Note del corso di Sistemi Concorrenti e Distribuiti

Sandro Tosi
Yuri Vignoli

Indice

1 Specifiche di Sistemi Concorrenti **3**

2 Algebre di Processi **9**
2.1 Labelled Transition Systems 10
2.2 Azioni interne ed esterne 11
2.3 Operatori 13
2.3.1 Processi Elementari 14
2.3.2 Operatori di Base 15
2.3.3 Operatori di Parallelismo 17
2.3.4 Operatori di Astrazione 19
2.3.5 Altri Operatori 21
2.4 Il Value-Passing 23

3 Il Calcolo CCS **25**
3.1 Struttura Sintattica di CCS 25
3.2 Semantica Operazionale di CCS 27

4 Nozioni preliminari di algebra **32**
4.1 Relazioni, preordini ed equivalenze 32
4.2 Contesti e congruenze 33

5 Equivalenze fra processi **34**
5.1 Equivalenza di traccia 34
5.2 Equivalenza di traccia completa 36
5.3 Equivalenza per fallimenti 36
5.4 Testing equivalence 37
5.5 Equivalenze di bisimulazione 39
5.5.1 Bisimulazione ed equivalenza forte 39
5.5.2 Bisimulazione ed equivalenza debole (osservazionale) . 41
5.6 Simulazione e doppia simulazione 42
5.7 Bisimulazione di branching 42

6 La bisimulazione **44**
6.1 Bisimulazione forte 44
6.1.1 Congruenza forte 50
6.2 Bisimulazione debole 51
6.2.1 Congruenza debole 53
6.3 Assiomatizzazione della bisimulazione 57

7 Testing equivalence **62**
7.1 Teoria di testing 62
7.2 Caratterizzazione alternativa 64
7.2.1 Caratterizzazione alternativa di *may* 65
7.2.2 Caratterizzazione alternativa di *must* 65
7.3 Assiomatizzazione di testing equivalence 66
7.3.1 Assiomi 68
7.3.2 Forme normali per testing 69
7.3.3 Completezza dell'assiomatizzazione 70

Capitolo 1

Specifiche di Sistemi Concorrenti

Concorrenza e Comunicazione

Data la sempre crescente complessità dei sistemi dinamici e distribuiti è diventata di primaria importanza la conoscenza di tecniche e strumenti atti ad analizzare, modellare e verificare se tali sistemi possiedono tutte le caratteristiche che da essi ci aspettiamo. In altre parole assegnare un *significato* a un sistema è esattamente ciò che è osservabile di esso e osservare un sistema è esattamente comunicare con esso.

Detti sistemi sono formati da molte componenti, ognuna delle quali partecipa *insieme* e *indipendentemente* con le altre componenti, ma mantenendo nello stesso tempo la propria identità. Tali componenti sono chiamate *agenti* e compiono attività computazionali indipendenti tra loro e che possono sovrapporsi nel tempo; quindi si dice che le loro esecuzioni procedono in *concorrenza*.

La singola esecuzione di un programma viene definita *processo*, perciò i sistemi concorrenti sono composti da processi multipli che agiscono contemporaneamente; se oltre a ciò gli *agenti* interagiscono tra di loro possiamo parlare di *comunicazione*.

I protocolli e i servizi di network rivestono un ruolo di fondamentale importantanza nell'informatica moderna e rappresentano i più comuni esempi di sistemi concorrenti.

Come si può intuire i sistemi concorrenti sono più complessi e articolati rispetto a quelli sequenziali:

- Un programma sequenziale procede su una sola linea temporale, mentre le diverse componenti in concorrenza tra loro si trovano in stati indipendenti.

- In caso di parallelismo tra processi, il numero di stati cresce esponenzialmente rispetto alle componenti (Nel caso sequenziale la crescita è lineare mentre in questo ambito è necessario conoscere quali combinazioni dei vari stati delle componenti possiamo avere nel sistema).

La comunicazione tra le diverse componenti avviene di solito tramite le due seguenti tecniche di base:

- condivisione di memoria;
- scambio esplicito di messaggi (sincrono o asincrono).

Possiamo inoltre affermare che esistono due modi di esecuzione per un sistema concorrente:

Esecuzione asincrona o interleaved In ogni istante di tempo, solo una componente alla volta effettua un'attività.

Esecuzione sincrona In ogni momento, tutte le componenti effettuano una certa attività.

Nei sistemi molto complessi si può avere il problema dell'*esplosione degli stati* perché risulta enorme il numero degli stati nella struttura globale. Oltre a questo problema, esistono altri comportamenti che creano difficoltà nell'analisi della concorrenza e che devono essere individuati:

Nondeterminismo Si ha quando un sistema può comportarsi in due o più modi differenti nonostante riceva un unico input. I sistemi paralleli hanno spesso questo comportamento a causa della comunicazione tra processi.

Deadlock In questo caso, si arriva in uno stato in cui nessuna componente del sistema può procedere (non ci sono azioni che possono essere eseguite). Solitamente questo fenomeno si verifica perché un gruppo di componenti attendono reciproci messaggi oppure perchè competono per condividere le risorse presenti nell'ambiente.
Nel modello a stati finiti di un processo, uno *stato di deadlock* è semplicemente uno stato senza transizioni in uscita.

Livelock E' presente quando un network (cioè una rete di processi paralleli) comunica internamente per un tempo infinito senza che qualsiasi componente comunichi con l'esterno. Per un osservatore può apparire simile alla situazione di deadlock, ma la presenza di attività interne crea ulteriori problemi. Il comportamento di livelock più comune è quello della *divergenza* per la quale un programma esegue una sequenza infinita di azioni interne.

Verificare la correttezza di un sistema tramite lo studio (ed eventualmente la risoluzione) di questi problemi è molto importante perché evita malfunzionamenti anche molto dannosi che si possono ritrovare a livello pratico.

Una *proprietà* è un attributo di un programma che risulta valido in qualunque possibile esecuzione del programma stesso. Nei programmi concorrenti le proprietà più interessanti si dividono in due categorie :

Safety Si garantisce che niente di cattivo avviene durante l'esecuzione del programma quindi non si raggiunge mai uno stato non corretto;

Liveness Si asserisce che qualcosa di buono prima o poi avviene, cioè nel futuro dell'esecuzione si raggiunge uno stato voluto.

Le più importanti proprietà di *Safety* sono:

- correttezza dello stato finale: nel sistema gli stati finali (cioè senza transizioni in uscita) non presentano alcun tipo di ambiguità o errore.
- assenza di *deadlock*: si evita che il sistema raggiunga una situazione nella quale non può più procedere.
- *mutua esclusione*: si impedisce che la stessa risorsa sia utilizzata da due processi nello stesso istante. In questo modo si evita il fenomeno negativo dell'*interferenza*, cioè l'aggiornamento scorretto dello stato in cui si trova la risorsa condivisa; tali errori sono molto difficili da individuare nei sistemi reali anche dopo test molto estesi.

Le più importanti proprietà di *Liveness* sono:

- terminazione: stabilisce che un programma prima o poi termina in uno stato finale. Questo è vero in generale per i programma sequenziali, ma nel caso della concorrenza si trovano frequentemente sistemi che non terminano.
- Progresso: si asserisce che, in qualunque stato si trovi il sistema, un'azione specificata sarà sempre eseguita. Da questa definizione si intuisce che il progresso è l'opposto di *starvation*, cioè la situazione in cui un'azione non è mai eseguita.

Oltre a safety e liveness, si possono anche osservare sistemi con un comportamento *divergente*. In questi casi il sistema può raggiungere uno stato in cui esegue un numero infinito di azioni che però non sono visibili dall'esterno. Si intuisce facilmente che se c'è questa caratteristica, allora non è detto che si raggiunga lo stato finale.

Altri comportamenti interessanti sono dati dalle *proprietà cicliche* che indicano la ripetizione periodica e infinita della stessa sequenza di azioni.

Quando si parla di *sistema reattivo* ci riferiamo ad un sistema che non lavora isolatamente, ma interagisce con altri nello svolgimento dei propri compiti e reagisce alle loro richieste. Perciò, di solito questi sistemi mostrano un'esecuzione concorrente dove le componenti possono competere per risorse condivise oppure coordinano le loro attività per ottenere un obiettivo comune.

Possiamo affermare che un processo è il comportamento di un sistema. La *Teoria dei Processi* studia proprio questi comportamenti ed è suddivisa in due attività principali:

1. Il *Modelling* è l'attività di rappresentare processi attraverso strutture matematiche o per mezzo delle espressioni di un linguaggio di descrizione per i sistemi. Con esso si crea il *modello* del sistema reale che ci interessa, cioè una sua rappresentazione semplificata. In genere, questa fase è immediatamente seguita dall'*implementazione* dei processi che compongono il sistema.

2. La *Verifica* consiste nel dimostrare le proprietà che questi processi forniscono al sistema. Per esempio si può verificare se due processi sono uguali e quindi se due sistemi si comportano nello stesso modo.

Lo *stato* di un processo è determinato dal valore delle variabili esplicite ed implicite che esso assume in un certo momento. Perciò l'esecuzione di un processo trasforma lo stato attraverso una serie di istruzioni le quali sono composte da una o più *azioni atomiche* che sono indivisibili.

Ogni azione di un processo può essere un'interazione con i processi vicini ed allora si parla di comunicazione, oppure le parti agiscono indipendentemente e quindi si ha la concorrenza.

Esistono molti tipi differenti di sistemi concorrenti: circuiti sincroni e asincroni, programmi con variabili condivise o che comunicano con lo scambio di messaggi, protocolli handshake. In quest'ultimo caso, definiamo l'*handshake* come un'azione indivisibile nella quale un valore è simultaneamente emesso da una componente e ricevuto da un'altra.

Semantiche delle Algebre di processi

Per studiare un sistema concorrente in modo matematico, sono state introdotte delle strutture algebriche con cui si può descrivere in maniera relativamente facile il modello che ci interessa. Questi formalismi sono detti *algebre di processi* e simulano la maggior parte dei comportamenti del sistema considerato.

Nelle applicazioni di queste algebre, la *specifica* e l'*implementazione* sono espresse come due processi distinti, ma la correttezza dei programmi si basa su una certa equivalenza di comportamento tra essi.

Ci sono tre differenti semantiche che permettono di interpretare un'algebra di processi:

- La *semantica operazionale* si occupa di come i vari costrutti di un particolare linguaggio possono essere eseguiti. Essa interpreta i programmi come diagrammi di transizione con azioni visibili e invisibili che permettono di muoversi tra i vari stati. Queste strutture descrivono il comportamento dei processi, ma non permettono di definire completamente il significato di un programma.
  Potremmo definire una semantica operazionale come una formalizzazione matematica di qualche strategia di implementazione del programma.

- La *semantica denotazionale* mappa un linguaggio in qualche modello astratto in modo che il valore di un programma composto è determinabile direttamente dai valori delle sue singole parti. In questo approccio, ad ogni frase del programma viene associata una denotazione, o *significato*, che di solito è composizionale, cioè il significato di ogni frase è una funzione dei significati delle sue sottofrasi.
  In genere questa semantica cerca di catturare il significato interno di un programma piuttosto che la strategia di implementazione, perciò è più astratta di quella operazionale ed è indipendente dalla macchina su cui si lavora.

- La *semantica algebrica* è definita da un insieme di leggi algebriche che sono gli assiomi di base in questa interpretazione. In questo caso, l'equivalenza tra processi è definita rispetto alle uguaglianze verificabili con queste leggi.
  Ad ogni istruzione sono associate *regole di inferenza* che indicano cosa è possibile asserire dopo l'istruzione, in base ai predicati veri prima dell'esecuzione.
  E' ragionevole considerarla come la più astratta tra le tre semantiche.

Per la semantica operazionale, in questo testo verranno descritti i diagrammi di transizione LTS in cui assumono notevole importanza le azioni che permettono al sistema di passare da uno stato all'altro. Invece, dal punto di vista algebrico, si mostrano le leggi per gli operatori più usati nella pratica ed in particolare viene descritto il calcolo CCS di Milner.

La semantica denotazionale è quella meno utilizzata nella teoria della concorrenza e qui verrà ignorata.

Equivalenze comportamentali e Logiche

Per confrontare due o più sistemi concorrenti, gli studiosi hanno introdotto molte *equivalenze comportamentali*. Ognuna di esse confronta due sistemi in base ad una certa categoria di comportamenti e per questo motivo si trovano diverse relazioni che legano le equivalenze tra loro. In particolare, assume notevole importanza il concetto di *bisimulazione*.

La nozione chiamata *equivalenza di osservazione* si riferisce a processi che esternamente seguono gli stessi cammini, ma possono differire nel comportamento interno.

Con la *Logica* possiamo definire un gran numero di proprietà che possono essere verificate o meno dai sistemi studiati. Due concetti generali che permettono di trattare una formula sono i seguenti:

Soddisfacibilità Una formula è soddisfacibile (o realizzabile) se esiste almeno un modello (ad esempio un LTS) in cui vale la proprietà che essa descrive.

Validità Una formula è valida se essa è soddisfacibile in tutti i modelli esistenti.

Se si vuole verificare la soddisfacibilità di una formula in un modello predeterminato, utilizziamo una tecnica chiamata *Model Checking*.

Capitolo 2

Algebre di Processi

Il primo passo per poter studiare ed analizzare un sistema concorrente è quello di utilizzare un formalismo che permetta di costruirne il modello matematico. Un *algebra di processi* è una formalismo che consente di modellare sistemi concorrenti che eventualmente interagiscono tra loro.

Gli elementi di base per ottenere questo formalismo sono le *azioni* e gli *operatori* (o combinatori). Questi ultimi permettono di costruire espressioni che simulano il comportamento del sistema considerato. Con questo simbolismo si facilita la specifica e la manipolazione di processi soprattutto in un computer. Nelle algebre più utilizzate gli operatori sono in numero ristretto, ma riescono a definire gran parte delle proprietà richieste perché possono simulare molti comportamenti di un sistema.

Sono state sviluppate molte teorie algebriche di questo tipo in modo da catturare i differenti aspetti di un sistema. La prima teoria che si occupava approfonditamente di comunicazione e concorrenza è quella di Petri (*Petri nets* negli anni '60). Essa è una generalizzazione della teoria degli automi che permette l'occorrenza di molte azioni (transizioni tra stati) indipendenti. In particolare si fa molta attenzione al concetto di *causalità tra azioni*.

Il CCS (o process calculus), che analizzeremo in dettaglio più avanti, fu proposto da Milner nel 1980; poi sono state realizzate altre algebre tra cui ricordiamo Lotos, Meije, ACP e TCSP.

Abbiamo già detto che esistono vari approcci per descrivere la semantica di un'algebra di processi. Dal punto di vista operazionale, possiamo utilizzare dei grafi di transizione che descrivono i comportamenti del sistema da modellare. In particolare, i due tipi di grafi più comuni sono le *Strutture di Kripke* (KS) ed i *Labelled Transition Systems* (LTS). Nel primo caso si etichettano gli stati del grafo per descrivere in che modo sono modificati dalle transizioni; nel secondo caso, come dice il nome, le transizioni sono etichettate con azioni che causano il passaggio da uno stato all'altro.

2.1 Labelled Transition Systems

I *sistemi di transizioni etichettate* furono introdotti da Keller nel 1976 come un modello formale per descrivere programmi paralleli ed in seguito sono stati usati per dare una semantica operazionale strutturale ai linguaggi di programmazione.

I sistemi di transizione sono quindi un modello relazionale astratto basati sulle nozioni primitive di *stato* e *transizione*. Molte propietà dei sistemi concorrenti possono essere studiate tramite questa rappresentazione; bisogna essere in grado di conoscere la capacità di tali sistemi di compiere azioni appartenenti ad un insieme predeterminato *Act*; azioni che possono essere istantanee o durature. Ovviamente un sistema sequenziale può effettuare al più un'azione nello stesso istante, certe azioni però possono essere azioni di sincronizzazione di un processo con il sistema concorrente di cui fa parte oppure segnali inviati dal resto del sistema al processo; è ovvio che quest'ultimo tipo di azioni occorrono solamente nel caso in cui i processi cooperino. Per essere più precisi forniamo una definizione formale di *Labelled Transition System* (LTS).

Un LTS è una quadrupla $(Q, q_0, Act \cup \{\tau\}, R)$ dove:

- Q è un insieme di *stati*;
- q_0 è lo *stato iniziale*;
- Act è un insieme finito e non vuoto di *azioni visibili*; τ è un'azione non contenuta in Act ed è utilizzato per rappresentare azioni interne alla struttura.
- $R \subseteq Q \times (Act \cup \{\tau\}) \times Q$ è la *relazione di transizione* tale che un'elemento $(s, \alpha, r) \in R$ se esiste la possibilità di passare da uno stato s ad un altro r tramite l'azione α.

Un LTS può essere rappresentato tramite un albero la cui radice è lo stato iniziale q_0 (gli stati iniziali sono rappresentati con un piccolo cerchio interno), le relazioni di transizione sono rappresentate dagli archi fra i nodi (archi etichettati con la azioni appartenenti ad $Act \cup \{\tau\}$), i nodi infine rappresentano gli stati appartenenti a Q.

Alcuni utili notazioni e definizioni che si riveleranno utili nel proseguio:

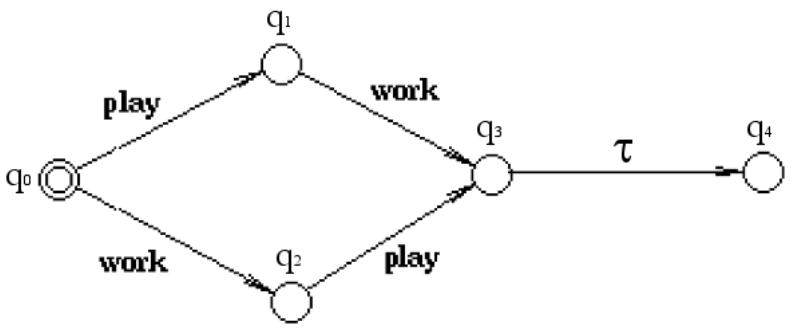
- Act verrà usato per denotare l'insieme delle azioni visibili; *a,b,c ...* denoteranno gli elementi di tale insieme;
- Act^* verrà usato per denotare l'insieme delle stringhe ottenibili a partire da A; s_1, s_2, $\cdots$ rappresentano gli elementi di tale insieme e ε la stringa vuota.
- Act_τ verrà usato per denotare le azioni appartenenti ad $Act \cup \{\tau\}$

Definizione 2.1. *Negli LTS, un* cammino *(o* computazione*)* π *è una sequenza* $(q_0, \alpha_0, q_1)\ (q_1, \alpha_1, q_2)\ (q_2, \alpha_2, q_3) \cdots$ *dove ogni tripla* $(q_i, \alpha_i, q_{i+1}) \in R$*. Un cammino può essere* finito *o* infinito *(in base al numero di transizioni che contiene) ed eventualmente può avere dei cicli al suo interno; possiamo inoltre trovare cammini* nulli *rappresentati da un sequenza vuota.*

Negli LTS le transizioni tra stati sono etichettate con azioni che possono essere visibili o invisibili, perciò è importante introdurre il significato ed il comportamento che assume un'azione all'interno di un sistema.

Quindi un Labeled Transition System è sostanzialmente un *automa non deterministico*. Nella *teoria degli automi*, un automa è rappresentato dal linguaggio che riconosce ed esso è equivalente ad un altro se e solo se i linguaggi accettati da essi sono uguali. Questo modo di determinare l'equivalenza crea problemi se si stanno trattando sistemi non-deterministici; perciò si inserisce la nozione di *bisimulazione* che permette di distinguere processi con lo stesso linguaggio, ma comportamenti diversi.

Esempio Bill-Ben 1: Vediamo adesso un semplice esempio di LTS formato da 5 stati (di cui uno iniziale), 3 azioni e 5 transizioni:



Dal nodo iniziale q_0 escono due archi distinti (transizioni $(q_0, play,\ q_1)$ e $(q_0, work, q_2)$) che raggiungono due stati differenti q_1 e q_2. Da questi si raggiunge sempre q_3 che ha un solo arco uscente (transizione (q_3, τ, q_4)) con il quale si raggiunge il nodo finale q_4.

2.2 Azioni interne ed esterne

Un'azione rappresenta un passo di computazione che viene fatto da un sistema per poter andare da uno stato all'altro. Nella maggior parte delle teorie algebriche, le azioni costituiscono un'interazione con l'ambiente esterno tramite determinate *porte* del sistema, oppure rappresentano una computazione interna.

In genere un'azione si indica con lo stesso simbolo della porta su cui agisce, perciò se denotiamo l'insieme delle porte nel sistema considerato con

il simbolo Λ, si trova che l'insieme di tutte le azioni visibili è il seguente:

$$Act \ = \ \{\alpha \ : \ \alpha \in \Lambda\}$$

La computazione interna che abbiamo introdotto è invece un'azione che viene eseguita nel sistema, ma in modo che nessun osservatore esterno possa vederla. Il simbolo utilizzato per questa azione è τ, perciò se vogliamo indicare l'insieme di tutte le azioni possibili si utilizza la notazione $Act \cup \{\tau\}$ (che spesso si abbrevia con Act_τ).

Questa transizione invisibile ha grande importanza nello studio delle equivalenze tra sistemi perché può essere considerata come quelle esterne oppure può avere un trattamento proprio a seconda del tipo di equivalenza che si sta analizzando. In questi ultimi casi si deve aggiungere la notazione ε che indica assenza di azioni visibili; quindi ε indica la possibilità di fare zero o più azioni τ (ε si utilizza quindi nello studio di equivalenze di osservazione).

Dal momento che una transizione tra due stati è accompagnata da un'azione, se dal processo P si passa a Q tramite $\alpha \in Act_\tau$, si parla di *derivazione* e si indica in questo modo:

$$P \xrightarrow{\alpha} Q$$

Se per passare da un processo all'altro servono più azioni si può anche utilizzare $s = (\alpha_1 \cdots \alpha_n)$ nella derivazione.

