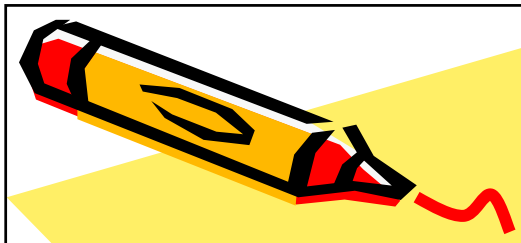
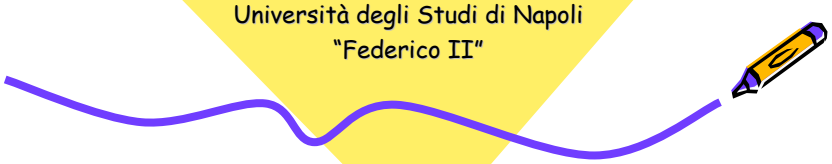


Communicating Sequential Processes (CSP)

Aniello Murano
Università degli Studi di Napoli
"Federico II"



Introduzione al CSP

- Il Communicating Sequential Proceesses (CSP) è stato inventato nel 1978 da Sir **Charles Antony Richard Hoare** (anche noto come **Tony Hoare** o **C.A.R. Hoare)**

- Hoare è nato nel Gennaio del 1934 ed è probabilmente meglio conosciuto per aver inventato nel 1960 il **Quicksort**, (**l'algoritmo di ordinamento più usato al mondo!!!**) e il linguaggio **ALGOL 60**
- Ha ricevuto nel 1980 il **Turing Award** per "i suoi contributi fondamentali alla definizione e alla progettazione di linguaggi di programmazione".

Il CSP

- Il **CSP** è un linguaggio formale per descrivere l'interazioni di processi concorrenti.
- Il CSP fa parte di quella teoria matematica della concorrenza nota con il nome di **algebra dei processi**, o process calculi.
- Il CSP è stato utilizzato in pratica come uno strumento di **specifica** e di **verifica** degli aspetti concorrenti di una notevole varietà di sistemi
- Le regole di CSP hanno ispirato il linguaggio di programmazione **occam** (il nome deriva da William of Ockham) oggi largamente utilizzato come linguaggio di programmazione parallela.
- Tra le recenti estensioni di CSP troviamo **JCSP** (CSP for Java http://www.cs.kent.ac.uk/projects/ofa/jcsp/) e **Timed CSP** che incorpora informazioni temporali per lavorare su sistemi real-time.
- Un software (free) utilizzato per controllare modelli formali espressi in CSP è **FDR2**, sviluppato dalla Formal Systems Europe Ltd. (www.fsel.com)
- La ricerca sulla teoria del CSP è tuttora attiva, così come il lavoro di incrementare il campo di applicabilità pratica.

Concorrenza/Parallelismo

- In informatica, la concorrenza concerne la condivisione di risorse tra più computazioni che vengono eseguite sovrapposte nel tempo.
- L'efficienza della concorrenza dipende dalle tecniche utilizzate per coordinare l'esecuzione delle computazioni, scambiare i dati, allocare memoria e schedulare i processi nel tempo in modo da minimizzare il tempo e massimizzare i risultati (si veda per esempio la tecnica greedy)
- I sistemi operativi sono sistemi concorrenti, progettati in modo da operare all'infinito, senza la possibilità di terminare in modo inatteso.
- Ricordiamo che un sistema concorrente è anche nondeterministico, come abbiamo visto per IMPGC

Esempi di Concorrenza/Parallelismo

- **Thread (di execution).**
  - I thread rappresentano un modo in cui un processo (programma) si può dividere in più task per essere eseguiti simultaneamente.
  - I **mulititread** possono essere eseguiti in parallelo su più computer
  - I multitread si dividono in time slicing (un solo processore) e multiprocessing (più processori).
  - I tread sono simili ai processi ma differiscono per il fatto che i pri-mi **condividono risorse** mentre i secondi sono indipendenti tra loro.
- **Processi**
  - Un processo è una esecuzione di una istanza di un programma.
  - Il sistema operativo mantiene i processi separati tra loro ed alloca ad ognuno le risorse richieste in modo che i processi non interferiscano tra loro ed evitare così fallimenti del sistema
  - Un sistema operativo multitasking passa tra processi dando l'impressione di una esecuzione simultanea.
  - Comunicazione tra processi: chiamate a procedure remote, messaggi, ect.

