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.

$$\mathrm{G}\ (\neg \mathrm{c}_p \vee \neg \mathrm{c}_\mathrm{q})$$

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.

$$\mathrm{G}((\mathrm{w}_p \vee \mathrm{w}_\mathrm{q}) \rightarrow \mathrm{F}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q}))$$

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

$$\mathrm{G}(\mathrm{w}_p \rightarrow \mathrm{Fc}_p)$$

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

$$\forall \mathrm{p},\ \mathrm{G}(\mathrm{w}_p \rightarrow \mathrm{F}\mathrm{c}_p)$$

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

$$\mathrm{AG}\ (\neg \mathrm{c}_p \vee \neg \mathrm{c}_\mathrm{q})$$
$$\mathrm{AG}(\mathrm{w}_p \rightarrow \mathrm{AF}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q})$$
$$\mathrm{AG}(\mathrm{w}_p \rightarrow \mathrm{AF}\mathrm{c}_p)$$

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:

$$\mathrm{G}((\mathrm{w}_p \vee \mathrm{w}_\mathrm{q}) \rightarrow \mathrm{F}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q}) = \mathrm{false} \rightarrow \mathrm{G}((\mathrm{l}_p \vee \mathrm{l}_\mathrm{q}) \rightarrow \mathrm{F}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q})$$
$$\mathrm{G}((\mathrm{l}_p \vee \mathrm{l}_\mathrm{q}) \rightarrow \mathrm{F}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q}) = \mathrm{true} \rightarrow \mathrm{G}((\mathrm{w}_p \vee \mathrm{w}_\mathrm{q}) \rightarrow \mathrm{F}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q})$$

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.

$$\text{state: } \{\text{begin, wait, critical, done}\};$$

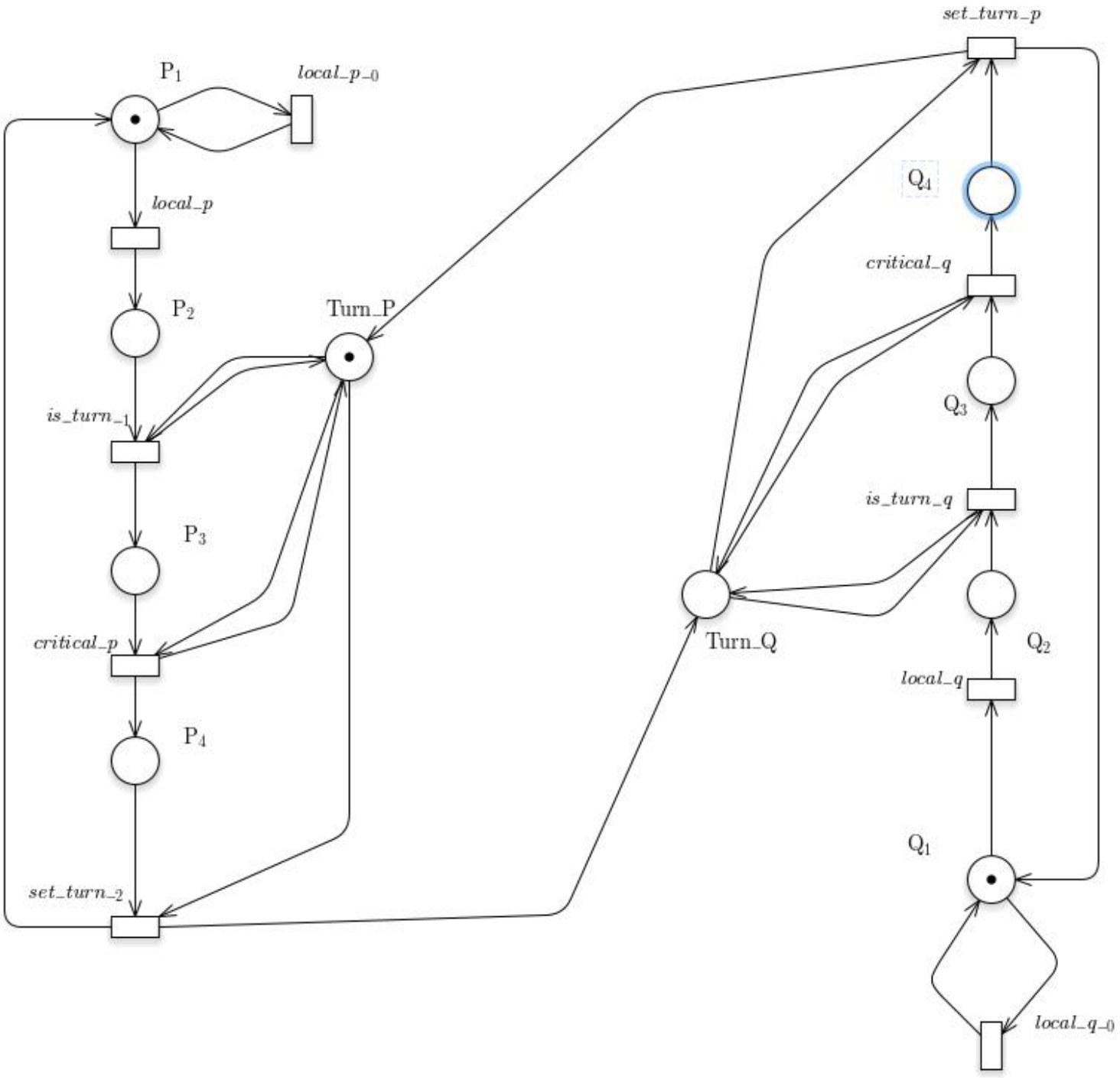
```
MODULE main
VAR
  turn: 1 .. 2;
  p: proc(turn, 1);
  q: proc(turn, 2);
ASSIGN
    init(turn) := 1;
    next(turn) :=
              case
                   q.state = done: 1;
                   p.state = done: 2;
                   TRUE: turn;
              esac;

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)


LTLSPEC -- no mutual exclusion
  G (p.state != critical | q.state != critical)
LTLSPEC -- no deadlock
  G ((p.state = wait | q.state = wait) -> F (p.state = critical | q.
      state = critical))
LTLSPEC -- 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))
```


set_turn_p
P1
local_p_0
local_p
Q4
critical_q
P2
Turn_P
is_turn_1
Q3
is_turn_q
P3
Turn_Q
Q2
critical_p
local_q
P4
Q1
set_turn_2
local_q_0

