Giornate di studio su:

"Programmazione Strutturata: Esperienze e Orientamenti"

Milano
23-24-25 giugno 1976

"Metodi per la specifica del coordinamento dei processi concorrenti"

P. Ancilotti, N. Lijtmaer
Istituto di Elaborazione della Informazione del Consiglio Nazionale delle Ricerche, Pisa.

M. Boari
Istituto di Automatica della Facoltà di Ingegneria, Università di Bologna.

Honeywell
Honeywell Informatica Systems Italia

1. *Introduzione*

Gli obiettivi dell'utilizzazione di tecniche di programmazione strutturata in un ambito di processi concorrenti sono sostanzialmente [1], [2] :

a) facilitare la stesura e la comprensione dei programmi indipendentemente dalla velocità relativa della loro esecuzione;
b) semplificare l'applicazione di metodi sistematici per l'analisi e la verifica dei programmi.

Il raggiungimento di tali obiettivi è sicuramente facilitato se si dispone di un linguaggio di programmazione dotato di opportuni costrutti sintattici, il cui impiego semplifichi al programmatore la soluzione dei vari problemi di sincronizzazione e renda eliminabili in fase di compilazione la maggior parte di errori "dipendenti dal tempo".

L'uso ad esempio di costrutti sintattici quali le *regioni critiche semplici e condizionali* [1], [9] per sincronizzare l'accesso da parte di più processi concorrenti alle risorse del sistema si è rivelato adatto nella progettazione di piccoli sistemi multiprogrammati dedicati a particolari applicazioni. Questi sistemi sono infatti caratterizzati da un numero limitato e fissato di processi le cui interazioni sono relative a poche risorse comuni e comunque poco frequenti; è lecito quindi consentire ai processi di controllare esplicitamente l'accesso alle risorse comuni utilizzando le *regioni critiche semplici e condizionali*.

Nel caso in cui è elevato il numero dei processi, che possono inoltre essere creati ed eliminati dinamicamente, ed è elevata la frequenza di utilizzazione delle risorse comuni del sistema (è questo il caso ad esempio dei sistemi operativi), consentire ai singoli processi un accesso non controllato a variabili comuni contenenti informazioni relative allo stato delle risorse e degli altri processi, rende più difficile la leggibilità dei programmi e più complessa la verifica della loro correttezza.

Una prima soluzione a questi problemi fu data da Dijkstra mediante l'introduzione del concetto di *segretaria* [5] . Secondo Dijkstra affinchè risulti più facilmente comprensibile l'evoluzione dei singoli processi sequenziali è necessario che ognuno di essi operi su variabili che gli appartengono unicamente. Le variabili comuni descriventi lo stato di ciascuna risorsa devono anch'esse appartenere ad un unico modulo, la *segretaria*, assieme alle procedure che operano su tali dati. I singoli processi che Dijkstra chiama i *direttori* non devono più avere una conoscenza esplicita delle variabili comuni ma accedono ad esse solamente u-

tilizzando le "azioni" (procedure) della *segretaria*.

Questo concetto di *segretaria*, se opportunamente formalizzato come costrutto linguistico, consente innanzitutto di nascondere ad un certo livello di astrazione i problemi di gestione delle risorse consentendo al programmatore di concentrarsi sulla struttura logica del suo programma; in secondo luogo potendosi verificare a livello di compilazione che solamente le procedure della *segretaria* hanno accesso ai dati comuni, è possibile garantire la consistenza logica della loro struttura mediante la verifica di relazioni di invarianza prima e dopo ogni chiamata alle procedure.

Il concetto di *segretaria* di Dijkstra rappresenta il primo passo verso la introduzione del concetto di *tipo di dati astratto* [6], nel campo della multiprogrammazione. Un *tipo di dati astratto* può infatti essere concepito come un insieme di operazioni (procedure) associate ad una struttura di dati che è quindi invisibile ad ogni altra operazione che non sia definita nell'ambito dello specifico *tipo di dati astratto*.

Il concetto che sta alla base del *tipo di dati astratto* è un concetto di astrazione simile a quello delle procedure. Si tratta cioè di separare la implementazione di un certo oggetto dalla sua utilizzazione. Una procedura può essere chiamata ignorando i dettagli della sua implementazione e conoscendo solo le sue specifiche (i parametri richiesti e l'effetto della chiamata alla procedura). In tale senso una qualunque modifica della implementazione non si ripercuote sul programma chiamato purchè non varino le specifiche della procedura.

Analogamente il *tipo di dato astratto* consente di separare la specifica di una struttura di dati dalla sua implementazione. Volendo estendere tale concetto ad un ambito multiprogrammato è necessario, poichè la struttura di dati diventa accessibile a più processi, specificare la sequenza con la quale le procedure possono accedere a tale struttura. In altre parole è necessario aggiungere alla definizione di *tipo di dati astratto* una specifica della sincronizzazione fra le esecuzioni delle procedure.

Un aspetto molto importante legato alla introduzione di costrutti linguistici che permettono di associare ad una struttura di dati comune sia le procedure che su esso operano sia le regole di accesso, è quello relativo alla protezione della struttura e alla garanzia che tali accessi avvengano nell'ordine specificato. Ciò permette di rivelare in fase di compilazione tutta una serie di errori "dipendenti dal tempo" che altrimenti sarebbero difficilmente rilevabi

li ed eliminare la necessità di introdurre "meccanismi hardware" per effettuare tali controlli durante l'esecuzione dei programmi.

In questo lavoro vengono presentati due diversi metodi di associare al concetto di *tipo di dati astratto* le specifiche di sincronizzazione. Nel primo, che dà luogo al concetto di *monitor* [3],[4], tali specifiche sono contenute nelle procedure cui è riservato il compito di sospendere i processi che non possono avere accesso alla struttura di dati. Nel secondo metodo le regole di sincronizzazione sono definite mediante espressioni regolari (*path expressions*), esterne alle procedure, che definiscono la sequenza con la quale le procedure possono avere accesso ai dati comuni.

