Università degli studi di Padova

Dipartimento di Matematica 'Tullio Levi-Civita'

Corso di Laurea Magistrale in Informatica

Linguaggi per il Global Computing

Esercizi & Progetto

Tommaso Sgarbanti - matr. 1185058

Indice

1 Esercizio D

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

1.1 Definizione di trace equivalence

Definita una traccia di un processo P come una sequenza $\alpha_1 \cdots \alpha_k \in \mathbf{Act}^*$ $(k \geq 0)$ tale che esiste una sequenza di transizioni $P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} P_2 \cdots P_{k-1} \xrightarrow{\alpha_k} P_k$, per qualche $P_1, \ldots, P_k$, scriviamo $Traces(P)$ per la collezione di tutte le tracce di P .

Due processi P e Q si dicono trace equivalent, indicato con $P \sim_{trace} Q$, sse $Traces(P) = Traces(Q)$.

1.2 Definizione di congruenza

La congruenza è una proprietà desiderabile quando si parla di relazioni di equivalenza. Significa che processi legati da una relazione di equivalenza R devono poter essere intercambiati come parte di un processo più grande senza che tale cambiamento influisca sul suo comportamento generale.

Più precisamente, dati due processi $P\,R\,Q$, R è una congruenza sse:

$\forall\, C[\,]\ ,\quad C[P]\,R\,C[Q]\ .$

1.3 Formalizzazione del problema

Dati P, Q ed R processi CCS e assumendo $P \sim_{trace} Q$, si dimostri che:

- $\alpha.P \sim_{trace} \alpha.Q$, per ogni azione α;
- $P + R \sim_{trace} Q + R$ e $R + P \sim_{trace} R + Q$, per ogni processo R;
- $P \,|\, R \sim_{trace} Q \,|\, R$ e $R \,|\, P \sim_{trace} R \,|\, Q$, per ogni processo R;
- $P[f] \sim_{trace} Q[f]$, per ogni relabelling f;
- $P \setminus L \sim_{trace} Q \setminus L$, per ogni insieme di etichette L.

1.4 Dimostrazione

La dimostrazione procede per induzione sui casi base sopra definiti:
dati P, Q ed R processi CCS l'ipotesi è $P \sim_{trace} Q$.

Contesto prefixing: $C[\,] = \alpha._{\_}$

La tesi da dimostrare è:

$\alpha.P \sim_{trace} \alpha.Q$, per ogni azione α .

Data l'ipotesi abbiamo che $Traces(P) = Traces(Q)$, ciò che è necessario verificare è che $Traces(\alpha.P) = \alpha.Traces(P)$, ma è proprio così, infatti:

$Traces(\alpha.P) = \{a.t \,|\, t \in Traces(P)\} = \alpha.Traces(P)$.

Analogalmente $Traces(\alpha.Q) = \alpha.Traces(Q)$.

Dunque se $Traces(P) = Traces(Q)$, allora $\alpha.Traces(P) = \alpha.Traces(Q)$, che si traduce in $Traces(\alpha.P) = Traces(\alpha.Q)$.

Ma se $Traces(\alpha.P) = Traces(\alpha.Q)$, allora
$\alpha.P \sim_{trace} \alpha.Q$, che è proprio la tesi.

Contesto choice: $C[\,] = \_ + R$ e $C[\,] = R + \_$

La tesi da dimostrare è:

$P + R \sim_{trace} Q + R$ e $R + P \sim_{trace} R + Q$,per ogni processo R .

Dimostriamo prima che $Traces(P + R) = Traces(P) \cup Traces(R)$:

($\subseteq$): L'ipotesi è $T = Traces(P + R)$, la tesi da dimostrare è $\forall t \in T$, $t \in Traces(P) \cup Traces(R)$.

Per la traccia vuota $t = \epsilon$ banalmente appartiene sia all'insieme delle tracce di P che di Q e dunque appartiene anche all'insieme dato dall'unione dei due.

Per tracce non vuote distinguiamo due diversi sottocasi, uno per ciascuna possibile transizione:

- $P + R \xrightarrow{\alpha} P'$ con la regola **(SUML)**:

$$\frac{P \xrightarrow{\alpha} P'}{P + R \xrightarrow{\alpha} P'} \quad .$$

  Pertanto $t \in \alpha.Traces(P')$, con $\alpha.Traces(P') \subseteq Traces(P)$, di conseguenza $t \in Traces(P) \implies t \in Traces(P) \cup Traces(R)$.
- $P + Q \xrightarrow{\alpha} R'$ con la regola **(SUMR)**:

$$\frac{R \xrightarrow{\alpha} R'}{P + R \xrightarrow{\alpha} R'} \quad .$$

  Pertanto $t \in \alpha.Traces(R')$, con $\alpha.Traces(R') \subseteq Traces(R)$, di conseguenza $t \in Traces(R) \implies t \in Traces(P) \cup Traces(R)$.

($\supseteq$): L'ipotesi è $T = Traces(P) \cup Traces(R)$, la tesi da dimostrare è $\forall t \in T$, $t \in Traces(P + R)$.

Per la traccia vuota $t = \epsilon$ banalmente appartiene all'insieme delle tracce di $P + R$.

Se una traccia, non vuota, $t \in Traces(P) \cup Traces(R)$ allora ci sono tre possibilità:

- $t \in Traces(P)$, allora in questo caso $P + R$ può effettuare una transizione grazie alla regola **(SUML)** $\implies t \in Traces(P + R)$.
- $t \in Traces(R)$, allora in questo caso $P + R$ può effettuare una transizione grazie alla regola **(SUMR)** $\implies t \in Traces(P + R)$.
- $t \in Traces(P)$ e $t \in Traces(R)$, allora in questo caso $P + R$ può effettuare una transizione grazie ad una regola tra **(SUML)** e **(SUMR)** $\implies t \in Traces(P + R)$.

Una volta dimostrato che $Traces(P + R) = Traces(P) \cup Traces(R)$, si ha che:

$P \sim_{trace} Q \implies Traces(P) = Traces(Q) \implies$
$Traces(P) \cup Traces(R) = Traces(Q) \cup Traces(R) \implies$
$Traces(P + R) = Traces(Q + R) \implies$
$P + R \sim_{trace} Q + R$ (e $R + P \sim_{trace} R + Q$) ,

che è proprio la tesi.

Contesto parallel composition: $C[\,] = \_ \,|\, R$ e $C[\,] = R \,|\, \_$

La tesi da dimostrare è:

$P \,|\, R \sim_{trace} Q \,|R$ e $R \,|\, P \sim_{trace} R \,|Q$, per ogni processo R .

E' necessario innanzitutto definire l'insieme $Traces(P \,|\, R)$, ma essendo possibile anche la sincronia tra i due processi è necessario prima definire un operatore ricorsivo che, prendendo in input due tracce, ne scorre le singole azioni considerando così anche le possibili sincronizzazioni. Utilizziamo il simbolo $|_t$ per riferirci a tale operatore e, indicando con t_1 e t_2 generiche tracce e con α e β azioni qualsiasi, definiamolo dunque,sfruttando il pattern matching, nel modo seguente:

$\epsilon \,|_t\, t_1 = \{t_1\}$
$t_1 \,|_t\, \epsilon = \{t_1\}$
$\alpha.t_1 \,|_t\, \overline{\alpha}.t_2 = \tau.(t_1 \,|_t\, t_2) \cup \alpha.(t_1 \,|_t\, \overline{\alpha}.t_2) \cup \overline{\alpha}.(\alpha.t_1 \,|_t\, t_2)$
$\alpha.t_1 \,|_t\, \beta.t_2 = \alpha.(t_1 \,|_t\, \beta.t_2) \cup \beta.(\alpha.t_1 \,|_t\, t_2)$

Grazie a questo operatore è possibile definire l'insieme delle tracce possibili di $P \,|\, R$ come:

$$Traces(P \,|\, R) = \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2$$

Dimostriamo che tale equivalenza valga:

($\subseteq$): L'ipotesi è $T = Traces(P \,|\, R)$, la tesi da dimostrare è $\forall\, t \in T$, $t \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2$.
Dimostrazione per induzione sulla lunghezza della traccia t :

Caso base: (Tracce t di lunghezza $k = 0$)
Dato $|t| = 0$, allora $t = \epsilon$.
Si osservi che $(t = \epsilon) \in (\epsilon \,|_t\, \epsilon = \{\epsilon\})$ e che $\epsilon \in Traces(P) \cap Traces(R)$, pertanto $\epsilon \,|_t\, \epsilon \subseteq \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2$.
Di conseguenza abbiamo che:

$$t = \epsilon \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2 \ ,$$

che è proprio la tesi.