2.3 Algebra dei processi

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

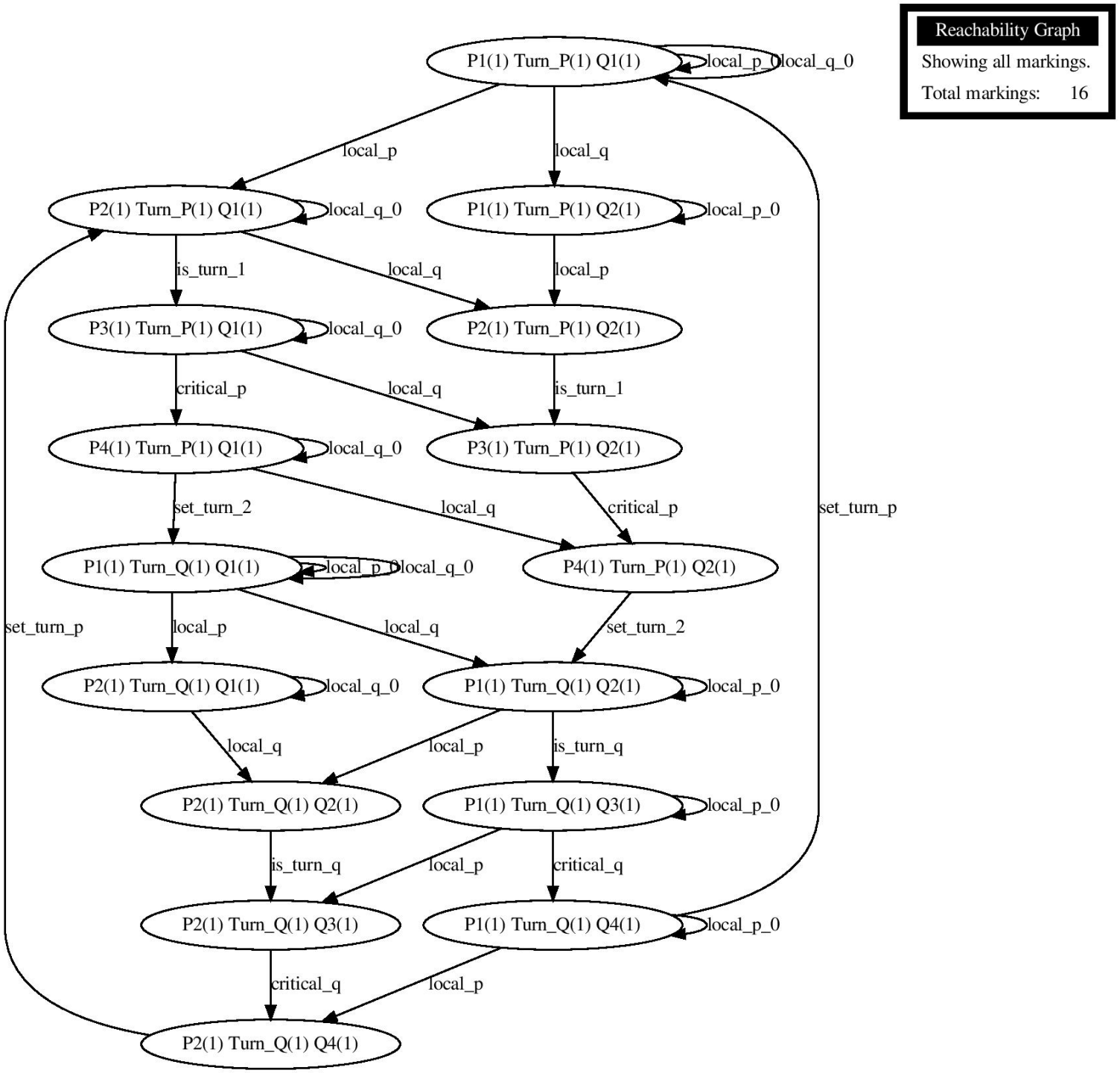
$$\text{System} = \{(P_1 \parallel Q_1) \parallel T_p\} \;/\; \text{Sync}$$
$$\text{Act} = \{\text{local}_p, \text{local}_q, \text{critical}_p, \text{critical}_q\}$$
$$\text{Sync} = \{\text{is}_p, \text{is}_q, \text{set}_p, \text{set}_q\}$$

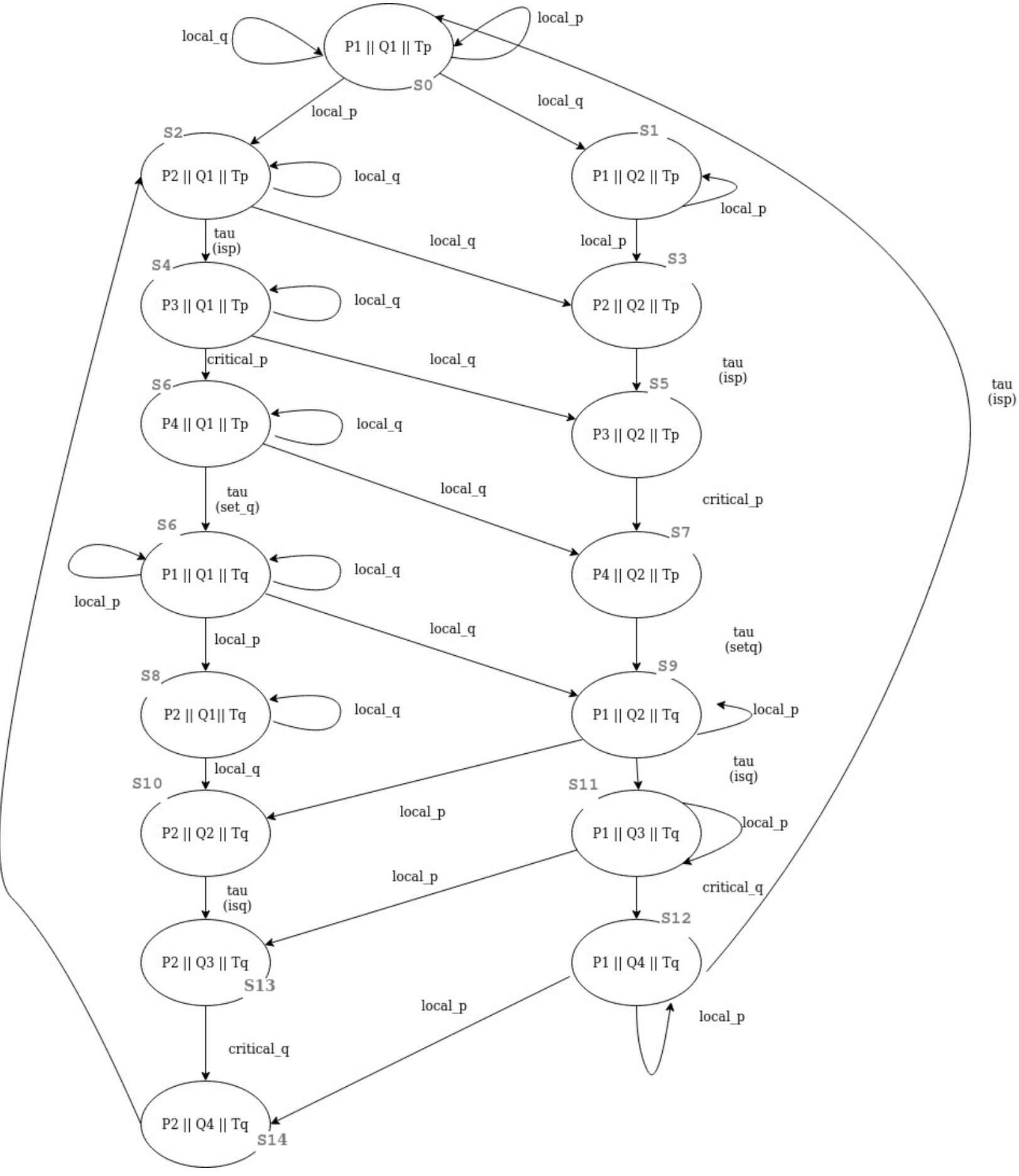
$$T_p ::= \text{is}_p.T_p + \text{set}_p.T_p + \text{set}_q.T_q$$
$$T_q ::= \text{is}_q.T_q + \text{set}_p.T_p + \text{set}_q.T_q$$

