Verifica con SPIN

Verifiche di correttezza in SPIN

- SPIN è un model checker
- Verifica correttezza = model checking
  - Dato un modello ed una proprietà (specifica) verifica se il modello soddisfa la proprietà
- Modelli sono programmi Promela
- Proprietà possono essere specificate con asserzioni che testano espressioni condizionali sullo stato del modello

Concetto di stato

- Stato di un programma:

  lista dei valori di tutte le variabili insieme al location counter (espresso con un int)

- Ad es nel programma

```
1  active proctype P() {
2          int value = 123;
3          int reversed;
4          reversed =  (value % 10) * 100 +
5                                  ((value / 10) % 10) * 10 + (value / 100);
6          printf("value = %d, reversed = %d\n", value, reversed)
7  }
```

- prima di eseguire la printf, lo stato è (123,321,6)

Concetto di computazione

- Computazione di un programma:

  sequenza di stati che comincia da uno stato iniziale e continua con tutti gli stati ad ogni istruzione eseguita (istruzioni determinano transizioni)

- Ad es. nel programma precedente esiste un'unica computazione:

  (123,0,4) -> (123,321,6) –> (123,321,7)

  - sono omessi gli stati dove il valore delle variabili non cambia

Asserzioni e spazio degli stati

- Spazio degli stati: insieme di stati possibili
  - prodotto cartesiano dell'insieme delle control location e degli insiemi di valori che possono assumere le variabili
- Usiamo le asserzioni per esprimere specifiche di correttezza
  - Sintassi: assert(condizione)
- Un'espressione condizionale partiziona lo spazio degli stati in due
  - in base ai sui valori di verità

Verifica con asserzioni

- Si possono inserire asserzioni tra due istruzioni consecutive
- Se la condizione è verificata, l'esecuzione procede all'istruzione successiva
- altrimenti, il programma entra in uno stato di errore
- Le asserzioni possono specificare:
  - pre-condizioni (proprietà che devono essere verificate negli stati iniziali)
  - post-condizioni (proprietà che devono essere verificate negli stati finali)

Esempio

```
active proctype P() {
    int dividend = 15;   int divisor  = 4;      int quotient, remainder,n;
    assert (dividend >= 0 && divisor > 0);  //pre-condizione
    quotient = remainder = 0;
    n = dividend;
    do
    :: n >= divisor ->
            quotient++;        n = n - divisor
    :: else ->       remainder = n;  break
    od;
    printf("%d divided by %d = %d, remainder = %d\n",
                                dividend, divisor, quotient, remainder);
    assert (0 <= remainder && remainder < divisor);  //post-condizioni
    assert (dividend == quotient * divisor + remainder);
}
```

Invarianti

- Un'asserzione in un loop specifica un'invariante
  - deve rimanere vera in tutte le iterazioni
- Un'invariante per il programma precedente:
  - il dividendo (dividend) deve sempre valere quotient * divisor + remainder + n:

```
assert (dividend == quotient * divisor + remainder + n);
```

  - il resto non può essere negativo ed è strettamente minore del divisore:

```
assert (0 <= remainder && remainder < divisor);
```

Programma per il calcolo del max

- Consideriamo il programma:

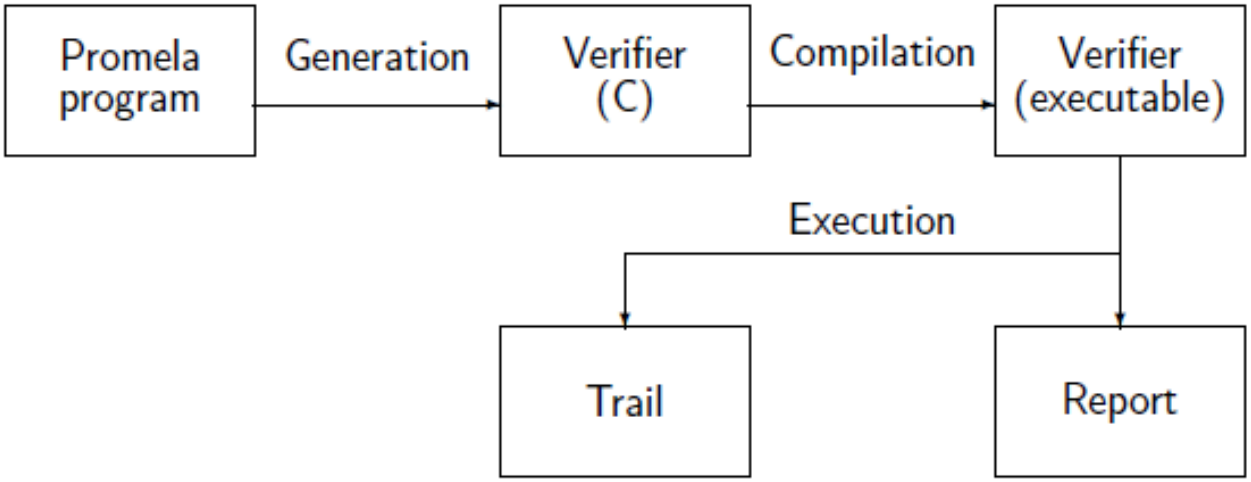
```
active proctype P() {
        int a = 5, b = 5, max;
        if
        :: a >= b -> max = a;
        :: b >= a -> max = b+1;
        fi;
        assert (a >= b -> max == a : max == b);
}
```

