Modelli della Concorrenza

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Le immagini presenti in questi appunti sono tratte dalle slides del corso e tutti i diritti delle stesse sono da destinarsi ai docenti del corso stesso.

Capitolo 2

Introduzione alla Concorrenza

La **concorrenza** è presente in diversi aspetti della quotidianità. Un primo esempio di **sistema concorrente** non legato all'informatica è la *cellula vivente*: può essere vista come un dispositivo che trasforma e manipola dati per ottenere un risultato. I vari processi all'interno di una cellula avvengono in modo concorrente, il che la rende un *sistema asincrono*. Un secondo esempio può essere quello dell'*orchestra musicale*: i vari componenti suonano spesso simultaneamente rappresentando un *sistema sincrono* (ovvero un sistema che funziona avendo una sorta di "cronometro" condiviso dai vari attori). Un esempio informatico invece è un *processore multicore* (anche se in realtà anche se fosse *monocore* sarebbe comunque un sistema concorrente per ovvie ragioni). Anche una *rete di calcolatori* è un modello concorrente, nonché i *modelli sociali umani*.

Caratteristiche comuni

I modelli concorrenti hanno alcuni aspetti comuni, tra cui:

- competizione per l'accesso alle risorse condivise
- cooperazione per un fine comune (che può portare a competizione)
- coordinamento di attività diverse
- sincronia e asincronia

Studio

Durante lo studio e la progettazione di sistemi concorrenti si hanno diversi problemi peculiari che rendono il tutto molto complesso. Un sistema concorrente mal progettato può avere effetti catastrofici.

Per poter sviluppare modelli concorrenti si necessita innanzitutto di:

- **linguaggi**, per specificare e rappresentare sistemi concorrenti.
  - **linguaggi di programmazione** (con l'uso di *thread*, *mutex*, scambio di messaggi, etc... con i vari problemi di *race condition*, uso di variabili condivise etc...).
  - linguaggi rappresentativi, come ad esempio una *partitura musicale* (nella quale si visualizza bene la natura *sincrona*).
  - **task graph (*grafo delle attività*)**, nel quale i nodi sono le attività (o eventi) mentre gli archi rappresentano una *relazione d'ordine parziale*, come per esempio una *relazione di precedenza* sui nodi.
  - **algebre di processi**, simile ad un sistema di equazioni, con simboli che rappresentano eventi del sistema concorrente e operatori atti a comporre fra loro i vari sottoprocessi del sistema concorrente. Ogni "equazione" descrive un processo che costituisce un elemento di un sistema concorrente.
  - **modelli**, per modellare sistemi concorrenti in astratto. Un esempio è dato dalle **reti di Petri**, che modellano un sistema concorrente partendo dalle nozioni di *stato locale* di uno dei componenti del sistema e di *evento locale* che ha un effetto su alcune componenti (e non tutte). Si ha quindi rappresentato un *sistema dinamico* che si evolve nel tempo e la cui evoluzione è rappresentata tramite *relazioni di flusso*).
- **logica**, per analizzare e specificare sistemi concorrenti.
- **model-checking**, per validare formule relative a proprietà di sistemi concorrenti.

Capitolo 3

Logica

Si ringrazia Marco Natali [1] *per questo ripasso.*

La logica è lo studio del ragionamento e dell'argomentazione e, in particolare, dei procedimenti inferenziali, rivolti a chiarire quali procedimenti di pensiero siano validi e quali no. Vi sono molteplici tipologie di logiche, come ad esempio la logica classica e le logiche costruttive, tutte accomunate dall'essere composte da 3 elementi:

- **Linguaggio**: insieme di simboli utilizzati nella Logica per definire le cose.
- **Sintassi**: insieme di regole che determina quali elementi appartengono o meno al linguaggio.
- **Semantica**: permette di dare un significato alle formule del linguaggio e determinare se rappresentano o meno la verità.

3.1 Logica proposizionale

Ci occupiamo della *logica classica* che si compone in *logica proposizionale* e *logica predicativa*. La logica proposizionale è quindi un tipo di logica classica che presenta come caratteristica principale quella di essere un linguaggio limitato, ovvero caratterizzato dal poter esprimere soltanto proposizioni senza possibilità di estensione ad una classe di persone.

[1] Link al repository: https://github.com/bigboss98/Appunti-1/tree/master/Primo Anno/Fondamenti

3.1.1 Sintassi

Il linguaggio di una logica proposizionale è composto dai seguenti elementi:

- Variabili Proposizionali atomiche (o elementari): $P, Q, R, p_i, \ldots$.
- Connettivi Proposizionali: $\wedge, \vee, \neg, \implies, \iff$
- Simboli Ausiliari: "(" e ")" (detti delimitatori)
- Costanti: T (*True, Vero,* $\top$) e F (*False, Falso,* $\bot$)

La sintassi di un linguaggio è composta da una serie di formule ben formate (FBF) definite induttivamente nel seguente modo:

1. Le costanti e le variabili proposizionali: $\top, \bot, p_i \in FBF$.
2. Se A e $B \in FBF$ allora $(A \wedge B)$,$(A \vee B)$,$(\neg A)$,$(A \implies B)$, $(A \iff B)$ sono delle formule ben formate.
3. nient'altro è una formula

In una formula ben formata le parentesi sono bilanciate.

Esempio 1. *Vediamo degli esempi:*

- $(P \wedge Q) \in FBF$ *è una formula ben formata*
- $(PQ \wedge R) \notin FBF$ *in quanto non si rispetta la sintassi del linguaggio definita.*

Definizione 1. *Sia* $A \in FBF$*, l'insieme delle sottoformule di* A *è definito come segue:*

1. *Se* A *è una costante o variabile proposizionale allora* A *stessa è la sua sottoformula.*
2. *Se* A *è una formula del tipo* $(\neg A')$ *allora le sottoformule di* A *sono* A *stessa e le sottoformule di* A'*;* $\neg$ *è detto connettivo principale e* A' *sottoformula immediata di* A*.*
3. *Se* A *è una formula del tipo* $B \circ C$*, allora le sottoformule di* A *sono* A *stessa e le sottoformule di* B *e* C*;* $\circ$ *è il connettivo principale e* B *e* C *sono le due sottoformule immediate di* A*.*

È possibile ridurre ed eliminare delle parentesi attraverso l'introduzione della precedenza tra gli operatori, definita come segue:

$$\neg, \wedge, \vee, \implies, \iff$$

In assenza di parentesi una formula va parentizzata privilegiando le sottoformule i cui connettivi principali hanno la precedenza più alta.
In caso di parità di precedenza vi è la convenzione di associare da destra a sinistra. Segue un esempio:

$$\neg A \wedge (\neg B \implies C) \vee D \text{ diventa } ((\neg A) \wedge ((\neg B) \implies C) \vee D)$$

Albero Sintattico

Definizione 2. *Un albero sintattico T è un albero binario coi nodi etichettati da simboli di L, che rappresenta la scomposizione di una formula ben formata X definita come segue:*

1. Se X è una formula atomica, l'albero binario che la rappresenta è composto soltanto dal nodo etichettato con X
2. Se $X = A \circ B$, X è rappresentata da un albero binario che ha la radice etichettata con $\circ$, i cui figli sinistri e destri sono la rappresentazione di A e B
3. Se $X = \neg A$, X è rappresentato dall'albero binario con radice etichettata con $\neg$, il cui figlio è la rappresentazione di A

Poiché una formula è definita mediante un albero sintattico, le proprietà di una formula possono essere dimostrate mediante induzione strutturale sulla formula, ossia dimostrare che la proprietà di una formula soddisfi i seguenti 3 casi:

- è verificata la proprietà per tutte le formule atomo A
- supposta verifica la proprietà per A, si verifica che la proprietà è verificata per $\neg A$
- supposta la proprietà verificata per A_1 e A_2, si verifica che la proprietà è verifica per $A_1 \circ A_2$, per ogni connettivo $\circ$.

3.1.2 Semantica

La semantica di una logica consente di dare un significato e un'interpretazione alle formule del Linguaggio.

Definizione 3. *Sia data una formula proposizionale P e sia $P_1, \ldots, P_n$, l'insieme degli atomi che compaiono nella formula A. Si definisce come* interpretazione *una funzione $v : \{P_1, \ldots, P_n\} \mapsto \{T, F\}$ che attribuisce un valore di verità a ciascun atomo della formula A.*
*$v : P \to \{0, 1\}$ è un'**assegnazione booleana***

I connettivi della Logica Proposizionale hanno i seguenti valori di verità:

A	B	$A \wedge B$	$A \vee B$	$\neg A$	$A \Longrightarrow B$	$A \Longleftrightarrow B$
F	F	F	F	T	T	T
F	T	F	T	T	T	F
T	F	F	T	F	F	F
T	T	T	T	F	T	T

Essendo ogni formula A definita mediante un unico albero sintattico, l'interpretazione v è ben definita e ciò comporta che data una formula A e un'interpretazione v, eseguendo la definizione induttiva dei valori di verità, si ottiene un unico $v(A)$.

Definizione 4. *Una formula nella logica proposizionale può essere di diversi tipi:*

- ***Valida o Tautologica:** la formula è soddisfatta da qualsiasi valutazione della Formula*
- ***Soddisfacibile NON Tautologica:** la formula è soddisfatta da qualche valutazione della formula ma non da tutte.*
- ***Falsificabile:** la formula non è soddisfatta da qualche valutazione della formula.*
- ***Contraddizione:** la formula non viene mai soddisfatta*

Teorema 1. *Si ha che:*

- *A è una formula valida se e solo se $\neg A$ è insoddisfacibile.*
- *A è soddisfacibile se e solo se $\neg A$ è falsificabile*

Modelli e decidibilità

Si definisce *modello*, indicato con $M \models A$, tutte le valutazioni booleane che rendono vera la formula A. Si definisce *contromodello*, indicato con $\not\models$, tutte le valutazioni booleane che rendono falsa la formula A.

La logica proposizionale è decidibile (posso sempre verificare il significato di una formula). Esiste infatti una procedura effettiva che stabilisce la validità o no di una formula, o se questa ad esempio è una tautologia. In particolare il verificare se una proposizione è tautologica o meno è l'operazione di decidibilità principale che si svolge nel calcolo proposizionale.

Definizione 5. *Se $M \models A$ per tutti gli M, allora A è una tautologia e si indica $\models A$.*

Definizione 6. *Se $M \models A$ per qualche M, allora A è soddisfacibile.*

Definizione 7. *Se $M \models A$ non è soddisfatta da nessun M, allora A è insoddisfacibile.*

3.1.3 Equivalenze Logiche

Definizione 8. *Date due formule A e B, si dice che A è* logicamente equivalente *a B, indicato con $A \equiv B$, se e solo se per ogni interpretazione v risulta $v(A) = v(B)$.*

Nella logica proposizionale sono definite le seguenti equivalenze logiche, indicate con $\equiv$:

1. **Idempotenza:**

$$A \vee A \equiv A$$
$$A \wedge A \equiv A$$

2. **Associatività:**

$$A \vee (B \vee C) \equiv (A \vee B) \vee C$$
$$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$$

3. **Commutatività:**

$$A \vee B \equiv B \vee A$$
$$A \wedge B \equiv B \wedge A$$

4. **Distributività:**

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$
$$A \wedge (B \vee C) \equiv (A \wedge B \vee (A \wedge C)$$

5. **Assorbimento:**

$$A \vee (A \wedge B) \equiv AA \wedge (A \vee B) \equiv A$$

6. **Doppia negazione:**

$$\neg\neg A \equiv A$$

7. **Leggi di De Morgan:**

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

8. **Terzo escluso:**

$$A \vee \neg A \equiv T$$

9. **Contrapposizione:**

$$A \Longrightarrow B \equiv \neg B \Longrightarrow \neg A$$

10. **Contraddizione**

$$A \wedge \neg A \equiv F$$

Completezza di insiemi di Connettivi

Un insieme di connettivi logici è completo se mediante i suoi connettivi si può esprimere un qualunque altro connettivo. Nella logica proposizionale valgono anche le seguenti equivalenze, utili per ridurre il linguaggio:

$$(A \Longrightarrow B) \equiv (\neg A \vee B)$$
$$(A \vee B) \equiv \neg(\neg A \wedge \neg B)$$
$$(A \wedge B) \equiv \neg(\neg A \vee \neg B)$$
$$(A \Longleftrightarrow B) \equiv (A \Longrightarrow B) \wedge (B \Longrightarrow A)$$

L'insieme dei connettivi $\{\neg, \vee, \wedge\}$, $\{\neg, \wedge\}$ e $\{\neg, \vee\}$ sono completi.

Capitolo 4

Correttezza di programmi sequenziali

Introduciamo l'argomento con un esempio.

Esempio 2. *Definiamo una funzione in C che riceve un vettore, un intero (la lunghezza del vettore) e e restituisce un ulteriore numero intero. Chiedendoci*

Listing 1 Esempio di funzione in C

```
int f(int n, const int v[]) {
  int x = v[0];
  int h = 1;
  while (h < n) {
    if (x < v[h])
    x = v[h];
    h = h + 1;
  }
  return x;
}
```

cosa fa la funzione scopriamo che si occupa di cercare il massimo in un vettore.
La strategia della funzione è quella di spostarsi lungo il vettore e conservare in x il valore massimo fino ad ora trovato. Arrivati alla fine del vettore so che in x avrò il valore massimo.
Più formalmente suppongo che n sia $n > 0$, per dire che ho almeno un elemento nel vettore. Suppongo inoltre che $v[i] \in \mathbb{Z}$, $\forall i \in \{0, \ldots, n-1\}$. Abbiamo fissato le ***condizioni iniziali****.*
All'inizio x è il massimo del sotto-vettore con solo il primo elemento ($v[0..0]$)

e dopo l'assegnamento di h in $v[o..h-1]$, che, con $h = 1$ mi conferma che x è il massimo in $v[0..0]$.
Al termine di una certa iterazione x contiene il massimo tra i valori compresi tra $v[0]$ e $v[h-1]$ (detto altrimenti il massimo in $v[0..h-1]$). Inoltre al fine di una certa iterazione mi aspetto che $h \leq n$.
Possiamo quindi dire che quando esco dal ciclo x è il massimo in $v[0..h-1]$ ma in questo momento $h = n$ e quindi x è il massimo del vettore.
Consideriamo ora la parte iterativa. All'inizio di ogni iterazione suppongo che x è il massimo in $v[0..h-1]$. Dopo l'istruzione di scelta x è il massimo in $v[0..h]$, comunque sia andata la scelta. Alla fine dell'iterazione, dopo l'incremento di h, avrò ancora che x è il massimo in $v[0..h-1]$. Ragiono quindi per induzione. Se all'inizio dell'iterazione e alla fine ho la stessa asserzione, ed è vera prima di iniziare l'iterazione, posso dire che ho una ***proprietà invariante*** *e vale anche al termine dell'ultima iterazione e quindi vale anche alla fine dell'esecuzione del programma.*

Nell'esempio notiamo in primis l'assenza di formalità. Si introducono quindi concetti:

1. **precondizione** che nell'esempio è fatta da $n > 0$ e $v[i] \in \mathbb{Z},\ \forall i \in \{0, \ldots, n-1\}$
2. **postcondizione** che nell'esempio si ritrova con l'asserzione x è il massimo in $v[0..h-1]$ e $h = n$, che scritto in modo formale diventa:

$$\left.\begin{array}{l} v[i] \leq x,\ \forall i \in \{0, \ldots, n-1\} \\ \exists\, i \in \{0, \ldots, n-1\} \text{ t.c. } v[i] = x \end{array}\right\} x = max(v[0..n-1])$$

Queste formule possono essere rese come formule proposizionali, tramite una congiunzione logica:

$$\begin{cases} v[0] \leq x \wedge v[1] \leq x \wedge \ldots v[n-1] \leq x \\ v[0] = x \vee v[1] = x \vee \ldots v[n-1] = x \end{cases}$$

Abbiamo studiato lo stato della memoria del programma in un certo istante tramite formule.

Definizione 9. *Definiamo* ***stato della memoria*** *come:*

$$s : V \to \mathbb{Z}$$

ovvero una funzione che mappa le variabili del programma (poste nell'insieme V) in $\mathbb{Z}$.

Fissato uno stato della memoria e una formula posso validare una formula in quello stato osservando le variabili e le relazioni aritmetiche della formula. Data una formula ϕ e uno stato s posso sapere se ϕ è valida in s.
Una formula che gode della **proprietà invariante** se vera all'inizio dell'iterazione è vera anche alla fine della stessa. Per capire se è invariante basta vedere lo stato di una formula ad inizio e fine di una iterazione.
L'esecuzione di una istruzione cambia lo stato della memoria. Potrebbe però accadere che una serie di istruzioni non facciano terminare il programma, perciò quanto detto sopra è in realtà un'approssimazione della realtà.

Definizione 10. *Definiamo la* ***specifica di correttezza di un programma*** *con la tripla:*

$$\alpha \ P \ \beta$$

dove:

- *α e β sono formule (definite con tutte le simbologie aritmetiche tra variabili, sia di conto che di relazione).*
- *P è un "programma" (anche un frammento o una singola istruzione) che modifica lo stato della memoria.*

α è la ***precondizione****, che supponiamo verificata nello stato iniziale e β è la* ***postcondizione****, che supponiamo valida dopo l'esecuzione del programma.*

Durante il corso useremo un linguaggio imperativo non reale semplificato. Dovremo anche definire una logica, definendo un apparato deduttivo, un insieme di regole per costruire dimostrazioni derivando nuove formule da quelle preesistenti. Useremo la **logica di Hoare**. Le formule della logica di Hoare sono triple di tipo ($\alpha P \beta$) quindi si tratta di una logica di tipo diverso anche se si appoggia su quella proposizionale.

4.1 Linguaggio semplificato

Definiamo quindi il linguaggio di programmazione imperativo semplificato che andremo ad utilizzare. Si userà una grammatica formale.
L'elemento fondamentale di questo linguaggio è il **comando**, che indica o una singola istruzione o un gruppo di istruzioni strutturate. Il simbolo usato nella grammatica per indicare un comando è "C". Un comando viene costruito tramite le **produzioni**, introdotte da "::=". Il comando più semplice è l'**assegnamento**, che usa l'operatore ":=" per assegnare un valore ad una variabile. Con il simbolo "E" indichiamo un simbolo non terminale della grammatica che sta per *espressione*. Una volta costruito semplici espressioni

possiamo combinarle, eseguendole in sequenza, inserendo un ";" tra due comandi.
In merito all'istruzione di scelta abbiamo l'istruzione "if", seguito da un'espressione booleana, seguito da "then", seguita da un comando, seguita da "else", seguita da un comando, e il tutto viene concluso da "endif". Qui abbiamo una prima semplificazioni dicendo che l'*else* è obbligatorio. Per l'iterazione abbiamo il "while", seguito da un'espressione booleana, seguito da "do" con poi il comando, il tutto concluso da endwhile. Infine abbiamo una istruzione speciale, chiamata "skip", che non fa nulla e avanza il *program counter* (con essa posso saltare il ramo alternativo dell'*if-else*).
Un'espressione booleana "B" può essere la costante "true", la costante "false" o del tipo "not B", "B and B", "B or B", "E < E" (e le altre), "E = E". Non si hanno tipi di dato ma supporremo di avere a che fare solo con *interi*. Non si ha la funzione di nozione o di classe. Questo linguaggio è comunque *Turing Complete*.

Listing 2 Esempio di programma D

```
x := a; y := b;
while x != y do
  if x < y then
    y := y - x;
  else
    x := x - y;
  endif
endwhile
```

Cerchiamo di capire se il programma D, sopra definito, soddisfa la tripla:

$$\{a > 0 \land b > 0\}\ D\ \{x = MCD(a, b)\}$$

Quindi mi chiedo se eseguendo il programma con uno stato della memoria dove a e b sono due interi positivi (precondizione) allora, alla fine dell'esecuzione di D, x sarà il massimo comune divisore tra a e b (postcondizione). Dobbiamo dimostrare la tripla e qualora non fosse vera bisogna confutarla trovando un caso in cui non è verificata (trovando uno stato che soddisfi la precondizione ma che, una volta eseguito il programma, la postcondizione non sia verificata).

4.2 Logica di Hoare

Definizione 11. *Una **dimostrazione**, in una logica data, è una sequenza di formule di quella logica che sono o assiomi (formule che riteniamo vere*

a priori) o formule derivate dalle precedenti tramite una regola di inferenza. Una regola di inferenza mi manda da un insieme di formule $\alpha_1, \alpha_2, \ldots, \alpha_n$ ad una nuova formula α e si indica con:

$$\frac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha}$$

Che si legge come: "se ho già derivato $\alpha_1, \alpha_2, \ldots, \alpha_n$ sono autorizzato a derivare α".
*Nel nostro caso ogni α_i è una tripla della **logica di Hoare**.*
Una dimostrazione si ottiene quindi applicando le regole di derivazione fino ad arrivare, se si riesce, ad una soluzione.

4.2.1 Regole di derivazione

Vediamo quindi le regole di derivazione, che sono associate alle regole del linguaggio sopra definito.

Skip

Definizione 12. *Partiamo con la regola per l'istruzione **skip**, che non facendo nulla non cambia lo stato della memoria e quindi la regola di derivazione non ha nessuna premessa:*

$$\frac{}{\{p\}\ skip\ \{p\}}$$

con p che è una formula proposizionale. Dopo lo skip p vale se valeva prima dello skip

Implicazione

Definizione 13. *La seconda è una regola che non ha un rapporto diretto con il linguaggio e chiameremo **regola di conseguenza (o dell'implicazione)**. Ha due premesse: una tripla appartenente alla logica di Hoare e una implicazione della logica proposizionale.*

$$\frac{p \implies p' \quad \{p'\}\ C\ \{q\}}{\{p\}\ C\ \{q\}}$$

Ovvero se eseguo C partendo da uno stato in cui vale p' allora dopo varrà q. Ma sappiamo anche che p implica p'. Quindi se nel mio stato della memoria vale p e quindi anche p' per l'implicazione. Posso quindi dire che se ho p ed eseguo C ottengo q.
Ho anche una forma speculare:

$$\frac{\{p\}\ C\ \{q'\} \quad q' \implies q}{\{p\}\ C\ \{q\}}$$

Sequenza

Definizione 14. *La terza regola è legata alla struttura di* ***sequenza*** *dei programmi:*

$$\frac{\{p\}\ C_1\ \{q\}\quad \{q\}\ C_2\ \{r\}}{\{p\}\ C_1; C_2\ \{r\}}$$

Assegnamento

Definizione 15. *La quarta regola riguarda l'****assegnamento****. Questa è l'unica regola non banale (come lo è lo skip) che non ha premesse. È la regola base per derivare le triple necessarie alle altre regole. Quindi, avendo E come espressione, x una variabile e p come una postcondizione, ovvero una formula che contiene gli identificatori di diverse variabili:*

$$\frac{}{\{p[E/x]\}\ x := E\ \{p\}}$$

Dove con $p[E/x]$, come precondizione, indichiamo una ***sostituzione*** *indicante che cerchiamo in p tutte le occorrenze x e le sostituiamo con E.*

Esempio 3. *Se ho $x := y + 1$ con E pari a $y + 1$ e con p pari a $\{x > 0\}$ come postcondizione data e cerco la precondizione. Quindi la tripla completa sarebbe:*

$$\{y + 1 > 0\}\ x := y + 1\ \{x > 0\}$$

E la tripla sappiamo che è vera *(se so che $y + 1$ è positivo, dopo che a x assegno $y + 1$ posso essere sicuro che anche x è positivo).*

Esempio 4. *Se ho $x := x+2$ con E pari a $x+2$ e con p pari a $\{x > 0 \land x \leq y\}$ come postcondizione data e cerco la precondizione. Quindi la tripla completa sarebbe:*

$$\{x + 2 > 0 \land x + 2 \leq y\}\ x := x + 2\ \{x > 0 \land x \leq y\}$$

e anche questa tripla è garantita dalla regola di sostituzione

Istruzione di scelta

Definizione 16. *La quinta regola è quella relativa all'****istruzione di scelta*** *che è della forma:*

$$\{p\}\ \textit{if } B \textit{ then } C \textit{ else } D \textit{ endif}\ \{q\}$$

Se la condizione B è vera eseguo C altrimenti D. In entrambi i casi alla fine deve valere la postcondizione q. Separiamo i due casi:

- *se suppongo vere p e B eseguo C arrivando in q:*

$$\{p \wedge B\}\ C\ \{q\}$$

- *se suppongo vera p ma falsa B avrò:*

$$\{p \wedge \neg B\}\ D\ \{q\}$$

Ricavo quindi la formula generale:

$$\frac{\{p \wedge B\}\ C\ \{q\} \qquad \{p \wedge \neg B\}\ D\ \{q\}}{\{p\}\ \textit{if } B \textit{ then } C \textit{ else } D \textit{ endif}\ \{q\}}$$

Come notazione usiamo che:

$$\vdash \{p\}\ C\ \{q\}$$

dove $\vdash$ segnala che la tripla è stata **dimostrata/derivabile** con le regole di derivazione (si parla quindi di *sintassi*, viene infatti ignorato il significato ma si cerca solo di applicare le regole, ottenendo al conclusione come risultato di una catena di regole).
Come notazione usiamo anche che:

$$\vDash \{p\}\ C\ \{q\}$$

dove $\vDash$ indica che la tripla è **vera** (si parla quindi di *semantica*, riferendosi al significato).
Dato che si ha **completezza** e **correttezza** dell'apparato deduttivo si ha hanno due situazioni.

- *ogni tripla **derivabile** è anche **vera** in qualsiasi interpretazione*
- *ogni tripla **vera** vorremmo fosse anche **derivabile** e il discorso verrà approfondito in seguito per la logica di Hoare*

$\vdash$ può avere a pedice una sigla per la regola rappresentata, ad esempio $\vdash_{ass}$ per l'assegnamento.

Esempio 5. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{y \geq 0\}\ x := 2 \cdot y + 1\ \{x > 0\}$$

uso la regola di assegnamento *(che non ha premesse) partendo dalla postcondizione. Sostituisco e ottengo:*

$$\vdash_{ass} \{2 \cdot y + 1 > 0\}\ x \coloneqq 2 \cdot y + 1\ \{x > 0\}$$

Procedo usando la regola di implicazione *(sapendo che* $y \geq 0 \implies 2y+1 > 0$*):*

$$\vdash_{impl} \{y \geq 0\}\ x \coloneqq 2 \cdot y + 1\ \{x > 0\}$$

Dimostrando quindi che la tripla è ***vera****.*

Esempio 6. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{z > 0\}\ x \coloneqq (y \cdot z) + 1\ \{x > 0\}$$

e vediamo che la tripla non è valida in quanto y potrebbe essere negativo e non portare alla positività di x. Formalmente cerchiamo un controesempio cercando di non soddisfare la postcondizione. Scelgo in memoria $z = 1$ *e* $y = -2$*. Abbiamo fatto quindi un ragionamento semantico. Provo a anche sintatticamente e giungo a:*

$$\vdash \{(y \cdot z + 1) > 0\}\ x \coloneqq (y \cdot z) + 1\ \{x > 0\}$$

che non è vero, quindi non posso proseguire.

Esempio 7. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{s = x^i\}\ i \coloneqq i + 1,\ s \coloneqq s \cdot x\ \{s = x^i\}$$

anche se uguali precondizione e postcondizione fanno riferimento a due momenti della memoria diversi e quindi i valori saranno diversi.
Abbiamo a che fare con un ***invariante*** *in quanto la formula non varia tra precondizione e postcondizione anche se i valori saranno diversi (in mezzo al processo posso violare comunque l'invarianza).*
Ragioniamo in modo puramente sintattico. Dobbiamo applicare due volte l'assegnamento (e spesso serve dopo anche la regola di implicazione) e una volta la sequenza. Anche qui partiamo dalla postcondizione risalendo via via alle precondizioni:

$$\vdash_{ass} \{sx = x^i\}\ s \coloneqq s \cdot x\ \{s = x^i\}$$

Ho creato quindi la condizione intermedia tra i due assegnamenti e proseguendo ho:

$$\vdash_{ass} \{sx = x^{i+1}\}\ i \coloneqq i + 1\ \{sx = x^i\}$$

per transitività si può scrivere:

$$\{sx = x^{i+1}\}\ i := i + 1,\ s := s \cdot x\ \{s = x^i\}$$

ma non coincide con quanto voglio dimostrare. Ragiono quindi in modo algebrico. In $\{sx = x^{i+1}\}\ i := i + 1\ \{sx = x^i\}$ *ho infatti:*

$$\{s\cancel{x} = x^{i+\cancel{1}}\} \to \{sx = x^i\},\ se\ x \neq 0$$

e quindi la formula iniziale è dimostrata.

Esempio 8. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{\top\}\ if\ x < 0\ then\ y := -2 \cdot x\ else\ y := 2 * x\ endif\ \{y \geq 0\}$$

Diciamo che C *rappresenta* $y := -2 \cdot x$ *e* D *rappresenta* $y := 2 * x$ *per praticità. Indichiamo la condizione booleana* $x < 0$ *con* B*. Tutta la condizione di scelta la chiamiamo* S *Quindi avremo:*

$$\{\top\}\ if\ B\ then\ C\ else\ D\ endif\ \{y \geq 0\}$$

che in modo ancora più compatto sarebbe:

$$\{\top\}\ S\ \{y \geq 0\}$$

Applico quindi la regola di derivazione al primo caso: *Ho quindi:*

$$\{\top \wedge x < 0\}\ C\ \ y \geq 0$$

che è uguale a:

$$\{x < 0\}\ C\ \ y \geq 0$$

Procedo ora con l'assegnamento:

$$\vdash_{ass} \{-2 \cdot x \geq 0\}\ y := -2 \cdot x\ \{y \geq 0\}$$

che però equivale algebricamente a:

$$\vdash_{ass} \{x \leq 0\}\ y := -2 \cdot x\ \{y \geq 0\}$$

ma siccome $x < 0 \implies x \leq 0$ *uso la regola dell'implicazione:*

$$\vdash_{impl} \{x < 0\}\ y := -2 \cdot x\ \{y \geq 0\}$$

Passo al secondo caso:

$$\{\top \wedge x \geq 0\}\ D\ y \geq 0$$

che è uguale a:

$$\{x \geq 0\}\ D\ y \geq 0$$

Procedo ora con l'assegnamento:

$$\vdash_{ass} \{2 \cdot x \geq 0\}\ y \coloneqq 2 \cdot x\ \{y \geq 0\}$$

che però equivale algebricamente a:

$$\vdash_{ass} \{x \geq 0\}\ y \coloneqq -2 \cdot x\ \{y \geq 0\}$$

e quindi è dimostrabile che:

$$\vdash \{\top\}\ S\ \{y \geq 0\}$$

Iterazione

Definizione 17. *Definiamo:*

- ***correttezza parziale****, dove la tripla viene letta supponendo a priori che l'esecuzione termini*
- ***correttezza totale****, dove la tripla viene letta dovendo anche dimostrare che l'esecuzione termini. Si procede quindi prima dimostrando la correttezza parziale aggiungendo poi la dimostrazione per l'esecuzione finita*

Definizione 18. *La sesta regola è quella relativa all'iterazione. Abbiamo quindi:*

$$\{p\}\ \textit{while}\ B\ \textit{do}\ C\ \textit{endwhile}\ \{q\}$$

ma non posso determinare a priori quante volte si eseguirà C, che modifica lo stato della memoria. Per comodità *while B do C endwhile* *la chiameremo W, quindi in modo compatto abbiamo:*

$$\{p\}\ W\ \{q\}$$

Partiamo con la correttezza parziale supponendo a priori che l'esecuzione termini. Si ha quindi che in q sicuramente B è falsa, altrimenti non si avrebbe terminazione. Quindi in realtà abbiamo:

$$\{p\}\ W\ \{q \wedge \neg B\}$$

Ipotizziamo che all'inizio B sia vera, quindi la precondizione sarà $\{i \wedge B\}$, con i rappresentante una nuova formula. In questi stati eseguiremo C. Suppongo di poter derivare, tramite C eseguito una sola volta, nuovamente i:

$$\{i \wedge B\}\ C\ \{i\}$$

Se vale questa tripla significa che i è un ***invariante*** *per C e quindi i viene chiamata* ***invariante di ciclo****. Nello stato raggiunto dopo C la condizione B può essere valida o meno. Qualora valga dopo la singola esecuzione di C allora avrei ancora $\{i \wedge B\}$ e dovrei eseguire nuovamente C e dopo varrà ancora i sicuramente e bisogna ristudiare B per capire come procedere, in quanto i resterà vera per qualsiasi numero di iterazioni di C. Si ha che i resterà vera, teoricamente, anche una volta "usciti" dal ciclo. Qualora non valga B si ha che:*

$$\{i \wedge \neg B\}\ W\ \{i \wedge \neg B\}$$

(dove si nota che i resta vera ma B impedisce di tornare nel ciclo).
Si ha quindi che:

$$\frac{\{i \wedge B\}\ C\ \{i\}}{\{i\}\ W\ \{i \wedge \neg B\}}$$

che è la ***regola dell'iterazione****. Scritta in modo completo:*

$$\frac{\{i \wedge B\}\ C\ \{i\}}{\{i\}\ \textit{while } B \textit{ do } C \textit{ endwhile } \{i \wedge \neg B\}}$$

Una nozione più forte di ***invariante*** *può essere espressa dicendo che*

$$\{i\}\ C\ \{i\}$$

che si differenzia da quella di ***invariante di ciclo*** *(dove mi interessa sapere che un invariante sia vero prima del ciclo anche se questa non è un proprietà intrinseca degli invarianti):*

$$\{i \wedge B\}\ C\ \{i\}$$

Ricordiamo che stiamo dando per scontata la ***terminazione*** *tramite la* ***correttezza parziale****.*
Facciamo qualche osservazione:

- *nella precondizione della conclusione non si ha B, in quanto il corpo dell'iterazione può anche non essere mai eseguito*

- *data un'istruzione iterativa posso avere più di un invariante. Si ha inoltre che ogni formula iterativa ha l'**invariante banale** $i = \top$. Possiamo avere anche una formula in cui compaiono variabili che non sono modificate della funzione iterativa e quindi l'intera formula è un invariante di ciclo. Studiamo quindi gli invarianti "più utili"*

- *nei casi pratici non consideriamo ovviamente iterazioni isolate ma iterazioni inserite in un programma. In questi casi quindi la scelta di un invariante adeguato dipende sia dall'iterazione che dall'intero contesto.*

  Esempio 9. *Ho un programma su cui voglio dimostrare:*

  $$\{p\}\ C;\ W;\ D\ \{q\}$$

  con W iterazione e C, D comandi.
  Spezziamo quindi il programma per fare la dimostrazione, tramite la regola della sequenza:

  $$\{p\}\ C\ \{r\}\ W\ \{z\}\ D\ \{q\}$$

  r sarà quindi la precondizione dell'iterazione W che porterà a z che sarà precondizione di D.
  Sapendo che la regola di derivazione per W è:

  $$\{i\}\ W\ \{i \wedge \neg B\}$$

  Confronto la tripla con la "catena" sopra espressa. Si nota che r dovrà implicare l'invariante per W.

Esempio 10. *Vediamo quindi un esempio completo.*
Si prenda il seguente programma:

Listing 3 Programma P

```
i := 0; s := 1;
while i < N do
  i := i + 1;
  s := s * x;
endwhile
```

Per comodità chiamo A i due assegnamenti iniziali, W l'iterazione e C il corpo dell'iterazione.
Ci proponiamo di derivare:

$$\{N \geq 0\}\ P\ \{s := x^N\}$$

x e N sono variabili, nella realtà costanti non venendo mai modificate da P, e il loro valore fa parte dello stato della memoria.
Concentriamoci in primis sull'iterazione cercando un invariante. Una strategia semplice è quella di simulare i primi passi di una esecuzione. Supponendo N comunque non nullo abbiamo, seguendo le due variabili i e s nelle varie iterazioni (procedendo verso destra nella tabella), procedendo in modo simbolico per s (usando quindi il simbolo x direttamente):

i	0	1	2	3	...
s	1	x	x^2	x^3	...

Quindi $s = x^i$ è il nostro invariante. Dimostro questa ipotesi è vera, dimostrando:

$$\{s = x^i \wedge i < N\}\ C\ \{s = x^i\}$$

sapendo che essa è la premessa alla regola di iterazione.
Procedo quindi con la dimostrazione, avendo l'assegnamento:

$$\vdash_{ass} \{sx = x^{i+1}\}\ C\ \{s = x^i\}$$

infatti i due assegnamenti sono indipendenti*, permettendo di applicare in una sola volta due regole di assegnamento.*
Sapendo che $sx = x^{i+1} \implies s = x^i$. Ho quindi mostrato che:

$$\vdash \{s = x^i\}\ C\ \{s = x^i\}$$

*dimostrando che ho effettivamente l'invariante $s = x^i$ (e nel dettaglio abbiamo trovato un **invariante forte**, che lo è a priori rispetto alla condizione booleana B).*
Vogliamo però ottenere come precondizione $\{s = x^i \wedge i < N\}$. Sapendo però che $s = x^i \wedge i < N$ è una regola "più forte" di $s = x^i$ (se è vera la prima sicuramente è vera anche la seconda) posso usare l'implicazione e dire che:

$$\{s = x^i \wedge i < N\} \implies \{s = x^i\}$$

quindi:

$$\vdash_{impl} \{s = x^i \wedge i < N\}\ C\ \{s = x^i\}$$

posso quindi applicare la regola dell'iterazione avendo la premessa corretta e ottenendo quindi:

$$\vdash_{iter} \{s = x^i\}\ W\ \{s = x^i \wedge i \geq N\}$$

ma la postcondizione non ci sta implicando $s = x^N$, per avere:

$$\{N \geq 0\}\ P\ \{s := x^N\}$$

bisogna quindi ***rafforzare*** *l'invariante, in modo che l'invariante implichi che alla fine dell'esecuzione $i = N$.*
Procediamo quindi con una nuova ipotesi, cercando di dimostrare che $i \leq N$ è un invariante:

$$\{i \leq N \wedge i < N\}\ C\ \{i \leq N\}$$

Questa precondizione contiene una formula che è più restrittiva dell'altra. Procedo con l'assegnamento (CAPIRE QUESTA PARTE):

$$\vdash_{ass} \{i + 1 \leq N\}\ C\ \{i \leq N\}$$

ma sapendo che $i < N \implies i + 1 \leq N$ ottengo:

$$\vdash_{impl} \{i < N\}\ C\ \{i \leq N\}$$

Dimostrando quindi che è un invariante.
Passo quindi all'iterazione:

$$\vdash_{iter} \{i \leq N\}\ W\ \{i \leq N \wedge i \geq N\}$$

e la postcondizione è uguale a dire che $i = N$.
Considerando quindi i due invarianti ottengo:

$$\vdash_{iter} \{s = x^i \wedge i \leq N\}\ W\ \{s = x^i \wedge i = N\}$$

Dobbiamo però dimostrare anche la parte degli assegnamenti iniziali:

$$\{N \geq 0\}\ A\ \{s = x^i \wedge i \leq N\}$$

Bisogna infatti vedere se anche le condizioni iniziali permettono all'invariante doppio di restare tale.
Applico quindi due volte l'assegnamento (partendo dal fondo); il primo con $s := 1$:

$$\vdash_{ass} \{1 = x^i \wedge i \leq N\}\ s := 1\ \{s = x^i \wedge i \leq N\}$$

Passo quindi a $i := 0$ seguendo il nuovo ordine delle condizioni:

$$\vdash_{ass} \{1 = x^0 \wedge 0 \leq N\}\ i := 0\ \{1 = x^i \wedge i \leq N\}$$

Applico quindi la sequenza con la prima parte (sapendo $1 = x^0$ è sempre vero):

$$\vdash_{seq} \{N \geq 0\}\ A\ \{s = x^i \wedge i \leq N\}$$

completando la dimostrazioni.
Da notare solo che x deve essere non nullo.

Altri esempi sono presenti sulla pagina del corso.

