Daniele Gorla

Fondamenti della teoria della concorrenza: il CCS[1]

Università di Roma "La Sapienza"

Laurea Specialistica in Informatica

Dispense per il Corso di Teoria della Concorrenza

11 giugno 2009

[1]Queste dispense sono una rielaborazione personale di materiale classico sul CCS; il loro scopo è descrivere formalmente il calcolo e le sue basi matematiche, e darne semplici esempi d'uso. I primi quattro capitoli sono una elaborazione del materiale sul CCS presente in [12] (primi sette capitoli); il quinto capitolo è stato scritto prendendo spunto da [6] per la sezione 5.1, [7] per la sezione 5.2 e [5] per la sezione 5.3. Si rimanda a tali riferimenti per un approfondimento del materiale presentato in queste note. Ringrazio gli studenti del corso di Teoria della Concorrenza Alessandro Cammarano, Ornela Dardha, Antonio Faonio, Antonio Iaccarino e Matteo Pontecorvo per una prima stesura in latex di queste dispense.

Indice

1 Comportamento di automi **2**
- 1.1 La visione 'tradizionale' di un automa . . . 2
- 1.2 Una visione 'alternativa' di un automa . . . 3

2 Processi non-deterministici **6**
- 2.1 Labeled Transition System (LTS) . . . 6
- 2.2 Simulazione e Bisimulazione . . . 7
- 2.3 Proprietà della bisimulazione . . . 8
- 2.4 Una sintassi per processi non-deterministici . . . 9
- 2.5 Dalla sintassi all'LTS . . . 11

3 Processi concorrenti e interattivi **12**
- 3.1 CCS . . . 12
- 3.2 Proprietà dei processi CCS . . . 15
- 3.3 Esempio: specifica vs implementazione . . . 17
- 3.4 Congruenza della bisimulazione . . . 17

4 Equivalenza osservazionale **21**
- 4.1 Bisimulazione debole . . . 21
- 4.2 Differenze tra bisimulazione forte e debole . . . 22
- 4.3 Esempi di uso della bisimulazione debole . . . 24
  - 4.3.1 Azienda . . . 25
  - 4.3.2 Lotteria . . . 26
  - 4.3.3 Scheduler . . . 27

5 Metodi di verifica della bisimulazione **30**
- 5.1 Un sistema di inferenza . . . 30
  - 5.1.1 Il sistema di inferenza . . . 30
  - 5.1.2 Correttezza e completezza (per processi finiti) . . . 31
  - 5.1.3 Bisimulazione debole ed esempio d'uso . . . 34
- 5.2 Approccio logico . . . 36

5.2.1 Sintassi 37
5.2.2 Relazione di Soddisfacibilità 37
5.2.3 Sotto-logiche 40
5.3 Approccio algoritmico 41
5.3.1 Un primo tentativo 41
5.3.2 Un algoritmo efficiente 42
5.3.3 Correttezza 44
5.3.4 Complessità 45
5.3.5 Un esempio 47

Capitolo 1

Comportamento di automi

Lo scopo di queste dispense è quello di mostrare le basi della teoria della concorrenza. Ciò verrà fatto presentando un modello di processi concorrenti noto come CCS (abbreviazione di *Calculus of Communicating Systems*) sviluppato da Robin Milner all'università di Edimburgo alla fine degli anni '70 - inizio anni '80 [11]. Va però notato che tale modello non è l'unico presente in letteratura; altre importanti alternative, più o meno contemporanee a CCS, sono il CSP (sviluppato a Oxford da Tony Hoare [3, 8]) e l'ACP (sviluppato in Olanda [1, 2]).
Il termine *concorrenza* è inteso qui come l'esecuzione simultanea di processi indipendenti che interagiscono. Pertanto, l'*interazione* è una nozione basilare della concorrenza e può essere vista come sincronizzazione, comunicazione, mutua esclusione, conflitto etc. Il nostro obiettivo è quello di modellare i processi e di dare un insieme di concetti basilari con i cui descriverne l'interazione in modo rigoroso. In pratica, si vuole emulare ciò che negli anni '30/'40 venne fatto per modellare la *computazione*, utilizzando macchine di Turing o il λ-calcolo.
Una volta definito il modello di processi, un passo importante sarà la definizione di nozioni di *equivalenza* tra processi, perché attraverso esse potremo sstudiare il comportamento dei processi e, per esempio, stabililre se una implementazione rispetta la sua specifica.
Partiremo modellando i processi come automi non-deterministici. Pertanto, cominciamo con alcuni richiami sulla teoria degli automi.

1.1 La visione 'tradizionale' di un automa

Nella teoria dei linguaggi formali, gli automi (finiti) vengono usati per descrivere/accettare linguaggi (regolari). Ad esempio, l'automa di figura 1.1 accetta il linguaggio contenente tutte e sole le sequenze di a lunghe almeno 1. Formalmente, un *automa (finito non-deterministico)* è una quintupla $M = (Q, Act, q_0, F, T)$, dove:

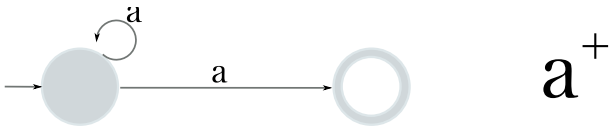


Figura 1.1: Un Automa

- Q è l'insieme degli *stati*,
- Act è l'insieme delle *azioni*,
- q_0 è lo stato *iniziale*,
- F è l'insime degli stati *finali*,
- T è la *relazione di transizione* ($T \subseteq Q \times Act \times Q$).

Il linguaggio di un automa M è definito come l'insieme di tutte le stringhe che portano l'automa M dallo stato iniziale a uno stato finale. Formalmente, $L(M) = \{w \in Act^* \mid T^*(q_0, w) \cap F \neq \emptyset\}$, dove $T^* : Q \times Act^* \to 2^Q$ è tale che

- $T^*(q, \varepsilon) = \{q\}$, per ogni $q \in Q$;
- $T^*(q, aw) = \bigcup_{r \in T(q,a)} T^*(r, w)$, per ogni $q \in Q$, $a \in Act$ e $w \in Act^*$.

Essendo gli automi usati (nella visione classica) come descrittori/accettori di linguaggi, l'equivalenza di automi viene naturalmente definita ponendo equivalenti tutti e soli quegli automi che descrivono/accettano lo stesso linguaggio. Formalmente, M_1 ed M_2 sono equivalenti se e solo se $L(M_1) = L(M_2)$.

1.2 Una visione 'alternativa' di un automa

La visione 'tradizionale' degli automi porta a interpretare una transizione $(q, a, q') \in T$ come: "se l'automa si trova nello stato q e *gli viene dato in input* il carattere a, allora evolve nello stato q'". La stessa transizione, però, può essere letta come: "se l'automa si trova nello stato q ed *esegue* l'azione a, allora evolve nello stato q'". Questa visione più 'comportamentale' di un automa è alla base del modello che andremo ora a discutere.
Presentiamo ora un semplice esempio che ci permetterà da un lato di discutere le potenzialità del modello considerato e dall'altro iniziare a ragionare sulla nozione di equivalenza da adottare nel nostro modello. Consideriamo due possibili implementazioni di una macchinetta distributrice di bevande, che per semplicità assumeremo essere solo caffè e tè, con il primo che costa 40 centesimi e il secondo 20:
Nell'implementazione di sinistra, dopo aver inserito 20 centesimi l'utente può scegliere se chiedere del tè o inserire altri 20 centesimi e chiedere del caffè;

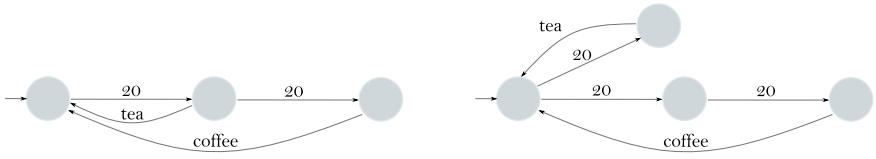


Figura 1.2: macchinetta del caffè/tè

nell'implementazione di destra questa scelta non viene fatta dall'utente ma dalla macchinetta al momento dell'insserimento dei primi 20 centesimi. Per l'utente, le due macchinette *non* sono equivalenti, nel senso che si *comportano* in maniera diversa e lui può accorgersene: mentre nella prima può sempre scegliere, dopo aver inserito 20 centesimi, se inserire altri 20 centesimi o chiedere il tè, questo non è possibile nella seconda macchinetta. Pertanto, vorremmo una nozione di equivalenza in grado di distinguere questi due automi. Infatti, l'equivalenza per linguaggi in questo caso non va bene, visto che i due automi accettano lo stesso linguaggio,[1] ovvero:

$$(20.tea + 20.20.coffe)^*$$

L'equivalenza per linguaggi non è pertanto adatta a eguagliare comportamenti di processi. Il problema di fondo (che è anche alla base della differenza tra le due implementazioni della macchinetta distributrice) è che l'equivalenza per linguaggi non distingue i seguenti automi:

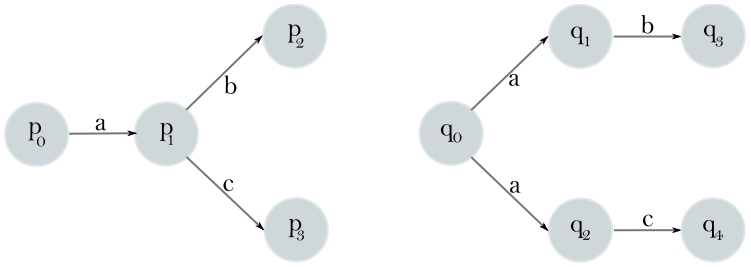


Figura 1.3: determinismo vs. non-determinismo

La differenza tra i comportamenti di questi due automi è che il secondo è "più non-deterministico" del primo, nel senso che compie una scelta (osservabile dall'esterno) che il primo non compie. Mentre l'equivalenza per linguaggi "appiattisce" il grado di non-determinismo (e infatti automi deterministici e non-deterministici sono equipotenti, nel senso che possono descrivere gli stessi linguaggi), noi vorremmo una nozione che tenga tale fattore in considerazione. In particolare, due automi sono equivalenti se dai loro stati iniziali si possono eseguire le stesse azioni e andare in stati da cui si possono ancora eseguire le

[1] Stiamo assumendo che entrambi gli automi hanno come unico stato finale lo stato iniziale.

stesse azioni.[2] Questo *non* è il caso nei due automi di figura 1.3 poiché, dopo la a, p_0 va in p_1 e quindi scegliere se fare b o c; al contrario, q_0, dopo aver fatto a, va in uno stato in cui o b o c non è eseguibile.
Formalmente, diremo che p è *simulato* da q se esiste una relazione $S \subseteq Q \times Q$ tale che:

- $(p, q) \in S$;
- $\forall (r, s) \in S \; \forall r \xrightarrow{a} r' \; \exists s \xrightarrow{a} s'.(r', s') \in S$.

Mostriamo adesso che i due automi introdotti non sono equivalenti. Per fare questo, proviamo a costruire la relazione S. Notiamo che questo non è possibile: infatti, affinché p_0 sia simulato da q_0, si dovrebbe avere che p_1 è simulato o da q_1 o da q_2. Ma se S contenesse (una tra) le coppie (p_1, q_1) e (p_1, q_2) non sarebbe una simulazione, visto che p_1 può fare una c (che invece q_1 non può fare) e una b (che invece q_2 non può fare). Invece, q_0 è simulato da p_0; ciò è mostrato dalla seguente relazione di simulazione:

$$S = \{(q_0, p_0), (q_1, p_1), (q_1', p_1), (q_2, p_2), (q_2', p_2')\}$$

Pertanto, i due processi non sono equivalenti.
Lo studio della relazione appena introdotta, delle sue proprietà e del suo uso concreto sarà il nucleo centrale di queste note.

[2] Questa nozione di equivalenza ricorda la condizione usata per la minimizzazione di un automa finito deterministico: due stati sono distinguibili se uno è finale e l'altro no (questo nel nostro caso corrisponderebbe al non poter eseguire le stesse azioni) o se, a fronte dello stesso input, vanno in stati distinguibili.

Capitolo 2

Processi non-deterministici

In questo capitolo definiamo formalmente il nostro modello di processi non-deterministici e l'equivalenza usata per ragionare su di essi.

2.1 Labeled Transition System (LTS)

In teoria della concorrenza, più che di automi si parla di *sistemi di trasizione etichettati* (LTS, dall'inglese *Labeled Transition System*). Le differenze principali tra i due formalismi sono:

1. negli automi il numero di stati è solitamente finito, mentre negli LTS no;
2. negli automi si fissa uno stato iniziale, mentre negli LTS qualunque stato può essere considerato iniziale (corrispondentemente a diversi comportamenti di processi);
3. negli automi si usa la nozione di stato finale per descrivere il linguaggio accettato dall'automa, mentre negli LTS (poiché il linguaggio non è molto rilevante) non serve avere tale nozione.

Da ora in poi, assumeremo fissato un insieme di nomi di azioni che verrà indicato con $\mathcal{N}$. Spesso questi nomi di azioni corrisponderanno a semplici lettere dell'alfabeto (a, b, c, etc.) o anche a nomi comuni ($tea, operaio$, etc.).

Definizione 2.1.1.

Un Labeled Transition System (LTS) *è una coppia* $(\mathcal{Q}, \mathcal{T})$*, dove* $\mathcal{Q}$ *è l'insieme degli stati e* $\mathcal{T}$ *è la relazione di transizione (*$\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{N} \times \mathcal{Q}$*).*

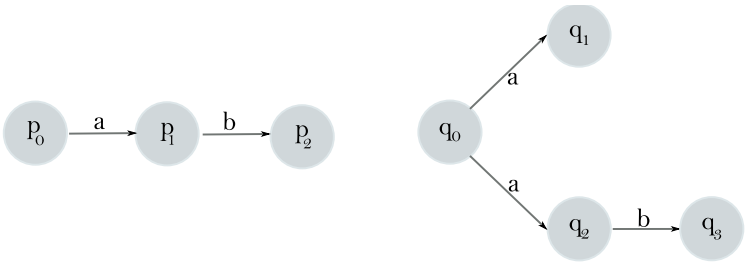
Useremo la notazione $s \xrightarrow{a} s'$ piuttosto che $\langle s, a, s' \rangle \in \mathcal{T}$. Questa notazione rende evidente che dallo stato s compiendo l'azione a si giunge nello stato s' (non si esclude la possibilità che s ed s' coincidano).

2.2 Simulazione e Bisimulazione

Definizione 2.2.1.