Ipotesi Induttiva:
Data una traccia t di lunghezza al più k, se $t \in Traces(P \,|\, R)$, allora $t \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2$.

Caso induttivo: (Tracce t di lunghezza $k+1$) In questo caso $P \,|\, R$ può evolvere seguendo una delle tre possibili regole: **(PARL)**, **(PARR)** e **(PAR)** .
Si procede dunque per casi:

- $P \,|\, R \xrightarrow{\alpha} P^{'} \,|\, R$ con la regola **(PARL)**:

$$\frac{P \xrightarrow{\alpha} P^{'}}{P \,|\, R \xrightarrow{\alpha} P^{'} \,|\, R} \quad .$$

In questo caso $t = \alpha.t^{'}$, con $\alpha \in Traces(P)$ e $t^{'} \in Traces(P^{'} \,|\, R)$.
Essendo $|t^{'}| \leq k$, è possibile applicare l'ipotesi induttiva risultando che:

$$t^{'} \in \bigcup_{t_1 \in Traces(P^{'})} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2 \implies$$
$$\exists t_1^{'} \in Traces(P^{'}),\ t_2^{'} \in Traces(R) \text{ tale che } t^{'} \in t_1^{'} \,|_t\, t_2^{'} \ .$$

Poniamo dunque:

$$t_1 = \alpha.t_1^{'} \in Traces(P) \text{ e } t_2 = t_2^{'} \in Traces(R) \ .$$

Calcolando $t_1 \,|_t t_2 = \alpha.t_1^{'} \,|_t\, t_2^{'}$ abbiamo due casi principali:

1. se $t_2^{'} = \epsilon$:

   $$\alpha.t_1^{'} \,|_t\, \epsilon = \{\alpha.t_1^{'}\} \ ;$$

2. altrimenti:

   $\alpha.t_1^{'} \,|_t\, t_2^{'} =$ UNION , dove

   $$\text{UNION} \supseteq \alpha.(t_1^{'} \,|_t\, t_2^{'}) \ .$$

Nel secondo caso la tesi

$$t = \alpha.t^{'} \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2$$

risulta automaticamente vera in quanto, come scritto sopra, $t^{'} \in t_1^{'} \,|_t\, t_2^{'}$ per ipotesi induttiva; nel primo caso è invece verificata in quanto $t^{'} \in t_1^{'} \,|_t\, \epsilon = \{t_1^{'}\} \implies t^{'} = t_1^{'}$.

– $P \,|\, R \xrightarrow{\alpha} P \,|\, R^{'}$ con la regola **(PARR)**: (Duale al precedente)

$$\frac{R \xrightarrow{\alpha} R^{'}}{P \,|\, R \xrightarrow{\alpha} P \,|\, R^{'}} \quad .$$

In questo caso $t = \alpha.t^{'}$, con $\alpha \in Traces(R)$ e $t^{'} \in Traces(P \,|\, R^{'})$.
Essendo $|t^{'}| \leq k$, è possibile applicare l'ipotesi induttiva risultando che:

$t^{'} \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R^{'})} t_1 \,|_t\, t_2 \implies$
$\exists t_1^{'} \in Traces(P)$, $t_2^{'} \in Traces(R^{'})$ tale che $t^{'} \in t_1^{'} \,|_t\, t_2^{'}$.

Poniamo dunque:

$t_1 = t_1^{'} \in Traces(P)$ e $t_2 = \alpha.t_2^{'} \in Traces(R)$.

Calcolando $t_1 \,|_t t_2 = t_1^{'} \,|_t\, \alpha.t_2^{'}$ abbiamo due casi principali:

1. se $t_1^{'} = \epsilon$:

   $\epsilon \,|_t\, \alpha.t_2^{'} = \{\alpha.t_2^{'}\}$;

2. altrimenti:

   $t_1^{'} \,|_t\, \alpha.t_2^{'} =$ UNION , dove
   UNION $\supseteq \alpha.(t_1^{'} \,|_t\, t_2^{'})$.

Nel secondo caso la tesi

$t = \alpha.t^{'} \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2$

risulta automaticamente vera in quanto, come scritto sopra, $t^{'} \in t_1^{'} \,|_t\, t_2^{'}$ per ipotesi induttiva; nel primo caso è invece verificata in quanto $t^{'} \in \epsilon \,|_t\, t_2^{'} = \{t_2^{'}\} \implies t^{'} = t_2^{'}$.

– $P \,|\, R \xrightarrow{\tau} P^{'} \,|\, R^{'}$ con la regola **(PAR)**: (Analogo al precedente)

$$\frac{P \xrightarrow{\alpha} P^{'} \quad R \xrightarrow{\overline{\alpha}} R^{'}}{P \,|\, R \xrightarrow{\tau} P^{'} \,|\, R^{'}} \quad .$$

In questo caso $t = \tau.t^{'}$, con $\alpha \in Traces(P)$, $\overline{\alpha} \in Traces(R)$ e $t^{'} \in Traces(P^{'} \,|\, R^{'})$.
Essendo $|t^{'}| \leq k$, è possibile applicare l'ipotesi induttiva risultando che:

$t^{'} \in \bigcup_{t_1 \in Traces(P^{'})} \bigcup_{t_2 \in Traces(R^{'})} t_1 \,|_t\, t_2 \implies$
$\exists t_1^{'} \in Traces(P^{'})$, $t_2^{'} \in Traces(R^{'})$ tale che $t^{'} \in t_1^{'} \,|_t\, t_2^{'}$.

Poniamo dunque:

$t_1 = \alpha.t_1^{'} \in Traces(P)$ e $t_2 = \overline{\alpha}.t_2^{'} \in Traces(R)$,

ottenendo che:

$t_1 \,|_t t_2 = \alpha.t_1^{'} \,|_t \overline{\alpha} t_2^{'} =$
$\tau.(t_1^{'} \,|_t t_2^{'}) \cup \alpha.(t_1^{'} \,|_t \overline{\alpha}.t_2^{'}) \cup \overline{\alpha}.(\alpha.t_1^{'} \,|_t t_2^{'})$.

Come detto prima, $t^{'} \in t_1^{'} \,|_t t_2^{'}$ per ipotesi induttiva, dunque:

$t = \tau.t^{'} \in \tau.(t_1^{'} \,|_t t_2^{'}) \cup \alpha.(t_1^{'} \,|_t \overline{\alpha}.t_2^{'}) \cup \overline{\alpha}.(\alpha.t_1^{'} \,|_t t_2^{'})$.

La tesi

$t = \tau.t^{'} \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t t_2$

è dunque verificata.

($\supseteq$): L'ipotesi è $T = \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t t_2$, la tesi da dimostrare è $\forall\, t \in T,\, t \in Traces(P \,|\, R)$.

