

# Esercizio Definizioni Matematiche

Francesco Mecca

May 22, 2020

## 1 Insiemi

### 1.1 Numeri naturali

I numeri naturali sono i numeri appartenenti all'insieme dei numeri naturali  $\mathbb{N}$ , ovvero tutti i numeri maggiori o uguali a 0.

Possiamo definire i numeri naturali utilizzando la rappresentazione di Von Neumann:

- definiamo la funzione *successore*( $n$ ) come:

$$\text{successore}(n) = n \cup \{n\}$$

- $0 = \emptyset$
- $1 = 0 \cup \{0\} = \{\emptyset\}$
- $2 = 1 \cup \{1\} = \{0, 1\}$
- $3 = 2 \cup \{2\} = \{0, 1, 2\}$
- $n = n-1 \cup \{n-1\} = \{0, 1, 2, \dots, n-1\}$

### 1.2 Numeri interi

I numeri interi sono i numeri appartenenti all'insieme dei numeri interi  $\mathbb{Z}$ , ovvero tutti i numeri il cui valore assoluto è un numero naturale.

Possiamo rappresentare intuitivamente l'insieme dei numeri interi  $\mathbb{Z}$  come  $\{n \mid \exists(a,b) \in \mathbb{N} \times \mathbb{N}, n = a-b\}$

### 1.3 Numeri razionali

I numeri razionali sono i numeri appartenenti all'insieme dei numeri razionali  $\mathbb{Z}$ , ovvero tutti i numeri rappresentabili tramite un numero razionale o come il limite di una sequenza di numeri razionali che non si ripete e non termina (numeri irrazionali).

### 1.4 Intersezione

L'intersezione fra due insiemi è a sua volta un insieme contenente gli elementi in comune fra i due insiemi:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

### 1.5 Unione

L'unione fra due insiemi è a sua volta un insieme contenente gli elementi dei due insiemi:

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

### 1.6 Differenza

La differenza fra due insiemi è a sua volta un insieme contenente tutti gli elementi presenti nell'insieme a sinistra della differenza ma non contenuti nell'insieme a destra:

$$A \setminus B = \{x \mid x \in A \wedge x \notin B\}$$

### 1.7 Insieme Potenza

L'insieme potenza di un insieme  $S$ ,  $\wp(S)$ , anche detto power set di  $S$  è l'insieme che contiene tutti i sottoinsiemi di  $S$ .

## 1.8 Complemento di un insieme

Il complemento di un insieme è a sua volta un insieme che contiene tutti gli elementi che non appartengono all'insieme di partenza:

$$A^c = \{a \mid a \notin A\}$$

## 1.9 Insieme contenuto

Un insieme  $A$  si dice contenuto in  $B$  se tutti gli elementi di  $A$  sono a loro volta elementi di  $B$ :

$$A \subseteq B \text{ iff } \forall a \in A, a \in B$$

## 1.10 Insieme strettamente contenuto

Un insieme  $A$  si dice strettamente contenuto in  $B$  se tutti gli elementi di  $A$  sono a loro volta elementi di  $B$  ma ci sono degli elementi di  $B$  che non appartengono ad  $A$ :

$$A \subset B \text{ iff } (\forall a \in A, a \in B) \wedge (\exists b \in B \mid b \notin A)$$

## 1.11 Prodotto Cartesiano

Il prodotto cartesiano di due insiemi è un insieme contenente tutte le coppie ordinate di cui il primo elemento appartiene al primo insieme ed il secondo elemento al secondo insieme:

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$$

## 1.12 Arietà $n$

Si definisce arietà di una relazione  $R$  il numero di insiemi a cui si applica quella relazione. Se una relazione ha arietà  $n$ :

$$R \subseteq A_1 \times A_2 \times \dots \times A_n$$

### 1.13 Relazione binaria

Si definisce una relazione  $R$  binaria quando  $R$  ha arietà 2:

$$R \subseteq A_1 \times A_2$$

### 1.14 Proprietà riflessiva

Considerato un insieme  $A$  e una relazione  $R$ , diciamo che  $R$  è una relazione riflessiva se:

$$\forall a \in A, aRa$$

### 1.15 Proprietà simmetrica

Considerato un insieme  $A$  e una relazione binaria  $R$ , diciamo che  $R$  è una relazione simmetrica se:

$$\forall a, b \in A, aRb \Leftrightarrow bRa$$

### 1.16 Proprietà transitiva

Considerato un insieme  $A$  e una relazione binaria  $R$ , diciamo che  $R$  è una relazione transitiva se:

$$\forall a, b, c \in A, aRb \wedge bRc \rightarrow aRc$$

### 1.17 Relazione di equivalenza

Una relazione binaria che è allo stesso tempo riflessiva, simmetrica e transitiva si dice relazione d'equivalenza.

### 1.18 Chiusura transitiva

Considerato un insieme  $A$  e una relazione binaria  $R$ , definiamo chiusura transitiva la più piccola relazione transitiva  $R^+$  sull'insieme  $A$  che contiene  $R$ :

$$R \subseteq R^+ \wedge (\forall T, R \subseteq T \rightarrow R^+ \subseteq T \text{ (} R^+ \text{ is minimal)})$$

Se la relazione  $R$  è transitiva, allora  $R=R^+$

### 1.19 Funzione

Definiamo funzione una relazione fra due insiemi  $A$  e  $B$  che associa un elemento dell'insieme  $A$  ad esattamente un elemento dell'insieme  $B$ :

$$f: X \mapsto Y$$

### 1.20 Funzione di arietà $n$

Possiamo definire una funzione di arietà  $n$  su un insieme  $S$  come:

$$f: S^n \mapsto S$$

### 1.21 Funzione iniettiva

Una funzione  $f: X \mapsto Y$  si dice iniettiva quando presi due elementi dell'insieme  $X$ , se la loro immagine è uguale ( $f(x)$ ), allora i due elementi sono uguali:

$$\forall x, x' \in X, f(x) = f(x') \rightarrow x = x'$$

### 1.22 Funzione suriettiva

Una funzione  $f: X \mapsto Y$  si dice suriettiva quando preso qualunque elemento di  $Y$ , questo ha una controimmagine  $x$  in  $X$ :

$$\forall y \in Y, \exists x \in X \quad f(x) = y$$

### 1.23 Funzione biettiva

Chiamiamo una funzione biettiva quando è allo stesso tempo iniettiva e suriettiva.

## **2 Linguaggi**

### **2.1 Alfabeto**

Un alfabeto è un insieme i cui membri sono simboli (che includono lettere, caratteri e numeri). Se  $L$  è un linguaggio formale, ossia un set finito o infinito di stringhe di finita lunghezza, allora l'alfabeto di  $L$ , indicato con  $\Sigma$ , è l'insieme di tutti i simboli che possono comparire in una qualunque stringa di  $L$ .

### **2.2 Stringa**

Una stringa è una sequenza finita di simboli di un alfabeto.

### **2.3 Lettera**

Una lettera di una string è un simbolo dell'alfabeto.

### **2.4 Stringa vuota**

Una stringa vuota è una stringa di lunghezza zero, anche detta  $\varepsilon$ .

### **2.5 Concatenazione**

La concatenazione di stringhe è l'operazione di unione dei caratteri di due stringhe preservando il loro ordine.

### **2.6 Ripetizione**

Si dice ripetizione l'operazione di concatenazione di una stringa con  $n$  copie di sé stessa.

### **2.7 Prefisso**

Si dice prefisso di una stringa la sottostringa che appare all'inizio della stringa.

## 2.8 Suffisso

Si dice suffisso di una stringa la sottostringa che appare alla fine della stringa.

## 3 Grafi

Un grafo è una coppia ordinata  $G = (V,E)$  che comprende un insieme  $V$  di vertici e un insieme  $E$  di coppie  $(e,v)$ .

### 3.1 Grafo diretto

Un grafo diretto è un grafo in cui gli archi hanno orientamento.

### 3.2 Grafo indiretto

Un grafo indiretto o semplice è un grafo in cui gli archi non hanno orientamento, ovvero:

$$\forall x,y \in V, (x,y) = (y,x)$$

### 3.3 Grafo bipartito

Un grafo si dice bipartito quando l'insieme di vertici  $V$  può essere diviso in due insiemi disgiunti e indipendenti  $W$  e  $X$ , di modo che ogni arco connetta un vertice in  $W$  con un vertice in  $X$  e si scrive  $G = (W,X,E)$ :

$$V = W \cup X \wedge W \cap X = \emptyset$$

### 3.4 Nodo sorgente

Un nodo si dice sorgente quando il numero di archi in ingresso è 0.

### 3.5 Nodo destinazione

Un nodo si dice destinazione quando il numero di archi in uscita è 0.

### 3.6 Funzione di etichettatura per archi e nodi

In un generico grafo  $G$ , è possibile definire funzioni di etichettatura o di colorazione dei nodi come, dato un insieme di etichette  $S$ :

$f: V \rightarrow S$  Definendo un insieme di

### 3.7 Cammino

Si dice cammino una sequenza di archi che collega una sequenza di vertici distinti.

### 3.8 Ciclo

Si definisce ciclo un cammino in cui il primo e l'ultimo vertice coincidono mentre tutti gli altri vertici si ripetono al più una volta.

### 3.9 Lunghezza del cammino

Si definisce lunghezza il numero di archi che compongono un cammino. In un grafo pesato la lunghezza di un cammino è costituita dalla somma del peso di ogni arco che lo compone. Un cammino in un grafo è una sequenza finita o infinita di archi che collegano una sequenza di vertici distinti l'uno dall'altro. Un cammino di lunghezza  $k$  è rappresentato da una sequenza alternata di  $k$  vertici ed archi.  $v_0, e_0, v_1, e_1, \dots, v_{k-1}, e_{k-1}, v_k$

### 3.10 Grafi fortemente connesso

Un grafo diretto si dice fortemente connesso se ogni vertice è raggiungibile da ogni altro vertice.

### 3.11 Componenti fortemente connesse

Si dicono componenti fortemente connesse le partizioni di un grafo diretto che sono fortemente connesse.

### 3.12 BSCC - Bottom Strongly Connected Component

Una componente fortemente connessa si dice BSCC quando nessun vertice al di fuori della BSCC è raggiungibile.

### 3.13 Albero

Si dice albero un grafo indiretto in cui ogni coppia di vertici è connessa da solo un arco. Ogni grafo indiretto, connesso e aciclico è un albero.

### 3.14 In e out degree di un nodo

Si dice in degree,  $indeg^-(v)$ , di un nodo il numero di archi entranti in quel nodo. Si dice out degree,  $outdeg^+(v)$ , di un nodo il numero di archi uscenti da quel nodo.

## 4 Matrici

Una matrice è un vettore bidimensionale di numeri o altri oggetti. La dimensione  $n \times m$  è data dal numero di righe  $n$  e il numero di colonne  $m$ .

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} = (a_{ij}) \in R^{m \times n}$$

### 4.1 Somma

La somma  $A+B$  di due matrici  $A, B$  è definito come:

$$(A+B)_{ij} = A_{ij} + B_{ij}$$

### 4.2 Prodotto

Definiamo il prodotto scalare di una matrice  $A$  per un fattore  $c$  come:

$$(cA)_{ij} = c \cdot A_{ij}$$

Definiamo il prodotto fra una matrice A di dimensione  $|n_a \times m_a|$  e una matrice B di dimensione  $|n_b \times m_b|$  quando  $m_a = n_b$  come:

$$AB_{ij} = \sum_{r=1}^n a_{ir} b_{rj}$$

Dato un vettore  $\vec{v}$  possiamo calcolare il prodotto di vettore per matrice considerando il vettore una matrice colonna e applicando lo stessa definizione del prodotto fra matrici (quindi la lunghezza di  $\vec{x}$  dovrà essere pari al numero di colonne della matrice).

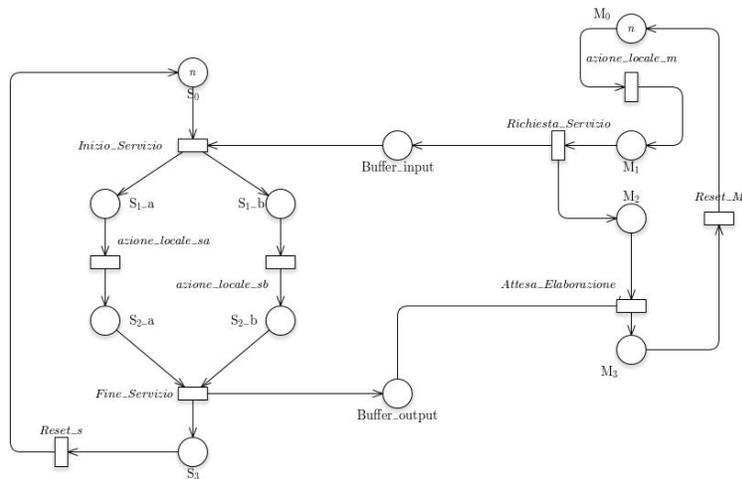
# Esercizio P/N

Francesco Mecca

May 22, 2020

## 1 Rete A

M master identici e S slave identici di tipo 1.



(n)

Figure 1: Modello della reteA

La figura rappresenta la rete di Petri P/T dell'esercizio A. Il master è modellato dai posti  $M_0$ ,  $M_1$ ,  $M_2$ ,  $M_3$  e dalle transizioni  $Azione\_Locale$ ,  $Richiesta\_Servizio$ ,  $Attesa\_Elaborazione$  e  $Reset\_M$ . Lo slave è modellato dai posti  $S_0$ ,  $S_1.a$ ,  $S_1.b$ ,  $S_2.a$ ,  $S_2.b$  e  $S_3$  e dalle transizioni  $Inizio\_Servizio$ ,

*Azione\_Locale\_S\_a*, *Azione\_Locale\_S\_b*, *Fine\_Servizio* e *Reset\_S*. La richiesta del servizio verso lo slave è gestita attraverso due buffer, posti *Buffer\_Input* e posto *Buffer\_Output*.

## 1.1 Risultati

Nella tabella vengono mostrate il numero di archi e di nodi al variare dei parametri M e S. Le cifre sono indicative dell'aumentare della dimensione dello spazio degli stati proporzionalmente al numero di marcature.

master, slaves	Nodi	Archi
1, 1	14	19
2, 2	94	222
3, 3	426	334
4, 4	1500	5610
5, 5	4422	18720
6, 6	11418	52998
7, 7	26598	132594
8, 8	57057	301158
9, 9	114400	632775
10, 10	216788	1246960
11, 11	391612	2328612
12, 12	678912	4153916
13, 13	1135668	7123272
14, 14	1841100	11802420
15, 15	2903124	18973020

## 1.2 Considerazioni su Fork/Join

Il modello non garantisce che avvenga il join di due processi dello stesso padre quando la marcatura degli slave è maggiore di 2. Si può garantire che avvenga il join di due processi forkati dallo stesso padre attraverso differenti strutture slaves o usando reti WN.

### 1.3 Riduzione

Una rete di petri può essere ridotta usando le seguenti tecniche:

- fusione
- eliminazione
- rimozione dei loop

Nelle figure vengono mostrate alcune fasi di riduzione della rete in analisi.

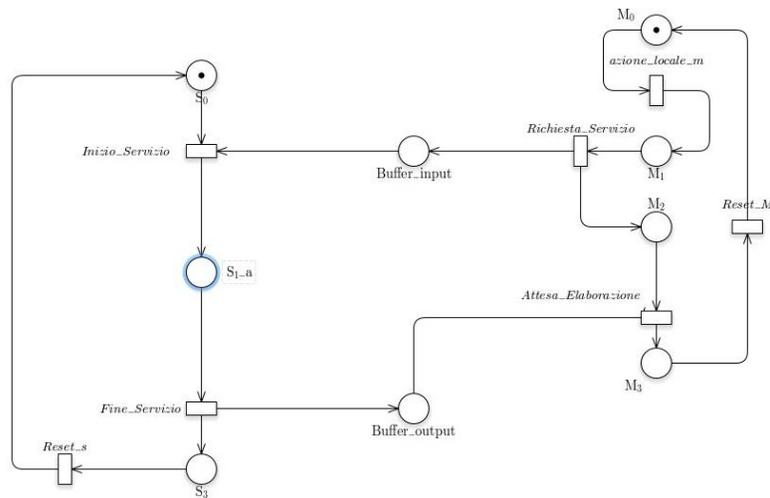


Figure 2: eliminazione di posti identici

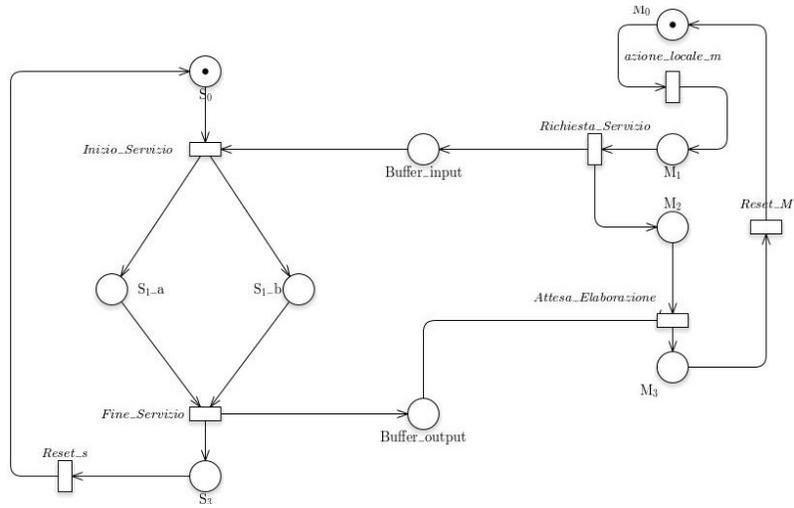


Figure 3: fusione di posti

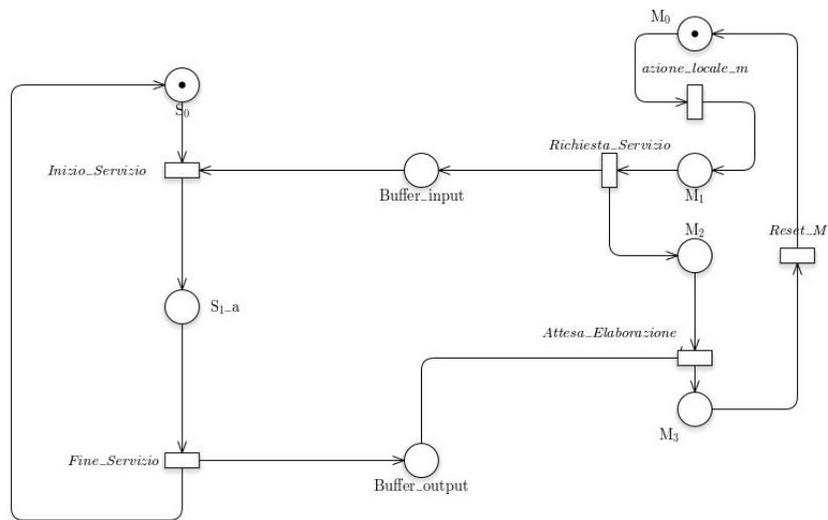


Figure 4: fusione di transizioni

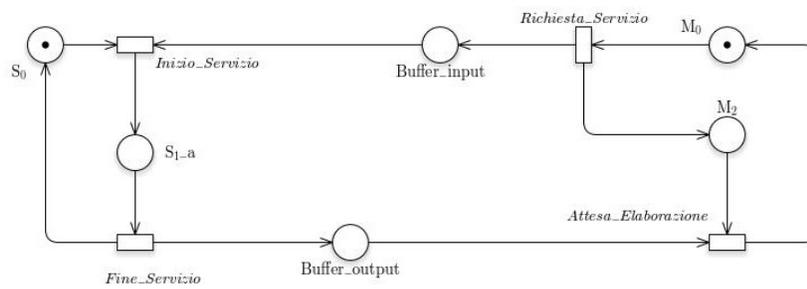


Figure 5: fusione di transizioni e posti

## 1.4 P e T invarianti

Tramite GreatSPN possiamo calcolare gli T- e P- semiflussi

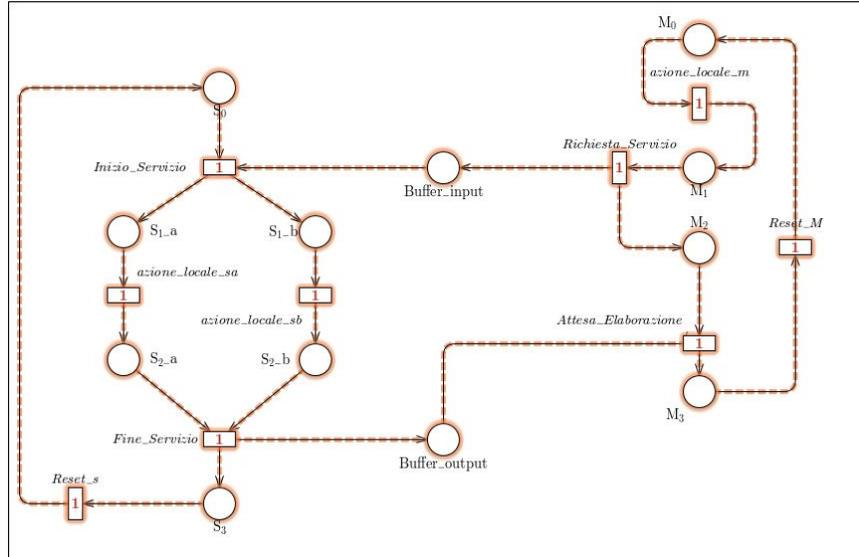


Figure 6: T-semiflows

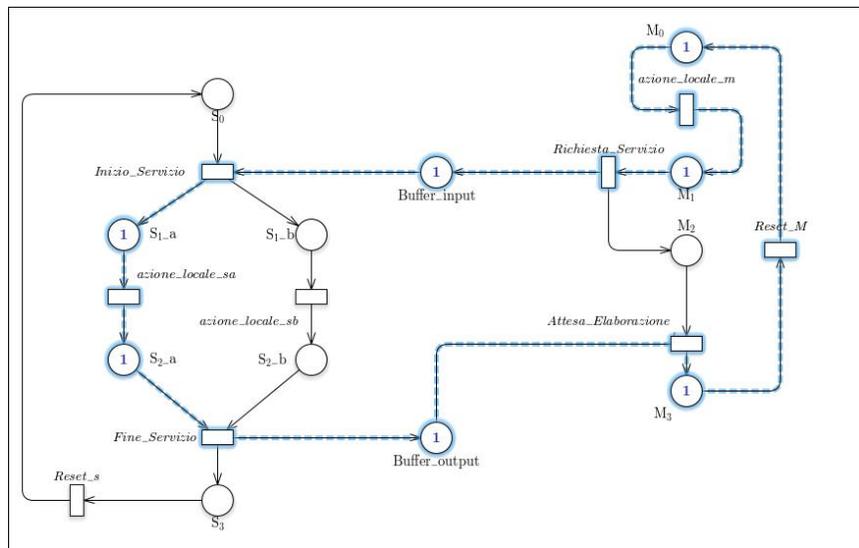


Figure 7: P-semiflows

Gli P-semiflussi sono i seguenti:

- $S0 + S1_a + S2_a + S3$
- $S0 + S1_b + S2_b + S3$
- $M0 + M1 + M2 + M3$
- $S1_a + S2_a + \text{Buffer\_output} + \text{Buffer\_input} + M0 + M1 + M3$
- $S1_b + S2_b + \text{Buffer\_output} + \text{Buffer\_input} + M0 + M1 + M3$

Il T-semiflusso è il seguente:

- $\text{Inizio\_servizio} + \text{azione\_locale\_sa} + \text{azione\_locale\_sb} +$   
 $\text{Fine\_servizio} + \text{azione\_locale\_m} + \text{Richiesta\_servizio} +$   
 $\text{Attesa\_elaborazione} + \text{Reset\_M} + \text{Reset\_S}$

e dato che comprende tutte le transizioni, il sistema rispetta la proprietà di liveness. Dato che la reteA è interamente coperta dagli P-semiflussi, possiamo affermare che la rete sia bounded. Gli P-semiflussi ci permettono di ricavare i seguenti invarianti lineari relativi ai marking  $m$ :

- $m[S0] + m[S1_a] + m[S2_a] + m[S3] = 1$
- $m[S0] + m[S1_b] + m[S2_b] + m[S3] = 1$
- $m[M0] + m[M1] + m[M2] + m[M3] = 1$
- $m[S1_a] + m[S2_a] + m[\text{Buffer\_output}] + m[\text{Buffer\_input}] + m[M0] + m[M1] + m[M3] = 1$
- $m[S1_b] + m[S2_b] + m[\text{Buffer\_output}] + m[\text{Buffer\_input}] + m[M0] + m[M1] + m[M3] = 1$

Dato che  $\forall p \in P, m[p] \geq 0$  possiamo affermare, a partire dalle precedenti uguaglianze che:

- ogni posto nei seguenti insieme è in mutua esclusione con gli elementi dello stesso insieme:

$\{S0, S1_a, S2_a, S3\}$

$\{S0, S1_b, S2_b, S3\}$

$\{M0, M1, M2, M3\}$

$\{S1_a, S2_a, \text{Buffer\_output}, \text{Buffer\_input}, M0, M1, M3\}$

$\{S1_b, S2_b, \text{Buffer\_output}, \text{Buffer\_input}, M0, M1, M3\}$

- $\forall p_i \in P, m[p_i] \leq 1$  (bounds)
- dato che i posti che sono gli unici *enablers* di una transizione sono i seguenti:

$$S1_a, S1_b, S3, M0, M1, M3$$

e quindi possiamo provare a dimostrare l'assenza di deadlock partendo dagli invarianti lineari relativi ai marking:

- $m[S0] + m[S2_a] = 1$
- $m[S0] + m[S2_b] = 1$
- $m[M2] = 1$
- $m[S2_a] + m[\text{Buffer\_output}] + m[\text{Buffer\_input}] = 1$
- $m[S2_b] + m[\text{Buffer\_output}] + m[\text{Buffer\_input}] = 1$

Dato che M2 è marcata, per far sì che *attesa\_elaborazione* non venga abilitata:

$$m[\text{Buffer\_output}] = 0$$

Inoltre per far sì che Inizio\_Servizio e Fine\_Servizio non vengano abilitate:

- $m[\text{Buffer\_input}] + M[S0] \leq 1$
- $m[S2_a] + m[S2_b] \leq 1$

Riassumendo, il sistema è il seguente:

- $m[S0] + m[S2_a] = 1$
- $m[S0] + m[S2_b] = 1$
- $m[S2_a] + m[\text{Buffer\_input}] = 1$
- $m[S2_b] + m[\text{Buffer\_input}] = 1$
- $m[\text{Buffer\_input}] + M[S0] \leq 1$
- $m[S2_a] + m[S2_b] \leq 1$

che per la legge di conservazione dei token, non può essere soddisfatto. Quindi nel sistema non vi è la possibilità di deadlock.

## 2 Rete B

M master identici, uno slave di tipo 1 e uno slave di tipo 1 scelti liberamente dai master.

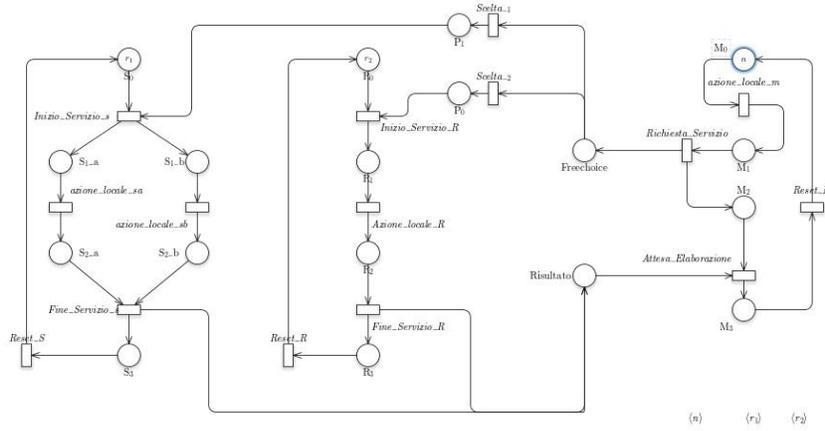


Figure 8: Modello della reteB

La figura rappresenta la rete di Petri P/T dell'esercizio B. Il master è modellato dai posti  $M_0$ ,  $M_1$ ,  $M_2$ ,  $M_3$  e dalle transizioni *Azione\_Locale*, *Richiesta\_Servizio*, *Attesa\_Elaborazione* e *Reset\_M*. Lo slave di tipo 1 è modellato dai posti  $S_0$ ,  $S_1_a$ ,  $S_1_b$ ,  $S_2_a$ ,  $S_2_b$  e  $S_3$  e dalle transizioni *Inizio\_Servizio*, *Azione\_Locale\_S\_a*, *Azione\_Locale\_S\_b*, *Fine\_Servizio* e *Reset\_S*. Lo slave di tipo 2 è modellato dai posti  $R_0$ ,  $R_1_a$ ,  $R_1_b$ ,  $R_2_a$ ,  $R_2_b$  e  $R_3$  e dalle transizioni *Inizio\_Servizio\_R*, *Azione\_Locale\_R*, *Fine\_Servizio* e *Reset\_R*. La richiesta del servizio verso lo slave scelto è gestita attraverso due buffer, *FreeChoice* e *Risultato*.

## 2.1 Risultati

Al variare del marking del master:

master, slaves	Stati	Archi
1, 2	40	76
2, 2	204	544
3, 2	728	2400
4, 2	2072	7896
5, 2	5040	21336
6, 2	10920	50064
7, 2	21648	105648
8, 2	39996	205260
9, 2	69784	373252
10, 2	116116	642928

Parametrizzando anche il marking sugli slaves (R+S):

master, slaves	Stati	Archi
1, 2	40	76
2, 2	204	544
4, 4	7265	32674
6, 6	113464	664234
8, 8	1073226	7405654
10, 10	7212128	55762000

## 2.2 Considerazioni su Fork/Join

Lo slave di tipo 1 processa una sola richiesta alla volta. Il master in attesa del risultato (M2) potrebbe ricevere il risultato di un lavoro richiesto da un altro master.

## 2.3 P e T invarianti

Tramite GreatSPN possiamo calcolare gli T- e P- semiflussi

Gli P-invarianti sono i seguenti:

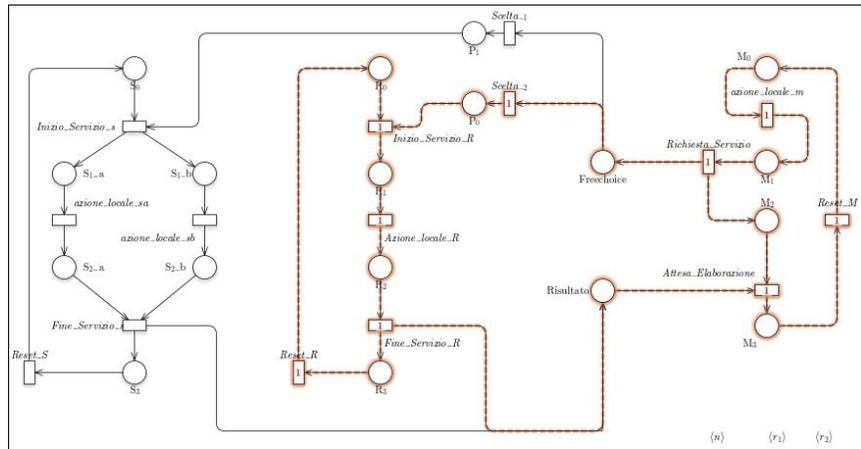


Figure 9: T-semiflows

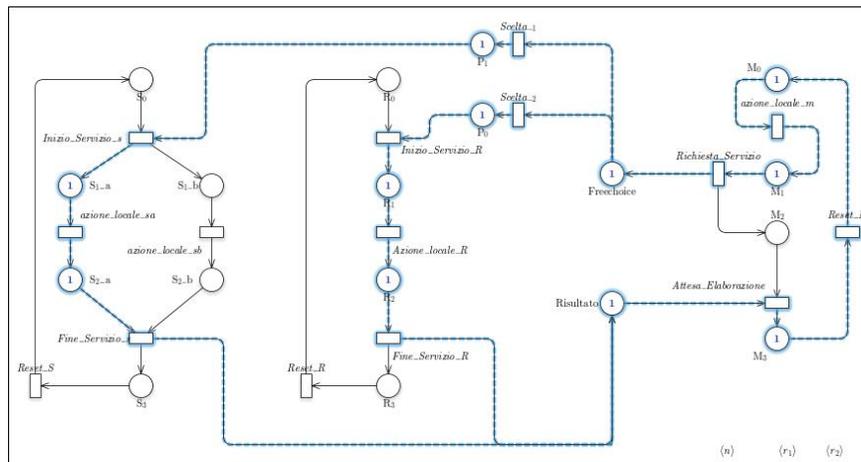


Figure 10: P-semiflows

- $S0 + S1\_a + S2\_a + S3$
- $S0 + S1\_b + S2\_b + S3$
- $R0 + R1 + R2 + R3$
- $M0 + M1 + M2 + M3$
- $S1\_a + S2\_a + R1 + R2 + M0 + M1 + M3 + \text{Freechoice} + P0 + P1 + \text{Risultato}$
- $S1\_b + S2\_b + R1 + R2 + M0 + M1 + M3 + \text{Freechoice} + P0 + P1 + \text{Risultato}$

Gli T-invarianti sono i seguenti:

- $\text{Inizio\_servizio\_R} + \text{azione\_locale\_R} +$   
 $\text{Fine\_servizio\_R} + \text{Reset\_R} + \text{azione\_locale\_m} + \text{Richiesta\_servizio} +$   
 $\text{Attesa\_elaborazione} + \text{Reset\_M} + \text{Scelta\_2}$
- $\text{Inizio\_servizio\_S} + \text{azione\_locale\_sa} + \text{azione\_locale\_sb} +$   
 $\text{Fine\_servizio\_S} + \text{Reset\_s} + \text{azione\_locale\_m} + \text{Richiesta\_servizio} +$   
 $\text{Attesa\_elaborazione} + \text{Reset\_m} + \text{Scelta\_1}$

Dato che ci sono due semiflussi, ognuno relativo alle transizioni dei due diversi slaves, c'è possibilità di starvation. Possiamo infatti immaginare una traccia di esecuzione in cui il master in seguito a FreeChoice sceglie sempre il primo slave. Questo non succederebbe in un sistema fair, ovvero se si obbliga un'automata che entra in uno stato infinite volte ad eseguire tutte le possibili transizioni da quello stato. In tal caso non avremmo starvation e la proprietà di liveness sarebbe rispettata.

Dato che la reteB è interamente coperta dagli P-semiflussi, possiamo affermare che la rete sia bounded. Dimostriamo invece che la rete non ha possibilità di deadlock.

- $m[S0] + m[S1\_a] + m[S2\_a] + m[S3] = 1$
- $m[S0] + m[S1_b] + m[S2_b] + m[S3] = 1$
- $m[R0] + m[R1] + m[R2] + m[R3] = 1$
- $m[M0] + m[M1] + m[M2] + m[M3] = 1$
- $m[S1\_a] + m[S2\_a] + m[R1] + m[R2] + m[M0] + m[M1] + m[M3] + m[Freechoice] + m[P0] + m[P1] + m[Risultato] = 1$
- $m[S1_b] + m[S2_b] + m[R1] + m[R2] + m[M0] + m[M1] + m[M3] + m[Freechoice] + m[P0] + m[P1] + m[Risultato] = 1$

I posti che sono gli unici enablers di una sola transizione sono:

$M0, M1, M3, R1, R2, R3, FreeChoice, S1_a, S1_b, S3$

Gli invarianti lineari dei marking diventano:

- $m[S0] + m[S2\_a] = 1$
- $m[S0] + m[S2_b] = 1$
- $m[R0] = 1$
- $m[M2] = 1$
- $m[S2\_a] + m[P0] + m[P1] + m[Risultato] = 1$
- $m[S2_b] + m[P0] + m[P1] + m[Risultato] = 1$

Dati i marking in R0 e M2, per far sì che */Inizio\_Servizio/\_R, Attesa\_Elaborazione, /Fine\_Servizio/s* e */Inizio\_Servizio/s* non vengano abilitati:

- $m[P0] = 0$
- $m[Risultato] = 0$
- $m[S2_a] + m[S2_b] \leq 1$
- $m[P1] + m[S0] \leq 1$

Il sistema si riduce a:

- $m[S0] + m[S2\_a] = 1$
- $m[S0] + m[S2_b] = 1$
- $m[S2\_a] + m[P1] = 1$
- $m[S2_b] + m[P1] = 1$
- $m[S2_a] + m[S2_b] \leq 1$
- $m[P1] + m[S0] \leq 1$

che non può essere soddisfatto per la legge di conservazione dei token.

### 3 Rete C

Due master identici, uno slave di tipo 1 e uno slave di tipo 1 scelti liberamente dai master.

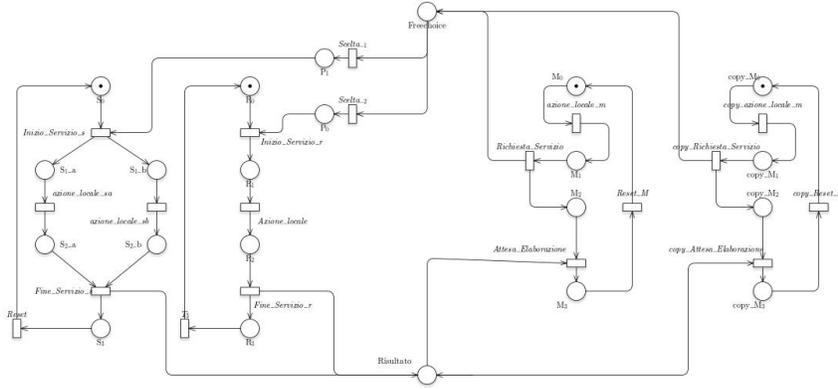


Figure 11: Modello della reteC

La figura rappresenta la rete di Petri P/T dell'esercizio C. Il master è modellato dai posti  $M_0, M_1, M_2, M_3$  e dalle transizioni *Azione\_Locale*, *Richiesta\_Servizio*, *Attesa\_Elaborazione* e *Reset\_M*. Lo slave di tipo 1 è modellato dai posti  $S_0, S_{1_a}, S_{1_b}, S_{2_a}, S_{2_b}$  e  $S_3$  e dalle transizioni *Inizio\_Servizio*, *Azione\_Locale\_S\_a*, *Azione\_Locale\_S\_b*, *Fine\_Servizio* e *Reset\_S* (il secondo master è una copia del primo). Lo slave di tipo 2 è modellato dai posti  $R_0, R_{1_a}, R_{1_b}, R_{2_a}, R_{2_b}$  e  $R_3$  e dalle transizioni *Inizio\_Servizio/\_R*, *Azione\_Locale\_R*, */Fine\_Servizio* e *Reset\_R*. La richiesta del servizio verso lo slave scelto è gestita attraverso due buffer, posti *FreeChoice* e *Risultato*.

#### 3.1 P e T invarianti

Tramite GreatSPN possiamo calcolare gli T- e P- semiflussi

Gli P-invarianti sono i seguenti:

- $S_0 + S_{1_a} + S_{2_a} + S_3$



- $S0 + S1_b + S2_b + S3$
- $R0 + R1 + R2 + R3$
- $M0 + M1 + M2 + M3$
- $\text{copy\_M0} + \text{copy\_M1} + \text{copy\_M2} + \text{copy\_M3}$
- $S1_a + S2_a + R1 + R2 + M0 + M1 + M3 + \text{Freechoice} + P0 + P1 + \text{Risultato} + \text{copy\_M0} + \text{copy\_M1} + \text{copy\_M3}$
- $S1_b + S2_b + R1 + R2 + M0 + M1 + M3 + \text{Freechoice} + P0 + P1 + \text{Risultato} + \text{copy\_M0} + \text{copy\_M1} + \text{copy\_M3}$

Dato che la reteC è interamente coperta dagli P-semiflussi, possiamo affermare che la rete sia bounded. Gli P-semiflussi ci permettono di ricavare i seguenti invarianti lineari relativi ai marking  $m$ :

- $m[S0] + m[S1_a] + m[S2_a] + m[S3] = 1$
- $m[S0] + m[S1_b] + m[S2_b] + m[S3] = 1$
- $m[R0] + m[R1] + m[R2] + m[R3] = 1$
- $m[M0] + m[M1] + m[M2] + m[M3] = 1$
- $m[\text{copy\_M0}] + m[\text{copy\_M1}] + m[\text{copy\_M2}] + m[\text{copy\_M3}] = 1$
- $m[S1_a] + m[S2_a] + m[R1] + m[R2] + m[M0] + m[M1] + m[M3] + m[\text{Freechoice}] + m[P0] + m[P1] + m[\text{Risultato}] + m[\text{copy\_M0}] + m[\text{copy\_M1}] + m[\text{copy\_M3}] = 1$
- $m[S1_b] + m[S2_b] + m[R1] + m[R2] + m[M0] + m[M1] + m[M3] + m[\text{Freechoice}] + m[P0] + m[P1] + m[\text{Risultato}] + m[\text{copy\_M0}] + m[\text{copy\_M1}] + m[\text{copy\_M3}] = 1$

Gli spazi *enablers* di una sola transizione sono i seguenti:

R1, R2, R3, S1<sub>a</sub>, S1<sub>b</sub>, S3, Risultato, M0, M1, M3, copy\_M0, copy\_M1, copy\_M3, FreeChoice

il sistema precedente diventa:

- $m[S0] + m[S2_a] = 1$
- $m[S0] + m[S2_b] = 1$
- $m[R0] = 1$
- $m[M2] = 1$
- $m[\text{copy\_M2}] = 1$
- $m[S2_b] + m[P0] + m[P1] = 1$
- $m[S2_a] + m[P0] + m[P1] = 1$

Dati i marking in R0 e M2 e copy\_M2, per far sì che */Inizio\_Servizio/\_R*, *Attesa\_Elaborazione*, *copy\_/Attesa\_Elaborazione/*, */Fine\_Servizio/s* e */Inizio\_Servizio/s* non vengano abilitati:

- $m[P0] = 0$
- $m[\text{Risultato}] = 0$
- $m[S2_a] + m[S2_b] \leq 1$
- $m[P1] + m[S0] \leq 1$

Il sistema si riduce allo stesso della precedente rete B:

- $m[S0] + m[S2_a] = 1$
- $m[S0] + m[S2_b] = 1$
- $m[S2_b] + m[P1] = 1$
- $m[S2_a] + m[P1] = 1$
- $m[S2_a] + m[S2_b] \leq 1$
- $m[P1] + m[S0] \leq 1$

e non può essere soddisfatto per la legge di conservazione dei token.

