Università degli Studi di Udine

Corso di Laurea Magistrale in Informatica

Sistemi di Transizioni Equi

Angelo Montanari e Donatella Gubiani

Nozioni di base - 1

Come rappresentare i sistemi di transizioni equi?

- Lo *stato* di un sistema di transizioni equo (STE) viene descritto attraverso (i valori di) un dato insieme di variabili che appartengono ad un universo di variabili $\mathcal{V}$ (*vocabolario*).
- Le variabili in $\mathcal{V}$ *sono tipate*.
- Per ogni variabile $x \in \mathcal{V}$ assumiamo che esista una "variabile" $x' \in \mathcal{V}$ (se la variabile x è associata ad uno stato, la variabile x' è associata allo stato successivo).

Nozioni di base - 2

- Un STE verrà descritto utilizzando la *logica del prim'ordine*:
  - termini/espressioni: costruiti a partire dalle variabili in $\mathcal{V}$ e dalle costanti usando le usuali funzioni
  - formule atomiche: proposizioni (variabili booleane) e predicati applicati ai termini
  - formule di stato/asserzioni: si ottengono dalle formule atomiche applicando connettivi logici e quantificatori
- Uno **stato** è un'interpretazione di $\mathcal{V}$ (che rispetta il tipo delle variabili) che assegna ad ogni variabile $u \in \mathcal{V}$ un valore $s[u]$ appartenente al dominio di u. Denotiamo con Σ l'insieme di tutti gli stati (se l'insieme degli stati è infinito, parliamo di sistema a stati infiniti).

Sistemi di Transizioni Equi

Un *Sistema di Transizioni Equo* (STE) è una quintupla

$$\langle V, \theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$$

dove

- $V = \{u_1, ..., u_n\}$ è un insieme di **variabili di sistema**; V può essere partizionato in:
  - *variabili di controllo* (control variable)
  - *variabili relative ai dati* (data variable)
- θ è una formula di stato soddisfacibile che caratterizza i possibili stati iniziali del sistema (**condizione iniziale**). Uno stato che soddisfa θ verrà detto stato iniziale.

L'Insieme Finito delle Transizioni

- $\mathcal{T}$ è un **insieme finito** di **transizioni**
  - ogni transizione $\tau \in \mathcal{T}$ è una funzione
    $$\tau : \Sigma \to 2^{\Sigma}$$
    che mappa ogni stato $s \in \Sigma$ in un insieme di stati $\tau(s) \subseteq \Sigma$
  - ogni stato in $\tau(s)$ è detto τ-*successore* di s
  - τ è *abilitata* in s se $\tau(s) \neq \emptyset$, altrimenti τ è *disabilitata*

Osservazione. La tripla $\langle V, \theta, \mathcal{T} \rangle$ definisce un sistema di transizioni classico.

Condizioni di Equità Debole e Forte

- $\mathcal{J} \subseteq \mathcal{T}$ è l'insieme delle transizioni per le quali chiediamo **giustizia** (equità debole), ossia per le quali escludiamo che possano esistere delle computazioni in cui esse siano costantemente abilitate da un certo punto (stato) in poi e mai eseguite.

- $\mathcal{C} \subseteq \mathcal{T}$ è l'insieme delle transizioni per le quali chiediamo **compassione** (equità forte), ossia per le quali escludiamo che possano esistere delle computazioni in cui esse siano abilitate un numero infinito di volte ed eseguite solo un numero finito di volte.

Un Paio di Osservazioni

Osservazione 1. La condizione di equità forte implica quella di equità debole, ma non viceversa.

Osservazione 2. Non è possibile verificare "al finito" (ossia analizzandone un prefisso finito) se una computazione rispetta o meno le condizioni di equità.

Rappresentazione delle Transizioni

Ogni transizione $\tau \in \mathcal{T}$ può essere rappresentata attraverso una formula della logica del prim'ordine $\rho_\tau(V, V')$, detta **relazione di transizione**.

La relazione $\rho_\tau(V, V')$ vuole esprimere il legame tra uno stato s e ciascuno dei suoi τ-successori $s' \in \tau(s)$ (la versione non prima delle variabili si riferisce ai loro valori in s, la versione prima ai loro valori in s').

Esempio: la formula $x' = x + 1$ stabilisce che il valore di x in s' ($s'[x]$) è uguale al valore di x in s più 1 ($s[x] + 1$).

Lo stato s' è un successore dello stato s attraverso la transizione τ (s' è un τ-successore di s) se $\rho_\tau(V, V')$ valuta a vero qualora si interpreti ogni $x \in V$ come $s[x]$ e ogni $x' \in V'$ come $s'[x]$.

Condizione di Abilitazione

Possiamo esprimere la **condizione di abilitazione** di una transizione τ (denotata $En(\tau)$) in uno stato s attraverso la formula $\exists V' \rho_\tau(V, V')$.

Nell'insieme di tutte le transizioni $\mathcal{T}$ è sempre presente la **idling transition**, denotata con τ_I e definita come $\rho_I : V' = V$ (s' è un τ_I-successore di s se e solo se s e s' concordano sul valore di tutte le variabili del sistema, inclusa la variabile di controllo).

Per τ_I non vengono richieste condizioni di equità: è sempre abilitata, ma potrebbe non essere mai eseguita.

Può essere utilizzata anche per "completare" computazioni finite.

Tutte le transizioni $\tau \in \mathcal{T}$ diverse da τ_I sono dette **transizioni diligenti**.

Computazioni di un STE - 1

Consideriamo una sequenza infinita di stati $\sigma = s_0, s_1, s_2, ...$, dove $s_i \in \Sigma$ per ogni i.

Diremo che $\tau \in \mathcal{T}$ è **abilitata** nella posizione k della sequenza se τ è abilitata nello stato s_k.

Diremo che $\tau \in \mathcal{T}$ è **eseguita** (presa) in posizione k se $s_{k+1} \in \tau(s_k)$.

Si noti che diverse transizioni possono essere (considerate come) eseguite nella stessa posizione.

Computazioni di un STE - 2

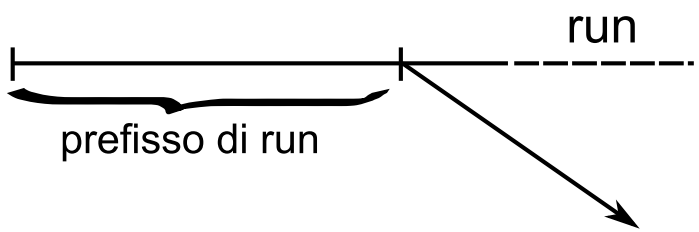
Una sequenza σ è una **computazione di un STE** P (**P-computazione**) se soddisfa le seguenti condizioni:

- (**inizialità**) s_0 soddisfa θ (s_0 è uno stato iniziale)
- (**consequenzialità**) per ogni $j = 0, 1, 2, ...$, s_{j+1} è un τ-successore di s_j per qualche τ
- (**giustizia**) per ogni $\tau \in \mathcal{J}$ non può mai accadere che τ sia costantemente abilitata da una data posizione k in poi e mai eseguita
- (**compassione**) per ogni $\tau \in \mathcal{C}$ non può mai accadere che τ sia abilitata un numero infinito di volte a partire da una data posizione k ed eseguita solo un numero finito di volte

Le condizioni di giustizia e compassione sono dette **requisiti di equità**.