Dimostrazione per induzione sulla lunghezza della traccia t :

Caso base: (Tracce t di lunghezza $k = 0$)
Dato $|t| = 0$, allora $t = \epsilon$.
La tesi è verificata dato che $\epsilon \in Traces(P \,|\, R)$

Ipotesi Induttiva:
Data una traccia t di lunghezza al più k, se
$t \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t t_2$, allora $t \in Traces(P \,|\, R)$.

Caso induttivo: (Tracce t di lunghezza $k+1$)
In questo caso t può essere della forma: $t = \alpha.t^{'}$, con $t_1 = \alpha.t_1^{'}$; $t = \beta.t^{'}$, con $t_2 = \beta.t_2^{'}$; $t = \tau.t^{'}$, con $t_1 = \alpha.t_1^{'}$ e $t_2 = \overline{\alpha}.t_2^{'}$, dove $t_1 \,|_t t_2 = t$, $t_1 \in Traces(P)$ e $t_2 \in Traces(R)$.
Distinguiamo dunque i casi sopra citati:

- $t = \alpha.t^{'}$:
  In questo caso $P \,|\, R \xrightarrow{\alpha} P^{'} \,|\, R$ con la regola **(PARL)**, con $\alpha \in Traces(P)$ e $t^{'} \in \bigcup_{t_1^{'} \in Traces(P^{'})} \bigcup_{t_2 \in Traces(R)} t_1^{'} \,|_t t_2$.
  In quanto $|t^{'}| \leq k$ è possibile applicarci l'ipotesi induttiva risultando che: $t^{'} \in Traces(P^{'} \,|\, R)$.
  Dato che $P \,|\, R \xrightarrow{\alpha} P^{'} \,|\, R$ e dato che $t^{'} \in Traces(P^{'} \,|\, R)$, allora $t = \alpha.t^{'} \in Traces(P \,|\, R)$, che è proprio la tesi.
- $t = \beta.t^{'}$: (Duale al precedente)
  In questo caso $P \,|\, R \xrightarrow{\alpha} P \,|\, R^{'}$ con la regola **(PARR)**, con $\beta \in Traces(R)$ e $t^{'} \in \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2^{'} \in Traces(R^{'})} t_1 \,|_t t_2^{'}$.
  In quanto $|t^{'}| \leq k$ è possibile applicarci l'ipotesi induttiva risultando che: $t^{'} \in Traces(P \,|\, R^{'})$.
  Dato che $P \,|\, R \xrightarrow{\alpha} P \,|\, R^{'}$ e dato che $t^{'} \in Traces(P \,|\, R^{'})$, allora $t = \beta.t^{'} \in Traces(P \,|\, R)$, che è proprio la tesi.

– $t = \tau.t^{'}$: (Analogo al precedente)
In questo caso $P \,|\, R \xrightarrow{\tau} P^{'} \,|\, R^{'}$ con la regola **(PAR)**, con $\alpha \in Traces(P)$, $\beta \in Traces(R)$ (con $\beta = \overline{\alpha}$) e $t^{'} \in \bigcup_{t_1^{'} \in Traces(P')} \bigcup_{t_2^{'} \in Traces(R')} t_1^{'} \,|_t\, t_2^{'}$.
In quanto $|t^{'}| \leq k$ è possibile applicarci l'ipotesi induttiva risultando che: $t^{'} \in Traces(P^{'} \,|\, R^{'})$.
Dato che $P \,|\, R \xrightarrow{\tau} P^{'} \,|\, R^{'}$ e dato che $t^{'} \in Traces(P^{'} \,|\, R^{'})$, allora $t = \tau.t^{'} \in Traces(P \,|\, R)$, che è proprio la tesi.

Una volta dimostrato che:

$$Traces(P \,|\, R) = \bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2 \ ,$$

abbiamo che:

$$P \sim_{trace} Q \implies Traces(P) = Traces(Q) \implies$$
$$\bigcup_{t_1 \in Traces(P)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2 = \bigcup_{t_1 \in Traces(Q)} \bigcup_{t_2 \in Traces(R)} t_1 \,|_t\, t_2 \implies$$
$$Traces(P \,|\, R) = Traces(Q \,|\, R) \implies$$
$$P \,|\, R \sim_{trace} Q \,|\, R \text{ (e } R \,|\, P \sim_{trace} R \,|\, Q) \ ,$$

che è proprio la tesi.

Contesto relabelling: $C[\ ] = \_[f]$

La tesi da dimostrare è $P[f] \sim_{trace} Q[f]$, per ogni relabelling f .

La funzione di relabelling $f : \text{Act} \to \text{Act}$ è definita come segue:

$$f(\tau) = \tau$$

$$f(\overline{\alpha}) = \overline{f(a)} \text{ per ogni etichetta } \alpha \ .$$

Dimostriamo che $Traces(P[f]) = Traces(Q[f])$:

($\subseteq$): L'ipotesi è $T = Traces(P[f])$, la tesi da dimostrare è $\forall\, t \in T$, $t \in Traces(Q[f])$.
Per ogni traccia $t \in Traces(P)$,$t = \alpha_i.\alpha_{i+1}.\cdots.\alpha_{i+k}$, dove $i, k \geq 0$.
Data f la funzione di relabelling applicata a P, si ha che:

$$\forall t \in Traces(P) \implies$$
$$t^{'} \in Traces(P[f]) \text{ , dove } t^{'} = f(\alpha_i).f(\alpha_{i+1}).\cdots.f(\alpha_{i+k}) \text{ e } i, k \geq 0 \ .$$

Essendo $t \in Traces(P)$ e, per ipotesi, $Traces(P) = Traces(Q)$, allora $t \in Traces(Q)$ e, conseguentemente, $t^{'} \in Traces(Q[f])$, che è proprio la tesi.

($\supseteq$): L'ipotesi è $T = Traces(Q[f])$, la tesi da dimostrare è $\forall\, t \in T$, $t \in Traces(P[f])$. (Duale al precedente)

Per ogni traccia $t \in Traces(Q)$, $t = \alpha_i.\alpha_{i+1}.\cdots.\alpha_{i+k}$, dove i, $k \geq 0$.
Data f la funzione di relabelling applicata a Q, si ha che:

$\forall t \in Traces(Q) \implies$

$t^{'} \in Traces(Q[f])$, dove $t^{'} = f(\alpha_i).f(\alpha_{i+1}).\cdots.f(\alpha_{i+k})$ e i, $k \geq 0$.

Essendo $t \in Traces(Q)$ e, per ipotesi, $Traces(P) = Traces(Q)$, allora $t \in Traces(P)$ e, conseguentemente, $t^{'} \in Traces(P[f])$, che è proprio la tesi.

Si è dunque dimostrato che $Traces(P[f]) = Traces(Q[f])$, il che implica ($\implies$) che $P[f] \sim_{trace} Q[f]$, che è proprio la tesi.

Contesto restriction: $C[\,] = \_ \setminus L$

La tesi da dimostrare è:

$P \setminus L \sim_{trace} Q \setminus L$, per ogni $L \in \mathcal{L}$.

Dimostriamo prima che

$Traces(P \setminus L) = Traces(P) \backslash A$, con $A = \{t | \alpha \in L \wedge \alpha \in t\}$:

($\subseteq$): L'ipotesi è $T = Traces(P \setminus L)$, la tesi da dimostrare è $\forall\, t \in T$, $t \in Traces(P) \backslash A$.

Dimostrazione per induzione sulla lunghezza della traccia t :

Caso base: (Tracce t di lunghezza $k = 0$)
Dato $|t| = 0$, allora $t = \epsilon \in Traces(P \setminus L)$.
La tesi è verificata dato che $\epsilon \in Traces(P) \backslash A$.

