Formalizzazione dei Sistemi Distribuiti

Mirko Viroli

DEIS - Università degli Studi di Bologna

mviroli@deis.unibo.it

Organizzazione del seminario

- L'importanza dei modelli per l'ingegneria dei sistemi complessi
- Il problema dei sistemi distribuiti
- La formalizzazione dei sistemi "non distribuiti"...
  - Macchine, Computabilità, Linguaggi, altri formalismi
- La formalizzazione dei sistemi "distribuiti"
  - Il problema dell'interazione
  - Macchine Interattive
  - Linguaggi per l'interazione
- Un framework di lavoro:
  - SOS + LTS
  - Esempi...

1/5

L’importanza dei modelli per l’ingegneria dei sistemi complessi

I modelli formali e l'ingegneria

- Modello : dà una rappresentazione astratta dei particolari di interesse di un sistema
- Esempi:
  - modelli fisici: rappresentazioni in scala..
  - modelli matematici: teoria delle probabilità..
- Quali modelli matematici per l'ingegneria?
  - descrivono in modo formale gli aspetti più complessi di un sistema da costruire
- Nell'ingegneria informatica:
  - danno una rappresentazione astratta e formale di entità e concetti, come:
    - comunicazione, algoritmi.. reti, computer,..

Separazione dei "Concerns"

- I modelli non devono rappresentare tutto il sistema

Tipicamente:

- Separazione delle problematiche in aspetti (concerns) il più possibile ortogonali fra loro
- Per ogni aspetto di interesse si definisce un modello che:
  - lo rappresenti come concetto "chiave"
  - che astragga da altri aspetti meno importanti
  - ossia: che si ponga al giusto livello di astrazione
- Per risolvere problemi estremamente complessi
  - si divide in diversi livelli di astrazione, affrontati in sequenza
  - ad esempio: top-down, bottom-up, o combinati

Il livello di astrazione

- La scelta del modello è essenziale per poter affrontare problemi complessi
- Esempio il linguaggio di programmazione
  - E' il modello dell'implementazione di un sistema
  - Su quali aspetti si concentra?
    - Assembler: rappresentazione interna del calcolatore
    - Lisp: rappresentazione matematico-funzionale
    - C++: rappresentazione ad oggetti (mappati su una memoria)
    - Java: rappresentazione ad oggetti (scollegati dalla memoria)
  - C++ o Java?: mi interessa l'allocazione degli oggetti in memoria?
- Una libreria di supporto (o un costrutto del linguaggio):
  - Può "alzare" il livello di astrazione
    - Es.: I/O + Serializzazione:

    ⇒ oggetti come entità sganciate dal RunTime

Ciclo di vita del software

- Specifica
  - Funzionamento del software
- Design
  - Organizzazione del software
- Implementazione
  - Costruire il software
- Validazione
  - Funziona correttamente?
- Che relazione fra fase e modello?
- I modelli formali possono essere d'ausilio in ogni fase
  - In questo seminario ci concentriamo sulla specifica

Il rapporto con le tecnologie

- Una tecnologia è spesso un fattore abilitante
  - Socket: comunicazione inter-processo
  - RMI: oggetto come "servizio" di un host
  - Corba: oggetto come "servizio" di un sistema
  - Cgi: procedura come URL nel web
  - Servlet: servizio come URL nel web
  - JSP: servizio come pagina web
- Ma consente anche di innalzare il livello di astrazione nel processo ingegneristico
  - specifica: alcuni aspetti non sono più da definire
  - design: alcuni aspetti non vanno più progettati
  - implementazione: alcuni "mattoni" sono già forniti
  - validazione: alcune proprietà sono già verificate

2/5

Il problema dei sistemi distribuiti

Cos'è un sistema "distribuito"?

- Generalmente:
  - un sistema composto da diverse entità, il cui comportamento collaborativo porta ad un risultato globale
- Tipici aspetti dei sistemi distribuiti:
  - comunicazione: trasmissione di informazione fra entità
  - sincronizzazione: nozione di "evento" comune
  - concorrenza: evoluzione contemporanea delle entità
  - non determinismo: latenza reti, perdita messaggi
- Queste problematiche esistono anche in sistemi non distribuiti:
  - Sistemi ad eventi: GUIs, Beans, ...
- Il punto chiave:
  - il concetto di INTERAZIONE: momento di sincronia/comunicazione fra entità concorrenti.

La complessità delle interazioni

- Come rappresentare le interazioni fra i componenti di un sistema distribuito:
  - quale informazione portano con loro?
  - qual è la causa che le genera?
  - da chi (e se) vengono ricevute/percepite, quando?
  - come raggiungono il destinatario?
  - quali relazioni fra gruppi di interazioni? (causalità, consistenza,..)?
  - che relazione fra interazione e cambiamento di stato?
- Come costruire un sistema ingegneristico che affronti questi aspetti?
- I modelli per i sistemi distribuiti devono consentire di comprenderne le problematiche, descriverle, progettarle, implementarle, validarle, etc...

La somma delle parti

- Ulteriore complessità:
  - dati n componenti ognuno col proprio funzionamento/compito
    - (Come si definisce il compito di ognuno?)
  - comporli
    - (Con quale "colla"?)
  - in modo da ottenere un certo comportamento globale
    - (Come si definisce il compito globale?)
- Come garantire che gli "invarianti" siano preservati?
- Come "governare" le interazioni tra loro?
- "Un sistema è più della somma delle parti"
- Modelli e tecnologie di *coordination*
  - Hanno come principale *concern* l'interazione

Obiettivi (o utopie?)