2. *Monitor*

2.1 - *Definizione e proprietà*

Il *monitor* è un costrutto sintattico che associa un insieme di procedure con una struttura di dati comune a più processi consentendo al compilatore di verificare che esse sono le sole operazioni permesse su quella struttura e assicurando la loro mutua esclusione. Introdotto per facilitare la programmazione strutturata di problemi in cui è necessario controllare l'assegnazione di una o più risorse tra più processi concorrenti secondo predeterminati algoritmi di gestione, il *monitor* costituisce una formalizzazione del concetto di *segretaria*.

Adottando una notazione simile a quella del Pascal si ha:

```
type nome del monitor = monitor
    dichiarazione dei dati locali al monitor;
    procedure entry P1 (parametri formali)
        begin . . . . . end;
    . . . . . .
    procedure entry Pn (parametri formali)
        begin . . . . . end;
    dichiarazione di procedure locali al monitor;
    begin
    inizializzazione dei dati locali al monitor;
    end
end
```

I dati locali sono le variabili comuni che descrivono lo stato della risorsa controllata dal *monitor*; le procedure P_i, le sole che possono agire in modo esclusivo sui dati locali, provvedono a controllare, secondo un determinato algoritmo, l'ordine con cui i processi hanno accesso alla risorsa. I processi possono accedere alla risorsa solamente chiamando una procedura del *monitor*; come meglio si vedrà nel seguito, sulla base dello stato della risorsa, la procedura chiamata decide se servire immediatamente il processo o sospenderlo in attesa del verificarsi di determinate condizioni.

Le procedure locali al *monitor* possono essere chiamate solamente dalle procedure P_i; infine le procedure P_i e le procedure locali possono accedere solamente a variabili dichiarate nel *monitor*. La chiamata di una procedura P_i da parte di un processo viene indicata mediante la notazione:

nome del monitor. P_i(... parametri effettivi...).

Possono esistere più attivazioni dello stesso *monitor*, per controllare ad esempio l'accesso a più risorse dello stesso tipo. Ciascuna di tali attivazioni viene dichiarata con la seguente notazione:

var *nome dell'attivazione: nome del monitor*.

L'attivazione di un *monitor* corrisponde all'allocazione in memoria dei dati locali e alla esecuzione delle procedure per la loro inizializzazione.

Come già si è detto lo scopo del *monitor* è quello di controllare l'assegnazione di una risorsa tra processi concorrenti in accordo a determinate politiche di gestione. Questa assegnazione avviene secondo due livelli. Il primo garantisce che un solo processo alla volta abbia accesso al *monitor* e quindi che le procedure del *monitor* siano eseguite rigorosamente una alla volta; diversamente non sarebbe prevedibile il loro effetto sulle variabili comuni. I processi che richiedono l'uso del *monitor* mentre una procedura è in esecuzione devono essere ritardati. Il secondo livello controlla l'ordine secondo cui i processi hanno accesso alla risorsa; quando un processo ha avuto accesso al *monitor* la sua richiesta di uso della risorsa può non essere soddisfatta dipendendo dallo stato della risorsa. In questo caso esso viene sospeso dalla procedura che sta eseguendo in una coda locale al *monitor*; non appena sospeso il processo perde l'accesso esclusivo alle variabili comuni in modo da consentire ad un altro processo di accedere al *monitor*, modificare tali variabili e creare così le condizioni per la sua riattivazione.

E' evidentemente necessario garantire ai processi sospesi, una volta riat-

tivati, la sicurezza di poter completare la procedura interrotta in un tempo finito. Ciò è possibile se l'esecuzione del processo sospeso viene ripresa immediatamente dopo la sua riattivazione e prima comunque che il processo che lo ha riattivato abbandoni il *monitor*. In questo modo infatti viene esclusa l'eventualità che un nuovo processo entri nel *monitor* e si impadronisca della risorsa prima del processo riattivato. Ci possono essere più condizioni per cui un processo può essere ritardato all'interno del *monitor*; per tenere conto di ciò Hoare [3] introduce un nuovo tipo di variabile, detto *condizione*. All'interno del *monitor* vengono dichiarate tante variabili di tipo *condizione* quante sono le condizioni per cui un processo può essere ritardato; ciascuna di queste variabili rappresenta una coda nella quale risiedono i processi sospesi.

Le procedure del *monitor* agiscono su tali variabili tramite le operazioni *cond. wait* e *cond. signal*. L'esecuzione della operazione *cond. wait* sospende il processo e lo introduce nella coda individuata dalla variabile *cond*; l'esecuzione dell'operazione *cond. signal* rende attivo un processo in attesa nella coda individuata dalla variabile *cond* ; tale processo può ora riprendere l'esecuzione della procedura dal punto in cui era stato interrotto.

Si noti che le variabili condizione e le operazioni *cond. wait* e *cond. signal* sono analoghe alle variabili *evento* e alle operazioni *await(e)* e *cause(e)* come introdotte da Hansen [4]. In entrambi i casi si ottengono code a molti processi gestite *first in-first out*.

Una soluzione diversa che dà alle procedure del *monitor* completo controllo sull'ordine con cui i processi utilizzano le risorse è rappresentata dalle code ad un solo processo [10]; esistono cioè nel monitor tante code quanti sono i processi che possono sospendersi, ciascuna delle quali può contenere un solo processo. Quando una procedura vuole ritardare il processo esegue l'operazione *delay* sulla variabile coda relativa; l'esecuzione dell'operazione *continue* sulla stessa variabile ha l'effetto di riattivare il processo sospeso.

2.2 - *Esempio : Gestione di un buffer circolare*

Il problema è noto ed è stato più volte trattato nella letteratura [3], [5]. Un insieme di processi, divisi tra produttori e consumatori, si scambiano messaggi tramite un buffer; in particolare i produttori generano messaggi e li depositano sul buffer e i consumatori li prelevano e li utilizzano. La comunicazione è soggetta a due vincoli:

a) un produttore non può inserire un messaggio nel buffer se questo è pieno;

b) il consumatore non può prelevare un messaggio dal buffer se questo è vuoto.

