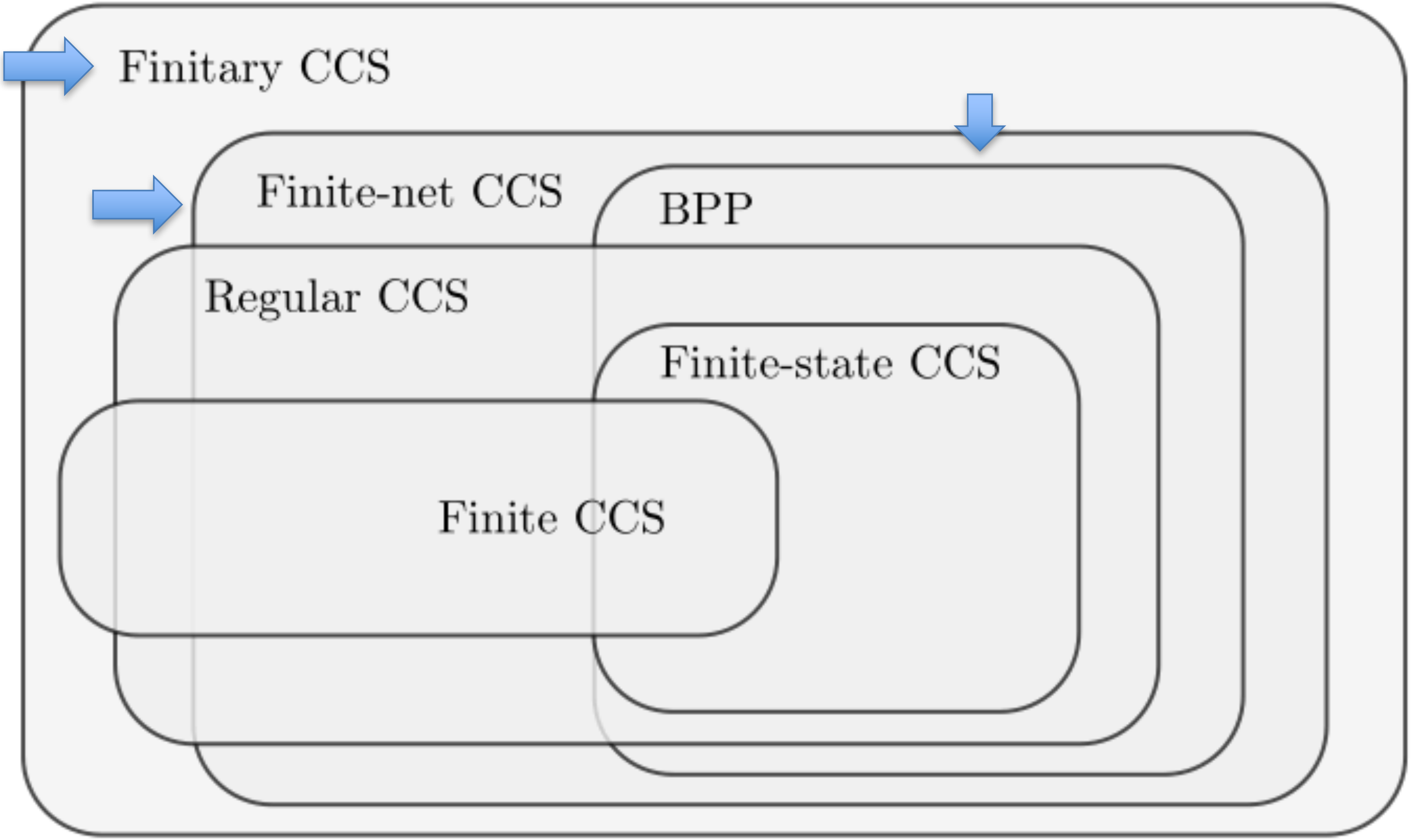
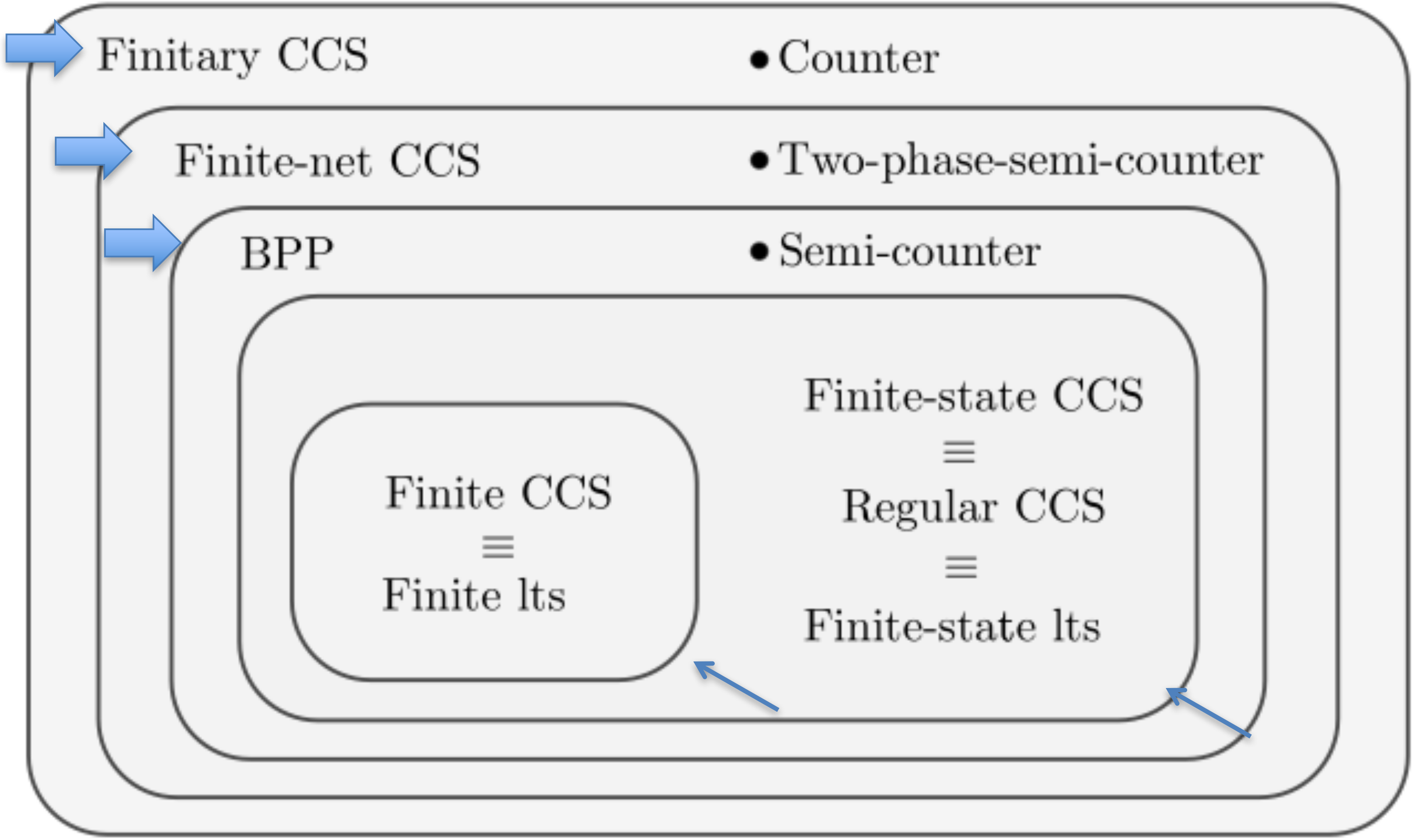
Lezione 12 MSC
CCS – sottoclassi di CCS seconda parte (di 2)

