

Esercizio finale

Lorenzo Dentis, lorenzo.dentis@edu.unito.it

14 dicembre 2022

1 Disclaimer

Il progetto in se è stato ben valutato dalla professoressa, qualche piccola imprecisione sulla parte delle definizioni. C'è un solo grande enorme errore che mi è costato due punti all'esame. Non ho intenzione di correggerlo perchè ormai ho dato l' esame, però mi sembra doveroso dichiararlo.

La proprietà di assenza di deadlock che la prof vuole venga verificata NON è la proprietà che ho verificato io. Non so come mi sia sfuggito ma io ho proprio cercato di dimostrare un'altra proprietà (in effetti più difficile da dimostrare con i tools forniti), cioè : "uno dei due processi è in grado di progredire". La prof non chiede questa proprietà.

C'è poi un commento nell'ultima sezione (3.10)

2 Introduzione

L'esercizio consiste nella verifica di 3 proprietà in diverse varianti di un algoritmo di mutua esclusione presentato sul libro a partire dall'algoritmo 3.2 fino all'algoritmo 3.10 denominato Algoritmo di Dekker. Le 3 proprietà da verificare sono:

- **Assenza di deadlock:** Se qualche processo cerca di accedere alla regione critica eventualmente un processo potrà farlo.
- **Mutua esclusione:** le istruzioni delle sezioni critiche di due o più processi non possono essere eseguite in modo interfogliato.
- **Assenza di starvation individuale:** Se un processo cerca di accedere alla regione critica eventualmente quel processo potrà farlo.

3 Algoritmo 3.2

Algorithm 3.2: First attempt	
integer turn \leftarrow 1	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await turn = 1	q2: await turn = 2
p3: critical section	q3: critical section
p4: turn \leftarrow 2	q4: turn \leftarrow 1

Questo primo algoritmo propone la mutua esclusione tramite una singola variabile *turn* che identifica quale processo tra *p* e *q* può accedere alla regione critica.

3.1 Rete di Petri

In questo particolare caso è stato possibile effettuare una composizione Name-based (Figura 2).

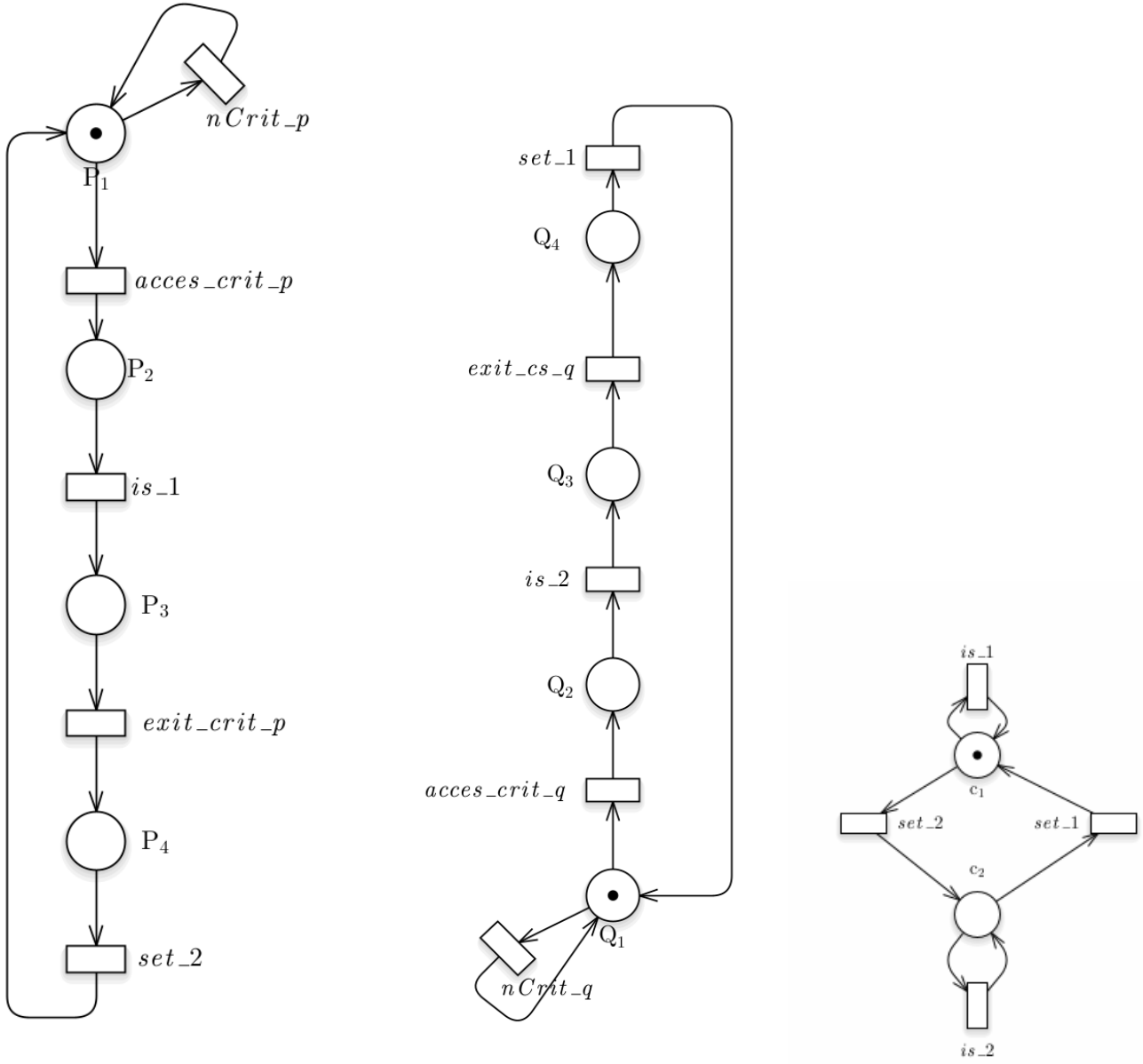
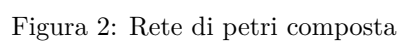


Figura 1: Rete di petri decomposta



3.1.1 RG

Il reachability graph, in figura 3, è composto da 16 stati raggiungibili e non presenta alcuno stato di deadlock.

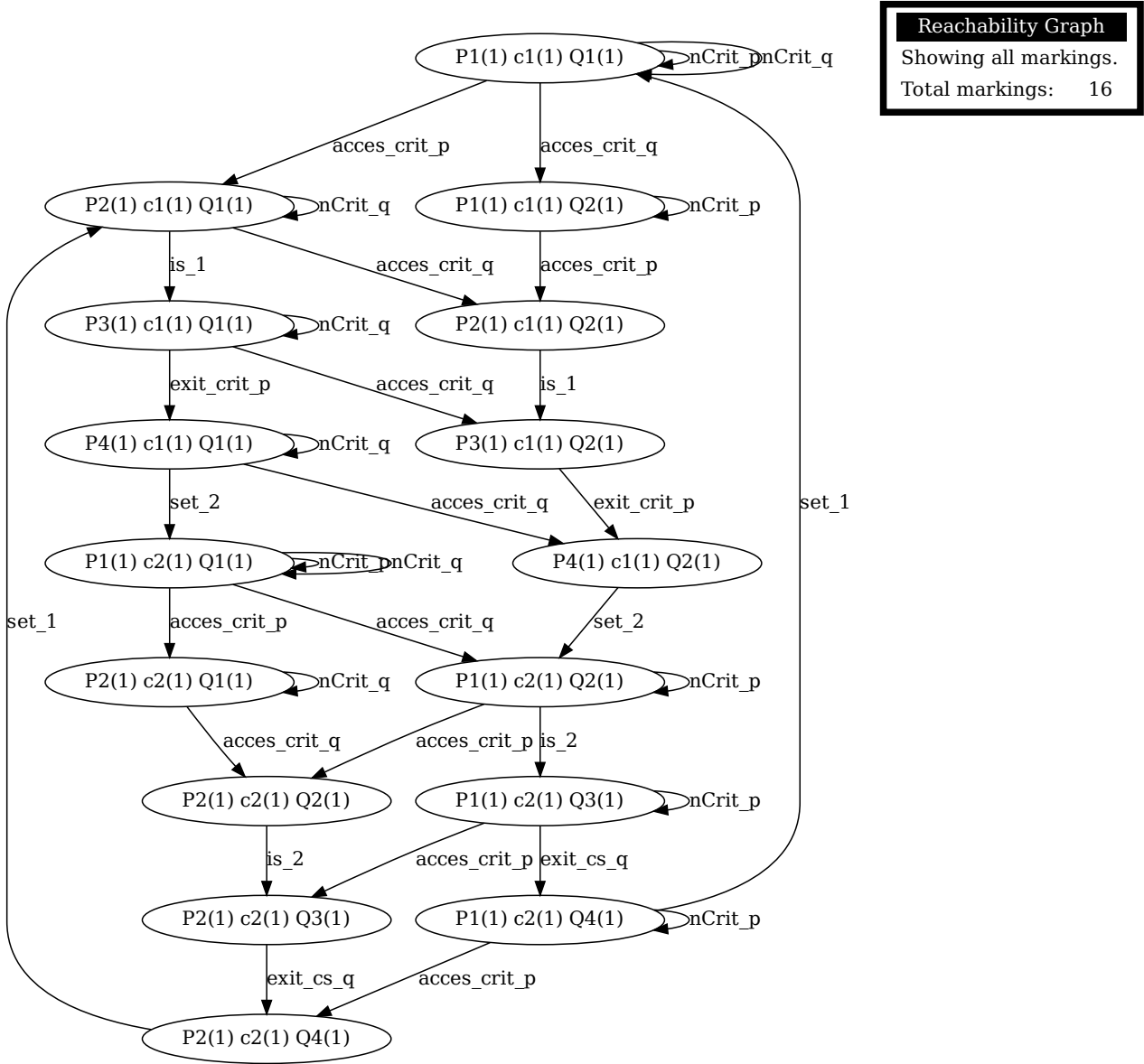
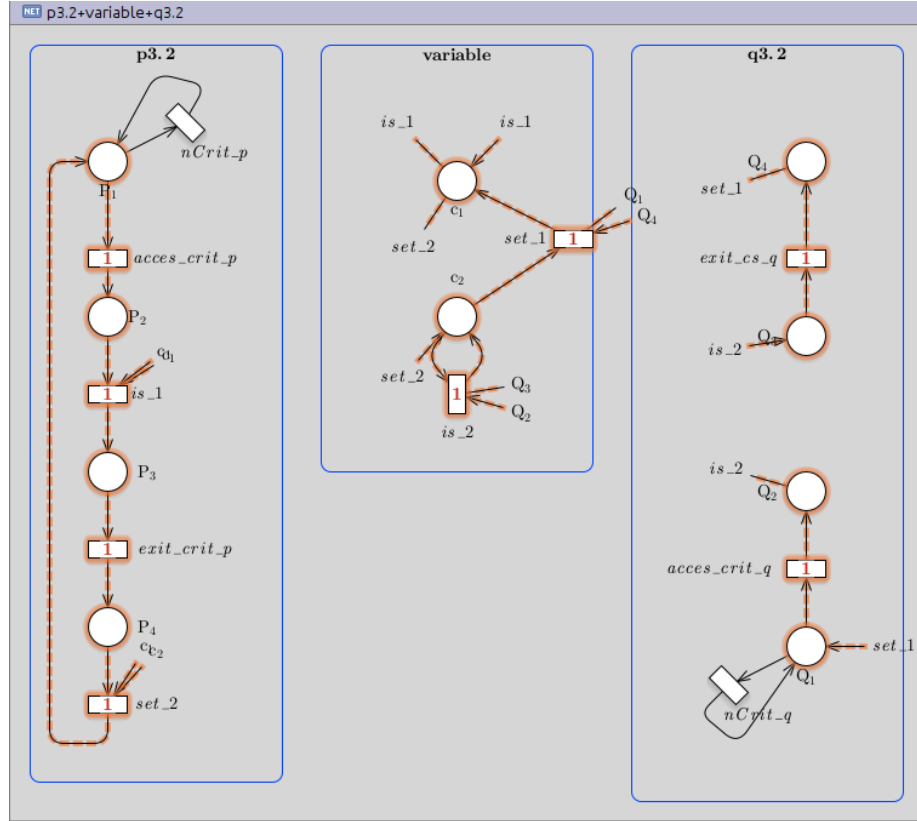


Figura 3: Reachability graph 3.2

3.1.2 Analisi strutturale

Il calcolo dei *semiflow* fornisce 3 *T-semiflow* minimali e 3 *P-semiflow* minimali, i 3 *P-semiflow* permettono di produrre dei P-invarianti e di studiare la boundedness, che in questo caso rivela che tutti i posti sono 1-bound. Invece dallo studio dei *T-semiflow* si può affermare che il sistema possiede la proprietà di *liveness* in quanto è possibile individuare una *ring sequence* attivabile dalla marcatura iniziale che riporta ad una situazione analoga alla situazione iniziale.

