Linguaggi per il global computing
Esercizi B e D + Barbershop

Federico Perin - 2029215 - Ottobre 2021

Indice

1 Esercizi

1.1 Esercizio B

Dimostrare che ogni processo CCS finito termina in un numero finito di passi.

1.1.1 Sintassi

$$P, Q ::= \alpha.P \mid (P \mid Q) \mid \sum_{i \in I} P_i \mid P \setminus L \mid P[f] \mid \mathbf{0}$$

Note:

Nel CCS finito non sono previste le costanti $\mathcal{K}$.

Il processo $\mathbf{0}$ ha un solo stato e non ha interazioni con altri processi.

La dimostrazione viene divisa in due casi:

- Nel primo caso si hanno somme finite, cioè l'insieme I contenente le scelte della somma non deterministica, è finito;
- Nel secondo caso l'insieme I sarà infinito.

1.1.2 Dimostrazione per somme finite

La dimostrazione prevede i seguenti punti:

1. Ogni processo del CCS finito termina con un numero finito di passi;
2. Ogni processo del CCS finito ha un numero finito di stati.

1.1.2.1 Punto 1

Dato che non esistono costanti $\mathcal{K}$, non è possibile rigenerare passi eseguiti in precedenza, perciò dopo un certo numero finito di passi ogni processo P in CCS finito terminerà perché non avrà più passi da eseguire. Si deduce perciò che ogni processo ha un numero di passi limitato da un limite superiore è quindi tale numero è finito.

Per dimostrare quanto scritto si procede attraverso una dimostrazione induttiva sull'altezza di derivazione di un processo P, con ipotesi induttiva: $P \xrightarrow{\alpha} P' \Rightarrow Size_p(P') < Size_p(P)$, dove $Size_p(P')$ si intende il numero di passi del processo P dopo aver fatto l'azione α.

Definiamo $Size_p(P)$:

$$Size_p(P) \begin{cases} P = \alpha.R, & 1 + Size_p R \\ P = \sum_{i \in I} P_i, & max(Size_p(P_i)) \\ P = R \mid Q, & Size_p R + Size_p Q \\ P = R \setminus L, & Size_p R \\ P = R[f], & Size_p R \end{cases}$$

Si dimostrerà che $Size_p(P)$ indica il limite superiore del numero di passi eseguiti dal processo P per terminare.

1.1.2.1.1 Dimostrazione

Caso Base:

$\overline{\alpha.P \xrightarrow{\alpha} P}$ ACT $\qquad Size_p(\alpha.P) = 1 + Size_p(P) > Size_p(P)$

Si ha che l'azione α concatenata al processo P aggiunge un passo in più, perciò risulta essere corretto il limite superiore $Size_p(\alpha.P) = 1 + Size_p(P)$ passi.

Caso Induttivo:

* $\dfrac{P_j \xrightarrow{\alpha} P'}{\sum\limits_{i \in I} P_i \xrightarrow{\alpha} P'}$ $SUM\ j \in I$

  Si osserva che nell'esecuzione di un passo α, per ipotesi induttiva vale che: $Size_p(P_j) > Size_p(P')$.

  Perciò $Size_p(\sum\limits_{i \in I} P_i) > Size_p(P_j) > Size_p(P')$ allora $max(Size_p(P_i)) > Size_p(P')$

  Il limite superiore $Size_p(\sum\limits_{i \in I} P_i) = max(Size_p(P_i))$ passi, risulta essere corretto.

* $\dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$ $PAR\text{-}L$

  Si osserva che nell'esecuzione di un passo α, per ipotesi induttiva vale che: $Size_p(P) > Size_p(P')$.

  E quindi dato che $Size_p(P|Q) = Size_p(P) + Size_p(Q)$ mentre $Size_p(P'|Q) = Size_p(P') + Size_p(Q)$, per ipotesi induttiva $Size_p(P|Q) > Size_p(P'|Q)$

* $\dfrac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$ $PAR\text{-}R$

  Si osserva che nell'esecuzione di un passo α, per ipotesi induttiva vale che: $Size_p(Q) > Size_p(Q')$

  E quindi dato che $Size_p(P|Q) = Size_p(P) + Size_p(Q)$ mentre
  $Size_p(P|Q') = Size_p(P) + Size_p(Q')$, per ipotesi induttiva
  $Size_p(P|Q) > Size_p(P|Q')$

* $\dfrac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$ $PAR\text{-}\tau$

  Si osserva che nell'esecuzione di un passo di sincronizzazione, per ipotesi induttiva vale che: $Size_p(P) > Size_p(P')$ e $Size_p(Q) > Size_p(Q')$

  E quindi dato che $Size_p(P|Q) = Size_p(P) + Size_p(Q)$ mentre
  $Size_p(P'|Q') = Size_p(P') + Size_p(Q')$, per ipotesi induttiva
  $Size_p(P|Q) > Size_p(P'|Q')$

Perciò si è dimostrato che il limite superiore $Size_p(P|Q)$ =$Size_p(P) + Size_p(Q)$ passi, risulta essere corretto in tutti i tre casi PAR-L, PAR-R e PAR-τ.

* $\dfrac{P \overset{\alpha}{\to} P'}{P\backslash L \overset{\alpha}{\to} P'\backslash L}$ RES se $\alpha, \overline{\alpha} \notin$ L

  Si osserva che nell'esecuzione di un passo α, per ipotesi induttiva vale che: $Size_p(P) > Size_p(P')$

  Applicare una restrizione ad un processo non fa aumentare il numero massimo di passi di esecuzione, ma avere una possibile variazione delle possibili interazioni con altri processi in parallelo, vale che:
  $Size_p(P\backslash L) = Size_p(P)$ mentre $Size_p(P'\backslash L) = Size_p(P')$, per ipotesi induttiva $Size_p(P\backslash L) > Size_p(P'\backslash L)$

  Perciò si è dimostrato che il limite superiore $Size_p(P\backslash L) = Size_p(P)$ passi, risulta essere corretto.

* $\dfrac{P \overset{\alpha}{\to} P'}{P[\mathbf{f}] \overset{f(\alpha)}{\to} P'[\mathbf{f}]}$ REL

  Si osserva che nell'esecuzione di un passo α, per ipotesi induttiva vale che: $Size_p(P) > Size_p(P')$

  Applicare un relabelling ad un processo non fa aumentare il numero massimo di passi di esecuzione, ma avere una possibile variazione delle possibili interazioni con altri processi in parallelo, vale che:
  $Size_p(P[\mathbf{f}]) = Size_p(P)$ mentre $Size_p(P'[\mathbf{f}]) = Size_p(P')$, per ipotesi induttiva $Size_p(P[\mathbf{f}]) > Size_p(P'[\mathbf{f}])$

  Perciò si è dimostrato che il limite superiore $Size_p(P[\mathbf{f}]) = Size_p(P)$ passi, risulta essere corretto.

1.1.2.2 Punto 2

Si vuole dimostrare che ogni processo CCS ha un numero finito di stati, cioè esiste un limite superiore del numero di stati di esecuzione per un processo P.

Definiamo $Size_s(P)$:

$$Size_s(P) \begin{cases} P = \mathbf{0}, & 1 \\ P = \alpha.R, & 1 + Size_s R \\ P = \sum_{i \in I} P_i, & \sum_{i \in I} Size_s P_i \\ P = R \mid Q, & Size_s R * Size_s Q \\ P = R \setminus L, & Size_s R \\ P = R[f], & Size_s R \end{cases}$$

Si dimostrerà di seguito che $Size_s(()$P) calcola il limite superiore del numero di stati dell'esecuzione del processo P.

1.1.2.2.1 Dimostrazione

Tramite dimostrazione per induzione sull'esecuzione di un processo P si dimostra che i limiti di $Size_s(P)$ sono corretti.

Caso Base:

0
Il processo **0** ha un solo stato, quello di partenza, per definizione quindi $Size_s(\mathbf{0}) = 1$ stato.

Caso Induttivo:

In un passo si raggiunge uno sotto processo dal quale poi in n passi finiti si raggiungerà un stato terminante. Si utilizza la seguente ipotesi induttiva:
Un processo composto da sotto processi con un limite superiore ai stati di esecuzione ha anch'esso un limite superiore di stati di esecuzione.
Tale ipotesi scritta verrà utilizzata per dimostrare che il sotto processo raggiunto, da lui allo stato terminante, ci saranno un numero finiti di stati perché limitati da un limiti superiore. Si ricorda inoltre che il numero di stati raggiunti è finito se gli n passi sono finiti. Grazie alla dimostrazione del punto 1 si sa che n è finito.