Proprietà delle Computazioni

- Uno stato s è P-**accessibile** se compare in qualche P-computazione.
- Ogni stato τ-successore di uno stato P-accessibile, per una qualche $\tau \in \mathcal{T}$, è P-accessibile.
- Ogni sequenza σ che soddisfa i requisiti di inizialità e consequenzialità, ma non necessariamente quelli di equità, è detta **run** di P. Ogni prefisso finito di un run è anche prefisso di una computazione (ne segue che ogni stato s che compare in un run è P-accessibile).



Un Semplice Linguaggio di Programmazione

Introduciamo ora un semplice **linguaggio di programmazione**, denominato **SPL**, che consente di sintetizzare programmi reattivi. Esso presenta un buon compromesso tra semplicità ed espressività.

- Un programma SPL è una sequenza finita di istruzioni, eventualmente etichettate.
- Le istruzioni SPL possono essere suddivise in un certo insieme di classi:
  - istruzioni di base
  - istruzioni schematiche
  - istruzioni composte
  - istruzioni composite raggruppate

Istruzioni di Base

L'insieme delle **istruzioni di base** comprende i seguenti elementi:

- skip
- assegnamento
- await
- send/receive (comunicazione sincrona e asincrona)
- request/release (primitive per la gestione dei semafori)

Skip

L'istruzione **skip**:

- Sintassi: `skip`
- Semantica: non ha alcun effetto (se non quello di far passare il controllo da prima a dopo l'istruzione..).

Assegnamento

L'istruzione di **assegnamento** (multiplo):

- Sintassi: $(u_1, \ldots, u_k)$:= $(e_1, \ldots, e_k)$
  - $u_1, \ldots, u_k$ è una sequenza di variabili
  - $e_1, \ldots, e_k$ è una sequenza di espressioni/termini
- Semantica: effettua simultaneamente le assegnazioni $u_1 := e_1, \ldots, u_k := e_k$
- Casi particolari:
  - vettori/sequenze (notazione abbreviata): $\overline{u}$:= $\overline{e}$
  - singoletti: u := e

Await

L'istruzione **await**:

- Sintassi: `await` c
  - c è un'espressione booleana (detta *guardia*)
- Semantica: attende che la guardia diventi vera; quando tale condizione si realizza, trasferisce il controllo da prima a dopo l'istruzione.
- Non modifica alcuna variabile relativa ai dati.
- Un'applicazione particolare:
  - `await false` $\equiv$ `halt`
    (blocca la computazione)

Istruzioni per lo Scambio Messaggi: Send

L'istruzione **send**:

- Sintassi: $\alpha \Leftarrow e$
  - α è un canale (un canale è una variabile di sistema di tipo particolare)
  - e è un'espressione di tipo compatibile con α
- Semantica: e viene valutata e il suo valore viene assegnato al canale α.

Istruzioni per lo Scambio Messaggi: Receive

L'istruzione **receive**:

- Sintassi: $\alpha \Rightarrow u$
  - α è un canale
  - u è una variabile di tipo compatibile con α
- Semantica: il valore α viene assegnato alla variabile u.
- Se il canale non contiene alcun valore, l'esecuzione dell'istruzione viene sospesa/ritardata (come vedremo, ciò può accadere solo con i canali asincroni..).

Istruzioni per la Gestione dei Semafori: Request/Release

Le istruzioni **request** e **release**:

- Sintassi: `request` r
- Semantica: l'istruzione viene eseguita solo se $r > 0$ (altrimenti la sua esecuzione viene sospesa/ritardata) e decrementa il valore di r di 1.

- Sintassi: `release` r
- Semantica: l'istruzione incrementa il valore di r di 1.

Istruzioni Schematiche

Le **istruzioni schematiche** rappresentano sequenze di istruzioni di cui non interessano i dettagli, ma solo le condizioni di terminazione.

Sono di particolare utilità nella modellazione di alcuni problemi classici, quali:

- mutua esclusione: noncritical/critical
- paradigma produttore/consumatore: produce/consume

La Mutua Esclusione: Noncritical/Critical

L'istruzione **noncritical**:

- Sintassi: `noncritical`
- Semantica: condensa in un'unica istruzione un'attività del programma non critica dal punto di vista della gestione della mutua esclusione (utilizzo di risorse private). **Non** si richiede che **termini** (è equivalente a τ_I).

L'istruzione **critical**:

- Sintassi: `critical`
- Semantica: condensa in un'unica istruzione un'attività del programma critica dal punto di vista della gestione della mutua esclusione (utilizzo di risorse condivise), che richiede un coordinamento fra i processi. Si richiede che **termini** (è pertanto necessario imporre condizioni di equità).

Il Paradigma Produttore/Consumatore: Produce/Consume

L'istruzione **produce**:

- Sintassi: `produce` x, dove x è una variabile
- Semantica: assegna a x un valore diverso da 0 (è l'unica operazione schematica che modifica il valore di variabili relative ai dati). Si richiede che **termini**.

L'istruzione **consume**:

- Sintassi: `consume` y, dove y è una variabile
- Semantica: modella l'attività di consumo nello schema produttore/consumatore; si assume che non modifichi il valore di y. Si richiede che **termini**.

Istruzioni Composte

L'insieme delle **istruzioni composte** comprende le seguenti operazioni:

- istruzione condizionale
- istruzione di concatenazione
- istruzione di selezione
- istruzione while
- istruzione di cooperazione
- istruzione blocco

Condizionale

L'istruzione **condizionale**:

- Sintassi: `if` c `then` S_1 `else` S_2
  - c è una condizione booleana
  - S_1 ed S_2 sono istruzioni
- Semantica: viene valutata c; se c è vera si procede con l'esecuzione di S_1, altrimenti si procede con l'esecuzione di S_2.
- A differenza dell'istruzione `await`, è sempre abilitata.
- Caso particolare:
  - `if` c `then` S_1 (se c è falsa, l'esecuzione termina in un passo)

Concatenazione

L'istruzione di **concatenazione**:

- Sintassi: S_1 ; S_2; . . . ; S_k
  - ogni S_i, con $i \in \{1, \ldots, k\}$, è un'istruzione
- Semantica: il primo passo dell'esecuzione dell'istruzione coincide col primo passo dell'esecuzione dell'istruzione S_1.
- Caso particolare:
  - `when` c `do` $S \equiv$ `await` c; S (comando con guardia)

Selezione

L'istruzione di **selezione**:

- Sintassi: S_1 `or` S_2 `or` ... `or` S_k
  - S_i con $i \in \{1, \ldots, k\}$ sono istruzioni
- Semantica: seleziona il sottoinsieme delle istruzioni abilitate e sceglie in modo non deterministico una di tali istruzioni quale istruzione da eseguire.
- Se nessuna istruzione è abilitata, l'esecuzione dell'istruzione di selezione viene sospesa/ritardata.
- Tale istruzione è spesso usata in combinazione con l'istruzione `when`:
  `if` $c_1 \rightarrow s_1 \ \square \ \ldots \ \square \ c_k \rightarrow s_k$ `fi` $\equiv$
  `[when` c_1 `do` s_1`]` `or` ... `or` `[when` c_k `do` s_k`]`