4.2.2 Correttezza totale

Analizziamo ora il caso in cui non si abbia certezza di terminazione di un'iterazione:

$$\{p\} \text{ while } B \text{ do } C \text{ endwhile } \{q\}$$

Distinguiamo i due tipi di correttezza, dal punto di vista della derivabilità, tramite:

$$\vdash^{parz} \{p\} \text{ C } \{q\}$$

$$e$$

$$\vdash^{tot} \{p\} \text{ C } \{q\}$$

Dal punto di vista semantico non ha invece senso distinguere di due casi e quindi si ha solo:

$$\vDash \{p\} \text{ C } \{q\}$$

In quanto dal punto di vista semantico o si ha terminazione o non si ha, non si hanno casistiche differenti a seconda di correttezza totale o parziale.
Bisognerà quindi dimostrare la terminazione.
Studiamo il caso semplice dove:

$$W = \text{ while } B \text{ do } C \text{ endwhile}$$

Cerchiamo un'espressione aritmetica E, dove compaiono le variabili del programma, costanti numeriche e operazioni aritmetiche. Cerchiamo anche un **invariante di ciclo** i (che quindi è una formula) per W. E e i devono soddisfare due condizioni:

1. $i \implies E \geq 0$, quindi se vale i allora l'espressione aritmetica ha valore ≥ 0 (il valore di E lo ottengo eseguendo le operazioni sulle costanti e sui valori, in quello stato della memoria, delle variabili)

2. $\vdash^{tot} \{i \wedge B \wedge E = k\} \text{ C } \{i \wedge E < k\}$, dove nella precondizione abbiamo la congiunzione logica tra l'invariante i, la condizione di ciclo B e tra $E = k$, avendo prima assegnato a k il valore effettivo di E nello stato in cui iniziamo a computare C. k quindi non deve essere una variabile del programma e non deve apparire in C (rappresentante il corpo della singola iterazione). Se vale la condizione 1 e l'invariante sappiamo che $E \geq 0 \implies k \geq 0$. La postcondizione è formata dall'invariante i e dal fatto che il valore di E dopo l'esecuzione sia strettamente minore di k (che è il valore di E prima dell'esecuzione). In pratica E decresce ad ogni singola iterazione, ovvero ad ogni esecuzione di C.

La notazione $\vdash^{tot}$ serve a escludere che C abbia altri cicli annidati (portando a dover dimostrare che anche i cicli interni terminano). Per praticità quindi supponiamo di non avere cicli interni (stiamo partendo dal ciclo più interno)

Spesso si è visrto che E si ricava da B. Se B è della forma $x > y$ allora E sarà $x - y$. In caso di $<$, $\leq$ e $\geq$ ricondursi a $>$. Si specifica che non è una regola formale ma si è visto che si fa praticamente sempre così.
In pratica, nella seconda condizione, uso E per concludere che una singola esecuzione dell'iterazione mi porta in uno stato in cui vale l'invariante, dove può valere o meno B (che potrebbe portare all'uscita dall'iterazione) ma in cui E ha un valore diverso, minore a quello di partenza ma mai minore di 0 per la prima condizione (in quanto l'invariante è sempre valido). Quindi, ad un certo punto, E raggiungerà il valore minimo e in quel momento o B è falsa o si ha una contraddizione (E dovrebbe diventare negativo), quindi si ha la terminazione.
Quindi se abbiamo dimostrato le due condizioni posso dire:

$$\vdash^{tot} \{i\}\ \mathrm{W}\ \{i \wedge \neg B\}$$

che è anche la conclusione della regola di derivazione dell'iterazione in caso di correttezza parziale.
Possiamo anche osservare due cose:

1. E non è una formula logica ma un'espressione aritmetica con valore numerico ($E \geq 0$ è una formula logica)

2. qualora si abbia $E = 0$ non si hanno problemi, 0 è solo un esempio, potremmo riscrivere la prima condizione come $E \geq n$, con n qualsiasi numero intero, basta che sia ben definito per poter permettere la ripetizione finita delle iterazioni

Chiameremo questa espressione aritmetica è **variante**.
L'invariante della correttezza totale non è sempre quello di quella parziale, magari è uguale, magari diverso o anche uguale solo in parte

Esempio 11. *Vediamo un esempio chiarificatore.*
Dato il programma (una sorta di conto alla rovescia):

Listing 4 Programma P

```
while x > 5 do
  x := x - 1;
endwhile
```

studio la correttezza totale della tripla:

$$\{x > 5\}\ P\ \{x = 5\}$$

Innanzitutto cerco un variante E, che decresce ad ogni singola iterazione è un invariante i come descritti sopra, quindi che, qualora ci sia l'invariante i allora $E \geq 0$.
Un primo invariante "banale" è $i = x \geq 5$.
In merito ad E notiamo che nel corpo dell'iterazione abbiamo un solo comando, dove il valore della variabile x viene decrementato, quindi un primo candidato per E potrebbe essere proprio x.
Per comodità scegliamo però come variante $x - 5$ per avere una corrispondenza diretta con l'invariante $x \geq 5$, essendo derivato dallo stesso invariante (basandoci in primis sulla prima condizione).
***Non sempre posso ottenere una corrispondenza tra variante e invariante**.*
Presi questo invariante e questo variante verifichiamo le due condizioni:

1. $x \geq 5 \implies x - 5 \geq 0$ *è ovviamente corretto perché si ottiene che* $x \geq 5 \implies x \geq 5$ *(infatti questo è il motivo per cui si è scelto* $x - 5$ *come variante)*

2. $\vdash^{tot} \{x \geq 5 \land x > 5 \land x - 5 = k\}\ x := x - 1\ \{x \geq 5 \land x - 5 < k\}$
   Applico quindi la regola dell'assegnamento (che si applica anche per la correttezza totale) e ottengo la precondizione per la postcondizione $\{x \geq 5 \land x - 5 < k\}$*:*

   $$\{x - 1 \geq 5 \land x - 1 - 5 < k\} = \{x \geq 6 \land x - 6 < k\}$$

   che però è diversa da $\{x \geq 5 \land x > 5 \land x - 5 = k\}$.
   Cerco quindi di applicare la regola dell'implicazione. Riguardando la precondizione originale noto che $x > 5$ *è "più forte" di* $x \geq 5$ *e quindi quest'ultimo può essere trascurato. Ricordando che stiamo lavorando su valori interi si ha che* $x > 5 \implies x \geq 6$*. Inoltre, sempre per il fatto che lavoriamo su numeri interi,*
   $x - 5 = k \implies x - 6 < k$ *e quindi:*

   $$\{x \geq 5 \land x > 5 \land x - 5 = k\} \implies \{x \geq 6 \land x - 6 < k\}$$

Abbiamo quindi dimostrato che il programma termina e vale la tripla iniziale. Qualora avessimo dovuto dimostrare la correttezza parziale si sarebbe dovuto procedere riutilizzando lo stesso invariante e procedendo studiando la negazione di B, ovvero $x \leq 5$ (che insieme danno la postcondizione).

Esempio 12. *vediamo un esempio non si ha alcuna variabile che decrementa nel corpo dell'iterazione:*

Listing 5 Programma P

```
while x < 5 do
  x := x + 1;
endwhile
```

studio la correttezza totale della tripla:

$$\{x < 5\}\ P\ \{x = 5\}$$

Dal punto di vista della correttezza parziale ragiono come al solito mentre per la correttezza totale la situazione è un po' diversa.
Ovviamente non posso usare x come variante ma posso sicuramente usare, per esempio, $-x$, che sicuramente decresce visto che x cresce, in entrambi i casi ad ogni iterazione. Riprendendo l'esempio sopra quindi posso usare $x \leq 5$ come invariante e, al posto di $-x$ che comunque andrebbe bene, secondo una ragionamento simile allo scorso esempio, $5 - x$ come variante (si nota che anche questo E è ricavabile da i, anche se questo non è sempre attuabile). Il resto della dimostrazione è analoga all'esempio precedente.

Correttezza e Completezza

In generale quando si sviluppa una logica, con un apparato deduttivo e un'interpretazione delle formule, siamo interessati a due proprietà generali della logica e dell'apparato deduttivo:

1. **correttezza**, ovvero il fatto che tutto quello che si può derivare con l'apparato deduttivo è effettivamente vero, ovvero $\vdash \implies \models$. Quindi se una tripla è derivabile è anche vera

2. **completezza**, ovvero il fatto che l'apparato deduttivo sia in grado di derivare tutte le formule vere (ovvero tutte le triple) vere. Si ha quindi $\models \implies \vdash$

Per la logica di Hoare vale la correttezza ma vale una proprietà di completezza è *relativa* in quanto nel corso di una dimostrazione di completezza occorre a volte usare deduzioni della logica proposizionale ma tali formule parlano anche di operazioni aritmetiche. Si ha che l'aritmetica può essere formalizzata attraverso un linguaggio logico con degli assiomi e delle regole di inferenza. Si hanno però i **teoremi di incompletezza dell'aritmetica** di Gödel che

dicono che l'aritmetica come teoria matematica è incompleta in quanto ci sono formule scritte nel linguaggio dell'aritmetica che sono vere ma non sono dimostrabili. Questa incompletezza si riverbera sulla logica di Hoare, che comunque, dal punto di vista deduttivo sulle triple, è completa.

4.2.3 Precondizione più debole

Dato un comando C e una formula q che interpretiamo come post condizione di C. Si cerca p tale per cui:

$$\vdash \{p\}\ \mathrm{C}\ \{q\}$$

Sia valida e quindi vera.
Ovviamente il caso interessante è quello totale.

Esempio 13. *Suppongo di avere come comando un singolo assegnamento* $y := 2 \cdot x - 1$ *e come postcondizione* $\{y > x\}$.
Cerco possibili precondizioni che, dopo l'assegnamento, portino a quella postcondizione. Ovviamente si avrebbero infiniti valori di x *che consentono di arrivare alla postcondizione (nonché altrettanti che non lo permettono).*

Si cerca quindi la "migliore" precondizione per arrivare ad una data postcondizione ma per farlo ci serve un criterio di confronto.
Introduciamo quindi alcuni simboli e nozioni utili:

- V è l'insieme della variabili di C
- $\Sigma = \{\sigma |\, \sigma : V \to \mathbb{Z}\}$ è l'insieme degli stati della memoria (ricordando che posso avere solo valori interi nel nostro linguaggio)
- Π è l'insieme di tutte le formule sull'insieme V
- $\sigma \vDash p$ significa che la formula p è vera nello stato σ e si ha che $\vDash \subseteq \Sigma \times \Pi$, con $\vDash$ che indica la veridicità di una formula in uno stato
- $t(\sigma) = \{p \in \Pi |\, \sigma \vDash p\}$ come la funzione che assegna ad uno stato l'insieme delle proposizioni, ovvero tutte le formule, che sono vere in σ
- $m(p) = \{\sigma \in \Sigma |\, \sigma \vDash p\}$ come la funzione che associa ad una formula l'insieme di tutti gli stati che soddisfano la formula

Tra le due funzioni finali si ha una sorta di "dualità" e agiscono in modo speculare tra loro. Se, partendo da Σ, applico $t(\sigma)$ e scopriamo che un certo stato p appartiene a Π e a tale p associamo l'insieme dato da $m(p)$ si avrà che $\sigma \in m(p)$.
Si possono fare altre osservazioni su queste due funzioni.
Prendiamo due formule p e q e cerco di capire se è possibile avere $m(p) = m(q)$. La risposta è positiva e si parla, in caso, di *equivalenza tra formule*. Fissati due stati σ_1 e σ_2 mi chiedo se possano appartenere allo stesso insieme di formule. In questo caso la risposta è negativa, in quanto se i due stati sono distinti allora ci deve essere almeno una variabile x che ha valore diverso nei due stati ma allora è facile trovare una formula che separa i due stati e tale formula potrà essere vera solo in uno dei due stati.
Dato $S \subseteq \Sigma$ e $F \subseteq \Pi$ (ragiono quindi su sottoinsiemi) si ha che:

- $t(S) = \{p \in \Pi | \forall \sigma \in S,\ \ \sigma \vDash p\} = \bigcap_{\sigma \in S} t(\sigma)$ (ragiono quindi su tutti gli stati di S e quindi sulle formule che sono vere in tutti questi stati)
- $m(F) = \{\sigma \in \Sigma | \forall p \in F\ \ \sigma \vDash p\} = \bigcap_{p \in \Pi} m(p)$ (ragiono quindi su tutte le formule di Π e quindi su tutti gli stati in devono essere soddisfatte tutte queste formule)

Abbiamo quindi esteso il dominio delle due funzioni a sottoinsiemi di stati e sottoinsiemi di formule.
Si può anche dimostrare che $S \subseteq m(t(s))$ e che $F \subseteq t(m(F))$. Inoltre, dati $A \subseteq B$, si può dimostrare che $m(B) \subseteq m(A)$ (se ho un insieme più grande di formule ho meno stati che le soddisfano tutte).
Si ha un rapporto tra la logica proposizionale (coi suoi connettivi logici) e l'insieme di stati (p e q sono formule di Π):

- $m(\neg p) = \Sigma \setminus m(p)$, quindi il complemento insiemistico di p
- $m(p \vee q) = m(p) \cup m(q)$, quindi all'unione dei due insiemi
- $m(p \wedge q) = m(p) \cap m(q)$ (da verificare) quindi all'intersezione dei due insiemi

L'implicazione merita un discorso a parte in quanto ha due "nature":

- **operatore logico**, che comporta che $p \implies q = \neg p \vee q$ e in tal caso si ha, in linea a quanto detto per gli altri connettivi:

$$m(p \implies q) = m(\neg p) \cup m(q)$$

- **relazione tra formule**, dove se p implica q (in tutti in casi in cui è vera p è vera q) allora $m(p) \subseteq m(q)$, intendendo che q è "più debole" (ovvero come formula ci da una conoscenza inferiore essendo q corrispondente ad un insieme più grande di stati) di p. Si ha a che fare con una relazione algebrica tra le due formule. Più "debole" non significa comunque "peggiore"

Posso quindi capire come definire la precondizione migliore per ottenere una tripla valida, insieme al comando C e alla postcondizione.
Un modo è quello di definire la migliore precondizione come la precondizione più debole p che presa come precondizione forma una tripla valida:

$$\vdash \{p\}\ \mathrm{C}\ \{q\}$$

se p è la precondizione più debole allora corrisponde al più grande insieme di stati tale che la tripla sia valida.
Questa è una scelta ragionevole perché è la precondizione che impone meno vincoli sullo stato iniziale, infatti determina tutti gli stati iniziali che garantiscono il raggiungimento della postcondizione. È sempre possibile calcolare la precondizione più debole.
Indichiamo, fissati C comando e q formula di postcondizione, con $wp(C, q)$ la precondizione più debole (**weakest precondition (*wp*)**).

Teorema 2 (proprietà fondamentale della condizione più debole)**.** *Si ha che* $\vDash \{p\}\ C\ \{q\}$ *(quindi la tripla è vera) sse:*

$$p \implies wp(C, q)$$

Teorema 3. *L'**esistenza** della precondizione più debole è garantita.*
*È garantita inoltre l'**unicità** della precondizione più debole (a meno di eventuali equivalenze logiche).*

Vediamo quindi la *regola di calcolo.*
Si hanno diversi casi a seconda dei tipi di comando:

- **assegnamento:** in questo caso la precondizione più debole è quella determinata dalla regola di derivazione introdotta per la correttezza parziale. Quindi dato un assegnamento del tipo $x := E$ e una postcondizione q ho che la precondizione più debole si ottiene sostituendo in q ogni occorrenza di x con l'espressione E, ottenendo quindi:

$$\{q[E/x]\}$$

- **sequenza:** in questo caso, se ho la sequenza di due comando C_1 e C_2 allora la precondizione più debole si calcola nel seguente modo: calcolo la precondizione più debole per C_2, ottenendo $wp(C_2, q)$, che sarà quindi la formula usata come postcondizione per calcolare la precondizione più debole di C_1, ovvero: $wp(C_1, wp(C_2, q))$. Questo non è altro che la condizione più debole della sequenza:

$$wp((C_1; C_2), q) \equiv wp(C_1, wp(C_2, q))$$

- **scelta:** in questo caso, avendo:

$$S = \text{ if } B \text{ then } C \text{ else } D \text{ endif}$$

fisso la postcondizione q e procedo come per la regola di derivazione. Separo i due casi in cui sia vera B o meno. Se B è vera devo eseguire C quindi calcolo, eseguendo C la precondizione più debole $wp(C, q)$. Qualora B non sia vera calcolo, eseguendo D, la precondizione più debole $wp(D, q)$. Nel complesso quindi, facendo la disgiunzione dei due casi, ottengo:

$$wp(S, q) \equiv (B \wedge wp(C, q)) \vee (\neg B \wedge wp(D, q))$$

- **iterazione:** in questo caso, avendo:

$$W = \text{while } B \text{ do } C \text{ endwhile}$$

È il caso più complicato.
Si ha che se $\neg B$ è nello stato iniziale dll'iterazione allora il corpo non viene eseguito (arrivando direttamente alla postcondizione), invece se vale B si avrà che W equivale a $C; W$, in quanto sicuramente almeno una volta eseguo C. Ma a questo punto potrei rifare lo stesso discorso se B è ancora vera (questo fino a che non ho $\neg B$, uscendo dal ciclo e arrivando a q), ragionando su W di $C; W$. Si arriva di fatto ad una regola ricorsiva:

$$wp(W, q) \equiv (\neg B \wedge q) \vee (B \wedge wp((C; W), q))$$

Applico quindi la regola della sequenza per la condizione più debole, arrivando a:

$$wp(W, q) \equiv (\neg B \wedge q) \vee (B \wedge wp(C, wp(W, q)))$$

Si nota che questa definizione ricorsiva ha un problema grave: non si ha un **caso base** che risolva la catena di chiamate ricorsive.

Purtroppo non si può usare il "trucco" dell'invariante come nella regola di definizione e questo comporta che non si ha un vero e proprio algoritmo unico ed effettivo per questa regola di ricerca della precondizione più debole

Si hanno dei casi estremi, ad esempio:

- precondizioni più deboli sempre false, ad esempio:

  $$wp(x := 5, x < 0) \equiv \bot$$

  ovvero se assegno a x il valore 5 non avrò mai x negativo, ovvero ho un insieme vuoto $\emptyset$ di stati che garantiscono quanto detto

- precondizioni più deboli sempre vere, ad esempio

  $$wp(x := 5, x \geq 0) \equiv \top$$

  ovvero se assegno a x il valore 5 si avrà che x è sempre positivo, quindi è vero per qualunque stato iniziale

Esempi vari

Esempio 14. *Vediamo un primo esempio semplice con solo la sequenza di due assegnamenti:*

Listing 6 Programma P

```
x := x+1;
y := y*x;
```

Dove i due assegnamenti sono rispettivamente A e B e si vuole la postcondizione $q = x < y$. Cerco quindi $wp(P, q)$.
Procediamo con la formula della sequenza (che ricordiamo procede "a ritroso").

- *Parto dal secondo assegnamento, B, e calcolo $wp(B, q)$.*
  Procedo con la regola e applico la sostituzione, ottenendo:

  $$r = wp(B, q) = x < y \cdot x$$

  quindi se $x < y \cdot x$ vale prima dell'esecuzione di B allora sicuramente si raggiunge uno stato finale dove vale la postcondizione q

- *lo stato in cui eseguiamo B, che abbiamo sopra chiamato r, è quello prodotto dall'esecuzione di A. Dobbiamo quindi calcolare $wp(A, r)$.*
  Procedo quindi ancora con l'assegnamento, ottenendo:

  $$wp(A, r) \equiv x + 1 < y(x + 1)$$

  che quindi, per composizione, è anche la precondizione più debole per P e q. Bisogna però fare dei controlli.
  Uso quindi le regole delle disequazioni:

  - *caso 1:* $x + 1 > 0 \implies y > 1$
  - *caso 2:* $x + 1 = 0 \implies 0 < 0$ *e quindi non vale q*
  - *caso 3:* $x + 1 < 0 \implies y < 0$

  Costruisco quindi la precondizione più debole prendendo la disgiunzione logica dei due casi validi:

  $$wp(P, q) \equiv (x \geq 0 \land y > 1) \lor (x < -1 \land y < 1)$$

  (sempre ricordando che lavoriamo con interi)

Esempio 15. *Vediamo un altro esempio:*

Listing 7 Programma P

```
x := x+a;
y := y-1;
```

Dove i due assegnamenti sono rispettivamente A e B e si vuole la postcondizione $q = (x = (b - y) \cdot a)$. Cerco quindi $wp(P, q)$.
Nella postcondizione ho una variabile b esterna alla sequenza di interesse fissato a priori.
Come prima ottengo:

$$r = wp(B, q) \equiv (x = (b - y + 1) \cdot a) = (x = (b - y) \cdot a)$$

e quindi:

$$wp(A, r) \equiv (x + a = (b - y) \cdot a + a)$$

quindi $x = (b - y) \cdot a$ è un invariante ed è anche la postcondizione, Quindi q è un invariante, accezione più forte del termine, per P.

Esempio 16. *Vediamo un altro esempio:*

Listing 8 Programma P

```
if y = 0 then
  x := 0;
else
  x := x * y;
endif
```

Dove i due comandi sono rispettivamente C e D e la condizione booleana è B. Come postcondizione voglio $q = (x = y)$.
Per il caso della scelta bisogna calcolare i due casi e usare poi la disgiunzione tra essi.
Calcolo quindi:

1. *il caso in cui vale B: $wp(C, q) = (0 = y)$, usando l'assegnamento*
2. *il caso in cui vale $\neg B$: $wp(D, q) = (x \cdot y) = y$, usando assegnamento. La formula però comporta che ho due casi possibili: $y = 0 \vee x = 1$ (il secondo appunto se ho $y \neq 0$)*

Applico quindi la regola della scelta, ottenendo:

$$wp(P, q) \equiv (y = 0 \wedge y = 0) \vee (y \neq 0 \wedge (y = 0 \vee x = 1))$$

Semplificando si ha che:

$$wp(P, q) \equiv (y = 0) \vee (x = 1 \wedge y \neq 0)$$

Esempio 17. *Vediamo un altro esempio:*

Listing 9 Programma P

```
while x > 0 do
  x := x - 1
endwhile
```

Con la condizione B e il comando C, nel corpo dell'iterazione W. Come postcondizione si vuole $q = (x = 0)$.
Seguendo le modalità discusse in merito alle tecniche relative al calcolo della precondizione più debole per l'iterazione, vediamo che:

$$(\neg B \wedge q) \equiv (x \leq 0 \wedge x = 0) \equiv (x = 0)$$

Per comodità chiamo $wp(W, q)$ r.
Dobbiamo ora studiare $B \wedge wp(C, wp(W, q))$ *che quindi per noi è, per l'alias appena definito,* $B \wedge wp(C, r)$*:*

$$B \wedge wp(C, r) \equiv x > 0 \wedge r[x - 1/x]$$

usando quindi l'assegnamento e quindi si ha che:

$$B \wedge wp(C, r) \equiv (x > 0) \wedge (x - 1 = 0 \vee (x - 1 > 0 \wedge r[x - 2/x]))$$

Siamo quindi in piena ricorsione.
Allo stato attuale ho quindi (usando ldots per non dover riscrivere tutto):

$$(\neg B \wedge q)(B \wedge wp(C, r)) \equiv (x = 0 \vee (x > 0 \wedge (\ldots)))$$

in realtà, andando avanti, si otterrebbe uno sviluppo regolare che porterebbe alla formula infinita:

$$(x = 0 \vee x = 1 \vee x = 2 \vee \ldots) \equiv x \geq 0$$

e quindi al precondizione più debole diventa:

$$wp(W, q) = (x \geq 0)$$

Si nota quindi come i casi di iterazione si risolvono solo con tecniche "non rigorose" ad hoc.

Estensioni del linguaggio

Abbiamo, in conclusione, sviluppato una tecnica di studio basata su un linguaggio di programmazione molto semplificato. Innanzitutto potremmo estendere il linguaggio usato, aggiungendo:

- il **do-while**:

  do C while B endwhile

  che nella realtà corrisponde a:

  C; while B do C endwhile

- il **repeat-until**:

  repeat C until B endrepeat

  che nella realtà corrisponde a:

  C; while not B do C endwhile

- il **ciclo for**:

  $$\text{for}(D;\ B;\ F)\ C\ \text{endfor}$$

  sempre convertendolo in un while:

  $$D;\ \text{while}\ B\ \text{do}\ C;\ F\ \text{endwhile}$$

- **procedure, metodi e funzioni**
- **array**, anche se solo in lettura se vogliamo applicare la logica di Hoare come l'abbiamo vista

Un altro limite è dato dal fatto che abbiamo usato solo il tipo intero, ma possiamo potenzialmente aggiungere anche gli altri senza cambiare le basi della logica introdotte.

4.2.4 Logica di Hoare per sviluppare programmi

Possiamo vedere le logiche di Hoare come **contratti** tra chi scrive il programma e l'utente. In questo contesto l'utente commissiona il programma specificando la postcondizione e chiede al programmatore di garantire che tale postcondizione venga garantita al termine dell'esecuzione. Per garantire la postcondizione q deve essere vera la precondizione p.
Vediamo un esempio.

Esempio 18. *Ci proponiamo di calcolare la radice quadrata intera di un certo $k \geq 0$, approssimando per difetto.*
Alla fine del programma voglio il risultato nella variabile x, quindi voglio:

$$\{k \geq 0\}\ P\ \{0 \leq x^2 \leq k < (x+1)^2)\}$$

con $(x+1)^2$ per la correttezza dell'approssimazione.
Ci si propone di fare il calcolo per approssimazioni successive, partendo da un valore più piccolo di quello corretto.
Possiamo spaccare q in:

$$0 \leq x^2 \leq k \quad e \quad k < (x+1)^2$$

la prima possiamo considerarla come invariante (anche solo $x^2 \leq k$).

All'inizio possiamo pensare che x dipenda da k per una certa funzione E. Si ha quindi un prototipo del genere (con un certo B, condizione booleana, e un certo F, incremento nel corpo, dipendenti da x e k):

Listing 10 Programma P

```
x := E(x);
while B(x,k) do
  x := F(x,k);
endwhile
```

Alla fine deve valere $x^2 \leq k \wedge \neg B$ per la regola dell'iterazione.
Come B posso porre $(x+1)^2 \leq k$ (ovvero la negazione della seconda parte della postcondizione, in modo da ottenere, con la negazione, q).
Si ottiene quindi:

Listing 11 Programma P

```
x := 0;
while (x+1)^2 <= k do
  x := x+1;
endwhile
```

Bisognerebbe comunque dimostrare invariante e terminazione per validare il programma.

Supponiamo un programma del tipo:

$$\{p\} \text{ A;W;C } \{q\}$$

che si risolve schematicamente con:

$$\{p\} \text{ A } \{inv\} \text{ W } \{inv \wedge \neg B\} \text{ C } \{q\}$$

Questo schema usa i cosiddetti **invarianti costruttivi**.

4.2.5 Ultime considerazioni

Vediamo qualche ultima considerazione sulla logica di Hoare.

- alcune applicazioni pratiche della logica di Hoare:
  - *java modelling language*, un linguaggio di specificazione scritto in Java, che permette di fare *design by contract* stabilendo delle precondizioni e delle postcondizioni

  - *Eiffel, programming by contract*
  - *assert.h* in C

- alcune applicazioni teoriche della logica di Hoare:
  - *semantica del programmi*, ovvero lo studio di cosa significa un programma, tramite:
    * *semantiche operazionali*, dove al programma viene associata una macchina astratta e, dato un programma viene associata una computazione sulla macchina astratta (come ad esempio la **Macchina di Turing** o le **Macchine a Registri**)
    * *semantiche assiomatiche*, dove vediamo, mano a mano che viene eseguito il programma, le asserzioni che vengono verificate
    * *semantiche denotazionali*, in cui un programma è visto come un trasformatore da input e output. Come una $f : I \to O$. SI basa sul **lambda calcolo**
    * *semantiche operazionali strutturate*

Si hanno quindi i due punti chiave che devono essere garantiti:

1. **terminazione** del programma
2. **composizionalità** tra più comandi per ottenere un programma

Le triple di Hoare vivono ancora nella **programmazione by contract**.

Capitolo 5

Calculus of Communicating Systems

Fino ad ora abbiamo considerato programmi sequenziali e abbiamo studiato la loro verifica, tramite le triple di Hoare. Si passa ora dal sequenziale al **concorrente**. Dati due comandi s_1 ed s_2 si ha che essi sono specificati in esecuzione concorrente con:

$$s_1|s_2$$

L'esecuzione in concorrenza può portare a diverse complicanze qualora non venga rispettato, per esempio, un certo ordine di esecuzione. Si ha quindi il **non determinismo**, potendo avere più risultati a seconda dell'ordine di esecuzione. Si perde la **composizionalità**.

Esempio 19. *Vediamo un esempio.*
Siano:

$$s_1 = \{x = V\}\ \ x = 2\ \ \{x = 2\}$$

$$s_2 = \{x = V\}\ \ x = 3\ \ \{x = 3\}$$

con V indicante un valore qualunque.
Posso avere:

$$\{x = V\}\ \ s_1|s_2\ \ \{x = 2 \vee x = 3\}$$

avendo non determinismo.
Definiamo anche:

$$s_1' = \{x = V_0\}\ \ x = 1; x = x + 1\ \ \{x = 2\}$$

e vediamo, a conferma della perdita di composizionalità e determinismo che:

$$\{x = V\}\ \ s_1'|s_2\ \ \{x = 2 \vee x = 3 \vee x = 4\}$$

Si studierà quindi la **semantica della programmazione concorrente/parallela (o dei sistemi distribuiti)**.
Il problema è stato studiato da Hoare e Milner (secondo punti di vista diversi) alla fine degli anni '70. Hoare ha introdotto un nuovo paradigma di programmazione, il paradigma **CSP (*communicating sequential processes*)**, il linguaggio macchina dei cosiddetti *transputer*. Non si ha più una memoria condivisa ma un insieme di processi ciascuno con una sua memoria privata. Si ha un'interazione tra processi tramite lo scambio di messaggi del tipo *hand-shacking*, avendo quindi la sincronizzazione, con lo scambio di informazioni. Viene fatto anche un processo particolare rappresentante la **memoria condivisa**. Avremo quindi:

$$x|s_1|s_2$$

con x che rappresenta la memoria condivisa dai due processi.
Ad oggi diversi linguaggi di programmazione adottano una "soluzione mista" rispetto a questo paradigma.
Milner propose in modo completamente indipendente da Hoare qualcosa di molto analogo. Milner propose infatti il **lambda calcolo (λ-*calcolo*)** per passare dal sequenziale al concorrente. Studia in modo approfondito la composizionalità, sfruttando la composizione tra funzioni, cercando di non perderla nel concorrente. Introduce quindi una sorta di λ-*calcolo concorrente*, introducendo il **Calculus of Communicating Systems (*CCS*)**, in cui pensa ad un calcolo algebrico per sistemi comunicanti. Adotta anche lui un paradigma che studia un sistema (non solo dal punto di vista della programmazione) formato da componenti, chiamati *processi*. Questi processi comunicano tramite lo scambio sincrono di messaggi, con il modello *hand-shacking*. Con a senza nulla indiciamo un processo generico (a potrebbe essere qualsiasi cosa) che invia e con $\overline{a}$ un processo generico che riceve. Un sistema quindi è un insieme di processi il cui comportamento è gestito da un calcolo algebrico, si punta alle **algebre di processi**, ovvero linguaggi di specifica di sistemi concorrenti che si ispirano al calcolo dei sistemi comunicanti. I messaggi di scambio corrispondono ad uno scambio di valori di variabili e questo è rappresentabile dall'algebra. I processi possono interagire anche con l'**ambiente esterno**. Dato un sistema S, si scrive:

$$S = p_1|p_2|p_3$$

se S è formato dai processi p_1, p_2 e p_3, processi che sono *interagenti* "a due a due". Ogni processo ha comunque una memoria privata.
Grazie alla comunicazione con l'*ambiente esterno* non si ha più un **sistema chiuso**. Pnueli (che creò anche le logiche temporali) e Harvel hanno definito

questi sistemi **sistemi reattivi**, per indicare che i sistemi reagiscono all'ambiente esterno.
Milner risolve il problema della *composizionalità* tramite l'uso di diverse *porte* che permettono ad un processo di comunicare con altri o con l'ambiente esterno. Quindi ogni processo può essere visto come un insieme di sottoprocessi *interagenti* che però interagiscono tramite sincronizzazione con i processi esterni tramite una porta (un sottoprocesso può comunicare con più processi esterni tramite più porte). Bisogna comunque mantenere il comportamento complessivo. Per il processo esterno è come se sostituissi il processo con cui comunica con il suo sottoprocesso. Si introduce infatti l'**equivalenza all'osservazione**, che permette di sostituire un processo s_i con s_i' se sono equivalenti rispetto all'*osservazione* ovvero sse un qualsiasi osservatore esterno non è in grado di distinguere i due processi. In questo caso *osservare* significa **interagire con il sistema** dove agisce il processo (questa è un'idea di "osservare" derivante dalla fisica moderna). Questo deve essere valido **per ogni possibile osservatore**. Se questo è garantito la sostituzione di un processo non va ad inficiare l'esecuzione complessiva, senza incorrere in *deadlock* o altre problematiche.

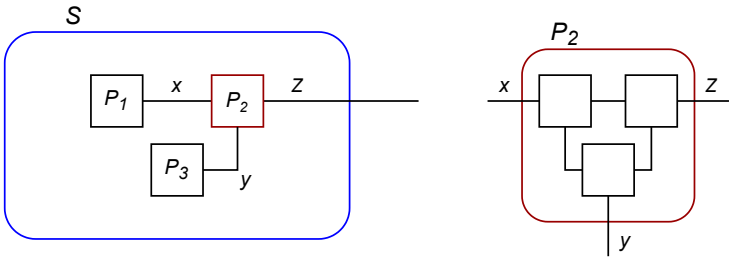


Figura 5.1: Rappresentazione stilizzata di un sistema S e del suo sottoprocesso p_2

Vedremo quindi:

- il calcolo "puro" di sistemi comunicanti CCS, definendone la semantica attraverso **Labeled Transition System, (*LTS*)** (*sistemi di transizione etichettati*), (avendo come nodi i processi e archi etichettati dalle azioni). Useremo le operazioni "+", "$\cdot$" e la ricorsione, che verranno definite tramite LTS
- l'equivalenza all'osservazione e la **bisimulazione**. Vedremo anche una tecnica di verifica della *bisimulazione*, tramite la **teoria dei giochi**

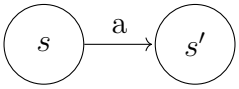


Figura 5.2: Esempio semplice di LTS

5.1 LTS

Presa una relazione binaria R su X, ovvero $R \subseteq X \times X$.
Si ha che:

- una relazione è **riflessiva** sse $\forall x \in X$ ho che $(x, x) \in R$ o, scritto diversamente, xRx ovvero ogni elemento è in relazione con se stesso
- una relazione è **simmetrica** sse ho $(x, y) \in R$ allora $(y, x) \in R$, $\forall\, x, y \in X$
- una relazione è **transitiva** se ho $(x, y) \in R \wedge (y, z) \in R$ allora ho $(x, z) \in R$

Se una relazione R è simmetrica, riflessiva e transitiva allora è una **relazione di equivalenza** e quindi posso ripartire X in classi di equivalenza:

$$[x] = \{y \in X |\ (x, y) \in R\}$$

Date due classi di equivalenza $[x_1]$ e $[x_2]$ si ha che o sono uguali (quindi tutti gli elementi di una sono nell'altra) oppure la loro intersezione è vuota.
Inoltre si ha che:

$$U_{x \in X}[x] = X$$

Si è quindi diviso X in sottoinsiemi disgiunti che coprono tutto X.
Siano R, R' e R'' relazioni binarie su X. Si ha che:
R' è la **chiusura riflessiva/simmetrica/transitiva** di R sse:

- $R' \subseteq R$
- R' è riflessiva/simmetrica/transitiva
- R' è la più piccola relazione che soddisfa i primi due puntu e quindi:

$$\forall R'' \text{ se } R \subseteq R'' \wedge R''\text{riflessiva/simmetrica/transitiva}$$

$$\text{allora } R' \subseteq R''$$

I **Labeled Transition System (*LTS*)** sono molto usati per rappresentare sistemi concorrenti. Questo modello ha origine dal modello degli automi a stati finiti, che però sono usati come riconoscitori di linguaggi.
Negli LTS non si ha l'obbligo di avere un insieme finito di stati.

Definizione 19. *Definiamo un LTS come la quadrupla:*

$$(S, Act, T, s_0)$$

dove:

- S *è un insieme, anche infinito, di stati*
- Act *è un'insieme di nomi di azioni*
- $T = \{(s, a, s')\}$*, con* $s, s' \in S$ *e* $a \in Act$ *oppure* $T \subseteq S \times Act \times A$*, è un insieme di triple rappresentanti le transizioni. Come scrittura si ha anche* $(s, a, s') \in T$ *oppure* $s \xrightarrow{a} s'$ *e quindi ho un arco etichettato con* a *tra due nodi etichettati con* s *e* s'*, come in figura 5.2*
- s_0 *stato iniziale*

La transizione $s \xrightarrow{a} s'$ *può essere estesa a* $w \in Act^*$ *avendo più azioni sse:*

- *se* $w = \varepsilon$ *allora* $s \equiv s'$
- *se* $w = a \cdot x,\ a \in Act,\ x \in Act^*$ *sse:*

$$s \xrightarrow{a} s'' \text{ e } s'' \xrightarrow{x} s'$$

Ho che $s \to s'$ *sse* $\exists a \in Act$ *t.c.* $s \xrightarrow{a} s'$.
Quindi ho:

$$\to = \cup_{a \in Act} \xrightarrow{a}$$

Ho che $s \xrightarrow{*} s'$ *sse* $\exists w \in Act^*$ *t.c.* $s \xrightarrow{w} s'$*. Si ha che:*

$$\xrightarrow{*} = \cup_{w \in Act^*} \xrightarrow{w}$$

$$\xrightarrow{*} \subseteq S \times S$$

La relazione $\xrightarrow{*}$ *è la chiusura riflessiva e transitiva della relazione* $\to$ *(tale relazione non è simmetrica), avendo sempre* $s \xrightarrow{*} s$ *ed essendo garantita la transitività.*

5.2 CCS

Definizione 20. *Per definire il **Calculus of Communicating Systems (CCS)** "puro" (astraendo l'aspetto delle strutture dati etc...) dobbiamo definire che abbiamo:*

- *K, ovvero un insieme di nomi di processi (buffer, sender,ldots) che possono anche essere simboli di un alfabeto*
- *A, ovvero un insieme di nomi di azioni, che sono o azioni di sincronizzazione con l'ambiente o le componenti del sistema (anche interne al singolo componente)*
- *$\overline{A}$, ovvero l'insieme di nomi delle coazioni contenute in A, $\forall\, a \in A\ \exists\, \overline{a} \in \overline{A}$, quindi:*
  $$\overline{A} = \{\overline{A} |\, a \in A\}$$
  ovviamente si ha che:
  $$\overline{\overline{a}} = a$$
- *$Act = A \cup \overline{A} \cup \{\tau\}$ dove $\tau \notin A$ corrisponde all'azione di sincronizzazione tra a e $\overline{a}$, ovvero la sincronizzazione è avvenuta. Le prime due sono azioni osservabili e si indica con:*
  $$\mathcal{L} = A \cup \overline{A}$$
  mentre τ non è osservabile.
  Ricordando che osservare un'azione significa poter interagire con essa.

Esempio 20. *Vediamo un esempio **NON** chiarificatore.*
Prendiamo un sistema S, scuola, che eroga una lezione $\overline{lez}$ e poi continua ciclicamente ad erogare lezioni:

$$S = \overline{lez} \cdot S$$

Lo studente ST segue la lezione e torna a studiare:

$$ST = lez \cdot ST$$

se S e ST si sincronizzano $(S|ST)$ lo studente osserva la lezione e il risultato non è più ossrvabile, essendo τ.

