Behavioral modeling with Petri Nets for Verification

Fabrice Kordon & Yann Thierry-Mieg

LIP6

Context of this work

- The present courseware has been elabor[…] the MODELPLEX European IST FP6 project (http://www.modelplex.org/).
- Co-funded by the European Commission, the MODELPLEX project involves 21 partners from 8 different countries.
- MODELPLEX aims at defining and developing a coherent infrastructure specifically for the application of MDE to the development and subsequent management of complex systems within a variety of industrial domains.
- To achieve the goal of large-scale adoption of MDE, MODELPLEX promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDE courseware provided here with the status of open-source software is produced under the EPL 1.0 license.

Outline

- Problems in software development
- Some consideration about distributed systems
- A first approach on behavioral modeling
- Introduction to Petri Nets
- Some formal definitions on Petri Nets
- Some properties of Petri Nets
- Component-based methodology for behavioral modeling
- An industrial example (verified middleware)
- Some conclusions & perspectives

- Fabrice.Kordon@lip6.fr
- LIP6, Université P. & M. Curie,
- Paris, France

- Companion-site : http://fabrice.kordon.name/ufsm

Objectives of the course

- Distributed computing is increasing
- Are we able to cope with increasing complexity of such systems?
- We need to specify systems more precisely
- From «boxes» to behavioral specification
- Behavioral modeling is important
- Simulation and testing are reaching limits
- There is a need for formal modeling

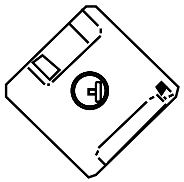
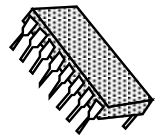
Contents of the course

- Problems in software development
- Some consideration about distributed systems
- A first approach on behavioral modeling
- Introduction to Petri Nets
- Some formal definitions on Petri Nets
- Some properties of Petri Nets
- The modeling operation (methodological considerations)
- Training
  - Use of a Petri Net environment: CPN-AMI
  - Three stages
    - play with one example model
    - model a simple system
    - model a more complex system
- Concluding remarks

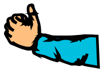
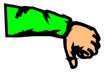
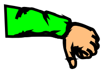
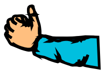
Problems in software development
(especially for distributed systems)

Hardware versus software

- "Hardware is, Software will"

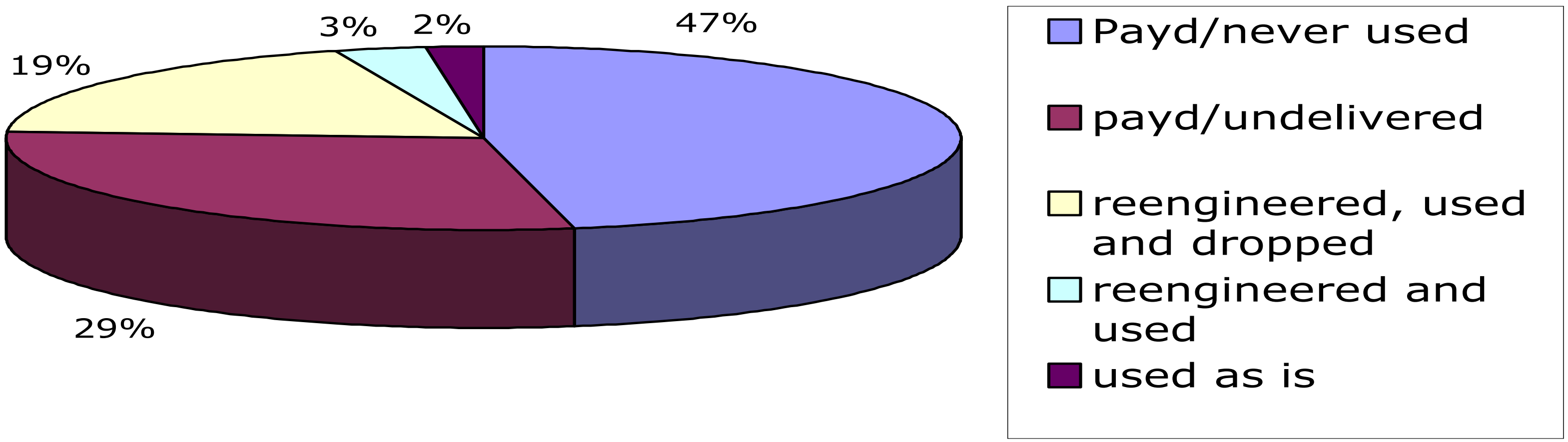
- What is different between soft and hard?

	Hardware	Software
Faster Higher abstraction level		
Rigid		
	?	?

- Both may be unreliable
  - Hardware: you die
  - Software: you sell maintenance

Is software risky? (1)

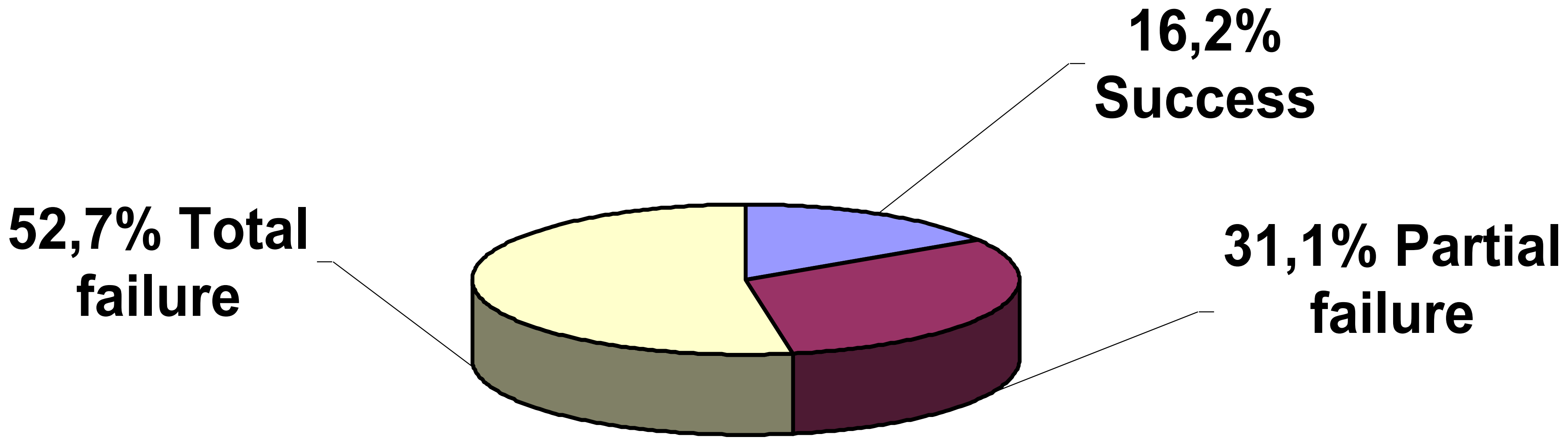
- Government Accounting Office (1979)
  - 9 projects
  - $7 000 000



Is software risky ? (2)

10

- Analysis on various project results in 1995 (The Standish Group)



Why is software risky?

Observations

- No standard (or a very few)
- Maintenance/evolution problems
- Very limited reuse

- Almost no method

The difference S/H can be explained

Why hardware is better

High production costs

Thus, a need for big series

No way to correct a bugged chip

Hardware people have to be prudent

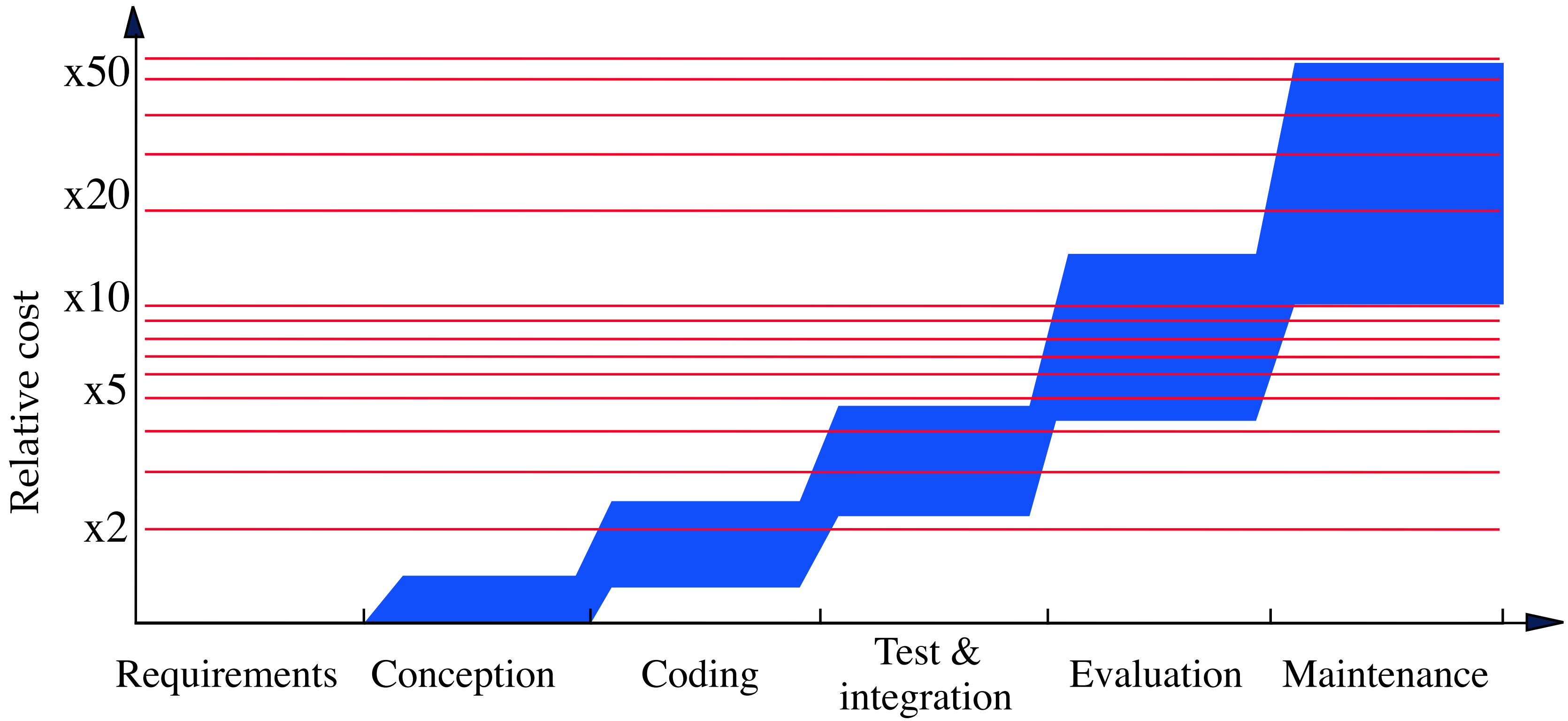
What is software

- A real product
- A "flexible" product
- Software production is not a «fully recognized» engineering discipline (such as for building bridges or buildings)

- There is no standard way to produce software
  - Can it be standardized since it is «brain juice»?
- Most project lead to an «original product»
- Like an œuvre d'art

Source: I.Sommerville ©1995

Observation 1: Correcting or introducing changes, compared costs

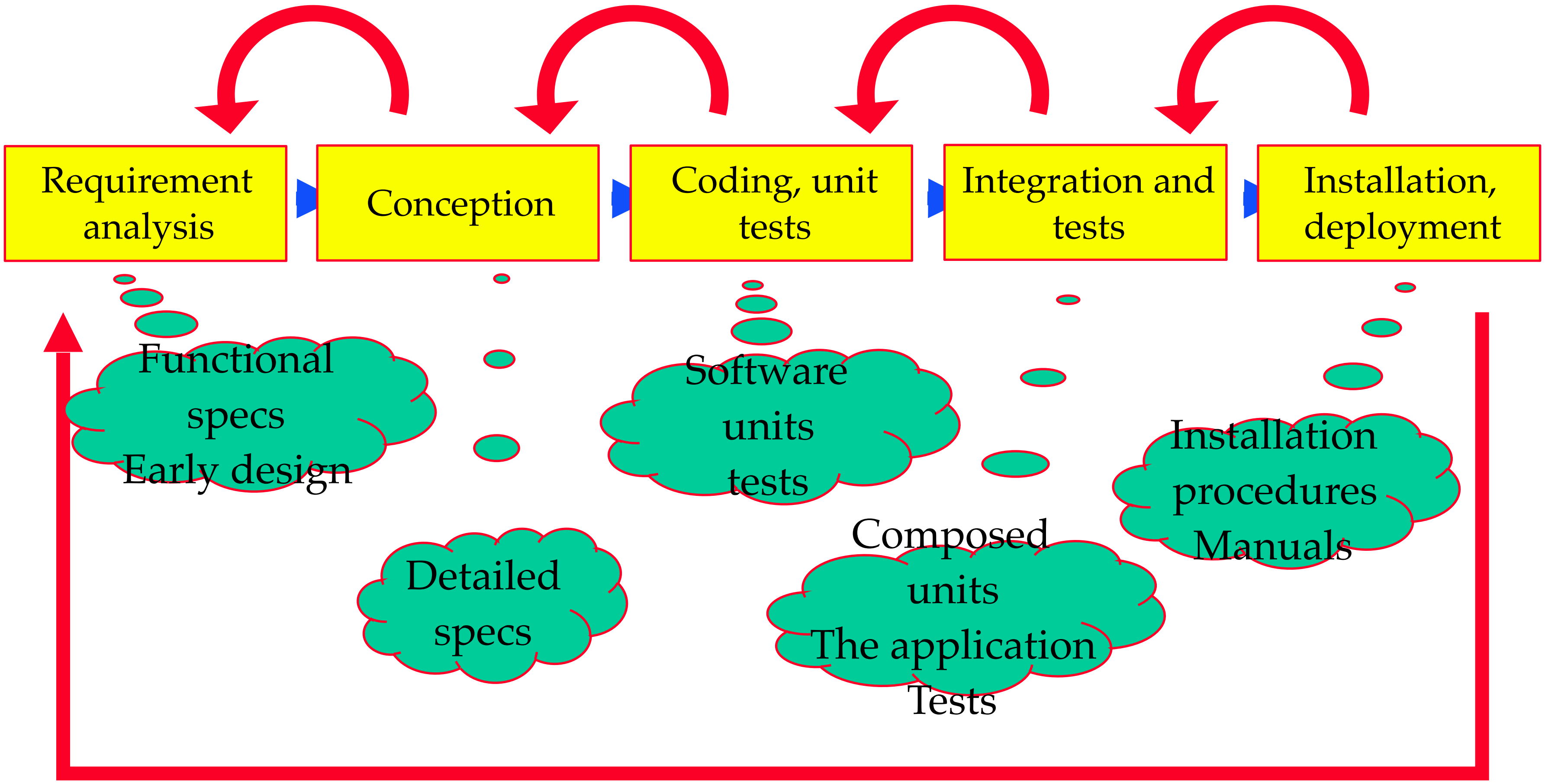


Characteristics of maintenance/evolution

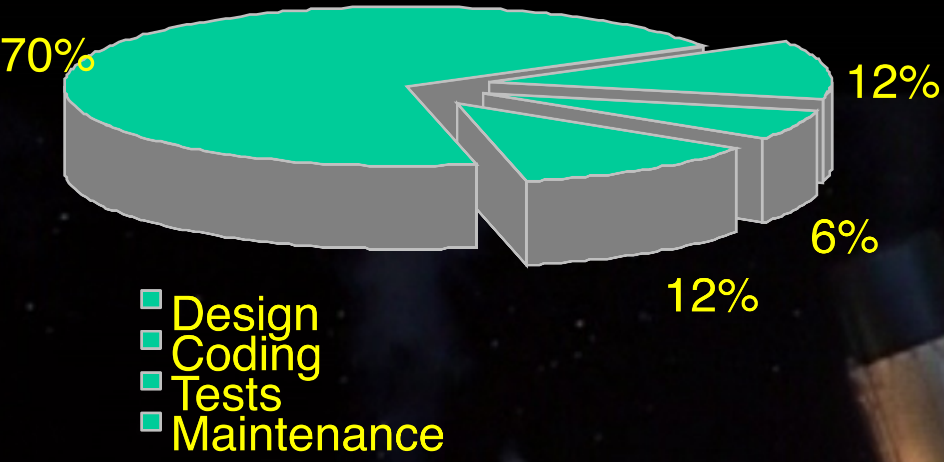
We observe

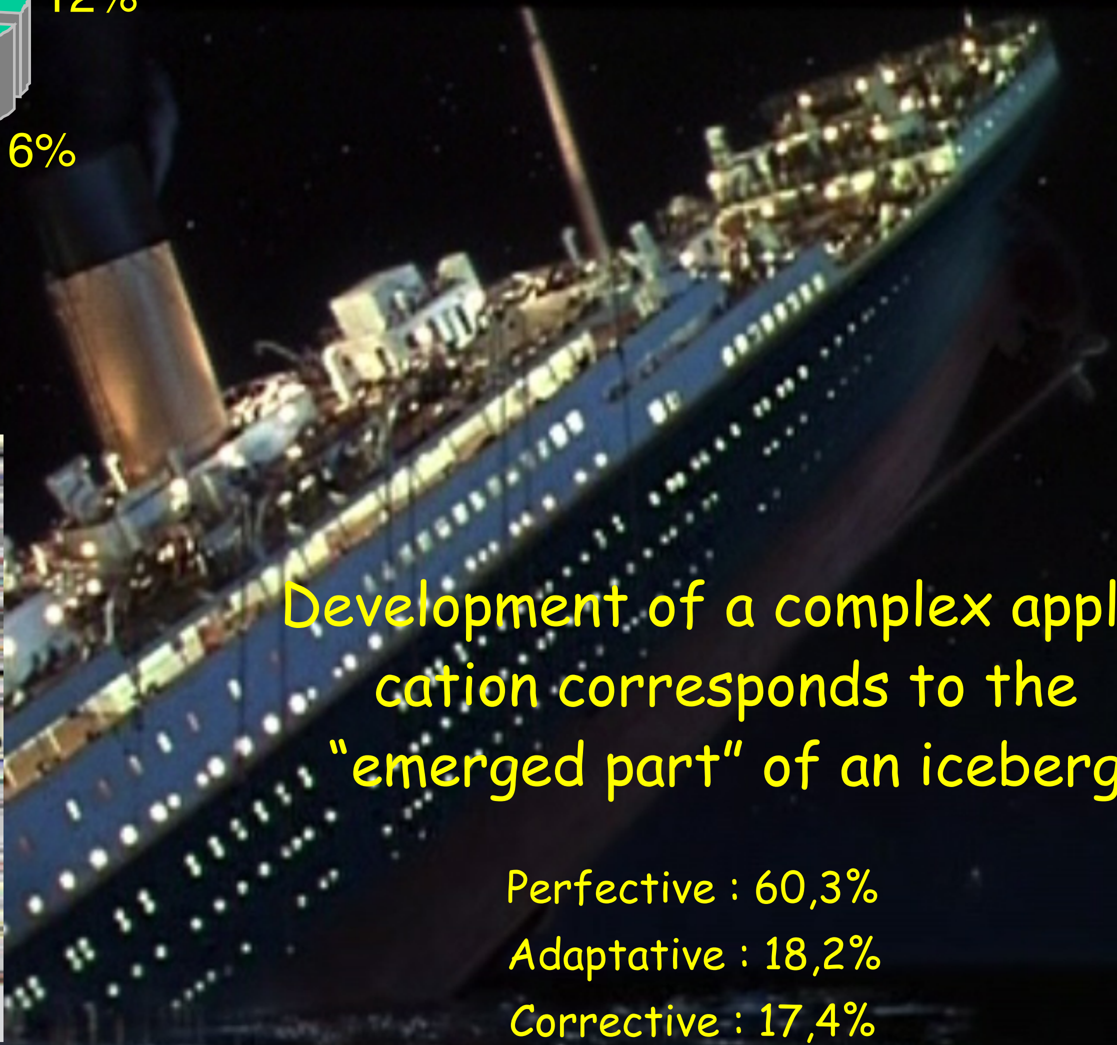
- Slow correction process
  - Collect reports
  - Analyze reports
  - Fixing/changing stuff
  - Installing a new version...
- Reduced teams
  - There is no way to maintain large teams when the product is in production
- Less and less safety when delivery gets far
  - Possible side effects of a fix/evolution... essentially for large software
  - It may be difficult to reconsider some choices
  - Limited memory from the design.

Intuitive vision of the «software life cycle»



Observation 2 : DIstribution cost for an application





Development of a complex appli-cation corresponds to the "emerged part" of an iceberg

Perfective : 60,3%

Adaptative : 18,2%

Corrective : 17,4%

others : 4,1%

What about model driven engineering?

- Development and Maintenance of industrial applications
  - Are more and more complex,
  - Technologies change rapidly,
  - New «social factors» (users) in such systems,
  - Can be sold in «temporal frames» that can be small.

«Software Chronic Crisis» (Gibbs, Scientific American)

$ 100.000.000.000 in 1996 (Source, Standish Group International)

Model driven engineering (prototyping)

IEEE : «A type of prototyping in which emphasis is placed on developping prototypes early in the development process to permit early feedback and analysis in support of the development process»

When systems are distributed,
this is even more complex!!!!!!

Traditional testing is inappropriate!

Some consideration about distributed systems

Lehman's Laws

- Continuing change
  - A program that is used in a real-world environment must change, or become progressively less useful in that environment.
- Increasing complexity
  - As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.
- Lehman and Belady, 1985

What's wrong with OOP?

1. OOA and OOD are domain driven
   - Designs are based on domain objects, not available components
   - Objects end up with rich interfaces, not plug
   - CONCLUSION: Hard to reconfigure and adapt objects
2. Implicit Architecture
   - Source code exposes class hierarchy, not run-time architecture!
   - Objects are wired, not plugged together
   - How the objects are wired is distributed amongst the objects
   - CONCLUSION: Hard to understand and hard to evolve
3. Implicit Reuse Contracts
   - Idioms and patterns are hidden in the code
   - CONCLUSION: Steep learning curve for development and evolution

What about Components?

stable

A software component is a unit of independent deployment without state

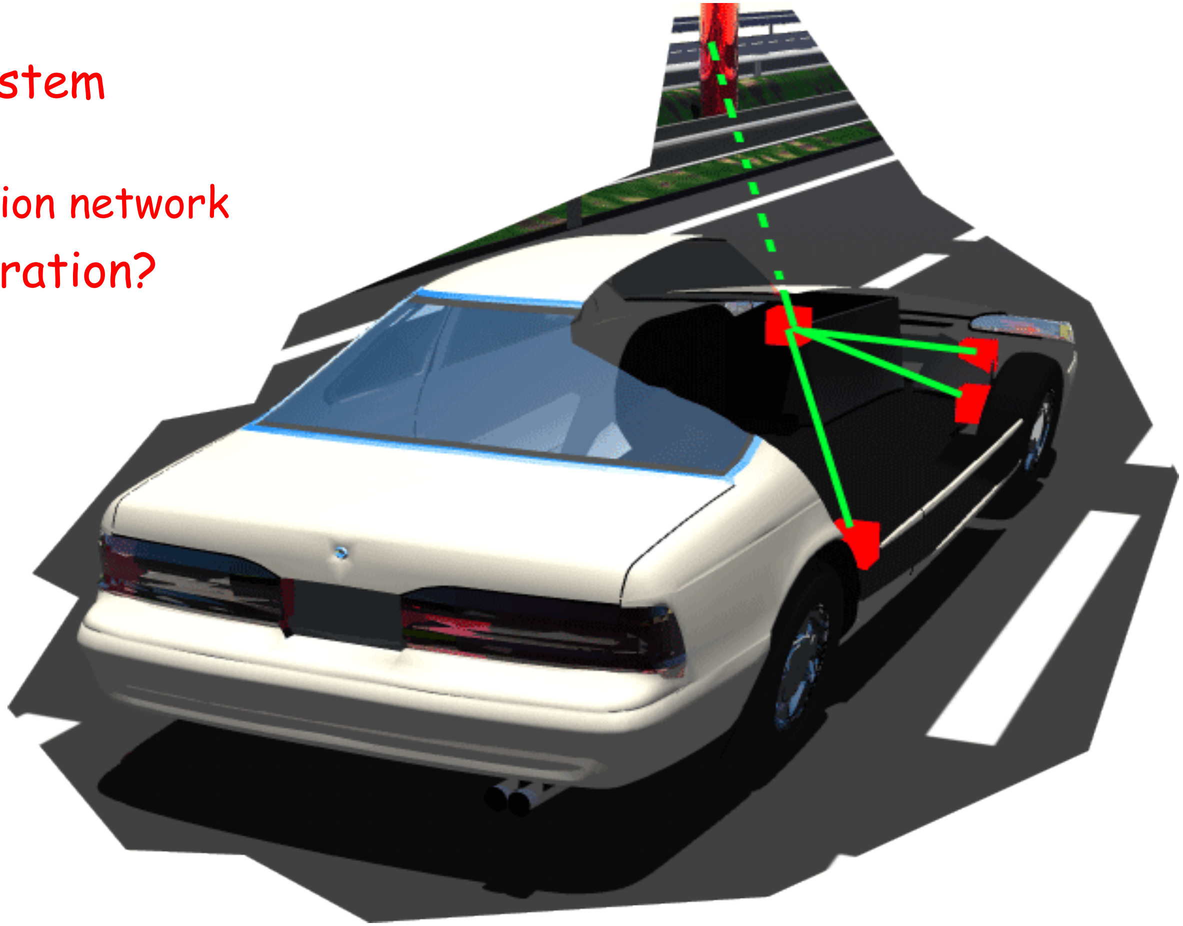
- We know how to build components!
- We don't understand how to compose flexible applications from components.
- We should be thinking more about composition than about components

What future for distributed systems?

