Formal methods
(with particular attention to concurrent and real-time systems)

Instructor: Pierluigi San Pietro
Course given in English, alsu using webex and recording.

Shared by joint UIC-PdM master and normal PdM’s Computer Science and Engineering (plus Automation Eng.) curriculum
Course presentation
- Outline
- Organization

- Introduction and motivation (lot of motivation needed):
  - what are formal methods?
  - Objectives, problems, criticisms, …promises and reality, hopes and hypes, academia and industry, …
  - FMs in the context of sw (system!) life cycle
  - The international FM community (organizes conferences, etc)
- The basics of FMs:
  - Introduction to Model checking
  - Hoare's approach to program specification and proof
- Coping with concurrent -distributed- and real-time systems: problems and approaches
- A short survey of main FMs for concurrent -and real-time- systems: Petri nets, CSP, timed automata, (timed) temporal logic, …
- Case studies

Organization

Minor organizational details:
Classes: Wed, Friday. 14.15–16.15 PM
TA Dr. Andrea Manini

Interact!
Through e-mail for non-technical issues;
For technical issues, I do NOT use email:
We can arrange an online/presence meeting

Organization.2

Class lectures and exercises

Tool demos (with 60’ presentations by groups of students)

Homework

- Individual or (much better!) team work
- developing a case study
- use of tools

Exam: Either written exam or Homework or Tool Demo

Homework assigned beginning of May, to be completed end of June.

- Working in group of 2-3 people is strongly suggested
- **UIC students must do the homework**

Tool demos usually available in April

- There are not enough tools (suitable for a good demo) for everybody…
- Working in groups of 3 people is mandatory.

A check on your background

Model checking

Temporal Logic

Theoretical Computer Science

- Automata theory, formal languages, computability, complexity, basics of compiler design

Mathematical logic

NB: these are not prerequisites!

Teaching material

- *Reference* books
  - **Baier/Katoen; Principles of Model Checking, MIT press, 2007**
  - Mandrioli/Ghezzi, Theoretical foundations …, J.Wiley
  - Furia, Mandrioli, Morzenti, Rossi; “Modeling Time in Computing” Springer, 2012
    - Shorter: (same authors) “Modeling Time in Computing: A Taxonomy and a Comparative Survey”, *ACM Computing Surveys* (*CSUR*) Volume 42 , Issue 2 , February 2010, 59 pages.
- Transparencies (.pdf) available onWeBeep
- Other scientific papers and reports (to be specified; possibly electronically available)
- (Public domain) tools

What are Formal methods?

- *Exploiting mathematical formalisms in the analysis and synthesis of-computer- systems*
- Common in traditional engineering, where mathematics is often hidden in software tools
- What is new/different in computer science/ SW engineering?
- Traditional mathematics (e.g. calculus) is not well suited (lack of continuity, lack of linearity, …)
- Boolean algebra and other "basic discrete formalisms" are good only for low level hardware design;
- In "normal" computer system design the programming language is the only -if any!- formalized medium:
  - Most design documentation is "semiformal" -whatever such a term means …-
- We want rigorous methods, with software tool support

The problems with *in*formal methods

- Lack of precision (see next slide):
  - ambiguous definitions/specifications
  - erroneous interpretations
    - user/producer
    - specifier/designer
  - shaky verification (the well known problems with testing)
-
- Unreliability
  - lack of safety/security: what if the program in the next slide were part of a critical system?
  - economic loss
- Lack of generality/reusability/portability
-
- Poor quality (Even MS realized that …)

Example of C Program

What should the following program evaluate to?

```
int main(void) {
      int x = 0;
      return (x = 1) + (x = 2);
}
```

According to the C “standard”, it is undefined

GCC4, MSVC: it returns 4

GCC3, ICC, Clang: it returns 3

By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc ”prove” it returns 4

Why?

```
int main(void) {
      int x = 0;
      return (x = 1) + (x = 2);
}
```

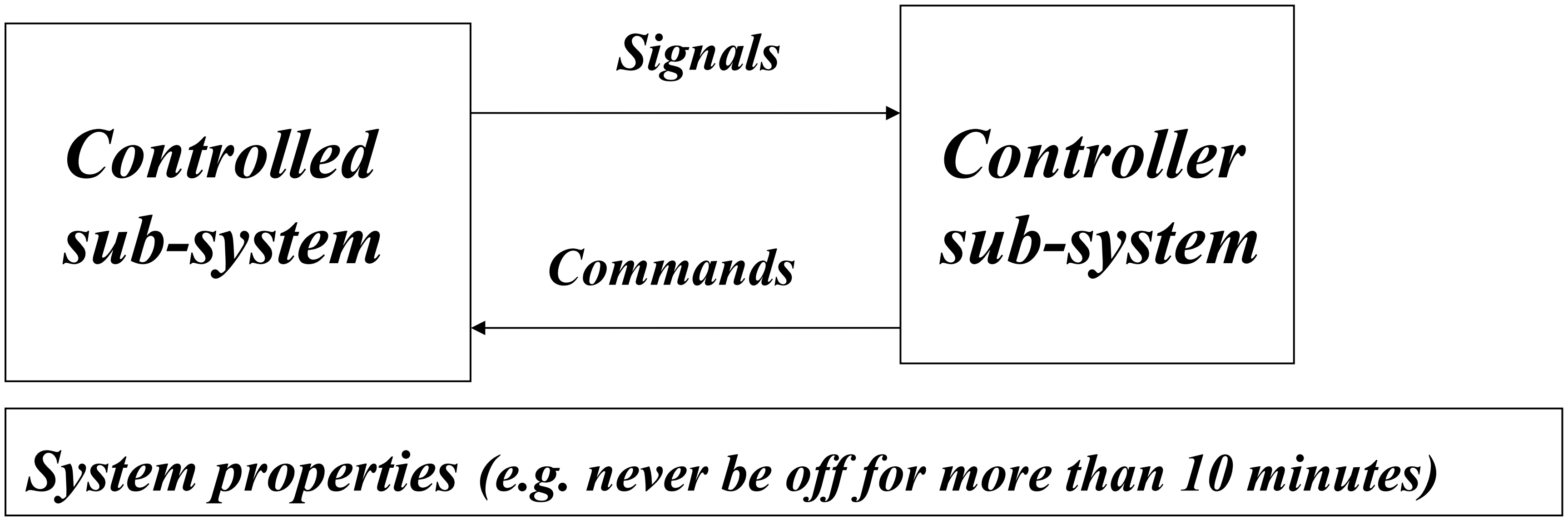
GGC4 optimizes compilation, reordering execution order, as long as it does not affect «well-defined» programs
int x = 0;
x = 1;
x = 2;
return x + x;

But above program was not well-defined. This would work for the program:

```
       int x = 0; int y = 0;
      return (x = 1) + (y = 2);
```

Formal methods: the ideal picture

- Everything is formalized: a mathematical model of everything is built.
- Every reasoning is based on mathematical analysis, supported by tools
- What is "everything"?
  - The system to be built
  - The system to be controlled
  - The (non-computer) to-be-controlled system / (computer-based) controller
  - The wished/feared properties (requirements)
  - …
  - The "context-dependent concept of *environment*"



- --------->
- "Everything" is mathematically certified
- Mathematical reasoning can be (partially) automatized
- ---->
- Increased reliability, ….
- Increased quality