* **ACT**
  Con l'azione α concatenata al processo P viene aggiunto a P uno stato in più, infatti $\alpha.P \overset{\alpha}{\to} P$. P per ipotesi induttiva ha al più $Size_s(P)$ stati, quindi $\alpha.P$ avrà al più $Size_s(\alpha.P) = 1 + Size_s(P)$ stati.

* **SUM**
  Si ha la seguente esecuzione: $\sum_{i \in I} P_i \overset{\alpha}{\to}$ P', dove qualunque P_i sia scelto per ipotesi induttiva ha un numero finito di stati d'esecuzione. Quindi tutti i processi che compongono la somma non deterministica hanno un numero finito di stati. Perciò $\sum_{i \in I} P_i$ ha al più $Size_s(\sum_{i \in I} P_i) = \sum_{i \in I} Size_s P_i$ stati inoltre, dato che il numero di processi P_i è finito, anche la somma dei stati lo sarà.

  È corretto il limite superiore presentato precedentemente perché potrebbero esserci stati non condivisi tra i vari processi P_i, quindi tutti i $Size_s(P_i)$.

* **PAR-L**
  Si ha la seguente esecuzione: $P|Q \overset{\alpha}{\to} P'|Q$, dove P|Q raggiunge lo stato P'|Q.

  Per ipotesi induttiva P'|Q ha un numero finito di stati d'esecuzione, cioè ha al più $Size_s(P'|Q) = Size_s(P')$ * $Size_s(Q)$ stati, quindi P|Q ha al più $Size_s(P|Q) = (Size_s(P') + 1)$ * $Size_s(Q)$ stati. Dato che è possibile che non ci sia alcun stato condiviso durante l'esecuzione dei due processi, il limite superiore ad essi è uguale al totale delle combinazioni possibili tra gli stati dei due singoli processi.

* **PAR-R**
  Si ha la seguente esecuzione: $P|Q \overset{\alpha}{\to} P|Q'$, dove P|Q raggiunge lo stato P|Q'.

Per ipotesi induttiva P|Q' ha un numero finito di stati d'esecuzione, cioè ha al più $Size_s(P|Q') = Size_s(P) * Size_s(Q')$ stati, quindi P|Q ha al più $Size_s(P|Q) = Size_s(P) * (Size_s(Q') + 1)$ stati. Dato che è possibile che non ci sia alcun stato condiviso durante l'esecuzione dei due processi, il limite superiore ad essi è uguale al totale delle combinazioni possibili tra gli stati dei due singoli processi.

* **PAR-τ**
  Si ha la seguente esecuzione: $P|Q \xrightarrow{\tau} P'|Q'$, dove P|Q raggiunge lo stato P'|Q'.

  Per ipotesi induttiva P'|Q' ha un numero finito di stati d'esecuzione, cioè ha al più $Size_s(P'|Q') = Size_s(P') * Size_s(Q')$ stati, quindi P|Q ha al più $Size_s(P|Q) = Size_s(P') * Size_s(Q') + 1$ stati. Dato che è possibile che non ci sia alcun stato condiviso durante l'esecuzione dei due processi, il limite superiore ad essi è uguale al totale delle combinazioni possibili tra gli stati dei due singoli processi.

  Dunque $Size_s(P|Q) = Size_s(P) * Size_s(Q)$ stati, risulta essere corretto.

* **RES**
  Si ha la seguente esecuzione: $P\backslash L \xrightarrow{\alpha} P'\backslash L$.

  Per ipotesi induttiva P' \L ha al più $Size_s(P'\backslash L) = Size_s(P')$ stati.
  Dato che $Size_s(P'\backslash L)$ è finito, allora P \L avrà al più:
  $Size_s(P\backslash L) = 1 + Size_s(P'\backslash L) = 1 + Size_s(P') = Size_s(P)$ stati, perché applicando una funzione di restrizione non si aumentano il numero massimo di stati d'esecuzione, vale quindi il limite superiore $Size_s(P\backslash L) = Size_s(P)$.

* **REL**
  Si ha la seguente esecuzione: $P[\mathbf{f}] \xrightarrow{f(\alpha)} P'[\mathbf{f}]$.

  Per ipotesi induttiva $P'[\mathbf{f}]$ ha al più $Size_s(P'[\mathbf{f}]) = Size_s(P')$ stati.
  Dato che $Size_s(P'[\mathbf{f}])$ è finito, allora $P[\mathbf{f}]$ avrà al più:
  $Size_s(P[\mathbf{f}]) = 1 + Size_s(P'[\mathbf{f}]) = 1 + Size_s(P') = Size_s(P)$ stati, perché applicando una funzione di relabelling non si aumentano il numero massimo di stati d'esecuzione ma si cambiano solo le possibili interazioni con altri processi in parallelo, vale quindi il limite superiore $Size_s(P[\mathbf{f}]) = Size_s(P)$.

1.1.2.3 Conclusione prima parte

Si è dimostrato nei punti precedenti che i processi definiti attraverso il CCS finito hanno sempre un limite superiore sia per il numero di passi e sia per il numero di stati d'esecuzione, di conseguenza il numero di passi e stati sono entrambi finiti.

1.1.3 Dimostrazione per somme infinite

Ora per quanto riguarda la somma non deterministica $\sum_{i \in I} P_i$, l'insieme I sarà infinito. Infatti i processi CCS che fanno parte della somma non deterministica, saranno infiniti e quindi hanno un numero infinito di stati ma hanno tutti una derivazione finita.

La dimostrazione sarà simile alla precedente per tutti i vari punti, tranne per il punto riguardante la somma non deterministica, infatti per dimostrare la finitezza dell'esecuzione si utilizzerà un'astrazione basata sulle generazioni della grammatica di CCS.

Si dimostra perciò per induzione sulla lunghezza di derivazione che ogni processo CCS termina in numero finito di passi.

Caso Base:

0

Il processo **0** termina in 0 passi per definizione.

Caso Induttivo:
La generazione della sequenza di interazioni avanza attraverso uno dei termini della grammatica e come ipotesi induttiva si ha che tale sequenza ha derivazione finita.

* **ACT, RES e REL**
  Con l'azione α concatenata al processo P si ha che, la derivazione produrrà un processo del tipo, $\alpha^{n+1}.P$ con $n \geq 0$, dove esiste un nuovo passo e uno nuovo stato nell'esecuzione di P. Perciò il numero di stati e di passi d'esecuzione rimangono finiti.

* **PAR-L, PAR-R, PAR-τ**
  Come dimostrato per il caso del CCS finito con somme finite, nell'esecuzione parallela la derivazione produrrà dei sotto processi che per ipotesi induttiva sono finiti e quindi il processo di cui fanno parte sarà anch'esso finito.

* **RES**
  Con **RES** si avrà che la derivazione produrrà un certo numero finito di restrizioni. Perciò si avranno sempre passi e stati in numero finito e quindi vale quanto dimostrato nel CCS finito con somme finite.

* **REL**
  Con **REL** si avrà che la derivazione produrrà un certo numero finito di relabelling. Perciò si avranno sempre passi e stati in numero finito e quindi vale quanto dimostrato nel CCS finito con somme finite.

* **SUM** In questo caso il processo derivato sarà del tipo $P_1 + P_2 + \ldots$. Per ipotesi induttiva ogni P_i ha una derivazione di lunghezza finita, perciò si può dimostrare che la derivazione del processo P è finita per induzione sul numero di sotto processi che vengono usati nella somma.

  Caso Base:

  Processo **0** per definizione è finito.

  Caso Induttivo:

  L'ipotesi induttiva afferma che la scelta non deterministica tra i processi ha una derivazione di lunghezza finita. Inoltre si deve tener conto che nella scelta non deterministica, viene scelto un solo processo da eseguire tra tutti quelli presenti.

