SPIN: COMUNICAZIONE TRA N PROCESSI ATTRAVERSO UN UNICO CANALE CONDIVISO

Tesina del corso
di
Metodi Formali nell'Ingegneria del Software
AA 2006-2007

Prof. Toni Mancini

Cusmai Francesco
Monteleone Marta
Orsi Silvia

Indice

Abstract 2
1. Introduzione: la gestione della coda in spin 3
2. Scopo del progetto 4
3. Diagrammi Uml 6
4. Definizione degli algoritmi 7
4.1. Algoritmo 1 7
4.2. Algoritmo 2 8
4.3. Algoritmo 3 8
5. Efficienza degli algoritmi in termini di tempo di esecuzione 9
6. Verifica di alcune proprietà ltl del sistema 12
7. Confronto fra le soluzioni proposte : vantaggi,svantaggi ed efficienza. 14
8. Possibili sviluppi futuri 16
9. Software utilizzato 17
10. Appendici 18
10.1. Appendice A : Particolari istruzioni promela usate 18
10.2. Appendice B : Tempi di esecuzione 22
10.3. Appendice C : Dettagli algoritmo1 24
10.3.1. Dal Diagramma a stati all'implementazione 24
10.3.2. Definizione del Processo 26
11. Riferimenti 27

Abstract

Il progetto ha avuto lo scopo di realizzare un sistema che, permettesse, a più processi promela, di comunicare fra loro, mediante un unico canale, garantendo che non si verificassero situazioni di stallo o starvation e che, l'ordine dei messaggi nella coda non fosse stravolto.
In particolare sono state realizzate tre possibili alternative testate su un piccolo sistema asincrono modellato con un diagramma uml.

1. Introduzione: la gestione della coda in spin

Spin gestisce i messaggi scambiati fra i vari processi promela attraverso un canale di comunicazione normalmente acceduto con una politica fifo. Un processo che vuole leggere dal canale dovrà eseguire una istruzione *receive*, un processo che vuole scrivere dovrà eseguire un'istruzione *send*.
L'istruzione standard di *receive* fornita da spin ha la seguente sintassi:

nomecanale?var1,var2,…,varn,TIPOMESSAGGIO
(var1…varn non sono obbligatorie)

se il messaggio affiorante in coda rispetta questa sintassi (in particolare se *TIPOMESSAGGIO* corrisponde al tipo del messaggio affiorante in coda) il processo che ne ha fatto richiesta avrà facoltà di estrarre il messaggio affiorante, e *var1*,*var2*,…,*varn* assumeranno il valore del messaggio estratto dalla coda .
È inoltre possibile far sì che un processo lettore stia in attesa di un messaggio con un preciso valore delle variabili:

nomecanale?eval(var1),eval(var2),..,eval(varn),TIPOMESSAGGIO
(eval è una funzione unaria che torna il valore della variabile che prende come argomento)

in questo caso il processo lettore estrarrà il messaggio dalla coda soltanto se il messaggio affiorante avrà i particolari valori delle variabili *var1*, …, *varn* e se di tipo *TIPOMESSAGGIO.*
Oltre a quella standard il sistema fornisce altri tipi di *receive*:

nomecanale??MSG

viene effettuato un accesso in un punto casuale della coda e soltanto se il messaggio in quel punto della coda è di tipo MSG allora verrà estratto dal processo che ne ha fatto richiesta, altrimenti la coda non subirà modifiche.

nomecanale?[MSG]

Questa funzione restituisce un booleano : più precisamente restituirà true se il messaggio affiorante nella coda è di tipo MSG, false altrimenti (non c'è side effect).

nomecanale?<MSG>

Questa receive restituisce un messaggio di tipo MSG, se questo è affiorante nella coda, al processo che ne ha fatto richiesta, facendolo però restare in coda (potrebbe essere utilizzata per comunicazioni in broadcast).
Analogamente alla receive Spin fornisce, oltre alla send standard, una send che permette una scrittura ordinata nella coda:

nomeCanale!!var1,var2,…varn

in questo modo il messaggio sarà inserito in ordine lessicografico nella coda.

Infine esistono anche delle istruzioni per appurare se la coda dei messaggi è vuota, utili per evitare che un processo rimanga bloccato cercando di leggere un messaggio che non c'è:

empty(nomecanale) : true se nomecanale è vuoto, false altrimenti.

nempty(nomecanale) : false se nomecanale è vuoto, true altrimenti
(necessario perché la sintassi promela non permette la scrittura di !empty).

2. Scopo del progetto

Ciò che ci si è proposti di fare in questo lavoro, è fornire una metodologia, applicata su un semplice esempio, per far si che *n* lettori e scrittori potessero interagire mediante un unico canale, senza dover creare un canale dedicato per ciascuna comunicazione e garantendo al tempo stesso che non si verifichino situazioni di stallo, starvation o stravolgimento dell'ordine dei messaggi.
Nelle figure riportate in seguito è illustrato quello che potrebbe verificarsi se più processi leggessero sulla stessa coda :

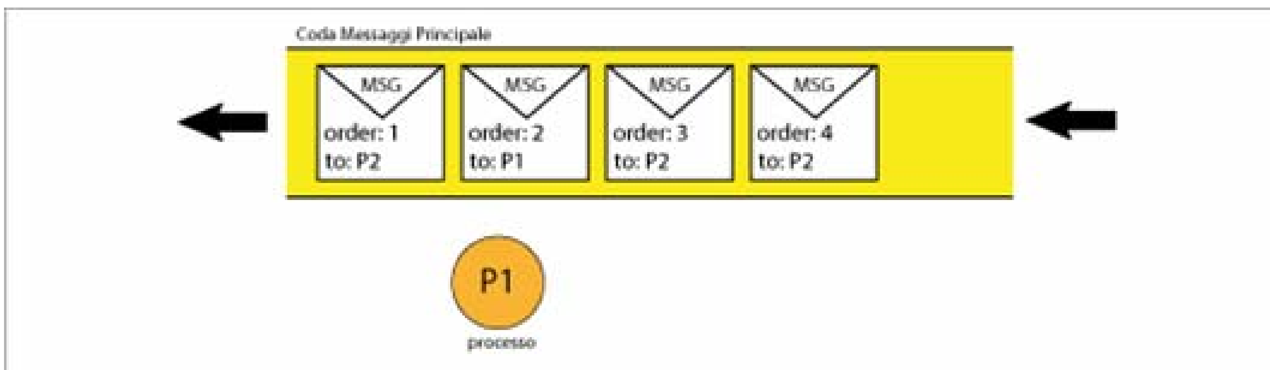


Figura1: un processo p1 ed una coda su cui hanno scritto p1 e p2.

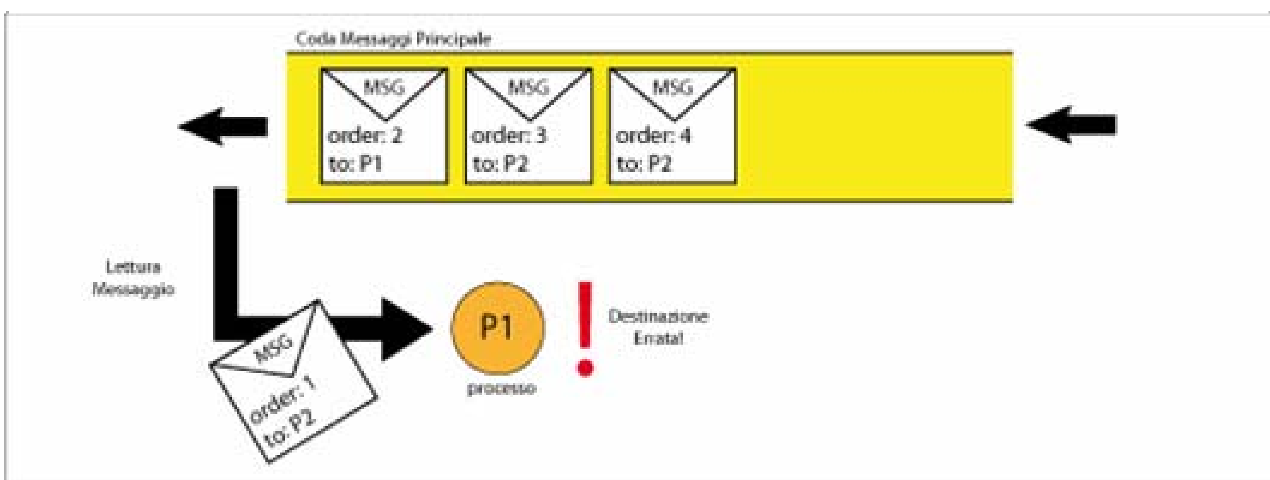


Figura2: p1 legge il messaggio affiorante dalla coda.