Gli T-invarianti sono i seguenti:

- $\text{Inizio\_Servizio}_r + \text{Azione\_Locale} + \text{Fine\_Servizio}_r + T3 + \text{azione\_locale}_m$   
+  $\text{Richiesta\_Servizio} + \text{Attesa\_Elaborazione} + \text{Reset\_M} + \text{Scelta}_1$
- $\text{Inizio\_Servizio}_s + \text{Azione\_Locale}_{sa} + \text{Azione\_Locale}_{sb} + \text{Fine\_Servizio}_s$   
+  $T3 + \text{azione\_locale}_m + \text{Richiesta\_Servizio} + \text{Attesa\_Elaborazione}$   
+  $\text{Reset\_M} + \text{Scelta}_1$

- $\text{Inizio\_Servizio}_r + \text{Azione\_Locale} + \text{Fine\_Servizio}_r + T3 + \text{Scelta}_2 + \text{copy}_{\text{azione\_locale}_m} + \text{copy\_Richiesta\_Servizio} + \text{copy\_Attesa\_Elaborazione} + \text{copy\_Reset}_m$
- $\text{Inizio\_Servizio}_s + \text{Azione\_Locale}_{sa} + \text{Azione\_Locale}_{sb} + \text{Fine\_Servizio}_s + \text{Reset} + \text{Scelta}_1 + \text{copy\_azione\_locale}_m + \text{copy\_Richiesta\_Servizio} + \text{copy\_Attesa\_Elaborazione} + \text{copy\_Reset}_m$

Come nella rete B, in assenza di fairness non possiamo rispettare la condizione di liveness e c'è possibilità di starvation.

## 4 Rete D

Due master identici, uno slave di tipo 1 e uno slave di tipo 1 scelti associati ciascuno ad un master diverso.

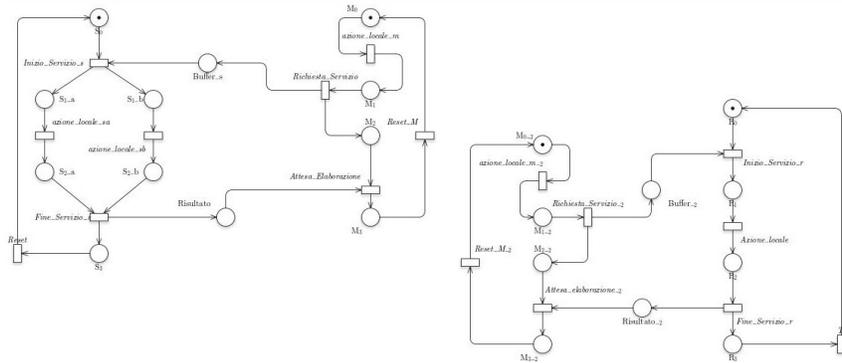


Figure 14: Modello della reteD

### 4.1 P e T invarianti

Tramite GreatSPN possiamo calcolare gli T- e P- semiflussi

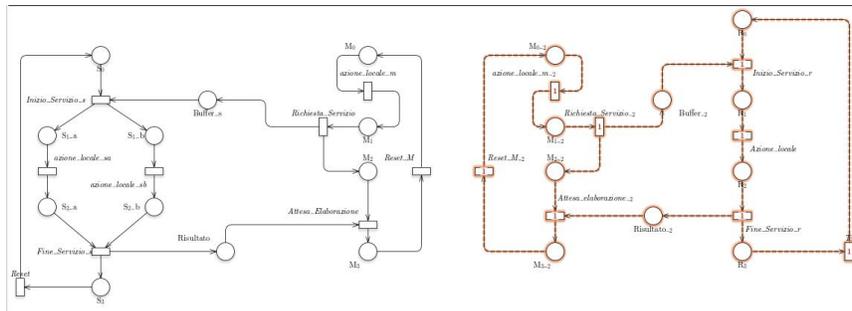


Figure 15: T-semiflows

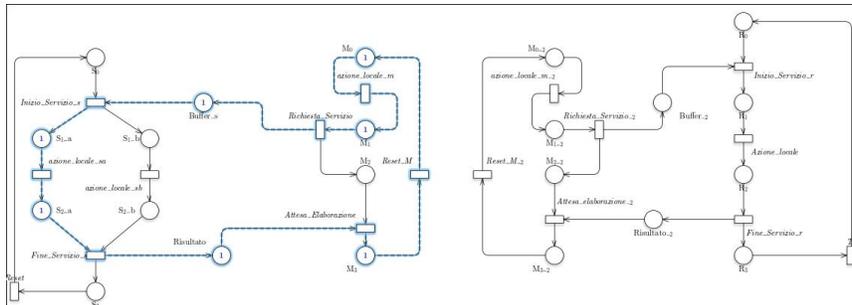


Figure 16: P-semiflows

Gli P-invarianti sono i seguenti:

- $S0 + S1_a + S2_a + S3$
- $S0 + S1_b + S2_b + S3$
- $R0 + R1 + R2 + R3$
- $M0 + M1 + M2 + M3$
- $S1_a + S2_a + M0 + M1 + M3 + Buffer_s + Risultato$
- $S1_b + S2_b + M0 + M1 + M3 + Buffer_s + Risultato$
- $M0_2 + M1_2 + M3_2$
- $R1 + R2 + M0_2 + M1_2 + M3_2 + Buffer_2 + Risultato_2$

Ai fini della dimostrazione dell'assenza di deadlock, possiamo notare che lo slave di tipo 2 è equivalente allo slave di tipo 1 se si applicano due riduzioni alla rete (vengono fusi in un unico posto  $S1_a-S2_a$  e  $S1_b-S2_b$ , poi eliminata la fork). Inoltre i master sono indipendenti fra di loro e ciascuno rispetta l'assenza di deadlock come già dimostrato nella rete A. Gli T-invarianti sono i seguenti:

- $Inizio\_Servizio_s + azione\_locale_{sa} + azione\_locale_{sb} + Fine\_Servizio_s + Reset + azione\_locale_m + Richiesta\_Servizio + Attesa\_Elaborazione + Reset_m$
- $Inizio\_Servizio_r + Azione\_locale + Fine\_Servizio_r + T3 azione\_locale_{m2} + Richiesta\_Servizio_2 + Attesa\_Elaborazione_2 + Reset_{m2}$

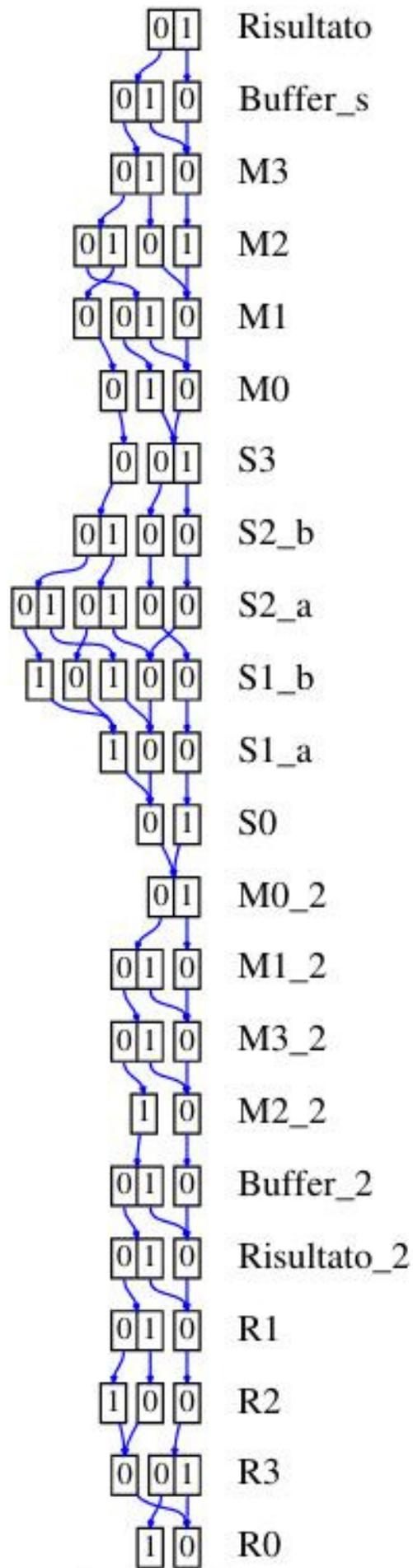
Come nella rete B, in assenza di fairness non possiamo rispettare la condizione di liveness e c'è possibilità di starvation.

## 4.2 Decision Diagram

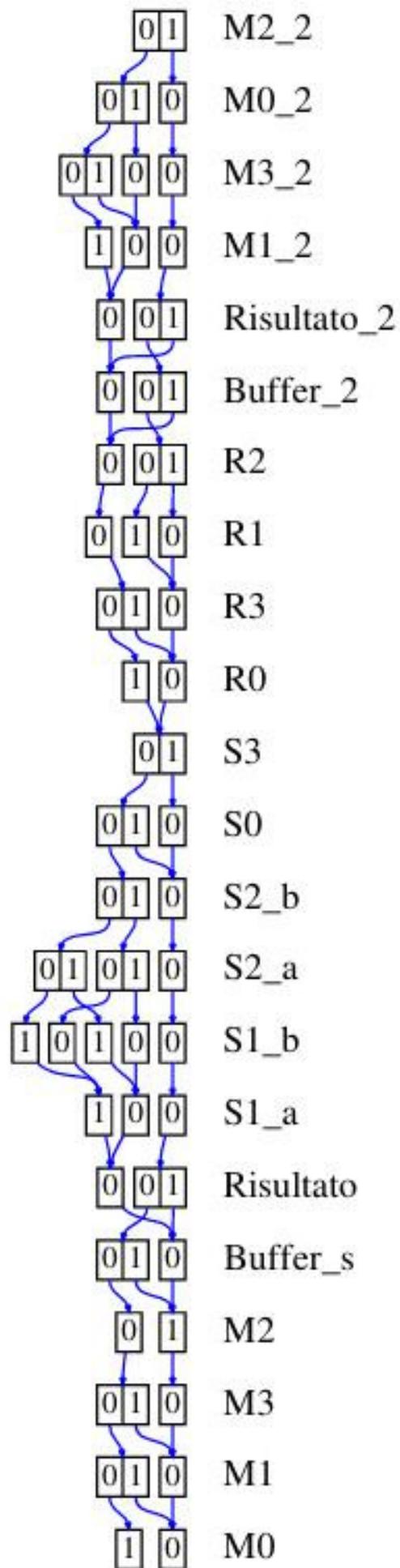
L'efficacia dei decision diagram sulla generazione dello stato degli spazi dipende fortemente dall'ordine delle variabili. Di seguito vengono mostrati i decision diagram usando per le assegnazioni i seguenti algoritmi:

- Sloan: un algoritmo di riduzione della banda di matrici sparse con una buona performance

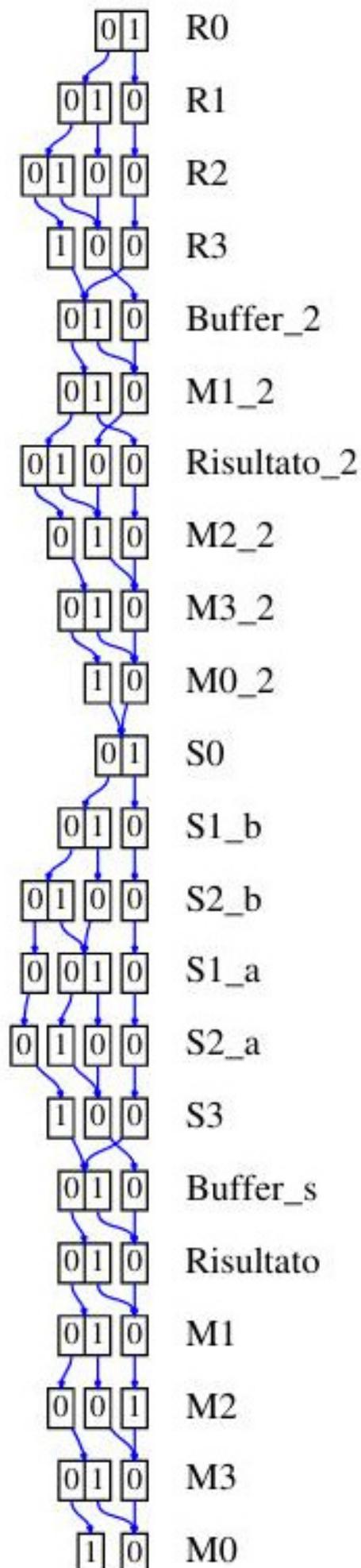
- Cuthill-McKee: un altro algoritmo di riduzione della banda di matrici sparse
- Tovchigrechko e Noack: due algoritmo appositamente ideati per le reti di Petri, anch'essi con una buona performance
- P-chaining: un algoritmo che sfrutta le informazioni strutturali della rete ma ha una bassa performance
- Gradient-P
- Gibbs-Poole-Stockmeier: un altro algoritmo matriciale che nella rete in analisi ha restituito il risultato peggiore