Definizione 21. *I processi CCS sono di fatto operazioni CCS e un sistema CCS è definito da una collezione di processi* $p \in K$ *e si avranno:*

$$p = \textit{espressione CCS}$$

e si avrà solo un'equazione $\forall\, p \in K$

Definiamo meglio i processi CCS e, ad ogni processo, l'LTS assegnato, che sarà del tipo:

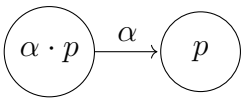
$$(processi, Act, R, p_0)$$

e, tramite le regole di inferenza (con premesse, eventualmente con condizioni e con conclusioni).
Dare un significato tramite LTS, regole di inferenza e sintassi è detto **semantica operazionale strutturale**.
Un processo CCS può essere:

- **Nil** o 0 che ha un solo stato senza transizioni

- **prefisso**, in cui si ha $\alpha \cdot p$, $\alpha \in Act$, $p \in processi$, con la regola di inferenza:

$$\frac{}{\alpha \cdot p \xrightarrow{\alpha} p}$$

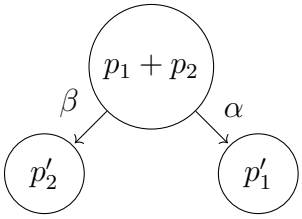


- **Somma**, $p_1 + p_2,\ p_1, p_2 \in processi$, con la regola di inferenza (il $+$ è la "scelta", come nella regex), avendo $\alpha, \beta \in Act$ e $p_1', p_2' \in processi$:

$$\frac{p_1 \xrightarrow{\alpha} p_1'}{p_1 + p_2 \xrightarrow{\alpha} p_1'}$$

oppure

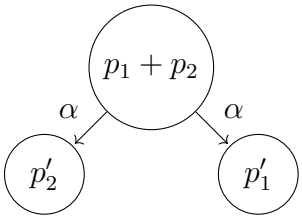
$$\frac{p_2 \xrightarrow{\beta} p_2'}{p_1 + p_2 \xrightarrow{\beta} p_2'}$$



- posso avere $\sum_{i \in I} p_i$ avendo **multiple somme di processi**:

$$\frac{p_j \xrightarrow{\alpha} p'_j}{\sum_{i \in I} p_i \xrightarrow{\alpha} p_j 0}, \; j \in J$$

e quindi se $I = \emptyset$ avrò che $\sum_{i \in I} p_i = Nil$.
Posso avere non determinismo avendo $p_1 = a \cdot p'_1$ e $p_2 = a \cdot p'_2$, avendo:



Esempio 21. *Vediamo un altro esempio. Dati:*

- $p_1 = \alpha \cdot p'_1$
- $p_2 = \beta \cdot p'_2$
- $p_3 = \gamma \cdot p'_3$

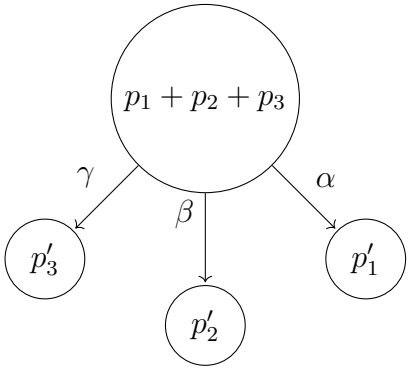
$$p_1 + p_2 + p_3 = \alpha \cdot p'_1 + \beta \cdot p'_2 + \gamma \cdot p'_3$$

avendo:

$$\frac{p_1 \xrightarrow{\alpha} p'_1}{p_1 + p_2 + p_3 \xrightarrow{\alpha} p'_1}$$

$$\frac{p_2 \xrightarrow{\alpha} p'_2}{p_1 + p_2 + p_3 \xrightarrow{\beta} p'_2}$$

$$\frac{p_3 \xrightarrow{\gamma} p'_3}{p_1 + p_2 + p_3 \xrightarrow{\gamma} p'_3}$$



- **composizione parallela**: $p_1|p_2$, avendo:

$$\frac{p_1 \xrightarrow{\alpha} p'_1}{p_1|p_2 \xrightarrow{\alpha} p'_1|p_2}$$

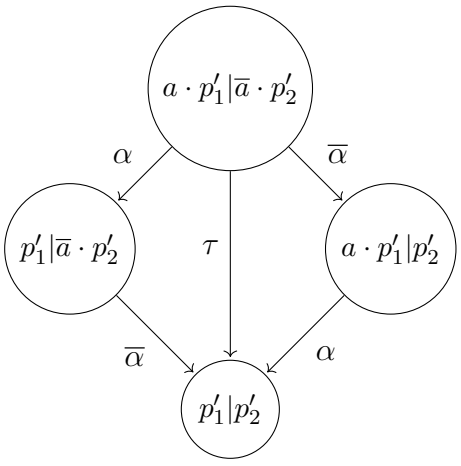
oppure

$$\frac{p_2 \xrightarrow{\alpha} p'_2}{p_1|p_2 \xrightarrow{\alpha} p_1|p'_2}$$

oppure

$$\frac{p_1 \xrightarrow{\alpha} p'_1 \wedge p_2 \xrightarrow{\overline{\alpha}} p'_2}{p_1|p_2 \xrightarrow{\tau} p'_1|p'_2}$$

e quindi, in quest'ultimo caso, non potremo più avere altre sincronizzazioni.
Quindi avendo avendo $p_1 = a \cdot p'_1$ e $p_2 = a \cdot p'_2$ ho che $p_1|p_2$ corrisponde a $a \cdot p'_1|\overline{a} \cdot p'_2$, e, l'LTS corrisponde a:



- **restrizione**, sia $L \subseteq A$. Si ha che:

  $$P_{\backslash L}$$

  significa che il processo P non può interagire con il suo ambiente con azioni in $L \cup \overline{L}$ ma le azioni in $L \cup \overline{L}$ sono locali a P. Avendo quindi:

  $$\frac{p \xrightarrow{\alpha} p'}{P_{\backslash L} \xrightarrow{\alpha} p'_{\backslash L}},\ \alpha, \overline{\alpha} \notin L,\ L \subseteq A$$

- **rietichettatura**, ovvero il cambiamento di nome ad una componente, ad una certa azione, per poter riusare tale nome. Ho quindi una funzione f tale che:

  $$F : Act \to Act$$

  inoltre devo avere sempre garantito che:

  - $f(\tau) = \tau$, ovvero le τ non cambiano
  - $f(\overline{a}) = \overline{f(a)}$ quindi un'azione soprassegnata può cambiare nome solo in un'altra soprassegnata

  Ho quindi $p_{[f]}$ tale che:

  $$\frac{p \xrightarrow{\alpha} p'}{p_{[f]} \xrightarrow{f(\alpha)} p'_{[f]}}$$

  Ho quindi che:

  $$\frac{p \xrightarrow{\alpha} p'}{k \xrightarrow{\alpha} p'} \iff k = p$$

Quindi dato un CCS posso associare un LTS per quanto riguarda la semantica.
Si ha una **precedenza degli operatori**, da quello con meno precedenza a quello con più precedenza:

1. restrizione
2. rietichettatura
3. prefisso
4. composizione parallela

5. somma

Esempio 22. *Avendo:*

$$R + a \cdot p \mid b \cdot Q_{\setminus L}$$

sarebbe:

$$R + ((a \cdot p) \mid b \cdot (Q_{\setminus L}))$$

Esempio 23. *Esercizio bonus:*

$$((a \cdot p_1' + \overline{b} \cdot p_1'')|(\overline{a} \cdot p_2' + b \cdot p_2''))_{\{a\}}$$

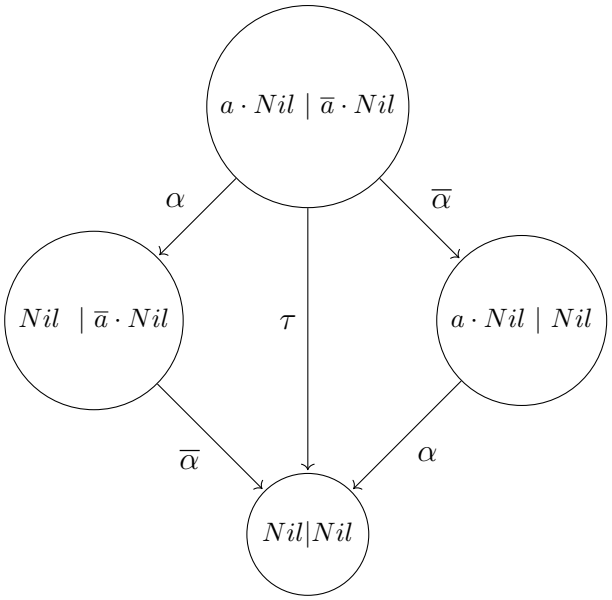
costruisco l'LTS associato ad a.

Analizziamo meglio la **composizione parallela**.

Esempio 24. *Si ha:*

$$a \cdot Nil \mid \overline{a} \cdot Nil$$

ovvero:



Posso quindi eseguire le due operazioni o in una sequenza o nell'altra.
Ho quindi una ***simulazione sequenziale non deterministica*** *del comportamento del sistema dato dalla composizione parallela.*

Quanto visto in questo esempio è possibile perché nell'ipotesi di Milner si ha che a e $\overline{a}$ sono **operazioni atomiche**.
Si hanno anche modelli di CCS modellati con:

- reti di Petri
- strutture ad eventi

ovvero due modelli in cui si considera la semantica basata sulla **true concurrency**, a *ordini parziali*, ovvero non si impongono sequenze quindi due processi o si sincronizzano o vengono eseguiti in modo concorrente.

Esempio 25. *Costruisco il sistema di transizioni della specifica:*

$$S = \overline{lez} \cdot S$$

avendo:

$$Uni = (M|LP)_{\{\backslash coin, caffe\}}$$

Ho,seguendo nei vari step le regole di inferenza legate alla sincronizzazione:

$$(coin \cdot \overline{caffe} \cdot M \mid \overline{lez} \cdot \overline{coin} \cdot caffe \cdot LP)_{\{\backslash coin, caffe\}}$$

eseguo $\overline{lez}$ *passo a:*

$$(coin \cdot \overline{caffe} \cdot M \mid \overline{coin} \cdot caffe \cdot LP)_{\{\backslash coin, caffe\}}$$

sincronizzo con coin e quindi passo con τ*:*

$$(\overline{caffe} \cdot M \mid caffe \cdot LP)_{\{\backslash coin, caffe\}}$$

sincronizzo con $caffe$ *e quindi passo con* τ*:*

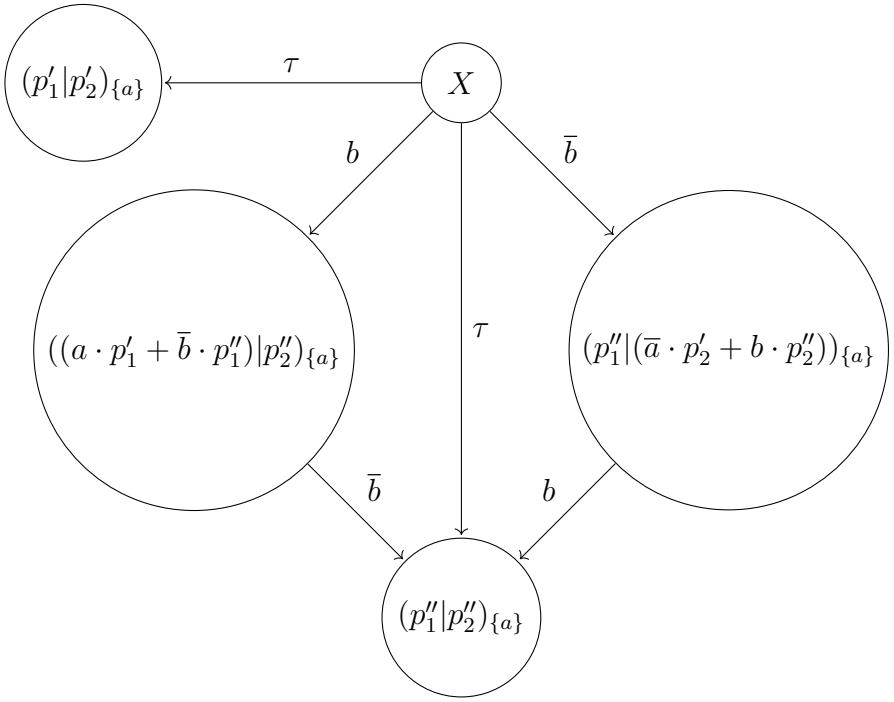
$$(coin \cdot \overline{caffe} \cdot M \mid \overline{lez} \cdot \overline{coin} \cdot caffe \cdot LP)_{\{\backslash coin, caffe\}}$$

tornando all'inizio.
(in realtà avrei i nodi per ogni formula e gli archi etichettati prima con $\overline{lez}$ *e poi con* τ *per gli ultimi due)*
*(**I due** τ **non possono avvenire ottemperantemente**).*

Esercizio 1. *Costruire LTS relativo a:*

$$X = ((a \cdot p_1' + \overline{b} \cdot p_1'')|(\overline{a} \cdot P_2' + b \cdot p_2''))_{\{a\}}$$

(nessuno può quindi interagire con l'ambiente tramite a *e* $\overline{a}$*)*



Posso infine chiedermi, vedendo l'esempio sopra, se Uni implementa S, vista come specifica. Quindi mi chiedo se posso quindi sostituire S con Uni. Innanzitutto per poter dire che una certa implementazione soddisfa (indicata con $implementazione \vDash specifica$) una certa specifica o se due implementazioni diverse soddisfano la stessa specifica ci serve una **relazione di equivalenza** tra processi CCS, ovvero una R:

$$R \subseteq P_{CCS} \times P_{CCS}$$

che sia:

- riflessiva
- simmetrica
- transitiva

Bisognerà inoltre astrarre:

- gli stati e considerare le azioni Act
- dalle sincronizzazioni interne, ovvero dalle τ
- rispetto al non determinismo

Milner poi asserisce che R deve essere inoltre una **congruenza** rispetto agli operatori del CCS.

Definizione 22. *Una relazione di equivalenza R è una **congruenza** sse:*

$$\forall\ p, q \in Proc_{CCS} \wedge \forall c[\cdot]\ \textit{contesto CCS}$$

(avendo quindi un contesto CCS sostituibile con qualcosa)
allora:

$$\textit{se } pRq \textit{ allora si ha } c[p]\ R\ c[q]$$

Esempio 26. *Preso $Uni = (M \mid LP)_{\{caffe,coin\}}$ posso dire che prendo il contesto:*

$$(\cdot \mid LP)_{\{caffe,coin\}}$$

se $M_1\, R\, M_2$ allora deve succedere che:

$$(M_1 \mid LP)_{\{caffe,coin\}}\ R\ (M_2 \mid LP)_{\{caffe,coin\}}$$

quindi posso sostituire l'uno con l'altro senza avere conseguenze, essendo equivalenti.

Teorema 4. *Dati due processi p_1 e p_2 a cui assegniamo i due LTS v_1 e v_2. I due processi sono equivalenti se v_1 e v_2 sono isomorfi. Possiamo in realtà cercare qualcosa di "meno forte" rispetto all'isomorfismo ma che garantisca lo stesso risultato.*
*Si va a vedere quindi se i due LTS **ammettono le stesse sequenze di operazioni**, prendendo l'**equivalenza forte** tra automi a stati finiti. Questa è detta **equivalenza rispetto alle tracce** (che vale anche per due programmi, che sono equivalenti se implementano la stessa sequenza di istruzioni, lo stesso algoritmo). È comunque più forte di dire che hanno la stessa precondizione e postcondizione.*
***Bisognerà discutere la cosa rispetto alla congruenza**.*
Preso $p \in Proc_{CCS}$ ho l'insieme delle traccie di p:

$$tracce(p) = \{w \in Act^* \mid \exists\, p' \in Proc_{CCS}\ p \xrightarrow{w} p'\}$$

Suppongo, per ora, di non considerare le sincronizzazioni (quindi senza τ).

Teorema 5. *Preso $p' \in Proc_{CCS}$ ho che è equivalente rispetto alle tracce a p'', scritto:*

$$p' \overset{T}{\sim} p''$$

sse:

$$tracce(p') = tracce(p'')$$

Esempio 27. *Vediamo quindi un esempio.*
Prendo il sistema:

$$(LP|M_i)\_\{coin, caffe\}$$

$$LP = \overline{lez}cdotcoin \cdot \overline{coffe} \cdot LP$$

suppongo di avere due macchinette M che erogano sia caffè che tea, per il primo basta una moneta e per il secondo due.
La prima macchina è::

$$M_1 = \overline{coin} \cdot (caffe \cdot M_1 + \overline{coin} \cdot tea \cdot M_1)$$

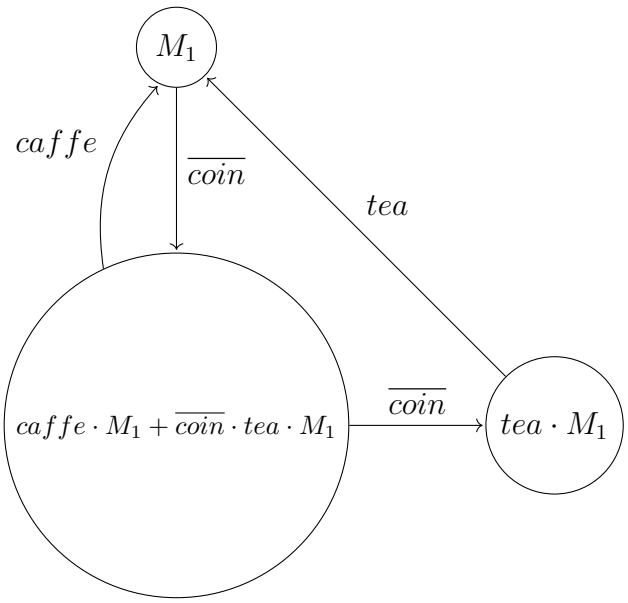
la seconda:

$$M_2 = (\overline{coin} \cdot caffe \cdot M_2) + (\overline{coin} \cdot coin \cdot tea \cdot M_2)$$

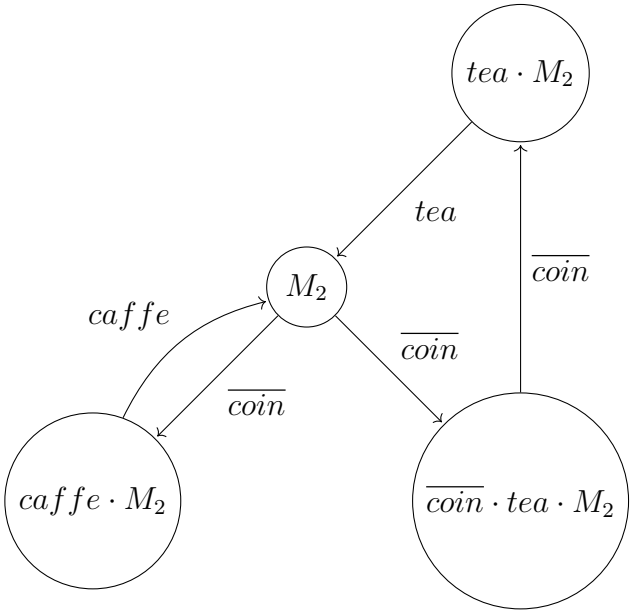
Mi chiedo se:

$$M_1 \overset{T}{\sim} M_2$$

Parto da M_1:



Passo ad M_2:



Si vede quindi che le tracce di M_1 sono le stesse di quelle di M_2. Presi:

$$(LP|M_1)\_\{coin, caffe\}$$

$$(LP|M_2)\_\{coin, caffe\}$$

Si nota che M_2 può andare in deadlock, avendo due processi che iniziano con $\overline{coin}$ (non posso sapere dove va e se vado sul nodo del tea la macchina si aspetta un'altra moneta, mentre magari si voleva il caffè). Quindi anche se le tracce sono le stesse ma non posso sostituire senza problemi le due macchinette, in quanto se sostituisco M_1 con M_2 rischio di andare in deadlock.

Lo studio delle tracce quindi non è più sufficiente nel caso di sistemi concorrenti. Si necessità quindi di una nozione più restrittiva.

5.2.1 Bisimulazione forte

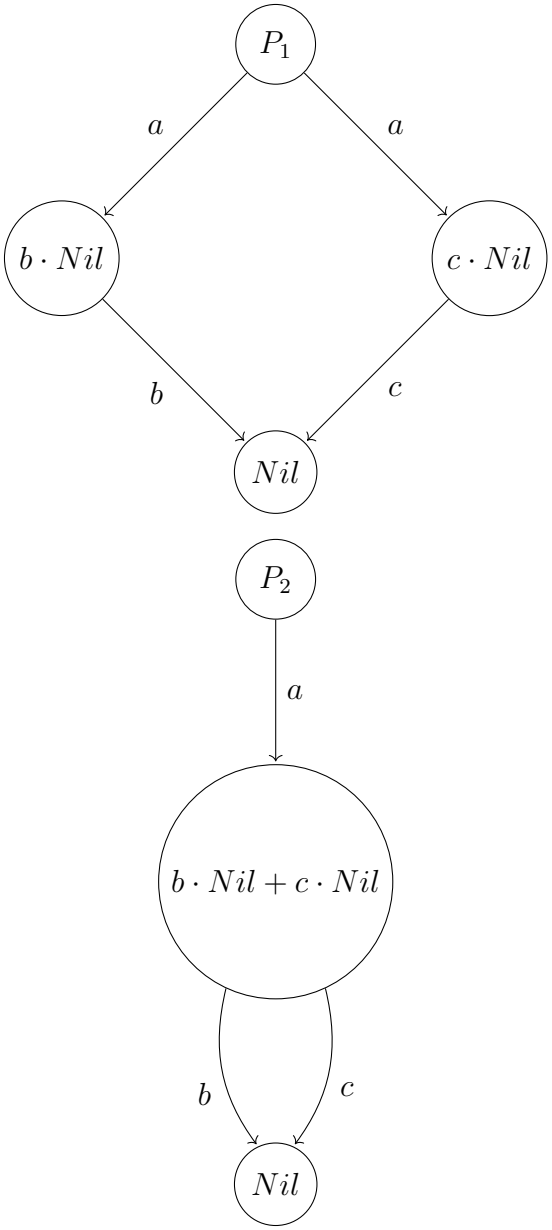
Per ovviare al problema sopra descritto si ha la **bisimulazione**. M_1 e M_2 non sono equivalenti rispetto alla **bisimulazione** e quindi non posso sostituire l'una con l'altra:

$$M_1 \overset{Bis}{\not\sim} M_2$$

Esempio 28. *Siano:*

$$P_1 = a \cdot b \cdot Nil + a \cdot c \cdot Nil$$

$$P_2 = a \cdot (b \cdot Nil + c \cdot Nil)$$



Avendo quindi:

$$T(P_1) = \{\varepsilon, a, ab, ac\}$$
$$T(P_2) = \{\varepsilon, a, ab, ac\}$$

quindi:

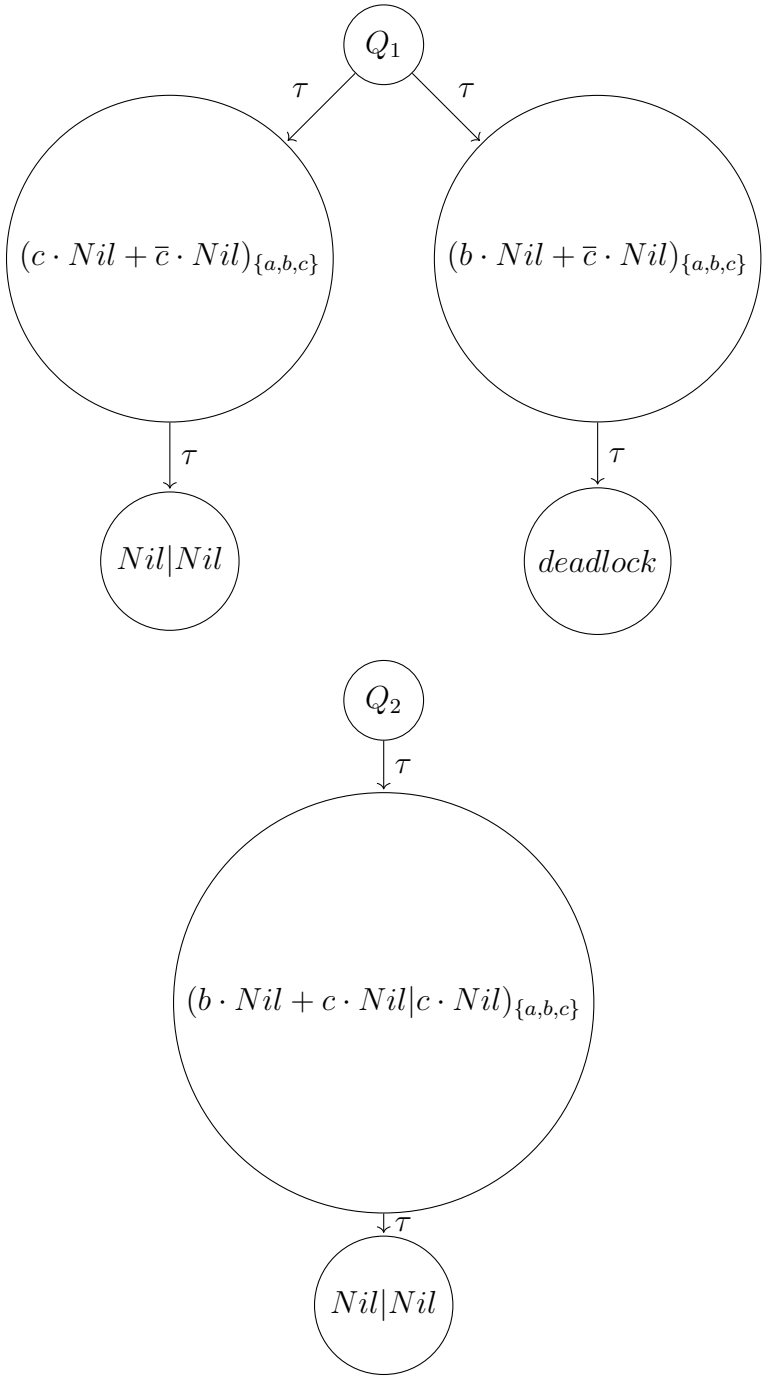
$$P_1 \sim^T P_2$$

Ma se ad esempio prendo:

$$Q_1 = (P_1 | \overline{a} \cdot \overline{b} \cdot Nil)_{\{a,b,c\}}$$

$$Q_2 = (P_2|\overline{a} \cdot \overline{b} \cdot Nil)_{\{a,b,c\}}$$

Ho:



Quindi il primo va in deadlock e il secondo no, quindi vedremo non sono equivalenti per la bisimulazione.

Definizione 23. *data una relazione binaria $R \subseteq Proc_{CCS} \times Proc_{CCS}$ è una relazione di* ***bisimulazione (forte)*** *sse:*

$$\forall\, p, q \in Proc_{CCS} : \;\; pRq \text{ vale che}$$

- $\forall \alpha \in Act = A \cup \overline{A} \cdot \tau$ *se ho* $p \xrightarrow{\alpha} p'$ *allora deve esistere un*

$$\exists q' \;\; t.c \; q \xrightarrow{\alpha} q' \;\; e \text{ si ha } p'Rq'$$

- $\forall \alpha \in Act = A \cup \overline{A} \cdot \tau$ *se ho* $q \xrightarrow{\alpha} q'$ *allora deve esistere un*

$$\exists p' \;\; t.c \; p \xrightarrow{\alpha} p' \;\; e \text{ si ha } p'Rq'$$

Due processi p e q sono fortemente bisimili:

$$p \sim^{Bis} q$$

sse $\exists\; R \subseteq Proc_{CCS} \times Proc_{CCS}$, relazione di bisimulazione forte tale che pRq. SI ha che:

$\sim^{Bis} = \cup\{R \subseteq Proc_{CCS} \times Proc_{CCS} |\; R$ *è una relazione di bisimulazione forte*$\}$

Teorema 6. *Se prendo $\sim^{Bis} \subseteq Proc_{CCS} \times Proc_{CCS}$ si dimostra che è:*

- *riflessiva*
- *simmetrica*
- *transitiva*

e quindi è una ***relazione di equivalenza****.*
Quindi:

$$p \sim^{Bis} q \iff \forall \alpha \in Act \;\; se \; p \xrightarrow{\alpha} p'$$

$$allora$$

$$\exists q' : q \xrightarrow{\alpha} q' \wedge p' \sim^{Bis} q'$$

$$e \;\; se$$

$$q \xrightarrow{\alpha} q'$$

$$allora$$

$$\exists p' : p \xrightarrow{\alpha} p' \wedge p' \sim^{Bis} q'$$

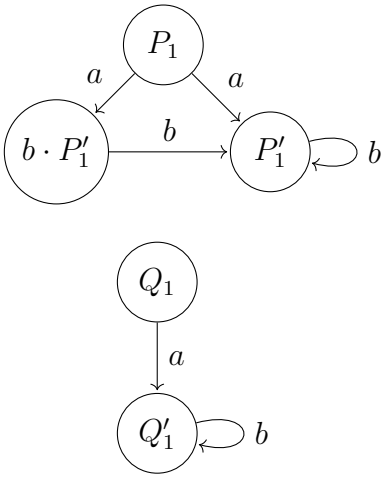
Teorema 7. *Se due processi sono fortemente bisimili allora sono sicuramente equivalenti rispetto alle tracce.*
NON VALE IL VICEVERSA *(come visto negli esempi sopra).*

Per vedere che due processi sono bisimili devo quindi, per ogni esecuzione, ottenere due processi ancora bisimili (potendo quindi fare le azioni corrispondenti da entrambe le parti).

Esempio 29. *esempio 28 ad esempio dopo a arrivo in $c \cdot Nil$ da una parte e $b \cdot Nil + c \cdot Nil$ ma banalmente dal secondo posso eseguire sia b che c, cosa che non posso fare nel primo, che può eseguire solo c.*

Esempio 30. *Siano:*

$$P_1 = a \cdot b \cdot P_1' + a \cdot P_1'$$
$$P_1' = b \cdot P_1'$$
$$Q_1 = a \cdot Q_1'$$
$$Q_1' = b \cdot Q_1'$$



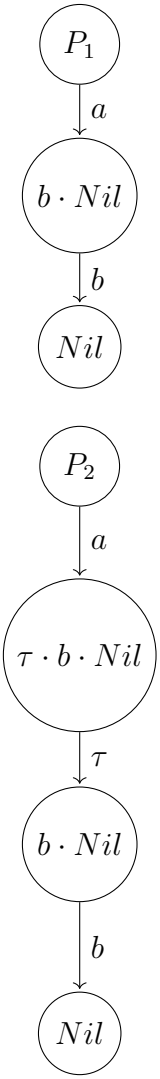
Quindi ho:

$$P_1 \sim^{Bis} Q_1$$

Vedendo che i passi che posso fare nell'uno li posso fare anche nell'altro.

Esempio 31. *Siano:*

$$P_1 = a \cdot b \cdot Nil$$
$$P_2 = a \cdot \tau \cdot b \cdot Nil$$



Ho che:

$$P_1 \not\sim^T P_2$$

e che:

$$P_1 \not\sim^{Bis} P_2$$

Ma si vede che i comportamenti sono simili.

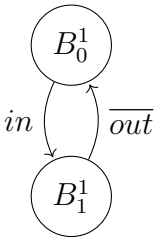
Cerco quindi di astrarre dalle interazioni interne (τ), introducendo l'**equivalenza debole**, volendo astrarre rispetto alle azioni τ, introducendo la **bisimulazione debole** (e nell'esempio precedente i due processi sono debolmente equivalenti sia per le tracce che per la bisimulazione).

Esempio 32. *Vediamo l'esempio dei buffer, dato un buffer di capacità r che contiene i elementi ho* B_i^R.
Sia quindi, su $A = \{in, out\}$, $\overline{A} = \{\overline{in}, \overline{out}\}$, *con A per azioni di ricezione e* $\overline{A}$ *per azioni di invio, il buffer a capacità 1:*

$$B_0^1 = in \cdot B_1^1$$

$$B_1^1 = \overline{out} \cdot B_0^1$$

Avendo il corrispondente LTS:



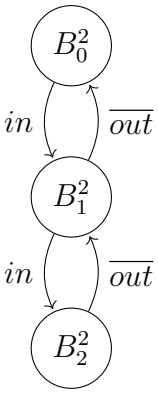
Passo al buffer a capacità 2:

$$B_0^2 = in \cdot B_1^2$$

$$B_1^2 = \overline{out} \cdot B_0^2 + in \cdot B_2^2$$

$$B_2^2 = \overline{out} \cdot B_1^2$$

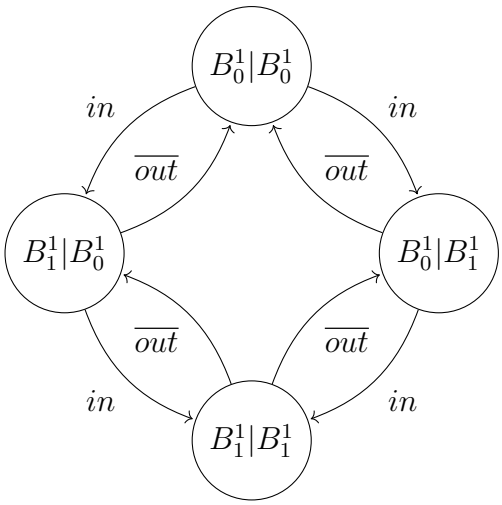
Avendo il corrispondente LTS:



Provo ora a sincronizzare due buffer a capacità 1:

$$B_0^1|B_0^1$$

Avendo il corrispondente LTS, non avendo la sincronizzazione:



Si ha quindi, in quest'ultimo modello, una ***simulazione sequenziale non deterministica della concorrenza****.*
Vediamo se si ha che:

$$(B_0^1|B_0^1) \sim^{Bis} B_0^2$$

B_0^2 *può essere messo in relazione con* $B_0^1|B_0^1$ *in quanto se vado in uno tra* $B_1^1|B_0^1$ *e* $B_0^1|B_1^1$ *posso sempre fare in e* $\overline{out}$*. Inoltre Posso fare un discorso analogo per* B_2^2 *e* $B_1^1|B_1^1$*. Essendo questi ultimi quindi bisimili lo sono anche il nodo centrale coi due possibili nodi nel caso della composizione e di conseguenza lo sono anche* B_0^2 *e* $B_0^1|B_0^1$*.*

Vediamo quindi le proprietà della bisimulazione forte.
In primis la bisimulazione forte è una **congruenza** rispetto agli operatori del CCS. Quindi se ho $p \sim^{Bis} q,\ \ p, q \in Proc_{CCS}$ ho che:

$$\alpha \cdot p \sim^{Bis} \alpha \cdot q,\ \ \forall\, \alpha \in Act$$

Ho inoltre che:

$$p + r \sim^{Bis} q + r \ \ \wedge \ \ r + p \sim^{Bis} r + q,\ \ \forall\, r \in Proc_{CCS}$$

Ho anche che;

$$p|r \sim^{Bis} q|r \ \ \wedge \ \ r|p \sim^{Bis} r|q,\ \ \forall\, \alpha \in Act$$

Ho poi che:

$$p_{[f]} \sim^{Bis} q_{[f]}, \ \forall f : Act \to Act \text{ funzione di rietichettatura}$$

Con f, ricordiamo, tale che:

- $f(\tau) = \tau$
- $f(\overline{a}) = \overline{f(a)}$

Ho infine che:

$$p_{\backslash L} \sim^{Bis} q_{\backslash L}, \ \forall L \in A$$

A queste regole possiamo aggiungere delle "leggi" legate a degli assiomi, legate al fatto che gli operatori hanno delle proprietà.
Ho per esempio, $\forall p, q, r \in Proc_{CCS}$:
Commutatività:

$$p + q \sim^{Bis} q + p$$
$$p|q \sim^{Bis} q|p$$

Distributività:

$$(p + q) + r \sim^{Bis} p + (q + r)$$
$$(p|q)|r \sim^{Bis} p|(q|r)$$

Leggi di assorbimento:

$$p + Nil \sim^{Bis} p$$
$$p|Nil \sim^{Bis} p$$

5.2.2 Bisimulazione debole

Abbiamo visto nell'esempio 31 come la bisimulazione forte rischi di essere troppo restrittiva.
Si passa quindi alla **equivalenza debole rispetto alle tracce**:

$$\approx^{T}$$

e alla **bisimulazione debole**:

$$\approx^{Bis}$$

Bisogna però cambiare funzione di transizione. Si passa da:

$$p \xrightarrow{a} p'$$

a:

$$p \stackrel{a}{\Rightarrow} p'$$

Dove p può fare tutte le τ che vuole prima e dopo fare a, per poi comportarsi come p', avendo quindi la **relazione di transizione debole**.

Definizione 24. *Definiamo* ***relazione/regola di transizione debole****, come:*

$$\Rightarrow \subseteq Proc_{CCS} \times Act \times Proc_{CCS}$$

avendo:

$$p \overset{a}{\Rightarrow} p', \;\; a \in A \cup \overline{A} \cup \tau$$

sse:

- *se $a = \tau$ posso eseguire una sequenza qualsiasi anche nulla di τ:*

  $$p \overset{\tau}{\rightarrow}^* p'$$

  fino ad arrivare a p' (se 0 ho che $p = p'$)

- *se $a \in A \cup \overline{A}$ ho che:*

  $$p \overset{\tau}{\rightarrow}^* \overset{a}{\rightarrow} \overset{\tau}{\rightarrow}^* p'$$

Estendo la relazione a sequenze di azioni, $w \in act^*$:

$$p \overset{w}{\Rightarrow} p'$$

sse:

- *se $w = \varepsilon$, sequenza vuota, o $w = \tau^*$, sequenza di τ, ho:*

  $$p \overset{\tau}{\rightarrow}^* p'$$

- *se $w = a_1 \dots a_n, \;\; a_i \in A \cup \overline{A}$ ho:*

  $$p \overset{a_1}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} p'$$

  dove ogni a_i può essere preceduto/seguito da una qualsiasi sequenza di τ.

