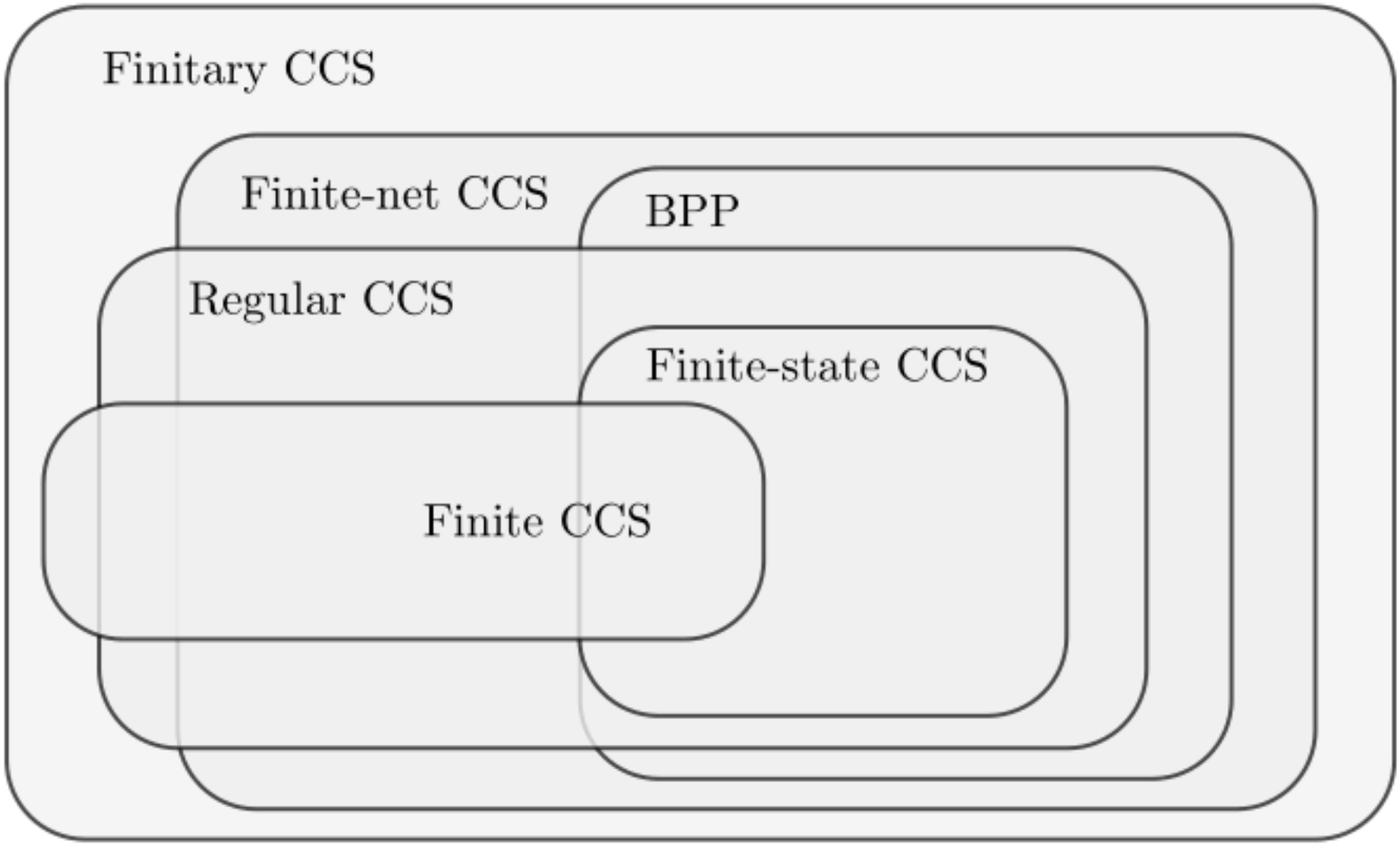
Lezione 11 MSC
CCS – sottoclassi di CCS
prima parte (di 2)