Sia $(\mathcal{Q}, \mathcal{T})$ un LTS. Una relazione binaria $\mathcal{S} \subseteq \mathcal{Q} \times \mathcal{Q}$ è una simulazione *se e solo se $\forall (p,q) \in \mathcal{S} \forall p \xrightarrow{a} p' \exists q \xrightarrow{a} q'$ tale che $(p', q') \in \mathcal{S}$.*
Diremo che p è simulato *da q se esiste una simulazione $\mathcal{S}$ tale che $(p,q) \in \mathcal{S}$.*
Diremo che $\mathcal{S}$ è una bisimulazione *se sia $\mathcal{S}$ che $\mathcal{S}^{-1}$ sono simulazioni (dove $\mathcal{S}^{-1} = \{(p,q) \ : \ (q,p) \in \mathcal{S}\}$).*
Due stati q e p sono equivalenti per bisimulazione *(o, più semplicemente* bisimili*) se esiste una bisimulazione $\mathcal{S}$ tale che $(p,q) \in \mathcal{S}$; scriveremo allora $p \sim q$.*

Il lettore noti che la simulazione è stata definita come una relazione sugli stati di un singolo LTS e non su quelli di due LTS distinti. Questa non è però una limitazione. Infatti, dati due LTS, se ne può fare l'unione disgiunta e lavorare con una relazione che permetta di simulare stati dell'unico LTS risultante.
Inoltre, l'intuizione alla base della bisimulazione è quella di uguagliare stati tali che il primo possa simulare il secondo e viceversa. Ciò però non vuol dire che basta traovare una simulazione di p tramite q e una simulazione di q tramite p. Lo mostriamo con un'esempio:



p_0 è simulato da q_0, come testimoniato dalla simulazione

$$S = \{(p_0, q_0), (p_1, q_2), (p_2, q_3)\}$$

e q_0 è simulato da p_0, come testimoniato dalla simulazione

$$S' = \{(q_0, p_0), (q_1, p_1), (q_2, p_1), (q_3, p_2)\}$$

Però p_0 non è bisimile a q_0: l'azione $q_0 \xrightarrow{a} q_1$ non è *bisimulabile* da nessuna azione di p_0 (infatti, $p_0 \xrightarrow{a} p_1$ non va bene, visto che p_1 può fare una b e quindi non può essere bisimile a q_1). Quindi, l'esistenza di due *diverse* simulazioni è una condizione più debole dell'essere bisimile (cioè l'esistenza di una simulazione la cui inversa sia ancora una simulazione).
Prima di studiare in dettaglio le proprietà della bisimulazione, vale la pena chiedersi se la relazione appena definita non sia troppo discriminante, ad esempio

non sia una differente definizione della funzione identità. Per fare ciò, mostriamo due LTS (o, volendo, un LTS con due componenti connesse) e la verifica (non banale) della loro equivalenza.

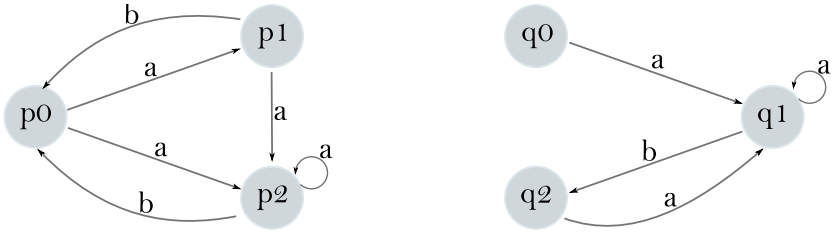


Figura 2.1: Esempio di bisimilarità

Per gli LTS in figura 2.1 utilizziamo la seguente relazione:

$$\mathcal{S} = \{(p_0, q_0), (p_1, q_1), (p_2, q_1), (p_0, q_2)\}$$

Consideriamo la coppia (p_0, q_0); per definizione si deve verificare che, per ogni azione $p \xrightarrow{a} p'$, esiste un azione $q \xrightarrow{a} q'$ tale che $(p', q') \in \mathcal{S}$. Da p_0 è possibile effettuare due azioni a che portano negli stati p_1 e p_2. Prendendo in considerazione q_0, si osserva che compiendo l'azione a si giunge allo stato q_1. Ci si chiede dunque se le due coppie (p_1, q_1) e (p_2, q_1) appartengano o meno alla simulazione $\mathcal{S}$. Tale vincolo è soddisfatto. Analizzando allo stesso modo le coppie restanti, si conclude che p_0 è simulato da q_0. Per completare, è necessario mostrare che anche

$$\mathcal{S}^{-1} = \{(q_0, p_0), (q_1, p_1), (q_1, p_2), (q_2, p_0)\}$$

è una simulazione. Procedendo in maniera del tutto equivalente a come è stato fatto precedentemente, si conclude che $\mathcal{S}$ è una bisimulazione.

2.3 Proprietà della bisimulazione

La bisimulazione gode di alcune importante proprietà. Mostriamo a seguire in questo paragrafo alcuni enunciati e le rispettive prove.

Teorema 2.1. *La bisimulazione è una relazione di equivalenza.*

Dimostrazione.

Riflessività: Si vuole mostrare che $q \sim q$, per ogni q. Tale proprietà è ovvia considerando la seguente relazione

$$\mathcal{S} = \{(p, p) : p \in \mathcal{Q}\}$$

e osservando che è una bisimulazione (è una simulazione e anche la sua inversa - cioè se stessa - lo è).

Simmetria: Si vuole mostrare che $p \sim q$ implica $q \sim p$, per ogni p, q. Per ipotesi sappiamo che esiste una simulazione $\mathcal{S}$ per cui $p \sim q$. Per definizione di bisimulazione si ha che $\mathcal{S}^{-1}$ è una simulazione; quindi $(q, p) \in \mathcal{S}^{-1}$, da cui $q \sim p$.

Transitività: Si vuole mostrare che $p \sim q$ e $q \sim r$ implicano $p \sim r$, per ogni p, q, r. Consideriamo la seguente relazione

$$\mathcal{S} = \{(x, z) : \exists y \ \ t.c. \ \ (x, y) \in \mathcal{S}_1 \wedge (y, z) \in \mathcal{S}_2\}$$

in cui $\mathcal{S}_1$ ed $\mathcal{S}_2$ sono bisimulazioni; mostriamo che $\mathcal{S}$ è una bisimulazione. Sia $(x, z) \in \mathcal{S}$ e $x \xrightarrow{a} x'$. Se (x, z) appartiene a $\mathcal{S}$, allora per definizione esise y tale che $(x, y) \in \mathcal{S}_1$ e $(y, z) \in \mathcal{S}_2$. Poichè S_1 è una bisimulazione, esiste $y \xrightarrow{a} y'$ tale che $(x', y') \in \mathcal{S}_1$. Poichè S_2 è una bisimulazione, esiste $z \xrightarrow{a} z'$ tale che $(y', z') \in \mathcal{S}_2$. Quindi, partendo da $x \xrightarrow{a} x'$, abbiamo trovato $z \xrightarrow{a} z'$ tale che $(x', z') \in \mathcal{S}$, visto che esiste un y' tale che $(x', y') \in \mathcal{S}_1$ e $(y', z') \in \mathcal{S}_2$.

□

Un altro risultato mostra che la bisimilarità è essa stessa una bisimulazione.

Teorema 2.2. *$\sim$ è una bisimulazione.*

Dimostrazione. Per dimostrare l'asserto è sufficiente mostrare che $\sim$ è una simulazione. Seguendo la definizione di similarità si vuole dimostrare che

$$\forall (p, q) \in \sim \ \ \forall p \xrightarrow{a} p' \ \exists q \xrightarrow{a} q' \ t.c. \ (p', q') \in \sim$$

Fissiamo una coppia (p, q) che appartiene a $\sim$. La bisimilarità di p e q implica l'esistenza di una bisimulazione $\mathcal{S}$ t.c. $(p, q) \in \mathcal{S}$. Dunque per ogni transizione $p \xrightarrow{a} p'$ esiste una transizione $q \xrightarrow{a} q'$ per cui $(p', q') \in \mathcal{S}$. Da questo ovviamente segue l'asserto. □

2.4 Una sintassi per processi non-deterministici

Il formalismo degli LTS è molto naturale e maneggevole per descrivere visivamente il comportamento di processi non-deterministici 'piccoli'. Tuttavia, quando il numero di stati e/o transizioni cresce (potenzialmente, potrebbe anche essere infinito), tale otazione diventa poco conveniente. Così come nei linguaggi regolari si usano grammatiche e/o espressioni regolari invece di automi quando il linguaggio da descrivere diventa troppo complesso, così noi ora definiremo una sintassi per processi che ci permetta di scrivere in maniera più compatta e leggibile gli LTS.
Consideriamo ad esempio la prima macchinetta distributrice di figura 1.2; se chiamiamo con A, B e C i suoi tre stati, possiamo descriverne il comportamento

con il seguente sistema di equazioni:

$$\begin{cases} A = 20.B \\ B = tea.A + 20.C \\ C = coffee.A \end{cases}$$

dove con '.' si indica la composizione sequenziale e con '+' la scelta non-deterministica. Sostituendo la terza equazione nella seconda e poi il risultato nella prima, si ottiene la seguente definizione ricorsiva del comportamento della macchinetta

$$A = 20.(tea.A + 20.coffee.A)$$

Chiaramente, per un LTS così semplice il guadagno di questa espressione rispetto all'LTS non è evidente; ma lasciamo immaginare al lettore casi più complessi in cui il vantaggio sia lampante.
Gli unici ingredienti usati per descrivere LTS sono quindi la composizione sequenziale di un'azione seguita da un processo, la scelta non-deterministica tra un insieme finito di azioni e la ricorsione. Tali ingredienti saranno per il momento gli unici operatori della nostra sintassi. Tuttavia, per semplificare la scrittura di processi, assumeremo di avere un insieme infinito Id di identificatori di processo e che le definizioni che daremo siano parametriche, come quelle per le procedure in un linguaggio di programmazione standard. Per ogni identificatore (denotato con lettere maiuscole $A, B, \dots$), assumeremo un'unica definizione della forma

$$A(x_1, x_2, .., x_n) \triangleq P$$

dove i nomi $x_1, x_2, .., x_n$ sono tutti distinti e sono inclusi nei nomi di P. Denotiamo con $P\{b_1/x_1 \dots b_n/x_n\}$ il processo P in cui il nome x_i è stato rimpiazzato dal nome b_i, per ogni $i = 1, .., n$.
Definiamo formalmente i processi.

Definizione 2.4.1.

L'insieme dei processi non-deterministici è definito dalla seguente sintassi:

$$P ::= \sum_{i \in I} \alpha_i.P_i \mid A(a_1 \dots a_n)$$

dove I è un insieme finito di indici e $\alpha_i \in \mathcal{A}$, per ogni $i \in I$.

Notiamo che, rispetto a quanto detto prima, abbiamo fuso in un unico operatore la composizione sequenziale e la scelta non-deterministica. Se l'insieme degli indici I è vuoto, allora $\sum_{i \in I} \alpha_i.\mathcal{P}_i$ è il processo terminato, che non è in gradi di compiere alcuna azione; per tale processo verrà utilizzato il simbolo $\mathbf{0}$.

2.5 Dalla sintassi all'LTS

Abbiamo mostrato come sia possibile, partendo da un LTS, generare un processo corrispondente. É anche possibile il passagio inverso (e quindi i due formalismi coincidono); le regole da utilizzare per compiere tale trasformazione sono:

$$\sum_{i\in I} \alpha_i.\mathcal{P}_i \xrightarrow{\alpha_j} \mathcal{P}_j \quad \text{per ogni } j \in I$$

$$\frac{P\{a_1/x_n \ldots a_n/x_n\} \xrightarrow{\alpha} P'}{A(a_1 \ldots a_n) \xrightarrow{\alpha} P'} \; A(x_1 \ldots x_n) \triangleq P$$

Si noti che tale definizione è un modo alternativo di definire induttivamente l'LTS di un processo: il caso base si ha per le somme, il caso induttivo per le invocazioni di processo.
Mostriamo ora l'uso della sintassi appena definita e del sistema di inferenza per la derivazione di LTS su due semplici esempi. Il primo esempio è un contatore per numeri naturali: c'è un processo C_0 che simula lo zero (può avere successori ma non predecessori) e per ogni $i > 0$ un processo C_i che può essere incrementato e decrementato. Assumendo le azioni *inc* e *dec*, ciò può essere formalizzato ponendo

$$C_0 = inc.C_1$$
$$C_i = inc.C_{i+1} + dec.C_{i-1} \quad \text{per ogni } i > 0$$

Usando le regole di inferenza, l'LTS per C_0 è

$$C_0 \xrightarrow{inc} C_1 \underset{dec}{\overset{inc}{\rightleftarrows}} C_2 \underset{dec}{\overset{inc}{\rightleftarrows}} C_3 \ldots$$

Si noti che tale LTS ha infiniti stati.
Progettiamo ora un buffer FIFO di dimensione massima due e contenente valori booleani; dunque uno stato generico del buffer (descritto dalla sequenza dei valori memorizzati correntemente nel buffer) apparterrà all'insieme $\{\epsilon, 0, 1, 00, 01, 10, 11\}$, dove con ϵ indichiamo la sequenza vuota. Siano i e j due elementi di $\{0, 1\}$; utilizzeremo la seguente notazione:

- B_ϵ indica il buffer vuoto;
- B_i indica il buffer con il solo bit i;
- B_{ij} indica il buffer in cui sono stati inseriti nell'ordine i bit i e j.

Denotando con in_i e con out_i le azioni di inserimento e di prelevamento dell'elemento i dal buffer, possiamo definire le equazioni dei processi in questo modo:

- $B_\epsilon = \sum_{i\in\{0,1\}} in_i.B_i$
- $B_i = \sum_{j\in\{0,1\}} in_j.B_{ij} + out_i.B_\epsilon$
- $B_{ij} = out_i.B_j$

Al lettore si lascia la costruzione dell'LTS a partire dalle precedente definizione.

Capitolo 3

Processi concorrenti e interattivi

Fino ad ora abbiamo considerato unicamente il caso dei processi in esecuzione isolata. In questo capitolo analizzeremo come più processi possano essere eseguiti concorrentemente (cioè in parallelo) interagendo tra loro. Una prima scelta da fare è il tipo di parallelismo che si vuole modellare. Nel periodo in cui il CCS è nato, la problematica tipica dei sistemi operativi era quella di avere più processi in esecuzione sulla stessa macchina, dotata di un unico processore.[1] Tale modello di parallelo viene detto a *interleaving* ed è radicalmente diverso da quello in cui i processi sono in esecuzione simultanea su più processori, detto modello *truly concurrent*.
Nello scenario a interleaving, i processi vengono eseguiti alternandosi sull'unico processore e interagiscono accedendo alla stessa memoria condivisa. Noi adotteremo un modello lievemente più astratto, dove i processi interagiscono sincronizzandosi su eventi, seguendo il modello produttore-consumatore (un processo produce un evento, un altro lo consuma).

3.1 CCS

Per modellare interazioni binarie tra processi basate su sincronizzazione produttore-consumatore, considereremo come azioni oltre che i semplici nomi $\mathcal{N}$, anche $\overline{\mathcal{N}} = \{\bar{a} \ : \ a \in \mathcal{N}\}$, dove i due insiemi sono disgiunti. Penseremo a due azioni a e $\bar{a}$ come azioni complementari che permetteranno a due processi di sincronizzarsi sull'evento a (per esempio, a denota l'attesa dell'evento e $\bar{a}$ la sua produzione). Quando due processi si sincronizzano, un osservatore esterno non ha modo di capire cosa sta succedendo nel sistema, se non che due processi si stanno sincronizzando; in altre parole, una sincronizzazione non è osservabile e, quindi, all'esterno produce un'azione 'silenziosa' special che denoteremo con τ. Pertanto, un osservatore che dall'esterno vede un'azione visibile (a o $\bar{a}$) sa che

[1] Questo scenario è rimasto attuale fino a pochi anni fa, quando sono emerse a livello commerciale le prime architetture multicore.

all'interno del sistema c'è un processo che consuma/produce un evento di nome a ed, eventualmente, può fornirlo/consumarlo al sistema osservato; se invece osserva τ non ha modo di interagire con il sistema e non sa quale evento è stato prodotto e consumato. In conclusione, l'insieme di azioni da considerare sono $\mathcal{A} = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$.
É inoltre utile poter forzare alcuni processi di un sistema a sincronizzarsi tra loro senza la possibilità di mostrare all'esterno alcune azioni. A tale scopo, si introduce un operatore di restrizione $P\backslash a$ che restringe lo scopo del nome a al processo P, con l'idea che a sia visibile solo all'interno di P. Questo è simile alle variabili locali in una procedura di un programma imperativo: il loro significato è ristretto alla procedura in cui sono definite ed eventuali variabili globali omonime definite esternamente alla procedura sono in realtà diverse dalla variabile locale.

