Relazione Esercizio Model Checking

Francesco Gallà, francesco.galla@edu.unito.it

1 Descrizione

Di seguito viene effettuata la valutazione di correttezza per alcuni algoritmi di mutua esclusione, mostrati in ordine di complessità. Per ognuno si mostrano il modello in NuSMV, GreatSPN e algebra dei processi. In ogni modello sono state controllate le tre proprietà essenziali, qui riportate:

1. Mutua esclusione
2. Assenza di Deadlock
3. Assenza di Starvation Individuale

2 Esercizio 3.2

Algorithm 3.2: First attempt

integer turn ← 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 ← 2	q4: turn ← 1

Figure 1: Descrizione dell'algoritmo

I modelli di seguito riportati utilizzano le seguenti specifiche per controllare le 3 proprietà di cui sopra (espresse in CTL):

1. AG !(p.state = critical & q.state = critical) **True**
2. AG (p.state = enter $->$ AF (p.state = critical | q.state = critical)) **False**
3. AG (p.state = enter $->$ AF p.state = critical) **False**

Nota: le proprietà 2 *e* 3 *sono equivalenti sostituendo q a p. Le proprietà in linguaggio LTL sono riportate di seguito.*

2.1 NuSMV

Il modello NuSMV implementa l'algoritmo più triviale per la mutua esclusione, ossia l'alternanza di due processi sull'accesso a una sezione critica in base a una variabile booleana *turn*. Sono stati utilizzati 4 stati, di cui:

local: Rappresenta l'esecuzione non critica e non forza il progresso

enter: Obbliga l'attesa per l'accesso, controllando la variabile *turn*

critical: Rappresenta la sezione critica

exit: Cambia il valore di *turn* per garantire l'accesso all'altro processo ed esce dalla SC.

```
MODULE main
VAR
  turn : 1..2;
  p : process user(turn, 1, 2);
  q : process user(turn, 2, 1);
ASSIGN
  init(turn) := 1;
SPEC -- PROGRESS
  AG (p.state = local -> EF p.state = enter )
SPEC -- MUTUAL EXCLUSION
  AG !(p.state = critical & q.state = critical)
SPEC -- DEADLOCK
  AG (p.state = enter -> AF (p.state = critical | q.state = critical))
SPEC -- INDIVIDUAL STARVATION
  AG (p.state = enter -> AF p.state = critical)

MODULE user(turnloc, myproc, otherproc)
VAR
  state : {local,enter,critical,exit};
ASSIGN
  init(state) := local;
  next(state) :=
    case
      state = local : {local, enter};
      state = enter & turnloc = myproc : critical;
      state = critical : exit;
      state = exit : local;
      TRUE : state;
    esac;
  next(turnloc) :=
    case
      state = exit : otherproc;
      TRUE : turnloc;
    esac;
FAIRNESS
  running
```

Il modello NuSmv model per l'algoritmo 3.2.

2.2 Rete di Petri

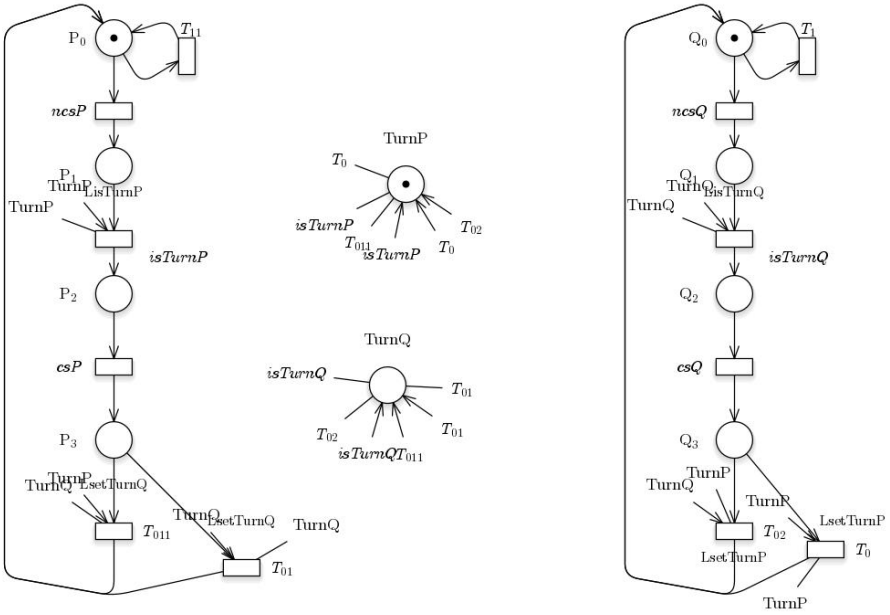

Figure 2: Modello P/T della rete 3.2

2.3 Mutua Esclusione

2.3.1 NuSmv

CTL: **True**

```
AG !(p.state = critical & q.state = critical)
```

LTL: **True**

```
G !(p.state = critical & q.state = critical)
```

2.3.2 GreatSPN

CTL: **True**

```
AG !(#P3 == 1 && #Q3 == 1)
```

La mutua esclusione è rispettata sia in NuSMV che in GreatSPN, in quanto non esiste un caso in cui entrambi i processi siano contemporaneamente nella sezione critica. È anche vero però che i processi si aspettano a vicenda nella fase di *enter*, pertanto il loro accesso sarà per forza alternato.

2.4 Assenza di Deadlock

2.4.1 NuSmv

CTL: **False**

```
AG (p.state = enter -> AF (p.state = critical | q.state = critical))
```

LTL: **False**

```
G (p.state = enter -> F (p.state = critical | q.state = critical))
```

2.4.2 GreatSPN

CTL: **False**

```
AG (#P2 == 1 -> AF ( #P3 == 1 || #Q3 == 1))
```

La proprietà di assenza di deadlock non può essere rispettata in quanto i processi si aspettano a vicenda nella fase di *enter*. Essendo inoltre che i processi nella fase *local* non sono obbligati al progresso, se un processo resta in local indefinitamente non permette il proseguimento dell'altro.
Infatti, solo nel caso in cui entrambi i processi richiedono la sezione critica, e quindi sono entrambi nella fase *enter*, si evita il deadlock.

Di seguito si riporta il controesempio alla proprietà.

```
-- specification AG (p.state = enter -> 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 = local
    q.state = local
  -> Input: 1.2 <-
    _process_selector_ = p
    running = FALSE
    q.running = FALSE
    p.running = TRUE
  -> State: 1.2 <-
    p.state = enter
```

```
-> Input: 1.3 <-
-> State: 1.3 <-
  p.state = critical
-> Input: 1.4 <-
-> State: 1.4 <-
  p.state = exit
-> Input: 1.5 <-
-> State: 1.5 <-
  turn = 2
  p.state = local
-> Input: 1.6 <-
-- Loop starts here
-> State: 1.6 <-
  p.state = enter
-> 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 <-
```

Il controesempio fornito da NuSmv per la proprietà 2.

```
--- AG ( #P2==1 -> AF (#P3 == 1 || #Q3 == 1 )) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.000777 sec
         Evaluation time: 0.000777 sec

Generated counter-example:
================================== Trace ===================================
Initial state is: P0(1), TurnP(1), Q0(1)
Initial state satisfies: E F (not ((not (P2 = 1)) or (not E G (not ((P3 = 1) or (Q3 = 1)))))).

1: P0(1), TurnP(1), Q0(1)
  State 1. satisfies: ((not (P2 = 1)) or (not E G (not ((P3 = 1) or (Q3 = 1))))).

  1.1: P0(1), TurnP(1), Q0(1)
    State 1.1. does not satisfy: (P2 = 1).

2: P1(1), TurnP(1), Q0(1)
  State 2. satisfies: ((not (P2 = 1)) or (not E G (not ((P3 = 1) or (Q3 = 1))))).

  2.1: P1(1), TurnP(1), Q0(1)
    State 2.1. does not satisfy: (P2 = 1).

3: P2(1), TurnP(1), Q0(1)
  State 3. does not satisfy: ((not (P2 = 1)) or (not E G (not ((P3 = 1) or (Q3 = 1))))).

  3.1: P2(1), TurnP(1), Q0(1)
    State 3.1.L. satisfies: (P2 = 1).

    State 3.1.R. satisfies: E G (not ((P3 = 1) or (Q3 = 1))). Start of loop.

    3.1.R.1: P2(1), TurnP(1), Q0(1)
      State 3.1.R.1. does not satisfy: ((P3 = 1) or (Q3 = 1)).

      3.1.R.1.1: P2(1), TurnP(1), Q0(1)
        State 3.1.R.1.1.L. does not satisfy: (P3 = 1).

        State 3.1.R.1.1.R. does not satisfy: (Q3 = 1).

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

Figure 3: Controesempio fornito da GreatSPN per la proprietà 2.

2.5 Assenza di Starvation Individuale

CTL: **False**

```
AG (p.state = enter -> AF (p.state = critical))
```

LTL: **False**

```
G (p.state = enter -> F (p.state = critical))
```

2.5.1 GreatSPN

CTL: **False**

```
AG (#P2 == 1 -> AF ( #P3 == 1 ))
```

L'assenza di starvation individuale non è rispettata in quanto la proprietà di assenza di deadlock non lo è. Di seguito si riporta il controesempio alla proprietà.

```
-- specification AG (p.state = enter -> AF p.state = critical) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 2.1 <-
    turn = 1
    p.state = local
    q.state = local
  -> Input: 2.2 <-
    _process_selector_ = p
    running = FALSE
    q.running = FALSE
    p.running = TRUE
  -> State: 2.2 <-
    p.state = enter
  -> Input: 2.3 <-
  -> State: 2.3 <-
    p.state = critical
  -> Input: 2.4 <-
  -> State: 2.4 <-
    p.state = exit
  -> Input: 2.5 <-
  -> State: 2.5 <-
    turn = 2
    p.state = local
  -> Input: 2.6 <-
  -- Loop starts here
  -> State: 2.6 <-
    p.state = enter
  -> Input: 2.7 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
  -- Loop starts here
  -> State: 2.7 <-
  -> Input: 2.8 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
  -- Loop starts here
  -> State: 2.8 <-
  -> Input: 2.9 <-
    _process_selector_ = main
    running = TRUE
    p.running = FALSE
  -> State: 2.9 <-
```

Il controesempio fornito da NuSmv per la proprietà 3.

```
--- AG ( #P2==1 -> AF (#P3 == 1)) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.00046 sec
         Evaluation time: 0.00046 sec

Generated counter-example:
================================== Trace ==================================
Initial state is: P0(1), TurnP(1), Q0(1)
Initial state satisfies: E F (not ((not (P2 = 1)) or (not E G (not (P3 = 1))))).

1: P0(1), TurnP(1), Q0(1)
  State 1. satisfies: ((not (P2 = 1)) or (not E G (not (P3 = 1)))).

  1.1: P0(1), TurnP(1), Q0(1)
    State 1.1. does not satisfy: (P2 = 1).

2: P1(1), TurnP(1), Q0(1)
  State 2. satisfies: ((not (P2 = 1)) or (not E G (not (P3 = 1)))).

  2.1: P1(1), TurnP(1), Q0(1)
    State 2.1. does not satisfy: (P2 = 1).

3: P2(1), TurnP(1), Q0(1)
  State 3. does not satisfy: ((not (P2 = 1)) or (not E G (not (P3 = 1)))).