Roberto Gorrieri

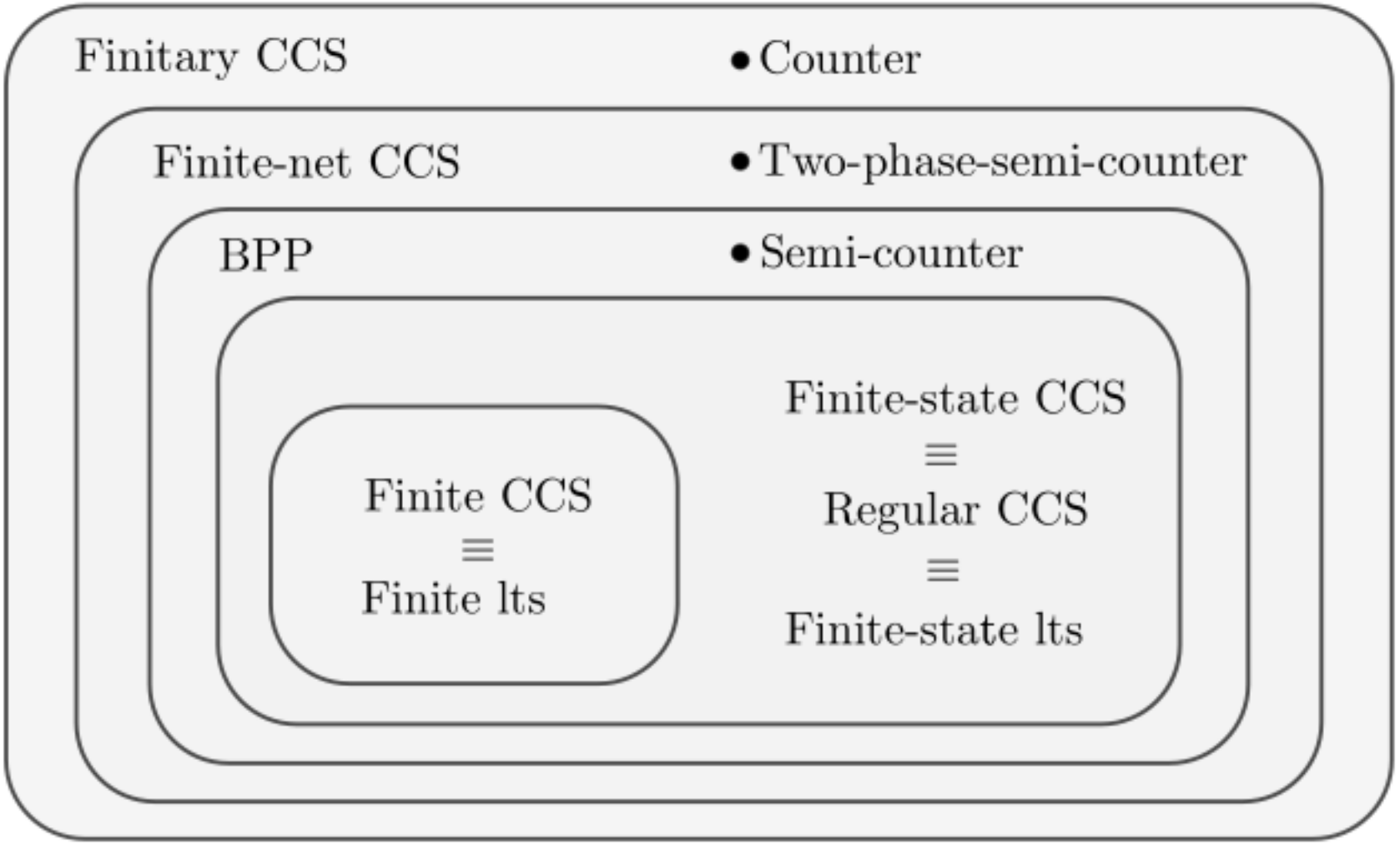
Gerarchia di sottolinguaggi

- Ponendo opportune restrizione sintattiche all'uso degli operatori di CCS, si ottengono un certo numero di sottolinguaggi, dando luogo ad una gerarchia "sintattica" di sotto-calcoli di CCS.
- Ognuno di questi sotto-calcoli ha un certo potere espressivo (cioè, capacità di rappresentare classi più o meno ampie di LTS, o classi diverse di linguaggi, o di risolvere problemi più complessi). Quindi una seconda gerarchia "semantica" delinea le relazioni semantiche tra questi sotto-linguaggi.

Syntax


Finitary CCS
Finite-net CCS
BPP
Regular CCS
Finite-state CCS
Finite CCS

Semantics


Finitary CCS
• Counter
Finite-net CCS
• Two-phase-semi-counter
BPP
• Semi-counter
Finite CCS
≡
Finite lts
Finite-state CCS
≡
Regular CCS
≡
Finite-state lts

Finite CCS

- Finite CCS è ottenuto proibendo l'uso delle costanti

$$p ::= \mathbf{0} \mid \mu.q \mid p+p$$
$$q ::= p \mid q|q \mid (\nu a)q$$

which can be more succinctly denoted with

$$p ::= \Sigma_{j\in J}\mu_j.p_j \mid p|p \mid (\nu a)p$$

- Limitato interesse da un punto di vista pratico (nessun comportamento ciclico o infinito)
- Utile per descrivere esempi semplici per discriminare tra semantiche (ad es. a.b + a.c vs a.(b + c))
- Per ragioni didattiche, ci restringeremo a processi finiti quando definiremo assiomatizzazioni di congruenze comportamentali

Finite CCS => finite LTS

Proposizione 3.5: Se p è in finite CCS, allora il suo LTS associato è finito (cioè aciclico, con un numero finito di stati e transizioni).

Dimostrazione: ovvio perché il linguaggio non offre nessun meccanismo di ricorsione/iterazione.

Si può offrire una prova più formale (nel prossimo lucido) usando la seguente definizione ausiliaria:

Exercise 3.30. (Size of a finite CCS process) The *size* of a finite CCS process p, denoted $size(p)$, is the number of prefixes occurring in p. Formally:

$$
\begin{aligned}
size(\mathbf{0}) &= 0 \\
size(\mu.p) &= 1 + size(p) \\
size(p_1 + p_2) &= size(p_1) + size(p_2) \\
size(p_1 \mid p_2) &= size(p_1) + size(p_2) \\
size((\nu a)p) &= size(p)
\end{aligned}
$$

Finite CCS => finite LTS(2)

Observe that, for any $p \in \mathscr{P}_{fin}$, $size(p)$ is a natural number and that if $size(p) = 0$, then $p \not\rightarrow$. Prove that, for any $p \in \mathscr{P}_{fin}$, if $p \xrightarrow{\mu} p'$, then $size(p') < size(p)$. This proof can be done by induction on the proof of transition $p \xrightarrow{\mu} p'$. This observation offers an alternative proof of Proposition 3.5: the transition relation must be acyclic because any path

$$q_1 \xrightarrow{\mu_1} q_2 \xrightarrow{\mu_2} \ldots q_n \xrightarrow{\mu_n} q_{n+1}$$

determines a descending chain $size(q_1) > size(q_2) > \ldots > size(q_n) > size(q_{n+1})$ that has 0 as lower bound. Hence, if $size(q_1) = k$, the path cannot be longer than k (actually, it may be shorter). (See also Exercise 4.9 on page 166 for the extension of function *size* to finitary CCS processes.) □

Esercizio (1)

- Ricava gli LTS associati alle seguenti coppie di processi (in finite CCS) e prova che sono strong bisimili:

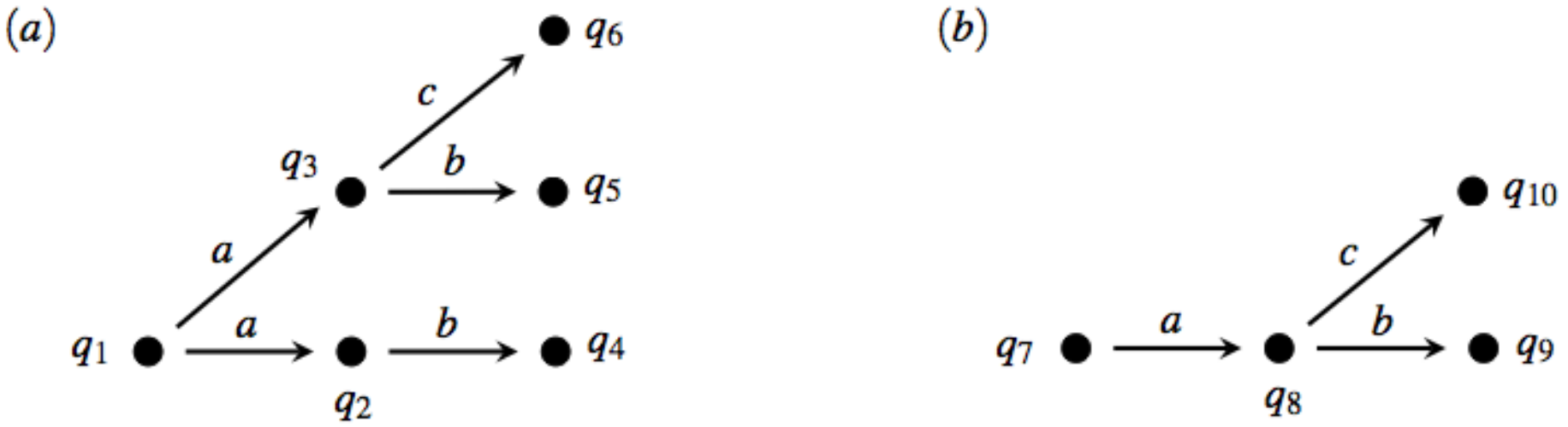
$$a.b + b.a \quad e \quad a \mid b$$

$$0 \quad e \quad (va)a.0$$

$$a.(vb)0 + a.0 \quad e \quad a.0 + a.0$$

Esercizio (2)