While

L'istruzione **while**:

- Sintassi: `while` c `do` S
  - c è un'espressione booleana
  - S è un'istruzione
- Semantica: si valuta c; se c è vera, si procede con l'esecuzione di S, altrimenti si esce dal ciclo (l'esecuzione dell'istruzione termina).
- L'esecuzione di tale istruzione non viene mai sospesa.
- Casi particolari:
  - `loop forever` $S \equiv$ `while` $TRUE$ `do` S
  - `for` $i := 1$ `to` n `do` $S \equiv$
    $i := 1$; `while` $i \leq n$ `do` $[S;\ i := i + 1]$

Cooperazione - 1

L'istruzione di **cooperazione**:

- Ha l'obiettivo di modellare l'esecuzione parallela di un certo insieme di istruzioni.
- Sintassi: $l:\quad [l_1 : S_1; \widehat{l_1} :] \;\|\; \ldots \;\|\; [l_k : S_k; \widehat{l_k} :]\; ;\; \widehat{l} :$
  - l, $\widehat{l}$, l_i e $\widehat{l_i}$, con $i \in \{1, \ldots, k\}$, sono **etichette**
  - S_i, con $i \in \{1, \ldots, k\}$, sono istruzioni

Cooperazione - 2

- Semantica:
  - il primo passo dell'esecuzione dell'istruzione di cooperazione è detto **entry step** (passo di entrata) e trasferisce il controllo da l a $l_1, \ldots, l_k$ (fa partire in parallelo l'esecuzione delle istruzioni $S_1, \ldots, S_k$).
  - l'ultimo passo dell'istruzione di cooperazione è detto **exit step** (passo di uscita) e trasferisce il controllo da $\widehat{l_1}, \ldots, \widehat{l_k}$ a $\widehat{l}$ (per terminare l'esecuzione dell'istruzione di cooperazione deve essere terminata l'esecuzione di tutte le istruzioni componenti).

Blocco

L'istruzione **blocco**:

- Sintassi: `[ local declaration;` S `]`
  - `local declaration` è una lista di istruzioni di dichiarazione di variabili del tipo:
    `local` $var_1, \ldots, var_n$`:` $type$ `where` φ
    la formula φ (opzionale) è della forma $y_1 = e_1, \ldots, y_m = e_m$ dove $y_1, \ldots, y_m$ sono variabili e $e_1, \ldots, e_m$ sono espressioni (che rispettano i tipi delle variabili) il cui valore dipende da variabili dichiarate esternamente al blocco (e da costanti)
  - S è un'istruzione, detta *corpo* del blocco
- Le istruzioni di dichiarazione vengono utilizzate per inizializzare il valore delle variabili locali.

Istruzioni Raggruppate - 1

- Vengono utilizzate per rendere atomiche (analogia con la nozione di transazione nelle basi di dati) alcune istruzioni composte, non tutte.
- La definizione delle istruzioni raggruppate si basa sulla nozione di istruzione composita, definita nel seguente modo:
  - `skip`, assegnazione, `await`, `send` e `receive` sono istruzioni composite di base
  - se S, $S_1, \ldots, S_k$ sono istruzioni composite, tali sono anche le seguenti istruzioni:
    - `when` c `do` S
    - `if` c `then` S_1 `else` S_2
    - S_1 `or ... or` S_k
    - $S_1; \ldots; S_k$

Istruzioni Raggruppate - 2

- Se S è un'istruzione composita, $\langle S \rangle$ è un'**istruzione raggruppata** a condizione che:
  - un'istruzione raggruppata che contiene un'istruzione di comunicazione (`send`/`receive`) sincrona sia della forma $\langle S_1;\ C;\ S_2 \rangle$, dove C è un'istruzione di comunicazione sincrona e S_1 ed S_2 non devono contenere istruzioni di comunicazione (S_1 ed S_2 possono mancare), ossia può comparire un'unica operazione `send` o `receive` sincrona
  - in un'istruzione raggruppata non possono comparire due o più istruzioni di comunicazione (`send`/`receive`) asincrona relative allo stesso canale

Esempio. Siano α e β due canali asincroni:

- $\langle \ldots \alpha \Leftarrow 1 \ldots \alpha \Leftarrow 3 \ldots \rangle$ NO
- $\langle \ldots \alpha \Leftarrow 1 \ldots \beta \Leftarrow 3 \ldots \rangle$ SI

I Programmi SPL

La struttura dei **programmi SPL**:

- Sintassi:
  P :: [declaration; [P_1::[$l_1 : S_1; \widehat{l_1}$:] || ... || P_k::[$l_k : S_k; \widehat{l_k}$:]]]
- I nomi del programma (P) e dei processi (P_i) sono opzionali.
- Sintassi simile alle istruzioni di cooperazione, ma non vi sono *entry step* e *exit step*.

Dichiarazione

- La **dichiarazione** (`declaration`) consiste in una lista di istruzioni del seguente tipo:

  `mode` $var_1, \ldots, var_n$: *type* `where` φ
  dove `mode` specifica il modo/tipo della dichiarazione e può assumere i valori:
  - `in` (specifica le variabili di input)
  - `local` (specifica le variabili locali)
  - `out` (specifica le variabili di output)

  La distinzione tra `local` e `out` è in realtà una distinzione di comodo, che ha il solo scopo di aumentare la leggibilità dei programmi.

Programma e Variabili

- Il programma non può modificare il valore delle variabili di tipo `in` (di sola lettura); può, invece, modificare il valore delle variabili `local` e `out`
- La formula φ (opzionale) impone dei vincoli sui valori iniziali delle variabili
  - nel caso `local`/`out` può assumere unicamente la forma $y = e$ (inizializzazione) e nell'espressione possono comparire unicamente variabili di tipo `in`
  - nel caso `in` non ci sono restrizioni sulla forma di φ, che può essere utilizzata per specificare l'insieme dei valori ammissibili per le variabili di input
- La congiunzione delle condizioni è definita come la **pre-condizione sui dati**.