Ipotesi Induttiva:
Data una traccia t di lunghezza al più k, se $t \in Traces(P \setminus L)$, allora $t \in Traces(P) \backslash A$.

Caso induttivo: (Tracce t di lunghezza $k+1$)
In questo caso $t = \alpha.t^{'}$, dove $\alpha \in$ Act .
Per ipotesi $t = \alpha.t^{'} \in Traces(P \backslash L)$, con $P \backslash L$ che può procedere solo con la regola **(RES)**:

$$\frac{P \xrightarrow{\alpha} P^{'}}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha, \overline{\alpha} \notin L \qquad .$$

La side-condiction impone $\alpha, \overline{\alpha} \notin L$, ma α , $\overline{\alpha} \notin L \implies \alpha, \overline{\alpha} \notin A$.
Inoltre, per la premessa della regola **(RES)**, abbiamo che $\alpha \in Traces(P)$ e dato che $|t^{'}| \leq k$, è possibile applicarci l'ipotesi induttiva: $t^{'} \in Traces(P^{'} \setminus L) \implies t^{'} \in Traces(P^{'}) \backslash A$.
Considerando che $\alpha \in Traces(P)$, $\alpha \notin A$ e $P \xrightarrow{\alpha} P^{'}$ e dato che $t^{'} \in Traces(P^{'}) \backslash A$, allora $t = \alpha.t^{'} \in Traces(P) \backslash A$, che è proprio la tesi.

($\supseteq$): L'ipotesi è $T = Traces(P)\backslash A$, la tesi da dimostrare è $\forall\, t \in T,\ t \in Traces(P \setminus L)$.

Dimostrazione per induzione sulla lunghezza della traccia t :

Caso base: (Tracce t di lunghezza $k = 0$)

Dato $|t| = 0$, allora $t = \epsilon \in Traces(P)\backslash A$.

La tesi è verificata dato che $\epsilon \in Traces(P \setminus L)$.

Ipotesi Induttiva:

Data una traccia t di lunghezza al più k, se $t \in Traces(P)\backslash A$, allora $t \in Traces(P \setminus L)$.

Caso induttivo: (Tracce t di lunghezza $k+1$)

In questo caso $t = \alpha.t^{'}$, dove $\alpha \in$ Act .

Per ipotesi, $t = \alpha.t^{'} \in Traces(P)\backslash A$, ma quindi anche $\alpha \in Traces(P)\backslash A$.

Dato che $A = \{t | \alpha \in L \wedge \alpha \in t\}$,

$\alpha \in Traces(P)\backslash A \implies \alpha \notin A \implies \alpha \notin L$,

quindi $P \setminus L$ può procedere con l'azione α grazie alla regola **(RES)**, infatti $\alpha \in Traces(P)$ rispetta la premessa della regola e α, $\overline{\alpha} \notin L$, side-condition della regola, è rispettata anch'essa.

Abbiamo dunque osservato che $\alpha \in Traces(P \setminus L)$ e che $P \setminus L \xrightarrow{\alpha} P^{'} \setminus L$; dato che $|t^{'}| \leq k$ è possibile applicare l'ipotesi induttiva: $t^{'} \in Traces(P^{'})\backslash A \implies t^{'} \in Traces(P^{'} \setminus L)$.

Se $\alpha \in Traces(P \setminus L)$, $P \setminus L \xrightarrow{\alpha} P^{'} \setminus L$ e $t^{'} \in Traces(P^{'} \setminus L)$, allora $t = \alpha.t^{'} \in Traces(P \setminus L)$, che è proprio la tesi.

Si è dunque dimostrato che $Traces(P \setminus L) = Traces(P)\backslash A$, con $A = \{t | \alpha \in L \wedge \alpha \in t\}$; dunque:

$$
\begin{aligned}
&P \sim_{trace} Q \implies Traces(P) = Traces(Q) \implies \\
&Traces(P)\backslash A = Traces(Q)\backslash A \implies \\
&Traces(P \setminus L) = Traces(Q \setminus L) \implies \\
&P \setminus L \sim_{trace} Q \setminus L \ ,
\end{aligned}
$$

che è proprio la tesi.

□

2 Esercizio U

Mostrare la coincidenza di string bisimilarity e strong bisimilarity.

2.1 Definizione di string bisimulation

Una relazione binaria R sull'insieme di stati di un LTS è una string bisimulation sse ogni volta che $s_1 R s_2$ e σ è una sequenza di azioni in Act :

– se $s_1 \xrightarrow{\sigma} s_1^{'}$, allora esiste una transizione $s_2 \xrightarrow{\sigma} s_2^{'}$ tale che $s_1^{'} R s_2^{'}$;

– se $s_2 \xrightarrow{\sigma} s_2^{'}$, allora esiste una transizione $s_1 \xrightarrow{\sigma} s_1^{'}$ tale che $s_1^{'} R s_2^{'}$.

Due stati s e $s^{'}$ sono string bisimilar, scritto $s \sim_{string} s^{'}$, sse esiste una string bisimulation che li collega.

2.2 Definizione di strong bisimulation

Una relazione binaria R sull'insieme di stati di un LTS è una strong bisimulation sse ogni volta che $s_1 R s_2$ e α è un'azione:

– se $s_1 \xrightarrow{\sigma} s_1^{'}$, allora esiste una transizione $s_2 \xrightarrow{\alpha} s_2^{'}$ tale che $s_1^{'} R s_2^{'}$;

– se $s_2 \xrightarrow{\sigma} s_2^{'}$, allora esiste una transizione $s_1 \xrightarrow{\alpha} s_1^{'}$ tale che $s_1^{'} R s_2^{'}$.

Due stati s e $s^{'}$ sono strong bisimilar, scritto $s \sim s^{'}$, sse esiste una strong bisimulation che li collega.

2.3 Formalizzazione del problema

Dato l'insieme di stati di un LTS, si dimostri che:

$$s \sim_{string} s^{'} \iff s \sim s^{'}$$

2.4 Dimostrazione

($\Longrightarrow$): L'ipotesi è $s \sim_{string} s^{'}$, la tesi da dimostrare è $s \sim s^{'}$.

Se $s \sim_{string} s^{'}$, allora esiste una relazione R, di string bisimulation, tale che $s \, R \, s^{'}$.

Dunque, dato che deve valere per qualsiasi sequenza di azioni possibile, consideriamo le sequenze di azioni di lunghezza pari a $|\sigma| = 1$, ovvero date da una singola azione $\alpha \in Act$.

In questo modo si ottiene che:

– se $s \xrightarrow{\alpha} s_1$, allora esiste una transizione $s^{'} \xrightarrow{\alpha} s_1^{'}$ tale che $s_1 R s_1^{'}$;

– se $s^{'} \xrightarrow{\alpha} s_1^{'}$, allora esiste una transizione $s \xrightarrow{\alpha} s_1$ tale che $s_1 R s_1^{'}$.

Ma questa è proprio la definizione di strong bisimulation.

Se esiste una strong bisimulation R che collega s e $s^{'}$ ($s \, R \, s^{'}$), allora s e $s^{'}$ sono strong bisimilar, ovvero $s \sim s^{'}$, che è proprio la tesi.

($\Longleftarrow$) L'ipotesi è $s \sim s^{'}$, la tesi da dimostrare è $s \sim_{string} s^{'}$.

Se $s \sim s^{'}$, allora esiste una relazione R, di strong bisimulation, tale che $s\,R\,s^{'}$, devo dimostrare che tale relazione è anche una relazione di string bisimulation. Dimostrazione per induzione sulla lunghezza della sequenza di azioni σ.