  3.1: P2(1), TurnP(1), Q0(1)
    State 3.1.L. satisfies: (P2 = 1).

    State 3.1.R. satisfies: E G (not (P3 = 1)). Start of loop.

    3.1.R.1: P2(1), TurnP(1), Q0(1)
      State 3.1.R.1. does not satisfy: (P3 = 1).

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

Figure 4: Controesempio fornito da GreatSPN per la proprietà 3.

2.6 Algebra dei Processi

Si presenta un modello in algebra dei processi per l'algoritmo 3.2. La nomenclatura vuole richiamare i posti e le transizioni definiti nella rete P/T, per agevolare il confronto tra il Derivation Graph in CCS e il Reachability Graph in GreatSPN.

Come si nota, il DG e il RG hanno lo stesso numero di nodi e archi corrispondenti. Per questo possiamo affermare l'equivalenza del modello CCS con il modello P/T.

```
Actions = {isTurnP, isTurnQ, setP, setQ, locP, locQ, criticalP,
    criticalQ}

TurnP = setP.TurnP + setQ.TurnQ + isTurnP.TurnP
TurnQ = setQ.TurnQ + setP.TurnP + isTurnQ.TurnQ

P1 = locP.P1 + locP.P2
P2 = [isTurnP].P3
P3 = criticalP.P4
P4 = [setQ].P1

Q1 = locQ.Q1 + locQ.Q2
Q2 = [isTurnQ].Q3
Q3 = criticalP.Q4
Q4 = [setP].Q1

Sync = {isTurnP, isTurnQ, setP, setQ}
Sys = {(P1 || Q1) || TurnP} / Sync
```

Modello CCS per l'algoritmo 3.2. *Nota: un'azione contornata da '[]' è considerata un'azione in output, tutte le altre sono considerate in input.*

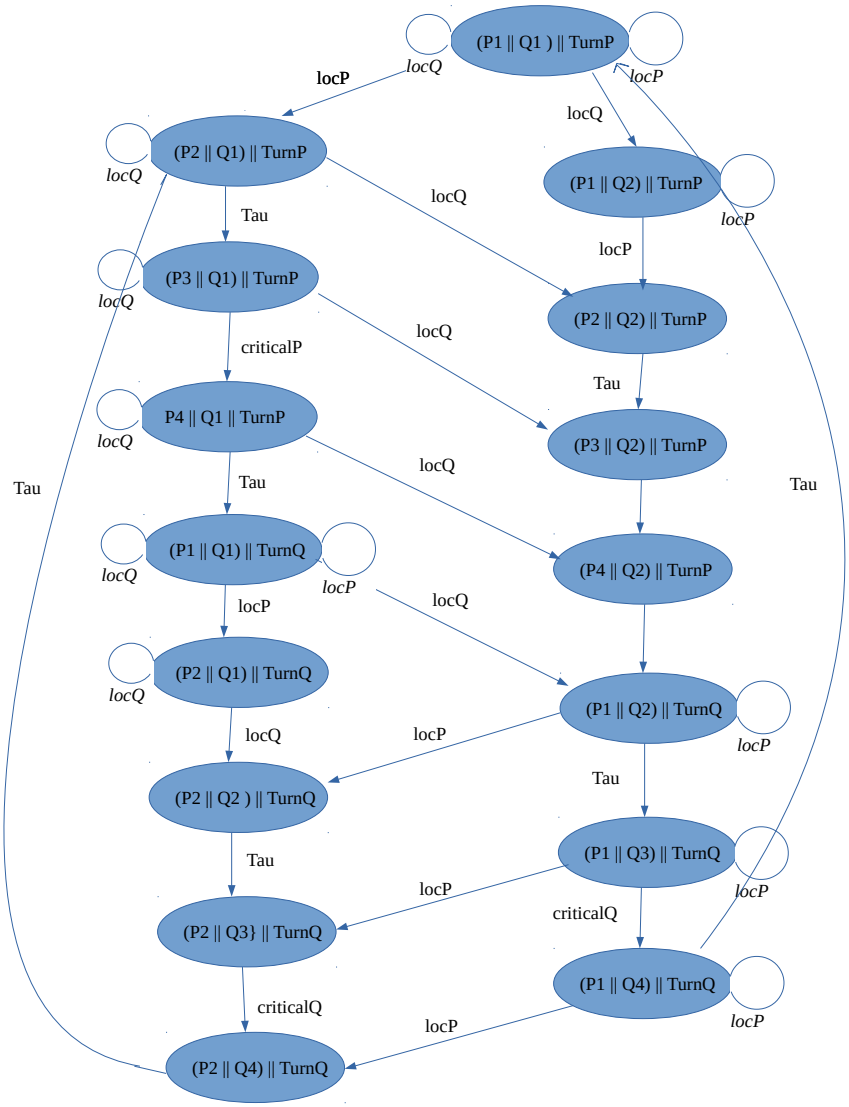

Figure 5: Derivation Graph per la rete 3.2

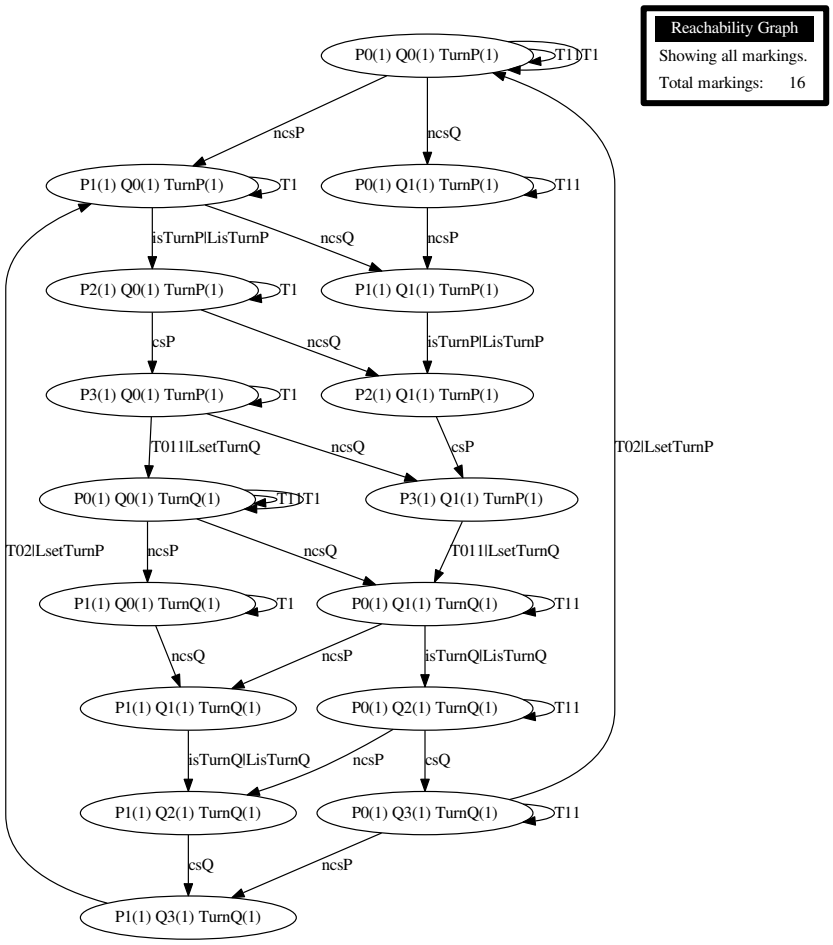

Figure 6: Reachability Graph di GreatSPN per la rete 3.2

3 Esercizio 3.5

Algorithm 3.5: First attempt (abbreviated)	
integer turn ← 1	
p	**q**
loop forever	loop forever
p1: await turn = 1	q1: await turn = 2
p2: turn ← 2	q2: turn ← 1

Figure 7: Descrizione dell'algoritmo

I modelli di seguito riportati utilizzano le seguenti specifiche per controllare le 3 proprietà di cui sopra (espresse in CTL):

1. AG !(p.state = exit & q.state = exit) **True**
2. AG (p.state = enter − > AF (p.state = exit | q.state = exit)) **False**
3. AG (p.state = enter − > AF p.state = exit) **False**

Nota: le proprietà 2 *e* 3 *sono equivalenti sostituendo q a p. Le proprietà LTL si ottengono rimuovendo gli operatori di universalità e esistenzialità.*

3.1 NuSMV

Il modello NuSMV implementa l'algoritmo semplificato per la mutua esclusione, ossia l'alternanza di due processi sull'accesso a una sezione critica in base a una variabile booleana *turn*. Sono stati utilizzati 2 stati, di cui:

enter: Obbliga l'attesa per l'accesso, controllando la variabile *turn*. In questo caso il progresso non è forzato: il processo può rimanere in attesa per un tempo indefinito.

exit: Cambia il valore di *turn* per garantire l'accesso all'altro processo e ritorna a *enter*.

Questo modello si presenta come una versione semplificata dell'algoritmo 3.2 (sono infatti stati rimossi gli stati di sezione non critica e critica, mantenendo solo la richiesta in *enter* e il passaggio del turno in *exit*). Pertanto, i risultati sono gli stessi: la mutua esclusione è rispettata, mentre l'assenza di deadlock e di starvation individuale non lo sono. Anche in questo caso i processi hanno un'alternanza all'accesso forzata dall'assegnazione in *exit* del turno al processo opposto dopo ogni accesso alla SC.

Si riportano di seguito i modelli NuSmv e GreatSPN. Le tracce di esecuzione sono simili all'esercizio 3.2.

```
MODULE main
VAR
  turn : 1..2;
  p : process user(turn, 1, 2);
  q : process user(turn, 2, 1);
ASSIGN
  init(turn) := 1;
SPEC -- MUTUAL EXCLUSION
  AG !(p.state = exit & q.state = exit)
SPEC -- DEADLOCK
  AG (p.state = enter -> AF (p.state = exit | q.state = exit))
SPEC -- INDIVIDUAL STARVATION
  AG (p.state = enter -> AF p.state = exit)

MODULE user(turnloc, myproc, otherproc)
VAR
  state : {enter,exit};
ASSIGN
  init(state) := enter;
  next(state) :=
    case
      state = enter & turnloc = myproc : {enter,exit};
      state = exit : enter;
      TRUE : state;
    esac;
  next(turnloc) :=
    case
      state = exit : otherproc;
      TRUE : turnloc;
    esac;
FAIRNESS
  running
```

Il modello NuSmv model per l'algoritmo 3.5.

3.2 Rete di Petri

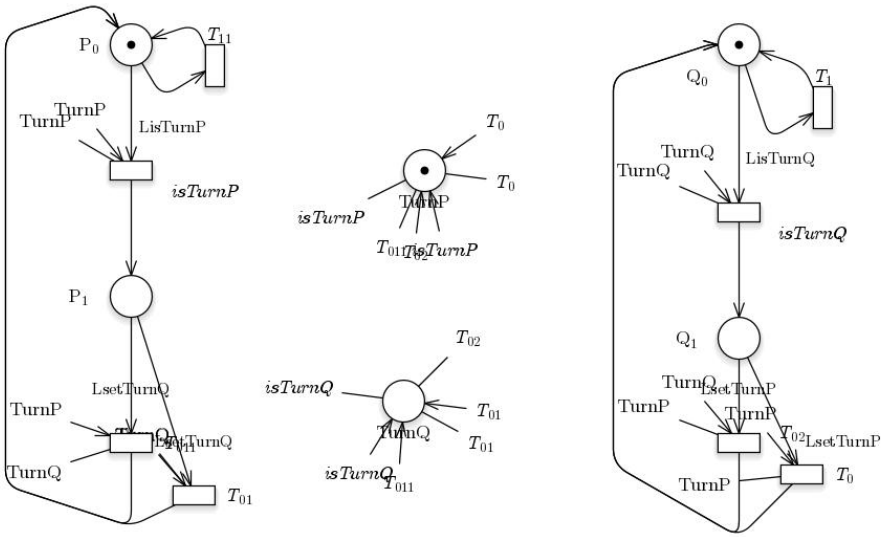

Figure 8: Modello P/T della rete 3.5

3.3 Algebra dei Processi

Si presenta un modello in algebra dei processi per l'algoritmo 3.5. La nomenclatura vuole richiamare i posti e le transizioni definiti nella rete P/T, per agevolare il confronto tra il Derivation Graph in CCS e il Reachability Graph in GreatSPN.

