Università degli Studi “Sapienza” di Roma

Corso di laurea specialistica in Ingegneria Informatica

SAPIENZA
UNIVERSITÀ DI ROMA

Tesina di Metodi formali nell'Ingegneria del Software

Modellazione e verifica formale di alcune proprietà di un algoritmo token based per la mutua esclusione in ambiente distribuito

anno accademico 2006/2007
Professor Toni Mancini

autore:

Francesco Barbaria
matricola: 795139

INDICE

1. Obiettivo e presentazione dell'algoritmo.. pag.3
   1.1 Introduzione... pag.3
   1.2 Scenario di funzionamento.. pag.4
2. Diagramma UML degli stati e delle transizioni.. pag.7
   2.1 Diagramma UML del processo “client”.. pag.7
   2.2 Diagramma UML del processo “coordinatore”.. pag.8
3. Modellazione in SPIN... pag.11
   3.1 Implementazione del processo “coordinatore”.. pag.13
   3.2 Implementazione del processo “client”.. pag.17
   3.3 Implementazione del generatore ed inizializzazione del sistema.............. pag.19
4. Utilizzo di Xspin per la simulazione e la verifica di alcune proprietà................ pag.20
   4.1 Verifica della proprietà di Safety... pag.23
   4.2 Verifica della proprietà di Liveness... pag.26
   4.3 Identificazione di un controesempio per una proprietà non soddisfatta..... pag.29
5. Supporto per N client.. pag.32
   5.1 Estensione ad un sistema a 3 client.. pag.33
6. Conclusioni e bibliografia.. pag.35

1. Obiettivo e presentazione dell'algoritmo

Scopo di questa tesina è modellare il comportamento di un algoritmo per la mutua esclusione in ambiente distribuito.

1.1 Introduzione

La **mutua esclusione** è un problema centrale nei sistemi distribuiti nei quali diversi processi devono accedere ad una stessa risorsa condivisa. Per regolare questo accesso e fare in modo che non esistano due processi che contemporaneamente possano andare ad utilizzare la risorsa condivisa sono stati ideati diversi algoritmi. L'algoritmo che abbiamo scelto di studiare appartiene alla classe degli algoritmi "*Token Based*", ovvero una particolare classe di algoritmi che utilizzano una risorsa aggiuntiva ed unica (il **token**, appunto) per stabilire chi abbia il diritto di accedere alla risorsa condivisa.

In particolare abbiamo scelto l'algoritmo *Token Based centralizzato*.
Il suo funzionamento è il seguente.

Per prima cosa esistono due tipi di processo: i processi "***client***" ed un processo "***coordinatore***". L'idea di fondo è molto intuitiva: il coordinatore gestisce l'invio del token.

Quando un processo client vuole accedere alla risorsa condivisa, invia una richiesta al coordinatore e, se quest'ultima è ritenuta elegibile, il coordinatore invia il token al processo. Una volta in possesso del token il processo può accedere alla risorsa condivisa e, una volta terminata la "*sezione critica*", invia di nuovo il token al coordinatore.

Per stabilire "l'elegibilità" di una richiesta, bisogna introdurre una risorsa addizionale fondamentale per il funzionamento di tutto l'algoritmo: i **vector clock**. Ogni processo "client" ed il coordinatore mantengono in memoria un vector clock, ovvero un vettore di dimensione pari ad N (dove N è il numero di processi "client" attivi nel sistema) ed inizialmente settato a zero.

Ogni volta che un processo vuole entrare in sezione critica (ovvero vuole accedere alla risorsa condivisa) aggiorna il suo vector clock in corrispondenza della sua posizione, incrementandolo di uno. Successivamente lo invia al coordinatore nel messaggio di request e contemporaneamente lo invia a tutti gli altri N-1 processi client attivi.
Alla ricezione di un messaggio da parte di un altro processo, si aggiorna ogni riga del proprio vector clock con il valore massimo tra il vector clock memorizzato in precedenza e quello ricevuto dal processo.

Partendo da questo presupposto, il comportamento del coordinatore è semplice: alla ricezione di una richiesta da parte di un processo X, il coordinatore verifica che il vector clock appena ricevuto da X sia >= (e non strettamente maggiore!) del vector clock memorizzato. Vale a dire: *tutte le righe dei due vector clock devono coincidere, tranne (ovviamente) la riga X*.

Se questa condizione è soddisfatta, il coordinatore invia il token al processo X, altrimenti memorizza la richiesta come "pendente", in attesa che diventi elegibile e soddisfi la condizione descritta in precedenza.

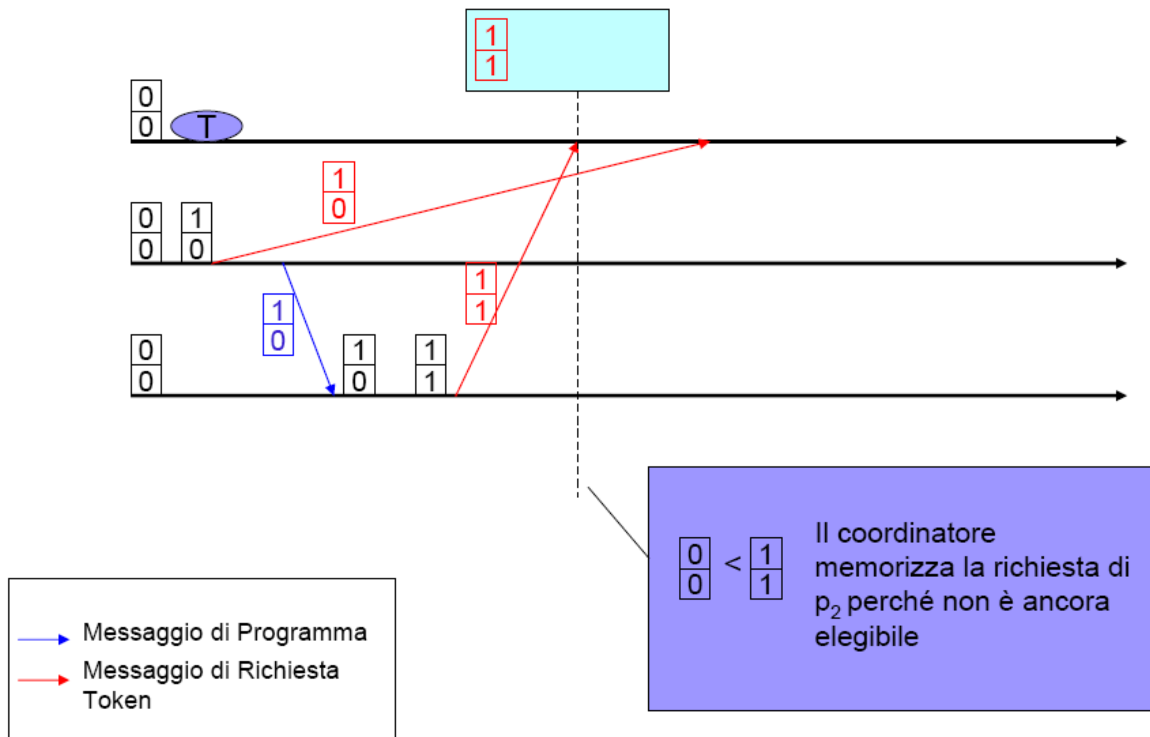
A fronte delle assunzioni di assenza di guasti da parte dei processi e di canali affidabili (ovvero niente messaggi persi, modificati o spuri) questo algoritmo garantisce ovviamente **Safety**, ovvero la mutua esclusione, grazie alla condizione di unicità del token (non possono esistere due processi che siano contemporaneamente in possesso del token)

Inoltre è garantita la **Liveness**: se un processo fa richiesta di accesso alla risorsa condivisa verrà sicuramente servito (non c'è **starvation** né **deadlock**)

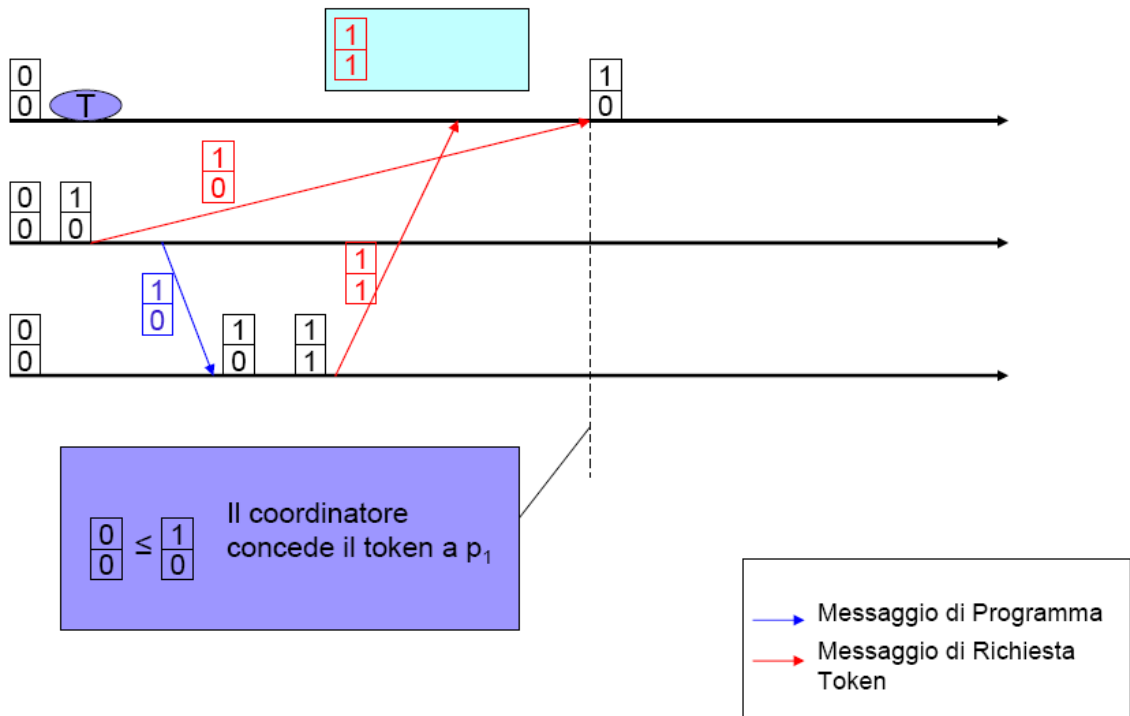
Scopo di questa tesina è proprio modellare il funzionamento di questo algoritmo distribuito e dimostrare, con l'ausilio degli strumenti messi a disposizione dal corso di metodi formali per l'ingegneria del software, la validità di queste due proprietà.

1.2 Scenario di funzionamento.

Per maggiore completezza, illustriamo adesso, con l'aiuto di alcune figure, uno scenario di funzionamento dell'algoritmo. In questo esempio abbiamo un coordinatore e due processi client che vogliono accedere alla risorsa condivisa. Il Vector clock avrà quindi dimensione 2 e memorizzerà nella riga 0 l'avanzamento del processo p1, nella riga 1 l'avanzamento del processo p2.