Primitive di comunicazione

- **Inter-Process Communication (IPC)** è un insieme di tecniche utilizzate per lo scambio di dati tra due o più tread di uno o più processi (indispensabile per un corretto funzionamento!!)
- Le tecniche di IPC si compongono di metodi per
  - La comunicazione tramite variabili (condivisione di memoria)
  - La comunicazione tramite messaggi
  - La sincronizzazione
  - Le chiamate a procedura remote
- Ci sono molte API (**Application Programming Interface)** che possono essere usate per IPC
  - Common Object Request Broker Architecture (CORBA);
  - Distributed Computing Environment (DCE);
  - Message Bus (MBUS);
  - anonymous pipes and named pipes;

Threads: shared variables

- Threads modeled by parallel evaluation in IMP

  $c_0 \parallel c_1$
- Commands in c_0 and c_1 may execute in any order

  $(X := 0 \parallel X := 1)$; **if** X=0 **then** c_0 **else** c_1
- Synchronization primitives
  - Used to control nondeterminism
  - Semaphores, monitors, etc

Calculi and Languages

- Several calculi and languages rely on message-passing:

  Communicating Sequential Processes (CSP) (Hoare, 1978)
  Occam (Jones)
  Calculus of Communicating Systems (CCS) (Milner, 1980)
  The Pi calculus (Milner, 1989 and others)
  Pict (Pierce and Turner)
  Concurrent ML (Reppy)
  Java RMI
  Erlang
  The Join calculus (Fournet, 1998)

Hoare e Milner individuarono un'azione atomica di sincronizzazione, con possibilità di scambio di valori, come la primitiva essenziale per la comunicazione (indipendente dal mezzo usato per la comunicazione)

Un semplice linguaggio:CSP

- Il linguaggio CSP è una architettura parallela caratterizzata da:
  - Scambio di messaggi tra processi
  - Possibilità per i thread di utilizzare memoria condivisa
- Lo scambio di messaggi avviene tramite **Canali**
- I canali permettono:
  - Uno scambio di valori tra processi
  - La sincronizzazione di processi
- Il numero dei canali è fisso:
  - Non è possibile creare o cancellare un canale durante l'esecuzione di un programma.
  - È possibile avere canali dedicati a singoli processi

Sintassi di CSP

c ::=	**skip** | **abort** | X := a | c_0; c_1	*standard commands*
Commands	| α!a	send value on channel (output)
	| α?X	receive value on channel (input)
	| $c_0 \parallel c_1$	parallel composition (locations disjoint)
	| c \ α	restriction: hide channel α
	| **if** g **fi**	alternative
	| **do** g **od**	iteration
g ::=	b → c	boolean guard
Guarded Commands	| b ∧ α?X → c	boolean receive guard
	| b ∧ α!a → c	boolean send guard
	| g_0 □ g_1	alternative

–Based on IMP: a ∈ AExp, b ∈ BExp, X ∈ Loc

Composizioni ben fondate

- Una composizione parallela $c_0 \parallel c_1$ è ben fondata se c_0 e c_1 non hanno una locazione comune.
- Un comando è ben fondato se tutti i suoi sottocomandi lo sono.
- Una restrizione $c\backslash\alpha$ nasconde il canale α in modo che possa essere utilizato solo per comunicazioni interne a c.

