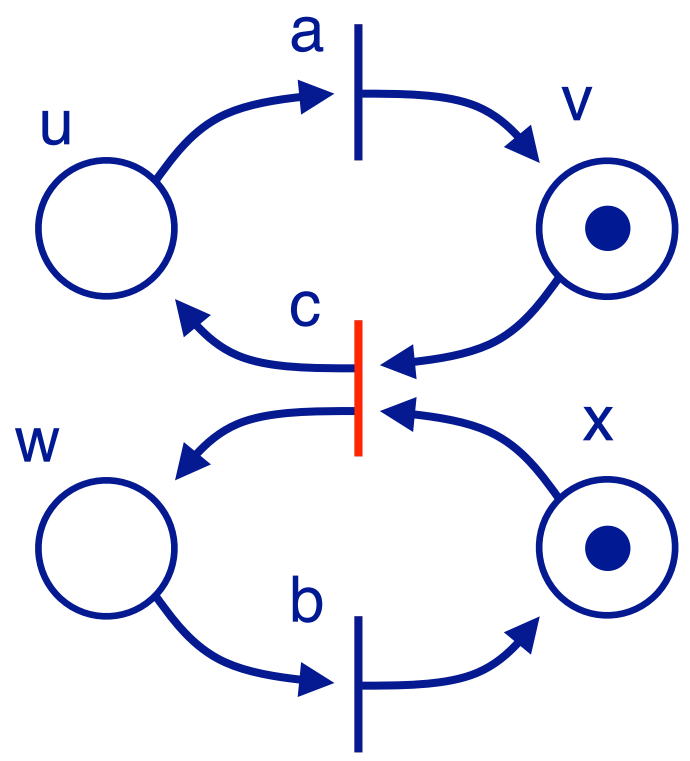


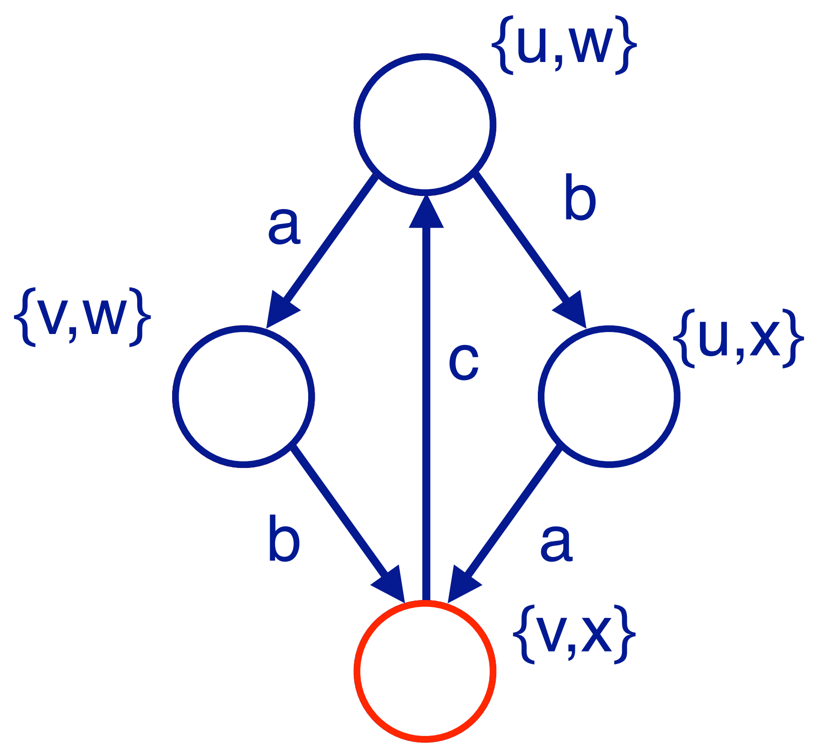


Precisely which nets can (cannot) be modelled by FSPs?

Finite State Nets

Some Petri nets can be modelled by FSPs

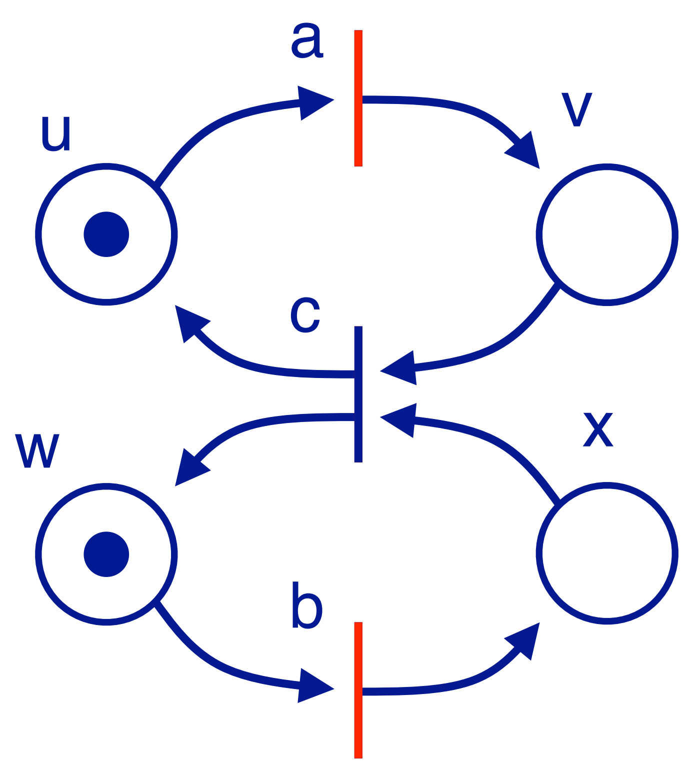


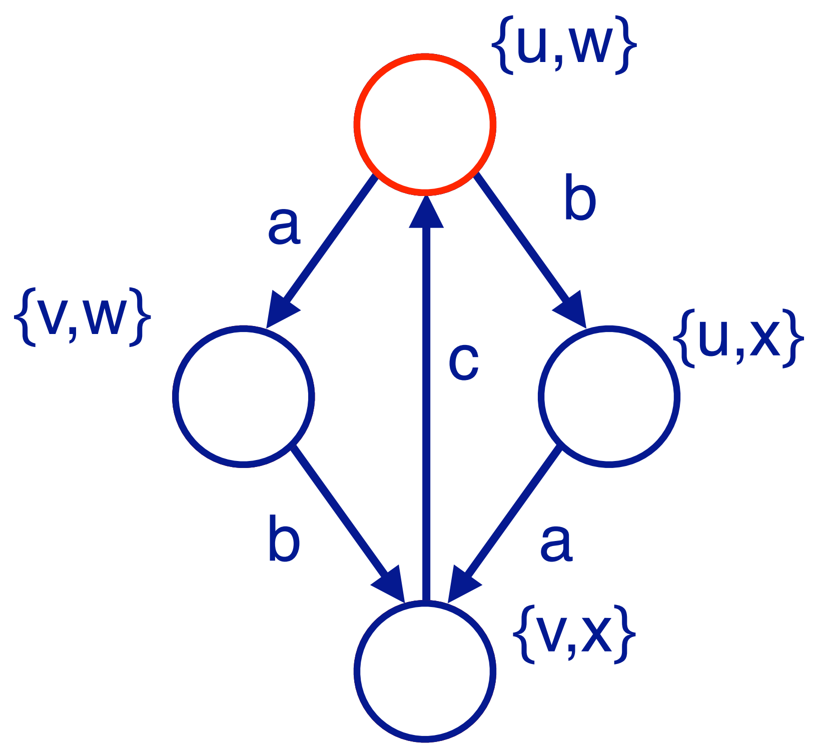


Precisely which nets can (cannot) be modelled by FSPs?