- Evolution of Distributed Systems is incredibly fast
- We are just at the beginning of their existence
- Todays solutions do not support «tomorrow's needs»
  - Scaling up
  - P2P approach
  - Hight reliability
- Problems with appropriate infrastructures?
- Needs for a «new paradigm»?
  - We wait for about 27 years since OO-languages

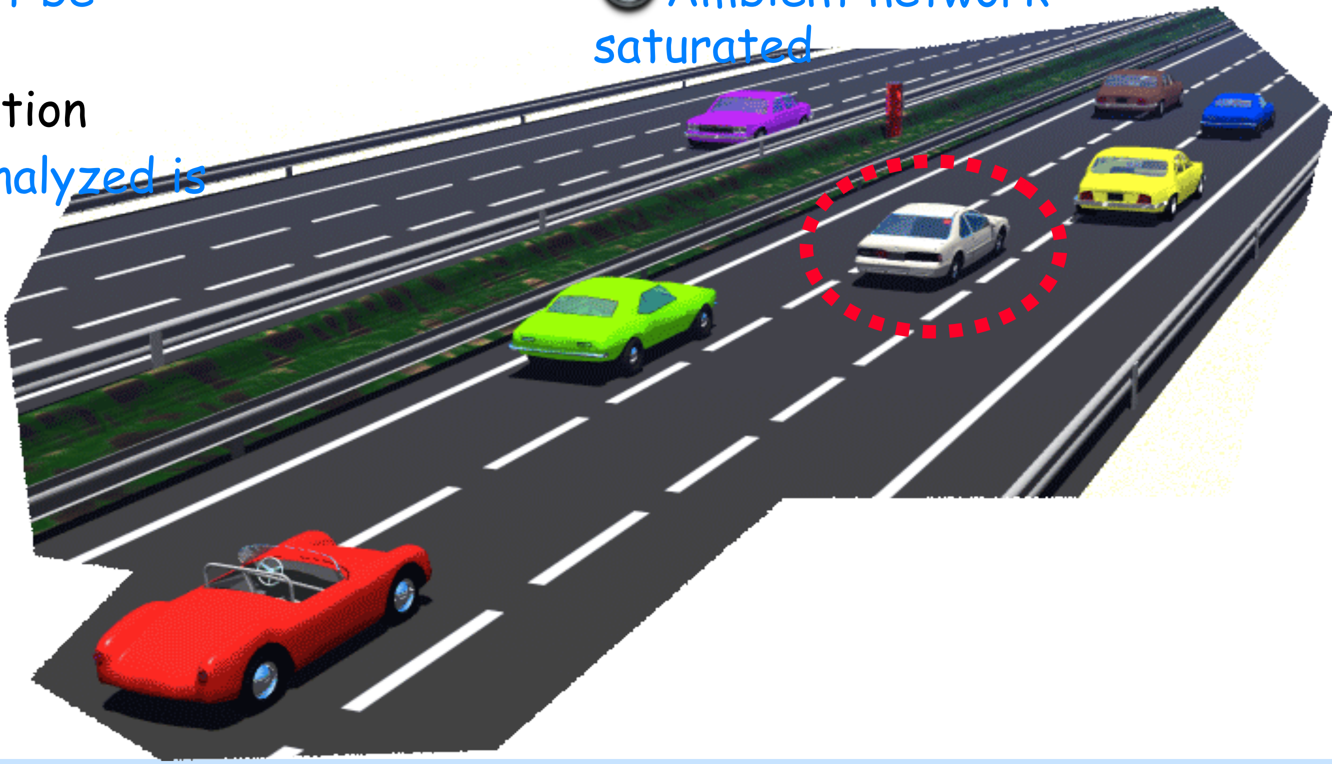
Example of future applications: automatic highway (1)

- A car = distributed system
  - Many processors
  - Specific interconnection network
- How to handle configuration?
  - Task affectation
  - Redundancy



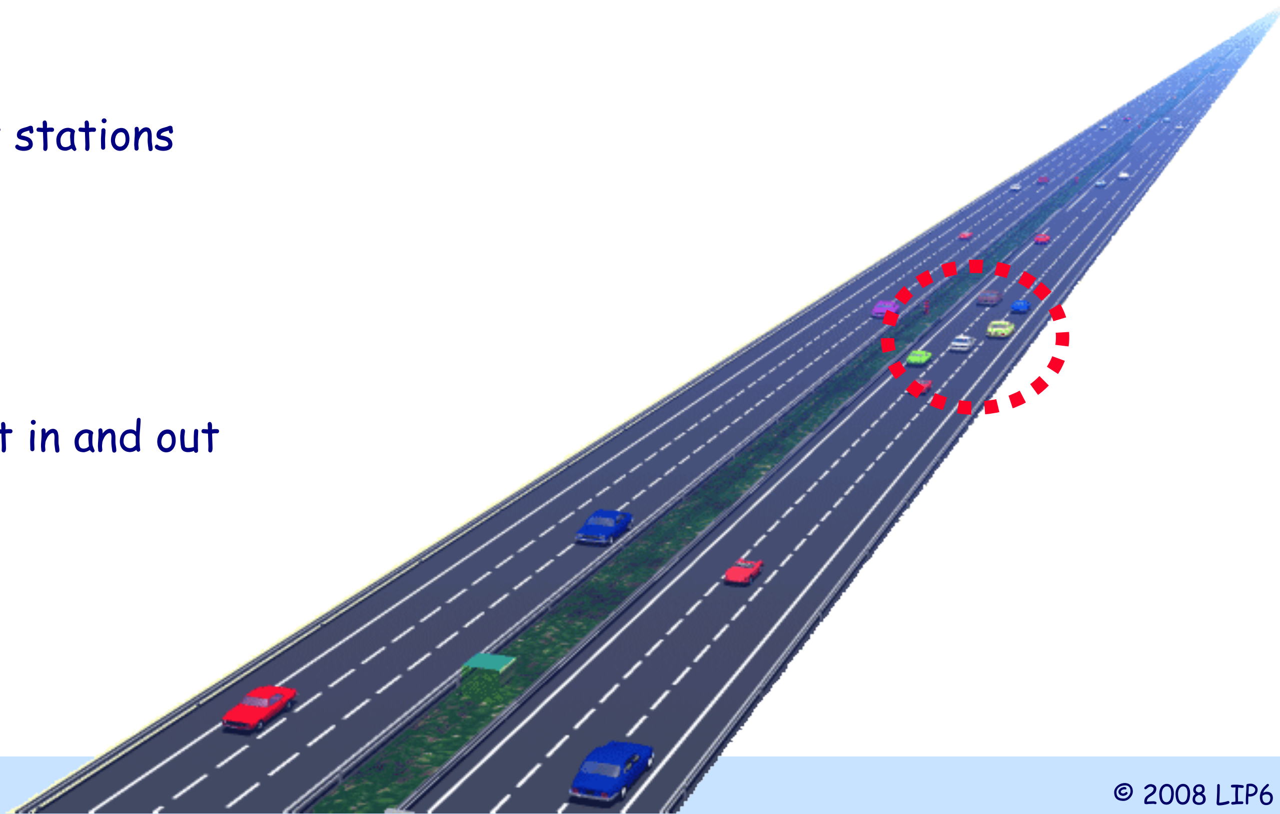
Example of future applications : automatic highway (2)

- Reliability of interactions
  - Modeling problem (p2p)
  - Analysis using formal methods
    - System must be deterministic
  - Program generation
    - What you analyzed is what you get
- Fault tolerance problems
  - Unreachable cars = ???
    - Car out
    - Car away for a while
    - Ambient network saturated



Example of future applications : automatic highway (3)

- Large scale system
  - Lots of actors
  - Length of the system
- Complex interoperability (p2p)
  - Car / car
  - Car / captors
  - Captors / management stations
- Dynamic adaptation
  - Management policies
  - Handling of events
  - Traffic control
  - Configuration: cars get in and out



Needs for distributed systems

Two «classes» of customers (and needs)

- Level 1:
  increase speed of development, integrate a know-how in tools (need for productivity)
  - Telecom, home applications, …
- Level 2:
  Increase the reliability of systems by using formal verification techniques
  - «Mission-critical» and/or «high-confidence» systems

- In both cases, there is a need for help in developing such systems
  - Modeling, verification, model transformation, etc.

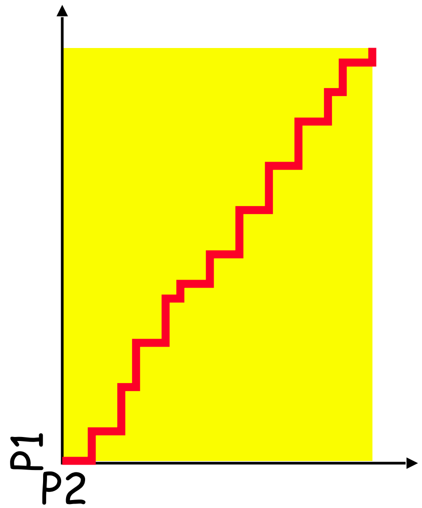
Why formal modeling behaviors of distributed systems?

- Because they are complex to capture
- Because we need to perform «automatic reasoning»
  - Detect bad behavior,
  - Ensure that some properties are preserved,
  - etc.
- Modeling at a behavioral level is CRITICAL for distributed systems
  - Especially when they become complex
  - Some studies of proposed solutions must be performed prior to implementation

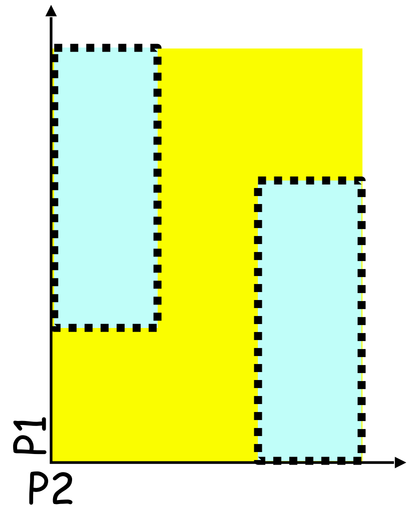
A first approach on behavioral modeling

Example of needs

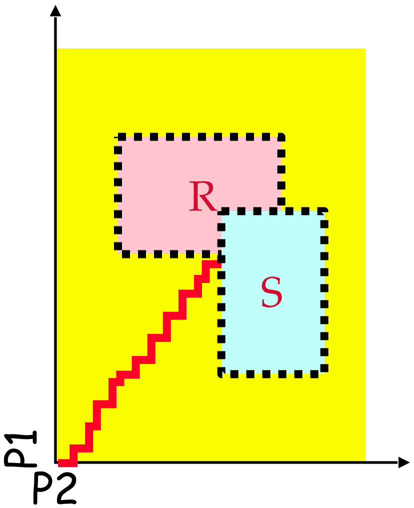
- Example: behavioral analysis
- Let us represent the execution of two processes...



No relationship



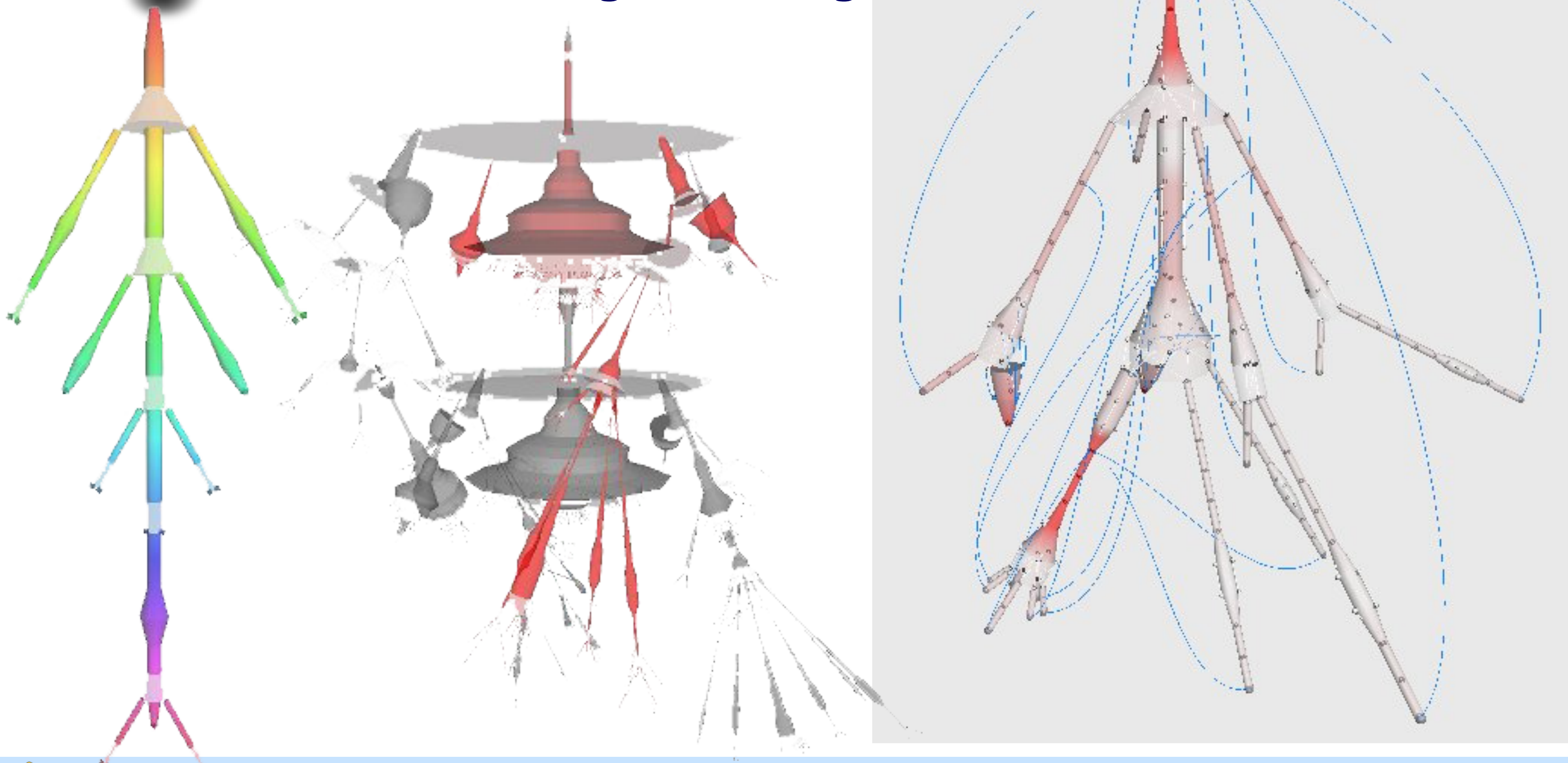
Proc1->Proc2



Proc1->S,R
Proc2->R,S

State space for a N-processes system...

- Each process = one dimension for execution
- Be aware of original things (dead-ends, etc.)



source : http://www.win.tue.nl/~fvha

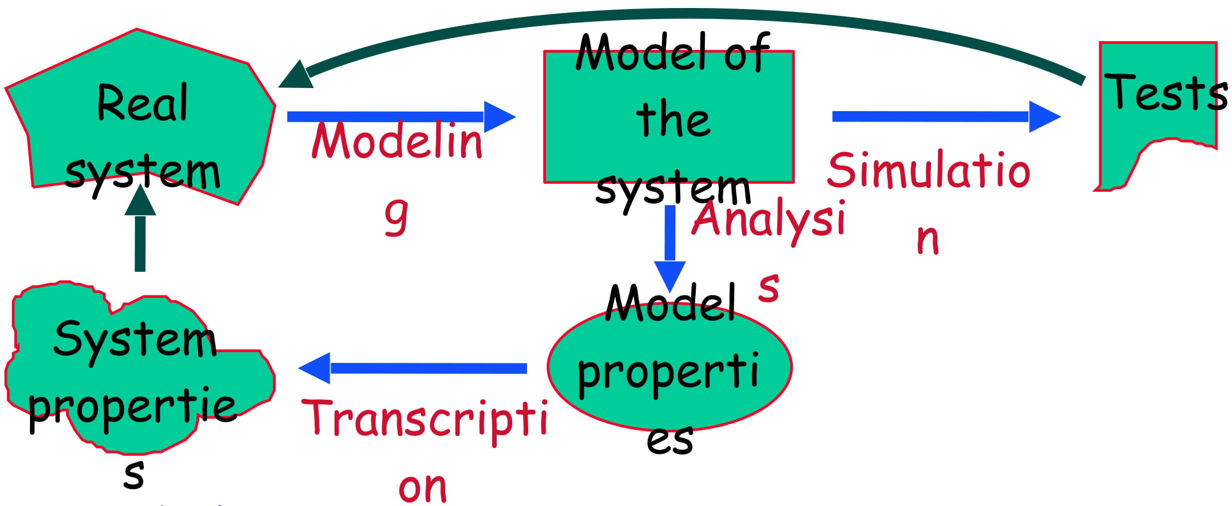
So, why modeling?

- To study the complexity of applications (here, due to the parallelism)
  - Communication
    - Between hosts
    - Between processes or threads
  - Concurrent access to resources
  - Synchronization
    - Rendez-vous,
    - Critical sections
    - Dedicated protocols
- There are other interesting domains for such an analysis
  - Real-time
  - Embedded
  - Hybrid...

All these domains complexity generates very complex problems (combinatorial explosion)

Modeling

- Objectives



Expected characteristics

- Easy modeling process
- Theoretical foundation

Easy expression of properties
CASE tools

Three types of notations
Natural language, Rigorous, formals

Natural language (or any informal ones)

"Natural"

Strutured text, graphics…

Might be "standardized"

Flow diagrams,

Textual algorithms…

m

- Nice and easy to define but…

- Ambiguous (multiples interpretations)
- Incomplete (partial specification)
- Inconsistent
- Various level of description
- Contradictory

Rigorous languages

Conceptual foundations

Propose a set of precise concepts

Syntactically defined

A grammar is proposed

Limited interpretation

Should prevent from any ambiguous interpretation

They support

m

- Execution (suitable description)
- Simple inconsistencies detection
- May support program generation

A few examples

m

SA-DT, SA-RT

HOOD, OMT, OOA

UML

Formal languages

Mathematical foundations

unambigious

Formal description of interactions

Support for formal verification

They support

m

- Execution
- Evaluation of the specification validity
- Detection of unconsistenties
- Verification of properties
- Program generation

A few examples

m

Z, B, VDM, Algebraic specifications,
State automata, Promela
Petri nets...

Theorem proving
Model checking based
Structural analysis

Introduction to Petri Nets

Formal methods: classification

Two types of formal methods

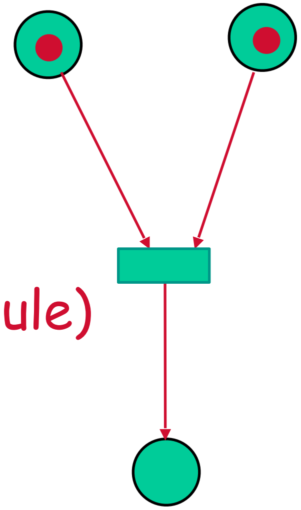
- Algebraic based
  - The system is described by means of axioms
  - The property to be demonstrated is a theorem
  - Demonstration can be helped by a «theorem prover»
  - Characteristics
    - supports infinite systems, parametric approach, difficult to automate
- state-exploration based
  - Behavior of the system is described by means of a formal language
  - The property to be demonstrated is a formula (invariant, causal)
  - Demonstration is performed by building the state space of the system
  - Characteristics
    - supports finite systems only, non parametric approach, easy to automate, counter-example provided automatically

Petri Nets

- Petri Nets approach is closer to model checking
  - State space generator...
  - ... but properties can be deduced from its structure
- Families of Petri Nets
  - Place/Transition
  - Colored
  - Stochastic
  - Timed
  - Algebraic...
- We will focus on «simple» Petri Nets: P/T

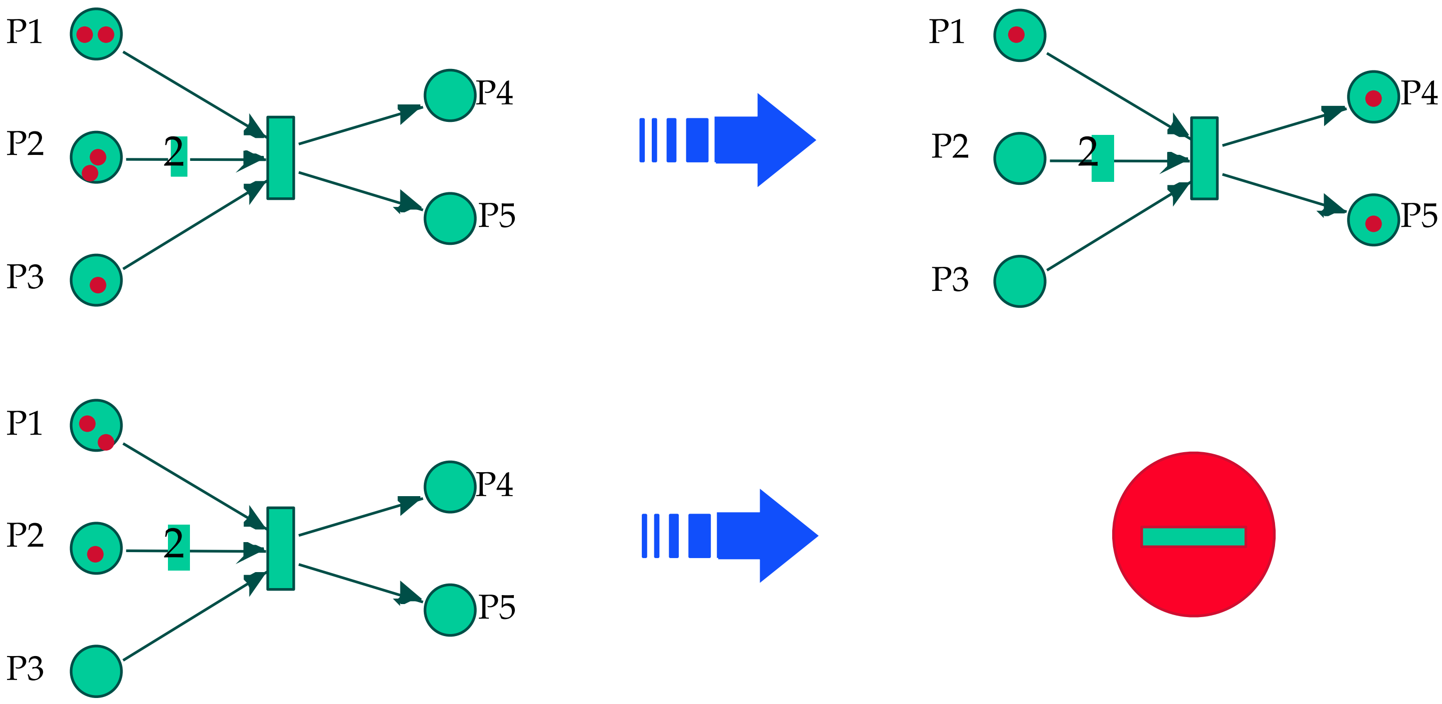
Elements in a Petri net

- Petri nets = bipartite graph
- A state transition model
  - Resources k Places
  - Evolution k Transitions
  - Evolution k Arcs + tokens (firing rule)