- Caratterizzare in modo formale il comportamento interattivo di un componente software
  - es.: emette un messaggio ogni secondo e ne riceve uno ogni due
- Aggregare componenti tra loro
  - in modo sincrono/asincrono,..
- Ottenere il comportamento del sistema risultante
- Cosa succede se sostituisco un componente con un altro?
  - il sistema funziona meglio, peggio, non funziona più....
- Data una descrizione formale ottenere in modo automatico un sistema dal corrispondente comportamento
- Riusare sistemi legacy con un approccio black-box:
  - osservando il suo comportamento interattivo

3/5

La formalizzazione dei sistemi “non distribuiti”

Computare = Trasformare

- Nella sua accezione più tradizionale, una computazione è una trasformazione:
  - Valore di ingresso: stringa su Σ, ossia un elemento di Σ^*
  - Valore di uscita: elemento di Δ^*
- Esempio:
  - invertire una stringa:
    - $\Sigma=\{1,2,3\}$, 1223 viene trasformata in 3221
- Un altra accezione di computazione (equivalente) è:
  - riconoscimento di una stringa, es.: 1* 2 3*
  - 111223 no 12333 si

Algoritmo = Funzione

- Posso rappresentare (modellare) ogni algoritmo in questo modo?
  - Ogni struttura dati può essere rappresentata come stringa di un linguaggio (es.: la sua rappresentazione in memoria)
  - Ogni algoritmo è un procedimento che prende una struttura dati in input e produce una struttura dati in output
    - Riconosce una stringa: $\Sigma^* \rightarrow$ Boolean
    - Trasforma una stringa: $\Sigma^* \rightarrow \Delta^*$
    - Ordinamento di un vettore: $V \rightarrow V$
    - Ricerca di un elemento in un vettore: $VxE \rightarrow$ Boolean
    - ...
- Quindi un algoritmo rappresenta una funzione matematica
- Un "server web" realizza un algoritmo?

Quali funzioni?

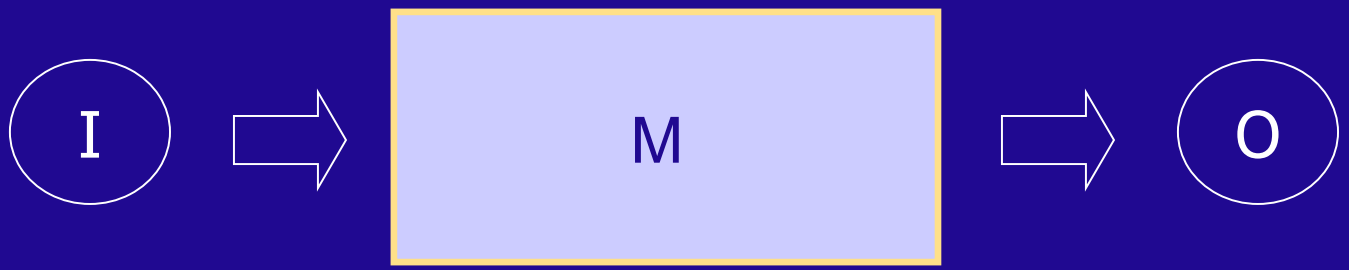
- Quali domini?
- Insiemi numerabili, ossia elementi rappresentabili in modo finito:
  - Finiti
  - Sono in corrispondenza 1:1 con i numeri naturali
  - Quali fra questi?
    - N,NxN,Q,R,{1,2,3}*,N*, successioni su N, successioni di bit
- Quindi ogni algoritmo, da un punto di vista concettuale, è rappresentabile come funzione

$$N \rightarrow N$$

- Il viceversa? Cos'è un algoritmo?
  - La descrizione di un procedimento "meccanico" che trasforma stringhe (o numeri interi)

Le macchine calcolatrici

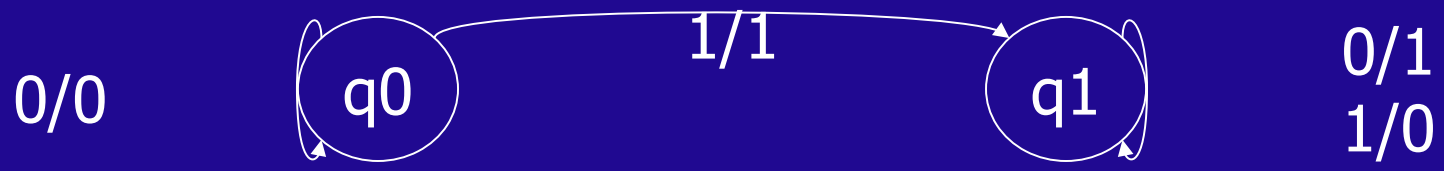
- Tipico modo per rappresentare un algoritmo
  - Modellarlo come "programmazione" di una macchina calcolatrice
- Algoritmo: programmazione della macchina
- Funzione: macchina
- Applicazione di funzione: esecuzione del processo associato alla macchina
- Schema di principio:



L'automa a stati finiti

- Ha un insieme finito di stati
  - prende un simbolo in input e a seconda dello stato corrente:
  - cambia stato e produce un simbolo in output
- Formalmente: $(\Sigma^* \rightarrow \Delta^*)$

  $<Q,\ \Sigma,\ \Delta,\ \delta:Q x \Sigma_\varepsilon \rightarrow Q x \Delta,\ q0,\ F \subseteq Q>$
- Esempio: $0^*\ 1\ (1|0)^* \rightarrow 0^*\ 1\ (0|1)^*$: 001010 → 001101
  - negazione di un numero espresso in complemento a due
  - <{q0,q1},{0,1},{0,1},{(q0,0→q0,0),(q0,1→q1,1),..},q0,{q1}>



Quale espressività

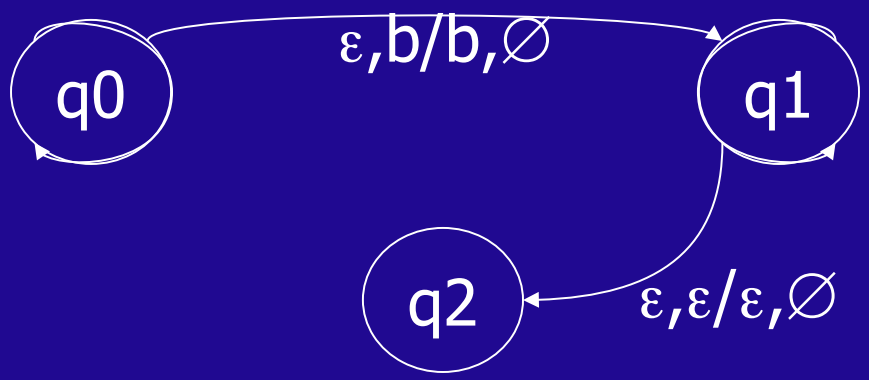
- L'automa a stati finiti sa riconoscere le espressioni regolari
  - Reg ::= s | ε | Reg* | Reg Reg | (Reg|Reg)
  - oppure ottenute con produzioni
    - A ::= ε, A ::= a, A ::= b A
- Es.: se l'input è un binario dispari (0|1)* 1 ritorna 1, altrimenti 0
  - A::=1, A::=0A, A::=1A
- Non sa riconoscere le espressioni context-free:
  - A::= <qualsiasi cosa> (e non: AB ::= CD)
  - possono essere messe nella forma: A::=a, A::= BC
  - esempio: $a^n bc^n$ ossia: S::=b| aSc
- Non possiamo riconoscere se le parentesi in una espressione matematica sono bilanciate..