- ha un errore nella seconda alternativa

Verificare un programma in SPIN

- Nei programmi deterministici senza input c'è solo una computazione
  - una singola simulazione random basta
- Per programmi concorrenti o non deterministici , verificare tutte le possibili computazioni può richiedere backtracking (sulle istruzioni eseguite)
- Per effettuare queste esplorazioni in maniera efficiente, SPIN costruisce per ogni programma un programma C detto verificatore

I tre passi della verifica in SPIN



- `spin.exe -a file.pml`
  (genera il verificatore `pan.c` dal codice sorgente)
- `gcc -o pan pan.c` (compila il verificatore)
- `pan` (esegue il verificatore e genera)

Traccia d'errore

- Quando viene individuato un errore viene generato un trail file (con nome uguale al nome del file sorgente e estensione .trail)
- Un trail file contiene l'informazione sulla traccia d'errore individuata da SPIN
- Per arrestare la ricerca al primo errore si esegue comando `pan` senza opzioni
  - `pan -e` crea trail files per tutti gli errori (`file.pml1.trail,....,file.pmlN.trail`)
  - `pan -cN` invece per fermarsi all'errore N

Uso dei trail files

- Trail files non sono di facile lettura
- Possono essere interpretati da SPIN nella simulazione guidata per ricostruire la computazione corrispondente nel programma

```
spin.exe -g -l -p -r -s -t -u250 file.pml
```

  - `-g/-l` per visualizzare variabili globali/locali
  - `-p` per visualizzare istruzioni
  - `-s/-r` per visualizzare send/receive
  - `-t` indica che trail file è file.pml.trail

  (`-tN` indica che trail file è file.pmlN.trail)

  si può usare anche –`k` seguito da nome trail file
  - `-uN` per interrompere dopo N passi

Visualizzazione computazione

- SPIN consente di visualizzare un'esecuzione random oppure interattiva
- Stesse opzioni che per simulazione guidata per la scelta delle informazioni sullo stato da visualizzare
- Per esecuzione random basta omettere `–t`
- Per esecuzione interattiva basta usare `–i` invece di `–t`

Variabile _nr_pr

- Problema: vogliamo stampare il valore di una variabile condivisa dopo che tutti i processi hanno terminato
- Serve un modo per sapere il numero di processi attivi
- In Promela la variabile predefinita _nr_pr ha questo valore
- Un'istruzione composta solo da una condizione, se questa è falsa, blocca l'esecuzione del processo
  - il processo resta in attesa fino a quando non diventa vero, come per l'if quando nessuna delle alternative può essere eseguita
- Per risolvere il problema, possiamo inserire nel processo *init* l'istruzione *(_nr_pr==1)* prima della printf

Consideriamo il programma

```
byte    n = 0;

proctype P() {
    byte temp,i=1;
    do
    :: (i>10) -> break
    :: else ->
            temp = n;
            n = temp + 1;
            i++;
    od
}

init {
    atomic {
        run P();
        run P()
    }
    (_nr_pr == 1);
    printf("The value is %d\n", n);
}
```

Quali sono i possibili valori per n?

Analizziamo il programma

- I due processi eseguono ciascuno 10 incrementi
- Se scegliamo come interleaving quello che esegue prima interamente un processo e poi l'altro otteniamo 20
- Chiaramente è il valore massimo calcolabile
  - ci sono 20 incrementi in tutto!
- Con perfect interleaving come nell'esempio con due processi che facevano un singolo incremento otteniamo 10
- E' il minimo possibile?

Verifica con asserzioni

- Possiamo usare le asserzioni ed effettuare la verifica
  - ad es. assert(n>9)
  - se 10 è il minimo allora l'asserzione è sempre vera e quindi abbiamo terminato
  - altrimenti proviamo per un numero basso (potremmo procedere come in una ricerca binaria per convergere più rapidamente)
- Il minimo sarà il valore k tale che
  - la verifica con assert(n>k) individua un errore e
  - con assert(n>=k) no

Verifichiamo il programma

```
byte    n = 0;

proctype P() {
    byte temp,i=1;
    do
    :: (i>10) -> break
    :: else ->
            temp = n;
            n = temp + 1;
            i++;
    od
}
```

```
init {
    atomic {
        run P();
        run P()
    }
    (_nr_pr == 1);
    printf("The value is %d\n", n);
     assert(n>k);
}
```

partiamo da k=5 e si procede come nella ricerca binaria

Critical section

- Un pezzo di codice che accede una risorsa condivisa
- Tipica richiesta: accesso alla risorsa soltanto da parte di un processo alla volta
- Esempi
  - un processo che deve aggiornare delle variabili correlate e che quindi non vi devono essere interferenze da parte di altri processi fino al compimento del lavoro
  - accesso alla stampante

Problema della critical section

- Un sistema con due o più processi
- Il codice dei processi è diviso tra *critical* e *non critical*
- Ogni processo non può terminare l'esecuzione nella critical section
- Problema: progettare un algoritmo che garantisca:
  - mutua esclusione (al massimo 1 processo in CS)
  - assenza di deadlock (mai qualche processo vuole eseguire la sua CS e nessuno ci riesce)
  - assenza di starvation (se un processo vuole eseguire la sua CS, alla fine ci riesce)