  Esempio 33.

  $$p \overset{ab}{\Rightarrow} p'$$

  potrebbe essere:

  $$p \overset{\tau}{\rightarrow}^* p_2 \overset{a}{\rightarrow} \overset{\tau}{\rightarrow}^* p_3 \overset{b}{\rightarrow} \overset{\tau}{\rightarrow}^* p'$$

  oppure:

  $$p \overset{\varepsilon}{\Rightarrow} p'$$

  comporta:

  $$p = p' \vee p \overset{\tau}{\rightarrow}^* p' \equiv p \overset{\tau}{\rightarrow}^* p'$$

Definizione 25. *Definiamo l'**equivalenza debole rispetto alle tracce debole**:*

$$p \approx^{T} q$$

sse:

$$Tracce_{\Rightarrow}(p) = Tracce_{\Rightarrow}(q)$$

ovvero, equivalentemente:

$$\forall w \in (a \cup \overline{A})^* \text{ ho che } p \overset{w}{\Rightarrow} \iff q \overset{w}{\Rightarrow}$$

(quindi se i due processi possono fare la stessa sequenza).
Ho quindi che:

$$Tracce_{\Rightarrow}(p) = \{w \in (a \cup \overline{A})^* | p \overset{w}{\Rightarrow}\}$$

Quindi o ho le stesse tracce con la regola debole ho ogni sequenza del primo processo la devo ritrovare nel secondo processo.
Banalmente prendo le tracce forti e cancello τ

Esempio 34. *Riprendo esempio 31.*
Ho:

$$Tracce_{\Rightarrow}(P_1) = \{\varepsilon, a, ab\}$$

$$Tracce_{\Rightarrow}(P_2) = \{\varepsilon, a, ab\}$$

e quindi ho che:

$$P_1 \approx^{T} P_2$$

Esempio 35. *Riprendendo l'esempio 28 ho che:*

$$Q_1 \approx^{T} Q_2$$

ma vedremo che comunque:

$$Q_1 \not\approx^{Bis} Q_2$$

Vediamo una definizione comoda di bisimulazione debole:

Definizione 26. *Dato $R \subseteq Proc_{CCS} \times Proc_{CCS}$ è una bisimulazione debole sse:*

$$\forall p, q \in Proc_{CCS} \text{ t.c. } pRq \text{ vale che } \forall a \in Act$$

$$\text{se } p \xrightarrow{a} p_1 \text{ allora } \exists q \overset{a}{\Rightarrow} q_1 \text{ tale che } p_1 R q_1$$

e deve valere il viceversa:

$$\text{se } q \xrightarrow{a} q_1 \text{ allora } \exists p \overset{a}{\Rightarrow} p_1 \text{ tale che } p_1 R q_1$$

Quindi due processi p q sono in bisimulazione debole:

$$p \approx^{bis} q$$

sse esiste una relazione di bisimulazione R tale che:

$$pRq$$

e si ha:

$$\approx^{Bis} = \cup\{R : \ R \text{ è di bisimulazione debole}\}$$

Regole del gioco

Vediamo le **regole del gioco** che spiegano come capire se due processi sono bisimili.
Per confrontare due processi CCS p e q uso un gioco $G(p, q)$ con due giocatori:

- l'**attaccante** che cerca di dimostrare $p \not\approx^{Bis} q$
- il **difensore** che cerca di dimostrare $p \approx^{Bis} q$

Un gioco è costituito da più **partite**. Una partita è una sequenza finita o infinita di configurazioni (p_n, q_n), con la configurazione iniziale:

$$(p, q) = (p_0, q_0)$$

una configurazione è detta **mano** e da ogni configurazione/mano corrente si passa a quella successiva tramite diverse regole:

- attaccante sceglie uno dei processi della configurazione corrente (p_i, q_i) e fa una $\xrightarrow{a}$ con la regola forte, passando alla successiva
- il difensore deve rispondere con una $\stackrel{a}{\Rightarrow}$, ovvero con la regola debole, sull'altro processo, passando alla successiva
- la nuova coppia p_{i+1}, q_{i+1} è la nuova configurazione
- la partita continua con l'altra mano

Se un giocatore non può più muovere l'altro vince.
Si ha però che **se la partita è infinita vince il difensore**.
SI ha inoltre che diverse partire possono avere vincitori diversi ma per ogni gioco un solo giocatore può vincere la partita. Si ha quindi il concetto di **strategia** che indica di volta in volta le regole che indicano la mossa per un certo giocatore. Tali regole dipendono solo dalla configurazione corrente.
Un giocatore ha una **strategia vincete** per $G(p, q)$ se, seguendo quella strategia, vince tutte le partite del gioco.

Teorema 8. *Per ogni gioco solo uno dei due giocatori ha una strategia vincente*

Teorema 9. *Si ha che:*

- *l'attaccante ha una strategia vincente sse* $p \not\approx^{Bis} q$
- *il difensore ha una strategia vincente sse* $p \approx^{Bis} q$

Uso quindi il gioco per dimostrare se due processi sono bisimili o meno. Si ha che:

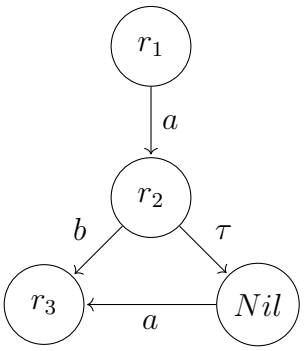
- per dimostrare che sono bisimili bisogna mostrare che per ogni mossa dell'attaccante il difensore ha sempre almeno una mossa che lo porterà a vincere.
- per vedere che non sono bisimili l'attaccante, in ogni configurazione, è sempre in grado di scegliere su quale processo operare e come, in modo che per ogni risposta del difensore l'attaccante avrà sempre una mossa che lo porterà a vincere.

Esempio 36. *Vediamo l'esempio di un gioco con due processi che dimostreremo essere bisimili:*

Siano:

$$r_1 = a \cdot (b \cdot Nil + \tau \cdot c \cdot Nil)$$

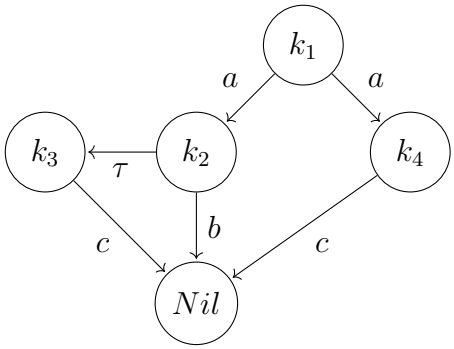
ovvero:



e:

$$k_1 = a \cdot (b \cdot Nil + \tau \cdot c \cdot Nil) + a \cdot c \cdot Nil$$

ovvero:



Vediamo quindi come si svolgono tutte le partite per $G(r_1, k_1)$*:*

- *all'inizio l'attaccante può fare* $r_1 \xrightarrow{a} r_2$ *e il difensore può fare* $k_1 \stackrel{a}{\Rightarrow} k_2$*, ed è la scelta vincente, avendo che* r_2 *e* k_2 *sono isomorfi. Una scelta buona su tre possibilità*
- *all'inizio l'attaccate può fare* $k_1 \xrightarrow{a} k_2$ *e il difensore può fare* $r_1 \stackrel{a}{\Rightarrow} r_2$*, ed è la scelta vincente, avendo che* r_2 *e* k_2 *sono isomorfi. Una scelta buona su due possibilità*
- *all'inizio l'attaccate può fare* $k_1 \xrightarrow{a} k_4$ *e il difensore può fare* $r_1 \stackrel{a}{\Rightarrow} r_3$*, ed è la scelta vincente, avendo che* r_3 *e* k_4 *sono isomorfi.Una scelta buona su tre possibilità*

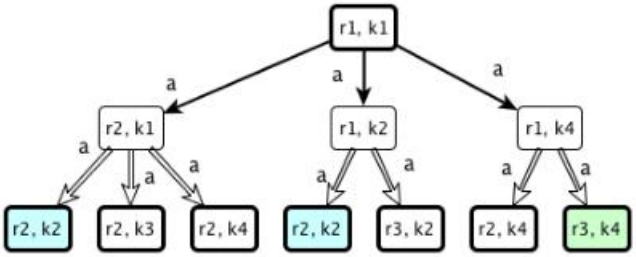
Il difensore ha quindi una strategia vincente. Ho quindi che la relazione di bisimilitudine:

$$R = \{(r_1, k_1), (r_2, k_2), (r_3, k_3), (r_3, k_4), (Nil, Nil)\}$$

Quindi:

$$r_1 \approx^{Bis} k_1$$

Posso fare una rappresentazione ad albero, che vediamo in modo parziale, segnando in verde le mosse vincenti dei difensori (con processi bisimili):

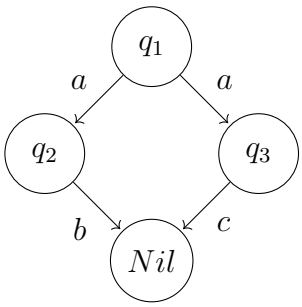


Esempio 37. *Vediamo l'esempio di un gioco con due processi che dimostreremo non essere bisimili:*

Siano:

$$q_1 = a \cdot b \cdot Nil + a \cdot c \cdot Nil$$

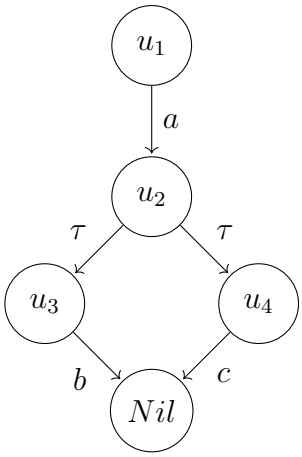
ovvero:



e:

$$u_1 = a \cdot (\tau \cdot b \cdot Nil + \tau \cdot c \cdot Nil)$$

ovvero:



Vediamo quindi come si svolgono tutte le partite per $G(q_1, u_1)$.
L'attaccante esegue $u_1 \xrightarrow{a} u_2$.
Il difensore ha due scelte:

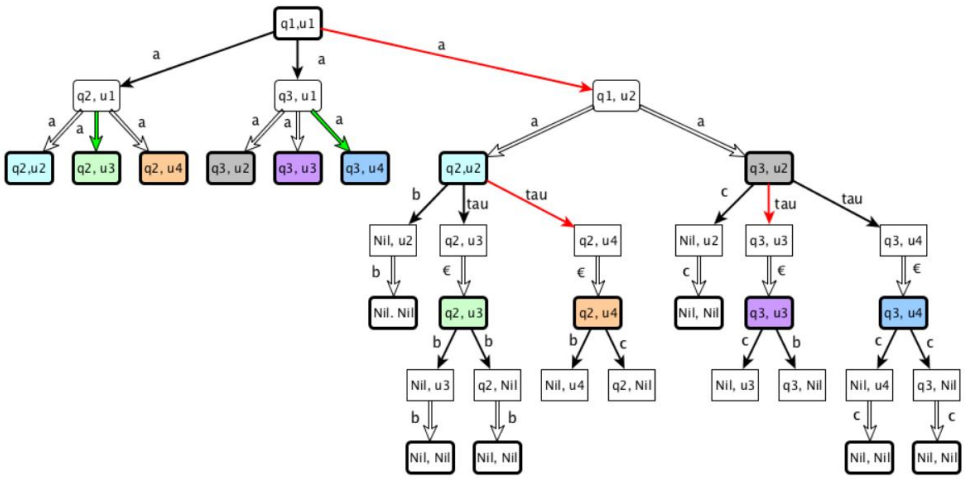
- $q_1 \xrightarrow{a} q_2$. *Sono quindi in* (q_2, u_2). *A questo punto* $u_2 \xrightarrow{\tau} u_4$ *a cui il difensore non può che rispondere con l'azione nulla,* $q_2 \xrightarrow{\tau} q_2$, *perdendo in quanto* $q_2 \not\approx^{Bis} u_4$ *dato che dal primo,* q_2 *posso fare solo b e dal secondo,* u_4 *solo c*

- $q_1 \xrightarrow{a} q_3$. *Sono quindi in* (q_3, u_2). *A questo punto* $u_2 \xrightarrow{\tau} u_3$ *a cui il difensore non può che rispondere con l'azione nulla,* $q_3 \xrightarrow{\tau} q_3$, *perdendo in quanto* $q_3 \not\approx^{Bis} u_3$ *dato che dal primo,* q_3, *posso fare solo c e dal secondo,* u_3, *solo b*

Quindi:

$$q_1 \not\approx^{Bis} u_1$$

Vediamo anche in questo caso un albero parziale, dove si vedono anche le mosse in cui l'attaccante perdeva. Sono di egual colore le configurazioni identiche e si hanno col bordo bold le foglie (non colorate) corrispondi alle vincite del difensore, che però non ha sempre una mossa per risponde attaccante, comportando la non bisimulazione:



Si hanno infatti 4 foglie non colorate e non con il contorno bold che segnalano la vittoria dell'attaccante.
Le foglie colorate rappresentano parti di alberi parziali.

- se sono bisimili il difensore deve poter trovare una mossa vincente per ogni mossa dell'attaccante e quindi per ogni possibile mossa intermedia, avendo quindi una strategia vincente

- se non sono bisimili si ha almeno una mossa dell'attaccante, in ogni configurazione, per cui, comunque scelga poi il difensore, l'attaccante avrà ancora una mossa vincente, avendo quindi una strategia vincente

Se non si vuole dimostrare la bisimilitudine ma la si vuole cercare da zero bisogna studiare tutte le possibili partite o, meglio, un insieme di partite che mi possa dimostrare quale dei due giocatori ha la ***strategia vincente****.*
Facciamo un piccolo "recap" della situazione.

Definizione 27. *Definiamo una* ***proprietà di equivalenza*** *tra processi CCS:*

$$\backsimeq \subseteq Proc_{CCS} \times Proc_{CCS}$$

Si ha che, dati $p, q \in Proc_{CCS}$*:*

$$se\ LTS(p) = LTS(q)\ allora\ p \backsimeq q$$

(ovvero se i due LTS sono isomorfi)
Vale per altri modelli di concorrenza oltre che a il CCS.
Si stanno considerando solo le azioni e non gli stati (si hanno altri approcci che considerano gli stati).

Teorema 10. *Si ha che, dati* $p, q \in Proc_{CCS}$*:*

$$p \backsimeq q \implies Tracce(p) = Tracce(q)$$

Qui le stesse sequenze di azioni possono essere eseguite su entrambi i processi.

Teorema 11. *Si ha che, dati* $\forall p, q \in Proc_{CCS}$*, qualora si abbia che* $p \backsimeq q$*, i due processi devono avere la stessa possibilità di generare deadlock nell'interazione con l'ambiente (ovvero l'insieme dei processi con cui p e q possono interagire). Quindi sostituendo p con q, qualora il primo non avesse deadlock, posso dire che anche il secondo non avrà deadlock se questi sono equivalenti.*

Teorema 12. *Si ha che* $\backsimeq$ *deve essere una* ***congruenza*** *rispetto agli operatori del CCS. Deve essere possibile sostituire un sottoprocesso con un suo equivalente senza modificare il comportamento complessivo del sistema. Sostituendo quindi un sottoprocesso con un suo equivalente il processo di partenza deve essere equivalente al nuovo processo ottenuto dopo la sostituzione, in qualsiasi contesto ci si trovi.*

Definizione 28. *Definiamo l'**equivalenza rispetto alle tracce forte** $\sim^T$. Dati $\forall p, q \in Proc_{CCS}$:*

$$LTS(p) = LTS(q) \implies p \sim^T q$$

Anche qui si estrae dagli stati, osservando solo le tracce, e si ha che:

$$p \sim^T q \iff Tracce(p) = Tracce(q)$$

Teorema 13. *Si dimostra che l'equivalenza rispetto alle tracce forte è una **congruenza** rispetto agli operatori CCS.*

Teorema 14. *La nozione di equivalenza rispetto alle tracce forte non garantisce di preservare il deadlock o l'assenza di deadlock nell'interazione con l'ambiente (come visto nell'esempio 27, dove l'equivalenza rispetto alle tracce forte non è sufficiente a garantire nulla rispetto al deadlock).*

Partendo da quanto sopra si introduce quindi concetto di **bisimulazione forte** (equivalenza forte rispetto alla bisimulazione), introdotto da Milner. Si ricorda che, $\forall p, q \in Proc_{CCS}$, astraendo dagli stati:

- $LTS(p) = LTS(q) \implies p \sim^{Bis} q$
- $p \sim^{Bis} q \implies Tracce(p) = Tracce(q)$. In altri termini si ha che $p \sim^{Bis} q \implies p \sim^T q$ e quindi:

  $$\sim^{Bis} \subseteq \sim^T$$

  non posso avere processi bisimili che non abbiano le stesse tracce
- la bisimulazione forte preserva il deadlock o l'assenza di deadlock nell'interazione con l'ambiente
- la bisimulazione forte è una congruenza rispetto agli operatori del CCS

Si ha però che la bisimulazione forte è troppo restrittiva infatti, come anche $\sim^T$, non astrae τ, avendo, per esempio, che $a \cdot b \cdot Nil \not\sim^{Bis} a \cdot \tau \cdot b \cdot Nil$. Non posso quindi confrontare alcuna specifica con sincronizzazioni interne alle componenti di un processo, rispetto alle azioni non osservabili.
Si è introdotta quindi la regola di **transizione debole**, $\stackrel{a}{\Rightarrow}$.
Si ha **l'equivalenza debole rispetto alle tracce**, $\approx^T$ è meno restrittiva rispetto a quella forte infatti:

$$p \sim^T q \implies p \approx^T q$$

e quindi:

$$\sim^T \subseteq \approx^T$$

Quindi se due processi sono equivalenti fortemente rispetto alle tracce lo saranno anche debolmente. L'equivalenza debole rispetto alle tracce ha quindi le seguenti proprietà, astraendo dagli stati:

- $LTS(p) = LTS(q) \implies p \approx^T q$
- l'equivalenza debole rispetto alle tracce è una congruenza rispetto agli operatori del CCS
- l'equivalenza debole rispetto alle tracce non garantisce di preservare il deadlock o l'assenza di deadlock nell'interazione con l'ambiente

Passando alla **bisimulazione debole** (equivalenza debole rispetto alla bisimulazione), $\approx^{Bis}$ si ha che è meno restrittiva rispetto a quella forte infatti:

$$p \sim^{Bis} q \implies p \approx^{Bis} q$$

e quindi:

$$\sim^{Bis} \subseteq \approx^{Bis}$$

Si ha quindi che la bisimulazione forte è più restrittiva ell'equivalenza rispetto alle tracce forte ed egualmente la bisimulazione debole è più restrittiva ell'equivalenza rispetto alle tracce debole:

$$p \sim^{Bis} q \implies p \sim^T q \text{ e } p \approx^{Bis} q \implies p \approx^T q$$

quindi si ha che:

$$\sim^{Bis} \subseteq \sim^T \text{ e } \approx^{Bis} \subseteq \approx^T$$

Non si hanno rapporti "misti" tra forte e debole.

Definizione 29. *Dato $p \in Proc_{CCS}$ si ha che esso è un **processo deterministico** sse vale che:*

$$\forall x \in Act, \text{se } p \xrightarrow{x} p' \text{ e } p \xrightarrow{x} p'' \text{ allora } p' = p''$$

Quindi se posso passare a p' o p'' con la stessa azione allora necessariamente questi due processi coincidono.

Esempio 38. *vediamo un esempio di p **non deterministico**:*

$$p = x \cdot p' + x \cdot p''$$

Teorema 15. *Siano* $p, q \in Proc_{CCS}$ ***deterministici*** *e sia* $p \sim^T q$ *(quindi anche* $p \approx^T q$*) allora si ha che:*

$$p \sim^{Bis} q$$

e quindi anche:

$$p \approx^{Bis} q$$

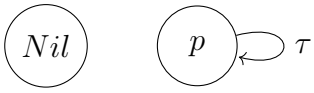
Avere processi deterministici con le stesse tracce non assicura che i due processi abbiano lo stesso LTS. Come esempio basta vedere un processo $p = a \cdot b \cdot a \cdot b \cdot p$ e uno $q = a \cdot b \cdot q$, dove il primo ha due nodi in più anche se sono deterministici con quindi equivalenza rispetto alle tracce e forte bisimilitudine, essendo quindi equivalenti, per entrambe le equivalenza, debolmente.
Se ho invece due processi non entrambi deterministici potrei avere equivalenza rispetto alle tracce (debole e forte) ma non bisimilitudine. Come esempio si prendano $p = a \cdot \tau \cdot b \cdot Nil$ e $q = a \cdot (\tau \cdot b \cdot Nil + \tau \cdot Nil)$, con quest'ultimo che non è un processo deterministico.
Basta quindi che un processo sia non deterministico per impedire il "collasso" dell'equivalenza rispetto alle tracce su quella rispetto alla bisimulazione.
Approfondiamo ora le proprietà della bisimulazione debole, tra cui:

- la bisimulazione debole, per come è definita (unione di tutte le relazioni di bisimulazione), è la più grande relazione di bisimulazione debole ed è di equivalenza, essendo:
  - riflessiva
  - simmetrica
  - transitiva
- la bisimulazione debole, come quella forte, preserva la possibilità di generare o meno deadlock nell'interazione con l'ambiente, quindi sostituendo un processo che non genera deadlock con un suo bisimile ho che anche il nuovo sistema non andrà in deadlock (se invece generava un deadlock ora andrà in deadlock). Questa cosa, si ricorda, non è vera per le l'equivalenza sulle tracce
- la bisimulazione debole, a differenza di quella forte, astrae da azioni non osservabili, le τ, e da cicli inosservabili, i τ loop. Se ho un ciclo infinito di τ si parla di **divergenza**.

Esempio 39. *Presi un processo Nil e uno $p = \tau \cdot p$ (quindi un processo infinito di sole τ, una divergenza) ho che:*

$$Nil \approx^{Bis} p$$

avendo:



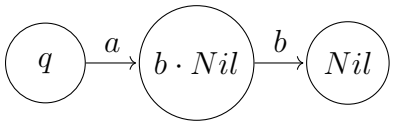
Esempio 40. *Siano $q = a \cdot b \cdot Nil$ e $s \cdot a \cdot v$ con $v = \tau \cdot v + b \cdot Nil$. Ho che:*

$$q \approx^{Bis} s$$

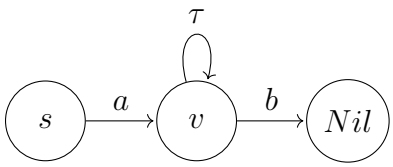
ma:

$$q \not\approx^{Bis} s$$

avendo:



e

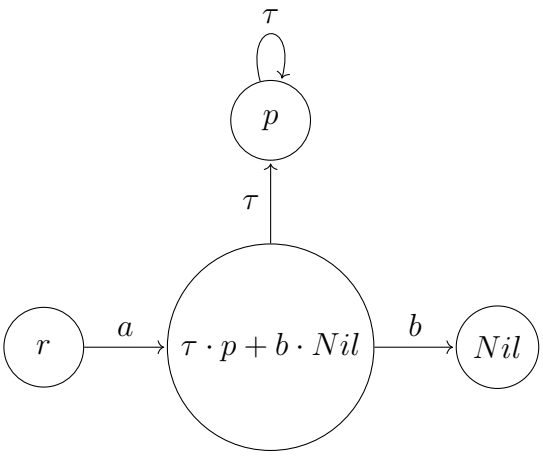


Esempio 41. *Presi $q = a \cdot b \cdot Nil$, $r = a \cdot (\tau \cdot p + b \cdot Nil)$ e $s \cdot a \cdot v$ con $v = \tau \cdot v + b \cdot Nil$. Ho che:*

$$q \not\approx^{Bis} r$$

$$s \not\approx^{Bis} r$$

avendo, oltre ai due LTS dell'esercizio precedente:



Ci si chiede quindi se la bisimulazione debole sia o meno una congruenza rispetto agli operatori del CCS.
Si ricorda che:

Definizione 30. *Data una relazione di equivalenza $R \subseteq Proc_{CCS} \times Proc_{CCS}$ essa è una congruenza se $\forall C[\cdot]$, contesto CCS con una certa variabile non specificata "$\cdot$", presi $p, q \in Proc_{CCS}$, ho che:*

$$pRq \implies C[p] \; R \; C[q]$$

Quindi comunque prendo una specifica CCS posso sostituire a piacere i due processi.

Teorema 16. *Si ha che $\forall\, p, q \in Proc_{CCS}$ se $p \approx^{Bis} q$, allora:*

- $a \cdot p \approx^{Bis} a \cdot q, \forall\, a \in Act = A \cup \overline{A} \cup \{\tau\}$
- $p|r \approx^{Bis} q|r \;\; \wedge \;\; r|p \approx^{Bis} r|q, \forall\, r \in Proc_{CCS}$
- $p[f] \approx^{Bis} q[f], \forall\, f$ *funzione di rietichettatura, ricordando che la funzione preserva nomi, co-nomi (quindi l'immagine di un co-nome è uguale al co-nome dell'immagine) e* τ
- $p_{\backslash L} \approx^{Bis} q_{\backslash L}, \forall\, L \subseteq A$, *ricordando che la restrizione rispetto a L comporta che il processo può eseguire le azioni in L solo se sono sincronizzazioni interne*

Si nota che nel teorema precedente abbiamo, in ordine:

- prefisso
- composizione parallela
- rietichettatura
- restrizione

Non si parla quindi dell'operatore di scelta "+" che infatti necessita di ulteriori approfondimenti. Vediamo un esempio:

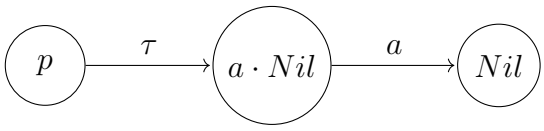
Esempio 42. *Possiamo, per esempio, dire che:*

$$\tau \cdot a \cdot Nil \approx^{Bis} a \cdot Nil$$

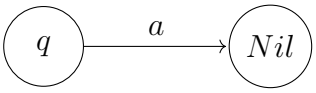
ma:

$$\tau \cdot a \cdot Nil + b \cdot Nil \not\approx^{Bis} a \cdot Nil + b \cdot Nil$$

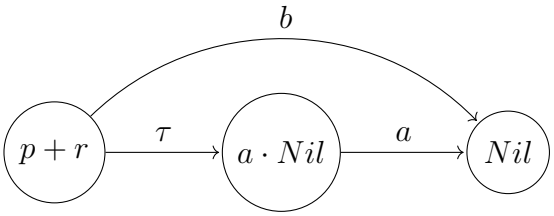
graficamente si ha, per il primo:



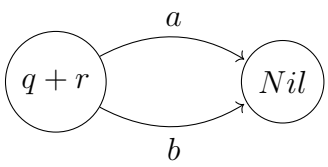
e:



ma componendo $r = b \cdot Nil$ ottengo:



e:



vedendo che non sono bisimili perché $a \cdot Nil$ non può avere corrispondenti nel secondo LTS.

Si ha quindi che la **bisimulazione debole non è una congruenza per il CCS**, a differenza di quella forte (secondo la quale, nell'esempio precedente, avremmo avuto direttamente $p \not\sim^{Bis} q$), in quanto non è una congruenza rispetto alla scelta.
Si ha inoltre che la bisimulazione debole non è una congruenza rispetto alla **ricorsione** (anche se non dimostriamo la cosa).
Possiamo quindi dire che la **bisimulazione debole è una congruenza rispetto agli operatori del CCS diversi da scelta e ricorsione.**
Vogliamo quindi identificare una relazione di congruenza, che è sempre una relazione binaria tra processi CCS, che sia la più grande relazione contenuta nella bisimulazione che sia però una congruenza rispetto a tutti gli operatori:

$$\approx^C \subseteq \approx^{Bis} \subseteq Proc_{CCS} \times Proc_{CCS}$$

Per il CCS puro senza ricorsione Milner ha introdotto un insieme finito di assiomi che possono essere visti come regole di riscrittura che preservano la **congruenza all'osservazione**, che è la più grande congruenza contenuta nella bisimulazione. Questo insieme di assiomi è quindi un insieme finito per il quale, presi due processi $p, q \in Proc_{CCS}$, se riesco, tramite questi assiomi, a trasformare l'uno nell'altro allora si ha che questi processi non sono solo bisimili ma anche congruenti (potendo quindi sostituire l'uno con l'altro in qualsiasi contesto che non includa la ricorsione). Questo insieme di assiomi, detto Ax, è:

- **corretto**, ovvero che se usando l'insieme di assiomi deduco che $p = q$ allora p è congruente a q rispetto a questa nuova nozione si congruenza:
$$Ax \vdash p = q \implies p \approx^C q$$
- **completo**, ovvero presi due processi che sono congruenti secondo questa nuova nozione di congruenza allora sicuramente questo insieme di assiomi è completo in quanto mi permette, tale insieme, di trascrivere p in q, ovvero:
$$p \approx^C q \implies AX \vdash p = q$$

Vediamo quindi questi assiomi:

- $p + (q + r) \approx^C (p + q) + r$ e $p|(q|r) \approx^C (p|q)|r$, ovvero l'assioma riguardo l'associatività di scelta e composizione parallela

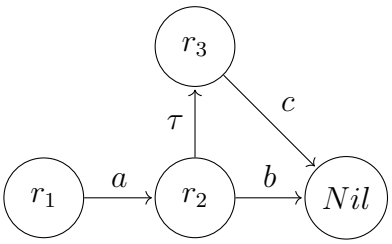
- $p + q \approx^C q + p$ e $p|q \approx^C q|p$, ovvero l'assioma riguardo la commutatività di scelta e composizione parallela
- $p+p \approx^C p$ ma $p|p \not\approx^C p$, ovvero si ha l'assioma per l'assorbimento riguardo la scelta ma non si ha riguardo la composizione parallela (si pensi per esempio a $p = \cdot Nil$ avrei a due volte mentre p può fare a una volta sola)
- $p + Nil \approx^C p$ e $p|Nil \approx^C p$, ovvero ovvero si ha l'assioma per l'assorbimento del Nil riguardo la scelta e la composizione parallela
- $p + \tau \cdot p \approx^C \tau \cdot p$, che è l'assioma che risolve il problema di avere una scelta con un una τ iniziale nella sequenza, che non può essere eliminata
- $\mu \cdot \tau \cdot p \approx \mu \cdot p, \ \forall \mu \in Act$, che tratta le τ interne alla sequenza, che può essere eliminata
- $\mu \cdot (p + \tau \cdot 1) \approx^C \mu \cdot (p + \tau \cdot q) + \mu \cdot q$, ovvero per questo assioma si ha che il primo che è una "semplificazione" del secondo (intuizione da esempio)

Esempio 43. *Vediamo l'ultimo assioma con un esempio.*
Siano:

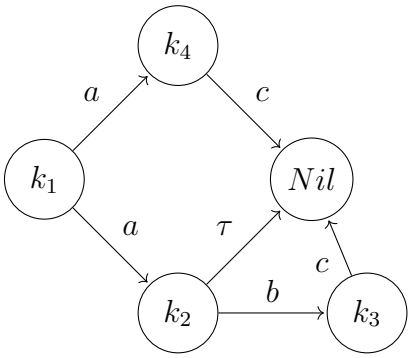
$$r_1 = a \cdot (b \cdot Nil + \tau \cdot c \cdot Nil)$$

$$k_1 = a \cdot (b \cdot Nil + \tau \cdot c \cdot Nil) + a \cdot c \cdot Nil$$

ovvero:



e:



e si ha vede che:

$$r_1 \approx^C k_1$$

con il primo che è una "semplificazione" del secondo.

Si hanno anche altri assiomi legati al fatto che p e q siano delle somme, ovvero:

$$p = \sum_i a_i \cdot p_i, \;\; a \in Act$$

$$q = \sum_j b_j \cdot q_j, \;\; b \in Act$$

avendo quindi i seguenti assiomi:

- $p|q \approx^C \sum_i a_i \cdot (p_i|q) + \sum_j b_j \cdot (p|q_j) + \sum_{a_i=\overline{b_j}} \tau \cdot (p_i|q_j)$ che è l'assioma detto **teorema di espansione di Milner** (nell'ultima parte ho che se a_i e b_j sono l'uno il complemento dell'altro allora posso sincronizzare su di essi i processi, che diventano una τ e proseguendo con $p_i|q_j$). È la generalizzazione dell'operatore parallelo con la regola d'inferenza vista ad inizio capitolo.

- $p[f] \approx^C \sum_i f(a_i) \cdot (p_i[f]), \;\; \forall\, f$ funzione di etichettatura, ovvero etichettare tutto il processo p corrisponde al fatto di ottenere congruo il processo che ottengo etichettando ogni i-sima azione delle componenti che sono in alternativa con il processo relativo rietichettato con f

- $p_{\setminus L} \approx^C \sum_{a_i, \overline{a_i} \notin L} a_i \cdot (p_{i \setminus L}), \;\; \forall\, L \subseteq A$ quindi considero solo le azioni per le quali non si ha al restrizione

Per cui la bisimulazione risulta tale per cui se usiamo $\approx^C$ si può modellare un sistema a passi successivi sapendo che posso sostituire un sottoprocesso con un altro ottenendo ancora un sistema bisimile al precedente.

Esempio 44. *Vediamo un esempio sull'assioma detto* ***teorema di espansione****.*
Si ha, sviluppando a partire dal primo:

$$a \cdot c \cdot Nil | b \cdot Nil \approx^C$$

$$a \cdot (c \cdot NIl | b \cdot Nil) + b \cdot (a \cdot c \cdot Nil | Nil) \approx^C$$

$$a \cdot (c \cdot (Nil | b \cdot Nil) + b \cdot (c \cdot Nil | Nil)) + b \cdot (a \cdot (c \cdot Nil | Nil)) \approx^C$$

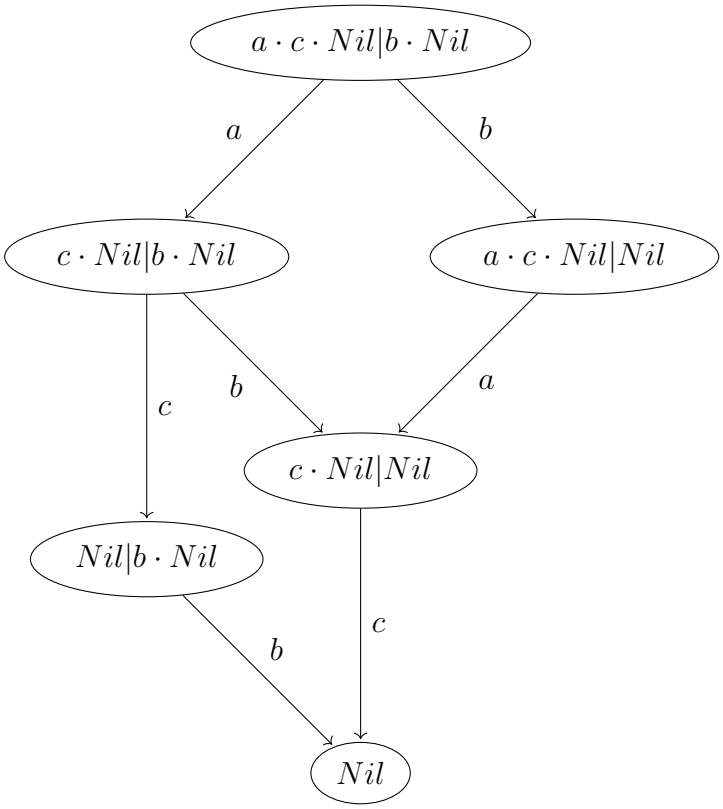
$$a \cdot (c \cdot b \cdot (NIl | Nil) + b \cdot c \cdot (Nil | NIl)) + b \cdot (a \cdot c \cdot (Nil | Nil)) \approx^C$$

$$a \cdot (c \cdot b \cdot Nil + b \cdot c \cdot Nil) + b \cdot a \cdot c \cdot Nil$$

e quindi:

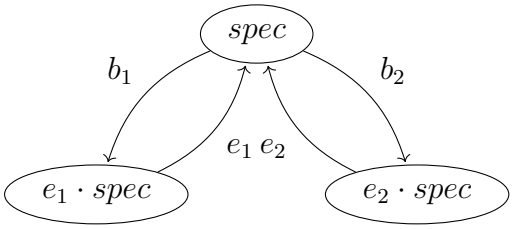
$$a \cdot c \cdot Nil | b \cdot Nil \approx^C a \cdot (c \cdot b \cdot Nil + b \cdot c \cdot Nil) + b \cdot a \cdot c \cdot Nil$$

infatti si ha:



Esempio 45. *Vediamo l'esempio della* ***mutua esclusione*** *con la primitiva di un* ***semaforo****.*
Si ha la specifica:

$$spec = b_1 \cdot e_1 \cdot spec + b_2 \cdot e_2 \cdot spec$$

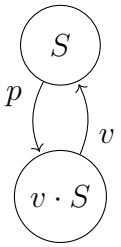


e l'implementazione:

$$sys = (A_1|S|A_2)_{\setminus\{p,v\}}$$

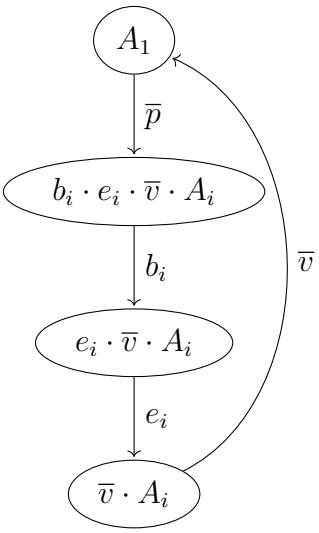
con:

$$S = p \cdot v \cdot S$$

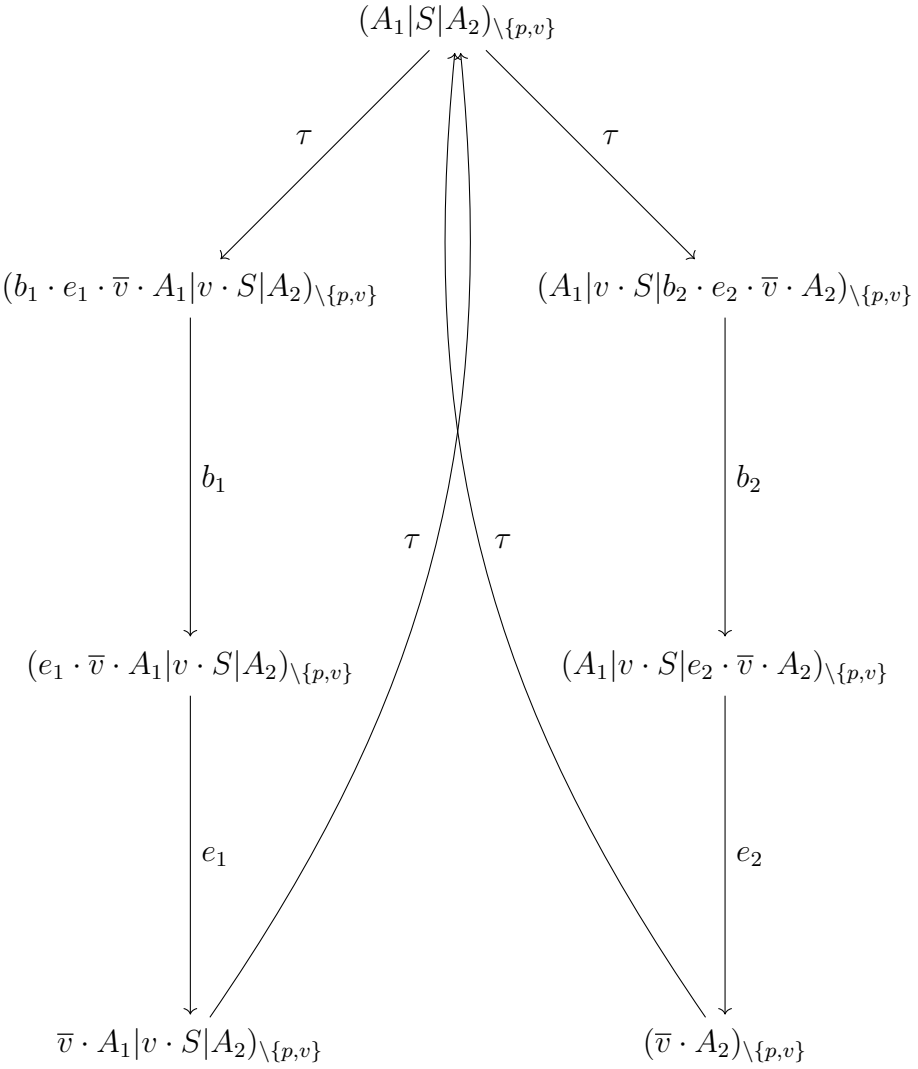


e con:

$$A_1 = \overline{p} \cdot b_2 \cdot e_1 \cdot \overline{v} \cdot A_1$$
$$A_2 = \overline{p} \cdot b_2 \cdot e_2 \cdot \overline{v} \cdot A_2$$



Si ha quindi:



Vedo se spec $\approx^{Bis}$ *sys ma ho che l'attaccante ha una strategia vincente, fa una* τ *sul primo processo in sys. Nella seconda mossa poi l'attaccante opera su spec facendo* b_2 *e il difensore perde:*

$$spec \not\approx^{Bis} sys$$

altri esempi su slide.