  Quindi aggiungendo il processo P_{n+1}alla scelta, che per ipotesi induttiva anch'esso ha derivazione finita; la scelta diventerà tra l'esecuzione di uno dei processi già presenti e il processo appena aggiunto P_{n+1}. Solo uno dei processi verrà eseguito e quindi la derivazione avrà una lunghezza superiormente limitata dalla massima lunghezza di derivazione dei sotto processi. Si è perciò dimostrato che l'insieme dei processi della scelta non deterministica può essere illimitato ma finito senza perdere la finitezza di esecuzione. Questo perché la derivazione esegue solo uno dei processi della scelta come scritto in precedenza e in più si sa che l'altezza della derivazione di P è limitata superiormente dall'altezza della derivazione del processo con l'altezza maggiore + 1, ovvero $max_h(P_i) + 1$.

Purtroppo pero il numero di stati che si ha durante l'esecuzione non è più limitato superiormente, perché essendo il limite superiore dato dalla $\sum_{i \in I} P_i$ con I infinito, il numero di processi CCS risulta essere infinito, e quindi la somma dei stati d'esecuzione sarà anch'essa infinita. Perciò il numero di stati non é limitato superiormente.

1.1.4 Conclusione

È stato dimostrato che ogni processo CCS finito termina in un numero finito di passi, indipendente dal fatto si utilizzi una grammatica che permette scelte non deterministiche in un insieme o infinito o finito. Si sottolinea che scelta di un insieme infinito o finito determinerà, se il numero di stati d'esecuzione raggiungibili sarà finito o infinito.

1.2 Esercizio D

Dimostrare che la trace equivalence è una congruenza per il CCS.

Prima di illustrare la dimostrazione si definisce che cosa si intende con i concetti di trace equivalence e congruenza.

Innanzitutto per tracce di un processo P che di seguito verrà indicata con Tr(P), si intendono tutte le possibili sequenze di interazioni $\alpha_1 .....\alpha_n \in Act$ con n >= 0 tale che esiste una sequenza di transizioni $P \overset{\alpha_1}{\to} P_1 \overset{\alpha_2}{\to} ... \overset{\alpha_n}{\to} P_n$, e quindi rappresentata tutte le possibili interazioni con un processo. Più formalmente Tr(P) = { $\alpha_1 .....\alpha_n | P \overset{\alpha_1}{\to} P_1 \overset{\alpha_2}{\to} ... \overset{\alpha_n}{\to} P_n$ }. Quindi due processi P e Q si dicono trace equivalence P$\sim_t$Q se Tr(P) = Tr(Q).

Per congruenza si intende, dati due processi P e Q in relazione tra loro (P R Q), allora per ogni contesto C[ ], C[P] R C[Q].

Perciò si dimostrerà che se P$\sim_t$Q $\Rightarrow$ $\forall$C[ ] C[P] $\sim_t$C[Q].

1.2.1 Dimostrazione

Siano P,Q e R processi CCS con P $\sim_t$ Q, allora

1. α.P $\sim_t$ α.Q
2. P + R $\sim_t$ Q + R
3. P|R $\sim_t$ Q|R
4. P\L $\sim_t$ Q\L
5. P[**f**] $\sim_t$ Q[**f**]

Si definiscono di seguito alcune terminologie che verranno usate durante la dimostrazione.

Si indica con ε la sequenza vuota di interazioni. Essendo vuoto tutti i processi CCS sono in grado di eseguirla.

Sia R un processo CCS, indichiamo con α.Tr(R) l'insieme {αt | t $\in$ Tr(R)}

1.2.1.1 Prefisso C[ ] = α.

Nel caso del contesto prefisso si ha che Tr($\alpha.P$) = α.Tr(P). Se questo è vero allora, grazie all'ipotesi Tr(P) = Tr(Q) si ha che α.Tr(P) = α.Tr(Q) = Tr($\alpha.Q$) e quindi vale che α.P $\sim_t$ α.Q

($\subseteq$)

Sia t $\in$ Tr($\alpha.P$) $\Rightarrow$ t $\in$ α.Tr(P)

Per induzione su |t|:

Caso Base |t| = 0

Allora t = $\varepsilon \in \alpha$.Tr(P)

Caso Induttivo |t| = n + 1

t ha una prima interazione seguita poi dalla traccia t'. Sappiamo che $\alpha.P \overset{\alpha}{\to}$ P $\overset{t'}{\to}$, ovvero α.P sa fare α grazie alla regola del prefisso che ne permette la transizione.

$$\frac{}{\alpha.P \overset{\alpha}{\to} P \overset{t'}{\to}} ACT$$

t = α.t' dove t' sarà una certa sequenza di interazioni con |t'| = n e per ipotesi induttiva t'∈ Tr(P). E quindi vale che t = α.t'∈ α.Tr(P).

(⊇)

Sia t ∈ α.Tr(P) ⇒ t ∈ Tr($\alpha.P$)

Suddividiamo il problema in due casi:

Caso t = ε

Per definizione t = ε ∈ Tr($\alpha.P$).

Caso t ≠ ε

Per definizione di α.Tr(P), t = α.t' con t'∈Tr(P). È quindi è possibili effettuare la seguente sequenza di transizioni $\alpha.P \overset{\alpha}{\to} P \overset{t'}{\to}$. Questo dimostra che $\alpha.P$ sa fare l'interazione t, allora t = α.t' ∈Tr(α.P).

Perciò si è dimostrato che Tr($\alpha.P$) = α.Tr(P) e quindi con Tr(P) = Tr(Q), α.P $\sim_t$ α.Q come si voleva dimostrare.

1.2.1.2 Contesto non deterministico C[ ] = (+ R)

Nel caso del contesto non deterministico tra i processi P e R le Tr(P + R) = Tr(P) ⋃ Tr(R). Se questo è vero, dato che per ipotesi Tr(P) = Tr(Q) e Tr(Q + R) = Tr(Q) ⋃ Tr(R), allora Tr(P + R) = Tr(Q + R) .

Perciò si deve dimostrare che il contesto non deterministico tra i processi P e R è uguale alla unione delle traccie dei due processi. Dimostrato questo ne consegue la veridicità di P + R $\sim_t$ Q + R.

(⊆)

Sia t ∈ Tr(P + R) ⇒ t ∈ (Tr(P) ⋃ Tr(R))
Per induzione su |t|:

Caso Base |t| = 0

Allora t = ε ∈(Tr(P) ⋃ Tr(R))

Caso Induttivo |t| = n + 1

t ha una prima interazione seguita poi dalla traccia t', quindi P + R $\stackrel{\alpha_1}{\to}$ X' $\stackrel{t'}{\to}$ ovvero viene applicata una transizione secondo la regola della somma non deterministica arrivando in certo processo X'. Ci sono perciò due possibilità:

- P+R ha effettuato una transizione usando la regola SUM-L:

$$\frac{P \stackrel{\alpha_1}{\to} P'}{P + R \stackrel{\alpha_1}{\to} P' \stackrel{t'}{\to}} \; SUM\text{-}L$$

  t = α_1.t' dove t' è una certa sequenza di interazioni con |t'| = n. Per ipotesi induttiva t' ∈ Tr(P'), quindi α_1.t' ∈ α_1.Tr(P') che ne consegue che t ∈ α_1.Tr(P'). Alla luce di ciò e grazie alla regola SUM-L che permette la transizione P $\stackrel{\alpha_1}{\to}$P', si può concludere che t ∈ Tr(P) .

- P+R ha effettuato una transizione usando la regola SUM-R:

$$\frac{R \stackrel{\alpha_1}{\to} R'}{P + R \stackrel{\alpha_1}{\to} R' \stackrel{t'}{\to}} \; SUM\text{-}R$$

  t = α_1.t' dove t' è una certa sequenza di interazioni con |t'| = n. Per ipotesi induttiva t' ∈ Tr(R'), quindi α_1.t' ∈ α_1.Tr(R') che ne consegue che t ∈ α_1.Tr(R'). Alla luce di ciò e grazie alla regola SUM-R che permette la transizione R $\stackrel{\alpha_1}{\to}$R', si può concludere che t ∈ Tr(R) .

(⊇)

Sia t ∈(Tr(P) $\bigcup$ Tr(R)) ⇒ t ∈Tr(P + R)
t può essere una traccia sia di P e sia di R oppure solo uno dei due.

Suddividiamo il problema in due casi:

Caso t = ε

Per definizione t = ε ∈ Tr(P + R).

Caso t ≠ ε

t è una sequenza non vuota di interazioni di P.