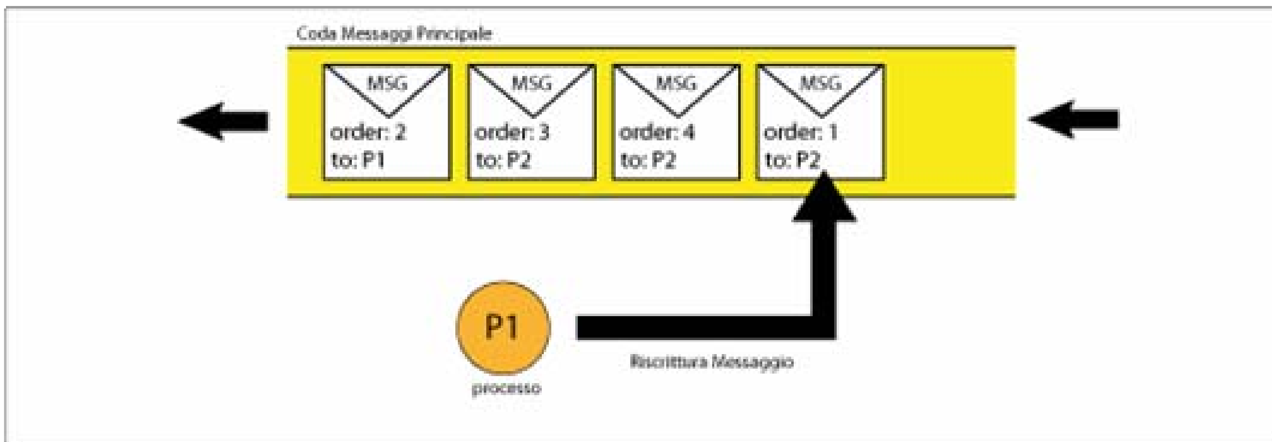


Figura3: p1 rimette il messaggio in coda perché non era indirizzato ad esso.

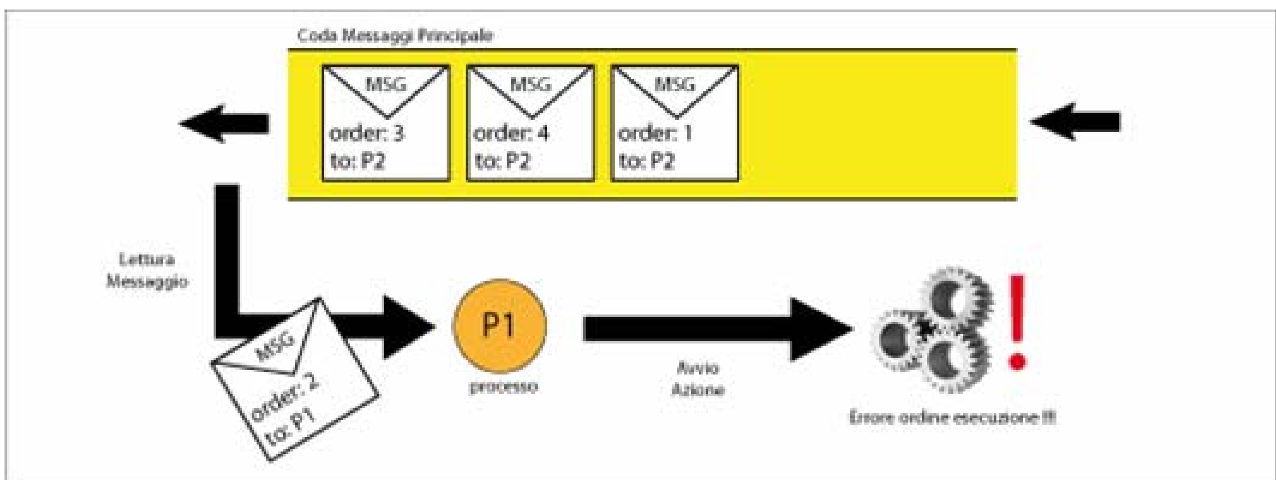


Figura4: p1 legge di nuovo il messaggio affiorante dalla coda che questa volta è indirizzato ad esso e fa qualche cosa; sfortunatamente l'ordine dei messaggi in coda è stato stravolto e così p2 gestirà il proprio messaggio solo dopo p1.

Di seguito forniamo tre algoritmi per la risoluzione del problema. Il diagramma uml che modella il sistema asincrono su cui essi sono stati applicati è riportato nelle figure 5,6,7 (i processi corrispondono ai sottosistemi sincroni mentre i messaggi scambiati fra i processi rappresentano gli eventi che fanno scattare le transizioni).
Negli algoritmi vengono inoltre utilizzate alcune istruzioni che promela fornisce per eseguire porzioni di codice in modo atomico:

atomic{sequenza istruzioni}
fa sì che l'esecuzione del codice fra parentesi graffe venga eseguita come una transazione

d_step{sequenza di istruzioni}
che è come atomic ma non permette di inserire nella sequenza di istruzioni anche cicli infiniti, istruzioni non deterministiche e salti (atomic lo consente).

Infine viene gestita la variabile di sistema di sola lettura timeout che diventa true nel momento in cui nessun processo può più eseguire (in genere si usa per uscire da situazioni di deadlock)[1].

[1] Nell'ultima versione del codice (quella allegata) timeout non è gestita perché si tratta del codice usato per misurare le prestazioni del sistema (era quindi necessario che tutti i processi terminassero).

