

Real-Time Systems = Sisteme de timp-real

1. Introducere Introduction (RTS)

Occupational skills: system engineering and computer designer, software engineer, IT project manager, software programmer, IT system designer, software developer for process control, system analyst

Knowledge/understanding:

- real-time requirements
- real-time concepts

Practical skills:

- specification of real-time applications
- design of real-time applications
- implementation of real-time applications
- verification of real-time applications

Abilities::

- operating under UNIX operation systems
- implementation using concurrent Java and Real-time Java

Prerequisites

Basic programming
Software engineering
Discrete event systems

Formal or non-formal approach?

Cours Grade:

The grade for the midterm exam (M);

The grade for the final exam (F);

Laboratory verification (L)

$$N=0.33M+0.33F+ 0.33L$$

Condition to obtain the credits: **$L \geq 5; M \geq 5; F \geq 5$**

Cuprins C1 –

1. Introducere -

- a. Paradigme - Paradigms
- b. Definiții de bază - Basic definitions
- c. Control de T-R - Real-time control
 - i. Tipuri de sisteme de control de T-R - Types of R-T control systems
 - 1. Sisteme înglobate - Embedded systems
 - 2. Sisteme de control industriale - Industrial control systems
 - ii. Caracteristicile sistemelor de control înglobate - Embedded control characteristics
- d. Caracteristicile STR - RTS characteristics
- e. Parametrii temporali - Temporal parameters
 - i. Temporizări nedeterministe - Non-deterministic timing
 - ii. Evenimente - Events
 - iii. Variante de implementare a controlului - Types of control implementations

Contents C1

Introduction

Paradigms

Basic definitions

Real-time control

Types of R-T control systems

Embedded systems

Industrial control systems

Embedded control characteristics

RTS characteristics

Temporal parameters

Non-deterministic timing

Events

Types of control implementations

a. Paradigme - Paradigms

Sisteme de prelucrare -

Processing systems:

- **batch systems** – secvențiale
- **on-line systems** – cu legătură directă
- **real-time systems** – sisteme de timp-real

Alt punct de vedere -

Another point of view:

- sisteme transformaționale - **transformational systems** → transformational programs – they take some input values and process them according to their functionality (functional programming)
- sisteme reactive - **reactive systems** → reactive programs – they react to some events

Programarea secvențială -

Sequential programming

Programare paralelă -

Parallel programming:

- Concurrent tasks
- Threads of execution
- Distributed programming

Concurrent programming

Pervasive computing

The idea that technology is moving beyond the personal computer to everyday devices with embedded technology and connectivity as computing devices become progressively smaller and more powerful. Also called ***ubiquitous computing***, pervasive computing is the result of computer technology advancing at exponential speeds -- a trend toward all man-made and some natural products having hardware and software.

Pervasive computing goes beyond the realm of personal computers: it is the idea that almost any device, can be imbedded with chips to connect the device to an infinite network of other devices. The goal of pervasive computing, which combines current network technologies with wireless computing, voice recognition, Internet capability and artificial intelligence, is to create an environment where the connectivity of devices is embedded in such a way that the connectivity is unobtrusive and always available.

Pervasive computing is the next generation computing environments with information & communication technology everywhere, for everyone, at all times.

Information and communication technology will be an integrated part of our environments: from toys, milk cartons and desktops to cars, factories and whole city areas - with integrated processors, sensors, and actuators connected via high-speed networks and combined with new visualization devices ranging from projections directly into the eye to large panorama displays.

Cyber-Physical Systems

The ***cyber-physical systems*** link different kind of entities (software and hardware) together with human actors.

They can be thought as being composed of some hardware components endowed with sensors and actuators and including some software (control) components that have to react and adapt to the changing environment.

Usually the cyber-physical applications are included in physical devices. Humans do not interact directly with these devices but using a cyber-physical layer that implements some control complex functions.

The control components have to react to ***internal and external asynchronous discrete events*** as well as to continuously changing of some variables. The general demands of such kind of hybrid control systems should be implemented by ***communicating concurrent tasks***.

Reactive Applications

The term *reactive* is often used linked to:

- programming,
- functional programming,
- systems and applications.

The reactive programming is generally considered an event-driven approach, unlike the *reactive systems* which are approached as being message-driven.

An event is a signal emitted by an entity when a specified state is attained.

A message is an item of data sent by an entity to a specified destination. The sender and the receiver of a message are decoupled, unlike the signaler and the handler of an event.

The event is emitted locally, while the message is transmitted to a distributed destination.

The functional reactive programming is a subset of asynchronous programming where the information drives the logic and determines the program state evolution instead of the control flows of the thread of executions.

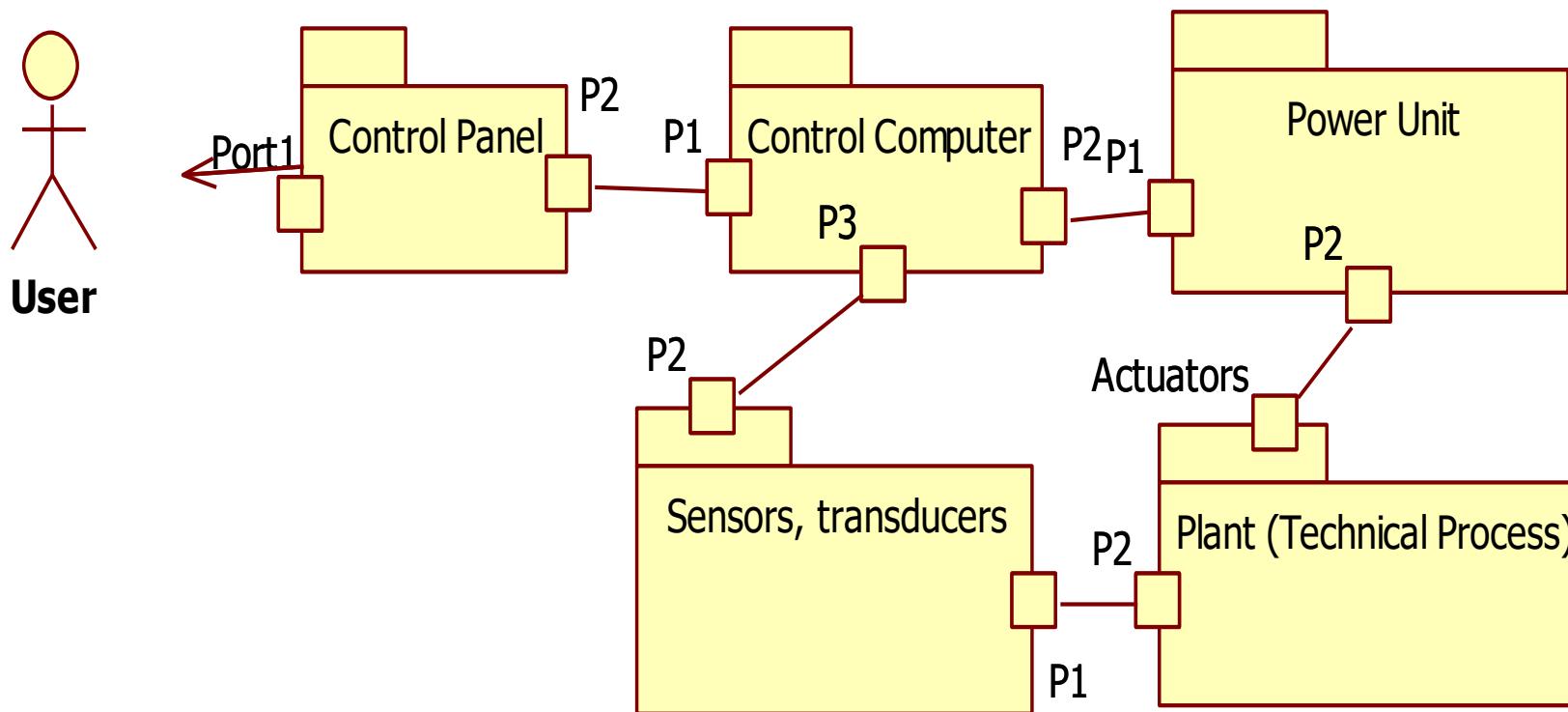
Often in the ***reactive applications***, the reactions concern only the different events, but the current proposed approach extends the reactions, beyond to the regular cases, to events characterized by different intensities involving sometimes asynchronous reactions to continuous values too.

The currently considered ***reactive applications*** are linked to given environments or plants. This requires the existence of some input and output channels. In the case of complex applications, for practical reasons, the expected reactions involve distributed architectures and implementations. As consequences, the conceived architectures should be based on models capable to cooperate for implementation of the reactive requirements using both events and messages.

The current development has the goals to conceive a kind of models that are capable to be used for:

- specification,
- model based design,
- verification,
- automatic model implementation
- integration and testing

Real Time Application – General Structure



Component diagram: ports that implement in/out methods

b. Definiții - Basic definitions

Definitions of RTS:

STR = un sistem de prelucrare care răspunde la evenimente generate extern sau intern într-o perioadă de timp finită și specificată (apriori).

RTS = An information processing system which has to respond to externally or internally generated input stimuli within a finite and specified period.

RTS= Real-time systems are those in which the correctness of the system depends not only on the logical results of the computation but also on the time at which the results are produced (and provided).

Source:Wikipedia

In computer science, **real-time computing (RToC)** is the study of hardware and software systems which are subject to a “*real-time constraint*”—i.e., operational deadlines from event to system response.

A ***non-real-time system*** is one for which there is no deadline specified and verified, even if fast response or high performance is desired or even preferred.

The needs of real-time software are often addressed in the context of *real-time operating systems*, and *synchronous programming languages*, which provide frameworks on which to build real-time application software.

A real time system may be one where its application can be considered (within context) to be *mission critical*.

Embedded systems

An *embedded system* is a computer system designed to perform dedicated functions with real-time computing constraints. It is embedded as part of a complete device, which often includes hardware and mechanical parts.

Conclusion:

RTS = Real-Time constraints specification + Design + Verification + Implementation + Integration + Testing

Tipuri de STR -

- **Sisteme de timp-real hard ---**
 - Control de timp-real ---
 - Sisteme inginerești --
 - Aplicații critice d.p.d.v. al siguranței ---
- **Sisteme de T-R soft ---**

deadlines are important but the systems still functions if the deadlines are occasionally missed. E.g.: multimedia applications.
- **Sisteme de T-R ferme ---**

if some responses (reactions) are missing the deadlines they are not necessary (usefully) any more, but the consequences are not disastrous.

Most of control systems are real-time systems.

Many people consider only the hard real-time systems as real-time systems.

Constrângeri de timp-real -

- constrângeri de T-R hard ---
- constrângeri de T-R soft ---

Types of RTSS:

Hard real-time systems = systems where it is absolutely imperative that the responses occur within the required deadline. Ex.: Control Real-Time (System, Computer) Engineering Systems Safety-critical applications

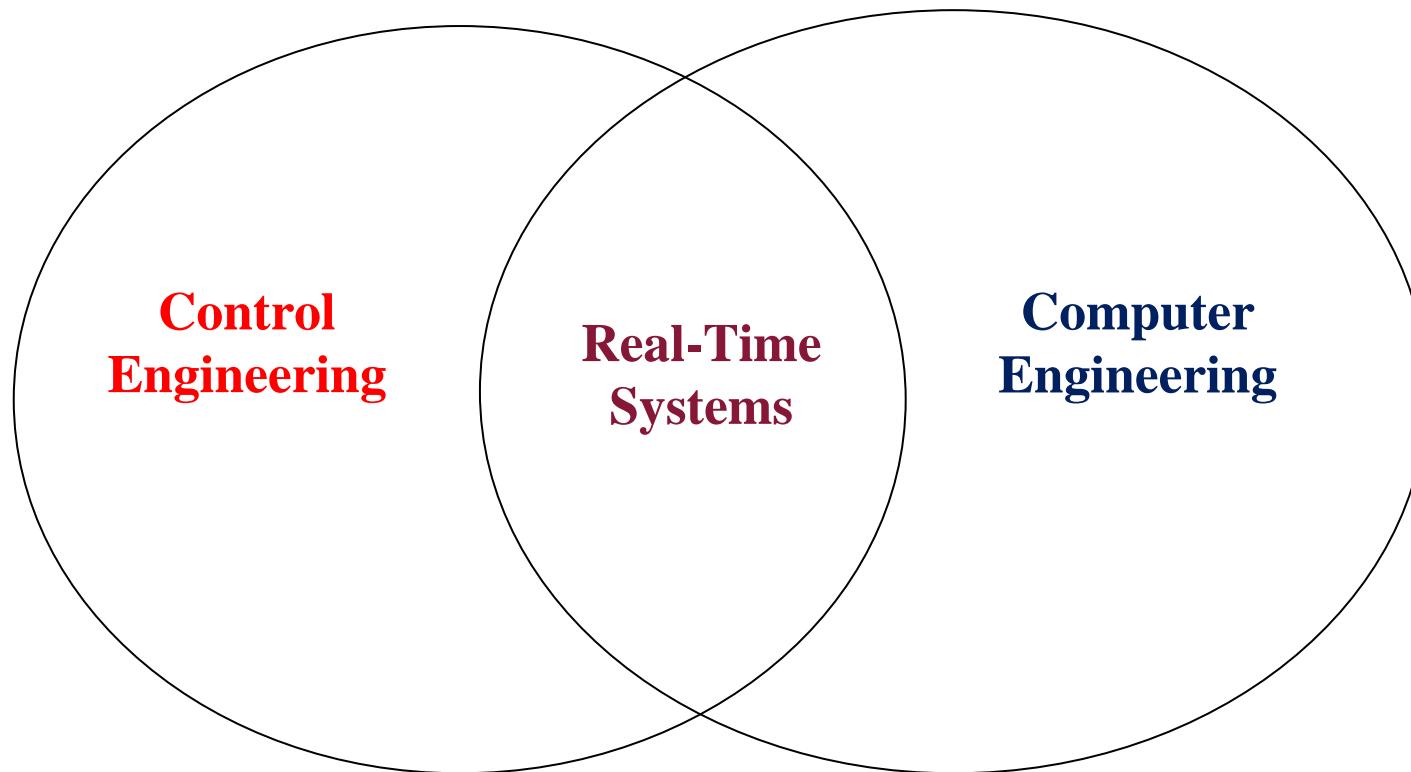
Soft real-time systems = systems where deadlines are important but the systems still functions if the deadlines are occasionally missed. E.g.: multimedia applications.

Firm real-time systems = systems where if some responses (reactions) are missing the deadlines they are not necessary (usefully) any more, but the consequences are not disastrous.

Real-time constraints:

hard real-time constraints
soft real-time constraints

c. Control de T-R - Real-Time Control



Tipuri -

1. Sisteme înglobate (încorporate) -

- dedicated control systems
- the computer is an embedded part of some piece of equipment
- they contain microprocessors (microcontrollers), real-time kernels, RTOS
- e.g.: aerospace, industrial robots, vehicular systems (cars, automotive), washing machines, health, home applications, etc.

2. Sisteme de control industrial -

- distributed control systems (DCS),
- programmable logic controllers (PLC),
- hierarchically organized,
- process industry,
- manufacturing industry,

Types of Real-Time Control Systems

Embedded Systems

Industrial Control Systems

Characteristicile sistemelor de control înglobate - **Embedded Control**

Characteristics 1:

- Limited computing and communication resources → resource efficiency
- Included in some products, (vehicles, tools, etc.)
- CPU time is limited,
- Lărgimea de bandă -- Communication bandwidth,
- Energy,
- Memory,
- Mod autonom de operare -- Autonomous operation → No human “operator” in the loop
- Au mai multe cazuri de utilizare și o funcționalitate complexă -- Several use-cases and complex functionality
- Deseori au nevoie de programe mari -- Often are necessary large amounts of software
- Deseori necesită garanții bazate pe demonstrații formale -- Often are necessary formal guarantees! Another solution – exhaustive tests

Caracteristici 2 --

- Eficiență cu resurse limitate --
 - Eficiență dimensiunii codului --
 - Eficiență în timpului execuției --
 - Eficiență utilizării energiei --
 - Eficiență greutății și dimensiunii --
 - Eficiență costului --
- Încredere în operarea autonomă --
 - Încredere --
 - Disponibilitate --
 - Siguranță --
 - Securitate --
 - Întreținere --

Embedded Control Characteristics 2:

- Limited resources Efficiency
 - Code-size efficiency
 - Run-time efficiency
 - Energy efficiency
 - Weight and size efficiency
 - Cost efficiency
- Autonomous Operation Dependability
 - Reliability
 - Availability
 - Safety
 - Security
 - Maintainability

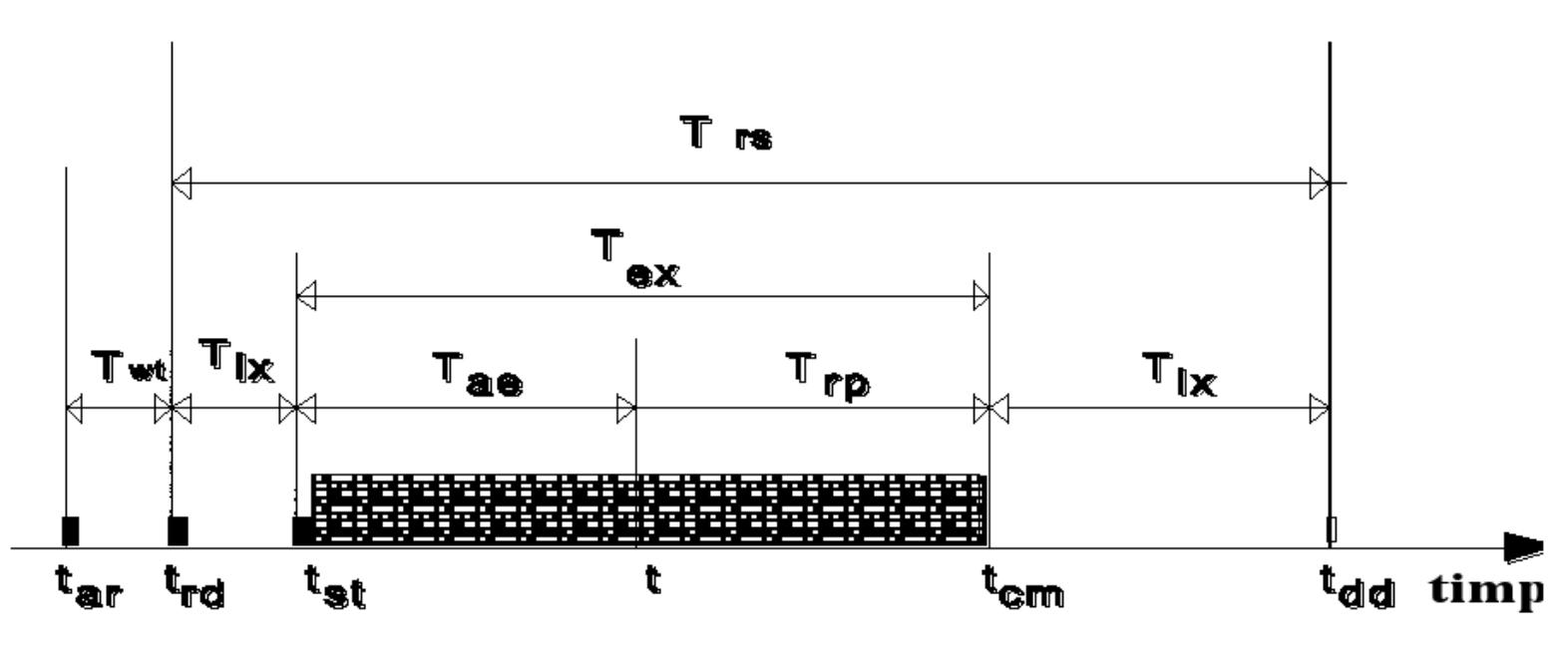
It is known that approximately **98%** of all processors produced are used in embedded applications.

- There are more than 16 billion embedded devices employed by 2010.
- The value added to the final product by embedded software is much higher than the cost of the embedded device itself.

d. Caracteristicile STR -- Real-Time System Characteristics:

- nereversibilitate -- **non-reversible**
- cerințe temporale -- **timing requirements**
- deterministe și predictibile -- **must be deterministic and predictable**
- timpi de răspuns în cazul cel mai defavorabil în loc de durata medie -- **worst-case response times** of interest rather than average-case response times
- mari și complexe -- **large and complex**
- distribuite -- **distributed**
- interacțiune strânsă cu hardware --- **tight interaction with hardware**
- critice d.p.d.v. al siguranței -- **safety critical**
- dependente de timpul de execuție -- **execution is time dependent**
- testarea dificilă -- **testing is difficult**
- operarea pe perioade mari de timp -- **operating over long time periods**

e. Parametrii temporali -- Temporal parameters



Time moments:

t_{ar} – arrival time

t_{rd} – ready time

t_{st} – start time

t_{cm} – completion time

t_{dd} – deadline

Durations:

T_{wt} – waiting time

T_{rs} – response time

T_{ae} – attained execution time

T_{rp} – remaining processing time

T_{lx} – laxity time

Temporizări nedeterministe --

Non-Deterministic Timing:

- Utilizarea în comun a resurselor -- Caused by sharing of computing resources - multiple tasks sharing the CPU
- preemțiunea, blocajul, inversare priorității, variația timpilor de execuție
 - preemptions, blocking, priority inversion, varying computation times, ...
 - Caused by sharing of network bandwidth
- bucle de control închise prin intermediul sistemului de comunicație -- control loops closed over communication networks
- întârzieri ale interfeței de rețea, întârzieri la punerea la coadă, întârzieri de transmisie, propagare și retrasmisie datelor, confirmarea, pierderea pachetelor -- network interface delay, queuing delay, transmission delay, propagation delay, resending delay, ACK delay, ... lost packets

Cum se poate minimiza sau elimina ne-determinismul? Cum afectează acesta performanța?

How can we minimize the non-determinism?

How does the non-determinism effect the control performance?

Evenimente ---

Real-Time systems must respond to external and/or internal events.

- Periodic events
- Aperiodic events - unbounded arrival frequency
- Sporadic (episodic) events - bounded arrival frequency

The real-time systems react to events after the calculation of the response using a certain amount of processing time. The reaction must be before the specified deadline! Often the RTS must react to more than one event during the same period of time.

Variante de implementare a controlului --

- antrenate de evenimente --
- antrenate de timp --

Events:

Types of control implementation:

- Event driven control
- Time driven control

*

*****END*****

*

2. Specification of R-T Applications

Contents

- 2.1. Construction of R-T applications (R-TAs)
- 2.2. Approaches of R-TAs
- 2.3. Specification and verification of R-TAs with Petri Nets
- 2.4. Specification and verification of R-TAs with Time Petri Nets
- 2.5. Specification and verification with Object Enhanced Time Petri Nets (OETPN)
- 2.6. Specification and verification with Stochastic Object Enhanced Time Petri Nets (SOETPN)

2.1. Construction of R-T applications

Real-Time (R-T) Systems concern:

- R-T applications
- R-T operating systems or R-T executives
- R-T programming languages
- R-T hardware
 - microcontrollers
 - microcomputers + I/O interfaces
 - computers + I/O interfaces
 - FPGAs
 - PLCs etc.

Construction of R-T Applications

R-T Applications involve:

- hardware
 - computer (R-T or Non R-T?)
 - I/O channels
 - controlled plant (sensors, detectors, transducers, effectors, actuators, +?)
- software
 - operating system (R-T or Non R-T, executives)
 - execution environment
 - application program
- control algorithms

Development of R-T Applications

Software ← **Waterfall model:**

1. Requirements specification → Validation
2. Design
3. Implementation
4. Verification
5. Integration
6. Testing (and validation)
7. Deployment
8. Maintenance

Control algorithm synthesis

← Control engineers

Control algorithms:

- discrete time system
- discrete event systems
- hybrid systems

What have to be specified?

- control algorithms
- plant
- relation controller - plant
- relation of users with plants and controllers

What have to be verified?

- all the previous
- software design and implementation
- relations hardware - software - environment

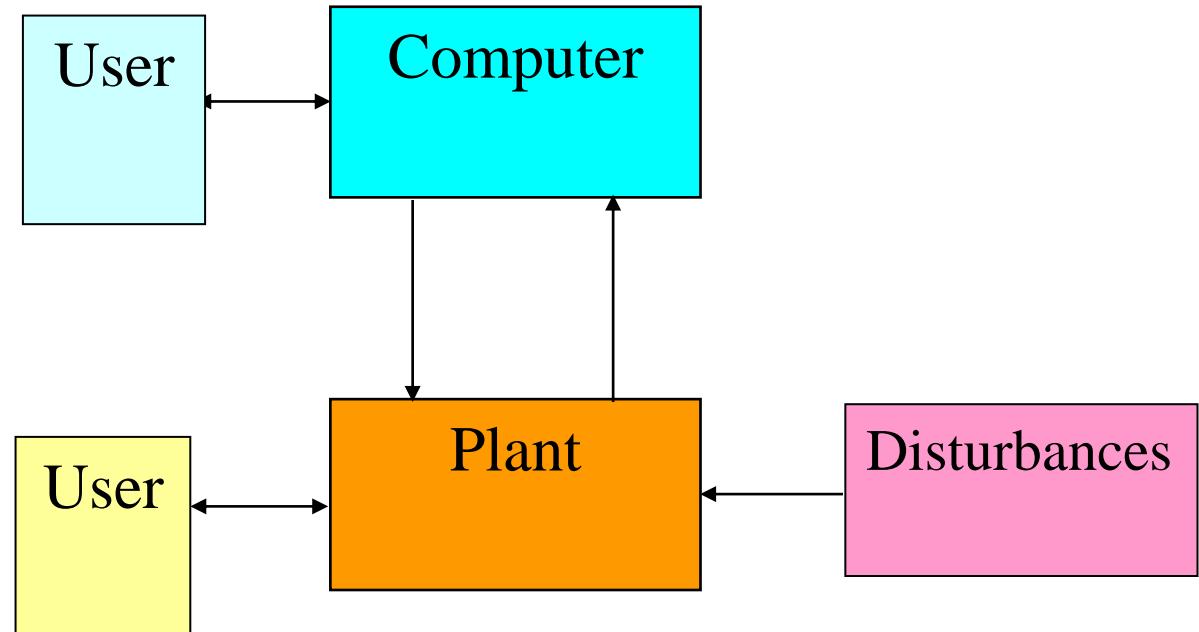


Fig. 1. Structure of a control application.

All of them are based on models:

Specification → models

Design → models (diagrams)

Implementation → model simulation and programs

Verification → models

Testing → models + test programs

Deployments → models

Maintenance → models

Formal development

vs.

Non-formal development

2.2. Approaches of R-T applications

The design is based on different types of ***models***.

Specification – it mentions what should has be modeled and built

Overall system description

- function- ***what*** it's supposed to do
- temporal behavior - ***when*** it does
- performance - ***how*** well it must do it
- the structure of the system - the components of the system
- interfaces - ***how*** it fits in with the environment
- constraints - do's and don'ts
- development - ***how*** must it be built

A tool for specifying real-time system should guarantee both correctness and completeness of the formal specification, as well as the satisfaction of the system behavior with respect to both the timing constraints and the high-level behavior descriptions.

The results: specifications

Approaches of R-T applications:

- *Synchronous* approach – activities with no durations (can be ignored)
- *Asynchronous* approach – activities have significant durations
- *Mixed* approach – activities with and without significant durations

The differences involve:

- design
- scheduling
- verification

*

*** **END** ***

*

2. Specification of R-TAs

Outline

2.3 Specification and Verification of R-TS with Petri Nets

- a. Ordinary (standard) PN
- b. Program flow description – Petri Net based Language
- c. Stochastic PN

2.4 Specification and Verification with Time and Timed PN

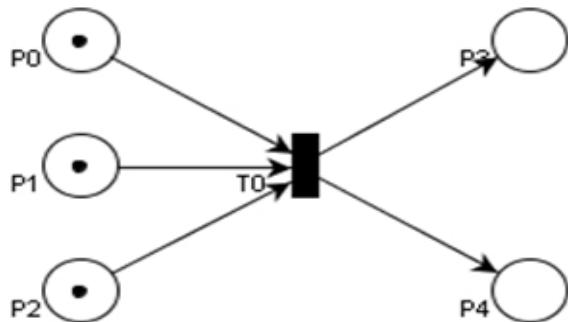
- a. Timed PN
- b. Transformation of timed transition PN to timed place PN
- c. Alternative
- d. Deadlock avoidance with timings
- e. Constraints specification and verification
- f. Time Interval Petri Nets
- g. Delay Time Petri Nets
- h. Applications of TPN and DTPN
- i. Time Petri Nets with Inhibitor Arcs

2.5 Specification and Verification with Object Enhanced Time Petri Nets

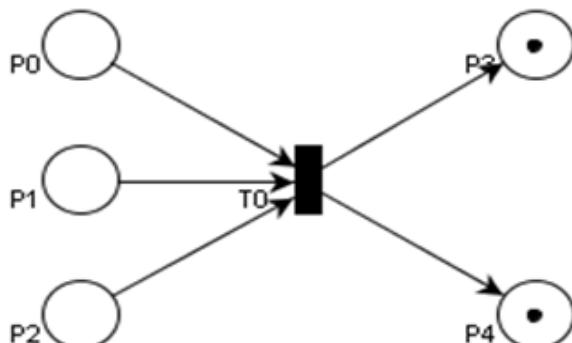
2.6. Specification and verification with Stochastic Object Enhanced Time Petri Nets (SOETPN)

2. 3 Specification and Verification of R-TAs with Petri Nets

a. Ordinary Petri Nets



Places - P_i
Transition - T_i
Enabled (fireable) transition
Execution (fire) of a transition



When is the transition executed?

Petri Nets Definitions

Definition 1.

They are bipartite graph. They are a collection of 4-tuple:

$$N = (P, T, \text{pre}, \text{post})$$

where

$P = \{p_1, p_2, \dots, p_m\}$ finite set of places ;

$T = \{t_1, t_2, \dots, t_n\}$ finite set of transitions;

$\text{pre}: P \times T \rightarrow \mathbf{Nn}$ (natural number set) ***backward incidence function***:

$$\text{pre}(p, t) = \begin{cases} 0, & \text{if there is not an arc from } p \text{ to } t \\ \neq 0, & \text{if there is an arc from } p \text{ to } t \end{cases}$$

$post: P \times T \rightarrow \mathbf{Nn}$ (natural number set) is *forward incidence function*:

$$post(t,p) = \begin{cases} 0, & \text{if there is not an arc from } t \text{ to } p \\ \neq 0, & \text{if there is an arc from } t \text{ to } p \end{cases}$$

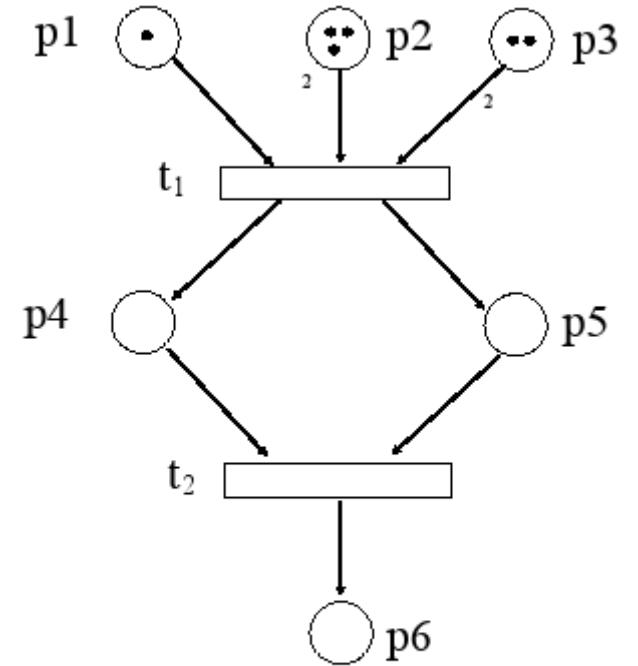
$N = (P, T, pre, post)$ ↪ structure without marking

$PN = (N, M_0)$ ↪ structure and marking

$M : P \rightarrow \mathbf{Nn}$ is the marking.

The marking describes the PN state.

$$\mathbf{M} = [M(p_1), M(p_2), \dots, M(p_m)]^T$$



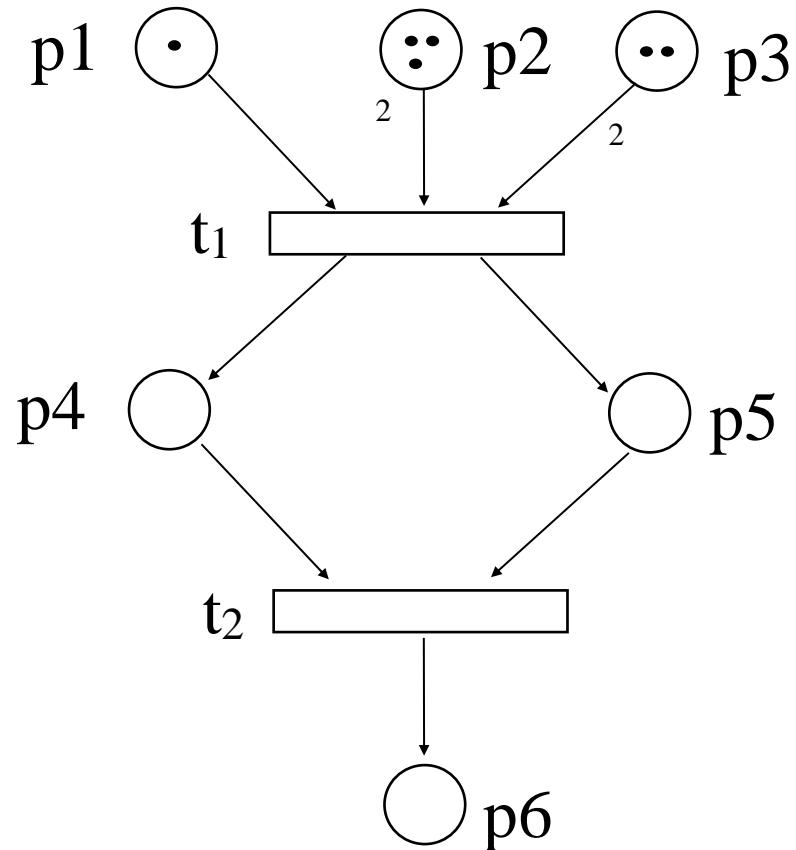
Ex.: Petri net

Existent relations:

$$P \cap T = \Phi \quad \text{and} \quad P \cup T \neq \Phi,$$

Marking constraints:

$$M(p) \leq K(p), \forall p \in P.$$



Ex.: Petri net

Definition 2.

$$N = (P, T, F, W)$$

$P = \{p_1, p_2, \dots, p_m\}$ finite set of places ;

$T = \{t_1, t_2, \dots, t_n\}$ finite set of transitions;

F is the set of oriented arcs. It describes the flows.

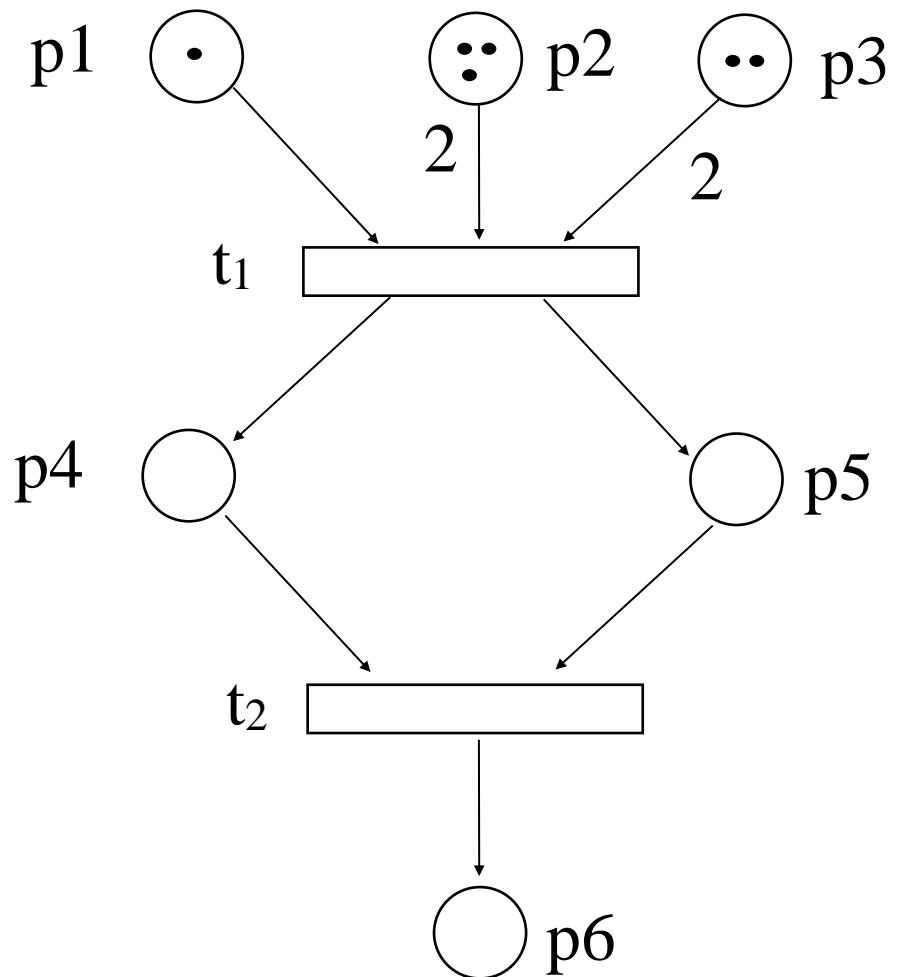
$$F \subset (P \times T) \cup (T \times P),$$

$W : F \rightarrow \mathbf{Nn}$, (\mathbf{Nn} – natural number set)

$W(p_i, t_j)$ or $W(t_j, p_i)$ assigns weight to arcs.

$M : P \rightarrow \mathbf{Nn}$ is the marking.

The marking describes the PN state.



Ex.: Petri net

Transition input place sets:

Transition $t \rightarrow$ input place set 0t

$${}^0t = \{p \mid p \in P, pre(p,t) \neq 0\}$$

or

$${}^0t = \{p \mid p \in P, \exists (p,t) \in F\}$$

Transition output place sets:

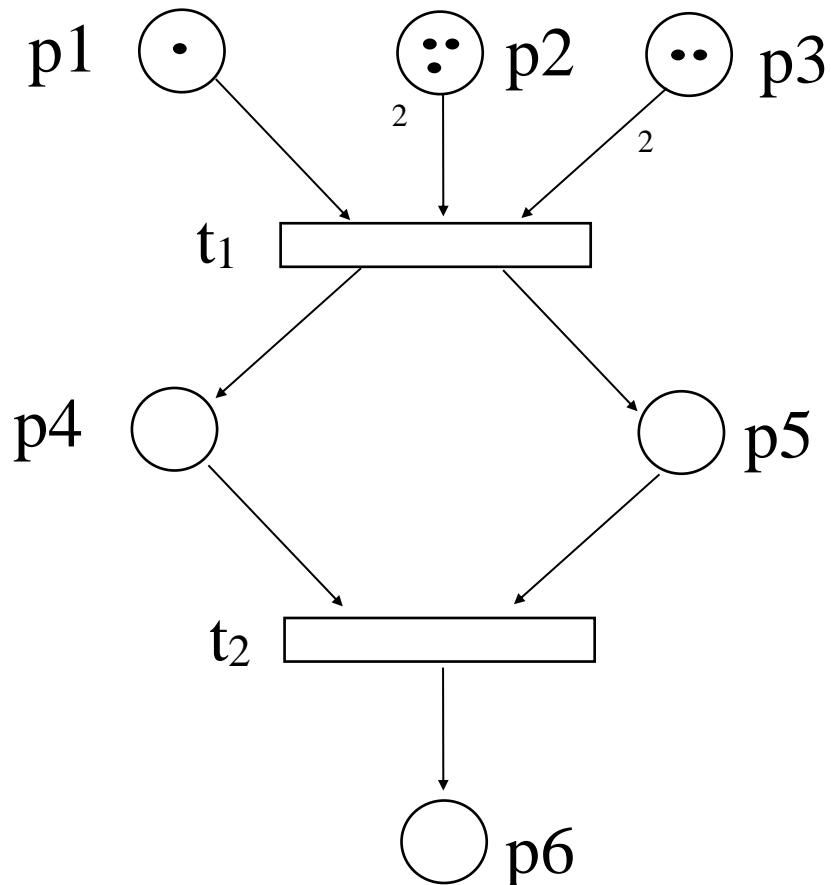
Transition $t \rightarrow$ output place set t^0 :

$$t^0 = \{p \mid p \in P, post(t,p) \neq 0\}$$

or

$$t^0 = \{p \mid p \in P, \exists (t,p) \in F\}$$

$$\mathbf{M} = [M(p_1), M(p_2), \dots, M(p_m)]^T$$



Ex.: Petri net

Place input transition sets:

Place p \rightarrow input transition set 0p

$${}^0p = \{ t \mid t \in T, post(t,p) \neq 0 \}$$

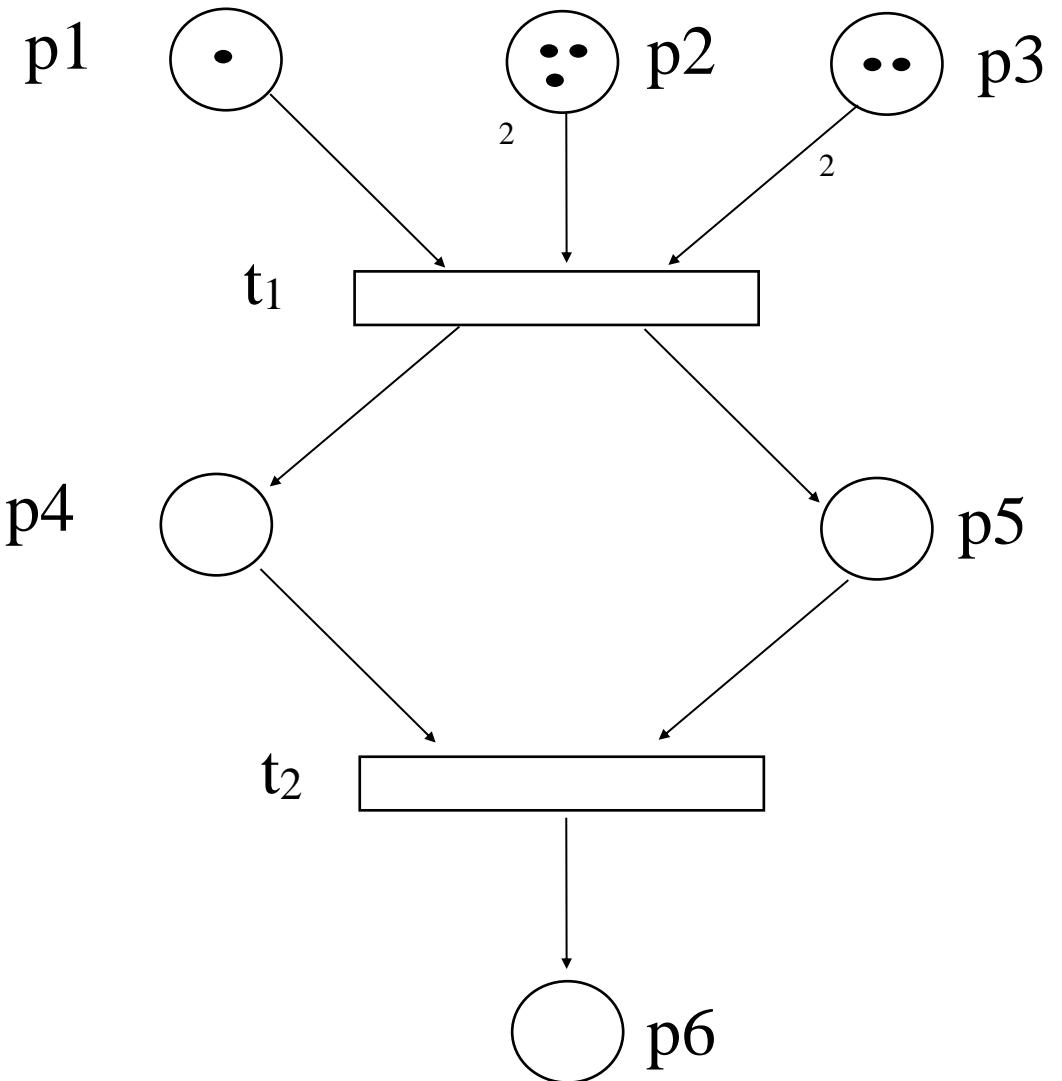
$${}^0p = \{ t \mid t \in T, \exists (t,p) \in F \}$$

Place output transition sets:

Place p \rightarrow output transition set p^0 :

$$p^0 = \{ t \mid t \in T, pre(p,t) \neq 0 \}$$

$$p^0 = \{ t \mid t \in T, \exists (p,t) \in F \}$$



Ex.: Petri net

Petri net incidence matrix C

$$c_{ij} = post(t_j, p_i) - pre(p_i, t_j), i = 1, \dots, m; j = 1, \dots, n$$

Exemple 1

$$P = \{p_1, \dots, p_6\},$$

$$T = \{t_1, t_2\},$$

$$\mathbf{M}_0 = [1, 3, 2, 0, 0, 0]^T,$$

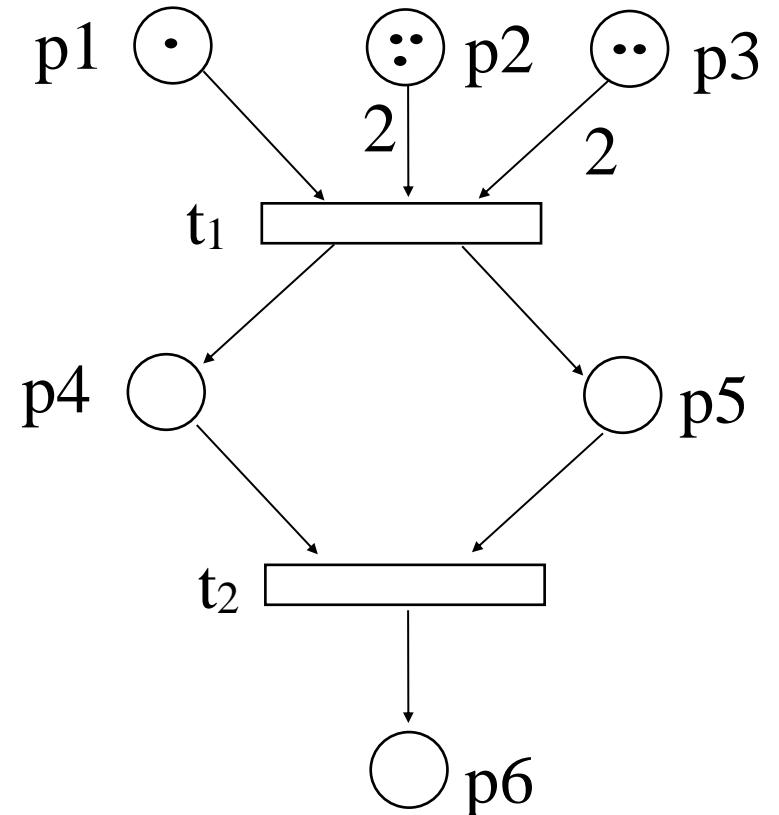
$${}^0t_1 = \{p_1, p_2, p_3\},$$

$$t_1^0 = \{p_4, p_5\},$$

$${}^0p_1 = \emptyset, p_1^0 = \{t_1\}.$$

$$pre(p_1, t_1) = 1, pre(p_2, t_1) = 2, \dots, pre(p_6, t_2) = 0,$$

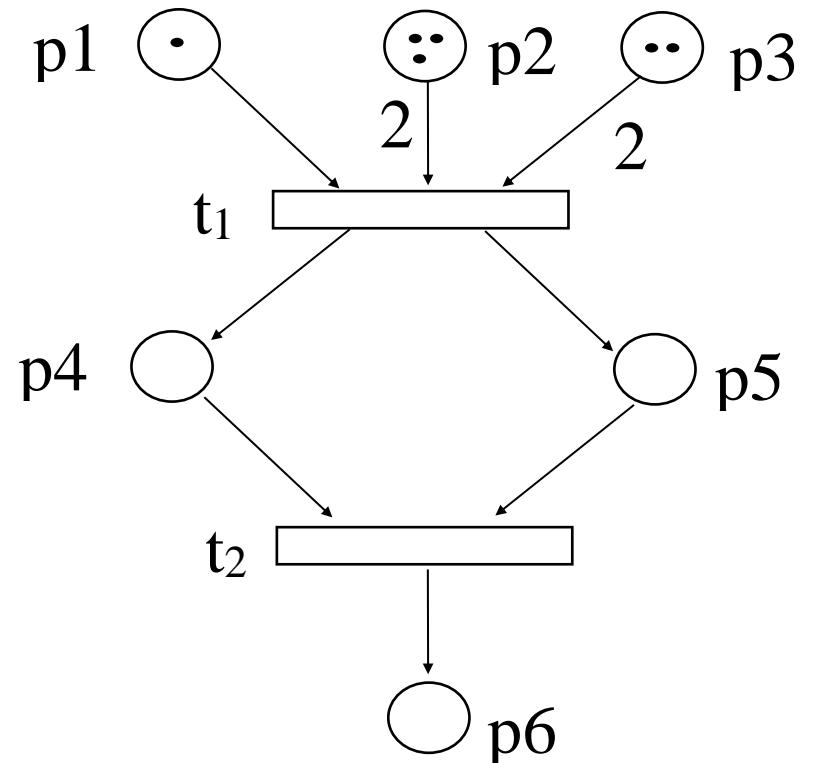
$$post(t_1, p_1) = 0, post(t_1, p_2) = 0, \dots, post(t_2, p_6) = 1.$$



Ex.: Petri net

Incidence matrix C:

$$C = \begin{bmatrix} -1 & 0 \\ -2 & 0 \\ -2 & 0 \\ 1 & -1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$$



Pre and Post matrix

$$C = \text{Post} - \text{Pre}$$

Ex.: Petri net

Enabled (executable) transition – Firing rule

Transition t is **enabled** for the current marking M iff:

$$M = [M(p_1), \dots, M(p_m)]^T$$

$$1. M(p) \geq pre(p,t) \quad \forall p \in {}^0t ;$$

$$2. K(p) \geq M(p) - pre(p,t) + post(t,p) \quad \forall p \in t^0.$$

An enabled transition may or may not be fired.

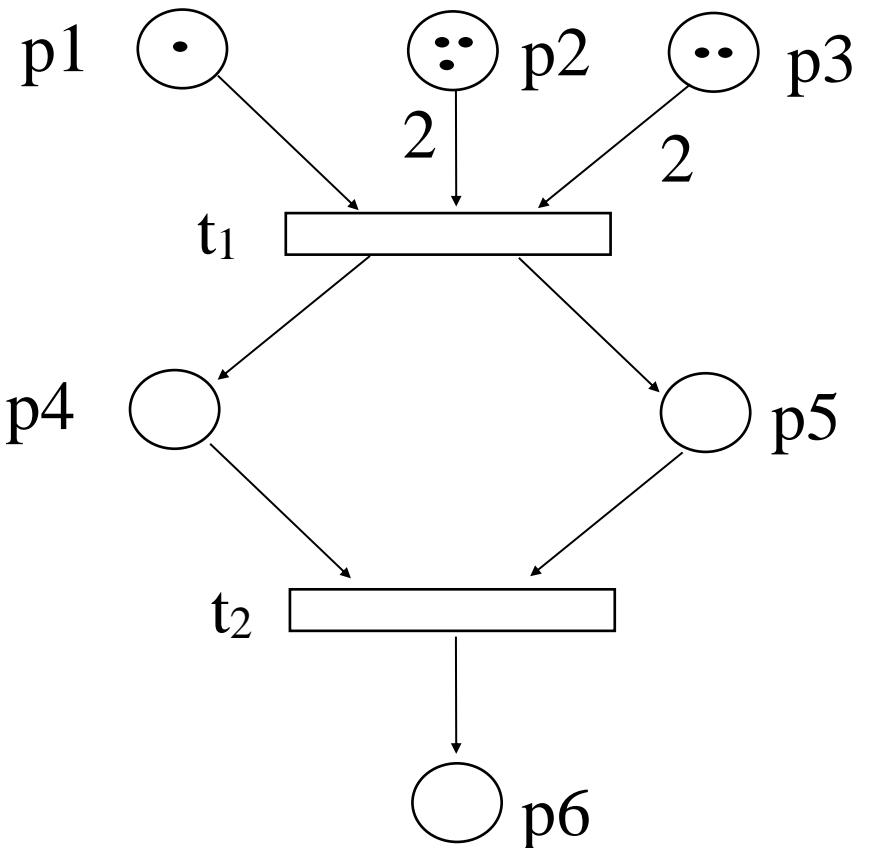
Dynamic behaviour

An enabled transition t removes $w(p,t)$ tokens from each input places and adds $w(t,p)$ tokens to each output places of t . Similar can be used $pre()$ and $post()$.

- weak transition rule
- strict transition rule

$$M \xrightarrow{t} M'$$

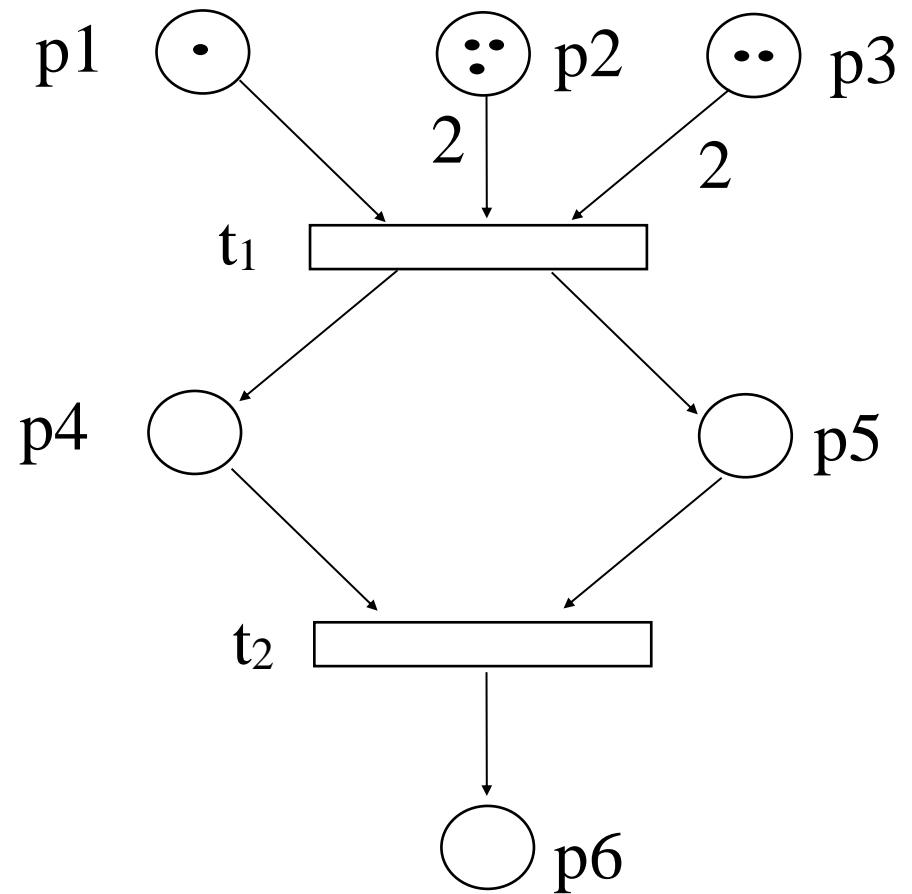
$$M'(p) = M(p) - pre(p,t) + post(p,t), \quad \forall p \in P$$



Ex.: Petri net

or

$$\mathbf{M}' = \mathbf{M} + col_t(\mathbf{C})$$

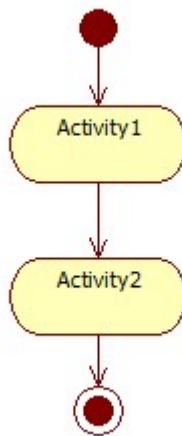


Ex.: Petri net

b. Program Flow Description

Flow = sequence of instructions

.....
a = 5;
b = 3;
x= a+b;
.....

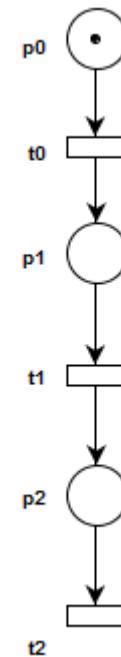


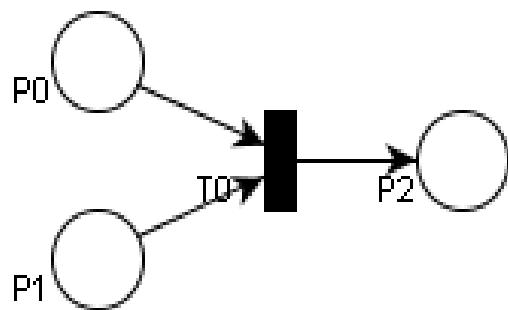
Sequence:

$$\sigma = t_0 * t_1 * t_2$$

Petri Net based Language (PNL)

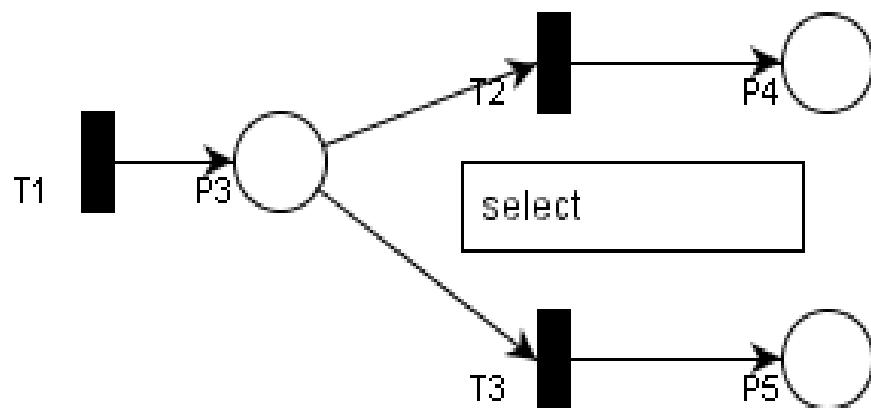
- formal language
- it describes precisely the behaviour





conditional execution

Alternative
Selection ***a OR b***
if condition then a
else b
 $\sigma = t_0 * (t_1 + t_2)$



select

Selection

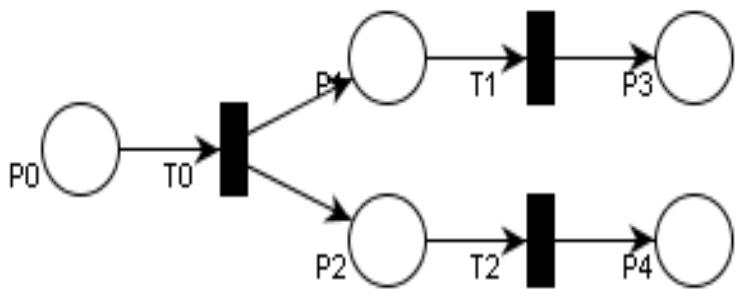
$$\sigma = T_1 * (T_2 + T_3) = \\ = T_1 * T_2 + T_1 * T_3$$

T1: int x=inputChannel.read();
if (x>0) **then** execute T2
else execute T3;

.....

T2:

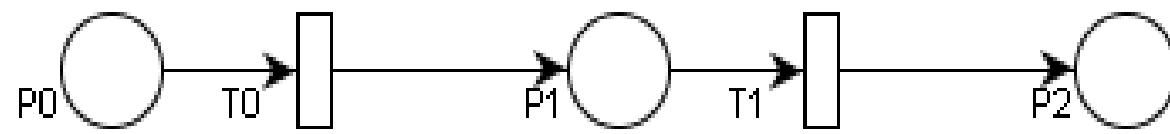
T3:



Concurrent execution:
 $\sigma = T_0 * (T_1 \& T_2) =$
 $= T_0 * T_1 * T_2 + T_0 * T_2 * T_1$
 $\sigma = T_0(?) * T_1(?)$

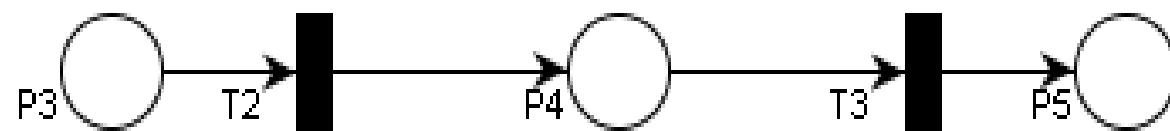
‘*’ \leftarrow sequence, $T_1 * T_2$
 ‘+’ \leftarrow selection, $T_1 + T_2$ (OR)
 ‘&’ \leftarrow concurrency $T_1 \& T_2 = T_1 * T_2 + T_2 * T_1$
 T_1 is concurrent to T_2

Split of execution
 ‘&’ \leftarrow AND,
 $T_1 \& T_2 = T_1 * T_2 + T_2 * T_1$

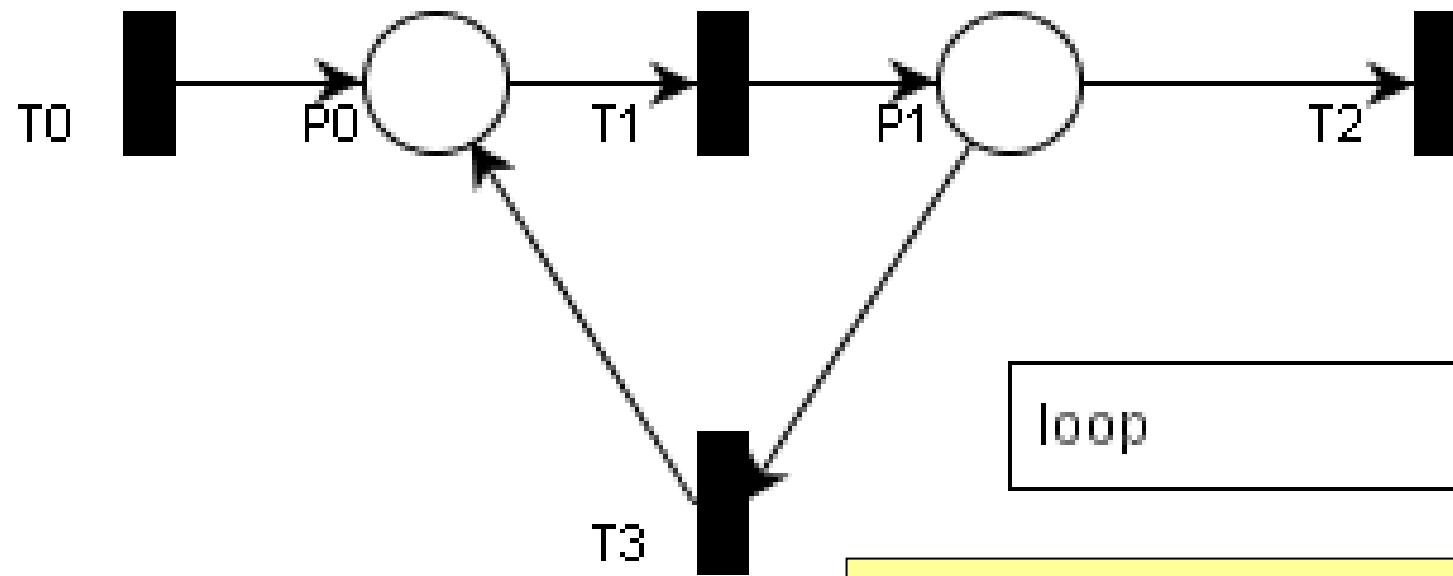


timed transition

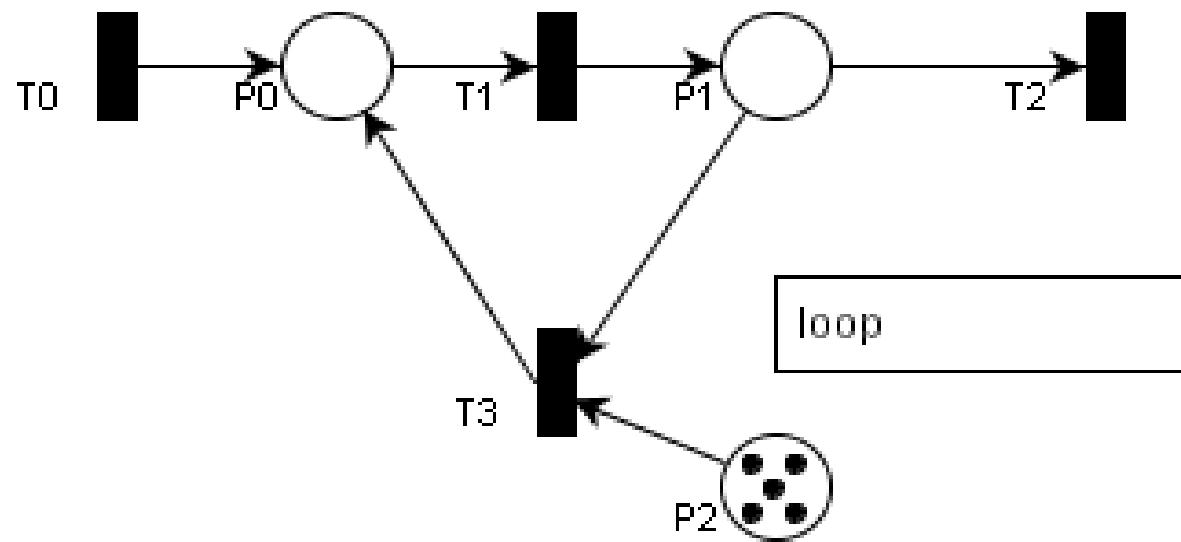
Sequential execution of transitions
 $\sigma = T_2 * T_3$



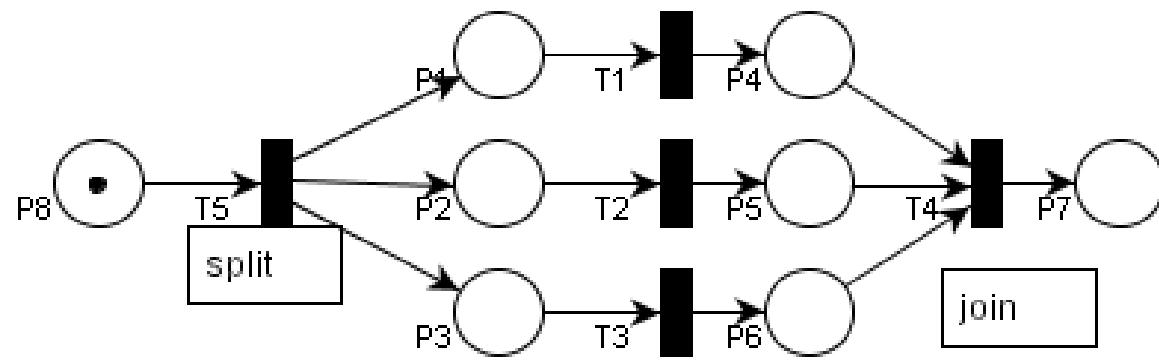
untimed transition



Undefined number of loop executions
 $\text{loop} = T_1 \# T_3$
 $\sigma = T_0 * T_1 * (T_3 \# T_1)^n * T_2$
 $n=1, 2, \dots$
 loop description:
 $\text{loop} = T_3 \# T_1$



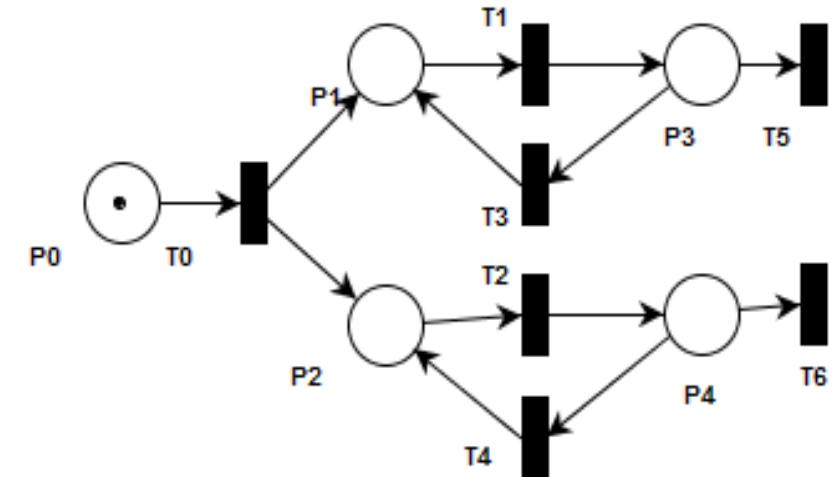
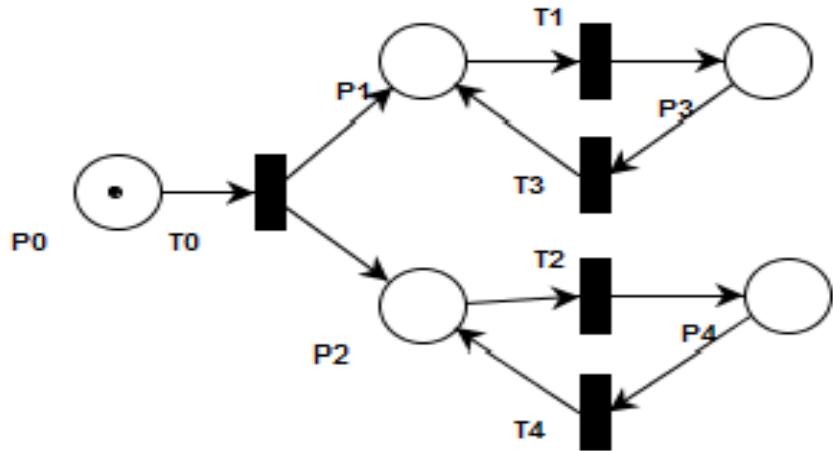
Predefined number of loop executions
 $\text{loop} = T_1 \# T_3$



$$\sigma = T_5 * (T_1 \& T_2 \& T_3) * T_4$$

$$\sigma = T_5 * (T_1 * T_2 * T_3 + T_1 * T_3 * T_2 + T_2 * T_3 * T_1 + T_2 * T_1 * T_3 + T_3 * T_2 * T_1 + T_3 * T_1 * T_2) * T_4.$$

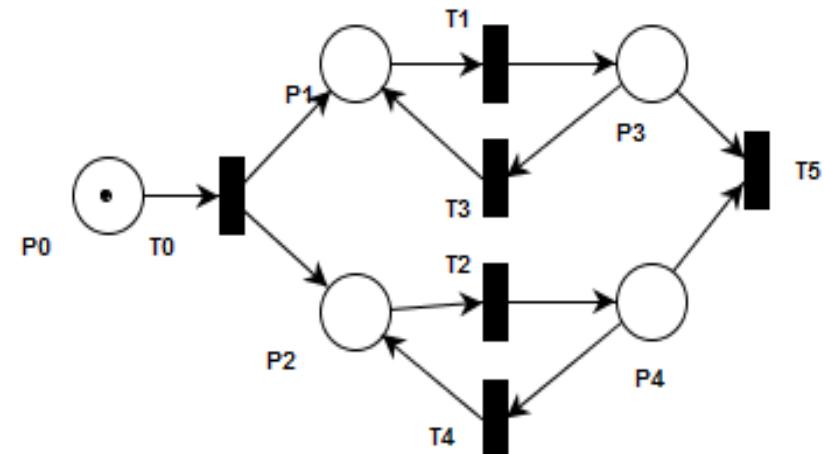
- Concurrent transitions: T_1, T_2, T_3

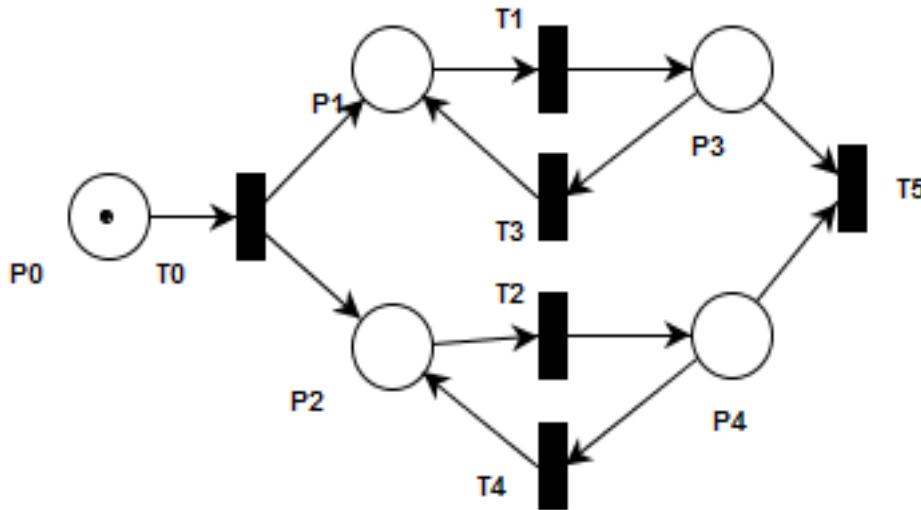


$\sigma = ?$

How many times is T_1 executed related to T_2 ?

$T_5 \leftarrow \text{join}$
 $\sigma = ?$





$$\sigma = T_0 * ((T_1 * (T_3 \# T_1)^m) \& (T_2 * (T_4 \# T_2)^n)) * T_5$$

$m, n = 0, 1, 2, \dots$

Concurrent sequences: ???

What can appear on the screen?

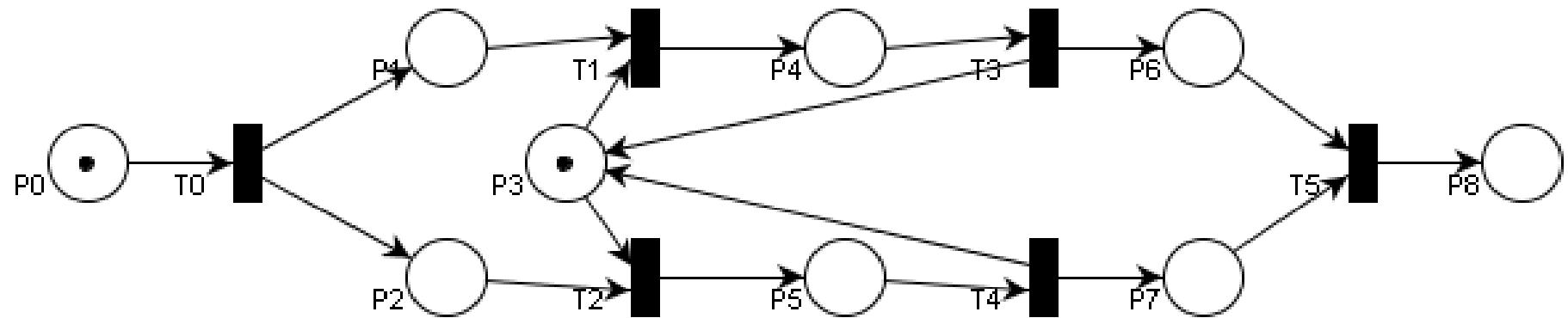
Identify the PN that executes:

$$\sigma = T_0 * ((T_1 * (T_3 \# T_1)^m * T_5) \& (T_2 * (T_4 \# T_2)^n * T_6)) =$$

$$\sigma = T_0 * (T_1 * T_3 * T_2 * T_4 * T_5 * T_6 \dots + \dots)$$

$m, n = 0, 1, 2, \dots$

$$\sigma = ???$$



Mutual exclusion $\rightarrow P_3$

What parts of the program are mutual excluded?

What can appear on the screen?

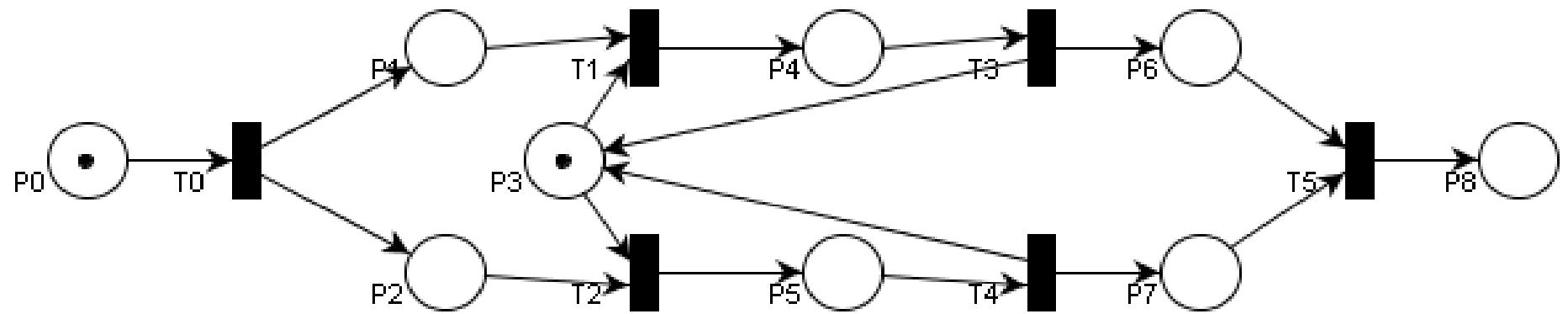
Constraints:

$\text{mutex}\{(T_1*T_3), (T_2*T_4)\}$

$$\sigma = T_0 * ((T_1 * T_3) \& (T_2 * T_4)) * T_5$$

mutex{(T₁*T₃), (T₂*T₄)}

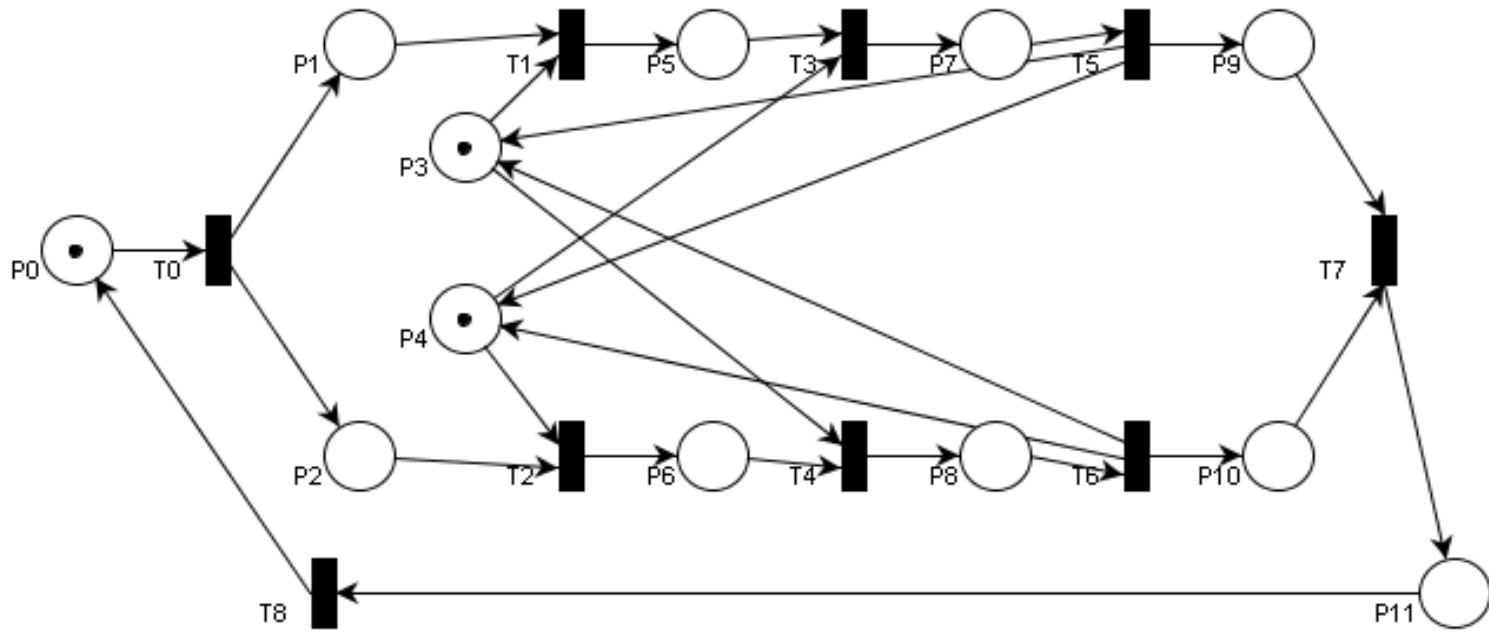
$$\sigma = T_0 * (T_1 * T_3 * T_2 * T_4 + T_2 * T_4 * T_1 * T_3) * T_5$$



Mutual exclusion $\rightarrow P_3$

What parts of the program are mutual excluded?

Constraints:
mutex{(T₁*T₃), (T₂*T₄)}



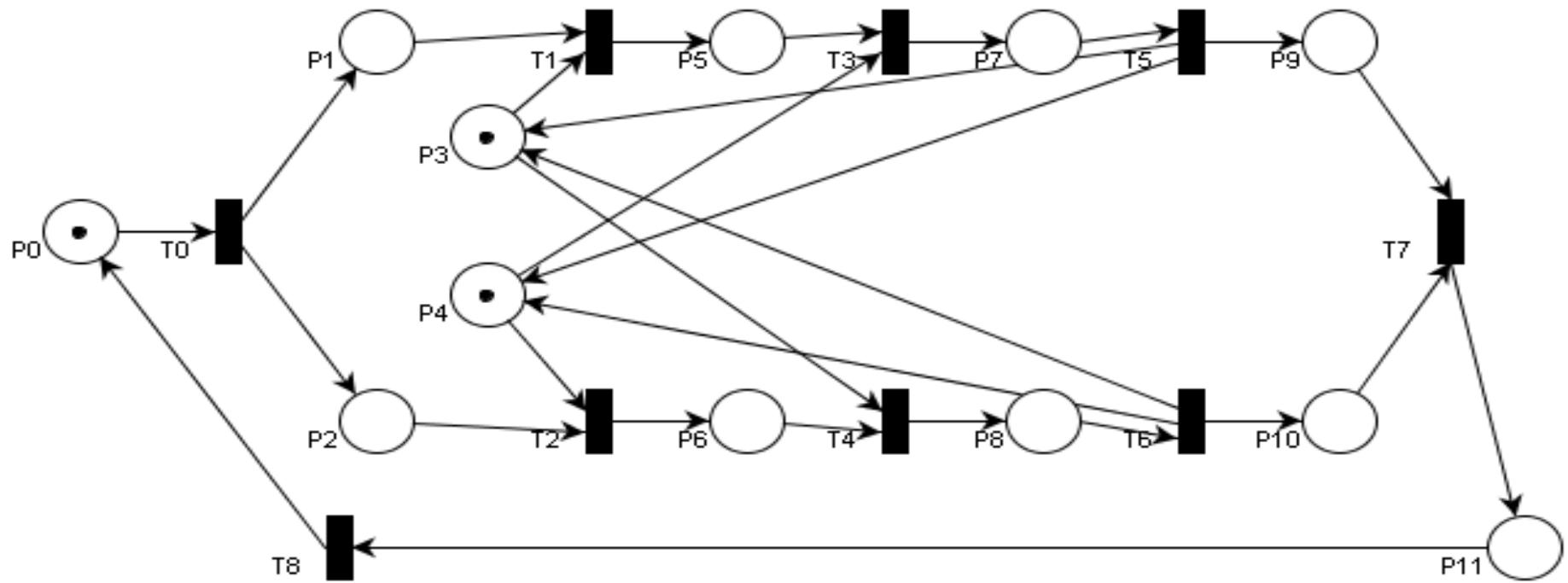
(2 resources controlled by P_3 and P_4) Deadlock $\rightarrow \delta$. Is a deadlock here?

$\sigma_1 = (T_0 * T_1 * T_3 * T_5 * T_2 * T_4 * T_6 * T_7) * T_8 * T_0 * \dots \rightarrow \text{NO}$

$\sigma_2 = T_0 * T_1 * T_2 * \dots = T_0 * T_1 * T_2 * \delta \rightarrow \text{YES}$

$\sigma_1 = T_0 * T_1 * T_3 * T_5 * T_2 * T_4 * T_6 * T_7 * T_8 * \sigma_1 \rightarrow$ unlimited number of (cyclic) executions

Constraints?

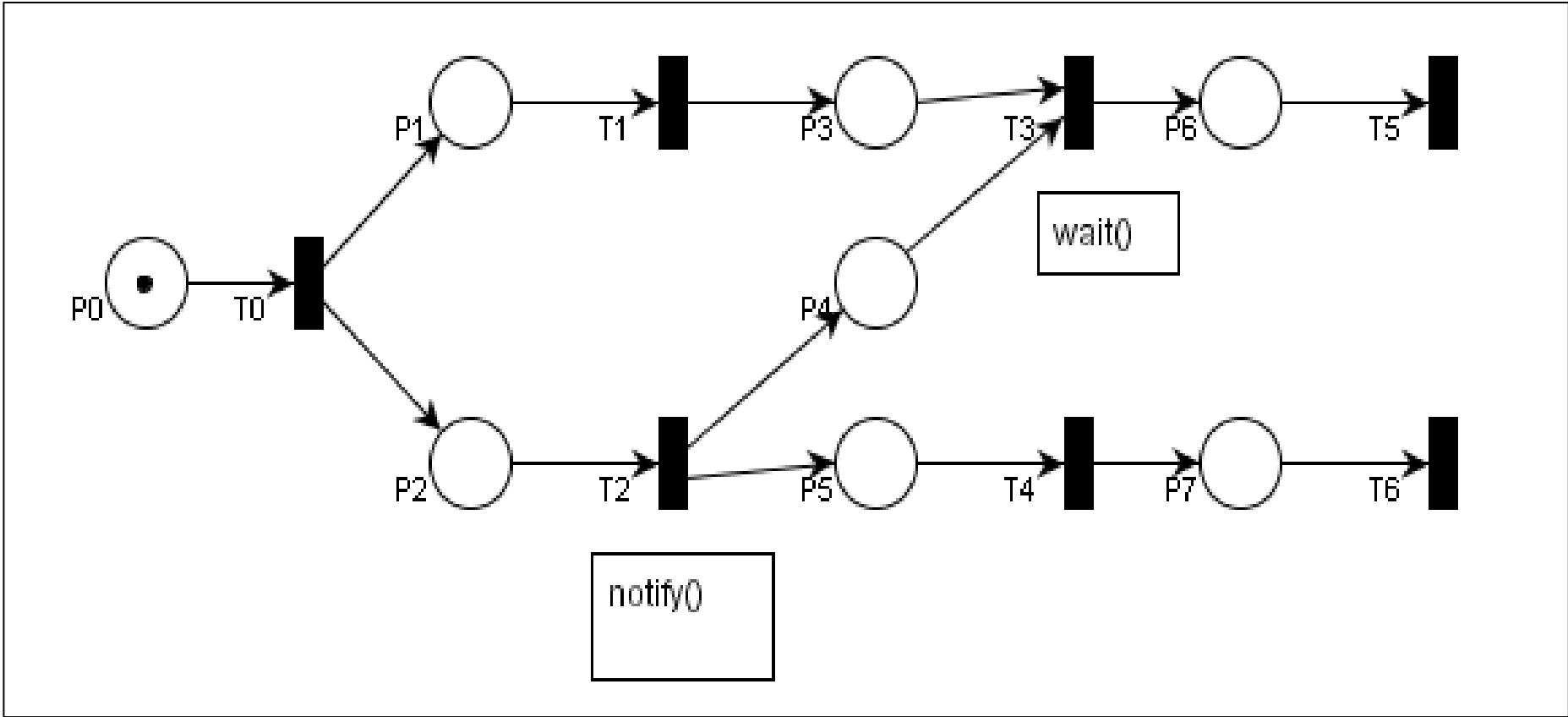


Constraints:

mutex_1{ (T₁*T₃*T₅), (T₄*T₆) }

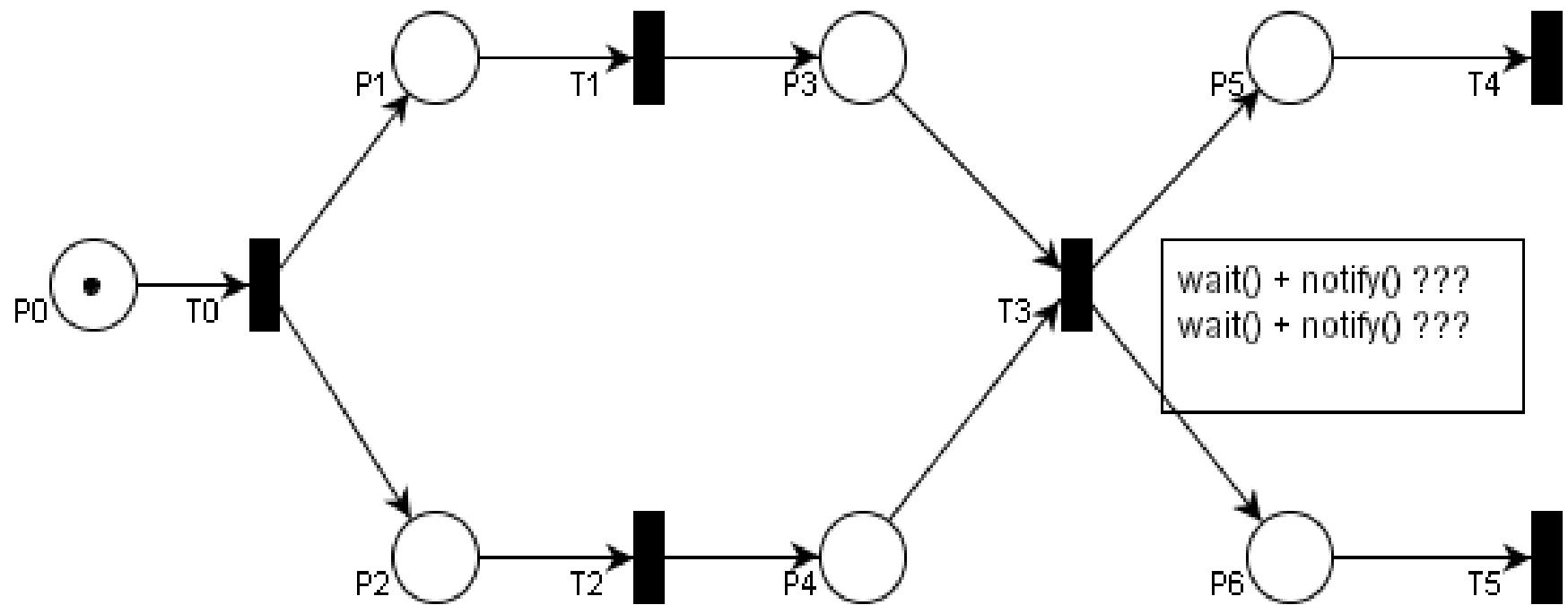
mutex_2{ (T₂*T₄*T₆), (T₃*T₅) }

$\sigma = T_0 * (T_1 * T_3 * T_5 * T_2 * T_4 * T_6 + T_2 * T_4 * T_6 * T_1 * T_3 * T_6 + T_1 * T_2 * \delta + T_2 * T_1 * \delta) * T_7 * T_8 * \sigma$
It depends on the selection!



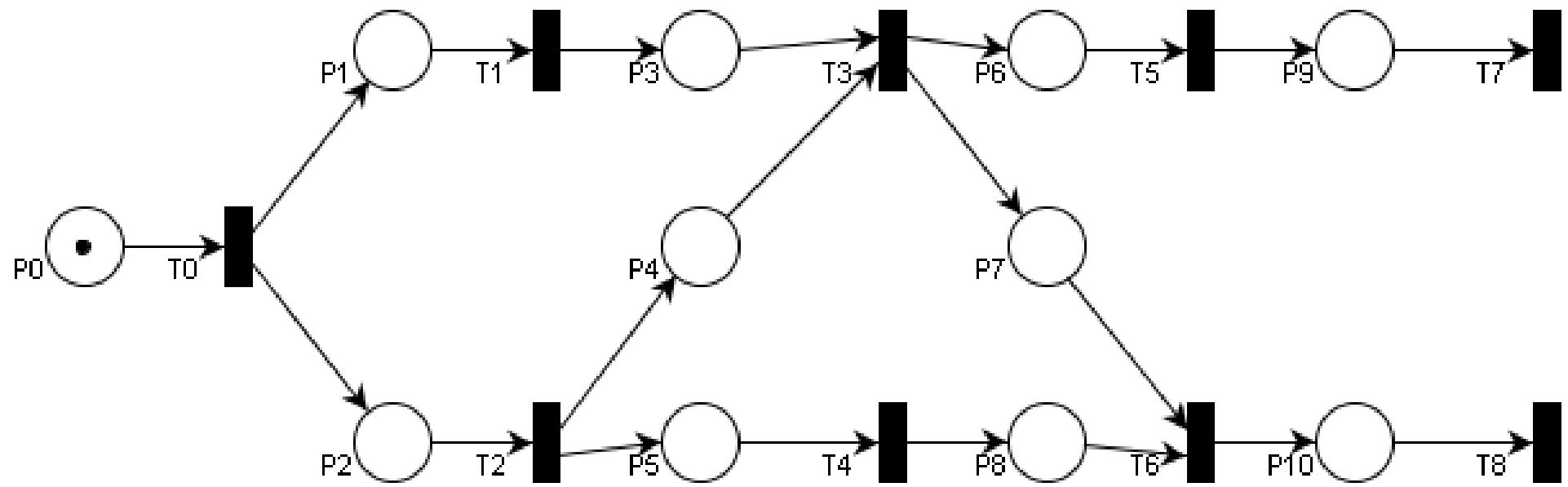
Synchronization (one direction)
 Java: `wait()` + `notify()`

Constraint:
 $T_2 < T_3$ (T_2 previous to T_3)
 $T_2 \rightarrow T_3$



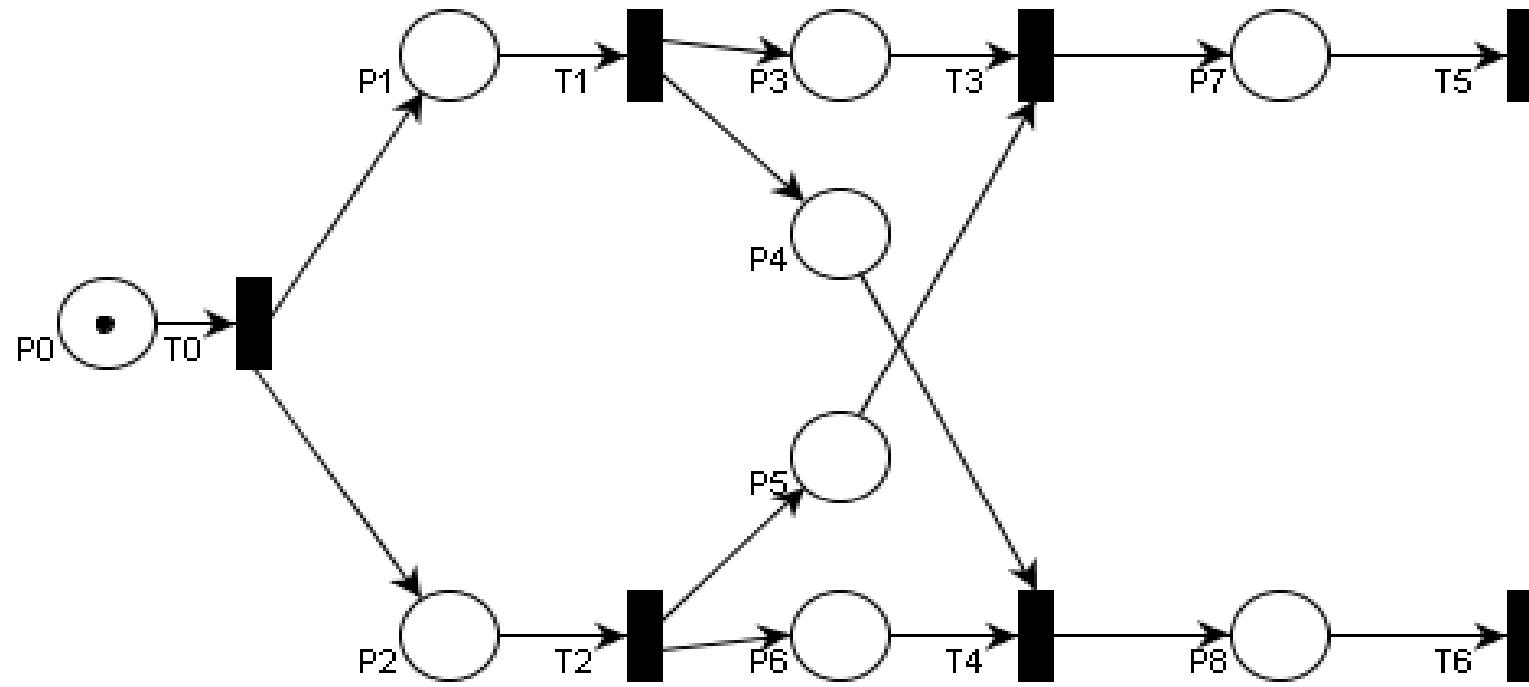
Double synchronization 1

How can this be programmed using Java language?
Barrier?



Double synchronization 2

How can this be programmed using Java language?

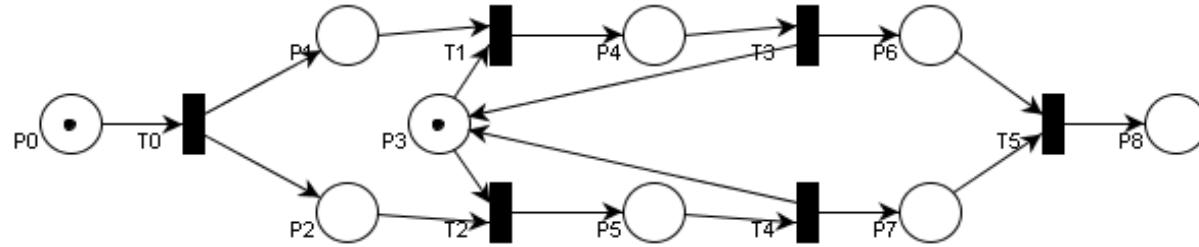


Double synchronization 3

How can this be programmed using Java language?

Verification

Reachability graph → Net states graph



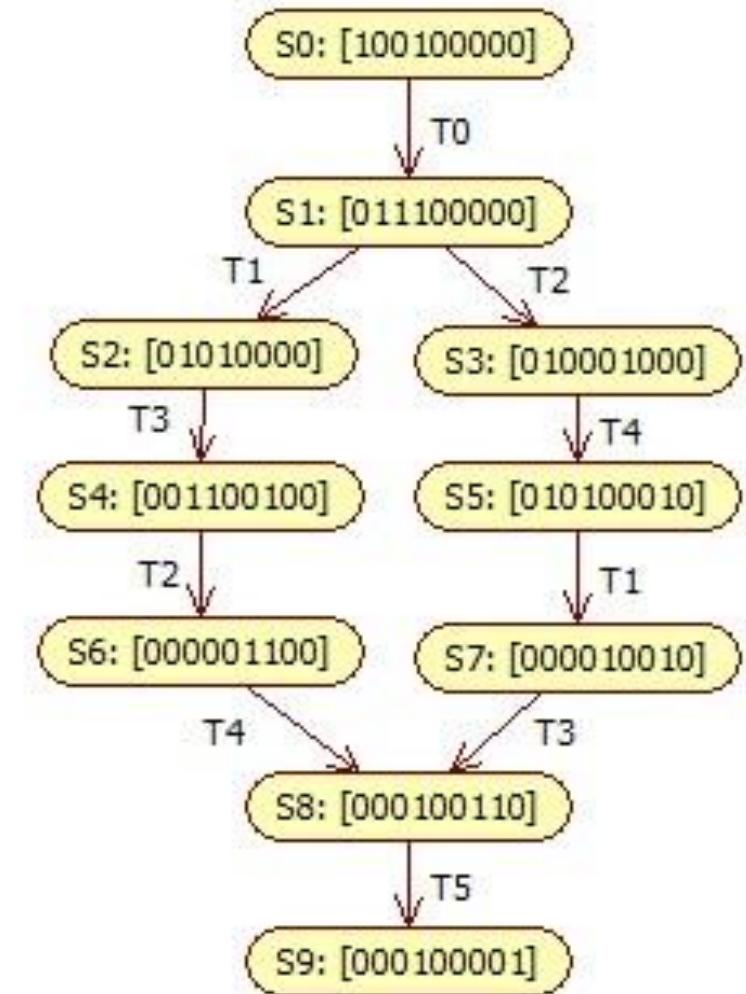
$S_i, i=0, 1, \dots, 9$; state identifiers

Testing: the evolutions follow all the possible traces. Set the appropriate initial markings.

$$\sigma = T_0 * (T_1 * T_3 * T_2 * T_4 + T_2 * T_4 * T_1 * T_3) * T_5 = \\ T_0 * T_1 * T_3 * T_2 * T_4 * T_5 + T_0 * T_2 * T_4 * T_1 * T_3 * T_5$$

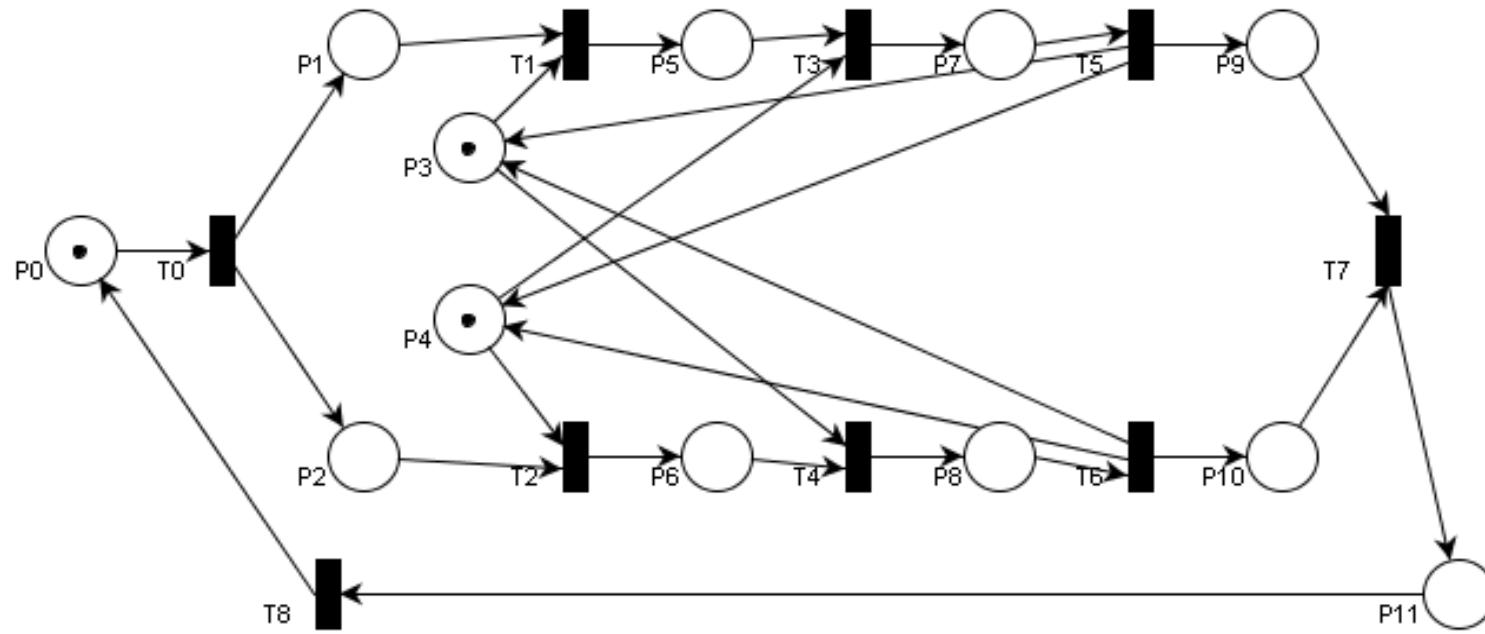
Net marking:

$[M(P_0), M(P_1), M(P_2), M(P_3), M(P_4), M(P_5), M(P_6), M(P_7), M(P_8)]$



State space dimension:
 $2^{|P|}$; $|P|=9$ is the cardinal of P
 Current state number: 10

Homework: build the reachability graph for the mutual exclusion example and analyze its behavior.



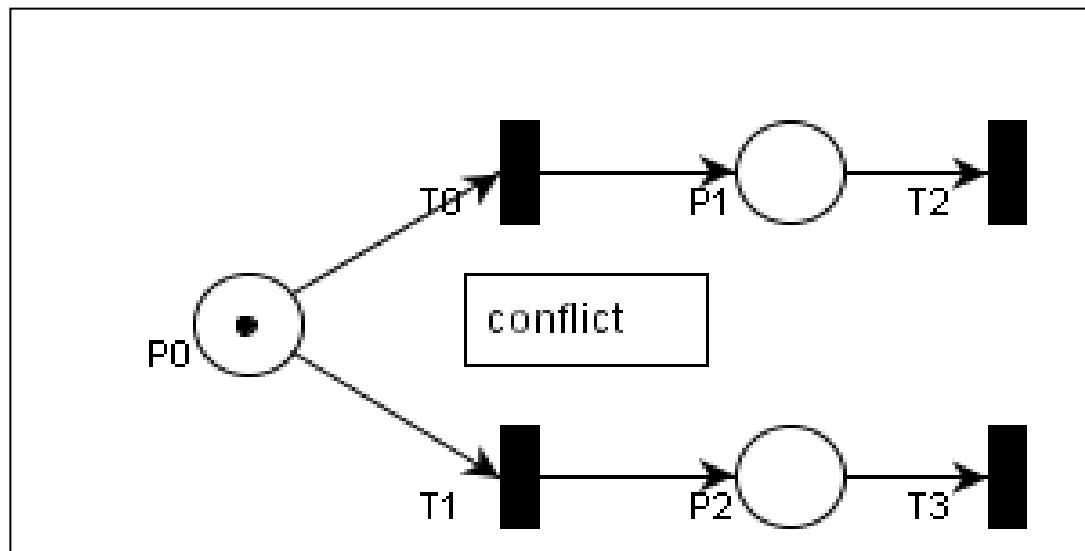
c. Stochastic PN

Random Petri Nets \leftrightarrow Stochastic Petri Nets (SPNs)

A *stochastic Petri net* is a five-tuple $SPN = (P, T, F, M_0, \Lambda)$ where:

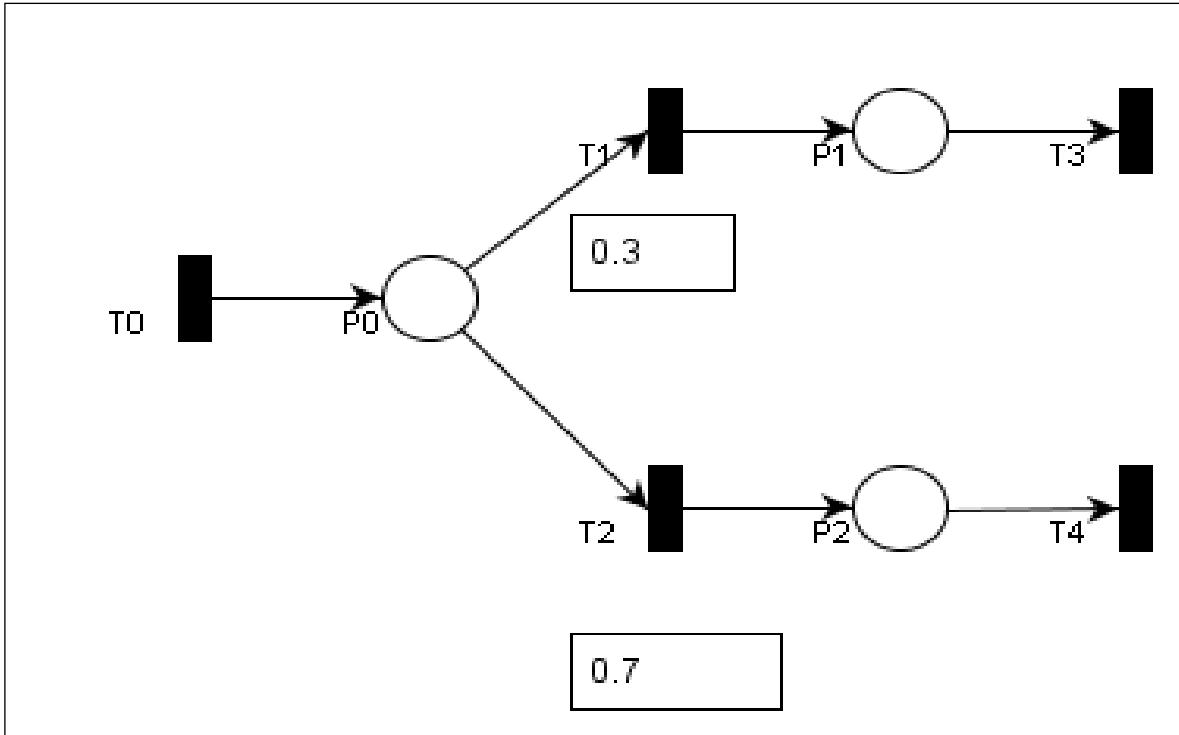
Λ : is the array of *firing rates* λ associated with the transitions.

The firing rate \rightarrow for set of random variables. These can be a function $\lambda(M)$ of the current marking.



Conflict

Which transition is executed?
→ selection



Conflict resolution – stochastic PN

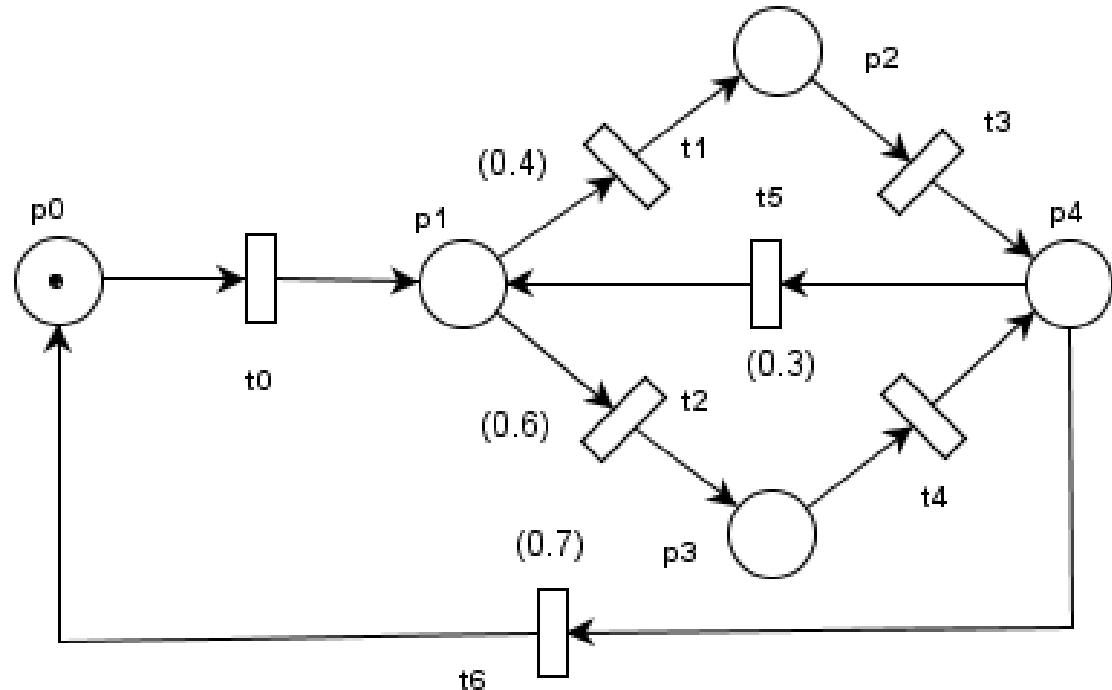
$$\text{prob}(T_1) + \text{prob}(T_2) = 0.3 + 0.7 = 1$$

How can this be programmed using Java language?

Transition	t1	t2	t5	t6
Probability	0.4	0.6	0.3	0.7

Reachability graph?

Particular SPNs are relevant
for buffer modeling.



*

END

*

2. Specification of RTAs (2.1,2.2,2.3)

2.4. Specification and Verification

of

RTAs with Time Petri Nets

Verification:

- Formal verification
- Simulation of the Real-Time Systems

Rational: complexity + quality

Contents

Specification and Verification of RTS with Petri Nets

- a. Ordinary PN
- b. Program flow description – Petri Net based Language
- c. Stochastic PN (SPN)
- d. 2.4 Timed and Time PN (TPN)
 - i. Timed PN
 - ii. Transformation of timed transition PN to timed place PN
 - iii. Enhanced Time Petri Nets
 - iv. Hybrid controllers with Discrete Events and Discrete Time + Interfaces
 - v. Alternative
 - vi. Deadlock avoidance with timings
 - vii. Constraints specification and verification
- e. Time Interval Petri Nets
- f. Delay Time Petri Nets (DTPNs)
- g. Applications of TPN and DTPN
- h. Time Petri nets with inhibitor arcs

Time

Time → augmented PNs

Taxonomy of time and classes of timed PN → temporal features

- Time
- discrete time → natural numbers
 - continuous time (represented by real numbers) → dens time

How van be the time measured? → By counting events → clock → *Real-Time clock*

- Time specifying
- sojourns times of tokens on places
 - firing delays of enabled transitions

Time → moment of time

- Place Time PN or Timed Place Petri Nets
- Transition Time PN or Timed Transition Petri Nets

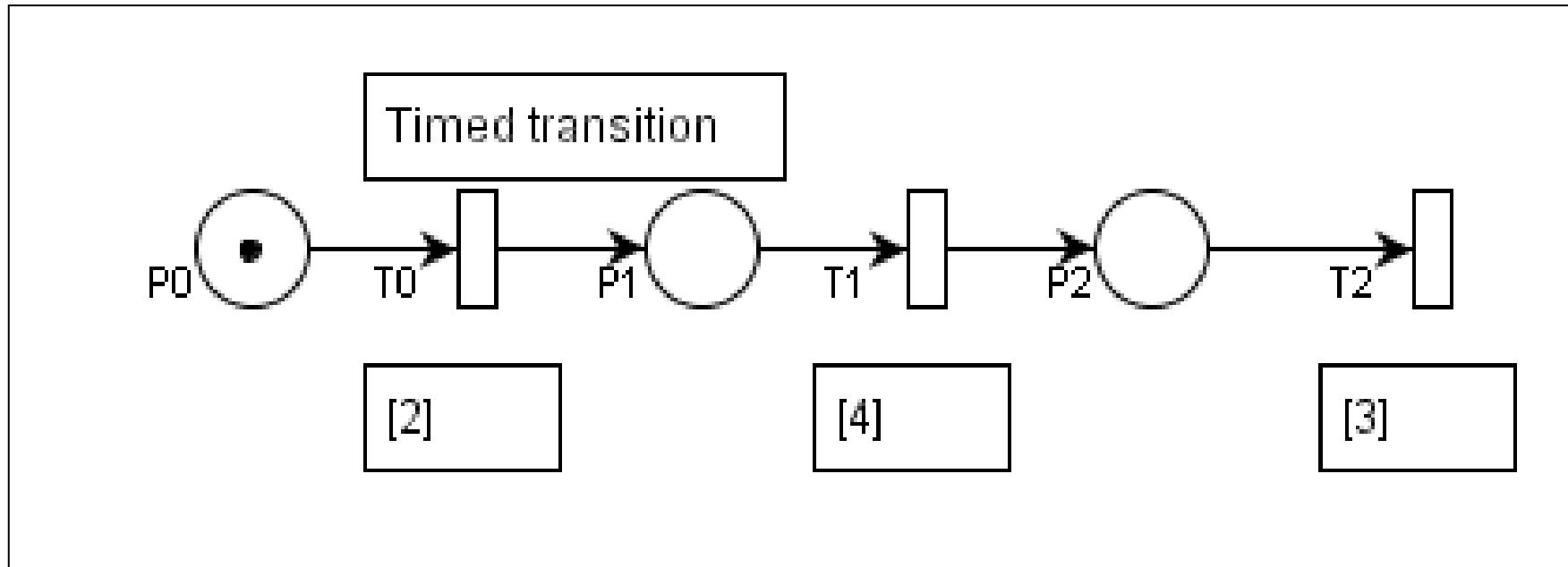
→ time interval – delayed (*scheduled*) enabled transition in the specified interval

- tokens with timestamp
- delay to extract the tokens from input places
 - delay to introduce the tokens to output places
- variable time intervals – variable delays

Timed Transition Petri Nets with:

- *Reserved tokens = preselection models*
 - verify if a transition is enabled
 - extract the tokens from its input places
 - wait the specified time
 - set the tokens in its output places
- *Unreserved tokens = race models*
 - find all the time all the enabled transitions
 - start a counter for each enabled transition
 - when a transition waiting time elapses, *if it is still enabled*, execute the transition
 - extract the tokens from its input places
 - set the tokens in its output places

2.4. Timed or Time PN



$$\sigma = T_0[2] * T_1[4] * T_2[3] = T_0(2) * T_1(6) * T_2(9)$$

How can this be programmed using Java language?

Notations:

$T_0[2]$ – relative time

$T_0(2)$ – absolute time

Delay significances:

- Wait = pure delay
- Activity with a relevant duration considered as a duration to change the state (i.e. modeled by place)

Time Petri Nets

What can a TPN model?

- A delay of an activity related to the previous one → $\text{wait}(time)$ → a delayed transition
- The occurring of an event at the specified moment of time → $\text{wait}(clockTime)$
- A fixed time execution of an activity → the state is changed after the execution is ended
→ $M(p)$ has assigned a fixed time
- A variable time execution of an activity → the state is changed after the variable duration of execution → $M(p)$ has assigned an interval
- An input event occurs during a specified interval → the corresponding transition has assigned a period for receiving the signal

What is used in implementation? → the pure waiting delay → $\text{wait}(time)$

The fixed or variable time execution of an activity is a consequence and not a demand!

This is used in verification and not directly in implementation.

In specification: when has to be performed something or to end the reaction to something (i.e. deadline).

What are the meanings of time for design, verification and testing?

TPN = Time Petri Nets

$N = (P, T, \text{Pre}, \text{Post}) ;$

$\text{PN} = (N, M_0) ;$

N_n – natural number set

$\text{TPN} = (N, M_0, SI)$

$SI : P, T \rightarrow N_n$

It is a mapping that assigns *relative static time intervals* to places or transitions.

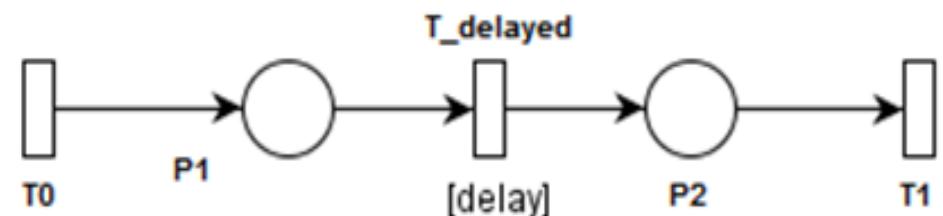
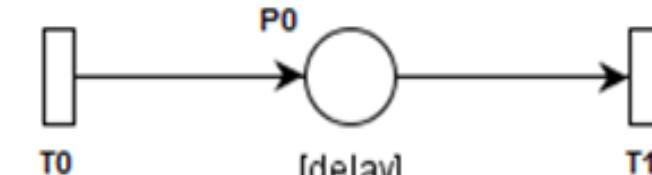
$SI(t) = [\text{eet}, \text{let}] \leftarrow$ Time Interval PN

- eet: earliest execution time
- let: latest execution time

If $eet = let \rightarrow$ pure fixed delay \leftarrow Timed PN

M_0 – initial marking

$M : P \rightarrow N_n ;$



Place timed PN could be transformed into Transition Timed PN and vice versa.

Net dynamics: execution of transitions.

Enabled transition – similar PN logic

Executable transition – it is chosen to be executed in its static time interval

Transition in execution – the net current time coincides with the time when the transition was scheduled for execution.

Places model activities.

Transitions model actions or events

→ the switch of the state

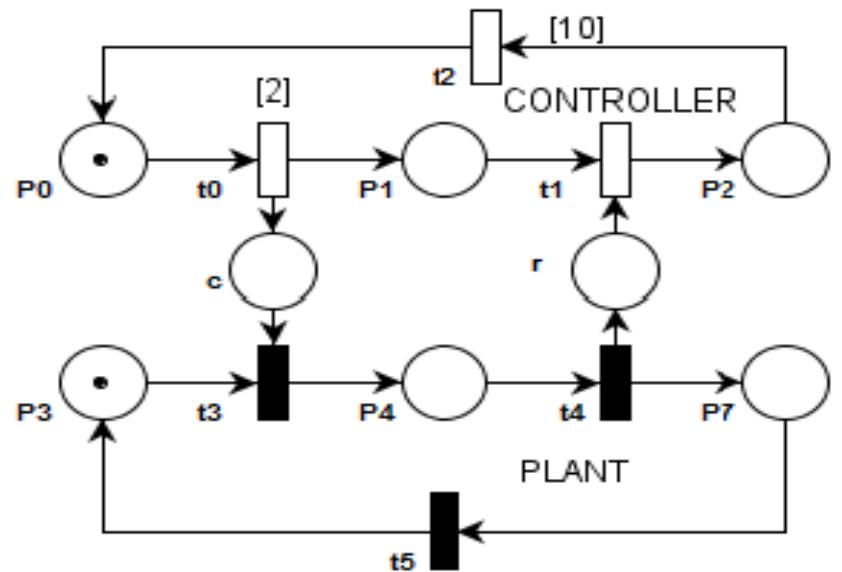
Reserved or unreserved tokens?

Enhanced Time Petri Nets (ETPN)

ETPN = Enhanced Time Petri Net

$$ETPN = (TPN; Inp, Out)$$

- It is an TPN with timed transitions and input/output reaction channels
- It has an input place set $Inp = \{p_{i1}, \dots, p_{im}\}$ where can be injected tokens by an external (relative to TPN) component – They are used to catch external input events.
- It has an output place set $Out = \{p_{o1}, \dots, p_{on}\}$ that can be read by an external (relative to TPN) component. – They are used to signal control events.
- ‘ φ ’ denotes NO input or output event



ETPN executor

It receives external events through input channels. It immediately (asynchronously) injects in the corresponding places tokens.

It receives clock events and counts down (decreases) the delay counters assigned to enable and scheduled transitions.

When an output place receives a token, the event is immediately signalled through the corresponding channel.

ETPNL

This can be described by ETPNL

(Enhanced Time Petri Nets based Language)

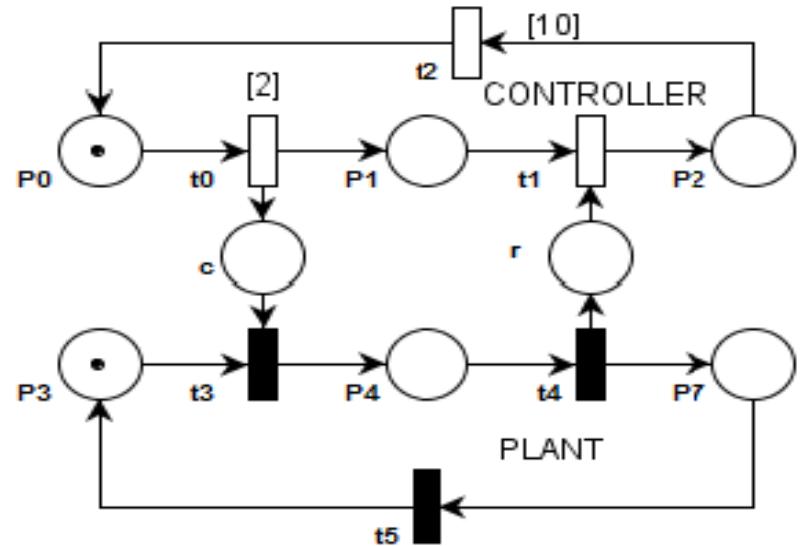
Added notations: $t_k[a,b]$

- t_k is a transition identifier
- a is an input channel chosen from the transition input place set; there is expected an external event to be signaled; when a natural number is set in this position, it corresponds to a pure delay.
- b is an output channel chosen from the transition output place set and it is used to signal an event to an outside destination.
- The symbol ‘ φ ’ is used in the position of a or b when no input or output event is specified.

For the *controller/plant* example:

$$\sigma_{\text{controller}} = ((t_0[2,c]^* t_1[r,\varphi]) \# t_2[10,\varphi])^n$$

$$\sigma_{\text{plant}} = ((t_3[c, \varphi]^* t_4[\varphi,r]) \# t_5[\varphi,\varphi])^n$$



Place c is an output channel for *controller* and it is an input channel for *plant*.

Place r is an input channel for *controller* and an output channel for *plant*.

What is the period of this system if the plant has no delays?
 $T_{\text{system}} = ?$

Example 1: sequential controller

$$\sigma = t_0[\tau_0, c_1] * t_1[\tau_1, \varphi] * t_2[\tau_2, c_2]$$

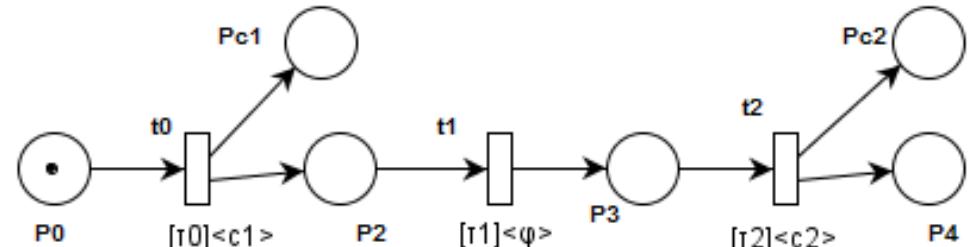
' φ ' denotes NO input or output event
 τ_0, τ_1 and τ_2 are pure delays.

Example 2: cyclic controller

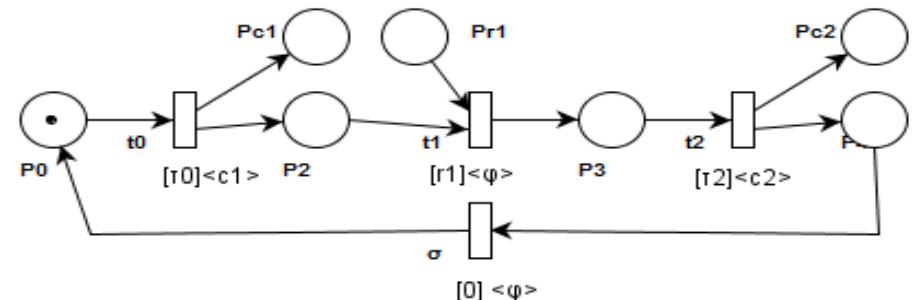
$$\sigma = (t_0[\tau_0, c_1] * t_1[r_1, \varphi] * t_2[\tau_2, c_2]) * t_\sigma[0, \varphi]^n$$

$$\sigma = (t_0[\tau_0, c_1] * (t_1[r_1, \varphi] * t_2[\tau_2, c_2])) * t_\sigma[0, \varphi] * \sigma$$

$$\sigma = (t_0[\tau_0, c_1] * t_1[r_1, \varphi] * t_2[\tau_2, c_2]) \# t_\sigma[0, \varphi]$$



ETPN model of the sequential controller



TPN model of a cyclic controller

Problem 1

Specification: Each time when one of the events e_1 or e_2 occurs, signal e_3 after 5 time units (t.u.).

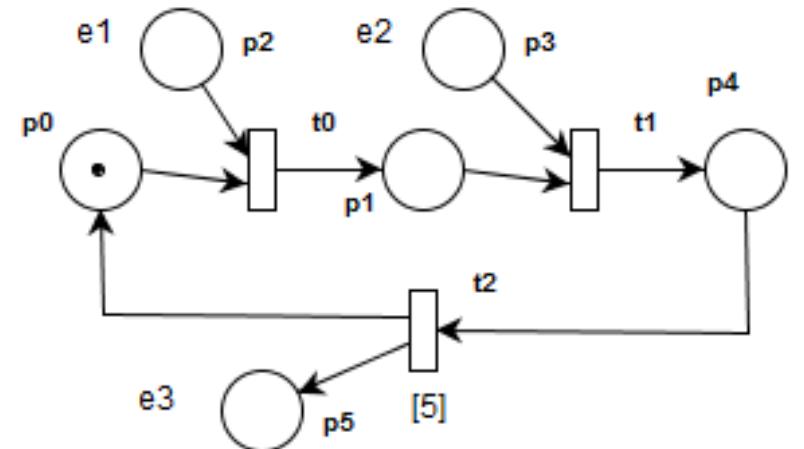
Is it correct?

$$\sigma = ((t_0[e_1, \varphi] * t_1[e_2, \varphi]) \# t_2[5, e_3])^n =$$

$$= ((t_0[e_1, \varphi] * t_1[e_2, \varphi]) \# t_2[5, e_3]) * \sigma$$

Correct solution?

$$\text{Inp} = \{p_2, p_3\}; \text{Out} = \{p_5\}$$



The operating system + environment provide for **implementation** the instructions:
 $\text{wait}(e_1); \text{wait}(e_2); \text{notify}(e_1); \text{notify}(e_2); \text{notify}(e_3);$

$$\sigma = ((t_0[\varphi,\varphi]^*((t_1[e_1,\varphi]) \# t_3[5,e_3])^m \& (t_2[e_2,\varphi] \# t_4[5,e_3])^n)$$

Implementation?

Available: *wait(e₁)*; *wait(e₂)*; *notify(e₁)*; *notify(e₂)*; *notify(e₃)*;

Problem 2

Specification: If one time one of the events e₁ or e₂ occurs, signal e₃ after 5 time units (t.u.) and exit.

