# Petri Nets

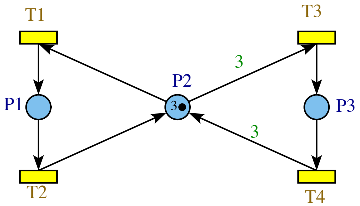
David Delfieu

October 28, 2018

# Plan du cours

1. Petri Nets
   - Basic Definitions
   - Caracteristics
   - Properties which are depending on initial Marking
   - Decidability of properties
   - Properties which are not depending on initial Marking
   - Computing of repetitive and conservative component
   - Reduction technics
2. Coding
   - Introduction to functional language
   - Coding a RdP in LISP/Racket

# example



## Place K-bounded

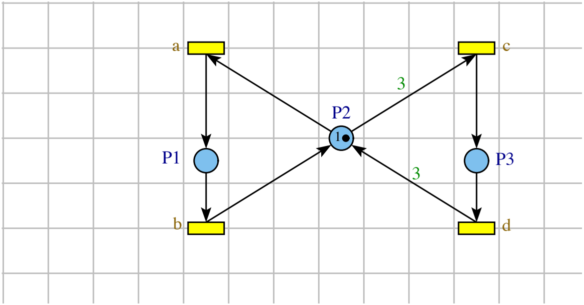
A place $p$ of a Marked Petri Net is k-bounded *iff*

$$\forall M' \in A(R; M_0) \quad M'(p) \leq k$$

If $k = 1$ the place is called “binary”. In the PN of the following figure $M_0(p_2) = 3$, $p_3$ is binary while $p_1$ and $p_2$ are 3-bounded.

# example

Let's modify the example :




With that Initial Marking , the PN is 1-bounded.

# K-bounded and binary PN

## PN K-bounded

A Marked Petri Net is K-bounded *iff* every places are K-bounded. A Marked Petri Net is is binary *iff* avery places are binary. The PN is called Safe.

In the previous figure, with an Initial Marking of $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ the PN is safe, *c* and *d* are be never fired anymore.

# Examples

Each time the sequence $s = a; b$ is fired, a token is added in $p_3$. This place is unbounded thus the PN is unbounded as well.

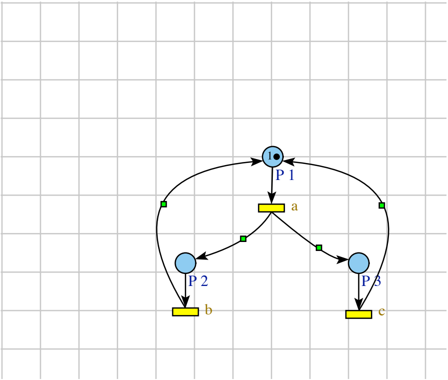



Figure: Unbounded PN

# Quasi-Liveness

## Quasi-alive Transition

A transition $t$ of a Marked Petri Net is quasi-alive *iff* it exists a firing sequence $s$ such as:

$$M_0 \xrightarrow{s} M' \text{ et } M' \xrightarrow{t}$$

Notation: $M_0 \xrightarrow{s;t}$

## Alive transition

A transition $t$ of a Marked Petri Net $N$ is alive *iff*

$$\forall M' \in A(R; M_0) \; \exists s \; M' \xrightarrow{s;t}$$

# Examples of Liveness

The transition *d* of the following figure is quasi-alive but not alive : *d* is reachable from the Initial Marking . But once it has been fired, it cannot be fired anymore.

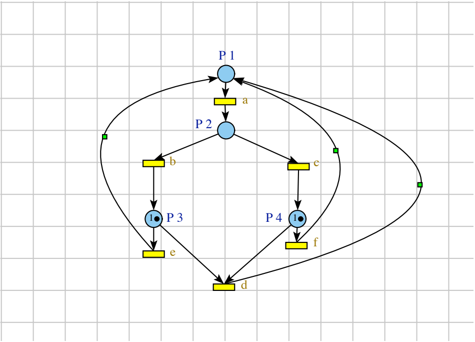
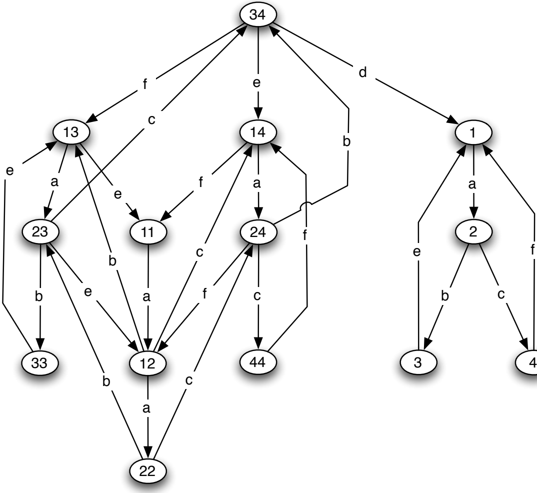



Figure: Quasi-alive transition and unlive transition

# Examples of Liveness

- *a*, *b*, *c*, *e*, *f* alive
- *d* quasi-alive



# Strong connexity

### Strong connex componant

$G_1$ is a *CFC* of the $G$ *iff* $\forall (s_0, s_1) \in G_1, \exists$ a path from $s_0$ to $s_1$

- Previous example : The entire figure is not strongly connected. After the firing of *d*, the control cannot go back in the left part.
- Suppress arc d : Two subfigures strongly connected.

# Alive Marked Petri Net

## Alive Marked Petri Net

A Marked Petri Net $N = < R, M_0 >$ is alive *iff* every transitions are alive.

## Remark

- An Alive PN guaranties that no blocking can be imputed to the Marked Petri Net
- An alive PN guaranties that none parties are dead.

# Example of alive Marked Petri Net

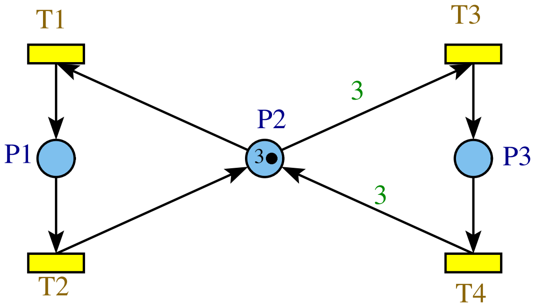



Figure: Alive Marked Petri Net

# Example of unlive PN

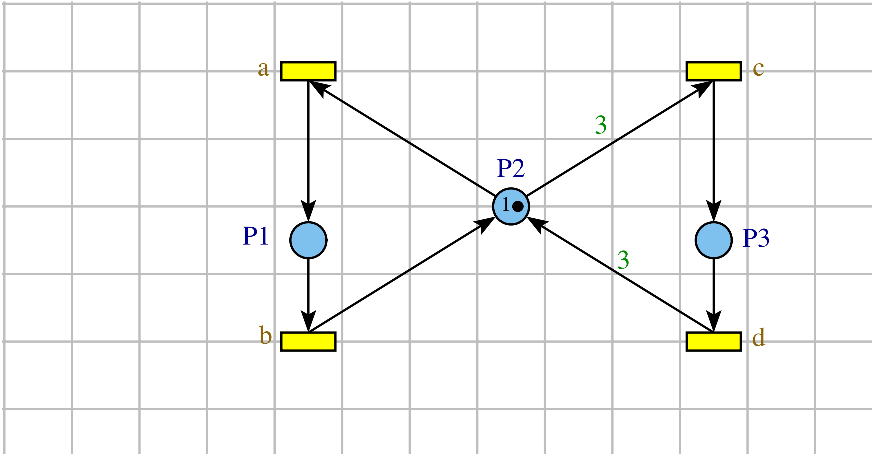
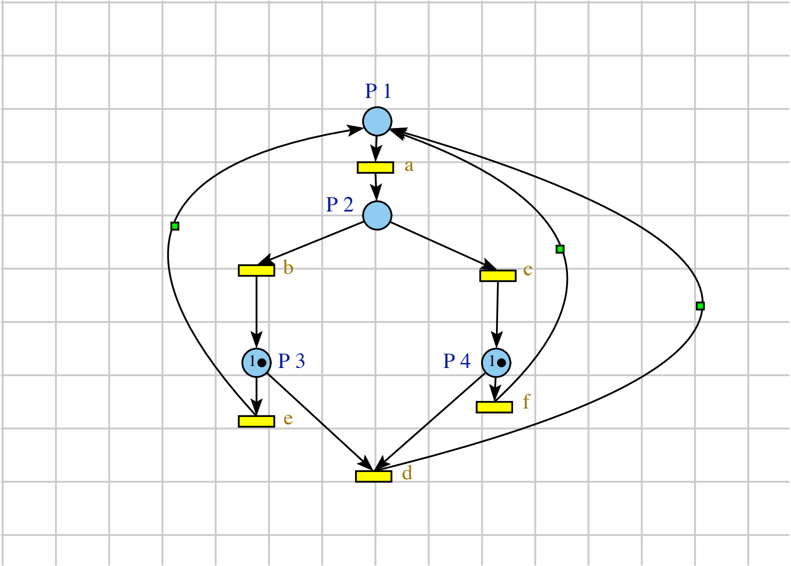



Figure: unlive PN

# Liveness and boundedness

- A PN can be unbounded and Alive.
- A alive PN gauranties that nones parts are dead.

# Example of unbouned alive PN



# Reinitialisability PN

## Reinitialisability PN

A marked PN $N =< R, M_0 >$ is Reinitialisability *iff* its marking graph is strongly connex:

$$\forall M' \in A(R; M_0)\ \exists s\ M' \xrightarrow{s} M_0$$

# Example of PN Reinitialisability

- Strong connexity $\Rightarrow$ every transition is firable $\Rightarrow$ PN is then Alive.
- if $M_0 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$ then PN is Alive AND Reinitialisability.