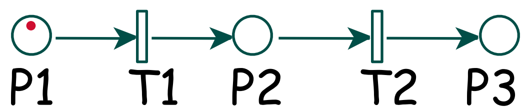
The firing rule

- Defines the behavior of the system

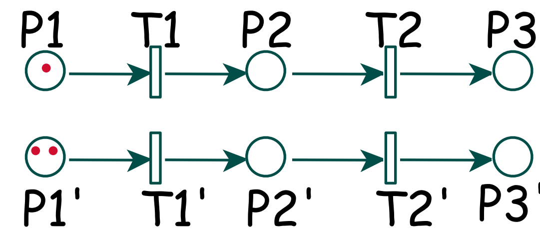


How to define the basics of distributed execution

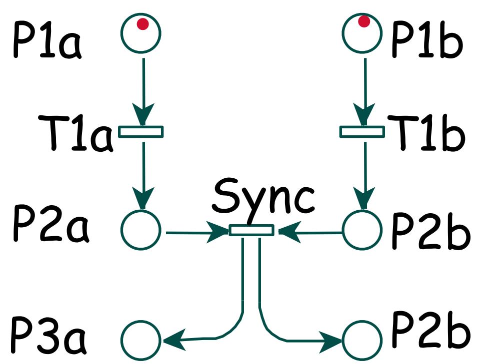
Sequence



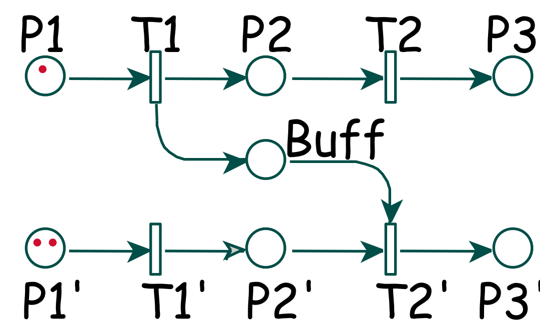
Parallelism



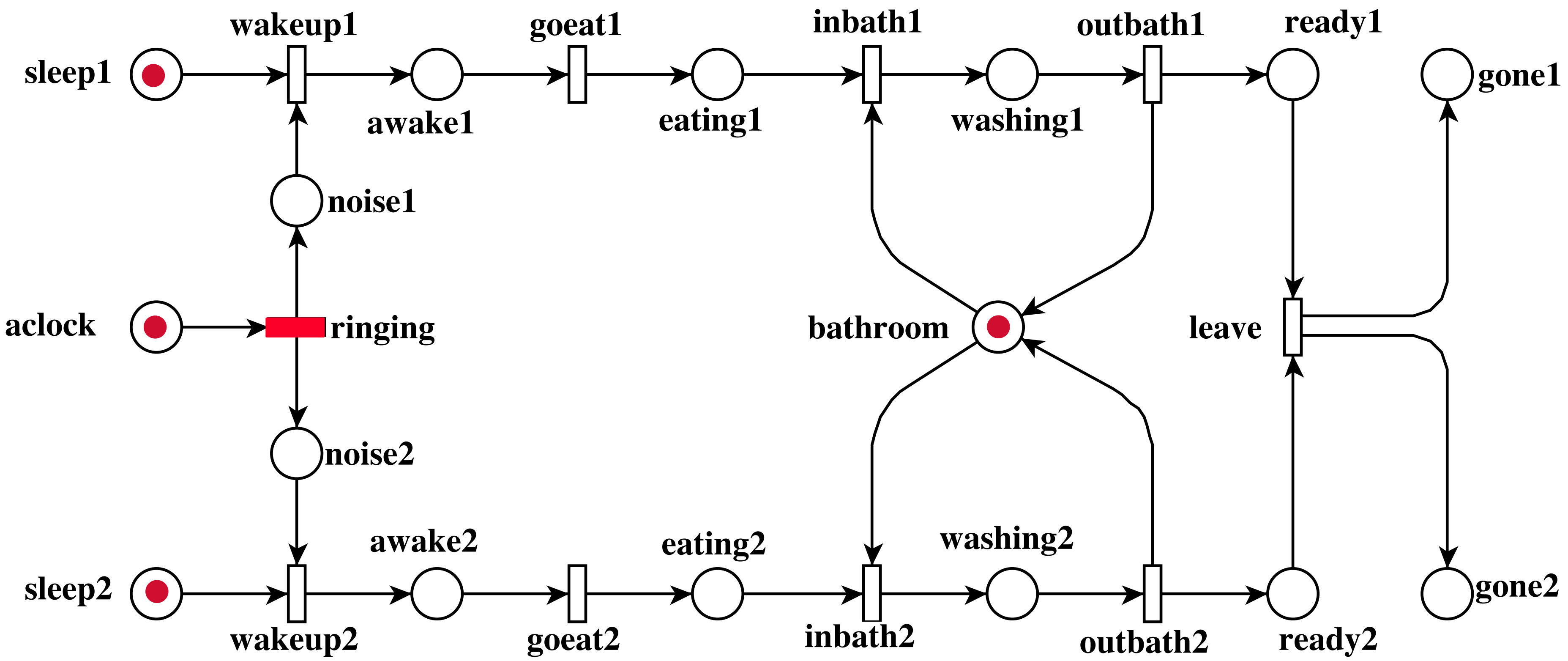
Synchronous communication



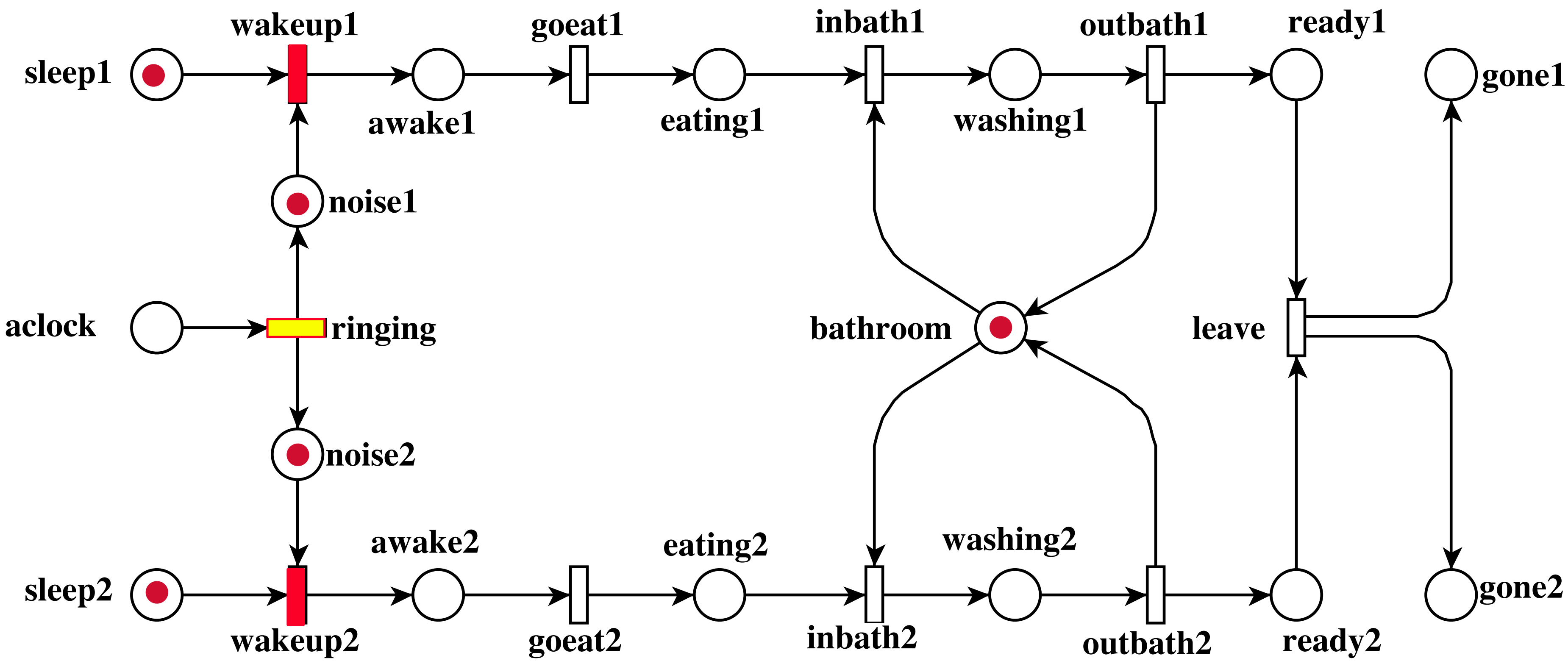
Asynchronous communication



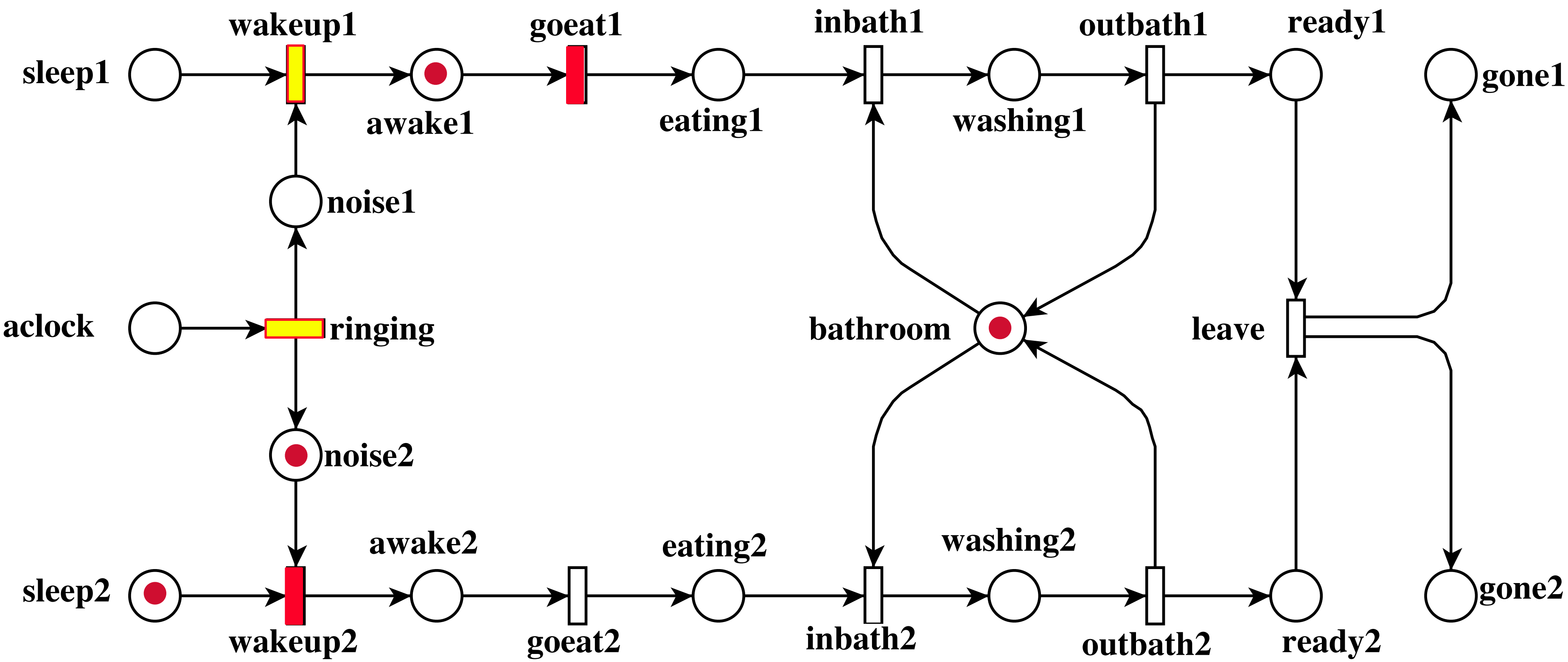
First example: two people waking up (1)



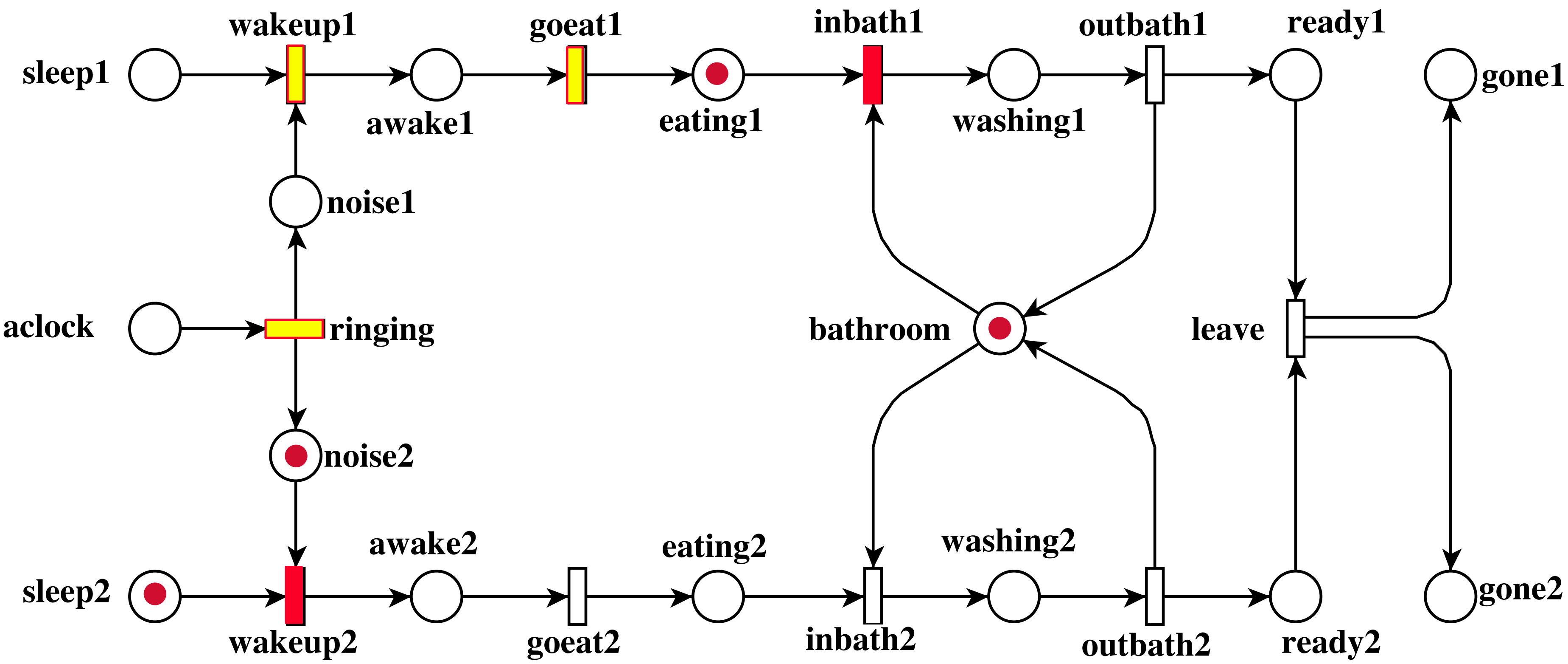
First example: two people waking up (2)



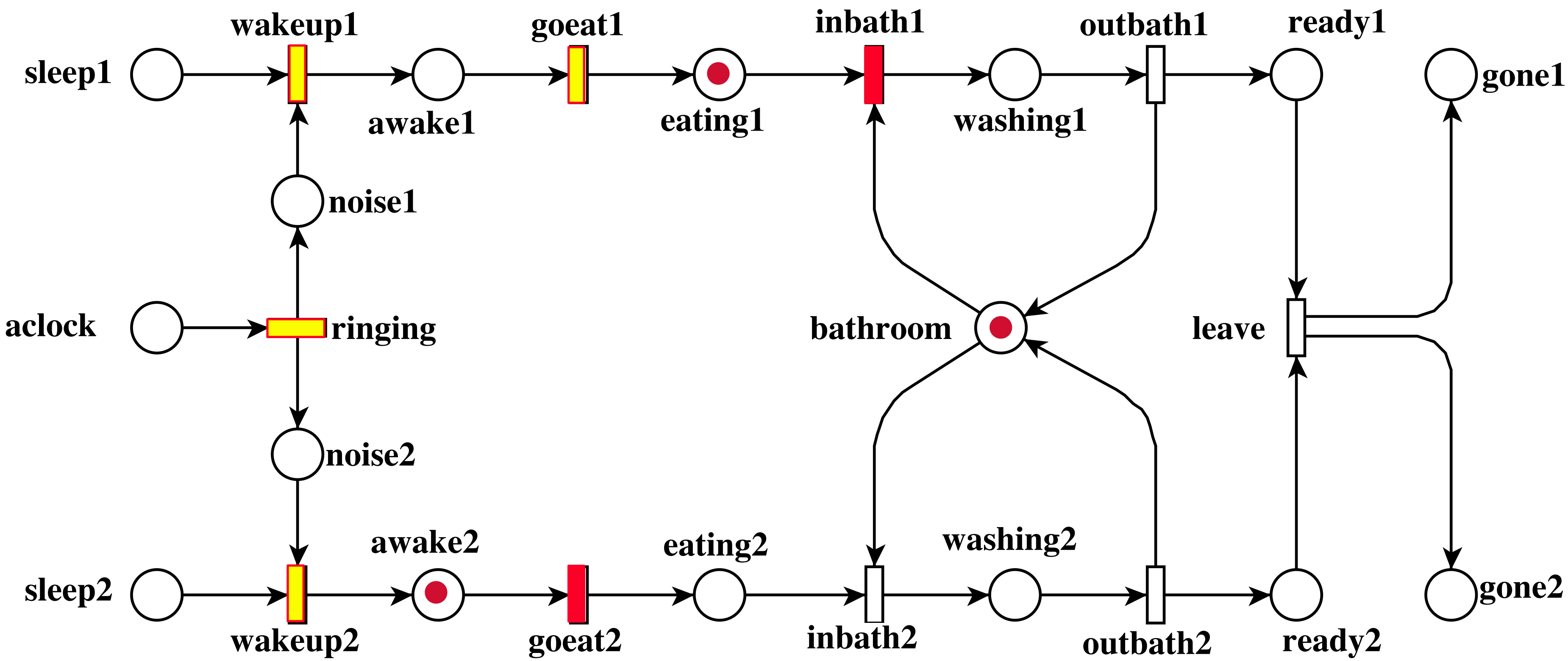
First example: two people waking up (3)



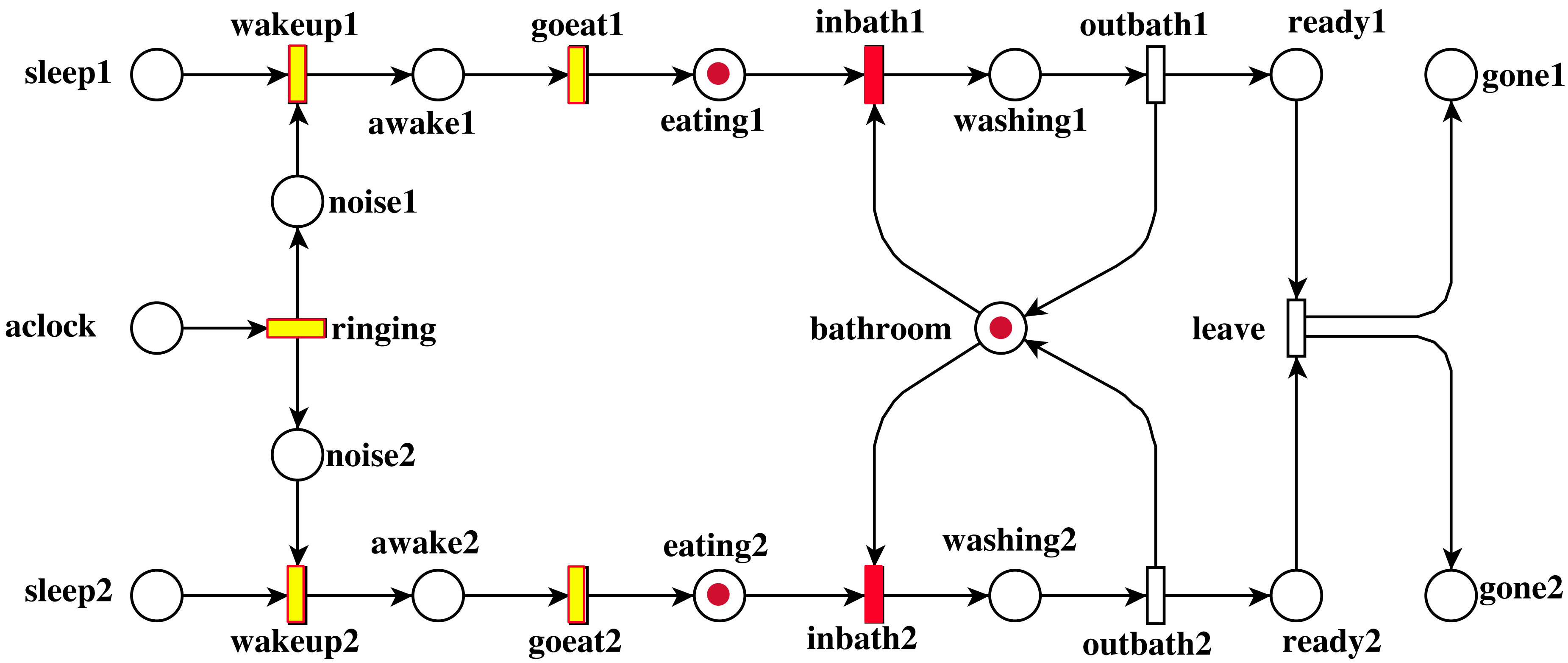
First example: two people waking up (4)



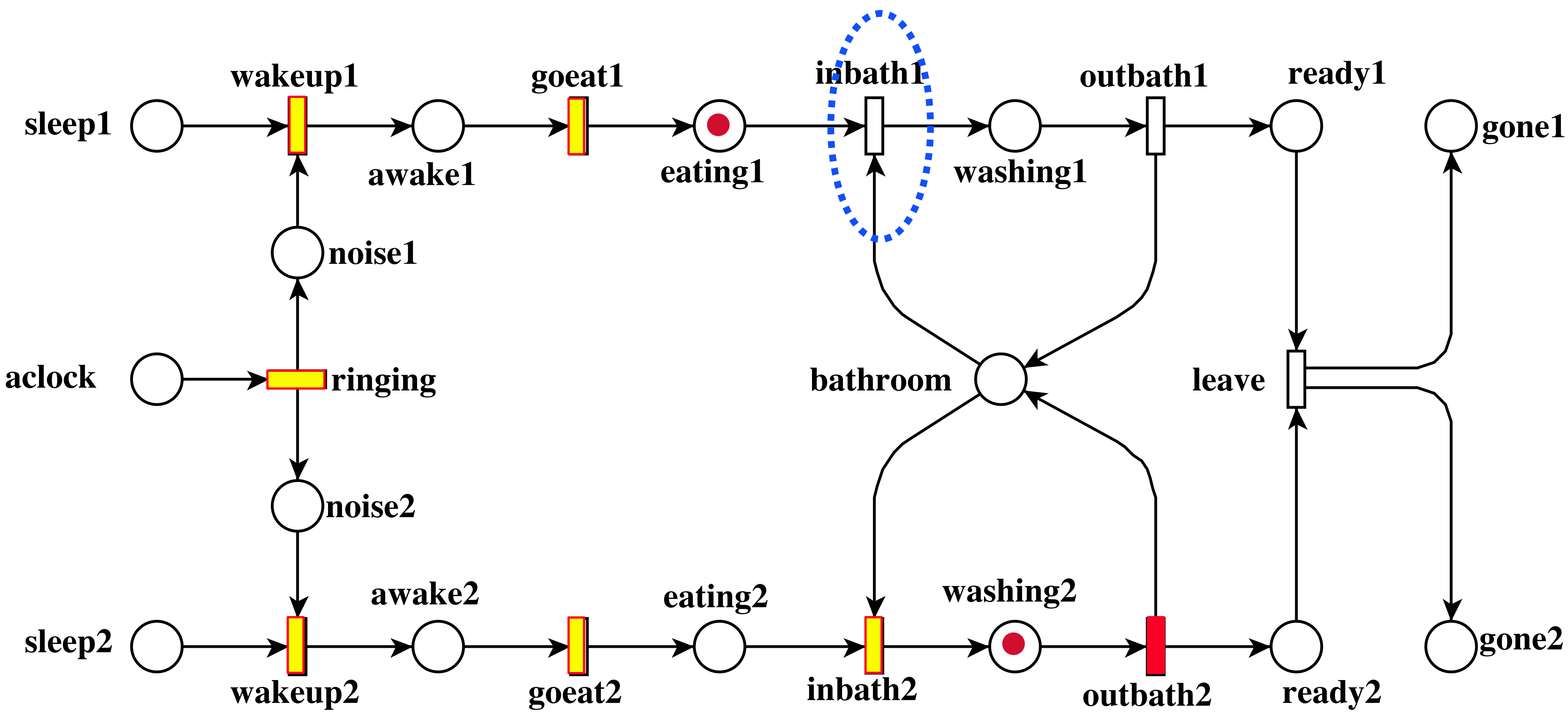
First example: two people waking up (5)



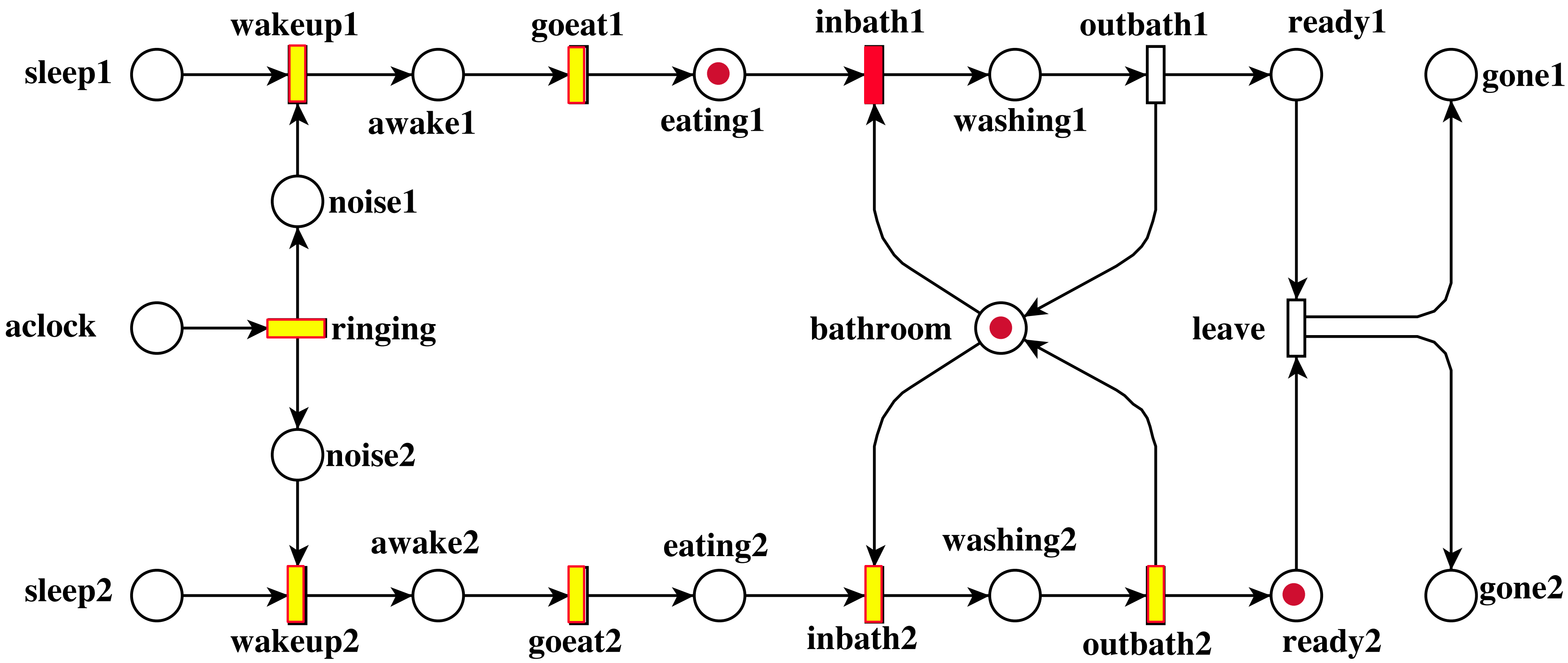
First example: two people waking up (6)



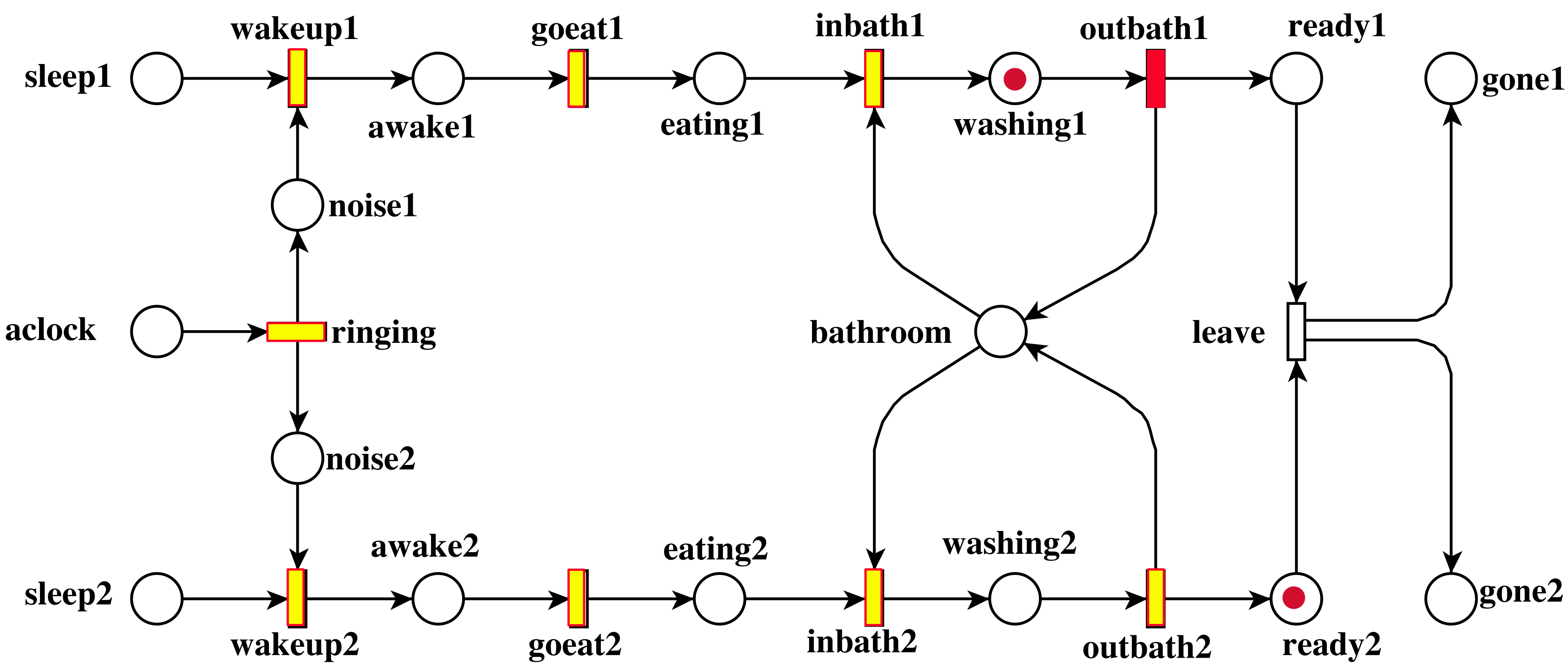
First example: two people waking up (7)