Nel caso in cui ci interessi il punto di vista di un osservatore esterno, le azioni interne dovranno essere ignorate nelle transizioni. Quindi si può avere la *discendenza* da P a Q tramite l'azione visibile $\alpha \in Act$:

$$P \stackrel{\alpha}{\Longrightarrow} Q$$

L'α-discendenza corrisponde alla derivazione di α, ma preceduta e seguita da zero o più azioni interne, ovvero:

$$P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* Q$$

Anche in questo caso possiamo utilizzare la sequenza di azioni visibili $s = (\alpha_1 \cdots \alpha_n)$.

L'ε-discendenza invece corrisponde alla derivazione di zero o più τ:

$$P \stackrel{\varepsilon}{\Longrightarrow} Q$$

In alcuni casi può essere interessante soltanto la capacità da parte del processo P di derivare o discendere un'azione α, mentre non ha importanza quale stato viene raggiunto. Queste situazioni più generiche sono indicate nei seguenti modi:

$$P \xrightarrow{\alpha} \qquad\qquad P \stackrel{\alpha}{\Longrightarrow}$$

Definizione 2.2. *Un processo P si dice* stabile *se nel suo stato iniziale non può eseguire una derivazione τ. In questi casi si utilizza la notazione $P \stackrel{\tau}{\not\longrightarrow}$.*

In alcune algebre di processi (tra le quali anche CCS) le azioni esterne si dividono in *azioni di input* e *azioni di output*.

Quando un processo esegue un'azione di input, significa che esso riceve un segnale su una determinata porta α ed in genere si indica l'azione proprio con il simbolo α. Nel caso dell'output, invece, è il processo stesso che emette il segnale attraverso una porta α ed in questo caso il simbolo viene soprabarrato ($\overline{\alpha}$).

α ed $\overline{\alpha}$ sono dette *azioni complementari* ed hanno un ruolo fondamentale nella comunicazione tra due processi che agiscono in parallelo.

Indicheremo con $\overline{Act}$ l'insieme delle azioni complementari di Act, e con $\mathcal{L}$ l'unione di questi due insiemi: $\mathcal{L} = Act \cup \overline{Act}$. Inoltre, dato un processo p, si indicherà con $\mathcal{L}(p)$ l'insieme delle azioni di p.

Esempio Bill-Ben 2: Riprendendo l'esempio introdotto per gli LTS, si osserva facilmente che le azioni visibili sono $play$ e $work$, mentre l'unica azione interna del sistema è τ. Se le azioni visibili sono degli output, in alcune algebre (compresa CCS) si scriverà $\overline{play}$ e $\overline{work}$.

Alcune derivazioni ovvie sono le seguenti:

$$q_0 \xrightarrow{play} q_1 \qquad q_0 \xrightarrow{work} q_2 \qquad q_1 \xrightarrow{work} q_3 \qquad q_2 \xrightarrow{play} q_3 \qquad q_3 \xrightarrow{\tau} q_4$$

Le tre discendenze più interessanti sono invece:

$$q_1 \overset{work}{\Longrightarrow} q_4 \qquad q_2 \overset{play}{\Longrightarrow} q_4 \qquad q_3 \overset{\varepsilon}{\Longrightarrow} q_4$$

2.3 Operatori

Gli operatori servono per comporre le azioni ed i processi in espressioni che descrivono il comportamento di un sistema concorrente. Ad ogni operatore è associato un certo numero di *assiomi* e *regole di transizione* (regole di inferenza) che permettono al sistema stesso di passare da uno stato ad un altro.

Solitamente una regola di transizione viene definita tramite una barra orizzontale sopra la quale si scrive un insieme di *premesse* $\pi_1, \cdots, \pi_n$ e sotto la quale si indica la *conclusione* $c \rightarrow c'$:

$$\frac{\pi_1 \, , \, \cdots \, , \pi_n}{c \rightarrow c'}$$

Data una regola di questo tipo possiamo dire che se valgono le condizioni $\pi_1 \, , \, \cdots \, , \pi_n$, allora è sicuramente valida l'implicazione $c \rightarrow c'$.

In un assioma invece non sono presenti le premesse, perciò la conclusione è sempre valida:

$$\frac{}{c \rightarrow c'}$$

Ogni algebra di processi ha un numero fissato di operatori, i quali possono essere *statici* o *dinamici*. Nel primo caso il combinatore *opr* è presente sia prima che dopo l'esecuzione della transizione in cui viene utilizzato, quindi la sua costruzione persiste attraverso qualunque sequenza di azioni:

$$\frac{P_{i1} \xrightarrow{\alpha_1} P'_{i1} \quad \cdots \quad P_{im} \xrightarrow{\alpha_m} P'_{im}}{opr(P_1, \cdots, P_n) \xrightarrow{\alpha} opr(P'_1, \cdots, P'_n)}$$

dove $\{i1, \cdots, 1m\} \subseteq \{1, \cdots, n\}$.

Nei combinatori dinamici, al contrario, *opr* è presente prima dell'azione, ma è assente dopo.

Inoltre gli operatori possono essere *deterministici* o *non deterministici* a seconda del modo in cui influenzano appunto il *determinismo* di un sistema.

Qui trattiamo gli operatori più utilizzati e conosciuti nella teoria. Possiamo suddividerli in gruppi per comprenderne meglio l'utilizzo.

2.3.1 Processi Elementari

Prima di descrivere gli operatori veri e propri, si devono indicare alcuni concetti fondamentali per la costruzione del modello di un sistema. Innanzitutto diciamo che ad ogni processo può essere assegnato un *nome* che lo identifica all'interno del sistema. Questo nome in genere è scritto in lettere maiuscole (o almeno quella iniziale).

I *processi elementari* costituiscono le componenti di base per poter descrivere qualunque sistema. Una singola azione α può da sola rappresentare un processo che esegue una sola transizione e poi si ferma.

Un'altro processo molto importante è quello *inattivo*. Esso viene solitamente indicato con i simboli 0 , *nil* o *stop* ed indica un processo che non esegue alcuna attività. Nel solito esempio Bill-Ben possiamo dire che una volta raggiunto lo stato q_4 si esegue il processo inattivo *stop* perché non ci sono più azioni da svolgere (nessun arco uscente).

EQUAZIONE DI DEFINIZIONE Questo operatore viene comunemente indicato con $\stackrel{def}{=}$, ma talvolta si trovano anche i simboli $=$ o $\stackrel{\Delta}{=}$.

Nella definizione $N \stackrel{def}{=} E$ si ha che l'espressione E (formata da altri operatori applicati ad azioni e processi) assume il termine N ed ogni volta che troveremo questo nome di processo si avrà il comportamento indicato da E. In questo caso la regola di transizione sarà:

$$\frac{E \xrightarrow{\alpha} E'}{N \xrightarrow{\alpha} E'} \quad (N \stackrel{def}{=} E)$$

Il nome di processo N può anche essere presente nell'espressione indicata a destra e questo permette ai sistemi di essere definiti ricorsivamente.

2.3.2 Operatori di Base

Permettono la costruzione di processi finiti semplici. I più importanti sono gli operatori per la sequenzializzazione delle azioni e quelli di scelta. Con essi possiamo già costruire grafi di processo come gli LTS.

SEQUENZIALIZZAZIONE Di solito questo operatore è formato semplicemente da un punto preceduto da un'azione qualsiasi del sistema e seguito da una nuova espressione di processi: $\alpha.E$
Il significato è l'esecuzione dell'azione α dopo la quale si prosegue con il comportamento descritto da E. Per questo operatore si trova il semplice assioma

$$\frac{}{\alpha.E \xrightarrow{\alpha} E}$$

perciò è un combinatore dinamico (l'operatore non è più presente a destra dell'implicazione conclusiva) e deterministico. Ad esempio l'espressione $\alpha.\beta.\tau.0$ indica un processo che esegue in sequenza le azioni visibili α β, poi fa un'azione interna ed infine si comporta come 0, cioè il processo che non svolge nessuna operazione.

COMPOSIZIONE SEQUENZIALE Date due espressioni di processo E ed F (costruite utilizzando processi, operatori ed azioni), la loro composizione sequenziale si indica con $E\,;\,F$. Questo operatore indica che F comincia solo quando l'espressione di sinistra E ha completato le sue azioni. In genere, l'azione finale è indicata con il simbolo $\surd$ (pronunciato tick) e le regole di transizione sono le seguenti:

$$\frac{E \xrightarrow{\alpha} E'}{E;F \xrightarrow{\alpha} E';F} \quad (\alpha \neq \surd) \qquad \frac{E \xrightarrow{\surd} E'}{E;F \xrightarrow{\tau} F}$$

Quindi E esegue tutte le sue azioni, ma dopo la sua azione finale si passa definitivamente all'espressione F. Tutte le azioni in E che si trovano dopo la $\surd$ non vengono effettuate.

SCELTA O SOMMA Questo operatore è dinamico e soprattutto può descrivere un sistema non deterministico. Si indica di solito con il simbolo + ai cui lati troviamo due espressioni di processo. Si parla di scelta perché se abbiamo E + F allora il sistema può eseguire le transizioni di E o quelle di F. In particolare si perde il determinismo quando queste due espressioni cominciano con la stessa azione α perché con lo stesso comportamento si possono seguire due strade diverse (ad esempio nel processo $\alpha.E + \alpha.F$).

Le due regole di transizione indicano quale scelta viene fatta.

$$\frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} \qquad \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}$$

E'possibile definire una somma più generale con più di due operandi. Questo avviene utilizzando il simbolo di *sommatoria* e si definisce la regola di transizione generale:

$$\frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i \in I} E_i \xrightarrow{\alpha} E'_j} \quad (j \in I)$$

Quindi con la somma si sceglie un solo cammino tra tutti quelli possibili. Ovviamente quando $I = \emptyset$ siamo di fronte al processo inattivo.

Nel processo $N \stackrel{def}{=} \beta.0 + \alpha.\gamma.0$ si può fare la sequenza di azioni α γ oppure si esegue soltanto β ed il processo si ferma. Se fosse $\beta = \alpha$ si troverebbe il non determinismo già all'inizio perché con la stessa azione si raggiungono due stati differenti.

Quello che abbiamo descritto è l'operatore di scelta più comunemente usato nelle algebre di processi, ma possiamo anche suddividerlo in due tipi di somma che si comportano in modo differente rispetto alle azioni interne.

Con l'operatore di *scelta interna pura* $E \oplus F$ la scelta del processo che deve proseguire avviene tramite azioni invisibili non controllabili dall'esterno:

$$\frac{}{E \oplus F \xrightarrow{\varepsilon} E} \qquad \frac{}{E \oplus F \xrightarrow{\varepsilon} F}$$

L'operatore di *scelta esterna* $E \,\Box\, F$ ha le seguenti regole di transizione:

$$\frac{E \xrightarrow{\alpha} E'}{E \Box F \xrightarrow{\alpha} E'} \quad (\alpha \neq \tau) \qquad \frac{E \xrightarrow{\varepsilon} E'}{E \Box F \xrightarrow{\varepsilon} E' \Box F}$$

in caso di azione interna (anche sapendo quale espressione la causa) non si viene effettuata la scelta.

Esempio Bill-Ben 3: Utilizzando gli operatori di base, possiamo descrivere l'esempio utilizzato in precedenza per gli LTS. Osservando il grafo di transizione disegnato nel paragrafo 2.1 si ottiene la seguente espressione con sequenzializzazione e somma:

$$play.\, work\, \tau.\, nil \;+\; work.\, play.\, \tau.\, nil$$

dove *nil* è il processo inattivo che non esegue alcuna attività.

2.3.3 Operatori di Parallelismo

Questi operatori esprimono il parallelismo e la concorrenza. Nella pratica, il comportamento dei sistemi è molto spesso dovuto all'esecuzione in parallelo di molti processi che possono condizionare l'esecuzione degli altri. Quindi il sistema completo è formato da tante entità che eseguono azioni di mutua comunicazione.

Per riuscire a modellare tali sistemi concorrenti, Milner introdusse il primo operatore binario per l'esecuzione in parallelo di due processi. In seguito sono stati introdotti altri operatori che gestiscono la concorrenza in diversi modi.

Composizione Parallela Questo operatore statico permette la concorrenza tra due processi che eseguono azioni complementari, quindi è presente nelle algebre di processi in cui si ha distinzione tra input e output su una porta.

Si denota con il simbolo | che mette in relazione due processi. Il sistema $P \mid Q$ esegue l'*interleaving* dei due processi oppure la *sincronizzazione*. Nel primo caso le due espressioni proseguono facendo le proprie azioni in sequenza (non necessariamente alternandosi), mentre si sincronizzano quando vengono eseguite due azioni complementari α e $\overline{\alpha}$. In questo caso il comportamento risultante è l'azione silente τ. In questo caso si può parlare di comunicazione tra due processi sulla porta α.

Stavolta le regole di transizione sono tre:

$$\frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

$$\frac{E \xrightarrow{\alpha} E' \ \wedge \ F \xrightarrow{\overline{\alpha}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

Esempio Bill-Ben 4 Supponiamo che il sistema Bill-Ben (finora trattato come un unico processo) sia invece formato da due processi che agiscono in parallelo; inoltre le azioni *play* e *work* sono degli output, mentre l'azione interna τ viene eseguita su una porta etichettata con *meet*.

I due processi svolgono la loro prima azione separatamente (emettendo i segnali di uscita dalle porte *play* e *work*) e dopo possono sincronizzarsi sulla porta *meet*:

$$\textsc{Bill} \stackrel{def}{=} \overline{play}.meet.nil$$

$$\textsc{Ben} \stackrel{def}{=} \overline{work}.\overline{meet}.nil$$

$$\textsc{Bill_Ben} \stackrel{def}{=} (\textsc{Bill} \mid \textsc{Ben})$$

Le derivazioni che risultano dalle regole di transizione viste sono:

- $\overline{play}.meet.nil \xrightarrow{\overline{play}} meet.nil$ che implica $\textsc{Bill} \xrightarrow{\overline{play}} meet.nil$ (dalla regola per $\overset{def}{=}$)
- $(\textsc{Bill}|\textsc{Ben}) \xrightarrow{\overline{play}} (meet.nil|\textsc{Ben})$
- $(meet.nil|\textsc{Ben}) \xrightarrow{\overline{work}} (meet.nil|\overline{meet}.nil)$
- $(meet.nil|\overline{meet}.nil) \xrightarrow{\tau} (nil|nil)$ (non ci sono più transizioni possibili)

Le ultime tre derivazioni coinvolgono l'operatore di composizione parallela per l'interleaving e per la sincronizzazione.

Operatore Parallelo In alcuni casi per la concorrenza tra due processi si trova questo combinatore $\|$ (anche Par) che rappresenta l'esecuzione concorrente in modo diverso. Infatti il sistema $P \| Q$ può eseguire l'interleaving come prima, oppure sincronizza i due processi quando essi eseguono una stessa azione α (le co-azioni $\overline{\alpha}$ non esistono). Inoltre il risultato per il sistema non è un passo interno, ma sempre l'azione visibile α. Le regole di transizione per l'interleaving sono le stesse, mentre quella per la sincronizzazione diventa:

$$\frac{E \xrightarrow{\alpha} E' \ , \ F \xrightarrow{\alpha} F'}{E \| F \xrightarrow{\alpha} E' \| F'}$$

Quindi nell'esempio Bill_Ben, dove ogni azione soprabarrata $\overline{\alpha}$ viene sostituita da quella solita α, si hanno le derivazioni:

$$(\textsc{Bill} \| \textsc{Ben}) \xrightarrow{play} (meet.nil \| \textsc{Ben}) \xrightarrow{work} (meet.nil \| meet.nil) \xrightarrow{meet} (nil \| nil)$$

Notiamo che in questo caso l'azione concorrente *meet* è visibile e quindi dovrà essere internalizzata in un altro modo (operatori di astrazione).

Interleaving Gli operatori di parallelismo che abbiamo appena visto hanno la proprietà che i processi possono sincronizzare su certe porte. Invece con l'operatore di interleaving si gestiscono processi completamente indipendenti l'un l'altro.

Indichiamo l'interleaving di due espressioni con $E \,|||\, F$; in questo caso il sistema eseguirà in sequenza le azioni dei due processi senza mai effettuare comunicazioni. Se sia E che F possono effettuare la stessa azione α, allora si ha una scelta non deterministica su quale delle due espressioni prosegue. Questo operatore ha le proprietà simmetrica, associativa e distributiva.

Esempio Bill-Ben 5: Nel nostro solito sistema, facendo l'interleaving tra Bill e Ben, si avrà la seguente sommatoria di processi:

$$\text{Bill} \;|||\; \text{Ben} \stackrel{def}{=} \begin{array}{l} \overline{work}.\overline{play}.meet.\overline{meet} + \overline{play}.\overline{work}.meet.\overline{meet} + \\ \overline{work}.\overline{play}.\overline{meet}.meet + \overline{play}.\overline{work}.\overline{meet}.meet \end{array}$$

2.3.4 Operatori di Astrazione

Spesso è utile rimuovere certe azioni rispetto alla visuale dell'ambiente. Questo significa che viene modificato il comportamento esterno del programma perché alcune azioni, inizialmente visibili, vengono *internalizzate* (cioè diventano τ). Gli operatori di astrazione permettono questo tipo di modifiche sui sistemi.

Operatore di Restrizione Può essere assegnato ad un processo P ed ha la definizione $\backslash L$ dove $L \subseteq Act$. Se L contiene una sola azione si scrive $\backslash\alpha$. La restrizione ha quindi come parametro un insieme di etichette che vengono internalizzate, cioè diventano locali all'interno del sistema e non sono più visibili da un osservatore esterno. Ovviamente l'azione τ non può essere ristretta perché è già interna al sistema. E' importante sottolineare che oltre alle azioni $\alpha \in L$, sono internalizzate anche tutte le eventuali azioni complementari $\overline{\alpha}$.

In conclusione, nel processo P ristretto da L le azioni presenti in L sono proibite a meno che non facciano parte di una sincronizzazione. Per questo motivo utilizzando questo operatore l'insieme di comportamenti osservabili diminuisce. Dalla regola di transizione si vede che oltre ad essere deterministico è anche un operatore statico:

$$\frac{E \stackrel{\alpha}{\longrightarrow} E'}{E \backslash L \stackrel{\alpha}{\longrightarrow} E' \backslash L} \quad (\alpha \,,\, \overline{\alpha} \notin L)$$

Nel seguente esempio si vede un sistema formato dal solo processo non deterministico Coin definito ricorsivamente. Con la restrizione sulla porta *heads*, il sistema può fare l'azione *toss* e proseguire con *tails*; se invece la scelta ricade sul sommando di destra, dopo *toss* non ci sono più comportamenti attuabili.

$$\textsc{Coin} \stackrel{def}{=} (toss.tails.\textsc{Coin} + toss.heads.\textsc{Coin}) \backslash heads$$

Le derivazioni potranno essere le seguenti:

- $\textsc{Coin} \stackrel{toss}{\longrightarrow} (heads.\textsc{Coin}) \backslash heads$ / *longrightarrow*
- $\textsc{Coin} \stackrel{toss}{\longrightarrow} (tails.\textsc{Coin}) \backslash heads \stackrel{tails}{\longrightarrow} \textsc{Coin}$
- $(heads.\textsc{Coin}) \backslash heads$ / *longrightarrow*

- $(tails.\textsc{Coin}) \setminus heads \xrightarrow{tails} \textsc{Coin}$

La restrizione assume grande importanza soprattutto se applicata ad un'azione che sincronizza due processi. Si parla perciò di *composizione ristretta* che talvolta viene indicata con

$$P \mid_L Q \stackrel{def}{=} (P \mid Q) \setminus L$$

Se abbiamo $(P|Q) \setminus \alpha$ e se i due processi hanno uno α e l'altro $\overline{\alpha}$ tra i loro comportamenti, sicuramente queste azioni non possono essere eseguite isolatamente, ma solo tramite la sincronizzazione sulla porta specificata ottenendo come risultato τ.

Esempio Bill-Ben 6: Nel sistema Bill_Ben, le regole di transizione della composizione parallela non vietano di fare $(meet.nil|\textsc{Ben}) \xrightarrow{meet} (nil|\textsc{Ben})$ evitando perciò la sincronizzazione. Restringendo l'azione *meet* su Bill_Ben sicuramente il cammino del sistema è:

$$(\textsc{Bill}|\textsc{Ben})\setminus meet \xrightarrow{\overline{play}} (meet.nil|\textsc{Ben})\setminus meet \xrightarrow{\overline{work}} (meet.nil|\overline{meet}.nil)\setminus meet$$

$$(meet.nil \mid \overline{meet}.nil)\setminus meet \xrightarrow{\tau} (nil \mid nil)\setminus meet$$

Ovviamente le prime due derivazioni singole posso essere scambiate di ordine e la composizione parallela finale equivale a *nil*. Si conclude che la comunicazione tra i due processi Bill e Ben avviene internamente al sistema sulla porta etichettata con *meet* e sempre dopo che i due hanno effettuato i propri output iniziali.

HIDING Un'altro combinatore che internalizza le azioni, ma agendo in modo differente dalla restrizione, è l'*hiding*. Dato un processo P, supponiamo di voler nascondere le azioni indicate nell'insieme L, cioè vogliamo che esse siano rese invisibili ad ogni osservatore esterno al sistema. Allora con P/L (P hiding L) qualsiasi $\alpha \in L$ diventa un'azione interna τ.