Definizione 3.1.1.

L'insieme dei processi CCS è definito dalla seguente sintassi:

$$P \ ::= \ \sum_{i \in I} \alpha_i.P_i \mid A(a_1 \dots a_n) \mid P|Q \mid P\backslash a$$

dove I è un insieme finito di indici e $\alpha_i \in \mathcal{A}$, per ogni $i \in I$.

Diamo dunque una semantica per i nuovi costrutti:

Regola di inferenza 3.1.1.

$$\sum_{i \in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j \quad \textit{per ogni } j \in I$$

$$\frac{P\{a_1/x_n \dots a_n/x_n\} \xrightarrow{\alpha} P'}{A(a_1 \dots a_n) \xrightarrow{\alpha} P'} \ A(x_1 \dots x_n) \stackrel{\triangle}{=} P$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1|P_2 \xrightarrow{\alpha} P_1'|P_2} \qquad \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1|P_2 \xrightarrow{\alpha} P_1|P_2'}$$

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{\bar{a}} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'} \qquad \frac{P_1 \xrightarrow{\bar{a}} P_1' \quad P_2 \xrightarrow{a} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P\backslash a \xrightarrow{\alpha} P'\backslash a} \ \alpha \notin \{a, \bar{a}\}$$

Studiamo un esempio per comprendere meglio la semantica dei nuovi costrutti. Siano

- $A \triangleq a.A'$
- $A' \triangleq \bar{b}.A$
- $B \triangleq b.B'$
- $B' \triangleq \bar{c}.B$

Il parallelo dei processi A e B ha il seguente LTS.

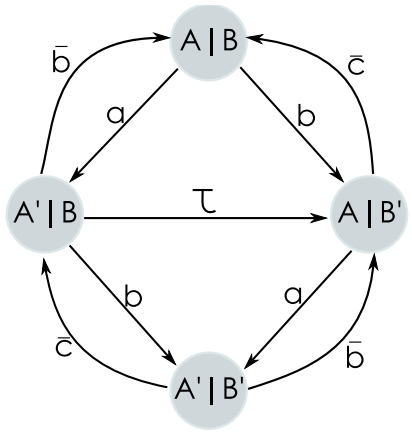


Figura 3.1: LTS di $A|B$

Notiamo che nella costruzione dell'LTS si perde la coscienza del parallelo. É infatti possibile, ammettendo il nuovo insieme di azioni, ottenere l'LTS di figura 3.1 mediante la sintassi della definizione 2.4.1. L'utilità del parallelo è duplice: da un lato, è l'operatore fondamentale in teoria della concorrenza; dall'altro permette una scrittura compatta e intuitiva dei processi. Inoltre, se si lascia una semantica ad interleaving per una truly concurrent, l'uso del parallelo diventa fondamentale.
Se volessimo fare in modo che solo un determinato insieme di eventi sia visibile all'esterno (ovvero attraverso l'LTS) è necessario restringere gli scopi dei processi. In figura 3.2 è mostrato l'LTS che deriva dalla restrizione del parallelo dei processi A e B sul nome b, ovvero il processo $(A \mid B)\backslash b$. L'effetto principale che si nota in figura è stato quello di eliminare transizioni etichettate con b (tale nome viene nascosto all'ambiente esterno). In generale, è possibile che interi stati scompaiano: questo sarebbe il caso se in $(A \mid B)\backslash b$ restringessimo lo scopo anche del nome c.

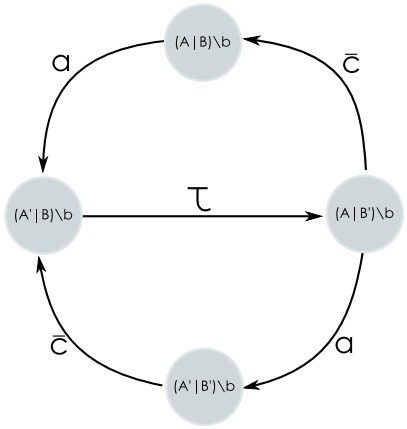


Figura 3.2: LTS di $A|B$ con restrizione su b

3.2 Proprietà dei processi CCS

In questa sezione mostreremo alcune proprietà dei processi CCS. La prima importante proprietà è quella del *finite branching*, che afferma che il numero di processi a cui si può giungere partendo da un processo iniziale è finito.

Teorema 3.1. $\mid \{P' : \exists \alpha . P \xrightarrow{\alpha} P'\} \mid \; < \infty$

Dimostrazione. Per ogni processo P, sia $\mathcal{T}(P)$ l'insieme degli alberi di derivazione per $P \xrightarrow{\alpha} P'$, per ogni$\alpha$ e per ogni P'. Denotiamo con k_P l'altezza massima di un albero in $\mathcal{T}(P)$. Per induzione su k_P, mostriamo che l'insieme $\{P' : \exists \alpha . P \xrightarrow{\alpha} P'\}$ ha cardinalità finita.

Caso base ($k = 0$):
In questo caso, deve essere $P \triangleq \sum_{i \in I} \alpha_i . P_i$. In tale caso abbiamo

$$\mid \{P' : \exists \alpha . P \xrightarrow{\alpha} P'\} \mid \; = \; \mid \bigcup_{i \in I} \{P_i\} \mid \; \leq |I| < \infty$$

Passo induttivo:
Dobbiamo analizzare tre casi.

1. $P \triangleq A(a_1 \dots a_n)$. In questo caso, tutti gli alberi di inferenza per P avranno la forma

$$\frac{Q\{a_1/x_n \dots a_n/x_n\} \xrightarrow{\alpha} Q'}{A(a_1 \dots a_n) \xrightarrow{\alpha} Q'} \; A(x_1 \dots x_n) \triangleq Q$$

dove gli alberi di inferenza per $Q\{a_1/x_n \dots a_n/x_n\}$ avranno altezza inferiore a k_P. Per induzione

$$\mid \{R : \exists \alpha . Q\{a_1/x_n \dots a_n/x_n\} \xrightarrow{\alpha} R\} \mid \; < \infty$$

La tesi segue osservando che

$$\{P' : \exists\alpha.P \xrightarrow{\alpha} P'\} = \{R : \exists\alpha.Q\{a_1/x_n \dots a_n/x_n\} \xrightarrow{\alpha} R\}$$

2. $P \triangleq Q\backslash a$. Per induzione si ha che

$$\mid \{Q' : \exists\alpha.Q \xrightarrow{\alpha} Q'\} \mid \; < \infty$$

e la tesi segue osservando che

$$\{P' : \exists\alpha.P \xrightarrow{\alpha} P'\} \subseteq \{Q' : \exists\alpha.Q \xrightarrow{\alpha} Q'\}$$

3. $P \triangleq P_1 \mid P_2$. Per questo caso, si osservi che:

$$\begin{aligned}\{P' : \exists\alpha.P \xrightarrow{\alpha} P'\} = \quad & \{P_1'|P_2 : \exists\alpha.P_1 \xrightarrow{\alpha} P_1'\} \\ & \cup\{P_1|P_2' : \exists\alpha.P_2 \xrightarrow{\alpha} P_2'\} \\ & \cup\{P_1'|P_2' : \exists a.P_1 \xrightarrow{a} P_1' \wedge P_2 \xrightarrow{\bar{a}} P_2'\} \\ & \cup\{P_1'|P_2' : \exists a.P_1 \xrightarrow{\bar{a}} P_1' \wedge P_2 \xrightarrow{a} P_2'\}\end{aligned}$$

La cardinalità è dunque al più la somma delle cardinalità dei 4 insiemi. Per induzione abbiamo che ogni insieme ha una cardinalità finita da cui segue l'asserto.

□

Teorema 3.2. *Per ogni $\sigma : \mathcal{N} \to \mathcal{N}$, se $P \xrightarrow{\alpha} P'$ allora $\sigma(P) \xrightarrow{\sigma(\alpha)} \sigma(P')$.*

Dimostrazione. Si procede per induzione sull'altezza dell'albero di derivazione per il giudizio $P \xrightarrow{\alpha} P'$.

Caso Base: L'unico caso da considerare sono le somme $\sum_{i\in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j$. In questo caso $\alpha = \alpha_j$ per un qualche $j \in I$ e $P' = P_j$ per il medesimo j. Il processo $\sigma(P)$ è uguale a $\sum_{i\in I} \sigma(\alpha_i).\sigma(P_i)$, da cui banalmente segue che $\sigma(P) \xrightarrow{\sigma(\alpha_j)} \sigma(P_j) \;\; \forall j \in I$.

Passo induttivo:
Come in precedenza è necessario analizzare tre casi.

1. $P \triangleq A(a_1 \dots a_n)$, $A(x_1 \dots x_n) \triangleq Q, Q\{a_1/x_n \dots a_n/x_n\} \xrightarrow{\alpha} P'$. La sostituzione σ fa si che $\sigma(P) = A(\sigma(a_1) \dots \sigma(a_n))$. Applicando l'ipotesi induttiva su Q si ottiene

$$Q\{\sigma(a_1)/x_n \dots \sigma(a_n)/x_n\} \xrightarrow{\sigma(\alpha_j)} \sigma(P')$$

da cui segue l'asserto.

2. $P \stackrel{\triangle}{=} Q\backslash a$. Si procede banalmente utilizzando l'ipotesi induttiva su $Q \xrightarrow{\alpha} Q'$.

3. $P \stackrel{\triangle}{=} P_1 \mid P_2$. Si lascia la prova al lettore; si definisca $\sigma(\tau) = \tau$ (visto che $\tau \notin \mathcal{N}$) e si analizzino tutti i possibili casi.

□

3.3 Esempio: specifica vs implementazione

Concludiamo questo capitolo con un'esempio sui semafori. Un semaforo n-ario $S^{(n)}(p, v)$ è un processo utilizzato per garantire che non ci siano più di n istanze della stessa attività in esecuzione concorrente. L'attività viene cominciata esibendo l'azione p e viene terminata esibendo l'azione v. Le specifiche per un semaforo unario e binario sono le seguenti:

$$\begin{aligned} S^{(1)} &\triangleq p \cdot S_1^{(1)} \\ S_1^{(1)} &\triangleq v \cdot S^{(1)} \end{aligned}$$

$$\begin{aligned} S^{(2)} &\triangleq p \cdot S_1^{(2)} \\ S_1^{(2)} &\triangleq p \cdot S_2^{(2)} + v \cdot S^{(2)} \\ S_2^{(2)} &\triangleq v \cdot S_1^{(2)} \end{aligned}$$

dove il pedice indica il numero delle attività che sono in esecuzione concorrente. Possiamo notare, intuitivamente, che un semaforo binario si comporta come due semafori unari messi in parallelo, ovvero $S^{(1)} \mid S^{(1)}$. Se vediamo $S^{(2)}$ come la specifica del comportamento atteso di un semaforo binario e $S^{(1)} \mid S^{(1)}$ come la sua implementazione concreta, possiamo dimostrare che

$$S^{(1)} \mid S^{(1)} \sim S^{(2)}$$

cioè, che l'implementazione data è corretta rispetto alla specifica. Per dimostrare questa equivalenza basta dimostrare che la relazione

$$R = \{(S^{(1)} \mid S^{(1)}\,,\, S^{(2)})\,,\, (S_1^{(1)} \mid S^{(1)}\,,\, S_1^{(2)})\,,\, (S^{(1)} \mid S_1^{(1)}\,,\, S_1^{(2)})\,,\, (S_1^{(1)} \mid S_1^{(1)}\,,\, S_2^{(2)})\}$$

è una bisimulazione forte. La prova è lasciata al lettore.

3.4 Congruenza della bisimulazione

Uno degli scopi principali di una nozione di equivalenza tra processi, e quindi della bisimulazione, è quello di fare ragionamenti equazionali del tipo "se P e Q sono equivalenti, allora possono essere usati in maniera intercambiabile in qualunque contesto d'esecuzione". Formalizziamo ora questo concetto e verifichiamo che la bisimulazione permette di fare ragionamenti di questo tipo.

Non è infatti detto, in generale, che ogni equivalenza sia preservata per contesti; non è detto, cioè, che ogni equivalenza sia una *congruenza*.
Prima di fornire una definizione rigorosa di contesto, cerchiamo di inquadrare intuitivamente questo concetto. Informalmente, osservando la figura 3.3 possiamo vederlo come un processo 'incompleto' o anche un processo particolare che permette di inserire un qualsiasi altro processo in una posizione fissata.

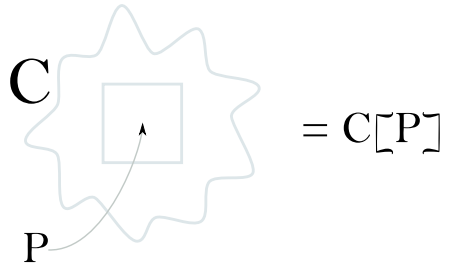


Figura 3.3: Schema logico del concetto di contesto

La posizione riservata per 'inserire' il processo verrà indicata con il simbolo $\Box$. Per esempio, se C è $(\Box \mid Q)\backslash a$, allora $C[P]$ è $(P \mid Q)\backslash a$; cioè, in $C[P]$ abbiamo sostituito il $\Box$ con il processo P. Passiamo alla definizione formale.

Definizione 3.4.1.

L'insieme C dei contesti è definito dalla seguente sintassi:

$$C \ ::= \ \Box \ \mid \ C|P \ \mid \ C\backslash a \ \mid \ \alpha.C + M$$

dove M è una somma.

Il lettore si sarà chiesto perchè la definizione di contesto non comprenda il termine $C + M$; il motivo di tale esclusione deriva dal fatto che tale termine consentirebbe di generare termini che non appartengono al linguaggio dei processi. Per esempio, prendendo $C \ ::= \ \Box + M$ e $P \ ::= \ Q|Q$, avremmo che $C[P]$ non appartiene ai termini del CCS, poichè lì tutti gli elementi di una somma sono sempre preceduti da un azione α.
Com'è stato anticipato l'obiettivo di tale capitolo è dimostrare che la bisimulazione è una congruenza. Definiamo formalmente tale proprietà.

Definizione 3.4.2.

Una relazione di equivalenza $\Re$ è una congruenza se e solo se

$$\forall (P, Q) \in \Re, \ \forall C.(C[P], C[Q]) \in \Re$$

A seguire mostriamo alcuni importanti proprietà che ci permetteranno di mostrare che la bisimulazione $\sim$ è una congruenza.