- Ricava due processi p e q di finite CCS, la cui semantica SOS genera due LTSisomorfi a questi e dimostra che non sono strong bisimili.



Representability theorem 1

- Ogni lts finito ridotto TS può essere denotato da un processo p di Finite CCS tale che l'lts associato a p via SOS è isomorfo a TS.

Exercise 3.29. (**Representability**) Generalize the exercises above, by showing that, for any reduced finite rooted lts TS, there exists a finite CCS process p such that the reachable lts $\mathscr{C}_p = (\mathscr{P}_p, sort(p), \rightarrow_p, p)$ is isomorphic to TS.

(*Hint:* Start from the deadlock states, each one represented by a process of the form $(\nu d)\mathbf{0}$ for some new name d, as exemplified in Exercise 3.25. A state q such that $T(q) = \{(q, \mu, q_k) \mid \exists \mu \in A, \exists q_k \in Q.\ q \xrightarrow{\mu} q_k\}$, which reaches only states q_k that are already represented by suitable CCS terms p_k, originates the finite CCS process $\Sigma_{(q,\mu,q_k)\in T(q)} \mu.p_k$.) □

Interleaving Model

- Anche se l'equivalenza di isomorfismo è la più concreta nozione di equivalenza che abbiamo definito, essa è anche già tanto astratta da non distinguere la concorrenza da sequenzialità (problema della scelta del modello)

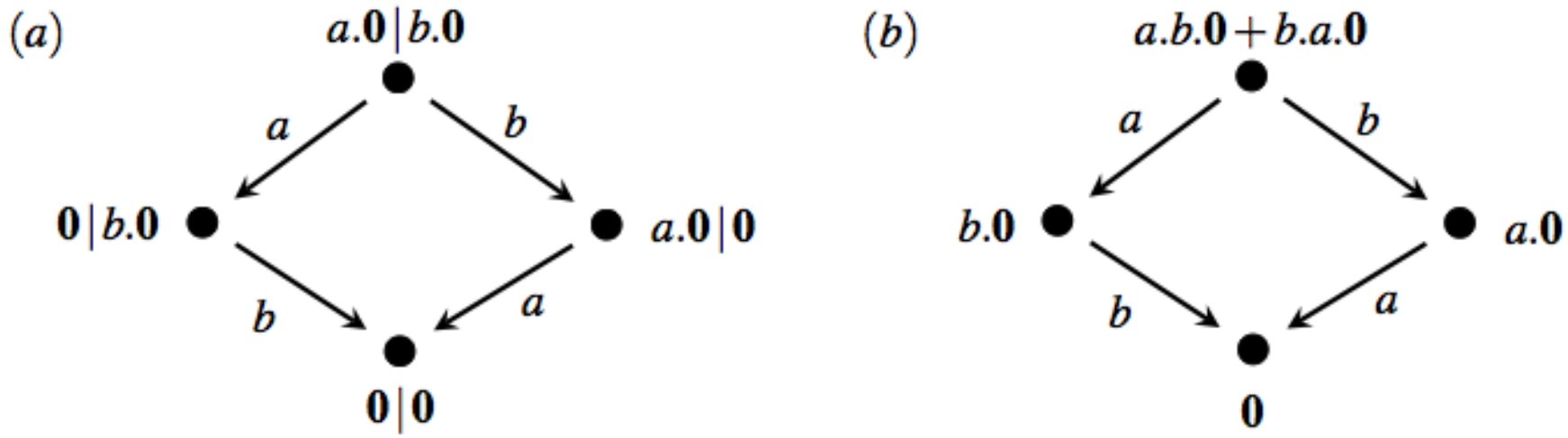


Fig. 2.9 Interleaving law: two isomorphic lts's

Modello distribuito: Petri nets

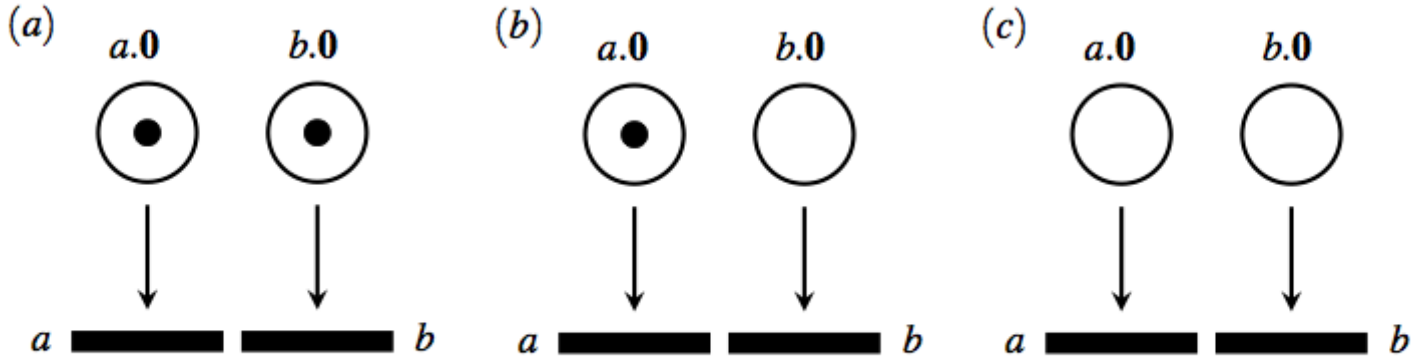


Fig. 7.2 The Petri net for $a \,|\, b$ in (a), after the firing of a in (b) and after the firing of both a and b in (c).