Dichiarazioni dei Canali

Un ruolo particolare è svolto dalle dichiarazioni dei **canali**:

- canali **sincroni**:
  `mode` $\alpha_1, \ldots, \alpha_n$: `channel of` *type*
- canali **asincroni**:
  `mode` $\alpha_1, \ldots, \alpha_n$: `channel [1..] of` *type* `where` φ
  dove `[1..]` indica la presenza di un buffer di capacità illimitata e φ specifica le condizioni sul contenuto iniziale del buffer (se φ è assente, si assume che il buffer sia inizialmente vuoto)

Un Esempio di Programma SPL

calcolo del massimo comun divisore con l'algoritmo di Euclide

in a, b: integer where a>0, b>0
local y1, y2: integer where y1=a, y2=b
out g: integer

$$
\left[\begin{array}{l}
\ell_0: \left[\begin{array}{l}
\ell_1: \text{while } y1 <> y2 \text{ do} \\
\qquad \ell_2: \left[\begin{array}{l}
\ell_3: [\ \ell_9: \text{await } y1 > y2;\ \ell_4: y1 := y1 - y2\ ] \\
\text{or} \\
\ell_5: [\ \ell_{10}: \text{await } y2 > y1;\ \ell_6: y2 := y2 - y1\ ]
\end{array}\right] \\
\ell_7: g := y1
\end{array}\right] \\
\ell_8:
\end{array}\right]
$$

Etichette

- Le **etichette** in SPL vengono utilizzate con due obiettivi:
  - identificare univocamente le istruzioni
  - localizzare il controllo
- Più etichette possono individuare la stessa locazione del controllo. La seguente **relazione di equivalenza** $\sim_L$ sull'insieme delle etichette consente di identificare etichette associate alla medesima locazione:
  - da l : $[l_1 : S_1; \ldots; l_k : S_k]$, si ha $l \sim_L l_1$
  - da l : $[l_1 : S_1$ `or ...` `or` $l_k : S_k]$, si ha $l \sim_L l_1 \sim_L \ldots \sim_L l_k$
  - da l : `[local declaration;` $l_1 : S_1]$, si ha $l \sim_L l_1$

Locazione del Controllo

Locazioni del controllo e classi dell'equivalenza $\sim_L$

- Una **locazione del controllo** è una classe di equivalenza $[l_i]$ della relazione $\sim_L$
- Esempio:
  - $[l_1] = \{l_0, l_1\}$
  - $[l_4] = \{l_4\}$
  - $[l_6] = \{l_6\}$
  - $[l_7] = \{l_7\}$
  - $[l_8] = \{l_8\}$
  - $[l_2] = \{l_2, l_3, l_9, l_5, l_{10}\}$

Un Esempio di Programma SPL (rivisto)

calcolo del massimo comun divisore con l'algoritmo di Euclide

in a, b: integer where a>0, b>0
local y1, y2: integer where y1=a, y2=b
out g: integer

$$
\left[\begin{array}{l}
\ell_1\text{: while y1<>y2 do} \\
\quad \ell_2\text{: } \left[\begin{array}{l} \ell_{2a}\text{: await y1>y2; } \ell_4\text{: y1:=y1-y2} \\ \text{or} \\ \ell_{2b}\text{: await y2>y1; } \ell_6\text{: y2:=y2-y1} \end{array}\right] \\
\ell_7\text{: g:=y1} \\
\ell_8\text{:}
\end{array}\right]
$$

Le Post-Locazioni - 1

- Al termine dell'esecuzione di una qualunque istruzione, ad eccezione del corpo del programma, viene raggiunta un'unica locazione, detta **post-locazione** dell'istruzione.

Il legame tra la post-locazione di un'istruzione composta e le post-locazioni delle istruzioni componenti è espressa dalle seguenti regole:

- Concatenazione: $S = [l_1{:}S_1;\ \ldots;\ l_k{:}S_k]$
  - $post(S) = post(S_k)$
  - per ogni i, con $i < k$, $post(S_i) = [l_{i+1}]$

Le Post-Locazioni - 2

- Cooperazione: $S = [l_1{:}S_1;\widehat{l_1}\ :] \ || \ldots || \ [l_k{:}S_k;\widehat{l_k}\ :]$
  - $post(S_i) = [\widehat{l_i}]$
  - per ogni i, j, con $i \neq j$, $post(S_i) \neq post(S_j)$ (c'è l'exit step)
- Selezione: $S = [l_1{:}S_1 \text{ or } \ldots \text{ or } l_k{:}S_k]$
  - $post(S) = post(S_1) = \ldots = post(S_k)$
- Condizionale: $S = \mathit{if}\ c\ \mathit{then}\ S_1\ \mathit{else}\ S_2$
  - $post(S) = post(S_1) = post(S_2)$
- While: $S = [l : \mathit{while}\ c\ \mathit{do}\ S']$
  - $post(S') = [l]$
- Blocco: $S = [\mathit{local\ declaration},\ S']$
  - $post(S) = post(S')$

Esempio

Consideriamo il programma SPL descritto in precedenza.
Le post-locazioni sono le seguenti:

- $post(l_1) = [l_7]$
- $post(l_2) = post(l_4) = post(l_6) = [l_1]$
- $post(l_{2a}) = [l_4]$
- $post(l_{2b}) = [l_6]$
- $post(l_7) = [l_8]$

Osservazione: abbiamo identificato le istruzioni con le loro etichette.

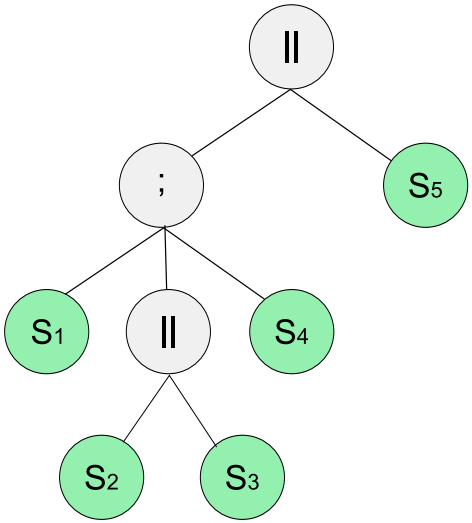
La Relazione di Antenato sull'Insieme delle Istruzioni di un Programma SPL - 1

- Se S' è una sottoistruzione di S, S è detto **antenato** di S'
  - per ogni S, S è *antenato* di S (riflessività)
  - se S è *antenato* di S_1 e S_1 è *antenato* di S_2, allora S è *antenato* di S_2 (transitività)
- S è detto *antenato comune* di S_1 ed S_2 se è *antenato* sia di S_1 che di S_2
- S è il *minimo antenato comune (LCA)* di S_1 e S_2 se
  - S è un *antenato comune* di S_1 e S_2
  - ogni *antenato comune* di S_1 ed S_2 è anche *antenato* di S

La Relazione di Antenato sull'Insieme delle Istruzioni di un Programma SPL - 2

Esempio: $[\ S_1;\ [\ S_2 \parallel S_3\ ];\ S_4\ ] \parallel S_5$

- LCA di S_2 ed S_3 è $S_2 \| S_3$
- LCA di S_2 ed $S_2 \| S_3$ è $S_2 \| S_3$
- LCA di S_2 ed S_4 è $[S_1;[S_2 \| S_3];S_4]$
- LCA di S_2 ed S_5 è l'intera istruzione



Istruzioni in Parallelo e in Conflitto

- Se S_i ed S_j hanno quale LCA un'istruzione di cooperazione S, con $S_i \neq S_j$, $S_i \neq S$ e $S_j \neq S$, S_i ed S_j si dicono **istruzioni parallele**;
- altrimenti, si dicono **istruzioni in conflitto**.
- Nell'esempio precedente
  - S_2 ed S_3 sono istruzioni parallele
  - S_2 ed $S_2||S_3$ sono istruzioni in conflitto
  - S_2 ed S_4 sono istruzioni in conflitto
  - S_2 ed S_5 sono istruzioni parallele

La Semantica e le Computazioni dei Programmi SPL

- La **semantica** di un programma SPL è un STE

$$\langle V, \theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$$

- Le **computazioni** di un programma SPL sono le computazioni del corrispondente programma STE.