Lemma 3.3. *Se $P \sim Q$ allora $\alpha.P + M \sim \alpha.Q + M$, per ogni α e M.*

Dimostrazione. A tal fine è sufficiente provare che

$$S = \{(\alpha.P + M, \alpha.Q + M) : P \sim Q, \alpha \in \mathcal{A}, M = \sum_{i \in I} \alpha_i.P_i\} \cup \sim$$

è una simulazione. Sia $(\alpha.P + M, \alpha.Q + M) \in S$ e $\alpha.P + M \xrightarrow{\beta} P'$. Il termine M è una somma quindi $\alpha.P + M = \alpha.P + \sum_{i \in I} \alpha_i.P_i$. Si possono verificare due casi:

1. $\beta = \alpha$ e $P' = P$
2. $\beta = \alpha_j$ per un $j \in I$ e $P' = P_j$

Nel primo caso, il processo $\alpha.Q + M$ può rispondere con l'azione α riducendosi a Q che è per definizione bisimile a P. Nel secondo, il processo $\alpha.Q + M$ può rispondere effettuando la medesima azione riducendosi allo stesso processo. □

Lemma 3.4. *Se $P \sim Q$ allora $P\backslash a \sim Q\backslash a$, per ogni a.*

Dimostrazione. Anche in questo caso mostriamo che

$$S = \{(P\backslash a, Q\backslash a) : \forall P \sim Q, \forall a \in \mathcal{N}\}$$

è una simulazione. Sia $(P\backslash a, Q\backslash a) \in S$ e $P\backslash a \xrightarrow{\alpha} P'$. Il processo P' avrà la forma $P''\backslash a$, per $P \xrightarrow{\alpha} P''$ e $\alpha \notin \{a, \bar{a}\}$. Banalmente il processo Q può effettuare la stessa azione α e ridursi in un processo $Q' = Q''\backslash a$. Dal momento che $P \sim Q$ si ha che $P'' \sim Q''$ da cui segue l'asserto. □

Lemma 3.5. *Se $P \sim Q$ allora $P|R \sim Q|R$, per ogni R.*

Dimostrazione. Mostriamo che

$$S = \{(P|R, Q|R) \ : \ P \sim Q\}$$

è una simulazione. Sia $(P|R, Q|R) \in S$ e sia $P|R \xrightarrow{\alpha} P'$. Dobbiamo analizzare tre casi:

1. $P \xrightarrow{\alpha} P''$ e $P' = P''|R$;
2. $R \xrightarrow{\alpha} R'$ e $P' = P|R'$;
3. $P \xrightarrow{a} P'' \wedge R \xrightarrow{\bar{a}} R''$ (o viceversa) e $P' = P''|R''$, con $\alpha = \tau$.

Analizziamo schematicamente i tre casi

1. Dato che $P \sim Q$, si ha che $Q \xrightarrow{\alpha} Q''$ e $P'' \sim Q''$; dunque $Q|R \xrightarrow{\alpha} Q''|R$ e $(P''|R, Q''|R) \in S$.
2. Banalmente, $Q|R \xrightarrow{\alpha} Q|R'$ e $(P|R', Q|R') \in S$.

3. Dato che $P \sim Q$, si ha che $Q \xrightarrow{a} Q''$ e $Q'' \sim P''$. Quindi, $Q|R \xrightarrow{\tau} Q''|R'' = Q'$ e $(P', Q') \in S$.

□

Dalle proprietà appena esposte segue il risultato che c'eravamo prefissati.

Teorema 3.6. *Se $P \sim Q$ allora $\forall C.\ C[P] \sim C[Q]$.*

Dimostrazione. Per induzione sulla struttura di C.

Caso Base: L'unico contesto da analizzare è il contesto $C = \square$. Per tale contesto si ha che $C[P] = P$, per ogni P. Dunque $C[P] \sim C[Q]$ per ipotesi.
Passo induttivo: Ragioniamo sulla struttura di C e sfruttiamo i lemmi precedenti. □

Per concludere, dimostriamo tre semplici equivalenze:

Proposizione 3.7.

- $\alpha.P + \alpha.P + M \sim \alpha.P + M$
- $a.P\backslash a \sim 0$
- $\bar{a}.P\backslash a \sim 0$

in cui il termine M indica una somma $\sum_{i \in I} \beta_i.P_i$.

Dimostrazione. Analizziamo i tre casi separatamente:

- Per entrambi i processi è possibile effettuare unicamente un'azione all'interno dell'insieme $\{\alpha\} \cup \{\beta_i \ : \ i \in I\}$. Effettuando una di queste azioni essi si riducono a due processi identici, che quindi sono banalmente bisimili.
- Per il processo di sinistra esiste un unica azione possibile, l'azione a su cui però c'è una restrizione. Pertanto, processo in questione non può far nulla e quindi si comporta come il processo **0**.
- Analogo al punto precedente.

□

Capitolo 4

Equivalenza osservazionale

L'equivalenza studiata finora è piuttosto discriminante, nel senso che distingue, ad esempio, $\tau.P$ da $\tau.tau.P$. Se un osservatore esterno può 'cotare' il numero di azioni non osservabili (cioè i τ), questa distinzione è auspicabile; ma se supponiamo che un osservatore non abbia accesso ad alcuna informazione interna al sistema, allora questa distinzione non è accettabile. In questo capitolo vogliamo introdurre una nozione di equivalenza che uguaglia i processi considerando solamente il loro comportamento osservabile, cioè le loro azioni visibili. Tali processi possono differire nella loro struttura interna e di conseguenza avere un comportamento interno diverso.

4.1 Bisimulazione debole

L'idea della nuova equivalenza è di ignorare i τ: pertanto, a un'azione visibile si deve rispondere con la stessa azione, possibilmente assieme ad azioni interne; a un'azione interna si deve rispondere con una sequenza (anche vuota) di azioni interne. Per formalizzare in maniera semplice questa idea, definiamo ora una relazione di transizione 'estesa', in cui in un singolo passo si possono fare più azioni silenziose, ma al più un'azione osservabile. Sia $\Longrightarrow$ la chiusura riflessiva e transitiva di $\stackrel{\tau}{\longrightarrow}$; cioè, $P \Longrightarrow P'$ se e solo se esistono $P_0, P_1, \ldots, P_k$ (per $k \geq 0$) tali che $P = P_0 \stackrel{\tau}{\longrightarrow} P_1 \stackrel{\tau}{\longrightarrow} \ldots \stackrel{\tau}{\longrightarrow} P_k = P'$. Definiamo ora la relazione $\stackrel{\widehat{\alpha}}{\Longrightarrow}$ come segue:

- se $\alpha = \tau$ allora $\stackrel{\widehat{\alpha}}{\Longrightarrow} \triangleq \Longrightarrow$;
- altrimenti $\stackrel{\widehat{\alpha}}{\Longrightarrow} \triangleq \Longrightarrow \stackrel{\alpha}{\longrightarrow} \Longrightarrow$.

Definiamo adesso la *simulazione debole* sugli stati di un LTS $(\mathcal{Q}, \mathcal{T})$.

Definizione 4.1.1.

S è una simulazione debole *se e solo se* $\forall (p,q) \in S\ \forall p \xrightarrow{\alpha} p'\ \exists\ q'$ *tale che* $q \stackrel{\widehat{\alpha}}{\Longrightarrow} q'$ *e* $(p',q') \in S$.
Una relazione S viene detta bisimulazione debole *se sia S che S^{-1} sono simulazioni deboli.*
Diciamo che p e q sono debolmente bisimili*, scritto $p \approx q$, se esiste una bisimulazione debole S tale che* $(p,q) \in S$.

Notiamo che la definizione di bisimulazione debole è analoga a quella della bisimulazione data nel capitolo 2 (detta anche *bisimulazione forte*), solo che nella risposta di q a p si usa la relazione "$\stackrel{\widehat{\alpha}}{\Longrightarrow}$" invece della relazione "$\xrightarrow{\alpha}$". Ciò permette di ignorare i τ nell'uguagliare processi, come ci eravamo prefissi di fare.
Analogamente alle corrispondenti proprietà per la bisimulazione forte, si possono dimostrare le seguenti proprietà per la bisimulazione debole.

Proposizione 4.1.

1. *Se $P \sim Q$ allora $P \approx Q$.*
2. *$\approx$ è un'equivalenza.*
3. *$\approx$ è una bisimulazione debole.*
4. *$\approx$ è una congruenza.*

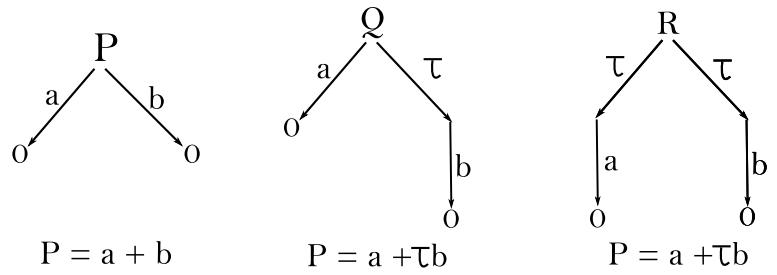
Inoltre, banalmente, si ha che

Proposizione 4.2. $\sim \subset \approx$.

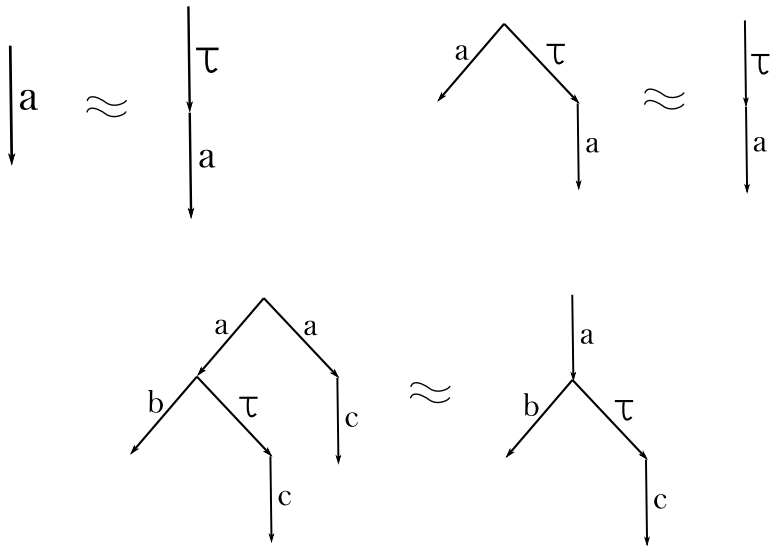
Quindi, ogni bisimulazione forte è anche una bisimulazione debole (ma non viceversa); pertanto, ogni coppia di processi fortemente bisimili è una coppia di processi debolmente bisimili.

4.2 Differenze tra bisimulazione forte e debole

Fino adesso abbiamo parlato delle analogie tra bisimulazione debole e bisimulazione forte. Passiamo adesso a considerare le loro differenze. Per come è definita, la bisimulazione debole è meno sensibile alle transizioni τ. Questo ci potrebbe portare a pensare che ogni processo è debolmente equivalente ad uno senza transizioni τ. Il seguente esempio mostra come la presenza di azioni interne porta a differenze nel comportamento dei processi:



I tre processi P, Q ed R sono a due a due debolmente non equivalenti. Ad esempio, consideriamo i processi P e Q e mostriamo che non esiste nessuna bisimulazione debole S che contenga la coppia (P, Q).[1] Supponiamo che una tale bisimulazione esista; pertanto, presa $Q \xrightarrow{\tau} b.\mathbf{0}$, deve esistere un P' tale che $P \stackrel{\tau}{\Longrightarrow} P'$ e $(P', b.\mathbf{0}) \in S$. L'unico P' che soddisfa $P \stackrel{\tau}{\Longrightarrow} P'$ è P stesso e quindi dovrebbe essere $(P, b.\mathbf{0}) \in S$. Ma questo contraddirebbe il fatto che S è una bisimulazione, in quanto $P \xrightarrow{a} \mathbf{0}$ mentre $b.\mathbf{0} \stackrel{a}{\not\Longrightarrow}$.
Invece, certi comportamenti sono equivalenti:



Generalizziamo adesso gli esempi precedenti con il seguente teorema:

Teorema 4.3. *Dato P un processo qualsiasi e M, N delle sommatorie, allora:*

1. $P \approx \tau.P$;
2. $M + N + \tau.N \approx M + \tau.N$;
3. $M + \alpha.P + \alpha.(N + \tau.P) \approx M + \alpha.(N + \tau.P)$.

[1] In modo simile si mostra che non esiste nessuna bisimulazione che contenga le coppie (P, R) e (Q, R).

Dimostrazione. Per dimostrare il teorema, basta esibire per ogni caso una bisimulazione debole. A tale scopo, basta prendere la chiusura simmetrica delle seguenti relazioni, che si dimostrano essere simulazioni deboli:

1. $S = \{(P, \tau.P)\} \cup Id$
2. $S = \{(M + N + \tau.N, M + \tau.N)\} \cup Id$
3. $S = \{(M + \alpha.P + \alpha.(N + \tau.P), M + \alpha.(N + \tau.P))\} \cup Id$

□

Un'altra equivalenza debole è la seguente:

$$M + \alpha.P + \tau.(N + \alpha.P) \approx M + \tau.(N + \alpha.P) \qquad (*)$$

la quale, anche se sembrerebbe una derivabile dalla (3), è in realtà equivalente alla (2); mostriamo cioè che (2) $\iff$ (*).

(2) $\implies$ (*):

$$\begin{array}{rcl} M + \alpha.P + \tau.(N + \alpha.P) \triangleq M' + \tau.N' & \approx & M' + N' + \tau.N' \\ & = & M + \alpha.P + N + \alpha.P + \tau.(N + \alpha.P) \\ & \approx & M + N + \alpha.P + \tau.(N + \alpha.P) \\ & = & M + N' + \tau.N' \\ & \approx & M + \tau.N' = M + \tau.(N + \alpha.P) \end{array}$$

La prima equivalenza debole si ha per (2); la seconda segue dalle proposizioni 3.7(1) e 4.2; infine,

(*) $\implies$ (2): Sia $N = \Sigma_{i=1}^{k} \alpha_i.P_i$; procediamo per casi:

- se $k = 0$, allora $N = \mathbf{0}$ e quindi $M + N + \tau.N = M + \tau.N$ per definizione.
- se invece $k \neq 0$, allora

$$\begin{array}{rcl} M + \tau.N & = & M + \tau.(\Sigma_{i=1}^{k} \alpha_i.P_i) = M + \tau.(\Sigma_{i=1}^{k-1} \alpha_i.P_i + \alpha_k.P_k) \\ & \approx & M + \alpha_k.P_k + \tau.(\Sigma_{i=1}^{k-1} \alpha_i.P_i + \alpha_k.P_k) \\ & = & M' + \tau.(\Sigma_{i=1}^{k-2} \alpha_i.P_i + \alpha_k.P_k + \alpha_{k-1}.P_{k-1}) \\ & \approx & M'\alpha_{k-1}.P_{k-1} + \tau.N \\ & \approx & \ldots \\ & \approx & M + \Sigma_{i=1}^{k} \alpha_i.P_i + \tau.N \end{array}$$

Notiamo che ogni occorrenza di $\approx$ deriva da (*) e abbiamo denotato con M' il processo $M + \alpha_k.P_k$.