FMs in the context of (SW or system?) life cycle

- All *systems* -including but not exclusively!- SW systems have their own life cycle
  - feasibility study,
  - requirements analysis and specification
  - design
  - verification,
  - …
  - maintenance
- their development exploits suitable *methods* (UML, OO analysis and design, agile, extreme, …)
- not all methods are exclusive … in one way or another they can be *integrated*
-

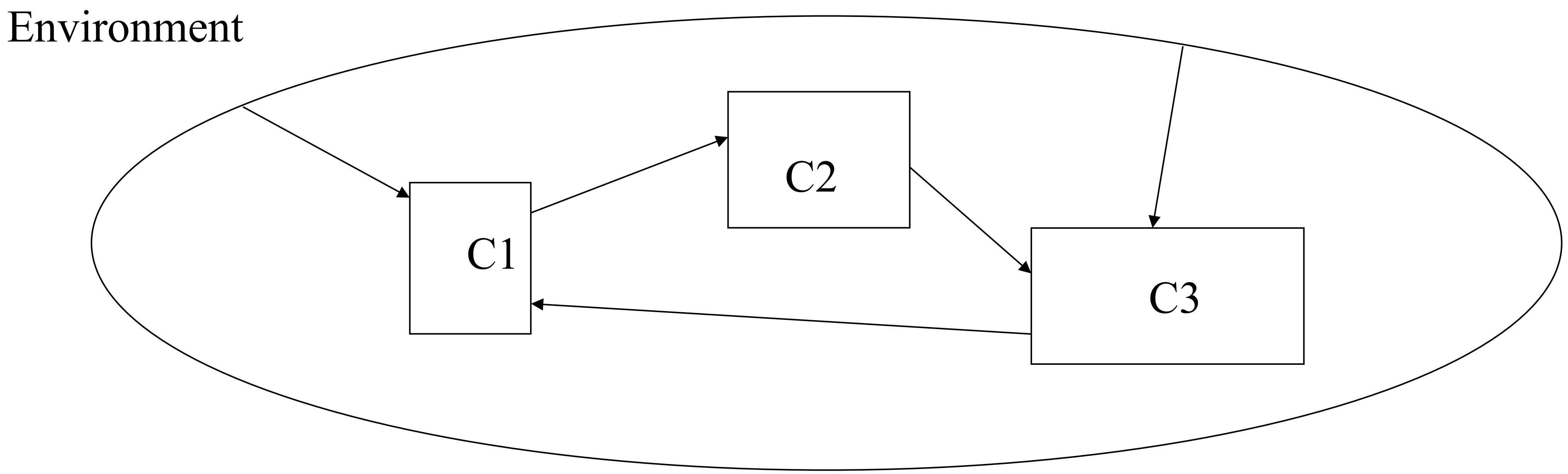
Our path towards specifying and proving (critical, real-time) systems:

- A classical method for the analysis of sequential programs
- Moving to concurrent and time-dependent systems
  - Operational formalism (Transition Systems, Timed Automata)
  - Logic formalisms (temporal logic: LTL, CTL, …)
- The basics of model checking
- Application and case studies
- Homework to apply the theoretical foundations using state-of-the-art tools.

For organizational reasons the “ideal path” must be slightly reshaped:

- A classical method for the analysis of sequential programs will be postponed
  - Hoare’s method for sequential programs
  - Method is based on pre and post conditions
  - A program is seen as a function from input to output

- What does it change when moving from a sequential to a concurrent/parallel system?
- Externally:
  - Is it still adequate to formalize a “problem” as a function to be computed, or a string to be translated, or accepted by some device?
    - In some cases yes: e.g., we wish to exploit parallelism to compute -faster- the inverse of a matrix, …
    - In general however, the picture is rather different:



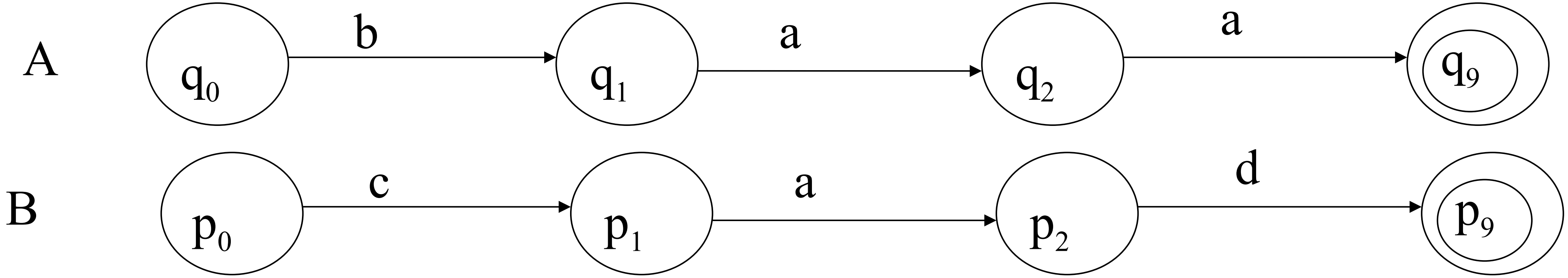
Is the analogy of the (Turing machine) tape(s) still valid?

The network era requires new models with **interactions** instead of **algorithms**

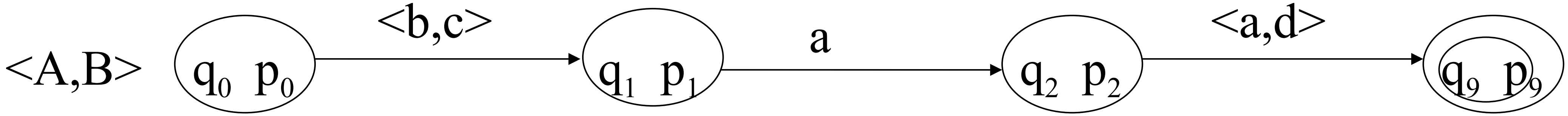
- Furthermore, for such systems, talking about <input data-computation-output data>, or begin-end of computation, is often inadequate: most computations are “never ending”
- We could -and will- resort to the notion of infinite words and related elaboration (a full theory of formal languages on infinite strings has been developed);
- But still we should deal with the *interleaving* of signals/data flowing through different “channels”: how can we describe and manage different sequencings in such signals? Should we consider all possible interleavings? Is this feasible/useful?
- Think e.g. to the following cases:
  - The system is a collection of PCs with independent users performing independent jobs
  - The system is a collection of processors managing signals coming from a plant (sensors) and human operators
- We can easily imagine that a further order of complexity will arise when *temporal relations* among such signals will become a critical issue (e.g. reacting to alarms)