3. Diagrammi Uml

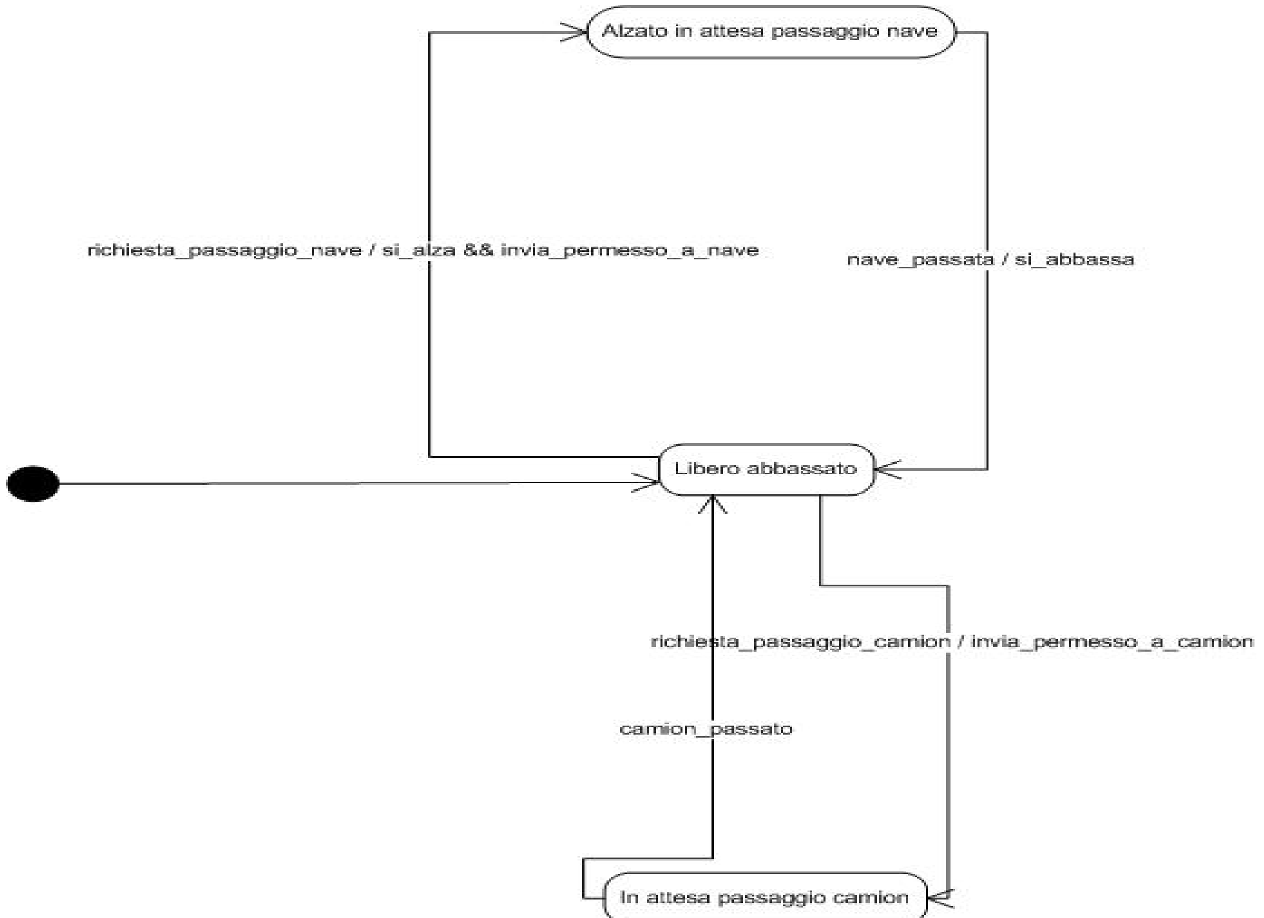


Figura5: il diagramma uml per il sistema ponte.

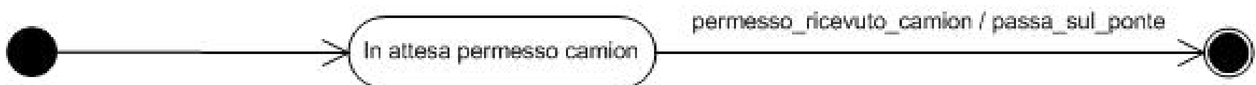


Figura6:il diagramma uml per il sistema camion.

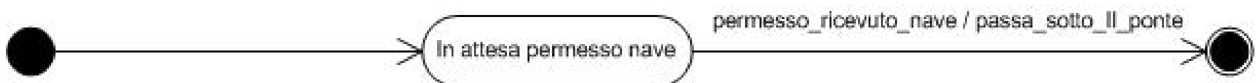


Figura7:il diagramma uml per il sistema nave.

4. Definizione degli algoritmi

4.1. Algoritmo 1

L'idea dell'algoritmo si basa su una creazione "meccanica" dei processi che compongono il sistema. Ogni processo rispecchia uno specifico diagramma degli stati e gestisce la lettura di messaggi in una coda comune rispettando un determinato ordine di esecuzione delle azioni dei messaggi anche in presenza di altri processi. Tale ordinamento (che chiameremo d'ora in poi semplicemente "ordine" del messaggio) viene intrinsecamente inserito all'interno del messaggio stesso dal sistema.
Per prima cosa si dovranno individuare le variabili atte a mantenere le informazioni interne del sistema stesso e a permettere l'interazione tra i vari processi che lo compongono. Altre variabili serviranno a gestire, con facilità, la trasformazione in modo meccanico di un diagramma di stato nel linguaggio adottato dal software spin, dando la possibilità di simulare una "navigazione" nel diagramma degli stati.
Un processo, dopo una fase di inizializzazione delle variabili ad esso associate, si addentrerà in un ciclo infinito (LOOP_LETTURA) nel quale leggerà un messaggio dalla coda (Coda_Messaggi). Dopo la lettura, il processo verificherà se il messaggio sia indirizzato a lui oppure no.
In caso di fallimento (il messaggio è diretto ad un altro processo) lo riscriverà nella coda mantenendo lo stesso valore di "ordine", così da garantire che venga mantenuto l'ordine di esecuzione dei messaggi della coda principale.
Nel caso in cui il messaggio sia destinato a lui il processo verificherà che sia quello con l'ordine giusto da eseguire, altrimenti lo riscriverà nella coda, mantenendo sempre lo stesso "ordine". Per verificare che il messaggio abbia l'ordine giusto si utilizza una variabile globale (current_order) che indica in ogni momento quale sia il messaggio con ordine più piccolo da eseguire.
Se il messaggio soddisfa le due condizioni precedenti (cioè che il messaggio è per il processo che lo legge ed è nell'ordine giusto di lettura) affinché si avvii l'azione corrispondente (cioè la transizione opportuna del diagramma uml) si dovrà verificare che il messaggio (cioè l'evento) sia quello atteso nello stato in cui il processo si trova e che le condizioni (se ci sono) siano verificate.
Nel caso si possa effettuare la transazione si avvierà un salto ad una porzione di codice che si preoccuperà di effettuare l'azione corrispondente al cambiamento di stato modificando tutte le variabili necessarie inclusa quella che rappresenta lo stato del processo ed inviare i messaggi ai processi con i quali si vuole avere un'interazione (si mettono i messaggi con gli eventi appositi nella coda).
Se invece la transizione non si può effettuare (l'evento non è atteso in quello stato) il messaggio viene reinserito nella coda ma viene incrementato l'ordine (in modo da non "ripescarlo" subito dopo).
Dopo aver elaborato il messaggio ed aver avviato le operazioni di gestione necessarie (sia se questo sia stato solo letto o se siano state avviate le azioni corrispondenti) si ritorna nuovamente all'inizio del loop per una nuova iterazione a meno che non sia stata richiesta la terminazione del processo (messaggio FINE).
Per ulteriori dettagli sulla traduzione del diagramma uml in codice promela si veda l'Appendice C.

4.2. Algoritmo 2

In quest'algoritmo viene utilizzata una send ordinata. Tutti i processi, inizialmente, inviano nel canale di comunicazione condiviso un messaggio di richiesta servizio con una priorità fissata maggiore di zero. Questo messaggio di richiesta viene inserito all'interno della coda attraverso una send ordinata. I messaggi saranno perciò ordinati in base alla priorità ad essi assegnata, e al loro pid.
Nel ciclo di lettura, ogni processo in attesa di un messaggio da leggere, estrae il messaggio affiorante dalla coda, controlla se è per lui e, se non è per lui, lo rimette in coda sempre attraverso una send ordinata.
In questo caso il fornitore del servizio può servire un processo per volta, pertanto, una volta che il processo fornitore del servizio avrà accolto la richiesta di un richiedente fra questi due processi verrà instaurata una comunicazione privilegiata, senza creare un nuovo canale, ma assegnando ai messaggi scambiati fra fornitore e fruitore una priorità privilegiata. Questi messaggi resteranno sempre in cima alla coda in quanto la send viene sempre eseguita tenendo conto dell'ordine lessicografico.
Svantaggio di questo algoritmo è l'overhead introdotto dalle molte estrazioni e reinserimenti nella coda.