Questo operatore statico è essenziale per ridurre la complessità di sistemi molto grandi perché è possibile minimizzarne la dimensione rimuovendo le azioni interne ottenute con l'hiding. In genere viene applicato a processi composti e le sue regole di transizione sono:

$$\frac{E \xrightarrow{\alpha} E'}{E/L \xrightarrow{\alpha} E'/L} \quad (\alpha \notin L) \qquad \frac{E \xrightarrow{\alpha} E'}{E/L \xrightarrow{\tau} E'/L} \quad (\alpha \in L)$$

Un esempio semplice per l'utilizzo di questo operatore è il seguente:

$$\textsc{User} \stackrel{def}{=} (acquire.use.release.\textsc{User})/use$$

Il processo User esegue in sequenza le azioni *acquire*, τ e *release* (si è nascosto *use*) e con una minimizzazione successiva si trova la semplice sequenza *acquire-release*.

Si osserva che, mentre la restrizione preserva il determinismo, non accade altrettanto per l'hiding. Questo avviene perché se un processo deterministico ha già una τ-derivazione, con l'hiding se ne può inserire un'altra che crea un cammino differente. Ad esempio in $P \stackrel{def}{=} (\tau.\beta.P + \alpha.\gamma.P)$ se aggiungiamo $/\alpha$, una stessa azione silente iniziale porta a due computazioni diverse (non determinismo).

Esempio Bill-Ben 7: Quando abbiamo visto il parallelismo BILL $\|$ BEN, l'azione finale *meet* era visibile. Un modo per renderla interna è quello di aggiungere l'operatore $/meet$ e quindi l'ultima azione sarà τ.

2.3.5 Altri Operatori

Relabelling Data la funzione $f : Act_\tau \to Act_\tau$ (quindi associa un'etichetta di azione con un'altra) si ottiene l'operatore statico $[f]$ chiamato *relabelling* (relabelling o renaming). Con esso si possono rinominare le azioni di un sistema in modo da gestire meglio la sincronizzazione e l'interleaving.

La f rispetta le seguenti regole:

$f(\tau) = \tau$ perché non si può rinominare l'azione interna

$f(\overline{\alpha}) = \overline{f(\alpha)}$ quindi si rietichettano anche i complementari

L'unica regola di transizione è molto semplice:

$$\frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

quindi $E[f]$ esegue le stesse transizioni di E eventualmente rinominate.

Le coppie della funzione possono anche essere scritte esplicitamente sotto la forma $[l'_1/l_1, \cdots, l'_n/l_n]$ assumendo che, per tutte le azioni α che non sono espresse dagli l_i, si abbia $f(\alpha) = \alpha$.

Il relabelling viene solitamente fatto per assicurare che i processi composti si sincronizzino sulle azioni desiderate. Per esempio vediamo come si comporta un processo Server che fornisce un servizio invocato dal Client:

$$\text{CLIENT} \stackrel{def}{=} \overline{call}.wait.continue.\text{CLIENT}$$

$$\text{SERVER} \stackrel{def}{=} request.service.\overline{reply}.\text{SERVER}$$

$$\text{CLIENT_SERVER} \stackrel{def}{=} (\text{CLIENT} \mid \text{SERVER}\,[call/request, reply/wait])$$

Qui il relabelling sul processo Server permette la sincronizzazione con il Client quando si incontrano le azioni call e reply come si vede dalle seguenti derivazioni:

- $\textsc{Server}\ [f] \overset{f(request)}{\longrightarrow} service.\overline{reply}.\textsc{Server}\ [f]$ con f(request) = call
- $\textsc{Client} \overset{\overline{call}}{\longrightarrow} wait.continue.\textsc{Client}$ perciò si ottiene la sincronizzazione
- $(\textsc{Client}|\textsc{Server}[f]) \overset{\tau}{\longrightarrow} (wait.continue.\textsc{Client}|service.\overline{reply}.\textsc{Server}[f])$

Dopo la transizione *service*, in modo analogo al precedente si ha la comunicazione sulla porta *reply* (per il relabelling eseguito su *wait*). Infine, con la derivazione di *continue*, si torna al sistema iniziale:

$$(wait.continue.\textsc{Client} \mid service.\overline{reply}.\textsc{Server}\ [f]) \overset{service}{\longrightarrow}$$

$$\overset{service}{\longrightarrow} (wait.continue.\textsc{Client} \mid \overline{reply}.\textsc{Server}\ [f]) \overset{\tau}{\longrightarrow}$$

$$\overset{\tau}{\longrightarrow} (continue.\textsc{Client} \mid \textsc{Server}\ [f]) \overset{continue}{\longrightarrow} \textsc{Client_Server}$$

Esempio Bill-Ben 8: Se vogliamo rinominare le azioni di output in italiano basta semplicemente aggiungere il relabelling ai due processi:

$\textsc{Bill}[gioca/play, incontro/meet]$ $\textsc{Ben}[lavora/work, incontro/meet]$

LINK Si indica con l'espressione $P^\frown Q$ e permette il collegamento attraverso la comunicazione (composizione ristretta) tra la porta di input di uno dei processi e l'output dell'altro. Con il Link può avvenire la comunicazione handshake tra le due parti. Più precisamente:

$$P^\frown Q \ = \ (P[\overline{\sigma}.nil/nil]|\sigma.\ Q)\backslash\sigma$$

RICORSIONE Introducendo l'equazione di definizione ($\overset{def}{=}$) abbiamo detto che in $N \overset{def}{=} E$ il nome di processo N può essere utilizzato anche nell'espressione E. Se questo avviene, siamo di fronte alla ricorsione, perché N richiama se stesso durante la sua esecuzione.

La ricorsione è facilmente osservabile in $C \overset{def}{=} in.\overline{out}.C$ dove il processo C esegue un'azione di input sulla porta *in* seguita da un'azione di output sulla porta *out*. A questo punto il processo torna a comportarsi come descritto nella definizione di C, perciò il risultato finale è una sequenza alternata e infinita di *in* e $\overline{out}$. Dato il termine recX.E

$$\frac{E[recX.E/X] \overset{\mu}{\longrightarrow} E'}{recX.E \overset{\mu}{\longrightarrow} E'}$$

con $\mu \in Act_\tau$ Ecco un semplice esempio per dell'utilizzo di questo operatore:

$$recX.a.X + b.nil \overset{a}{\longrightarrow} recX.a.X + b.nil$$

2.4 Il Value-Passing

Finora abbiamo parlato di azioni ed operatori ai quali non è associato alcun valore. Ci siamo infatti riferiti al *calcolo di base* per modellare sistemi concorrenti. Esiste però la possibilità di utilizzare *calcoli con Value-Passing* nei quali, oltre alla pura sincronizzazione, si esprime la comunicazione di valori di qualsiasi tipo; infatti questo tipo di calcolo tramite opportuni accorgimenti può essere ricondotto al calcolo base.

Per semplificare lo studio, assumiamo che tutti i valori appartengono ad un fissato insieme V che può contenere qualunque tipo di elementi. L'esempio più immediato che si può fare è il seguente:

$$P \stackrel{def}{=} in(x).\,\overline{out}(x).\,P \quad \text{tale che} \quad x \in V$$

Con questa definizione, il processo P può ricevere in input (porta *in*) uno qualsiasi dei valori in V e subito dopo spedirlo come output (dalla porta *out*).

Le derivazioni di questo sistema saranno le seguenti:

$$P \stackrel{in(v)}{\longrightarrow} \overline{out}(v).\,P \stackrel{\overline{out}(v)}{\longrightarrow} P$$

dove v è l'elemento dell'insieme V che passa nel sistema.

Se l'insieme V è finito, allora si può tradurre qualunque processo in uno descritto con il calcolo senza Value-Passing. Nell'esempio sopra basta dare la possibilità a P di scegliere uno qualunque degli elementi di V da prendere in input e questo si fa utilizzando la seguente sommatoria:

$$P \stackrel{def}{=} \sum_{v \in V} in_v.\,\overline{out_v}.\,C$$

dove in_v e $\overline{out_v}$ sono rispettivamente le azioni di input e output relative ai diversi $v \in$V (al variare di $v \in$V).

Nel caso visto sopra la variabile è associata ad un'azione visibile (τ non può riferirsi a nessun valore), ma essa può anche essere collegata ad un processo. La forma generica $P(x) \stackrel{def}{=} E$ può essere tradotta nel seguente insieme di definizioni del calcolo di base:

$$\{P_v \stackrel{def}{=} E[v/x] \quad \text{t.c.} \quad v \in V\}$$

dove i vari P_v sono nuovi nomi di processi in cui al posto della variabile si inserisce uno dei valori del dominio.

Altri operatori che subiscono modifiche in questa traduzione sono la restrizione e il relabelling. Nel primo caso l'insieme di restrizione L diventa $\{\alpha_v : \alpha \in L, v \in$V$\}$; la funzione di relabelling f è modificata in modo che $\hat{f}(\alpha_v) = f(\alpha)_v$.

$(x = 3)$

$$\textsc{Countdown}(\text{x}) \stackrel{def}{=} tick.\ \textsc{Countdown}(\text{x-1}) \quad \text{con} \quad x \neq 0$$

$$\textsc{Countdown}(0) \stackrel{def}{=} beep.\ nil$$

Esempio 2.1. Con il Value-Passing si può definire il seguente conto alla rovescia in modo veloce:

Senza Value-Passing si ottiene una descrizione più lunga ma con le stesse informazioni:

$$\textsc{Countdown}_3 \stackrel{def}{=} tick.\ \textsc{Countdown}_2$$

$$\textsc{Countdown}_2 \stackrel{def}{=} tick.\ \textsc{Countdown}_1$$

$$\textsc{Countdown}_1 \stackrel{def}{=} tick.\ \textsc{Countdown}_0$$

$$\textsc{Countdown}_0 \stackrel{def}{=} beep.\ nil$$

Capitolo 3

Il Calcolo CCS

In questo capitolo introduciamo il CCS, cioè una delle algebre di processi più conosciute ed utilizzate nella teoria della concorrenza. Il *Calcolo dei Sistemi di Comunicazione* (Calculus of Communicating Systems) fu introdotto nel 1980 per studiare le proprietà strutturali di sistemi composti. Questo calcolo ha una struttura semplice, ma molto efficiente nel modellamento di sistemi concorrenti.

E' composto da un piccolo insieme di operatori con cui si costruisce un'ampia varietà di descrizioni di sistemi. I blocchi di base di queste descrizioni sono le azioni le quali rappresentano passi di esecuzione interna oppure interazioni potenziali con l'ambiente esterno (attraverso input e output). Le azioni visibili prendono il nome della porta su cui agiscono e se sono output vengono soprabarrati. In genere l'insieme di tutte le azioni di questo calcolo si indica con:

$$A_{ccs} = Act \cup \{\tau\} \qquad \text{(Act è l'insieme di azioni visibili)}$$

In CCS, i processi sono indicati da una stringa che inizia con lettera maiuscola (anche tutto maiuscolo), mentre le azioni svolte da un processo sono stringhe con lettere minuscole. Gli operatori sono pochi, ma con essi si possono simulare quasi tutti i comportamenti di un sistema.

3.1 Struttura Sintattica di CCS

L'insieme delle azioni del CCS è definito da $A_{ccs} = Act \cup \{\tau\}$ dove Act comprende tutte le azioni esterne (input e output) che possono essere svolte dal sistema, mentre τ rappresenta l'azione interna.

Gli operatori di base del CCS sono i seguenti (indichiamo con E, F, G etc. espressioni formate da processi, azioni ed operatori):

Sequenzializzazione $\alpha.E$; $\overline{\alpha}.E$; $\tau.E$

Somma $E + F$; $\sum_{i \in I} E_i$ (I è un insieme indicizzato)

Composizione parallela $E \mid F$

Restrizione $E \setminus L \quad (L \subseteq A_{ccs} - \{\tau\})$

Relabelling $E[f] \quad (f : A_{ccs} \rightarrow A_{ccs})$

Equazione di Definizione $NAME \stackrel{def}{=} E$

Schematizzando, i termini di questo calcolo sono generati dalla seguente Grammatica espressa in *Backus Normal Form*:

$$P \ ::= \ nil \ \Big| \ \alpha.P \ \Big| \ P + Q \ \Big| \ P|Q \ \Big| \ P \setminus L \ \Big| \ P[f]$$

dove P e Q sono processi generici mentre nil è il processo inattivo che non esegue alcun comportamento.

Con CCS si possono simulare molti operatori presenti in altre algebre di processi. Vediamo due semplici esempi:

Hiding $P \,/\, L \stackrel{def}{=} (P \mid Ever(\overline{l_1}) \mid \cdots \mid Ever(\overline{l_n})) \setminus L$ dove $L = \{l_1, ..., l_n\}$ ed $Ever(l_i) = l_i.Ever(l_i)$. Con Ever si ripetono continuamente le azioni $\overline{l_i}$ e quindi appena il processo P svolge un certo l_j si ha la sincronizzazione che dà come risultato τ (l_j non è più visibile e questo è l'obiettivo dell'Hiding).

Link $P^\frown Q \stackrel{def}{=} (P[m/out]|Q[m/in]) \setminus m$ dove i due processi interagiscono tramite le porte in ed $\overline{out}$, perciò si ha la sincronizzazione sostituendo le etichette con m ed $\overline{m}$.

Abbiamo visto che alcuni operatori preservano il determinismo al contrario di altri. Supponendo che i processi P, Q e P_i siano deterministici, riassumiamo il comportamento dei combinatori presenti in CCS:

- 0, $\alpha.\, P$ e $P \setminus L$ sono sempre deterministici.
- $\sum_i \ \alpha_i.\, P_i$ lo è solo se gli α_i sono tutti distinti.
- $P \mid Q$ se i due processi non hanno azioni in comune o complementari tra loro.
- $P[f]$ se la funzione di relabelling è iniettiva sulle azioni di P.

Esempio Bill-Ben 9: Utilizzando l'algebra CCS, il sistema concorrente Bill_Ben può essere definito nel seguente modo:

BILL	$\stackrel{def}{=}$	$\overline{play}.\, meet.\, nil$
BEN	$\stackrel{def}{=}$	$\overline{work}.\, \overline{meet}.\, nil$
BILL_BEN	$\stackrel{def}{=}$	$(\text{BILL} \mid \text{BEN}) \backslash \{meet\}$

3.2 Semantica Operazionale di CCS

Con *semantica operazionale* di un'algebra di processi si intendono i passi di esecuzione che possono essere fatti da un processo. Questi comportamenti sono descritti dagli assiomi e dalle regole di transizione per gli operatori che nel CCS sono le seguenti:

Sequenzializzazione $\dfrac{}{\alpha.E \stackrel{\alpha}{\longrightarrow} E}$

Somma $\dfrac{E \stackrel{\alpha}{\longrightarrow} E'}{E + F \stackrel{\alpha}{\longrightarrow} E'} \qquad \dfrac{F \stackrel{\alpha}{\longrightarrow} F'}{E + F \stackrel{\alpha}{\longrightarrow} F'}$

Composizione parallela $\dfrac{E \stackrel{\alpha}{\longrightarrow} E'}{E|F \stackrel{\alpha}{\longrightarrow} E'|F} \qquad \dfrac{F \stackrel{\alpha}{\longrightarrow} F'}{E|F \stackrel{\alpha}{\longrightarrow} E|F'}$

$$\dfrac{E \stackrel{\alpha}{\longrightarrow} E' \ , \ F \stackrel{\overline{\alpha}}{\longrightarrow} F'}{E|F \stackrel{\tau}{\longrightarrow} E'|F'}$$

Restrizione $\dfrac{E \stackrel{\alpha}{\longrightarrow} E'}{E \backslash L \stackrel{\alpha}{\longrightarrow} E' \backslash L} \quad (\alpha \, , \, \overline{\alpha} \notin L)$

Relabelling $\dfrac{E \stackrel{\alpha}{\longrightarrow} E'}{E[f] \stackrel{f(\alpha)}{\longrightarrow} E'[f]}$

Equazione di definizione $\dfrac{E \stackrel{\alpha}{\longrightarrow} E'}{N \stackrel{\alpha}{\longrightarrow} E'} \quad (N \stackrel{def}{=} E)$

Possiamo associare l'algebra CCS ad un LTS in cui l'insieme di stati è formato dai processi del sistema, l'insieme di azioni visibili *Act* è composto dalle azioni esterne A_{ccs} (a cui si aggiunge quella interna τ) ed R comprende tutte le possibili derivazioni nel sistema (date dalle regole di transizione per gli operatori).

Più precisamente, la semantica transizionale di CCS può essere definita con il seguente labelled transition system:

$$\mathcal{L} \ = \ (\mathcal{P} \, , \, P_0 \, , \, A_{ccs} \, , \, \{\stackrel{\alpha}{\longrightarrow} : \alpha \in A_{ccs}\})$$

dove $\mathcal{P}$ è la classe delle *espressioni di processi* che costituiscono la sintassi dei termini CCS e P_0 è il processo iniziale del sistema.

Inoltre, per organizzare meglio un modello in CCS, esistono delle leggi equazionali che possono semplificare la struttura di un sistema. Queste leggi si applicano agli operatori e si dividono in:
– *Statiche*: si riferiscono ai combinatori statici di Composizione parallela, Restrizione e Relabelling.
– *Dinamiche*: coinvolgono gli operatori dinamici di Sequenzializzazione, Somma e Ricorsione.
– di *Espansione*: è una legge molto utile che coinvolge entrambi i gruppi.

LEGGI DINAMICHE Le *leggi monoidi* riguardano la somma:

1. $P + Q \;=\; Q + P$ (commutatività)
2. $P + (Q + R) \;=\; (P + Q) + R$ (associatività)
3. $P + nil \;=\; P$ (*nil* è l'elemento neutro per la somma)
4. $P + P \;=\; P$

τ-LAWS La seconda serie delle *leggi dinamiche* riguarda la sequenzializzazione e si focalizza sul significato dell'azione τ

1. $\alpha.\tau.\, Q \;= \alpha.\, Q$
2. $P + \tau.\, P \;= \tau.\, P$
3. $\alpha.(\, P + \tau.\, Q) + \alpha.\, Q \;= \alpha.(\, P + \tau.\, Q).$

Queste leggi possono apparire strane ad un primo esame, ma andando ad analizzare i relativi alberi di derivazione ci si può rendere conto della correttezza di tali leggi. Per esempio una conseguenza diretta delle $\tau - LAWS$ è la seguente uguaglianza:

$$P + \tau.(\, P + \; Q) = \tau.(\, P + \; Q).$$

Essa è facilmente dimostrabile usando la legge 2) delle $\tau - LAWS$. Il membro destro dell'uguaglianza, P, rappresenta la capacità di compiere azioni inizialmente possibili ma che rimangono tali anche dopo l'azione τ, in pratica le possibilità del sistema restano immutate anche posticipando le azioni di P dopo l'azione τ.

Queste sono le uniche leggi valide per l'azione τ, se ammettessimo la legge $\tau.P = P$ comporterebbe la seguente uguaglianza: $\alpha.P + \tau.b.Q = a.P + b.Q.$; questo comporterebbe un'errore perchè se il membro sinistro compie l'azione τ non sara più possibile compiere l'azione a, possibilità che il membro destro non possiede.

LEGGI STATICHE *Leggi della Composizione parallela*:

1. $P \mid Q = Q \mid P$ (commutatività)
2. $P \mid (Q \mid R) = (P \mid Q) \mid R$ (associatività)
3. $P \mid nil = P$ (*nil* è l'elemento identità anche per la composizione)

Leggi della Relabelling:

1. $P \setminus L = P$ (se P non ha azioni in $L \cup \overline{L}$)
2. $P \setminus K \setminus L = P \setminus (K \cup L)$
3. $P[f] \setminus L = P \setminus f^{-1}(L)[f]$
4. $(P \mid Q) \setminus L = P \setminus L \mid Q \setminus L$ (se P e Q non sincronizzano su azioni di L)

La legge (1) assicura che le azioni $\alpha \in$ Ł non appariranno visibili nel sistema, l'operatore di restizione diventa così superfluo. La Legge (2) è una ovvia conseguenza logica; a legge (3) asserisce che l'operatore di restrizione e di relabelling commutano mediante piccoli aggiustamenti, la legge (4) infine, è un tipo di legge distributiva che ci permette di distribuire l'operazione di restrizione sull'operazione di composizione parallela sino a quando le azioni interessate alla restizione non sono possibili veicoli di comunicazione.

Leggi del Relabelling (con Id funzione identità):

1. $P[Id] = P$
2. $P[f] = P[f']$ (se f e f' sono uguali sull'insieme di azioni di P)
3. $P[f][f'] = P[f' \circ f]$
4. $(P \mid Q)[f] = P[f] \mid Q[f]$ (se f ristretta alle azioni di $P \mid Q$ è iniettiva)

La legge (1) è la funzione identità, la legge (2) assicura che f e f' sortiranno lo stesso effetto su P, la legge (3) sfrutta la definizione di composizione di funzioni e a legge (4) assicura che non verranno creati più canali di comunicazione di quelli esistenti prima di distribuire l'operatore di relabelling (garantito dall'iniettività).

Alcuni esempi che si possono ottenere combinando le leggi statiche

Esempio 3.1.

1. $P[b/a] = P \; se \; a, \overline{a} \notin \mathcal{L}(P)$
2. $P\backslash a = P[b/a]\backslash b = P \; se \; b, \overline{b} \notin \mathcal{L}(P)$

3. $P\backslash[b/c] \;=\; P[b/c]\backslash a \; se \; b, c \neq a$

Nel esempio 2(1) si utilizzano le leggi (2) e (1) delle leggi di relabelling; nell'esempio 2(2) si utilizzano le proposizioni (2) e (1) delle leggi di restizione e la proposizione (1) delle leggi di relabelling; nell'esempio 2(3) si utilizzza la proposizione (2) e (1) delle leggi di relabelling.

LEGGE DI ESPANSIONE Spesso un sistema concorrente viene definito tramite Composizione parallela di molti processi sui quali viene effettuato un Relabelling in modo che possano sincronizzarsi tra loro, per poi applicare una Restrizione sull'insieme delle azioni di sincronizzazione. L'espansione permette di elaborare le azioni di questi P_i e trasformare la descrizione del sistema in una sommatoria di termini del tipo $\alpha.\, P$ (*Standard Concurrent Form*).