Afew preliminary and basic steps

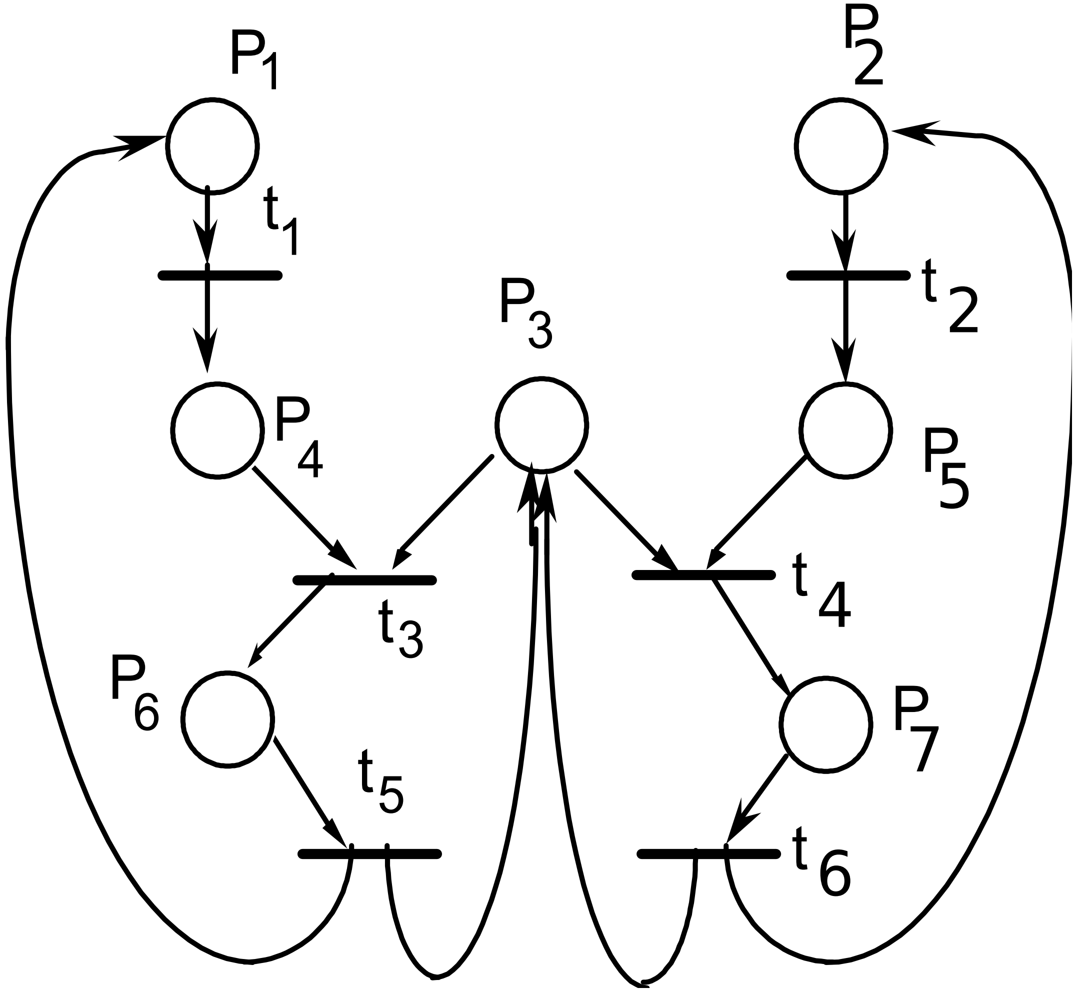
- Let us first look at the issue from an operational point of view:
- A *system* is simply a collection of *abstract machines* -often called *processes*.
- In some cases we can easily and naturally build a *global state* as the composition of *local states* of single machines:



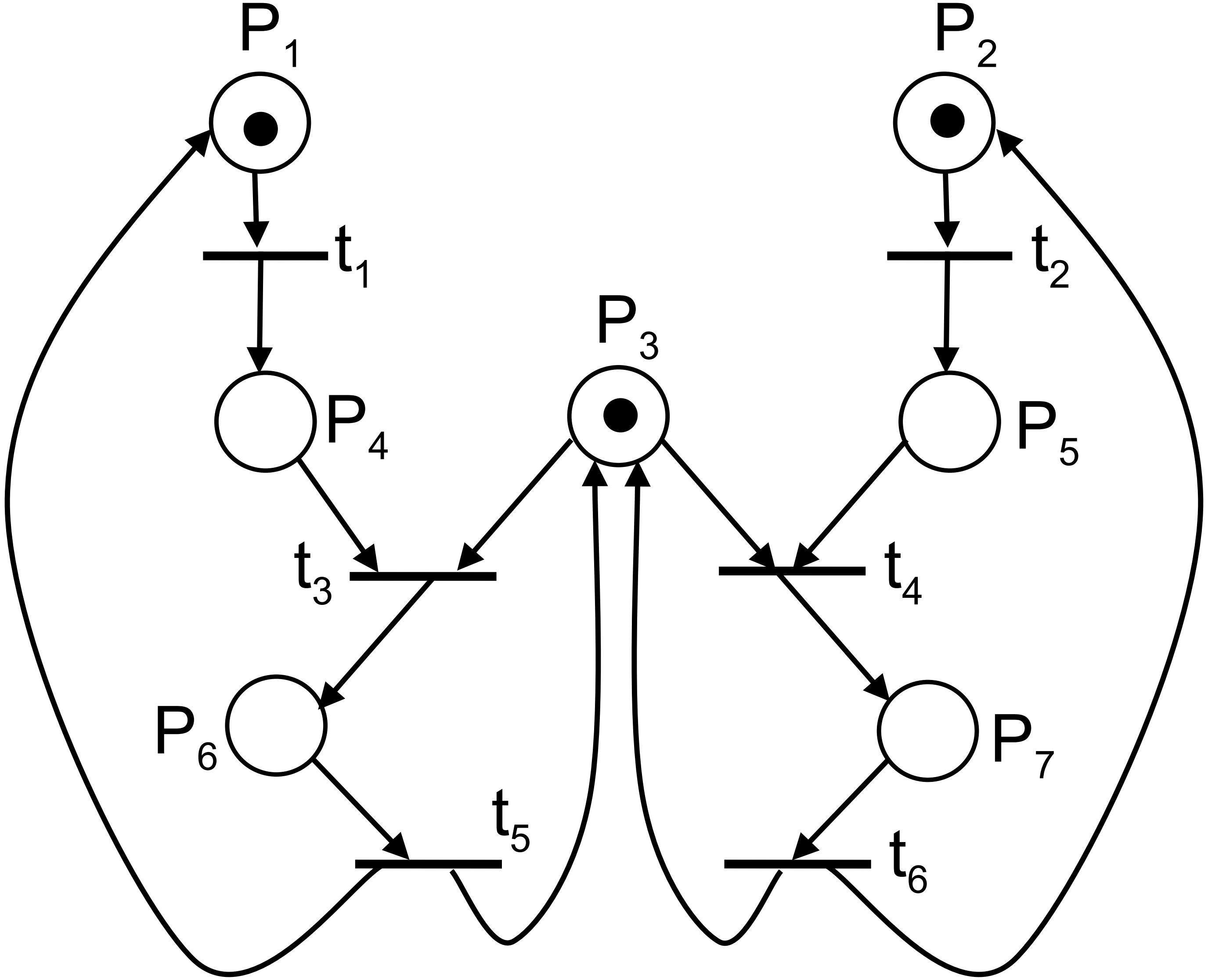
Let us recall the construction of “intersection machines”