Come si nota, il DG e il RG hanno lo stesso numero di nodi e archi corrispondenti. Per questo possiamo affermare l'equivalenza del modello CCS con il modello P/T.

```
Actions = {isTurnP, isTurnQ, setP, setQ, waitP, waitQ}

TurnP = setP.TurnP + setQ.TurnQ + isTurnP.TurnP
TurnQ = setQ.TurnQ + setP.TurnP + isTurnQ.TurnQ

P1 = [isTurnP].P2 + waitP.P1
P2 = [setQ].P1

Q1 = [isTurnQ].Q2 + waitQ.Q1
Q2 = [setP].Q1


Sync = {isTurnP, isTurnQ, setP, setQ}
Sys = {(P1 || Q1) || TurnP} / Sync
```

Modello CCS per l'algoritmo 3.5. *Nota: un'azione contornata da '[]' è considerata un'azione in output, tutte le altre sono considerate in input.*

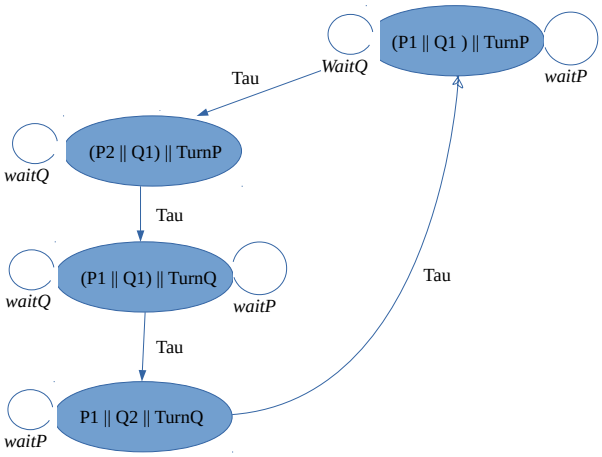

Figure 9: Derivation Graph per la rete 3.5

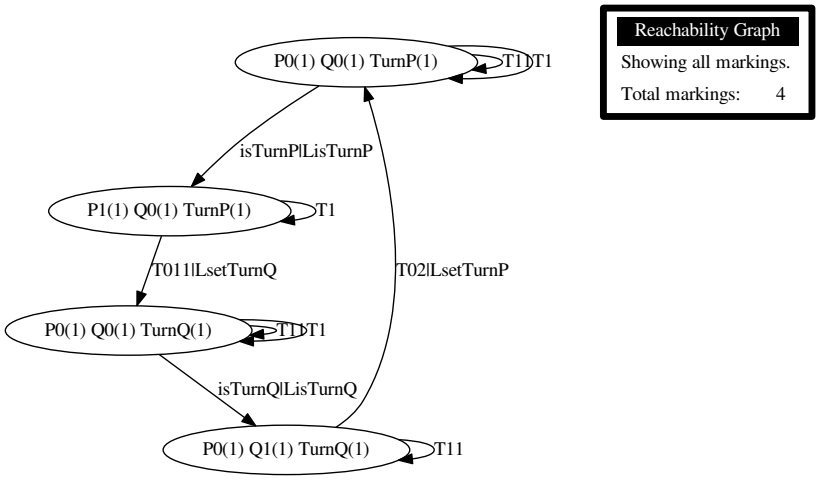

Figure 10: Reachability Graph di GreatSPN per la rete 3.5

4 Esercizio 3.6

Algorithm 3.6: Second attempt	
boolean wantp ← false, wantq ← 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 ← true	q3: wantq ← true
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Figure 11: Descrizione dell'algoritmo

I modelli di seguito riportati utilizzano le seguenti specifiche per controllare le 3 proprietà di cui sopra (espresse in CTL):

1. AG !(p.state = critical & q.state = critical) **False**
2. AG (p.state = enter − > AF (p.state = critical | q.state = critical)) **True**
3. AG (p.state = enter − > AF p.state = critical) **False**

Nota: le proprietà 2 *e* 3 *sono equivalenti sostituendo q a p. Le proprietà LTL si ottengono rimuovendo gli operatori di universalità e esistenzialità.*

4.1 NuSMV

Il modello NuSMV implementa il primo algoritmo per la mutua esclusione basato su due variabili anzichè una condivisa. Sono stati utilizzati 5 stati, di cui:

local: Sezione non critica.

enter: Obbliga l'attesa per l'accesso, controllando la variabile *want* opposta. In questo caso il progresso non è forzato: il processo può rimanere in attesa per un tempo indefinito.

set: Ottenuto l'accesso, assegna alla variabile il valore $TRUE$.

critical: Sezione critica.

exit: Cambia il valore di *want* a $FALSE$ e ritorna a *local*.

Il modello utilizza due variabili, *wantp* e *wantq*, una per ogni processo. Non si utilizza più la variabile *turn*, ma i processi conoscono solo lo stato delle due variabili.

Si riportano di seguito i modelli NuSmv e GreatSPN.

```
MODULE main
VAR
  wp : boolean;
  wq : boolean;
  p : process user(wp,wq);
  q : process user(wq,wp);
ASSIGN
  init(wp) := FALSE;
  init(wq) := FALSE;

SPEC -- PROGRESS
  AG (p.state = local -> EF p.state = enter )
SPEC -- MUTUAL EXCLUSION
  AG !(p.state = critical & q.state = critical)
SPEC -- DEADLOCK
  AG (p.state = enter -> AF (p.state = critical | q.state = critical))
SPEC -- INDIVIDUAL STARVATION
  AG (p.state = enter -> AF p.state = critical)

MODULE user(wp, wq)
VAR
  state : {local,enter,set,critical,exit};
ASSIGN
  init(state) := local;
  next(state) :=
    case
      state = local : {local, enter};
      state = enter & wq = FALSE : set;
      state = set : critical;
      state = critical : exit;
      state = exit : local;
      TRUE : state;
    esac;
  next(wp) :=
    case
      state = set : TRUE;
      state = exit : FALSE;
      TRUE : wp;
    esac;
FAIRNESS
  running
```

Il modello NuSmv per l'algoritmo 3.6.

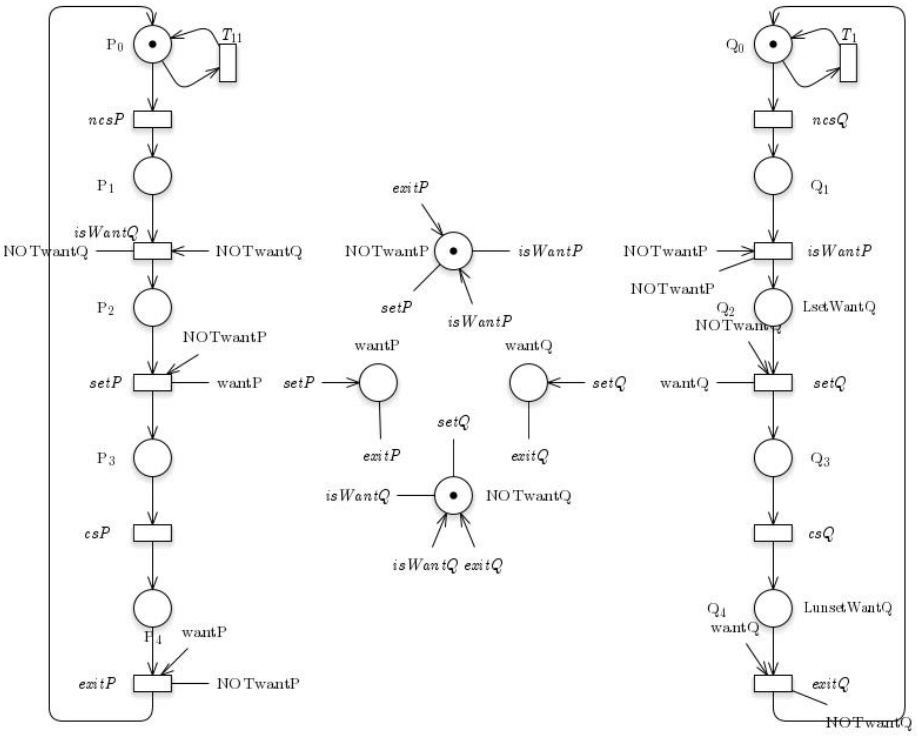

Figure 12: Modello GreatSPN per l'algoritmo 3.6

4.2 Mutua Esclusione

4.2.1 NuSmv

CTL: **False**

```
AG !(p.state = critical & q.state = critical)
```

LTL: **False**

```
G !(p.state = critical & q.state = critical)
```

4.2.2 GreatSPN

CTL: **False**

```
AG !(#P4 == 1 && #Q4 == 1)
```

La mutua esclusione non è rispettata dall'algoritmo 3.6, come si vede dall'esecuzione di controesempio di NuSmv e GreatSPN. Se i due processi si trovano entrambi in stato di *enter*: $wantp == wantq == FALSE$, pertanto è possibile che riescano entrambi a passare in stato di *set*.

Si riportano di seguito i controesempi.

```
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 <-
    wp = FALSE
    wq = 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 = enter
  -> Input: 1.3 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
  -> State: 1.3 <-
    q.state = enter
  -> Input: 1.4 <-
  -> State: 1.4 <-
    q.state = set
  -> Input: 1.5 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
  -> State: 1.5 <-
    p.state = set
  -> Input: 1.6 <-
  -> State: 1.6 <-
    wp = TRUE
    p.state = critical
  -> Input: 1.7 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
  -> State: 1.7 <-
    wq = TRUE
    q.state = critical
```

```
Generated counter-example:
================================== Trace ==================================
Initial state is: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
Initial state satisfies: E F (not (not ((P3 = 1) and (Q3 = 1)))).

1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
  State 1. does not satisfy: ((P3 = 1) and (Q3 = 1)).

  1.1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
    State 1.1. does not satisfy: (P3 = 1).

2: P0(1), NOTwantP(1), NOTwantQ(1), Q1(1)
  State 2. does not satisfy: ((P3 = 1) and (Q3 = 1)).

  2.1: P0(1), NOTwantP(1), NOTwantQ(1), Q1(1)
    State 2.1. does not satisfy: (P3 = 1).

3: P0(1), NOTwantP(1), NOTwantQ(1), Q2(1)
  State 3. does not satisfy: ((P3 = 1) and (Q3 = 1)).

  3.1: P0(1), NOTwantP(1), NOTwantQ(1), Q2(1)
    State 3.1. does not satisfy: (P3 = 1).

4: P1(1), NOTwantP(1), NOTwantQ(1), Q2(1)
  State 4. does not satisfy: ((P3 = 1) and (Q3 = 1)).

  4.1: P1(1), NOTwantP(1), NOTwantQ(1), Q2(1)
    State 4.1. does not satisfy: (P3 = 1).

5: NOTwantP(1), P2(1), NOTwantQ(1), Q2(1)
  State 5. does not satisfy: ((P3 = 1) and (Q3 = 1)).

  5.1: NOTwantP(1), P2(1), NOTwantQ(1), Q2(1)
    State 5.1. does not satisfy: (P3 = 1).

6: P3(1), wantP(1), NOTwantQ(1), Q2(1)
  State 6. does not satisfy: ((P3 = 1) and (Q3 = 1)).

  6.1: P3(1), wantP(1), NOTwantQ(1), Q2(1)
    State 6.1. does not satisfy: (Q3 = 1).

7: P3(1), wantP(1), wantQ(1), Q3(1)
  State 7. satisfies: ((P3 = 1) and (Q3 = 1)).

  7.1: P3(1), wantP(1), wantQ(1), Q3(1)
    State 7.1.L. satisfies: (P3 = 1).

    State 7.1.R. satisfies: (Q3 = 1).
```

Figure 13: Controesempio per la mutua esclusione di GreatSPN.

