di Andrea de Prisco

Il modello CSP

CSP sta per Communicating Sequential Process e rappresenta un modello per esprimere la semantica della comunicazione inter process «a nomi espliciti dei moduli» vista lo scorso mese su queste stesse pagine. Vedremo, questo mese, anche alcuni esempi di programmazione multitasking utilizzando appunto il modello CSP. Parleremo dunque di processi, porte, messaggi e del controllo del non determinismo.

Processi Sequenziali

La «S» di CSP sta appunto per «Sequential». Ma non avevamo a che fare con processi «paralleli»? Sembra che ci sia una bella fregatura, sotto, ma non è vero. Infatti ogni processo è, in sé, sequenziale; ma tanti processi sequenziali eseguiti insieme formano una applicazione parallela. Detto in altre parole ogni processo è formato da una «sequenza» di istruzioni da eseguire rigorosamente l'una dopo l'altra, ma più processi possono essere eseguiti «parallelamente» e sono tra loro in grado di comunicare (non sottovalutate la «C» di «CSP»).

Ora, non è per stravolgervi immediatamente le idee, sappiate che la «sequenzialità» di un processo CSP è a livello di processo, ma non di istruzioni. Ovvero un processo, è formato da una sequenza di istruzioni, ma queste possono essere al loro interno «parallele».

Il caso limite è del processo CSP che è formato da un solo comando parallelo: nel momento in cui il processo parte, tutte le istruzioni che compongono il comando parallelo sono eseguite contemporaneamente. Quindi la sequenzialità dei processi CSP è più a livello semantico che non nella pratica.

Send e Receive

Nel modello CSP i comandi di Send e Receive per inivare e ricevere messaggi hanno la seguente sintassi:

```
NomeProcesso!Messaggio
```

corrisponde a Send(NomeProcesso, Messaggio) mentre:

```
NomeProcesso?Variabile
```

corrisponde a Receive(NomeProcesso, Variabile). In pratica, a distinguere il comando di Send da quello di Receive è solo il segno di interpunzione tra i due parametri che nel primo caso è un punto esclamativo, nel secondo un punto interrogativo. In figura 1 trovate un esempio di comunicazione tra due processi CSP: il processo mittente invia 100 valori al processo destinatario che provvede a stamparli; la sequenza di valori termina con l'invio della costante 0 (zero) da parte di A che viene riconosciuta da B come marcatore di fine flusso.

La comunicazione, in questo modello, è simmetrica, sincrona. Ovvero avviene sempre solo tra due partner (un mittente e un destinatario) e l'istante logico in cui l'operazione è completata è lo stesso per tutt'e due. Se il mittente o il destinatario arriva in anticipo all'appuntamento per lo scambio messaggio, attenderà il proprio partner prima di completare l'operazione e di procedere con l'istruzione successiva.

Se il partner di una comunicazione (sia questo il mittente oppure il destinatario) termina prima di aver portato a termine la stessa, il comando di I/O fallisce facendo terminare con fallimento anche il processo in attesa. Se ad esempio, tornando nuovamente alla figura 1, a causa di un errore dell'istruzione «write» del processo B questo termina prima di aver ricevuto e stampato tutti i messaggi inviati da A, anche il mittente termina (con fallimento) nel momento in cui tenta di eseguire la Send col partner ormai non più in esecuzione.

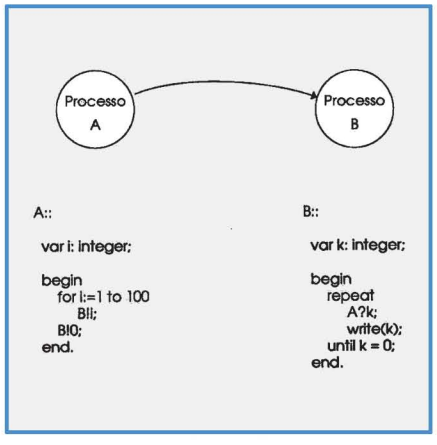

Figura 1 - Esempio di comunicazione tra processi CSP (vedi testo).

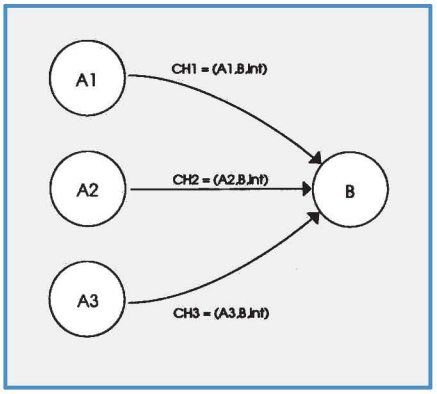

Figura 2 - Controllo del nondeterminismo: il processo B riceve messaggi da tre processi mittenti. Grazie ai comandi con guardia è possibile forzare scelte determinate.