2 *T-semiflow* interessanti sono quelli che interessano il posto P_1 ed il posto Q_1 in quanto rappresentano una completa esecuzione della *ncs*.



3.1.3 Model Checking GreatSPN

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg(\#P3==1 \ \&\& \ \#Q3==1)$ **true**.
- assenza di starvation: $AG((\#P2 > 0) \rightarrow (AF i\#P3 > 0))$ ed anche $AG((\#P2 > 0) \rightarrow (AF i\#P3 > 0))$. Entrambe risultano **false**.
Anche inserendo i fairness constrain $\#P1 > 0$ e $\#Q1 > 0$ non è garantita l'assenza di starvation. Il model checker fornisce come controprova il caso in cui P è fermo al posto 2, la variabile $turn$ vale 2 e Q cicla all'infinito in sezione non critica.
L'unico constrain che garantirebbe l'assenza di starvation sarebbe $\#P2 > 0$ e $\#Q2 > 0$ cioè l'imposizione di progresso in regione non critica, in quanto obbligherebbe il processo Q a richiedere la sezione critica e modificare il valore della variabile $turn$.
- deadlock: $AG AF ((\#P1==1) \parallel (\#Q1 == 1))$ **true**. Come ci aspettavamo dalle analisi strutturali e dal DG il sistema non va in deadlock.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(\#P3==1 \ \&\& \ \#Q3==1)$ **true**.
- assenza di starvation: $G F (\#P2==1) \rightarrow G F (\#P3 == 1)$ ed anche $G F (\#Q2==1) \rightarrow G F (\#Q3 == 1)$. Entrambe risultano **false**.
- deadlock: $G F ((\#P1 == 1) \parallel (\#Q1 == 1))$ **true**.

3.2 Algebra dei processi

La codifica del sistema in CCS risulta essere:

$$SYS = (P_1 \parallel Q_1 \parallel T_1) / \{isT_1, setT_1, isT_2, setT_2\}$$

$$P_1 = ncsP.P_1 + ncsP.P_2$$

$$P_2 = isT_1.P_3$$

$$P_3 = csP.P_4$$

$$P_4 = setT_2.P_4$$

$$Q_1 = ncsQ.Q_1 + ncsQ.Q_2$$

$$Q_2 = isT_2.Q_3$$

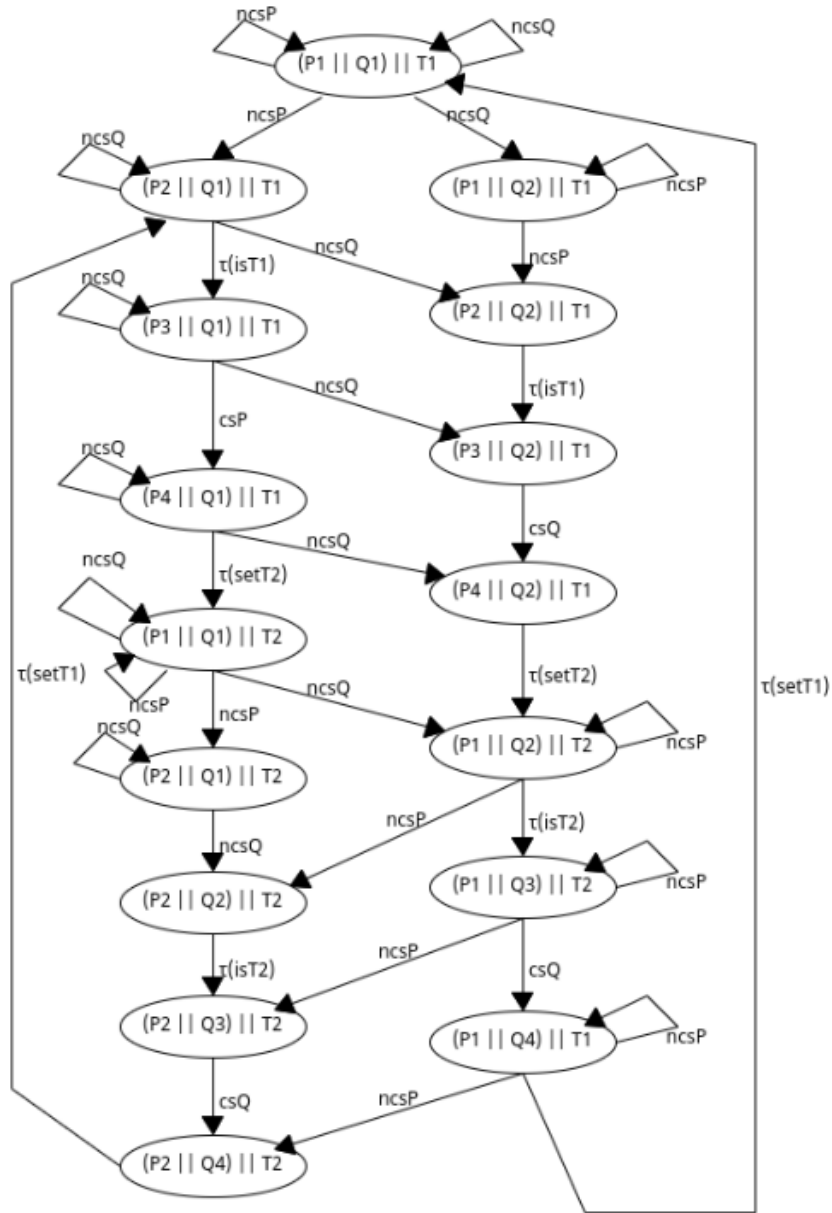
$$Q_3 = csQ.Q_4$$

$$Q_4 = setT_1.Q_4$$

$$T_1 = \overline{isT_1}.T_1 + \overline{setT_2}.T_2$$

$$T_1 = \overline{isT_2}.T_2 + \overline{setT_1}.T_1$$

Da cui segue il seguente derivation graph (in cui la riduzione è omessa per chiarezza).



Come si può notare il *DG* è composto da 16 stati, esattamente il numero di stati raggiungibili del Reachability Graph, questo è un risultato aspettato in linea con quanto analizzato nelle corrispondenti parti dell'esercizio produttore-consumatore.

3.3 NuSMV

L'implementazione del sistema tramite il linguaggio di NuSMV sfrutta la similarità che c'è tra il processo P ed il processo Q. Infatti i due programmi svolgono

le stesse identiche operazioni, ma su variabili differenti, quindi basta dichiarare i processi andando ad inserire correttamente i parametri. Ad esempio il processo *p* avrà 4 stati: lo stato *s1* da cui potrà proseguire richiedendo la sezione critica, oppure rimanendo in *s1*, lo stato *s3* che corrisponde alla sezione critica e lo stato *s4* che corrisponde all'uscita dalla sezione critica ed la ripetizione del programma completo.

Più interessante è lo stato *s4*, dove si nota il riuso del codice. Per poter accedere alla sezione critica il turno deve essere 1, quindi il processo *P* va a confrontare il valore della variabile *turn* con lo il valore della variabile *var_wait* che in questo caso è 1, se corrispondono il programma prosegue in *s3*. Operazione simile viene effettuata quando *P* va ad impostare il valore della variabile *turn* in uscita dalla sezione critica.

```

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

MODULE proc(turn, var_wait, var_set)
VAR
    state : {s1, s2, s3, s4};
ASSIGN
    init(state) := s1;
    next(state) :=
        case
            state = s1 : {s2, s1};
            state = s2 & (turn = var_wait) : s3;
            state = s3 : s4;
            state = s4 : s1;
            TRUE : state;
        esac;
    next(turn) :=
        case
            state = s4 : var_set;
            TRUE : turn;
        esac;

```

Il comando `print_reachable_states` mostra 16 stati raggiungibili di 32 possibili, in linea con la dimensione del Derivation Graph e del Reachability Graph. Tra tutti gli stati raggiungibili non è presente alcuno stato di Deadlock.

3.3.1 Model Checking NuSMV

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg((p.state = s3) \wedge (q.state = s3))$ **true**.
- assenza di starvation: $AG ((p.state = s2) \rightarrow (AF p.state = s3))$ ed anche $AG ((q.state = s2) \rightarrow (AF q.state = s3))$. Entrambe risultano **false**.

Anche inserendo il fairness constrain *FAIRNESS running* non è garantita l'assenza di starvation. Il model checker fornisce come controprova il caso in cui p è fermo allo stato s2, la variabile turn vale 2 e q cicla all'infinito in sezione non critica. Viene data la possibilità al processo p di eseguire il suo codice, ma essendo $turn = 2$ questi non può entrare in regione critica.

- deadlock: $AG AF ((p.state = s1) \vee (q.state = s1))$ **true**. Come ci aspettavamo dalle analisi strutturali e dal DG il sistema non va in deadlock.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(p.state = s3 \wedge q.state = s3)$ **true**.
- assenza di starvation: $G (p.state = s2 \rightarrow F p.state = s3)$ ed anche $G (q.state = s2 \rightarrow F q.state = s3)$. Entrambe risultano **false**.

Anche inserendo il fairness constrain *FAIRNESS running* non è garantita l'assenza di starvation. Il model checker fornisce come controprova un caso più lungo del necessario in cui comunque alla fine ci si riconduce alla situazione in cui p è fermo allo stato s2, la variabile turn vale 2 e q cicla all'infinito in sezione non critica.

- deadlock: $G F((p.running) \vee (q.running))$ **false**. In questo caso sembra verificarsi deadlock, perchè esiste una esecuzione in cui solo il processo Main viene eseguito.

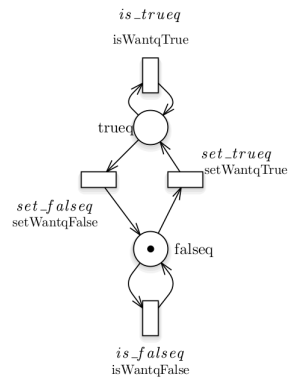
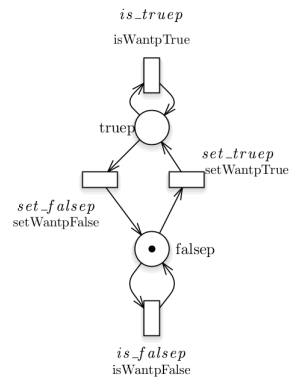
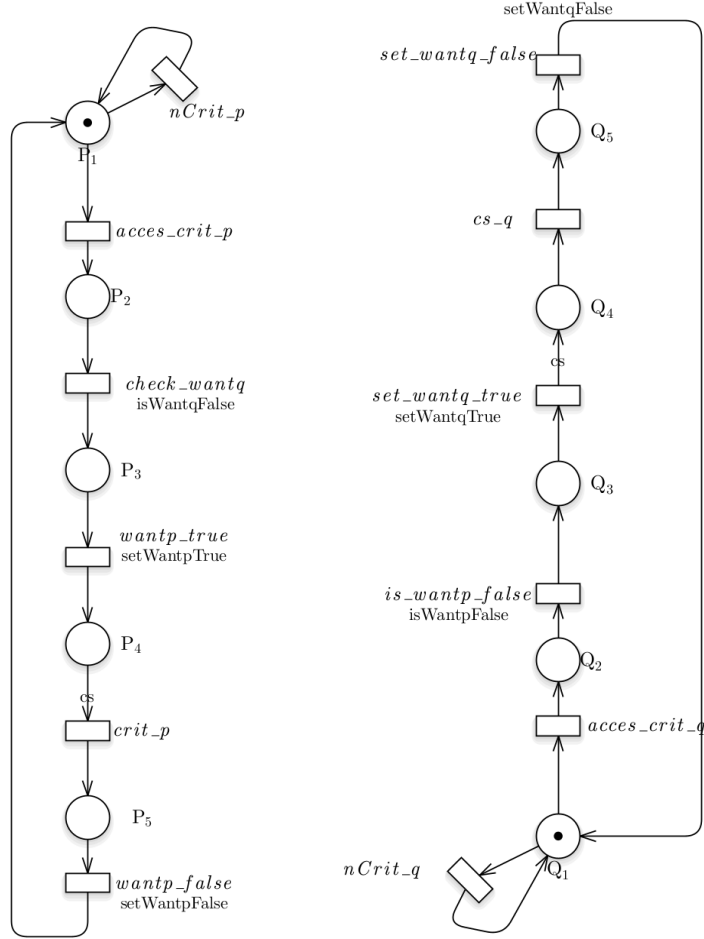
Quindi il problema non è tanto la presenza di deadlock quanto l'incapacità di scrivere la formula in modo da tenere conto anche dell'esecuzione del processo Main. Non riuscendo a riferirmi al modulo main dall'interno del main stesso ho "risolto" imponendo *FAIRNESS running*.