Prima soluzione

```
bool wantP = false, wantQ = false;

active proctype P() {
    do
    ::
        printf("Non critical section P\n");
        wantP = true;
        printf("Critical section P\n");
        wantP = false
    od
}
```

```
active proctype Q() {
    do
    ::
        printf("Non critical section Q\n");
        wantQ = true;
        printf("Critical section Q\n");
        wantQ = false
    od
}
```

Aggiungiamo asserzioni e contatore

```
bool wantP = false, wantQ = false;
byte critical = 0;

active proctype P() {
    do
    :: printf("Non critical section P\n");
       wantP = true;   critical++;
       printf("Critical section P\n");
       assert (critical <= 1);
       critical--;
       wantP = false
    od
}
```

```
active proctype Q() {
      do
      :: printf("Non critical section Q\n");
         wantQ = true;
         critical++;
         printf("Critical section Q\n");
         assert (critical <= 1);
         critical--;
         wantQ = false
      od
}
```

Sincronizzazione

Primitive di sincronizzazione

- servono a gestire l'accesso a risorse condivise
- Semaforo:
  - inizializzato con il numero di risorse disponibili;
  - operazioni wait e signal:
  - wait: viene decrementato contatore, se il suo valore è negativo il processo viene sospeso in attesa che il valore diventi >= 0, e quindi si procede con l'esecuzione
  - signal: operazione sempre eseguibile, viene incrementato il valore del contatore
- Lock
  - per accedere alla risorsa un processo deve acquisirne il lock
  - a seconda del tipo di risorsa l'acquisizione del lock è esclusiva o può essere rilasciata a più processi

Primitive di sincronizzazione

- Monitor:
  - può essere usato da due o più thread
  - realizza automaticamente la mutua esclusione
  - primitiva di più alto livello rispetto a lock
- Promela non ha primitive di sincronizzazione
  - per implementare meccanismi di sincronizzazione si usa il concetto di *eseguibilità* delle istruzioni
  - l'esecuzione di un processo si blocca ad una istruzione non eseguibile e resta in attesa finchè non diventa eseguibile
- Eseguibilità:
  - per ogni istruzione in Promela, sono stabilite delle condizioni per la sua eseguibilità
  - espressioni condizionali sono eseguibili se e solo se sono valutate true
  - printf e assegnamenti sono sempre eseguibili

Ultimo programma "critical section"

```
bool wantP = false, wantQ = false;
byte critical = 0;

active proctype P() {
    do
    :: printf("Non critical section P\n");
       wantP = true;   critical++;
        printf("Critical section P\n");
        assert (critical <= 1);
        critical--;
        wantP = false
    od
}
```

```
active proctype Q() {
      do
      :: printf("Non critical section Q\n");
        wantQ = true;
         critical++;
        printf("Critical section Q\n");
         assert (critical <= 1);
         critical--;
         wantQ = false
      od
}
```

Mutua esclusione non soddisfatta: nessuna sincronizzazione tra i processi

Sincronizzazione by blocking

- Prima di entrare nella sezione critica viene letta la variabile dell'altro processo
  - Se l'altro processo è in sezione critica, allora attendi
- Busy waiting: il processo esegue un loop per perdere tempo in attesa che si sblocchi la condizione

  (In Promela, la seconda alternativa non serve)

```
do
:: !wantQ -> break
:: else -> skip
od
```

- Alcuni linguaggi supportano *blocking statements* e non è necessario scrivere loop che perdono tempo
- In Promela, l'effetto del blocking statement è ottenuto attraverso il concetto di eseguibilità
  - E' sufficiente inserire l'istruzione !wantQ

Terza versione programma "sezione critica"

```
bool wantP = false, wantQ = false;
byte critical = 0;

active proctype P() {
   do
   :: printf("Non critical section P\n");
      wantP = true;
      !wantQ;
      printf("Critical section P\n");
      wantP = false
   od
}
```

```
active proctype Q() {
     do
     :: printf("Non critical section Q\n");
        wantQ = true;
        !wantP;
        printf("Critical section Q\n");
        wantQ = false
     od
}
```