Semantica di CSP

- Come per IMPGC, utilizziamo una semantica operazionale basata su singoli passi non interrompibili.
- Anche per CSP, una configurazione è data da un comando e uno stato o solamente da uno stato.
- Si consideri allora la seguente configurazione di un comando

$$\langle \alpha ? X; c, \sigma \rangle$$

- Intuitivamente, questa rappresenta un comando che è pronto a ricevere un valore da associare a X tramite una comunicazione sincronizzata lungo il canale α.
- Questo può avvenire se comando è in parallelo con uno pronto a eseguire un'azione complementare di output sul canale α.
- La relativa semantica deve dunque esprimere questa dipendenza dall'ambiente. Questo può essere fatto sfruttando la teoria degli automi.

Labeled Transition Semantics

- Values

  $V ::= n$

- Labels

  $\lambda \in \text{Label} ::= \alpha!n \mid \alpha?n$ con $\alpha \in \textbf{Chan}$ & $n \in \textbf{Num}$

- Relations

  $\langle \text{Com}, \Sigma \rangle \rightsquigarrow \langle \text{Com}, \Sigma \rangle$

  $\langle \text{Com}, \Sigma \rangle \overset{\lambda}{\rightsquigarrow} \langle \text{Com}, \Sigma \rangle$

Dunque, la semantica ammette transizioni del tipo

$$\langle \alpha?X\,; c_0, \sigma \rangle \xrightarrow{\alpha?n} \langle c_0, \sigma[n/X] \rangle.$$

Che esprime il fatto che il comando $\alpha?X;c_0$ può ricevere un valore n e memorizzarlo nella locazione X modificando lo stato

Comunicazioni interne

- Si consideri la seguente transizione

$$\langle (\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma \rangle \rightarrow \langle c_0 \parallel c_1, \sigma[n/X] \rangle$$

- In questo caso la transizione non viene etichettata poiché si tratta di comunicazione interna, senza effetti sull'ambiente.
- Esistono anche transizione composte (dovute per esempio all'utilizzo di un canale α da più processi):

$$\langle (\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma \rangle \xrightarrow{\alpha?n} \langle c_0 \parallel (\alpha!e; c_1), \sigma[n/X] \rangle$$

- Questo cattura la possibilità che il primo componente riceve un valore dall'ambiente piuttosto che dal secondo componente
- Presentiamo adesso la semantica formale come estensione di quella di IMPGC dove ammettiamo comunicazione nelle guardie, e l'uso del comando vuoto E per uniformare le configurazioni

Alcune semplificazioni nelle regole

- Nelle regole di derivazione che daremo nelle prossime slide, utilizzeremo le seguenti semplificazioni:
- Una configurazione σ può essere anche vista come una coppia <c, σ> dove c è il comando "nullo" (equivalente dello skip).
- Utilizziamo il valore λ per indicare che una transizione può essere etichettata con $\alpha?X$ oppure $<\alpha!X$ oppure anche senza etichetta

Regole per i comandi (1)

Come per IMP

$$\langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma \qquad \frac{\langle a, \sigma \rangle \rightarrow n}{\langle X := a, \sigma \rangle \rightarrow \sigma[n/X]}$$

Se non ottengo un valore dall'ambiente su α non eseguo il comado e non cambio lo stato

$$\langle \alpha?X, \sigma \rangle \xrightarrow{\alpha?n} \sigma[n/X] \qquad \frac{\langle a, \sigma \rangle \rightarrow n}{\langle \alpha!a, \sigma \rangle \xrightarrow{\alpha!n} \sigma}$$

Devo valutare l'espressione a per eseguire il comando e mandare un numero all'ambiente

Nota che c'_0 può essere il comando nullo (contiene le 2 corr. regole diIMPGC)

gc per essere valutato ed eseguito può aver bisogno di λ

$$\frac{\langle c_0, \sigma \rangle \xrightarrow{\lambda} \langle c'_0, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \xrightarrow{\lambda} \langle c'_0; c_1, \sigma' \rangle} \qquad \frac{\langle gc, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle \mathbf{if}\ gc\ \mathbf{fi}, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}$$