Le azioni manipolare sono di due tipi: quelle delle singole componenti e quelle che diventano azioni interne perché derivano da una comunicazione tra due processi.

Sia $P \equiv (P_1[f_1] \mid \cdots \mid P_n[f_n]) \setminus L$ allora:

$$\begin{aligned} P \;=\; & \sum \{\, f_i(\alpha).\, (P_1[f_1] \mid \cdots \mid P_i'[f_i] \mid \cdots \mid P_n[f_n]) \setminus L \;: \\ & \qquad P_i \xrightarrow{\alpha} P_i' \,,\; f_i(\alpha) \notin (L \cup \overline{L}) \,\} \\ + \; & \sum \{\, \tau.\, (P_1[f_1] \mid \cdots \mid P_i'[f_i] \mid \cdots \mid P_j'[f_j] \mid \cdots \mid P_n[f_n]) \setminus L \;: \\ & \qquad P_i \xrightarrow{\alpha} P_i' \,,\; P_j \xrightarrow{\beta} P_j' \,,\; f_i(\alpha) = \overline{f_j(\beta)} \,\} \end{aligned}$$

Diamo adesso la versione senza l'operazione di relabelling:

$P \equiv (P_1 \mid \cdots \mid P_n) \setminus L$ allora:

$$\begin{aligned} P \;=\; & \sum \{\, \alpha.\, (P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_n) \setminus L \;: \\ & \qquad P_i \xrightarrow{\alpha} P_i' \,,\; \alpha \notin (L \cup \overline{L}) \,\} \\ + \; & \sum \{\, \tau.\, (P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_j' \mid \cdots \mid P_n) \setminus L \;: \\ & \qquad P_i \xrightarrow{\alpha} P_i' \,,\; P_j \xrightarrow{\overline{\alpha}} P_j' \,,\; i < j \,\} \end{aligned}$$

Vediamo un semplice esempio in cui può essere molto utile la legge di espansione:

Esempio 3.2.

$$P \stackrel{def}{=} \alpha.\, P' + \beta.\, P'' \qquad\qquad Q \stackrel{def}{=} \overline{\alpha}.\, Q' + \gamma.\, Q'' \qquad\qquad R \stackrel{def}{=} (P \mid Q)\backslash\alpha$$

Nella composizione ristretta R ci sono due azioni del primo tipo (β e γ) ed una comunicazione sull'azione α. Quindi si ottiene:

$$R \stackrel{def}{=} \beta.\, (P'' \mid Q) \setminus \alpha \;+\; \gamma.\, (P \mid Q'') \setminus \alpha \;+\; \tau.\, (P' \mid Q') \setminus \alpha$$

Se conosciamo i processi P', P'', Q', Q'' si può iterare il procedimento fino ad avere una forma concorrente standard con solo prefissi e somme.

Esempio Bill-Ben 10: Nel solito esempio, si arriva alla forma concorrente standard con i seguenti passaggi:

$$\begin{array}{lcl}
\textsc{Bill_Ben} & \stackrel{def}{=} & \overline{play}.\,(meet.nil \mid \textsc{Ben}) \setminus \{meet\} + \overline{work}.\,(\textsc{Bill} \mid \overline{meet}.\,nil) \setminus \{meet\} \\
\textsc{Bill_Ben} & \stackrel{def}{=} & \overline{play}.\,\overline{work}.\,(meet.nil \mid \overline{meet}.\,nil) \setminus \{meet\} + \\
 & & \overline{work}.\,\overline{play}.\,(meet.nil \mid \overline{meet}.\,nil) \setminus \{meet\} \\
\textsc{Bill_Ben} & \stackrel{def}{=} & \overline{play}.\,\overline{work}.\,\tau.\,nil \;+\; \overline{work}.\,\overline{play}.\,\tau.\,nil
\end{array}$$

Nell'ultimo passaggio si è immediatamente trasformato $(nil|nil)\setminus\{meet\}$ nel semplice processo inattivo nil.

Capitolo 4

Nozioni preliminari di algebra

In questo breve capitolo introdurremo alcune nozioni algebriche, probabilmente già note, ma che ci saranno utili per i capitoli seguenti.

4.1 Relazioni, preordini ed equivalenze

Definizione 4.1 (Relazione (binaria)). *Una relazione $\mathcal{R}$ da un insieme $\mathcal{A}$ in un insieme $\mathcal{B}$ è un sottoinsieme del prodotto cartesiano dei due insiemi, cioè*

$$\mathcal{R} : \mathcal{A} \to \mathcal{B} \subseteq \mathcal{A} \times \mathcal{B}$$

Per una relazione possono essere verificate alcune proprietà:

Definizione 4.2 (Proprietà delle relazioni). *Sia $\mathcal{R}$ una relazione da $\mathcal{A}$ in $\mathcal{A}$, allora si dice che $\mathcal{R}$ è*

riflessiva *sse* $\forall x \in \mathcal{A}$, *implica* $(x, x) \in \mathcal{R}$

transitiva *sse* $\forall x, y, z \in \mathcal{A}$, $(x, y) \in \mathcal{R}$ *e* $(y, z) \in \mathcal{R}$ *implica* $(x, z) \in \mathcal{R}$

simmetrica *sse* $\forall x, y \in \mathcal{A}$, $(x, y) \in \mathcal{R}$ *implica* $(y, x) \in \mathcal{R}$

antisimmetrica *sse* $\forall x, y \in \mathcal{A}$, $(x, y) \in \mathcal{R}$ *e* $(y, x) \in \mathcal{R}$ *implica* $x = y$

Introduciamo la nozione di inversa di una relazione $\mathcal{R}$ e di composizione di due relazioni $\mathcal{R}_1$ e $\mathcal{R}_2$

$$\begin{aligned} \mathcal{R}^{-1} &= \{(y, x) : (x, y) \in \mathcal{R}\} \\ \mathcal{R}_1\mathcal{R}_2 &= \{(x, z) : (x, y) \in \mathcal{R}_1 \wedge (y, z) \in \mathcal{R}_2 \text{ per qualche } y\} \end{aligned}$$

Alcune relazioni , che possiedono determinate proprietà, vengono chiamate in modo caratterizzante; definiamo infatti:

Definizione 4.3 (Equivalenza). *Una relazione binaria $\mathcal{R}$ su un insieme $\mathcal{A}$ è chiamata* relazione di equivalenza *se è riflessiva, simmetrica e transitiva.*

Definizione 4.4 (Preordine). *Una relazione binaria $\mathcal{R}$ su un insieme $\mathcal{A}$ è chiamata* preordine *se è riflessiva e transitiva.*

Solitamente i preordini vengono indicati con il simbolo $\sqsubseteq$; essendo una relazione, è possibile trovare l'inverso di un preordine, e ciò ci consente di definire

Definizione 4.5 (Kernel di un preordine). *Dato il preordine $\sqsubseteq$, definiamo il* kernel del preordine *come il preordine stesso intersecato con la relazione inversa, cioè $\simeq = \sqsubseteq \cap \sqsubseteq^{-1}$.*

Questa operazione equivale a fare la chiusura simmetrica del preordine, ottenendo così una relazione di equivalenza. Questa possibilità di generare in modo molto diretto una relazione di equivalenza da un preordine ci sarà molto utile in seguito.

4.2 Contesti e congruenze

Introduciamo brevemente due concetti che in seguito saranno molto utilizzati:

Definizione 4.6 (Contesto). *Un contesto è un'espressione algebrica con un 'buco', che indicheremo con $C[\,]$*

Un esempio di contesto è $b + [\,]$ dove al posto di $[\,]$ possiamo sostituire una qualsiasi espressione della nostra algebra.

Definizione 4.7 (Congruenza). *Una relazione $\sim_{\mathcal{C}}$ è una* congruenza *se:* $\forall C[\,]$

$$P \sim_{\mathcal{C}} Q \quad \Rightarrow \quad C[P] \sim_{\mathcal{C}} C[Q]$$

Capitolo 5

Equivalenze fra processi

Risulta interessante ed utile stabilire se due sistemi sono equivalenti e nel caso si utilizzi lo stesso formalismo sia per la specifica del sistema (quanto gli si chiede di fare) sia per la sua implementazione (in quale modo si realizza) allora è anche possibile provare che una certa implementazione è corretta rispetto ad una certa specifica.

Ancora più interessante è poi la possibilità di sostituire una corposa specifica con una più compatta, una volta che se ne sia provata la loro equivalenza: è quello che viene detto 'sostituire uguali per uguali'. É infatti utile avere la possibilità di scambiare sottosistemi di provata equivalenza, nel senso che un sottosistema può prendere il posto di un'altro come parte di un sistema più grande senza modificare il comportamento del sistema globale.

Le equivalenze che andremo a vedere sono tutte basate sull'idea che due sistemi sono equivalenti se nessuna osservazione esterna li può distinguere; non è di interesse la struttura interna di un processo, ma il suo comportamento in relazione al mondo esterno: ciò che viene chiamato il *comportamento estensionale* od *osservazionale* del sistema.

5.1 Equivalenza di traccia

Un modo naturale di considerare due sistemi equivalenti consiste nel vedere se possono eseguire le stesse sequenze di azioni visibili, astraendo in questo modo dalle azioni interne (invisibili) di un sistema: cioè, due processi sono equivalenti se consentono lo stesso insieme di sequenze di azioni.

Definizione 5.1 (Traccia). *La traccia $s \in Act^*$ di un processo P è una sequenza di azioni $\alpha_1, \ldots, \alpha_n$ tale che esiste un certo Q raggiungibile da P eseguendo le azioni di s. Si parla di* traccia forte *se $P \xrightarrow{s} Q$ e di* traccia debole *se $P \stackrel{s}{\Longrightarrow} Q$.*

Possiamo ora introdurre il concetto di equivalenza di traccia:

Definizione 5.2 (Equivalenza di traccia). *Due sistemi P e Q si dicono* fortemente equivalenti di traccia*, $P \sim_t Q$ se*

$$\forall s \in Act_\tau{}^* \qquad P \xrightarrow{s} \iff Q \xrightarrow{s}$$

Analogamente, P e Q si dicono debolmente equivalenti di traccia*, $P \approx_t Q$ se*

$$\forall s \in Act^* \qquad P \stackrel{s}{\Rightarrow} \iff Q \stackrel{s}{\Rightarrow}$$

La differenza tra queste due equivalenze risiede nel modo di interpretare le azioni invisibili τ: nel caso dell'equivalenza forte le azioni τ sono viste al pari delle altre azioni visibili, mentre nell'equivalenza debole vengono usate, al posto delle derivazioni, le discendenze: una *discendenza* è una derivazione debole in cui si possono eseguire zero o più azioni τ all'interno della traccia di azioni visibili s.

L'equivalenza $\sim_t$ è quella che viene ormai utilizzata da tempo nella teoria degli automi e dei linguaggi: infatti, vengono identificati come equivalenti processi che possono generare lo stesso linguaggio (insieme di stringhe sull'alfabeto dato); considerando però sistemi che interagiscono con l'ambiente esterno, diventa importante sapere se certe comunicazioni avranno sempre luogo oppure se esiste la possibilità di deadlock: la trace equivalence non consente di differenziare sistemi con comportamenti diversi in questo senso. Cerchiamo di chiarire quanto detto con un esempio: si vede che $\alpha.nil \approx_t \alpha.nil + \tau.nil$ però, mentre nel primo processo si esegue sempre l'azione visibile α, nel secondo caso con un'azione interna (e quindi una sequenza visibile ε) si raggiunge proprio uno stato di deadlock, in quanto tutte le azioni esterne sono proibite.

Un'altro fatto è di notevole importanza: si vede facilmente che $\alpha.(\beta.nil + \gamma.nil) \sim_t \alpha.\beta.nil + \alpha.\gamma.nil$. É infatti intuitivo vedere che non ci sono modo per distinguere questi due processi considerando solo le azioni che possono eseguire: i due processi hanno le medesime traccie ($\alpha\beta$ e $\alpha\gamma$) ma nel processo di destra, una volta eseguita l'azione iniziale α si può fare solo una delle due transazioni, mentre nel caso di sinistra si ha ancora la possibilità di scegliere. Questo significa che l'equivalenza vale negli stati iniziali, ma non più negli stati raggiunti successivamente.

Una nota a suo favore va comunque fatta: l'equivalenza di traccia è preservata da tutti gli operatori di base ed è perciò una congruenza: questa affermazione vedremo non sarà sempre vero ed alcune equivalenze più importanti, non preservate dalla somma, dovranno essere ridefinite perchè questa proprietà sia verificata.

5.2 Equivalenza di traccia completa

Definizione 5.3 (Traccia completa). *La sequenza di azioni $s \in Act_\tau^*$ è una* traccia completa *per un processo p se $p \xrightarrow{s} p' \wedge p' \not\rightarrow$*

Quindi si considerano soltanto le sequenze di azioni che partono da p e raggiungono uno stato senza più transizioni possibili. L'insieme delle tracce complete di P si indica con $CT(P)$.

Definizione 5.4 (Equivalenza di traccia completa). *Due processi p e q si dicono* equivalenti di traccia completa*, indicato come $p \sim_{ct} q$ se sono equivalenti di traccia ed hanno le stesse tracce complete, cioè:*

$$p \sim_{ct} q \quad \Leftrightarrow \quad p \sim_t q \wedge CT(p) = CT(q)$$

Da questa definizione si intuisce banalmente che se due processi sono in relazione $\sim_{ct}$ allora essi sono sicuramente equivalenti di traccia. Però il contrario non è sempre vero e questo lo si capisce subito dal seguente semplice esempio:

$$\alpha.\,\beta\,.nil + \alpha.\,nil \;\sim_t\; \alpha.\,\beta\,.nil \qquad\qquad \alpha.\,\beta\,.nil + \alpha.\,nil \;\not\sim_{ct}\; \alpha.\,\beta\,.nil$$

infatti i due processi hanno le stesso tracce ε, α e $\alpha\beta$, ma a sinistra si trova la traccia completa α che non è presente nel processo di destra (in cui dopo α si può sempre fare una β-derivazione).

5.3 Equivalenza per fallimenti

Come detto poco sopra, il fatto che l'equivalenza a traccie eguagli un processo che termina in deadlock con uno che non lo fa, la rende in generale troppo debole; un'altra interessante equivalenza è la *testing equivalence* che sembra essere la più piccola equivalenza in grado di differenziare processi che terminano in uno stato di deadlock da processi che invece non terminano in quello stato.

Definizione 5.5 (Fallimento). *Un fallimento è una coppia (s, L), dove s è una traccia ed L è un insieme di etichette di azioni.*

Definizione 5.6. *Un fallimento (s, L) appartiene ad un processo P (per il caso strong) se esiste un processo P' tale che*

- $P \xrightarrow{s} P'$
- $P' \overset{\tau}{\not\rightarrow}$
- $\forall l \in L, P' \overset{l}{\not\rightarrow}$

Per il caso weak, invece, si ha

- $P \stackrel{s}{\Longrightarrow} P'$
- $P' \stackrel{\tau}{\not\rightarrow}$
- $\forall l \in L, P' \stackrel{l}{\not\rightarrow}$

La definizione precedente dice che il processo P può derivare (o discendere, se ci troviamo nel caso weak) la sequenza s ed arrivare in uno stato in cui non è possibile fare nessuna ulteriore azione (neanche τ) se l'ambiente consente solo le azioni in L.

Definizione 5.7 (Equivalenza per fallimenti). *Due processi P e Q si dicono* equivalenti per fallimenti *se possiedono esattamente gli stessi fallimenti; nel caso l'equivalenza sia forte si indica con $P \sim_f Q$, mentre se è debole con $P \approx_f Q$.*

É facile vedere che l'equivalenza per fallimenti implica l'equivalenza a traccie, in quanto una traccia è parte di un fallimento.

5.4 Testing equivalence

Un'elegante caratterizzazione alternativa della failure equivalence è chiamata *testing equivalence*, secondo la quale due processi sono equivalenti quando passano gli stessi test.

Consideriamo dunque un processo P definito su Act_τ ed un osservatore O definito su $Act_\tau \cup \{w\}$ dove $w \notin Act$ è l'azione che indica il soddisfacimento dell'osservatore. Un'osservatore può essere visto come un agente che svolge dei test specifici sui processi. Il risultato del test di O su P viene indicato con una *computazione* $c \in Comp(O, P)$, l'insieme non vuoto di tutte le computazioni possibili. Naturalmente ora dobbiamo definire cos'è una computazione:

Definizione 5.8 (Computazione). *Dato un processo P ed un osservatore O con stati iniziali rispettivamente p e o, si definisce una* computazione *da $< p, o >$ come una sequenza finita di coppie di stati $< p_n, o_n >$ dove:*

1. *$< p_0, o_0 >$ corrisponde a $< p, o >$;*
2. *i) $< p_n, o_n > \stackrel{\tau}{\longrightarrow} < p_{n+1}, o_{n+1} >$ se $p_n \stackrel{\tau}{\longrightarrow} p_{n+1}$ e $o_n = o_{n+1}$ oppure $o_n \stackrel{\tau}{\longrightarrow} o_{n+1}$ e $p_n = p_{n+1}$*
   ii) $< p_n, o_n > \stackrel{\alpha}{\longrightarrow} < p_{n+1}, o_{n+1} >$ se $p_n \stackrel{\alpha}{\longrightarrow} p_{n+1}$ e $o_n \stackrel{\alpha}{\longrightarrow} o_{n+1}$
3. *Se la sequenza è finita allora l'ultimo elemento $< p_k, o_k > \stackrel{\mu}{\not\rightarrow}$ per ogni $\mu \in Act_\tau$.*

Si ha quindi:

i) P *may satisfy* O se esiste una sequenza $s \in Act^*$ tale che $p_0 \stackrel{s}{\Rightarrow}$, $o_0 \stackrel{s}{\Longrightarrow} o_n$ e $o_n \stackrel{w}{\rightarrow}$;

ii) P *must satisfy* O se per ogni computazione $< p_0, o_0 > \stackrel{\mu_1}{\longrightarrow} < p_1, o_1 > \stackrel{\mu_2}{\longrightarrow} \ldots$ $\exists n \geq 0$ tale che $o_n \stackrel{w}{\rightarrow}$

Il significato di quanto scritto sopra è che nel caso *may* deve esistere almeno una computazione in cui due processi eseguono la stessa sequenza di azioni ed alla fine il processo osservatore è soddisfatto; nel caso *must*, qualsiasi computazione deve portare al soddisfacimento dell'osservatore.

Con le definizioni appena riportate si possono introdurre i preordini (indichiamo con $\mathcal{O}$ l'insieme degli osservatori):

$$P \sqsubseteq_{may} Q \quad \Longleftrightarrow \quad \forall O \in \mathcal{O} \quad P \, may \, O \Rightarrow Q \, may \, O$$

$$P \sqsubseteq_{must} Q \quad \Longleftrightarrow \quad \forall O \in \mathcal{O} \quad P \, must \, O \Rightarrow Q \, must \, O$$

Come detto, prendendo in considerazione i kernel di questi due preordini possiamo definire la *may-equivalence* e la *must-equivalence* rispettivamente in questo modo:

$$P \simeq_{may} Q \quad \Longleftrightarrow \quad P \sqsubseteq_{may} Q \quad \wedge \quad Q \sqsubseteq_{may} P$$

$$P \simeq_{must} Q \quad \Longleftrightarrow \quad P \sqsubseteq_{must} Q \quad \wedge \quad Q \sqsubseteq_{must} P$$

Infine possiamo definire il *preordine di testing* e l'*equivalenza di testing* (o *testing equivalence*) tra due processi come:

$$P \sqsubseteq_{test} Q \quad \Longleftrightarrow \quad P \sqsubseteq_{may} Q \quad \wedge \quad P \sqsubseteq_{must} Q$$

$$P \simeq_{test} Q \quad \Longleftrightarrow \quad P \simeq_{may} Q \quad \wedge \quad P \simeq_{must} Q$$

La testing equivalence uguaglia tutti i processi che possono superare gli stessi test, ovvero soddisfare gli stessi osservatori, mentre nell'equivalenza per fallimenti avevamo che i processi venivano uguagliati in base ai test che *non* venivano superati. Come già detto all'inizio, queste due equivalenze sono identiche in CCS.

5.5 Equivalenze di bisimulazione

Introduciamo ora alcune delle più usate equivalenze per lo studio del comportamento di sistemi concorrenti: le equivalenze di *bisimulazione*. Questa nozione di equivalenza tra processi si basa, come le altre, sull'idea che vogliamo poter distinguere due processi solo se questa differenza può essere identificata da un processo esterno che interagisce con loro.

In questa parte daremo sono una prima introduzione al significato di questa relazione cercando di coglierne il senso: i concetti verranno poi ripresi in seguito in un discorso più ampio e completo.

Introdurremo prima una nozione in cui le azioni τ, quelle azioni che sono interne ai processi, vengono considerate esattamente come le altre azioni e questo ci porterà a definire una equivalenza abbastanza stretta che chiameremo *equivalenza forte* o *bisimulazione forte*. Noteremo poi che, dal momento che le azioni interne non possono essere osservate od individuate da un processo esterno, queste potranno non essere considerate; ciò ci porterà ad una nozione di equivalenza più debole che chiameremo *equivalenza osservazionale*

Il motivo per cui mostreremo prima l'equivalenza forte è dato dalla sua semplicità e dal fatto che possiede interessanti proprietà algebriche: infatti risulta essere una *congruenza*, cioè viene preservata da tutti gli operatori della nostra algebra.

Il nostro punto di partenza rimane sempre quello di ottenere un'equivalenza più forte di quella a traccie che, come ci ricordiamo, non è sufficiente a caratterizzare l'equivalenza fra processi in modo soddisfacente.

Inoltre, quello che la bisimulazione introduce e che la rende così importante è la valutazione dell'equivalenza fatta sia prima che dopo aver eseguito azioni comuni.