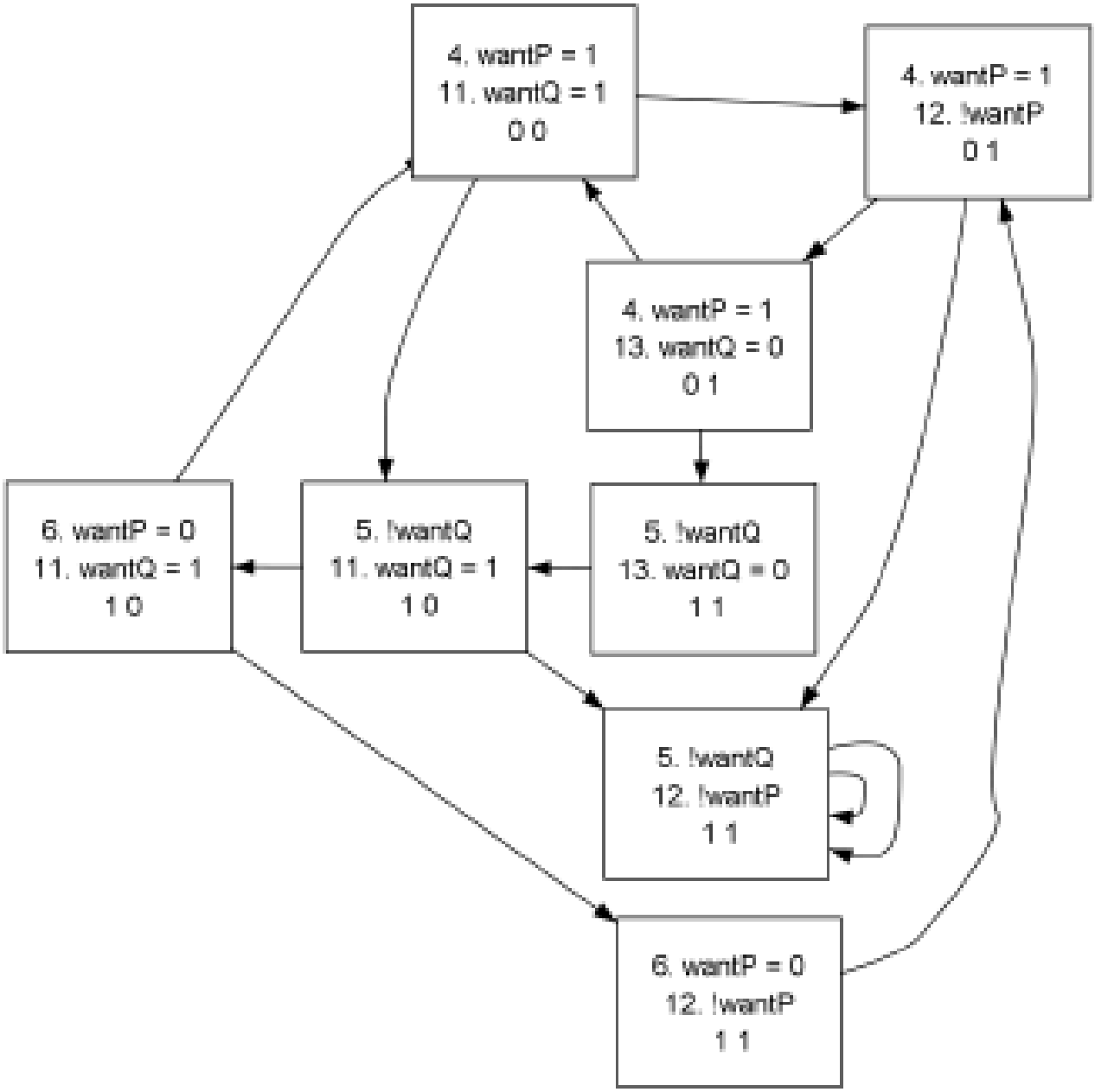
Non ancora corretto: presenza di deadlock (entrambi i processi in attesa)

Diagramma sistema di transizione a stati

- Grafo (finito) direzionato
  - vertici: stati del sistema (prodotto cartesiano control location di ogni processo, domini di tutte le variabili)
  - archi: vi è un arco da s a s' se esiste una transizione eseguibile di un processo che modifica s in s'
- Fissati uno o più stati iniziali, uno stato è raggiungibile se è esplorato in una computazione del programma
- Solitamente, la taglia del sottografo degli stati raggiungibili è molto più piccolo del grafo di transizione

Diagramma di:

```
1.  bool wantP = false,
2.           wantQ = false;
3.  active proctype P() {
4.     do ::  wantP = true;
5.              !wantQ;
6.             wantP = false
7.     od
8.  }
10. active proctype Q() {
11.    do ::  wantQ = true;
12.       !wantP;
13.       wantQ = false
14.    od
15.  }
```



Osservazioni

- Nel programma usato per il diagramma abbiamo omesso le printf
  - non modificano lo stato del programma, analogo a skip
  - si possono omettere senza alterare risultato verifica
- Numero stati totale programma: 3*3*2*2 = 36
- Numero stati raggiungibili: 8
- Mutua esclusione vale sse nessuno stato del tipo (6. wantP=false, 13. wantQ=false, x,y) è raggiungibile
- Deadlock coincide con un vertice pozzo che non corrisponde ad uno stato finale
- Nel nostro caso il programma chiaramente rispetta mutua esclusione, ma non è deadlock free:
  - (5. !wantQ,12. !wantP,1,1) è raggiungibile

Semplice algoritmo di reachability

1. Inizializza Reach con stati iniziali marcati unexplored
2. Per ogni stato unexplored in Reach: esegui tutte le transizioni eseguibili e aggiungi gli stati così visitati in Reach. Se non erano già in Reach, marcali unexplored
3. Termina quando tutti gli stati in Reach sono marcati explored

Semafori in Promela

- Implementiamo un semaforo con una variabile sem di tipo byte
- La variabile viene aggiornata con codice nei processi che ne fanno uso
- wait(sem) è implementato testando se sem>0 e quindi decrementando sem
  - per il corretto funzionamento del semaforo le due istruzioni devono essere eseguite una dietro l'altra senza interleaving con istruzioni di altri processi
  - usiamo atomic
- signal(sem) consiste in un semplice incremento
  - non presenta problemi