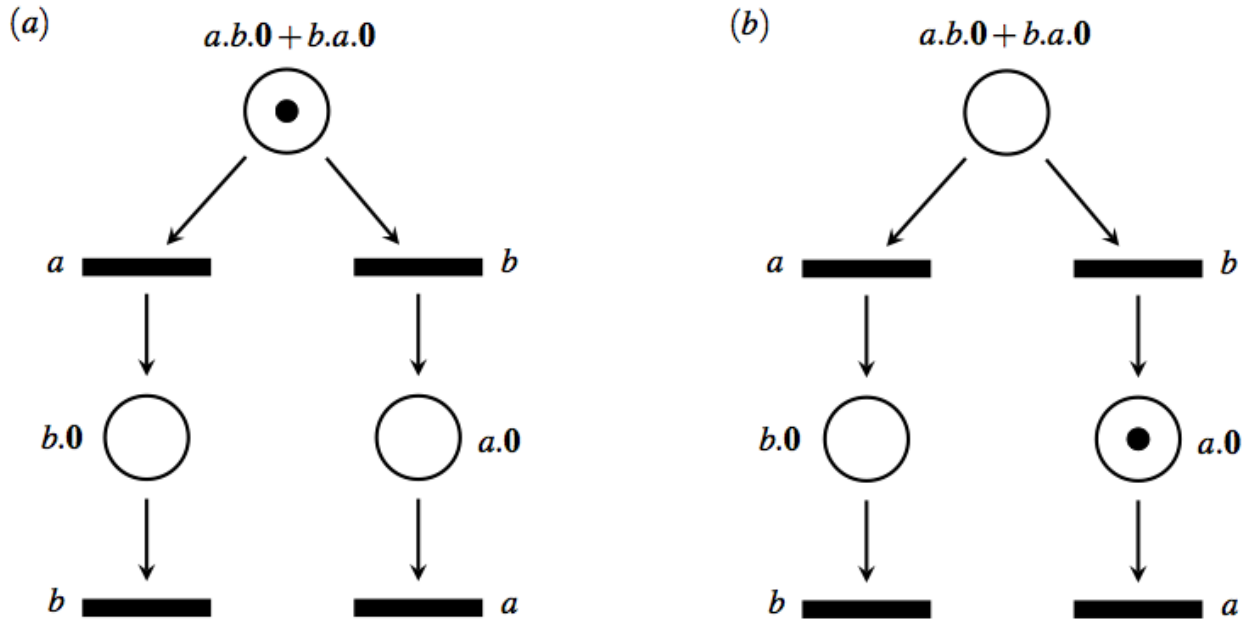


Fig. 7.3 The Petri net for $a.b + b.a$ in (a) and after the firing of b in (b).

Finite-state CCS

- Sintassi molto semplice:

  p ::= 0 | μ.q | p+p q ::= p | C

  dove si assume che le costanti C = q siano sempre definite e guardate, e che l'insieme Const(q) delle costanti usate da q sia finito.

- Sintassi alternativa:

$$p ::= \Sigma_{j \in J} \mu_j . p_j \mid C$$

Finite-state CCS => finite-state LTS

Theorem 3.1. (Finite number of reachable states) *For any finite-state CCS process p, the set $\mathscr{P}_p$ of its reachable states is finite.*

Proof. We define the number $\chi(p,\emptyset)$ as an upper-bound on the number of possible processes/states reachable from p:

$$\chi(\mathbf{0},I)=1 \quad \chi(p_1+p_2,I)=\chi(p_1,I)+\chi(p_2,I)+1 \quad \chi(\mu.p,I)=1+\chi(p,I)$$

$$\chi(A,I)=\begin{cases}0 & \text{if } A\in I,\\ 1+\chi(p,I\cup\{A\}) & \text{if } A\stackrel{def}{=}p \text{ and } A\notin I\end{cases}$$

As the set $Const(p)$ of process constants used in p is finite, it follows that the calculation of $\chi(p,\emptyset)$ always terminates (it cannot loop, because of its second argument recording the constants already met), returning a finite number. □

Corollary 3.2. (Finite-state lts) *For any finite-state CCS process p, the lts reachable from p, $\mathscr{C}_p=(\mathscr{P}_p,sort(p),\rightarrow_p,p)$, is finite-state.*

Proof. By Theorem 3.1, the set of reachable states $\mathscr{P}_p$ is finite. By Corollary 3.1 on page 99, $\mathscr{C}_p$ is finitely-branching. Hence, also sort(p) must be finite, as required by Definition 2.7 on page 25. □

Representability theorem 2

Theorem 3.2. (Representability) *For any reduced finite-state rooted lts TS, there exists a finite-state CCS process p such that the reachable lts* $\mathscr{C}_p = (\mathscr{P}_p, sort(p), \rightarrow_p, p)$ *is isomorphic to TS.*

Proof. Let $TS = (Q, A, \rightarrow_1, q_0)$, *with* $Q = \{q_0, q_1, \ldots, q_n\}$. *We define a process constant* C_i *in correspondence of state* q_i, *for* $i = 0, 1, , \ldots, n$, *defined as follows: if* q_i *is a deadlock, then* $C_i \stackrel{def}{=} \mathbf{0}$; *if* $T(q_i) = \{(q_i, \mu, q_k) \mid \exists \mu \in A, \exists q_k \in Q.\ q_i \xrightarrow{\mu}_1 q_k\}$, *then* $C_i = \Sigma_{(q_i,\mu,q_k)\in T(q_i)} \mu.C_k$. *Let us consider* $\mathscr{C}_{C_0} = (\mathscr{P}_{C_0}, sort(C_0), \rightarrow_2, C_0)$. *It is not difficult to see that* $\mathscr{P}_{C_0} = \{C_0, C_1 \ldots, C_n\}$ *because TS is reduced. Hence, the bijection we are looking for is* $f : Q \rightarrow \mathscr{P}_{C_0}$ *defined as* $f(q_i) = C_i$. *It is also immediate to observe that the two conditions of isomorphism are satisfied, namely:*

- $C_0 = f(q_0)$, *and*
- $q \xrightarrow{a}_1 q'$ *iff* $f(q) \xrightarrow{a}_2 f(q')$

Hence, f is indeed an lts isomorphism. □

Finite-state CCS languages are regular

Definition 3.6. (Finite-state CCS language) A language $L \subseteq (\mathscr{L} \cup \overline{\mathscr{L}})^*$ is a *finite-state CCS language* if there exists a finite state CCS process p such that the set of its weak completed traces is L, i.e., $WCTr(p) = L$. □

Proposition 3.6. (Finite-state CCS languages are regular languages) *The class of finite-state CCS languages coincides with the class of regular languages.*

Proof. By Remark 2.7, all, and only, the regular languages can be represented by finite-state LTSs, up to weak completed trace equivalence. By Corollary 3.2 and Theorem 3.2, finite-state CCS processes generate all, and only, finite-state LTSs. Hence, the class of finite-state CCS languages coincides with the class of regular languages. □

Esempio (1)

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

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

$$VM'' \stackrel{def}{=} coin.VM_1' \qquad VM_1' \stackrel{def}{=} ask\text{-}esp.VM_2' + ask\text{-}am.VM_3'$$
$$VM_2' \stackrel{def}{=} \overline{esp\text{-}coffee}.VM'' \qquad VM_3' \stackrel{def}{=} \overline{am\text{-}coffee}.VM''$$

Esercizio: Costruisci gli lts associati a VM e VM''. Dimostra che i due lts's sono isomorfi.