4.3 Assenza di Deadlock

4.3.1 NuSmv

CTL: **True**

```
AG (p.state = enter -> AF (p.state = critical | q.state = critical))
```

LTL: **True**

```
G (p.state = enter -> F (p.state = critical | q.state = critical))
```

4.3.2 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 || #Q4 == 1))
```

La proprietà di assenza di deadlock è vera nel caso di NuSmv, in quanto non è possibile che un processo rimanga in attesa in *enter* per un tempo indefinito: se l'altro processo entra, dovrà per forza uscire e cambiare il valore di *want*, altrimenti la variabile *want* gli permette l'accesso a *set*.

In GreatSPN, la proprietà risulta invece falsa, in quanto è possibile che un processo entri in $P1$ (attesa) ma l'altro esegua continuamente la transizione di non progresso. Possiamo forzare questa proprietà estendendo la definizione di assenza di deadlock, utilizzando quindi la seguente formula CTL:

```
AG (#P1 == 1 -> EF (#P4 == 1 || #Q4 == 1))
```

Che viene valutata **True**.

4.4 Assenza di Starvation Individuale

CTL: **False**

```
AG (p.state = enter -> AF (p.state = critical))
```

LTL: **False**

```
G (p.state = enter -> F (p.state = critical))
```

4.4.1 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 ))
```

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

```
-- specification AG (p.state = enter -> AF p.state = critical) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 2.1 <-
    wp = FALSE
    wq = FALSE
    p.state = local
    q.state = local
  -> Input: 2.2 <-
    _process_selector_ = p
    running = FALSE
    q.running = FALSE
    p.running = TRUE
  -- Loop starts here
  -> State: 2.2 <-
    p.state = enter
  -> Input: 2.3 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
  -> State: 2.3 <-
    q.state = enter
  -> Input: 2.4 <-
  -> State: 2.4 <-
    q.state = set
  -> Input: 2.5 <-
  -> State: 2.5 <-
    wq = TRUE
    q.state = critical
  -> Input: 2.6 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
  -> State: 2.6 <-
  -> Input: 2.7 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
  -> State: 2.7 <-
    q.state = exit
  -> Input: 2.8 <-
  -> State: 2.8 <-
    wq = FALSE
    q.state = local
```

```
--- AG ((#P1 == 1 )-> AF (#P4 == 1)) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.000571 sec
         Evaluation time: 0.000571 sec

Generated counter-example:
================================== Trace ==================================
Initial state is: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
Initial state satisfies: E F (not ((not (P1 = 1)) or (not E G (not (P4 = 1))))).

1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
  State 1. satisfies: ((not (P1 = 1)) or (not E G (not (P4 = 1)))).

  1.1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
    State 1.1. does not satisfy: (P1 = 1).

2: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
  State 2. does not satisfy: ((not (P1 = 1)) or (not E G (not (P4 = 1)))).

  2.1: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
    State 2.1.L. satisfies: (P1 = 1).

    State 2.1.R. satisfies: E G (not (P4 = 1)). Start of loop.

    2.1.R.1: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
      State 2.1.R.1. does not satisfy: (P4 = 1).

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

Figure 14: Controesempio per la starvation individuale di GreatSPN.

5 Esercizio 3.8

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

Figure 15: Descrizione dell'algoritmo

I modelli di seguito riportati utilizzano le seguenti specifiche per controllare le 3 proprietà di cui sopra (espresse in CTL):

1. AG !(p.state = critical & q.state = critical) **True**
2. AG (p.state = set $->$ AF (p.state = critical | q.state = critical)) **False**
3. AG (p.state = set $->$ AF p.state = critical) **False**

Nota: le proprietà 2 *e* 3 *sono equivalenti sostituendo q a p. Le proprietà LTL si ottengono rimuovendo gli operatori di universalità e esistenzialità.*

5.1 NuSMV

Il secondo algoritmo di mutua esclusione basato su due variabili è simile al 3.6 ma inverte gli stati *set* e *enter*, di modo da segnalare la sua intenzione di entrare nella sezione critica prima di mettersi in attesa. Sono stati utilizzati 5 stati, di cui:

local: Sezione non critica.

set: Richiede l'accesso assegnando alla variabile il valore $TRUE$.

enter: Obbliga l'attesa per l'accesso, controllando la variabile *want* opposta. In questo caso il progresso non è forzato: il processo può rimanere in attesa per un tempo indefinito.

critical: Sezione critica.

exit: Cambia il valore di *want* a $FALSE$ e ritorna a *local*.

Il modello utilizza due variabili, *wantp* e *wantq*, una per ogni processo. Non si utilizza più la variabile *turn*, ma i processi conoscono solo lo stato delle due variabili.

Si riportano di seguito i modelli NuSmv e GreatSPN.

```
MODULE main
VAR
  wp : boolean;
  wq : boolean;
  p : process user(wp,wq);
  q : process user(wq,wp);
ASSIGN
  init(wp) := FALSE;
  init(wq) := FALSE;

SPEC -- PROGRESS
  AG (p.state = local -> EF p.state = enter )
SPEC -- MUTUAL EXCLUSION
  AG !(p.state = critical & q.state = critical)
SPEC -- DEADLOCK
  AG (p.state = set -> AF (p.state = critical | q.state = critical))
SPEC -- INDIVIDUAL STARVATION
  AG (p.state = set -> AF p.state = critical)

MODULE user(wp, wq)
VAR
  state : {local,set,enter,critical,exit};
ASSIGN
  init(state) := local;
  next(state) :=
    case
      state = local : {local, set};
      state = set : enter;
      state = enter & wq = FALSE : critical;
      state = critical : exit;
      state = exit : local;
      TRUE : state;
    esac;
  next(wp) :=
    case
      state = set : TRUE;
      state = exit : FALSE;
      TRUE : wp;
    esac;
FAIRNESS
  running
```

Il modello NuSmv per l'algoritmo 3.8.

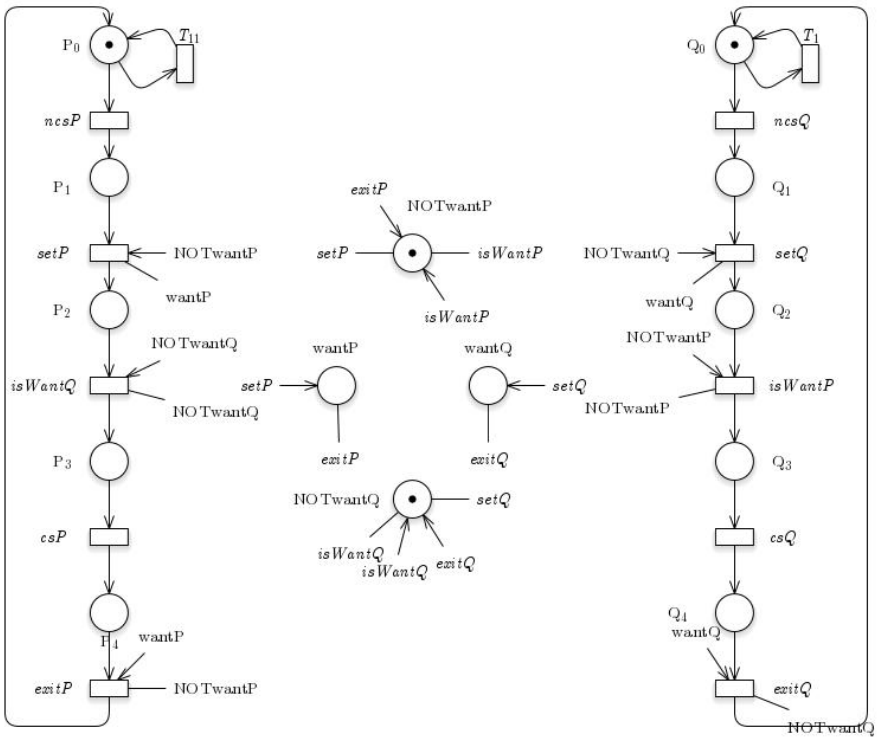

Figure 16: Modello GreatSPN per l'algoritmo 3.8

5.2 Mutua Esclusione

5.2.1 NuSmv

CTL: **True**

```
AG !(p.state = critical & q.state = critical)
```

LTL: **True**

```
G !(p.state = critical & q.state = critical)
```

5.2.2 GreatSPN

CTL: **True**

```
AG !(#P4 == 1 && #Q4 == 1)
```

La mutua esclusione è rispettata in quanto la modifica della variabile *want* viene effettuata prima della fase di controllo. Questo permette di evitare il caso, presente nell'esercizio 3.6, in cui l'esecuzione interfogliata dei due processi portava all'accesso di entrambi nella sezione critica.

5.3 Assenza di Deadlock

5.3.1 NuSmv

CTL: **False**

```
AG (p.state = enter -> AF (p.state = critical | q.state = critical))
```

LTL: **False**

```
G (p.state = enter -> F (p.state = critical | q.state = critical))
```

5.3.2 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 || #Q4 == 1))
```

L'assenza di deadlock non è rispettata nè in NuSmv ne in GreatSPN, in quanto se entrambi i processi eseguono *set*, entrambe le variabili *want* avranno valore $TRUE$ non permettendo il progresso di nessuno dei due. Si riportano i controesempi.

```
-- specification AG (p.state = set -> 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 <-
    wp = FALSE
    wq = 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 = set
  -> Input: 1.3 <-
    _process_selector_ = q
    q.running = TRUE
```

```
  p.running = FALSE
-> State: 1.3 <-
  q.state = set
-> Input: 1.4 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 1.4 <-
  wp = TRUE
  p.state = enter
-> Input: 1.5 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-- Loop starts here
-> State: 1.5 <-
  wq = TRUE
  q.state = enter
-> Input: 1.6 <-
-- Loop starts here
-> State: 1.6 <-
-> Input: 1.7 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-- Loop starts here
-> State: 1.7 <-
-> Input: 1.8 <-
  _process_selector_ = main
  running = TRUE
  p.running = FALSE
-> State: 1.8 <-
```