Se t ∈ Tr(P), si sa che esiste la sequenza di transizione P $\stackrel{\alpha_1}{\to}$ P' $\stackrel{t'}{\to}$. Quindi t = α_1.t' con t' ∈ Tr(P'). Fatta questa premessa, applicando la regola SUM-L si ottiene che $P + R \stackrel{\alpha_1}{\to} P' \stackrel{t'}{\to}$, questo dimostra che P + R sa fare t, allora t ∈ Tr(P + R).

Se t ∈ Tr(R), si sa che esiste la sequenza di transizione R $\stackrel{\alpha_1}{\to}$ R' $\stackrel{t'}{\to}$. Quindi t = α_1.t' con t' ∈ Tr(R'). Fatta questa premessa, applicando la regola SUM-R si ottiene che $P + R \stackrel{\alpha_1}{\to} R' \stackrel{t'}{\to}$, questo dimostra che P + R sa fare t, allora t ∈ Tr(P + R).

Se t appartiene sia a P che R, qualsiasi regola venga applicata per fare la transizione vale sempre t ∈ Tr(P + R).

Perciò si è dimostrato che Tr(P + R) = Tr(P) $\bigcup$ Tr(R) e quindi con Tr(P) = Tr(Q), P + R $\sim_t$ Q + R come si voleva dimostrare.

1.2.1.3 Contesto parallelo C[ ] = (| R)

Intuitivamente le traccie Tr(P|R) sono tutte le possibili combinazione tra Tr(P) e Tr(R), cioè quindi tutte le loro interazioni e sincronizzazioni. Se tale intuizione è vera allora dato che Tr(P) = Tr(Q), si potrebbe sostituire P con Q nelle Tr(P|R) ed ottenere le stesse combinazioni della versione precedente, varrebbe perciò Tr(P|R) = Tr(Q|R) e di conseguenza P|R $\sim_t$ Q|R.

Per dimostrare ciò, si definisce un insieme Tr(P,R) che contiene tutte e sole le sequenze di interazioni che si possono ottenere combinando le Tr(P) e le Tr(R) seguendo le regole del parallelo, PAR-L, PAR-R e PAR-τ.

Quindi si definisce la seguente funzione C che date due traccie ne esegue la loro combinazione:

$$C(a,b) = \begin{cases} max(x,y) & \text{se } a = \varepsilon \text{ OR } b = \varepsilon \\ \alpha_1 C(a', \alpha_2 b') \bigcup \alpha_2 C(\alpha_1 a', b') & \text{se } a = \alpha_1 a' \text{ AND } b = \alpha_2 b' \\ \tau C(x', y') & \text{se } a = \alpha a' \text{ AND } b = \overline{\alpha} b' \end{cases}$$

dove Tr(P,R) = { C(a,b) | a *in* Tr(P), b *in* Tr(R)}.

Si procede con la dimostrazione Tr(P|R) = Tr(P,R):

($\subseteq$)

Sia t $\in$Tr(P|R) $\Rightarrow$ t $\in$(P,R).
Ovvero la funzione C sa eseguire la traccia t. Per induzione su |t|:

Caso Base |t| = 0

Allora t = ε, quindi $\varepsilon \in$ Tr(P) e $\varepsilon \in$ R per definizione. Perciò applicando Tr(P,R), si ha che $C(\varepsilon, \varepsilon)$ { max(ε, ε) } = $\{\varepsilon\}$ dove t $\in$ $\{\varepsilon\}$.

Caso Induttivo |t| = n + 1

Si ha che t = $\alpha_1.t'$ con |t'| = n ovvero, t ha una prima interazione seguita poi dalla traccia t', quindi si ha P|R $\xrightarrow{\alpha_1}$ X' $\xrightarrow{t'}$ con t'$\in$ Tr(X'). Viene perciò applicata una transizione secondo la regola del parallelo arrivando in un certo processo X'. Vi sono perciò tre possibilità:

- P|R ha effettuato una transizione usando la regola PAR-L:

  $$\frac{P \xrightarrow{\alpha_1} P'}{P|R \xrightarrow{\alpha_1} P'|R \xrightarrow{t'}} \ PAR\text{-}L$$

  t' $\in$ Tr(P'|R), per ipotesi induttiva t' $\in$ Tr(P',R) allora esistono a' $\in$ Tr(P'), b' $\in$ Tr(R), tale che t' $\in$ C(a',b'). Voglio pero dimostrare che a $\in$ Tr(P), b $\in$ Tr(R), tale che t $\in$ C(a,b). Come mostrato precedentemente con l'applicazione della regola PAR-L esiste un interazione α_1 tale che P $\xrightarrow{\alpha_1}$ P' con α_1a' $\in$ Tr(P).
  Perciò mostriamo che t $\in$ C(a = α_1a', b = b') = Tr(P, R). Quindi:

- Se b' = ε
  Si sa che C(a', ε) = {a'}, inoltre per ipotesi induttiva
  t' ∈ Tr(P',R) = { C(a', ε) | a' ∈ Tr(P'), ε ∈ Tr(R)} quindi risulta che t' = a', ma allora dato che α_1a' ∈ Tr(P) posso concludere che t = α_1t' ∈ C(α_1a', b') e quindi t ∈ Tr(P, R).
- Se b' ≠ ε
  Per ipotesi induttiva t' ∈ C(a', b'). Si sa che dalla definizione di C, C(α_1a', b') può assumere il valore $\alpha_1 C$(a', b'), ma allora t = α_1.t ∈ $\alpha_1 C$(a', b') e quindi t ∈ Tr(P, R).

- P|R ha effettuato una transizione usando la regola PAR-R:

$$\frac{R \stackrel{\alpha_1}{\rightarrow} R'}{P|R \stackrel{\alpha_1}{\rightarrow} P|R' \stackrel{t'}{\rightarrow}} \; \textit{PAR-R}$$

t' ∈ Tr(P|R'), per ipotesi induttiva t' ∈ Tr(P,R') allora esistono a' ∈ Tr(P), b' ∈ Tr(R'), tale che t' ∈ C(a',b'). Voglio pero dimostrare che a ∈ Tr(P), b ∈ Tr(R), tale che t ∈ C(a,b). Come mostrato precedentemente con l'applicazione della regola PAR-R esiste un interazione α_1 tale che R $\stackrel{\alpha_1}{\rightarrow}$ R' con α_1b' ∈ Tr(R).
Perciò mostriamo che t ∈ C(a = a', b = α_1b') = Tr(P, R). Quindi:

- Se a' = ε
  Si sa che C(ε, b') = {b'}, inoltre per ipotesi induttiva
  t' ∈ Tr(P,R') = { C(ε, b') | ε ∈ Tr(P'), b' ∈ Tr(R)} quindi risulta che t' = b', ma allora dato che α_1b' ∈ Tr(R) posso concludere che t = α_1t' ∈ C(a', α_1b') e quindi t ∈ Tr(P, R).
- Se a' ≠ ε
  Per ipotesi induttiva t' ∈ C(a', b'). Si sa che dalla definizione di C, C(a', α_1b') può assumere il valore $\alpha_1 C$(a', b'), ma allora t = α_1.t ∈ $\alpha_1 C$(a', b') e quindi t ∈ Tr(P, R).

- P|R ha effettuato una transizione usando la regola PAR-τ:

$$\frac{P \stackrel{\alpha_1}{\rightarrow} P' \qquad R \stackrel{\overline{\alpha_1}}{\rightarrow} R'}{P|R \stackrel{\tau_1}{\rightarrow} P'|R' \stackrel{t'}{\rightarrow}} \; \textit{PAR-}\tau$$

t' ∈ Tr(P'|R'), per ipotesi induttiva t' ∈ Tr(P', R') allora esistono a' ∈ Tr(P'), b' ∈ Tr(R'), tale che t' ∈ C(a',b'). Voglio pero dimostrare che a ∈ Tr(P), b ∈ Tr(R), tale che t ∈ C(a,b). Come mostrato precedentemente con l'applicazione della regola PAR-τ esistono le interazioni α_1 e $\overline{\alpha_1}$ tale che P $\stackrel{\alpha_1}{\rightarrow}$ P' e R $\stackrel{\overline{\alpha}_1}{\rightarrow}$ R' con α_1a' ∈ Tr(P) e $\overline{\alpha}_1$b' ∈ Tr(R).
Perciò mostriamo che t ∈ C(a = α_1a', b = $\overline{\alpha}_1$b') = Tr(P, R).
Quindi dato che si sincronizzano nessuna delle due traccie possono essere vuote, e per ipotesi induttiva t' ∈ C(a', b'), sapendo che dalla definizione di C, C(α_1a', $\overline{\alpha}_1$b') può assumere il valore τC(a', b'), allora t = τt' ∈ τC(a', b') e quindi t ∈ Tr(P, R).