Programma “critical section” con semafori

```
byte sem = 1;
active proctype P() {
  do
  :: printf("Noncritical section P\n");
     atomic {          // wait(sem)
        sem>0;
        sem--
     }
     printf("Critical section P\n");
     sem++             // signal(sem)
  od
}
```

```
active proctype Q() {
  do
  ::  printf("Noncritical section Q\n");
      atomic {         // wait(sem)
         sem>0;
         sem--
      }
      printf("Critical section Q\n");
      sem++            // signal(sem)
  od
}
```

Trasferimento da sorgente a destinazione

```
byte input, output;

active proctype Source() {
 byte i=1;
  do :: ( i > 10 ) -> break
        :: else ->  input == 0;  input = i;  i++
  od
}
active proctype Destination() {
  do  :: output != 0;
           printf("Output = %d\n", output);
           output = 0
  od
}
```

```
active proctype Relay() {
  do
  :: atomic {
        input != 0;
        output == 0;
        if
        :: output = input
        :: skip
        fi
     }
     input = 0
  od
}
```

Uso di atomic in Relay

- Esecuzione del blocco atomic:
  - Relay attende che un nuovo input sia stato prodotto e che l'output sia stato consumato
  - quindi nonderterministicamente trasferisce il valore dell'input sull'output oppure lo ingnora
- Rimpiazziamo atomic con d_step: due problemi
  - nessun nondeterminismo, nessun input viene scartato
  - non è consentito bloccare il processo con la seconda espressione (output == 0)

Limitazioni con d_step

- Eccetto per la prima linea del blocco (guardia), le istruzioni devono essere non-blocking
  - in particolare, non possono essere espressioni condizionali
- Non è consentito avere jump nella sequenza o dalla sequenza con break e goto
- Non determinismo risolto sempre scegliendo la prima alternativa eseguibile

Nondeterminismo

- Concetto fondamentale in verifica
- In uno stato ci possono essere diverse transizioni eseguibili
  - succede naturalmente nei sistemi concorrenti
- Può dipendere da mancanza di dettaglio (in astrazione)
- Da non confondere con probabilità
  - non vi è un concetto di probabilità tra le diverse alternative
  - anche in sistemi dipendenti da generatori casuali in verifica occorre esplorare tutte le possibilità con probabilità non nulla e quindi non determinismo è il concetto giusto

Uso nondeterminismo per generare valori

- Generare nondeterministicamente due valori

```
active proctype Client(){
        if
        :: true -> request = 1
        :: true -> request = 2
        fi
}
```

- Non è necessaria la guardia true

```
        if
        :: request = 1
        :: request = 2
        fi
```

- per generare una sequenza di lunghezza infinita

```
        do
        :: request = 1
        :: request = 2
        od
```

- in un range arbitrario

```
        do
        :: number < HIGH -> number ++
        :: break
        od
```

(nota i numeri generati non sono equiprobabili, ma sono tutti possibili)

Terminazione di processi

- Non esente da deadlock
- Quando in stato con wantP=true e wantQ=true (dopo i rispettivi assegnamenti),

  nessuna mossa possibile
- terminazione per timeout:

  “invalid end state”
- Per def. un processo che termina, deve terminare dopo aver eseguito la sua ultima istruzione

```
bool wantP = false, wantQ = false;
active proctype P() {
  do ::   wantP = true;
          !wantQ;
          wantP = false
  od
}
active proctype Q() {
  do ::   wantQ = true;
          !wantP;
          wantQ = false
  od
}
```

Sistema client-server

- Se si esegue verifica, il programma termina segnalando un *invalid end state*
- Il numero dei passi di Client fissato, e quindi Server1 e Server2 restano in attesa di richiesta una volta che Client ha terminato
- Ma questo non è un errore:
  - è un comportamento normale che i server restino in attesa della prossima richiesta

```
byte request = 0;
active proctype Server1() {
    do
    :: request == 1;   request = 0
    od
}
active proctype Server2() {
    do
    ::request == 2; request=0
    od
}
active proctype Client() {
    request = 1;
    request == 0;
    request = 2;
    request == 0
}
```

Notazione per “valid end states”

- Si può inserire un’etichetta il cui nome cominci per end per indicare che si tratta di un valid end point anche se non è l’ultima istruzione

```
active proctype Server1(){
  endserver:
      do
      :: request == 1 -> ..
      od
}
```

Ordine di terminazione

- Un processo termina dopo che esegue l'ultima istruzione
- Un processo è attivo finchè muore
- I processi istanziati con active proctype vengono istanziati nell'ordine in cui sono scritti
- SPIN usa uno stack per l'allocazione dei processi
  - un processo può morire sse è l'ultimo processo attivato
- Provare programma con finished==2 (terminano tutti i processi) e finished==3 (nessun processo termina; timeout per tutti)

```
byte request = 0;  byte finished = 0;
active proctype Server1() {
  request == 1;   request = 0;
  finished++
}
active proctype Server2() {
  request == 2;  request = 0;
  finished++
}
active proctype Client() {
  request = 1; request == 0;
  request = 2;  request == 0;
  finished == 2;
}
```

Windows NT Bluetooth driver (modello)

- Usa le seguenti variabili:
  - pendingIo: it counts the number of threads that are currently executing in the driver. It is initialized to one in the constructor, increased by one when a new thread enters the driver, and decreased by one when a thread leaves.
  - stopFlag: it becomes true when a thread tries to stop the driver.
  - stopEvent: it models a stopping event, fired when pendingIo becomes zero.The field is initialized to false and set to true when the event happens.
  - stopped: it is introduced only to check a safety property. Initially false, it is set to true when the driver is successfully stopped.