λ perché c_0 o c_1 possono essere comandi di input/output

$$\frac{\langle gc, \sigma \rangle \xrightarrow{\lambda} \langle c, \sigma' \rangle}{\langle\ \mathbf{do}\ gc\ \mathbf{od}, \sigma \rangle \xrightarrow{\lambda} \langle c;\ \mathbf{do}\ gc\ \mathbf{od}, \sigma' \rangle} \qquad \frac{\langle gc, \sigma \rangle \rightarrow \mathbf{fail}}{\langle\ \mathbf{do}\ gc\ \mathbf{od}, \sigma \rangle \rightarrow \sigma}$$

Regole per i comandi (2)

$$\frac{\langle c_0,\sigma\rangle \xrightarrow{\lambda} \langle c_0',\sigma'\rangle}{\langle c_0 \parallel c_1,\sigma\rangle \xrightarrow{\lambda} \langle c_0' \parallel c_1,\sigma'\rangle} \qquad \frac{\langle c_1,\sigma\rangle \xrightarrow{\lambda} \langle c_1',\sigma'\rangle}{\langle c_0 \parallel c_1,\sigma\rangle \xrightarrow{\lambda} \langle c_0 \parallel c_1',\sigma'\rangle}$$

$$\frac{\langle c_0,\sigma\rangle \xrightarrow{\alpha?n} \langle c_0',\sigma'\rangle \;\; \langle c_1,\sigma\rangle \xrightarrow{\alpha!n} \langle c_1',\sigma\rangle}{\langle c_0 \parallel c_1,\sigma\rangle \rightarrow \langle c_0' \parallel c_1',\sigma'\rangle} \qquad \frac{\langle c_0,\sigma\rangle \xrightarrow{\alpha!n} \langle c_0',\sigma\rangle \;\; \langle c_1,\sigma\rangle \xrightarrow{\alpha?n} \langle c_1',\sigma'\rangle}{\langle c_0 \parallel c_1,\sigma\rangle \rightarrow \langle c_0' \parallel c_1',\sigma'\rangle}$$

Caso in cui i due comandi comunicano tra loro su α. Il caso in cui c_0 e c_1 sono comandi senza input e output è nelle due regole precedenti con λ vuoto

$$\frac{\langle c,\sigma\rangle \xrightarrow{\lambda} \langle c',\sigma'\rangle}{\langle c \setminus \alpha,\sigma\rangle \xrightarrow{\lambda} \langle c' \setminus \alpha,\sigma'\rangle} \text{ provided neither } \lambda \equiv \alpha?n \text{ nor } \lambda \equiv \alpha!n$$

Regole per i comandi con guardia

$$\frac{\langle b,\sigma\rangle \rightarrow \text{true}}{\langle b \rightarrow c,\sigma\rangle \rightarrow \langle c,\sigma\rangle} \qquad \frac{\langle b,\sigma\rangle \rightarrow \text{false}}{\langle b \rightarrow c,\sigma\rangle \rightarrow \text{fail}}$$

$$\frac{\langle b,\sigma\rangle \rightarrow \text{false}}{\langle b \wedge \alpha?X \rightarrow c,\sigma\rangle \rightarrow \text{fail}} \qquad \frac{\langle b,\sigma\rangle \rightarrow \text{false}}{\langle b \wedge \alpha!a \rightarrow c,\sigma\rangle \rightarrow \text{fail}}$$

$$\frac{\langle gc_0,\sigma\rangle \rightarrow \text{fail} \;\; \langle gc_1,\sigma\rangle \rightarrow \text{fail}}{\langle gc_0 [\!] gc_1,\sigma\rangle \rightarrow \text{fail}}$$

$$\frac{\langle b,\sigma\rangle \rightarrow \text{true}}{\langle b \wedge \alpha?X \rightarrow c,\sigma\rangle \xrightarrow{\alpha?n} \langle c,\sigma[n/X]\rangle} \qquad \frac{\langle b,\sigma\rangle \rightarrow \text{true} \;\; \langle a,\sigma\rangle \rightarrow n}{\langle b \wedge \alpha!a \rightarrow c,\sigma\rangle \xrightarrow{\alpha!n} \langle c,\sigma\rangle}$$

$$\frac{\langle gc_0,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}{\langle gc_0 [\!] gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle} \qquad \frac{\langle gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}{\langle gc_0 [\!] gc_1,\sigma\rangle \xrightarrow{\lambda} \langle c,\sigma'\rangle}$$