Dati due processi p_1 e p_2 abbiamo che la composizione parallela è detta a **semantica interleaving**, in quanto si vede dalle regole di inferenza che posso eseguire o uno o l'altro o sincronizzare, grazie all'assunzione dell'atomicità

delle azioni La cosa è visibile per esempio con:

$$p = a \cdot Nil | b \cdot Nil$$

$$q = a \cdot b \cdot Nil + b \cdot a \cdot Nil$$

Avendo:

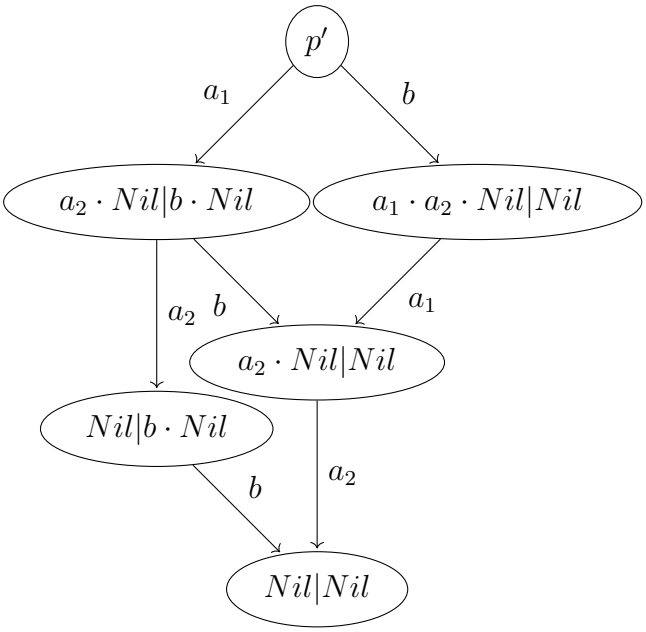
$$p \sim^{Bis} q$$

Togliendo l'atomicità, mettendo a come a_1, a_2 si ha:

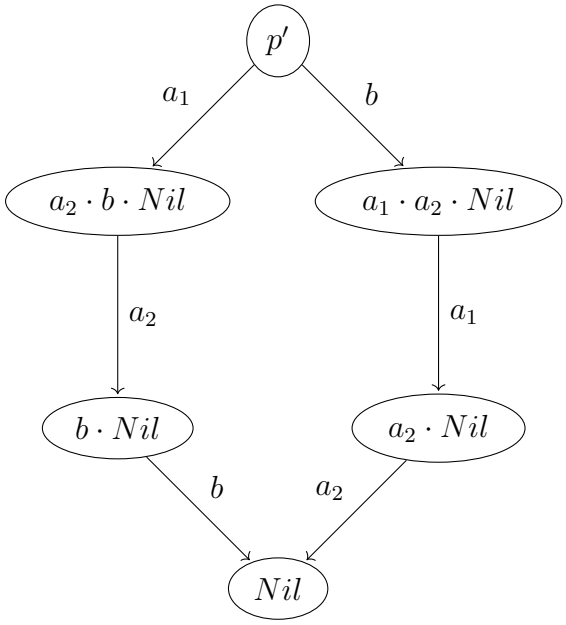
$$p' = p_{[a \leftarrow a_1 \cdot a_2]} = a_1 \cdot a_2 \cdot Nil \mid b.Nil$$

$$q' = q_{[a \leftarrow a_1, a_2]} = a_1 \cdot a_2 \cdot b \cdot Nil + b \cdot a_1 \cdot a_2 \cdot Nil$$

Avendo:



e:



e si ha che $p' \not\sim^T q'$. Se le azioni non sono più atomiche quindi "rompo" la composizione parallela, avrei bisogno infatti di una semantica basata sulla **true concurrency**.
Nella letteratura è stata definita la bisimulazione definita differenziando azioni concorrenti e in sequenza (per esempio tramite le reti di Petri, che si basano sulla *true concurrency*).
Sulle slide esempio di un protocollo di comunicazione fatto con CCS e poi con reti di Petri e ulteriori esempi di partite per la bisimulazione.

Capitolo 6

Reti di Petri

Abbiamo introdotto un'algebra di processi come CCS dove processi sequenziali interagiscono tra loro tramite handshaking. Un altro modello usato, con varie implementazioni, sono gli automi a stati finiti (usati per reti neurali, riconoscitori di linguaggi, modelli di protocolli di comunicazioni etc...).
Si passa ora alle **reti di Petri**, introdotte da Petri nel 1962 nella sua tesi di dottorato, partendo da una critica ai modelli fatti tramite automi a stati per protocolli di comunicazioni.

La nuova teoria dei sistemi è basata sui principi della fisica quantistica, il modello, nell'ottica di Petri, non è quindi un modello chiuso ma è in grado di **comunicare con l'ambiente** (in sintonia con le teorie della fisica). Vuole una teoria dei sistemi in grado di descrivere il flusso di informazioni e che permetta di analizzare sistemi con un'organizzazione complessa.
La sua teoria generale dei sistemi che processano informazione si basa su, tra le altre cose:

- comunicazione
- sincronizzazione
- flusso di informazione
- relazione di concorrenza e indipendenza causale

A partire dagli anni '70 questa teoria si è sviluppata dal punto di vista dell'espressività, portando allo studio di diverse classi di reti. Si sviluppano anche tecniche formali di analisi e di verifica delle reti (tramite algebra lineare, teoria dei grafi etc...). Le reti di Petri hanno tantissimi ambiti d'uso, tra i quali:

- specificare protocolli di comunicazione
- disegnare circuiti asincroni
- algoritmi concorrenti/paralleli
- modellare sistemi organizzativi
- modellare db distribuiti
- specificare sistemi di controllo industriali
- modellare sistemi biologici e per modellare reazioni chimiche
- valutare le prestazioni (tramite reti stocastiche e temporizzate)

Definizione 31. *I **sistemi di transizione etichettati** sono definiti come gli automi a stati finiti ma senza essere visti come riconoscitori di linguaggi infatti un sistema è formato da un insieme, solitamente finito, di stati globali S. Si ha poi un alfabeto delle possibili azioni che può eseguire il sistema. Si hanno anche delle relazioni di transizioni, ovvero delle transizioni che permettono di specificare come, attraverso un'azione, si passa da uno stato ad un altro. Le transizioni si rappresentano con archi etichettati tra i nodi, che rappresentano gli stati. Le etichette degli archi rappresentano le azioni necessarie alla trasformazione. L'insieme delle azioni viene chiamato E mentre $T \subseteq S \times E \times S$ è l'insieme degli archi etichettati. Può essere, opzionalmente, individuato uno stato iniziale s_0. Un sistema non è obbligato a "terminare", quindi non si ha obbligatoriamente uno stato finale.*
Riassumendo quindi un sistema di transizione etichettato è un quadrupla:

$$A = (S, E, T, s_0)$$

La critica di Petri è che in un sistema distribuito non sia individuabile uno **stato globale**, che in un sistema distribuito le trasformazioni di stato siano **localizzate** e non globali, che non esista un sistema di riferimento temporale unico (si possono avere più assi temporali in un sistema distribuito). Quindi la simulazione sequenziale non deterministica (semantica a "interleaving") dei sistemi distribuiti è una forzatura e non rappresenta le reali caratteristiche del comportamento del sistema, ovvero la località, la distribuzione degli eventi e la relazione di dipendenza causale e non causale tra gli eventi.

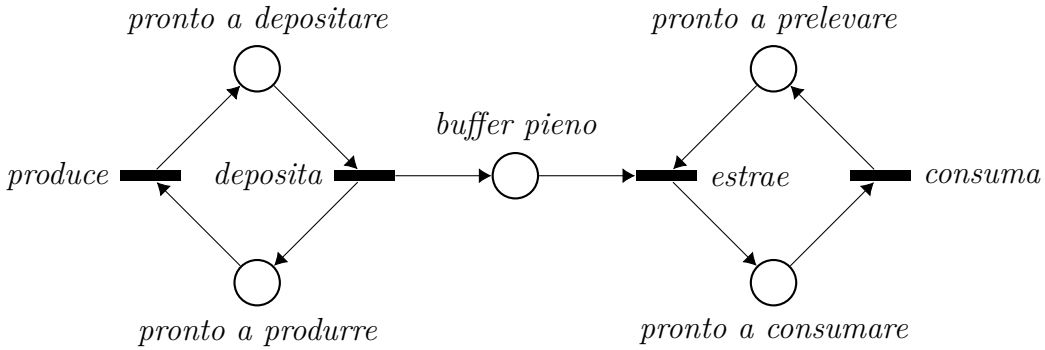
6.1 Sistemi elementari

Introduciamo quindi i **sistemi elementari**.
Per introdurre i sistemi elementari delle reti di Petri, ovvero una classe molto semplice e astratta, partiamo da un esempio:

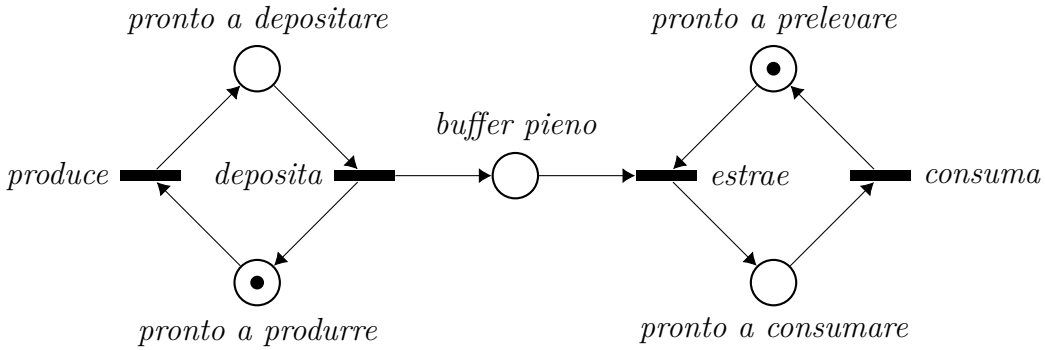
Esempio 46. *Vediamo l'esempio del Produttore e del Consumatore.*
Si ha un sistema con una componente Produttore che produce elementi e li deposita in un buffer che ha un'unica posizione (quindi o è pieno o è vuoto) e con un consumatore che preleva dal buffer un elemento per poi consumarlo ed essere pronto a prelevare un altro elemento. Si ha un comportamento ciclico. Usiamo quindi le reti di Petri, col modello dei sistemi elementari, per rappresentare questo modello. Bisogna quindi individuare le proprietà fondamentali locali del sistema.
Partiamo dal produttore, che può avere 2 stati locali:

1. *pronto per produrre*
2. *pronto per depositare*

*Usiamo i **cerchi** per rappresentare condizioni locali che sono associabili a delle proposizioni della logica che possono essere vere o false. Queste preposizioni sono quindi stati locali. Gli eventi locali vengono invece rappresentati con un **rettangolo**. Un evento ha un arco entrante da uno stato che rappresenta le precondizioni di quell'evento (che devono essere vere per permettere l'occorrenza dell'evento). L'occorrenza dell'evento rende false le precondizioni e rende vere le postcondizioni (che sono stati raggiungibili con un arco uscente da un evento). Si ha quindi che il produttore può depositare solo se il buffer non è pieno, quindi le postcondizioni di un evento devono essere false affinché l'evento possa occorrere (oltre alle precondizioni vere).*
Passiamo al consumatore che estrae solo se il buffer è pieno ed è pronto a prelevare. Si procede poi con la stessa logica del produttore di cambiamento tra vero e falso delle varie condizioni locali.
In questo esempio si hanno quindi condizioni che sono preposizioni booleane e rappresentano stati locali. Si ha quindi a sinistra il produttore, a destra il consumatore e in mezzo il buffer:

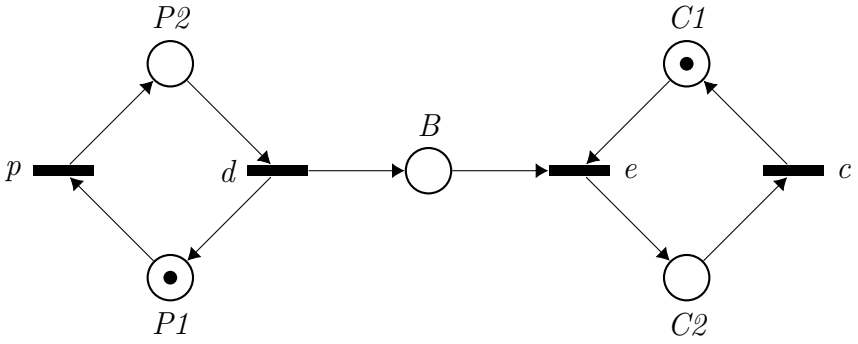


Lo stato globale del sistema è dato da una collezione di stati locali. Per segnare tali condizioni mettiamo un punto pieno dentro il cerchio e queste condizioni "abilitano" i vari eventi:



Si può arrivare ad una configurazione dove, per esempio, sia l'evento produce, del produttore, che l'evento preleva, del consumatore, sono abilitati. Si ha quindi che i due eventi possono occorrere in modo ***concorrente*** *infatti i due eventi sono* ***indipendenti*** *in quanto condizionati da* ***precondizioni e postcondizioni completamente disgiunte****. Due eventi che occorrono in maniera concorrente lo possono fare in qualsiasi ordine, non si ha infatti una sequenza temporale specifica tra i due.*
In questo sistema quindi siano solo stati locali ed eventi localizzati e non stati ed eventi globali. Un evento dipende solo dalle sue precondizioni e dalle sue postcondizioni.
Se rappresentiamo con delle marche le condizioni vere possiamo simulare il comportamento del sistema con il gioco delle marche che mostra come l'evoluzione delle condizioni avviene all'occorrenza degli eventi.
La simula di un tale sistema può comunque avvenire con un sistema di transizioni etichettato, ovvero con un automa a stati finiti, che rappresenta gli stati globali corrispondenti alle diverse combinazioni di stati locali che di volta in volta sono veri.

Gli archi vengono etichettati con gli eventi che comportano un cambiamento di stato globale:



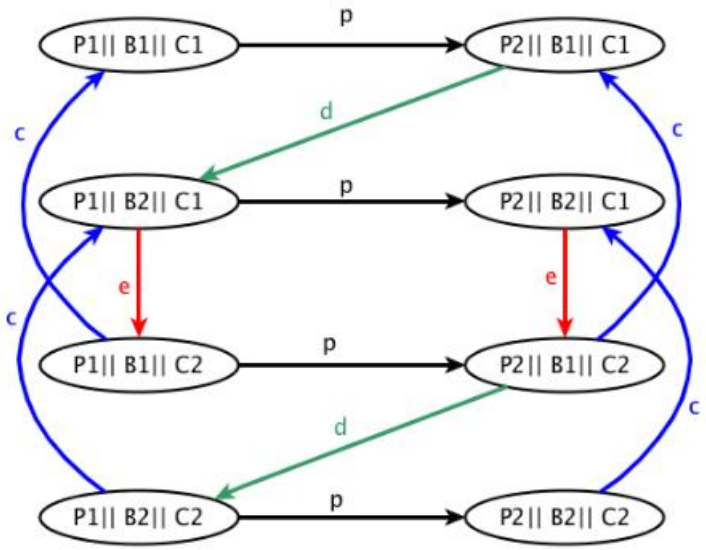


Figura 6.1: Rappresentazione del sistema con un automa a stati finiti che rappresenta stati globali

Lo stesso evento in algebra di processi sarebbe (espandendo il buffer B in un modo che poi specificheremo per specificare pieno e vuoto):

$$S = (P1|B1|C1)_{\{\backslash deposita, estae\}}$$

$$P1 = produce \cdot P2$$

$$P2 = \overline{deposita} \cdot P1$$

$$C1 = estrae \cdot C2$$

$$C2 = consuma \cdot C1$$

$$B1 = deposita \cdot B2$$

$$B2 = \overline{estae} \cdot B1$$

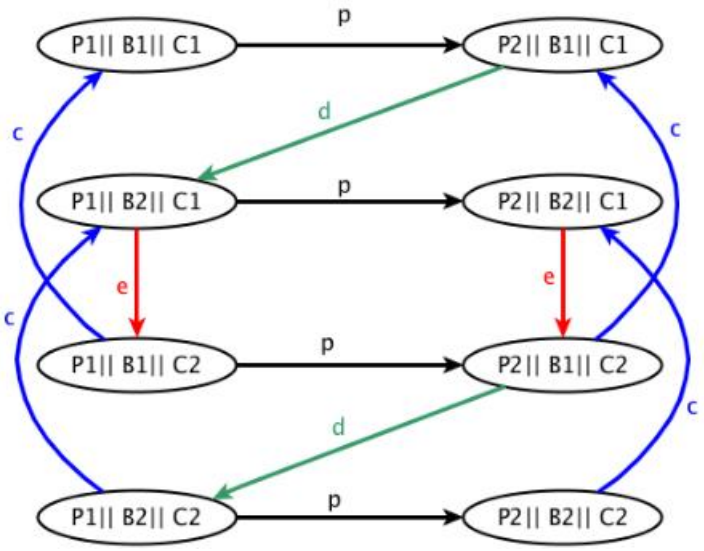


Figura 6.2: Rappresentazione del sistema con un automa a stati finiti che rappresenta stati globali

Passiamo ora alla formalizzazione di questi aspetti.

Definizione 32. *Una **rete elementare** è definita come una tripla:*

$$N = (B, E, F)$$

dove:

- *B è un insieme finito di **condizioni**, ovvero stati locali, proprietà/proposizioni booleane etc.... Vengono rappresentate con un cerchio*
- *E è un insieme finito di **eventi**, ovvero trasformazioni locali di stato e transizioni locali. Vengono rappresentate con un quadrato o con un rettangolo pieno*

- *F è una **relazione di flusso** che connette condizioni ad eventi ed eventi a condizioni. Si ha quindi che:*

$$F \subseteq (B \times E) \cup (E \times B)$$

*Le relazioni di flusso sono rappresentate da archi orientati. Inoltre la relazione di flusso è tale per cui non esistano **elementi isolati**, in quanto non avrebbero senso, in un tale sistema, eventi isolati (che non modificherebbero mai una condizione) o condizioni isolate (che non verrebbero mai modificate da un evento). Si ha, formalmente, che:*

$$dom(F) \cup ran(F) = B \cup E$$

ovvero non ho condizioni/eventi isolati, in quanto non avrebbero senso (potrei anche non modellarle), avrei una condizione costante e un evento che non accade mai, quindi dominio e codominio di F coprono l'insieme di condizioni ed eventi

Si ha che:

$$B \cap E = \emptyset$$

$$B \cup E \neq \emptyset$$

Ovvero gli insiemi delle condizioni e degli eventi sono tra loro disgiunti e non vuoti.
Sia ora x un elemento qualsiasi della rete, ovvero x può essere o una condizione o un evento, formalmente:

$$x \in B \cup E$$

Si ha che, dato $X = B \cup E$:

- *$^\bullet x = \{y \in X : (y, x) \in F\}$ rappresenta l'insieme di tutti gli elementi y che sono connessi dalla relazione di flusso ad x, ovvero si ha un arco da y a x. Sono quindi i **pre-elementi** di x, ovvero le precondizioni, se x è un evento, o i pre-eventi, se x è una condizione*

- *$x^\bullet = \{y \in X : (x, y) \in F\}$ rappresenta l'insieme di tutti gli elementi y che sono connessi dalla relazione di flusso a partire da x, ovvero si ha un arco da x a y. Sono quindi i **post-elementi** di x, ovvero le postcondizioni, se x è un evento, o i post-eventi, se x è una condizione*

Posso estendere questa notazione ad insiemi di elementi. Sia A un insieme qualsiasi di elementi, che possono quindi essere sia condizioni che eventi:

$$A \subseteq B \cup E$$

Si ha quindi che i pre-elementi dell'insieme A sono rappresentati con:

$$^\bullet A = \cup_{x \in A} {}^\bullet x$$

ovvero l'unione dei pre-elementi di ogni singolo elemento dell'insieme A. Analogamente si ha che i post-elementi dell'insieme A sono rappresentati con:

$$A^\bullet = \cup_{x \in A} x^\bullet$$

ovvero l'unione dei post-elementi di ogni singolo elemento dell'insieme A. Nelle reti c'è sempre una relazione di ***dualità*** *tra due elementi, per esempio tra condizioni ed eventi, tra pre-eventi e post-eventi, tra pre-condizioni e post-condizioni. Inoltre si ha la caratteristica della* ***località****, quindi si hanno stati locali e trasformazioni di stato locali*

La rete $N = (B, E, F)$ descrive la *struttura statica del sistema*, il comportamento é definito attraverso le nozioni di **caso (o configurazione)** e di **regola di scatto (o di transizione)**.
Una rete può anche essere suddivisa in sotto-reti, seguendo l'esempio sopra si potrebbe avere una sotto-rete per il produttore, una per il consumatore e anche una per il buffer.

Definizione 33. *Un* ***caso*** *(o* ***configurazione****) é un insieme di condizioni $c \subseteq B$ che rappresentano l'insieme di condizioni vere in una certa configurazione del sistema, un insieme di* ***stati locali*** *che collettivamente individuano lo* ***stato globale*** *del sistema.*
Graficamente le condizioni vere presentano un puntino in mezzo al cerchio mentre le condizioni false solo un cerchio vuoto:

Definizione 34. *Sia $N = (B, E, F)$ una rete elementare e sia $c \subseteq B$ una certa configurazione (non serve quindi necessariamente conoscere tutto lo stato del sistema). La* ***regola di scatto*** *mi permette di stabilire quando un evento $e \in E$ è abilitato, ovvero può occorrere, in c sse:*

$$^\bullet e \subseteq c \ e \ e^\bullet \cap c = \emptyset$$

ovvero sse tutte le precondizioni dell'evento sono vere (e quindi sono contenute nella configurazione c) e sse tutte le postcondizioni sono false (quindi non si hanno intersezioni tra le postcondizioni e la configurazione).
L'occorrenza (l'abilitazione) di e in c si denota con la scrittura:

$$c[e >$$

Se un evento e è abilitato in c, ovvero c[e >, si ha che quando e occorre in c genera un nuovo caso c' e si usa la notazione:

$$c[e > c'$$

Si ha quindi che c' è così calcolabile:

$$c' = (c -^{\bullet} e) \cup e^{\bullet}$$

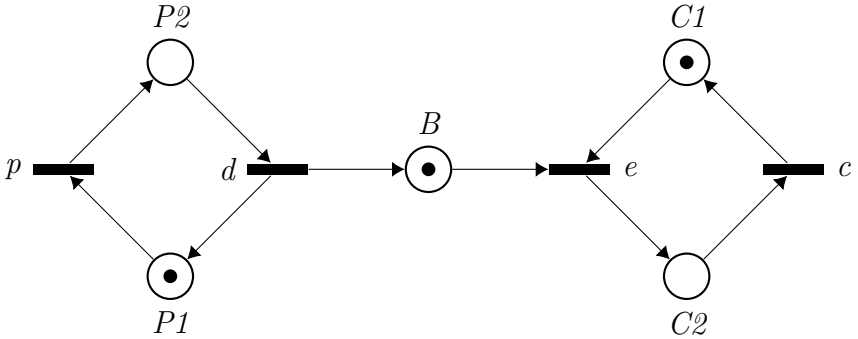
Ovvero togliendo da c tutte le precondizioni dell'evento e e aggiungendo quindi tutte le postcondizioni di e

Le reti si basano sul **principio di estensionalità**, ovvero sul fatto che il cambiamento di stato è locale:

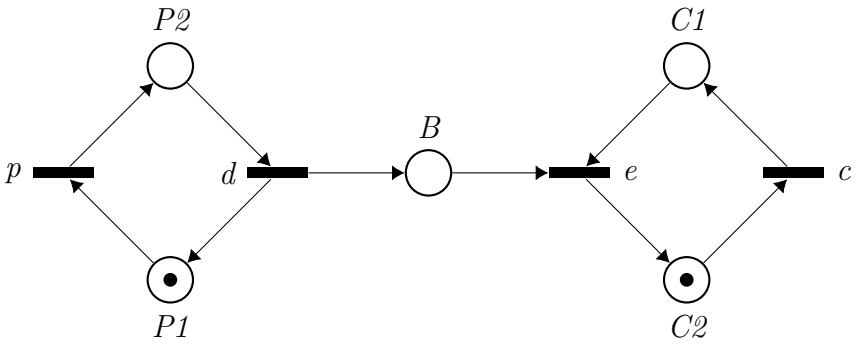
un evento è completamente caratterizzato dai cambiamenti che produce negli stati locali, tali cambiamenti sono indipendenti dalla particolare configurazione in cui l'evento occorre.

L'importante è che le precondizioni di un evento siano vere e le postcondizioni false (siamo comunque interessati solo alla validità delle condizioni che riguardano l'evento).

Esempio 47. *Partendo da:*



Può scattare e e portare a:



Quindi dato $C = \{P1, B, C1\}$ *ho che, dopo lo scatto dell'evento e (possibile visto che tutte le postcondizioni di e sono disabilitate):*

$$C[e > C'$$

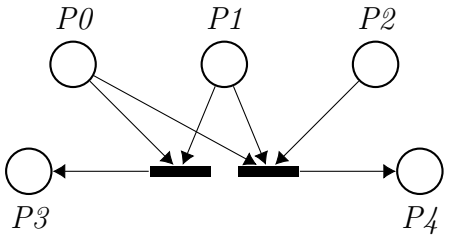
con $C' = \{P1, C2\}$. *(*$P1$ *resta inalterato, per il* ***principio di estensionalità****).*

Definizione 35. *Sia* $N = (B, E, F)$ *una rete elementare. Possiamo definire due tipologie di rete:*

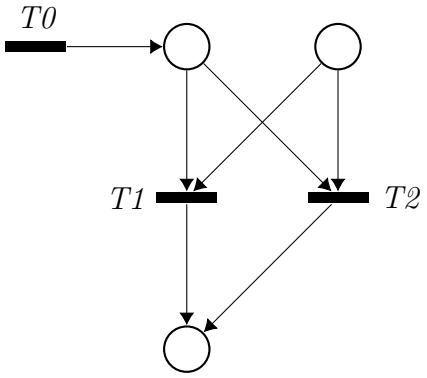
1. *N è definita* ***semplice*** *sse:*

$$\forall x, y \in B \cup E,\ ({}^\bullet x = {}^\bullet y) \wedge (x^\bullet = y^\bullet) \Rightarrow x = y$$

Ovvero per ogni coppia di elementi (che siano quindi eventi o condizioni) se i loro pre-elementi e i loro post-elementi coincidono allora non ha senso distinguere x *e* y. *Vediamo un esempio di rete non semplice, in quanto* $P0$ *e* $P1$ *rappresentano la stessa condizione e quindi non ha senso distinguerli:*



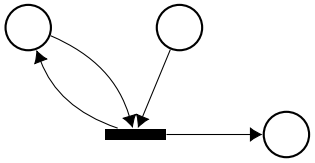
Vale anche per eventi. Vediamo un esempio dove non ha senso distinguere $T1$ e $T2$:



2. *N è definita **pura** sse:*

$$\forall e \in E : \ {}^\bullet e \cap e^\bullet = \emptyset$$

*Ovvero se per ogni evento non esiste una precondizione che sia anche postcondizioni. Si ha quindi un **cappio** (detto anche **side condition**) tra un evento e una condizione. Avere questa situazione comporta che l'evento non può scattare in quanto la condizione che per lui è sia una precondizione che una postcondizioni non può essere contemporaneamente vera e falsa, l'evento non potrà mai scattare e quindi non potrà mai essere osservato. Non avrebbe quindi senso modellarlo. Vediamo un esempio:*



Considereremo sempre reti pure.

Definizione 36. *Data una rete elementare $N = (B, E, F)$ e sia $U \subseteq E$ un sottoinsieme di eventi e siano $c, c_1, c_2 \in B$ tre configurazioni. Si ha che:*

- *U è un **insieme di eventi indipendenti** sse:*

$$\forall e_1, e_2 \in U : \ e_1 \neq e_2 \Rightarrow ({}^\bullet e_1 \cup e_1^\bullet) \cap ({}^\bullet e_2 \cup e_2^\bullet) = \emptyset$$

ovvero per ogni coppia distinta di eventi nell'insieme U si ha che le precondizioni e le postcondizioni dei due eventi sono completamente disgiunte.

- *U è un **passo abilitato**, ovvero un insieme di eventi concorrenti in una certa configurazione c, che si indica con:*

$$c[U >$$

sse:

$$U \text{ è un insieme di eventi indipendenti } \wedge\ \forall e \in U :\ c[e >$$

U quindi deve essere un insieme di eventi indipendenti e ogni evento in U è abilitato in c, quindi le sue precondizioni sono vere e le sue postcondizioni sono false. Si ha quindi che U è un insieme di eventi abitati in maniera concorrente in c

- *U è un **passo** dalla configurazione c_1 alla configurazione c_2, che si indica con:*

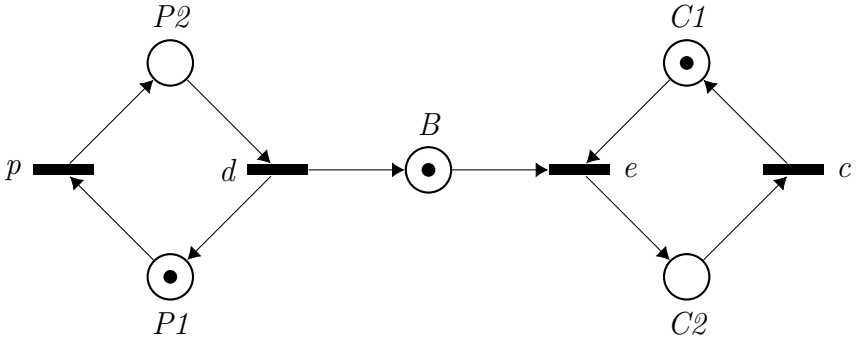
$$c_1[U > c_2$$

sse:

$$(c_1[U) \wedge \left(c_2 = (c_1 -^\bullet U) \cup U^\bullet\right)$$

ovvero sse U è un passo abilitato in c_1 e lo scatto degli eventi in U porta alla configurazione c_2 che si ottiene togliendo da c_1 l'insieme delle precondizioni degli eventi in U e aggiungendo quindi l'insieme delle postcondizioni degli eventi in U.

Esempio 48. *Preso:*



si ha che:

- *$\{p, e\}, \{p, c\}, \{d, c\}$ sono esempi di insiemi di eventi indipendenti*
- *$\{p, e\}$ è un passo abilitato in $\{P_1, B, C_1\}$*

- $\{P_1, B, C_1\}[\{p, w\} > \{P_2, C_2\}$ *è lo scatto del passo* $\{p, e\}$ *ci porta in* $\{P_2, C_2\}$

Diamo ora una definizione formale di **sistema elementare**.

Definizione 37. *Un* ***sistema elementare*** $\Sigma = (B, E, F; c_{in})$ *è definito come una rete* $N = (B, E, F)$ *e a cui è associato un caso iniziale, una configurazione iniziale, ovvero un sottoinsieme di condizioni che rappresentano lo stato iniziale da cui inizia la computazione e l'evoluzione del sistema. Formalmente il caso iniziale si indica con* $C_{in} \in B$

Definizione 38. *Dato un sistema elementare* $\Sigma = (B, E, F; c_{in})$ *si indica con* C_Σ *l'insieme dei* ***casi raggiungibili*** *da tale sistema a partire dal caso iniziale* c_{in}*.*
Formalmente l'insieme dei casi raggiungibili è il di piccolo sottoinsieme dell'insieme delle parti di B*, ovvero* 2^B*, tale che:*

- $c_{in} \in C_\Sigma$*, ovvero sicuramente il caso iniziale appartiene all'insieme dei casi raggiungibili*
- *se* $c \in C_\Sigma$*,* $U \subseteq E$ *e* $c' \subseteq B$ *sono tali che* $c[U > c'$ *allora* $c' \in C_\Sigma$*, ovvero se ho un generico caso c che appartiene ai casi raggiungibili, se ho un insieme di eventi U tale che questo insieme di eventi (che abbiamo visto essere indipendenti, per la definizione di passo abilitato) è abilitato in c in un unico passo e la sua occorrenza mi porta in* c'*, allora anche* c' *appartiene a* C_Σ*.*

Questa è una definizione data per ***induzione strutturale****, nel primo punto si ha la base, nel secondo l'ipotesi e la conseguenza*

Definizione 39. *Dato un sistema elementare* $\Sigma = (B, E, F; c_{in})$ *si indica con* U_Σ *l'****insieme dei passi*** *di* Σ*, ovvero di tutti i possibili insiemi di eventi indipendenti che possono occorrere in qualche caso. Formalmente:*

$$U_\Sigma = \{U \subseteq E \mid \exists\, c, c' \in C_\Sigma : \; c[U > c'\}$$

Ovvero l'insieme dei sottoinsiemi di eventi tali per cui esistano due casi raggiungibili in C_Σ *e* U *è abilitato in* c *e il suo scatto mi porta in* c'*.*

Definiamo ora il comportamento dei sistemi elementari.

Definizione 40. *Sia* $\Sigma = (B, E, F; c_{in})$ *un sistema elementare e siano* $c_i \in C_\Sigma$ *ed* $e_i \in E$*.*
Definiamo;

- *un* ***comportamento sequenziale*** *come una sequenza di eventi che possono occorrere dal caso iniziale. Facendo scattare in maniera sequenziale gli eventi uno alla volta in c_n:*

  $$c_{in}[e_1 > c_1[e_2 > \ldots [e_n > c_n$$

  Scrittura che può essere alleggerita in:

  $$c_{in}[e_1 e_2 \ldots e_n > c_n$$

  Possiamo dire di avere a che fare con una ***simulazione sequenziale non deterministica, detta anche semantica a interleaving****, infatti ho più eventi abilitati da prendere uno alla volta*

- *un* ***comportamento non sequenziale****, in quanto possiamo anche considerare insiemi di eventi, ovvero passi. Considero quindi sequenze di passi, avendo a che fare con la* ***step semantics****. Non ho quindi una simulazione sequenziale non deterministica in quanto dal caso iniziale faccio scattare un insieme di eventi, in maniera concorrente (e quindi senza ordine specificato), per poi far scattare un altro insieme di eventi fino ad arrivare a c_n:*

  $$c_{in}[U_1 > c_1[U_2 > \ldots [U_n > c_n$$

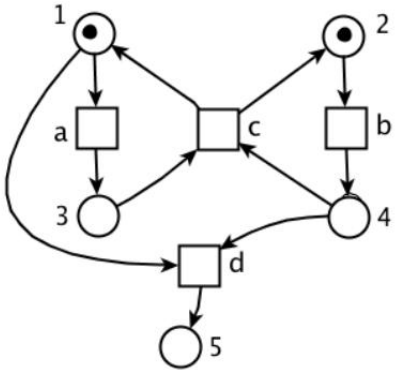
  Scrittura che può essere alleggerita in:

  $$c_{in}[U_1 U_2 \ldots U_n > c_n$$

  Gli insiemi U_i non sono insiemi massimali abilitati ma sottoinsiemi indipendenti e abilitati in c_{in}.
  Posso avere anche un altro tipo di ***comportamento non sequenziale****, definito da Petri stesso, in una* ***semantica ad ordini parziali*** *(si parla anche di* ***true concurrency****) in cui si definiscono processi non sequenziali. Il comportamento di tale sistema viene registrato in una rete di Petri*

In ogni caso si considerano sia sequenze finite che infinite (con cicli) di eventi o passi.

Esempio 49. *Dato il sistema elementare Σ:*



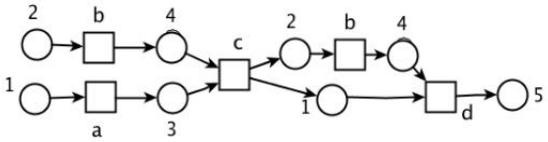
si ha, per esempio, la seguente sequenza di occorrenza di eventi:

$$\{1,2\}[a>\{3,2\}[b>\{3,4\}[c>\{1,2\}[b>\{1,4\}[d>\{5\}$$

arrivati in "5" abbiamo un caso finale, ovvero una situazione di ***deadlock****, in quanto il sistema non può evolvere ulteriormente.*
Vediamo anche la seguente possibile sequenza di passi. In "1" e "2" sia "a" che "b" sono indipendenti e sono entrambi abilitati (scattano in maniera concorrente in un unico passo, ovviamente posso avere passi con lo scatto di un solo evento):

$$\{1,2\}[\{a,b\}>\{3,4\}[\{c\}>\{1,2\}[\{b\}>\{1,4\}$$

Come ricordato posso finire in una sequenza infinita.
Potrei modellare un processo non sequenziale di Σ:



Vediamo ora come modellare e registrare il comportamento del sistema. Un modo è usando il **grafo dei casi raggiungibili** (equivalentemente a quanto si faceva con il CCS).

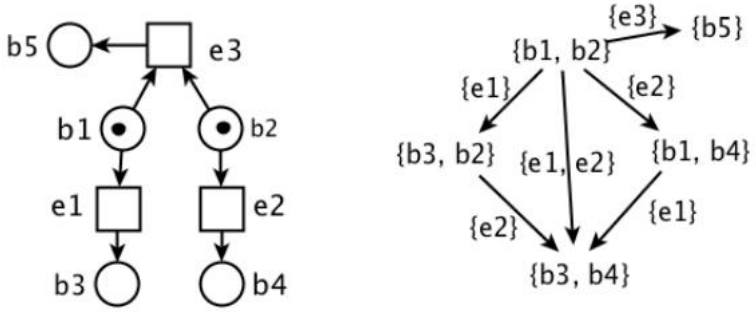


Figura 6.3: Esempio di sistema Σ e il suo grafo dei casi del sistema Σ

Definizione 41. *Il **grafo dei casi raggiungibili** di un sistema elementare $\Sigma = (B, E, F; c_{in})$ è il sistema di transizioni etichettato:*

$$CG_\Sigma = (C_\Sigma, U_\Sigma, A, c_{in})$$

dove:

- *C_Σ è l'insieme dei nodi del grafo, ovvero gli stati globali, i casi raggiungibili dal sistema Σ*
- *U_Σ, è l'alfabeto, ovvero i passi del sistema rappresentano l'alfabeto*
- *A è l'insieme di archi etichettati, formalmente definito come:*

  $$A = \{(c, U, c') |\, c, c' \in C_\Sigma, U \in U_\Sigma, c[U > c'\}$$

  ovvero sono archi che connettono uno caso c con un caso c' e sono etichettati con un passo U sse U è abilitato in c e porta in c'. Ovviamente c e c' sono devono essere raggiungibili e U deve appartenere all'insieme dei passi di Σ

Esempio in figura 6.3.

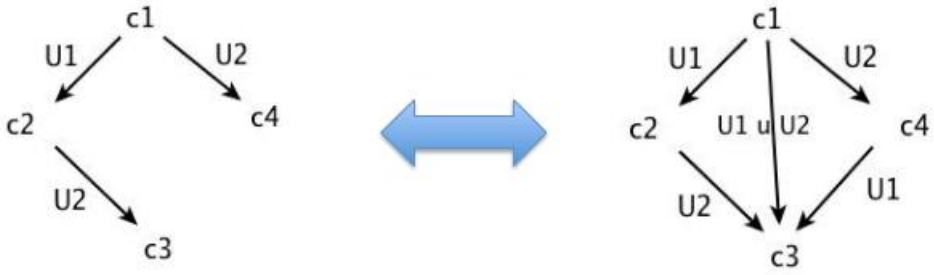
6.1.1 Diamond Property

Dato un sistema elementare $\Sigma = (B, E, F; c_{in})$ e il suo grafo dei casi $CG_\Sigma = (C_\Sigma, U_\Sigma, A, c_{in})$ si ha che il grafo soddisfa una particolare proprietà, detta **diamond property**, tipica solo dei sistemi elementari.

Definizione 42. *La **diamond property** stabilisce una proprietà della struttura del grafo della rete elementare, ovvero, dati $U_1, U_2 \in U_\Sigma$ tali che:*

- $U_1 \cap U_2 = \emptyset$
- $U_1 \neq \emptyset$
- $U_2 \neq \emptyset$

e dati $c_i \in C_\Sigma$ *allora vale, per esempio:*



ovvero se posso rilevare come sottografo una struttura come quella a sinistra nell'immagine allora sicuramente tale sottografo contiene anche gli archi per ottenere l'immagine di destra.