Roberto Gorrieri

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-state CCS
≡
Regular CCS
≡
Finite-state lts
Finite CCS
≡
Finite lts

BPP: Basic Parallel Processes

- BPP è una estensione sintattica di finite-state CCS, che permette di usare anche il parallelo

$$s ::= \mathbf{0} \mid \mu.p \mid s+s$$
$$p ::= s \mid p|p \mid C$$

- Si assume che il parallelo non consenta la sincronizzazione (puro asincrono)
- Come sempre, le costanti sono definite e guardate e per ogni p, Const(p) è finito.
- Sintassi compatta:

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

BPP - osservazioni

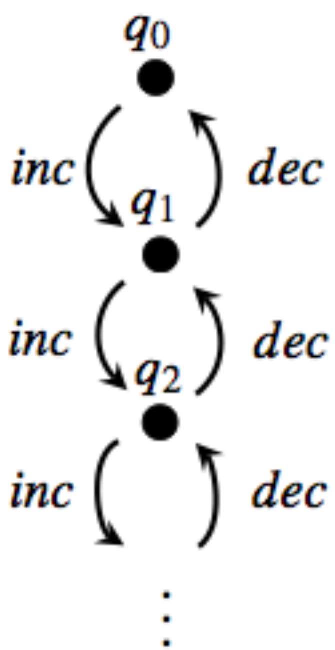
- Sintatticamente: BPP è una superclasse di finite-state CCS, ma non contiene regular CCS perché
  - La restrizione può essere usata da regular CCS
  - Il parallelo può essere usato in definizioni di costanti per BPP (ad es. C = a.0 | b.C)
- Semanticamente: BPP è una superclasse di finite-state CCS/regular CCS perché esprime tutti gli lts finiti e, in aggiunta, molti lts infiniti.

Semi-counter in full CCS

- Definizione in (full) CCS con infinite costanti:

$$SCount_0 \stackrel{def}{=} inc.SCount_1$$

$$SCount_n \stackrel{def}{=} inc.SCount_{n+1} + dec.SCount_{n-1} \quad n > 0$$



- $SCount_i$ non può essere equivalente a $Scount_{i+1}$ perché solo il secondo può eseguire la traccia composta da i+1 occorrenze consecutive dell'azione dec; perciò, possiamo concludere che nessun processo finite-state CCS può essere equivalente a SCount0.

Semi-counter in BPP

- Usando una sola costante, possiamo definire il processo BPP

$$SC \stackrel{def}{=} inc.(SC \mid dec.\mathbf{0})$$

- che non genera un lts isomorfo a quello del lucido precedente, ma si può dimostrare che SC e $SCount_0$ sono strong bisimili.
- Esercizio: prova a disegnare un frammento iniziale dell'lts per SC.

SC vs SCount$_0$ (1)

Now we prove that $SCount_0$ and SC are strongly bisimilar. Consider the relation

$$R = \{(SCount_n, SC \mid \Pi_{i=1}^{n} dec.\mathbf{0}) \mid n \geq 0\}.$$

It is not difficult to see that it is a strong bisimulation up to $\sim$, where we take advantage of the fact that parallel composition is associative[16], commutative, with $\mathbf{0}$ as neutral element, with respect to strong bisimilarity $\sim$. This is proved in Section 4.1.1, Proposition 4.2, on page 162. Moreover, we are also using the fact that $\sim$ is a congruence for parallel composition, i.e., if $p \sim q$, then $p \mid r \sim q \mid r$ for all r; this is proved in Theorem 4.1 on page 178. (For concrete details, see Example 4.2 on page 163.)