Caso base: (Sequenze di azioni σ di lunghezza $k = 1$)

Dato che, come visto nel punto precedente, per sequenze di azioni di lunghezza pari a $k = 1$ le definizioni di strong bisimulation e string bisimulation coincidono, se $s\,R\,s^{'}$ allora $s \sim_{string} s^{'}$, che è proprio la tesi.

Ipotesi Induttiva:

Se $s\,R\,s^{'}$, con R relazione di strong bisimulation, allora R è anche una relazione di string bisimulation per sequenze di azioni σ di lunghezza al più k.

Caso induttivo: (Sequenze di azioni σ di lunghezza $k+1$)

Dato che $s\,R\,s^{'}$, con R relazione di strong bisimulation sappiamo, per ipotesi induttiva, che R è anche una relazione di string bisimulation per sequenze di azioni $\sigma^{'}$ di lunghezza al più k :

- se $s \xrightarrow{\sigma^{'}} s_k$, allora esiste una transizione $s^{'} \xrightarrow{\sigma^{'}} s_k^{'}$ tale che $s_k\,R\,s_k^{'}$;
- se $s^{'} \xrightarrow{\sigma^{'}} s_k^{'}$, allora esiste una transizione $s \xrightarrow{\sigma^{'}} s_k$ tale che $s_k\,R\,s_k^{'}$.

Se $s_k\,R\,s_k^{'}$, ricordando che R è una relazione di strong bisimulation, abbiamo che:

- se $s_k \xrightarrow{\alpha} s_{k+1}$, allora esiste una transizione $s_k^{'} \xrightarrow{\alpha} s_{k+1}^{'}$ tale che $s_{k+1} R s_{k+1}^{'}$;
- se $s_k^{'} \xrightarrow{\alpha} s_{k+1}^{'}$, allora esiste una transizione $s_k \xrightarrow{\alpha} s_{k+1}$ tale che $s_{k+1}\,R\,s_{k+1}^{'}$.

Posto quanto sopra si può osservare che:

- se $s \xrightarrow{\sigma^{'}} s_k \xrightarrow{\alpha} s_{k+1}$, allora esiste una transizione $s^{'} \xrightarrow{\sigma^{'}} s_k^{'} \xrightarrow{\alpha} s_{k+1}^{'}$ tale che $s_{k+1}\,R\,s_{k+1}^{'}$;
- se $s^{'} \xrightarrow{\sigma^{'}} s_k^{'} \xrightarrow{\alpha} s_{k+1}^{'}$, allora esiste una transizione $s \xrightarrow{\sigma^{'}} s_k \xrightarrow{\alpha} s_{k+1}$ tale che $s_{k+1}\,R\,s_{k+1}^{'}$.

Riscrivendo tale definizione con l'impiego della sequenza di azioni $\sigma = \sigma^{'}.\alpha$, si ottiene:

- se $s \xrightarrow{\sigma} s_{k+1}$, allora esiste una transizione $s^{'} \xrightarrow{\sigma} s_{k+1}^{'}$ tale che $s_{k+1} R s_{k+1}^{'}$;

– se $s^{'} \xrightarrow{\sigma} s^{'}_{k+1}$, allora esiste una transizione $s \xrightarrow{\sigma} s_{k+1}$ tale che $s_{k+1} \, R \, s^{'}_{k+1}$.

Che è la definizione di string bisimulation per sequenze di azioni di lunghezza $|\sigma| = k + 1$, che è proprio ciò che cercavamo.

Dimostrato che le relazioni di strong bisimulation e string bisimulation coincidono risulta che: $s \sim s^{'} \implies \exists R$ strong bisimulation t.c. $sRs^{'} \implies s \sim_{string} s^{'}$, che è proprio la tesi.

□

3 Prisoner's game

Cinquanta prigionieri tenuti in celle separate hanno avuto la possibilità di essere rilasciati: Di volta in volta uno di loro verrà portato in una stanza speciale (in nessun ordine particolare, anche più volte consecutivamente, ma con una schedulazione fair per evitare attese infinite) e poi portato indietro nella cella.

La stanza è completamente vuota a eccezione di un interruttore che può accendere o spegnere la luce (la luce non è visibile dall'esterno e non può essere rotta). In qualsiasi momento, se qualcuno di loro dice che tutti i prigionieri sono già entrati nella stanza almeno una volta e questo è vero, allora tutti i prigionieri saranno rilasciati (ma se è falso, allora la possibilità termina e non saranno mai rilasciati). Prima che la sfida inizi, i prigionieri hanno la possibilità di discutere insieme alcuni "protocolli" da seguire.

Riesci a trovare una strategia vincente per i prigionieri?

Si noti che lo stato iniziale della luce nella stanza non è noto.

3.1 Strategia vincente

Una strategia vincente per i prigionieri è quella di designare uno di loro tale che possa solo spegnere la luce e che, quando la trova accesa, deve sempre spegnerla tenendo a mente il numero di volte che lo fa. Gli altri prigionieri, invece, possono solo accendere la luce e devono farlo le prime due volte, e soltanto la prime due, in cui entrando nella stanza la trovano spenta.

Il prigioniero addetto allo spegnimento della luce dovrà dichiarare che tutti i prigionieri sono già entrati nella stanza almeno una volta, quando avrà contato di aver spento la luce $2 * (n - 1)$ volte, dove n è il numero dei prigionieri.

In questo modo se lo stato iniziale della luce è spento, allora i prigionieri saranno entrati tutti almeno due volte, mentre se lo stato iniziale della luce è acceso un prigioniero sarà entrato almeno una volta e tutti gli altri almeno due volte. In qualsiasi caso sarà sicuro che almeno una volta sono entrati tutti nella stanza, così com'è richiesto per vincere la sfida.

3.2 Modellazione in CCS

Definiamo prima di tutto la lampadina con due espressioni CCS, LIGHTon per descrivere il comportamento della lampadina accesa e LIGHToff per descrivere il comportamento della lampadina spenta. Inseriamo anche due canali per gestire l'accesso alla stanza e due per decretare se il gioco va avanti o se vi è una dichiarazione che porrebbe fine della sfida (non consentendo successivi accessi); la stanza altro non è che una sezione critica nella quale solo un prigioniero alla volta può accedere:

$\text{LIGHTon} = \overline{\text{enter}}.\overline{\text{isON}}.(\overline{\text{exit}}.\text{LIGHTon} + \text{turnOFF}.\overline{\text{exit}}.(\text{continue}.\text{LIGHToff} + \text{finish}.\text{END}))$;

$\text{LIGHToff} = \overline{\text{enter}}.\overline{\text{isOFF}}.(\overline{\text{exit}}.\text{LIGHToff} + \text{turnON}.\overline{\text{exit}}.\text{LIGHTon})$;

$\text{END} = 0$.

Per comodità eliminiamo un canale unendo l'accesso alla stanza con l'informazione sullo stato della lampadina:

$\text{LIGHTon} = \overline{\text{enterON}}.(\overline{\text{exit}}.\text{LIGHTon} + \text{turnOFF}.\overline{\text{exit}}.(\text{continue}.\text{LIGHToff} + \text{finish}.\text{END}))$;

$\text{LIGHToff} = \overline{\text{enterOFF}}.(\overline{\text{exit}}.\text{LIGHToff} + \text{turnON}.\overline{\text{exit}}.\text{LIGHTon})$.