Si supponga il buffer suddiviso in N porzioni identiche, ciascuna delle quali contiene un messaggio, e gestito come una lista circolare. Siano *lunghezza*, *testa*, *coda* tre variabili intere comuni a tutti i processi che rappresentano rispettivamente il numero di messaggi contenuti nel buffer e la posizione del primo e ultimo dei suoi elementi contenenti i messaggi.

Siano inoltre:

send (x:messaggio) e
receive (var x:messaggio)

le due procedure utilizzate dai processi per accedere al buffer. Le due condizioni per cui un processo può attendere possono essere rappresentate tramite le variabili:

non vuoto : condizione
non pieno : condizione

Il *monitor* risulta quindi:

```
type buffer circolare = monitor
  var buffer: array 0...N-1 of messaggi;
      testa, coda: 0...N-1 ;
      lunghezza  : 0...N   ;
      non pieno, non vuoto : condizione;
  procedure entry send (x:messaggio);
    begin
      if lunghezza = N then non pieno.wait;
      buffer(coda): = x;
      coda: = (coda+1) mod N;
      lunghezza: = lunghezza+1;
      non vuoto.signal;
    end
  procedure entry receive (var x:messaggio);
    begin
      if lunghezza = 0 then non vuoto.wait;
      x:= buffer(testa);
      testa: = (testa+1) mod N;
      lunghezza: = lunghezza - 1
      non pieno.signal
    end
```

```
        begin "inizializzazione"
              lunghezza := 0; testa := 0; coda := 0
        end
end
```

Si noti che il tipo messaggio si presume già definito.

La soluzione proposta esclude la possibilità che più processi siano presenti contemporaneamente sul buffer; infatti un solo processo alla volta può avere accesso alle procedure del monitor. D'altra parte per il corretto funzionamento del sistema è sufficiente garantire che due processi non accedano contemporaneamente alla stessa posizione del buffer; la soluzione proposta è quindi di più restrittiva del necessario. Essa trova la sua giustificazione nell'ipotesi che il tempo impiegato a produrre o consumare una risorsa risulti molto maggiore rispetto al tempo necessario per appendere o rimuovere un messaggio dal buffer. L'impossibilità che più processi contemporaneamente possano accedere alla stessa struttura di dati, propria del costrutto di *monitor*, può quindi limitare in taluni casi, il grado di parallelismo nel sistema. Per ridurre l'attesa da parte di processi che logicamente potrebbero continuare è necessario, come criterio generale da seguire nel progettare i *monitor* che il tempo speso nell'esecuzione delle sue procedure sia limitato.

2.3 - *Implementazione del monitor*

Il tipo di implementazione riportato è quello introdotto da Hoare in[3] . La condizione di mutua esclusione tra le procedure del *monitor* può essere semplicemente ottenuta associando ad ogni *monitor* un semaforo *mutex* inizializzato a 1; la richiesta da parte di un processo di utilizzare una procedura equivale all'esecuzione di una *wait (mutex)* mentre in uscita dalla procedura viene eseguita una *signal (mutex)*. Ciò assicura che le singole procedure siano eseguite una alla volta.

L'esigenza di garantire ai processi sospesi, non appena riattivati, l'immediata ripresa della procedura interrotta comporta la sospensione del processo che ha eseguito la *cond. signal*. Viene quindi introdotto per ciascun *monitor* un secondo semaforo *urgent* (inizializzato a 0), sul quale i processi che hanno eseguito la *cond. signal* si sospendano tramite una *wait (urgent)*. Prima di abbandonare il *monitor* è quindi necessario verificare che nessun processo sia in coda a tale semaforo. Indicando con *urgentcount* un contatore (inizializzato a zero) del numero di processi sospesi sul semaforo *urgent*, l'uscita da u

na procedura del *monitor* viene così codificata:

if urgentcount > 0 then signal (urgent) else signal (mutex).

Indicando per ciascuna variabile *condizione:*

condsem un semaforo, inizialmente zero, sul quale un processo si sospende tramite una *wait (condsem);*

condcount un contatore, inizialmente zero, per tenere conto del numero dei processi sospesi,

l'operazione *cond.wait* può essere realizzata nel seguente modo:

```
condcount : = condcount + 1 ;
if urgentcount > 0 then signal(urgent) else signal(mutex);
wait (condsem);
condcount : = condcount - 1
```

l'operazione *cond.signal:*

```
urgentcount : = urgentcount + 1;
if condcount > 0 then begin signal(condsem); wait(urgent) end;
urgentcount : = urgentcount - 1.
```

Una soluzione più semplice si ha se si impone che l'operazione *cond.signal* sia eseguita sempre come ultima operazione di ogni procedura; in questo caso non è necessario introdurre il semaforo *urgent* e quindi il contatore *urgentcount*. Si ha pertanto per la *cond.wait:*

```
condcount : = condcount + 1
signal(mutex);
wait(condsem);
condcount : = condcount - 1
```

e per la *cond.signal:*

```
if condcount > 0 then signal(condsem)
                 else signal(mutex)
```

Quest'ultima soluzione che ha il pregio di ridurre l'"overhead" del sistema imposto dall'uso del *monitor*, non sembra alla luce delle attuali esperienze eccessivamente restrittiva. Risulta infatti abbastanza naturale, ove necessario, concludere ogni procedura del *monitor* con l'operazione *cond.signal*. Entrambe le soluzioni tuttavia soddisfano la seguente condizione: il *monitor* non può essere liberato e reso disponibile ai processi esterni fino a quando sono presenti all'interno processi in grado di completare l'esecuzione di una procedura.

Analoga risulta la implementazione delle operazioni *delay* e *continue*.