4 Algoritmo 3.6

Algorithm 3.6: Second attempt	
boolean wantp \leftarrow false, wantq \leftarrow false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await wantq = false	q2: await wantp = false
p3: wantp \leftarrow true	q3: wantq \leftarrow true
p4: critical section	q4: critical section
p5: wantp \leftarrow false	q5: wantq \leftarrow false

Questo algoritmo propone la mutua esclusione tramite due variabili *wantp* e *wantq* che indicano se un processo vuole entrare in sezione critica. Un processo può entrare in sezione critica solamente se l'altro non vuole. quando il processo entra in sezione critica imposta la variabile relativa a lui a true.

4.1 Rete di Petri



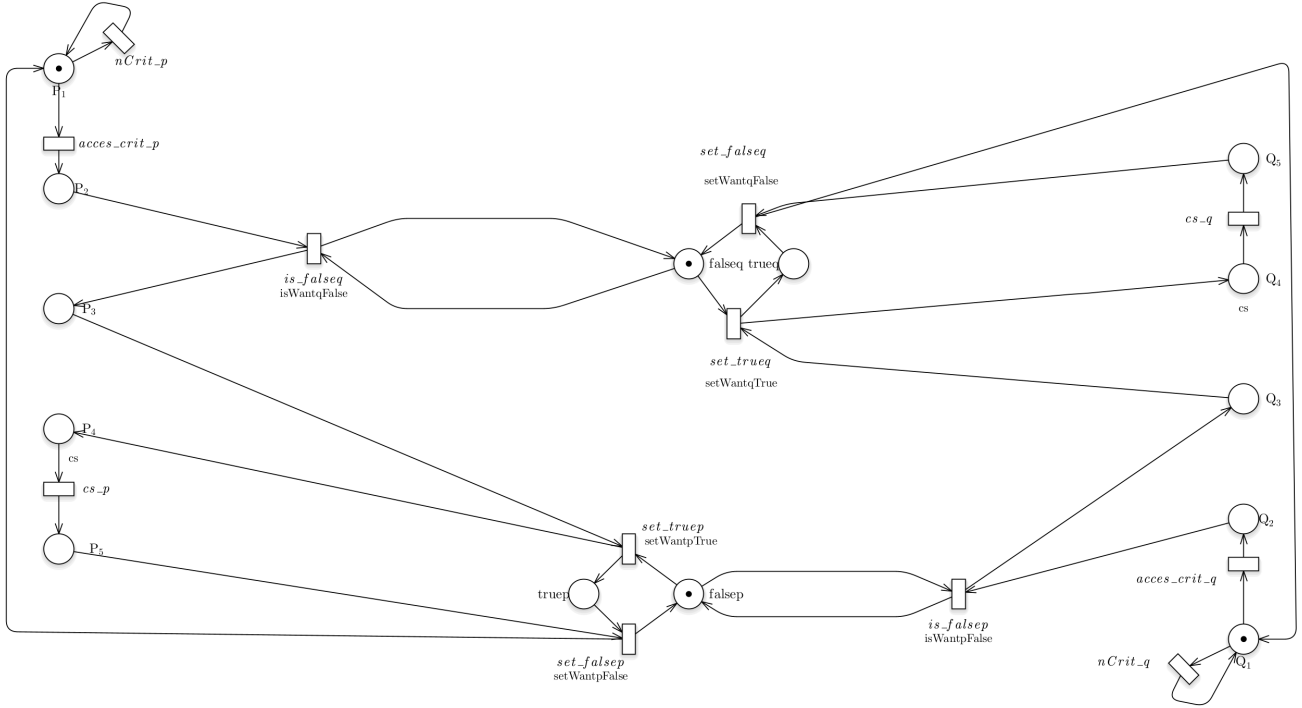


Figura 4: Rete di petri composta

4.1.1 RG

Il reachability graph, in figura 5, è composto da 25 stati raggiungibili e non presenta alcun deadlock.

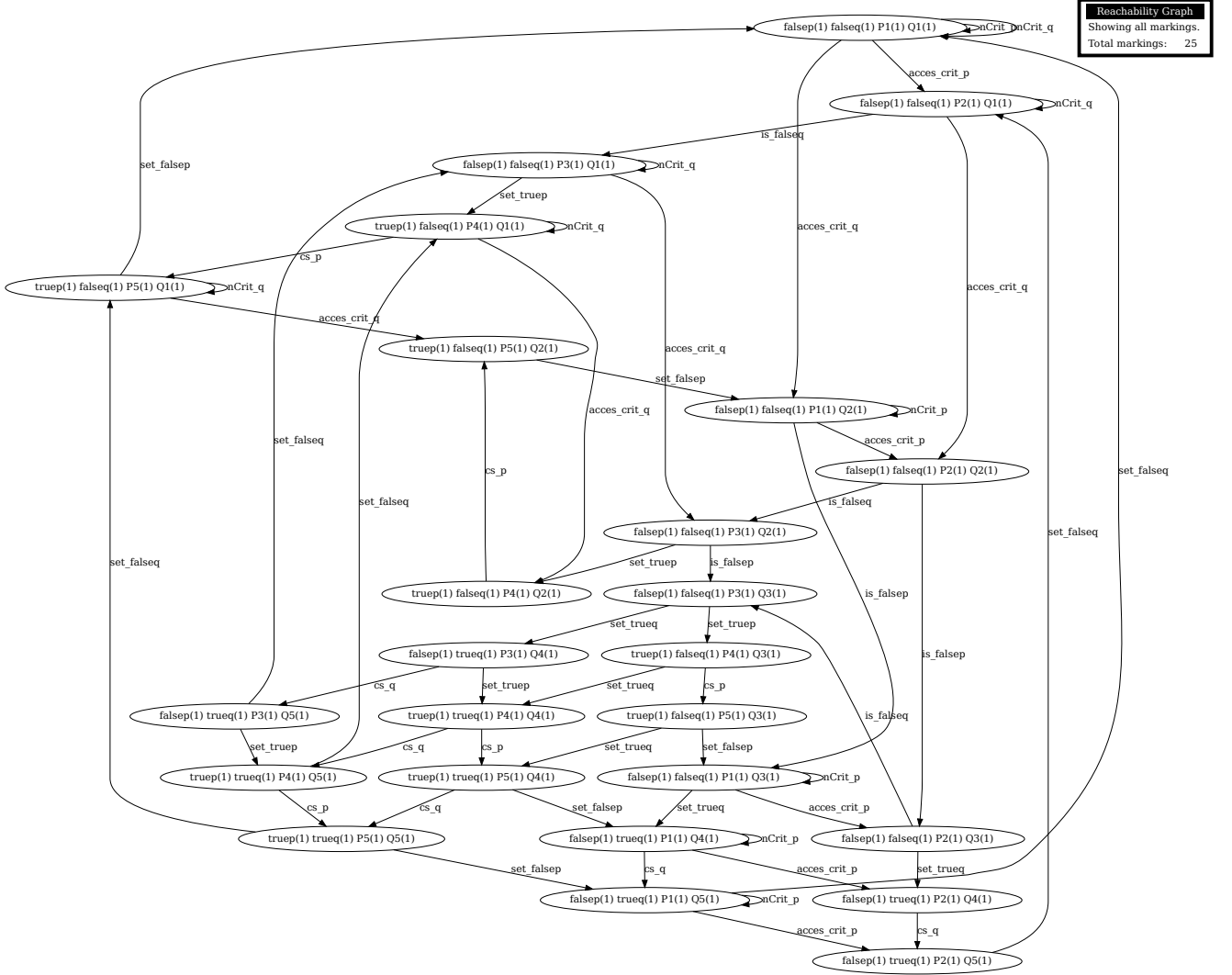


Figura 5: Reachability graph 3.6

4.1.2 Analisi strutturale

Il calcolo dei *semiflow* fornisce 4 *T-semiflow* minimali e 8 *P-semiflow* minimali, gli 8 *P-semiflow* permettono di produrre dei P-invarianti e di studiare la boundedness, che in questo caso rivela che tutti i posti sono 1-bound.

I *T-semiflow* garantiscono che le due variabili **falseq** e **faslep** non assumono mai contemporaneamente il valore di vero e di falso e che ogni processo è sempre in esattamente un solo valore di program counter.

4.1.3 Model Checking GreatSPN

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg(\#P_4 == 1 \ \&\& \ \#Q_4 == 1)$ **false**.
Il model checker fornisce come controesempio l'esecuzione $\{P_1Q_1, P_1Q_2, P_2Q_2, P_2Q_3, P_3Q_3, P_3Q_4, P_4Q_4\}$.
- assenza di starvation: $AG((\#P_2 > 0) \rightarrow (AF \#P_4 > 0))$ ed anche $AG((\#P_2 > 0) \rightarrow (AF \#P_4 > 0))$. Entrambe risultano **false**.
In questo caso risulta più complicato inserire dei *fairness constraint* sensati. Infatti l'assenza di starvation dipende moltissimo dallo scheduling utilizzato. Inserendo il fairness constrain più "elementare", $\#P_1 > 0$ e $\#Q_1 > 0$ cioè "viene quantomeno fornito tempo di CPU al processo p", non è garantita l'assenza di starvation. Il model checker fornisce come controprova il caso in cui *P* è fermo al posto 2, e viene solo eseguito *Q* che cicla all'infinito in sezione non critica.
Un altro constrain che garantirebbe l'assenza di starvation sarebbe $\#P_2 > 0$ e $\#Q_2 > 0$ cioè l'imposizione di progresso in regione non critica.
- deadlock: $AG AF ((\#P_1 == 1) \parallel (\#Q_1 == 1))$ **true**. Come ci aspettavamo dalle analisi strutturali e dal DG il sistema non va in deadlock.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(\#P_4 == 1 \ \&\& \ \#Q_4 == 1)$ **false**.
- assenza di starvation: $G F (\#P_2 == 1) \rightarrow G F (\#P_4 == 1)$ ed anche $G F (\#Q_2 == 1) \rightarrow G F (\#Q_4 == 1)$. Entrambe risultano **false**.
- deadlock: $G F ((\#P_1 == 1) \parallel (\#Q_1 == 1))$ **true**.

4.2 Algebra dei processi

La codifica del sistema in CCS risulta essere:

$$SYS = (P_1 || Q_1 || \overline{noWantP} || \overline{noWantQ}) / \{falseWp, falseWq, trueWp, trueWq, notWp, notWq\}$$

$$P_1 = ncsP.P_1 + ncsP.P_2$$

$$P_2 = notWq.P_3$$

$$P_3 = trueWp.P_4$$

$$P_4 = csP.P_4$$

$$P_5 = falseWp.P_1$$

$$Q_1 = ncsP.P_1 + ncsP.P_2$$

$$Q_2 = notWp.P_3$$

$$Q_3 = trueWq.P_4$$

$$Q_4 = csP.P_4$$

$$Q_5 = falseWq.P_1$$

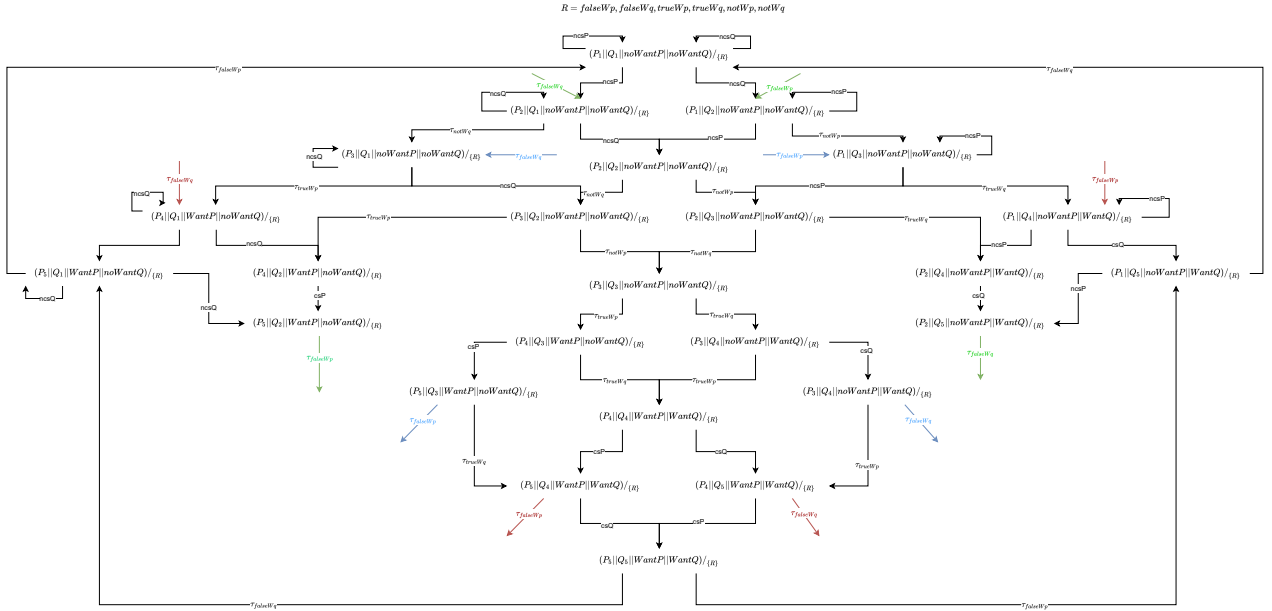
$$\overline{WantP} = \overline{falseWp}.noWantP$$

$$noWantP = \overline{notWp}.noWantP + \overline{trueWp}.WantP$$

$$\overline{WantQ} = \overline{falseWq}.noWantQ$$

$$noWantQ = \overline{notWq}.noWantQ + \overline{trueWq}.WantQ$$

Da cui segue il seguente derivation graph.



