Lezione 2 MSC
Overview del corso – Seconda parte

Roberto Gorrieri

Modelli e Sistemi Concorrenti

- Come modellare un sistema concorrente
- Quando due sistemi sono equivalenti
- Quali facilitazioni linguistiche sono disponibili
- Proprietà di congruenza e assiomatizzazioni
- Logiche di specifica di proprietà
- Strumenti automatici di verifica
- Espressività
- Reti di Petri

Un linguaggio per descrivere modelli

Perché può essere utile avere un tale linguaggio?

- Fornire una supporto linguistico per descrivere succintamente modelli di un sistema, possibilmente in modo composizionale
- Fornire una strumento di prototipazione del modello (se il linguaggio usato è eseguibile)
- Fornire supporto per un'analisi composizionale del modello
- Fornire supporto per un ragionamento equazionale

Supporto linguistico (1)

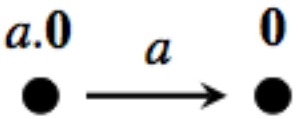
- Eccetto per sistemi relativamente piccoli, una rappresentazione grafica per mezzo di LTS è laboriosa; in effetti, sistemi reali hanno migliaia (o perfino milioni) di stati, il che rende difficile (se non impossibile) definirli e disegnarli.
- Necessità di una rappresentazione testuale (lineare) implicita per mezzo di un termine in un qualche linguaggio per descrivere sistemi concorrenti. (Pensa alle espressioni regolari vs linguaggi regolari)
- Semantica operazionale che associa un LTS ad ogni termine del linguaggio.
- Teorema di rappresentazione: ogni LTS (a stati finiti) ha un termine che lo rappresenta.

Supporto linguistico (2)

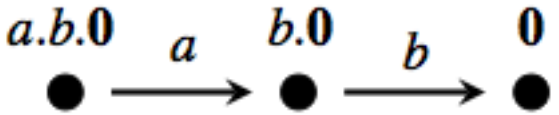
- Operatore **0** (detta nil , processo vuoto)
- Operatore di prefisso _._, ad esempio a.p (dove a è un'azione e p un processo): e.g., a.0
- Definizione di costante (nome di processo) A, permette comporta-menti ciclici e ricorsivi.

- Lts per nil

- Lts per a.0

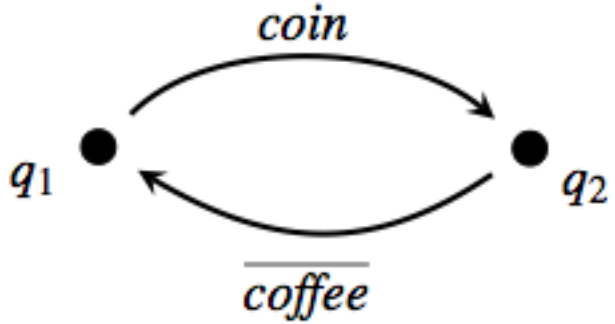


- Lts per a.b.0



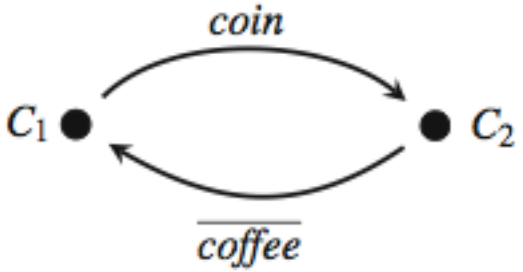
- Lts per A = a.A

Da LTS a termine, da termine a LTS



- C_1 e C_2 sono nomi di processi (costanti)
- L'operatore _._ è detto prefisso (sequenzialità)
- I due LTS sono isomorfi

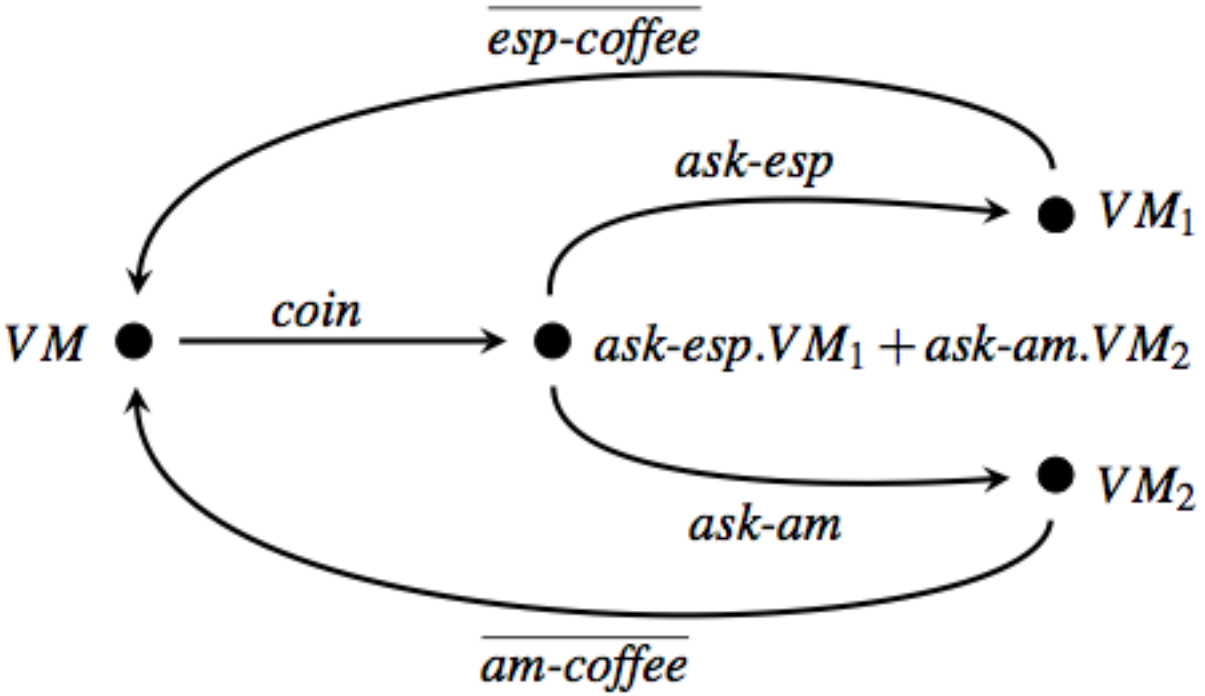
$$C_1 \stackrel{def}{=} coin.C_2 \text{ and } C_2 \stackrel{def}{=} \overline{coffee}.C_1$$