Esempio (2)

Example 3.4. **(Vending machines)** Let us consider the finite-state lts's for the two vending-machines discussed in Exercise 2.16. A possible finite-state process for case (i) is the following:

$$AVM \stackrel{def}{=} coin.(coin.ask\text{-}esp.\overline{esp\text{-}coffee}.AVM + ask\text{-}am.\overline{am\text{-}coffee}.AVM)$$

A possible finite-state process for the case (ii) is the following:

$$0VM \stackrel{def}{=} coin.1VM$$
$$1VM \stackrel{def}{=} ask\text{-}am.\overline{am\text{-}coffee}.0VM + coin.2VM$$
$$2VM \stackrel{def}{=} ask\text{-}esp.\overline{esp\text{-}coffee}.0VM + ask\text{-}am.\overline{am\text{-}coffee}.1VM$$

where the three constants $0VM, 1VM, 2VM$ stand for the number of coins collected by the machine. □

Esercizi

Exercise 3.31. Comment the expected behaviour of the following unfair vending machine:

$$0VM \stackrel{def}{=} coin.1VM + coin.0VM$$
$$1VM \stackrel{def}{=} ask\text{-}am.(\overline{am\text{-}coffee}.0VM + \tau.1VM) + coin.2VM$$
$$2VM \stackrel{def}{=} ask\text{-}esp.\overline{esp\text{-}coffee}.0VM + ask\text{-}am.(\overline{am\text{-}coffee}.1VM + \tau.2VM)$$

□

Exercise 3.32. Define a finite-state process for a vending machine that accepts in input coins of 1 or 2 euros, sells espresso for 2 euros and american coffee for 1 euro, keeps credit up to 4 euros (it refuses to input coins otherwise), does not steal money and does not allow for a beverage if the previously delivered one has not been collected. □

Exercise 3.33. Modify the vending machine $0VM$ of Example 3.4, so that it returns the coins inserted in case something goes wrong (e.g., no more water available), by modeling this situation by an internal transition to an erroneous state that handles the exception by returning the coins and entering a deadlock state. □

Regular CCS

- Finite-state CCS è sufficientemente espressivo, dato che può rappresentare tutti gli lts a stati finiti.
- Tuttavia, da un punto di vista modellistico, è carente: mancano gli operatori di parallelo e restrizione che sono molto utili quando uno vuole modellare un sistema complesso in modo composizionale.
- Regular CCS vuole risolvere questo problema, senza aumentare il potere espressivo del linguaggio, cioè regular CCS genera ancora lts a stati finiti pur offrendo gli operatori di parallelo e restrizione (in modo limitato).

Regular CCS - syntax

- I processi regular CCS sono generati da:

$$s ::= \mathbf{0} \mid \mu.t \mid s+s$$
$$t ::= s \mid C$$
$$p ::= t \mid (\nu a)p \mid p|p$$

- O, alternativamente, da

$$t ::= \Sigma_{j\in J}\mu_j.t_j \mid C$$
$$p ::= t \mid (\nu a)p \mid p|p$$

dove si assume che le costanti siano definite e guardate e l'insieme Const(p) sia finito.

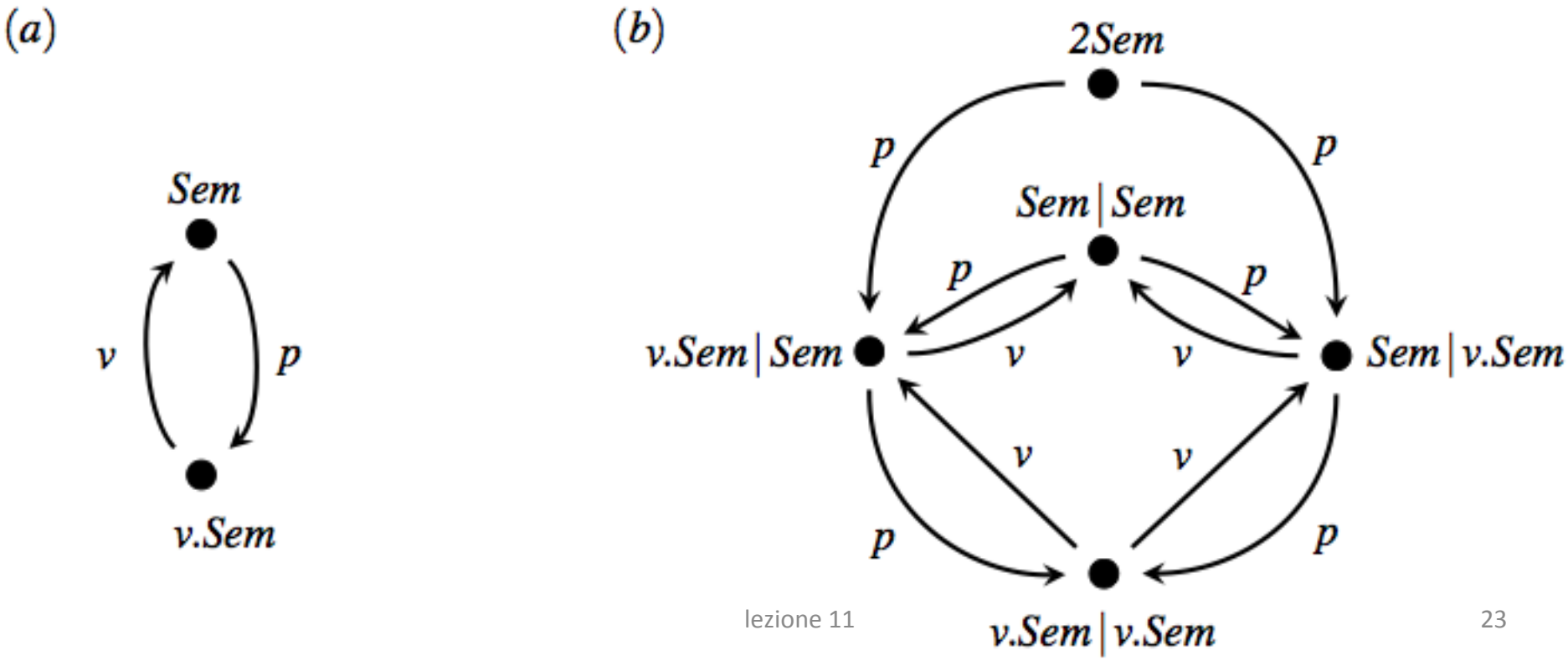
Osserva che C ha body in t, ovvero restrizione e parallelo non possono occorrere nel body di una costante! (l'ammettiamo, per comodità, solo se la costante è non ricorsiva)