Controllo del nondeterminismo

Sul numero scorso abbiamo anticipato il problema del controllo del nondeterminismo.

Ovvero come pilotare adeguatamente le scelte del sistema davanti a situazioni non definibili a priori e riguardanti la comunicazione tra un processo destinatario e molti processi mittenti attraverso altrettanti canali di comunicazione.

Nel modello CSP esistono le cosiddette «guardie» che sono generalmente composte da due componenti: una espressione booleana e un comando di ingresso (una Receive).

Una guardia può anche essere composta da solo una delle due componenti; se il comando di ingresso è presente, la guardia è detta «guardia di ingresso».

Le guardie sono utilizzate, come vedremo meglio in seguito, in due tipi di comandi: il comando alternativo e il comando ripetitivo.

Il primo serve per effettuare un'unica scelta tra un certo numero di possibilità e poi proseguire nell'elaborazione; il secondo serve per eseguire un loop sulle alternative che ciclo dopo ciclo evolveranno in maniera diversa. Ma per il momento ritorniamo sulle nostre guardie.

Una guarda d'ingresso, dicevamo, è formata da una coppia «espressione booleana — comando d'ingresso». Essa può trovarsi in uno solo dei seguneti tre stati: sospesa, fallita, verificata.

Una guardia è sospesa quando l'espressione booleana è vera (o al più assente) e il comando d'ingresso attende la comunicazione da parte del partner tuttora in esecuzione (non terminato).

Una guardia è fallita se l'espressione booleana è falsa oppure il processo partner indicato nel comando d'ingresso è terminato.

Una guardia si dice verificata se l'espressione booleana è vera (o assente) e il mittente indicato ha tentato la comunicazione con noi.

Ad ogni guardia è associata, sia nel caso del comando alternativo che in quello ripetitivo, una lista di comandi da eseguire in seguito alla scelta da parte del sistema di quella determinata guardia verificata. In pratica il controllo del non determinismo avviene indicando una lista di guardie e una corrispondente serie di liste di comandi da eseguire.

La sintassi del comando alternativo è questa:

```
[ G1 -> LC1
[] G2 -> LC2
:
:
[]Gn -> LCn];
```

dove le varie Gx rappresentano le guardie e LCx le liste di comandi ad esse associate. Facciamo un primo esempio: il processo B riceve messaggi dai processi A1, A2, A3. Questo che segue è un pezzo di codice CSP per eseguire comandi diversi a seconda del mittente del messaggio ricevuto, senza ulteriore filtro da parte di espressioni booleane.

```
B::
var msg: integer;
begin
[ A1?msg -> write("ricevo da A1")
[] A2?msg -> write("ricevo da A2")
[] A3?msg -> write("ricevo da A3")
];
end.
```

L'effetto è quello di stampare la stringa corrispondente al processo che ha inviato il messaggio. La domanda da porsi, a questo punto, è sicuramente questa: cosa succede nel caso in cui nessun mittente sta spedendo messaggi oppure che tutti i mittenti sono terminati o, ancora, che più d'un mittente contemporaneamente invia messaggi a B? È il caso, per l'appunto, delle guardie tutte sospese, tutte fallite o con più d'una guardia verificata. Nel primo caso il processo destinatario semplicemente rimane in attesa che l'attuale situazione di «tutte le guardie sospese» evolva in una delle due possibili ulteriori alternative. Nel caso di guardie tutte fallite, il comando alternativo fallisce e quindi il processo destinatario, conseguentemente, termina anch'esso con fallimento. Il terzo caso, il più interessante, è quello della contemporanea verifica di più guardie: in questo caso sarà il sistema a scegliere il comando di ingresso da eseguire e conseguentemente la lista di comandi corrispondente. Da segnalare il fatto che non è possibile per il processo destinatario (che ha eseguito il comando alternativo) forzare in alcun modo la scelta che spetta di diritto al sistema.