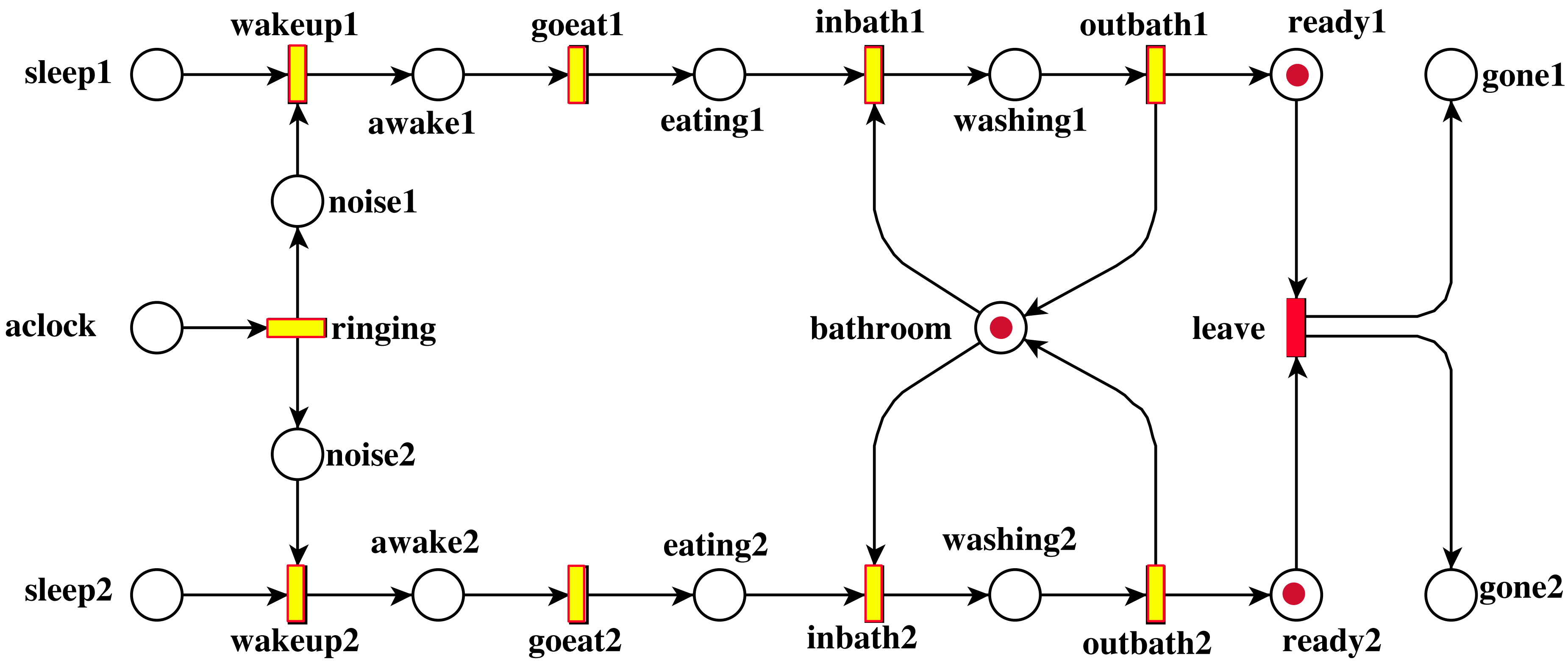
First example: two people waking up (8)



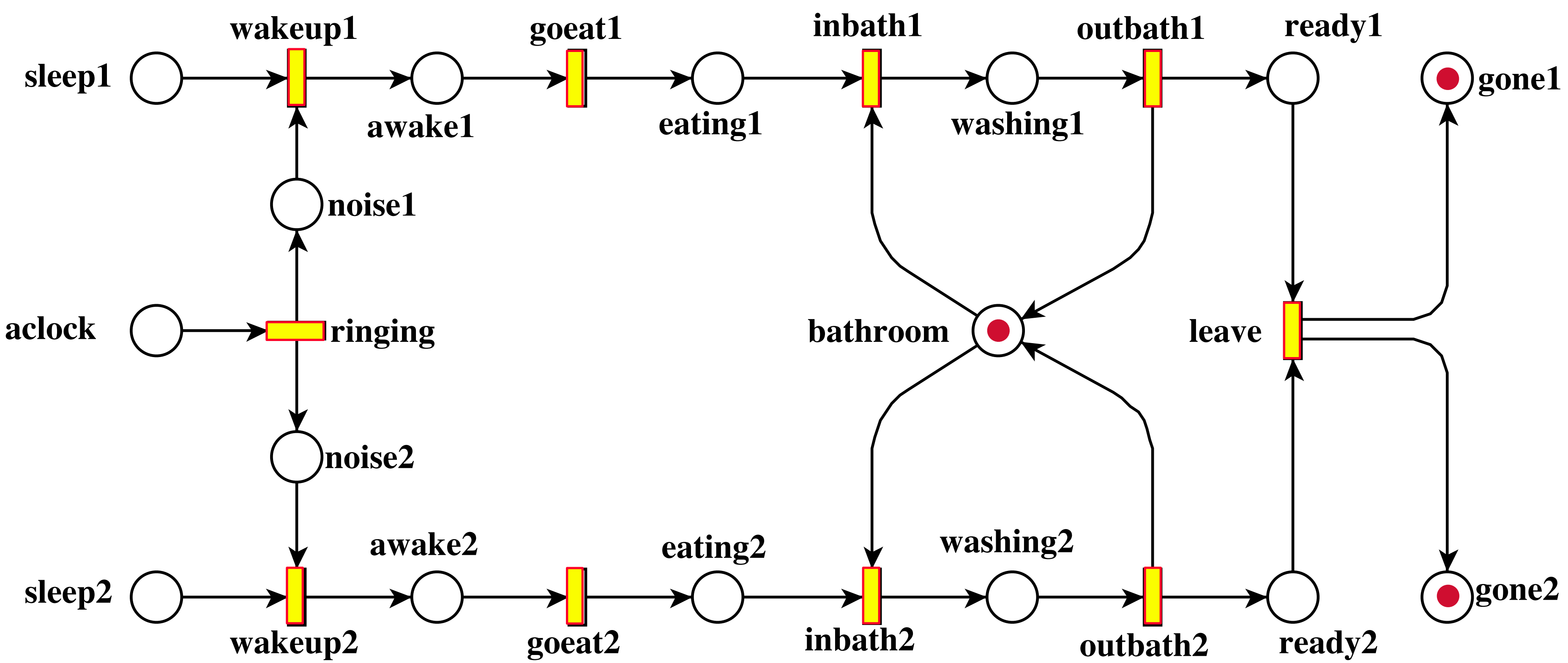
First example:
two people waking up (9)



First example: two people waking up (10)

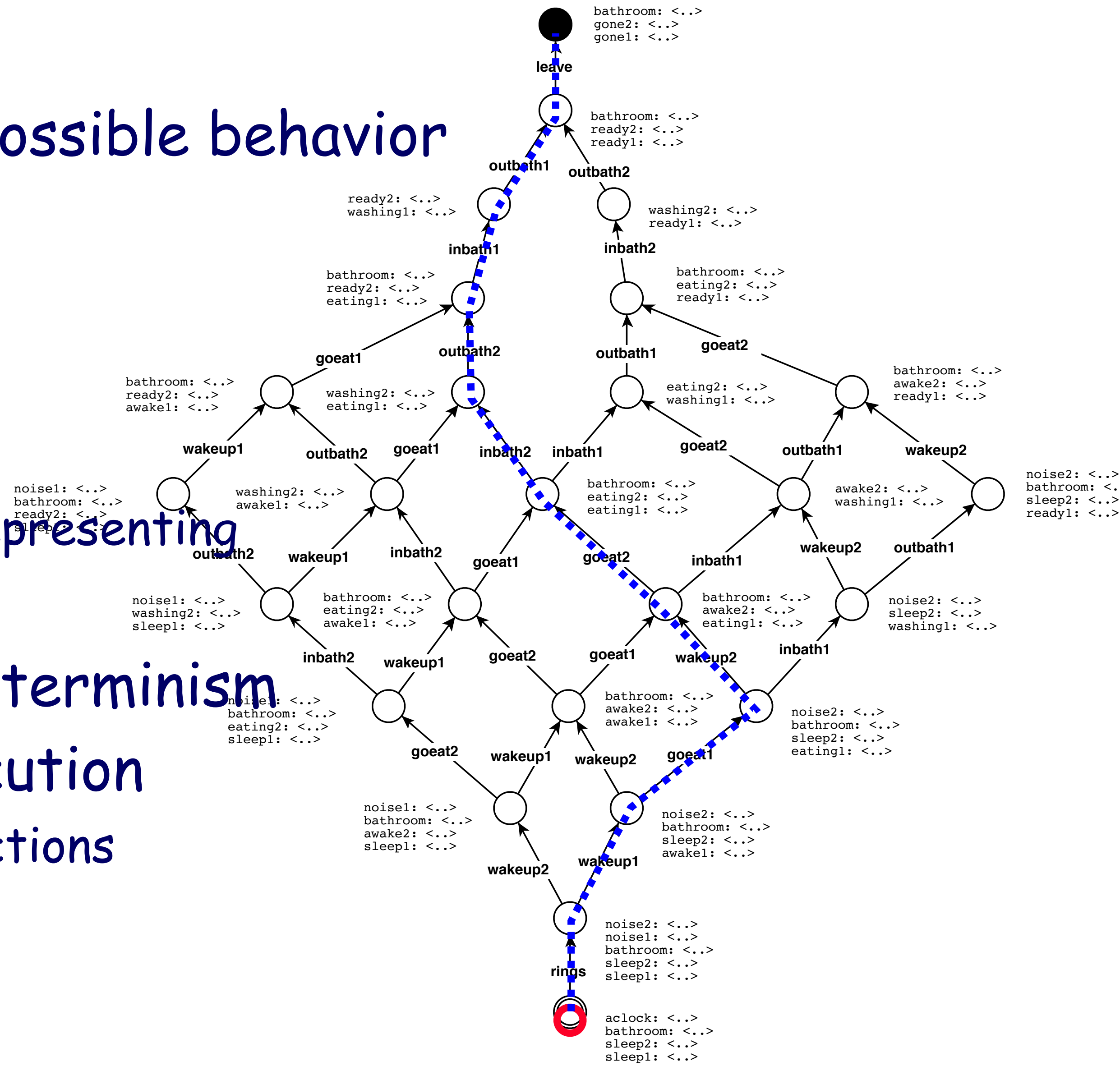


First example: two people waking up (11)



The state space for this model

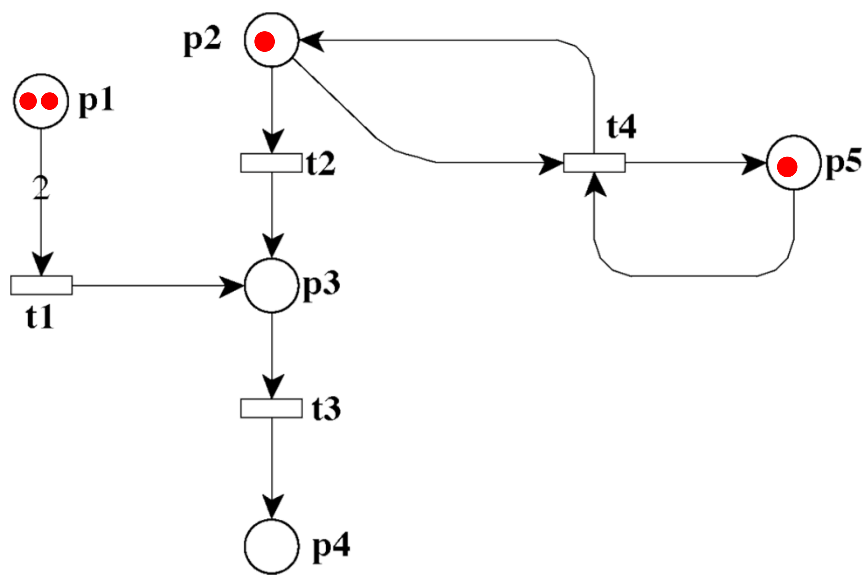
- Expresses all possible behavior in the system
  - 26 states
  - 38 arcs
- One state
  - Integer vector representing marking of places
- Expresses indeterminism of a parallel execution
  - Interleaving of actions

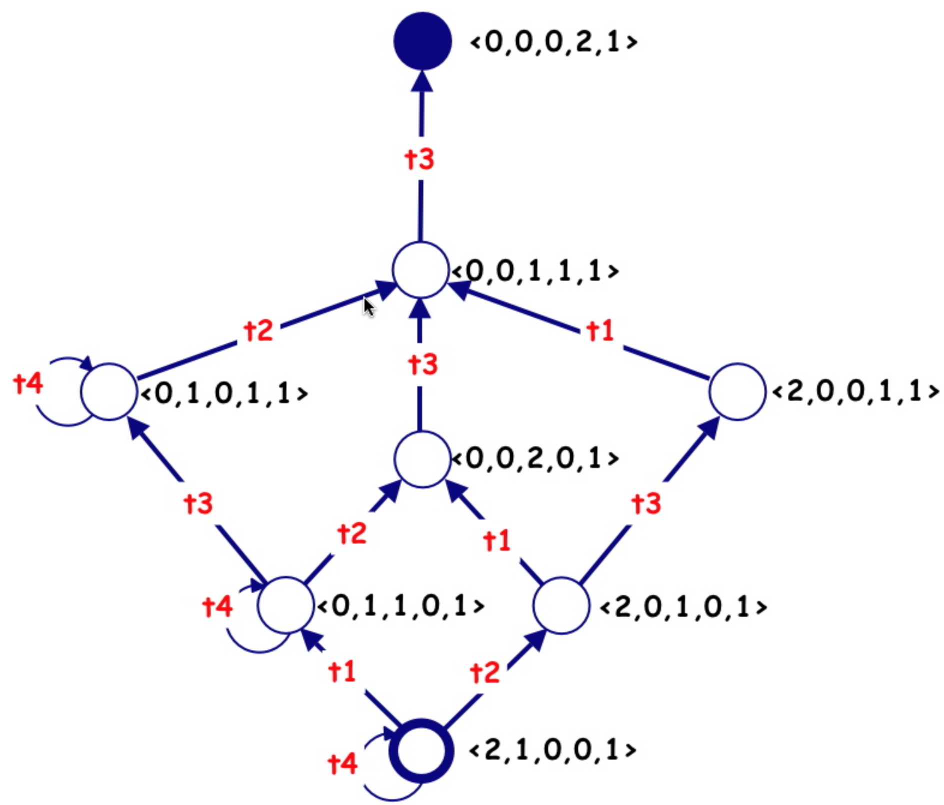


Building the state space (also called reachability graph)

- It is important to relate the network with its reachability graph
- Representation of a state as a vector of place marking

p1	P2	P3	P4	P5





Some formal definitions on Petri Nets

What is a Petri Net

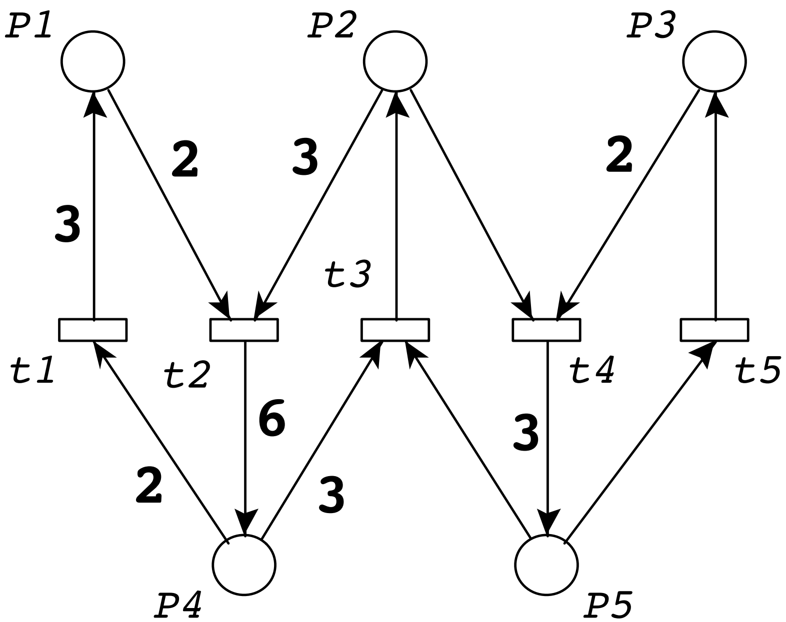
- Definition: a Petri net is a tuple $PN =< P, T, Pre, Post >$ where
  - P = finite (and non empty) set of places
    - ✓Represents «resources»
  - T = finite (and non empty) set of transitions distinct from P
    - ✓Represents relationships between resource consumption and resource production
  - $Pre : P \times T \rightarrow \mathbb{N}$

    $Pre(p, t) = n$ represents how the firing of t is related to a resource in p if n = 0, then, no relation, if n > 0, then, n tokens are required in p to fire t
  - $Post : P \times T \rightarrow \mathbb{N}$

    $Post(p, t) = n$ represents how the firing of t is generates tokens in p, n tokens are produced in p when t is fired
  - M_0 = the initial state noted of the system
- $< PN, M_0 >$ denotes a System with its initial state

Initial marking, example

- Initial marking



$$M_0 = \begin{vmatrix} 3 \\ 4 \\ 2 \\ 0 \\ 0 \end{vmatrix}$$

- Remind, each state in the state space is represented using a vector of places

Firing a transition

Firing rule

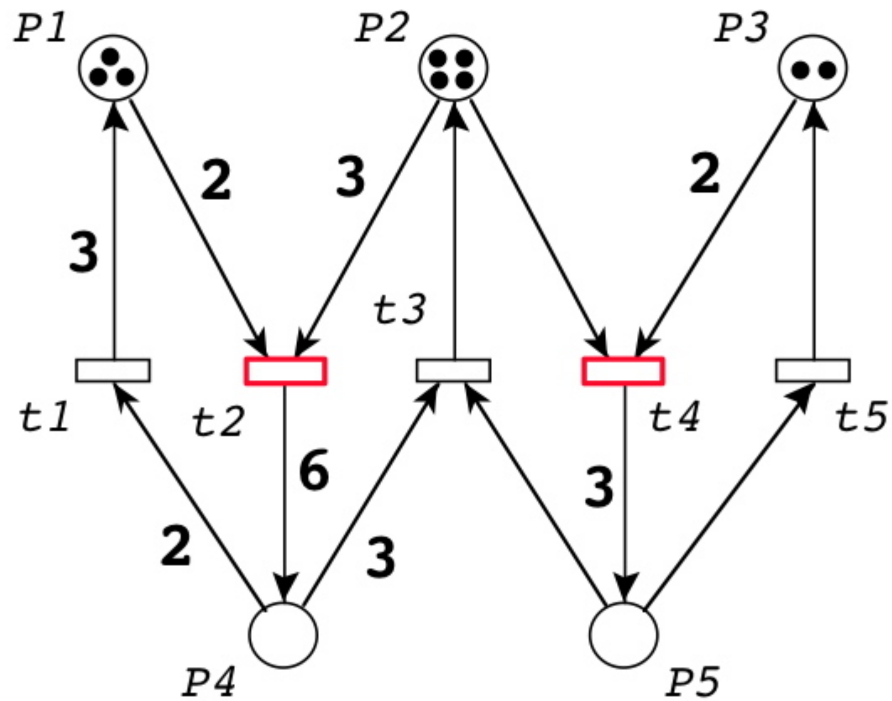
- $\bullet t \in T$ can be fired from a marking M iff $forall p \in P, M(p) \geq Pre(p,t)$
- if t can be fired, then, its firing leads to a new state M' build as follow

$$\forall p \in P, M'(p) = M(p) - Pre(p,t) + Post(p,t)$$

- Firing of t is noted: *M[p>M'*

«Firability» of a transition, example

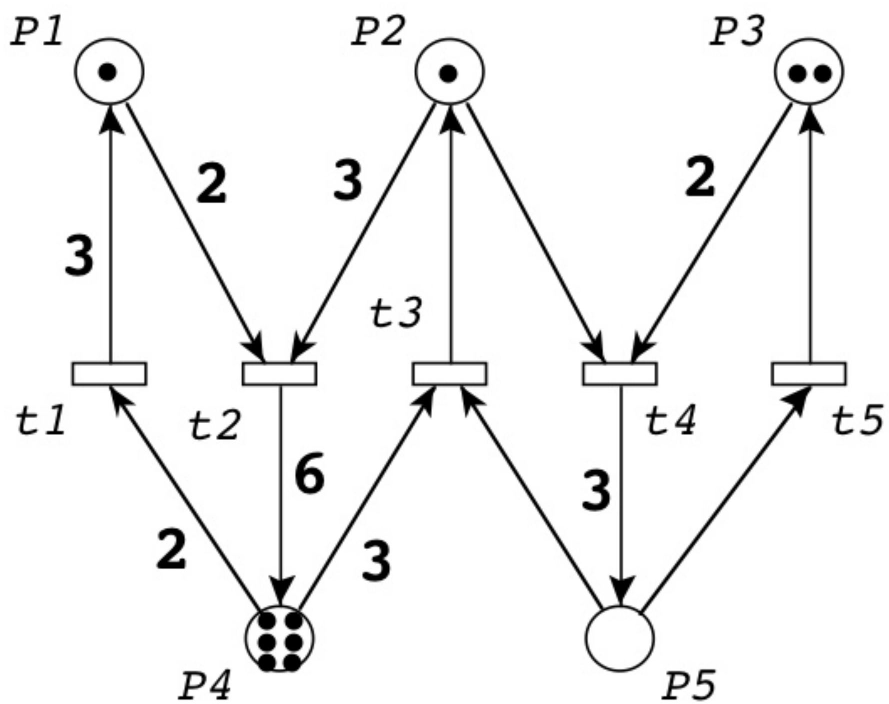
- *t2* and *t4* can be fired from M_0



- We can note this: $M_0[t2>$ and $M_0[t4>$

Firing a transition, example (1)

- Let us fire $t2$ from M_0, then, we reach a new state M_1

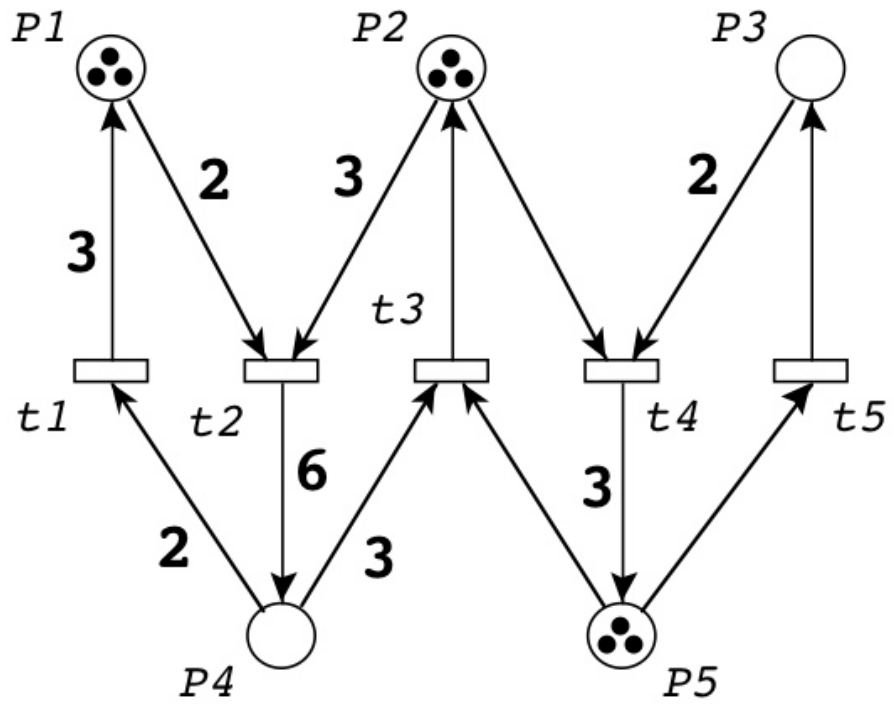


$$M_1 = \begin{vmatrix} 1 \\ 2 \\ 2 \\ 6 \\ 0 \end{vmatrix}$$

- This can be noted $M_0[t2 > M_1$

Firing a transition, example (2)

- If we fire *t4* from M_0, then we reach a new state M_2



$$M_2 = \begin{vmatrix} 3 \\ 3 \\ 0 \\ 0 \\ 3 \end{vmatrix}$$

- This can be noted $M_0[t4 > M_2$

Firing sequence

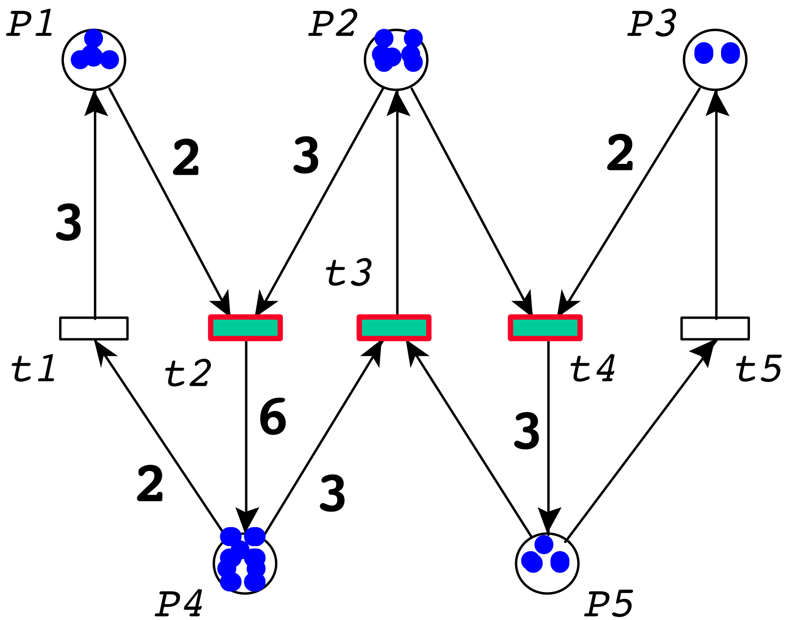
- **Definition:**

A sequence of firing from M_0 to M_n is a word $t_0 \ldots t_{n-1}$ where there exists marking $M_1, \ldots, M_{n-1}$ verifying

$$M_0[t_2 > M_1 \ \ldots \ M_{n_1}[t_n > M_n$$

Firing sequence, example

t2 t4 t3 is a firing sequence from M_0



$$M_0 [t_2 > M_1 [t_4 > M_3 [t_3 > M_4$$

$$M_0 = \begin{vmatrix} 3 \\ 4 \\ 2 \\ 0 \\ 0 \end{vmatrix} \quad M_1 = \begin{vmatrix} 1 \\ 2 \\ 2 \\ 6 \\ 0 \end{vmatrix} \quad M_3 = \begin{vmatrix} 1 \\ 0 \\ 0 \\ 6 \\ 3 \end{vmatrix} \quad M_4 = \begin{vmatrix} 1 \\ 1 \\ 0 \\ 3 \\ 2 \end{vmatrix}$$

Incidence matrix

- Let be PN a Petri net. We define W the incidence matrix of PN where:

$$W = Post - Pre$$

- From the firing aspect, let us consider that for $M[t > M'$, we have:

$$\forall p \in P,$$
$$M'(p) = M(p) + Post(p,t) - Pre(p,t)$$
$$\text{or}$$
$$M'(p) = M(p) + W(p,t)$$

Reachability Graph
(state space for the system)

- Definition: the reachability graph for a system $< PN, M_0 >$ is a transition system (a transition graph) $< Q, \Delta, \lambda, q_0 >$ where:
  - Q is the set of marking that can be reached in PN from M_0

$$Q = \{M \mid M \in \mathbb{N}^P \text{ and } \exists \sigma \in T^* / \ M_0[\sigma > M\}$$

  - Δ is the set of arcs that relates two reachable states in PN from M_0

$$\{(q_1, q_2) \in Q \times Q \mid t \in T, q_1[t > q_2\}$$

  - λ represents arc label (name of the transition fired in PN)
  - q_0 represents the initial marking M_0

Sample algorithm
to build the state space

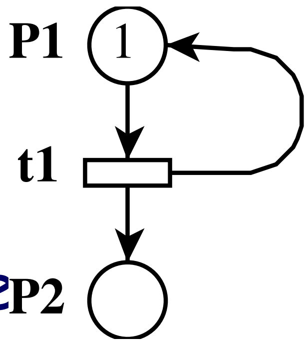
- Easy to understand...