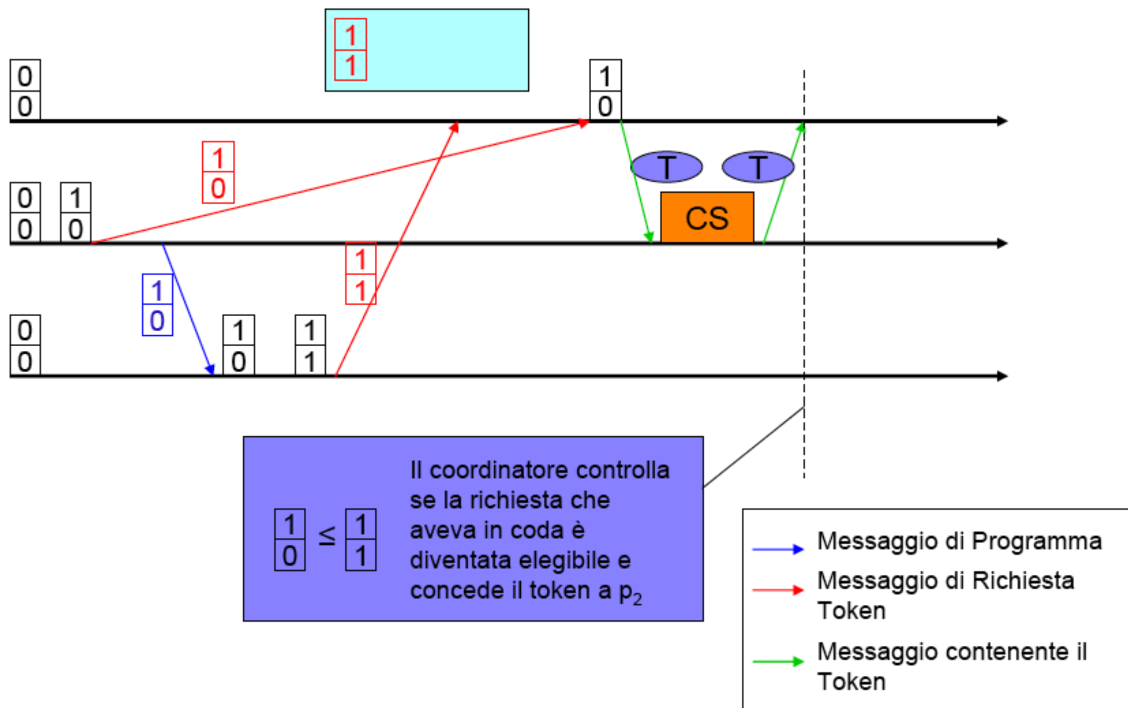


In questa prima figura è rappresentato un ritardo nella comunicazione tra il primo processo ed il coordinatore. Quando il coordinatore riceve la richiesta da parte del secondo processo, la considera non elegibile perché il valore del vector clock ricevuto nella prima posizione è maggiore del valore memorizzato. Questo vuol dire che c'è un'altra richiesta (avvenuta prima) che non è ancora arrivata al coordinatore. Per questo motivo la richiesta di p2 viene memorizzata ed il token non viene inviato.

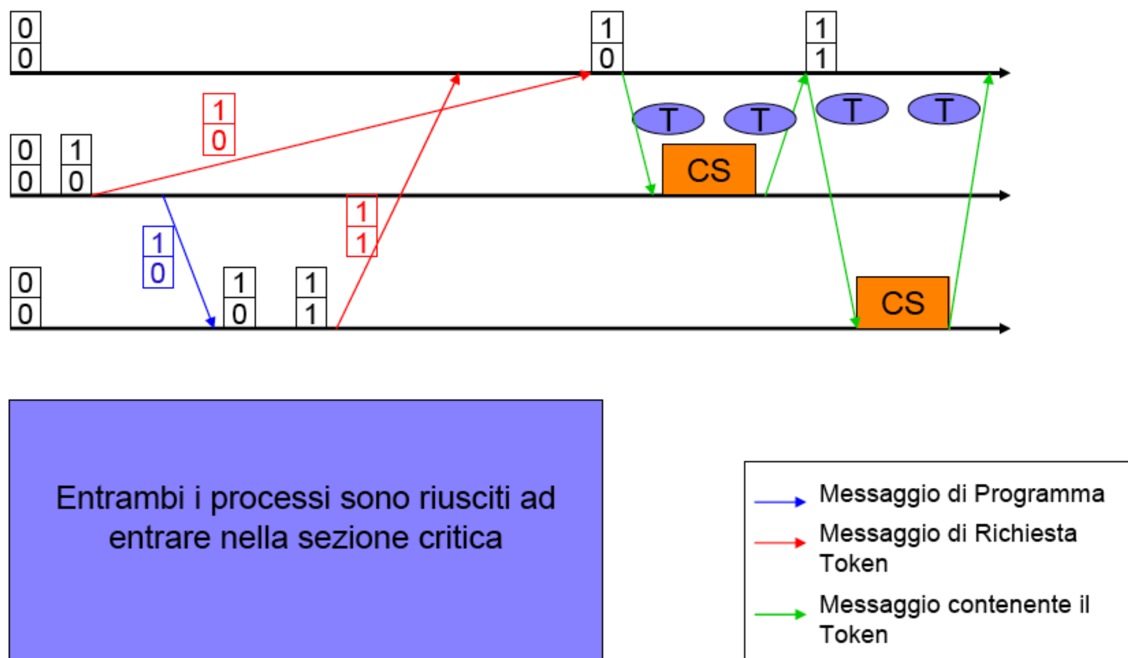


Dopo aver ricevuto la richiesta di p1, il coordinatore la considera elegibile: il vector clock ricevuto ha i valori uguali a quelli memorizzati su tutte le righe tranne una (la riga 0, relativa al processo richiedente).

Il coordinatore invia quindi il token al processo richiedente ed aggiorna il proprio vector clock.



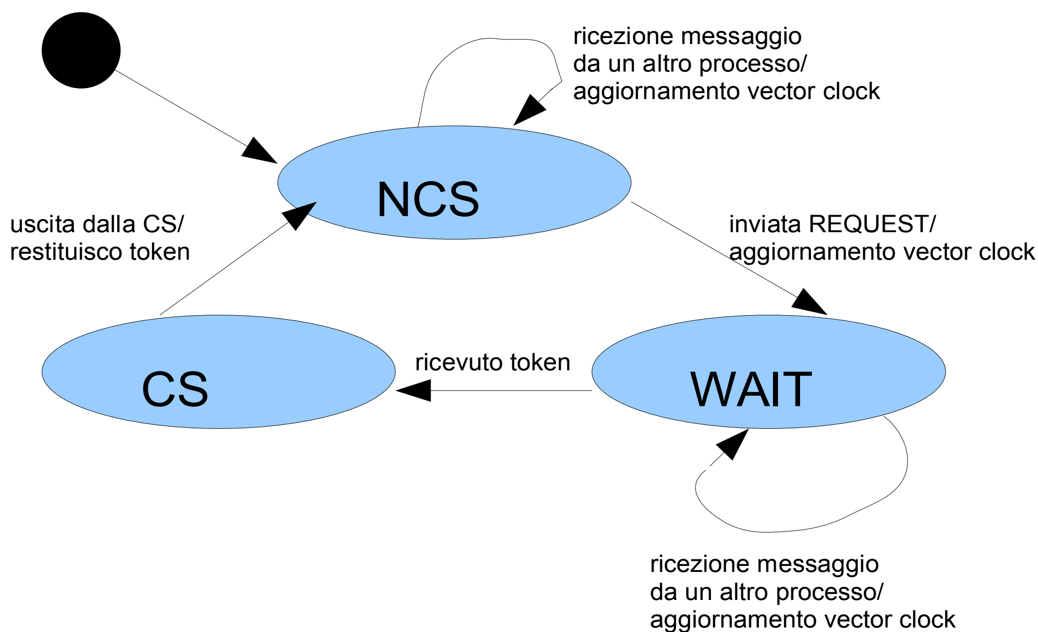
Quando il processo p1 esce dalla CS, invia il token nuovamente al coordinatore e quest'ultimo può verificare se la richiesta memorizzata in coda (da parte di p2) sia nel frattempo divenuta elegibile. Come si può facilmente verificare, è così. Il coordinatore invia quindi il token al processo p2 ed aggiorna nuovamente il proprio vector clock:



2. Diagramma UML degli stati e delle transizioni

Per prima cosa dobbiamo modellare, con un diagramma UML degli stati e delle transizioni, il funzionamento delle due differenti tipologie di processo: il coordinatore ed il processo "client". E' il primo passo per verificare che l'algoritmo funziona correttamente e rispetta le proprietà che vogliamo dimostrare.

2.1 Diagramma UML del processo client



Il processo client può essere modellato con tre stati principali. Un primo stato di "*Not Critical Section*" (**NCS**) nel quale il suo funzionamento è standard. Quando un processo si trova in questo stato possono verificarsi due eventi principali: si riceve un messaggio "di servizio" da parte di un altro processo oppure si verifica la necessità di accedere alla risorsa condivisa.

Nel primo caso (ricezione di un messaggio "di servizio") l'azione che consegue è quella di aggiornare il vector clock con i valori ricevuti dal processo, ma lo stato non viene variato (si rimane in NCS).
Nel secondo caso (necessità di accesso alla risorsa condivisa) viene ugualmente aggiornato il vector clock, incrementandolo di uno nella riga opportuna, ma vengono anche eseguite le seguenti azioni: viene inviato un messaggio "di servizio" agli altri N-

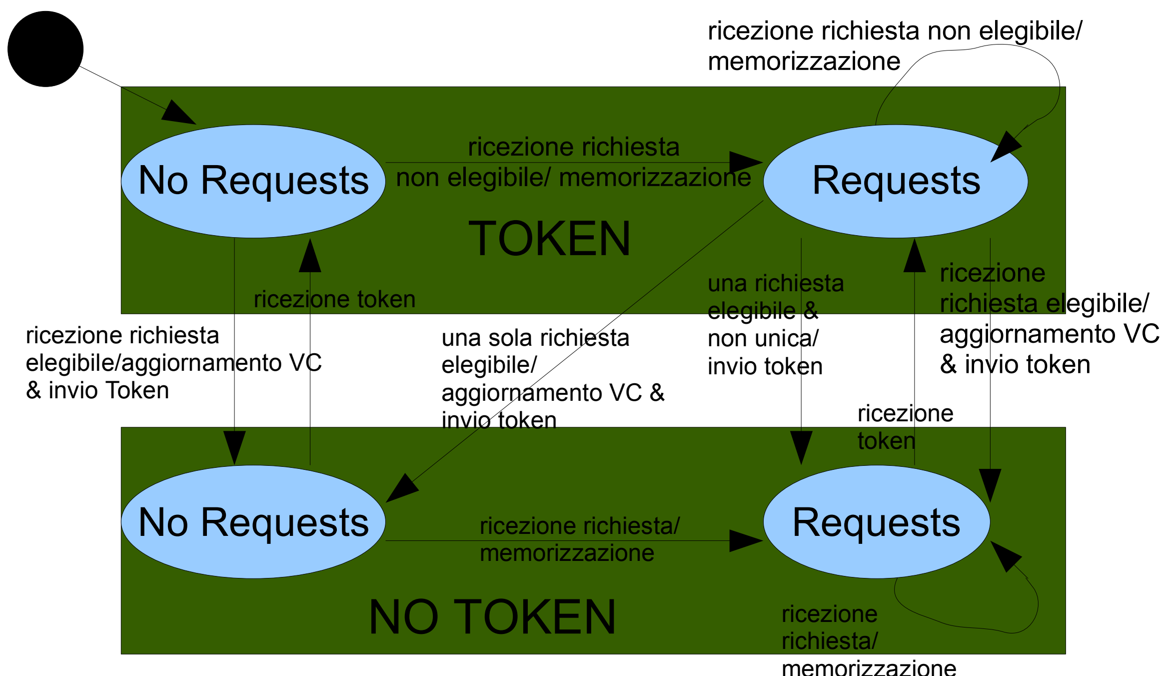
1 processi, un messaggio di richiesta al processo coordinatore ed si passa in uno stato di attesa del token (**WAIT**)

Nello stato di WAIT possono ancora avvenire due tipologie di evento: ricezione di un messaggio "di servizio" da parte di un altro processo (nel qual caso il comportamento sarà identico allo stato precedente) oppure ricezione del TOKEN.