4.3. Algoritmo 3

Quest'algoritmo differisce dal precedente in quanto, ora, un processo estrae un messaggio dalla coda se e, solo se, l'estrazione è eseguibile.
Questo controllo viene effettuato attraverso l'istruzione *nomecanale?[]* , se l'istruzione avrà esito positivo il processo interessato andrà a leggere dalla coda il proprio messaggio in modo atomico.
In questo caso si evita l'overhead introdotto dai molteplici inserimenti e reinserimenti nella coda.

5. Efficienza degli algoritmi in termini di tempo di esecuzione

Per verificare l'efficienza delle soluzioni proposte abbiamo misurato i tempi di esecuzione (per la simulazione del comportamento del sistema) di insiemi test di processi (vedi Appendice B).
In seguito riportiamo i grafici che riassumono l'efficienza dei singoli algoritmi e poi un grafico che pone a confronto le curve ottenute.

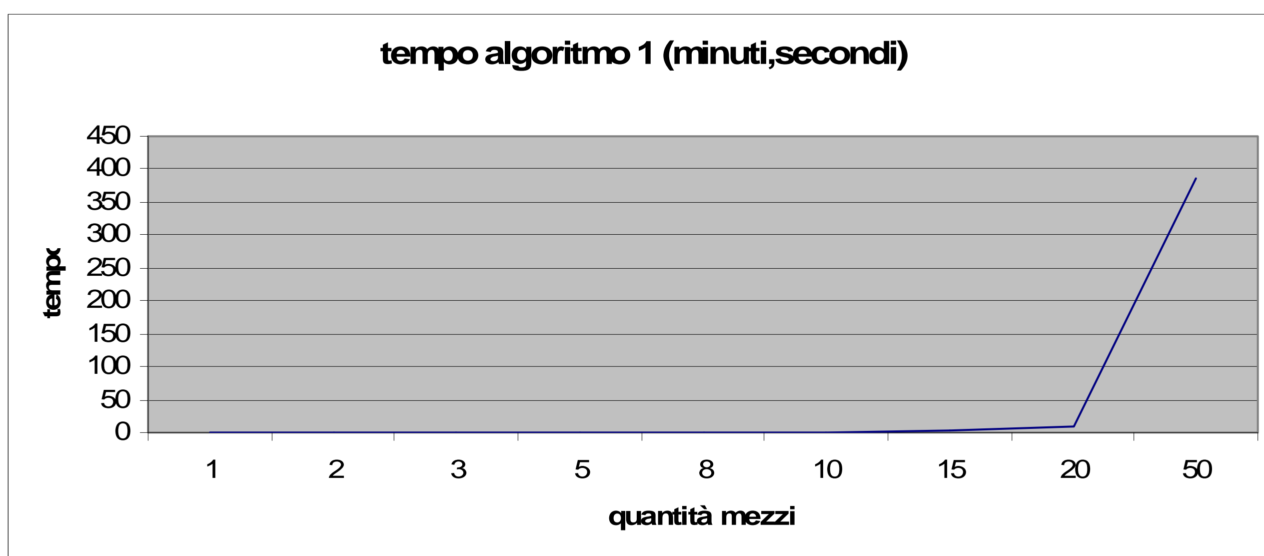


Figura8: tempo impiegato dall'algoritmo 1 per servire diversi insiemi di processi.

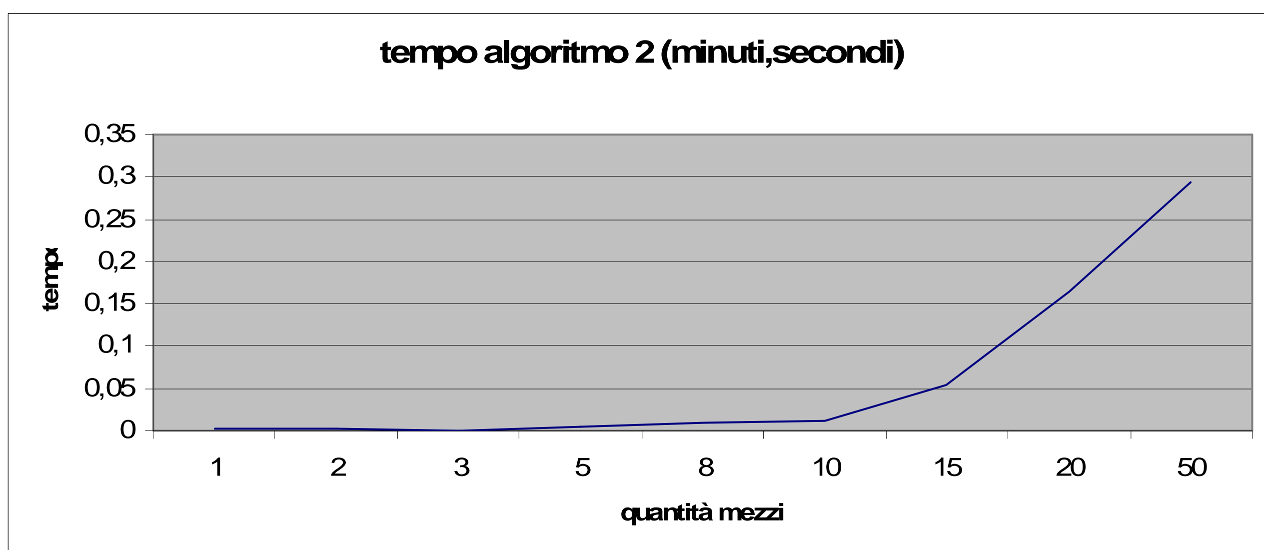


Figura9: tempo impiegato dall'algoritmo 2 per servire diversi insiemi di processi.

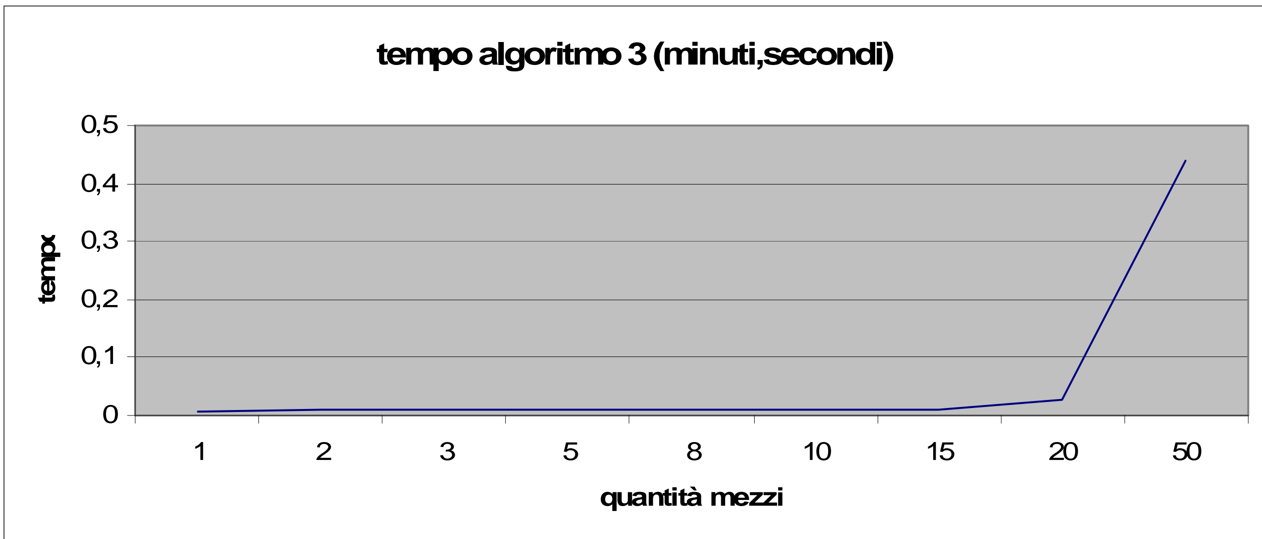


Figura10: tempo impiegato dall'algoritmo 3 per servire diversi insiemi di processi.