Finite State Nets

Some Petri nets can be modelled by FSPs

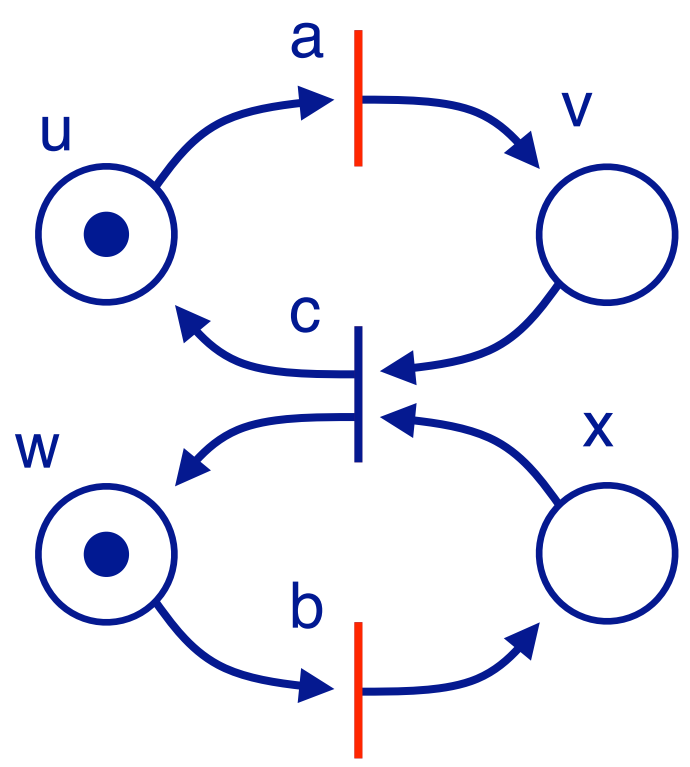


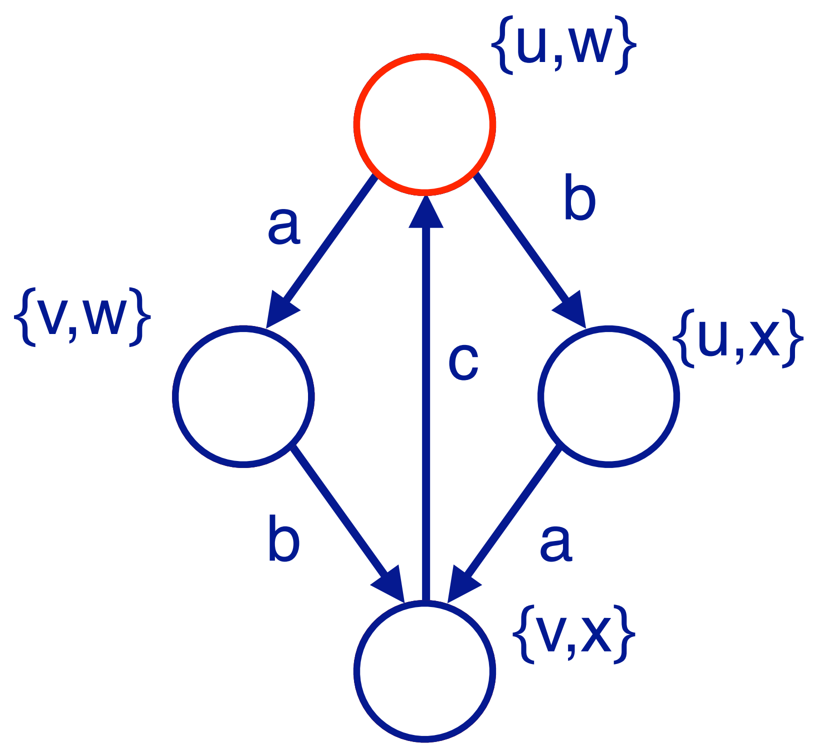


Precisely which nets can (cannot) be modelled by FSPs?

Finite State Nets

Some Petri nets can be modelled by FSPs





Precisely which nets can (cannot) be modelled by FSPs?

Any bounded net, i.e., with a finite reachability set, can be modeled by a FSA. Simply introduce one state for each reachable marking FSA indicating which net transitions fire between the states, and addd a single token for the initial marking.

In the example, there are just four reachable markings, hence four states in the FSP.

NB: An infinite reachability set does not guarantee that an equivalent FSP does not exist. A counterexample is the first example of the slide deck, which models the regular language a*b, so even though it has an infinite reachability set, it is equivalent to the FSP (0)—a→(0), (0)—b→(STOP).

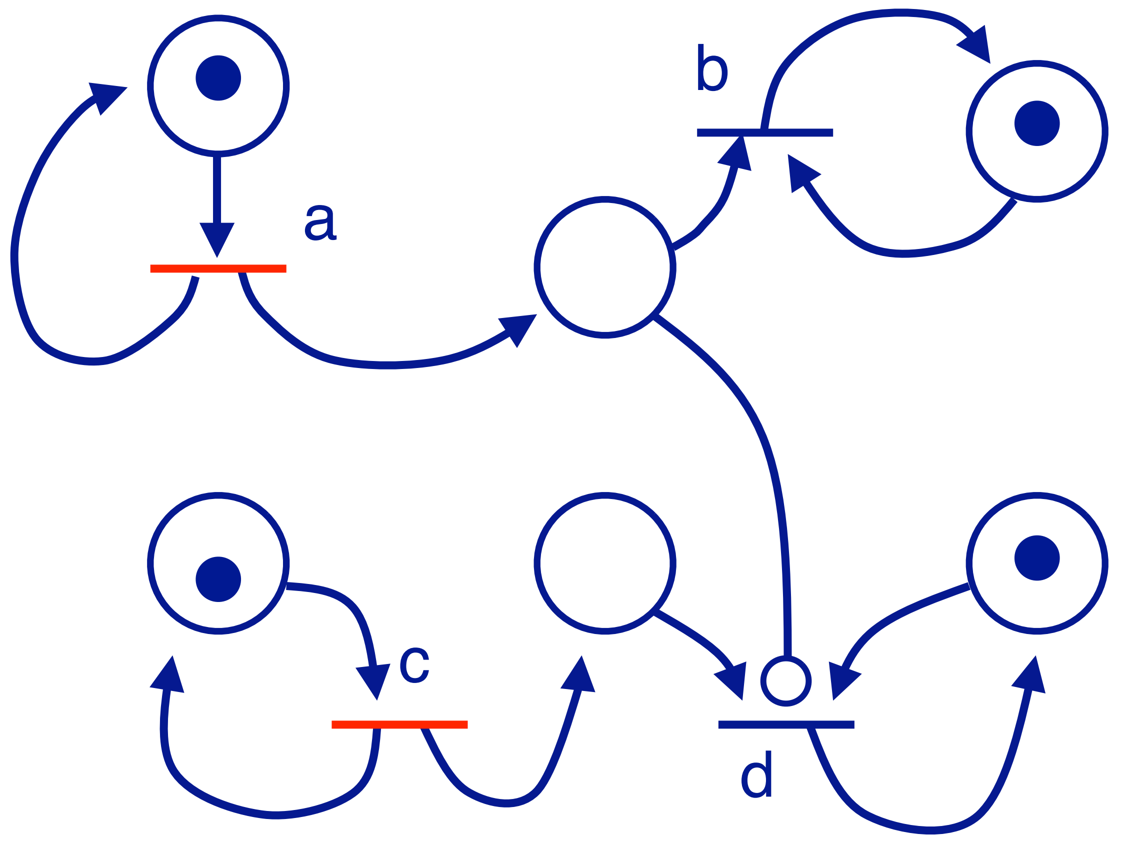
Zero-testing Nets

Petri nets are not computationally complete

> Cannot model “zero testing”

> Cannot model priorities

A zero-testing net: An equal number of a and b transitions may fire as a sequence during any sequence of matching c and d transitions.
(#a ≥ #b, #c ≥ #d)