V: Le Variabili - 1

- V colleziona l'insieme $Y = \{y_1, y_2, ..., y_n\}$ delle **variabili del programma** e una singola **variabile di controllo** π che assume come valori le locazioni del controllo durante l'esecuzione del programma

$$V = \{y_1, y_2, ..., y_n\} \cup \{\pi\}$$

- **Esempio**: algoritmo del massimo comun divisore
  - $V = \{\pi, a, b, y_1, y_2, g\}$,
    dove il dominio di π è l'insieme delle parti di
    $\{[l_1], [l_2], [l_4], [l_6], [l_7], [l_8]\}$

V: Le Variabili - 2

Osservazioni

- Il valore della variabile π è un insieme di locazioni $\{[l_1], ..., [l_k]\}$ (caso limite, insieme singoletto)
- per ogni canale asincrono α, viene introdotta una variabile α (con $\alpha \in Y$), i cui valori sono liste di elementi appartenenti al tipo del canale
- non vengono introdotte variabili per i canali sincroni

Utili **abbreviazioni**:

- per ogni etichetta l del programma, at_l è un'abbreviazione di $[l] \in \pi$
- per ogni etichetta l del programma, at'_l è un'abbreviazione di $[l] \in \pi'$

θ: La Condizione Iniziale

Dato il programma

P :: [declaration; [P_1::[l_1 : S_1; $\widehat{l_1}$:] || ... ||
P_k::[l_k : S_k; $\widehat{l_k}$:]]]

la **condizione iniziale** θ consiste nella congiunzione φ delle formule che compaiono nella clausola `where` delle dichiarazioni di variabili più la condizione sul controllo che forza quale locazione iniziale del controllo l'insieme delle locazioni di ingresso:

$$\theta = \varphi \wedge \pi = \{[l_1], ..., [l_k]\}$$

Esempio: algoritmo del massimo comun divisore

$$\theta = a > 0 \wedge b > 0 \wedge y_1 = a \wedge y_2 = b \wedge \pi = \{[l_1]\}$$

$\mathcal{T}$: Le Transizioni

L'**insieme delle transizioni** $\mathcal{T}$

- L'insieme $\mathcal{T}$ consiste di tutte e sole le transizioni STE che implementano le istruzioni del programma SPL dato, più la transizione τ_I (caratterizzata dalla relazione di transizione $\rho_I : V' = V$).
- Esclusa τ_I, tutte le altre transizioni sono autodisabilitanti (la loro esecuzione cambia almeno il valore della variabile di controllo π).

$\mathcal{T}$: Alcune Utili Abbreviazioni

- La scrittura $move(L, \widehat{L})$ è un'abbreviazione per $L \subseteq \pi \wedge \pi' = (\pi - L) \cup \widehat{L}$
- Come caso particolare, utilizziamo la scrittura $move(l, \widehat{l})$ per $move(\{[l]\}, \{[\widehat{l}]\})$ $(\equiv \{[l]\} \subseteq \pi \wedge \pi' = (\pi - \{[l]\}) \cup \{[\widehat{l}]\})$
- Dato che ogni transizione lascia inalterato il valore della maggior parte delle variabili, usiamo l'abbreviazione

$$pres(U) \equiv \bigwedge_{u \in U} (u' = u)$$

$\mathcal{T}$: Istruzioni di Base - 1

- l: `skip`; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$
  - modifica solo la locazione
- l: $\overline{u}$:=$\overline{e}$; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge \overline{u}' = \overline{e} \wedge pres(Y - \{\overline{u}\})$
  - modifica la locazione e il valore delle variabili in $\overline{u}$
- l: `await` c; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge c \wedge pres(Y)$
  - la transizione è abilitata quando la condizione c è vera; quando eseguita, modifica solo la locazione

$\mathcal{T}$: Istruzioni di Base - 2

- Send asincrono: l: $\alpha \Leftarrow e$; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge \alpha' = \alpha \cdot e \wedge pres(Y - \{\alpha\})$, dove $\cdot$ è l'operazione di concatenazione
  - il nuovo valore di α è dato dalla concatenazione del valore corrente di α col valore di e

- Receive asincrono: l: $\alpha \Rightarrow u$; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge |\alpha| > 0 \wedge \alpha = u' \cdot \alpha' \wedge pres(Y - \{u, \alpha\})$
  - la transizione è abilitata solo se il canale non è vuoto

$\mathcal{T}$: Istruzioni di Base - 3

- Send/receive sincrono:
  l: $\alpha \Leftarrow e; \widehat{l}$; $\quad m$: $\alpha \Rightarrow u; \widehat{m}$:
  - $\rho_{l,m}$: $move(\{l, m\}, \{\widehat{l}, \widehat{m}\}) \wedge u' = e \wedge pres(Y - \{u\})$
  - la transizione è abilitata se sono soddisfatte sia at_l sia at_m

- Request: l: `request` $r; \widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge r > 0 \wedge r' = r - 1 \wedge pres(Y - \{r\})$
  - la transizione è abilitata solo se r è positivo; quando viene eseguita, decrementa di 1 il valore di r

- Release: l: `release` $r; \widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge r' = r + 1 \wedge pres(Y - \{r\})$
  - quando viene eseguita, incrementa di 1 il valore di r

$\mathcal{T}$: Istruzioni Schematiche - 1

- l: `noncritical`; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$
- l: `critical`; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$

Come vedremo quando descriveremo le componenti legate alle condizioni di equità, ciò che differenzia la seconda dalla prima è la condizione di terminazione.

$\mathcal{T}$: Istruzioni Schematiche - 2

- l: `produce` x; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge x' \neq 0 \wedge pres(Y - \{x\})$
- l: `consume` y; $\widehat{l}$:
  - ρ_l: $move(l, \widehat{l}) \wedge pres(Y)$

Per entrambe è richiesta la terminazione.

