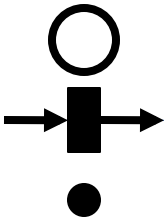
SPP (Synchro et Prog Parallèle)

Unit 7: Petri Nets

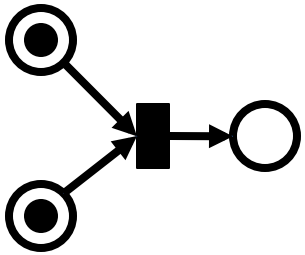
François Taïani

Petri Nets

- Invented by Carl Adam Petri (1939)
- Main elements
  - places
  - transitions
  - tokens



- Transitions consume and produce tokens
  - consume one or more tokens from one or more places
  - produces one or more tokens into one or more places

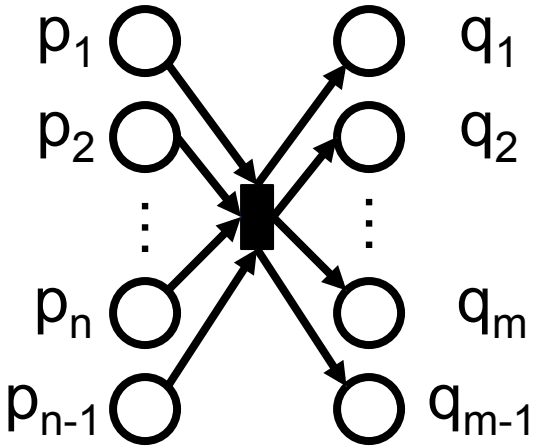


Tokens

- Tokens & places
  - ➔ One place can contain 0, 1, 2, … tokens
- Token & Transitions
  - ➔ Transitions need tokens to fire
  - ➔ Tokens need transitions to arrive in a place
- Meaning: Tokens
  - ➔ represent the dynamic part of a Petri net
  - ➔ model either resources (produced and consumed)
  - ➔ or the execution pointer of a program

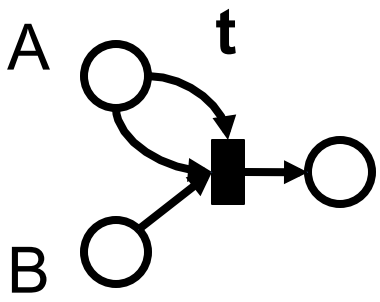
Transitions

- Transitions
  - ➔ link a set of *input places* to a set of *output places*
  - ➔ a transition needs to be *enabled* before it can fire
  - ➔ enabling = all input places contain one token
  - ➔ firing = input places lose 1 token, output places gain 1