L'automa push down

- Ha un insieme di stati finiti, e uno stack
  - prende un input, e a seconda dello stato e del top dello stack:
  - cambia stato, produce un simbolo in output, fa un pop/push
- Formalmente: ($\Sigma^* \rightarrow \Delta^*$, con simboli nello stack Γ)

  $<Q,\ \Sigma,\ \Delta,\ \Gamma,\ \delta: Q x \Sigma_\varepsilon x \Gamma_\varepsilon \rightarrow Q x \Delta x \Gamma^*,\ q0,\ F \subseteq Q>$
- Esempio: invertire una stringa di bit: 010110ε

b,ε/ε,b
b,0/ε,0b
b,1/ε,1b



ε,b/b,∅

Quale espressività

- Riconosce le stringhe di linguaggi context-free
- Non può riconoscere ogni tipo di linguaggio, esempio:
  - come riconoscere stringhe del tipo $a^n b^n c^n$
- E' necessario disporre di una struttura di supporto più flessibile della pila

La macchina di Turing

- Ha un insieme di stati finiti, e un nastro (illimitato) con testina
  - legge un valore, e a seconda dello stato:
  - cambia stato, scrive un valore/sposta la testina
- Formalmente: $(\Sigma^* \rightarrow \Sigma^*)$

  $<Q, \Sigma, \delta:Q\times\Sigma \rightarrow Q\times\Sigma_\varepsilon\times\{L,R\}, q0, qRej>$
- Con ipotesi del tipo:
  - L'effettivo input è delimitato fra due simboli speciali (...#.....#...)
  - La testina è puntata sul # più a sinistra....
- Si può dimostrare che l'espressività non cambia avendo a disposizione più nastri:
  - di cui uno di lettura, uno di scrittura, ed altri di "lavoro".
  - solito schema input/output

Le funzioni ricorsive parziali

- Quali funzioni sugli interi si possono rappresentare?
  - Le macchine di Turing sono numerabili
  - Le funzioni da N a N sono non numerabili

  $\rightarrow$Ci sono una infinità non numerabile di funzioni non rappresentabili
- Quelle rappresentabili si chiamano computabili (ricorsive):
  - Funzioni di base:
    - fz(n)=0
    - fz(n)=n+1
    - fi(n1,...nk)=ni
  - Composizione:
    - g, h1,...hk:NxNx..xN$\rightarrow$N, allora f=g(h1(..),h2(..),..,hk(..))
    - g,h allora:
      - f(x1,..,xn,0)=g(x1,..,xn)
      - f(x1,..,xn,y+1)=h(x1,..,xn,y,g(x1,..,xn))
    - g:NxNx..xN, allora f(x1,..,xn)=$\min_y${g(x1,..,xn,y)=0} o $\perp$

La tesi di Church

- Quali algoritmi non si possono rappresentare?
  - Si definiscono tramite il procedimento di aritmetizzazione (Gödel)
    - Associo ad ogni macchina di Turing My un numero y
    - Funzione: g(x,y)= 1 se My termina la computazione di x, o 0
    - Ciò porterebbe ad una contraddizione.... <<<g(x,x)>>>
- Un formalismo non può predicare su se stesso! (Gödel)
  - Ciò limita fortemente la validazione dei programmi!
- Ci serve un formalismo più potente?

- 1 Nessun formalismo per modellare computazioni meccaniche può essere più potente delle MT
- 2 Ogni algoritmo può essere codificato in termini di una MT

Computare con un linguaggio

- Quale relazione fra le MT e i linguaggi di programmazione?
- La macchina di Stepherdson e Sturgis è espressiva come la MT
  - N registri contenenti numeri interi
  - Un insieme di istruzioni del tipo
  - L: inc(i)
  - L: jdec(i,Lo)
- Ossia:
  - Un Assembler con registri ad interi e solo due tipi di istruzioni sarebbe sufficiente a codificare ogni algoritmo
- Perchè allora i programmatori non usano solo l'Assembler?
  - E il livello d'astrazione???????

Il lambda-calcolo

- Un modello computazionale basato solo sul concetto di:
  - applicazione di funzione
- Sintassi e semantica

$$L ::= \lambda x.L \mid x \mid LL$$

$$(\lambda x.L)M \rightarrow L[M/x]$$

- λ: applicazione, x:parametro , L body, M argomento (input)
- Esempio di beta-reduction:

$$(\lambda x.xx)(\lambda y.(\lambda z.yz)) \rightarrow (\lambda y.(\lambda z.yz))\ (\lambda y.(\lambda z.yz))$$

Lambda: Semantica

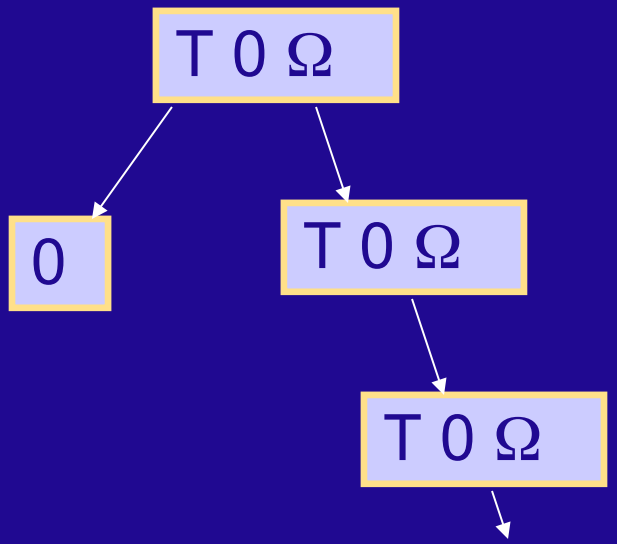
- Input: una Lambda (ce ne sono una infinità numerabile)
- Trasformazione: applico una qualsiasi riduzione
- Quando non ce ne sono più.... quello che resta è il risultato
  - Si chiama Lambda in forma normale
- Qual'è il programma della macchina "lambda"?
- Esempio: il calcolo dei predicati
  - Denoto: $T = \lambda t.\lambda f.t$, $F = \lambda t.\lambda f.f$
  - $Not(x) = \lambda x.x\ F\ T = \lambda x.x\ (\lambda t.\lambda f.f)\ (\lambda t.\lambda f.t)$
  - Es.: $Not\ T \rightarrow T\ F\ T \rightarrow F$
  - $And(x,y) = \lambda x.\lambda y.\ x\ y\ F$
  - Es.: $And\ T\ F \rightarrow T\ F\ F \rightarrow F$

