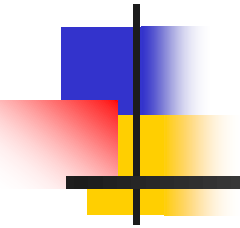
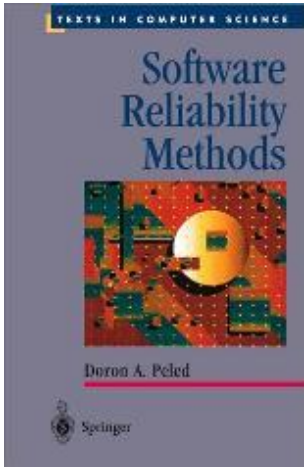


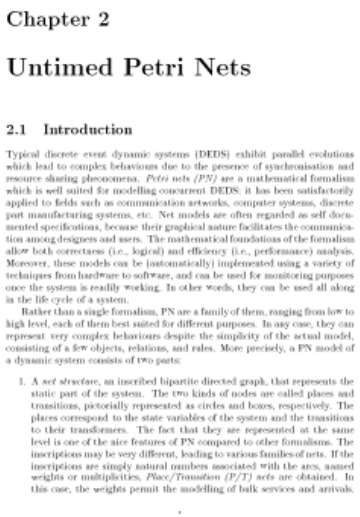
VPC 17-18
Algebre dei processi

Prof.ssa Susanna Donatelli
Universita' di Torino
www.di.unito.it
susi@di.unito.it

Reference material books:



Prof. Doron A. Peled
(University of Warwick, UK)



Chapter 2

Untimed Petri Nets

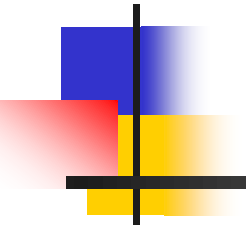
2.1 Introduction

Typical discrete event dynamic systems (DEDS) exhibit parallel evolutions which lead to complex behaviours due to the presence of synchronisation and resource sharing phenomena. *Petri nets (PN)* are a mathematical formalism which is well suited for modelling concurrent DEDS: it has been satisfactorily applied to fields such as communication networks, computer systems, discrete part manufacturing systems, etc. Net models are often regarded as self documented specifications, because their graphical nature facilitates the communication among designers and users. The mathematical foundations of the formalism allow both correctness (i.e., logical) and efficiency (i.e., performance) analysis. Moreover, these models can be (automatically) implemented using a variety of techniques from hardware to software, and can be used for monitoring purposes once the system is readily working. In other words, they can be used all along in the life cycle of a system.

Rather than a single formalism, PN are a family of them, ranging from low to high level, each of them best suited for different purposes. In any case, they can represent very complex behaviours despite the simplicity of the actual model, consisting of a few objects, relations, and rules. More precisely, a PN model of a dynamic system consists of two parts:

1. A *net structure*, an inscribed bipartite directed graph, that represents the static part of the system. The two kinds of nodes are called places and transitions, pictorially represented as circles and boxes, respectively. The places correspond to the state variables of the system and the transitions to their transformers. The fact that they are represented at the same level is one of the nice features of PN compared to other formalisms. The inscriptions may be very different, leading to various families of nets. If the inscriptions are simply natural numbers associated with the arcs, named weights or multiplicities, *Place/Transition (P/T) nets* are obtained. In this case, the weights permit the modelling of bulk services and arrivals.

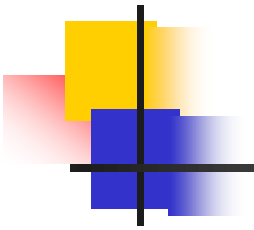
Notes of the EU-sponsored Jaca MATCH school



Acknowledgements

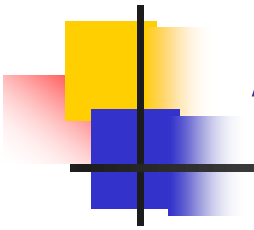
Transparencies adapted from the course notes and trasparencies of

- Prof. Jane Hillston (universita' di Edimburgo)
- Prof.ssa Marina Ribaudo (universita' di Genova)



Process Algebra

(Book: Chapter 8)



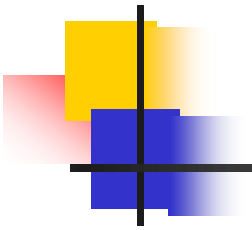
Algebra? Processi?

Def.: Sia I un insieme non vuoto. Diciamo struttura algebrica su I l'insieme

$$A = \{I, \alpha_1, \ldots\ldots \alpha_K\}$$

dove $\alpha_1, \ldots\ldots \alpha_K$ sono operazioni interne in I, rispettivamente a $n_1, \ldots, n_k$ argomenti

I è l'insieme dei processi



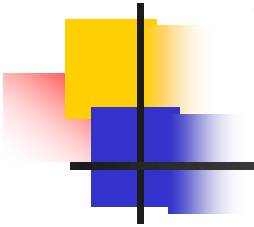
The Main Issue

Q: When are two models equivalent?

A: When they satisfy certain properties.

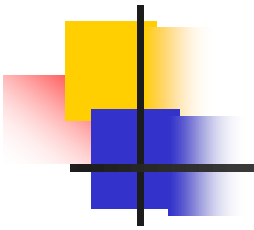
Q: Does this mean that the models have different executions?

What is process algebra?



- An abstract description for nondeterministic and concurrent systems.
- Focuses on the transitions observed rather than on the states reached.
- Interactions between independent processes as communication (no shared variables)
- Main correctness criterion: conformance between two models, but exhaustive state space generation and analysis is also possible
- Uses: system refinement, model checking, testing.
- *Defining algebraic laws for the process operators, which allow process expressions to be manipulated using equational reasoning*

Process algebra ingredients

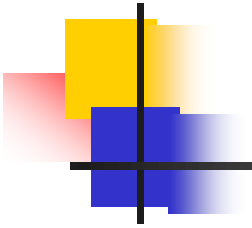


- *Act = {a,b,c, … }*, a set of actions
- $\forall a \in Act$, $\exists$ co-action $\underline{a} \in Act$
- τ-silent action, $\underline{\tau} = \tau$

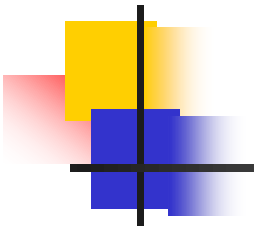
We shall define:

- a syntax for defining legal agents
- a semantics for defining how agents evolves
- equivalence rules to allow distinguishing agents

CCS - calculus of communicating systems [Milner]. Syntax



- *a,b,c*, … actions, A, B, C, …. - agents.
- $\underline{a},\underline{b},\underline{c}$, coactions of *a,b,c*. τ-silent action.
- 0 (o nil) - terminate.
- (E) – evaluation order
- *a*.E – prefixing. Execute a, then behave like E.
- E+E - nondeterministic choice.
- E || E - parallel composition.
- E\L - restriction: cannot use actions of L.
- E[f] - apply mapping function f between actions and agent names.



Conventions

- “.” has higher priority than “+”.
- “.0” or “.(0||0||…||0)” is omitted.

Semantics (proof rule and axioms). Structural Operational Semantics SOS

- *a*.p –a→ p
  - (a.p evolves with a in p)
- p—*a*→p′ |-- p+q –*a*→ p′
  - (given that p evolves with a in p′, then p+q evolves with a in p′)
- q—*a*→q′ |-- p+q –*a*→ q′
- p—*a*→p′ |-- p||q –*a*→ p′||q
- q—*a*→q′ |-- p||q –*a*→ p||q′
- p—*a*→p′, q—$\underline{a}$→q′ |-- p||q –τ→ p′||q′
- p—*a*→p′ , *a* $\notin$ R |-- p\R –*a*→ p′\R
- p—a→p′ |-- p[m]—m(*a*)→p′[m]

SOS rules and examples

$a.E \xrightarrow{a} E$ (Axiom)

Thus, $a.(b.(c /\!/ \underline{c}) + d) \xrightarrow{a} (b.(c /\!/ \underline{c}) + d)$.

Action Prefixing

$a.E \xrightarrow{a} E$ (Axiom)

Thus, $a.(\underbrace{b.(c||\underline{c})+d}) \xrightarrow{a} (\underbrace{b.(c||\underline{c})+d}).$

a. E --a--> E

Choice

$$\frac{E \xrightarrow{a} E'}{(E+F) \xrightarrow{a} E'} \qquad \frac{F \xrightarrow{a} F'}{(E+F) \xrightarrow{a} F'}$$

$b.(c//\underline{c}) \xrightarrow{b} (c//\underline{c})$.

Thus,

$(b.(c//\underline{c})+e) \xrightarrow{b} (c//\underline{c})$.

If $E \xrightarrow{a} E'$ and $F \xrightarrow{a} F'$, then E+F has *a* nondeterministic choice.

Concurrent Composition

$$\frac{E \xrightarrow{a} E'}{E||F \xrightarrow{a} E'||F} \qquad \frac{F \xrightarrow{a} F'}{E||F \xrightarrow{a} E||F'}$$

$$\frac{E \xrightarrow{a} E',\ F \xrightarrow{\underline{a}} F'}{E||F \xrightarrow{\tau} E'||F'}$$

$c \xrightarrow{c} \text{nil}$, $\underline{c} \xrightarrow{\underline{c}} \text{nil}$, then $c||\underline{c} \xrightarrow{\tau} \text{nil}||\text{nil}$, $c||\underline{c} \xrightarrow{c} \text{nil}||\underline{c}$, $c||\underline{c} \xrightarrow{\underline{c}} c||\text{nil}$.