4.3 Esempi di uso della bisimulazione debole

Vediamo ora degli esempi di equivalenza debole tra sistemi, composti da diverse componenti concorrenti che interagiscono, e la loro specifica. Ovvero che:

$$Sistema \approx Specifica$$

4.3.1 Azienda

Un'azienda è in grado di gestire 3 tipi di lavori: facili (F), medi (M) e difficili (D). Un'attività dell'azienda consiste nel ricevere in input un lavoro (di qualsiasi tipo) e produrne in output una elaborazione. Pertanto, la specifica di un'aattività è la seguente:

$$\begin{array}{rcl} A & \triangleq & i_F.A' + i_M.A' + i_D.A' \\ A' & \triangleq & \overline{o}.A \end{array}$$

dove le azioni i_F, i_M, i_D rappresentano l'input di un lavoro facile, medio o difficile, e $\overline{o}$ rappresenta la produzione di un output. L'azienda è data dalla composizione parallela di due attività, ovvero:

$$Azienda \triangleq A|A$$

Una possibile implementazione di questa specifica può essere ottenuta avendo due lavoratori che fanno in parallelo i vari tipi di lavoro. Per i lavori facili non usano macchinari; per i lavoro medi usano o un macchinario speciale oppure uno generale; per i lavori difficile usano il macchinario speciale. Esiste, però, un solo macchinario speciale e uno solo generale che i lavoratori devono condividere. La specifica di un lavoratore è quindi:

$$\begin{array}{rcl} L & \triangleq & i_F.L_F + i_M.L_M + i_D.L_D \\ L_F & \triangleq & \overline{o}.L \\ L_M & \triangleq & \overline{rg}.\overline{lg}.L_F + \overline{rs}.\overline{ls}.L_F \\ L_D & \triangleq & \overline{rs}.\overline{ls}.L_F \\ S & \triangleq & rs.S' \\ S' & \triangleq & ls.S \\ G & \triangleq & rg.G' \\ G' & \triangleq & lg.G \end{array}$$

dove rg ed rs servono per richiedere il macchinario generale o speciale, lg ed ls servono per liberare il macchinario generale o speciale, e S e G implementano dei semafori sui due tipi di macchinari (si veda a tale riguardo la sezione 3.3). Il sistema in questione è dato da:

$$Lavoratori \triangleq (L \mid L \mid G|S)\backslash_{\{rg,rs,lg,ls\}}$$

Vogliamo dimostrare la seguente equivalenza debole:

$$Azienda \approx Lavoratori$$

Ci basta esibire una bisimulazione debole. Per semplicità, denotiamo con N l'insieme $\{rg, rs, lg, ls\}$, e siano $x, y \in \{F, M, D\}$. La reelazione cercata è la chiusura simmetrica della relazione di simulazione debole:

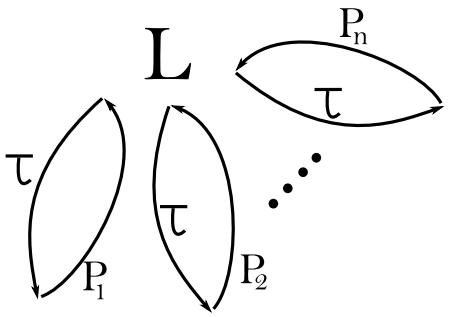
$$\begin{array}{rl} R = \{ & (A|A, (L|L \mid G|S)\backslash_N)\ ,\ (A|A', (L|L_x \mid G|S)\backslash_N)\ , \\ & (A|A', (L|\overline{lg} \cdot L_F \mid G'|S)\backslash_N)\ ,\ (A|A', (L|\overline{ls} \cdot L_F \mid G|S')\backslash_N)\ , \\ & (A'|A', (L_y|L_x \mid G|S)\backslash_N)\ ,\ (A'|A', (L_y|\overline{lg} \cdot L_F \mid G'|S))\backslash_N\ , \\ & (A'|A', (L_y|\overline{ls} \cdot L_F \mid G|S'))\backslash_N\ ,\ (A'|A', (\overline{ls} \cdot L_F|\overline{ls} \cdot L_F \mid G'|S')\backslash_N)\} \end{array}$$

4.3.2 Lotteria

Vogliamo costruire una lotteria L che permetterà di selezionare in modo casuale una pallina da un sacchetto che ne contiene n; dopo ogni estrazione, si rimette la pallina estratta nel sacchetto e si ripete la procedura. La specifica della lotteria è la seguente:

$$L \triangleq \tau.\bar{p}_1.L + \tau.\bar{p}_2.L + \ldots + \tau.\bar{p}_n.L$$

dove i τ rappresentano l'estrazione di una pallina e p_i è l'azione che comunica all'esterno il valore della pallina selezionata. L'LTS risultante da questa specifica è



A partire da questa specifica, costruiamo un sistema con n componenti, uno per ogni pallina. Ogni componente può trovarsi in 3 stati: A (in attesa di essere abilitata all'estrazione), B (abilitata, con possibilità di essere estratta o abilitare la componente successiva) o C (estratta, in attesa di comunicare all'esterno il suo valore e di riavviare il processo):

$$\begin{array}{lll} A_i = & a_i.B_i \\ B_i = & \tau.C_i + \bar{a}_{(i+1) \mod n}.A_i \\ C_i = & \bar{p}_i.B_i \end{array}$$

Il sistema è quindi

$$Lott \triangleq (B_1|A_1|\ldots|A_n)\backslash_{\{a_1,\ldots,a_n\}}$$

che genera l'LTS di figura 4.1 dove

$$\begin{array}{lll} L_i = & (A_1|A_2|\ldots A_{i-1}|B_i|A_{i+1}|\ldots|A_n)\backslash_{\{a_1,\ldots a_n\}} \\ L'_i = & (A_1|A_2|\ldots A_{i-1}|C_i|A_{i+1}|\ldots|A_n)\backslash_{\{a_1,\ldots a_n\}} \end{array}$$

Vogliamo mostrare che questa implementazione del sistema è corretta rispetto alla specifica data, ovvero che

$$L \approx Lott$$

Dimostriamo un risultato più generale, cioè che per ogni $i = 1, .., n$ si ha che

$$L \approx L_i$$

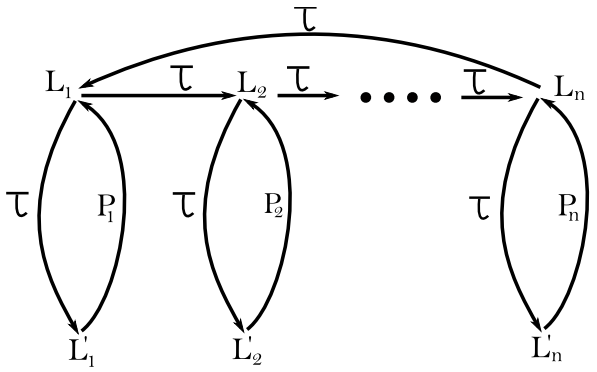


Figura 4.1:

Visto che $Lott = L_1$, ciò basta per concludere. Per fare questo, basta mostrare che

$$S \triangleq \{(L, L_i) \mid 0 \leq i \leq n\} \cup \{(p_i.L, L'_i) \mid 0 \leq i \leq n\}$$

e S^{-1} sono simulazioni deboli.

- Scegliamo una coppia (L, L_i) e consideriamo $L \xrightarrow{\tau} p_j.L$; banalmente, $L_i \Longrightarrow L'_j$ e $(p_j.L, L_j) \in S$. Scegliamo ora una coppia $(p_i.L, L_i)$ e consideriamo $p_i.L \xrightarrow{p_i} L$; banalmente, $L'_i \xrightarrow{p_i} L_i$ e $(L, L_i) \in S$. Quindi, S è una simulazione debole.
- Consideriamo (L_i, L) e $L_i \xrightarrow{\tau} L'_i$; allora $L \xrightarrow{\tau} p_i.L$ e $(L'_i, p_i.L) \in S^{-1}$. Consideriamo $L_i \xrightarrow{\tau} L_{(i+1) \mod n}$; allora $L \Longrightarrow L$ e $(L_{(i+1) \mod n}, L) \in S^{-1}$. Quindi S^{-1} è una simulazione debole.

4.3.3 Scheduler

Un insieme di processi P_i (per $0 \leq i \leq n$) deve svolgere un certo compito ripetutamente. Per fare questo, uno scheduler deve garantire che i processi inizino il loro task ciclicamente, cominciando da P_1. Ogni processo segnala la terminazione del suo compito allo scheduler. Le diverse esecuzioni possono sovrapporsi, ma lo scheduler deve garantire che ogni processo P_i finisca la sua performance prima di cominciarne un'altra (con lo stesso indice i). Ogni processo P_i richiede di iniziare il suo task tramite l'azione a_i e segnala allo scheduler la sua terminazione tramite l'azione b_i. Quindi, lo scheduler deve garantire che le a_i occorrano ciclicamente, cominciando con a_1, e per ogni i le a_i si devono alternare con le b_i, cominciando con a_i. La specifica dello scheduler è:

$$S_{i,X} = \begin{cases} \Sigma_{j \in X} b_j.S_{i,X-\{j\}} & \text{se } i \in X \\ a_i.S_{(i+1) \mod n, X \cup \{i\}} + \Sigma_{j \in X} b_j.S_{i,X-\{j\}} & \text{altrimenti} \end{cases}$$

$S_{i,X}$ denota il sistema in attesa che si attivi il processo P_i e in cui sono in esecuzione i processi i cui indici sono in X. La configurazione iniziale è $S_{1,\emptyset}$.

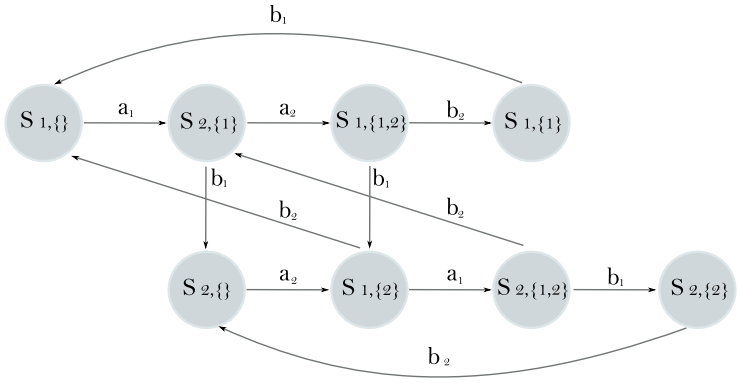
Implementiamo adesso lo scheduler nel seguente modo:

$$\begin{aligned} A_i = &\quad a_i.\bar{c}_{(i+1) \mod n}.b_i.c_i.A_i \\ S = &\quad (A_1|c_2.A_2|\ldots|c_n.A_n)\backslash_{\{c_1\ldots c_n\}} \end{aligned}$$

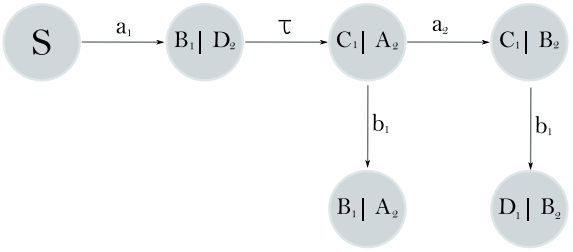
In questa implementazione, le azioni di tipo a e b hanno lo stesso ruolo che nella specifica: segnalare l'inizio e la fine di un processo. Le azioni di tipo $\bar{c}$ servono per segnalare al processo di indice successivo che può iniziare a operare quando vuole, mentre quelle di tipo c servono per rivecere dal processo di indice precedente tale segnalazione.
Vogliamo ora dimostrare che

$$S \approx S_{1,\emptyset}$$

Diamo l'LTS per $S_{1,\emptyset}$, fissando per semplicità $n = 2$:



mentre (una parte del)l'LTS per S è:



dove scriviamo

$$\begin{aligned} A_i = &\quad a_i.B_i \\ B_i = &\quad \bar{c}_{(i+1) \mod n}.C_i \\ C_i = &\quad b_i.D_i \\ D_i = &\quad c_i.A_i \\ S = &\quad (A_1|D_2)\backslash_{\{c_1,c_2\}} \end{aligned}$$

Si vede che l'implementazione dello scheduler non risulta equivalente alla specifica, poiché $S_{1,\{1,2\}}$ può fare b_2 mentre $C_1 \mid B_2$ no.

Invece, un'implementazione corretta è la seguente:

$$\begin{aligned} A_i = &\quad a_i.\bar{c}_{(i+1) \mod n}.(b_i.c_i.A_i + c_i.b_i.A_i) \\ S = &\quad (A_1|D_2|\ldots|D_n)\backslash_{\{c_1\ldots c_n\}} \end{aligned}$$

Si invita il lettore a scrivere l'LTS completo di questa seconda implementazione (richiede 12 stati) e mostrare che è debolmente bisimile all'LTS della specifica.

Capitolo 5

Metodi di verifica della bisimulazione

Per concludere, proponiamo ora alcuni metodi alternativi per dimostrare che due processi sono (o non sono) bisimili. Iniziamo fornendo un sistema di inferenza, grazie a cui la bisimilarità tra due processi verrà stabilita costruendo una derivazione di tale uguaglianza a partire da assiomi e usando regole di inferenza. Presentiamo poi un approccio logico, secondo cui due processi sono bisimili se e solo se soddisfano lo stesso insieme di formule di una opportuna logica modale; tale approccio risulterà molto naturale per mostrare che due processi *non* sono bisimili. Infine, presenteremo un approccio algoritmico, in grado di verificare in maniera efficiente se due stati di un LTS sono bisimili o meno.

5.1 Un sistema di inferenza

In questo approccio, ci limiteremo a considerare processi finiti, cioè costruiti senza far ricorso a definizioni ricorsive di processo. Una gestione (limitata) di processi ricorsivi è possibile e rimandiamo il lettore interessato a [6] per ulteriori approfondimenti.

5.1.1 Il sistema di inferenza

Partiamo col fornire gli assiomi e le regole di inferenza.

Assiomi per la Somma:

$$\vdash M + \mathbf{0} = M$$
$$\vdash M_1 + M_2 = M_2 + M_1$$
$$\vdash M_1 + (M_2 + M_3) = (M_1 + M_2) + M_3$$
$$\vdash M + M = M$$

Assioma per il Parallelo:

$$\begin{aligned} \vdash \sum_i \alpha_i.P_i \mid \sum_j \beta_j.Q_j \quad = \quad & \sum_i \alpha_i(P_i \mid \sum_j \beta_j.Q_j) \quad + \\ & \sum_j \beta_j(\sum_i \alpha_i.P_i \mid Q_j) \quad + \\ & \sum_{\alpha_i = \overline{\beta_j}} \tau(P_i \mid Q_j) \end{aligned}$$

Assiomi per la Restrizione:

$$\begin{aligned} &\vdash \mathbf{0}\backslash a = \mathbf{0} \\ &\vdash (\sum_i \alpha_i.P_i)\backslash a = \sum_i (\alpha_i.P_i)\backslash a \\ &\vdash (\alpha.P)\backslash a = \begin{cases} \mathbf{0} & \text{se } \alpha \in \{a, \bar{a}\} \\ \alpha.(P\backslash a) & \text{altrimenti} \end{cases} \end{aligned}$$

Regole di Inferenza:

$$\vdash P = P$$

$$\frac{\vdash P = Q}{\vdash Q = P}$$

$$\frac{\vdash P = Q \quad \vdash Q = R}{\vdash P = R}$$

$$\frac{\vdash P = Q}{\vdash C[P] = C[Q]}$$

5.1.2 Correttezza e completezza (per processi finiti)

Il seguente teorema afferma che due processi sono bisimili se e soltanto se la loro uguaglianza è dimostrabile nel sistema di inferenza.