L'unico modo è quello di utilizzare guardie con espressioni booleane in modo da limitare (al limite a una sola) le guardie verificabili, fatto salvo però il fatto che comunque se ci sono più guardie verificate anche aggiungendo le espressioni booleane toccherà comunque al sistema sceglierne una. Proviamo, nell'esempio appena visto, ad aggiungere anche tre espressioni booleane: tre variabili di tipo bool opportunamente inizializzate.

```
B::

var msg: integer;
var a1,a2,a3: bool;

begin
a1:= TRUE;
a2:= FALSE;
a3:= TRUE;

[ a1,A1?msg -> write("ricevo da A1")
[] a2,A2?msg -> write("ricevo da A2")
[] a3,A3?msg -> write("ricevo da A3")
];
end.
```

La seconda guardia «a2,A2?msg» è fallita (in partenza) valendo FALSE la sua espressione booleana. Dunque sono in gioco solo la prima e la terza guardia. Se avessimo voluto forzare la scelta su un determinato mittente sarebbe bastato porre a TRUE una sola delle tre espressioni booleane. È chiaro che tutto ha un senso proprio perché queste espressioni sono pilotabili a tempo di esecuzione, in funzione dello stato interno del processo destinatario.

Dal comando alternativo si esce, dunque, quando almeno una guardia si verifica e, conseguentemente, un comando di ingresso con relativa lista comandi sono eseguiti.

Il comando ripetitivo, dal canto suo, ha una sintassi pressoché identica a quella del comando alternativo. L'unica differenza consiste in un asterisco posto all'inizio del comando:

```
*[ G1 -> LC1
 [] G2 -> LC2
 :
 :
 []Gn -> LCn];
```

Ben diversa è, invece, la semantica: il comando ripetitivo non viene eseguito una sola volta ma è continuamente rieseguito fino a quando tutte le guardie non sono fallite. E di solito è usato proprio per eseguire loop «principali», come nell'esempio di un processo buffer mostrato in figura 3. Il programma CSP che descrive quel processo buffer è quantomai semplice: si tratta infatti di un buffer a un solo ingresso e una

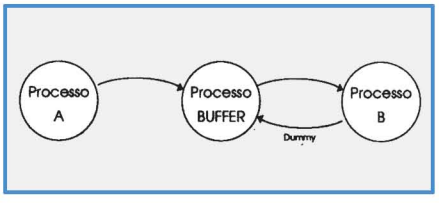

Figura 3 - Processo Buffer tra un produttore e un consumatore. Il messaggio di «pura sincronizzazione» Dummy serve per richiedere al processo Buffer un nuovo dato immagazzinato.

sola uscita dove quindi accede un solo processo produttore e un solo processo consumatore. Il buffer vero e proprio è contenuto nel processo stesso, sotto forma di un array ad «n» posizioni. Per semplicità supponiamo che i dati inviati e prelevati dal buffer siano dei normalissimi interi. Proviamo a scrivere il (semplicissimo) codice:

```
BUFFER::

var buffer: array[0..255] of integer;
var in, out, tot, tmp : integer;

begin
in    := 0;
out   := 0;
tot   := 0;

*[  tot<256,A?buffer[in]
       -> in := (in+1) AND 255;
          tot := tot + 1;
 [] tot>0,B?tmp
       -> B!buffer[out];
          out := (out+1) AND 255;
          tot := tot - 1;

 ];
end.
```

Questo esempio dovrebbe chiarire anche eventuali lacune sul funzionamento delle guardie e del comando ripetitivo. Commentiamolo un po'. Anzi, prima di descrivere il comportamento interno del processo buffer vediamo cosa succede all'esterno. Il processo «A» (produttore) per inserire i suoi output nel buffer non fa altro che inviare a questo il valore da inserire. In pratica effettua una semplice «send» sul processo buffer, diciamo così:

```
BUFFER!valore;
```

il processo «B» (consumatore) non può effettuare direttamente una «Receive» per prelevare i dati dal buffer, ma ne effettua richiesta inviando un messaggio di sincronizzazione che ha il solo scopo di comandare al buffer di spedirgli il prossimo valore. In pratica per ricevere il dato «B» eseguirà la seguente coppia di istruzioni:

```
BUFFER!dummy;
BUFFER?valore;
```

dove «dummy» è una costante qualsiasi di tipo intero e «valore» e una variabile targa, sempre di tipo intero, nella quale al termine del comando di ingresso troverà il valore letto nel buffer. Detto questo la descrizione del processo buffer è semplicissima: in pratica nel loop indotto dal comando ripetitivo il processo buffer riceve solo richieste di inserimento se il contatore «tot» (che contiene costantemente il numero di elementi presenti nella coda) è nullo, solo richieste di estrazione se il contatore è al suo massimo, 256, di inserimento o di estrazione per valori intermedi. Il tutto «automaticamente regolato» dalle guardie che dinamicamente falliscono, si verificano o sono sospese, a seconda dello stato del buffer e dell'attività dei processi produttore e consumatore. Le variabili «in» e «out» contengono rispettivamente la posizione di inserimento ed estrazione nel buffer e sono incrementate di volta in volta modulo 255 trattandosi di un buffer circolare. Il comando ripetitivo termina quando tutt'e due le guardie falliscono: non essendo possibile ciò a causa dell'espressione logica (non è possibile che «tot» sia contemporaneamente superiore di 255 e inferiore di 1) questa situazione si potrà verificare solo in caso di terminazione di almeno uno dei due processi partner e relativo riempimento o completo svuotamento della coda. Cosa che è, evidentemente, più che giusta: a che servirebbe mai un buffer completamente pieno al quale nessuno accede più per svuotarlo oppure un buffer completamente vuoto nel quale nessuno mai inserirà più nulla?