Ora definiamo il comportamento del prigioniere che ha il potere — e il dovere — di spegnere la lampadina tenendo il conto di quante volte lo fa. Dal momento che deve contare fino ad un certo numero, $2 * (n - 1)$, dove n è il numero dei prigionieri, è necessario definire $2 * (n - 1)$ espressioni CCS:

$\text{Pcounter}_i = \text{enterON}.\overline{\text{turnOFF}}.\text{exit}.\overline{\text{continue}}.\text{Pcounter}_{i+1} + \text{enterOFF}.\text{exit}.\text{Pcounter}_i \qquad \text{per } 0 \leq i \leq 2 * (n-1) - 2 \text{ , dove } i \in (N)$;

$\text{Pcounter}_{2*(n-1)-1} = \text{enterON}.\overline{\text{turnOFF}}.\text{exit}.\overline{\text{finish}}.0 + \text{enterOFF}.\text{exit}.\text{Pcounter}_{2*(n-1)-1}$.

Gli altri prigionieri, detti prigionieri semplici, sono definiti con tre espressioni: Ptic0 che ne descrive il comportamento prima che accendano la luce la prima volta, Ptic1 prima che l'accendano per la seconda volta e Ptic2 dopo che l'hanno già accesa tutte e due le volte.

$\text{Ptic0} = \text{enterON}.\text{exit}.\text{Ptic0} + \text{enterOFF}.\overline{\text{turnON}}.\text{exit}.\text{Ptic1}$.

$\text{Ptic1} = \text{enterON}.\text{exit}.\text{Ptic1} + \text{enterOFF}.\overline{\text{turnON}}.\text{exit}.\text{Ptic2}$.

$\text{Ptic2} = \text{enterON}.\text{exit}.\text{Ptic2} + \text{enterOFF}.\text{exit}.\text{Ptic2}$.

Definite le espressioni di base, incapsuliamole in espressioni composte che rispecchiano meglio le entità in gioco.

La lampadina a stato non noto:

$\text{LIGHT} = \text{LIGHTon} + \text{LIGHToff}$.

I prigionieri:

$\text{PRISONERS} = \text{Pcounter0} | \underbrace{\text{Ptic0}|\text{Ptic0}|\cdots|\text{Ptic0}}_{n-1}$.

Sia $L \in \mathcal{L}$ l'insieme di tutti i nomi dei canali:

$L = \{\text{enterON, enterOFF, exit, turnON, turnOFF, continue, finish}\}$,

allora la descrizione in CCS del gioco dei prigionieri con i prigionieri vincenti è:

$\text{PrisonerGameSuccess} = (\text{PRISONERS} \,|\, \text{LIGHT}) \setminus L$.

3.2.1 Verifiche di proprietà tramite CCS

Ci si aspetta che la sfida termini solo dopo che la luce è stata spenta $2 * (n - 1)$ volte. Per controllare tale specifica aggiungiamo un output verso l'esterno al processo del prigioniero addetto al conteggio, tale output verrà effettuato ogniqualvolta spegnerà la luce:

$\text{PcounterSpec}_i = \text{enterON}.\overline{\text{turnOFF}}.\overline{off}.\text{exit}.\overline{\text{continue}}.\text{PcounterSpec}_{i+1} +$
$\text{enterOFF}.\text{exit}.\text{PcounterSpec}_i \qquad \text{per } 0 \leq i \leq 2 * (n-1) - 2$,dove $i \in (N)$;

$\text{PcounterSpec}_{2*(n-1)-1} = \text{enterON}.\overline{\text{turnOFF}}.\overline{off}.\text{exit}.\overline{\text{finish}}.0 +$
$\text{enterOFF}.\text{exit}.\text{PcounterSpec}_{2*(n-1)-1}$.

E un output verso l'esterno che verrà effettuato quando la sfida sarà terminata:

$\text{ENDSpec} = \overline{end}.0$.

Vengono modificati di conseguenza i processi LIGHTon e LIGHToff, il processo LIGHT, che chiamerà le due nuove versioni di LIGHTon e LIGHToff, e il processo PRISONERS, che chiamerà PcounterSpec0 invece che Pcounter0:

$\text{LIGHTonSpec} = \overline{\text{enterON}}.(\overline{\text{exit}}.\text{LIGHTonSpec} +$
$\text{turnOFF}.\overline{\text{exit}}.(\text{continue}.\text{LIGHToffSpec} + \text{finish}.\text{ENDSpec}))$;

$\text{LIGHToffSpec} = \overline{\text{enterOFF}}.(\overline{\text{exit}}.\text{LIGHToffSpec} + \text{turnON}.\overline{\text{exit}}.\text{LIGHTonSpec})$;

$\text{LIGHTSpec} = \text{LIGHTonSpec} + \text{LIGHToffSpec}$;

$\text{PRISONERSSpec} = \text{PcounterSpec}_0 | \underbrace{\text{Ptic0}|\text{Ptic0}|\cdots|\text{Ptic0}}_{n-1}$.

Data la nuova descrizione CCS:

$$\text{PrisonerGameSuccessSpec} = (\text{PRISONERSSpec} \,|\, \text{LIGHTSpec}) \setminus L \ .$$

E la specifica:

$$\text{Spec} = \underbrace{\overline{\text{off}}.\overline{\text{off}}.\cdots.\overline{\text{off}}}_{2*(n-1)}.\overline{\text{end}}.0 \ .$$

Risulta:

$$\text{Spec} \approx \text{PrisonerGameSuccessSpec} \ .$$

In maniera analoga si possono controllare due diverse specifiche sulla base del numero di volte in cui viene accesa la luce. Infatti è semplice osservare come, utilizzando uno stesso processo per ciascun prigioniero, ognuno di essi possa accendere la luce al massimo due volte. Considerato ciò e considerato che solo Pcounter può porre fine alla prova, dimostrando che la sfida termina dopo 2*(n-1) volte che la luce è stata accesa (o dopo 2*(n-1)-1 volte nel caso in cui nello stato iniziale la luce sia accesa) si dimostra che la sfida termina soltanto dopo che tutti i prigionieri sono entrati almeno una volta nella stanza o, più precisamente, almeno due volte (tranne nel caso vi sia la luce accesa all'inizio, in quel caso per uno dei prgionieri è garantito solo un accesso).

Al fine di dimostrare ciò si utilizza il processo originale per il prigioniero addetto alla conta e si modificano invece le espressioni che descrivono il comportamento dei prigionieri semplici:

$$\text{PticSpec0} = \text{enterON.exit.PticSpec0} + \text{enterOFF}.\overline{\text{turnON}}.\overline{\text{on}}.\text{exit.PticSpec1} \ .$$

$$\text{PticSpec1} = \text{enterON.exit.PticSpec1} + \text{enterOFF}.\overline{\text{turnON}}.\overline{\text{on}}.\text{exit.Ptic2} \ .$$

Si modifica di conseguenza PRISONERS:

$$\text{PRISONERSSpec}\_ = \text{Pcounter0}|\underbrace{\text{PticSpec0}|\text{PticSpec0}|\cdots|\text{PticSpec0}}_{n-1} \ .$$

Dando due nuove descrizioni CCS, una per ognuno dei due stati iniziali possibili:

$$\text{PrisonerGameSuccessSpecON} = (\text{PRISONERSSpec}\_ \,|\, \text{LIGHTonSpec}) \setminus L \ ;$$

$$\text{PrisonerGameSuccessSpecOFF} = (\text{PRISONERSSpec}\_ \,|\, \text{LIGHToffSpec}) \setminus L \ .$$

E due specifiche, una per ciascun stato iniziale possibile:

$$\text{SpecON} = \underbrace{\overline{\text{on}}.\overline{\text{on}}.\cdots.\overline{\text{on}}}_{2*(n-1)-1}.\overline{\text{end}}.0 \ .$$

$$\text{SpecOFF} = \underbrace{\overline{\text{on}}.\overline{\text{on}}.\cdots.\overline{\text{on}}}_{2*(n-1)}.\overline{\text{end}}.0 \ .$$

Due specifiche diverse perchè, nel caso la lampadina fosse inizialmente accesa, allora uno dei prigionieri l'accenderebbe solo una volta prima che la sfida finisca invece che due come tutti gli altri.