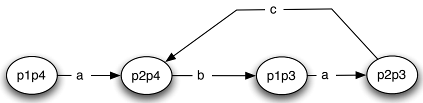



Figure: non Reinitialisability PN

# Exercices

Let's find the good properties in the following exercices:

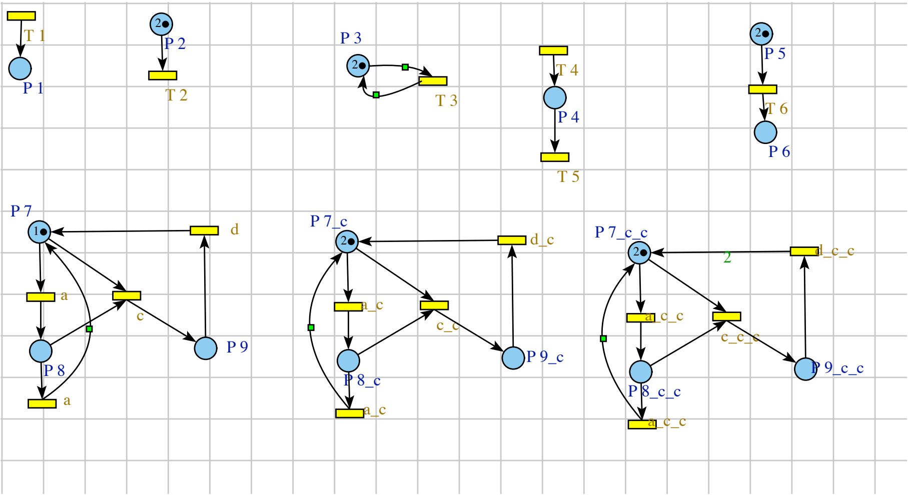



Figure: Non Reinitialisability PN

# Conclusion

The presented properties depend of the Initial Marking . In the next section we focuss on the properties that are not dependant of the initial marking.

# Conservatives componants and place invariants

$M(p_1) + M(p_2) = 1$.
*a* and *b* let this sum unchanged.

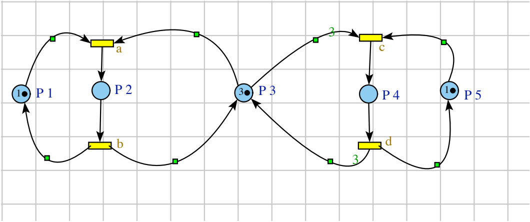



Figure: Conservative componant

- $\forall M \in A(R, M_0) \quad M(p_1) + M(p_2) = 1$
- $\forall M \in A(R, M_0) \quad M(p_1) + M(p_2) = M_0(p_1) + M_0(p_2)$

# Conservative componants and place invariants

- The linear form $M(p_1) + M(p_2)$ : *linear invariant of places*.
- Region $(p_1, p_2)$ $(a, b)$ : Conservative componant.

## linear invariant of place

A linear invariant of place is a linear fonction of the marking places and for which “only the value” depends of Initial Marking .

## Conservative Component

An equality binding two linear relations of the marking places defines a region called a conservative component. It does not depend of Initial Marking .

# Exercice

Exercice : Find intuitively the place invariants ans the conservative componants of

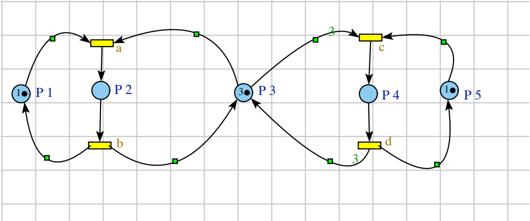



Figure: Place Invariant

# Solution

- $M(p_2) + M(p_3) + 3M(p_4) = 3$ Linear Invariant
- $M(p_2) + M(p_3) + 3 * M(p_4) = M_0(p_2) + M_0(p_3) + 3 * M_0(p_4)$
  - Defines a conservative component
  - The region that contains $p_2, p_3, p_4, c, a$

# Computing the linear invariants of places

- Let's Consider the fundamental equation of a PN .
- Let consider a column vector $f^T$ of dimension $\mathcal{P}$
- Let's Multiply the fundamental equation by the vector $f^T$ :

$$f^T.M' = f^T.M + f^T.C.\overline{s}$$

# Computing the linear invariants of places

## Conservative componant

A Conservative Componant (*CC*) of a PN is a solution of the equation $f^T.C = 0$

- A *CC* defines a sub Petri net from which the places corrresponds to the non null componants of $f$ with the transitions of the initial PN .
- If $f^T.C = 0$ we have $f^T.M' = f^T.M_0 \; \forall \; M \in A(R, M_0)$. The solutions of the equation are the linear invariant of places.

# Invariant of place

If $f^T.C = 0$, consequently, we have :

$$f^T.M' = f^T.M_0 \ \forall \ M \in A(R, M_0)$$

This equation is the corresponding place invariant.
Remarks :

- The expression of a place invariant depends of the Initial Marking .
- A Conservative Componant ($CC$) of a $PN$ defines a conservative subnet whatever the Initial Marking .

# Positive $CC$

Only positives solutions are of interest :

$$f^T.C = 0$$

with $f > 0$.

Moreover, as $C$ is a integer matrix the solutions of the equations will rationnal and we will be able to bring back those rationals to integer numbers.

# Linear invariant of transitions and repetitives stationnary componant

* $c; d$ : unchanged
* $(c, d) + (P_3, P_4, P_5)$
* $(cd)^n$ : unchanged

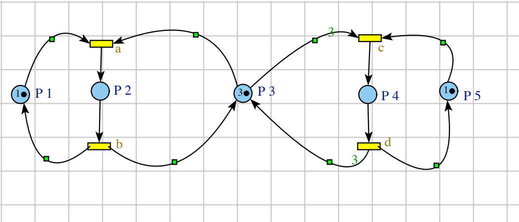



Figure: Linear invariant of transitions

# Invariant of transitions

## Invariant of transitions

An invariant of transitions is a firing sequence of transitions effectively firable from Initial Marking but that does not modify the marking of the net.

Remarks :

- This sequence must be effectively reachable from the Initial Marking .
- This sequence can be fired an arbitrary number of time.

# Stationary repetitive component

## Stationary repetitive component

A Stationary repetitive component is a solution $\overline{s}$ of the equation $C.\overline{s} = 0$.

If $C.\overline{s} = 0$ that's mean that the firing of s is neutral for the net.

# Remarks: Stationary repetitive component

Remarks:

- We will restrain to the positive solutions of this equation : $C.\overline{s} = 0$ with $\overline{s} > 0$.
- A negative value would correspond to a backward firing of a transition.
- The subnet that is defined from the transitions belonging to a sequence which is an invariant of transition.
- This subnet (region) contains the transitions of the sequence, with every places connected to the transitions.

# Summary

## Conservative componant (CC)

A $CC$ is a solution of $f^T.C = 0$ that defines a region independantly of the Initial Marking .

## Invariant of places

- Let's $f$ a vector, then $f^T.M = f^T.M_0$ is an invariant of places.
- The value $f^T.M_0$ depends obviously of the Initial Marking .

# Summary (suite)

## Stationary repetitive component

A Stationary repetitive component is a solution $\overline{s}$ of the equation $C.\overline{s} = 0$. This is a subnet of the PN initial. it does not depend of the Initial Marking .

## Invariant of transitions

This is a sequence of transition $s$ such as $C.\overline{s} = 0$ with $\overline{s} > 0$. it depends of the Initial Marking because $s$ must be effectively firable from Initial Marking .

# K-bounded Petri Net

Remarks :

- A PN is k-bounded *iff* if it possesses a finite marking graph.
- If the marking graph is not finite, it exists at least a sequence which is not stationary repetitive.

# Properties allowing to establish the decidability of the Boundedness

The decidability of the Boundedness results of the following properties:

## Monotony

$\exists M, M', s$ such as $M \xrightarrow{s}$ and $(M' > M) \implies (M' \xrightarrow{s})$

## Non K-bounded Petri Net

If $\exists M, M' \in A(R; M)$ such as $M \xrightarrow{s} M'$ and $M' > M$ then the PN is not K-bounded.

# Properties allowing to establish the decidability of the Boundedness

## Lemma of Karp and Miller

Every infinite series of vectors constitued of positive or null integers :

$$v_1, v_2, ....v_k, ...$$

is such as it exists at least 2 where :

$$v_i, v_j \text{ with } i < j \text{ such as } v_i < v_j \text{ where } v_i = v_j$$

# Properties allowing to establishthe decidability of the k-bornitude

## An increasing monotone infinite serie of markings

1. Let $s$ an infinite sequence of firings of transitions. The serie of markings is a serie of vectors of positive or null integers.
2. The previous lemma implies then the existence of a sous-sequence $s$ such as : $M \xrightarrow{s} M'$ and $M \leq M'$

The sequences defined at point 2 are caracteristics of non stationnary sequences of infinite lenght. Its follows that the net is not bounded.

# Algorithm

The Algorithm is based on the computing of the marking graph.

1. The starting point is the Initial Marking :
   - For every firable transition the new marking is computed.
   - Then we iterate on the set of new markings.
2. Stop condition:
   a. We found an already found marking, the current branch in way of exploration is stopped.
   b. We found a marking strictly greater than a marking already found. $\Rightarrow$ The algorithm is totally stopped.

a. If the Algorithm is ended by the stop condition *a*, the net is k-bounded. So, it hasn't got any infinite branch.

b. If the Algorithm is ended by the stop condition *b* then the net is unbounded.

# Terminaison of the algorithm

The algorithm ends in any case because:

- We cannot get branch of infinite lenght without encounter a marking that is strictly greater to an already found marking: Karp and Miller lemma
- The graph is finite since :
  - the number of arcs is finite (T is of finite dimension)
  - the number of markings is finite (stop of the algo).

# Decidability of Properties

- The Boundedness is decidable for a Marked Petri Net
- Re-initialisable, Alive and quasi-alive are decidable

# The establishing of properties

Prove order of properties :

1. Computation of marking graph $\Rightarrow$ k-bornitude,
2. Reinitialisability,
3. quasi-alive,
4. Alive.

# Relations between the Properties

## Rénitialisable and connexity

The PN $R$ is Reinitialisability for the Initial Marking $M_0 \iff GA(R, M_0)$ strongly connex.