Supporto linguistico (3)

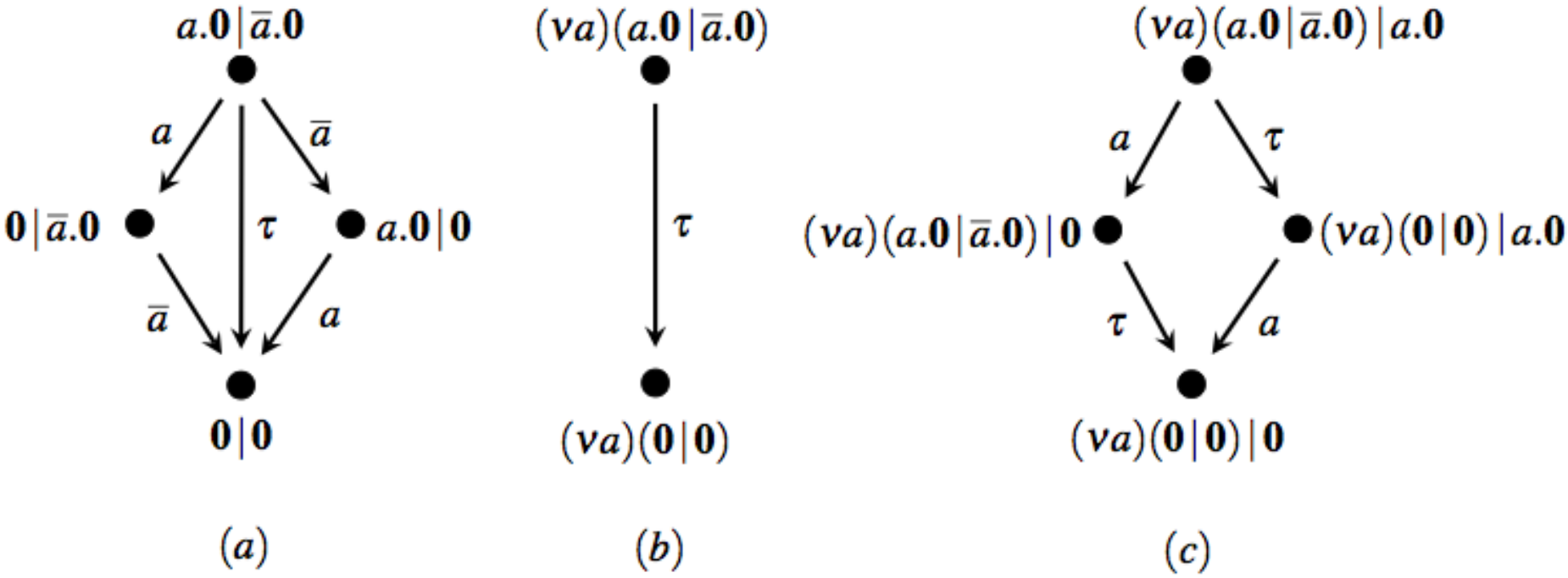
$$VM \stackrel{def}{=} coin.(ask\text{-}esp.VM_1 + ask\text{-}am.VM_2)$$

where $VM_1 \stackrel{def}{=} \overline{esp\text{-}coffee}.VM$ and $VM_2 \stackrel{def}{=} \overline{am\text{-}coffee}.VM$.



Linguaggio CCS

Oltre all'operatore nil, a quello di prefisso ., di scelta +, e le costanti, in CCS esistono gli operatori di composizione parallela | e di restrizione (va).



Semantica Operazionale Strutturata

(Pref) $\dfrac{}{\mu.p \xrightarrow{\mu} p}$ (Cons) $\dfrac{p \xrightarrow{\mu} p'}{C \xrightarrow{\mu} p'}$ $C \stackrel{def}{=} p$

(Sum_1) $\dfrac{p \xrightarrow{\mu} p'}{p+q \xrightarrow{\mu} p'}$ (Sum_2) $\dfrac{q \xrightarrow{\mu} q'}{p+q \xrightarrow{\mu} q'}$