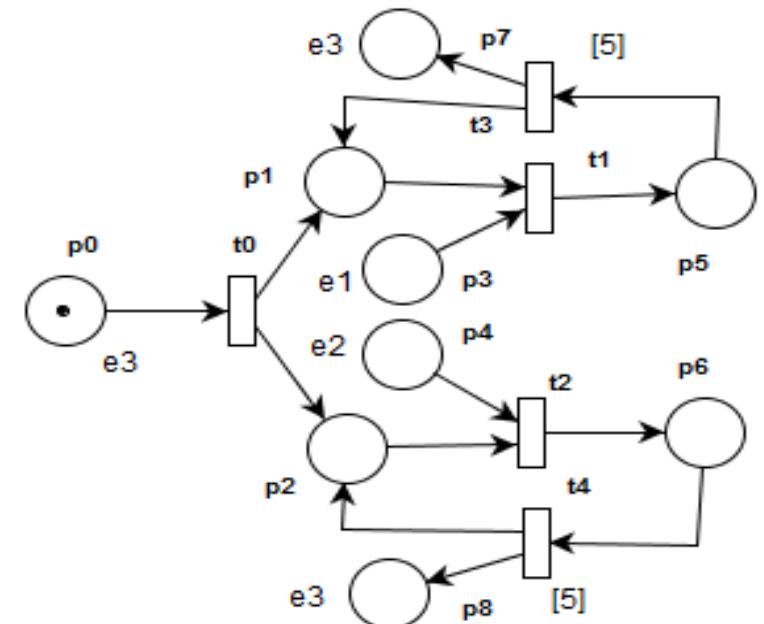
$$\sigma = (t_1[e_1,\varphi]) + t_2[e_2,\varphi]) * t_3[5,e_3]$$

Available: *wait()*, *wait(e₁)*; *wait(e₂)*; *notify(e₁)*; *notify(e₂)*; *notify(e₃)*;

Draw the ETPN

Write the Java implementation. (Notice: *wait()?*)

Draw the ETPN model of the provided ETPNL expression:
 $\sigma = (t_1[e_1,\varphi]) + t_2[e_2,\varphi]) \# t_3[5,e_3]$



Hybrid Controller with Discrete Event Interfaces:

Inp= $\{p_{i1}, p_{i2}, p_{i3}\}$

Out= $\{p_{o1}, p_{o2}, p_{o3}\}$

Discrete Time Interfaces:

Inp= $\{ch_i\}$

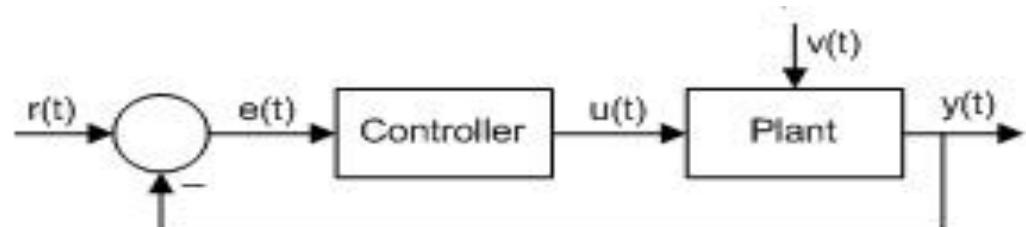
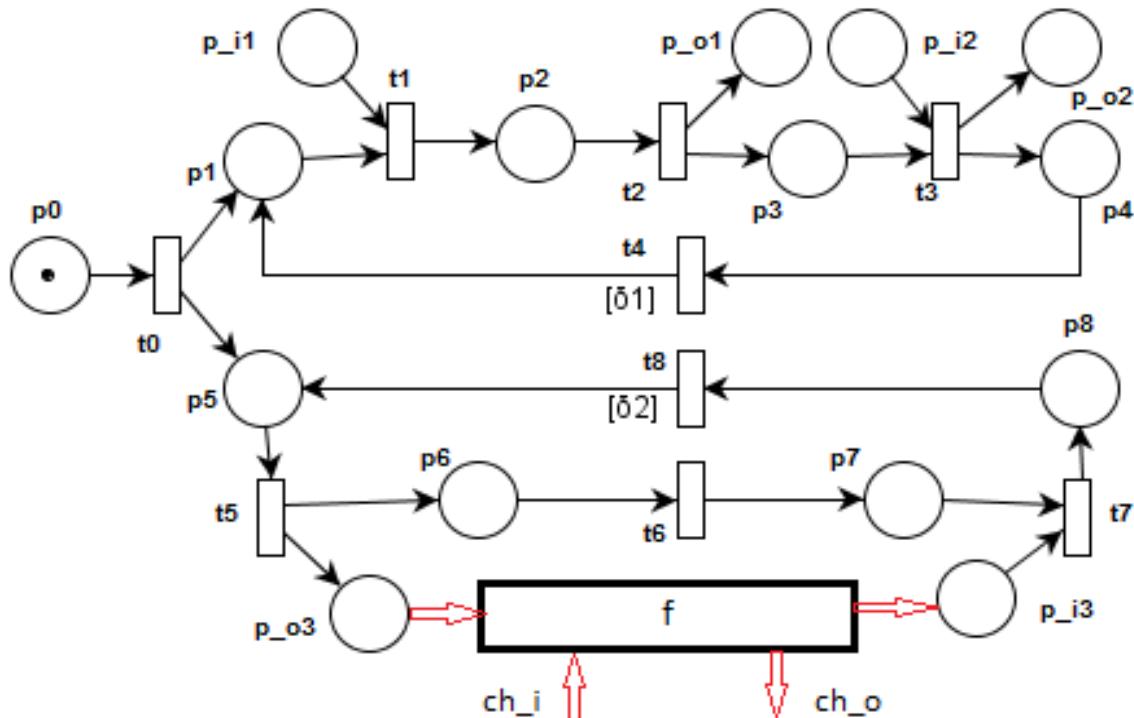
Out= $\{ch_o\}$

f : a proportional controller

$ch_o = K \cdot (ref - ch_i)$; K fixed.

$$\sigma = t_0[\varphi; \varphi] * (((t_1[r_1; \varphi] * t_2[\varphi; c_1] * t_3[i_2, o_2]) \# t_4[\delta_1; \varphi])^m \&$$

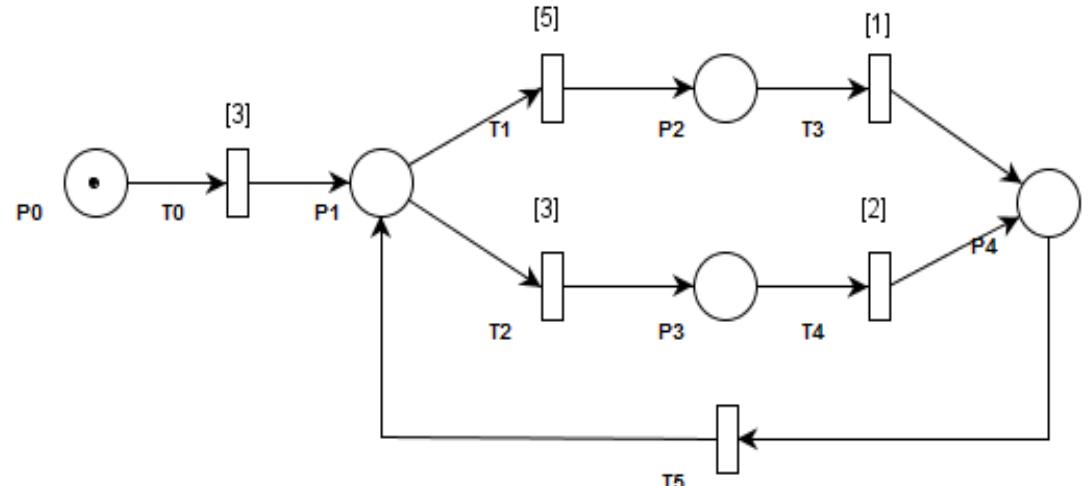
$$((t_5[\varphi; o_3] * t_6[\varphi; \varphi]) * t_7[i_3; \varphi]) \# t_8[\delta_2, \varphi])^n)$$



TPN Example

Conflict → Different semantics:

- Execute the soonest;
- Chose one according to given probability and reserve the tokens; ← stochastic PN
- Execute the latest



Transition Time (Timed) Petri Nets (used further)

$$\mathbf{M} = [M(p_1), M(p_2), \dots, M(p_m)]^T$$

SI : $P \rightarrow N^n$ it associates delays for the extraction of the tokens from places

SI : $T \rightarrow N^n$ it associates a delay for transition executions

$$\sigma = T_0[3;\varphi]^*((T_1[5;\varphi]*T_3[1;\varphi])\&(T_2[3;\varphi]*T_4[2;\varphi]))\#T_5[0;\varphi])$$

Chose a semantic and write the sequence of transitions seen on the screen!

Token reservation when a transition is chosen for execution! ← preselection model

TPN model implementation:

- Activities without relevant durations (relative to deadlines) → synchronous approach
- Activities with relevant durations (relative to deadlines) → asynchronous approach

P/T Petri Net Implementation (i.e. EXECUTOR)

P/T PN simulator:

Input: Pre, Post, M_0 , P, T

Output: M, transition executed

Initialization: M = M_0 ; execList

for ever do

// find all the enabled (executable) transitions:

execList = empty; // the list of executable transitions at the current time

for all t of T do

exec = true;

for all $p \in {}^0t$ do // $p \in P$?

if ($M(p) < \text{Pre}((p,t))$) **then** exec = false;

end for

if (exec) add t to *execList*

end for

* **for all executable transitions t from the list *execList* do**

// execute the transition t:

$M = M - \text{Pre}[t] + \text{Post}[t]$; //column t of the corresponding matrix

write M and t;

end for

Untimed PN

No conflict transitions and free executable transitions.

The algorithm is built for the case when there are no conflict transitions.

Modify the algorithm for the case when there are conflicts.

Analyze if the simulation is fair.

Relative to the TPN the following *semantics* are used.

The TPN controller models are deterministic and fulfill the following assumptions:

1. The TPN has no conflicts or free elections.
2. If more than one timed transition is executable from the same marking, the transition with the shortest delay is first chosen for execution;
3. If the TPN model has conflicts or free elections, the following semantics are accepted: the order of transitions chosen for execution is given by their index;
4. The system works with reserved tokens. A transition that started the execution cannot be cancelled by another one with a shorter delay;
5. The places of the sets P_C and P_R are used exclusively only for input or output operations respectively.
6. The transitions correspond to actions and they involve no durations or delays.

Rational of simulation:

- build a tool to verify the behavior of a modeled system
- build a framework to construct program

TPN Implementation (fixed transition delays) - EXECUTOR

TPN simulator:

← synchronous approach = no transition durations

Input: Pre, Post, M_0 , P, T, D; // D transitions delay vector

Initialization: $M_t = M = M_0$, execList, pendingTransList; // M_t – temporar marking
for ever do

//find all the enabled (later executable) transitions:

execList = empty; // the list of executable transitions at the current time

do

for all t of T **do**

exec = true;

for all $p \in P$ **do** // or $p \in {}^0t$?

if ($M(p) < \text{Pre}((p,t))$) **then** exec = false;

end for

if (exec) add t to execList

end for

* choose an executable transition t from the list execList;

* start the execution of the transition t:

$M = M - \text{Pre}[t]$; //column t of the corresponding matrix

$M_{\text{temp}} = M - \text{Pre}[t] + \text{Post}[t]$; //used for strict transition execution rule

add t to pendingTransList;

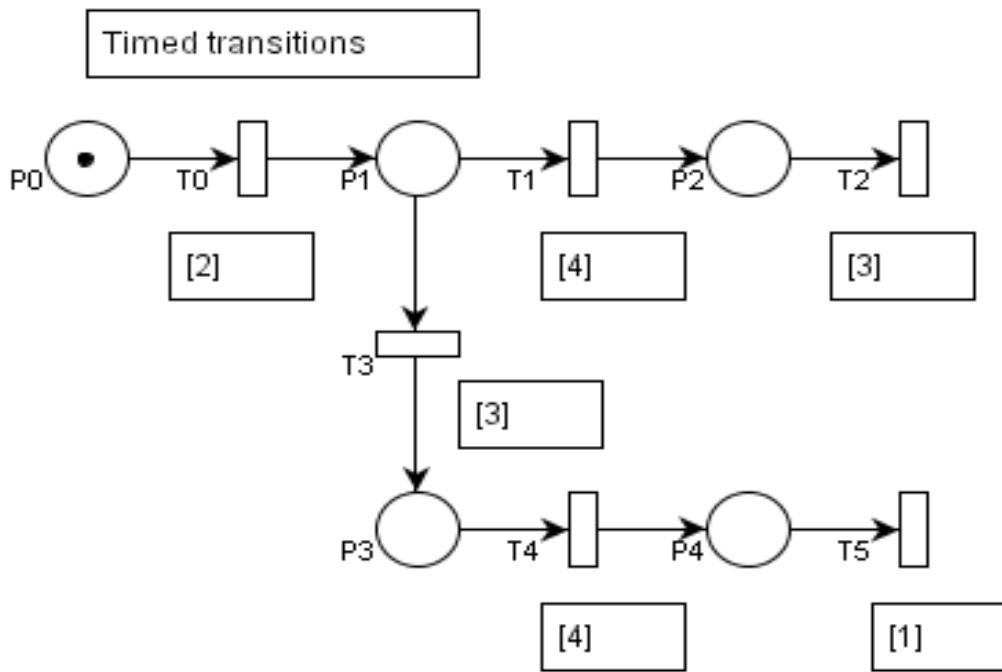
$\text{Delay}[t] = D[t]$;

A started transition extracts the token from the input places.

When delay time will expire the transitions are finished injecting the tokens into output places.

```
while(execList is not empty)
  * decrease the delay times from the pendingTransList;
    if (Delay[t] becomes 0) then *end execution of t:
      M = M + Pos[t];
      * remove t form pendingTransList;
      write M and t://signal t event for control
      wait(1 t.u.);
end for
```

TPN properties and examples



$$\begin{aligned}\sigma &= T_0[2] * (T_1[4] * T_2[3] + T_3[3] * T_4[4] * T_5[1]) = \\ &= T_0(2) * T_3(5) * T_4(9) * T_5(10)\end{aligned}$$

Using different semantics will provide different results.

Selection:

$$\sigma = T_i + T_j$$

Sequence:

$$\sigma = T_i \cdot T_j$$

E.g.:

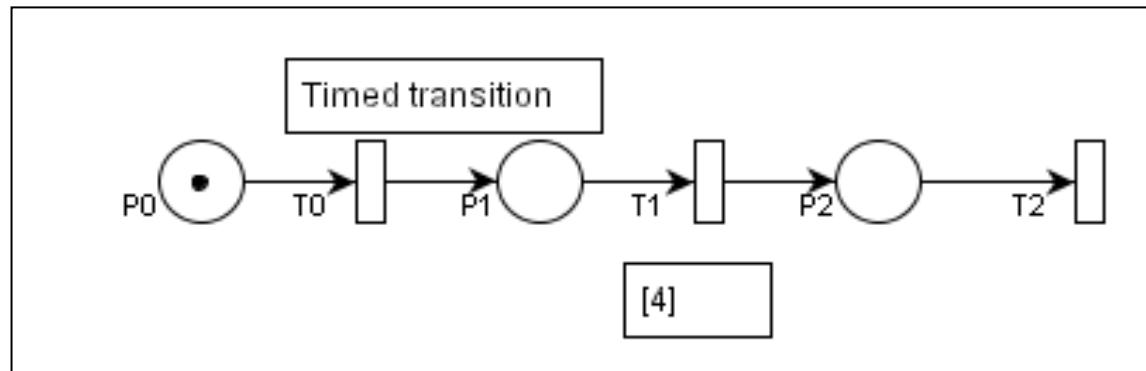
$$\sigma = T_h \cdot (T_i + T_j)$$

Different semantics:

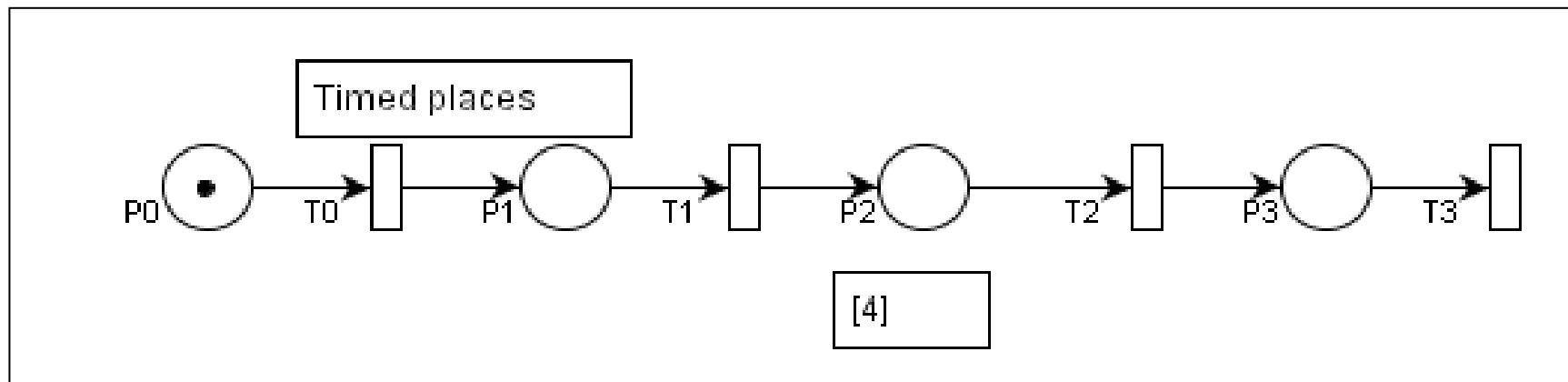
- Execute the soonest;
- Choose one according to given probability and reserve the tokens; ← stochastic PN
- Execute the latest

Notations: $t[\text{delay}]$ ← relative time from the previous event;
 $t(x)$ ← absolute time; x is counted from the time of the net execution starting.

Transformation of timed transition PN to timed place PN

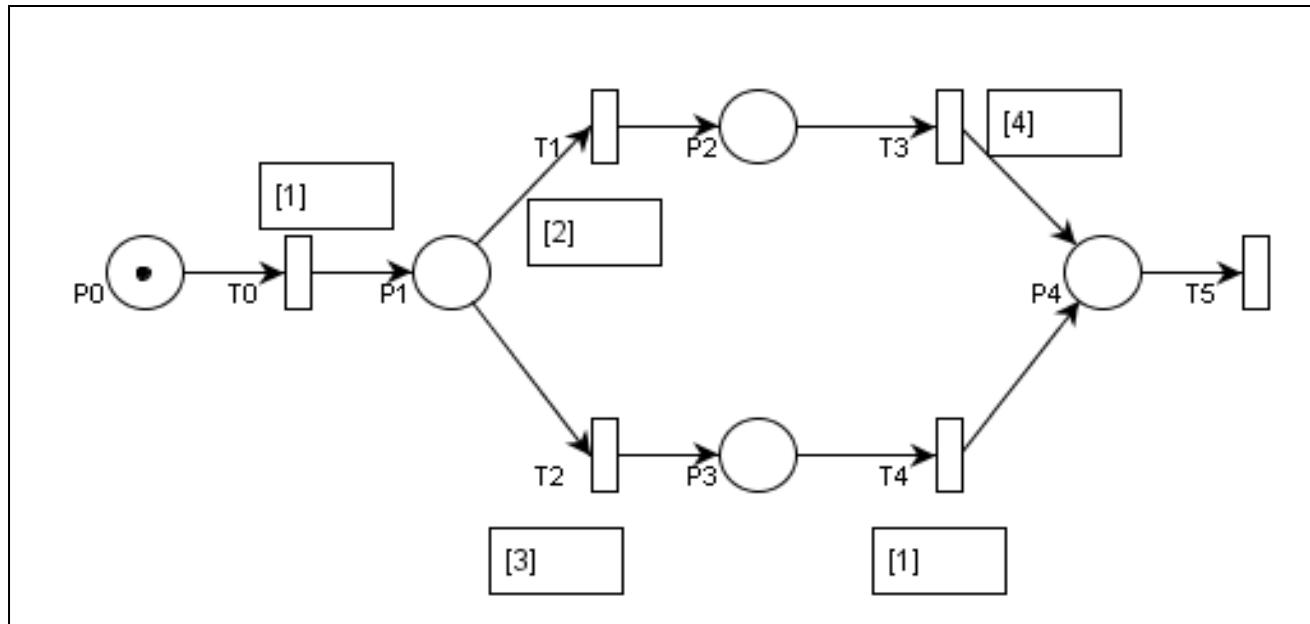


When the transformation is acceptable?
What is the meaning of the delay?



$$T_1[4] \leftrightarrow \{T_1[0], P_2[4], T_2[0]\}$$

Alternative



Conflict: T_1, T_2

Chose a semantic

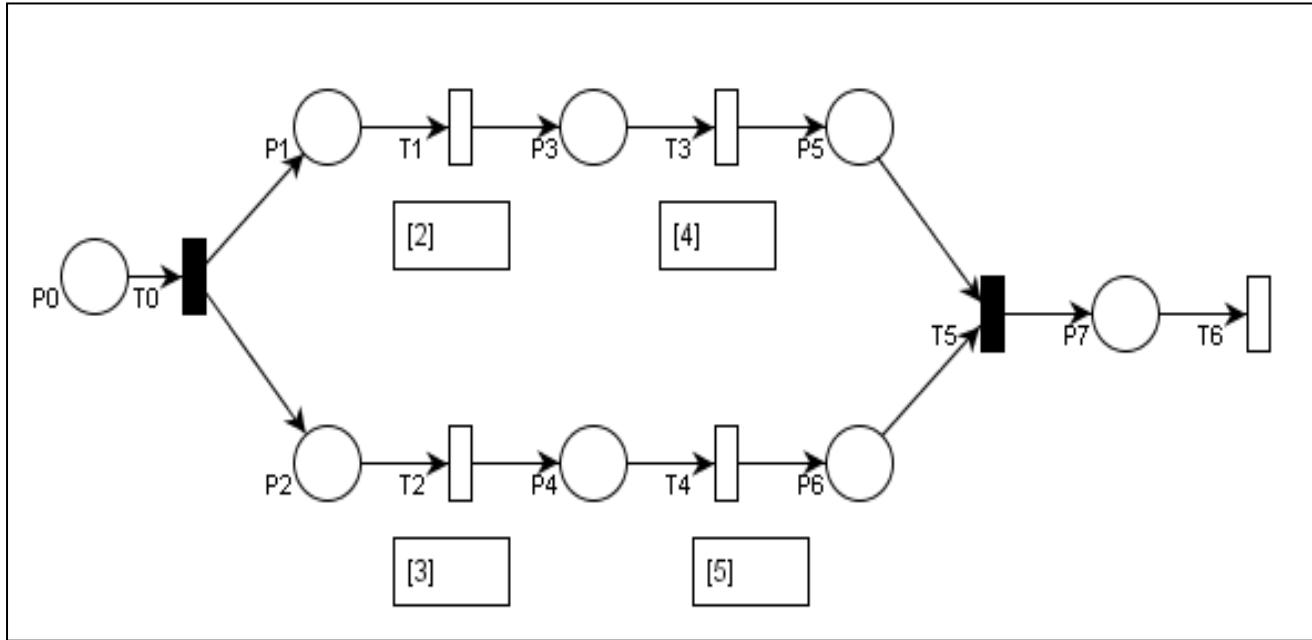
When is the transition T_5 executed?

$$\sigma = T_0[1] * (T_1[2] * T_3[4] + T_2[3] * T_4[1]) * T_5[0] = T_0(1) * T_1(3) * T_3(7) * T_5(7_+)$$

→ $T_5(4_+)$

Synchronous approach = NO durations

Concurrent execution



There are:

- ***no durations,***
- ***only pure delays.***
- ➔ ***wait(delay)***

When is the transition T_6 executed?

Notation:

Concurrent execution
of T_i and T_j :

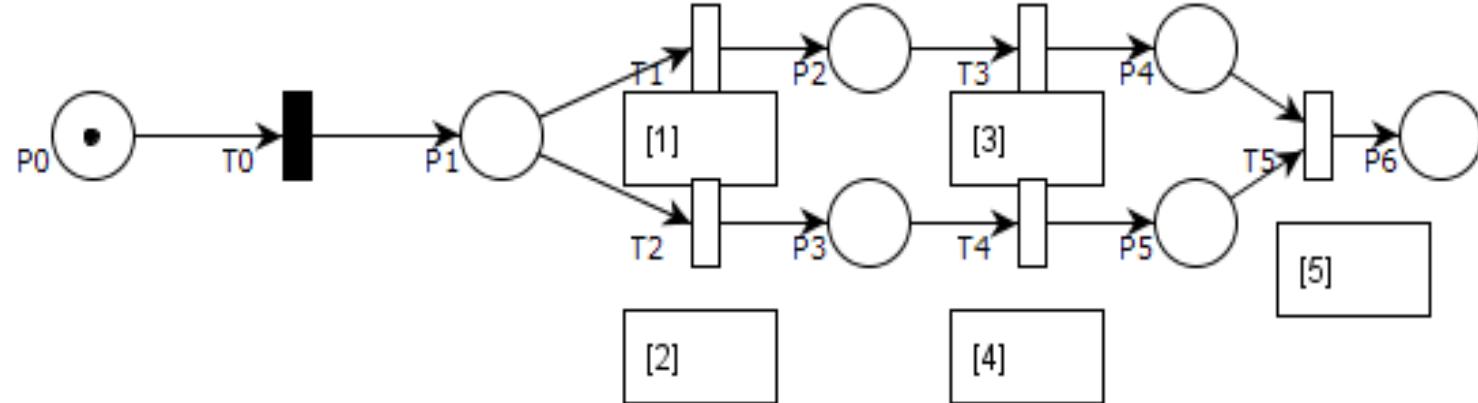
$$\sigma = T_i & T_j = ?$$

- fixed delays ← no intervals

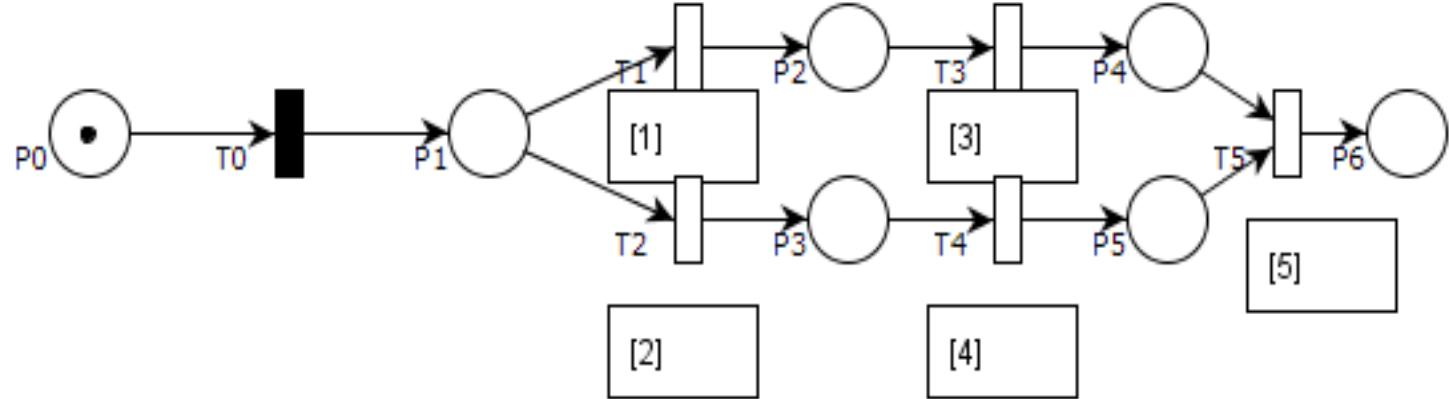
When is the transition T_5 executed?

$$\begin{aligned} \sigma = T_0[0] * ((T_1[2]*T_3[4]) \& (T_2[3]*T_4[5])) * T_5[0] * T_6[0] = \\ T_0(0) * ((T_1(2)*T_3(6)) \& (T_2(3)*T_4(8))) * T_5(8_+) * T_6(8_{++}) \end{aligned}$$

➔ $T_6(8_{++})$ T_0 and T_5 were supposed to be executed immediately.



When is the transition T_5 executed?



$$\sigma = T_0(0_+) * ((T_1[1]*T_3[3]) + (T_2[2]*T_4[4])) * \delta$$

→ 6 is the blocking action

$$\sigma = T_0(0_+) * T_1(1) * T_3(4) * \delta(5_+)$$

The system is blocked at $t=5_+$ t. u. and remains in this state forever.

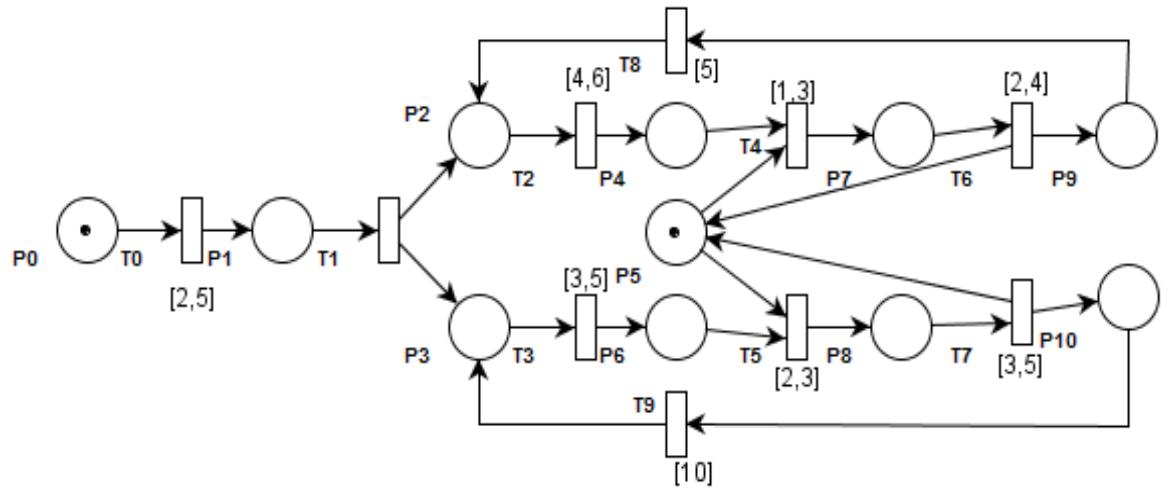
⇒ ***T₅ is not executed – it is never enabled.***

Time (Interval) Petri Nets (T(I)PN) \leftarrow TPN

$$\mathbf{N} = (P, T, F, W)$$

$$\mathbf{PN} = (\mathbf{N}; M)$$

- $F \subseteq (P \times T) \cup (T \times P)$
finite *set of arcs*,
- $W : F \rightarrow \{0,1\}$
weight of arcs,
- $M_0 : P \rightarrow \mathbb{N}_n$,
- $PN = (P; T; F; W; M_0)$ - ordinary Petri Net



$$\mathbf{TPN} = (\mathbf{PN}; SI)$$

$$SI : T \rightarrow SI : T \rightarrow \mathbb{N} \times \mathbb{N}$$

Interval semantics:

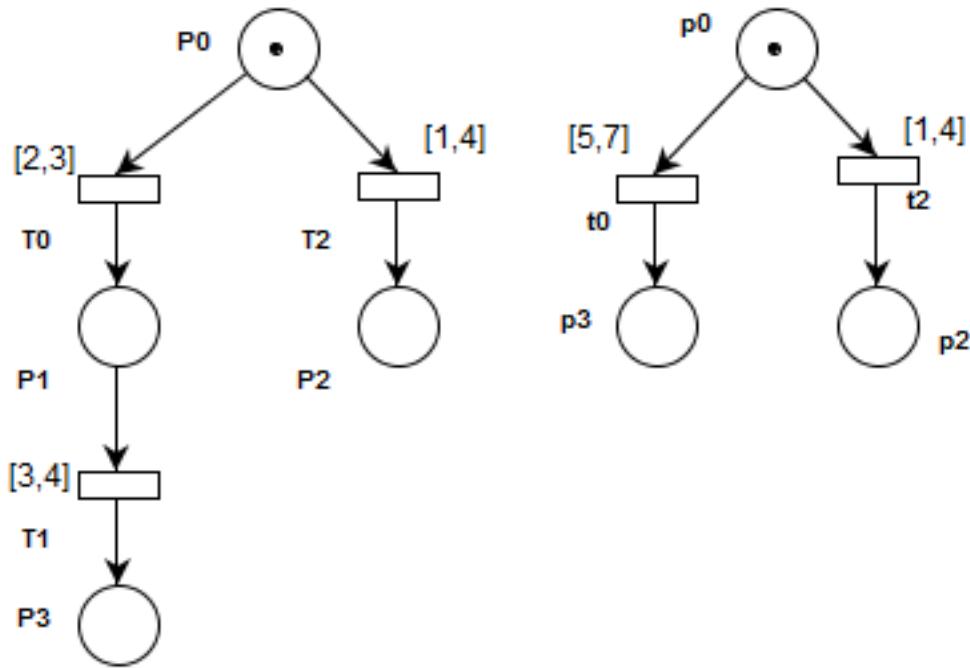
- variable delay of transition executions (\mathbb{N} no negative natural numbers),
- off line schedule or
- online selection of the execution times.

Activities with variable delays:

- search in a file, or in data structure,
- message transmission,
- scheduling,
- mechanical moves etc.

T(I)PN reduction

Transition fusion
Are they equivalent?



When are two PNs, TPNs, T(I)PNs etc. equivalent?

What are the criteria to evaluate the equivalence?

- They reach the same final state? (i.e. marking)
- They execute the same sequences of some transitions?

If they are TPNs, they have to execute the transitions at the same times!

Delay Time Petri Nets (DTPNs)

$$(DTPN) = (P, T, F, W, M_0, SI, SIF)$$

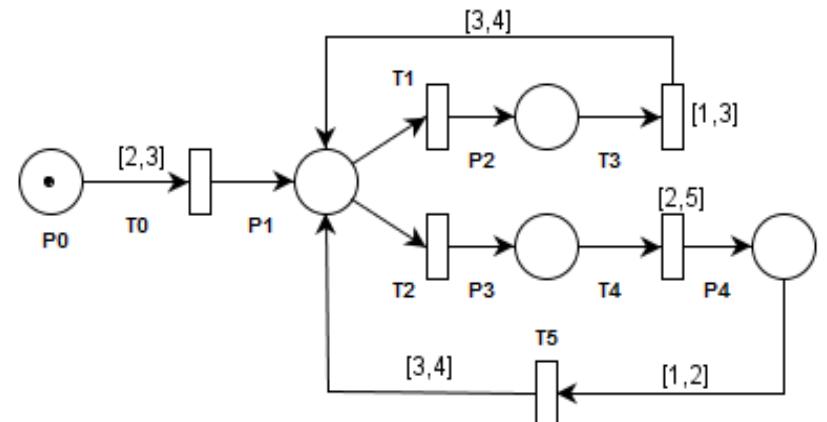
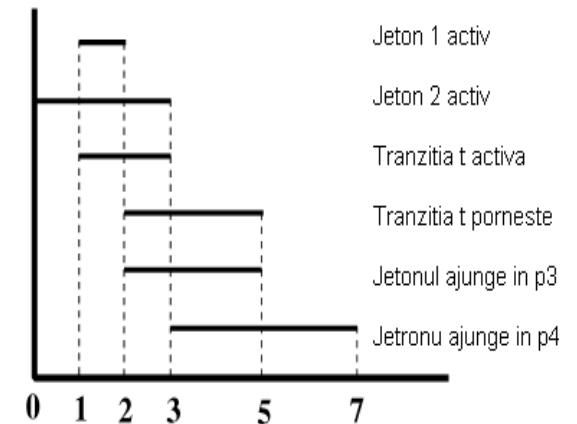
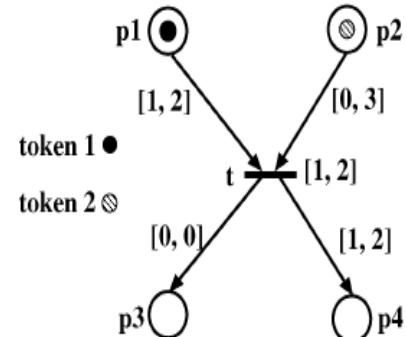
where:

- TIPN = (P, T, F, W, M_0, SIM)
- $F \subseteq (P \times T) \cup (T \times P)$
- $SIF : F \rightarrow Q \times (Q \cup +\infty)$;
- SI: assign static intervals
- Q non-negative rational numbers

What are the potential uses of DTPN?

What can DTPN model?

- schedule the transition executions
- synthesise a controller to a given model



Flexible Galvanization Line

M: mobile robot that moves containers. The diving move is ignored.

R_0, R_n : conveyors

$R_i, i=1, \dots, n-1$: basins (sinks) resources;

d: conveyor sensor – a container arrived;

c: control signal {move right, left, halt}

s: sensor signaling the reaching the position in front of a basin.

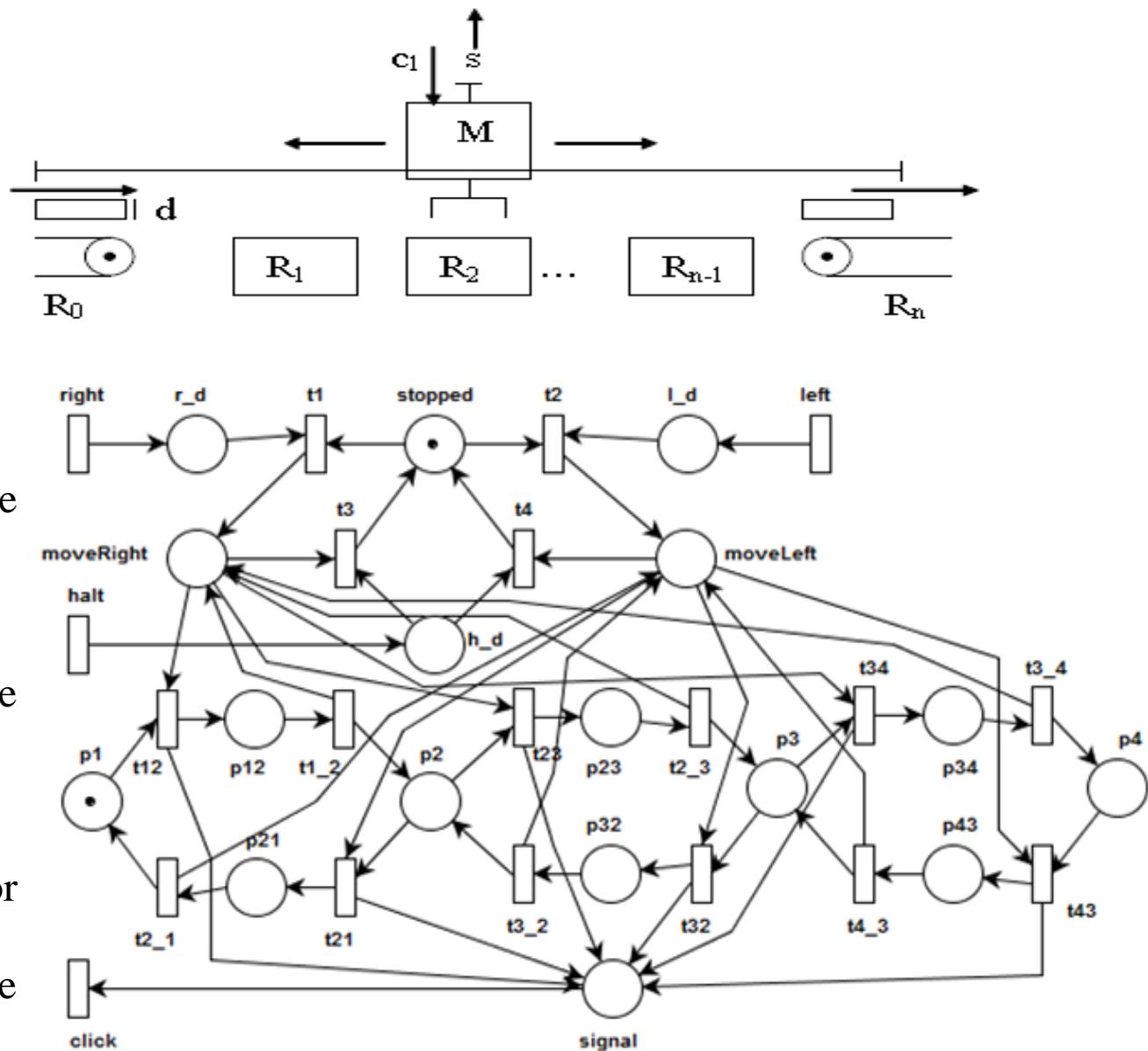
Robot specification:

Move left or right and stop in front of the next basin.

Controller specification:

Receive demands from the operator for moving left and right.

Signal the operator when the moves were performed.



Processing technology:

A container has to spend specified durations in basins according to:

$$\text{E.g.: } R_i[60] * R_j[80] * R_k[40]$$

Real-Time Systems

→ R-T requirements

1) *robot positioning = moving tolerance*

→ positioning tolerance = 10 cm

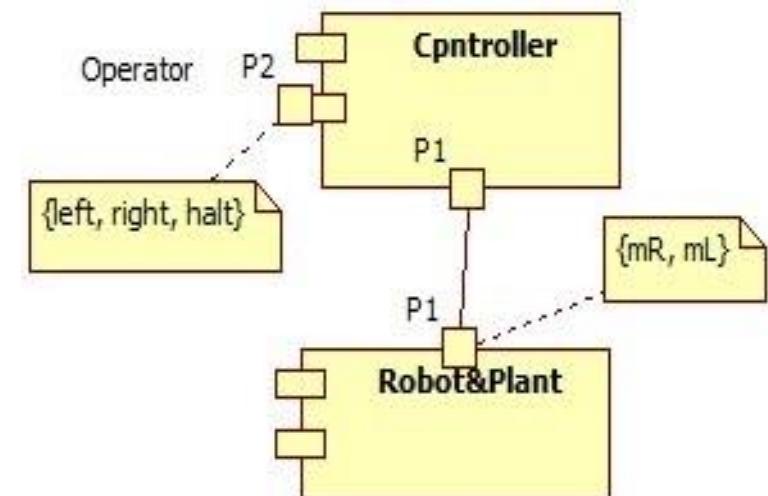
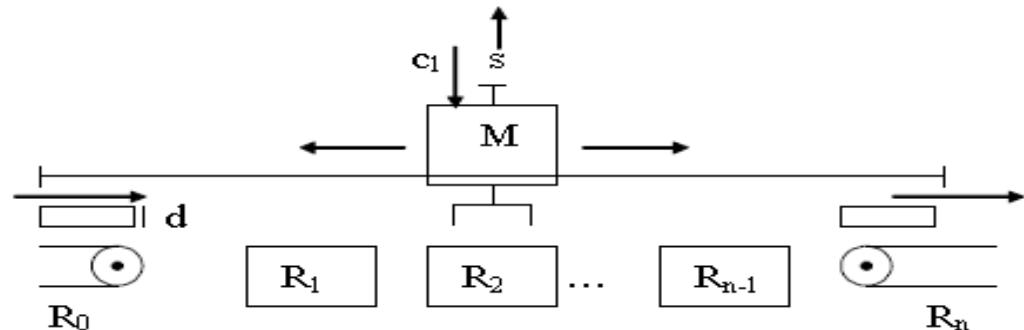
- robot speed = 20 cm/sec

→ positioningDeadline = 0.5 sec.

2) *Resource occupancy* = the time spent by a container in basin

E.g.: the duration spent in R_i has to be 60 sec, but not longer than 75 sec. → processingDeadline=15 sec.

For another basin can be different.



Processing mode:

- Single container processed = monotasking = single-tasking processing
- Multi container processing = multitasking

Four basins modeled by the places p_1, p_2, p_3 and p_4 .

Plant interface:

Inp = {mR, mL} // move Right, Left

Out = {mR, mL, sg} // signal

Controller interface:

Inp = {r_d, l_d, h_d}

Out = {mR, mL}

Operator demands: right, left, halt.

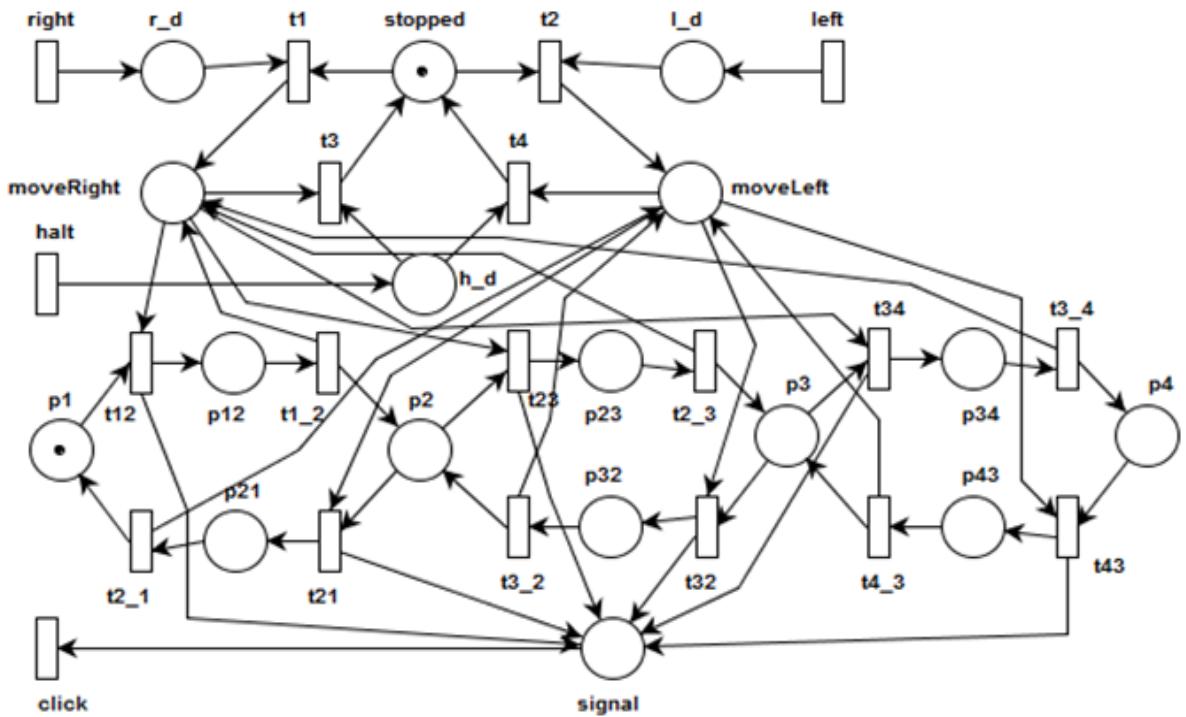
Controller ETPNL model:

$$\sigma_c = ((t_1[r_d, \varphi] * t_3[h_d, \varphi]) + (t_2[l_d, \varphi] * t_4[h_d, \varphi])) * \sigma_c$$

Plant model:

$$(t_{12}[mR, sg] * t_{1_2}[5, mR]) * ((t_{21}[mL, sg] * t_{1_2}[5, mL]) + (t_{23}[mR, sg] * t_{2_3}[5, mR])) * ((t_{32}[mL, sg] * t_{3_2}[5, mL]) + (t_{34}[mR, sg] * t_{3_4}[\varphi, mR])) * (t_{43}[mL, sg] * t_{4_3}[5, mL])$$

The move duration from a basin to its neighbor lasts 5 t.u.



Node	Represent	Modeling
p_1, p_2, p_3, p_4	Basin R_x ; $x=1,2,3,4$	Robot position
		Container state
p_{12}	Robot move	Moving from p_1 to p_2
p_{2_1}	Robot move	Moving from p_2 to p_1
t_{12}	Robot start the move	from p_1 to p_2
t_{1_2}	Robot move stop	Arrival in p_2
t_{21}	Robot start the move	from p_2 to p_{21}
t_{2_1}	Robot move stop	Arrival in p_1

Homework: describe the operator task for processing completely a container.
Add sensors and specify the application for a completely automatic control (execution of a technology).
Specify the *multiprocessing implementation of 2 containers*.

Real-time analysis of application specification

Can the system fulfill the deadline requirements? → temporal verification
Temporal analysis: operator + controller + robot/plant

Implementation

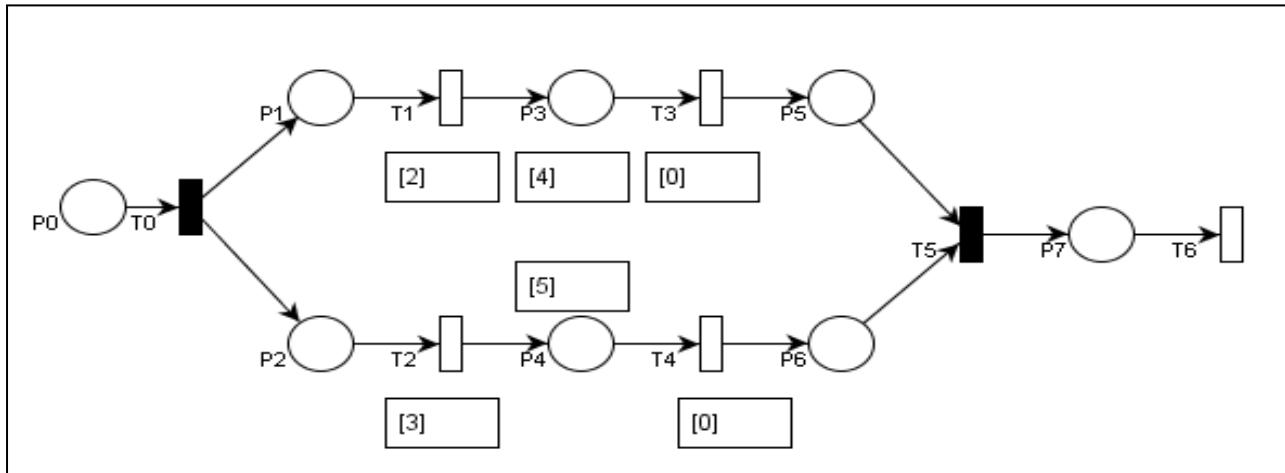
Controller implementation: $\sigma_c = (t_1[r_d, \varphi] * t_3[h_d, \varphi]) + (t_2[l_d, \varphi] * t_4[h_d, \varphi])$

Are the deadlines implemented?

What are temporal features that have to be implemented?

Asynchronous approach

Concurrent execution on the same processor

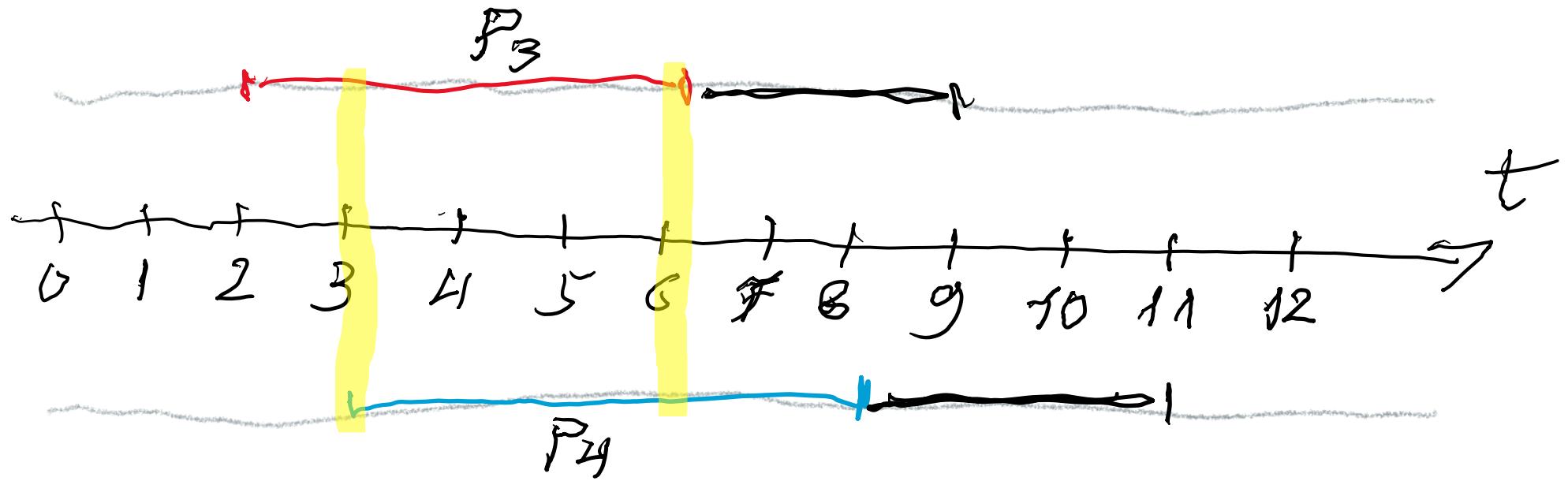
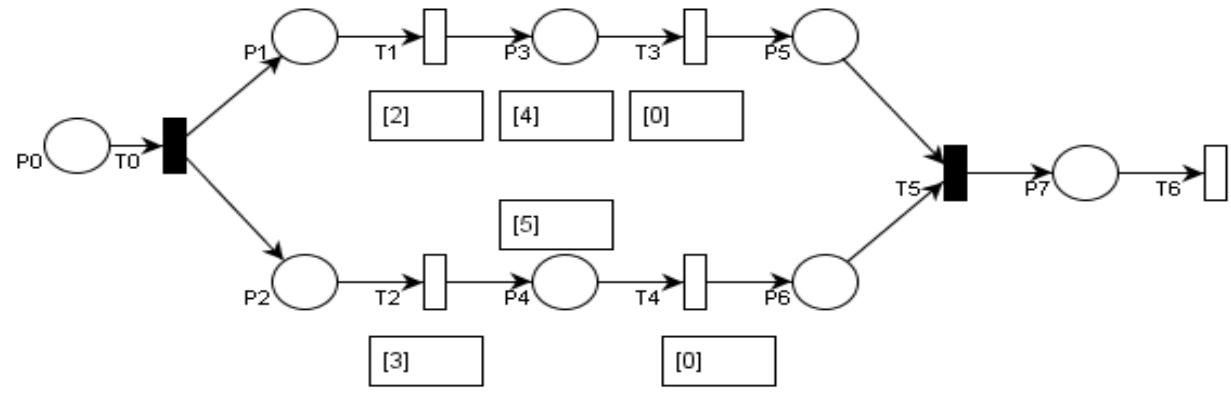


*The activities have relevant durations!
They are modeled by places!!! → P₃, P₄
Delayed transitions model waits!*

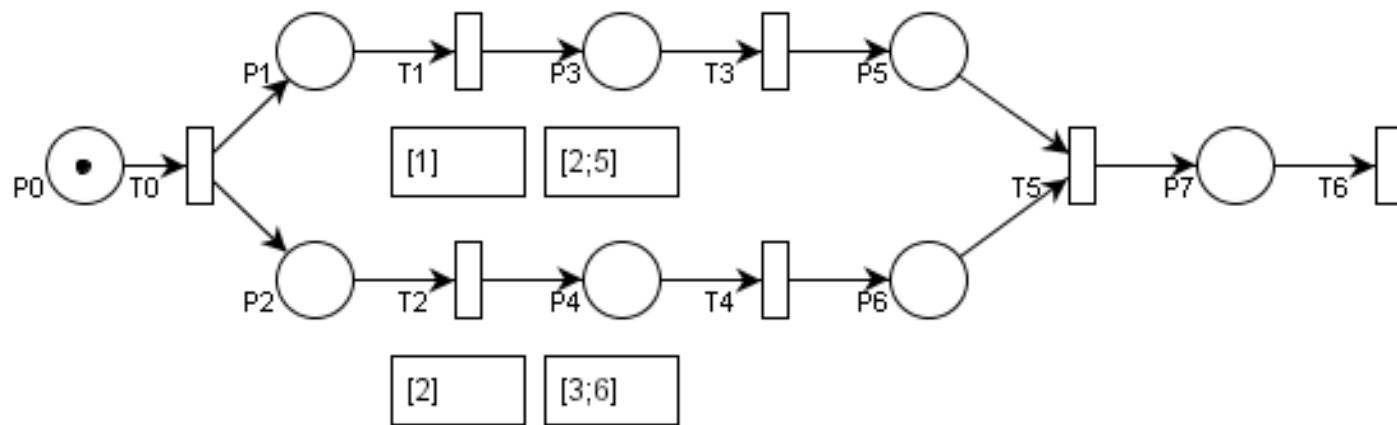
- **Monoprocessor** system
- two threads implementation
- timesharing
- equal priorities

P₃ and P₄ are concurrent!!!
– time-slicing
When is the transition T₆ executed?

$$\sigma = T_0[0] * (T_1[2] * T_3[?] \& T_2[3] * T_4[?]) * T_5[0] * T_6[0] = \\ T_0(0) * (T_1(2) * T_3(9) \& T_2(3) * T_4(11)) * T_5(11_+) * T_6(11_{++}) \\ \rightarrow T_6(11_{++})$$



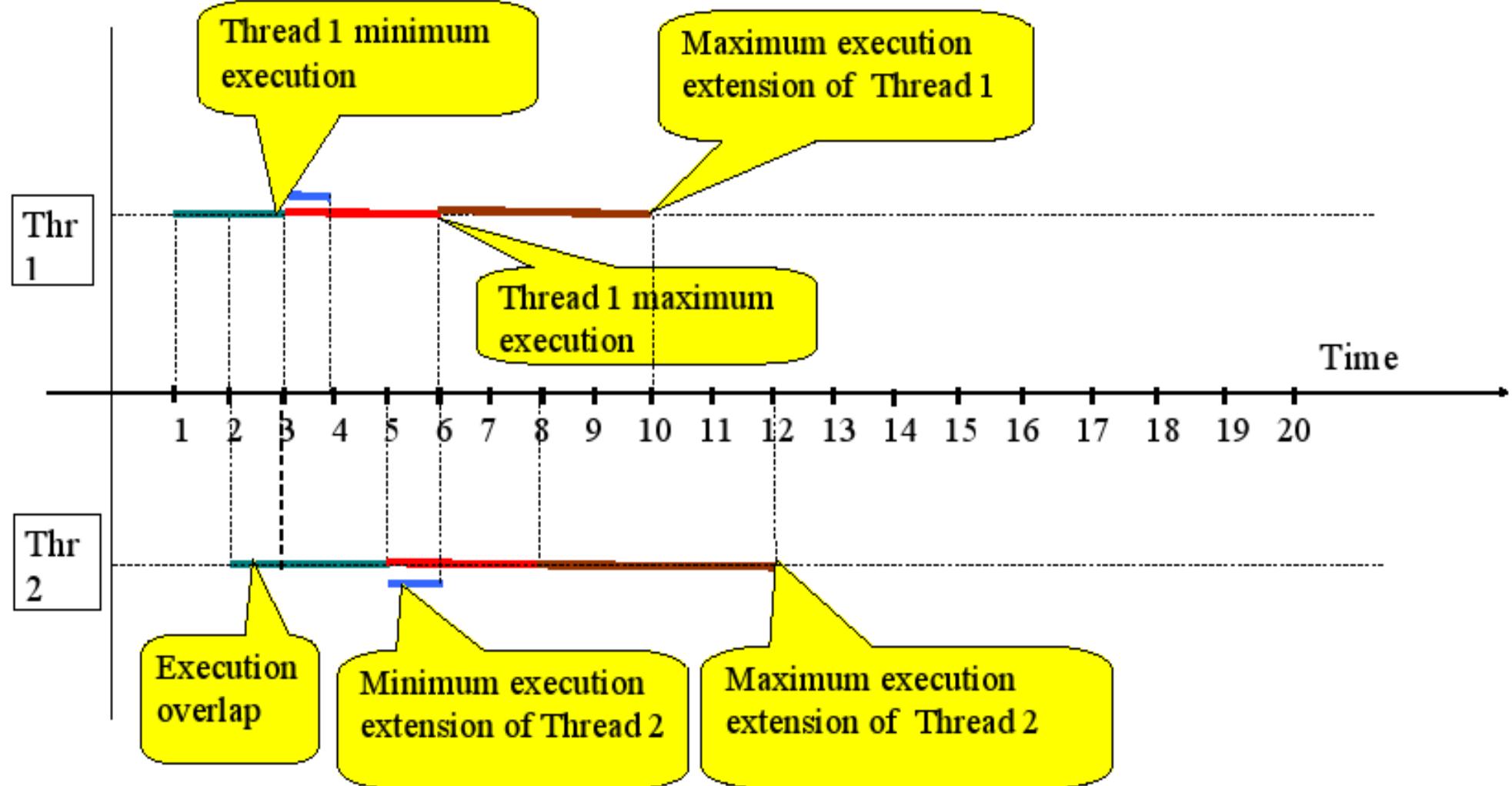
Asynchronous Approach with Variable Durations



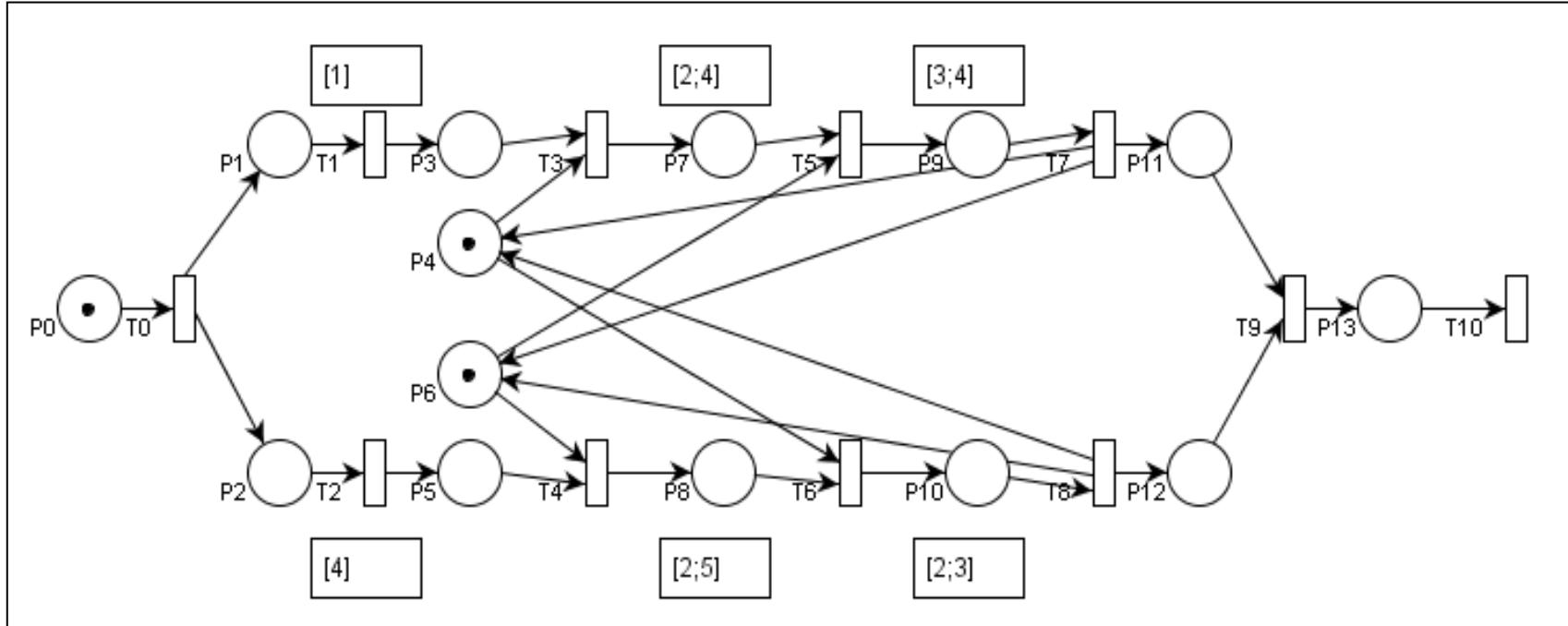
$$\begin{aligned}\sigma &= T_0[0]*((T_1[1]*T_3[?;?])\&(T_2[2]*T_4[?;?]))*T_5[0]*T_6[0]= \\ &\quad T_0(0)*(T_1(2)*T_3(4;10) \& T_2(3)*T_4(6;12))*T_5[0]*T_6[0]= \\ &= T_0(0) *T_5(6_+;12_+)*T_6[0] \\ \rightarrow & T_6(6_{++}; 12_{++})\end{aligned}$$

Conclusion: Variable executions introduce variable delays!
→ Not static interval times, but variable intervals to delay the transitions. ← Difficult for formal demonstration.

The effect of concurrent execution of two threads on the same processor (timeslicing):



Asynchronous approach - Deadlock with variable timing

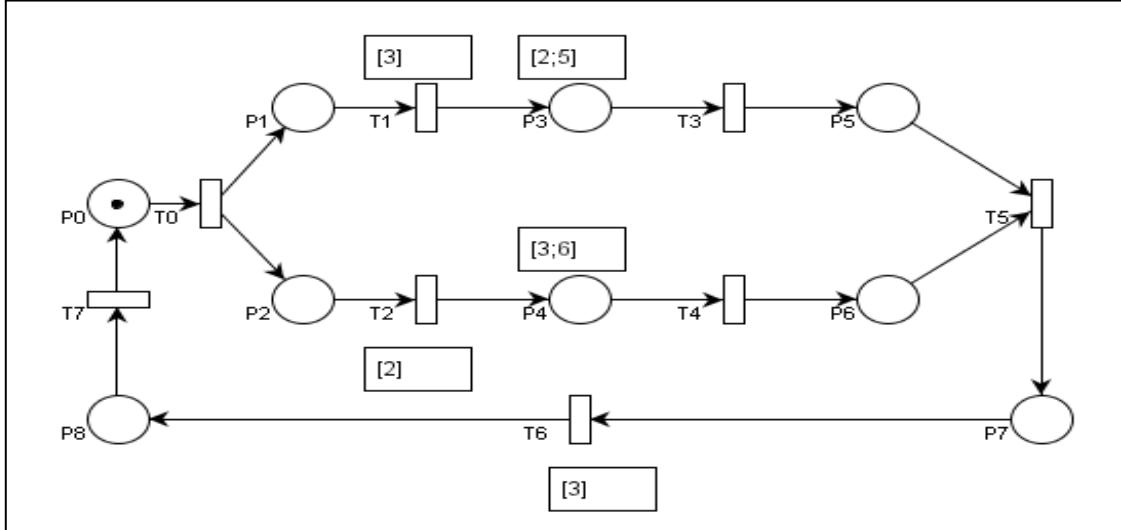


$\sigma_1 = T_0[0] * (T_1(1) * T_3(1+) * T_5(3) * T_7(6+) \& T_2(6+) * T_4(6++) * T_6(8) * T_8(10+)) * T_9[0] * T_{11}[0]$
→ NO Deadlock

$\sigma_2 = T_0[0] * ((T_1(1) * T_3(1+) * T_5(5?) * \delta) \& (T_2(4) * T_4(4+) * \delta)) * T_9[0] * T_{11}[0]$ → **Deadlock**
 Conclusion: Sometimes the program is deadlocked and sometimes is not.

Robustness: the operating systems can add undeclared variable delays.
 Do they have effect on the application behavior?

Constraints specification and verification



- **Monoprocessor** system
- two threads implementations
- timesharing
- equal priorities
- variable durations

Period timing constraint is **12 t.u.**
Can it be met?

$$\sigma_1 = T_0[0_+] * (T_1(3) * T_3(7;13) \& T_2(2) * T_4(7;13)) * T_5(7_+;13_+) * T_6[3] * T_7[0] * \\ * T_0(10_{++};16_{++}) \cdot \dots$$

Conclusion: Sometimes YES and sometimes NO → **Answer: NO for real-time systems.**

IF periodConstraint=17 → YES

Homework: Get an implementation such that the period is always 18 t.u.

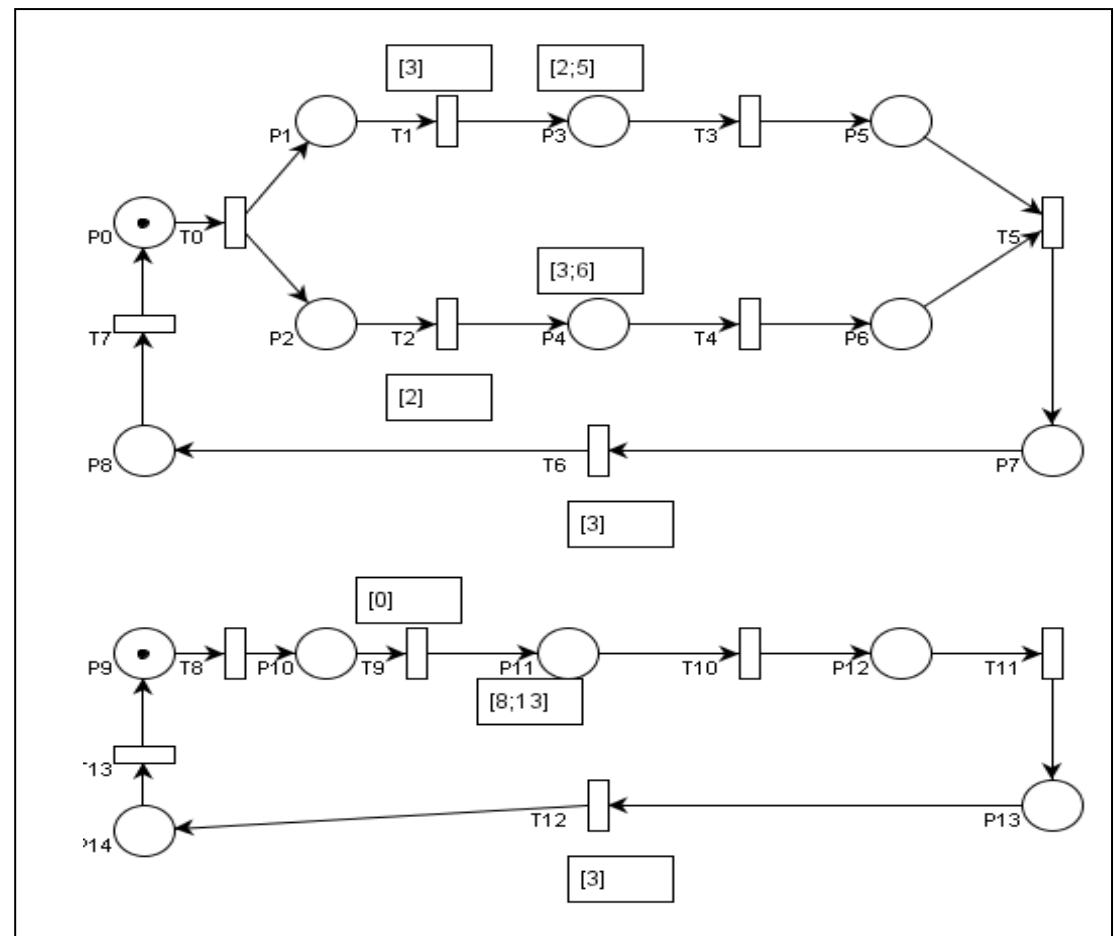
How can this be programmed using Java language?

Method to solve the timing problems:

Petri Net graph reduction

This reduces the graph complexity.

- sequences
- loops
- parallel executions
- splits
- joins



Probleme:

1. Se dă o rețea Petri (modelul unei aplicații).
 - a. Să se analizeze comportamentul temporal al aplicației pe care o modelează
 - b. Să se găsească o implementare a modelului printr-un program:
 - multithreading
 - multiprocessing
2. Se dă o aplicație printr-un program (multithreading sau multiprocessing)
 - a. Să se construiască un model al lui prin rețele Petri (ordinare, stochastice sau de timp)
 - b. Să se analizeze comportamentul temporal al aplicației.
3. Se dă un controller TPN cu specificațiile teemporale.
Să se conceapă un program de test a controllerului pentru îndeplinirea cerințelor temporale?

Specification and Analysis of Real-Time Applications Using (TPNs)

Time Petri Nets with *Inhibitor Arcs* → TPNIA

Petri nets are models.

No model is perfect. Some models are usefully!

TPN (time Petri net) denotes a Petri net with time intervals associated to transitions

TPNIA denotes a time Petri net with inhibitors arcs

$$\text{TPNIA} = (P, T, B, F, I_{nh}, M_0, I)$$

P and T are non empty finite disjoint sets (place set and transition set)

B is the *backward incident function* $B : T \times P \rightarrow \mathbb{Z}$

Z is the set of integers

$B(t,p) = w > 0$ - weight (rom.: ponderea arcului).

It associates weights to *input arcs* of the transitions t.

$B(t,p) = 0 \rightarrow$ There is no arc between p and t.

F is the *forward incidence function*

$$F : T \times P \rightarrow \mathbb{Z}$$

$F(t,p) = w > 0$ - weight (rom.: ponderea arcului)

It associates weights to input arcs of the transitions t .

$F(t,p) = 0 \rightarrow$ There is no arc between t and p .

I_{nh} is the **set of inhibitors arcs**

$$I_{nh} \subseteq P \times T$$

M_0 is the *initial marking*

$$M_0 : P \rightarrow \mathbb{N}$$

N is the set of non negative integers.

The *marking of the net*

$$M : P \rightarrow \mathbb{N}$$

$M(p) = n$ It associates a number of n tokens to the place p .

It describes the current net state.

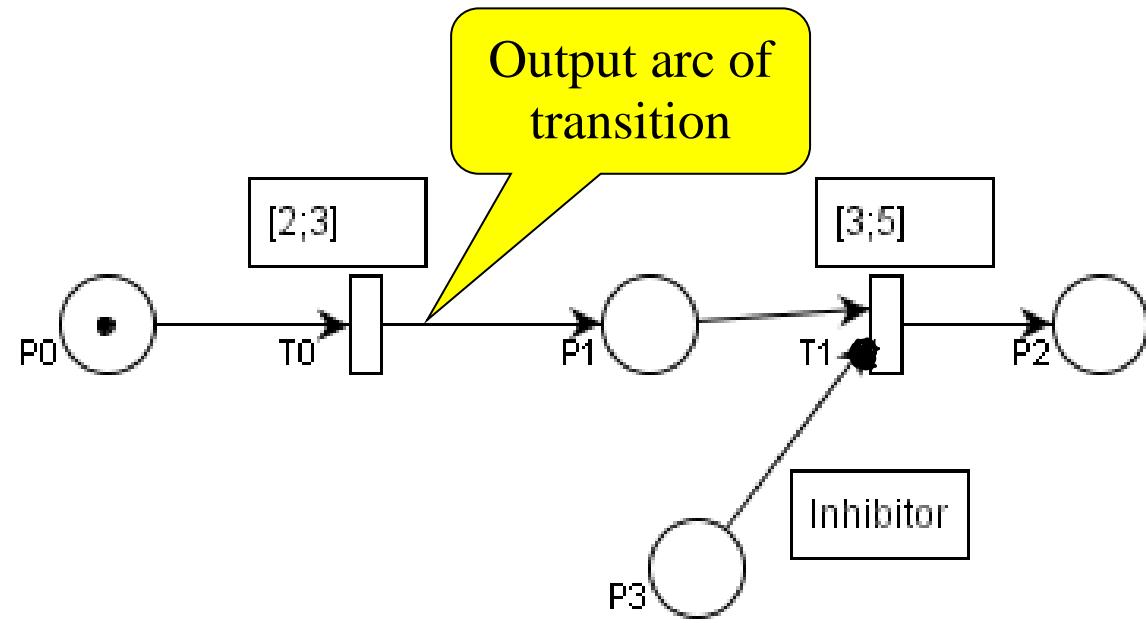
I is the **static firing interval function**

$I(t) = [a;b]$ It associates to the transition t a pair of timing intervals (i.e. a and b).

a is the earliest firing time (EFT)

b is the latest firing time (LFT)

$$I : T \rightarrow \mathbb{N} \times (\mathbb{N} \cup \infty)$$



A ***transition is enabled*** to be executed in a marking M, denoted by $M[t >]$, if and only if:

$$\begin{aligned} \forall p \in P \quad M(p) \geq B(t, p) &\quad \text{if } (p, t) \notin I_{nh} \\ M(p) = 0 &\quad \text{if } (p, t) \in I_{nh} \end{aligned}$$

Let τ be an instant in time when transition t is enabled. It cannot be fired before $(\tau+a)$ but must be fired before $(\tau+b)$.

Let be θ the moment at which the transition t is chosen to be executed.

It must fulfill the constraint $a \leq \theta \leq b$ (in a *strong firing model*). The transition t is executed at $(\tau+\theta)$ and it changes the net state from the marking M to marking M' (denoted by $M[t > M']$) in an instantaneous and atomic process with two phases:

Phase 1: *token withdrawal* $\forall p \in P, M'(p) = M(p) - B(t, p)$

Phase 2: *token deposit* $\forall p \in P, M'(p) = M(p) + F(t, p)$

In a *non time-strict transition* ($b < \infty$) the timing constraint is $a \leq \theta < b$.

Processing models

1. *Time division based execution*

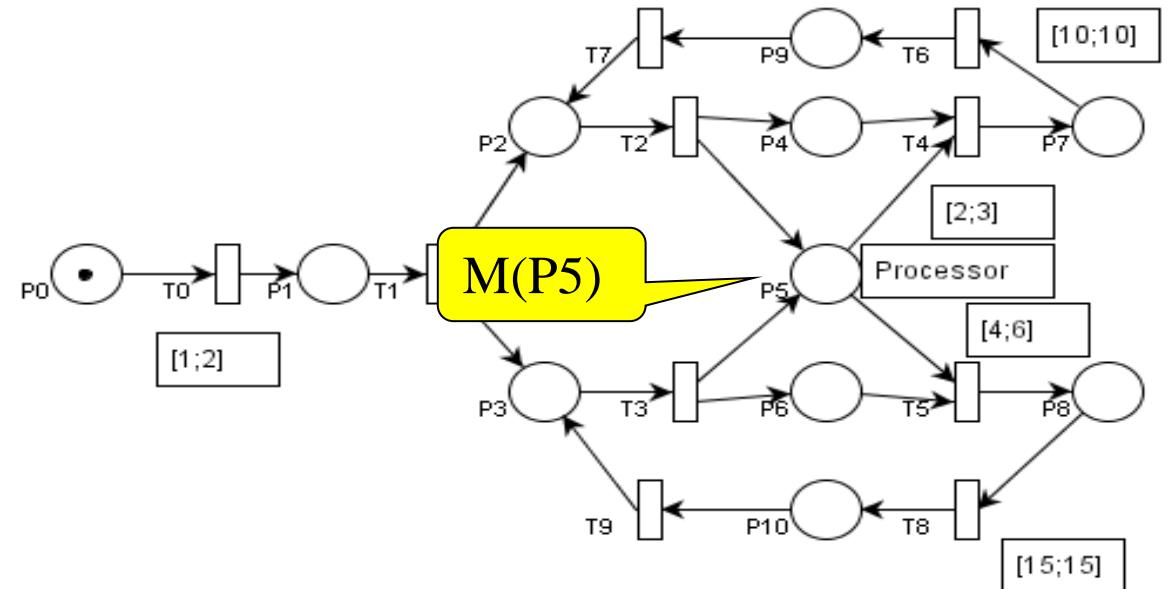
- equal slices
- slices granted proportionally to their priorities

2. *Single thread based execution*

- without preemption – a thread that starts its execution looses the processor only after it finishes the current demand
- with preemption considering their priorities – at each moment of time the thread with the highest priority is in execution state. If a thread becomes executable, it preempts the processor from threads with lower priorities.

1. Time division based execution = Time sharing = time slicing

Let δ be the time slice duration.



P5 represents the processor occupation by the two threads.

P4 and P6 represent the execution of the main (relevant) activities of the two threads. So, the timing T4 and T5 respectively must be extended when the processor is used simultaneously by the two threads because the processor time is divided (considered here equally) between the threads.

What are the periods of thread executions?

What is the processor loading?

A transition t is enabled at τ t. u. Let θ be the relative moment at which the transition t is chosen to be executed. The transition is executed at the absolute moment of time $(\tau+\theta)$.

Assumption a: If more than one thread is executable simultaneously, the processor time is divided equally between the threads.

If two or more transitions are executed simultaneously the place P_5 is loaded with the number of tokens equal with the number of threads. **The transition execution semantics is changed!**

The duration of θ ($a \leq \theta \leq b$) must be extended due to processor time division at θ' :

$$\theta' = \frac{\theta}{\gamma'} / \sum_{i=0}^{\gamma'-1} \frac{\delta}{M(p_x, i)} \quad \text{where } \delta \text{ is a time slice duration and } M(p_x, i) \text{ represent the number of tokens in the place } p_x \text{ at the absolute time } (\tau+i).$$

If $\gamma = \theta \cdot \delta$, then $\gamma' = \theta' \cdot \delta$

If $\delta=1$ then $\gamma = \theta$ and $\gamma' = \theta'$. It is assumed that the processor is allocated to each thread an entire slice.

The implementation is based on a time server with the tick = δ . So, γ' and θ' must be calculated recursively.

The processor loading is given by formula:

$$U = \frac{\theta_1 + \theta_2}{10 + \theta_1 + 15 + \theta_2}$$

Assumption b: If more threads are simultaneously executable they get execution times proportionally with their priority (represented by a natural number).

Example: Thread1 has the priority 3 and Thread2 has the priority 5. The time slice durations fulfills:

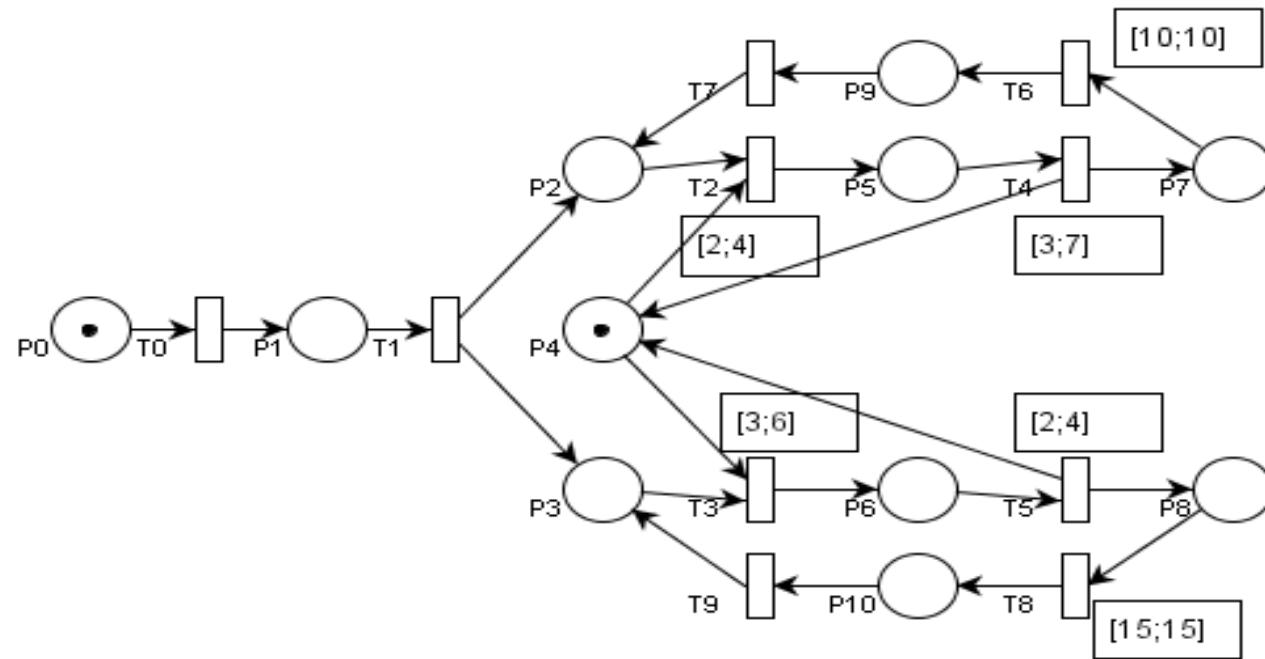
$$\frac{\delta_1}{3} = \frac{\delta_2}{5} = \frac{\delta}{1}$$

How can θ'_1 and θ'_2 be calculated in this case?

How can the processor loading be calculated in this case? Is it different from the previous case?

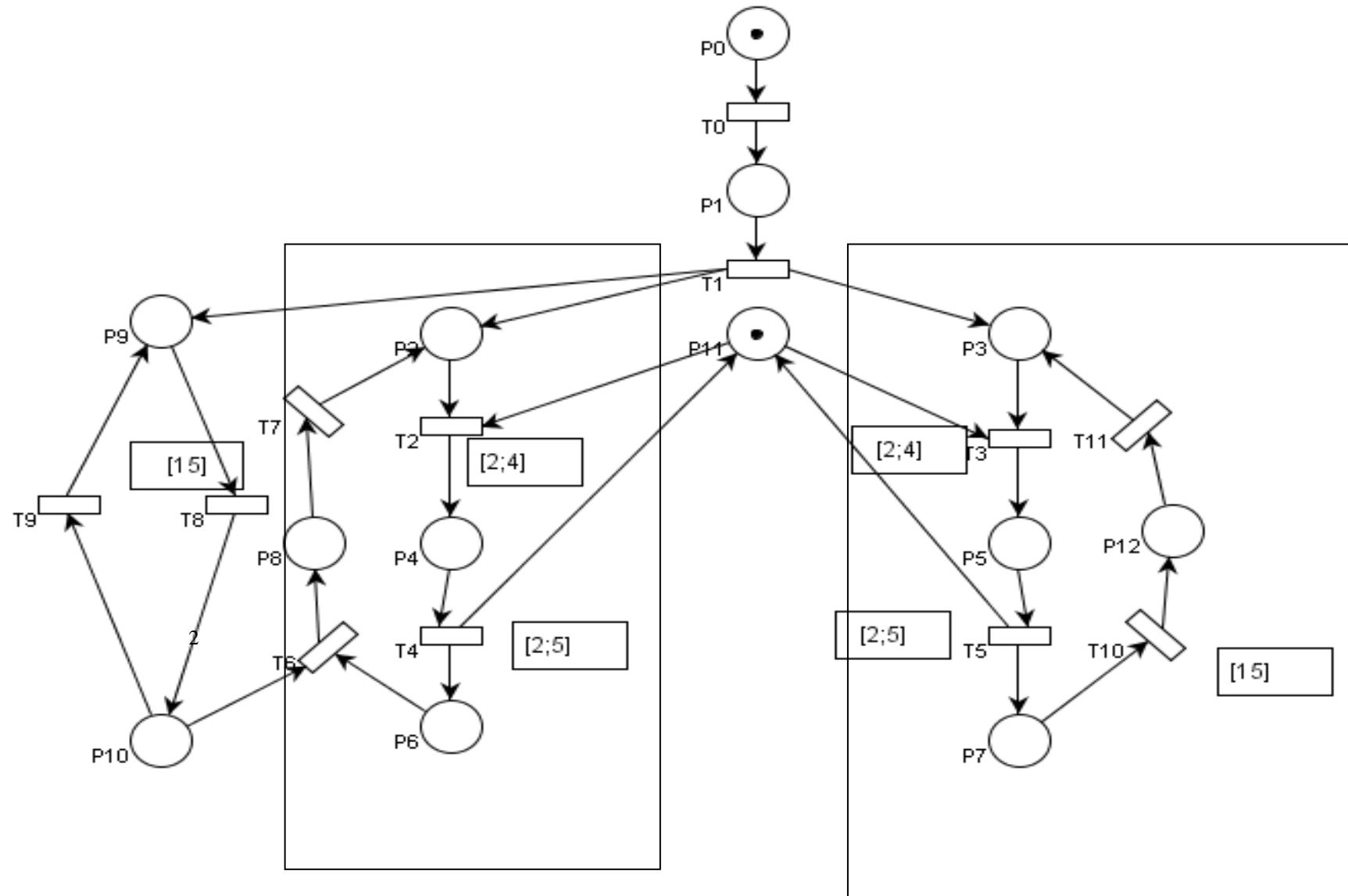
2. Single thread based execution

Assumption a: When a thread started its execution another one cannot start its execution. → **Mutual exclusion! The processor is a resource!**



When a thread starts its execution it blocks the starting of another thread execution.
How can the thread periods be calculated? Are they fixed?

How are the threads' periods from the next PN? Can they be calculated?
 Add a subnet that leads to fixed period.



Assumption b: The operating system works with preemption. When a higher priority thread becomes executable, it preempts the processor (if this is necessary) from a lower priority thread currently in execution.

Let be θ the moment at which the transition t is chosen to be executed.

It must fulfill the constraint $a \leq \theta \leq b$ (in a *strong firing model*). The transition t is executed at $(\tau+\theta)$ and it changes the net state from the marking M to marking M' (denoted by $M [t > M']$) in an instantaneous and atomic process with two phases:

Phase 1: *token withdrawal* $\forall p \in P, M'(p) = M(p) - B(t, p)$

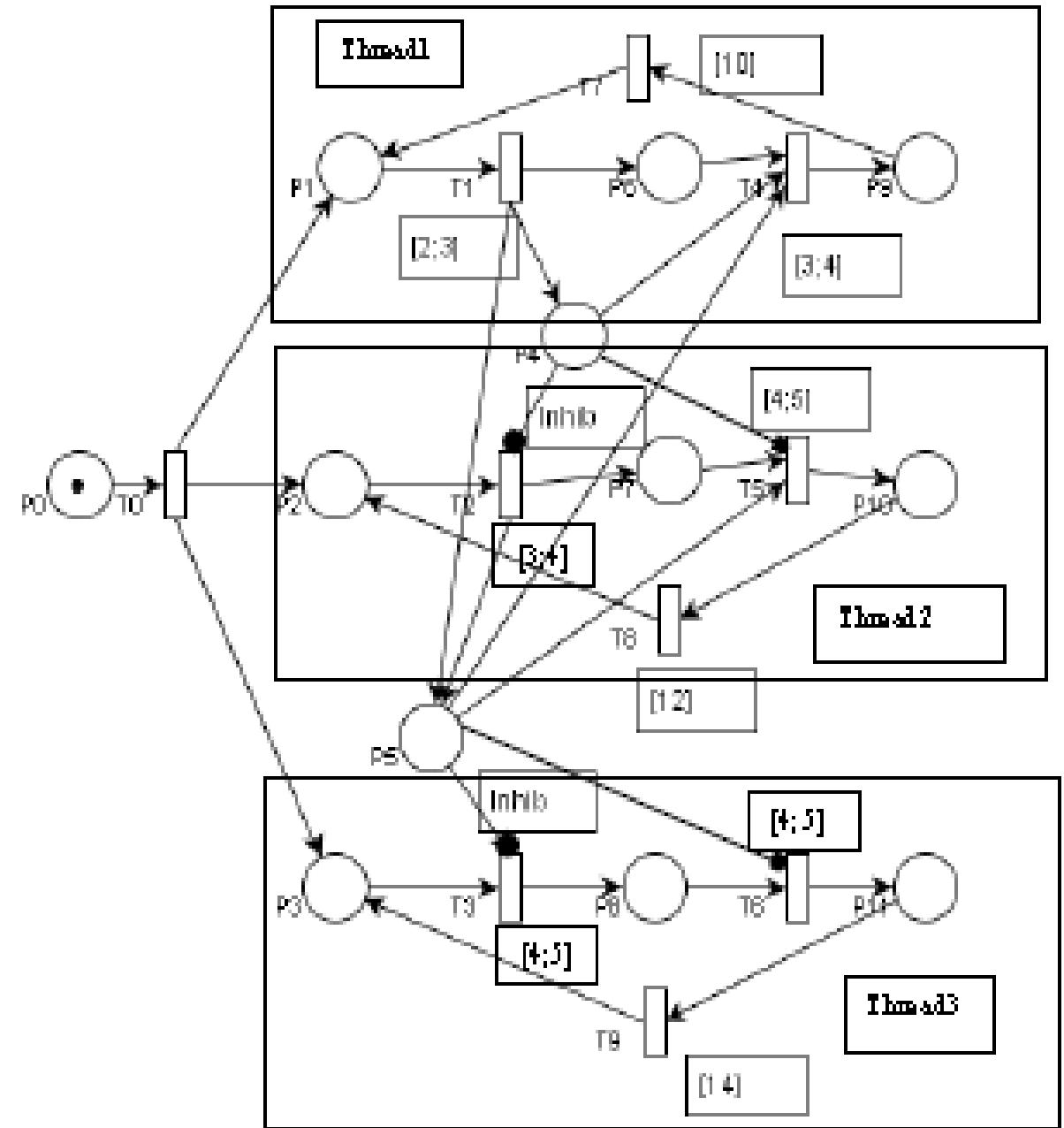
Phase 2: *token deposit* consist of $\forall p \in P, M'(p) = M(p) + F(t, p)$ and takes place at $(\tau+\theta)$

if no other thread was executed during period $(a, \tau+\theta)$

else the deposit takes place at $(\tau+\theta+\Delta)$ where Δ is the duration due to the higher priority thread interruptions.

The transition execution semantics is changed!

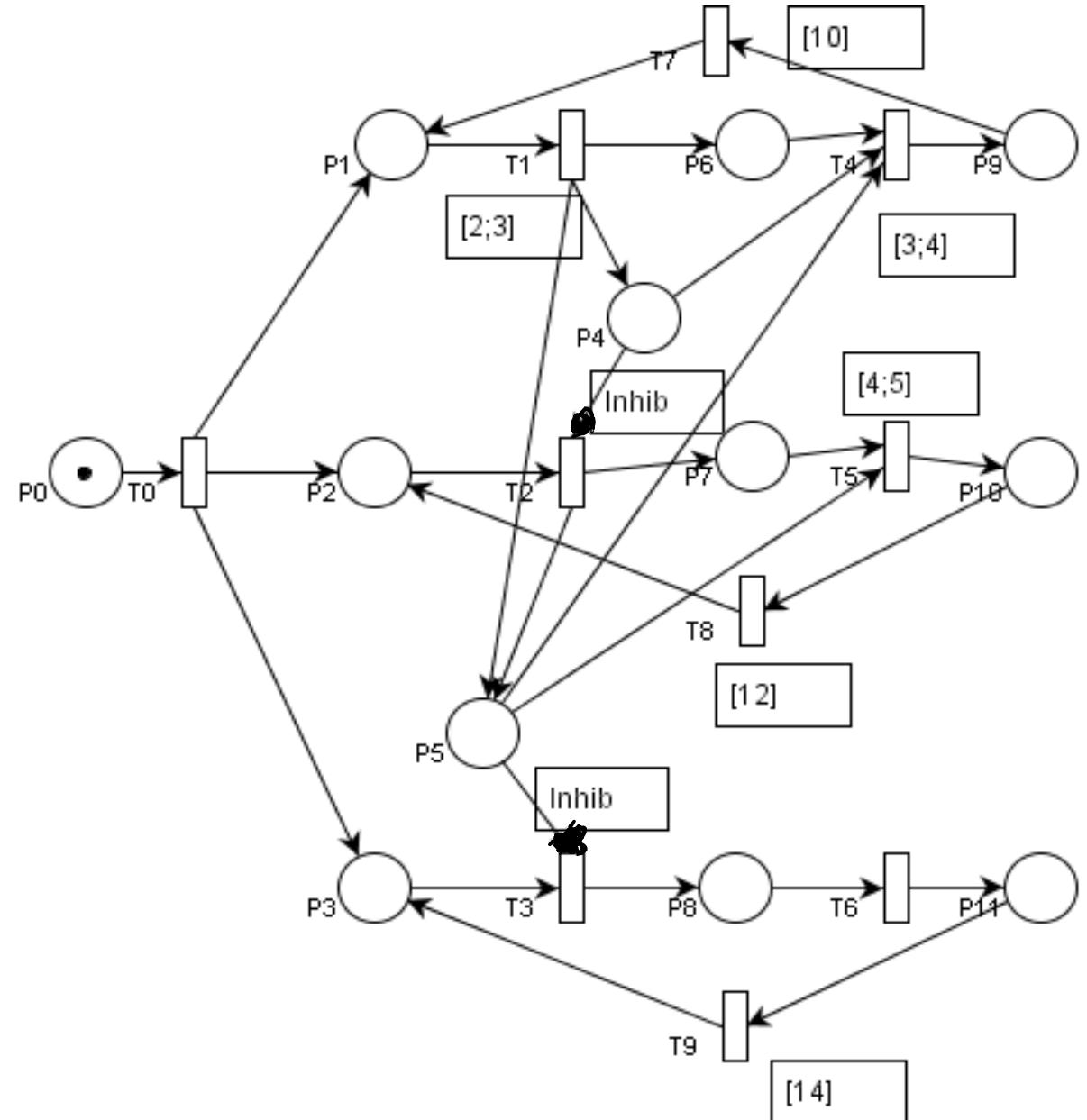
How can this be described?



The inhibitor arcs delay the transition execution!

Which are the thread with the higher priority and the thread with the lower priority?

Does the processor loading differ when the processing works with preemption than it works without preemption?



Conclusions

What was described (modeled)?

- 1) sequential execution
- 2) split of execution (fork and join) and parallel execution
- 3) concurrency (concurrent execution)
- 4) mutual exclusions
- 5) delays (fixed and variables)
- 6) deadlocks
- 7) timing constraints
- 8) periodic execution
- 9) priorities and preemption
- 10) synchronous and asynchronous approaches
- 11) inhibitor arcs
- 12) interaction with the environment
- 13) fixed period executions

Advantages of Petri Nets:

1. Easy to understand
2. Easy to implement (or simulate)

Drawbacks of ordinary Petri Nets:

1. difficult use for large and complex systems (\rightarrow hierarchical PN?)
2. difficult to describe the processor preemption

Solution for large systems: components integrating Petri Nets

*

END

*

2. Specification of RTAs (2.1, 2.2, 2.3, 2.4, 2.5)

2.5 Specification and Verification of RTAs with Object Enhanced Time Petri Nets (OETPN)

(Rețele Petri de timp cu/și obiecte înzestrare)

1. Justification
2. Definitions
3. Construction
4. Properties
5. Examples of OER-TPN models
6. Utility and uses of OER-TPN models. Controller synthesis

1. Justification

OETPN can model:

- Logical operations (AND, OR, NOT, AND-NOT, OR-NOT, OR-exclusive, etc.{},
- Mutual exclusions and synchronizations,
- The use of shared and limited resources,
- Deadlocks,
- Unlimited time (unending) executions,
- Inhibitor arcs,
- Reset arcs,
- Complex data structures,
- Complex conditions for executions,
- Complex operations,
- Distributed structures and executions,
- Concurrency,
- Passive object construction and destruction,
- Thread creations and destructions
- Task migration

Real-time features:

- Execution delays,
- Temporal information,
- Priorities (scheduling information)
- Temporal parameters
 - Deadlines
 - Execution durations

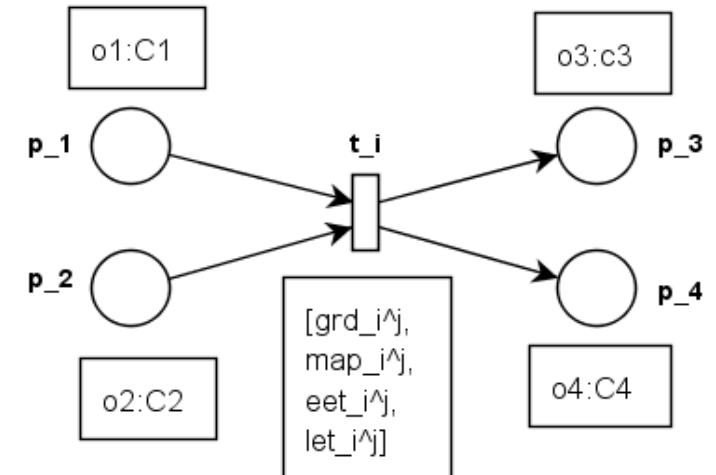
Goals OETPN conception:

- To endow the ETPN (Enhanced Time Petri Net) models with the features of Object-Oriented Programming.
- To sustain the software development in all the phases.
- To help the verification of the phase's outcomes.

2. Definitions

The following notations are used:

- $N = (P, T; F)$ is a net with two kinds of disjoint node sets
- a finite place set $P = \{p_1, p_2, \dots, p_m\}$, ($m \geq 0$),
- a finite transition set $T = \{t_1, t_2, \dots, t_n\}$, ($n \geq 0$),
- $F \subseteq P \times T \cup T \times P$ flow relation,
- ${}^0t = \{p \in P | (p, t) \in F\}$ the transition t input place set
- $t^0 = \{p \in P | (t, p) \in F\}$ the transition t output place set
- $Inp = \{p_{i1}, p_{i2}, \dots\} \subseteq P$ are external inputs that inserts tokens into the net,
- $Out = \{p_{o1}, p_{o2}, \dots\} \subseteq P$ are output channels that extract tokens from the net sending them to channel destination nodes,
- $Types = \{Class1, Class2, \dots\}$ represents the set of software classes of the net,
- $type: P \rightarrow Types$ is a mapping that assigns to each place a class (i.e. type), (the mapping $type(p)$ is a class)
- the marking $M(p)$ of a place p is a reference to an object of the type $type(p)$ and the token is the referred object or *null* (denoted by φ) when there is no token.



- a guard of a transition t_i denoted by grd^j_i is a mapping:

$$grd_i^j : \prod_{p \in {}^0t} \{Domain(M(p)) \cup \Phi\} \rightarrow \{\text{true}, \text{false}\}$$

where *true* and *false* are the values of the Boolean set. Φ denotes the empty set. The set of all guards assigned to a transition t_i is denoted by grd_i , and Grd denotes the set of all the guards of the net.

- a transition t_i mapping denoted map_i is a mapping

$$map_i^j : \prod_{p \in {}^0t} \{Domain(M(p)) \cup \Phi\} \rightarrow \prod_{p \in t^0} \{Domain(M(p)) \cup \Phi\}$$

- $Domain(M(p))$ defines for the token that can be set in the place p the set (possibly very complex and unlimited as dimension) of the values assignable to the object's attributes. All the mappings of a transition t_i are included in the set map_i and Map represents the set of all the mappings of the net.
- a transition t_i *earliest execution time* (delay) and *latest execution time* (delay) are mappings $eet_i^j, let_i^j :$

$$eet_i^j, let_i^j : \prod_{p \in {}^0t} \{Domain(M(p)) \cup \Phi\} \rightarrow \mathbb{R}$$

\mathbb{R} means the set of real numbers considering the time as being dense. Usually the set \mathbb{N} of the natural numbers is used when the time is accepted as being discrete and it is measured with the clock accuracy. Only a transition input place can be used for storing its temporal information. The set of all earliest execution time mappings assigned to a transition t_i is denoted by eet_i , and

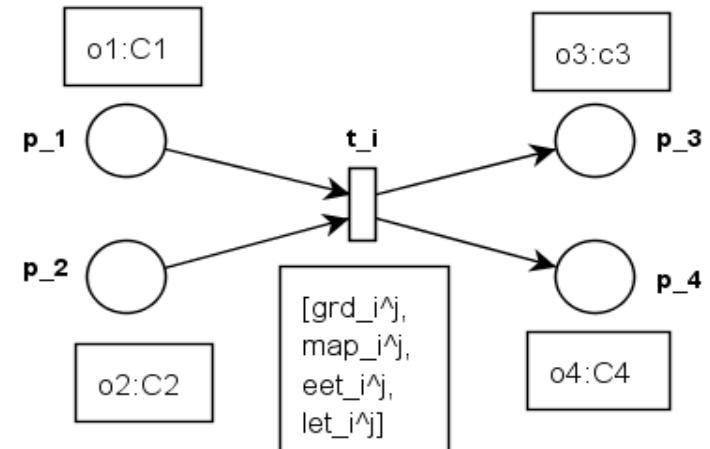
Eet denotes the set of all the delay mapping sets of the net. The set of all latest execution time mappings assigned to a transition t_i is denoted by let_i , and Let denotes the set of all the delay mapping sets of the net.

The ***definition of an OETPN*** is:

$$OETPN = (P, T, F, Inp, Out, \text{Types}, \text{type}, \text{Grd}, \text{Map}, \text{Eet}, \text{Let}, \Lambda, M, \text{init}, \text{end})$$

where:

- $P, T, F, Inp, Out, \text{Types}, \text{type}, \text{Grd}, \text{Map}, \text{Eet}, \text{Let}$ have the previous meanings.
- $\Lambda = \{\lambda_t | t \in T, \lambda_t \subseteq \text{grd}_t \times \text{map}_t \times \text{eet}_t \times \text{let}_t\}$ is the set of relations that assigns to each transition t a guard condition, a mapping, an earliest execution time and a latest execution time. The sets $\lambda_t = (\text{grd}_t^j, \text{map}_t^j, \text{eet}_t^j, \text{let}_t^j) (j=1,2,\dots)$ represent the model *evolution rule set of the transition t*.
- M is the marking vector with $M(p)$ the marking of the place p that identifies the object token of the type $\text{type}(p)$. $M(p) = \varphi$ means there is no pointer to an existing object, and it is equivalent to $M(p) = 0$ of the classical Petri nets.
- init is a function that initializes the net state (i. e. the markings) with values from the creator's domain $\text{init}: \text{Domain}(\text{creator}) \rightarrow \text{Domain}(M)$
- end is a function that terminates the OETPN model execution and provides to the net creator the processed information $\text{end}: \text{Domain}(M) \rightarrow \text{Domain}(\text{creator})$



Transition execution

The OETPN *marking* is a vector \mathbf{M} with the elements $M(p)$ as references to objects if the corresponding places have tokens or φ (i.e. null) if they do not.

A transition t is *enabled* or *admissible* at the current moment τ_c from the current marking \mathbf{M} if there is a fulfilled guard condition in λ_t .

It is supposed that only one guard condition of a transition t can be true from any given marking. If more than one guard condition can be fulfilled simultaneously for a transition, the first found is taken into account.

Similarly, it is supposed that there are no conflict transitions for any marking. If there are conflict transitions, the transition with the lowest index wins the election.

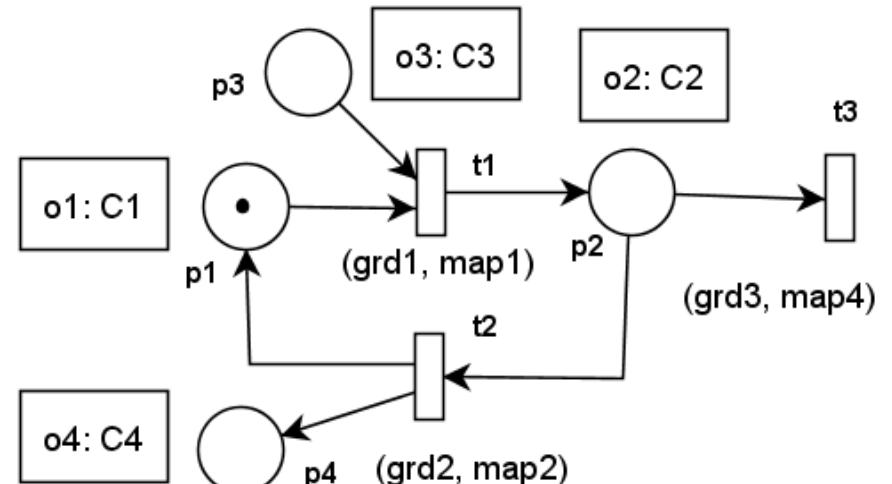
An enabled transition at the τ_c is *fireable* or *executable at the time moment* $\tau = \tau_c + \theta$, with θ a value fulfilling the constraint $eet_t \leq \theta \leq let_t$. The transition is executed when the current time reaches τ .

Description of an OETPN model

OETPN with representation of the assigned tokens and the guards and mappings assigned to transitions. The temporal relations are (temporarily) ignored.

Notations:

- o_1, o_2, o_3, o_4 the instances assigned to the places p_1, \dots, p_4 . They represent the tokens and referred by the marking $M(p)$.
- C_1, C_2, C_3, C_4 the classes (types) assigned to the corresponding places.
- $(\text{grd}1, \text{map}1), (\text{grd}2, \text{map}2), (\text{grd}3, \text{map}3)$ are flow (i. e. evolution) relations assigned to transitions t_1, t_2 and t_3 respectively.
- $\text{Inp}=\{p_3\}; \text{Out}=\{p_4\}$



The evolution relations can contain methods that call the *getter* and *setter* methods defined in the instances and the classes (i.e. static methods) assigned to places.

Logic: a token is an object that store information

Implementation: the token is a pointer to an object instance that store information

The transitions represent the transformation of information.

OETPN model temporal descriptions

(see the attached figure)

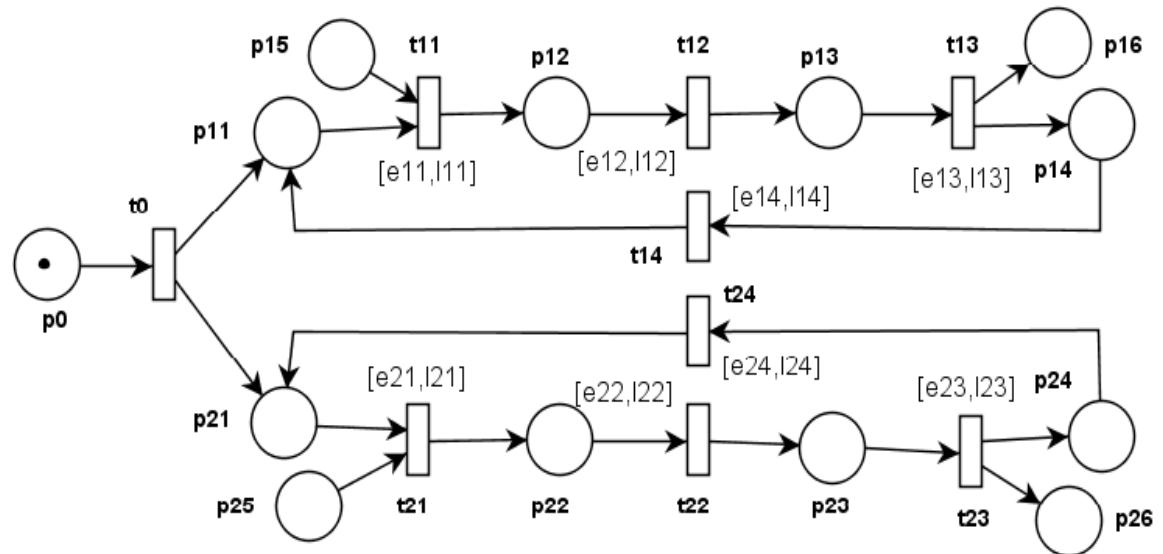
The execution of the transition t when it is executable (after being enabled) transforms the anterior state S_a (anterior, or transition input state) in S_p (posterior or transition output state) according to the evolution relation (grd, map). The executions of $grds$ and $maps$ could need relevant durations.

These are included in $[eet, let]$.

The states are given by the contents of objects referred by the place sets 0t and t^0 .

The OETPN model temporal descriptions differ according to the phase where they are used:

- *Specification* – they describe that the state *is changed* after the transition is enabled between *eet* and *let*.
- *Program or algorithm synthesis* – they describe that the state *has to be changed* between *eet* and *let*. In implementation this correspond to: $x=random(eet,let); wait(x);$
- *Program implementation verification* – they describe when ends the execution of the evolution relations (grd, map) relative to the moment when the input state is reached. The notation $[eet, let]$ can be changed to signify *eet* for the worst case execution time and *let* for the deadline of the state transformation.

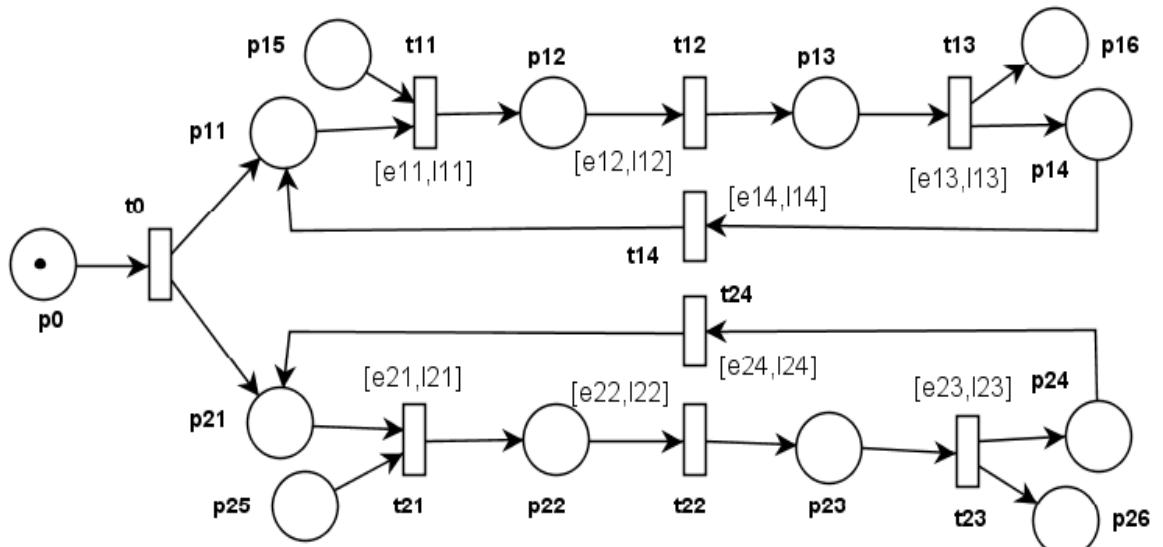


Observations for implementation:

- dynamically creation and removal of objects require asynchronous object relocation;
- object relocation requires relevant amounts of time.

R-T program description

An OETPN model is built for the purpose to meet the R-T constraints.



R-T approaches:

- *Synchronous approach* – the executions of the evolution relations have no relevant durations related to the shortest deadline of any evolution relation;
- *Asynchronous approach* – the executions of the evolution relations have (need) relevant durations related to the shortest deadline of any evolution relation;
- *Mixed approach* – some of the executions of the evolution relations have relevant durations related to the shortest deadline of any evolution relation;

Proposed solutions (implementation specification):

- the tasks composing the OETPN models are partitioned in two sets:
 - *Real-time (R-T) task set* – tasks that do not create dynamically objects – RT tasks use for instantiation only classes that do not lead to dynamical object creation and removal. A R-T task creates its structure at the beginning (before the starting) and it does not change further its structure during the execution.
 - *Non-R-T (NR-T) task set* - tasks that need dynamical object creation and removal – NR-T tasks can use any class; it can change its structure during execution.
- If a R-T task executes an activity with the *Worst Case Execution Time* (WCET) non-ignorable relative to the smallest deadline of any activity of any task, it requests the permission from the scheduler.
- If a R-T task has to execute an activity with ignorable WCET, it does not need the permission of the scheduler.
- The R-T tasks do not interfere with the NR-T tasks.

Inheritance or extension

E. g. class X extends Y

Object Oriented Programming inheritance → it *extends*

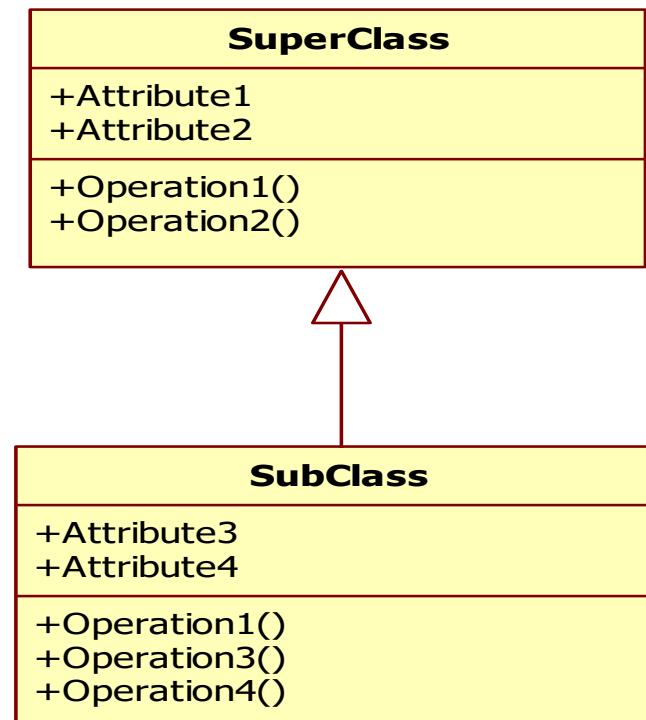
Superclass → *subclass*

- *method*

- tokens = (pointer to) instances

- guard & mapping → setter & getter + methods defined for guards, mappings and temporal relations.

Class methods could be vague defined (or not explicit defined) in an earlier phase of the development.



Extending and reduction of OETPN models

These are not OOP concepts. They belong to software engineering.

An OETPN model can be extended by adding new nodes, branches or subnets to the previous structure.

An OETPN model can be reduced by removing branches, nodes or subnets.

Utility:

- development
- maintenance

Evolution rule syntax

The rules of evolution assigned to a transition t_i with the input place set ${}^o t_i = \{p_1, p_2\}$ and the output place set $t_i {}^o = \{p_3, p_4\}$ can be described according to the following syntax:

$type(m(p_1)) = type(p_1) = \{x: float; y: int\}$ or $type(m(p_1)): \{x: float; y: int\}$

$type(m(p_2)) = \{x: float; y: float; z: boolean; method(x,y):= (x + 2 \cdot z)\}$

$type(m(p_3)) = \{x: float;\}$

$type(m(p_4)) = \{x: float; y: boolean; z: float;\}$

$grd_i^1 := (m(p_1) \neq \varphi \text{ and } m(p_2) \neq \varphi) \rightarrow map_i^1 := (m(p_3).x = (m(p_1).y + 1.2); m(p_4).y = m(p_2).z)$

$grd_i^2 := ((m(p_1).x \geq 0) \text{ and } m(p_2) = \varphi) \rightarrow map_i^2 := (m(p_3).x = (m(p_1).x - 2.0); m(p_4).x = m(p_2).x)$

When no confusion is possible, some shorter forms can be used:

$x_i = m(p_i).x = p_i.x$; where x_i is a variable x of the token set in $m(p_i)$.

The previous relation $m(p_3).x = (m(p_1).y + 1.2)$ can be written as $x_3 = y_1 + 1.2$.

Guard conditions can describe logic relations including AND, OR, NAND etc.

OR relations can be described by using more guard expression too.

All the expressions of the guard conditions and the mappings have to be compatible with the involved token types.

The place types can be of primitive data or class types. The class types can be defined in the environment package or by the developer.

The known class methods can be used in guard conditions or mappings as in the following example:

$$map^3_3 = (m(p_3).x = m(p_1).x + 10 \cdot Math.random());$$

Object instance methods included in tokens can be used such as in the following example:

$$m(p_4).x = m(p_1).x + m(p_2).method(x,y);$$

3. OETPN models construction

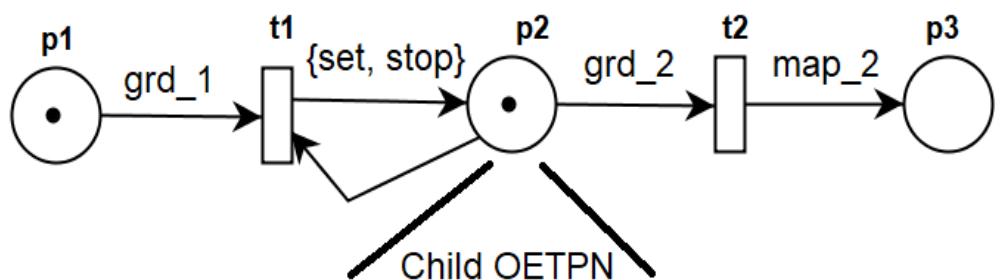
Hierarchical OETPN models

Parent OETPN model creates a subnet named Child OETPN in the place p_2 .

After creation it is initialized by *init()* method and started. Until is running, it is an active object and cannot be used for guard evaluation and mappings. It can be stopped by its parent if the transition used for instantiation is enabled again.

An OETPN child can end the execution terminating its sequence of transitions or calling the *end()* method. Once an active object ends its execution, it becomes a passive object and can be used for guard evaluations or mappings.

Subnet creation - Parent-child example



Conclusion: A concurrent application is composed of:

- *active objects* and
- *passive objects*
- the information stored in an active object are not allowed to be accessed from outside before it is stopped. ← **safety rule**

E. g. Child OETPN creation

OETPN parent model PN_1 creates the OETPN child model PN_2.

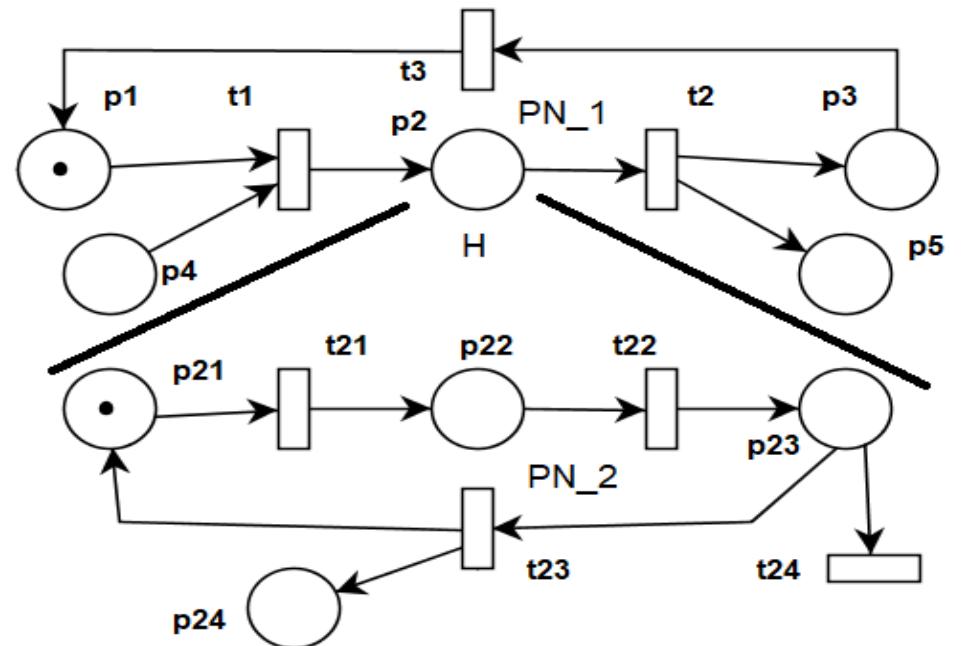
PN_1 was executed/created by the program *main()* method. PN_1 and PN_2 are concurrently executed. When PN_2 ends, it can be used for assessing the guard of transition t2.

Specifications:

- $\text{type}(p4) = \{\text{parameters}\}$
- $\text{type}(p2) = \text{PN_2}$

The transition t1 waits input channel p4 (when $M(p1) \neq \emptyset$; sometimes we use $p1 \neq \emptyset$). Its map1 creates the task OETPN named PN_2 in p2. After initialization (*init()*) PN_2 is started.

The transition t2 cannot be executed (due to the lack of the token in p2) until PN_2 ends its execution.



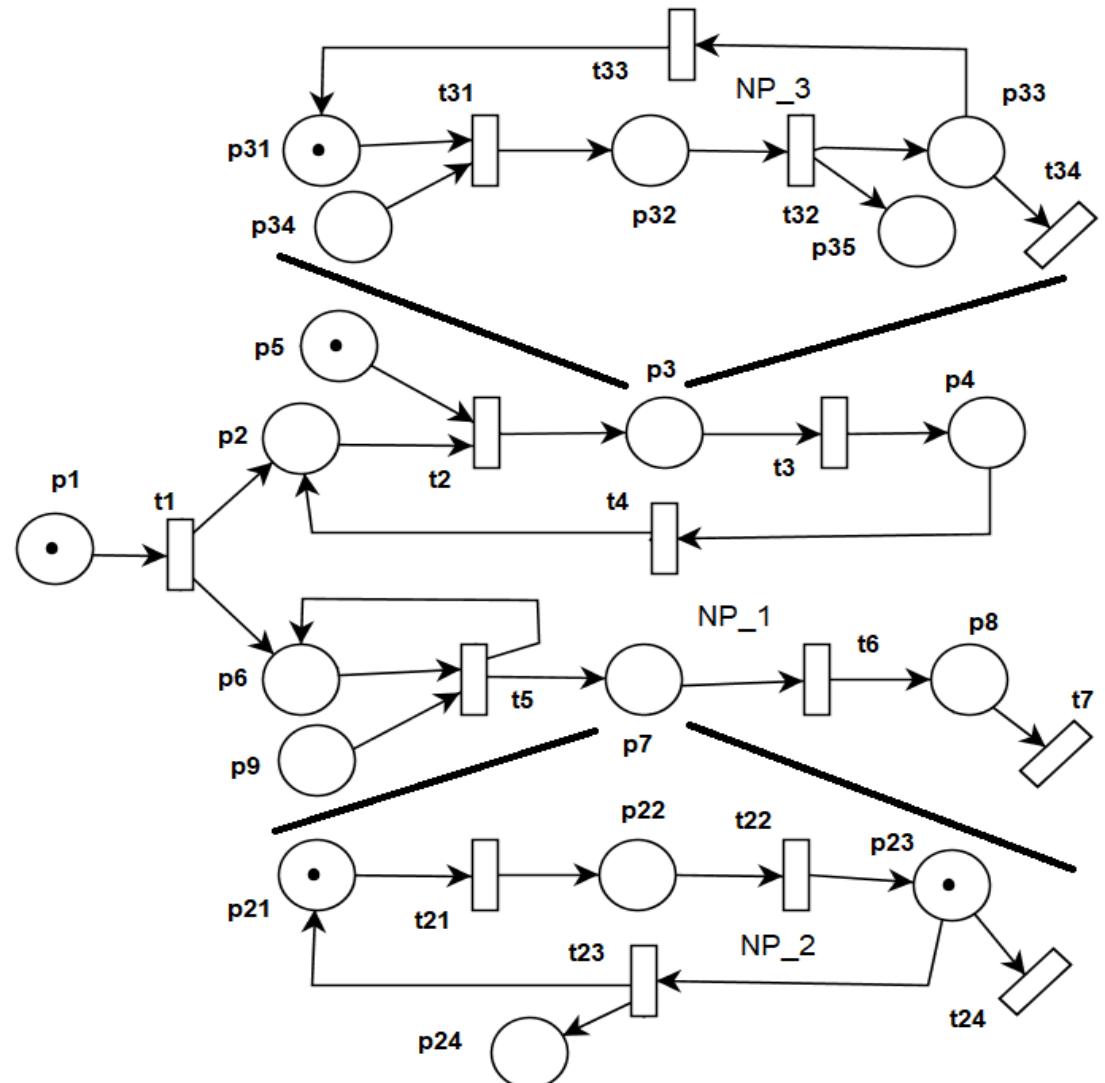
Concurrent execution

NP_1 is parent.

NP_2 and NP_3 are children.

The children can communicate using *Inp/Out* channels.

Recommendation: Do not read and write directly from/to different tasks (threads).



Factorial example

Specification:

The user provides by an input channel an integer number k to calculate its factorial number $f = k! = 1 \cdot 2 \cdot \dots \cdot k$ and print the value on the screen.

If the user demands, during the execution of $k!$, another value k' , stop the previous calculus and start the calculus of the new $k'!$.

The

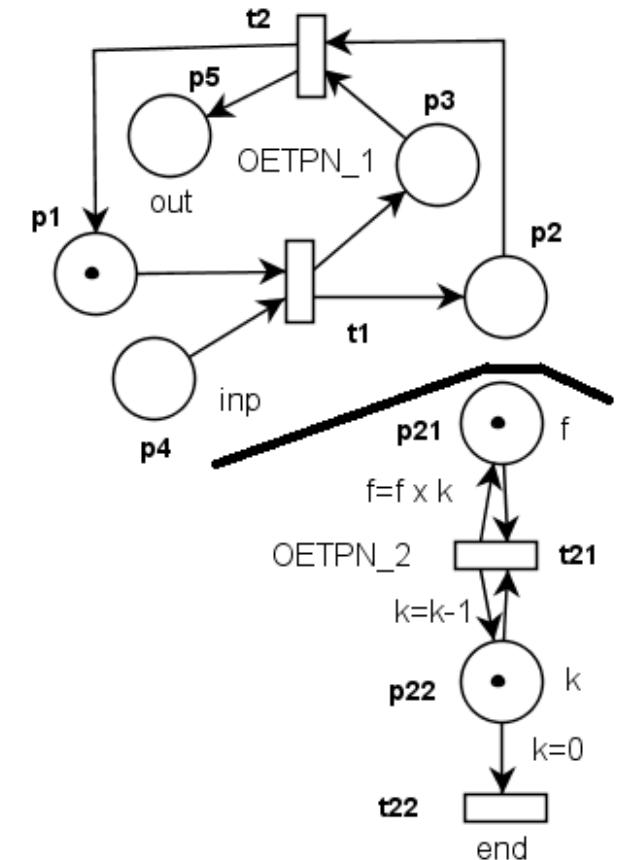
OETPN_1 has an input channel $p4$ and an output channel $p5$ for printing the result.

$$\begin{aligned} type(p1) &= type(p4) = type(p3) = type(p5) = \{x: int\} \text{ (i.e. integer)} \\ type(p2) &= OETPN_2 \end{aligned}$$

The marking of OETPN_1 is $[M(p1), M(p2), M(p3), M(p4), M(p5)]$

$$Domain(p1) = Domain(p3) = Domain(p4) = Domain(p5) = \mathbb{N} \cup \varphi$$

where \mathbb{N} represents the natural number set and φ denotes the empty set representing here the lack of a token.



The OETPN_1 transitions have the evolution relations:

- $t1$:
 - $\{(grd^1_1:=(M(p2)=\varphi \text{ and } M(p3)\neq\varphi) ; map^1_1:=(p3.x=p4.x; p2.OETPN_2; init():=p21.x=1; p22.x=p4.x; p2.OETPN_2=active)\}$
 - $\{(grd^2_1:=(M(p2)\neq\varphi \text{ and } p2.state=active); map^2_1:=(p3.x=p4.x; p2.end(); p2.OETPN_2; init():=p21.x=1; p22.x=p4.x; p2.OETPN=active)\}$
- $t2$:
 - $\{(grd^1_2:=(M(p3)\neq\varphi \text{ and } M(p4)=\varphi); map^1_2:=(p1.x=p3.x; p5.x=\varphi)\}$
 - $\{(grd^2_2:=(M(p4)\neq\varphi); map^2_2:=(p1.x=1; p5.x=p21.x); p2.OETPN=passive\}$

Transition $t1$ creates (if the condition $grd1 \wedge 1$ is fulfilled) a new thread referenced by the place $p2$. $OETPN_2$ markings are initialized by $init()$ method with information taken from the creator domain (i.e. the value set in $p2$).

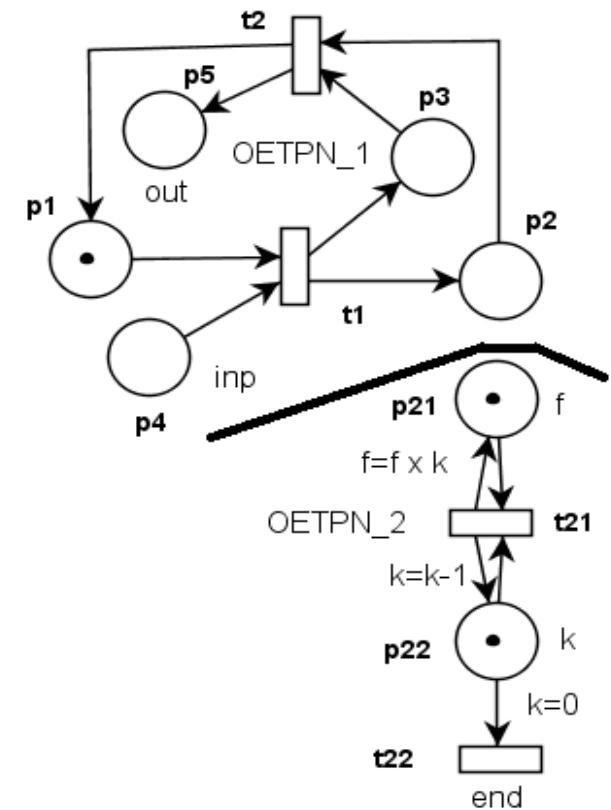
$OETPN_2$ has the marking $[M(p21), M(p22)]$. The places $p21$ and $p22$ have the same type and domain as $p1, p3, p4$ and $p5$. Its transitions have the evolution relations:

- $t21: \{(grd_{21}:=(M(p21)\neq\varphi \text{ and } M(p22)\neq\varphi \text{ and } p22.x>0); map_{21}:=(p21.x=p21.x \cdot p22.x; p22.x=p22.x-1))\}$
- $t22: \{grd_{22}:=(M(p22)\neq\varphi \text{ and } p22.x=0); map_{22}:=(end(); p2.OETPN=passive)\}$

The parent ($OETPN_1$) and the child ($OETPN_2$) can be executed concurrently.