$\mathcal{T}$: Istruzioni Composte - 1

- l: `if` c `then` $l_1 : S_1$ `else` $l_2 : S_2$; $\widehat{l}$:
  - ρ_l: $\rho_l^T \vee \rho_l^F$, dove
    ρ_l^T: $move(l, l_1) \wedge c \wedge pres(Y)$
    ρ_l^F: $move(l, l_2) \wedge \neg c \wedge pres(Y)$
  - genera un'unica transizione, non una coppia di transizioni, ed è, quindi, sempre abilitata (uno dei due disgiunti è soddisfatto)
  - caso particolare:
    l: `if` c `then` $l_1 : S_1$; $\widehat{l}$
    viene modificato il disgiunto ρ_l^F nel seguente modo:
    ρ_l^F: $move(l, \widehat{l}) \wedge \neg c \wedge pres(Y)$

$\mathcal{T}$: Istruzioni Composte - 2

- l: `while` c `do` $[\widetilde{l} : \widetilde{S}]; \widehat{l}$:
  - ρ_l: $\rho_l^T \vee \rho_l^F$, dove ρ_l^T: $move(l, \widetilde{l}) \wedge c \wedge pres(Y)$ e ρ_l^F: $move(l, \widehat{l}) \wedge \neg c \wedge pres(Y)$
  - genera un'unica transizione, non una coppia di transizioni, ed è, quindi, sempre abilitata (uno dei due disgiunti è soddisfatto)
  - la semantica della sottoistruzione $\widetilde{S}$ e le etichette: $\widetilde{l} : \widetilde{S}; l$:
- l: $[\ [l_1 : S_1; \widehat{l_1} :] \ || \ \ldots \ || \ [l_k : S_k; \widehat{l_k} :]]; \widehat{l}$:
  - ρ_l^E: $move(l, \{l_1, ..., l_k\}) \wedge pres(Y)$
  - ρ_l^X: $move(\{\widehat{l_1}, ..., \widehat{l_k}\}, \widehat{l}) \wedge pres(Y)$
  - non si considera il caso del corpo del programma

Ogni transizione appartenente all'implementazione delle istruzioni composte di concatenazione, selezione e blocco può essere attribuita ad una delle loro sottoistruzioni.

$\mathcal{T}$: Istruzioni Raggruppate - 1

Le **istruzioni raggruppate**: $\langle S \rangle$

Dato che le istruzioni raggruppate sono istruzioni atomiche (eseguite in un solo passo), dobbiamo determinare il loro effetto complessivo.

A tale fine introduciamo una **relazione di trasformazione dei dati** $\delta[S]$ che fa riferimento unicamente alle variabili in Y.

$\mathcal{T}$: Istruzioni Raggruppate - 2

Nei **casi più semplici** (`skip`, assegnamento, `await`, `send` asincrono e `receive` asincrono), $\delta[S]$ è definita come ρ_l, con l etichetta di S, privata di $move(l, \widehat{l})$

Nei **casi più complessi**, è definita nel seguente modo:

- when:
  - $\delta[\texttt{when } c \texttt{ do } S]$: $c \wedge \delta[S]$
- condizionale:
  - $\delta[\texttt{if } c \texttt{ then } S_1 \texttt{ else } S_2]$: $(c \wedge \delta[S_1]) \vee (\neg c \wedge \delta[S_2])$
  - caso particolare:
    $\delta[\texttt{if } c \texttt{ then } S]$: $(c \wedge \delta[S]) \vee (\neg c \wedge pres(Y))$

$\mathcal{T}$: Istruzioni Raggruppate - 3

- selezione:
  - $\delta[S_1 \texttt{ or } \ldots \texttt{ or } S_k]$: $\delta[S_1] \vee \ldots \vee \delta[S_k]$
- concatenazione:
  - $\delta[S_1; S_2]$: $\exists \widetilde{Y}(\delta[S_1](Y, \widetilde{Y}) \wedge \delta[S_2](\widetilde{Y}, Y'))$

Esempio

$\delta[x := x + 2; x := x - 3]$:
$\exists \widetilde{x}(\widetilde{x} = x + 2 \wedge x' = \widetilde{x} - 3) \wedge \mathit{pres}(Y - \{x\})$

$\mathcal{T}$: Istruzioni Raggruppate - 4

Le **istruzioni raggruppate**

- l: $\langle S \rangle$; $\widehat{l}$:
  - ρ_l: $\delta[S] \wedge move(l, \widehat{l})$ (ovviamente diversa dalla semantica di l: S; $\widehat{l}$:)

Istruzioni raggruppate che contengono **comunicazioni sincrone**

- l: $\langle T_1; \alpha \Leftarrow e; S_1 \rangle$; $\widehat{l}$:
  m: $\langle T_2; \alpha \Rightarrow u; S_2 \rangle$; $\widehat{m}$:
  - Assumiamo per semplicità che le variabili in $T_1 \cup S_1$ siano disgiunte quelle in $T_2 \cup \{u\} \cup S_2$ (unica variabile comune il canale sincrono α). In tal caso,

    $\rho_{l,m}$: $move(\{l, m\}, \{\widehat{l}, \widehat{m}\}) \wedge \delta[T_1; T_2; u := e; S_1; S_2]$

$\mathcal{J}$ e $\mathcal{C}$: Le Condizioni di Equità

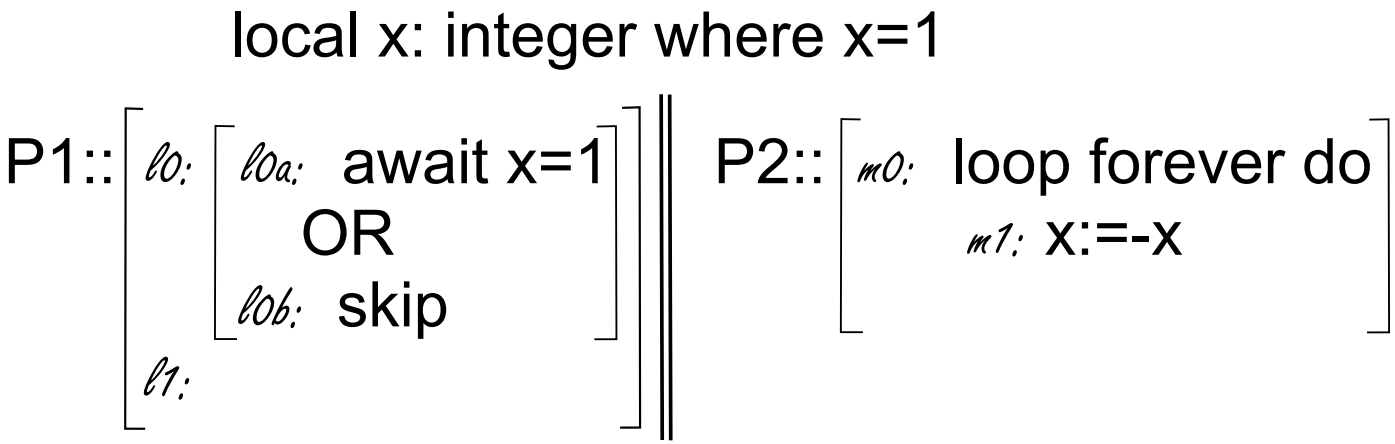
Transizioni per le quali chiediamo **giustizia**

- $\mathcal{J}$ include tutte le transizioni, ad eccezione della transizione idling e delle transizioni associate alle occorrenze dell'istruzione `noncritical` (il controllo può permanere indefinitamente di fronte ad esse)

Transizioni per le quali chiediamo **compassione**

- $\mathcal{C}$ include tutte le transizioni associate a
  - le istruzioni di comunicazione `send` e `receive`
  - le istruzioni raggruppate che contengono istruzioni di comunicazione
  - l'istruzione `request` per la gestione dei semafori (non l'istruzione `release` che esegue il rilascio di una risorsa, operazione per la quale non c'è ovviamente competizione)