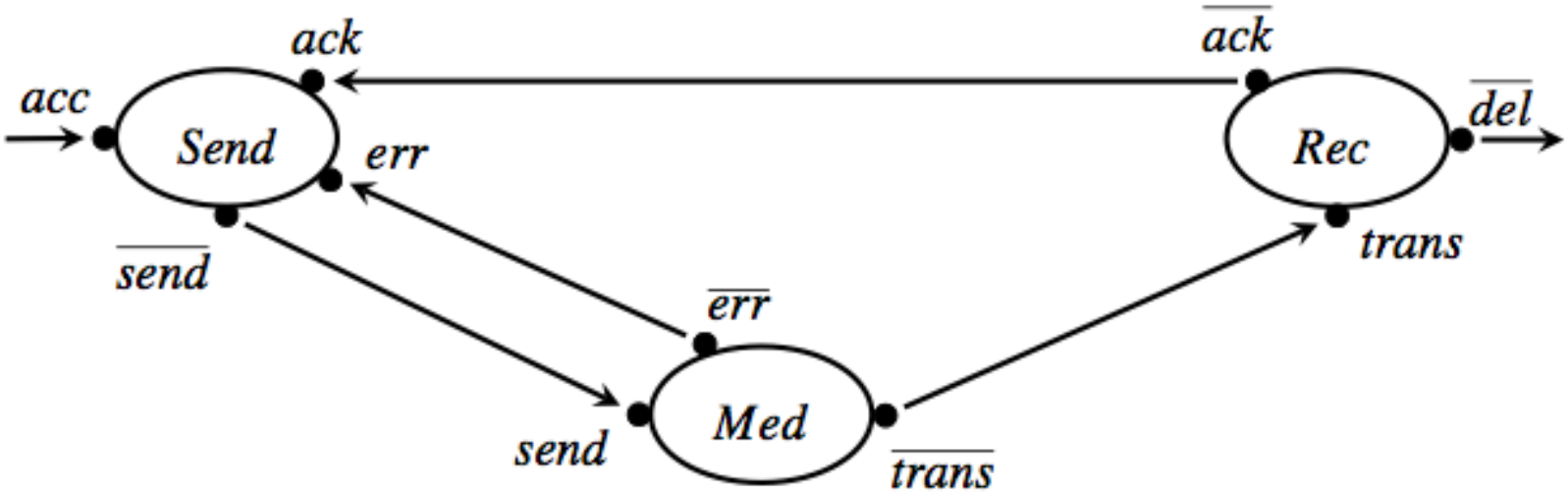
(Par_1) $\dfrac{p \xrightarrow{\mu} p'}{p \mid q \xrightarrow{\mu} p' \mid q}$ (Par_2) $\dfrac{q \xrightarrow{\mu} q'}{p \mid q \xrightarrow{\mu} p \mid q'}$

(Com) $\dfrac{p \xrightarrow{\alpha} p' \qquad q \xrightarrow{\overline{\alpha}} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$ (Res) $\dfrac{p \xrightarrow{\mu} p'}{(\nu a)p \xrightarrow{\mu} (\nu a)p'}$ $\mu \neq a, \bar{a}$

Table 3.1 Structural Operational Semantics: syntax-driven axiom and inference rules.

Un semplice protocollo (1)

Ci sono 3 componenti: un sender *Send* raccoglie un messaggio dall'ambiente sul port *acc* (accept) e lo inoltra ad un medium *Med*, che a sua volta lo inoltra a un receiver *Rec*, che lo manderà sul port *del* (deliver) all'ambiente esterno e poi manderà un messaggio di *ack* al sender, così che il ciclo può essere ripetuto.
Il medium *Med* può fallire: questo si astrae attraverso una transizione interna verso uno stato di errore *Err*; il quale *Err* chiederà a *Send* di rispedire il messaggio.
Qui sotto la rappresentazione a flow graph. Nel prossimo lucido il programma CCS.



Un semplice protocollo (2)

$$Protocol \stackrel{def}{=} (\nu send, error, trans, ack)((Send \mid Med) \mid Rec)$$

where the three finite-state CCS components are specified such:

$$Send \stackrel{def}{=} acc.Sending \qquad Med \stackrel{def}{=} send.Med'$$

$$Sending \stackrel{def}{=} \overline{send}.Wait \qquad Med' \stackrel{def}{=} \tau.Err + \overline{trans}.Med$$

$$Wait \stackrel{def}{=} ack.Send + error.Sending \qquad Err \stackrel{def}{=} \overline{error}.Med$$

$$Rec \stackrel{def}{=} trans.Del$$

$$Del \stackrel{def}{=} \overline{del}.Ack$$

$$Ack \stackrel{def}{=} \overline{ack}.Rec$$

Un semplice protocollo (3)