Come si può notare il *DG* è composto da 25 stati, esattamente il numero di stati raggiungibili del Reachability Graph, questo è un risultato atteso in

linea con quanto analizzato nelle corrispondenti parti dell'esercizio produttore-consumatore.

4.3 NuSMV

L'implementazione del sistema tramite il linguaggio di NuSMV sfrutta la similarità che c'è tra il processo P ed il processo Q, come nel caso precedente. Questa volta però non è solo una variabile ad essere interessata dallo scambio di parametri, bensì le due variabili *wantp* e *wantq*.

Un paragone che rende facilmente intuibile il comportamento dei due moduli è leggere i due parametri come la variabile che indica "io voglio entrare in sezione critica" e "l'altro vuole entrare in sezione critica", ed è per questo che le due variabili sono state chiamate *want_me* e *want_other*. Per il resto il riuso del codice è identico al caso 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;

MODULE proc(want_me, want_other)
VAR
    state : {s1, s2, s3, s4, s5};
ASSIGN
    init(state) := s1;
    next(state) :=
        case
            state = s1 : {s2, s1};
            state = s2 & (want_other = FALSE) : s3;
            state = s3 : s4;
            state = s4 : s5;
            state = s5 : s1;
            TRUE : state;
        esac;
    next(want_me) :=
        case
            state = s3 : TRUE;
            state = s5 : FALSE;
            TRUE : want_me;
        esac;
FAIRNESS running;

```

Il comando `print_reachable_states` mostra 25 stati raggiungibili di 100 possibili, in linea con la dimensione del Derivation Graph e del Reachability Graph.

Tra tutti gli stati raggiungibili non è presente alcuno stato di Deadlock ma è presente uno stato in cui la mutua esclusione viene violata ($p.state = s4, q.state = s4, wantp = true, wantq = true$).

4.3.1 Model Checking NuSMV

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg((p.state = s4) \ \&\ (q.state = s4))$ **false**.
Il model checker fornisce come controesempio l'esecuzione $\{p.state = s1 \ q.state = s1, p.state = s1 \ q.state = s2, p.state = s2 \ q.state = s2, p.state = s2 \ q.state = s3, p.state = s3 \ q.state = s3, p.state = s3 \ q.state = s4, p.state = s4 \ q.state = s4\}$.
- assenza di starvation: $AG ((p.state = s2) \rightarrow (AF \ p.state = s3))$ ed anche $AG ((q.state = s2) \rightarrow (AF \ q.state = s3))$. Entrambe risultano **false**.
Anche inserendo il fairness constrain *FAIRNESS running* non è garantita l'assenza di starvation. Il model checker fornisce come controprova il caso in cui p è fermo allo stato s2, q esegue un ciclo completo del programma (tornando in s1) e 2 volte viene passata l'esecuzione al processo p. Ma p è bloccato da *wantq* e non può proseguire.
Questo caso specifico fa parte di una più ampia classe di esecuzioni in cui viene concessa l'esecuzione del processo p sempre quando questi è bloccato prima dell'accesso in sezione critica.
- deadlock: $AG \ AF \ ((p.state = s1) \mid (q.state = s1))$ **true**. Come ci aspettavamo dalle analisi strutturali e dal DG il sistema non va in deadlock.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(p.state = s4 \ \&\ q.state = s4)$ **false**.
Come nel caso di CTL il sistema non rispetta la mutua esclusione, il controesempio fornito è lo stesso di CTL.
- assenza di starvation: $G (p.state = s2 \rightarrow F \ p.state = s3)$ ed anche $G (q.state = s2 \rightarrow F \ q.state = s3)$. Entrambe risultano **false**.
Anche questo caso fornisce un controesempio identico a quello fornito in fase di analisi con CTL.
- deadlock: $G \ F((p.running) \mid (q.running))$ **false**. Come in sezione 3 si pone di nuovo il problema di fare riferimento al modulo main dall'interno del modulo stesso. Viene sfruttato di nuovo *FAIRNESS running* per ottenere un risultato coerente con l'analisi.

5 Confronto tramite bisimulazione tra 3.2 e 3.6

Per maggior leggibilità ho modificato i derivation graph dei due algoritmi, identificando con X oppure Y seguiti da un apice lo stato.

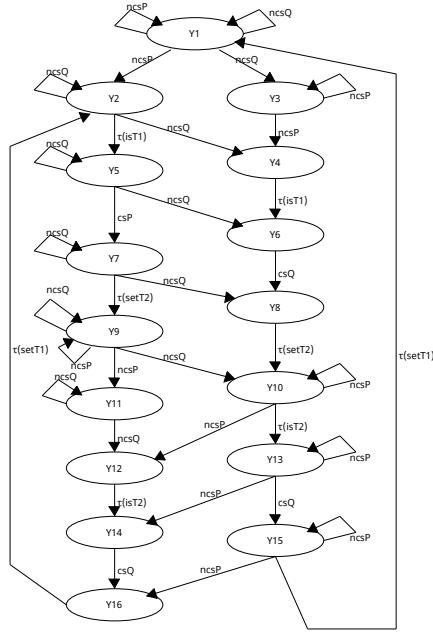


Figura 6: Derivation graph 3.2, Y

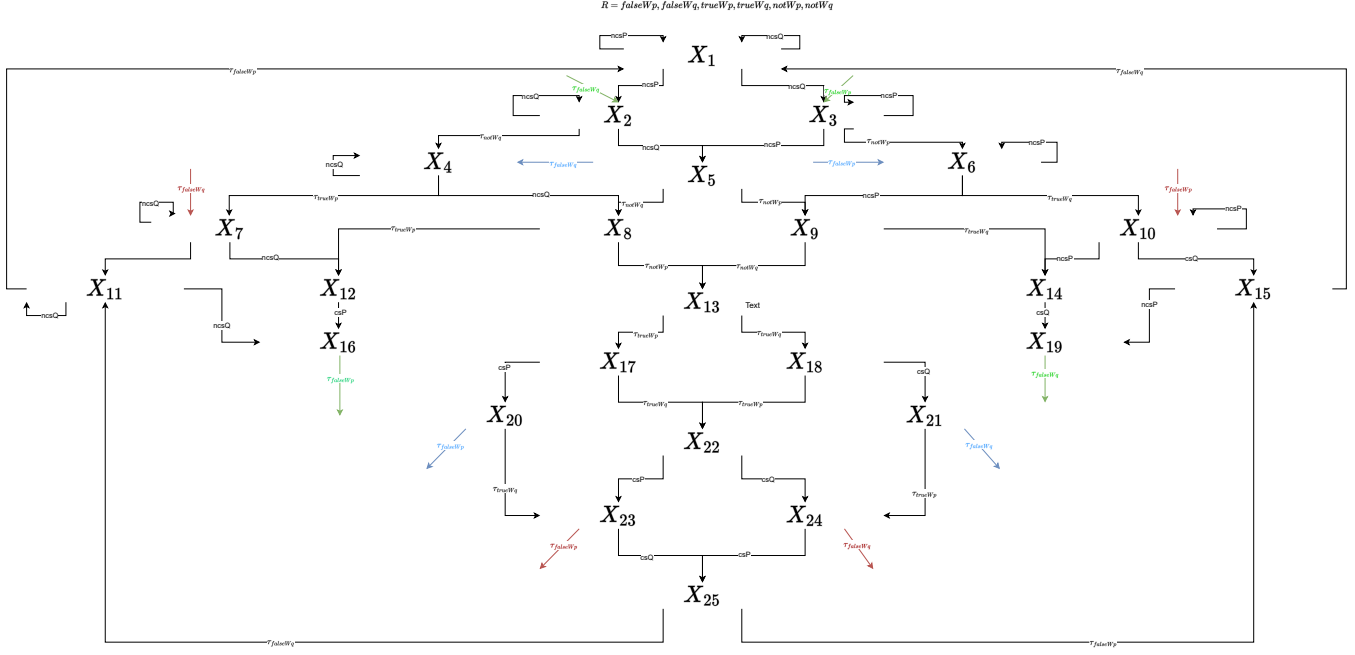


Figura 7: Derivation graph 3.6, X

Su questi due *DG* si può eseguire l'algoritmo di bisimulazione.

$A : \{X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8, Y_9, Y_{10}, Y_{11}, Y_{12}, Y_{13}, Y_{14}, Y_{15}, Y_{16}\}$

Divido l'insieme A sull'azione *ncsQ*

$A : \{X_1, X_2, X_4, X_7, X_{11}, Y_1, Y_2, Y_5, Y_7, Y_9, Y_{11}\}$

$B : \{X_3, X_5, X_6, X_8, X_9, X_{10}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, Y_3, Y_4, Y_6, Y_8, Y_{10}, Y_{12}, Y_{13}, Y_{14}, Y_{15}, Y_{16}\}$

Divido l'insieme B sull'azione *ncsP*

$A : \{X_1, X_2, X_4, X_7, X_{11}, Y_1, Y_2, Y_5, Y_7, Y_9, Y_{11}\}$

$B : \{X_5, X_8, X_9, X_{12}, X_{13}, X_{14}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, Y_4, Y_6, Y_8, Y_{12}, Y_{14}, Y_{16}\}$

$C : \{X_3, X_6, X_{10}, X_{15}, Y_3, Y_{10}, Y_{13}, Y_{15}\}$

Divido l'insieme A sull'azione τ