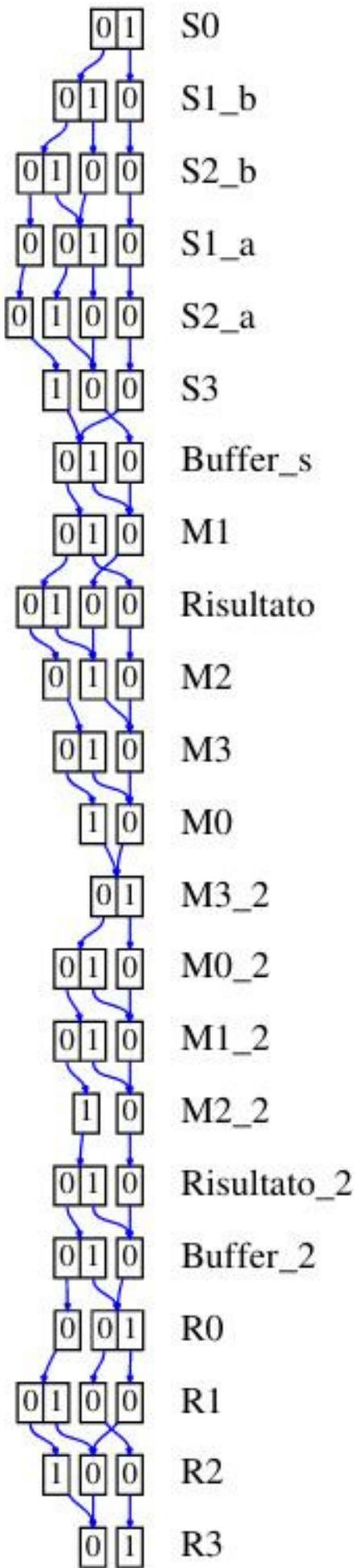
Algoritmo di Sloan



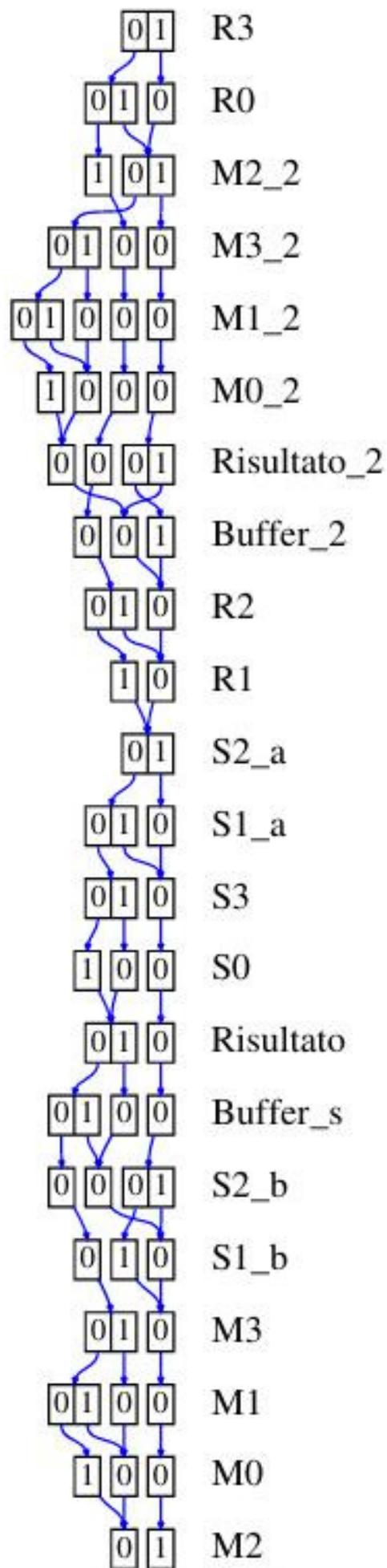
Algoritmo di Cuthull-McKee



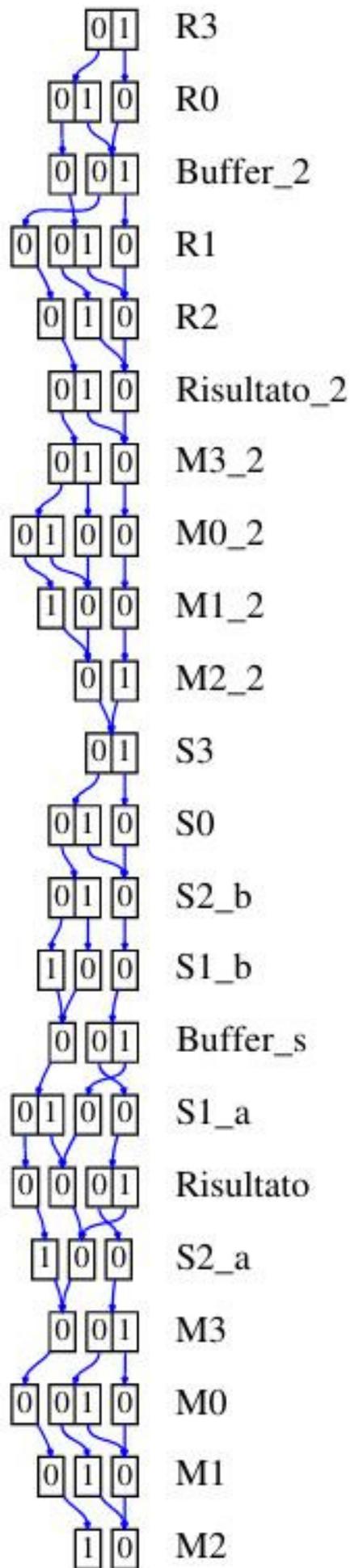
Algoritmo di Tovchigrechko



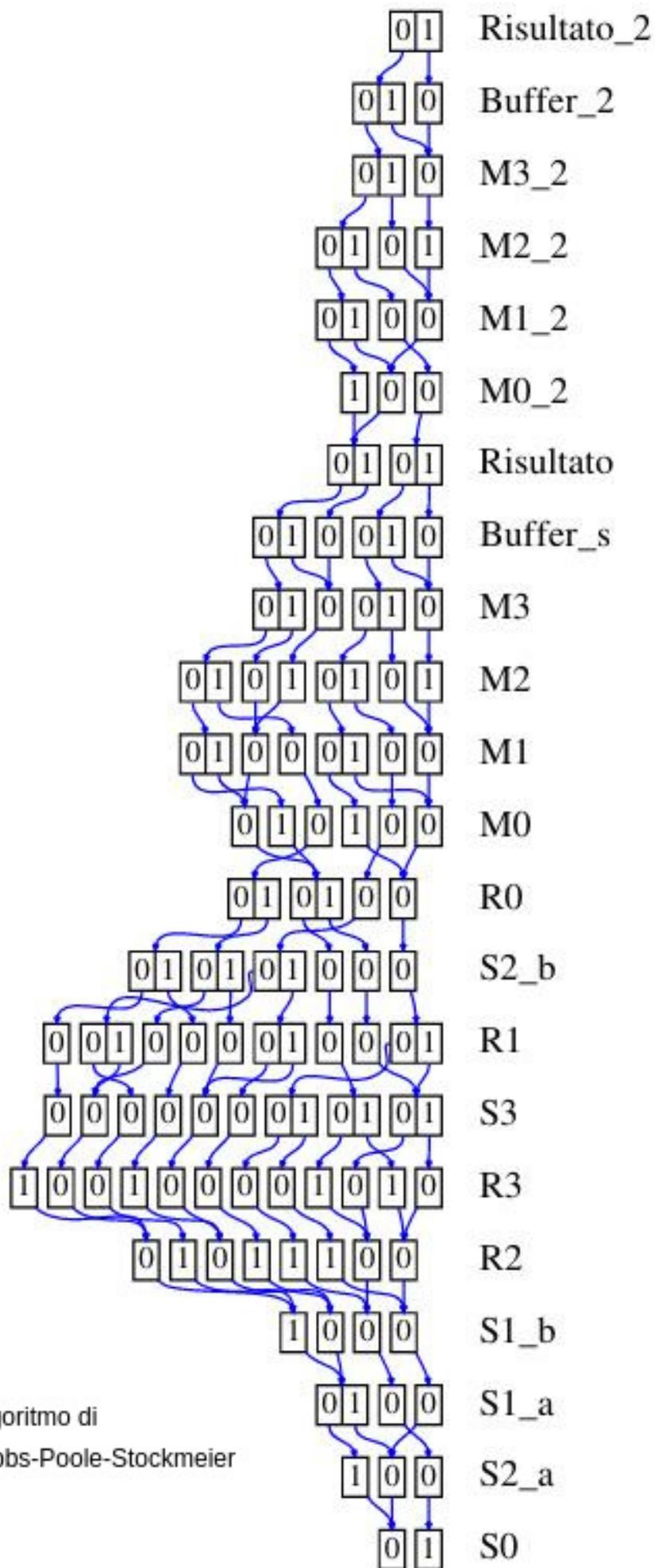
Algoritmo di Noack



Algoritmo P-Chain



Algoritmo Gradient-P



Algoritmo di  
Gibbs-Poole-Stockmeier

# Esercizio CPN

Francesco Mecca

May 22, 2020

## 1 Rete E

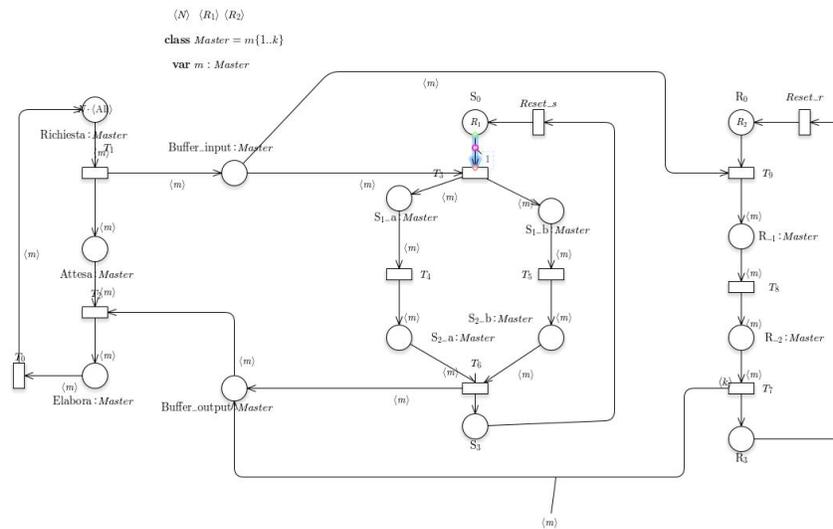


Figure 1: Rete E

I master sono modellati dai posti Richiesta, Attesa e Elabora. Lo slave di tipo 1 è modellato dai posti  $S_0$ ,  $S_{1\_a}$ ,  $S_{2\_a}$ ,  $S_{1\_a}$ ,  $S_{1\_b}$  e  $S_3$ . Lo slave di tipo 2 è modellato dai posti  $R_0$ ,  $R_1$ ,  $R_2$  e  $R_3$ . La richiesta agli slave è modellata attraverso due posti:  $Buffer\_Input$ ,  $Buffer\_output$ .

La classe di colori utilizzata è Master, che permette di distinguere i tre master  $m1$ ,  $m2$ ,  $m3$ , mentre il color domain è  $m$  e viene utilizzato in tutte le transizioni dove c'è necessità di distinguere l'origine del marking.

### 1.1 Reachability Graph

Le seguenti tabelle elencano la dimensione dello spazio degli stati al variare del numero di master (N) e di slave (Sn e Rn).

N	Sn, Rn	Marking RG	Marking SRG
1	1	1024	232
1	2	4080	888
1	3	9280	2032
1	4	16480	3616
2	1	28480	5240
2	2	201708	35874
2	3	678240	119488
3	1	310400	54080
3	2	3151680	538380

L'utilizzo del Symbolic Reachability Graph permette di ridurre il numero di marking di un fattore  $>4$ .

### 1.2 Reachability Graph al variare delle classi di colore

k	Marking RG	Marking SRG
3	1024	232
4	5632	460
5	29696	804
6	151552	1288
7	753664	1936
8	3670016	2772
9	17563648	3820
10	82837504	5104

La tabella mostra l'aumentare dei marking del Reachability Graph all'aumentare delle classi di colore. Benché la crescita dello spazio degli stati sia lineare in entrambi i casi, notiamo come lo spazio del SRG sia notevolmente più contenuto dello spazio del RG. Inoltre, il rapporto marking RG/SRG (dello stato degli spazi) aumenta proporzionalmente al valore di  $k$ .

A conferma di quanto espresso, notiamo che anche con valori di marcatura iniziale maggiore per il master e gli slave, notiamo lo stesso effetto sul rapporto di marking RG/SRG (limitatamente alle possibilità di calcolo).

$N, S_n, R_n$	$k$	Marking RG	Marking SRG
2, 2, 2	3	201708	35874
2, 2, 2	4	4190624	203136
2, 2, 2	5	78523200	875478

### 1.3 Considerazioni su Fork/Join

Non può avvenire un join fra due processi con diverso padre quando:

- $N = 1$ : sullo stesso spazio non può essere presente più di un token con lo stesso colore
- $S_n = 1$ : in questo secondo caso, come nei precedenti esercizi, avviene un solo fork alla volta.

Di seguito viene mostrata un'esecuzione in cui è possibile che avvenga il join di due processi con lo diverso padre:

- con  $N > 1$  consideriamo che nella rete vi siano al momento iniziale due token dello stesso colore "m0"
- entrambi i token vengono inseriti nel Buffer\_input e in seguito inviati allo slave di tipo 1.
- quando  $S_n > 1$  uno dei due token può eseguire lo scatto attraverso la transizione T3 prima che l'altro token arrivi alla transizione T6 (in cui avviene il join).

## 2 Rete F

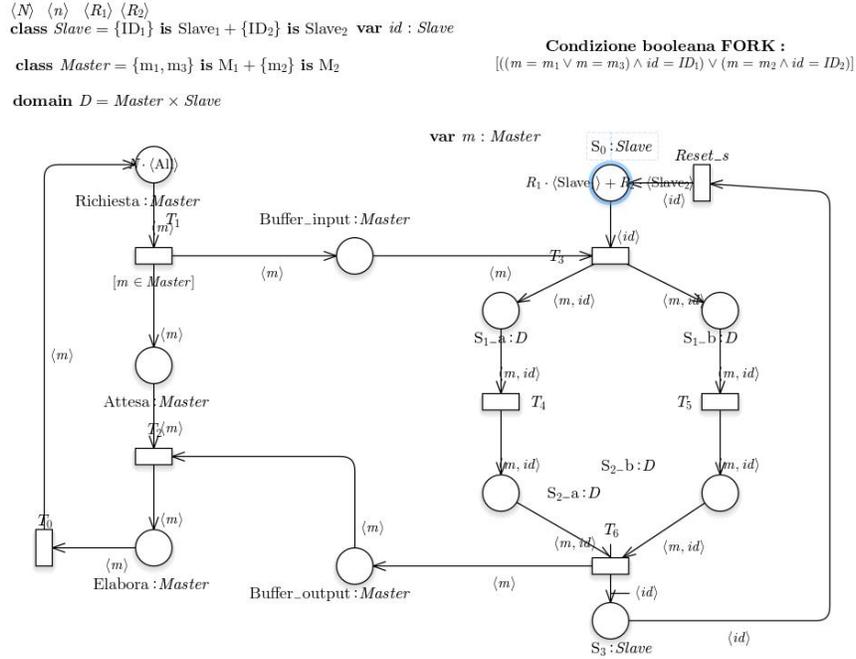


Figure 2: Rete F

I master sono modellati dai posti Richiesta, Attesa e Elabora. Gli slave di tipo 1 sono modellato dai posti S0, S1\_a, S2\_a, S1\_a, S1\_b e S3. La richiesta agli slave è modellata attraverso due posti: Buffer\_Input, Buffer\_output.

Nella rete sono presenti due classi di colore:

- Master, che permette di distinguere i diversi master  $m1, m2, m3$
- Slave, che permette di distinguere i due slaves  $ID1$  e  $ID2$

La classe Slave è divisa in due sottoclassi, Slave1 e Slave2, per facilitare in seguito la parametrizzazione degli slaves. Anche la classe Master è divisa in due sottoclassi utilizzate nell'espressione booleane della transizione Fork. Il dominio  $D = Master \times Slave$  è utilizzato per la multiplicity degli archi fra fork e join, in modo da tener traccia delle richieste effettuate.

## 2.1 Reachability Graph

Le seguenti tabelle elencano la dimensione dello spazio degli stati al variare del numero di master (N) e di slave ( $S_n$  e  $R_n$ ).

N	R1, R2	Marking RG	Marking SRG
1	1	768	432
1	2	2560	1440
1	3	5376	3024
1	4	5184	9216
2	1	18720	9720
2	2	97696	50481
2	3	267120	137376
3	1	192000	97600
3	2	1309440	663520

Rispetto alla precedente rete, il rapporto stato degli spazi di RG/SRG è minore.

## 2.2 Reachability Graph al variare delle classi di colore

La seguente tabella mostra l'incremento dello spazio degli stati all'aumentare degli elementi della classe di color Master. Come nella precedente rete, il rapporto marking RG/SRG aumenta proporzionalmente alla cardinalità della classe.

M1	M2	Marking RG	Marking SRG
2	1	768	432
3	1	3840	960
2	2	4096	1296
3	2	20480	2880
3	3	102400	6400
4	3	491520	12000
4	4	2359296	22500
5	4	11010048	37800
5	5	51380224	63504

Nella seguente tabella viene evidenziato l'incremento dello stato degli spazi all'aumentare degli elementi della classe di colore Slave.

Slave1	Slave2	Marking RG	Marking SRG
1	1	768	432
1	2	2048	720
2	1	2688	864
2	2	7168	1440
2	3	17920	2016
3	2	22528	2160
3	3	56320	3024
3	4	135168	3888
4	3	163840	4032
4	4	393216	5184
4	5	917504	6336
5	4	1081344	6480
5	5	2523136	7920

Notiamo che l'alta cardinalità della classe Slave abbia avuto un impatto minore sulla dimensione del RG rispetto alla classe master. Il rapporto RG/SRG segue quanto detto finora ed è il più grande fra tutte le reti analizzate.

### 2.3 Considerazioni su Fork/Join

Posso fare le stesse considerazioni che nella rete precedente (rete E), tenendo in considerazione che avendo due diverse tipologie di slave, un join fra due processi con padre diverso avverrà solo quando

$$(R_1 > 1 \vee R_2 > 1) \wedge N > 1$$

# Esercizio di analisi degli algoritmi di mutua esclusione

Francesco Mecca

May 22, 2020

## 1 Proprietà del modello

Ogni modello successivamente mostrato rispetta le seguenti proprietà: siano  $p$  e  $q$  due generici processi,

1. Mutua esclusione: garantisce che al più un solo processo è nella sezione critica ad ogni istante. È una proprietà di safety.

$$G (\neg c_p \vee \neg c_q)$$

2. Assenza di deadlock: ogni qualvolta un processo è in attesa di entrare nella sezione critica, eventualmente verrà concesso ad un processo di entrare nella sezione critica. È una proprietà di liveness.

$$G((w_p \vee w_q) \rightarrow F(c_p \vee c_q))$$

3. Assenza di starvation individuale: ogni qualvolta un processo è in attesa di entrare nella sezione critica, eventualmente gli verrà concesso.

$$G(w_p \rightarrow Fc_p)$$

Rispetto all'assenza di deadlock, è una proprietà di liveness ancora più forte della precedente.

$$\forall p, G(w_p \rightarrow Fc_p)$$

Possiamo convertire queste tre formule LTL in formule equivalenti CTL anteponendo l'operatore di stato A:

$$\begin{aligned} &AG (\neg c_p \vee \neg c_q) \\ &AG(w_p \rightarrow AF(c_p \vee c_q)) \\ &AG(w_p \rightarrow AFc_p) \end{aligned}$$

Benché non tutte le formule LTL possono essere convertite in una formula CTL equivalente anteponendo ad ogni operatore temporale l'operatore di stato A, per queste tre proprietà possiamo.

Si specifica che i processi non sono forzati a progredire al di fuori della regione critica (possono rimanere per un tempo indeterminato nella sezione *azione locale*  $l_p$ ) e quindi:

$$\begin{aligned} G((w_p \vee w_q) \rightarrow F(c_p \vee c_q)) = \text{false} &\rightarrow G((l_p \vee l_q) \rightarrow F(c_p \vee c_q)) \\ G((l_p \vee l_q) \rightarrow F(c_p \vee c_q)) = \text{true} &\rightarrow G((w_p \vee w_q) \rightarrow F(c_p \vee c_q)) \end{aligned}$$

## 2 Algoritmo 3.2

Due processi iterano all'infinito seguendo questo pseudocodice

```
while true:
    non-critical section
    await turn = ID
    critical section
    turn <- (ID%2)+1
```

### 2.1 NuSMV

Si è deciso di modellare l'algoritmo usando per ognuno dei due processi un'enumerazione di 4 stati ed una variabile turno di tipo intero.

```
state: {begin, wait, critical, done};
```

```

MODULE main
VAR
  turn: 1 .. 2;
  p: proc(turn, 1);
  q: proc(turn, 2);
ASSIGN
  init(turn) := 1;
  next(turn) :=
    case
      q.state = done: 1;
      p.state = done: 2;
      TRUE: turn;
    esac;

CTLSPEC -- no mutual exclusion
  AG (p.state != critical | q.state != critical)

CTLSPEC -- no deadlock
  AG ((p.state = wait | q.state = wait) -> AF (p.state = critical |
    q.state = critical))

CTLSPEC -- no individual starvation
  AG (p.state = wait -> AF p.state = critical)
CTLSPEC
  AG (q.state = wait -> AF q.state = critical)
CTLSPEC
  AG (q.state = wait -> EF q.state = critical)

LTLSPPEC -- no mutual exclusion
  G (p.state != critical | q.state != critical)
LTLSPPEC -- no deadlock
  G ((p.state = wait | q.state = wait) -> F (p.state = critical | q.
    state = critical))
LTLSPPEC -- no individual starvation
  G (p.state = wait -> F p.state = critical)

```

```

LTLSPEC
  G (q.state = wait -> F q.state = critical)

MODULE proc(turn, id) -- Model a process taking turn
VAR
  state: {begin, wait, critical, done};
ASSIGN
  init(state) := begin;
  next(state) :=
    case
      state = begin: {begin, wait};
      state = wait & turn = id: critical;
      state = critical: done;
      state = done: begin;
      TRUE: state;
    esac;

```

---

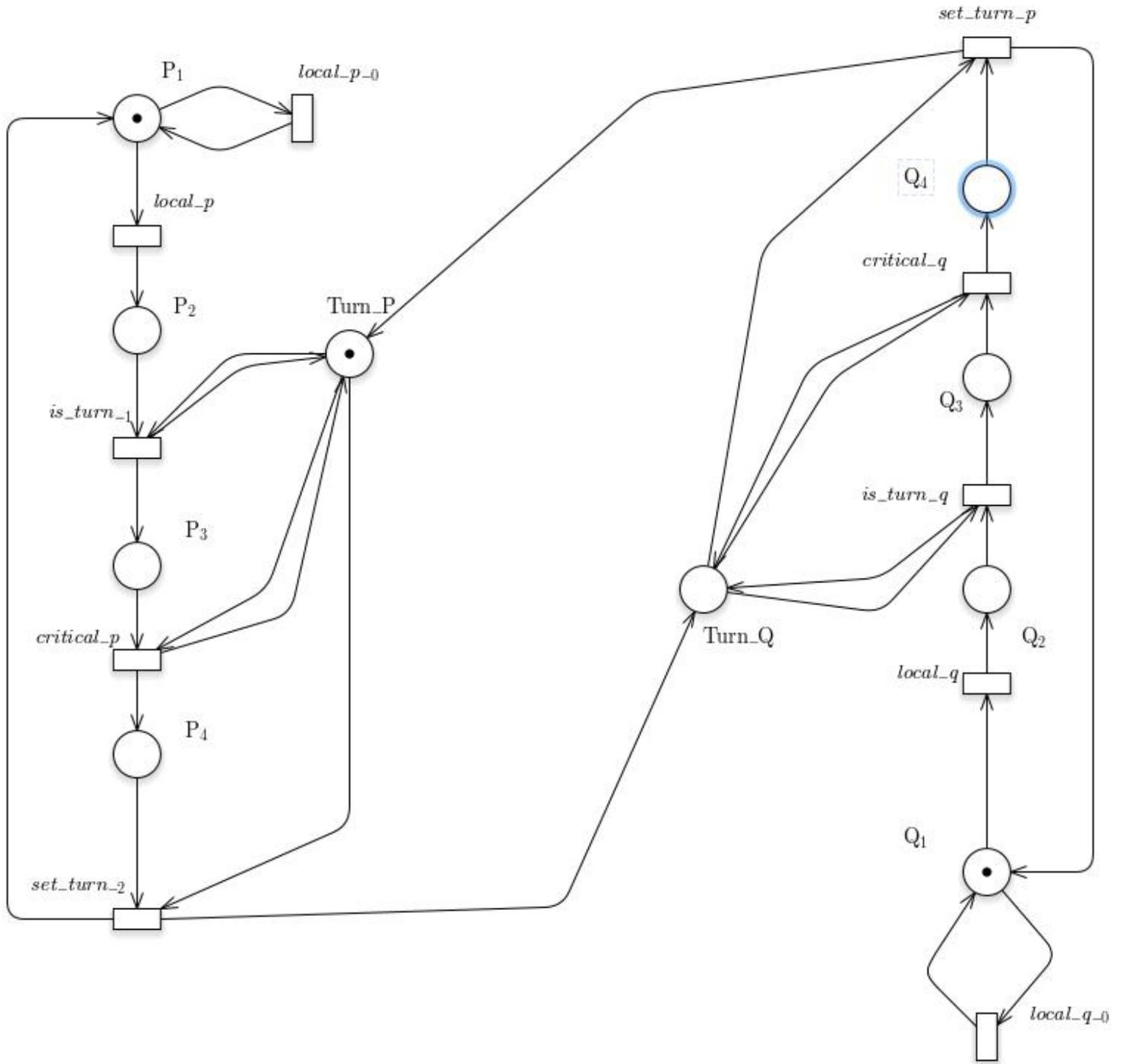
## 2.2 GreatSPN

Il codice utilizzato per le proprietà CTL è il seguente:

```

AG(!(#P3 == 1) || !(#P4 == 1))
AG ((#P1==1 || #Q1 == 1) -> AF (#P3 == 1 || #Q3 == 1))
AG (#P1==1 -> AF (#P3 == 1))
AG (#Q1==1 -> AF (#Q3 == 1))
AG (#Q1==1 -> EF (#Q3 == 1))

```



### 2.3 Algebra dei processi

Riportiamo il modello dell'algoritmo 3.2 secondo l'algebra dei processi CSP.

$$\begin{aligned}\text{System} &= \{(P_1 \parallel Q_1) \parallel T_p\} / \text{Sync} \\ \text{Act} &= \{\text{local}_p, \text{local}_q, \text{critical}_p, \text{critical}_q\} \\ \text{Sync} &= \{\text{is}_p, \text{is}_q, \text{set}_p, \text{set}_q\}\end{aligned}$$

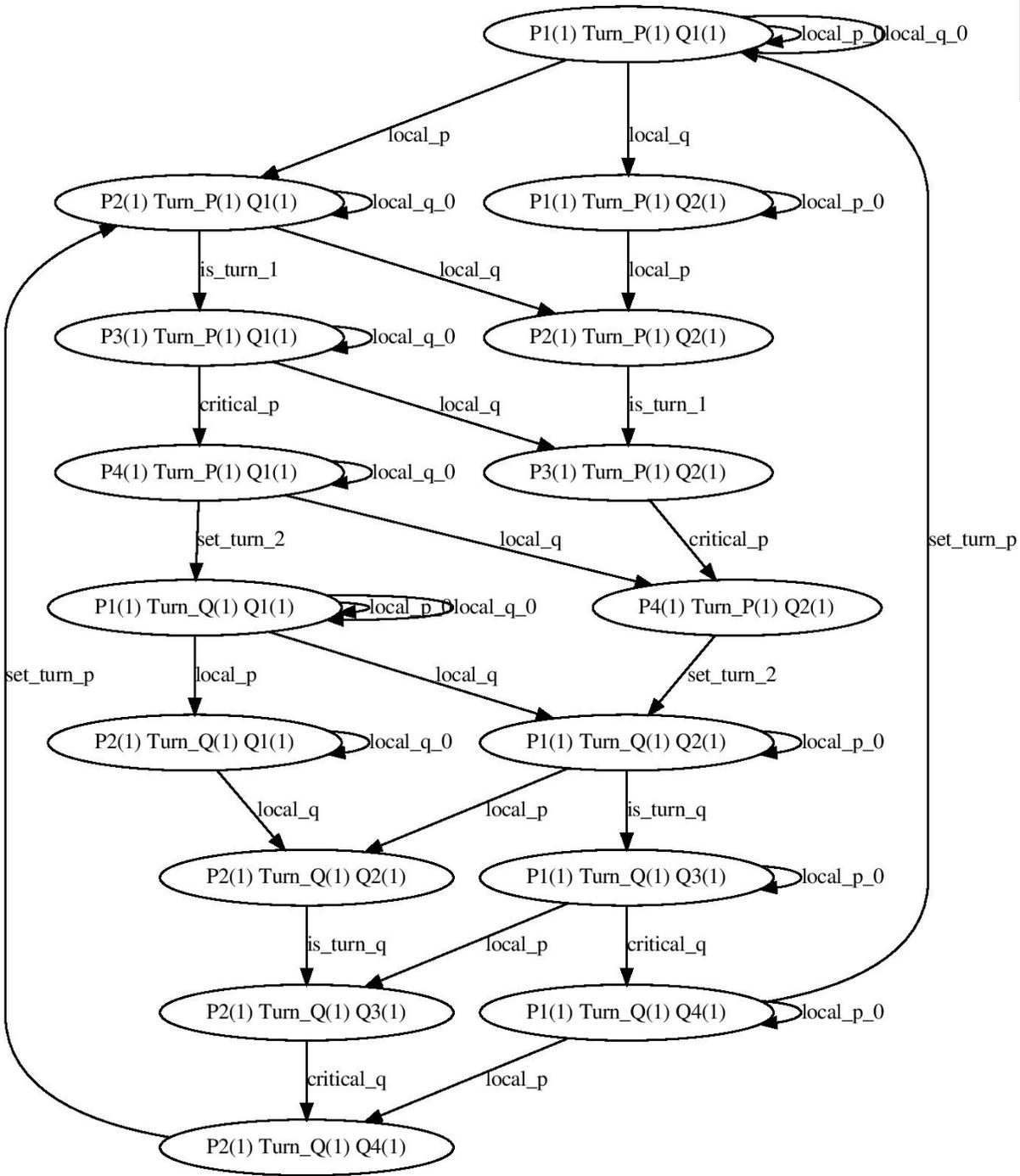
$$\begin{aligned}T_p &::= \text{is}_p.T_p + \text{set}_p.T_p + \text{set}_q.T_q \\ T_q &::= \text{is}_q.T_q + \text{set}_p.T_p + \text{set}_q.T_q\end{aligned}$$

$$\begin{aligned}P_1 &::= \text{local}_p.P_2 + \text{local}_p.P_1 \\ P_2 &::= \text{is}_p.P_3 \\ P_3 &::= \text{critical}_p.P_4 \\ P_4 &::= \text{set}_q.P_1\end{aligned}$$

$$\begin{aligned}Q_1 &::= \text{local}_q.Q_2 + \text{local}_q.Q_1 \\ Q_2 &::= \text{is}_q.Q_3 \\ Q_3 &::= \text{critical}_q.Q_4 \\ Q_4 &::= \text{set}_p.Q_1\end{aligned}$$

Seguono il Reachability Graph costruito con GreatSPN e il Derivation Graph.