Restriction

$$\frac{E \xrightarrow{a} E', \quad a, \underline{a} \notin R}{E\backslash R \xrightarrow{-a} E'\backslash R}$$

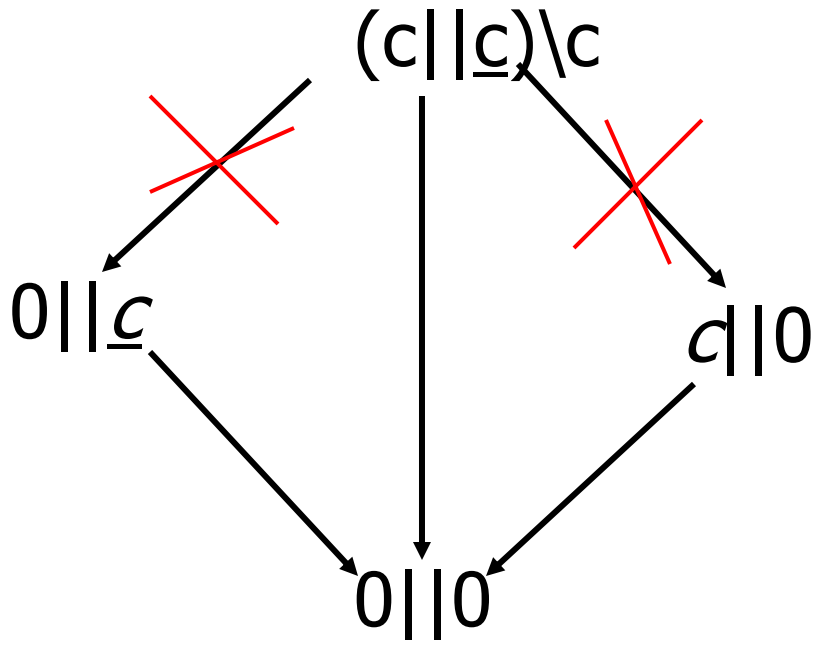
Example of interplay between parallel composition and restriction:

$c||\underline{c} \xrightarrow{-\underline{c}} c||0 \qquad c||\underline{c} \xrightarrow{c} 0||\underline{c} \qquad c||\underline{c} \xrightarrow{\tau} 0||0$

then

$(c||\underline{c}) \setminus \{c\} \xrightarrow{-\tau} (0||0) \setminus \{c\}$

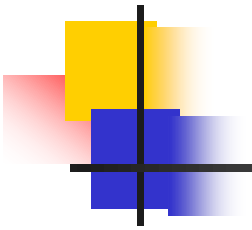
Consequence of restriction



Relabeling

$$\frac{E \text{—}a\rightarrow E'}{E[m] \text{–}m(a)\rightarrow E'[m]}$$

No axioms/rules for agent 0.



Equational definition of agents

The syntax presented only allows finite terms with finite behaviours

Process algebras usually allow an agent definition of the type

$P =_{def} a.(b.P)$

infinite behaviour (infinite traces)

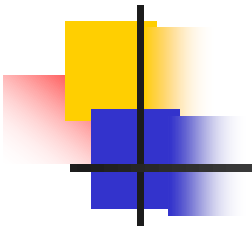
$Q =_{def} a||Q$

infinite terms and infinite behaviour

Equational Definition

$\underline{E \xrightarrow{a} E',\ A = E}$

$A \xrightarrow{a} E'$



Derivation Graph

The derivatives D(E) of an agent E are the agents derived from E by applying the proof rules

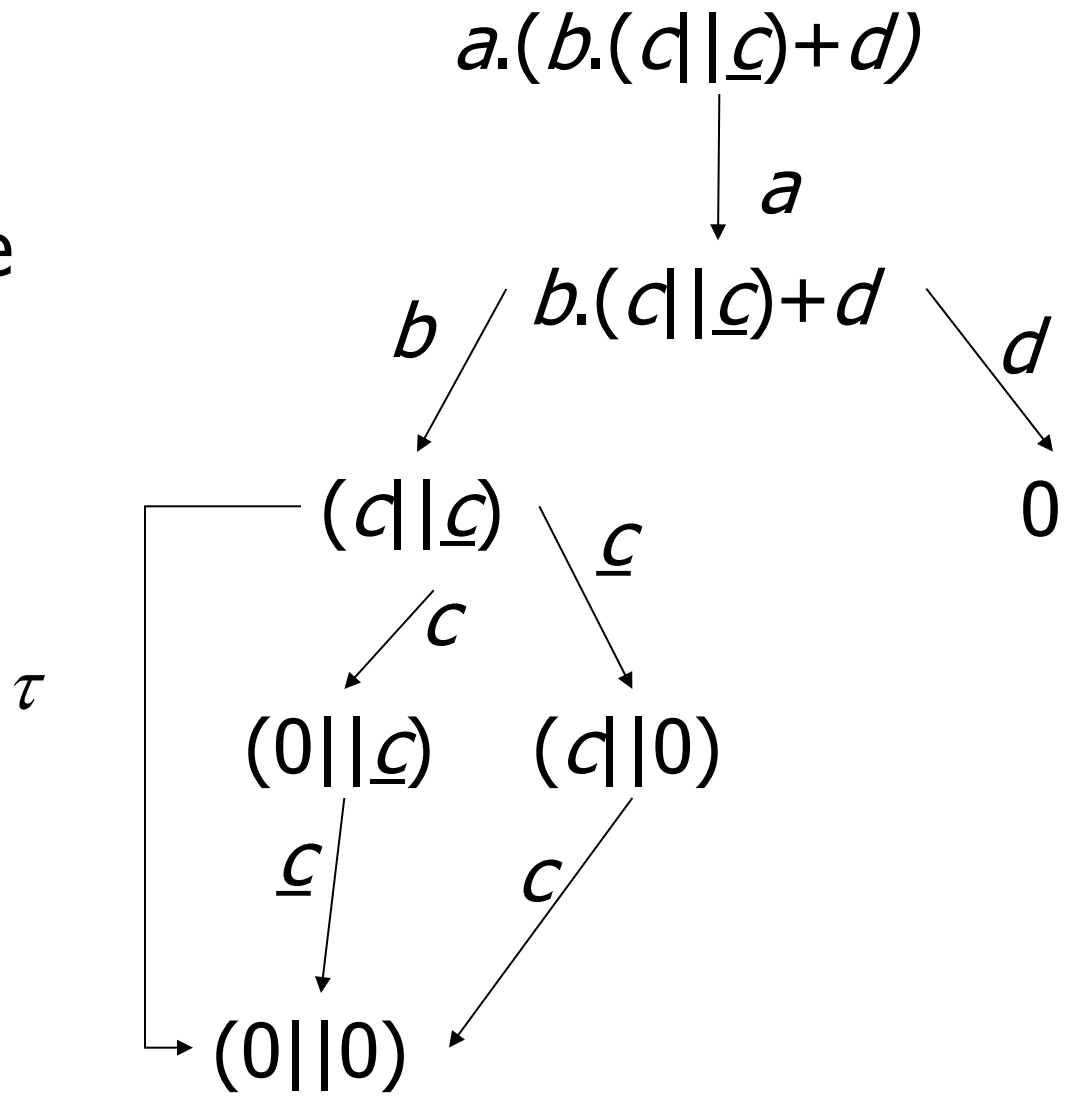
The derivation graph DG(E) is the graph (N,A) where N = D(E) and (n1,n2)\in A iff n1→n2 according to the proof rules

Derivations

Exercise: complete each arc with the name of the axiom/proof being applied

P=a.b.P

Q = a||Q

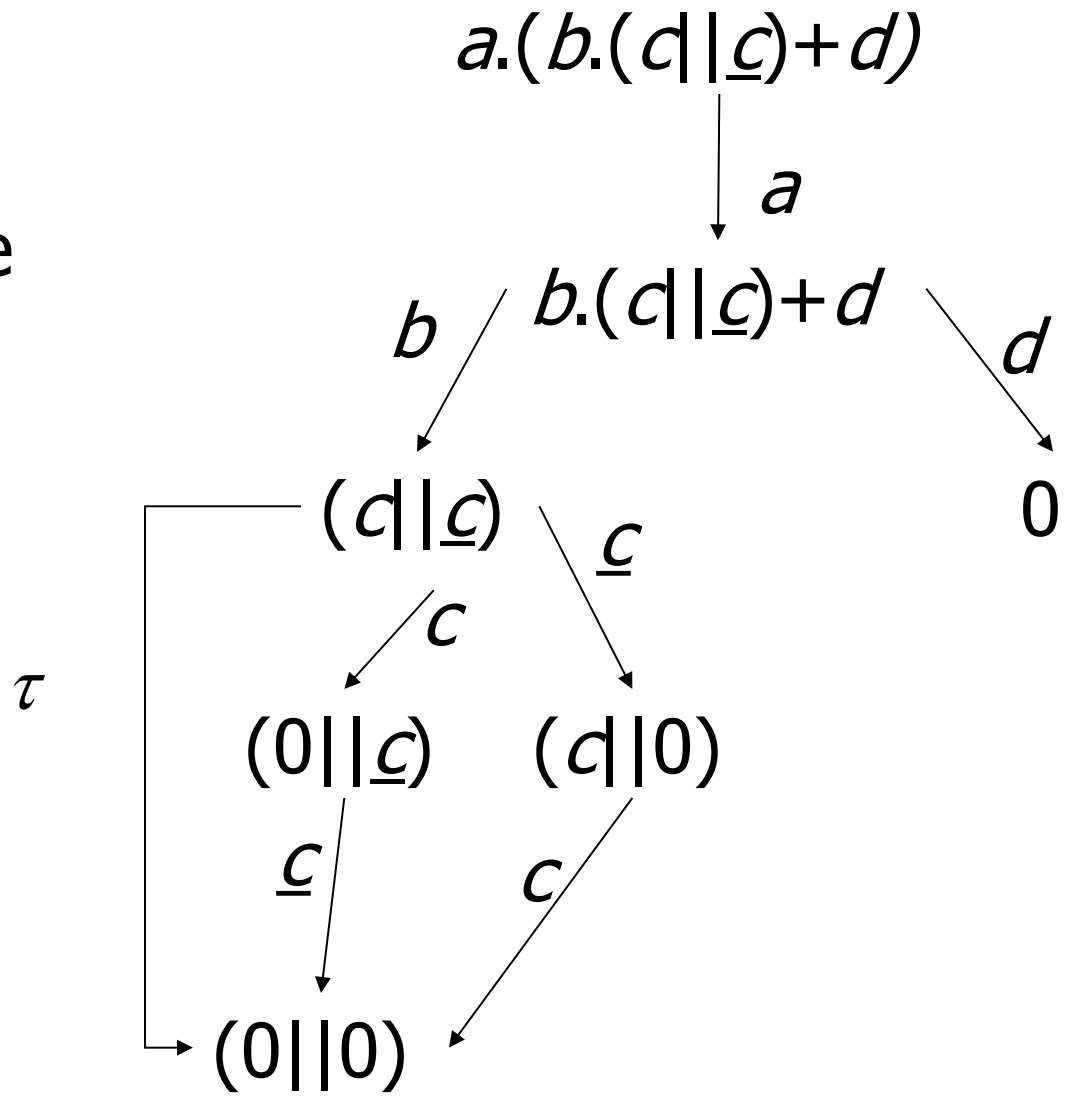


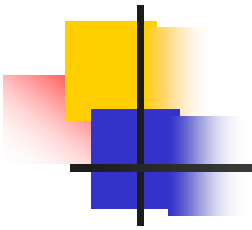
Derivations

Exercise: complete each arc with the name of the axiom/proof being applied

P=a.b.P

Q = a||Q





Example - buffer (dal Peled)

A one place buffer:

Empty1 = put.Full1

Full1 = get.Empty1

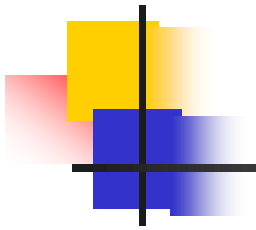
A two place buffer:

Empty2 = put.Half2

Half2 = put.Full2 + get.Empty2

Full2 = get.Half2

Example - buffer



Using the one place buffer:

Prod = loc.put.Prod Cons = get.loc.Cons

System = (Prod || Cons || Empty1) restricted over put and get

Using the two place buffer:

Prod = ….. Cons = …..