# Sketch of Prove

- Reinitialisability: $\forall M_i, M_j \in A(R, M), \exists$ then a path from $M_i$ to $M_0$.
- In the same manner, $M_j$ is reachable from $M_0$ by the definition of the establishong of the marking graph. Then it exists a path of $M_0$ a $M_j$.
- By transitivity it exists then a path from $M_i$ a $M_j$. This for every pair of existing marking.
- The graph is strongly connex.

# Relations between properties

## Rénitialisable and Liveness

$R$ Reinitialisable for M $\Longrightarrow$ ($R$ quasi-alive $\Longleftrightarrow$ $R$ Alive)

# Sketch of proove

1. $R$ alive $\Longrightarrow$ quasi-alive.
2. Reciprocal:
   - If $R$ is Reinitialisable: From every rechable marking $M_i$, we can go back to $M_0$
   - As it is quasi-alive, considering any transition $t$, it exists at least one path from $M_0$ leading to $M_j$ where $t$ is firable
   - By transitivity for any marking $M_i$, for any transition $t$, we can exhibit a path leading to a marking $M_j$ where the transition $t$ is firable.
   - $R$ is alive

# Liveness

1. If the net is k-bounded, the graph marking can be computed
2. If the graph is strongly connex, it is then Reinitialisable,
3. If for every transition *t*, it can be cheked that every transition *t* is quasi-alive (*t* appears at least one time in the marking graph)
4. The net is Alive.

# Reduction

## Equivalence relative to Good properties

Two nets are equivalent relatively to the good properties (Alive, Reinitialisability, quasi-alive, K-bounded) *iff* the nets verify the same subset of Properties.

Goal : Fibd the reduction rules which preserv this properties.

# Simple cases of substitution of places

Let's $t_e$ and $t_s$ the incomming and outtransitions d'betweene and of sortie.
If this place verifie the conditions suivantes :

- If $Post(p, t_e) = Pre(p, t_s)$ : egalite the weights entrant and sortant
- $\forall p' \in P$ If $p' \neq p$ then $Pre(p', t_s) = 0$ : pas d'another place that $p$ en betweene of $t_s$

Alors this place is substituable.

# Substitution

$p$ can be suppressed by substituing $t_e$ and $t_s$ by the transition $t_{es}$

$$\forall p' \in P \; Pre(p', t_{es}) = Pre(p', t_e)$$

$$\forall p' \in P \; Post(p', t_{es}) = Post(p', t_e) + Post(p', t_s)$$

# Examples : Simple substitutions of places

The transitions *a* and *b* are susbstitued by the transition *ab*

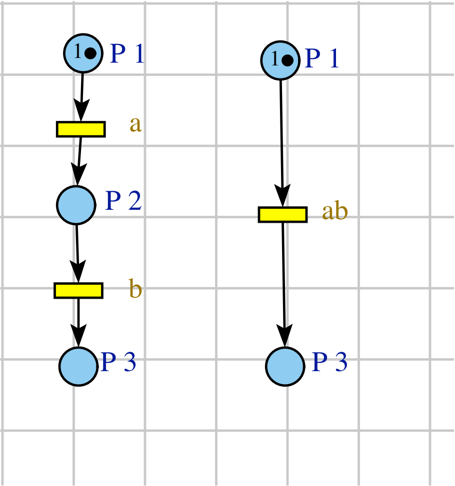



Figure: Simple substitution

# Examples : Simple substitutions of places

This example shows that there is no restriction on the transition *a*. We will note that the place $p_3$ is marked. As this token must necessarily come in $P_5$, in the reduced net, we put it in this place.

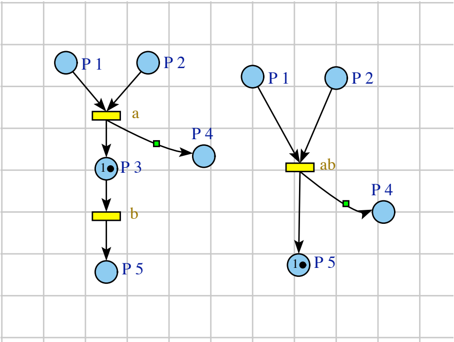



Figure: Simple case of substitution

# Places with several transitions in input and in output

Condition : every weight must be equal.
Moreover, if this place verifies the following conditions:

- The outgoing transitions of $P$ must not have any other input places except $P$
- $\nexists$ an outgoing transition $t_j$ which is in the same time an input transition and an output transition (loop).
- $\exists$ at least an output transition which is not a well transition.
- A well transition does not have any outcomming places.

Then this place is substituable
Remark : There is no condition on the input transition.

# Examples : Restrictions

this may not be the case

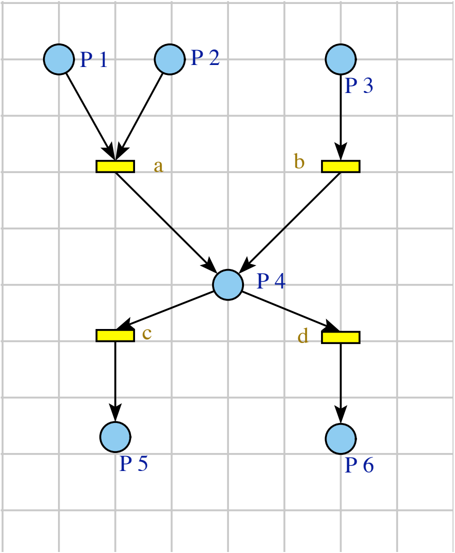
- If the reduction carries on a marked place, the reduced net may not be reinitialisable anymore.
- If in the initial net, an output transition has no output place, there is no more equivalence in the point of vue of the of k-boundedness.
- If in the initial net is K-bounded the value $k$ is lost.

# Examples : complex substitutions of places

Considering every pair formed with an input transition and an output transition, we have the equality of the weights :

$$Post(P_4, a) = Post(P_4, b) =$$

$$= Pre(P_4, c) = Pre(P_4, b)$$



# Examples : Complexes of substitution of places

net réduit

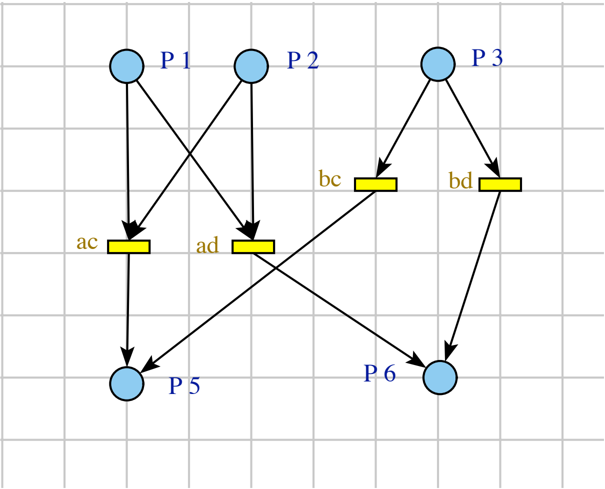



Figure: Complex substitutions

# Implicit places

## Implicit place

- An implicit place is a place from which the marking is a linear combinaison of places.
- Moreover, theses output transitions does not introduce any extra firing condition.

## Degenerated implicit places

If a place is connected only by elementary loops, its marking is constant. And this is an *implicite degenerated* place.

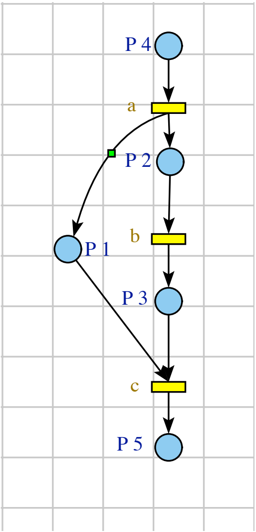
# Implicit places

Remarks :

- An implicit place is quite useless because it does not influate on theses output transitions.
- An implicit place belongs to a concervative componant.

# Exercice

- Let's show that $p_1$ is implicit, by the computing of the C.C. relative to the places $p_2$ and $p_3$.
- Compute the linear invariants of places for $M_0 = (0, 0, 0, 1, 0)$ and $M_0' = (0, 1, 0, 0, 0)$



# Solution

- For $M_0$ we have
  $M(p_1) = M(p_2) + M(p_3)$
- For $M'_0$ we have
  $M(p_1) = M(p_2) + M(p_3) - 1$

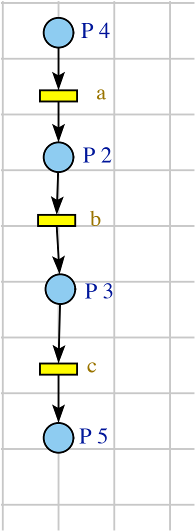
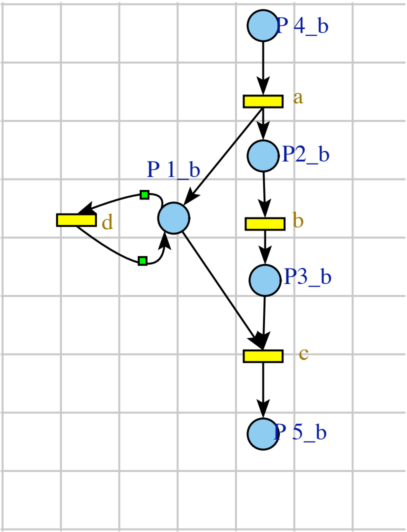



Figure: Implicit place

# Counterexample

In this example, $d$ can occur only between the firing of $a$ and $c$. The suppression of $p_{1b}$ would cancel this property. Firing $d$ implies a condition on the firing of $c$, then the place $p_{1b}$ cannot be implicite.



# Degenerated implicit place

If a place is only connected by elementary loops, this transition is implicit relatively to the empty set, Its marking is constant. Its a degenerated implicit place.