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

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

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


Reachability Graph
Showing all markings.
Total markings: 16
P1(1) Turn_P(1) Q1(1)
local_p_0
local_q_0
local_p
local_q
P2(1) Turn_P(1) Q1(1)
local_q_0
P1(1) Turn_P(1) Q2(1)
local_p_0
is_turn_1
local_q
local_p
P3(1) Turn_P(1) Q1(1)
local_q_0
P2(1) Turn_P(1) Q2(1)
critical_p
local_q
is_turn_1
P4(1) Turn_P(1) Q1(1)
local_q_0
P3(1) Turn_P(1) Q2(1)
set_turn_2
local_q
critical_p
set_turn_p
P1(1) Turn_Q(1) Q1(1)
local_p_0
local_q_0
P4(1) Turn_P(1) Q2(1)
set_turn_p
local_p
local_q
set_turn_2
P2(1) Turn_Q(1) Q1(1)
local_q_0
P1(1) Turn_Q(1) Q2(1)
local_p_0
local_q
local_p
is_turn_q
P2(1) Turn_Q(1) Q2(1)
P1(1) Turn_Q(1) Q3(1)
local_p_0
is_turn_q
local_p
critical_q
P2(1) Turn_Q(1) Q3(1)
P1(1) Turn_Q(1) Q4(1)
local_p_0
critical_q
local_p
P2(1) Turn_Q(1) Q4(1)


P1 || Q1 || Tp
S0
local_q
local_p
local_p
local_q
S2
P2 || Q1 || Tp
local_q
S1
P1 || Q2 || Tp
local_p
tau
(isp)
local_q
local_p
S4
P3 || Q1 || Tp
local_q
S3
P2 || Q2 || Tp
critical_p
local_q
tau
(isp)
S6
P4 || Q1 || Tp
local_q
S5
P3 || Q2 || Tp
tau
(isp)
tau
(set_q)
local_q
critical_p
S6
P1 || Q1 || Tq
local_q
S7
P4 || Q2 || Tp
local_p
local_p
local_q
tau
(setq)
S8
P2 || Q1|| Tq
local_q
S9
P1 || Q2 || Tq
local_p
local_q
tau
(isq)
S10
P2 || Q2 || Tq
local_p
S11
P1 || Q3 || Tq
local_p
local_p
critical_q
tau
(isq)
S12
P2 || Q3 || Tq
S13
P1 || Q4 || Tq
local_p
local_p
critical_q
P2 || Q4 || Tq
S14

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

$$\text{AG } (w_p \text{ -> EF } c_p)$$

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)


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

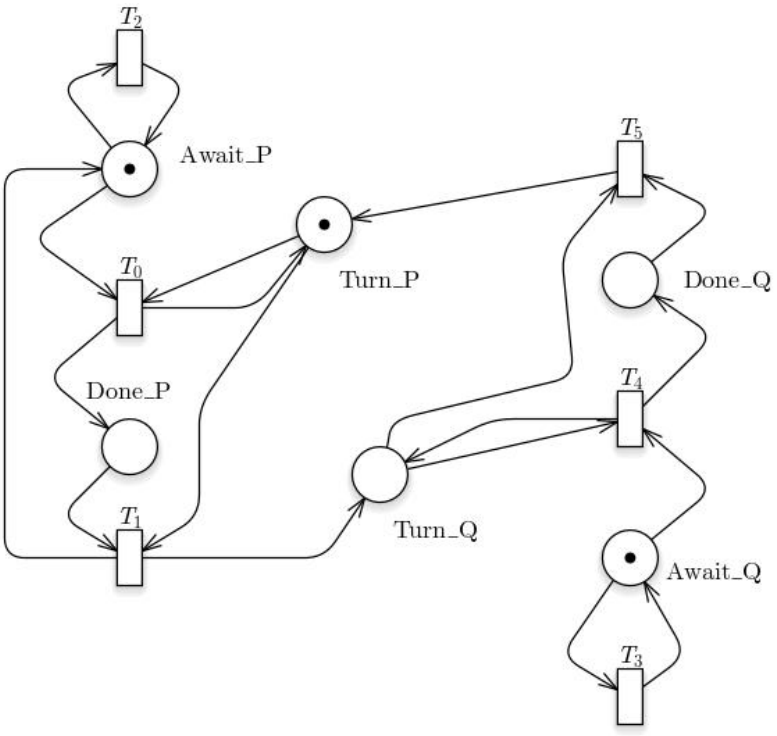


Figure 1: Rete 3.5

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};