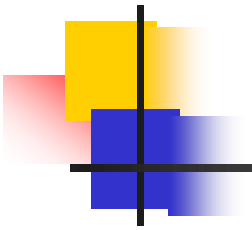
System = (Prod || Cons || Empty2) restricted over put and get

A two place buffer:

Empty2 = put.Half2

Half2 = put.Full2 + get.Empty2

Full2 = get.Half2



Example - buffer

A one place buffer:

Empty1 = put.Full1

Full1 = get.Empty1

A two place buffer using two one place buffer:

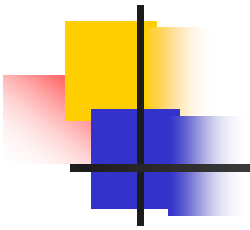
Empty2 = Empty1 || Empty1 correct?

A two place buffer:

Empty2 = put.Half2

Half2 = put.Full2 + get.Empty2

Full2 = get.Half2



Buffer

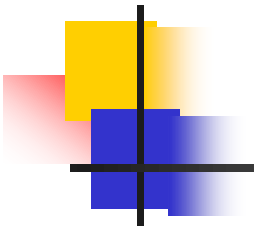
Solution and derivaton graph of

Sys = (Prod || Cons||E2)/{put, get}

con E2 = Empty1||Empty1

e Empty1 = put.Full1

Full1 = get.Empty1



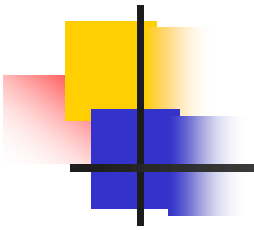
Buffer

Sys = (Prod || Cons||E2)/{put, get}

E2 = Empty1||Empty1

Empty1 = put.Full1

Full1 = get.Empty1



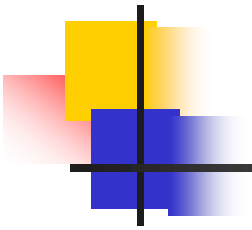
Buffer

Sys = (Prod || Cons||E2)/{put, get}

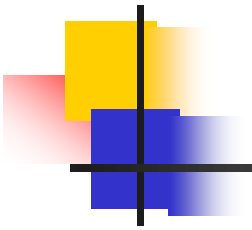
E2 = Empty1||Empty1

Empty1 = put.Full1

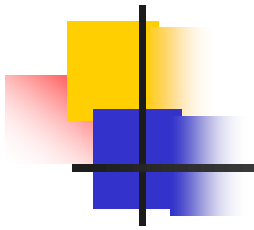
Full1 = get.Empty1



Example - buffer



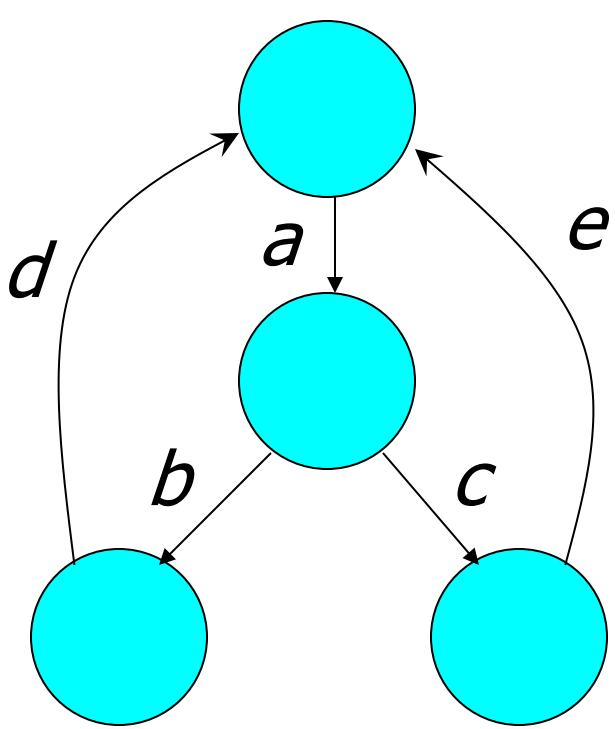
Example - buffer

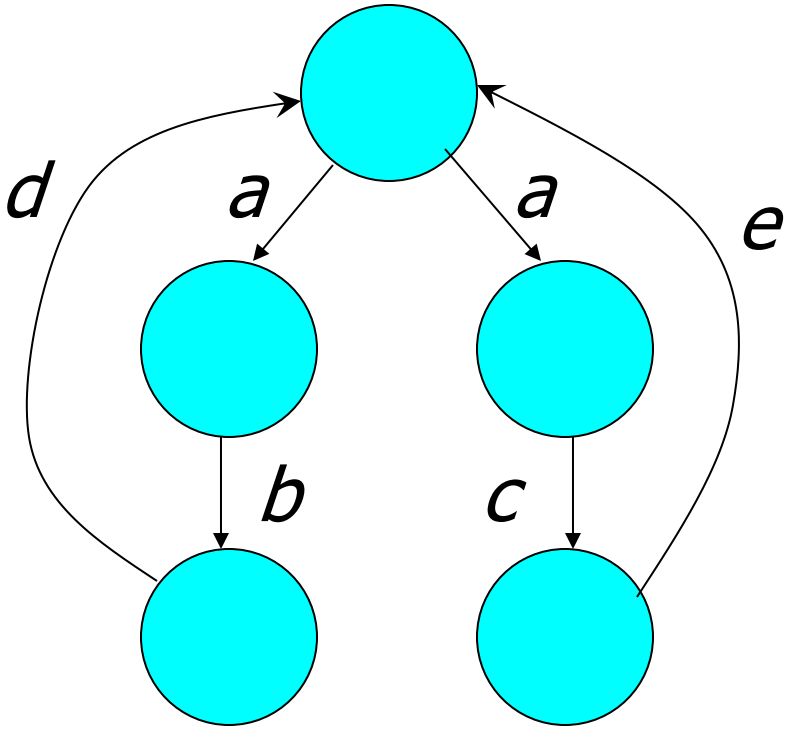


Example - buffer

Using Petri Nets…..

Different models may have the same set of executions!