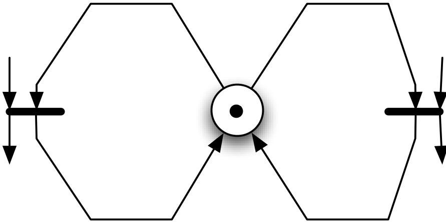


Figure: Degenerated implicit place

# Identical places

## Identical places

Two places $p_1$ and $p_2$ are identical If they have the same input and output transitions.

Remark : $p_1$ is then implicite relatively to $p_2$.

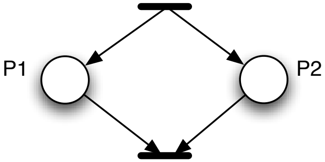



Figure: Identical places

# Neutral transitions

## neutral transition

A transition *t*is neutral (also called an identity transition), if is only connected to the net by elementary loops, such as :

$$Pre(., t) = Post(., t)$$

Remark :
Its firing does not modify the net:

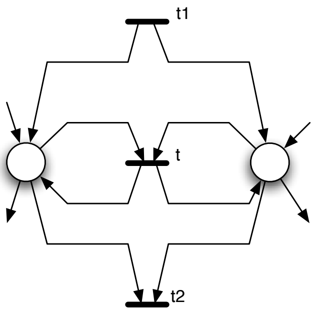



Figure: *t* is neutral

# reduction of a neutral transition

## remarks

To suppress a neutral transition $t$ while respecting the liveness property, one must make sure that $t$ is alive or else it could suppress a dead part of the net:
One could then transform a non alive net in an alive net !

# counterexample of the reduction of a neutral transition

In this figure, *d*is not alive. After the firing of *a*, *d* is no more firable. If one suppress the neutral transition *d* the reduced RdP will be alive.

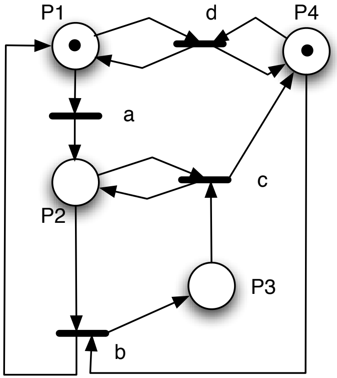



Figure: *d* is neutral goal cannot be suppressed

# reduction of a neutral transition

## Conditions of suppression of a neutral transition $t$

To suppress a neutral transition $t$:

- Either it exists a transition $t_1$ (present in the reduced net) which enable $t$ in the initial net:

$$Post(., t1) = Pre(., t)$$

- Either it exists $t_2$ which carry, in the initial net, the same firing conditions:

$$Pre(., t2) = Pre(., t)$$

# Identical transitions

## Identical transitions

Transitions $t1$ and $t2$ are identical iff:

$$Pre(., t_1) = Pre(., t_2)$$

$$Post(., t_1) = Post(., t_2)$$

One can be suppresses preserving the properties of the RdP.

# Example: Simplification of Identical transitions

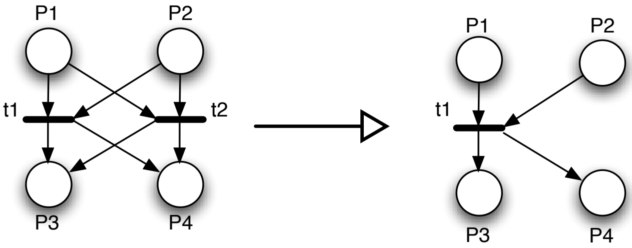



Figure: Suppression of *t2*

# Computing principe

## Computing principe

The principes of the computing are the same for the types of Componants. One looks to resolve : $f^T.C = 0$:

- For the conservative componants: One only has to solve this equation.
- For the repetitive stationnary sequence, one has to apply the method we the transpose of $C$.

# Principe

This equation can be expressed by:

$$C^T.f = 0$$

Considering a net constitued of $n$ places and $m$ transitions then we get a system of $m$ linear equations with a null solution vector:

$$\left.\begin{array}{ccccccc} c_{11}.f1 & + & \dots & + & c_{n1}.f_n & = & 0 \\ \dots & + & \dots & + & \dots & = & 0 \\ c_{1m}.f1 & + & \dots & + & c_{nm}.f_n & = & 0 \end{array}\right\}$$

# Reasonning

$$C^T.f = 0$$

- The set of the solutions form a vectorial space.
- The degenerated solution $f^T = 0$ has no interest.
- The base of the solution vectorial space gives the set of conservative componants of the PN.

# Dimension of the vectorial spaces

## Rank of matrix

The rank of a matrix is the dimension the greater square su-matrix where the determinant is non zero.

Let's $r$ the rank of $C$. The dimension of the vectorial space will be

$$dim_p = n - r$$

The dimension of the vectorial space of the components will be:

$$dim_t = m - r$$

# Gauss method

## Gauss method

It's a systemantic method of resolution for the linear equations. it is based we the fact that the following transformations does not change the solution:

- Exchange of 2 lines.
- Multiplication of a line by a non null scalar.
- Addition of a line to another.

# Variable changing

La Gauss method triangularizes $C$. Fir that one have to make linear combinaisons we lines of $C$. Theses operations will be memorised by change we variables we the vector $f$:

- Multiplication: $f_i$ by $a.f_i$ with $a \neq 0$,
- Addition: $f_i$ by $f_i' + f_j'$
- Exchange of lines: $f_i$ by $f_j'$ and $f_j$ by $f_i'$

# Obtention of the components

- $C$ is not necesserally square. If $r$ is the rank, it exists $S$ of dimension $r$ such as:
  $det(S) \neq 0$ and then the sub-system is formed of independant linear equations with one solution.
- Let's consider $C'$ the result of the resolution.

$$C' = \left[\begin{array}{c|c} S & S' \\ \hline (0) & (0) \end{array}\right]$$

- the columns that does not belong to $S$: Sub-matrix $S'$ will be linear combinaisons of the columns of $S$.

# Obtention of the components

$$C' = \left[\begin{array}{c|c} S & S' \\ \hline (0) & (0) \end{array}\right]$$

- The $n - r$ lines that does not belong to $S$ neither to $S'$ will only contains null values.
- Indeed, they corresponds to variables that does not appear in no equation then their values are free.
- Then we have the following system:

$$f'^{T}.C' = 0 \quad \text{with} \quad \begin{array}{ccc} f & = & F.f' \\ \text{et} \quad C' & = & F^{T}.C \end{array}$$

# Obtention of the components

$$f'^{T}.C' = 0 \quad \text{with} \quad f = F.f'$$

- $F$ is a regular matrix (non null determinant) which depicts all the changes in variables which have been realized for the resolution.
- We obtain $S$ we the form:

$$\begin{bmatrix} s_{11} & s_{12} & s_{13} & \dots & s_{1r} \\ 0 & s_{22} = 1 & s_{23} & \dots & s_{2r} \\ 0 & 0 & s_{33} = 1 & \dots & s_{3r} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & s_{rr} = 1 \end{bmatrix}$$

# Solution of S

- The square sub-system $S$ :

$$\begin{bmatrix} s_{11} & s_{12} & s_{13} & \dots & s_{1r} \\ 0 & s_{22}=1 & s_{23} & \dots & s_{2r} \\ 0 & 0 & s_{33}=1 & \dots & s_{3r} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & s_{rr}=1 \end{bmatrix} \begin{bmatrix} f'_1 \\ f'_2 \\ f'_3 \\ \dots \\ f'_r \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- Gives:

$$\begin{aligned} f'_1 &= 0 \\ s_{12} f'_1 + f'_2 &= 0 \\ &\dots \\ s_{1r} f'_1 + s_{2r} f'_2 + \dots + f'_r &= 0 \end{aligned}$$

# Solution of S

- The system has an unique solution relativeley to $r$ variable $f'_1, f'_2, ..., f'_r$ which is the degenerated solution.
- Then the solutions, upon the form $f'$ are such as the first $r$ components of the vector are ever equal to zero.
- In contrast, the $n - r$ other components can be freely chosen.

# Vectorial space

- The obtention of the vectorial space is given with the choice of the solutions for which one and only one ine the components of $f'$ is not null.
- goal it is the computing of $F$ which gives the base of the conservative componants.

# Triangularisation

- Let's suppose we the way of the triangularisation of the system and that the $i-1$ lines and the $i-1$ columns have been triangularised-:

$$\begin{bmatrix} 1 & c_{21} & \dots & c_{i1} & \dots & c_{m1} & 0 \\ 0 & 1 & \dots & \dots & \dots & c_{m2} & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & 0 & 0 & c_{ii} & \dots & c_{mi} & 0 \\ 0 & 0 & 0 & c_{i(i+1)} & \dots & c_{m(i+1)} & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & 0 & 0 & c_{in} & \dots & c_{mn} & 0 \end{bmatrix}$$

- Considering the column $i$ : all the $c_{i(i+1)} \dots c_{in}$ must be zeroed.

# Triangularisation

- To zero theses coefficients, we compute:

$$\forall k \in [i+1, n] \quad Ligne_k \longleftarrow \; Ligne_k * c_{ii} \; - \; Ligne_i * c_{ki}$$

- If $c_{ii} = 0$ Then
  We realize the permutations into the $m - i$ columns of $C$ to obtain a non null $c_{ii}$.
- If all the elements are null, the line $i$ is nul. Then $f_i$ has been eliminated of the system, and will produce an element of the vectorial space of the solutions.
- In this case we permut the $n - i$ lines of $C$ to make appear a non null line.
- If we find one: We permut the columns to have a non null $c_{ii}$.
- Or else : The algo has ended: $C$ is transformed in $C'$, and $F$ give us the base of the vectorial space of solutions.