Si possono fare delle prove:

1. **prima prova:**
   Dimostriamo che possiamo passare all'immagine di destra da quella di sinistra aggiungendo i due archi mancanti, nella figura 42.
   Per semplicità diciamo che U_i è un singolo evento e_i, con $i = 1, 2$. Siano inoltre $c_1, c_2 \in C_\Sigma$, ovvero sono casi raggiungibili, ed $e_1, e_2 \in E$ tali che $c_1[e_1 > c_2[e_2 >$ e $c_1[e_2 >$, i due eventi quindi sono abilitati in sequenza e da c_1 è anche abilitato c_2 . Si vuole dimostrare che:

   $$({}^\bullet e_1 \cup e_1^\bullet) \cap ({}^\bullet e_2 \cup e_2^\bullet) = \emptyset$$

   ovvero che i due eventi sono indipendenti, che sono entrambi abilitati e che sono eseguibili in qualsiasi ordine.

Da $c_1[e_1 >$ e $c_1[e_2 >$ segue che:

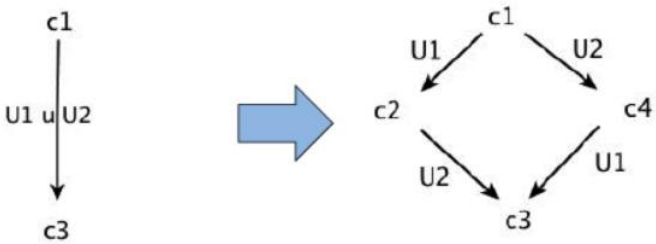
- ${}^\bullet e_1 \cap e_2^\bullet = \emptyset$
- ${}^\bullet e_2 \cap e_1^\bullet = \emptyset$

infatti se e_1 e e_2 sono entrambi abilitati in c_1, le loro pre-condizioni sono vere e le post-condizioni false, e quindi non è possibile che una condizione sia contemporaneamente precondizione di e_1 (vera) e anche postcondizione di e_2 (falsa), e viceversa. Quindi le precondizioni di un evento sono disgiunte dalle postcondizioni dell'altro. Inoltre dal fatto che ho $c_1[e_1 > c_2[e_2$, ovvero che da c_1 è abilitato e_1 e che dopo lo scatto di e_1 è ancora abilitato e_2 possiamo dire che:

- $e_1^\bullet \cap e_2^\bullet = \emptyset$
- ${}^\bullet e_1 \cap {}^\bullet e_2 = \emptyset$

in c_2, infatti, le pre-condizioni di e_1 sono false mentre le precondizioni di e_2 sono vere e quindi e_1 e e_2 non possono avere precondizioni in comune; inoltre sempre in c_2 le postcondizioni di e_1 sono vere, mentre quelle di e_2 sono false, e quindi e_1 e e_2 non possono avere post-condizioni in comune. Quindi le precondizioni dei due eventi sono disgiunte, come del resto anche le postcondizioni, in quanto i due eventi sono sequenziali.
Si è quindi dimostrato che i due eventi hanno precondizioni e postcondizioni completamente disgiunte e quindi la tesi è verificata

2. **seconda prova:**
   Analizzando la situazione:



Si supponga che $U_1 \cup U_2 \in U_\Sigma$ e che si abbiano:

- $U_1 \cap U_2 = \emptyset$, ovvero sono disgiunti
- $U_1 \neq \emptyset$
- $U_2 \neq \emptyset$

allora se $c_1[(U_1 \cup U_2) > c_3$, quindi è abilitato il passo $U_1 \cup U_2$ in c_1, sicuramente si ha che sono abilitati anche i singoli passi:

- $c_1[U_1 >$
- $c_1[U_2 >$

resta da dimostrare che dopo lo scatto di U_1 è ancora abilitato U_2 in c_2. Ma se $U_1 \cup U_2$ è un passo abilitato significa che posso eseguirli in qualsiasi ordine, quindi anche prima U_1 e poi U_2, e questo comporta sicuramente che U_2 è abilitato e che porta a c_3. Analogamente invertendo U_1 e U_2, formalmente:

- $c_1[U_1 > c_2[U_2 > c_3$
- $c_1[U_2 > c_4[U_1 > c_3$

Si dimostra così che l'immagine di sinistra comporta quella di destra.

Grazie alla diamond property possiamo non considerare il grafo dei casi raggiungibili ma solo il **grafo dei casi sequenziale**:

Definizione 43. *Un **grafo dei casi sequenziale** del sistema elementare $\Sigma = (B, E, F; c_{in})$ è una quadrupla:*

$$SCG_\Sigma = (C_\Sigma, E, A, c_{in})$$

dove le etichette sono i singoli eventi (mentre il resto rimane definito come nel grafo dei casi raggiungibili). Formalmente si ha quindi che:

$$A = \{(c, e, c') |\, c, c' \in C_\Sigma,\, e \in E :\, c[e > c'\}$$

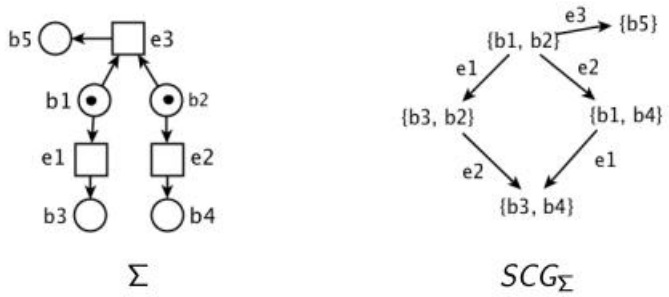


Figura 6.4: Esempio di grafo dei casi sequenziale

Si registra quindi l'occorrenza di un evento alla volta. Il grafo dei casi sequenziale è quindi il sistema di transizione con gli archi etichettati dai singoli eventi.

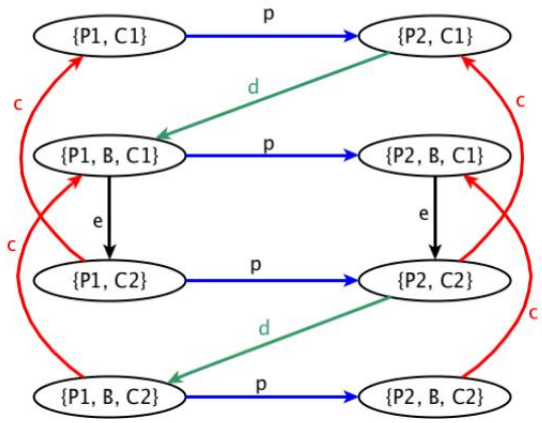


Figura 6.5: Esempio di grafo dei casi sequenziale dell'esempio con produttore e consumatore (con marche in $P1$ e $C1$)

Riprendendo l'immagine precedente si ha che per la diamond property possono aggiungere l'arco "centrale" che trasformerebbe nuovamente il grafo dei casi sequenziale in quello dei casi raggiungibili quindi: *per la diamond property, nei sistemi elementari il grafo dei casi e il grafo dei casi sequenziale sono* ***sintatticamente equivalenti****, ovvero possono essere ricavati a vicenda.*
Questo implica il fatto che due sistemi elementari hanno grafi dei casi ***isomorfi*** *sse hanno grafi dei casi sequenziali isomorfi.*

6.1.2 Isomorfismo tra Sistemi di Transizione Etichettati

Si ricorda che:

Si parla di isomorfismo quando due strutture complesse si possono applicare l'una sull'altra, cioè far corrispondere l'una all'altra, in modo tale che per ogni parte di una delle strutture ci sia una parte corrispondente nell'altra struttura; in questo contesto diciamo che due parti sono corrispondenti se hanno un ruolo simile nelle rispettive strutture.

Diamo ora una definizione formale di isomorfismo tra sistemi di transizione etichettati, che possono quindi essere grafi dei casi o grafi dei casi sequenziali.

Definizione 44. *Siano dati due sistemi di transizione etichettati:*
$A_1 = (S_1, E_1, T_1, s_{01})$ *e* $A_2 = (S_2, E_2, T_2, s_{02})$.
e siano date due ***mappe biunivoche****:*

1. $\alpha : S_1 \to S_2$, *ovvero che passa dagli stati del primo sistema a quelli del secondo*
2. $\beta : E_1 \to E_2$, *ovvero che passa dagli eventi del primo sistema a quelli del secondo*

allora:

$$\langle \alpha, \beta \rangle : A_1 = (S_1, E_1, T_1, s_{01}) \to A_2 = (S_2, E_2, T_2, s_{02})$$

è un ***isomorfismo*** *sse:*

- $\alpha(s_{01}) = s_{02}$, *ovvero l'immagine dello stato iniziale del primo sistema coincide con lo stato iniziale del secondo*
- $\forall s, s' \in S_1, \forall e \in E_1 : (s, e, s') \in T_1 \Leftrightarrow (\alpha(s), \beta(e), \alpha(s')) \in T_2$ *ovvero per ogni coppia di stati del primo sistema, tra cui esiste un arco etichettato e, vale che esiste un arco, etichettato con l'immagine di e, nel secondo sistema che va dall'immagine del primo stato considerato del primo sistema all'immagine del secondo stato considerato del secondo sistema, e viceversa*

Definizione 45. *Si definiscono due* ***sistemi equivalenti*** *sse hanno grafi dei casi sequenziali, e quindi di conseguenza anche grafi dei casi,* isomorfi.
Due sistemi equivalenti accettano ed eseguono le stesse sequenze di eventi

6.1.3 Il Problema della Sintesi

Si presenta ora un problema tipico dell'informatica, il **problema della sintesi**, ovvero dato un comportamento, o meglio una sua specifica, decidere se esiste un'implementazione di tale specifica, ovvero un modello, che abbia esattamente quel comportamento.
In questo caso dato un sistema di transizioni etichettato $A = (S, E, T, s_0)$, con:

- S insieme degli stati
- E insieme delle etichette, ovvero degli eventi
- T insieme delle transizioni
- s_0 stato iniziale

ci si propone di stabilire se esiste un sistema elementare $\Sigma = (B, E, F; c_{in})$, tale che l'insieme degli eventi del sistema esattamente l'insieme delle etichette di A e tale che il suo grafo dei casi SCG_Σ sia isomorfo ad A. Ci si propone anche di costruirlo.
Il problema è stato risolto mediante la cosiddetta **teoria delle regioni** (che però non verrà trattato nel corso). Una **regione** comunque è un particolare sottoinsiemi di stati, legati tramite una certa condizione. Si può però dire che A dovrà soddisfare la diamond property, in quanto altrimenti non sarebbe un sistema di transizioni che potrebbe corrispondere al comportamento di un sistema elementare.

6.1.4 Contatti

Definizione 46. *Sia $\Sigma = (B, E, F; c_{in})$ un sistema elementare e siano $e \in E$ un evento e $c \in C_\Sigma$ un caso raggiungibile dal caso iniziale. Allora si ha che (e, c) è un **contatto** sse:*

$$^\bullet e \subseteq c \wedge e^\bullet \cap c \neq \emptyset$$

Ovvero, in termini pratici, siamo nel caso in cui un evento e ha le precondizioni vere, si ha quindi che $^\bullet e \subseteq c$, e l'evento non ha tutte le postcondizioni false, quindi $e^\bullet \cap c \neq \emptyset$, allora si dice che l'evento e è in una situazione di contatto e quindi non può scattare

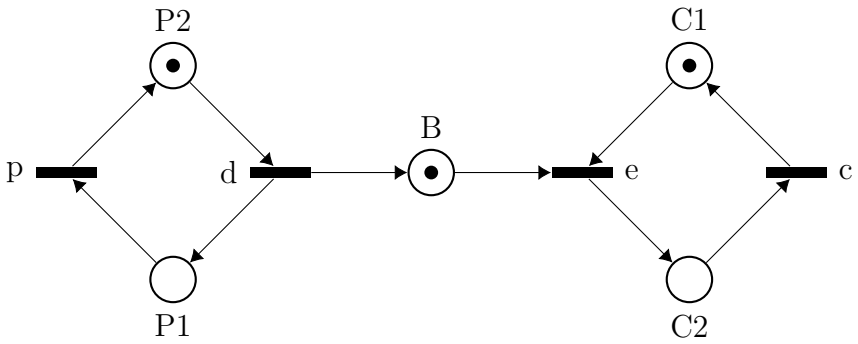


Figura 6.6: Esempio dove l'evento d è in una situazione di contatto

Definizione 47. *Sia $\Sigma = (B, E, F; c_{in})$ un sistema elementare. Si dice che il sistema è* ***senza contatti*** *sse:*

$$\forall e \in E,\ \forall c \in C_\Sigma \text{ si ha che } {}^\bullet e \subseteq c \Rightarrow e^\bullet \cap c = \emptyset$$

ovvero per ogni evento e per ogni caso raggiungibile dal caso iniziale succede sempre che se le precondizioni sono vere, ovvero ${}^\bullet e \subseteq c$, allora le postcondizioni sono false, ovvero disgiunte dal caso considerato ($e^\bullet \cap c = \emptyset$)

Ci si chiede se sia possibile trasformare un sistema elementare Σ, con contatti, in uno Σ', senza contatti, senza però modificarne il comportamento.
La risposta a questo quesito è affermativa e la procedura consiste nell'aggiungere a Σ il complemento di ogni condizione che crea situazione di contatto, ottenendo così un sistema Σ' con grafo dei casi isomorfo a quello di Σ.
Per aggiungere il complemento, data la condizione x, si aggiunge la condizione *not* x che sarà vera tutte le volte che x è falsa e viceversa. Per ottenere questo risultato la nuova condizione avrà come pre-eventi i post-eventi di x e come post-eventi i pre-eventi di x. Ovvero connetto la nuova condizione agli stessi eventi di quella vecchia ma con archi orientati in senso opposto. Ovviamente le inizializzazioni delle due condizioni dovranno essere opposte (una vera e l'altra falsa).

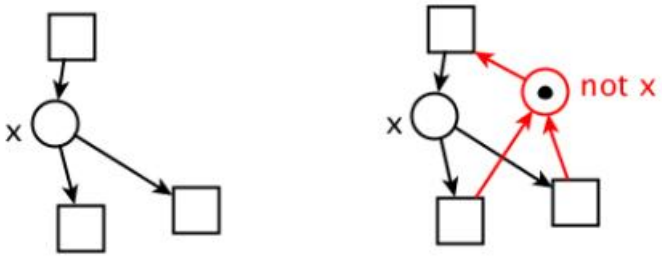


Figura 6.7: Esempio con l'ottenimento del complemento di un sistema

Se un sistema è senza contatti si ha una regola di contatto semplificata:

Definizione 48. *Sia $\Sigma = (B, E, F; c_{in})$ un sistema elementare* ***senza contatti****. Sapendo che se le precondizioni di un evento sono vere allora sicuramente le postcondizioni di quell'evento sono false in quel caso. Dato che questo avviene per ogni evento e per ogni caso raggiungibile dal caso iniziale per verificare che un evento e sia abilitato in un caso raggiungibile c è sufficiente verificare che le precondizioni di e siano vere (in quanto automaticamente le postcondizioni saranno false). In maniera formale quindi si ha che:*

$$c[e \text{ sse } {}^\bullet e \subseteq c, \quad con\ e \in E, c \in C_\Sigma$$

semplificando di molto la ***regola di scatto****:*

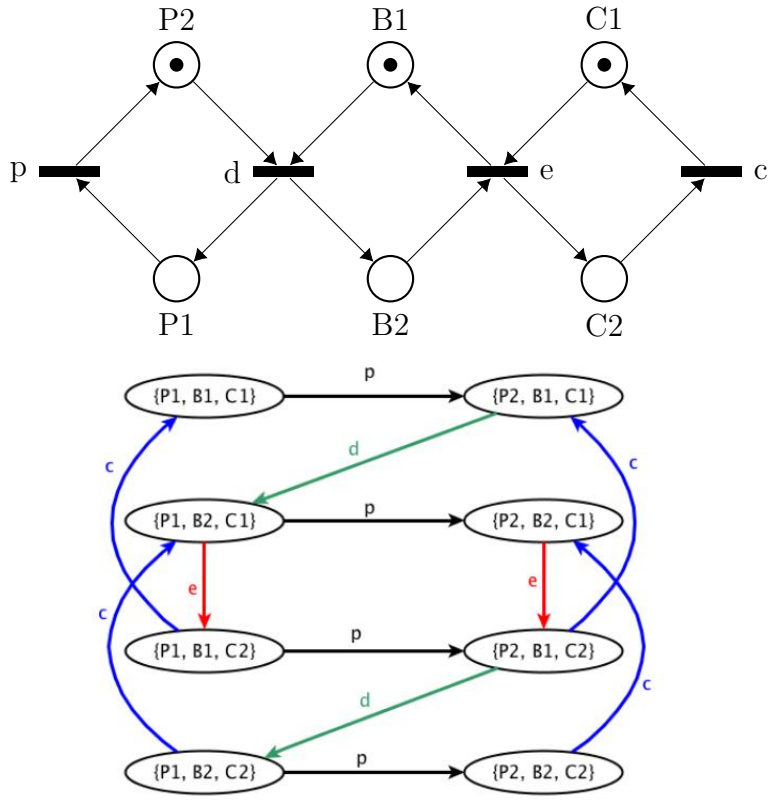


Figura 6.8: Esempio con il complemento del sistema produttore-consumatore (dove è stato aggiunto solo il complemento di "buffer-pieno", ottenendo così sia $B1$ che $B2$, in quanto le altre condizioni avevano già il loro complemento). In aggiunta si ha anche il grafo dei casi sequenziale corrispondente al nuovo sistema senza contatti (grafo che è isomorfo a quello ottenibile al sistema con contatti)

6.1.5 Situazioni Fondamentali

Sequenza

Definizione 49. *Sia $\Sigma = (B, E, F; c_{in})$ un sistema elementare, con contatti o meno e siano $c \in C_\Sigma$ un caso raggiungibile dal caso iniziale e $e_1, e_2 \in E$ due eventi.*
*Si ha che e_1 ed e_2 sono **in sequenza** nel caso raggiungibile c sse:*

$$c[e_1 > \wedge \neg c[e_2 \wedge c[e_1 e_2 >$$

ovvero in c è abilitato e_1 ma non e_2 ma, dopo lo scatto di e_1, e_2 diventa abilitato. Quindi in c è possibile attivare prima e_1 e poi e_2 in sequenza.
*Si ha quindi una relazione di **dipendenza causale tra** e_1 **ed** e_2, ovvero qualche postcondizione di e_1 è precondizione di e_2 (che quindi può occorrere solo se precedentemente è occorso e_1).*

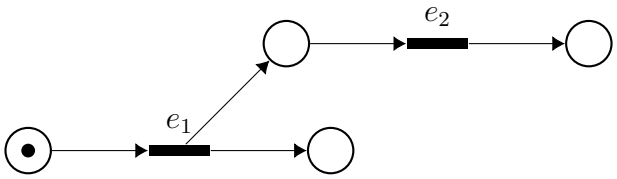


Figura 6.9: Esempio di sequenza tra e_1 ed e_2

Concorrenza

Definizione 50. *Sia $\Sigma = (B, E, F; c_{in})$ un sistema elementare, con contatti o meno e siano $c \in C_\Sigma$ un caso raggiungibile dal caso iniziale e $e_1, e_2 \in E$ due eventi.*
*Si ha che i due eventi sono **concorrenti** nel caso raggiungibile c sse:*

$$c[\{e_1, e_2\} >$$

ovvero se possono essere abilitati in unico passo o, detto in maniera diversa, se sono indipendenti ed entrambi abilitati in c

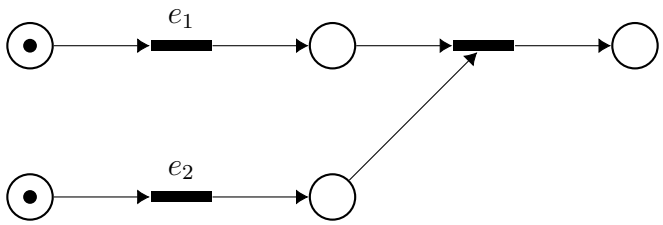


Figura 6.10: Esempio di concorrenza tra e_1 ed e_2

Conflitto

Definizione 51. *Sia $\Sigma = (B, E, F; c_{in})$ un sistema elementare, con contatti o meno e siano $c \in C_\Sigma$ un caso raggiungibile dal caso iniziale e $e_1, e_2 \in E$ due eventi.*
Si ha che e_1 ed e_2 sono in conflitto sse:

$$c[e_1 > \wedge c[e_2 \wedge \neg c[\{e_1, e_2\} >$$

ovvero i due eventi sono entrambi abilitati (quindi le precondizioni sono vere mentre le postcondizioni son false) ma l'occorrenza di uno disabilità l'altro, quindi non possono essere abilitati in un unico passo, in quanto non sono indipendenti. Ci sono due casi:

1. *i due eventi hanno una precondizione in comune, e in tal caso si parla di **conflitto forward** (ovvero in avanti)*

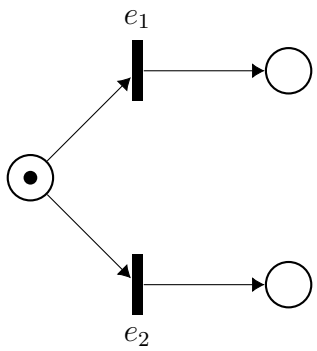


Figura 6.11: Esempio di conflitto forward tra e_1 ed e_2

2. *i due eventi hanno una postcondizione in comune, e in tal caso si parla di **conflitto backward** (ovvero all'indietro)*

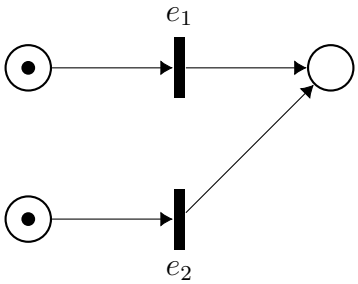


Figura 6.12: Esempio di conflitto backward tra e_1 ed e_2

Si ha quindi una situazione di ***non determinismo****, non essendo specificato quale dei due eventi scatterà prima (e lo scatto di uno impedisce lo scatto dell'altro). Posso ritrovarmi nel caso in cui effettivamente un evento scatta, cambiando lo stato del sistema. In tal caso, in un'ottica completamente deterministica, si deve assumere che* ***l'ambiente*** *abbia fornito un'informazione riguardo il conflitto, ovvero c'è stato qualcosa di esterno che ha permesso ad uno dei due eventi di scattare ugualmente. Ho quindi guadagnato dell'informazione.*

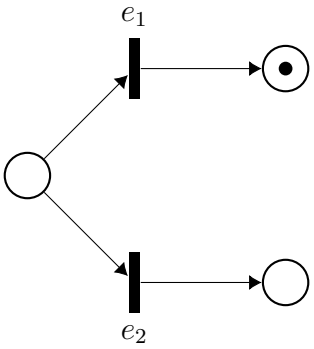


Figura 6.13: Esempio di conflitto con l'intervento dell'ambiente tra e_1 ed e_2 (con conseguente guadagno di informazione)

Ci sono però casi in cui in ogni caso non si può avere informazione su quale esempio sia scattato. Si può ipotizzare che tale informazione fosse presente nello stato precedente del sistema (una sola condizione attiva, per esempio). Quindi l'informazione, finita nell'ambiente (ricevuta dall'ambiente), si è persa.

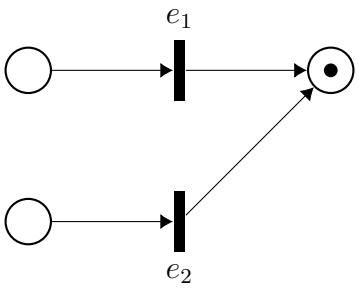


Figura 6.14: Esempio di conflitto tra e_1 ed e_2 con conseguente perdita di informazione (si può ipotizzare, per esempio, che nello stato precedente la precondizione di e_1 fosse attiva mentre quella dio e_2 fosse inattiva)

Il modello, nell'ottica di Petri, non è quindi un ***modello chiuso*** *ma è in grado di comunicare con l'ambiente (in sintonia con le teorie della fisica).*

Confusione

Definizione 52. *La situazione di* ***confusione*** *è una mistura di situazioni di concorrenza e di conflitto. Si hanno 2 tipi di confusione, entrambe ammissibili:*

1. *detta* ***confusione asimmetrica*** *considera il fatto di avere un caso raggiungibile e due eventi abilitati, nella figura e_1 ed e_2, in maniera concorrente in c, che nella figura consiste nel caso $\{b_1, b_2, b_3\}$. I due eventi sono quindi indipendenti. Nella figura lo scatto dei due eventi porterebbe allo stato $c' = \{b_4, b_5\}$. Bisogna analizzare però nel dettaglio il sistema. Se prima occorre e_1 non si ha alcun conflitto mentre se occorre prima e_2 (che porterebbe in $\{b_1, b_3, b_4\}$) si crea un conflitto tra e_1 ed e_3, che viene risolto a favore di e_1 e a sfavore di e_3.* ***Non è possibile stabilire oggettivamente se è stato sciolto un conflitto****. Sono quindi in una situazione di confusione in quanto non so se è stata effettuata o meno una scelta.*

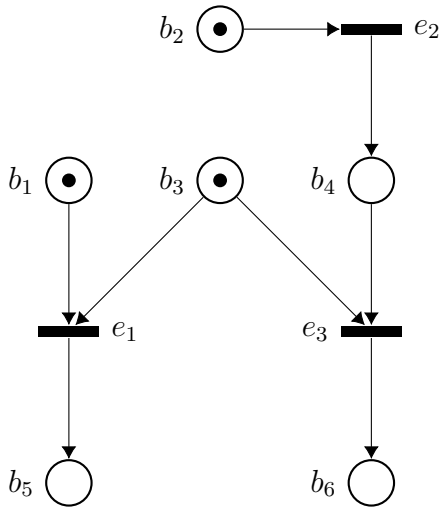


Figura 6.15: Esempio di confusione asimmetrica tra e_1 ed e_2

Una soluzione che vedremo sarà la scomposizione in più componenti del sistema

2. *detta* ***confusione simmetrica****, nome dovuto al fatto che la rete risulta disegnata in modo simmetrico, comportando delle problematiche. Prendiamo nell'immagine il caso raggiungibile* $c = \{b_1, b_2\}$ *e i due eventi* e_1 *ed* e_3*, abilitati in maniera concorrente. Si ha che* $c[\{e_1, e_3\} > c'$*, con* $c' = \{b_3, b_5\}$*. Anche in questo caso si hanno dei conflitti, infatti sia* e_1 *che* e_3 *sono in conflitto con* e_2 *(in quanto se uno dei due viene eseguito* e_2 *non può più occorrere). Anche in questo caso non posso stabilire se il conflitto è stato risolto nel momento in cui arrivo in* c' *(ovvero se si è deciso di fare* e_1 *piuttosto che* e_2 *o* e_3 *piuttosto che* e_2*).*
Anche in questo caso potremo dividere in componenti (una che esegue e_1 *ed* e_2 *e un'altra che esegue* e_1 *ed* e_3*, con* e_2 *che è una sincronizzazione tra le due componenti).*
Non si può dire chi ha deciso e chi ha la responsabilità di decidere quale evento deve occorrere, se alla componente di b_1 *o a quella di* b_2*.*

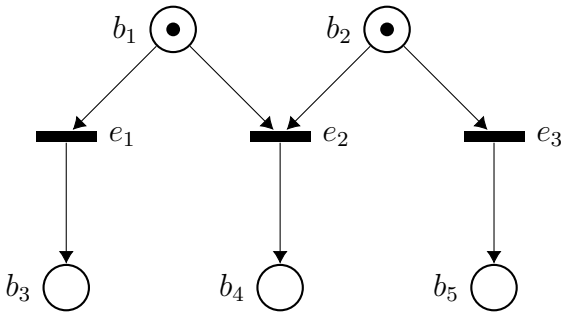
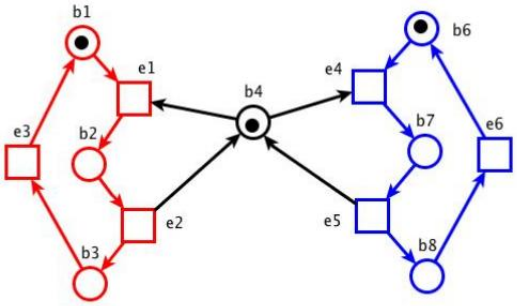


Figura 6.16: Esempio di confusione simmetrica tra e_1 ed e_2

Petri era convinto, nella sua visione deterministica e senza conflitti, che l'avere una situazione di confusione nel modello fosse dovuto al non aver esplicitato alcuni aspetti o di aver costruito male il modello, con informazioni parziali e non complete sul modello e sull'ambiente.
Un altro studioso, Einar Smith, molto vicino a Petri ha invece dimostrato come la confusione sia inevitabile

Vediamo un esempio famoso, detto della **mutua esclusione**, portato da Smith per spiegare come la confusione sia inevitabile nella realtà.

Esempio 50. *Si analizza il seguente sistema:*



In rosso e in Blu abbiamo specificate le due componenti del sistema, che condividono una risorsa, ovvero la condizione b_4, che rappresenta che la risorsa è libera e a disposizione. L'evento e_1 e evento e_4 rappresentano eventi di acquisizione della risorsa (rispettivamente per la prima e per la seconda componente) e quindi le loro precondizioni rappresentano la necessità di acquisirla. Tra questi due eventi c'è una ***situazione di conflitto****. Le condizioni b_2 e b_7, ovvero le rispettive postcondizioni dei due eventi, rappresentano che la risorsa è in uso per la rispettiva componente mentre gli eventi e_2 ed e_5 rappresentano il rilascio della risorsa condivisa, sempre per la rispettiva componente, arrivando rispettivamente nella componente b_3 e b_8. Ovviamente la risorsa non può essere contemporaneamente in uso da entrambe le risorse, quindi b_2 e b_7 non possono essere contemporaneamente marcate, ovvero vere, e per questo si parla di mutua esclusione (se una delle due è vera l'altra deve essere necessariamente falsa). D'altro canto gli eventi e_3 ed e_6 possono invece occorrere in modo concorrente senza conflitti.*
Scrivendo formalmente si ha che, nel caso che la risorsa sia stata acquisita dalla componente rossa e successivamente rilasciata:

$$\{b_3, b_4, b_6\}[\{e_3, e_4\} > \{b_1, b_7\}$$

ma se scatta prima e_3 ho il conflitto tra e_1 ed e_4, se scatta prima e_4 non ho conflitti con e_1. Quindi non ho informazioni sulla risoluzione del conflitto.
Capire se è confusione simmetrica

6.1.6 Sottoreti

Partiamo subito con una definizione formale:

Definizione 53. *Siano $N = (B, E, F)$ e $N_1 = (B_1, E_1, F_1)$ due reti elementari.*
Si dice che N_1 è ***sottorete*** *di N sse:*

- $B_1 \subseteq B$, *quindi l'insieme delle condizioni della rete* N_1 *è sottoinsieme di quello della rete* N
- $E_1 \subseteq E$, *quindi l'insieme degli eventi della rete* N_1 *è sottoinsieme di quello della rete* N
- $F_1 = F \cap [(B_1 \times E_1) \cup (E_1 \times B_1)]$, *ovvero la relazione di flusso di* N_1 *è definita come la restrizione della relazione di flusso di* N *rispetto alle condizioni e* B_1 *e agli eventi* E_1 *(tengo quindi solo gli archi di* N *che connettono eventi e condizioni di* N_1*)*

Definizione 54. *Siano* $N = (B, E, F)$ *e* $N_1 = (B_1, E_1, F_1)$ *due reti elementari.*
Si dice che N_1 *è* ***sottorete generata da*** B_1 *di* N *(ovvero di sottorete generata da un insieme di condizioni) sse:*

- $B_1 \subseteq B$, *quindi l'insieme delle condizioni della rete* N_1 *è sottoinsieme di quello della rete* N
- $E_1 = {}^\bullet B_1 \cup B_1^\bullet$, *ovvero come eventi si hanno tutti quegli eventi che sono collegati in* N *alle condizioni incluse nell'insieme di condizioni* B_1*, prendendo quindi tutti i pre-eventi e i post-eventi delle condizioni dell'insieme* B_1
- $F_1 = F \cap [(B_1 \times E_1) \cup (E_1 \times B_1)]$, *ovvero la relazione di flusso di* N_1 *è definita come la restrizione della relazione di flusso di* N *rispetto alle condizioni* B_1 *e agli eventi* E_1

Non ho quindi una sottorete generata da un insieme arbitrario di condizioni ed eventi ma questi ultimi sono direttamente presi in relazione all'insieme delle condizioni scelto

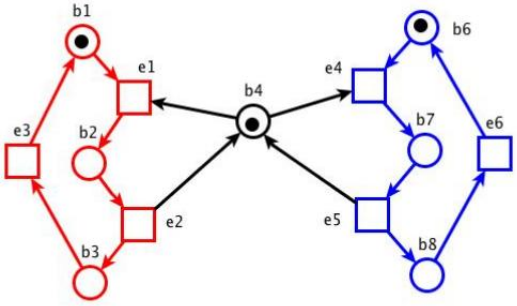
Definizione 55. *Siano* $N = (B, E, F)$ *e* $N_1 = (B_1, E_1, F_1)$ *due reti elementari.*
Si dice che N_1 *è* ***sottorete generata da*** E_1 *di* N *(ovvero di sottorete generata da un insieme di eventi) sse:*

- $B_1 = {}^\bullet E_1 \cup E_1^\bullet$, *ovvero come condizioni si hanno tutte quelle condizioni che sono collegati in* N *agli eventi inclusi nell'insieme di eventi* E_1*, prendendo quindi tutte e precondizioni e le postcondizioni degli eventi dell'insieme* E_1
- $E_1 \subseteq E$, *quindi l'insieme degli eventi della rete* N_1 *è sottoinsieme di quello della rete* N

- $F_1 = F \cap [(B_1 \times E_1) \cup (E_1 \times B_1)]$, *ovvero la relazione di flusso di* N_1 *è definita come la restrizione della relazione di flusso di* N *rispetto alle condizioni* B_1 *e agli eventi* E_1

Non ho quindi una sottorete generata da un insieme arbitrario di condizioni ed eventi ma le prime sono direttamente prese in relazione all'insieme degli eventi scelto

Esempio 51. *Tornando all'esempio della mutua esclusione:*



si ha, per esempio:

- *in rosso si ha la sottorete* $N' = (\{b_1, b_2, b_3\}, \{e_1, e_2, e_3\}, F')$, *che è la sottorete generata dall'insieme di condizioni* $B' = \{b_1, b_2, b_3\}$
- *in blu si ha la sottorete* $N' = (\{b_6, b_7, b_8\}, \{e_4, e_5, e_6\}, F')$, *che è la sottorete generata dall'insieme di condizioni* $B' = \{b_6, b_7, b_8\}$
- *si ha la sottorete* $N' = (\{b_2, b_4, b_7\}, \{e_1, e_2, e_4, e_5\}, F')$, *che è la sottorete generata dall'insieme di condizioni* $B' = \{b_2, b_4, b_7\}$
- *si ha la sottorete* $N' = (\{b_1, b_2, b_3, b_4\}, \{e_1, e_2, e_3\}, F')$, *che è la sottorete generata da dall'insieme di eventi* $E' = \{e_1, e_2, e_3\}$

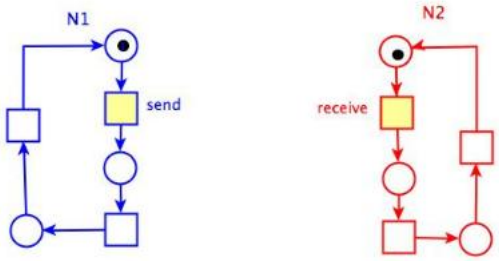
6.1.7 Operazioni di Composizione per Reti di Petri

Data una rete $N = (B, E, F, c_0)$ questa può essere ottenuta componendo altre reti di Petri. Si hanno in letteratura 3 modi principali:

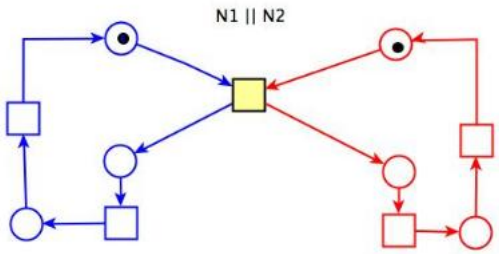
1. la **composizione sincrona**
2. la **composizione asincrona**
3. la **composizione mista, tra sincrona e asincrona**

Iniziamo informalmente a vedere degli esempi pratici.

Esempio 52. *Supponiamo di avere i modelli di due componenti, N_1, con un evento che corrisponde ad un'azione di invio, ed N_2, con un evento che corrisponde ad un'azione di ricezione:*



Supponiamo che invio e ricezione siano eventi corrispondenti all'hand-shacking, ovvero l'invio avviene solo se può avvenire la ricezione.
Vado quindi a sincronizzare questi due eventi, che diventano quindi un'unico evento nella rete composta, che è abilitato se le due precondizioni, nelle due componenti sono entrambe vere:

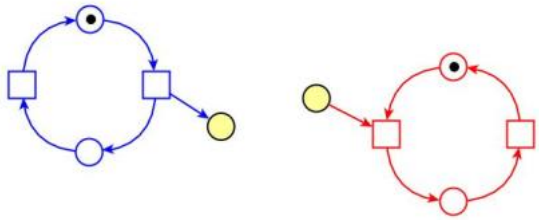


La composizione viene indicata con:

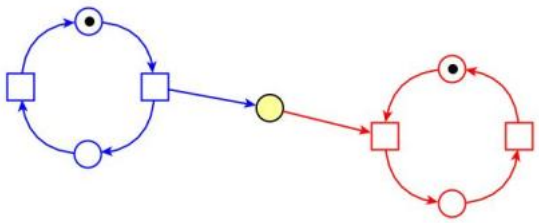
$$N_1||N_2$$

Lo scatto dell'evento, in maniera sincrona, rende vere le postcondizioni nelle due componenti e false le due precondizioni.
Abbiamo appena visto un esempio di ***composizione sincrona***

Esempio 53. *Supponiamo di avere i modelli di due componenti, N_1, che invia in un canale un messaggio (per esempio in un buffer), e N_2, che riceverà il messaggio solo quando esso sarà disponibile:*



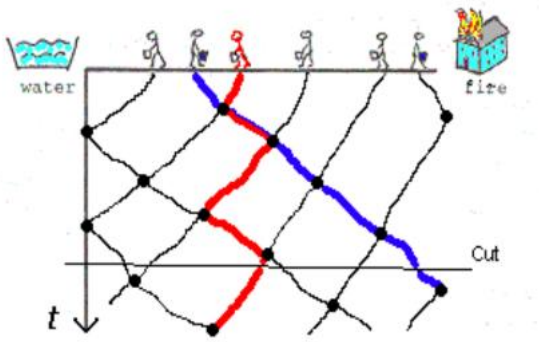
In questo caso, a differenza dell'esempio precedente, non identifichiamo eventi ma condizioni. Identifico quindi il canale (le due condizioni) come uno solo, che avrà il pre-evento in una componente e il post-evento nell'altra. Quindi il pre-evento, nella componente N_1, può scattare solo se questa nuova condizione condivisa, il canale, è libera, indipendentemente dalla componente N_2. D'altro canto l'evento in N_2 può scattare solo se la condizione condivisa è marcata, indipendentemente dallo stato della prima componente, liberando il canale di comunicazione. Avvio e ricezione (dopo che il messaggio è stato inviato può essere letto in un qualsiasi futuro) non sono sincronizzati e si ha quindi a che fare con un esempio di ***composizione asincrona****.*
Si avrebbe:



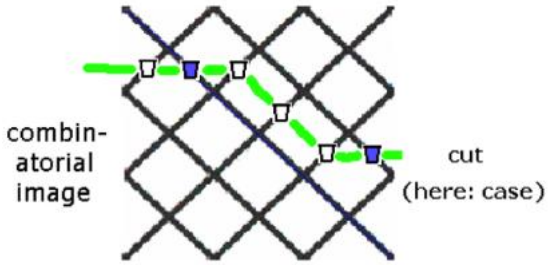
Sarà interessante studiare come la composizione di due componenti, per esempio, senza deadlock non comporta, in generale, l'ottenimento di una rete priva di deadlock.