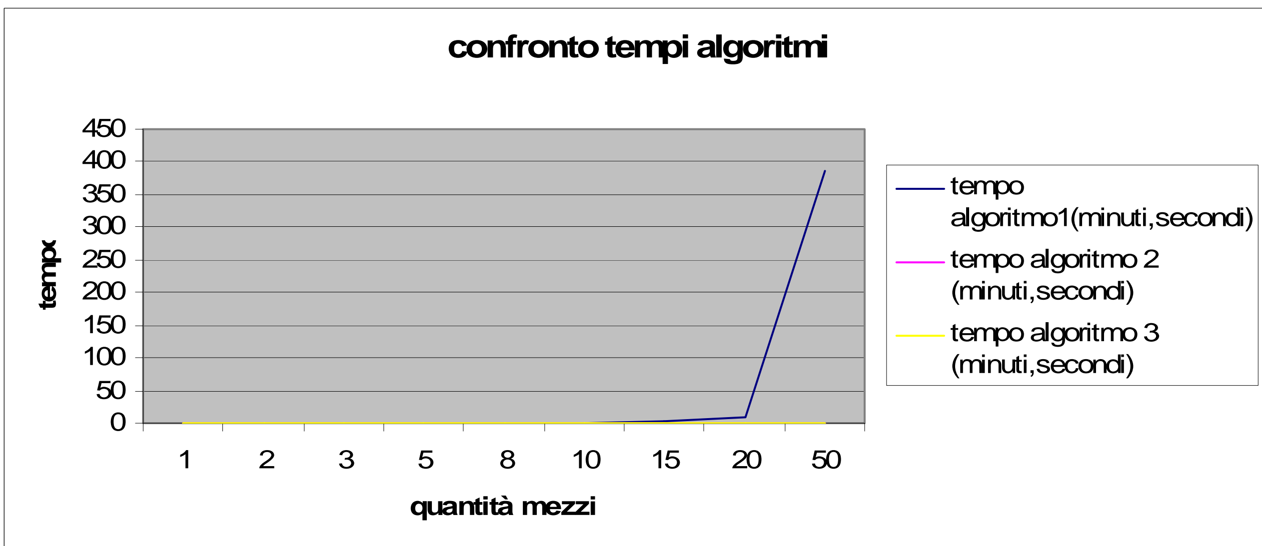


Figura11: confronto fra le curve ottenute (la due e la tre sono praticamente scomparse).

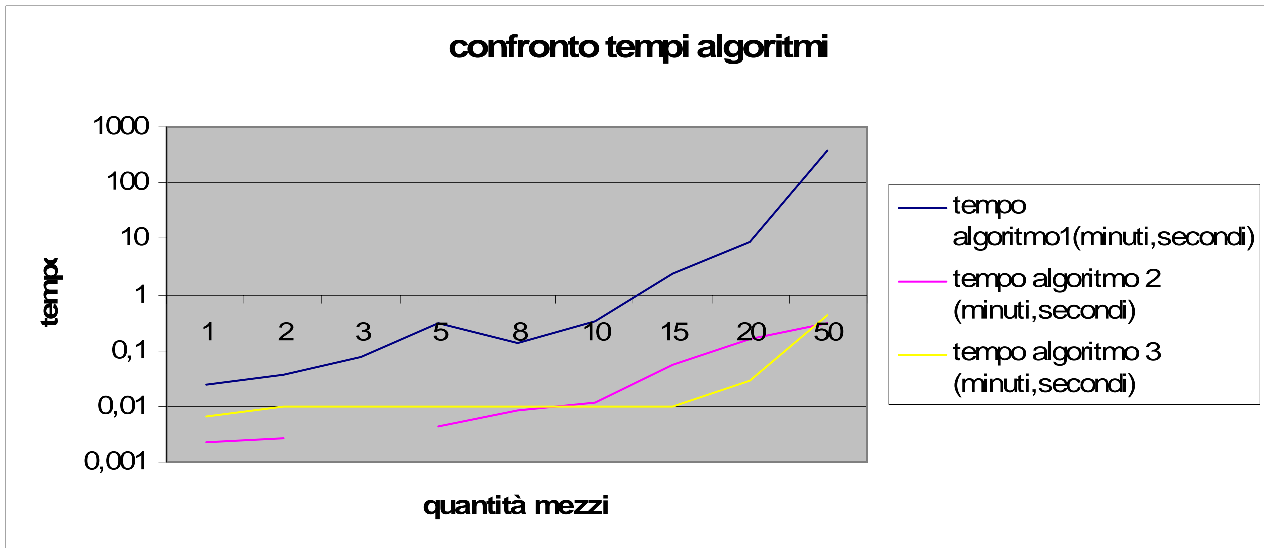


Figura12: confronto fra le curve ottenute in scala logaritmica (si nota che le curve due e tre sono nettamente al di sotto della uno).

6. Verifica di alcune proprietà ltl del sistema

È stato verificato che il sistema rispettasse alcune proprietà (espresse in logica ltl):

- i processi sono serviti in ordine
- una nave non passa sotto il ponte se il ponte è abbassato
- un camion non passa sopra il ponte se il ponte è alzato

La verifica è stata effettuata solo per tre processi in quanto la logica usata è proposizionale quindi non si possono esprimere proprietà per n individui a meno di scrivere n formule (tre formule ci sembravano ragionevoli).
Ovviamente le proprietà sono state espresse in modo diverso a seconda dell'algoritmo cui si riferivano e precisamente:

primo algoritmo

- <> (p -> s) && <> (a -> l) (se in futuro il ponte termina allora camion e nave hanno già terminato e se in futuro la nave termina allora il camion ha terminato; infatti i processi terminano subito dopo essere stati serviti ed il ponte termina quando non ci sono più processi da servire)
- [] ((p || a) -> ! l) (vale sempre che se il ponte è abbassato allora la nave non passa)
- [] (p -> ! l) (vale sempre che se il ponte è alzato allora il camion non passa)

secondo algoritmo

- <> (p -> (a && n)) && <> (a -> n) (se in futuro il ponte termina allora ponte e nave hanno già terminato e se in futuro la nave termina allora il camion ha terminato; infatti i processi terminano subito dopo essere stati serviti ed il ponte termina quando non ci sono più processi da servire)
- <>(n && p) -> ((n U (!p)) && (n U (!a))) (se in futuro la nave aspetta il permesso di passare ma il ponte sta facendo passare il camion allora la nave rimane in quello stato finchè il camion non è passato)
- <>(c && p) -> ((c U (!p)) && (c U (!a))) (se in futuro la nave aspetta il permesso di passare ma il ponte sta facendo passare il camion allora la nave rimane in quello stato finchè il camion non è passato)

terzo algoritmo

- le proprietà sono espresse come per il secondo

In particolare per il secondo algoritmo le proprietà sono state espresse in modo diverso rispetto al primo perché in questo caso XSPIN (l'interfaccia usata per il model checking) usando *globally* non riesce a terminare la ricerca segnalando che fino al punto in cui arriva la formula è non valida ma la ricerca non è stata completata.
Per il terzo algoritmo le proprietà da verificare sono espresse come per il secondo poiché il codice è molto simile ma sfortunatamente XSPIN non termina la ricerca per nessuna di esse quindi non abbiamo ottenuto prove formali.

In realtà nei numerosi test effettuati con diversi numeri di processi si verifica che solo l'algoritmo uno serve le richieste in ordine perciò possiamo avere una certa confidenza che l'algoritmo uno mantenga l'ordine per richieste superiori a due mentre per gli altri algoritmi non lo sappiamo (nel caso dell'algoritmo due abbiamo dimostrato formalmente che l'ordine viene mantenuto anche con quattro e cinque processi).
I file usati per verificare le proprietà ltl (modificati con etichette che indicano i vari punti del codice) e i file generati da XSPIN sono riportati in allegato.