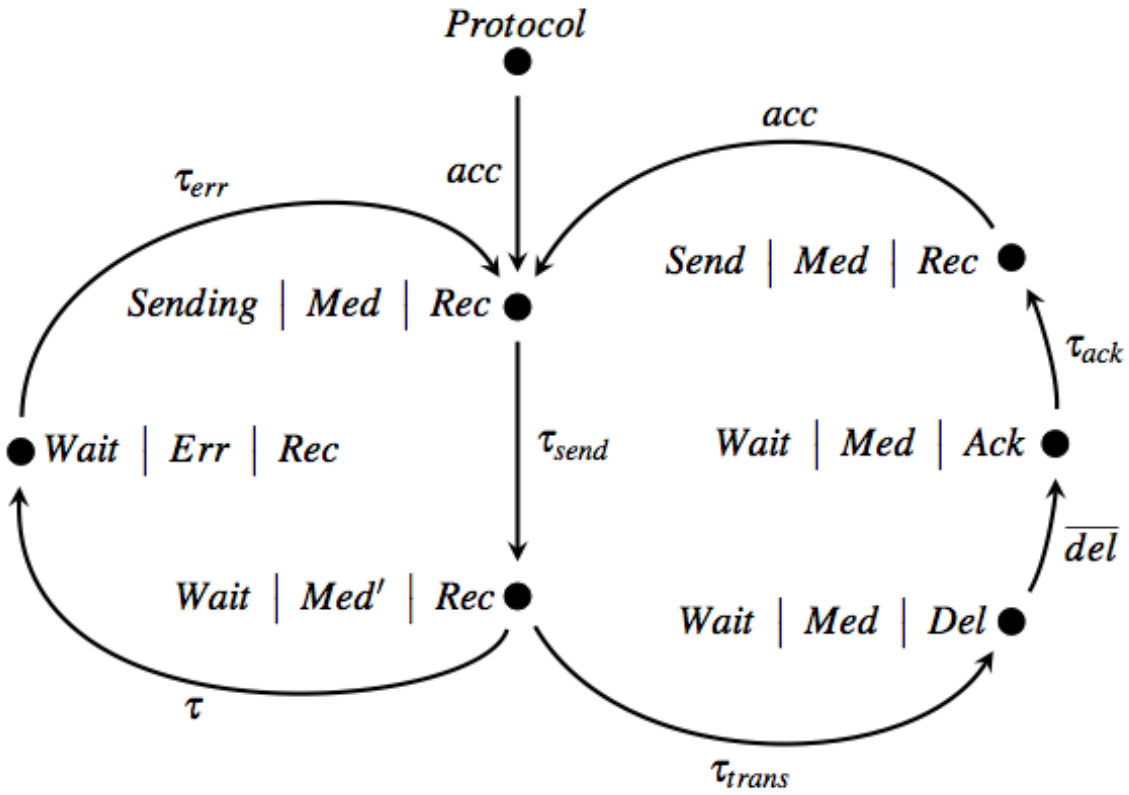


Fig. 3.8 The behaviour of the simple communication protocol.

Modellazione composizionale

Quando un sistema complesso deve essere modellato, può essere conveniente:

- identificare i sottocomponenti sequenziali del sistema, (ad esempio, *Send*, *Med*, *Rec* per il protocollo appena visto)
- modellare questi separatamente (specifiche CCS individuali), ed eventualmente analizzare loro proprietà locali (generando modelli LTS individuali),
- generare il modello completo (via semantica per il termine CCS complesso), eventualmente anche per composizione dei modelli LTS individuali, allo scopo di analizzare proprietà globali del sistema.

Prototipazione

- Una volta descritto il sistema in CCS, è possibile fare early-prototyping, cioè eseguirlo per fare simulazioni del comportamento (cioè semplici run di esecuzione) e testing sullo stesso (cioè vedere come il sistema si comporta quando sottoposto a certe interazioni), senza necessariamente costruirmi tutto lo spazio degli stati (che potrebbe anche essere infinito).
- Questo è utile per fare delle prime analisi, seppur sommarie, di correttezza del modello che stiamo definendo in CCS. (E.g., se una run del protocollo di prima andasse in deadlock, saprei già di aver commesso un errore)

Modelli e Sistemi Concorrenti

- Come modellare un sistema concorrente
- Quando due sistemi sono equivalenti
- Quali facilitazioni linguistiche sono disponibili
- Proprietà di congruenza e assiomatizzazioni
- Logiche di specifica di proprietà
- Strumenti automatici di verifica
- Espressività
- Reti di Petri

Cos'è una congruenza?

- Una relazione di equivalenza su un algebra di termini (come CCS, che usa operatori algebrici per costruire sistemi complessi a partire da sistemi più elementari) è una congruenza se è preservata dagli operatori (chiusa per contesti):
- Se $P \approx Q$, allora $a.P \approx a.Q$

  $P + R \approx Q + R$

  $P \mid R \approx Q \mid R$

  $(va)P \approx (va)Q$

  Se questo vale, allora l'equivalenza $\approx$ è detta congruenza comportamentale.

Perché è utile avere congruenze?

- Equivalenza ➔ Intercambiabilità
- Congruenza ➔ Intercambiabilità in ogni contesto!
- Un sistema complesso è composto da molte sottocomponenti; se una di queste si guasta, possiamo sostituirla con un'altra che sia "solo" equivalente, ma non congruente?

  Se $P \approx Q$, ma non è vero che $P \mid R \approx Q \mid R$ (cioè $\approx$ non è composizionale rispetto al parallelo), allora quando sostituiamo P con Q, il comportamento del nuovo sistema Q|R non è più lo stesso di P|R.

Congruenza ➔ Equivalence-checking composizionale

- Supponiamo di dover confrontare

$$p_1 \mid p_2 \quad \text{e} \quad q_1 \mid q_2$$

- Se dimostriamo che p_1 è congruente a q_1 e che p_2 è congruente a q_2, allora, per composizionalità, siamo sicuri che $p_1 \mid p_2$ è congruente a $q_1 \mid q_2$, risparmiando molto in termini di complessità.
- In generale, la congruenza è la base necessaria per avere la composizionalità.