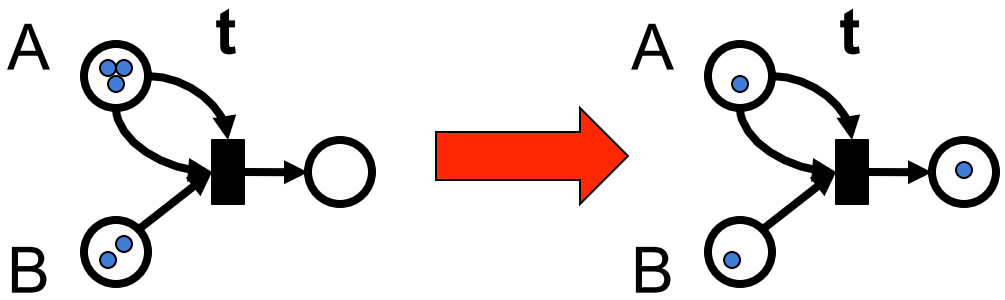


Multiple Edges

- Multiple edges possible between a place & a transition



➔ Transition **t** needs two tokens in **A** to fire

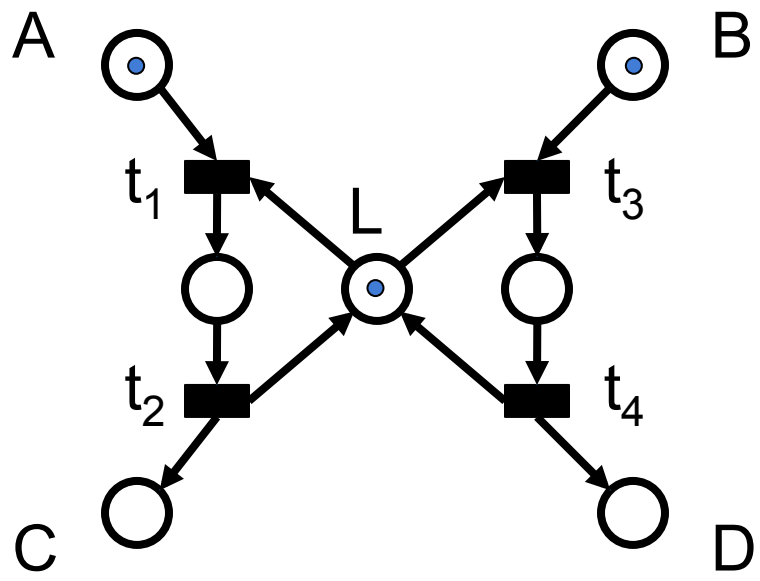


Executing a Petri Net

- While some transitions are enabled
  - ➔ select one of the enabled transitions
  - ➔ fire this transitions (e.g. update tokens)
- Vocabulary
  - ➔ distribution of tokens in places: “marking”
  - ➔ start marking = initial marking (M_0)

Small Exercise

- What are the possible executions of this Petri net?

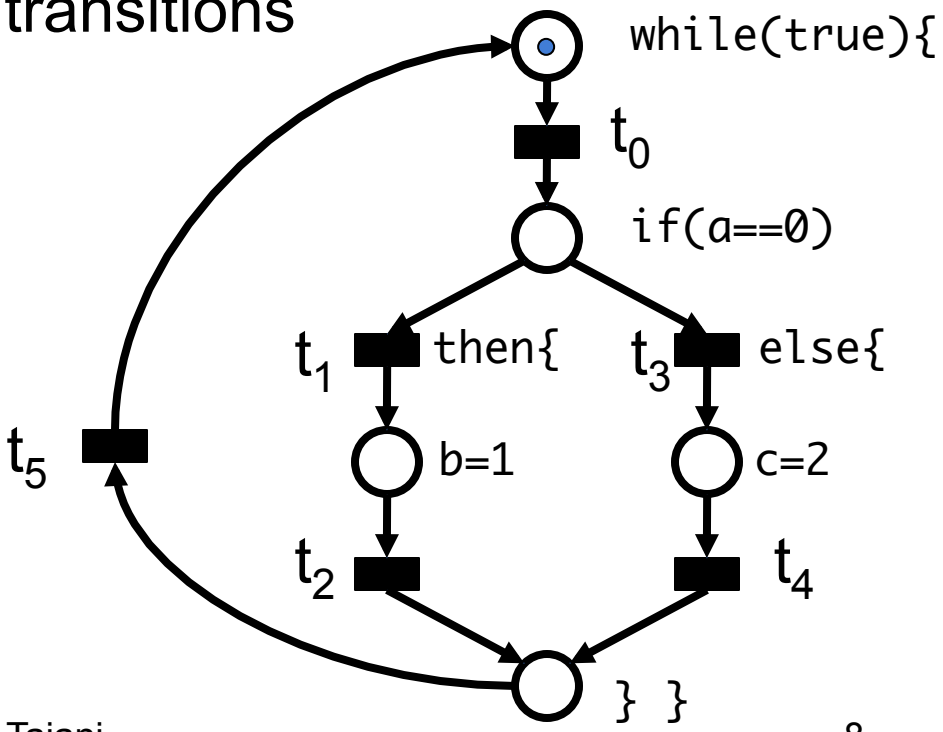


Link with Parallelism

Petri nets can be used to model flow of parallel code

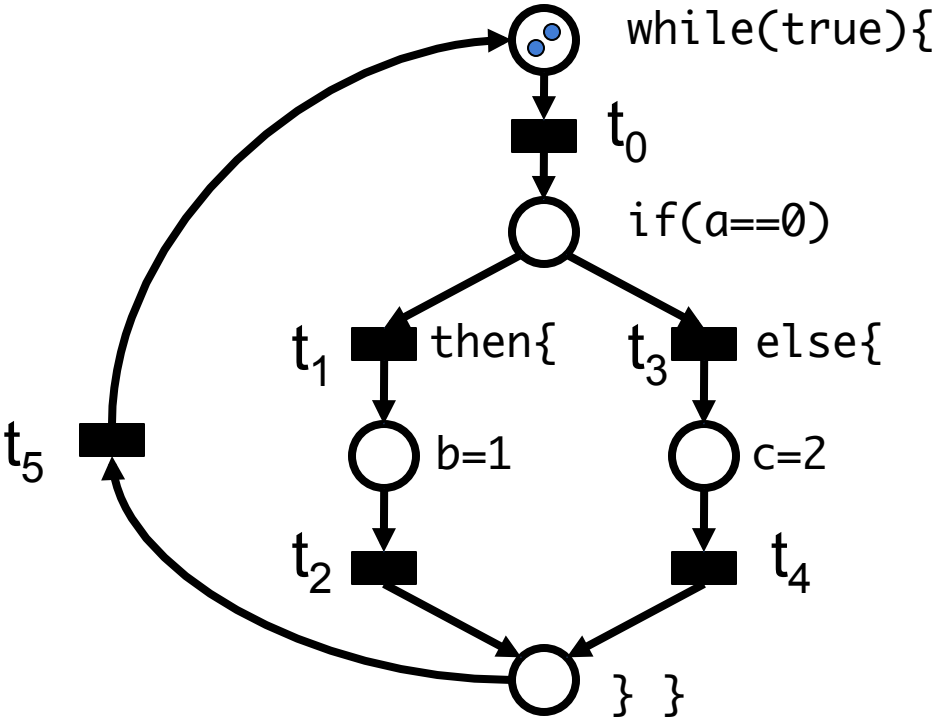
- Sequential code -> sequence of transitions
- Loops -> transition looping back
- Conditions -> alternative transitions

```
while (true) {
  if (a==0) then {
    b = 1
  } else {
    c = 2
  }
}
```