Questo secondo evento permette il passaggio allo stato di "*Critical Section*" (**CS**), che rappresenta lo stato in cui il processo può accedere alla risorsa condivisa.

Come si vede dal diagramma, abbiamo deciso di non modellare, in questo stato, la ricezione di messaggi "di servizio". E' stata una scelta dettata per comodità, perchè abbiamo fatto l'assunzione che l'accesso alla sezione critica sia breve e che ogni processo che entri in sezione critica sia costretto – prima o poi – ad uscirne. Per questo motivo, l'unico evento che può verificarsi in questo stato è quello di operazione terminata sulla risorsa condivisa. In questo caso il processo tornerà semplicemente nello stato di NCS e continuerà il suo normale funzionamento.

2.2 Diagramma UML del processo coordinatore



Il funzionamento del processo coordinatore è leggermente più complicato.
Per comodità abbiamo deciso di rappresentare per prima cosa due "*macrostati*": uno stato in cui il coordinatore è in possesso del token ed uno stato in cui il coordinatore è in attesa della restituzione del token da parte di un processo.

All'interno di questi due macrostati sono identificabili due *sotto-stati* in qualche modo simmetrici: stato in cui sono memorizzate delle richieste "pendenti" e stato in cui non ci sono altre richieste "pendenti" in memoria.

Il coordinatore, alla sua creazione, si trova chiaramente nello stato in cui è in possesso del token e non ci sono altre richieste memorizzate ("*Token, No Requests*", **TNR**). Da qui può far fronte a due tipologie di evento: ricezione di una richiesta elegibile o ricezione di una richiesta non elegibile. Nel primo caso il coordinatore compie le seguenti azioni: aggiorna il proprio vector clock, invia il token al processo richiedente, passa nello stato di "*No Token, No Requests*" (**NTNR**). Nel secondo caso, invece, il coordinatore memorizza la richiesta e passa nello stato di "*Token, Requests*" (**TR**), ovvero lo stato in cui il coordinatore è ancora in possesso del token ma ci sono alcune richieste che dovranno essere servite.

Nello stato denominato **TR** possono avvenire ben quattro tipologie diverse di eventi: ricezione di una richiesta elegibile, ricezione di una richiesta non elegibile, elegibilità di una richiesta presente singolarmente in memoria, elegibilità di una richiesta presente non singolarmente in memoria.
Analizziamo questi quattro casi.

Nel primo caso (ricezione di una richiesta elegibile) il coordinatore invierà il token al processo richiedente (ovviamente dopo aver aggiornato il proprio vector clock) e passerà ad uno stato di "*No Token, Requests*" (**NTR**) ad indicare uno stato in cui il coordinatore non è più in possesso del token ed inoltre che sono presenti altre richieste in memoria che dovranno essere servite.

Nel secondo caso, ovvero in caso di ricezione di una richiesta non elegibile, quest'ultima verrà solamente memorizzata e lo stato rimarrà invariato.

Gli altri due eventi sono quelli più particolari.

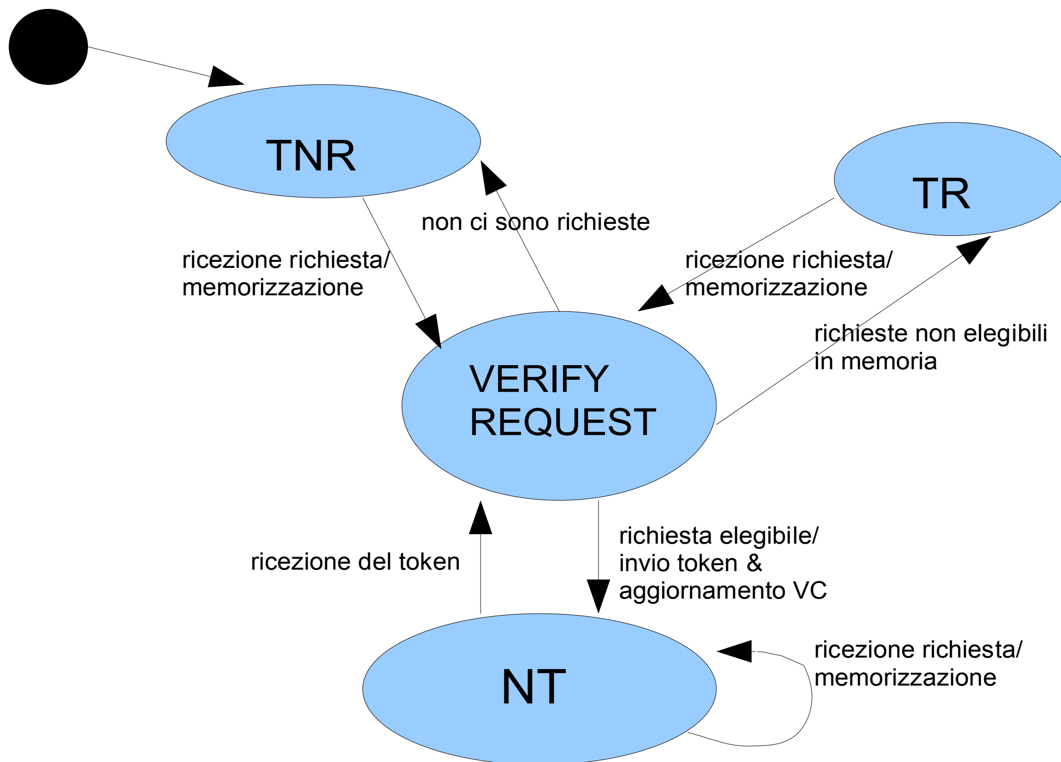
Appena entrato nello stato di **TR**, infatti, il coordinatore dovrà effettuare un controllo per verificare l'eventuale presenza di richieste pendenti diventate, nel frattempo, elegibili. In caso affermativo verrà fatto un nuovo controllo per verificare se questa richiesta pendente sia unica o meno. Nel primo caso, dopo aver inviato il token al processo (ed aver aggiornato il proprio vector clock), il coordinatore passerà ad uno stato di "*No Token, No Requests*" (**NTNR**). Nel secondo caso, invece, il coordinatore passerà allo stato di "*No Token, Requests*" (**NTR**).

Nello stato **NTR** possono avvenire due soli eventi: ricezione di una richiesta (che viene memorizzata nell'elenco delle richieste pendenti) oppure ricezione del token (e conseguente passaggio di stato allo stato di **TR**)

Infine, l'ultimo stato che rimane da analizzare è quello di **NTNR**. Anche qui possono avvenire due sole tipologie di evento: ricezione di una richiesta oppure ricezione del token. Nel primo caso, oltre a memorizzare la richiesta, il coordinatore passerà nello stato di **NTR**. Nel secondo caso, si tornerà semplicemente nello stato di **TNR**, in attesa di nuove richieste da servire.

Si noti che questo non è il solo modo per modellare il comportamento del coordinatore.

Una soluzione forse più elegante sarebbe stata la seguente:



Come si nota, in quest'altra soluzione è prevista la presenza di uno stato dedicato alla verifica dell'elegibilità delle richieste memorizzate.

Anche se quest'ultima soluzione è più elegante, abbiamo deciso di implementare la prima perché ci è sembrato che fosse una divisione in primo luogo più chiara ed inoltre che mettesse in evidenza maggiormente il concetto di “stato”. Lo stato di verifica che abbiamo rappresentato in questa seconda soluzione è in realtà uno stato “fittizio”, che il coordinatore attraversa solamente per decidere se inviare o meno il token.

Ci è sembrato invece più corretto incorporare questa verifica all'interno dello stato “TR” e da lì generare i quattro diversi eventi.

3. Modellazione in SPIN

Dopo aver modellato in UML gli stati e le transizioni del nostro sistema, dobbiamo utilizzare un model checker per verificarne le varie proprietà. I tool più indicati per questo tipo di lavoro sono **nuSMV** e **SPIN**.

Il primo è stato analizzato e studiato durante il corso di Metodi Formali nell'ingegneria del Software, mentre il secondo è un model checker pensato proprio per verificare protocolli di comunicazione come il nostro. Tra le altre caratteristiche, presenta la possibilità di creare canali di comunicazione punto-punto a differenza di nuSMV nel quale l'unico metodo a disposizione per simulare l'invio e la ricezione di messaggi tra processi sono i riferimenti condivisi.

Inoltre SPIN permette di gestire comodamente strutture dati più complesse come gli array, dei quali già sappiamo che faremo un abbondante uso.

Infine, proprio perché SPIN nasce come verificatore per la comunicazione tra processi, presenta diverse opzioni per verificare alcune proprietà essenziali di questo tipo di protocolli (rilevamento di deadlock, starvation ecc...)

Per questo motivo abbiamo pensato che fosse più conveniente utilizzare un model checker diverso da quello già studiato, ma che fosse pensato apposta per studiare protocolli come il nostro.

Per utilizzare SPIN dobbiamo per prima cosa definire formalmente il sistema attraverso il linguaggio di modellazione Promela. Come vedremo, la sua sintassi è molto intuitiva (C-like) ed è quindi molto facile gestire le strutture dati e le varie funzioni che dovremo implementare.

Per praticità abbiamo deciso di sviluppare un sistema con un coordinatore e due soli processi "client". Estendere il sistema ad un modello a tre client è senza dubbio possibile, ma richiederebbe un appesantimento del codice senza modificarne, di fatto, la logica interna. Per questo motivo abbiamo pensato che realizzare un sistema a due client fosse sufficiente.

Oltre a questi tre processi si è reso necessario implementare un ulteriore processo "generatore di sezione critica". Una delle limitazioni di SPIN rispetto a nuSMV, infatti, è che la generazione di eventi casuali - come appunto la necessità da parte di un processo di entrare in sezione critica - deve essere implementata manualmente. Uno dei metodi più usati per fare questo è ricorrere ad una funzione dedicata (nel nostro caso denominata "generator") che si occupa di inviare in maniera "random" messaggi di tipo **ENTERCS** ai due processi. Come vedremo, alla ricezione di questi ultimi, i processi simuleranno l'intenzione di accedere alla risorsa condivisa, seguendo poi il funzionamento dell'algoritmo.

ENTERCS è uno dei quattro tipi di messaggio che si sono resi necessari per modellare il nostro sistema. Gli altri tre tipi sono: **REQUEST** (messaggio di richiesta di entrata in sezione critica da parte di un processo per il coordinatore), **SERVICE** (messaggio "di servizio" che viene inviato tra processi per aggiornare i vector clock) e **TOKEN** (messaggio contenente il token, che può essere spedito dal coordinatore ad un processo o viceversa).