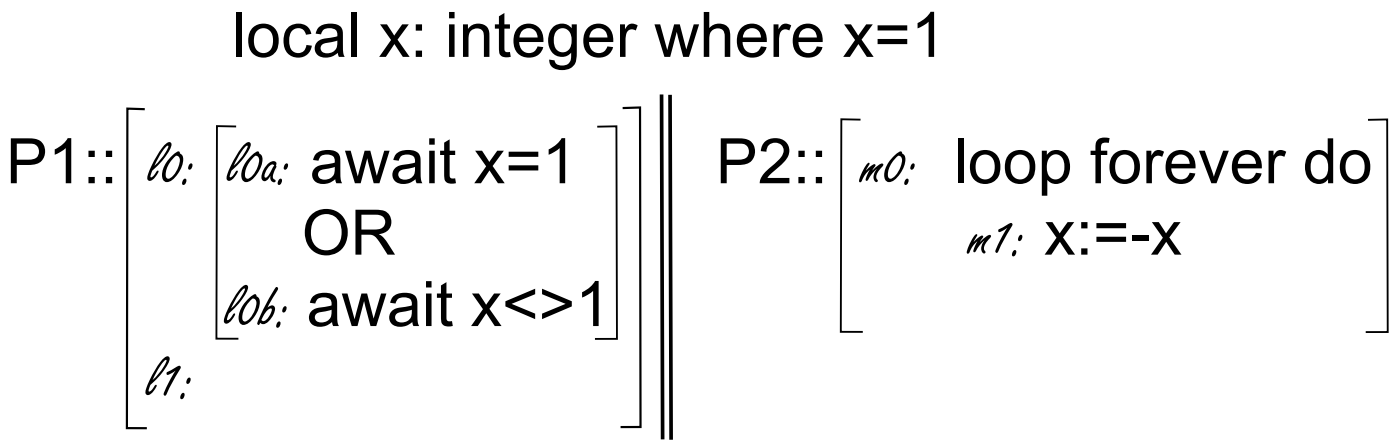
Esempio di Programma Reattivo



Proprietà del Programma Reattivo

- $P2$ non termina
- Esistono computazioni in cui $P1$ non termina?
  - Unica computazione candidata è:
    $\sigma = < \pi = \{l_o, m_0\}, x = 1 > \rightarrow^{m_0}$
    $< \pi = \{l_o, m_1\}, x = 1 > \rightarrow^{m_1}$
    $< \pi = \{l_o, m_0\}, x = -1 > \rightarrow^{m_0}$
    $< \pi = \{l_o, m_1\}, x = -1 > \rightarrow^{m_1}$
    $< \pi = \{l_o, m_0\}, x = 1 > \rightarrow^{m_0} \ldots$
  - **Non è una computazione ammissibile**: il requisito di giustizia è rispettato per l_{0a}, in quanto non è costantemente abilitata, ma non per l_{0b}, che è costantemente abilitata e mai eseguita

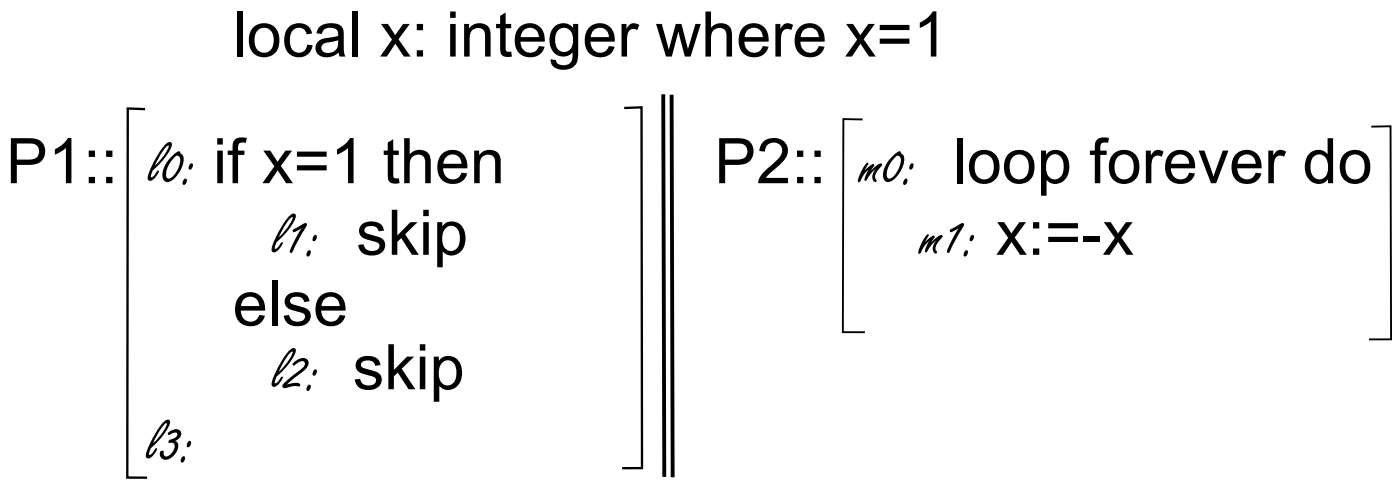
Esempio di Programma Reattivo - Variante 1



Proprietà del Programma Reattivo - Variante 1

- $P2$ non termina mai
- Esistono computazioni in cui $P1$ non termina?
  - L'unica computazione candidata è quella individuata in precedenza
  - In questo caso **è una computazione ammissibile**: il requisito di giustizia è rispettato sia per l_{0a} sia per l_{0b}, in quanto nessuna delle due è costantemente abilitata

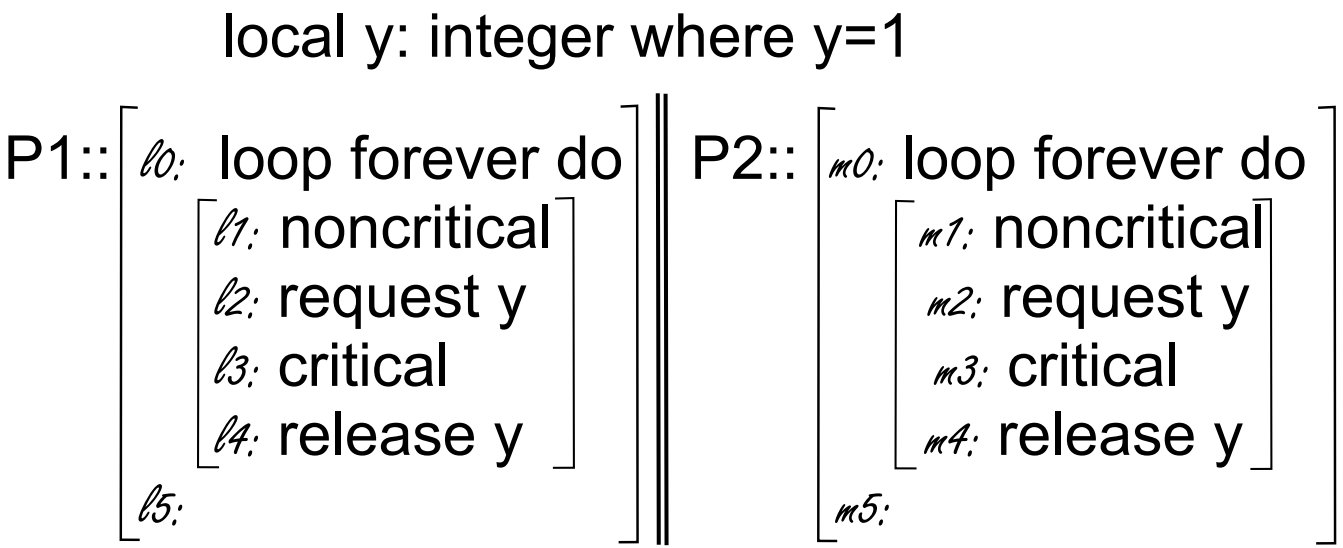
Esempio di Programma Reattivo - Variante 2