5.5.1 Bisimulazione ed equivalenza forte

Ciò che si vuole ottenere è una relazione di equivalenza che soddisfi la seguente proprietà:

> p e q sono equivalenti sse, per ogni azione α, ogni derivazione di p con α è equivalente a qualche derivazione di q con α, e viceversa.

con $p, q \in \mathcal{P}$, l'insieme dei processi.

Scrivendo in modo formale, utilizzando il simbolo $\sim$ per la nostra equivalenza, si ha:

p $\sim$ q sse, $\forall \alpha \in Act$ $\qquad (*_\sim)$

1. $\forall p' : p \xrightarrow{\alpha} p' \quad \exists q' : q \xrightarrow{\alpha} q' \quad \wedge \quad p' \sim q'$;
2. $\forall q' : q \xrightarrow{\alpha} q' \quad \exists p' : p \xrightarrow{\alpha} p' \quad \wedge \quad p' \sim q'$.

Purtroppo quanto appena scritto non è una definizione, in quanto esistono molte relazioni di equivalenza che soddisfano questa proprietà! Ci accontenteremmo di eguagliare più processi possibili, fintantoche la proprietà sopra è verificata: quello che allora stiamo cercando è la più grande (più debole o più generosa, in quanto identifica più processi) relazione $\sim$ che soddisfi questa proprietà. Per raggiungere questo obiettivo dovremo compiere alcuni passi intermedi, il primo dei quali è la definizione di bisimulazione forte:

Definizione 5.9 (Bisimulazione forte). *Una relazione binaria* $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ *è una* bisimulazione forte *se* $(p, q) \in \mathcal{S}$ *implica,* $\forall \alpha \in Act$*:*

- $p \xrightarrow{\alpha} p' \quad allora \quad \exists q' : q \xrightarrow{\alpha} q' \quad \wedge \quad (p', q') \in \mathcal{S}$
- $q \xrightarrow{\alpha} q' \quad allora \quad \exists p' : p \xrightarrow{\alpha} p' \quad \wedge \quad (p', q') \in \mathcal{S}$

Come si può notare, ogni relazione $\sim$ che soddisfa la proprietà $(*_{\sim})$, è anche una bisimulazione. La condizione della definizione, però, è più debole della condizione in $(*_{sim})$, in quanto al posto di 'sse' abbiamo 'implica', e dunque si hanno molte bisimulazioni forti: anche $\mathcal{S} = \emptyset$ è una bisimulazione forte!

Definiamo allora l'equivalenza forte:

Definizione 5.10 (Equivalenza forte). *p e q sono* fortemente equivalenti *oppure* fortemente bisimili*,* $p \sim q$*, se esiste una bisimulazione forte* $\mathcal{S}$ *tale che* $(p, q) \in \mathcal{S}$*. In modo equivalente:*

$$\sim = \bigcup \{\mathcal{S} \ \ tale\ che \ \ \mathcal{S} \ \ sia\ una\ bisimulazione\ forte\}$$

La relazione di equivalenza forte è la più grande bisimulazione forte, ovvero quella che contiene tutte le bisimulazioni forti.

L'importanza della bisimulazione forte è che permette di associare sistemi che, all'interno di uno stesso ambiente, si comportano *esattamente* nello stesso modo; ma se da un lato questo è un vantaggio, da un'altro è un vincolo troppo stretto dal momento che risulta essere fin troppo sensibile alle azioni interne dei processi: spesso non è necessario considerarle al pari delle azioni visibili. Per questo motivo vengono usate delle altre equivalenze, dette *deboli* in quanto le azioni invisibili non vengono considerate, e che risultano comunque di grande importanza.

Alcuni importanti risultati sono che:

$$\sim \implies \sim_t$$

$$\sim \implies \simeq_{test} \implies \approx_t$$

5.5.2 Bisimulazione ed equivalenza debole (osservazionale)

Abbiamo appena visto la nozione di bisimulazione forte, nella quale ogni azione α di un processo deve corrispondere ad un'azione α dell'altro, anche per le azioni τ. Quanto faremo adesso sarà rilassare questa richiesta ed ottenendo dunque una relazione più debole (in quanto identifica più processi): ciò che si richiede ora è che ogni azione τ venga corrisposta con zero o più azioni τ. Questo ci porterà alla definizione di *bisimulazione debole* e quindi alla definizione di una *equivalenza debole* o, come viene più spesso chiamata, dell'*equivalenza osservazionale*.

Procederemo in modo analogo a quanto fatto per l'equivalenza forte, rimandando al capitolo relativo alla bisimulazione la visione di una serie di importanti risultati.

Cerchiamo dunque una nozione di equivalenza, che chiameremo *equivalenza osservazionale*, che rispetti la seguente proprietà:

> p e q sono osservazionalmente equivalenti sse, per ogni azione α, ogni derivazione da p tramite α è osservazionalmente equivalente a qualche discendenza da q tramite l'azione α, e similmente per il viceversa.

Chiameremo questa equivalenza $\approx$ e formalmente scriviamo:

$$\text{p} \approx \text{q sse, } \forall \alpha \in Act \qquad (*_{\approx})$$

1. $\forall p' : p \xrightarrow{\alpha} p' \quad \exists q' : q \stackrel{\hat{\alpha}}{\Longrightarrow} q' \quad \wedge \quad p' \approx q';$
2. $\forall q' : q \xrightarrow{\alpha} q' \quad \exists p' : p \stackrel{\hat{\alpha}}{\Longrightarrow} p' \quad \wedge \quad p' \approx q'.$

dove

$$\stackrel{\hat{\alpha}}{\Rightarrow} = \begin{cases} \stackrel{\alpha}{\Rightarrow} & a \neq \tau \\ \stackrel{\varepsilon}{\Rightarrow} & a = \tau \end{cases}$$

Operando in modo analogo a quanto fatto per la strong bisimulation, cerchiamo ora la più grande relazione che soddisfi $(*_{\approx})$; procediamo dunque come già visto:

Definizione 5.11 (Bisimulazione debole). *Una relazione binaria* $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ *è una* bisimulazione debole *se* $(p, q) \in \mathcal{S}$ *implica,* $\forall \alpha \in Act$*:*

- $p \xrightarrow{\alpha} p' \quad$ *allora* $\quad \exists q' : q \stackrel{\hat{\alpha}}{\Longrightarrow} q' \quad \wedge \quad (p', q') \in \mathcal{S}$
- $q \xrightarrow{\alpha} q' \quad$ *allora* $\quad \exists p' : p \stackrel{\hat{\alpha}}{\Longrightarrow} p' \quad \wedge \quad (p', q') \in \mathcal{S}$

Possiamo ora definire l'equivalenza osservazionale, o bisimilarità:

Definizione 5.12 (Equivalenza osservazionale). *p e q sono* osservazionalmente equivalenti *oppure* bisimili (debolmente)*, $p \approx q$, se esiste una bisimulazione debole $\mathcal{S}$ tale che $(p, q) \in \mathcal{S}$. In modo equivalente:*

$$\approx = \bigcup \{\mathcal{S} \text{ tale che } \mathcal{S} \text{ sia una bisimulazione}\}$$

5.6 Simulazione e doppia simulazione

La nozione di preordine $\sqsubseteq$ introdotta nel capitolo 4 può anche essere intesa come 'meno definito di' nel senso che $p \sqsubseteq q$ si può leggere come 'p è meno definito di q', p può compiere meno azioni di q o, simmetricamente, q può fare almeno tante azioni quante ne fa p (tendenzialmente di più); si può anche intendere che p *è simulabile da* q:

Definizione 5.13 (Simulazione). *$p \sqsubseteq q$ sse $\forall \alpha \in Act_\tau^*$, se $p \xrightarrow{\alpha} p'$ allora per qualche q' si a che $q \xrightarrow{\alpha} q' \wedge p' \sqsubseteq q'$.*

$$P = \alpha.\beta.nil \quad Q = \alpha.nil + \alpha.\beta.nil \quad \Rightarrow \quad P \sqsubseteq Q \quad (\text{ma anche } Q \sqsubseteq P)$$

Naturalmente, essendo un preordine, della simulazione se ne può prendere il kernel ed ottenere la *doppia simulazione* $\simeq$, la quale però non coincide con la *bisimulazione* $\sim$, vediamo perchè con un esempio: prendendo sempre gli stessi processi definiti sopra, si vede che questi sono doppiamente simili ma non sono bisimili in quanto $P \xrightarrow{\alpha} nil \wedge Q \xrightarrow{\alpha} Q'$ ma $nil \not\sim Q'$

5.7 Bisimulazione di branching

Questa equivalenza si pone tra la weak e la strong bisimulation, in quanto da una parte tratta le azioni invisibili in modo diverso dalle azioni visibili, ma cambia il modo di considerare gli stati intermedi.L'idea di questa relazione è quella di rilassare le richieste della strong bisimulation in modo da non differenziare processi come $p = \alpha.\beta.nil$ e $q = \alpha.\tau.\tau.\beta.nil$: chiaramente si ha che $p \not\sim q$, ma possiamo notare che dopo che entrambi hanno eseguito l'azione α, i processi risultanti sono debolmente bisimili e le azioni τ nel mezzo non fanno cambiare classe di equivalenza agli stati intermedi. Possiamo vedere la bisimulazione di branching come il modo di uguagliare un processo che fa un azione ed un altro processo che fa tante azioni τ per poter giungere a poter fare la stessa azione passando per stati equivalenti.

Definizione 5.14 (Bisimulazione di branching). *Sia T un LTS e sia $\mathcal{R}$ una relazione sugli stati di T; $\mathcal{R}$ è una* bisimulazione di branching *su T se è simmetrica e soddisfa la seguente condizione: se $(r, s) \in \mathcal{R}$ e $r \xrightarrow{\alpha} r'$ allora implica*

- $\alpha = \tau \quad \wedge \quad (r', s) \in \mathcal{R}$ *oppure,*
- $\exists s', s'' : \; s \overset{\varepsilon}{\Longrightarrow} s' \overset{\alpha}{\longrightarrow} s'' \quad \wedge \quad (r, s') \in \mathcal{R} \quad \wedge \quad (r', s'') \in \mathcal{R}$

La bisimulazione di branching considera due sistemi equivalenti soltanto se ci sono le stesse traccie visibili ed in esse gli stati intermedi, anche quelli raggiunti attraverso azioni τ, hanno comportamento equivalente.

Definizione 5.15 (Equivalenza di branching). *Sia T un LTS e p, q appartengano agli stati di T, allora $p \approx_b q$ (p* bisimile di branching *a q, oppure p* branching equivalent *a q) se e solo se esiste una bisimulazione di branching $\mathcal{R}$ tale che $p\mathcal{R}q$ ($(p, q) \in \mathcal{R}$).*

Capitolo 6

La bisimulazione

Quanto ci proponiamo di fare in questo capitolo è riprendere il discorso iniziato nella sezione delle equivalenze, mostrando alcune proprietà della bisimulazione e fornendo anche un'assiomatizzazione completa sia per strong che per weak bisimulation; vedremo anche che per il nostro intento di sostituire uguali per uguali necessiteremo di aver a che fare con una congruenza e come la strong bisimulation abbia questa proprietà mentre per la weak dovremmo fornire una caratterizzazione alternativa (anche se molto simile) per ottenere questa proprietà.

6.1 Bisimulazione forte

La prima proprietà che vogliamo mostrare è come la strong bisimulation sia preservata da alcune operazioni sulle relazioni (teniamo a mente quanto visto in precedenza su relazioni inverse e composte):

Proposizione 6.1. *Si assuma che ogni* $\mathcal{S}_i$ $(i = 1, 2, \dots)$ *sia una bisimulazione forte, allora anche le seguenti relazioni sono bisimulazioni forti:*

1. $Id_{\mathcal{P}}$
2. $\mathcal{S}_i^{-1}$
3. $\mathcal{S}_1\mathcal{S}_2$
4. $\cup_{i\in I}\mathcal{S}_i$

Dimostrazione. Proveremo solo la 3, le altre sono altrettanto facili. Supponiamo dunque che

$$(p, r) \in \mathcal{S}_1\mathcal{S}_2$$

Per qualche q, avremo che

$$(p,q) \in \mathcal{S}_1 \quad \wedge \quad (q,r) \in \mathcal{S}_2$$

Sia adesso $p \xrightarrow{\alpha} p'$. Allora per qualche q' avremo, dal momento che $(p,q) \in \mathcal{S}_1$

$$q \xrightarrow{\alpha} q' \quad \wedge \quad (p',q') \in \mathcal{S}_1$$

Inoltre, poichè $(q,r) \in \mathcal{S}_2$ si ha, per qualche r'

$$r \xrightarrow{\alpha} r' \quad \wedge \quad (q',r') \in \mathcal{S}_1$$

Da questo si ottiene $(p',r') \in \mathcal{S}_1\mathcal{S}_2$. In modo del tutto simile si trova che se $r \xrightarrow{\alpha} r'$, allora possiamo trovare un p' tale che $p \xrightarrow{\alpha} p'$ e $(p',r') \in \mathcal{S}_1\mathcal{S}_2$. □

Dimostriamo due proprietà accennate in precedenza ma che non abbiamo provato:

Proposizione 6.2.

1. *$\sim$ è la più larga bisimulazione forte;*
2. *$\sim$ è una relazione di equivalenza.*

Dimostrazione. Dimostriamo i due asserti separatamente:

1. Dal punto 4. della proposizione 6.1 si vede come l'unione di bisimulazioni forti sia ancora una bisimulazione forte: dunque, anche $\sim$ lo è e comprende tutte le altre bisimulazioni.

2. Per provare che sia una relazione di equivalenza, cioè che sia riflessiva simmetrica e transitiva, mostriamo che valgono queste proprietà:

   Riflessività: $\forall p \quad p \sim p$ dal punto 1. della proposizione 6.1;

   Simmetria: se $p \sim q$ allora $(p,q) \in \mathcal{S}$ per qualche bisimulazione forte $\mathcal{S}$. Allora $(q,p) \in \mathcal{S}^{-1}$ e quindi $q \sim p$ per il punto 2. della proposizione 6.1;

   Transitività: se $p \sim q$ e se $q \sim r$ allora $(p,q) \in \mathcal{S}_1$ e $(q,r) \in \mathcal{S}_2$ per qualche bisimulazione forte $\mathcal{S}_1$ ed $\mathcal{S}_2$. Dunque $(p,r) \in \mathcal{S}_1\mathcal{S}_2$ e quindi $p \sim r$ per il punto 3. della proposizione 6.1.

□

Vogliamo dunque provare che $\sim$ soddisfa le proprietà che abbiamo chiamato $(*_\sim)$. Sappiamo che metà dell'implicazione vale, in quanto 'sse' è stato rimpiazzato da 'implica', dal momento che $\sim$ è una bisimulazione forte; rimane dunque da provare l'altra metà. Per fare questo, definiamo dapprima una nuova relazione, $\sim'$ in termini di $\sim$ nel modo seguente:

Definizione 6.1. *$p \sim' q$ sse, $\forall \alpha \in Act_\tau$,*

1. $\forall p' : p \xrightarrow{\alpha} p' \quad \exists q' : q \xrightarrow{\alpha} q' \quad \wedge \quad p' \sim q'$
2. $\forall q' : q \xrightarrow{\alpha} q' \quad \exists p' : p \xrightarrow{\alpha} p' \quad \wedge \quad p' \sim q'.$

Dalla proposizione 6.2, punto 1., sappiamo che $\sim$ è una bisimulazione forte, possiamo allora dedurre che

$$p \sim q \quad \text{implica} \quad p \sim' q \qquad (\#)$$

Rimane da provare il viceversa, cioè che $p \sim' q$ implica $p \sim q$. Il seguente risultato è sufficiente:

Lemma 6.1. *La relazione $\sim'$ è una bisimulazione forte.*

Dimostrazione. Sia $p \sim' q$ e $p \xrightarrow{\alpha} p'$. E' sufficiente trovare un q' tale che $q \xrightarrow{\alpha} q'$ e $p' \sim' q'$. Ma per la definizione di $\sim'$ possiamo infatti trovare questo q' tale che $q \xrightarrow{\alpha} q'$ e $p' \sim q'$. Grazie a $(\#)$ sappiamo allora che $p' \sim' q'$, e con questo si conclude la prova. □

Si è quindi dimostrato che $\sim$ soddisfa $(*_\sim)$:

Proposizione 6.3. *$p \sim q$ sse, $\forall \alpha \in Act_\tau$*

- $p \xrightarrow{\alpha} p' \quad$ *allora* $\quad \exists q' : q \xrightarrow{\alpha} q' \quad \wedge \quad p' \sim q'$
- $q \xrightarrow{\alpha} q' \quad$ *allora* $\quad \exists p' : p \xrightarrow{\alpha} p' \quad \wedge \quad p' \sim q'$

Quello che vogliamo fare è provare che molte delle leggi di inferenza introdotte per CCS possono essere provare vere se si interpreta '=' come la bisimulazione forte; quanto proveremo adesso risulta valido anche per la nozione di bisimulazione debole (più generosa). In ogni caso, le τ laws non sono valide per la bisimulazione forte ma soltanto per la nozione debole.

Proposizione 6.4. *Le monoid laws:*

1. $p + q \sim q + p$
2. $p + (q + r) \sim (p + q) + r$
3. $p + p \sim p$

4. $p + nil \sim p$

Dimostrazione. Dimostreremo solo il punto 2.; gli altri seguono procedimenti simili. Supponiamo che

$$p + (q + r) \xrightarrow{\alpha} p'$$

Allora dalla semantica della regola Somma sia ha che $p \xrightarrow{\alpha} p'$ oppure $q \xrightarrow{\alpha} p'$ oppure $r \xrightarrow{\alpha} p'$. In ognuno dei casi possiamo facilmente inferire da Somma che $(p+q)+r \xrightarrow{\alpha} p'$. Questo dimostra la proprietà (i) di $(*_\sim)$, il punto (ii) si dimostra in modo simile. □

Proposizione 6.5. *Le static laws:*

1. $p|q \sim q|p$
2. $p|(q|r) \sim (p|q)|r$
3. $p|nil \sim p$
4. $p \setminus L \sim p \quad se\, \mathcal{L}(p) \cap (L \cup \overline{L}) = \emptyset$
5. $p \setminus K \setminus L \sim p \setminus (K \cup L)$
6. $p[f] \setminus L \sim p \setminus f^{-1}(L)[f]$
7. $(p|q) \setminus L \sim p \setminus L | q \setminus L \quad if\, \mathcal{L}(p) \cap \overline{\mathcal{L}(q)} \cap (L \cup \overline{L}) = \emptyset$
8. $p[Id] \sim p$
9. $p[f] \sim p[f']$
10. $p[f][f'] \sim p[f \circ f']$
11. $(p|q)[f] \sim p[f]|q[f]$

Dimostrazione. Tutte queste leggi possono essere dimostrate esibendo un'appropriata bisimulazione forte. La più difficile è la 2. e per questo la analizziamo subito.

Per questo punto, abbiamo bisogno di mostrare che $\mathcal{S}$ è una bisimulazione forte, dove

$$\mathcal{S} = \{(\, p_1|(p_2|p_3), (p_1|p_2)|p_3\,) \,:\, p_1, p_2, p_3 \in \mathcal{P}\}$$

Supponiamo ora che $p_1|(p_2|p_3) \xrightarrow{\alpha} q$. Ci sono tre casi, con relativi sottocasi, che possono verificarsi:

Caso 1 $p_1 \xrightarrow{\alpha} p_1'$, e $q \equiv p_1'|(p_2|p_3)$.
E' facile allora mostrare che $(p_1|p_2)|p_3 \xrightarrow{\alpha} r \equiv (p_1'|p_2)|p_3$, e chiaramente $(q, r) \in \mathcal{S}$.

Caso 2 $p_2|p_3 \xrightarrow{\alpha} p'_{23}$ e $q \equiv p_1|p'_{23}$

Caso 2.1 $p_2 \xrightarrow{\alpha} p'_2$ e $p'_{23} \equiv p'_2|p_3$
Allora $q \equiv p_1|(p'_2|p_3)$, ed è facile vedere che $(p_1|p_2)|p_3 \xrightarrow{\alpha} r \equiv (p_1|p'_2)|p_3$, e chiaramente $(q, r) \in \mathcal{S}$.

Caso 2.2 $p_3 \xrightarrow{\alpha} p'_3$ e $p'_{23} \equiv p_2|p'_3$
In modo simile al Caso 2.1.

Caso 2.3 $\alpha = \tau, p_2 \xrightarrow{l} p'_2 \ \wedge \ p_3 \xrightarrow{\overline{l}} p'_3$ e $p'_{23} \equiv p'_2|p'_3$
Allora $q \equiv p_1|(p'_2|p'_3)$, ed è facile vedere che $(p_1|p_2)|p_3 \xrightarrow{\tau} r \equiv (p_1|p'_2)|p'_3$, e chiaramente $(q, r) \in \mathcal{S}$

Caso 3 $\alpha = \tau, p_1 \xrightarrow{l} p'_1 \ \wedge \ p_2|p_3 \xrightarrow{\overline{l}} p'_{23}$ e $q \equiv p'_1|p'_{23}$

Caso 3.1 $p_2 \xrightarrow{\overline{l}} p'_2$ e $p'_{23} \equiv p'_2|p_3$
Allora $q \equiv p'_1|(p'_2|p_3)$, ed è facile vedere che $(p_1|p_2)|p_3 \xrightarrow{\tau} r \equiv (p'_1|p'_2)|p_3$, e chiaramente $(q, r) \in \mathcal{S}$

Caso 3.2 $p_3 \xrightarrow{\overline{l}} p'_3$ e $p'_{23} \equiv p_2|p'_3$
In modo simile al Caso 3.1

Questo prova la condizione 1. della definizione 5.9; la condizione 2. segue da un'argomentazione simmetrica, ed abbiamo così mostrato che $\mathcal{S}$ è una strong bisimulation.