$$\begin{aligned} A &: \{X_1, X_7, Y_1, Y_5, Y_9, Y_{11}\} \\ D &: \{X_2, X_4, X_{11}, Y_2, Y_7\} \\ B &: \{X_5, X_8, X_9, X_{12}, X_{13}, X_{14}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, \\ &\quad X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, Y_4, Y_6, Y_8, Y_{12}, Y_{14}, Y_{16}\} \\ C &: \{X_3, X_6, X_{10}, X_{15}, Y_3, Y_{10}, Y_{13}, Y_{15}\} \end{aligned}$$

Divido l'insieme A sull'azione $ncsP$

$$\begin{aligned} A &: \{X_1, Y_1, Y_9\} \\ E &: \{X_7, Y_5, Y_{11}\} \\ B &: \{X_5, X_8, X_9, X_{12}, X_{13}, X_{14}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, \\ &\quad X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, Y_4, Y_6, Y_8, Y_{12}, Y_{14}, Y_{16}\} \\ C &: \{X_3, X_6, X_{10}, X_{15}, Y_3, Y_{10}, Y_{13}, Y_{15}\} \\ D &: \{X_2, X_4, X_{11}, Y_2, Y_7\} \end{aligned}$$

Divido l'insieme D sull'azione $ncsP$

$$\begin{aligned} A &: \{X_1, Y_1, Y_9\} \\ B &: \{X_5, X_8, X_9, X_{12}, X_{13}, X_{14}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, \\ &\quad X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, Y_4, Y_6, Y_8, Y_{12}, Y_{14}, Y_{16}\} \\ C &: \{X_3, X_6, X_{10}, X_{15}, Y_3, Y_{10}, Y_{13}, Y_{15}\} \\ D &: \{X_2, X_4\} \\ F &: \{X_{11}, Y_2, Y_7\} \\ E &: \{X_7, Y_5, Y_{11}\} \end{aligned}$$

Divido l'insieme A sull'azione $ncsP$

$$\begin{aligned} A &: \{X_1, Y_9\} \\ G &: \{Y_1\} \\ B &: \{X_5, X_8, X_9, X_{12}, X_{13}, X_{14}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, \\ &\quad X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, Y_4, Y_6, Y_8, Y_{12}, Y_{14}, Y_{16}\} \\ C &: \{X_3, X_6, X_{10}, X_{15}, Y_3, Y_{10}, Y_{13}, Y_{15}\} \\ D &: \{X_2, X_4\} \\ E &: \{X_7, Y_5, Y_{11}\} \\ F &: \{X_{11}, Y_2, Y_7\} \end{aligned}$$

Dato che X_1 e Y_1 si trovano rispettivamente in A e E quindi non sono in relazione di bisimulazione tra loro.

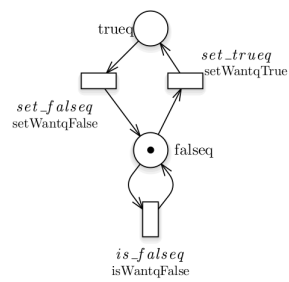
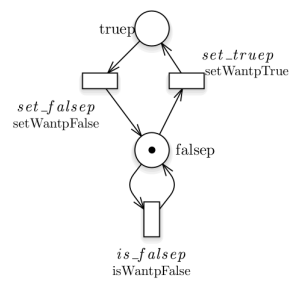
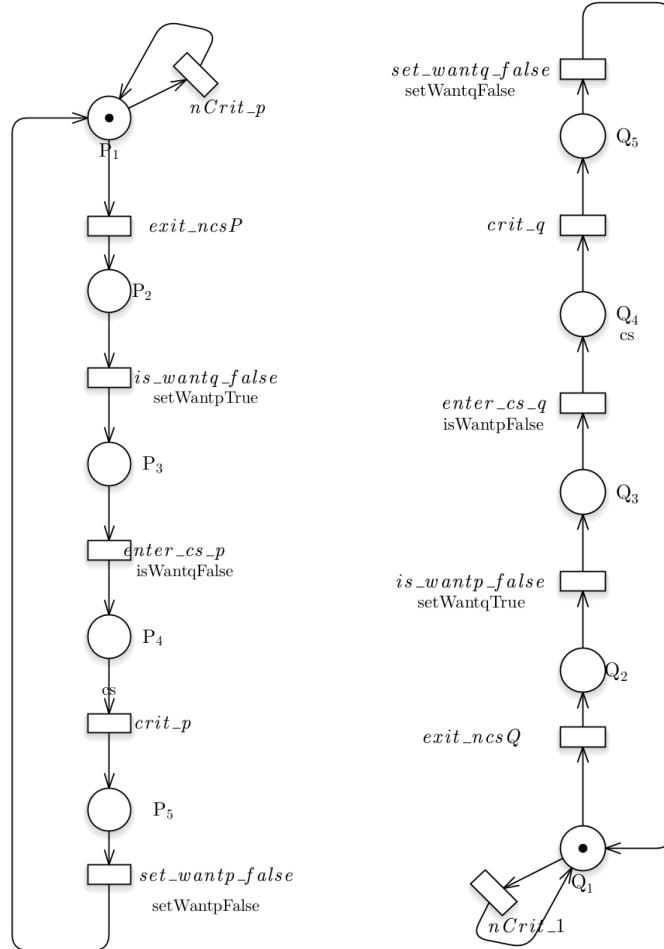
6 Algoritmo 3.8

Algorithm 3.8: Third attempt	
boolean wantp \leftarrow false, wantq \leftarrow false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp \leftarrow true	q2: wantq \leftarrow true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp \leftarrow false	q5: wantq \leftarrow false

Questo algoritmo cerca di risolvere il problema del 3.6 (sezione 4) andando ad invertire l'operazione di setting della variabile che si riferisce al voler entrare in sezione critica e l'operazione di attesa sulla variabile che indica l'intenzione dell'altro processo

6.1 Rete di Petri

In questo caso è stata anche effettuata una modifica alle variabili rimuovendo le transizioni `is_true`, in quanto nell'algoritmo 3.8 non viene mai controllato se una variabile è vera.



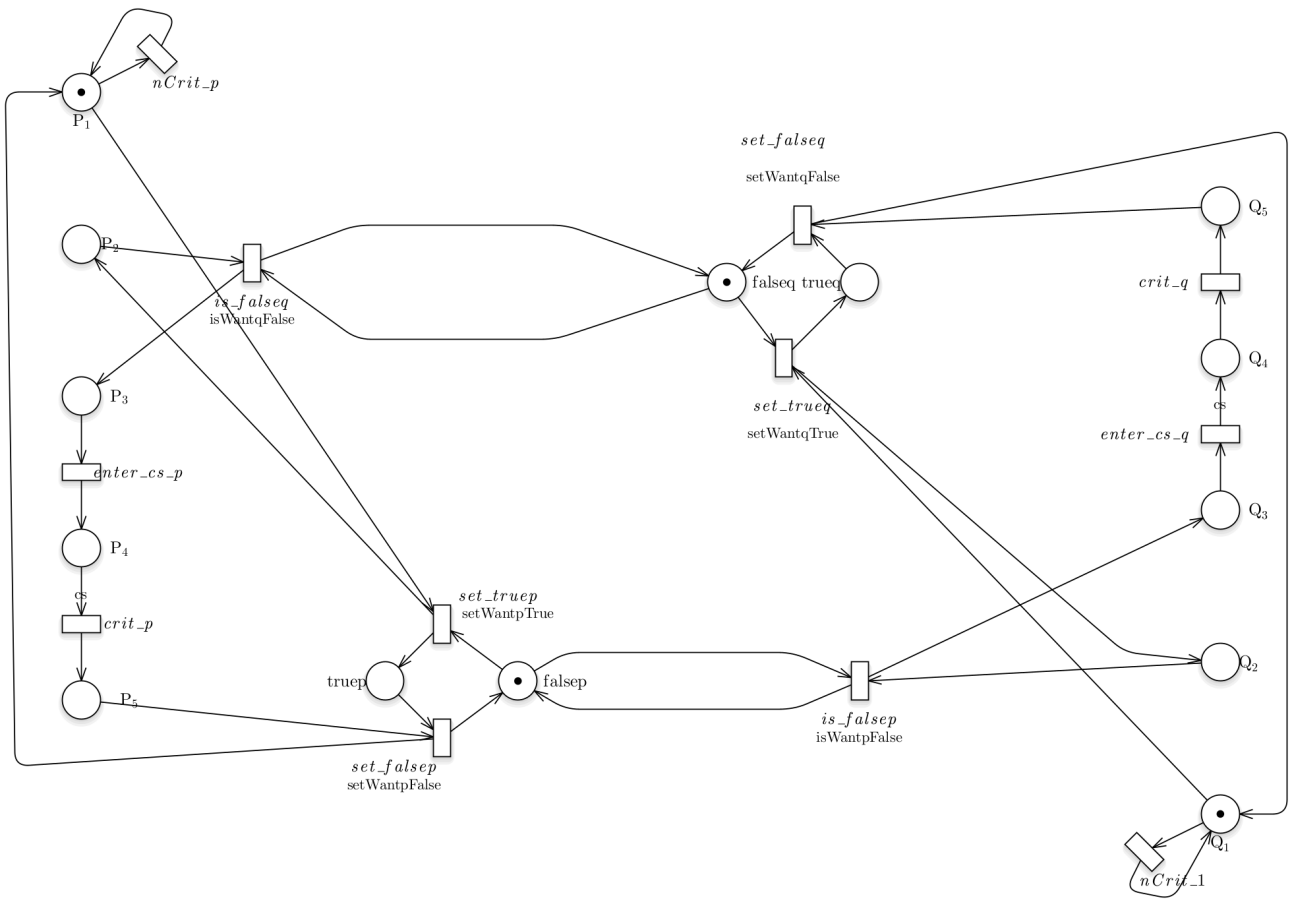


Figura 8: Rete di petri composta

6.1.1 RG

Il reachability graph, in figura 9, è composto 21 stati, di cui uno rappresenta una situazione di Deadlock.

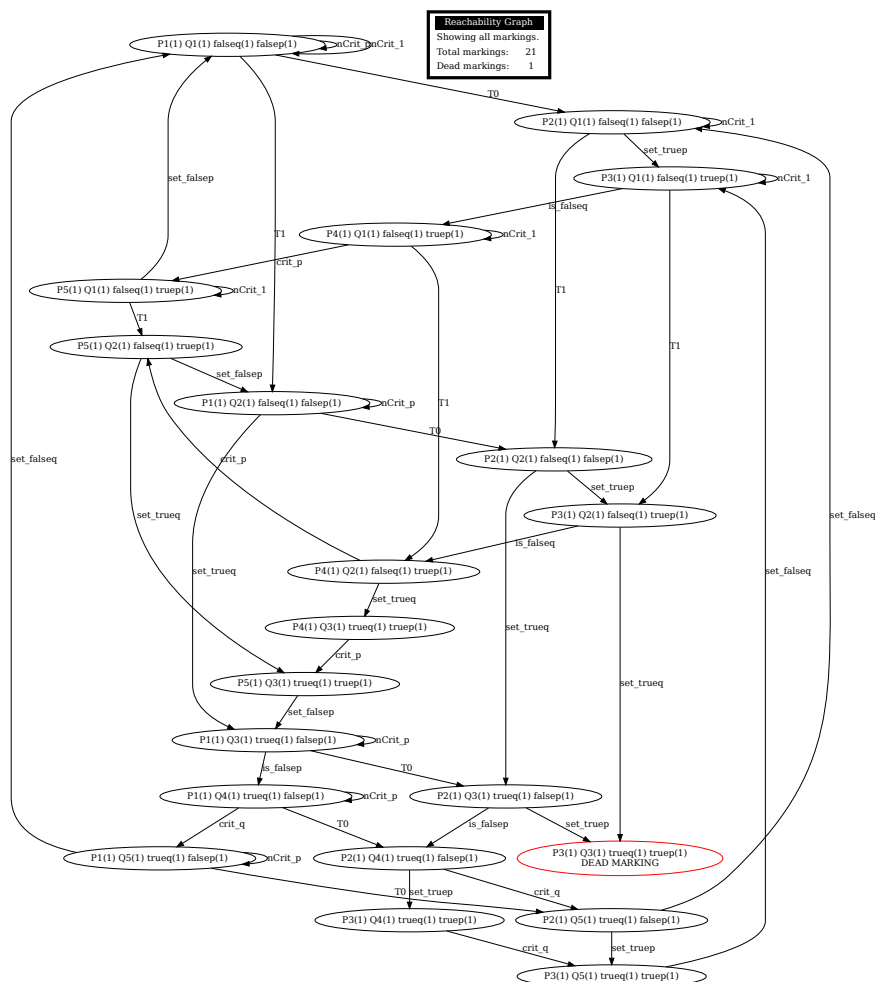


Figura 9: Reachability graph 3.8

6.1.2 Analisi strutturale

Il calcolo dei *semiflow* è sostanzialmente identico a quello effettuato in sezione 4.1.2 in quanto la struttura della rete non è cambiata molto. Lo studio dei *P-invarianti* rivela che tutti i posti sono 1-bound, che le variabili assumono un solo valore di verità alla volta e che i token che rappresentano il program counter si trovano in un solo posto alla volta per ogni processo. Anche lo studio dei *T-semiflow* è pressochè identico. Nonostante il sistema abbia uno stato di deadlock una *firing sequence* che riporta il sistema nella marcatura iniziale esiste, quindi il sistema possiede la proprietà di *liveness*.

6.1.3 Model Checking GreatSPN

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg(\#P4==1 \ \&\& \ \#Q4==1)$ **true**.
Il sistema rispetta la mutua esclusione.
- deadlock: $AG \ AF ((\#P1==1) \ || \ (\#Q1 == 1))$ **false**.
Il sistema va in deadlock, il controesempio presentato dal Model Checker è l'esecuzione $\{P_1Q_1 \rightarrow P_1Q_2 \rightarrow P_2Q_2\}$.
- assenza di starvation: $AG((\#P2 > 0) \rightarrow (AF \ i\#P4 > 0))$ ed anche $AG((\#P2 > 0) \rightarrow (AF \ i\#P4 > 0))$. Entrambe risultano **false**.
Essendoci deadlock c'è sempre la possibilità di starvation. Se il sistema va in deadlock dopo l'esecuzione presentata nel punto precedente entrambi i processi presentano starvation.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(\#P4==1 \ \&\& \ \#Q4==1)$ **true**.
- assenza di starvation: $G \ F (\#P2==1) \rightarrow G \ F (\#P4 == 1)$ ed anche $G \ F (\#Q2==1) \rightarrow G \ F (\#Q4 == 1)$. Entrambe risultano **false**.
- deadlock: $G \ F ((\#P1 == 1) \ || \ (\#Q1 == 1))$ **false**.

6.2 NuSMV

L'implementazione del sistema tramite il linguaggio di NuSMV è pressochè identica a quella della sezione 4.3, l'unica differenza sta nella posizione del controllo `!want_other` che questa volta viene effettuato dopo aver impostato la variabile `want_me` a true.