Allo scopo di assicurare un'efficiente implementazione delle operazioni *cond. wait* e *cond. signal* e garantire la loro validità in un vasto campo di applicazione, l'algoritmo di gestione della coda associata alla variabile *cond* è normalmente quello *first-in, first-out*. Qualora si voglia avere un maggiore controllo sulla strategia di gestione si può ad esempio precisare nell'operazione *cond. wait* un'indicazione della priorità *p* del processo. Si ha cioè:

cond. wait (p) .

Il processo che esegue tale operazione viene sospeso e introdotto nella coda della variabile *cond* con l'indicazione della sua priorità. Dopo l'esecuzione dell'operazione *cond. signal* viene immediatamente ripresa la esecuzione del processo col valore più elevato di *p*. Ovviamente per impedire che un processo con bassa priorità attenda indefinitamente occorre che le priorità siano modificate dinamicamente in funzione del tempo di attesa dei processi.

L'algoritmo di gestione comunemente realizzato per dare ai processi l'accesso alle procedure del *monitor* è di tipo *first-in, first-out*, in conformità all'ipotesi fatta che il tempo speso dai processi nell'esecuzione delle procedure e quindi il tempo di attesa all'esterno del *monitor*, sia limitato.

2.4 - *Impiego del monitor nella strutturazione di sistemi operativi*

Una delle proprietà fondamentali del *monitor* è la possibilità che le sue procedure hanno di chiamare procedure definite entro altri *monitors*.

Questa proprietà rende possibile definire una gerarchia nella gestione delle risorse di un sistema e fa del *monitor* un concetto utilmente impiegabile nella progettazione di un sistema operativo gerarchicamente strutturato.

Recentemente Hansen [10], [11] ha definito un linguaggio, *Concurrent Pascal*, per la programmazione strutturata di sistemi operativi, che estende il linguaggio Pascal, con la introduzione del costrutto *monitor* (oltre ai costrutti *classe* e *processo*). Il *monitor* come definito da Hansen differisce da quello definito precedentemente proprio per la proprietà di fare comparire esplicitamente i diritti di accesso per altri *monitors*. Si consideri ad esempio il problema del buffer circolare trattato precedentemente e si supponga che il buffer sia contenuto su disco. Il problema può essere affrontato definendo due *monitors*, uno per ciascuna risorsa interessata (buffer e disco); il primo, *buffer su disco*, controlla l'accesso al buffer tramite le procedure *send* e *receive*, già defini-

te ed ha accesso ad un secondo *monitor*, *disco*, che controlla la risorsa disco tramite le procedure *read* e *write* . Si ha cioè:

```
type buffer su disco = monitor (disco:monitor, base: integer)
    begin buffer:array 0...N-1 of messaggi;
          testa, coda: 0...N-1;
          lunghezza  : 0...N  ;
          non pieno, non vuoto: condizione;
    procedure entry send (x:messaggio);
        begin
           if lunghezza = N then non pieno.wait;
              disco.write(base + coda, x)
              coda: = (coda+1)mod N
              lunghezza: = lunghezza+1;
              non vuoto.signal
        end
    procedure entry receive (var x:messaggio)
        begin
        end        "analoga alla procedura send"
    begin "inizializzazione"
        lunghezza: = 0; testa: = 0; coda: = 0
    end
end
```

Si noti che nella dichiarazione del *monitor* compaiono, tra parentesi, due variabili che non sono locali al monitor. La prima, *disco* , è di tipo monitor e la seconda, *base*, di tipo integer rappresenta l'indirizzo iniziale del buffer sul disco. Il secondo *monitor* ha questa forma:

```
type disco = monitor
   procedure entry write (indirizzo su disco: integer, x:messaggio)
   begin
   end        "scrittura su disco"
   procedure entry read (indirizzo su disco: integer, var x:messaggio)
   begin
   end        "lettura su disco"
end
```

Poichè l'esecuzione di una procedura del *monitor* richiede l'esecuzione di

ulteriori chiamate al *monitor* stesso, è necessario impedire, per evitare possibili situazioni di "deadlock", che un *monitor* chiami se stesso in modo ricorsivo. Il compilatore del *Concurrent Pascal* oltre ad assicurare che le variabili comuni dichiarate entro il *monitor* siano accessibili solo dalle procedure del *monitor* e che le procedure accedano solamente a variabili dichiarate entro il *monitor* , ha la possibilità di verificare che i diritti di accesso siano ordinati in maniera gerarchica. A tale scopo sono imposte le seguenti regole:

a) ogni procedura deve essere dichiarata prima di essere chiamata;
b) una procedura di un *monitor* non può chiamare una procedura di un altro *monitor* che noncompaia tra i suoi diritti di accesso;
c) una procedura di un *monitor* non può chiamare se stessa nè un'altra procedura dello stesso *monitor* (a meno che quest'ultima sia una procedura locale cioè non accessibile dell'esterno).

3. *Path Expression*

3.1 - *Definizione e proprietà*

Illustriamo ora un secondo metodo che permette di associare ad una struttura di dati comune, sia le procedure che su essa possano essere eseguite, sia le regole di sincronizzazione tra le chiamate a tali procedure da parte di più processi concorrenti. Questo metodo consiste nell'associare alla definizione di *tipo di dato astratto*, relativo alla struttura di dati comune, la specifica delle regole di sincronizzazione. Tale specifica è esterna alle procedure stesse ed è descritta mediante un insieme di espressioni regolari, dette *path expressions*.

L'insieme di un tipo di dato astratto e delle *path expressions* relative alle sue procedure , permette di descrivere quali procedure possono essere chiamate per accedere alla struttura di dati e come queste procedure devono essere sincronizzate per permettere alla struttura di dati di essere suddivisa tra più processi.

Una *path expression* è costituita da un elenco di nomi di procedure e da un insieme di regole che specificano l'ordine con cui tali procedure devono essere eseguite. Un processo che chiama una di tali procedure potrà quindi proseguire o no la sua esecuzione a seconda delle regole specificate dalla *path expression*. Ogni *path expression* viene implementata da un meccanismo di controllo che opera

sul "prologo" e sull'"epilogo" d'ogni procedura. "Prologo" ed "epilogo" sono generati automaticamente in fase di compilazione. Un processo eseguente il "prologo" d'una procedura richiede al meccanismo di controllo l'autorizzazione ad operare sulla struttura di dati; durante l'esecuzione dell'"epilogo" gli notifica la fine della esecuzione della procedura chiamata.