Regular CCS => finite-state LTS

- Un processo di regular CCS p è ottenuto come composizione parallela (o restrizione) di alcuni processi finite-state CCS p_1, …p_n.
- Allora, ciascun p_i genera un lts a stati finiti.
- Come esercizio, abbiamo visto che se p_i ha k stati e p_j ha h stati, allora $p_i | p_j$ ha k*h stati.
- Come esercizio, abbiamo visto che se p_i ha k stati, allora (va)p_i ha al massimo k stati.
- Conseguenza: il processo regular CCS p deve avere un numero finito di stati.
- Dato che l'lts per p è finitely-branching (come conseguenza della guardatezza delle costanti), allora l'lts per p è proprio finite-state.

Esempio: Semaforo (1)

- Sem = p.v.Sem 2Sem = Sem | Sem

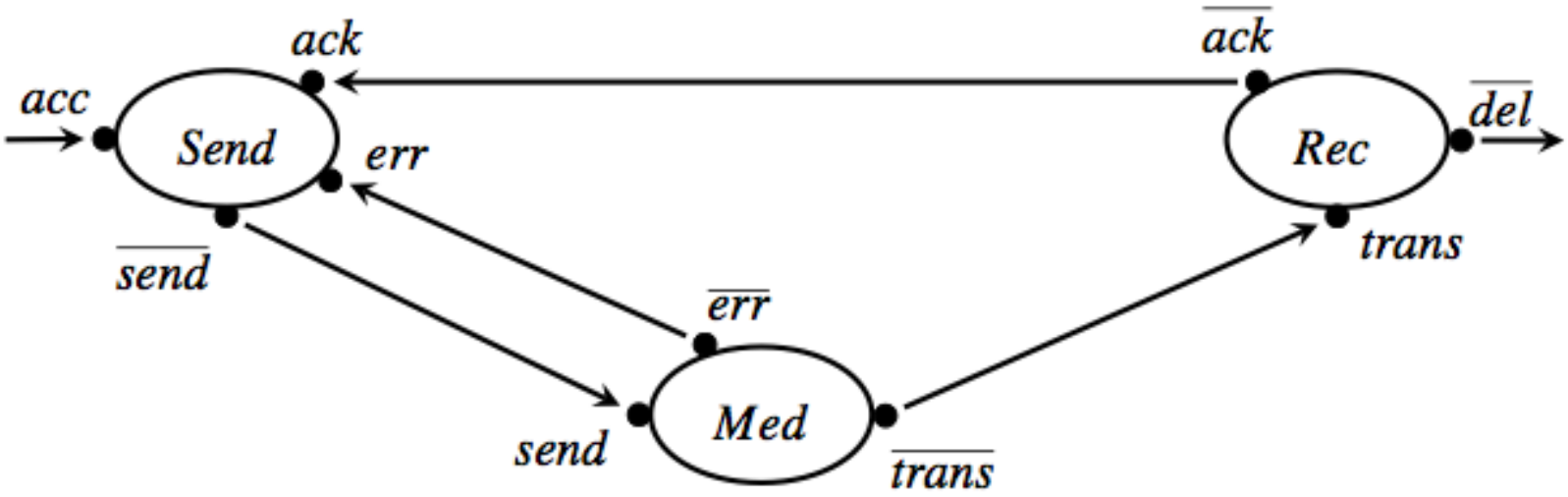


Esempio: Semaforo (2)

- nSem = $\prod_{1 \le n}$Sem ha 2^n+1 stati!
- Dimostrare che nSem ha una certa proprietà può essere impossibile per n molto grande.
- Ragionamento composizionale: vogliamo dimostrare che nSem è deadlock-free. Poiché la proprietà è composizionale (basta che uno dei componenti sia deadlock-free affinchè l'intero sistema sia deadlock-free), ci basta verificare che un singolo Sem (2 stati!) sia deadlock-free!

Un semplice protocollo (1): modellazione composizionale

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)