($\supseteq$)

Sia t $\in$Tr(P,R) $\Rightarrow$ t $\in$(P|R)
Per induzione su |t|:

Caso Base |t| = 0

t = ε, allora $\varepsilon \in$ Tr(P | R) per definizione.

Caso Induttivo |t| = n + 1

t ha una prima interazione α_1 seguita poi dalla traccia t', ovvero t = α_1t'. Si ricorda inoltre che esiste una a $\in$ Tr(P), b $\in$ Tr(R), tale che t $\in$ C(a,b). Ci sono perciò tre possibili α_1:

- $\alpha_1 \in$ Tr(P), si ha quindi una sequenza di transizione P $\overset{\alpha_1}{\rightarrow}$ P' $\overset{a'}{\rightarrow}$ con a = $\alpha_1.a'$ con a'$\in$ Tr(P). Posso perciò applicare PAR-L:

  $$\frac{P \overset{\alpha_1}{\rightarrow} P'}{P|R \overset{\alpha_1}{\rightarrow} P'|R \overset{a'}{\rightarrow}} \; \textit{PAR-L}$$

  Dall'applicazione della regola PAR-L posso dedurre che $\alpha_1 \in$ Tr(P|R), |t'| = n per ipotesi induttiva t' $\in$ Tr(P'|R), allora t = α_1t' $\in$ Tr(P|R).

- $\alpha_1 \in$ Tr(R), si ha quindi una sequenza di transizione R $\overset{\alpha_1}{\rightarrow}$ R' $\overset{b'}{\rightarrow}$ con b = $\alpha_1.b'$ con b'$\in$ Tr(R). Posso perciò applicare PAR-R:

  $$\frac{R \overset{\alpha_1}{\rightarrow} R'}{P|R \overset{\alpha_1}{\rightarrow} P|R' \overset{b'}{\rightarrow}} \; \textit{PAR-R}$$

  Dall'applicazione della regola PAR-R posso dedurre che $\alpha_1 \in$ Tr(P|R), |t'| = n per ipotesi induttiva t' $\in$ Tr(P|R'), allora t = α_1t' $\in$ Tr(P|R).

- $\alpha_1 \in$ Tr(P) AND $\alpha_1 \in$ Tr(R), si ha quindi una sequenza di transizione P $\overset{\alpha_1}{\rightarrow}$ P' $\overset{a'}{\rightarrow}$ con a = α_1.a' con a'$\in$ Tr(P), e un altra sequenza di transizioni R $\overset{\overline{\alpha}_1}{\rightarrow}$ R' $\overset{b'}{\rightarrow}$ con b = $\overline{\alpha}_1.b'$ con b'$\in$ Tr(R). Posso perciò applicare PAR-τ:

  $$\frac{P \overset{\alpha_1}{\rightarrow} P' \qquad R \overset{\overline{\alpha_1}}{\rightarrow} R'}{P|R \overset{\tau_1}{\rightarrow} P'|R' \overset{t'}{\rightarrow}} \; \textit{PAR-}\tau$$

  Dall'applicazione della regola PAR-τ posso dedurre che $\tau_1 \in$ Tr(P|R), |t'| = n per ipotesi induttiva t' $\in$ Tr(P'|R'), allora t = τ_1t' $\in$ Tr(P|R).

Quindi sia Tr(P) = Tr(Q) e Tr(P,R) = { C(a,b) | a *in* Tr(P), b *in* Tr(R)} = Tr(P | R), allora posso sostituire le traccie di P con quelle di Q per ottenere { C(a,b) | a *in* Tr(Q), b *in* Tr(R)} = Tr(P,R) = Tr(Q | R), dimostrando che P|R $\sim_t$ Q|R.

1.2.1.4 Contesto restrizione C[ ] = \L

Il caso del contesto restrizione L sul processo P ha la seguente uguaglianza:
Tr(P\L) = Tr(P)\{$\alpha_1...\alpha_n|\alpha_i$ ∈L} cioè le traccie che stanno in Tr(P) non ci sono nell'insieme di restrizione L. Se questo è vero, dato che per ipotesi Tr(P) = Tr(Q) e quindi Tr(Q\L) = Tr(Q)\{$\alpha_1...\alpha_n|\alpha_i$ ∈L}, allora Tr(P\L) = Tr(Q\L).

Perciò si deve dimostrare che il contesto restrizione L sul processo P è uguale a Tr(P)\{$\alpha_1...\alpha_n|\alpha_i$ ∈L}. Dimostrato questo ne consegue la veridicità di P\L $\sim_t$ Q\L.

(⊆)

Sia t ∈Tr(P\L) ⇒ t ∈Tr(P)\{$\alpha_1...\alpha_n|\alpha_i$ ∈L}
Per induzione su |t|:

Caso Base |t| = 0

Allora t = $\varepsilon \in$ Tr(P)\{$\alpha_1...\alpha_n|\alpha_i$ ∈L} per definizione.

Caso Induttivo |t| = n + 1

t ha una prima interazione seguita poi dalla traccia t', quindi P\L $\stackrel{\alpha_1}{\rightarrow}$ X' $\stackrel{t'}{\rightarrow}$ ovvero viene applicata una transizione secondo la regola della restrizione arrivando in un certo processo X', quindi:

$$\frac{P \stackrel{\alpha_1}{\rightarrow} P'}{P\backslash L \stackrel{\alpha_1}{\rightarrow} P'\backslash L \stackrel{t'}{\rightarrow}} \; RES \text{ se } \alpha_1, \overline{\alpha_1} \notin \text{L}$$

t = $\alpha_1.t'$ con |t'| = n, per ipotesi induttiva t' ∈ Tr(P')\{$\alpha_1...\alpha_n|\alpha_i$ ∈L}.
Dato che $\alpha_1, \overline{\alpha_1} \notin$ L quindi t = α.t' ∉ {$\alpha_1...\alpha_n|\alpha_i$ ∈L} allora vale che t = $\alpha_1.t' \in$ Tr(P)\{$\alpha_1...\alpha_n|\alpha_i$ ∈L}.

(⊇)

Sia t ∈Tr(P)\{$\alpha_1...\alpha_n|\alpha_i$ ∈L} ⇒ t ∈Tr(P\L).
Per induzione su |t|:

Caso Base |t| = 0

Allora t = $\varepsilon \in$ Tr(P\L) per definizione.

Caso Induttivo |t| = n + 1

t ha una prima interazione seguita poi dalla traccia t', cioè t = α_1.t'. Si ha quindi una transizione P $\stackrel{\alpha_1}{\rightarrow}$ P', ciò è permesso dalla regola della restrizione, quindi:

$$\frac{P \stackrel{\alpha_1}{\rightarrow} P'}{P\backslash L \stackrel{\alpha_1}{\rightarrow} P'\backslash L \stackrel{t'}{\rightarrow}} \; RES \text{ se } \alpha_1, \overline{\alpha_1} \notin \{\alpha_1...\alpha_n|\alpha_i \in \text{L}\}$$

Questo dimostra che P\L sa fare l'interazione α_1, perciò per ipotesi induttiva t'∈P'\L allora α_1.t' ∈ Tr(P\L).

Quindi dato che Tr(P\L) = Tr(P)\{$\alpha_1 ... \alpha_n | \alpha_i$ ∈L} con Tr(P) = Tr(Q) allora si è dimostrato che P\L $\sim_t$ Q\L.

1.2.1.5 Contesto relabelling C[ ] = [f]

Nel caso del contesto relabelling sul processo P si ha che:
Data la funzione $f : Act \rightarrow Act$, le traccie di P[**f**] sono:
f(ϵ) = ϵ
f($\alpha.t$) = f(α).f(t)
Quindi voglio dimostrare che Tr(P[**f**]) = {f(t)| t ∈ Tr(P)}. Se questo è vero, dato che Tr(P) = Tr(Q) si può sostituire Tr(P) con Tr(Q) scrivendo {f(t)| t ∈ Tr(Q)} e grazie alla uguaglianza scritta precedentemente, allora Tr(P[**f**]) = Tr(Q[**f**]).