**Reachability Graph**  
 Showing all markings.  
 Total markings: 16





## 2.4 Risultati

Nella tabella mostriamo i risultati ottenuti

	NuSMV	GreatSPN
Mutua Esclusione	True	True
Assenza di deadlock	False	False
No Starvation	False	False

Il risultato della possibilità di deadlock non deve stupire: la specifica non obbliga un processo a terminare la fase begin. Ne segue che anche l'assenza di starvation individuale non è garantita. NuSMV conferma quanto detto mostrandoci un trace che fa da controesempio alla formula CTL:

---

```
-- specification
    AG (p.state = wait -> AF (p.state = critical | q.state = critical)) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
    turn = 1
    p.state = begin
    q.state = begin
-> Input: 1.2 <-
    _process_selector_ = p
    running = FALSE
    q.running = FALSE
    p.running = TRUE
-> State: 1.2 <-
    p.state = wait
-> Input: 1.3 <-
-> State: 1.3 <-
    p.state = critical
-> Input: 1.4 <-
```

```
-> State: 1.4 <-
  p.state = done
-> Input: 1.5 <-
-> State: 1.5 <-
  turn = 2
  p.state = begin
-> Input: 1.6 <-
-- Loop starts here
-> State: 1.6 <-
  p.state = wait
-> Input: 1.7 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-- Loop starts here
-> State: 1.7 <-
-> Input: 1.8 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-- Loop starts here
-> State: 1.8 <-
-> Input: 1.9 <-
  _process_selector_ = main
  running = TRUE
  p.running = FALSE
-> State: 1.9 <-
```

---

La traccia mostra che il processo q rimane nella fase begin e p, dopo essere entrato nella regione critica una volta, rimane bloccato in begin. Lo stesso trace mostra la possibilità di starvation del processo. La rete modellata con GreatSPN ci conferma quanto visto con NuSMV. Inoltre il trace di GreatSPN ci mostra che non potendo forzare la fairness del modello, i processi possono iterare indefinitamente nella transizione iniziale, impedendo quindi il progresso dell'altro processo.

---

Initial state is: P1(1), Turn\_P(1), Q1(1)

Initial state satisfies: E F (not ((not (Q1 = 1)) or (not E G (not (Q3 = 1))))).

1: P1(1), Turn\_P(1), Q1(1)

State 1. does not satisfy: ((not (Q1 = 1)) or (not E G (not (Q3 = 1)))).

1.1: P1(1), Turn\_P(1), Q1(1)

State 1.1.L. satisfies: (Q1 = 1).

State 1.1.R. satisfies: E G (not (Q3 = 1)). Start of loop.

1.1.R.1: P1(1), Turn\_P(1), Q1(1)

State 1.1.R.1. does not satisfy: (Q3 = 1).

1.1.R.2: loop back to state 1.1.R.1.

---

Prendiamo in considerazione la seguente formula CTL

$$AG (w_p \rightarrow EF c_p)$$

Sia GreatSPN che NuSMV ci mostrano che il sistema modellato rispetta questa proprietà. Questo significa che un processo in attesa di entrare nella sezione critica ha la possibilità di compiere progresso, ma solo nel caso in cui, come si intuisce dal controesempio precedente, l'altro processo decida di entrare in attesa a sua volta.

### 3 Algoritmo 3.5

Due processi iterano all'infinito seguendo questo pseudocodice

```
while true:
    await turn = ID
    turn <- (ID%2)+1
```

#### 3.1 NuSMV

Si è deciso di modellare l'algoritmo usando per ognuno dei due processi un'enumerazione di due stati e un contatore di turni intero

state: {await, done};

Non si è utilizzata la direttiva *process* di NuSMV dato che un automa a stati finiti è sufficiente a dimostrare le proprietà del modello.

---

```
MODULE main
VAR
    turn: 1..2;
    p: proc(turn, 1);
    q: proc(turn, 2);
ASSIGN
    init(turn) := 1;
    next(turn) := case
```

```
        p.state = done: 2;  
        q.state = done: 1;  
        TRUE: turn;  
    esac;
```

CTLSPEC -- no mutual exclusion

```
    AG (p.state != done | q.state != done)
```

CTLSPEC -- no deadlock

```
    AG ((p.state = await | q.state = await) -> AF (p.state = done | q.  
        state = done))
```

CTLSPEC -- no individual starvation

```
    AG (p.state = await -> AF p.state = done)
```

CTLSPEC

```
    AG (q.state = await -> AF q.state = done)
```

CTLSPEC -- prova: path senza starvation

```
    AG (q.state = await -> EF q.state = done)
```

LTLSPPEC -- no mutual exclusion

```
    G (p.state != done | q.state != done)
```

LTLSPPEC -- no deadlock

```
    G ((p.state = await | q.state = await) -> F (p.state = done | q.  
        state = done))
```

LTLSPPEC -- no individual starvation

```
    G (p.state = await -> F p.state = done)
```

LTLSPPEC

```
    G (q.state = await -> F q.state = done)
```

MODULE proc(turn, id)

VAR

```
state: {await, done};
ASSIGN
init(state) := await;
next(state) := case
    turn = id: {await, done};
    state = await: await;
    state = done: await;
esac;
```

---

### 3.2 GreatSPN

Il codice utilizzato per le proprietà CTL è il seguente:

```
AG(!#Await_P == 1 || !#Await_Q == 1) = true
```

```
AG ((#Wait_P==1 || #Await_Q == 1) -> AF (#Done_P == 1 || #Done_Q == 1)) = false
```

```
AG (#Await_P==1 -> AF (#Done_P == 1)) = false
```

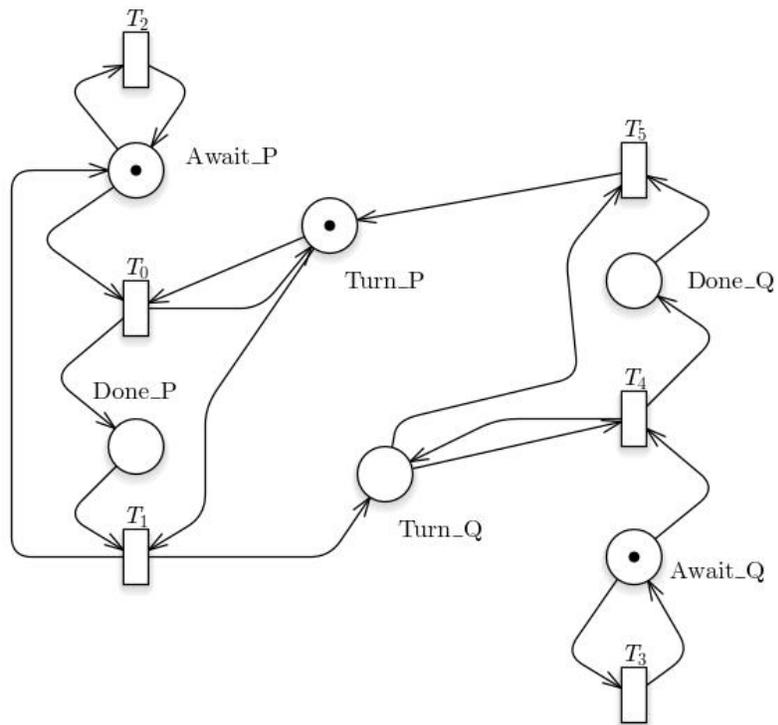


Figure 1: Rete 3.5

### 3.3 Risultati

Nella tabella mostriamo i risultati ottenuti

	NuSMV	GreatSPN
Mutua Esclusione	true	true
Assenza di deadlock	false	false
No Starvation	false	false

I risultati confermano il fatto che questo algoritmo è solamente una versione semplificata del precedente (*begin* e *wait* sono stati fusi in *await* e *done* rimosso) e pertanto il trace di controesempio fornito da NuSMV e da GreatSPN presentano riflettono i risultati di questa semplificazione.

- Possibilità di deadlock:

```
- specification AG ((p.state = await | q.state = await) -> AF (p.state = done | q
- as demonstrated by the following execution sequence
```

```
Trace Description: CTL Counterexample
```

```
Trace Type: Counterexample
```

```
- Loop starts here
```

```
-> State: 1.1 <-
```

```
    turn = 1
```

```
    p.state = await
```

```
    q.state = await
```

```
-> State: 1.2 <-
```

- Possibilità di starvation: i processi non compiono progresso iterando infinite volte su *await*.

```
- specification AG (p.state = await -> AF p.state = done) is false
```

```
- as demonstrated by the following execution sequence
```

```
Trace Description: CTL Counterexample
```

```
Trace Type: Counterexample
```

```
- Loop starts here
```

```
-> State: 2.1 <-
```

```
    turn = 1
```

```
    p.state = await
```

```

    q.state = await
-> State: 2.2 <-

- specification AG (q.state = await -> AF q.state = done) is false
- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
- Loop starts here
-> State: 3.1 <-
    turn = 1
    p.state = await
    q.state = await
-> State: 3.2 <-

```

- Possibilità di compiere progresso:

```

specification AG (q.state = await -> EF q.state = done) is true

```

## 4 Algoritmo 3.6

Due processi iterano all'infinito seguendo questo pseudocodice

while true:

```

    non-critical section
    await wantQ = false
    wantP <- true
    critical section
    wantP <- false

```

### 4.1 NuSMV

Si è deciso di modellare l'algoritmo usando per ognuno dei due processi usando 5 stati

state: {local, await, critical, setTrue, setFalse};

---

```
MODULE main
VAR
  wantP: boolean;
  wantQ: boolean;
  p: process proc(wantP, wantQ);
  q: process proc(wantQ, wantP);
ASSIGN
  init(wantP) := FALSE;
  init(wantQ) := FALSE;

CTLSPEC -- no mutual exclusion
  AG (p.state != critical | q.state != critical)

CTLSPEC -- no deadlock
  AG ((p.state = await | q.state = await) -> AF (p.state = critical
    | q.state = critical))

CTLSPEC -- no individual starvation
  AG (p.state = await -> AF p.state = critical)
CTLSPEC
  AG (q.state = await -> AF q.state = critical)

CTLSPEC -- prova: path senza starvation
  AG (q.state = await -> EF q.state = critical)

LTLSPPEC -- no mutual exclusion
  G (p.state != critical | q.state != critical)
LTLSPPEC -- no deadlock
  G ((p.state = await | q.state = await) -> F (p.state = critical |
    q.state = critical))
LTLSPPEC -- no individual starvation
  G (p.state = await -> F p.state = critical)
```

```
LTLSPEC
  G (q.state = await -> F q.state = critical)

MODULE proc(mine, other)
VAR
  state: {local, await, critical, setTrue, setFalse};
ASSIGN
  init(state) := local;
  next(state) := case
    state = local: {local, await};
    state = await & other = FALSE: setTrue;
    state = await: await;
    state = setTrue: critical;
    state = critical: setFalse;
    state = setFalse: local;
  esac;
  next(mine) := case
    state = setTrue: TRUE;
    state = setFalse: FALSE;
    TRUE: mine;
  esac;

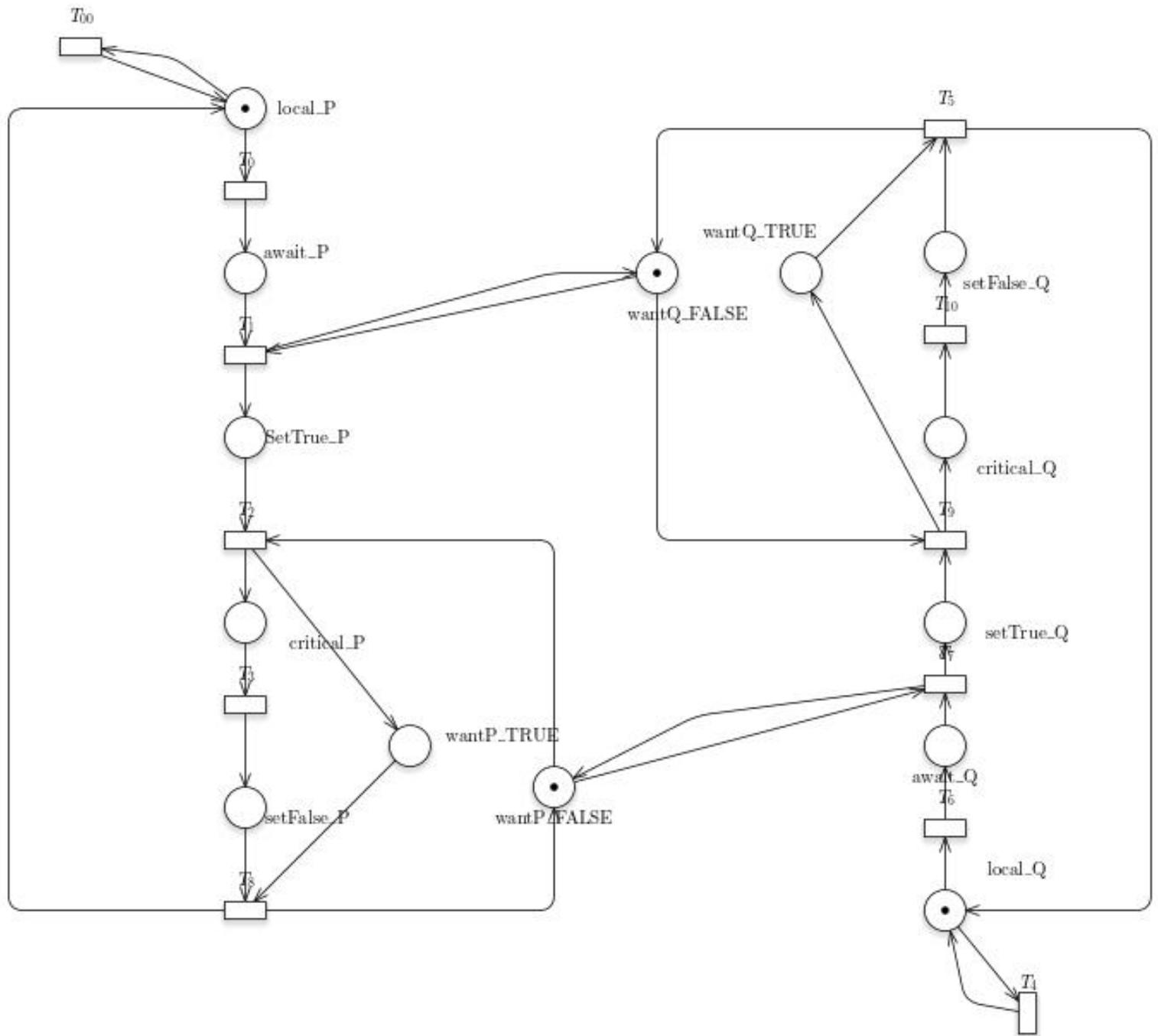
FAIRNESS
  running
```

---

## 4.2 GreatSPN

Il codice utilizzato per le proprietà CTL è il seguente:

```
AG (!(#P4 == 1) || !(#Q4 == 1))
AG ((#P2==1 || #Q2 == 1) -> AF (#P4 == 1 || #Q4 == 1))
AG (#P1 == 1 -> EF(#Q4==1 || #Q4 == 1))
AG (#P2 == 1 -> AF (#P4 == 1))
AG (#Q2 == 1 -> AF (#Q4 == 1))
```



### 4.3 Algebra dei processi

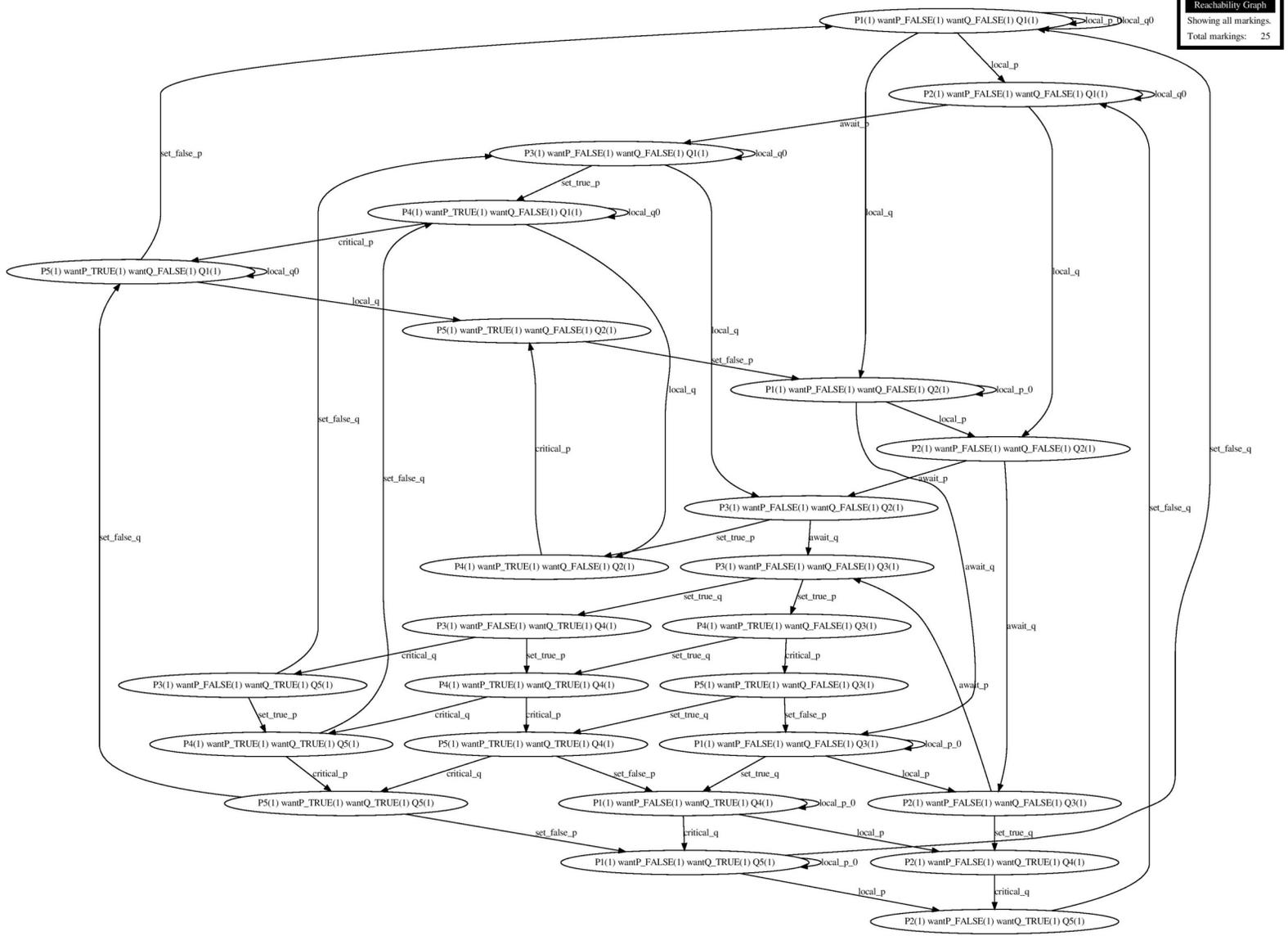
L'algoritmo 3.6 modellato secondo NuSMV e GreatSPN mostra che non c'è la mutua esclusione. Questo viene evidenziato anche dal fatto che i nodi del Reachability Graph corrispondono al prodotto cartesiano  $P \times Q$ . Anche il Derivation Graph del modello in algebra dei processi ha 25 nodi ed è equivalente al Reachability Graph.