Analisi composizionale – in generale

- Se una proprietà P è tale per cui se P vale per p_1 e per p_2 allora vale per $p_1 | p_2$, allora la proprietà P può essere verificata in modo composizionale. (E.g., P = "Il sistema termina sempre", oppure Q = "il sistema è sicuro"))
- In generale, assumendo che lo spazio degli stati di ogni p_i (per $1 \leq i \leq n$) sia composto da 10 stati, allora lo spazio degli stati per $p_1 \mid p_2 \mid \ldots \mid p_n$ è composto da 10^n stati: quindi, verificare la proprietà P sul sistema composto è esponenziale nel numero dei componenti. Con un ragionamento composizionale, invece, basterebbe fare, per n volte, la verifica di P su un sistema di 10 stati (complessità lineare).
- N.B. $p_1 | p_2$ potrebbe soddisfare la proprietà Q di sopra anche se p_1 o p_2 non la soddisfano. Allora, potremmo non avvantaggiarci della tecnica composizionale. (Per la proprietà P di sopra invece vale un "se e solo se" quindi è sufficiente applicare la tecnica composizionale)

Proprietà algebriche delle congruenze

- Spesso le congruenze comportamentali hanno associate alcune leggi intuitive. Ad esempio, per la bisimulazione valgono (anche) le seguenti:

$$p + (q + r) \sim (p + q) + r \qquad p \mid (q \mid r) \sim (p \mid q) \mid r$$

$$p + q \sim q + p \qquad p \mid q \sim q \mid p$$

$$p + \mathbf{0} \sim p \qquad p \mid \mathbf{0} \sim p$$

$$p + p \sim p$$

- Mentre non vale la seguente:

$$\mu.(p + q) \not\sim \mu.p + \mu.q$$

Ragionamento equazionale

- Definizione di teorie equazionali (dette assiomatizzazioni) che caratterizzano le congruenze.
- Fino ad ora, due sistemi potevano essere dimostrati congruenti attraverso una opportuna ispezione del loro spazio degli stati (equivalence-checking)
- Ma ora, dato che i due LTS sono in realtà termini di un linguaggio, la loro congruenza può essere dimostrata sintatticamente, mostrando che il primo termine può essere eguagliato al secondo per mezzo di una dimostrazione di deduzione equazionale.

Assiomi per bisimulation (T trace)

A1	Associativity	$x+(y+z)=(x+y)+z$
A2	Commutativity	$x+y=y+x$
A3	Identity	$x+\mathbf{0}=x$
A4	Idempotence	$x+x=x$
T	Distributivity	$\mu.(x+y)=\mu.x+\mu.y$

Table 4.2 Axioms for choice.

R1		$(\nu a)\mathbf{0}=\mathbf{0}$
R2	if $\mu \notin \{a,\bar{a}\}$	$(\nu a)\mu.x=\mu.(\nu a)x$
R3	if $\mu \in \{a,\bar{a}\}$	$(\nu a)\mu.x=\mathbf{0}$
R4		$(\nu a)(x+y)=(\nu a)x+(\nu a)y$
Exp	if $x=\Sigma_{i=1}^{n}\mu_i.x_i$ and $y=\Sigma_{j=1}^{m}\mu'_j.y_j$	
	$x\|y = \Sigma_i\mu_i.(x_i\|y) + \Sigma_j\mu'_j.(x\|y_j) + \Sigma_{i,j:\overline{\mu_i}=\mu'_j}\tau.(x_i\|y_j)$	

Table 4.3 Axioms for restriction and Expansion law.

Esempio di prova

$$\dfrac{\dfrac{\dfrac{}{x+y=y+x}}{b.w+a.\mathbf{0}=a.\mathbf{0}+b.w}\qquad\dfrac{}{x+(y+z)=(x+y)+z}}{a.\mathbf{0}+(b.w+a.\mathbf{0})=a.\mathbf{0}+(a.\mathbf{0}+b.w)}\qquad\dfrac{\dfrac{}{x+x=x}}{a.\mathbf{0}+a.\mathbf{0}=a.\mathbf{0}}$$

$$\dfrac{a.\mathbf{0}+(b.w+a.\mathbf{0})=(a.\mathbf{0}+a.\mathbf{0})+b.w}{(a.\mathbf{0}+a.\mathbf{0})+b.w=a.\mathbf{0}+b.w}$$

$$a.\mathbf{0}+(b.w+a.\mathbf{0})=a.\mathbf{0}+b.w$$

Modelli e Sistemi Concorrenti

- Come modellare un sistema concorrente
- Quando due sistemi sono equivalenti
- Quali facilitazioni linguistiche sono disponibili
- Proprietà di congruenza e assiomatizzazioni
- Logiche di specifica di proprietà
- Strumenti automatici di verifica
- Espressività
- Reti di Petri

Oltre all'equivalence-checking (1)

- Data una implementazione *Impl* (di solito scritta in CCS), per verificare se questa è corretta bisogna:
  - Definirsi una specifica *Spec*, di solito scritta anch'essa in CCS, che prescriva il comportamento astratto (corretto) che il sistema deve soddisfare.
  - Identificare una opportuna equivalenza $\approx$
  - Verificare che valga *Spec* $\approx$ *Impl*
- Ma non è sufficientemente flessibile per descrivere proprietà "puntuali/parziali" di un sistema. Ad esempio: "il sistema può fare subito l'azione a ma non l'azione b", oppure "comunque faccia a ora, poi può subito fare b", oppure anche "il sistema non può mai andare in deadlock".