Dining philosophers problem

- In informatica, il problema dei filosofi a cena è un classico problema di sincronizzazione di multi-processi
- 5 filosofi sono seduti ad un tavolo davanti ad un piatto di spaghetti con una forchetta ad ogni lato (dunque, 5 filosofi, 5 piatti e 5 forchette).
- Si supponga che ciascuna filosofo ha bisogno di due forchette per mangiare e che esse siano prese una alla volta. Ciascun filosofo, dopo aver preso entrambe le forchette, mangia per un po', poi posa le forchette e comincia a pensare e poi ciclicamente riprende le forchette.
- Il problema è scrivere un algoritmo che evita starvation e deadlock.
  - Un Deadlock si può avere se tutti i filosofi hanno una sola forchetta e nessuno ne può prendere un'altra (catena di deadlock)
  - Una Starvation si può verificare se un filosofo non può prendere entrambe le forchette.
- La non disponibilità di forchette è analoga alla mancanza di risorse per un programma eseguito su un computer, dovuta alla scarsità di risorse. Una chiara situazione di **concorrenza**

Murano Aniello
Fond. LP - Ventesima Lezione

Solution for n Dining Philosophers

PROGRAM = $PHIL_{0..N-1}$ || $CHOPSTICK_{0..N-1}$

$PHIL_i$ = **do** true →
think;
$pickup_i$!0; $pickup_{i\oplus1}$!0;
eat;
$putdown_i$!0; $putdown_{i\oplus1}$!0
od

$CHOPSTICK_i$ = **do** true ∧ $pickup_i$?x → $putdown_i$?x **od**

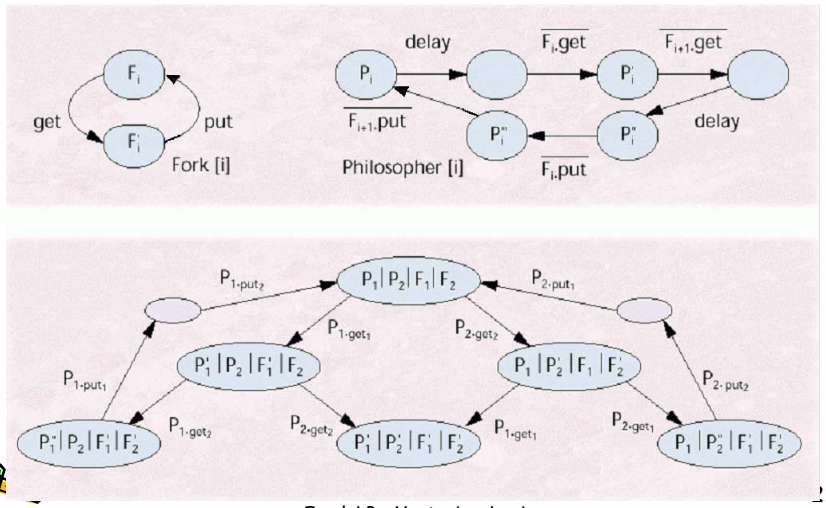
– Define i⊕1 to be (i + 1) mod n

- Issues
  - Can have deadlock and/or starvation
  - Uses dummy messages for synchronization
  - Chopstick cannot protect itself form other philosophers

CS 386L Fall 2004 10

Murano Aniello
Fond. LP - Ventesima Lezione

Verifying the philosophers' problem



Esempio pag.331 Winskel

Prossima lezione

- Rivedere i comandi di CSP dettagliatamente. Ripartire dalla sintassi, e per ogni comando dare prima una spiegazione intuitiva e poi una valutazione formale.
- Fare un esercizio sul linguaggio di Dijkstra come l'equivalenza di due comandi
- Fare un esempio su CSP e/o l'esempio a pag. 331 di Winskel.