$$\begin{aligned} \text{System} &= \{(P_1 \parallel Q_1) \parallel (\text{Want}_{p0} \parallel \text{Want}_{q0})\} / \text{Sync} \\ \text{Sync} &= \{ \text{isTrue}_p, \text{isFalse}_p, \text{setTrue}_p, \text{setFalse}_p, \\ &\quad \text{isTrue}_q, \text{isFalse}_q, \text{setTrue}_q, \text{setFalse}_q \} \\ S &= \{ \text{local}_p, \text{critical}_p, \text{local}_q, \text{critical}_q \} \end{aligned}$$

$$\begin{aligned} \text{Want}_{p1} &::= \text{isTrue}_p.\text{Want}_{p1} + \text{setTrue}_p.\text{Want}_{p1} + \text{setFalse}_p.\text{Want}_{p0} \\ \text{Want}_{p0} &::= \text{isFalse}_p.\text{Want}_{p0} + \text{setFalse}_p.\text{Want}_{p0} + \text{setTrue}_p.\text{Want}_{p1} \\ P_1 &::= \text{local}_p.P_1 + \text{local}_p.P_2 \\ P_2 &::= \text{isFalse}_q.P_3 \\ P_3 &::= \text{setTrue}_p.P_4 \\ P_4 &::= \text{critical}_p.P_5 \\ P_5 &::= \text{setFalse}_p.P_1 \end{aligned}$$

$$\begin{aligned} \text{Want}_{q1} &::= \text{isTrue}_q.\text{Want}_{q1} + \text{setTrue}_q.\text{Want}_{q1} + \text{setFalse}_q.\text{Want}_{q0} \\ \text{Want}_{q0} &::= \text{isFalse}_q.\text{Want}_{q0} + \text{setFalse}_q.\text{Want}_{q0} + \text{setTrue}_q.\text{Want}_{q1} \\ Q_1 &::= \text{local}_q.Q_1 + \text{local}_q.Q_2 \\ Q_2 &::= \text{isFalse}_p.Q_3 \\ Q_3 &::= \text{setTrue}_q.Q_4 \\ Q_4 &::= \text{critical}_q.Q_5 \\ Q_5 &::= \text{setFalse}_q.Q_1 \end{aligned}$$

Seguono il Reachability Graph costruito con GreatSPN e il Derivation Graph.





## 4.4 Risultati

Nella tabella mostriamo i risultati ottenuti

	NuSMV	GreatSPN
Mutua Esclusione	false	false
Assenza di deadlock	true	false
No Starvation	false	false

Il trace di NuSMV mostra che la mutua esclusione può non essere rispettata dato che le due variabili *wantP* e *wantQ* possono essere contemporaneamente false.

---

```
-- specification AG (p.state != critical | q.state != critical) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  wantP = FALSE
  wantQ = FALSE
  p.state = local
  q.state = local
-> Input: 1.2 <-
  _process_selector_ = p
  running = FALSE
  q.running = FALSE
  p.running = TRUE
-> State: 1.2 <-
  p.state = await
-> Input: 1.3 <-
-> State: 1.3 <-
  p.state = setTrue
-> Input: 1.4 <-
  _process_selector_ = q
```

```
q.running = TRUE
p.running = FALSE
-> State: 1.4 <-
  q.state = await
-> Input: 1.5 <-
-> State: 1.5 <-
  q.state = setTrue
-> Input: 1.6 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 1.6 <-
  wantP = TRUE
  p.state = critical
-> Input: 1.7 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-> State: 1.7 <-
  wantQ = TRUE
  q.state = critical
```

---

L'incoerenza del risultato dell'assenza di deadlock è spiegabile dal fatto che nel caso di NuSMV non è possibile che un processo rimanga nel primo stato (*local*) per un tempo indefinito mentre nel caso di GreatSPN è possibile che il processo P vada nello spazio *await* e il processo Q decida di rimanere nel loop *local\_Q* all'infinito. Notiamo che se forziamo i processi a fare del progresso dallo stato *local*, allora la formula CTL

$$AG(w_p \rightarrow AF(c_p \vee c_q))$$

risulta rispettata. Ancora meglio, piuttosto che rimuovere una transizione possibile, possiamo restringere la verifica dell'assenza di deadlock alla seguente formula CTL

$$AG(w_p \rightarrow EF(c_p \parallel c_q))$$

$$AG(\#await\_P == 1 \rightarrow EF(\#critical\_Q == 1 \parallel \#critical\_P == 1))$$

che risulta rispettata. L'assenza di starvation individuale non è rispettata né in NuSmv né in GreatSPN in quanto è possibile che uno dei due processi continui ad accedere alla sezione critica senza permettere all'altro di fare lo stesso.

---

```
-- specification AG (q.state = await -> AF q.state = critical) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 3.1 <-
  wantP = FALSE
  wantQ = FALSE
  p.state = local
  q.state = local
-> Input: 3.2 <-
  _process_selector_ = q
  running = FALSE
  q.running = TRUE
```

```

    p.running = FALSE
-- Loop starts here
-> State: 3.2 <-
    q.state = await
-> Input: 3.3 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
-> State: 3.3 <-
    p.state = await
-> Input: 3.4 <-
-> State: 3.4 <-
    p.state = setTrue
-> Input: 3.5 <-
-> State: 3.5 <-
    wantP = TRUE
    p.state = critical
-> Input: 3.6 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
-> State: 3.6 <-
-> Input: 3.7 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
-> State: 3.7 <-
    p.state = setFalse
-> Input: 3.8 <-
-> State: 3.8 <-
    wantP = FALSE
    p.state = local

```



## 5 Algoritmo 3.8

Due processi iterano all'infinito seguendo questo pseudocodice

```
while true:
    non-critical section
    wantP <- true
    await wantQ = false
    critical section
    wantP <- false
```

### 5.1 NuSMV

Si è deciso di modellare l'algoritmo allo stesso modo del precedente.

---

```
MODULE main
VAR
    wantP: boolean;
    wantQ: boolean;
    p: process proc(wantP, wantQ);
    q: process proc(wantQ, wantP);
ASSIGN
    init(wantP) := FALSE;
    init(wantQ) := FALSE;

CTLSPEC -- no mutual exclusion
    AG (p.state != critical | q.state != critical)

CTLSPEC -- no deadlock
    AG ((p.state = await | q.state = await) -> AF (p.state = critical
    | q.state = critical))

CTLSPEC -- no individual starvation
    AG (p.state = await -> AF p.state = critical)
CTLSPEC
```

```

AG (q.state = await -> AF q.state = critical)

CTLSPPEC -- prova: path senza starvation
AG (q.state = await -> EF q.state = critical)

LTLSPPEC -- no mutual exclusion
G (p.state != critical | q.state != critical)
LTLSPPEC -- no deadlock
G ((p.state = await | q.state = await) -> F (p.state = critical |
    q.state = critical))
LTLSPPEC -- no individual starvation
G (p.state = await -> F p.state = critical)
LTLSPPEC
G (q.state = await -> F q.state = critical)

```

```

MODULE proc(mine, other)
VAR
    state: {local, await, critical, setTrue, setFalse};
ASSIGN
    init(state) := local;
    next(state) := case
        state = local: {local, setTrue};
        state = await & other = FALSE: critical;
        state = await: await;
        state = setTrue: await;
        state = critical: setFalse;
        state = setFalse: local;
    esac;
    next(mine) := case
        state = setTrue: TRUE;
        state = setFalse: FALSE;
        TRUE: mine;
    esac;

```

```

FAIRNESS

```

running

---

## 5.2 GreatSPN

Il codice utilizzato per le proprietà CTL è il seguente:

```

AG(!(#critical_P == 1) || !(#critical_Q == 1))
AG ((#await_P==1 || #await_Q == 1) -> AF (#critical_P == 1 || #critical_Q == 1))
AG (#await_P==1 -> AF (#critical_P == 1))
AG (#await_Q==1 -> AF (#critical_Q == 1))
  
```

Inoltre, per confermare alcune ipotesi, si è testata anche la seguente formula CTL:

```

AG(#await_P == 1 -> EF(#critical_Q==1 \vert\vert \vert\vert #critical_P == 1))
  
```

che conferma la presenza di deadlock causata dalle due variabili booleane.

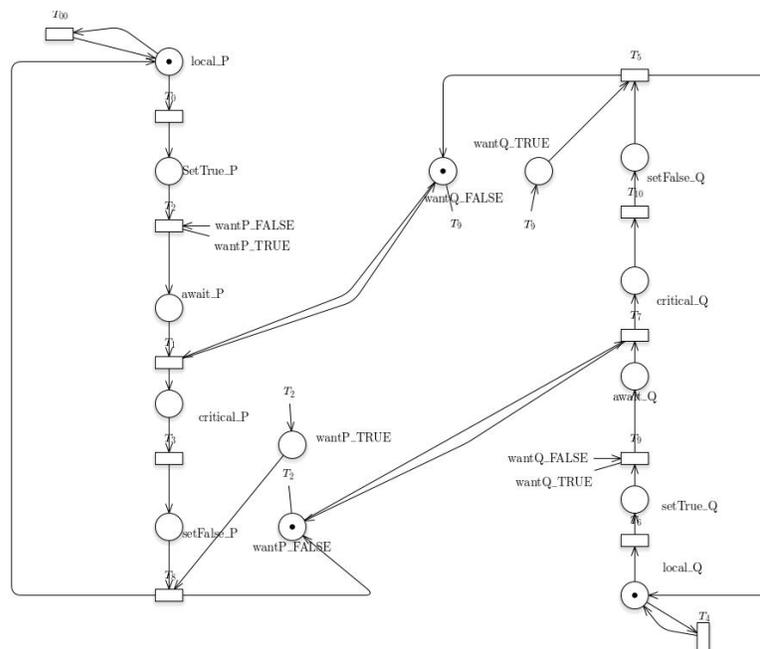


Figure 2: Rete 3.8

### 5.3 Risultati

Nella tabella mostriamo i risultati ottenuti

	NuSMV	GreatSPN
Mutua Esclusione	True	True
Assenza di deadlock	False	False
No Starvation	False	False

Questo algoritmo rispetto al precedente garantisce la mutua esclusione, ma non ci permette di evitare il deadlock (e di conseguenza neanche la starvation individuale).

Di seguito riportiamo la traccia di NuSMV che mostra che il deadlock avviene quando i progetti eseguono *setTrue* nello stesso momento.

---

```
-- specification AG ((p.state = await | q.state = await) -> AF (p.state = critical | q
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  wantP = FALSE
  wantQ = FALSE
  p.state = local
  q.state = local
-> Input: 1.2 <-
  _process_selector_ = p
  running = FALSE
  q.running = FALSE
  p.running = TRUE
-> State: 1.2 <-
  p.state = setTrue
-> Input: 1.3 <-
-> State: 1.3 <-
  wantP = TRUE
```

```

    p.state = await
-> Input: 1.4 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
-> State: 1.4 <-
    q.state = setTrue
-> Input: 1.5 <-
-- Loop starts here
-> State: 1.5 <-
    wantQ = TRUE
    q.state = await
-> Input: 1.6 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
-- Loop starts here
-> State: 1.6 <-
-> Input: 1.7 <-
    _process_selector_ = main
    running = TRUE
    p.running = FALSE
-- Loop starts here
-> State: 1.7 <-
-> Input: 1.8 <-
    _process_selector_ = q
    running = FALSE
    q.running = TRUE
-- Loop starts here
-> State: 1.8 <-
-> Input: 1.9 <-
    _process_selector_ = p

```

```
q.running = FALSE
p.running = TRUE
-- Loop starts here
-> State: 1.9 <-
-> Input: 1.10 <-
  _process_selector_ = main
  running = TRUE
  p.running = FALSE
-> State: 1.10 <-
```

---

Mostriamo invece il controesempio generato da GreatSPN che ci mostra una condizione di starvation individuale, dove, come in precedenza, il processo Q rimane in *local\_Q*.

---

Generated counter-example:

===== Trace =====

Initial state is: local\_P(1), wantP\_FALSE(1), wantQ\_FALSE(1), local\_Q(1)

Initial state satisfies:

E F (not ((not (await\_P = 1)) or (not E G (not (critical\_P = 1))))).

1: local\_P(1), wantP\_FALSE(1), wantQ\_FALSE(1), local\_Q(1)

State 1. satisfies: ((not (await\_P = 1)) or (not E G (not (critical\_P = 1)))).

1.1: local\_P(1), wantP\_FALSE(1), wantQ\_FALSE(1), local\_Q(1)

State 1.1. does not satisfy: (await\_P = 1).

2: SetTrue\_P(1), wantP\_FALSE(1), wantQ\_FALSE(1), local\_Q(1)

State 2. satisfies: ((not (await\_P = 1)) or (not E G (not (critical\_P = 1)))).

2.1: SetTrue\_P(1), wantP\_FALSE(1), wantQ\_FALSE(1), local\_Q(1)

State 2.1. does not satisfy: (await\_P = 1).

3: wantP\_TRUE(1), await\_P(1), wantQ\_FALSE(1), local\_Q(1)

State 3. does not satisfy: ((not (await\_P = 1)) or (not E G (not (critical\_P = 1))))

3.1: wantP\_TRUE(1), await\_P(1), wantQ\_FALSE(1), local\_Q(1)

State 3.1.L. satisfies: (await\_P = 1).

State 3.1.R. satisfies: E G (not (critical\_P = 1)). Start of loop.

3.1.R.1: wantP\_TRUE(1), await\_P(1), wantQ\_FALSE(1), local\_Q(1)

State 3.1.R.1. does not satisfy: (critical\_P = 1).

3.1.R.2: loop back to state 3.1.R.1.

---

## 6 Algoritmo 3.9

Due processi iterano all'infinito seguendo questo pseudocodice

```
while true:
    non-critical section
    wantP <- true
    while wantQ
wantP <- false
wantP <- true
    critical section
    wantP <- false
```

(l'altro processo segue uno pseudocodice simmetrico)

### 6.1 NuSMV

Si è deciso di modellare l'algoritmo usando per ognuno dei due processi utilizzando l'espressione *process* per simulare di NuSMV e per ciascun process una variabile *state* di tipo enumerazione

```
state: {local, critical, setTrue, setFalse, resetTrue, resetFalse};
```

Benché non fosse necessario distinguere il set della variabile booleana *wantP*, si è preferito farlo in quanto l'enumerazione di tutti gli stati possibili, come evidenziato dal codice che segue, non risulta complesso.

---

```
MODULE main
VAR
    wantP: boolean;
    wantQ: boolean;
    p: process proc(wantP, wantQ); -- PROCESS: http://nusmv.fbk.eu/
        NuSMV/userman/v21/nusmv\_3.html#SEC27
    q: process proc(wantQ, wantP);
ASSIGN
```

```

    init(wantP) := FALSE;
    init(wantQ) := FALSE;

CTLSPEC -- no mutual exclusion
    AG (p.state != critical | q.state != critical)

CTLSPEC -- no deadlock
    AG ((p.state = setTrue | q.state = setTrue) -> AF (p.state =
        critical | q.state = critical))

CTLSPEC -- no individual starvation
    AG (p.state = setTrue -> AF p.state = critical)
CTLSPEC
    AG (q.state = setTrue -> AF q.state = critical)

LTLSPEC -- no mutual exclusion
    G (p.state != critical | q.state != critical)
LTLSPEC -- no deadlock
    G ((p.state = setTrue | q.state = setTrue) -> F (p.state =
        critical | q.state = critical))
LTLSPEC -- no individual starvation
    G (p.state = setTrue -> F p.state = critical)
LTLSPEC
    G (q.state = setTrue -> F q.state = critical)

MODULE proc(mine, other)
VAR
    state: {local, critical, setTrue, setFalse, resetTrue,
            resetFalse};
ASSIGN
    init(state) := local;
    next(state) := case
        state = local: {local, setTrue};
        state = setTrue & other = TRUE: resetFalse;
        state = setTrue & other = FALSE: critical;
        state = resetFalse: resetTrue;

```

```
        state = resetTrue & other = TRUE: resetFalse;
        state = resetTrue & other = FALSE: critical;
        state = critical: setFalse;
        state = setFalse: local;
    esac;
next(mine) := case
    state = setTrue: TRUE;
    state = setFalse: FALSE;
    state = resetTrue: TRUE;
    state = resetFalse: FALSE;
    TRUE: mine;
esac;
FAIRNESS
    running
```

---

## 6.2 GreatSPN

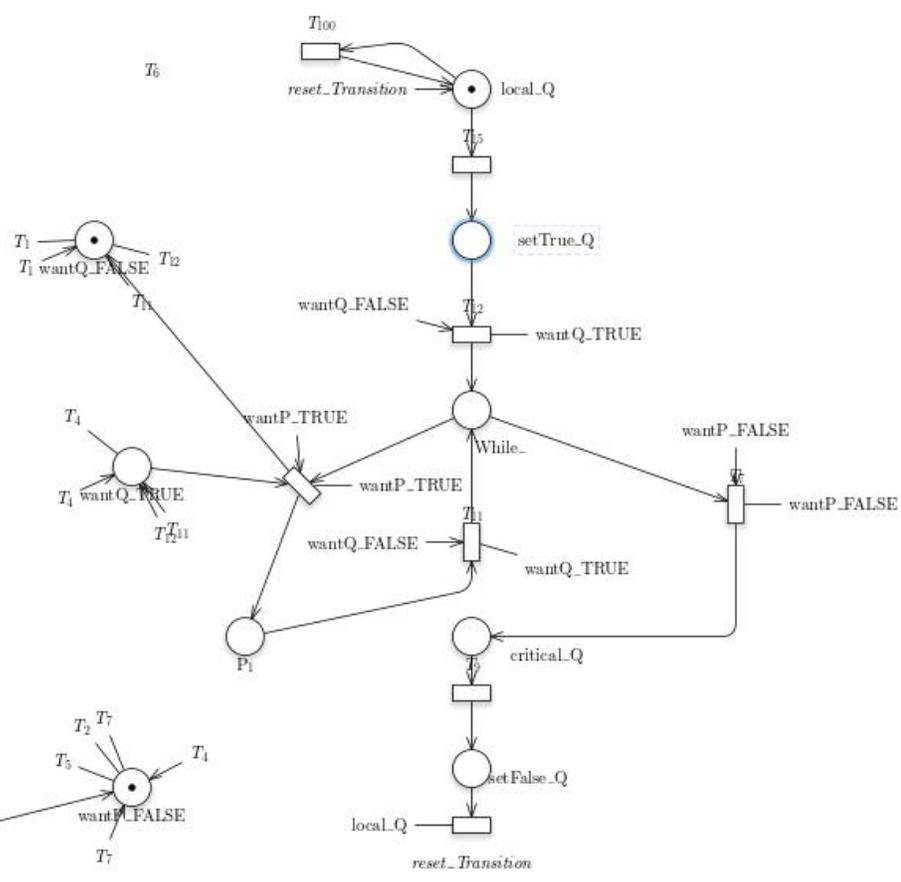
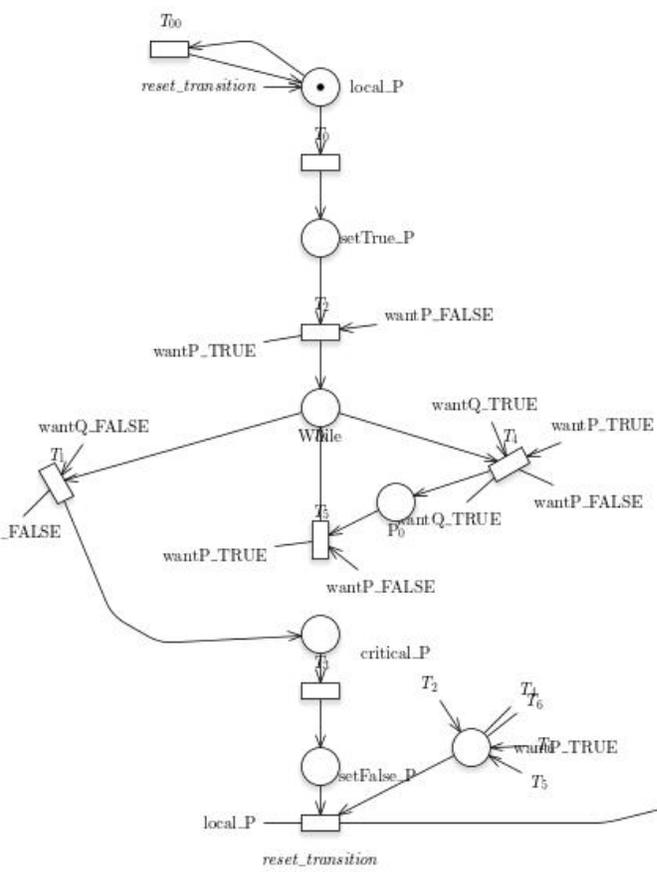
Il codice utilizzato per le proprietà CTL è il seguente:

```
AG(!(#critical_P == 1) || !(#critical_Q == 1))
AG ((#setTrue_P==1 || #setTrue_Q == 1) -> AF (#critical_P == 1 || #critical_Q == 1))
AG (#setTrue_P==1 -> AF (#critical_P == 1))
AG (#setTrue_Q==1 -> AF (#critical_Q == 1))
```

Inoltre, per confermare alcune ipotesi, si è testata anche la seguente formula CTL:

```
AG(#setTrue_P == 1 -> EF(#critical_Q==1    #critical_P == 1))
```

che conferma la presenza di deadlock causata dalle due variabili booleane.