$newSates = M_0$
$G = < \{M_0\}, \emptyset, id, M_0 >$
while newSates $\neq \emptyset$ **do**
    crtState = extracElem (newSates)
    newSates = newSates - crtState
    for $\forall t \in T$ **do**
        if crtState $[t >$ **then**
            crtState $[t >$ nextState
            if $\neg$ nextState $\in$ G **then**
                Create nextState
                G = G + nextState
                newSates = newSates + nextState
            fi
            G = G + arc between crstState and nextState
        fi
    done
done
return G

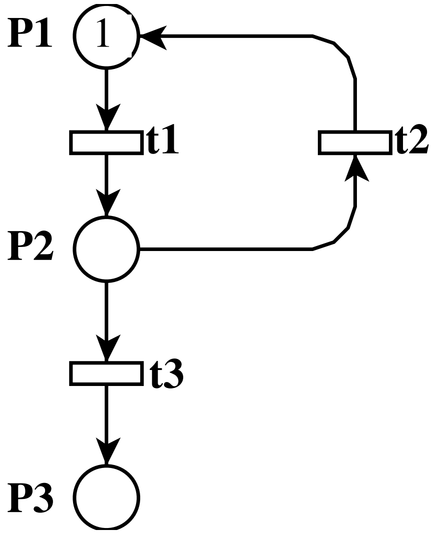
- ... but is

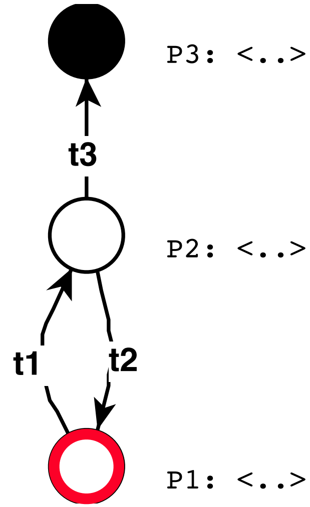
Some remarks on the reachability graph

- The generated state space (reachability graph) is related to both
  - *PN*
  - M_0



- A state space can be infinite
- A finite state space may contains infinite sequences





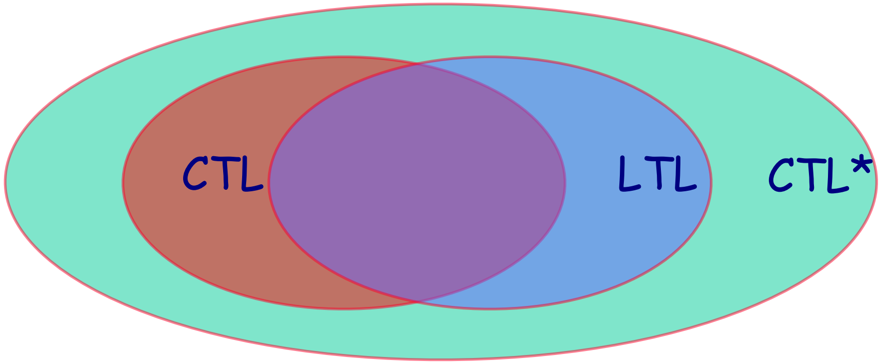
Some properties of Petri Nets

Type of properties

- Behavioral properties
  - Verification of a formula on the associated state space
    - Need to deploy the reachability graph
  - Two types of behavioral properties
    - **Safety** (formula to be verified by all states)
      use of formula on states or on transitions
    - **Causal** (relation between two or more states)
      use of temporal logic
- Structural properties
  - Related to the structure of the specification
    - No need to compute the reachability graph
  - The correspond to patterns in the reachability graph

Model checking and temporal logic

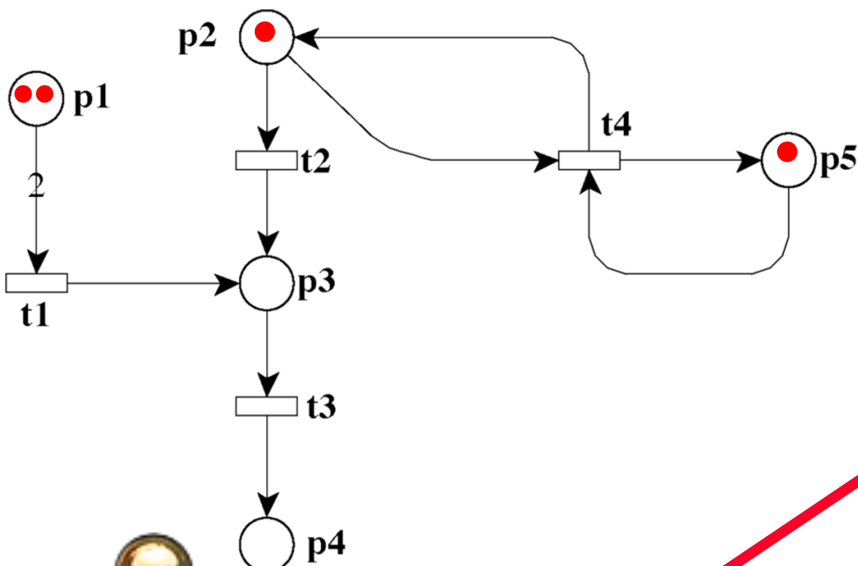
- Temporal ≠ timed management
  - Causality between two actions
  - Set up «good» relationship between critical events in the system
- Safety
  - Search for a given state configuration
- Temporal
  - Operators
    - possible in the future, always in the future, eventually
  - Atomic properies
    - safety-like formulæ
- Several temporal logic
  - CTL (computation tree logic)
  - LTL (linear time logic)
  - CTL* (both)

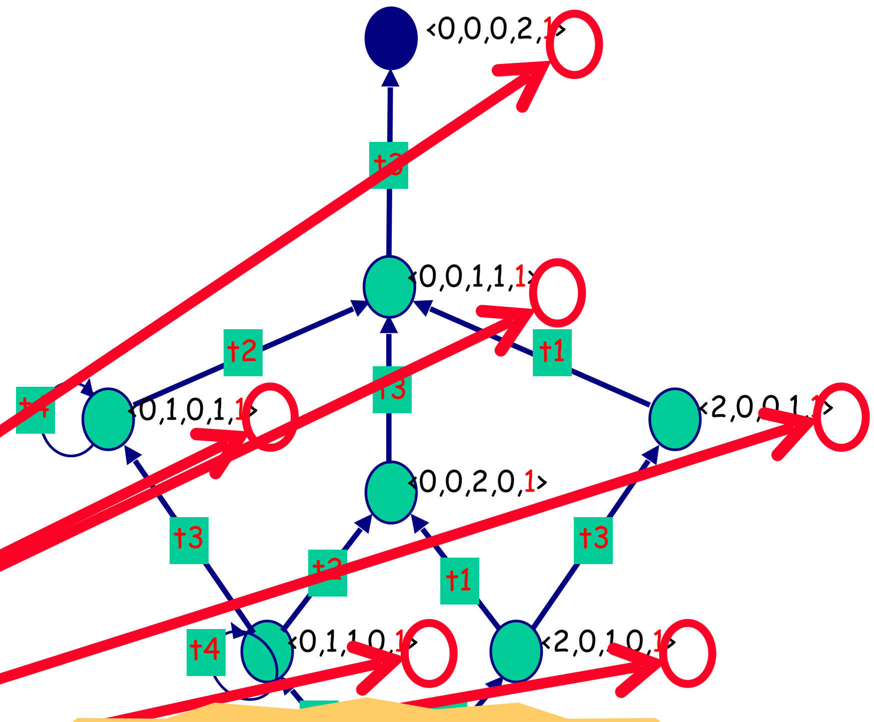


useful to check for specific states (safety) or causal properties (temporal formulæ)

Place invariants

- Pondered marking over a set of places = constant (depends on the initial marking)
  - This formula is verified all over the reachability graph



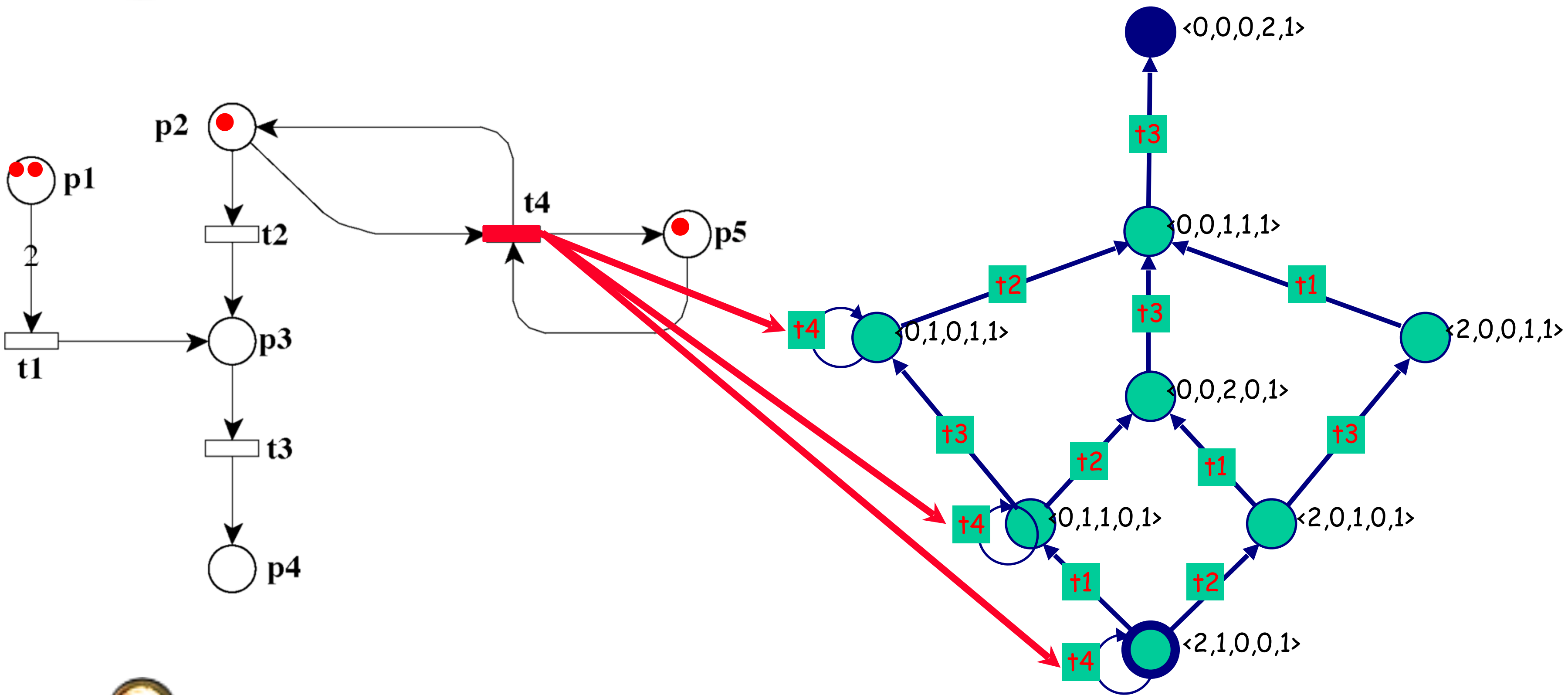


- On the example:
  - 2*p2 + 2*p3 + 2*p4 + p1
  - p5

useful to check for sequences (threads) or to verify mutual exclusion

Transition invariants

- Stationary sequence (when it can be fired)

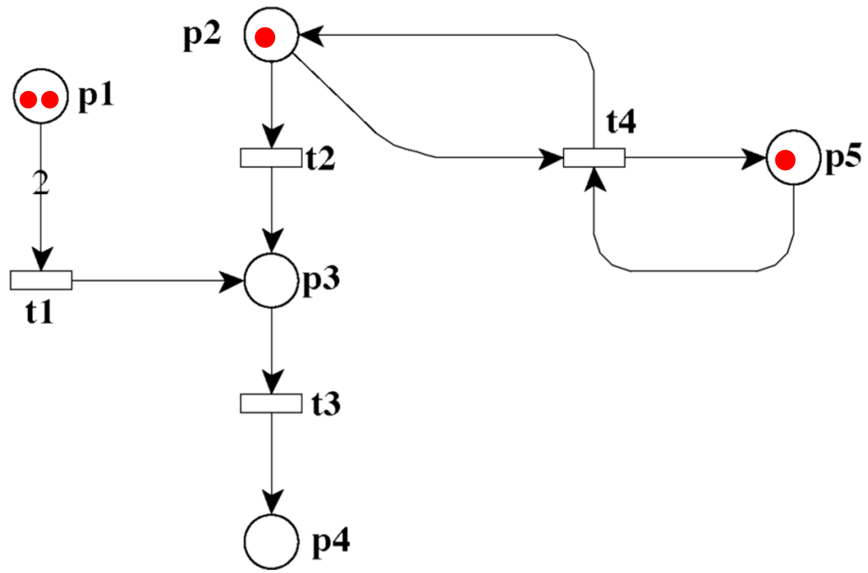


- In the example:
  - t4

useful to check for
expected ciclic behavior

Structural bounds

- Min/Max number of token in a place
- WARNING: structural means may never be reached
  - Depends on the initial state of the system
- On the example:
  - p2 : [0 ... 1]
  - p3 : [0 ... 2]
  - p4 : [0 ... 2]
  - p1 : [0 ... 2]
  - p5 : [1 ... 1]



useful to check for communication bounds and feasibility of model checking

Component-based methodology for behavioral modeling

Modeling strategy

- Model = «story»
  - How to build the model (what abstraction level, what choices)
  - The story relies on components (execution sequences, threads, etc.)
  - The story brings modeling hypotheses
- Thus, there are «expected properties»
  - «Good questions » must be raised for a given specification
- Typical example: structural properties (several use)
  - To check the design
    - Such properties should be there (otherwise, things could be wrong)
  - Then, to verify the model
    - Properties dedicated to the expected properties

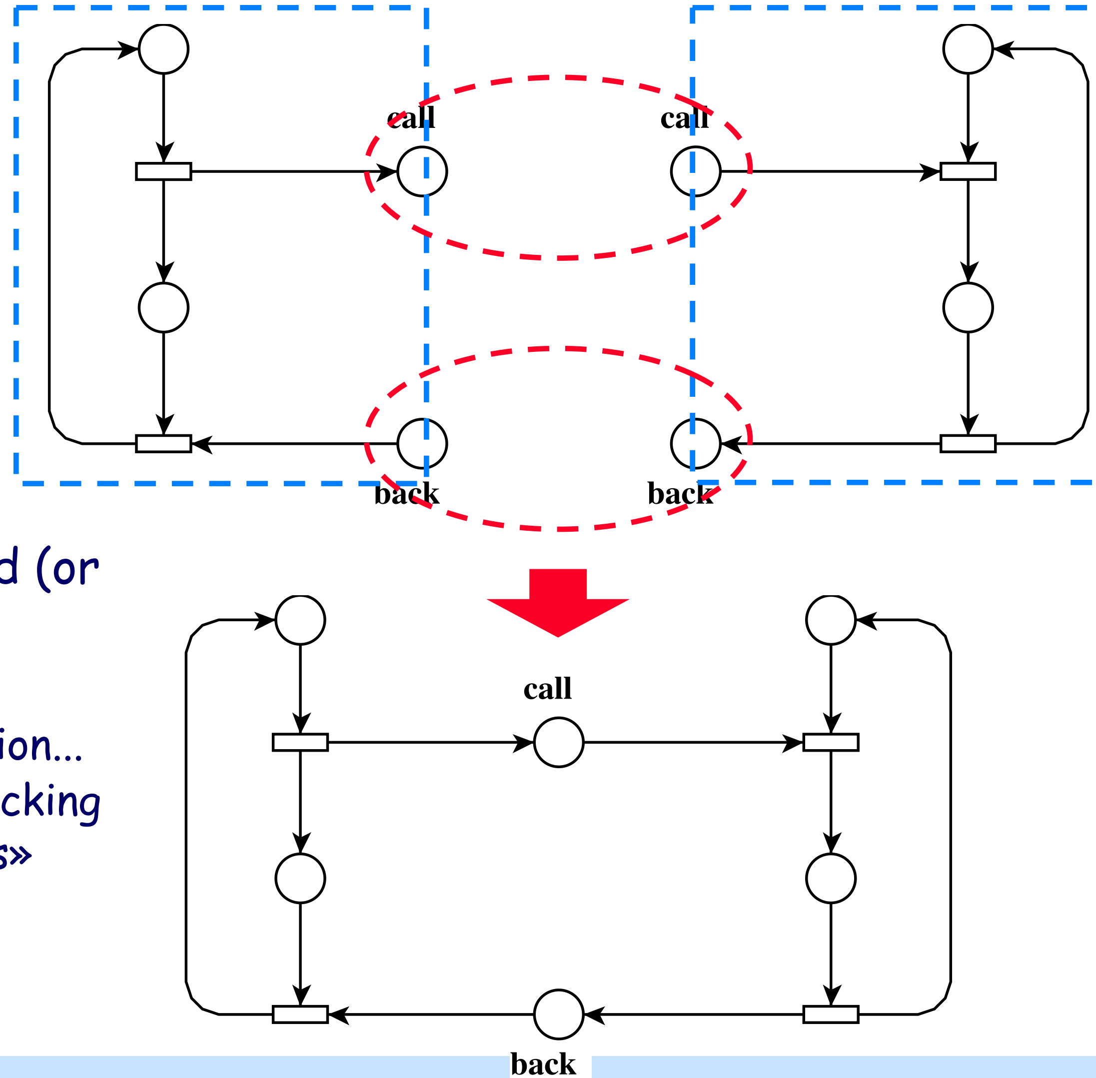
Modeling and verification process

- The process
  - Evaluate what do you want to model (1)
  - Evaluate what properties do you want to verify (2)
  - Select your abstractions (according to 1 and 2)
  - Design your model
  - Check for «expected properties» (from the story)
  - Verify the model's properties
- Such a process may seems complex for «simple» models
  - It is the only way to avoid waste of time for larger ones
- For larger models, it is necessary to combine with modularity
  - Then, the process is refined at each level
    - The process is applied for each module - local verification
    - Assemblage is then performed
    - The process is then applied for the entire module

Module interactions

- Basic interactions
  - Channel place — asynchronous
  - Shared transition — synchronous
- More elaborated
  - Subnets with specific behavior assembled using basic interactions
- But sophisticated interaction can be resumed to the basic ones
  - Sophisticated interaction is seen as a component (glue in the previous slide)
- Advantage:
  - Keep on canonical mechanisms
  - Encapsulation of high level mechanisms (UML?)
  - Preservation of some properties (under certain conditions)

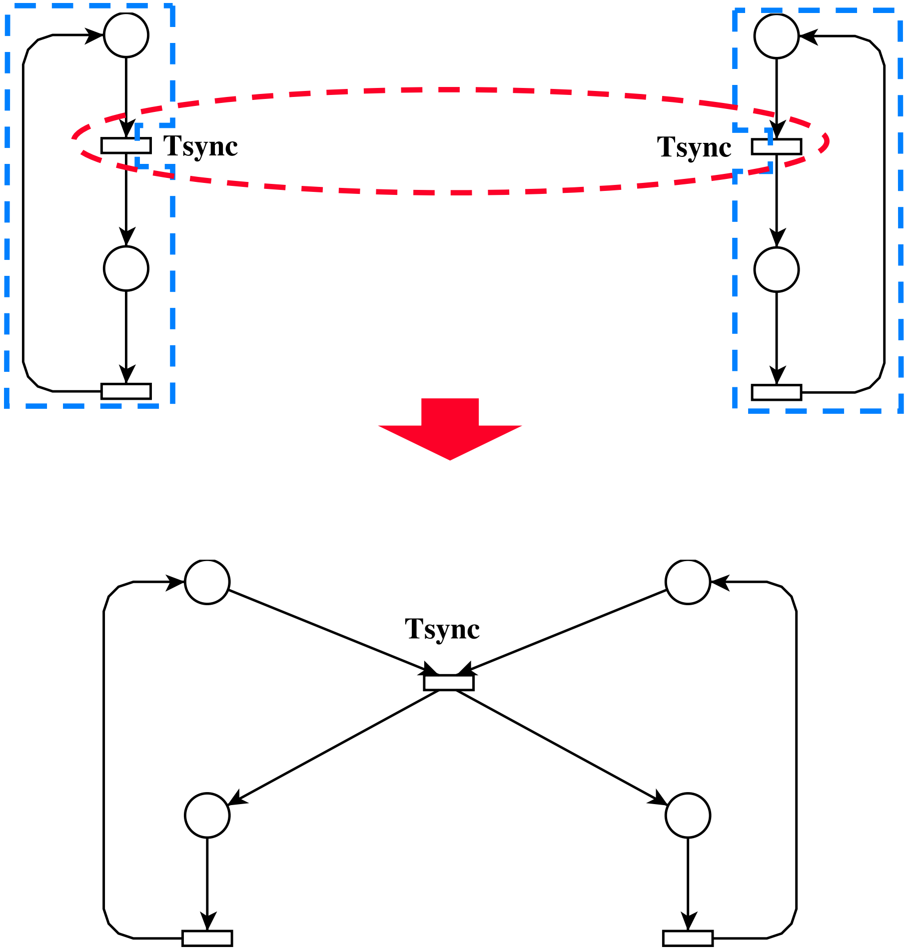
Channel places - place fusion



Preserved properties

- P-invariants may be found (or composed) in the resulted model
  - Under certain configuration…
  - This is useful to keep tracking the «expected properties»