First, observe that for $n = 0$, the pair in R is $(SCount_0, SC \mid \mathbf{0})$. If R is a bisimulation up to $\sim$, then $SCount_0 \sim SC \mid \mathbf{0}$. As $SC \mid \mathbf{0} \sim SC$, by transitivity we get our expected result: $SCount_0 \sim SC$. Now, let us prove that R is indeed a strong bisimulation up to $\sim$.

SC vs $SCount_0$ (2)

One direction:

Assume that $SCount_n \xrightarrow{\alpha} q$. Then *either* $\alpha = inc$ and $q = SCount_{n+1}$, *or* $n > 0$, $\alpha = dec$ and $q = SCount_{n-1}$. In the former case, the matching transition is

$$SC \mid \Pi_{i=1}^{n} dec.\mathbf{0} \xrightarrow{inc} (SC \mid dec.\mathbf{0}) \mid \Pi_{i=1}^{n} dec.\mathbf{0}$$

(where the reached state is bisimilar to $SC \mid \Pi_{i=1}^{n+1} dec.\mathbf{0}$) and the pair $(SCount_{n+1}, SC \mid \Pi_{i=1}^{n+1} dec.\mathbf{0}) \in R$.

In the latter case, (one of) the matching *dec* transition(s) starts from $SC \mid \Pi_{i=1}^{n} dec.\mathbf{0}$ and reaches $(SC \mid \Pi_{i=1}^{n-1} dec.\mathbf{0}) \mid \mathbf{0}$, which is strongly bisimilar to $SC \mid \Pi_{i=1}^{n-1} dec.\mathbf{0}$, and the pair $(SCount_{n-1}, SC \mid \Pi_{i=1}^{n-1} dec.\mathbf{0}) \in R$.

SC vs SCount$_0$ (3)

The other direction:

Assume now $SC \,|\, \Pi_{i=1}^{n} dec.\mathbf{0} \xrightarrow{\alpha} p$. Then, by inspecting the rules for parallel composition:

1. *Either* $SC \xrightarrow{inc} SC \,|\, dec.\mathbf{0}$ and thus $\alpha = inc$ and process $p = (SC \,|\, dec.\mathbf{0}) \,|\, \Pi_{i=1}^{n} dec.\mathbf{0}$ (which is bisimilar to $SC \,|\, \Pi_{i=1}^{n+1} dec.\mathbf{0}$). In such a case, the matching transition is $SCount_n \xrightarrow{inc} SCount_{n+1}$, and the pair $(SCount_{n+1}, SC \,|\, \Pi_{i=1}^{n+1} dec.\mathbf{0})$ is in R.
2. *Or* $n > 0$, $\alpha = dec$ and p is one of the following three processes: $(SC \,|\, \mathbf{0}) \,|\, \Pi_{i=1}^{n-1} dec.\mathbf{0}$ or $(SC \,|\, \Pi_{i=1}^{n-1} dec.\mathbf{0}) \,|\, \mathbf{0}$ or $(((SC \,|\, \Pi_{i=1}^{n-k} dec.\mathbf{0}) \,|\, \mathbf{0}) \,|\, \Pi_{i=1}^{k-1} dec.\mathbf{0})$ for some $1 \leq k < n$.
   In any case, p is strongly bisimilar to $SC \,|\, \Pi_{i=1}^{n-1} dec.\mathbf{0}$. The matching transition is $SCount_n \xrightarrow{dec} SCount_{n-1}$ and the pair $(SCount_{n-1}, SC \,|\, \Pi_{i=1}^{n-1} dec.\mathbf{0})$ is in R.

And this completes the proof. So, we have shown that a semi-counter can be represented, modulo $\sim$, by a simple BPP process. □

BPP languages

- *L* è un *BPP language* se esiste un processo BPP *p* tale che *WCTr*(*p*) = *L*.
- Tutti i linguaggi regolari sono esprimibili da BPP
- Un processo BBP può generare un linguaggio context-free.
- Le tracce complete di $A \overset{def}{=} a.(A \mid b.\mathbf{0}) + c.\mathbf{0}$,

  non costituiscono un linguaggio regolare, perché se interseco questo linguaggio con a*cb* (che è regolare) non ottengo un linguaggio regolare:

  $CTr(A) \cap a^*cb^* = \{a^k cb^k \mid k \geq 0\}$

- Vedremo che non tutti i linguaggi context-free sono inclusi nei linguaggi BPP.