Risulta che:

$\text{SpecON} \approx \text{PrisonerGameSuccessSpecON}$;

$\text{SpecOFF} \approx \text{PrisonerGameSuccessSpecOFF}$.

3.2.2 Verifiche di proprietà tramite HML

Vi sono diverse proprietà che è importante verificare con la logica di Hennessy-Milner:

- che sia garantita la mutua esclusione per l'accesso alla stanza;
- la presenza — non desiderata — di livelock;
- che ogni sequenza di transizioni completa, derivante da una schedulazione fair, porti prima o poi alla fine della sfida;
- che una volta dichiarata la fine della sfida alcun prigioniero sia all'interno della stanza e che nessuno di essi possa più accedervi.

Per la verifica della prima proprietà, ovvero che solo un prigioniero alla volta può trovarsi all'interno della stanza, basti pensare che l'accesso alla stanza è modellato dal processo LIGHT, dunque essendo un unico processo che include una choice e non una parallelizzazione potrà sincronizzarsi con un solo processo prigioniero alla volta. Tuttavia si procede a verificare formalmente tale proprietà utilizzando la logica di Hennessy-Milner, a tale scopo si modificano i processi dei prigionieri in modo tale che effettuino un output all'esterno ogniqualvolta accedono alla stanza e un output all'esterno ogniqualvolta ne escono (subito prima di aver effettivamente liberato la stanza, altrimenti potrebbe tornare occupata prima che venga effettuato tale output):

$$\text{PcounterHolder}_i = \text{enterON}.\overline{\text{busy}}.\overline{\text{turnOFF}}.\overline{\text{free}}.\text{exit}.\overline{\text{continue}}.\text{PcounterHolder}_{i+1} + \text{enterOFF}.\overline{\text{busy}}.\overline{\text{free}}.\text{exit}.\text{PcounterHolder}_i \qquad \text{per } 0 \leq i \leq 2*(n-1)-2 \text{ ,dove } i \in (N) \text{ .}$$

$$\text{PcounterHolder}_{2*(n-1)-1} = \text{enterON}.\overline{\text{busy}}.\overline{\text{turnOFF}}.\overline{\text{free}}.\text{exit}.\overline{\text{finish}}.0 + \text{enterOFF}.\overline{\text{busy}}.\overline{\text{free}}.\text{exit}.\text{PcounterHolder}_{2*(n-1)-1}$$

$$\text{PticHolder0} = \text{enterON}.\overline{\text{busy}}.\overline{\text{free}}.\text{exit}.\text{PticHolder0} + \text{enterOFF}.\overline{\text{busy}}.\overline{\text{turnON}}.\overline{\text{free}}.\text{exit}.\text{PticHolder1} \text{ .}$$

$$\text{PticHolder1} = \text{enterON}.\overline{\text{busy}}.\overline{\text{free}}.\text{exit}.\text{PticHolder1} + \text{enterOFF}.\overline{\text{busy}}.\overline{\text{turnON}}.\overline{\text{free}}.\text{exit}.\text{PticHolder2} \text{ .}$$

$$\text{PticHolder2} = \text{enterON}.\overline{\text{busy}}.\overline{\text{free}}.\text{exit}.\text{PticHolder2} + \text{enterOFF}.\overline{\text{busy}}.\overline{\text{free}}.\text{exit}.\text{PticHolder2} \text{ .}$$

$$\text{PRISONERSHolder} = \text{PcounterHolder0} | \underbrace{\text{PticHolder0}|\text{PticHolder0}|\cdots|\text{PticHolder0}}_{n-1} .$$

La descrizione CCS diventa:

$$\text{PrisonerGameSuccessME} = (\text{PRISONERSHolder} \,|\, \text{LIGHT}) \setminus L .$$

La formula HML:

ME max= ((< $\overline{\text{busy}}$ > T and [tau]F and [$\overline{\text{free}}$]F and [$\overline{\text{busy}}$]NoBusyUFree) or [$\overline{\text{busy}}$]F) and [−]ME ;

NoBusyUFree min= (< $\overline{\text{free}}$ > T and [[$\overline{\text{busy}}$]]F and [$\overline{\text{free}}$][tau][$\overline{\text{busy}}$]F) or ([$\overline{\text{busy}}$]F and < − > T and [−]NoBusyUFree) .

Le uniche tre azioni possibili sono: τ, $\overline{\text{busy}}$ e $\overline{\text{free}}$. La formula verifica innanzitutto che se è possibile eseguire un $\overline{\text{busy}}$, allora è l'unica transizione possibile. Non sono infatti presenti ne τ ne $\overline{\text{free}}$, dato che $\overline{\text{busy}}$ è l'azione subito successiva all'accesso della sezione critica ciò significa che tra l'entrata di un prigioniero nella stanza e l'output esterno che manda non è possibile nessun altra azione, tantomeno l'entrata di un ulteriore prigioniero. Successivamente si verifica, con l'utilizzo dell'espressione NoBusyUFree, che fino a che non sia possibile una $\overline{\text{free}}$ non è possibile alcuna $\overline{\text{busy}}$ e che, anche talora fosse possibile una $\overline{\text{free}}$, non sono comunque possibili percorsi alternativi, dunque composti da zero o più τ, che portino ad una $\overline{\text{busy}}$. Viene verificato anche che qualora venga eseguita una $\overline{\text{free}}$, non è possibile eseguire una $\overline{\text{busy}}$ dopo una sola azione successiva, questo perchè il rilascio della sezione critica avviene subito dopo la $\overline{\text{free}}$ quindi, prima che sia possibile una $\overline{\text{busy}}$, devono passare almeno altre due azioni successive alla $\overline{\text{free}}$, una che libera la stanza e una che la occupa; in questo modo si è anche verificato che tra una $\overline{\text{free}}$ e l'effettivo rilascio della stanza non vi siano accessi.

Risulta:

$$\text{PrisonerGameSuccessME} \models \text{ME} .$$

In conclusione si è dimostrato che non è possibile che due prigionieri si trovino contemporaneamente all'interno della della stanza.

Si procede ora a verificare la presenza di cicli di livelock. Considerando che i prigionieri sono chiamati senza seguire alcun preciso ordine, dunque con la possibilità di chiamare anche più volte consecutivamente lo stesso prigioniero, e considerate le regole SOS per processi paralleli, non è possibile garantire una schedulazione fair e, di conseguenza, l'assenza di cicli di livelock.

$$\text{PrisonerGameSuccess} \models \text{Livelock} ;$$

Livelock min= LOOP or < − > Livelock ;

LOOP max=< tau > LOOP .

La proprietà è soddisfatta: ciò significa che, come ci si aspettava, sono presenti cicli di livelock.

Si continua verificando che ogni sequenza di transizioni completa, derivante da una schedulazione fair, porti prima o poi alla fine della sfida. Si utilizza la seguente descrizione CCS:

PrisonerGameSuccessENDS = (PRISONERS | LIGHTSpec) \ L .

In tal modo si considera la sfida terminata quando può fare un $\overline{\text{end}}$ e poi più nessuna azione. Utilizzando la descrizione iniziale —PrisonerGameSuccess— si sarebbe dovuta considerare terminata la sfida quando non sarebbe stata possibile più nessuna azione, ma in questo caso si sarebbe considerata terminata la sfida anche in una situazione di deadlock, cosa differente e non gradita.