Controesempio di NuSmv per l'assenza di deadlock dell'esercizio 3.8.

```
--- AG ( #P1 == 1 -> AF (#P4 == 1 || #Q4 == 1)) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.000792 sec
         Evaluation time: 0.000793 sec

Generated counter-example:
================================== Trace ==================================
Initial state is: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
Initial state satisfies: E F (not ((not (P1 = 1)) or (not E G (not ((P4 = 1) or (Q4 = 1)))))).

1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
  State 1. satisfies: ((not (P1 = 1)) or (not E G (not ((P4 = 1) or (Q4 = 1))))).

  1.1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
    State 1.1. does not satisfy: (P1 = 1).

2: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
  State 2. does not satisfy: ((not (P1 = 1)) or (not E G (not ((P4 = 1) or (Q4 = 1))))).

  2.1: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
    State 2.1.L. satisfies: (P1 = 1).

    State 2.1.R. satisfies: E G (not ((P4 = 1) or (Q4 = 1))). Start of loop.

    2.1.R.1: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
      State 2.1.R.1. does not satisfy: ((P4 = 1) or (Q4 = 1)).

      2.1.R.1.1: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
        State 2.1.R.1.1.L. does not satisfy: (P4 = 1).

        State 2.1.R.1.1.R. does not satisfy: (Q4 = 1).

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

Figure 17: Controesempio per l'assenza di deadlock di GreatSPN.

5.4 Assenza di Starvation Individuale

CTL: **False**

```
AG (p.state = enter -> AF (p.state = critical))
```

LTL: **False**

```
G (p.state = enter -> F (p.state = critical))
```

5.4.1 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 ))
```

Essendo che la proprietà di assenza di deadlock è falsa, è lecito che anche quella di starvation individuale lo sia. Si riportano i controesempi.

```
-- specification AG (p.state = set -> AF p.state = critical) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 2.1 <-
    wp = FALSE
    wq = FALSE
    p.state = local
    q.state = local
  -> Input: 2.2 <-
    _process_selector_ = p
    running = FALSE
    q.running = FALSE
    p.running = TRUE
  -> State: 2.2 <-
    p.state = set
  -> Input: 2.3 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
  -> State: 2.3 <-
    q.state = set
  -> Input: 2.4 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
  -> State: 2.4 <-
    wp = TRUE
    p.state = enter
  -> Input: 2.5 <-
```

```
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-- Loop starts here
-> State: 2.5 <-
  wq = TRUE
  q.state = enter
-> Input: 2.6 <-
-- Loop starts here
-> State: 2.6 <-
-> Input: 2.7 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-- Loop starts here
-> State: 2.7 <-
-> Input: 2.8 <-
  _process_selector_ = main
  running = TRUE
  p.running = FALSE
-> State: 2.8 <-
```

Controesempio per la starvation individuale di NuSmv.

```
--- AG ( #P1 == 1 -> AF #P4 == 1 ) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.000748 sec
         Evaluation time: 0.000749 sec

Generated counter-example:
================================== Trace ==================================
Initial state is: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
Initial state satisfies: E F (not ((not (P1 = 1)) or (not E G (not (P4 = 1))))).

1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
  State 1. satisfies: ((not (P1 = 1)) or (not E G (not (P4 = 1)))).

  1.1: P0(1), NOTwantP(1), NOTwantQ(1), Q0(1)
    State 1.1. does not satisfy: (P1 = 1).

2: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
  State 2. does not satisfy: ((not (P1 = 1)) or (not E G (not (P4 = 1)))).

  2.1: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
    State 2.1.L. satisfies: (P1 = 1).

    State 2.1.R. satisfies: E G (not (P4 = 1)). Start of loop.

    2.1.R.1: P1(1), NOTwantP(1), NOTwantQ(1), Q0(1)
      State 2.1.R.1. does not satisfy: (P4 = 1).

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

Figure 18: Controesempio per l'assenza di starvation individuale di GreatSPN.

6 Esercizio 3.9

Algorithm 3.9: Fourth attempt

boolean wantp ← false, wantq ← false

	p		q
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	while wantq	q3:	while wantp
p4:	wantp ← false	q4:	wantq ← false
p5:	wantp ← true	q5:	wantq ← true
p6:	critical section	q6:	critical section
p7:	wantp ← false	q7:	wantq ← false

Figure 19: Descrizione dell'algoritmo

I modelli di seguito riportati utilizzano le seguenti specifiche per controllare

le 3 proprietà di cui sopra (espresse in CTL):

1. AG !(p.state = critical & q.state = critical) **True**
2. AG (p.state = set $->$ AF (p.state = critical | q.state = critical)) **False**
3. AG (p.state = set $->$ AF p.state = critical) **False**

Nota: le proprietà 2 *e* 3 *sono equivalenti sostituendo q a p. Le proprietà LTL si ottengono rimuovendo gli operatori di universalità e esistenzialità.*

6.1 NuSMV

Il modello implementa l'algoritmo 3.8 migliorato per mitigare la situazione di deadlock risultante dall'esecuzione interfogliata dei due processi. Entrambi eseguono un loop in *busy waiting* alternando il valore di *want* da $FALSE$ a $TRUE$, di modo da permettere l'accesso nel caso l'altro processo sia in attesa. Il deadlock non è però evitato del tutto: se i due processi entrano nel loop in maniera sincrona (un'istruzione alla volta) non riusciranno mai a proseguire alla SC, come mostrano i controesempi seguenti.

Sono stati utilizzati 7 stati, di cui:

local: Sezione non critica.

set: Richiede l'accesso assegnando alla variabile il valore $TRUE$.

enter: Lo stato inizia un loop che viene eseguito finchè la *want* opposta non diventa $FALSE$. Il loop itera sugli stati *loopUnset* e *loopSet*.

loopUnset: Cambia il valore della variabile *want* propria a $FALSE$, per permettere l'accesso all'altro processo nel caso questo sia già in *enter*.

loopSet: Cambia il valore della variabile *want* propria a $TRUE$, per segnalare l'intezione di entrare nella sezione critica.

critical: Sezione critica.

exit: Cambia il valore di *want* a $FALSE$ e ritorna a *local*.

Il modello utilizza due variabili, *wantp* e *wantq*, una per ogni processo. Non si utilizza più la variabile *turn*, ma i processi conoscono solo lo stato delle due variabili.

```
MODULE main
VAR
  wp : boolean;
  wq : boolean;
  p : process user(wp,wq);
  q : process user(wq,wp);
```

```
ASSIGN
  init(wp) := FALSE;
  init(wq) := FALSE;

SPEC -- PROGRESS
  AG (p.state = local -> EF p.state = enter )
SPEC -- MUTUAL EXCLUSION
  AG !(p.state = critical & q.state = critical)
SPEC -- DEADLOCK
  AG (p.state = set -> AF (p.state = critical | q.state = critical))
SPEC -- INDIVIDUAL STARVATION
  AG (p.state = set -> AF p.state = critical)

MODULE user(wp, wq)
VAR
  state : {local,set,enter,loopUnset, loopSet, critical, exit};
ASSIGN
  init(state) := local;
  next(state) :=
    case
      state = local : {local, set};
      state = set : enter;
      state = enter & wq = FALSE : critical;
      state = enter & wq = TRUE : loopUnset;
      state = loopUnset : loopSet;
      state = loopSet : enter;
      state = critical : exit;
      state = exit : local;
      TRUE : state;
    esac;
  next(wp) :=
    case
      state = set : TRUE;
      state = loopUnset : FALSE;
      state = loopSet : TRUE;
      state = exit : FALSE;
      TRUE : wp;
    esac;
FAIRNESS
  running
```

Il modello NuSmv per l'algoritmo 3.9.

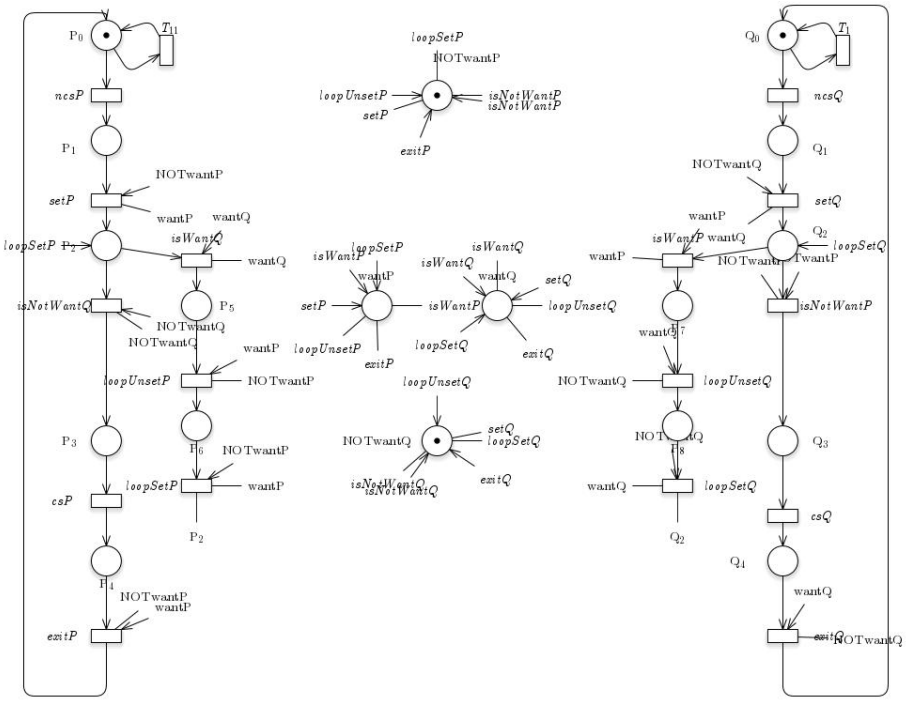

Figure 20: Modello GreatSPN per l'algoritmo 3.9

6.2 Mutua Esclusione

6.2.1 NuSmv

CTL: **True**

```
AG !(p.state = critical & q.state = critical)
```

LTL: **True**

```
G !(p.state = critical & q.state = critical)
```

6.2.2 GreatSPN

CTL: **True**

```
AG !(#P4 == 1 && #Q4 == 1)
```

La mutua esclusione è rispettata in quanto la modifica della variabile *want* viene effettuata prima della fase di controllo. Il *busy waiting* non cambia l'algoritmo da rispetto al 3.8 da questo punto di vista.

6.3 Assenza di Deadlock

6.3.1 NuSmv

CTL: **False**

```
AG (p.state = enter -> AF (p.state = critical | q.state = critical))
```

LTL: **False**

```
G (p.state = enter -> F (p.state = critical | q.state = critical))
```

6.3.2 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 || #Q4 == 1))
```

L'assenza di deadlock non è rispettata nè in NuSmv ne in GreatSPN. Anche con il *busy waiting* si ripresenta il problema dell'esercizio 3.8: se i processi sono entrambi nello stato di attesa (*enter*), entrambi entrano nel loop e ogni volta che viene eseguito il controllo sulle variabili, entrambe saranno $TRUE$. Si riportano i controesempi.

```
-- specification AG (p.state = set -> 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 <-
    wp = FALSE
    wq = 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 = set
  -> Input: 1.3 <-
    _process_selector_ = q
    q.running = TRUE
```

```
  p.running = FALSE
-> State: 1.3 <-
-> Input: 1.4 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 1.4 <-
  wp = TRUE
  p.state = enter
-> Input: 1.5 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-> State: 1.5 <-
-> Input: 1.6 <-
-> State: 1.6 <-
-> Input: 1.7 <-
-> State: 1.7 <-
  q.state = set
-> Input: 1.8 <-
-> State: 1.8 <-
  wq = TRUE
  q.state = enter
-> Input: 1.9 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 1.9 <-
  p.state = loopUnset
-> Input: 1.10 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-- Loop starts here
-> State: 1.10 <-
  q.state = loopUnset
-> Input: 1.11 <-
-> State: 1.11 <-
  wq = FALSE
  q.state = loopSet
-> Input: 1.12 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 1.12 <-
  wp = FALSE
  p.state = loopSet
-> Input: 1.13 <-
-> State: 1.13 <-
  wp = TRUE
```

```
  p.state = enter
-> Input: 1.14 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-> State: 1.14 <-
  wq = TRUE
  q.state = enter
-> Input: 1.15 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 1.15 <-
  p.state = loopUnset
-> Input: 1.16 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-> State: 1.16 <-
  q.state = loopUnset
```