(⊆)

Sia t∈ Tr(P[**f**]) ⇒ t ∈ {f(t)| t ∈ Tr(P)}.
Per induzione su |t|:

Caso Base |t| = 0

Allora t = ε ∈ {f(ε)| ε ∈ Tr(P)} per definizione.

Caso Induttivo |t| = n + 1

t ha una prima interazione seguita poi dalla traccia t', quindi P[**f**] $\stackrel{\alpha_1}{\rightarrow}$ X' $\stackrel{t'}{\rightarrow}$ ovvero viene applicata una transizione secondo la regola del relabelling arrivando in un certo processo X', quindi:

$$\frac{P \stackrel{\alpha_1}{\rightarrow} P'}{P[\mathbf{f}] \stackrel{f(\alpha_1)}{\rightarrow} P'[\mathbf{f}] \stackrel{f(t')}{\rightarrow}} \; REL$$

t = $f(\alpha_1).f(t)'$ con |t'| = n, per ipotesi induttiva si ha che f(t') ∈ {f(t')| t' ∈ Tr(P')}, allora f(α).f(t') ∈ {f(t)| t ∈ Tr(P)}.

(⊇)

Sia t∈{f(t)| t ∈ Tr(P)} ⇒ t∈Tr(P[**f**]).
Per induzione su |t|:

Caso Base |t| = 0

Allora t = ε ∈ Tr(P[**f**]) per definizione.

Caso Induttivo |t| = n + 1

t ha una prima interazione seguita poi dalla traccia t', cioè t = α_1.t'. Si ha quindi una transizione P $\stackrel{\alpha_1}{\rightarrow}$ P', ciò è permesso dalla regola del relabelling, quindi:

$$\frac{P \overset{\alpha_1}{\to} P'}{P[\mathbf{f}] \overset{f(\alpha_1)}{\to} P'[\mathbf{f}] \overset{f(t')}{\to}} \; REL$$

Questo dimostra che P[**f**] sa fare l'interazione α_1, perciò per ipotesi induttiva t'$\in$ P'[**f**] allora α_1.t' $\in$ Tr(P[**f**]).

Quindi dato che Tr(P[**f**]) = {f(t)| t $\in$ Tr(P)} con Tr(P) = Tr(Q) allora si è dimostrato che P[**f**] $\sim_t$ Q[**f**].

1.2.2 Conclusione

Si è dimostrato con i vari casi della dimostrazione precedente, che per ogni possibile contesto che può essere usato, la trace equivalence risulta essere è una congruenza per il CCS.

2 Barbershop

Il problema del Barbiere formato da 2 tipi di processi, un processo barbiere che effettua tagli di barba/capelli a dei processi clienti, e un insieme di processi clienti che vogliono effettuare un taglio. Il negozio del barbiere prevede l'esistenza di una sala d'attesa con n sedie, e la stanza del barbiere con una sedia dove viene effettuato il taglio. Se non ci sono clienti che aspettano di essere serviti, il barbiere dorme. Se arriva un cliente nel negozio vi sono tre casi possibili; tutte le sedie sono occupate da altri clienti, e quindi il cliente se ne va dal negozio, oppure il barbiere è occupato e c'è almeno una sedia libera nella sala d'attesa, il cliente può sedersi ed aspettare che il barbiere si liberi ed effettui il taglio, infine se il barbiere sta dormendo, il barbiere si sveglierà e effettuerà il taglio al cliente.

É importante rispettare i seguenti vincoli:

- I processi cliente dovrebbero ricevere il taglio;
- Se un cliente arriva quando il negozio è pieno, se ne va;
- Il barbiere dovrebbe effettuare i tagli;
- Il barbiere serve i clienti uno per volta.

2.1 Una possibile soluzione

Il libro Little Book of Semaphores suggerisce la seguente soluzione:

Sia n = 2, quindi una sedia per la sala d'attesa e una per la barberia.

Si vuole utilizzare un mutex per proteggere la sezione critica al cui interno vi è un contattore di clienti presenti nel negozio. Quindi il contattore per essere modificato si dovrà garantire la mutua esclusione. Quando un cliente entra nel negozio dovrà entrare nella sezione critica per controllare che il contattore sia uguale a n, se lo è allora esce dal negozio, se invece non lo è incrementa il contattore ed esce dalla sezione critica. Una volta incrementato il cliente segnala attraverso un semaforo Customer la sua presenza al barbiere e si mette in attesa nel semaforo Barber per attendere il servizio del barbiere.

Una volta che il barbiere segnala la presa in carico del cliente sul semaforo Barber, viene effettuato il taglio e il cliente e il barbiere si sincronizzano sui semafori CustomerDone e BarberDone per garantire che il taglio sia stato fatto e che il barbiere possa effettuare un taglio ad un altro cliente in attesa se c'è altrimenti torna a dormire. Il cliente dopo il taglio entra di nuovo nella sezione critica per decrementare il contattore in modo tale da uscire dal negozio.

Il barbiere nello specifico all'inizio rimane in attesa sul semaforo Customer aspettando l'arrivo di un cliente (simula il fatto che stia dormendo), una volta svegliato, segnala sul semaforo Barber di essere pronto con il taglio e prendere in carico un cliente(il primo che sincronizza con il segnale), effettua il taglio si sincronizza con il cliente che ha ricevuto il servizio. Se ci sono altri clienti in attesa passa direttamente al nuovo lavoro da effettuare senza mettersi a dormire, invece se non c'è nessuno torna a dormire e si ferma sul semaforo Customer.

Di seguito viene mostrata una possibile soluzione in pseudo-codice.

```
mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
mutex.signal()

customer.signal()
barber.wait()

# getHairCut()

customerDone.signal()
barberDone.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

Figura 1: Definizione processo cliente.

```
customer.wait()
barber.signal()