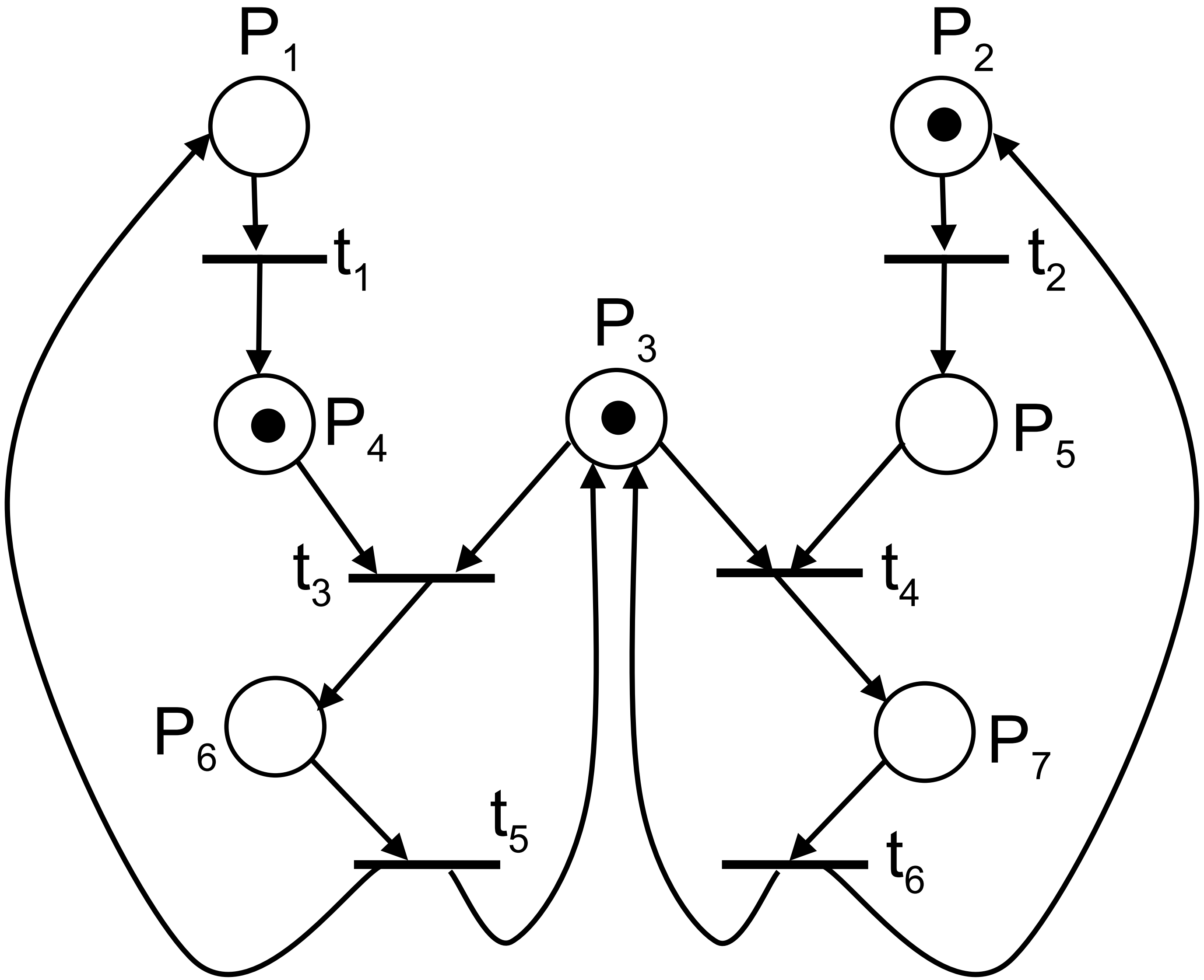
- When dealing with concurrent systems it is often inconvenient -or even impossible- to “view” a global state which evolves synchronously throughout every component
- Each process evolves autonomously and only occasionally synchronizes with other processes
- Typical *asynchronous* abstract machines are *Petri nets*.
- (already known?)



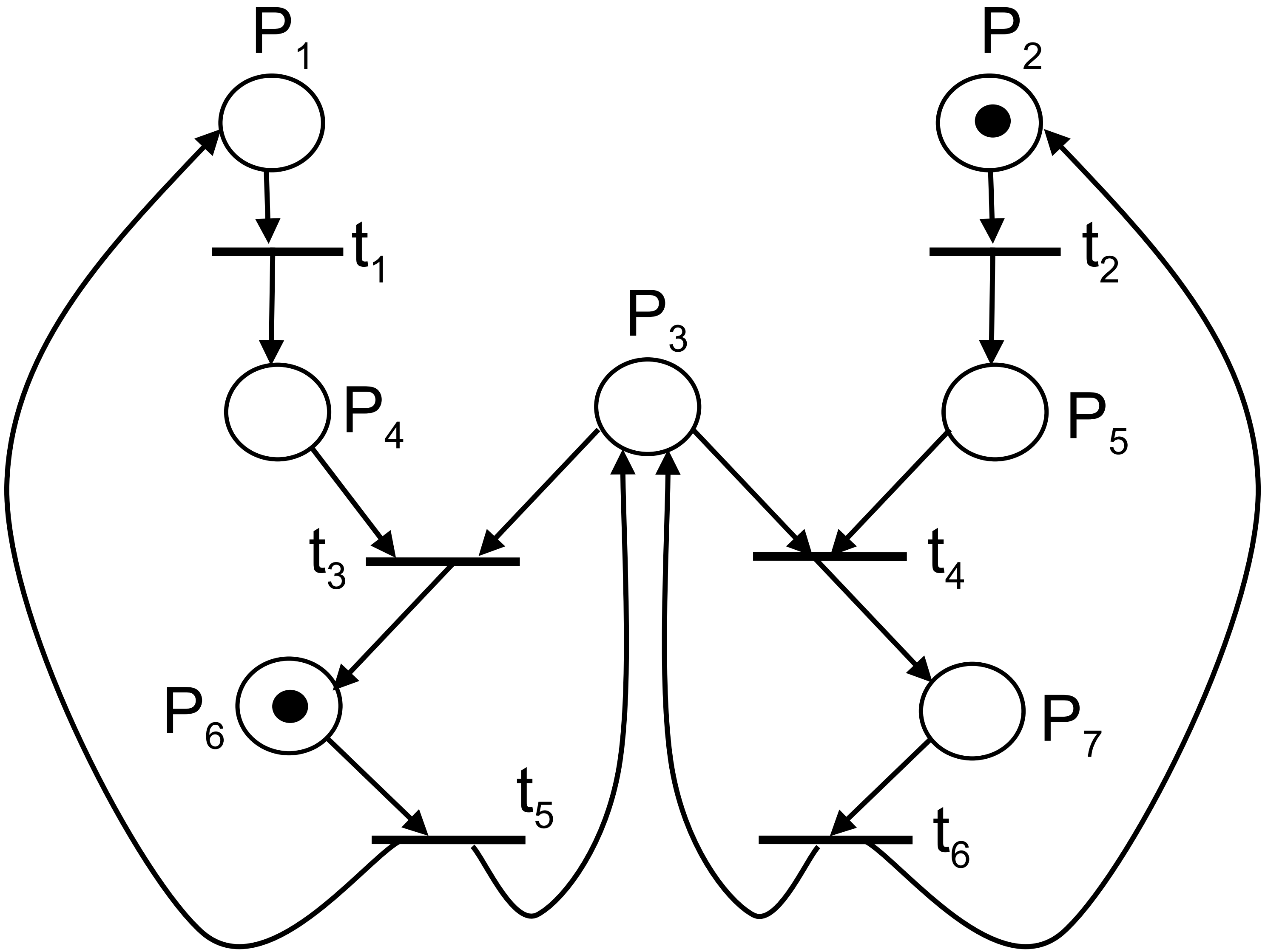
PNs marking and firing sequences:



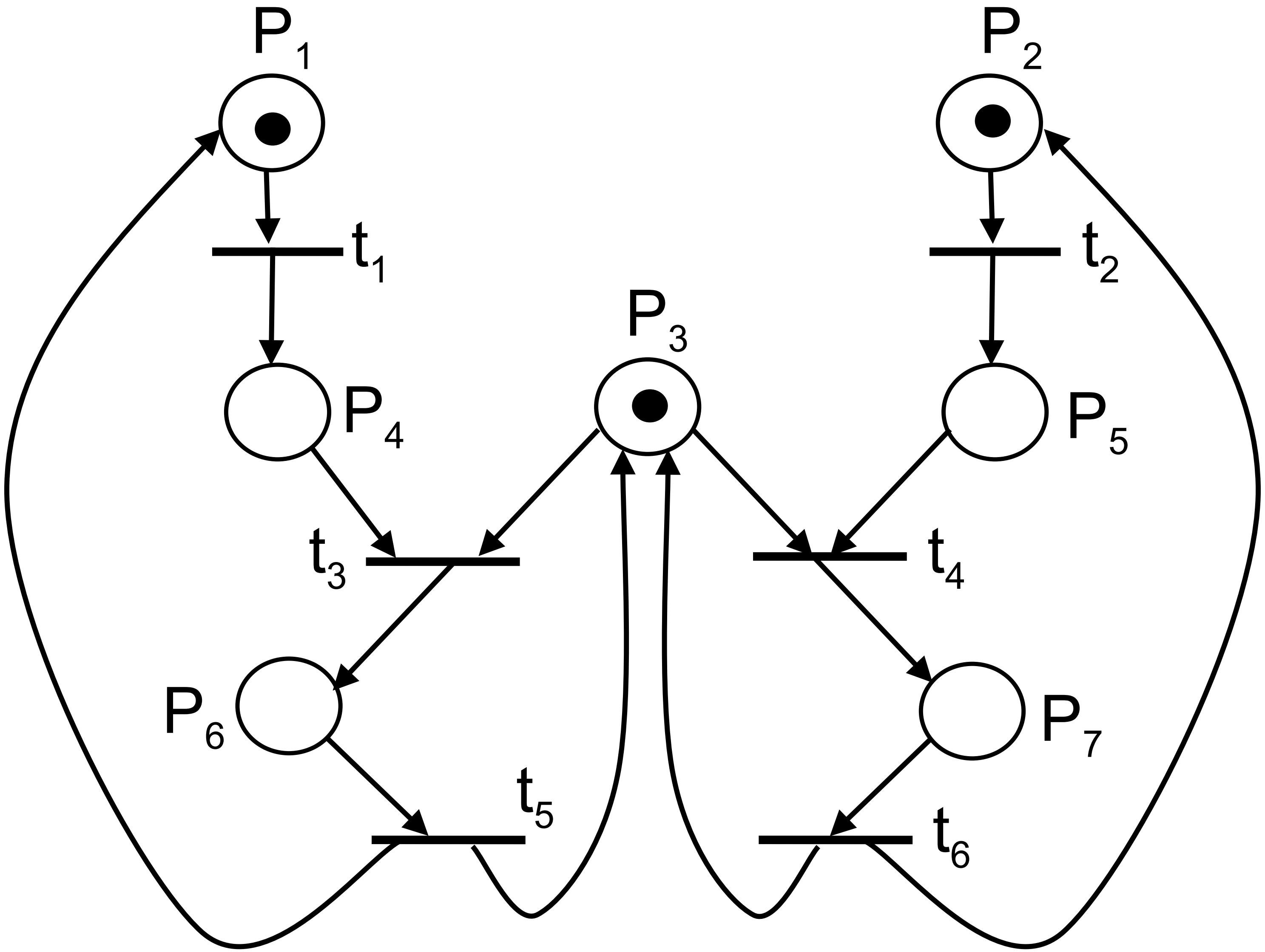
t_1 fires …



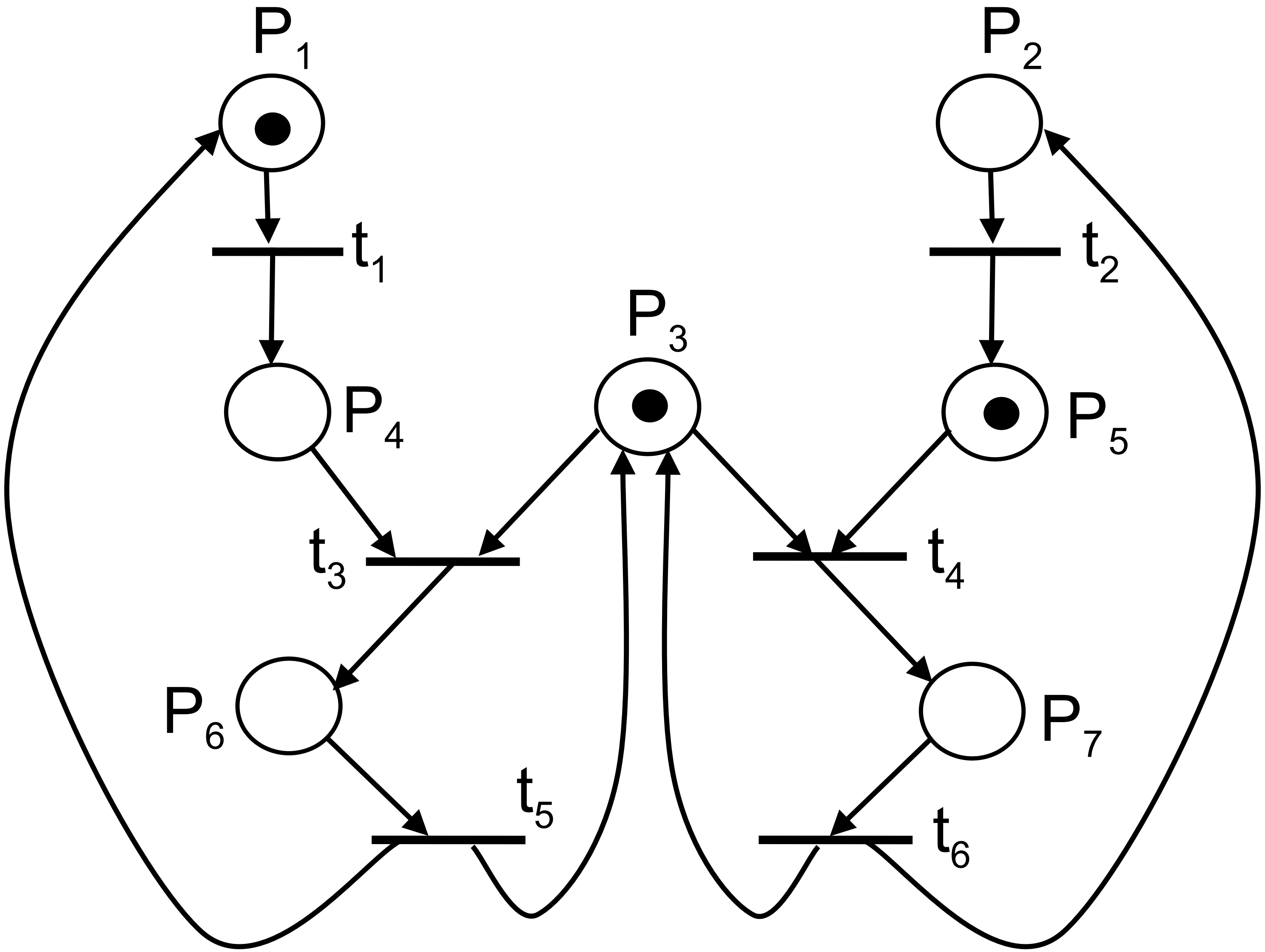
… t_3 fires …



… t_5 fires …