Transition Fusion

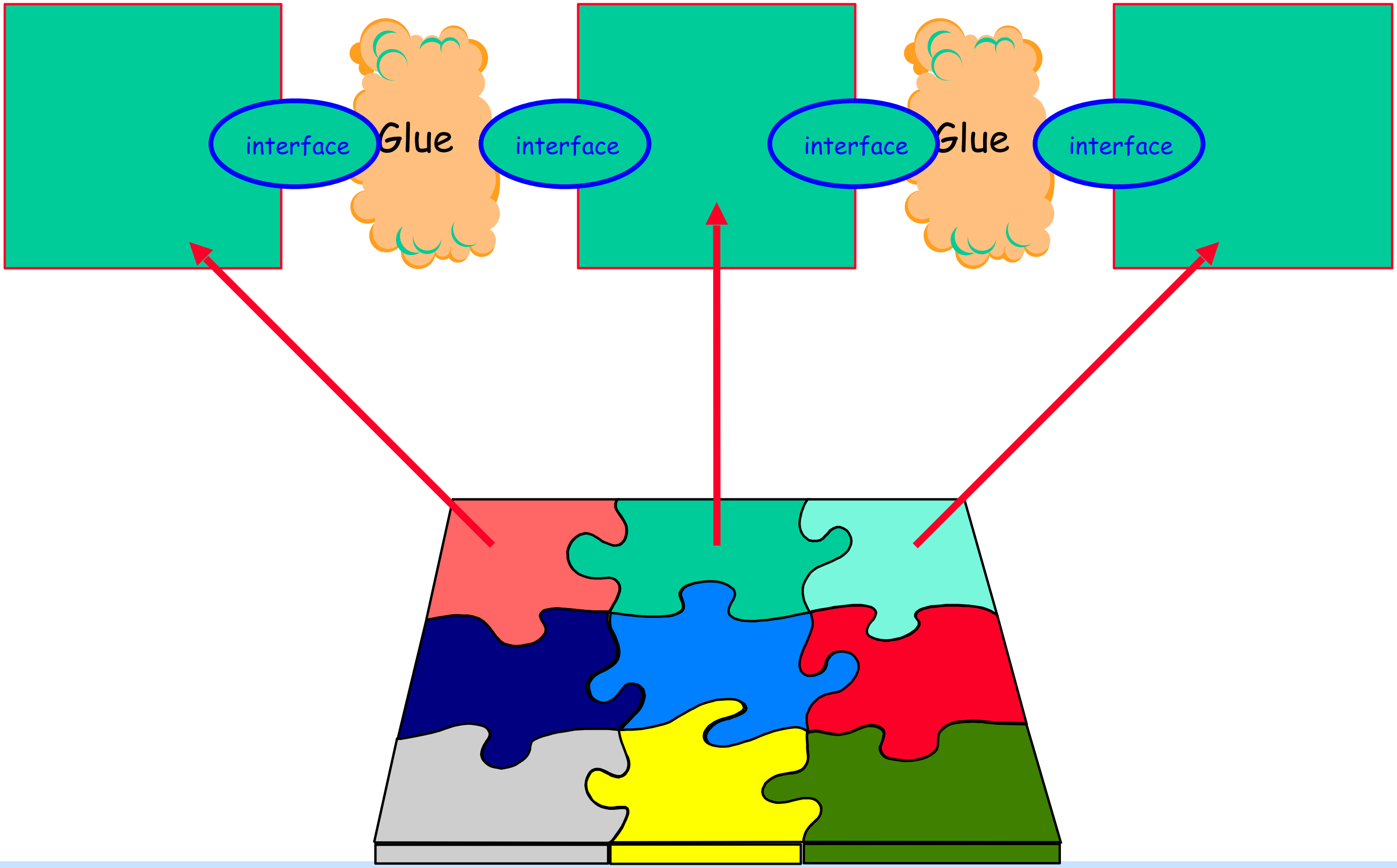


Remark

- P-invariants of the resulting model are a superset of the union of component's invariants
  - Under certain conditions

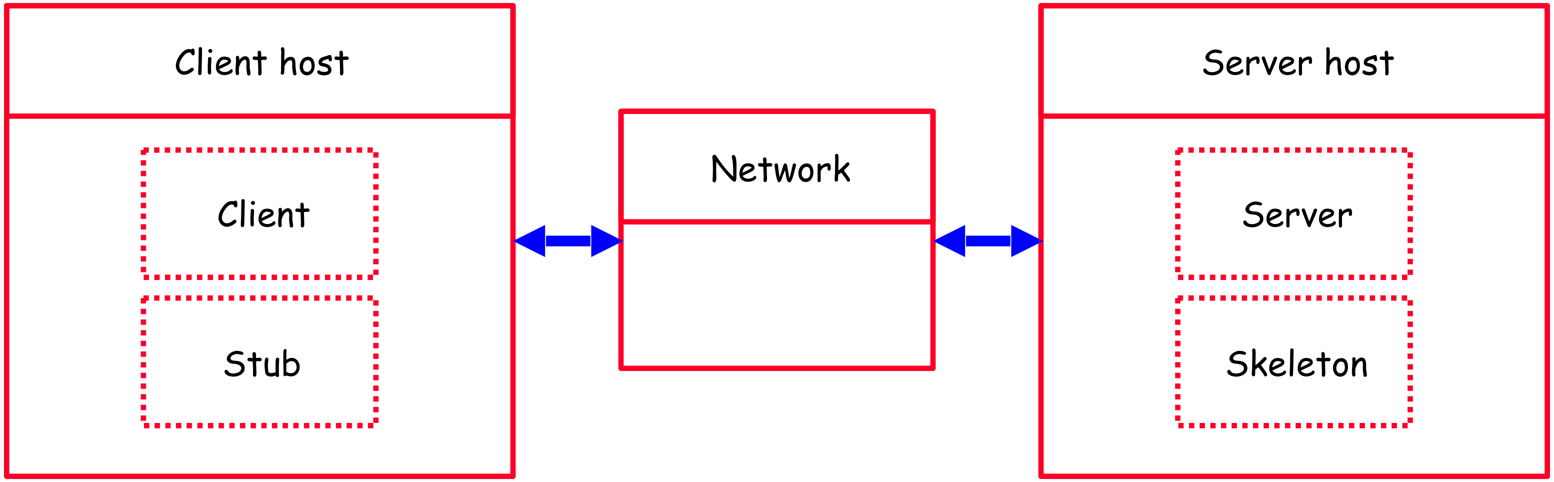
Modularity and basic interactions

- Objective: manage large applications

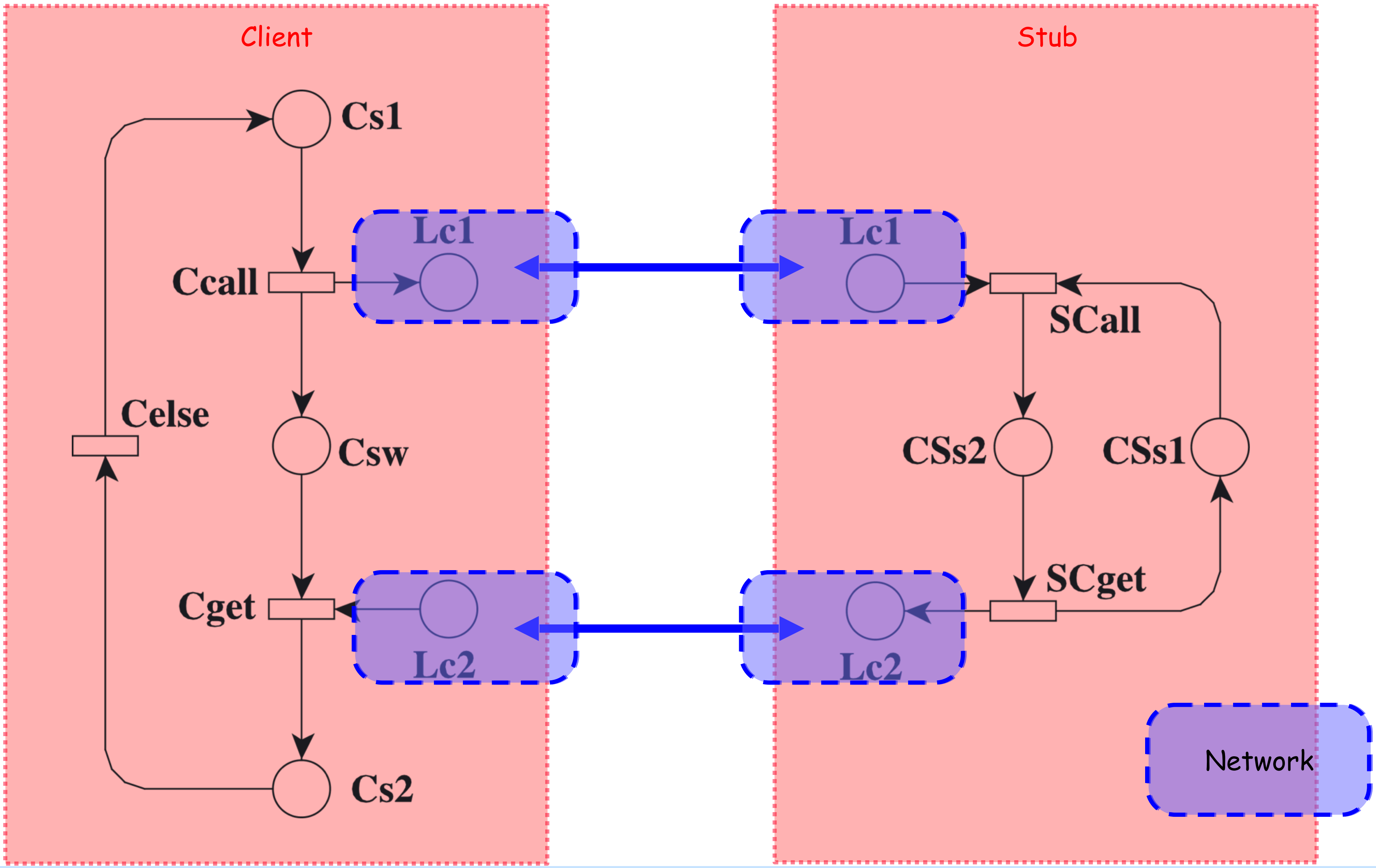


Applying the process to a simple example

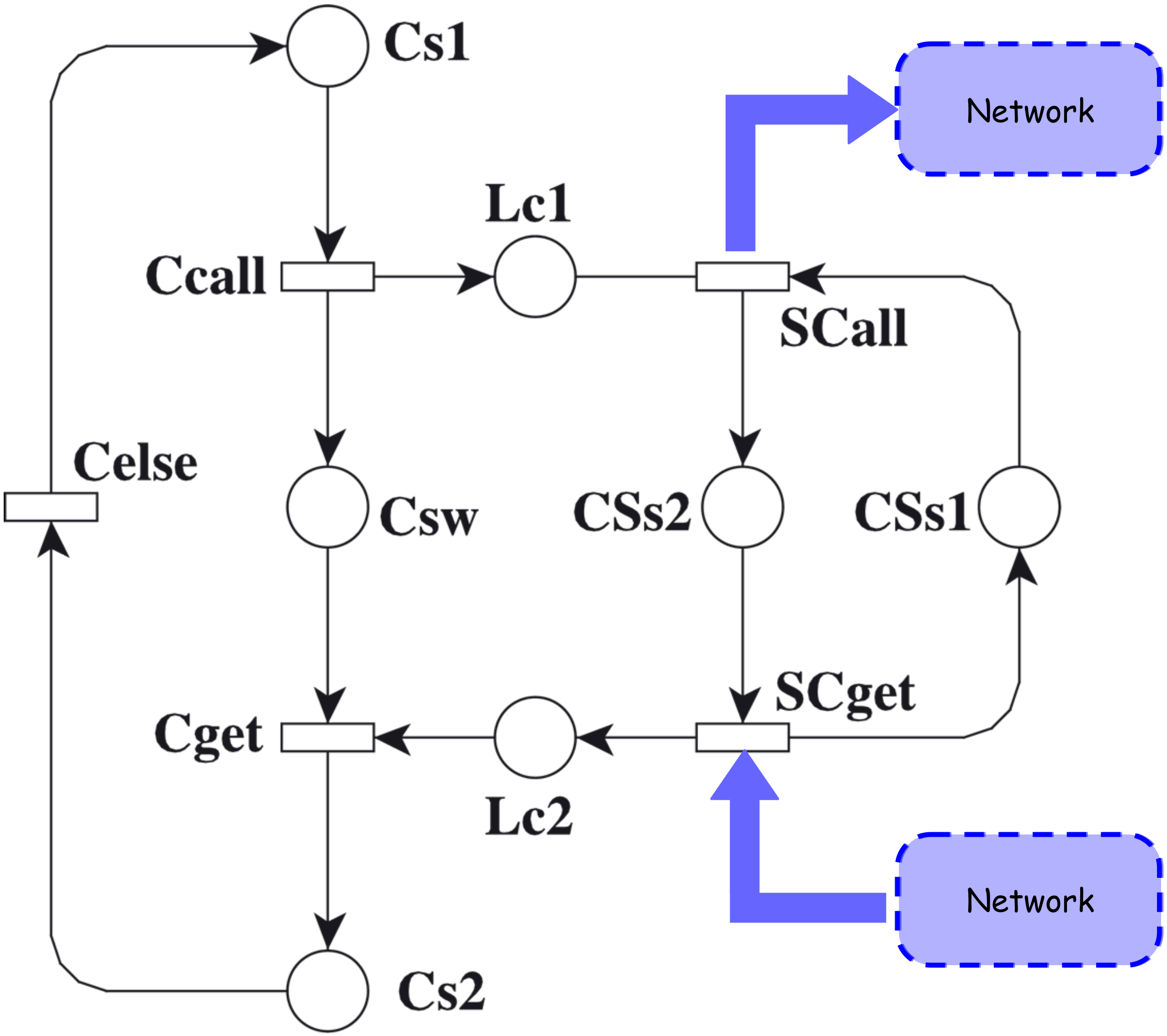
- Modeling two simple CORBA components
  - A client
  - A server
  - Both cooperate to send/receive requests



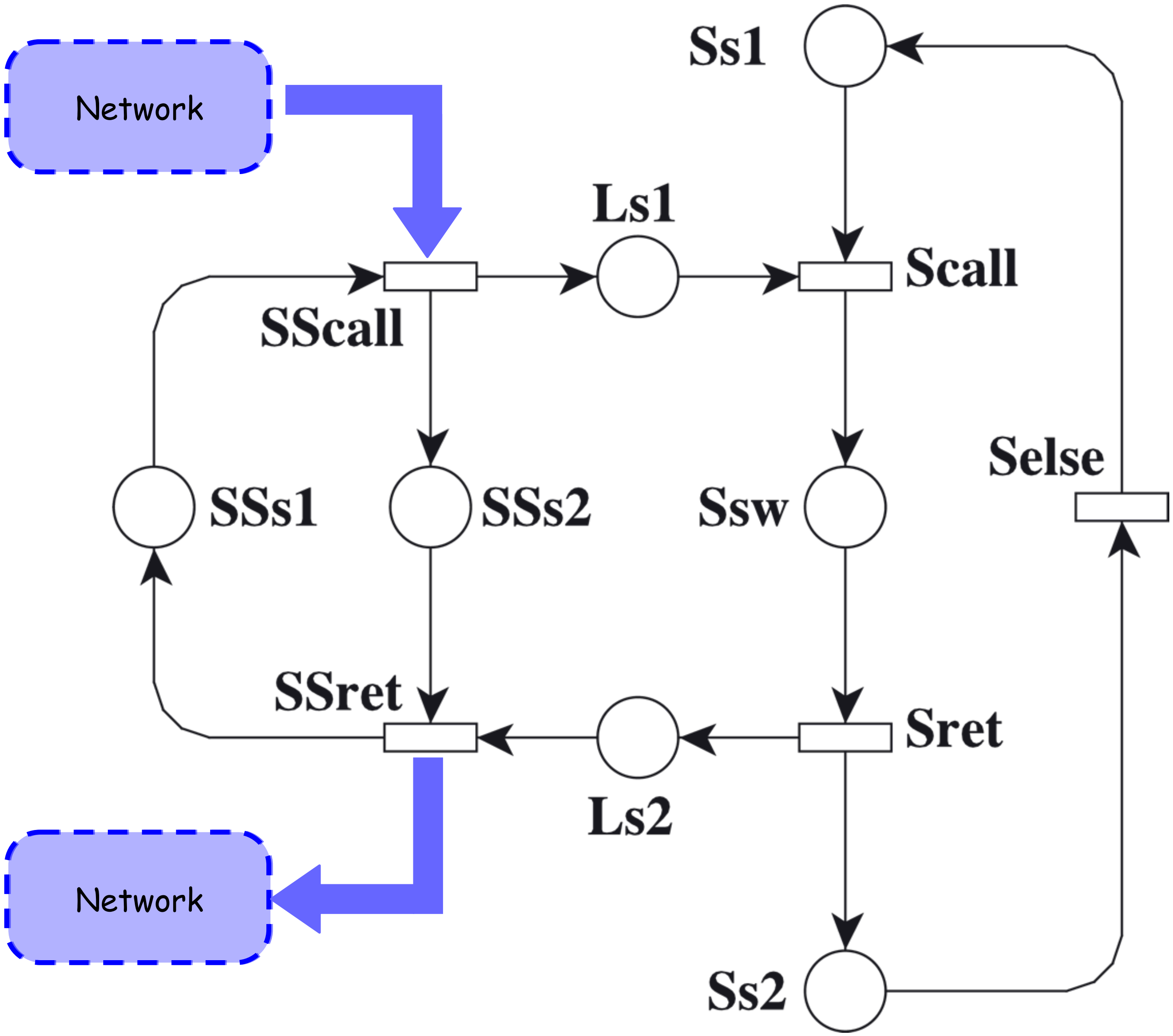
Modeling and assembling the client side



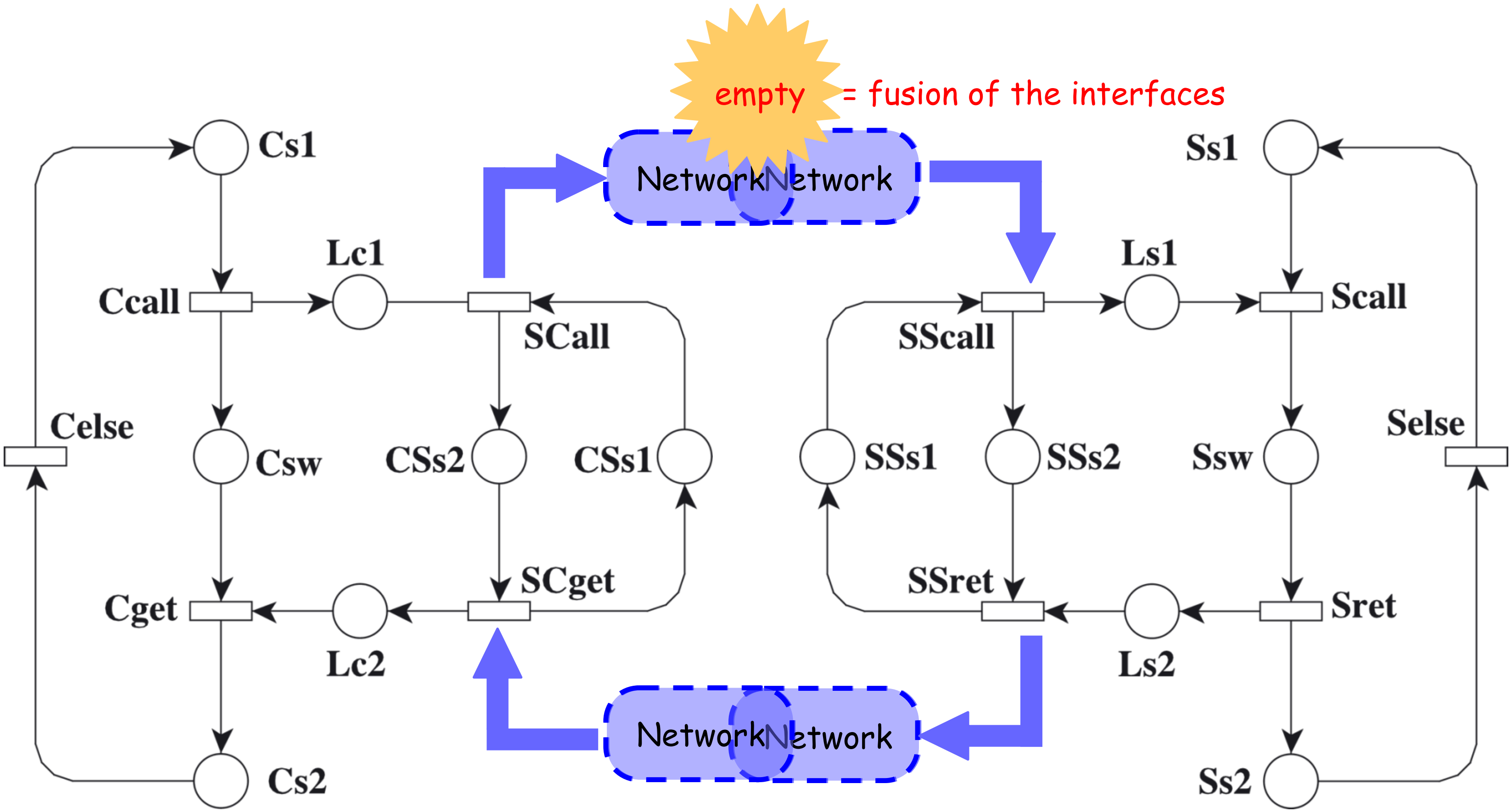
Client side: assembled



Server side (same approach)

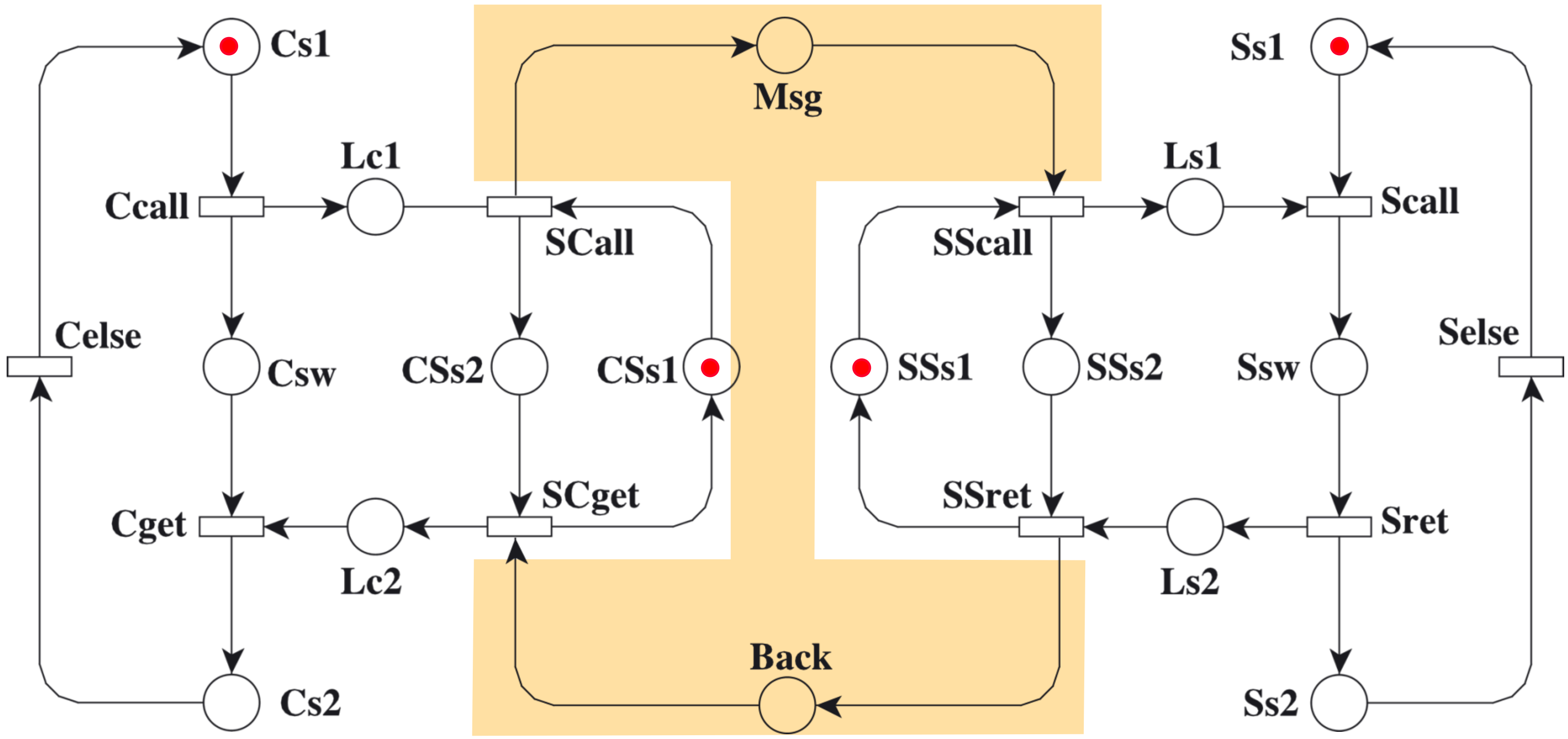


Assembling (higher level)



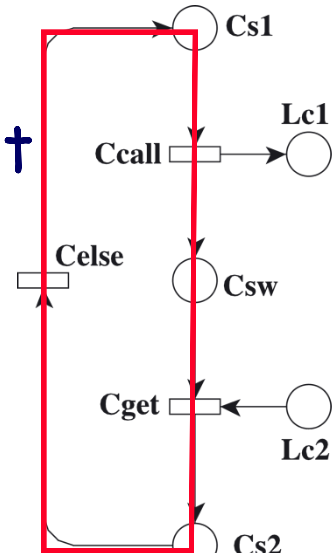
Assembling (higher level)