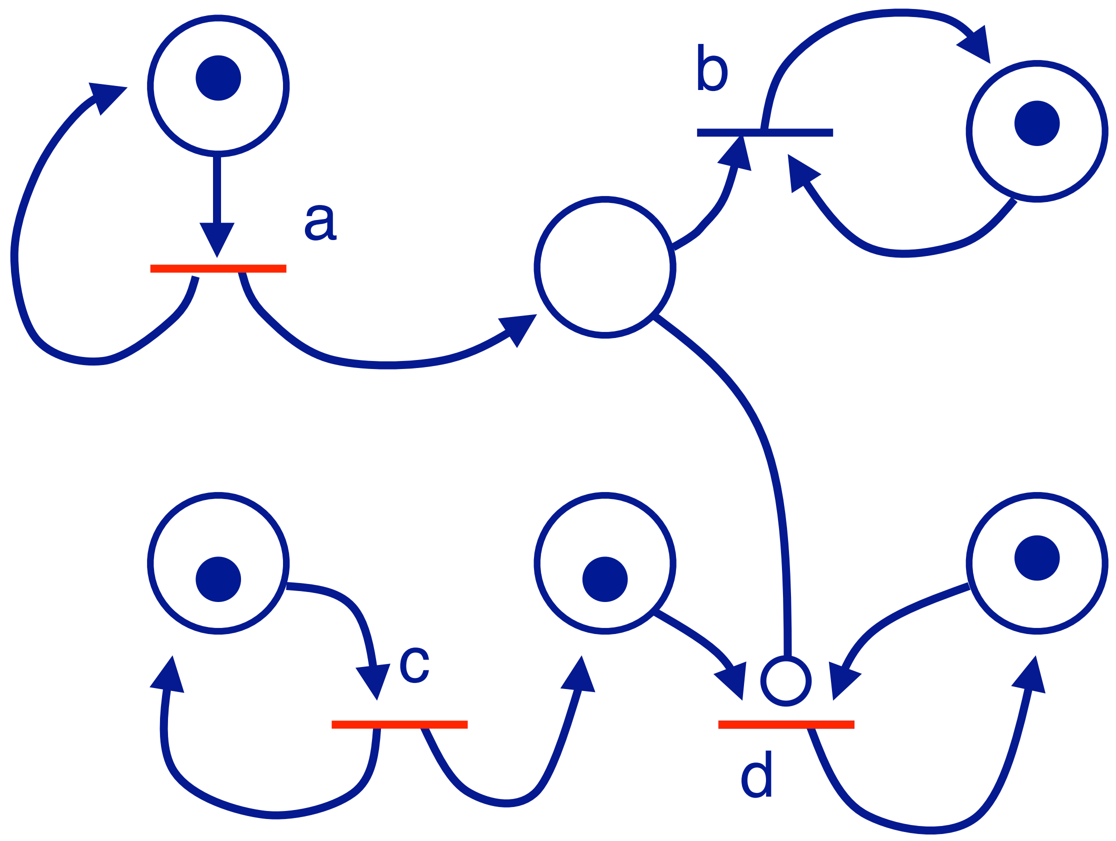


Zero-testing Nets

Petri nets are not computationally complete

> Cannot model "zero testing"

> Cannot model priorities



A zero-testing net: An equal number of a and b transitions may fire as a sequence during any sequence of matching c and d transitions.
(#a ≥ #b, #c ≥ #d)

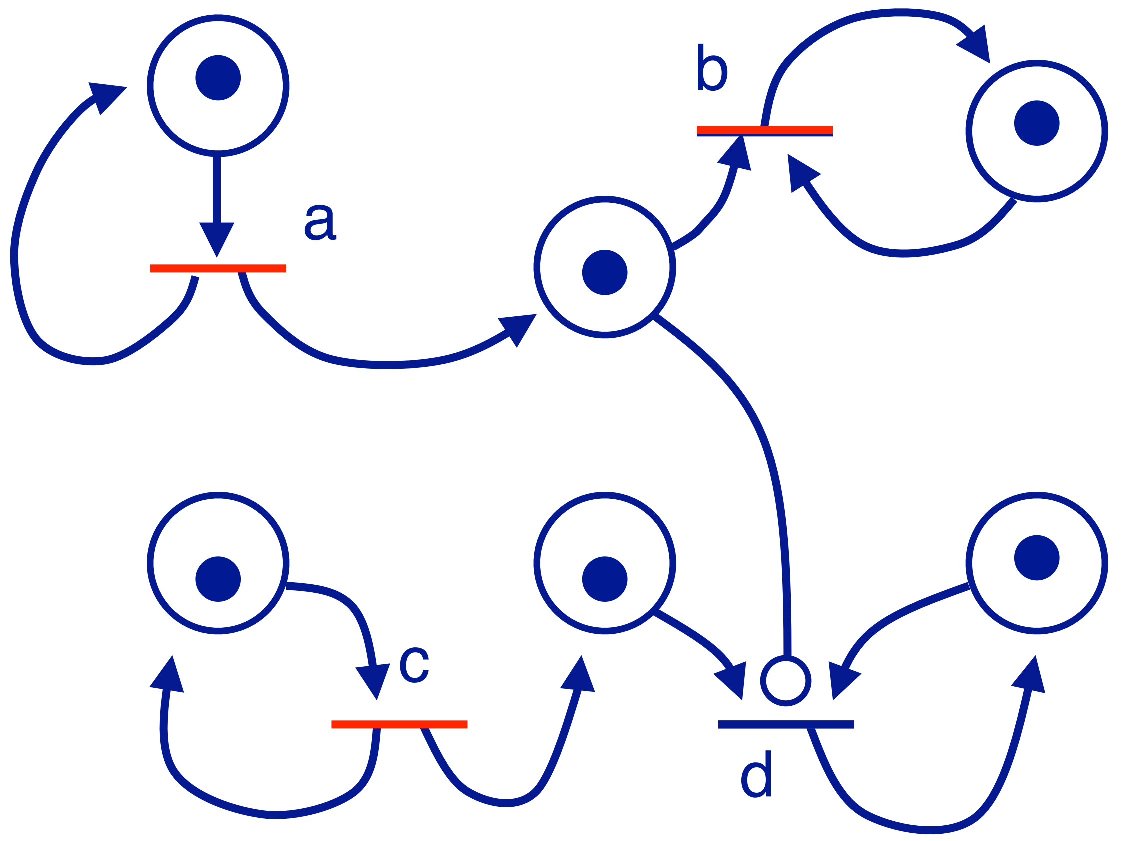
Zero-testing Nets

Petri nets are not computationally complete

> Cannot model "zero testing"

> Cannot model priorities

A zero-testing net: An equal number of a and b transitions may fire as a sequence during any sequence of matching c and d transitions.
(#a ≥ #b, #c ≥ #d)



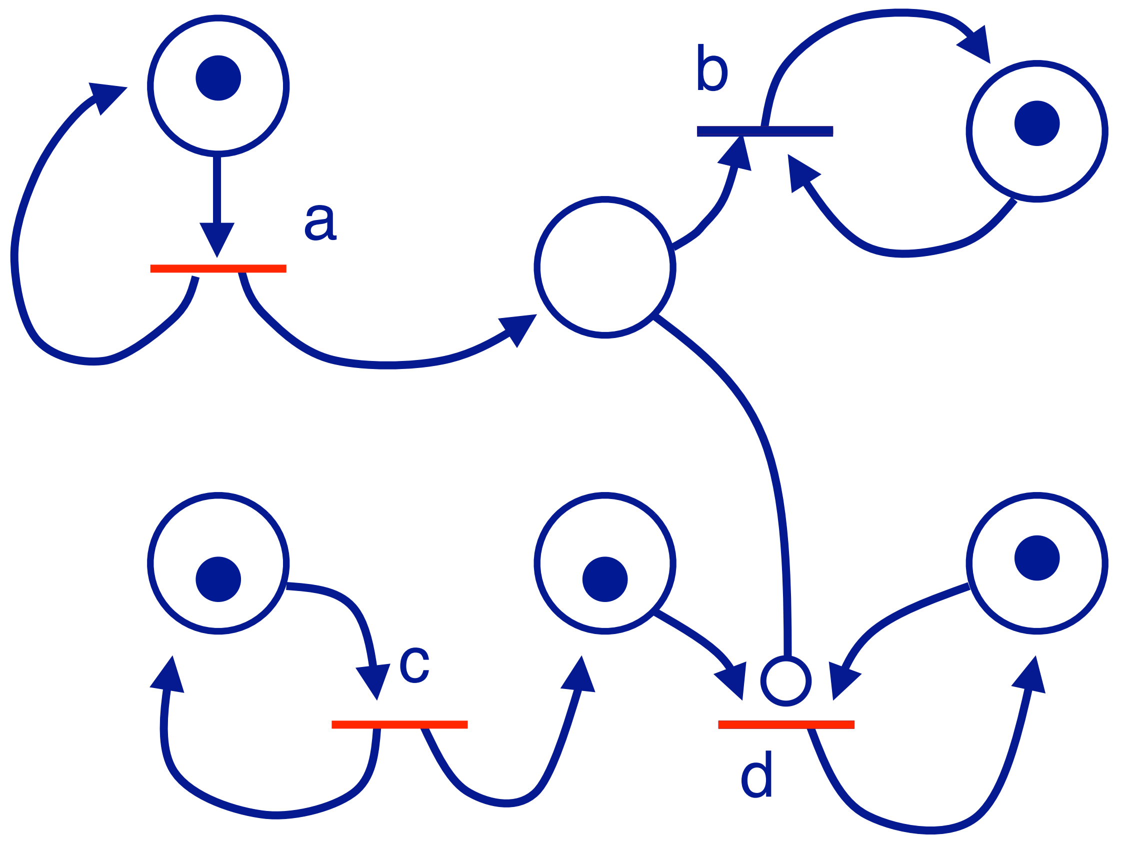
Zero-testing Nets

Petri nets are not computationally complete

> Cannot model "zero testing"