Lambda: l'aritmetica

- I numeri naturali:
  - 0 = λz.z (funzione identità)
  - 1 = λz.λs.sz (applica il secondo argomento al primo)
  - 2 = λz.λs.s(sz) (etc...)
  - ...
  - succ N = λn.λz.λs.s (n z s)
    - succ 1 → λz.λs.s (1 z s) → λz.λs.s (sz) = 2
  - add M N = λm.λn.λz.λs.n (m z s) s
    - add 1 1 → λz.λs.1 (1 z s) s → λz.λs.1 sz s → λz.λs.s (sz) = 2
- La ricorsione?
  - Tramite Y= λf. (λx.f(x x)) (λx.f(x x)), Y F → → F(Y F) ... F(F(Y F))
  - Oppure Ω = (λx.xx) (λx.xx) → Ω → Ω → Ω

Church-Rosser

- Consideriamo un grafo con Lambda nei nodi e archi per riduzioni
  - In generale:
    - Lambda fortemente normalizzabile: la computazione termina
    - Lambda (deb.) normalizzabile: la computazione può terminare
    - Lambda non normalizzabile: la computazione non termina



  - Qualunque sia la radice, non ci può essere più di una foglia
- Massima espressività ⇔ non terminazione

Lambda: combinatori

- Ulteriore semplificazione....
- Definiti i 3 combinatori:
  - $I = \lambda x.x$
  - $K = \lambda x.\lambda y.x$
  - $S = \lambda x.\lambda y.\lambda z.(xz)(yz)$
- Ogni Lambda, è esprimibile come composizione di combinatori:
  - I K I, S I (K I) I, ...
  - T = K, F = K I, 0 = I, 1= ?, 2 = ?, ...

Applicazioni del lambda

- E' uno dei modelli più semplici per la computazione
  - ogni step computazionale rappresentato come applicazione
- Estensioni del lambda sono utilizzate per descrivere formalmente sintassi e semantica dei linguaggi di programmazione:
  - Es.: l'ereditarietà dei linguaggi ad oggetti
  - I tipi di dato generico (template)
- Ha ispirato linguaggi di programmazione funzionale:
  - Lisp, Haskell, ML, ...

4/5

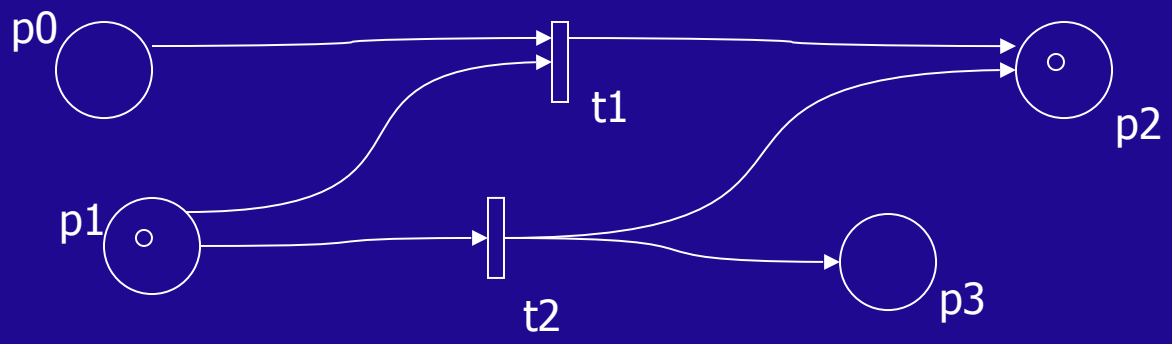
La formalizzazione dei sistemi “distribuiti”

Aggiungere l'interazione...

- I sistemi software moderni:
  - server web
  - ambienti di programmazione visuale
  - il sistema gnutella
- Si può rappresentare il loro funzionamento in termini di una trasformazione ingresso/uscita? NO!!!!!
- Tuttavia, questo può valere per alcune loro (rilevanti) sotto-parti:
  - trasformazione di un XML in HTML
  - compilazione di un file sorgente Java in bytecode
  - ...
- Servono modelli in grado di rappresentare adeguatamente le astrazioni tipiche dei sistemi distribuiti