BPP languages (2)

- Un processo BPP può generare linguaggi non context-free.
- Il processo BPP $B \stackrel{def}{=} a.(B \,|\, b.\mathbf{0}) + c.(B \,|\, d.\mathbf{0}) + e.\mathbf{0}$

è tale che CTr(B) non può essere context-free. Infatti, se fosse libero, allora anche l'intersezione con il linguaggio regolare a*c*b*d*e dovrebbe essere libera; ma tale linguaggio è

$$CTr(B) \cap a^*c^*b^*d^*e = \{a^k c^n b^k d^n e \mid k,n \geq 0\}$$

che è un tipico esempio di linguaggio context-dependent

BPP – bisimulation equivalence is decidable

- Il problema di decidere l'equivalenza per bisimulazione forte su processi BPP è stato dimostrato decidibile (PSPACE-complete)
- L'equivalenza per bisimulazione è l'unica equivalenza decidibile su una classe di sistemi a stati infiniti.
- Per weak bisimilarity ci sono al momento solo risultati parziali (ad esempio, confronto fra processi BPP e processi finite-state CCS)
- Vale decidibilità di branching bisimilarity for normed BPP processes (cioè processi BPP che possono sempre terminare)

Finite-net CCS

- Sintatticamente: Finite-net CCS è una estensione di BPP (permette di usare anche la restrizione, seppur non nel corpo di una costante, e la sincronizzazione) ma non di regular CCS (permette di usare il parallelo nel corpo di una costante, ma regular permette di fare mix di parallelo e restrizione)

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

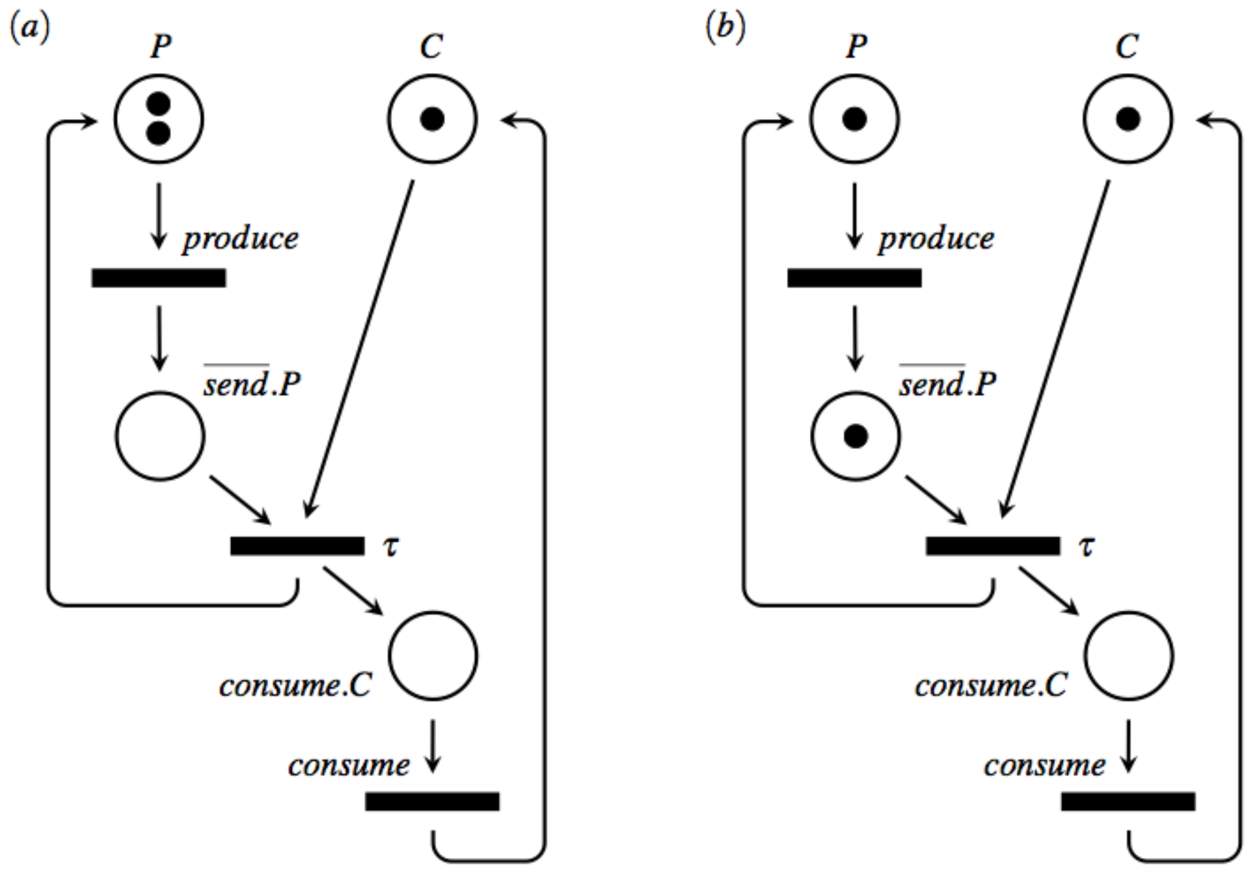
Alternativamente

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

Finite-net CCS (2)

- Semanticamente: Finite-net CCS è una estensione di BPP, nel senso che ci sono processi finite-net CCS che non possono essere rappresentati attraverso un BPP.
- Interesse:
  - ad ogni processo di Finite-net CCS si può associare una rete di Petri finita
  - ad ogni rete di Petri finita (le cui transizioni hanno al massimo 2 archi in entrata) si può associare un processo finite-net CCS (Representability theorem)