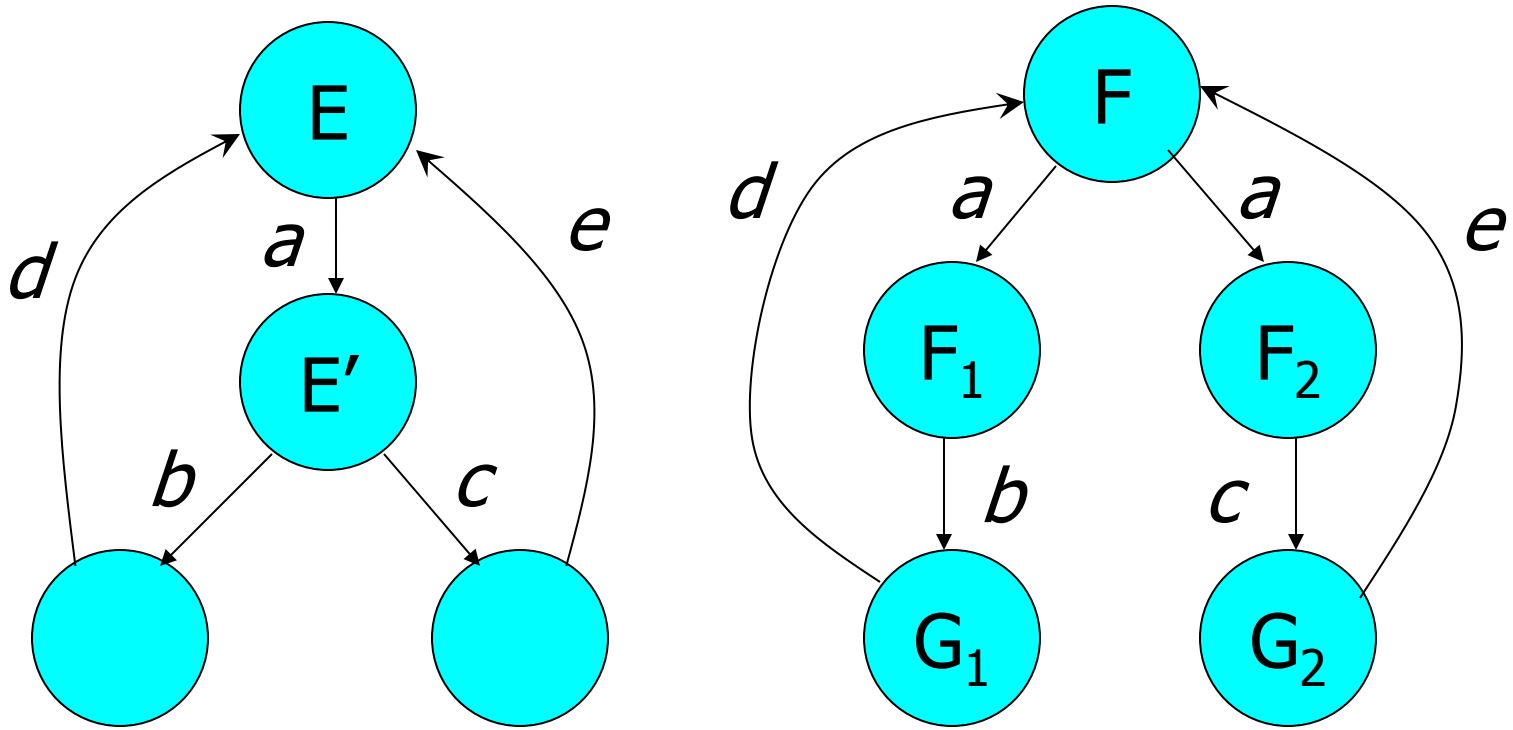




a-insert coin, *b*-press coffee, *c*-press milked coffee

d-obtain coffee, *e*-obtain milked coffee

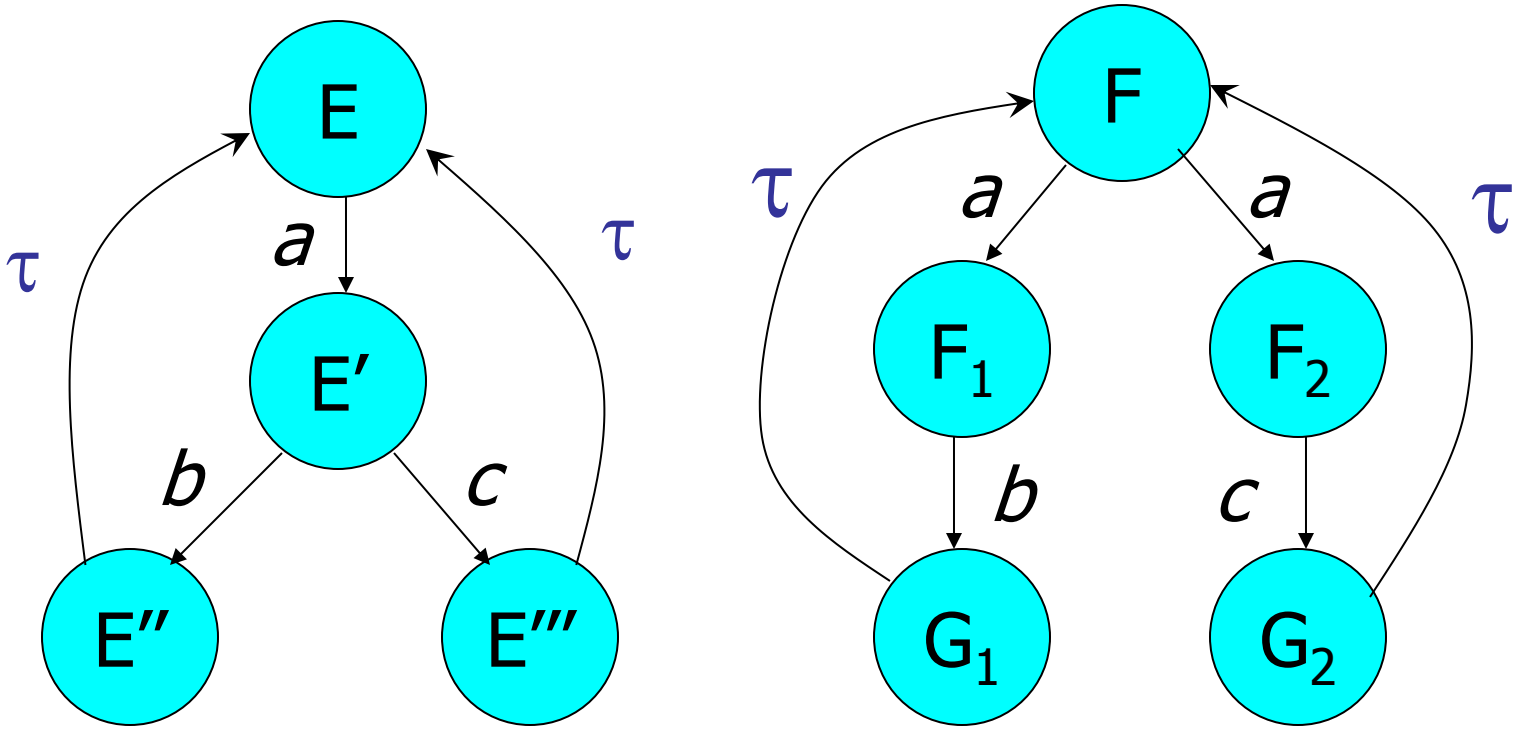
Actions: Act={*a*,*b*,*c*,*d*}∪{τ}.
Agents: E, E′, F, F1, F2, G1, G2, …



Agent E may *evolve* into agent E′.

Agent F may evolve into F_1 or F_2.

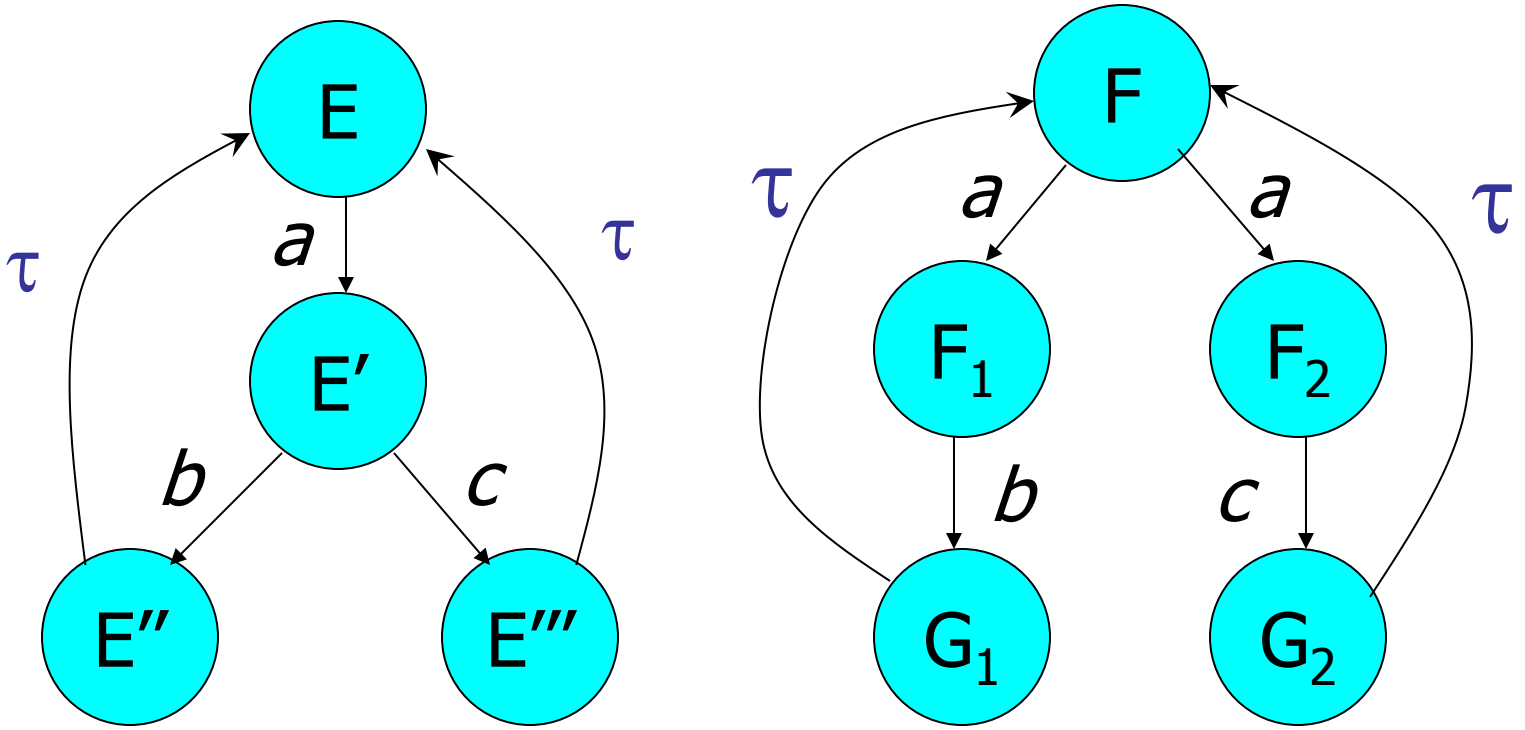
Events.



a-insert coin, *b*-get blue stamp, *c*-get red-stamp

… from a user point of view they might be equivalent

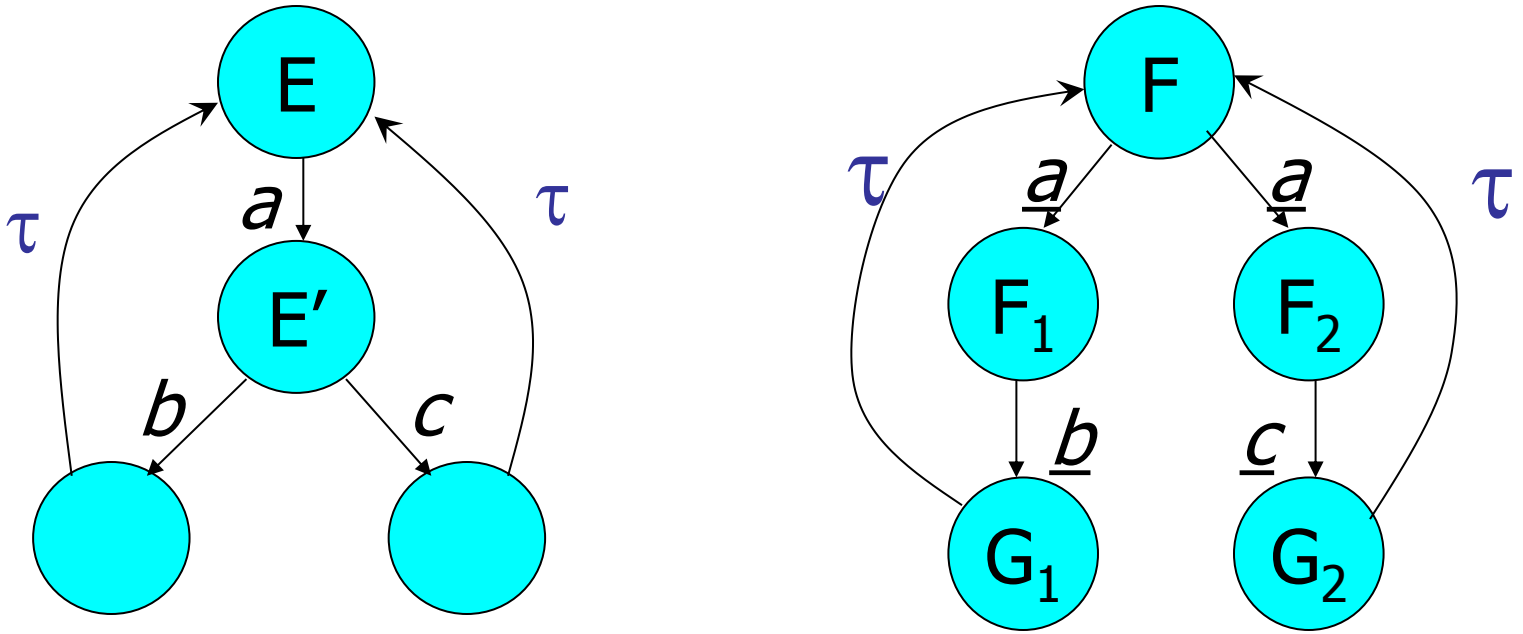
Events.



F—*a*→F_1, F—*a*→F_2, F1—*b*→G_1, F_2—*c*→G_2, G_1—τ→F, G_2—τ→F.