Controesempio di NuSmv per l'assenza di deadlock dell'esercizio 3.9.

```
--- AG ( #P1 == 1 -> (#P4 == 1 || #Q4 == 1)) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.000414 sec
         Evaluation time: 0.000414 sec

Generated counter-example:
=================================== Trace ===================================
Initial state is: Q0(1), NOTwantQ(1), NOTwantP(1), P0(1)
Initial state satisfies: E F (not ((not (P1 = 1)) or ((P4 = 1) or (Q4 = 1)))).

1: Q0(1), NOTwantQ(1), NOTwantP(1), P0(1)
  State 1. satisfies: ((not (P1 = 1)) or ((P4 = 1) or (Q4 = 1))).

  1.1: Q0(1), NOTwantQ(1), NOTwantP(1), P0(1)
    State 1.1. does not satisfy: (P1 = 1).

2: Q0(1), NOTwantQ(1), NOTwantP(1), P1(1)
  State 2. does not satisfy: ((not (P1 = 1)) or ((P4 = 1) or (Q4 = 1))).

  2.1: Q0(1), NOTwantQ(1), NOTwantP(1), P1(1)
    State 2.1.L. satisfies: (P1 = 1).

    State 2.1.R. does not satisfy: ((P4 = 1) or (Q4 = 1)).

    2.1.R.1: Q0(1), NOTwantQ(1), NOTwantP(1), P1(1)
      State 2.1.R.1.L. does not satisfy: (P4 = 1).

      State 2.1.R.1.R. does not satisfy: (Q4 = 1).
```

Figure 21: Controesempio per l'assenza di deadlock di GreatSPN.

6.4 Assenza di Starvation Individuale

CTL: **False**

```
AG (p.state = enter -> AF (p.state = critical))
```

LTL: **False**

```
G (p.state = enter -> F (p.state = critical))
```

6.4.1 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 ))
```

Essendo che la proprietà di assenza di deadlock è falsa, è lecito che anche quella di starvation individuale lo sia. Si riportano i controesempi.

```
-- specification AG (p.state = set -> AF p.state = critical) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 2.1 <-
    wp = FALSE
    wq = FALSE
    p.state = local
    q.state = local
  -> Input: 2.2 <-
    _process_selector_ = p
    running = FALSE
    q.running = FALSE
    p.running = TRUE
  -> State: 2.2 <-
    p.state = set
  -> Input: 2.3 <-
    _process_selector_ = q
    q.running = TRUE
    p.running = FALSE
  -> State: 2.3 <-
  -> Input: 2.4 <-
    _process_selector_ = p
    q.running = FALSE
    p.running = TRUE
  -- Loop starts here
  -> State: 2.4 <-
    wp = TRUE
    p.state = enter
  -> Input: 2.5 <-
```

```
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-- Loop starts here
-> State: 2.5 <-
-> Input: 2.6 <-
-- Loop starts here
-> State: 2.6 <-
-> Input: 2.7 <-
-> State: 2.7 <-
  q.state = set
-> Input: 2.8 <-
-> State: 2.8 <-
  wq = TRUE
  q.state = enter
-> Input: 2.9 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 2.9 <-
  p.state = loopUnset
-> Input: 2.10 <-
-> State: 2.10 <-
  wp = FALSE
  p.state = loopSet
-> Input: 2.11 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-> State: 2.11 <-
  q.state = critical
-> Input: 2.12 <-
  _process_selector_ = p
  q.running = FALSE
  p.running = TRUE
-> State: 2.12 <-
  wp = TRUE
  p.state = enter
-> Input: 2.13 <-
  _process_selector_ = q
  q.running = TRUE
  p.running = FALSE
-> State: 2.13 <-
  q.state = exit
-> Input: 2.14 <-
-> State: 2.14 <-
  wq = FALSE
  q.state = local
```

Controesempio per la starvation individuale di NuSmv.

```
--- AG ( #P1 == 1 -> #P4 == 1 ) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.000304 sec
         Evaluation time: 0.000304 sec

Generated counter-example:
================================== Trace ==================================
Initial state is: Q0(1), NOTwantQ(1), NOTwantP(1), P0(1)
Initial state satisfies: E F (not ((not (P1 = 1)) or (P4 = 1))).

1: Q0(1), NOTwantQ(1), NOTwantP(1), P0(1)
  State 1. satisfies: ((not (P1 = 1)) or (P4 = 1)).

  1.1: Q0(1), NOTwantQ(1), NOTwantP(1), P0(1)
    State 1.1. does not satisfy: (P1 = 1).

2: Q0(1), NOTwantQ(1), NOTwantP(1), P1(1)
  State 2. does not satisfy: ((not (P1 = 1)) or (P4 = 1)).

  2.1: Q0(1), NOTwantQ(1), NOTwantP(1), P1(1)
    State 2.1.L. satisfies: (P1 = 1).

    State 2.1.R. does not satisfy: (P4 = 1).
```

Figure 22: Controesempio per l'assenza di starvation individuale di GreatSPN.

7 Esercizio 3.10

Algorithm 3.10: Dekker's algorithm

boolean wantp ← false, wantq ← false
integer turn ← 1

	p		q
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	while wantq	q3:	while wantp
p4:	if turn = 2	q4:	if turn = 1
p5:	wantp ← false	q5:	wantq ← false
p6:	await turn = 1	q6:	await turn = 2
p7:	wantp ← true	q7:	wantq ← true
p8:	critical section	q8:	critical section
p9:	turn ← 2	q9:	turn ← 1
p10:	wantp ← false	q10:	wantq ← false

Figure 23: Descrizione dell'algoritmo

I modelli di seguito riportati utilizzano le seguenti specifiche per controllare le 3 proprietà di cui sopra (espresse in CTL):

1. AG !(p.state = critical & q.state = critical) **True**
2. AG (p.state = set − > AF (p.state = critical | q.state = critical)) **True**
3. AG (p.state = set − > AF p.state = critical) **True**

Nota: le proprietà 2 *e* 3 *sono equivalenti sostituendo q a p. Le proprietà LTL si ottengono rimuovendo gli operatori di universalità e esistenzialità.*

7.1 NuSMV

Il modello implementa l'algoritmo di Dekker, che fa uso sia delle due variabili booleane che della variabile *turn*, riuscendo a rispettare tutte e 3 le proprietà. Sono stati utilizzati 10 stati, di cui:

local: Sezione non critica.

set: Richiede l'accesso assegnando alla variabile il valore $TRUE$.

enter: Lo stato inizia un loop che viene eseguito finchè la *want* opposta non diventa $FALSE$. Il loop itera sugli stati *checkTurn*, *loopUnset*, *waitTurn* e *loopSet*.

checkTurn: Controlla che il valore della variabile *turn* sia corrispondente al processo opposto. Se così è, allora permette la discesa nel loop. Altrimenti il processo ritorna a *enter*.

loopUnset: Cambia il valore della variabile *want* propria a $FALSE$, per permettere l'accesso all'altro processo nel caso questo sia già in *enter*.

waitTurn: Obbliga l'attesa del proprio turno controllando la variabile *turn*.

loopSet: Cambia il valore della variabile *want* propria a $TRUE$, per segnalare l'intezione di entrare nella sezione critica.

critical: Sezione critica.

setTurn: Assegna il valore di *turn* all'altro processo.

exit: Cambia il valore di *want* a $FALSE$ e ritorna a *local*.

Ogni processo è a conoscenza di *wantp*, *wantq*, della variabile *turn*, del proprio *id* e di quello dell'altro processo.

```
MODULE main
VAR
  wp : boolean;
  wq : boolean;
  turn : 1..2;
  p : process user(wp,wq, turn, 1, 2);
  q : process user(wq,wp, turn, 2, 1);
ASSIGN
  init(wp) := FALSE;
  init(wq) := FALSE;
  init(turn) := 1;

SPEC -- PROGRESS
  AG (p.state = local -> EF p.state = enter )
SPEC -- MUTUAL EXCLUSION
  AG !(p.state = critical & q.state = critical)
SPEC -- DEADLOCK
  AG (p.state = set -> AF (p.state = critical | q.state = critical))
SPEC -- INDIVIDUAL STARVATION
  AG (p.state = set -> AF p.state = critical)

MODULE user(wp, wq, turn, me, other)
VAR
  state : {local, set, enter, checkTurn, loopUnset, waitTurn, loopSet,
       critical, setTurn, exit};
ASSIGN
  init(state) := local;
  next(state) :=
    case
      state = local : {local, set};
```

```
      state = set : enter;

      state = enter & wq = FALSE : critical; -- Go to CS

      state = enter & wq = TRUE : checkTurn; -- Start waiting loop
      state = checkTurn & turn = me : enter;  -- Go back to loop check
      state = checkTurn & turn = other : loopUnset; -- Descend in loop
      state = loopUnset : waitTurn;
      state = waitTurn & turn = me : loopSet; -- Wait until turn == 1
      state = loopSet : enter;               -- End of loop

      state = critical : exit;
      state = setTurn : exit;
      state = exit : local;
      TRUE : state;
    esac;

  next(wp) :=
    case
      state = set : TRUE;
      state = loopUnset : FALSE;
      state = loopSet : TRUE;
      state = exit : FALSE;
      TRUE : wp;
    esac;
  next(turn) :=
    case
      state = setTurn : other;
      TRUE : turn;
    esac;
FAIRNESS
  running
```

Modello NuSmv per l'algoritmo di Dekker.

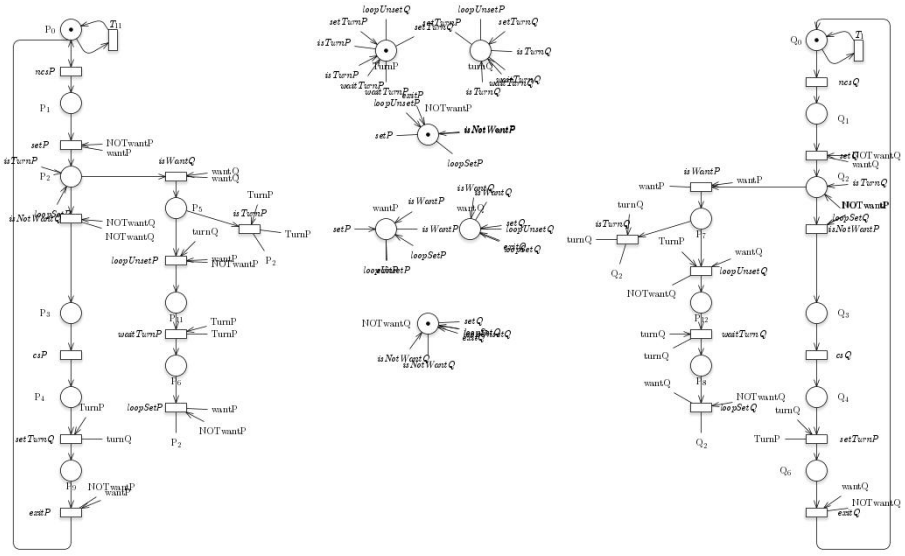

Figure 24: Modello in GreatSPN dell'algoritmo di Dekker

7.2 Mutua Esclusione

7.2.1 NuSmv

CTL: **True**