The call of *end()* method stops the execution of OETPN_2 and the marking M(p21) stores the last value of f. When the execution ends normally, the last value is the factorial number. When the execution is interrupted by the parent, the value stored in M(p21) is reloaded with ‘1’.

When the execution of OETPN_2 ends, the marking in the place p2 is a reference toward an object that contains the value of $f=k!$. The current example has no delays (eet, let) specified. The implicit timings are given by the user timed demands and the execution duration of $k!$ calculus.



Formal description:

$$PNL \text{ OETPN_2} \rightarrow (t21)^k * t22; TPNL \text{ OETPN_2} \rightarrow t21[k \cdot \delta]^* t22[0_+]$$

$$\text{OETPN_1} \rightarrow t1^* ((t21)^k * t22) \# t2$$

$$ETPNL \rightarrow t1[p4;p2]\#t2[p2;p5]; t21[\varphi;\varphi] * (t21[\varphi;\varphi] + t22[\varphi;\varphi]) = t21[\varphi;\varphi]^* t22[\varphi;\varphi]$$

TPNL \rightarrow $t1[e1,11]\#t2[0,k \cdot \delta]$; where δ is the duration of one multiplication.

Is here a benefit obtained by multitasking implementation?

Can it be conceived by a single task?

Combinations

The number of distinct subsets with k elements that can be chosen from a set with n elements is

$$C_n^k = \frac{n!}{k! \cdot (n - k)!}$$

and is “ n choose k ” when ($k \leq n$).

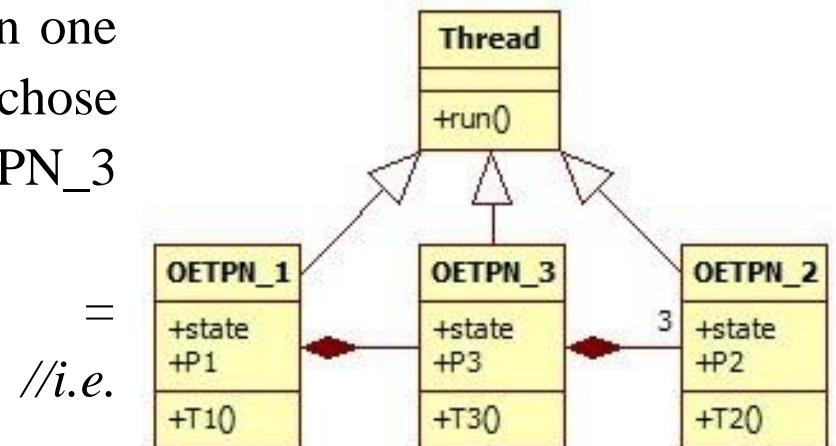
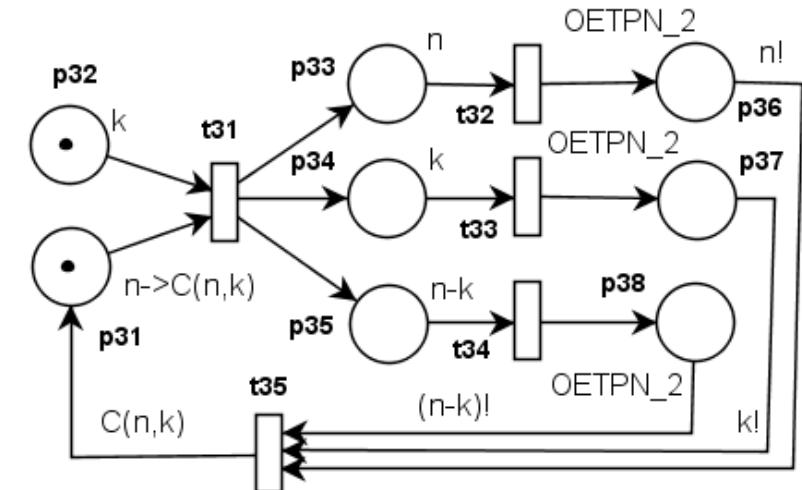
The transformation of the previous application in one that calculates the combinations ($C(n,k)$) of n elements chose k involves the construction in the place p_2 of the OETPN_3 model shown in the attached figure. It has the types:

$\text{type}(p_{31})$

$\text{type}(p_{32})=\text{type}(p_{33})=\text{type}(p_{34})=\text{type}(p_{35})=\text{int};$

//integer

$\text{type}(p_{36})=\text{type}(p_{37})=\text{type}(p_{38})= \text{OETPN_2}$



The places p_{31} and p_{32} of OETPN_3 are uploaded with the value of n and k respectively. The transition t_{31} calculates the value $(n-k)$ and set it in the place p_{35} . The transitions t_{32} , t_{33} and t_{34} create the OETPN_2 children (i.e. threads) that calculates the values $n!$, $k!$ and $(n-k)!$ respectively. At the end, the transition t_{35} calculates and sets in p_{31} the value $C(n,k)$.

The current example represents a three levels hierarchical model that concurrently executes until 5 threads as can be seen from the UML (Unified Modeling Language) class diagram in Fig.

An OETPN represents a task that is implemented by a thread of execution. An OETPN token has the attribute *state* with the value ‘0’ (meaning it does not exist, i.e. is φ), ‘1’ when it can be used for guard evaluation and mapping calculus, or ‘2’ when it is running (i.e. is active and cannot be used as token until it ends its execution or is interrupted). The attribute $P1$, $P2$ or $P3$ represent the set of OETPN places that refer the passive tokens (i.e. passive objects). The operations $T1()$, $T2()$ and $T3()$ represent the transition evolution rules (guards, mappings and temporal functions). The object-oriented programming getter and setter methods can be included in passive and active tokens.

Formal description:

$PNL \rightarrow t31 * (t32 \& t33 \& t34) \# t35;$

$OETPN_2 \rightarrow (t21)^k * t22; ;$

$TPNL OETPN_2 \rightarrow t21[k \cdot \delta] * t22[0_+]$

Is here a benefit obtained by multitasking implementation?

Can it be conceived by a single task?

Need it the same duration on a quad-core system? // single processor!

$TPNL \rightarrow t31[0_+] * ((t32[v] * t21[n \cdot \delta]) * t22[0_+]) \& (t33[v] * t21[k \cdot \delta]) * t22[0_+] \& (t34[v] * t21[(n-k) \cdot \delta]) * t22[0_+])$

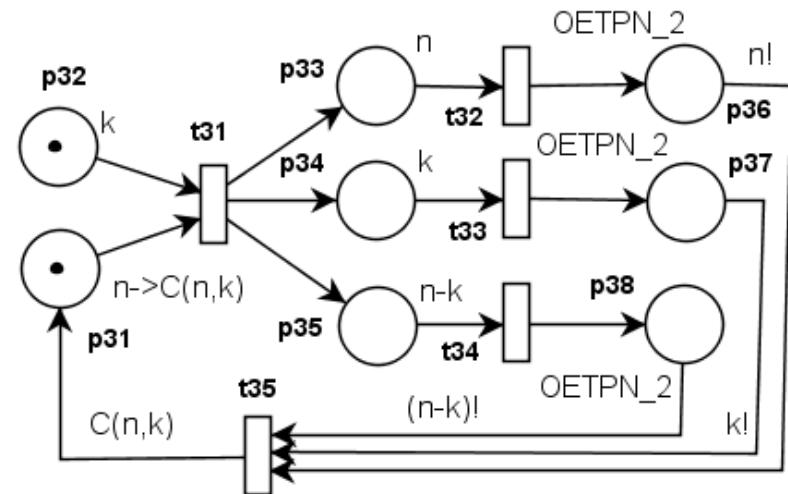
Where v is the duration of a thread instantiation. δ is the duration of a multiplication.

When is it benefic to use the multitasking from the implementation perspective?

Is it better to use an object with a method implementing the factorial?

Should be there 3 objects implementing the factorial method?

Homework: write the ETPNL description.



4. OETPN model properties

OETPN model state vs. program state (i. e. model execution state)

The OETPN model state

- the marking $M=[M(p_1), M(p_2), \dots, M(p_m)]$,
- pending transition execution,
- the active object state (executing, stopped),
- input and output channel states are included in markings \leftarrow managed by the execution environment.

The program has:

- Passive objects,
- Active objects and
- Pending events.

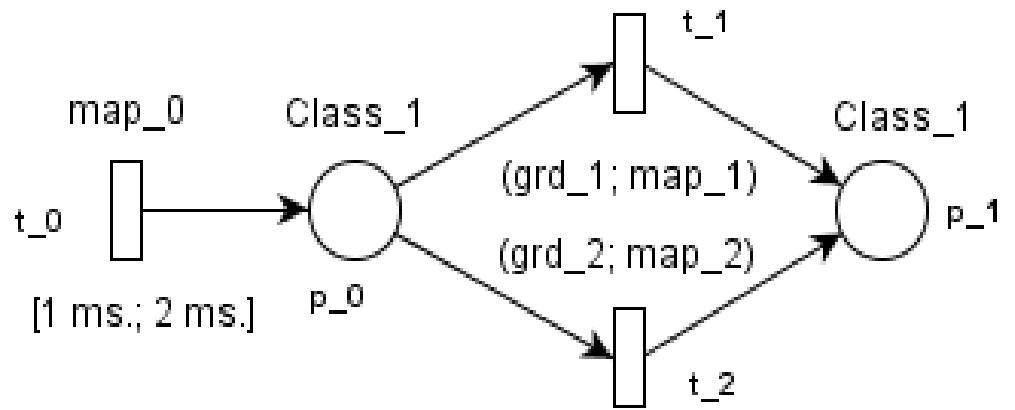
Selection

OETPN selection by:

- Different transitions
- Different rules assigned to the same transitions.

- $p_0, p_1: Class_1 \{x:float; y:float\}$
- $t_0: \{(grd_0:=true; map_0:= (p_0.x=random(); p_0.y=random()))\}$
- $t_1: \{(grd_1:= x>y); map_1:= (p_1.x=p_0.x-p_0.y; p_1.y=p_0.x+p_0.y)\}$
- $t_2: \{(grd_2:= x<y); map_2:= (p_1.x=p_0.x+p_0.y; p_1.y=p_0.y-p_0.x)\}$

Selection with different transitions



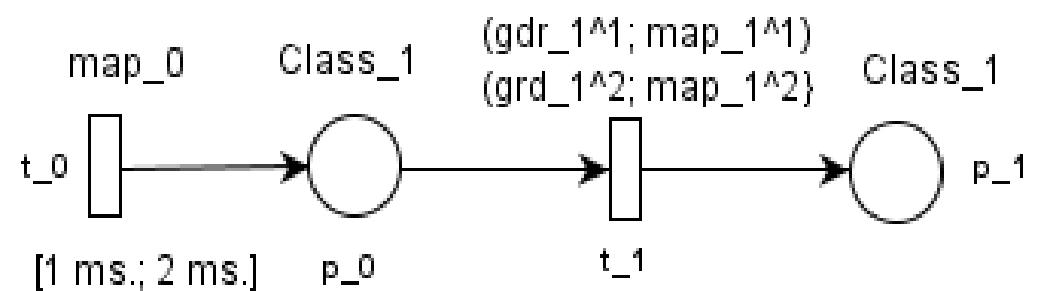
The conflicts can be avoided guard conditions.

Selection achieved by different rules (grd, map).

- $grd_{1^1} := grd_1; map_{1^1} = map_1$
- $grd_{1^2} := grd_2; map_{1^2} = map_2$

Rational: diminish the graph dimension.

Selection based on different guards.



Inhibitor and reset arcs

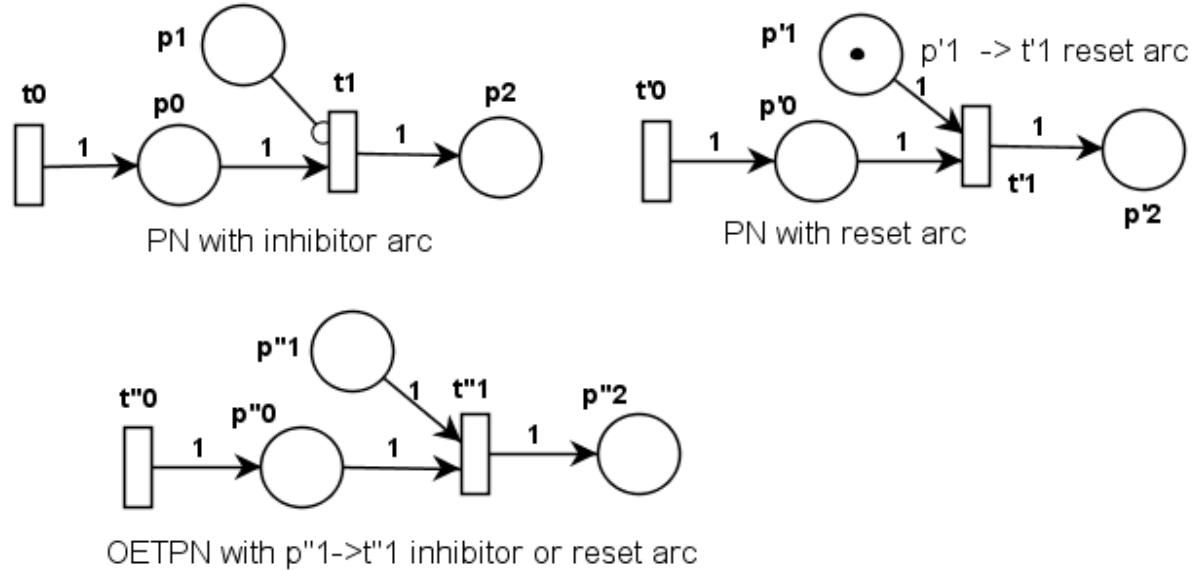
Goal: construct an OETPN model that behaves as PN with inhibitor arcs, or as OETPN with reset arcs.

OETPN inhibitor arc

t_1 :

$\text{grd}^1_1 := (\text{M}(p_1) = \varphi \text{ and } \text{M}(p_0) \neq \varphi);$

$\text{grd}^2_1 := ((\text{M}(p_1) \neq \varphi \text{ and } \text{M}(p_0) \neq \varphi) \text{ OR } (\text{M}(p_1) = \varphi \text{ and } \text{M}(p_0) \neq \varphi));$



Utilization:

- inhibitor arc – in certain condition blocks the execution;
- reset arc – in certain condition use the information stored in p_1 , but if it missing go further
- the mappings are conceived according to the need.

Non-blocking read

$Inp = \{p_3\}; Out = \{p_4\}$

ETPNL description:

$t_1[p_3;\varphi]^*t_2[\varphi;p_4]$

Goal: read p_3 , but if the information is missing, go further and notice that.

Blocking avoidance:

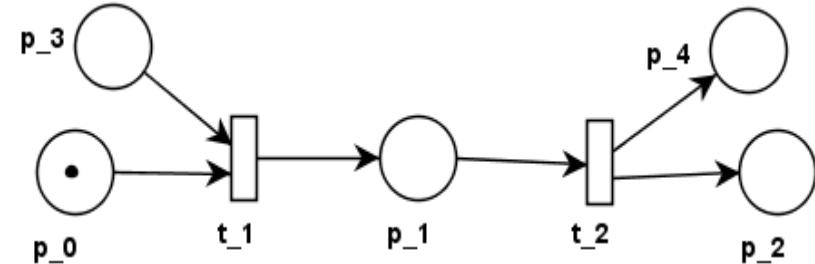
$t_1: \{(grd^1_1 := (M(p_0) \neq \varphi \text{ and } M(p_3) \neq \varphi), map^1_1);$
 $(grd^2_1 := (M(p_0) \neq \varphi \text{ and } M(p_3) = \varphi), map^2_1)\}$

Specification:

- $\text{type}(p_3): \{x: \text{float}\}$
- $\text{type}(p_1): \{x: \text{float}; v: \text{Boolean}\}$

Transition t_2 is conceived with 2 rules (when the read was performed or not) including different mappings:

$t_2: \{(grd^1_2 := (p_1.v), map^1_2); // \text{read performed}$
 $(grd^2_2 := p_1.v = \text{false}), map^2_2\} // \text{read not performed}$

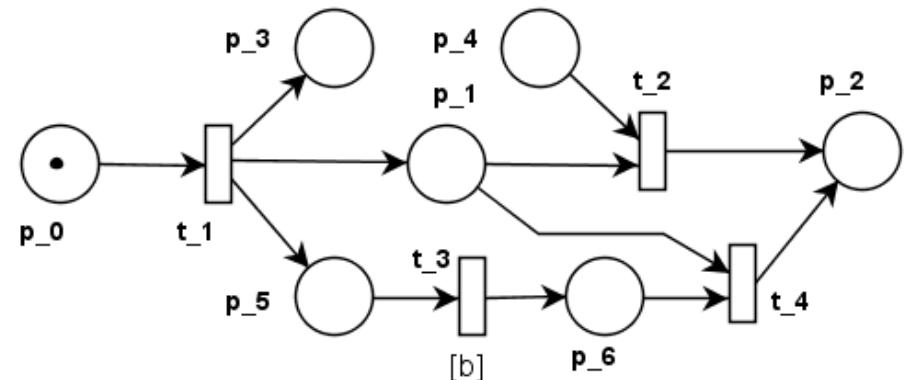


Bounded read wait (i. e. bounded blocking read)

Goal: the task waits a limited time to receive the demanded information.

Inp={p4, p6}; Out={p3, p6}

ETPNL description:

$$\begin{aligned} t1[\varphi;p3]^*(t2[p4;\varphi]\&(t3[b;\varphi]^*t4[p1;\varphi]) = \\ t1[\varphi;p3]^*(t3[b;p6] \& (t2[p4;\varphi]+(t4[p6;\varphi])) \end{aligned}$$


The value b specifies (in t.u.) the maximum waiting time. Channel p_3 demands the information. If the read is not performed in maximum b t.u., the task goes further performing something else (i. e. t_4 instead of t_2).

Transition t_2 performs the read by:

$$\begin{aligned} \text{grd2:}= (\text{M}(p1) \neq \varphi \text{ and } \text{M}(p4) \neq \varphi); \text{map2} \\ \text{grd4:}= ((\text{M}(p1) \neq \varphi \text{ and } \text{M}(p6) \neq \varphi); \text{map4}) \end{aligned}$$

Different values are set in p_2 if the read was performed or not.

If the read was performed, the token set in p_6 remain unused. How can be avoided its use in a next bounded read.

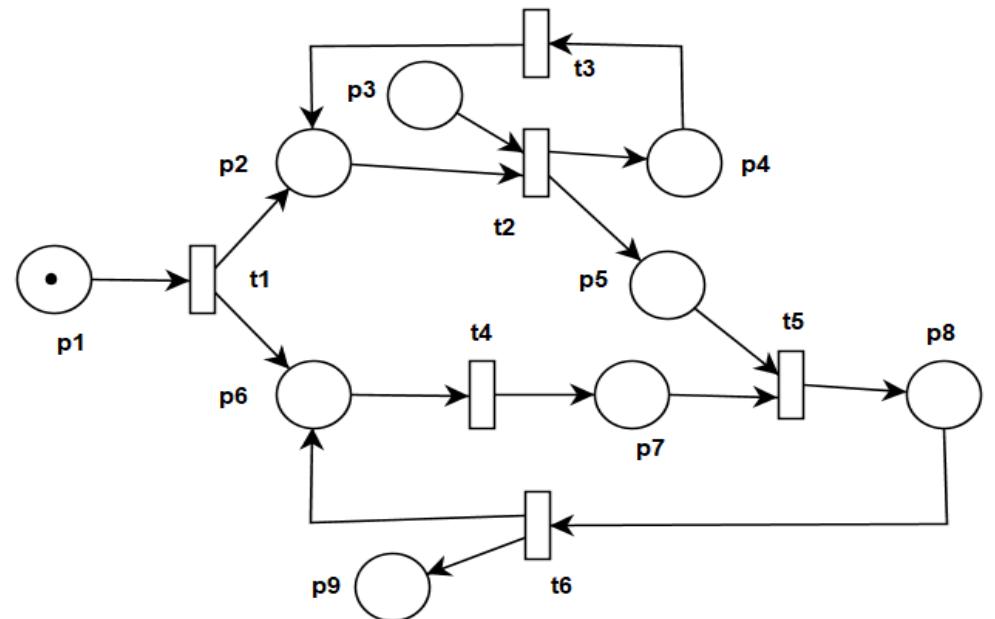
Non-blocking operations and non-blocking communication

Goal: Tasks with and without synchronization.

Inp={?}; Out={?}

PNL descriptions:

- $t2 \# t3$
- $(t4 * t5) \# t6$.



Homework: write the ETPNL descriptions

Conceive the relevant guards and mappings for the cases:

- t2 reads p4 set a value in p6; t5 reads the value set in p6 and moves further;
- t2 cannot read p4 and set a value in p6; t5 is executed without the read of p6

Un-end loop and executions

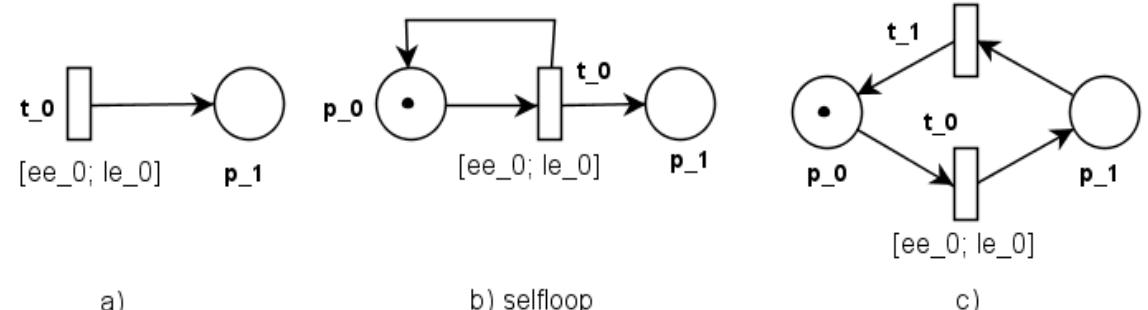
(Bucle și execuții fără sfârșit)

Goal: Models that generate periodically tokens in the place p_{-1} .

- a) t_0 generates deterministically the same tokens or randomly different tokens;
- b) and c) t_0 can generate tokens considering the initial condition set in p_0 .

The difference between b) and c): c) “consumes” the initial token, unlike b) that replaces it

Infinite executions



Deadlock and deadlock avoidance

PNL description:

$$\sigma_1 = (t_1 * t_2 * t_3) \# t_4$$

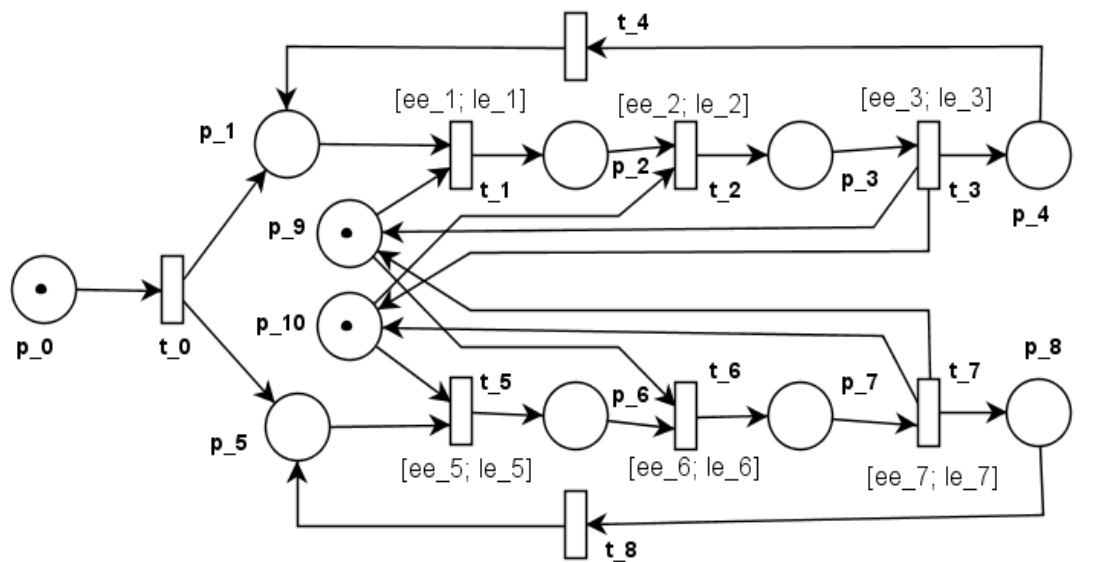
$$\sigma_2 = (t_5 * t_6 * t_7) \# t_8.$$

The places p_9 and p_{10} correspond to two resources successively (sequentially) acquired by the tasks. This can lead to deadlock.

Goal: If the system is deadlocked

- a) remove it by rolling back \leftarrow release the resource
- b) Avoid the entrance in deadlock by *trylock* \leftarrow try to acquire a resource but do not do it if the other is reserved.

Deadlock



Solution: conceive the guards of tranzitions t_2*t_3 and t_6*t_7 .

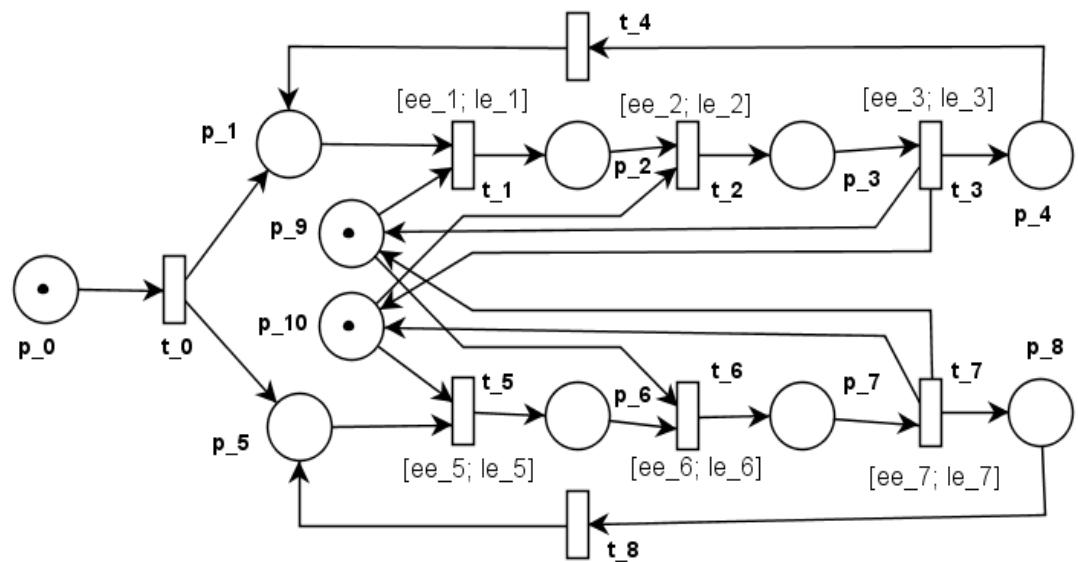
Types of the tokens in p_3 and p_7 contain Boolean variables $v1$ și $v2$ marking if the resources a reserved.

For upper task t_2 and t_3 :

- t_2 :
 - $(\text{grd_2}^1 := (p_2 \neq \varphi \text{ and } p_{10} \neq \varphi); \text{map_2}^1 := (p_3.v1 = \text{true}; p_3.v2 = \text{true}),$
 - $(\text{grd_2}^2 := (p_2 \neq \varphi \text{ and } p_{10} = \varphi); \text{map_2}^2 := (p_3.v1 = \text{true}; p_3.v2 = \text{false}))$
- t_3 :
 - $(\text{grd_3}^1 := (p_3.v1 = \text{true} \text{ and } p_3.v2 = \text{true}); \text{map_3}^1 := (\text{use the resources, release the resources})$
 - $(\text{grd_3}^2 := (p_3.v1 = \text{true} \text{ and } p_3.v2 = \text{false}); \text{map_3}^2 := (\text{release the resource } p_9))$

Tranzitions t_6 și t_7 have assigned similar rules (grd, map).

Soluția de mai sus realizează ceea ce se numește în software engineering „roll-back”.



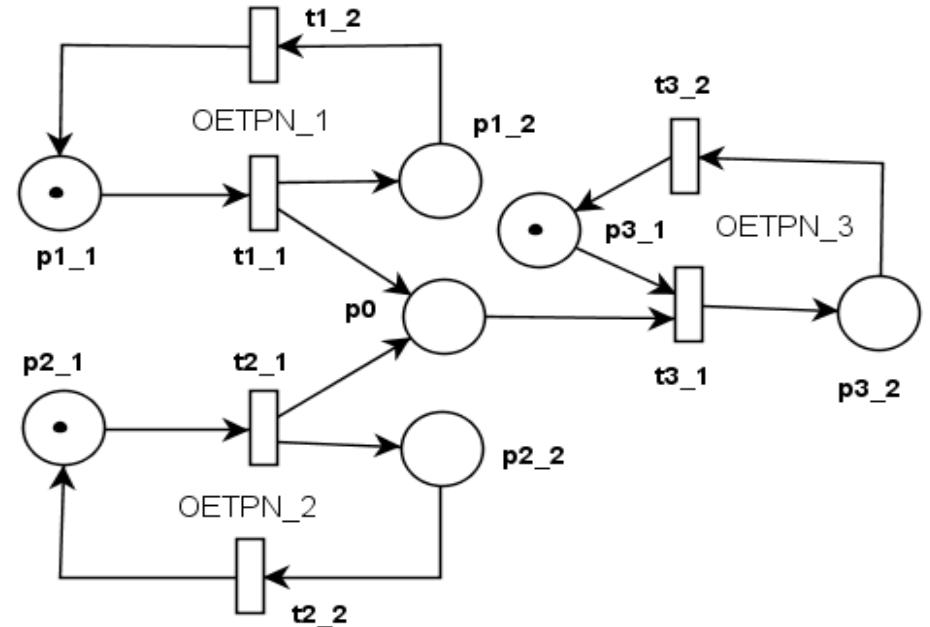
5. Examples of OETPN models

Partitioned OETPNs with shared output channels

$\text{Out}_1 = \text{Out}_2 = \{p_0\}$; $\text{Inp}_3 = \{p_0\}$

OETPN₁, OETPN₂, OETPN₃ are executed by different tasks (threads).

Solution to avoid the uncertain behavior: mutual exclusion for writing.



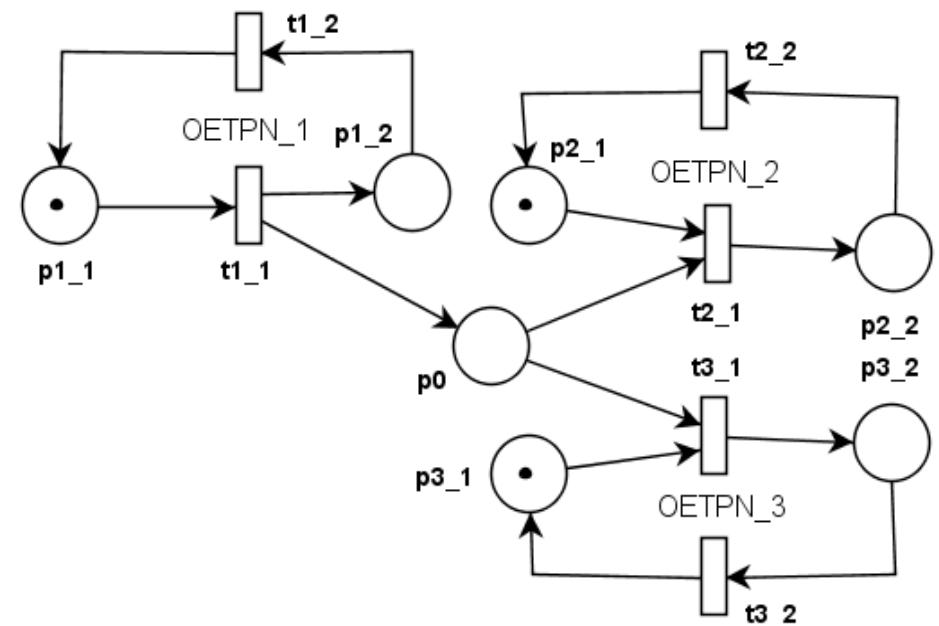
OETPN models with shared inputs

Submodels OER-TPN_2 and OER-TPN_3 take the token from p0 → conflict.

Independent OTPN executors.

Solutions in OETPN:

- Set in p0 tokens with “address”. Conceive the *grds* of t2_1 and t3_1 to discern the destination of information.
- Use the mutual exclusion → semaphore



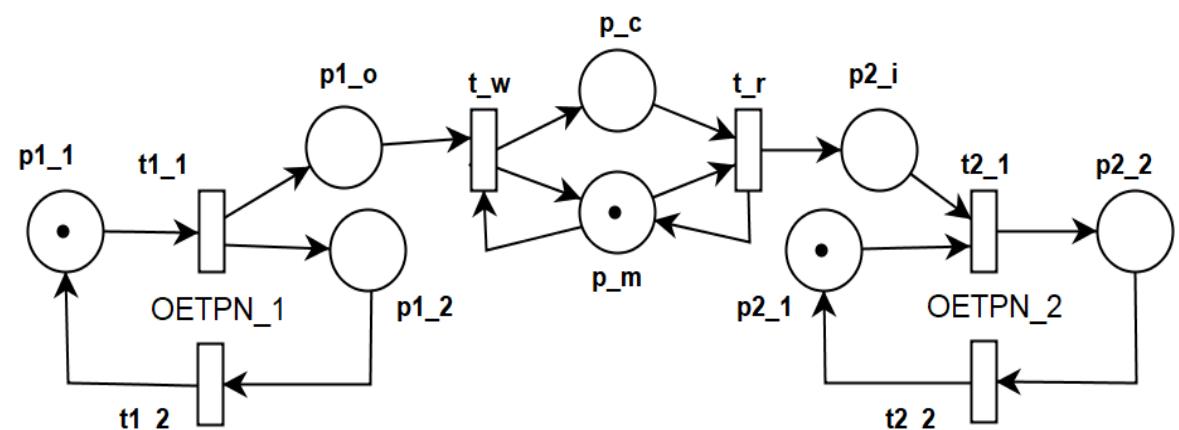
OETPNs communication

The module $t_w * t_r$ serves for communication.

$\text{Out}_1 = \{p1_o\}; \text{Inp}_2 = \{p2_i\}$

p_c serves as place for communication.
 p_m serves for mutual exclusion.

The task system can be implemented with or without synchronization.



Mutual exclusion and waiting queue

Semaphores → waiting queue: with FIFO (first-in-first-out) or non-FIFO.

The semaphore is implemented by the parent or by one of the synchronized tasks.

Semaphore primitives: init, request (acquire) and release.

The semaphore is a passive object.

Semaforul este creat ca un obiect OETPN de către un părinte (eventual metoda *main()*) și este utilizat de către modele OETPN pentru sincronizare folosind canalele de intrare/ieșire. Semaforul este un obiect pasiv (pasive block) care are metodele *init*, *request* și *release* pentru inițializarea lui (setarea valorii inițiale), obținerea accesului și respectiv renunțare la acces.

Mutex Class

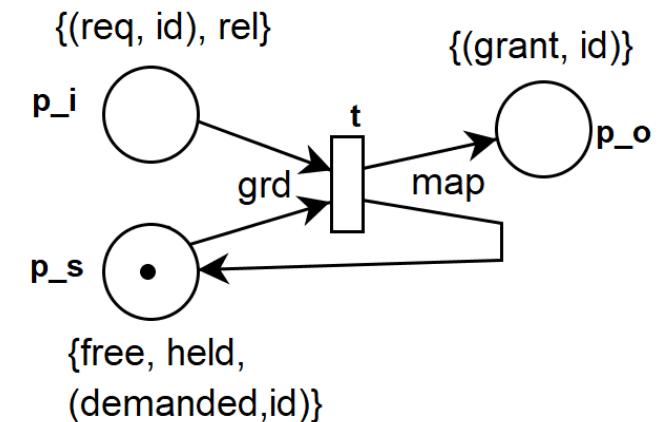
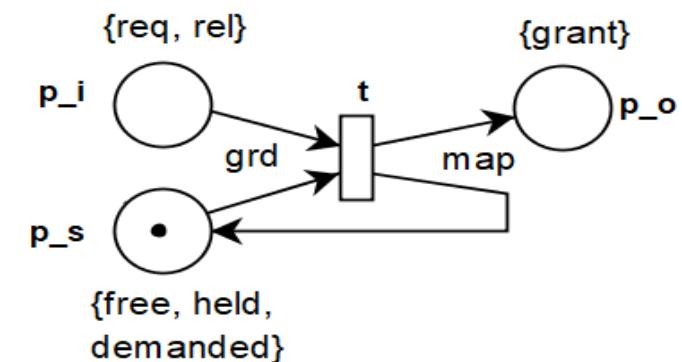
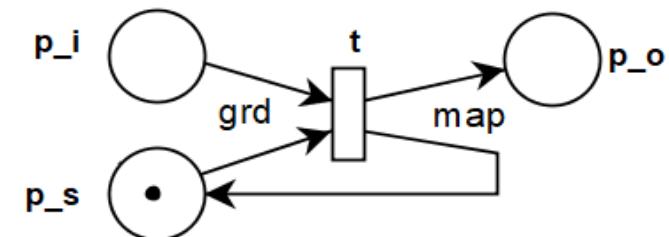
- req: request;
- id: requester identifier;
- rel: release;
- grant: signal; idem id;
- free: the semaphore state is free;
- held: the semaphore is granted to an OER-TPN;
- demanded: the semaphore is requested, and it will be granted when an OETPN task releases it.
- p_i: input channel place
- p_o: output channel place
- p_s: mutex state place

$\text{type}(p_i) = \text{type}(p_s) = \text{type}(p_o) = \text{String};$

Evolution relations:

$\text{grd}^1 := (p_i.value = (\text{req}, \text{id}) \ \& \ p_s.value = \text{free});$

$\text{map}^1 := (p_o.value = (\text{grant}, \text{id}); p_s.value = \text{held});$



grd² := p_i.value = rel & p_s.value = (demanded,id);

map² := p_o.value = (grant,id); p_s.value = held;

grd³ := p_i.value = (req,id) & p_s.value = held;

map³ := p_o.value = φ; p_s.value = (demanded,id);

grd⁴ := p_i.value = rel & p_s.value = held;

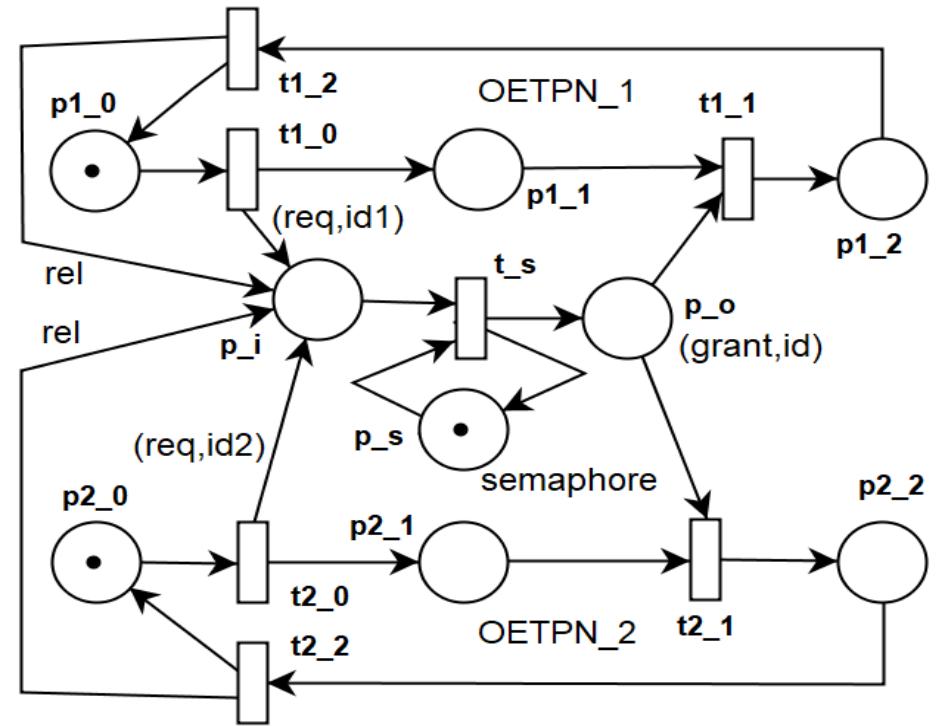
map⁴ := p_o.value = φ; p_s.value = free;

OETPN models synchronization

Goal: guarantee the FIFO.

Homework: write the guards and mappings.

.



6. Utilization of OETPN Models

OETPN model synthesis

Partition the application using components.

Conceive the model (algorithm) for each component:

- conceive the OETPN structure;
- conceive the information structure;
- conceive the guards and mapping, as well as the *eet* and *let* when they are executed;
- conceive the model interfaces with its environment and for their cooperation/collaboration.

⇒ ***Structure and behavior***

Synthesis methods:

- Analytical → humans add or remove information, properties, attribute or operations;
- Automatic → program synthesizes them using simulations and improvement algorithms
 - Heuristic
 - Deterministic
- Interactive → human-program cooperations
 - Simulations
 - Assessing methods
 - Numerical and graphical

Control Synthesis of Reactive Applications

Cyber-physical system example

Plant <-- generic term ← hardware implemented
(vehicle, tool, robot, device ~software can be added)

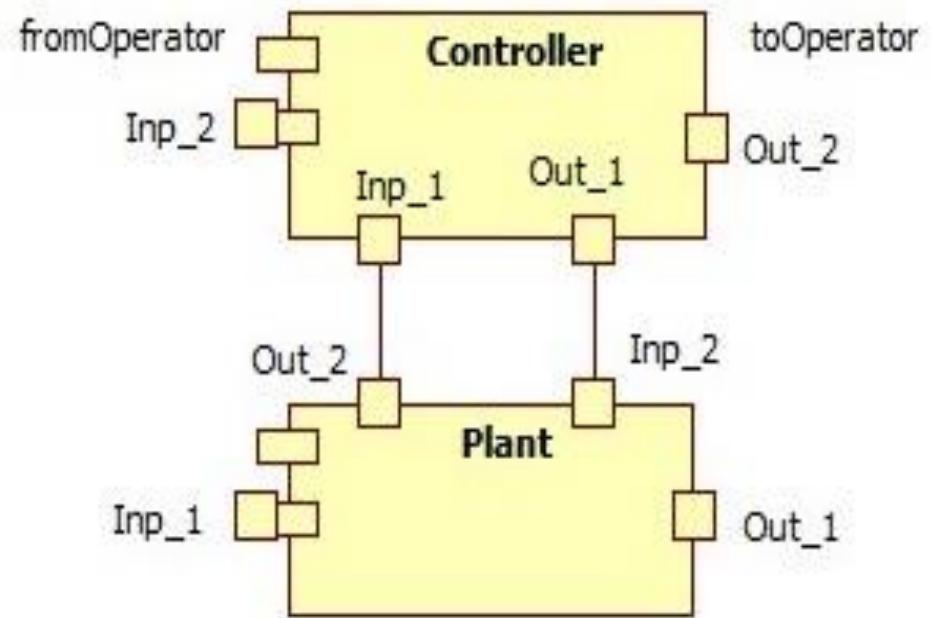
Plant has specified:

- Initial conditions
- Input and output channels
- behavior

Controller <-- generic term ← software implemented

Specify the Plant-Controller interactions:

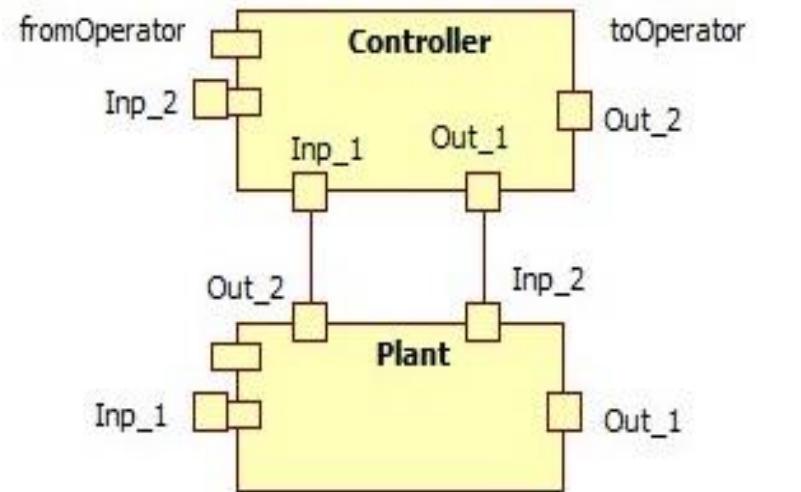
- sensors, transducers etc.
- effector, actuators etc
- events
- continue variables – flow information (data)



- Event driven
- Time driven

Se precizează pentru Plant:

- canalele de intrare Inp_1 care crează evenimente sau fluxuri de date necontrolate și necunoscute (de către Controller) în general; tipurile acestor informații trebuie să fie compatibile cu locațiile instalației; uneori operatori umani pot intra în interacțiune cu instalația prin unele dintre canale de intrare din setul Inp_1 ;
- Canalele de ieșire Out_1 care generează evenimente și fluxuri de date în exteriorul instalației, dar care nu sunt (în general) măsurate direct (sau cel puțin, nu toate) de către Controller; uneori operatori umani pot primi informații despre instalație prin intermediul unor canale din setul Out_1 ;
- Canalele de ieșire Out_2 furnizează evenimente și fluxuri de date care pot fi preluate de către Controller; tipurile acestora trebuie să fie compatibile cu tipurile locațiilor corespunzătoare din Controller;
- Canalele de intrare Inp_2 sunt furnizoare de informații (controlabile) către instalație; trebuie precizate dacă sunt de tip evenimente sau fluxuri de date și se specifică tipurile lor;

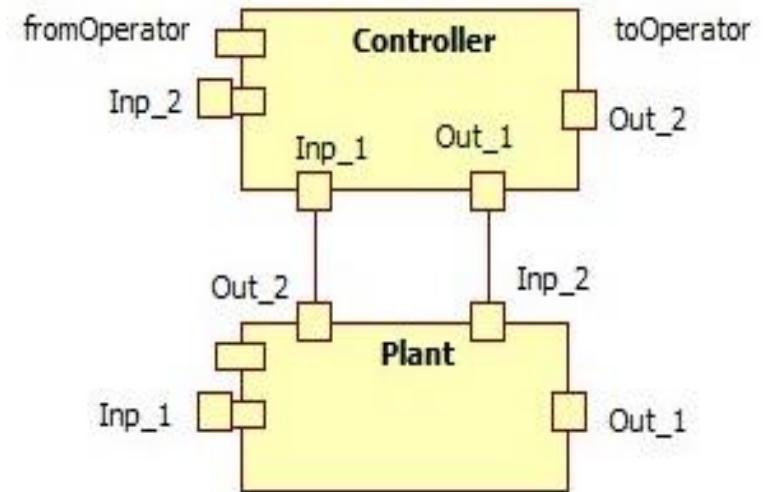


Prin *controller* se înțelege aici un program care interacționează cu Plant și îl determină pe acesta din urmă să aibă o evoluție cerută prin cerințele de specificare.

Cerințele de specificare (engl.: *specification requirements*) descriu cum trebuie să se comporte Plant.

Este posibil ca un operator uman să interacționeze cu Controller sau chiar și cu Plant în timpul evoluției.

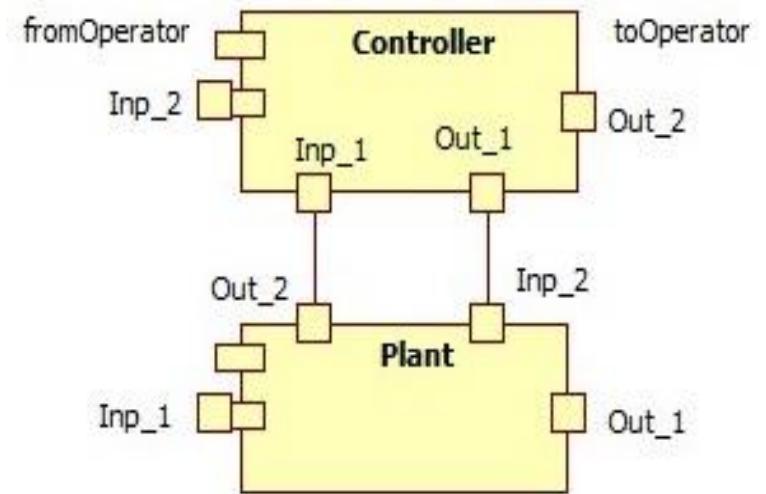
Cerințele pot include texte furnizând informații într-o formă informală.



Din cerințele de specificare se construiește modelul OETPN_R (sau un set de modele) care descrie formal modul de comportare cerut al lui Plant.

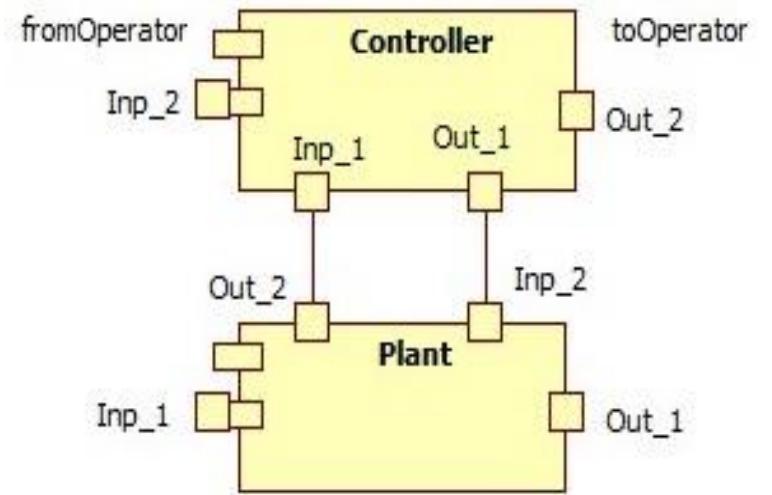
Exemple de cerințe:

- Dacă Plant se află într-o stare precizată și dacă intrările primesc informații (evenimente sau fluxurile date) specificate, atunci Plant trebuie să evolueze pe o anumită traекторie cerută de stări, sau să evite o stare, eventual o secvență de stări;
- Dacă Plant se află într-o anumită stare și se produce un eveniment dat, Plant trebuie să reacționeze conform unor cerințe date, cum ar fi să producă un eveniment sau o secvență de evenimente;



Specificațiile controllerului conțin:

- Canalele de intrare **Inp_1** prin care se pot primi informații de la Plant; ele trebuie să fie compatibile cu canalele de ieșire din Plant; ele pot fi de tip eveniment sau fluxuri de date;
- Canalele de ieșire **Out_1** prin care Controller poate influența comportamentul lui Plant; acestea furnizează informații de tip eveniment sau fluxuri de date; ele trebuie să fie compatibile cu canalele **Inp_2** din Plant;
- Canalele **Inp_2** servesc pentru preluarea unor comenzi de la operator sub forma de evenimente sau fluxuri de date;
- Canalele **Out_2** furnizează informații operatorului sub formă de evenimente sau fluxuri de date; sunt necesare dispozitive capabile pentru preluarea sau afișarea tipului de informații furnizat.

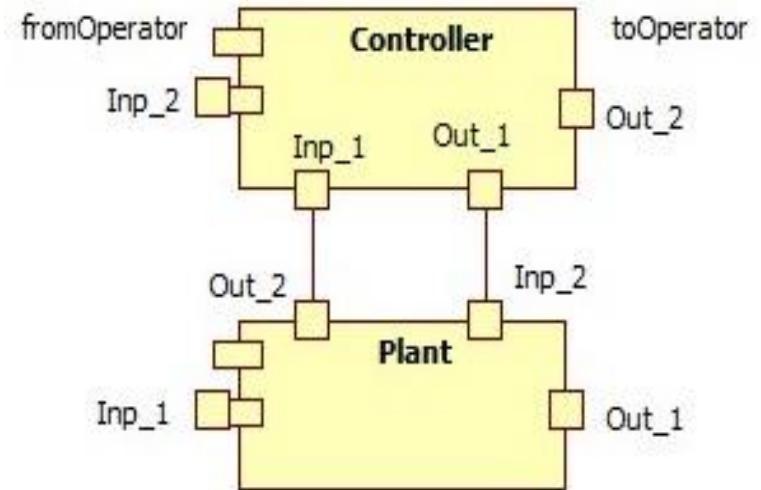


Problema formală a sintezei controllerului:

- determinarea modelului controllerului OETPN_C
- determinarea stării inițiale a lui OETPN_C pentru cazul în care OETPN_P se află în starea M_P^0 , iar comportamentul lui Plant corespunde cerințelor OETPN_R cu M_R^0 .

Pentru unele probleme se specifică intrările din exterior pentru Plant și pentru Controller.

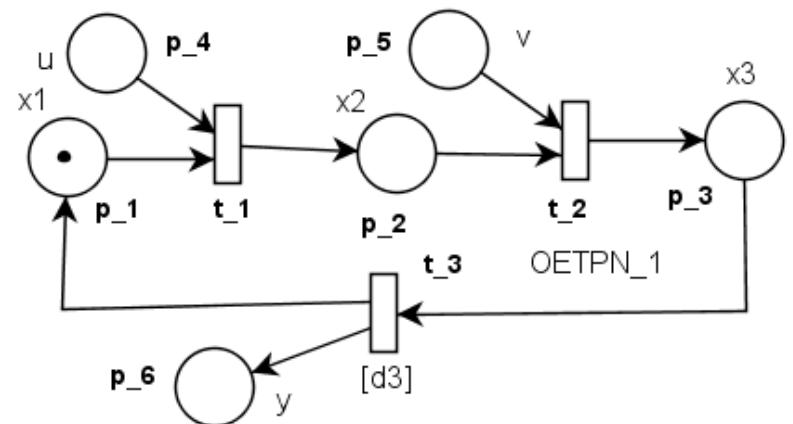
Primele se numesc uneori perturbații, iar cele din urmă exprimă deseori dorințele sau comenziile operatorului.



Example: Control OETPN model synthesis

Inp = {p₄, p₅}; Out = {p₆}
 u control input;
 v disturbance;
 y measurement;
 d₃ delay;
 x₁, x₂, x₃ plant state variables.

Plant model.



Declarații

Se dă modelul OETPN_1 reprezentat în Fig.. cu specificațiile:

$\text{type}(m(p_i)) = \{x: \text{float}\}; i = 1, 2, \dots, 6$

d₃ este o întârziere fixă care determină perioada de modificare a stării.

Starea este reprezentată de [x₁, x₂, x₃, u, v, y].

Setul canalelor de intrare este: Inp={p₄, p₅}

Setul canalelor de ieșire este: Out={p₆}

Notations:

$x_i = m(p_i).x$; x stored in the token set in p_i ; $i = 1, 2$,

$u = m(p_4).x$; $v = m(p_5).x$; $y = m(p_6).x$;

Initial state: $x_1=10$

Locațiile p_4 și p_5 sunt canale de intrare, cu u o variabilă de control, iar v o perturbație aleatoare.

Constraints: $-1 \leq v \leq 1$; $-2 \leq u \leq 2$;

Evolution rules:

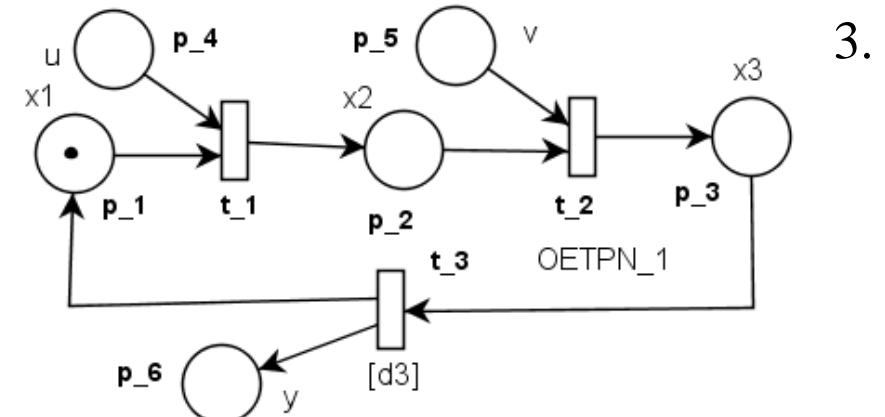
- t_1 :

- $grd^1_1 := ((m(p_1) \neq \varphi) \text{ and } (m(p_4) \neq \varphi)) \rightarrow map^1_1 := x_2 = x_1 + u;$
- $grd^2_1 := (m(p_1) \neq \varphi) \text{ and } (m(p_4) = \varphi) \rightarrow map^2_1 := x_2 = x_1;$

- t_2 :

- $grd^1_2 := ((m(p_2) \neq \varphi) \text{ and } (m(p_5) \neq \varphi)) \rightarrow map^1_2 := x_3 = x_2 + v;$
- $grd^2_2 := (m(p_2) \neq \varphi) \text{ and } (m(p_5) = \varphi) \rightarrow map^2_2 := x_3 = x_2 - 0.1;$

- t_3 : $grd_3 := m(p_3) \neq \varphi \rightarrow map_3 := (y = x_3; x_1 = x_3);$



Controller model.

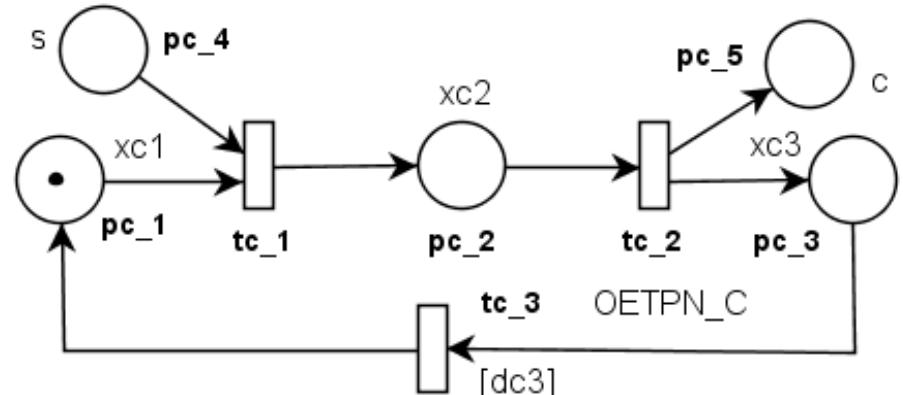
Requirements – performance

Cerințe

Se cere să se determine (sintetizeze) un controller pentru OETPN_1 astfel încât să se mențină valoarea lui y cât mai aproape de 5.0: $\|5.0 - y\| = 0$. (de fapt $\rightarrow 0$)

Se mai pot da ca cerințe:

- abaterea maximă a lui y față de valoarea 5.0 în contextul în care
 - v este un eveniment asincron având valoarea amplitudinii maximă sau minimă admisă
 - v este un flux de date având un anumit profil specificat prin frecvență, amplitudine (eventual variația ei), offset
- durata de corecție (răspuns) a unei abateri de tip eveniment asincron sau variații de tip treaptă în fluxul de date



Proposed solution OETPN_C

Link $p_6 \rightarrow pc_4$; $pc_5 \rightarrow p_4$

Canalul de intrare pc_4 trebuie conectat cu p_6 , iar cel de ieșire pc_5 cu p_4 .

$\text{type}(pc_1) = \text{type}(pc_2) = \text{type}(pc_3) = \text{type}(pc_4)$

$= \text{type}(pc_5) = \{x: \text{float}\}$

Notations: $xc1 = m(pc_1).x$; $xc2 = m(pc_2).x$;

$xc3 = m(pc_3).x$; $s = m(pc_4).x$; $c = m(pc_5).x$;

Proposed rules:

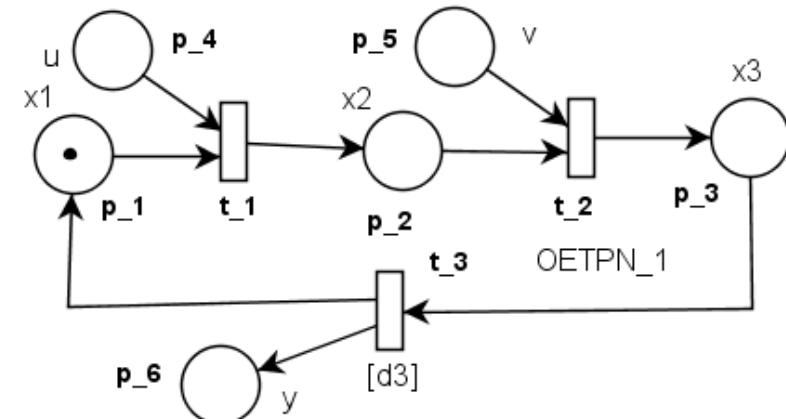
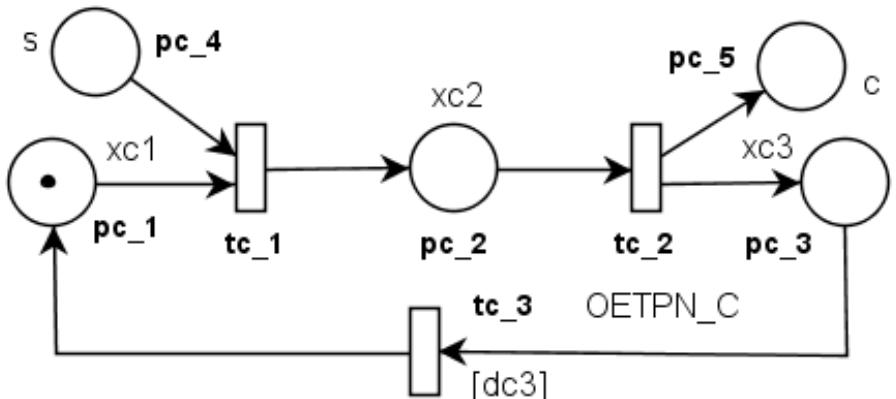
- tc_1 :

- $\text{grd}^1_1 := ((m(pc_1) \neq \varphi) \text{ and } (m(pc_4) \neq \varphi)) \rightarrow \text{map}^1_1 := xc2 = u;$

- $\text{grd}^2_1 := (m(pc_1) \neq \varphi) \text{ and } (m(pc_4) = \varphi) \rightarrow \text{map}^2_1 := xc2 = xc1;$

- tc_2 : $\text{grd}_2 := (m(pc_2) \neq \varphi) \rightarrow \text{map}^1_2 := (xc3 = xc2; c = sc2 - 5.0;)$

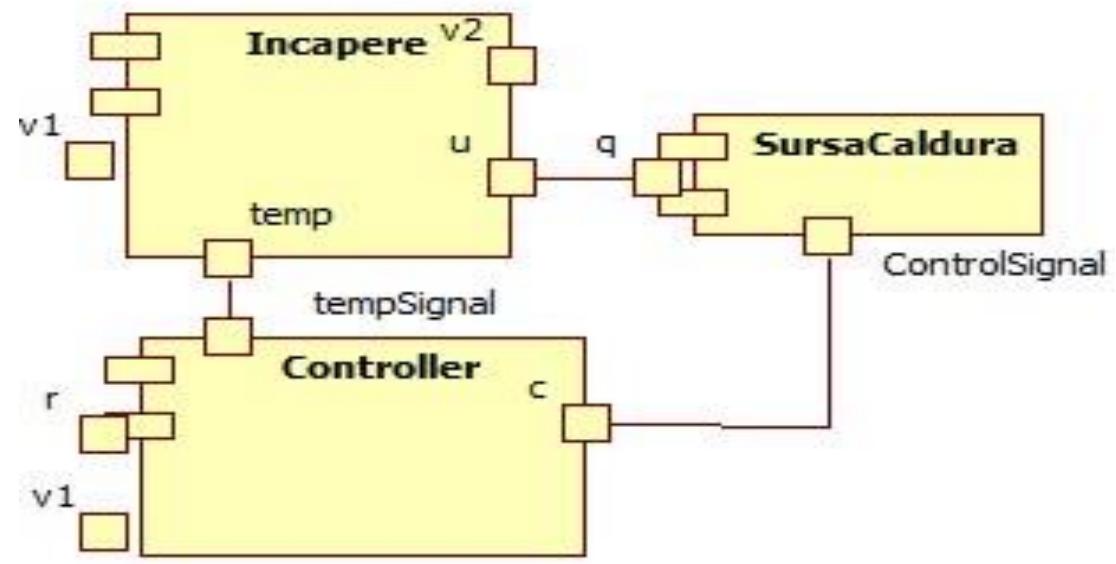
- tc_3 : $\text{grd}_3 := m(pc_3) \neq \varphi \rightarrow \text{map}_3 := xc1 = xc3;$



Room temperature control

Se consideră cazul unei încăperi conectată la o sursă de căldură în care trebuie să se controleze temperatura conform referinței r date de un operator.

Temperatura din încăpere este perturbată de răcirea determinată de temperatura variabilă $v1$ a mediul înconjurător și de căldura $v2$ variabilă introdusă de radiațiile solare.



Specificații:

Temperatura x din încăpere variază conform ecuației:

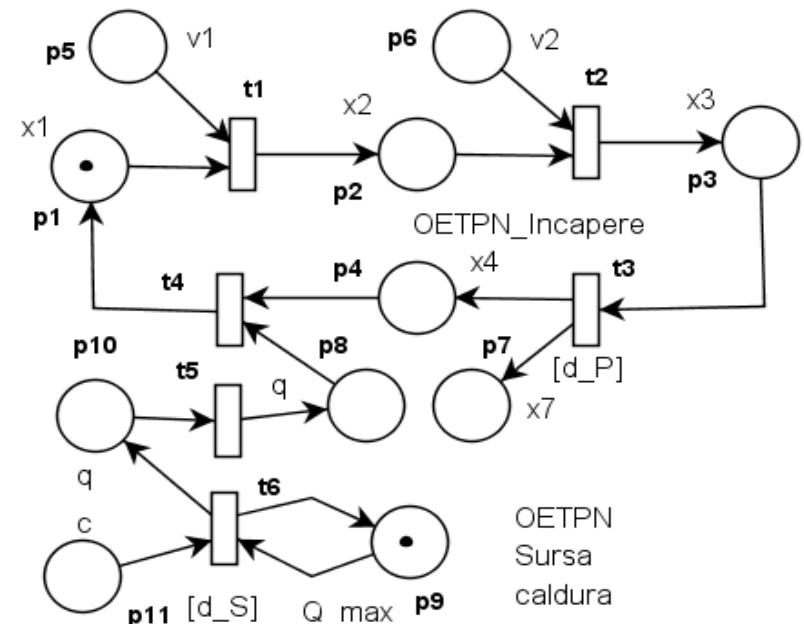
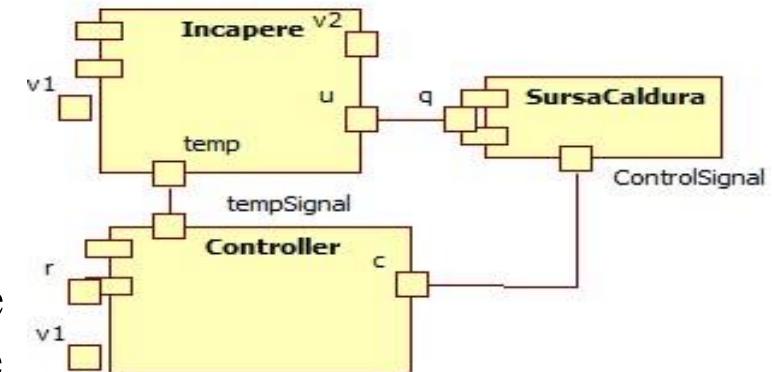
$$x(t+1) = x(t) - a((v1 - x), t) + b \cdot v2(t) + u(t)$$

cu $u(t) = e \cdot q(t)$ și $a((v1 - x), t)$ o funcție de răcire dependentă de diferența dintre temperaturile din încăpere și mediu. e este coeficientul de transformare (conversie) a căldurii (energiei) în temperatură.

Sursa de căldură generează cantitatea de căldură:

$$q(t) = c \cdot Q_{max}$$

cu Q_{max} energia (căldura) maximă pe care o poate furniza sursa, iar c comanda referitor la cantitatea debitată la momentul t pe durata unui tact.



Specificații controller

Controllerul primește referința r și valoarea temperaturii exterioare $v1$.

El calculează valoarea comenzi c pe care o transmite sursei de căldură. Până la modificarea referinței sistemul lucrează cu valoarea anterioară.

Temperatura $v1$ poate varia între -20 și +40 °C.

Perturbația nemăsurată $v2$ poate influența temperatura între +10 și +20 °C.

Cerințe de performanță:

Să se controleze temperatura conform referinței cu o abatere maximă de -1, +1 °C.

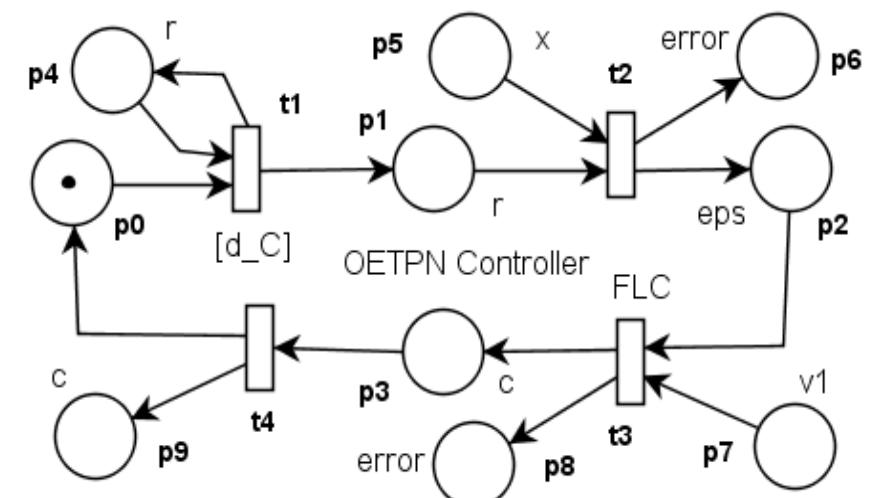
Observații:

Performanța controlului temperaturii poate fi influențată de gradienții variațiilor perturbațiilor $v1$ și $v2$.

Dacă poate fi situația pentru $v1$ astfel încât răcirea este mai mare decât capacitatea maximă de încălzire

$$a((v1 - x), t) > e \cdot Q_{max}$$

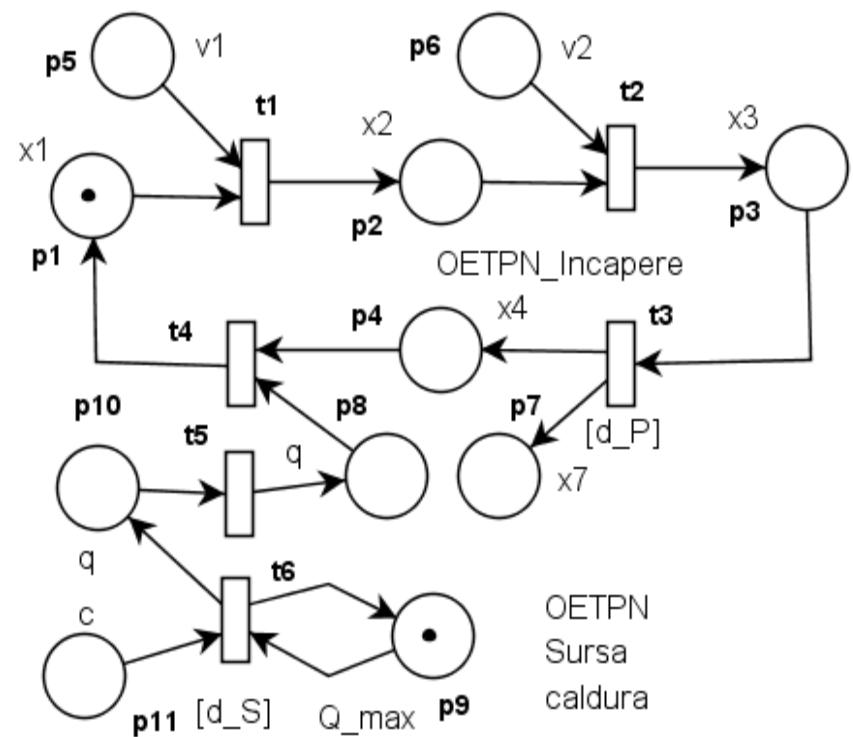
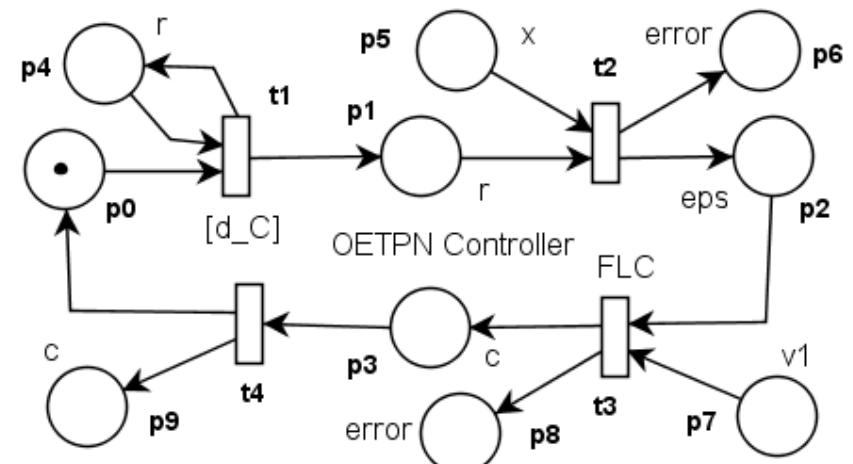
atunci sistemul nu poate îndeplini cerința de performanță.



Cerințe pentru situații anormale:

Dacă nu primește controllerul valoarea perturbației $v1$, sistemul lucrează mai departe, dar semnalează anomalia.

Dacă nu primește controllerul valoarea x a temperaturii interioare, acesta semnalează anomalia și nu mai introduce căldură în încăpere până la remedierea situației.



*

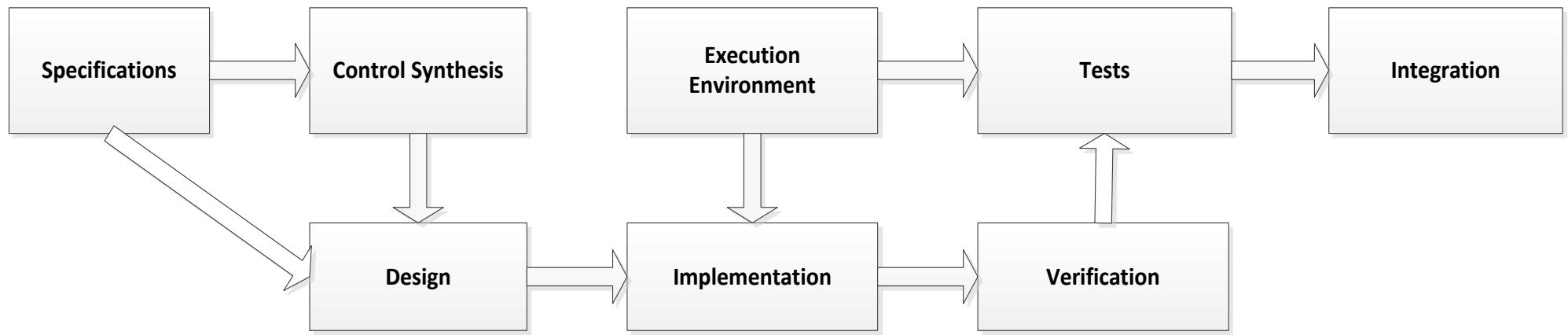
* * * *

END

* * * *

*

Ch. 3. Design of R-T Applications



Specification: non-formal (texts) or formal description

- declarations
- requirements

Verification of specifications → Petri net based verification ← formal verification

E.g.: UML Activity diagram ←→ Petri nets based verification

Design: structure (e.g. class diagrams = static verification) + behavior (dynamic verification)

Behavior ← state machines, sequence diagrams, object collaboration (communication diagrams) ←→ Petri nets

Implementation ←→ state machines ← Petri net based verification

- 3.1. Relations of UML diagrams with different kinds of PNs**
- 3.2. Transformation of Petri Nets into State Machines**
- 3.3. Extending UML**
- 3.4. Using UML Real-Time**
- 3.5. Task structure diagram**

3.1. Relations of UML diagrams with different kinds of PNs

Use case diagrams are used in analysis phase of a project to identify the system functionality. They describe the interaction of people or external device with the system under design.

Correspondence to PNs → ETPN. Formal verification: ETPNL.

Both describe the interaction with external devices and persons (i. e. actors).

Sequence diagrams are used in the analysis and design phases. They describe the interactions between different entities and provide details of how are carried out operations. They depict the structure and the chronology of event flows and data flows representing the communication relationships between objects. They show the order of events and synchronizations. Can describe the concurrent evolutions.

Correspondence to PNs → ETPN. Formal verification: ETPNL.

Activity diagrams are used to describe the procedural flow of actions as part of an activity. They model coordinated activities that provide services. They show the events needed to achieve some operation and represent the relations between events and activities. They show the order of events and synchronizations. Can describe the concurrent evolutions.

Correspondence to PNs → ETPN. Formal verification: ETPNL.

Object collaboration diagrams (OCD) = communication diagrams = interaction diagrams describe a collection of objects that interact to implement some behaviors. They model the system functionality and visualize the relationship between objects collaborating to perform tasks, and model the logic of the implementation for a complex operation. An OCD describe one task. They can show the order of events and the order of sending the messages.

Correspondence to PNs → OETPN. Formal verification: OETPN → ETPN → ETPNL.

State chart diagrams = state machine diagrams (SM) model the system dynamicity. They describe all the possible states of object and the events that involve these. They can model the object creation, the state modifications and its destruction. For more objects representations is needed an SM for each of them.

Correspondence to PNs: OETPN. Formal verification: OETPN → ETPN → ETPNL.

Class diagrams describe the objects with their information structures and show the communication with their users. They describe the responsibilities and model the static view of applications. They describe features such as: object configuration (composition, aggregation, association, generalization). They can represent the object dependencies (e.g. clients use suppliers).

Correspondence to PNs: OETPNs use the information structures in the place types. The arcs show which and where are used the information. OETPNs do not show the information structures. These must be described as model annotations.

Formal verification: the OETPN structure show relations of object, so can be verified if they are compatible.

Timing diagrams shows the temporal representation when events occur and the event temporal relations.

Correspondence to PNs: TPNs. Formal verification: TPNL.

Component diagrams (CD) are used to partition system's required functionality and to show where it has been distributed (deployed).

Correspondence to PNs → no PNs correspondence. CDs are added for software development. Formal verification: the components integrate tasks or structured information. The tasks can be modeled by PNs, the information structures no.

Deployment Diagrams are used to describe the hardware components of the system that implements the application, the execution environments and the artifacts deployed on the hardware. They show the hardware system topology modeling the hardware elements and their communication links.

Correspondence to PNs: OETPN can model the physical elements (object devices etc.) Formal verification: ETPNL for the concurrent behaviors.

Interaction overview diagrams show the flow of control between interacting nodes. They include initial nodes, flow final nodes, activity final nodes, decision nodes, merge nodes, fork nodes, and join nodes.

Correspondence to PNs: OETPN. Formal verification: ETPN → ETPNL.

UML Package Diagram → No PNs correspondence.

What features have OETPN diagrams and are missing in UML diagrams?

What features are better described in PNs than in UML?

3.2. Transformation of Petri Nets into State Machines (and reverse for verification)

PNs can be used for:

- specifications
- design
- verification

State Machines (SMs) were used for the design of applications.

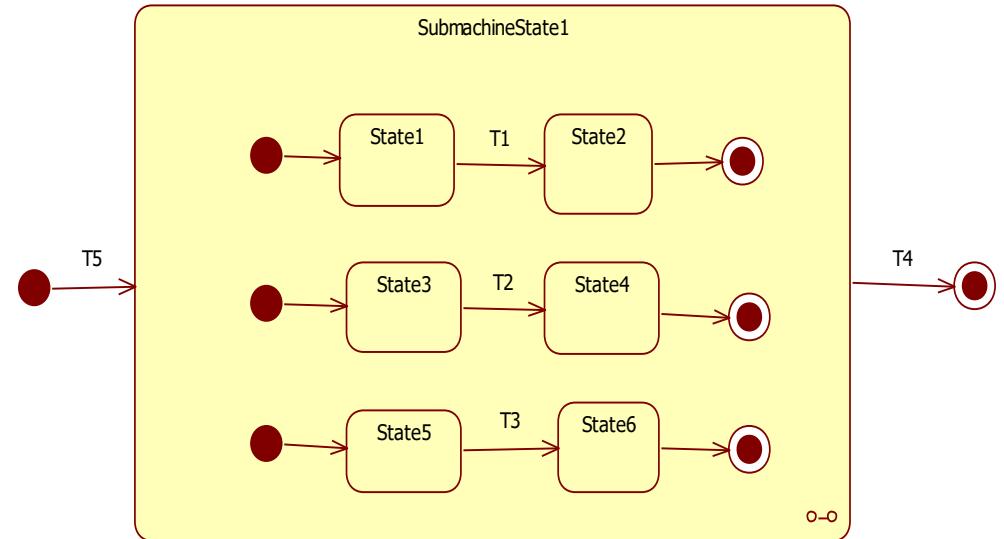
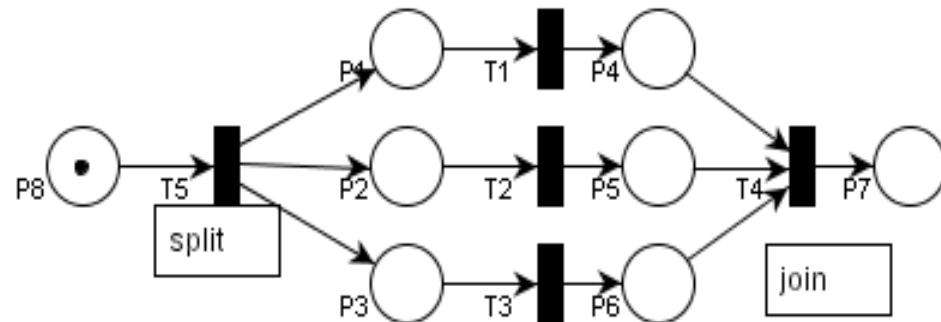
When and why is the transformation necessary?
Can the TPN be implemented using a single thread?
Can the TPN with IA (inhibitor arcs) be implemented using a single thread?

The reasons of transformation:

- PN → SM: implementation of the specified applications
- SM → PN: verification of the implemented applications

A PN is equivalent (isomorphic) to a set of SMs IFF both generate the same sequences of events (i.e. transitions).

How can a PN be transformed into a set of state machines?



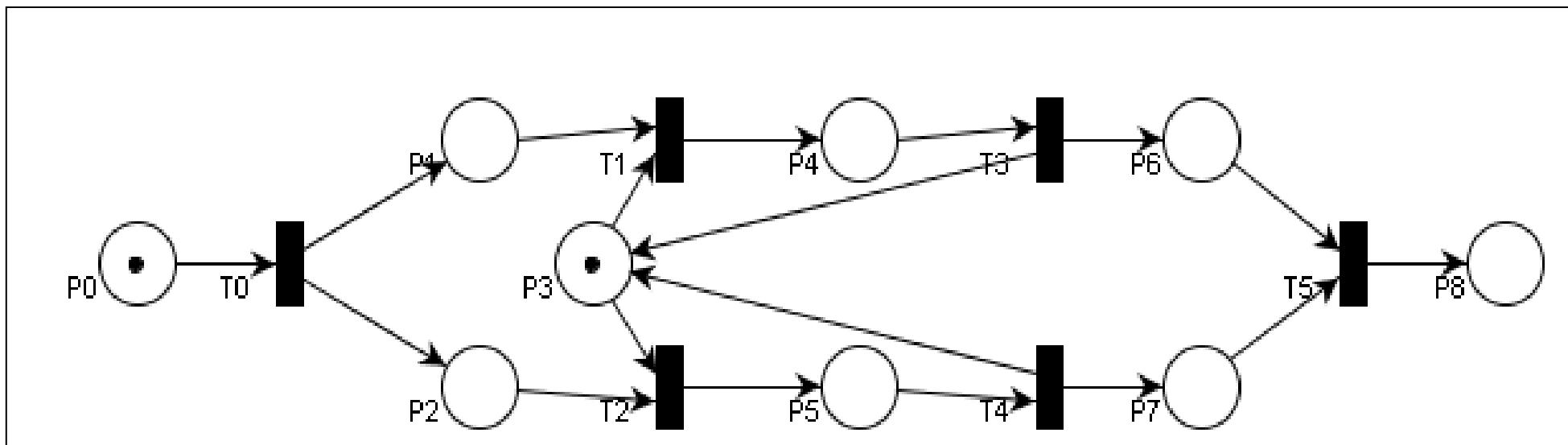
$$\sigma = T_5 \cdot (T_1 \& T_2 \& T_3) \cdot T_4$$

$$\sigma = T_5 \cdot (T_1 \cdot T_2 \cdot T_3 + T_1 \cdot T_3 \cdot T_2 + T_2 \cdot T_3 \cdot T_1 + T_2 \cdot T_1 \cdot T_3 + T_3 \cdot T_2 \cdot T_1 + T_3 \cdot T_1 \cdot T_2) \cdot T_4.$$

They have the same sequence of events. → They are equivalent.

SM transition notation: *Waited_event[Guard_condition]/Signalled_event*

Build the equivalent set of state machines.



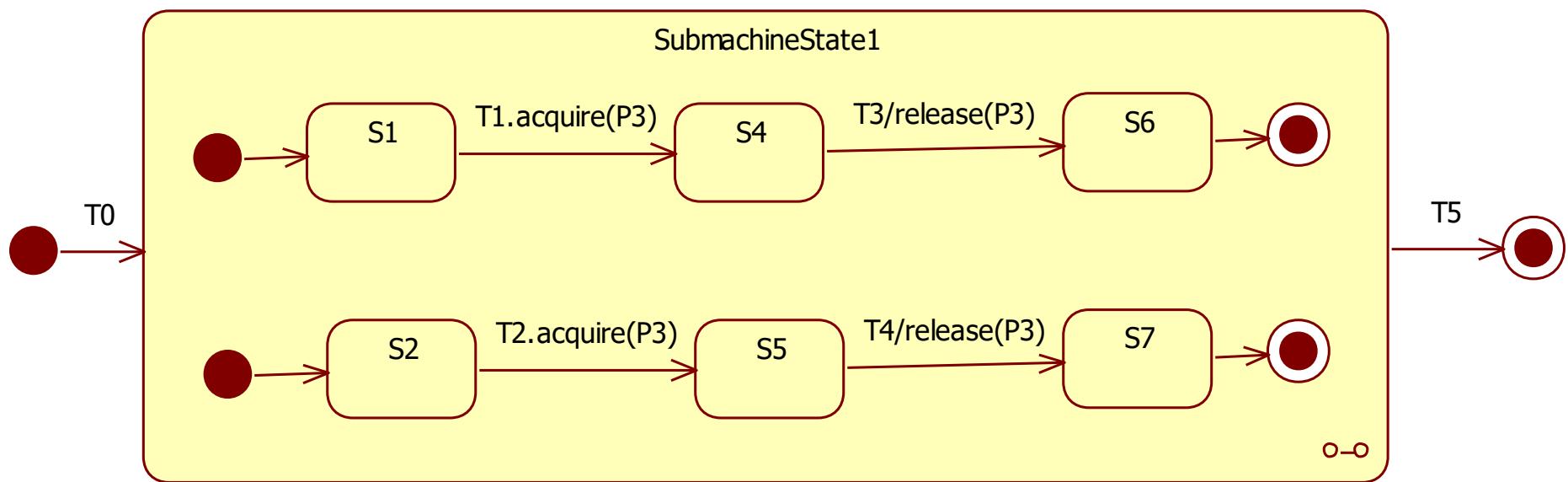
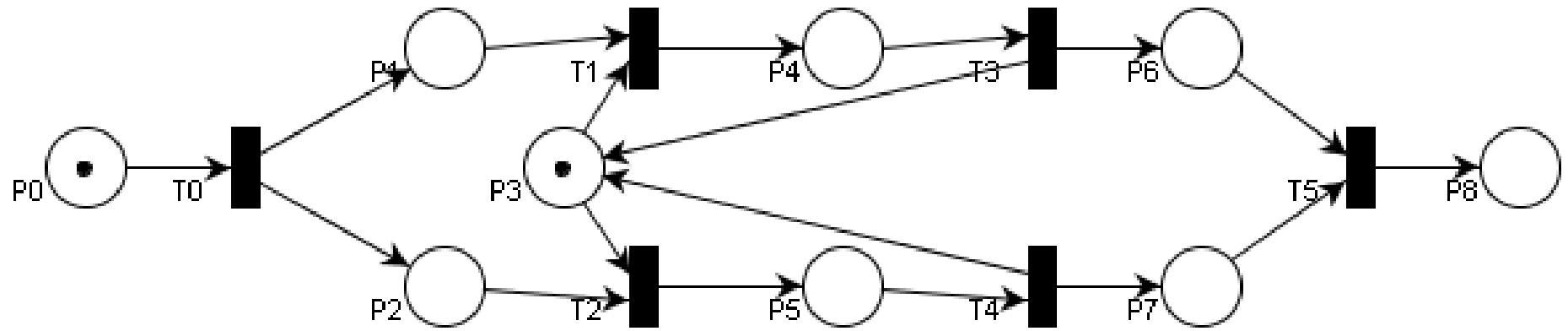
acquire(P3) must be atomic
release(P3) must be atomic

Semaphore
+permits: int
+acquire() +release()

$\sigma = ???$

<<add here the content>>

Mutual exclusion $\rightarrow P_3$
What parts of the program are mutual excluded?



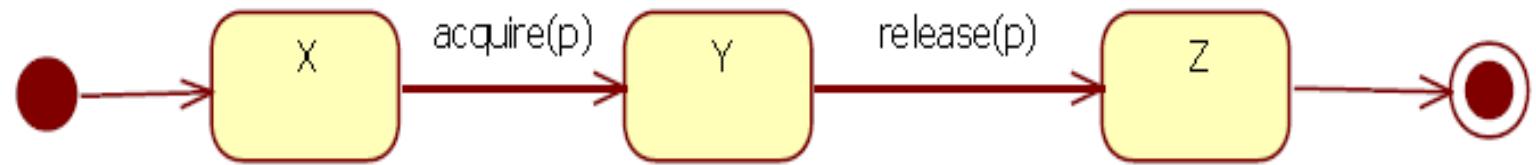


Fig. a. Simplified representation

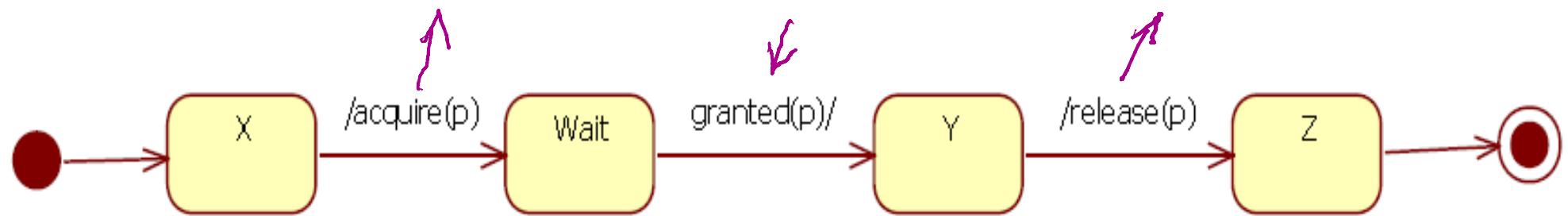
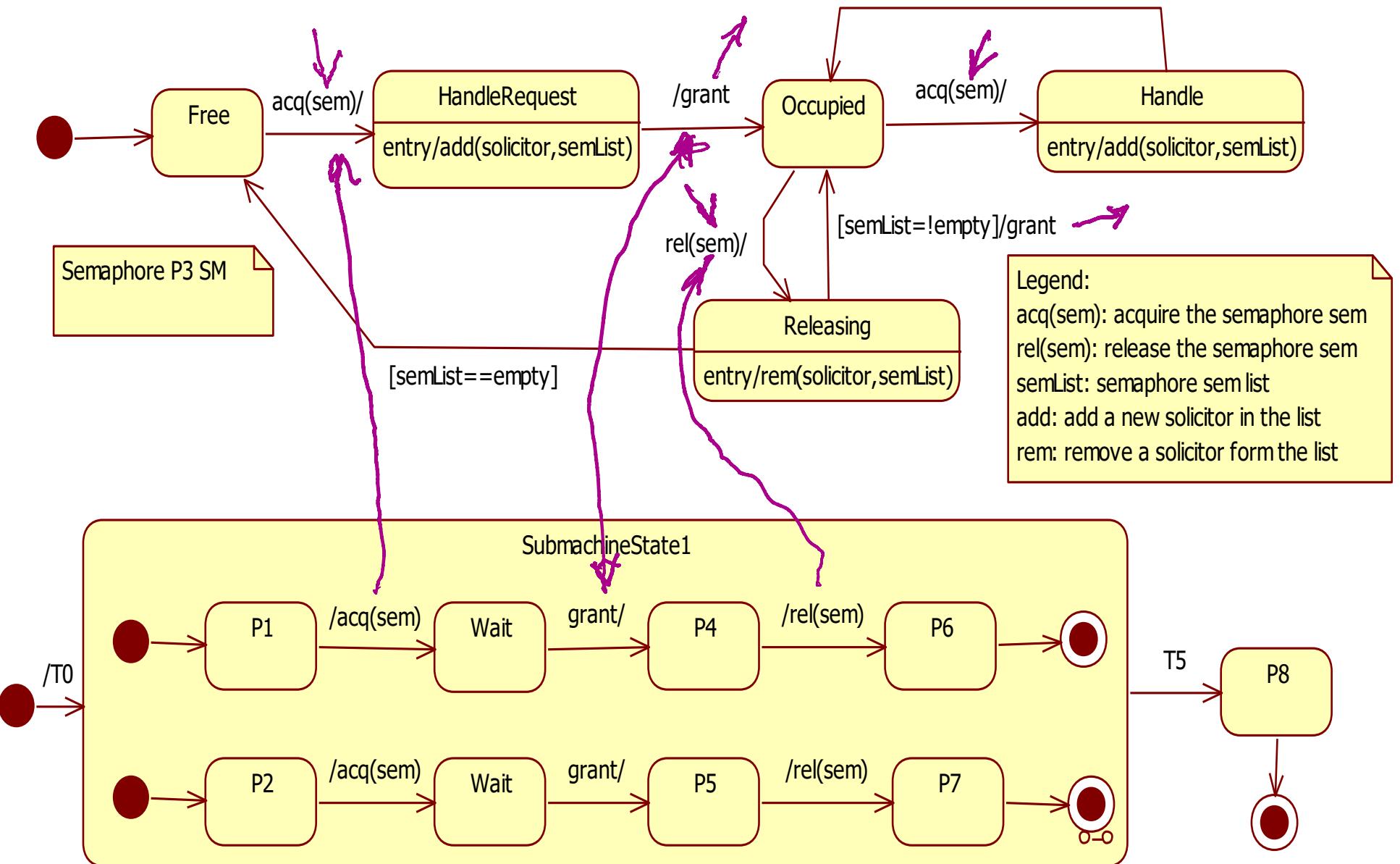
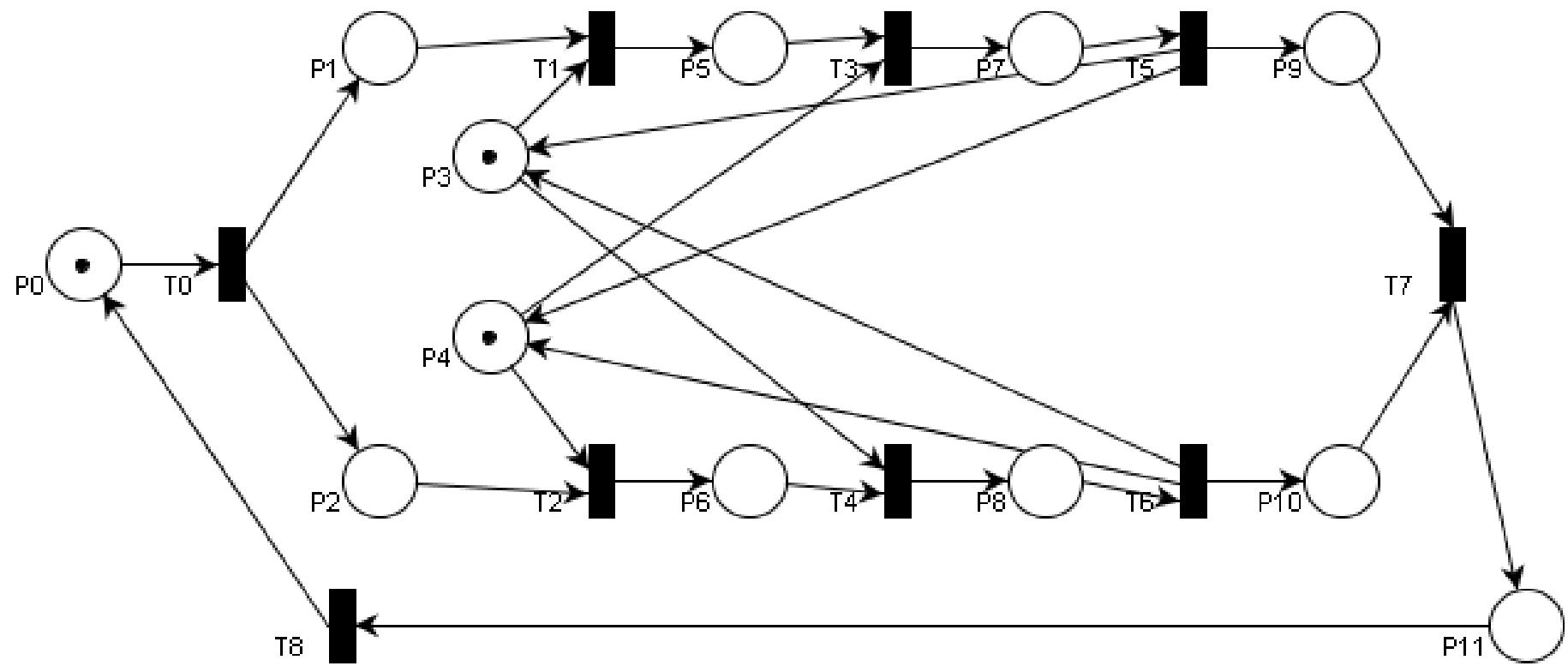


Fig. b. Detailed representation



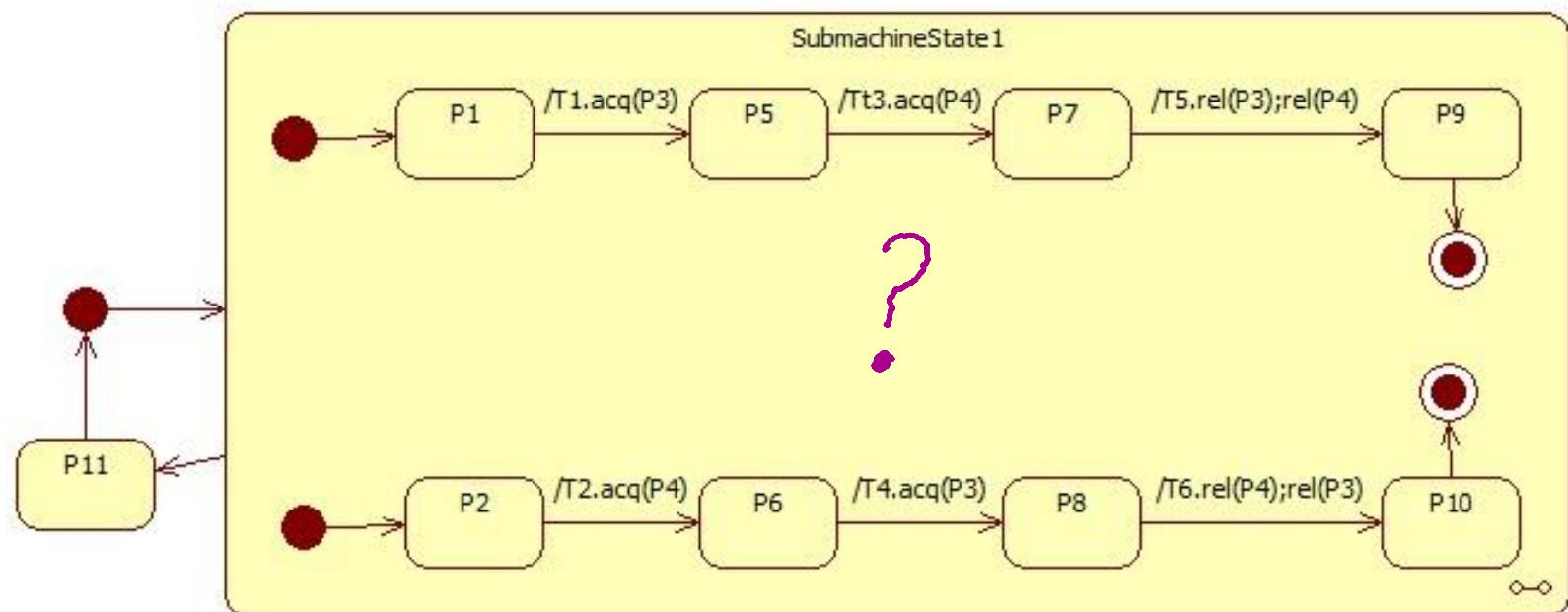
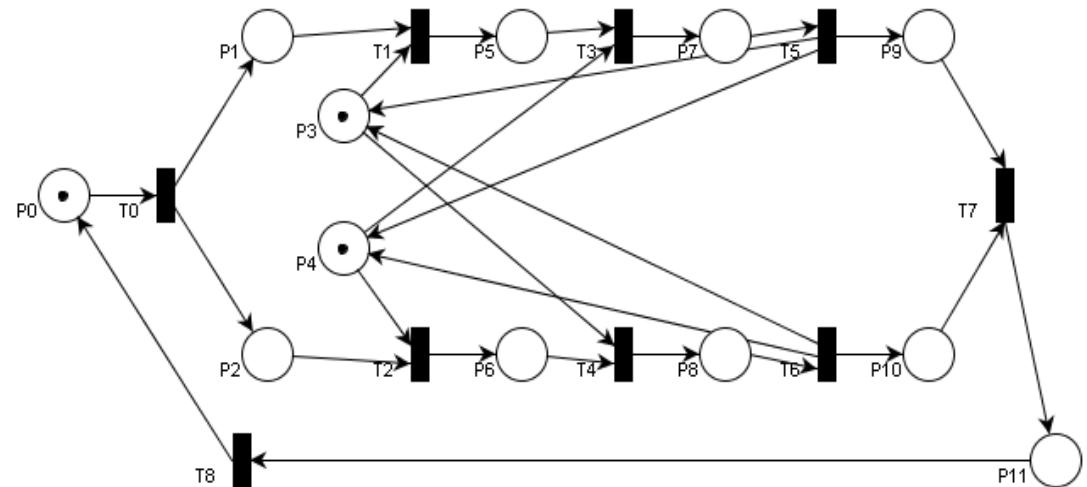


Draw the corresponding set of state machines.

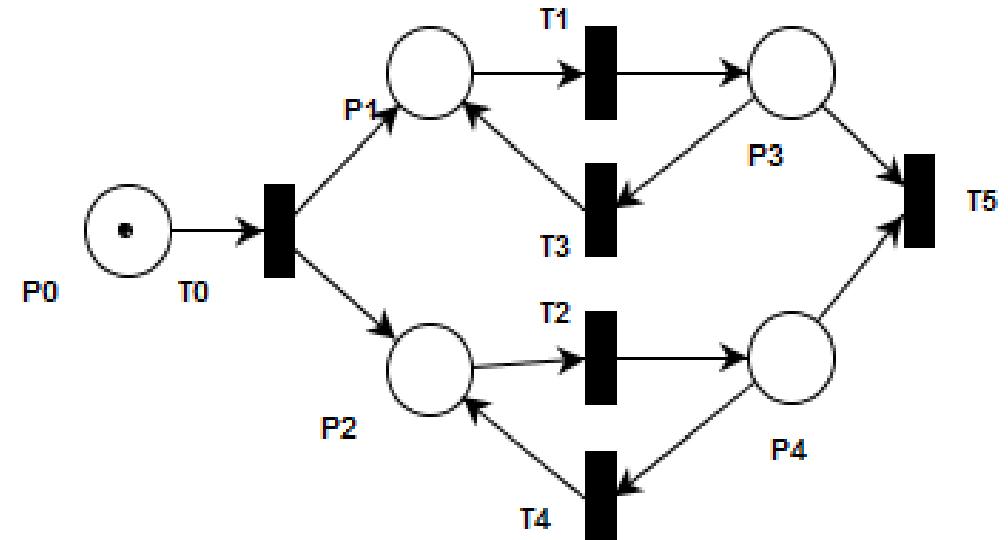
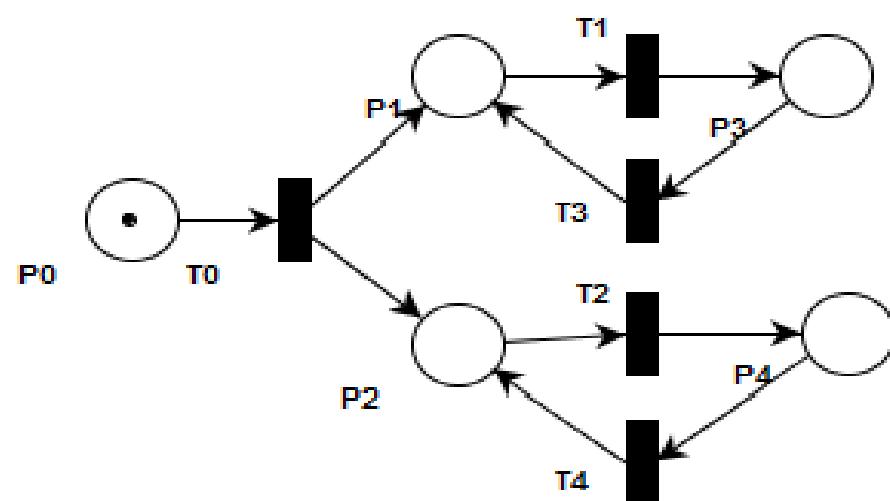
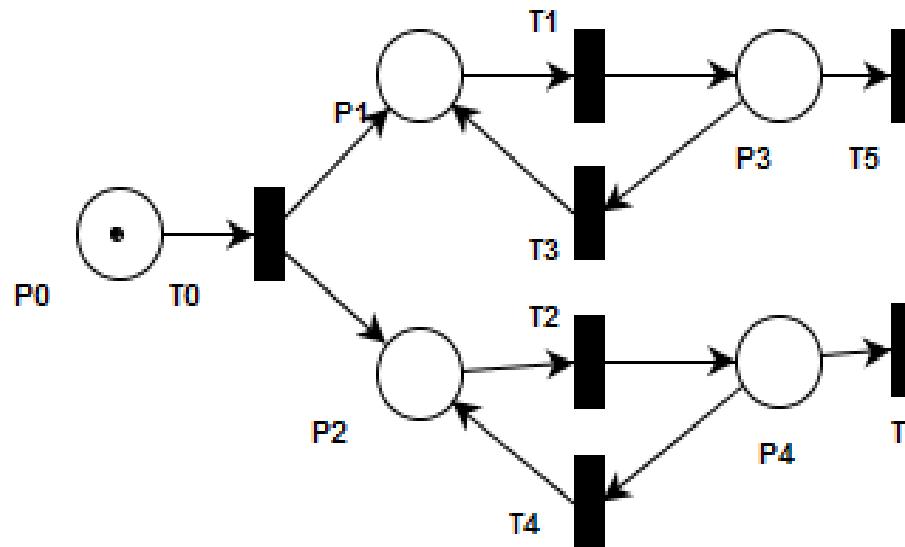
Building the equivalent set of state machines!

Can you see the potential deadlock in the state machine diagrams?

Can the potential deadlock easily be seen in the PN?

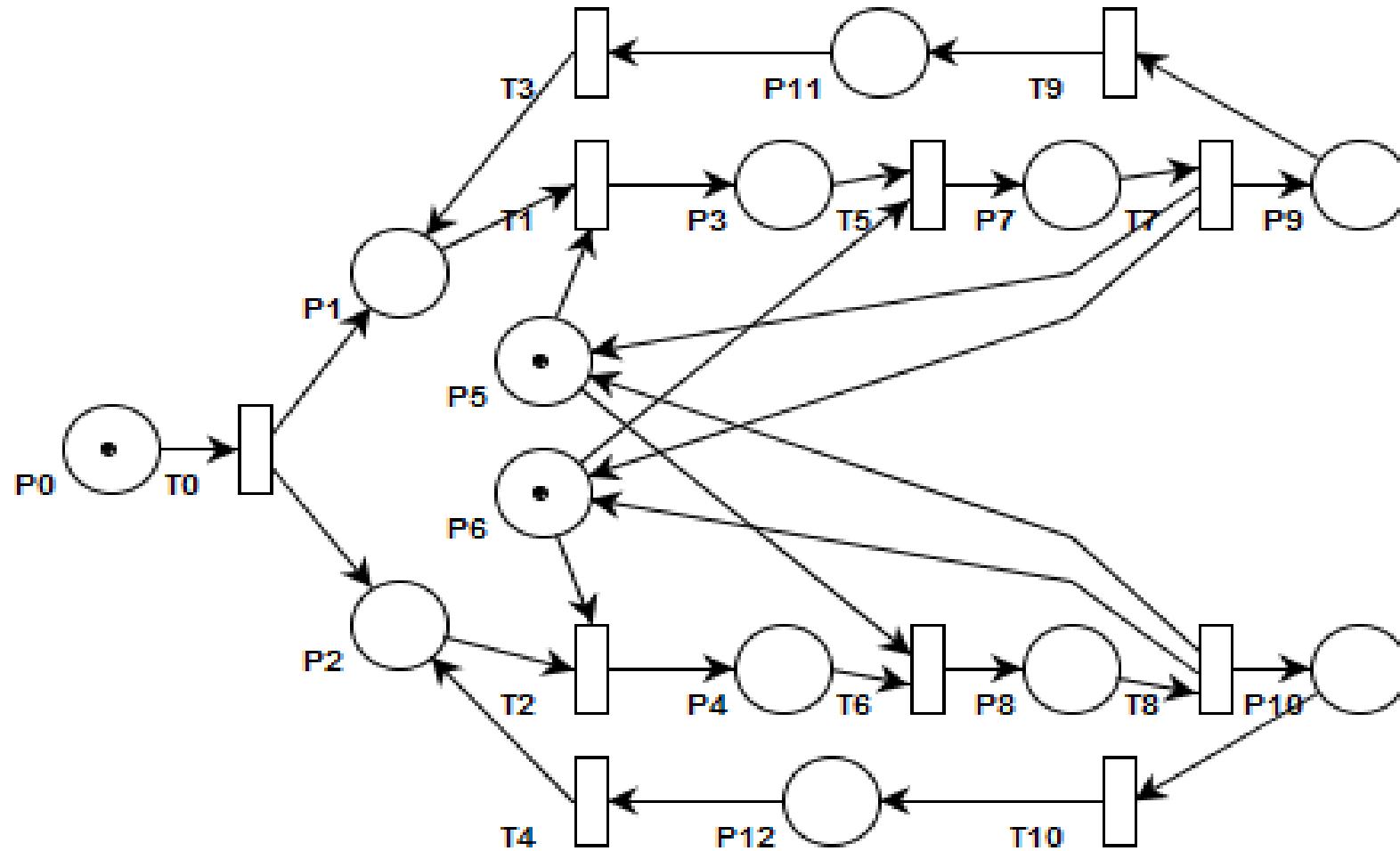


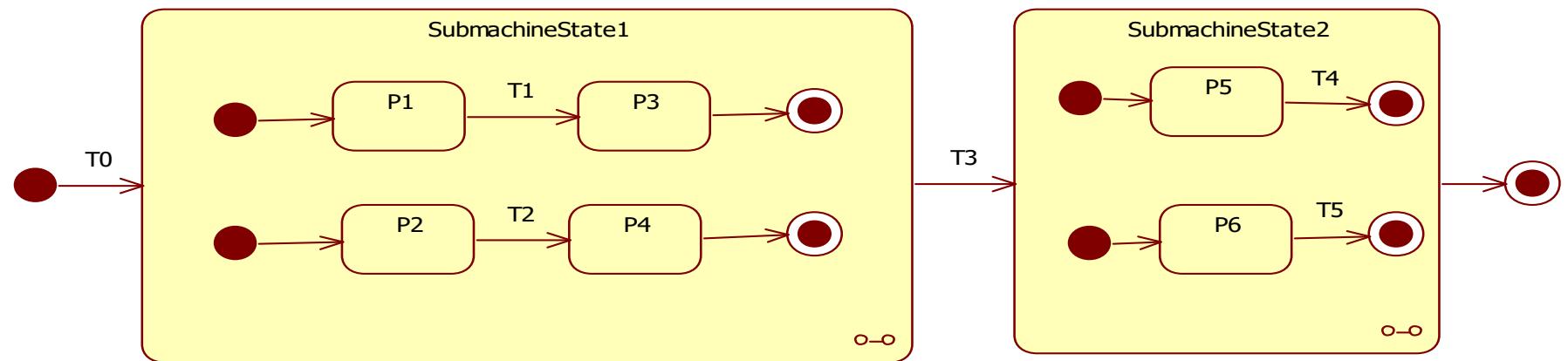
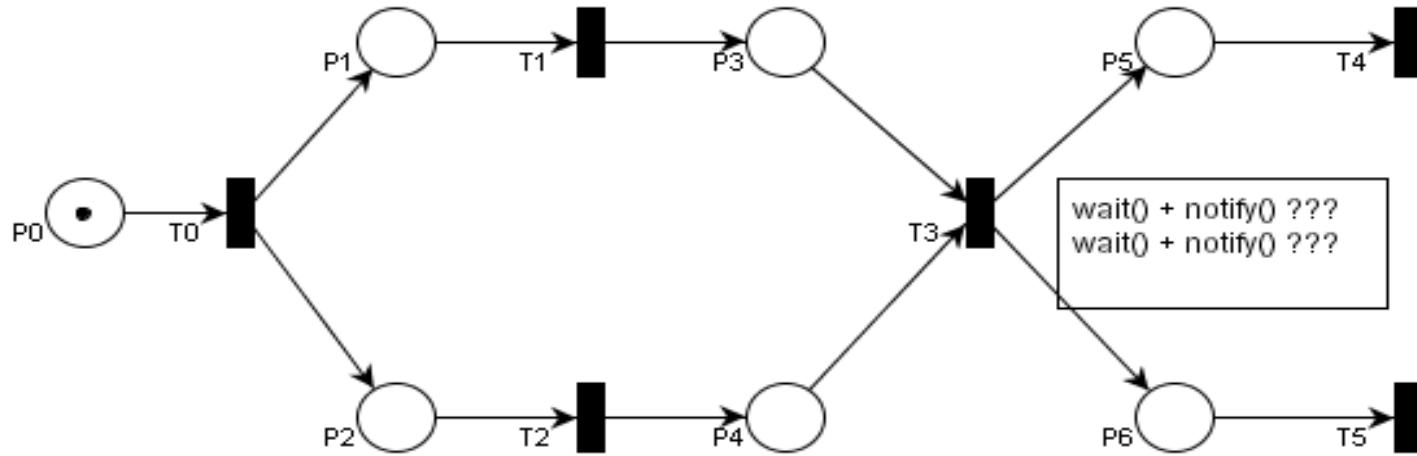
Build the equivalent set of the state machines and prove the equivalent behavior



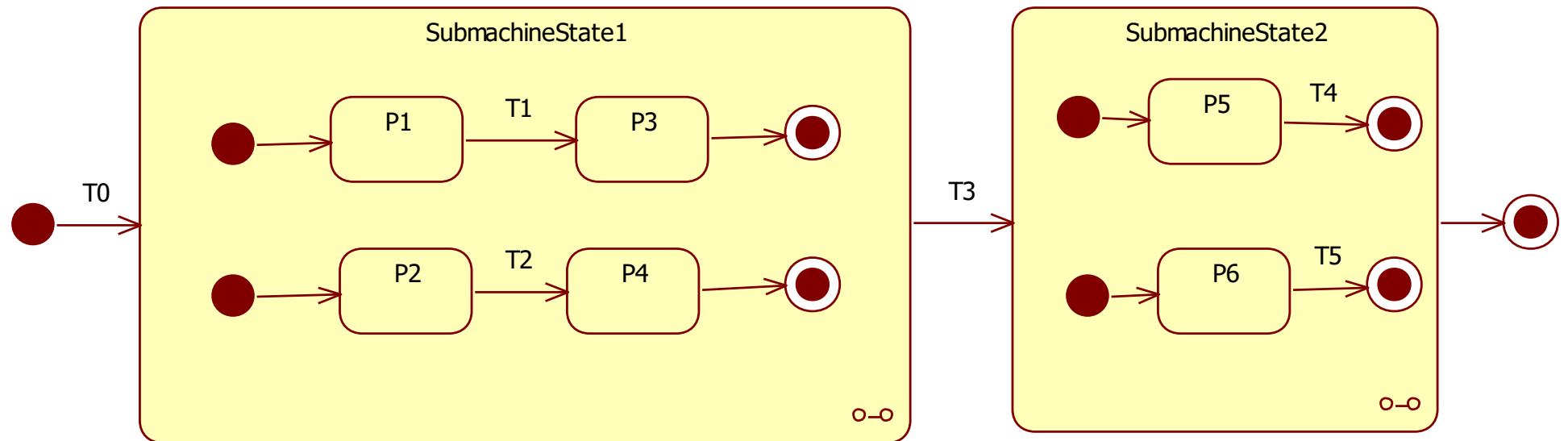
Home work:
 Conceive for the Petri nets
 the equivalent state
 machines.
 Verify that they can
 execute the same sequence
 of transitions.

Build the equivalent set of state machines.





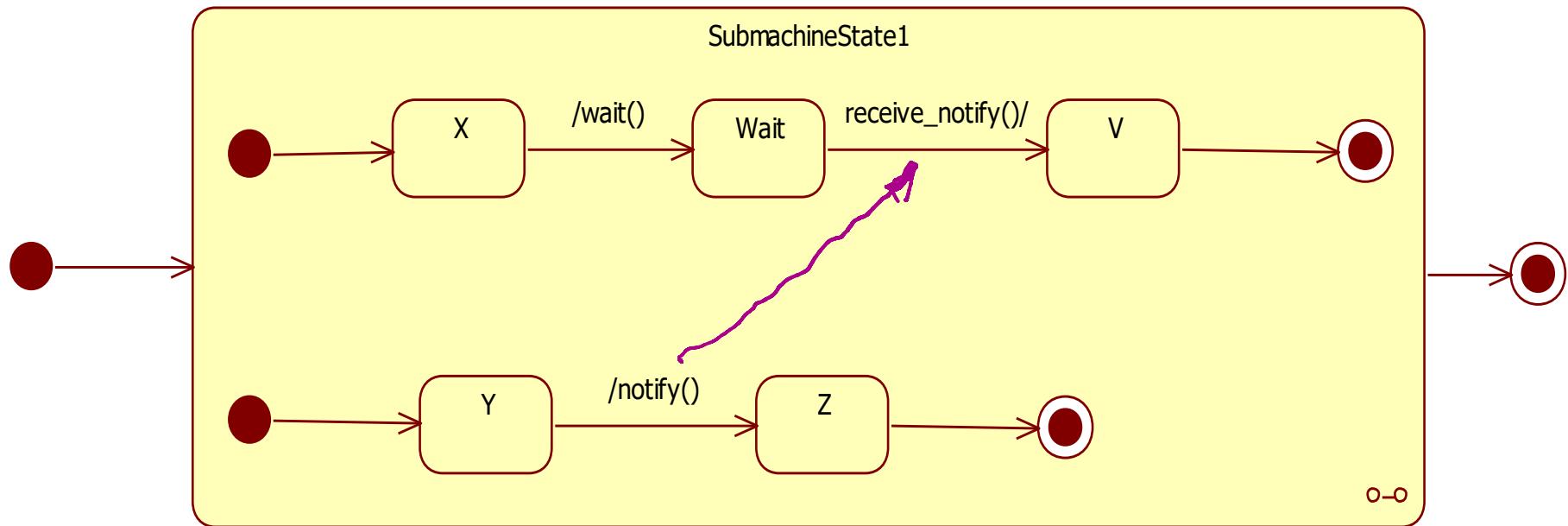
Double synchronization 1
 Prove the equivalence of the two models!
 Try the semaphore implementation. (!?)



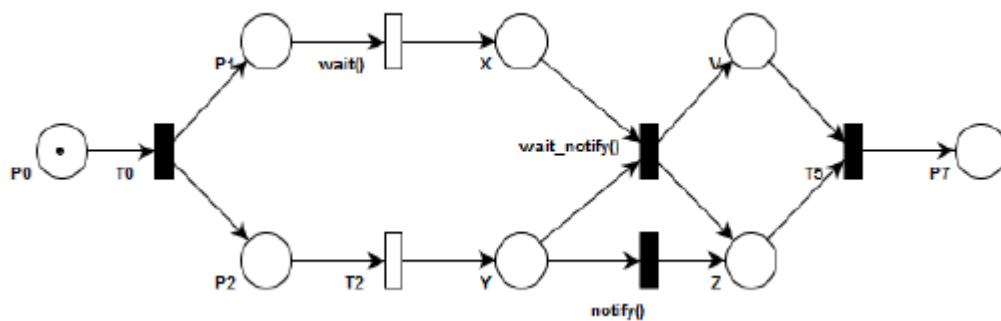
Implementation solutions:

- Barrier
- join()

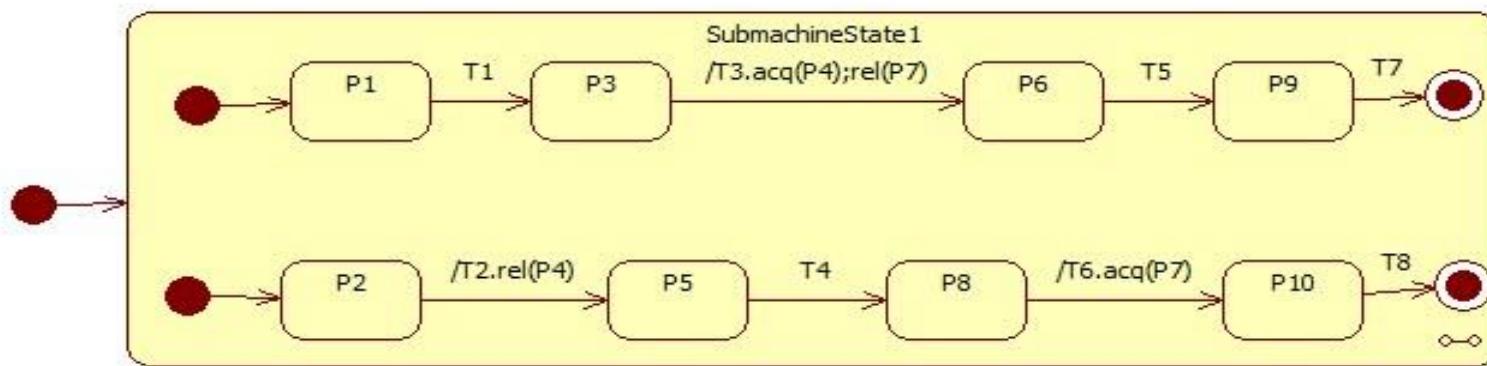
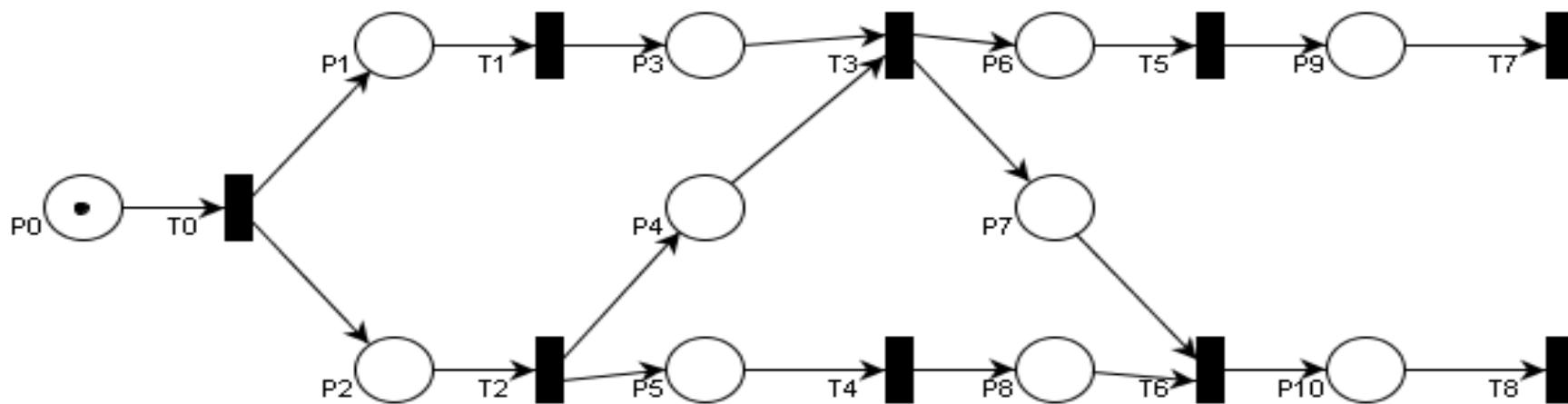
wait() – notify() model:



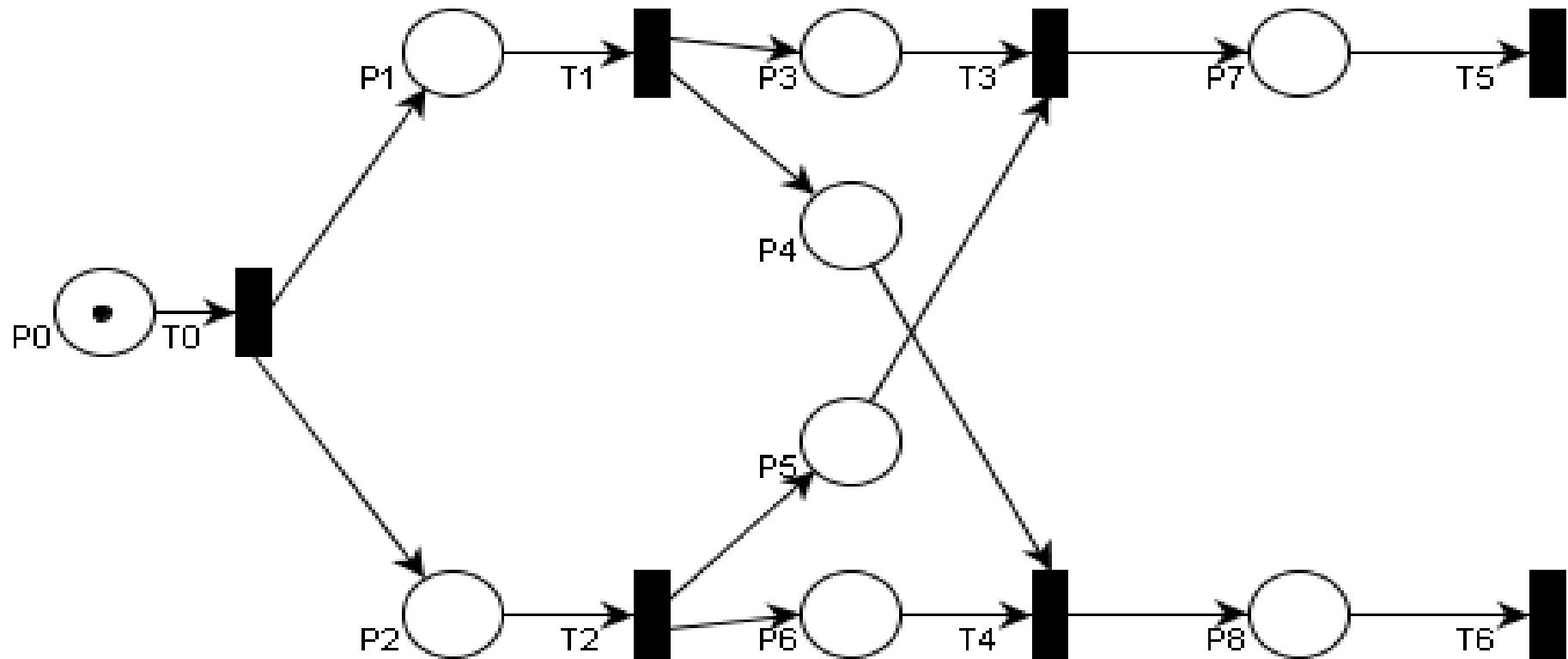
If T_2 and $notify()$ are executed before $wait()$, the end markings are $m(X)=1$, $m(Z)=1$.



Wait - notify synchronization
Prove the equivalence of the two models!



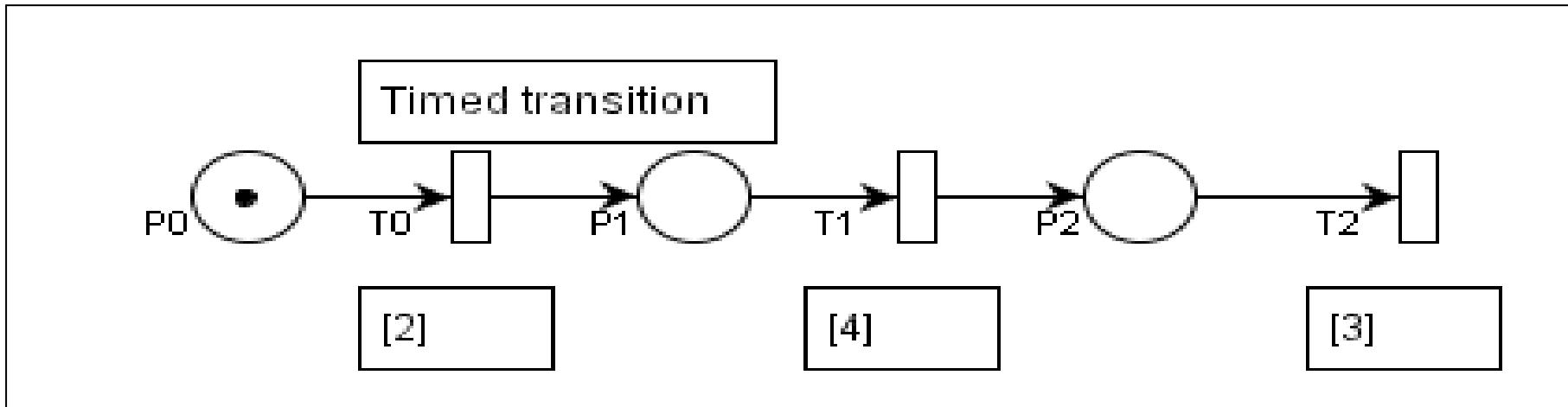
Double synchronization 2
 Prove the equivalence of the two models!
 Initialization: P4. permits(0); P7.permits(0)



Double synchronization 3

How can this be programmed using Java language?