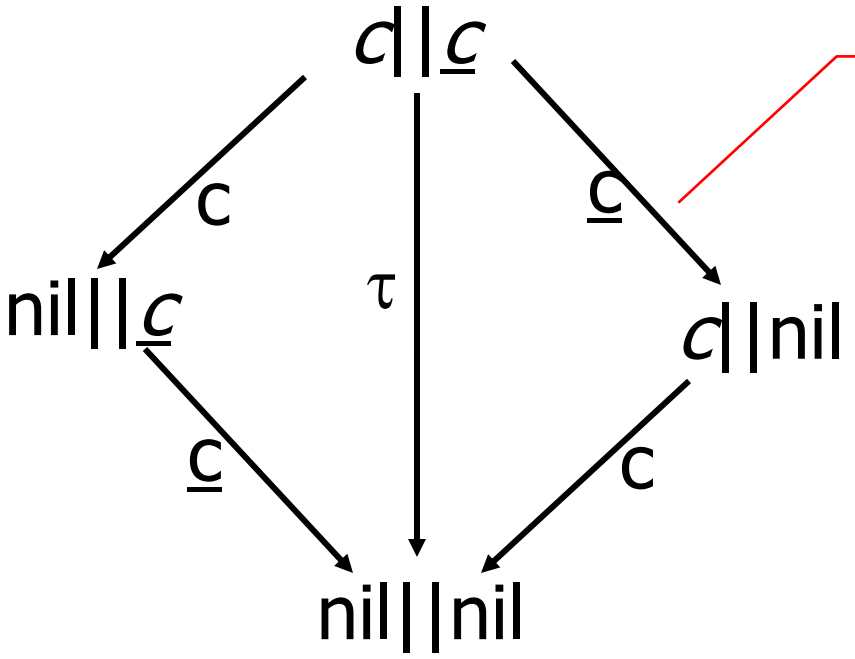
E—*a*→E′, ……

Actions and co-actions - interacting agents



For each action *a*, except for τ, there is a co-action $\underline{a}$. *a* and $\underline{a}$ interact (*a* input, $\underline{a}$ output). The coaction of $\underline{a}$ is *a*.

Concurrent Composition

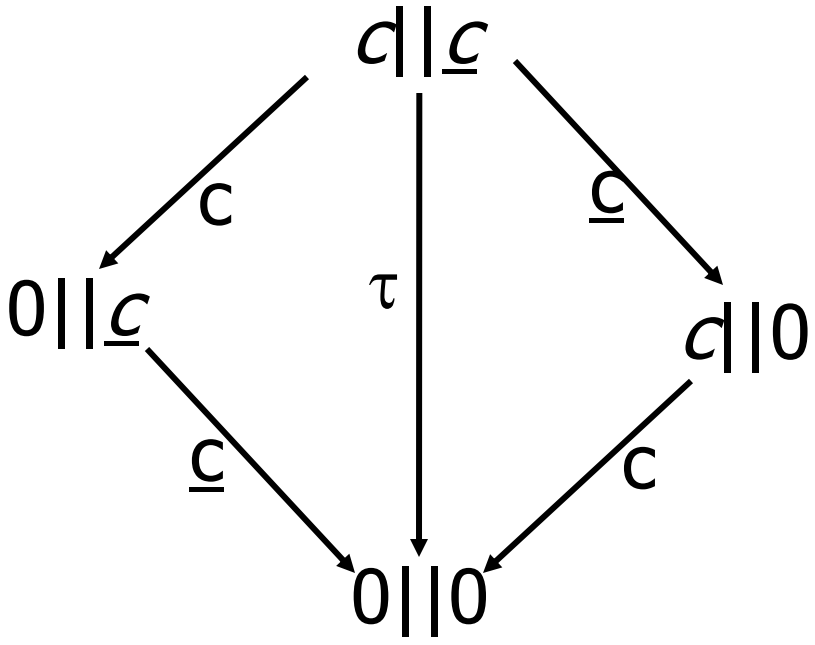


Derivation graph of an agent, D(c||c̄): (V,A), where V is the sets of all derivatives that can be derived from the agent c||c, through one or more application of the axioms or of the proof rules, and A is the set of arcs that represents the single application of a axiom or of a proof rule

Exercise: try to reproduce the same behavior with two nets, one can execute c and the other c̄.

Exercise: derivation graph of (c||c) and of (c||c̄)||c

Concurrent Composition



C||C

Derivation graph of (c||c)
and of (c||c̄)||c

$$\frac{E \xrightarrow{a} E'}{E||F \xrightarrow{a} E'||F} \qquad \frac{F \xrightarrow{a} F'}{E||F \xrightarrow{a} E||F'}$$

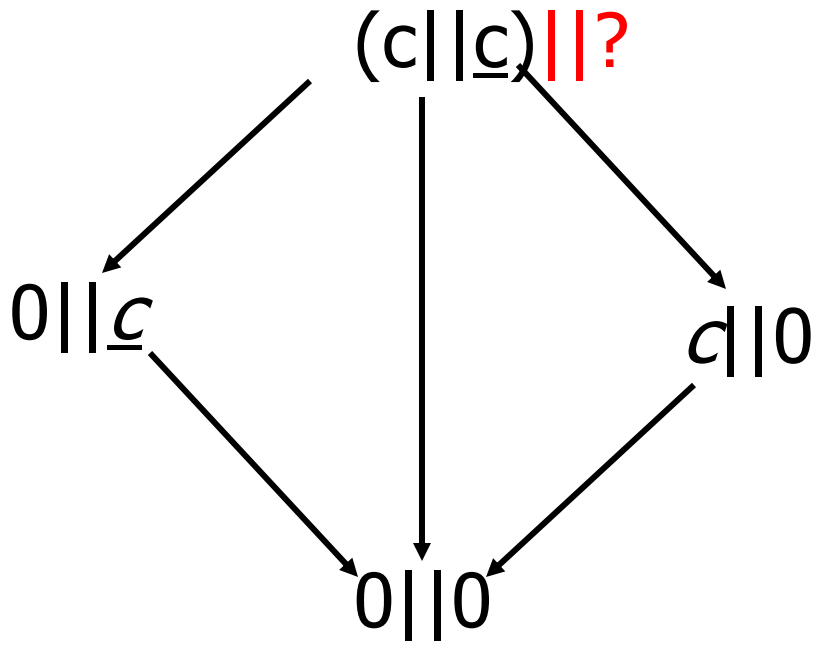
$$\frac{E \xrightarrow{a} E',\ F \xrightarrow{\underline{a}} F'}{E||F \xrightarrow{\tau} E'||F'}$$

C||C

Concurrent Composition

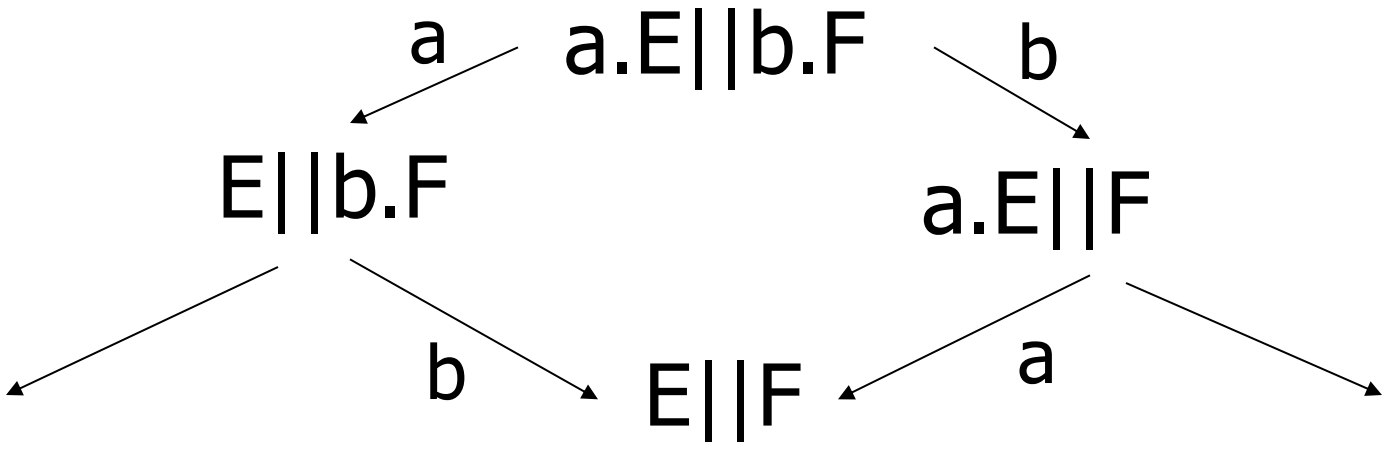
Derivation graph of (c||c)||c

Synchronization among more than two agents

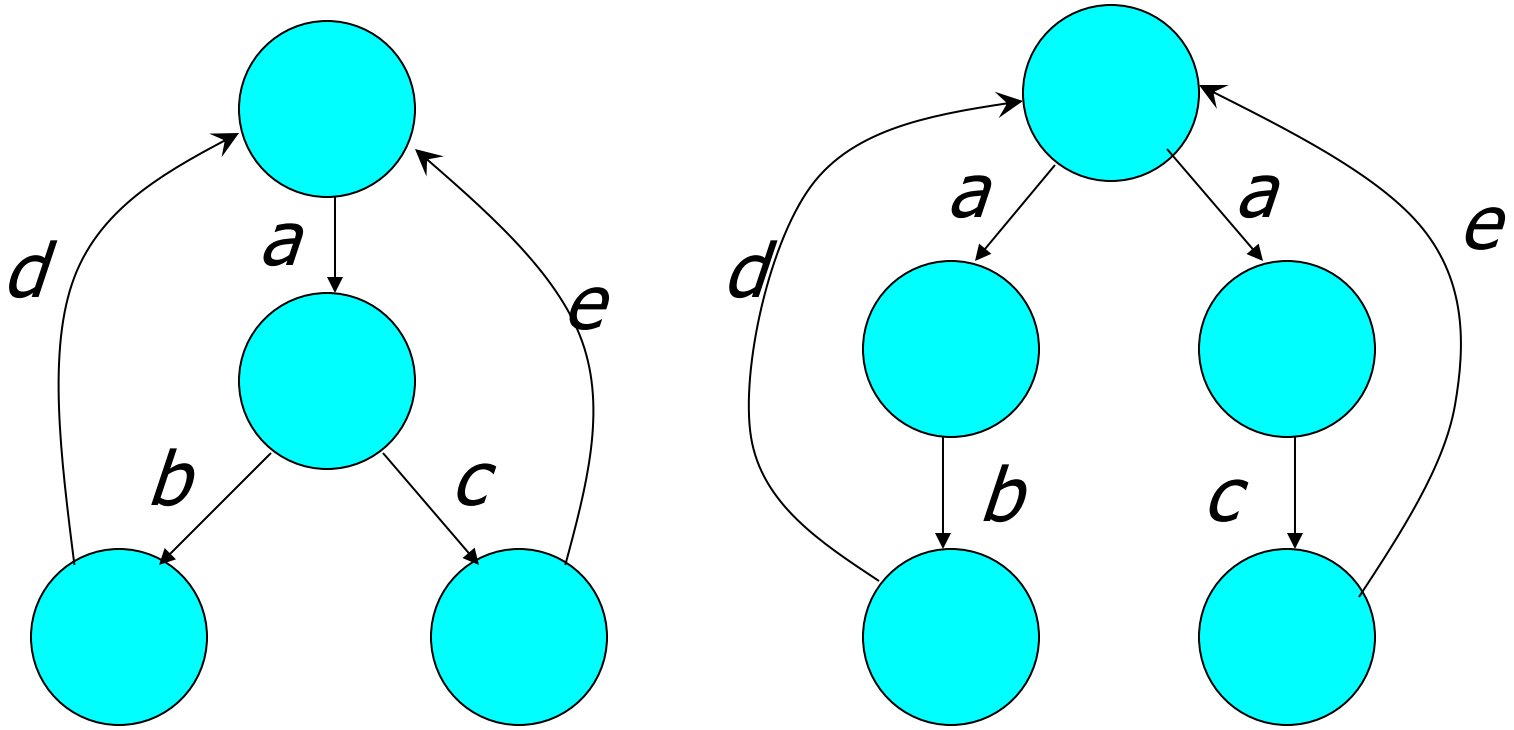


Try to have more than two agents (for example three) that synchronize on the sam action. Can I get this behaviour using CCS?