```

MODULE main
VAR
    wantp : boolean;
    wantq : boolean;
    p : process proc(wantp, wantq);
    q : process proc(wantq, wantp);
ASSIGN
    init(wantp) := FALSE;
    init(wantq) := FALSE;

SPEC AG (!(( p.state = s4 ) & ( q.state = s4 ))
SPEC AG (( p.state = s2 ) -> (AF p.state = s4 ))
SPEC AG EF (( p.state = s1 ) | ( q.state = s1 ))

LTLSPEC G !(p.state = s4 & q.state = s4)
LTLSPEC G (p.state = s2 -> F(p.state = s4))
LTLSPEC G F (( p.state = s1 ) | ( q.state = s1 ) )

MODULE proc(want_me, want_other)
VAR
    state : {s1, s2, s3, s4, s5};
ASSIGN
    init(state) := s1;
    next(state) :=
        case
            state = s1 : {s2, s1};
            state = s2 : s3;
            state = s3 & !want_other: s4;
            state = s4 : s5;
            state = s5 : s1;
            TRUE : state;
        esac;
    next(want_me) :=
        case
            state = s2 : TRUE;
            state = s5 : FALSE;
            TRUE : want_me;
        esac;

```

Il comando `print_reachable_states` mostra 21 stati raggiungibili di 100 possibili, in linea con la dimensione del Reachability Graph. È presente uno stato di deadlock : *p.state = s3, q.state = s3, wantp = true, wantq = true.*

6.2.1 Model Checking NuSMV

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg((p.state = s4) \wedge (q.state = s4))$ **true**.
- deadlock: $AG AF ((p.state = s1) \wedge (q.state = s1))$ **false**.
Il sistema va in deadlock, il controesempio presentato dal Model Checker è una esecuzione che fa eseguire tutti gli stati di q tornando a q.state= s1 e poi esegue $\{P_1Q_1 \rightarrow P_1Q_2 \rightarrow P_2Q_2\}$ che lo porta in deadlock.
- assenza di starvation: $AG ((p.state = s2) \rightarrow (AF p.state = s3))$ ed anche $AG ((q.state = s2) \rightarrow (AF q.state = s3))$. Entrambe risultano **false**.
Essendoci deadlock c'è sempre la possibilità di starvation. Se il sistema va in deadlock dopo l'esecuzione presentata precedente entrambi i processi presentano starvation. L'esempio presentato dal Model Checker è lo stesso del caso di deadlock.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(p.state = s4 \wedge q.state = s4)$ **true**.
- deadlock: $G F((p.running) \wedge (q.running))$ **false**.
Il sistema va in deadlock, il controesempio presentato dal Model Checker è una esecuzione che fa eseguire tutti gli stati di q tornando a q.state= s1 e poi esegue $\{P_1Q_1 \rightarrow P_1Q_2 \rightarrow P_2Q_2\}$ che lo porta in deadlock.
- assenza di starvation: $G (p.state = s2 \rightarrow F p.state = s3)$ ed anche $G (q.state = s2 \rightarrow F q.state = s3)$. Entrambe risultano **false**.
Essendoci deadlock c'è sempre la possibilità di starvation. Se il sistema va in deadlock dopo l'esecuzione presentata precedente entrambi i processi presentano starvation. L'esempio presentato dal Model Checker è lo stesso del caso di deadlock in CTL.

7 Confronto e riduzione di 3.6 e 3.8

7.1 Riduzione Algoritmo 3.6

Per prima cosa con l'operazione *RC2* è possibile eliminare i due self-loop *nCrit_p* e *nCrit_q*. I posti P_4, P_5 e Q_4, Q_5 possono essere ridotti tramite *RA1*. Le transizioni *access_crit_p* e *access_crit_q* possono essere rimosse tramite l'utilizzo di *RA1*. Si può infine unire *is_falseq* con *set_truep* e *is_falsep* con *set_trueq* grazie a *RA2*. La rete risultante è visibile in figura 10

7.2 Riduzione Algoritmo 3.8

Per prima cosa con l'operazione *RC2* è possibile eliminare i due self-loop *nCrit_p* e *nCrit_q*. Le transizioni *access_crit_p* e *access_crit_q* possono essere rimosse tramite l'utilizzo di *RA1*, unendo i posti P_1, P_2 e Q_1, Q_2 . I posti P_4, P_5 e Q_4, Q_5 possono essere ridotti tramite *RA1*. La rete risultante è visibile in figura 11

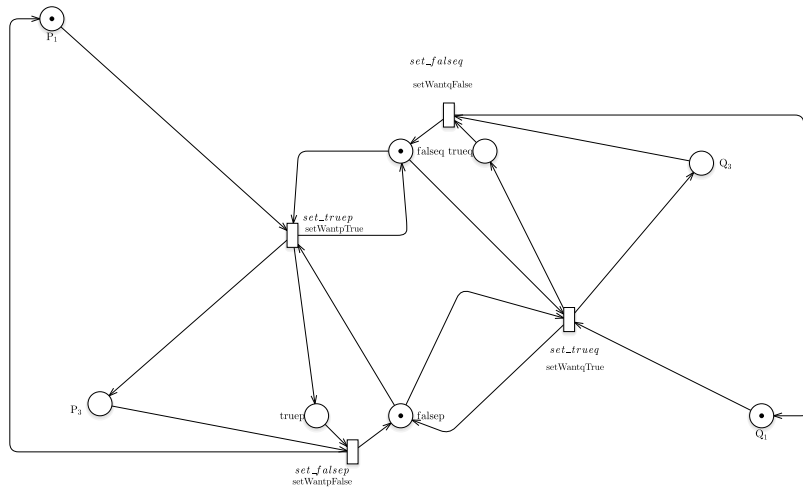


Figura 10: Rete algoritmo 3.6 ridotta

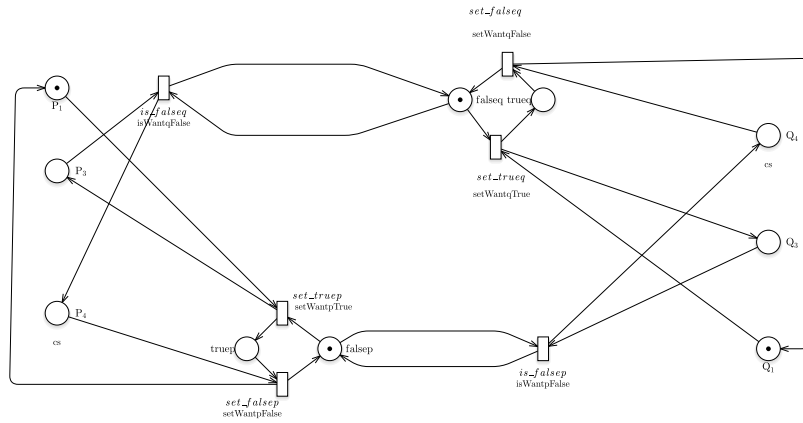
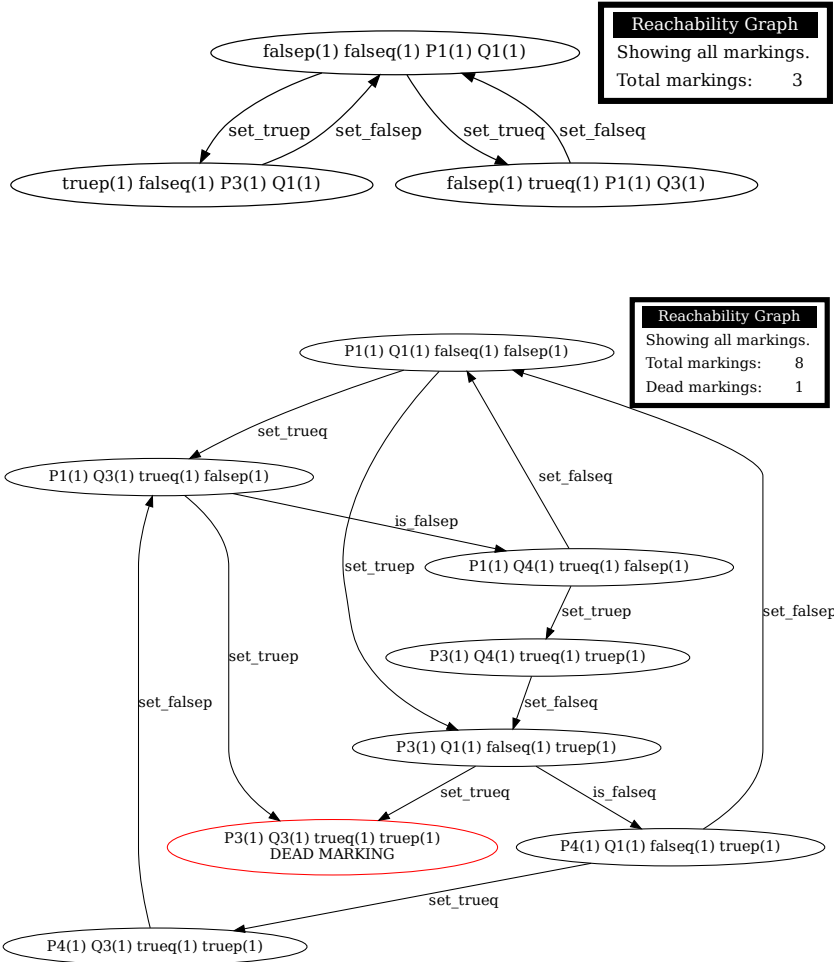


Figura 11: Rete algoritmo 3.8 ridotta

7.3 Confronto



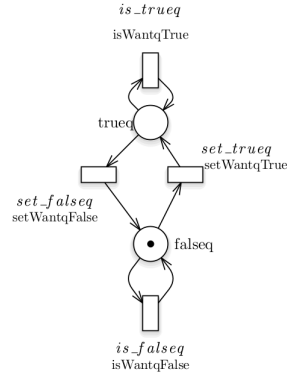
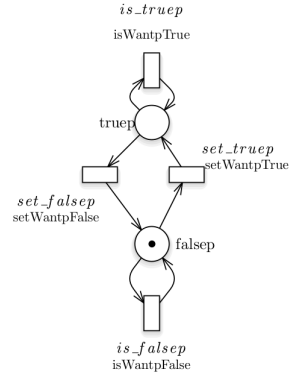
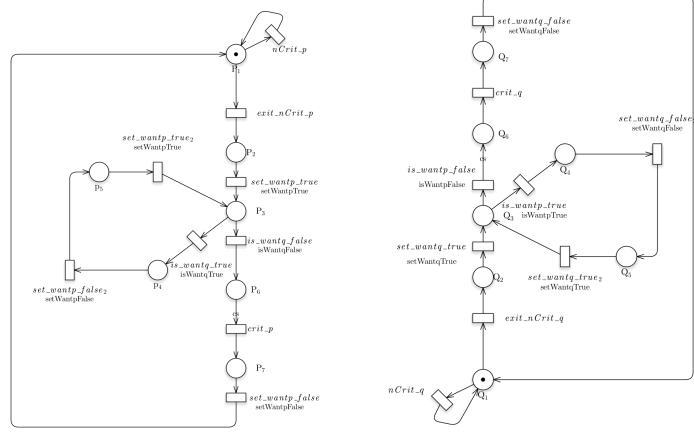
I due sistemi non sono equivalenti alle tracce, come prova si può prendere la sequenza $[\text{set_truep}, \text{set_trueq}]$ che può essere eseguita solo dall'algoritmo 3.8 e non dal 3.6. Non essendo equivalenti alle tracce sicuramente non sono equivalenti ai fallimenti, alla simulazione ed alla bisimulazione.

8 Algoritmo 3.9

Algorithm 3.9: Fourth attempt	
boolean wantp \leftarrow false, wantq \leftarrow false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp \leftarrow true	q2: wantq \leftarrow true
p3: while wantq	q3: while wantp
p4: wantp \leftarrow false	q4: wantq \leftarrow false
p5: wantp \leftarrow true	q5: wantq \leftarrow true
p6: critical section	q6: critical section
p7: wantp \leftarrow false	q7: wantq \leftarrow false

Questo algoritmo cerca di risolvere il deadlock del 3.8 4 andando a resettare il valore della variabile che si riferisce al voler entrare in sezione critica qualora non si riesca ad entrare.

8.1 Rete di Petri



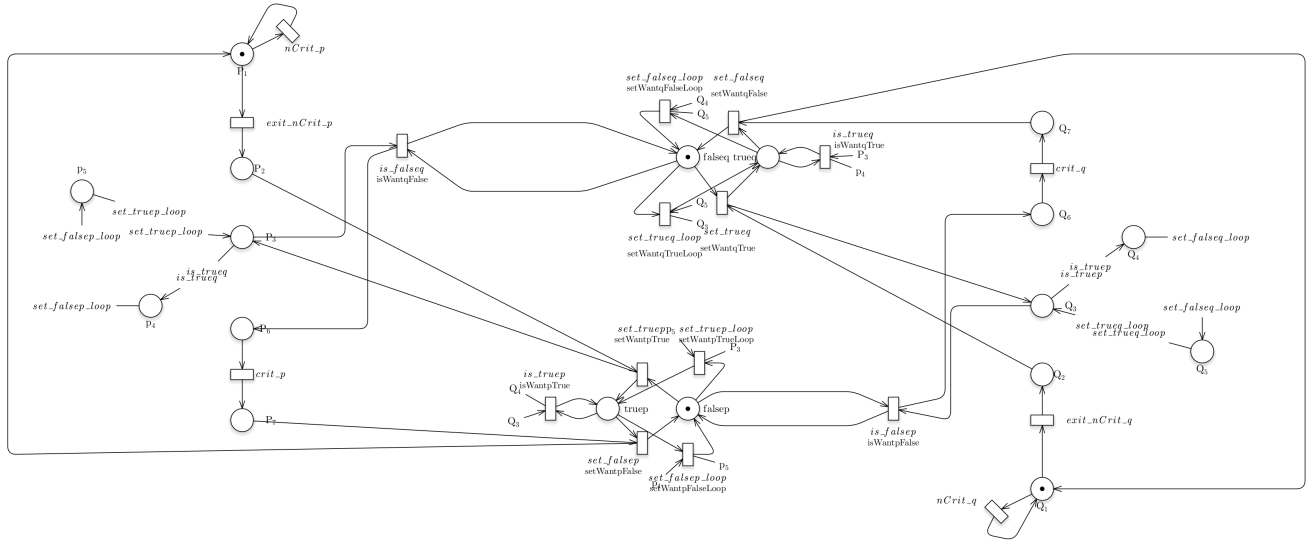


Figura 12: Rete di petri composta

8.1.1 RG

Il reachability graph, in figura 13, è composto da 45 stati, di cui nessuno rappresentante una situazione di Deadlock.



Figura 13: Reachability graph 3.9

8.1.2 Analisi strutturale

Il calcolo dei *semiflow* fornisce 6 *T-semiflow* minimali e 8 *P-semiflow* minimali, lo studio dei P-invarianti rivela che tutti i posti sono 1-bound come negli altri casi e permette di affermare che le variabili p e q assumono un solo valore di verità. Invece dallo studio dei *T-semiflow* non emerge solo la *liveness* (come prevedibile) ma è interessante notare la presenza dei *T-semiflow* che interessano i posti [P3, P4, P5] e [Q3, Q4, Q5] cioè le esecuzioni del ciclo di P e Q.

8.1.3 Model Checking GreatSPN

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg(\#P6==1 \ \&\& \ \#Q6==1)$ **true**.
Il sistema rispetta la mutua esclusione.
- deadlock: $AG \ AF \ ((\#P1==1) \parallel (\#Q1 == 1))$ **false**.
Il sistema sembra andare in deadlock, il controesempio presentato dal Model Checker è l'esecuzione in cui entrambi i processi entrano nel loop di richiesta e si bloccano a vicenda l'uscita.
Ma questo non è deadlock, è starvation, quindi la formula logica usata per controllare l'assenza di deadlock in questo caso va modificata.
- deadlock corretto: $AG \ AF \ (((\#P1==1) \parallel (\#Q1 == 1)) \parallel ((\#P4==1) \parallel (\#Q4 == 1)))$ **true**.
In questo modo viene controllato che i processi siano liberi di eseguire o il ciclo esterno (dallo stato 1 fino al ritorno in stato 1) o almeno quello interno (dallo stato 3 ritorno allo stato 3).
- assenza di starvation: $AG((\#P2 > 0) \rightarrow (AF \ \#P6 > 0))$ ed anche $AG((\#P2 > 0) \rightarrow (AF \ \#Q6 > 0))$. Entrambe risultano **false**.
Come anticipato prima si può verificare starvation. Sia perchè un processo riesce sempre ad entrare in CS escludendo l'accesso all' altro (situazione mostrata come controesempio dal Model Checker) oppure perchè entrambi i processi si bloccano a vicenda l'accesso.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(\#P4==1 \ \&\& \ \#Q4==1)$ **true**.
- assenza di starvation: $G \ F \ (\#P2==1) \rightarrow G \ F \ (\#P6 == 1)$ ed anche $G \ F \ (\#Q2==1) \rightarrow G \ F \ (\#Q6 == 1)$. Entrambe risultano **false**.
- deadlock: $G \ F \ ((\#P1 == 1) \parallel (\#Q1 == 1))$ **false**.
- deadlock corretto: $G \ F \ ((\#P1==1) \parallel (\#Q1 == 1)) \parallel G \ F \ ((\#P4==1) \parallel (\#Q4 == 1))$ **true**.

8.2 NuSMV

L'implementazione del sistema tramite il linguaggio di NuSMV non presenta particolari differenze rispetto allo pseudocodice 3.9. Allo stato 5 vi è il salto che permette di creare il ciclo interno e si può uscire da questo ciclo solamente se la variabile *want_other* è falsa quando il processo è nello stato s3.

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

MODULE proc(want_me, want_other)
VAR
    state : {s1, s2, s3, s4, s5, s6, s7};
ASSIGN
    init(state) := s1;
    next(state) :=
        case
            state = s1 : {s2, s1};
            state = s2 : s3;
            state = s3 & want_other: s4;
            state = s3 & !want_other: s6;
            state = s4 : s5;
            state = s5 : s3;
            state = s6 : s7;
            state = s7 : s1;
            TRUE : state;
        esac;
    next(want_me) :=
        case
            state = s2 : TRUE;
            state = s5 : TRUE;
            state = s7 : FALSE;
            TRUE : want_me;
        esac;
```