Le altre parti della proposizione sono trattate in maniera simile. Per 4. si deve provare che, per un fissato L, la relazione

$$\mathcal{S} = \{(P \setminus L, P) \ : \ p \in \mathcal{P}, \mathcal{L}(p) \cap (L \cup \overline{L}) = \emptyset\}$$

è una bisimulazione forte. Ora, se $(P \setminus L, P) \in \mathcal{S}$, allora (per la condizione sopra) $p \xrightarrow{\alpha} p'$ implica che $\alpha, \overline{\alpha} \notin L$; rimane da mostrare che $(p' \setminus L, p') \in \mathcal{S}$. Ma $\mathcal{L}(p') \subseteq \mathcal{L}(p)$ per ogni p' derivazione di p, così $\mathcal{L}(p') \cap (L \cup \overline{L}) = \emptyset$, e di conseguenza $(p' \setminus L, p') \in \mathcal{S}$ come richiesto.

Le restanti parti non introducono altre problematiche.

□

Proposizione 6.6 (Expansion law). *Sia* $p \equiv (p_1[f_1]| \ldots |p_n[f_n]) \setminus L$, *con* $n \geq 1$. *Allora*

$$\begin{aligned} p \ \sim \ & \sum \{f_i(\alpha).(p_1[f_1]| \ldots |p'_i[f_i]| \ldots |p_n[f_n]) \setminus L : \\ & \quad p_i \xrightarrow{\alpha} p'_i, \ f_i(\alpha) \notin L \cup \overline{L}\Big\} \\ + \ & \sum \Big\{\tau.(p_1[f_1]| \ldots |p'_i[f_i]| \ldots |p'_j[f_j]| \ldots |p_n[f_n]) \setminus L : \\ & \quad p_i \xrightarrow{l_1} p'_i, \ p_j \xrightarrow{l_2} p'_j, \ f_i(l_1) = \overline{f_j(l_2)}, \ i < j\Big\} \end{aligned}$$

Dimostrazione. Consideriamo dapprima il caso semplice nel quale non sono presenti Restrizioni e Relabelling. In fatti, possiamo provare quanto segue per induzione su n:

Se $p \equiv p_1|\ldots|p_n,\ n \geq 1$, allora $\qquad (*_+)$

$$\begin{array}{rcl} p & \sim & \sum\left\{\alpha.(p_1|\ldots|p'_i|\ldots|p_n)\ :\ 1 \leq i \leq n,\ p_i \xrightarrow{\alpha} p'_i\right\} \\ & + & \sum\left\{\tau.(p_1|\ldots|p'_i|\ldots|p'_j]|\ldots|p_n)\ : \right. \\ & & \left. \quad 1 \leq i < j \leq n,\ p_i \xrightarrow{l} p'_i,\ p_j \xrightarrow{\overline{l}} p'_j\right\} \end{array}$$

Per $n = 1$ ci si riduce a provare che $p_1 \sim \sum\{\alpha.p'_1\ :\ p_1 \xrightarrow{\alpha} p'_1\}$, il che è immediato. Assumiamo vero il risultato per n e consideriamo $r \equiv p|p_{n+1}$. E' immediato, dall'applicazione delle regole di Composizione parallela che

$$\begin{array}{rcl} r & \sim & \sum\{\alpha.(p'|p_{n+1})\ :\ p \xrightarrow{\alpha} p'\} \\ & + & \sum\{\alpha.(p|p'_{n+1})\ :\ p_{n+1} \xrightarrow{\alpha} p'_{n+1}\} \\ & + & \sum\{\tau.(p'|p'_{n+1})\ :\ p \xrightarrow{l} p'\ \wedge\ p_{n+1} \xrightarrow{\overline{l}} p'_{n+1}\} \end{array}$$

Usiamo ora l'assunzione iniziale per $p \equiv p_1|\ldots|p_n,\ n \geq 1$, allora la parte destra può essere riformulata ne modo seguente (si noti che la prima sommatoria può essere divisa in due parti, a seconda che $p \xrightarrow{\alpha} p'$ sia dovuto ad un singolo p_i oppure all'interazione tra p_i e p_j):

$$\begin{array}{rcl} & & \sum\left\{\alpha.(p_1|\ldots|p'_i|\ldots|p_n|p_{n+1})\ :\ 1 \leq i \leq n,\ p \xrightarrow{\alpha} p'\right\} \\ & + & \sum\left\{\tau.(p_1|\ldots|p'_i|\ldots|p'_j|\ldots|p_n|p_{n+1})\ : \right. \\ & & \left. \quad 1 \leq i < j \leq n,\ p_i \xrightarrow{l} p'_i\ \wedge\ p_j \xrightarrow{\overline{l}} p'_j\right\} \\ & + & \sum\left\{\alpha.(p_1|\ldots|p_n|p'_{n+1})\ :\ p_{n+1} \xrightarrow{\alpha} p'_{n+1}\right\} \\ & + & \sum\{\tau.(p_1|\ldots|p'_i|\ldots|p_n|p_{n+1})\ : \\ & & \left. \quad 1 \leq i \leq n,\ p_i \xrightarrow{l} p'_i\ \wedge\ p_{n+1} \xrightarrow{\overline{l}} p'_{n+1}\right\} \end{array}$$

Possiamo allora ricombinare la prima con la terza sommatoria e la seconda con la quarta per ottenere quanto si cercava:

$$\begin{array}{rcl} r & \sim & \sum\left\{\alpha.(p_1|\ldots|p'_i|\ldots|p_{n+1})\ :\ 1 \leq i \leq n+1,\ p_i \xrightarrow{\alpha} p'_i\right\} \\ & + & \sum\left\{\tau.(p_1|\ldots|p'_i|\ldots|p'_j|\ldots|p_{n+1})\ : \right. \\ & & \left. \quad 1 \leq i < j \leq n+1,\ p_i \xrightarrow{l} p'_i,\ p_j \xrightarrow{\overline{l}} p'_j\right\} \end{array}$$

Possiamo dunque estendere $(*_+)$ per provare il teorema nella forma in cui è stato enunciato. Possiamo introdurre per prima il Relabelling, considerando $p_i \equiv q_i[f_i]$ in $(*_+)$, ed osservando che p_i ha una transizione $p_i \xrightarrow{\alpha} p'_i$ sse

q_i ha una transizione $q_i \xrightarrow{\beta} q_i'$ tale che $\alpha = f_i(\beta)$ e $p_i' = q_i'[f_i]$. Possiamo infine introdurre la Restrizione, usando l'equivalenza forte

$$q \setminus L \sim \sum\{\beta.(q' \setminus L) : q_i \xrightarrow{\beta} q_i', \beta \notin L \cup \overline{L}\}$$

dove $q \equiv q_1[f_1]|\ldots|q_n[f_n]$.

□

Abbiamo quindi dimostrato tutte le leggi viste sostitudendo al simbolo '=' quello di bisimulazione forte '$\sim$'. Ciò che adesso vogliamo fare è provare che $\sim$ è una relazione di congruenza, cioè che se due processi sono bisimili fortemente, allora dove ce n'è uno si può sostituire l'altro.

6.1.1 Congruenza forte

Ora vogliamo mostrare che la bisimualzione forte è una congruenza

Dimostriamo dunque che la bisimulazione forte è sostituitiva per tutti i combinatori:

Proposizione 6.7. *Sia $p_1 \sim p_2$. Allora*

1. $\alpha.p_1 \sim \alpha.p_2$
2. $p_1 + q \sim p_2 + q$
3. $p_1|q \sim p_2|q$
4. $p_1 \setminus L \sim p_2 \setminus L$
5. $p_1[f] \sim p_2[f]$

Dimostrazione. Per 1., possiamo facilmente dedurre le condizioni 1. e 2. della proposizione 6.3 sostituendo $\alpha.p$ e $\alpha.q$ al posto di p e q.

La prova per 2. è simile.

Per 3., invece, si deve mostrare che $\mathcal{S}$ è una bisimulazione, dove

$$\mathcal{S} = \{(p_1|q, p_2|q) : p_1 \sim p_2\}$$

Supponiamo allora che $(p_1|q, p_2|q) \in \mathcal{S}$. Sia $p_1|q \xrightarrow{\alpha} r$. Ci sono tre casi possibili:

Caso 1 $p_1 \xrightarrow{\alpha} p_1'$ e $r \equiv p_1'|q$

Allora, dal momento che $p_1 \sim p_2$, abbiamo che $p_2 \xrightarrow{\alpha} p_2'$ con $p_1' \sim p_2'$; da questo si ha $p_2|q \xrightarrow{\alpha} p_2'|q$ e $(p_1'|q, p_2'|q) \in \mathcal{S}$.

Caso 2 $q \xrightarrow{\alpha} q'$ e $r \equiv p_1|q'$

Allora anche $p_2|q \xrightarrow{\alpha} p_2|q'$ e $(p_1|q', p_2|q') \in \mathcal{S}$.

Caso 3 $\alpha = \tau, p_1 \xrightarrow{l} p_1' \ \wedge \ q \xrightarrow{\overline{l}} q'$ e $r \equiv p_1'|q_1$

Allora, dal momento che $p_1 \sim p_2$, abbiamo che $p_2 \xrightarrow{l} p_2'$ con $p_1' \sim p_2'$; da questo si ha $p_2|q \xrightarrow{\tau} p_2'|q'$ e $(p_1'|q', p_2'|q') \in \mathcal{S}$.

Con un argomento simmetrico, si completa la prova che $\mathcal{S}$ è una bisimulazione forte.

Le prove per 4. e 5. sono simili.

□

6.2 Bisimulazione debole

Come fatto per la bisimulazione forte, intendiamo ora dimostrare proprietà analoghe anche per la bisimulazione debole.

Proposizione 6.8. *Si assuma che ogni* $\mathcal{S}_i$ $(i = 1, 2, \dots)$ *sia una bisimulazione, allora anche le seguenti relazioni sono bisimulazioni:*

1. $Id_{\mathcal{P}}$
2. $\mathcal{S}_i^{-1}$
3. $\mathcal{S}_1\mathcal{S}_2$
4. $\cup_{i\in I}\mathcal{S}_i$

Dimostrazione. La dimostrazione si svolge in modo simile a quanto fatto in precedenza (proposizione 6.1); nel punto 3. abbiamo però bisogno del risultato che se $(q, r) \in \mathcal{S}_i$ e $q \stackrel{\hat{\alpha}}{\Longrightarrow} q'$ allora, per qualche r', $r \stackrel{\hat{\alpha}}{\Longrightarrow} r'$ e $(q', r') \in \mathcal{S}_i$. □

Proposizione 6.9.

1. $\approx$ *è la più larga bisimulazione;*
2. $\approx$ *è una relazione di equivalenza.*

Dimostrazione. Anche in questo caso, la dimostrazione segue la linea della proposizione 6.2, solo che ci si deve basare sul risultato ottenuto sopra. □

Continuando con l'analogia, vogliamo dunque provare che $\approx$ soddisfa $(*_{\approx})$: definiamo dunque una nuova relazione, $\approx'$ in termini di $\approx$ nel modo seguente:

Definizione 6.2. $p \approx' q$ *sse,* $\forall \alpha \in Act_\tau$,

1. $\forall p' : p \xrightarrow{\alpha} p' \quad \exists q' : q \stackrel{\hat{\alpha}}{\Longrightarrow} q' \quad \wedge \quad p' \approx q'$

2. $\forall q' : q \xrightarrow{\alpha} q' \quad \exists p' : p \stackrel{\hat{\alpha}}{\Longrightarrow} p' \quad \wedge \quad p' \approx q'.$

Dalla proposizione 6.2, punto 1., sappiamo che $\approx$ è una bisimulazione debole, e possiamo dedurre similmente che $p \approx q$ implica $p \approx' q$, rimane da provare che $p \approx' q$ implica $p \approx q$:

Lemma 6.2. *La relazione* $\approx'$ *è una bisimulazione debole.*

Dimostrazione. Analogo al lemma 6.1. □

Abbiamo dunque dimostrato che $\approx$ soddisfa $(*_{\approx})$, cioè che:

Proposizione 6.10. $p \approx q$ *sse,* $\forall \alpha \in Act_\tau$

- $p \xrightarrow{\alpha} p' \quad$ *allora* $\quad \exists q' : q \stackrel{\hat{\alpha}}{\Longrightarrow} q' \quad \wedge \quad p' \approx q'$
- $q \xrightarrow{\alpha} q' \quad$ *allora* $\quad \exists p' : p \stackrel{\hat{\alpha}}{\Longrightarrow} p' \quad \wedge \quad p' \approx q'$

Presentiamo ancora ulteriori proprietà della bisimulazione, e guardiamo proprio al caso emblematico che la distingue da $\sim$:

Proposizione 6.11. $p \approx \tau.p$

Dimostrazione. Dal momento che abbiamo mostrato l'uguaglianza di $\approx$ e $\approx'$, proveremo $p \approx' \tau.p$. Da prima consideriamo ogni azione $p \xrightarrow{\alpha} p'$ di p: chiaramente $\tau.p \xrightarrow{\tau} p \xrightarrow{\alpha} p'$, quindi $\tau.p \stackrel{\hat{\alpha}}{\Longrightarrow} p'$, e sappiamo che $p' \approx p'$. D'altra parte, consideriamo la sola azione $\tau.p \xrightarrow{\tau} p$ di $\tau.p$: questa è chiaramente corrisposta dall'azione nulla $p \stackrel{\varepsilon}{\Longrightarrow} p$ di p, poichè $\hat{\tau} = \varepsilon$. Abbiamo dunque che $p \approx' \tau.p$ e di conseguenza $p \approx \tau.p$. □

Quello che abbiamo dimostrato è proprio la potenza della bisimulazione: consente di ignorare le azioni τ durante l'investigazione sulla bisimilarità! Purtroppo però, proprio a causa della prelazione dell'azione τ, la bisimulazione non è preservata dalla Somma, come si vede da questo esempio:

$$b.nil \approx \tau.b.nil$$

$$a.nil + b.nil \not\approx a.nil + \tau.b.nil$$

Per questo, la nozione di '$\approx$' non è quella di uguaglianza '=', ma le due nozioni sono molto vicine: vedremo in seguito come un modifica minima nella definizione di $\approx$ li renda concetti equivalenti.

Definizione 6.3. *p è stabile se non ha derivazioni* τ.

Proposizione 6.12. *Se* $p \approx q$ *e sono entrambi stabili, allora* $p = q$.

Da questo risultato, sembra che la differenza tra bisimulazione ed uguaglianza stia esclusivamente nella presenza di azioni τ iniziali. Il seguente risultato lo evidenzia ancora maggiormente:

Proposizione 6.13. *Se $p \approx q$ allora $\alpha.p = \alpha.q$.*

Questo implica che la bisimulazione viene preservata dalla Sequenzalizzazione (in questo caso è addirittura rafforzata!) e da tutti gli altri operatori ad eccezione della Somma.

6.2.1 Congruenza debole

Ciò che si vuole individuare è la più grande relazione di congruenza che includa $\approx$. Per fare questo, inizieremo definendo in modo differente la nozione di bisimulazione e proveremo che la bisimulazione è preservata per tutti i combinatori ad eccezione della Somma. Daremo infine la definizione di uguaglianza '=' e vedremo che questa è effettivamente una relazione di congruenza, che giustifica tutte le normali manipolazioni algebriche (la sostituzione di uguali per uguali). Mostreremo come le τ laws valgano ed infine mostreremo un insieme di assiomi che sono completi (dai quali tutte le equazioni valide possono essere derivate) per i processi finiti ed a stati finiti.

Abbiamo visto in precedenza il concetto di bisimulazione, ed in particolare la più larga relazione di bisimulazione $\approx$, che abbiamo chiamato equivalenza osservazionale o bisimiliarità. Iniziamo ora a dare una caratterizzazione alternativa di bisimulazione in termini di *esperimenti*. Consideriamo $p \stackrel{s}{\Longrightarrow} p'$, dove $s \in \mathcal{L}^*$, un esperimento su p: consiste nell'eseguire p per un po' ed osservare la sequenza $s = l_1 \dots l_n$.Se $s = \varepsilon$, scriveremo $p \Rightarrow p'$ (se, cioè, p non è stabile, ciò può essere dovuto ad una o più azioni τ). $p \Rightarrow p'$ è comunque un esperimento, dal momento che p' ammette meno esperimenti di p: ad esempio, $p \equiv a.nil + \tau.nil$ ammette l'esperimento a, ma possiamo fare $p \Rightarrow nil$ e quest'ultimo non ammette un esperimento a.

Proposizione 6.14. *$\mathcal{S}$ è una bisimulazione sse, $\forall (p,q) \in \mathcal{S}$ e $s \in \mathcal{L}^*$:*

1. $\forall p' : p \stackrel{s}{\Longrightarrow} p' \quad \exists q' : q \stackrel{s}{\Longrightarrow} q' \quad \wedge \quad p' \,\mathcal{S}\, q'$;
2. $\forall q' : q \stackrel{s}{\Longrightarrow} q' \quad \exists p' : p \stackrel{s}{\Longrightarrow} p' \quad \wedge \quad p' \,\mathcal{S}\, q'$.

Dimostrazione. ($\Rightarrow$) Assumiamo che $\mathcal{S}$ sia una bisimulazione. Sia $p \stackrel{s}{\Longrightarrow} p'$, così che $p \stackrel{t}{\longrightarrow} p'$ dove $t = (\alpha_1 \dots \alpha_n) \in Act^*_{tau}$ e $\hat{t} = s$. Allora $p \stackrel{\alpha_1}{\longrightarrow} p_1 \stackrel{\alpha_2}{\longrightarrow} \cdots \stackrel{\alpha_n}{\longrightarrow} p'$ e dal momento che $\mathcal{S}$ è una bisimulazione abbiamo anche che $q \stackrel{\hat{\alpha}_1}{\Longrightarrow} q_1 \stackrel{\hat{\alpha}_2}{\Longrightarrow} \cdots \stackrel{\hat{\alpha}_n}{\Longrightarrow} q'$ con $p' \,\mathcal{S}\, q'$. Da questo $q \stackrel{\hat{t}}{\Longrightarrow} q'$, e quindi $q \stackrel{s}{\Longrightarrow} q'$.

($\Leftarrow$) Assumendo 1., sia $p \stackrel{\alpha}{\longrightarrow} p'$. Se $\alpha = \tau$ allora $p \stackrel{\varepsilon}{\Longrightarrow} p'$ e così da 1. si ha $q \stackrel{\varepsilon}{\Longrightarrow} q'$ con $p' \mathcal{S} q'$; ma $\hat{\alpha} = \varepsilon$, quindi $q \stackrel{\hat{\alpha}}{\Longrightarrow} q'$ come richiesto. Se $\alpha = l$ allora $p \stackrel{l}{\Longrightarrow} p'$, e grazie a 1. $q \stackrel{l}{\Longrightarrow} q'$ con $p' \mathcal{S} q'$; ma $\hat{l} = l$ e quindi $q \stackrel{\hat{l}}{\Longrightarrow} q'$ come richiesto. Assumendo 2., similmente si deriva la seconda condizione per la bisimulazione.

□

Il motivo per includere l'esperimento vuoto $p \Rightarrow p'$ nella proposizione di sopra è che, senza di esso, avremmo che $a.nil + \tau.nil \approx a.nil$ certamente falso nella nostra definizione originaria.

Guardiamo ora alla sostitutività di $\approx$. Abbiamo già accennato al fatto che, in generale, $\approx$ non è preservata da Somma:

$$b.nil \approx \tau.b.nil$$

mentre

$$a.nil + b.nil \not\approx a.nil + \tau.b.nil$$

D'altro canto, abbiamo che Prefix rafforza la bisimilarità verso l'uguaglianza; per ora ci limitiamo a dimostarare che:

Proposizione 6.15. *La bisimilarità è preservata da Composizione parallela, Restrizione e Relabelling: se $p \approx q$ allora $p|r \approx q|r$, $p\backslash L \approx q\backslash L$ e $p[S] \approx q[S]$.*

Dimostrazione. Proveremo solo il risultato per la Composizione parallela, il metodo per gli altri due è semplice e segue la stessa strada. Ci è sufficiente mostrare che

$$\mathcal{S} = \{(p|r, q|r) \ : \ p, q, r \in \mathcal{P} \ \wedge \ p \approx q\}$$

è una bisimulazione. Sia allora $p|r \xrightarrow{\alpha} p'|r'$ (dal momento che tutte le derivazioni devono avere questa forma).

Caso 1: $p \xrightarrow{\alpha} p' \ \wedge r \equiv r'$
Allora $q \stackrel{\hat{\alpha}}{\Longrightarrow} q'$ per qualche q' con $p' \approx q'$, quindi $q|r \stackrel{\hat{\alpha}}{\Longrightarrow} q'|r$ e dunque $(p|r, q|r) \in \mathcal{S}$ come richiesto.

Caso 2: $r \xrightarrow{\alpha} r' \ \wedge p \equiv p'$
Allora anche $q|r \xrightarrow{\alpha} q|r'$, e $(p|r', q|r') \in \mathcal{S}$ come richiesto.

Caso 3: $\alpha = \tau,\ p \xrightarrow{l} p' \ \wedge r \xrightarrow{\overline{l}} r'$
Allora $q \stackrel{l}{\Longrightarrow} q'$ per qualche q' con $p' \approx q'$, quindi $q|r \stackrel{\tau}{\Longrightarrow} q'|r'$ e dunque $(p'|r', q'|r') \in \mathcal{S}$ come richiesto.

Da questo, tramite un argomento simmetrico, si ottiene che $\mathcal{S}$ è una bisimulazione. □

Quanto detto vale poichè l'azione invisibile τ non può mai essere ristretta o rinominata.

Come appena mostrato, la bisimulazione non è completamente sostitutiva ($p \approx q$ non implica $p + r \approx q + r$), mentre noi cerchiamo una nozione di uguaglianza, $p = q$, che implichi $p \approx q$ e che sia completamente sostitutiva: cerchiamo una relazione di congruenza, che sia la più larga relazione a contenere $\approx$.