Per dichiarare in SPIN che i messaggi possono essere solo di questa tipologia, si usa la seguente notazione:

```
mtype {REQUEST, SERVICE, TOKEN, ENTERCS};

chan com_c_1 = [4] of {mtype, int, int};  /* da coordinatore a p1 */
chan com_c_2 = [4] of {mtype, int, int}; /* da coordinatore a p2 */
chan com_1_2 = [4] of {mtype, int, int};  /* da p1 a p2 */
chan com_1_c = [4] of {mtype, int, int}; /* da p1 a coordinatore */
chan com_2_c = [4] of {mtype, int, int}; /* da p2 a coordinatore */
chan com_2_1 = [4] of {mtype, int, int}; /*da p2 a p1*/
chan com_g_1 = [0] of {mtype};            /* tra generatore e p1 */
chan com_g_2 = [0] of {mtype};      /* tra generatore e p2 */
```

In sostanza, abbiamo definito otto canali di comunicazione. I primi sei sono tutte le combinazioni possibili tra coordinatore ed i due processi mentre gli ultimi due sono i canali che verranno usati dal generatore di sezione critica per inviare i messaggi di **ENTERCS** ai due processi.

I primi sei canali sono ovviamente quelli più importanti, perché permettono la comunicazione tra i processi che effettivamente siamo intenzionati a studiare. La dicitura:

```
chan com_c_1 = [4] of {mtype, int, int};  /* da coordinatore a p1 */
```

significa che i messaggi che transiteranno su questo canale avranno la seguente struttura: il tipo del messaggio (esclusivamente uno dei quattro tipi elencati nella prima linea di codice) seguito da due interi. I due interi in questione rappresenteranno le due righe del vector clock, che ogni processo deve sempre inviare, sia in caso di **REQUEST** del token sia in caso di **SERVICE**.

Vedremo come queste saranno trattate più in seguito.

A proposito dei canali di comunicazione è importante sottolineare questa scelta di progetto che abbiamo fatto: avremmo potuto utilizzare un solo canale per far comunicare due processi tra loro, permettendo ad entrambi di leggere e scrivere, ovvero creare per la comunicazione tra tutti i processi tre soli canali anziché sei. Questa soluzione presenta però un inconveniente importante: un processo potrebbe scrivere sul canale ed immediatamente leggere lo stesso messaggio da lui inviato. Il problema è aggirabile inserendo un opportuno controllo sul destinatario del messaggio ma questo avrebbe complicato la scrittura del codice.

La soluzione che abbiamo scelto di adottare è invece quella di dedicare, per ogni coppia di processi P1/P2, due diversi canali: uno per la scrittura da parte di P1 e la lettura da parte di P2 e un altro canale per il viceversa.

Si tratta di una soluzione notevolmente più dispendiosa in termini di utilizzo di risorse (e la pagheremo al momento della verifica delle proprietà dell'algoritmo, come vedremo) ma snellisce notevolmente il codice e ci risparmia un gran numero di controlli sul destinatario dei messaggi.

Dopo la definizione dei canali di comunicazione abbiamo definito alcune variabili globali:

```
int id_coord;
int max_iter_p1 = 3;
int max_iter_p2 = 3;
bool p1_waiting=false;
bool p2_waiting=false;
```

`id_coord` servirà per memorizzare l'identificativo del coordinatore, `max_iter_p1` e `max_iter_p2` indicheranno il numero di iterazioni (ovvero il numero di volte in cui verrà inviato il messaggio di **ENTERCS** da parte del generatore) per ciascun processo, ed infine `p1_waiting` e `p2_waiting` indicheranno se il coordinatore ha ricevuto una richiesta "pendente" da parte dei relativi processi.

3.1 Implementazione del coordinatore

Dopo aver definito i mattoni principali con i quali costruiremo il nostro modello, passiamo ora a descrivere l'implementazione effettiva dei vari processi, definiti in Promela come `proctype`.

Il primo – e senza dubbio il più complicato – dei processi implementati è il coordinatore.

```
proctype coordinatore(chan primo_in,secondo_in, primo_out, secondo_out) {

	int req_p1[2];
	int req_p2[2];
	int vectorClock[2];
	vectorClock[0]=0;
	vectorClock[1]=0;
	id_coord = _pid;
	int vc_0, vc_1;
```

Il coordinatore lavora su quattro canali: `primo_in` e `secondo_in` per la ricezione dei messaggi da parte dei processi e `primo_out` e `secondo_out` per l'invio dei messaggi ai due processi.

Al suo interno gestisce numerose variabili locali. Due vettori di interi `req_p1[2]` e `req_p2[2]` per memorizzare le eventuali richieste "pendenti" da parte del primo e del secondo processo, un vettore `vectorClock[2]` che rappresenta a tutti gli effetti il vector clock del coordinatore, due interi `vc_0` e `vc_1` che verranno utilizzati per memorizzare le due righe dei vector clock ricevuti nei messaggi.

Analizziamo adesso stato per stato il funzionamento del codice:

```
	TNR:		/*stato in cui il coordinatore ha il token e non ha altre
			richieste pendenti*/

		do
		:: primo_in?REQUEST(vc_0,vc_1) ->
			if
			:: !(vc_0 > vectorClock[0] && vc_1 > vectorClock[1]) ->
				if
```

```
                    :: vc_0> vectorClock[0] -> vectorClock[0]=vc_0;
                    :: vc_1>vectorClock[1] -> vectorClock[1]=vc_1;
              fi;
              primo_out!TOKEN(vectorClock[0],vectorClock[1]);
              goto NTNR;
        :: else ->
                    req_p1[0]=vc_0; req_p1[1]=vc_1;
                    p1_waiting=true; goto TR;
        fi;

  :: secondo_in?REQUEST(vc_0,vc_1) ->
        if
        [...]

  od;
```

Come già visto nella descrizione del diagramma UML, nello stato **TNR** può solamente avvenire una ricezione di una richiesta da parte di uno dei due processi. All'interno del costrutto "do", il coordinatore si pone in attesa che arrivi su uno dei due canali (`primo_in` e `secondo_in`) un messaggio di tipo ***REQUEST***. Non appena questo avviene si attiva immediatamente un controllo sul vector clock appena ricevuto.

Poiché il nostro sistema è a due client, il controllo da effettuare è solamente che NON entrambe le le righe siano strettamente maggiori del nostro vector clock, perché, come abbiamo visto, questo vorrebbe dire che la richiesta non è elegibile.

Se la condizione è verificata, aggiorniamo le righe del nostro vector clock con il massimo tra il valore precedentemente memorizzato e il valore ricevuto dal processo, inviamo il ***TOKEN*** sul canale relativo (`primo_out` nel caso che la richiesta sia arrivata dal canale primo_in) e andiamo nello stato **NTNR**.

Se invece la richiesta non è elegibile, la memorizziamo all'interno del vettore relativo (`req_p1` nel caso che la richiesta sia arrivata dal canale `primo_in`), settiamo la variabile `p1_waiting` o `p2_waiting` a true per segnalare che c'è una richiesta pendente da parte del processo relativo ed infine andiamo nello stato **TR**.

Analizziamo ora lo stato **NTNR**

```
NTNR:             /*stato in cui il coordinatore non ha il token e non ha
                  altre richieste pendenti*/
      do
      :: primo_in?REQUEST(vc_0,vc_1) ->
                              req_p1[0] = vc_0;
                              req_p1[1]= vc_1;  /* inserisco il processo
                                                tra le richieste pendenti*/
                              p1_waiting=true;
                              goto NTR;         /* e vado nello stato NTR*/

      :: secondo_in?REQUEST(vc_0,vc_1) ->
                              [...]

      :: primo_in?TOKEN(vc_0,vc_1) ->
                              goto TNR;   /*se ricevo il token vado
                                          direttamente nello stato TNR*/
      :: secondo_in?TOKEN(vc_0,vc_1) ->
                              goto TNR;
      od;
```

Come abbiamo visto, in questo stato possono avvenire due sole tipologie di

evento: ricezione di una ***REQUEST*** da parte di un processo oppure ricezione del ***TOKEN***.

Nel primo caso, senza effettuare alcun controllo sull'elegibilità o meno della richiesta, si provvederà a memorizzarla all'interno del vettore corrispondente (`req_p1` in caso la richiesta sia arrivata dal primo client, `req_p2` altrimenti) e successivamente il processo passerà nello stato di **NTR**.

Nel secondo caso, molto più semplicemente, il coordinatore tornerà nello stato di **TNR**, dove si metterà in attesa di nuove ***REQUEST*** da parte dei client.

Passiamo ora a descrivere lo stato **TR**, che è a tutti gli effetti lo stato più importante e complicato di questo processo.

```
TR:         /*stato in cui il coordinatore ha il token e ha altre
            richieste pendenti*/
      do

      :: p1_waiting==true &&
            !(req_p1[0] > vectorClock[0] && req_p1[1] > vectorClock[1])->
                  if
                  :: req_p1[0]>vectorClock[0] ->
                        vectorClock[0]=req_p1[0];
                  :: req_p1[1]>vectorClock[1] ->
                        vectorClock[1]=req_p1[1];
                  fi;
                  primo_out!TOKEN(vectorClock[0],vectorClock[1]);
                  req_p1[0]=0; req_p1[1]=0; p1_waiting=false;
                  if
                  :: p2_waiting -> goto NTR;
                  :: else -> goto NTNR;
                  fi;



      :: p2_waiting==true &&
            !(req_p2[0]>vectorClock[0] && req_p2[1] > vectorClock[1])->

                  [...]


      :: primo_in?REQUEST(vc_0,vc_1) ->
            if
            ::  !(vc_0 > vectorClock[0] && vc_1 > vectorClock[1]) ->
                  vectorClock[0]=vc_0; vectorClock[1]=vc_1;
                  primo_out!TOKEN(vectorClock[0],vectorClock[1]);
                  goto NTR;
            :: else ->
                  req_p1[0]=vc_0; req_p1[1]=vc_1; p1_waiting=true;
                  goto TR;
            fi;

      :: secondo_in?REQUEST(vc_0,vc_1) ->

                  [...]
      od;
```

All'interno del *do* viene effettuato un controllo sulle richieste “pendenti”. In

particolare si verificano le seguenti proprietà: che la variabile `p1_waiting` (ed analogamente `p2_waiting`) sia settata a `true` e che i valori memorizzati all'interno di `req_p1` (ed analogamente `req_p2`) non siano tutti strettamente maggiori dei valori del vector clock (come abbiamo visto, è questa la condizione di elegibilità della richiesta).

Quindi, se c'è una richiesta elegibile in memoria, il coordinatore aggiornerà il proprio vector clock, invierà il ***TOKEN*** sul rispettivo canale in uscita (`primo_out` o `secondo_out`) e cancellerà la richiesta memorizzata (settando a zero i valori del vettore `req_p1` o `req_p2` e ponendo a false la variabile `p1_waiting` o `p2_waiting`).

Infine, verrà effettuato un ultimo controllo per verificare se ci siano altre richieste pendenti o meno in memoria. In caso affermativo, il coordinatore passerà nello stato **NTR**, altrimenti si porrà nello stato di **NTNR**.