Modelling Multithreading

- Several tokens!
  - ➔ e.g. here for 2 threads



Note on previous example

- Note all aspects of program modelled
  - ➔ in program if condition is deterministic
  - ➔ in Petri net: random: one or the other might happen
- Consequence
  - ➔ PN runs = superset of real runs
  - ➔ Some Petri net runs might not be possible in real code
  - ➔ Important to keep in mind when doing analysis

Approximation of Modelling

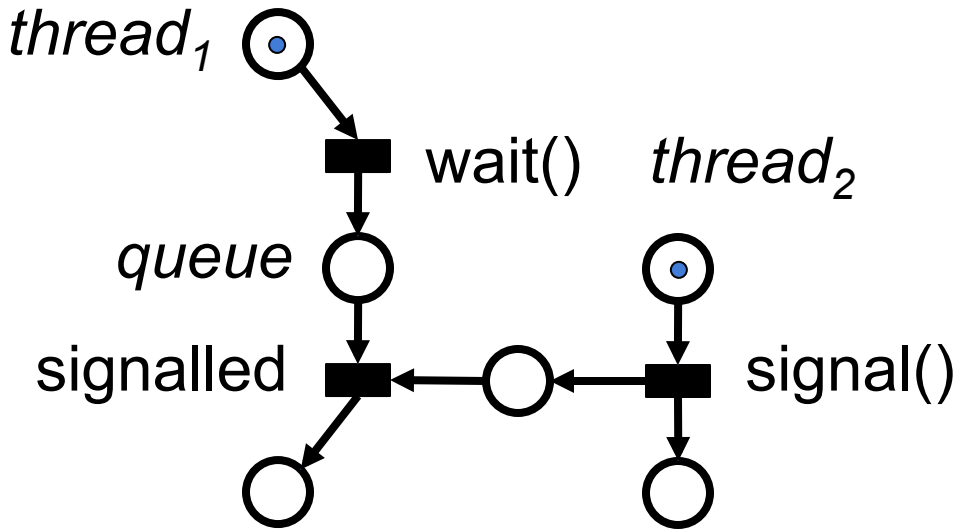
- Most often {code runs} $\subsetneq$ {PN runs}
- Implication: when {code runs} $\subsetneq$ {PN runs} then
  - ➔ all PN runs “safe” -> all application runs “safe”
  - ➔ some PN runs “unsafe” -> application might be unsafe
  - ➔ some PN runs block -> application might block
  - ➔ all PN runs lively -> ?
- (Note: Common issue in formal analysis)
- In cases when code runs = PN runs (mod congruence)
  - ➔ safety, un-safety, deadlocks, and liveliness translate

Modelling Resources

- Simply as a shared place with n tokens
  - ➔ n = number of resources
- a lock = one place with one token
  - ➔ lock: consuming transition going out from place
  - ➔ unlock: producing transition coming into place
- a semaphore = one place with n tokens
  - ➔ down: consuming transition going out from place
  - ➔ up: producing transition coming into place

Modelling Synchronisation

- Condition variables can be approximated
  - ➔ waiting queue = a place, waiting threads: tokens in place
  - ➔ “wait()” moving to queue
  - ➔ “signal()” enabling out transition



- **Quiz:** What are the problems with this model?

Problem with previous model

- If waiting queue is empty when signal() triggered
  - ➔ the PN model remembers the signal operation
  - ➔ a real condition variable would not
  - ➔ OK is can prove this never happens
  - ➔ otherwise PN extension needed (zero testing)
- Impossible to realise “notifyAll”
  - ➔ possible (but complex) with zero testing
- Fundamental reason
  - ➔ PN are not Turing Complete
  - ➔ (They are with zero-testing)

Exercise

- Ping-Pong example from TD
  - ➔ reminder: what assumptions are we making on locks?

```
shared lock l1, l2
init() { l2.lock() } // l2 initialised as locked

thread t1 is
  while(true) {
    l1.lock()
    println("ping”)
    l2.unlock()
  }
end
```

```
thread t2 is
  while(true) {
    l2.lock()
    println("pong")
    l1.unlock()
  }
end
```

- Exercise: Represent above program as a Petri Net
  - ➔ Is there a 1-1 mapping between your PN and the program?