The network

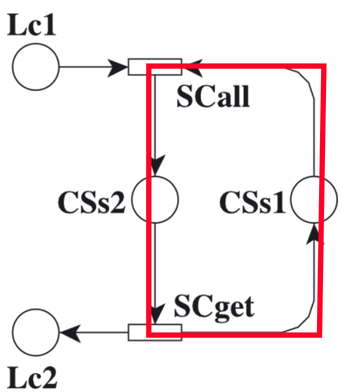


Controls at every stage

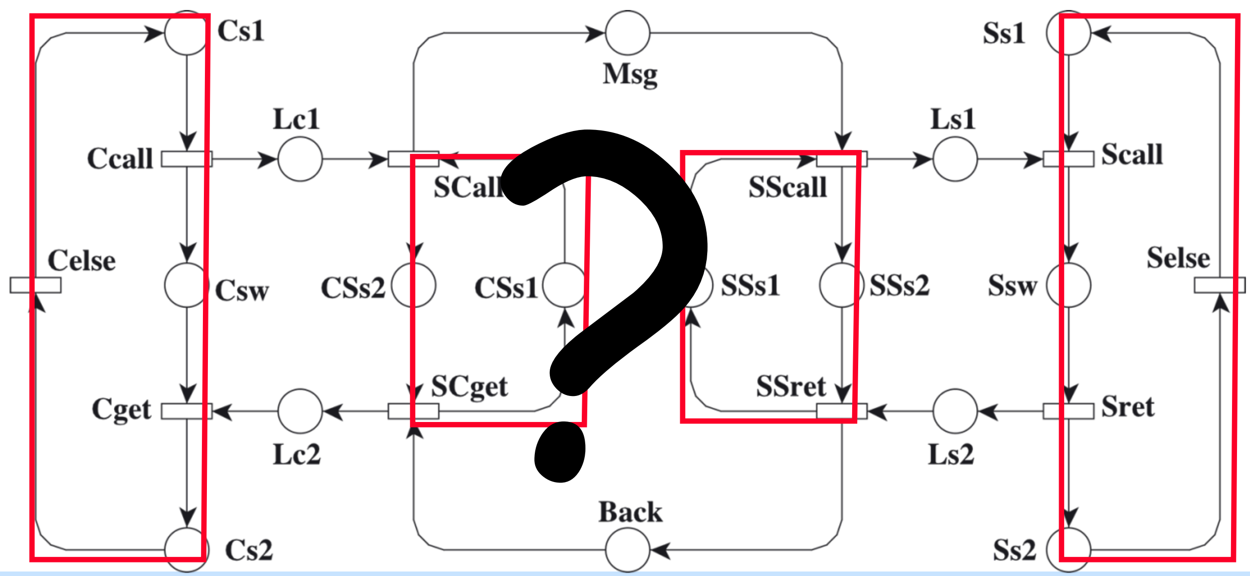
- For the client



- For the client stub



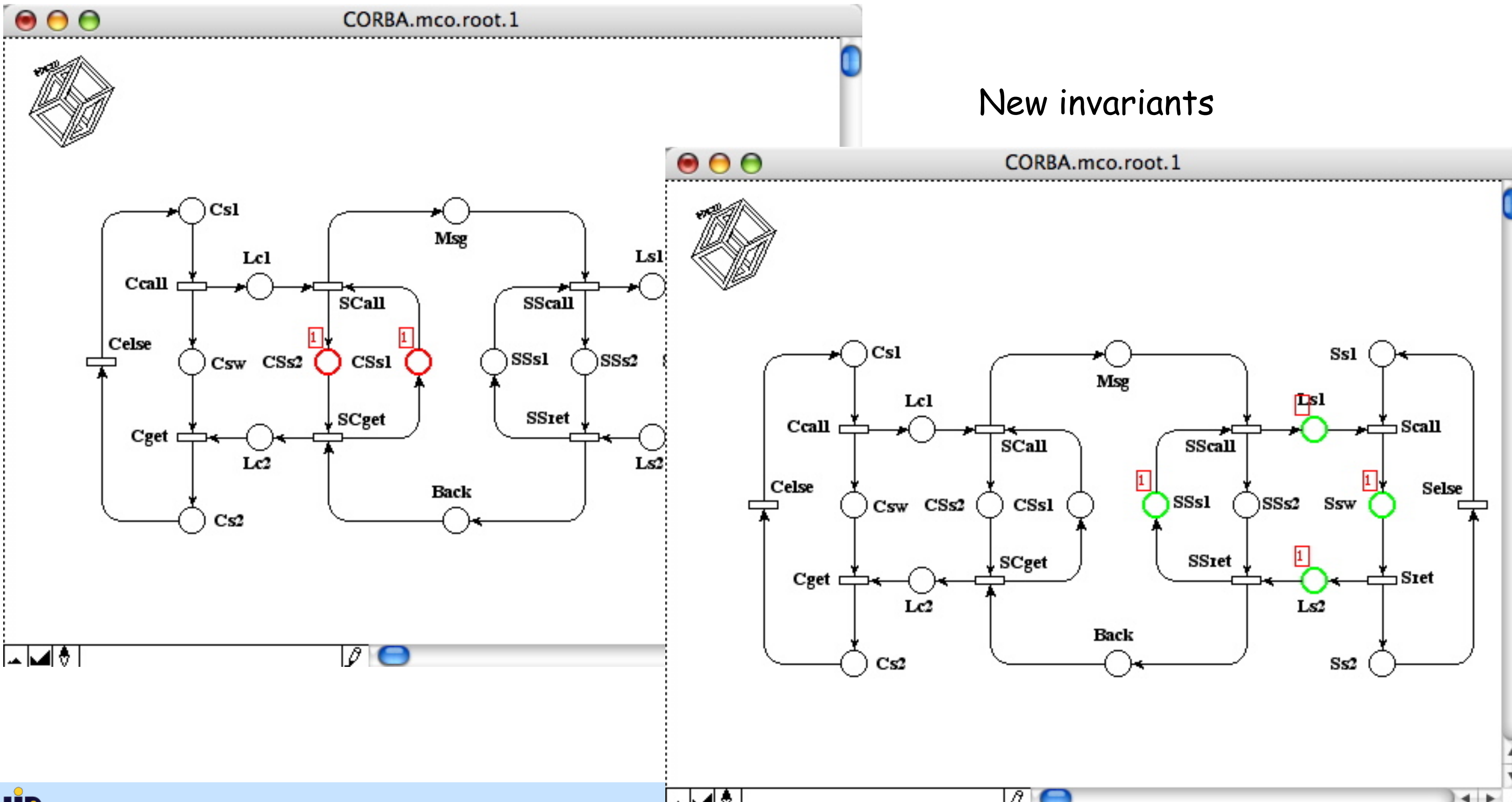
- For assembled Client
  - local communication loop
- For the server and stub and assembled server side
  - As for the client and client stub!
- For the whole system
  - the computed ones
  - and some related to communication



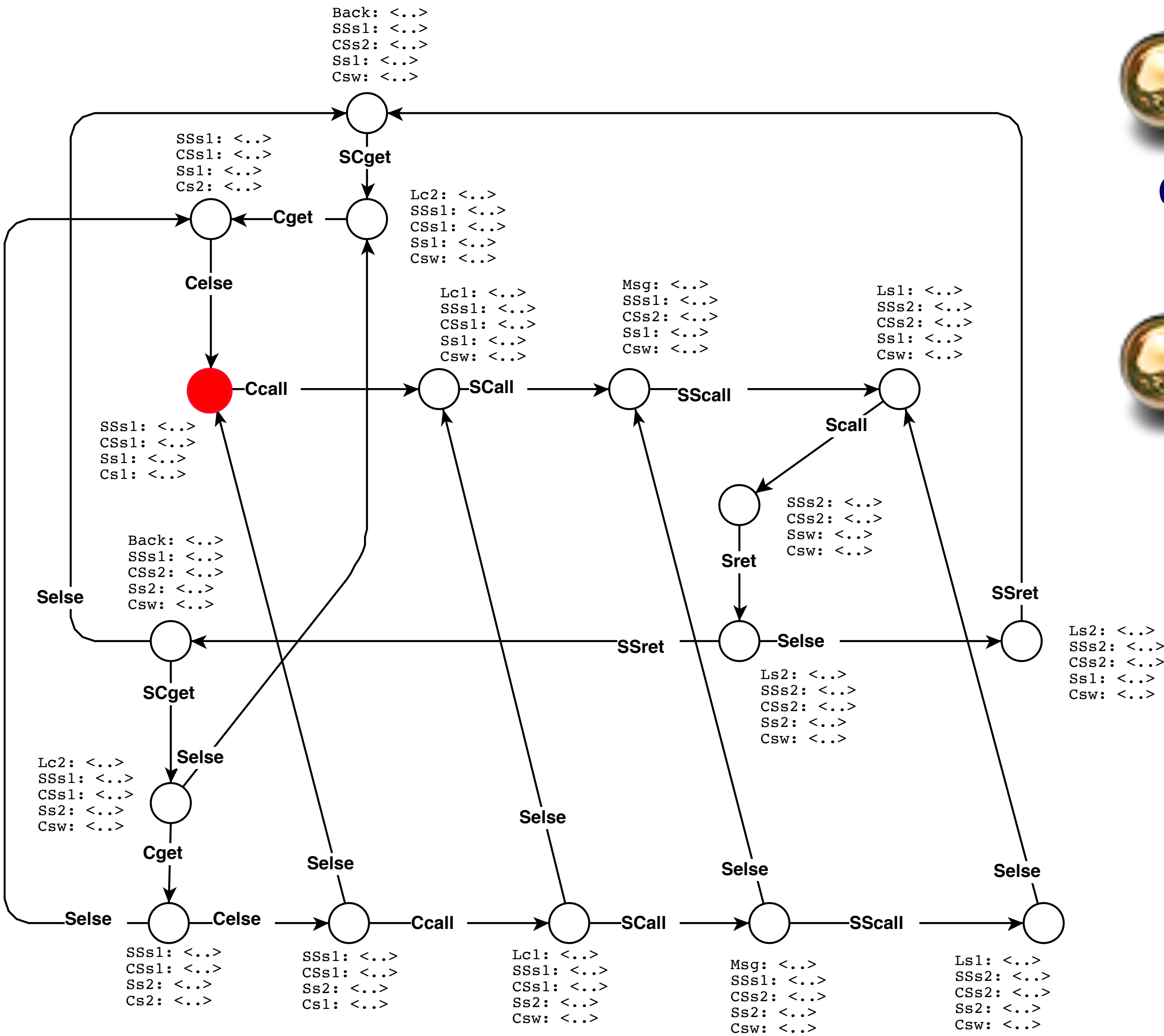
The invariants (from CPN-AMI)

Expected invariants

New invariants

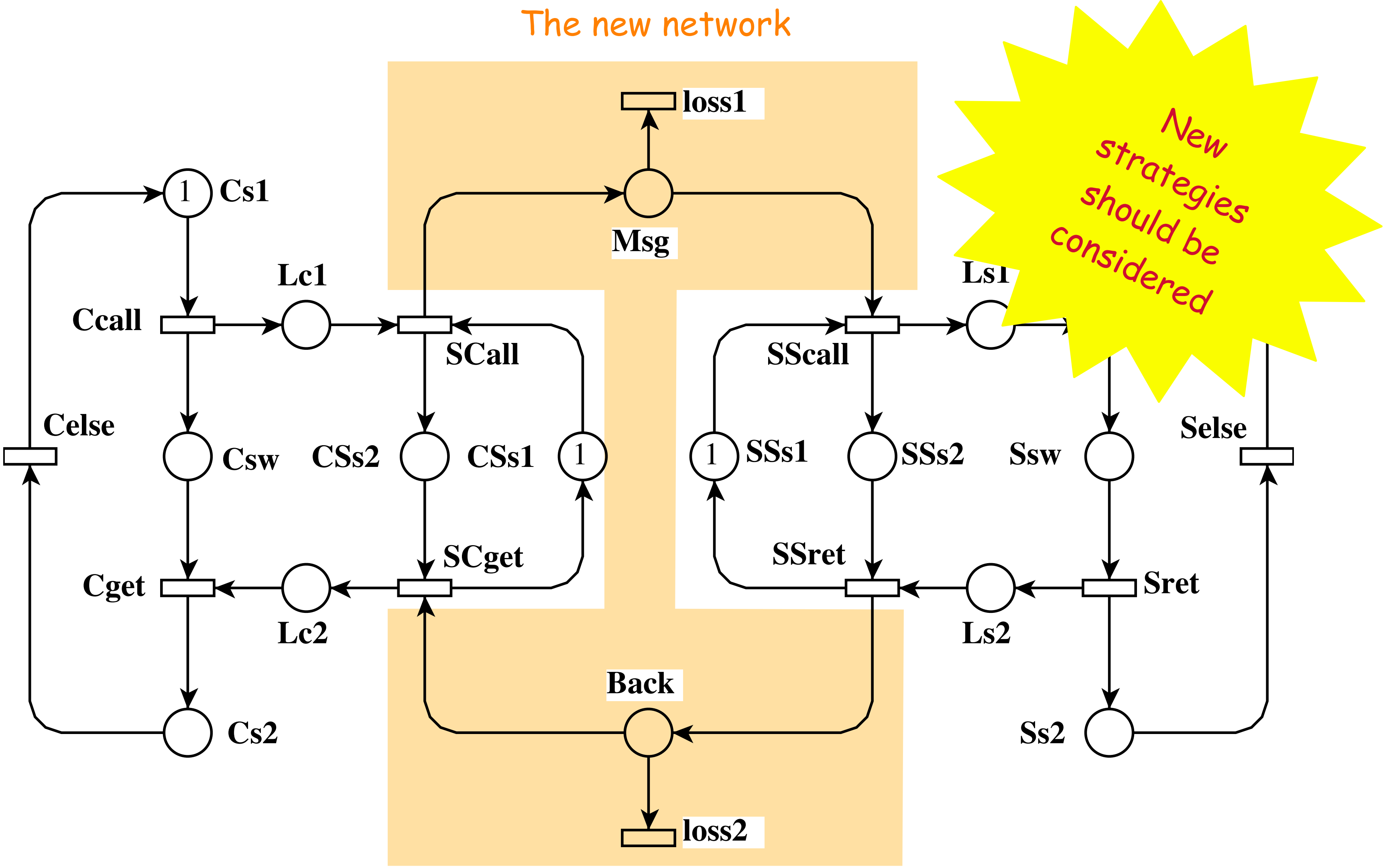


Elements of analysis (from CPN-AMI)



- 17 nodes and 24 arcs
- Good properties
  - No deadlock (loop)
  - Protocol without loss
    - Safe network

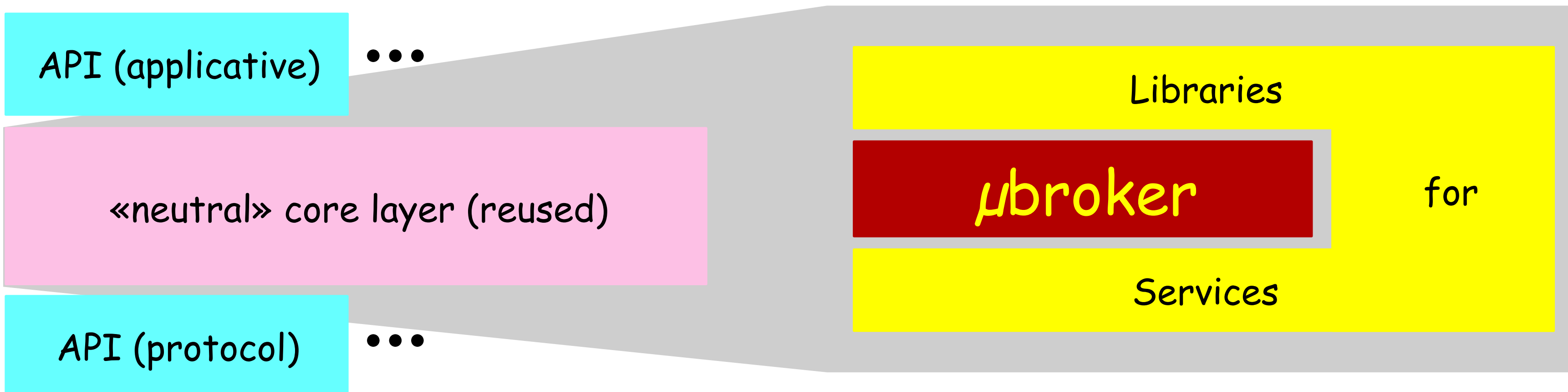
Variation?


The new network
loss1
Cs1
1
Msg
Lc1
Ls1
Ccall
SCall
SScall
Celse
Csw
CSs2
CSs1
1
1
SSs1
SSs2
Ssw
Selse
SCget
SSret
Cget
Lc2
Sret
Ls2
Back
Cs2
Ss2
loss2
New strategies should be considered

An industrial example (verified middleware)

Introduction: what is PolyORB

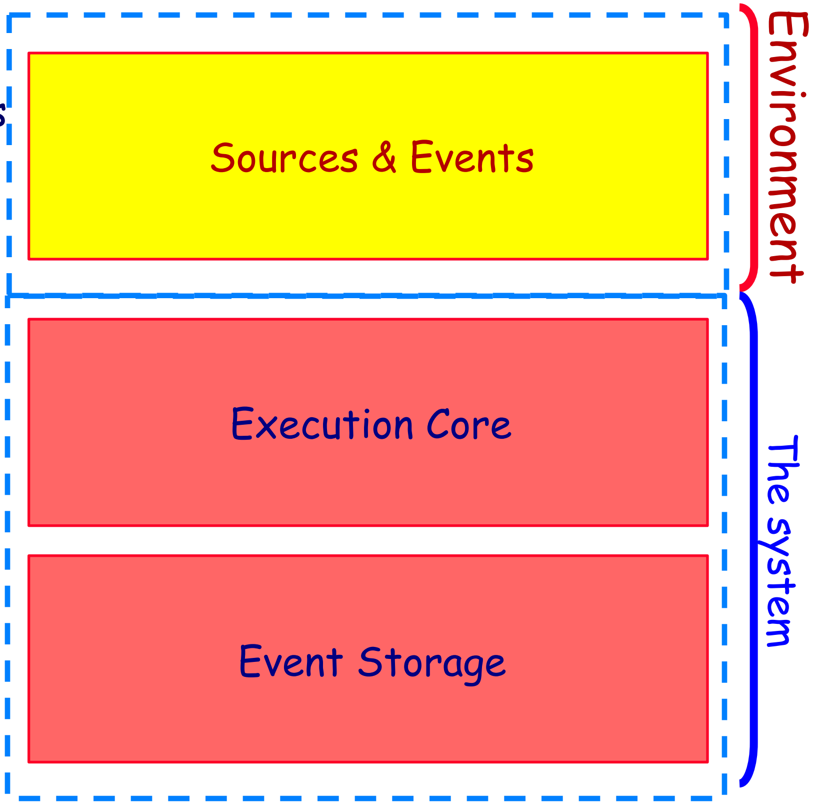
- Schizophrenic middleware
  - Experience gained on a middleware architecture
  - A very generic middleware + can be verified
  - http://www.polyorb.eu.org
- What is PolyORB's global architecture



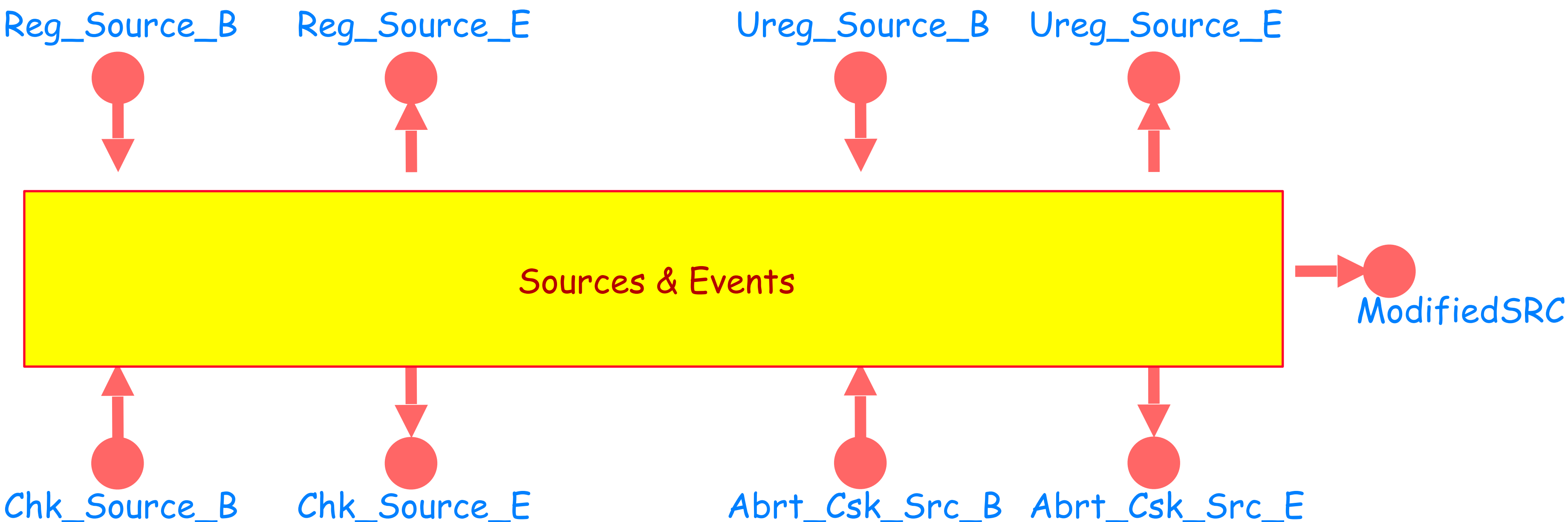
Similar to a scheduler in an operating system

μBroker's structure

- Split the specification
  - Environment: represents identified and required behavior only
  - System: represents the implemented solution according to expected properties
- Environment
  - Behavior, Sources (how many)
  - Events
- System
  - Store incoming events (to be processed)
    - Choice of a store policy (FIFO, priority, etc.)
  - Execution Core
    - Choice of a strategy
      - No tasking
      - Leader/Follower
      - Half-sync/Hald-async,
      - etc.

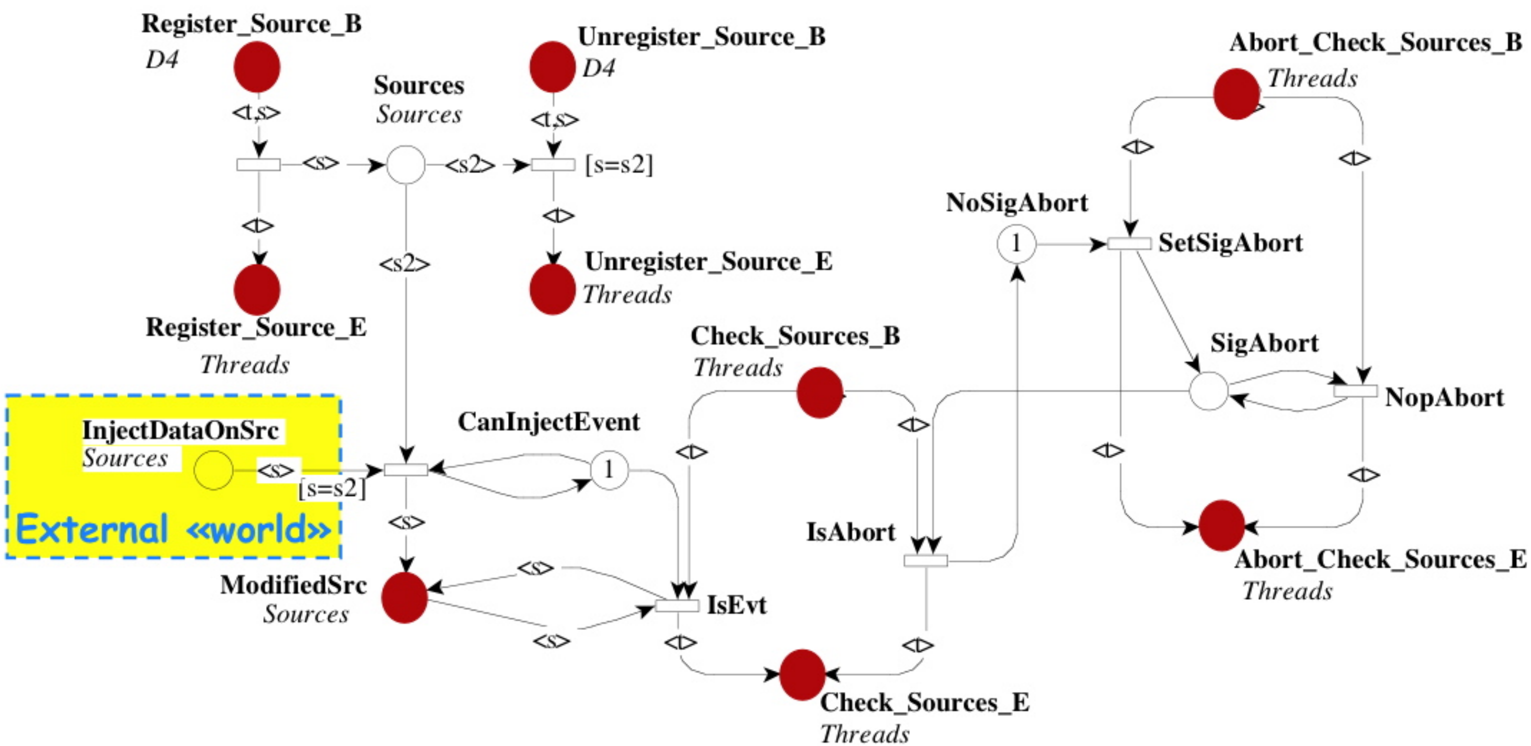


Sources & events : interface



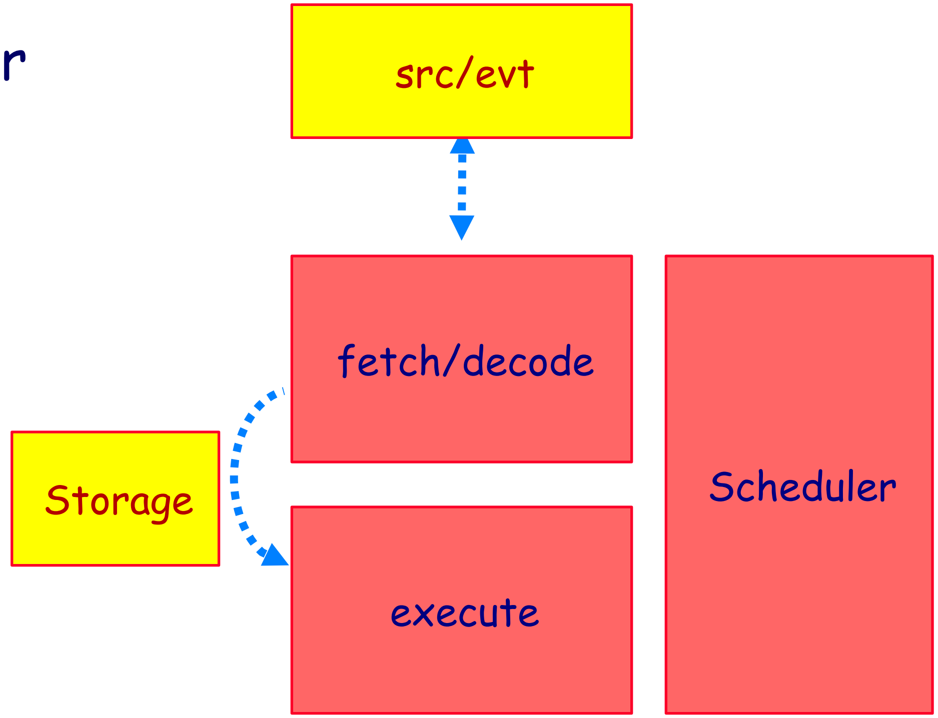
Sources & events: hypotheses and implementation

- Hypotheses:
  - Sources are statically declared (number of sources remains constant in a configuration)
  - Modeling choice: recycling of events in the model