Schemi di sincronizzazione tipici definiti dalle *path expressions* riguardano, per esempio:

a) la necessità di sequenzializzare l'esecuzione delle procedure (*sequence*);
b) una qualunque scelta alternativa tra diverse procedure da attivare (*selection*);
c) la ripetizione ciclica delle procedure (*repetition*);
d) l'esecuzione simultanea di procedure (*simultaneous execution*).

La notazione adottata per esprimere le regole di sincronizzazione tramite *path expressions* è stata proposta da R.H. Campbell e A.N. Habermann [8]. Secondo questa notazione un'espressione i cui elementi sono separati da punto e virgola rappresenta lo schema di sequenzializzazione. Per esempio:

a; b; c

specifica che le procedure a, b e c devono essere eseguite nell'ordine indicato. Solo quando cioè un processo ha eseguito la procedura a, lo stesso processo o un altro può eseguire la procedura b.

La selezione fra un insieme di procedure viene rappresentata mediante una espressione i cui elementi sono separati da virgole. Per esempio:

d, e, f

specifica la mutua esclusione tra le procedure d, e, f.

Racchiudere una espressione tra le parole chiavi path, end implica la ripetizione ciclica dell'attivazione delle procedure nominate nella espressione, secondo le regole precedentemente elencate. Per esempio,

path a; b; c end

specifica che le procedure a, b e c possono essere eseguite ciclicamente nell'ordine indicato.

Racchiudere un'espressione tra parentesi graffe significa permettere l'accesso concorrente alle procedure specificate. Per esempio la *path expression*

{a}

permette a vari processi di eseguire simultaneamente la procedura a.

Una volta che un processo ha iniziato l'esecuzione di a, altri processi possono cioè eseguire a senza nessun ritardo. Appena l'ultimo di questi ha terminato, la *path expression* si considera completata.

Questi schemi di sincronizzazione possono essere combinati per formare *path expression* più complete.

Per esempio la *path expression* :

$$\underline{\text{path}}\ P_1,\ ((P_2,\ P_3);\ \{P_4\}\ ;\ P_5)\ \underline{\text{end}}$$

denota che la procedura P_1 può essere eseguita in alternativa alla espressione racchiusa tra le parentesi tonde esterne. Cioè, in alternativa a P_1 può essere eseguita P_2 o P_3, quindi P_4 in modo simultaneo e finita l'ultima esecuzione di P_4, può avvenire l'esecuzione di P_5. Una volta eseguita una qualunque delle due alternative la *path expression*, viene ripetuta ciclicamente; cioè avviene una nuova scelta tra P_1 e P_2 o P_3.

Con lo scopo di semplificare la realizzazione del meccanismo di controllo vengono introdotti alcuni vincoli sulla scrittura delle *path expression* e precisamente :

a) il nome di una procedura può comparire una sola volta in tutte le *path expression*. E' però permesso ridefinire la stessa procedura con un altro nome.
b) una *path expression* che denota una ripetizione ciclica non può essere inserita all'interno di altre *path expressions*.
c) Coppie di parentesi graffe non possono essere nidificate.

Come si vedrà nel seguito tali vincoli non introducono grosse restrizioni reali.

3.2 - *Esempi di specifiche tramite path expressions*

Vediamo adesso due esempi che mettono in evidenza la facilità con cui le "*path expressions*" permettono di specificare la sincronizzazione tra processi.

a) $$\underline{\text{path}}\ \{\textit{read}\}\ ,\ \textit{write}\ \underline{\text{end}}$$

specifica l'esecuzione ciclica delle procedure *read* e *write* in modo alternativo. Mentre le esecuzioni della procedura *read* possono essere concorrenti, l'esecuzione della *write* è mutuamente esclusiva. Cioè, mentre vari processi che richiedono la lettura possono procedere simultaneamente, solo un processo

alla volta può scrivere.

b) path {*read*} , {*WRITE*} end
path *write* end

dove *WRITE* rappresenta una procedura così definita:

procedure entry *WRITE* begin *write* end

Rispetto all'esempio precedente si introduce una modifica nella strategia di gestione nel senso che se un processo esegue la procedura *WRITE*, altri processi pronti ad eseguire la stessa procedura possono continuare purchè l'operazione di *write* sia mutuamente esclusiva. Si noti che sono necessarie due *path expressions* per specificare due vincoli di sincronizzazione distinti: uno fra le operazioni di lettura e scrittura e l'altro per garantire che le operazioni di scrittura avvengano una alla volta.

Considerando che ogni procedura non può comparire in più di una *path expression* è necessario ridefinire la procedura *write*. Infatti questa restrizione permette al compilatore di generare un meccanismo di controllo indipendente per ogni *path expression*.

3.3 - *Gestione di un buffer circolare*

Consideriamo ora come esempio di applicazione delle *path expressions* lo stesso problema trattato nel paragrafo 2.2.

Introduciamo il *tipo di dato astratto buffer circolare* specificato a sua volta tramite tipi più elementari: *elemento* e *puntatore*. Si noti che il tipo *messaggio* si presume già definito.

```
type elemento =
    var x:messaggio
    path write; read end
    procedure entry write (m:messaggio);
        begin
        end     x := m
    procedure entry read (var m:messaggio)
        begin
        end     m := x
end
```

```
type puntatore =
    var P:integer
    path incrementa end;
    procedure entry incrementa (var I:integer);
        begin
                P : = (P+1) mod N;
                I : = P
        end
    begin "inizializzazione"
                P : = 0
        end
end
```

Nelle due definizioni di tipo *elemento* e *puntatore* compaiono due *path expressions*: nella prima si esprime la necessità di garantire, su un determinato elemento del buffer, che ad una operazione di scrittura faccia seguito una operazione di lettura; la seconda specifica che solo un processo alla volta può incrementare una variabile di tipo puntatore.