### 6.3 Risultati

Nella tabella mostriamo i risultati ottenuti

	NuSMV	GreatSPN
Mutua Esclusione	true	true
Assenza di deadlock	false	false
No Starvation	false	false

Il deadlock si verifica quando i entrambe le variabili booleane sono uguali a *true*. Ad esempio, se si esegue in locksetop il loop con la condizione booleana sulla variabile dell'altro processo, si verifica la condizione di deadlock. Viene riportato il trace di NuSMV che conferma questa ipotesi. GreatSPN fallisce per mancanza di RAM sulla macchina usata ma è facile riprodurre il deadlock manualmente.

```
-- specification AG ((p.state = setTrue | q.state = setTrue) -> AF (p.state = critical
-- as demonstrated by the following execution sequence
```

Trace Description: CTL Counterexample

Trace Type: Counterexample

```
-> State: 1.1 <-
  wantP = FALSE
  wantQ = FALSE
  p.state = local
  q.state = local
-> Input: 1.2 <-
  _process_selector_ = p
  running = FALSE
  q.running = FALSE
  p.running = TRUE
-> State: 1.2 <-
  p.state = setTrue
-> Input: 1.3 <-
-> State: 1.3 <-
  wantP = TRUE
```

```

    p.state = critical
-> Input: 1.4 <-
-> State: 1.4 <-
    p.state = setFalse
-> Input: 1.5 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
-> State: 1.5 <-
    q.state = setTrue
-> Input: 1.6 <-
-> State: 1.6 <-
    wantQ = TRUE
    q.state = resetFalse
-> Input: 1.7 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
-> State: 1.7 <-
    wantP = FALSE
    p.state = local
-> Input: 1.8 <-
-> State: 1.8 <-
    p.state = setTrue
-> Input: 1.9 <-
-- Loop starts here
-> State: 1.9 <-
    wantP = TRUE
    p.state = resetFalse
-> Input: 1.10 <-
    _process_selector_ = q
    q.running = TRUE

```

```
    p.running = FALSE
-> State: 1.10 <-
    wantQ = FALSE
    q.state = resetTrue
-> Input: 1.11 <-
-- Loop starts here
-> State: 1.11 <-
    wantQ = TRUE
    q.state = resetFalse
-> Input: 1.12 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
-> State: 1.12 <-
    wantP = FALSE
    p.state = resetTrue
-> Input: 1.13 <-
-- Loop starts here
-> State: 1.13 <-
    wantP = TRUE
    p.state = resetFalse
-> Input: 1.14 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
-> State: 1.14 <-
    wantQ = FALSE
    q.state = resetTrue
-> Input: 1.15 <-
-- Loop starts here
-> State: 1.15 <-
    wantQ = TRUE
```

```
    q.state = resetFalse
-> Input: 1.16 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
-> State: 1.16 <-
    wantP = FALSE
    p.state = resetTrue
-> Input: 1.17 <-
-> State: 1.17 <-
    wantP = TRUE
    p.state = resetFalse
```

---

Di seguito il trace che conferma la presenza di starvation individuale, sia in GreatSPN che NuSMV.

---

Generated counter-example:

```
===== Trace =====
Initial state is: local_Q(1), wantQ_FALSE(1), wantP_FALSE(1), local_P(1)
Initial state satisfies:
  E F (not ((not (setTrue_Q = 1)) or (not E G (not (critical_Q = 1))))).

1: local_Q(1), wantQ_FALSE(1), wantP_FALSE(1), local_P(1)
  State 1. satisfies: ((not (setTrue_Q = 1)) or (not E G (not (critical_Q = 1)))).

  1.1: local_Q(1), wantQ_FALSE(1), wantP_FALSE(1), local_P(1)
    State 1.1. does not satisfy: (setTrue_Q = 1).

2: setTrue_Q(1), wantQ_FALSE(1), wantP_FALSE(1), local_P(1)
  State 2. does not satisfy: ((not (setTrue_Q = 1)) or (not E G (not (critical_Q = 1)))).

  2.1: setTrue_Q(1), wantQ_FALSE(1), wantP_FALSE(1), local_P(1)
    State 2.1.L. satisfies: (setTrue_Q = 1).

    State 2.1.R. satisfies: E G (not (critical_Q = 1)). Start of loop.

    2.1.R.1: setTrue_Q(1), wantQ_FALSE(1), wantP_FALSE(1), local_P(1)
      State 2.1.R.1. does not satisfy: (critical_Q = 1).

    2.1.R.2: loop back to state 2.1.R.1.
```

---

---

```
-- specification AG (q.state = setTrue -> AF q.state = critical) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 3.1 <-
    wantP = FALSE
    wantQ = FALSE
    p.state = local
    q.state = local
-> Input: 3.2 <-
    _process_selector_ = q
    running = FALSE
    q.running = TRUE
    p.running = FALSE
-> State: 3.2 <-
    q.state = setTrue
-> Input: 3.3 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
-> State: 3.3 <-
    p.state = setTrue
-> Input: 3.4 <-
-> State: 3.4 <-
    wantP = TRUE
    p.state = critical
-> Input: 3.5 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
```

```

-> State: 3.5 <-
  wantQ = TRUE
  q.state = resetFalse
-> Input: 3.6 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-- Loop starts here
-> State: 3.6 <-
  p.state = setFalse
-> Input: 3.7 <-
  _process_selector_ = main
  running = TRUE
  p.running = FALSE
-- Loop starts here
-> State: 3.7 <-
-> Input: 3.8 <-
  _process_selector_ = q
  running = FALSE
  q.running = TRUE
-> State: 3.8 <-
  wantQ = FALSE
  q.state = resetTrue
-> Input: 3.9 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 3.9 <-
  wantP = FALSE
  p.state = local
-> Input: 3.10 <-
-> State: 3.10 <-

```

```
p.state = setTrue
-> Input: 3.11 <-
-> State: 3.11 <-
  wantP = TRUE
  p.state = critical
-> Input: 3.12 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-> State: 3.12 <-
  wantQ = TRUE
  q.state = resetFalse
-> Input: 3.13 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 3.13 <-
  p.state = setFalse
```

---

## 7 Algoritmo di Dekker per la mutua esclusione

Due processi iterano all'infinito seguendo questo pseudocodice

```
while true:
    non-critical section
    wantP <- true
    while wantQ:
if turn = 2:
    wantp <- false
    await turn = 1
    wantp <- true
    critical section
    turn <- 2
    wantp <- false
```

(l'altro processo segue uno pseudocodice simmetrico)

### 7.1 NuSMV

Si è deciso di modellare l'algoritmo usando per ognuno dei due processi usando 9 stati

- state: {local, set\_true, while, loop\_set\_false, await, loop\_set\_true, critical, switch\_turn, set\_false};

---

```
MODULE main
VAR
    turn: 1..2;
    wantP: boolean;
    wantQ: boolean;
    p: process proc(turn, 1, 2, wantP, wantQ);
    q: process proc(turn, 2, 1, wantQ, wantP);
ASSIGN
    init(turn) := 1;
```

```

init(wantP) := FALSE;
init(wantQ) := FALSE;

CTLSPEC -- mutual exclusion
  AG !(p.state = critical & q.state = critical)

CTLSPEC -- no deadlock
  AG ((p.state = await | q.state = await) -> AF(p.state = critical
    | q.state = critical));

CTLSPEC -- no starvation
  AG (p.state = await -> AF(p.state = critical));
CTLSPEC -- no starvation
  AG (q.state = await -> AF(q.state = critical));

LTLSPEC -- mutual exclusion
  G !(p.state = critical & q.state = critical)
LTLSPEC -- no deadlock
  G ((p.state = await | q.state = await) -> F(p.state = critical |
    q.state = critical));
LTLSPEC -- no starvation
  G (p.state = await -> F(p.state = critical));
LTLSPEC -- no starvation
  G (q.state = await -> F(q.state = critical));

MODULE proc(turn, ID, otherID, mine, other)
VAR
  state: {local, set_true, while, loop_set_false, await,
    loop_set_true, critical, switch_turn, set_false};

ASSIGN
  init(state) := local;
  next(state) := case
    state = local: {local, set_true};
    state = set_true: while;
    state = while & other = FALSE: critical;

```

```

state = critical: switch_turn;
state = switch_turn: set_false;
state = set_false: local;

-- while loop
state = while & other = TRUE & turn = ID: while;
state = while & other = TRUE & turn = otherID:
    loop_set_false;
state = loop_set_false: await;
state = await & turn = otherID: await;
state = await & turn = ID: loop_set_true;
state = loop_set_true: while;
esac;
next(turn) := case
    state = switch_turn: otherID;
    TRUE: turn;
esac;
next(mine) := case -- change want{P,Q}
    state = set_true: TRUE;
    state = loop_set_true: TRUE;
    state = set_false: FALSE;
    state = loop_set_false: FALSE;
    TRUE: mine;
esac;

```

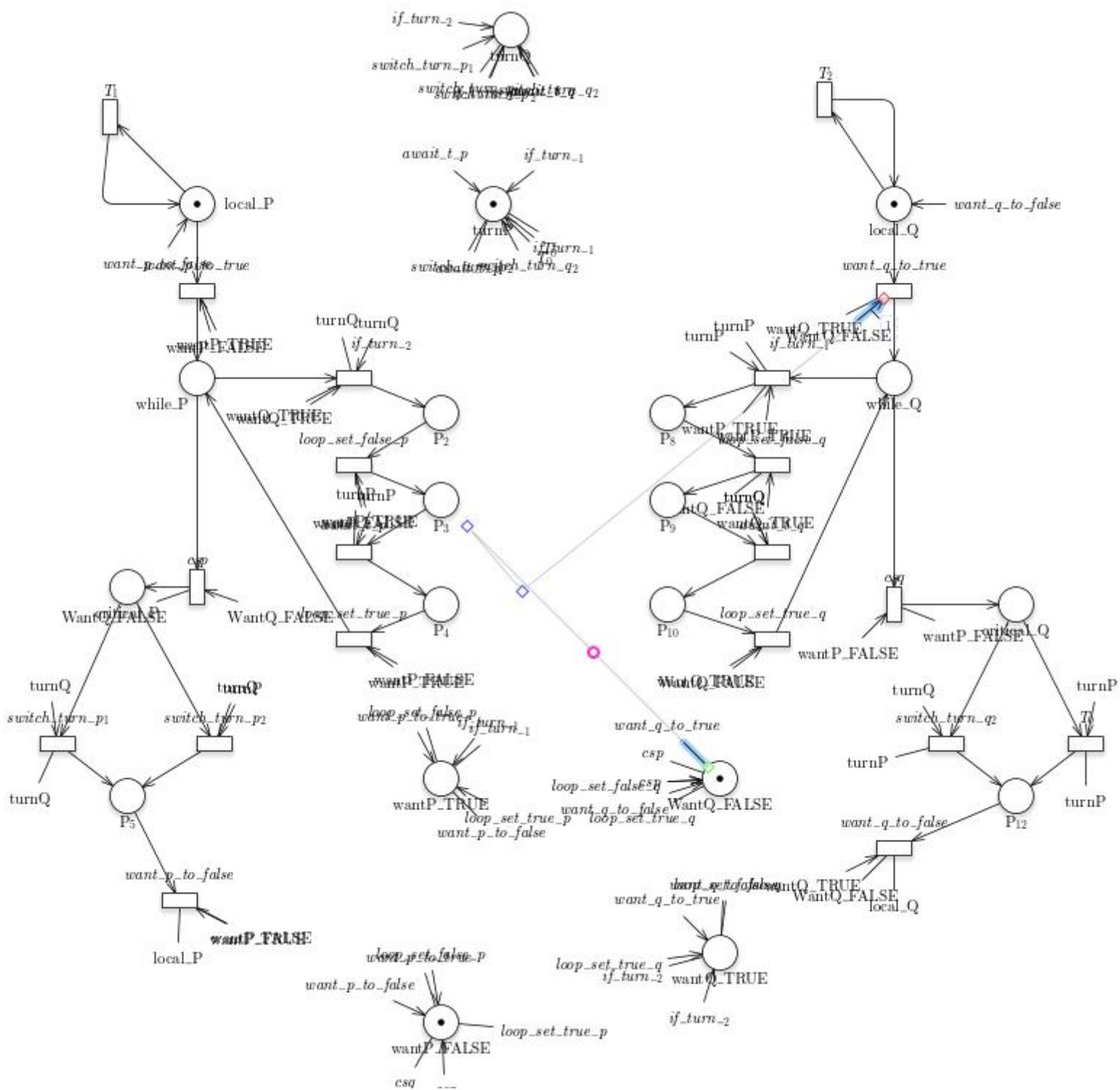
FAIRNESS running

---

## 7.2 GreatSPN

Il codice utilizzato per le proprietà CTL è il seguente:

```
AG(!(#critical_P == 1) || !(#critical_Q == 1))
AG ((#local_P==1 || #local_Q == 1) -> AF (#critical_P == 1 || \#critical_Q == 1))
AG (#local_P==1 -> AF (#critical_P == 1))
AG (#local_Q==1 -> AF (#critical_Q == 1))
```



Ci aspettiamo che come in precedenza ci sia deadlock perché viene data ai processi la possibilità di rimanere su *local*. Forzando i processi a fare del progresso dallo stato *local*, allora la formula CTL

$$AG((l_p \parallel l_q) \rightarrow AF(c_p \vee c_q))$$

risulta rispettata. Ancora meglio, piuttosto che rimuovere una transizione possibile, possiamo restringere la verifica dell'assenza di deadlock alla seguente formula CTL

$$\begin{aligned} &AG(w_p \rightarrow EF(c_p \parallel c_q)) \\ &AG(\#await\_P == 1 \rightarrow EF(\#critical\_Q == 1 \parallel \#critical\_P == 1)) \end{aligned}$$

che risulta rispettata.

C'è possibilità di starvation individuale perché a differenza di NuSMV non possiamo rispettare il requisito di fairness.

### 7.3 Risultati

Nella tabella sono mostrati i risultati ottenuti

	NuSMV	GreatSPN
Mutua Esclusione	true	true
Assenza di deadlock	true	false
No Starvation	true	false

## 8 Equivalenza in algebra dei processi

Precedentemente abbiamo modellato usando l'algebra dei processi l'algoritmo 3.2 e l'algoritmo 3.6. Entrambi hanno le seguenti azioni:

$$S = \{local_p, critical_p, local_q, critical_q\} \cup \{\tau\}$$

### 8.1 Bisimulazione

Si applica l'algoritmo del partizionamento per verificare l'equivalenza tramite bisimulazione. Questo lo stato iniziale

{S0, R0}, {S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S14, S13  
R1, R2, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15,  
R16, R17, R18, R19, R20, R21, R22, R23, R24}

Applico lo split considerando l'azione *critical<sub>p</sub>*

{S0, R0}, {S4, S5, R4, R14, R18, R19, R24}  
{S1, S2, S3, S6, S7, S8, S9, S10, S11, S12, S14, R1, R2, R3, R5, R6,  
R7, R8, R9, R10, R11, R12, R13, R15, R16, R17, R20, R21, R22, R23}

Applico lo split considerando l'azione *critical<sub>q</sub>*

{S0, R0}, {S4, S5, R4, R14, R19, R24}, {R18}, {S11, S13, R9, R11, R16, R22}  
{S1, S2, S3, S6, S7, S8, S9, S10, S12, S14, R1, R2, R3, R5, R6,  
R7, R8, R10, R12, R13, R15, R17, R20, R21, R23}

Notiamo che R18 compare come unico elemento di un insieme. Benché l'algoritmo non sia terminato, questo ci basta per concludere che non c'è bisimulazione fra i due algoritmi. Nell'algoritmo 3.2 non compare nessuno stato da cui è possibile effettuare l'azione *critical<sub>p</sub>* o l'azione *critical<sub>q</sub>*. Questo risultato si poteva dedurre dal fatto che l'algoritmo 3.6 a differenza del 3.2 non rispetta la mutua esclusione e quindi vi sono degli stati non rappresentabili nel modello dell'algoritmo 3.2. Si noti che non è strettamente necessario mascherare le azioni *is<sub>p</sub>*, *is<sub>/q</sub>*, *set<sub>p</sub>*, *set<sub>/q</sub>* in quanto presenti in entrambi i modelli. Nel caso di bisimulazione fra due modelli, l'algoritmo raggiunge la terminazione quando non è più possibile partizionare gli insieme in base alle azioni comuni e abbiamo come risultato che in ogni set compare uno stato di un modello insieme ad uno o più stati dell'altro.

## 8.2 Trace equivalence

La trace equivalence è più debole della bisimulation equivalence:

$$p \sim q \rightarrow p \simeq_T q$$

ma non viceversa

$$p \sim q \not\Rightarrow p \simeq_T q$$

Dato che l'algoritmo 3.6 non rispetta la mutua esclusione, possiamo affermare che una sequenza  $s$  dove  $critical_p$  e  $critical_q$  appaiono in successione, non è una sequenza valida per il modello dell'algoritmo 3.2 ma lo è per il modello dell'algoritmo 3.6.

$$s = \dots critical_p critical_q \dots$$

$$s = \dots critical_q critical_p \dots$$

Questo ci permette di dire che non c'è tracce di equivalenze fra i due modelli.

## 9 Riduzione

Le reti dell'algoritmo 3.6 e 3.8 differiscono per la successione delle istruzioni di *setTrue* e *await* e permettono simmetricamente le stesse riduzioni:

rete 3.6	rete 3.8
rimozione self loop	rimozione self loop
fusione posti P1-P2	fusione posti local_P, setTrue_P
fusione posti P4-P5	fusione posti critical_P, setFalse_P
fusione posti Q1-Q2	fusione posti local_Q, setTrue_Q
fusione posti Q4-Q5	fusione posti critical_Q, setFalse_Q

Riportiamo le immagini delle due reti ridotte. Le transizioni relative a *setTrue* e *await* non sono riducibili in quanto hanno archi uscenti. Possiamo affermare che le due reti non sono equivalenti, come era deducibile dal fatto che rispettano diverse proprietà.

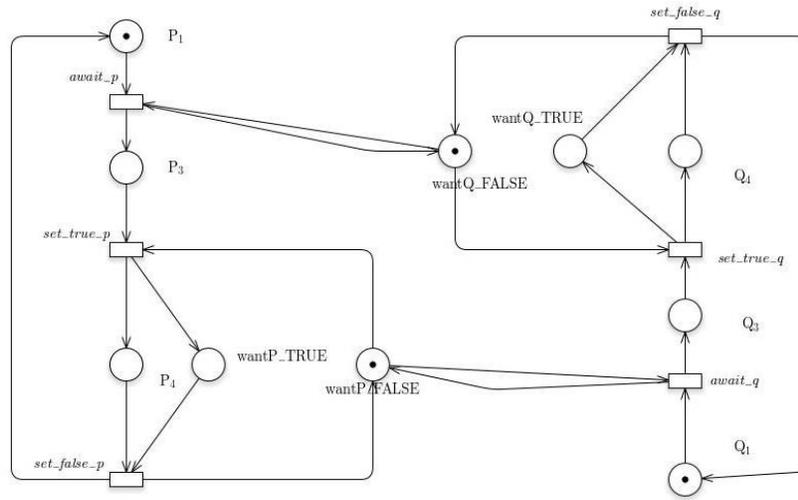


Figure 3: Riduzione rete 3.6

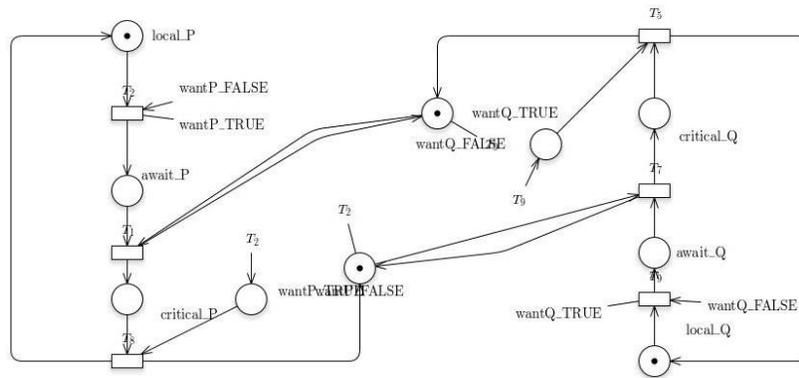


Figure 4: Riduzione rete 3.8

# Esercizio Timed Automata

Francesco Mecca

May 22, 2020

## 1 Modello A

Sono stati usati tre template per rappresentare un sender, un link wd un receiver

```
sender = Sender();
receiver = Receiver();
link = Link();
system sender, receiver, link;
```

### 1.1 Sender

Lo stato iniziale dell'automata è *wait* che simula l'attesa dell'arrivo di un pacchetto che viene processato nell'intervallo  $[1;2]$  del clock *sc*. Lo stato successivo è *send\_pkg* che invia il pacchetto sul canale di sincronizzazione urgente *ML*. Il sender passa allo stato *wait\_ack* che simula l'attesa di un pacchetto sul canale di sincronizzazione urgente *LM* e resetta il clock *sc*. Viene simulata la verifica dell'ack in un intervallo di tempo  $[2;4]$  e successivamente l'automa torna allo stato iniziale resettando il clock *sc*.