Definizione 6.4. *p e q sono* uguali *o* (osservazionalmente) congruenti*, $p = q$, se* $\forall \alpha$

1. $\forall p' : p \xrightarrow{\alpha} p' \quad \exists q' : q \stackrel{\alpha}{\Longrightarrow} q' \quad \wedge \quad p' \approx q'$;
2. $\forall q' : q \xrightarrow{\alpha} q' \quad \exists p' : p \stackrel{\alpha}{\Longrightarrow} p' \quad \wedge \quad p' \approx q'$.

Si noti la somiglianza con la definizione di bisimulazione: infatti, differiscono soltanto in un aspetto, e cioè qui abbiamo $\stackrel{\alpha}{\Rightarrow}$ al posto di $\stackrel{\hat{\alpha}}{\Rightarrow}$. Questo implica che ogni azione di p o q deve essere corrisposta da almeno una azione dell'altro processo. Si noti inoltre come questo si applichi solo all'azione iniziale, in quanto poi si richiede che $p' \approx q'$ e non $p' = q'$.

Guardiamo un risultato interessante, che ci mostra come le nozioni di uguaglianza e bisimilarità siano molto vicine (l'ipotesi che faremo è che si possa sempre costruire un nuovo nome indipendentemente da quanti se ne sono usati finora indicando con $\mathcal{L}(p)$ il linguaggio generato da p e con $\mathcal{L}$ il linguaggio di tutta la nostra algebra)

Proposizione 6.16. *Assumiamo che $\mathcal{L}(p) \cup \mathcal{L}(q) \subset \mathcal{L}$. Allora $p = q$ sse, per ogni r, $p + r \approx q + r$.*

Dimostrazione. ($\Rightarrow$) E' facile vedere che la seguente è una bisimulazione:

$$\{(p + r, q + r) \ : \ p, q, r \in \mathcal{P} \text{ e } p = q\} \cup \approx$$

($\Leftarrow$) Proviamo il contropositivo. Supponiamo dunque che $p \neq q$. Allora, per esempio, avremo che ci sono α e p' tali che $p \xrightarrow{\alpha} p'$ ma che per qualsiasi $q \stackrel{\alpha}{\Longrightarrow} q'$ si ha $p' \not\approx q'$. Scegliamo ora $r \equiv l.nil$, dove l non è nel linguaggio di p o di q. Chiaramente $p + r \xrightarrow{\alpha} p'$; si deve allora mostrare che qualsiasi $q + r \stackrel{\hat{\alpha}}{\Longrightarrow} q'$ si ha $p' \not\approx q'$. Se $\alpha = \tau$ e $q \equiv q + r$, allora $p' \not\approx q'$ in quanto q' ha un'azione l che p non ha; d'altra parte $q + r \stackrel{\alpha}{\Longrightarrow} q'$ e quindi $q \stackrel{\alpha}{\Longrightarrow} q'$ (in quanto $r \stackrel{\alpha}{\Longrightarrow} q'$ è impossibile dal momento che $\alpha \neq l$), e quindi ancora $p' \not\approx q'$.

□

Vediamo ora come l'uguaglianza si trovi nel mezzo tra congruenza forte e bisimilarità:

Proposizione 6.17. *$p \sim q$ implica $p = q$, e $p = q$ implica $p \approx q$.*

Dimostrazione. Come prima cosa notiamo come $p \sim q$ implica $p \approx q$: discende direttamente dal fatto che ogni bisimulazione forte è anche una bisimulazione, facilmente verificabile dalle definizioni. Ora, per vedere che $p \sim q$ implica $p = q$, si usa la proprietà ($*_\sim$) della bisimulazione forte unita con la definizione di uguaglianza data poco sopra.

Per quanto riguarda la seconda parte, dalla proprosizione sopra abbiamo che $p = q$ implica $p + nil \approx q + nil$, ma $p + nil \sim p$ e quindi $p + nil \approx p$, da cui segue $p \approx q$. □

Quindi, tutte le leggi valide per $\sim$ e $=$ lo sono anche per la bisimilarità, mentre non vale il viceversa: abbiamo che $p \approx \tau.p$ mentre in generale abbiamo $p \neq \tau.p$.

Facciamo il primo passo per dimostrare che l'uguaglianza è una congruenza mostrando che:

Proposizione 6.18. *L'uguaglianza è una relazione di equivalenza.*

Dimostrazione. Il modo più semplice per provare questo risultato è sfruttare la proposizione 6.16 assieme al fatto che $\approx$ è un'equivalenza. □

Vediamo come, attraverso la Sequenzalizzazione, si rafforzi la bisimilarità verso la uguaglianza, già osservato nella sezione precedente:

Proposizione 6.19. *Se $p \sim q$ allora $\alpha.p = \alpha.q$.*

Dimostrazione. Discende direttamente dalla definizione di uguaglianza. □

Si deve ora provare che tutti gli operatori preservano l'ugaglianza:

Proposizione 6.20. *se $p = q$ allora $\alpha.p = \alpha.q$, $p + r = q + r$, $p|r = q|r$, $p \setminus L = q \setminus L$ e $p[f] = q[f]$.*

Dimostrazione. La prima, discende direttamente dalla proposizione precedente. Per la seconda, si richiede che $(p + r) + r' \approx (q + r) + r'$, per ogni r'; sappiamo però che $p + (r + r') \approx q + (r + r')$, dato che $p = q$ e che l'associatività, in quanto vale per $\sim$, vale anche per $\approx$.

Le rimanenti prove sono applicazioni dirette della definizione di uguaglianza. Per la Composizione parallela, un'analisi a casi come nella proposizione 6.15 è necessaria; le altre due sono più dirette. □

Raccogliendo tutti i risultati ottenuti si arriva a poter affermare che l'uguaglianza è una congruenza.

Avevamo lasciato in sospeso alcune proprietà, che ora riprendiamo dimostrando:

Proposizione 6.21. *Le τ laws:*

1. $\alpha.\tau.p = \alpha.p$
2. $P + \tau.p = \tau.p$
3. $\alpha.(p + \tau.q) + \alpha.q = \alpha.(p + \tau.q)$

Dimostrazione. In modo diretto dalla definizione di uguaglianza. Soltanto nel punto 1. si è usato il risultato che $p \approx \tau.p$. □

Mostriamo anche che

Proposizione 6.22. *Se $p \approx q$ e sono entrambi stabili, allora $p = q$.*

Dimostrazione. Sia $p \xrightarrow{\alpha} p'$, il che implica che $\alpha \neq \tau$ (per la stabilità); allora $q \stackrel{\hat{\alpha}}{\Longrightarrow} q'$ con $p' \approx q'$. Ma allora $q \stackrel{\alpha}{\Longrightarrow} q'$ in quanto $\stackrel{\hat{\alpha}}{\Longrightarrow}$ e $\stackrel{\alpha}{\Longrightarrow}$ si equivalgono se $\alpha \neq \tau$, e quindi $p = q$ segue dalla definizione. □

Proviamo infine un risultato tanto inatteso quanto importante e che mostra ancora una volta come la nozione di bisimilarità e di uguaglianza siano vicine; questo risultato venne per la prima volta provato da Matthew Hennessey e verrà poi sfruttato in seguito per provare la completezza dell'insieme di assiomi che introdurremo:

Proposizione 6.23 (Hennessey). $p \approx q$ *sse* $p = q$ *o* $\tau.p = q$ *o* $p = \tau.q$

Dimostrazione. ($\Leftarrow$) Sappiamo che $p = q$ implica $p \approx q$; le altre implicazioni seguono da $p \approx \tau.p$.

($\Rightarrow$) Assumiamo $p \approx q$, e consideriamo i tre casi possibili. Primo, supponiamo che $p \xrightarrow{\tau} p' \approx q$ per qualche p'; è facile vedere allora come $p = \tau.q$. Secondo, supponiamo che $q \xrightarrow{\tau} q' \approx p$ per qualche q'; in modo simile si mostra che $\tau.p = q$. Se nessuna delle due condizioni si è verificata, allora si può mostrare che $p = q$ nel modo seguente: consideriamo dapprima che $p \xrightarrow{l} p'$; dal momento che $p \approx q$ abbiamo che $q \stackrel{\hat{l}}{\Longrightarrow} q' \approx p'$, e cioè $q \stackrel{l}{\Longrightarrow} q' \approx p'$ come richiesto. Nell'altro caso, e cioè se $p \xrightarrow{\tau} p'$ allora $q \Longrightarrow q' \approx p'$, e q' non può essere q stesso per ipotesi e quindi $q \stackrel{\tau}{\Longrightarrow} q' \approx p'$ come richiesto. Grazie alla simmetria segue che $p = q$. □

Quanti abbiamo fatto in questa sezione è stato trovare una nozione di uguaglianza completamente sostitutiva e molto vicina alla bisimilarità; a questo punto potremmo obiettare che la bisimilarità è quasi ridondante. Invece, le dimostrazioni che stabiliscono una bisimilarità sono molto convenienti, ed inoltre, stabilire una bisimilarità è molto naturale ed a volte anche meccanizzabile.

6.3 Assiomatizzazione della bisimulazione

Definizione 6.5. *Un processo è* finito *se contiene solo Somma finita e nessuna ricorsione.*

E' chiaro che attraverso l'expansion law, ogni processo finito può essere uguagliato ad uno che non contiene Composizione parallela, Restrizioni o Relabelling.

Definizione 6.6. *L'equazione E di un processo è* seriale *se non contiene Composizione parallela, Restrizioni o Relabelling, e le equazioni che definiscono ogni Costante in E non contengono Composizione parallela, Restrizioni o Relabelling.*

Dunque, attraverso l'expansion law ogni processo può essere uguagliato ad un processo seriale.

Definiamo ora due insiemi di assiomi, uno per Somme ed uno per Sequenzalizzazione:

	Assiomi $\mathcal{A}_1$
A1	$p + q = q + p$
A2	$p + (q + r) = (p + q) + r$
A3	$p + p = p$
A4	$p + nil = p$

	Assiomi $\mathcal{A}_2$
$\mathcal{A}_1$	
A5	$\alpha.\tau.p = \alpha.p$
A6	$p + \tau.p = \tau.p$
A7	$\alpha.(p + \tau.q) + \alpha.q = \alpha(p + \tau.q)$

Sappiamo già che la seguente è conseguenza di $\mathcal{A}_2$:

$$\textbf{A6}' \qquad p + \tau.(p + q) = \tau.(p + q)$$

Con l'aiuto dell'expansion law ogni equazione valida tra processi finiti segue da $\mathcal{A}_2$: questo significa che abbiamo una completa assiomatizzazione per i processi finiti, e questo significa che possiamo mostrare che $\mathcal{A}_2$ sono corretti e completi per processi finiti seriali.

In quello che segue useremo $p = q$ per indicare che p e q sono uguali per la definizione 6.4 di uguaglianza, mentre $\mathcal{A} \vdash p = q$ intendiamo dire che l'uguaglianza può essere provata da un ragionamento equazionale dagli assiomi $\mathcal{A}$; useremo $\equiv$ per l'identità sintattica.

Consideriamo la potenza dell'insieme $\mathcal{A}_1$:

Definizione 6.7 (Standard form). *p è in standard form (abbreviato in s.f.) se*

$$p \equiv \sum_{i=1}^{m} \alpha_i.p_i$$

dove ogni p_i è anch'esso in standard form.

Si vede che nil è in s.f.: è il caso limite $m = 0$. Inoltre, si noti come l'ordine degli addendi possa essere ignorato grazie agli assiomi **A1** ed **A2**.

Lemma 6.3. *Per ogni processo p, esiste un processo p' in s.f. tale che*

$$\mathcal{A}_1 \vdash p = p'$$

Dimostrazione. Con l'uso degli assiomi $\mathcal{A}_1$, il termine nil può essere eliminato da ogni Somma, ed il risultato è in s.f. □

Il risultato seguente è molto importante e mostra come $\mathcal{A}_1$ sia corretto e completo rispetto alla nozione di strong congruence.

Proposizione 6.24 (Correttezza e completezza). *$p \sim q$ sse $\mathcal{A}_1 \vdash p = q$*

Dimostrazione. ($\Leftarrow$) Correttezza. Quello che bisogna osservare è che tutti gli assiomi $\mathcal{A}_1$ sono ancora veri se sostituiamo a '=' '$\sim$'; questo risultato è già stato provato nella parte della congruenza forte (proposizione 6.7).

($\Rightarrow$) Completezza. Assumiamo $p \sim q$ ed inoltre che p e q siano entrambi in standard form, cioè della forma $p \equiv \sum_{i=1}^{m} \alpha_i.p_i$ e $q \equiv \sum_{j=1}^{n} \alpha_j.q_j$. Proveremo la proposizione per induzione sull'altezza massima di p e q, dove l'altezza di p è definita come il massimo numero di Sequenzalizzazioni innestate in p.

Se la massima profondità è pari a zero, allora entrambi i processi sono nil (in quanto entrambi in standard form) ed il ragionamento equazionale ci garantisce che che $\mathcal{A}_1 \vdash nil = nil$ (la riflessività è parte del ragionamento equazionale).

In caso contrario, sia $\alpha.p'$ un sommando di p. Allora $p \xrightarrow{\alpha} p'$, e dal fatto che $p \sim q$, esiste un certo q' tale che $q \xrightarrow{\alpha} q' \sim p'$. Poichè q è in standard form, allora $\alpha.q'$ è un sommando di q e per induzione $\mathcal{A}_1 \vdash p' = q'$ e quindi il sommando $\alpha.p'$ di p può essere provato uguale ad un sommando di q. Similmente, ogni sommando $\beta.q'$ di q può essere provato uguale ad un sommando di p. Segue dunque che $\mathcal{A}_1 \vdash p = q$, usando l'assioma **A3** per eliminare i sommandi duplicati (e gli assiomi **A1** ed **A2** per riordinare e raggruppare i sommandi, ove necessario).

□

Definizione 6.8 (Full standard form). *p è in full standard form (f.s.f.) se*

1. *$p \equiv \sum_{i=1}^{m} \alpha_i.p_i$, dove ogni p_i è anch'esso in full standard form;*

2. *per ogni $p \stackrel{\alpha}{\Longrightarrow} p'$, allora $p \xrightarrow{\alpha} p'$*

Possiamo pensare alla f.s.f. come 'saturata' nel seguente senso: per ogni $p \stackrel{\alpha}{\Longrightarrow} p'$ allora $\alpha.p'$ appare come un sommando di p. Vedremo ora come ogni s.f. può essere saturata utilizzando $\mathcal{A}_2$

Lemma 6.4 (Saturazione). *Se $p \stackrel{\alpha}{\Longrightarrow} p'$, allora $\mathcal{A}_2 \vdash p = p + \alpha.p'$.*

Dimostrazione. Svolgiamo la prova per induzione sulla struttura di p. Consideriamo tre casi per $p \stackrel{\alpha}{\Longrightarrow} p'$:

Caso 1 $\alpha.p'$ è un sommando di p. La conclusione vale grazie all'assioma **A3**.

Caso 2 $\alpha.q$ è un sommando di p e $q \stackrel{\tau}{\Longrightarrow} p'$. Allora per induzione $\mathcal{A}_2 \vdash q = q + \tau.p'$, quindi

$$\begin{array}{rcl} \mathcal{A}_2 \vdash p & = & p + \alpha.q \quad \text{da } \mathbf{A3} \\ & = & p + \alpha.(q + \tau.p') \\ & = & p + \alpha.(q + \tau.p') + \alpha.p' \quad \text{da } \mathbf{A7} \\ & = & p + \alpha.p' \quad \text{dai passi precedenti al contrario} \end{array}$$

Caso 3 $\tau.q$ è un sommando di p e $q \stackrel{\alpha}{\Longrightarrow} p'$. Allora per induzione $\mathcal{A}_2 \vdash q = q + \alpha.p'$, quindi

$$\begin{array}{rcl} \mathcal{A}_2 \vdash p & = & p + \tau.q \quad \text{da } \mathbf{A3} \\ & = & p + \tau.q + q \quad \text{da } \mathbf{A6} \\ & = & p + \tau.q + q + \alpha.p' \\ & = & p + \alpha.p' \quad \text{dai passi precedenti al contrario} \end{array}$$

e questo completa la prova.

□

Con l'aiuto di questo lemma, possiamo trasformare ogni s.f. in una f.s.f. equivalente:

Lemma 6.5. *Per ogni processo p in s.f. esiste un processo p' in f.s.f. di uguale profondità, tale che $\mathcal{A}_2 \vdash p = p'$*

Dimostrazione. Per induzione sulla struttura di p. Per il caso base, $p \equiv nil$ e p è già in f.s.f.. In caso contrario, per ogni sommando $\beta.q$ di p possiamo assumere per induzione che q sia già convertito, grazie ad $\mathcal{A}_2$, in f.s.f. senza incremento di profondità. Consideriamo dunque tutte le coppie (α_i, p_i), $1 \leq i \leq k$, tale che $p \stackrel{\alpha_i}{\Longrightarrow} p_i$ ma non $p \stackrel{\alpha_i}{\longrightarrow} p_i$. Ogni p_i deve essere un f.s.f. in quanto deve essere una sottoespressione di qualche sommando $\beta.q$ di p. Dunque

$$p' \equiv p + \alpha_1.p_1 + \cdots + \alpha_k.p_k$$

è in f.s.f. di uguale profondità di p, e grazie al lemma di saturazione, $\mathcal{A}_2 \vdash p = p'$.

□

Siamo adesso in grado di dimostrare un risultato importante di questa sezione, cioè che $\mathcal{A}_2$ è corretto e completo per la nozione di uguaglianza su processi finiti seriali:

Proposizione 6.25 (correttezza e completezza). *$p = q$ sse $\mathcal{A}_2 \vdash p = q$*

Dimostrazione. ($\Leftarrow$) Correttezza. Si deve solamente osservare che gli assiomi $\mathcal{A}_2$ sono tutti veri, quando intendiamo '=' come l'ugualianza come definita in definizione 6.4.

($\Rightarrow$) Completezza. Possiamo assumere che p e q siano in f.s.f.; la prova procederà per induzione sulla somma delle profondità di p e q.

Quando la somma è pari a zero, allora si ha $p \equiv nil \equiv q$, ed il risultato è banale. In caso contrario assumiamo $p = q$, e sia $\alpha.p'$ un sommando di p. Miriamo a provare che q ha un sommando provabilmente uguale ad $\alpha.p'$. Ora $p \xrightarrow{\alpha} p'$, quindi esiste un q' tale che $q \stackrel{\alpha}{\Longrightarrow} q'$ e $p' \approx q'$. Di più, $q \xrightarrow{\alpha} q'$ in quanto q è in f.s.f., e quindi $\alpha.q'$ è un sommando di q.

Non possiamo applicare immediatamente l'induzione, in quanto sappiamo soltanto che $p' \approx q'$ e non $p' = q'$. Ma dall'Hennessey theorem (proposizione 6.23) sappiamo che allora $p' = q'$ o $p' = \tau.q'$ o $\tau.p' = q'$.

Nel primo caso, dal momento che p' e q' sono in f.s.f. e di profondità minore di p e q, per induzione $\mathcal{A}_2 \vdash p' = q'$, quindi $\mathcal{A}_2 \vdash \alpha.p' = \alpha.q'$. Nel secondo caso dobbiamo dapprima convertire $\tau.q'$ in f.s.f. prima di poter applicare l'induzione. Dal lemma precedente esiste un q'' in f.s.f. di uguale profondità di $\tau.q'$, tale che $\mathcal{A}_2 \vdash \tau.q' = q''$; ma la somma delle profondità di p' e q'' e minore di uno della somma delle profondità di p e q, e quindi per induzione possiamo inferire che $\mathcal{A}_2 \vdash p' = q''$ e quindi $\mathcal{A}_2 \vdash p' = \tau.q'$ e grazie all'assioma $A5$ si ha che $\mathcal{A}_2 \vdash \alpha.p' = \alpha.q'$. Il terzo caso è simile a questo.

Dunque, in ognuno dei tre casi, abbiamo mostrato che da $\mathcal{A}_2$ ogni sommando di $\alpha.p'$ di p può essere provato uguale ad un sommando di q. In modo simile, ogni sommando $\beta.q'$ di q può essere provato uguale ad un sommando di p. In fine, utilizzando l'assioma $A3$ per eliminare gli eventuali sommandi duplicati, possiamo concludere che $\mathcal{A}_2 \vdash p = q$

□

Capitolo 7

Testing equivalence

Come fatto nel capitolo precedente, riprendiamo ora quanto visto sulla testing equivalence, cercando di darne una visione più organica ed esaustiva di questa equivalenze.

Riprenderemo dapprima alcune definizioni già introdotte, per poi cercare un'assiomatizzazione completa di questa equivalenza.

7.1 Teoria di testing

Il comportamento esterno di programmi o processi, e più in generale di sistemi, può essere investigato tramite l'utilizzo di una serie di test. Risulta inoltre importante non soltanto sapere se, dato un particolare test, un processo risponde favorevolmente oppure no, ma anche se il processo risponde in maniera consistente ogni volta che il test viene eseguito.

Possiamo pensare, in generale, ad un insieme di processi e ad un insieme di test; due processi saranno equivalenti (rispetto a questo insieme di test) se superano esattamente gli stessi test. Questa naturale equivalenza può essere divisa in due preordini sui processi: il primo formulato in termini dell'*abilità di rispondere positivamente ad un particolare test* il secondo in termini dell'*impossibilità di non rispondere positivamente ad un particolare test*; nell'ultimo caso vedremo come un processo p sia *meno di* un processo q se, per ogni test a qui p risponde positivamente, anche q risponde positivamente. Attraverso la congiunzione di questi due preordini se ne otterrà un terzo, quello che caratterizza l'impianto teorico che stiamo discutendo.