Il comando `print_reachable_states` mostra 45 stati raggiungibili di 196 possibili, in linea con la dimensione del Reachability Graph. Non è presente alcuno stato di deadlock.

8.2.1 Model Checking NuSMV

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg((p.state = s6) \wedge (q.state = s6))$ **true**.
- deadlock: $AG AF ((p.state = s1) \wedge (q.state = s1))$ **false**.
Si verifica di nuovo la stessa situazione presentata in GreatSPN, la formula è incorretta per verificare l'assenza di deadlock. Va quindi riscritta.
- deadlock corretto: $AG AF (((p.state = s1) \wedge (q.state = s1)) \vee ((p.state = s4) \wedge (q.state = s4)))$ **true**.
- assenza di starvation: $AG ((p.state = s2) \rightarrow (AF p.state = s3))$ ed anche $AG ((q.state = s2) \rightarrow (AF q.state = s3))$. Entrambe risultano **false**.

Come visto nel Model Checking in GreatSPN si può verificare starvation sia perchè un processo blocca l'altro e va solo lui in critical section ma anche perchè entrambi si bloccano l'accesso a vicenda. Il controesempio fornito dal MC è una istanza del secondo tipo.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(p.state = s4 \wedge q.state = s4)$ **true**.
- deadlock: $G F((p.running) \wedge (q.running))$ **true**.
In questo caso il problema di definizione della formula logica "il processo può essere eseguito" non si pone, in quanto NuSMV permette di utilizzare la clausola **running** per stabilire se un processo sta venendo eseguito o no (questo solo in LTL).
Quindi il Model Checker mostra correttamente l'assenza di Deadlock.
- assenza di starvation: $G (p.state = s2 \rightarrow F p.state = s3)$ ed anche $G (q.state = s2 \rightarrow F q.state = s3)$. Entrambe risultano **false**.
L'esempio presentato dal Model Checker è lo stesso dell'analisi con CTL.

9 Algoritmo 3.10

Algorithm 3.10: Dekker's algorithm	
boolean wantp \leftarrow false, wantq \leftarrow false	
integer turn \leftarrow 1	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp \leftarrow true	q2: wantq \leftarrow true
p3: while wantq	q3: while wantp
p4: if turn = 2	q4: if turn = 1
p5: wantp \leftarrow false	q5: wantq \leftarrow false
p6: await turn = 1	q6: await turn = 2
p7: wantp \leftarrow true	q7: wantq \leftarrow true
p8: critical section	q8: critical section
p9: turn \leftarrow 2	q9: turn \leftarrow 1
p10: wantp \leftarrow false	q10: wantq \leftarrow false

L'algoritmo in figura risolve il problema della starvation dell'algoritmo 3.9 8 sfruttando l'idea dell'algoritmo 3.2 3 di inserire una variabile turn per decidere quale processo entrerà in sezione critica. A differenza del 3.2 però il controllo del turno viene effettuato solamente se c'è effettivamente qualcun'altro in attesa di entrare in sezione critica.

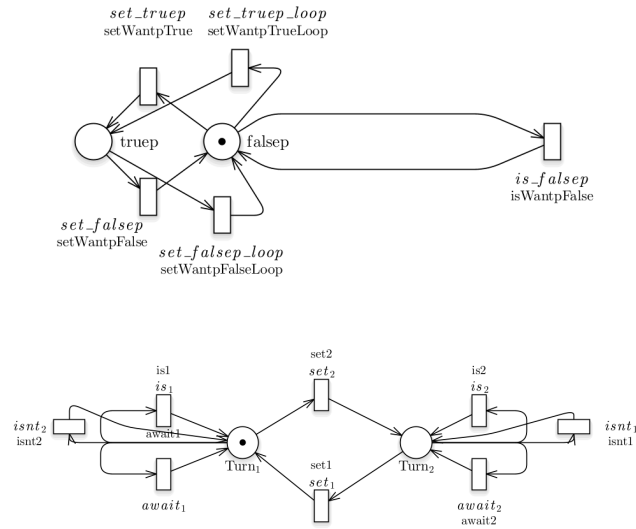
Per ogni operazione effettuabile sulle variabili è stata aggiunta una nuova transizione alle stesse. Quindi la variabile *wantP* avrà due transizioni atte al settarla vera o falsa, una usata fuori dal ciclo interno ed una all'interno (transizione *set_truep*). Stesso discorso vale per la variabile turn. A differenza di quella presentata in sezione 3

9.1 Disclaimer

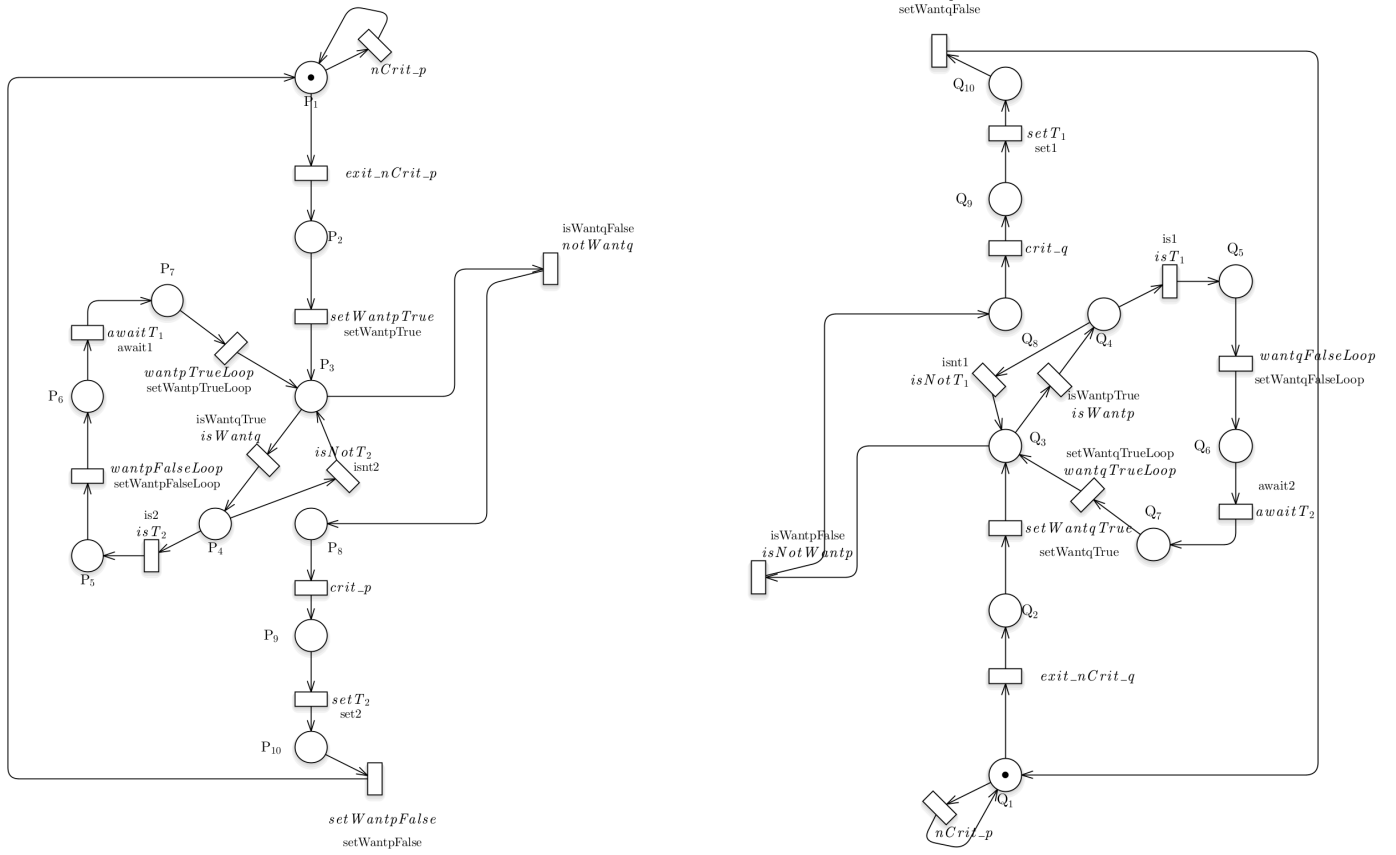
Questo algoritmo non dovrebbe soffrire di starvarion, invece la sua implementazione in Petri net ne soffre. Questo è effettivamente un *tranello* della prof. Vuole che vi rendiate conto della cosa e, per quanto non sia impossibile generare un PN con un arbitraggio che rende fair il modello, basta che all'esame le dite ciò.