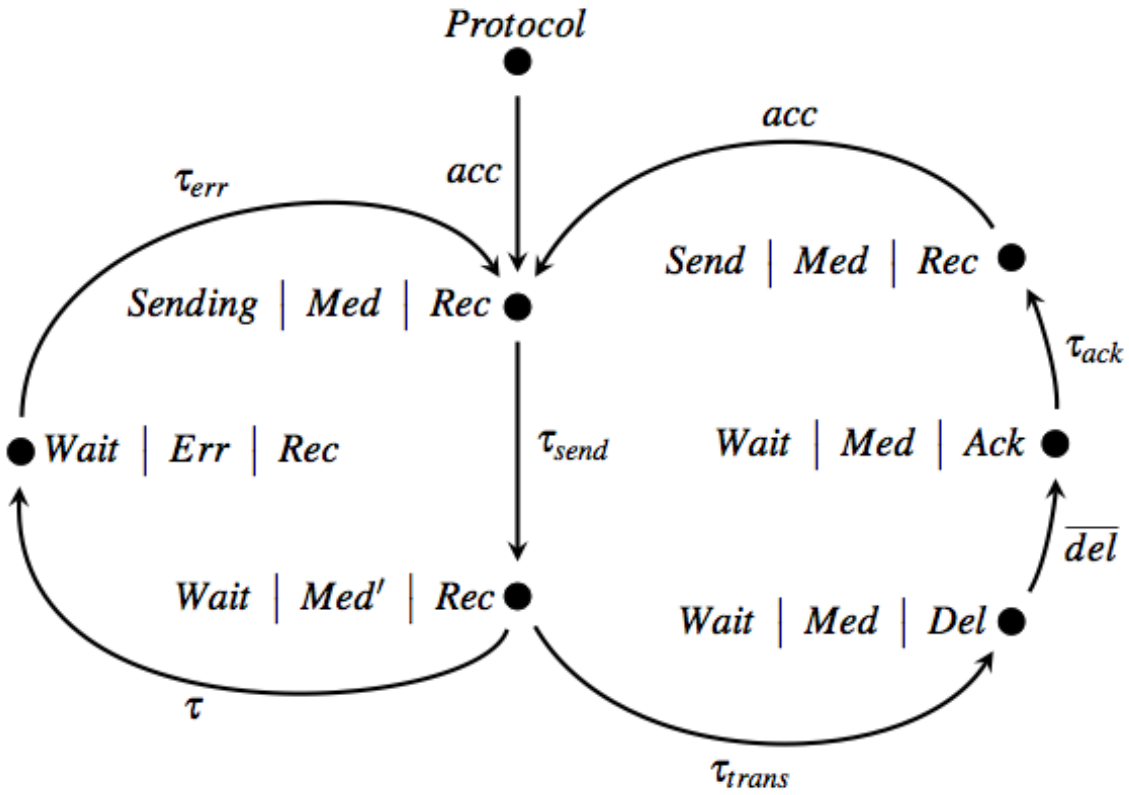


Fig. 3.8 The behaviour of the simple communication protocol.

Specifica ed equivalence checking

- Specifica sequenziale:

ProtSpec = acc.’del.ProtSpec

- Implementazione distribuita:

Protocol quello del lucido precedente

- Equivalence checking:

ProtSpec ≈ Protocol vale perché R è una weak bisimulation

$$R = \{(Protocol, ProtSpec), (Sending \,|\, Med \,|\, Rec, \overline{del}.ProtSpec), (Wait \,|\, Med' \,|\, Rec, \overline{del}.ProtSpec), (Wait \,|\, Err \,|\, Rec, \overline{del}.ProtSpec), (Wait \,|\, Med \,|\, Del, \overline{del}.ProtSpec), (Wait \,|\, Med \,|\, Ack, ProtSpec), (Send \,|\, Med \,|\, Rec, ProtSpec)\}$$

Two-position buffer (1)

- Specifica sequenziale:

B0 = in.B1 B1 = in.B2 + ‘out.B0 B2 = ‘out.B1

- Implementazione parallela (non rispetta l’ordine)

B = in.B’ B’ = ‘out.B B|B è l’implementazione

- Equivalence checking: B0 ~ B|B

Esercizio: costruire gli lts per B0 e per B|B.
Individuare una strong bisimulation contenente la coppia (B0, B|B)

Two-position buffer (2)

- Specifica sequenziale:

B0 = in.B1 B1 = in.B2 + ‘out.B0 B2 = ‘out.B1

- Implementazione pipeline (rispetta l’ordine)

Buff = (v d)(Buf1 | Buf2)

Buf1 = in.’d.Buf1 Buf2 = d.’out.Buf2

- Equivalence checking: B0 ≈ Buff
- Esercizio: costruire l’lts per Buff e individuare una weak bisimulation contenente la coppia (B0, Buff)
- Osservazione: poiché non si trasmettono valori, non si vede una vera differenza tra le due implementazioni (anche la specifica è ambigua).

2-Producers-1-Consumer

- 2PC = (v send)((P | P) | C)

  P = produce.'send.P C = send.consume.C
- Osserva che la restrizione su send garantisce che solo il consumer C può consumare i beni prodotti dai due producers P.
- Esercizio: disegna l'lts per 2PC (9 stati e 18 transizioni). Può andare in deadlock?
- Esercizio: quante azioni produce possono essere fatte prima che una consume diventi obbligata?
- Esercizio: prova a disegnare (se ci riesci!) l'lts per il sistema che ha 3 produttori e 2 consumatori.

Producer-Consumer with a one-position buffer

- PBC = (v in out)((P1|B)|C1)

  P1 = produce.'in.P1 C1 = out.consume.C1

  B = in.B' B' = 'out.B

- 2PC = (v send)((P | P) | C)

  P = produce.'send.P C = send.consume.C

Domanda: è vero che PBC ~ 2PC? NO. La traccia strong "produce produce" c'è solo per 2PC.

Domanda: è vero che PBC ≈ 2PC? SI.

Producer-Consumer with a two-position buffer

- P2BC = (v in out)((P1|B0)|C1)

  P1 = produce.'in.P1 C1 = out.consume.C1

  B0 = in.B1 B1 = in.B2 + 'out.B0 B2 = 'out.B1
- PBC = (v in out)((P1|B)|C1)

Domanda: è vero che PBC ≈ P2BC? NO: conta le produce …

- P2BC' = (v in out)((P1|(B|B))|C1)

Domanda: è vero che P2BC ~ P2BC'? SI anche per congruenza di ~

- P2BC'' = (v in out)((P1|Buff)|C1)

Buff = (v d)(Buf1 | Buf2) Buf1 = in.'d.Buf1 Buf2 = d.'out.Buf2

Domanda: è vero che P2BC ≈ P2BC''? SI anche per congruenza di ≈

Buffer a n posizioni (1)

- Specifica sequenziale

$$B_0 \stackrel{def}{=} in.B_1$$
$$B_i \stackrel{def}{=} in.B_{i+1} + \overline{out}.B_{i-1} \quad \text{for } 0 < i < n$$
$$B_n \stackrel{def}{=} \overline{out}.B_{n-1}$$

- Implementazione parallela

$$B^n = \prod_{1 \le i \le n} B$$

dove B = in.B’ B’ = ‘out.B

Buffer a n posizioni (2)

- Strong bisimulation up to ~ (sfruttando associatività e commutatitivtà del parallelo)

$$R = \{(B_k, (\Pi_{i=1}^{k} \overline{out}.B) \,|\, B^{n-k}) \;\mid\; 0 \leq k \leq n\}$$

First, observe that for $k = 0$, the pair in R is $(B_0, \mathbf{0} \,|\, B^n)$. If R is a bisimulation up to $\sim$, then $B_0 \sim \mathbf{0} \,|\, B^n$. As $\mathbf{0} \,|\, B^n \sim B^n$, by transitivity we get $B_0 \sim B^n$. Now, we will prove that R is indeed a strong bisimulation up to $\sim$, i.e., we will prove that:

1. if $B_k \xrightarrow{\alpha} p$ for some action α and some process p, then there are some q and q' such that $(\Pi_{i=1}^{k} \overline{out}.B) \,|\, B^{n-k} \xrightarrow{\alpha} q$ with $q' \sim q$ and $(p, q') \in R$ (this is enough for application of the up-to technique, as $\sim$ is reflexive, hence $p \sim p$). And, symmetrically,
2. if $(\Pi_{i=1}^{k} \overline{out}.B) \,|\, B^{n-k} \xrightarrow{\alpha} q$ for some action α and some process q, then there are some processes q' and p such that $q \sim q'$, $B_k \xrightarrow{\alpha} p$ and $(p, q') \in R$.