Iniziamo dunque ad introdurre con maggior dettaglio quello di cui ci occuperemo:

Processi: un insieme di termini chiusi sull'alfabeto Act_τ. L'insieme viene indicato come $\mathcal{P}$;

Osservatori: un insieme di termini chiusi sull'alfabeto $Act_\tau \cup \{w\}$ con $w \notin Act_\tau$ indicato con $\mathcal{O}$.

Stati: sono coppie $< p, o >$ dove p è lo stato di un processo e o è lo stato di un osservatore. Uno stato si dice soddisfatto se o può compiere un'azione w.

Computazione: dato un processo $p \in \mathcal{P}$ ed un osservatore $o \in \mathcal{O}$ rispettivamente con stato iniziale p ed o si definisce una computazione da $< p, o >$ come una sequenza finita od infinita di stati $< p_n, o_n >$ dove:

1. $< p_0, o_0 >$ corrisponde a $< p, o >$;
2. i) $< p_n, o_n > \xrightarrow{\tau} < p_{n+1}, o_{n+1} >$ se $p_n \xrightarrow{\tau} p_{n+1}$ e $o_n = o_{n+1}$ oppure $o_n \xrightarrow{\tau} o_{n+1}$ e $p_n = p_{n+1}$
   ii) $< p_n, o_n > \xrightarrow{\alpha} < p_{n+1}, o_{n+1} >$ se $p_n \xrightarrow{\alpha} p_{n+1}$ e $o_n \xrightarrow{\alpha} o_{n+1}$
3. Se la sequenza è finita allora l'ultimo elemento $< p_k, o_k > \overset{\mu}{\not\rightarrow}$ per ogni $\mu \in Act_\tau$.

Come detto, un osservatore è un processo che esegue test su altri processi comunicando con essi, in un certo senso guidandone l'esecuzione. Un test su un processo avrà esito positivo se il suo osservatore raggiunge uno stato in cui può eseguire l'azione w, che comunica il successo.

Definizione 7.1 (Osservazione). *Dato un processo P con alfabeto Act_τ e un osservatore O con alfabeto $Act_\tau \cup \{w\}$ un'*osservazione *è una sequenza del tipo:*

$$P|O = P_0|O_0 \xrightarrow{\tau} P_1|O_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n|O_n \overset{\tau}{\not\rightarrow}$$

cioè una sequenza di transizioni τ o w.

Un'osservazione si dice *soddisfacente* se $\exists i : O_i \xrightarrow{w}$, cioè se esiste uno stato in cui l'osservatore è soddisfatto; questa definizione non implica l'arresto dell'osservazione, ma richiede che si sia passati per uno stato che consentiva di eseguire un'azione w

Da quanto detto finora possiamo introdurre due concetti importanti per il proseguio:

Definizione 7.2. *Siano P e O rispettivamente un processo ed un osservatore, allora:*

i) P may satisfy *O se esiste una sequenza $s \in Act^*$ tale che $p_0 \overset{s}{\Rightarrow}$, $o_0 \overset{s}{\Longrightarrow} o_n$ e $o_n \xrightarrow{w}$;*

ii) P must satisfy *O se per ogni computazione $< p_0, o_0 > \xrightarrow{\mu_1} < p_1, o_1 > \xrightarrow{\mu_2}$ $\ldots$ $\exists n \geq 0$ tale che $o_n \overset{w}{\rightarrow}$*

La definizione sopra indica che un processo *may satisfy* un osservatore se esiste un'osservazione soddisfacente per $P|O$, mentre *must satisfy* un osservatore se tutte le osservazioni per $P|O$ sono soddisfacenti.

Con l'introduzione di *may/must satisfy* è possibile introdurre i tre preordini di questa teoria:

Definizione 7.3. *Sia $\mathcal{O}$ l'insieme degli osservatori, allora si possono definire i seguenti preordini sui processi:*

- $P \sqsubseteq_{may} Q$ *sse* $\forall O \in \mathcal{O} \quad P$ *may satisfy* $O \Rightarrow Q$ *may satisfy* O
- $P \sqsubseteq_{must} Q$ *sse* $\forall O \in \mathcal{O} \quad P$ *must satisfy* $O \Rightarrow Q$ *must satisfy* O
- $P \sqsubseteq_{test} Q$ *sse* $P \sqsubseteq_{may} Q \ \wedge \ P \sqsubseteq_{must} Q$

Definiti i preordini, prendendone il kernel, possiamo ottenere le relazioni di equivalenze corrispondenti:

Definizione 7.4. *Le relazioni di equivalenza sono:*

- $P \simeq_{may} Q$ sse $(P \sqsubseteq_{may} Q) \wedge (Q \sqsubseteq_{may} P)$
- $P \simeq_{must} Q$ sse $(P \sqsubseteq_{must} Q) \wedge (Q \sqsubseteq_{must} P)$
- $P \simeq_{test} Q$ sse $(P \simeq_{may} Q) \wedge (P \simeq_{must} Q)$

7.2 Caratterizzazione alternativa

Come già espresso in precedenza, dato un insieme di processi ed un insieme di test rilevanti due processi sono considerati equivalenti se superano esattamente gli stessi test. E' abbastanza intuitivo vedere che le equivalenze (e quindi anche i preordini) ottenuti seguendo questa via sono difficili da verificare. Risulta relativamente semplice provare che due processi non sono equivalenti (basta trovare *un* controesempio) mentre diventa assai più complicato o comunque lungo e tedioso verificare che due processi siano equivalenti (bisogna prendere in considerazione *tutti* i possibili osservatori e le risultanti osservazioni).

Possiamo però trovare una caratterizzazione alternativa delle equivalenze in modo da ovviare a questo inconveniente: prenderemo in considerazione le sequenze di azioni che ogni sistema può eseguire e l'insieme di azioni che il sistema deve accettare, rendendoci indipendenti dalla nozione di osservazione; controlleremo, cioè, che i due sistemi possano eseguire le stesse sequenze di azioni e che, dopo ogni sequenza, possano scegliere la loro prossima azione dallo stesso insieme di azioni.

Inoltre questa caratterizzazione alternativa apre uno spiraglio che chiarisce le connessioni con le altre equivalenze studiate.

7.2.1 Caratterizzazione alternativa di *may*

Seguendo quanto premesso la definizione alternativa del preordine *may* (e di conseguenza dell'equivalenza) è il seguente:

$$P \sqsubseteq_{may} Q \text{ se } \forall s \in Act^* \quad P \stackrel{s}{\Rightarrow} \text{ implica } Q \stackrel{s}{\Rightarrow}$$

Da questa definizione si vede che utilizzando il preordine di *may* l'equivalenza indotta è quella a traccie deboli.

7.2.2 Caratterizzazione alternativa di *must*

La definizione alternativa del preordine di *must* non è così diretta come la precedente: abbiamo bisogno di alcune premesse per poter proseguire:

Definizione 7.5.

- *(P after s)* $= \{P' \mid P \stackrel{s}{\Longrightarrow} P'\}$ *è l'insieme dei processi che P può raggiungere dopo aver fatto* $s \in Act^*$.
- *(P must a) sse* $P \stackrel{\tau}{\Longrightarrow} P'$ *implica* $P' \stackrel{a}{\Longrightarrow}$, *cioè P può evolvere con* τ *e raggiungere stati in cui può fare a.*
- *(P must L) sse* $\exists a \in L$ *t.c. P must a.*
- *(*$\mathbb{P}$ *must L) sse* $\forall P \in \mathbb{P}$ *P must L.*

Tramite queste premesse, possiamo introdurre la caratterizzazione alternativa di *must*:

Definizione 7.6. $P \sqsubseteq_{must} Q$ *se* $\forall s \in Act^*$, $\forall L \subseteq Act$, *con L finito,* $P \stackrel{s}{\Rightarrow}$ *implica:*

- $Q \stackrel{s}{\Rightarrow}$
- *((P after s) must L) implica ((Q after s) must L)*
  cioè per ciascun dei processi raggiunti da P dopo aver fatto $\stackrel{s}{\Rightarrow}$ *esiste un'azione di L che possono fare dopo zero o più* τ.

Il vantaggio di questa diversa caratterizzazione è che la sequenza di azioni interessanti sono un numero finito e quindi la prova si riduce drasticamente in quanto per ogni sequenza x non interessante si ha che (P after x)$= \emptyset$, come mostra il seguente

Lemma 7.1. *(P after s) must* $\emptyset$ *sse* $s \notin$ *Traces(P).*

Dimostrazione. ($\Leftarrow$) E' banale, in quanto se $s \notin$ Traces(P) questo implica che (P after s) $= \emptyset$.

($\Rightarrow$) Supponiamo che $s \in$ Traces(P). Allora sappiamo che esiste un P' tale che $P \stackrel{s}{\Longrightarrow} P'$ e per nessun $a \in \emptyset$ si ha che $P' \stackrel{a}{\longrightarrow}$ e quindi si ha che (P after s) not must $\emptyset$. □

7.3 Assiomatizzazione di testing equivalence

Seguendo sempre la linea applicata per la bisimulazione, cerchiamo ora un'insieme di assiomi corretto e completo rispetto alla nozione di testing equivalence. Prima di fare questo, vediamo le relazione tra questa nuova equivalenza e proprio la bisimulazione debole:

Teorema 7.1. *Dati due processi p e q tali che $p \approx q$, questo implica $p \sqsubseteq_{may} q$.*

Dimostrazione. Dimostriamo l'implicazione inversa delle negazioni, e cioè: $p \not\sqsubseteq_{may} q \Rightarrow p \not\approx q$.

Secondo la definizione alternativa di $\sqsubseteq_{may}$ si ha:

$$P \not\sqsubseteq_{may} Q \text{ implica che } \exists s : P \stackrel{s}{\Longrightarrow} \wedge\, Q \stackrel{s}{\not\Rightarrow}$$

Procediamo dunque per induzione sulla struttura di s:

Caso base: s=a

Sostituiamo questa ipotesi nella relazione scritta sopra e si ottiene $P \stackrel{a}{\Longrightarrow} \wedge\, Q \stackrel{a}{\not\Rightarrow}$. Si possono allora distinguere due casi per $P \stackrel{a}{\Rightarrow}$:

1. $P \stackrel{a}{\rightarrow} \wedge\, Q \stackrel{a}{\not\Rightarrow}$.
   Ma dalla definizione di $\approx$ si ottiene che $P \not\approx Q$.
2. $P \stackrel{\tau}{\longrightarrow} P' \stackrel{a}{\Longrightarrow} \wedge\, Q \stackrel{a}{\not\Rightarrow}$.
   Dimostriamo dunque la tesi per induzione sulla somma delle profondità di P e Q. Ci troviamo difronte a tre casi per $Q \stackrel{a}{\not\Rightarrow}$:
   a) $Q \stackrel{b}{\rightarrow}$ con $b \neq a$.
   Essendo $b \neq a$ per definizione di bisimulazione si ha direttamente $P \not\approx Q$.
   b) $Q \stackrel{\tau}{\not\rightarrow}$.
   Quindi si ha che $P \stackrel{\tau}{\rightarrow} P'$ e $Q \stackrel{\epsilon}{\Rightarrow} Q$. Assumiamo per assurdo $P' \approx Q$, allora per definizione si avrebbe $P \approx Q$. Ma per induzione interna, la somma delle profondità di P' e Q è minore di 1 rispetto a quella di P e Q, per cui si otterrebbe l'assurdo $P' \not\approx Q$.
   Anche in questo caso si ottiene che $P \not\approx Q$.
   c) $Q \stackrel{\tau}{\longrightarrow} Q' \stackrel{a}{\not\Rightarrow}$.
   Avremmo quindi $P \stackrel{\tau}{\rightarrow} P' \stackrel{a}{\Rightarrow}$. La somma delle profondità di P' e Q' è inferiore di 2 rispetto a quella di P e Q, perciò applicando l'ipotesi induttiva interna otteniamo che $P' \not\approx Q'$, da cui, nuovamente, che $P \not\approx Q$.

Passo induttivo: $s = as' \wedge\ P \stackrel{s}{\Rightarrow} \wedge Q \stackrel{s}{\not\Rightarrow}$.

Quindi, si ha $P \stackrel{a}{\Longrightarrow} P' \stackrel{s'}{\Rightarrow}$. Si devono prendere in considerazione due casi:

1. $Q \stackrel{a}{\not\Rightarrow}$.
   Si può riapplicare il caso base.
2. $Q \stackrel{a}{\Longrightarrow} Q' \stackrel{s'}{\not\Rightarrow}$.
   Applicando l'ipotesi induttiva per P' e Q' e si ha $P' \not\approx Q'$, da cui si ottiene $P \not\approx Q$.

□

Dimostriamo un teorema simile anche per il preordine di *must*:

Teorema 7.2. *Dati due processi p e q tali che* $p \approx q$*, questo implica* $p \sqsubseteq_{must} q$.

Dimostrazione. La dimostrazione segue il metodo visto in precedenza: vedremo cioè che $P \not\sqsubseteq_{must} Q \Rightarrow P \not\approx Q$.

Secondo la definizione alternativa di $\sqsubseteq_{must}$ si ha:

$$P \not\sqsubseteq_{must} Q \text{ implica che } \forall L \subseteq Act, \exists s :$$
$$\forall P' :\ P \stackrel{s}{\Longrightarrow} P' \wedge\ Init(P') \cap L \neq \emptyset$$
$$\exists Q' :\ Q \stackrel{s}{\Longrightarrow} Q' \wedge\ Init(Q') \cap L = \emptyset,$$

dove con $Init(P)$ intendiamo l'insieme delle azioni iniziali che il processo P può compiere.

Quanto scritto sopra significa che esiste un processo Q' che non può fare nessuna azione di L, mentre P ne può fare almeno una: sia questa azione $a \in L$. Quindi $P \stackrel{s}{\Longrightarrow} P' \stackrel{a}{\Rightarrow}$ e $Q \stackrel{s}{\Longrightarrow} Q' \stackrel{a}{\not\Rightarrow}$. Da cui si ricava $P' \not\approx Q'$ e di conseguenza $P \not\approx Q$.

□

Possiamo facilmente mostrare che:

Teorema 7.3. *Dati due processi p e q tali che* $p \approx q$*, questo implica* $p \sqsubseteq_{test} q$.

Dimostrazione. La prova è immediata: dalla definizione di $\sqsubseteq_{test}$ e dai due precedenti teoremi la tesi è banalmente verificata. □

I tre risultati precedenti sono molto importanti per la correttezza dell'assiomatizzazione che andiamo cercando, in quanto provano che gli assiomi per la bisimulazione debole sono validi anche per i preordini della testing equivalence.

7.3.1 Assiomi

Quando si definisce un'assiomatizzazione si utilizza molto la regola di sostituzione per le dimostrazioni, quindi è necessario che la relazione considerata sia una congruenza: purtroppo per la testing equivalence questa proprietà non è vera:

Teorema 7.4. *La testing equivalence non è una congruenza.*

Dimostrazione. La dimostrazione è molto semplice, in quanto basta mostrare un contesto che non preserva la sostitutività. Prendiamo l'equazione $a \simeq_{test} \tau.a$ vera poiché

- $a \simeq_{may} \tau.a$ in quanto i due processi hanno le stesse tracce deboli $\stackrel{a}{\Rightarrow}$
- $a \simeq_{must} \tau.a$ in quanto gli unici osservatori interessanti sono w e $\bar{a}.w$.

Consideriamo il contesto b+[ ] e si ottiene $b + a \not\simeq_{test} b + \tau.a$: prendiamo infatti l'osservatore $O = \bar{b}.w$ ed otteniamo:

- $b + a$ must satisfy O
- $b + \tau.a$ not must satisfy O
  in quanto la computazione $b + \tau.a|\bar{b}.w \stackrel{\tau}{\rightarrow} a|\bar{b}.w$ non è soddisfacente.

□

In maniera simile a quanto fatto per la weak bisimulation, daremo ora una definizione leggermente differente della testing equivalence in modo che la proprietà di essere una congruenza sia rispettata:

Definizione 7.7 (di testing congruence). $P \sqsubseteq^{C}_{test} Q$ *sse* $P \sqsubseteq_{test} Q \wedge (P \stackrel{\tau}{\rightarrow} \Rightarrow\ Q \stackrel{\tau}{\rightarrow})$.

La differenza è minima ma fondmentale: si richiede infatti che se la prima azione di P è un τ, allora anche Q fa come prima azione proprio τ.

Mostriamo ora quali sono gli assiomi di questa teoria:

Assiomi per testing congruence

(A1) $P + Q = Q + P$

(A2) $(P + Q) + R = P + (Q + R)$

(A3) $P + P = P$

(A4) $P + nil = P$

(N1) $\mu.P + \mu.Q = \mu.(\tau.P + \tau.Q)$ (non importa se scelgo prima o dopo l'azione μ)

(N2) $P + \tau.Q \sqsubseteq \tau.(P + Q)$ (posso fare oppure no il τ iniziale)

(N3) $\mu.P + \tau.(\mu.Q + R) = \tau.(\mu.P + \mu.Q + R)$

(N4) $\tau.P \sqsubseteq P$

Questi assiomi sono tutti e soli gli assiomi per la testing congruence: gli assiomi **(A1)**-**(A4)** sono quelli della bisimulazione, mentre **(N1)**-**(N4)** sostituiscono le τ laws. Questo insieme di assiomi viene detto $\mathcal{A}$....

Assiomi per *must* preorder

Aggiungendo ad $\mathcal{A}$ l'assioma

(E1) $\tau.P + \tau.Q \sqsubseteq P$

si ottiene l'assiomatizzazione per il preordine di must.

Assiomi per *may* preorder

Aggiungendo ad $\mathcal{A}$ l'assioma

(F1) $P \sqsubseteq \tau.P + \tau.Q$

si ottiene invece l'assiomatizzazione per il preordine di may.

7.3.2 Forme normali per testing

In modo analogo a quanto fatto con bisimulazione definiamo ora delle forme normali che ci aiuteranno per verificare la completezza degli assiomi.

Definizione 7.8 (di insieme saturato). *Sia $\mathcal{L}$ un insieme non vuoto di insiemi finiti, cioè $\mathcal{L} \subseteq \mathbb{P}(Act)$. Sia $Act(\mathcal{L}) = \{a | a \in L, L \in \mathcal{L}\}$ l'insieme di tutte le azioni di $\mathcal{L}$. Si dice che $\mathcal{L}$ è* saturato *se $\forall K \subseteq Act$*

$$L \in \mathcal{L} \wedge\ L \subseteq K \subseteq Act(\mathcal{L})\ \textit{implica}\ K \in \mathcal{L}$$

Cerchiamo di spiegare intuitivamente cosa si intende con la definizione precedente: sia $\mathcal{L}$ un insieme di insiemi di azioni; estraiamo da questo tutte le azioni elementari e mettiamole in un insieme chiamato $Act(\mathcal{L})$. Prendiamo ora un sottoinsieme di azioni K: se possiamo trovare sempre un elemento L appartenente a $\mathcal{L}$ che sia sottoinsieme di K, e se questo K contiene azioni presenti anche in $Act(\mathcal{L})$ allora possiamo dire che $\mathcal{L}$ è saturato.

Definizione 7.9 (Forma normale). *Un processo è in forma normale se è in una delle due forme:*

a) $\sum\{\tau.\sum\{a.P_a | a \in L\} | L \in \mathcal{L}\}$ *con P_a in forma normale*

b) $\sum\{a.P_a | a \in L\}$ *con* P_a *in forma normale*

dove $\mathcal{L}$ *è un insieme saturato*

Questa forma normale ci consente di stratificare un processo in fasi do si fanno solo azioni invisibili o solo azioni visibili.

Definizione 7.10 (Weak normal form). *Un processo è in weak normal form (w.n.f.) se è del tipo:*

$$\sum\{a.P_a | a \in L\} \text{ con } P_a \text{ in weak normal form}$$

Tramite questa nozione, i processi sono visti come alberi deterministici di azioni visibili. Questa è la forma normale utilizzata per l'assiomatizzazione della may, che è una congruenza in quanto coincide con l'equivalenza a tracce. Inoltre abbiamo questo risultato:

Teorema 7.5. *Ogni processo in forma normale può essere portato in weak normal form.*

Dimostrazione. La dimostrazione viene fatta per induzione sulla profondità del processo.

Caso base: nil, il quale è già in w.n.f.

Passo induttivo: sia P il nostro processo ed $\alpha \in Init(P)$, allora si hanno due casi possibili:

1. $\alpha = \tau$
   Si può eliminare il τ tramite gli assiomi **(N4)** e **(F1)** e sul resto del processo si applica l'ipotesi induttiva.
2. $\alpha \neq \tau$
   Se α è distinto da tutti i $\beta \in Init(P)$ allora, tramite l'ipotesi induttiva, si ottiene la tesi.
   Se, invece, $\alpha = \beta$ allora tramite l'assioma **(N1)** e l'ipotesi induttiva si ottiene nuovamente la tesi.

□

7.3.3 Completezza dell'assiomatizzazione

Teorema 7.6 (Completezza). *Dati due processi P e Q tali che* $P \simeq_{test} Q$*, allora questo implica* $(\mathbf{A1}) - (\mathbf{N4}) \vdash P = Q$.

Dimostrazione. (Traccia) Supponiamo che P e Q siano in forma normale (nel caso non lo fossero, abbiamo già mostrato che è possibile trasformarli in processi equivalenti ma in n.f.). Allora avremo $P = Q$ se ogni sommando in P è presente anche in Q.

Supponiamo, per assurdo, che P abbia un sommando che Q non ha, allora per la saturazione delle forme normali avremmo un insieme L tale che P must L ma Q not must L che porta all'assurdo $P \not\approx_{test} Q$. □

Si ha inoltre che

Teorema 7.7. $P \sqsubseteq_{may} Q$ *implica* $(\mathbf{A1}) - (\mathbf{N4}) - (\mathbf{F1}) \vdash P \sqsubseteq Q$

di cui non forniremo una dimostrazione.