Esempio di rete di Petri



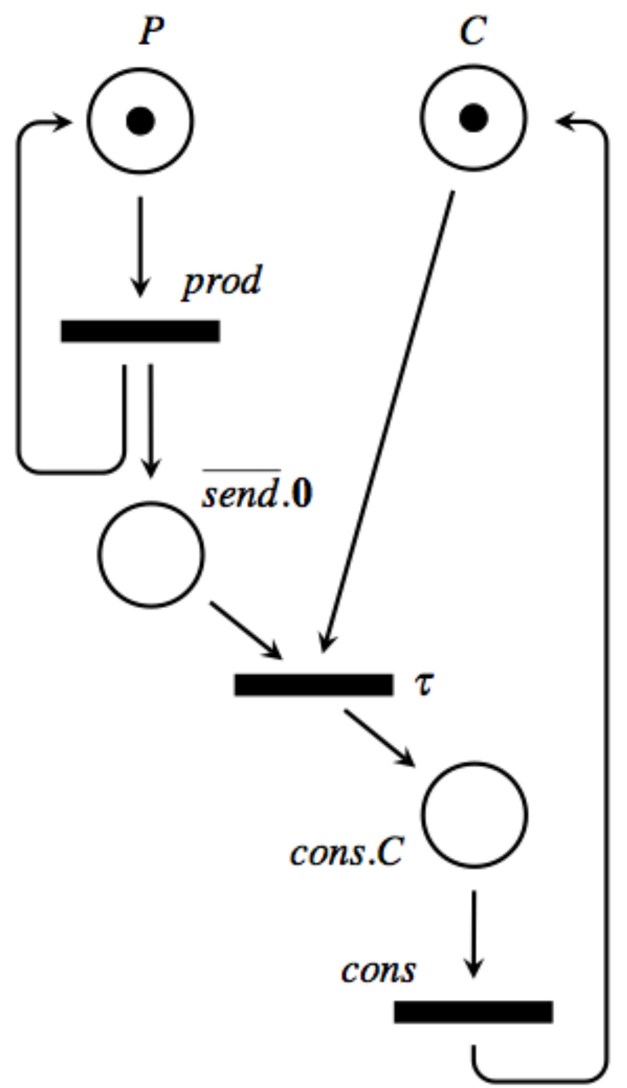
2PC = (*ν* send) ((P|P)|C)

P = produce.’send.P C = send.consume.C

Esempio di rete di Petri (2)

- UPC = (ν send)(P|C)
- P = prod.(P|'send.0)
- C = send.cons.C

- UPC è un processo di finite-net CCS
- Osserva che sul posto 'send.0 si possono accumulare un numero illimitato di tokens



Finite-net CCS (3)

- Proprietà decidibili di Finite-net CCS via semantica a reti di Petri:
  - Reachability: dati p e q finite-net CCS processes, posso verificare se q è raggiungibile da p.
  - Strong bisimilarity tra un finite-net CCS process q ed un finite-state CCS process p.
  - Strong regularity: dato un finite-net CCS proccess q, posso verificare se esiste un finite-state CCS process p ad esso strong bisimile.
  - Bisimulation equivalence è però in generale indecidibile fra due processi finite-net p e q (perché così è sulle reti di Petri finite)

Unbounded Producer/Consumer

- Produttore illimitato -- BPP

  Pr = produce.(Pr | ‘send.0)
- Consumatore – finite-state

  C = send.consume.C
- Unbounded Producer/Consumer -- finite-net CCS

  UPC = (ν send)(Pr | C)
- Esercizio: dato un buffer illimitato UB = in.(UB|’out.0) dimostra che UPC è weak bisimile a

  PUBC = (ν in out)(P1 | UB) | C1   con

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

Esempio (1)

- Two-phase-Semi-counter: prima solo inc, poi solo dec (tante quante)

$$
\begin{aligned}
2PSC &\stackrel{def}{=} (\nu d)INC \\
INC &\stackrel{def}{=} inc.(INC \,|\, d.dec.\mathbf{0}) + \tau.DEC \\
DEC &\stackrel{def}{=} \overline{d}.DEC
\end{aligned}
$$

- Le tracce weak complete sono tutte della forma $inc^n dec^n$ per $n \geq 0$
- Si può dimostrare che non è possibile trovare un processo BPP weak trace equivalent a 2PSC.