Se non ci sono richieste elegibili in memoria il processo si porrà in attesa di una ***REQUEST*** da parte di uno dei due processi client. Al suo arrivo, verificherà se la richiesta arrivata sia elegibile o meno. In caso affermativo, si scatenerà la sequenza di azioni relativa all'invio del ***TOKEN*** al client (ovvero aggiornamento del vector clock, invio del token, passaggio allo stato **NTR**) mentre, in caso negativo, il coordinatore memorizzerà la richiesta ricevuta aggiornando il relativo vettore `req` e settando a `true` la variabile `p1_waiting` (o `p2_waiting`), per poi restare nello stesso stato **TR** (`goto TR`)

L'ultimo stato del coordinatore che resta da analizzare è lo stato di **NTR**.

```
NTR:        /*stato in cui il coordinatore ha il token e non ha altre
            richieste pendenti*/
      do
      :: primo_in?TOKEN(vc_0,vc_1) ->
                              goto TR;
      :: secondo_in?TOKEN(vc_0,vc_1) ->
                              goto TR;
      :: primo_in?REQUEST(vc_0,vc_1) ->
                                    req_p1[0]=vc_0; req_p2[1]=vc_1;
                                    p1_waiting=true; goto NTR;
      :: secondo_in?REQUEST(vc_0,vc_1) ->
                                    req_p2[0]=vc_0; req_p2[1]=vc_1;
                                    p2_waiting=true; goto NTR;
      od;
```

In questo stato possono avvenire due soli eventi: ricezione del ***TOKEN*** o arrivo di una ***REQUEST*** da parte di un client. Nel primo caso, l'azione che conseguirà sarà un semplice cambio di stato da **NTR** a **TR**. Nel secondo caso, come al solito, si memorizzerà la richiesta del client relativo e si resterà nello stato di **NTR** (`goto NTR`)

3.2 Implementazione del client

Dopo aver analizzato l'implementazione del coordinatore passiamo ad analizzare l'implementazione del client (processo). La definizione del processo e le sue variabili locali sono le seguenti:

```
proctype processo(chan coord_in,process_in,generatore_in,coord_out,process_out)
{

      int myPid=_pid-2;
      int vectorClock[2];
      vectorClock[0]=0;
      vectorClock[1]=0;
      int vc_0,vc_1;
```

Come avevamo anticipato, per ogni processo client avremo tre canali di ingresso e due canali in uscita. Nell'ordine: un canale per ricevere messaggi dal coordinatore (***TOKEN***), uno per ricevere messaggi dall'altro processo (***SERVICE***), uno per ricevere messaggi dal generatore (***ENTERCS***), uno per mandare messaggi al coordinatore (***REQUEST*** e ***TOKEN***) e uno per mandare messaggi all'altro processo (***SERVICE***)

Le variabili locali sono simili a quelle del coordinatore e non necessitano di un'ulteriore spiegazione. L'unica differenza è che la variabile myPid non viene settata con l'identificativo del processo ma con il valore: _pid-2. Più avanti vedremo come questo ci servirà per aggiornare correttamente l'array *vectorClock*.

Analizzeremo ora l'implementazione dei vari stati. Abbiamo già visto dalla modellazione in UML che saranno in tutto tre, e che il funzionamento degli eventi che li guideranno sarà più facile rispetto al coordinatore.

Il primo stato è **NCS**, ovvero lo stato in cui il processo non è in sezione critica ed è in attesa di ricevere una richiesta da parte del generatore.

```
NCS:
      do

      :: generatore_in?ENTERCS ->
            vectorClock[myPid] = vectorClock[myPid]+1;
            coord_out!REQUEST(vectorClock[0],vectorClock[1]);
            process_out!SERVICE(vectorClock[0],vectorClock[1]);
            goto WAIT;

      :: process_in?SERVICE(vc_0,vc_1) ->
            if
            :: vc_0 > vectorClock[0] -> vectorClock[0] = vc_0;
            :: vc_1 > vectorClock[1] -> vectorClock[1] = vc_1;
            fi;
      od;
```

In questo stato possono avvenire due eventi: ricezione di un messaggio di tipo ***ENTERCS*** da parte del generatore, oppure ricezione di un messaggio di tipo ***SERVICE***

dall'altro processo.

Nel primo caso si aumenterà di uno la riga del vector clock relativa al processo e successivamente si invierà un messaggio di tipo ***REQUEST*** al coordinatore ed uno di tipo ***SERVICE*** all'altro processo prima di passare allo stato di **WAIT**. Si noti che l'incremento del vector clock funziona correttamente perché il *_pid* si comporta come un contatore e al momento di effettuare l'inizializzazione del sistema ci sono due processi eseguiti prima dei client. (per questo abbiamo settato il valore di *myPid* a *_pid -2*)

Nel secondo caso, invece (ricezione di un messaggio ***SERVICE***) l'unica azione da compiere sarà l'aggiornamento delle righe del vector clock quando il valore ricevuto dal processo sia maggiore di quello memorizzato nella variabile locale.

Lo stato di **WAIT** presenta la seguente struttura:

```
WAIT:
      do
      :: coord_in?TOKEN(vc_0,vc_1) ->
            printf("SONO IL PROCESSO %d, E STO PER ENTRARE IN CS \n",
            myPid);
            goto CS;

      :: process_in?SERVICE (vc_0,vc_1) ->
            if
            :: vc_0>vectorClock[0] -> vectorClock[0] = vc_0;
            :: vc_1>vectorClock[1] -> vectorClock[1] = vc_1;
            fi;
      od;
```

Come sappiamo, in questo stato il processo client si pone in attesa del ***TOKEN*** da parte del coordinatore. Una volta ricevuto, verrà stampato a video un messaggio di conferma della ricezione del token e il processo passerà nello stato di **CS**.

Anche nella fase di **WAIT** il processo potrebbe ricevere un messaggio di tipo ***SERVICE***. In questo caso, come già visto in precedenza, l'unica azione che conseguirebbe sarebbe l'aggiornamento del vector clock.

L'ultimo stato che dobbiamo analizzare è quello relativo alla sezione critica.

```
CS:
      i=0;
      printf("SONO IL PROCESSO %d E SONO IN CS \n", myPid);
      do
            :: i < N -> i++;
            :: i==N -> coord_out!TOKEN(vectorClock[0],vectorClock[1]);
                        goto NCS;
      od;
```

In questo caso ci è sembrato logico simulare l'accesso alla sezione critica con una semplice stampa a video del messaggio "*Sono il processo X e sono in CS*". Dopo questa operazione il client resterà in **CS** per un numero N di step definito dall'utente prima di restituire il ***TOKEN*** al coordinatore e portarsi nello stato iniziale di **NCS**, in attesa di nuovi ***ENTERCS*** da parte del generatore di sezione critica.

3.3 Implementazione del generatore ed inizializzazione del sistema

Come abbiamo già detto, Spin non offre la possibilità di generare implicitamente eventi casuali. Per questo motivo si è resa necessaria l'implementazione di un generatore di sezione critica, ovvero un processo che ciclicamente inviasse messaggi di tipo ***ENTERCS*** ai due processi.

Il generatore è stato implementato in questo modo:

```
proctype generator(chan p1, p2){
      {
      do

            :: !p1_waiting -> p1!ENTERCS; max_iter_p1=max_iter_p1-1;
            :: !p2_waiting -> p2!ENTERCS; max_iter_p2=max_iter_p2-1;

      od;
      }unless{max_iter_p1==0 || max_iter_p2==0}
}
```

Ovvero, quando uno dei due processi non è in stato di "waiting" viene inviato un messaggio di ***ENTERCS*** e viene scalato di uno il valore di `max_iter_p1` (o `max_iter_p2`), inizialmente settato a 3. Quando una delle due variabili raggiunge il valore di 0 il generatore si disattiva, dando così fine all'intero processo.

L'inizializzazione del sistema in spin avviene attraverso la definizione di un particolare processo: il processo `init`. Nel nostro caso, è stato definito come segue:

```
init {
      /* processo coordinatore*/
      run coordinatore(com_1_c, com_2_c, com_c_1, com_c_2);

      /* processo 1*/
      run processo(com_c_1, com_2_1, com_g_1, com_1_c, com_1_2);

      /* processo 2*/
      run processo(com_c_2, com_1_2, com_g_2, com_2_c, com_2_1);

      /* generatore di sezioni critiche */
      run generator(com_g_1, com_g_2);
}
```

Come si vede, vengono attivati nell'ordine il coordinatore, i due processi ed il generatore, utilizzando come parametri i canali di comunicazione che avevamo definito in precedenza.

4. Utilizzo di Xspin per la simulazione e la verifica di alcune proprietà

Per utilizzare in maniera più pratica Spin ed avere un riscontro diretto sulla simulazione del sistema che abbiamo implementato, abbiamo deciso di utilizzare l'interfaccia grafica Xspin. Attraverso Xspin è possibile, tra le altre cose, simulare il comportamento dei vari processi, verificare la presenza di deadlock o la validità di formule espresse in LTL.

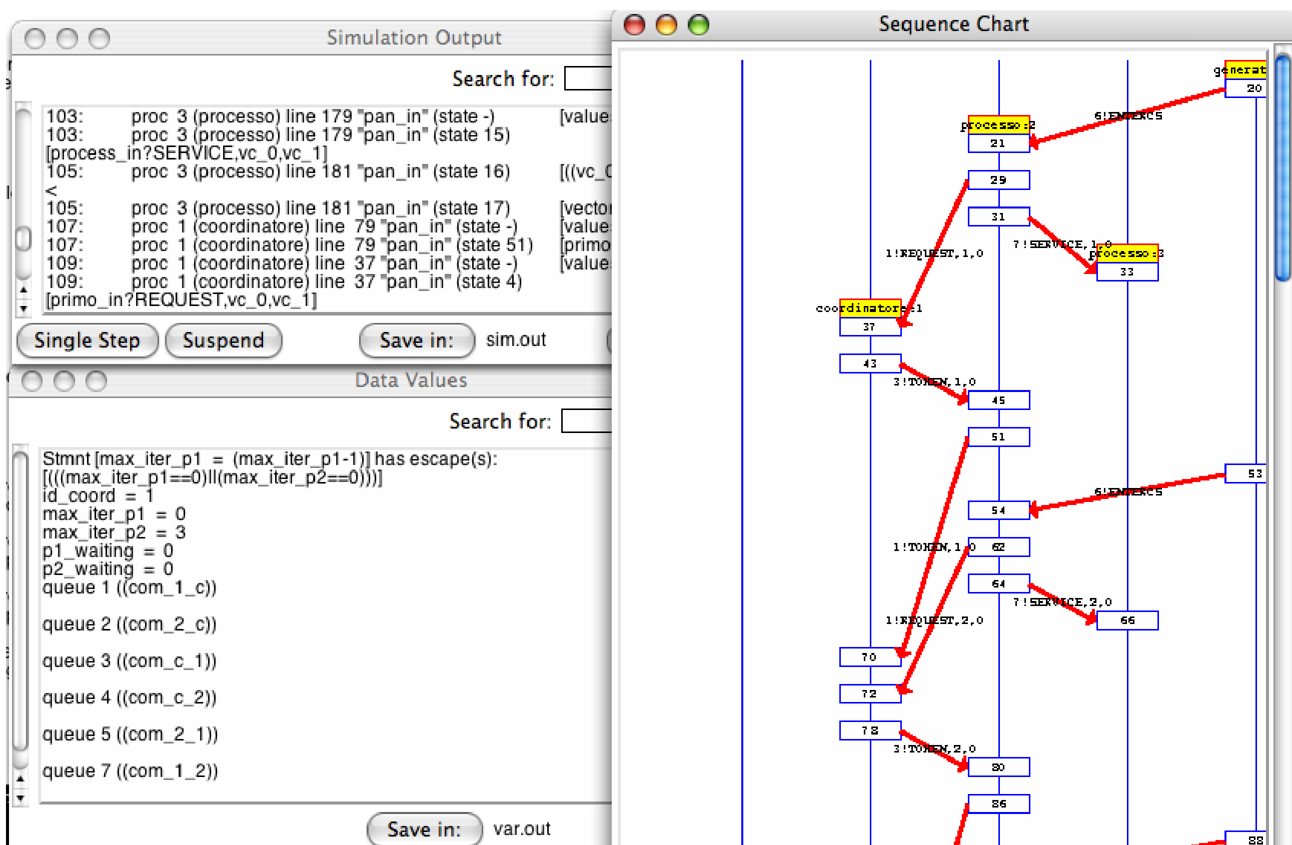
La prima operazione da fare è verificare la correttezza sintattica del nostro modello.
Cliccando su *Run Syntax Check*, Xspin genera ed esegue il comando

```
spin -a -v pan_in
```

e stampa a video il risultato dell'operazione: `no syntax errors`.