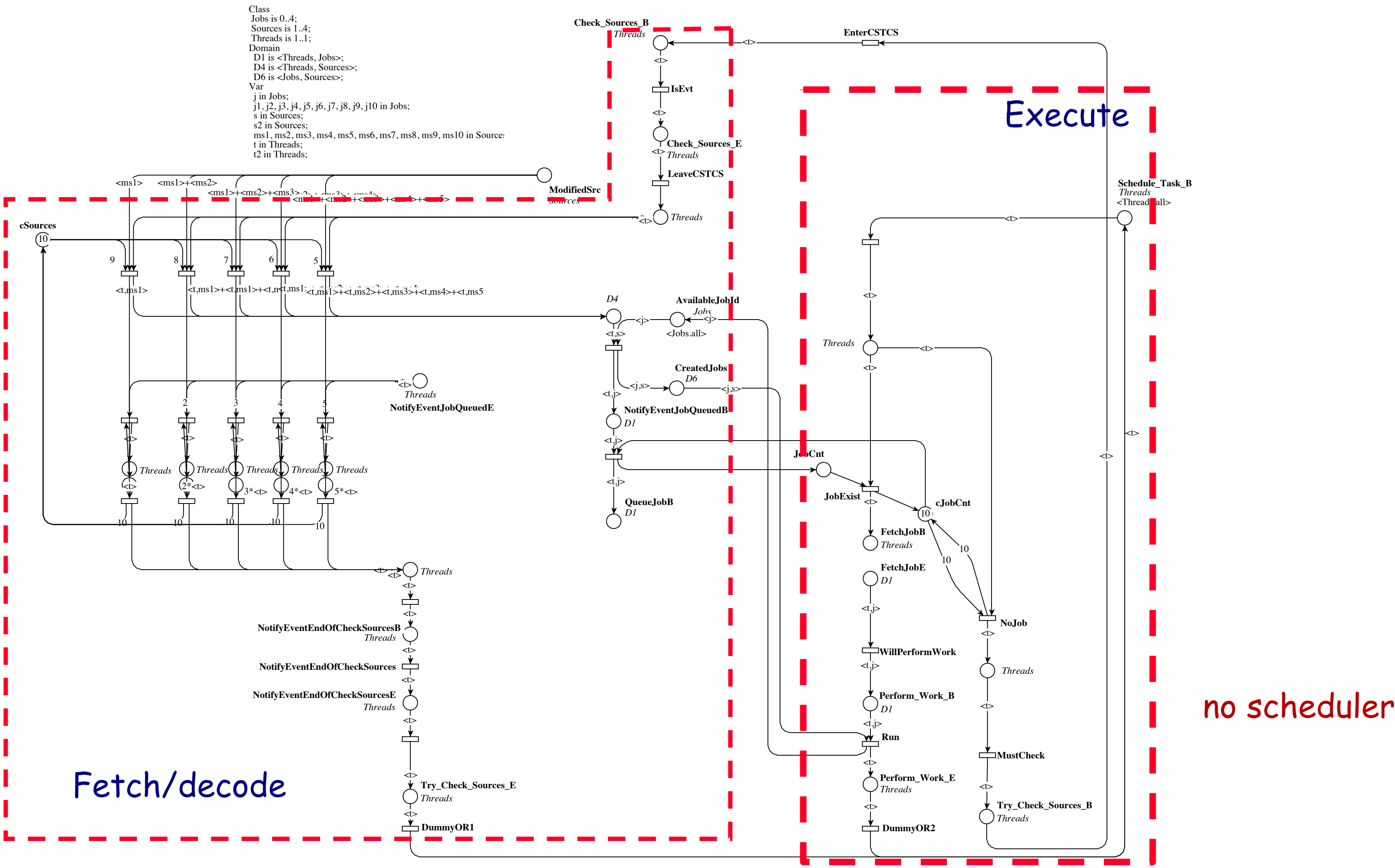


Structuring the System Core

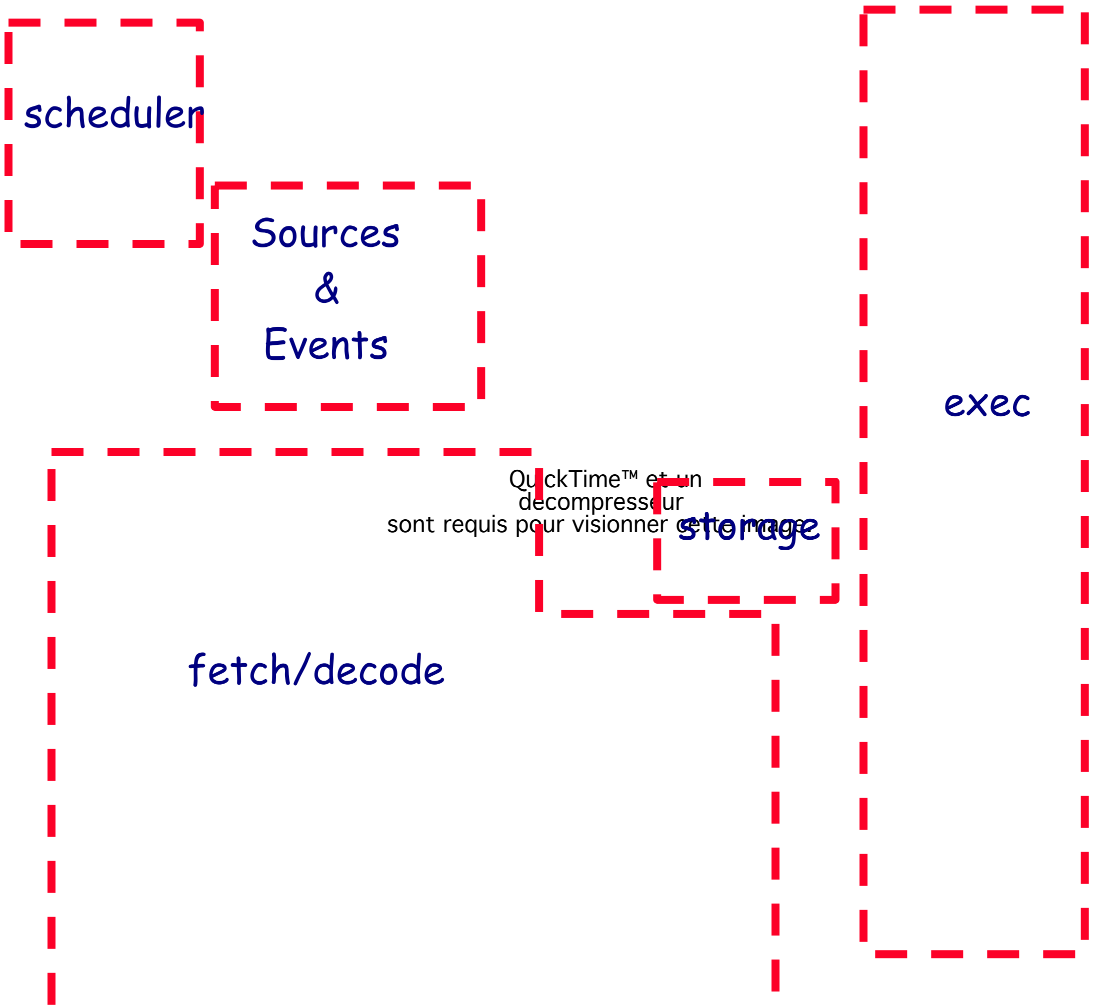
- Dispatching of actions
  - Fetch/Decode and Execute
  - Similar to a micro-processor
- Event storage between the leader thread and the follower ones
  - Using the storage component
- A scheduler must choose the thread to be executed (if multithreaded policy)
- Several possible implementations
  - No tasking
  - Leader Follower
  - others not experienced yet



µBroker, a first model (no-tasking = mono-threaded)



μBroker, a new model (for fun!): FIFO+multithread (leader/follower)



- 89 places
- 72 transitions
- 289 arcs

Parameters

- S_{max}
  - # of sources
- T_{max}
  - # of threads
- B_{size}
  - FIFO size

Properties

- P_0
  - symmetries : threads and sources are not ordered
- P_0 is a preliminary property
  - Enables the use of symmetries and generation of the symbolic reachability graph
- P_1
  - No deadlock: the system never blocks
- P_2
  - FIFO management: no possible attempt to insert an event twice in the same FIFO slot
- P_3
  - No starvation: Any incoming event will be processed

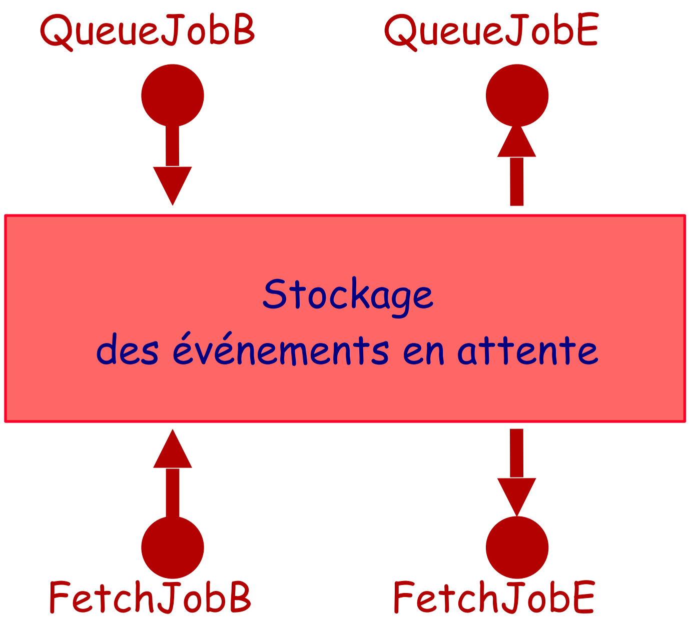
Such a model can be analyzed with appropriate tools!!!

AND UNDER APPROPRIATE CONDITIONS

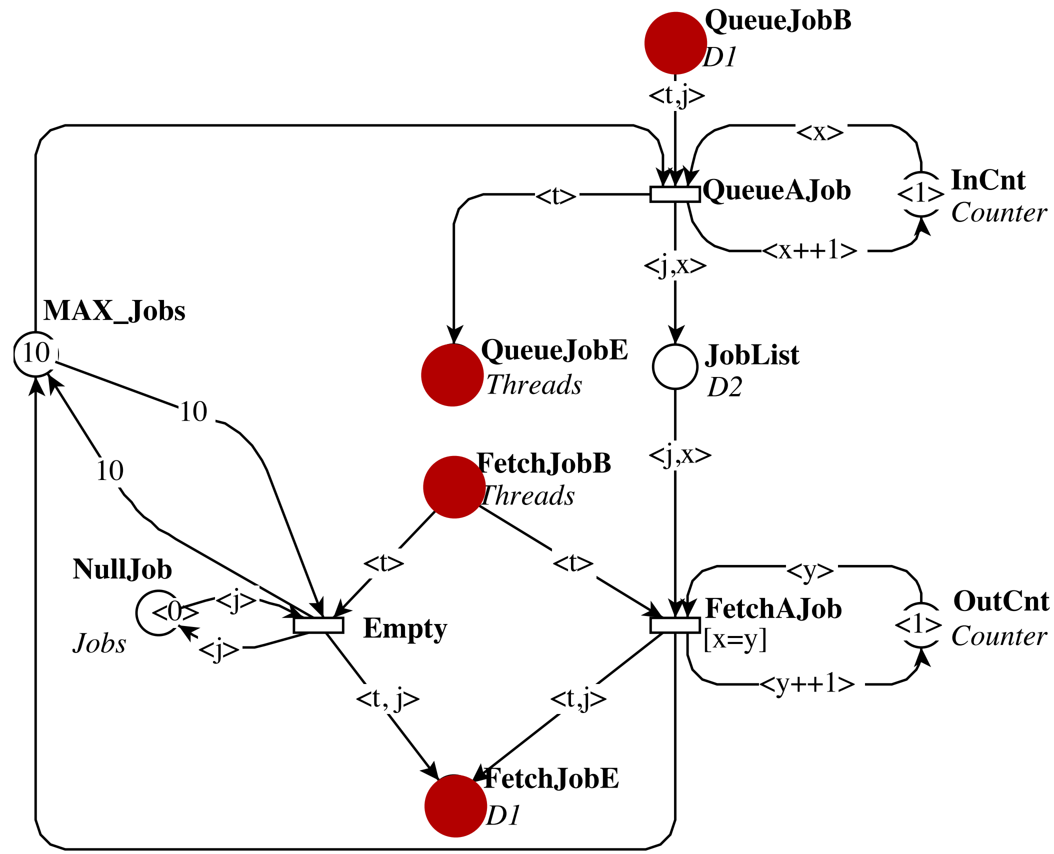
About the appropriate conditions...

First view at the event-storage component

- Component's interface

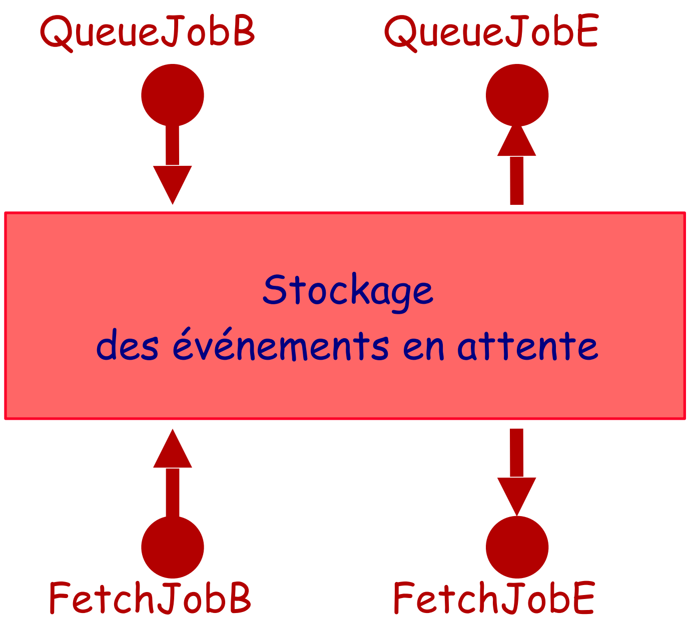


Component's implementation

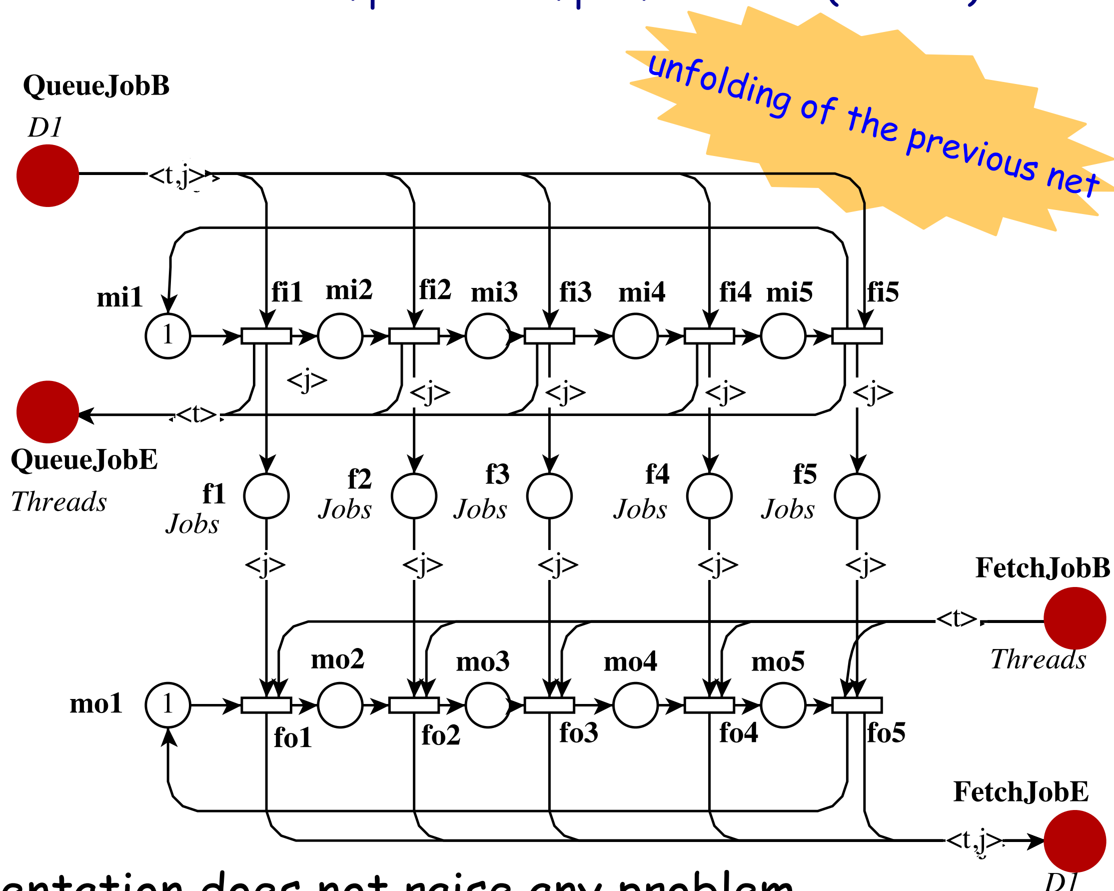


Optimized view at the event-storage component

- Component's interface



Component's implementation (5 slots)

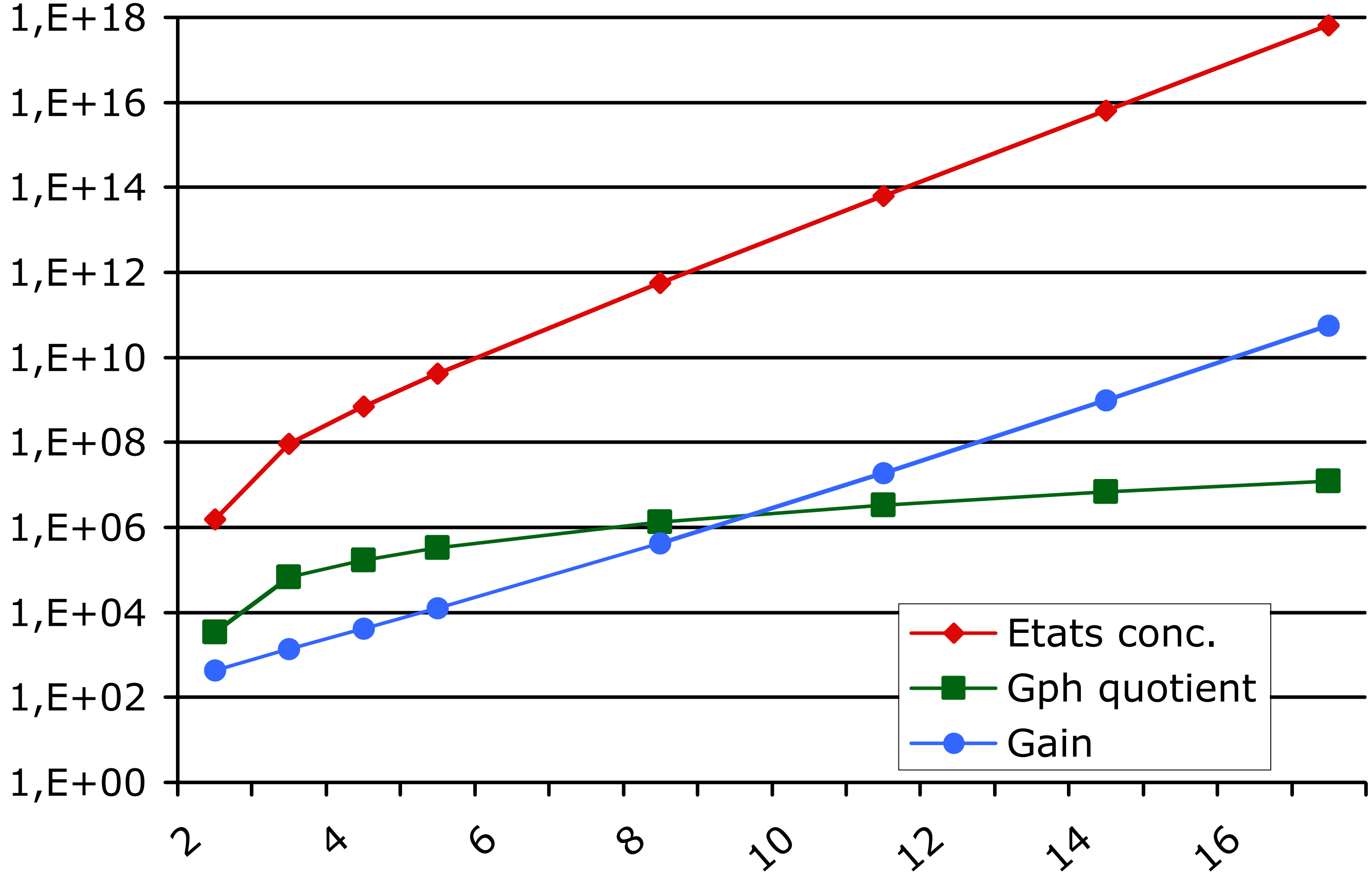


Changing the implementation does not raise any problem

This implementations does not destroy the symmetry (P_0 is verified)

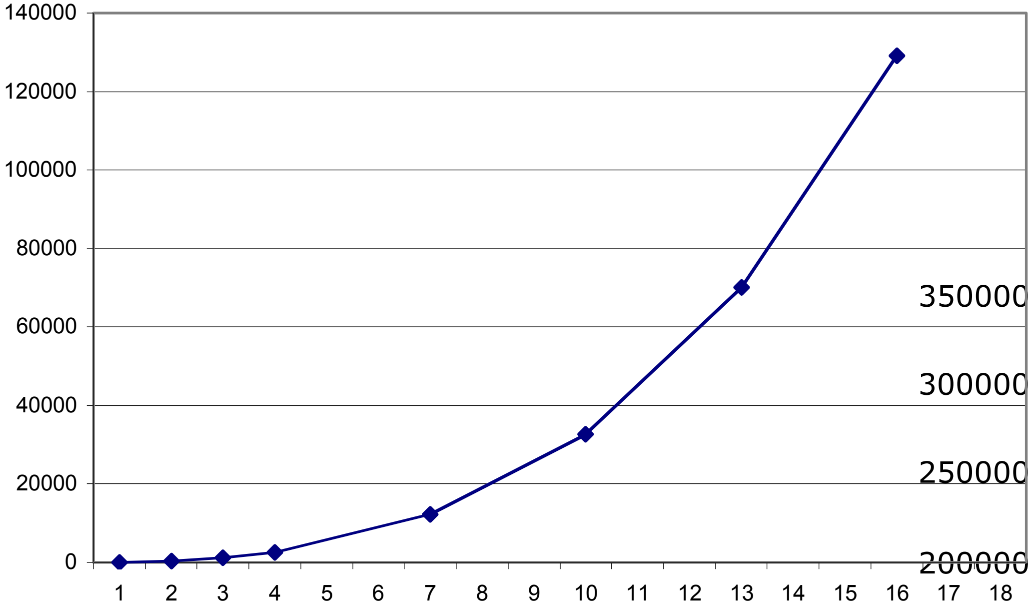
Benchmarks: State space size

$S_{max} = 5$, $B_{size} = 5$, T_{max} Varies

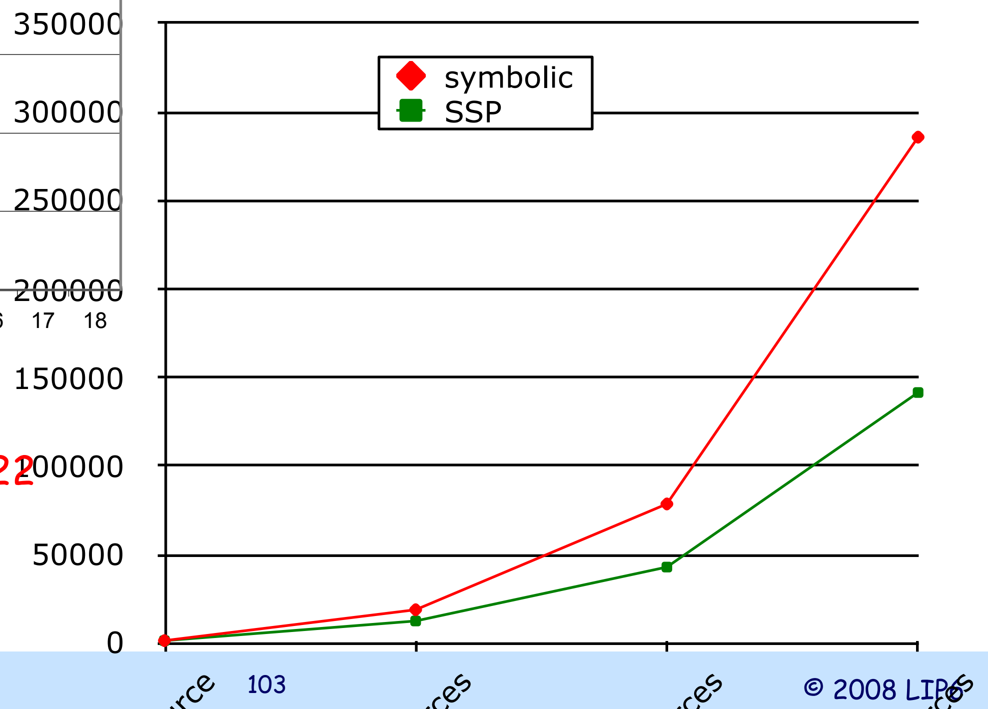


Benchmarks: execution time for

Execution time to produce the full state space (mono-processor)



For P_3, number of visited states (due to an asymmetry)



Experiences in parallel model checking (less than one hour for 17 threads on a 22 bi-processor nodes cluster)

Some conclusions and perspectives

Conclusion

- It is possible to use Petri Nets for the verification of very complex systems
  - This was performed using CPN-AMI («around version 3.0»)
  - But everything was done «by hand»
- There is a need for appropriate tools if ones want to manage large specifications
  - Usable by engineers
  - Connected to standards?
    - Is UML OK? How to make it usable?
    - Already experienced: Torino, Hamburg, etc.
    - LfP : an UML profile (RNTL-MORSE project)

http://www.lip6.fr/cpn-ami

- So far what has been introduced in CPN-AMI
  - PetriScript: a language to generate Petri Nets
    - Constructors
    - Operators (merge, fusion, manipulation of sets of places or transitions)
  - New optimization techniques for model checking
    - Use of the Petri Net's structure (the SPIN community has a similar strategy)
    - Use of new compact representations...

Perspectives

- The industry is interested
  - Critical systems
- There is a need to manage time and/or performances too
  - Even for distributed systems
- Relation to implementation
  - Possible is specific cases (such as PolyORB)
  - However, this is a challenge (MDA, Prototyping)
- New experiences to be done with the new developed tools
  - More with PolyORB
    - Verification of a given configuration
    - Integration in the Production process
  - Intelligent Transports Systems
    - Validate strategies at an early stage of the design