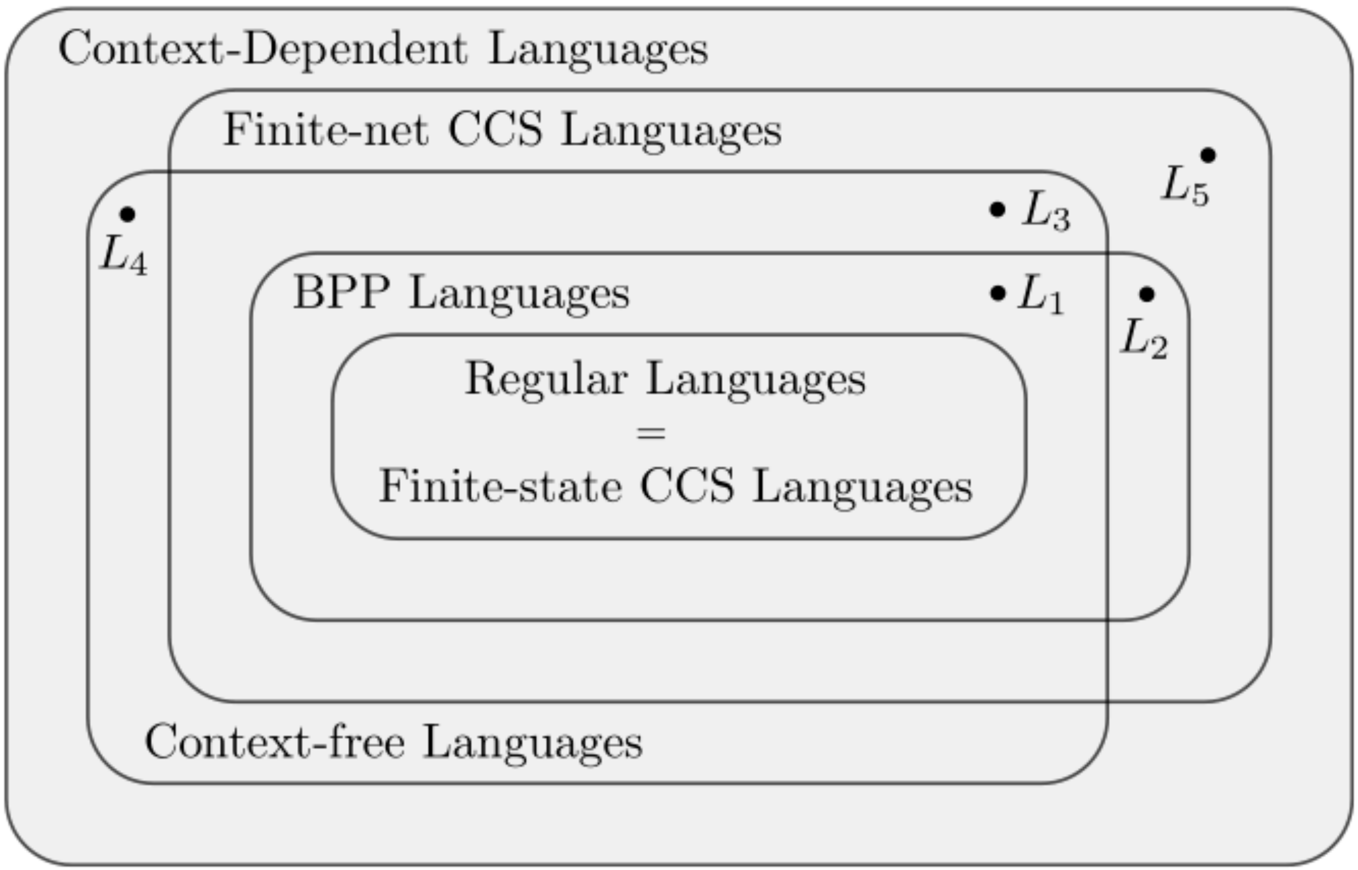
Esempio (2)

- Linguaggio $a^n b^m c^m$ $m \leq n$

$$
\begin{array}{ll}
ABC & \stackrel{def}{=} (\nu d, e, f)A \\
A & \stackrel{def}{=} a.(A \mid d.b.\overline{e}) + \tau.B \\
B & \stackrel{def}{=} \overline{d}.e.(B \mid f.c.\mathbf{0}) + \tau.C \\
C & \stackrel{def}{=} \overline{f}.C
\end{array}
$$

Gerarchia di linguaggi

Definition 3.8. (Finite-net CCS language) A language $L \subseteq (\mathscr{L} \cup \overline{\mathscr{L}})^*$ is a *finite-net CCS language* if there exists a finite-net CCS process p such that the set of its weak completed traces is L, i.e., $WCTr(p) = L$. □



Gerarchia di linguaggi (2)

- L_1 is the set $CTr(A)$ for $A \stackrel{def}{=} a.(A \,|\, b.\mathbf{0}) + c.\mathbf{0}$, as discussed in Exercise 3.57.
- $L_2 = CTr(B)$ for $B \stackrel{def}{=} a.(B \,|\, b.\mathbf{0}) + c.(B \,|\, d.\mathbf{0}) + e.\mathbf{0}$, as discussed in Example 3.13.
- $L_3 = \{a^n c b^n \mid n \geq 0\}$, as discussed in Exercise 3.61 (see also Example 3.15).
- L_4 is the language $\{ww^R \mid w \in \{a,b\}^*\}$, discussed after Exercise 3.62, and realized in finitary CCS in Exercise 3.65.
- L_5 is the context-dependent language $\{a^n b^m c^m \mid 0 \leq m \leq n\}$ discussed in Example 3.16.

- Il linguaggio L4 è un linguaggio libero non rappresentabile con reti di Petri finite, quindi nemmeno in finite-net CCS.

Finitary CCS

- Sintatticamente: è il più grande sotto-linguaggio di CCS che include tutti gli altri, in particolare finite-net CCS perché permette di usare la restrizione dentro il corpo di una costante.
  - Unico vincolo è che p deve avere un insieme Const(p) finito.
- Semanticamente:
  - è strettamente più espressivo di finite-net CCS (Counter)
  - Dimostreremo che è anche Turing-completo.
  - Tuttavia, esistono processi (full) CCS che non hanno nessun processo finitary CCS equivalente.

Counter in full CCS

- Contatore, simile a semi-counter, ma può fare test su zero.

$$Counter_0 \stackrel{def}{=} zero.Counter_0 + inc.Counter_1$$
$$Counter_n \stackrel{def}{=} inc.Counter_{n+1} + dec.Counter_{n-1} \quad n > 0$$

- Nessun processo finite-net CCS q può essere equivalente a $Counter_0$ perché:
  - Ogni processo finite-net CCS genera una rete di Petri finita,
  - Agerwala ha dimostrato che nessuna rete di Petri finita può fedelmente rappresentare un contatore con test su zero.
  - In conclusione, nessun processo finite-net CCS può rappresentare un counter.