7. Confronto fra le soluzioni proposte : vantaggi,svantaggi ed efficienza.

Di seguito riassumiamo e mettiamo a confronto in una tabella le principali proprietà degli algoritmi proposti (con **V** indichiamo quelle che possono considerarsi dei vantaggi e con **S** quelle che possono considerarsi degli svantaggi).

PROPRIETA'	**ALGORITMO1**	**ALGORITMO2**	**ALGORITMO3**
Linee di codice	***S**: In termini di linee di codice è il più complesso, necessita di molte variabili e la ricezione dei messaggi è molto complessa (bisogna estrarre il messaggio affiorante, controllare tipo e destinatario ed eventualmente reinserirlo).*	***V**: Meno linee di codice rispetto al precedente in quanto sono richieste poche variabili e la riscrittura dei messaggi in coda avviene in modo più semplice perchè la coda è ordinata (tuttavia si estrae ancora il messaggio affiorante e lo si controlla).*	***V**: Come il precedente ma le linee di codice sono ancora meno. Il messaggio affiorante dalla coda è estratto da un particolare processo sse il processo in questione è in attesa esattamente di quel messaggio.*
Ordine dei messaggi	***V**: L'ordinamento dei messaggi viene mantenuto gestendo un campo order del messaggio stesso con un algoritmo esterno.*	***V**: La coda è ordinata perché i messaggi vengono inseriti in modo ordinato sulla base dei campi del messaggio stesso.*	***V**: La coda è ordinata perché i messaggi vengono inseriti in modo ordinato sulla base dei campi del messaggio stesso.*
Carico	***S**: Il carico maggiore (cioè la ricezione) è sul ricevente.*	***S**: Il carico maggiore (cioè la ricezione) è sul ricevente.*	***S**: Il carico maggiore (cioè la ricezione) è sul ricevente.*
Inserimenti e reinserimenti	***S**: Molte estrazioni e reinserimenti in coda.*	***S**: Molte estrazioni e reinserimenti in coda.*	***V**: Il numero di estrazioni e reinserimenti è ridottissimo rispetto ai precedenti (un messaggio viene estratto solo se è del tipo desiderato).*
Starvation	***V**: Tutti i processi prima o poi vengono serviti.*	***V**: Tutti i processi prima o poi vengono serviti.*	***V**: Tutti i processi prima o poi vengono serviti.*
Ordine di servizio	***V**: I processi vengono serviti esattamente nell'ordine in cui hanno fatto la richiesta.*	***S**: Non c'è garanzia che due processi dello stesso tipo vengano serviti nell'ordine in cui hanno fatto la richiesta anche se prima o poi verranno serviti.*	***S**: Non c'è garanzia che due processi dello stesso tipo vengano serviti nell'ordine in cui hanno fatto la richiesta anche se prima o poi verranno serviti.*
Numero di processi serviti	***S**: Con un solo canale fino a 202 circa nel caso migliore (dovuto*	***S**: Con un solo canale fino a 202 circa nel caso migliore(dovuto al fatto che*	***S**: Con un solo canale fino a 202 circa nel caso migliore(dovuto al fatto che*

	al fatto che il canale in spin contiene al più 255 messaggi).	*il canale in spin contiene al più 255 messaggi) .*	*il canale in spin contiene al più 255 messaggi) .*
Tempi di esecuzione per il numero massimo di processi serviti	***S**:Circa 386m,0s,624msec (solo cinquanta a causa dell'eccessivo tempo richiesto).*	***S**: Circa 87m,23s,063msec.*	***V**: Circa 4s,400 msec.*
Proprietà LTL verificate	***V**: Se ci sono due processi diversi vengono serviti in ordine.* *Se il ponte è alzato un camion non passa.* *Se il ponte è abbassato una nave non passa.*	***S**: Se ci sono due processi diversi vengono serviti in ordine (con un numero maggiore di processi questa proprietà non è garantita).* *Se il ponte è alzato un camion non passa.* *Se il ponte è abbassato una nave non passa.*	***S**: nessuna delle proprietà precedenti può essere provata formalmente.*

8. Possibili sviluppi futuri

Il sistema realizzato in questo progetto potrebbe essere ampliato od anche solo modificato prendendo in considerazione aspetti che in questa tesina sono stati individuati ma non gestiti.

- Adottare una politica con master in scrittura e lettura
  Lo stesso sistema potrebbe essere implementato utilizzando un processo master che coordini la lettura dei messaggi leggendo lui stesso dalla coda principale e smistando poi successivamente il messaggio al processo a cui deve essere inviato. Tale politica può essere anche utilizzata per il processo di scrittura (utile per risolvere sistemi distribuiti o in cui si hanno dei ritardi notevoli).

- Rilevamento perdita di messaggi
  Dovendo far comunicare il master con i singoli processi attraverso una coda l'inconveniente in cui potremmo ricadere è quello in cui il messaggio viene inviato alla coda ma poi o viene perso o ci mette molto tempo prima di arrivare a destinazione. Il problema da risolvere in questo caso sarebbe la gestione dell'arrivo dei messaggi così da evitare sia che messaggi vengano persi e non rinviati, sia che lascino il master in attesa per un tempo che potrebbe essere anche infinito. Tale problema si potrebbe risolvere con l’inserimento di un ciclo di invio mediante ACK.

- Gestione interattiva dell’arrivo dei mezzi (creazione dei processi)
  Il sistema per mandare in esecuzione i vari processi non utilizza un programma utente ma semplicemente la funzione run per ogni processo che deve essere lanciato. In futuro potrebbe essere implementato un programma utente.

- Gestione più evoluta per condizioni complesse.

- Creazione di un sistema di verifica dei vincoli
  Alcuni vincoli del sistema possono non essere facilmente verificabili all’interno di un processo ma solo con una visuale globale del sistema (ad esempio verificare che nel sistema si possano avere al massimo due camion ed una nave). Per risolvere questo problema, si potrebbe creare un processo parallelo che si occupi, in ogni istante, di verificare che siano sempre rispettati tutti i vincoli del sistema e, quando non possibile, di ripristinare il funzionamento dello stesso.

- Gestione di una lista di attesa (ad esempio dei mezzi)
  Si potrebbe creare un sistema di buffer per mantenere l’ordine delle richieste di attesa. Ad esempio ad ogni arrivo di un nuovo mezzo si potrebbe mettere il messaggio di notifica dentro un’altra coda, unica per tutti i processi, che gestisce i processi in attesa, così da migliorare le prestazioni del sistema.

- Evoluzione degli algoritmi presentati per ottimizzare i tempi di computazione e le prestazioni generali del sistema.

9. Software utilizzato

- SPIN versione 430 giugno 2007, per l’esecuzione dei processi.
- XSPIN versione 430 giugno 2007 e MINGW versione 5.1.3 per la scrittura del codice promela.
- XSPIN versione 430 giugno 2007, per la verifica di proprietà ltl del sistema.
- Microsoft Visio versione 2003, per il disegno dei diagrammi uml.
- CGWIN versione 1.5.25-7, per la misura dei tempi di esecuzione del sistema.

10. Appendici

10.1. Appendice A : Particolari istruzioni promela usate

NOME
empty – funzione booleana per testare se un canale è vuoto.

SINTASSI
empty(*nomeCanale*)

DESCRIZIONE
Empty è una funzione predefinita che prende come argomento il nome di un canale e restituisce true se non ci sono messaggi nel canale e false altrimenti.
empty(nomeCanale)
È equivakente a
len(nomeCanale) == 0

ESEMPIO

```
chan q = [8] of { mtype };
d_step
{
        do
                :: q?_
                :: empty(q) -> break
        od;
        skip
}
```

NOME
nempty – funzione booleana per testare se un canale non è vuoto.