Il secondo passo da effettuare è simulare il funzionamento del nostro sistema. Cliccando su "*Set simulation parameters*" viene aperta una schermata con tutte le opzioni di simulazione disponibili, riguardanti il numero e il tipo di finestre che si desidera visualizzare. Lasciando i parametri impostati di default e cliccando su *Start* vengono aperte tre finestre: **Simulation output**, per visualizzare passo per passo il comportamento dell'algoritmo ad ogni istante temporale, **Data Values**, per visualizzare i valori assunti dalle variabili globali nel corso della simulazione, ed infine la finestra di **Sequence Chart**, nella quale verrà rappresentato in un diagramma la comunicazione tra i vari processi.

Cliccando su *Run* la simulazione ha inizio e le finestre vengono man mano popolate.



Dopo 152 passi la simulazione termina per timeout perché, per semplice comodità, non abbiamo rappresentato degli stati finali all'interno del nostro sistema, che quindi va in "stallo" quando il generatore smette di inviare messaggi di tipo ***ENTERCS***.

Ispezionando la **Sequence Chart** possiamo vedere come il comportamento del sistema era quello atteso. Spin ha provveduto a simulare i ritardi nella comunicazione tra i vari processi ma l'algoritmo implementato si è dimostrato robusto ed ha fronteggiato correttamente ogni tipo di ritardo.

Come ci aspettavamo, al termine della simulazione il processo coordinatore è bloccato nello stato di **TNR**, ovvero lo stato in cui possiede il token e non ci sono richieste pendenti, mentre i due processi sono nello stato di **NCS**.

Questo vuol dire che non è stato rilevato alcun **deadlock** dal sistema finché il generatore di sezione critica era attivo, mentre ancora non possiamo dire nulla sulla **Starvation** e sulla **Safety** del sistema.

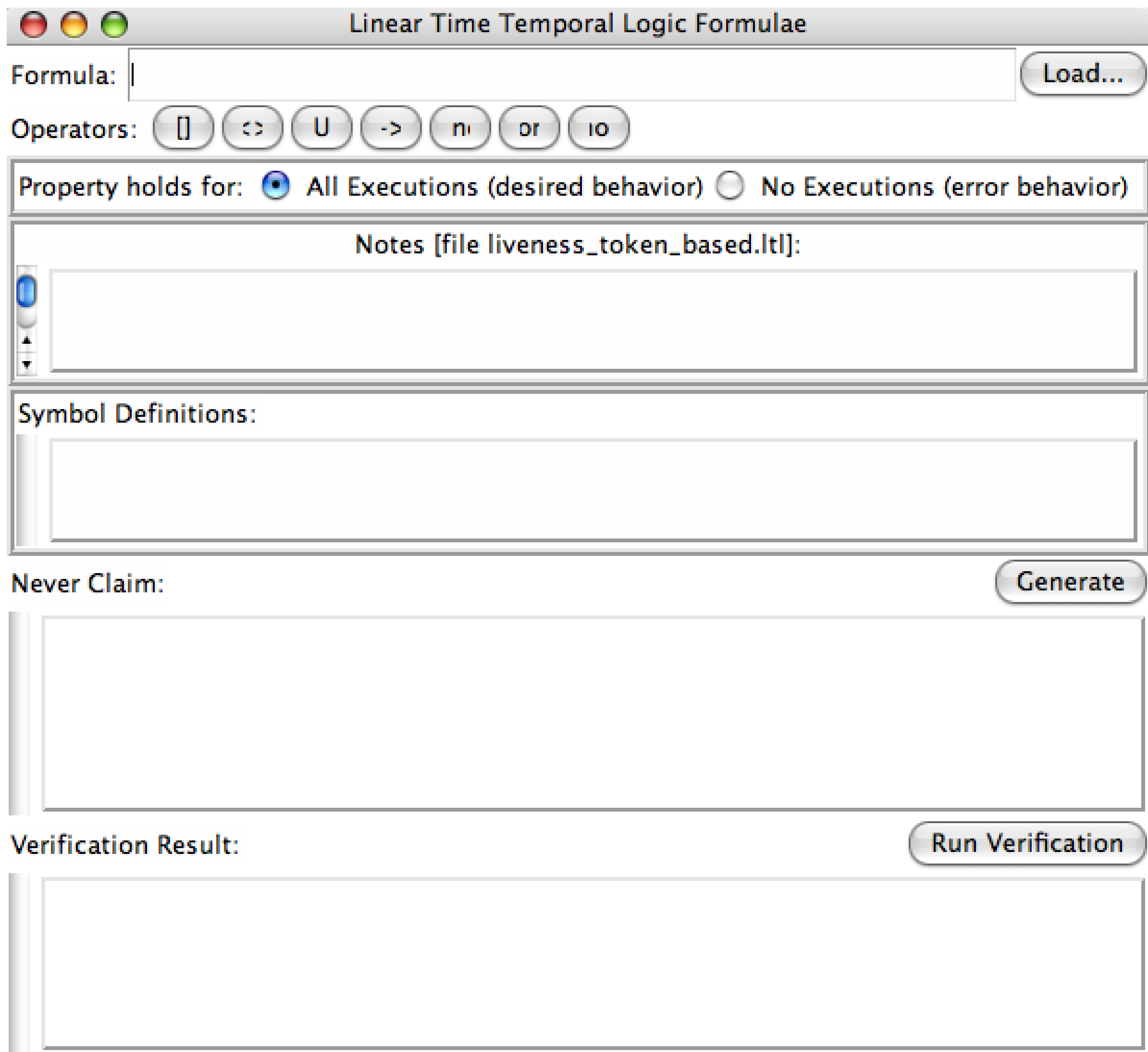
Per poter verificare proprietà più complesse come queste, Spin mette a disposizione un costrutto ad hoc chiamato "*never claim*". Un *never claim* è semplicemente una sequenza di espressioni che rappresentano un comportamento che non desideriamo che avvenga.

E' possibile quindi utilizzare questa funzione per dimostrare la validità di alcune proprietà, semplicemente dichiarando come "*never claim*" il loro contrario e verificandone la validità.

Inoltre, per maggiore comodità, Spin fornisce un traduttore automatico da formula LTL a *never claim*, sebbene non tutti gli operatori temporali siano supportati.

Per sfruttare tutte queste caratteristiche di Spin con l'interfaccia grafica Xspin è sufficiente cliccare su *LTL Property management*. Da qui una finestra ci permetterà di scrivere una formula LTL (o di caricare una formula precedentemente salvata) e di tradurla in *never claim* con un semplice click.

Prima di fare questo però bisognerà associare le variabili utilizzate nella formula LTL agli stati del nostro sistema. Potremo fare questo attraverso un elenco di "*#define*" che scriveremo all'interno del campo "*Symbol definitions*".



Una volta generato il *never claim* relativo alla formula LTL introdotta potremo verificarlo cliccando su "*run verification*". Se la proprietà è verificata, comparirà un messaggio di conferma sulla validità della formula LTL inserita, altrimenti verrà generato un controesempio che sarà possibile simulare attraverso la finestra di Sequence Chart di cui abbiamo già parlato.

Dopo aver introdotto gli strumenti messi a disposizione da Spin per la verifica di formule LTL, passiamo ad analizzare le proprietà del nostro sistema che vogliamo dimostrare.

4.1 Verifica della proprietà di Safety

La prima di queste proprietà è la Safety, ovvero la Mutua Esclusione. Dal momento che i processi che devono accedere alla sezione critica sono solamente due, possiamo formulare la proprietà nel seguente modo:

[] (in_CS_1 -> ! in_CS_2)

L'operatore [] rappresenta l'operatore LTL *Globally*, quindi desideriamo che sia sempre verificata la seguente proprietà: se il processo p1 è in Critical Section (**CS**), non lo è il processo p2 e viceversa.

A questo punto dobbiamo definire i simboli "in_CS_1" ed "in_CS_2". Lo facciamo nell'apposito riquadro:

#define in_CS_1 processo[2]@CS
#define in_CS_2 processo[3]@CS

Ovvero le due variabili rappresentano i processi con identificativo 2 e 3 (l'identificativo 1 è lasciato al coordinatore) nello stato **CS**.

Cliccando sul pulsante "generate", Xspin esegue il comando:

```
spin -f "!( []  (in_CS_1 -> ! in_CS_2) )"
```

con il quale viene generato il never claim corrispondente alla formula LTL inserita (negata):

```
/*
 * Formula As Typed: []  (in_CS_1 -> ! in_CS_2)
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]  (in_CS_1 -> ! in_CS_2))
 * (formalizing violations of the original)
 */

never {    /* !([]  (in_CS_1 -> ! in_CS_2)) */
      T0_init:
      if
      :: ((in_CS_1) && (in_CS_2)) -> goto accept_all
      :: (1) -> goto T0_init
      fi;
      accept_all:
      skip
}
```

Una volto generato il *never claim* possiamo verificarlo cliccando sul pulsante "*Run verification*". Lasciando le opzioni di verifica di default viene eseguita una verifica di tipo esaustivo.

Per completare la verifica, però, non sono sufficienti i 128 Mbytes impostati di default e dobbiamo quindi settare la memoria al massimo disponibile sul nostro computer (1024 Mbytes). Xspin genera quindi i seguenti comandi:

```
spin -a -X -N pan.ltl  pan_in

gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=1024 -DXUSAFE -DNOFAIR  pan.c
time ./pan -v -X -m10000 -w19  -a -c1
```

con i quali effettua la verifica formale del never claim.

La verifica viene completata in 11.2 secondi, rilasciando il seguente report:

```
(Spin Version 5.1.4 -- 27 January 2008)
      + Partial Order Reduction

Full statespace search for:
      never claim             +
      assertion violations    + (if within scope of claim)
      acceptance   cycles     + (fairness disabled)
      invalid end states      - (disabled by never claim)


State-vector 456 byte, depth reached 236, errors: 0
   708598 states, stored
   755432 states, matched
  1464030 transitions (= stored+matched)
        0 atomic steps
hash conflicts:    345130 (resolved)


Stats on memory usage (in Megabytes):
  318.964   equivalent memory usage for states (stored*(State-vector +
overhead))
  306.696   actual memory usage for states (compression: 96.15%)
            state-vector as stored = 438 byte + 16 byte overhead
    2.000   memory used for hash table (-w19)
    0.343   memory used for DFS stack (-m10000)
  308.203   total actual memory usage
unreached in proctype coordinatore
      line 144, "pan.___", state 126, "req_p1[0] = vc_0"
      line 144, "pan.___", state 128, "p1_waiting = 1"
      line 148, "pan.___", state 131, "req_p2[0] = vc_0"
      line 148, "pan.___", state 133, "p2_waiting = 1"
      line 151, "pan.___", state 138, "-end-"
      (5 of 138 states)
unreached in proctype processo
      line 173, "pan.___", state 9, "printf('errore con il PID\n')"
      line 205, "pan.___", state 41, "-end-"
      (2 of 41 states)
unreached in proctype generator
      (0 of 13 states)
unreached in proctype :init:
      (0 of 5 states)
unreached in proctype :never:
      line 268, "pan.___", state 8, "-end-"
      (1 of 8 states)


pan: elapsed time 11.2 seconds
```

```
pan: rate  63551.39 states/second
pan: avg transition delay 7.616e-06 usec
11.real        10.10 user          0.88 sys
```

Il risultato della verifica è che la formula espressa precedentemente in LTL è valida, ovvero non è stato possibile trovare un controesempio nel quale i due processi fossero contemporaneamente nello stato di CS.