Examples

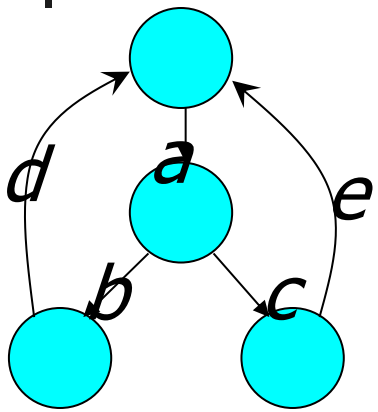


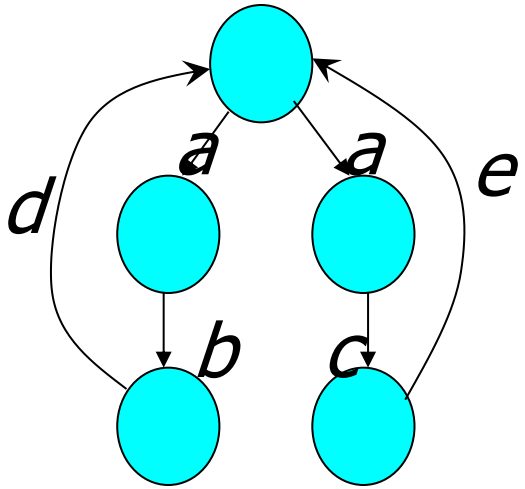
Equivalent Petri Nets models



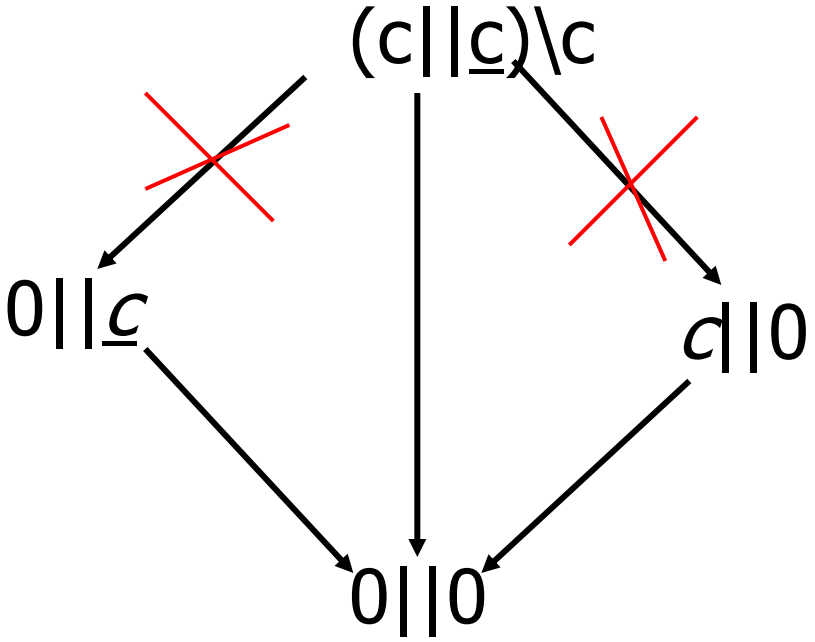
Can we consider these as reachability graphs of two different nets? Labels are transitions names? Or transition labels?

Equivalent Petri Nets models



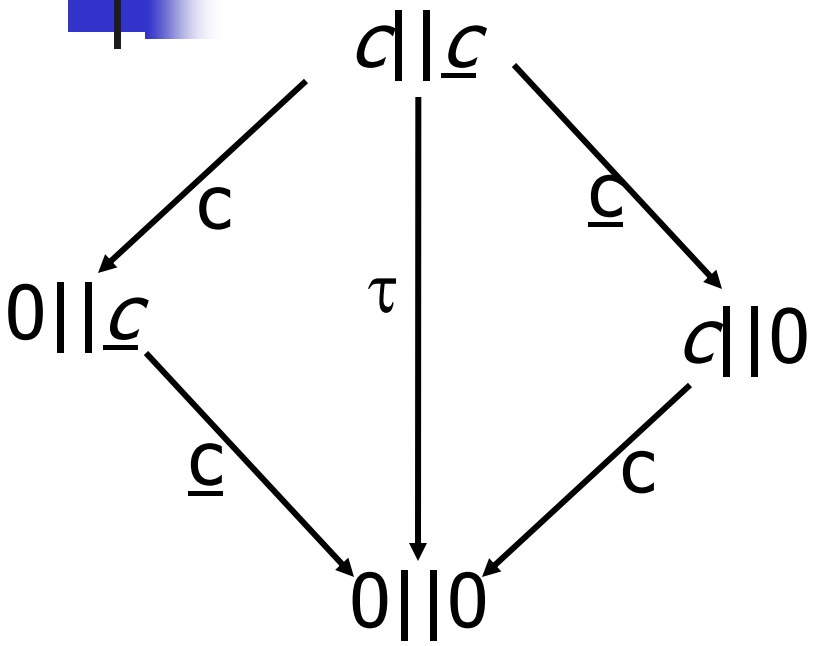


Consequence of restriction



Try to reproduce the same behavior with two nets, one can execute c and the other c̄.

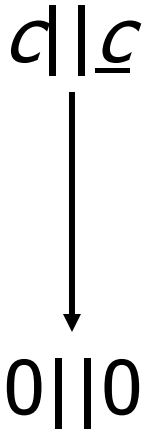
Concurrent Composition



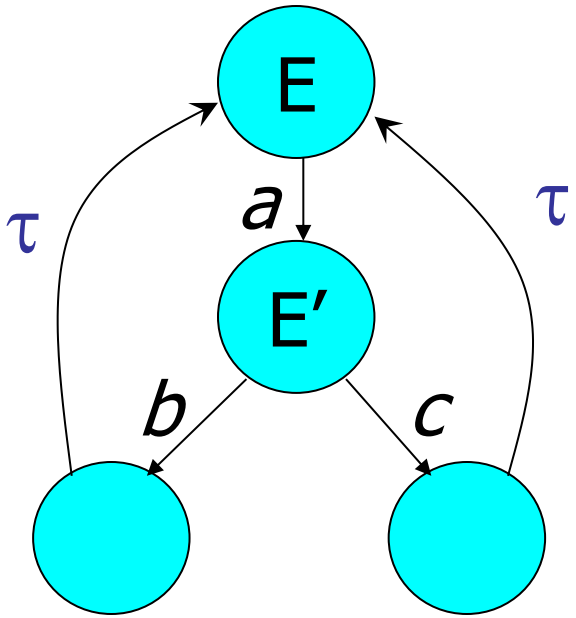
reproduce the same behavior with two nets, one can execute c and the other c̲.

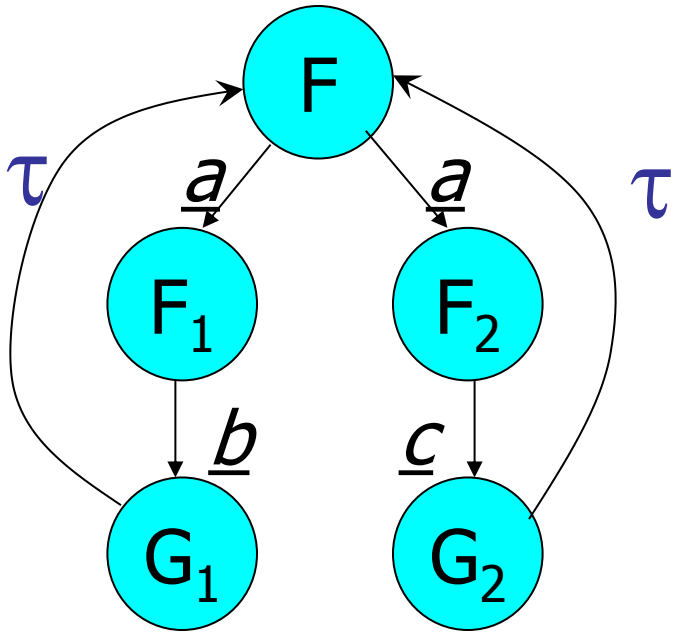
Concurrent Composition

Given two nets N1 and N2 of similar behaviour (one transition each, labelled c and $\underline{c}$ respectively) produce two process algebra terms such that the RG and the derivation graph are isomorphic and equal to the below



Equational Definition





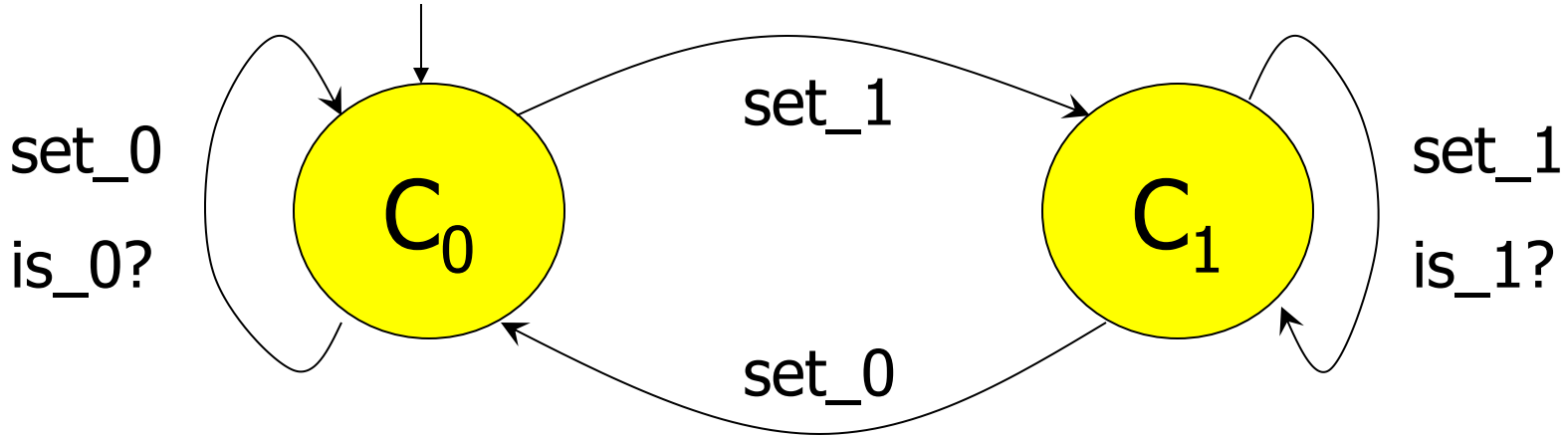
E=*a*.(*b*.τ.E+c.τ.E)