Oltre all'equivalence-checking (2)

- Dato *Impl*, descritto ad esempio in CCS, per verificare se questa è corretta, uno potrebbe
  - definire un set di proprietà, descritte attraverso una qualche logica, che rappresenterebbe la specifica del sistema
  - Definire la relazione di soddisfacimento $\models$ $Impl \models F$ dove F è una proprietà nel set della specifica.
  - Verificare che la formula F sia davvero soddisfatta da *Impl*.
- Questo è il cosiddetto Model Checking

Hennessy-Milner Logic (HML)

- Logica modale: esprime “cosa può accadere ora”, ovvero possibilità <-> e necessità [-]
- [ask-esp]<esp-coffee>tt se chiedo espresso, potrò avere espresso
- [ask-am][esp-coffee]ff se chiedo am-coffee, non potrò avere espresso
- [coin](<ask-esp>tt $\wedge$ <ask-am>tt) se inserisco una moneta, potrò selezionare sia espresso che am-coffee
- Quest’ultima formula vale per la vending machine giusta, ma non per quella impolite, perché dopo aver inserito la moneta, non avrò entrambe le possibilità.

Hennessy-Milner Theorem

Sotto opportune condizioni, due processi P e Q sono bisimili se e soltanto se soddisfano le stesse formule della logica HML.

Relazione precisa fra equivalence-checking e model-checking.

HML con ricorsione

- HML esprime proprietà semplici, di possibilità e necessità (logica modale)
- Può essere utile avere logiche che esprimono proprietà temporali (logica temporale), del tipo:
  - "la vending machine non ruba mai i soldi" (safety property: something bad can never happen)
  - "la vending machine, dopo l'inserzione di una moneta, darà prima o poi una bevanda" (liveness property: something good will happen eventually)
- Necessità di parlare di minimi e massimi punti fissi (faremo una breve introduzione alla teoria del punto fisso).

Modelli e Sistemi Concorrenti

- Come modellare un sistema concorrente
- Quando due sistemi sono equivalenti
- Quali facilitazioni linguistiche sono disponibili
- Proprietà di congruenza e assiomatizzazioni
- Logiche di specifica di proprietà
- Strumenti automatici di verifica
- Espressività
- Reti di Petri

Uno strumento di verifica automatica: Il Concurrency Workbench (CWB)

- Il CWB è uno strumento automatico che può, tra l'altro:
  - Definire processi in (un dialetto di) CCS ed eseguire vari tipi di analisi su questi processi, quali analisi dello spazio degli stati, o fare equivalence-checking per varie equivalenze.
  - Definire proprietà in una logica modale potente (tipo HML con ricorsione) e verificare se un dato processo soddisfa tale proprietà (model checking)
  - Derivare automaticamente formule logiche che distinguono processi non equivalenti.
  - Simulare interattivamente il comportamento di un processo, guidando l'esecuzione attraverso il suo spazio degli stati in modo controllato.

CWB: difetti

- Strumento un po' datato (anni '90)
- Gira solo per piattaforma unix/linux
- Editor di comandi a linea (very old fashioned)
- Non offre visualizzazione dello spazio degli stati
- In grado di gestire spazi degli stati relativamente piccoli (alcune migliaia di stati), quindi casi di studio un po' limitati, ma sufficienti per i nostri scopi.

Guardare

http://homepages.inf.ed.ac.uk/perdita/cwb

per tutte le informazioni, sintassi, etc…

Caso di studio

- Modellazione, analisi e verifica di un semplice algoritmo di mutua esclusione (Peterson)
- Cosa vuol dire “garantire mutua esclusione” con equivalence-checking?
- Come si definisce la proprietà di mutua esclusione in HML?

Modelli e Sistemi Concorrenti

- Come modellare un sistema concorrente
- Quando due sistemi sono equivalenti
- Quali facilitazioni linguistiche sono disponibili
- Proprietà di congruenza e assiomatizzazioni
- Logiche di specifica di proprietà
- Strumenti automatici di verifica
- Espressività
- Reti di Petri

Espressività

- Cosa è in grado di calcolare un linguaggio?

Dipende dalla semantica comportamentale che si intende utilizzare:

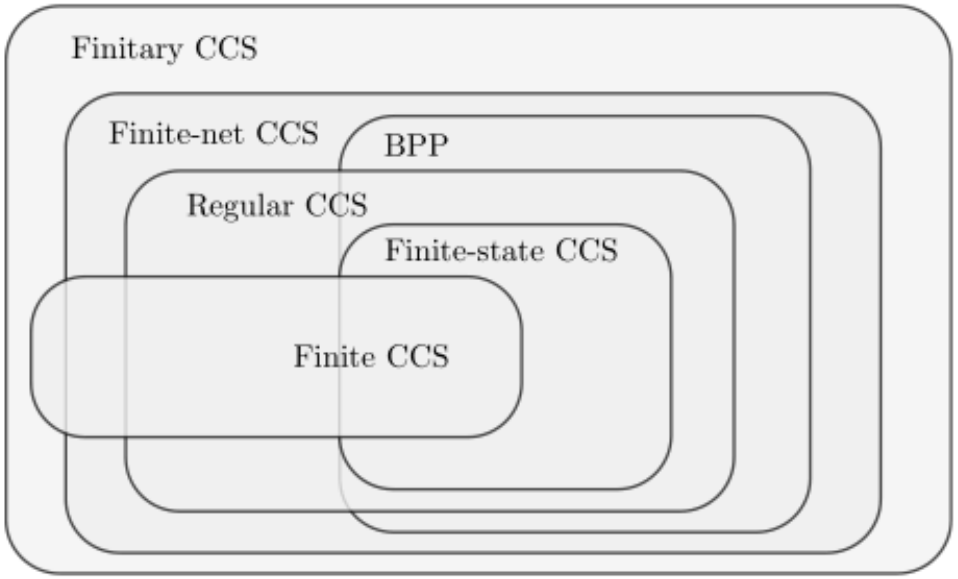
  - Semantica a tracce = classe di linguaggi
  - Semantica per isomorfismo = classe di sistemi di transizione

- Quali problemi è in grado di risolvere?
  - Calcola tutte le funzioni calcolabili? Cioè è Turing-completo?
  - Risolve certe classi di problemi in distributed computing con certi vincoli opportuni?