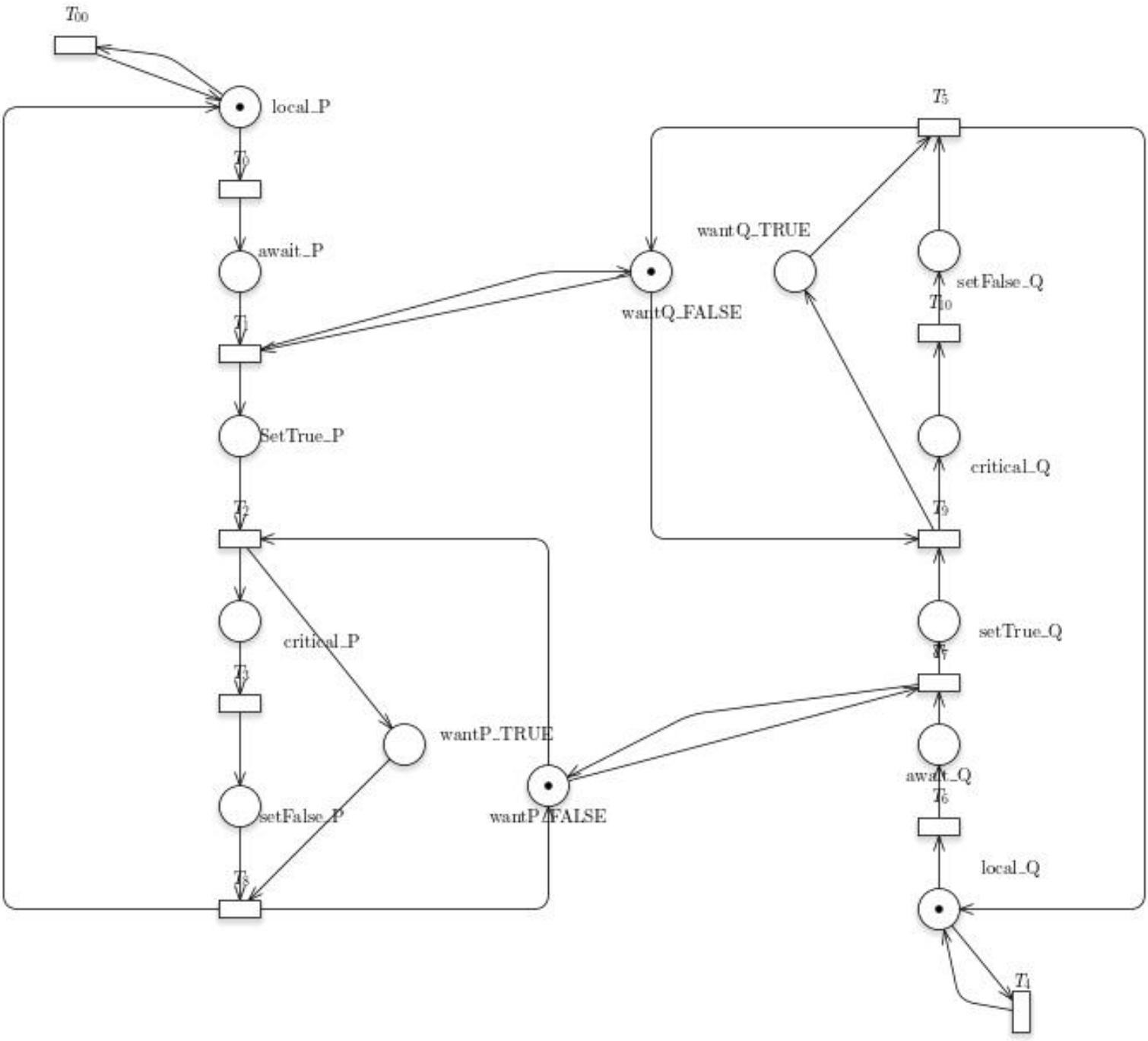
```
MODULE main
VAR
    wantP: boolean;
    wantQ: boolean;
    p: process proc(wantP, wantQ);
    q: process proc(wantQ, wantP);
ASSIGN
    init(wantP) := FALSE;
    init(wantQ) := FALSE;

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)



LTLSPEC -- no 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 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))
```


T_{00}
local_P
T_0
await_P
T_1
SetTrue_P
T_2
critical_P
T_3
wantP_TRUE
setFalse_P
T_8
wantP_FALSE
T_5
wantQ_TRUE
wantQ_FALSE
setFalse_Q
T_{10}
critical_Q
T_9
setTrue_Q
T_7
await_Q
T_6
local_Q
T_4