F=*a*.*b*.τ.F+*a*.*c*.τ.F

E—*a*→E′, A=E

A—*a*→E′

Modeling binary variable



$C_0 = is\_0? \cdot C_0 + set\_1 \cdot C_1 + set\_0 \cdot C_0$

$C_1 = is\_1? \cdot C_1 + set\_0 \cdot C_0 + set\_1 \cdot C_1$

Modeling binary variable

C_0=is_0? . C_0 + set_1 . C_1 + set_0 . C_0

C_1=is_1? . C_1 + set_0 . C_0 + set_1 . C_1

Dekker's algorithm

```
boolean c1 initially 1;
boolean c2 initially 1;
integer (1..2) turn initially 1;
```

```
P1::while true do
      begin
        non-critical section 1
        c1:=0;
        while c2=0 do
           begin
              if turn=2 then
                begin
                   c1:=1;
                   wait until turn=1;
                end
           end
        critical section 1
        c1:=1;
        turn:=2
     end.
```

```
P2::while true do
      begin
        non-critical section 2
        c2:=0;
        while c1=0 do
           begin
              if turn=1 then
                begin
                   c2:=1;
                   wait until turn=2;
                end
           end
        critical section 2
        c2:=1;
        turn:=1
     end.
```

Dekker's algorithm

```
boolean c1 initially 1;
boolean c2 initially 1;
integer (1..2) turn initially 1;
```

```
P1- p::while true do
      begin
        non-critical section 1
        c1:=0; wantp:= true
        while c2=0 wantq do
          begin
            if turn=2 then
              begin
                c1:=1; wantp:= false
                wait until turn=1;
                wantp:= true
              end
          end
        critical section 1
        c1:=1; turn:=2
        turn:=2; wantp:= false
      end.
```

Semantica variabili:

wantp == true → p vuole accedere alla critical section (CS)

wantq == true → q vuole accedere alla critical section (CS)

turn: chi ha il turno, e' una variabile che serve a redimere i casi in cui ambedue i processi facciano una richiesta (cioe' quando sia wantp che wantq sono a true). turn==1 → processo p

turn==2 → processo q

Logica dell'algoritmo: quando ambedue i processi vogliono accedere (wantp e wantq a true) entrano ambedue nel while, ma quello che ha turn in suo favore (supponiamo sia q) "passa" per primo, mentre l'altro aspetta sulla wait. Per permettere a q di uscire dal while p deve mettere a false la sua richiesta di accesso (wantp a false appena prima della wait). Quando poi turn volge a suo favore si sblocca la wait e p rimette wantp a true, in modo da poter eventualmente accedere alla CS (se nel frattempo wantq e' andato a false) avendo la variabile wantp correttamente settata a true (senno' q potrebbe, mentre p e' in regione critica, ritornare ad eseguire le istruzioni non CS e poi settare wantq a zero e, trovando wantp false, uscire immediatamente dal while e accedere a sua volta alla CS, che e' il problema che abbiamo riscontrato in classe).

Non penso che l'ordine delle ultime due istruzioni sia rilevante, ma non ho controllato

Dekker's algorithm

```
boolean c1 initially 1;
boolean c2 initially 1;
integer (1..2) turn initially 1;
```

```
P1- p::while true do
      begin
        non-critical section 1
        c1:=0; wantp:= true
        while c2=0 wantq do
           begin
              if turn=2 then
                begin
                   c1:=1; wantp:= false
                   wait until turn=1;
                   wantp:= true
                end
           end
        critical section 1
        c1:=1; turn:=2
        turn:=2; wantp:= false
      end.
```

```
P2- q::while true do
      begin
        non-critical section 2
        c2:=0; wantq:= true
        while c1=0 wantp do
           begin
              if turn=1 then
                begin
                   c2:=1; wantq:= false
                   wait until turn=2;
                   wantq:= true
                end
           end
        critical section 2
        c2:=1; ; turn:=1
        turn:=1; wantq:= false
      end.
```

Dekker's algorithm

```
P1::while true do
      begin
        non-critical section 1
        c1:=0;
        while c2=0 do
          begin
            if turn=2 then
              begin
                c1:=1;
                wait until turn=1;
              end
          end
        critical section 1
        c1:=1;
        turn:=2
      end.
```

Translate into process algebra and into Petri nets

Build the derivation graph of the process algebra term and the RG of the Petri net and compare

CSP-like process algebra

CSP defined by Hoare in '83

Similar to CCS but no notion of action and co-action

$$P ::= Nil \mid a.P \mid P + P \mid P \|_S P \mid P/L \mid A$$

Parallel Composition

$$\frac{P \xrightarrow{a} P'}{P \|_S Q \xrightarrow{a} P' \|_S Q} \; a \notin S \qquad \frac{Q \xrightarrow{a} Q'}{P \|_S Q \xrightarrow{a} P \|_S Q'} \; a \notin S$$

$$\frac{P \xrightarrow{a} P' \;\; Q \xrightarrow{a} Q'}{P \|_S Q \xrightarrow{a} P' \|_S Q'} \; a \in S$$

CSP-like process algebra

Consequences of the new proof rule for the parallel composition :

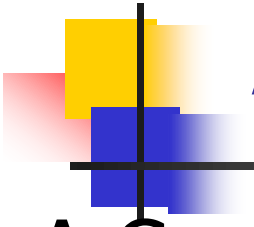
$$(c.0||_{\{c\}}c.0)||_{\{c\}}c.0$$

CSP-like process algebra

Consequences of the new proof rule for the parallel composition :

$$(c.0||c.0)||_{\{c\}}c.0$$

Derivation graph



Algebra

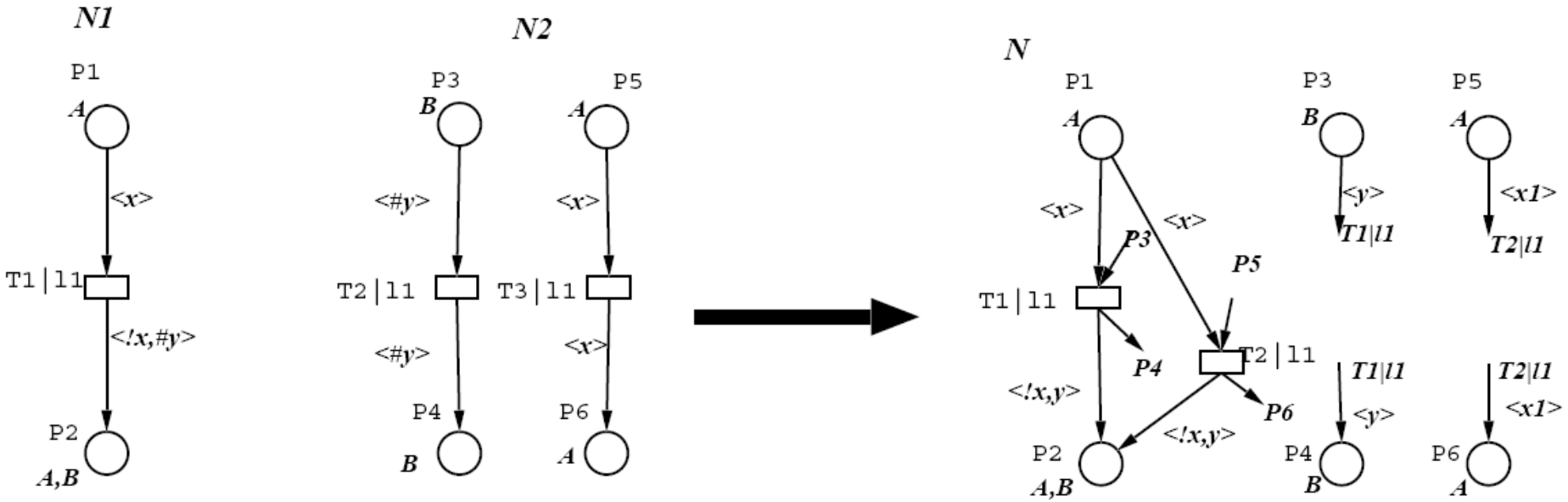
A C program that takes a set of LGSPN and LSWN nets and it superposes them on transition/place (subset of) labels

It allows non-injective multilabelling (cross product of trans. and places and n-process synchronization)

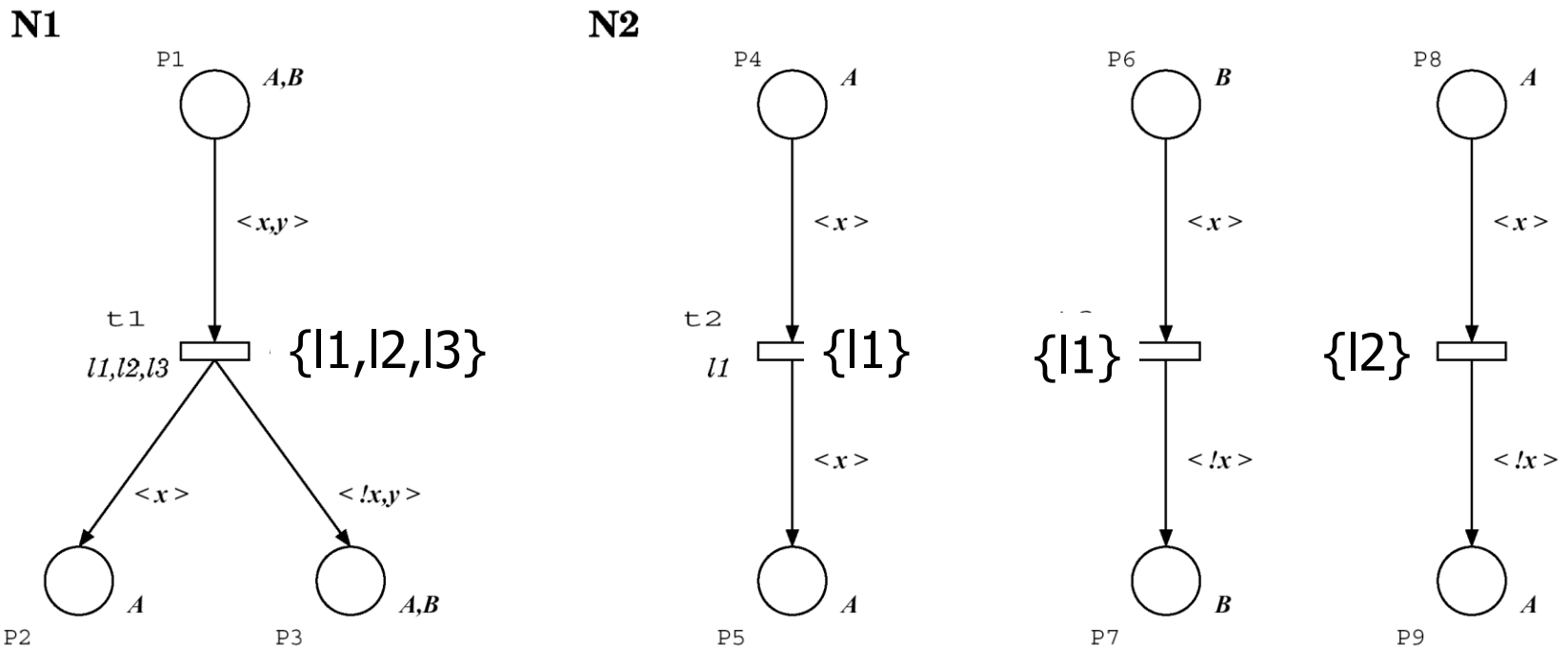
Algebra

A C program that takes a set of LGSPN and LSWN nets and it superposes them on transition/place (subset of) labels