Insomma quello che ho fatto io è andato bene, le ho spiegato che non avevo trovato il modo e lei mi ha risposto qualcosa sulla falsa riga di "non mi aspettavo lo trovaste, mi aspettavo mi chiedeste aiuto"

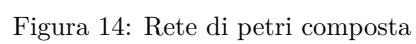
9.2 Rete di Petri



Variabili P e Q e variabile $Turn$



Processi P e Q



9.2.1 RG

Il reachability graph, in figura 15, è composto da 134 stati, di cui nessuno rappresentante una situazione di Deadlock.

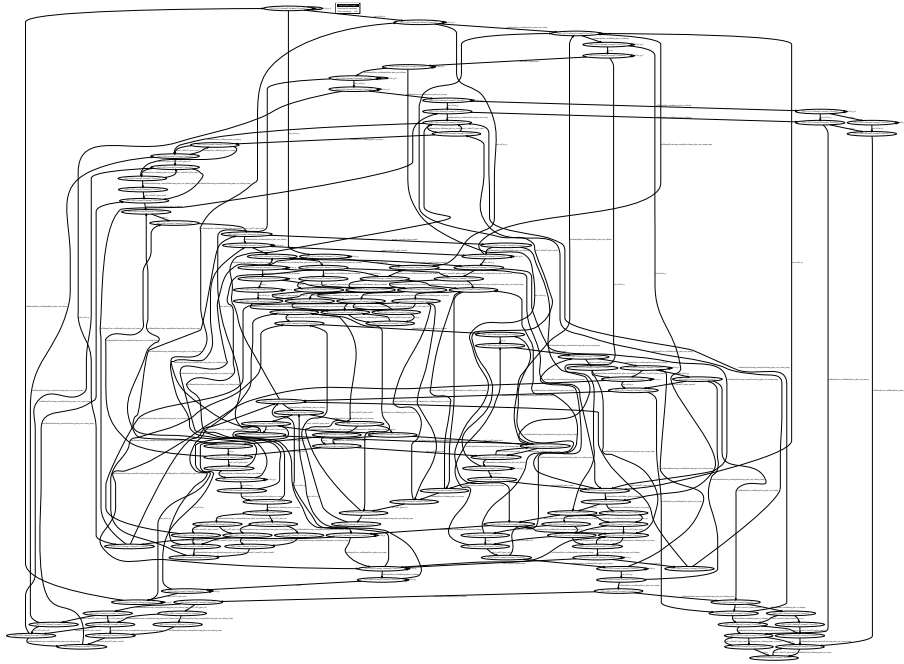


Figura 15: Reachability graph 3.10

9.2.2 Analisi strutturale

In quest'ultimo caso possiamo identificare 9 *P-semiflow* minimali e 7 *T-semiflow* minimali. Le variabili *P* e *Q* possono assumere un solo valore di verità ed la marcatura della variabile *Turn* può essere solamente nel posto *Turn₁* o alternativamente nel posto *Turn₂*. Come negli altri casi il sistema è 1-bound.

L'analisi dei *T-semiflow* mostra che il sistema possiede la proprietà di *liveness* ed è interessante analizzarne 3. Come nel caso precedente (sezione 8.1.2) sono presenti due semiflow che corrispondono ai cicli ma a differenza di tutti i casi precedenti sono assenti i semiflow che coinvolgono separatamente tutti i posti di *P* e tutti i posti di *Q*. Vi è invece un unico flow che interessa tutti i posti di entrambi i processi, è quindi possibile attraverso una esecuzione dei due processi ritornare allo stato iniziale, questo rende la rete una rete **consistente**.

9.2.3 Model Checking GreatSPN

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg(\#P8==1 \wedge \#Q8==1)$ **true**.
- assenza di starvation: $AG((\#P3 > 0) \rightarrow (AF \#P8 > 0))$ ed anche $AG((\#Q3 > 0) \rightarrow (AF \#Q6 > 0))$. Entrambe risultano **false**.

Il motivo di questa incongruenza è dovuto ad un errore nei fairness constraint $(\#P2 > 0 \parallel \#Q2 > 0)$. Infatti questi non escludono la possibilità che un processo rimanga in uno stato relativo alla richiesta della sezione critica e l'esecuzione viene concessa solo all'altro processo che cicla in sezione non critica. Questa computazione non fair deve essere gestita in altro modo.

- assenza di starvation imponendo progresso in regione non critica: Fairness constraint $\#P2 > 0 \parallel \#Q2 > 0$. Entrambe risultano **true**.

Un modo di imporre una computazione fair è garantire il progresso del processo opposto a quello di cui si vuole verificare l'assenza di starvation. La proposizione $(\#Q2 == 1)$ ad esempio garantisce che il processo *Q* uscirà dalla sezione non critica e tenterà l'accesso in *cs*, riuscendo ad accedere o permettendo l'esecuzione di *P*. In tal modo però i processi sono obbligati al progresso in sezione non critica. Un'implementazione alternativa consiste nella creazione di un arbitro che assegni il *quanto* di tempo di esecuzione ai processi a seconda di come viene costruito, di fatto specificando la politica di scheduling.

- deadlock: $AG AF ((\#P1==1) \parallel (\#Q1 == 1))$ **true**.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(\#P8==1 \wedge \#Q8==1)$ **true**.
- assenza di starvation: $G F (\#P3==1) \rightarrow G F (\#P8 == 1)$ ed anche $G F (\#Q3==1) \rightarrow G F (\#Q8 == 1)$. Entrambe risultano **false**.

In questo caso si verifica starvation, anche se non dovrebbe succedere. Ciò è dovuto ai *fairness constraint*. Non è infatti vero che se un processo richiede l'accesso in sezione critica, poichè potrebbe avvenire una computazione non fair in cui il processo che richiede l'accesso rimane bloccato su uno

qualsiasi dei posti 3, ..., 8 mentre l'altro processo prosegue in sezione non critica.

- assenza di starvation imponendo progresso in regione non critica: $G F (\#Q2 == 1) \rightarrow (G F (\#Q3 == 1) \rightarrow G F (\#Q8 == 1))$ E la proposizione corrispondente per Q risultano **true**.

Un modo di imporre una computazione fair è garantire il progresso del processo opposto a quello di cui si vuole verificare l'assenza di starvation. La proposizione $(\#Q2 == 1)$ ad esempio garantisce che il processo Q uscirà dalla sezione non critica e tenterà l'accesso in cs , riuscendo ad accedere o permettendo l'esecuzione di P . In tal modo però i processi sono obbligati al progresso in sezione non critica. Un'implementazione alternativa consiste nella creazione di un arbitro che assegni il *quanto* di tempo di esecuzione ai processi a seconda di come viene costruito, di fatto specificando la politica di scheduling.

- deadlock: $G F ((\#P1 == 1) \parallel (\#Q1 == 1))$ **true**.

9.3 NuSMV

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

MODULE proc(want_me, want_other, turn, my_turn, other_turn)
VAR
    state : {s1, s2, s3, s4, s5, s6, s7, s8, s9, s10};
ASSIGN
    init(state) := s1;
    next(state) :=
        case
            state = s1 : {s2, s1};
            state = s2 : s3;
            state = s3 & want_other: s4;
            state = s3 & !want_other: s8;
            state = s4 & (turn = other_turn): s5 ;
            state = s4 & !(turn = other_turn): s3 ;
            state = s5 : s6;
            state = s6 & (turn = my_turn): s7;
            state = s7 : s3;
            state = s8 : s9;
            state = s9 : s10;
            state = s10 : s1;
            TRUE : state;
        esac;
    next(want_me) :=
        case
            state = s2 : TRUE;
            state = s5 : FALSE;
            state = s7 : TRUE;
            state = s10 : FALSE;
            TRUE : want_me;
        esac;
    next(turn) :=
        case
            state = s9 : other_turn;
            TRUE : turn;
        esac;
FAIRNESS running;
```

Il comando `print_reachable_states` mostra 134 stati raggiungibili di 800 possibili, in linea con la dimensione del Reachability Graph. Non è presente alcuno stato di deadlock.

9.3.1 Model Checking NuSMV

Sono state verificate le seguenti formule CTL:

- mutua esclusione: $AG \neg((p.state = s8) \wedge (q.state = s8))$ **true**.
- assenza di starvation: $AG ((p.state = s3) \rightarrow (AF p.state = s8))$ ed anche $AG ((q.state = s3) \rightarrow (AF q.state = s8))$. Entrambe risultano **true**.
- assenza di deadlock: $AG AF ((p.state = s1) \vee (q.state = s1))$ **true**.

Sono state verificate le seguenti formule LTL:

- mutua esclusione: $G \neg(p.state = s8 \wedge q.state = s8)$ **true**.
- assenza di starvation: $G (p.state = s3 \rightarrow F p.state = s8)$ ed anche $G (q.state = s3 \rightarrow F q.state = s8)$. Entrambe risultano **true**.
A differenza del model checker di *GreatSPN*, *NuSMV* implementa la possibilità di selezionare uno scheduler fair tramite il comando **FAIRNESS running**, non vi è quindi necessità di inserire constraint nella proposizione.
- assenza di deadlock: $G F((p.running) \vee (q.running))$ **true**.

10 Confronto

Come si può notare dalla seguente tabella non è presente nessuna differenza tra le implementazioni effettuate con *GreatSPN* e le implementazioni in *NuSMV*.

GreatSPN				NuSMV		
	Stati raggiungibili	Assenza di deadlock	Mutua esclusione	Stati raggiungibili	Assenza di deadlock	Mutua esclusione
3.2	16	true	true	16	true	true
3.6	25	true	false	25	true	false
3.8	21	false	true	21	false	true
3.9	45	true	true	45	true	true
3.10	134	true	true	134	true	true

L'unica differenza riguarda la starvation dell'algoritmo 3.10, non rappresentata in tabella.

L'implementazione con *GreatSPN* soffre infatti di starvation, quella con *NuSMV* invece no. Come appurato in precedenza la causa di ciò è la differenza nella definizione dei fairness constraint.

Mentre *NuSMV* offre uno scheduler fair implicito, *GreatSPN* richiede che questi sia esplicito. Non essendo stato in grado di esprimere una politica di scheduling con il solo utilizzo di formule CTL e LTL non ho potuto dimostrare che l'algoritmo 3.10 senza obbligo di progresso in sezione non critica garantisce l'assenza di starvation.

Come affermato in precedenza una possibile soluzione sarebbe stata quella di costruire un arbitro, ciò mi avrebbe permesso di implementare una politica di scheduling e di impostare dei fairness constraint.

Non ho seguito questa strada in quanto l'implementazione con l'arbitro si sarebbe discostata dal modello di partenza. Ho preferito rimanere il più aderente possibile al codice degli algoritmi perchè suppongo sia meglio verificare formalmente un modello il più simile possibile al modello iniziale ed anche perchè l'implementazione dello scheduling avrebbe causato discrepanze tra il modello fatto con GreatSPN e quello fatto con NuSMV.