4.3 Algebra dei processi

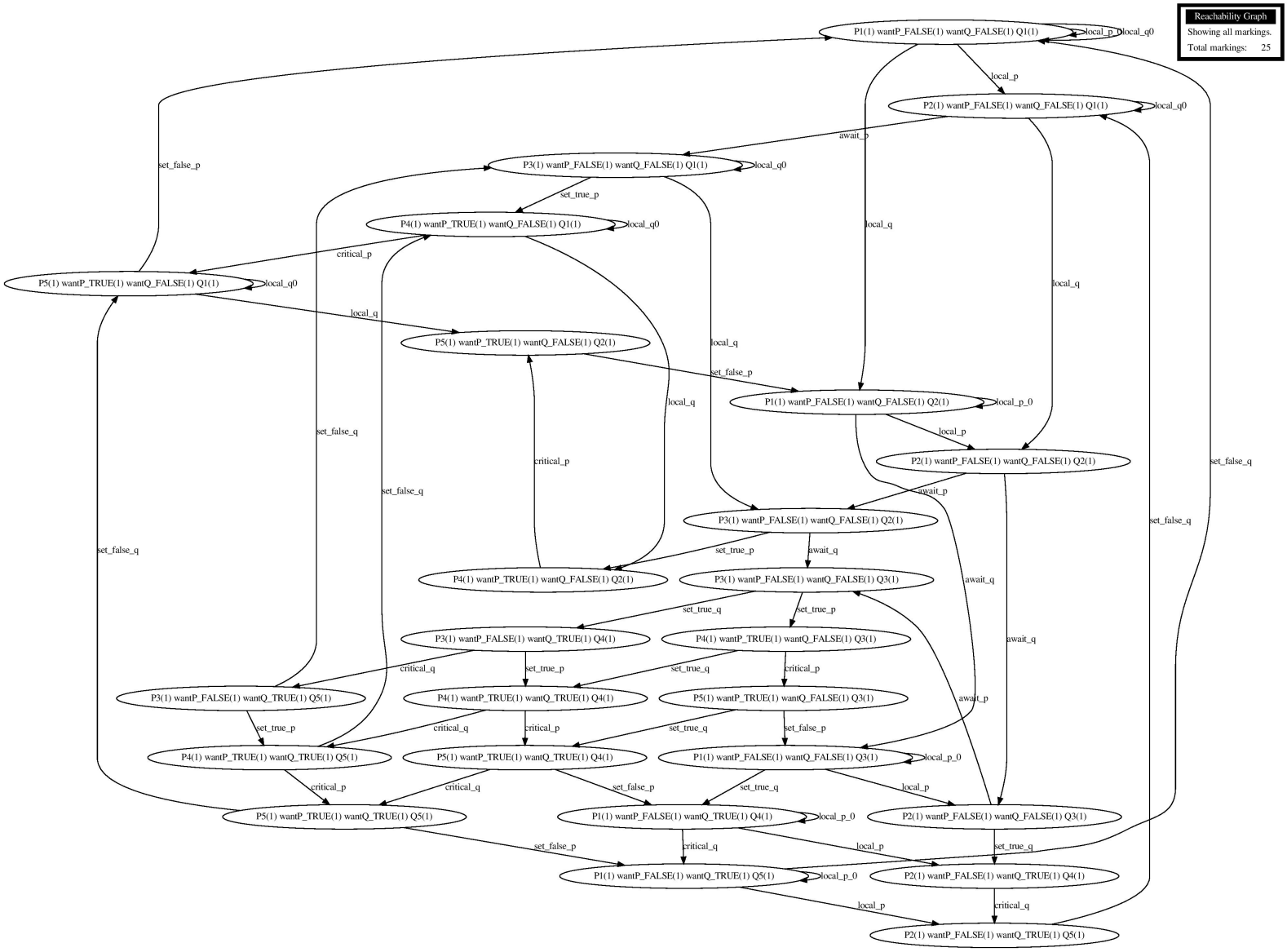
L'algoritmo 3.6 modellato secondo NuSMV e GreatSPN mostra che non c'è la mutua esclusione. Questo viene evidenziato anche dal fatto che i nodi del Reachability Graph corrispondo al prodotto cartesiano P×Q. Anche il Derivation Graph del modello in algebra dei processi ha 25 nodi ed è equivalente al Reachability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

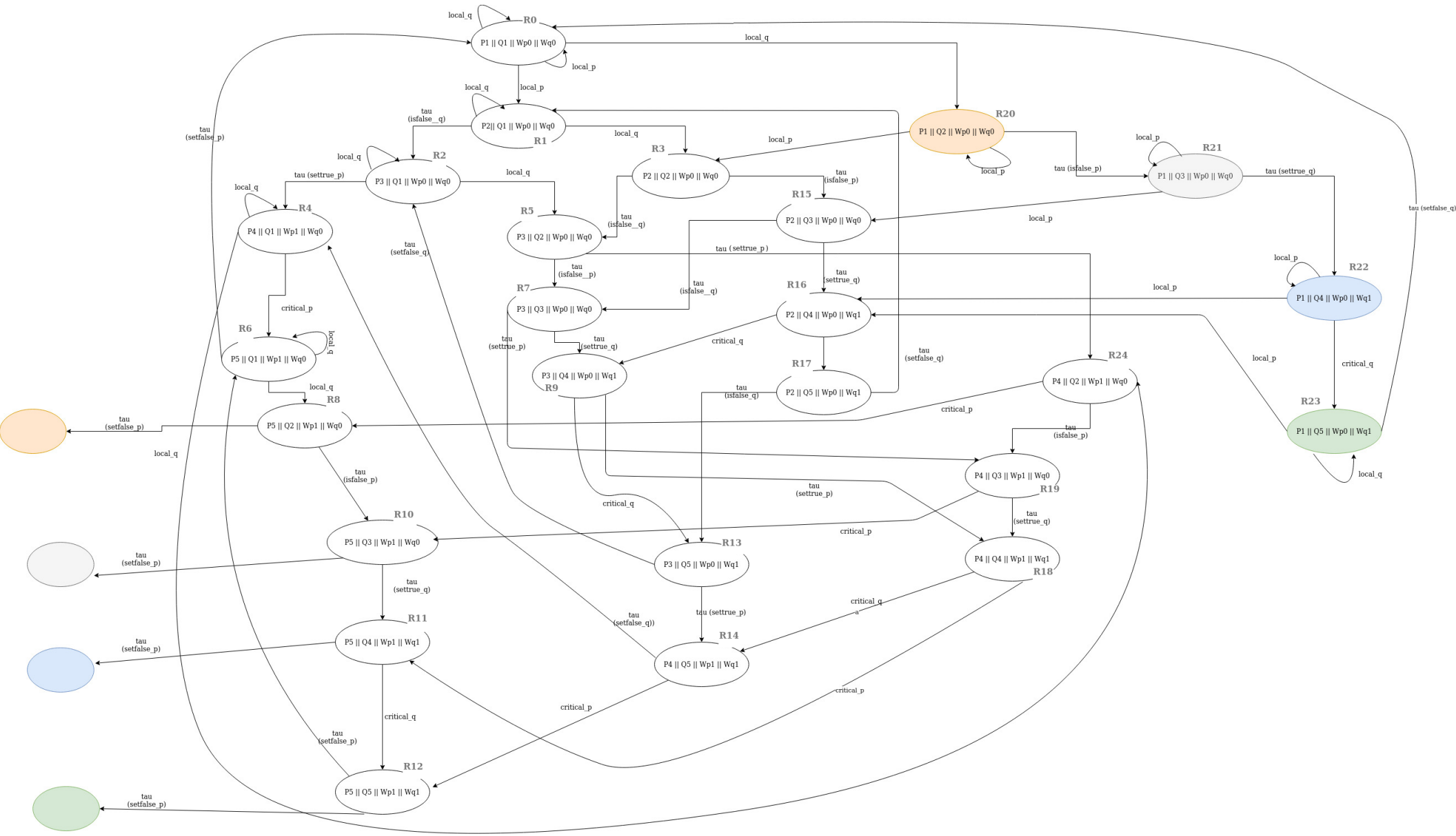
$$
\begin{aligned}
&\text{System} = \{(P_1 \parallel Q_1) \parallel (\text{Want}_{p0} \parallel \text{Want}_{q0})\} \;/\; \text{Sync} \\
&\text{Sync} = \{\ \text{isTrue}_p,\ \text{isFalse}_p,\ \text{setTrue}_p,\ \text{setFalse}_p, \\
&\qquad \text{isTrue}_q,\ \text{isFalse}_q,\ \text{setTrue}_q,\ \text{setFalse}_q\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