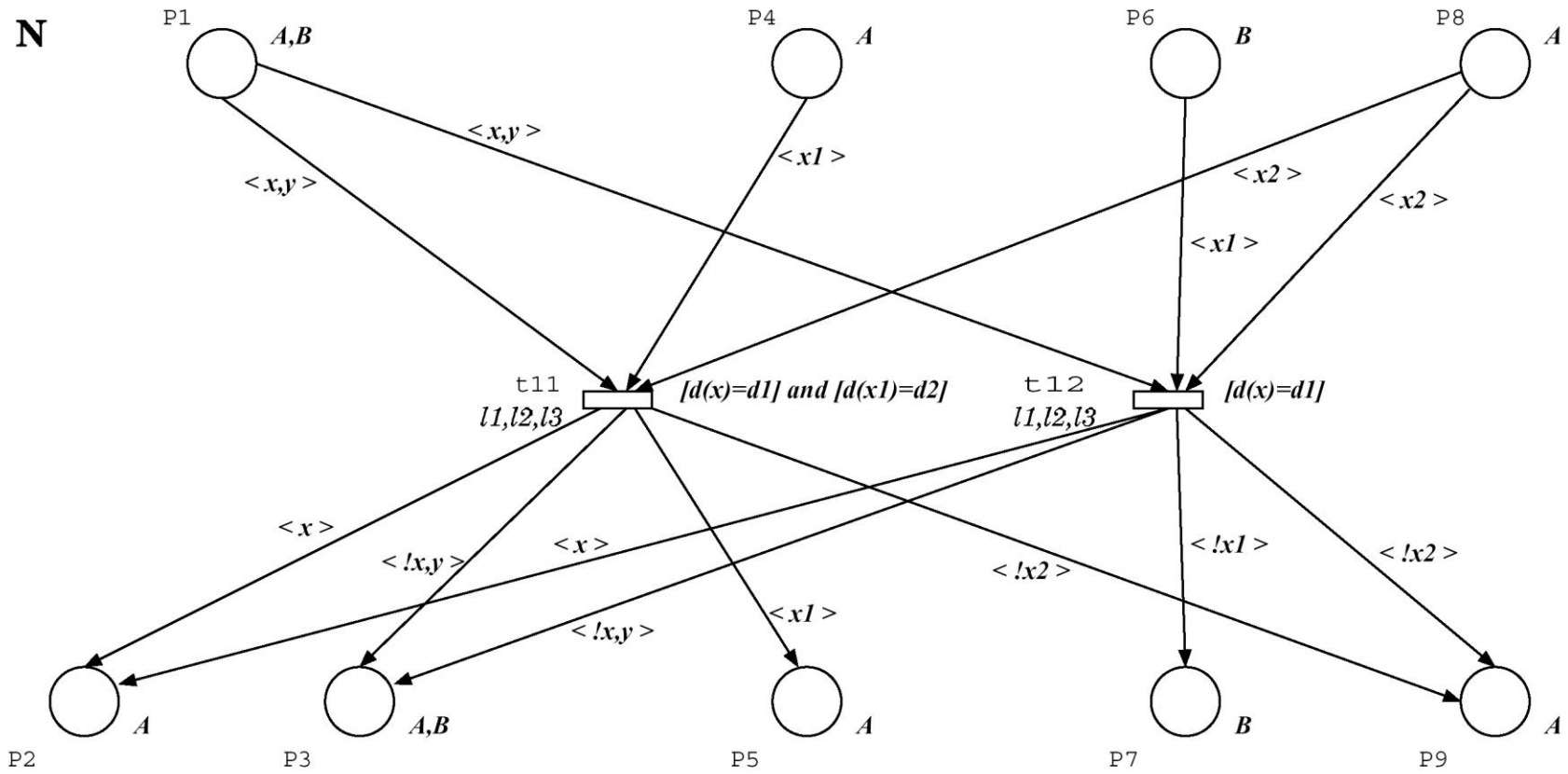
$$N1 \;||_{\{l1\}}\; N2$$



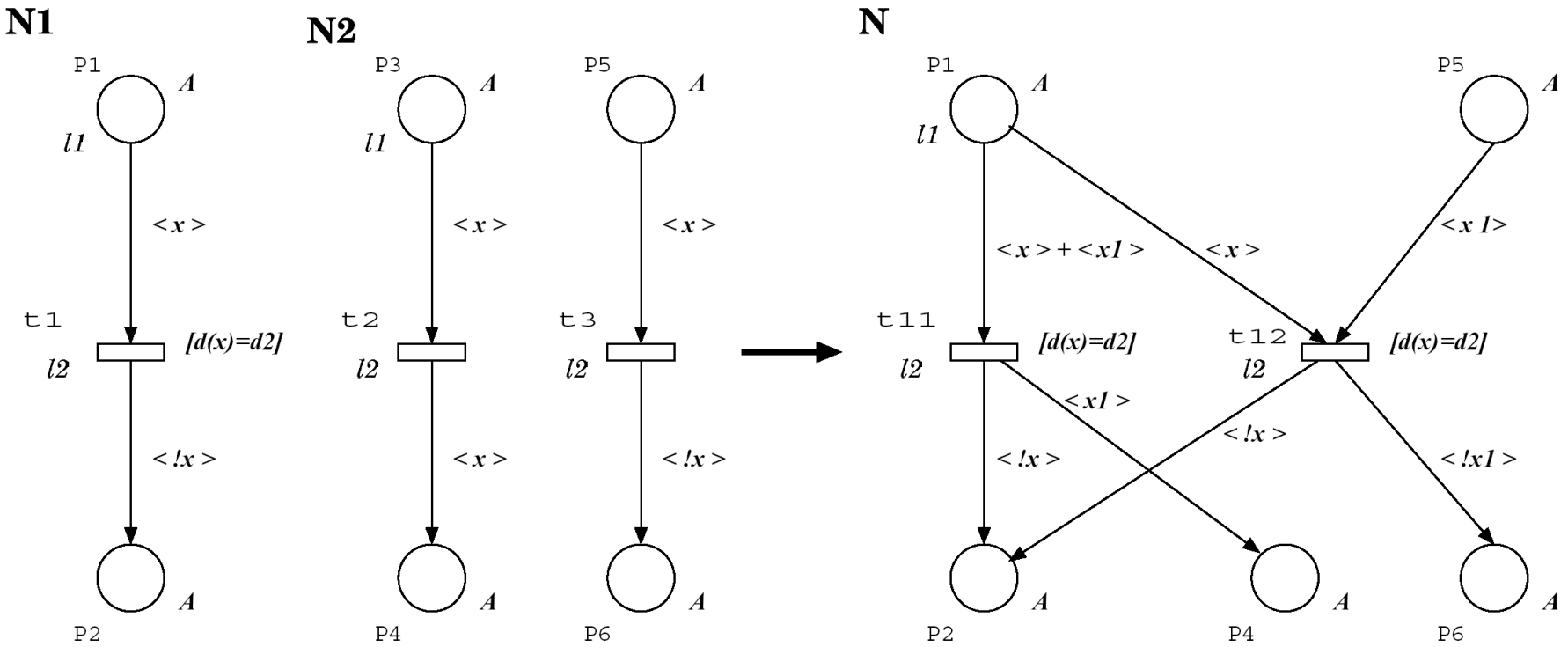
Algebra: transition superposition



Algebra: transition superposition



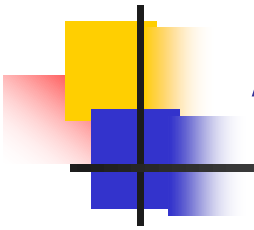
Algebra: place and transition superposition



Algebra script

```
>> algebra net1 net2 op labels net
                              [plac. Dx Dy]
```

- `net1 net2` input nets
- `op =` t or p for type of superposition
- `labels` to be considered by `op`
- `net` resulting net
- `[plac. Dx Dy]` placement of `net1 net2` in `net`



Algebra

Additional remove function to eliminate

- # in tags (used to avoid renaming of variables)
- labels (solvers crash if tags have a "|" char)

Modeling binary variable

C_0=is_0? . C_0 + set_1 . C_1 + set_0 . C_0

C_1=is_1? . C_1 + set_0 . C_0 + set_1 . C_1

Dekker's algorithm

```
boolean c1 initially 1;
boolean c2 initially 1;
integer (1..2) turn initially 1;
```

```
P1::while true do
      begin
        non-critical section 1
        c1:=0;
        while c2=0 do
           begin
              if turn=2 then
                begin
                   c1:=1;
                   wait until turn=1;
                end
           end
        critical section 1
        c1:=1;
        turn:=2
     end.
```

```
P2::while true do
      begin
        non-critical section 2
        c2:=0;
        while c1=0 do
           begin
              if turn=1 then
                begin
                   c2:=1;
                   wait until turn=2;
                end
           end
        critical section 2
        c2:=1;
        turn:=1
     end.
```

Dekker's algorithm

boolean c1 initially 1;

boolean c2 initially 1;

integer (1..2) turn initially 1;

```
P1::while true do
      begin
        non-critical section 1
        c1:=0;
        while c2=0 do
           begin
              if turn=2 then
                begin
                   c1:=1;
                   wait until turn=1;
                end
           end
        critical section 1
        c1:=1;
        turn:=2
     end.
```