```
AG !(p.state = critical & q.state = critical)
```

LTL: **True**

```
G !(p.state = critical & q.state = critical)
```

7.2.2 GreatSPN

CTL: **True**

```
AG !(#P4 == 1 && #Q4 == 1)
```

7.3 Assenza di Deadlock

7.3.1 NuSmv

CTL: **True**

```
AG (p.state = enter -> AF (p.state = critical | q.state = critical))
```

LTL: **True**

```
G (p.state = enter -> F (p.state = critical | q.state = critical))
```

7.3.2 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 || #Q4 == 1))
```

In NuSmv, dove è possibile imporre il requisito di fairness, l'assenza di deadlock è rispettatà. In GreatSPN invece non è possibile imporre la fairness e per questo motivo può accadere che si verifichi deadlock perchè uno dei processi rimane fermo (non esegue transizioni di progresso).

```
--- AG (#P1 == 1 -> AF (#Q4 == 1 || #P4 == 1)) ---
         Formula name: MEASURE0
         Evaluation: false
         Sat-set generation time: 0.007309 sec
         Evaluation time: 0.007314 sec

Generated counter-example:
================================== Trace ==================================
Initial state is: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P0(1)
Initial state satisfies: E F (not ((not (P1 = 1)) or (not E G (not ((Q4 = 1) or (P4 = 1)))))).

1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P0(1)
  State 1. satisfies: ((not (P1 = 1)) or (not E G (not ((Q4 = 1) or (P4 = 1))))).

  1.1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P0(1)
    State 1.1. does not satisfy: (P1 = 1).

2: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P1(1)
  State 2. does not satisfy: ((not (P1 = 1)) or (not E G (not ((Q4 = 1) or (P4 = 1))))).

  2.1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P1(1)
    State 2.1.L. satisfies: (P1 = 1).

    State 2.1.R. satisfies: E G (not ((Q4 = 1) or (P4 = 1))). Start of loop.

    2.1.R.1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P1(1)
      State 2.1.R.1. does not satisfy: ((Q4 = 1) or (P4 = 1)).

      2.1.R.1.1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P1(1)
        State 2.1.R.1.1.L. does not satisfy: (Q4 = 1).

        State 2.1.R.1.1.R. does not satisfy: (P4 = 1).

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

Figure 25: Controesempio per l'assenza di deadlock di GreatSPN.

7.4 Assenza di Starvation Individuale

CTL: **True**

```
AG (p.state = enter -> AF (p.state = critical))
```

LTL: **True**

```
G (p.state = enter -> F (p.state = critical))
```

7.4.1 GreatSPN

CTL: **False**

```
AG (#P1 == 1 -> AF (#P4 == 1 ))
```

Come per l'assenza di deadlock, anche per la starvation individuale NuSmv con $FAIRNESSrunning$ ci garantisce che non si verifichi starvation, mentre per GreatSPN non si può dire lo stesso.

```
--- AG (#P1 == 1 -> AF #P4 == 1) ---
        Formula name: MEASURE0
        Evaluation: false
        Sat-set generation time: 0.005327 sec
        Evaluation time: 0.005328 sec

Generated counter-example:
================================== Trace ==================================
Initial state is: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P0(1)
Initial state satisfies: E F (not ((not (P1 = 1)) or (not E G (not (P4 = 1))))).

1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P0(1)
  State 1. satisfies: ((not (P1 = 1)) or (not E G (not (P4 = 1)))).

  1.1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P0(1)
    State 1.1. does not satisfy: (P1 = 1).

2: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P1(1)
  State 2. does not satisfy: ((not (P1 = 1)) or (not E G (not (P4 = 1)))).

  2.1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P1(1)
    State 2.1.L. satisfies: (P1 = 1).

    State 2.1.R. satisfies: E G (not (P4 = 1)). Start of loop.

    2.1.R.1: Q0(1), NOTwantQ(1), TurnP(1), NOTwantP(1), P1(1)
      State 2.1.R.1. does not satisfy: (P4 = 1).

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

Figure 26: Controesempio per l'assenza di starvation individuale di GreatSPN.

7.5 Confronto tra NuSmv e GreatSPN

Proprietà degli algoritmi in NuSmv			
Esercizio	Mutua Esclusione	No Deadlock	No Starvation Ind.
3.2	True	False	False
3.5	True	False	False
3.6	False	True	False
3.8	True	False	False
3.9	True	False	False
3.10	True	True	True

Proprietà degli algoritmi in GreatSPN			
Esercizio	Mutua Esclusione	No Deadlock	No Starvation Ind.
3.2	True	False	False
3.5	True	False	False
3.6	False	**False**	False
3.8	True	False	False
3.9	True	False	False
3.10	True	**False**	**False**

7.6 Bisimulazione: equivalenza tra esercizi 3.2 e 3.5

Si vuole eguagliare i due modelli in algebra dei processi per determinare l'equivalenza dei due algoritmi. Per fare ciò, si utilizza la tecnica di bisimulazione estesa, ovvero l'algoritmo che considera le azioni τ, nel modello segnate come *Sync*.

Si riportano di seguito i derivation graph per i due algoritmi, sostituendo le azioni in *Sync* a τ.

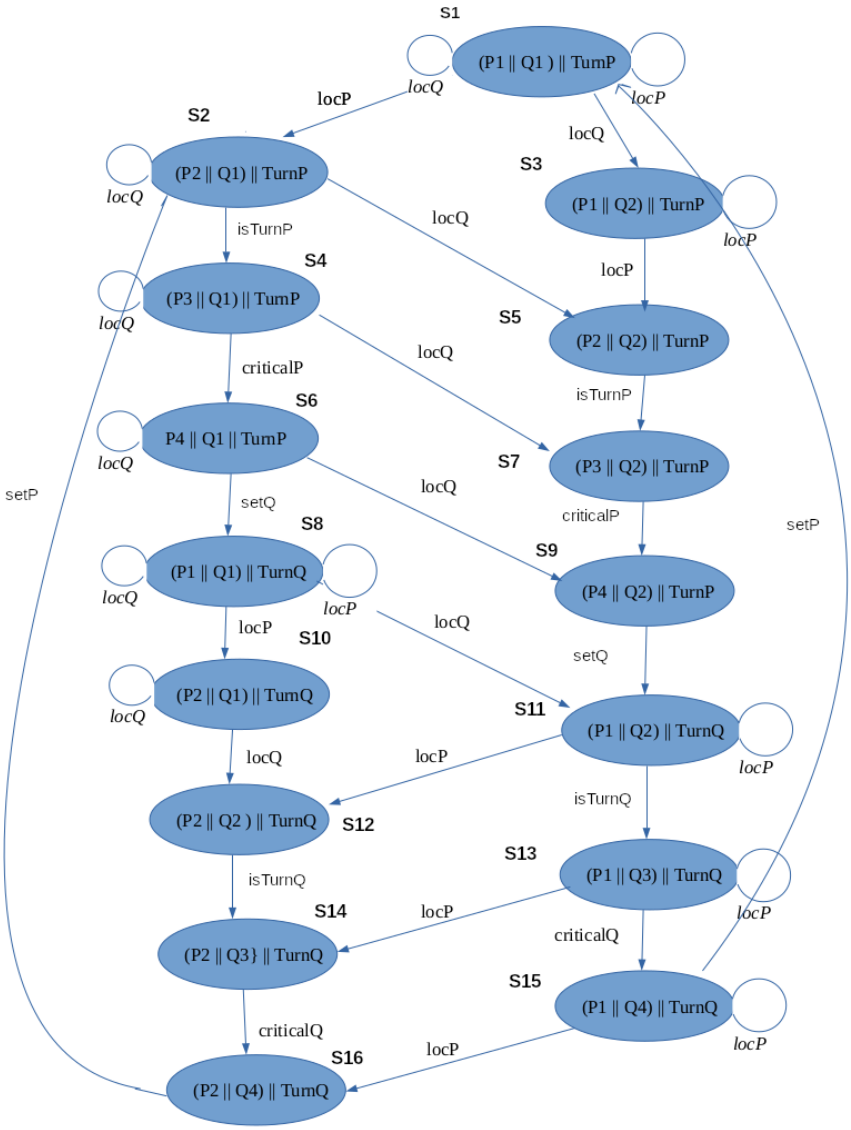

Figure 27: Derivation Graph per la rete 3.2

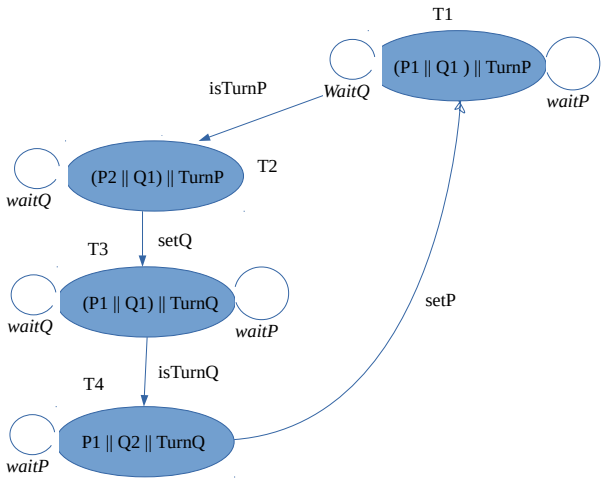

Figure 28: Derivation Graph per la rete 3.5

Gli stati in comune tra i due modelli sono: **isTurnP, isTurnQ, setP, setQ**. Si applica lo *split* utilizzando questi stati come Azioni. Il set iniziale è rappre-

sentato dall'insieme degli stati dei due processi.