6.2 Processi non sequenziali

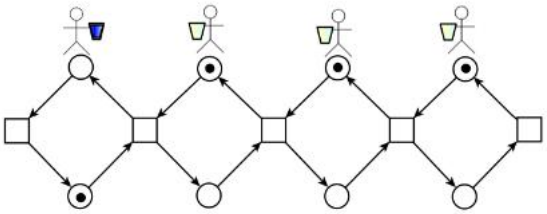
Parliamo ora di sistemi non sequenziali, ad ordini parziali.
Vediamo un esempio sempre usato da Petri. Petri pensò ad un sistema in cui diversi omini, spostandosi avanti nel tempo, portassero dei secchi per andare a spegnere un fuoco (dato che il sentiero è stretto si studia protocollo per cui se hanno il secchio vuoto si muovono a sinistra fino alla fonte d'acqua e a quel punto ci si sposta a destra). Se due omini si incontrano lungo la strada si scambiano il secchio e invertono la direzione di marcia:



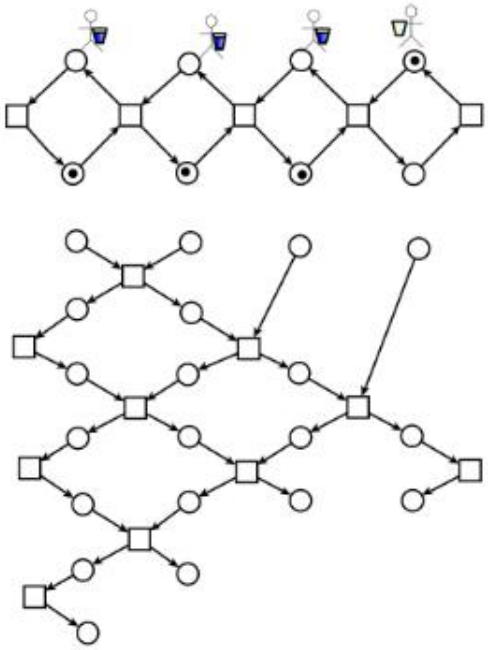
Un modello classico è quello di far scorrere il tempo come nell'immagine vedendo gli spostamenti di ogni omino. Nell'immagine si identifica in rosso la storia di un omino e in blu la storia del secchio pieno. Ma a Petri non va bene questo modello avendo a che fare col il tempo, avendo un modello non discreto.
Cerca quindi un modello combinatorio in cui si hanno varie configurazioni che possono essere osservate (con quindi non necessariamente una sola scala del tempo). Si può osservare una configurazione del tipo:



con relazioni di dipendenza degli eventi (le righe blu) e situazioni di indipendenza (la riga verde). Gli omini potrebbero essere quindi modellati da una rete con gli incontri tra omini che diventano sincronizzazioni. L'evoluzione del sistema può essere registrato salvando informazioni vere. Nel sistema la prima transizione a sinistra è quindi il prelevamento dell'acqua e l'ultimo a destra il versamento sul fuoco:

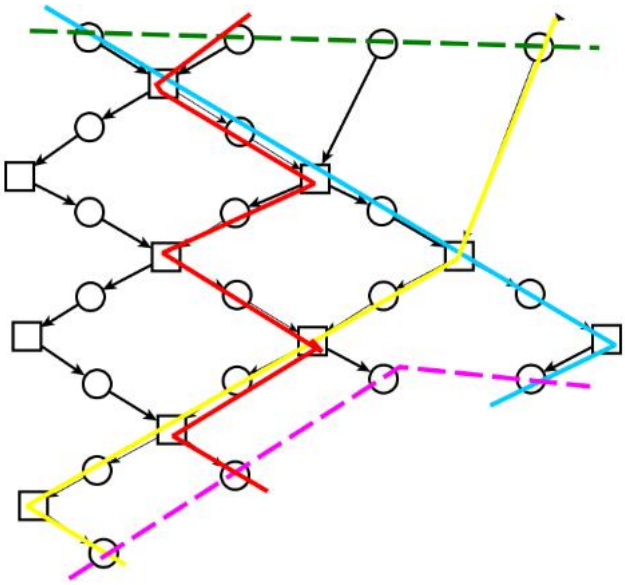


Gli eventi in mezzo sono i cambiamenti di secchio.
l'ordine degli archi segue il secchio, verso sinistra con il secchio vuoto e verso destra con il secchio pieno. Le condizioni in alto praticamente sono il secchio vuoto mentre in basso pieno.
L'evoluzione del sistema posso memorizzarlo tramite condizioni vere. Possiamo quindi produrre la rete in base alle condizioni che possono scattare (in corrispondenza degli elementi della rete), ottenendo:



Tale modello è infinito.

Posso modellare i vari percorsi tramite linee colorate (percorso del secchio pieno in blu, percorso dell'omino in rosso, secchio vuoto in giallo, etc...) con quelle tratteggiate in alto (verde) che rappresentano lo stato iniziale e quelle in basso (viola) quello finale, dove blocco il sistema (dopo il quale si ricomincia da capo):



Tra le varie linee colorate (non quelle tratteggiate) si hanno relazioni di dipendenza.
Definiamo il tutto in modo formale.

Definizione 56. *Definiamo:*

$$N = (B, E, F)$$

come una ***rete causale****, detta anche rete di occorrenze senza conflitti, sse:*

- $\forall b \in B : |{}^\bullet b| \leq 1 \wedge |b^\bullet| \leq 1$*, ovvero non si hanno conflitti, quindi per ogni condizione si ha al più un pre evento e un post evento (avendo quindi al più un arco entrante e al più uno uscente)*
- $\forall a, y \in B \cup E : (x, y) \in F^+ \implies (y, x) \notin F^+$*, ovvero non si hanno cicli, quindi presi due elementi collegati da una sequenza di archi orientati, avendo un cammino tra i due elementi (F^+ è la chiusura transitiva della relazione F) non ho anche un cammino opposto tra i due*

- $\forall e \in E : \{x \in B \cup E | \, xF^*e\}$ *è finito, ovvero si ha un numero finito di pre-elementi di un certo elemento*

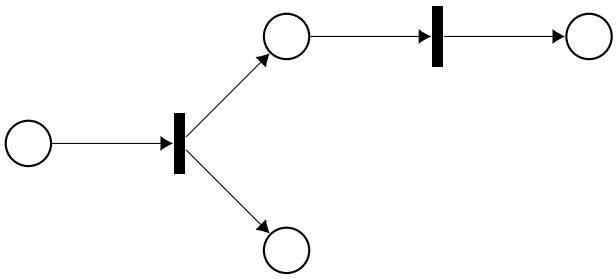
Sono quindi reti che registrano un comportamento e quindi non si hanno conflitti (che in caso sono sciolti registrando solo quello che è effettivamente successo e non quello che potrebbe succedere). Si registra una run del sistema. Non si hanno nemmeno cicli perché ogni ripetizione dell'evento viene concatenata a quella prima (come detto nell'esempio dopo le righe tratteggiate in viola si cominciava da capo).
La rete può essere quindi si infinita ma è composta da un insieme di elementi finito che si ripete. In ogni caso il passato di un evento è finito e registrato, anche se nel complesso il comportamento è infinito "in avanti".
Con una rete causale si possono non distinguere più condizioni ed eventi.

Ad una rete causale è possibile associare un ordine parziale:

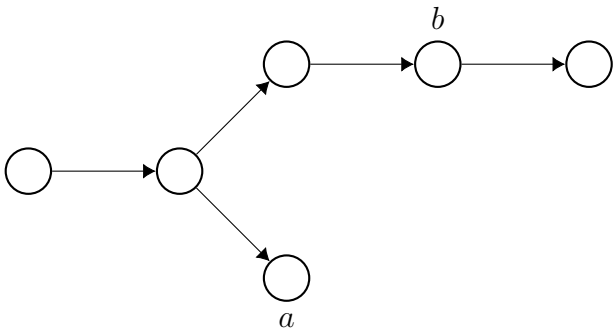
$$(X, \leq) = (B \cup E, F^*)$$

Dicendo che un elemento "è minore" di un altro se esiste un cammino orientato dall'uno all'altro (si specifica che F^* non mi farà mai identificare x con x).

Esempio 54. *Vediamo un esempio di rete causale:*

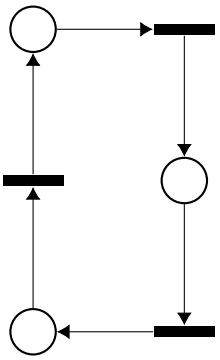


che quindi avendo un ordine parziale ed essendo causale posso scrivere come:

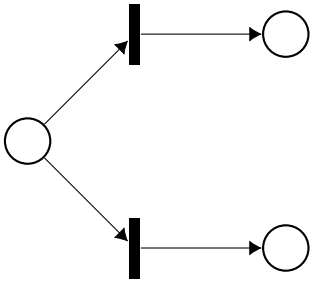


*L'ordine **parziale** si rileva dal fatto che, per esempio, non si ha alcuna relazione d'ordine tra a e b.*

Esempio 55. *Vediamo anche due esempi di reti non causali:*



avendo un ciclo.
Oppure:



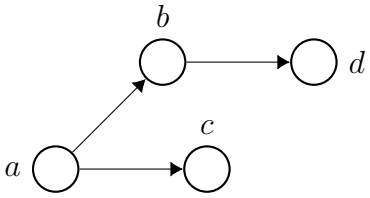
Avendo un conflitto.

Definizione 57. *Data una rete causale $N = (B, E, F)$ e dato un ordine parziale $(X, \leq)$ con $X = B \cup E$ si ha che si può interpretare la relazione d'ordine come indipendenza o dipendenza causale, ovvero presi $x, y \in X$ come elementi che occorrono nella storia di $X = B \cup E$ si hanno le seguenti diciture:*

- *$x \leq y$ (avendo un cammino da x a y) corrisponde a x **causa** y, ovvero si ha una relazione di dipendenza causale tra i due*
- *x **li** y indica che $x \leq y \vee y \leq x$ e quindi corrisponde a x **e** y **sono causalmente dipendenti**. Si ha che **li** può venire letto come linea (x in linea con y) avendo che uno dei due precede l'altro*

- x ***co*** *y indica che $\neg(x < y) \wedge \neg(y < x)$ e quindi corrisponde a x* ***e*** *y* ***sono causalmente indipendenti****, avendo che i due elementi non si precedono a vicenda, non avendo ordine tra loro. Si ha che* ***co*** *sta per concurrency*

Esempio 56. *Presa la rete causale (già compattata):*



Si ha che:

- b ***co*** c
- c ***co*** d
- $\neg(b$ ***co*** $d)$
- c ***li*** a
- a ***li*** d
- $\neg(c$ ***li*** $d)$

Si ha quindi che ***li*** e **co** sono:

- simmetriche
- **non** transitive
- riflessive

che sono comunque proprietà derivate dalla teoria degli ordini parziali.
Possiamo ora considerare sottoinsiemi dell'ordine parziali in cui la relazione **li** e la relazione **co** sono transitive.

Definizione 58. *Data una rete causale $N = (B, E, F)$ e dato un ordine parziale $(X, \leq)$ con $X = B \cup E$ definiamo:*

$$C \subseteq X$$

come:

- ***co-set*** *sse* $\forall x, y \in C : x$ ***co*** y*, quindi* C *è una clique della relazione* ***co***
- ***taglio*** *sse* C *è un co-set massimale (tutti gli elementi nel taglio sono in relazione* ***co****)*

Definiamo C *come co-set massimale sse* $\forall\, y \in X \backslash C$ *si ha che:*

$$\exists c \in C : y \not\textbf{co} c$$

(nel dettaglio esiste almeno un c*).*
Quindi in C *definito o come* ***co-set*** *o come* ***taglio*** *si ha che vale la transitività.*

Definiamo:

$$L \subseteq X$$

come:

- ***li-set*** *sse* $\forall x, y \in L : x$ ***li*** y
- ***linea*** *sse* L *è un li-set massimale*

rivedere la seguente affermazione:
Definiamo L *come li-set massimale sse* $\forall\, y \in X \backslash L$ *si ha che:*

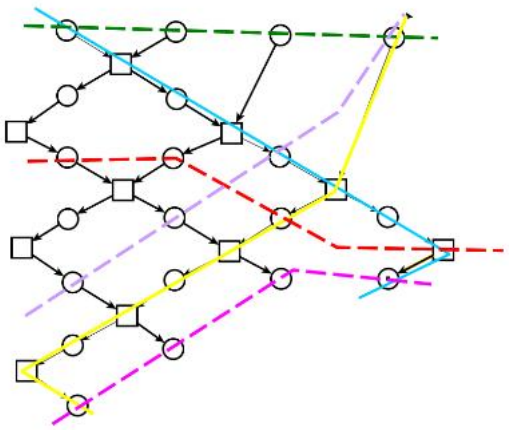
$$\exists l \in L : y \not\textbf{li} l$$

Si ha quindi che:

- *in un* ***co-set*** *la relazione* ***co*** *è transitiva*
- *in un* ***li-set*** *la relazione* ***li*** *è transitiva.*

Tagli e linee possono essere fatti sia di condizioni che di eventi.

Esempio 57. *Tornando all'esempio del secchio:*



si ha che:

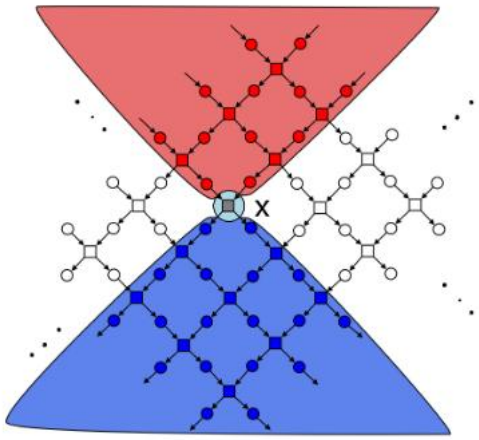
- *la storia del secchio è un insieme di elementi in relazione **li** e, nello specifico, ad esempio la parte azzurra o quella gialla, è una linea che descrive un sottoprocesso*
- *le parti tratteggiate, sono invece un co-set e, nel dettaglio, sono un taglio (infatti se cerco di aggiungere un qualsiasi altro elemento otterrei una relazione di dipendenza con uno degli elementi già presenti)*

I tagli fatti di sole condizioni rappresentano casi raggiungibili (nell'immagine quello verde che quello viola che quello viola-chiaro).
Una rete causale quindi registra il comportamento di un sistema elementare. Si hanno quindi, con i tagli, possibili osservazioni di configurazioni possibili nella storia del sistema.

Definizione 59. *Grazie alle reti causali, preso un elemento $x \in X$, possiamo definire:*

- *$past(x)$, ovvero il passato dell'elemento, tutti gli elementi in relazione $\leq$ di x*
- *$future(x)$, ovvero il futuro dell'elemento, tutti gli elementi in relazione $\geq$ di x*

Visualizzabili, per esempio, nell'immagine, rispettivamente in rosso e blu:

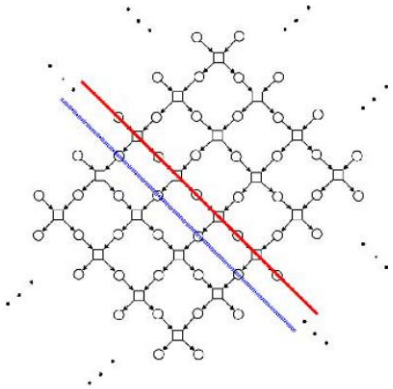


*Gli elementi nell'anti-cono (la parte bianca) sono in relazione **co** con x e quindi possono essere concorrenti.*

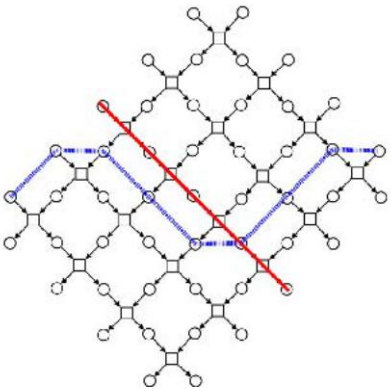
Definizione 60. *Data una rete causale $N = (B, E, F)$ e dato un ordine parziale $(X, \leq)$ con $X = B \cup E$ definiamo che la rete è* ***K-densa*** *se ogni linea ed ogni taglio si intersecano in un punto, tutti hanno un punto comune. Formalmente:*

$$\forall\, h \in Linee(N), \forall\, c \in Tagli(N) : |h \cap c| = 1$$

Con $Linee(N)$ e $Tagli(N)$ che sono rispettivamente li insiemi di tutte le linee e dei tagli.
Se N è finita è anche ***K-densa****, se sono infinite non è detto.*
Una rete non K-densa potrebbe essere (in rosso una linea e in blu un taglio):

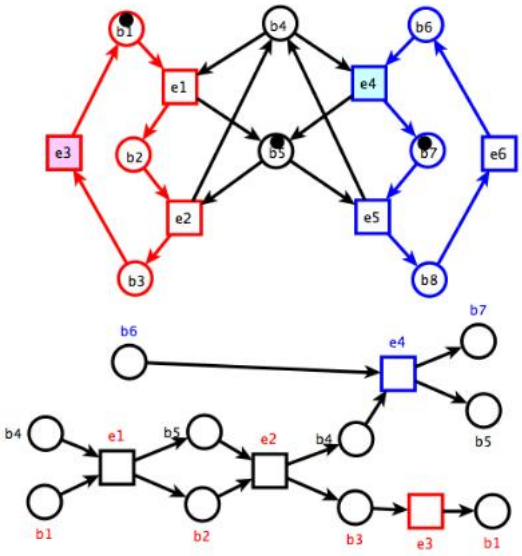


(si nota che è una rete infinita).
e una K-densa:



(si nota che è una rete finita).
(K significa combinatoria).

Esempio 58. *Vediamo anche il solito esempio della mutua esclusione:*



Con sia il sistema elementare che la rete causale associata (dove si vede ad esempio che e_4 ed e_3 sono in relazione ***co****, le prime tre condizioni a sinistra (b_1, b_4, b_6) sono un taglio anche le ultime a destra (b_1, b_5, b_7)).*

Definizione 61. *Sia $\Sigma = (S, T, F, c_{in})$ un sistema elementare senza contatti e finito, tale che $S \cup T$ sia finito. Si ha che, con ϕ che mappa dalla rete causale al sistema elementare:*

$$\langle N = (B, E, F), \phi \rangle$$

è un processo non sequenziale di Σ sse:

- *(B, E, F) è una rete causale (si ammettono condizioni isolate)*
- *$\phi : B \cup E \to S \cup T$ è una mappa tale che:*
  - *$\phi(B) \subseteq S, \phi(E) \subseteq T$*
  - *$\forall x, y \in B \cup E : \phi(x) = \phi(y) \implies (x \leq y) \vee (y \leq x)$ non avendo quindi concorrenza*
  - *$\forall e \in E : \phi({}^\bullet e) = {}^\bullet\phi(e) \wedge \phi(e^\bullet) = \phi(e)^\bullet$ quindi le precondizioni di un evento nella rete causale devono corrispondere alle pre dell'evento nel sistema elementare (e così anche per le post)*

- $\phi(Min(N)) = c_{in}$,
  con $Min(N) = \{x \in B \cup E | \nexists y_{(}y, x) \in F\}$ *che sono gli stati locali iniziali, ovvero se prendo gli elementi minimali della rete causali, che non hanno un arco entrante, essi sono mappati nel caso iniziale del sistema elementare*

Se ho queste proprietà la rete causale è una registrazione del sistema elementare.
In tal caso si ha che $N = (B, E, F)$ *è K-densa (sia che sia finita che infinita), avendo il sistema di partenza finito. Le linee sono quindi sottoprocessi sequenziali e i tagli possibili configurazioni sempre raggiungibili.*
Inoltre si ha che:

$$\forall K \in B \ K \text{ è } B\text{-taglio di } N \text{ t.c. } K \text{ è finito} \wedge \exists c \in C_\Sigma : \phi(K) = c$$

Dato un sistema ho tanti processi non sequenziali che rappresentano esecuzioni del sistema. Ci si chiede quindi se ho un unico oggetto che rappresenta tutti i possibili run del sistema.

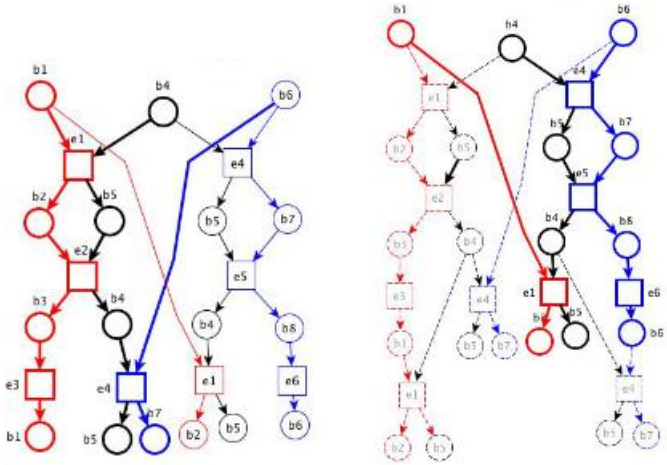


Figura 6.17: Esempio con più processi del modello della mutua esclusione modellati da un'unica rete con conflitti in avanti.

Su slide, quelle dedicate agli esercizi, ulteriori esempi.

Definizione 62. *Data una rete causale* $N = (B, E, F)$, *con* $X = B \cup E$, *diciamo che è una* ***rete di occorrenze*** *sse:*

- $\forall b \in B : |{}^\bullet b| = 1$, *ovvero ho conflitti solo in avanti*

- $\forall\, x, y \in B \cup E : (x, y) \in F^+ \implies (y, x) \notin F^+$, *ovvero non ho cicli, in quanto le ripetizioni sono già registrate*
- $\forall\, e \in E : \{x \in B \cup E | x F^* e\}$, *ovvero il passato di un evento, è finito*
- *la* ***relazione di conflitto*** $\#$ *non è riflessiva, avendo:*

$$\# \subset X \times X$$

definita come:

$$x \# y \iff \exists e_1, e_2 \in E : {}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset \wedge e_1 \leq x \wedge e_2 \leq y$$

ovvero due elementi sono in conflitto sse sono in "alternativa" ovvero se si hanno due eventi che condividono precondizioni (sicuramente non le post non avendo conflitti in avanti) con i due eventi che causano i due elementi x e y, allora anche questi due sono in conflitto. Vediamo degli esempi:

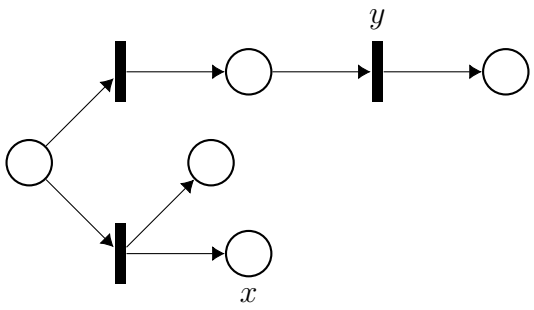


Figura 6.18: Esempio con x e y non in conflitto. Nel dettaglio questa è una rete di occorrenze

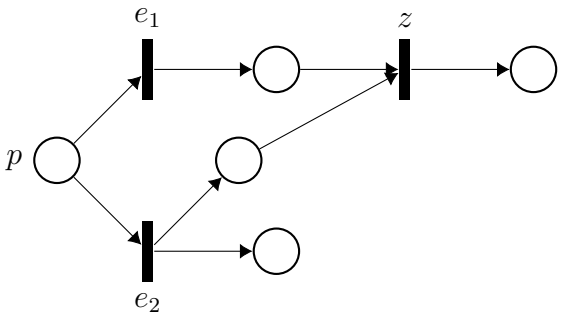


Figura 6.19: Esempio con z che è in conflitto con se stessa (avendo $e_1 < z$ e $e_2 < z$, con i due eventi che condividono p) e quindi non può essere una rete di occorrenze

È comunque possibile associare ad una rete di occorrenze N un ordine parziale.

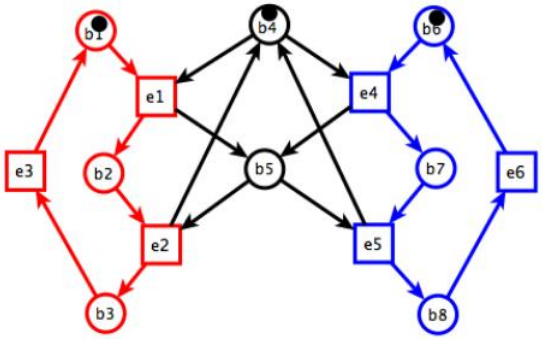
Definizione 63. *Sia $\Sigma = (S, T, F, c_{in})$ un sistema elementare senza contatti e finito. Si ha che:*

$$\langle N = (B, E, F), \phi \rangle$$

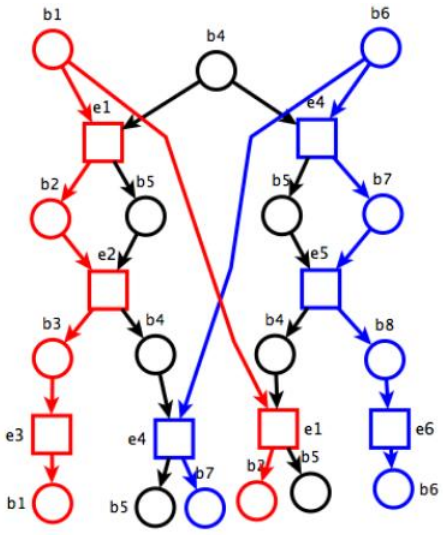
*è un **processo ramificato** di Σ sse:*

- *(B, E, F) è una rete di occorrenze (si ammettono condizioni isolate, volendo poter registrare il caso iniziale come un insieme di condizioni che magari non vengono modificate)*
- *$\phi : B \cup E \to S \cup T$ è una mappa tale che:*
  - *$\phi(B) \subseteq S, \phi(E) \subseteq T$*
  - *$\forall e_1, e_2 \in E : ({}^\bullet e_1 = {}^\bullet e_2 \wedge \phi(e_1) = \phi(e_2)) \implies e_1 = e_2$, ovvero se due eventi condividono le precondizioni e corrispondono allo stesso eventi del sistema secondo la mappa allora necessariamente sono lo stesso evento*
  - *$\forall e \in E : \phi({}^\bullet e) = {}^\bullet\phi(e) \wedge \phi(e^\bullet) = \phi(e)^\bullet$ quindi le precondizioni di un evento nella rete causale devono corrispondere alle pre dell'evento nel sistema elementare (e così anche per le post)*
  - *$\phi(Min(N)) = c_{in}$, quindi il taglio iniziale deve essere il caso iniziale del sistema*

Esempio 59. *Riprendendo il solito esempio di mutua esclusione:*



si ha il seguente processo ramificato:



Cerchiamo ora di definire meglio l'unfolding, quell'oggetto che rappresenta tutti i possibili sottoprocessi non sequenziali di del sistema, ovvero tutti i possibili comportamenti del sistema.
Definiamo prima dei "prerequisiti".

Definizione 64. *Sia* $\Sigma = (S, T, F, c_{in})$ *un sistema elementare senza contatti e finito, e siano:*

$$\Pi_1 = \langle N_1; \phi_1 \rangle$$

$$\Pi_2 = \langle N_2; \phi_2 \rangle$$

due processi ramificati di Σ.
Si ha che Π_1 *è un* ***prefisso*** *di* Π_2 *(e quindi* Π_1 *è un sottoprocesso di* Π_2*) sse:*

- N_1 *è una sottorete di* N_2
- $\phi_{2|N_1} = \phi_1$ *ovvero* ϕ_2 *ristretto a* N_1 *è uguale a* ϕ_1

Si ha che Σ *ammette un unico processo ramificato che è massimale rispetto alla relazione di prefisso tra processi e tale processo è detto* ***unfolding*** *di* Σ*:*

$$Unf(\Sigma)$$

Quindi è il processo ramificato più grande, largo il più possibile e lungo infinito se il sistema ha comportamento ciclico.
Ha senso studiare un prefisso finito dell'unfolding che permette di determinare

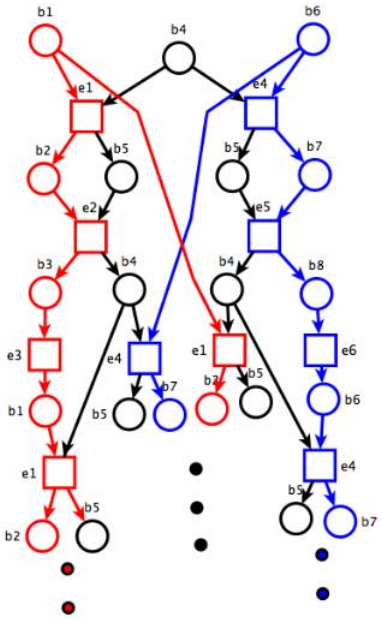
tutto l'unfolding.
Tale processo non sequenziale è un processo ramificato:

$$\Pi = \langle N; \phi \rangle$$

tale che N è una rete causale, senza conflitti. Il processo Π è anche detto ***run*** *di N.*
L'unfolding tiene traccia di tutti i processi sequenziali di un sistema elementare.

Rivedere bene questa parte (lezione 30 novembre, con anche parte di esercizi).

Esempio 60. *Riprendendo il solito esempio di mutua esclusione si ha il seguente unfolding infinito:*



Si è cercato di capire se un prefisso dell'unfolding basti a studiare l'intero sistema.

Definizione 65. *Definiamo* ***prefisso completo*** *un prefisso che include tutti i casi raggiungibili.*

Più si ha concorrenza e più la discrepanza tra unfolding e sistema di transizioni aumenta a favore del prefisso finito dell'unfolding (???). Con concorrenza alta si hanno algoritmi più efficienti di calcolo dell'unfolding (e anche di model checking).

Capitolo 7

Logiche temporali e model-checking

Per capire l'importanza di questo argomento pensiamo di avere un codice concorrente scritto in un certo linguaggio, con l'utilizzo dei *thread*, con un produttore, un consumatore e un buffer. I primi due estendono la classe *thread* mentre il buffer ha i metodi *synchronized* per il prelevamento e il deposito. Questo codice simula il modello produttore-consumatore già approfondito. Bisogna capire se questo programma funzioni correttamente ma non posso usare le tecniche viste precedentemente per valutare la correttezza di programmi (logica di Hoare etc...) in quanto abbiamo a che fare con un programma che non inizia, produce dati e finisce ma, generalmente, si ha a che fare con un programma senza stato finale, che esegue infinitamente le sue parti in modo concorrente. Per il modello appena ipotizzato possiamo dire che è corretto se, ad esempio:

- ogni oggetto prodotto venga prima o poi consumato
- nessun oggetto venga consumato più di una volta
- il sistema non vada in deadlock

Pensiamo ora ad una rete di Petri "complessa" con due processi che condividono in un mutua esclusione una certa risorsa. Si ha quindi una sezione critica dove un processo usa tale risorsa. Possiamo volere, per esempio, che i due processi non siano mai contemporaneamente nella sezione critica (userebbero insieme la risorsa) e che prima o poi un processo entri in zona critica. Un metodo è controllare l'intero spazio delle marcature raggiungibili e verificare che tutto vada come voluto, per la prima richiesta, ma ci sono tecniche migliori.

Un altro caso d'uso è quello dei circuiti asincroni.
Volendo si può adattare la logica di Hoare a modelli concorrenti ma solo se si ha comunque un output finale e non un'evoluzione ciclica infinita.

7.1 Logica PLTL/LTL

Fino'ora ci si è basati sulla logica proposizionale ma tale logica ha dei limiti, si introduce quindi la **logica PLTL (*Propositional Linear Time Logic*)** che introduce il concetto di **tempo**, in ottica **lineare**, nel processo logico. Tale logica è anche abbreviata in **LTL (*Linear Time Logic*)**.
La logica PLTL viene usata per **model checking** e lo scopo generale che ci si propone è quello di presentare un approccio formale alla progettazione e all'implementazione di sistemi basato su un linguaggio formale che permetta di specificarli e ragionare sulle loro proprietà. Si vogliono studiare sia sistemi grandi, come software o addirittura sistemi operativi, che piccoli, come per esempio una coppia di semafori. Tutti questi sistemi hanno però la medesima caratteristica, ovvero assumono in ogni **istante di tempo** un determinato **stato** definito dalle **variabili** del sistema stesso. Si hanno quindi nel sistema, in ogni istante di tempo:

- uno **stato**, univocamente definito dai valori delle variabili
- una **transizione** che segnala un passaggio da uno stato ad un altro
- una **computazione** che rappresenta una sequenza di stati di cui ogni coppia forma una transizione. Si introduce quindi il concetto **temporale**

Il **tempo** viene pensato come un oggetto discreto, cadenzato dalle transizioni, su cui può essere definita anche una relazione d'ordine (potrebbe servire che un processo termini prima, dopo o nello stesso tempo di un altro).

Definizione 66. *Si definisce **sistema reattivo** una componente:*

- *non terminante e interattivo*
- *che può leggere il proprio input non solo all'inizio della computazione e che può produrre output non solo alla fine della sua computazione (si possono quindi avere multipli input e multipli output)*
- *che interagisce con altri componenti distribuiti o concorrenti*

Sono quindi:

- *sistemi concorrenti, distribuiti e asincroni*
- *Non obbediscono al paradigma input-computazione-output*
- *non si possono analizzare con gli strumenti della logica di Hoare*

avendo come esempi di comportamento:

- *"se un messaggio è stato spedito, prima o poi sarà consegnato al destinatario"*
- *"La spia d'allarme resta accesa fino a quando il dispositivo viene spento"*
- *"a partire da qualsiasi stato è possibile riportare il sistema allo stato iniziale"*

Bisogna stabilire la correttezza di un sistema reattivo esprimendo il criterio di correttezza come formula di un opportuno linguaggio logico. Il sistema reattivo si modella tramite un sistema di transizioni, tramite i **modelli di Kripke**, e valutiamo se la formula è vera nel sistema di transizioni, tramite logiche temporali e algoritmi specifici.
Si hanno sistemi non terminanti che usano un numero **finito** di variabili e di conseguenza si ha un numero di stati, in cui transita il sistema, **finito**. Si hanno infatti sistemi di transizione finiti che *"comprimono"* sistemi non terminanti, con computazioni di lunghezza infinita, in una rappresentazione finita. Si ricorda che si possono inoltre categorizzare le proprietà che vogliamo specificare come:

- **proprietà di safety**
- **proprietà di liveness**
- **proprietà di fairness**

Introduciamo quindi meglio i **modelli di Kripke**.

Definizione 67. *Definiamo un* ***sistema di transizioni*** *A come:*

$$A = (Q, T)$$

con:

- *Q come insieme di stati (potrebbe essere infinito ma studieremo i casi finiti)*
- *$T \subseteq Q \times Q$ insieme delle transizioni di stato che portano da uno stato ad un altro*

Definizione 68. *Definiamo **cammino** come:*

$$\pi = q_0 q_1 q_2 \ldots, \;\; q_i \in T, \;\; \forall\, i$$

Un cammino può essere infinito.
*Definiamo **cammino massimale** un cammino che non può più essere esteso (generalmente studieremo casi dove i cammini massimali sono infiniti).*

Definizione 69. *Definiamo un **modello di Kripke** A come un sistema di transazioni a cui ad ogni stato è associato, dato l'insieme delle proposizioni atomiche $AP = \{z_1, z_2, \ldots z_n\}$, un insieme di proposizioni atomiche che sono vere in quello stato:*

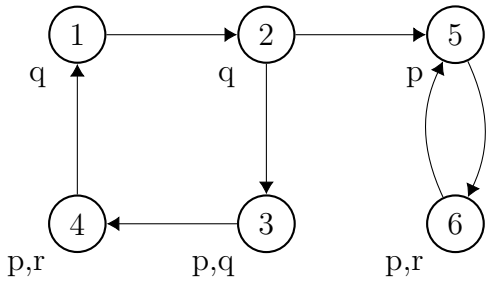
$$A = (Q, T, I)$$

*avendo una **funzione di interpretazione** I, che mi restituisce l'insieme delle proposizioni atomiche vere in uno stato, tale che:*

$$I : Q \to 2^{AP}$$

Potrei avere stati dove nessuna proposizione atomica è vera avendo il sottoinsieme associato vuoto.
Non si specifica di base uno stato iniziale ma si può arricchire la definizione con uno stato iniziale.

Esempio 61. *Vediamo un esempio:*



Dove:

- *$AP = \{p, q, r\}$, insieme delle proposizioni atomiche*

- $Q = \{1, 2, 3, 4, 5, 6\}$, *insieme degli stati (fingiamo che* 1 *sia lo stato iniziale)*
- $T = \{(1, 2), (2, 3), (2, 5), (5, 6), (6, 5), \cdots\}$, *insieme delle transizioni*
- $I(4) = \{p, r\}$, $I(2) = \{q\}$ *etc...*, *gli altri sono indicati in basso a sinistra di ogni stato*
- $1, 2, 5$ *è un cammino non massimale*
- $1, 2, 5, 6, \overline{5, 6}$ *è un cammino massimale (qui indico con la linea sopra gli stati in cui poi si continua a ciclare)*
- $1, 2, 3, 4, \overline{1, 2, 3, 4}$ *è un cammino massimale*
- $1, 2, 3, 4, 1, 2, 5, 6\overline{5, 6}$ *è un cammino massimale*

Definizione 70. *Definiamo le famiglie di cammini massimali, indicandole in una notazione tipo regex, (* per indicare una o più ripetizione e $^\omega$ per indicare un ciclo infinito), come le "tipologie" di cammino che si possono avere.*
(Il prof non ha davvero dato una definizione)

Esempio 62. *Nell'esempio sopra si hanno le seguenti famiglie di cammini massimali:*

- $(1234)^\omega$
- $(1234)^*(12)(56)^\omega$

Si può quindi già intuire che:

le logiche temporali sono frammenti della logica del primo ordine

Infatti con le logiche temporali, come la *logica PLTL*, si supera il limite di rappresentazione temporale della logica proposizionale, permettendo, per esempio, di rappresentare proposizioni del tipo *"prima o poi", "accade sempre che" etc...*.
Si ha un **tempo lineare e discreto**, che si può pensare di rappresentare nella logica classica proposizionale (per esempio come indice) ma questo comporta il doversi dotare di infinite variabili proposizionali (corrispondenti ad ogni *"step"* temporale) e comporta l'ottenimento di una formula proposizionale di lunghezza infinita. Si può pensare di passare allo studio di questi casi con la *logica predicativa* ma sarebbe ben più del necessario, in quanto

le procedure sarebbero di complessità molto elevata, a causa della grande espressività della logica predicativa. Inoltre la logica dei predicati è indecidibile.
Si cerca quindi la via di mezzo cercando logiche specializzate, meno espressive della logica predicativa ma decidibili rispetto ai problemi presi in considerazione. Si cerca una logica con procedure efficienti rispetto all'ampiezza della descrizione del problema in analisi (ovvero tipicamente l'ampiezza del sistema di transizioni), che è direttamente proporzionale al numero di stati del sistema (lineare rispetto al numero degli stati), e rispetto alla lunghezza della formula che esprime la proprietà da testare.

7.1.1 Sintassi

Vediamo innanzitutto la **sintassi** della logica PLTL che chiamiamo anche LTL.
Si ha a che fare con un *vocabolario* più ampio rispetto a quelli della logica proposizionale, vengono infatti aggiunti dei **connettivi** utili alla rappresentazione temporale richiesta.

Definizione 71. *Nella logica PLTL, oltre ai connettivi logici classici della logica proporzionale, si definiscono i seguenti connettivi:*

- ***X**, detto anche **next** o **tomorrow**. È un connettivo unario e viene rappresentato con* $\circ$
- ***F**, detto anche **sometime** o **future**. È un connettivo unario e viene rappresentato con* $\diamond$
- ***G**, detto anche **globally** o **always**. È un connettivo unario e viene rappresentato con* $\Box$
- ***U**, detto anche **until**. È un connettivo binario*

*Viene inoltre indicato con V l'**insieme delle variabili proposizionali**, l'**insieme delle proposizioni atomiche**, variabili che hanno lo stesso significato della logica proposizionale, quindi ogni variabile corrisponde ad una singola proposizione del linguaggio naturale.*

Qui chiamo V quello che prima chiamavo AP per non dover riscrivere tutti gli appunti tratti da metodi formali.