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

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

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


Reachability Graph
Showing all markings.
Total markings: 25
P1(1) wantP_FALSE(1) wantQ_FALSE(1) Q1(1)
local_p
local_q0
P2(1) wantP_FALSE(1) wantQ_FALSE(1) Q1(1)
P3(1) wantP_FALSE(1) wantQ_FALSE(1) Q1(1)
P4(1) wantP_TRUE(1) wantQ_FALSE(1) Q1(1)
P5(1) wantP_TRUE(1) wantQ_FALSE(1) Q1(1)
P5(1) wantP_TRUE(1) wantQ_FALSE(1) Q2(1)
P1(1) wantP_FALSE(1) wantQ_FALSE(1) Q2(1)
local_p_0
P2(1) wantP_FALSE(1) wantQ_FALSE(1) Q2(1)
P3(1) wantP_FALSE(1) wantQ_FALSE(1) Q2(1)
P4(1) wantP_TRUE(1) wantQ_FALSE(1) Q2(1)
P3(1) wantP_FALSE(1) wantQ_FALSE(1) Q3(1)
P3(1) wantP_FALSE(1) wantQ_TRUE(1) Q4(1)
P4(1) wantP_TRUE(1) wantQ_FALSE(1) Q3(1)
P3(1) wantP_FALSE(1) wantQ_TRUE(1) Q5(1)
P4(1) wantP_TRUE(1) wantQ_TRUE(1) Q4(1)
P5(1) wantP_TRUE(1) wantQ_FALSE(1) Q3(1)
P4(1) wantP_TRUE(1) wantQ_TRUE(1) Q5(1)
P5(1) wantP_TRUE(1) wantQ_TRUE(1) Q4(1)
P1(1) wantP_FALSE(1) wantQ_FALSE(1) Q3(1)
P5(1) wantP_TRUE(1) wantQ_TRUE(1) Q5(1)
P1(1) wantP_FALSE(1) wantQ_TRUE(1) Q4(1)
P2(1) wantP_FALSE(1) wantQ_FALSE(1) Q3(1)
P1(1) wantP_FALSE(1) wantQ_TRUE(1) Q5(1)
P2(1) wantP_FALSE(1) wantQ_TRUE(1) Q4(1)
P2(1) wantP_FALSE(1) wantQ_TRUE(1) Q5(1)
local_p
local_q
local_q0
local_p_0
await_p
await_q
set_true_p
set_false_p
set_true_q
set_false_q
critical_p
critical_q


R0
P1 || Q1 || Wp0 || Wq0
R1
P2|| Q1 || Wp0 || Wq0
R2
P3 || Q1 || Wp0 || Wq0
R3
P2 || Q2 || Wp0 || Wq0
R4
P4 || Q1 || Wp1 || Wq0
R5
P3 || Q2 || Wp0 || Wq0
R6
P5 || Q1 || Wp1 || Wq0
R7
P3 || Q3 || Wp0 || Wq0
R8
P5 || Q2 || Wp1 || Wq0
R9
P3 || Q4 || Wp0 || Wq1
R10
P5 || Q3 || Wp1 || Wq0
R11
P5 || Q4 || Wp1 || Wq1
R12
P5 || Q5 || Wp1 || Wq1
R13
P3 || Q5 || Wp0 || Wq1
R14
P4 || Q5 || Wp1 || Wq1
R15
P2 || Q3 || Wp0 || Wq0
R16
P2 || Q4 || Wp0 || Wq1
R17
P2 || Q5 || Wp0 || Wq1
R18
P4 || Q4 || Wp1 || Wq1
R19
P4 || Q3 || Wp1 || Wq0
R20
P1 || Q2 || Wp0 || Wq0
R21
P1 || Q3 || Wp0 || Wq0
R22
P1 || Q4 || Wp0 || Wq1
R23
P1 || Q5 || Wp0 || Wq1
R24
P4 || Q2 || Wp1 || Wq0
local_q
local_p
tau (setfalse_p)
tau (isfalse__q)
tau (settrue_p)
tau (isfalse__p)
tau (settrue_q)
tau (isfalse_q)
tau (setfalse_q)
tau (settrue_p)
tau (isfalse_p)
critical_p
critical_q

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

$$\mathrm{AG}(\mathrm{w}_p \rightarrow \mathrm{AF}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q})$$

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

AG ($\mathrm{w}_p \rightarrow$ EF (c_p || c_q))
AG(#await_P == 1 -> EF(#critical_Q==1 || #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)