Counter in Finitary CCS

- Processo Finitary CCS che usa solo 3 costanti:

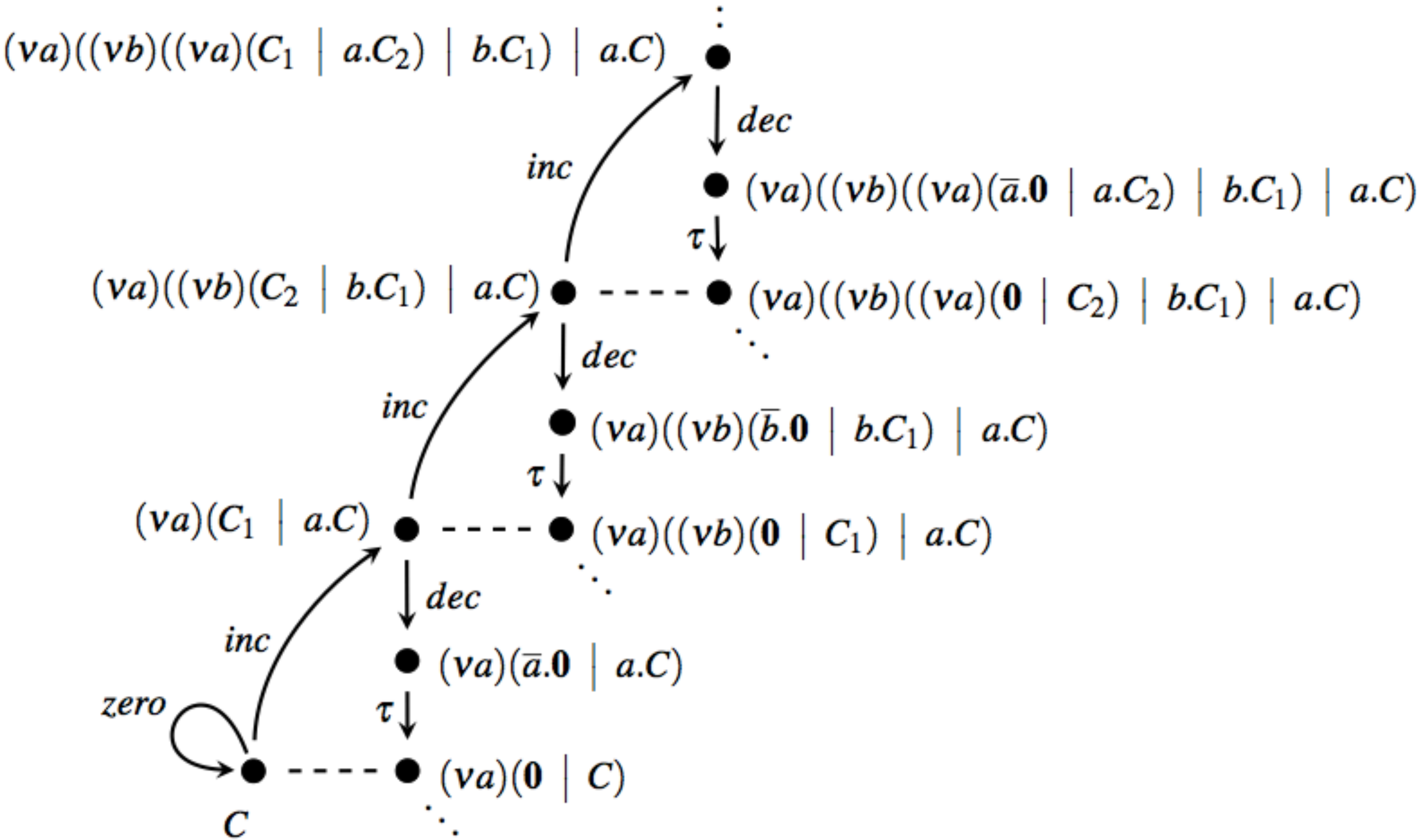
$$C \stackrel{def}{=} zero.C + inc.((\nu a)(C_1 \,|\, a.C))$$

$$C_1 \stackrel{def}{=} dec.\bar{a}.\mathbf{0} + inc.((\nu b)(C_2 \,|\, b.C_1))$$

$$C_2 \stackrel{def}{=} dec.\bar{b}.\mathbf{0} + inc.((\nu a)(C_1 \,|\, a.C_2))$$

- Si alternano le attivazioni di istanze di C1 (dispari) e C2 (pari), sempre dentro nuove restrizioni create ogni volta che si esegue inc.
- Il numero rappresentato da un termine è dato dal numero di restrizioni "attive" presenti.

Counter in Finitary CCS (2)



$Counter_0$ e C sono weak bisimili

- Si può vedere che $Counter_0$ e C sono weak bisimili.
- Definiamo

  $p_0 = C$ $\qquad$ $p_1 = (\nu a)(x|a.C)$ dove x è un place-holder

  $p_{2n} = p_{2n-1}[(\nu b)(x|b.C1)/x]$ per $n > 0$

  $p_{2n+1} = p_{2n}[(\nu a)(x|a.C2)/x]$ per $n > 0$

La relazione

$$\begin{aligned} R = \; & \{(C, Counter_0)\} \\ & \cup \; \{(p_{2n}[C_2/x], Counter_{2n}) \mid n > 0\} \\ & \cup \; \{(p_{2n+1}[C_1/x], Counter_{2n+1}) \mid n \geq 0\} \\ & \cup \; \{(p_{2n+1}[\bar{a}.\mathbf{0}/x], Counter_{2n}) \mid n \geq 0\} \\ & \cup \; \{(p_{2n}[\bar{b}.\mathbf{0}/x], Counter_{2n-1}) \mid n > 0\}. \end{aligned}$$

è una weak bisimulation up to ≈.