Using Timed PN

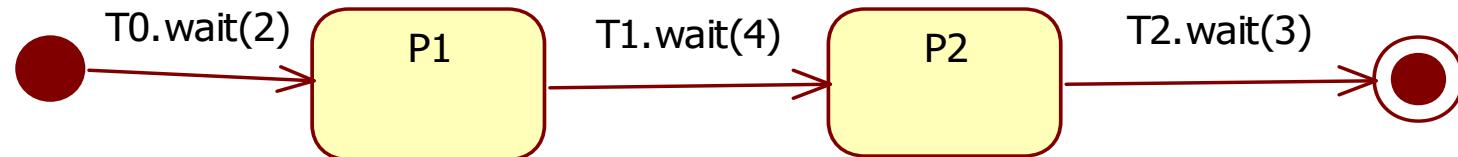


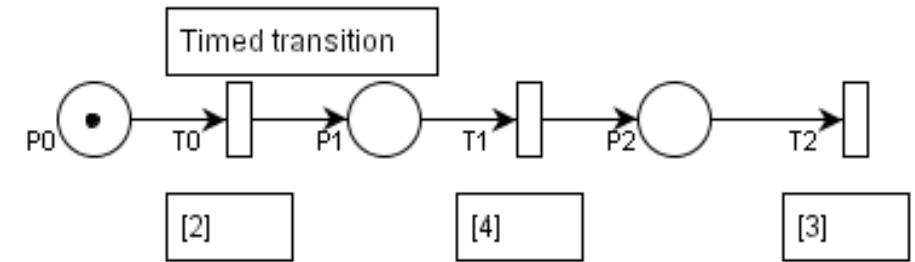
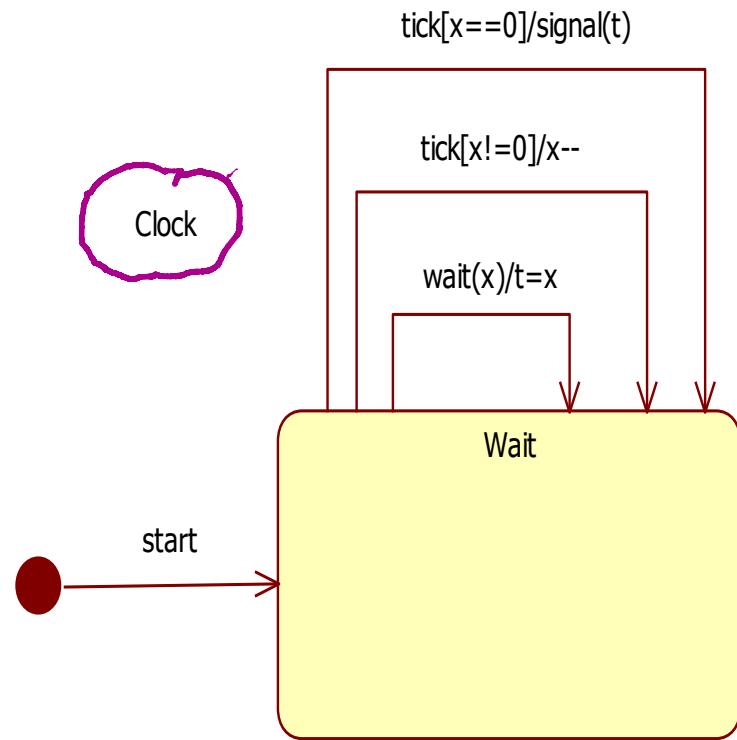
$$\sigma = T_0[2] \cdot T_1[4] \cdot T_2[3] = T_0(2) \cdot T_1(6) \cdot T_2(9)$$

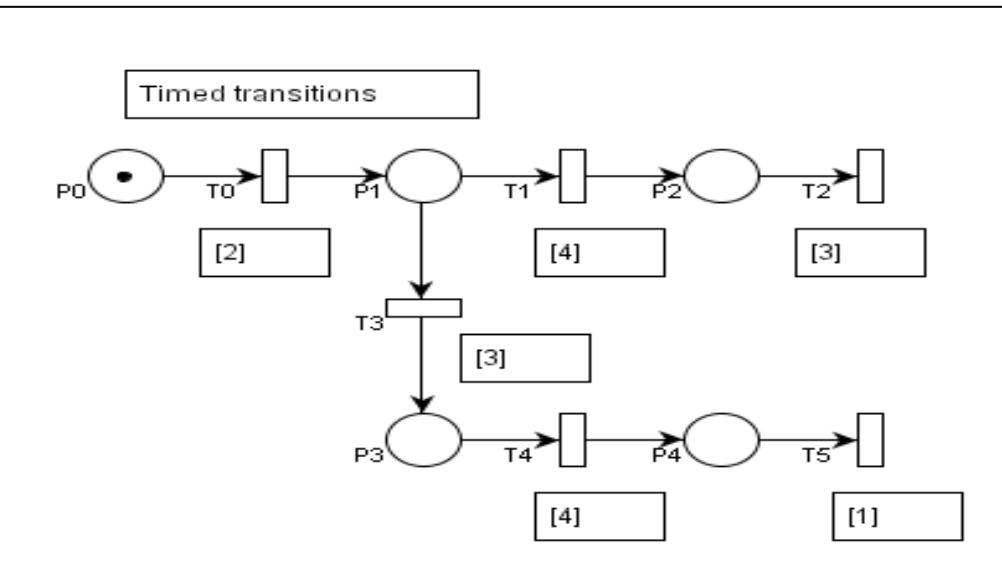
Prove the equivalence!

$T_0[2]$ – relative time

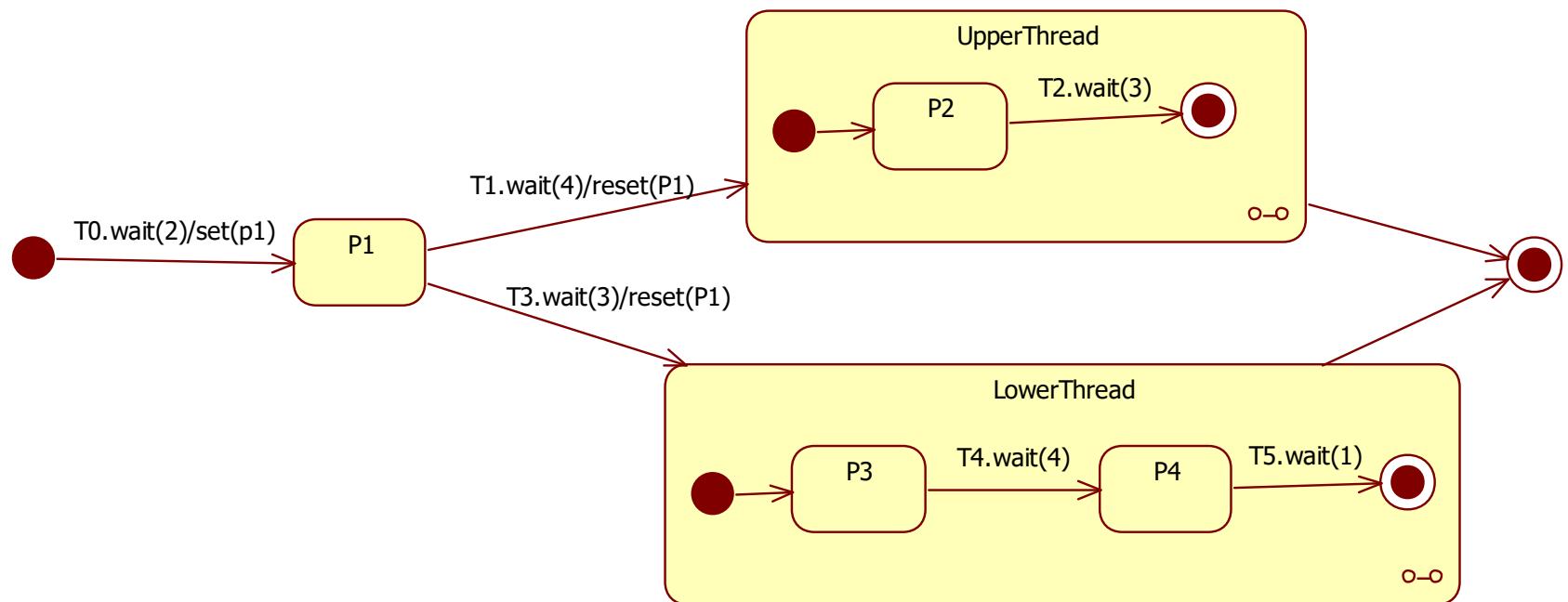
$T_0(2)$ – absolute time

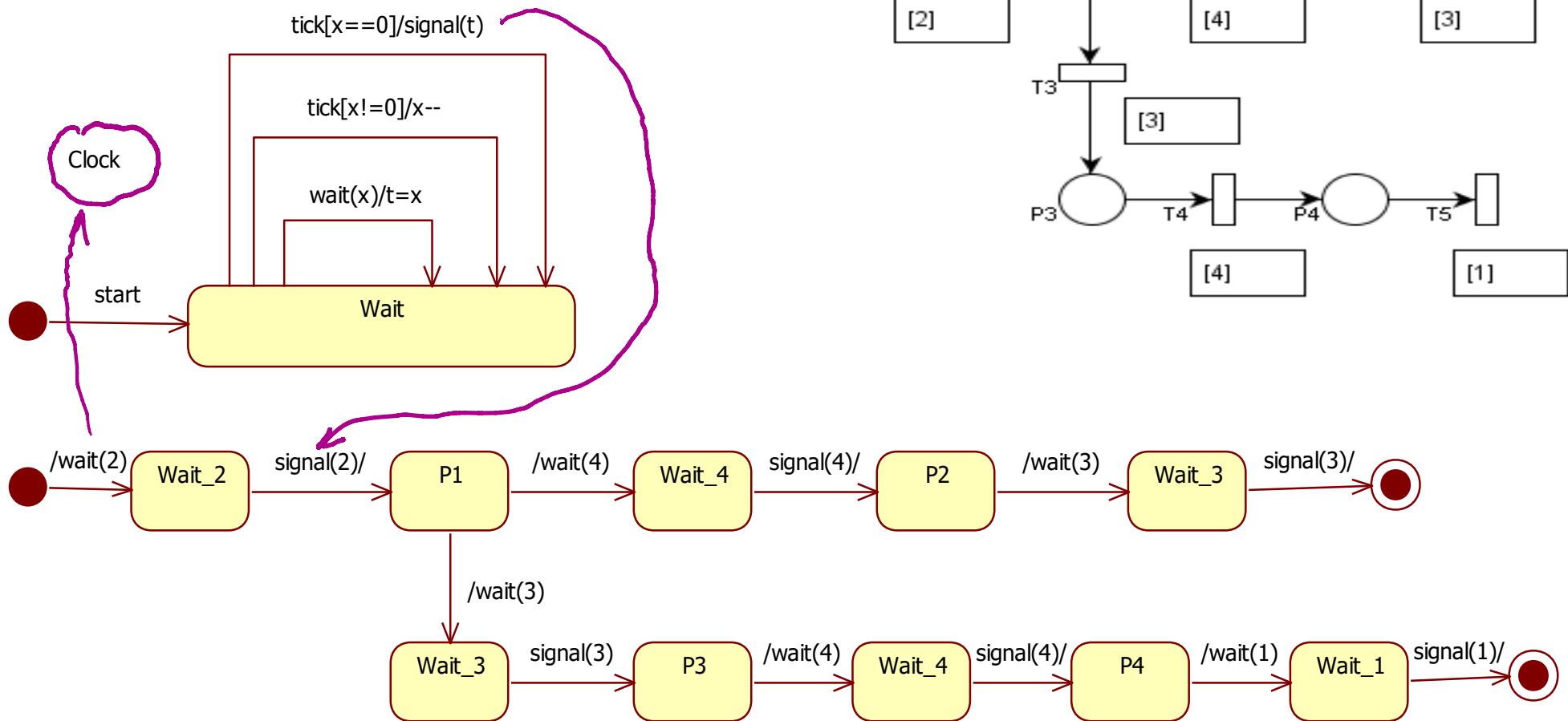




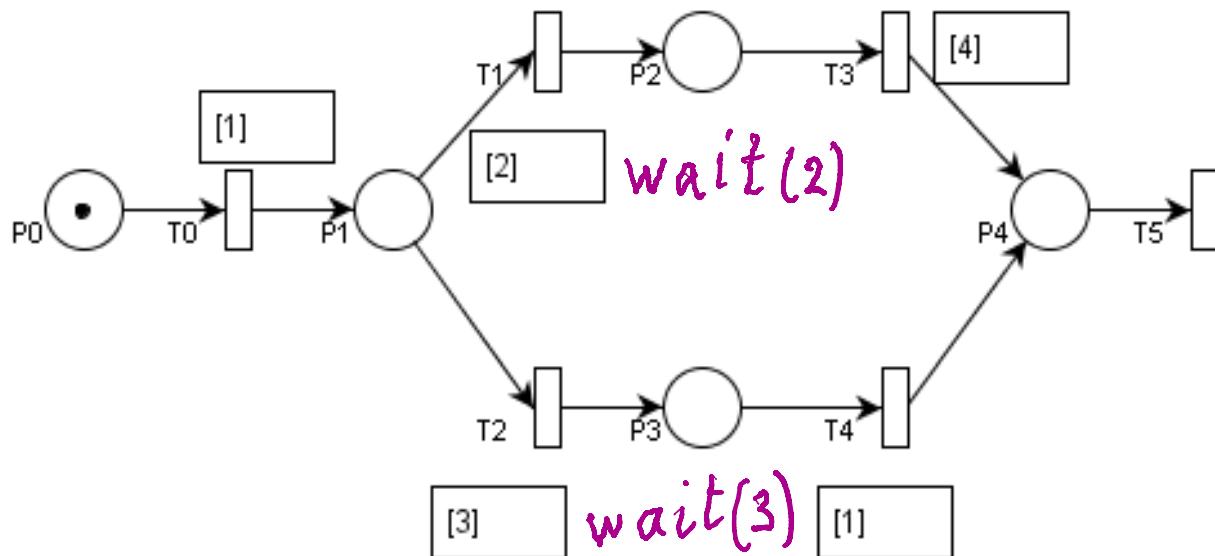


Prove the equivalence!





Alternative



Build the state machine model and prove the equivalence!

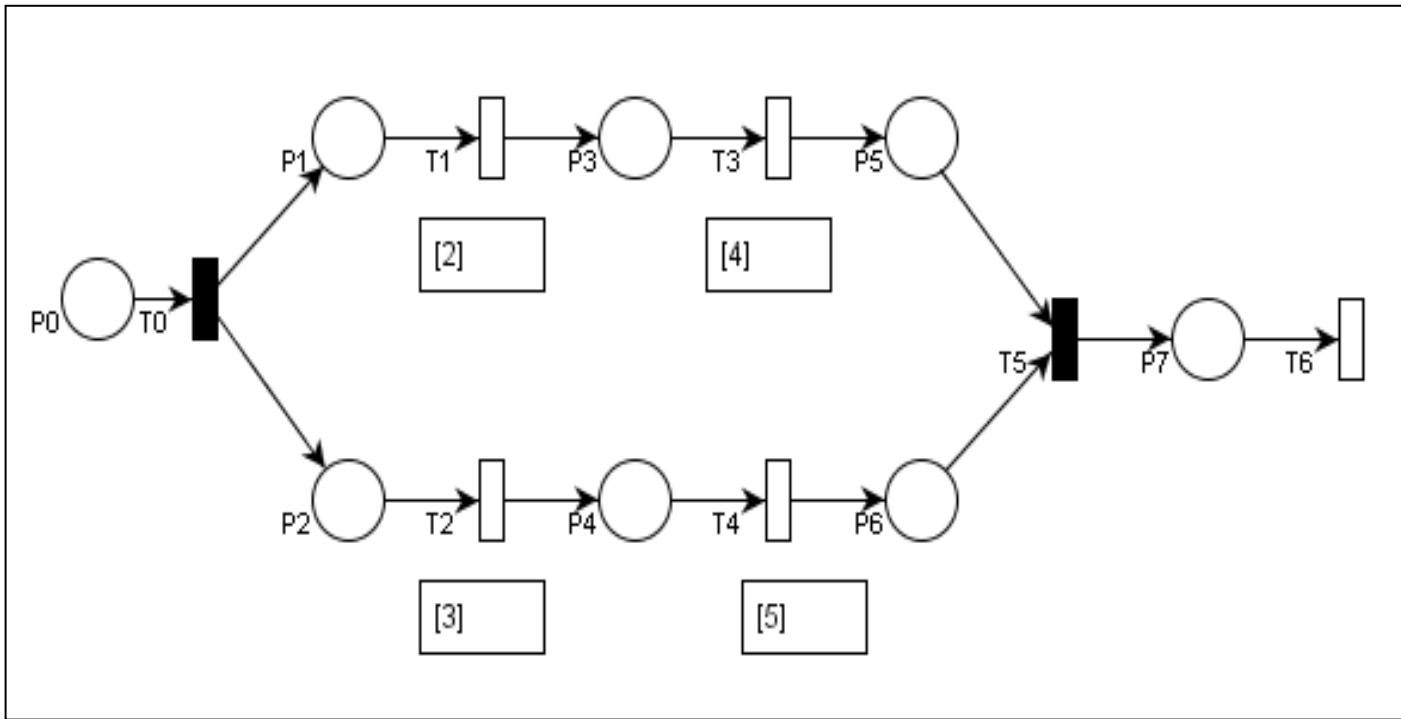
$$\sigma = T_0[1] \cdot (T_1[2] \cdot T_3[4] + T_2[3] \cdot T_4[1]) \cdot T_5[0] = \\ T_0(1) \cdot T_2(3) \cdot T_3(7) \cdot T_5(7_+) \rightarrow T_5(7_+)$$

Home work:

Conceive for the following Petri nets the equivalent state machines.

Verify that they can execute the same sequence of transitions.

Synchronous approach Concurrent execution

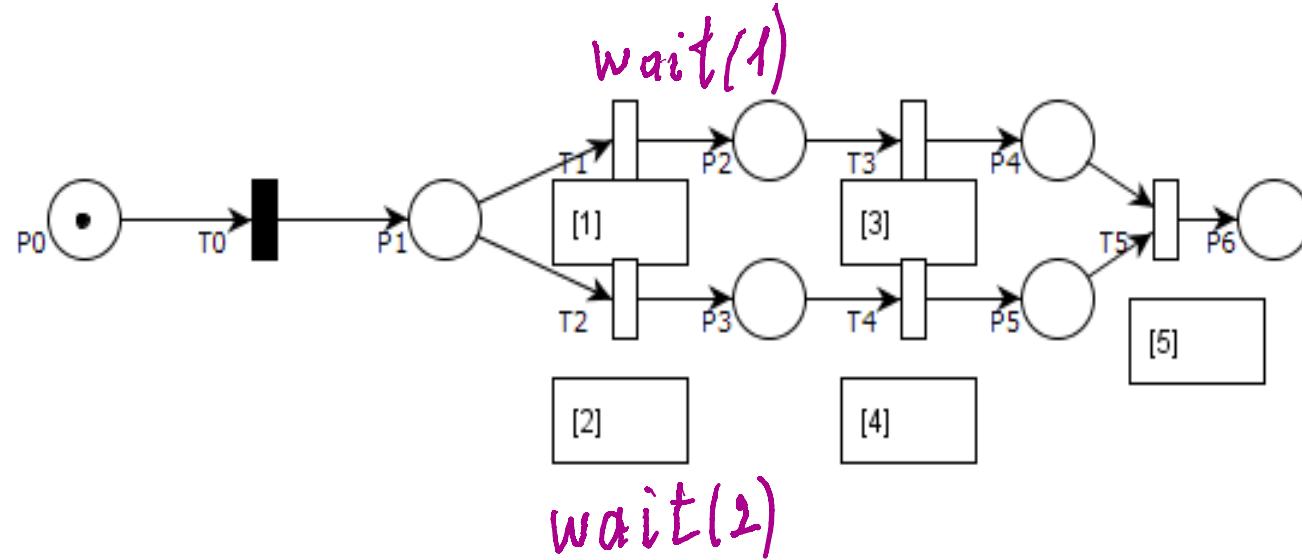


Build the state machine model and prove the equivalence!

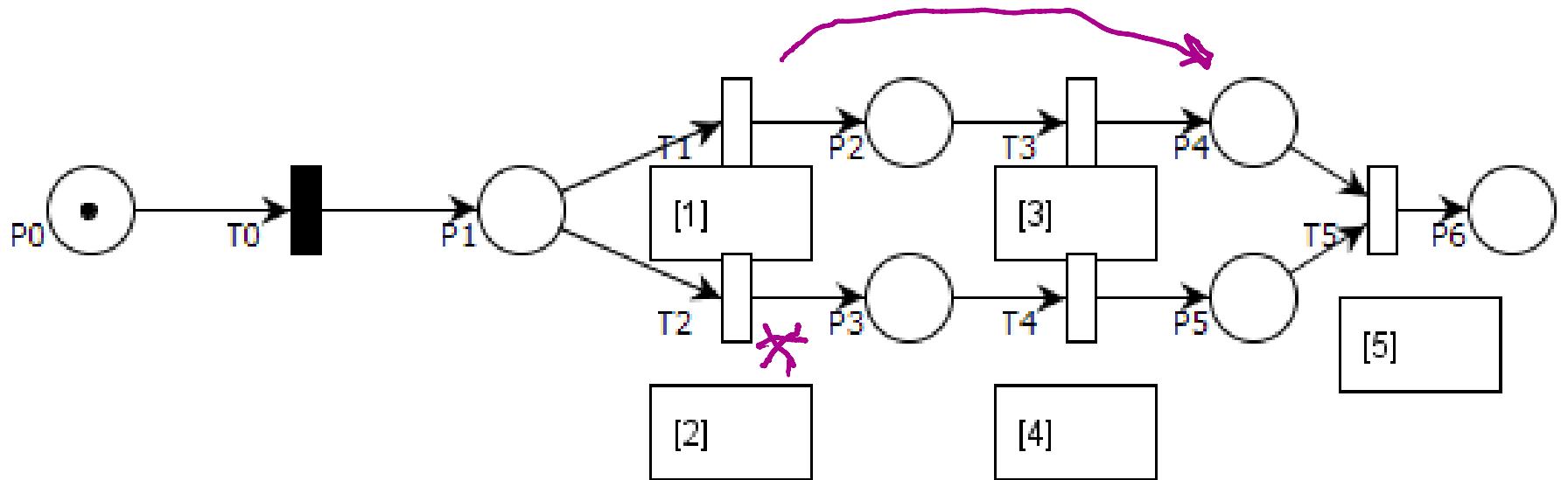
$$\begin{aligned}\sigma = & T_0[0] \cdot ((T_1[2] \cdot T_3[4]) \& (T_2[3] \cdot T_4[5])) \cdot T_5[0] \cdot T_6[0] = \\ & T_0(0) \cdot ((T_1(2) \cdot T_3(6)) \& (T_2(3) \cdot T_4(8))) \cdot T_5(8_+) \cdot T_6(8_{++}) \\ \rightarrow & T_8(8_{++})\end{aligned}$$

T_0 and T_5 were supposed to be executed immediately.

When is the transition T_5 executed?



Build the state machine model and prove the equivalence!



$$\sigma = T_0(0_+) \cdot ((T_1[1] \cdot T_3[3]) + (T_2[2] \cdot T_4[4])) \cdot 6$$

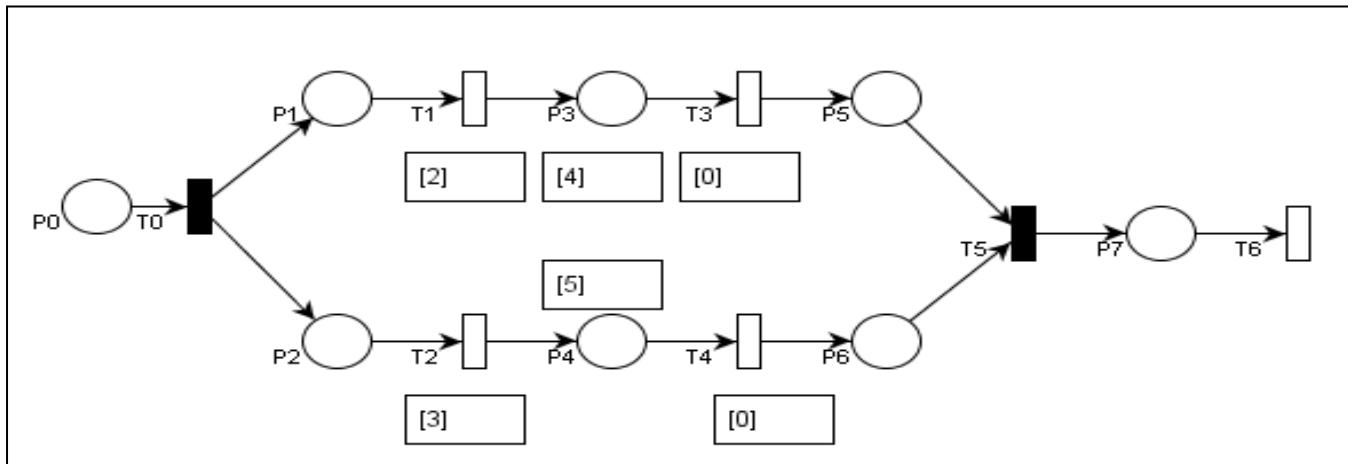
→ 6 is the blocking action

$$\sigma = T_0(0_+) \cdot T_1(1) \cdot T_3(4) \cdot 6(5_+)$$

The system analyzer detects that no more transitions can be executed (it is blocked) at $t=5_+$ t. u. and remains in this state forever.

Asynchronous approach

Concurrent execution on the same processor

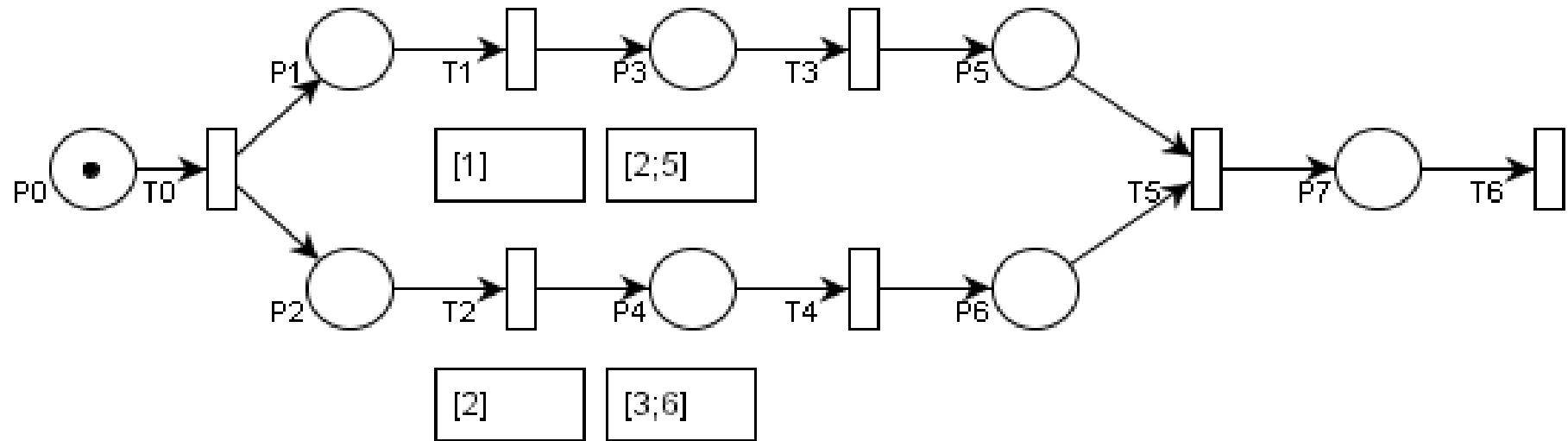


- Monoprocessor system
- two threads implementation
- timesharing
- equal priorities

$$\begin{aligned}\sigma = & T_0[0] \cdot (T_1[2] \cdot T_3[?] \& T_2[3] \cdot T_4[?]) \cdot T_5[0] \cdot T_6[0] = \\ & T_0(0) \cdot (T_1(2) \cdot T_3(9) \& T_2(3) \cdot T_4(11)) \cdot T_5(11+) \cdot T_6(11++) \\ \rightarrow & T_6(11++)\end{aligned}$$

State machines?

Asynchronous Approach with Variable Durations



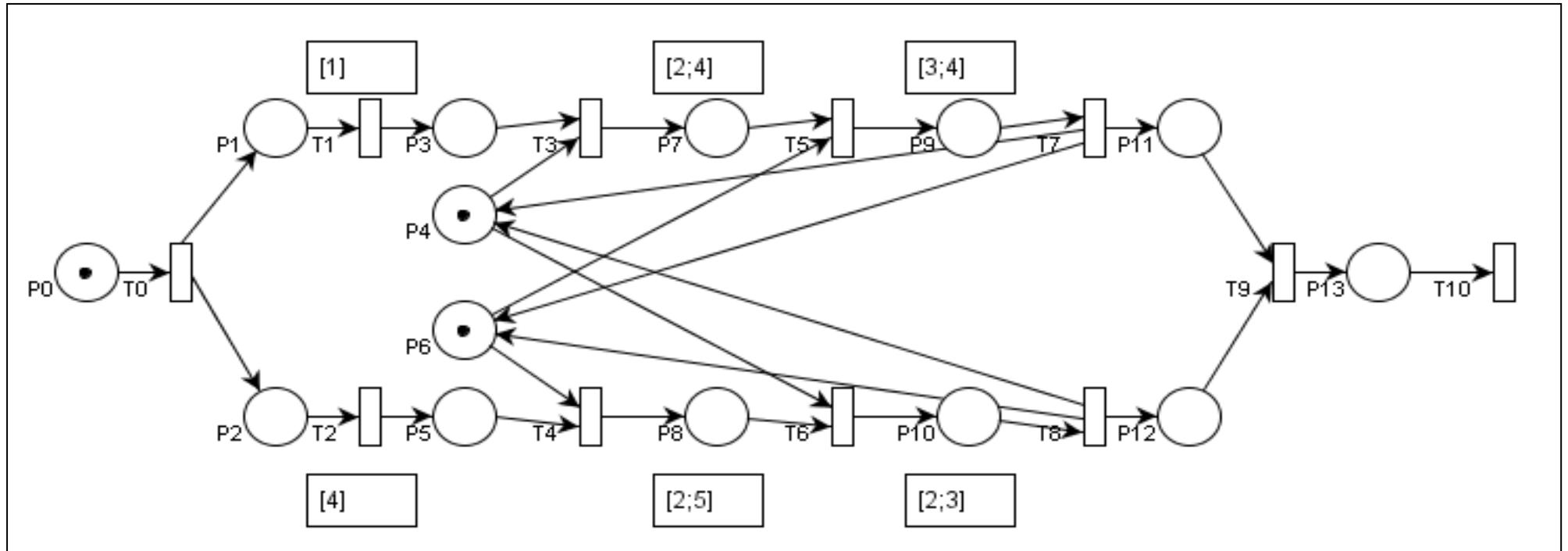
$$\begin{aligned}
 \sigma = & T_0[0] \cdot ((T_1[1] \cdot T_3[?;?]) \& (T_2[2] \cdot T_4[?;?])) \cdot T_5[0] \cdot T_6[0] = \\
 & T_0(0) \cdot (T_1(2) \cdot T_3(4;10) \& T_2(3) \cdot T_4(6;12)) \cdot T_5[0] \cdot T_6[0] = \\
 = & T_0(0) \cdot T_5(6_+;12_+) \cdot T_6[0] \\
 \Rightarrow & T_6(6_{++}; 12_{++})
 \end{aligned}$$

Build the state machine model
and prove the equivalence!

StateName

entry/EntryAction1
do/DoAction1
exit/ExitAction1

Asynchronous approach - Deadlock with variable timing



$$\sigma_1 = T_0[0] \cdot (T_1(1) \cdot T_3(1_+) \cdot T_5(3) \cdot T_7(6_+) \& T_2(6_+) \cdot T_4(6_{++}) \cdot T_6(8) \cdot T_8(10_+)) \cdot T_9[0] \cdot T_{11}[0]$$

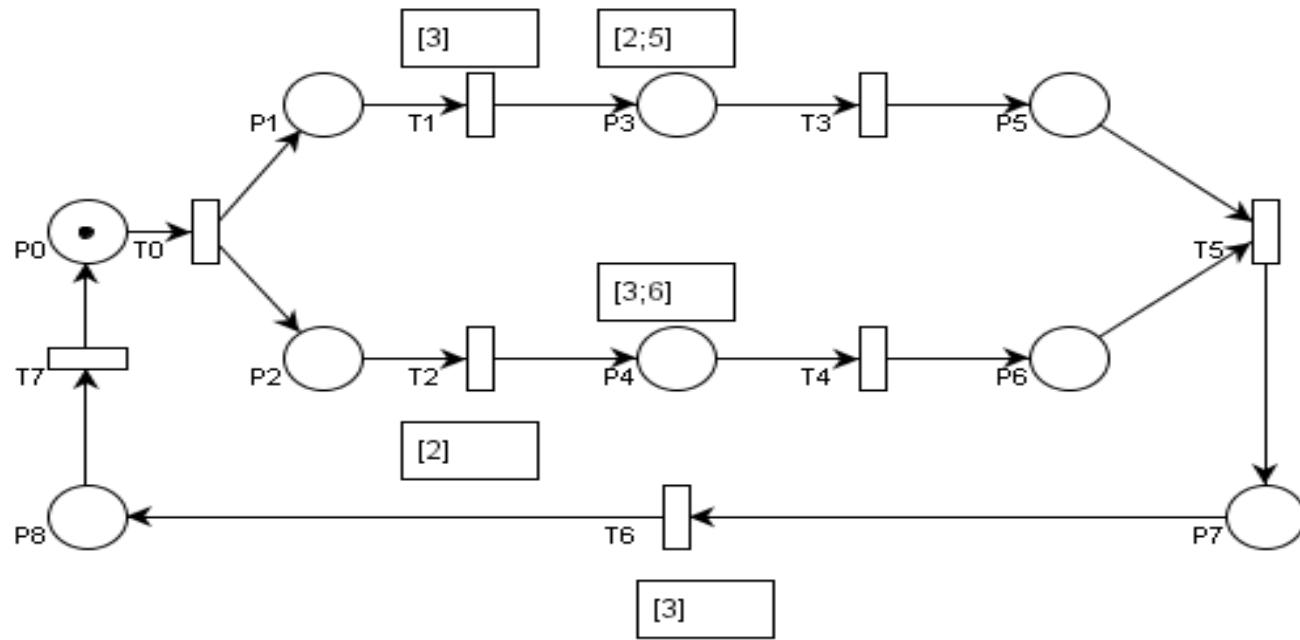
→ NO Deadlock

$$\sigma_2 = T_0[0] \cdot ((T_1(1) \cdot T_3(1_+) \cdot \textcolor{red}{T_5(5?)}) \cdot \delta) \& (T_2(4) \cdot T_4(4_+) \cdot \delta) \cdot T_9[0] \cdot T_{11}[0]$$

→ Deadlock

Conclusion: Sometimes the program is deadlocked and sometimes is not.

Constraints specification and verification



- Monoprocessor system
- two threads implementations
- timesharing
- equal priorities
- variable durations

Period timing constraint is 12 t.u.
Can it be met?

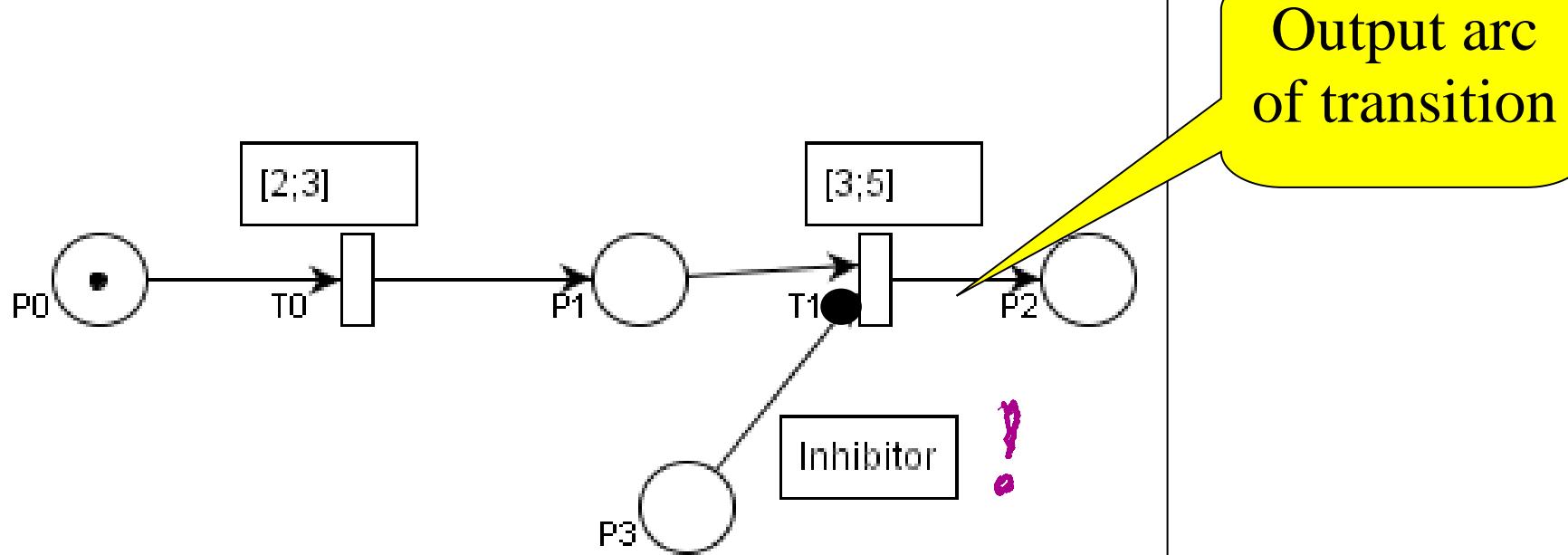
$$\sigma_1 = T_0[0_+] \cdot (T_1(3) \cdot T_3(7;13) \& T_2(2) \cdot T_4(7;13)) \cdot T_5(7_+;13_+) \cdot T_6[3] \cdot T_7[0] \cdot \\ \cdot T_0(10_{++};16_{++}) \cdot \dots$$

Conclusion: Sometimes YES and sometimes NO → Answer: NO for real-time systems.

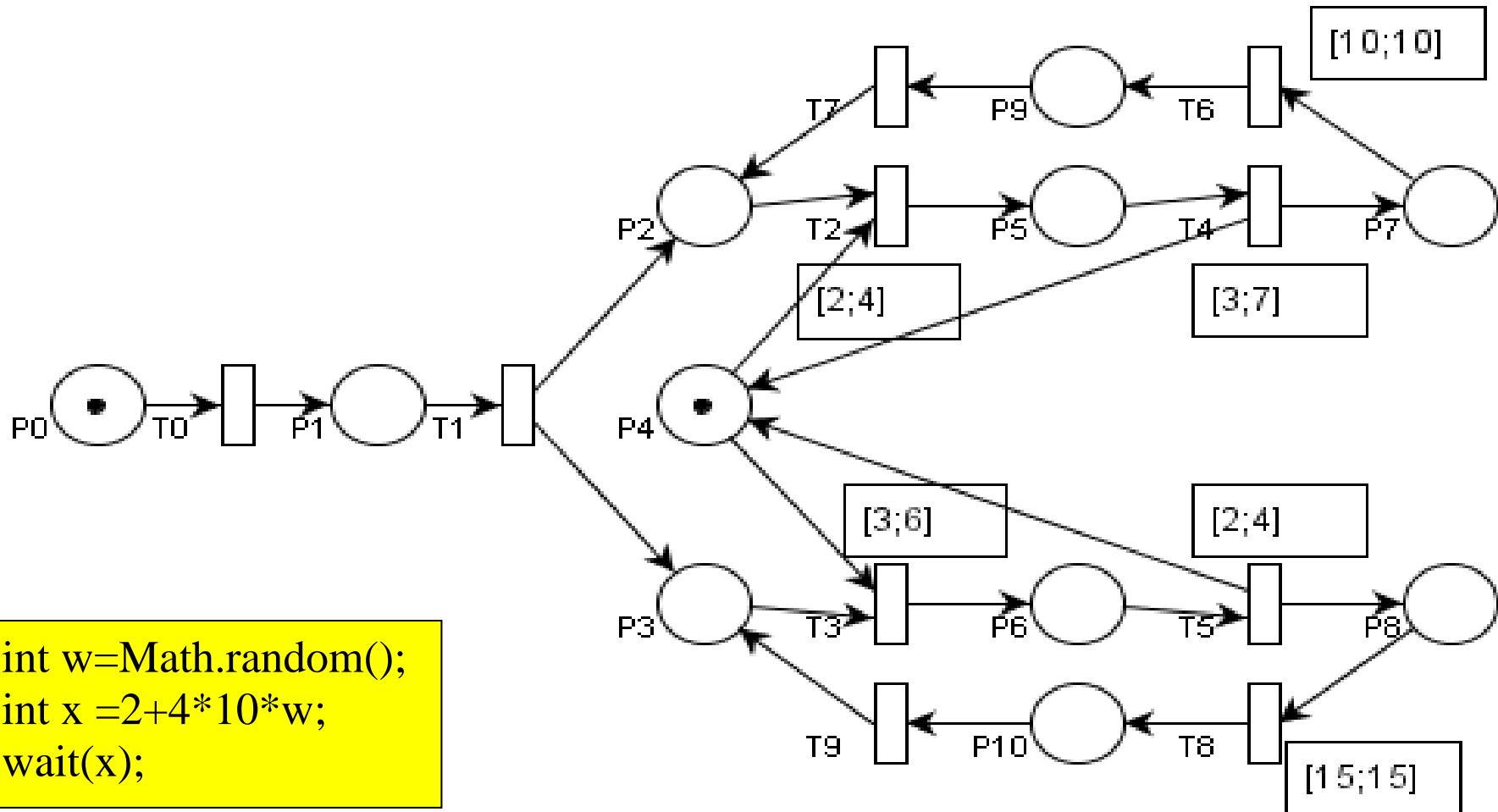
IF periodConstraint=17 → YES

Homework: Get an implementation such that the period is always 18 t.u.

How can this be programmed using Java language?



Build the state machine model and prove the equivalence!



Build the state machine model and prove the equivalence!

Semaphore

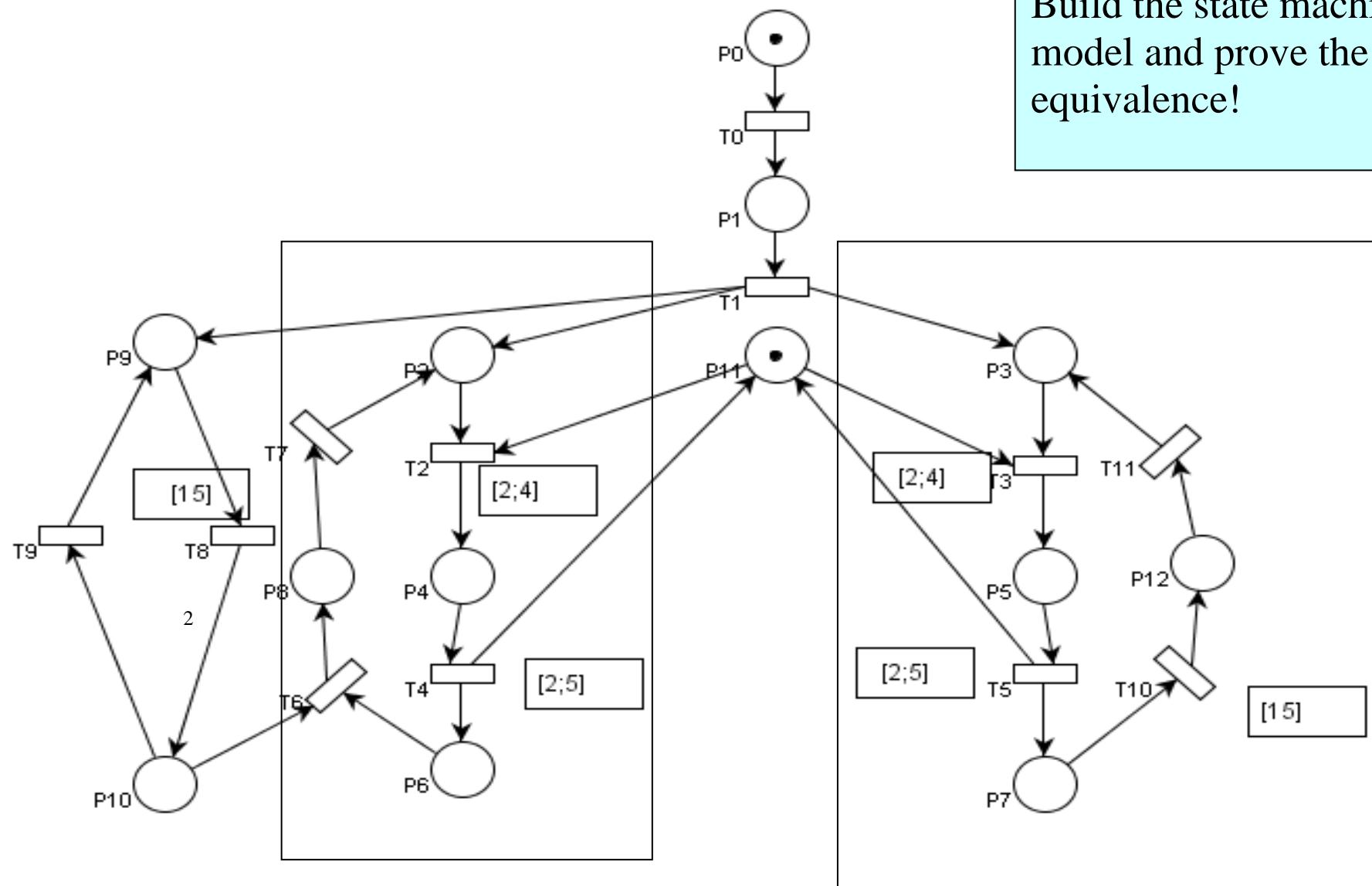
+permits: int
+acquire()
+release()

When a thread starts its execution it blocks the starting of another thread execution.
How can the thread periods be calculated? Are they fixed?

How are the threads' periods from the next PN? Can they be calculated?

Homework: Construct a TPN with the fixed periods of the corresponding threads.

Build the state machine
model and prove the
equivalence!



*

END

*

Ch. 3. Design of R-T Applications

3.1. Relations of UML diagrams with different kinds of PNs

3.2. Transformation of Petri Nets into State Machines

3.3. Extending UML

3.4. Using UML Real-Time

3.5. Task structure diagram

New:

- timings
- UML-RT Capsule diagram

You have studied (Software Engineering):

1. Goals and Scope
2. Principles of Visual Modeling
3. Outline of UML and Unified Process
4. Specification – Requirements
5. Use-Case Models
 - a. Use-Case Diagram
 - b. Activity Diagram
6. Design Models
 - a. Interaction Diagrams
 - i. Sequence Diagram
 - ii. Communication Diagram
 - iii. Timing Diagram
 - iv. Interaction Overview Diagram
 - b. State Machine Diagram
 - c. Class Diagram
 - d. Task Structure on Object Communication Diagrams
7. Implementation Diagrams
 - a. Component Diagram
 - b. Composite Structure Diagram
8. Deployment Models
 - a. Deployment Diagram

3.3. Extending UML

Current goals:

- adapt and extend the UML diagrams to catch the real-time features.
- adapt the UML diagrams for real-time reactive (control) applications.

What are the deadlines? → Nonfunctional parameters

Where are the deadlines implemented (programmed) in the standard Java source code? ← They are not implemented. They are used for verification. The deadlines can be implemented in Realtime Java but not in standard Java.

Where are others temporal parameters implemented?

Introduction

Where can UML be used?

- specification
- design
- implementation
- verification (*partially*)
- testing
- maintenance

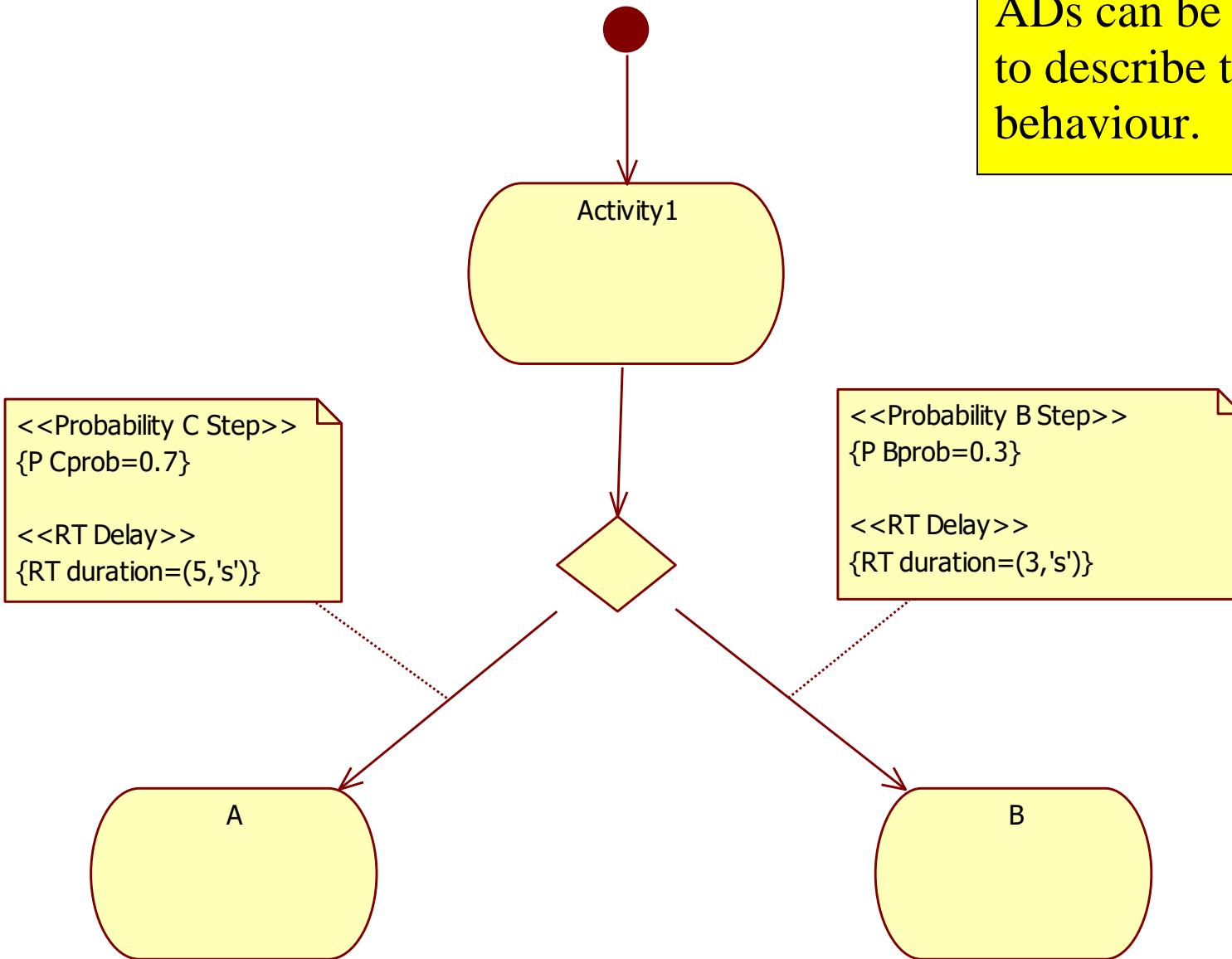
UML must be extended with real-time features!

Activity Diagrams (ADs)

- ADs are used for ***requirement analysis*** of real-time systems.
- ADs are ***isomorphic*** with state charts (i.e. state machines). So, they are isomorphic with different kinds of Petri nets.
- ADs are most used as concurrent flowcharts.
- ADs are suited for description or representation of algorithms (like flowcharts).
- A behavior can be modeled as a set of control flows with operations:
 - sequences,
 - alternatives
 - loops
 - forks
 - joins.

This feature is similar to PNs.

ADs cannot be used for behavior verifications. The Petri nets can be simulated and more than this, the model can be verified by their PN based languages.



ADs can be extended to describe timing behaviour.

Transformations of ADs to PNs and vice versa

E.g.: Simple elevator system (two floors)

M: motor

st_up: start up control signal;

st_down: start down;

req_up: request up;

req_down: request down;

ar_up: arrived up // event signaled by sensor;

ar_down: arrived down // event signaled by sensor.

Is it a real time application? Why?

Has it deadlines? ← Positioning accuracy

How can be the *deadlines* calculated (found)?

Add the specifications required for the deadline calculation.

Tasks:

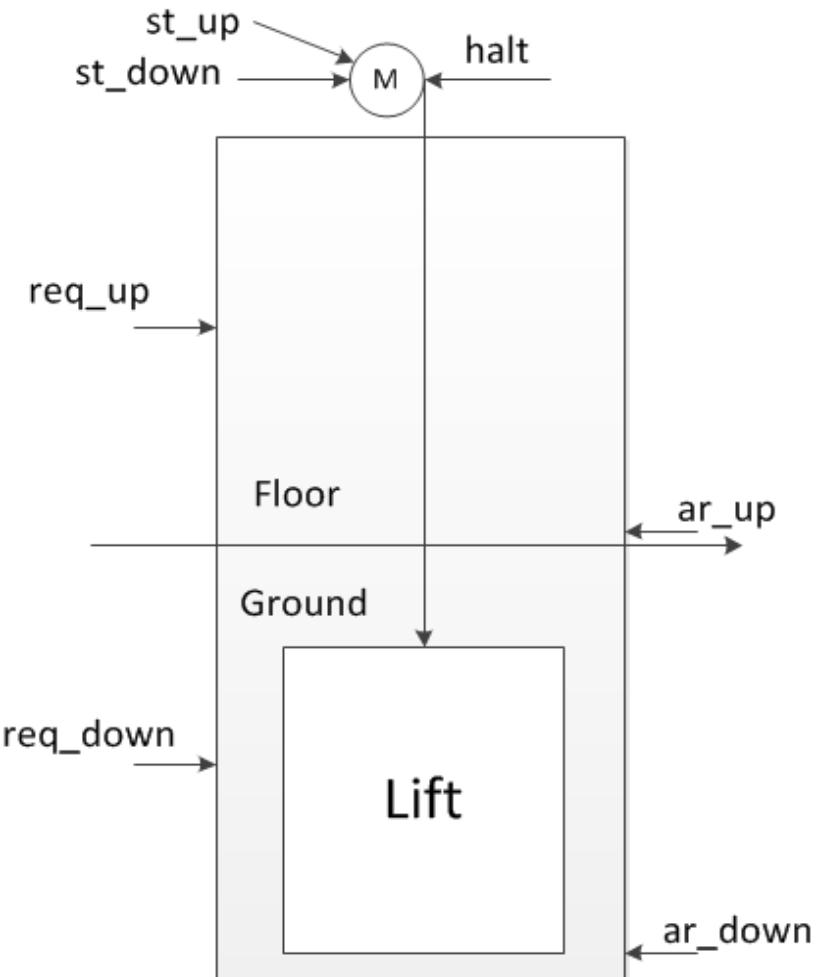
Build the the component diagram that includes:

controller, plant and user. ← homework

Build the plant ETPN model!

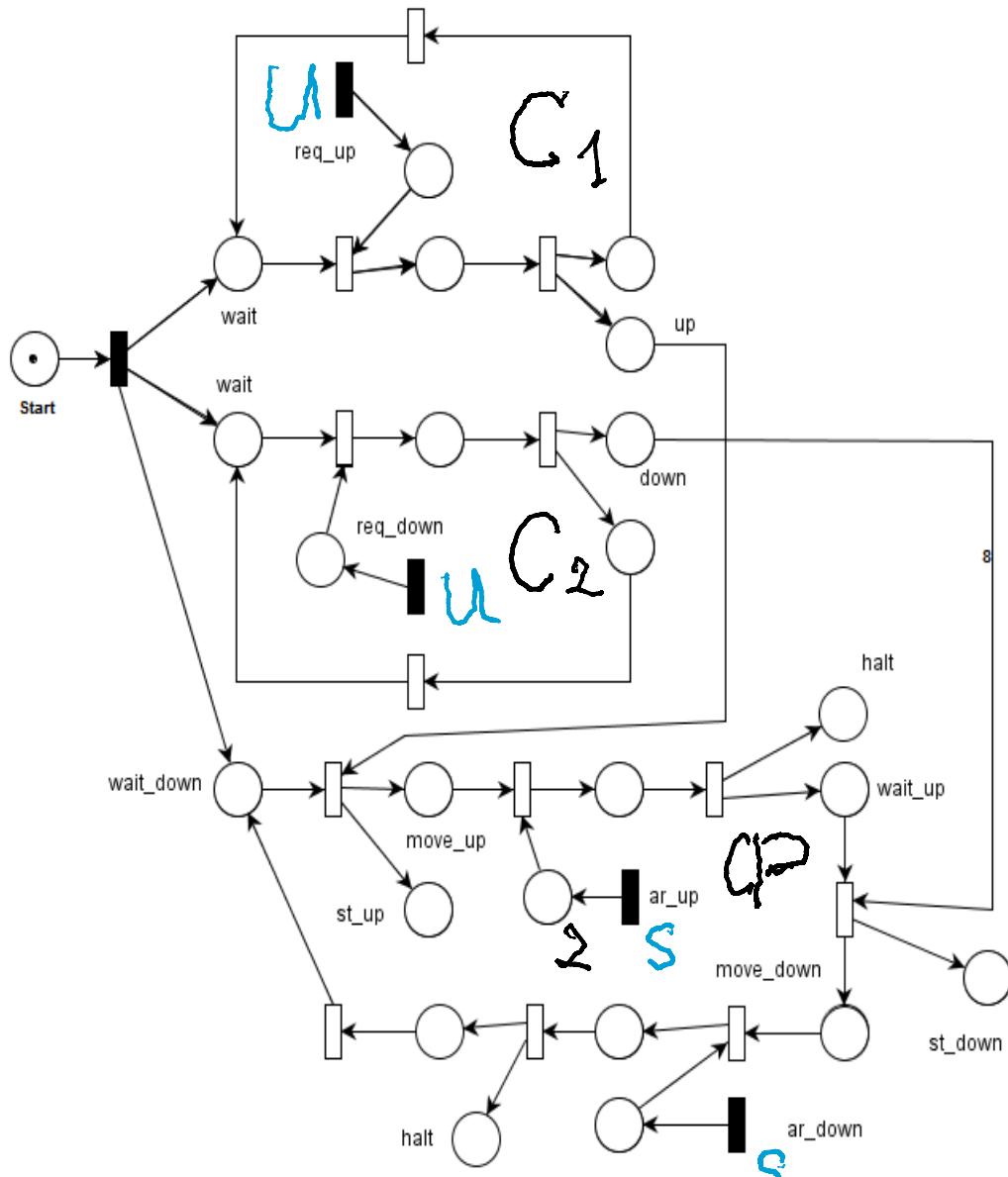
Build the controller ETPN model.

Build the activity diagram model.

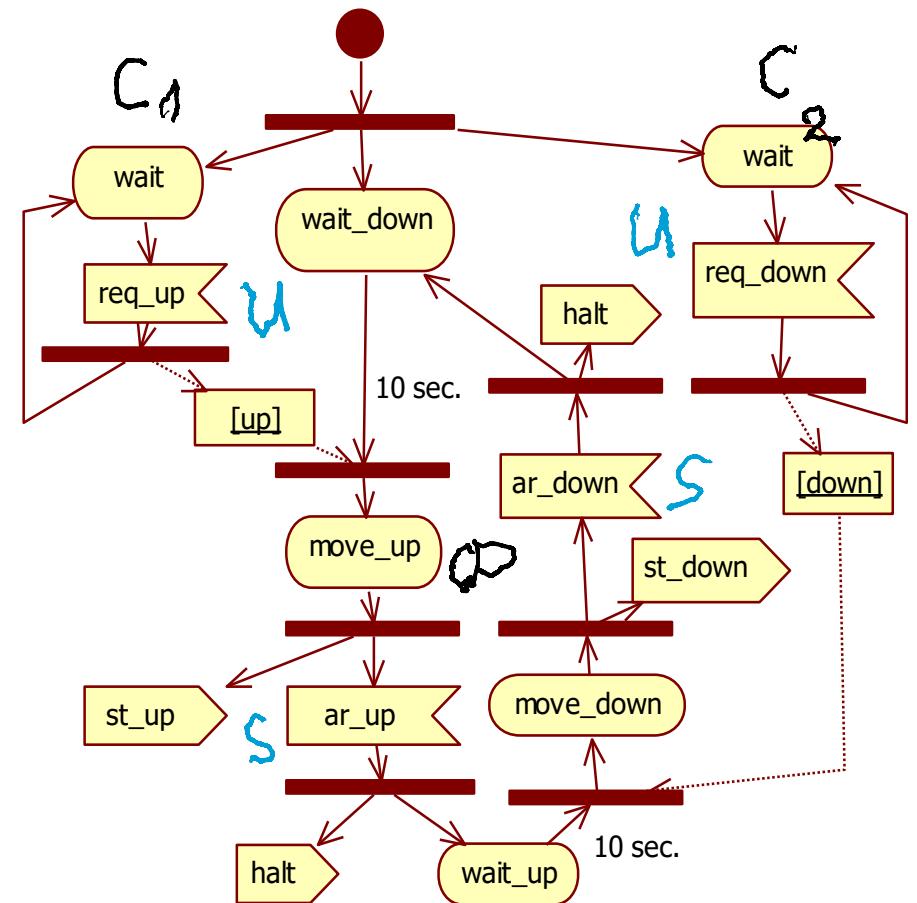


InpChannelSet = {.....}

OutChannelSet = {.....}



ETPN model of the controller and plant



$e_1 = ar_up \rightarrow halt$; positioning accuracy $\leq x$
 $mm \rightarrow D_1 = 1\text{ ms}$
 $e_2 = ar_down \rightarrow halt$; idem $\rightarrow D_2 = 1\text{ ms.}$
(e.g.)
 $e_3 = req_up \rightarrow st_up$; $D_3 = 1000\text{ ms.}$ (depends on the state)
 $e_4 = req_down \rightarrow st_down$; $D_4 = 1000\text{ ms.}$

Home work: Conceive the plant ETPN model. Write the ETPNL formal descriptions of all components..

Construct the AD model and the OETPN model of a lift plant for 4 levels (floors)!

Can the Ads be used for verification?

What have to be verified and what can be verified with ADs?

How can be proved that AD and ETPN are izomorphic?

Conception approach decision and verification

Selection criteria for using synchronous approach or asynchronous approach (i.e. single (mono) tasking or multitasking approach)

Real-Time problem definition:

$E = \{e_1, e_2, \dots\}$ set of input events;

$A = \{a_1, a_2, \dots\}$ set of activities

$e_i \rightarrow reactionSignal_i; D_i$ (deadline); $i = 1, 2, \dots, n$;

The reaction to an input event requires the execution of a sequence of activities.

Let θ_i be the duration of all the activities involved by the response to the input event (i. e. reaction); $i = 1, 2, \dots, n$;

Let $E_k = \{e_i, e_j, \dots\}$ be the worst case when the mentioned event occurs simultaneously and Θ_k be the worst case duration of all the involved activities.

The application can be tackled using the *asynchronous approach* with a single task if and only if

$$\text{For } E_k \text{ and all } e_i \text{ of } E_k \rightarrow \Theta_k \leq D_i$$

The multitasking approach is compulsory if there exists at least one E_k and e_i in E_k such that the previous condition is not fulfilled.

How many tasks are needed for implementation of an application that requires more than one thread?

How can be approached the simple elevator control system?

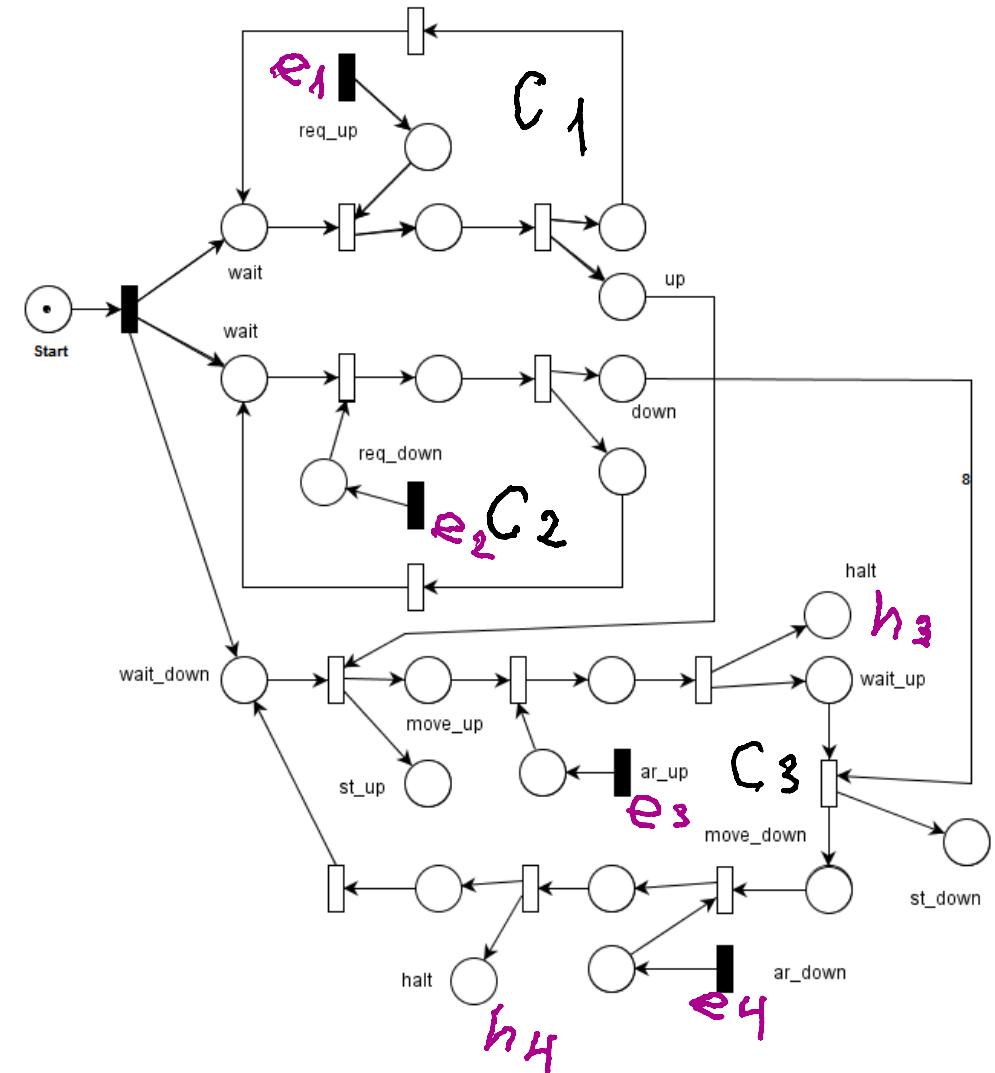
- Synchronous approach
- Asynchronous approach

What must be implemented? → C_1, C_2, C_3 ;
Concurrent events: $\{e_1, e_2, e_3\}, \{e_1, e_2, e_4\}$
Let us suppose that the activities involved in $C_2 \& C_3$ by e_2 and e_3 to halt ($e_3 \rightarrow h_3$) requires less than D_3 , but there is no time for performing any another activity before the expected reaction (deadline). This leads to conclusion: the loop must be implemented by an independent task. The other loops can fulfill their temporal constraints if they are implemented in a single thread.

Conclusion: the application must be implemented by 2 tasks: $C_1 + C_2 \& C_3$.

Question: are needed here priorities?

What are the roll of the priorities?



req_up → st_up;
req_down → st_down.

Principles of activities deployment (partitioning) on tasks for the asynchronous approach:

1. Find a partition (E_1, E_2, \dots) of the input event set that can happen concurrently at the same time and that involve reactions with specified deadlines.
2. Verify if the RT constraints are met for each set E_i .
3. Improve the partition to get a minimum number of tasks that fulfill the RT constraints in an independent context.
4. Verify if there are events assigned to different sets E_k that can occur simultaneously.
Verify if all of them (independently) fulfill the temporal requirements.
5. Assign priorities to tasks and a **scheduling** criterion (algorithm). Verify if the tasks fulfill the temporal requirements.

UML Design Models

a. Interaction Diagrams

- i. Sequence Diagrams
- ii. Communication Diagram
- iii. Timing Diagram
- iv. Interaction Overview Diagram

UML Object notation:

ObjectName'/'ClassifierRoleName ‘:’ ClassifierName[‘,’ClassifierName]

Combinations:

:C – unnamed instance from a class C

/R – unnamed instance playing the role R

/R:C - unnamed instance from a class C playing the role R

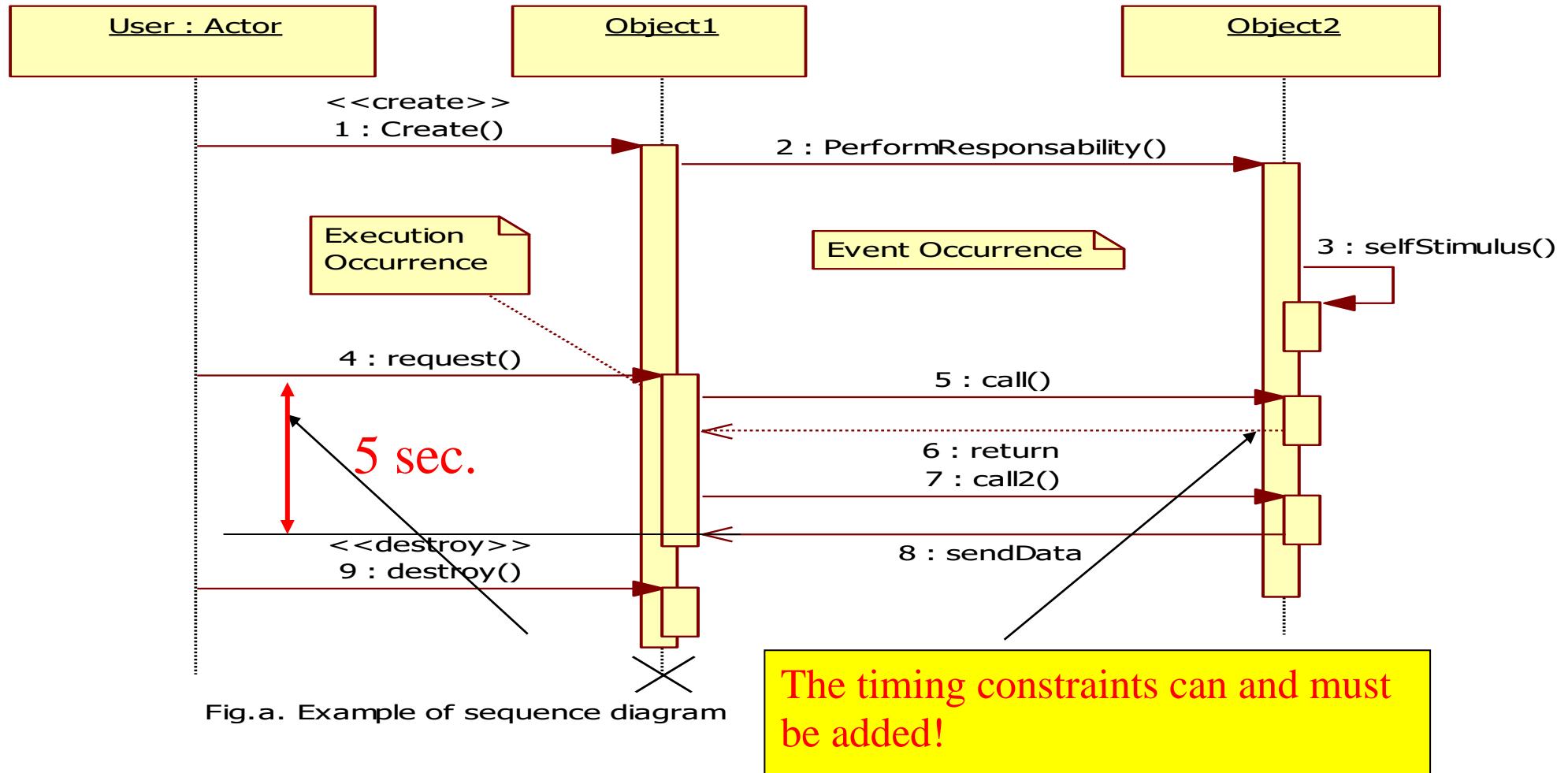
O/R – instance named O playing the role R

O/R:C – instance named O from a class C playing the role R

O – instance named O

/ - unnamed instance, unnamed class, unnamed role

i. Sequence Diagram



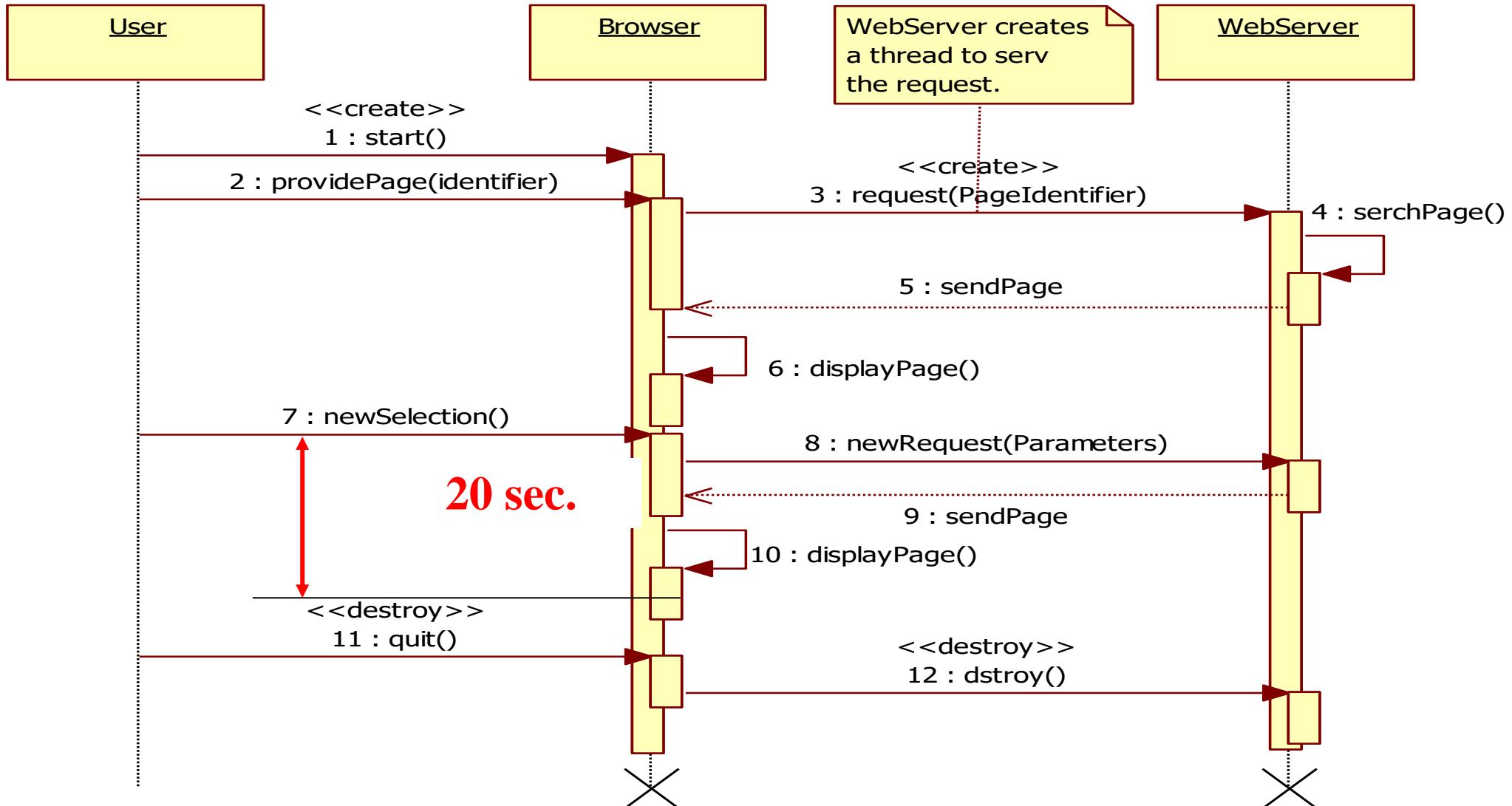


Fig. b. Interaction Browser-WebServer. Sequence Diagram.

A ***Sequence diagram*** describes interactions among objects.

The interactions are represented in a chronological order.
The sequence diagram contains the objects participating in the interaction and describes the messages they send.

An **object** is represented as a vertical dashed line called the "lifeline."
The lifeline represents the existence of the object during represented particular time.
An object symbol is drawn at the head of the lifeline.
The object represented at the start of the lifetime shows the name of the object and optionally its class separated by a colon. All are underlined.

A **message** is a communication between objects that transports information. The message specifies the activity expected to be performed.
It is represented as a horizontal solid arrow from the lifeline of one object to the lifeline of another object.
The reflexive message can be represented by arrow that starts and finishes on the same lifeline.
The arrow is labeled with the name of the message and its parameters. The arrow may also be labeled with a sequence number.

Execution Occurrence represents the relative time that the flow of control is performed in an object.

Execution occurrence is shown as narrow rectangles on object lifelines.

Event Occurrence represents the sending or receipt of messages.

Interaction Occurrence is a reference to an interaction within the definition of another interaction.

Hierarchical numbering is used for all messages on a dependent message.

The dependent message is the message whose execution occurrence the other messages originate in.

Notes describe the flow of events textually.

Representations - Notations:



Synchronous call of a method
ex. getValue(Parameters)



Return from a call



Synchronous event signal



Asynchronous call of a method



Asynchronous event signal

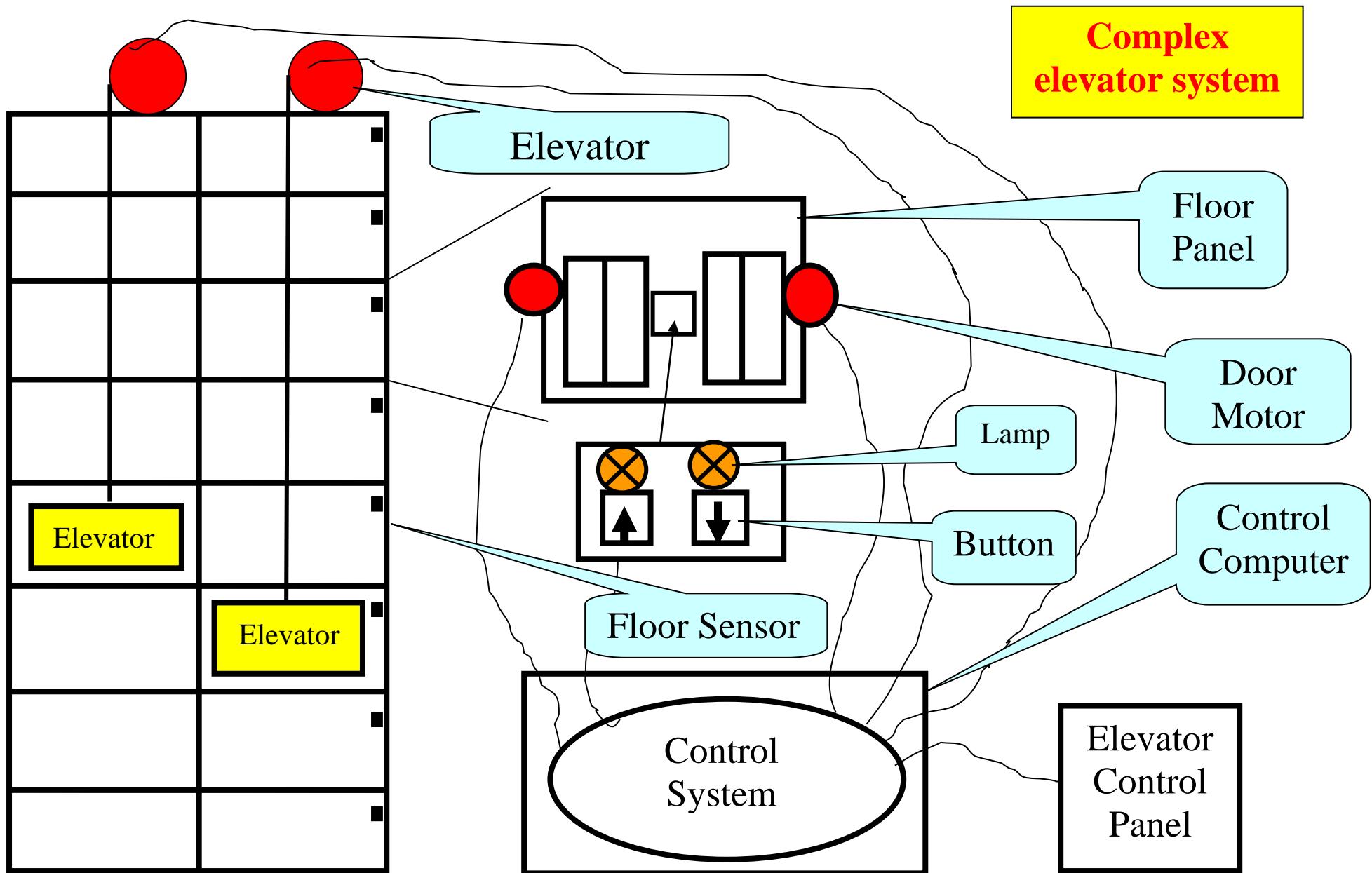
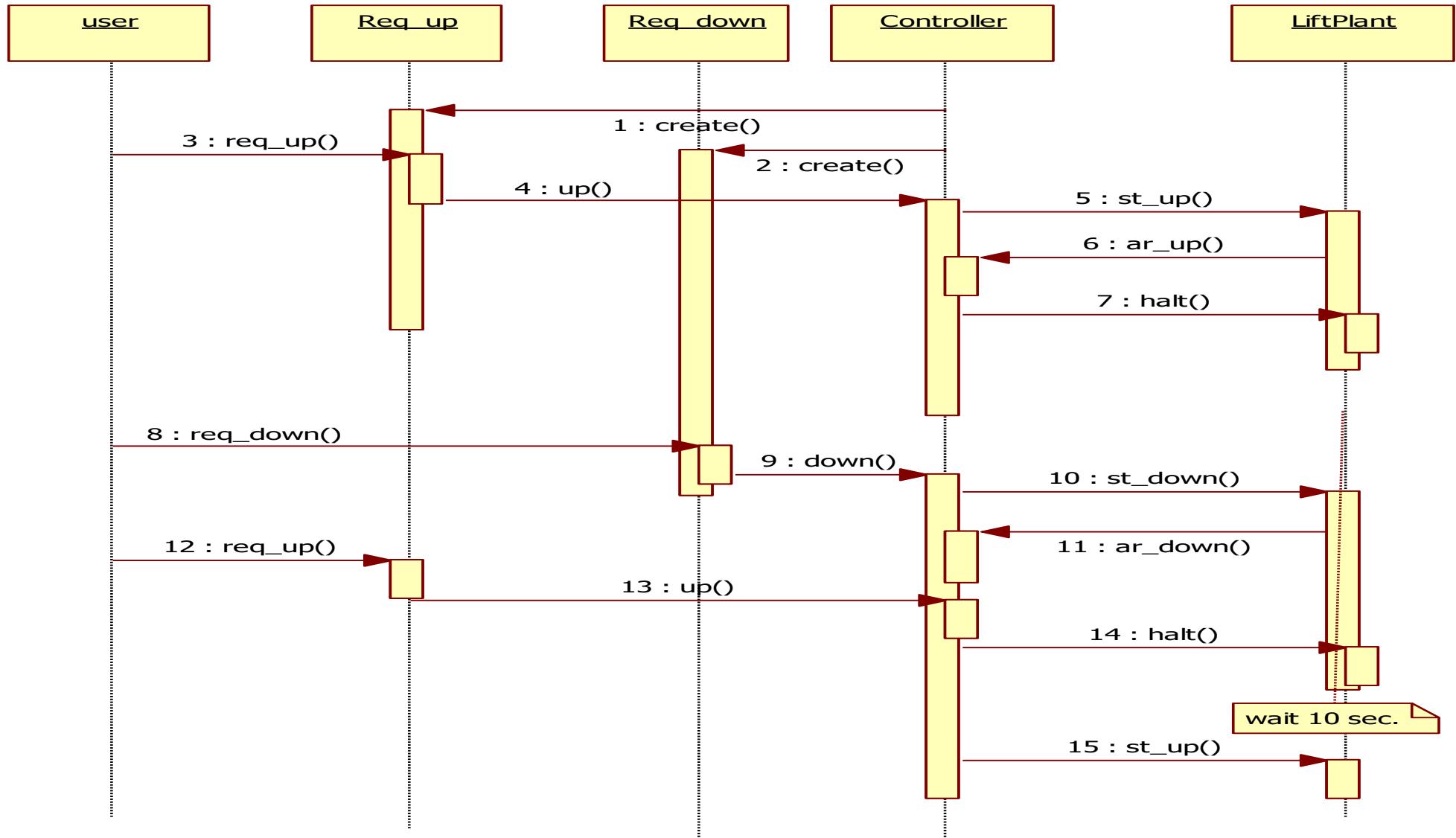


Fig. x. Elevator functional diagram.



Simple lift system – sequence diagram. **Add the timings!**

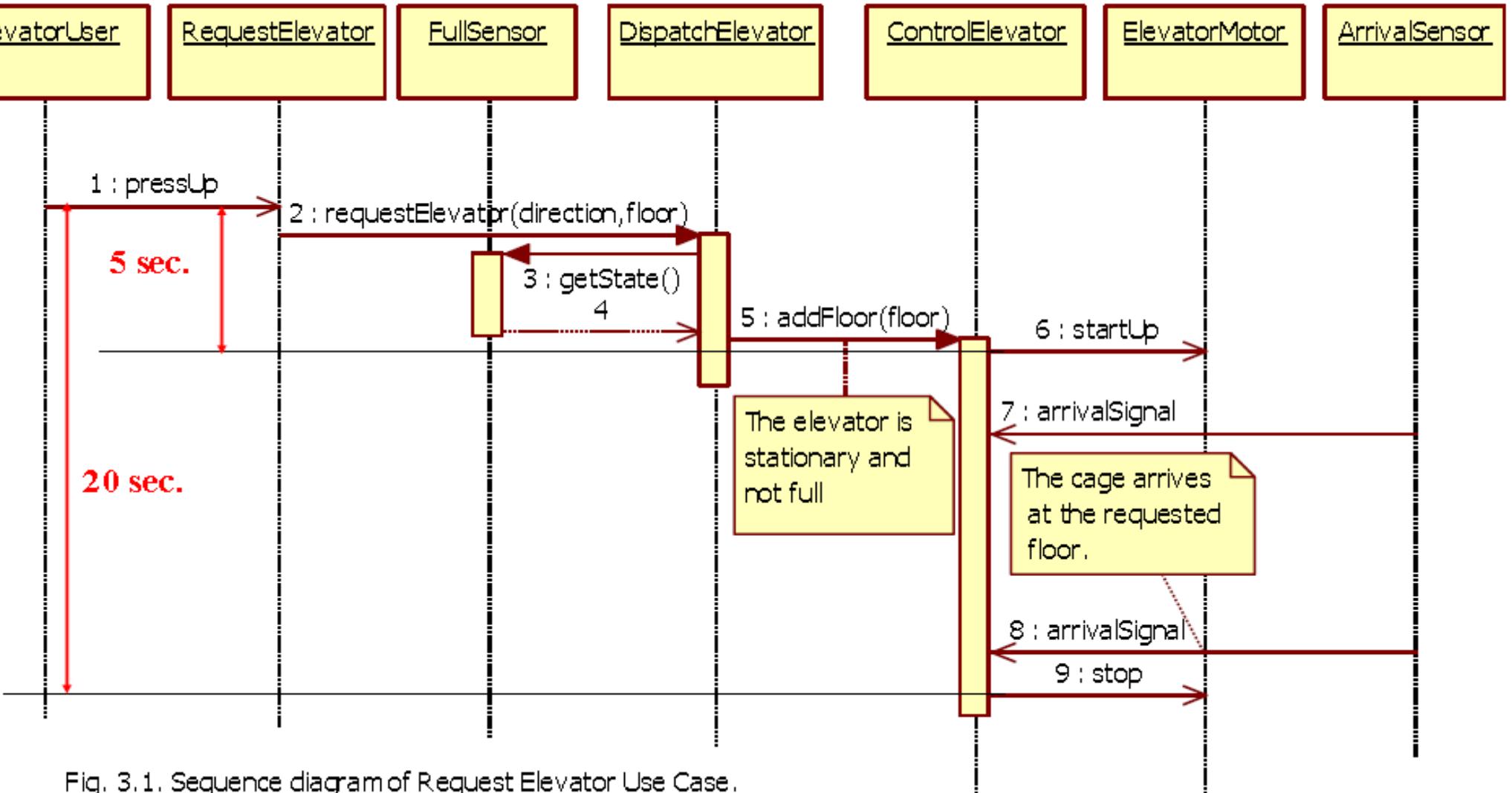


Fig. 3.1. Sequence diagram of Request Elevator Use Case.

Implementation recommendations

The destination object of the message (or signal) can be *active or passive*.

The interaction can be:

- **synchronous (call or signal)**

- the called object wait to be “asked” → object synchronization.
- the called object wait to be notified → object synchronization.

- **asynchronous (call or signal)**

- the destination object doesn’t wait the interaction → interruption or delay (***pending events***).

1. How can asynchronous call of a method be implemented in Java?

- a. The object implementing the called method is passive.
- b. The object implementing the called method is active. It executes the *run()* method when the call arrives. It must be interrupted to execute something else.

2. How can asynchronous signal of an event be implemented in Java?

- a. The destination object of the signal is passive. Can it react?
- b. The destination object of the signal is active. It executes the *run()* method when the signal arrives. It must be interrupted and must execute something else.

RT Questions:

What are the timing constraints roles?

Where do they appear in the software program?

How can be verified if the programs fulfill or not the timing (temporal) requirements?

ii. Communication Diagrams (Object Collaboration Diagrams; OCDs)

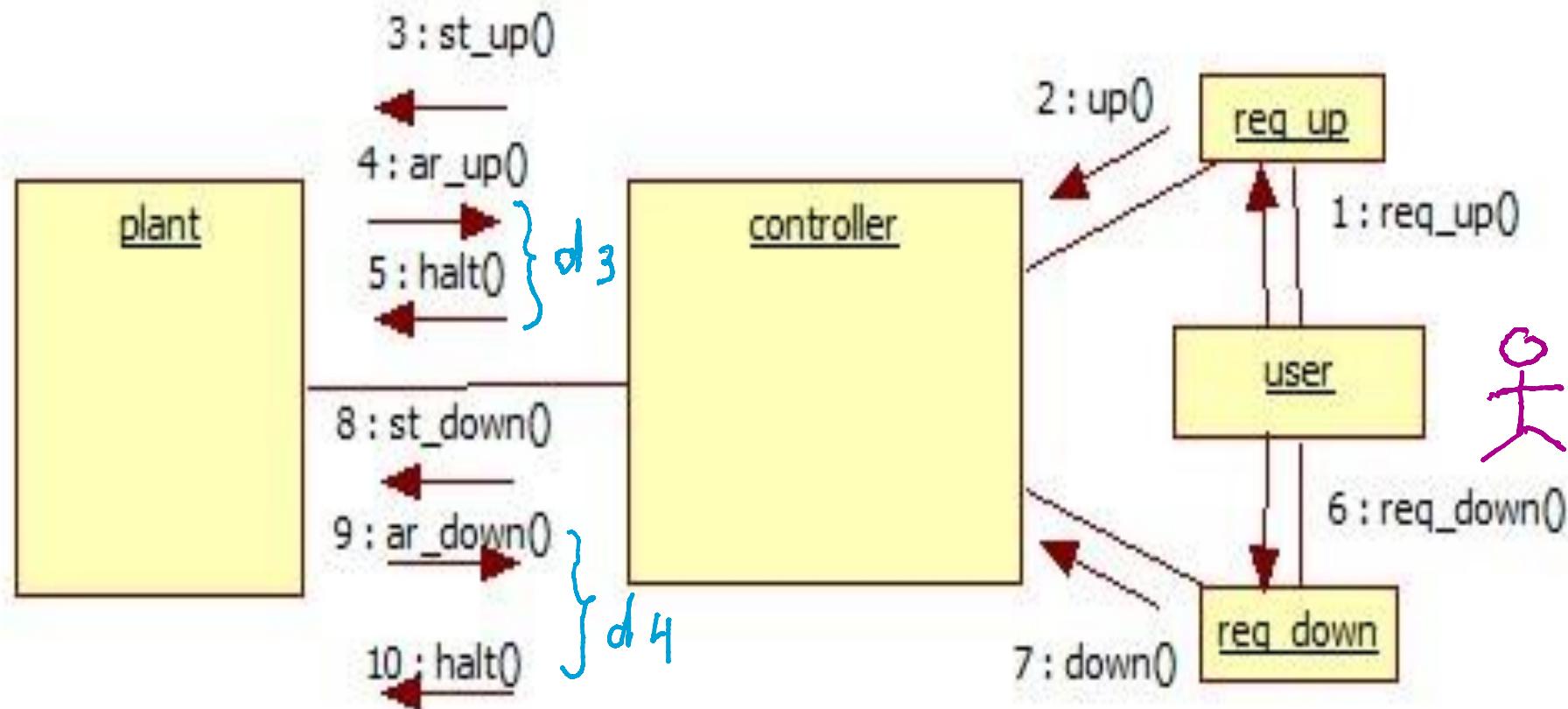
They describe links between a set of interaction objects. It focuses on space.

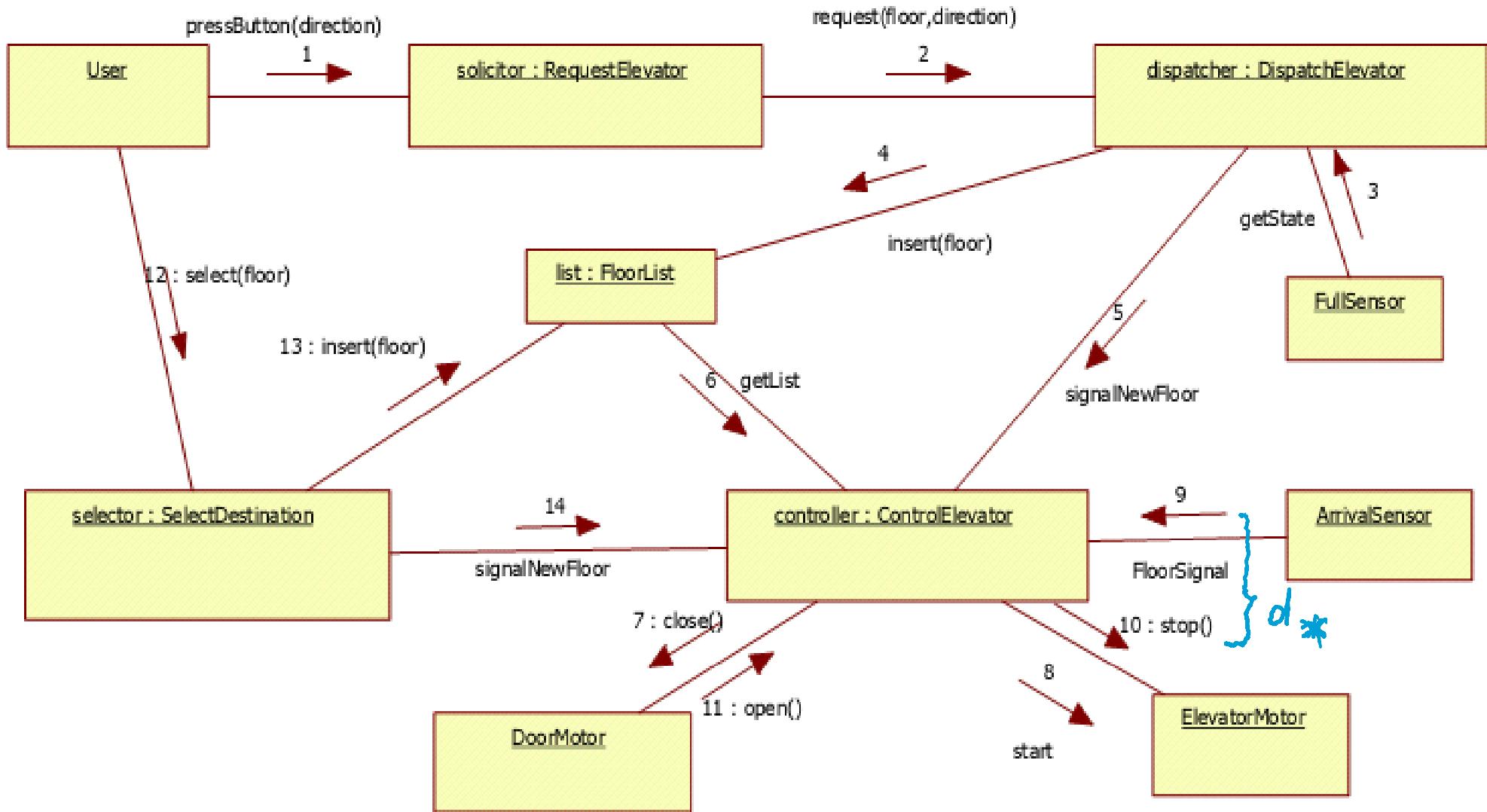
UML 2 Communication diagrams show the message flows between objects in an Object Oriented application.

They are based on relationships between classes:

- association
- composition
- dependency
- inheritance

Simple elevator - collaboration diagram





Complex elevator: Where are the timing constraints?
How can they be added in OCDs?

Homework: write the events that need bounded time reactions and specify their deadlines.

iii. Timing Diagrams

It is used to explore the behaviors of one or more objects throughout a given period of time.

Representations:

- concise notation
- robust notation

Where should and can be added the timing constraints?

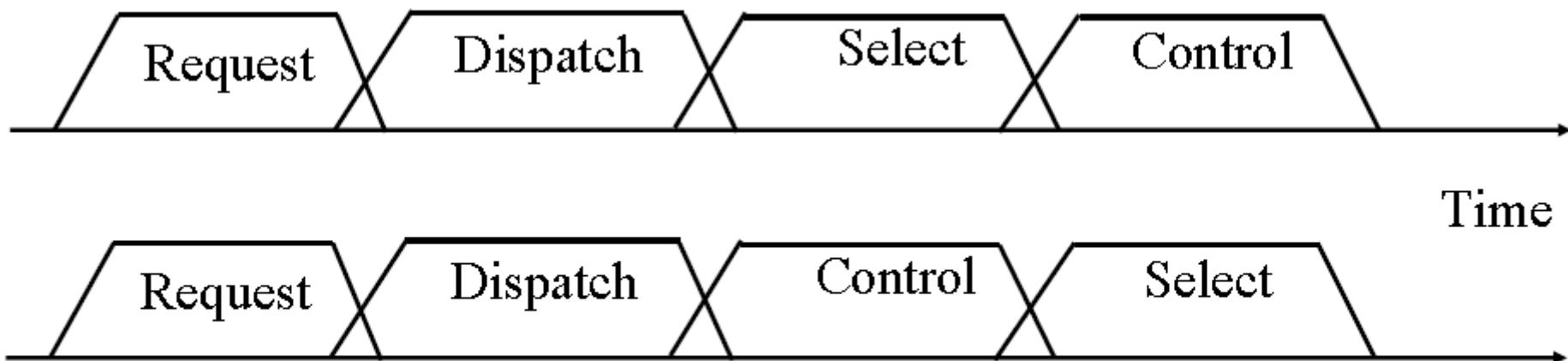


Fig. x. Concise notation.

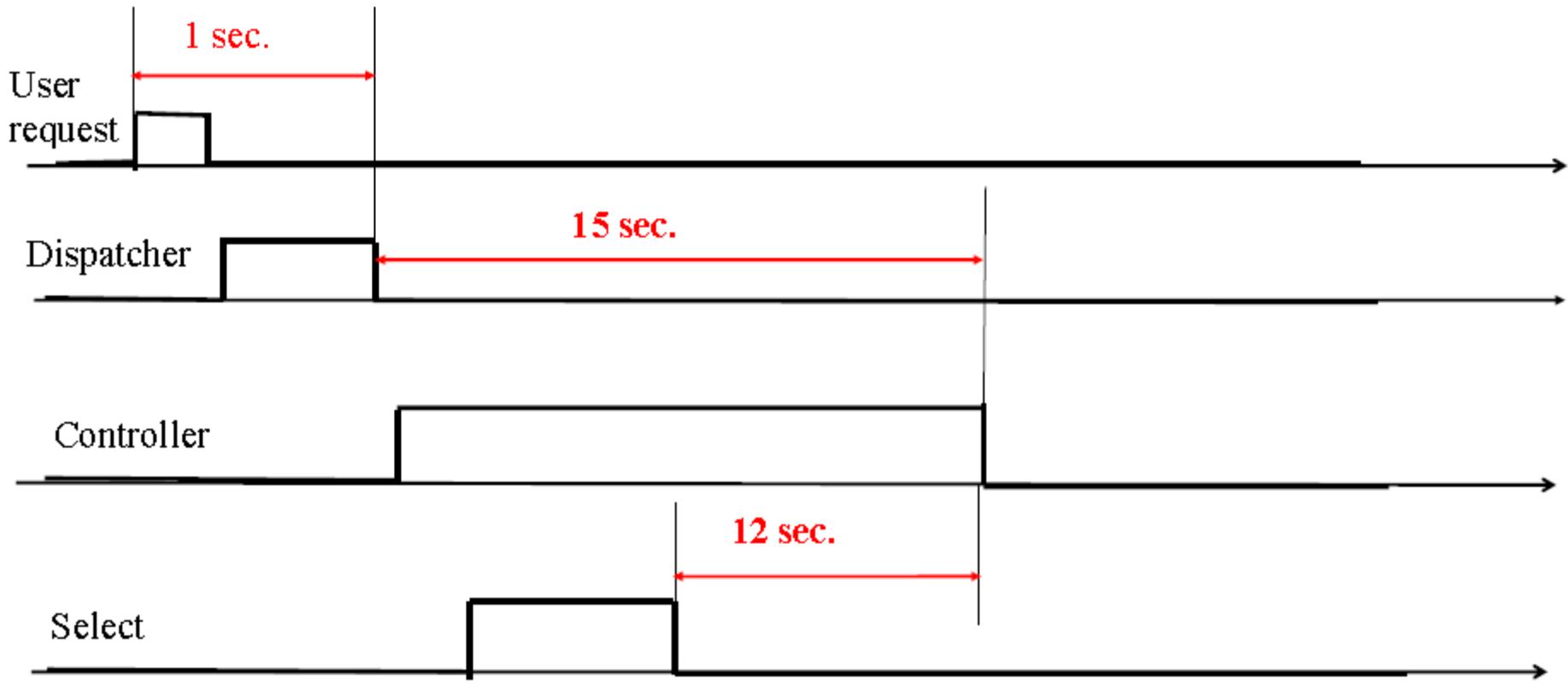


Fig. y. Robust representation of Elevator Timing Diagram.

b. State Machine Diagrams (in UML 2) (Formerly named Statecharts in UML 1)

State machine diagrams depict the dynamic behavior of an entity based on its response to events.

They show how the entities react to various events depending on their current state that. It specifies the sequence of events to which system goes through during its life time at reaction to the events.

UML state machine diagrams are used to describe:

- complex behavior of classes, actors, subsystems, or components.
- real-time systems.

State

In computer science and automata theory, a **state** is a unique configuration of information in a program or machine.

It comprises all the information necessary to determine the further evolution of the system for a given input.

Transitions

A *transition* describes a change from one state to another.

It can be triggered by an event that is either internal or external to the modeled entity.

A transition can be the result of the invocation of an operation that causes an important change of the entity state.

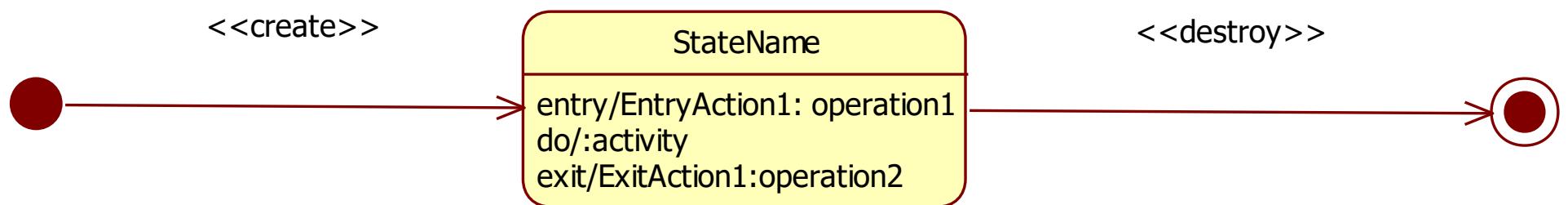
Actions and activity

An **action** is an operation that is invoked by/on the entity.

An **activity** is an operation or set of operations performed by entity when it is on a particular state.

Events:

- Time Event
- Change Event
- Call Event
- Signal Event



Guards

A guard is a condition that must be fulfilled (true) for the transition to be triggered and are optionally represented.

Representation: *trigger [guard]effect*

or *trigger [guard]/list of methods*

trigger – the cause of the transition

guard - logic expression

- free language expression

effect – operation(s) or action executed as effect of the transition – it is automatically invoked by object when the transition occurs.

Pseudostates:

- *initial* or *default*
- *terminal* or *final* – the object does not accept anymore events – it is about to be destroyed
- *junction* – it joins multiple transitions.
- *branch* or *conditional* – junction with guard
- *choice point* – junction that executes its action list before going to the next transition
- *shallow history* – it indicates that the default state of a composite state is the last visited – it does not include nested state.
- *deep history* – it indicates that the default state of a composite state is the last visited of that composite state, including substate nested arbitrarily deeply.
- *fork*
- *join* ≠ *junction*.
- *entry point* – for a composite state – it serves as a connector between its nested state and its peer state.
- *exit point* – for a composite state – similar.

Observation:
All of them can be modeled in PNs

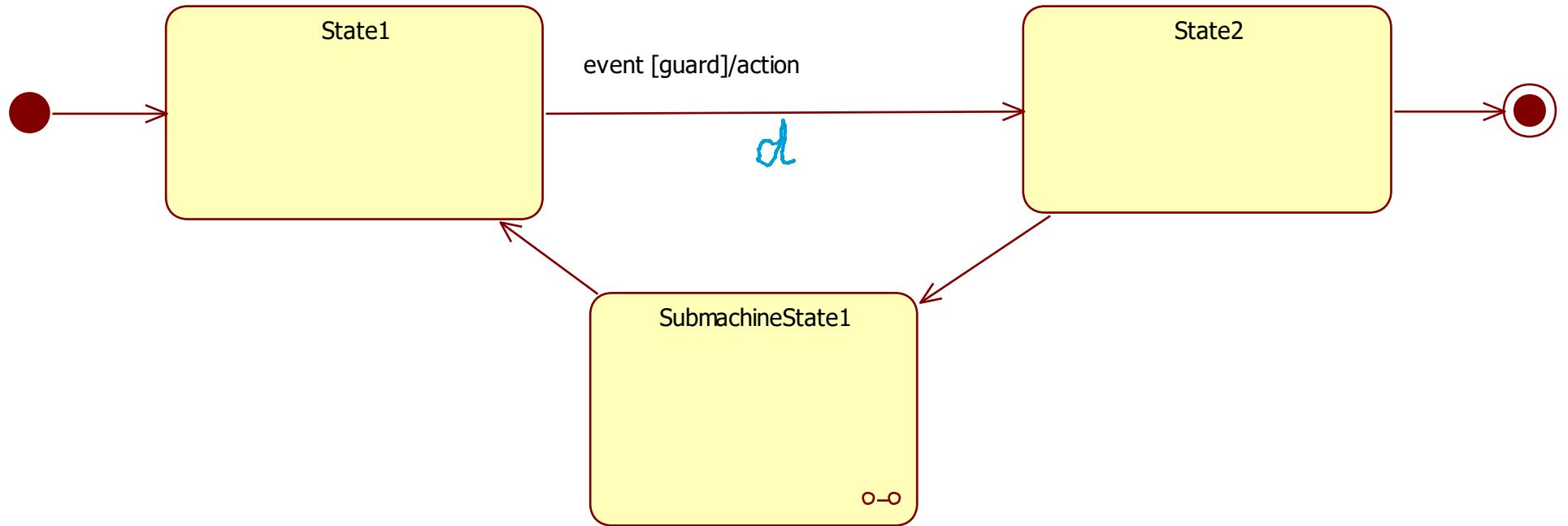
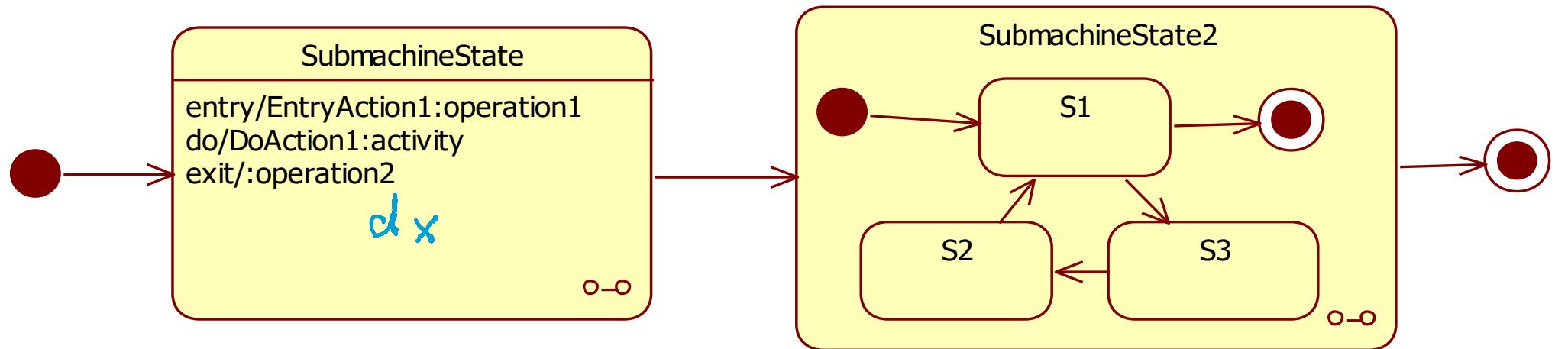


Fig. 1.x. Example of State Machine Diagram.

Where are the timing constraints?



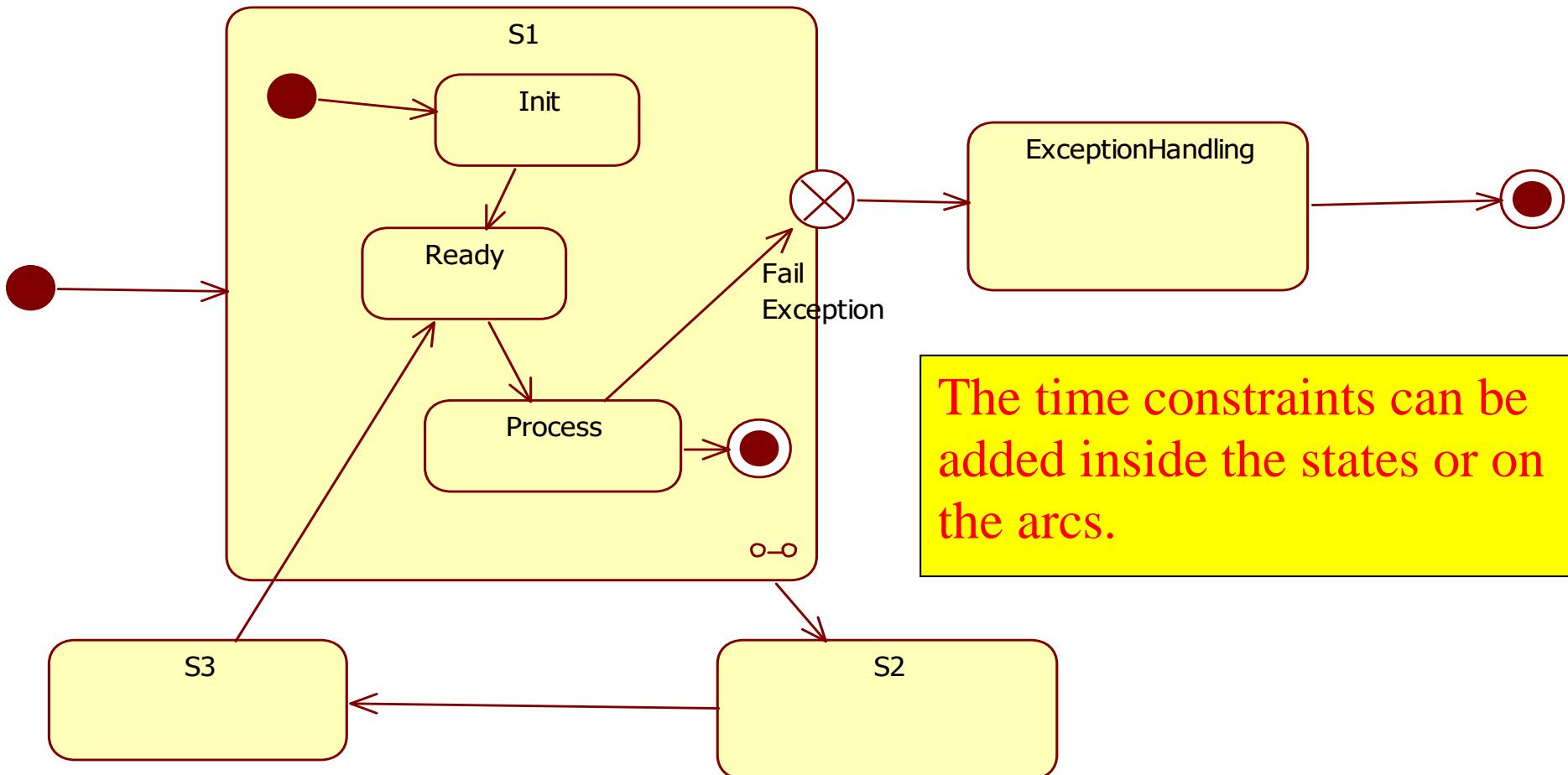
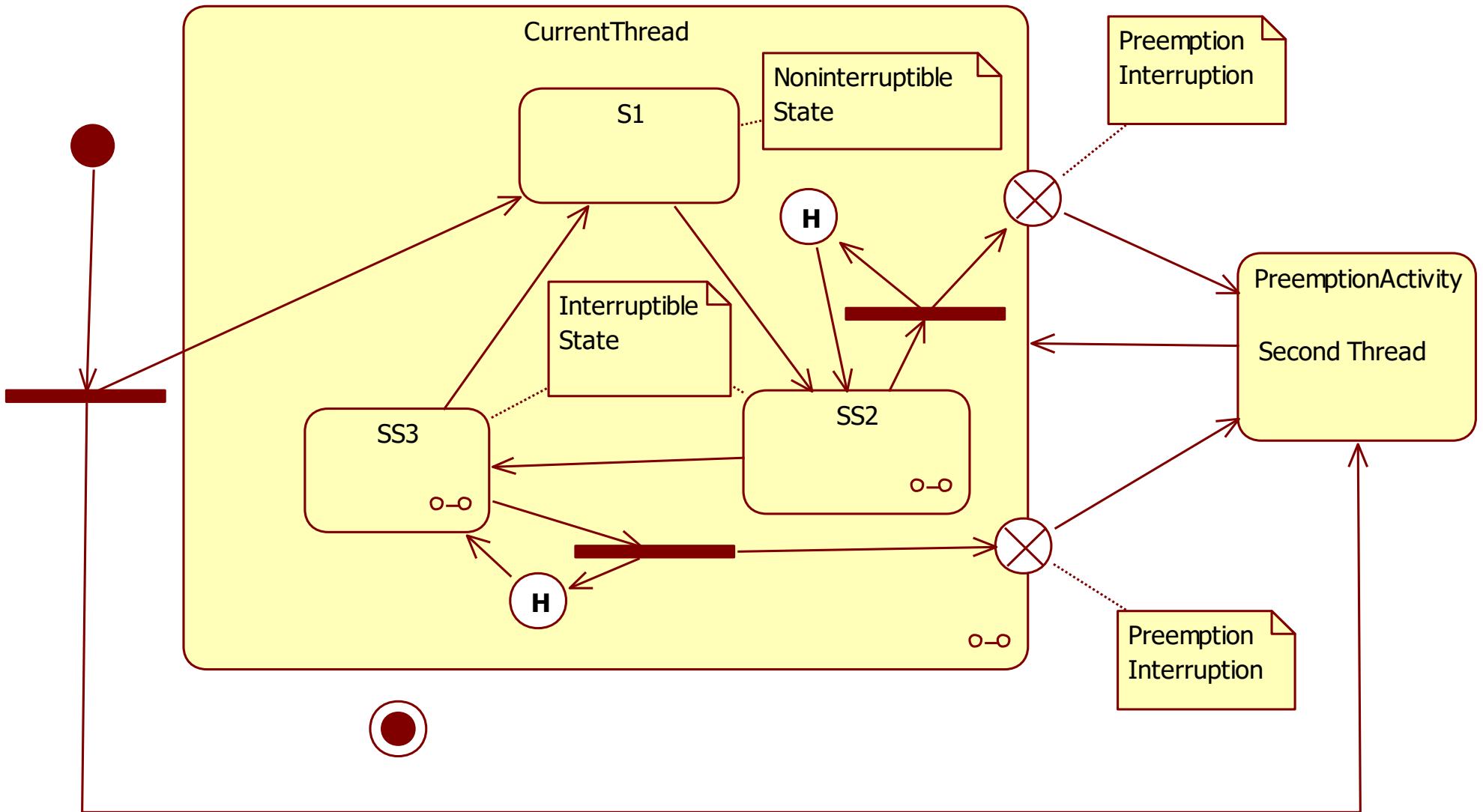


Fig. 1.z. Example of State Machine Diagram with end of flow and init state avoidance.



Using the history and exception pseudostate for preemption description.

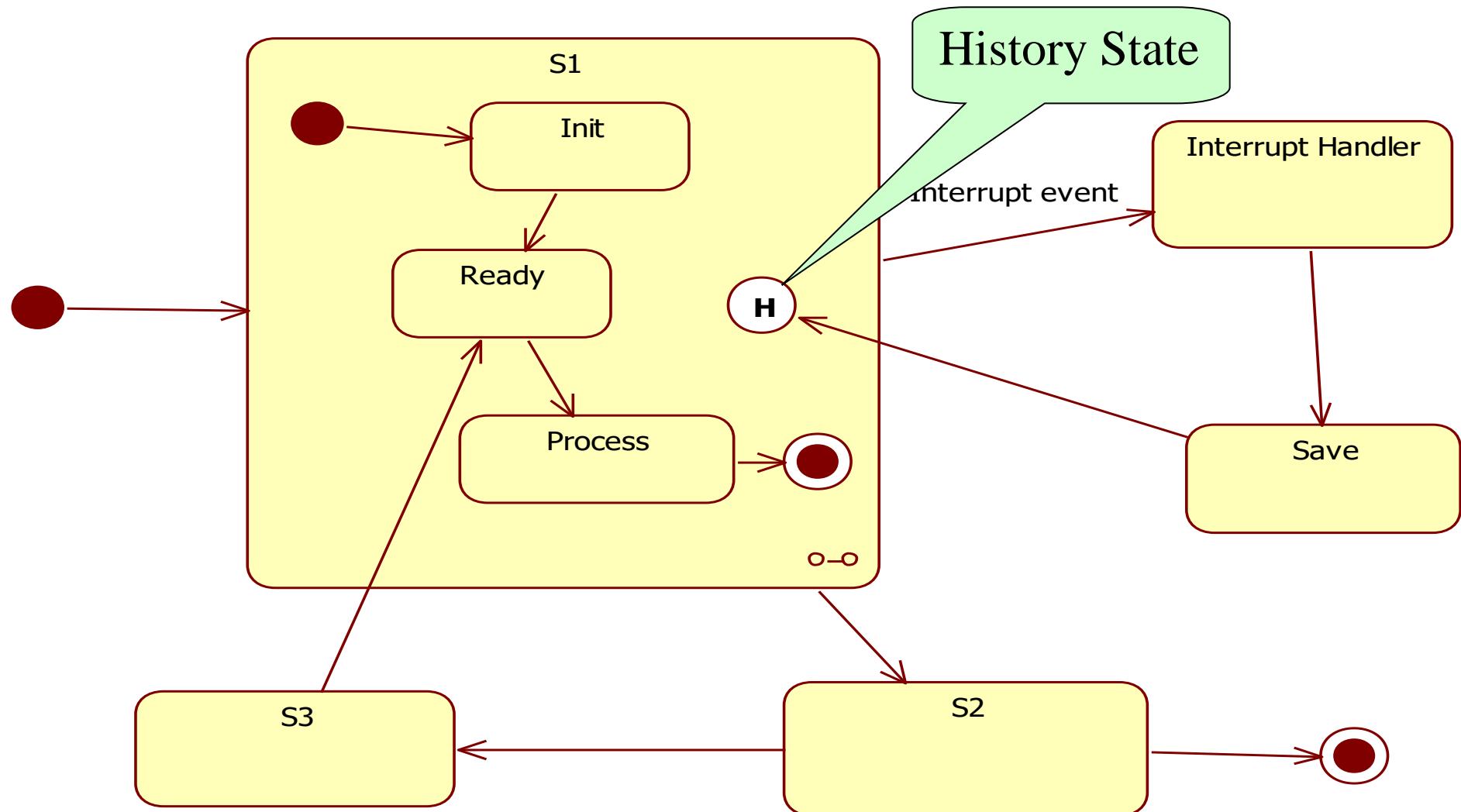


Fig. 1.z. Example of State Machine Diagram with History State.

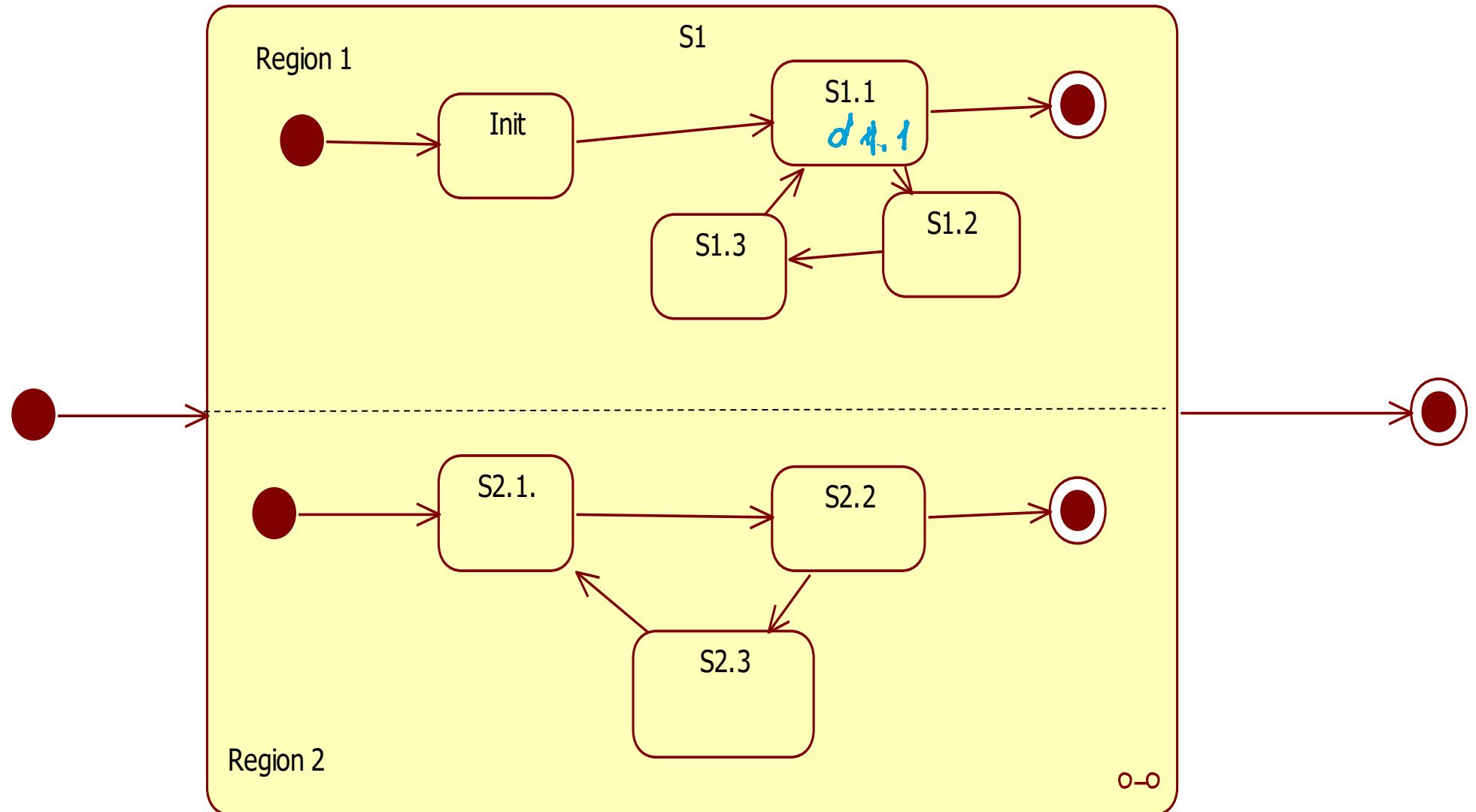


Fig. 1.u. Example of State Machine Diagram with two regions.

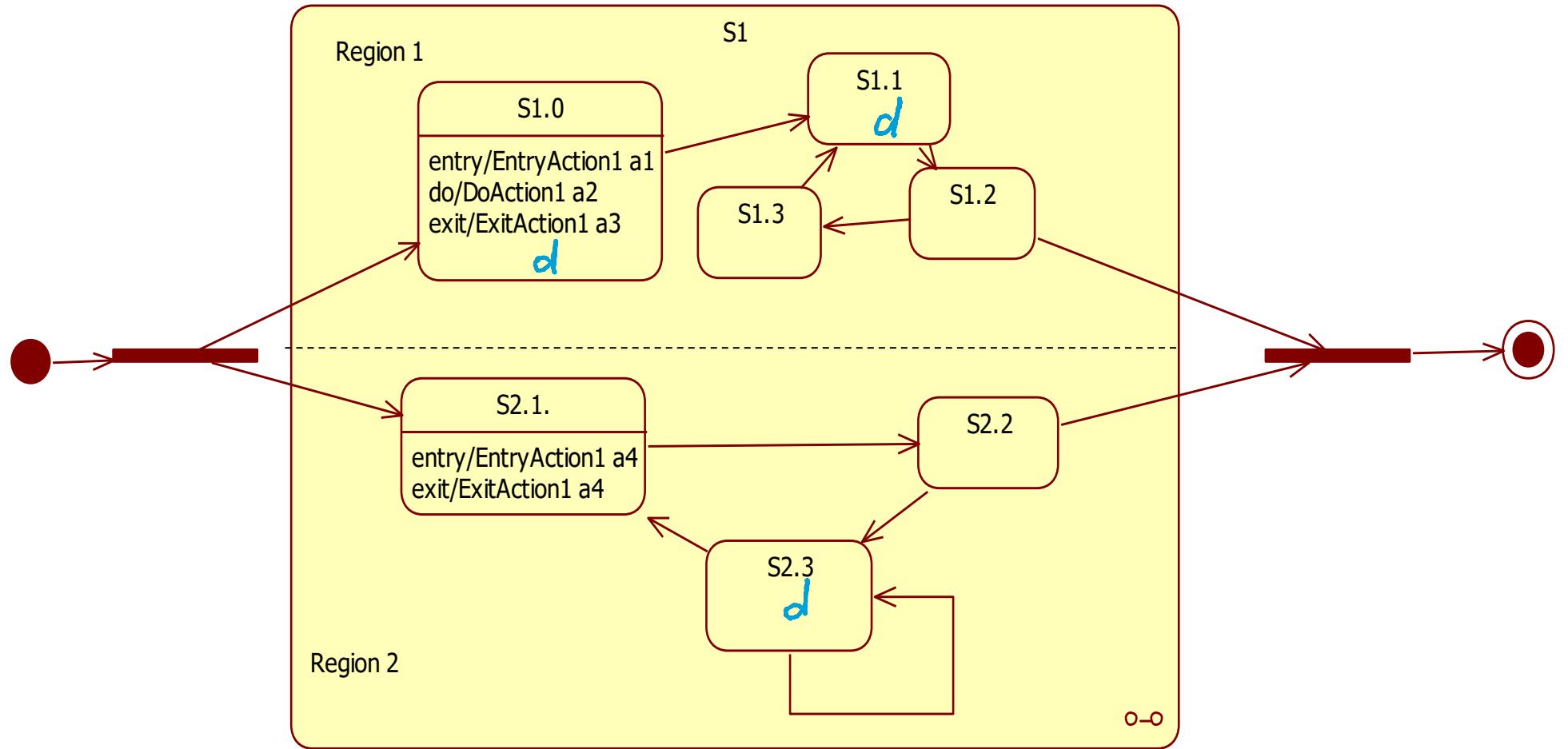


Fig. 1.u. Example of State Machine Diagram with two regions and synchronization.