> Cannot model priorities

<u>A zero-testing net:</u> An equal number of a and b transitions may fire as a sequence during any sequence of matching c and d transitions.
(#a ≥ #b, #c ≥ #d)



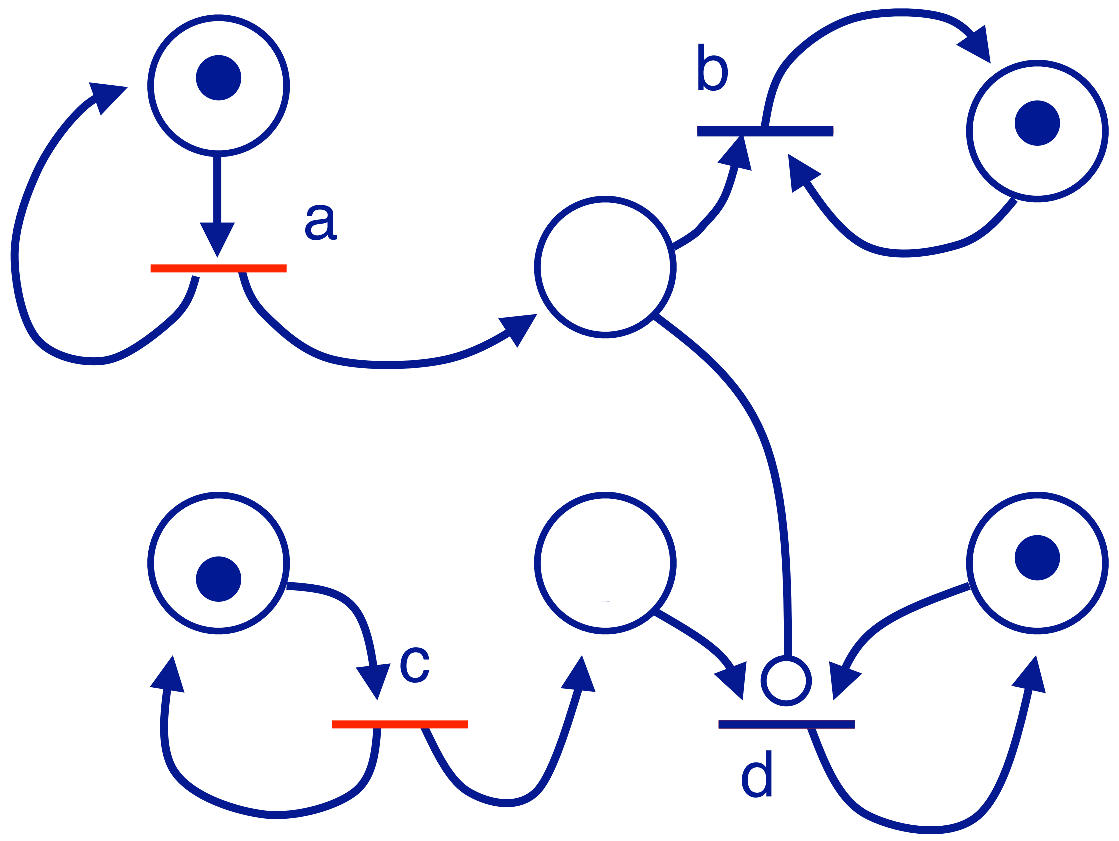
Zero-testing Nets

Petri nets are not computationally complete

> Cannot model "zero testing"

> Cannot model priorities

A zero-testing net: An equal number of a and b transitions may fire as a sequence during any sequence of matching c and d transitions.
(#a ≥ #b, #c ≥ #d)



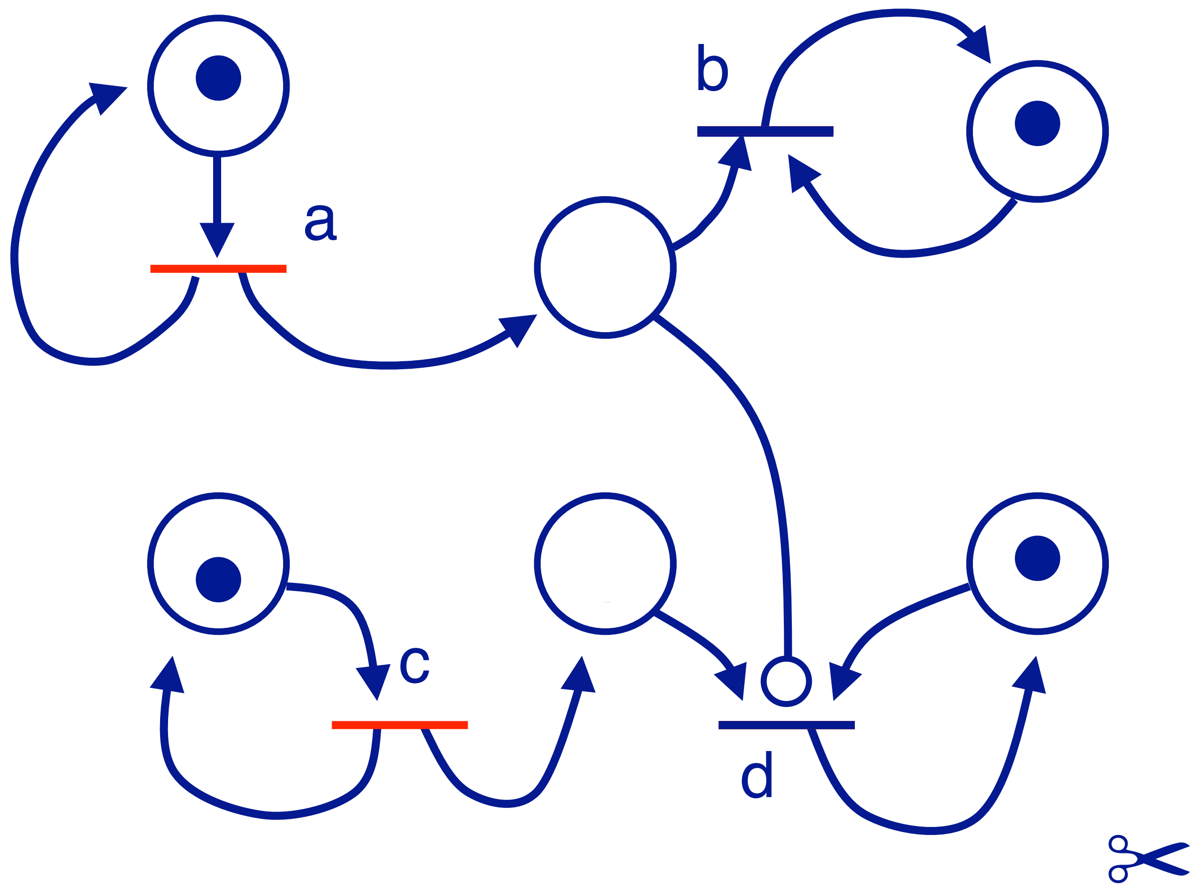
Zero-testing Nets

Petri nets are not computationally complete

> Cannot model “zero testing”

> Cannot model priorities

A zero-testing net: An equal number of a and b transitions may fire as a sequence during any sequence of matching c and d transitions.
(#a ≥ #b, #c ≥ #d)



Petri nets are strictly more powerful than FSPs (or FSAs), but less powerful than Turing machines. (Turing machines can generate languages that nets cannot, just as nets can generate languages that FSAs cannot.)