Avendo già definito i tipi *elemento* e *puntatore* introduciamo il tipo *buffer circolare*. Su una variabile di tipo *buffer circolare* agiscono concorrentemente più processi che possono scambiarsi messaggi.

```
type buffer circolare =
    var buffer: array 0...N-1 of elemento;
    L,S : puntatore;
    procedure entry send (m:messaggio)
        begin J:integer;
                J: = S. incrementa;
                buffer[J]. write: = m
        end
    procedura entry receive (var m:messaggio)
        begin
                J:integer
                J: = L. incrementa;
                m: = buffer[J]. read;
        end
end
```

Si noti che non esistendo nessun vincolo esplicito di sincronizzazione tra le procedure *send* e *receive*, più processi possono contemporaneamente inviare e ricevere messaggi purchè agiscano su elementi distinti del buffer. Infatti, mentre in questa definizione di *tipo di dato astratto* non esiste nessuna *path expression*, la specifica di mutua esclusione su ogni elemento del buffer viene indicata esplicitamente dalla path *incrementa* end presente nella definizione di tipo *contatore* e la condizione di sincronizzazione tra produttori e consumatori è indicata tramite la path *write; read* end. Si ottiene così il massimo grado di parallelismo consentito dalla natura del problema.

3.4 - *Implementazione*

Campbell ed Habermann, nel loro lavoro [8] , presentano un algoritmo ricorsivo mediante il quale un compilatore è in grado di generare, automaticamente, i meccanismi di controllo per ogni *path expression* rappresentata nella notazione proposta.

Tale metodo consiste nel generare un "prologo" ed un "epilogo" per ogni procedura il cui nome compare in una *path expression*. Il "prologo" e "l'epilogo" consistono in appropriate sequenze di operazioni *wait* e *signal* su variabili di tipo semaforo e la loro esecuzione permette di coordinare le azioni sulla struttura di dati, controllata dalla *path expression*, secondo la strategia di sincronizzazione ivi specificata.

Prima di procedere alla descrizione dell'algoritmo è conveniente introdurre due operazioni, *ww* e *ss* , che vengono utilizzate dall'algoritmo stesso nella generazione dei meccanismi di controllo. Tali operazioni hanno tre parametri, un contatore e due semafori, e sono definite in termini delle operazioni *wait* e *signal*.

```
procedure ww (var c: contatore; var s_m, s_i: semaforo);
    begin
        wait (s_m);
        c: = c+1 ;
        if c = 1 then wait(s_i) ;
        signal (s_m)
    end
procedure ss (var c: contatore; var s_m, s_i: semaforo);
    begin
        wait (s_m);
        c: = c-1 ;
        if c = 0 then signal (s_i);
        signal (s_m)
    end
```

I tipi contatore e semaforo si considerano già definiti. Le procedure *ww* e *ss* servono per generare il "prologo" e "l'epilogo" di procedure di cui è ammessa l'esecuzione simultanea da parte di più processi; il semaforo s_m è usato per operare in modo esclusivo sul contatore C.

Nella descrizione dell'algoritmo con il simbolo <espressione> verrà indicata la *path expression* su cui si opera. Durante l'esecuzione dell'algoritmo la <espressione>, viene generalmente racchiusa tra due operazioni di sincronizzazione, indicate genericamente con O_L e O_R. L'operazione O_L, che precede la <espressione>, può essere o una *wait* oppure un'operazione *ww*. Viceversa la operazione O_R, che segue la espressione può essere o una *signal* o una operazione *ss*.

L'algoritmo consiste di due passi consecutivi:

Passo 1 : si sceglie un semaforo s_1 inizializzato ad 1 e si modifica l'espressione:

path <espressione> end in
wait (s_1) <espressione> *signal* (s_1)

quindi si esegue il passo 2 operando su <espressione>.

fine passo 1

Passo 2: si esamina la <espressione> e a seconda dello schema di sincronizzazione ivi specificato si esegue uno dei seguenti rami:

a) *sequence*. In tal caso <espressione> è costituita da

<espressione 1>; <espressione 2>

Si sceglie un semaforo s_2 inizializzato a zero e si modifica <espressione> in:

<espressione 1> *signal* (s_2) *wait* (s_2) <espressione 2>

quindi si esegue il passo 2 di nuovo sia per <espressione 1> che per <espressione 2>.

fine passo 2

b) *selection*. In tal caso <espressione> è costituita da

<espressione 1>, <espressione 2>

Indicando con O_L e O_R le operazioni di sincronizzazione che racchiudono <espressione>, cioè:

O_L <espressione 1>, <espressione 2> O_R

si effettua la seguente modifica:

$$O_L < \text{espressione 1} > O_R \; O_L < \text{espressione 2} > O_R$$

quindi si esegue il passo 2 di nuovo sia per <espressione 1> che per <espressione 2>

fine passo 2

c) *Simultaneous execution*. In tal caso <espressione> è costituita da:

$$\left| < \text{espressione 1} > \right|$$

Le operazioni O_L e O_R racchiudenti espressione saranno rispettivamente una operazione *wait* ed un'operazione *signal* eseguite rispettivamente su due semafori s_i e s_j:

$$wait\ (s_i) \left| < \text{espressione 1} > \right| signal\ (s_j)\ .$$

Ciò è dovuto al fatto che, come stabilito in precedenza, le parentesi graffe non possono essere nidificate in una *path expression*. Si effettua la seguente modifica su O_L e O_R e sulle parentesi graffe:

$$ww\ (c, s_m, s_i) < \text{espressione} > ss\ (c, s_m, s_j)$$

quindi si esegue il passo 2 di nuovo per <espressione 1>.

fine passo 2

d) *Nome di una procedura* . In tale caso <espressione> è costituito esclusivamente dal nome di una delle procedure che devono essere sincronizzate.