# Change of variables

$$\forall k \in [i+1, n] \quad Ligne_k \longleftarrow \; Ligne_k * c_{ii} \; - \; Ligne_i * c_{ki}$$

- This operation in $C$ is memorized in the vector column by the following change: $f_i \;=\; f'_i - c_{ki}.f'_k$
  $f_k \;=\; c_{ii}.f'_k$
  $f_j \;=\; f'_j \quad \forall j \quad \neq i, k$

# Change of variables

- For a column $j$ of $C$ :
  $c_{1j}.f_1 + \ldots + c_{ij}.f_i + \ldots + c_{kj}.f_k + \ldots + c_{nj}.f_n = 0$
- with the change of variable :
  $c_{1j}.f'_1 + \ldots + c_{ij}.(f'_i - c_{ki}f'_k) + \ldots + c_{kj}.c_{ii}f'_k + \ldots + c_{nj}.f'_n = 0$
- We develop $c_{ij}$ and we factorise $f'_k$:
  $c_{1j}.f'_1 + \ldots + c_{ij}.f'_i + \ldots + (c_{kj}.c_{ii} - c_{ij}.c_{ki})f'_k + \ldots + c_{nj}.f'_n = 0$
- For the column $i \longrightarrow j = i$ : We verify that $f'_k$ is eliminated of the equation.

# Successive transformations of $f$

- In the beginning $f$ is an identity matrix.
- Then $f$ undergo the transformations done on $C$ to become $F$.

$$\left\{ \begin{array}{lcll} f_k & = & c_{ii}.f'_k & \\ f_i & = & f'_i & - c_{ki}.f'_k \\ f_j & = & f'_j & \forall j \neq i, k \end{array} \right. \quad \begin{bmatrix} f_1 \\ f_i \\ f_k \\ f_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -c_{ki} & 0 \\ 0 & 0 & c_{ii} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f'_1 \\ f'_i \\ f'_k \\ f'_n \end{bmatrix}$$

- Matrix $F$ memorizes the change of variables.

# Exercice

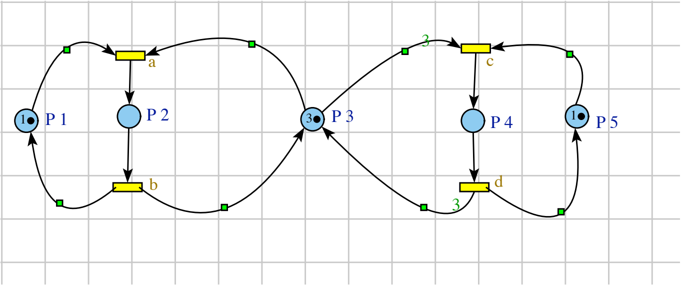



Figure: $t$ is neutral

Let's find the conservative componants and the invariants of places for the intial marking.

# Simplified algorithm for positive solutions

- Only the computing of the columns of the matrix $F$ corresponding to the $n - r$ components of the base the solutions of $F$ is usefull. The other columns can be erased.
- At each step of the elimination of a variable:
  - the column of $C$ which has allowed to eliminate a variable is erased
  - and the corresponding line is erased.
  - moreover the line in $F$ is also erased.
- $F$ memorizes the computed operations.

# Simplified algorithm for positive solutions

- The goal is to realize an algorithm which favours in a first step, the positive linear combinaisons in the erasing of variable :
  $f_i + n.f_j$ with $n > 0$.
- We examine the columns of $C$ and the signs the termes of theses columns.
- This test allows to distinguish four cases that we examine by increasing order.
- According to the case, we realize a specific action and we iter the examination of $C$.

# Simplified algorithm for positive solutions

## Step 1

We search a column with **only one** non null term:

1. We erase of $C$ the line that is associated at the non null variable.
2. We erase of $C$ the column that is associated at the non null variable.
3. We erase of $F$ the column that is associated at this variable.

We iter this step then we go to the step 2.

# Simplified algorithm for positive solutions

## Step 2

- We search a column with **only one non null term** with a given sign (postive or negative).
- If this column does not existe let's go in step 3.
- All the other terms are null or opposite signs.
- We erase then of $C$ the line that is associated at this variable by positive combinaisons of lines of type : $Ligne_k * c_{ii} + Ligne_i * c_{ki}$

Back to step 1.

# Simplified algorithm for positive solutions

## Step 2 : Erase operations

1. We erase of $C$ the line that is associated to this variable.
2. We erase of $C$ the column that is associated to this variable.
3. We memorize in the vector $F$ the linear combinaison that has been used.

# Simplified algorithm for positive solutions

## Step 3

- We search a column with $i \geq 2$ strictly positive components and $j \geq 2$ strictly negative components.
- if such a column does not exist go to step 4.
- With the help of a positive component, let erase the $j - 1$ negative components (step 2)
- Then it remains **only on** negative component and a set of components positives that we are going to erase with the step 2.

Remark :
We have chosen a positive component among $i$ and a component among $j$ : There may be $i * j$ different solutions.

# Simplified algorithm for positive solutions

## Step 4

- We now search a column with all the components that non null and of the same sign.
- We carry on the triangularisation with the combinaisons that are not positives.
- Now : either we can go to step 1: it exists a column with a unique non null component,
- Either the algorithm stop and then:
  - Either all the columns are null
  - Either there is no column anymore in the matrix
- the vector $F$ gives the base of the solutions.

# Exercice 1

Find the conservative components and the invariant of places with the simplified algorithm for positive solutions for the following example:

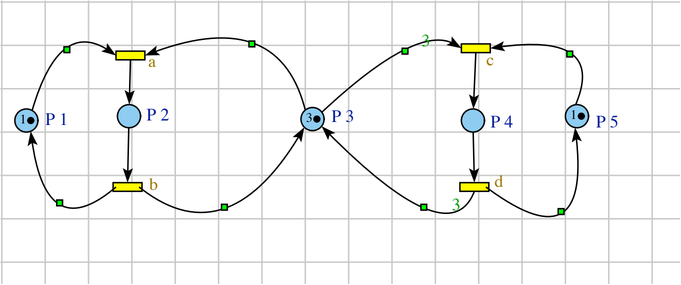



Figure: *t* is neutral

# Exercice 2

Find the conservative components and the invariant of places with the simplified algorithm for positive solutions for the following example:

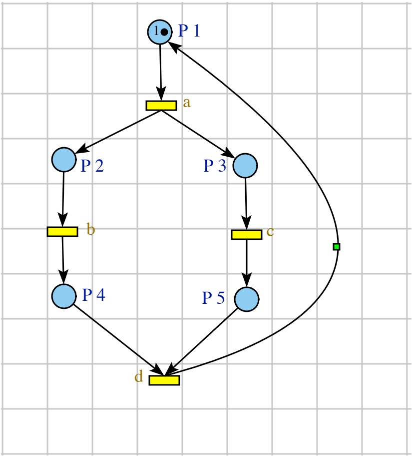



Figure:

# Exercice 3

Find the conservative components and the invariant of places with the simplified algorithm for positive solutions for the following example:

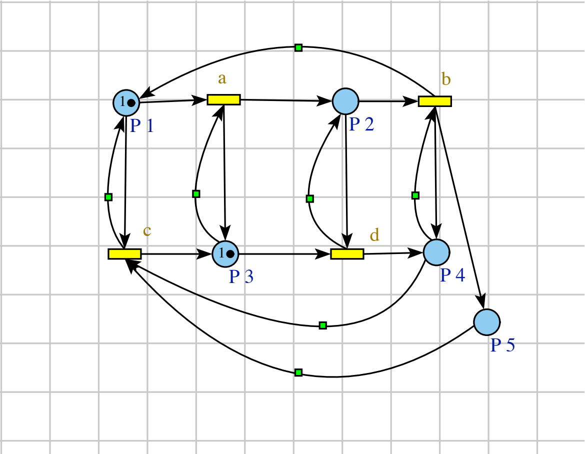



Figure:

# Relations between the good properties

**UNBOUNDED** Marked PN $\not\Longrightarrow^{MG}$ liveness or reinitialisability

- Place invariants allow to show that certain places are bounded.
- Transition Invariants are not sufficient conditions of bounded region : the transitions belonging to the invariant may not be reachable.

# Cover of conservative componants

## Cover of conservative componants (*CCC*)

It exists a *Cover* of conservative componants (resp. repetitive stationnary) if one can find a *positive* component or a set of elementary *positives* components which cover all the places (resp. transitions).

## Cover of places

A RdP for which, it exists a cover of conservative componants $f^T > 0$, is k-bounded **whatever its initial marking**

# Cover of conservative componants

## Calcul the k-bornes

The linear forme $f^T.M = f^T.M_0$ allows to compute a bound for all the places belonging to a cover of conservative componants.

- For the places $p$ of a $CCC$ we have the property :

$$\forall p \in P, f(p).M(p) \leq f^T.M_0$$

- Thus because $p$ belongs to the $CCC$ :

$$f(p).M(p) \leq f'^T.C' = f^T.M_0$$

- and then :

$$M(p) \leq \frac{f^T.M_0}{f(p)}$$

# Cover of conservative componants

Remark : A RdP can be bounded for a marking without *CCC* like in this example:

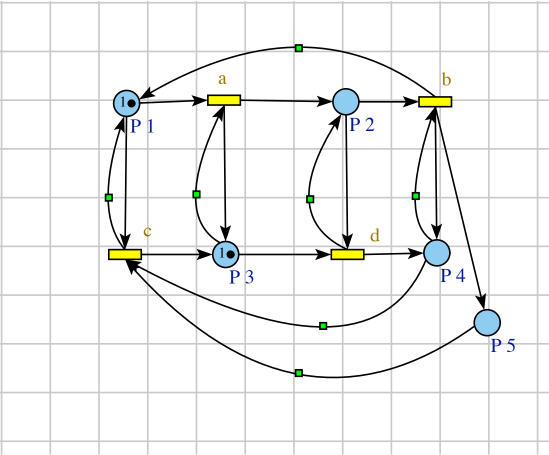



Figure: CCC

# boundedness, liveness and Cover of transitiion

## Cover of transition and the good properties

Every PN which is in the same time alive, bounded for an initial marking is such that a cover of stationnary repetitive transition $\overline{s}$ exists.

Indeed:

- Bounded $\Rightarrow$ Finite Number fini of marking that enable a given transition $t$.
- Alive $\Rightarrow$ sequences of finite lenght
- Alive + bounded $\Rightarrow$ it exists a loop passing through $t$ $\Rightarrow$ it exists a stationnary repetitive transition.

The reciprocal is false :
One only has to take a null initial marking and the PN is no more alive as no transitions are firable.