Proprietà del Programma Reattivo - Variante 2

- $P2$ non termina mai
- Esistono computazioni in cui $P1$ non termina?
  - Le computazioni candidate sono quelle in cui da un certo punto in poi la locazione l_0 (rispettivamente, l_1, l_2) appartiene stabilmente al valore di π
  - **Non sono computazioni ammissibili**: (la transizione corrispondente al)l'istruzione `if` è sempre abilitata e quindi deve essere prima o poi essere eseguita; se non viene mai eseguita si viola la condizione di giustizia, se viene eseguita, l'istruzione `skip` raggiunta è costantemente abilitata e quindi deve essere prima o poi essere eseguita; se non viene mai eseguita si viola la condizione di giustizia

Il Problema della Mutua Esclusione: il Programma MUX_SEM



La Proprietà di Mutua Esclusione

- **Mutua esclusione**: nessuna computazione del programma può includere uno stato nel quale entrambi i processi siano all'interno della loro sezione critica ($P1$ in $l3$ e $P2$ in $m3$).
- Se $P1$ si trova in $l3$, l'istruzione $l2$ ha posto il valore di y a 0 (decrementandolo di 1) e, fino a quando non viene eseguita l'istruzione $l4$, ogni eventuale tentativo di $P2$ di eseguire $m2$ non ha successo (la condizione abilitante $y > 0$ non è soddisfatta). Un argomento analogo vale nel caso in cui $P2$ si trovi in $m3$.

  Si noti che il processo $P1$ non può rimanere indefinitamente nella sezione critica, ossia non può rimanere indefinitamente nella posizione $l3$, in ragione del requisito di giustizia per l'istruzione `critical` (analoga osservazione vale per $P2$).

La Proprietà di Accessibilità - 1

- **Accessibilità**: ogni stato della computazione in cui un programma è all'uscita della sua sezione non critica ($P1$ in $l2$ o $P2$ in $m2$) deve essere seguito da uno stato in cui esso è all'interno della sezione critica ($P1$ in $l3$ o $P2$ in $m3$)
- Mostriamo come ogni computazione in cui ciò non accade non sia ammissibile.

  Assumiamo che $P1$ venga bloccato in $l2$ (il caso di $P2$ in $m2$ è analogo). Se da un certo punto in poi y rimane costantemente uguale a 1 ($P2$ indefinitamente nella sua sezione non critica), $l2$ è costantemente abilitata e mai eseguita (viene violato il requisito di giustizia che è implicato dal requisito da compassione).

La Proprietà di Accessibilità - 2

- Se y non rimane costantemente uguale a 1, ciò può accadere solo in ragione di una computazione della seguente forma:
  σ: $< \pi = \{l_0, m_0\}, y = 1 > \rightarrow \ldots < \pi = \{l_2, m_2\}, y = 1 > \rightarrow^{m_2}$
  $< \pi = \{l_2, m_3\}, y = 0 > \rightarrow^{m_3} < \pi = \{l_2, m_4\}, y = 0 > \rightarrow^{m_4}$
  $< \pi = \{l_2, m_0\}, y = 1 > \rightarrow^{m_0} < \pi = \{l_2, m_1\}, y = 1 > \rightarrow^{m_1}$
  $< \pi = \{l_2, m_2\}, y = 1 > \rightarrow \ldots$
  Tale computazione visita infinite volte lo stato
  $< \pi = \{l_2, m_2\}, y = 1 >$ nel quale la transizione l_2 è abilitata (viene violato il requisito di compassione).
  Ciò consente di concludere che il programma soddisfa la proprietà di accessibilità.

Il Problema della Mutua Esclusione: il Programma MUX_WHEN

local y: integer where y=1

P1:: [ℓ_0: loop forever do
[ℓ_1: noncritical
ℓ_2: <when y>0 do y:=y-1>
ℓ_3: critical
ℓ_4: y:=y+1]
ℓ_5:]

||

P2:: [m_0: loop forever do
[m_1: noncritical
m_2: <when y>0 do y:=y-1>
m_3: critical
m_4: y:=y+1]
m_5:]

Le Proprietà di Esclusività e di Accessibilità

- **Soddisfa** la proprietà di **mutua esclusione**: lo si può mostrare con un argomento analogo a quello usato per il programma precedente (è fondamentale l'uso di una istruzione raggruppata in $l2$ e $m2$).
- **Non soddisfa** la proprietà di **accessibilità**: la computazione descritta in precedenza in cui $l2$ risulta abilitata un'infinità di volte (ma non costantemente) e mai presa da un certo punto in poi è ammissibile (per $l2$ ed $m2$ è richiesta giustizia, non compassione).

La Proprietà di Accessibilità "Comune"

- La variante proposta **soddisfa** una forma debole di accessibilità, detta **accessibilità comune**, che afferma che ogni stato di una computazione in cui un processo si trova all'uscita della sua sezione non critica (ad esempio, la locazione $l2$ per $P1$) è seguita da uno stato in cui qualche processo, non necessariamente lo stesso, si trova all'interno della sua sezione critica (ad esempio, la locazione $m3$ per $P2$ e la locazione $l3$ per $P1$).

Produttore-Consumatore con Buffer

local send, ack: channel [1..] of integer
where send=Λ , ack=[1,...,1]

Prod:: [local x, t: integer
ℓ_0: loop forever do
[ℓ_1: produce x
ℓ_2: ack $\Rightarrow$ t
ℓ_3: send $\Leftarrow$ x]] $\|$ Cons:: [local y: integer
m_0: loop forever do
[m_1: send $\Rightarrow$ y
m_2: ack $\Leftarrow$ 1
m_3: consume y]]

dove ack contiene inizialmente n occorrenze di 1; $l2$ può essere eseguita solo se ack non è vuoto, $m1$ solo se send non è vuoto

Il Programma Fair-Merge

out b : channel of integer
local a1, a2 : channel of integer

P1:: [local x1: integer
ℓ0: loop forever do
[ℓ1: produce x1
ℓ2: a1 ⇐ 2•x1 + 1]] ||

P2:: [local x2: integer
k0: loop forever do
[k1: produce x2
k2: a2 ⇐ 2•x2]] ||

Merger:: [local y: integer
m0: loop forever do
m1: [m1a: a1 ⇒ y
or
m1b: a2 ⇒ y]
m2: b ⇐ y] ||

Cons:: [local z: integer
n0: loop forever do
[n1: b ⇒ z
n2: consume z]]