Windows NT Bluetooth driver (modello)

- Descrizione del funzionamento del driver:
  - The driver has two types of threads: stoppers and adders.
  - A stopper calls method stop to halt the driver. It first sets stopFlag to true before decrementing pendingIo via a call to dec. The method dec fires stopEvent when pendingIo is zero.
  - An adder calls the method add to perform I/O in the driver. It calls the method inc to increment pendingIo; inc returns a successful status if stopFlag is not yet set. It then asserts that stopped is false before start performing I/O in the driver. The adder decrements pendingIo before exiting.

Bluetooth driver versione 1

int pendingIo; boolean stopFlag, stopEvent, stopped;
init{ pendingIo = 1; stopFlag = stopEvent = stopped = false; }

```
static void add() {
    int status = inc();
    if (status > 0) {
        assert(!stopped);
          // Performs I/O
    }
    dec();
}

static int inc() {
    if (stopFlag) {return -1;}
    atomic{ pendingIo++;}
    return 1;
}
```

```
static void stop() {
    stopFlag = true;    dec();
    while (!stopEvent) {}
    stopped = true;
}

static void dec() {
    int pio;
    atomic{
        pendingIo--;
        pio = pendingIo;
    }
    if (pio == 0)    stopEvent = true;
}
```

Esercizio

- Modellare la versione 1 del Bluetooth driver in Promela e verificarne correttezza (rispetto alle asserzioni)
  - Una configurazione ammissibile presenta un processo stopper e un numero arbitrario di processi adder (considerare i casi con uno, due e tre adder)
- Se la verifica di correttezza rivela un errore, provare a fornire una nuova versione (non banale) del driver corretta

Bluetooth driver versione 2 (inc modificato)

```
int pendingIo; boolean stopFlag, stopEvent, stopped;
init { pendingIo = 1; stopFlag = stopEvent = stopped = false; }
```

```
static void add() {
    int status = inc();
    if (status > 0) {
        assert(!stopped);
          // Performs I/O
    }
    dec();
}

static int inc() {
    int status;
    atomic{ pendingIo++;}
    if (stopFlag) { dec();  status = -1;}
    else status = 1;
    return status;
}
```

```
static void stop() {
    stopFlag = true;    dec();
    while (!stopEvent) {}
    stopped = true;
}

static void dec() {
    int pio;
    atomic {
        pendingIo--;
        pio = pendingIo;
    }
    if (pio == 0)    stopEvent = true;
}
```

Esercizio

- Modellare la versione 2 del Bluetooth driver in Promela e verificarne correttezza (rispetto alle asserzioni) nelle configurazioni ammissibili con uno, due e tre adder, rispettivamente

- Se la verifica di correttezza rivela un errore, provare a fornire una nuova versione (non banale) del driver corretta

Bluetooth driver versione 3 (add modificato)

```
int pendingIo; boolean stopFlag, stopEvent, stopped;
init { pendingIo = 1; stopFlag = stopEvent = stopped = false; }
```

```
static void add() {
    int status = inc();
    if (status > 0) {
        assert(!stopped);
          // Performs I/O
        dec(d);
    }
}

static int inc() {
    int status;
    atomic{ pendingIo++;}
    if (stopFlag) { dec();  status = -1;}
    else status = 1;
    return status;
}
```

```
static void stop() {
    stopFlag = true;    dec();
    while (stopEvent) {}
    d.stopped = true;
}

static void dec() {
    int pio;
    atomic {
        pendingIo--;
        pio = pendingIo;
    }
    if (pio == 0)    stopEvent = true;
}
```

Esercizio

- Modellare la versione 3 del Bluetooth driver in Promela e verificarne la correttezza (rispetto alle asserzioni) nelle configurazioni ammissibili con uno, due e tre adder, rispettivamente

- Se si rilassa il requisito di ammissibilità in modo che possono esserci anche più stopper, la versione 3 è corretta?

Verifica con Logica Temporale

Specifiche proprietà in SPIN

- Asserzioni specificano proprietà per punti fissati del programma (per fissate control locations)
  - istruzioni che testano se una data condizione è verificata ed in caso contrario mandano in uno stato di errore

- Molte proprietà di correttezza si riferiscono al comportamento del sistema in generale non solo a specifiche locazioni

Mutua esclusione

- Algoritmi distribuiti token-passing
  - Mutua esclusione viene realizzata passando un token (il processo che detiene il token è autorizzato ad utilizzare la risorsa)
  - Proprietà: *in ogni stato di ogni computazione c'è al più un token*
- Algoritmi che usano contatore (critical)
  - Proprietà: *critical <=1 è un'invariante di ogni computazione*

Assenza di deadlock

- Nessuna istruzione eseguibile
  - *In ogni stato di ogni computazione:*

    nessuna istruzione eseguibile implica che

    termine processo oppure locazione etichettata finale (label che comincia per end)
- Accesso alla CS
  - *In ogni stato di ogni computazione:*

    se almeno un processo tenta di accedere alla CS

    allora *uno di quelli che tentano prima o poi ci riesce*

Altre invarianti globali

- Indici di un array (in Java)
  - *In ogni stato di ogni computazione:*

  l'indice di ogni array è compreso tra 0 e la lunghezza (quest'ultima esclusa)

- Assenza di starvation
  - *In ogni stato di ogni computazione:*

    se un processo tenta di accedere CS

    allora prima o poi ci riesce

Linear Temporal Logic (LTL)

- Formalismo sufficiente a specificare tutte le proprietà viste
- In generale, permette di esprimere proprietà che fanno riferimento all'ordine temporale con cui si verificano gli eventi
- Tipici operatori temporali:
  - "prima o poi", "alla fine" (*eventually*, ◊)
  - "sempre" (*always*, □)
  - "prossimo istante" (*next*, **X**)
  - .....