Linear Time Temporal Logic Formulae

Formula: [] (in_CS_1 -> ! in_CS_2) Load...

Operators: [] <> U -> and or not

Property holds for: All Executions (desired behavior) No Executions (error behavior)

Notes [file safety_token_based.ltl]:

```
Use Load to open a file or a template.
```

Symbol Definitions:

```
#define in_CS_1 processo[2]@CS
#define in_CS_2 processo[3]@CS
```

Never Claim: Generate

```
        /*
         * Formula As Typed: []  (in_CS_1 -> ! in_CS_2)
         * The Never Claim Below Corresponds
         * To The Negated Formula !([]  (in_CS_1 -> ! in_CS_2))
         * (formalizing violations of the original)
         */

never {    /* !([]  (in_CS_1 -> ! in_CS_2)) */
```

Verification Result: valid Run Verification

```
warning: use of a rendezvous stmnts in the escape
        of an unless clause, if present, could make p.o. reduction
        invalid (use -DNOREDUCE to avoid this)
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 0: Claim reached state 5 (line 263)
```

Help Clear Close Save As...

Nel prossimo paragrafo verificheremo la proprietà di **Liveness**.

4.2 Verifica della proprietà di Liveness

Abbiamo già verificato l'assenza di ***deadlock*** all'interno del sistema, che sarebbero stati rilevati automaticamente dalla prima simulazione con Spin. Siamo quindi interessati a verificare l'assenza di ***starvation***, ovvero: ogni volta che un processo chiede l'accesso alla sezione critica, viene prima o poi servito.

Possiamo esprimere questa proprietà in LTL nel seguente modo:

[] ((in_WAIT_1 -> <> in_CS_1) &&(in_WAIT_2 -> <> in_CS_2))

L'operatore <> corrisponde all'operatore *Future* in LTL. Stiamo quindi dicendo che, se un processo si trova nello stato di **WAIT**, prima o poi entrerà nello stato di **CS**.

Come abbiamo già fatto per la verifica della **Safety**, definiamo le variabili nel seguente modo:

#define in_CS_1 processo[2]@CS
#define in_CS_2 processo[3]@CS
#define in_WAIT_1 processo[2]@WAIT
#define in_WAIT_2 processo[3]@WAIT

Ancora una volta, generiamo il never claim corrispondente alla formula negata:

```
/*
 * Formula As Typed: []  ((in_WAIT_1 -> <>  in_CS_1) &&(in_WAIT_2 -> <>
                             in_CS_2))
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]  ((in_WAIT_1 -> <>  in_CS_1) &&(in_WAIT_2
                                   -> <>  in_CS_2)))
 * (formalizing violations of the original)
 */

never {    /* !([]  ((in_WAIT_1 -> <>  in_CS_1) &&(in_WAIT_2 -> <>
            in_CS_2))) */
T0_init:
      if
      :: (! ((in_CS_2)) && (in_WAIT_2)) -> goto accept_S5
      :: (! ((in_CS_1)) && (in_WAIT_1)) -> goto accept_S10
      :: (1) -> goto T0_init
      fi;
accept_S5:
      if
      :: (! ((in_CS_2))) -> goto accept_S5
      fi;
accept_S10:
      if
      :: (! ((in_CS_1))) -> goto accept_S10
      fi;
}
```

Chiedendo al sistema di verificare questa proprietà, Spin impiega 51.8 secondi per definirla valida e produrre il seguente report:

```
(Spin Version 5.1.4 -- 27 January 2008)
      + Partial Order Reduction

Full statespace search for:
      never claim             +
      assertion violations    + (if within scope of claim)
      acceptance   cycles     + (fairness disabled)
      invalid end states      - (disabled by never claim)

State-vector 456 byte, depth reached 236, errors: 0
  1424428 states, stored (2.14026e+06 visited)
  3848168 states, matched
  5988426 transitions (= visited+matched)
        0 atomic steps
hash conflicts:   3129535 (resolved)

Stats on memory usage (in Megabytes):
  641.184   equivalent memory usage for states (stored*(State-vector +
overhead))
  617.788   actual memory usage for states (compression: 96.35%)
            state-vector as stored = 439 byte + 16 byte overhead
    8.000   memory used for hash table (-w21)
    0.343   memory used for DFS stack (-m10000)
    1.628   memory lost to fragmentation
  624.504   total actual memory usage
unreached in proctype coordinatore
      line 144, "pan.___", state 126, "req_p1[0] = vc_0"
      line 144, "pan.___", state 128, "p1_waiting = 1"
      line 148, "pan.___", state 131, "req_p2[0] = vc_0"
      line 148, "pan.___", state 133, "p2_waiting = 1"
      line 151, "pan.___", state 138, "-end-"
      (5 of 138 states)
unreached in proctype processo
      line 173, "pan.___", state 9, "printf('errore con il PID\n')"
      line 205, "pan.___", state 41, "-end-"
      (2 of 41 states)
unreached in proctype generator
      (0 of 13 states)
unreached in proctype :init:
      (0 of 5 states)
unreached in proctype :never:
      line 277, "pan.___", state 17, "-end-"
      (1 of 17 states)


pan: elapsed time 51.8 seconds
pan: rate 41285.841 states/second
pan: avg transition delay 8.6567e-06 usec
52.real        43.03 user         2.33 sys
```

Linear Time Temporal Logic Formulae

Formula: [] ((in_WAIT_1 -> <> in_CS_1) &&(in_WAIT_2 -> <> in_CS_2))| Load...

Operators: [] <> U -> and or not

Property holds for: All Executions (desired behavior) No Executions (error behavior)

Notes [file liveness_token_based.ltl]:

```
Use Load to open a file or a template.
```

Symbol Definitions:

```
#define in_CS_1 processo[2]@CS
#define in_CS_2 processo[3]@CS
#define in_WAIT_1 processo[2]@WAIT
#define in_WAIT_2 processo[3]@WAIT
```

Never Claim: Generate

```
        /*
         * Formula As Typed: []  ((in_WAIT_1 -> <>  in_CS_1) &&(in_WAIT_2 -> <>
in_CS_2))
         * The Never Claim Below Corresponds
         * To The Negated Formula !([]  ((in_WAIT_1 -> <>  in_CS_1) &&(in_WAIT_2
-> <>  in_CS_2)))
         * (formalizing violations of the original)
         */
```

Verification Result: valid Run Verification

```
warning: use of a rendezvous stmnts in the escape
        of an unless clause, if present, could make p.o. reduction
        invalid (use -DNOREDUCE to avoid this)
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 0: Claim reached state 7 (line 265)
depth 0: Claim reached state 7 (line 266)
```

Help Clear Close Save As..

L'algoritmo proposto ed implementato soddisfa quindi anche la proprietà di **Liveness**.

4.3 Identificazione di un controesempio per una proprietà non soddisfatta

Per completezza, mostriamo adesso un caso in cui Spin dimostra la non validità di una formula, fornendo un controesempio.

Vogliamo ad esempio verificare se è vero che entrambi i processi entreranno almeno una volta in sezione critica. La formula LTL che sintetizza questa proprietà è la seguente:

<> in_CS_1 && <> in_CS_2

Ovvero: c'è un istante futuro in cui il processo 1 entra in sezione critica ed uno in cui il processo 2 entra in sezione critica.

Si noti che, per come abbiamo progettato l'algoritmo, ed in particolare il generatore di sezione critica, un processo potrebbe decidere di non andare mai in sezione critica.

Il never claim prodotto da Spin è il seguente:

```
/*
 * Formula As Typed: <>  in_CS_1 && <>  in_CS_2
 * The Never Claim Below Corresponds
 * To The Negated Formula !(<>  in_CS_1 && <>  in_CS_2)
 * (formalizing violations of the original)
 */

never {    /* !(<>  in_CS_1 && <>  in_CS_2) */
accept_init:
T0_init:
      if
      :: (! ((in_CS_2))) -> goto accept_S2
      :: (! ((in_CS_1))) -> goto accept_S5
      fi;
accept_S2:
T0_S2:
      if
      :: (! ((in_CS_2))) -> goto accept_S2
      fi;
accept_S5:
T0_S5:
      if
      :: (! ((in_CS_1))) -> goto accept_S5
      fi;
}
```

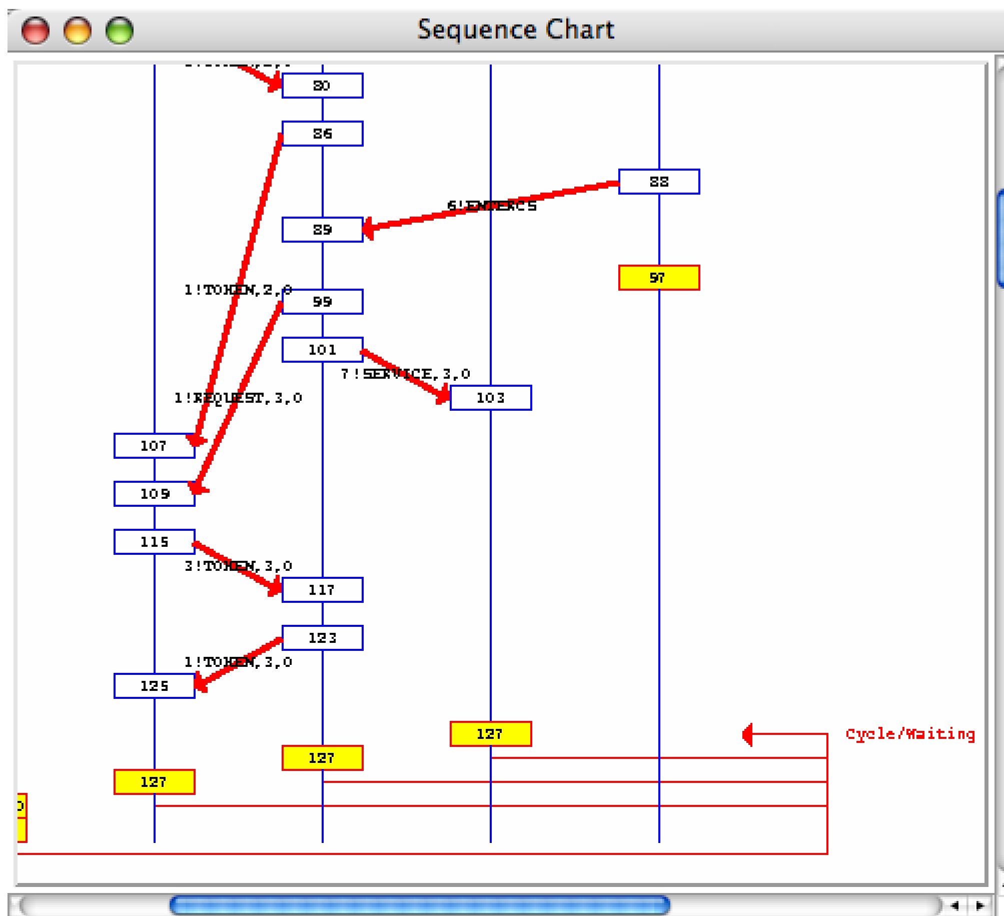
Chiedendo di verificare questa proprietà, il sistema la riconosce come non valida e permette di simulare attraverso il **Sequence Chart** ed il **Simulation Output** un possibile controesempio.