Buffer a n posizioni (3)

(Case 1) If $k < n$, process B_k can do $B_k \xrightarrow{in} B_{k+1}$ and $(\Pi_{i=1}^{k} \overline{out}.B) \,|\, B^{n-k}$ can respond with *in* reaching $(\Pi_{i=1}^{k} \overline{out}.B) \,|\, \overline{out}.B \,|\, B^{n-(k+1)}$, which is strongly bisimilar to $(\Pi_{i=1}^{k+1} \overline{out}.B) \,|\, B^{n-(k+1)}$ and the pair $(B_{k+1}, (\Pi_{i=1}^{k+1} \overline{out}.B) \,|\, B^{n-(k+1)})$ is in R.

If $k > 0$, B_k can also do $B_k \xrightarrow{\overline{out}} B_{k-1}$ and $(\Pi_{i=1}^{k} \overline{out}.B) \,|\, B^{n-k}$ can respond by reaching $(\Pi_{i=1}^{k-1} \overline{out}.B \,|\, B) \,|\, B^{n-k}$, which is strongly bisimilar to $(\Pi_{i=1}^{k-1} \overline{out}.B) \,|\, B^{n-(k-1)}$ and the pair $(B_{k-1}, (\Pi_{i=1}^{k-1} \overline{out}.B) \,|\, B^{n-(k-1)})$ is in R.

(Case 2) Besides the *in*-labeled transition already considered in the previous case, $(\Pi_{i=1}^{k} \overline{out}.B) \,|\, B^{n-k}$ has other $n-(k-1)$ *in*-labeled transitions, all reaching states that are strongly bisimilar to $(\Pi_{i=1}^{k+1} \overline{out}.B) \,|\, B^{n-(k+1)}$ by associativity and commutativity of parallel composition. For instance, one of these state is $\Pi_{i=1}^{k} \overline{out}.B \,|\, B^{j} \,|\, \overline{out}.B \,|\, B^{n-(k+j+1)}$ for $1 \leq j,k \leq n$, $j+k < n$. To any of these transitions, B_k responds with $B_k \xrightarrow{in} B_{k+1}$ and $(B_{k+1}, (\Pi_{i=1}^{k+1} \overline{out}.B) \,|\, B^{n-(k+1)})$ is in R.

Similarly, if $k > 0$, besides the $\overline{out}$-labeled transition discussed above, process $(\Pi_{i=1}^{k} \overline{out}.B) \,|\, B^{n-k}$ has other $k-1$ $\overline{out}$-labeled transitions, all reaching states that are strongly bisimilar to $(\Pi_{i=1}^{k-1} \overline{out}.B) \,|\, B^{n-(k-1)}$. To any of these, B_k may respond with $B_k \xrightarrow{\overline{out}} B_{k-1}$ and $(B_{k-1}, (\Pi_{i=1}^{k-1} \overline{out}.B) \,|\, B^{n-(k-1)})$ is in R. And this completes the proof. □

Linking

- Se vogliamo collegare il port a di p al port b di q, possiamo farlo con l'operatore derivato di linking o pipelining

$$p \frown q = (\nu d)(p\{d/a\} \mid q\{d/b\})$$

dove d è un nome non in uso né da p né da q.

- Esempio: mettere in pipeline due one-position buffers

$$Buf \stackrel{def}{=} B \frown B, \text{ where } B \stackrel{def}{=} in.\overline{out}.B$$

$$Buf \stackrel{def}{=} (\nu d)(B\{d/out\} \mid B\{d/in\})$$

$$B_{\{d/out\}} \stackrel{def}{=} in.\overline{d}.B_{\{d/out\}} \text{ and } B_{\{d/in\}} \stackrel{def}{=} d.\overline{out}.B_{\{d/in\}}$$

Buffer a n posizioni (4)

- Specifica sequenziale

$$B_0 \stackrel{def}{=} in.B_1$$

$$B_i \stackrel{def}{=} in.B_{i+1} + \overline{out}.B_{i-1} \quad \text{for } 0 < i < n$$

$$B_n \stackrel{def}{=} \overline{out}.B_{n-1}$$

Implementazione pipeline?

Some notation first. PB is used to denote either term B or term $\overline{out}.B$. Term $PB_\pi^{(n,k)}$ is used to denote the term

$$PB_1 \frown PB_2 \frown \ldots \frown PB_n$$

where the linking operator applied to two terms PB_i and PB_{i+1} (for $i = 1, \ldots n-1$) connects the *out* of PB_i to the *in* of PB_{i+1}, and where π is a binary vector of length n with only k elements set to 1: $\pi(i) = 1$ if PB_i is $\overline{out}.B$, $\pi(i) = 0$ if PB_i is B. If $\pi(i) = 1$ and $\pi(i+1) = 0$, for $0 \leq i < n$, we denote by $\pi[i+1/i]$ the vector $(\pi(1), \ldots, \pi(i-1), 0, 1, \pi(i+2), \ldots, \pi(n))$. Note that $PB_0^{(n,0)}$ – where the subscript 0 stands for a vector of 0's – is composed only of B components, as well as $PB_1^{(n,n)}$ – where the subscript 1 stands for a vector of 1's – is composed only of $\overline{out}.B$ components.

Buffer a n posizioni (5)

- Esiste una weak bisimulation? SI, ma non facile ….
  (ovviamente si assume che il linking sia associativo)

$$R = \{(B_k, PB_\pi^{(n,k)}) \mid 0 \leq k \leq n \text{ and } \pi \text{ is of length } n \text{ with } k \text{ elements set to } 1\}$$

- Bisogna dimostrare che valgono proprietà accessorie:

$\overline{out}.B \frown B \xrightarrow{\tau} B \frown \overline{out}.B$ perciò

$PB_\pi^{(n,k)} \xrightarrow{\tau} PB_{\pi[i+1/i]}^{(n,k)}$ 0 < k < n, supponendo che i e i+1 siano gli indici dei due processi contigui coinvolti

(R_1) if $n \geq 1$, $PB_1^{(n,n)} \frown B \stackrel{\tau}{\Longrightarrow} B \frown PB_1^{(n,n)}$;

(R_2) similarly, if $n \geq 1$, $\overline{out}.B \frown PB_0^{(n,0)} \stackrel{\tau}{\Longrightarrow} PB_0^{(n,0)} \frown \overline{out}.B$.