SINTASSI
nempty(*nomeCanale*)

DESCRIZIONE
Nempty è una funzione predefinita che prende come argomento il nome di un canale e restituisce true se non ci sono messaggi nel canale e false altrimenti.
L'uso di nempty è reso necessario dal fatto che la sintassi promela proibisce di scrivere !empty(nomeCanale).

nempty(nomeCanale)
è equivalente a
len(nomeCanale) != 0

NOME
Atomic - per definire un frammento di codice che deve essere eseguito in maniera atomica.

SINTASSI
atomic {sequenza di istruzioni}

DESCRIZIONE
- La sequenza di istruzioni racchiusa fra parentesi graffe viene eseguita in maniera atomica (l'esecuzione non è interleaved con altri processi).
- La sequenza può contenere qualsiasi istruzione promela (anche non deterministica).
- Se una qualsiasi istruzione della sequenza si blocca l'atomicità viene persa e prima che il processo possa tornare ad eseguire dovrà competere per le risorse con gli altri processi attivi.
- Se il blocco avviene su una istruzione rendezvous il controllo passa al ricevente e se anche la receive è in una sequenza atomica verrà completato il protocollo anziché permettere l'esecuzione ad altri processi.

ESEMPIO
```
Atomic
 {
        /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
}
```

NOME
D_step - per definire un frammento di codice che deve essere eseguito in maniera atomica.

SINTASSI
D_step {sequenza di istruzioni}

DESCRIZIONE
- La sequenza di istruzioni racchiusa fra parentesi graffe viene eseguita in maniera atomica (l'esecuzione non è interleaved con altri processi).
- La sequenza può contenere qualsiasi istruzione promela (tranne non deterministica).
- Se una qualsiasi istruzione della sequenza si blocca l'atomicità viene persa e prima che il processo possa tornare ad eseguire dovrà competere per le risorse con gli altri processi attivi.
- Se il blocco avviene su una istruzione rendezvous il controllo passa al ricevente e se anche la receive è in una sequenza atomica verrà completato il protocollo anziché permettere l'esecuzione ad altri processi.

ESEMPIO
```
D_step
{
         /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
}
```

NOME
send!!- per definire un messaggio in un canale in ordine lessicografico (in base ai campi del messaggio: prima il primo, poi il secondo, ecc.).

SINTASSI
nomeCanale!!msg

DESCRIZIONE
Inserisce il messaggio nel canale seguendo un ordine lessicografico basato sui campi del messaggio stesso.

ESEMPIO
```
x!!3; x!!2; x!!1; x!4;
x?1; x?2; x?3; x?4
```

NOME
poll?[msg]- serve per vedere se si può effettuare una receive; infatti restituisce true se il messaggio msg è presente nel canale, false altrimenti..

SINTASSI
nomeCanale[msg]

DESCRIZIONE
Restituisce true se msg è presente nel canale, false altrimenti.

ESEMPIO
```
nomeCanale?[msg]->skip;
```

NOME
timeout- variabile di sola lettura del sistema che segnala quando tutti i processi sono bloccati; tipicamente viene usata per uscire da situazioni di stallo.

DESCRIZIONE
'E true se nessun processo attivo può eseguire, false altrimenti.

ESEMPIO
```
active proctype watchdog()
{
	do
		:: timeout -> guard!reset
	od
}
```

NOME
eval(me)- trasforma il valore di una variabile in costante; tipicamente si usa nelle receive per vedere se il valore di una certa variabile è presente in un canale.

SINTASSI
eval(nomeVariabile)

DESCRIZIONE
Trasforma nomeVariabile in una costante corrispondente al valore assegnato alla variabile.

ESEMPIO
```
x = ack; q?eval(x)  /* same as: q?ack */
x = msg; q?eval(x) /* same as: q?msg */
```

NOME
receive?<MSG>

SINTASSI
nomeCanale?<MSG>

DESCRIZIONE
Restituisce al processo che ha fatto la richiesta il messaggio MSG se questo affiora dalla coda ma senza toglierlo dal canale.
Spesso viene usata per mandare messaggi in broadcast.
Esiste anche la variante receive??<MSG> che si comporta come receive?<MSG> ma prende il messaggio MSG qualsiasi sia la sua posizione nella coda.

ESEMPIO

```
	proctype p1{
	….
		nomeCanale?<msg>->printf("p1 ha letto\n");
	…
	}

proctype p2{
…
nomeCanale?<msg>->printf("p2 ha letto\n");
…
}

	/*se uno dei due processi legge prima dell'altro chi legge dopo non rimane bloccato perchè il
	messaggio è ancora nel canale*/
```

10.2. Appendice B : Tempi di esecuzione

Qui di seguito vengono riportati i tempi di esecuzione degli algoritmi in base al numero di processi che interagiscono nel sistema.
"Quantità di mezzi" indica il numero di processi totali che interagiscono nel sistema (non comprendendo il processo ponte poichè il numero di ponti è sempre pari ad 1 né il processo init che serve ad avviare tutti gli altri).

Algoritmo 1

quantità di mezzi (unità)	tempo di esecuzione (minuti.secondi.millisecondi)
1	0.0.244
2	0.0.367
3	0.0.741
5	0.3.147
8	0.13.968
10	0.33.077
15	2.4.276
20	8.8.903
50	386.0.624

**** A causa dell'elevato tempo richiesto già con cinquanta unità per questo algoritmo non è stato possibile calcolare i tempi di esecuzione per cento e duecento unità.*

Algoritmo 2

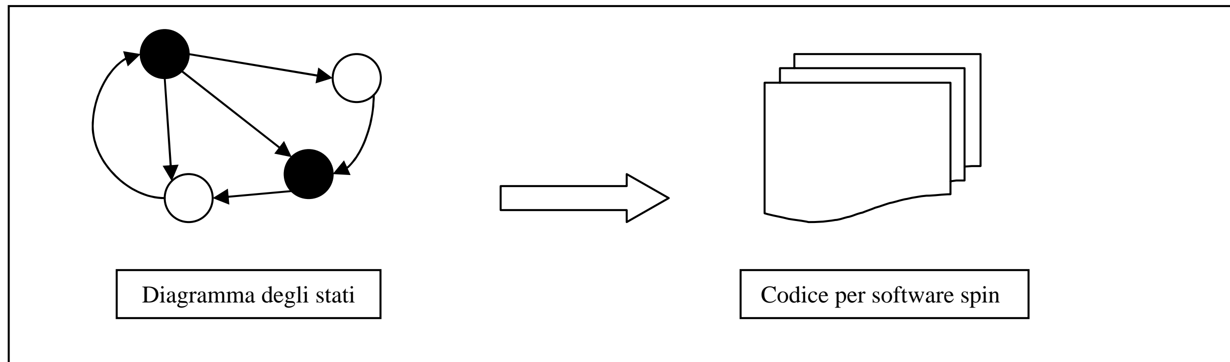
quantità di mezzi (unità)	tempo di esecuzione (minuti.secondi.millisecondi)
1	0.0.23
2	0.0.27
3	0.0.32
5	0.0.43
8	0.0.86
10	0.0.113
15	0.0.551
20	0.1.645
50	0.29.356
100	6.27.58
200	87.23.063

Algoritmo 3

quantità di mezzi (unità)	tempo di esecuzione (minuti.secondi.millisecondi)
1	0.0.67
2	0.0.100
3	0.0.100
5	0.0.100
8	0.0.100
10	0.0.100
15	0.0.100
20	0.0.283
50	0.0.600
100	0.1.450
200	0.4.400

10.3. Appendice C : Dettagli algoritmo 1

10.3.1. Dal Diagramma a stati all'implementazione



I passi base per scrivere il codice a partire dal diagramma a stati possono essere riassunti utilizzando i seguenti punti:

1) Si costruiscono delle variabili globali per ogni elemento che possono identificare lo stato globale del sistema o che possono essere utili per il normale funzionamento del sistema stesso:
   1-1. Una variabile per ogni oggetto che fa parte del sistema che indica il suo stato effettivo (esempio: posizione_ponte...).
   1-2. Variabili utili per verificare i vincoli e le condizioni presenti nei diagrammi degli stati o utili per poter procedere con l'esecuzione del sistema (esempio: discarica_capienza, discarica_occupata...ed altre eventuali variabili per le condizioni).