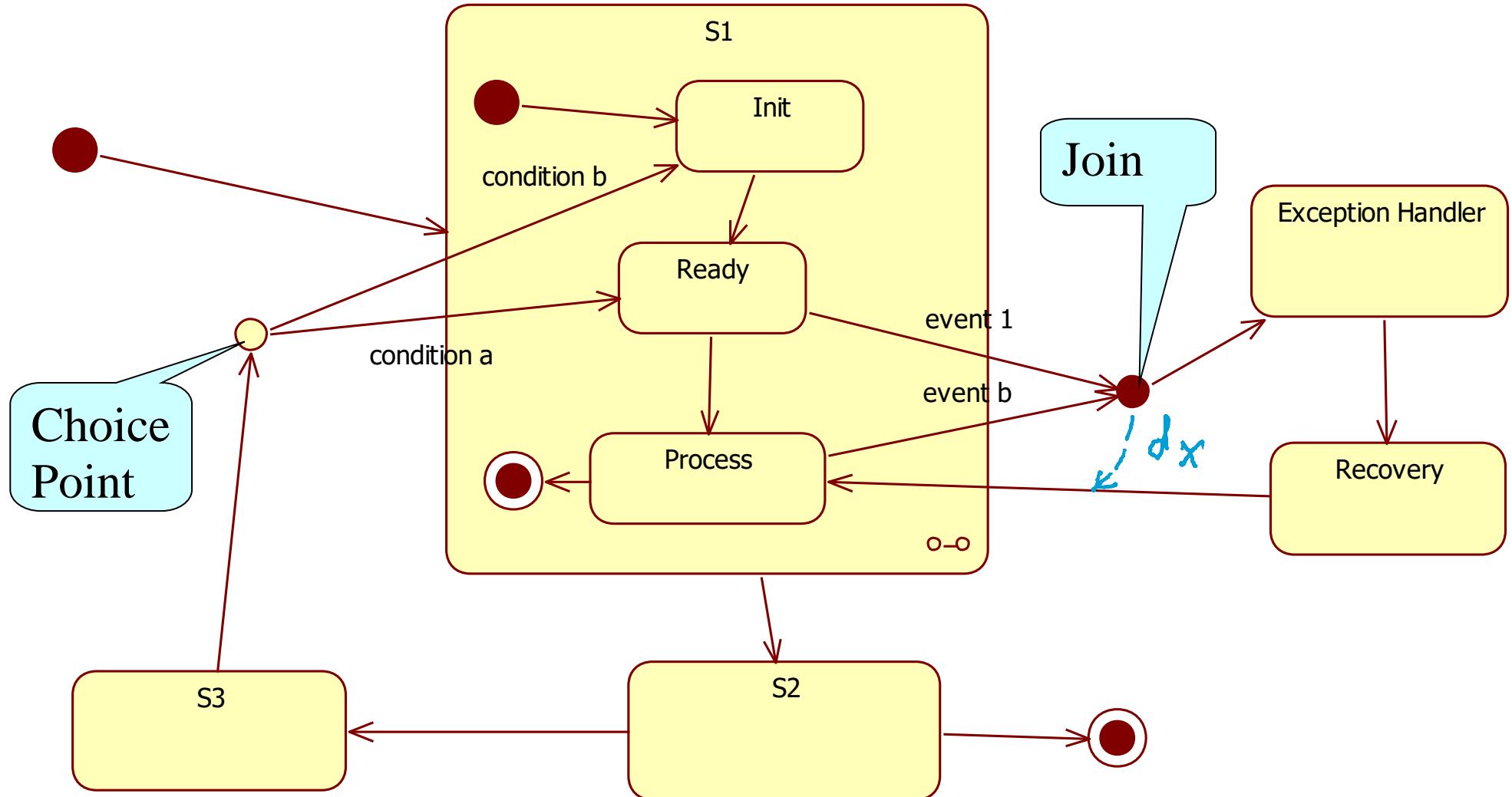
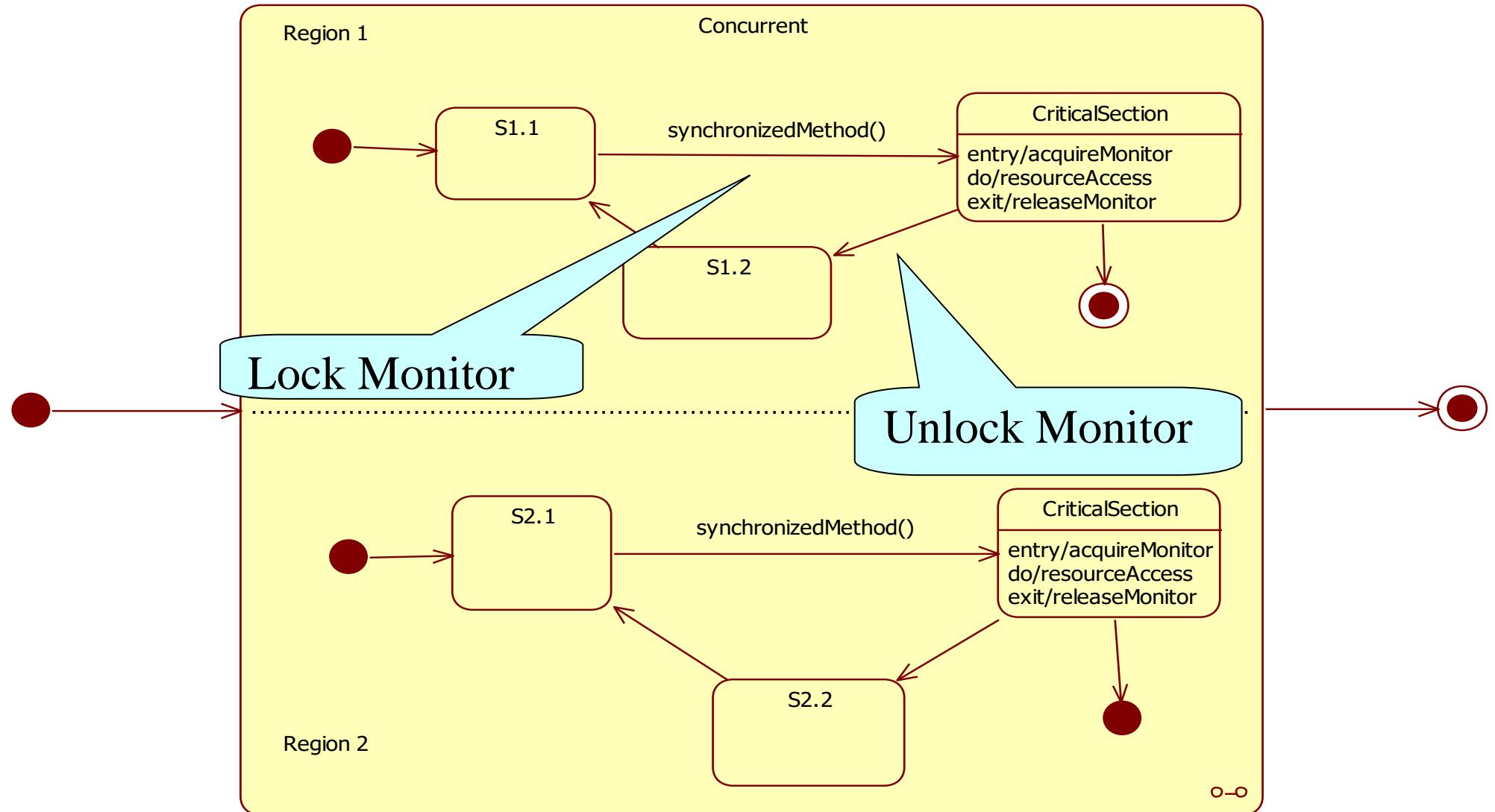


Fig. 1.z. Example of State Machine Diagram with Choice and Join pseudostate.

Critical Sections → Mutual Exclusion



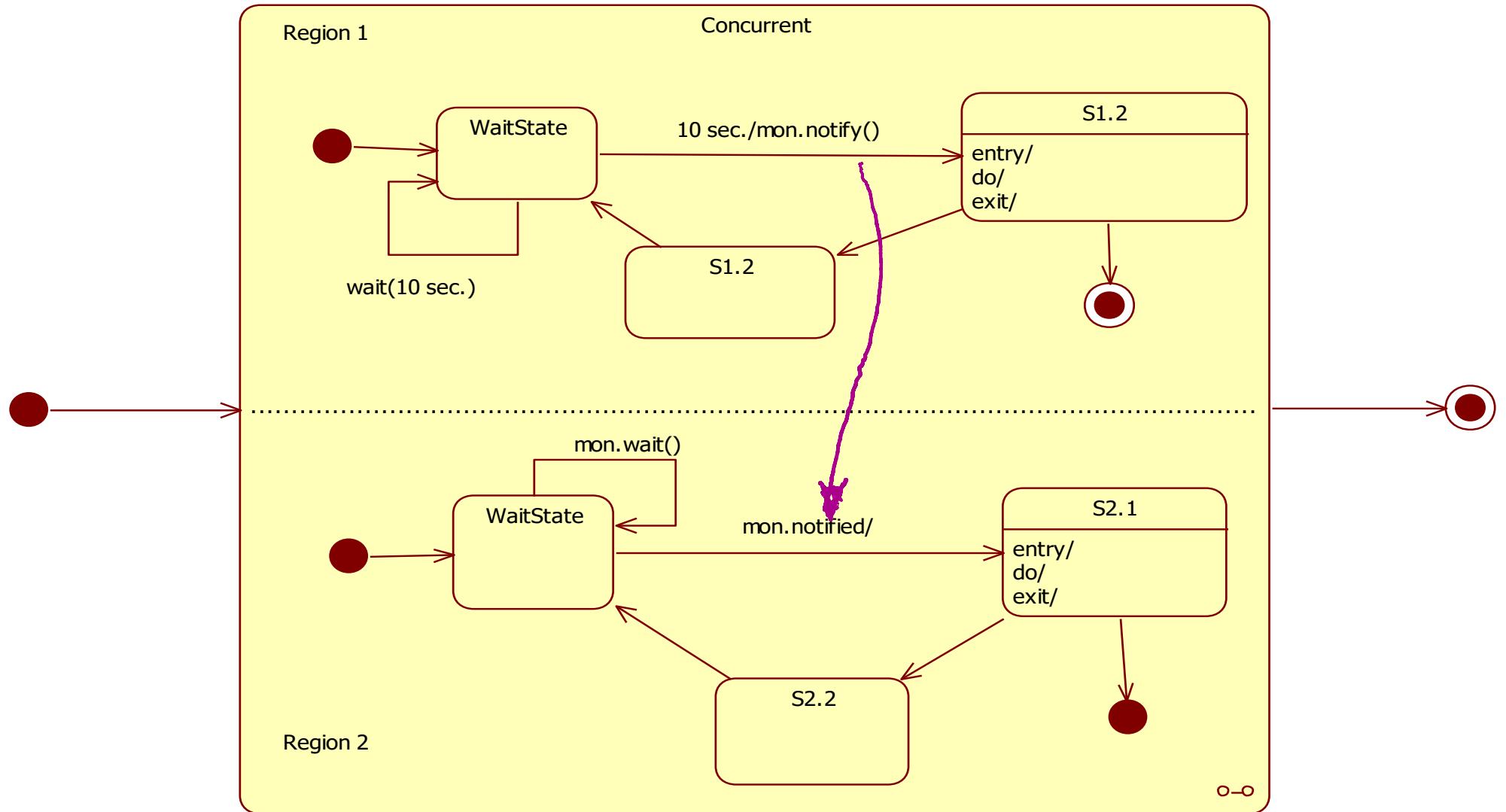


Fig. 1.v. Java synchronization of two threads.

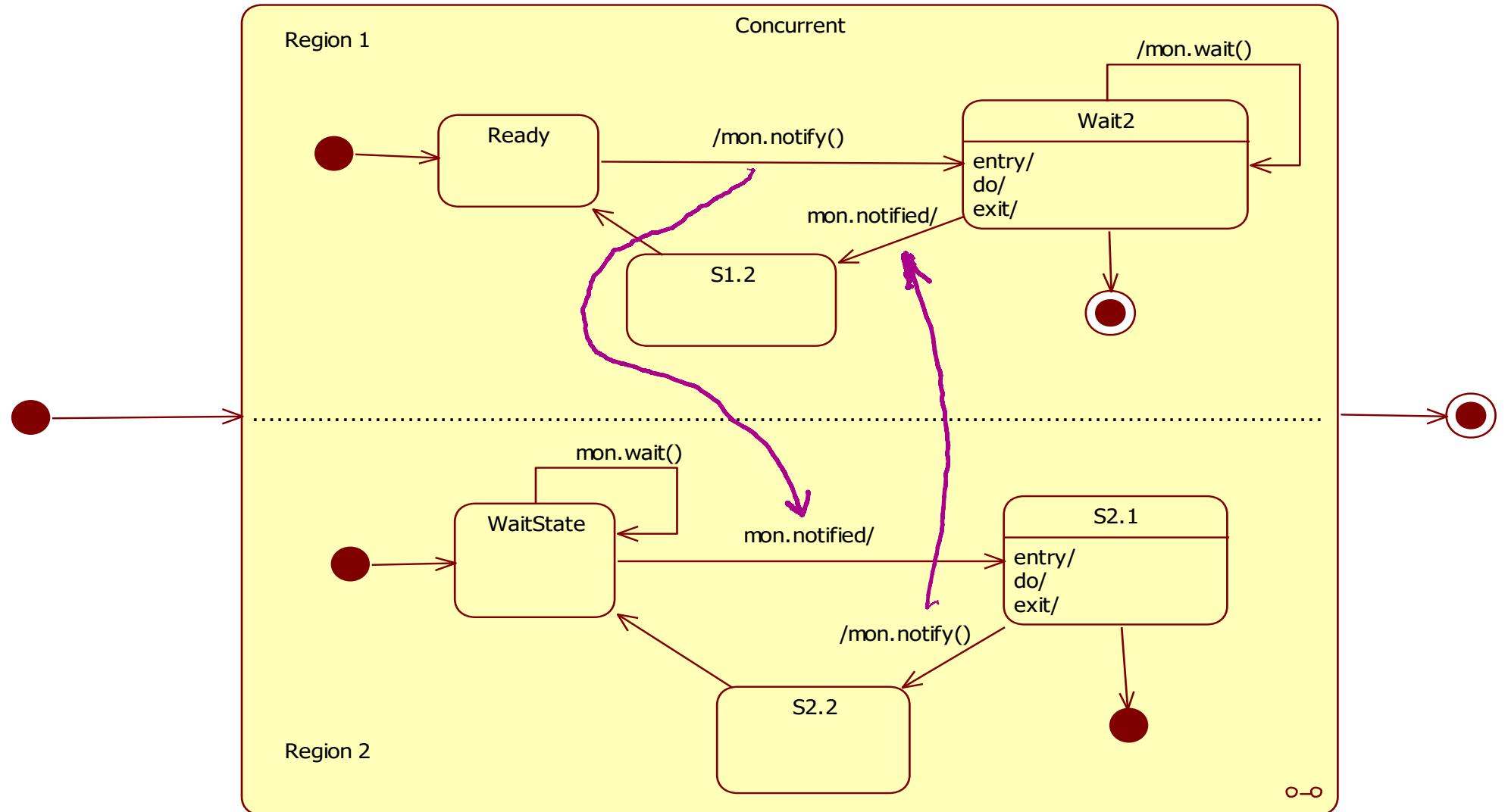
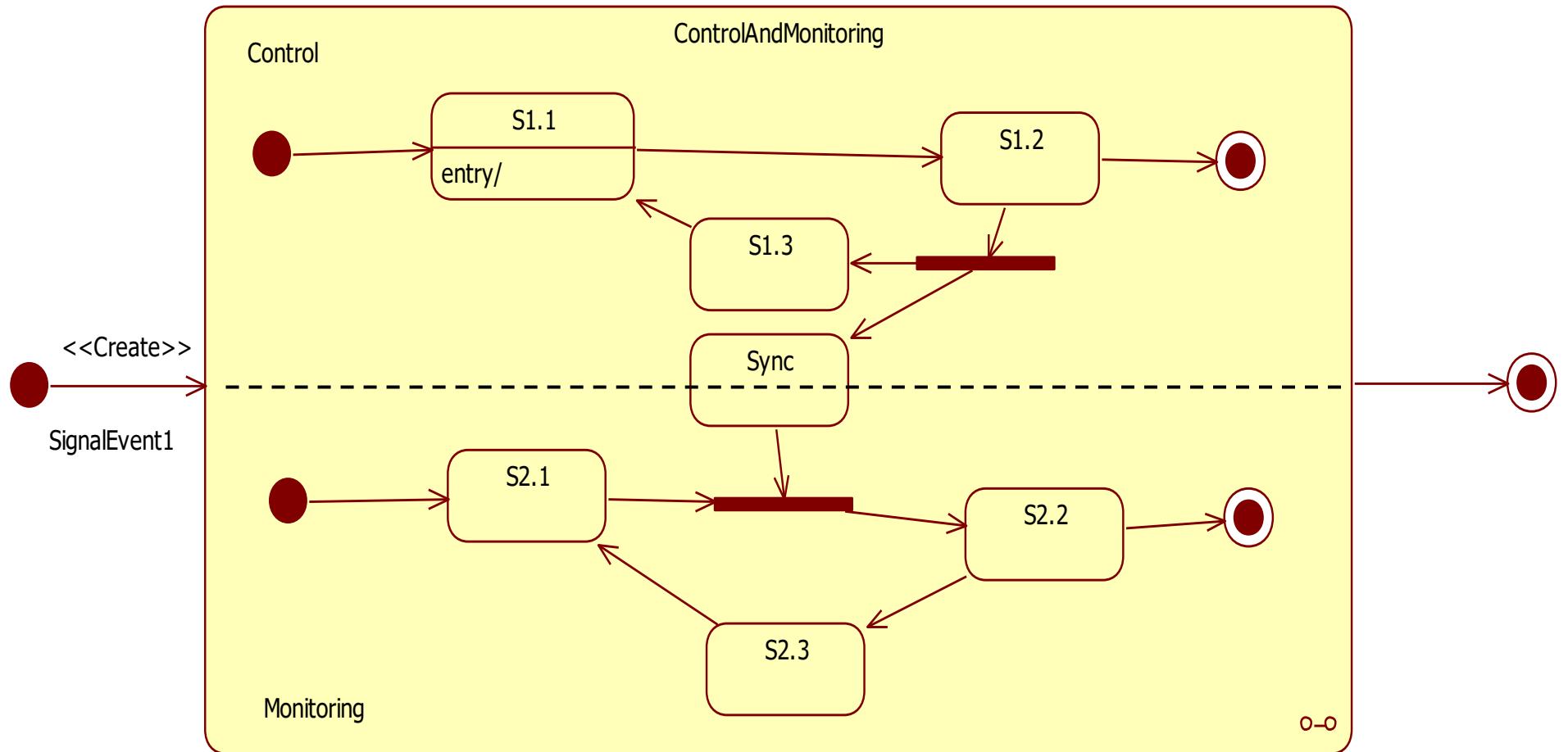


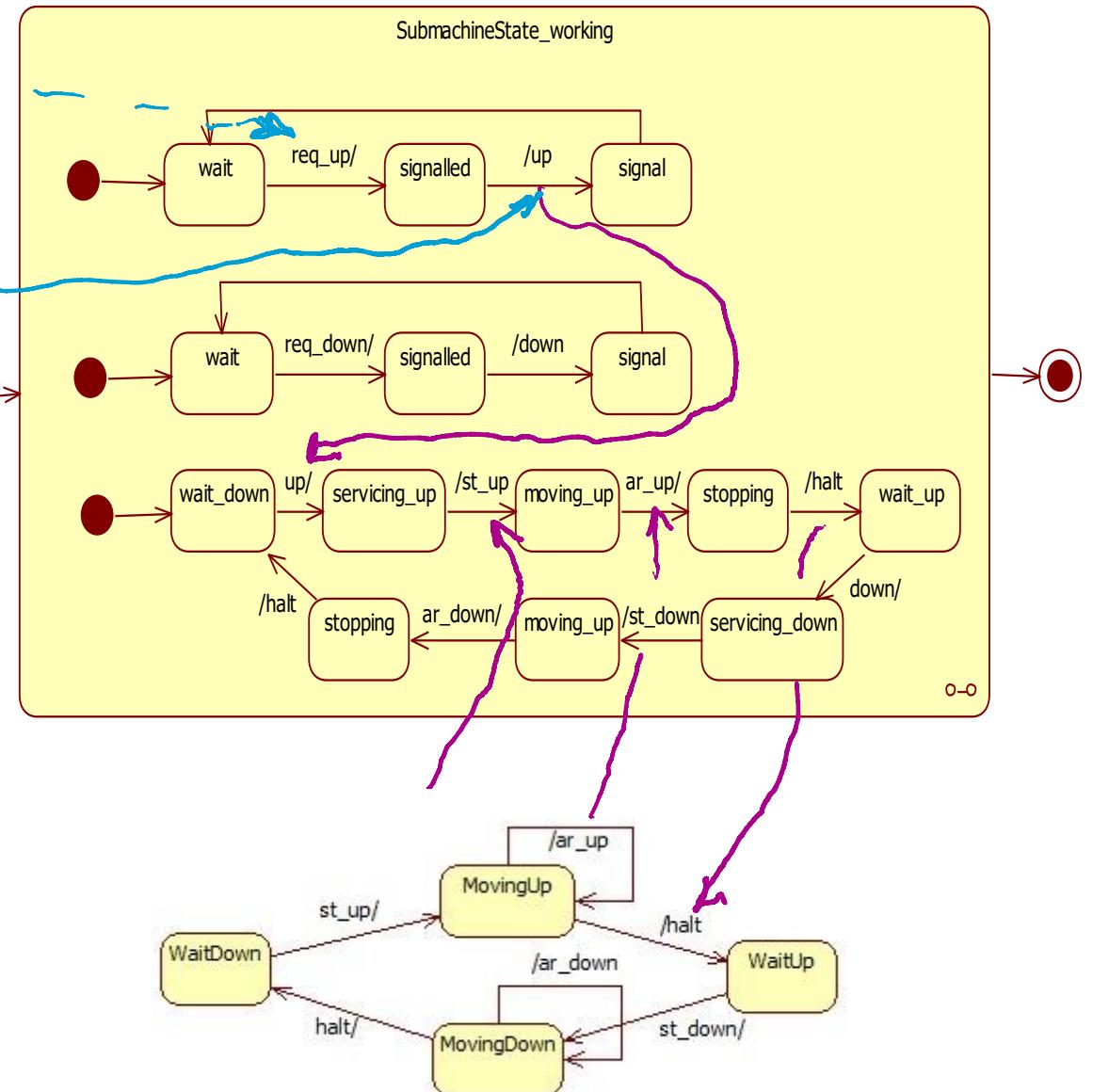
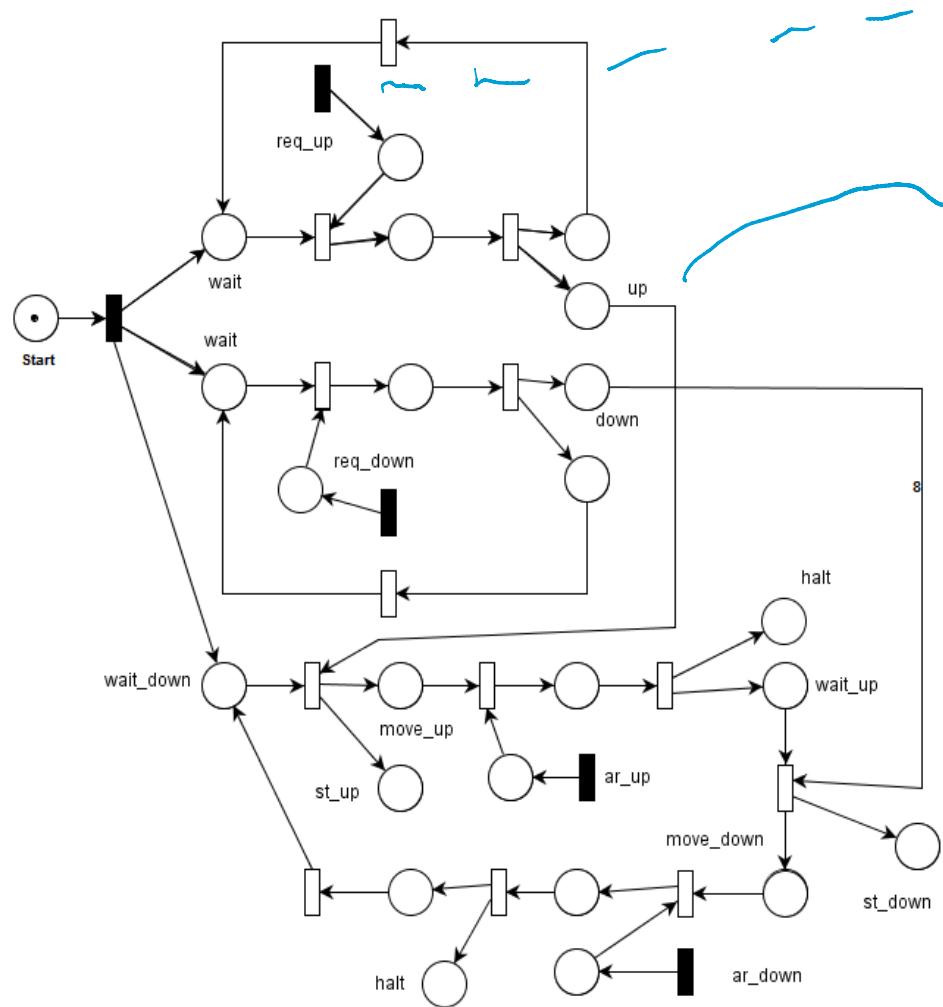
Fig. 1.w. Java mutual synchronization of two threads.

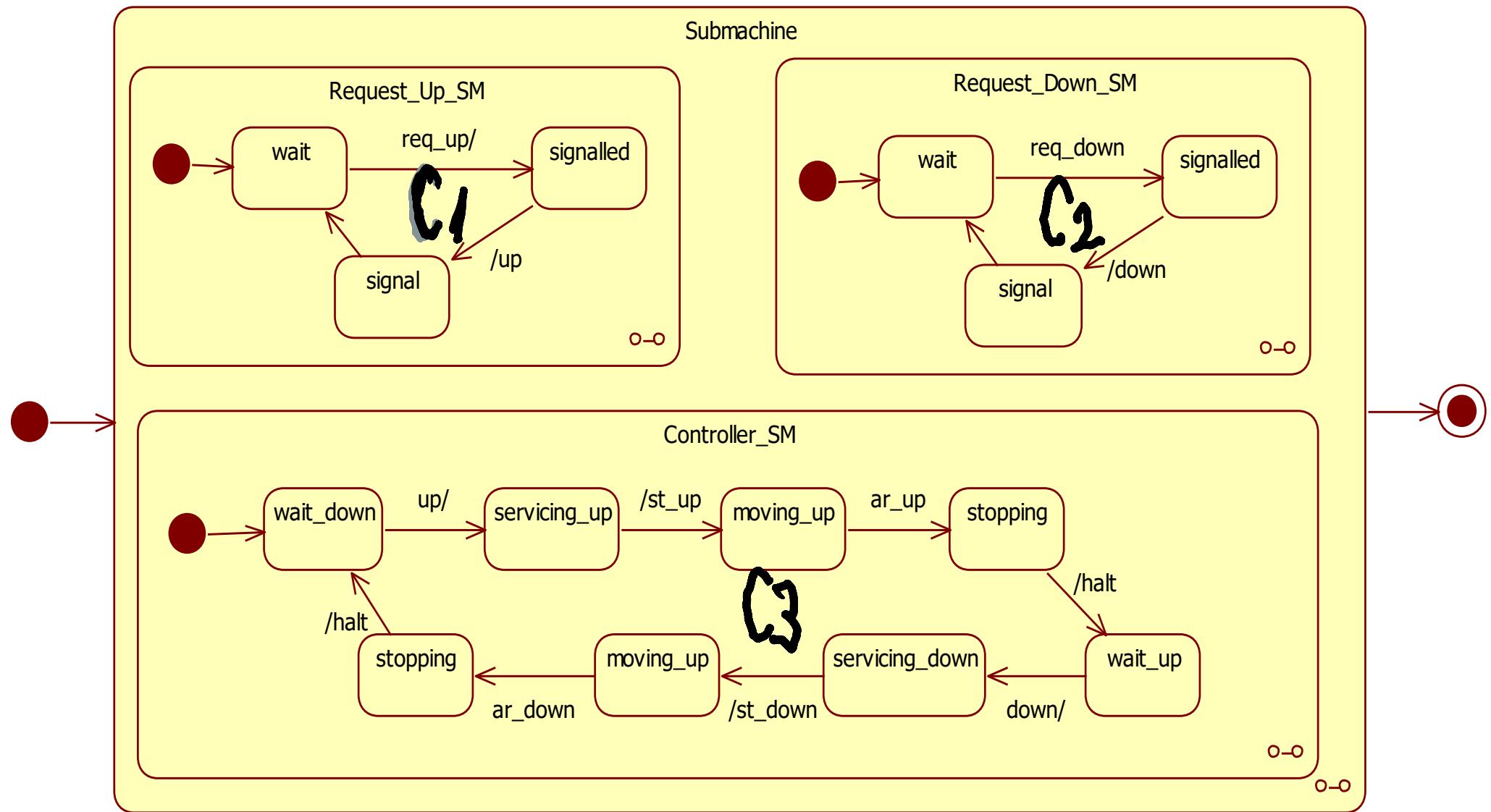


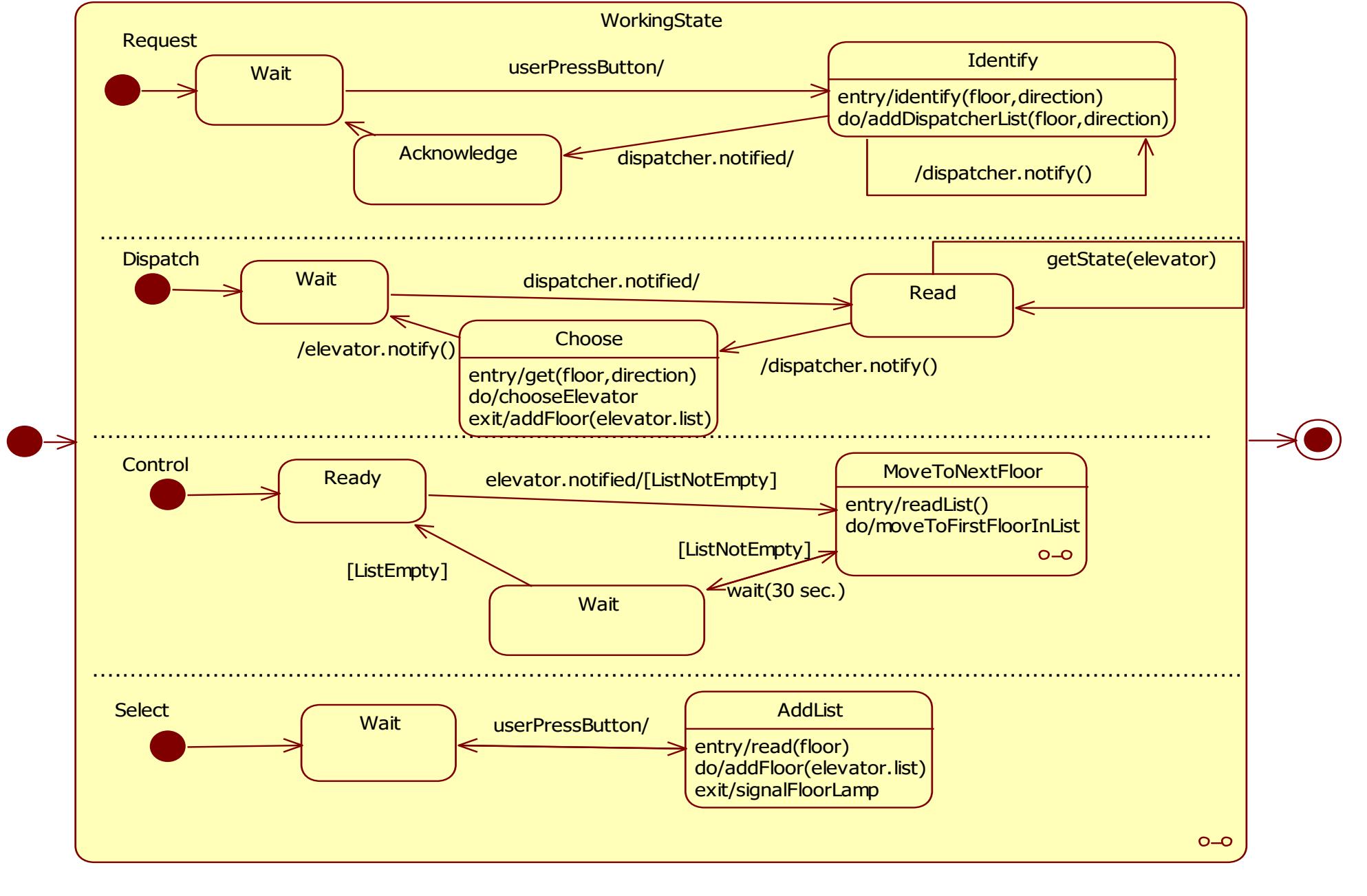
State machine with Sync PseudoState

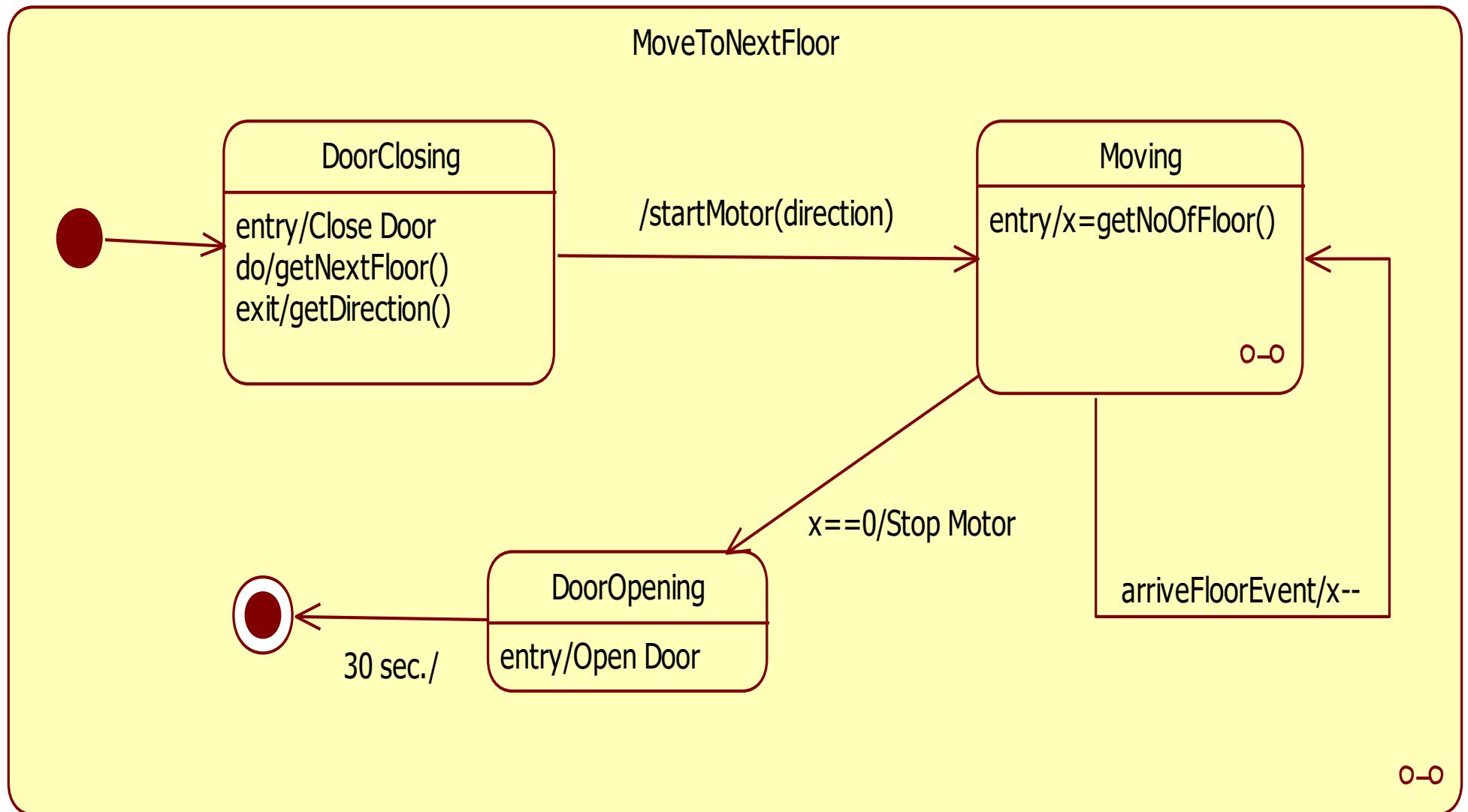
It is similar to an event signal! Some authors recommend the last solution.

Simple elevator system

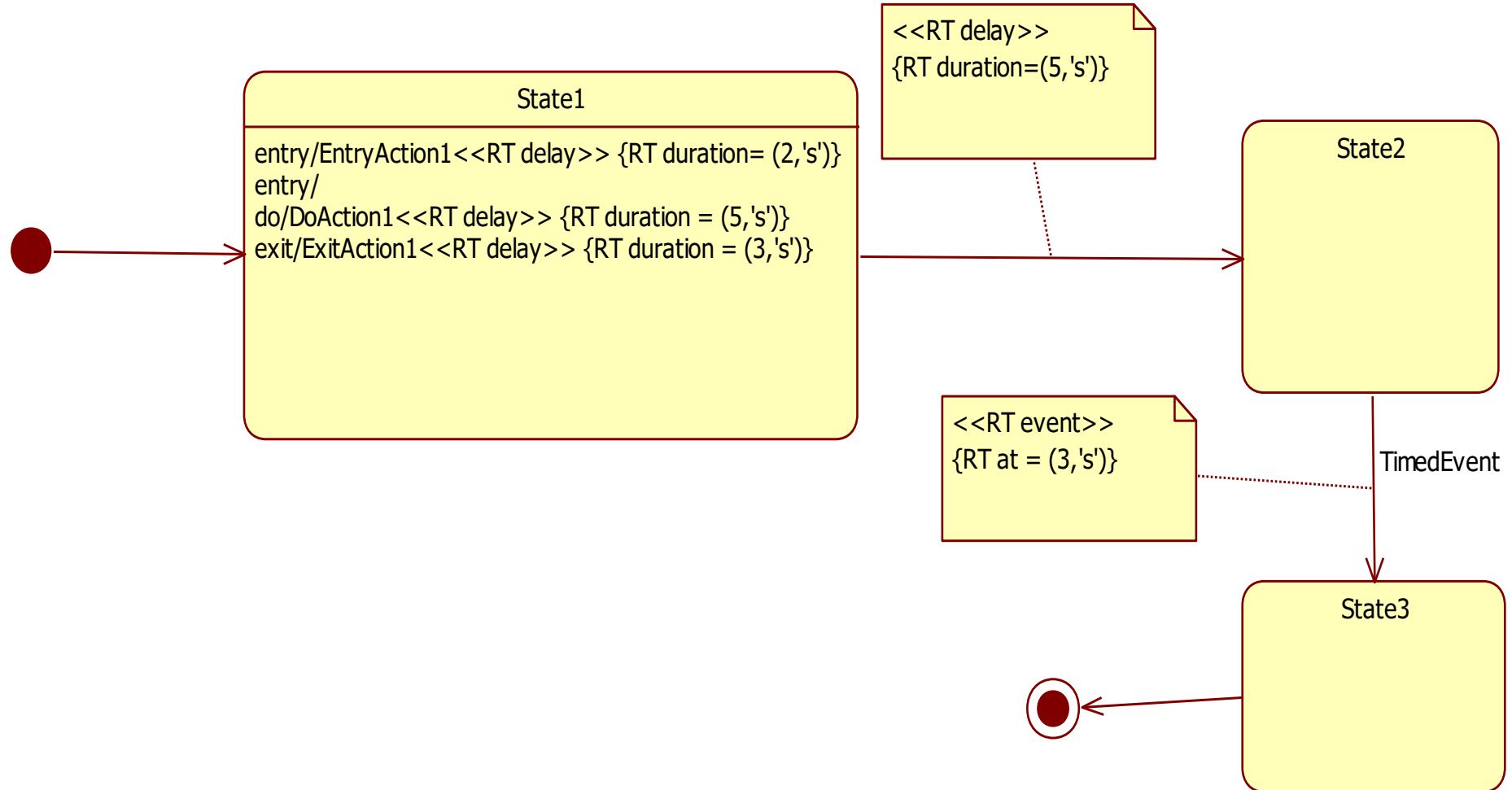








Conclusion: where can be the timing added in SM



3.4. Using UML Real-Time

ROOM = Real-time Object Oriented Modeling
ROOM → UML RT

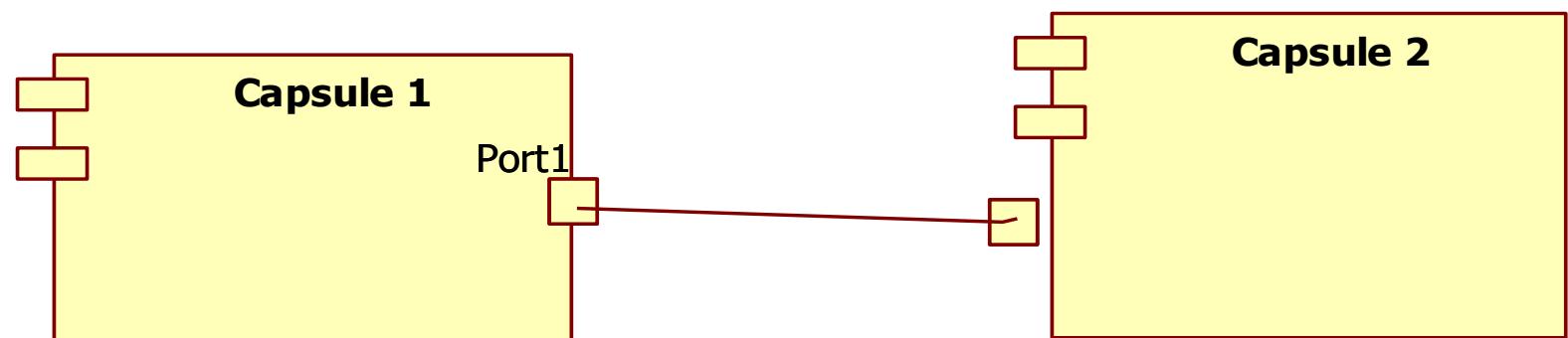
RT UML elements:

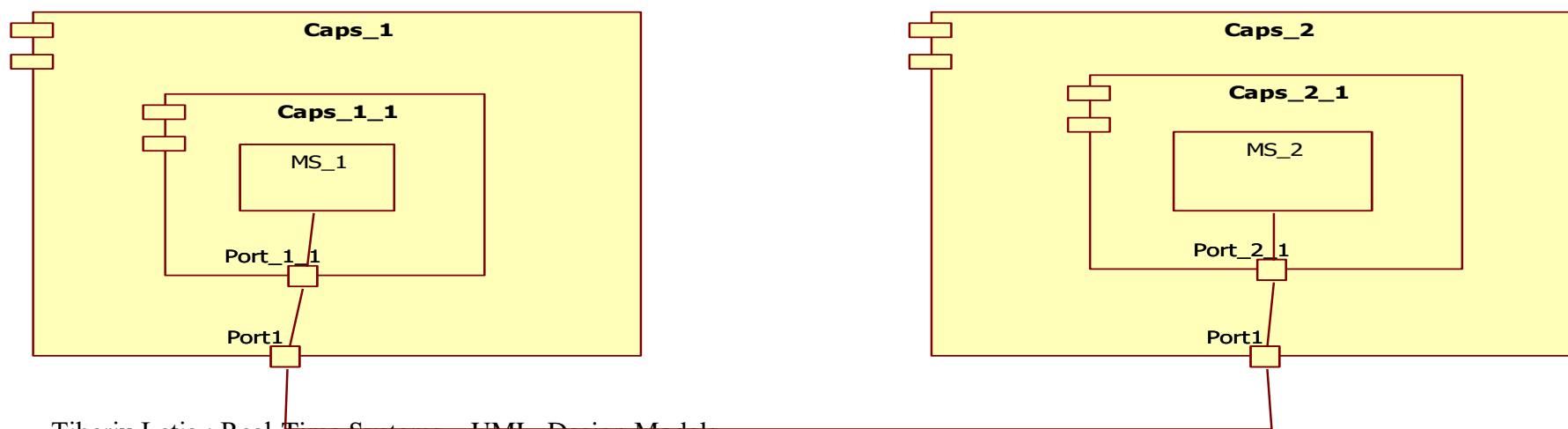
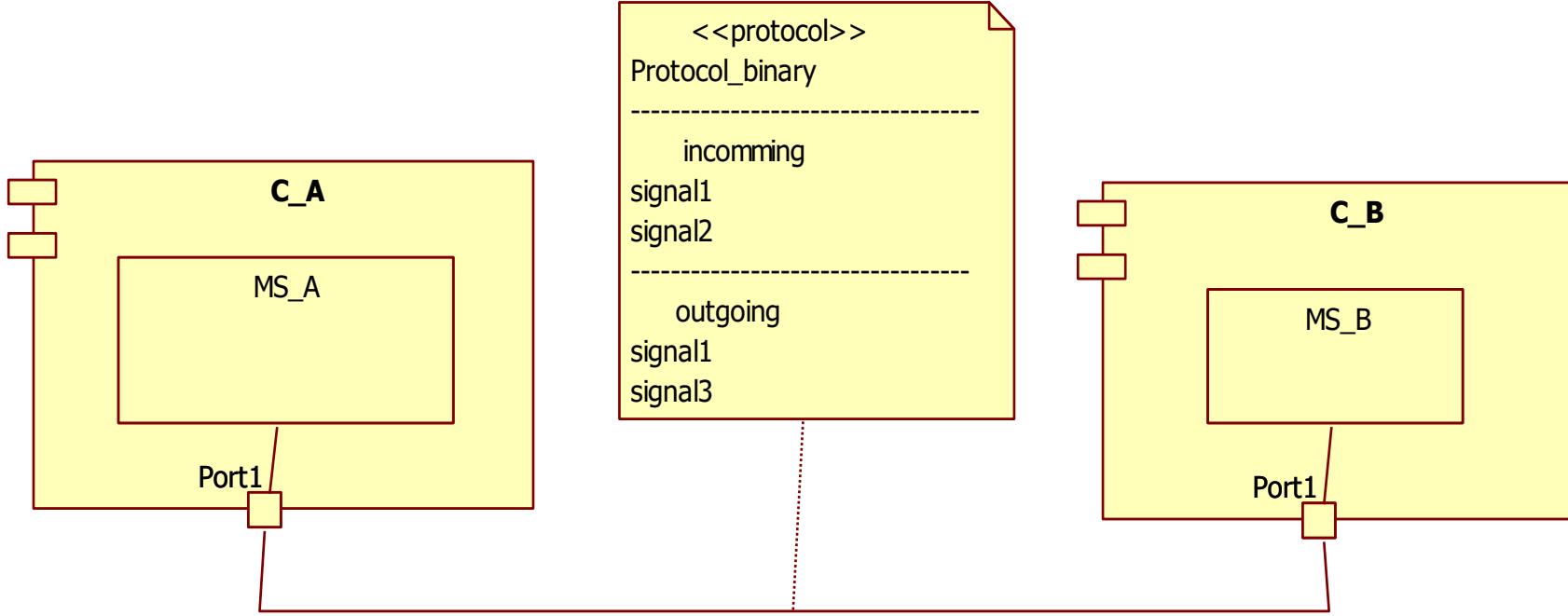
Capsule is an active class that has a skeleton similar to a class, has attributes, methods and additional a *state machine*.

A capsule has its own logical thread and initiates control typically representing an active unit of computation.

Port. A capsule has ports (*relay port* or *end port*).

Protocol. The protocols are rules which define signals that can be exchanged by ports in capsules. They define a set of messages which are *In Signals* or *Out Signals*.





Protocol State Machines (PSMs)

PSMs describe protocols between interfaces and ports.

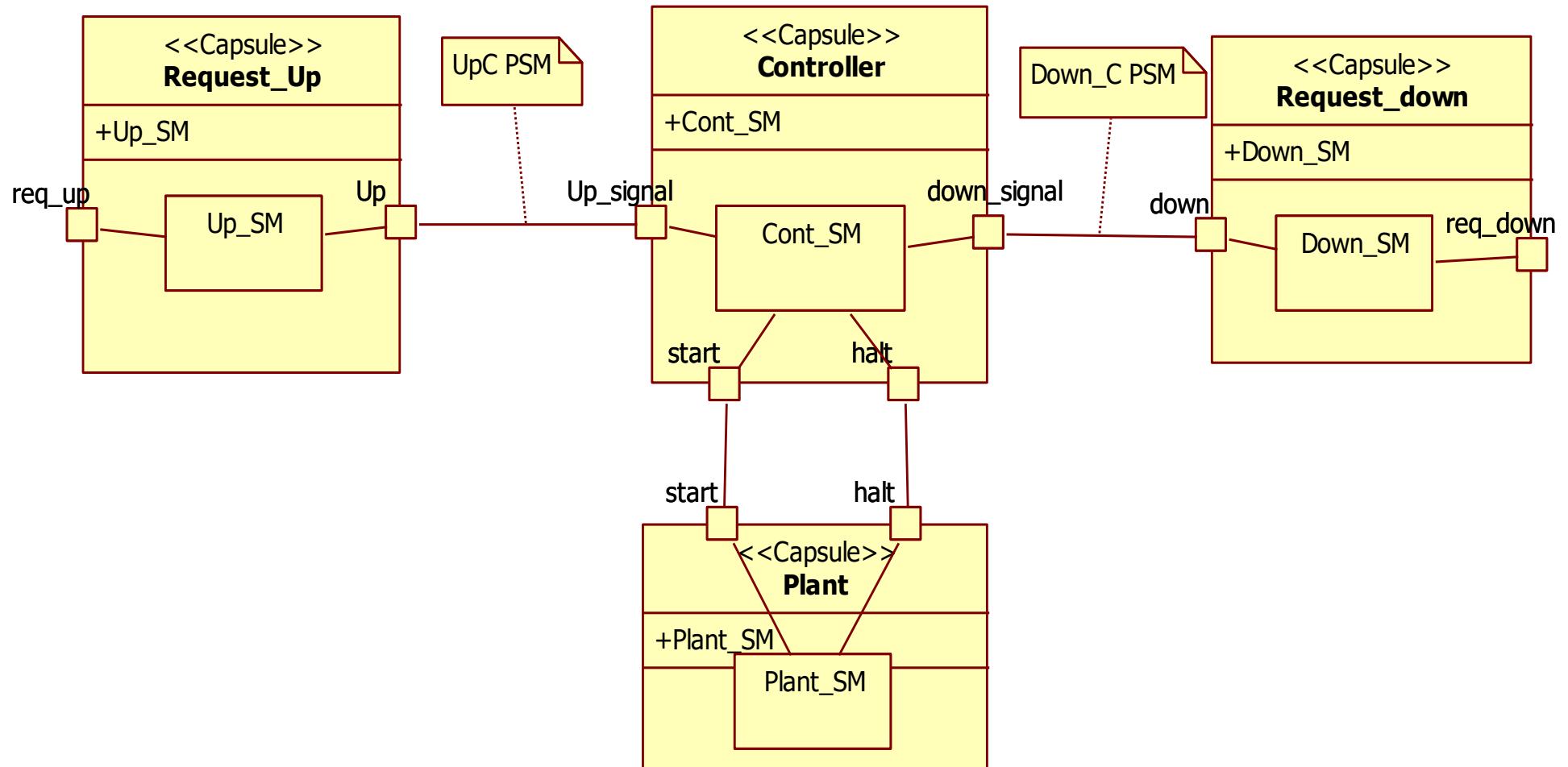
PSMs describe the events and allowed sequences of events in the protocols. They refer to the classifiers.

A PSM defines which operations of a classifier may be called in a given state under specified preconditions. It specifies the pre- and the post-condition for the execution of an operation.

Pre-condition and post-condition are shown on the transitions as guards.

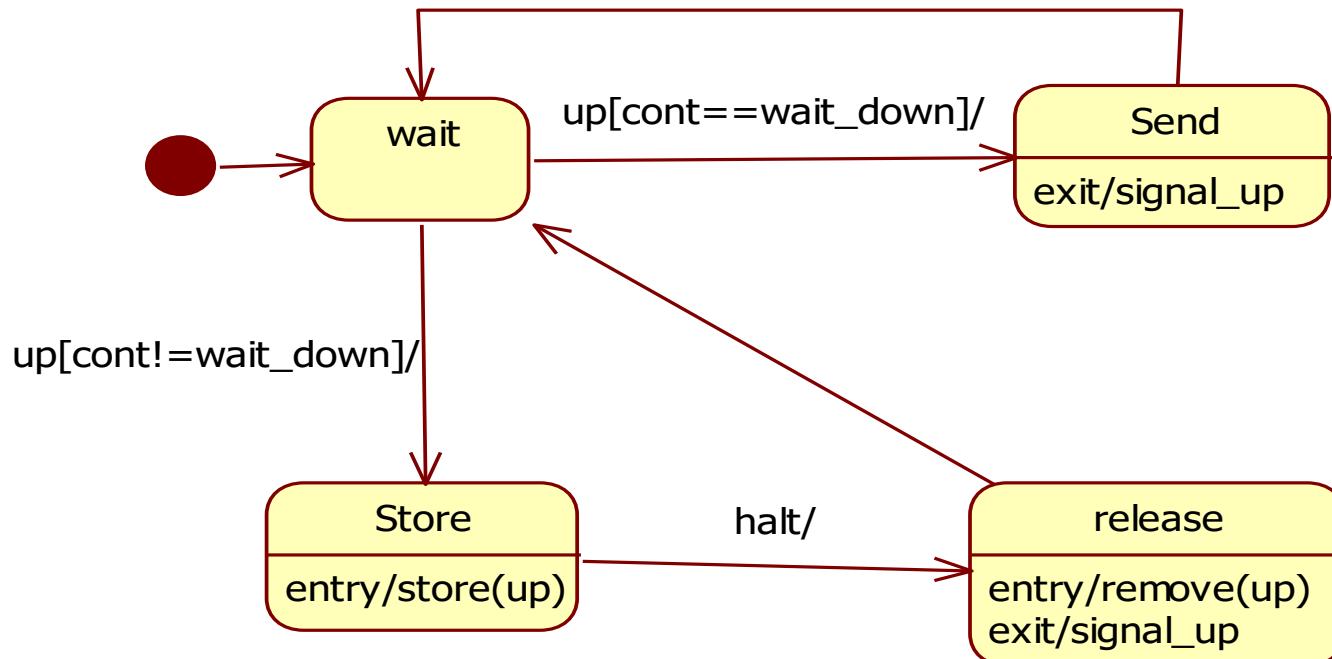
They formalize the protocol of interactions.

Simple elevator system – Capsule diagram



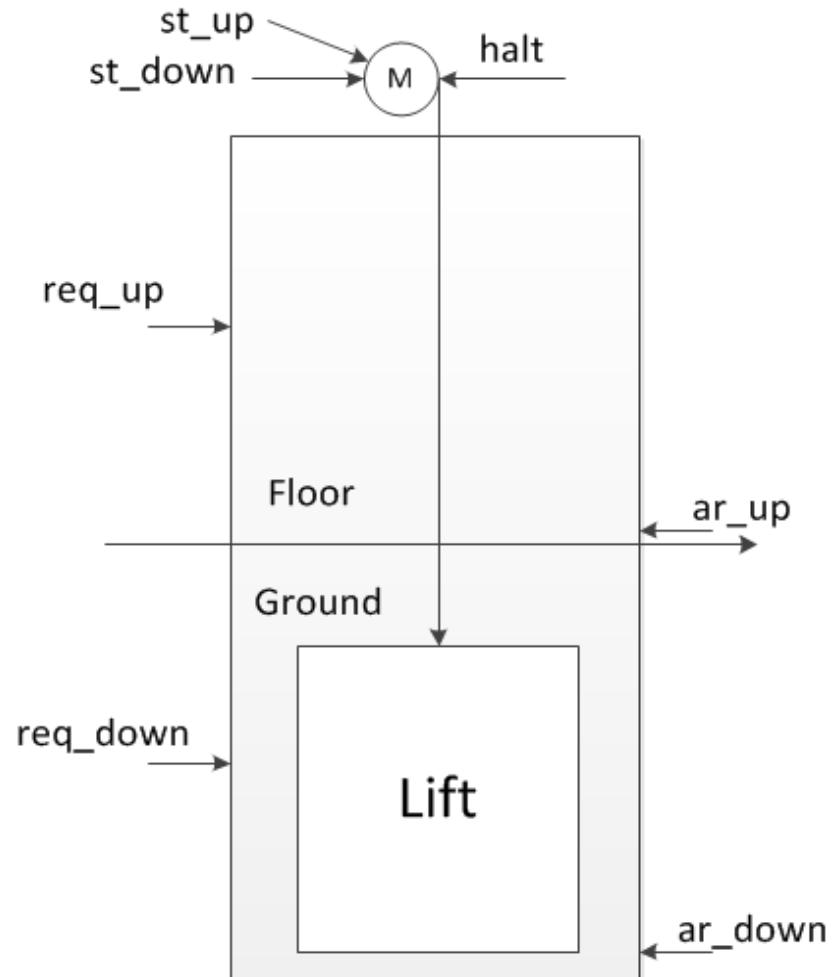
Up_request – controller Protocol State Machine (UpC – PSM)

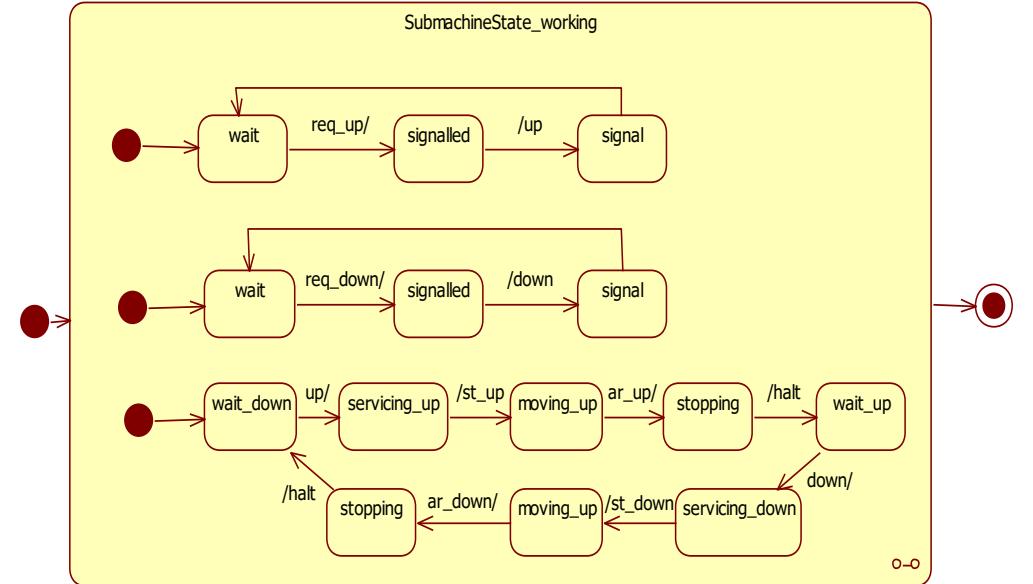
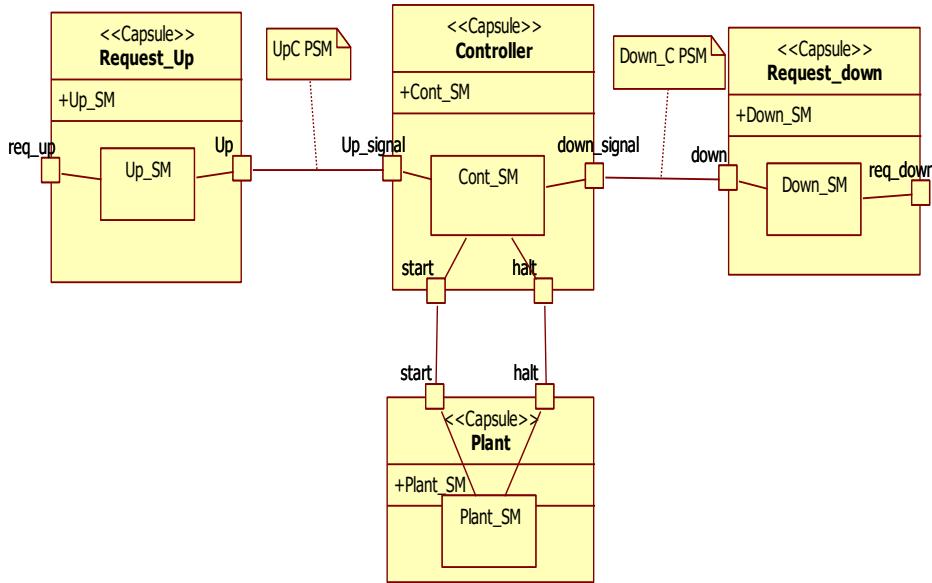
UpC PSM, Controller PSM and DownC PSM can be used for behavior verification.



Homework: Draw:

1. Controller state machine that has to be implemented in *Controller capsule*.
2. Controller – plant PSM
3. Write the significant source code for simple elevator system (RT UML implementation)
4. Compare the RT UML solution with OETPN solution → write the conclusions.





Port interfaces:

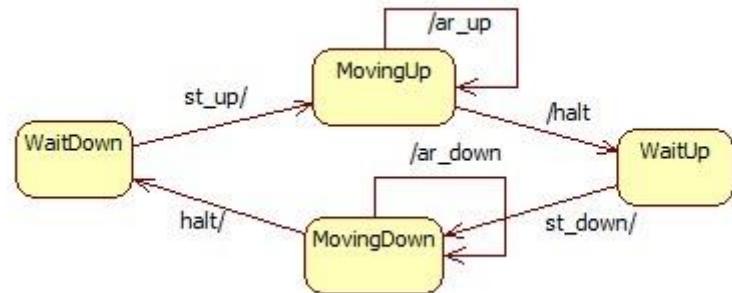
Capsule Request_Up

Port request: incoming method *req_up()*

Port Up: outgoing method *upHandler()*

run() {

```
    while(true) {
        wait(); \\ up request
        .....
    }
}
```



3.5. Task structure diagram

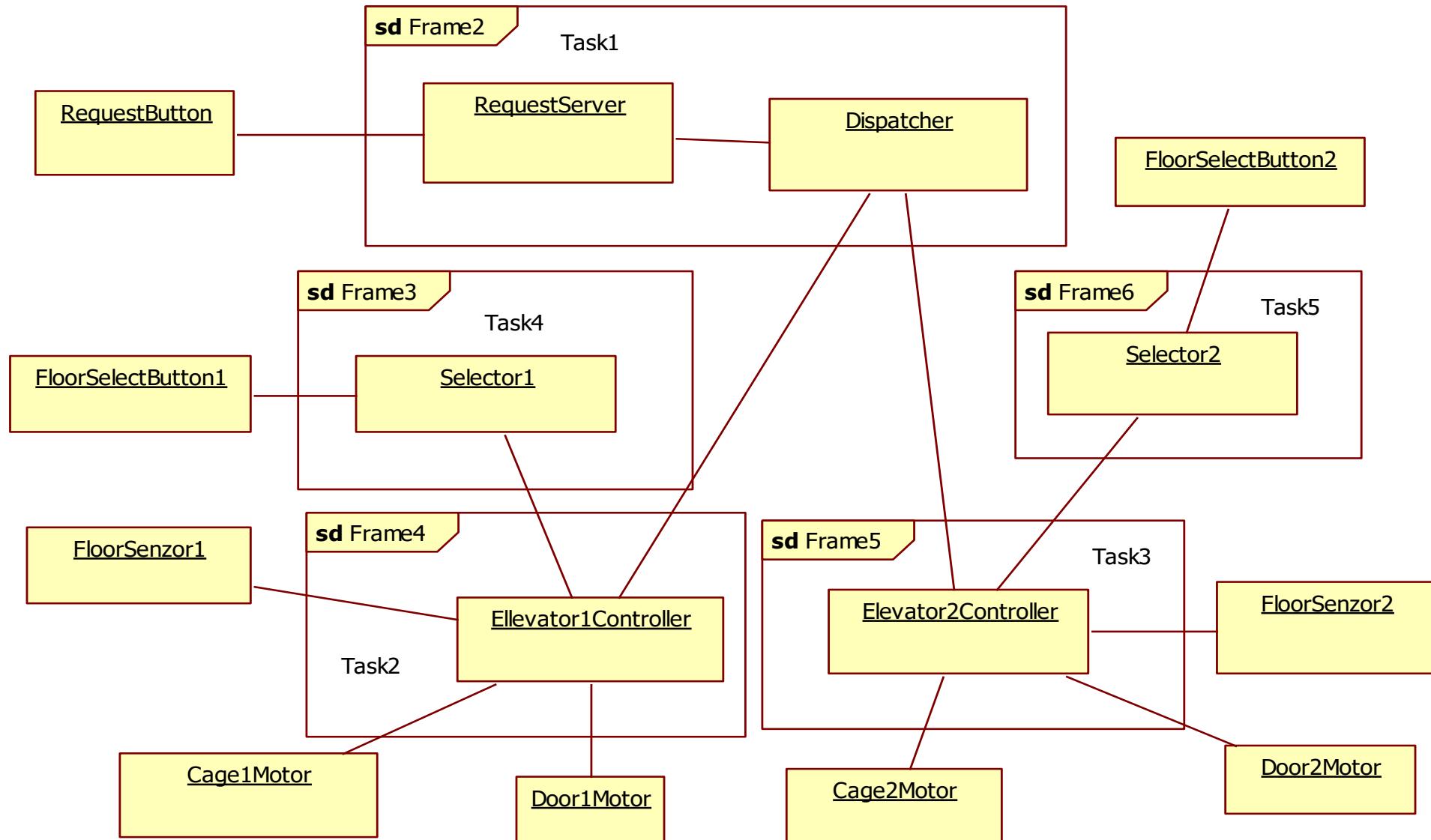
Task Structure on Object Communication Diagram

Tasks are not UML elements.

Goal:

Organize the object communication diagram highlighting the tasks.

Example: complex elevator control system



What is the role of task structure diagram?

Homework:

- 1. Find the actions and the activities performed by each task.**
- 2. Draw the object interactions (method calls and event signals) on the previous diagram. Qualify them as synchronous or asynchronous calls or events.**
- 3. Add timing to task structure diagram.**

*

END

*

4. Implementation of Real-Time Applications

4.1. Introduction

4.2. Methods for Implementation (Synchronous and Asynchronous Approach)

4.3. Concurrency Notions and Concepts

4.4. Real-Time Operation Systems (RTOS)

4.5. Concurrency in Standard Java

4.6. Realtime Java

4.1. Introduction

Development of Real-Time Applications involve:

- Specification
- Software and Hardware Design
- Implementation of Real-Time Applications (RTAs)
 - with Real-Time Operating Systems (RTOS)
 - Priority pre-emption (fixed or dynamic) → Priority based scheduling
 - Interrupt servicing
 - without Real-Time Operating Systems (non RTOS)
 - Time slicing
 - Cyclic execution
 - Real-Time Programming Language (environment)
- Verification
- Testing
- Maintenance

Development Approaches of R-T Applications are classified in:

- ***synchronous approaches*** ↔ involving actions without durations (ignorable relative to deadlines)
- ***asynchronous approaches*** ↔ involving activities with significant (relevant) durations

The specifications of RTAs concern input events and read of input data (from input channels) that involve actions, activities and output reactions concerning output events, send or write of data (through output channels).

Some of them have to fulfill real-time requirements.

The main goal of a R-T implementation is to execute the requested actions and activities such that they meet the R-T constraints.

4.2. Methods for Implementation (Conception: Synchronous and Asynchronous Approach)

There are the models → they have to be implemented.

The controller synthesis leads to the controller model.

From the Real-Time System point of view the controller software design could require:

- the *synchronous approach* – the controller contains only actions without significant durations,
- the *asynchronous approach* – the controller contains activities with relative long durations,
- the *mixed approach*.

A RTA can be implemented by a *single task* or by *multiple tasks* executed by a RTOS or by an Execution Environment (EE). The interaction with plant, user etc. are provided by I/O channels managed by RTOS or EE.

The RTOS or EE provide an interrupt service that signals the application when an event occurred or requested/expected data arrived. This service can be thought as one or more tasks working independently for application. A similar service signals outside the computer (or microcontroller) the events demanded by application, or send data through destination channels.

The durations of sending and receiving information through I/O channels could be not ignorable for fast applications.

EE or the applications have to convert the information such that the links user-application and application-plant be consistent (compatible).

The design outcomes are (considered here) PN, ETPN, OETPN or UML design models.

UML

The implementation of UML design is clear: each state machine (or state chart) requires a task. Some method of I/O channels that have relevant durations and are called by tasks could involve (variable) delays that modify the initial conceived temporal behaviors.

ETPN

The ETPN models have (usually) only actions and calls of I/O channels.

If the durations of I/O communications are ignorable, the ETPN models can be implemented by single tasks. This means the synchronous approach is possible. Partitioning the implementation of this case in multiple tasks has no benefit from the execution point of view. Multiple tasks implementation could have benefits for testing or maintenance.

When the durations of I/O communications are not accepted as ignorable, the ETPN model has to be implemented by multiple tasks. This means the asynchronous approach is needed. The concurrent calls with relevant durations have to be implemented in different tasks. This allows the (RTOS or EE) scheduler to organize the execution for an improved behavior.

OETPN

The implementation of OETPN models can be decided in a similar manner (as ETPN models). If the evolution relations (i.e. guards, mappings and timings) and the calls of I/O channel methods have ignorable durations, the application can be implemented by a single task. This task called *executor* performs the execution of all the transitions (i.e. evolution relations). The concurrent transitions are executed sequentially in the same tic. If there are no conflicts the sequential execution does not change the system behavior.

When the durations of the mentioned activities are not ignorable, the multiple tasks implementation is compulsory (i.e. the asynchronous approach).

The conception of OETPN models is based on some sequences of transitions. An OER-TPN model shows the concurrent sequences of transitions. The partition of these sequences to different tasks should concern not only the implementation reasons, but the testing and the maintenance reasons too.

The OETPN models contain the needed synchronization (precedence, mutual exclusions, etc.) of their sequences of transitions.

OETPN

Recommendations for partitioning the OETPN models in different tasks:

- 1) If two sequences of transitions (i. e. activities with long durations) are concurrent, they should be included in different tasks.
- 2) If the mentioned sequences (implemented in different tasks) are changing information using shared places, these can be implemented by the parent of the tasks or by one of them. Both tasks must have references toward the shared places. **Should be there mutual exclusion used?**
- 3) If a sequence of transitions read (receive) or send (write) data through an I/O channel requiring relative long durations, these activities should be implemented by different tasks. If a task waits for the ending the transfer (sometime unspecified duration), it can exceed its required time synchronization.

Controller model: PN and ETPN

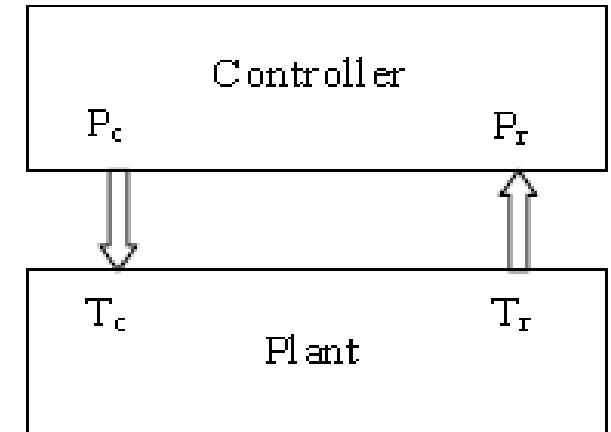
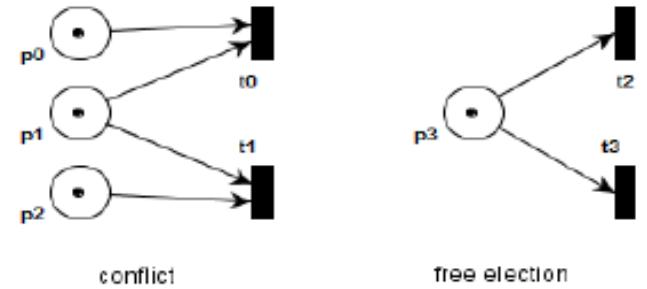
PN models with or without conflicts.

Controller model: Petri Net (PN)

The model is deterministic and fulfills the following *assumptions*:

1. The controller PN model has no conflicts or free elections or
2. If the PN model has conflicts or free elections the following semantics is accepted: the order of transitions chosen for execution is given by their index.
3. The places of the sets P_r and P_c are used exclusive only for input or output operations respectively.

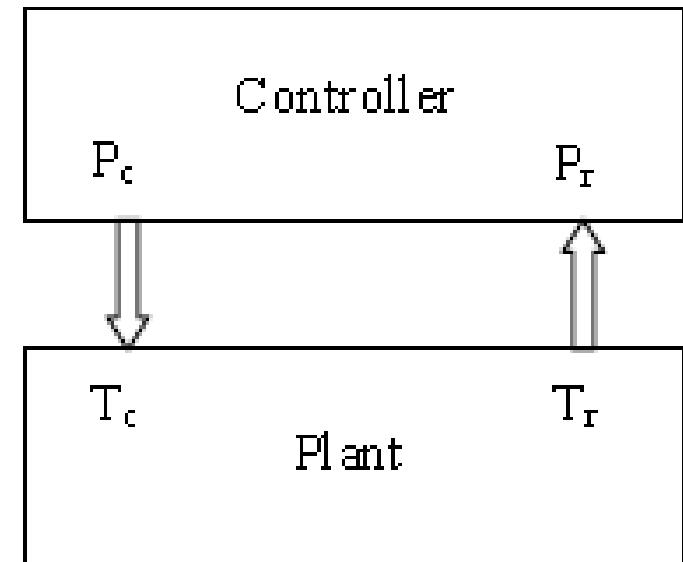
The transitions correspond to actions and they involve no durations or delays.



PN

Notations:

- B is the backward incidence matrix.
- F is the forward incidence matrix.
- B_t is the column t of B .
- F_t is the column t of F .
- M is the marking vector.
- $P_r = \{ p_{r1}, p_{r2}, \dots \}$ is the set of the reaction places.
- $P_c = \{ p_{c1}, p_{c2}, \dots \}$ is the set of the control places.
- T is the set of the controller model transitions.
- T_c is the set of the plant controllable transitions.
- T_r is the set of the plant reactive transitions.
- M_r is the marking vector corresponding to P_r .
- M_c is the marking vector corresponding to P_c .
- n is the number of transitions.



No activities with long durations are involved! → single task (executor).

Controller dynamics:

M_0 – the PN initial state

A transition t is *fireable* or *executable* if $M \geq B_t$

The execution of the transition t involves the modification of the marking according to the relation:

$$M = M - B_t + F_t$$

PN Controller algorithm:

Input: M, B, F, P_r, P_c

for ever do

for all t of T **do**

read M_r //from P_r

fill M_r to M

if t is fireable **then** $M = M - B_t + F_t$

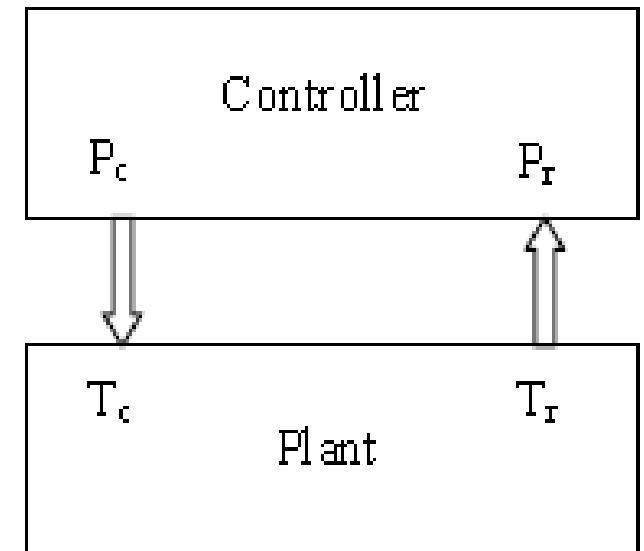
end for

extract M_c from M

write M_c //in P_c

end for

interface:
- input port_r; // $M_r \leftarrow P_r$
- output port_c; // $M_c \rightarrow P_c$



Observations:

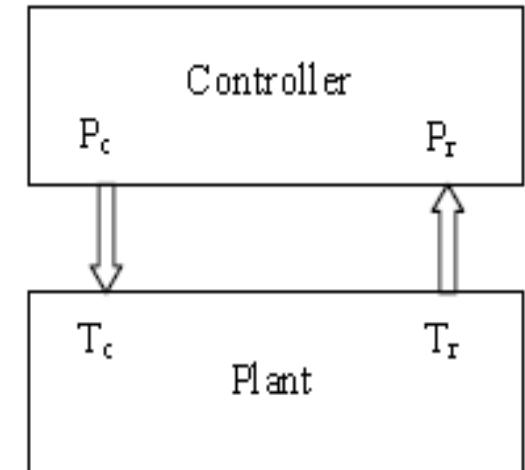
1. No multithreading system is required.
2. The reaction delays are given by:
 - a. the number of transitions,
 - b. input and output channel speed,
 - c. the processor speed.
3. No processor interrupt system is used.
4. No clock is used.
5. The solution can be used for simple applications implemented on FPGAs or microcontrollers.

Controller model: Enhanced Time Petri Net (ETPN)

The controller model is deterministic, and it has timed (delayed) transitions.

The model is deterministic and fulfills the following *assumptions*:

1. The controller PN model has no conflicts or free elections or
2. If the PN model has conflicts or free elections the following semantics are accepted: the order of transitions chosen for execution is given by their index.
3. If more than one timed transition is executable from the same marking, the transition with the shortest delay is first chosen for execution.
4. The system works with reserved tokens. A transition that started the execution cannot be cancelled by another one with a shorter delay.



Notations:

- $D = [d_0, d_1, \dots, d_{n-1}]^T$ is a delay vector with d_i the delay of the transition t_i .
- $\Theta = [\theta_0, \theta_1, \dots, \theta_{n-1}]^T$ a temporal vector
- Inp = P_r ; input channels
- Out = P_c ; output channels

ETPN Controller algorithm:

Input: M, B, F, Pr, P_c, D

Initialization: Θ = [0, 0, ..., 0]^T

for ever do

for all t of T for

read M_r //from P_r

fill M_r to M

if t is fireable then

 M = M - B_t

 θ_i = d_i

end if

for i=0, n-1 do

if θ_i ≥ 0 **then**

 θ_i = θ_i - 1

if (θ_i == 0) **then** M = M + F_i

end if

end for

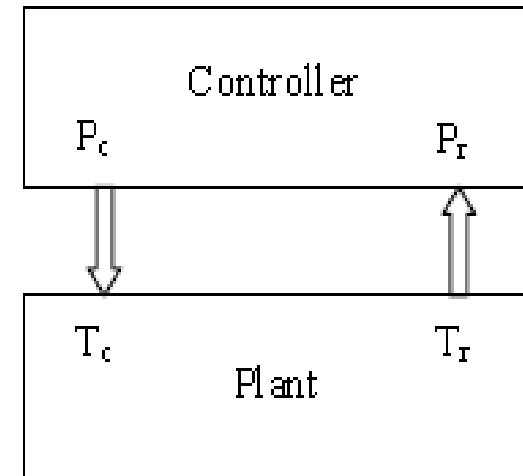
end for

extract M_c from M

write M_c //in P_c

wait(1) // one time unit delay

end for



OETPN synchronous approach → single thread *Executor*

OER-TPN executor algorithm:

Input: $F, M_0, P, T, D, Grd\&Map, Eet, Let,, Out, Inp;$
Initialization: $M = M^0$, $execList = empty$;

repeat

- wait(event);*
- if** event is *tic* **then**

 - * decrease the Delays of the transitions in *execList*;

- else**

 - receive(Inp);*
 - * update M ;

- end if;**
- repeat**

 - for all** $t_i \in T$ **do**

 - if** there is met at least one *grd* in the t_i Grd_i list for $M(p)$, $p \in {}^o t_i$, **then**

 - * move the tokens of ${}^o t_i$ from M to M_t ;
 - * add t_i to *execList*;
 - $Delay[t_i] = eet(t_i)$;

 - end if;**

 - end for;**
 - for all** t_i in *execList* **do**

 - if** ($Delay[t_i]$ is 0) **then**

 - * remove t_i from *execList*;
 - * calculate the tokens for M in t_i^o ;
 - * remove the tokens from M_t for all ${}^o t_i$;
 - * set the tokens in t_i^o and start the active tokens;

 - end if**
 - if** $t_i \in Out$ **then**

 - send(Out);*

 - end if**

 - end for**

until there is no transition that can be executed;

until the time horizon;

END algorithm;

OETPN asynchronous approach → multiple threads executions

Specifications: activities assigned to t1_1, t2_2 needs long durations and mutual exclusion. The solution can be used for a variable number of tasks.

Example 1:

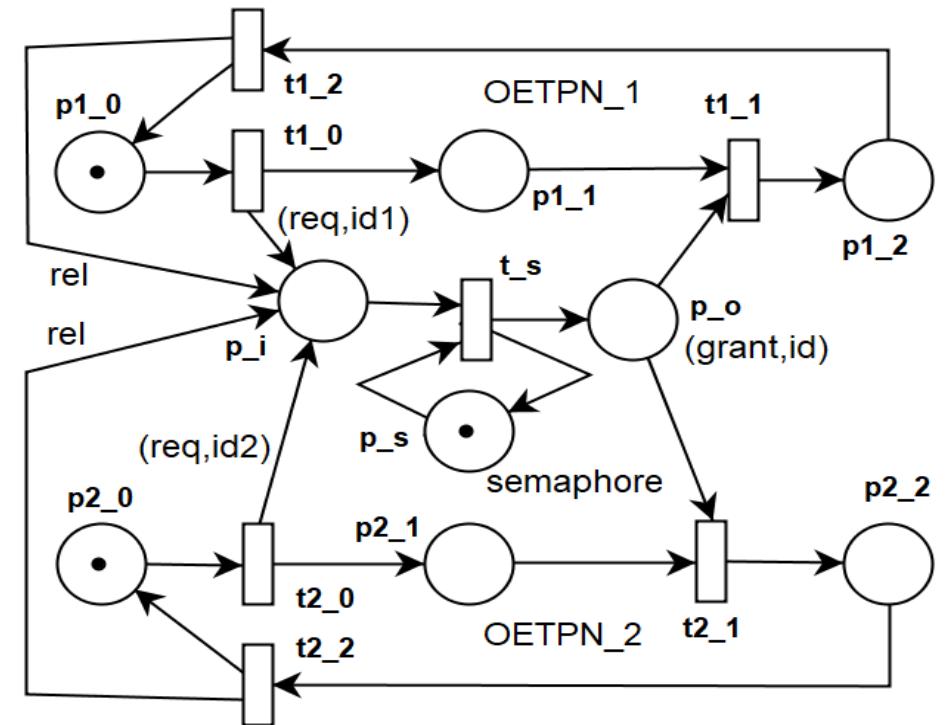
Task_1: $(t1_0 * t1_1) \# t1_2$

Task_2: $(t2_0 * t2_1) \# t2_2$

Sub_OETPN: $p_i - t_s - p_o$

→ Parent rol:

- Create Sub_OETPN
- Create the children OETPN_1 and OETPN_2
- Manage the semaphore.



Question: what would happen if a task acquired the semaphore, and it is broken (stopped) before to release it?

Example 2:

Specification: a user demands the moving of the motor.
The motor has to be stopped when the sensor signals it.

How many tasks are necessary for the attached model implementation if the writing to p_{o1} channel requires a relevant duration? The rest of the mappings (activities) have ignorable durations.

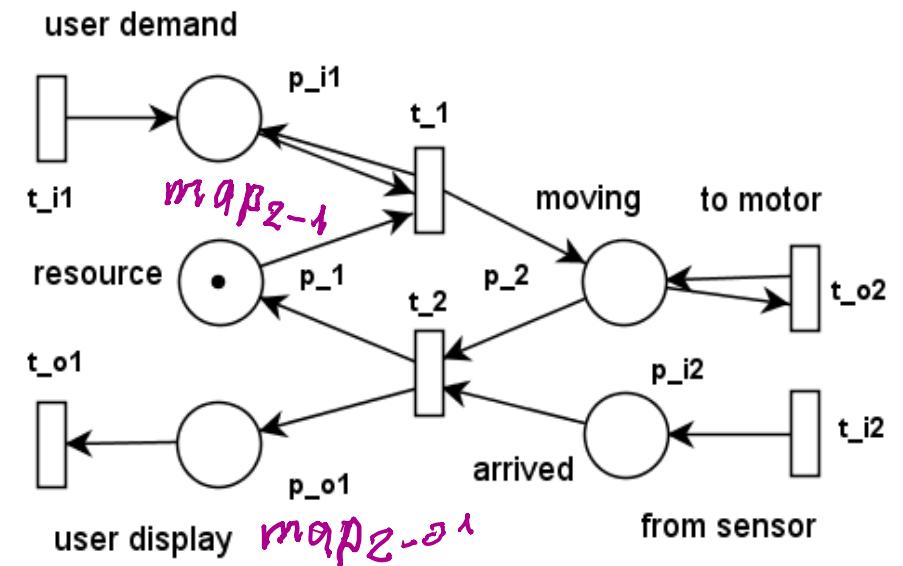
Task_1: $t_1 \# t_2$

Task_2: t_{o2}

Task_3: t_{i1}

Task_4: t_{o1}

Task_5: t_{i2}



Observation: the mappings $map_2\text{-}1$ and $map_2\text{-}o1$ are concurrently executed. The delay avoidance can be obtained by adding Task_4 for implementing the long duration activity.

What are the concurrent activities?

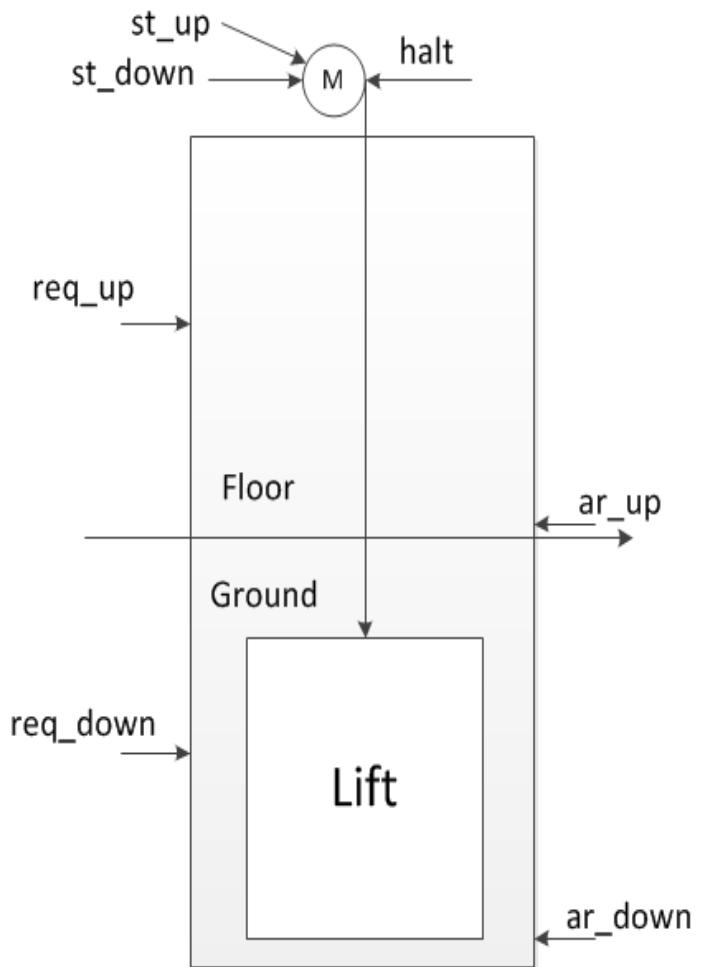
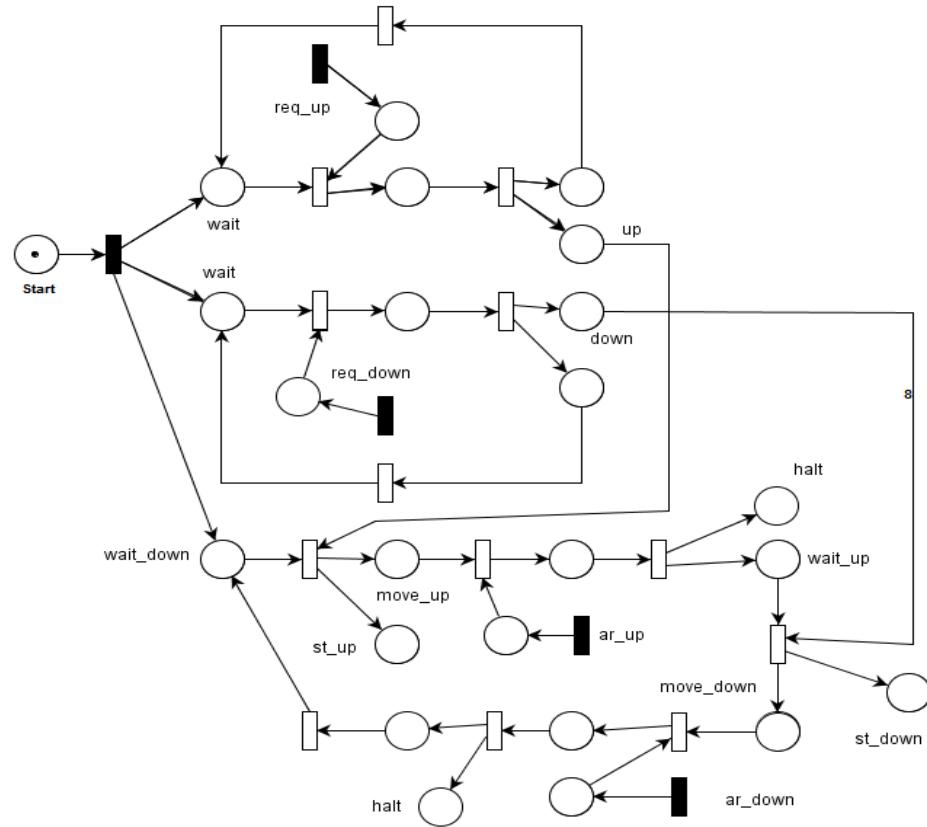
Solutions based on the processor Interrupt System (IS)

Link the transitions t_i from T_r to the IS. Each interrupt routine of t_i contains the instruction: $M(p_i)=1$. The algorithms remain the same except the *read* instruction that has to be removed. The interrupt routine and the algorithm are executed concurrently.

What is more convenient: polling or interrupts?

Polling consists of the interrogation of each input if the information are available.

Homework: Solve the implementation of the simple elevator control problem using interrupts. Use ETPN and OETPN models. Write the tasks sequences!



*

**** END 4.1 & 4.2 ****

*

4. Implementation of Real-Time Applications

4.1. Introduction

4.2. Methods for Implementation (Synchronous and Asynchronous Approach)

4.3. Concurrency Notions and Concepts

4.4. Real-Time Operation Systems

4.5. Concurrency in Standard Java

4.6. Realtime Java

4.3. Concurrency Notions and Concepts

The case when the synchronous approach is not applicable! → Concurrent implementation is compulsory

Concurrency – Concurrent programming

Parallel Paradigms → Processes (*Tasks*) or Threads of execution

Logical concurrency – they must be executed sometimes in the same time

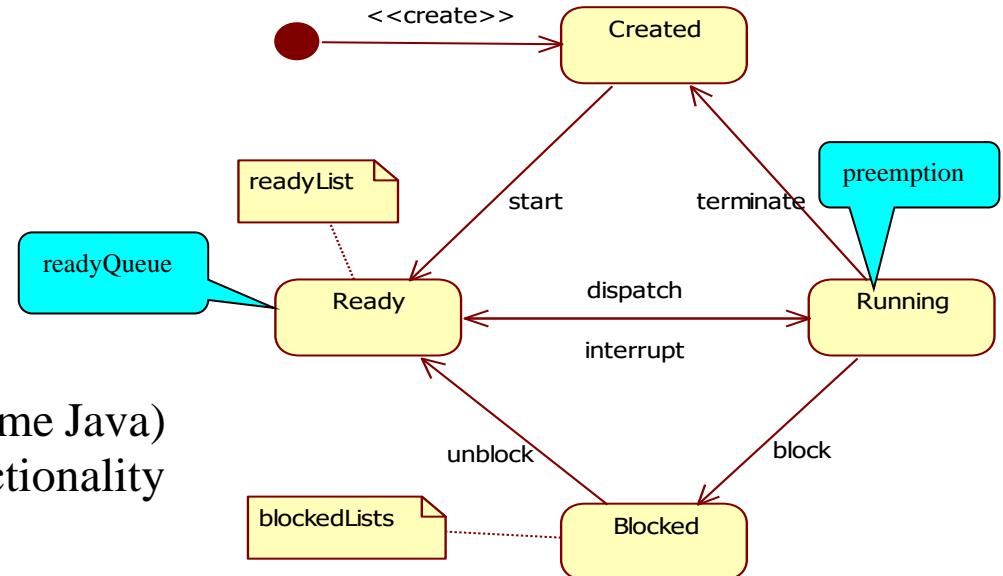
Time sharing concurrency → context switching

Possibilities:

- ***multiprogramming*** = multitasking – processes or tasks share a single processor; the processes multiplex their execution on a single processor
- ***multiprocessing*** – multiprocessor with shared memory; the processes multiplex their execution on a multiprocessor system with tightly coupled processors
- ***distributed processing*** – multiple processor each with local memory; the processes multiplex their execution on several central processor unit system connected through a network ← message communication

Approaches of RTAs Implementation:

- with RTOS
 - Time slicing
 - Priority pre-emption (fixed or dynamic)
 - → Priority based *scheduling*
 - Interrupt servicing
- without RTOS
 - Cyclic executive
 - RT programming languages (e.g. Ada, Realtime Java)
 - The language itself provides the functionality required by a RTA



Time slicing - Each task gets the processor for a defined period of time.

This kind of operating systems is applied to: soft, slow (systems) RTAs with few tasks.

The solution is inflexible, inefficient, and not feasible for complex systems.

Why are the threads blocked?

- wait for time (timing)
- wait for external events (ex.: I/O operations, event synchronizations)
- wait for internal events (ex.: semaphore acquire/release, lock/unlock; mutual exclusions; signal, wait-notify)

4.4. Real-Time Operation System (RTOS)

- sequential languages (like C) with primitives (uninterrupted procedures)
- RT kernel for process handling
- The ***kernel*** of a RTOS is an "abstraction layer" that hides from application software the hardware details of the processor (or set of processors) upon which the application software will run.
- The main difference between general-computing operating systems and real-time operating systems consists in the deterministic timing behavior of the latest.
- Deterministic timing means that operating system services consume only known and expected amounts of time.
- Non-real-time operating systems are usually quite non-deterministic. Their services can involve random delays into application software and thus cause slow and variable responsiveness of an application at unexpected times.

Multi-Core Architecture: involves the use of multiple lower-cost processors instead of cost-expensive high-performance processor. Some authors name them logical processor.

Monolithic Kernels: all functionality provided by operating systems is achieved within kernel itself.

Microkernels: it reduces the services provided by kernel including all services which are not essentially necessary into *user space* as isolated processes (e.g. dispatcher, scheduler, memory manager, network driver etc.)

Task scheduling

- a) priority based scheduling
- b) interrupt based scheduling
- c) cyclic execution

a) Priority based scheduling → preemptive or non-preemptive systems

Tasks are released in priority order and run until they are blocked or pre-empted. The approach involves:

- each task has associated a priority (number)
- provides timelines and efficient processor utilization
- unbounded priority inversion can appear

Priority inversion is the scenario where a low priority process holds a shared resource that is required by a high priority task. The higher priority process is blocked until the lower priority process releases the resource.

→ inversion of the priorities.

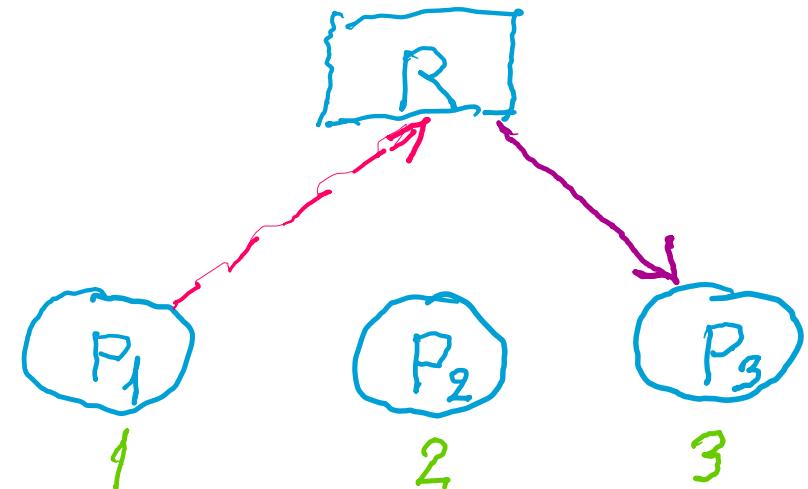
If another process with the priority between the precedents two that doesn't need any resource becomes executable (enters in the readyQueue), it is executed before the higher priority process.

E.g. of priority inversion:

A set of three processes: P₁, P₂ and P₃ with the priorities 1, 2 and 3 respectively (1 is the highest) have to be executed. P₃ holds a resource R that is required by P₁. P₃ is executed before P₁. P₂ that doesn't depend on any resource is executed also before P₁ because it preempts P₃. So, P₁ is executed the last.

P_3	
P_2	
P_3	
P_1	

	P_3
	P_4
	P_2
	P_3



b) Interrupt based scheduling

- Tasks are activated by interrupt service requests (IRQ)— they are released by arrival of corresponding IRQ.
- It can be used with or without RTOS.
- It is suitable for small and fast systems.

c) Cyclic executives

- Each task runs in sequence on the processor to completion or periodically.
- Benefits
 - simplistic: no RTOS, no mutual exclusion, etc.
 - deterministic: no independent running threads
- Problems
 - poor responsiveness, especially with many tasks
 - failure prone
 - rigid and not easily modified.

Process = Task

It is a sequential flow of control within the systems that are executed independently.

Processes can share resources:

- processor (s)
- memories (zones)
- data structures
- critical sections
- I/O channels
- Peripherals etc.

Processes can have priorities dependent on

- rate
- deadline
- criticality

Priorities:

- fixed
- dynamic

Processes can be executed:

- periodically
- episodically

Advantages of concurrent processes

- Reflect the natural parallelism from the problem domain,
- Increase the system reactivity,
- Greater scheduling flexibility – critical process with hard deadlines can have higher priority.

Disadvantages of concurrent processing

- Often the identification of processes and the scheduling to meet the timing requirements is difficult.
- Context switching involves a hidden performance cost.
- Increase the system complexity.
- Difficult to debug non-deterministic systems.
- RTOSs have higher cost than non RTOS.

Fixed-time task switching

The time it takes to do task switching is of interest when evaluating an operating system.

A general-computing (non-preemptive) operating system might do task switching only at timer tick times, which might for example be ten milliseconds apart. Then if the need for a task switch arises anywhere within a 10-millisecond timeframe, the actual task switch would occur only at the end of the current 10-millisecond period. Such a delay would be unacceptable in most real-time embedded systems.

A preemptive task scheduler may need to search through arrays of tasks to determine which task should be made to run next. If there are more tasks to search through, the search will take longer. Such searches are often done by general-computing operating systems, thus making them non-deterministic.

Some RTOS avoid such searches by using incrementally updated tables that allow the task scheduler to identify the task that should run next in a rapid fixed-time fashion.

Requirements for R-T Operating systems

- Timers - the ability to set and read high resolution internal timers,
- Priority scheduling - A priority-based preemptive scheduling - support for scheduling of real-time processes;
- user dynamic resources management,
- Shared memory -The ability to map common physical space into independent process-specific virtual spaces,
- Real-time files - The ability to create and access files with deterministic performance.
- Semaphores - Efficient synchronization primitives (acquire – P & release - V)
- Inter-process communication - Synchronous and asynchronous message-passing capabilities with facilities for flow and resource control,
- Asynchronous event notification - A mechanism which provides queuing capabilities, deterministic delivery, and minimal data passing,
- Process memory locking - The ability to guarantee memory-residence for sections of a process virtual address space,
- Asynchronous I/O - The ability to overlap applications processing and I/O operations initiated by the application.
- guaranteed interrupt response,
- mechanisms for inter-process cooperation,
- high speed data acquisition,
- I/O support – I/O Synchronized - The ability to establish assurance of I/O completion time at different logical levels of completion.

Specifications are provided by:

- ***THE AMERICAN NATIONAL STANDARD ORGANIZATION (ANSI)***
- ***THE INTERNATIONAL STANDARD ORGANIZATION (ISO)***
- ***THE INSTITUTE OF ELECTRONIC AND ELECTRICAL ENGINEERS (IEEE)***
- **UNIX → THE PORTABLE OPERATION INTERFACE FOR COMPUTER ENVIRONMENTS (POSIX).** *THE IEEE STANDARDS COMMITTEE 1003 . → POSIX 1003.4 Functions*

Memory models

Processes may have:

- shared address space
 - light-weight process or thread
 - shared variables and code
 - process = procedure (Modula 2) or run method (in Java)
- separate address space
 - independent of each other
 - message – passing.

Processes vs. Threads (UNIX)

The processes are protected against each other. A process cannot disturb another process execution. The threads are not protected from each other.

Thread creation, destruction and management are cheaper than process creation, destruction and management. The creation of a thread is eight times shorter (cheaper) than a process creation.

The switching of the execution from a thread to another is much shorter than from a process to another.

Context Switches

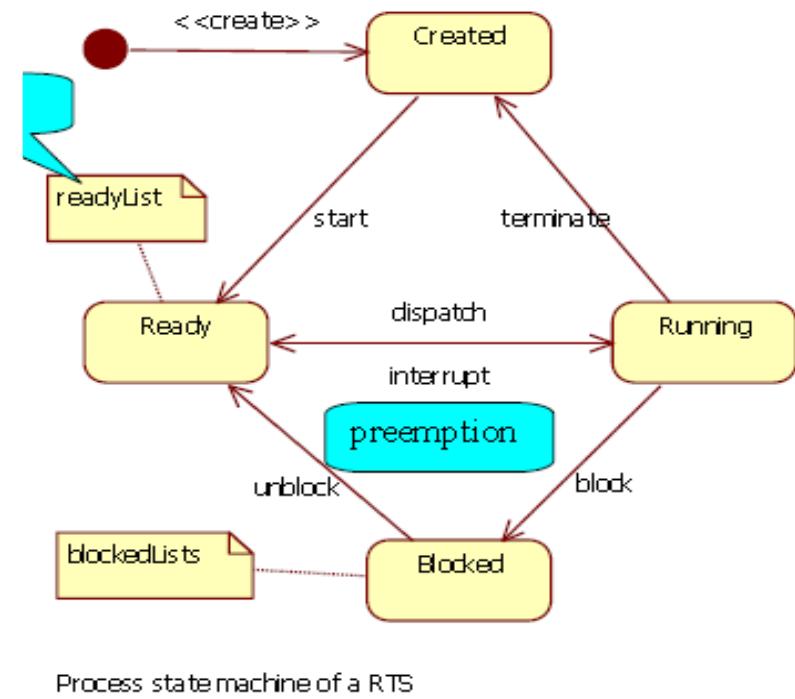
A **context switch** takes place when the system changes the running process (thread). The context of the previously running process is stored and the context of the process to run next is restored.

It is initiated by a call to Scheduler

- directly by a running procedure
- from within an interrupt handler

It happens:

- when the running process voluntarily releases the CPU
- when the running process performs an operation that may cause a blocked process to become ready
- when an interrupt has occurred which may have caused a blocked process to become ready



Priorities

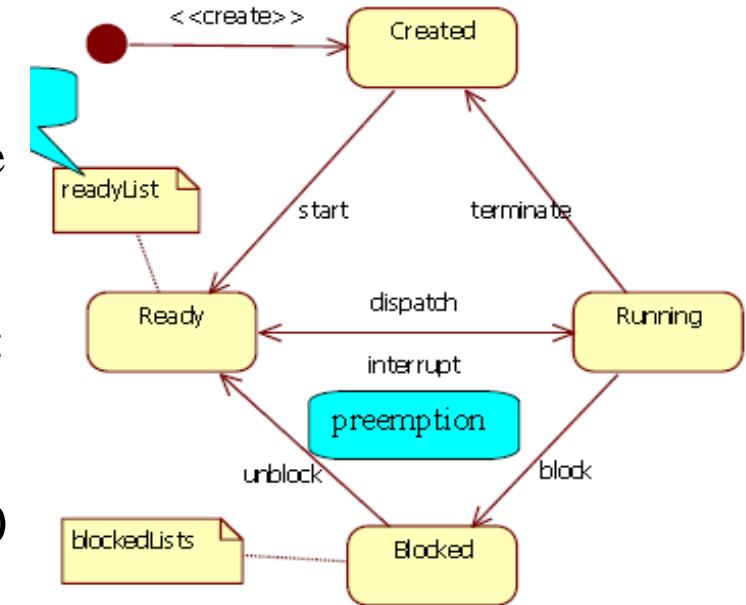
Each process has assigned a number that reflects the importance of its time demands.

UNIX: Low priority number = high priority. Priority range: 1 – max integer

Java: High priority number = high priority. Priority range: 10 – 1; (38 – 1 ? ↪ Realtime Java)

Using priority-based scheduling, processes that are **Ready** to execute are stored in ReadyQueue according to their priority.

Scheduler can change the order of task execution considering the task priorities and their deadlines.



Process state machine of a RTS

Priority implementation

A process (thread) insertion or removal in/from the ReadyQueue may lead to context switch.

If the change results in a new process being the first in ReadyQueue, a context switch takes place.

- ***preemptive scheduling*** – the context switch takes place immediately (the running process is preempted)
- ***non-preemptive scheduling*** – the running process continues until it voluntarily releases the CPU, then the context switch takes place
- scheduling based on ***preemption points*** – the running process continues until it reaches an pre-emption point, then the context switch takes place

Assigning priorities (1)

Non-trivial & Global knowledge

Two ways to assign priorities:

- ad hoc rules
 - important time requirements → high priority
 - time consuming computations → low priority
 - conflicts, no guarantees
- scheduling theory

Often it is easier to assign deadlines than to assign priorities.

Assigning priorities (2)

With priority-based scheduling, the priority of a process is fixed. → *Fixed priority scheduling*

In the real-time system is easier to define the deadline of the processes and then to decide which process should be executed first. → *Earliest Deadline First (EDF) scheduling*

The deadline can be viewed as a *dynamic priority* that changes as time proceeds.

Process state diagram

All the processes pass through I to Z

I – initial state

R – runnable

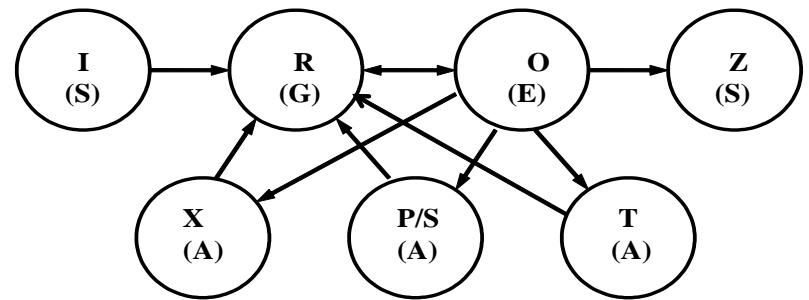
O – operating – kernel running

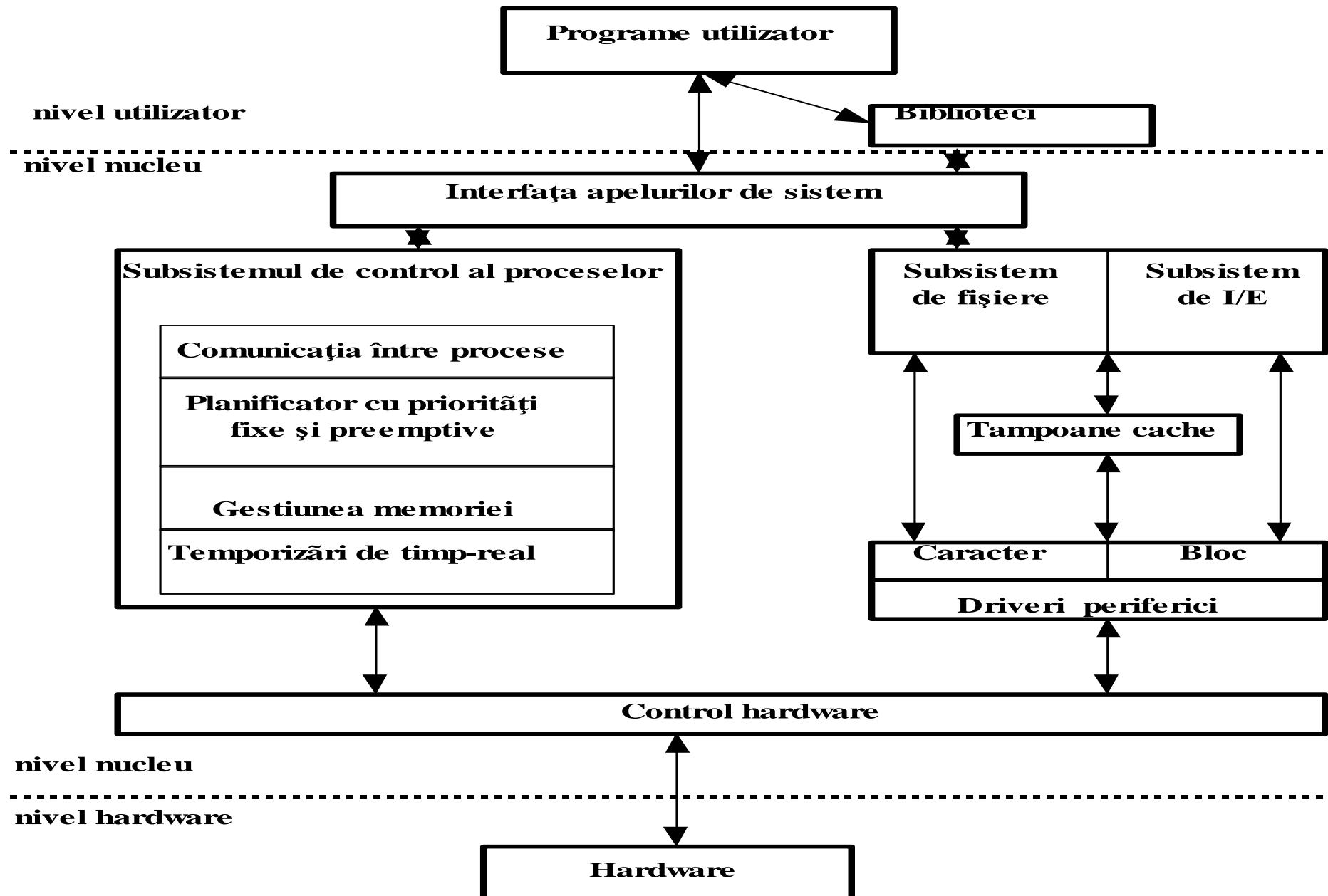
X- blocked state

P/S – waiting - unblocked by P (process) or S (system) signal

T – time wait – the processes entered here due to a delay request and will be unblocked when the wait time expires.

Z – Zombie \leftarrow `exit()` – the processes called *exit()* – they cannot be reactivated again.





Dynamic memory allocation

Determinism of service times is a relevant issue for the **dynamic allocation** of RAM memory. Many non-real-time operating systems offer memory allocation services from what is termed as **Heap**.

The **malloc** (memory allocation) and **free** services (system calls) known in C-language work from a heap.

Tasks can temporarily borrow some memory from the operating system's heap by calling "**malloc**", and specifying the size of memory buffer needed.

When this task (or another task) is finishes the usage of this memory buffer it returns the buffer to the operating system by calling "**free**."

The operating system will then return the buffer to the heap, where its memory might be used again, perhaps as part of a larger buffer. Often the buffer may be broken into several smaller buffers in the future.

Heaps suffer from a phenomenon called **External Memory Fragmentation** that may cause the heap services to degrade.

This fragmentation problem can be solved by so-called **garbage collection** (defragmentation) software.

Unfortunately, "garbage collection" algorithms are often wildly non-deterministic – injecting randomly-appearing random-duration delays into heap services. These are often seen in the memory allocation services of general-computing non-real-time operating systems.

Memory allocation at the process creation

```
fork()  
  
#include <stdio.h>  
main () {  
    puts ("Inceputul testului fork");  
    fork();  
    puts ("Sfarsitul testului fork");  
}
```

Parent process writes:
Inceputul testului fork
Sfarsitul testului fork

Child process writes:
Sfarsitul testului fork

Context switch = load & store information:
processor general (CPU)
registers
program counter
(pointer to text)
pointer to un-initialized data
stack pointer
pending signals
process state etc.

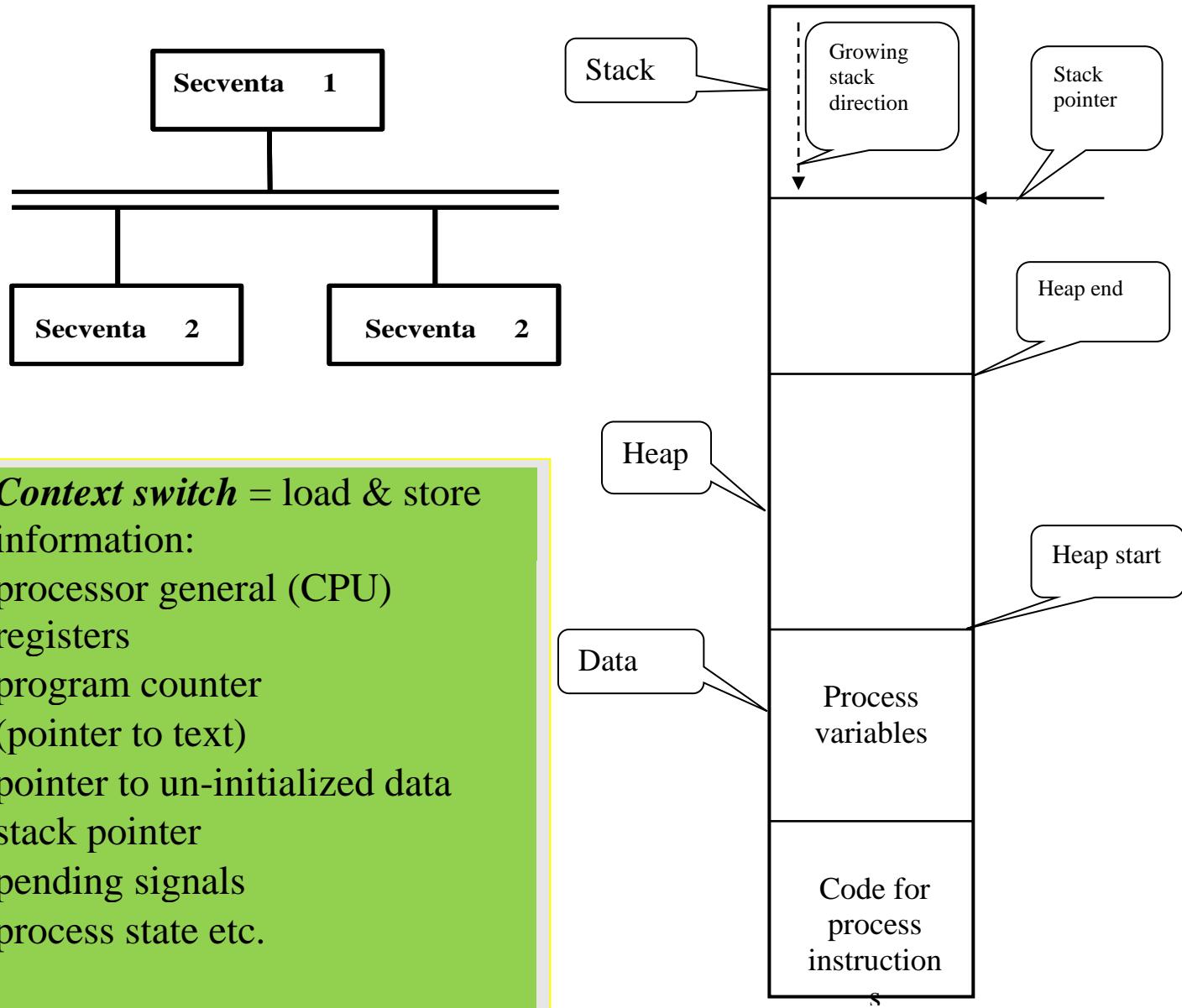


Fig. 2. Memory content of a

Reentrant code

Code that may be called by more than one process at the same time must be *reentrant*.

If the code is reentrant its multiplexing provides correct results.

E.g.: A function that uses global variables for returning results.

Functions that use temporal variables must have been saved in caller space when switching occurs. → Context switch

Non-reentrant code

It cannot be called by more than one process in the same time.

// E.g. non-reentrant code:

```
int[] o3=new int[3];
int[] method(int[] o1, int[] o2) {
    for (i=0;i<3;i++) {
        o3[i]= o3[i] + o1[i]+o2[i];
        o2[i]= o3[i];
    }
    return o3;
}
```

// *o3 remains changed*

//the input parameter o2 is (remains) changed

Conclusion: non-setter method!

```
int x;
.....
//method definition
int add(int i, int j) {
    x=x+i+j;
    return x;
}
```

PID control algorithm can be constructed reentrant or non-reentrant.
How should it be when it is used by ten control loops?

Homework:

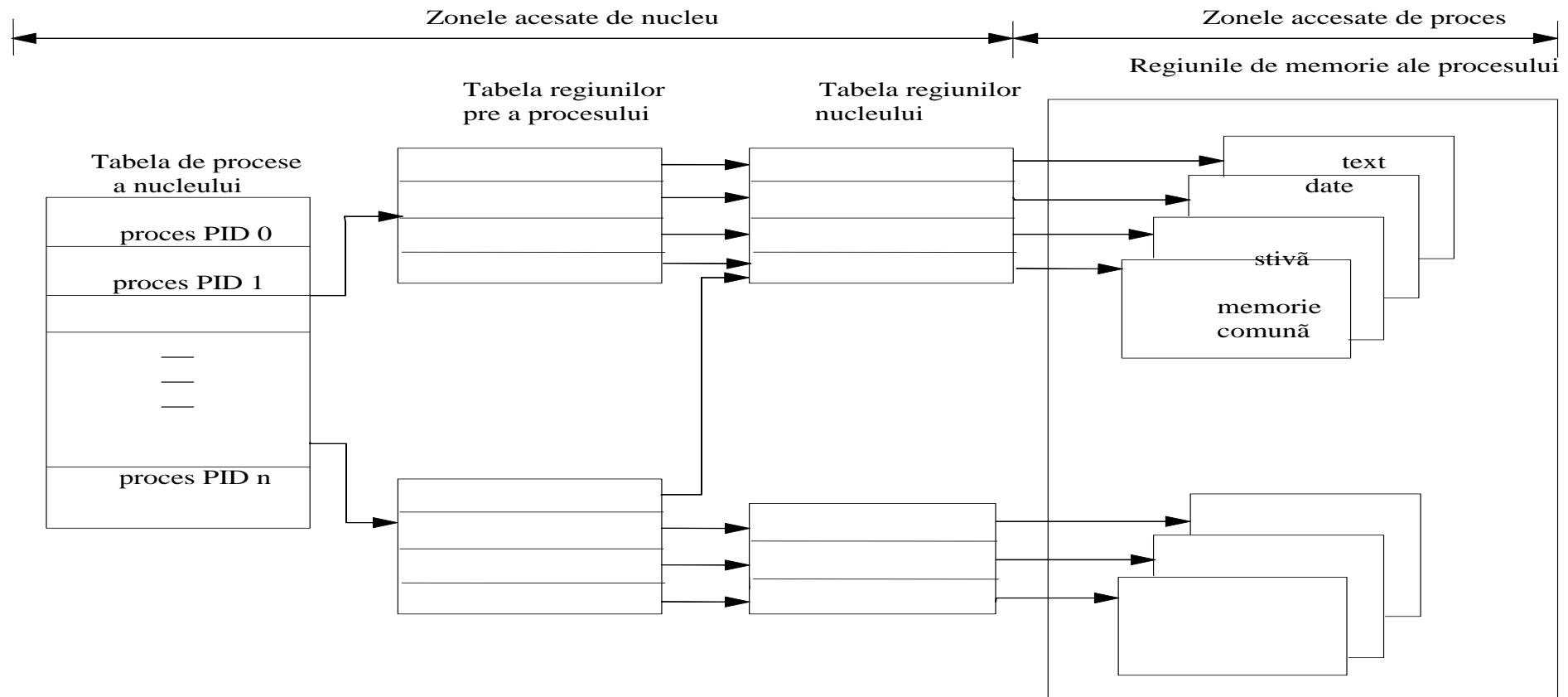
Conceive an example of reentrant code using an OETPN model. It has two tasks that use (repeatedly) and two shared objects. One shared object has a reentrant method and the other a non-reentrant method. Provide the Java code for the shared objects. Do not impose unnecessary constraints but impose the necessary constraint.

Process Representation

Conceptually a process/thread consists of:

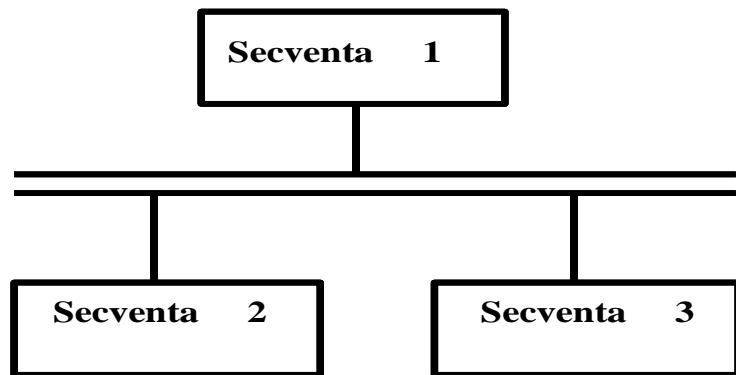
- the code to be executed
 - parameter-less procedure
 - Java: run method
- a stack
 - local variables of the process
 - arguments and local variables of procedures called by the process
 - when suspended: storage place for the values of programmable registers and
 - program counter
- a process record (process (task) control block)
 - administrative information
 - priority, a pointer to the code of the process, a pointer to the stack of the process, etc

UNIX task table structure



Memory structure of a program with two processes

Can the memory influence the application reactivity?
Can the memory involve the deadlines missing?



fork() → {0 , -1} – child , parent

```

#include <stdio.h>
main () {
    int pid_copil;
    printf ("Proces parinte - secventa 1\n");
    if ((pid_copil=fork () )==0)
        printf ("Proces copil - secventa 3 \n");
    else printf ("Proces parinte - secventa 2 \n");
}
  
```

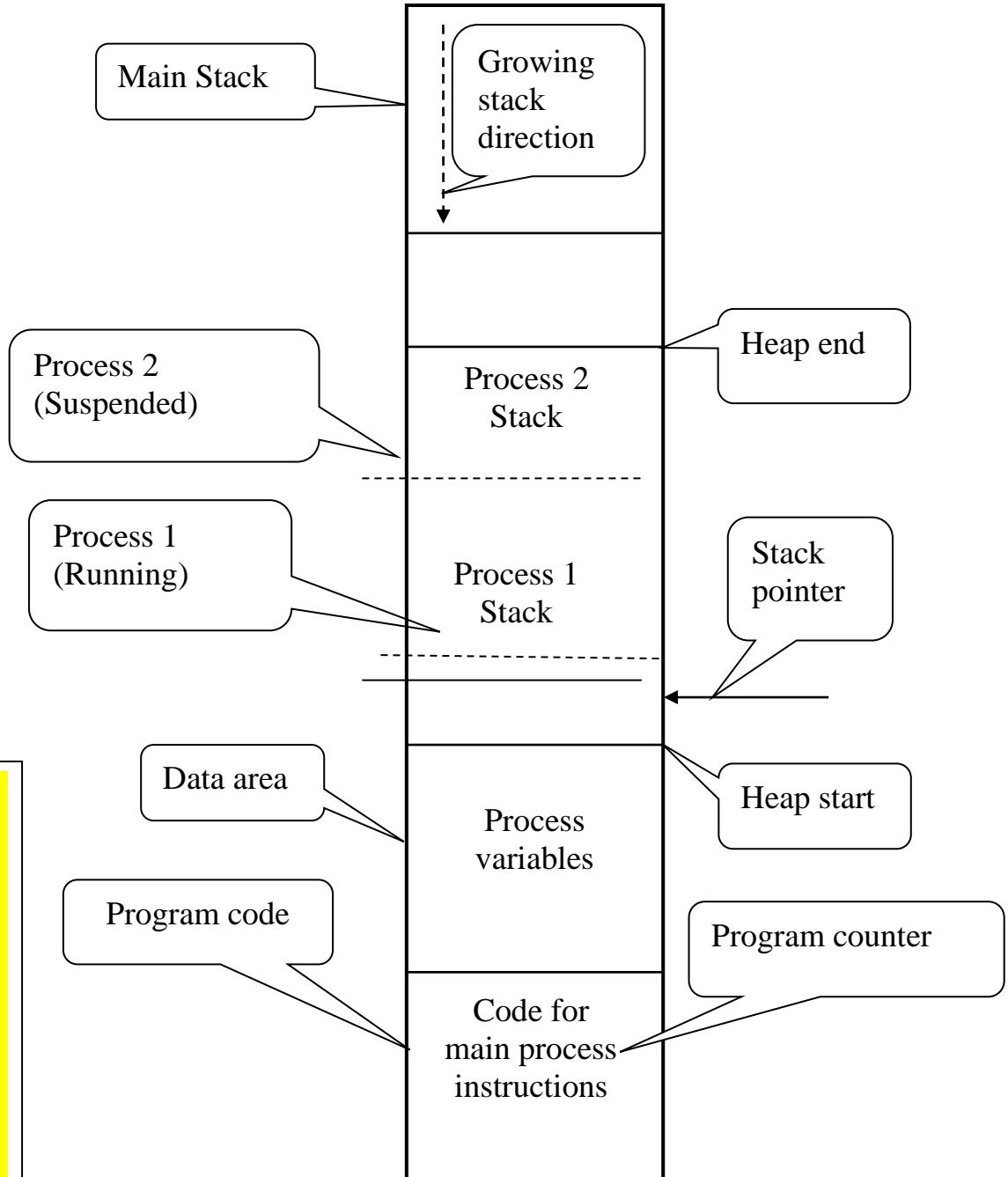


Fig. 2. Memory content of a process.

Inter-task communication and synchronization

RTOSs offer mechanisms for communication and synchronization between tasks. This is necessary in a preemptive environment of many tasks, because without them the tasks might well communicate corrupted information or otherwise interfere with each other.

For instance, a task might be preempted when it is in the middle of updating a structure of data. If another task preempts it reads from that structure, the first task will read a combination of some areas of newly updated data plus some areas of data that have not yet been updated.

RTOS's mechanisms for communication and synchronization between tasks are provided to avoid these kinds of errors.

Most RTOSs provide several mechanisms, with each mechanism optimized for reliably passing a different kind of information from task to task.

A communication between tasks in embedded systems is the ***passing of data*** from one task to another.

Many RTOSs offer a ***message passing mechanism*** for doing this. Each message can contain an array or buffer of data.

Common Resources

- shared variables
- external devices (printer, screen, keyboard etc.)
- non-reentrant code

All have to be used inside *critical sections*.

Process1
init(mutex);
.....
wait(mutex);
*acces critical section;
signal(mutex);
.....

Process2
.....
wait(mutex);
*access critical section;
signal(mutex);
.....

Events

- synchronous signaling
- asynchronous process interrupt

Mutual exclusion

Semaphores: {counter & waiting queue}
Non negative counter + two operations
wait(semaphore); P(sem) ↵ passeren
acquire(semaphore);
signal(semaphore); V(sem) ↵ vriegeven
release(semaphore);

Implementation:

- require **shared memory** creation
- share memory attachment
- semaphore creation
- operation definition (implementation)

Event signaling

Process1
.....	wait(event);
.....	signal(event);
.....

Interrupts & Time

Interrupts:

- clock interrupts
- time primitives
- periodic processes

Computer external communication:

- polling
- interrupts

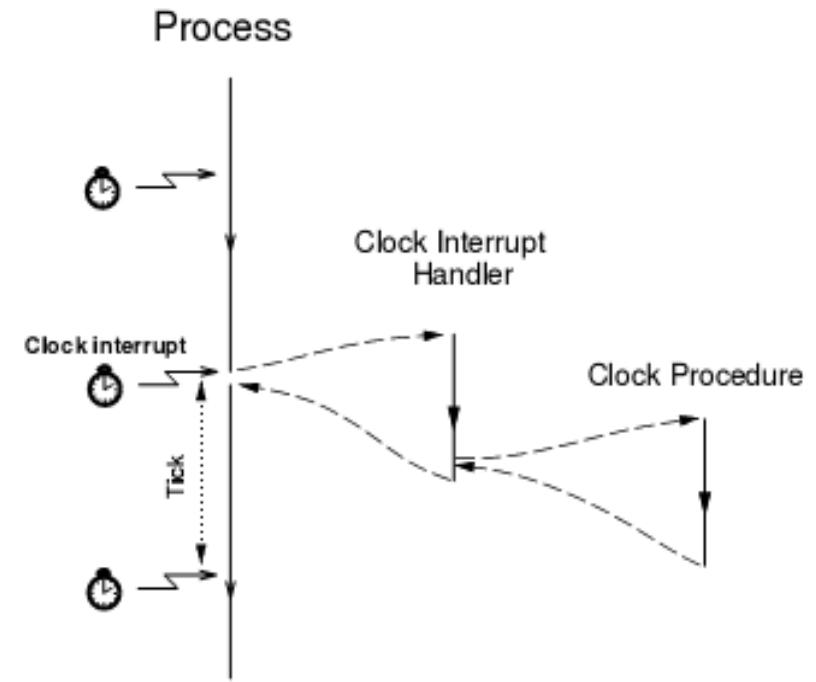
Event → a context switch may be initiated from the interrupt handler.

Program counter must always be saved and restored

The interrupt handler role is to save away and restore the registers it uses.

Context saved:

- on the stack of the interrupted process
- on a special stack common to all interrupts
- in a specialized set of registers (DSPs, PowerPC, ...)



Interrupt priorities

Disabling the interrupts in the kernel causes all interrupt levels to be disabled. → *Hardware priorities*

There is possible to store only a limited number of pending interrupts.

Interrupt handlers need to be short and efficient.

Example of interrupt usage → for time consuming processing in device processes.

Event-Based Clock Interrupts

Clock interrupts from a variable time source (e.g. external hardware timing chip) instead of a fixed clock.

When a process is inserted in *TimeQueue* the kernel sets up the timer to give an interrupt at the wake-up time of the first process in TimeQueue.

When the clock interrupt occurs, a context switch to the first process is performed and the timing chip is set up to give an interrupt at the wake-up time of the new first process in TimeQueue.

Foreground-Background Scheduler

Foreground tasks (e.g. controllers) execute in interrupt handlers.

The background task runs as the main program loop.

It is a common way to achieve simple concurrency on low-end implementation platforms that do not support any real-time kernels.