Teorema 5.1. *Siano P e Q processi finiti. $P \sim Q$ se e solo se $\vdash P = Q$*

Dimostriamo ora le due implicazioni del teorema isolatamente.

Teorema (Correttezza): Se $\vdash P = Q$ allora $P \sim Q$.

Dimostrazione.

- per ogni assioma $\vdash LHS = RHS$, consideriamo $\{(LHS, RHS)\} \cup Id$ e dimostriamo che è una bisimulazione;
- le regole di inferenza valgono per la bisimulazione poichè essa è una equivalenza e una congruenza.

□

Per dimostrare l'implicazione inversa, abbiamo bisogno di alcune nozioni tecniche e risultati preliminari.

Definizione 5.1.1.

P *è in* standard form *se e solo se* $P \triangleq \sum_i \alpha_i.P_i$ *e* $\forall_i\ P_i$ *è in standard form.*

Lemma 5.2. $\forall P\ \exists\ P'$ *in standard form tale che* $\vdash P = P'$

Dimostrazione. Per induzione sulla struttura di P.

Caso base:($P \triangleq \mathbf{0}$). Basta prendere $P' \triangleq \mathbf{0}$ e concludere per riflessività

Passo induttivo: Dobbiamo analizzare tre casi.

1. $P \triangleq \sum_i \alpha_i.P_i$. Per induzione abbiamo che $\forall P_i\ \exists\ P_i'$ in standard form tale che $\vdash P_i = P_i'$. Da $\vdash P_i = P_i'$, per chiusura di contesto, segue che $\vdash \alpha_i P_i = \alpha_i P_i'$, da cui segue che

$$\vdash \underbrace{\sum_i \alpha_i P_i}_{P} = \underbrace{\sum_i \alpha_i P_i'}_{P'}$$

2. $P \triangleq P_1|P_2$. Per induzione abbiamo che $\exists\ P_1', P_2'$ in standard form tali che $\vdash P_1 = P_1'$ e $\vdash P_2 = P_2'$, dove $P_1' = \sum_i \alpha_i.R_i$ e $P_2' = \sum_j \beta_j.Q_j$. Da ciò e per congruenza, segue che

$$\begin{aligned} \vdash \overbrace{P_1|P_2}^{P} &= \sum_i \alpha_i.R_i \mid \sum_j \beta_j.Q_j \\ &= \sum_i \alpha_i(R_i \mid \sum_j \beta_j.Q_j) + \sum_j \beta_j(\sum_i \alpha_i.R_i \mid Q_j) + \sum_{\alpha_i = \overline{\beta_j}} \tau(R_i \mid Q_j) \\ &= \ldots \\ &= P' \end{aligned}$$

   dove il procedimento di eliminazione del parallelo tra forme standard è ripetuto finché non ci sono più occorrenze di '|' nel processo.

3. $P \triangleq Q\backslash a$. Per induzione abbiamo che $\exists\ Q'$ in standard form tale che

$\vdash Q = Q'$, dove $Q' = \sum_{i \in I} \alpha_i.R_i$. Da ciò e per congruenza, segue che

$$\vdash \overbrace{Q\backslash a}^{P} = Q'\backslash a = \sum_{i \in I}(\alpha_i.R_i)\backslash a = \sum_{i \in I'} \alpha_i(R_i\backslash a)$$

con $I' \triangleq \{i \in I : \alpha_i \notin \{a, \bar{a}\}\}$ e il procedimento di eliminazione della restrizione è ripetuto fino ad eliminare tale operatore. □

Teorema (Completezza): Se $P \sim Q$ allora $\vdash P = Q$.

Dimostrazione. Per il lemma appena dimostrato, abbiamo che $\exists\ P', Q'$ in standard form tali che $\vdash Q = Q'$ e $\vdash P = P'$, dove

$$P' \triangleq \sum_{i=1}^{n} \alpha_i.P_i \quad \text{e} \quad Q' \triangleq \sum_{j=1}^{m} \beta_j.Q_j$$

Basta quindi dimostrare che $\vdash P' = Q'$ e per transitività avremo che $\vdash P = Q$. Questa prova è fatta per induzione sulla massima altezza dell'albero sintattico che descrive P' e Q', cioè $max\{h(P'), h(Q')\}$.

Caso base (0): in questo caso $P' = Q' = \mathbf{0}$ e si conclude banalmente.
Induzione:

$$\begin{aligned} P' \xrightarrow{\alpha_1} P_1 &\Rightarrow P \xrightarrow{\alpha_1} \hat{P} \text{ tale che } P_1 \sim \hat{P} \\ &\Rightarrow Q \xrightarrow{\alpha_1} \hat{Q} \text{ tale che } \hat{P} \sim \hat{Q} \\ &\Rightarrow Q' \xrightarrow{\alpha_1} Q'' \text{ tale che } \hat{Q} \sim Q'' \end{aligned}$$

dove la prima e la terza implicazione si hanno grazie al lemma 5.2 e al teorema di correttezza, mentre la seconda implicazione vale per ipotesi. Inoltre, per definizione del processo Q', deve essere che $\alpha_1 = \beta_{j_1}$ e $Q'' = Q_{j_1}$, per un qualche j_1.
Per transitività otteniamo che $P_1 \sim Q_{j_1}$ da cui, per induzione, segue che $\vdash P_1 = Q_{j_1}$. Consideriamo adesso come contesto

$$C \triangleq \alpha_1[] + \sum_{i=2}^{n} \alpha_i.P_i$$

Allora

$$\vdash \overbrace{\alpha_1.P_1 + \sum_{i=2}^{n} \alpha_i.P_i}^{P'} = \beta_{j_1}.Q_{j_1} + \sum_{i=2}^{n} \alpha_i.P_i$$

Iterando questo ragionamento su ogni addendo di P' si può concludere che

$$\begin{aligned}\vdash P' &= \beta_{j_1}.Q_{j_1} + \sum_{i=2}^{n}\alpha_i.P_i \\ &= \beta_{j_1}.Q_{j_1} + \beta_{j_2}.Q_{j_2} + \sum_{i=3}^{n}\alpha_i.P_i \\ &= \beta_{j_1}.Q_{j_1} + \beta_{j_2}.Q_{j_2} + \beta_{j_3}.Q_{j_3} + \sum_{i=4}^{n}\alpha_i.P_i \\ &= \dots \\ &= \sum_{i=1}^{n}\beta_{j_i}.Q_{j_i}\end{aligned}$$

Con passaggi analoghi si dimostra che

$$\vdash Q' = \sum_{j=1}^{m}\alpha_{i_j}.P_{i_j}$$

Sommando infine membro a membro otteniamo

$$\vdash P' + \sum_{j=1}^{m}\alpha_{i_j}.P_{i_j} = Q' + \sum_{i=1}^{n}\beta_{j_i}.Q_{j_i}$$

che, per idempotenza, implica $\vdash P' = Q'$. □

5.1.3 Bisimulazione debole ed esempio d'uso

Abbiamo trattato finora la bisimulazione forte. Per assiomatizzare la bisimulazione debole, si aggiungono i seguenti tre assiomi:

$$\begin{aligned}&\vdash \alpha.P = \alpha.\tau.P \\ &\vdash P + \tau.P = P \\ &\vdash \alpha.(P + \tau.Q) = \alpha.(P + \tau.Q) + \alpha.Q\end{aligned}$$

A questo punto, è possibile dimostrare un teorema di correttezza e completezza analogo a quello appena visto per la bisimulazione forte; i dettagli tecnici sono simili e si rimanda il lettore interessato a [11].

Per concludere la presentazione di questo approccio, mostriamo come il sistema di inferenza possa essere usato per stabilire la bisimilarità di due semplici processi. Consideriamo, come esempio, un server per lo scambio di messaggi. Tale server, nella sua versione minimale, riceve una richiesta di invio di messaggi e restituisce la conferma dell'avvenuta ricezione; pertanto, la sua specifica è:

$$Spec \triangleq send\ .\ \overline{rcv}$$

Il comportamento di tale server può essere implementato da tre processi in parallelo: uno gestisce il pulsante *send*, uno spedisce effettivamente il messaggio

(tramite l'azione ristretta *put*) e aspetta il segnale di avvenuta ricezione del messaggio (tramite il segnale *go*) e l'ultimo uno fornisce all'utente l'esito della spedizione.

$$\left.\begin{array}{lcl} S & \stackrel{\triangle}{=} & send\,.\,\overline{put} \\ M & \stackrel{\triangle}{=} & put\,.\,\overline{go} \\ R & \stackrel{\triangle}{=} & go\,.\,\overline{rcv} \end{array}\right\} Impl \stackrel{\triangle}{=} (S|M|R)\backslash\{put, go\}$$

Vogliamo ora mostrare che la specifica è equivalente (cioè, debolmente bisimile) all'implementazione. Consideriamo il parallelo dei processi M e R; per l'assioma del parallelo si ottiene che

$$\vdash M|R = put.(\overline{go}|R) + go.(M|\overline{rcv})$$

Applicando lo stesso assioma al parallelo dei tre processi otteniamo che

$$\vdash S|(M|R) = send.(\overline{put}|(M|R)) + put.(\overline{go}|R|S) + go.(\overline{rcv}|S|M)$$

Restringendo su *put* e *go* e applicando il secondo assioma della restrizione, otteniamo che

$$\begin{array}{rcll} \vdash Impl & = & (send.(\overline{put}|M|R))\backslash\{put, go\} & + \\ & & (put.(\overline{go}|R|S))\backslash\{put, go\} & + \\ & & (go.(\overline{rcv}|S|M))\backslash\{put, go\} & \end{array}$$

Applichiamo adesso il terzo assioma della restrizione ai tre addendi:

- $(send.(\overline{put}|M|R))\backslash\{put, go\} = send.(\overline{put}|(M|R))\backslash\{put, go\}$, in quanto $send \notin \{put, \overline{put}, go, \overline{go}\}$;
- $(put.(\overline{go}|R|S))\backslash\{put, go\} = \mathbf{0}$;
- $(go.(\overline{rcv}|S|M))\backslash\{put, go\} = \mathbf{0}$.

da cui $\vdash Impl = send.(\overline{put}|(M|R))\backslash\{put, go\}$.
Procediamo in maniera simile su $(\overline{put}|M|R)\backslash\{put, go\}$; dopo la sincronizzazione su *put*, otteniamo che

$$\vdash Impl = send.\tau.(\overline{go}|R)\backslash\{put, go\}$$

Grazie al primo assioma per la bisimulazione debole otteniamo che

$$\vdash Impl = send.(\overline{go}|R)\backslash\{put, go\}$$

Di nuovo i processi si sincronizzano, questa volta sulla variabile *go*:

$$\vdash Impl = send.\tau.(\overline{rcv})\backslash\{put, go\}$$

che, come prima, porta a

$$\vdash Impl = send.(\overline{rcv}.\mathbf{0})\backslash\{put, go\}$$

A questo punto basta applicare il terzo assioma della restrizione per avere che

$$\vdash Impl = send.\overline{rcv}.\mathbf{0}\backslash\{put, go\}$$

Infine, per il primo assioma della restrizione, otteniamo

$$\vdash Impl = send.\overline{rcv}.\mathbf{0}$$

cioè, $\vdash Impl = Spec$.

5.2 Approccio logico

Come abbiamo visto, un processo può essere descritto dall' insieme delle sequenze di azioni che *sa fare*: questa caretterizzazione riflette la visione classica della teoria degli automi in cui una macchina è identificata dal'insieme di stringhe da lei accettate (nel nostro caso la stringa ha come alfabeto l'insieme delle azioni Action). Due processi che *sanno fare* le stesse sequenze non sono però necessariamente bisimili :

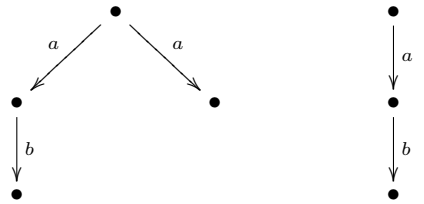


Figura 5.1: Due LTS non bisimili

Possiamo, in un modo più esaustivo, indentificare un processo con l'insieme delle **proprietà** di cui esso gode, dove una proprietà, sempre rimanendo su un piano informale, è una sentenza su quello che *sa* e *non sa* fare il processo. Per esempio , in figura 5.1 abbiamo che:

il processo p_1 sa fare l'azione ***a*** *e poi non può fare l'azione* ***b***

al contrario

il processo p_2 dopo aver fatto l'azione ***a*** *ha sempre la possibilità di fare l'azione* ***b***[1].

[1] da notare che le due *proprietà* sono una la negata dell'altra

Definiremo ora una sintassi per poter esprimere queste proprietà in modo formale, poi daremo una relazione di soddisfacibilità tra processi e proprietà e infine dimostreremo che godere delle stesse proprietà è condizione necessaria e sufficente per la bisimulazione.

5.2.1 Sintassi

$$\varphi \ := \ \mathtt{TT} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \ \diamond a\varphi \quad \textit{dove } a \in \mathsf{Action}$$

Il linguaggio generato dalla grammatica verrà indicato con Form, l'insieme delle formule, ogni elemento dell'insieme verrà chiamato **formula**

5.2.2 Relazione di Soddisfacibilità

Sia $\vDash \subseteq \ \mathsf{Proc} \times \mathsf{Form}$ una relazione binaria tra processi e formule definita. Diciamo che il processo p **soddisfa** la formula f e usiamo la notazione $p \vDash f$ se solo se $(p, f) \in \vDash$.
Sia $p \in \mathsf{Proc}$ un generico processo,**a** $\in \mathsf{Action}$ una generica azione. La relazione $\vDash$ può essere definita induttivamente nel modo seguente :

$$p \vDash \mathtt{TT} \tag{5.2.1}$$

$$p \vDash \neg\varphi \ \textit{sse} \ p \nvDash \varphi \tag{5.2.2}$$

$$p \vDash \varphi_1 \wedge \varphi_2 \ \textit{sse} \ p \vDash \varphi_1 \ \wedge \ p \vDash \varphi_2 \tag{5.2.3}$$

$$p \vDash \diamond a\varphi \ \textit{sse} \ \exists p' : p \xrightarrow{a} p' \ \wedge \ p' \vDash \varphi \tag{5.2.4}$$

Ovviamente, possono essere usati il falso (indicandolo con FF) e gli operatori classici della logica come la disgiunzione $\vee$ (può essere derivato dalla congiunzione e la negazione tramite legge di De Morgan) l'implicazione $\Rightarrow$. Un altro operatore logico molto utile è il 'box': definiamo $\neg \diamond a \triangleq \Box a \neg\varphi$. Usando 5.2.3 e 5.2.4 otteniamo che

$$p \vDash \Box a\varphi \ \textit{sse} \ p \vDash \neg \diamond a \neg\varphi \ \ \textit{sse} \ \ p \nvDash \diamond a \neg\varphi \ \ \textit{sse}$$
$$\nexists p' : p \xrightarrow{a} p' \ \wedge \ p' \vDash \neg\varphi \ \textit{sse} \ \forall p' : p \overset{a}{\nrightarrow} p' \ \vee \ p' \vDash \varphi$$

Dall'ultima affermazione abbiamo che $p \vDash \Box a\varphi$ se e solo se

$$\forall p' : p \xrightarrow{a} p' \Rightarrow p' \vDash \varphi$$

Si consideri la formula $\Box a\mathtt{FF}$. Quand'è che un processo può soddisfare tale formula? Per definizione, questo si ha solo se per ogni processo p' risultante dall'esecuzione da parte di p dell'azione **a** vale $p' \vDash \mathtt{FF}$ ma, dall'assioma (5.2.1) e dalla regola (5.2.2) si ha che ciò non può valere, qualunque sia p' e quindi $p \vDash \Box a\mathtt{FF}$ è vera se solo se il processo p *non sa fare* l'azione **a**.