# cutHair()

customerDone.wait()
barberDone.signal()
```

Figura 2: Definizione processo barbiere.

2.2 Modellazione in CCS

Quindi le entità utilizzate nel programma CCS sono:

- $\mathsf{Customer}_i$: Processo cliente che riceve il taglio;
- Count_i: Mutex con al suo interno il contattore del numero di clienti presenti nel negozio;
- Barber: Processo barbiere che effettua il taglio;
- Sys: Sistema.

Di seguito si mostra un esempio del sistema con n = 2 e tre clienti.

Count1 = enter.incExit.Count2;
Count2 = enter.(incExit.CountB + decExit.Count1);
CountB = enter.(balk.CountB + decExit.Count2);

Customer1 ='enter.enter1.exit1.('incExit.C1 + 'balk.Customer1);
C1 = semCustomer.'semBarber.getHairCut1.semCustomerDone.'semBarberDone.
'enter.enter1.exit1.'decExit.Customer1;

Customer2 = 'enter.enter2.exit2.('incExit.C2 + 'balk.Customer2);
C2 = semCustomer.'semBarber.getHairCut2.semCustomerDone.'semBarberDone.
'enter.enter2.exit2.'decExit.Customer2;

Customer3 = 'enter.enter3.exit3.('incExit.C3 + 'balk.Customer3);
C3 = semCustomer.'semBarber.getHairCut3.semCustomerDone.'semBarberDone.

'enter.enter3.exit3.'decExit.Customer3;

Barber = 'semCustomer.semBarber.cutHair.'semCustomerDone.semBarberDone.Barber;

set L = { enter, incExit, decExit, balk, semCustomer, semCustomerDone, semBarber, semBarberDone};

Sys = (Customer1|Customer2|Customer3|Count1|Barber) \L;

2.2.1 Codifica Mutex e contattore clienti

Per codificare il contattore e il *mutex* si è deciso di utilizzare un unico processo, o meglio un insieme di processi che codificano il mutex e il contattore, perciò abbiamo:

Count1 = enter.incExit.Count2;

Codifica il contatore che passa da zero a uno gestendo il tutto in mutua esclusione. Per interagire con il contattore i processi clienti devono sincronizzarsi su enter che risulta essere un canale ristretto, ciò permette la mutua esclusione che sarà dimostrata in seguito. Per poter incrementare il contattore e uscire dalla sezione critica i processi cliente utilizzeranno il canale incExit anch'esso ristretto, mentre il processo Count1 andrà nel processo Count2 per mantenere il contattore a uno e per dare la possibilità di decrementare o incrementare un successivo processo cliente.

Count2 = enter.(incExit.CountB + decExit.Count1);

Codifica il contatore che passa da uno a due e/o da uno a zero, gestendo il tutto in mutua esclusione. Analogamente a Count1 c'è enter per la sincronizzazione in mutua esclusione. Viene data una scelta in più su come proseguire l'esecuzione. A seconda di cosa richiede il processo cliente può essere data la possibilità di incrementare il contattore e uscire, sempre attraverso il canale incExit, oppure il decremento attraverso decExit(canale ristretto). La scelta dipende dello stato d'esecuzione in cui si trova il processo cliente. Nel caso incrementi, Count2 va in CountB altrimenti torna in Count1.

CountB = enter.(balk.CountB + decExit.Count2);

Codifica il contatore che non può essere più incrementato perché uguale a n, e l'azione di decremento. Il funzionamento è uguale a Count2 tranne per il fatto che incExit non esiste ma c'è balk(canale ristretto) usato per simulare l'uscita dal locale del cliente nel caso in cui voglia ricevere un taglio ma non ci sono sedie disponibili.

2.2.2 Codifica Customer

La codifica dei clienti avviene nel seguente modo:

Customer1 ='enter.enter1.exit1.('incExit.C1 + 'balk.Customer1);
C1 = semCustomer.'semBarber.getHairCut1.semCustomerDone.'semBarberDone.
'enter.enter1.exit1.'decExit.Customer1;

Il cliente per poter controllare se può entrare si sincronizza sul canale enter rimando in attesa della possibilità di entrare nella sezione critica. Una volta entrato controlla se può incrementare oppure no il contattore, quindi sceglie se eseguire incExit o back, tale scelta dipende da cosa offre il processo count. Nel caso sia CountB, l'unica azione possibile per il cliente è eseguire balk che lo fa uscire dalla sezione critica e non lo fa entrare nel negozio. Nel caso invece sia Count1 o Count2 il cliente esegue incExit cosi che incrementi il contattore e esca dalla sezione critica. Successivamente si sincronizza con il barbiere, che nel caso in cui il barbiere sia libero passa a ricevere il taglio. Finito il taglio si sincronizza subito con il barbiere per poi entrare nella sezione critica in mutua esclusione e decrementare il contattore per simulare la sua uscita dal negozio.

2.2.3 Codifica Barber

La codifica del Barbiere avviene nel seguente modo:

Barber = 'semCustomer.semBarber.cutHair'semCustomerDone.semBarberDone.Barber;

Il barbiere rimane in attesa di un nuovo cliente da servire su semCustomer se non ci sono cliente in attesa. Una volta arrivato un cliente si sincronizza con esso ed esegue il taglio, successivamente si sincronizza con il cliente appena servito per garantire che sia avvenuto il taglio per poi tornare all'inizio della sua esecuzione per poter eseguire un nuovo cliente se c'é, altrimenti attende un nuovo cliente.

2.2.4 Codifica del sistema

L'intero sistema viene codificato nel seguente modo:

Sys = (Customer1|Customer2|Customer3|Count1|Barber) \L;
set L = { enter, incExit, decExit, balk, semCustomer, semCustomerDone, semBarber, semBarberDone};

Il sistema viene rappresentato attraverso una composizione parallela dei processi Customer, Barber e Count1. Tutti i canali vengono ristretti per permettere la sincronizzazione dei processi, escluso cutHair, getHairCut$_i$, enter$_i$, exit$_i$ usato per mostrare un comportamento all'esterno.

2.3 Verifica della correttezza attraverso CWB

Di seguito si verificano se il programma definito rispetta tutte le caratteristiche stabilite dalla definizione del problema, attraverso l'uso della Edinburgh Concurrency Workbench (CWB).

2.3.1 Trace Equivalence

Dato che il sistema con 3 clienti e n = 2 risulta avere 752 stati, è troppo complesso scrivere una specifica che catturi il comportamento dell'intero sistema. Quindi trovare un specifica che sia bisimile a Sys non è molto interesse quello che ci interessa è perciò catturare alcune caratteristiche intessenti utili per la verifica. Si procede quindi nello scrivere due specifiche generali che catturino, una la mutua esclusione nella modifica del contattore e l'altra che ogni cliente viene servito uno alla volta. Si hanno le seguenti:

SpecM = enter1.exit1.SpecM + enter2.exit2.SpecM + enter3.exit3SpecM;

SpecC = cutHair.getHairCut1.SpecC + getHairCut1.cutHair.SpecC +
cutHair.getHairCut2.SpecC + getHairCut2.cutHair.SpecC +
cutHair.getHairCut3.SpecC + getHairCut3.cutHair.SpecC;

Si nota che unendo nel modo corretto queste due specifiche si può costruire la specifica bisimile a Sys, ma come detto risulta essere troppo complesso e quello che ci interessa è la cattura delle caratteristiche chiave del sistema, le quali garantiscono il corretto funzionamento.

Quindi si prendono due versioni di Sys, Sys1 versione senza i canali cutHair e getHairCut$_i$ e Sys2 versione senza i canali enter$_i$ e exit$_i$.

Per dimostrare che Sys1 e Sys2 hanno le caratteristiche descritte dalle specifiche SpecM e SpecC si ricorre all'uso della Trace Equivalence. Si vuole che le traccie di Sys1 siano Tr(Sys1) = Tr(SpecM), mentre le traccie di Sys2 siano Tr(Sys2) = Tr(SpecC).
Vogliamo perciò che le traccie di Sys1 siano sequenze di enter$_i$ e exit$_i$ ben accoppiate mentre per le traccie di Sys2 siano sequenze di cutHair e getHairCut$_i$ anch'essi ben accoppiate.

Attraverso la CWB con il commando **mayeq(Sys1,SpecM)** otteniamo la conferma che Tr(Sys1) = Tr(SpecM). Analogamente con **mayeq(Sys2,SpecC)** si dimostra che Tr(Sys2) = Tr(SpecC).

Si sottolinea che nonostante Sys1 e Sys2 hanno comportamenti esterni differenti, al loro interno sono garantite le due proprietà che si sono verificate con la Trace Equivalence, dato che si è solo modificato il comportamento esterno che ha origine dalla struttura interna del sistema rimasta in variata.

2.3.2 Verifica tramite HML

Passiamo ora a dimostrare alcune proprietà chiave del Sys attraverso la logica di Hennessy-Milner per una dimostrazione più formale.

Nelle verifiche verranno usate le seguenti formule:

- **Inv(P)** = max(X.(P & [-]X));
  Sempre vera la proprietà P.