Verifichiamo prima di tutto che dopo $\overline{\text{end}}$ non sia possibile nessun'altra azione, in questo modo si verifica che il raggiungimento di $\overline{\text{end}}$ decreta effettivamente la fine della sfida. Nella definizione di tale formula è importante tenere conto del fatto che le uniche azioni possibili in PrisonerGameSuccesEnds sono τ e $\overline{\text{end}}$:

PrisonerGameSuccessENDS $\models$ ENDandSTOP ;

ENDandSTOP max= ((< $\overline{\text{end}}$ > T and [τ]F and < $\overline{\text{end}}$ > [$-$]F) or
(< τ > T and [$\overline{\text{end}}$]F)) and [τ]ENDandSTOP .

Con questa formula si sono verificate anche altre due proprietà: che sono sempre possibili transizioni finché non avviene un $\overline{\text{end}}$, ovvero che non vi sono situazioni di deadlock, e che se si può fare un $\overline{\text{end}}$ allora è l'unica azione possibile, ovvero che una volta che il prigioniero addetto alla conta dichiara che sono entrati tutti nella stanza, l'unica azione possibile è $\overline{\text{end}}$ e poi più nessuna azione, dunque da quando avviene tale dichiarazione nessun prigioniero può più entrare o uscire dalla stanza.

Dato che le uniche azioni possibili sono τ ed $\overline{\text{end}}$ e considerato che dopo $\overline{\text{end}}$ non è possibile compiere nessun'altra azione, basta verificare che in tutti gli stati raggiungibili con zero o più τ sia possibile raggiungere un $\overline{\text{end}}$ per affermare che qualsiasi sequenza di transizione completa, risultante da una schedulazione fair — ovvero che non resti per sempre all'interno di cicli di livelock — porta prima o poi alla fine della sfida:

PrisonerGameSuccessENDS $\models$ ENDreachability ;

ENDreachability max=<< $\overline{\text{end}}$ >> T and [τ]ENDreachability .

Non resta ora che verificare che una volta dichiarata la fine della sfida nessun prigioniero sia all'interno della stanza e che nessuno possa più accedervi (quest'ultima già verificata). Per controllare tale proprietà si utilizza la seguente descrizione CCS:

PrisonerGameSuccessFREE = (PRISONERSHolder | LIGHTSpec) \ L .

In questo modo è possibile controllare che nessun prigioniero sia all'interno della stanza verificando che tutti gli $\overline{\text{end}}$ siano preceduti esattamente da una $\overline{\text{free}}$ e due τ: una che corrisponde all'effettiva uscita del prigioniero dalla stanza e l'altra alla dichiarazione che porta al termine della sfida. Abbiamo già osservato che dopo $\overline{\text{end}}$ non sono più possibili altre azioni, ciò significa che nessun prigioniero può più entrare (o uscire) dalla stanza. Considerando che è garantita la mutua esclusione, già verificata precedentemente, basta dimostrare che una $\overline{\text{free}}$ è l'ultima e unica possibile transizione prima delle due τ che portano a $\overline{\text{end}}$ per avere la garanzia che la sfida termini senza alcun prigioniero all'interno della stanza. Per aver la certezza che nessuna delle due τ successive alla $\overline{\text{free}}$ e precedenti a $\overline{\text{end}}$ siano un accesso alla stanza, si controlla anche che non sia possibile alcuna transizione $\overline{\text{busy}}$:

PrisonerGameSuccessFREE $\models$ FreeRoom ;

FreeRoom max= ((< $\overline{\text{free}}$ >< τ >< τ >< $\overline{\text{end}}$ > T and [$\overline{\text{free}}$][τ][τ] < $\overline{\text{end}}$ > T and [$\overline{\text{free}}$][[$\overline{\text{busy}}$]]F) or [−][−][−][$\overline{\text{end}}$]F) and [−]FreeRoom .

3.3 Implementazione in Go

L'implementazione in Go è stata realizzata seguendo le espressioni definite precedentemente in CCS. Sono stati ricalcati gli stessi algoritmi con gli stessi canali e con gli stessi segnali di input e di output.

Dato che lo stato iniziale della luce è non noto, all'avvio del programma viene inizializzato in maniera pseudo-casuale.

Di seguito viene mostrato il codice che descrive il comportamento della lampadina e che modella l'accesso alla stanza, conforme alle espressioni CCS LIGHTon e LIGHToff:

```
func (self *Bulb) Run (enterON, enterOFF, exit chan bool) {
        // Inizializzazione casuale dello stato della luce
        rand.Seed(time.Now().UTC().UnixNano())
        rdn := rand.Intn(2)
        state := self.states[rdn]

        fmt.Printf("The initial state of the light is: %s\n", state)
        for {
                if state == "LIGHTon" {
                        // LIGHTon
                        enterON <- true
                        select {
                        case exit <- true:
                                state = self.states[0]
                        case  <- self.turnOFF:
                                exit <- true
                                select {
                                        case <- self.continue_:
                                                state = self.states[1]
                                        case <- self.finish:
                                                fmt.Printf("Game ended!\n")
                                                return // END
                                }
                        }
                } else {
                        // LIGHToff
```

```
                        enterOFF <- true
                        select {
                        case exit <- true:
                                state = self.states[1]
                        case  <- self.turnON:
                                exit <- true
                                state = self.states[0]
                        }
                }
        }
}
```

, il comportamento del prigioniero addetto a spegnere la luce, a contare e a determinare la fine della sfida, descritto in CCS da Pcounter_i :

```
func (self *Pcounter) Run (turnOFF, turnON, continue_, finish chan bool, n
    int) {
  counter := 0
  for {
    select {
      case <- self.enterON:
        fmt.Printf( "Prisoner %s enters the room with the light on.\n", self.
            name)
        fmt.Printf( "Prisoner %s turns off the light for the %d time.\n",
            self.name, counter+1)
        turnOFF <- true
        fmt.Printf( "Prisoner %s exits the room.\n", self.name)
        <- self.exit
        counter = counter + 1
        if counter == 2*(n-1) {
          fmt.Printf( "Prisoner %s says that all the prisoners have already
              entered the room at least once.\n", self.name)
          finish <- true
          return
        } else {
          continue_ <- true
        }
      case  <- self.enterOFF:
        fmt.Printf( "Prisoner %s enters the room with the light off.\n", self
            .name)
        fmt.Printf( "Prisoner %s does nothing and exits the room.\n", self.
            name)
        <- self.exit
    }
  }
}
```

e il comportamento degli altri prigionieri, che devono accendere la luce le prime due volte che entrando nella stanza la trovano spenta, descritto in CCS da Ptic_i :

```
func (self *Ptic) Run (turnOFF, turnON, continue_, finish chan bool) {
  counter := 0
  for {
      select {
        case  <- self.enterON:
          fmt.Printf( "Prisoner %d enters the room with the light on.\n",
              self.name)
          fmt.Printf( "Prisoner %d does nothing and exits the room.\n", self.
              name)
          <- self.exit
        case <- self.enterOFF:
          fmt.Printf( "Prisoner %d enters the room with the light off.\n",
              self.name)
          if counter == 2 { // Ptic2
            fmt.Printf( "Prisoner %d does nothing and exits the room.\n",
                self.name)
            <- self.exit
```

```
            } else{
              fmt.Printf( "Prisoner %d turns on the light for the %d time.\n",
                  self.name, counter+1)
              turnON <- true
              fmt.Printf( "Prisoner %d exits the room.\n", self.name)
              <- self.exit
              counter = counter + 1
            }
        }
    }
}
```

I vari processi, protagonisti del problema, sono eseguiti in maniera concorrente sfruttando le potenzialità delle goroutine.

Tale implementazione è stata realizzata permettendo un numero qualsiasi di prigionieri, che si può passare in input da linea di comando oppure, una volta avviato il programma, inserendolo da console.