Adding almost any feature to nets (such as zero-testing), however, will make them Turing-complete.

In the example, d can only fire if the zero-input place between a and b is empty, i.e., if a and b have fired exactly the same number of times. This behaviour is impossible to express with a “plain” Petri net.

Other Variants

There exist countless variants of Petri nets

Coloured Petri nets:

> Tokens are "coloured" to represent different kinds of resources

Augmented Petri nets:

> Transitions additionally depend on external conditions

Timed Petri nets:

> A duration is associated with each transition

Augmented Petri nets have been used to model “active databases” in which activities are triggered when some event takes place, such as an integrity constraint being violated.

Applications of Petri nets

Modelling information systems:

> Workflow
> Hypertext (possible transitions)
> Dynamic aspects of OODB design

Roadmap

> Definition:
    —places, transitions, inputs, outputs
    —firing enabled transitions
> Modelling:
    —concurrency and synchronization
> Properties of nets:
    —liveness, boundedness
> **Implementing Petri net models:**
    —**centralized and decentralized schemes**

Implementing Petri nets

We can implement Petri net structures in either *centralized* or *decentralized* fashion:

Centralized:

> A single "net manager" monitors the current state of the net, and fires enabled transitions.

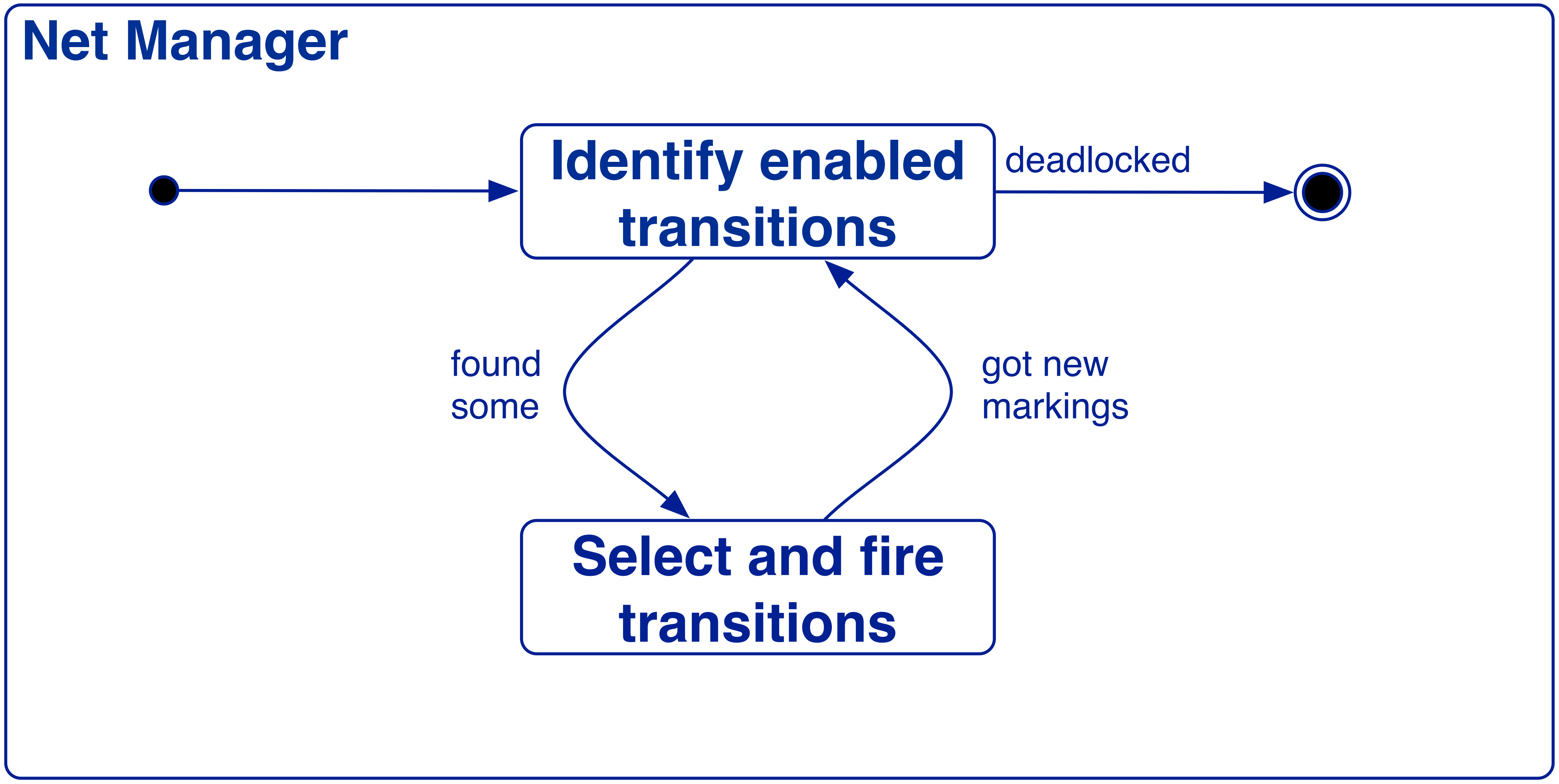
Decentralized:

> Transitions are processes, places are shared resources, and transitions compete to obtain tokens.

The centralized scheme just implements the formal definition of Petri nets, but has no “real” concurrency. The distributed version is truly concurrent, but the difficulty is in realizing the atomic nature of firing transitions competing for the same tokens.

Centralized schemes

In one possible centralized scheme, the Manager selects and fires enabled transitions.

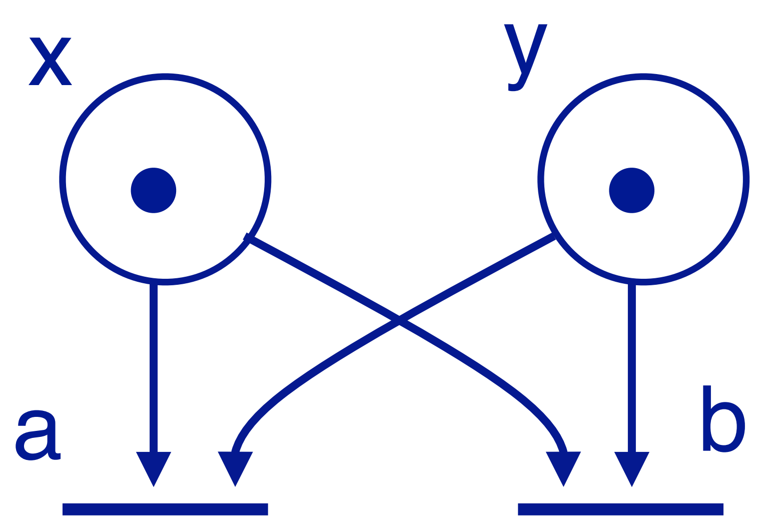


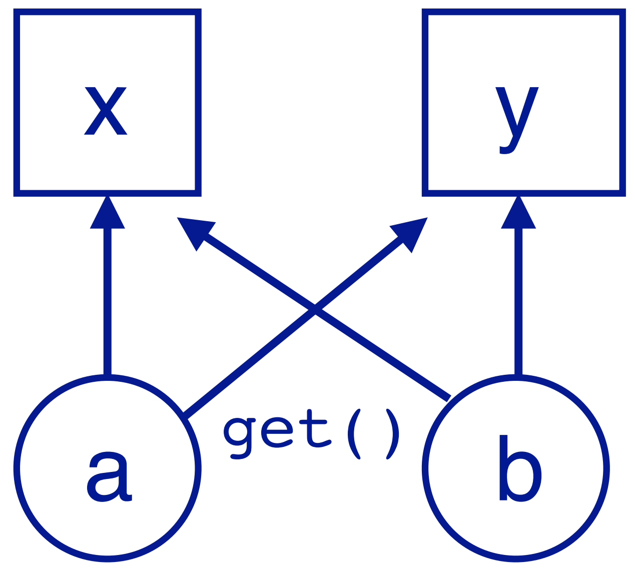
Concurrently enabled transitions can be fired in parallel.

Possible problems: starvation (dining phils); deadlock (no detection) ...