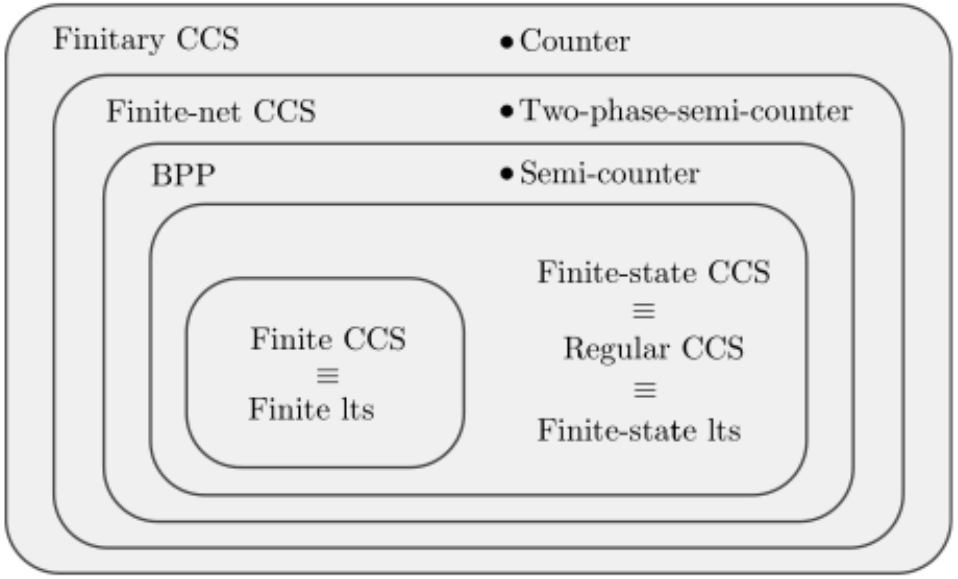
Confrontare il potere espressivo (1)

- Strumento: Teoria dei linguaggi formali
- Gerarchia di Chomsky: linguaggi regolari, liberi, contestuali, ricorsivamente enumerabili.
- Utile per confrontare linguaggi non Turing-completi. Lo useremo per confrontare sotto-calcoli di CCS che non sono Turing-completi (mentre CCS lo è)

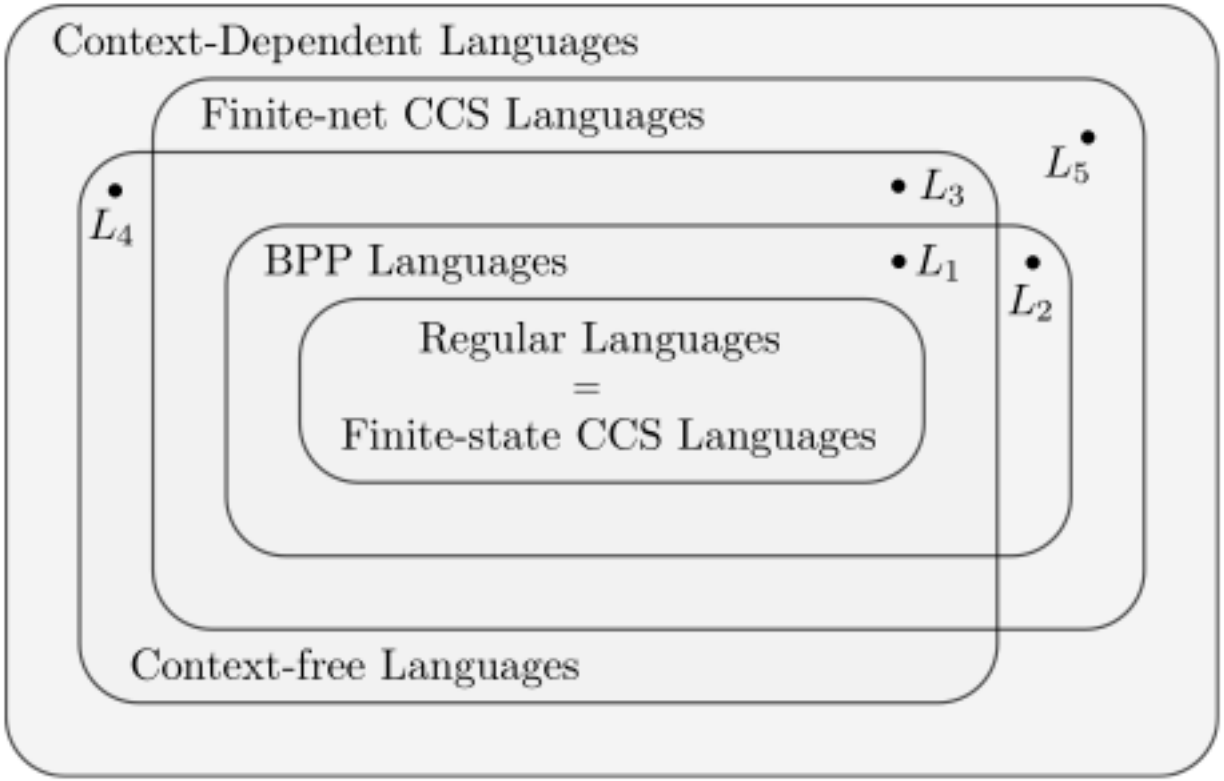
Syntax



Semantics



Relazioni con Gerarchia di Chomsky



Confrontare il potere espressivo (2)

- Strumento: Encoding
- Per dimostrare che due linguaggi Turing-completi sono equamente espressivi, si può definire un encoding (compilazione che preserva la semantica) da uno all'altro e viceversa.
- Utile per dimostrare che alcuni operatori aggiuntivi (scelta interna, hiding, composizione sequenziale) sono ridondanti per CCS, modulo semantica per bisimulazione.