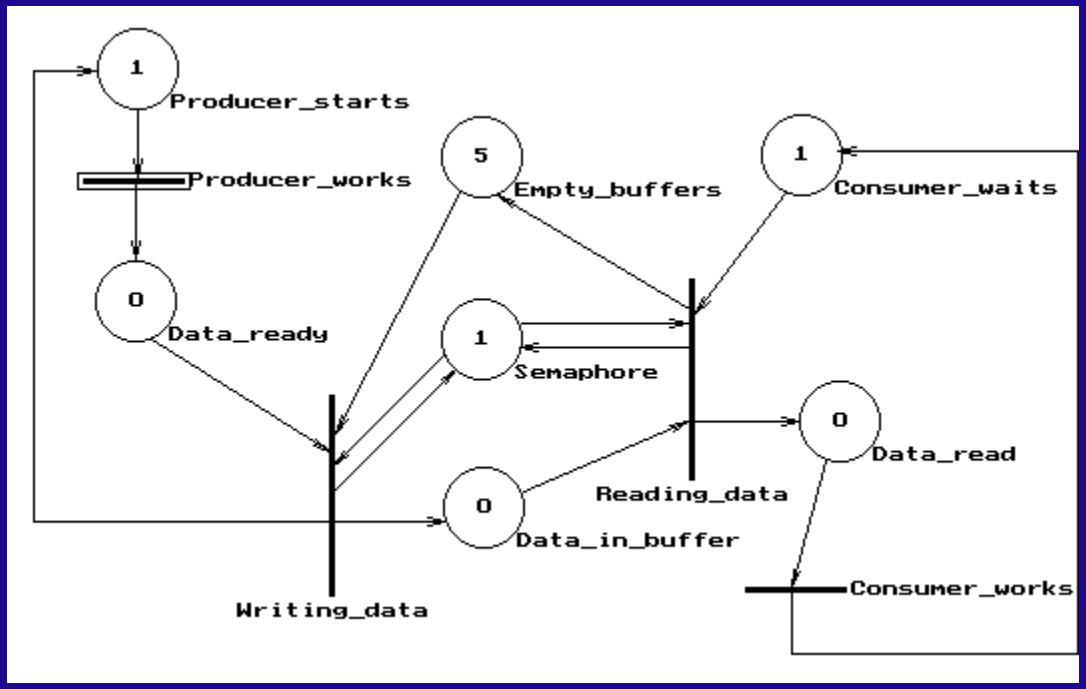
Reti di Petri

- E' un modello orientato alla concorrenza
- E' un grafo orientato che connette "posti" e "transizioni"
- Formalmente: $<P,T,IF:PxT \rightarrow N,OF:TxP \rightarrow N>$
- Marcatura: $M:P \rightarrow N$, Transizione t abilitata: $M(p) \geq IF(p,t)\ \forall p$
- Transizione di una marcatura: $M \Rightarrow M'$,
  - Esiste t abilitata: per ogni p, $M'(p)=M(p)-IF(p,t)+OF(t,p)$



Applicazioni

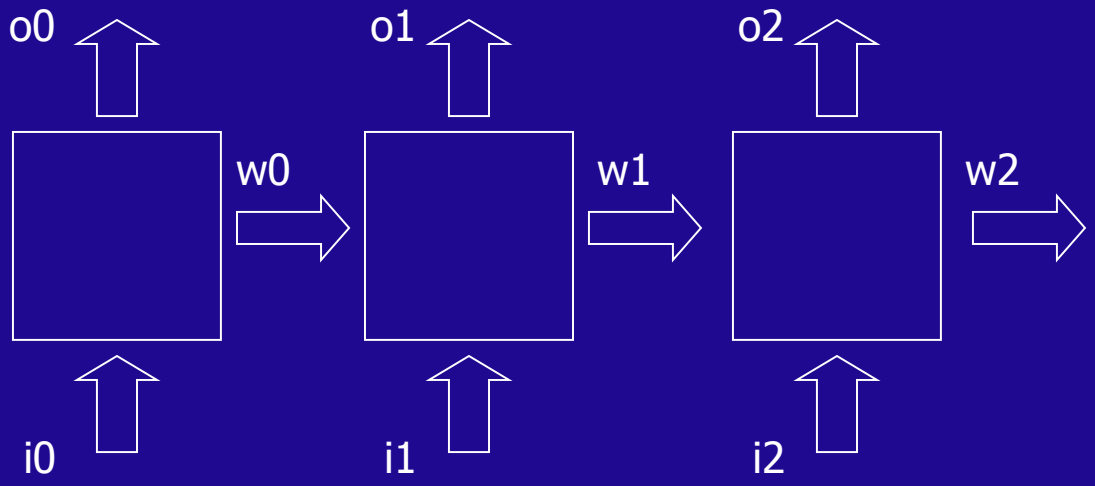
- Rappresenta l'evoluzione di uno stato "distribuito"



- E' comunque un automa (non.determ.) che trasforma marcature
- interazione vs. trasformazione

La Persistent Turing Machine

- E' una Turing Machine a tre nastri input/work/output
- Modello computazionale:
  - Accetta un input
  - Usa work e input per modificare work e output
  - Quando la computazione termina, emette output.. e ricomincia
- A differenza della TM, il contenuto del nastro Work persiste



Computare interaction histories

- Tuttavia la sua semantica è diversa
- La sua espressività è caratterizzata da:
  - una funzione da N* a N*, non più da N a N
- Comunque il set di PTM rimane numerabile
- Infatti, ogni PTM è simulabile da una TM:
  - f (i0,i1,i2,....,in)=on oppure f (i0,i1,i2,....,in)=o0;o1;..;on
- Ma non può cogliere il concetto di causalità

- Il formalismo della PTM è il primo tentativo (1997) per dare una diversa caratterizzazione al concetto di computabilità.

I linguaggi per l'interazione

- I linguaggi hanno primitive di interazione fin dagli anni 70
- Es.: operazioni di read e write inserite in mezzo a cicli... etc...
- Successivamente:
  - Socket, per la comunicazione e sincronizzazione inter-processo
- OOP:
  - concettualmente: è intrinsecamente interattiva
  - lo è "ancor di più" con la programmazione ad eventi
- Quale "fondazione" per questi linguaggi?
- Esiste un analogo del lambda-calcolo?
  - La via dell'estensione del lambda-calcolo verso l'interazione è fallita, e' stato necessario rivedere completamente il modello!

Il CCS (Calculus for communicating systems)

- E' un calcolo per rappresentare "processi" e "interazioni"
- Azioni (o interazioni) interleaved:
  - Azione "silente" τ: rappresenta una evoluzione interna
  - Nomi e co-nomi a,$\underline{a}$: coppie di mutua interazione fra processi

$$P ::= a(x).P \mid \overline{a}(x).P \mid \tau.P \mid P + P \mid (P|P) \mid P/a \mid !P \mid 0$$

- Esempi:

$$a(x).\overline{b}(x) \qquad !a(x).\overline{b}(x)$$
$$a(x).P \mid \overline{a}(y).Q \qquad a(x).(b(y) + c(y))$$

CCS: semantica

- P può evolvere in P' per effetto dell'azione α:

$$P \xrightarrow{\alpha} P'$$

- Operatori | e + commutativi e associativi, !P ≡ P | !P
- Regole di transizione (evoluzione dei processi):

$$\alpha.P \xrightarrow{\alpha} P \quad \text{(ACT)} \qquad \alpha.P + Q \xrightarrow{\alpha} P \quad \text{(SUM)}$$