### 1.2 Receiver

Il receiver è simmetrico al sender. Dallo stato iniziale di *wait* viene simulato l'arrivo di un pacchetto sul canale di sincronizzazione urgente *LR* che permette al receiver di resettare il clock *rc* e avanzare nello stato *recv\_pkg*,

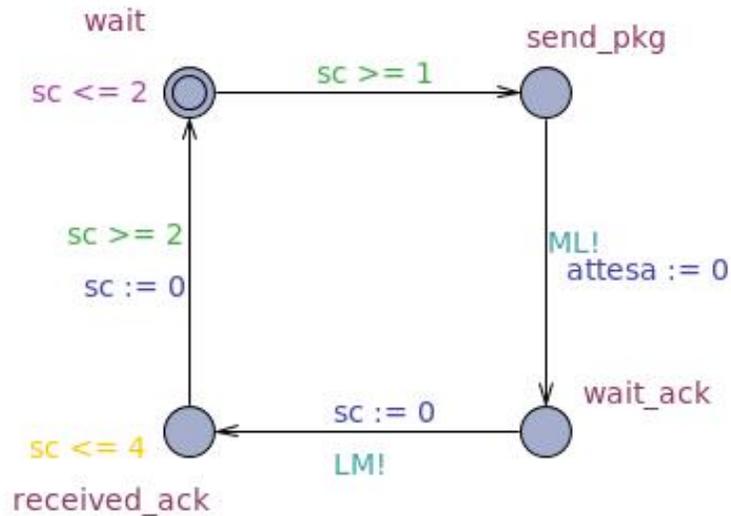


Figure 1: Sender protocollo A

simulando la preparazione del pacchetto in un intervallo di tempo  $[2;4]$ . La transizione dallo stato *done\_pkg* a *send\_ack* simula la generazione di un pacchetto di ack in un intervallo di tempo  $[2;4]$  ed tornare allo stato iniziale dopo aver inviato il pacchetto nel canale di sincronizzazione urgente *RL*.

### 1.3 Link

Il link è modellato come un canale perfetto senza perdite. Lo stato interno *idle* simula l'attesa di un pacchetto da parte del receiver o del sender. Lo stato *received\_pkg* simula l'arrivo di un pacchetto da parte del sender sul canale *ML* che viene processato e spedito al receiver sul canale *LR*. Simmetricamente le stesse operazioni di attesa, processing e invio avvengono con i pacchetti di ack di cui si simula l'arrivo nello stato *received\_ack* dal canale di trasmissione *RL* e l'invio verso il sender dallo stato *resend\_ack* attraverso il canale di trasmissione *LM*. Per il processing dei pacchetti l'intervallo di

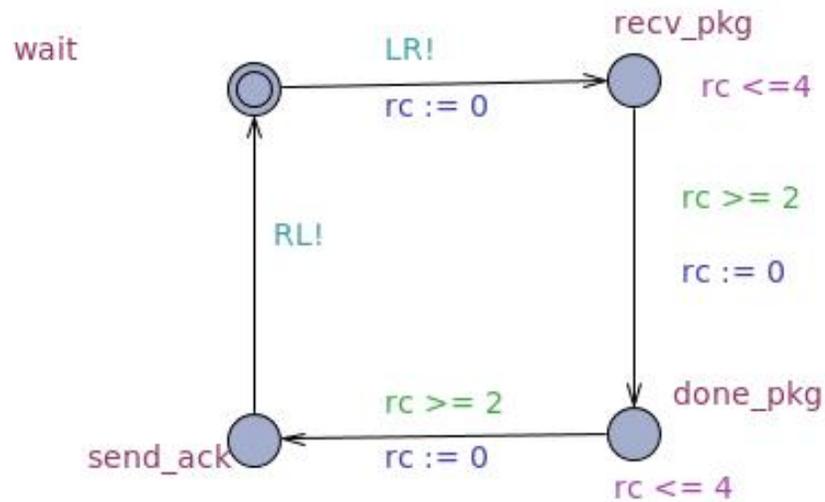


Figure 2: Receiver protocollo A

tempo è sempre  $[2;4]$ .

### 1.4 Proprietà

Sono state verificate tre proprietà TCTL sul modello:

- Assenza di deadlock

$A[]$  (not deadlock)

che viene rispettata e indica che la traccia del sistema è infinita

- Il sender riceverà sempre un ack

$AF$  sender.received\_ack

$A<>$  sender.received\_ack

la proprietà risulta vera in quanto il link non ha perdite.

- Il receiver riceverà sempre un pacchetto

AF receiver.recv\_pkg  
A<> receiver.recv\_pkg

è una proprietà rispettata in quando non ci sono constraint sul tempo di attesa del pacchetto.

### 1.5 Tempo attesa

Il tempo di attesa viene calcolato attraverso il simulatore di Uppaal. È stato inserito un clock che viene resettato ogni qualvolta viene inviato un pacchetto e una volta processato l'ack. L'intervallo di tempo che intercorre dall'invio del messaggio alla recezione del suo ack è [11, 22).

## 2 Modello B

È stato riutilizzato il sistema A e modificato il link per simulare la possibilità di perdita o corruzione di un pacchetto.

## 2.1 Link

Come si denota dall'immagine il link è simile a quello del modello A eccezion fatta per la possibilità di procedere non deterministicamente dagli spazi *received\_pkg* e *received\_ack* verso *loss*. Mostriamo un trace in cui il link perde

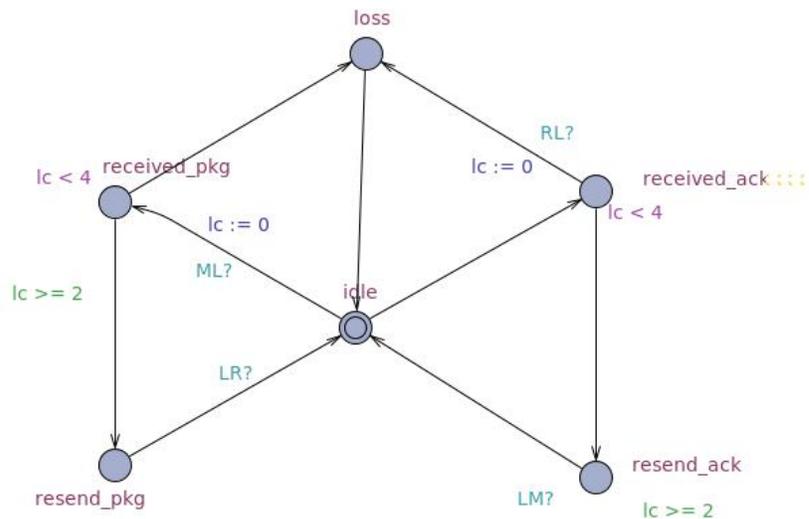


Figure 4: Link protocollo B

un pacchetto di ack inviato dal sender e il sistema va in deadlock. Dalla figura possiamo notare la transizione che simula la perdita, ovvero l'arco *received\_ack*  $\rightarrow$  *loss*.

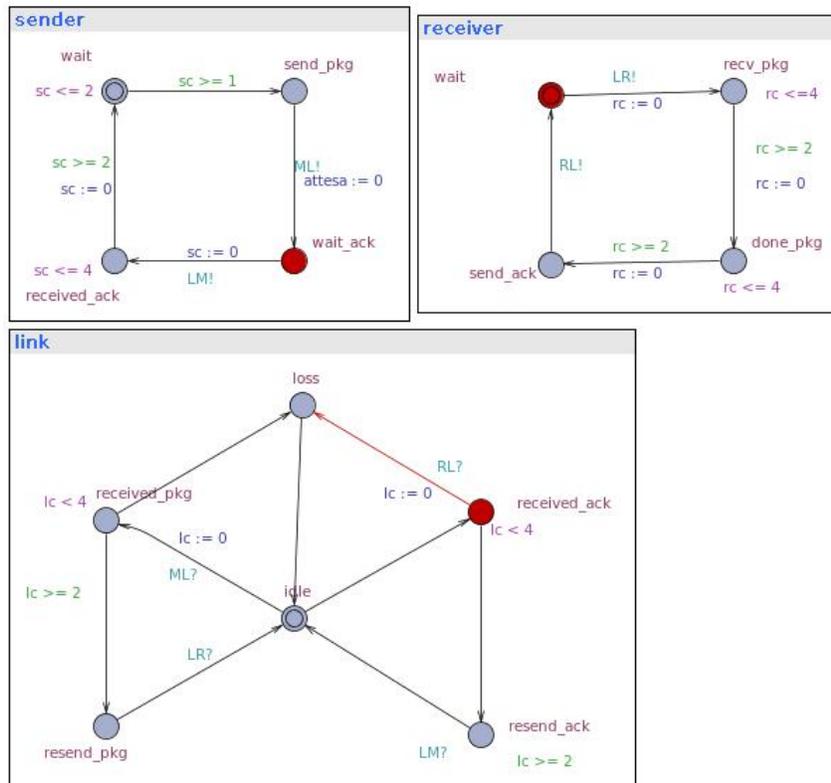
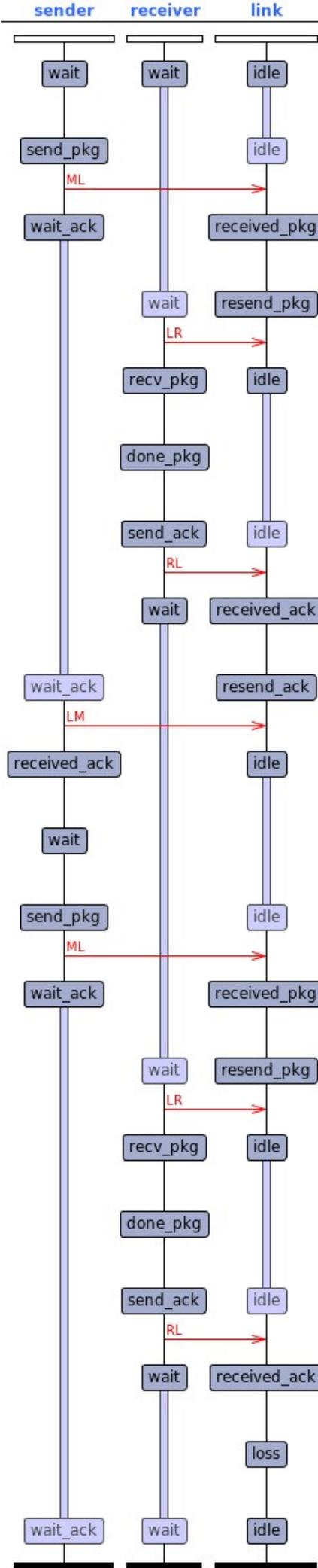


Figure 5: Trace con deadlock



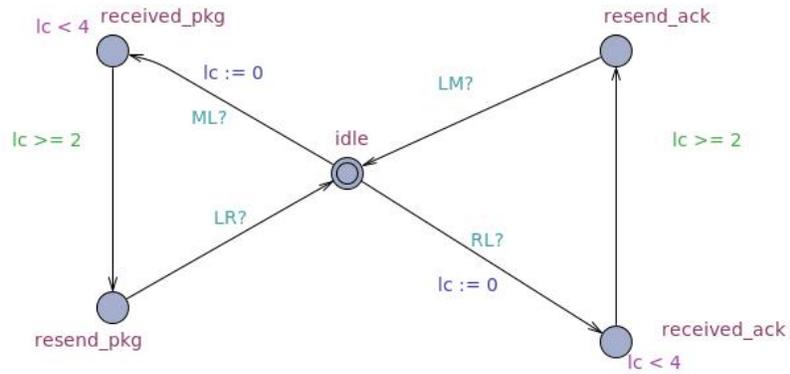


Figure 3: Link protocollo A

## 2.2 Proprietà

Sono state verificate quattro proprietà TCTL sul modello:

- Assenza di deadlock

$A[]$  (not deadlock)

che non è rispettata

- Il sender compierà sempre del progresso

$\text{sender.wait\_ack} \rightarrow (\text{sender.received\_ack})$

la proprietà non risulta vera in quanto il link ha perdite

- Il receiver riceverà sempre un pacchetto

$AF \text{ receiver.recv\_pkg}$

$A\langle\rangle \text{ receiver.recv\_pkg}$

che come per la proprietà precedente non può risultare vera

- Il receiver può ricevere un pacchetto

EF receiver.recv\_pkg  
E<> receiver.recv\_pkg

la proprietà risulta vera perché il link in alcuni path riesce ad inviare il pacchetto (non avvengono perdite sul canale)

### 2.3 Tempo attesa

Non si devono fare modifiche al modello rispetto al precedente per stabilire il tempo che intercorre dall'invio di un messaggio all'arrivo dell'ack. Bisogna però notare che in caso di deadlock l'ack non arriva mai al mittente e quindi effettivamente possiamo calcolare solo il tempo di ricezione minimo, ovvero 11 cicli del clock.

## 3 Modello C

È stato riutilizzato il sistema B e modificati sender e receiver per simulare la possibilità di recovery in seguito alla perdita o corruzione di uno o più pacchetti. Vengono anche mantenute due flag binarie globali, *ack* e *frame* per tenere traccia dei pacchetti inviati e degli ack ricevuti.

### 3.1 Receiver

Di seguito viene mostrato il receiver del modello C. Rispetto al Modello B il receiver utilizza una flag binaria locale *r\_frame* per tenere traccia degli pacchetti già ricevuti e processati.

### 3.2 Sender

Il sender è stato modellato raddoppiando il numero di posti e archi, escludendo per lo stato iniziale, in modo da poter processare nella parte destra i frame di tipo 0 e nella parte sinistra i frame di tipo 1. Inoltre sono stati aggiunti due timer che, nel momento in cui l'automata è fermo su *wait\_ack* allo scadere del timeout, permettono di spostarsi nello spazio *lost* e riprovare l'invio del

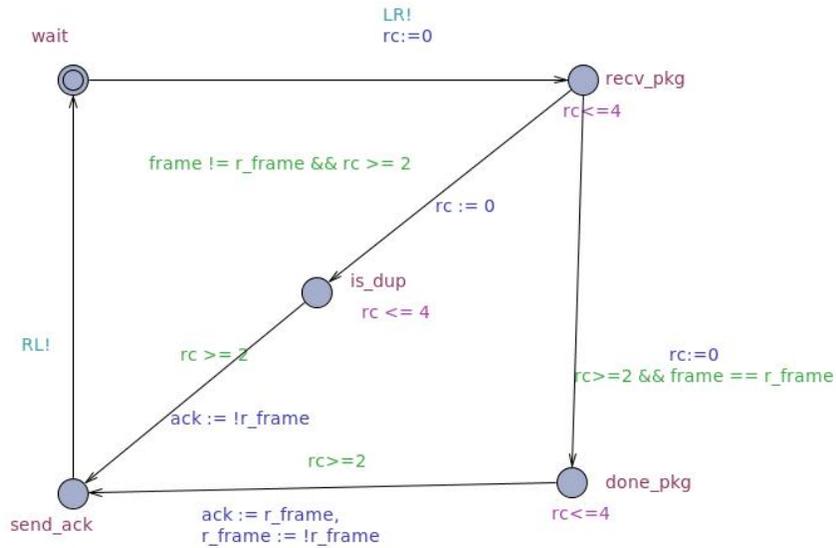


Figure 6: Receiver protocollo C

pacchetto. Si nota subito che i due timer sono indipendenti l'uno dall'altro e vi è la possibilità di modellare i sender senza duplicare il numero di spazi.

La variabile che regola il timeout è un numero intero uguale o maggiore del più grande fra i seguenti valore:

- il massimo tempo necessario all'invio di un pacchetto che il sender non ha processato
- il massimo tempo necessario all'invio di un pacchetto che il sender ha già processato

Se non si rispetta questo constraint il sistema va in deadlock. Nel modello utilizzato lo spazio *is\_dup* e *done\_pkg* del receiver hanno lo stesso invariante e il timeout *TOUT* ha valore minimo 16.

### 3.3 Proprietà

Sono state verificate cinque proprietà TCTL sul modello:

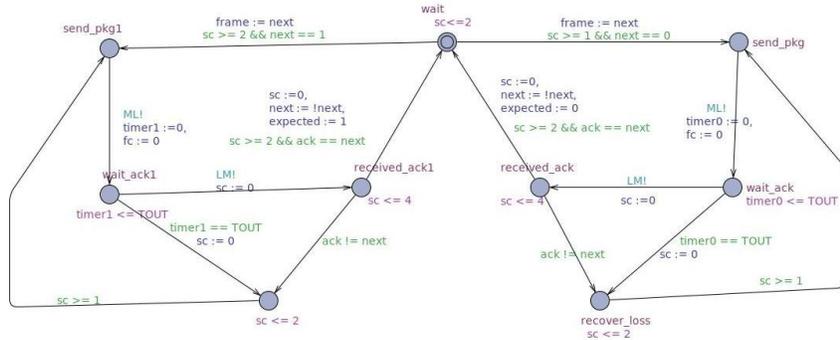


Figure 7: Sender protocollo C (due timer)

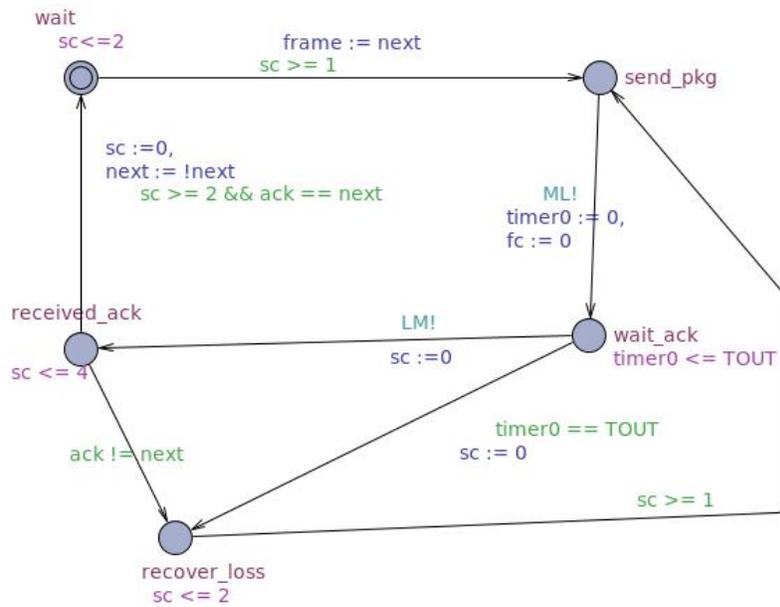


Figure 8: Sender protocollo C (un timer)

- In caso di perdita il sender o il receiver tentano nuovamente di inviare il pacchetto

$\text{link.loss} \rightarrow (\text{sender.send\_pkg} \parallel \text{sender.send\_pkg1} \parallel \text{receiver.send\_ack})$

questa proprietà è rispettata.

- Assenza di deadlock

$A[]$  (not deadlock)

questa proprietà è rispettata.

- Nonostante una perdita nel link, il sender o il receiver riceveranno il pacchetto

$\text{link.loss} \rightarrow (\text{sender.send\_pkg} \parallel \text{receiver.send\_ack})$

la proprietà è rispettata

- Il receiver riceverà sempre un pacchetto

$\text{sender.wait\_ack} \parallel \text{sender.wait\_ack1} \rightarrow (\text{sender.received\_ack} \parallel \text{sender.received\_ack1})$

questa proprietà non risulta vera dato che ci possono essere delle perdite nel canale

- C'è almeno un caso in cui l'ack ricevuto è quello atteso

$E\langle\rangle$  (ack == expected)

questa proprietà risulta vera e conferma la correttezza del meccanismo di recovery loss.

### 3.4 Tempo attesa

In questo modello se si considera l'intervallo di tempo che intercorre dal primo tentativo di invio alla ricezione dell'ack, non è possibile calcolare il valore massimo dell'intervallo. Questo perché non è possibile prevedere quanti tentativi saranno necessari per un corretto invio. Se invece si decide di resettare il clock ad ogni tentativo di invio, ovvero di calcolare l'intervallo di tempo che intercorre fra un pacchetto inviato in una situazione in cui il link non avrà perdite fino all'invio del relativo ack, allora si può considerare il modello equivalente al modello A, e quindi l'intervallo è il medesimo.