*

END

*

4.5. Concurrency Programming in Standard Java

Concurrent Programming

1. Introduction
2. Cooperation
3. Thread creation and execution
4. Control of thread execution
5. Deadlock
6. ThreadGroup
7. Thread intercommunication (Observers)
8. Java classes for concurrency
9. Implementation of OETPN models
10. Implementation of UML RT diagrams
11. Timing programming in Standard Java
12. Java 8 Concurrency
13. Concurrent architectures

1. Introduction

What does concurrency mean?

How can a single processor to execute simultaneously more than one thread?

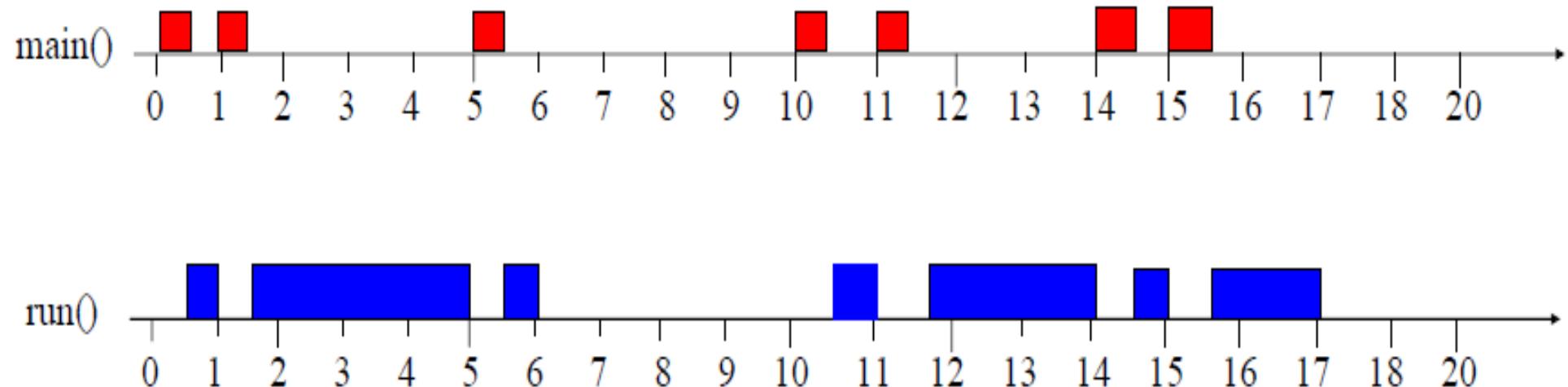


Fig. x. Representation of the concurrent execution of two threads.

Concurrency implementation:

- Multiprogramming = multitasking
- multiprocessing = multiproces
- multithreading = multifir = multifilar
- distributed programming = programare distribuită
-
- Does the multithreading implementation load the processor more or less?
-
- **What is the reason of multithreading implementation?**
-
- ***Independent execution or cooperation?***

Java Execution Model

JVM does:

- consider the threads as abstractions inside it;
- holds within the thread objects all information related to the threads (the thread's stack, the current instruction of the thread, bookkeeping information);
 - performs context switching between threads by saving and restoring thread contexts;
 - keeps program counter that points at the current instruction of the executing thread;
 - keeps the global program counter that points at the current instruction of the JVM.

2. Cooperation

Execution:

divizarea timpului procesorului = time-sharing

programare multifir = multi-threading programming → time-slicing

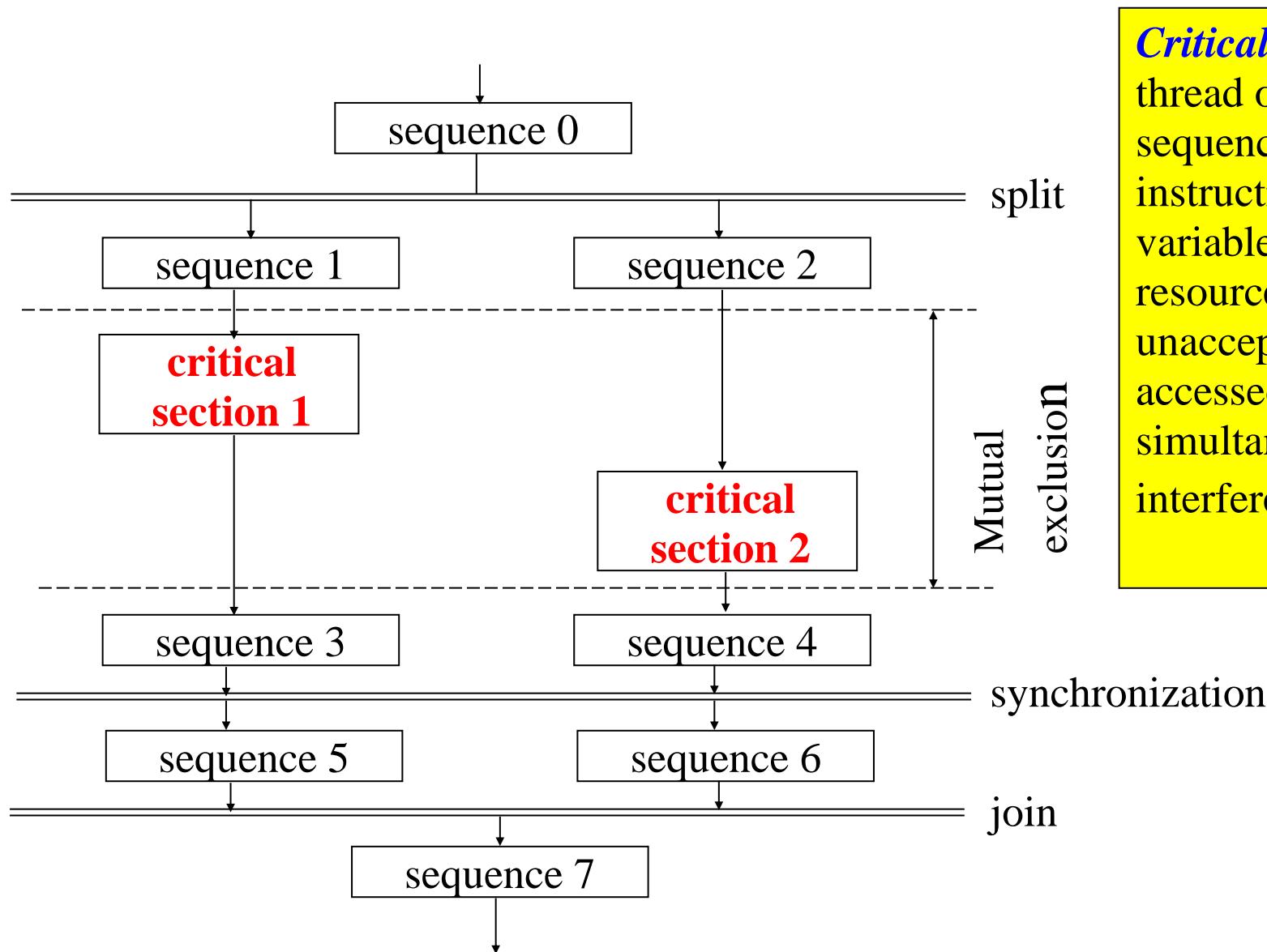
Microsoft Windows: Program=process; process ↔ threads

thread ← thread of execution, thread of control

Thread cooperation:

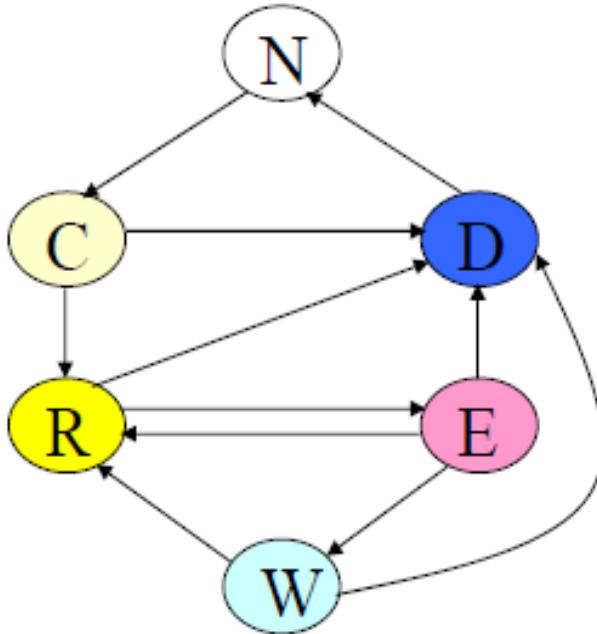
- communication – information interchanging
- synchronization – ordering thread sequences
- shared resource usage
- timing – physical time → real-time clock

Coordinated diagram of a concurrent program (*similar with UML activity diagram*):



Critical section =
thread or process
sequence of
instructions that uses
variables or
resources
unacceptable to be
accessed (executed)
simultaneously (by
interference).

N – Non-existent,
 C – Created,
 D – Dead = zombie,
 R – Ready to execution,
 E – Execution =running,
 W – Waiting = blocked.

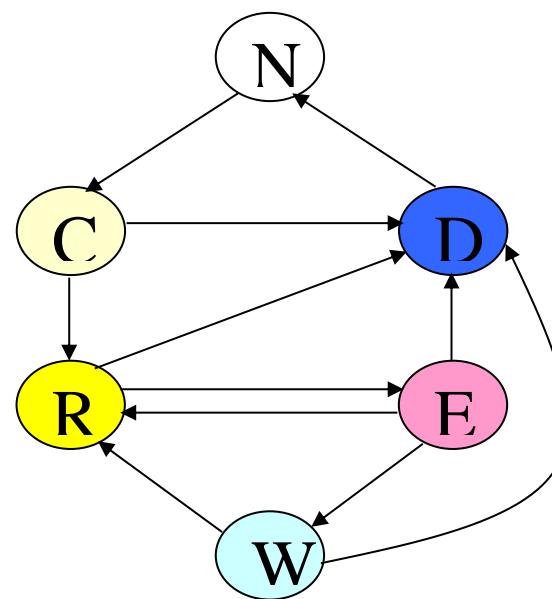


State machine of Java threads
 It describes the thread's transitions.

Transition	Transition mode
N → C	= new
D → N	destroy() și =null
C → D	stop() – <i>deprecated</i>
C → R	start()
R → E	Dispatcher after Scheduler proceeding (planificatorul); Dispatche first in the queues
E → R	yield(), timing
E → D	catch(), stop() , exit(), regular termination
E → W	suspend() , wait(), join(), sleep(), I/O operation, call of a <i>synchronized</i> method
W → R	resume() , notify(), notifyAll(), join(), end wait() , sleep() , I/O operation end, synchronized method end
W → D	stop() , shutdown() , shutdownNow()

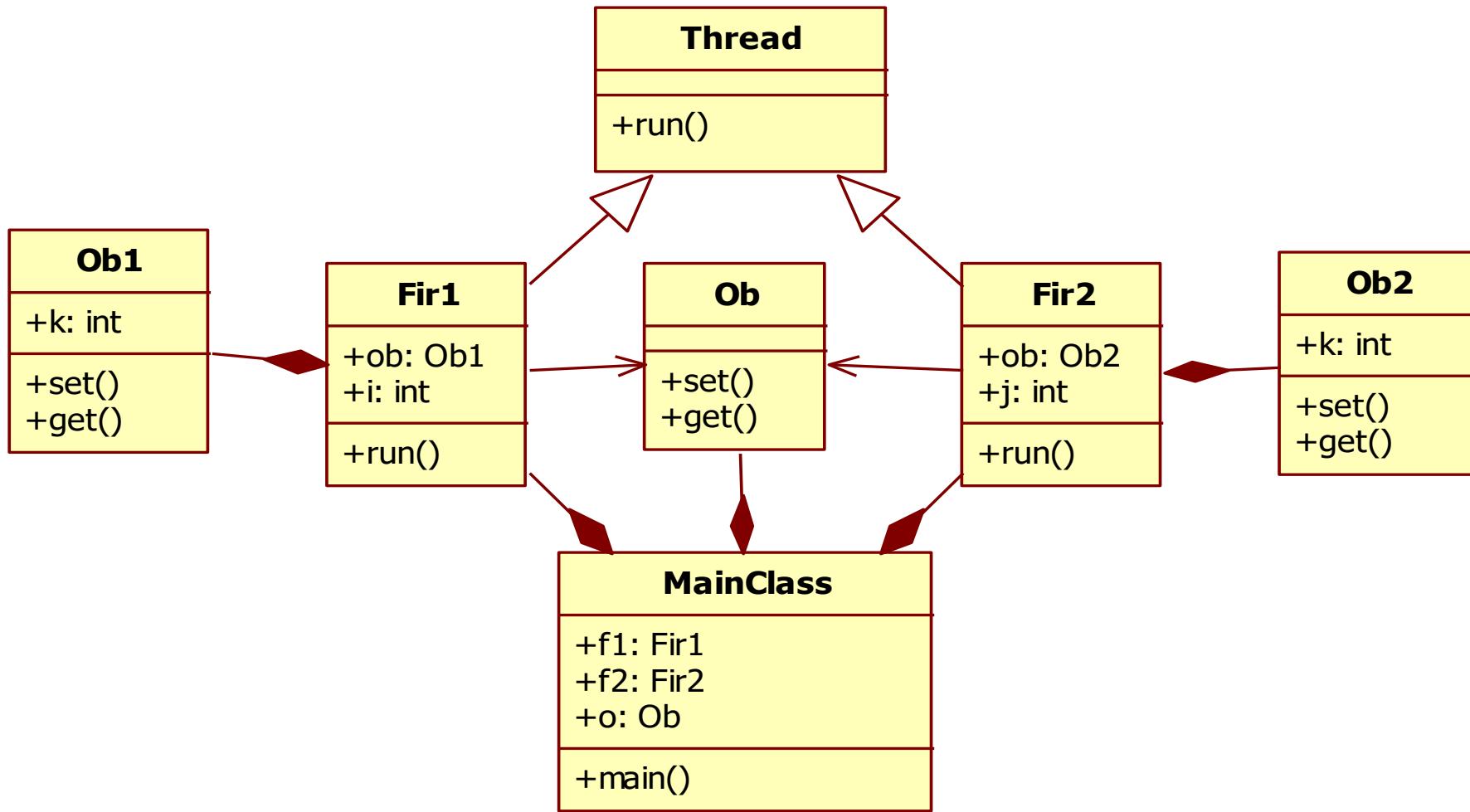
A thread is moved from a state to another when:

- it calls some methods (ex. synchronized method) → **software interrupt**
- a **hardware interrupt** occurs (real-time clock, peripheral device etc.)
- another thread being in execution calls some methods (e.g. a synchronized method)



Variables of threads

- Thread variables and main() thread variables.
- Regular - the variables are stored into the main memory..
- Each thread has a working memory where are stored working copies of the used variables.
- The main memory has a *master* copy of each variable.
- There are rules when the variable copies are transferred into the main memory and reverse.
- The sensible data should be declared *volatile* → atomic transfer.



How are the variables of the threads managed?

Java multithreading programming is achieved using classes and interfaces of *java.lang package*:

- *class Thread*
- *interface Runnable*
- *class ThreadGroup*

public class Thread extends Object implements Runnable

- MAX_PRIORITY = 10
- MIN_PRIORITY = 1
- NORM_PRIORITY - default = 5.

Constructors:

- *public Thread()* – instantiates a new thread
- *public Thread(Runnable target)*
- *public Thread(String name)*
- *public Thread(ThreadGroup group, Runnable target)*
- *public Thread(Thread group, Runnable target, String name)*

3. Thread Creation and Execution

A thread (an *active object*) execution involves:

- declaration like any Java object,
- instantiation or creation like any Java object,
- launching with *start()* method unlike other Java *passive object*,
- stopped → *stop()*,
- finally destroyed and removed from the memory.

A thread is executed until all its activities have been performed, it is stopped by another or it calls *exit()* method.

Thread creation by Thread class extension

```
public class NumeClasaFir extends Thread {  
    public NumeClasaFir() {//constructor  
    .....  
}  
    public void start() {  
        .....  
    }  
    public void stop() {//deprecated ← overwritten  
        .....  
    }  
    public void run() {//method executed concurrently with others Java threads  
    .....  
}  
}
```

Method run():

```
boolean loop=true;  
run() {  
    while(loop) {  
        .....  
    }  
}  
public void start() {//e.g. of overwriting the Thread class start method  
if(firNou==null) {//if the thread is dead, create another one  
    firNou=new Thread(this);  
    firNou.start();  
}  
}  
public void stop() {  
    if(this!=null) this.stop();//SAU  
    loop=false; //loop logical variable ← it stops (breaks) the execution  
}
```

Runnable interface implementation

```
public class Target extends SuperClass implements Runnable {  
    public void run() {  
        // declarations  
        // method body;  
    }  
}  
.....
```

All the threads have names
setName(String givenName)
Thread() – no name Thread-X

```
Target tinta=new Target();  
Thread fir=new Thread(tinta);  
fir.start();
```

```

import javax.swing.*;
public class HidenThread extends JFrame implements Runnable{
    JTextField tf;
    boolean loop=true;
    public HidenThread() {
        super();
        this.setSize(250,150);
        tf=new JTextField(20);
        tf.setBounds(20,20,120,20);
        this.add(tf);
        setVisible(true);
    }
    public void run() {
        int i=0;
        while(loop) {
            tf.setText("interation= "+i);
            try{Thread.sleep(1000);}catch(InterruptedException e) {}
            i++;
        }
    }
}

```

E.g. Target thread solution:

```

public class MainClass {
    public static void main(String[] args)
    {
        HidenThread ht=new
        HidenThread();
        Thread thread=new Thread(ht);
        thread.start();
    }
}

```

Homework: Application class diagram.

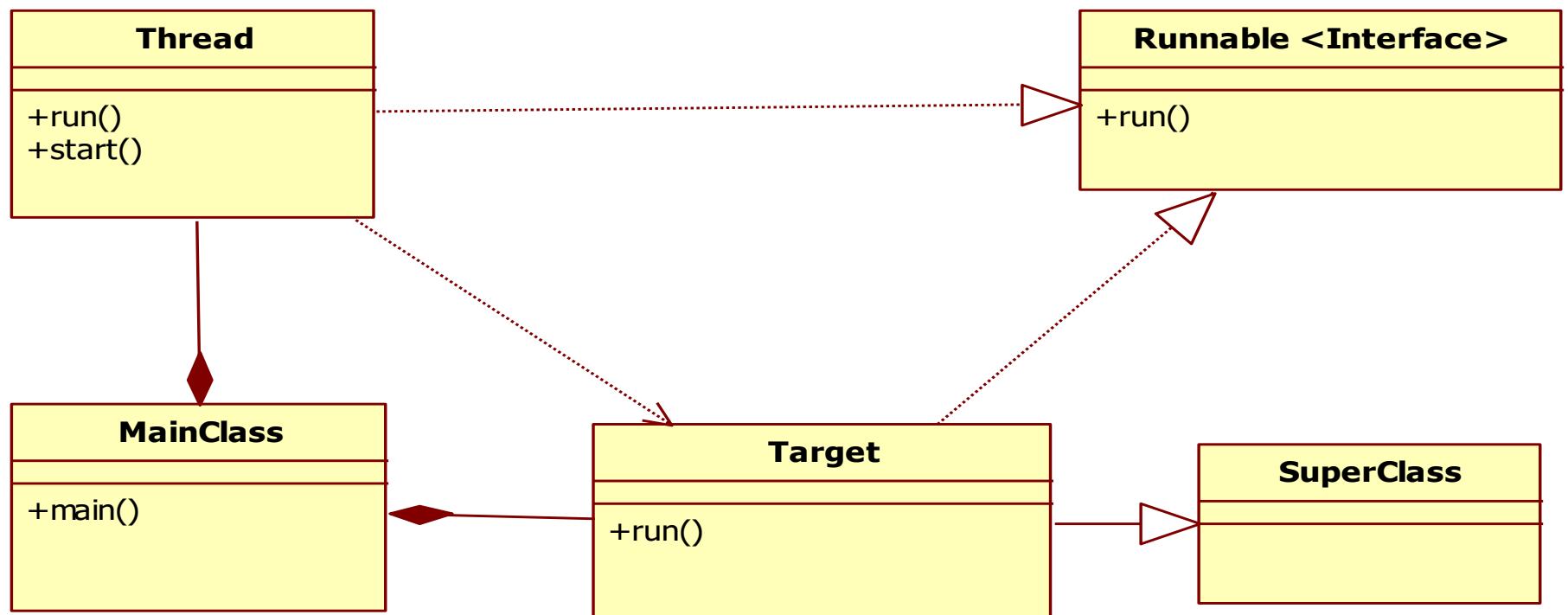


Fig. x. Diagrama claselor unui program care creeaza un fir dintr-o clasa care extinde alta clasa.

Add here the relevant methods and attributes

Thread starting

fir.start();

Selflaunching -

```
class Autolansatoare extends Thread {  
    Autolansatoare() {  
        start();  
    }  
    public void run() {  
        .....  
    }  
}
```

Metoda *isAlive()* - return *true* if the thread execution is not ended.

Waiting the termination of other threads (children)

```
try {  
    fir1.join(); //waits the thread  
    fir0.join(); //  
    System.out.println("S-a terminat executia firelor.");  
}catch (InterruptedException e)  
{System.out.println("Unul dintre fire a fost intrerupt!");}
```

```
fir0=null; //destroy the thread  
fir1=null;  
//clean the memory  
System.gc();
```

Homework

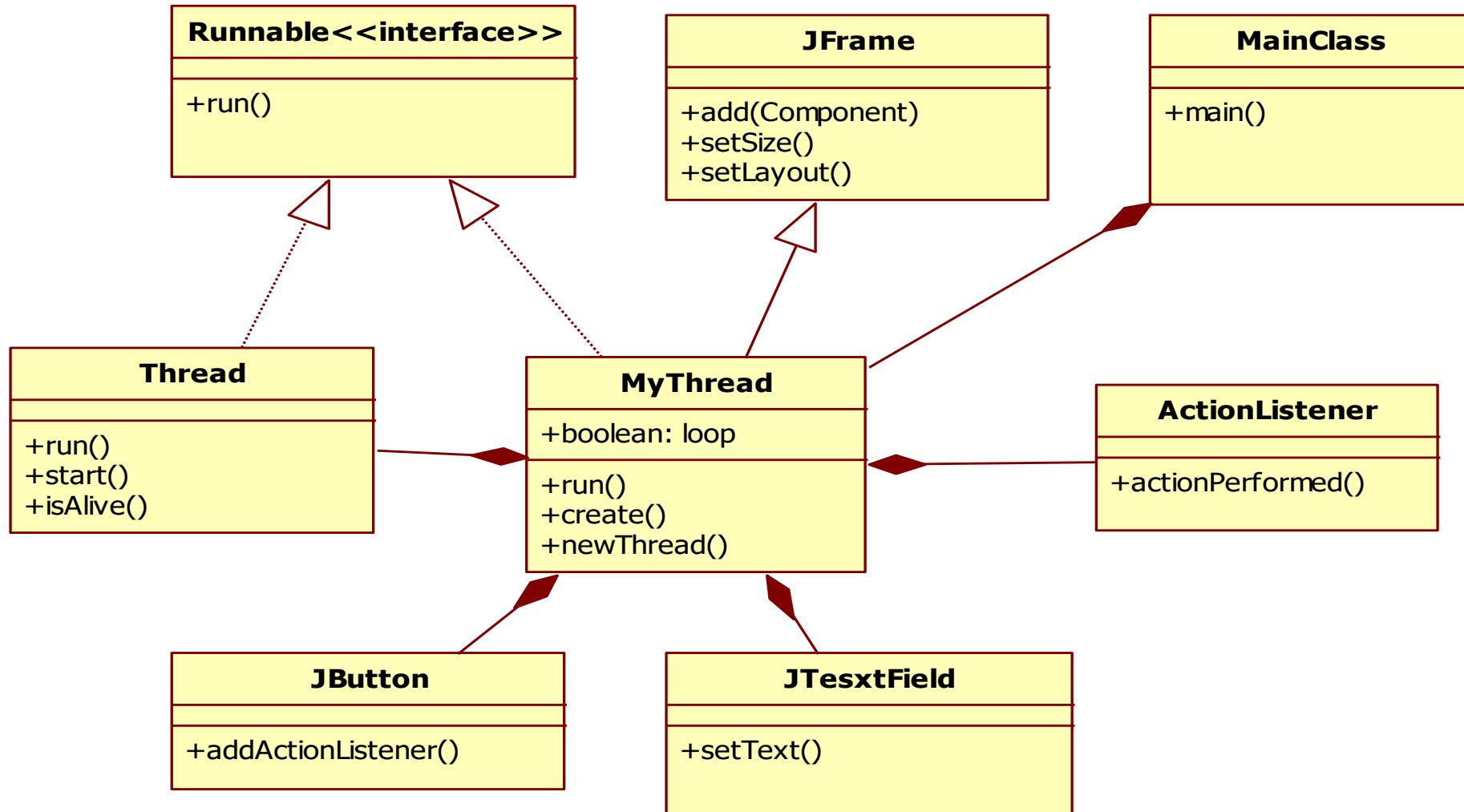
Clasa Thread are metodele `setName()` și `getName()`.

Construiți o aplicație în care:

- firul Main construiește un alt fir (cu Thread)
- firului secundar î se setează un nume
- atât MainThread cât și firul secundar creează câte o fereastră grafică (cu Frame sau JFrame)
- în ferestrele grafice se creează un buton (Button) și un câmp de text (TextField)
- dacă se face clic pe butoane, vor scrie în câmpul de text numele firului

Întrebare: Acțiunile butoanelor (secvențele de instrucțiuni asociate lor) în contul căror fire vor fi executate?

E.g. Application with target threads



```
public class MyThread extends JFrame implements Runnable{  
    JTextField tf1,tf2;  
    boolean loop=false;  
    JButton but1,but2,but3;  
    String threadName="Firul meu";  
    Thread t;  
    public MyThread() {  
        super();  
        t=new Thread(this);  
        .....  
    }  
}
```

```

public void run() {
    int i=0;
    loop=true;
    while(loop) {
        tf1.setText("interation= "+i);
        tf2.setText(this.getName()+""
        pr="+Thread.currentThread().getPriority());
        try{
            Thread.sleep(1000);
        }catch(InterruptedException e) { }
        i++;
    }
    loop=true;
}
ActionListener ac1=new ActionListener(){ //Stop
    public void actionPerformed(ActionEvent eu){
        tf2.setText(Thread.currentThread().getName()+""
        pr="+Thread.currentThread().getPriority());
        if(t.isAlive()) loop=false;
    }
};

```

```

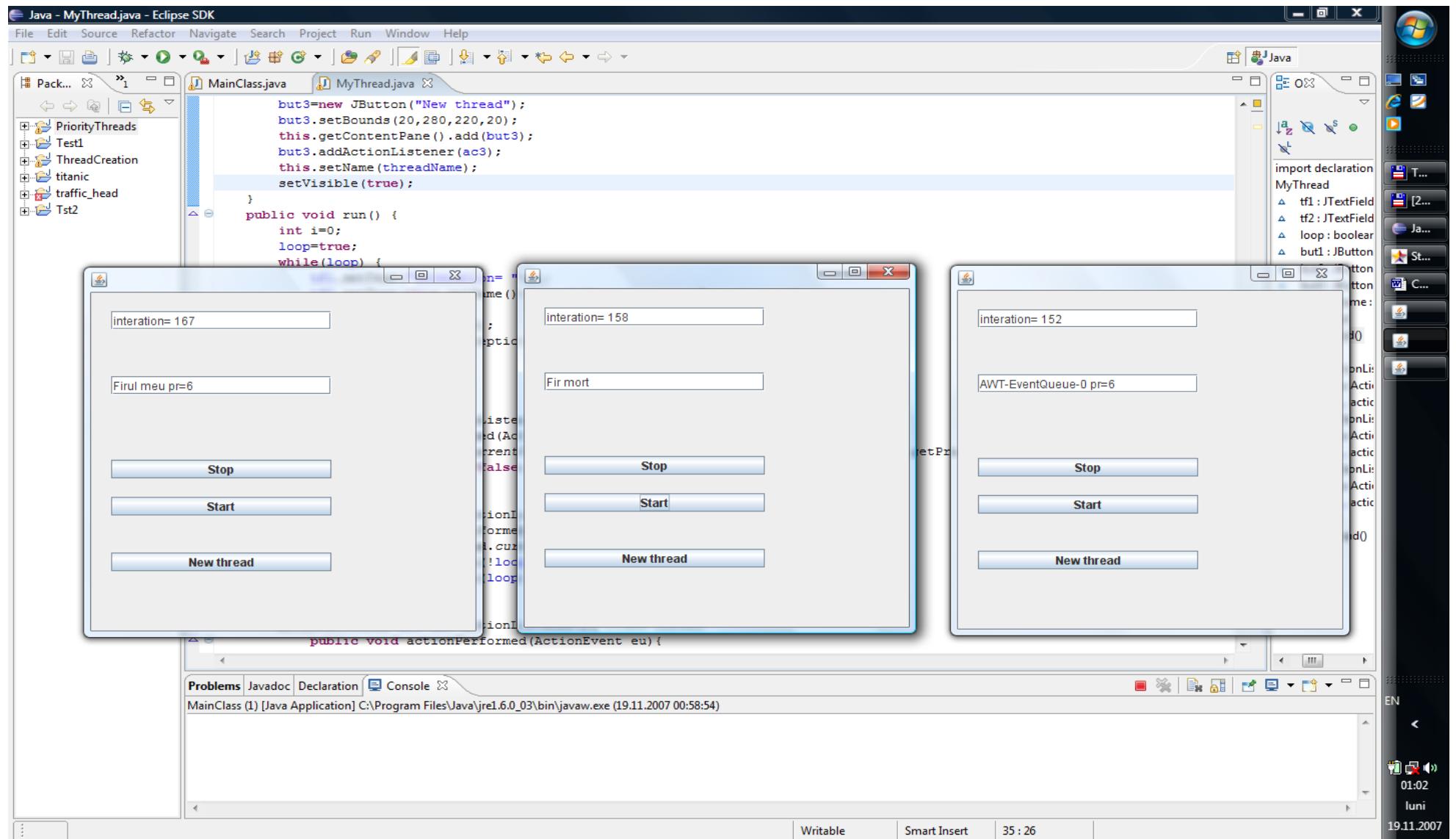
ActionListener ac2=new ActionListener()//Start
    public void actionPerformed(ActionEvent eu){
        tf2.setText(Thread.currentThread().getName());
        if(!t.isAlive()&&(!loop)) t.start();
        if(!t.isAlive()&&(loop)) tf2.setText("Fir mort");
    }
};

ActionListener ac3=new ActionListener(){ /New thread
    public void actionPerformed(ActionEvent eu){
        tf2.setText(Thread.currentThread().getName());
        newThread();
    }
};

public Thread create(){
    Thread thread=new Thread(this);
    thread.start();
    return thread;
}

public Thread newThread(){
    MyThread ht=new MyThread();
    Thread thread=new Thread(ht);
    return thread;  }

```



Can the priorities be modified during execution?

→ Has the application dynamic priorities?

What shows the screen after changing the priorities during application execution?

4. Thread Execution Control

Deprecated methods – *suspend()* and *resume()* – **not recommended**

referintaFir.stop(); //stop()

referintaFir.suspend();

referintaFir.resume();

referintaFir.interrupt();//deprecated

//interrupts the current thread execution – Monitors???

Thread synchronization

1) Synchronization with objects

- main Thread
- AWT Threads
- run() Threads

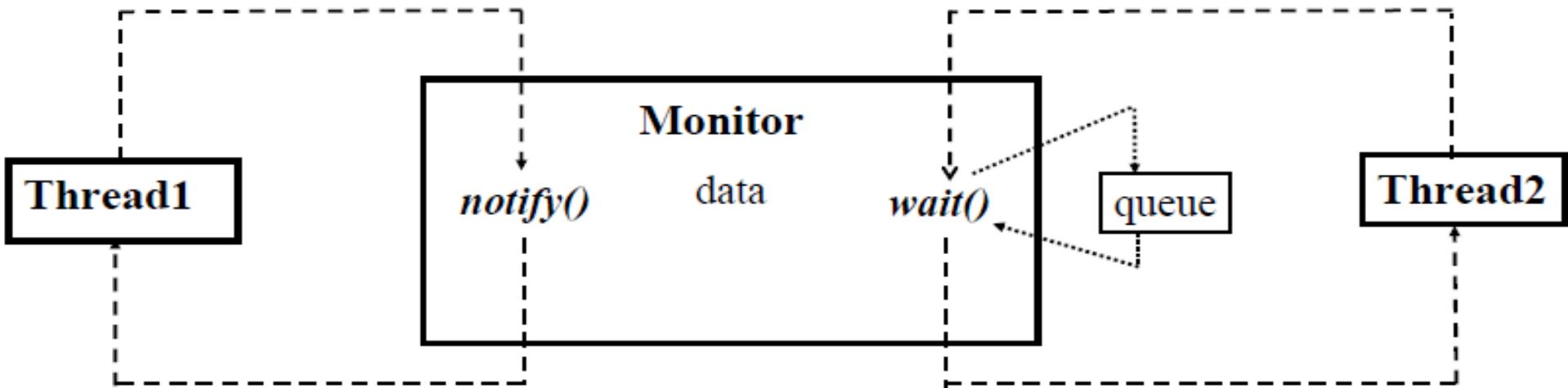
synchronized → monitor

Synchronized object, method, sequence of instructions etc. → **monitor**

Each monitor → waiting queue

synchronized method from different objects of the same class !?!

The monitor is equivalent to a lock



The call of a synchronized method (of an object) is accepted only if the thread is the owner of the object's monitor. Otherwise an exception is signaled.

Only one *synchronized* method from an object can be executed at a moment of time. The synchronized methods of an (same) object cannot be executed by interference.

Nested monitors

→ **nested monitor** = monitoare încubate = utilizarea în cascădă a monitoarelor → troubles?

The threads can be involved into competition for locks acquisition.
The lock involves atomic actions

Reentrant Synchronization

A thread cannot acquire a lock owned by another thread.

A thread *can* acquire a lock that it already owns. → **reentrant** synchronization

2) *Synchronization by methods*

These methods are executed sequentially!

Thread synchronization with the physical time

Object → *wait(duration)*.

```
try {  
    wait();  
} catch(InterruptedException e) {}
```

Thread *sleep(durata)*

```
int duration= 100;  
try{sleep(duration); }catch(InterruptedException e){}
```

MainThread (mainClass) → Thread.sleep().

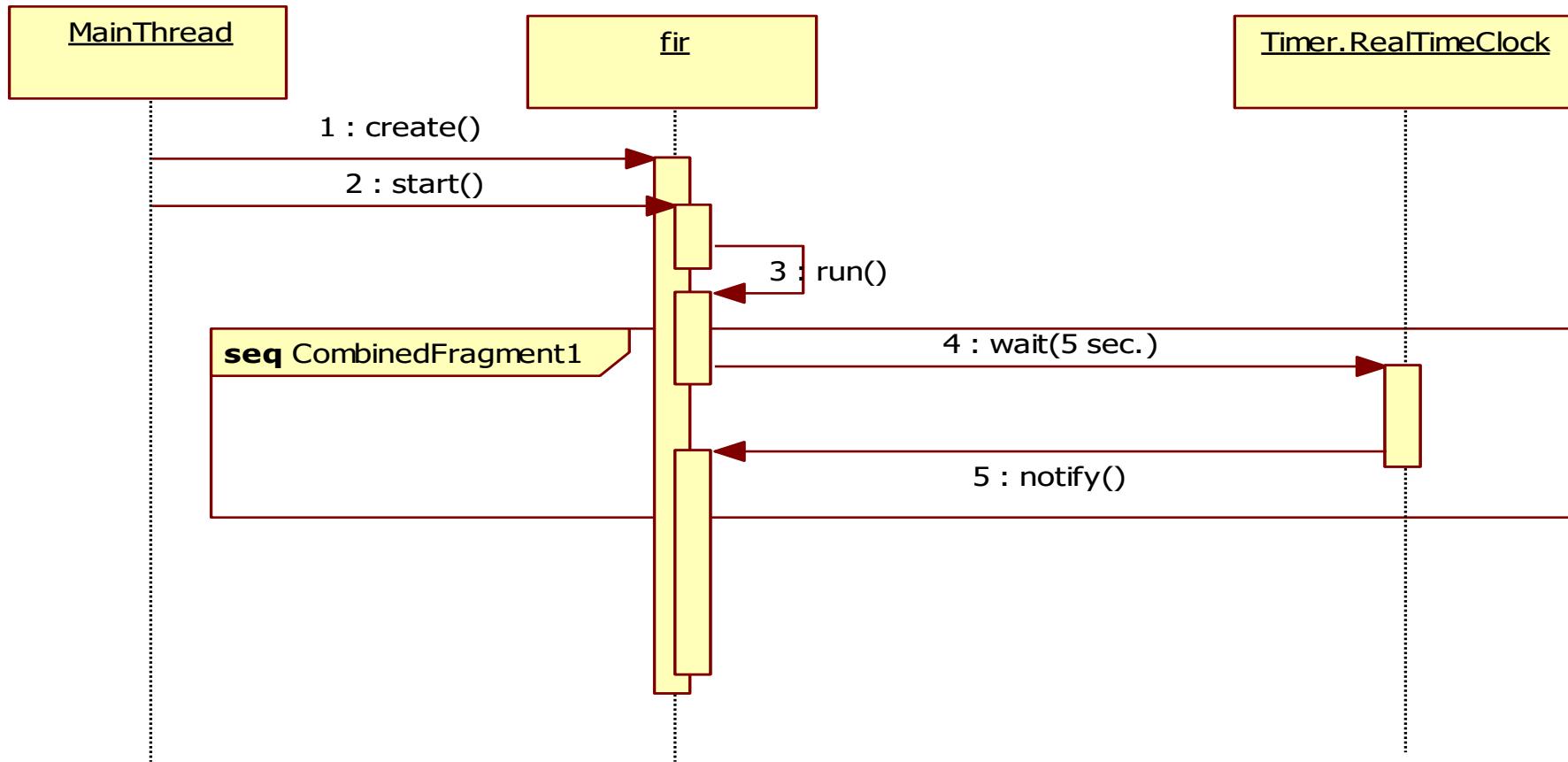


Fig. Y. Temporizarea firelor cu timpul fizic dat de ceasul de timp-real.

Exemplu:

```
class ResursaAtomica {  
    private int a,b;  
    public ResursaAtomica(int i, int j) {  
        a=i;  
        b=j;  
    }  
    public synchronized void update(int da, int db) {  
        a=a+da;  
        b=b+db;  
    }  
    public synchronized void get(int[] t) {  
        t[0]=a;  
        t[1]=b;  
    }  
}
```

Usage:

```
ResursaAtomica res=new ResursaAtomica(1,2);  
res.update(3,4);
```

Different objects?!

Memory consistency → **volatile** –atributes **long** and **double**

3) Synchronization on blocks

Block; { }

```
synchronized(this) {  
    //mutual exclusion implementation  
}
```

```
synchronized (Expression) {  
    * Sequence of instructions;  
}
```

```
String s1,s2;  
.....  
synchronized(s1+s2) {..... }
```

null ➔ NullPointerException.

Sets of critical sections

Utilization of *wait()* and *notify()*

Recommended for synchronization.

Methods *wait()*, *notify()* and *notifyAll()* from *Object*

- *wait()* –
- *wait(long timeout)* – *timeout*;
- *wait(long timeout, int nanos)* –
wait() → *InterruptedException*,
try{} and *catch {}*.

notify() → one thread

notifyAll() → all the threads

→ *IllegalMonitorStateException*.

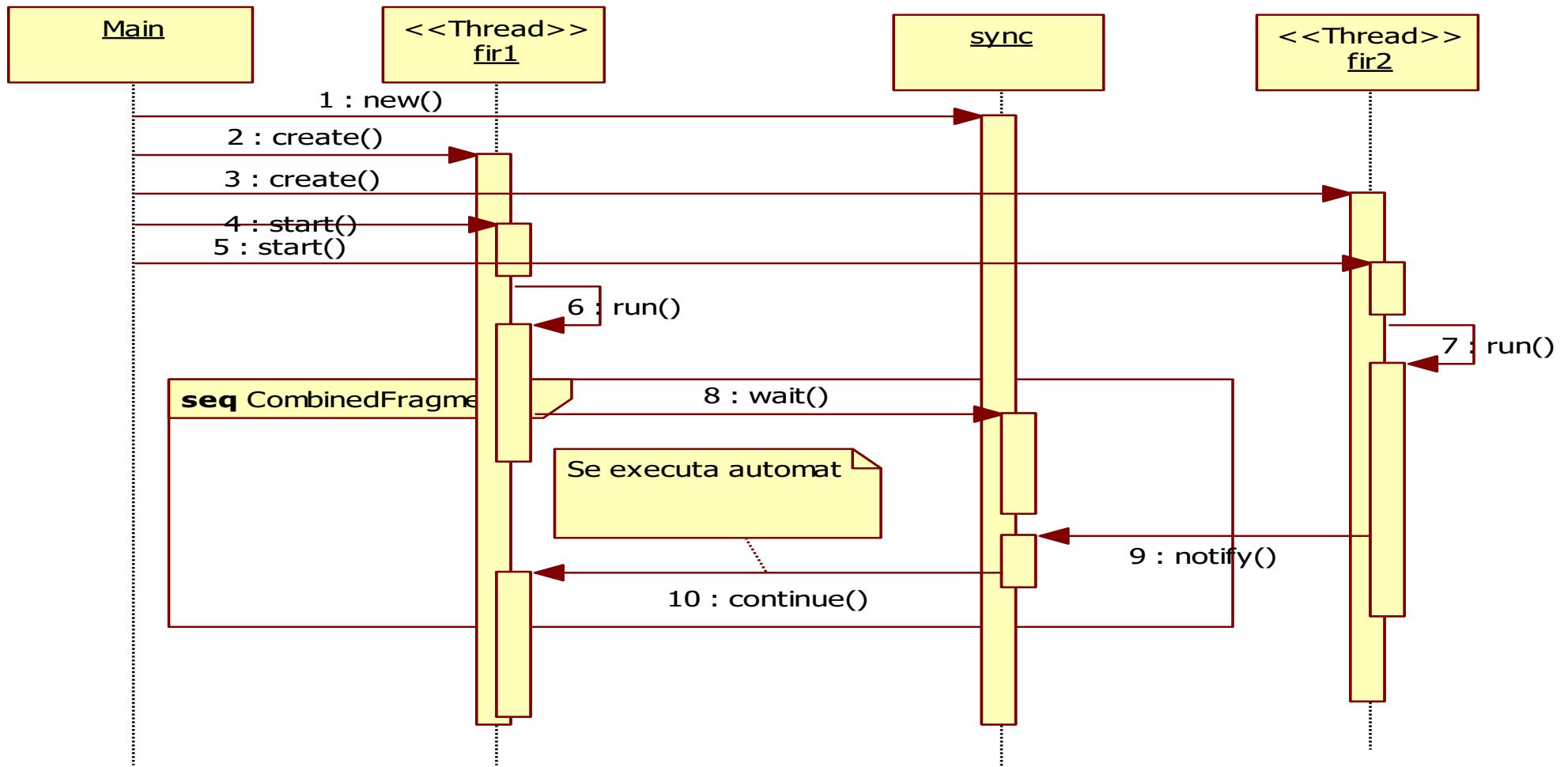


Fig. z. Sincronizarea a două fire.

```
public synchronized void run() {  
    .....  
    try {  
        wait();  
    } catch(InterruptedException e) {}  
    .....  
}
```

Waiting:

```
synchronized(ob) {ob.wait();}
```

Awake:

```
synchronized(ob) {ob.notify();}
```

```
/* Clasa pentru sincronizare */
public class Synch {
    public synchronized void monWait() {
        try {
            wait();
        }catch(InterruptedException e) {
            System.out.println("Eroare: Firul e intrerupt");
        }
    }
    public synchronized void monNotify()
    {
        notify();
    }
}
Synch s=new Synch();
s.monWait();
s.monNotify();
monNotifyAll() ???
```

Stopping the thread execution

Not recommended:

fir.stop(); //→ asynchronous interrupt

Recomended solution:

```
class Fir extends Thread  
{  
    boolean stp=true;  
    public void run()  
    {  
        while(stp)  
        {  
            .....  
            //instructions execute periodic  
            .....  
        }  
    }  
}
```

In another thread:

```
.....  
fir.stp=false;  
.....
```

Monitor utilization

Provider:

- wait room for deposit
- put + increment
- signal the deposit

User:

- wait if there is no deposit
- take – decrement
- signal the take

E.g.

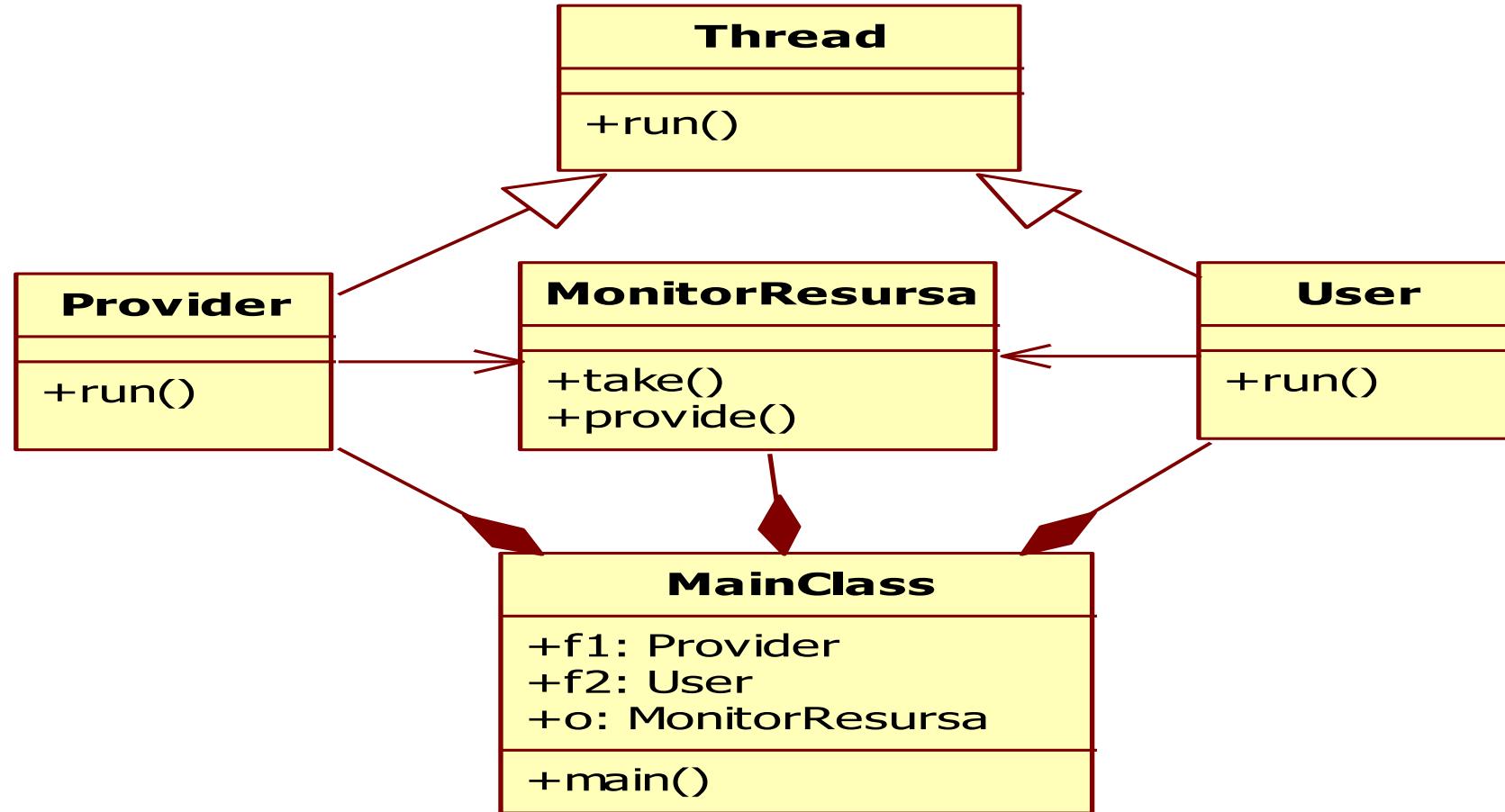
- create a resource;
- a producer
- 2 users
- test program

```
/* Creeaza monitorul unei resurse cu capacitatea data si metode de acces*/
public class MonitorResursa {
    private int n, cap;
    public MonitorResursa(int val, int cp) {
        n=val;
        cap=cp;
    }
    public synchronized void take() throws InterruptedException {
        while(n==0) wait();
        n--;
        System.out.println("take n= "+n);
        notify();
    }
    public synchronized void provide() throws InterruptedException
    {
        while(n==cap) wait();
        n++;
        System.out.println("provide n= "+n);
        notify();
    }
}
```

```
public class Provider extends Thread{  
    MonitorResursa mr;  
    public Provider(MonitorResursa m) {  
        mr=m;  
    }  
    public void run() {  
        while(true){  
            int i =(int)(100*Math.random());  
            try{sleep(i);}catch(InterruptedException e){}  
            try{  
                mr.provide();  
            }catch(InterruptedException e){}  
        }  
    }  
}
```

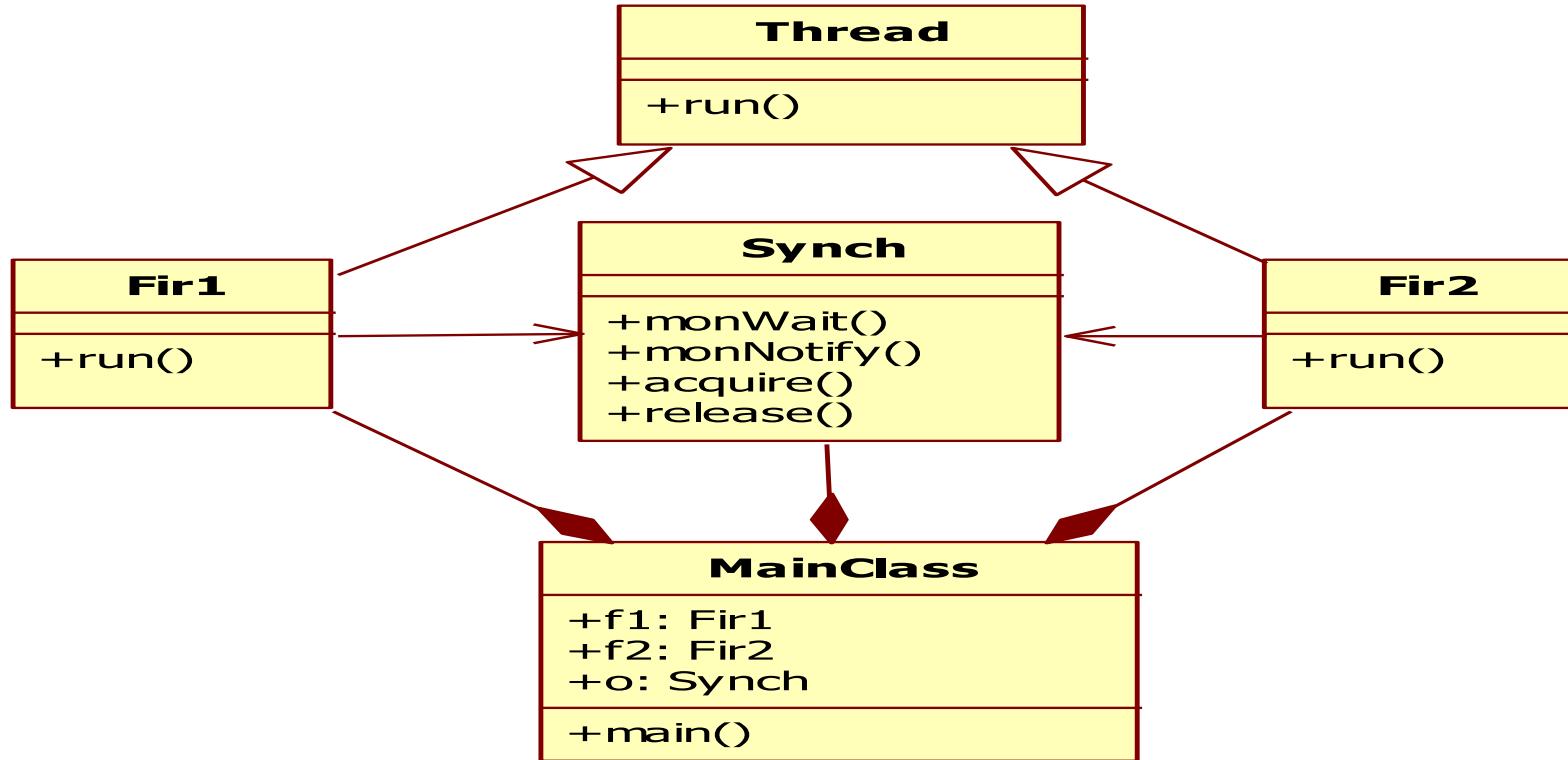
```
public class User extends Thread{  
    MonitorResursa mr;  
    public User(MonitorResursa m) {  
        mr=m;  
    }  
    public void run() {  
        while(true){  
            int i =(int)(100*Math.random());  
            try{sleep(i);}catch(InterruptedException e) {}  
            try{  
                mr.take();  
            }catch(InterruptedException e) {}  
        }  
    }  
}
```

```
/* Testarea utilizarii unui monitor cu un furnizor si 2
utilizatori*/
public class MainClass {
    public static void main(String[] args) {
        MonitorResursa m=new MonitorResursa(0,5);
        Provider p=new Provider(m);
        User u=new User(m);
        p.start();
        u.start();
        User u2=new User(m);
        u2.start();
    }
}
```



Homework: Implement the mutual exclusion using only monitors, `wait()` and `notify()` methods.

E.g. of implementation



5. Deadlock

Deadlock = blocking

Two or more threads can be involved in a deadlock

starving

main:

- create 2 shared resources mon1 și mon2
- create 2 threads fir1 și fir2
- wait the thread to end, if not ask the operator .

Thread 0:

- print: the thread is active
- acquire mon1
- pause 2 sec.
- acquire mon2
- end.

Thread 0:

- print: the thread is active
- acquire mon2
- pause 2 sec.
- acquire mon1
- end.

➔ **deadlock!**

MS JVIEW Class1

Auto A

```
S-a creat un monitor mon0
S-a creat un monitor mon1
Firul main creaza si starteaza doua fire
Creare fir cu numele Thread-0
Creare fir cu numele Thread-1
Este startat firul: Thread-1
Acesta este fir1
Fir1 achizitioneaza mon1
Este startat firul: Thread-0
Fir1 are mon1 si incearca sa achitioneze mon0
Acesta este fir 0
Fir0 achizitioneaza mon0
Fir0 are mon0
fir0 incearca sa achitioneze mon1
A revenit main in activitate!
main sa opreasca firele?

fir0 viu este: true
fir1 viu este: true
Terminare main.
```

Resource control

Resource:

- active
- passive

Resource management and resource access control
([Gestionarea resurselor – Controlul resurselor](#))

- resource manager - [gestionarul resursei](#)
- resource representative - [reprezentantul resursei](#)

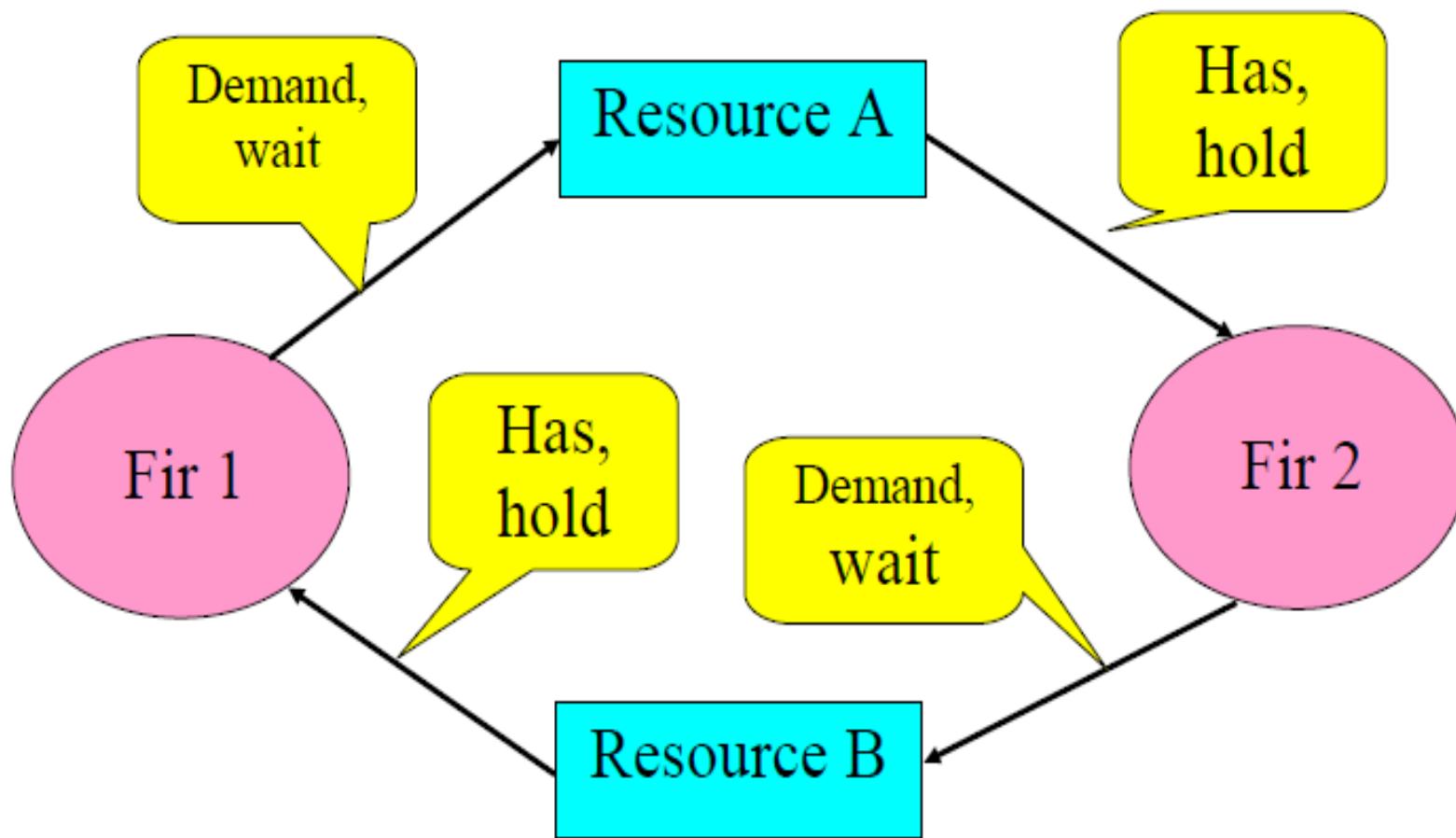


Fig. x. Ciclic waiting (deadlock).

```

import java.util.concurrent.*;
public class Fir extends Thread {
    boolean stp=true;
    Sem s1,s2;
    int pause;
    public Fir(Sem sem1, Sem sem2, String nm, int ps) {
        s1=sem1;
        s2=sem2;
        setName(nm);
        pause=ps;
    }
    public void run() {
        while(stp){
            try {
                s1.acquire();
                System.out.println("Fir "+this.getName()+" has "+s1.getName());
            }catch(InterruptedException e) {System.out.println("Fir
"+this.getName()+
                    " is interrupted "+s1.getName());}
            try{Thread.sleep(pause);}catch(InterruptedException ex) { }
        }
    }
}

```

Rezultat:

Fir main is started
 Fir Fir1 is started
 Fir Fir2 is started
 Fir Fir2 has Semaphore2
 Fir Fir1 has Semaphore1

```

try {
    s2.acquire();
    System.out.println("Fir "+this.getName()+" has "+s2.getName());
} catch(InterruptedException e) {
    System.out.println("Fir "+this.getName()+" is interrupted
    "+s2.getName());
}
s1.release();
s2.release();
}
}
}

```

```

public class MainClass {
    public static void main(String[] args) {
        System.out.println("Fir "+Thread.currentThread().getName()+" is started ");
        Sem s1=new Sem("Semaphore1",1);
        Sem s2=new Sem("Semaphore2",1);
        Fir f1=new Fir(s1,s2,"Fir1", 1111);
        Fir f2=new Fir(s2,s1,"Fir2",2222);
    }
}

```

```
f1.start();
f2.start();
try{
    f1.join();
    f2.join();
}catch(InterruptedException e) { }
}
}
```

Conditions for deadlock

Coffman, Elphick și Shoshani (1971) → if they are fulfilled ***simultaneously***

- 1) *serially reusable resources*) – mutual exclusion. E.g. synchronized
- 2) *incremental acquisition*) –
- 3) *no preemption*) –
- 4) *wait-for cycle* – circular waiting chain.

Deadlock solutions

- prevention
- avoidance
- detection
- recovery

Prevention → NO simultaneously –

- 1) mutual exclusions - **Excluderea reciprocă** –
- 2) hold and wait for another resource - **Deține și așteaptă după altă resursă** –
- 3) no preemption - **Fără preemțiune**
- 4) circular wait - **Așteptare circulară**

Solution: atomic resource acquisition

Detection:

- use the resource allocation graph – ← PNs

-

Recovery:

- break the mutual exclusion
- abandon one or more threads -
- preemption of some resources → chose victims –
Exceptions at the deadlock detection

Homework:

1. Deadlock avoidance. Use Lock to avoid deadlocks. Use the method tryLock().
2. Deadlock prevention. There are 3 threads (f1, f2, f3) and 3 resources (s1, s2, s3).

f1 → s1, s2

f2 → s2, s3

f3 → s3, s1

3. Deadlock recovery. Use a supervisor thread to detect the deadlock and to break it.

6. ThreadGroup

ThreadGroup numeGrup=ThreadGroup(“Gr1”);

ThreadGroup numeGrup2=ThreadGoup(numeGrup, “Gr2”);

Main:

- construiește trei fire individuale cu numele fir0, fir1 și fir2
- construiește un gr1
- construiește un grup gr2
- construiește două fire cu numele fir3 și fir4 pe care le pune în gr2
- startează firele
- așteaptă 10 de secunde ca să avanseze firele
- scrie numărul de fire active din grupuri
- scrie numărul de fire active
- oprește firele din execuție
- așteaptă să termine firele
- termină

Firele:

- setează numele și prioritatea lui
- execută câte o secvență și o contorizează cât timp îi permite main
- scrie prioritatea firului și numărul de secvențe executate

7. Thread Intercommunication

Communication Structures. Different structures are possible:

- many-to-one
- many-to-many
- one-to-one
- one-to-many

1. *Shared objects* –

2. *Streams* - [Prin fluxuri](#)

public PipedInputStream(PipedOutputStream src)

public PipedOutputStream(PipedInputStream snk)

3. *get() and set() methods* in the active objects –

4. *Mailbox Communication*

Synchronization models:

- **Asynchronous**: the sender process proceeds immediately after having sent a message. It requires a buffer for sent but unread messages. Used in the course.
- **Synchronous**: the sender proceeds only when the message has been received. **Rendez-vous**.
- **Remote Invocation**: the sender proceeds only when a reply has been received from the receiver process. **Extended rendez-vous**.

Message queue -

LinkedBlockingQueue

PriorityBlockingQueue

Observers

package ***java.util***

Class Observable

Interface Observer

Interface Observer

void update(Observable o, Object arg)

o - obiectul care a transmis notificarea

arg - obiectul observabil să transmită alte informații cum ar fi:

- articolul de date în care s-a făcut modificarea sau
- evenimentul sau obiectul care a determinat modificarea.

Class Observable

Methods:

- *public synchronized void addObserver(Observer o)* - înregistrează un obiect Observer la acest obiect Observable
- *public synchronized void deleteObserver(Observer o)* - elimină înregistrarea unui obiect Observer de la acest obiect Observable
- *public synchronized int countObservers()* - numără obiectele Observers înregistrate la acest obiect Observable
- *public synchronized void notifyObservers()* - semnalează obiectele Observers înregistrate (practic invocă metoda lor update)
- *public synchronized void notifyObservers(Object arg)* - idem mai sus, dar se transmite și un argument care poate fi preluat de obiectul Observer
- *protected synchronized void setChanged()* - setează un indicator (flag) care semnifică începerea schimbării stării obiectului
- *protected synchronized void clearChanged()* - șterge indicatorul de mai sus
- *protected synchronized boolean hasChanged()* - returnează *true* dacă a nu a fost șters și *false* în caz contrar

```
/** Observable object construction */
import java.util.*;
class Observat extends Observable {
    private int nr=0;
    private String nume;
    public String modicator;
    Observat(String s) {
        nume=new String(s);
    }
    public void incr(String s) {
        setChanged();
        modicator=s;
        nr++;
        notifyObservers(modicator);
        clearChanged();
    }
    public void set(int i, String s) {
        setChanged();
```

```
modificator=new String(s);
nr=i;
notifyObservers(modificator);
clearChanged();
}
public int get() {
    return nr;
}
public String toString() {
    return nume;
}
}
```

Observers:

```
import java.util.*;  
class Observator implements Observer {  
    public synchronized void update(Observable o, Object arg) {  
        Observat ob= (Observat) o;  
        System.out.println("A fost modificat "+o+" la "+ob.get()+"de "+arg);  
        System.out.println("Activat de firul "+  
                           Thread.currentThread().getName());  
    }  
}
```

```
class Actualizare extends Thread {  
    Observat ob;  
    String nume;  
    Actualizare(Observat o, String s) {  
        ob=o;  
        nume=new String(s);  
    }  
    public void run() {  
        while (true) {  
            ob.incr(nume);  
            try {sleep(3333);}  
            catch(InterruptedException e) {}  
        }  
    }  
}
```

```
class Aleator extends Thread {  
    Observat ob;  
    String nume;  
    Aleator(Observat o, String s) {  
        ob=o;  
        nume=new String(s);  
    }  
    public void run() {  
        while (true) {  
            int i= (int)(100*Math.random());  
            ob.set(i,nume);  
            try {sleep(4444);}  
            catch(InterruptedException e) {}  
        }  
    }  
}
```

```
/** Clasa pentru testarea obiectelor observabile si a observatorilor lor */
public class TestObserv
{
    public static void main (String[] args)
    {
        Observat obs1=new Observat("obs1");
        Observat obs2=new Observat("obs2");
        Observator ob1=new Observator();
        Observator ob2=new Observator();
        obs1.addObserver(ob1);
        obs2.addObserver(ob1);
        Actualizare act=new Actualizare(obs1, "ACT");
        Aleator alt=new Aleator(obs2, "ALT");
        act.start();
        alt.start();
        while (true) {
            obs1.incr("MAIN");
            obs2.incr("MAIN");
    
```

```
        try {Thread.sleep(5555);}  
        catch(InterruptedException e) { }  
    }  
}
```

Înregistrarea:

```
Observat obs1=new Observat("obs1");  
Observat obs2=new Observat("obs2");  
Observer ob1=new Observer();  
Observer ob2=new Observer();  
obs1.addObserver(ob1);  
obs2.addObserver(ob1);
```

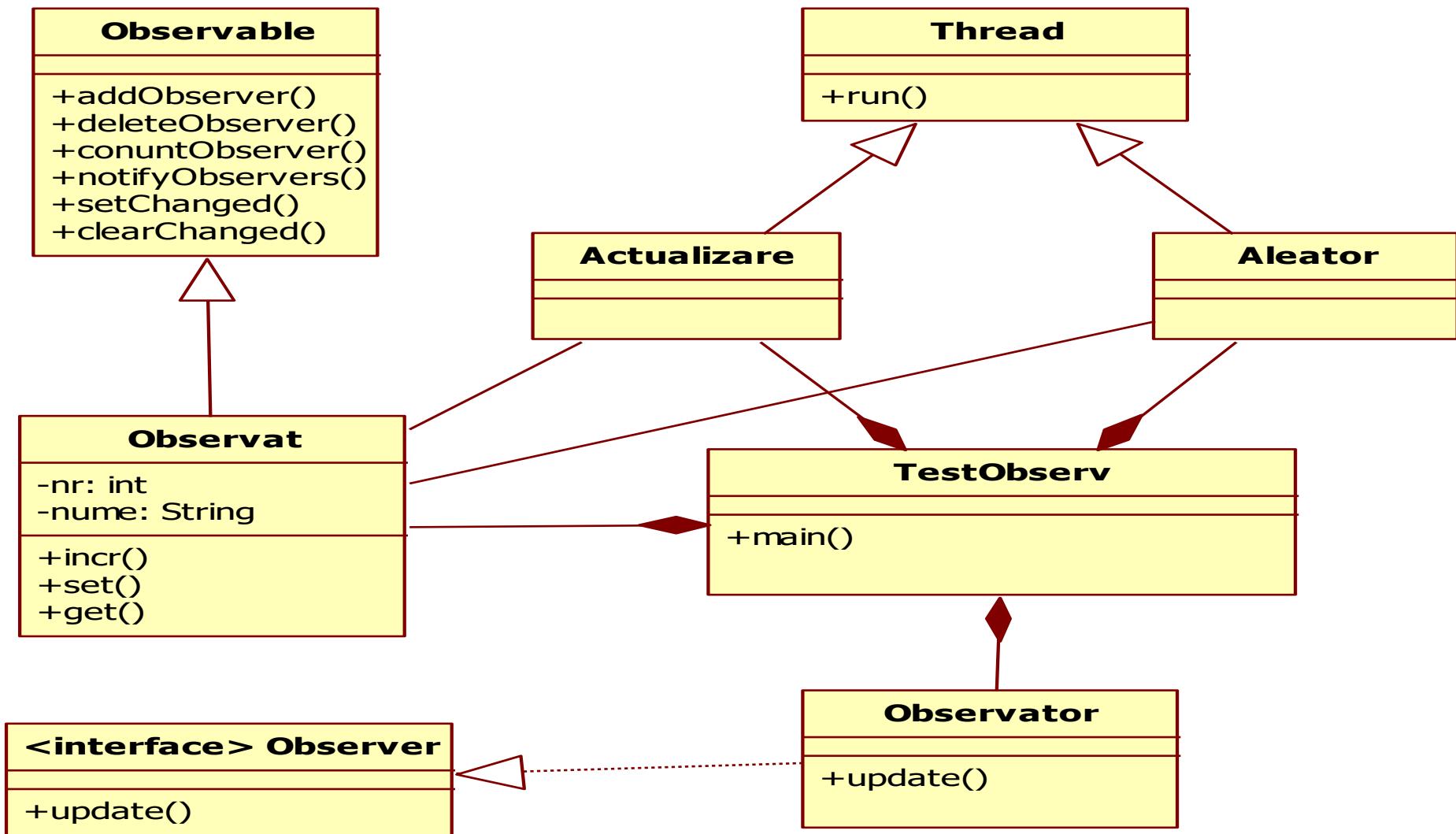
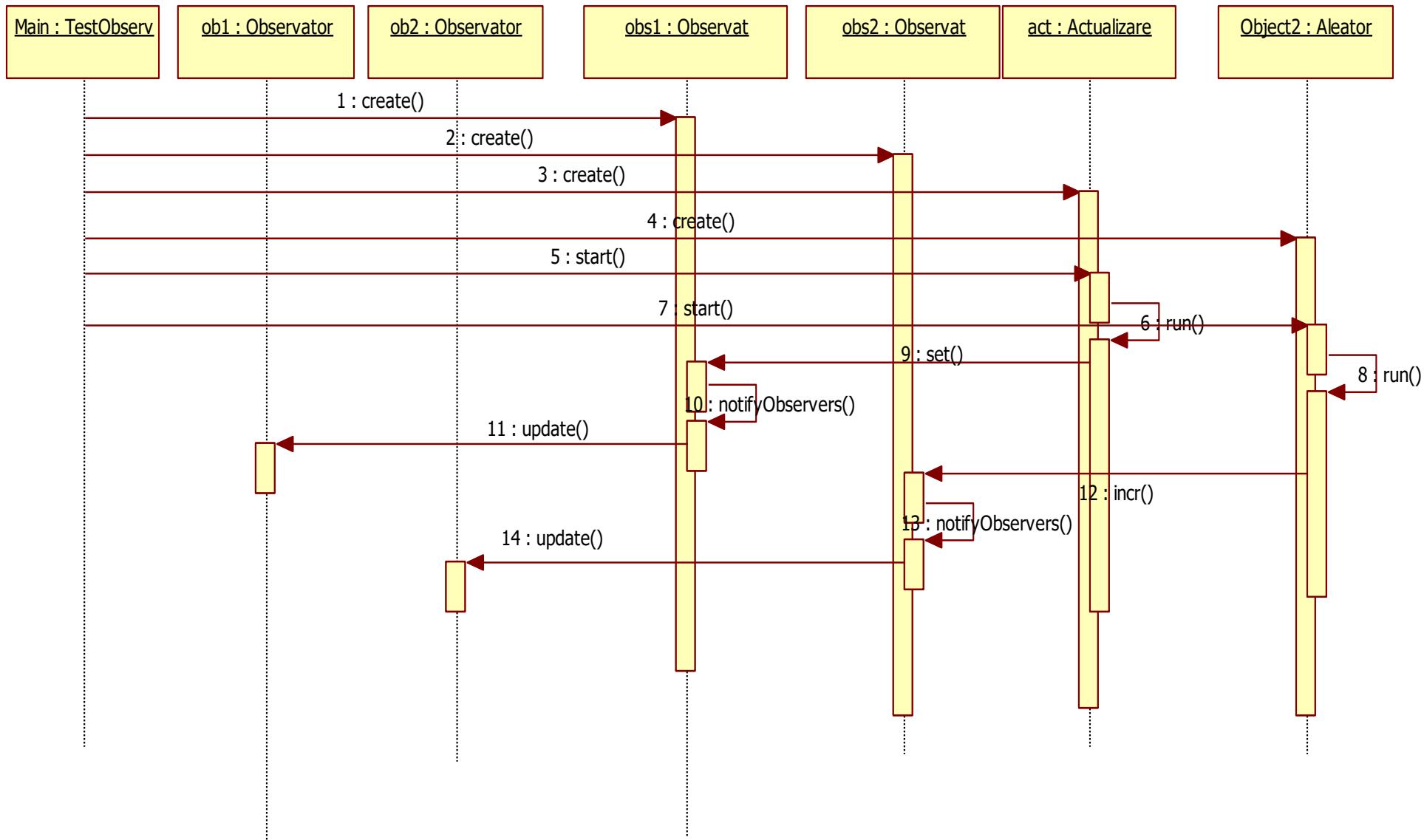


Diagrama claselor a aplicatiei de observare a schimbarilor dintr-un obiect



Homework: Draw the PN, ETPN and OETPN models of the Observer application.

→ reverse engineering!

8. Java packages for concurrency

`java.util`

`java.util.concurrent`

`java.util.concurrent.atomic` → classes for atomic primitiv data access

`java.util.concurrent.lock`

Package `java.util.concurrent`

`ConcurrentLinkedQueue<E>`

`CyclicBarrier`

`DelayQueue`

`Executors`

`LinkedBlockingQueue`

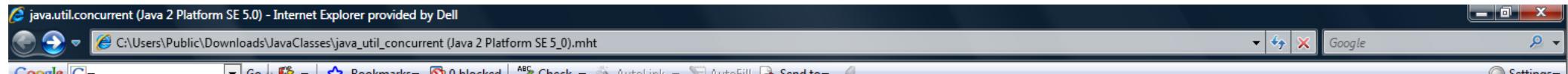
`PriorityBlockingQueue`

`ScheduledPoolExecutor`

`Semaphore`

`SynchronousQueue` – `take()` \leftrightarrow `put()` synchronized each other

`ThreadPoolExecutor`



Class Summary	
AbstractExecutorService	Provides default implementation of ExecutorService execution methods.
ArrayBlockingQueue<E>	A bounded blocking queue backed by an array.
ConcurrentHashMap<K,V>	A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates.
ConcurrentLinkedQueue<E>	An unbounded thread-safe queue based on linked nodes.
CopyOnWriteArrayList<E>	A thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.
CopyOnWriteArraySet<E>	A set that uses CopyOnWriteArrayList for all of its operations.
CountDownLatch	A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
CyclicBarrier	A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
DelayQueue<E extends Delayed>	An unbounded blocking queue of Delayed elements, in which an element can only be taken when its delay has expired.
Exchanger<V>	A synchronization point at which two threads can exchange objects.
ExecutorCompletionService<V>	A CompletionService that uses a supplied Executor to execute tasks.
Executors	Factory and utility methods for Executor , ExecutorService , ScheduledExecutorService , ThreadFactory , and Callable classes defined in this package.
FutureTask<V>	A cancellable asynchronous computation.
LinkedBlockingQueue<E>	An optionally-bounded blocking queue based on linked nodes.
PriorityBlockingQueue<E>	An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations.
ScheduledThreadPoolExecutor	A ThreadPoolExecutor that can additionally schedule commands to run after a given delay, or to execute periodically.
Semaphore	A counting semaphore.
SynchronousQueue<E>	A blocking queue in which each put must wait for a take , and vice versa.
ThreadPoolExecutor	An ExecutorService that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.
ThreadPoolExecutor.AbortPolicy	A handler for rejected tasks that throws a RejectedExecutionException .
ThreadPoolExecutor.CallerRunsPolicy	A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute method, unless the executor has been shut down, in which case the task is discarded.
ThreadPoolExecutor.DiscardOldestPolicy	A handler for rejected tasks that discards the oldest unhandled request and then retries execute , unless the executor is shut down, in which case the task is discarded.
ThreadPoolExecutor.DiscardPolicy	A handler for rejected tasks that silently discards the rejected task.

Enum Summary	
TimeUnit	A TimeUnit represents time durations at a given unit of granularity and provides utility methods to convert across units, and to perform timing and delay operations in these units.

Exception Summary	
BrokenBarrierException	Exception thrown when a thread tries to wait upon a barrier that is in a broken state, or which enters the broken state while the thread is waiting.
CancellationException	Exception indicating that the result of a value-producing task, such as a FutureTask , cannot be retrieved because the task was cancelled.

Class CyclikBarrier

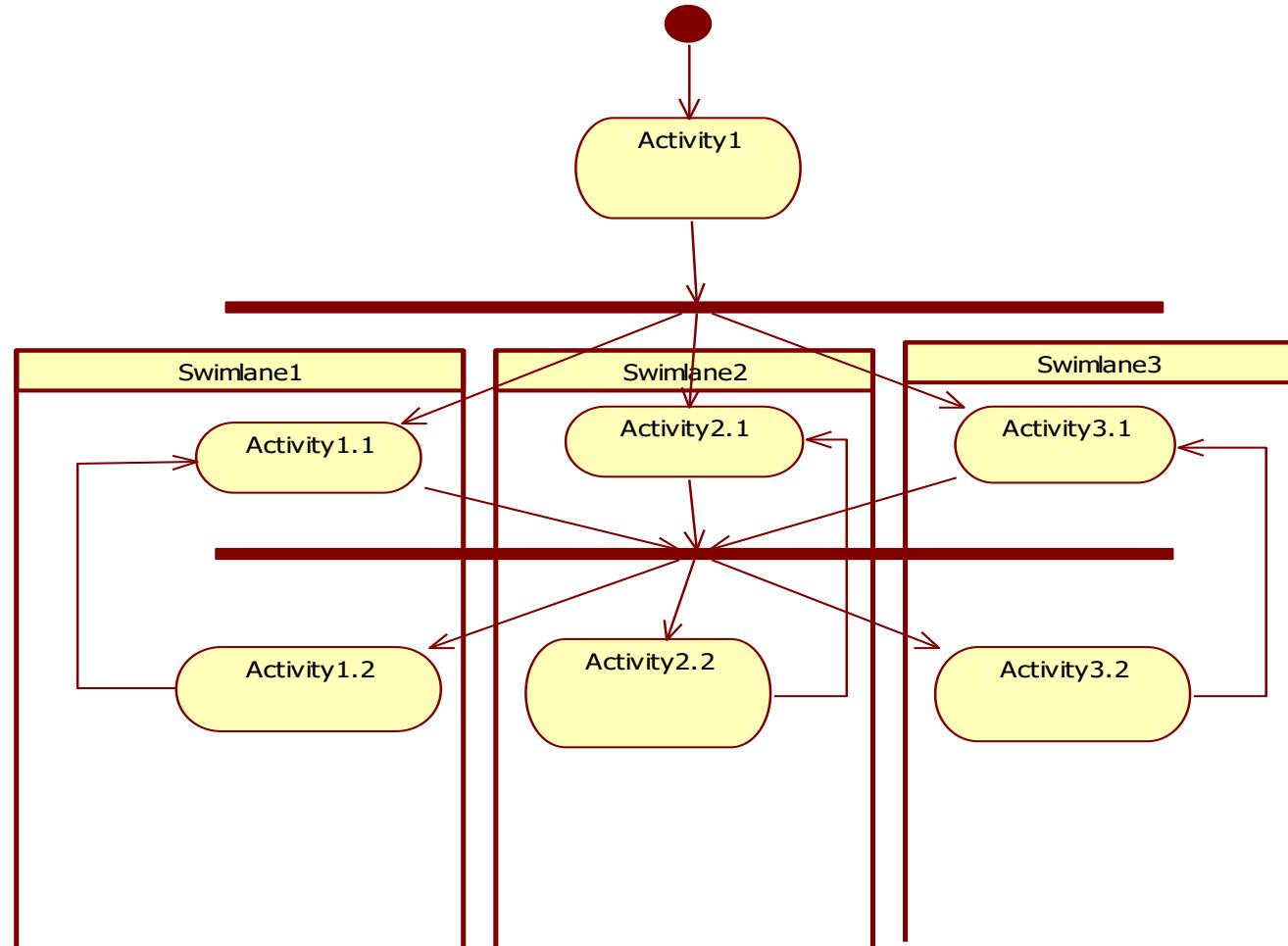
Constructor:

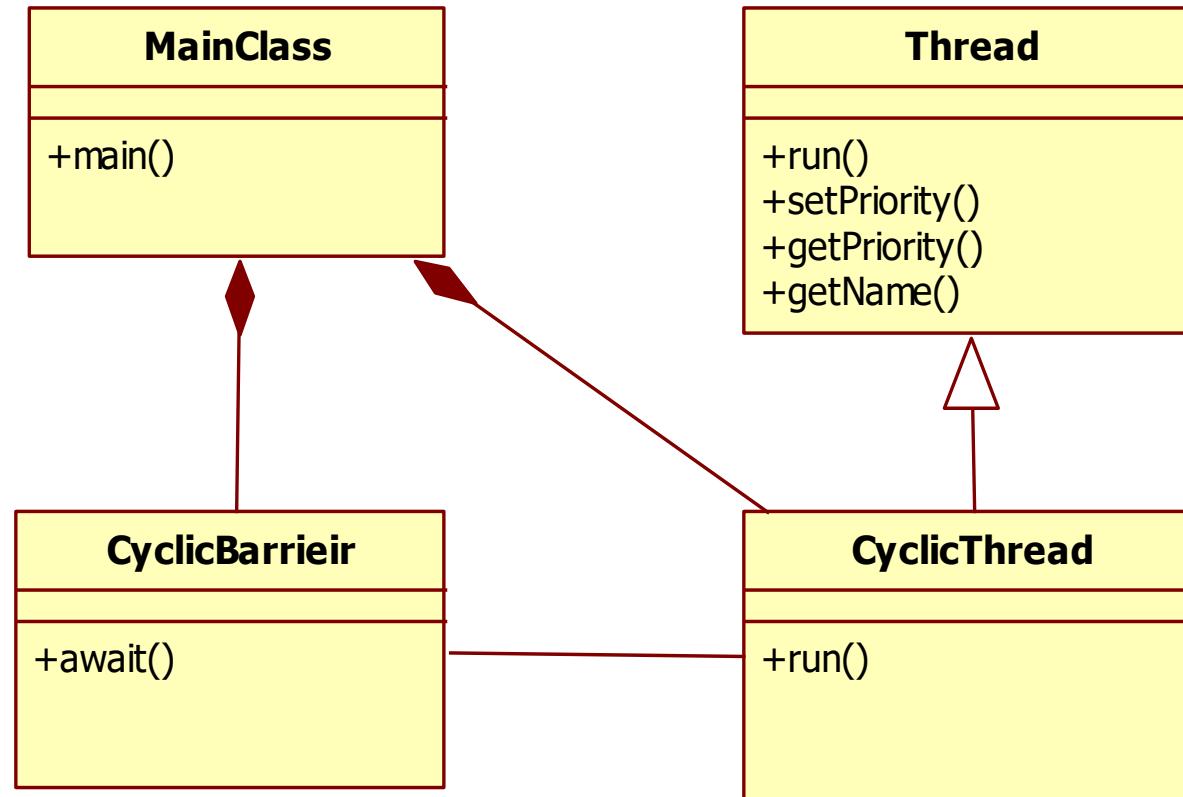
CyclicBarrier(int parties)

*CyclicBarrier(int parties,
Runnable r)*

Methods:

`public int await() throws
InterruptedException,
BrokenBarrierException`





Test1: Barrier without thread

```
import java.util.concurrent.*;  
public class CyclicThread extends Thread {  
    CyclicBarrier barrier;  
    long time;//periodic waiting time  
    int priority;  
    public CyclicThread(CyclicBarrier bar, long tm, int pr) {  
        barrier=bar;  
        time=tm;  
        priority=pr;  
        setPriority(priority);  
        this.start();  
    }  
    public void run() {  
        while(true){  
            System.out.println("Fir "+getName()+" performs Activity 1"+"  
priority="+getPriority());  
            try{  
                sleep(time);  
            }  
        }  
    }  
}
```

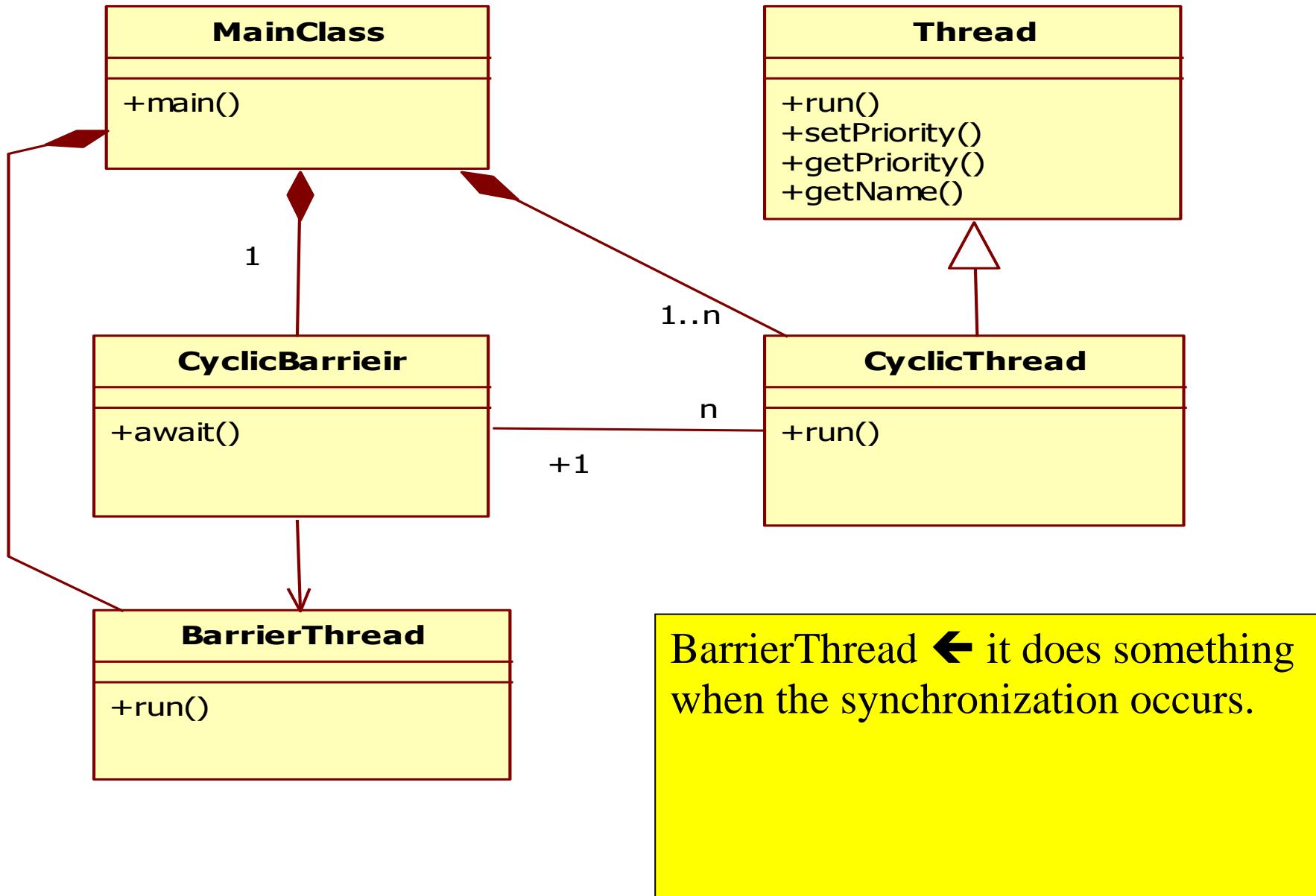
CyclicBarrier methods:

```
int wait(long timeout, TimeUnit unit)  
reset()  
boolean isBroken()  
int getParties()
```

```
        barrier.await();
    }catch(InterruptedException ex) {
        System.out.println("Fir "+getName()+" interupted");
    }catch(BrokenBarrierException be) {
        System.out.println("Barrier "+barrier.toString()+" is interupted=
"+barrier.isBroken());
    }
    System.out.println("Fir "+getName()+" performs Activity 2"+
priority="+getPriority());
}
}
```

```
import java.util.concurrent.*;
public class MainClass {

    public static void main(String[] args) {
        System.out.println("Fir "+Thread.currentThread().getName()+""
performs Activity 1");
CyclicBarrier b=new CyclicBarrier(3);
        CyclicThread ct1=new CyclicThread(b,1000,2);//Swimlane1
        CyclicThread ct2=new CyclicThread(b,2000,4);//Swimlane2
        CyclicThread ct3=new CyclicThread(b,3000,7);//Swimlane3
    }
}
```



Test2: Barrier with thread

```
public class BarrierThread extends Thread{
    boolean act=true;
    public void run() {
        System.out.println("Barrier proceeds");
    }
}

import java.util.concurrent.*;
public class MainClass {
    public static void main(String[] args) {
        System.out.println("Fir "+Thread.currentThread().getName()+" performs Activity 1");
        BarrierThread barThread=new BarrierThread();
        CyclicBarrier b=new CyclicBarrier(3,barThread);
        CyclicThread ct1=new CyclicThread(b,1000,2);
        CyclicThread ct2=new CyclicThread(b,2000,4);
        CyclicThread ct3=new CyclicThread(b,3000,7);
    }
}
```

Fir main performs Activity 1
Fir Thread-1 performs Activity 1 with priority=2
Fir Thread-2 performs Activity 1 with priority=4
Fir Thread-3 performs Activity 1 with priority=7
Barrier proceeds
Fir Thread-3 performs Activity 2 with priority=7
Fir Thread-3 performs Activity 1 with priority=7
Fir Thread-1 performs Activity 2 with priority=2
Fir Thread-2 performs Activity 2 with priority=4
Fir Thread-2 performs Activity 1 with priority=4
Fir Thread-1 performs Activity 1 with priority=2
Barrier proceeds

java.util.concurrent.Semaphore

-implements Serializable

Constructors:

Semaphore(int permits) – Creates a Semaphoren with a given number of permits and nonfairness setting

Semaphore(int permits, boolean fair) – Creates a Semaphoren with a given number of permits and with given fairness setting

 fair=true → FIFO

 fair=false → no FIFO gurantee

Methods:

void acquire() – blocking until one is available

void acquire(int permits) – blocking until one is available

void acquireUninterruptibly(int permits)

void release()

boolean tryAcquire()

boolean tryAcquire(int permits, long timeout, TimeUnit unit)

Homework: Use semaphores to implement a barrier.

Package java.concurrent.locks

Interface Summary

<u>Condition</u>	Condition factors out the Object monitor methods (wait , notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations.
<u>Lock</u>	Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.
<u>ReadWriteLock</u>	A ReadWriteLock maintains a pair of associated locks , one for read-only operations and one for writing.

Class Summary

<u>AbstractQueuedSynchronizer</u>	Provides a framework for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on first-in-first-out (FIFO) wait queues.
-----------------------------------	--

<u>LockSupport</u>	Basic thread blocking primitives for creating locks and other synchronization classes.
<u>ReentrantLock</u>	A reentrant mutual exclusion <u>Lock</u> with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.
<u>ReentrantReadWriteLock</u>	An implementation of <u>ReadWriteLock</u> supporting similar semantics to <u>ReentrantLock</u> .
<u>ReentrantReadWriteLock.ReadLock</u>	The lock returned by method <u>ReentrantReadWriteLock.readLock()</u> .
<u>ReentrantReadWriteLock.WriteLock</u>	The lock returned by method <u>ReentrantReadWriteLock.writeLock()</u> .

A **lock** is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a [ReadWriteLock](#).

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
} catch (...) .....
```

Lock interface

Method Summary

void	<u>lock</u> ()	Acquires the lock.
void	<u>lockInterruptibly</u> ()	Acquires the lock unless the current thread is <u>interrupted</u> .
<u>Condition</u>	<u>newCondition</u> ()	Returns a new <u>Condition</u> instance that is bound to this Lock instance.
boolean	<u>tryLock</u> ()	Acquires the lock only if it is free at the time of invocation.
boolean	<u>tryLock</u> (long time, <u>TimeUnit</u> unit)	Acquires the lock if it is free within the given waiting time and the current thread has not been <u>interrupted</u> .
void	<u>unlock</u> ()	Releases the lock.

```
Lock lock = ...;
if (lock.tryLock()) {
    try {
        // manipulate protected state
    } finally {
        lock.unlock();
    }
} else {
    // perform alternative actions
}
```

Class ReentrantLock

Constructor Summary

[ReentrantLock \(\)](#)

Creates an instance of ReentrantLock.

[ReentrantLock \(boolean fair\)](#)

Creates an instance of ReentrantLock with the given fairness policy.

Method Summary

int [getHoldCount \(\)](#)

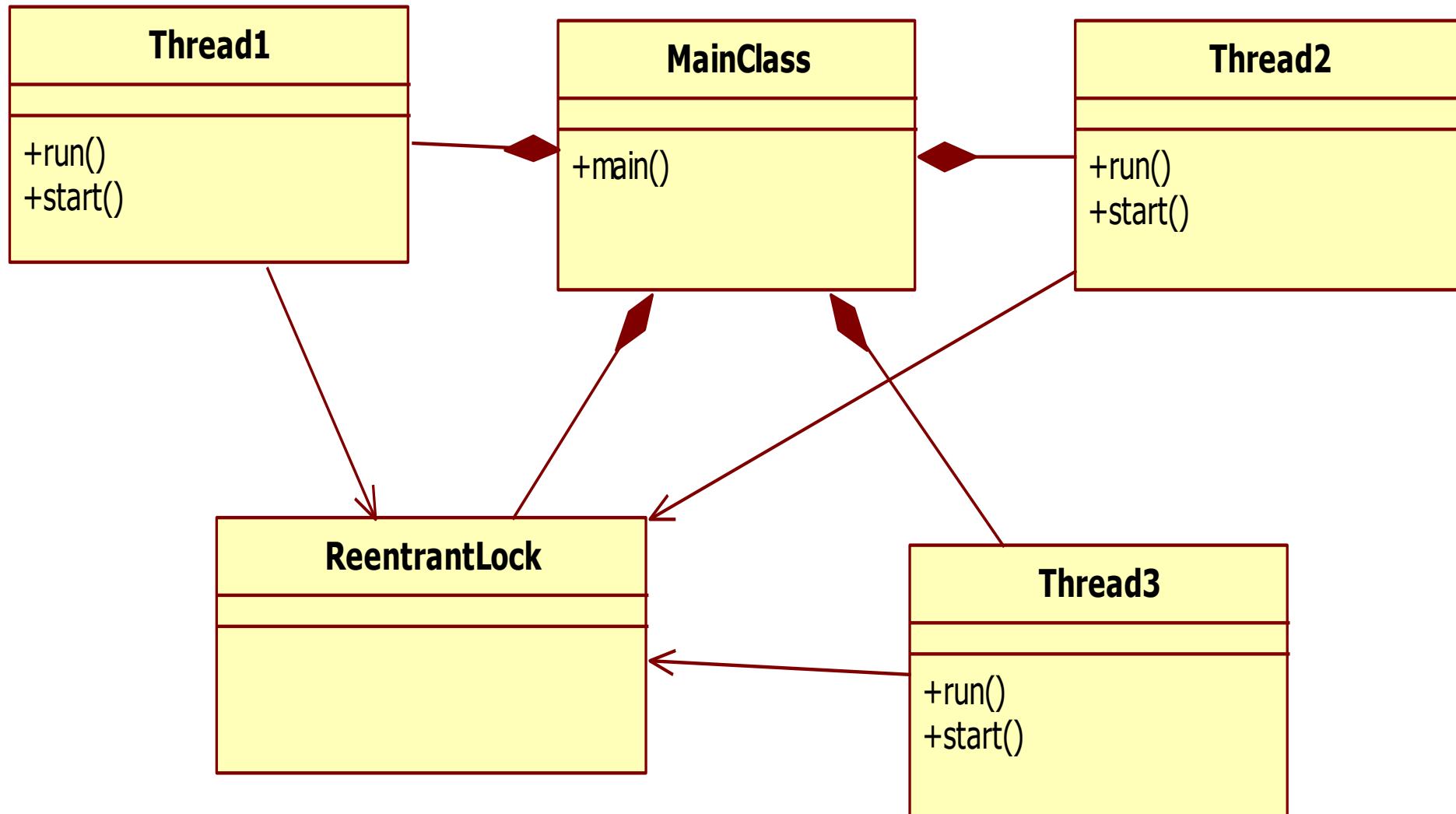
Queries the number of holds on this lock by the current thread.

protected Thread	<u>getOwner ()</u>	Returns the thread that currently owns this lock, or null if not owned.
protected Collection< Thread>	<u>getQueuedThreads ()</u>	Returns a collection containing threads that may be waiting to acquire this lock.
int	<u>getQueueLength ()</u>	Returns an estimate of the number of threads waiting to acquire this lock.
protected Collection< Thread>	<u>getWaitingThreads (Condition condition)</u>	Returns a collection containing those threads that may be waiting on the given condition associated with this lock.
int	<u>getWaitQueueLength (Condition condition)</u>	

		Returns an estimate of the number of threads waiting on the given condition associated with this lock.
boolean	<u>hasQueuedThread</u> (<u>Thread</u> thread)	Queries whether the given thread is waiting to acquire this lock.
boolean	<u>hasQueuedThreads</u> ()	Queries whether any threads are waiting to acquire this lock.
boolean	<u>hasWaiters</u> (<u>Condition</u> condition)	Queries whether any threads are waiting on the given condition associated with this lock.
boolean	<u>isFair</u> ()	Returns true if this lock has fairness set true.

	boolean <u>isHeldByCurrentThread()</u>	Queries if this lock is held by the current thread.
	boolean <u>isLocked()</u>	Queries if this lock is held by any thread.
	void <u>lock()</u>	Acquires the lock.
	void <u>lockInterruptibly()</u>	Acquires the lock unless the current thread is <u>interrupted</u> .
<u>Condition</u>	<u>newCondition()</u>	Returns a <u>Condition</u> instance for use with this <u>Lock</u> instance.

<code>String</code>	<u>toString()</u>	Returns a string identifying this lock, as well as its lock state.
<code>boolean</code>	<u>tryLock()</u>	Acquires the lock only if it is not held by another thread at the time of invocation.
<code>boolean</code>	<u>tryLock(long timeout, TimeUnit unit)</u>	Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been <u>interrupted</u> .
<code>void</code>	<u>unlock()</u>	Attempts to release this lock.



```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.concurrent.locks.*;
public class Thread1 extends Thread {
    boolean stop=false;
    ReentrantLock lck;
    long mil;
    public Thread1(ReentrantLock loc){
        lck=loc;
        this.setName("Fir1");
        this.start();
    }
    public void run() {
        while(!stop) {
            lck.lock();
            mil=getTime();
            System.out.println("Lock acquired by:
"+currentThread().getName()+" at "+mil);
        }
    }
}
```

```
try{Thread.sleep(1000);}catch(InterruptedException ex) { }
lck.unlock();
mil=getTime();
System.out.println("Lock released by:
"+currentThread().getName()+" at "+mil);
try{Thread.sleep(2000);}catch(InterruptedException ex) { }
}
}

public long getTime(){
    Calendar calendar=new GregorianCalendar();
    long time=calendar.getTimeInMillis();
    return time;
}
}
```

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.TimeUnit;
public class Thread3 extends Thread {
    boolean stop=false,b;
    ReentrantLock lck;
    long mil;
    public Thread3(ReentrantLock loc){
        lck=loc;
        this.setPriority(9);
        this.setName("Fir3");
        this.start();
    }
    public void run() {
        while(!stop) {
            try{
                b=lck.tryLock(1111, TimeUnit.MILLISECONDS);
```

```

mil=getTime();
if(b) System.out.println("Lock acquired by:
    "+currentThread().getName()+" at "+mil);
else System.out.println("Lock acquisition failed by: "+
    currentThread().getName()+" at "+mil);
Thread.sleep(2000);
}catch(InterruptedException e){ }
if(lck.isHeldByCurrentThread()){
    mil=getTime();
    System.out.println("Lock released by:
        "+currentThread().getName()+" at "+mil);
    lck.unlock();
}
else System.out.println("Lock was not occupied by:
    "+currentThread().getName());
}
public long getTime(){

```

```
    Calendar calendar=new GregorianCalendar();
    long time=calendar.getTimeInMillis();
    return time;
}
}
```

```
import java.util.concurrent.locks.*;
public class MainClass {
    public static void main(String[] args) {
        while(true){
            ReentrantLock rl=new ReentrantLock();
            Thread1 t1=new Thread1(rl);
            Thread2 t2=new Thread2(rl);
            Thread3 t3=new Thread3(rl);
            try{Thread.sleep(1000);}catch(InterruptedException ex) { }
            System.out.println("Main: Lock waited by "+rl.getQueueLength()+" threads");
        }
    }
}
```

Lock acquired by: Fir1 at 1196175029502
Main: Lock waited by 2 threads
Lock acquired by: Fir3 at 1196175030501
Lock released by: Fir1 at 1196175030503
Lock acquired by: Fir2 at 1196175030503
Lock acquisition failed by: Fir3 at 1196175030613
Main: Lock waited by 2 threads
Lock acquired by: Fir1 at 1196175031501
Lock released by: Fir2 at 1196175031503
Lock released by: Fir3 at 1196175032501
Lock acquired by: Fir3 at 1196175032501
Lock acquired by: Fir3 at 1196175032501
Main: Lock waited by 1 threads
Lock released by: Fir1 at 1196175032501
Lock acquired by: Fir3 at 1196175032502
Lock acquired by: Fir1 at 1196175032503
Lock was not occupied by: Fir3
Main: Lock waited by 2 threads

```
Lock released by: Fir1 at 1196175033503
Lock acquired by: Fir3 at 1196175033503
Lock acquired by: Fir1 at 1196175033503
Lock released by: Fir3 at 1196175034501
Lock released by: Fir3 at 1196175034501
Lock acquired by: Fir3 at 1196175034501
Lock acquired by: Fir3 at 1196175034501
Lock released by: Fir3 at 1196175034502
Lock acquired by: Fir3 at 1196175034502
```

9. Implementation of OETPN models

OETPN model single thread implementation was presented.

Current approach: OETPN models asynchronous implementation

Development from PN to OETPN

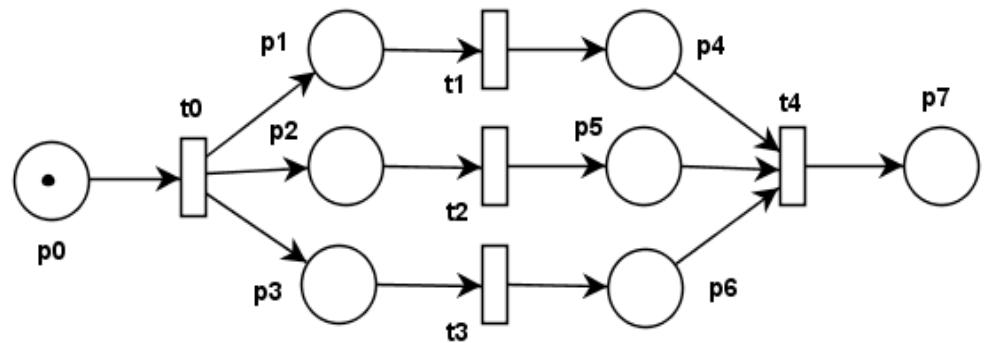
There are some differences between Petri nets (PNs) representation, formal descriptions and implementations.

If we make a ZOOM, the differences disappear. Some nodes are merged (often) in PNs for making them easier understandable.

Concurrent implementation

The attached PN has the *PNL* (*Petri Net based Language*) formal description:

$$\sigma = t0 * (t1 \& t2 \& t3) * t4$$



TPNL (*Time Petri Net based Language*) description:

$$\sigma = t0[a0] * (t1[a1] \& t2[a2] \& t3[a3]) * t4[a4]$$

where a_i ($i=1,2,\dots$) is a pure delay.

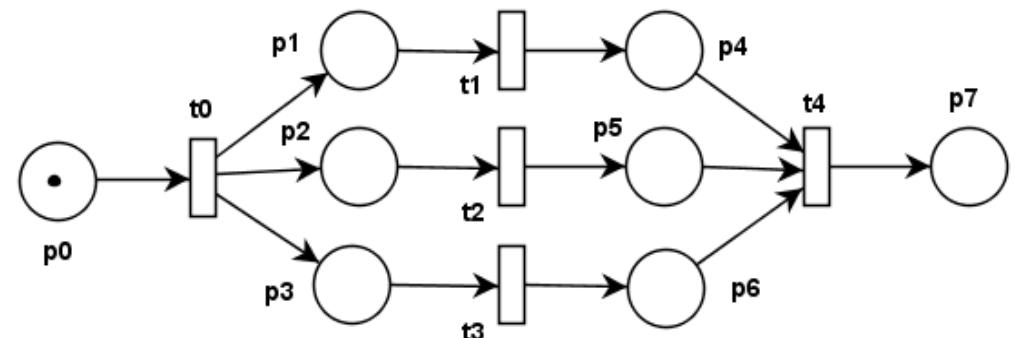
TIPNL (*Time Interval Petri Net based Language*)

$$\sigma = t0[e0,l0] * (t1[e1,l1] \& t2[e2,l2] \& t3[e3,l3]) * t4[e4,l4]$$

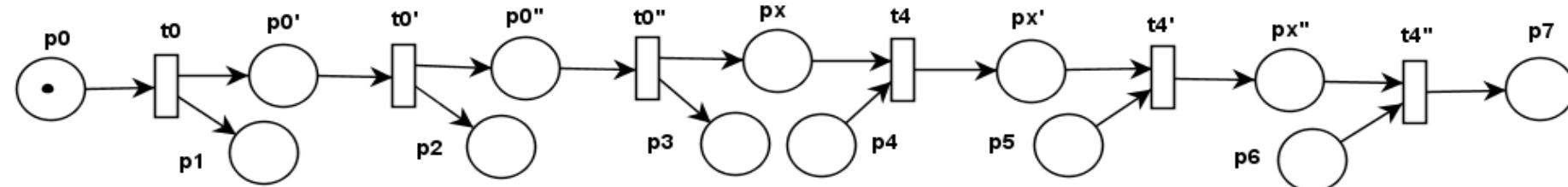
where $[e_i, l_i]$ is a time interval with e_i earliest (start of the time interval) and l_i latest (end of the time interval)

ETPNL (Enhanced Time Petri Net based Language) with the notations:

- $t_k[i,o]$ where i is an input channel and o an output channel;
- $t_k[5,o]$ where 5 is a delay and o an output channel;
- $t_k[\varphi,o]$ where φ means the lack of an input event (external or clock event) and o an output channel
- $t_k[i,\varphi]$ where φ means the lack of an output event;



The corresponding ETPNL description of the above model is:



$$\sigma = t0[\varphi, p1] * t0'[\varphi, p2] * t0''[\varphi, p3] * (t1[\varphi, \varphi] \& t2[\varphi, \varphi] \& t3[\varphi, \varphi]) * t4[p4, \varphi] * t4'[p5, \varphi] * t4''[p6, \varphi]$$

Tasks:

T0: $t0$ in PNL or $t0[\varphi, p1] * t0'[\varphi, p2] * t0''[\varphi, p3] * t4[p4, \varphi] * t4'[p5, \varphi] * t4''[p6, \varphi]$ in ETPNL

T1: $t1$ in PNL and $t1[\varphi, \varphi]$ in ETPNL etc.

OETPNL

t0: grd() {p0 ≠ φ} \there is the token in p0 it is equivalent to m(p0)=1 in the classical PN
map() {p1.create(T1); T1.start(); p0'.setToken();}

\The transition creates a task of type *T1* in *p1* and starts it, then sets a token in *p0'*.

In OETPN φ means the lack of token (no activated or referred object) in the mentioned place at the current moment of time.

Implementation:

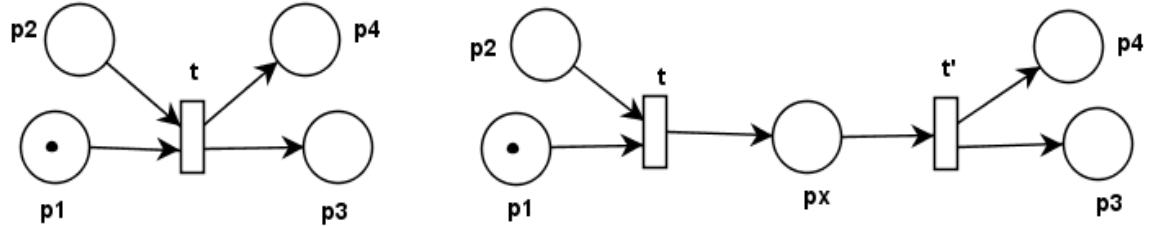
\T0:

```
run() {           \ main() ?  
    T1 task1=new T1();  
    T2 task2=new T2();  
    T3 task3=new T3();  
    task1.start();  
    task2.start();  
    task3.start();  
    task1.join();  
    task2.join();  
    task3.join();  
}
```

```
\ task T1  
run() {  
    execute t1;  
}
```

I/O operations

$$t[p2,p4] \rightarrow t[p2,\varphi]^* t'[\varphi,p4]$$



Observation: they are executed sequentially.

The implementation is:

read(p2); \\ or input(p2); receive(p2); etc.

write(p4); \\ or print(p4); send(p4); etx.

If the information is stored in the object referred by px , this can be implemented by
 $px.value = read(p2)$

OETPN Model Asynchronous Approach Implementation

The OETPN models that met the synchronous approach condition can be implemented by a single thread or multiple threads. Those that do not fulfill this condition, need the multithreading (i.e. multitasking) implementation.

The OETPN presented in the attached figure has the formal ETPNL description:

$$\sigma = t0^*(t1[e1,\varphi]\#t3[5,e3])\&(t2[e2,\varphi]\#t4[5,e3]))$$

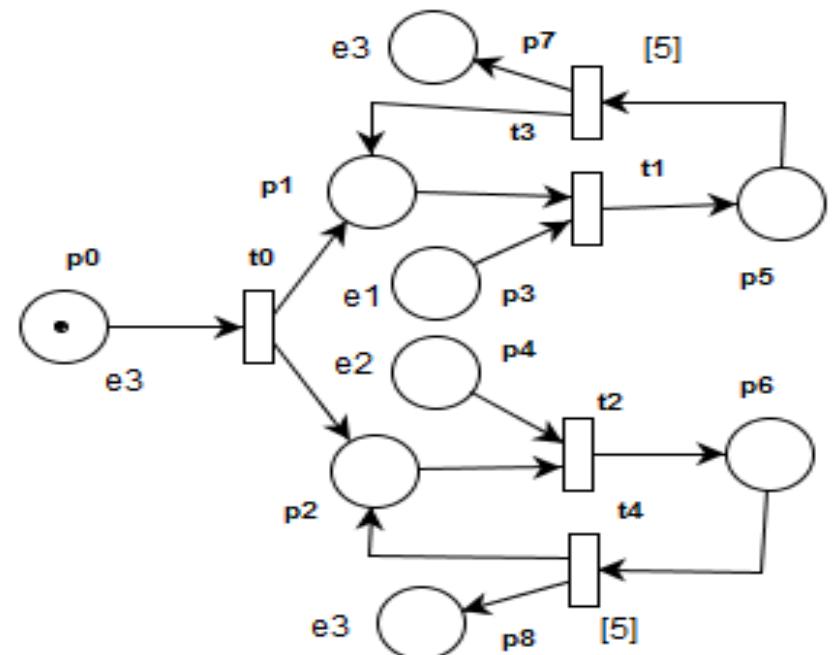
Place types are:

$$type(p0)= Clp0;$$

$$type(p1)= Clp1;$$

.....

$$type(p8)= Clp8;$$



Transition evolution relations are:

- t1: $grd_1(Cl_{p1}, Cl_{p3})$; $map_1(Cl_{p1}, Cl_{p3}, Cl_{p5})$;
- $grd_2(Cl_{p1}, Cl_{p3})$; $map_2(Cl_{p1}, Cl_{p3}, Cl_{p5})$;
- t3: $grd=true$; $map(Cl_{p5}, Cl_{p1}, Cl_{p7})$; $eet()=let()=5$;
-

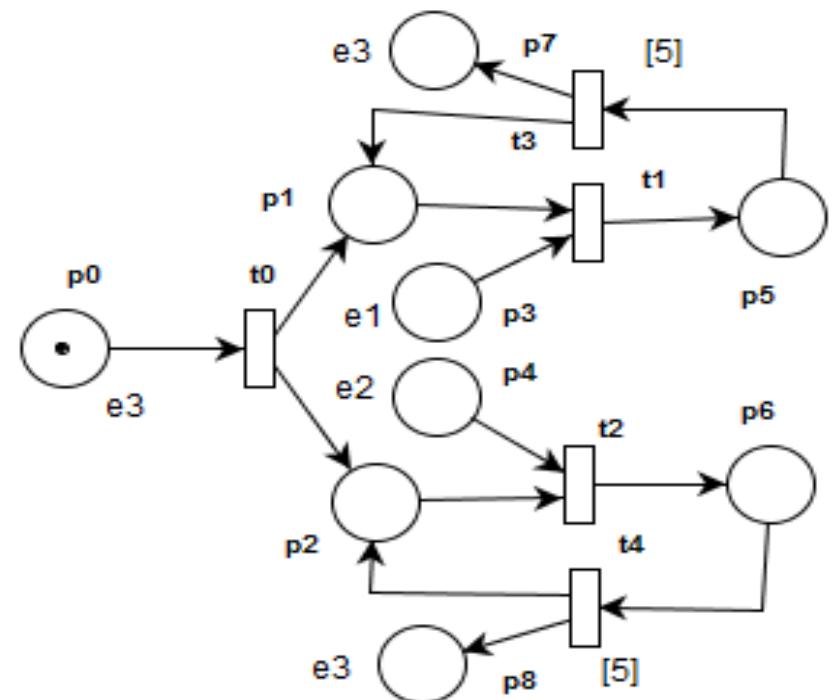
Implementation recommendation:

Each transition tk ($k=0, 1, \dots$) is implemented in class Cltk endowed with the methods $grd()$, $map()$, $eet()$ and $let()$.

It can be implemented by 3 or 2 threads (i.e. tasks).

Tasks:

- T1: $t0$
- T2: $t1[e1, \varphi] \# t3[5, e3]$
- T3: $t2[e2, \varphi] \# t4[5, e3]$

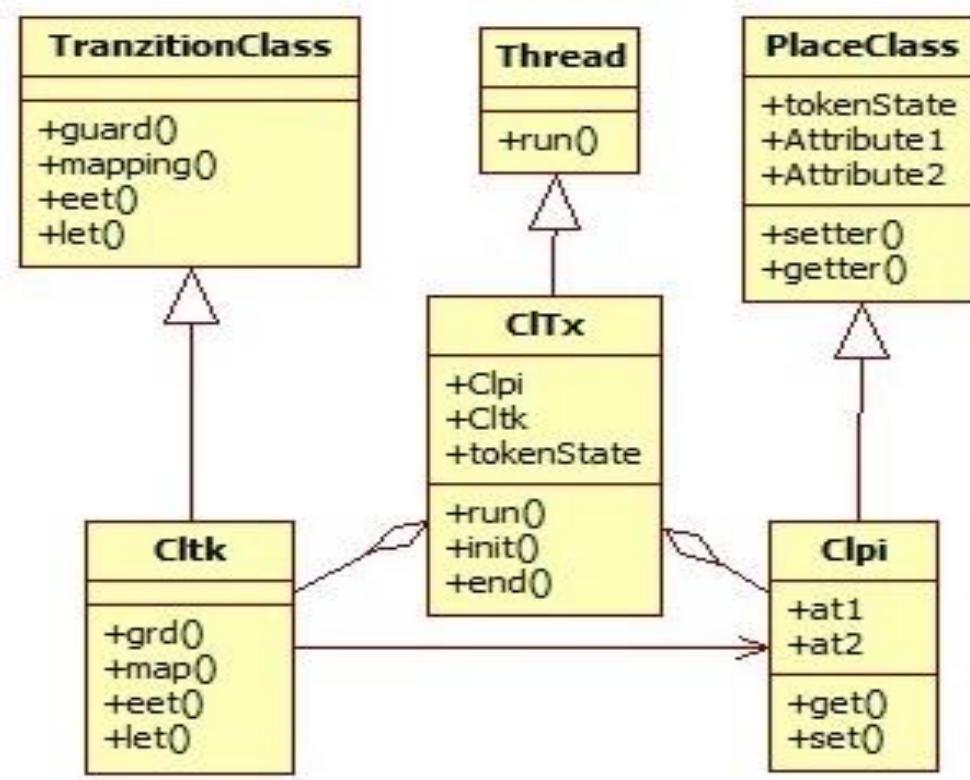


CIT1 (mainClass) \task T1

```
{  
    main()  
    {  
        CIT2 T2=new CIT2();  
        CIT3 T3=new CIT3();  
        T2.init(); \\ task initialization  
        T2.start();  
        T3.start();  
    }  
}
```

CIT2 extends Thread {\task T2; T3 is similar
boolean tokenState;

```
    init() {}  
    end() {}  
    run() { tokenState=true;  
        Create places  
        Clp1 p1=new Clp1();  
        marking init() method could be required  
        Clp2 p2=new Clp2();  
        .....  
    }
```



```
T1: t0  
T2: t1[e1,φ]#t3[5,e3]  
T3: t2[e2,φ]#t4[5,e3]
```

\Create transitions

Clt1 t1=new Clt1();

Clt2 t2=new Clt2();

.....

\execute periodically the evolution relations

while(true) {

 read(p3); \from keyboard – wait for information

 if(t1.grd_1(p1,p3)) then t1.map_1(p1,p3,p5);

 if(t1.grd_2(p1,p3)) then t1.map_2(p1,p3,p5);

 wait(5);

 t3.map(p5); \set(p1); set(p7) print on screen

}

 tokenState=false;

}

}

Inter-task communication

Tasks:

$T0: t1[\varphi, p6]$

$T1: t2[p3, p5] \# t3[\varphi, \varphi]$

$T2: (t4[\varphi, \varphi] * t5[p5, \varphi]) \# t6[\varphi, p9]$

a) **Synchronization with unbounded time wait**

- based on Semaphore

$t2 \rightarrow t5; \backslash\backslash$

$ClT0 \{$

 main() {

 Semaphore sem=new Semaphore(0);

 ClT1 T1=new ClT1(sem);

 ClT2 T2=new ClT2(sem);

 }

}

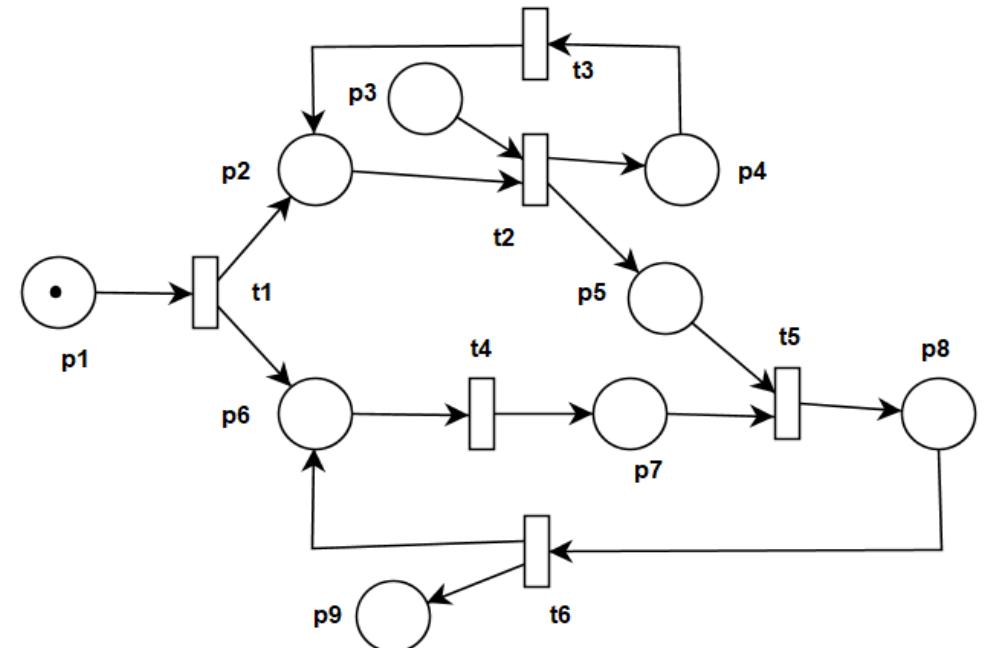
$ClT1$ extends Thread{

 boolean tokenState;

 Semaphore semRef;

 ClT1(Semaphore sem){

 semRef=sem;



$ClT2$ extends Thread {

.....

run() {

$\backslash\backslash t4$

If($t5.grd(p5, p7)$ then $sem.acquire()$;

.....

}

}

```

}
run() { tokenState=true;
    while(true) {
        read(p3); \\ from keyboard - wait
        if(t2.grd(p2,p3)) then
            {sem.release(); t2.map(p2,p3,p4);}
            .....
    }
    tokenState=false;
}

```

b) Synchronization with bounded time wait

The waiting thread (T2) executes:

```

i=0;
while (i<b) { \wait maximum b t.u.
    wait(1);
    if(t5.grd(p7,p5)) then i=b;
    i++;
}

```

c) (synchronization with) *Bounded time asynchronous wait*

```
CIT0 {  
    main() {  
        ReentrantLock reel=new ReentrantLock();  
        CIT1 T1=new CIT1(reel);  
        CIT2 T2=new CIT2(reel);  
    }  
}
```

```
CIT1 extends Thread{  
    Reentrant rL;  
    CIT1(ReentrantLock reel){  
        rL=reel;  
    }  
    run() {  
        while(true) {  
            rL.lock();  
            read(p3); \\ from keyboard - wait  
            if(t2.grd(p2,p3)) then  
                {rL.unlock(); map(p2,p3,p4);}  
            .....  
        }  
    }  
}
```

```
CIT2 extends Thread {  
    .....  
    run() {  
        \\t4 ....  
        \\try to get info no longer than b t.u.  
        If(t5.grd(p7) then rL.tryLock(b);  
        rL.unlock();  
        .....  
    }  
}
```

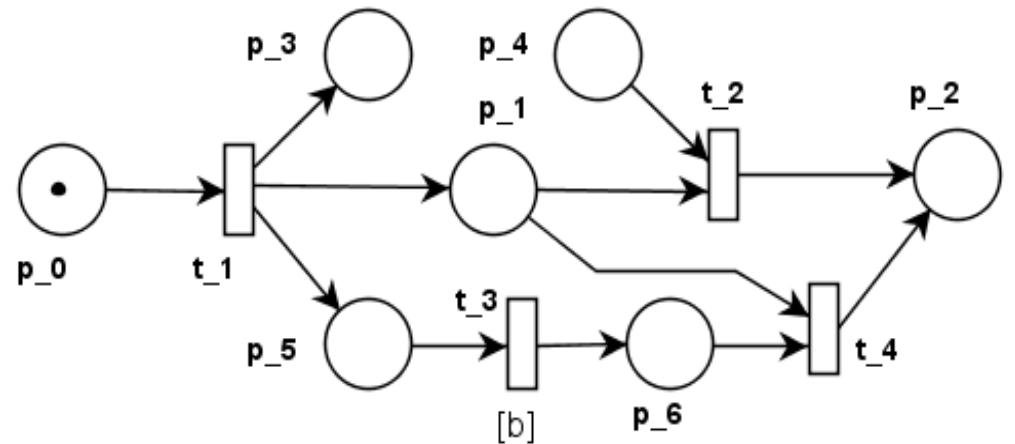
}

}

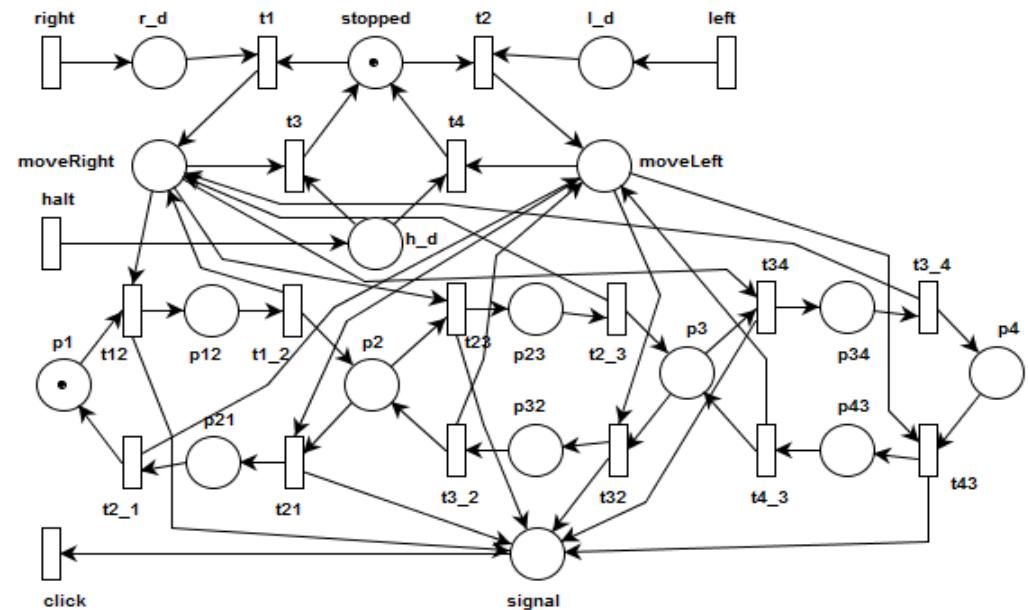
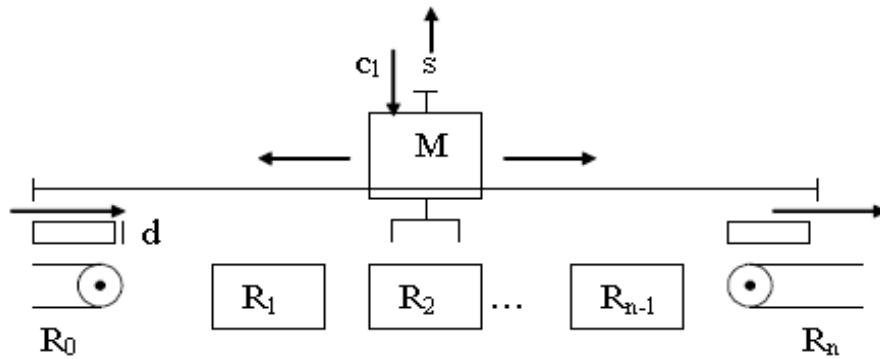
d) Bounded asynchronous wait with side thread (see the attached figure)

A new thread is launched that set a token in the place p_6 after b t.u.

p_3 request the information (or signal the reaching of the wait state and p_4 is used to catch asynchronously the event or information.



OETPN: Tema 1



Se consideră linia de galvanizare din subcapitolul *Specification-2.4.* pentru care s-a dat modelul ETPN din figura alăturată. Robotul mobil M ia câte un container de pe banda transportoare R_0 , îl pune în resursele hardware (bazinele) R_1, \dots, R_{n-1} conform cerințelor date de tehnologie. În final, containerul ajunge pe banda transportoare R_n și dispare din instalație. Ordinea de parcuregere a resurselor este dată de tehnologia curentă.

Varianta 1: se cunosc timpii de deplasare a lui M de la o resursă la alta.

Varianta 2: trebuie să se monteze câte un senzor pe resurse pentru a semnaliza (evenimentul) poziționarea lui M în dreptul resursei.

Se cere să se conceapă un model OETPN al instalației care este echivalent funcțional cu modelul ETPN (prezentat în diagrama alăturată).

Pentru modelul OETPN se cer: structura, tipurile locațiilor și relațiile de evoluție (grd, map, eet, let) ale tranzițiilor.

Criteriul de evaluare: un model OETPN al instalației este mai bun decât altul (OETPN) dacă are un număr mai mic de noduri, desă descrie complet comportamentul instalației.