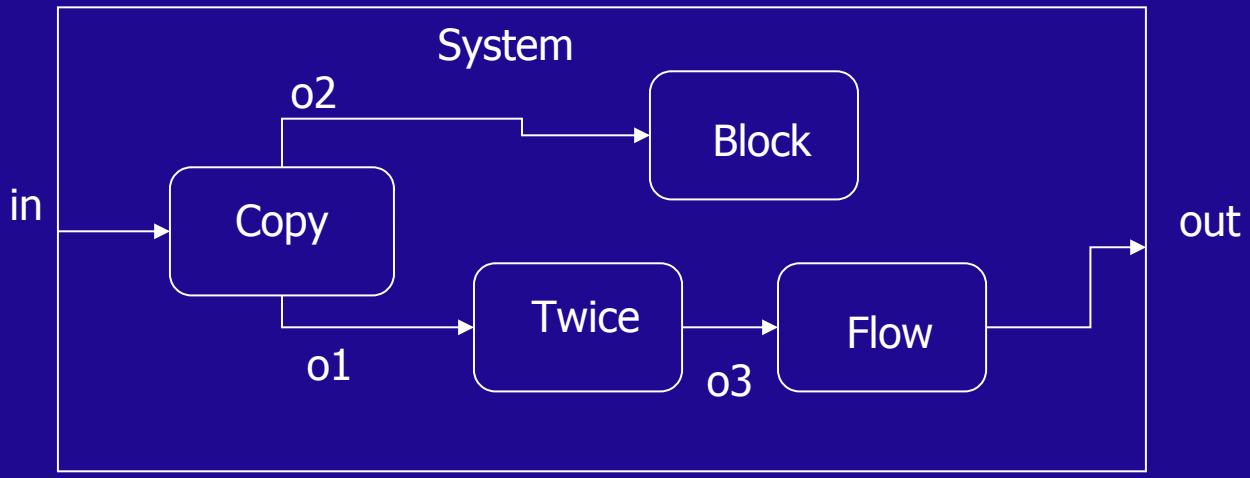
$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{(PAR)} \qquad \frac{P \xrightarrow{\lambda} P' \qquad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \text{(REA)}$$

- Esempio:

$$a(x).\overline{b}(x) \mid \overline{a}(5) + \overline{a}(6) \xrightarrow{\tau} \overline{b}(6) \mid 0 \xrightarrow{\overline{b}6} 0$$

Esempio

$$
\begin{array}{ll}
Twice(in, out) := & !in(x).\overline{out}(2x) \\
Copy(in, out_1, out_2) := & !in(x).\overline{out_1}(x).\overline{out_2}(x) \\
Block(in) := & !in(x) \\
Flow(in, out) := & !in(x).\overline{out}(x) \\
System(in, out) := & Copy(in, o1, o2) | Twice(o1, o3) | \\
 & Flow(o3, out) | Block(o2)
\end{array}
$$



$$System(in, out) \mid \overline{in}(5) \xrightarrow{\tau}{}^{*} System(in, out) \mid \overline{out}(10).0 \xrightarrow{\overline{out}(10)} System(in, out)$$

L'observation equivalence

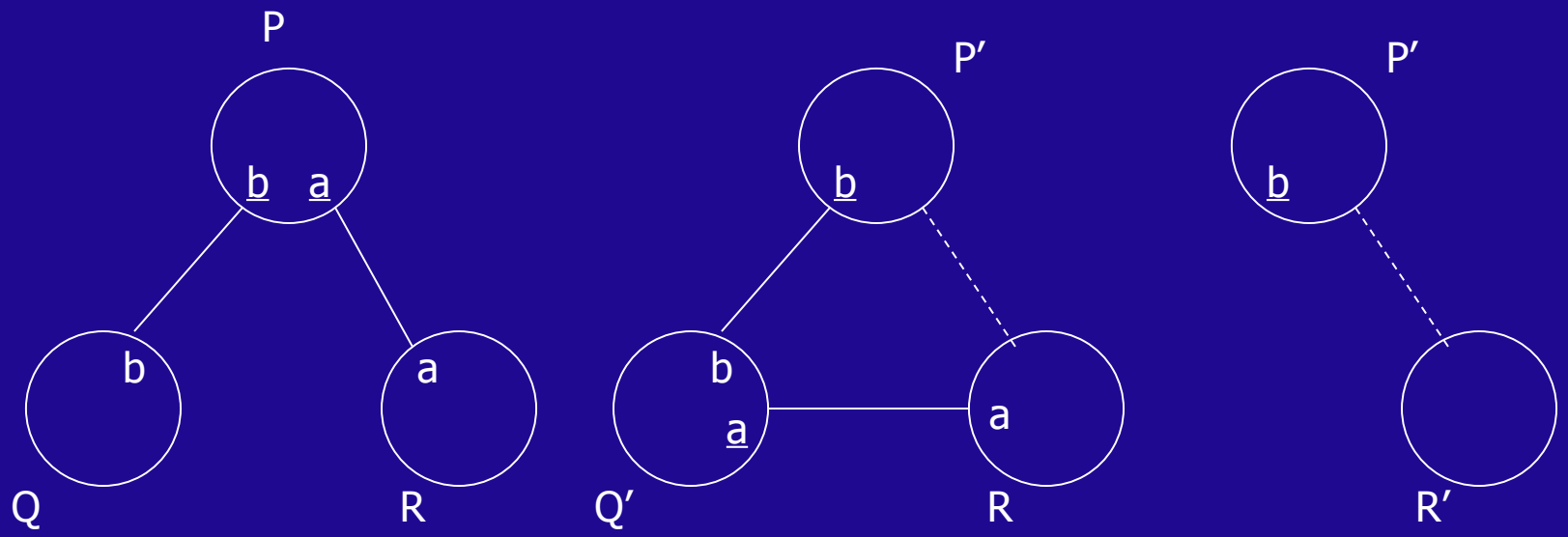
- Come caratterizzare il comportamento di un sistema?
  - Per mezzo di quello che è possibile osservare ai "morsetti"!
- Nel nostro caso, System(in,out):
  - riceve messaggi del tipo in(x)
  - emette messaggi del tipo out(2x)
- E' sincrono? preserva l'ordine?
- Per i sistemi finiti (senza "!"), è possibile stabilire se due componenti hanno lo stesso comportamento osservabile
  - ammettono le stesse interaction histories (trace semantics,....)
  - le interaction histories si simulano a vicenda (bisimulation,...)
- Per sistemi finiti... il problema non è, in generale, computabile