Sintassi LTL

- Una formula f è:
  - una combinazione booleana di proposizioni atomiche
  - $\neg f'$ (negazione di f')
  - $f_1 \wedge f_2$ (AND di f_1 e f_2)
  - $f_1 \vee f_2$ (OR di f_1 e f_2)
  - $\Diamond f'$ (eventually f')
  - $\Box f'$ (always f')
  - $\bigcirc f'$ (next f')
  - $f_1 U f_2$ (f_1 until f_2)

simbolo in SPIN:	
	&&
	||
	<>
	[]
	X
	U

Notazione e osservazioni

- Gli operatori logici -> e <-> sono consentiti in SPIN e sono definiti attraverso le equivalenze:
  - (p->q) ≡ (!p || q) e (p<->q) ≡ (p->q) && (q->p)
- LTL è chiusa rispetto alla negazione
  - !<>p ≡ [ ]!p, !(pUq) ≡ [ ] !q || (!q U (!p && !q)), !Xp ≡ X!p
- SPIN ha un operatore duale per until:
  - pVq ≡ !(!p U !q)
- Altre equivalenze:
  - <>p ≡ (true U p), pUq ≡ q || (p && X(pUq))

LTL model-checking con SPIN

- SPIN accetta proprietà di correttezza espresse in LTL e risolve LTL model-checking
  - le strutture di kripke sono i diagrammi di transizione dei programmi Promela
  - le proposizioni atomiche sono: (nomi iniziano per minuscola)
    - espressioni condizionali tra parentesi tonde (nota che per condizioni che fanno uso di variabili locali occorre specificare il processo, es. P:x)
    - variabili booleane
    - macro che assegnano un nome a espressioni condizionali (definite con #define come in C)
    - etichette di control locations (es. P@cs, dove cs è un etichetta usata nel processo P)

Procedimento di verifica

- una formula LTL φ viene tradotta in una never claim per φ
  - essenzialmente un programma Promela congiuntamente ad una condizione di Büchi (stati di accettazione definiti da etichette inizianti per accept)
  - opzione `-f` "ltl-formula" (`-F` mette never claim in file .ltl)
  - opzione `-N` file.ltl se never claim in file.ltl

- lo spazio degli stati del programma da verificare e della never claim vengono quindi esplorati in parallelo alla ricerca di errori (esplorazione del prodotto cartesiano dei rispettivi sistemi di transizione)

- errore determinato dalla computazione che viene accettata in base alla condizione di Büchi della never claim

Esempio di never claim

- la never claim per ![]mutex è:

```
never  {    /* !([] mutex) */
T0_init:
        if      :: (! ((mutex)) -> goto accept_all
                :: (1) -> goto T0_init
        fi;
accept_all:   skip
}
```

Safety properties

- una computazione è *safe* se "tutto ciò che accade è buono"
- in altri termini "niente di cattivo accade"
- in LTL: [] good ≡ !<>!good ≡!<>bad

```
int critical;     bool wantP,wantQ;

#define mutex (critical <=1)

active proctype P() {
        do  :: wantP = true;
                !wantQ;
                critical++;
                critical--;
                wantP = false;
        od
}
```

```
FORMULA LTL: []mutex

active proctype Q() {
        do
        :: wantQ = true;
                !wantP;
                critical++;
                critical--;
                wantQ = false;
        od
}
```

Altro esempio (non usa variabili aggiuntive)

```
bool wantP,wantQ;

#define mutex !(P@cs && Q@cs)

active proctype P() {
        do
        :: wantP = true;
                !wantQ;
    cs: wantP = false;
        od
}
```

```
FORMULA LTL: [] mutex


active proctype Q() {
        do
        :: wantQ = true;
                !wantP;
    cs: wantQ = false;
        od
}
```

Altro esempio

```
bool wantP,wantQ,csp,csq;

active proctype P() {
        do
        :: wantP = true;
                !wantQ;
                csp = true;
                csp = false;
                wantP = false;
        od
}
```

```
FORMULA LTL: []!(csp && csq)


active proctype Q() {
        do
        :: wantQ = true;
                !wantP;
                csq = true;
                csq = false;
                wantQ = false;
        od
}
```

Liveness

- "qualcosa di buono prima o poi accade nella computazione"
- nell'esempio precedente, <>csp è vero se e solo se P entra nella sua critical section ad un certo punto della computazione
- falsificazione richiede computazione infinita
  - contro-esempio di lunghezza infinita
- Proprietà: se esiste un contro-esempio ne esiste uno finitamente rappresentabile nella forma di un cammino terminante in un ciclo (cappio)
- Verifica con SPIN cerca per un "accepting cycle"
  - usare con `pan` opzione `-a`