Analysis Petri Nets

- Possible to enumerate all reachable markings
  - ➔ markings linked to each other through transitions
  - ➔ result = *reachability graph* (might be infinite)
- Depends on where PN starts: M_0
  - ➔ $R(M_0)$ = all markings reachable from M_0
- Allow us to answer reachability questions
  - ➔ are bad states reachable -> (possibly) unsafe code
  - ➔ no bad state reachable -> safe code
  - ➔ some desirable state always reachable -> lively code
  - ➔ some desirable state becomes unreachable: unlively code

Example with Ping and Pong

Bad state

- places for println("ping”) and println("ping”) both taken
  - ➔ violation of mutual exclusion, unsafe

Desirable states

- states always reachable where println("ping”) taken
  - ➔ ping thread never blocked, lively
- states always reachable where println(“pong”) taken
  - ➔ pong thread never blocked, lively

Special properties (for this example)

- ping and pong states alternate

Exercise

- Construct reachability graph of ping / pong example
  - ➔ Is the program safe?
  - ➔ Is the program lively?

Matrix interpretation

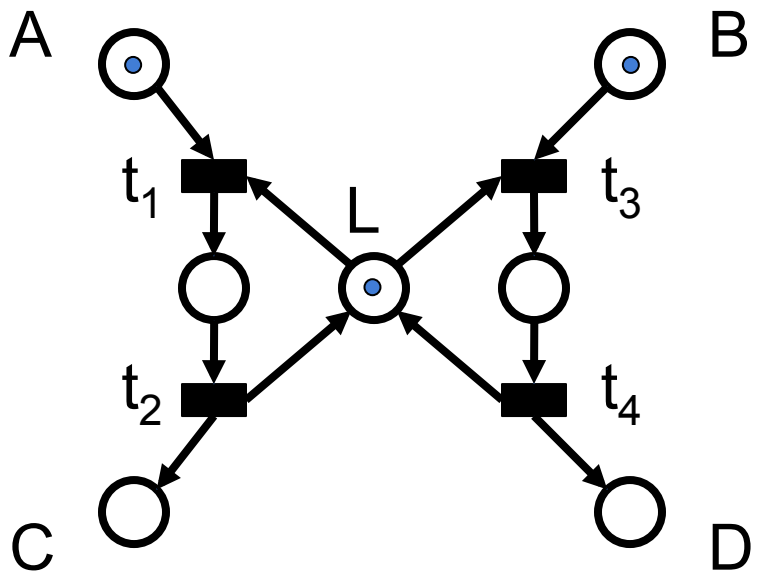
- A marking in a Petri net
  - ➔ a vector: one entry per place
- A transition: also a vector
  - ➔ -x in places where consuming
  - ➔ +y in places where producing
- A complete Petri net: Transition matrix M
  - ➔ m rows × n columns
  - ➔ each row: one transition
  - ➔ each column: change to place for each transition
    (note: some definition exchange rows and columns)

Computing with Matrix Rep

- If $v_0 = v(M_0)$ is initial vector
- $t_{i1}, t_{i2}, t_{i3}, \ldots t_{ik}$ sequence of transitions fired
  - ➔ $T = [nb\_t_1, nb\_t_2, nb\_t_3, \ldots]$ vector of transitions fired
  - ➔ nb_t_i = nb of times t_i appears in sequence
- Marking reached is
  - ➔ $v_k = v_0 + T \times M$
  - ➔ provided no place is ever negative during sequence (i.e. the sequence is actually possible)

Exercise

- Write down the transition matrix for the following PN
- Compute marking if t1, t2, t3, t4 are all fired



Extensions to PN

- We have seen one: zero-testing
- Others
  - ➔ inhibitor arcs (prevent transitions)
  - ➔ coloured PNs (tokens of different types)
  - ➔ hierarchical PNs, object PNs
  - ➔ timed PNs, stochastic PNs
- Software (many!)
  - ➔ http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html
- Link to similar formalisms:
  - ➔ Statecharts, UML state machines

Summary

- Petri Nets: Powerful formalism to model concurrency
  - ➔ sequential flow
  - ➔ tests (if), loops (while, for)
  - ➔ locks, semaphore, synchronisation (with approximation)
- Can be used for automatic analysis
  - ➔ reachability graph
  - ➔ link to linear algebra and in particular matrix algebra
- Can help detect:
  - ➔ unsafe states
  - ➔ deadlocks
  - ➔ lack of liveliness