Il pi-calculus

- Estensione del CCS che consente:
  - di definire processi la cui struttura evolve
  - di rappresentare qualunque computazione
  - e' il framework candidato a definire l'espressività dell' "interazione"
- Idea base:
  - l'informazione passata sui canali è a sua volta il nome di un canale
  - ogni invio di messaggio altera la struttura del processo
    - chi lo riceve ha un link in più su cui può emettere/ricevere
    - un link scompare quando un processo non usa più il suo nome
- Sintassi e semantica:
  - *sostanzialmente* simile a quella del CCS

pi: Esempio

$$P := \overline{b}a.\overline{b}5.P'$$
$$Q := b(x).b(y).\overline{x}y.0$$
$$R := a(x).R'$$
$$System := P \mid Q \mid R$$
$$System \xrightarrow{\tau}\xrightarrow{\tau} P' \mid \overline{a}5.0 \mid R \xrightarrow{\tau} P' \mid R'$$



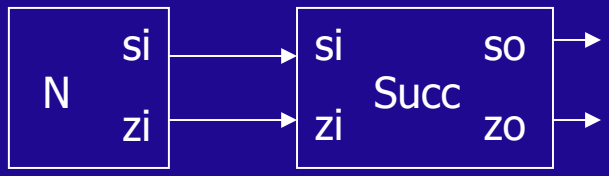
pi: l'aritmetica

- Si può rappresentare l'aritmetica in termini di processi:
- Un numero N è un processo:
  - è identificato da due canali s,z
  - manda N segnali su s e uno su z
  - poi termina (0)

$$0(s,z) := \overline{z}$$
$$1(s,z) := \overline{s}.\overline{z}$$
$$2(s,z) := \overline{s}.\overline{s}.\overline{z}$$

- Successivo è un processo che va attaccato ad un numero:

$$Copy(s_i, z_i, s_o, z_o) := s_i.\overline{s_o}.Succ(s_i, z_i, s_o, z_o) + z_i.\overline{z_o}$$
$$Succ(s_i, z_i, s_o, z_o) := \overline{s_o}.Copy(s_i, z_i, s_o, z_o)$$

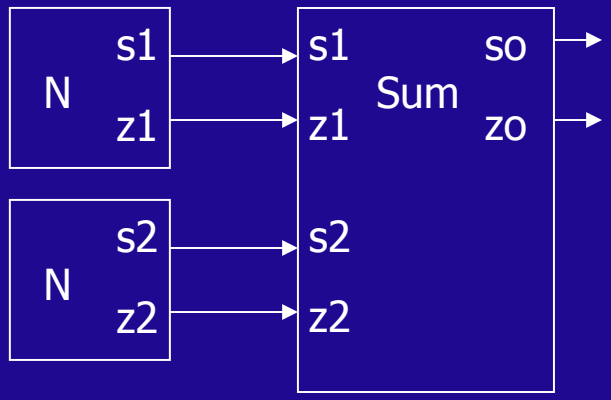


$$1(s_i, z_i) \mid Succ(s_i, z_i, s_o, z_o) \equiv 2(s_o, z_o)$$

pi: altra aritmetica...

- Somma:

$$Sum(s_1, z_1, s_2, z_2, s_o, z_o) ::= \quad s_i.\overline{s_o}.Sum(s_1, z_1, s_2, z_2, s_o, z_o) + z_1.Copy(s_2, z_2, s_o, z_o)$$



- In generale:
  - Per ogni lambda-expression è possibile costruire un processo pi equivalente, in modo che la transizione silente simuli la riduzione

aI-pi-calculus

- pi-calculus è molto espressivo, ma molto più complesso se messo a confronto col lambda
- Vari tentativi di semplificazione
- a-pi-calculus: solo comunicazioni asincrone
  - La continuazione di un invio messaggio è sempre nulla
  - Processo che invia messaggio = messaggio in attesta di essere ric.
- I-pi-calculus: si inviano solo nomi "privati"
  - Semplificazione sintattica e di gestione dei nomi
  - Invio e ricezione diventano simmetrici
- aI-pi

$$P ::= \overline{x}(y) \quad | \quad x(y).P \quad | \quad (P|P) \quad | \quad !P$$

5/5

Un Framework di Lavoro

Applicazioni alla modellizzazione

- Lambda-calculus e Pi-calculus consentono di rappresentare ogni behaviour trasformazionale ed interattivo...
- Ma spesso non è utile vedere ogni computazione come applicazione di una lambda o comunicazione di un nome
- Tuttavia possiamo usare i concetti di riduzione e transizione per:
  - descrivere l'evoluzione interna dei componenti
  - descrivere in che modo interagiscono con l'environment
  - descrivere (se esiste) la colla fra i componenti
- Quali applicazioni?
  - Specifica: consentono di definire in dettaglio il sistema
  - Design: veicolano in modo più "sicuro" verso una implementazione
  - Implementazione: generazione automatica del codice?
  - Validazione: applicazione tool di verifica ancora limitata....

Transition Systems

- L'evoluzione "interna" di un componente C è rappresentabile:
  - <S,→> con:
    - S insieme degli stati ammissibili per C
    - ∀ →⊆ SxS, transizioni (riduzioni) da stato a stato ammissibili
- Le interazioni di un componente C sono rappresentabili con:
  - <S,→,A> con:
    - S insieme degli stati
    - A insieme delle azioni
    - ∀ →⊆ SxAxS, transizione da stato a stato per mezzo di una azione
- ∀ → definiti da un insieme di regole "dichiarative":

$$\frac{condition}{s \rightarrow s'} \qquad \frac{condition}{s \xrightarrow{a} s'}$$

Regole di transizione

- Es.: contatore S=N={0,1,2,3,...}
  - può "spontaneamente" incrementare di valore (clock interno)
  - accetta comandi di *reset*, *inc*
  - arrivato a 50 emette un segnale di *alarm*
- <S,$\rightarrow_\tau$,$\rightarrow$,A={reset,inc,alarm}> con $\rightarrow_\tau \subseteq$ SxS e $\rightarrow \subseteq$ SxAxS

$$\frac{n < 50}{n \longrightarrow_\tau n+1} \qquad \frac{-}{50 \xrightarrow{alarm} 0} \qquad \frac{n < 50}{n \xrightarrow{inc} n+1} \qquad \frac{-}{n \xrightarrow{reset} 0}$$

$$0 \longrightarrow_\tau 1 \longrightarrow_\tau 2 \xrightarrow{reset} 0 \longrightarrow_\tau 1 \longrightarrow_\tau$$
$$\longrightarrow_\tau 2 \xrightarrow{inc} 3 \longrightarrow_\tau 4 \longrightarrow_\tau \ldots \longrightarrow_\tau 50 \xrightarrow{alarm} 0$$

- Può modellare un oggetto contatore? Quali metodi?