Soluție descrisă parțial

$type(p_i1) = type(p_1) = type(p_o1) = type(p_i2) = \{R_k; int\}$

$type(p_2) = \{(R_k, R_{k+1}): (int, int)\}$

$t_1: p_i1.x = p_1.y \rightarrow map(p_2) = (x, y); map(p_i1) = \varphi$

$p_i1.x \neq p_1.y \rightarrow map(p_2) = (x, y);$

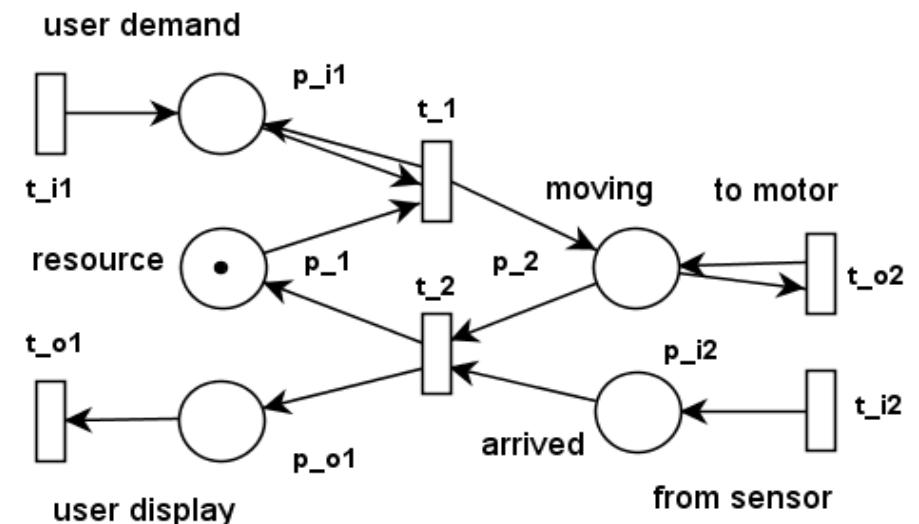
$t_2: (m(p_2) \neq \varphi) \& (m(p_i2) \neq \varphi) \rightarrow map(p_1): p_1.x = p_i2.x$
 $map(p_o1): p_o1.x = p_i2.x$

$t_o2:$

$p_2.x = p_2.y \rightarrow halt \leftarrow 0$

$p_2.x > p_2.y \rightarrow right \leftarrow +1$

$p_2.x < p_2.y \rightarrow left \leftarrow -1$



$t_i2: map(p_i2)=r;$
 $\textcolor{blue}{\backslash r}$ is the resource index

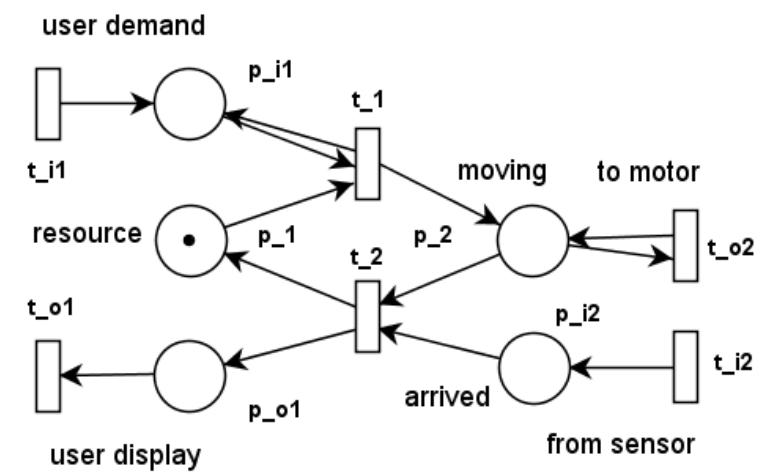
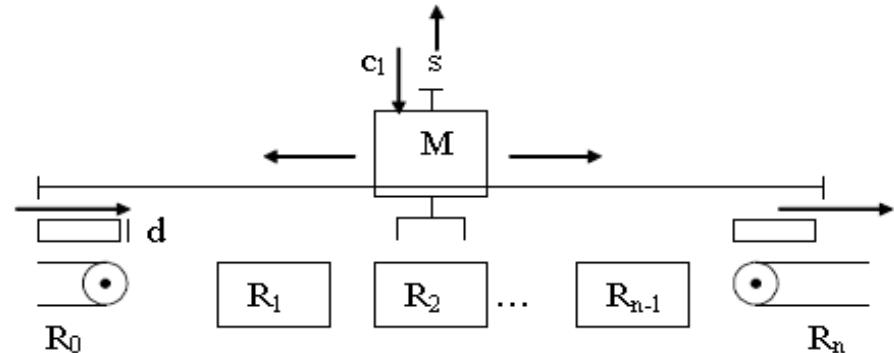
OETPN: Tema 2

Se cere să se construiască o aplicație de control a liniei de galvanizare reprezentată prin modelul OETPN determinat anterior. Aplicația de control primește de la utilizator o tehnologie descrisă printr-o secvență de forma:

$$\sigma = R_0[0,0] * R_i[eet_i, let_i] * R_k[eet_k, let_k] * R_j[eet_j, let_j] * \dots * R_n[0,0]$$

unde R_k este resursa unde trebuie pus containerul, eet_k reprezintă durata de timp minimă cât trebuie să stea în R_k , iar let_k reprezintă durata de timp maximă până la care containerul trebuie mutat în următoarea resursă.

Se cer arhitectura sistemului dată prin diagrama de componente, iar pentru supervisorul sintetizat: structura, tipurile locațiilor și relațiile de evoluție (comportament).



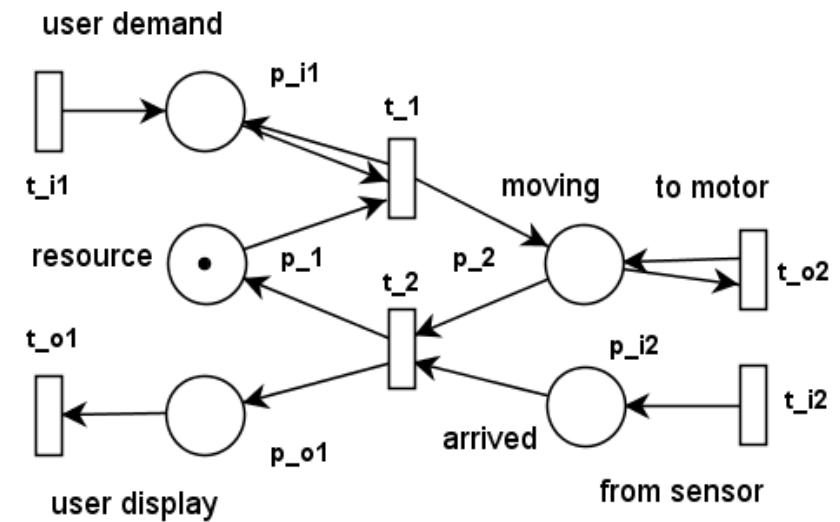
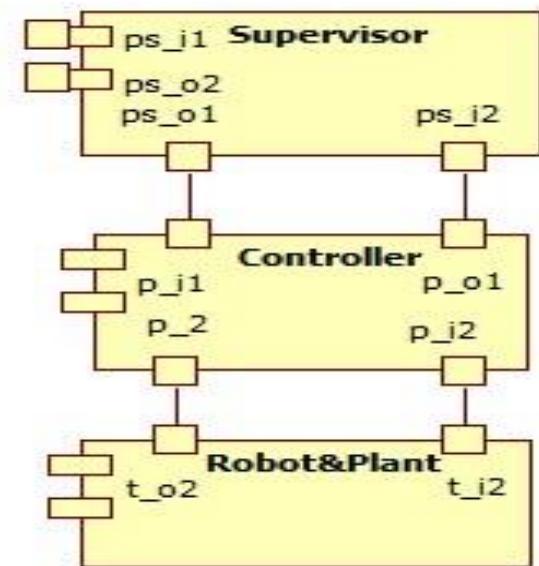
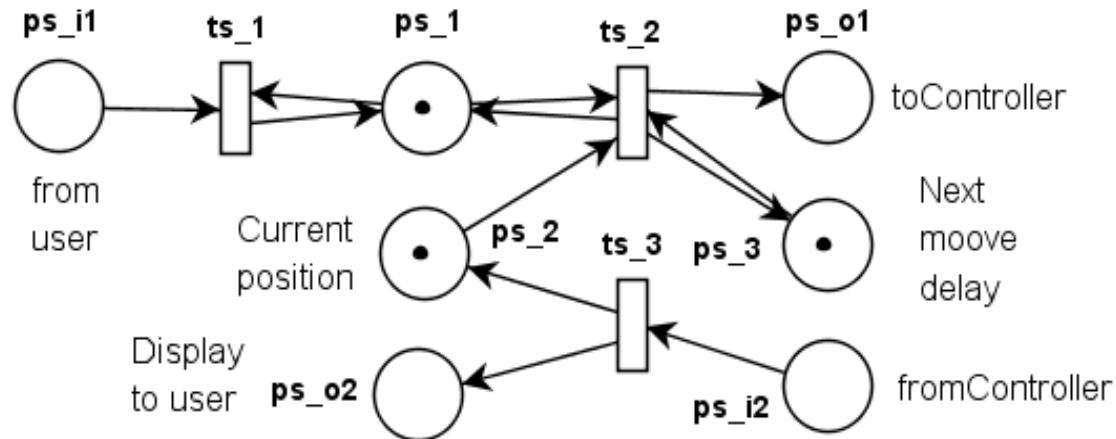
Solution

The Architecture is composed of *Supervisor* and *Controller*.

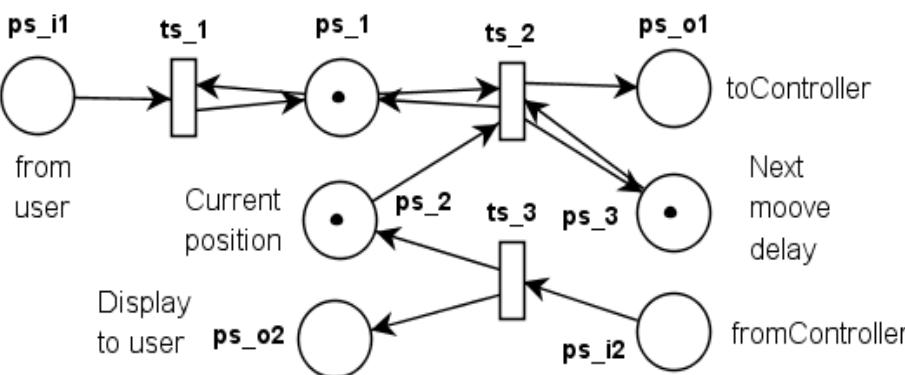
Supervisor receives demands (Resource, duration) from user and add them to sequence of activities.

Supervisor transforms the sequence

$\sigma = R_0[0,0]*R_i[eet_i, let_i]* R_k[eet_k, let_k]* R_j[eet_j, let_j]* \dots *R_n[0,0]$
in demands that are sent periodically and have to be executed by
Controller.



Controller



Supervisor gets from the user through input channel ps_i1 a new resource ,r' where the robot has to move with the minimum ,e' and maximum ,l' times specifying the time interval requested for robot to rest there.

$type(ps_i1) = Cl_ps-i1 \{int r; int e, int l\}$

$type(ps_1) = Cl_ps-1$ extends $List$; [\it](#) has methods:

$addElement()$, $getFirst()$;

$type(ps_2)$

$type(ps_o1)=type(ps_ps_3)=type(ps_i2)=type(ps_o2)=int;$

$init(): \{ps_2.v=0; ps_3.delay=0; ps_o1=\varphi; ps_1.list \}$

According to solution of Tema 1, *Controller* receives a demand to move the robot from the current position to another position.

Controller
confirms the performing of the demand.

Controller

$type(p_i1) = type(p_1)=$
 $type(p_o1)=type(p_i2)=\{R_k; int\}$
 $type(p_2) = \{(R_k, R_{k+1}): (int,int)\}$
 $t_1:$

$p_i1.x = p_1.y \rightarrow map(p_2) = (x,y);$

$map(p_i1) = \varphi$

$p_i1.x \neq p_1.y \rightarrow map(p_2) = (x,y);$

$t_2:$

$(m(p_2) \neq \varphi) \& (m(p_i2) \neq \varphi) \rightarrow$

$map(p_1): p_1.x=p_i2.x$

$map(p_o1): p_o1.x=p_i2.x$

$t_o2:$

$p_2.x = p_2.y \rightarrow halt$

$p_2.x > p_2.y \rightarrow right \leftarrow +1$

$p_2.x < p_2.y \rightarrow left \leftarrow -1$

ts_1: add element received in ps_i1 to the List stored in ps_1. ↵ It has to be an independent task.

ts_2: gets the first element of List and extarct the resource ,r'. If it is different of the robot current position stored in ps_2, a transition mapping set ,r' in ps_o1 and the element ,e' value in ps_3.

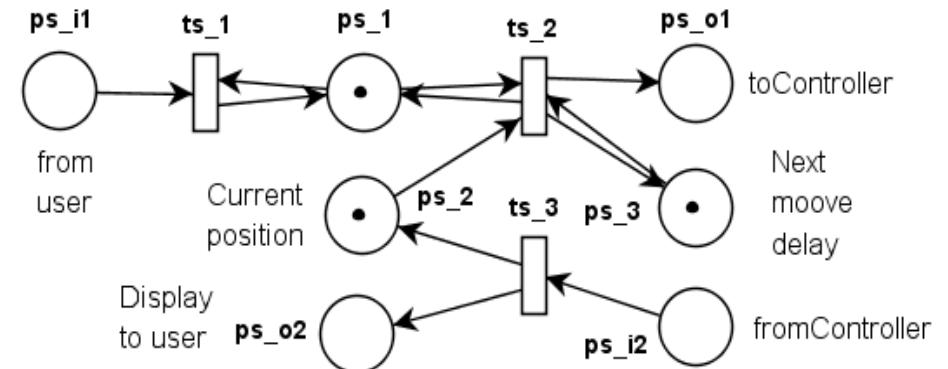
Question: The delay is for ts-1[e]? Not a random delay between e and l :ts-1[e,l]?

Answer: There is the message sent by user from the keyboard. The user register one by one an element (r,e,l) where the robot has to go.

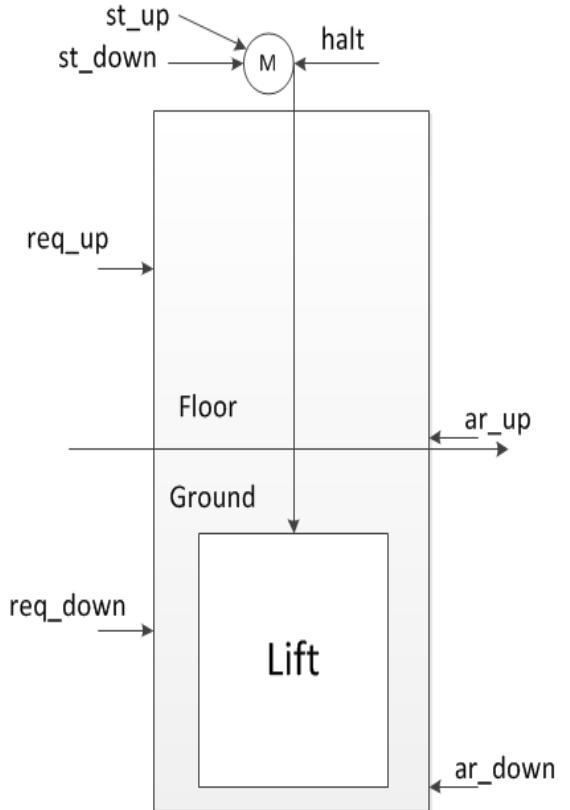
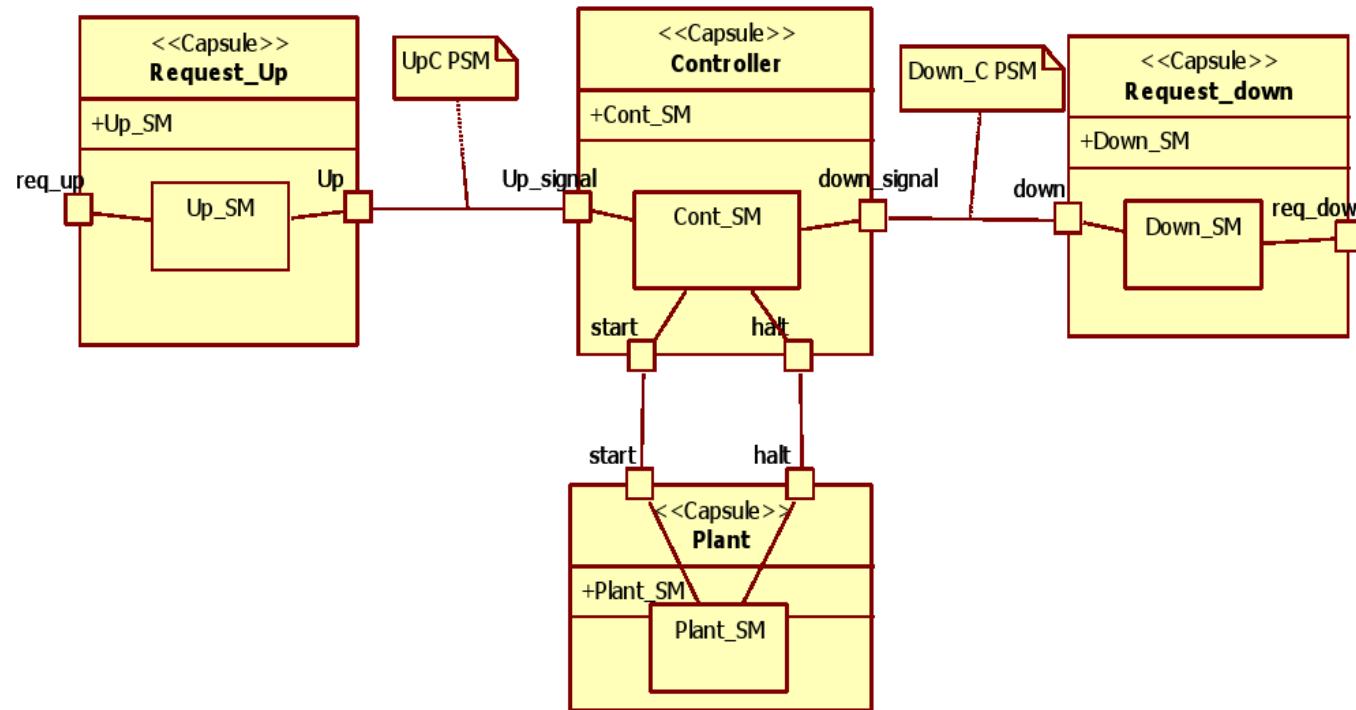
Supervisor receives through ps_i2 (input channel) a signal when the the robot reaches the ,r' responce and it is stoped there.

ts_3: sets the current robot position in ps_2 and sends through ps_o2 (output channel) to user the robot position to be displayed on the screen.

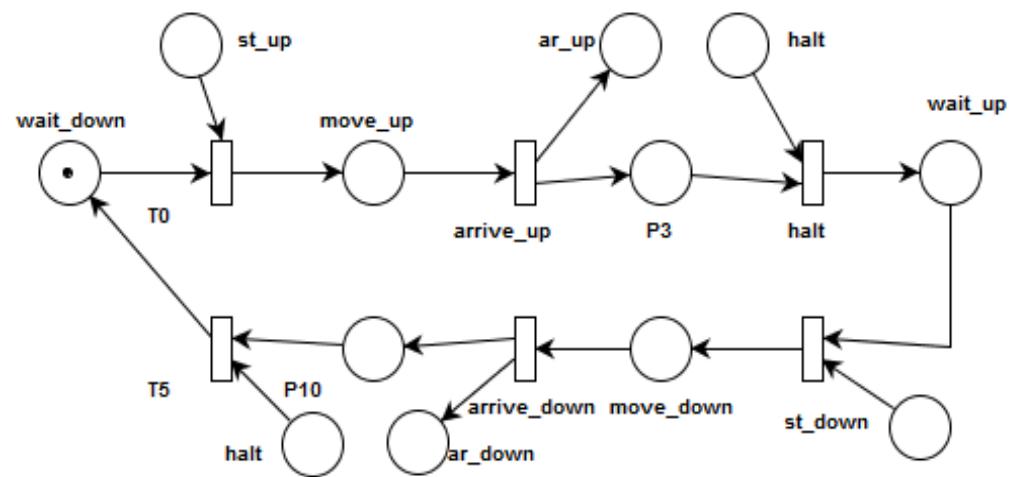
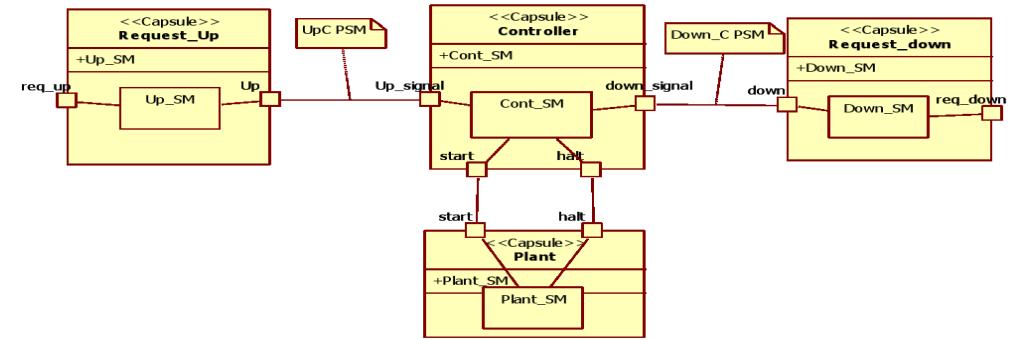
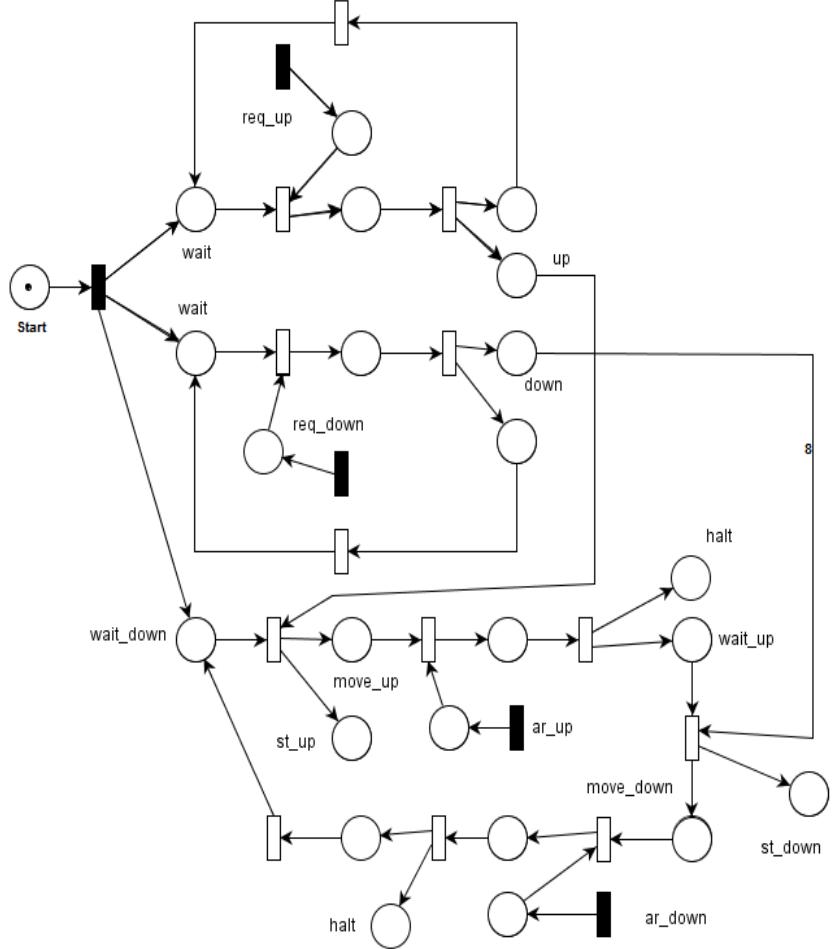
ts-2: waits the delay stored in ps_3 and proceeds with the next element of the List stored in ps_1. The procedure continues until no element remained in ps_1.List .

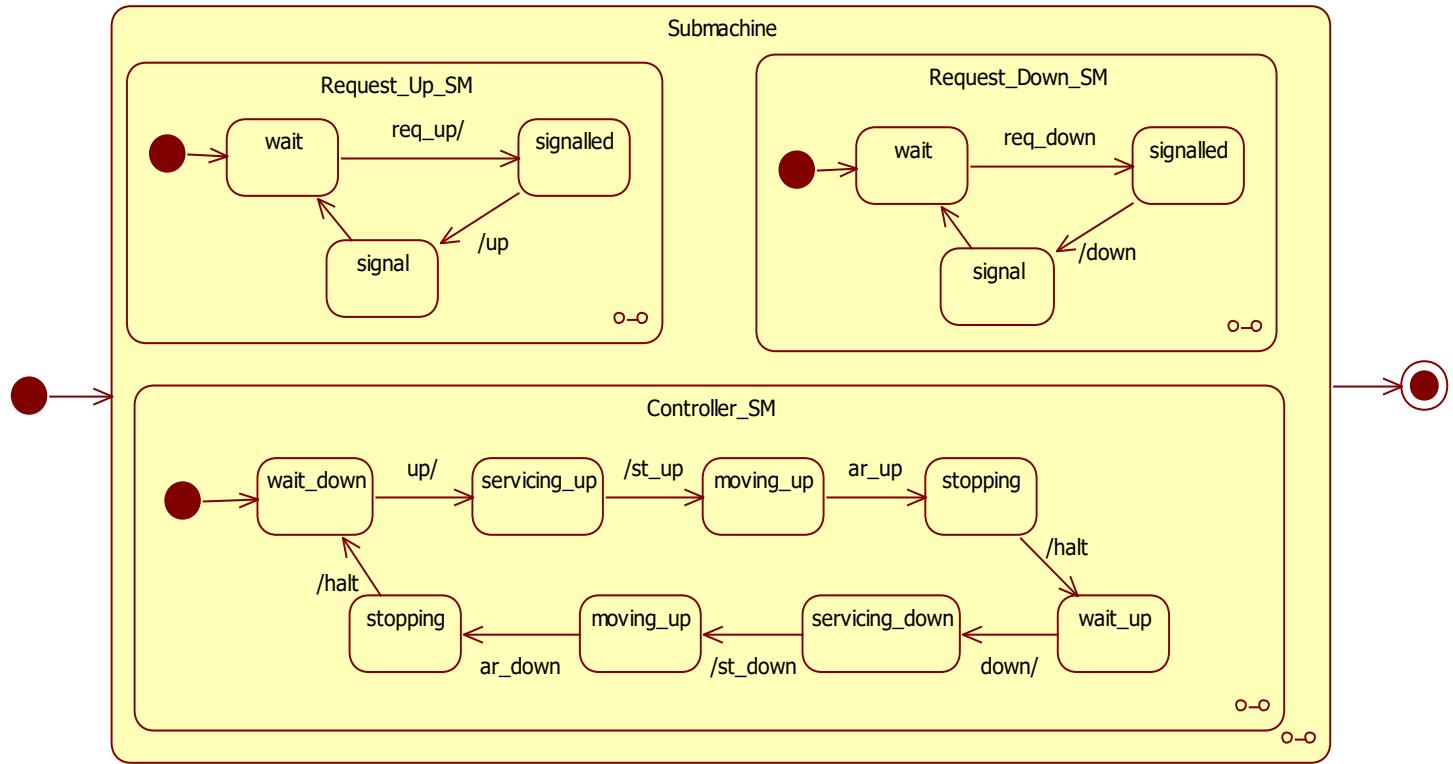


10. Implementation of UML RT Diagrams

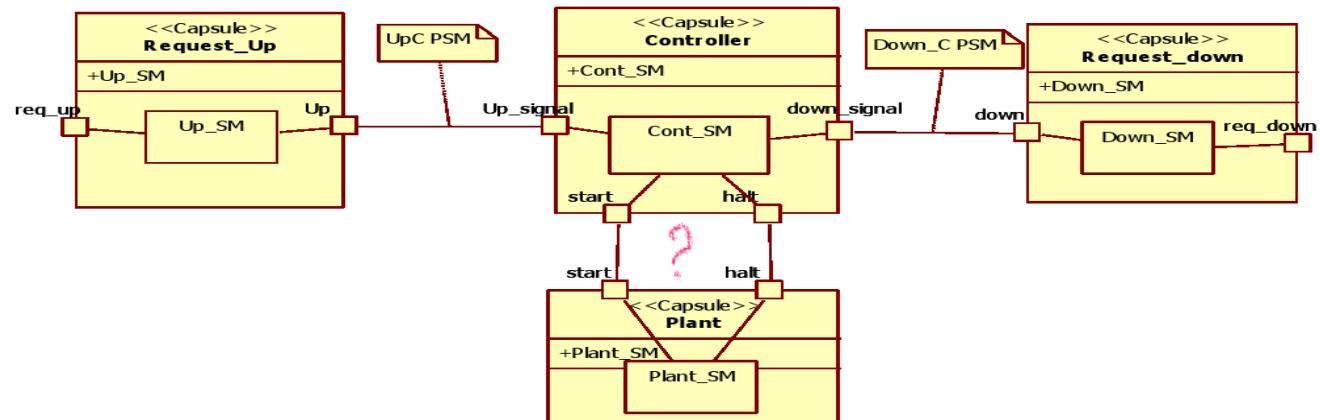


Simple elevator system – Capsule diagram





Add on capsule diagramme the events:
ar_up
ar_down



Requirements:

- the interface implemented by each port
- the state machine implemented by each capsule
- the control algorithms
- the class diagram
- the source program

Request_up specification:

Variant 1: pooling

Request_up reads periodically (0.1 sec.) the request up channel and sets a Boolean variable *UP true* if the button up is pressed or *false* if it is not pressed.

If the button is pressed *Request_up* demands *Controller* to send the cage up.

Request_up algorithm:

initialization: UP= false;

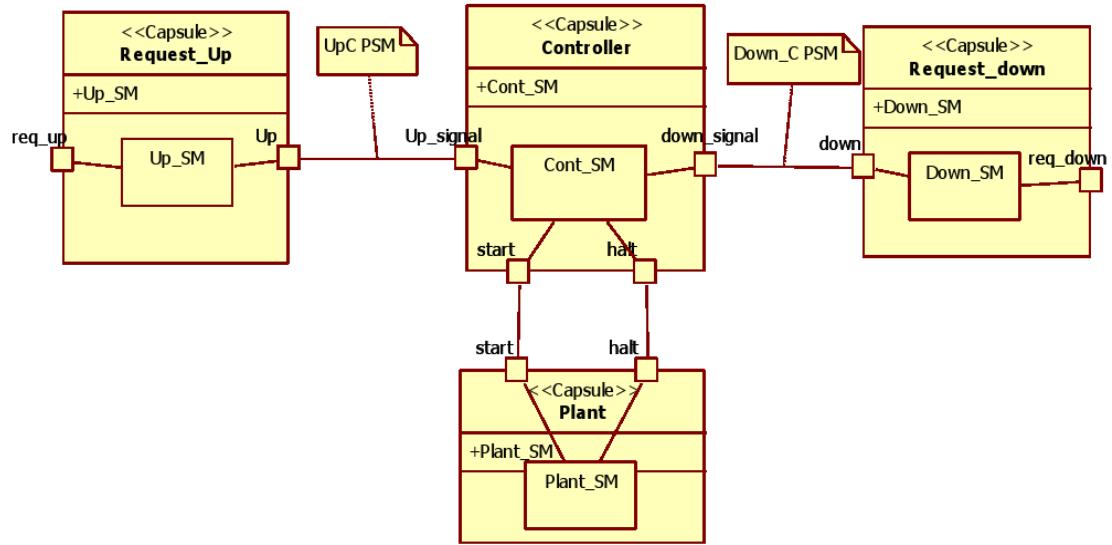
while(true)

wait(0.1 sec.);

 UP=read_up_ch();

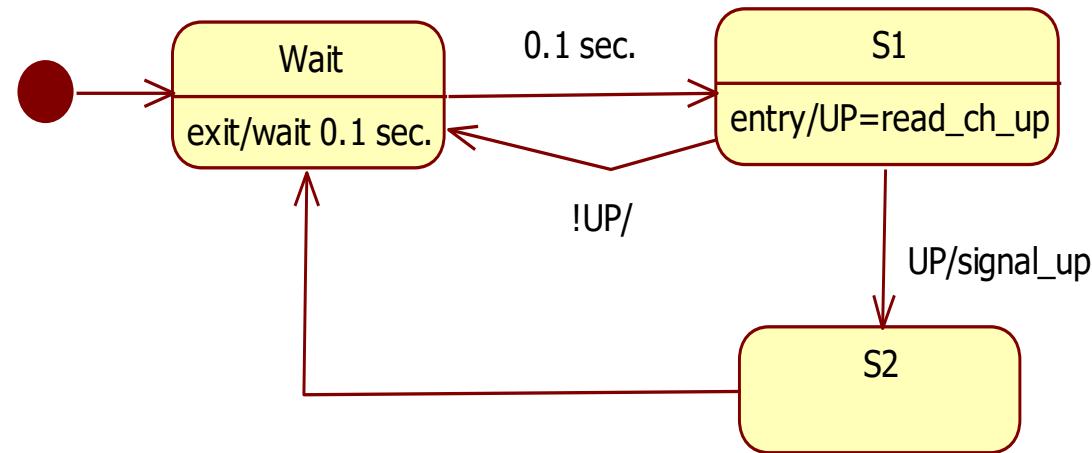
if UP signal_up();

endwhile;



```
interface Up {  
    signal_up();  
}
```

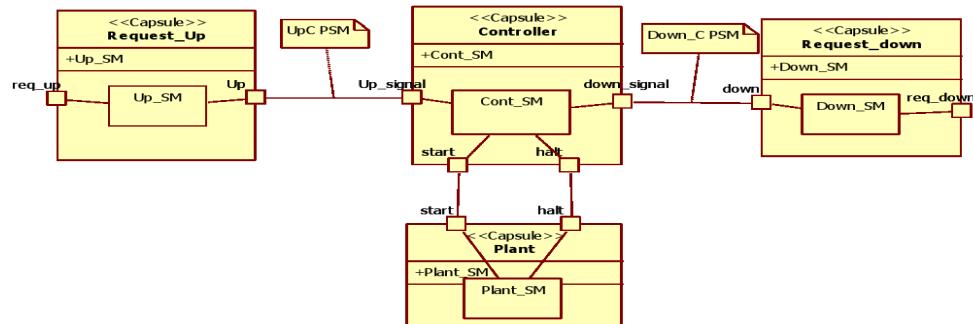
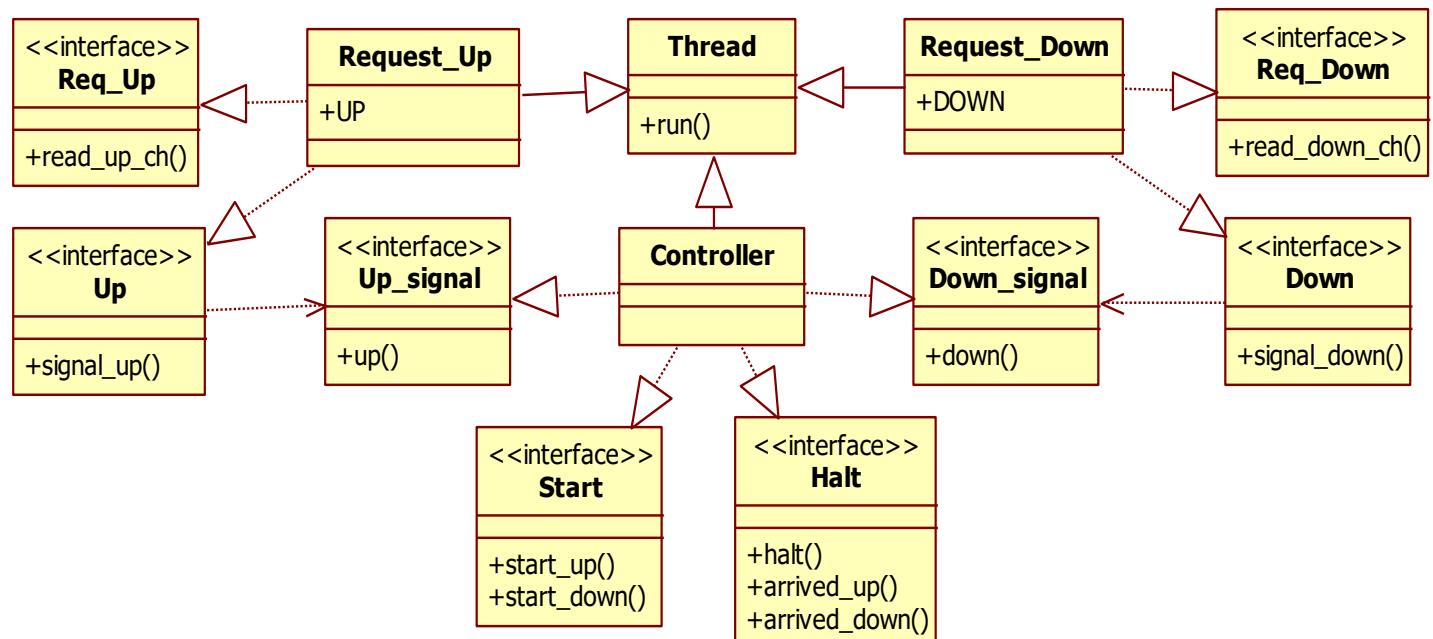
```
interface Req_up {  
    read_up_ch();  
}
```



```

class Request_up
extends Thread
implements Req_up,
Up {
    boolean UP;
    void run() {
        .....
    }
    void signal_up() {
        control.up();
        // object controller
    }
}

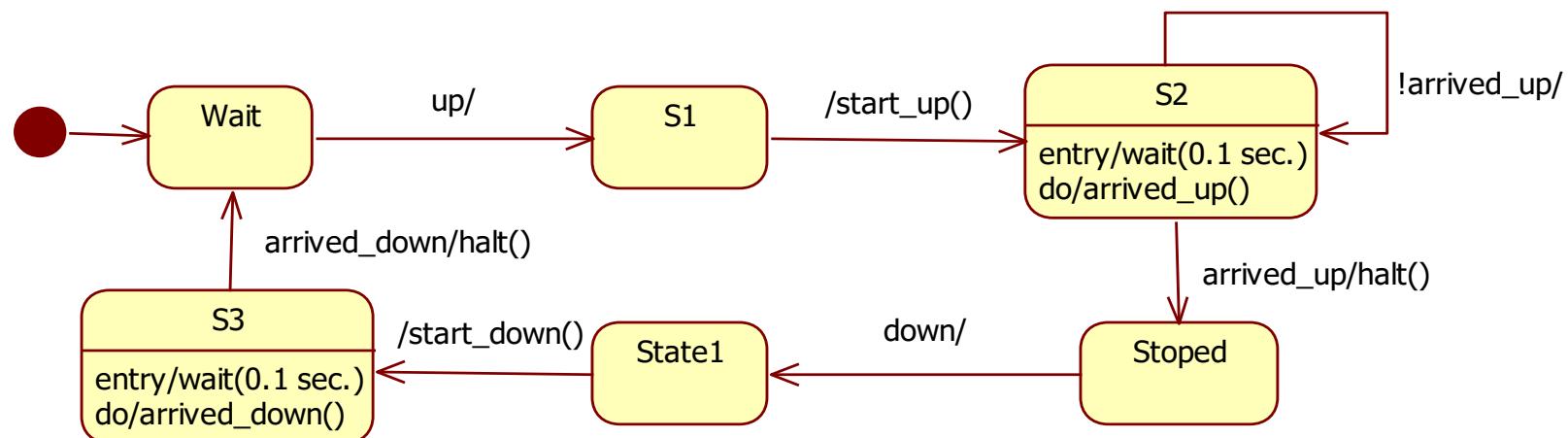
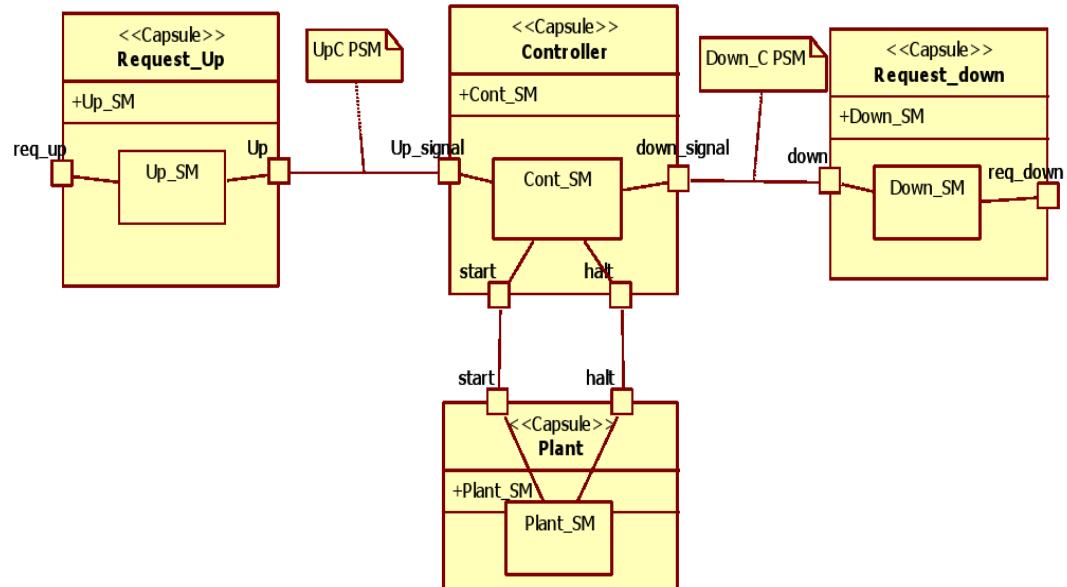
```



Controller specification

It waits to be signaled to send (up, down) the cage. *up()* method (from *Up_signal* interface) signals to send up the cage. Similar is *down()*.

Controller call the *start_up()* method to move the cage up. It reads ***periodically*** the *arrived_up()* method. When it reads *true*, the controller call *halt()* method to stop the motor.



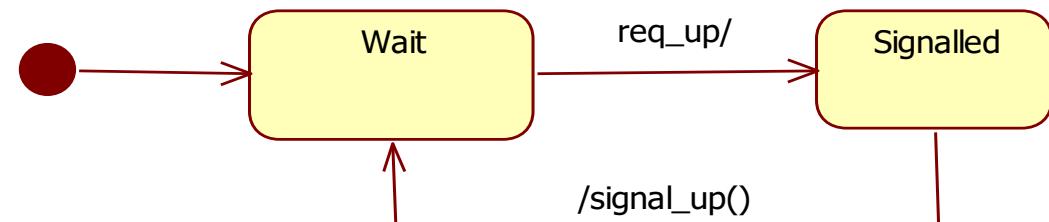
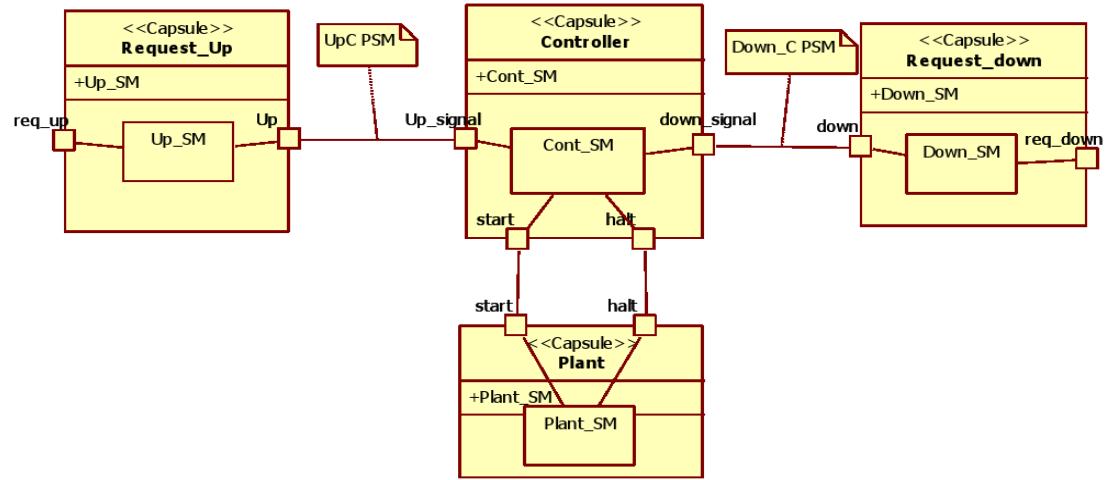
Variant 2: interruption

Request_up is signaled when the button up is pressed. Operating system signals the *req_up* event.

Request_up waits until *req_up* is signaled and demands *Controller* the cage up.

Request_up algorithm:

```
while(true)
    wait(req_up);
    signal_up();
endwhile;
```



*

END

*

4.5.11.Timing Programming in Java (Concurrent Programming)

4.5.11.1Timing problems

4.5.11.2 Classes and solutions for timing

4.5.11.3 Priorities of timed threads

4.5.12 Java 8 Concurrency

4.5.13 Concurrent Architectures

4.5.11.1 Timing Problems

N – Non-existent,

C – Created,

D – Dead = zombie,

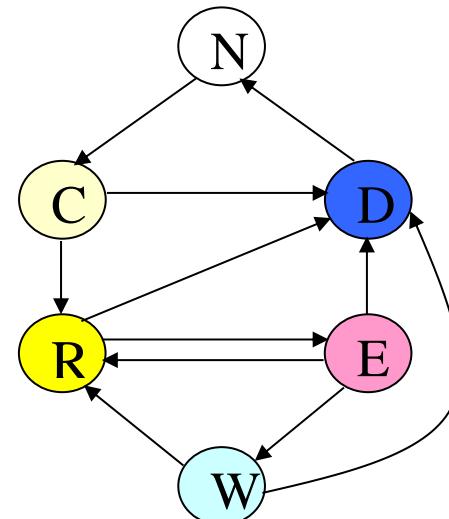
R – Ready to execution,

E – Execution =running,

W – Waiting = blocked.

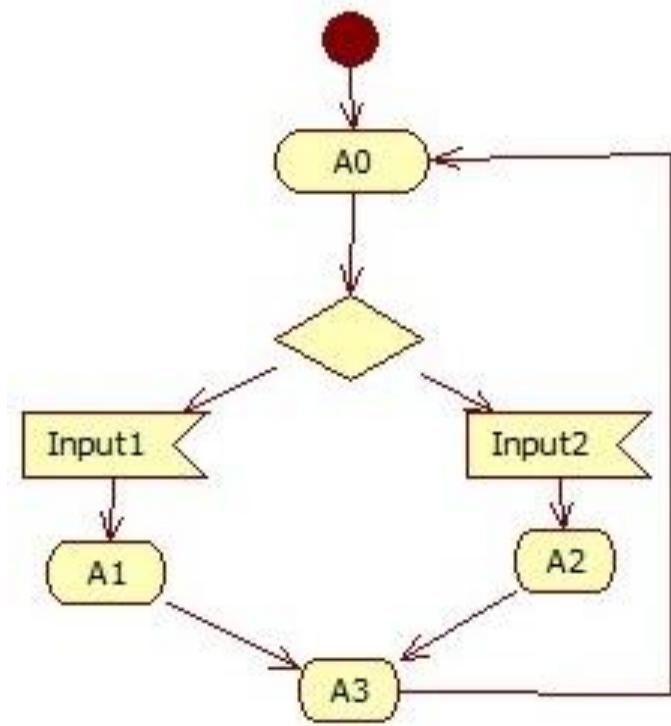
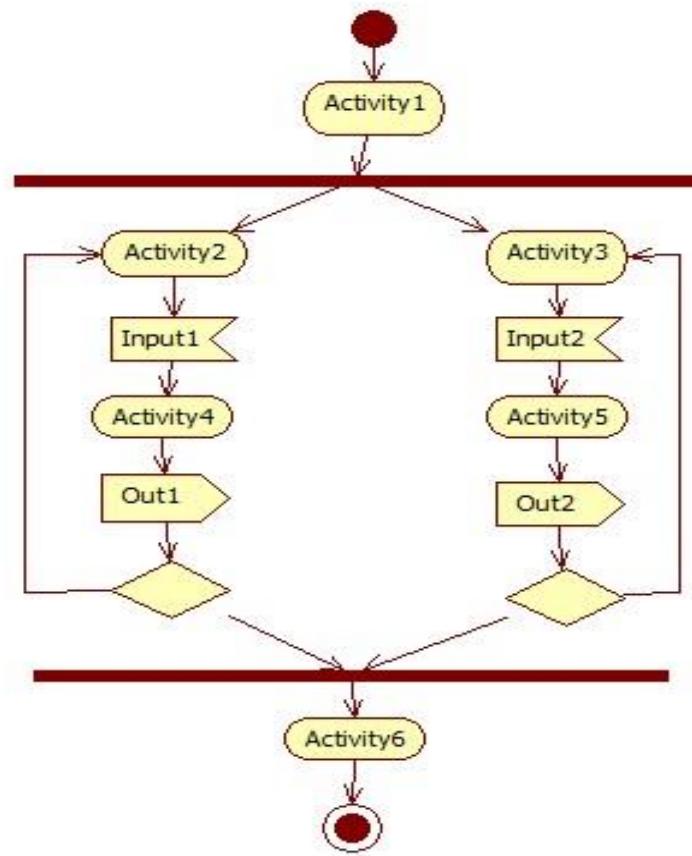
Goals:

- synchronization with physical time
 - sleep()
 - wait()
 - time server
- synchronization with timed messages (information)
 - delay queue



State diagrams of Java threads

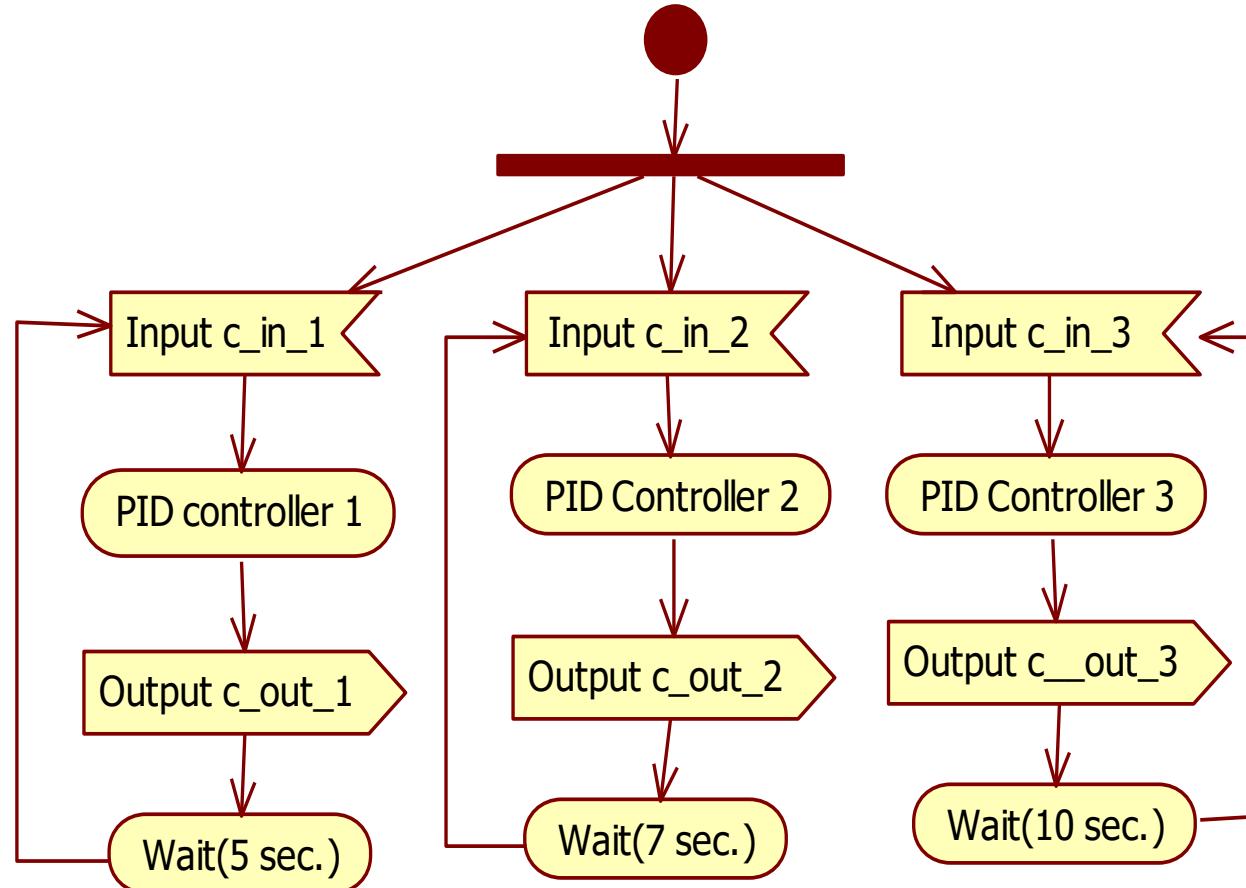
a) Non-blocking Read



Homework: a) conceive the equivalent OETPN model
b) conceive a non-blocking read implementation for the two input channels Input1 and Input2. (Input1 ← Keyboard; Input2 ← awtEvent)

b) Periodic read. *Problem requirement:*

Execute the threads with the periods 5, 7 and 10 t. u. respectively.



Implemented method:

```
public long getTime(){  
    Calendar calendar=new GregorianCalendar();  
    long time=calendar.getTimeInMillis();  
    return time;  
}
```

Testing the fulfilling of allocated duration

```
while(stp)  
{  
    ant=getTime()  
    //periodical execution  
    .....  
    post=getTime()  
    if((post-ant)>allocatedDuration) throw new MyException(.....)  
    //sau anunta un observator ca nu poate indeplini cerintele de TR  
}
```

Can it be disturbed?

Ce semnalizează firul?

Trebuie prinsă excepția de către un obiect de monitorizare (NU monitorul unui obiect).

4.5.11.2. Classes and solutions for timing

Package java.util: Classes Timer and TimerTask

java.util.concurrent: Class ScheduledThreadPoolExecutor

Timer – Each Timer object is a single background thread that is used to execute all of timer's tasks sequentially.

Constructor Summary

Timer ()

Creates a new timer.

Timer (boolean isDaemon)

Creates a new timer whose associated thread may be specified to run as a daemon.

Method Summary

void **cancel**()

Terminates this timer, discarding any currently scheduled tasks.

void **schedule**(TimerTask task, Date time)

Schedules the specified task for execution at the specified time.

void **schedule**(TimerTask task, Date firstTime,
long period)

Schedules the specified task for repeated *fixed-delay execution*,
beginning at the specified time.

void **schedule**(TimerTask task, long delay)

Schedules the specified task for execution after the specified
delay.

void **schedule**(TimerTask task, long delay,
long period)

Schedules the specified task for repeated *fixed-delay execution*,
beginning after the specified delay.

void **scheduleAtFixedRate**(TimerTask task,
Date firstTime, long period)

Schedules the specified task for repeated *fixed-rate execution*, beginning at the specified time.

```
void scheduleAtFixedRate (TimerTask task, long delay,  
long period)
```

Schedules the specified task for repeated *fixed-rate execution*, beginning after the specified delay.

This class ***does not offer real-time guarantees***. It schedules tasks using the `Object.wait(long)` method.

public void schedule(TimerTask task, long delay, long period)

TimerTask

public class TimerTask implements Runnable

Methods:

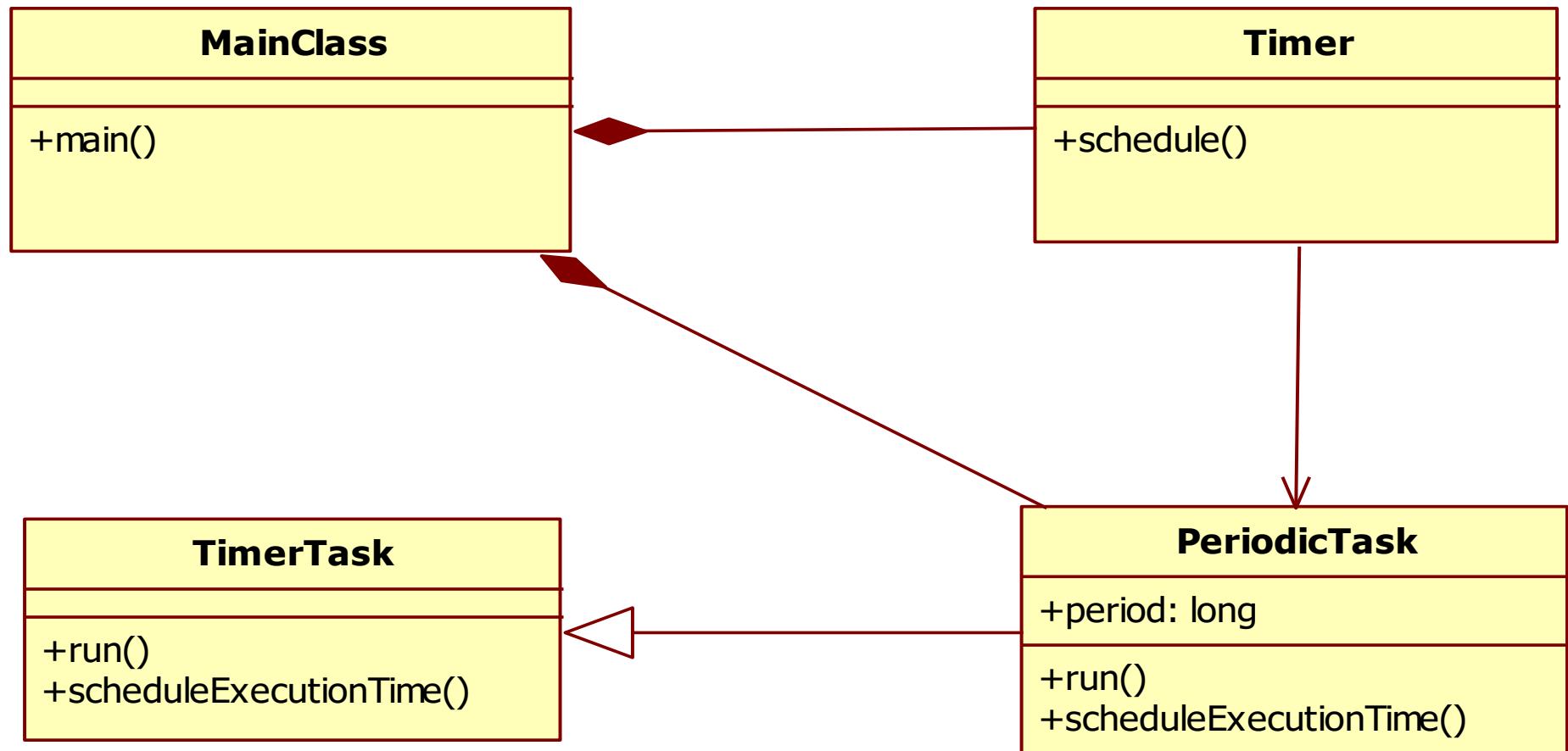
run()

long scheduleExecutionTime()

Method Summary

boolean <u>cancel()</u>	Cancels this timer task.
abstract void <u>run()</u>	The action to be performed by this timer task.
long <u>scheduledExecutionTime()</u>	Returns the <i>scheduled</i> execution time of the most recent <i>actual</i> execution of this task.

E.g.



```
import java.util.TimerTask;
import java.util.Date;

public class PeriodicTask extends TimerTask {
    int i=0;
    long period;
    String name;
    Date dt;
    public PeriodicTask(long p, String n){
        period =p;
        name=new String(n);
    }
    public void run() { //overwrite the run() method
        System.out.println("PeriodicTask - "+name+": i="+i);
        long set=this.scheduledExecutionTime();
        System.out.println(name+": scheduleExecutionTime="+set);
        i++;
        dt=new Date();
        System.out.println("Current time is: "+dt.getTime()+"; Thread name is "
    }
}
```

```

        +Thread.currentThread().getName()+" ; Priority is
        "+Thread.currentThread().getPriority());
    }
}

import java.util.Timer;
public class MainClass {
    public static void main(String[] args) {
        PeriodicTask pt1=new PeriodicTask(1000,"pt1");
        PeriodicTask pt2=new PeriodicTask(2000,"pt2");
        Timer tt=new Timer();
        tt.schedule(pt1,1000,pt1.period);
        tt.schedule(pt2,1000,pt2.period);
    }
}

```

**Homework: build the TPN
model and analyze the
application behavior.**

Displayed values:

```
PeriodicTask - pt1: i=0
pt1: scheduleExecutionTime=1196703908305
Current time is: 1196703908306; Thread name is Timer-0 ;
Priority is 5
PeriodicTask - pt2: i=0
pt2: scheduleExecutionTime=1196703908306
Current time is: 1196703908306; Thread name is Timer-0 ;
Priority is 5
PeriodicTask - pt1: i=1
pt1: scheduleExecutionTime=1196703909305
Current time is: 1196703909305; Thread name is Timer-0 ;
Priority is 5
PeriodicTask - pt1: i=2
pt1: scheduleExecutionTime=1196703910305
Current time is: 1196703910305; Thread name is Timer-0 ;
Priority is 5
PeriodicTask - pt2: i=1
pt2: scheduleExecutionTime=1196703910306
Current time is: 1196703910307; Thread name is Timer-0 ;
Priority is 5
PeriodicTask - pt1: i=3
```

```
pt1: scheduleExecutionTime=1196703911305
Current time is: 1196703911305; Thread name is Timer-0 ;
Priority is 5
PeriodicTask - pt1: i=4
```

Package `java.util.concurrent`

`ConcurrentLinkedQueue<E>`

`CyclicBarrier`

`DelayQueue`

`Executors`

`LinkedBlockingQueue`

`PriorityBlockingQueue`

`ScheduledThreadPoolExecutor`

`Semaphore`

`SynchronousQueue`

`ThreadPoolExecutor`

DelayQueue usage

Class DelayQueue

Is used to create Delay organized queue used to producer consumer problem.

Constructors:

`DelayQueue()` –Creates a DelayQueue that is initially empty.

`DelayQueue(Collection c)` –Creates a DelayQueue initially contains the element of the given collection.

Methods:

`boolean add(E o)` – E ← Element

`void clear()`

`Iterator iterator()`

boolean offer(E o) – Insert the specific element into the queue (right position)

`void put(E o)` – Adds the E to queue

`boolean offer(E o, long timeout, TimeUnit unit)`

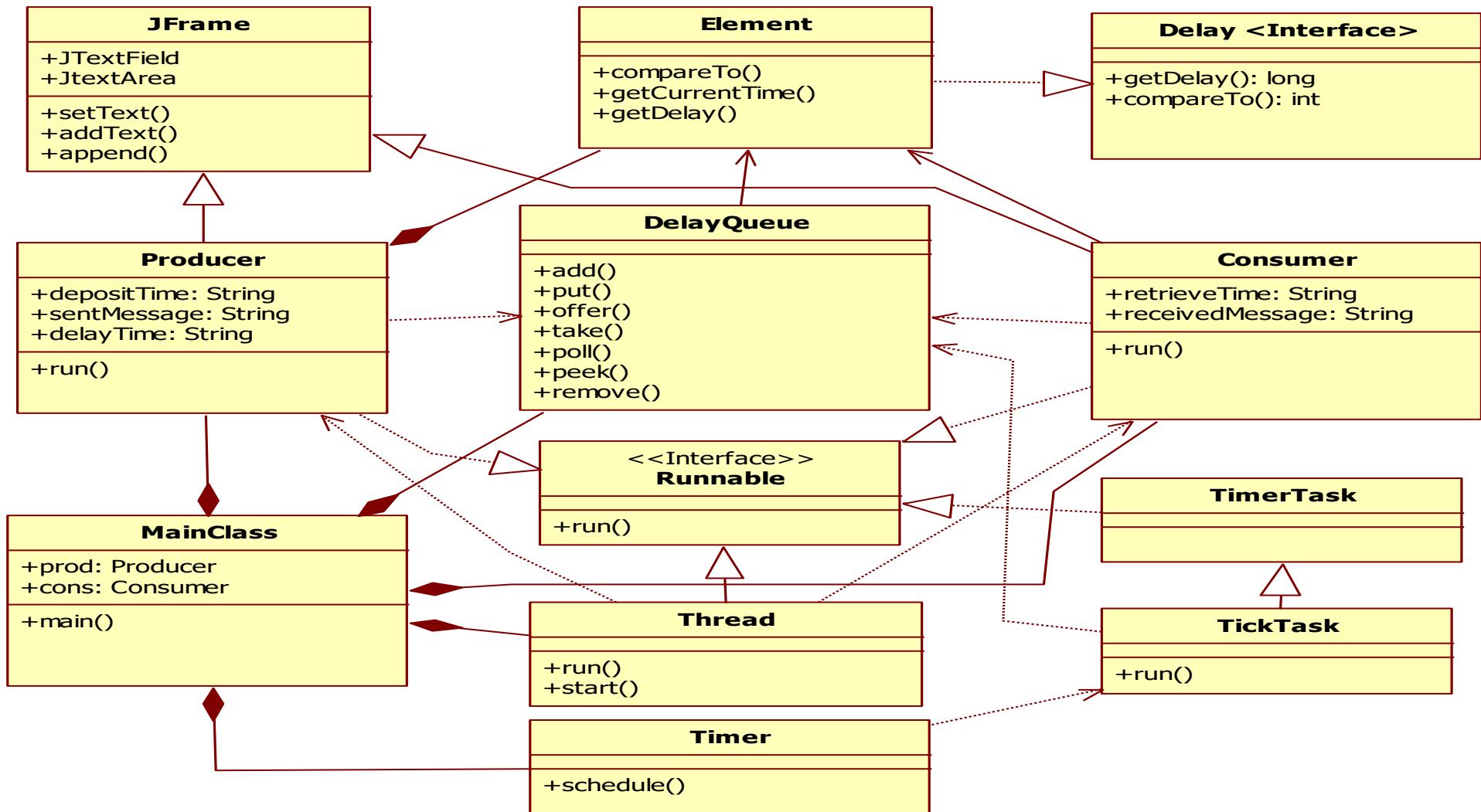
E take() – retrieve and removes

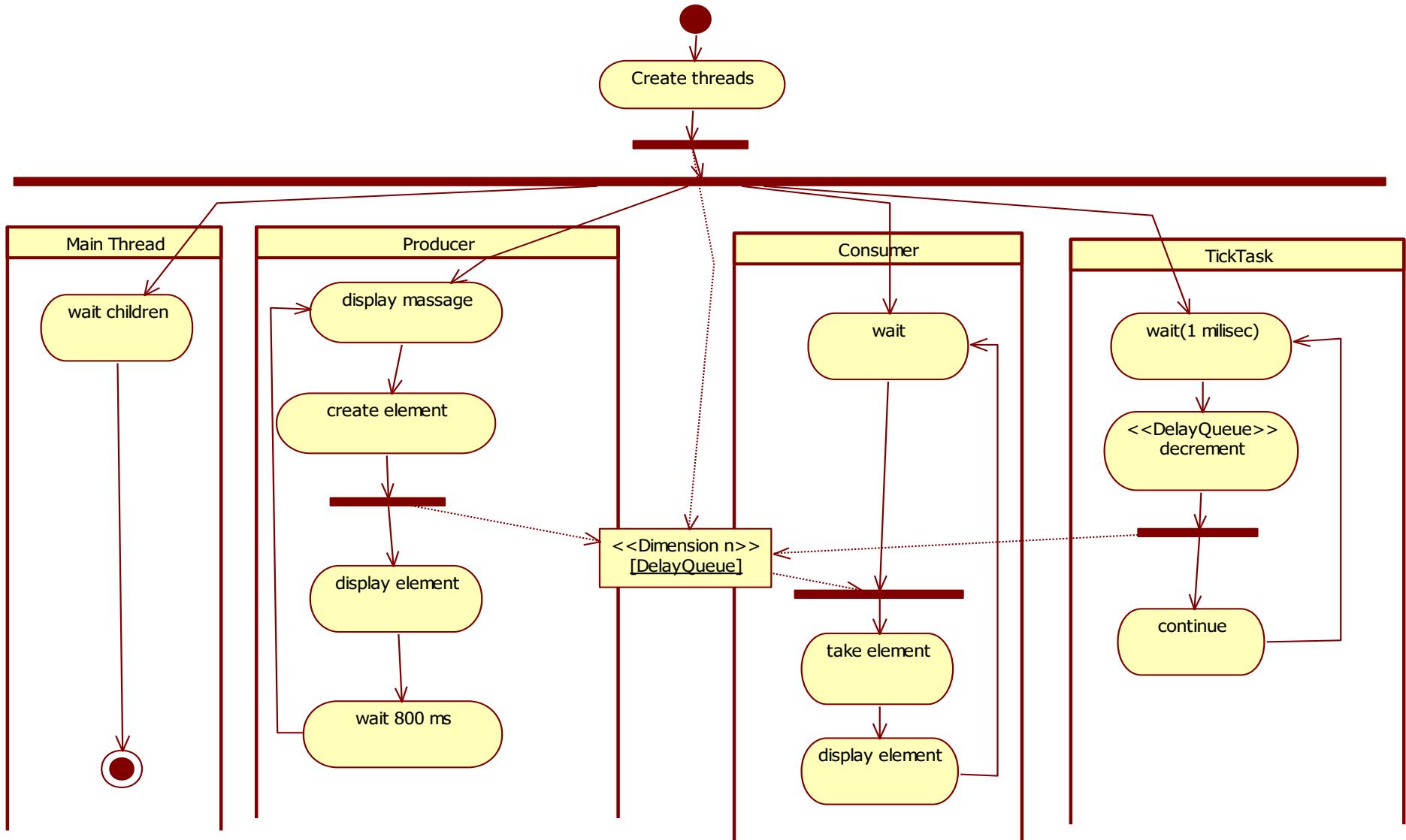
E poll(long timeout, TimeUnit unit) – retrieve and removes the head of this queue, waiting if necessary up to the specified waiting time if no element with unexpired delay are present on this queue.

E poll()

E peek() – Retrieves but not removes the head of this queue

boolean remove(Object o)





DelayQueue Application – Activity diagram

Class Element used for the DelayQueue

```
import java.util.*;
import java.util.concurrent.*;
public class Element implements Delayed{
    String unu;
    String doi;
    long del;//milliseconds
    Element(String s1, String s2, long d) {
        unu=new String(s1);
        doi=new String(s2);
        del=d;
    }
    public long getDelay(TimeUnit tum) {
        return del;
    }
    public int compareTo(Object ob) {
        int x=0;
        Element tmp=(Element) ob;
        if(this.del<tmp.del) x=-1;
```

```

        if(this.del>tmp.del) x= 1;
        if(this.del==tmp.del)x= 0;
        return x;
    }
    public String getCurrentTime(){
        Calendar calendar=new GregorianCalendar();
        int hour=calendar.get(Calendar.HOUR);
        int minute=calendar.get(Calendar.MINUTE);
        int second=calendar.get(Calendar.SECOND);
        long milisec=calendar.getTimeInMillis();
        return ("H="+hour+" Min="+minute+ "Sec="+second+"; Milisec="+milisec );
    }
    public String toString(){
        return (unu+" si "+doi+ "; del="+this.getDelay(TimeUnit.MILLISECONDS));
    }
}

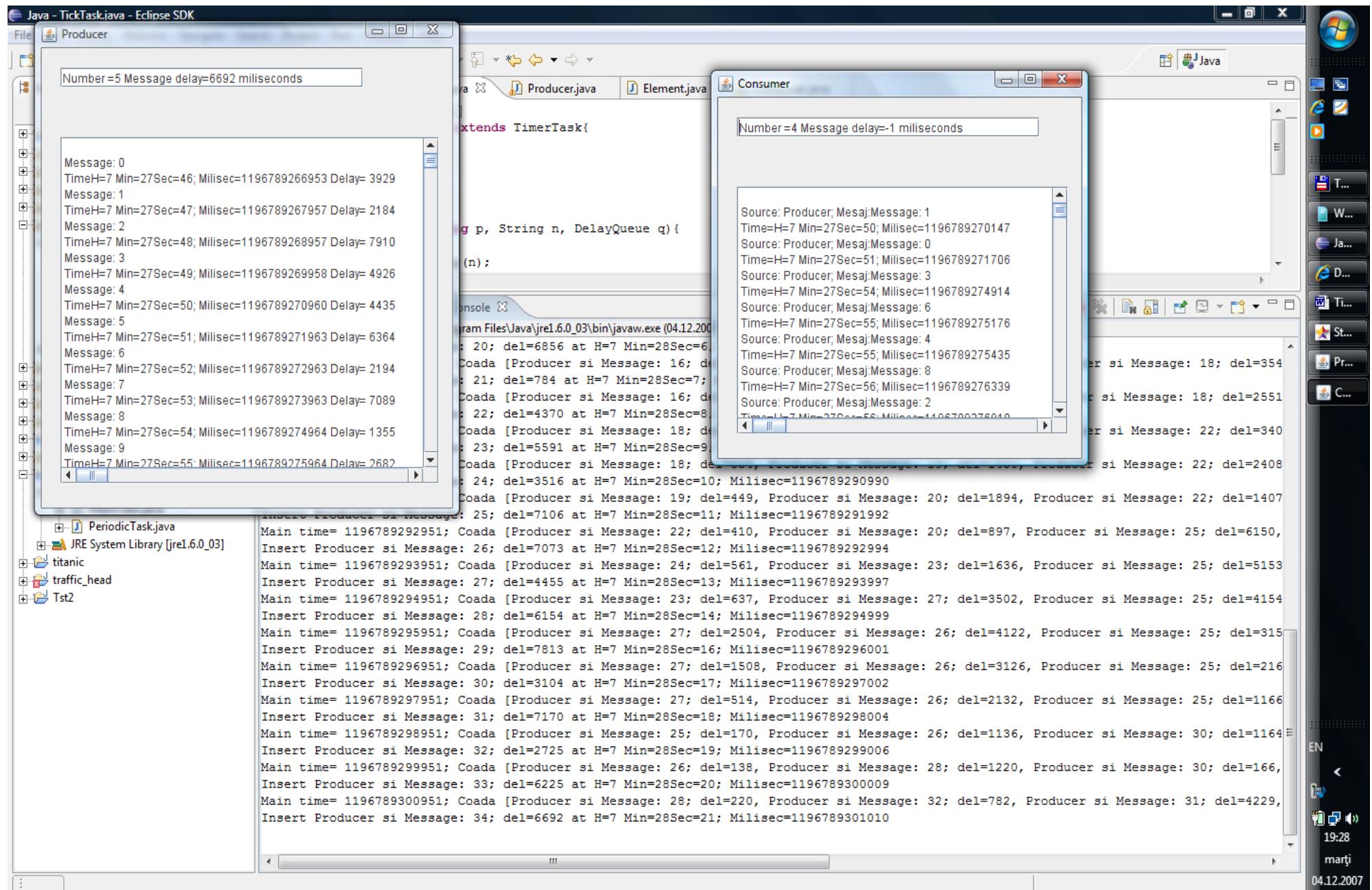
```

Producer's run() method

```
public void run() {  
    int i=0;  
    while(act) {  
        ms=message+i;  
        ta.append("\n "+ ms);  
        long lng= (long) (8000*Math.random());  
        int nr=delQ.size();  
        tf1.setText("Number =" +nr+ " Message delay=" +lng+ " miliseconds" );  
        el=new Element(name,ms,lng);  
        delQ.offer(el,lng,TimeUnit.MILLISECONDS);  
        ta.append("\n Time"+el.getCurrentTime()+" Delay= "+el.del);  
        try{Thread.sleep(800);}catch(InterruptedException ex){ }  
        i++;  
    }  
}
```

Consumer's run() method

```
public void run() {
    while(act) {
        try {el = (Element)delQ.poll(1000, TimeUnit.MILLISECONDS);}
        catch(InterruptedException ei) {//merge si take()
            System.out.println("Nu este mesaj in coada la"+el.getCurrentTime());
        }
        if (el!=null) {
            source=el.unu;
            ms=el.doi;
            ta.append("\n Source: "+ source+"; Mesaj:"+ ms);
            ta.append("\n Time="+el.getCurrentTime());
            int nr=delQ.size();
            tf1.setText("Number =" +nr+ " Message delay=" +el.del+ " miliseconds"
);
            ta.append("\n Time"+el.getCurrentTime());
        }
    }
}
```



```
Main time= 1196789266951; Coada [Producer si Message: 0;  
del=3929]  
Insert Producer si Message: 0; del=3929 at H=7 Min=27Sec=46;  
Milisec=1196789266953  
Main time= 1196789267951; Coada [Producer si Message: 0;  
del=3741]  
Insert Producer si Message: 1; del=2185 at H=7 Min=27Sec=47;  
Milisec=1196789267957  
Main time= 1196789268951; Coada [Producer si Message: 1;  
del=1195, Producer si Message: 0; ➔del=2746]  
Insert Producer si Message: 2; del=7910 at H=7 Min=27Sec=48;  
Milisec=1196789268957  
Main time= 1196789269951; Coada [Producer si Message: 1;  
del=195, Producer si Message: 0; ➔del=1746, Producer si  
Message: 2; del=6916]  
Insert Producer si Message: 3; del=4926 at H=7 Min=27Sec=49;  
Milisec=1196789269958  
Main time= 1196789270951; Coada [Producer si Message: 0;  
del=751, Producer si Message: 3; ➔del=3938, Producer si  
Message: 2; del=5921]  
Insert Producer si Message: 4; del=4435 at H=7 Min=27Sec=50;  
Milisec=1196789270960
```

java.util.concurrent: Class ScheduledThreadPoolExecutor

java.lang.Object

 └ java.util.concurrent.AbstractExecutorService

 └ java.util.concurrent.ThreadPoolExecutor

 └

java.util.concurrent.ScheduledThreadPoolExecutor

A [ThreadPoolExecutor](#) that can additionally schedule commands to run after a given delay, or to execute periodically. This class is preferable to [Timer](#) when multiple worker threads are needed, or when the additional flexibility or capabilities of [ThreadPoolExecutor](#) (which this class extends) are required.

Delayed tasks execute no sooner than they are enabled, but without any real-time guarantees about when, after they are enabled, they will commence. Tasks scheduled for exactly the same execution time are enabled in first-in-first-out (FIFO) order of submission.

While this class inherits from [ThreadPoolExecutor](#), a few of the inherited tuning methods are not useful for it. In particular, because it acts as a fixed-sized pool using corePoolSize threads and an unbounded queue, adjustments to maximumPoolSize have no useful effect.

Extension notes: This class overrides [AbstractExecutorService](#) submit methods to generate internal objects to control per-task delays and scheduling. To preserve functionality, any further overrides of these methods in subclasses must invoke superclass versions, which effectively disables additional task customization. However, this class provides alternative protected extension method `decorateTask` (one version each for `Runnable` and `Callable`) that can be used to customize the concrete task types used to execute commands entered via `execute`, `submit`, `schedule`, `scheduleAtFixedRate`, and `scheduleWithFixedDelay`. By default, a `ScheduledThreadPoolExecutor` uses a task type extending [FutureTask](#).

It may be modified or replaced using subclasses of the form:

```
public class CustomScheduledExecutor extends  
ScheduledThreadPoolExecutor {  
    static class CustomTask<V> implements  
RunnableScheduledFuture<V> { ... }  
    protected <V> RunnableScheduledFuture<V> decorateTask(  
        Runnable r, RunnableScheduledFuture<V> task) {  
        return new CustomTask<V>(r, task);  
    }
```

```
protected <V> RunnableScheduledFuture<V> decorateTask(  
    Callable<V> c, RunnableScheduledFuture<V> task) {  
    return new CustomTask<V>(c, task);  
}  
// ... add constructors, etc.  
}
```

Methods:

public Future<?> submit(Runnable task)

Description copied from interface: [ExecutorService](#)

Submits a Runnable task for execution and returns a Future representing that task.

The Future's get method will return null upon *successful* completion.

Specified by:

[submit](#) in interface [ExecutorService](#)

Overrides:

[submit](#) in class [AbstractExecutorService](#)

Parameters:

task - the task to submit

Returns:

a Future representing pending completion of the task

```
public <T> Future<T> submit(Callable<T> task)
```

Description copied from interface: [ExecutorService](#)

Submits a value-returning task for execution and returns a Future representing the pending results of the task. The Future's get method will return the task's result upon successful completion.

If you would like to immediately block waiting for a task, you can use constructions of the form `result = exec.submit(aCallable).get();`

Note: The [Executors](#) class includes a set of methods that can convert some other common closure-like objects, for example, [PrivilegedAction](#) to [Callable](#) form so they can be submitted.

Specified by:

[submit](#) in interface [ExecutorService](#)

Overrides:

[submit](#) in class [AbstractExecutorService](#)

Parameters:

task - the task to submit

Returns:

a Future representing pending completion of the task

```
public void execute(Runnable command)
```

Executes command with zero required delay. This has effect equivalent to schedule(command, 0, anyUnit).

```
public ScheduledFuture schedule(Runnable command, long delay,  
TimeUnit unit)
```

Creates and executes a one-shot action that becomes enabled after the given delay.

```
public ScheduledFuture schedule(Callable callable, long delay, TimeUnit unit)
```

Description copied from interface: [ScheduledExecutorService](#)

Creates and executes a ScheduledFuture that becomes enabled after the given delay.

Specified by: [schedule](#) in interface [ScheduledExecutorService](#)

Parameters:

callable - the function to execute

delay - the time from now to delay execution

unit - the time unit of the delay parameter

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,  
long initialDelay, long period, TimeUnit unit)
```

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 * period, and so on. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor. If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.

Specified by: [scheduleAtFixedRate](#) in interface [ScheduledExecutorService](#)

Parameters:

command - the task to execute

initialDelay - the time to delay first execution

period - the period between successive executions

unit - the time unit of the initialDelay and period parameters

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,  
long initialDelay, long delay, TimeUnit unit)
```

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one

execution and the commencement of the next. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor.

Specified by: [scheduleWithFixedDelay](#) in interface [ScheduledExecutorService](#)

Parameters:

command - the task to execute

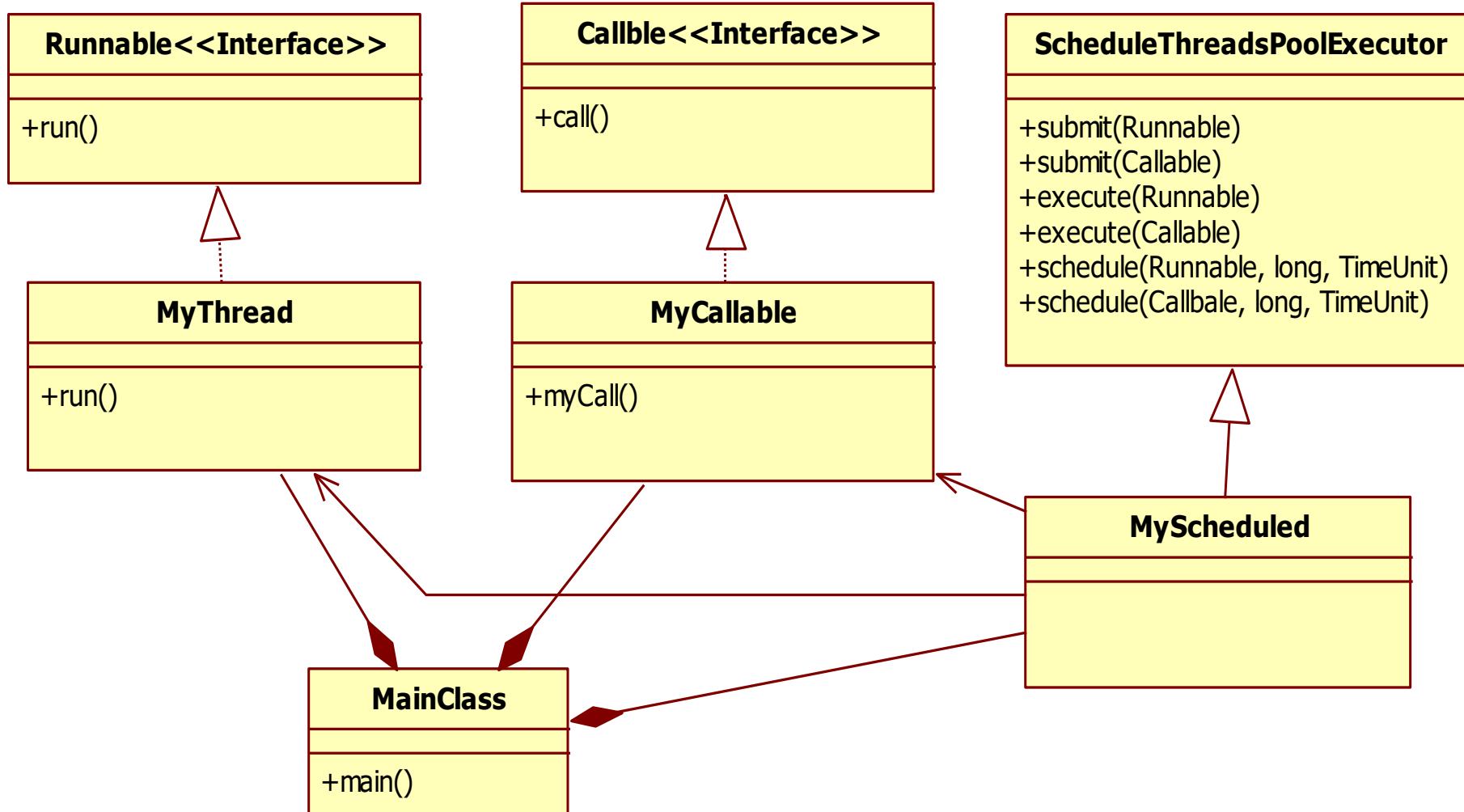
initialDelay - the time to delay first execution

delay - the delay between the termination of one execution and the commencement of the next

unit - the time unit of the initialDelay and delay parameters

Returns:

a ScheduledFuture representing pending completion of the task, and whose get() method will throw an exception upon cancellation



4.5.11.3 Priorities of timed threads

Child Thread priority = parent thread priority if no setPriority() is used.

A thread activated by a time server is executed at its own priority.

If a busy thread occupies entire processor time (in a mono-processor system) and its priority is higher than the time server priority, another thread cannot be activated by time server.

If a thread is activated (and it is registered to event) by an event, it is executed at the event listener (often awtEvent) priority.

Interrupt threads, System threads (clock), JVM threads (garbage collector, variable updater etc.) and user (regular and demons) threads are executed concurrently.

What are the daemon priorities?

4.5.12 Java 8 Concurrency

4.11.1. Executors Lambda Expression

They express instances of functional interfaces

An interface with a single object method is called functional interface

(See Software engineering course → anonymous classes)

Functionality → a method argument or code as data

Runnable interface → *run()* is such an example

Example:

```
interface FunctionalInterface
{
    void abstractFunction(String x1, String x2);
}

public class Test
{
    public static void main(String args[])
    {
        String s1="lambda ";
        String s2="expression ";
        int i=3;
        // lambda expression to implement above functional interface.
        // Implements abstractFunction(String x1, String x2)
        FunctionalInterface fobj = (String x1, String x2) -> {
            String x=x1+x2 +i;
            System.out.println(x);
        };

        // Call above lambda expression and print ← method output .
        fobj.abstractFunction(s1,s2);
    }
}
```

}

Create and launch a thread

```
Runnable task = () -> {  
  
    String threadName = Thread.currentThread().getName();  
    System.out.println("New thread: " + threadName);  
};  
  
task.run();
```

Compare with:

```
Thread thread = new Thread(task);  
thread.start();
```

```

public class MainClass {

    public static void main(String[] args) {

        Runnable task = () -> {
            String threadName = Thread.currentThread().getName();
            System.out.println("New thread: " + threadName);
        };

        task.run();
        // Compare to

        Thread thread = new Thread(task);
        thread.start();
    }
}

```

Output

New thread: main
 New thread: Thread-0

Thread sleep example

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

```

import java.util.concurrent.*;
public class MainClass {
    public static void main(String[] args) {
        Runnable task2 = () -> {
            try {
                String name = Thread.currentThread().getName();
                System.out.println("Old name: " + name);
                Thread.currentThread().setName("MyThread");
                name = Thread.currentThread().getName();
                System.out.println("WAIT " + name);
                TimeUnit.SECONDS.sleep(2);
                System.out.println("AWAKEN " + name);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        };
        Thread thread = new Thread(task2);
        thread.start();
    }
}

```

Output:
 Old name: Thread-0
 WAIT MyThread
 AWAKEN MyThread

4.5.12.2 Callable Example

```
package callable;
import java.util.concurrent.*;
import java.util.Arrays;
import java.util.List;
public class MainCl {

    public static void main(String[] args) {
        System.out.println("start");

        ExecutorService executor = Executors.newWorkStealingPool();

        List<Callable<String>> callables = Arrays.asList(
            () -> "task1",
            () -> "task2",
            () -> "task3");
        try {
            executor.invokeAll(callables)
                .stream()
```

```
.map(future -> {
    try {
        return future.get();
    }
    catch (Exception e) {
        throw new IllegalStateException(e);
    }
})
.forEach(System.out::println);
}catch (InterruptedException ex) {
    System.out.println("Interrupted thread");
}
}
```

4.5.13 Concurrent Architecture (Arhitecturi conceptuale concurente)

Basic Architecture: client-server

Server:

- secvențial - sequential
- multifir - multithreading

Arhitectures:

- Filter Pipeline = Conducte pentru filtrare
- Supervisor-Worker = Supervizor-muncitor
- Announcer-Listener = Crainic-ascultător
- Planned tasks execution

1) Filter Pipeline = Conducte pentru filtrare

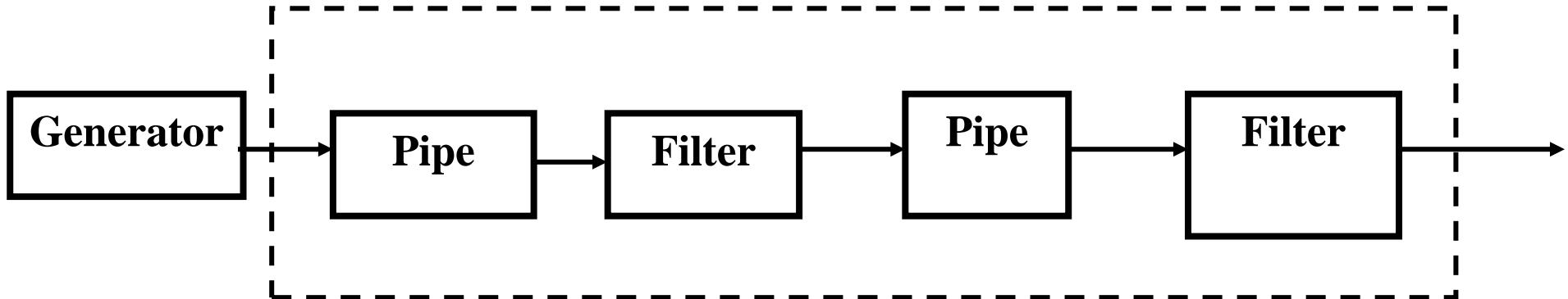
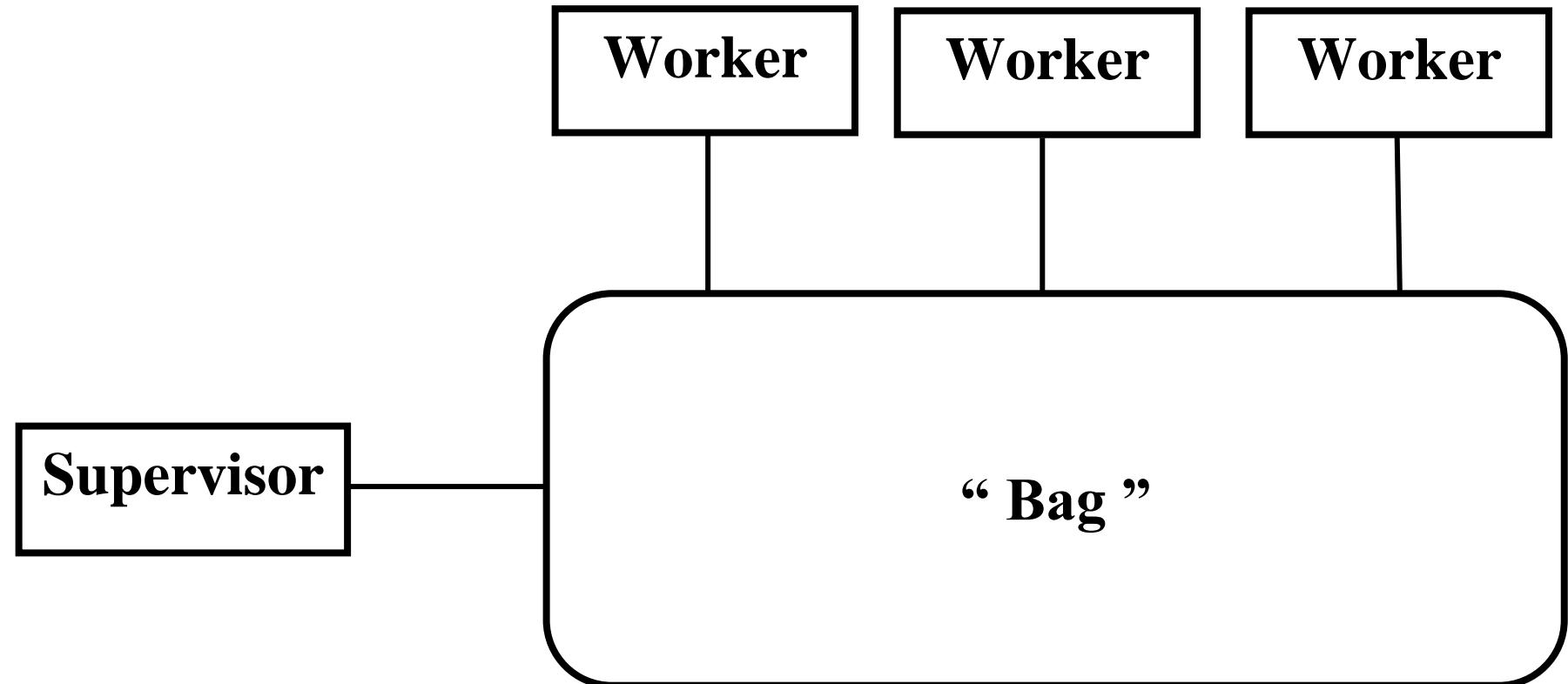


Image processing

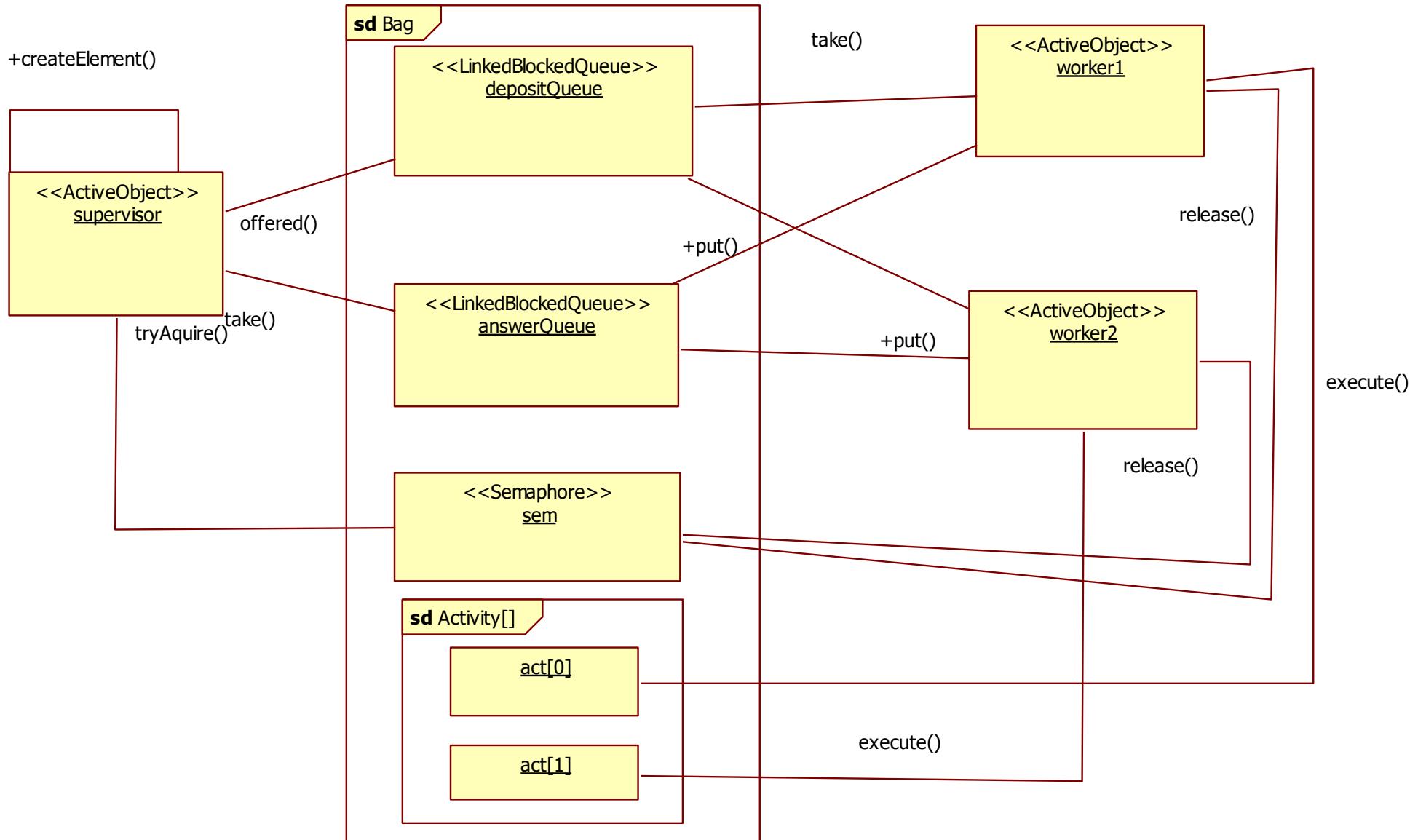
2) Supervisor-Worker = (Supervizor-muncitori)



Supervizorul este responsabil de generarea unui set de sarcini și plasarea lor în sac. Supervizorul colectează rezultatele din sac și determină când s-a terminat prelucrarea. Fiecare muncitor ia repetat sarcini din sac, le prelucrează și plasează rezultatele în sac. Activitățile muncitorului continuă până când supervisorul cere oprirea lor. Arhitectura poate fi extinsă pentru paraleлизarea (efectivă) a prelucrării pe principiul „divide et impera” (divide-and conquer = divide și cucerește) în care muncitorii pun și ei noi sarcini în sac ca rezultat al prelucrărilor lor. Deci rezultatul prelucrării poate fi interpretat ca un nou set de sarcini.

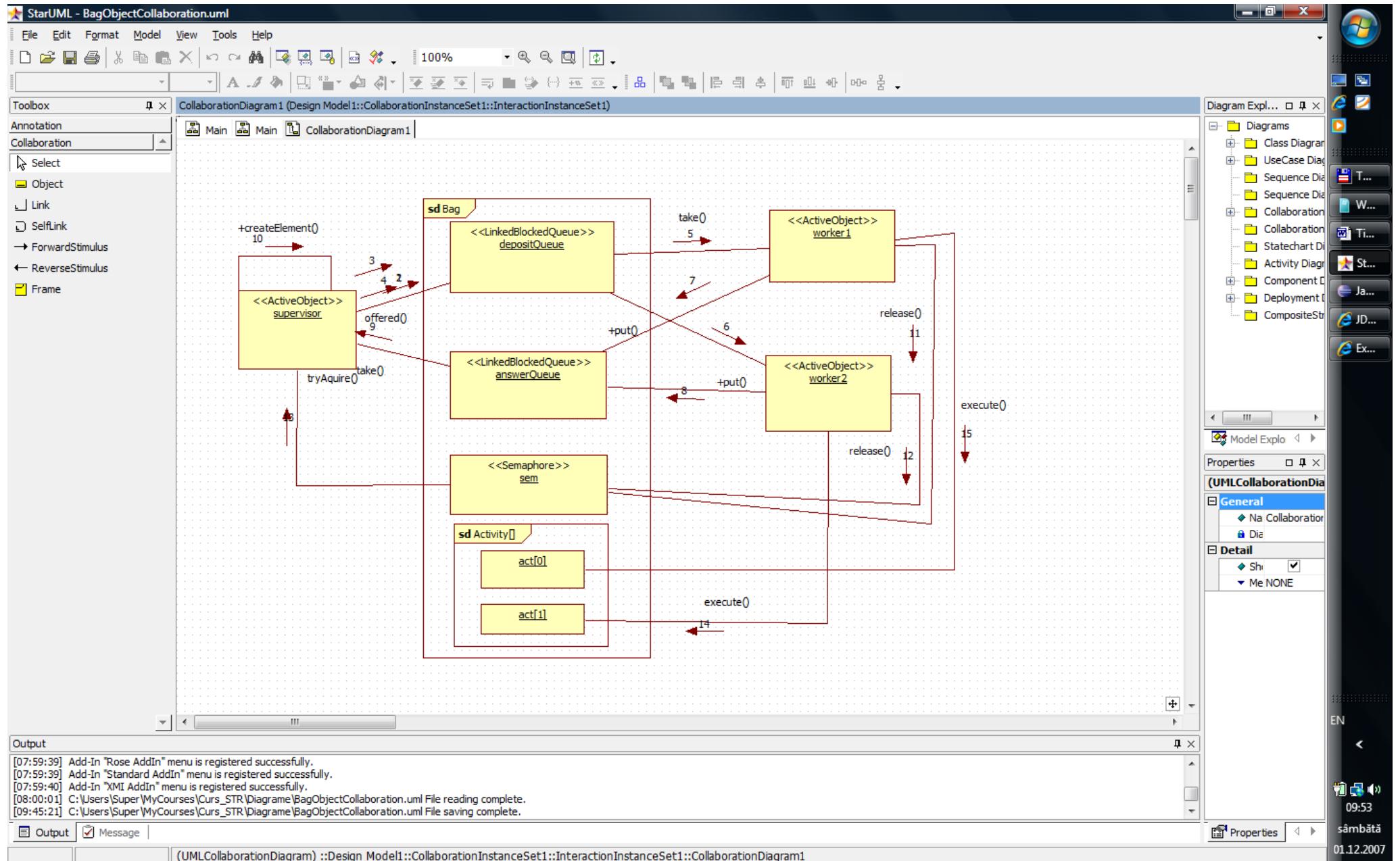
Arhitectura este potrivită pentru sisteme multiprocesor cu un muncitor implementat pe fiecare procesor.

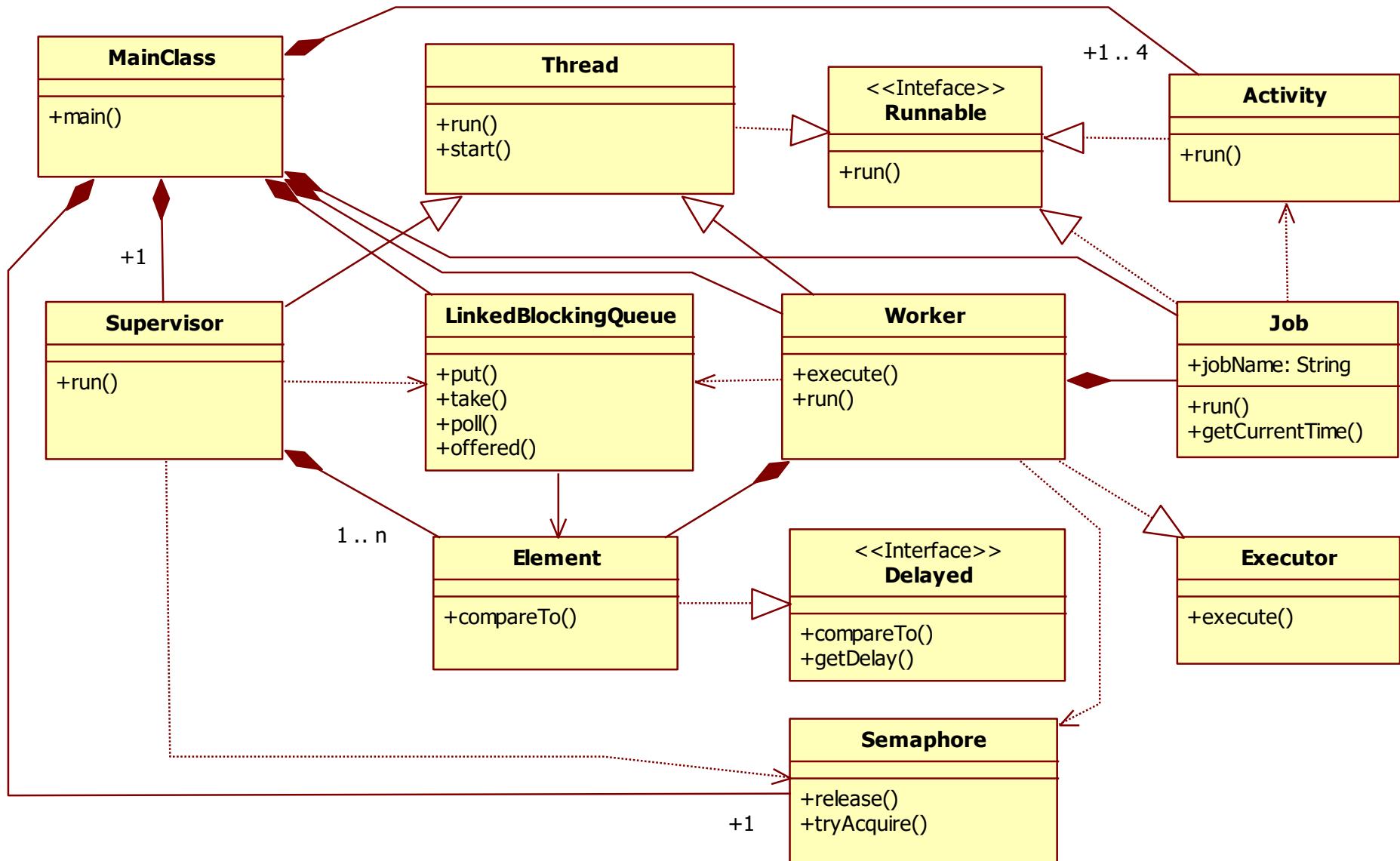
O soluție posibilă:



Object communication diagram for Bag architecture.

Tiberiu Leția : Real-Time Systems – Timing Programming – Java Programming





Class diagram for Bag architecture.

Supervisor's run() method:

```
public void run() {  
    int jobNumber=0;  
    while(stp) {  
        long lng= (long) (4000*Math.random());//get a deadline  
        ms=ms+jobNumber;  
        int k=(int) (4*Math.random());//get an activity  
        jb=new Job(supName,ms,lng,k);//create a job  
        System.out.println("Supervisor created: Job Number ="+jobNumber+";  
                           Activity: "+k+"; Deadline=" + lng+" miliseconds");  
        try{  
            depositQueue.offer(jb,lng,TimeUnit.MILLISECONDS); //deposit job  
        }catch(InterruptedException e) {System.out.println("Interrupted");}  
        System.out.println("Supervisor offered the job "+jobNumber+" at "+  
                           Job.getCurrentTime()+"; Deadline= "+jb.del);  
        try{ Thread.sleep(800);}catch(InterruptedException ex){}  
        if(bool=sem.tryAcquire()) {  
            try{ ans=(Job)answerQueue.take();}catch(InterruptedException ex){}  
        }  
    }  
}
```

```
        System.out.println("Confirmed execution: "+ans.name+" at Time:  
          "+Job.getCurrentTime());  
    }  
    jobNumber++;  
}  
}
```

Worker's methods:

```
public void run() {
    int nr=0;
    while(stp) {
        try { jb= (Job)depositQueue.poll(1000,TimeUnit.MILLISECONDS);}
        catch(InterruptedException ei) {
            System.out.println("Nu este Job in coada la"+Job.getCurrentTime());
        }
        if (jb!=null) {
            jbSource=jb.solicitor;
            ms=jb.name;
            System.out.println(workerName+" executes for: "+ jbSource+"; "+
                ms+" Time: "+Job.getCurrentTime());
            System.out.println("Job =" +ms+ " Deadline=" +jb.del+ " miliseconds" );
            execute(act[jb.activityIndex]);
            Job answer=new Job(workerName,ms,nr,jb.activityIndex);
            try {
                answerQueue.put(answer);
                sem.release();
            }
```

```
        }catch(InterruptedException e) {}  
    }  
}  
  
public void execute(Runnable r){  
    r.run();  
}
```

```
public void execute(Runnable r) {  
    Thread t=new Thread(r);  
    t.start();  
}
```

What's the difference?

The application displays:

- Supervisor created: Job Number =0; Activity: 2;
Deadline=3995 miliseconds
- Supervisor offered the job 0 at H=8 Min=54 Sec=30;
Milisec=1196492070918; Deadline= 3995

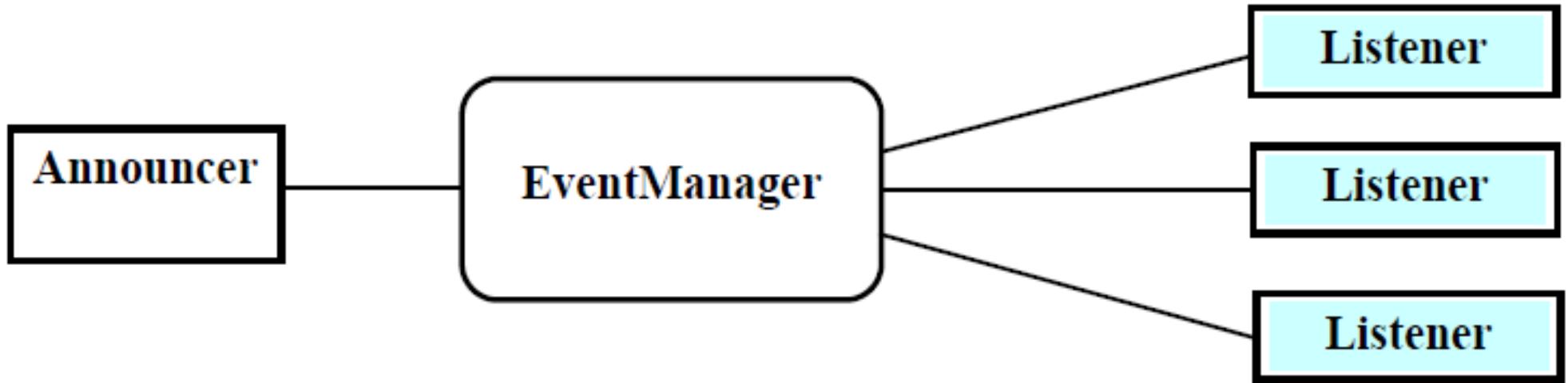
- Worker A executes for: supervisor; Job 0 Time: H=8 Min=54 Sec=30; Milisec=1196492070918
- Job =Job 0 Deadline=3995 miliseconds
- Thread-1 executed ActivityName2 with index 2 at H=8 Min=54 Sec=30; Milisec=1196492070919
- Confirmed execution: Job 0 at Time: H=8 Min=54 Sec=31; Milisec=1196492071719
- Supervisor created: Job Number =1; Activity: 1; Deadline=1539 miliseconds
- Worker B executes for: supervisor; Job 01 Time: H=8 Min=54 Sec=31; Milisec=1196492071719
- Job =Job 01 Deadline=1539 miliseconds
- Thread-2 executed ActivityName1 with index 1 at H=8 Min=54 Sec=31; Milisec=1196492071720
- Supervisor offered the job 1 at H=8 Min=54 Sec=31; Milisec=1196492071720; Deadline= 1539
- Confirmed execution: Job 01 at Time: H=8 Min=54 Sec=32; Milisec=1196492072520

- Supervisor created: Job Number =2; Activity: 3;
Deadline=1003 miliseconds
- Worker B executes for: supervisor; Job 012 Time: H=8
Min=54 Sec=32; Milisec=1196492072520
- Job =Job 012 Deadline=1003 miliseconds

Does the application fulfill the deadlines?

What can be done if it doesn't fulfill them?

3) Announcer-Listener (Crainic-ascultător)



Event based architecture

Event ≠ message

An event is a signal emitted by an entity when a specified state is attained. A message is an item of data sent by an entity to a specified destination. The sender and the receiver of a message are decoupled, unlike the signaler and the handler of an event. The event is emitted locally, while the message is transmitted to a distributed destination.

Este un exemplu de **arhitectură bazată pe evenimente**.

Crainicul anunță că s-a produs un eveniment.

Ascultătorii pot alege să recepționeze un subset de evenimente prin înregistrarea unui **șablon** în gestionarul de evenimente. Numai evenimentele care se potrivesc cu șablonul lui sunt semnalate mai departe ascultătorului.

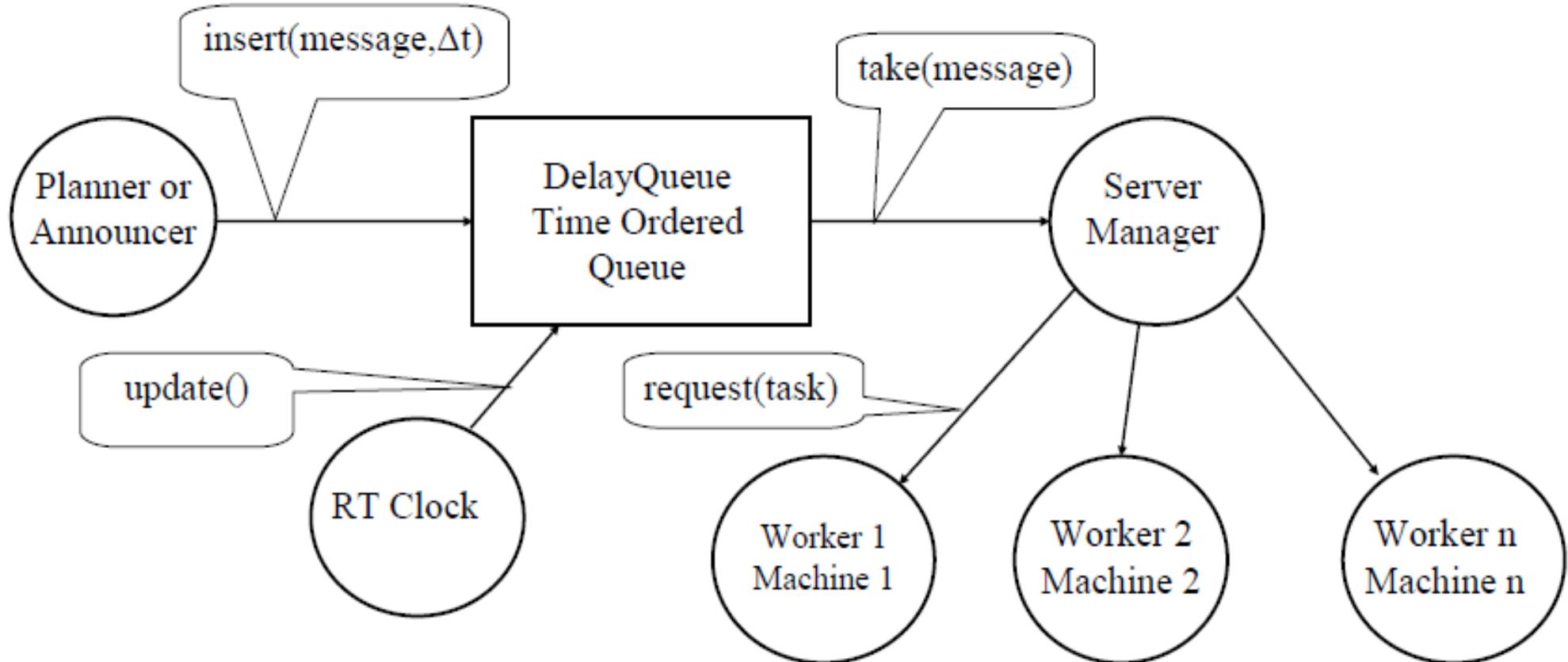
Arhitectura se poate aplica și **recursiv**.

Crainicul este izolat – nu cunoaște câți ascultători sunt activi.

Mecanismul se numește **invocare implicită**.

Cum se implementează aşa ceva?

4) Planned tasks execution



The message contains the requested activity and the time (delay) to perform it.

General real-time questions:

Can be detected if the application doesn't meet the deadlines? Is the fulfilling of the timing constraints dependent of a particular computer?

What can be done if the application doesn't fulfill the timing requirements?

Can be known before starting (a priory) if the application meets the deadlines?

Can be detected what stave off the application reactions?

*

END

*

Realtime Java

sau / OR

Real-Time Java

Realtime Java - Content

Justificarea extensiei Realtime Java

1. Direcțiile de dezvoltare ale extensiei Realtime Java

2. Timpul

a. Timpul în specificații

b. Clase pentru gestionarea timpului

c. Clase pentru temporizări

3. Gestionarea memoriei

4. Fire de timp-real

a. Diagrama claselor interfeței de timp-real

b. RealtimeThread

c. NoHeapRealtimeThread

5. Asincronismul

6. Planificarea

7. Sincronizarea

și

1. Justification of Realtime Java extension
2. Realtime Java extension developments
3. Time
 - a. Specifications of time
 - b. Classes for time management
 - c. Classes for timing
4. Memory management
5. Real-time Threads
 - a. Class diagrams and R-T interface
 - b. Realtime Thread
 - c. NoHeapRealtimeThread
6. Asynchronism
7. Scheduling
8. Synchronization

Întrebări

1. Care sunt facilitățile care trebuie să le ofere un limbaj de programare pentru a realiza aplicații de timp-real implementate pe obiecte?
2. Cum se creează obiecte de timp-real (real-time objects)?
obiecte
 - pasive
 - active
3. Unde se implementează activitățile executate de obiectele active? → run()
4. Care sunt relațiile unui obiect activ cu obiectele pasive ce sunt solicitate pentru efectuarea unor calcule sau furnizarea unor informații?
5. Care este nivelul de prioritate la care se execută activitățile obiectelor pasive?
6. Ce se întâmplă cu un obiect activ care solicită o activitate unui obiect pasiv ce este implementată de o metodă (synchronized) ocupată de un alt obiect pasiv?
7. Ce se întâmplă dacă un obiect activ solicită executarea unei metode (diferită de run()) dintr-un alt obiect activ?
8. Ce se întâmplă dacă un obiect activ cere executarea unei metode dintr-un obiect pasiv care pentru moment nu se află în memorie deoarece nu are loc?

9. Ce se întâmplă la semnalizarea unui eveniment extern? – Cum și când reacționează obiectul responsabil de tratarea evenimentului?
10. Cum se poate verifica respectarea îndeplinirii cerințelor de timp-real de către aplicațiile implementate prin obiecte?

1. Justification of Realtime Java extension

Standard Java deficiencies for RT

1. Time definition and utilization
2. Memory relations
 - allocation
 - object instantiation
 - access
3. File creation and access
4. Scheduling
 - lack of strict thread priorities
 - true preemption
 - priority inversion
 - sync. and async. event processing
 - access to hardware interrupts

RT Java Justification

GC is executed:

- on request: System.gc()
- on demand: when new() is invoked
- in background: when the system is idle

RT Java key features

- 1) Precise memory management
- 2) Direct memory access
- 3) Asynchronous communication
→ threads are safely interrupted
- 4) Precise timing specification
- 5) RT threads
- 6) Thread scheduling

1. Justificarea extensiei Realtime Java

Motivele pentru care Java standard (sau convențională) nu este potrivită pentru dezvoltarea sistemelor de timp-real sunt:

1. Java standard este slabă din punctul de vedere al *lucrului cu timpul*. Este nevoie de mai multe facilități pentru sincronizarea firelor (obiectelor) cu timpul fizic.
2. Java standard este slabă din punctul de vedere al *relației cu memoria internă*:
 - are un control prea grosier al alocării memoriei interne și nu are acces la memoria de bază (engl.: raw memory)
 - nu se pot crea obiecte care sunt rezidente în memoria internă, chiar și atunci când pentru moment nu este nevoie de ele
 - crearea obiectelor poate necesita un interval de timp variabil ceea ce poate duce la nerespectarea unor cerințe de timp-real dacă acestea sunt dependente de obiectele nou create
 - nu are durete deterministe pentru accesarea zonelor de memorie
3. Java standard nu are durete deterministe pentru *crearea și accesarea fișierelor*.
4. *Planificarea* din Java standard este nesatisfăcătoare deoarece:
 - JVM este concepută să fie implementată pe cât mai multe sisteme de operare (platforme) și doar puține dintre acestea oferă facilități de timp-real
 - este slabă din punctul de vedere al planificării firelor pentru execuție:

- are prea puține niveluri de prioritate (numai 10), iar dintre acestea multe nu sunt funcționale pe unele sisteme de operare (vezi Microsoft Windows xx)
- planificarea în Java standard este slab definită pentru a permite implementarea ei pe cât mai multe platforme
- colectorul de reziduuri are un nivel de eligibilitate pentru execuție fix (mult mai mare decât celealte fire Java, sau poate fi mai mic decât toate) ceea ce poate duce la întârzierea unor fire critice din punctul de vedere al timpului; când lucrează GC?
- nu acceptă implementarea unor reguli de planificare impuse de către utilizatori
- lipsește preemțiunea firelor ceea ce face ca unele fire mai puțin importante să întârzie execuția unor fire cu prioritate mai ridicată

Bollela, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M., Belliardi, R., 2000, *The Real-Time Specification for Java*. The Real-Time for Java Expert Group. Ed. Addison-Wesley, ISBN 0-201-70323-8. - www.rtj.org.

**Real-Time Core extensions by J-Consortium
- www.j-consortium.org**

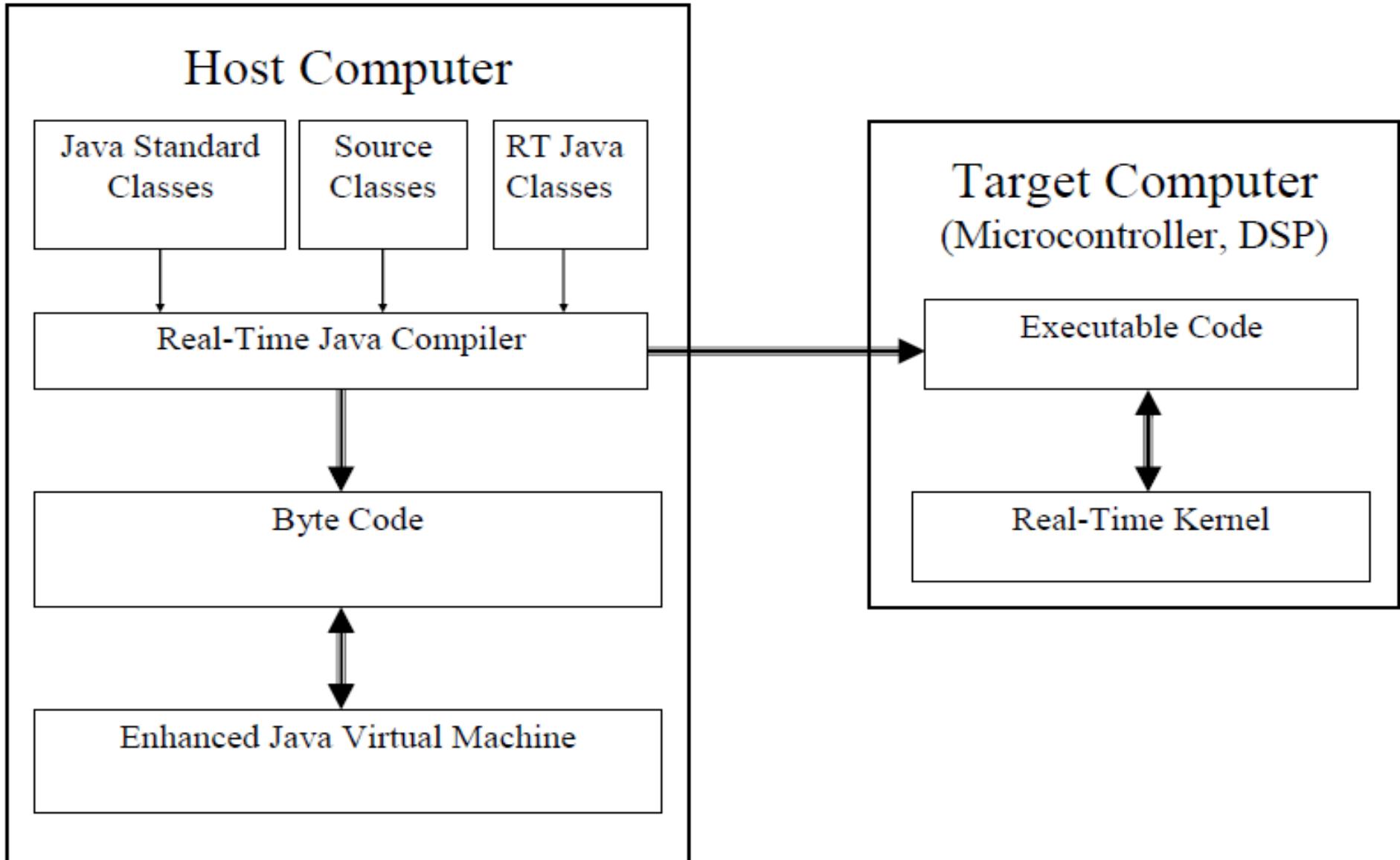
RTSJ version 1.0.1 in 2005 (free!)

2. Realtime Java extension developments

1. GC with bounded latency
2. Real-Time thread relationships
 - a. accurate times (relative, absolute)
 - b. RT clocks
3. Synchronization
4. Asynchronous event handling
5. Asynchronous termination
6. Thread scheduling → mutual exclusion with priority inversion avoidance
7. Memory management

2. Direcțiile de dezvoltare ale extensiei Realtime Java

1. S-au introdus clase noi și metode noi pentru lucrul cu timpul.
2. S-au introdus tipuri noi de memorii și moduri diferite de accesare și gestionare a acestora.
3. S-au introdus clase noi cu o semantică mai puternică pentru crearea firelor de timp-real și pentru gestionarea acestora implicând:
 - precizarea parametrilor firelor de timp-real
 - verificarea realizabilității planificării (execuției) firelor de timp-real
 - semnalarea nerespectării cerințelor de timp-real și implementarea unor handler-e pentru tratarea acestor excepții
 - posibilitatea implementării unor planificatoare de către utilizatori
4. S-au introdus mecanisme evolute pentru semnalarea și tratarea evenimentelor asincrone.
5. S-a făcut posibil transferul asincron al controlului



Real-Time Java Utilization

Realtime Thread – Example

```
class RealtimeThread extends Thread
```

```
RealtimeThread (RTT)
```

RTT is executed as a `java.lang.Thread` in the heap as a memory current context but it has a higher priority than standard threads, garbage collector, just-in-time compiler or any other running in the system.

```
import javax.realtime.RealtimeThread;
public class MyTestRTT {
    public static void main(...) {
        RealtimeThread rtt= new RealtimeThread() {
            public void run() {
                // RT executed code
            }
        };
        rtt.start();
    }
}
```

3. Time

Physical time synchronization

Clasa *java.util.Timer* (Standard Java)

- facilitate pentru fire periodice
- lucrează în background

High resolution time

Clasa *javax.swing.Timer* – produce unul sau mai multe evenimente după o întârziere specificată. Este utilă în animație.

Realtime Java:

- time accuracy - nanoseconds
- absolute time and relative time (time intervals)
- rational time – internal event tick in a relative time

HighResolutionTime Class

Abstract.

Methods:

- absolute, relative times etc.

AbsoluteTime Class

Constructors

Methods:

- time operations
- time comparison
- get time

RelativeTime Class

- time accuracy nanoseconds

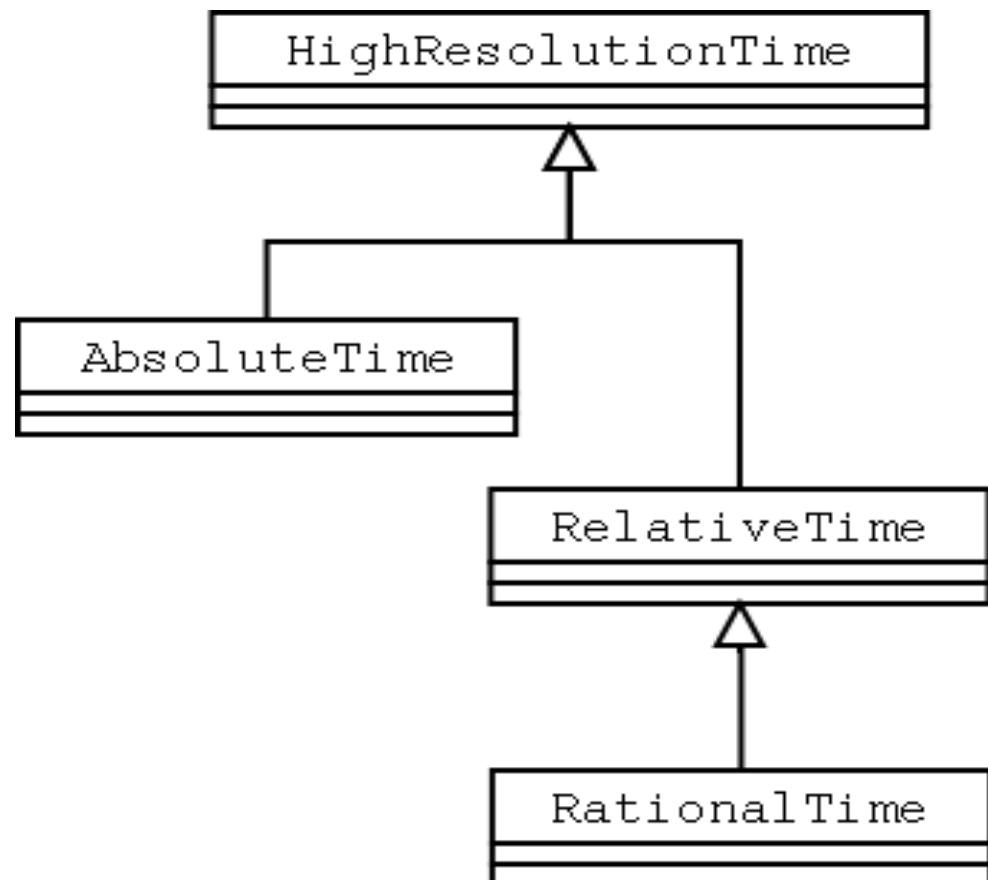
Methods:

- time operations (adds, subtract, comparison etc.)
- get absolute time

RationalTime Class

→ It divides the time intervals → periodic event

No multithreading guarantees.



Timers

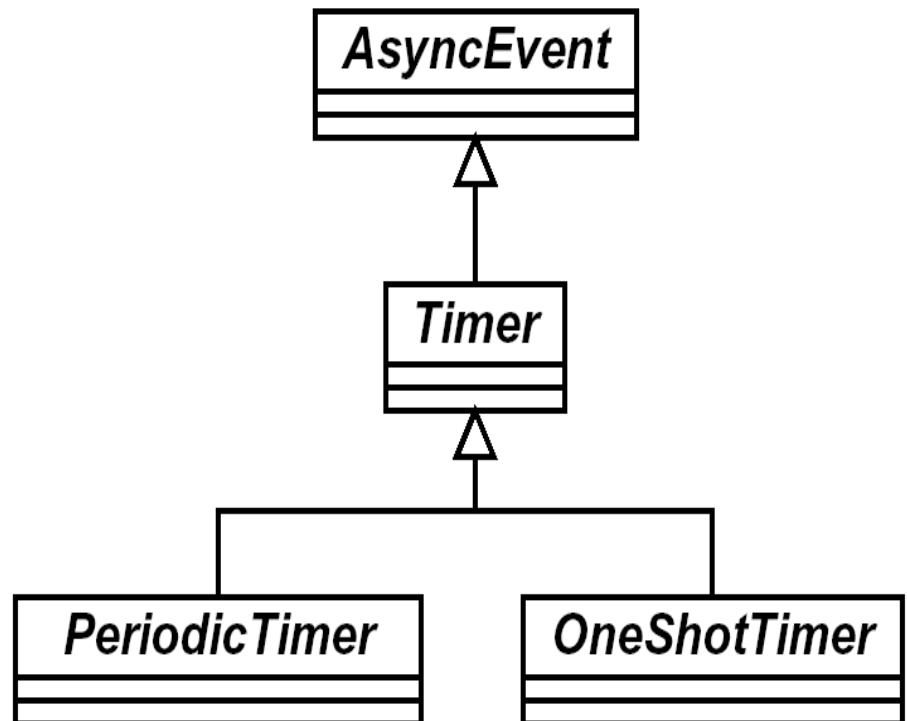
Clock Abstract Class (javax.realtime)

Methods:

getRealtimeClock()
getResolution()
getTime()
setTime()

Real-Time Clock

```
import javax.realtime.*;
public class MainCl {
    public static void main(...) {
        RelativeTime rt=
Clock.getRealtimeClock().getResoloution();
        System.out.println(rt.toString());
    }
}
```



AsyncEvent Class

Methods:

- assigne handlers
- *createReleaseParameters()*
- executes *fire()* etc.