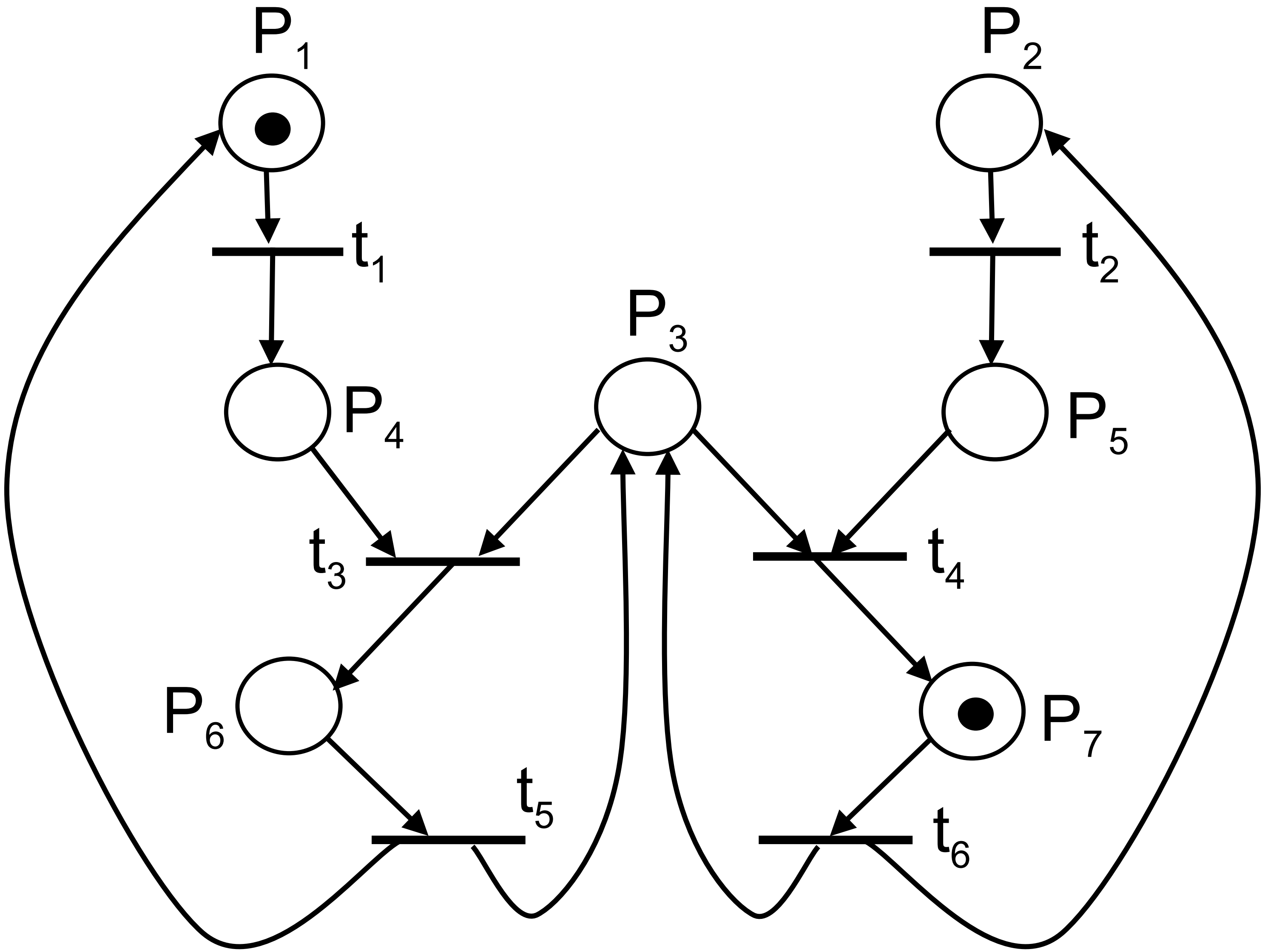


…



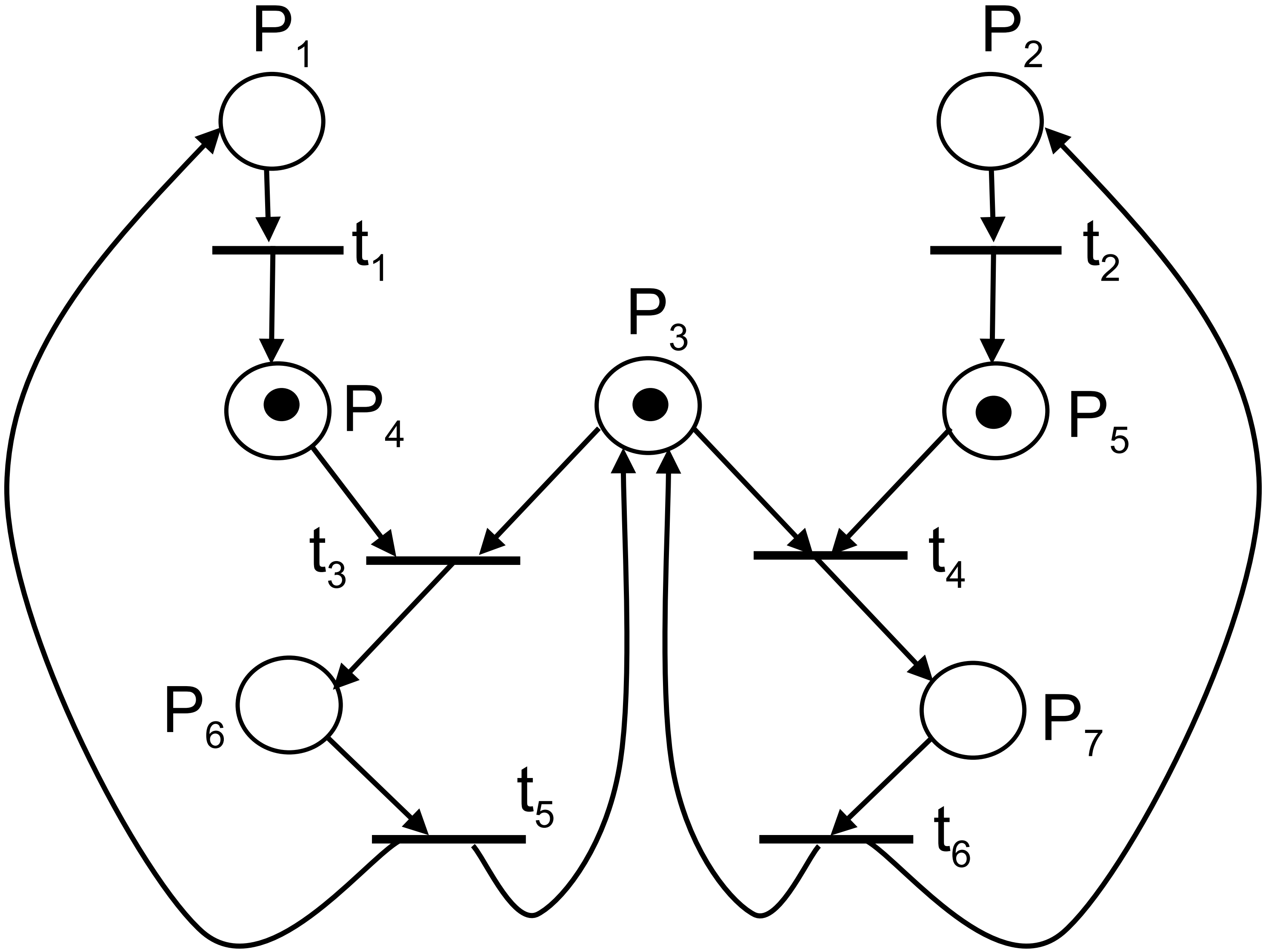
P_1
P_2
t_1
t_2
P_3
P_4
P_5
t_3
t_4
P_6
P_7
t_5
t_6

....