Comunicazione sincrona

- Rappresentiamo un insieme di componenti, ognuno con stato nell'insieme C e con nome id
- Ogni componente invia o riceve un numero:
  - <C, $\rightarrow_s$, $\rightarrow_r$,IdxN>
- Un "sistema" S è una aggregato di componenti:
  - <S::= ε | (<id,c>|S), $\rightarrow$ >

$$\frac{c_1 \xrightarrow{id_2,n}_s c_1' \qquad c_2 \xrightarrow{id_1,n}_r c_2'}{\langle id_1, c_1\rangle \mid \langle id_2, c_2\rangle \mid s \longrightarrow \langle id_1, c_1'\rangle \mid \langle id_2, c_2'\rangle \mid s}$$

- E' un buon modello per la comunicazione sincrona?

Comunicazione asincrona

- Bisogna aggiungere il concetto di messaggio pendente (MP):
  - è stato inviato ma non ancora ricevuto
- Un "sistema" S è una aggregato di componenti e MP:
  - <S::= ε | (<id,c>|S) | ([id,n]|S), → >

$$\frac{c \xrightarrow{id_r, n_r}_s c'}{\langle id, c\rangle \mid s \longrightarrow \langle id, c'\rangle \mid [id_r, n_r] \mid s}$$

$$\frac{c \xrightarrow{id, n}_r c'}{\langle id, c\rangle \mid [id, n] \mid s \longrightarrow \langle id, c'\rangle \mid s}$$

- Quale ordering dei messaggi?

Comunicazione asincrona ordered

- Bisogna aggiungere il concetto di coda dei messaggi pendenti:
  - ad esempio: una unica per tutto il sistema (altre scelte?)
- Un "sistema" S è una aggregato di componenti e ha una coda:
  - R::= ε | (<id,c>|R), <S::= [id,n]* ⊕ R, → >

$$\frac{c \xrightarrow{id',n'}_s c'}{[id_q,n_q]^* \oplus \langle id,c\rangle \mid r \longrightarrow [id_q,n_q]^*;[id',n'] \oplus \langle id,c'\rangle \mid r}$$

$$\frac{c \xrightarrow{id,n}_r c'}{[id,n];[id_q,n_q]^* \oplus \langle id,c\rangle \mid r \longrightarrow [id_q,n_q]^* \oplus \langle id,c'\rangle \mid r}$$

- Si va verso l'idea di una astrazione intermedia ai componenti

Coordination

- Prevedere un luogo concettuale che realizza la colla fra i comp.
  - ne governa le interazioni
  - rappresenta una infrastruttura di "raccordo" fra i componenti
  - separazione interazione, computazione
- Esempio, blackboards:
  - lavagna: i messaggi possono essere messi (out), letti (rd), consumati (in).
  - "rd" e "in" utilizzano il pattern matching: out(1,2,3)... rd(X,2,3)
  - "rd" e "in" potrebbero non rispondere immediatamente
- Disaccoppiamento
  - temporale: non c'è sincronia fra chi manda e riceve
  - spaziale: non si specifica il ricevente

Coordination medium

- Quale caratterizzazione ai morsetti per un coordination medium:
  - accetta un messaggio alla volta (IE)
  - risponde con un insieme di messaggi (P(OE))
  - nel frattempo non ne accetta altri
- Non c'è più comunicazione peer-to-peer
  - il medium gestisce ogni interazione
  - chi manda un msg non specifica il destinatario
  - chi lo riceve non sa chi lo ha mandato

Transizioni

- <CM,→, IExP(OE)>
  - IE::=[id,out(t)] | [id,rd(t)] | [id,in(t)] – OE::=[id,t]
- Semantica del sistema: (es.: comunicazioni tutte sincrone)
  - R::= ε | (<id,c>|R), <S::=CM ⊕ R, → >

$$\frac{c_s \xrightarrow{op}_s c_s' \qquad cm \xrightarrow{[id_s,op],\{[id_1,t_1],\ldots,[id_n,t_n]\}} cm' \qquad c_1 \xrightarrow{t_1}_r c_1' \qquad \ldots \qquad c_n \xrightarrow{t_n}_r c_n'}{cm \oplus \langle id_s, c_s\rangle \mid \langle id_1, c_1\rangle \mid \ldots \mid \langle id_n, c_n\rangle \longrightarrow cm' \oplus \langle id_s, c_s'\rangle \mid \langle id_1, c_1'\rangle \mid \ldots \mid \langle id_n, c_n'\rangle}$$

Nell'ambito del corso...

- Come usare tutto ciò nell'ambito del corso (progetti)
- Almeno per quello che riguarda la specifica:
  - Per chi progetta sistemi di una certa complessità
  - (dal punto di vista delle interazioni fra le sue sottoparti)
- Fornire una specifica di alcuni aspetti di interesse con:
  - Transition Systems (comunicazione, protocolli)
  - CCS (organizzazione generale del sistema)
  - Reti di Petri (evoluzione del sistema, politiche di sincronizzazione)
  - Diagramma degli stati....
- Ed enfatizzare come la specifica ha guidato verso una implementazione "sicura"

Riferimenti:

- Sulla teoria della computabilità:
  - Carlo Ghezzi e Dino Mandrioli, “Informatica Teorica”, CittàStudi (24)
- Sul lambda calcolo:
  - H.P. Barendregt, “The Lambda Calculus”, North Holland, 1990
- Sulle reti di petri:
  - J. Peterson, “Petri Net Theory and the Modelling of Systems” Prentice - Hall 1981
- Sul problema interazione vs. algoritmi:
  - P.Wegner, “Why Interaction is More Powerful than Algorithms”, Communications of the ACM, 40(5), 1997.
- Sul pi-calcolo:
  - R.Milner, “Communicating and Mobile Systems: the pi-calculus”, Cambridge University Press (540)