2) Per ogni stato del diagramma di stato si costruisce una costante (tramite define) del tipo "nome_diagramma_degli_stati_nome_stato" (esempio: #define PONTE_LIBERO_ABBASSATO); per ogni evento si definisce un'altra costante con nome "nome_diagramma_degli_stati_nome_evento" (esempio: PONTE_RICHIESTA_CAMION) e per ogni azione un'altra costante con nome "nome_azione", dove necessario.

3) Per ogni processo si costruisce una costante che identifica il processo stesso con nome tipo: ID_"nome_diagramma_di_stato" (esempio: ID_PONTE).

4) Si costruisce un processo con nome Processo_"nome_diagramma_di_stato" (esempio: proctype Processo_Ponte(Process_Id)), il quale prende in ingresso (quando si esegue run) l'identificativo del processo (la costante definita nel punto 3), cosi da poterlo identificare durante l'esecuzione del sistema (più eventuali altri parametri di ingresso necessari al processo).

5) Scrivo il processo per la gestione del diagramma di stato associato.
   5-1. Creo una variabile di stato locale per indicare in quale stato si trova attualmente il processo (esempio: stato_ponte).
   5-2. Inizializzo le variabili locali del processo e le variabili globali se necessarie.
   5-3. Avvio un ciclo LOOP per la lettura dei messaggi.
   5-4. Verifico se il messaggio è diretto a questo processo.

5-5. Nel caso negativo, "riscrivo" il messaggio in coda (salto direttamente al punto 5.10).
5-6. Altrimenti (se il messaggio è per il processo), verifico che il messaggio possa essere processato**.
5-7. Nel caso non sia il più vecchio, "riscrivo" il messaggio in coda (salto direttamente al punto 5.10).
5-8. Nel caso l'evento non sia gestito o non valgano le condizioni, "riscrivo" il messaggio (incrementando il suo ordine in questo caso!) nella coda ed incremento l'ordine del sistema per passare al prossimo messaggio da eseguire (il current_order).
5-9. Se il messaggio è idoneo eseguo un salto ad una label (eseguendo un salto al punto 5.11) per gestire la transazione di stato per quella determinata azione (con nome label tipo: "nome_diagramma_degli_stati_nome_stato_di_partenza_nome_evento", ad esempio PONTE_STATO_LIBERO_ABBASSATO_RICHIESTA_CAMION).
5-10. Salto nuovamente al ciclo di lettura (punto 5-3). (Qui si arriva nel caso si verificassero degli errori!)
5-11. Per ogni transazione esistente costruisco una label: "nome_diagramma_degli_stati_nome_stato_di_partenza_nome_evento" (ad esempio: PONTE_STATO_LIBERO_ABBASSATO_RICHIESTA_CAMION) la quale:
- 5.11.1. verifica le condizioni
- 5.11.2. esegue l'azione corrispondente
- 5.11.3. esegue il passaggio di stato necessario
- 5.11.4. incrementa il current_order
- 5.11.5. salta nuovamente al ciclo di lettura (punto 5-3)

6) Si scrive il processo init per inizializzare il sistema:
6-1. Inizializzo le variabili di stato generale del sistema.
6-2. Avvio i processi assegnandogli gli identificativi scelti al passo 3.

** Per essere processato il messaggio deve possedere le seguenti caratteristiche:
1) deve essere il messaggio più "vecchio" tra tutti i messaggi presenti nella coda***;
2) devono essere contemporaneamente soddisfatte le condizioni.

*** Il messaggio più vecchio può essere ricavato inserendo ad ogni messaggio un intero "msg_order" che è univoco per tutti i messaggi e viene incrementato ogni volta che un nuovo messaggio viene scritto nella coda dei messaggi generali del sistema (ATTENZIONE: viene scritto per un nuovo messaggio! Se viene riscritto un messaggio nella coda questo mantiene lo stesso order!!!). Si utilizzano delle variabili globali "current_order" e "max_order" per identificare rispettivamente l'order che si deve processare in questo momento (si presuppone che sia il più piccolo!) ed il numero massimo dell'order finora assegnato (cosi da sapere sempre quale è il prossimo order da assegnare!).

NOTA: Supponiamo che l'order di ogni messaggio sia corretto, ovvero che se nella realtà ho che prima deve essere abbassato il ponte e poi può passare il camion allora i "msg_order" devono stare rispettivamente ad 1 e 2, cioè la coda può contenere messaggi non ordinati ma gli order dei singoli messaggi devono corrispondere alla realtà (il sistema non riordina anche gli order non corrispondenti alla realtà ma li prende come buoni, tranne in casi particolari).

10.3.2. Definizione del Processo

```
proctype Processo_NOME_DIAGRAMMA_STATI ( int Process_Id )
{
  inizializzazione delle variabili locali
  inizializzazione stato iniziale processo
  LOOP_LETTURA: ……………………………………………………………………………… inizio ciclo di lettura dei messaggi
  {
    lettura del messaggio
    if ( messaggio.destinazione != Process_Id )……………………… ……………………verifica di appartenenza del messaggio
    {
      riscrittura del messaggio nella coda di sistema……………………………………… verifica fallita (il messaggio non e' per questo processo)
    }
    else  ………………………………………………………………………………………destinazione esatta
    {
        if ( messaggio.ordine > sistema.ordine_corrente ) … ……………………….……verifica della sequenzializzazione dei messaggi (msg_order)
        {
          riscrittura del messaggio nella coda di sistema …………… messaggio ordine maggiore di quello da eseguire (msg_order > current_order)
        }
        else ………………………………………………………………………….……… è il messaggio da eseguire
        {
            if ( gestito(messaggio.evento) ) ………………………………………………….verifica se l'evento è gestito
            {
              salto alla transazione di stato associata (codice sottostante) … …………messaggio gestito allo stato corrente del processo
            }
            else ………………………………………………………………………………. messaggio non gestito
            {
                riscrittura del messaggio nella coda di sistema incrementando l'ordine (del messaggio e quello corrente)
            }
        }
    }
   ritorno nel LOOP_LETTURA;

  gestione delle transazioni di stato …………………………eseguo l'azione, la transazione, modifico lo stato del sistema e torno nel loop di lettura
  {
    if ( not precondizioni transazione soddisfatte ) …………verifica delle precondizioni
    {
      gestisco l'errore
    }
    else
    {
      eseguo azione
      cambio le variabili esterne
      invio i messaggi agli altri processi per l'interazione
      cambiamento di stato
      indico di passare al prossimo messaggio da gestire (incrementando l'ordine di esecuzione : current_order++)
    }
    ritorno nel LOOP_LETTURA o salto allo stato di terminazione del processo (salta in STATO_FINE)
  }

 STATO_FINE:
 {
   gestisce la terminazione del processo
 }
}
```

11. Riferimenti

- Sito di spin : http://spinroot.com/spin/whatispin.html
- SPIN Beginners' Tutorial ,Theo Ruys.
- Advanced SPIN Tutorial ,Theo Ruys & Gerard Holzmann.
- Tesina del corso di MFIS "Modellazione e verifica formale di algoritmi per la mutua esclusione in ambiente distribuito", F. Delle Fave & V.Gheri.
- Tesina del corso di MFIS "Verifica formale del software:SPIN", D.Nifosi.