Confrontare il potere espressivo (3)

- Strumento: utilizzare un problema di distributed computing
- Per dimostrare che due linguaggi (Turing-completi) NON sono equamente espressivi, si può individuare un problema per il quale uno offre una soluzione e l'altro no.
- In CCS non è possibile fornire una soluzione deadlock-free, divergence-free, symmetric, fully distributed del problema dei filosofi a cena. Questo è invece possibile farlo in Multi-CCS.

Modelli e Sistemi Concorrenti

- Come modellare un sistema concorrente
- Quando due sistemi sono equivalenti
- Quali facilitazioni linguistiche sono disponibili
- Proprietà di congruenza e assiomatizzazioni
- Logiche di specifica di proprietà
- Strumenti automatici di verifica
- Espressività
- Reti di Petri

Modello distribuito non-interleaving

- Stato globale

= multinsieme di stati locali

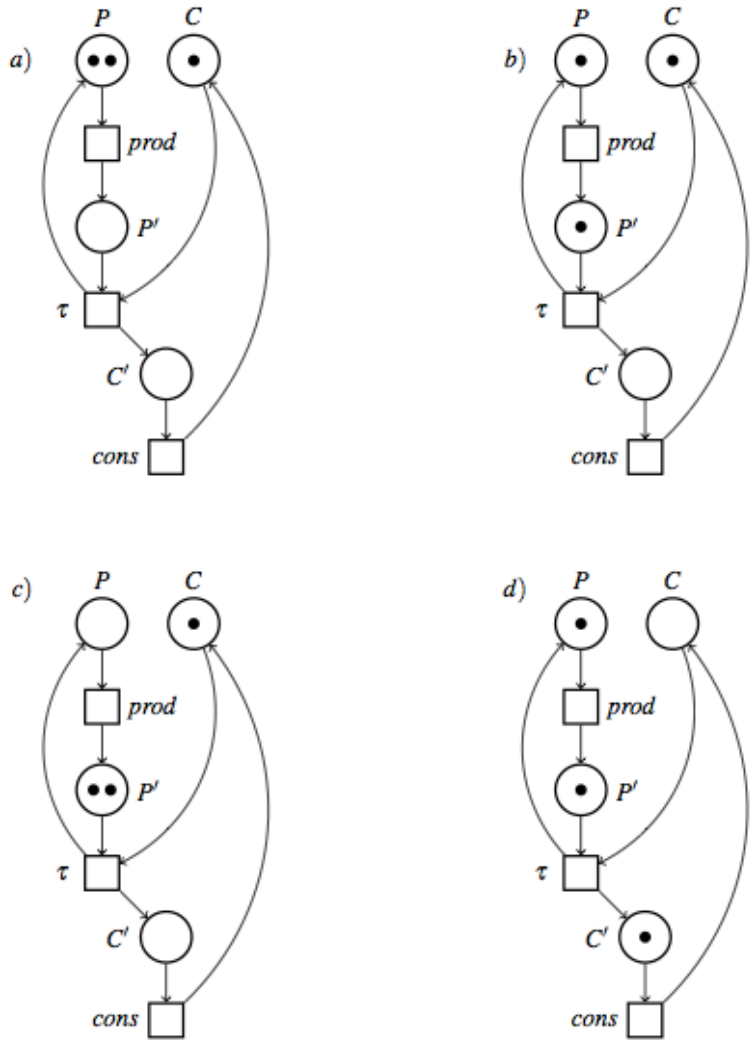
- Transizioni coinvolgono

solo qualche stato locale

- Token game
- Parallelismo esplicito

- Esempio: 2 produttori e

  un consumatore



Caratteristiche

- Equivalenze comportamentali “sequenziali”, ma anche “parallele” (caso limite: isomorfismo)
- Spazio degli stati globali che può essere infinito anche se la rete è finita.
- Proprietà decidibili (ad esempio, reachability)
- Nel caso base (detto Place/Transition) non sono Turing-complete
- Reti nonpermissive: Turing-complete

Linguaggi per rappresentare reti di Petri finite

- Gerarchia di classi di reti di Petri finite …
- e corrispondente gerarchia di linguaggi, ognuno in grado di rappresentare tutte e sole le reti di quella classe.
- Base per “computabilità distribuita”: un linguaggio è completo rispetto ad una classe di modelli se li rappresenta tutti a meno di isomorfismo (o di una opportuna equivalenza “distribuita”)

Un caso particolare: reti BPP

- Reti in cui le transizioni consumano un solo token
- Corrispondenza tra reti BPP e algebra di processi BPP (sotto-calcolo di CCS)
- Team bisimulation su BPP nets come generalizzazione della strong bisimulation per LTS: equivalenza distribuita (no state-explosion problem)
- Assiomatizzazione della team bisimulation equivalence.