$Counter_0$ e C sono weak bisimili (2)

- (C, $Counter_0$) è coppia di bisimulazione. Infatti,
  - se C–zero→C, allora $Counter_0$–zero→$Counter_0$ e (C, $Counter_0$) è in R.
  - se C–inc→(va)(C1 |a.C), allora $Counter_0$–inc→ $Counter_1$ e (p_1[C1/x],$Counter_1$) appartiene al terzo gruppo di R (quando n = 0).
  - Simmetricamente se $Counter_0$ muove.

Counter_0 e C sono weak bisimili (3)

- Prendiamo le coppie (p_{2n+1}[C1/x], Counter_{2n+1}) per $n \geq 0$ e dimostriamo che sono ok. Infatti,
  - se p_{2n+1}[C1/x]–inc→ p_{2n+1}[(νb)(C2|b.C1)/x] che è proprio p_{2n+2}[C2/x], allora

    Counter_{2n+1}–inc→Counter_{2n+2} e (p_{2n+2}[C2/x], Counter_{2n+2}) appartiene al secondo gruppo in R.
  - se p_{2n+1}[C1/x]–dec→ p_{2n+1} [$\bar{a}$.0/x], allora

    Counter_{2n+1}–dec→Counter_{2n} e (p_{2n+1}[$\bar{a}$.0/x], Counter_{2n}) appartiene al quarto gruppo in R.
  - Simmetricamente se Counter_{2n+1} muove.

$Counter_0$ e C sono weak bisimili (4)

- Prendiamo la coppia ($p_1[\bar{a}.0/x]$, $Counter_0$)
- Ora, $p_1[\bar{a}.0/x] = (\nu\ a)(\bar{a}.0 \mid a.C)$
  - Solo una transizione: $p_1[\bar{a}.0/x]$ –tau→ $(\nu\ a)(0 \mid C)$
  - Inoltre $(\nu\ a)(0 \mid C) \sim C$ e
  - $Counter_0 =\varepsilon\Rightarrow Counter_0 \approx Counter_0$ e
    (C, $Counter_0$) appartiene ad R.
- Da $Counter_0$ due transizioni:
  - $Counter_0$–zero→$Counter_0$, allora $p_1[\bar{a}.0/x]$ –tau→ $(\nu\ a)(0 \mid C)$ –zero→ $(\nu\ a)(0 \mid C) \sim C$ e (C, $Counter_0$) in R.
  - $Counter_0$–inc→$Counter_1$, allora $p_1[\bar{a}.0/x]$ –tau→ $(\nu\ a)(0 \mid C)$ –inc→ $(\nu\ a)(0 \mid (\nu a)(C1 \mid a.C)) \sim (\nu a)(C1 \mid a.C) = p1[C1/x]$ e la coppia ($p_1[C1/x]$,$Counter_1$) in R.

$Counter_0$ e C sono weak bisimili (5)

- Prendiamo le coppie $(p_{2n+1}[\bar{a}.0/x], Counter_{2n})$ per $n > 0$ e dimostriamo che sono in R.
- Infatti, $p_{2n+1}[\bar{a}.0/x]$ è $p_{2n}[(\nu\ a)(x \mid a.C2\ )/x][\bar{a}.0/x] =$
  $= p_{2n}\ [(\nu\ a)(\bar{a}.0 \mid a.C2\ )/x]$.
  - Solo una transizione:
    $p_{2n}[(\nu\ a)(\bar{a}.0 \mid a.C2\ )/x] \text{–tau}\rightarrow p_{2n}[(\nu\ a)(0 \mid C2\ )/x]$
  - Inoltre $p_{2n}[(\nu\ a)(0 \mid C2\ )/x] \sim p_{2n}[C2\ /x]$ e
  - $Counter_{2n} {=}\varepsilon{=}{>} Counter_{2n} \approx Counter_{2n}$ e $(p_{2n}[C2/x], Counter_{2n})$ appartiene al secondo gruppo in R.
- Se $Counter_{2n}$ muove, allora …. (altri casi come esercizio)

Full CCS

- Nessun vincolo sintattico sul numero delle costanti.
- Full CCS è più espressivo di Finitary CCS. Infatti:
  - Proposizione: se p è un processo di Finitary CCS, allora l'insieme delle sue azioni eseguibili sort(p) è finito. (perché sort(p) è un sottoinsieme delle azioni che sintatticamente occorrono libere in p)
  - Il processo full CCS A_0, dove le infinte costanti A_k sono definite così $A_k = a_k.A_{k+1}$ (per $k \geq 0$) è tale che sort(A_0) è infinito = $\{a_0, a_1, a_2, ...\}$

Finite CCS vs altri sotto-CCS

- Sintatticamente Finite CCS non è contenuto in regular CCS (mentre lo è semanticamente): ad esempio a.(a.0 | b.0) è un processo finite CCS che non è in regular CCS (posso solo applicare il prefisso ad un processo sequenziale in regular CCS)
- Sintatticamente Finite CCS non è contenuto in finite-net CCS (mentre lo è semanticamente): ad esempio a.((νa)a.0 | b.0) è un processo finite CCS che non è in finite-net CCS (non posso applicare il prefisso ad un processo con restrizione in finite-net CCS)
- Sintatticamente Regular CCS non è contenuto in finite-net CCS (mentre lo è semanticamente): ad es. a.0 | (νa)b.0 è un processo regular (e anche finite) CCS che non è in finite-net CCS.