Critical section con starvation

```
bool wantP = false, wantQ = false;
bool csp = false;

active proctype Q() {
      do
      :: wantQ = true;
         do
          :: wantP -> wantQ = false;
             wantQ = true
          :: else -> break
         od;
         wantQ = false
      od
}
```

```
FORMULA LTL: <>csp

active proctype P() {
      do :: wantP = true;
            do :: wantQ -> wantP = false;
                  wantP = true
               :: else  -> break
            od;
            csp = false;
            csp = true;
            wantP = false
      od
}
```

Contro-esempio calcolato da SPIN

- "Start cycle" è utilizzato da SPIN per denotare l'inizio del ciclo nel contro-esempio

```
<<<<<START OF CYCLE>>>>>
1 Q     22  wantQ = 1
Process Statement          wantQ
1 Q     26  else             1
1 Q     28  wantQ = 0        1
```

Fairness

- Il contro-esempio precedente ha un problema:
  - al processo P non viene mai consentito di eseguire un'istruzione anche se eseguibile
  - non viene data a P una possibilità
  - computazione manca di "fairness«

- Una computazione è *weakly fair* se:

se un'istruzione è sempre eseguibile allora prima o poi verrà eseguita

Attivazione weak fairness in SPIN

- Usare opzione `-f` insieme a `-a` in pan per selezionare weak fairness
- contro-esempio

```
1 Q     22  wantQ = 1
Process Statement         wantQ
1 Q     26  else                1
                    ⋮
Process Statement         wantP      wantQ
1 Q     22  wantQ = 1               1          0
1 Q     24  wantP           1          1
0 P     10  wantQ           1          1
<<<<<START OF CYCLE>>>>>
1 Q       24  wantQ = 0        1          1
                    ⋮
0 P       10  wantQ             1          1
```

Limitazione a computazioni weakly fair

- programma seguente termina sempre se e solo se ci si restringe a computazioni weakly fair

```
int n = 0;
bool flag = false;

active proctype q() {
        flag = true
}
```

```
active proctype p() {
        do
        :: flag -> break;
        :: else -> n = 1 -
n;
        od
}
```

Fallimento nella non-critical section

```
#define mutex !(P@cs && Q@cs)
#define live (Q@cs)
byte  turn = 1;


active proctype Q() {
            do
            :: (turn == 2);
         cs:  turn = 1
            od
}
```

```
FORMULA <> live

active proctype P() {
            do
            :: if
                :: true
                :: true -> false
               fi;
              (turn == 1);
         cs: turn = 2
            od
}
```

Risultato esempio precedente (ciclo vuoto)

```
spin: couldn't find claim 2 (ignored)
0 P     17  1
<<<<<START OF CYCLE>>>>>
spin: trail ends after 5 steps
#processes: 2
  5:     proc  1 (Q) first-ncs.pml:25 (state 3)
  5:     proc  0 (P) first-ncs.pml:17 (state 3)
2 processes created
Exit-Status 0
```

Algoritmo di mutua esclusione di Peterson (con failure in NCS)

```
#define ptry P@try
#define qcs  Q@cs
#define pcs  P@cs

bool   wantP, wantQ;
byte   last = 1;

active proctype Q() {
   do ::  wantQ = true;
          last = 2;
try:   (wantP == false) || (last == 1);
cs:    wantQ = false
   od
}
```

```
FORMULA: []<> qcs

active proctype P() {
   do
   :: if
      :: true
      :: true -> false
      fi;
      wantP = true;
      last = 1;
try:   (wantQ == false) || (last == 2);
cs:    wantP = false
   od
}
```

Proprietà in LTL

- Latching: da un certo punto in poi una proprietà sempre vera
  - <>[ ] prop
- Ad es., un sistema può essere progettato in modo che una volta che una componente ha un fallimento allora tutte le sue variabili sono azzerate
  - <>failQ -> <>[ ] !wantQ
- Infinitely often: una proprietà vera un numero infinito di volte, non necessariamente sempre
  - [ ]<>prop
- Ad es., se una risorsa è richiesta infinite volte allora deve essere attribuita infinite volte
  - [ ]<> wantP -> [ ]<> P@cs

Proprietà in LTL

- Precedence: q non diventa vera prima di p
  - !q U p
- One-bounded overtaking: se il processo P tenta di accedere alla CS, allora processo Q può accedere alla CS al più una volta prima di P
  - [ ](ptry ->

    (!Q@cs U (Q@cs U (!Q@cs U P@cs))))

Operatore Next

- utilità limitata in sistemi concorrenti o distribuiti
  - concetto di tempo rilassato
  - in un sistema client-server, siamo interessati a sapere se un cliente riceve un servizio ma non interessa se lo riceve nel prossimo stato o dopo 5 stati
- non è *stutter-invariant*: se viene duplicato uno stato di un modello di una formula contenente X non siamo garantiti di avere ancora un modello della formula
- gli algoritmi di SPIN sono più efficienti se si usano specifiche *stutter-invariant*

Esercizi

- Modellare in LTL le seguenti proprietà:
  - Response property: ogni volta che una risorsa è richiesta da un processo alla fine deve essere attribuita
  - Time response property: ogni volta che una risorsa è richiesta da un processo deve essere attribuita entro 5 passi
  - Pattern: le proprietà p, q e r devono essere vere in questo ordine e in maniera mutuamente esclusiva
  - Release: p deve restare vera fino a quando q diventa vera (nota q può non diventare mai vera)