L'operazione O_L presente alla sinistra di <espressione> (che può essere o una *wait* o una operazione *ww*) viene inclusa nel "prologo" della procedura.

L'operazione O_R presente a destra di <espressione> (che può essere o una *signal* o un'operazione *ss*) viene inserita nell'"epilogo" della procedura.

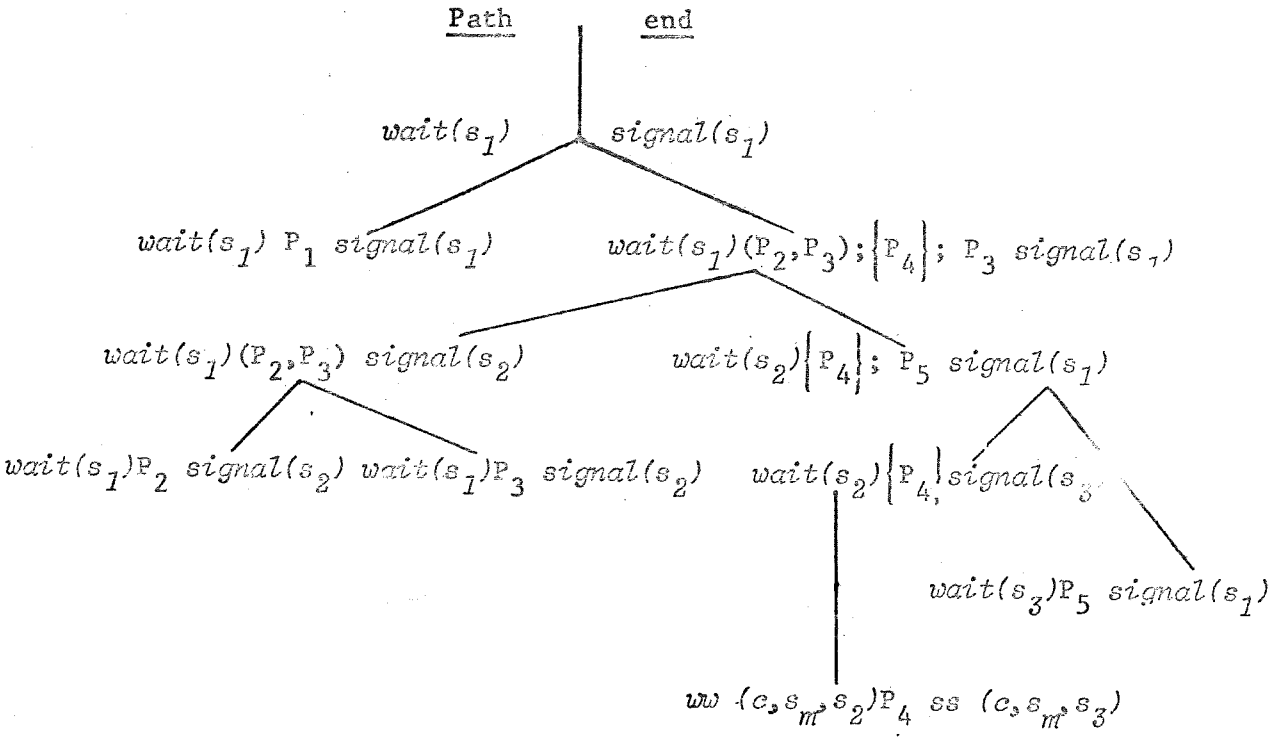
fine passo 2.

Schematicamente è possibile rappresentare l'evoluzione del precedente algoritmo mediante un albero. Ogni passo dell'algoritmo corrisponde ad un nodo dell'albero che viene racchiuso tra le operazioni di sincronizzazione O_L e O_R che sono state generate fino al corrispondente passo dell'algoritmo.

Illustriamo, per esempio, come viene tradotta la *path expression* :

path P_1, $((P_2,P_3); \{P_4\}; P_5)$ end

già vista nel paragrafo 3.1:



Il compilatore modifica quindi le procedure P_1,P_2,P_3,P_4,P_5 nel modo seguente:

var s_1,s_m: semaforo
var s_2,s_3: semaforo
var c : contatore

Procedures:

P_1 : begin *wait*(s_1) ; <corpo di P_1> ; *signal*(s_1) ; end
P_2 : begin *wait*(s_1) ; <corpo di P_2> ; *signal*(s_2) ; end
P_3 : begin *wait*(s_1) ; <corpo di P_3> ; *signal*(s_2) end
P_4 : begin *ww* (c,s_m,s_2) ; <corpo di P_4> ; *ss* (c,s_m,s_3) end
P_5 : begin *wait*(s_3) ; <corpo di P_5> ; *signal*(s_1) ; end
end
begin "inizializzazione"
$s_1 := 1$; $s_m := 1$; $s_2 := 0$; $s_3 := 0$; $c := 0$
end

4. *Conclusioni*

Sia i tipi di dati astratti combinati con le *path expressions*, sia i *monitors*, forniscono meccanismi linguistici atti a controllare la concorrenza nei sistemi multiprogrammati. Esistono però alcune differenze tra i due approcci.

Una prima differenza consiste nel fatto che le operazioni d'un *monitor* non possono essere eseguite concorrentemente; inoltre la sospensione di un processo all'interno d'un *monitor*, le cui procedure sono chiamate da un altro *monitor*, ritarda l'esecuzione delle procedure di questo ultimo da parte di altri processi.

Nel caso delle *path expressions*, invece, è possibile specificare l'esecuzione contemporanea di una procedura da parte di più processi. Inoltre nel caso di tipi di dati astratti implementati in termini di altri tipi, se l'esecuzione di una procedura di un tipo A viene ritardata nell'accesso ad un tipo più interno B, non sono necessariamente ritardate le esecuzioni delle altre procedure di A purchè operino su oggetti diversi anche se dello stesso tipo di B.