Decentralized schemes

In decentralized schemes transitions are processes and tokens are resources held by places:



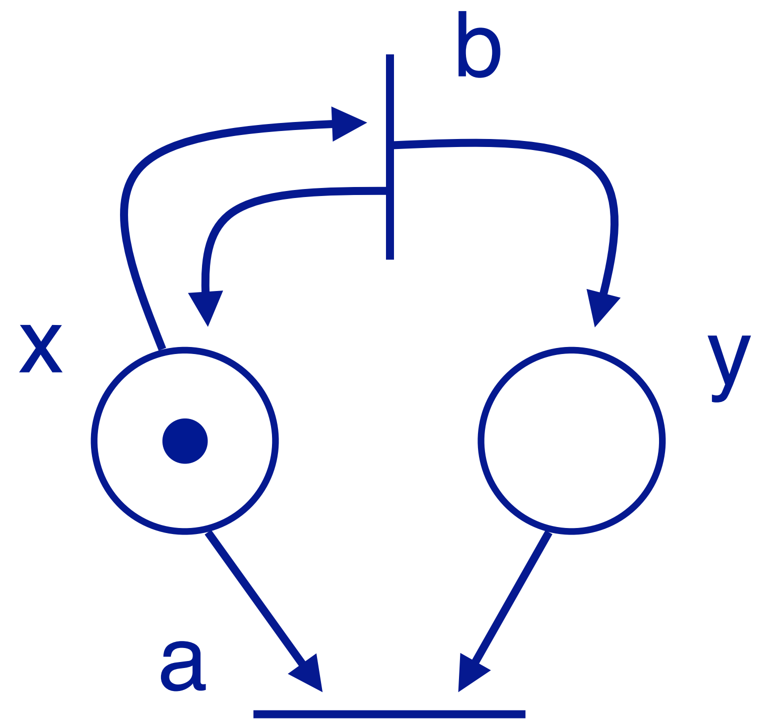


Transitions can be implemented as *thread-per-message gateways* so the same transition can be fired more than once if enough tokens are available.

The idea is that the availability of a token in a place will trigger a new thread in transitions that input that place.

Transactions

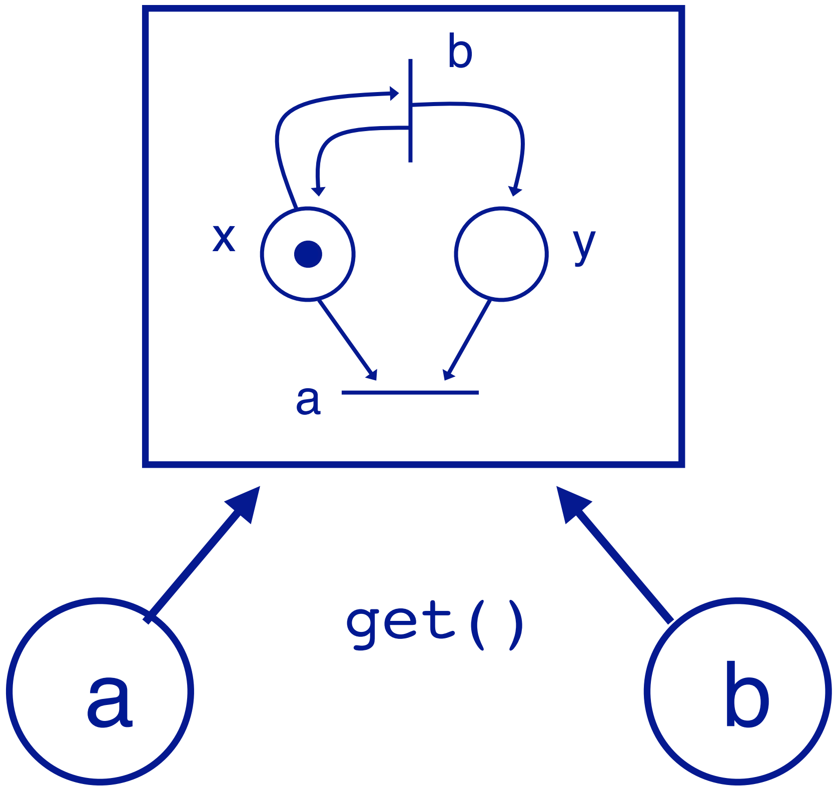
Transitions attempting to fire must grab their input tokens as an atomic transaction, or the net may deadlock even though there are enabled transitions!



If a and b are implemented by independent processes, and x and y by shared resources, this net can deadlock even though b is enabled if a (incorrectly) grabs x and waits for y.

Coordinated interaction

A simple solution is to treat the state of the entire net as a single, shared resource:



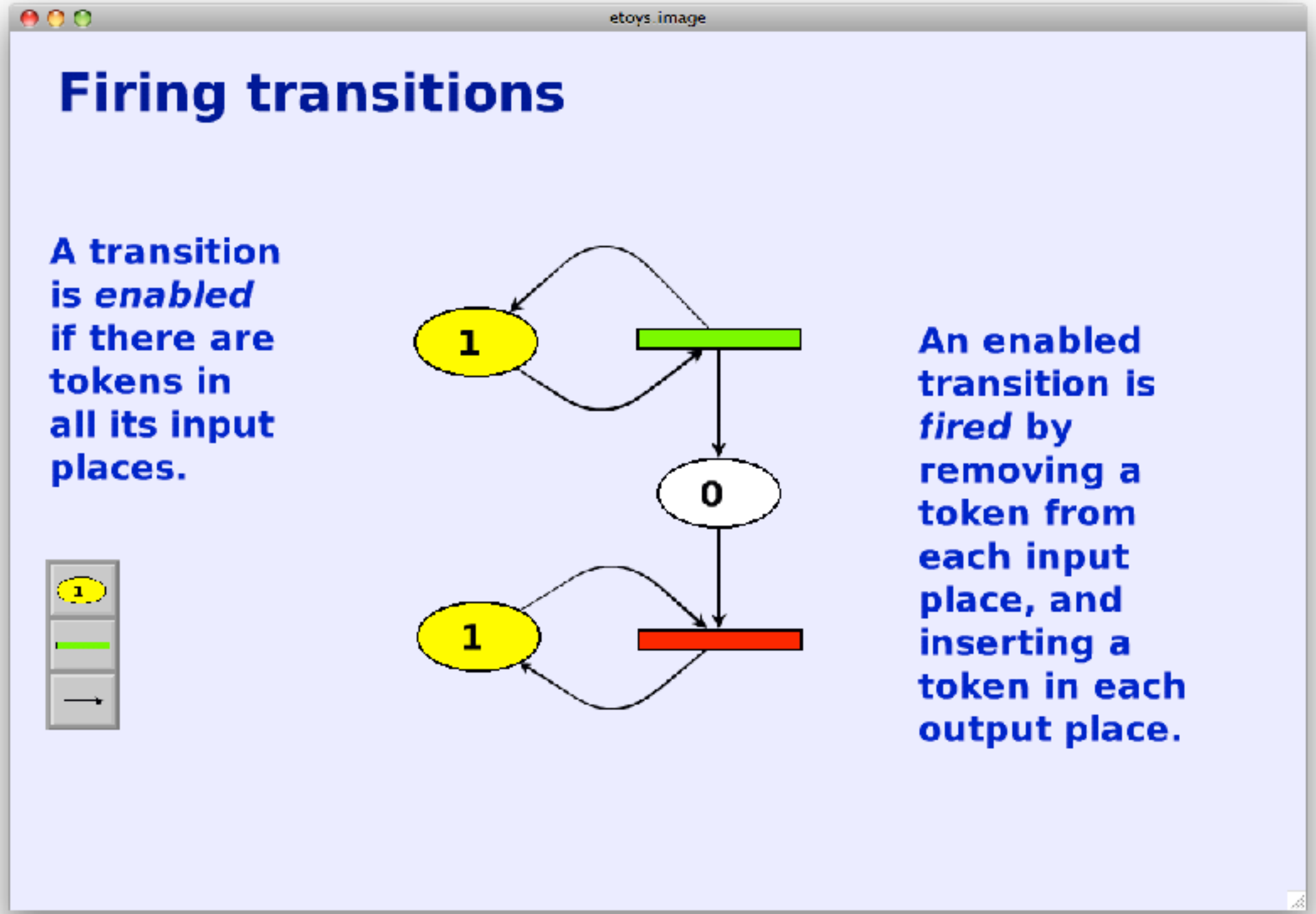
After a transition fires, it notifies waiting transitions.