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

LTLSPEC -- no 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 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, 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{} #critical_P == 1))
```

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

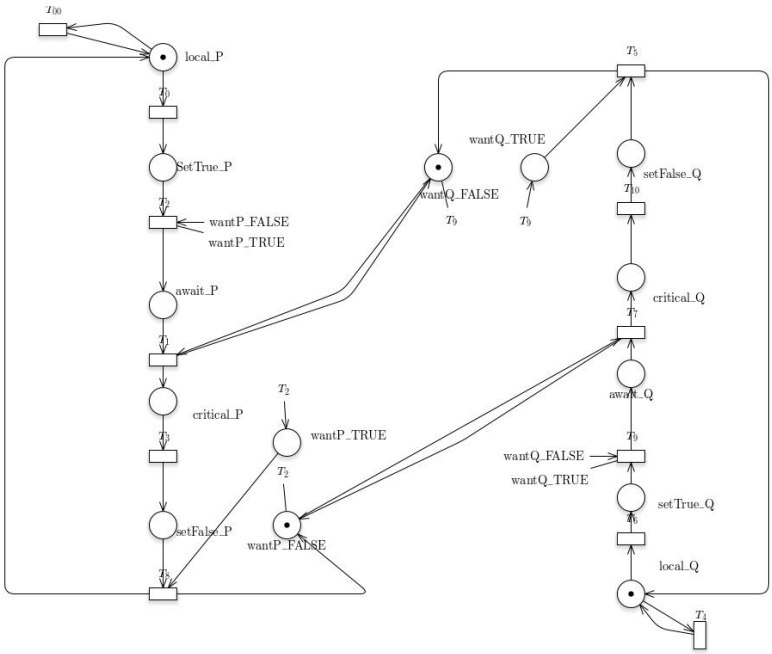


Figure 2: Rete 3.8

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

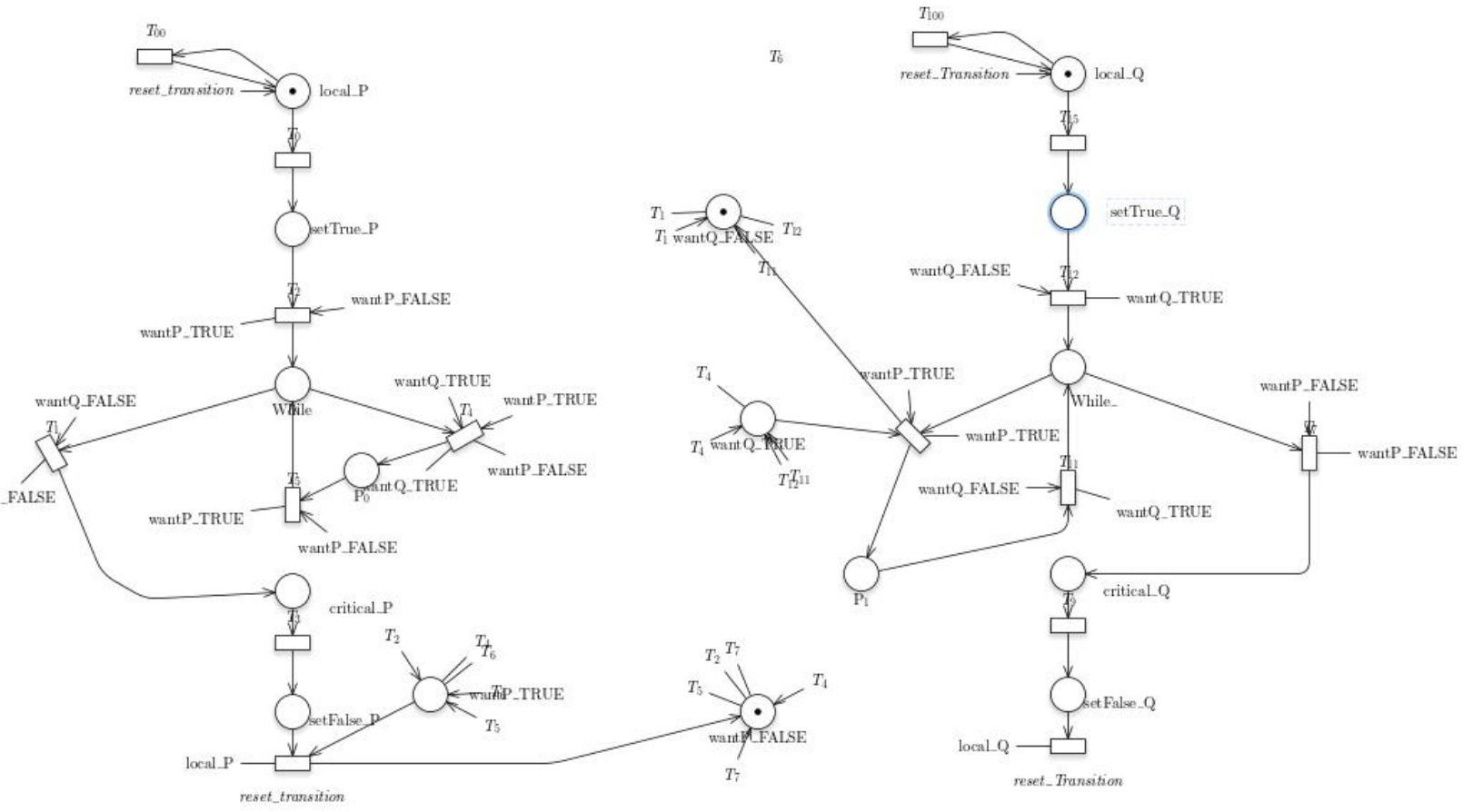
Il codice utilizzato per le proprietà CTL è il seguente:

```
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.


reset_transition
local_P
setTrue_P
wantP_FALSE
wantP_TRUE
While
wantQ_TRUE
wantQ_FALSE
wantP_TRUE
wantP_FALSE
wantQ_TRUE
wantP_TRUE
wantP_FALSE
critical_P
setFalse_P
local_P
reset_transition
reset_Transition
local_Q
setTrue_Q
wantQ_FALSE
wantQ_TRUE
wantP_TRUE
While_
wantP_FALSE
wantQ_FALSE
wantQ_TRUE
wantP_FALSE
critical_Q
setFalse_Q
local_Q
reset_Transition