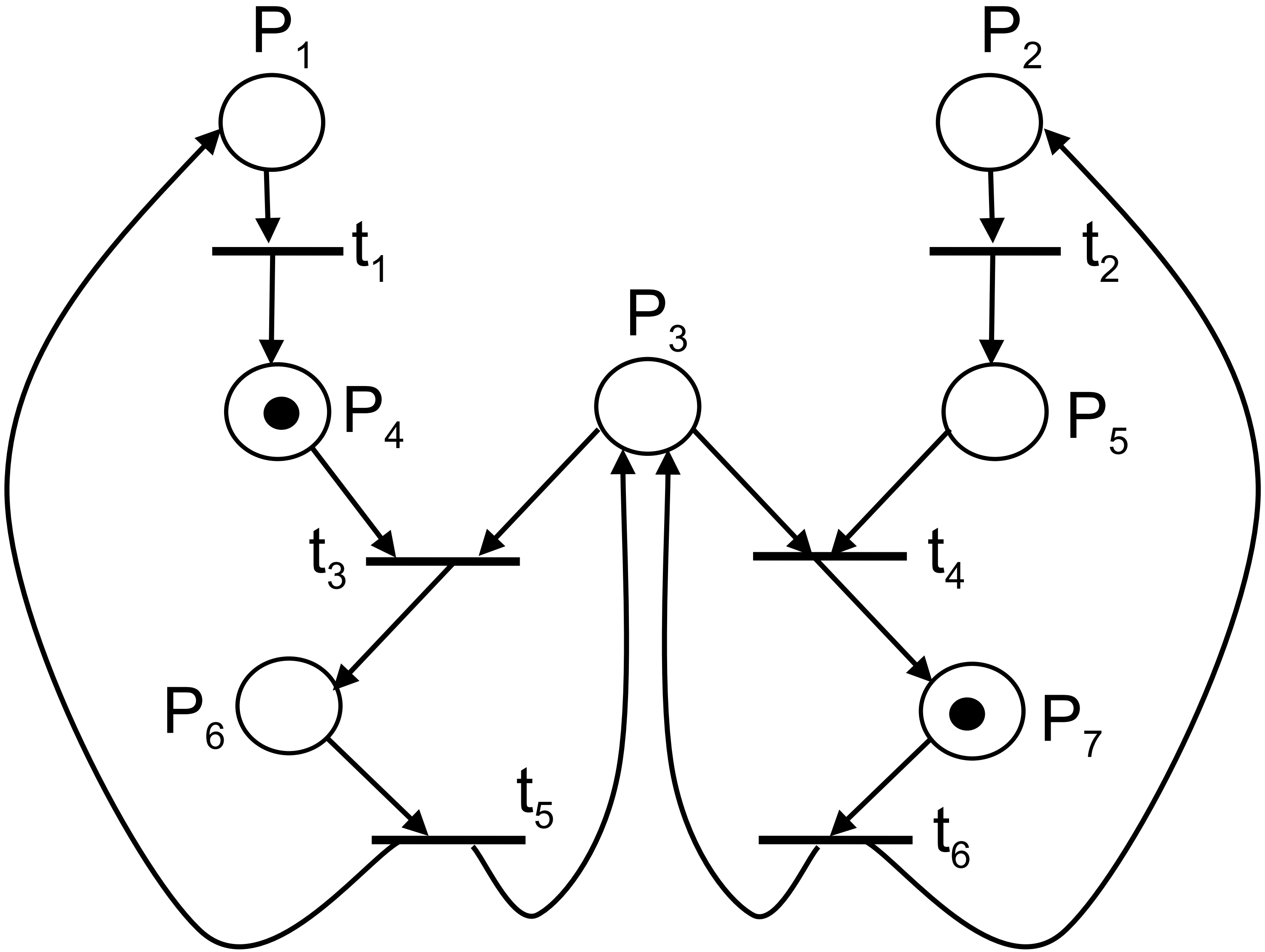


P1
P2
t1
t2
P3
P4
P5
t3
t4
P6
P7
t5
t6

t_1 and t_2 fire simultaneously …

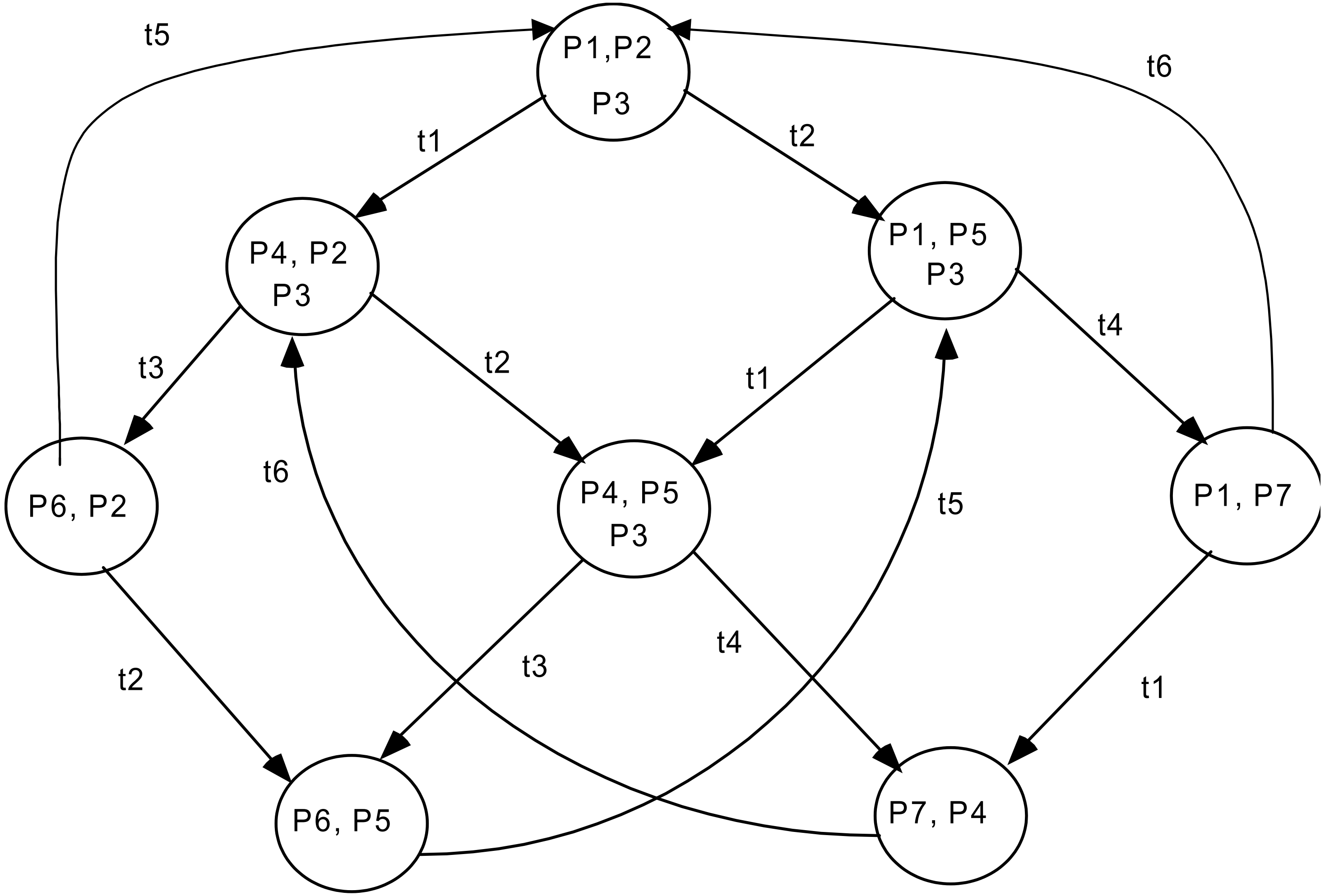


....

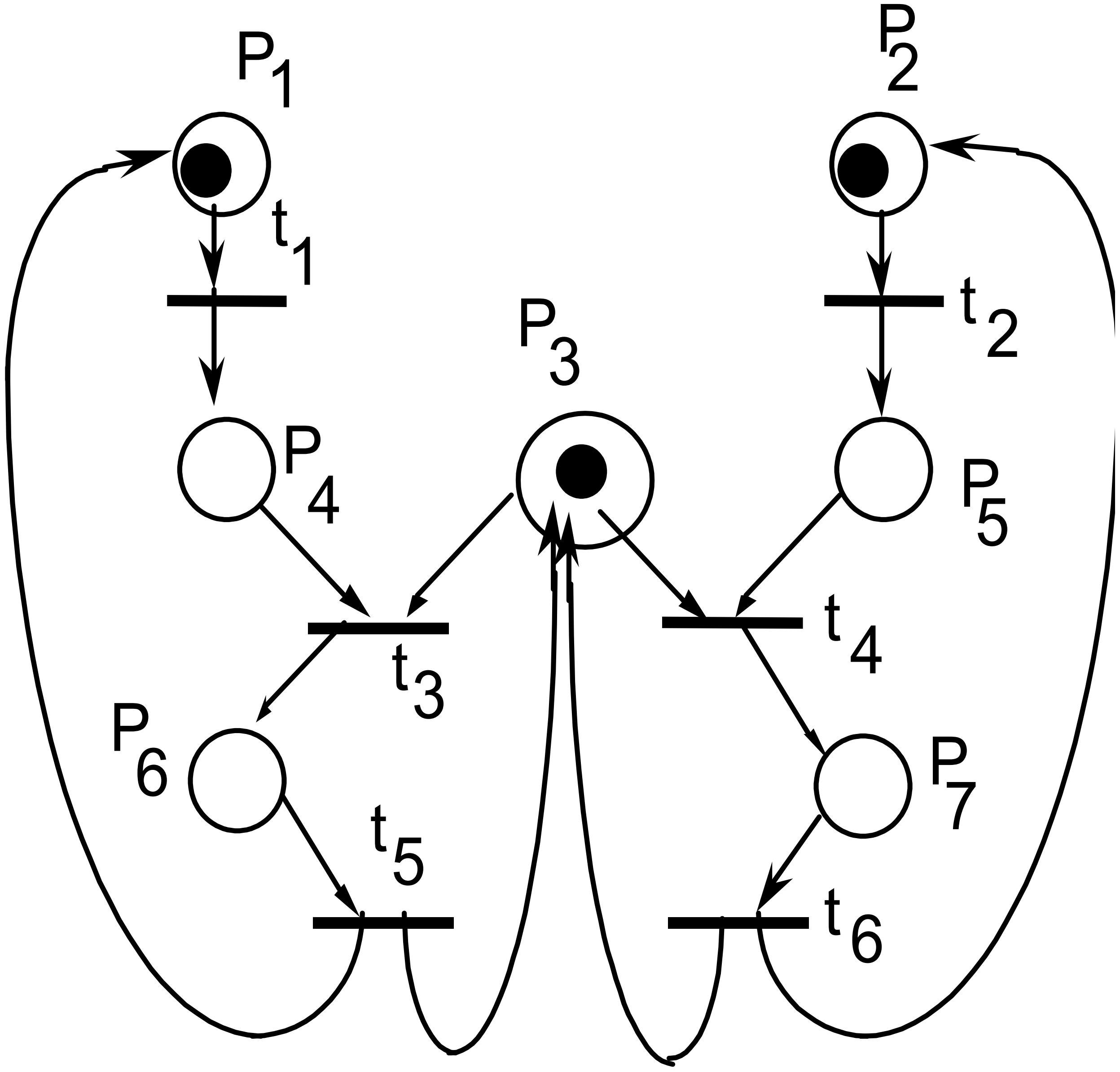


P1
P2
t1
t2
P3
P4
P5
t3
t4
P6
P7
t5
t6

In some cases we can formally reduce one machine to another one: A FSM description



… of the more understandable Petri net



- The FSM formalizes the *interleaving semantics* of the PN
- However synchronous and asynchronous abstract machines have some "philosophical" differences (the "religious war" of asynchronous people against synchronous ones …)
- In some cases however, it is simply impossible to talk about a global system state:
  - in distributed systems the various components are physically located in different locations
  - they communicate through signals flowing in channels
  - when components evolution proceeds at speeds comparable with the light speed it is meaningless saying that "at a given time t component C1 is in state Qa and component C2 in state Qb so that the global system is in state <Qa, Qb>

When time comes into play ...

- … things become even more critical
- Side remark:
- Unlike traditional engineering (see dynamical system theory) computer science tends to abstract away from time and to deal with it in a fairly separate fashion (complexity/performance evaluation)
- This may work fine in several cases but certainly not for real-time systems, whose correctness, by definition, does depend on time behavior.
- We must deal with:
  - time occurrence -and order- of events;
  - duration of actions and states
  - in critical cases such time values may depend on data values and conversely
  - (e.g. if the reaction to an alarm depends on some computation, whose duration, in turn, depends on some input data)

Time has been “added” to formalisms in several ways

- In operational formalisms:
  - durations have been added to transitions (whether FSM transitions or PN transitions, or …)
  - there is a large literature of timed PNs (time added to transitions, to places, to arcs, ….)
  - often such “added times” are constant values; in more sophisticated cases they may depend on data
  - Some approaches consider time as “yet another system variable, t”. Time elapsing is formalized by assignments to such a tick variable.
- In descriptive formalisms:
  - Pure temporal logic leaves time implicit and with no metrics: time is infinitely “elastic”: typical operators such as **eventually**, **until** predicate the occurrence and order of events/states, but do not measure how long these things take
  - Several “metric temporal logics” have been defined (bounded **until**, bounded **eventually**, ….)