- **Pos(P)** = min(X. (P | <-> X));
  Esiste un stato in cui vale la proprietà P.
- **WeakEven(P)** = min(X. (P | «eps»<-tau>«eps»T & [[eps]][-tau][[eps]] X)));
  Prima o poi l'esecuzione eseguirà un stato in cui vale P attraverso passi interni e esterni.
- **WUntil(P,Q)** = max(X. Q | (P & [-]X));
  Until in versione weak. Vale la proprietà P finché non diventa vera Q cioè l'esecuzione arriva un stato dove può vale Q.
- **SUntil(P,Q)** = min(X. Q | (P & <->T & [-]X));
  Until in versione strong. Vale la proprietà P finché non diventa vera Q.

2.3.2.1 Assenza di deadlock

Il sistema riesce sempre a eseguire un passo senza rimanere bloccato.

NoDeadlock = Inv(<->T);

con il commando **checkprop(Sys, Nodeadlock)** si ottiene **true**, quindi il sistema non va mai in deadlock.

2.3.2.2 Presenza di Livelock

Siano:

TauLoop = max (X. <tau> X);
prop Livelock = Pos(TauLoop);

Attreverso **checkprop(Sys, Livelock)** si ottiene **false**, quindi non vi è la presenza di Livelock.

2.3.2.3 Mutua esclusione contattore

L'accesso alla sezione critica che contiene il contattore avviene in mutua esclusione, sia per incrementare e sia per decrementare il contattore. Quindi solo un processo può eseguire exit perchè solo un processo cliente puo trovarsi nella sezione critica.

MutexC = Inv(([[exit1]]F | [[exit2]]F) & ([[exit2]]F | [[exit3]]F) & ([[exit1]]F | [[exit3]]F));

È sempre vero che al più solo uno dei tre processi è in grado di fare exit_i. Con il commando **checkprop(Sys, MutexC)** si ottiene **true**, quindi vale la mutua esclusione.

2.3.2.4 Mutua esclusione nell'esecuzione del taglio

Ogni cliente viene servito uno alla volta, quindi non e possibile che due clienti vengono serviti contemporaneamente dal barbiere ma al più solo uno. Si dimostra perciò la mutua esclusione nell'esecuzione del taglio.

MutexB = Inv(([[getHairCut1]]F | [[getHairCut2]]F) & ([[getHairCut2]]F |
[[getHairCut3]]F) & ([[getHairCut1]]F | [[getHairCut3]]F));

È sempre vero che al più solo uno dei tre processi è in grado di fare $\mathsf{getHairCut}_i$. Con il commando **checkprop(Sys, MutexB)** si ottiene **true**, quindi vale la mutua esclusione.

2.3.2.5 Verifica comportamento del barbiere nell'attesa dell'arrivo di un nuovo cliente

Il barbiere aspetta, cioè dorme, finché non entra un cliente. Quindi a livello di codice CCS il processo Barber rimane fermo in 'semCustomer finché non si sincronizza con un processo $\mathsf{Customer}_i$ attraverso l'azione semCustomer, dopo di che Barber può proseguire la sua esecuzione.

Per dimostrare tale proprietà si inseriscono nel sistema Sys, i canali entered in tutti i processi C_i, quindi:

C1 = semCustomer.entered.'semBarber.getHairCut1.semCustomerDone.'semBarberDone.
'enter.enter1.exit1.'decExit.Customer1;

Mentre nel processo Barber si inserisce waked, quindi:

Barber = 'semCustomer.waked.semBarber.cutHair'semCustomerDone.semBarberDone.Barber;

UntilB = Inv(WUntil([[waked]]F, «entered»T));

È sempre vero che il processo Barber non sa fare waked finché un processo $\mathsf{Customer}_i$ può fare entered. La possibilità di poter fare entered la si ha solo se c'è stata una sincronizzazione, cioè il barbiere si è svegliato e il cliente è entrato e successivamente può segnalarlo. Quindi eseguendo **checkprop(Sys, UntilB)** si ottiene **true**, vale perciò la proprietà.

Non sarebbe andata bene la versione con lo strong until perché non è sempre vero che il processo Barber non sa fare waked finché un processo $\mathsf{Customer}_i$ esegue entered, infatti:

UntilB = Inv(SUntil([[waked]]F, «entered»T));

Allora **checkprop(Sys, UntilB)** ritorna **false**.

2.3.2.6 Verifica comportamento del cliente nell'attesa di essere servito

Il cliente aspetta, finché il barbiere non gli da il segnale di sedersi per il taglio. Quindi a livello di codice CCS il processo $\mathsf{Customer}_i$ rimane fermo in 'semBarber finché non si sincronizza con un processo Barber attraverso l'azione semBarber, dopo di che $\mathsf{Customer}_i$ può proseguire la sua esecuzione.

$$\begin{aligned}\mathsf{UntilC} = \mathsf{Inv}(&\mathsf{WUntil}([[\mathsf{getHairCut1}]]\mathsf{F},\ «\mathsf{cutHair}»\mathsf{T})\\ &|\ \mathsf{WUntil}([[\mathsf{getHairCut2}]]\mathsf{F},\ «\mathsf{cutHair}»\mathsf{T})\\ &|\ \mathsf{WUntil}([[\mathsf{getHairCut3}]]\mathsf{F},\ «\mathsf{cutHair}»\mathsf{T}));\end{aligned}$$

È sempre vero che il processo $\mathsf{Customer}_i$ non sa fare $\mathsf{getHairCut}_i$ finché un processo Barber può fare cutHair. La possibilità di poter fare cutHair la si ha solo se c'è stata una sincronizzazione. Si è aggiunta la clausola OR perché può accadere che nonostante si abbia la possibilità di poter fare cutHair non è detto che il processo $\mathsf{Customer}_i$ sa fare $\mathsf{getHairCut}_i$ perché a causa della mutua esclusione dopo la sincronizzazione al più solo uno sa fare l'azione. Quindi eseguendo **checkprop(Sys, UntilC)** si ottiene **true**, vale perciò la proprietà.

2.3.2.7 Fairness

Purtroppo la soluzione data al problema non garantisce una piena fairness, cioè può accadere che un processo $\mathsf{Customer}_j$ prenda per molte volte possesso del Barber lasciando gli altri processi $\mathsf{Customer}_{k\neq j}$ in attesa di un taglio. Vi è quindi solo garantita la possibilità che si possa ricevere il taglio. Per dimostrare ciò si aggiunge ad ogni processo $\mathsf{Customer}_i$ si aggiunge il canale will_iper esprimere la volontà di effettuare un taglio, quindi:

Customer1 = will.'enter.enter1.exit1.('incExit.C1 + 'balk.Customer1);

Sia la seguente formula:

$$\begin{aligned}\mathsf{FairC} = \mathsf{Inv}((&[\mathsf{will1}]\ \mathsf{Pos}(«\mathsf{getHairCut1}»\mathsf{T}))\\ &\&\ ([\mathsf{will2}]\ \mathsf{Pos}(«\mathsf{getHairCut2}»\mathsf{T}))\\ &\&\ ([\mathsf{will3}]\ \mathsf{Pos}(«\mathsf{getHairCut3}»\mathsf{T})));\end{aligned}$$

Una volta espressa la volontà di eseguire un taglio esiste uno stato in cui è possibile effettuarlo. Quindi eseguendo **checkprop(Sys, FairC)** si ottiene **true**, esiste perciò uno stato in cui è possibile eseguire il taglio. Non è detto pero che venga eseguito questo, infatti:

$$\begin{aligned}\mathsf{FairC2} = \mathsf{Inv}((&[\mathsf{will1}]\ \mathsf{WeakEven}(«\mathsf{getHairCut1}»\mathsf{T}))\\ &\&\ ([\mathsf{will2}]\ \mathsf{WeakEven}(«\mathsf{getHairCut2}»\mathsf{T}))\\ &\&\ ([\mathsf{will3}]\ \mathsf{WeakEven}(«\mathsf{getHairCut3}»\mathsf{T})));\end{aligned}$$

È sempre vero che dopo aver espresso la volonta di eseguire un taglio attraverso will_j prima o poi il processo $\mathsf{Customer}_j$ lo riceverà. Ma per checkprop(Sys, FairC2) risulta essere **false**, e quindi vale ciò che si è detto precedentemente, non vale la fairness.

Vi è garantita pero che non ci sia la possibilità di attesa infinita per ogni processo $\mathsf{Customer}_i$, infatti siano:

$$\begin{aligned}&\mathsf{WaitForever1} = \mathsf{max}\ (\mathsf{X}.\ «\mathsf{getHairCut1}»\ \mathsf{X});\\ &\mathsf{WaitForever2} = \mathsf{max}\ (\mathsf{X}.\ «\mathsf{getHairCut2}»\ \mathsf{X});\\ &\mathsf{WaitForever3} = \mathsf{max}\ (\mathsf{X}.\ «\mathsf{getHairCut3}»\ \mathsf{X});\end{aligned}$$

$$\mathsf{WaitForeverWill} = (\mathsf{Pos}(«\mathsf{will1}»\ \mathsf{WaitForever1}) \mid (\mathsf{Pos}(«\mathsf{will2}»\ \mathsf{WaitForever2})) \mid (\mathsf{Pos}(«\mathsf{will3}»\ \mathsf{WaitForever3}))\ );$$

Attraverso **checkprop(Sys, WaitForeverWill)** si ottiene **false**, quindi non vi è la possibilità di avere attesa infinita nel ricevere il taglio.

Si può ragionare sul fatto che è possibile modificare il sistema in modo che sia garantita la fairness.

Un modo soluzione diciamo "semplice" può essere quella di far terminare i processi$\mathsf{Customer}_i$ una volta effettuato un taglio, e quando sono terminati tutti ci sia un processo preposto a riattivarli tutti. Questo modifica garantisce che prima o poi tutti i processi $\mathsf{Customer}_i$ eseguano un taglio.

Una soluzione più raffinata può essere quella di impostare un ordine d'esecuzione tale da garantire che tutti eseguano un taglio, quindi si potrebbe realizzare una sorta di coda FIFO in modo tale da stabilire un ordine d'esecuzione dando a tutti i clienti un turno in cui verrà eseguito il taglio.