Questi aspetti sono messi in rilievo dalle implementazioni proposte per operare su un buffer circolare (par.2.2; 3.3). Infatti, mentre nella prima implementazione ogni operazione di accesso al buffer circolare costituisce un'unica sezione critica, nella seconda soltanto le operazioni sui singoli elementi del buffer sono mutuamente esclusivi. Si nota, però che il tempo di esecuzione di ogni operazione nel caso dei *monitors* è, in generale, limitato e dunque questo aspetto incide relativamente sulla efficienza del sistema.

Una seconda osservazione corrisponde al fatto che mentre nel caso dei *monitors* la strategia di gestione è implicita nelle sue operazioni, le *path expressions* esprimono in forma esplicita e concisa le condizioni di sincronizzazione. La sincronizzazione risulta così una proprietà globale di un tipo di dati astratto; le operazioni associate a quel tipo possono quindi essere definite indipendentemente l'una dall'altra. Modifiche nella strategia di sincronizzazione comportano in questo caso solo un cambio di *path expressions*. Inoltre la specifica della sincronizzazione è indipendente da una particolare implementazione. Questa generalità però introduce un "overhead" superiore a quello introdotto tramite *monitors*.

I due meccanismi linguistici presentati sono adatti alla formulazione di relazioni invarianti che facilitano una possibile verifica formale di correttezza [3], [7], [12]. L'introduzione di questi costrutti linguistici permette di effet-

tuare a tempo di compilazione il controllo degli accessi ad un oggetto, estendendo al campo della multiprogrammazione i risultati ottenuti mediante l'introduzione del concetto di tipo di dato astratto nella programmazione sequenziale.

Allo stato attuale non esiste nessun linguaggio di programmazione che implementi le *path expressions*; i *monitors* sono stati invece introdotti nel linguaggio *Pascal* come l'unico metodo per controllare la concorrenza dando origine al *Concurrent Pascal*.

Bibliografia

[1] P.Brinch Hansen - *Concurrent Programming Concepts* - A.C.M. Computing Surveys (December 1973) pp. 232-245.

[2] P.Ancilotti, M.Boari, N.Lijtmaer - *Tecniche di Programmazione Strutturata Estese ad un Ambito di Processi Concorrenti* - Rivista di Informatica, vol.4, 3-4 pp.335-356.

[3] C.A.R. Hoare - *Monitors: An Operating System Structuring Concept* - Communications A.C.M. 17,10 (oct.1974), pp.549-557.

[4] P.Brinch Hansen - *Operating System Principles* - Prentice-Hall 1973.

[5] E.W.Dijkstra - *Hierarchical Ordering of Sequential Processes* - Operating System Techniques - Academic Press 1972, pp.90-93.

[6] L.Flon - *Program Design With Abstract Data Types* - Department of Computer Science, Carnegie-Mellon University, Pittsburg, Pa. (June 1975).

[7] L.Flon, A.N.Habermann - *Towards the Construction of Verifiable Software Systems* - Sigplan Notices, pp.141-148 - Proceedings of Conference on Data Abstraction, Definition and Structure (March 1976) Salt Lake City, UTAH.

[8] R.H.Campbell, A.N.Habermann - *The Specification of Process Synchronization by Path Expressions* - Lecture Notes in Computer Science vol. 16, Spinger-Verley (1974).

[9] C.A.R.Hoare - *Towards a Theory of Parallel Programming* - Operating System Techniques - Academic Press 1972 - pp.61-71.

[10] P.Brinch Hansen - *The Programming Language Concurrent Pascal* - International Summer School On Language Hierarchies and Interfaces - Munich (July 1975).

[11] P.Brinch Hansen - *A Programming Methodology for Operating System Design* - Proc. IFIP 74 Congress (Aug.1974) pp.394-397.

12 J.H.Howard - *Proving Monitors* - Communications A.C.M. (May 1976) pp.273-278.

ERRATA CORRIGE

pag.	rigo	ERRATA	CORRIGE
I	IO	la maggior parte di	la maggior parte degli
2	5	potendosi	potendo
2	32	che su esso operano	che su essa operano
3	9	la sequenza con la quale	le sequenze con le quali
4	I3	*nome del monitor.* P_i	*nome dell'attivazione del monitor.*
6	6	del primo ed ultimo dei suoi elementi contenenti messaggi	del primo dei suoi elementi conten i messaggi e quello del primo elem vuoto
IO	I	*monitor, disco*	*monitor, periferica* di tipo *disco*
IO	3	(*disco*:monitor,	(*periferica*:*disco*,
IO	II	*disco.write*	*periferica.write*
IO	25	*disco* è di tipo monitor	*periferica* di tipo *disco* è un *moni*
IO	27	il secondo *monitor*	il *monitor disco*
I9	5	; $\{P_4\}$; P_3 *signal*(s_1)	; $\{P_4\}$; P_5 *signal*(s_1)
I8	I5	espressione	espressioneI

```
type puntatore
    var P:integer
    path incrementa end;
    procedure entry incrementa (var I:integer);
        begin
                P := (P+1) mod N;
                I := P ;
        end
    begin "inizializzazione"
                P := 0;
    end
end
```

Nelle due definizioni di tipo *elemento* e *puntatore* compaiono due path expressions: nella prima si esprime la necessità di garantire, su un determinato elemento del buffer, che ad una operazione di scrittura faccia seguito una operazione di lettura; la seconda specifica che solo un processo alla volta può incrementare una variabile di tipo puntatore.

Avendo già definito i tipi *elemento* e *puntatore* introduciamo il tipo *buffer circolare*. Su una variabile di tipo *buffer circolare* agiscono concorrentemente più processi che possono scambiarsi messaggi.

```
type buffer circolare
    var buffer: array 0...N-1 of elemento;
        L,S : puntatore ;
    procedure entry send (m:messaggio)
        begin J:integer;
              S.incrementa(J);
              buffer[J].write(m);
        end
    procedure entry receive  (var m:messaggio )
        begin
              J:integer ;
              L.incrementa(J);
              buffer[J];read(m);
        end
end
```