# Complementarity between the analyze by reduction and the computing of the components

- Let's consider a *PN* which does not have the good properties.
- The method by reduction is not able to bring a diagnostic to the anomalies: non boundedness or deadlock
- The computing of the *CC* allows to determine the places which are bounded. The non bounded places does not belong to theses components.
- For the non alive transitions: no repetitive stationnary positives pass through by them.

# Characterization of the marking

The different ways to characterize markings:

- Enumeration of reachable markings.
- Fundamental equation: we obtain a set which includes the set of the reachable markings : we can find a sequence such as $C.\overline{s} \geq 0$ but $s$ is not necessarly reachable.
- The *CCC* which includes the two previous as it is illustrated in the following slide.

# Exercice

- Find $A(R, M_0)$
- Find the set the vectors $\overline{s}$ which give a marking $M' > 0$ at partir of marking initial.
- Find the set the reachable marking by the fundamental equation
- Find the conservative componants

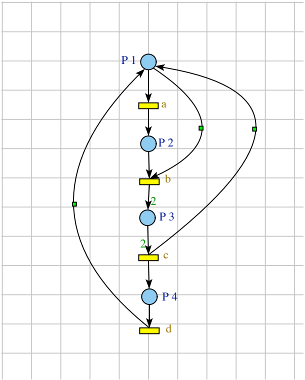



Figure: Characterization of the markings

# Characterization of the markings

- The reachable markings constitues the smallest set.
- The fundamental equation does not give any information about the reachability of markings. This equation furnish an overset of reachable markings.
- The *CCC* contitutes an overset of the fundamental equation.

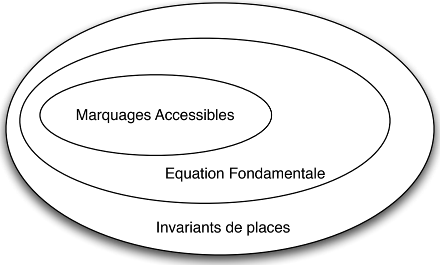



Figure: Characterization of the markings

# Racket

Language :

- Write script shell.
- Prototype a complex animation, GUI including regular expression, thread.
- Use of classes, modules, components.
- Use of files, flux, ...
- Compilation, cros-compilation.

# Racket et LISP

- *LISP* : Artificial Intelligence [1]
- LISP functional language
- Racket recent evolution of LISP
- MacOS, LINUX and Windows
- Prefixed Language with parenthesis.
  - $f(a, b, c, d)$ is noted $(f\ a\ b\ c\ d)$
  - $1 + 2 + 3 + 4$ is noted $(+\ 1\ 2\ 3\ 4)$

[1] McCarthy, 1958, "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I ", CACM

# Language Syntax : EBNF

$$
\begin{array}{lcc}
\textit{expression} & \rightarrow & \textit{atom} \mid \textit{list} \\
\textit{atom} & \rightarrow & \textit{number} \mid \textit{identifier} \mid \textit{string} \mid \textit{character} \mid \textit{operator} \\
\textit{list} & \rightarrow & (\ \textit{expression}^{*}\ )
\end{array}
$$

# Numbers

- Exact number:
  - 17 ou 5,9999999999999999
  - Rational number : 1/2, -3/4
- Non exact number:
  - 2.0, 3.14e+87
  - +inf.0, -inf.0,
  - +nan.0, -nan.0
  - Complex : 2.0+3.0i

# Boolean

#t et #f
Examples:

```
> (= 2 (+ 1 1))
#t
> (boolean? #t)
#t
> (boolean? #f)
#t
> (boolean? "no")
#f
> (if "no" 1 0)
1
```

# Flux

```
(read [in]) → any
  in : input-port? = (current-input-port)
```

```
(write datum [out]) → void? (display or print)
  datum : any/c
  out : output-port? = (current-output-port)
```

- Reads and returns a single datum from in.
- If in has a handler associated to it via port-read-handler, then the handler is called.
- Otherwise, the default reader is used,
- Writes datum to out, normally in such a way that instances of core datatypes can be read back in.
- If out has a handler associated to it via port-write-handler, then the handler is called.
- Otherwise, the default printer is used.

# String

A string is defined by "... "

```
> "Apple"
"Apple"
> (display "Apple")
Apple
```

# Symbol

## Symbol

A Symbol is an object which can represent:

- a constant,
- a variable,
- a function.

The evaluation of a symbol gives its value. The quote prevents the evaluation of the symbol.

# Symbol Racket

- A Symbol corresponds to an atomic value.
- The extern representation of a symbol is string character: The identifier.
- The identifier may be prefixed by a simple cote ’ tp prevt its evaluation
- string->symbol : gives the identifier (string character)
- Two symbols can be compared in performant way by eq?

# Examples:

```
> 'a
'a
> (symbol? 'a)
#t
> (define a 3)
> a
3
> (define b (+ a 3))
> b
6
> (+ a 7)
10
> (define (c p) (- 1 p))
> (c 3)
-2
> (c a)
-2
> (c b)
-5
```

# Pairs and lists

## Pair

A pair is a link between two elements:

$$( a \,.\, b )$$

- *cons* is the operator that builds a pair,
- *car* extracts the first element *a*,
- *cdr* extracts the second element *b*.

# Example of pairs

Examples:

```
> (cons 1 2)
(1 . 2)
> (cons (cons 1 2) 3)
((1 . 2) . 3)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? (cons 1 2))
#t
```

# Lists

## List

A list is either the empty list ((), null ou empty), either a singular pair such as the *car* is an element and the *cdr* is a list.

*list* is also an operator making lists with several elements.

# Examples: lists

```
> null
   ()
> (cons 3 null)
   (3)
 > (list 'a 'b 'c)
(a b c)
> (define L1 (list 'a 'b 'c 'd))
> L1
(a b c d)
> (define P '(P0 P1 P2))
> P
'(P0 P1 P2)
> (car P)
'P0
> (cdr P)
'(P1 P2)
```

# Examples: Lists

```
> (cons P T)
'((P0 P1 P2) T0 T1 T2)
> (cons P (list T))
'((P0 P1 P2) (T0 T1 T2))
> (define PT (cons P (list T)))
> PT
'((P0 P1 P2) (T0 T1 T2))
> (car PT)
'(P0 P1 P2)
> (cdr PT)
'((T0 T1 T2))
> (cadr PT)
'(T0 T1 T2)
```

# Functions on lists

append, reverse, member, equal?, eq? :

```
> (append P T)
'(P0 P1 P2 T0 T1 T2)
> (length PT)
  2
> (list-ref P 0)
'P0
> (reverse PT)
'((T0 T1 T2) (P0 P1 P2))
> (member 'P1 P)
'(P1 P2)
> P
'(P0 P1 P2)
> (member 'P4 P)
#f
> (equal? '(p1 p2) '(p1 p2))
#t
> (eq? '(p1 p2) '(p1 p2))
#f
```

# Basics functions

length, list-ref:

```
> (length P)
3
> (list-ref P 0)
'P1
```

# PN structure

```
> P
'(P1 P2 P3)
> T
'(T1 T2 T3 T4)
```

> $^{\bullet}P$

```
'((P1 T1) (P2 T2 T4) (P3 T3))
```

> $^{\diamond}P$

```
'((P1 T2) (P2 T1 T3) (P3 T4))
```

> $^{\bullet}T$

```
'((T1 P2) (T2 P1) (T3 P2) (T4 P3))
```

> $^{\diamond}T$

```
'((T1 P1) (T2 P2) (T3 P3) (T4 P2))
```

> $M_0$

```
'(0 3 0)
> Arc
'((P2 T1 1) (T1 P1 1)(P1 T2 1)(T2 P2 1)(P2 T3 3)(T3 P3 1)
(P3 T4 1)(T4 P2 3))
>
```

# Functions

## Function

The functions can be considered as datas or parameters. For example, it is possible to write functions which take functions as parameters and which may return functions.
Syntax : (define (f1 a b c d f2) ... Body of the function...)
If a function has two parameters, it's has an alternative representation:

$$(Op\ a\ b)\ \equiv\ (a\ .\ Op\ .\ b)$$

# Functions

```
>(define (print E)
   (for ([elt E])
     (printf "~a " elt)))
>(print P)
(P1 P2 P3)
```

# Functions

Example : Sort has a a sequence and a comparaison operator in parameter.

```
> (sort '(1 3 4 2) <)
  (1 2 3 4)
> (sort '("aardvark" "dingo" "cow" "bear") string<?)
  ("aardvark" "bear" "cow" "dingo")
> (sort `((9 a) (3 b) (4 c))
      (lambda (x y) (< (first x) (first y))))
((3 b) (4 c) (9 a))
>  (define (twice f v)
    (f (f v)))
> (twice sqrt 16)
  2
```

# Lambda function

## Lambda function

Lambda function is an anonymous function either at local use or unique use.

- It is defined in the body of a function and its definition is then valid for the entire body of the function.
- In interpredeted mode, the use is uniq.

# Lambda function

```
> ((lambda (x) x) 10)
10
> ((lambda (x y) (list y x)) 1 2)
(2 1)
> ((lambda (x [y 5]) (list y x)) 1 2)
(2 1)
> ((lambda (x [y 5]) (list y x)) 1)
(5 1)

> (twice (lambda (s) (string-append s "!")) "hello")
"hello!!"
```

# Structures

```
( define-struct mystructure (name x y) )
```

mystructure is defined as a structure. This definition induce the methods:

- $mystructure - name$,$mystructure - x$ and $mystructure - y$ access to the members of the structure.
- $make - mystructure$ builder
- *mystructure*? return a boolean

Example :

```
> (define-struct mystructure (name x y) )
> (define C1 (make-mystructure "toto" 5 12))
> (mystructure-x C1)
5
> (mystructure-name C1)
toto
> (mystructure? C1)
#t
```

# Conditional structure

```
( if expr expr expr )
 ( and expr )
( or expr )
 ( cond {[expr expr]}* )
```

# IF

Example:

```
> (if (> 2 3) "greater" "lower")
"lower"
```

# AND

Example:

```
(define (sup a b)
    (if (and (number? a) (number? b))
        (if (< a b) (display "greater") (display "lower"))
        (display "not comparable")))
> (sup 2 5)
lower
> (sup 2 "toto")
not comparable
```

# COND

```
(cond  [... ]
       [... ]
       [else ...] )
```

Example :

```
(define (analyseRdP P T)
  (cond
    [(and (empty? P) (empty? T)) (print "Null PN") ]
    [(or (empty? P) (empty? T)) (print "Degenerated PN     ")]
    [else (printf
           "there is ~a places and ~a transitions"
           (length P)
           (length T))]))
```

# Recursivity

## Recursive Function

The definition of $f$ is recursive if its definition contains a call of $f$.