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;
```

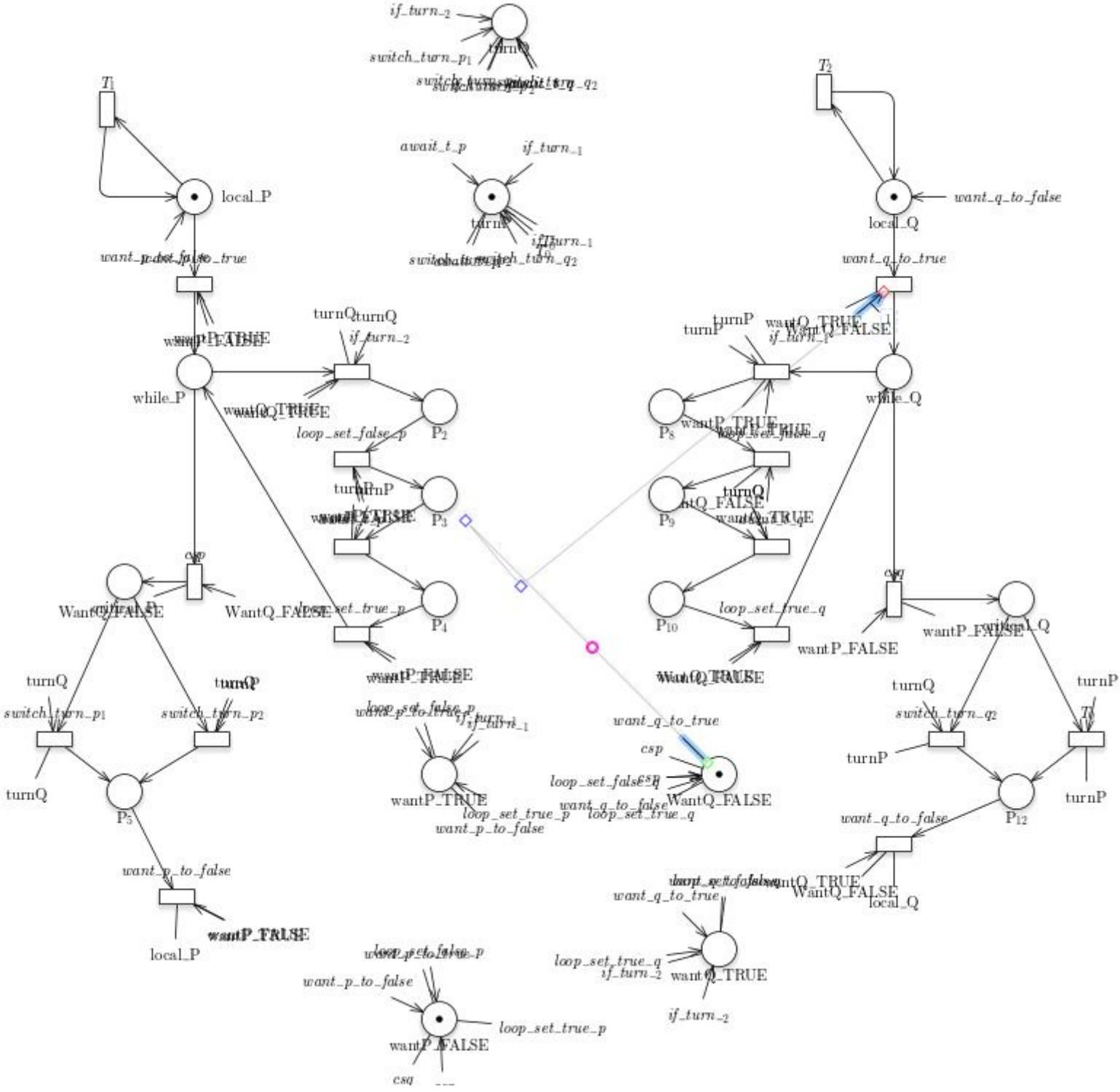
```
    init(wantP) := FALSE;
    init(wantQ) := FALSE;

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))
```


if_turn_2
switch_turn_p1
T1
local_P
await_t_p
if_turn_1
want_p_set_false_to_true
turnQ
while_P
loop_set_false_p
P2
P3
csp
WantQ_FALSE
loop_set_true_p
P4
switch_turn_p2
P5
want_p_to_false
wantP_TRUE
T2
local_Q
want_q_to_false
want_q_to_true
turnP
while_Q
P8
P9
P10
wantP_FALSE
switch_turn_q2
P12
loop_set_false_q
loop_set_true_q
want_q_to_false
wantQ_TRUE
wantQ_FALSE
if_turn_2

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

$$\mathrm{AG}((\mathrm{l}_p \mid\mid \mathrm{l}_\mathrm{q}) \rightarrow \mathrm{AF}(\mathrm{c}_p \vee \mathrm{c}_\mathrm{q})$$

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

AG ($\mathrm{w}_p \rightarrow$ EF ($\mathrm{c}_p \mid\mid \mathrm{c}_\mathrm{q}$))
AG(#await_P == 1 -> EF(#critical_Q==1 || #critical_P == 1))

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:

$$\mathrm{S} = \{\mathrm{local}_p, \mathrm{critical}_p, \mathrm{local}_\mathrm{q}, \mathrm{critical}_\mathrm{q}\} \cup \{\tau\}$$

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 $critical_p$

{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 $critical_q$

{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 $critical_p$ o l'azione $critical_q$. 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_p, $is\_/q$, set_p, set_/q/ 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:

$$p \sim q \rightarrow p \backsimeq_T q$$

ma non viceversa

$$p \sim q \nrightarrow p \trianglelefteq_T q$$

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.

$$s = \ldots \text{critical}_p \text{critical}_q \ldots$$
$$s = \ldots \text{critical}_q \text{critical}_p \ldots$$

Questo ci permette di dire che non c'è trace equivalence 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.

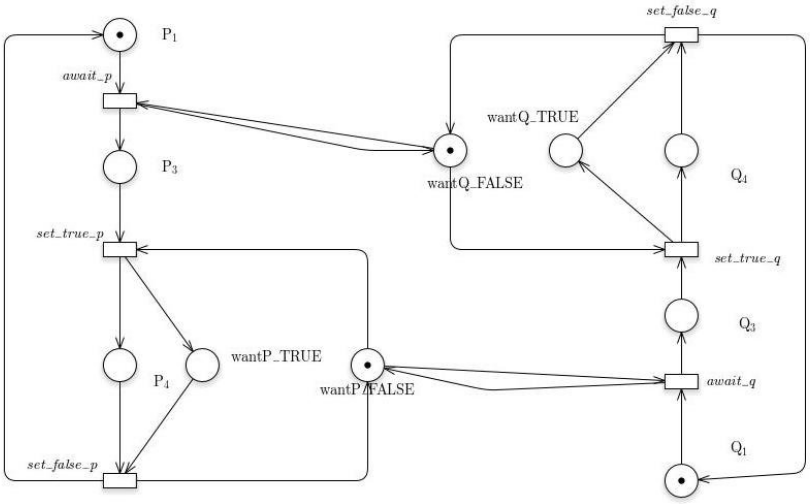


Figure 3: Riduzione rete 3.6

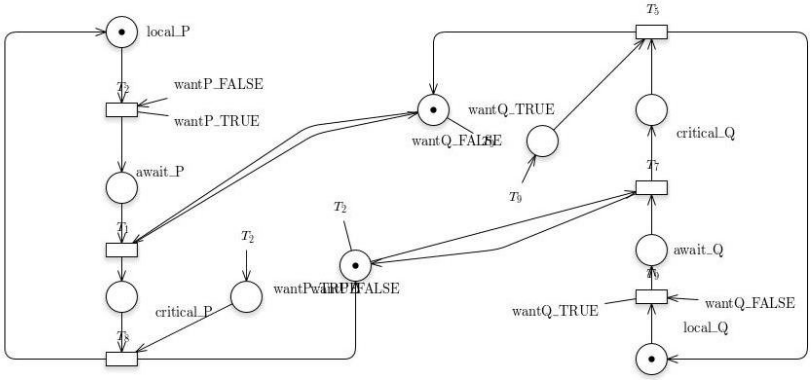


Figure 4: Riduzione rete 3.8