Tornando all'esempio di figura 5.1 possiamo ora, tramite questo linguaggio formale, scrivere la proprietà :

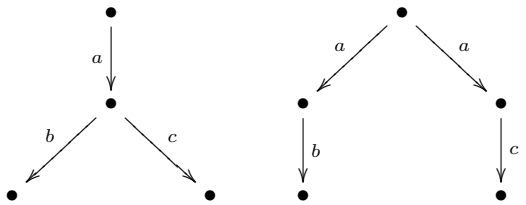
$$f \triangleq \diamond a \Box b\mathtt{FF}$$

Abbiamo che $p_2 \vDash f$ mentre $p_1 \nvDash f$. Similmente, consideriamo gli LTS risultanti dalle seguenti definizioni di processo:

$$C_1 \triangleq \mathbf{a}.C_1$$

$$C_2 \triangleq \mathbf{a}.C_2 + \mathbf{a}$$

Anche questi due processi non sono bisimili e anche in questo caso esiste una formula soddisfatta da un processo e non dall'altro. $\diamond a \Box a\mathtt{FF}$ ovvero *esiste una transizione etichettata con* $\boldsymbol{a}$ *che mi permette di non fare più l'azione* $\boldsymbol{a}$.
Consideriamo infine Ora, possiamo usare due formule per distinguere i due



p_1 p_2

processi (non bisimili);

$\Box a \diamond b\mathtt{TT}$ è soddisfatta da p_1 ma non da p_2
$\diamond a(\diamond b\mathtt{TT} \wedge \diamond c\mathtt{TT})$ soddisfatta da p_1 ma non da p_2

Definiamo ora l'insieme delle formule soddisfatte da un processo come

$$L(p) = \{f \in \mathsf{Form} \text{ t.c } p \vDash f\}$$

Andiamo ora a dimostrare che due processi sono bisimili se e solo se soddisfano le stesse formule logiche. Per semplificare la prova assumeremo di avere congiunzioni su un insieme numerabile di formule

Teorema 1.

$$p \sim q \textit{ se solo se } L(p) = L(q)$$

abbiamo, quindi, un potente strumento per poter dimostrare quando due processi *non* sono bisimili: infatti basterà trovare una formula soddisfatta dall'uno ma non dall'altro. In termini di teoria della complessità tale formula f soddisfatta dal processo p sse non soddisfatta da processo q è un certificato per il linguaggio $\{(p,q) \text{ t.c. } p \not\sim q \ \wedge p, q \in Proc\}$.

Dimostrazione. ($\Rightarrow$) Per induzione sulla struttura dell'albero sintattico delle formule dimostriamo che:

$$\varphi \in L(p) \text{ sse } \varphi \in L(q)$$

BASE $\varphi \triangleq \texttt{TT}$ per l'assioma (5.2.1) questo è vero per ogni processo quindi anche per p e q.
PASSO Supponiamo che per ogni formula con albero sintattico di altezza al più h l'affermazione sia valida. Sia $h+1$ l'altezza dell'albero sintattico di φ. Distinguiamo l'operatore più esterno in φ.

1. $\varphi \triangleq \bigwedge_{i \in I} \varphi_i$
   Supponiamo che $\varphi \in L(p)$ si ha che

   $$\varphi \in L(p) \iff p \vDash \varphi \iff \forall i \in I \; p \vDash \varphi_i \iff \forall i \in I \; \varphi_i \in L(p)$$

   per definizione ogni formula φ_i ha un albero sintattico con altezza minore uguale a h quindi per loro vale l'ipotesi induttiva $\varphi_i \in L(p) \iff \varphi_i \in L(q)$.Da ciò,

   $$\forall i \in I \; \varphi_i \in L(q) \iff \forall i \in I \; q \vDash \varphi_i \iff \bigwedge_{i \in I} \varphi_i \in L(q) \iff \varphi \in L(q)$$

2. $\varphi \triangleq \neg\varphi'$

   $$\varphi \in L(p) \iff \neg\varphi' \in L(p) \iff \varphi' \notin L(p)$$

   L'altezza dell'albero sintattico di φ' è h e quindi possiamo applicare l'ipotesi induttiva

   $$\varphi' \notin L(p) \iff \varphi' \notin L(q) \iff \neg\varphi' \in L(q)$$

3. $\varphi \triangleq \Diamond a\varphi'$

   $$\varphi \in L(p) \iff \exists p' : p \xrightarrow{a} p' \; \wedge \; p' \vDash \varphi'$$
   $$p' \vDash L(p') \iff \varphi' \in L(p')$$

   Per ipotesi abbiamo che $p \sim q$ e quindi $\exists q' : q \xrightarrow{a} q' \; \wedge \; p' \sim q'$. Dato che l'altezza dell'albero sintattico di φ' è strettamente minore rispetto a quella di φ, abbiamo che, per ipotesi induttiva, vale $\varphi' \in L(q')$[2] è per definizione del diamond abbiamo $q' \vDash \varphi'$ da cui $q \vDash \Diamond a\varphi' = \varphi$.
   Fino ad ora abbiamo dimostrato che $\varphi \in L(p) \Rightarrow \varphi \in L(q)$. Per compleatare la dimostrazione bisognerebbe dimostrare l'implicazione inversa, ma analizzando bene la dimostrazione si nota che basta invertire p e q per ottenere il se solo se.

[2] In questo punto della dimostrazione l'implicazione si applica la definizione di bisimulazione e non è un se solo se

($\Leftarrow$) Dimostreremo che $R \triangleq \{(p,q) : L(p) = L(q)\}$ è una simulazione; data la simmetria della relazione di ugualianza tra insiemi se R è una simulazione allora è una bisimulazione.
Sia $(p,q) \in R \ \wedge \ p \xrightarrow{a} p'$.Consideriamo

$$\varphi \triangleq \diamond a\varphi' \text{ dove } \varphi' \triangleq \bigwedge_{\varphi'' \in L(p')} \varphi''$$

Per costruzione $p \vDash \varphi$, quindi $q \vDash \varphi$ questo perchè $L(q) = L(p)$; allora $\exists q' : q \xrightarrow{a} q' \ \wedge \ q' \vDash \varphi'$.
Ma allora $\forall \varphi'' \in L(p)$,$\varphi'' \in L(q)$. Quindi $L(p') \subseteq L(q')$.
Supponiamo che l'inclusione sia propria: $L(p') \subset L(q')$. Quindi $\exists \hat{\varphi} : \ \hat{\varphi} \in L(q') \ \wedge \ \hat{\varphi} \notin L(p')$
Allora $\neg\hat{\varphi} \in L(p')$ il che implicherebbe $\neg\hat{\varphi} \in L(q')$ la cosa sarebbe assurda poichè $L(q')$ non può contenere una formula e la sua negata, tale assurdo viene da aver supposto che l'inclusione sia propria. □

5.2.3 Sotto-logiche

Proviamo a considerare linguaggi di formule prive di alcuni operatori.

Una logica senza negazione

$$\varphi \ := \ \mathtt{TT} \mid \varphi \wedge \varphi \mid \ \diamond a\varphi \ \ \textit{dove } a \in \mathsf{Action}$$

Questa sintassi è priva della negazione: la formula $\Box a\mathtt{FF}$ non è nel linguaggio. Quindi possiamo esprimere sentenze solo su quello che il processo *sa fare*.
Chiamiamo $L_\neg(q)$ l'insieme delle formule, con la sintassi ridotta, soddisfatte dal processo q.

Teorema 2.

$$p \textit{ simula } q \iff L_\neg(q) \subseteq L_\neg(p)$$

Dimostrazione. ($\Rightarrow$) Sia $\varphi \in L_\neg(q)$. Per induzione sulla altezza dell'albero sintattico dimostreremo $\varphi \in L_\neg(p)$.
BASE TT appartiene ad ogni processo.
PASSO

1. $\varphi \triangleq \wedge_i \varphi_i$. Simile al caso 1 del teorema 1.
2. $\varphi \triangleq \diamond a\varphi'$. Simile al caso 3 del teorema 1 ma qui non si va a dimostrare che l'inclusione non può essere propria.

□

Come Corollario segue che c'è una doppia simulazione tra p e q se e solo se $L_\neg(p) = L_\neg(q)$.
Tornando all'esempio di figura 5.1 si ha che $L_\neg(p_2) = L_\neg(p_1)$

Una logica senza negazione e congiunzione

$$\varphi \ := \ \mathtt{TT} \mid \ \diamond a\varphi \ \ \textit{dove } a \in \mathsf{Action}$$

Come prima chiamiamo $L_{\neg,\wedge}(p)$ l'insieme delle formule della sottologica soddisfatta dal processo p.
Chiamiamo con $\mathsf{Ling}(p)$ le stringe **accettate** dall'automa isomorfo al LTS p in cui ogni stato è **accettante**.

Teorema 3.

$$L_{\wedge,\neg}(p) = L_{\wedge,\neg}(q) \iff \mathit{Ling}(q) = \mathit{Ling}(p)$$

Dimostrazione. Sia $\varphi = \diamond a_1, \ldots, \diamond a_n \mathtt{TT} \in L_{\wedge,\neg}(p)$. Per definizione quesi si ha se e solo se $\exists p_1, \ldots, p_n$ t.c. $p \xrightarrow{a_1} p_1 \ldots \xrightarrow{a_n} p_n$ e, quindi, se e solo se $a_1, \ldots, a_n \in \mathsf{Ling}(p)$. □

5.3 Approccio algoritmico

La bisimulazione è già stata analizzata tramite un modello assiomatico ed un modello logico. Mostreremo che il problema di decidere se due stati sono bisimili può essere affrontato anche tramite l'utilizzo di algoritmi supponendo, ovviamente, che l'insieme degli stati sia finito.
Consideriamo un LTS generico definito dalla tripla $(S, \Lambda, \rightarrow)$ dove

- S è un insieme finito di stati.
- Λ è un insieme finito di etichette.
- $\rightarrow \subseteq S \times \Lambda \times S$ è la relazione ternaria che utilizziamo per descrivere le transizioni di stato. La notazione $p \xrightarrow{\alpha} q$, con $p, q \in S$ e $\alpha \in \Lambda$ equivale alla tripla $(p, \alpha, q) \in \rightarrow$.

Sappiamo che due stati $p, q \in S$ sono bisimili se esiste una relazione binaria $R \subseteq S \times S$ tale che sia R che R^{-1} sono simulazioni, e la coppia (p, q) appartiene a R. Tuttavia ottenere e verificare l'insieme R può essere un lavoro oneroso; il nostro obiettivo è trovare un algoritmo in grado di automatizzare questo processo in maniera efficiente.

5.3.1 Un primo tentativo

Consideriamo il seguente procedimento:

1. Trasforma LTS in un automa *non deterministico* a stati finiti, dove ogni stato può assumere il ruolo di stato finale.
2. Trasforma l'automa ottenuto in un automa deterministico[3].

[3] Powerset Construction Algorithm [9], cap. 2

3. Applica all'automa ottenuto l'algoritmo di minimizzazione[4].

4. OUTPUT: Due stati sono bisimili se e solo se sono nello stesso stato dell'automa minimo.

Notiamo che ogni singolo passo effettuato termina, quindi il nostro procedimento è un algoritmo. Inoltre è banalmente corretto, infatti se i due stati del LTS si trovano nello stesso stato dell'automa minimo, allora possono simulare a vicenda tutte le possibili transizioni. Possiamo quindi affermare che esiste almeno un algoritmo che risolve il problema della bisimilarità. Ma quanto è efficiente?
In realtà questa soluzione non è affatto applicabile in un contesto reale: infatti l'algoritmo di Powerset Construction ha complessità esponenziale rispetto al numero degli stati dell'automa in input. Fortunatamente è disponibile un algoritmo che risolve il problema in tempo polinomiale.

5.3.2 Un algoritmo efficiente

Il problema della bisimulazione può essere affrontato utilizzando un algoritmo che risolve un problema presente in molteplici ambiti: il *Relational Coarsest Partition Problem* (RCPP). Il RCPP prevede il raffinamento in classi (o blocchi) di un insieme partendo dalla sua partizione più grossolana (che è l'insieme di partenza stesso), seguendo una regola specifica che indica come separare gli elementi nelle relative classi. L'algoritmo che presentiamo è stato proposto nel 1983 da Kanellakis e Smolka [10] ed opera in tempo polinomiale nel numero degli stati del LTS. Possiamo ricapitolare l'algoritmo nei seguenti passi:

1. Parti dalla partizione più grossolana possibile (l'insieme stesso)

2. Finchè è possibile, dividi la partizione che non contiene elementi equivalenti utilizzando la regola di separazione.

3. OUTPUT: Due stati sono bisimili se, al termine dell'algoritmo, si trovano nello stesso blocco.

La regola di separazione: Prima di scrivere l'algoritmo nei dettagli è necessario definire la regola che ci permette di separare due stati in due blocchi differenti.
Sia B un blocco generico della partizione, e siano s e t due stati contenuti in B. Indichiamo con blocco(s) tutti gli stati che si trovano nello stesso blocco di s. Le nostre premesse impongono che i due stati s e t vengano separati (e quindi il blocco B va partizionato in due blocchi distinti) se e solo se:

$$\exists\, \alpha, s' : s \xrightarrow{\alpha} s' \quad \text{e} \quad \forall\, t' \quad \text{si ha che} \quad t \xrightarrow{\alpha} t' \vee t' \notin \text{blocco}(s') \tag{5.2.5}$$

Questa condizione dice che, affinchè s e t debbano essere separati, deve esistere una transizione $s \xrightarrow{\alpha} s'$ tale che per ogni transizione di t, la transizione $t \xrightarrow{\alpha} t'$

[4] Brzozowski's Minimization Algorithm, [4]

non può esistere oppure (se esiste) t' non appartiene all'insieme degli stati del blocco di s' (ovvero quello che raggiunge s). In altre parole s e t non devono raggiungere con la medesima transizione $\xrightarrow{\alpha}$ due stati destinazione che si trovano nello stesso blocco.
Per implementare questo concetto nell'algoritmo utilizziamo la seguente notazione che è del tutto equivalente alla regola di separazione appena descritta. Definiamo il seguente insieme:

$$tgt_\alpha(s) \triangleq \{B' \quad \text{tale che} \quad \exists\, s' : s \xrightarrow{\alpha} s' \wedge s' \in B'\}$$

L'insieme $tgt_\alpha(s)$ è quindi l'insieme dei blocchi che contengono gli stati raggiungibili da s effettuando la transizione α. E' banale osservare che la (5.2.5) può essere riscritta utilizzando la notazione introdotta come:

$$\exists\, \alpha : tgt_\alpha(s) \neq tgt_\alpha(t) \tag{5.2.6}$$

L'algoritmo: Riportiamo l'algoritmo completo di Kanellakis-Smolka in pseudocodice.