7.6.1 Bisimulazione: Stato iniziale

```
{S1,T1}{T2,T3,T4,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16}
```

7.6.2 Split su setP

Si applica lo split sull'ultimo set:

```
{T2,T3,T4,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16}
```

Si ottiene:

```
{S1,T1}{T4,S15,S16}{T2,T3,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14}
```

7.6.3 Split su isTurnQ

Si applica lo split sull'ultimo set.

```
{T2,T3,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14}
```

Si ottiene:

```
{S1,T1}{T4,S15,S16}{T3,S12,S11}{T2,S2,S3,S4,S5,S6,S7,S8,S9,S10,S13,S14}
```

7.6.4 Split su setQ

Si applica lo split sull'ultimo set.

```
{T2,S2,S3,S4,S5,S6,S7,S8,S9,S10,S13,S14}
```

Si ottiene:

```
{S1,T1}{T4,S15,S16}{T3,S12,S11}{T2,S6,S9}{S2,S3,S4,S5,S7,S8,S10,S13,S14}
```

7.6.5 Split su isTurnP

Si applica lo split sul primo set.

```
{S1,T1}
```

Si ottiene:

```
{S1}{T1}{T4,S15,S16}{T3,S12,S11}{T2,S6,S9}{S2,S3,S4,S5,S7,S8,S10,S13,S14}
```

7.6.6 Split su setQ

Si applica lo split sul quarto set.

```
{T2,S6,S9}
```

Si ottiene:

```
{S1}{T1}{T4,S15,S16}{T3,S12,S11}{T2,S9}{S6}{S2,S3,S4,S5,S7,S8,S10,S13,S14}
```

7.6.7 Split su isTurnQ

Si applica lo split sul quarto set.

```
{T3,S12,S11}
```

Si ottiene:

```
{S1}{T1}{T4,S15,S16}{T3}{S11,S12}{T2,S6}{S9}{S2,S3,S4,S5,S7,S8,S10,S13,S14}
```

7.6.8 Split su setP

Si applica lo split sul terzo set.

```
{T4,S15,S16}
```

Si ottiene:

```
{S1}{T1}{T4}{S15,S16}{T3}{S11,S12}{T2,S6}{S9}{S2,S3,S4,S5,S7,S8,S10,S13,S14}
```

7.6.9 Split su setQ

Si applica lo spit sul settimo set.

```
{T2,S6}
```

Si ottiene:

```
{S1}{T1}{T4}{S15,S16}{T3}{S11,S12}{T2}{S6}{S9}{S2,S3,S4,S5,S7,S8,S10,S13,S14}
```

7.6.10 Bisimulazione: conclusione

L'algoritmo termina in quanto non è più possibile dividere alcun set sulla base delle transizioni comuni. Dopo aver applicato gli *splitting* di cui sopra, i set sono partizionati nella seguente maniera:

{T1}{T4}{T3}{T2}{S1}{S15,S16}{S11,S12}{S6}{S9}{S2,S3,S4,S5,S7,S8,S10,S13,S14}

Come si può notare, gli stati dell'algoritmo 3.5 sono gli unici elementi nel proprio set. Se ci fosse bisimulazione, essi dovrebbero essere accoppiati a uno o più stati dell'algoritmo 3.2. Possiamo quindi affermare che **non c'è bisimulazione** tra i due algoritmi.

7.7 Riduzione Strutturale in modelli di reti P/T

7.7.1 3.2 - 3.5

Si vogliono applicare le 4 tecniche viste di riduzione strutturale per confrontare le reti 3.2 e 3.5. Si riportano i due modelli in reti P/T.

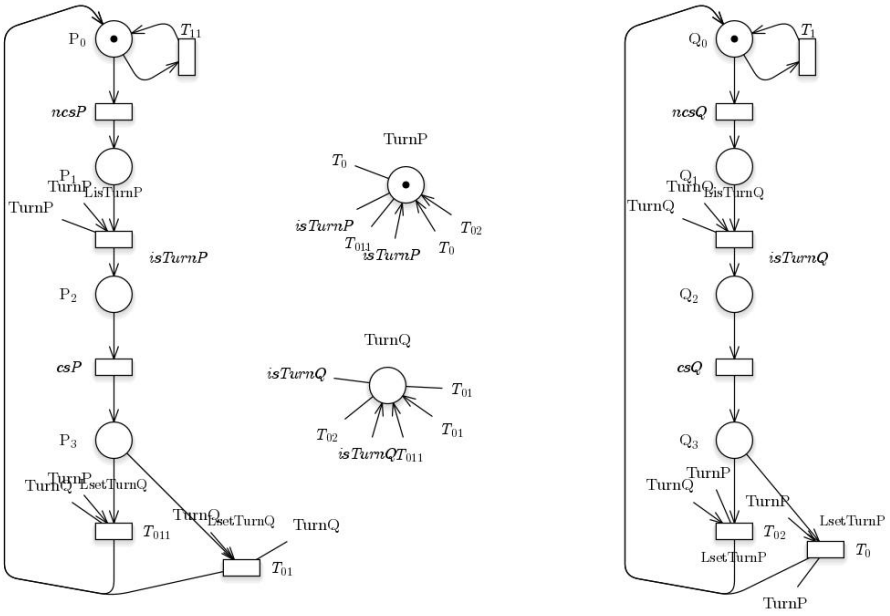

Figure 29: Modello P/T della rete 3.2

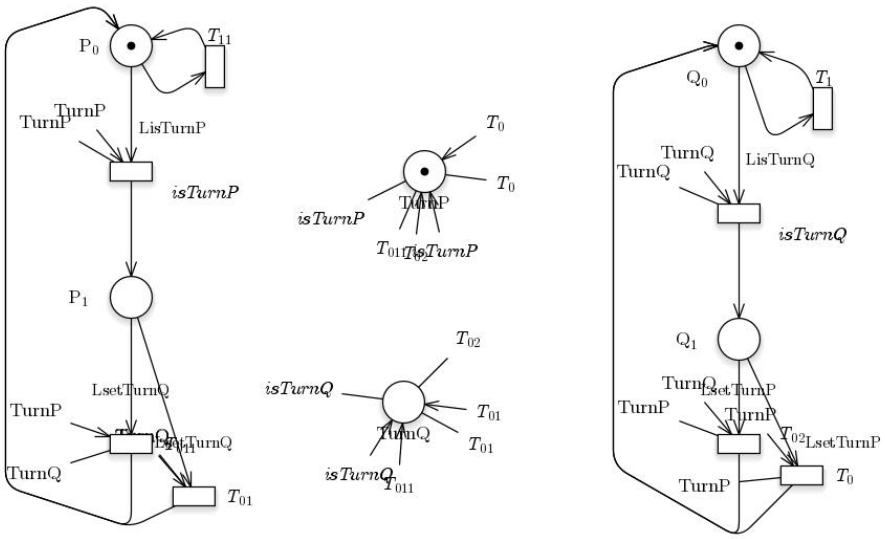

Figure 30: Modello P/T della rete 3.5

La rete 3.5 rappresenta un modello semplificato dell'algoritmo 3.2, pertanto si riduce la rete 3.2.

Si applica:

Fusion of places: La fusione dei posti $P0, P1$ - $Q0, Q1$ e $P2, P3$ - $Q2, Q3$ (quindi su entrambe le transizioni *ncs* e *cs*)

Ottenendo una rete equivalente alla 3.5.

7.7.2 3.6 - 3.8

Si vogliono applicare le 4 tecniche viste di riduzione strutturale per confrontare le reti 3.6 e 3.8. Si riportano i due modelli in reti P/T.

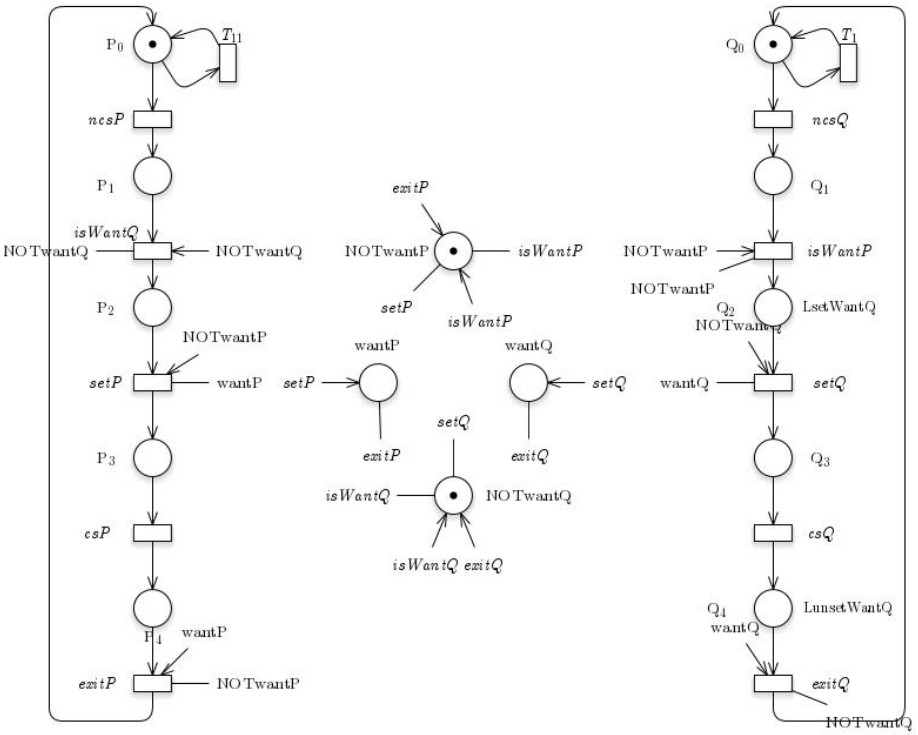

Figure 31: Modello P/T della rete 3.6

La rete 3.6 e la rete 3.8 si differenziano per l'inversione delle istruzioni di *set* e *wait*.

Si applica:

Fusion of places: La fusione dei posti $P0, P1$ - $Q0, Q1$ su entrambe le reti.

Fusion of places: La fusione dei posti $P3, P4$ - $Q3, Q4$ su entrambe le reti.

Self-loop elimination: L'eliminazione dei self-loop delle transizioni $T1, T11$ su entrambe le reti.

A questo punto si puo' notare che le transizioni $set\{P, Q\}$ e $isWant\{P, Q\}$ non sono riducibili in quanto non presentano nessuna delle strutture che permettono la riduzione. Le due reti non sono quindi equivalenti.

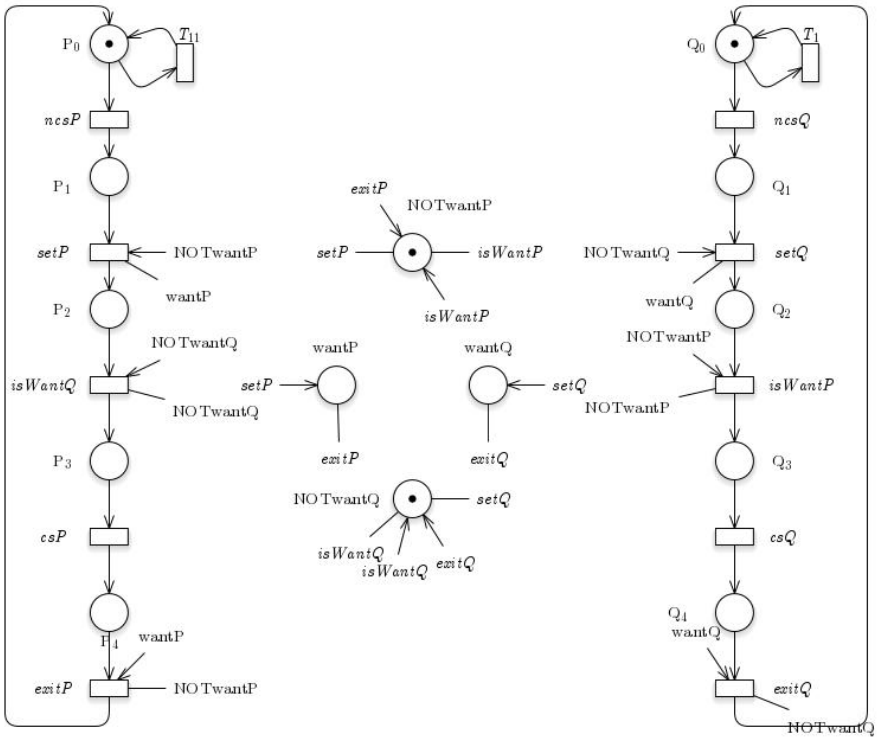

Figure 32: Modello P/T della rete 3.8