Example: We want to determine if an element belong to a list of element.

```
(define (member? elt E)
  (cond
    [(empty? E) #f]
    [(equal? (car E) elt) #t]
    [else (member? elt (cdr E))]))
```

# Recursivity

Example: We want to determine if an element belong to a list of lists or elements.

```
(define (member2 elt E)
  (cond
    [(empty? E) #f]
    [(equal? (car E) elt) #t]
    [(list? (car E)) (or (member2 elt (car E)) (member2 elt (
       cdr E)))]
    [else (member2 elt (cdr E))]))
```

# List and pairs

Exercise : We want a function that in a list of pair (car cdr) returns the cdr when the car is equal to elt

```
(define (conjoints elt list)
  (cond
    [(empty? list) empty]
    [(if (equal? (caar list) elt)
         (cdr (car list))
         (conjoints elt (cdr list)))]))
```

use:

(conjoints 'P2 $^\bullet P$) = (T2 T4)
(conjoints 'P4 $^\bullet P$) = () ; *void list*
(conjoints 'P3 $^\diamond P$) = (T4)

# Antecedent and successor of a place or a transition

```
;  Antecedents of pt:
(define (• pt)
  (if (membre pt T)(conjoints pt •T)(conjoints pt •P)))

; Successors of pt:
(define (◇ pt)
  (if (membre pt T)(conjoints pt ◇T)(conjoints pt ◇P)))
```

# Antecedents of antecedents and successors of successors of a place or a transition

```
;  Antecedent of antecedents of pt:
(define (•• pt)
  (cond
    [(list? (• pt)) (for/list ([e (• pt)]) (• e))]
    [else (• (• pt))]))

; Successor of sucessors of pt:
(define (◇◇ pt)
  (cond
    [(list? (◇ pt)) (for/list ([e (◇ pt)]) (◇ e))]
    [else (◇ (◇ pt))]))
```

# *for*

```
(for (for-clause ...) body ...+)

for-clause    =   [id seq-expr]
    |   [(id ...) seq-expr]
    |   #:when guard-expr
    |   #:unless guard-expr

   seq-expr    :   sequence?
```

# *for*

- The for-clause for is a set of pairs $[id_0\ seq_0], [id_1\ seq_1],\ ..., [id_n\ seq_n],\ ...$
- $id_i$ will match to $seq_i$
- The ids are then bounded in the body, which is evaluated, and whose results are ignored.
- Iteration continues with the next element in each sequence and with fresh locations for each id.
- If any for-clause has the form #:when guard-expr, then only the preceding clauses determine iteration as above : Filter
- A #:break guard-expr clause is similar to a #:unless guard-expr clause, but when #:break avoids evaluation of the bodys, it also effectively ends all sequences.

# Example with *for*

```
> (for ([i '(1 2 3)]
        [j "abc"]
        #:when (odd? i)
        [k #(#t #f)])
    (display (list i j k)))
(1 a #t)(1 a #f)(3 c #t)(3 c #f)
> (for ([(i j) (list (cons "a"  1) (cons "b" . 20))])
    (display (list i j)))
(b 20)(a 1)
```

# for / or

(for/or (for-clause ...) body ...+)

- Iterates like for, but when last expression of body produces a value other than #f, then iteration terminates, and the result of the for/or expression is this value (the value that is different of #f).
- If the body is never evaluated, then the result of the for/or expression is #f.
- Otherwise, the result is #f.

```
Examples:
> (for/or ([i '(1 2 3 "x")])
    (print i) (i . < . 3))
1#t
> (for/or ([i '(1 2 3 4)])
    i)
1
> (for/or ([i '()])
    (error "doesn't get here"))
#f
```

# *for / or*

```
(define (Pre t)
  (cond
    [(empty? •T) empty]
    [else (for/or ([pre •T])
               (and (equal? t (car pre))
                          (cdr pre)))]))

(define (Post t)
  (cond
    [(empty? ◦T) empty]
    [else (for/or ([post ◦T])
        (and (equal? t (car post))
                          (cdr post)))]))
```

# for / and

(for/and (for-clause ...) body ...+)

- Iterates like for, but when last expression of body produces #f, then iteration terminates, and the result of the for/and expression is #f.
- If the body is never evaluated, then the result of the for/and expression is #t.
- Otherwise, the result is the result from the last evaluation of body.

# *for / and*

```
> (for/and ([i '(1 2 3 "x")])
    (i . < . 3))
#f
> (for/and ([i '(1 2 3 4)])
    i)
4
> (for/and ([i '()])
    (error "doesn't get here"))
#t
```

# State Graph

Formal definition:

**Definition**

A PN is a State Graph $\iff \forall t \in T, |{}^{\bullet}t| = 1 \wedge |t^{\bullet}| = 1$

Coding ?

# State Graph

Formal definition:

## Definition

A PN is a State Graph $\iff \forall t \in T, |{}^{\bullet}t| = 1 \wedge |t^{\bullet}| = 1$

Coding:

```
(define (StateGraph?)
  (cond
    [(or (empty? T) (empty? P)) #f]
    [else (for/and ([t T]) (and
                          (equal? (length (• t)) 1)
                          (equal? (length (◇ t)) 1)))]))

(StateGraph?)
```

# Event Graph

Formal definition:

**Definition**

A PN is an Event Graph $\iff \forall p \in P, |{}^{\bullet}p| = 1 \wedge |p^{\bullet}| = 1$

Coding ?

# Event Graph

Formal definition:

**Definition**

A PN is an Event Graph $\iff \forall p \in P, |{}^\bullet p| = 1 \land |p^\bullet| = 1$

Coding:

```
(define (EvtGraph?)
 (cond
   [(or (empty? P) (empty? T)) #f]
   [else (for/and ([p P]) (and
                      (equal? (length (• p)) 1)
                      (equal? (length (◇ p)) 1)))]))
```

# PN without conflicts

Formal definition:

**Definition**

A PN is without conflicts $\iff \forall p \in P, |p^\bullet| < 2$

Coding ?

# PN without conflicts

Formal definition:

**Definition**

A PN is without conflicts $\iff \forall p \in P, |p^\bullet| < 2$

Coding:

```
(define (PnWithoutConflict?)
  (cond
    [(or (empty? P) (empty? T)) #f]
    [else (for/and ([p P]) (< (length (° p)) 2))]))
```

# Conflict

Formal definition:

## Definition

A conflict is defined by a place $p$ et a set $\{t_1, t_2, ..., t_n\}$ such as $p^\bullet = \{t_1, t_2, ..., t_n\}$

Coding ?

# Conflict

Formal definition:

**Definition**

A conflict is defined by a place $p$ et a set $\{t_1, t_2, ..., t_n\}$ such as $p^\bullet = \{t_1, t_2, ..., t_n\}$

Coding:

```
(define (conflict P)
  (cond
    [(empty? P) empty]
    [(> (length (◇ (car P))) 1)
     (cons (cons (car P) (◇ (car P))) (conflict (cdr P)))]
    [else (conflict (cdr P))]))

    (define conflicts (Conflits))
```

# Conflict?

Exercise: Does a place or a transition belong to a conflict ?

`(conflict? pt)` $\Rightarrow$ `boolean`

# Conflict?

Does a place or a transition belong to a conflict ?
Coding:

```
(define (conflict? pt)
  (cond
    [(empty? conflicts) #f]
    [else (member? pt conflicts)]))
```

# Free Choice Conflict

Formal definition:

**Definition**

A PN is free choice iff every conflict $c =< p_c, \{t_1, t_2, ..., t_n\}_c >$ of the net is such as $\neg(\exists p' \in P / p' \neq p_c \wedge p'^{\bullet} \cap \{t_1, t_2, ..., t_n\}_c \neq \emptyset)$

Coding?

# Free Choice Conflict

Formal definition:

## Definition

A PN is free choice iff every conflict $c =< p_c, \{t_1, t_2, ..., t_n\}_c >$ of the net is such as $\neg(\exists p' \in P / p' \neq p_c \wedge p'^{\bullet} \cap \{t_1, t_2, ..., t_n\}_c \neq \emptyset)$

Coding:

```
(define (FreeChoice?)
    (for/and ([c conflicts])
      (not (for/or ([p1 P] #:when (not (equal? p1 (car c))))
           (inter (° p1) (cdr c))))))
```

# Simple choice

Formal definition:

## Definition

A PN is simple choice if every transition is only implied in one conflict
Let's $C$ the set of conflicts:

$$\forall c_1, c_2 \in C \wedge c_1 \neq c_2, \forall t_i \in c_1, t_i \notin c_2$$

Coding ?

# Simple choice

Formal definition:

## Definition

A PN is simple choice if every transition is only implied in one conflict
Let's $C$ the set of conflicts:

$$\forall c_1, c_2 \in C \wedge c_1 \neq c_2, \forall t_i \in c_1, t_i \notin c_2$$

Coding:

```
(define (simpleChoice?)
    (for/and ([c conflicts])
      (for/and ([ti (^ (car c))])
        (not (membre2 ti (autre c conflicts))))))

(simpleChoice?)
```

# Pure

Formal definition:

**Definition**

A PN is pure iff $\forall t \in T : Pre(., t).Post(., t) = 0$

Coding ?