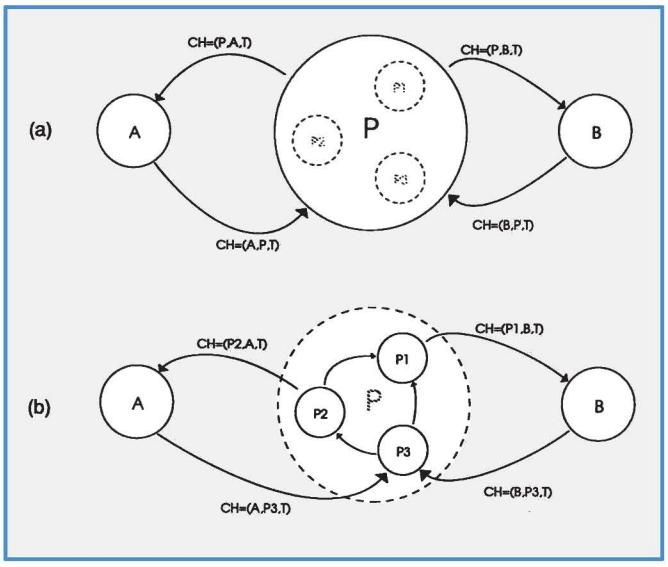

Figura 4
Grazie al comando parallelo (vedi testo) è possibile creare processi figli. Nel disegno il processo P (figura 4a) crea tre figli P1, P2, P3 (figura 4b). La «relazione di parentela» col processo che li ha creati permette loro di sostituirsi come mittenti o destinatari nelle comunicazioni coinvolgenti il padre, senza che i relativi partner si accorgano di ciò.

Comando parallelo

Per finire questa puntata di Multitasking diamo uno sguardo anche al comando parallelo citato all'inizio dell'articolo. Si usa per lanciare in parallelo processi figli: in pratica un processo può ad un certo punto lanciare altri processi e continuare la sua esecuzione quando tutti i processi da esso creati sono terminati. Il comando parallelo, infatti, è un comando come gli altri e dunque l'esecuzione delle istruzioni seguenti può avvenire solo a terminazione del comando corrente. La sintassi è simile a quella dei comandi visti prima, ovviamente non sono presenti guardie:

```
ProcessoPadre::
[ ProcessoFiglio1
||ProcessoFiglio2
 :
 :
|| ProcessoFiglioN];
```

In pratica quando viene incontrato nel processo padre un comando parallelo, l'esecuzione di questa viene sospesa ed inizia quella di tutti i processi figli, in parallelo. Questi sono essi stessi processi a tutti gli effetti e possono creare a loro volta altri figli e/o comunicare tra loro. La relazione di «parentela» col padre permette loro di sostituirsi come mittenti o destinatari di comunicazioni senza che i processi «esterni» si accorgano di ciò. Se al termine di tutti i processi figli (subordinata a loro volta dalla terminazione di eventuali processi nipoti) nessun processo è fallito il processo padre può continuare la sua esecuzione «sequenziale». Nel caso, invece, che uno o più processi figli termini con fallimento esistono due possibilità: la prima produce il fallimento anche del processo padre (e quindi il fallimento si propaga) la seconda prevede una lista di comandi per la gestione dell'eccezione.

Conclusione

Dal prossimo numero lasceremo da parte tutti gli sforzi «teorici» per dedicarci più da vicino agli aspetti «pratici». Non è escluso che vi coinvolgeremo anche con quiz e domande su problemi inerenti la programmazione parallela. Il nostro cavallo di battaglia sarà comunque l'OCCAM, il linguaggio di programmazione dei Transputer, con il quale vi mostreremo perfino la «magia» del calcolo parallelo. Speriamo solo di non stufarvi tanto presto: le cose da vedere sono ancora tante e, secondo noi, sempre più interessanti. Arrivederci. MC