Algorithm 1 RCPP

```
1:  INPUT: LTS = (S, Λ, →), s ∈ S, t ∈ S
2:  P ← {S}
3:  for all B ∈ P do
4:     for all α ∈ Λ do
5:        I = split(B, α, P)
6:        if I ≠ {B} then
7:           P = (P \ {B}) ∪ I
8:           goto 3
9:        end if
10:    end for
11: end for
12: for all B ∈ P do
13:    if {s, t} ⊆ B then
14:       return true
15:    end if
16: end for
17: return false
```

Algorithm 2 split (funzione)

```
1: INPUT: (B, α, P)
2: s ∈ B
3: B1 = B2 = ∅
4: for all t ∈ B do
5:    if tgt_α(s) = tgt_α(t) then
6:       B1 = B1 ∪ {t}
7:    else
8:       B2 = B2 ∪ {t}
9:    end if
10: end for
11: if B2 = ∅ then
12:    return {B}
13: else
14:    return {B1, B2}
15: end if
```

5.3.3 Correttezza

Si osserva facilmente che l'algoritmo termina. Infatti ogni ciclo opera al più su un numero finito di elementi (facciamo notare che la cardinalità di $\mathcal{P}$ può crescere fino a $|S|$ elementi ed in quel caso il comando a linea 6 dell'algoritmo RCPP è sempre falso permettendo la conclusione dell'algoritmo). Resta da dimostrare che due stati sono bisimili se e solo se appartengono allo stesso blocco, ovvero definendo $\mathcal{P}_f$ l'insieme $\mathcal{P}$ a fine algoritmo

$$s \sim t \iff \exists\ B \in \mathcal{P}_f : \{s,t\} \subseteq B$$

Dimostrazione: ($\Leftarrow$) Vogliamo dimostrare che se $\exists\ B \in \mathcal{P}_f : \{s,t\} \subseteq B$ allora $s \sim t$.
Consideriamo la seguente relazione R così definita

$$R = \{(x,y) : \exists\ B \in \mathcal{P}_f, (x,y) \in B \times B\}$$

In altre parole R contiene tutte le coppie di stati che sono presenti in uno stesso blocco in $\mathcal{P}_f$[5]. Ora se dimostriamo che R è una bisimulazione abbiamo terminato perchè $\{s,t\}$ è presente in R per definizione.
Osserviamo il comportamento di una generica transizione di stato $x \xrightarrow{\alpha} x' : (x,y) \in R$. Visto che il blocco che contiene x non è più raffinabile in quanto l'algoritmo è terminato, sappiamo che $tgt_\alpha(x) = tgt_\alpha(y)$, che implica l'esistenza di uno stato y' tale che $y \xrightarrow{\alpha} y'$ ed inoltre blocco(x') = blocco(y'), ovvero x' e y' appartengono ad uno stesso blocco contenuto in $\mathcal{P}_f$. Ma per come abbiamo definito R la coppia $(x',y') \in R$. Abbiamo appena mostrato che

[5] Per fare un esempio possiamo rappresentare gli stati attraverso i naturali, quindi (per un LTS con solo quattro stati) $S = \{1,2,3,4\}$. Supponiamo ora che $\mathcal{P}_f = \{\{1,3\},\{2,4\}\}$, allora avremo $R = \{(1,3),(3,1),(2,4),(4,2),(1,1),(2,2),(3,3),(4,4)\}$.

R è una simulazione valida per ogni coppia che contiene, tuttavia $R = R^{-1}$ per come è stata definita e quindi è anche una bisimulazione.

($\Rightarrow$) Vogliamo dimostare che se $s \sim t$ allora $\exists\ B \in \mathcal{P}_f : \{s, t\} \subseteq B$. Supponiamo che s bisimula t ma i due stati appartengono a due blocchi differenti, ovvero blocco(s) $\neq$ blocco(t). All'inizio dell'algoritmo, $\mathcal{P}$ contiene in blocco con tutti gli stati, quindi s e t sono stati separati in un passo dell'algoritmo che assumeremo essere l'$(i+1)$-esimo. Quindi al passo i-esimo, esisteva una partizione $\mathcal{P}_i = \{B_1, B_2, \ldots, B_k\}$ tale che $\exists\ j : 1 \leq j \leq k$ e gli elementi s, t appartenevano al blocco B_j. Ed al passo $(i+1)$-esimo veniva eseguita la seguente separazione: $\mathcal{P}_{i+1} = \{B_1, B_2, \ldots, B'_j, B''_j, \ldots, B_k\}$ con $s \in B'_j$ e $t \in B''_j$. Ma se s e t sono stati separati allora esiste una transizione α tale che $tgt_\alpha(s) \neq tgt_\alpha(t)$, che per definizione vuol dire che o

$$\exists\ s' : s \xrightarrow{\alpha} s'\ \wedge\ \forall\ t'\ ,\ t \overset{\alpha}{\not\to} t' \vee \text{blocco}(s') \neq \text{blocco}(t')$$

oppure

$$\exists\ t' : t \xrightarrow{\alpha} t'\ \wedge\ \forall\ s'\ ,\ s \overset{\alpha}{\not\to} s' \vee \text{blocco}(t') \neq \text{blocco}(s')$$

Ma questo è *assurdo* perchè s e t sono stati assunti bisimili.

5.3.4 Complessità

Per analizzare il costo complessivo dell'algoritmo proposto è necessario procedere per passi. Assumiamo un LTS generico $(S, \Lambda, \rightarrow)$ con $|S| = n$ e $|\ \rightarrow\ | = m$. Iniziamo con il ciclo più esterno che è dominato dall'istruzione *goto* (linea 8): questa viene eseguita ogni volta si verifica un partizionamento, ovvero l'insieme mandato in input alla funzione split viene diviso in due nuovi insiemi ognuno dei quali contiene almeno uno stato. Questa operazione può essere ripetuta al più $O(n)$ volte perchè, partendo dall'insieme più generale che contiene tutti gli stati, nel caso peggiore è possibile partizionare uno stato alla volta in un nuovo blocco fino a creare n blocchi ognuno con un singolo stato. Il confronto $I \neq B$ (linea 6), può essere in generale effettuato con un numero di operazioni costante (in particolare utilizzando una lista per mantenere le informazioni sugli insiemi) e costa quindi $O(1)$. Il ciclo che emette il risultato (dalla riga 12 alla riga 16) ha costo al più $O(n)$. Restano da analizzare i due cicli *for* annidati ed il costo della funzione split. Osservando il codice possiamo ricapitolare in questo modo le operazioni svolte in questa parte di programma:

- Un ciclo for per tutti i blocchi di $B \in \mathcal{P}$ - [riga 3]
- Un ciclo for per tutte le etichette $\alpha \in \Lambda$ - [riga 4]
- Un ciclo for per tutti gli stati t contenuti nel blocco scelto B - [riga 4 funzione split]
- Il controllo $tgt_\alpha(s) = tgt_\alpha(t)$ eseguito per gli stati s e t scelti - [riga 5 funzione split]

Contiamo le operazioni svolte trasformando questi cicli in una funzione matematica:

$$\sum_{B\in\mathcal{P}}\sum_{\alpha\in\Lambda}\sum_{t\in B}|\{t' : t \xrightarrow{\alpha} t'\}| =$$

Dimostreremo che il controllo $tgt_\alpha(s) = tgt_\alpha(t)$ si può effettuare analizzando ogni α-transizione degli stati del blocco B una sola volta. Dato che gli insiemi B e Λ sono disgiunti posso scambiare posizione alla seconda ed alla terza sommatoria

$$\begin{aligned} &= \sum_{B\in\mathcal{P}}\sum_{t\in B}\sum_{\alpha\in\Lambda}|\{t' : t \xrightarrow{\alpha} t'\}| \quad = \\ &= \sum_{t\in S}\sum_{\alpha\in\Lambda}|\{t' : t \xrightarrow{\alpha} t'\}| \qquad = |\to| = m \end{aligned}$$

La complessità del nostro algoritmo è quindi $O(mn)$, ma quale struttura dati è necessaria affinchè i controlli effettuati nel nostro algoritmo siano efficienti? L'idea è quella di associare ad ogni blocco B (che viene generato durante l'esecuzione dell'algoritmo) una stringa binaria che avrà il compito di *indicizzare* i blocchi presenti attraverso un ordinamento lessicografico. Si parte assegnando al primo blocco (che è S) l'indice binario di un solo bit 0. Quando un blocco subisce una ripartizione in due blocchi separati allora questi prendono come indice la stringa binaria del padre con un bit aggiunto alla fine che vale rispettivamente 0 o 1. Ad esempio consideriamo $S = \{1, 2, 3, 4, 5\}$ e il suo indice è 0. Dopo il primo passo viene partizionato in $(B_1 = \{1, 2, 4\}, B_2 = \{3, 5\})$ e le stringhe binarie associate saranno **0**0 (per B_1) e **0**1 (per B_2). Successivamente B_2 viene diviso in $(B_3 = \{3\}, B_4 = \{5\})$ e le stringhe associate saranno quindi **01**0 (per B_3) e **01**1 per (per B_4) etc...
Questa costruzione soddista un ordinamento lessicografico che sfrutteremo nel seguente modo: trasformiamo ogni terna $(s, \alpha, t) \in\to$, in (s, α, σ) dove σ è l'indice binario del blocco che contiene t. Il risultato finale è una *matrice delle transizioni*:

s	α	σ
s_0	α_0	σ_0^0
s_0	α_0	σ_1^0
$\vdots$	$\vdots$	$\vdots$
s_0	α_0	$\sigma_{p_0}^0$
s_0	α_1	σ_0^1
s_0	α_1	σ_1^1
$\vdots$	$\vdots$	$\vdots$
s_1	α_0	σ_0^0
$\vdots$	$\vdots$	$\vdots$
s_n	$\ldots$	$\ldots$

Questa struttura permette di eseguire dei confronti in maniera molto efficiente e garantisce all'implementazione di rispettare il costo dell'algoritmo calcolato

teoricamente. Infatti se $B = \{s_1, \ldots, s_k\}$ per provare a fare lo *split* di B rispetto ad s_0 e α_0 basta analizzare:

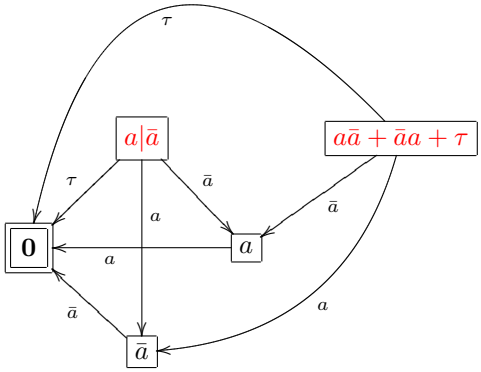
1. la prima transizione di ogni $s_i \in B$: quelle che hanno stessa azione e stesso indice di blocco di s_1 con α_0 vanno in B_1, le altre in B_2.
2. la seconda transizione di ogni $s_i \in B$: quelle che hanno stessa azione e stesso indice di blocco di s_i con α_0 restano in B_1, le altre vanno in B_2.
3. e così via finchè non si esaminano tutte le α_0-transizioni di s_0. A quel punto tutti gli stati di B_1 che hanno altre α_0-transizioni sono spostati in B_2.

5.3.5 Un esempio

Abbiamo presentato un algoritmo efficiente per verificare la bisimulazione tra due stati di un LTS. Vediamone un'applicazione su un caso concreto. Consideriamo i seguenti processi:

- $P_1 \triangleq a|\bar{a}$
- $P_2 \triangleq a\bar{a} + \bar{a}a + \tau$

Vogliamo verificare se gli stati iniziali (evidenziati in rosso) di questi processi sono bisimili. Teoricamente andrebbe costruito un LTS dato dall'unione disgiunta dei due processi, ma in questo caso per risparmiare spazio evitiamo di duplicare stati associati agli stessi processi.



Osservando il diagramma si nota facilmente che i due stati iniziali sono bisimili, ma vogliamo applicare l'algoritmo per verificare formalmente questa intuizione. Per facilitarsi il compito enumeriamo gli stati del nostro LTS in questo modo:

Valore assegnato	Stato
1	$a\|\bar{a}$
2	$a\bar{a} + \bar{a}a + \tau$
3	$\bar{a}$
4	a
5	$\mathbf{0}$

Eseguiamo quindi l'algoritmo in tutti i suoi passi.

1. $\mathcal{P} = \{\{1,2,3,4,5\}\}$, considero l'azione a e scelgo il blocco $B = \{1,2,3,4,5\}$.
2. Eseguo $split(B, a, \mathcal{P})$ e viene effettuata una partizione di B in due blocchi che prendono il suo posto in $\mathcal{P}$.
3. $\mathcal{P} = \{\{1,2,4\},\{3,5\}\}$, considero l'azione $\bar{a}$ e scelgo il blocco $B = \{1,2,4\}$.
4. Eseguo $split(B, a, \mathcal{P})$ e viene effettuata una partizione di B in due blocchi che prendono il suo posto in $\mathcal{P}$.
5. $\mathcal{P} = \{\{1,2\},\{4\},\{3,5\}\}$, considero l'azione a e scelgo il blocco $B = \{3,5\}$.
6. Eseguo $split(B, a, \mathcal{P})$ e viene effettuata una partizione di B in due blocchi che prendono il suo posto in $\mathcal{P}$.
7. $\mathcal{P} = \{\{1,2\},\{4\},\{3\},\{5\}\}$; a questo punto, qualunque blocco e azione si scelga non viene effettuata nessuna divisione. L'algoritmo pertanto termina.
8. Controllo se gli stati 1 e 2 sono contenuti in uno stesso blocco; visto che così è, concludo che i due processi sono bisimili.

Il lettore provi per esercizio a mostrare in maniera analoga che i processi

- $P_1 \triangleq a|\bar{a}$
- $P_2 \triangleq a\bar{a} + \bar{a}a$

non sono bisimili.

Bibliografia

[1] J.A. Bergstra, J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[2] J.C.M. Baeten, W.P. Weijland. Process algebra. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[3] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe. A Thoery of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, 1984.

[4] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, volume 12 of MRI Symposia Series, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, 1962

[5] R. Cleaveland, O. Sokolsky. Equivalence and preorder checking for finite-state systems. Handbook of Process Algebra, chapter 6, pages 391–424. North Holland, 2001.

[6] R. Cleaveland, S.A. Smolka. Process algebra. In *Encyclopedia of Electrical Engineering*, John Wiley 1999.

[7] M.C.B. Hennessy, R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[8] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[9] J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley Publishing, Reading Massachusetts, 1979.

[10] P.C. Kanellakis and S.A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. Proc. ACM Symp. Principles of Distributed Computing, pp. 228-240, 1983.

[11] R. Milner. A Calculus of Communicating Systems. Lecture Notes in Computer Science 92, Springer 1980.

[12] R. Milner. Communicating and Mobile Systems. Cambridge University Press, 1999.