# Pure

Formal definition:

**Definition**

A PN is pure iff $\forall t \in T : Pre(., t).Post(., t) = 0$

Coding:

```
(define (Pure?)
  (cond
    [(or (empty? P) (empty? T)) #t]
    [else (for/and ([t T])
            (not (inter? (• t) (◦t))))]))
```

# PN without loops

Formal definition:

## Definition

A PN is without loops if it exists a transition $t_j$ and a place $p_i$ that is in the same time input and output place of $t_j$ then $t_j$ has at least another inuput place that is not and output place

$$\forall t \in T, \text{ IF } (\exists p \in P / p \in \bullet t \wedge p \in t^{\bullet}) \text{ THEN } \exists p' \in^{\bullet} t \wedge p' \notin t^{\bullet}$$

Coding ?

# PN without loops

Formal definition:

## Definition

A PN is without loops if it exists a transition $t_j$ and a place $p_i$ that is in the same time input and output place of $t_j$ then $t_j$ has at least another inuput place that is not and output place

$$\forall t \in T, \text{ IF } (\exists p \in P / p \in \bullet t \land p \in t^{\bullet}) \text{ THEN } \exists p' \in^{\bullet} t \land p' \notin t^{\bullet}$$

Coding:

```
(define (WithoutLoops?)
  (cond
    [(or (empty? P) (empty? T)) #t]
    [else (for/and ([t T]
                #:when (inter? (• t) (◇ t)))
                (for/or ([p  (• t)])
                  (not (membre p (◇ t))))))]))
```

# Vectors

## Vector

- A vector is a fixed-length array.
- Access and update of vectors elements are normally constant-time operations.
- Numbered from 0 to $n - 1$

```
Examples:
  > #("a" "b" "c")
  #("a" "b" "c")
  > (vector-ref #("a" "b" "c") 1)
  "b"
  > (vector-ref #(P (T1 T2 T3)) 0)
  P
    > (vector-ref #(P (T1 T2 T3)) 1)
   (T1 T2 T3)
```

# Hashing table

- A hash table implements a mapping from keys to values, where both keys and values can be arbitrary Racket values,
- Access and update to the table are normally constant-time operations.
- Keys are compared using equal?, eqv?, or eq?, depending on whether the hash table is created with make-hash, make-hasheqv, or make-hasheq.

# Hashing table

Examples:

```
> (define GDM_s (make-hash))
> (hash-set! GDM_s #(120000) (list 'T1 'T2))
> GDM_s
#hash((#(120000) . (T1 T2)))
>  (hash-ref ht #(120000))
(T1 T2)
> (hash-ref ht #(121000))
hash-ref: no value found for key: #(121000)
```

# hash-has-key?

```
(hash-has-key? hash key) → boolean?
  hash : hash?
  key : any/c
```

Returns #t if hash contains a value for the given key, #f otherwise.

# for/hash

(for/hash (for-clause ...) body-or-break ... body)

- Like for/list, but the result is an immutable hash table;
- for/hash creates a table using equal? to distinguish keys
- The last expression in the bodys must return two values: a key and a value to extend the hash table accumulated by the iteration.

# $W$ : Weight of arcs

```
(define W
    (for/hash ([e Arc])
            (values (drop-right e 1) (last e))))
```

# Position of an element *e* in a list *l*

```
(define (list-pos e l)
  (cond
    [(not (membre  e l)) #f]
    [(equal? e (car l)) 0]
    [else (+ 1 (list-pos e (cdr l)))]))
```

# Marking of p in M : M(p)

- mark returns M(p)
- otherwise #f

```
(define (mark p M)
  (cond
    [(not (membre p P)) #f]
    [(vector? M) (vector-ref M (list-pos p P))]
    [else #f]))
```

# Weight of an arc

- readW: gives the weight of an arc
- Key: defined by (p.t) or (t.p)

```
(define (readW key)
  (cond
    [(hash-has-key? W key) (hash-ref W key)]
    [else 0]))
```

# Effective Conflicts

## Definition

**Effective Conflicts:** Transitions $t_1$ **...** $t_n$ are in *effective conflics* iff they are in conflict and that it $\exists M$ reachable from $M_0$ such as:

$$M \geq Pre(., t_1)$$

$$...$$

$$M \geq Pre(., t_n)$$

Coding ?

```
(define (EffectiveConflict? M)
  (let ([C conflicts])
    (cond
      [(empty? C) #f]
      [(for/list ([c C] #:when
          (for/and ([t (cdr c)])
             (for/and ([p (• t)])
              (>= (mark p M)  (readW (list p t))
        ))))c)])))
```

# Structural parallelism

## Definition

**Structural parallelism:** Two transitions $t_1$ et $t_2$ are in *Structural parallelism* iff :

$$Pre(., t_1) \times Pre(., t_2) = 0$$

```
(define (Sparallel? t1 t2)
  (not (inter? (• t1) (• t2))))
```

# Effective parallelism

## Definition

**Effective parallelism:** Two transitions $t_1$ et $t_2$ are in *Effective parallelism* iff they are in structural parallelism structurel and that it $\exists M$ rechable from $M_0$ such as :

$$M \geq Pre(., t_1)$$

$$M \geq Pre(., t_2)$$

```
(define (Eparallel? t1 t2 M)
  (and (Sparallel? t1 t2)
       (for*/and ([p1 (• t1)] [p2 (• t2)])
           (and (>= (mark p1 M) (readW (list p1 t1)))
                (>= (mark p2 M) (readW (list p2 t2)))))))
```

# $M_1 >= M_2$

Definition :

$$M_1 \geq M_2 \iff \forall p \in P,\ M_1(p) \geq M_2(p)$$

Coding:

```
(define (>>= M1 M2)
 (cond
   [(equal? (vector-length M2) 0) #t]
   [else (for/and ([v1 M1] [v2 M2]) (>= v1 v2))]))
```

# $M_1 > M_2$

Definition :

$$M_1 > M_2 \iff \forall p \in P,\ M_1(p) \geq M2(p) \wedge \exists p' \in P,\ M_1(p') > M_2(p')$$

Coding:

```
(define (>> M1 M2)
   (cond
    [((vector-ref M1 0) . > . (vector-ref M2 0))
     ((vector-drop M1 1) . >>= . (vector-drop M2 1))
        ]
    [((vector-ref M1 0) . = . (vector-ref M2 0))
     ((vector-drop M1 1) . >> . (vector-drop M2 1))]
    [else #f]))
```

# $t$ firable in $M$?

Definition:

$$M \rightarrow^{t} \iff \forall p \in P,\ M(p) \geq Pre(p,t)$$

Coding:

```
(define (->? M t)
  (let ([•t (• t)])
    (cond
      [(not (membre t T)) #f]
      [(empty? •t) #t]
      [else (for/and ([p •t])
        (>= (mark p M) (hash-ref W (list p t))))])))
```

# Set of enabled transitions in a Marking

Definition:

$$\{t \in T,\ M \to^{t}\}$$

Coding:

```
(define (enabled M)
  (filter (lambda(t) (M . ->? . t)) T))
```

# $M', M \rightarrow^{t} M'$

Definition:

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

Coding:

```
(define (-> M t)
  (list->vector(for/list ([p P])
       (+ (- (mark p M) (readW (list p t)))
            (readW (list t p))))))
```

# M' is it newly produced ?

Definition :

$$M \rightarrow^t M', M' \in^? A(R, M_0)$$

Coding:

```
(define (new-marking? M t)
  (cond
    [(empty? t) (not (hash-has-key? GDM_s M))]
    [else (if (M . ->? . t)
              (not (hash-has-key? GDM_s (M . -> . t)))
              #f )]))
```

# Stop condition computing $A(R, M_0)$

Definition :

$$let\ M, \exists ?\ M_i \in\ A(R, M_0), \forall p \in P \wedge M(p) > M_i(p)$$

Coding:

```
(define (>_Once M table)
 (for/first ([(Mi v) table] #:when (M . >> . Mi ))#t))
```

# HashTable $\Leftarrow$ New Marking

```
(define (Write clef table value)
     (if (hash-set! table clef value) #t #f))
```

# Computing $A(R, M_0)$

## $A(R, M_0)$

$$
\begin{array}{lcll}
M_0 & \in & A(R, M_0) & \\
M' & \in & A(R, M_0) & \iff \exists M \in A(R, M_0) \wedge t \in T \wedge M \to^t M'
\end{array}
$$

Algorithm:

```
A(R,M0) <- M0;
E0 <- Enabled (M0);
ITER(A(R,M0),E0);
```

# Computing $A(R, M_0)$

ITER(M,E)

```
FOR all t ∈ T DO {
  Let M′ such as M →^t M′
  If M′ ∈ A(R, M0) Then A(R, M_0) += (M →^t M′)
  Else {
    If ∃M_i ∈ A(R, M_0) such as M′ > M_i
    Then STOP the PN is unbounded
    Else {
      Let E′ ⇐ Enabled(M′);
      A(R, M0) += (M′, E′);
      ITER (M′, E′);
    }
  }
}
```

# Computing $A(R, M_0)$

```
(define (GDM)
  (let ([firable_M0 (enabled M_0)])
    (begin
      (hash-set! GDM_s M_0 (list empty firable_M0))
      (GDM_it M_0 firable_M0)
     )))
```

# Computing $A(R, M_0)$

```
(define (GDM_it M firable)
  (cond
    [(empty? firable) empty]
    [else (for/list ([t firable])
       (let ([Mi (M . -> . t)])
         (begin
           (if (hash-has-key? GDM_s Mi)
                (Write Mi GDM_s (cons (cons t
                  (car (hash-ref GDM_s Mi)))
                  (cdr (hash-ref GDM_s Mi))))
                (if (Mi . >_Once . GDM_s)
                 (begin (display "\n The net is unbounded\n")
                        (abort #f))
                 (let ([outputs (enabled Mi)])
                    (begin
                       (Write Mi GDM_s (list (list t) outputs))
                       (GDM_it Mi outputs))) ))
         )))
     ]))
```

# Computation time

```
(define (test)
  (let ([deb (current-inexact-milliseconds)])
           (begin
             (GDM)
             (let ([fin (current-inexact-milliseconds)])
               (display (- fin deb))))))
```