This solution combines the centralized and distributed approaches. NB: We can represent the entire state as an object in a shared one-slot buffer!

Distributed scheme — idea: define equivalence classes via input relation x~y if {x,y} in I(a) for some a.

Can you think of other, simple approaches that maximize concurrency while avoiding deadlock?

Petit Petri — a Petri Net Editor built with Etoys



Firing transitions

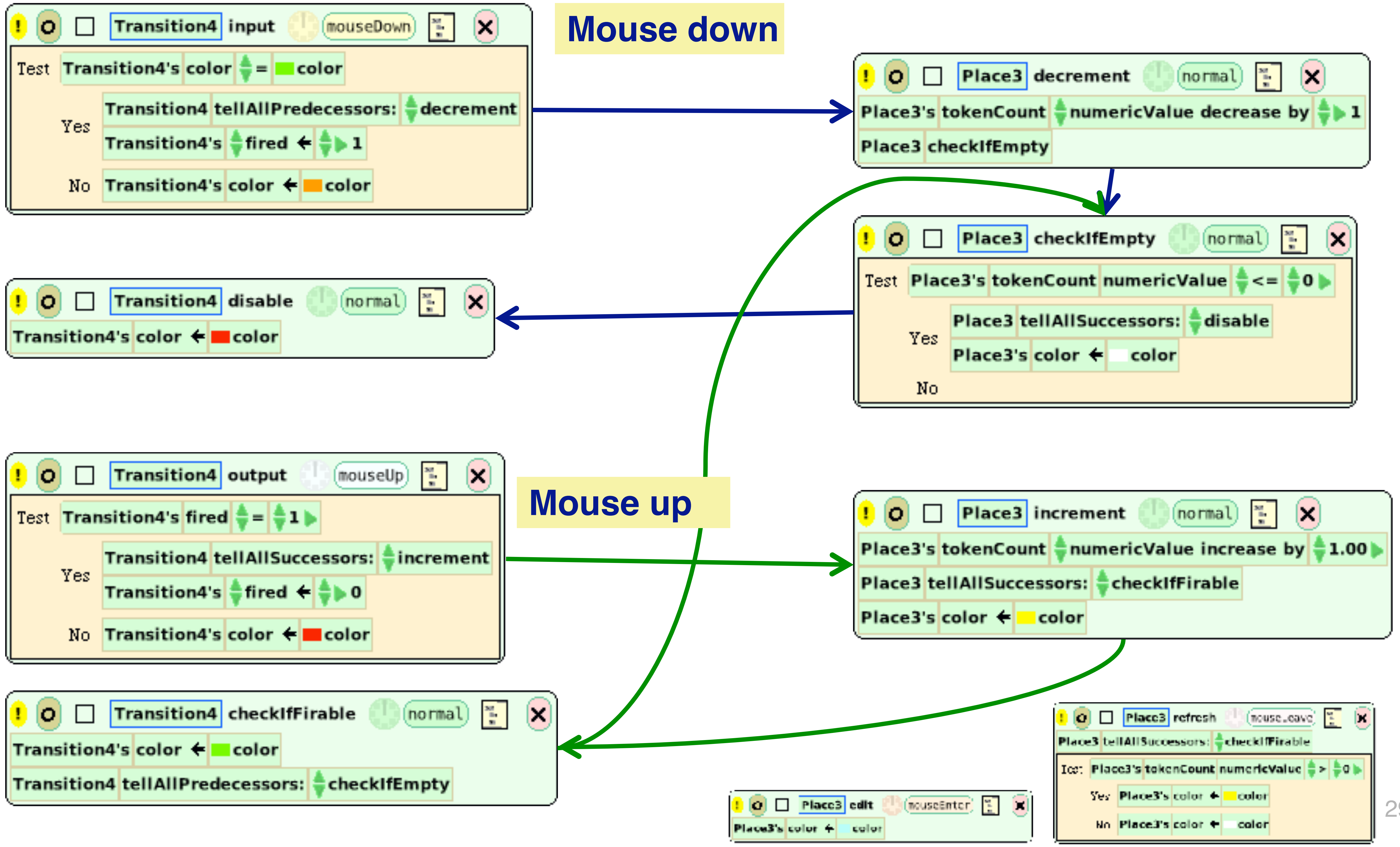
A transition is *enabled* if there are tokens in all its input places.

An enabled transition is *fired* by removing a token from each input place, and inserting a token in each output place.

Petit Petri is a Petri Net editor and simulator implemented in eToys.

See: http://scg.unibe.ch/download/petitpetri/

Etoys implementation



Mouse down

```
Transition4 input  mouseDown
Test  Transition4's color = 🟩 color
Yes   Transition4 tellAllPredecessors: decrement
      Transition4's fired ← 1
No    Transition4's color ← 🟧 color
```

```
Place3 decrement  normal
Place3's tokenCount numericValue decrease by 1
Place3 checkIfEmpty
```

```
Place3 checkIfEmpty  normal
Test  Place3's tokenCount numericValue <= 0
Yes   Place3 tellAllSuccessors: disable
      Place3's color ← ⬜ color
No
```

```
Transition4 disable  normal
Transition4's color ← 🟥 color
```

Mouse up

```
Transition4 output  mouseUp
Test  Transition4's fired = 1
Yes   Transition4 tellAllSuccessors: increment
      Transition4's fired ← 0
No    Transition4's color ← 🟥 color
```

```
Place3 increment  normal
Place3's tokenCount numericValue increase by 1.00
Place3 tellAllSuccessors: checkIfFirable
Place3's color ← 🟨 color
```

```
Transition4 checkIfFirable  normal
Transition4's color ← 🟩 color
Transition4 tellAllPredecessors: checkIfEmpty
```

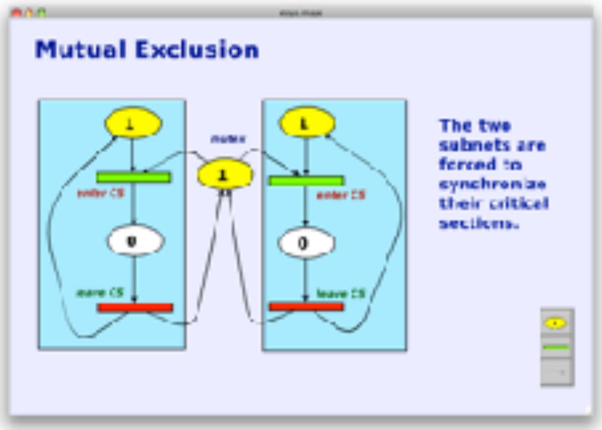
```
Place3 edit  mouseEnter
Place3's color ← color
```

```
Place3 refresh  mouseLeave
Place3 tellAllSuccessors: checkIfFirable
Test: Place3's tokenCount numericValue > 0
Yes   Place3's color ← 🟨 color
No    Place3's color ← ⬜ color
```

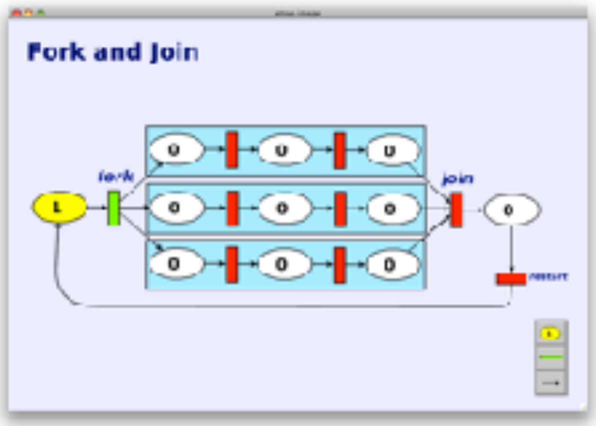
The entire implementation consists of 9 simple event driven scripts. After building a net, the user can simulate its execution by clicking on an enabled transition. *Mouse down* will instruct all input places to decrease their token count by one, and mouse up will instruct output places to increase their token count. When a place’s token count drops to zero, it tells its successor transitions to become disabled. When a place’s token count increases, it tells its successors to check if they are enabled (checkIfFirable optimistically sets itself to green, and then asks its predecessors to disable it if any of them are empty).