Verification Result: not valid

Run Verification

```
warning: use of a rendezvous stmnts in the escape
         of an unless clause, if present, could make p.o. reduction
         invalid (use -DNOREDUCE to avoid this)
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 0: Claim reached state 5 (line 266)
depth 2: Claim reached state 9 (line 272)
```

Help Clear Close Save As..



Come ci aspettavamo, Spin propone un esempio nel quale il generatore invia per tre volte il messaggio di ***ENTERCS*** allo stesso processo per poi disattivarsi.

Notiamo che potremmo modificare facilmente il generatore per fare in modo che soddisfi anche questa proprietà (ad esempio dichiarando esplicitamente di inviare il messaggio ***ENTERCS*** ad entrambi i processi) ma crediamo che sia più corretta l'implementazione già proposta, perché anche nei sistemi reali possono esistere processi che non necessitano mai di accedere alla risorsa condivisa.

5. Supporto per N client

Come abbiamo anticipato, abbiamo modellato il sistema con due soli client. E' possibile estendere il modello in modo da supportare un numero arbitrario di processi client? Nel modello proposto, per ogni coppia di processi è stato inizializzato un canale di comunicazione. Questo tipo di procedimento – dispendioso in termini di utilizzo di risorse, ma molto efficace e realistico – rende impossibile l'estensione del modello ad un numero arbitrario di processi, perché, per ogni processo aggiuntivo, l'utente dovrebbe farsi carico di aggiungere i canali di comunicazione e modificare il codice dei processi, per modellare il comportamento del sistema nello scambio di messaggi con ogni processo aggiunto.

Il modello proposto quindi è evidentemente inadatto a questo tipo di lavoro. Da qui consegue ovviamente una nuova domanda: è necessario questo gran numero di canali di comunicazione? Non si potrebbe utilizzare un unico canale per far comunicare tutti gli N processi?

Per rispondere a questa domanda abbiamo in primo luogo studiato la tesina di Cusmai, Monteleone e Orsi relativa proprio alla comunicazione in Spin tra N processi attraverso un unico canale condiviso. Il problema affrontato da questi tre colleghi, però, è radicalmente diverso dal nostro: essi si propongono di realizzare degli algoritmi che permettano lo scambio di messaggi tra N processi, *mantenendo invariato l'ordine di ricezione dei* messaggi (ovvero: se p1 manda un messaggio prima di p2, il messaggio di p1 verrà letto prima di p2). E' evidente che questo è un grosso limite e non risponde alle nostre esigenze: se il nostro sistema non presentasse dei ritardi nella comunicazione sarebbe addirittura inutile un algoritmo come quello che abbiamo presentato.

D'altro canto, Spin offre la possibilità di accedere ad una coda in maniera casuale, ovvero di prelevare dei messaggi che non affiorino dalla coda ma siano in mezzo agli altri (attraverso la funzionalità: *nomecanale??MSG*). Questa funzionalità è molto utile per realizzare quello che ci siamo prefissati, perché renderebbe l'ordine di ricezione dei messaggi diverso dall'ordine di invio. Anche nel realizzare il sistema con un accesso ad un punto casuale della coda, abbiamo però incontrato un problema purtroppo insormontabile. Creando un solo canale per la comunicazione tra processi e coordinatore ed un altro per la comunicazione tra tutti i processi si sono verificati costantemente dei problemi di deadlock/starvation del seguente tipo: il coordinatore invia il token per il processo p1 sul canale condiviso, il processo p2 estrae il messaggio contenente il token, verifica di non essere lui il destinatario e lo invia nuovamente nella coda, ancora una volta anticipa p1 estraendo il messaggio dalla coda eccetera... In sostanza: il processo p2, avendo accesso allo stesso canale di p1, potrebbe ostacolarne la ricezione del messaggio.

Un ulteriore limite evidente per estendere il modello ad N client riguarda il tipo di canale. Come abbiamo visto, il vector clock viene inviato tra i processi "riga per riga", perché i canali di comunicazione possono essere solo di "type" e non di array. Poiché però il numero di righe del vector clock aumenta con l'aumentare dei processi, è evidente che i canali di comunicazione (e di conseguenza tutto il codice) dovranno essere modificati manualmente con l'aggiunta di ogni nuovo processo.

Realizzare quindi un'estensione del modello da 2 ad N processi a scelta dell'utente è possibile solo attraverso la realizzazione di un generatore di codice Spin, che per ovvi motivi di tempo esula dagli obiettivi di questa tesina. Ad ogni modo, nel paragrafo successivo mostreremo l'estensione del modello da 2 a 3 processi, evidenziando le modifiche al codice che devono essere effettuate con l'aggiunta di ogni nuovo processo.

5.1 Estensione ad un sistema a 3 client

Estendere il modello proposto da 2 a 3 client richiede un buon numero di modifiche al codice. La prima di queste riguarda la dimensione dei canali di comunicazione: se prima erano di tipo

```
{mtype, int, int}
```

ora saranno di tipo

```
{mtype, int, int, int}
```

perché, come anticipato nel paragrafo precedente, il numero di righe del vector clock è aumentato. Inoltre dobbiamo inserire i nuovi canali di comunicazione per il terzo processo:

```
chan com_c_3 = [4] of {mtype, int, int, int};        /*da coordinatore a 3*/
chan com_1_3 = [4] of {mtype, int, int, int};        /* da proc 1 a proc 3 */
chan com_2_3 = [4] of {mtype, int, int, int};        /* da proc 2 a proc 3 */
chan com_3_1 = [4] of {mtype, int, int, int};        /* da proc 3 a proc 1 */
chan com_3_2 = [4] of {mtype, int, int, int};        /* da proc 3 a proc 2  */
chan com_3_c = [4] of {mtype, int, int, int};  /* da proc 3 a coordinatore  */
chan com_g_3 = [0] of {mtype};                 /* da generatore a proc 3 */
```

Anche nelle variabili globali dovremo inserire quelle analoghe agli altri client (`p3_waiting` e `max_iter_p3`)

Nella definizione del coordinatore dobbiamo inserire i due nuovi canali: `terzo_in` e `terzo_out`:

```
proctype coordinatore(chan primo_in,secondo_in,terzo_in,primo_out,secondo_out,
terzo_out)
```

Anche nelle variabili locali dovremo modificare la dimensione dei vector clock ed inserire il vettore d'appoggio `req_p3` (dal funzionamento analogo agli altri vettori `req`)

Una modifica importante da fare riguarda il controllo di elegibilità: se con due processi era sufficiente effettuare il controllo: `!(vc_0>vectorClock[0] && vc_1>vectorClock[1])`

con l'introduzione di un terzo processo dobbiamo effettuare un controllo più approfondito: dobbiamo verificare che solo la riga relativa al processo richiedente sia superiore al valore del vector clock memorizzato e che le altre due righe siano minori o uguali. Quindi, nel caso in cui il messaggio sia ricevuto dal primo client, dovremo effettuare questo controllo: `(vc_1 <= vectorClock[1]) && (vc_2<=vectorClock[2])`

e analogamente per tutti gli altri casi (notare che non serve controllare il valore di vc_0 perché sarà sicuramente maggiore del valore memorizzato all'interno dell'array)

Inoltre dovremo gestire il comportamento in caso di ricezione di una ***REQUEST*** da parte del terzo processo. In maniera analoga agli altri due processi, modelliamo anche questo caso con il controllo:

```
:: terzo_in?REQUEST(vc_0,vc_1,vc_2) -> [...]
```

Il discorso è analogo per tutti gli altri stati: dovremo sempre prevedere i casi in cui arrivino ***REQUEST*** da parte del terzo processo oppure il terzo processo esca dalla sezione critica e restituisca il ***TOKEN***.

Quando il coordinatore si trova nello stato di **TR** dovrà tenere conto della presenza del terzo processo nel controllare che le richieste memorizzate siano elegibili, effettuando lo stesso controllo che abbiamo riportato sopra:

```
p1_waiting==true&&req_p1[1]<=vectorClock[1]&&req_p1[2]<=vectorClock[2]-> [...]
```

Anche per quanto riguarda il processo client bisogna ovviamente effettuare qualche modifica. Nella definizione del processo dovremo tener conto del fatto che i client sono adesso tre e non più due, quindi ciascun processo dovrà interagire con tre canali di comunicazione in uscita (coordinatore, processo 1, processo 2) e quattro in entrata:

```
proctype processo(chan coord_in,process1_in,process2_in,generatore_in,coord_out,
process1_out,process2_out)
```

Nel corpo del processo dovremo chiaramente definire il comportamento per la comunicazione con il nuovo processo, ma la struttura è analoga alla precedente, sia per l'invio che per la ricezione dei messaggi ***SERVICE***.

Le ultime modifiche riguardano il generatore di sezione critica (modifiche banali) ed il processo *init*. In quest'ultimo dovremo creare i processi passando i corretti canali di comunicazione che abbiamo definito in precedenza:

```
/* processo coordinatore*/
run coordinatore(com_1_c,com_2_c,com_3_c,com_c_1,com_c_2,com_c_3);

/* processo 1*/
run processo(com_c_1,com_2_1,com_3_1,com_g_1,com_1_c,com_1_2,com_1_3);

/* processo 2*/
run processo(com_c_2,com_1_2,com_3_2,com_g_2,com_2_c,com_2_1,com_2_3);

/* processo 3*/
run processo(com_c_3,com_1_3,com_2_3,com_g_3,com_3_c,com_3_1,com_3_2);

/* generatore di sezioni critiche */
run generator(com_g_1,com_g_2,com_g_3);
```

Eseguendo il programma si può verificare che la simulazione viene effettuata correttamente, dimostrando l'assenza di deadlock.

6. Conclusioni

Abbiamo modellato e dimostrato attraverso l'uso dei metodi formali le proprietà di **Safety** e **Liveness** di un algoritmo per la mutua esclusione in ambiente distribuito basato su token.

L'utilizzo di un model checker come SPIN ci ha permesso di simularne facilmente il funzionamento, grazie a diverse caratteristiche (sintassi C-like, possibilità di gestire array, opzioni per il rilevamento dei deadlock...) che ne fanno il tool di riferimento per la verifica formale degli algoritmi di comunicazione tra processi.

E' stato inoltre un ottimo esercizio approfondire in maniera pratica gli aspetti teorici studiati durante il corso di Metodi formali per l'ingegneria del software.

Bibliografia

- *Promela and SPIN man pages* (http://www.spinroot.com)
- *Ruys "**SPIN Beginners' Tutorial**"*
- *Dwyer, Hatcliff "**Introduction to SPIN**"*
- *Nifosi "**SPIN**"*
- *Delle Fave, Gheri "**Modellazione e verifica formale di algoritmi per la mutua esclusione in ambiente distribuito**"*
- *Cusmai, Monteleone, Orsi "**Spin: comunicazione tra N processi attraverso un unico canale condiviso**"*
- *Spoletini "**The model Checker SPIN**"*