The objects execute a *Runnable*.

Timer Abstract Class

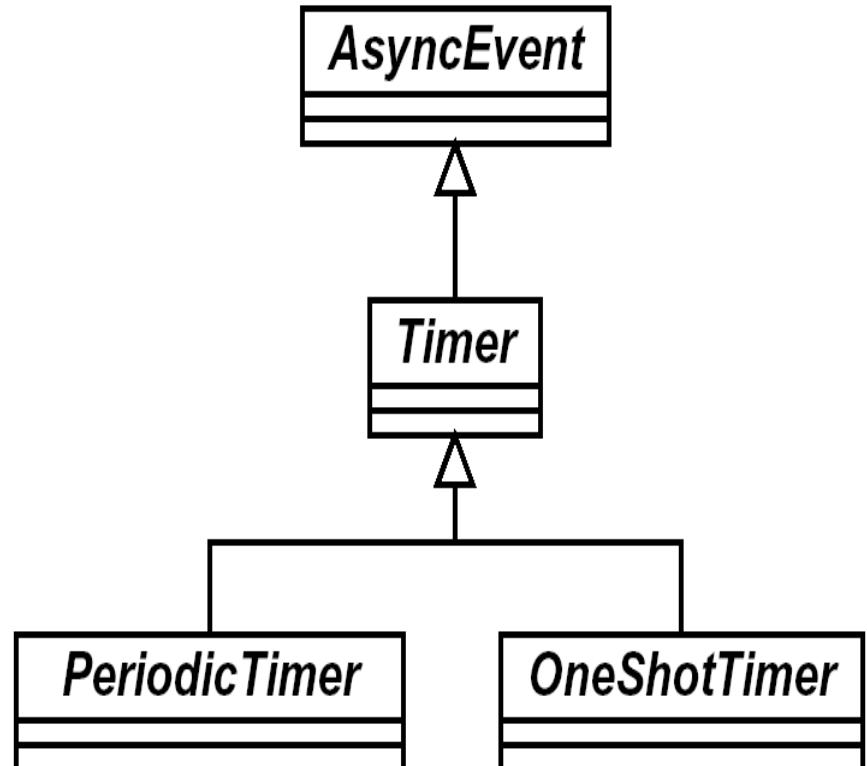
Methods:

- *createReleaseParameters()*
- *getClock()*
- *reschedule(HighResolutionTime time)*
- *start() and stop()*

OneShotTimer Class - permite crearea unui eveniment asincron care se va întâmpla o dată

```
public OneShotTimer (HighResolutionTime start,
Clock clock,
AsyncEventHandler handler)
```

Creează o instanță AsyncEvent pe baza ceasului dat care va executa metoda sa la expirarea timpului dat



Parameters:

start - start time for timer

clock - The timer will increment based on this clock

handler - The AsyncEventHandler that will be scheduled when fire is executed

PeriodicTimer Class

– idem, se execută un obiect AsyncEvent periodic.

4. Memory management

(Gestionarea memoriei)

În Java standard (convențională) memoria este gestionată automat în zona **heap**.

Colectorul de reziduuri lucrează automat – fără controlul programului sau la apelul *System.gc()*.

Necesități:

- un mecanism de alocare a memorie independent de algoritmul GC
- programul să controleze efectele lui GC
- programul să poată aloca și elimina obiectele fără interferență GC

Realtime Java for memory management:

- defines memory zones outside the heap
- defines regions in *scope memory* with limited time duration
- defines objects in memory zone with duration equal to application
- defines regions in a specified physical memory
- defines memory allocation dimension for RT threads
- the program can receive information about GC and can modify the GC behavior.

Unele clase ***MemoryArea*** au un comportament liniar – ignoră variațiile determinate de hardware (caches), iar memoria este alocată relativ polinomial $f(n)$ unde n este dimensiunea obiectului.

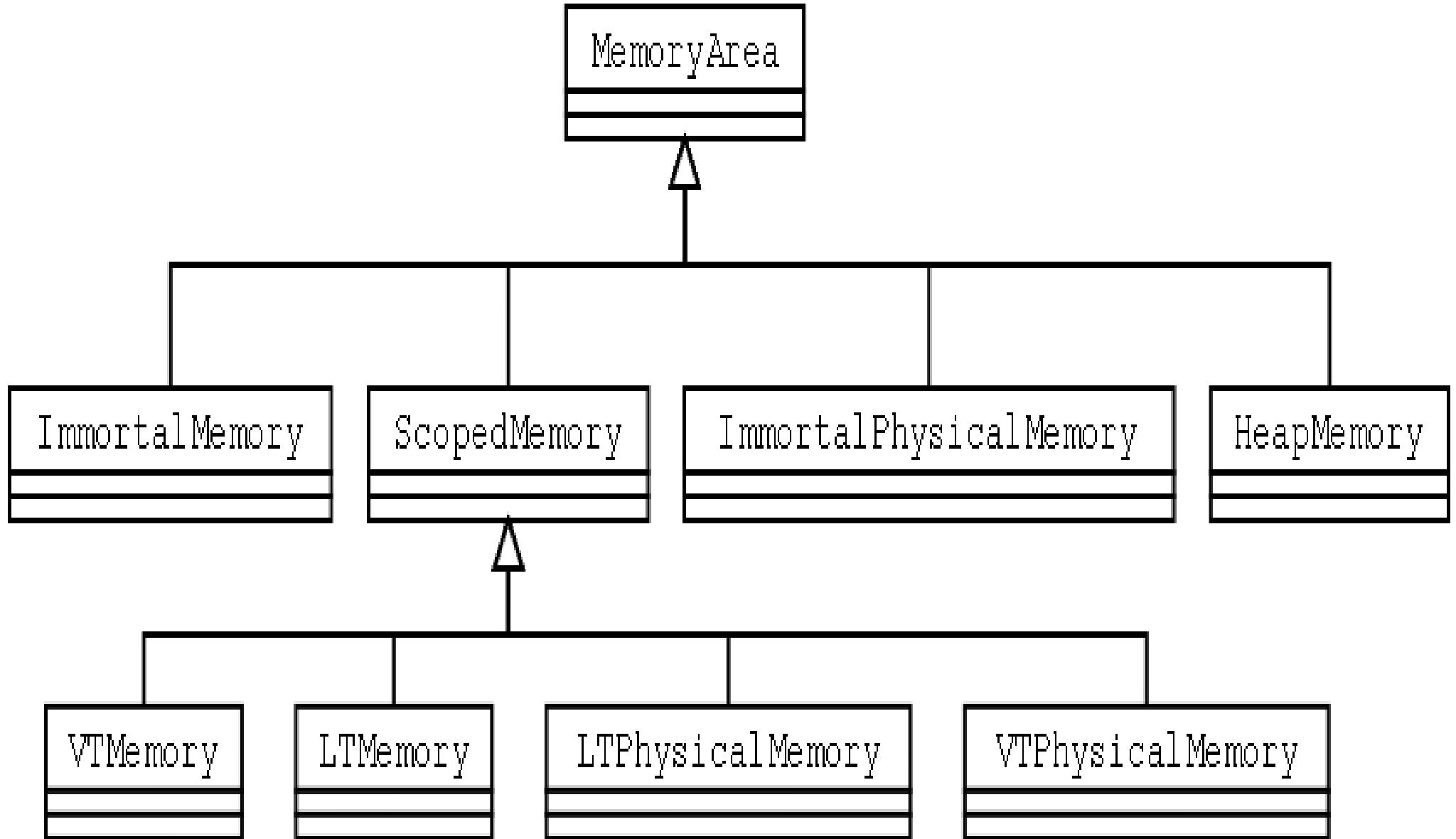
$$f(n) < C \cdot n$$

→ are un timp fix și determinist de alocare

Modelul Realtime Java este inspirat din MemoryRegions

RTSJ definește două tipuri de zone de memorie. Astfel se elimină întârzierile nepredictibile produse deobicei de colectorul de reziduuri (GC):

- Immortal memory ține obiecte care nu sunt distruse decât atunci când se termină programul
- Scoped memory sunt utilizate atunci când un fir lucrează într-o secțiune particulară, sau un domeniu (scope) a unui program, cum ar fi o metodă (de exemplu). Obiectele create în o astfel de memorie sunt automat distruse când firul părăsește domeniul. Ambele tipuri de memorie nu sunt sub acțiunea aleatoare a lui GC.



MemoryArea Class

Abstract class. Methods:

- *enter()* – asociază zona de memorie de firul de timp-real curent pe durata execuției metodei *run()* – a firului transferat ca argument la construire
 - *enter(Runnable logic)* – idem dar pentru firul dat ca argument
 - public void
executeInArea(java.lang.Runnable logic)
throws InaccessibleAreaException
 - public static MemoryArea
getMemoryArea(java.lang.Object object)
 - public long **memoryConsumed()**
 - public java.lang.Object
newArray(java.lang.Class type,
int number) throws
java.lang.IllegalAccessException,
java.lang.InstantiationException
- alocă un tablou în această zonă de memorie

Parameters:

type – The class of the elements of the new array.

number – The number of elements in the new array.

Returns:

A new array of class type, of number elements.

- public java.lang.Object
newInstance(java.lang.Class type)
throws java.lang.IllegalAccessException,
java.lang.InstantiationException
- public void
executeInArea(java.lang.Runnable logic)
throws InaccessibleAreaException

HeapMemory Class

(singleton)

Permite alocarea din memorii *scoped* de obiecte în heap-ul Java

Mehtods:

public static HeapMemory **instance()**

Returns a pointer to the singleton HeapMemory space.

Returns:

The singleton HeapMemory object.

public long **memoryRemaining()**

Description copied from class: MemoryArea

An approximation to the total amount of memory currently available for future allocated objects, measured in bytes.

Overrides:

memoryRemaining in class MemoryArea

Returns:

The amount of remaining memory in bytes

ImmortalMemory Class

Este o resursă de memorie împărțită cu alte fire. Obiectele alocate în ea trăiesc până la sfârșitul aplicației.

Sunt eliminate de GC – unii algoritmi cer scanarea ei.

Un obiect nemuritor poate conține referințe numai la obiecte nemuritoare sau la obiecte din heap.

Obiectele nemuritoare nu sunt eliminate de GC – continuă să existe și după ce nu mai sunt referite.

```
public static ImmortalMemory instance()
```

Returns a pointer to the singleton ImmortalMemory space.

Returns:

The singleton ImmortalMemory object.

ImmortalPhysicalMemory Class

Pot fi mai multe instanțe ale ei

```
public ImmortalPhysicalMemory(java.lang.Object type,  
    long size) throws java.lang.SecurityException,  
    SizeOutOfBoundsException,  
    UnsupportedPhysicalMemoryException,  
MemoryTypeConflictException
```

Parameters:

type – An Object representing the type of memory required (e.g., *dma*, *shared*) – used to define the base address and control the mapping.

size – The size of the area in bytes.

ScopedMemory Class

Abstract class.

Rezolvă problema spațiului de memorie cu o durată limitată.

Zona este validă atât timp cât există un fir R-T.

Se creează câte o referință pentru fiecare accesoriu dinul dintre modurile:

- se creează un fir R-T cu un obiect ScopedMemory asociat lui
- un fir R-T execută metoda *enter()* pentru zona de memorie

Când se termină execuția metodei *enter()* se execută finalizarea (eliminarea) pentru toate obiectele din zona de memorie.

Există reguli de referire a obiectelor din și spre această zonă de memorie.

Subclasses:

LTMemory – LinearTime:

- este alocată liniar
- instanțele ei nu sunt în Java heap

LTPhysicalMemory: idem

- dar pentru memorie fizică

VTMemory – VariableTime

- alocată în timp variabil
- obiectele construite aici nu sunt sub acțiunea lui GC

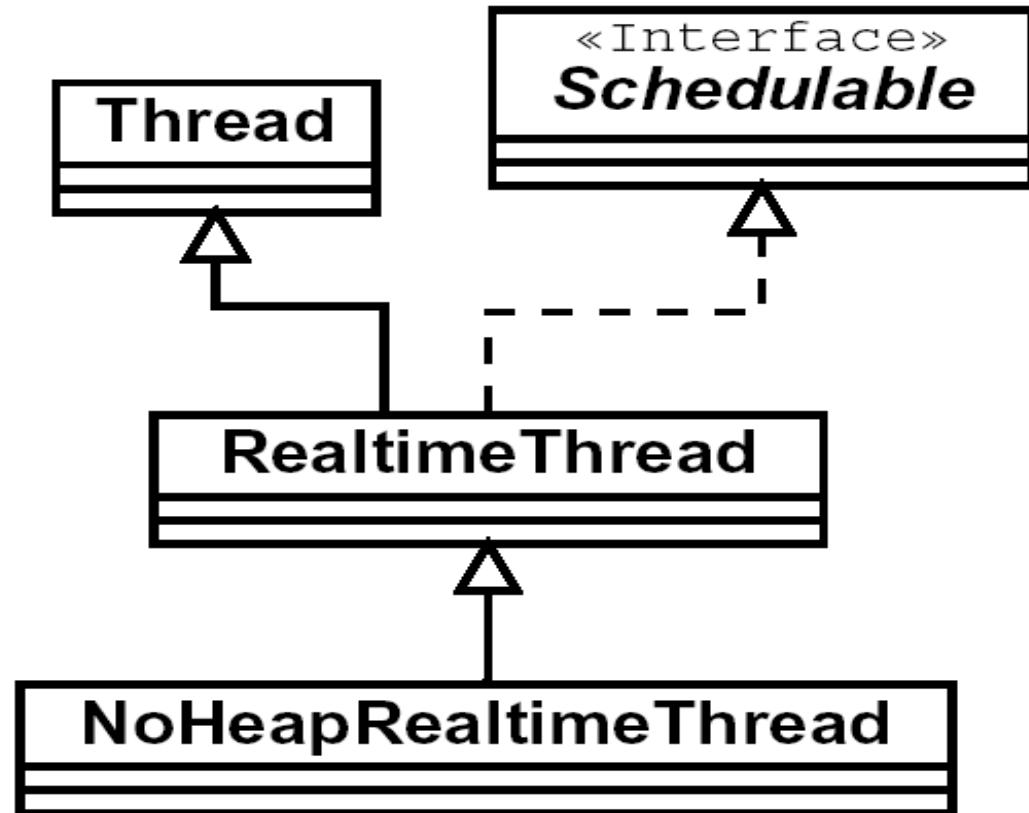
VTPhysicalMemory

5. Real-Time Threads

(Fire de timp-real)

Real-Time Thread:

- has classes for thread creation with scheduling parameters
- allows the memory zones allocation outside the heap
- allows the semantics for scheduling of asynchronous events
- allows preemption



Using NoHeapRealtimeThread (NHRTT)

NHRTT cannot access the heap.

NHRTT is not synchronized with the GC → It can preempt the GC

NHRTT objects must be created or started in *scoped memory region* or *immortal memory*.

```
import javax.realtime.*;
public class MyTestNHRTT {
    public static void main(...) {
        NoHeapRealtimeThread nhrtt= new NoHeapRealtimeThread(null,
            ImmortalMemory.instance())
        {
            public void run() {
                // NHRTT execution code
            }
        };
    }
}
```

It is executed inside Immortal Memory
NO scheduling parameters

Interface **Schedulable**

```
public boolean addToFeasibility()  
public boolean removeFromFeasibility()  
public MemoryParameters getMemoryParameters()  
public void setMemoryParameters(MemoryParameters memory)  
public Scheduler getScheduler()  
public void setScheduler(Scheduler scheduler)  
           throws java.lang.IllegalThreadStateException  
public void  
setSchedulingParameters(SchedulingParameters scheduling)  
public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)  
public boolean setReleaseParametersIfFeasible(ReleaseParameters release)  
public boolean setMemoryParametersIfFeasible(MemoryParameters memParam)
```

RealtimeThread Class

```
public RealtimeThread(SchedulingParameters scheduling,  
ReleaseParameters release,  
MemoryParameters memory,  
MemoryArea area,  
ProcessingGroupParameters group,  
java.lang.Runnable logic)
```

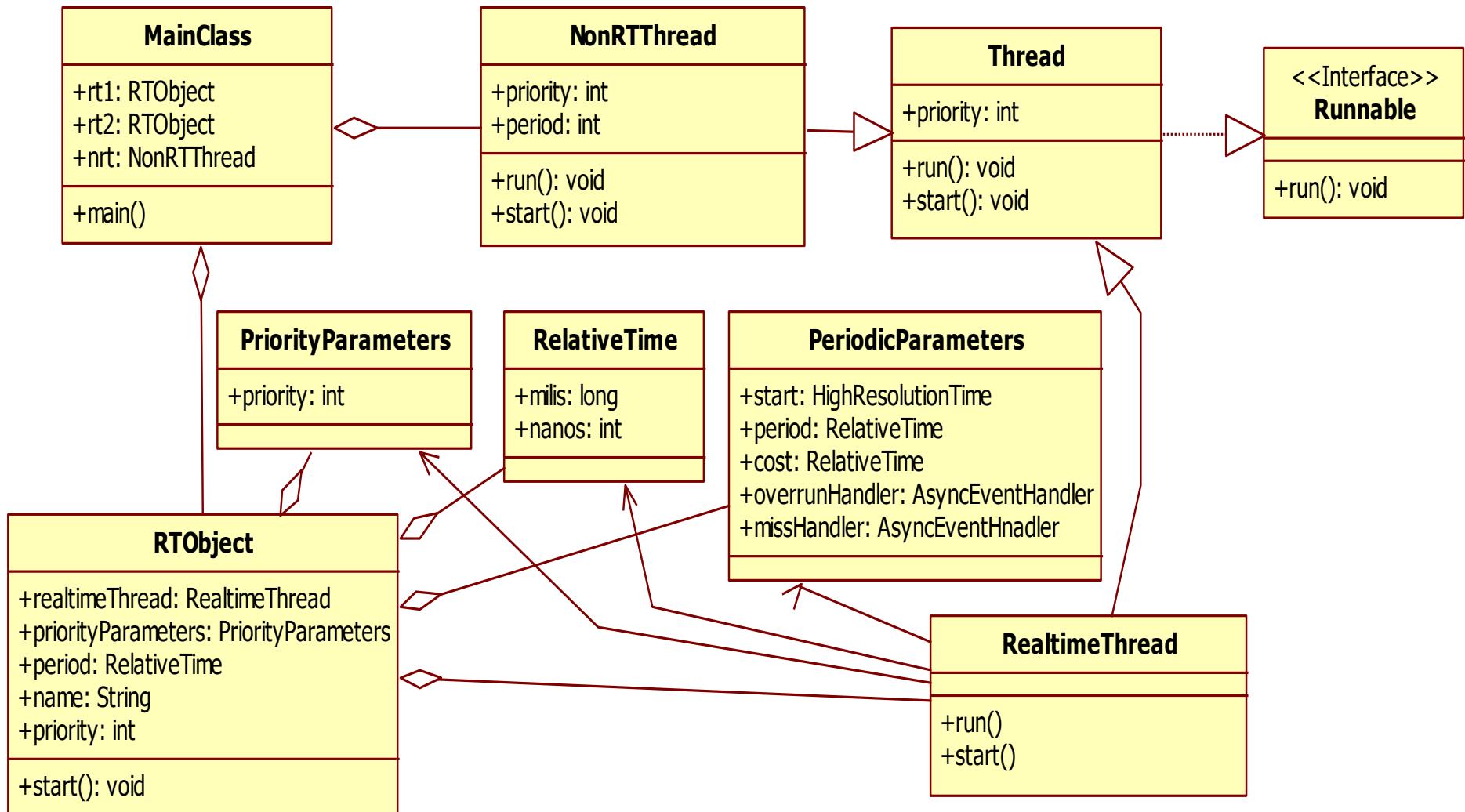
```
public static RealtimeThread currentRealtimeThread()  
throws java.lang.ClassCastException
```

```
public boolean waitForNextPeriod()  
throws java.lang.IllegalThreadStateException
```

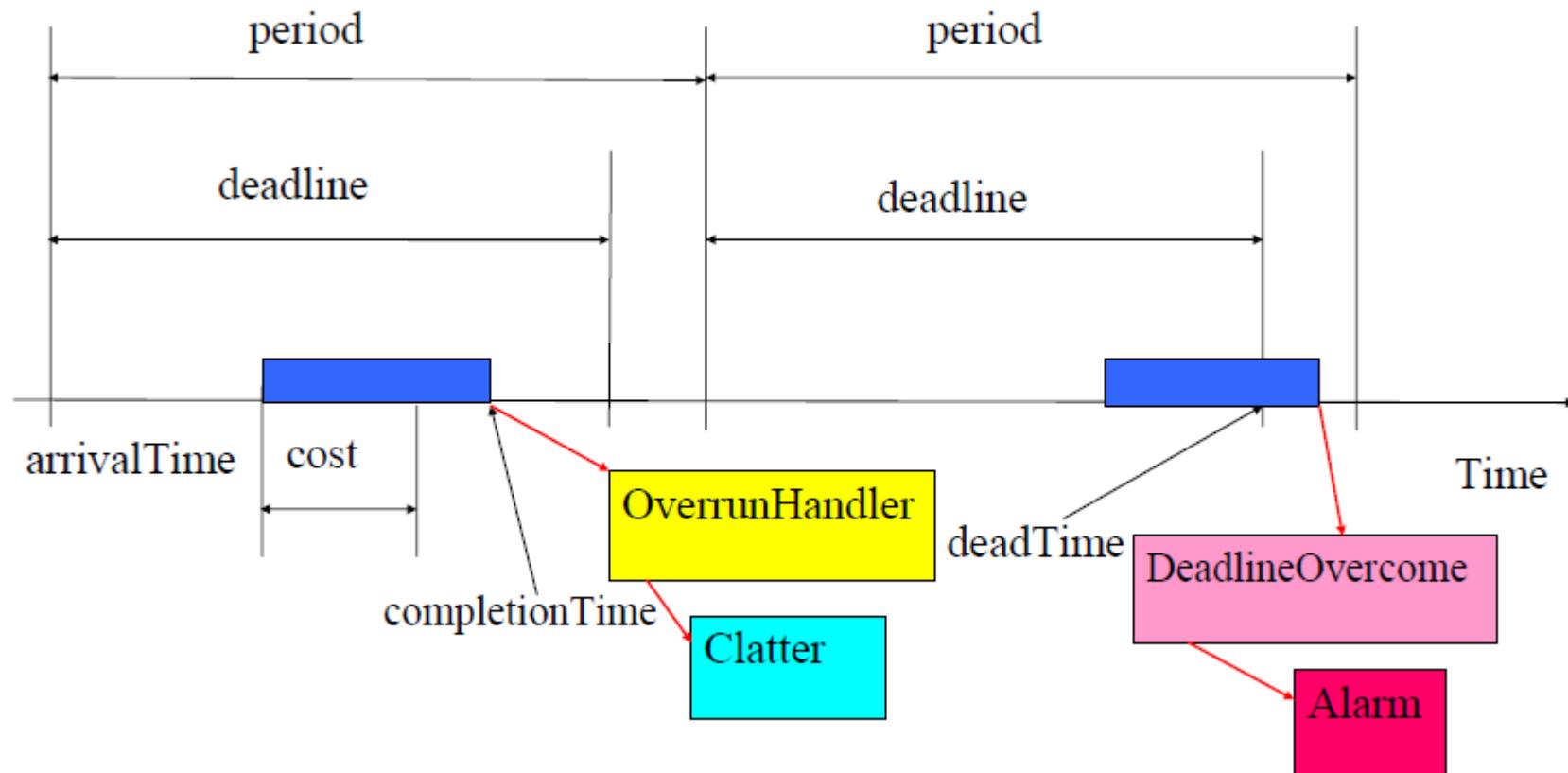
```
public static MemoryArea getCurrentMemoryArea()
```

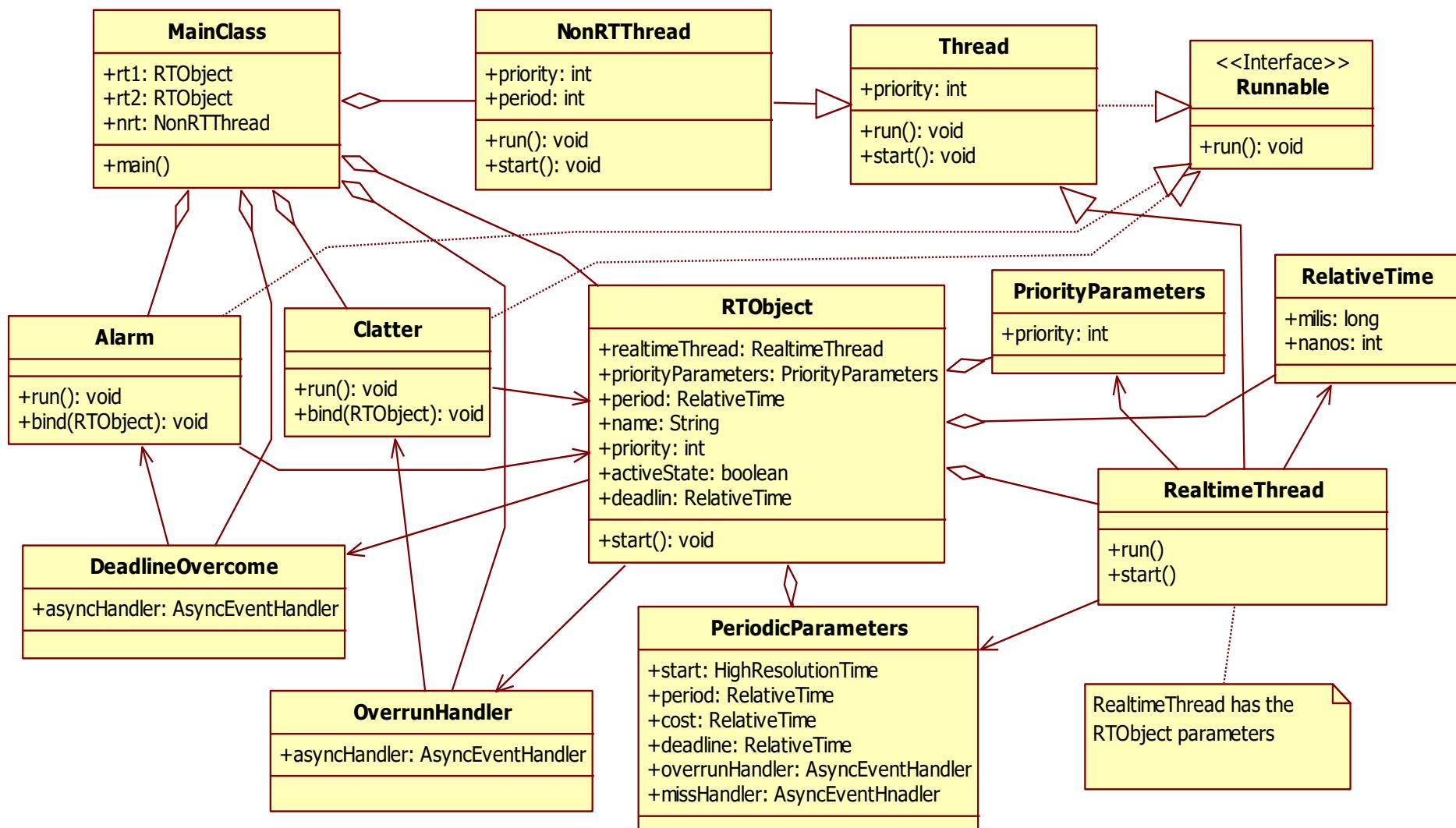
NoHeapRealtimeThread Class

- obiectele ei pot preemta pe GC
- nu se acceptă să refere, să manevreze sau să aloce obiecte din heap



Test Application for Cost Overrunning and Deadline Missing





6. Asynchrony

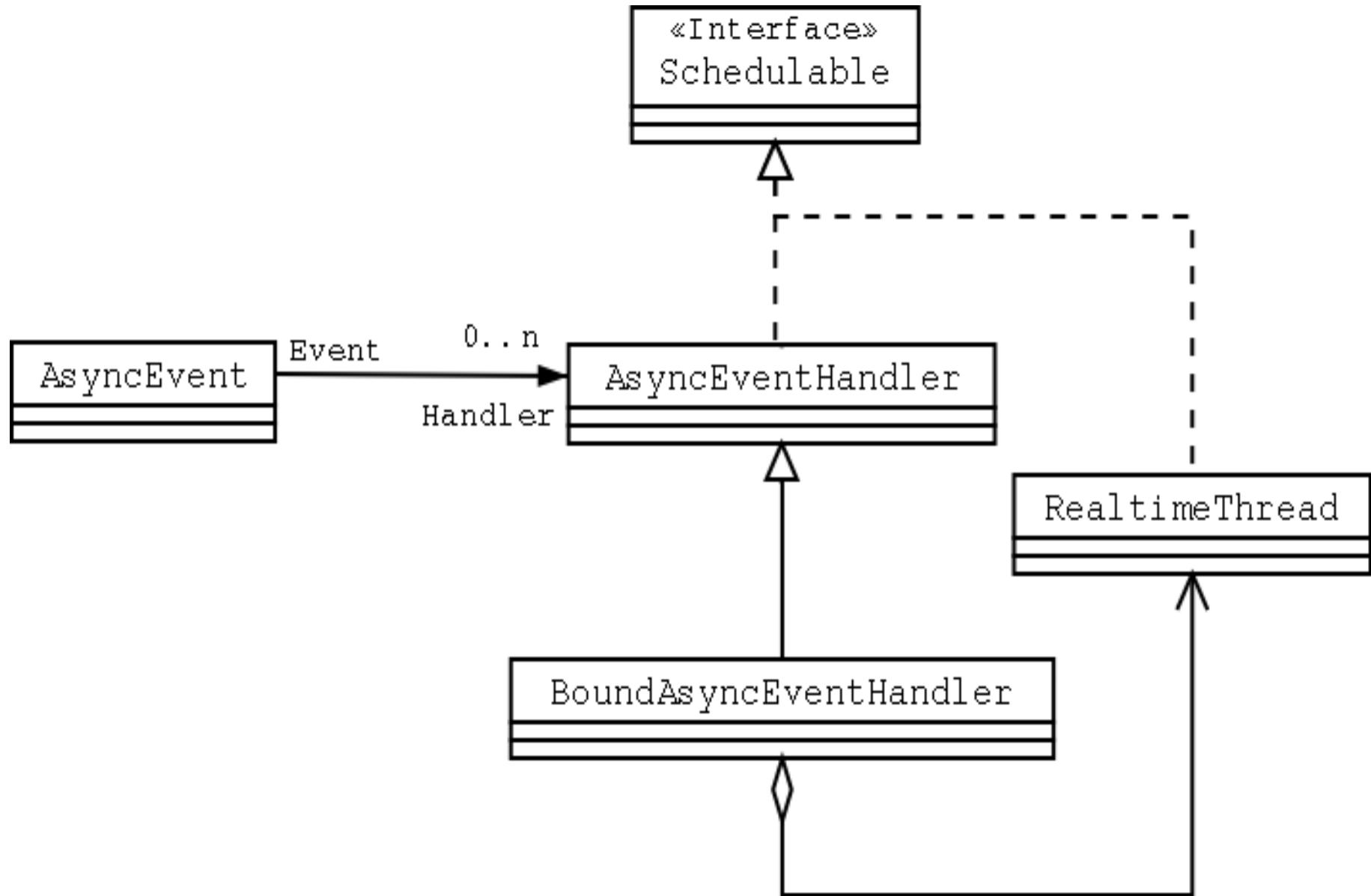
(Asincronismul)

- tratarea asincronă a evenimentelor – adaptarea la asincronismul lumii reale → planificarea reacției programului la evenimente
- transferul asincron al controlului – la schimbări dure în lumea reală trebuie să se transfere execuția logică în alt punct → extinde mecanismul excepțiilor din Java – schimbă locul de control cu alt fir – transfer în mod asincron
- terminarea asincronă a firelor – la schimbări drastice în lumea reală trebuie să se transfere controlul în exteriorul lor și în mod normal

Conține clase care leagă producerea unui eveniment intern sau extern de execuția unui program logic.

În Java convențională se putea utiliza *Thread.interrupt()* care:

- îintrerupe execuție firului ← Nu lucrează bine – e depreciată. Idem *stop()*
- dacă e blocat pe *wait()*, *wait(long)*, *wait(long,int)*, *sleep(long)*, *sleep(long,int)* semnalează o excepție *InterruptedException*
- dacă e blocat pe o operație I/O îintrerupe canalul de comunicație



Tratarea evenimentelor asincrone în Realtime Java

Se bazează pe interfața *Interruptible* (care are metoda)

```
public void  
run(AsynchronouslyInterruptedException exception)  
    throws AsynchronouslyInterruptedException
```

și pe

```
public class AsynchronouslyInterruptedException  
extends java.lang.InterruptedException
```

Se aruncă o excepție la încercarea de transfer asincron a controlului pentru un
RealtimeThread.

Schedulable Interface

public interface Schedulable extends java.lang.Runnable

Metode:

- public boolean **addToFeasibility()**
- public MemoryParameters **getMemoryParameters()**
- public ReleaseParameters **getReleaseParameters()**
- public Scheduler **getScheduler()**
- public SchedulingParameters **getSchedulingParameters()**
- public void
setMemoryParameters(MemoryParameters memory)
- public void
setReleaseParameters(ReleaseParameters release)
- public void **setScheduler(**Scheduler scheduler)
throws java.lang.IllegalThreadStateException
- public void
setSchedulingParameters(SchedulingParameters scheduling
)

- public boolean **setSchedulingParametersIfFeasible**(
SchedulingParameters scheduling)
- public boolean
setReleaseParametersIfFeasible(ReleaseParameters releas
e)
- public boolean **setMemoryParametersIfFeasible**(
MemoryParameters memParam)

AsyncEventHandler Class

implementează `java.lang.Runnable`, Schedulable

Instanțiere → codul care este planificat să răspundă la producerea unui eveniment.

Ex. de constructor:

```
public AsyncEventHandler (SchedulingParameters scheduling,  
                         ReleaseParameters release,  
                         MemoryParameters memory,  
                         MemoryArea area,  
                         ProcessingGroupParameters group,  
                         boolean nonheap)
```

Methods:

- `run()`
- `public void handleAsyncEvent()` - dacă s-a conceput un alt fir pentru a răspunde
- celelalte metode din interfața *Schedulable*

Clasa **BoundAsyncEventHandler**

implementează *Schedulable*

Poate fi utilizată pentru a se garanta că are asociat un fir permanent dedicat.

```
public  
BoundAsyncEventHandler(SchedulingParameters scheduling,  
                 ReleaseParameters release, MemoryParameters memory,  
                 MemoryArea area, ProcessingGroupParameters group,  
                 boolean nonheap, java.lang.Runnable logic)
```

POSIXSignalHandler Class

este utilizată pentru tratarea semnalelor UNIX

Are câmpuri:

static int SIGALRM, SIGINT, SIGCLD etc.

Methods:

- public static void **addHandler**(int signal,
AsyncEventHandler handler)
- public static void **removeHandler**(int signal,
AsyncEventHandler handler)
- public static void **setHandler**(int signal,
AsyncEventHandler handler)

Asynchronous Transfer Control

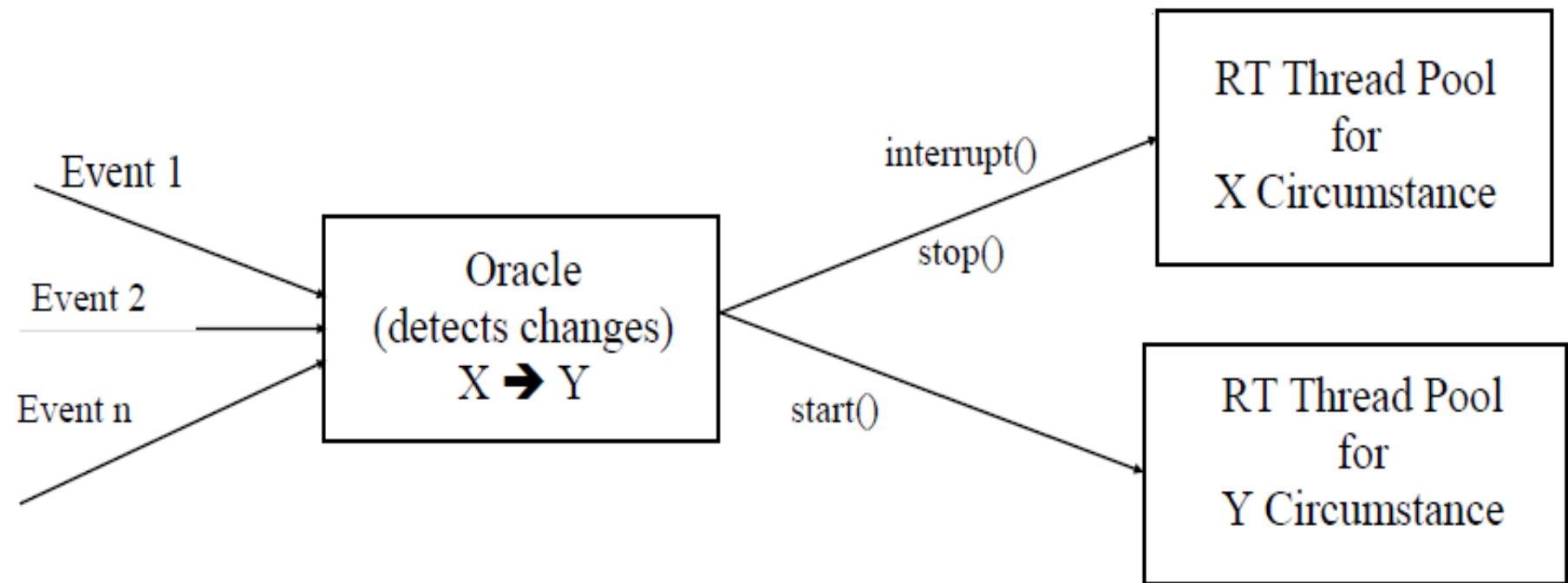
Uneori durata de calcul a unui algoritm este foarte mult variabilă. De exemplu un algoritm iterativ care produce rezultate rafinate la fiecare iterație.

Controlul asincron al transferului de la o prelucrare la codul transmis de rezultat → transfer la expirarea timpului de execuție

Terminarea asincronă a firelor de timp real (Asynchronous Real-Time Thread Termination)

Aceasta face posibil ca toate metodele firelor de timp-real să fie înterruptibile.

Se creează un *oracol* care monitorizează lumea externă prin legarea unui număr de evenimente de schimbarea prelucrării:



7. Scheduling

Planificarea

Scopuri:

- asigură temporizări și execuțiile predictibile ale secvențelor de instrucțiuni.
- să nu se piardă nici un deadline

Se generalizează conceptul de eligibilitate.

Scheduling obiect ← orice obiect al unei clase care implementează Schedulable. Este gestionat de planificatorul de bază.

Clasele care dau instanțe planificabile sunt:

- *RealtimeThread*
- *NoHeapRealtimeThread*
- *AsyncEventHandler*

Utilizează 28 de niveluri de întrerupere (Java standard are 10) cu priorități fixe de tip preemptiv. O planificare încearcă să optimizeze o măsură particulară (dată de programator). Analiza realizabilității determină dacă planificarea are o valoare acceptabilă pentru metrică.

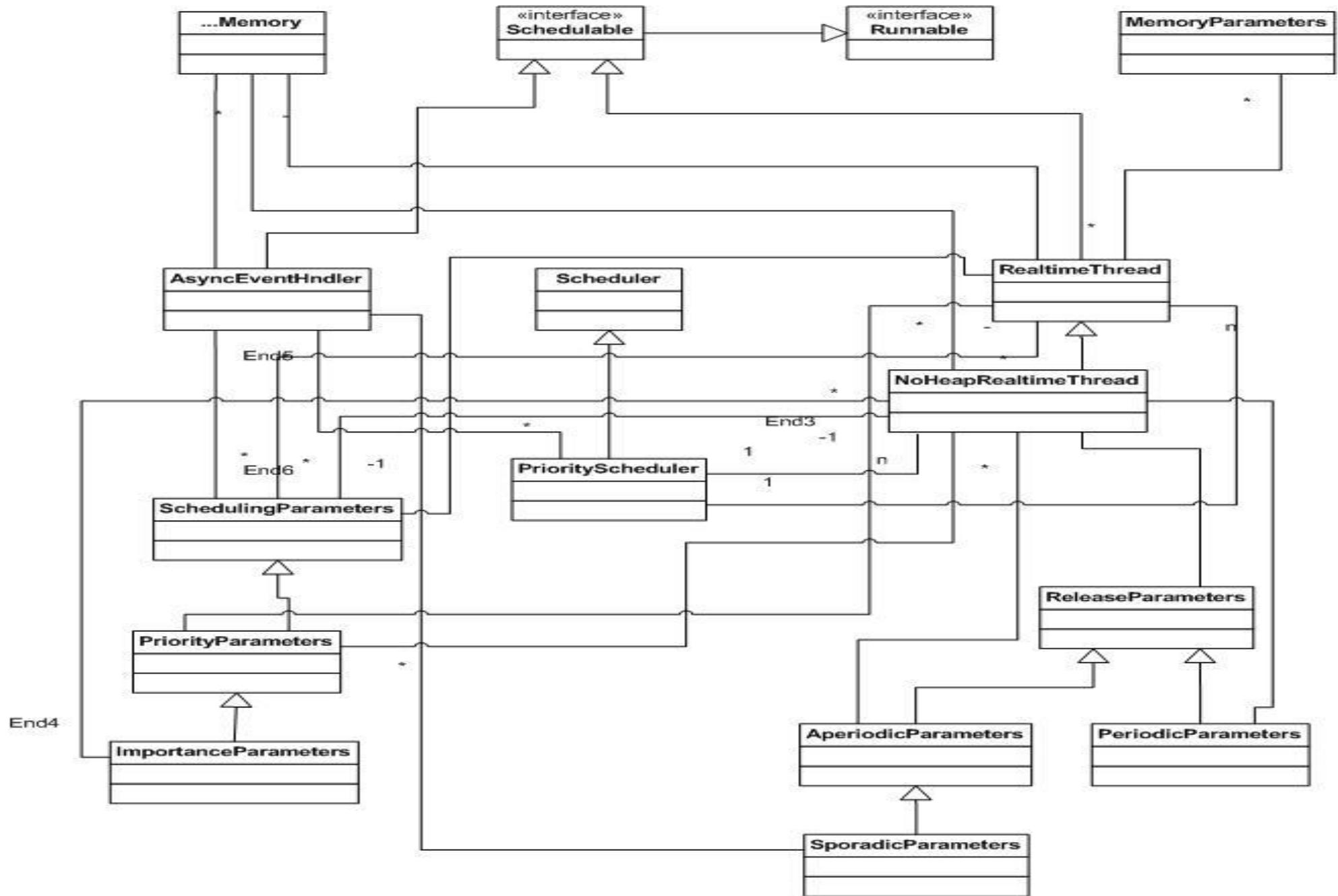
Scheduler Class

- abstract

Creează obiecte planificatoare.

Methods:

- protected abstract boolean **addToFeasibility**(Schedulable schedulable)
- public boolean **setIfFeasible**(Schedulable schedulable,
ReleaseParameters release, MemoryParameters memory)



PriorityScheduler Class

Extinde Scheduler. Este planificatorul de bază.

Methods:

```
public static PriorityScheduler instance()
```

```
protected boolean  
addToFeasibility(Schedulable schedulable)
```

```
protected boolean  
removeFromFeasibility(Schedulable schedulable)
```

```
public boolean setIfFeasible(Schedulable schedulable,  
                           ReleaseParameters release,  
                           MemoryParameters memory)
```

SchedulingParameters Class

Este abstractă. Specifică parametrii de planificare. Instanțele ei determină ordinea de execuție a firelor.

Constructor:

```
public SchedulingParameters()
```

Clasa PriorityParameters

Planificatorul de bază este dat de PriorityScheduler.

Constructor:

```
public PriorityParameters(int priority)
```

Clasa **ImportanceParameters**

În unele sisteme un proces fizic extern determină perioada unor fire.

```
public ImportanceParameters(int priority,  
                           int importance)
```

```
public int getPriority()
```

```
public void setPriority(int priority)  
           throws java.lang.IllegalArgumentException
```

Clasele:

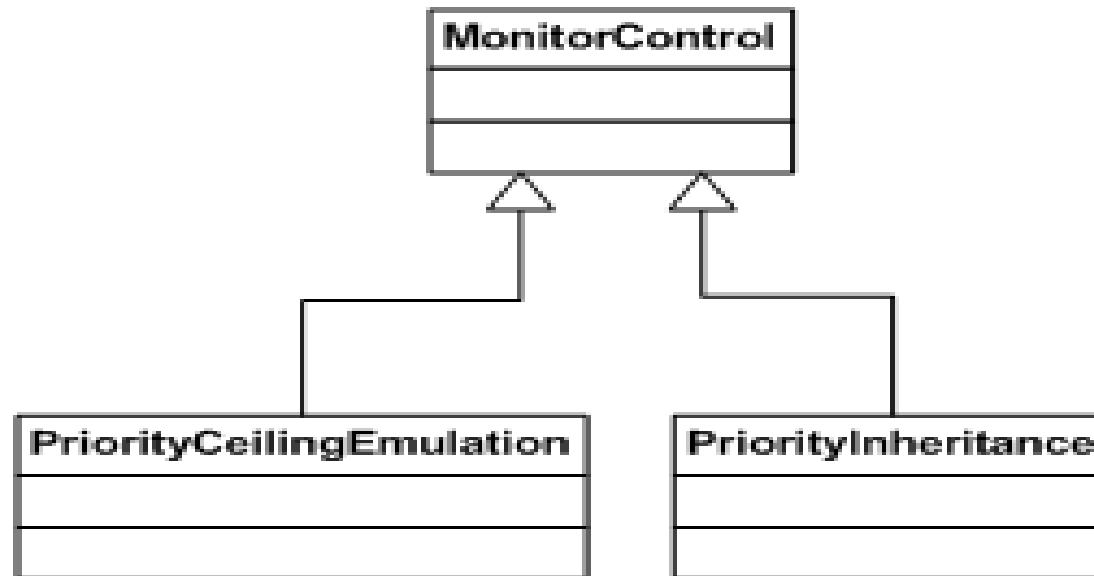
- SporadicParameters
- AperiodicParameters
- ReleaseParameters

8. Thread Syncrhonization

Sincronizarea între fire

Conține clase care permit:

- implementarea algoritmului de plafonare a priorității la obiecte individuale
- setarea implicită a algoritmului de evitare a inversiunii priorității
- comunicarea fără așteptare (wait-free) între firele de timp-real și firele normale (convenționale)



MonitorControl Abstract Class.

Clasa ***PriorityCeilingEmulation*** specifică protocolul de emulare a plafonării priorității.

Afectează prioritatea firelor care intră într-un monitor. Firelor intrate li se ridică prioritatea la prioritatea de bază a monitorului. La ieșire se restaurează prioritatea inițială.

Constructor:

```
public PriorityCeilingEmulation(int ceiling)
```

Clasa ***PriorityInheritance***

Se creează un obiect de acest tip. Un fir care intră în aşteptare pentru un monitor va avea prioritatea efectivă ridicată la prioritatea obiectului. Când iese i se va restaura prioritatea la nivelul anterior.

Priority Avoidance Protocol

Evitarea inversării priorității (priority avoidance):

Dacă un fir t1 încearcă să achiziționeze un zăvor (lock, monitor) ținut de un fir t2 cu o prioritate mai coborâtă, în acest caz prioritatea lui t2 este ridicată la cea a lui t1 atât timp cât t2 deține zăvorul. Recursiv se întâmplă mai departe dacă t2 așteaptă după un fir t3 cu o prioritate mai mică.

Se permite unui programator să suprascrie regulile sistemului.

Priority Ceiling Protocol

Plafonarea priorității

Reguli:

- Un monitor primește o prioritate de plafonare (de plafon = ceiling priority) când este creat. Ea este cea mai mare prioritate a unui fir care poate încerca să-l achiziționeze.
- Când firul intră în codul sincronizat de monitor, prioritatea lui va fi ridicată la cea a monitorului (priority ceiling). Astfel se asigură excluderea mutuală a accesului și în același timp nu poate fi preemtat de nici un fir care ar putea solicita monitorul.
- Dacă printr-o eroare de programare un fir are o prioritate mai mică decât cea a monitorului pe care încearcă să-l achiziționeze se va semnala o excepție.

→ Un fir Java normal nu poate avea o eligibilitate mai mare decât a unui fir RT.

Firele Java normale nu pot avea priorități mai mari decât GC.

Există mecanisme care garantează că *synchronized* evită blocarea firelor RT.

WaitFreeWriteQueue Class

Facilitează comunicarea și sincronizarea între instanțele RealtimeThread și Thread. Accesul la obiectele comune ale celor două poate duce la întârzieri ale execuțiilor datorită lui GC.

Metoda *write()* nu blochează un fir RT care încearcă să scrie într-o coadă plină. Pe o coadă plină i se returnează *false*.

Metoda *read()* este sincronizată și poate fi apelată de mai multe fire.

Constructor:

```
public WaitFreeWriteQueue(java.lang.Thread writer,  
                          java.lang.Thread reader, int maximum, MemoryArea memory)  
    throws java.lang.IllegalArgumentException,  
          java.lang.InstantiationException,  
          java.lang.ClassNotFoundException,  
          java.lang.IllegalAccessException  
  
public void clear()  
public boolean isEmpty()
```

```
public boolean isFull()
public java.lang.Object read()
public int size()
public boolean force(java.lang.Object object)
           throws MemoryScopeException
public boolean write(java.lang.Object object)
           throws MemoryScopeException
```

Similar sunt:

Class WaitFreeReadQueue

Class WaitFreeDequeue – nu are sincronizare

*

* END *

* RT Java *

*

5. Scheduling (Planificarea)

Scheduling Algorithms
Scheduling Tests
Verification of Temporal Constraints
Fulfilling

Content of Cap. 5 Scheduling

1. Introduction. Justification of scheduling
2. Tasks, thread of execution and real-time objects
3. Tasks and threads characteristics
4. Design and implementation of scheduling algorithms
5. Definition of Scheduling Problems
6. Aperiodic Task Scheduling
7. Periodic Task Scheduling. Types of scheduling algorithms and verification methods
8. Periodic Task Scheduling. Algorithms for independent tasks
9. Periodic Task Scheduling. Algorithms for communicating tasks
10. Algorithms for fair execution
11. Algorithms for multiprocessor systems

1. Introduction. Justification of scheduling

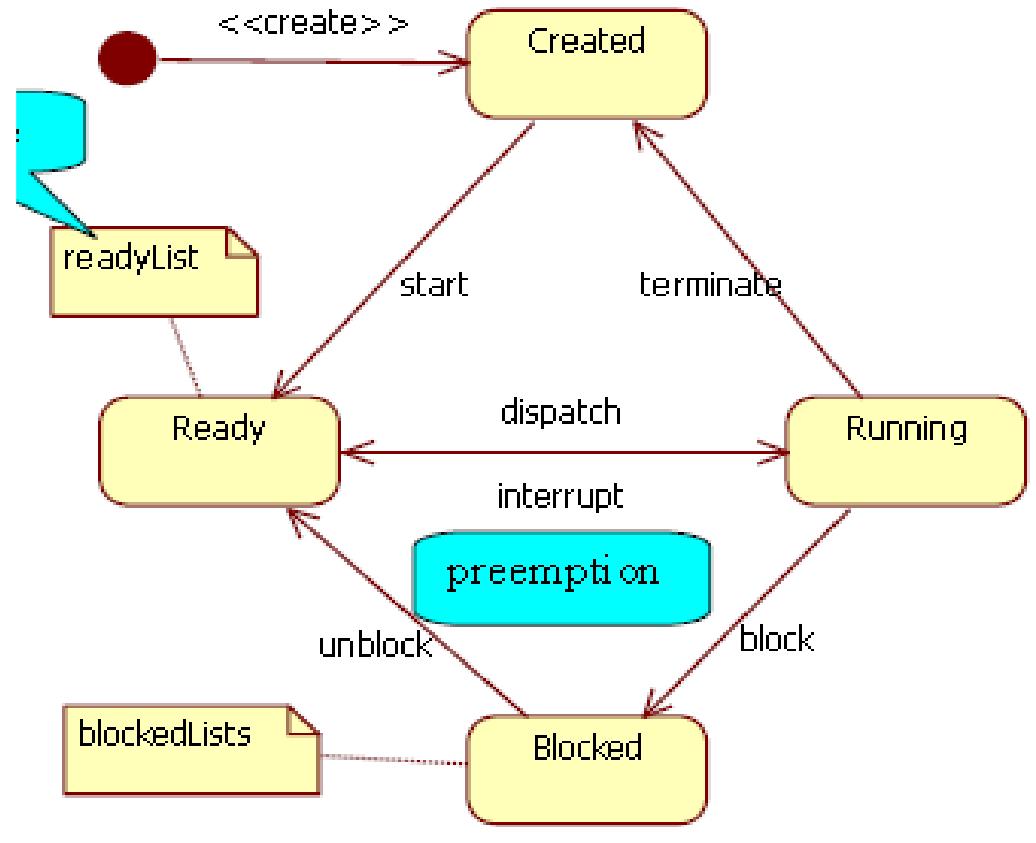
- planning
- scheduling
- dispatching

Scheduling = to order the executions of tasks of a real-time application.

Tasks should react at internal or external events in specified period of times.

Scheduling determines the manner of allocation of processor and other resources to tasks from the **ready queue** such that the real-time requirements are met.

→ *Scheduler* – implemented by operating system or application



Process state machine of a RTS

Scheduling problems are studied in:

- *Operational Research*
 - job shop and flow-shop problems
 - scheduling of machines, orders, batches, projects, ..
 - resources: machines, factory cells, unit processes,..
 - static (off-line) techniques
- *Computer Science*
 - schedule tasks in a mono- or multi-processor environment
 - dynamic techniques

Scheduling subjects:

- *viable (feasible)* scheduling
- set of scheduling tests
- scheduling of a set of tasks with *soft real-time constraints*
- scheduling of a set of tasks with *hard real-time constraints*
- optimal scheduling?
- optimal scheduling algorithm – there is no one better
- *stable scheduling* algorithm – if not all tasks can fulfill their temporal requirements, the lower priority tasks are those that miss the deadlines

Scheduling examples

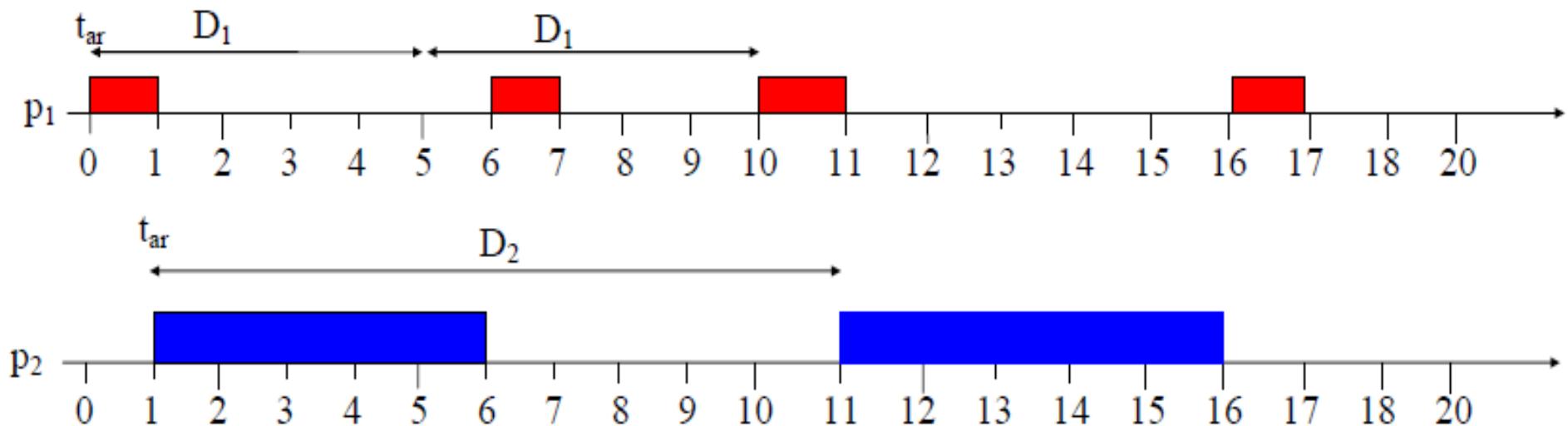
Set of two tasks: p_1, p_2

$T_1 = 5 \text{ ms}$ $T_2 = 10 \text{ ms}$ - period of tasks

$C_1 = 1 \text{ ms}$ $C_2 = 5 \text{ ms}$ Computation time + switching time

$D_1 = T_1 = 5 \text{ ms}$ $D_2 = T_2 = 10 \text{ ms}$ Deadline – response time

Case 1. **Non-preemptive system** with static priorities; p_1 arrives sooner than p_2 (or at the same moment of time).

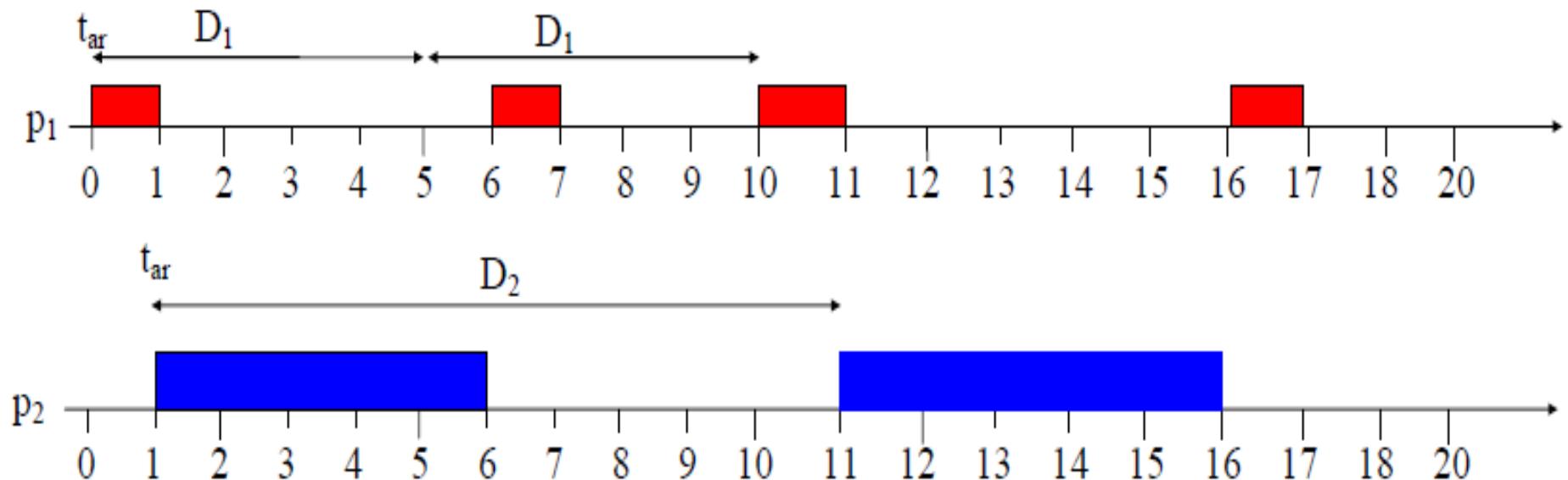


Conclusion: the system fulfills the temporal requirements!

The processor usage:

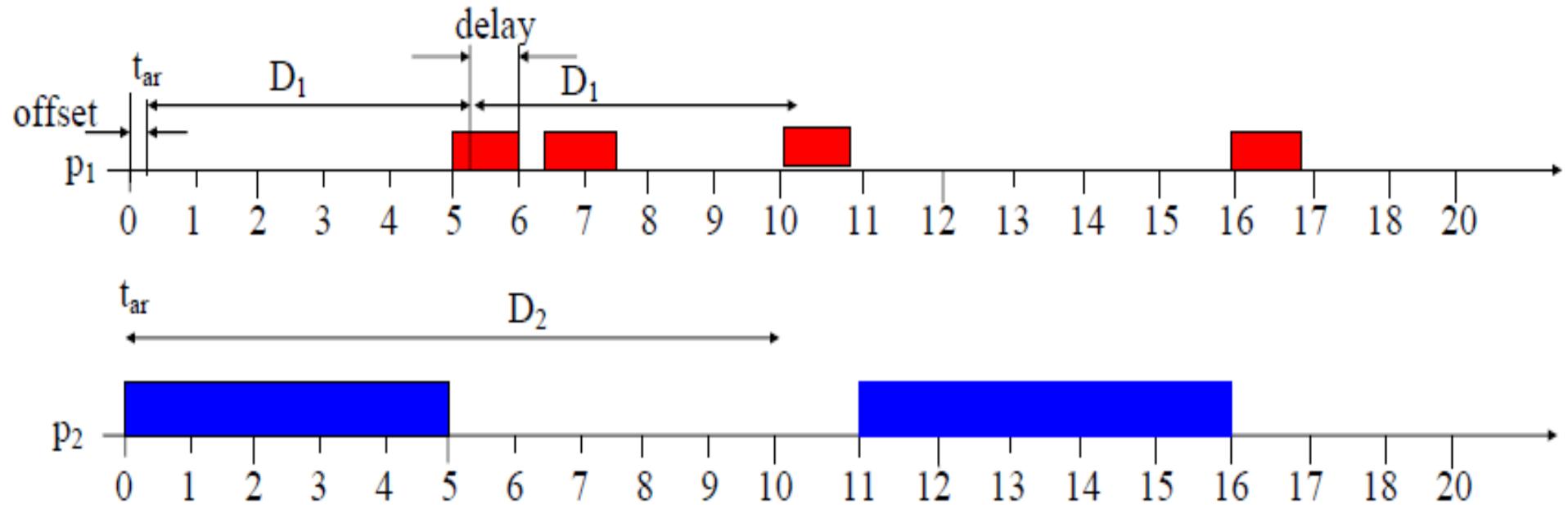
$$\rightarrow 10 \text{ ms} = T_2 = T$$

$$u = (2 \cdot C_1 + C_2) / 10 = 7 / 10 = 0.7 = 70 \%$$



New case: $u=?$

Case 2. **Non-preemptive system** with static priorities; but p_2 arrives sooner than p_1 at least 1 μ s.



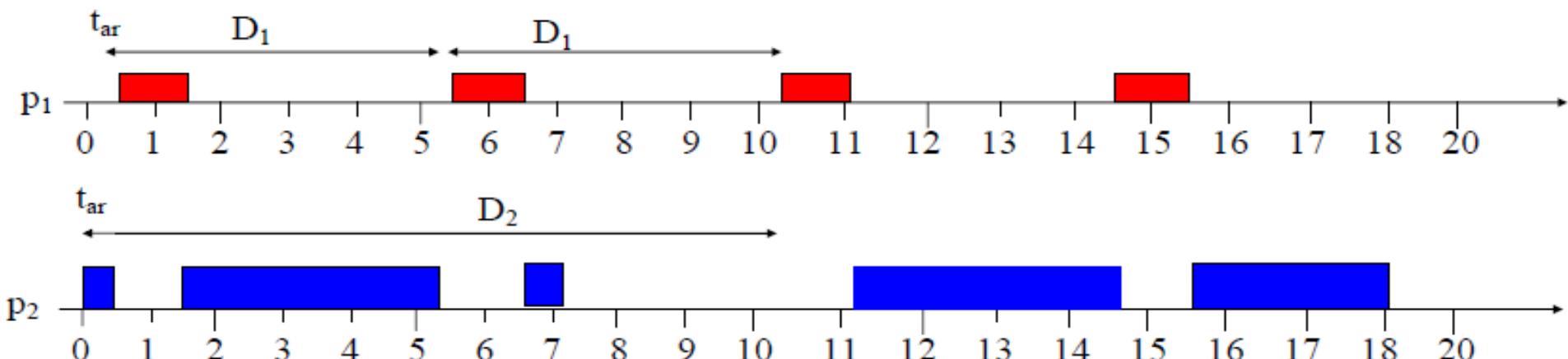
Conclusion for case 2: *the temporal requirements are not fulfilled – even the processor loading is the same.*

Conclusion for the cases 1 + 2: the fulfilling of the temporal requirements depends on the arrival order.

General conclusion: the non-preemptive systems don't guarantee the real-time requirements fulfilling even if the processor is low loaded.

If the task computing times are very short, the fulfilling of temporal constraints can be met.

Case 3. The same set of tasks with a *preemptive system*. $\text{priority}(p_1)$ greater than $\text{priority}(p_2)$

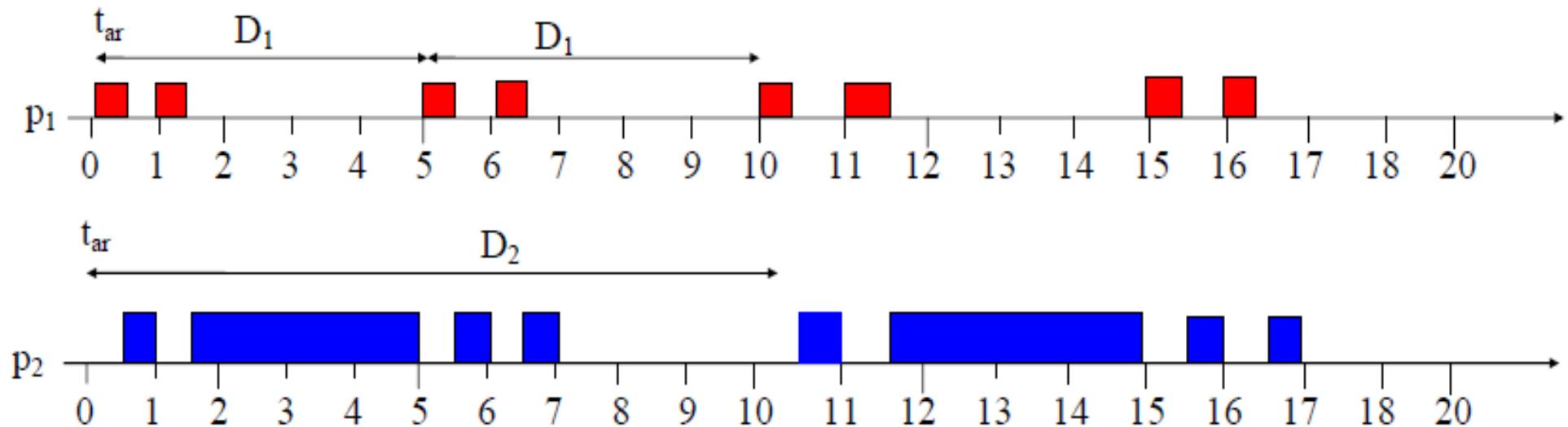


Conclusion: The system fulfills all the deadlines. The order of task arrivals doesn't matter!

Case 4. The same set of tasks implemented on a **time slicing system**.

The system uses an allocation of 0.5 ms for each task indifferent of tasks priority. It uses *round robin algorithm*.

The tasks have the same arrival times.



Conclusion: The system fulfills the deadlines. During the most agglomerated period each task is executed (works) with n times lower speed.

Analysis:

The n tasks are not active all the time.

The worst case is considered. → The n tasks are active during the execution of the current task.

Let τ be the processor slice allocation.

The worst case – the current task gets the processor after all the other $n-1$ tasks have got it. So, it can be delayed with:

$(n-1) \cdot \tau$ t. u. before starting.

If all the n tasks are active the current response is executed during C/n t. u.

The tasks p_i fulfills the deadline if:

$$n \cdot \tau < D_i \text{ and } C_i \cdot n < D_i$$

Conclusion: such a system is acceptable for a low loaded processor and large response times.

Static Cyclic Scheduling ([Planificarea statică ciclică a execuției](#))

Task order is given by deadlines!

Static Cyclic Scheduling

- off-line approach
- configuration algorithm generates an execution table or calendar
- many different algorithms (optimization)
- the table repeats cyclically – static cyclic scheduling
- works for both non-preemptive and preemptive scheduling
- the run-time dispatcher simply follows the table
 - sets up an hardware interrupt at the time when a context switch should be performed (preemptive)
 - starts the first task in the calendar
 - when the hardware interrupt arrives the first task is preempted and next task is run,

2. Tasks, thread of executions and real-time objects

Programs:

- process → tasks
- thread of executions → tasks

Objects.

Real-time objects → transform the problem in task scheduling!

Objects:

- active
- passive

How can be obtained the (worst case) computation times?

3. Characteristics of tasks or threads

Criticalitate (*criticality*) – importanța respectării termenului asociat taskului

Prioritatea dorită (*preference priority*) – importanța unor sarcini și implicit ordinea în care trebuie să fie executate dacă nu pot fi respectate toate cerințele temporale.

Prioritatea de planificare (*scheduling priority*) – obținută în urma aplicării algoritmului de planificare.

Urgență (*urgency*) – critic din punctul de vedere al timpului \neq importanță

Temporal characteristics:

- periodic tasks
- aperiodic tasks
- sporadic (episodic) tasks
- hyperperiodic tasks

Task Parameters

Periodic tasks:

- $C = \text{worst case computation time}$
- $T = \text{period} \rightarrow C < T ; \text{ If } C > T \rightarrow \text{it gathers jobs!}$
- $R = \text{Response time } R -$
- $D = \text{deadline} \rightarrow C < D \text{ and } \text{Requirement } R < D !!!$

$T = D$ or $T > D$;
if $T < D \rightarrow$ could gather jobs

Aperiodic tasks

- C = worst case computation time –
- Minimum duration between two demands (arrival times) $T = T_{\min} = 0$
- If $C > T$ it gathers jobs
- Maximum duration between two demands (arrival times) $T_{\max} = \infty$
- D deadline → $D < T$; If $T=0 \rightarrow C > T \rightarrow$ it gather jobs

Solution:

Sporadic or episodic task:

- C = worst case computation time –
- minimum durations between two demands (arrival times) $T = T_{\min} > 0$
- $C < T$
- maximum durations between two demands (arrival times) $T_{\max} = \infty$
- D deadline → $D < T$ or $D=T$

Hyperperiodic task:

- C = worst case computation time –
- minimum durations between two demands (arrival times) – hiperperiod
 $T_{\min} = C$
- maximum durations between demands (arrival times) $T = T_{\max} = D$
- D = deadline $\rightarrow D > T_{\min}$ and $D = T_{\max}$

Notății:

$D_{arrival}$ – distanță temporală între două cereri successive

D_{term} – distanță temporală maximă între două terminări succesive

Proces Task	$D_{arrival}$		D_{term}		Processor loading	
	Min.	max.	min.	max.	min.	max.
Periodic (C,T,D)	T	T	T	2T-C	C/T	C/T
Hyperperiodic (C,T=D)	C	C	C	T	C/T	C/C=1
Aperiodic (C,D)	0 ? sau C	∞	C	∞	0	1
Sporadic (C,T=T _{min} ,D)	T_{min}	∞	T_{min}	∞	0	$C/T_{min} = C/T$

R – response time $\rightarrow R \leq D$

$R \geq C$

$C \leq T ; C \geq T \rightarrow$ gather jobs

4. Proiectarea și implementarea algoritmilor de planificare

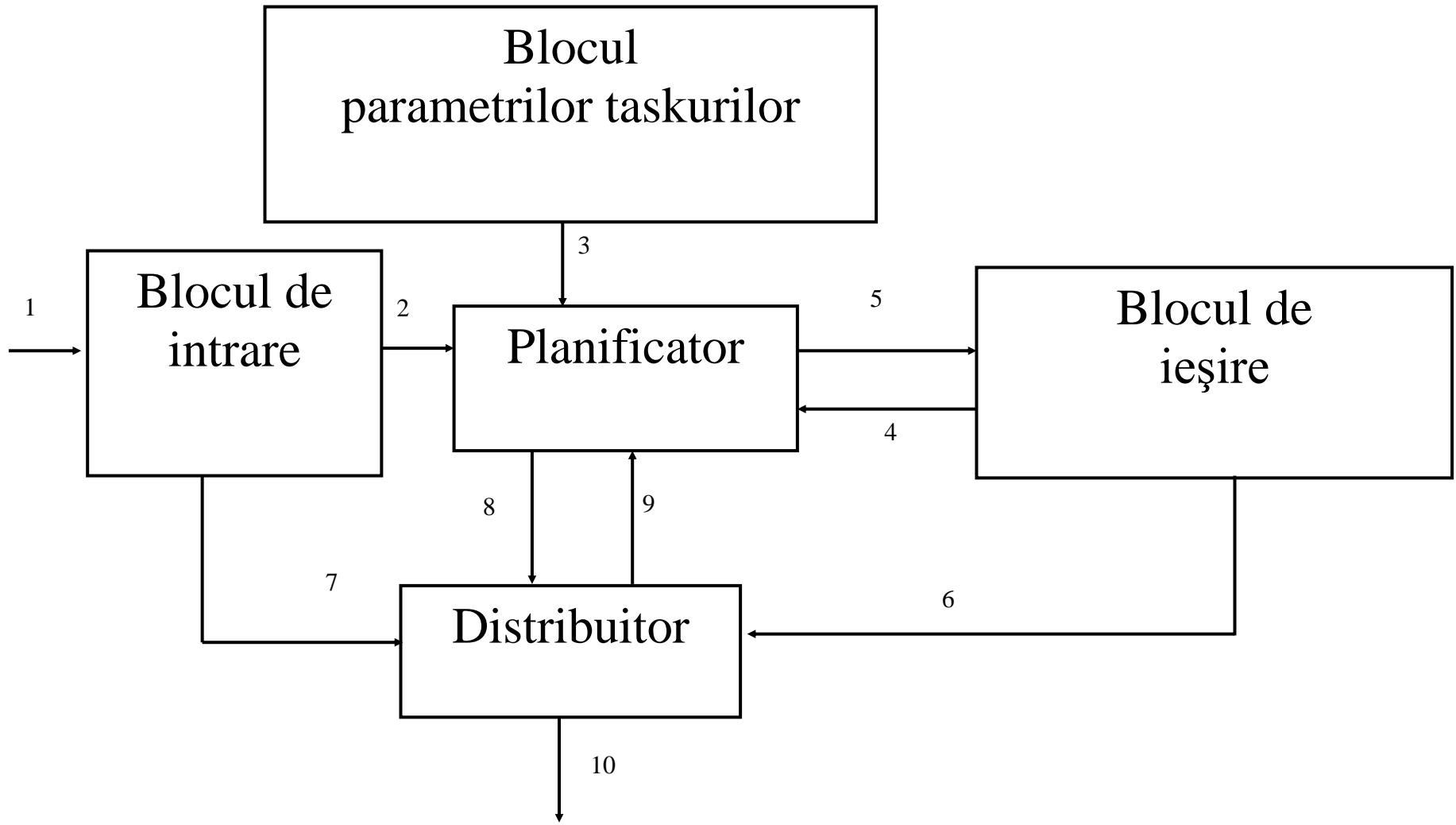
Proiectarea poate fi realizată pe trei niveluri:

- specificația de planificare; - scheduling specification
- regulile de planificare (cum sunt implementate specificațiile); - scheduling policy
- mecanismul (nucleul și interfața hardware împreună cu distribuitorul)
 - kernel mechanism + hardware interface + dispatcher

Un algoritm de planificare performant ar trebui să ia în considerare următoarele:

- existența termenelor de timp-real și tipul lor
- parametrii (în special, duratele de execuție ale) taskurilor sau ale secvențelor de instrucțiuni care compun răspunsul
- momentele în care vor avea loc evenimentele sincrone
- prezicerea momentelor în care se vor întâmpla evenimentele asincrone
- resursele comune disponibile
- duratele de timp rămase și necesare pentru execuția taskurilor sau a secvențelor care sunt implicate de evenimente
- urgența unor răspunsuri
- importanța răspunsurilor
- utilitatea răspunsurilor

La planificarea on-line execuția algoritmului implică o supraîncărcare a procesorului = overhead (este adăugată la timpul de comutare – switching time)



Tipuri de planificatoare:

1) *Implementare off line* – calculele se fac înainte de startarea taskurilor de către:

- utilizator
- programul aplicației

→ Prioritățile sunt fixe → sunt prioritați statice – nu se modifică în timpul execuție aplicației

Utilizează informații disponibile înainte de startarea taskurilor aplicației:

- duratele maxime de execuție
- perioadele taskurilor periodice sau perioadele minime de activare pentru cele sporadice
- deadline-urile
- importanța taskurilor

2) *Implementare on-line* – calculele se fac în timpul execuției sistemului de taskuri de către:

- aplicație sau
- nucleul sistemului de operare

→ Lucrează cu priorități dinamice – se modifică în timpul execuției
Algoritmul on-line utilizează și informații disponibile numai în timpul execuției:

- unele taskuri activate la un moment dat ale căror momente de activare nu sunt cunoscute
- timpul rămas până la deadline-uri
- unele taskuri nu utilizează timpul maxim de execuție alocat – încarcă mai puțin uneori procesorul
- modificarea importanței unor răspunsuri datorită stării curente ale aplicației

Sistemele de operare pot lucra cu priorități statice sau dinamice. La fel și mediile de execuție.

Implementarea în nucleul de timp-real

Implementarea planificatoarelor în nucleul sistemelor de operare are avantajul consumării unui timp mai redus pentru execuția lor.

Dezavantajele implementării planificatoarelor complexe în nucleu sunt:

- Este dificilă construirea în nucleu a unor algoritmi care să ia în considerare un număr mare de parametri.
- Este incomod să se transmită un număr mare de parametri de planificare din programe în nucleul sistemului de operare
- Utilizarea inadecvată a unora dintre algoritmii de planificare complecși poate duce la scăderea performanțelor și implicit a eficienței sistemului
- Planificatoarele complexe necesită pentru execuție intervale de timp relativ mari, care în general nu pot fi estimate.

Implementarea de către utilizatori

Funcția de planificare a proceselor poate fi implementată într-un proces al utilizatorului care se execută cu prioritate maximă.

Avantajele sunt:

- Este mai simplă transmiterea parametrilor între procesele utilizatorului decât de la acestea la nucleu.
- Utilizatorul poate implementa algoritmul care se potrivește cel mai bine aplicației.
- Se pot implementa condiții (precum cele de precedență) pentru ordonare, care ar fi mai dificil de specificat pentru nucleu.

Se poate construi un **server la nivelul utilizatorului** care să activeze taskurile în ordinea dorită.

Taskurile pot avea puncte de preemțiune.

5. Definition of Scheduling Problems

The general scheduling problem is described by:

- Set of n tasks (jobs) $J = \{J_1, J_2, \dots, J_n\}$
- Set of m processors $P = \{P_1, P_2, \dots, P_m\}$
- Set of r resources $\text{Res} = \{\text{Res}_1, \text{Res}_2, \dots, \text{Res}_r\}$
- Precedence relations among tasks are given through directed acyclic graphs.
- Time constraints can be specified for each task.
It is required that all the tasks meet the timing constraints.

The *computing system* can be:

- preemptive
- non-preemptive
- time slicing

Task parameters

For each task J_i :

- a_i denotes (absolute) ***arrival time*** – the time when the event is signaled;
- r_i denotes (absolute) ***release time*** – the time when the task is activated;
- s_i denotes (absolute) ***start time*** – the time when the task starts its execution;
- C_i denotes the worst case ***computation time***
- $C_i(t)$ denotes the worst case (remained) computation time at the time instant t ;
- D_i denotes the relative ***deadline*** (time constraint);
- d_i denotes (absolute) deadline (time constraint);
- c_i denotes the worst case completion (finishing) time;
- T_i denotes the ***period*** (pseudo period for sporadic tasks);
- L_i denotes the ***laxity*** (slack time) at the arrival time;
- $L_i(t)$ denotes the (remained) laxity (slack time) at the time instant t ;
- p_i denotes the ***priority*** (an integer number) meaning the task's importance;

Scheduling means the assigning processors from P and resources from Res to tasks from J in order to complete all tasks under imposed constraints.

Tasks are:

- periodic
- aperiodic
- sporadic (episodic) – minimum specified time distance between consecutive arrival times

The jobs are represented by:

$$J_i = (a_i, C_i, T_i, D_i), i=1,2, \dots, n$$

Another tasks parameter is j_i denoting the jitter = the relative (periodic) **arrival time variation**.

The laxity (or slack time) value at task arrival times is:

$$L_i = D_i - C_i, i=1,2, \dots, n; \text{ or}$$

$$L_i(a_i) = d_i - c_i, i=1,2, \dots, n;$$

Other relations:

$$d_i = a_i + D_i$$

$$c_i \leq d_i \text{ (time constraint)}$$

$$L_i(t) = D_i - C_i(t)$$

Complexity of Scheduling Algorithms – Complexitatea algoritmilor

A **polynomial algorithm** is one whose time complexity grows as a polynomial function p of the input length n of an instance.

Ex.: Sorting by insertion algorithm is $O(n^2)$ (complexity) upper bounded. Let $taskList$ be the list of the n tasks. This has to be sorted taking into account their period (representing the element's value).

The $Sort(taskList, deadline)$ algorithm is:

input: taskList, deadlines

output: readyList

1. Create the list $readyList$, initially empty.
2. Take and remove the *first* element of $taskList$, search its position sequentially in $readyList$ comparing the deadline of *first* with deadline of each element from $readyList$. Insert *first* in $readyList$ according to the increasing of their deadlines.
3. If the $taskList$ is not empty jump to 2; else ***return(readyList)***.

Initially *taskList* has n elements and *readyList* has zero elements. The time duration involves in the worst-case $n \cdot n$ decisions (comparisons). Some closer evaluation can be found.

For some algorithms the evaluation is not so simple. Each algorithm whose complexity is not possible to be evaluated using the previous manner is considered ***exponential time algorithm***.

A particular type is *NP* (***non deterministic polynomial***) class complexity. This contains problems that can be solved in polynomial time on a *nondeterministic* Turing machine.

The ***NP-complete complexity*** characterizes a class of problems that can be solved by polynomial time algorithm if an oracle can be found. For this class of problems if a solution is found, it can be verified in polynomial time that the solution is correct or incorrect.

The ***NP-hard complexity*** characterizes a class of problems. Let P be such a problem. Someone can demonstrate that all the problems from NP-complete can be transformed in polynomial time into P, but cannot demonstrate that P belongs to NP-complete.

Ready queue

The ready queue managed by operating system's kernel can be organized as:

- **list** – a data structure accessed sequentially
- **heap** – a data structure with elements partially ordered (sorted) such that finding either the minimum or the maximum (but not both) of the elements is computationally inexpensive.

Formally a **heap** is a **binary tree** with a key in each node, such that all leaves of the tree are on two adjacent levels.

The duration of searching of an element in a heap is $O(\log n)$ complexity, meanwhile the searching it in a list is $O(n)$ complexity. The reason of organizing the **readyQueue** as a list or a heap is given by the number of tasks. When the number of tasks is large the heap implementation is justified, otherwise the simplicity of list implementation is preferable.

Solving NP-complete problems:

1. **approximation** → search of almost optimal solution instead of the optimal
2. **randomization** → get a faster average running time and allow the algorithm to fail with some small problems
3. **restriction** → restrict the structure of the input (ex. planar graph). Faster algorithms are usually possible.
4. **parameterization** → often there are fast algorithm if certain parameters of the input are fixed.
5. **heuristic** → an algorithm that works “reasonable well” on many cases, but for which there is no proof that it is always fast and always produces a good result.

Classification of Scheduling Algorithms

1. **Preemptive** → the running task can be interrupted, and the processor can be allocated to another one for scheduling reason
2. **Non preemptive** → a task once started is executed by the processor until completion. Scheduling (new processor allocation) decision are taken when a task terminates its execution.
3. **Static** → scheduling decisions are based on fixed parameters.
4. **Dynamic** → scheduling decisions are based on dynamic parameters that can be changed during execution (runtime).
5. **Off-line** → it is executed on the entire tasks set before tasks activation (arrival). The generated schedule is stored in tables and later used by dispatchers.
6. **On-line** → the schedule decisions are taken at (during) runtime every time a new task is activated (arrival times) or the last running complete.
7. **Optimal** → the algorithm minimizes a given cost function defined over the tasks set. For example, there is no another algorithm that fulfill more constraints than the optimal one.

8. **Heuristic** → it searches for a feasible scheduling solution using a heuristic objective function.

9. **Clairvoyant** → it knows the future (in advance, a priori) arrival times of all the tasks.

Cost functions:

Average response time:

$$\text{Response} = \frac{1}{n} \sum_{i=1}^n (c_i - a_i)$$

Total completion time:

$$\text{Completion}_{\text{total}} = \max_i(c_i) - \min_i(a_i)$$

Weighted (w_i coefficients) sum of completion times:

$$\text{Completion}_{\text{weighted}} = \sum_{i=1}^n w_i \cdot c_i$$

Maximum lateness:

$$\text{Lateness}_{\text{max}} = \max_i(c_i - d_i)$$

Maximum number of late tasks:

$$\text{Number}_{\text{late}} = \sum_{i=1}^n \text{miss}(c_i)$$

where

$$\text{miss}(c_i) = 0 \text{ if } c_i \leq d_i$$

$$\text{miss}(c_i) = 1 \text{ otherwise}$$

6. Aperiodic Task Scheduling (Independent tasks!)

Algorithm Earliest Due Date (EDD)

Assumptions:

- uniprocessor system (environment)
- the preemption is not relevant
- set of n independent *aperiodic* tasks (J_1, \dots, J_n) with
- synchronous arrival times; $a_i = 0$ for each task J_i ($i=1, \dots, n$)
- the tasks deadlines d_i ($i=1, 2, \dots, n$) are known
- the worst case computation times C_i ($i=1, 2, \dots, n$) are known

The *EDD algorithm* executes at any time instant the task with the earliest absolute deadline. The algorithm is applied off-line.

It can be demonstrated that EDD algorithm is optimal related to minimizing the maximum lateness.

EDD algorithm analysis

The EDD algorithm complexity is $O(n \cdot (\log n))$.

Real-time requirements:

$$\forall i = 1, \dots, n; c_i \leq d_i;$$

c_i : completion time

Assumption: the tasks J_1, \dots, J_n are listed by increasing deadlines

Worst case completion time:

$$c_i = \sum_{k=1}^i C_k$$

The condition (test) to guarantee that the RT requirements are met:

$$\forall i = 1, \dots, n; c_i = \sum_{k=1}^i C_k \leq d_i$$

C_i : worst case computation time

Algorithm Earliest Deadline First (EDF)

Assumptions:

- uniprocessor system (environment)
- *preemptive environment* (i.e. the tasks are preemptable)
- set of n independent *aperiodic* tasks (J_1, \dots, J_n) with
- arbitrary (asynchronous) arrival times; it is acceptable that $a_i \neq 0$ for some tasks J_i
- the tasks deadlines d_i ($i=1, 2, \dots, n$) are known
- the worst case computation times C_i ($i=1, 2, \dots, n$) are known

The EDF algorithm executes at each time instant the task with the earliest deadline among the ready tasks. The algorithm *is applied on-line*.

It can be demonstrated that *EDF algorithm is optimal* related to minimizing the maximum lateness.

The EDD algorithm is activated (on-line) at each time instant when a new task arrives. It performs the ordering of the current readyQueue according to their absolute deadlines.

EDF algorithm analysis

The EDF algorithm complexity is $O(n)$ if the readyQueue is implemented as a *list* and $O(\log n)$ if it is implemented as a *heap*.

Worst case completion time:

$$c_i = \sum_{k=1}^i C_k(t)$$

The condition (test) to guarantee that the RT requirements are met:

$$\forall i = 1, \dots, n; c_i = \sum_{k=1}^i C_k(t) \leq d_i$$

The test can be performed by the following algorithm:

```
EDF_Test(readyQueue, Jnew) {  
    t=currentTime();  
    readyQueue=insert(readyQueue, Jnew, deadline)  
    c0=0;  
    for(each Ji ∈ readyQueue) {  
        ci = ci-1(t)+Ci;  
        if(ci > di) return(INFEASIBLE)  
    }  
    return(FEASIBLE)  
}
```

The function *insert(queue, element, parameter)* inserts the element into the queue according to the given parameter.

Algorithm EDF in non-preemptive environment

Assumptions:

- uniprocessor system (environment)
- non-preemptive environment (i.e. the tasks are **not** preemptable)
- set of n independent aperiodic tasks (J_1, \dots, J_n) with
- arbitrary arrival times; it is acceptable that $a_i \neq 0$ for some tasks J_i
- the tasks deadlines d_i ($i=1, 2, \dots, n$) are known
- the worst case computation times C_i ($i=1, 2, \dots, n$) are known

The EDF algorithm chooses to execute at program start or when a task is completed the task with the earliest deadline among the ready tasks. The algorithm is applied *on-line*.

EDF algorithm in no-preemptive environment analysis

The problem is appreciated as being NP-hard in this case. EDF algorithm has no *a priori* knowledge about task arrival times. It is not optimal in this case as can be seen from the following example. EDF algorithm schedules J_2 first and then J_1 . This leads to missing the deadline of J_2 , meanwhile the schedule J_1 first and J_2 after fulfills all the deadlines.

If the scheduler benefits from an oracle that leads to the situation where the task arrival times are known *a priori*, the scheduling algorithm can work off-line. The problem can be solved in this case by a branch-and-bound algorithm. Even this case is more complicated than it looks at the first sight.

Parameters	J_1	J_2
a_i	0	1
C_i	4	2
$d_i=D_i$	7	5

Scheduling aperiodic tasks based on priorities in non-preemptive environment

Due to the fact that solving the scheduling problem with EDF algorithm is complicated in a non-preemptive environment, the fixed priority scheduling is proposed.

Assumptions:

- uniprocessor system (environment)
- non-preemptive environment (i.e. the tasks are not preemptable)
- set of n independent **aperiodic** tasks (J_1, \dots, J_n) with
- arbitrary arrival times; it is acceptable that $a_i \neq 0$ for some tasks J_i
- the tasks deadlines d_i ($i=1,2,\dots,n$) are known but not used by scheduler
- the tasks have assigned priorities (given by fixed integer number)
- the worst case computation times C_i ($i=1,2,\dots,n$) are known

The scheduling algorithm chooses the execution of the task with the highest priority among the ready tasks at the start of program or when a task completes its execution. The scheduling algorithm is applied **off-line**. The dispatcher works on-line.

The priority algorithm analysis

It is obvious that the algorithm is not optimal. The question is what are the conditions to be fulfilled by the given set of tasks (in given environment) such that usage of this algorithm for scheduling guarantees that the real-time requirements are met?

In the worst case is when all the n tasks ***arrive at the same time***. That involves all the tasks are executed in the order given by their priorities. The tasks are supposed to be listed in the order given by their priorities.

The condition to meet the task deadlines leads to:

$$\forall i = 1, \dots, n; c_i = \sum_{k=1}^i C_k(t) \leq d_i$$

If not all the tasks can meet their deadlines, the lower priority tasks are those that miss them. ***The task deadlines are used only for real-time requirements verification.***

EDF with Precedence Constraints

Some activities have to be performed before other.

Solution:

Delay the start (release) time of each task according to precedence relations.

$$s_i \geq c_j \quad \text{if} \quad J_j \rightarrow J_i$$

Algorithm: EDF is applied to the readyQueue

7. Periodic Tasks Scheduling

Tipuri de algoritmi de planificare și metode de verificare

Monoprocessor system: (Temă: completați schema următoare)

- independent = non-comunicating
 - fixed priorities
 - $D_i = T_i \rightarrow$ RMS
 - $D_i < T_i \rightarrow ?$
 - dynamic priorities
 - $D_i = T_i \rightarrow ?$
 - $D_i < T_i \rightarrow ?$
- communicating (au resurse comune, schimbă informații între ele)
 - fixed priorities
 - $D_i = T_i \rightarrow ?$
 - $D_i < T_i \rightarrow ?$
 - dynamic priorities
 - $D_i = T_i \rightarrow ?$
 - $D_i < T_i \rightarrow ?$

Scheduling tests:

- 1.based on processor loading (e.g.. $u < 70\%$)
- 2.based on response time ($R < D$)

Analysis

- For hard real-time systems the deadlines must always be met
- Off-line analysis (before the system is started) required to check so that there are no circumstances that could lead to missed deadlines
- A system is *unschedulable* if the scheduler will not find a way to switch between the tasks such that the deadlines are met.
- The analysis is *sufficient* if, when it answers "Yes", all deadlines will be met.
- The analysis is *necessary* if, when it answers "No", there really is a situation where deadlines could be missed.
- The analysis is *exact* if it is both sufficient and necessary.
- A sufficient analysis is an absolute requirement and we like it to be as close to necessary as possible.

Task Scheduling on Mono-processor and Preemptive Systems

**Algoritmi pentru sisteme monoprocesor
Sisteme preemptive**

Independent tasks

Deadlines equal with the periods $D_i = T_i$, $i=1,2, \dots, n$

No-precedence relations.

Rate Monotonic Scheduling (RMS) Algorithm –

Ordonare monotonă după frecvență

Liu și Layland (1973)

The tasks are demanded at the start of each period.

Fixed priorities. The tasks are ordered considering their periods.

Perioadă mai mică → prioritate mai mare.

RMS **is stable!!!**

Test 1. The scheduling is feasible if:

$$C_1/T_1 + C_2/T_2 + \dots + C_n/T_n \leq n(2^{1/n} - 1)$$

Task number	Processor loading
2	82,8%
3	77,9%
4	75,7%
5	74,3%
∞	69%

C_i – the worst case computation time
 T_i – the period
 n – the task number

The algorithm is applied off line → Fixed priorities

The priorities are allocated statically before the starting of the application.

Can be applied to sporadic tasks if $T_i = T_{\min, i}$ ↵ no optimal guaranties.

Test 2. RMS scheduling is feasible if:

(Lehoczky §.a., 1989)

for all the task i , $\min_{0 < t < T_i} U_i(t) \leq 1$

where

$$U_i(t) = \sum_{j=1}^i C_i \left\lceil \frac{t}{T_j} \right\rceil / t$$

$\lceil x \rceil$ - is the smallest integer bigger or equal to x

It is preferable to be calculated in the preemption points:

$$P_i = \{k \cdot T_j \mid 1 \leq j \leq i, k=1, \dots, \lfloor T_i/T_j \rfloor\}$$

where $\lfloor x \rfloor$ - is the biggest integer smaller or equal to x .

The scheduling is feasible if:

for all the task i , $\min_{t \in T_i} U_i(t) \leq 1$

Test 3. The set of n tasks are schedulable with RMS if:
Buttazzo (sufficient condition – not necessary)

$$\prod_{i=1}^n (U_i + 1) \leq 2 \quad \Leftrightarrow \text{hyperbolic bound}$$

where

$$U_i = \frac{C_i}{T_i}$$

Observation:

$T_i = D_i \rightarrow \text{Deadline Monotonic Scheduling} = \text{Rate Monotonic Scheduling if } T = D$

Earliest Deadline First (EDF) Algorithm –

Se execută mai întâi taskul cu deadline –ul cel mai apropiat în momentul respectiv

Independent tasks. EDF *is optimal but not stable.*

The algorithm is applied online.

Test:

$$C_1/T_1 + C_2/T_2 + \dots + C_n/T_n \leq 1$$

- The processor loading has to be bound to 100%.
- Dynamic priorities

Usually it is implemented at the application level (see Realtime Java)

The Deadlines smaller than the periods $D_i < T_i$, $i=1,2, \dots, n$, could be sporadic tasks too

$$T_{\min, i} = T_i$$

Deadline Monotonic Scheduling (DMS) –

Se planifică taskurile monoton după deadline-uri.

off-line applied

Deadline Monotonic Scheduling

The rate monotonic policy is not very good when $D \leq T$.

An infrequent but urgent task would still be given a low priority.

The *deadline monotonic* ordering policy works better.

A task with a short deadline D gets a high priority.

This policy has been proved optimal when $D \leq T$ (if the system is infeasible schedulable with the deadline monotonic ordering, then it is infeasible schedulable with all other orderings).

With $D \leq T$ we can control the jitter in control delay.

The response time calculations from the rate monotonic theory is also applicable on deadline monotonic scheduling.

Interference for the worst case is:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j$$

- exprimă suma duratelor de execuție a taskurilor (j) cu prioritate mai mare decât i mărită cu numărul lor de activări pe durata D_i a taskului i.

Test: The task set is schedulable with DMS = (planificarea după DMS este viabilă = validă = fezabilă) if:

$$\frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1$$

Static Priority Scheduling (SPS) – (Fixed Priority Scheduling = FPS)

The task priorities are given taking into account their criticality.

The algorithm is applied offline.

Requirements: $T_i > D_i$, $i=1,2, \dots, n$

The tasks are ordered using their priorities.

Test 1. Feasibility (Park et al., 1996)

Find the smallest

$$M_i = \sum_{j=1}^i \frac{C_j}{T_j}$$

fulfilling the constraint

$$\sum_{j=1}^i C_j \left\lceil \frac{D_i}{T_j} \right\rceil = D_i$$

The task set is schedulable if for all the tasks i:

$$\sum_{j=1}^n \frac{C_j}{T_j} \leq M_i$$

M_i is the processor loading due to the tasks with higher priorities than task i.

Test 2: Joseph și Pandya (1986)

How can this be calculated?

For all the task i, $R_i \leq D_i$

where

$$R_i = C_i + I_i$$

It is the worst case computation time.

and

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Shortest Job First (SJF) –

Se ordonează taskurile după duratele de execuție (Computation time – C_i , $i=1,2, \dots, n$).

Taskurile cu durate mai scurte sunt executate mai devreme. Se presupune că cele mai critice din punctul de vedere al timpului au durate de execuție mai scurte.

- Static priorities
- Offline application
- Verification: similar to SPS.

Least Laxity First (LLF)

Se ordonează taskurile după timpii liberi până la deadline.

Taskurile cu duratele libere mai scurte sunt executate mai devreme

- Static priorities
- Offline application
- Verification: similar to SPS.

9. Resource Access Protocols

Communicating tasks

The deadlines equal to periods $D_i = T_i$, $i=1,2, \dots, n$ /

The mutual exclusion is required/

Rate Monotonic Scheduling (RMS) –

Se ordonează taskurile după frecvență.

Periodic or sporadic tasks with periodic server

It uses **priority ceiling protocol** (*plafonarea priorității*) (Realtime Java)

Notation: B_i - **blocking factor**

B_i - the worst case longest period of delaying a task i by a lower priority task.

Factorul de blocare este cel mai lung timp cu care poate fi întârziat un task i de către un alt task aflat în execuție cu prioritatea mai mică decât a lui.

Exemplu: Un task cu prioritatea mai mică (sau egală) decât a lui a achiziționat un semafor pe care îl vrea și el.

Test 1

$$\sum_{j=1}^i \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1)$$

B_i – blocking time = timpul de blocare – timpul de execuție pentru cazul cel mai defavorabil al celei mai lungi secțiuni critice al taskurilor cu prioritatea mai mică decât i

Expresia B_i/T_i exprimă cazul cel mai defavorabil al blocării pe perioada T_i a taskului i de către taskurile mai puțin prioritare (sau egale).

Test 2.

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} + \max\left(\frac{B_1}{T_1}, \dots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{1/n} - 1)$$

Earliest Deadline First Scheduling (EDFS) –

Se ordonează taskurile după deadline-uri.

Spre deosebire de cazul taskurilor independente pot apărea blocări datorate taskurilor mai puțin prioritare care achiziționează resurse (sau variabile).

It works with dynamic priorities and kernelized monitor protocol.

Utilizează **protocolul cu monitoare în nucleu**:

Toate taskurile care sunt într-o secțiune critică nu sunt preemtibile. Aceasta poate duce la o utilizare scăzută a procesorului.

Orice task care vrea să intre într-o secțiune critică trebuie să achiziționeze un monitor (aflat în nucleu). Funcțiile de preemtiune se realizează tot prin intermediul nucleului. Dacă monitorul este ocupat, nu se permite preemtiunea.

Test:

$$\sum_{i=1}^n \frac{C_i + B}{T_i} \leq 1$$

If the priority ceiling protocol is used:

Dacă se lucrează cu **protocolul cu plafonarea priorității** testul este:

$$\sum_{i=1}^n \left(\frac{C_i}{T_i} + \frac{B_i}{T_i} \right) \leq 1$$

Protocolul cu plafonarea priorității= Priority Ceiling Protocol

Este o metodă de prevenire a inversării priorității și limitare a timpilor de blocare. El este potrivit pentru reguli de planificare cu priorități statice.

Conform protocolului se extinde moștenirea priorității cu următoarele reguli:

a) se alocă fiecărei variabile partajate o *valoare de plafon* definită ca fiind prioritatea de bază maximă a tuturor taskurilor care o pot utiliza.

b) se permite unui task să rezerve o variabilă numai dacă prioritatea sa activă (currentă) este mai mare decât valoarea de plafon a oricărei variabile care este rezervată în momentul respectiv de celelalte taskuri.

c) prioritatea activă a unui task care zăvorește (rezervă) o variabilă este ridicată la prioritatea cea mai mare a taskurilor care aşteaptă după variabilă.

d) când un task este blocat pe o variabilă, se transmite prioritatea sa taskului care deține variabila în acel moment

În cazul cel mai defavorabil cu acest protocol invocarea unui task este blocată cel mult o dată (conform cu protocolul moștenirii priorității), deși el poate fi blocat chiar dacă nu accesează nici o variabilă partajată.

Se poate starta execuția unui task de prioritate mare chiar și când unele resurse de care el are nevoie sunt rezervate de taskuri cu prioritate mai mică.

Un efect secundar al protocolului este prevenirea blocajelor.

O versiune dinamică a protocolului cu plafonarea priorității este disponibilă pentru utilizarea cu algoritmul EDF unde *valoarea de plafon* pentru fiecare *variabilă* se schimbă cu termenele ferme ale taskurilor activate.

Invocarea unui task poate modifica valorile de plafonare ale variabilelor. Deci se schimbă nivelurile la care pot fi rezervate variabilele în mod dinamic. Algoritmul EDF determină prioritatea taskurilor, deci el determină valorile de plafonare.

O variabilă rezervată de un task nu este rechiziționată, dar un alt task o poate bloca atunci când este eliberată numai dacă are prioritatea cea mai mare, adică are cel mai apropiat termen ferm (pentru EDF).

Deadlines smaller than periods $D_i < T_i$, $i=1,2, \dots, n$ and periodic tasks

Earliest Deadline First Scheduling (EDFS)

– Se ordonează taskurile după deadline-uri.

It uses stack resource protocol.

Controlează rezervarea semafoarelor. Acest protocol se poate utiliza atât pentru priorități dinamice cât și statice.

- Se asociază un nivel de preemtiune fiecărui task.
- Se menține un plafon curent dinamic pentru fiecare variabilă.
- Când este accesată o variabilă (comună) se ridică plafonul variabilei cel puțin la nivelul oricărui task care o poate utiliza.
- Se permite invocarea unui task (activarea lui) numai dacă nivelul său de preemtiune depășește plafonul curent al oricărei variabile comune.

→ Protocolul garantează că orice task poate fi activat numai dacă sunt disponibile toate resursele de care are nevoie.

Test:

$$\sum_{j=1}^i \frac{C_j}{D_j} + \frac{B_i}{D_i} \leq 1$$

Deadlines smaller than periods $D_i \leq T_i$, $i=1,2, \dots, n$ and sporadic tasks

Static Priority Scheduling (SPS)

Se aplică la taskuri periodice și sporadice (setul conține ambele tipuri). Cele sporadice se asimilează cu taskuri periodice.

Test:

For all the task i , have to be fulfilled $R_i \leq D_i$,

where $R_i = C_i + B_i + I_i$

and

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

How can it be calculated?

10. Algorithms for fair executions

First In First Out (FIFO)

- primul intrat în starea gata de execuție, executat mai devreme
Nu merge la aplicații de timp-real!

Round - robin

Algoritmul carusel simplu – round-robin

Equal slices to all the tasks

Se alocă taskurilor cuante de timp:

- egale sau
- proporționale cu prioritățile lor

Avantaj: nu blochează alte taskuri în cazul unor anomalii de programare.

Verificare (RT): vezi exemplul (time sharing) din paragraful 1.

Multi-level queue

Algoritmul carusel multiplu –

Se organizează taskurile în cozi situate pe mai multe niveluri.

Se execută mai întâi taskurile de pe nivelul cel mai de sus. Taskurile sunt executate în ordinea de intrare pe nivelul respectiv (FIFO).

Un task care nu-și termină execuția în **slotul** de timp acordat pe un anumit nivel trece pe nivelul imediat inferior.

Se execută taskurile de pe un anumit nivel numai dacă nu există taskuri pe nivelurile superioare.

Verificare (RT): dificil! Poate fi utilizat modul de mai sus.

Reservation-Based Scheduling

If a task overruns (executes longer than anticipated) this will affect other tasks negatively

- In priority-based systems the priority decides which tasks that will be affected
- In deadline-based systems all tasks will be affected

The goal is to provide temporal protection between tasks that guarantees that a certain task or group of tasks receives a certain amount of the CPU time.

Functional protection provided by the memory management unit in conventional OS (and in some RTOS)

Reservation-based Scheduling: Priority-based system 1

Each task set gets exactly its share of the CPU.

The scheduler can be viewed as consisting of as many ready queues as there are reservation sets.

An external timer is set up to generate interrupts when it is time to switch which ready-queue that is active.

There is one idle process in each ready-queue.

Reservation-based Scheduling: Priority-based system 2

Each task set gets at least its share of the CPU. There is one ready-queue.

The algorithm makes sure that the tasks belonging to the currently serviced task set have higher priority than the tasks in the tasks sets which are not serviced.

An external timer is set up to generate interrupts when it is time to switch between the tasks sets.

Lower the priorities of the tasks that have been serviced and raise the priorities of the tasks that should be serviced.

11. Scheduling on multi-procesor system

It is NP-hard!.

Are un număr foarte mare de variante.

Nu este nevoie de o planificare optimală ci de una care respectă cerințele temporale.

Se urmărește realizarea încărcării echilibrate a procesoarelor.

Se utilizează uneori metode euristice (probabilistice) – dezavantaj: nu se cunoaște durata până când se găsește o **soluție viabilă** (se respectă cerințele temporale).

Uzual se alocă procesoarele taskurilor pe durata unor sloturi de timp.

Se utilizează:

- algoritmi genetici,
- rețele neuronale

Multiprocessor Scheduling

Approach: generalizing the problem of a mono-processor to a multiprocessor system.

The jobs are:

$$J_i = (a_i, C_i, D_i), i=1, 2, \dots, n$$

where

- a_i : denotes the (absolute) arrival (i.e. release) time
- C_i : denotes the worst case computation time
- D_i : denotes the relative deadline

The laxities values at task start times are:

$$L_i = D_i - C_i, i=1, 2, \dots, n; \text{ or}$$

$$L_i(a_i) = d_i - c_i, i=1, 2, \dots, n;$$

where

- d_i denotes the absolute deadline
- c_i denotes the completion time

Scheduling Single-Instance Tasks

Example: Three single-instance tasks and two identical processors.

$$J_1 = (0, 1, 2) \rightarrow L_1 = 1 = d_1$$

$$J_2 = (0, 2, 4) \rightarrow L_2 = 2 = d_2$$

$$J_3 = (0, 4, 5) \rightarrow L_3 = 1 = d_3$$

Scheduling using **Least Laxity First (LLF)** can be seen on Figure 1.

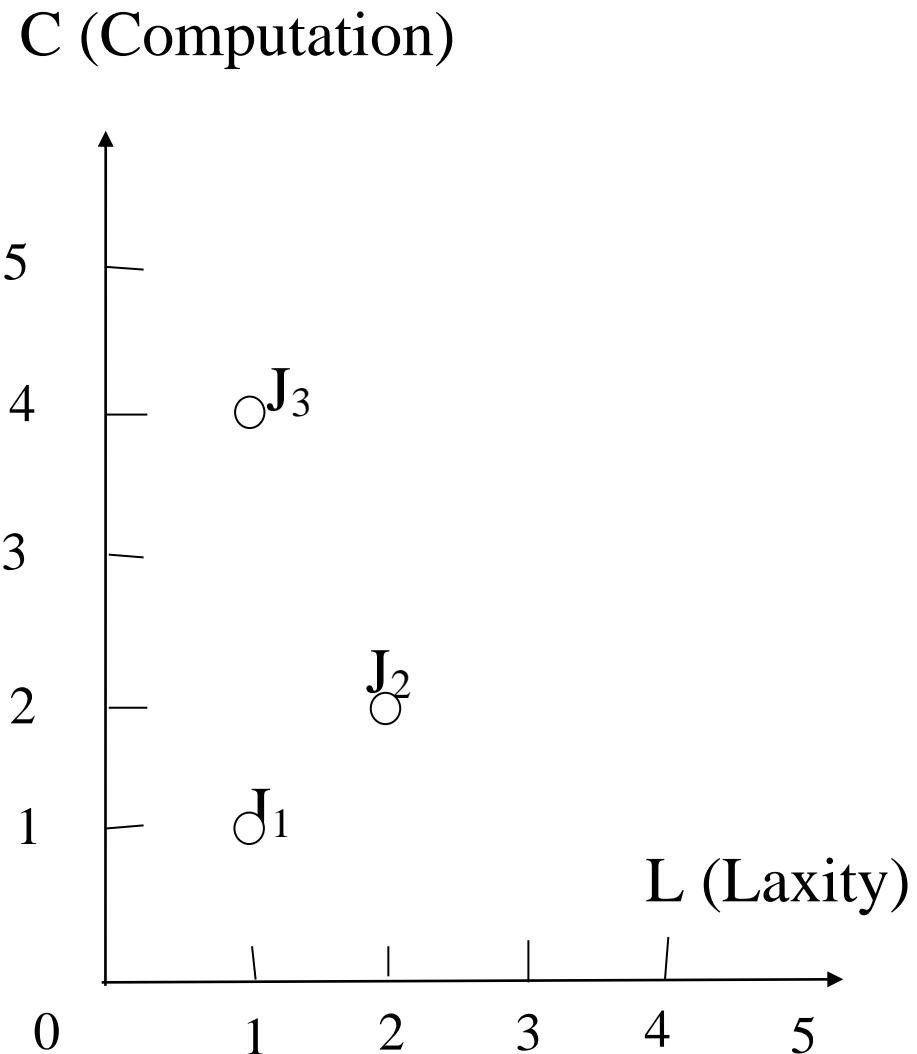


Fig. 2. Scheduling game board

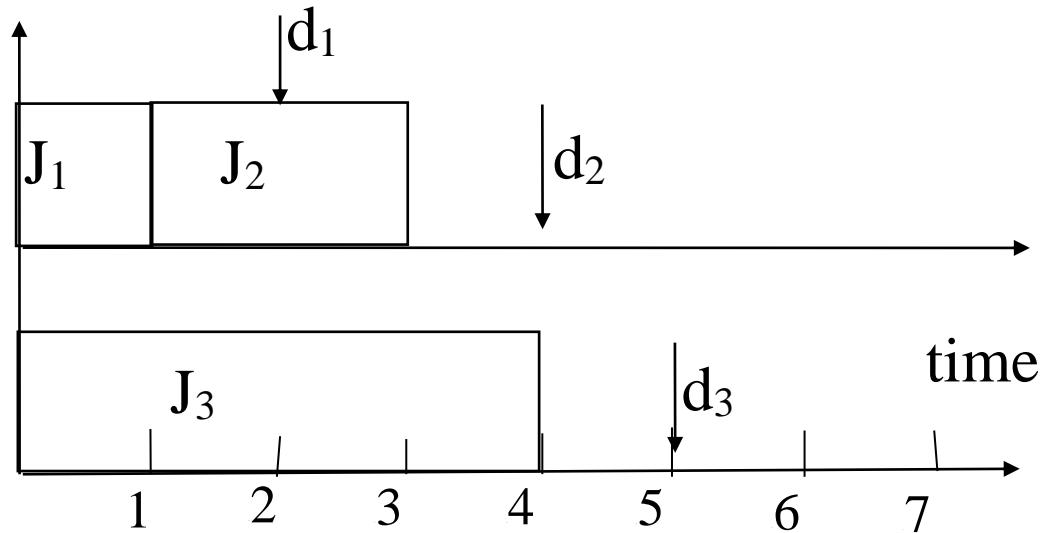


Fig. 1. Gantt diagram of the task's scheduling on a multiprocessor system.

Figure 2 represents *scheduling game board* for the two tasks on the multiprocessor system.

Homework

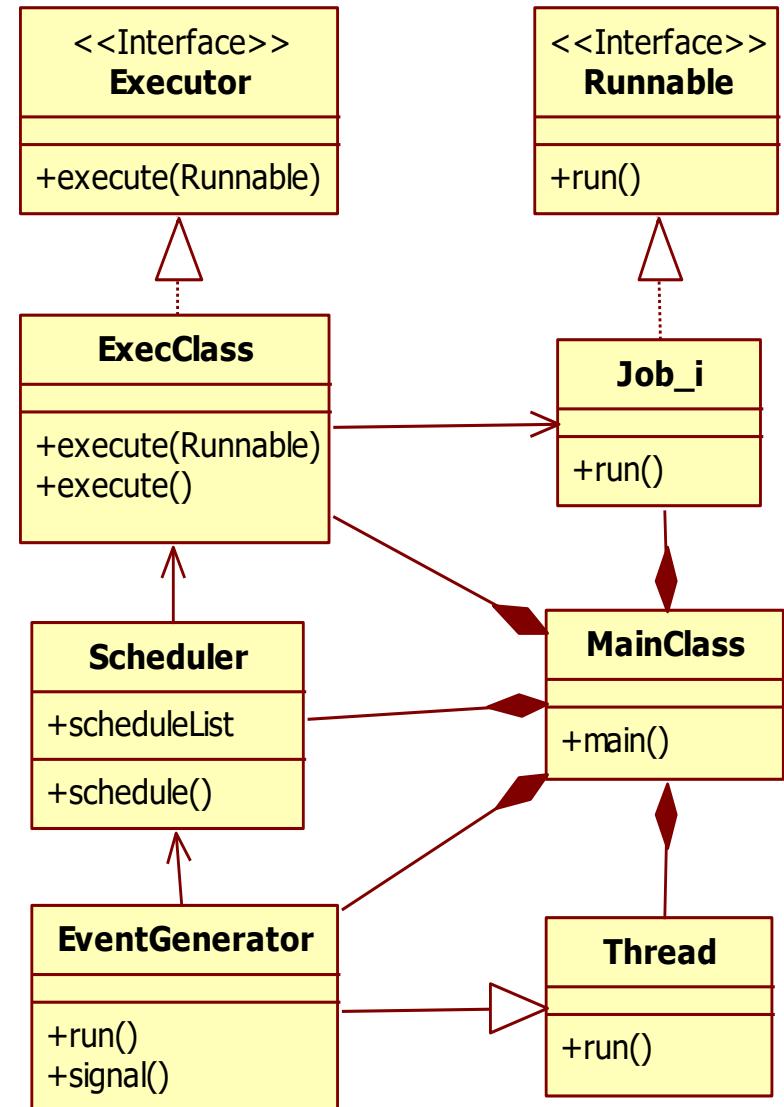
Conceive a LLF scheduler that schedules an application on:

- a two processor system
- non-preemptive system
- J_1, \dots, J_n

Requirements:

- scheduler launch only the jobs that can be executed without interference
- EventGenerator generates the events such that the scheduling is feasible. ($e_i \rightarrow J_i$)
- Calculates the event rates of e_i , $i=1,\dots,4$.
- write the scheduler algorithm

	J_1	J_2	J_3	J_4
C	2	4	6	4
D	6	8	12	14
Event				



```

import java.util.concurrent.*;
public class ExecClass implements
Executor{
    Runnable rr;
    public ExecClass(Runnable r) {
        rr=r;
        execute();
    }
    public void execute(Runnable r) {
        new Thread(r).start();
    }
    public void execute()
    {
        System.out.println("Execute rr");
        execute(rr);
    }
}

```

Suggestion:
classes

```

public class MainClass {
    public static void
        main(String[] args)
    {
        Fir1 f1=new Fir1();
        Fir2 f2=new Fir2();
        ExecClass exec=
            new ExecClass(f1);
        exec.execute(f1);
        exec.execute(f2);
    }
}

```

```

public class Fir1 implements Runnable {
    public void run() {
        System.out.println("Fir1");
    }
}

```

When time elapses the laxity times and the computation time of tasks that are in execution are modified on the base of the following formulas:

$$L_i(t+1) = L_i(t)$$

$$C_i(t+1) = C_i(t) - 1$$

The time parameters of tasks that are not executed are modified on the following manner:

$$L_i(t+1) = L_i(t) - 1$$

$$C_i(t+1) = C_i(t)$$

For the given example:

t=0 \rightarrow C₁=1; L₁=1; C₂=2; L₂=2; C₃=4; L₃=1;
t=1 \rightarrow C₁=0; L₁=1; C₂=2; L₂=2; C₃=3; L₃=1;
t=2 \rightarrow C₂=1; L₂=1; C₃=2; L₃=1;
t=3 \rightarrow C₂=0; L₂=1; C₃=1; L₃=1;
t=4 \rightarrow C₃=0; L₃=1;

Conclusions:

- The set of tasks fulfills the timing constraints.
- During simulation of execution the tasks that are executed approached the horizontal (L) axes, while those that are not executed approached the vertical (C) axes.

LLF Scheduling test of single-instance tasks.

- set of n single instance tasks J_1, J_2, \dots, J_n ; $J_i = (s_i, C_i, D_i)$, $i=1,2, \dots, n$
- m identical processors
- t denotes the time

Usually $m < n$ is considered. The case when $m \geq n$ is trivial.

Test algorithm:

1. Set $t=0$ and construct the list \mathbf{l} of tasks, initial empty.
2. For each task J_i fulfilling $s_i = t$ perform
 - a. $L_i(t) = D_i - C_i$
 - b. $C_i'(t) = C_i$
 - c. add the task to the list \mathbf{l} and order the list.
3. Choose the first m tasks from \mathbf{l} and mark them as being *in execution*.
4. For the tasks from the list \mathbf{l} marked *in execution* calculate
 - a. $L_i(t+1) = L_i(t)$
 - b. $C_i'(t+1) = C_i'(t) - 1$
5. Remove from the list \mathbf{l} the tasks J_i that fulfills $C_i'(t+1) = 0$.
6. For the tasks from the list \mathbf{l} unmarked *in execution* calculate
 - a. $L_i(t+1) = L_i(t) - 1$
 - b. $C_i'(t+1) = C_i'(t)$
7. If there are tasks J_i in \mathbf{l} that fulfill $L_i(t+1) < 0$, signal that **they do not fulfill the timing constraints**.
8. If there is a task J_i such that $C_i'(t+1) > 0$ or unscheduled tasks jump at the step 2; otherwise *STOP*.

Sufficient Conditions for Conflict-Free Task Sets

The previous test is applicable for set of tasks with a priori known arrival times, computation times and deadlines. It cannot be applied to tasks without a priori knowledge of deadlines, computation time and start times. The problem can be solved only if the set of tasks does not have subsets that conflict with each other.

To solve the problem, the scheduling game board is divided into three disjoint regions. Let k be a positive integer denoting a number of time units. The regions are:

$$R_1(k) = \{J_j : D_j \leq k\}$$

$$R_2(k) = \{J_j : L_j \leq k \text{ and } D_j > k\}$$

$$R_3(k) = \{J_j : L_j > k\}$$

The surplus computing function is defined by:

$$F(k, t) = kn - \sum_{R_1} C_j(t) - \sum_{R_2} (k - L_j(t))$$

The function provides the surplus of computing power in terms of available processor time units between given instance t and k units in the future.

The sufficient condition for scheduling a set of tasks, which start at $t=0$, to meet their deadlines is that for all $k>0$, $F(k,0) \geq 0$.

Schedulability test: If a schedule of a set of single-instance tasks, whose start times are the same, exists such that they meet the deadlines, then the same set can be scheduled even if their start times are different and not known a priori. Knowledge of pre-assigned deadlines and computations time are necessary. The least-laxity-first algorithm has to be used.

Scheduling Periodic Tasks

The problem consists of scheduling of a set of n independent, preemptable (at discrete time instance) and periodic tasks J_1, J_2, \dots, J_n on a m multiprocessor system. It is solved by Dertouzous and Mok (1989).

$$J_i = (s_i, C_i, T_i, D_i) , i=1, 2, \dots, n$$

Initially $s_i = 0$ for $i=1, 2, \dots, n$.

Schedulability test:

The necessary feasible schedulable condition is:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq m$$

Let T and t be

$$T = GCD(T_1, T_2, \dots, T_n)$$

$$t = GCD(T, T(C_1/T_1), T(C_2/T_2), \dots, T(C_n/T_n))$$

where GCD is the Greatest Common Divisor.

The sufficient condition that the set of tasks to be feasible schedulable is t is integral. That means each task computation time is a multiple of GCD of their periods.

Ex.: Let

$$J_1 = (0, 2, 4, 4)$$

$$J_2 = (0, 5, 10, 10)$$

$$J_3 = (0, 6, 12, 12)$$

$$J_4 = (0, 8, 16, 16)$$

be a set of independent, preemptable and periodic tasks.

The processor loading is

$$U = \frac{2}{4} + \frac{5}{10} + \frac{6}{12} + \frac{8}{16} = 2$$

The sufficient condition requires that

$$T = \text{GCD}(4, 10, 12, 16) = 2$$

$$t = \text{GCD}(2, 2(2/4), 2(5/10), 2(6/12), 2(8/16)) = 1 \text{ be an integral value.}$$

So, the set of tasks is feasible schedulable.

Scheduling with RMS leads to the following processor assignment:

- Schedule J_1 on the processor P_1
- Schedule J_2 on the processor P_2
- Schedule the tasks J_3 and J_4 during the remained time slots of each processor

This does not lead to a feasible scheduling.

Scheduling with EDF or LLF leads to a feasible scheduling.

Considering acceptable the tasks migration between processors with $U \leq m$ represents a sufficient condition. In this case each task is executed only a T time slices.

*

* * *

** * * * *

* Thank you! *

** * * * *

*

OETPN Scheduling Analysis

1. Independent tasks

Task1:

PNL: $(t1*t2*t3)\#t4$

TIPNL: $(t1[0;\infty]*t2[5;7]*t3[1;2])\#t4[10;10]$

The times assigned to the transitions $t_2, t_3, t_6, t_7, t_{10}$ and t_{11} represent the durations of the activities performed by the task before the transition occurrences.

The times assigned to transitions t_1, t_4, t_5, t_8, t_9 and t_{12} are pure delays (waits).

ETPNL: $(t1(i1,\varphi)*t2(\varphi,\varphi)*t3(\varphi,c1))\#t4(10,\varphi)$

Develop the OETPN model

OETPN \Rightarrow TIPNL

Implement the OETPN model

Verify the temporal behavior

Deadline: $\text{deadln}(t1,t3) = 20;$
 $\text{deadln}(i1,c1) = 25;$

Task2 and Task3 are similar.

What kind of tasks are they? Periodic or episodic?

Change them in **non-blocking** read.

What kind of tasks are they now?

Transform them to execute with constant period.

- a) episodic tasks; mono-processor
- b) periodic tasks; mono-processor
- c) episodic tasks; dual processors
- d) periodic tasks; dual processors

The processing system is:

Mono-processor, non-preemptive, fixed priorities.

Mono-processor, preemptive, dynamic priorities

Dual-processor system, non-preemptive, fixed priorities

Dual-processor system, preemptive, dynamic priorities

For all the tasks verify if the R-T constraints are met.

a) Episodic = aperiodic = sporadic on single-processor system

a.1) Mono-processor, non-preemptive, fixed priorities

Task parameters: Task1(C1, **T1**, D1, pr1) = (7, X, 25); Task2(C2, **T2**, D2, pr2); Task3(....)

Cost calculus: Task1 \rightarrow sporadic; $t2[5;7]*t3[1;2] \Rightarrow C1$ is the *worst case computation time*

$t1[i1,l1] \Rightarrow C1 = l2 + l3 = 7 + 2 = 9$; etc.

EDD: priorities assigned according to their deadlines $\Rightarrow \dots \dots \dots$?

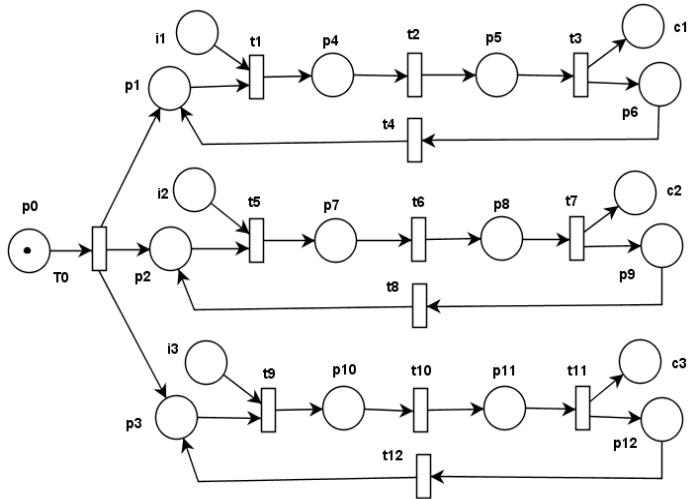
Test: c_i is the completion time

$$\forall i = 1, \dots, n; c_i = \sum_{k=1}^i C_k \leq d_i$$

$c1=\dots?; c2=\dots?; c3=\dots? \Rightarrow$ The deadlines are met?

a.2) Mono-Processor, preemptive, dynamic priorities

EDF:?



Assigned activities:

- Factorial ($n!$)
- Combinations (C_n^k) etc.

Perform measurement tests when no other task are executed on your computer. \Rightarrow get the activities' durations!

Experiment requests:

- Find on your computer the maximum n and k that met the deadlines
- Use a scheduling test for comparison.

Precedence relations

b) Periodic tasks on single-processor system

$C_1 = 9$; $T_1 = 20$?; Constant period \rightarrow non-blocking read!

$C_2 = ?$ $T_2 = ?$ $C_3 = ?$ $T_3 = ?$

- Fixed priorities

RMS \rightarrow processor (loading) Utilization = $C_1/T_1 + \dots + C_3/T_3 = ? \rightarrow$ The deadlines are met

- Dynamic priorities

EDF \rightarrow Utilization =

DMS: Interferences $I_1 = ?$

$$D_1 = ? \quad D_2 = ? \text{ etc.} \quad I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j$$

$$\text{Utilization} \quad \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1$$

SPS:

For all the task i , $R_i \leq D_i$

$$\text{Where } R_i = C_i + I_i \quad I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$I_1 = 0$; $\rightarrow R_1 = C_1$; $\rightarrow I_1 = \dots ?$

c) Episodic tasks on dual-processor system

- Non-preemptive system

$L_1 = D_1 - C_1 = 25 - 9 = 16$; $L_2 = ?$; $L_3 = ?$

LLF: least laxity first $\rightarrow pr_1 > pr_2 > pr_3 ?$

Response times: $R_1 = ?$; $R_2 = ?$; $R_3 = ?$

Possible combinations: Task1*Task3 & Task2; Task2*Task3&Task1;

\rightarrow Deadline verifications: $C_1 + C_2 \leq \text{deadln(Task2)} = D_2$; $C_2 + C_3 \leq \text{deadln(Task3)} = D_3$

d) Periodic tasks on dual processor-system

LLF: $L_1 = ?$; $L_2 = ?$; $L_3 = ?$

- Possible combinations: idem above;

2. Communicating tasks

Mutual exclusion: mutex{t7*t8, t12*t13}

mutex{t1*t2, t7*t8, t12*t13}

- Precedence relations examples:

t3->t7; t8->t12;

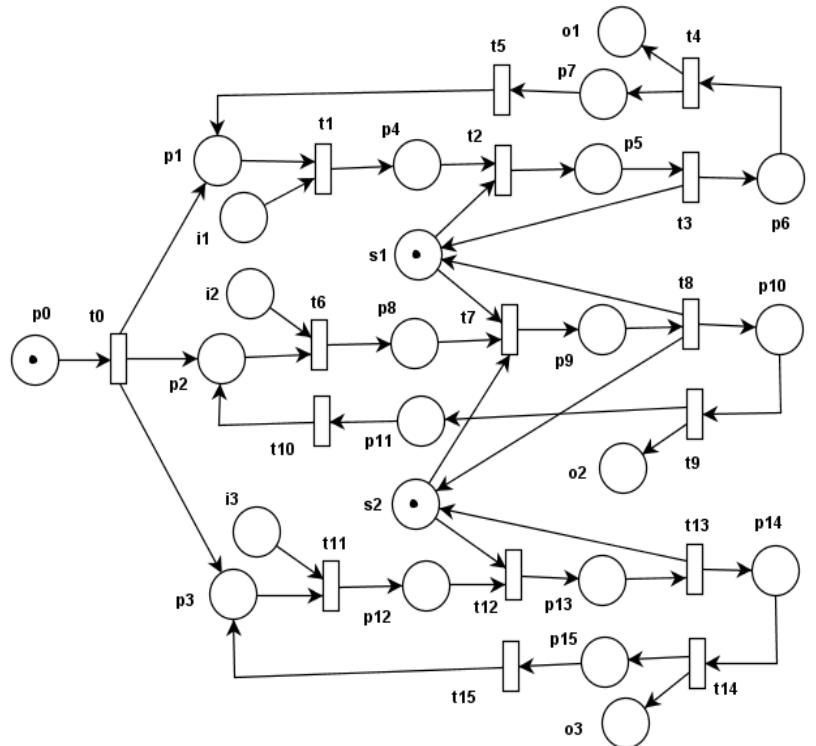
a) Deadlines equal periods

- Single processor

RMS:

- Periodic server \rightarrow grant the processor according to the order of their rates
- it has to use the priority ceiling protocol
- \rightarrow fixed priorities

$$\sum_{j=1}^i \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1)$$



or

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} + \max\left(\frac{B_1}{T_1}, \dots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{1/n} - 1)$$

T1, T2, T3 \rightarrow pr(Tasks1); pr(Task2); pr(Task3)

ti[ei, li]; i=1, 2, ...

Notations: duration of ti[ei, li] is max{ei, li} = li

C1 = l1 + l2 + l3 + l4

Mutex{ }

B_i is the blocking time: B₁ = l7 + l8; B₂ = l12 + l13

\rightarrow

b) Deadlines smaller than periods

- single processor
- fixed priorities

For all the task i, have to be fulfilled R_i ≤ D_i,

where R_i = C_i + B_i + I_i

and

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

c) Dual-processor

Possible combinations: Task1*Task2 & Tasks3; etc.

LLF: C1 + C2

$\leq_{\text{deadln}} (\text{Task2}) = D_2; C_2 + C_3 \leq_{\text{deadln}} (\text{Task3}) = D_3$