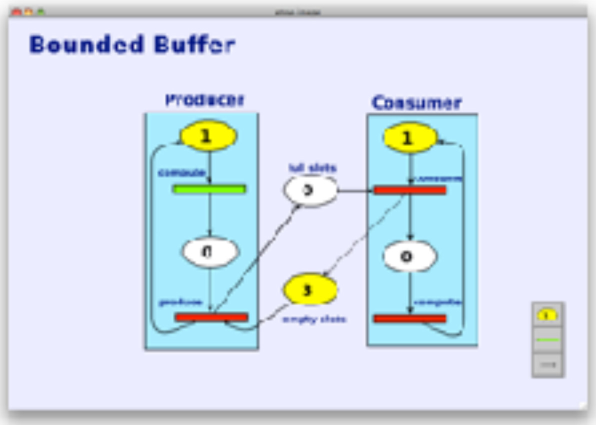
Mouse enter and *mouse leave* for a place will update its color, leaving it yellow if it is empty.

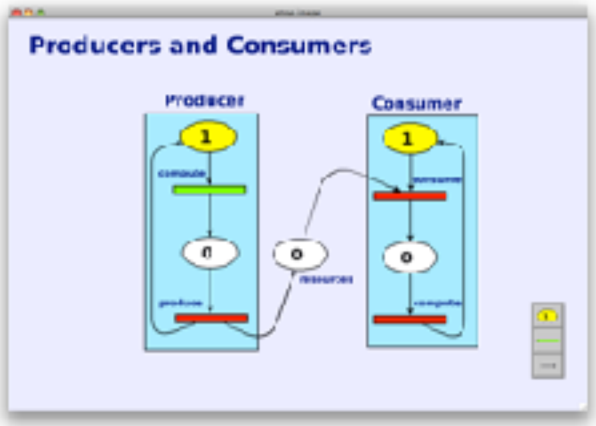
Examples

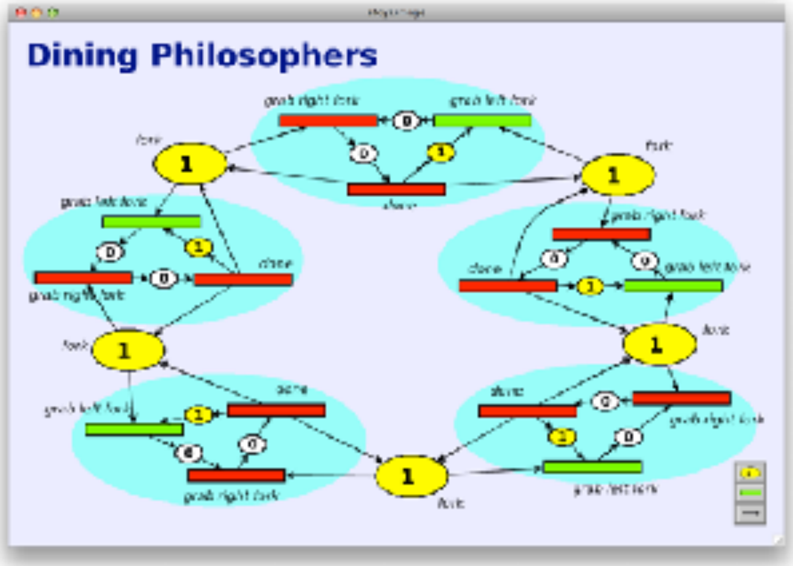

Mutual Exclusion

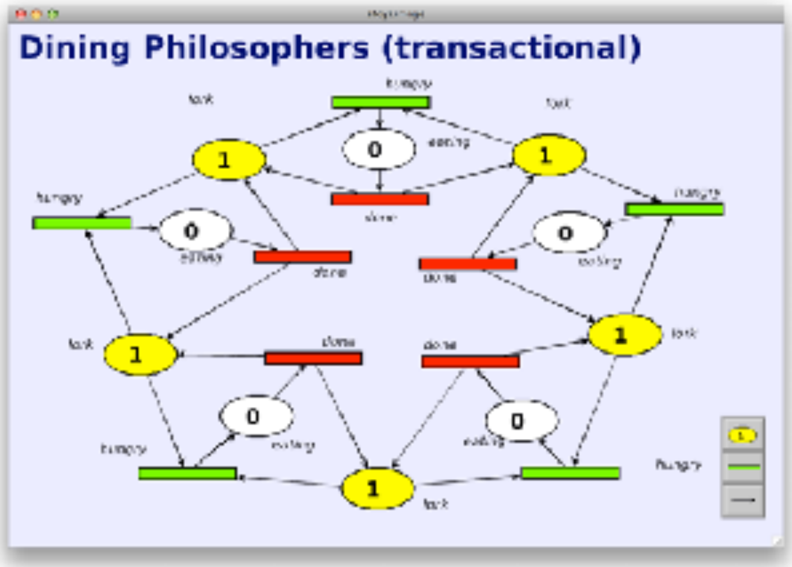
The two subnets are forced to synchronize their critical sections.

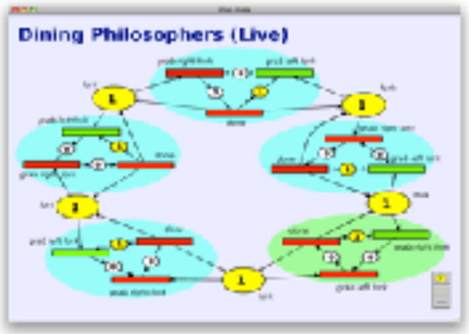

Fork and Join

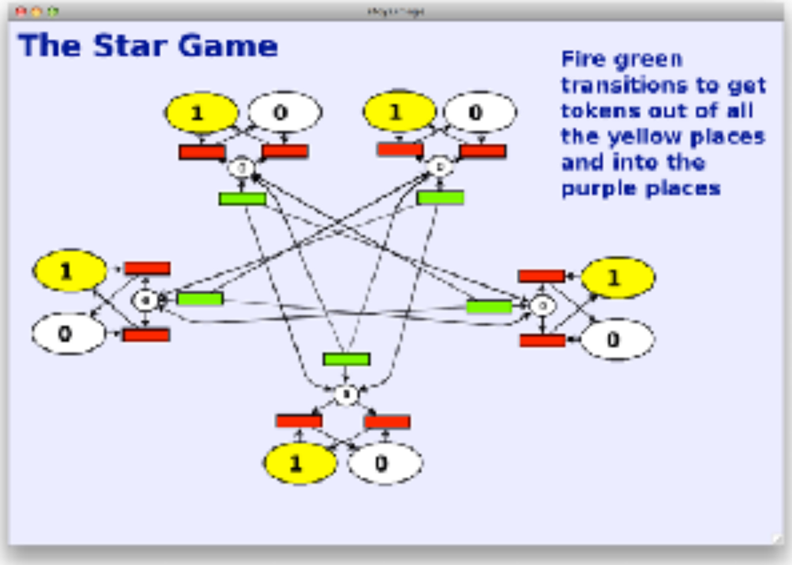

Bounded Buffer


Producers and Consumers


Dining Philosophers


Dining Philosophers (transactional)


Dining Philosophers (Live)


The Star Game

Fire green transitions to get tokens out of all the yellow places and into the purple places

Several examples from the lecture are implemented in Petit Petri, as well as a few different versions of Dining Philosophers and the “Star Game” (the goal is to move all five tokens from their current place to their neighbour).

What you should know!

> *How are Petri nets formally specified?*
> *How can nets model concurrency and synchronization?*
> *What is the “reachability set” of a net? How can you compute this set?*
> *What kinds of Petri nets can be modelled by finite state processes?*
> *How can a (bad) implementation of a Petri net deadlock even though there are enabled transitions?*
> *If you implement a Petri net model, why is it a good idea to realize transitions as “thread-per-message gateways”?*

Can you answer these questions?

- *What are some simple conditions for guaranteeing that a net is bounded?*
- *How would you model the Dining Philosophers problem as a Petri net? Is such a net bounded? Is it conservative? Live?*
- *What could you add to Petri nets to make them Turing-complete?*
- *What constraints could you put on a Petri net to make it fair?*

creative commons

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/