Definizione 72. *Diamo ora una definizione, formale, procedendo per induzione di **formula ben formata** del linguaggio PLTL (definendo così l'aspetto sintattico della logica PLTL):*

- *$\forall p \in V$ vale che p è una formula del linguaggio PLTL, ovvero le variabili proposizionali della logica classica sono formule del linguaggio PLTL. Questa è una* formula atomica
- *i simboli $\top$ (detto anche* true *che semanticamente semplifica una tautologia) e $\bot$ (detto anche false o* bottom *che semanticamente specifica una formula sempre falsa, ovvero una contraddizione) sono formule del linguaggio PLTL. Anche questa è una* formula atomica
- *se A è una formula del linguaggio PLTL allora lo sono anche:*
  - $\neg A$
  - $\mathbf{X}A$
  - $\mathbf{F}A$
  - $\mathbf{G}A$
- *se A e B sono formule del linguaggio PLTL allora lo sono anche:*
  - $A \rightarrow B$
  - $A \wedge B$
  - $A \vee B$
  - $A\mathbf{U}B$ *(anche* $\mathbf{U}(A, B)$*)*
- *nient'altro appartiene all'insieme delle formule del linguaggio PLTL*

Quindi si nota come ***l'insieme delle formule PLTL contiene quello delle formule classiche della logica proporzionale****. Si ha quindi che l'insieme delle formule PLTL non è altro che un'estensione di quello delle formule classiche della logica proposizionale.*

Si hanno varianti della logica temporale con anche operatori relativi al passato ma non sono utili in merito al model checking.

Definizione 73. *La sintassi delle formule del linguaggio PLTL possono essere definite usando anche la notazione* ***BNF (Backus-Naur Form o Backus Normal Form)****, ovvero, $\forall p \in V$:*

$$A ::= p \mid \top \mid \bot \mid (\neg A) \mid (A \wedge A) \mid (A \vee A) \mid (A \rightarrow A) \mid (\mathbf{X}A) \mid (\mathbf{F}A) \mid (\mathbf{G}A) \mid (A\mathbf{U}A)$$

Non si è usata quindi una definizione ricorsiva ma viene invece usata una ***grammatica***

La BNF (Backus-Naur Form o Backus Normal Form) è una metasintassi, ovvero un formalismo attraverso cui è possibile descrivere la sintassi di linguaggi formali (il prefisso meta ha proprio a che vedere con la natura circolare di questa definizione). Si tratta di uno strumento molto usato per descrivere in modo preciso e non ambiguo la sintassi dei linguaggi di programmazione, dei protocolli di rete e così via, benché non manchino in letteratura esempi di sue applicazioni a contesti anche non informatici e addirittura non tecnologici. La BNF viene usata nella maggior parte dei testi sulla teoria dei linguaggi di programmazione e in molti testi introduttivi su specifici linguaggi.
In termini formali, la BNF può essere vista come un formalismo per descrivere grammatiche libere dal contesto.
Una specifica BNF è un insieme di regole di derivazione ciascuna espressa nella forma:

*<simbolo> ::= _espressione_*

Il cammino nel modello di Kripke corrisponde ad una esecuzione di un sistema, che è una sequenza di stati.

7.1.2 Semantica

Definizione 74. *La **semantica**, ovvero il significato dei connettivi della logica PLTL, è data usando i cosiddetti **modelli lineari** (si ricorda l'uso di un tempo lineare).*
Si consideri una struttura algebrica di questo tipo:

$$M = \langle S, \rho, \rightarrow, \Vdash \rangle$$

dove:

- S *è un **insieme infinito di stati**, detti anche **mondi***
- $\rho \in S$ *è uno stato del modello detto anche **root** o **radice**. In tale stato viene codificato il **tempo zero***
- $\rightarrow$ *è una relazione binaria su S detta **relazione di transizione** la quale introduce un **ordinamento lineare** sugli elementi di S. Si ha quindi che:*

  $$\rightarrow \subseteq S \times S$$

  *e, $\forall \alpha \in S$, **esiste ed è unico** $\beta \in S$ tale che vale:*

  $$\alpha \rightarrow \beta$$

Preso quindi uno stato qualsiasi ho un solo modo per passare ad un altro stato (esiste quindi un "prima" e un "dopo").
Inoltre vale che, $\forall \alpha \in S,\ \alpha \not\rightarrow \rho$*, dove* $\rho \in S$ *è l'unico elemento di* S *a godere di questa proprietà (in quanto* ρ *codifica il tempo zero).*

- $\Vdash$ *è una relazione binaria, inclusa in* $S \times V$*, detta* ***relazione di soddisfacibilità****. Solitamente con* $\alpha \Vdash p$ *si indica che* $(\alpha, p) \in \Vdash$ *è valido, ovvero che* α ***soddisfa*** p*, con* $\alpha \in S$ *e* $p \in V$

La struttura M *viene detta* ***modello per PLTL****.*
Si nota come questi modelli lineari seguano un ordinamento simile a quello che si ha tra i numeri in $\mathbb{N}$*.*
Si nota come la semantica della logica PLTL sia drasticamente più complessa di quella della logica classica proposizionale

Bisogna dare ora significato ai connettivi PLTL. Per i connettivi della logica proposizionale si usano tavole di verità e induzione ma per la logica PLTL le cose sono un po' diverse.
Per dare un significato alle formule del linguaggio PLTL bisogna basarsi sui modelli per PLTL.

Definizione 75. *Vediamo quindi la semantica dei vari operatori.*
Diciamo anche che un a formula è vera rispetto ad uno stato q *del modello di Kripke se è vera in tutti i cammini massimali* π *che partono da* q*, dicendo che la formula* α *è soddisfatta in* π*.*
Per comodità indichiamo un cammino massimale generico con:

$$\pi = q_0 q_1 \dots q_n$$

e un suffisso di ordine i*:*

$$\pi^{(i)} = q_i q_{i+1} \dots q_j,\ \textit{avendo}\ \pi = \pi^{(i)}$$

La formula è vera rispetto ad uno stato q *del modello di Kripke se è vera in tutti i cammini a partire da* q*.*
Si indica con:

$$\pi \vDash \alpha$$

per indicare che α *è vera/valida in* π *(o che* π *soddisfa* α*).*

Se pensiamo ai modelli di Kripke si ha, dati α, β *due formule e* p *una proposizione atomica:*

1. $\pi \vDash p$ *sse* $p \in I(q_0)$, *con* q_0 *stato iniziale scelto*
2. $\pi \vDash \neg\alpha$ *sse* $\pi \nvDash \alpha$
3. $\pi \vDash \alpha \vee \beta$ *sse* $\pi \vDash \alpha$ *o* $\pi \vDash \beta$

Potenzialmente bastano questi due connettivi per definire tutti gli altri per la logica proposizionale.
Passando agli operatori temporali, con $\alpha \in FBF$ *(formule ben formate) si ha:*

1. $\pi \vDash \mathbf{X}\,\alpha$ *sse* $\pi^{(1)} \vDash \alpha$ *(quindi se nello stato successivo vale* α*)*
2. $\pi \vDash \mathbf{F}\,\alpha$ *sse* $\exists i \in \mathbb{N} : \pi^{(i)} \vDash \alpha$ *quindi prima o poi* α *sarà vera (quindi può anche essere vera nello stato iniziale con* $i = 0$*)*
3. $\pi \vDash \mathbf{G}\,\alpha$ *sse* $\forall\, i \in \mathbb{N} : \pi^{(i)} \vDash \alpha$*, quindi* α *deve essere vero in tutti i suffissi di* π *(e quindi anche nello stato iniziale)*
4. $\pi \vDash \alpha\,\mathbf{U}\,\beta$ *sse* $\pi^{(i)} \vDash \beta$ *(che sarebbe* $\pi \vDash \mathbf{F}\,\beta$*)* ***e*** $\forall\, h, 0 \leq h < i$, $\pi^{(h)} \vDash \alpha$ *quindi esiste uno stato* i *dove vale* β *e in tutti gli stati precedenti (escluso* i*) vale* α*. Se* β *è vera in* π *posso dire che l'until è valido, trascurando la parte dopo l'and della formula (modificando la definizione possiamo dire che la seconda parte vale solo "se esistono stati precedenti").*

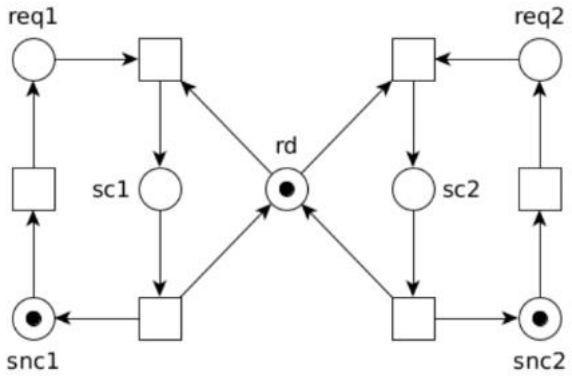
Si nota che la definizione di **modello PLTL** non esclude che due stati diversi possano soddisfare le stesse variabili, ovvero che si possano comportare nello stesso modo *(come se nulla cambiasse nel tempo).*

Esempio 63. *Vediamo degli esempi (considerando cammini infiniti):*

- $\mathbf{FG}\,\alpha$ *indica che* α *è invariante da un certo punto in poi*
- $\mathbf{GF}\,\alpha$ *indica che* α *è vera in un numero infinito di stati (ma non necessariamente tutti)*
- $\mathbf{G} \neq (cs_1 \wedge cs_2)$ *è la mutua esclusione, avendo* cs_1 *e* cs_2 *per indicare che il processo 1 e 2 sono rispettivamente in sezione critica*
- $\mathbf{G}(req \implies \mathbf{XF}\,ack)$*, avendo* req *richiesta pendente di un server e* ack *aknowledge, per dire che se si ha una richiesta pendente prima o poi si avrà il messaggio di ack subito dopo la req (non avviene in contemporaneamente ma nello stato dopo)*

- $\mathbf{G}(req \implies (req\,\mathbf{U}\,ack))$ *per dire che è sempre vero che se si ha una richiesta da un certo punto in poi si dovrà avere un ack, con la req che rimane vera fino al momento prima dell'ack*
- $\mathbf{G}(req \implies ((req \wedge \neg ack)\,\mathbf{U}\,(ack \wedge \neg req)))$ *per dire che se ho una richiesta pendente posso avere che resti tale fino all'ack ma quando arriva l'ack la req deve essere "negativizzata" (non serve il next perché tanto non possono valere nello stesso stato per come è stata scritta la formula)*

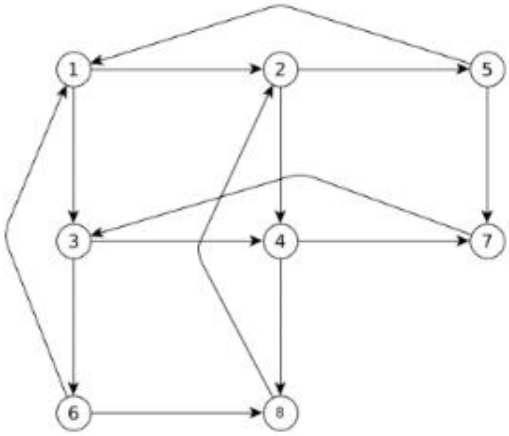
Esempio 64. *Presa la rete:*



con al mutua esclusione, avendo:

- *rd la risorsa disponibile*
- *sc*1 *e sc*2 *le due sezioni critiche*
- *snc*1 *e snc*2 *le due sezioni non critiche*
- *req*1 *e req*2 *le due richieste pendenti di usare la risorsa*

avendo il grafo dei casi (ipotizzando per semplicità 1 *come stato iniziale):*



con:

- $1 = \{rd, snc1, snc2\}$
- $2 = \{rd, req1, snc2\}$
- $3 = \{rd, snc1, req2\}$
- $4 = \{rd, req1, req2\}$
- $5 = \{sc1, snc2\}$
- $6 = \{snc1, sc2\}$
- $7 = \{sc1, req2\}$
- $8 = \{req1, sc2\}$

che può essere letto come modello di Kripke, con l'elenco sopra che può essere visto come la funzione di interpretazione dello stesso.
Studiamo $\mathbf{G} \neq (cs_1 \wedge cs_2)$ *e* $\mathbf{G}\,(req1 \implies \mathbf{F}\,sc1)$ *(è sempre vero che se vale* $req1$ *allora prima o poi la richiesta verrà soddisfatta).*
Per la prima formula possiamo dire che è una ***proprietà di safety****, esprimendo che non si può raggiungere uno stato "pericoloso", come l'uso contemporaneo della risorsa. La verifico anche solo scorrendo le interpretazioni degli stati e vedendo che non esiste una stato in cui compaiono sia* $sc1$ *che* $sc2$ *(questo perché ho il globally e basta quindi vale semplicemente per tutti gli stati).*
Non posso fare lo stesso per la seconda. Ho sempre una globally con quindi una proprietà invariante ma ho all'interno un altro operatore temporale

quindi non posso semplicemente scorrere la lista delle interpretazioni degli stati. In ogni caso questa seconda formula non è valida in quando potrei avere $(1)(2,4,8)^\omega$ *in cui anche il secondo processo fa richiesta e ci entra. Posso entrare in un loop in cui solamente il processo due prende ogni volta la risorsa quindi la formula non è valida in quel cammino massimale.*
Possiamo comunque trovare cammini in cui vale, come $(1,3,6)^\omega$*, in quanto in quei tre stati* $req1$ *è sempre falsa.*

Introduciamo nuovi operatori temporali:

- **weak until**:
$$\alpha \,\mathbf{W}\, \beta \equiv \mathbf{G}\,\alpha \vee (\alpha \,\mathbf{U}\, \beta)$$
quindi vale se α è sempre vera o se vale l'until classico
- **release**:
$$\alpha \,\mathbf{R}\, \beta \text{ sse } \forall\, k \geq 0 : (\pi^{(k)} \vDash \beta \vee \exists\, h < k : \pi^{(h)} \vDash \alpha)$$
ovvero:
$$\alpha \,\mathbf{R}\, \beta \equiv \beta \,\mathbf{W}\, (\alpha \wedge \beta)$$
ovvero vale se β è sempre vero o ad un certo punto β è vero è diventa vero anche α e dall'istante successivo β può diventare falso. Praticamente α libera β, che negli istanti successivi può essere falsa

Per tradurre un enunciato in linguaggio naturale in logica temporale per prima cosa, occorre individuare le proposizioni atomiche; in seguito si analizza la struttura della frase, eventualmente riformulandola in una frase equivalente con espressioni corrispondentiagli operatori temporali.

Definizione 76. *Definiamo l'equivalenza tra due formule come:*

$$\alpha \equiv \beta \iff \forall\, \pi : (\pi \vDash \alpha \leftrightarrow \pi \vDash \beta)$$

per ogni cammino π*. Quindi varia il modo per esprimere la stessa cosa, per ogni cammino* α *è soddisfatta nel cammino sse lo è anche* β *(e avendo il sse vale il viceversa).*

Esempio 65. *Vediamo esempi di formule equivalenti:*

- $\mathbf{F}\,\alpha \equiv \alpha \vee \mathbf{XF}\,\alpha$
infatti prima o poi diventerà vero α *equivale a dire che vale nello stato corrente e nello stato prossimo vale che prima o poi sarà vero* α

- $\mathbf{G}\,\alpha \equiv \alpha \wedge \mathbf{XG}\,\alpha$
  infatti dire che vale sempre α è come dire che ora vale α e che nel prossimo stato vale il globally di α

- $\alpha\mathbf{U}\,\beta \equiv \beta \vee (\alpha \wedge \mathbf{X}\,(\alpha\,\mathbf{U}\,\beta)$
  ovvero l'until è equivalente a dire che ora vale β o che ora vale α ma nello stato successivo varrà l'until

Si evidenzia in tutti e tre i casi la natura ricorsiva delle formule della logica temporale.
Vediamo infatti altri esempi:

- $\mathbf{FGF}\,\alpha \equiv \mathbf{GF}\,\alpha$ *quindi aggiungere il primo future in* $\boldsymbol{FGF}$ *è inutile*
- $\mathbf{GFG}\,\alpha \equiv \mathbf{FG}\,\alpha$ *quindi aggiungere il primo globally in* $\boldsymbol{GFG}$ *è inutile*

Prendiamo la formula:

$$\top\,\mathbf{U}\,\alpha$$

ma per definizione ci basta dire che α prima o poi diventi vera e quindi basta:

$$\top\,\mathbf{U}\,\alpha \equiv F\alpha$$

Il future può sempre essere rappresentato dall'until.
Vediamo un altra formula:

$$\neg\mathbf{F}\,(\neg\alpha)$$

ma questo equivale a dire:

$$\neg\mathbf{F}\,(\neg\alpha) = \mathbf{G}\,\alpha$$

Quindi il globally può essere derivato dal future che è derivato dall'until e quindi **anche il globally potrebbe essere rimosso**.
Si può dimostrare che il **next non è derivabile** e che **l'until, a sua volta, non è derivabile**.

Definizione 77. *Definiamo un* ***insieme minale di operatori****. L'insieme* $\{\mathbf{X}, \mathbf{U}\}$ *forma un insieme minimale di operatori, dal quale possiamo derivare tutti gli altri (****F,G,W,R****).*
Potenzialmente potrei quindi usare solo until e next.
Si hanno anche altri insiemi minimali di operatori volendo (ma in tutti ci sarà l'until essendo l'unico operatore binario, non potendo derivarlo da uno unario).

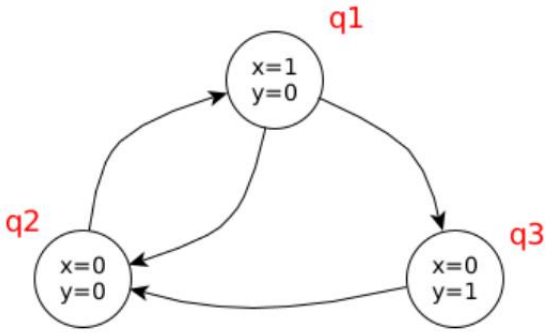
Studiamo ora meglio la negazione di connettivi logici.
Pensiamo a "non è vero che $\mathbf{F}\,\alpha$". Possiamo riformularla con "Non è vero che in ogni cammino,prima o poi α diventa vera", cioè "esiste un cammino nel quale α è sempre falsa". D'altro canto $\neg\mathbf{F}\,\alpha$ significa "in ogni cammino non è vero che prima o poi α diventa vera", che equivale a $\mathbf{G}\,(\neg\alpha)$, quindi in realtà $\neg\mathbf{F}\,\alpha$ non è la negazione logica di $\mathbf{F}\,\alpha$ perché nasconde il concetto di "per ogni cammino", con quindi il quantificatore $\forall$. Dalla logica predicativa sappiamo che la negazione di una formula dove compare un quantificatore equivale a cambiare il quantificatore e aggiungere la negazione e quindi la riformulazione giusta sarebbe "esiste un cammino in cui vale $\mathbf{F}\,\alpha$", che non può essere rappresentato in logica temporale LTL (**CONTROLLARE**). Si ha quindi un limite espressivo, non può esprimere proprietà del tipo:

esiste un cammino in cui α

Esempio 66. *Vediamo esempi di proprietà di questo tipo:*

- *"è sempre possibile ritornare allo stato iniziale" ("è possibile" si traduce in "esiste un cammino")*
- *"e il processo P chiede la risorsa, è possibile raggiungere uno stato di deadlock"*

Esempio 67. *Si prenda:*



che è un modello di Kripke che rappresenta le evoluzioni di un programma con due variabili x e y. I vari q_i rappresentano i possibili stati.
Studiamo:

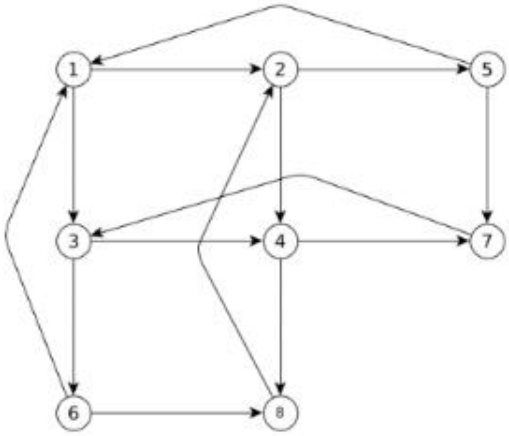
- $\mathbf{G}\,(x = 0 \vee y = 0)$
- $\mathbf{GF}\,(y = 0)$

- $\mathbf{GF}\,(y = 1)$
- $\mathbf{G}\,(x = 1 \implies \mathbf{F}\,(y = 1))$

per farlo bisogna studiare ogni possibile evoluzione a partire da ogni stato (che di volta in volta diventa uno stato iniziale).

- *per il primo basta osservare i vari stati e osservare che vale l'argomento del globally in ogni stato, quindi qualsiasi sia lo stato iniziale varrà il globally (visto che passo in stati in cui vale sempre l'argomento) e quindi la formula è verificata*
- *parto dalla formula più interna (dopo aver valutato dove vale l'argomento, ovvero in q_1 e q_2), ovvero il future, e studiare dove vale. Per proprietà del future vedo che posso partire da q_1 e dire che vale (dato che vale anche in q_1 stesso), posso partire da q_2 per lo stesso ragionamento di q_1. Ragiono ora su q_3, dove non vale $y = 0$, ma da lui parte un solo arco e porta in uno stato dove vale $y = 0$ quindi ogni ca,mino che parte da q_3 soddisfa $y = 0$. Il future vale quindi in ogni stato e di conseguenza anche il globally e quindi la formula è verificata*
- *ragiono come per il caso 2. Per q_3 vale ovviamente il future. Passo a q_1 e già qui vedo che potrei avere un cammino massimale con solo q_1 e q_2 e quindi non vale il future. Analogamente si fa lo stesso per q_2, che non soddisfa il future. Il globally quindi non vale in q_1 e q_2, non valendo lì il future. Studiamo ora q_3 ma anche qui non vale in quanto potrei, in uno dei cammini da q_3 entrare nel loop tra q_1 e q_2, quindi anche per q_3 non vale il globally e quindi la formula non è verificata*
- *prendo il cammino che parte da q_1 e cicla infinitamente con q_3 e q_2, in questo cammino la formula è verificata. In ogni cammino in qui da q_1 passo a q_3 la formula è verificata ma non lo è nel solito loop tra q_1 e q_2 e quindi la formula non è verificata*

Esempio 68. *Studiamo il modello di Kripke della mutua esclusione:*



con:

- $1 = \{rd, snc1, snc2\}$
- $2 = \{rd, req1, snc2\}$
- $3 = \{rd, snc1, req2\}$
- $4 = \{rd, req1, req2\}$
- $5 = \{sc1, snc2\}$
- $6 = \{snc1, sc2\}$
- $7 = \{sc1, req2\}$
- $8 = \{req1, sc2\}$

avendo:

- rd *la risorsa disponibile*
- $sc1$ *e* $sc2$ *le due sezioni critiche*
- $snc1$ *e* $snc2$ *le due sezioni non critiche*
- $req1$ *e* $req2$ *le due richieste pendenti di usare la risorsa*

Si richiede:

- *i due processi non sono mai contemporaneamente nella sezione critica*

- *se un processo richiede la risorsa, prima o poi entrerà nella sezione critica, escludendo la starvation*
- *se un solo processo richiede la risorsa, deve poter accedere alla sezione critica*

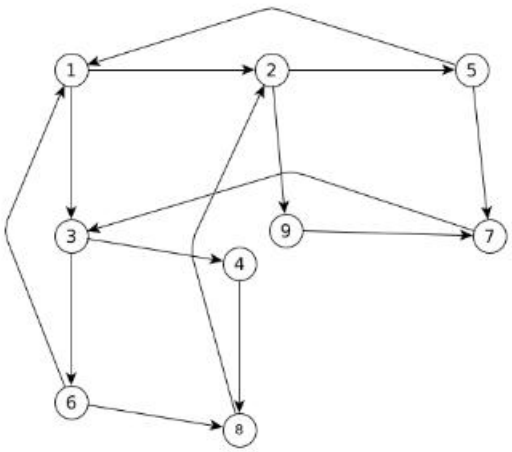
Esprimiamo ciò in logica temporale:

- $\mathbf{G}\,\neg(sc1 \wedge sc2)$
- $\mathbf{G}\,(req1 \implies \mathbf{F}\,sc1)$
- *non è esprimibile in logica temporale in quanto esprime una possibilità*

Studiamo quindi, informalmente i primi due requisiti:

- *la prima proprietà è verificata non avendo stati dove valgono contemporaneamente i due argomenti*
- *la seconda non è verificata avendo un cammino massimale* $(1)(2,4,8)^{\omega}$ *che impedisce al primo processo di entrare in sezione critica (c'è anche un cammino massimale dove non entra mai il secondo)*

Intuitivamente posso ragionare anche sul terzo requisito possiamo dire che è valida in quanto può accedere infinitamente alla risorsa con l'altro che non cerca mai di accederci.
Per validare anche la seconda formula possiamo ricordarci l'ultimo accesso e in caso di conflitto dare priorità sull'altro accesso. Nel modello di Kripke avremmo:

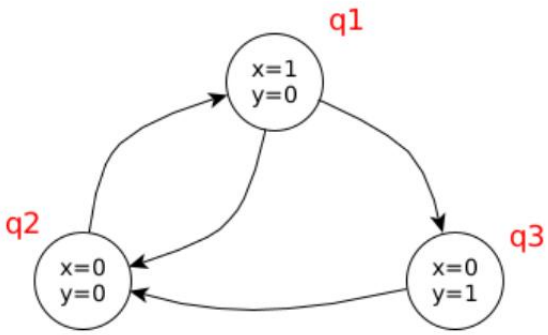


dove lo stato 4 viene sdoppiato in 4 e 9, che hanno le stesse proposizioni atomiche, $rd, req1, re2$ ma da entrambi ora parte un singolo arco che rispettivamente porta dove l'altro processo è in pending, dandogli poi i privilegi di accesso. Si garantisce comunque la mutua esclusione, si soddisfa la seconda proprietà (garantendo alternanza di accesso).

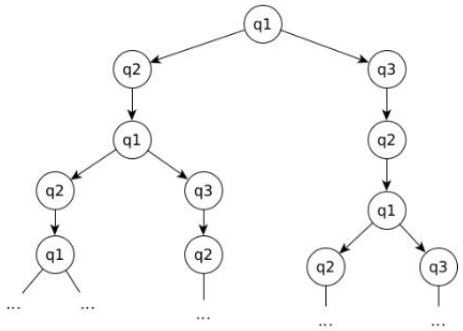
7.2 Computation tree logic

Studiamo ora **alberi di computazione** derivati da modelli di Kripke.

Esempio 69. *Prendiamo il modello:*



Scegliamo come nodo iniziale q_1 e costruiamo un albero che rappresenta le possibili computazioni:



Definizione 78. *Propriamente definiamo **computazione** un cammino dell'albero a partire dalla radice.*

Passiamo ora alla sintassi delle formule ben formate della **Computational Tree Logic (*CTL*)**.

Definizione 79. *Si ha che, dato AP insieme delle proposizioni atomiche:*

- $\forall p \in AP, p \in FBF$
- $\forall\, \alpha, \beta \in FBF$ *valgono gli operatori della logica proposizionale*

Aggiungiamo ora i nuovi connettori, ***A*** *e* ***E****, che si associano ai connettivi temporali:*

- **A** *indica il "per ogni cammino"*
- **E** *indica il "esiste almeno un cammino tale che"*

Questi operatori possono precedere un solo operatore temporale per volta e ogni operatore temporale deve essere preceduto da un quantificatore*.*
Si ha quindi che:

- $\mathbf{AX}\,\alpha, \mathbf{EX}\,\alpha \in FBF$
- $\mathbf{AF}\,\alpha, \mathbf{EF}\,\alpha \in FBF$
- $\mathbf{AG}\,\alpha, \mathbf{EG}\,\alpha \in FBF$
- $\mathbf{A}\,(\alpha\,\mathbf{U}\,\beta)$, $\mathbf{E}\,(\alpha\,\mathbf{U}\,\beta) \in FBF$

Si hanno formule, come $\mathbf{AFG}\,\alpha$*, non sono una formula CTL.*
Si hanno formule LTL che non sono formule CTL, e viceversa*.*

Vari esempi/esercizi su slide del 14 Dicembre.
Si ha che molte proprietà possono essere espresse sia in PLTL che in CTL:

- **proprietà invarianti**, con $\mathbf{G}\,\alpha$ e con $\mathbf{AG}\,\alpha$ in CTL
- **proprietà di reattività** (proprietà invarianti condizionali), con $\mathbf{G}\,(\alpha \implies \mathbf{F}\,\beta)$ e con $\mathbf{AG}\,(\alpha \implies \mathbf{AF}\,\beta)$ in CTL

Ma si hanno anche proprietà non rappresentabili. Prendiamo ad esempio la **reset property**:

$$\mathbf{AG}\,\mathbf{EF}\,\alpha$$

ovvero *da ogni stato raggiungibile in ogni cammino è sempre possibile raggiungere uno stato nel quale vale* α. Questa non può essere espressa in LTL ma solo in CTL.
Vediamo una formula che può essere espressa solo in PLTL:

$$\mathbf{FG}\,\alpha$$

ovvero *in ogni cammino, prima o poi si raggiungerà uno statoa partire dal quale α rimane sempre vera.* Questa non può essere espressa in CTL (se prendo l'albero ho cammini in cui non è vero che α rimane sempre vero) ma solo in PLTL.
Su slide dimostrazione della cosa tramite confronti.
Si ha un'estensione di CTL e PLTL, detta CTL*, nella quale si mantengono i quantificatori sui camini ma si elimina il vincolo CTL per il quale ogni operatore temporale deve avere un quantificatore (posso quindi avere formule del tipo $\mathbf{EFG}\,\alpha$).
Potrei avere formule di CTL* che non sono esprimibili in CTL e PLTL, estendendone la capacità espressiva ma aumentando la complessità degli algoritmi associati. In pratica l'insieme delle formule CTL* contiene gli altri due (che per di più hanno un'intersezione tra loro).

7.3 Model checking

Definizione 80. *Due modelli di Kripke M_1 e M_2 con stati iniziali q_0 e s_0, si ha che essi sono* ***equivalenti*** *rispetto ad una logica I (che può essere PLTL, CTL etc...) se, per ogni formula $\alpha \in FBF_I$, si ha:*

$$M_1, q_0 \vDash \alpha \iff M_2, s_0 \vDash \alpha$$

Avendo che i due modelli sono effettivamente indistinguibili.
L'equivalenza è quindi in funzione di proprietà della logica stessa.
Esempi su slide.

Definizione 81. *Si ha che una relazione parziale $\leq$ su A:*

$$\leq \subseteq A \times A$$

è:

- ***riflessiva****, infatti* $x \leq x,\ \forall\, x \in A$
- ***antisimmetrica****, infatti* $(x \leq y \ \wedge\ y \leq x) \implies x = y,\ \forall x, y \in A$
- ***transitiva****, infatti* $(x \leq y \ \wedge\ y \leq z) \implies x \leq z, \forall\, x, y, z \in A$

La notazione:

$$x < y$$

significa;

$$x \leq y \ \wedge\ x \neq y$$

Definizione 82. *Dato $(A, \leq)$ un insieme parzialmente ordinato e $B \subseteq A$ si definiscono:*

- $x \in A$ *è un* ***maggiorante*** *di B se $y \leq x, \ \forall y \in B$*
- $x \in A$ *è un* ***minorante*** *di B se $x \leq y, \ \forall y \in B$*
- B^* *come* ***insieme dei maggioranti di*** B
- B_* *come* ***insieme dei minoranti di*** B
- B *è* ***superiormente limitato*** *se $B^* \neq \emptyset$*
- B *è* ***inferiormente limitato*** *se $B_* \neq \emptyset$*
- $x \in B$ *è il* ***minimo*** *di B se $x \leq y, \ \forall y \in B$*
- $x \in B$ *è il* ***massimo*** *di B se $y \leq x, \ \forall y \in B$*
- $x \in B$ *è il* ***minimale*** *di B se $y \leq x \implies y = x$*
- $x \in B$ *è il* ***massimale*** *di B se $x \leq y \implies y = x$*

Definizione 83. *Si ha che:*

- *se x è il minimo di B^* allora x è l'****estremo superiore (join)*** *di B:*
$$x = \sup B = \bigvee B$$
- *se x è il massimo di B_* allora x è l'****estremo inferiore (meet)*** *di B:*
$$x = \inf B = \bigwedge B$$

In particolare se $B = \{x, y\}$ si ha che $x \vee y$ indica $\bigvee B$ (se esiste) e $x \wedge y$ indica $\bigwedge B$ (se esiste).

Definizione 84. *Definiamo* ***reticolo*** *un insieme parzialmente ordinato $(L, \leq)$ tale che, $\forall x, y \in L$ esistono $x \vee y$ e $x \wedge y$.*

Definizione 85. *Definiamo* ***reticolo completo*** *se $\bigvee B$ e $\bigwedge B$ esistono per ogni $B \subseteq L$.*

Definizione 86. *Prendiamo due insiemi parzialmente ordinati* $(A, \leq)$ *e* $(B, \leq)$*, si ha che una funzione:*

$$f : A \to B$$

è detta ***monotona*** *se,* $\forall\, x, y \in A$ *vale:*

$$x \leq y \implies f(x) \leq f(y)$$

ovvero se ***preserva la relazione d'ordine****.*
Vale anche nel caso particolare in cui A *e* B *coincidono.*

Definizione 87. *Considerando funzioni con dominio e codominio coincidenti, ovvero del tipo:*

$$f : X \to X$$

si ha che un elemento $x \in X$ *è un* ***punto fisso*** *di* f *se:*

$$f(x) = x$$

Esempio 70. *Vediamo qualche esempio:*

- $f : \mathbb{R} \to \mathbb{R}$*,* $f(x) = x^2$ *e l'insieme dei punti fissi è* $\{0, 1\}$ *(si nota che la funzione non è monotona)*
- $f : \mathbb{R}^+ \to \mathbb{R}$*,* $f(x) = \log x$ *e l'insieme dei punti fissi è* $\emptyset$ *(si nota che la funzione è monotona)*
- $f : \mathbb{R} \to \mathbb{R}$*,* $f(x) = x$ *(ovvero l'****identità****) e l'insieme dei punti fissi è* $\mathbb{R}$

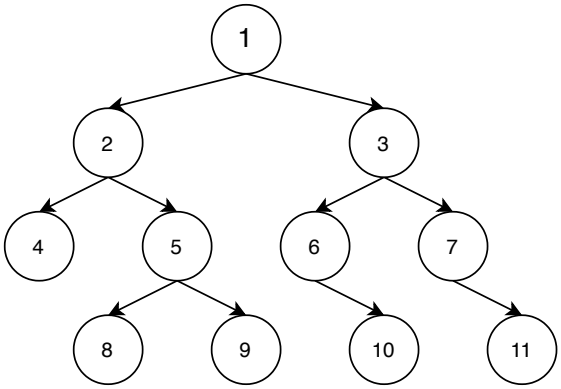
Preso un insieme parzialmente ordinato $(A, \leq)$ e $f : A \to A$ monotona. Ci si chiede se esistono un massimo e un minimo punto fisso.

Esempio 71. *Prendiamo* $A = 2^{\mathbb{N}}$ *e* $S \subseteq \mathbb{N}$*. Vediamo vari esempi:*

- $f(S) = S \cup \{2, 7\}$*, una funzione monotona. Si ha che tutti i sottoinsiemi che contengono* 2 *o* 7 *sono punti fissi (ogni insieme delle parti di* $\{2, 7\}$*), tali sottoinsiemi sono anche i minimi punti fissi. Si ha un massimo punto fisso che è* $\mathbb{N}$
- $f(S) = S \cap \{2, 7, 8\}$*, una funzione monotona. Si ha che il sottoinsieme* $\{2, 7, 8\}$ *è un punto fisso. D'altro canto anche ogni insieme appartenente all'insieme delle parti di* $\{2, 7, 8\}$ *è punto fisso. Il massimo punto fisso è* $\{2, 7, 8\}$ *e come minimo punto fisso abbiamo* $\emptyset$

Si nota una sorta di simmetria tra massimi e minimi punti fissi nel caso di unione o intersezione

Esempio 72. *Sia* $A = \{1, \ldots, 11\}$ *visti come stringhe e con questi 11 elementi si costruisca un albero binario orientato:*



Prendiamo l'insieme $(\mathcal{P}(A), \subseteq)$ *con* $\mathcal{P}$ *che indica l'insieme delle parti. Si ha:*

$$f : \mathcal{P}(A) \to \mathcal{P}(A)$$

con:

$$F(S))S \cup \{x \in A | x \text{ è figlio di un } y \in S\}$$

ad esempio:

$$f(\{2, 6\}) = \{2, 6, 4, 5, 10\}$$

quindi $f(\{2, 6\})$ *non è un punto fisso.*
Si ha anche che:

$$f(\{2, 6, 4, 5, 10\}) = \{2, 6, 4, 5, 10, 8, 9\} = M$$

Ma quindi:

$$f(M) = M \qquad f(\emptyset) = \emptyset$$

I punti fissi di questa funzione sono i cosiddetti ***insiemi chiusi verso il basso****, ovvero un insieme che se contiene un certo nodo contiene anche tutti i discendenti di quel nodo (ovvero tutti i sottoalberi discendenti).*
Il massimo punto fisso è A stesso mentre il minimo punto fisso è l'insieme vuoto (ed esistono vari punti fissi minimali).

Teorema 17 (di Knaster-Tarski)**.** *Sia* $(L, \leq)$ *un reticolo completo (quindi per ogni coppia si hanno join e meet e quindi ogni sottoinsieme finito di elementi di L ha ancora join e meet, anche i sottoinsiemi infiniti) e* $f : L \to L$ *una*

funzione monotona. Si ha in tal caso che f ha un punto fisso minimo e un punto fisso massimo.
Detto altrimenti l'insieme dei punti fissi forma un reticolo completo (e quindi necessariamente si ha un minimo e un massimo).
***Riascolare esempio detto prima della pausa**.*

Dimostrazione. Per semplicità consideriamo $L = \mathcal{P}(A)$ per un certo insieme A, sapendo che l'insieme delle parti è un reticolo completo (nonché un'algebra booleana).
Sappiamo che $f : 2^A \to 2^A$ (con 2^A che è $\mathcal{P}(A)$ è monotona.
Costruiamo l'insieme:

$$\mathcal{Z} = \{T \subseteq A | \, f(T) \subseteq T\}$$

Chiamiamo gli elementi di $\mathcal{Z}$ **punti pre-fissi**. Si ha che $\mathcal{Z}$ non può essere vuoto qualsiasi sia A (se reticolo) e qualsiasi sia f (se monotona) in quanto contiene almeno gli elementi di A.
Poniamo che f abbia dei punti fissi. Si ha che $\mathcal{Z}$ conterebbe tutti questi punti fissi.
Diciamo che:

$$m = \bigcap \mathcal{Z}$$

e si ha che $m \subseteq A$ (m potrebbe essere $\emptyset$).
Si ha che $\forall S \in \mathcal{Z}$ $m \subseteq S$ e ,per la monotonia:

$$f(m) \subseteq f(S)$$

ma quindi si ha che:

$$f(m) \subseteq f(S) \subseteq S$$

Quindi ogni immagine di m è contenuta in S.
Possiamo quindi dire che è contenuto nell'intersezione degli elementi di $\mathcal{Z}$:

$$f(m) \subseteq \bigcap \mathcal{Z} = m$$

e quindi:

$$m \in \mathcal{Z}$$

e quindi m è il minimo di $\mathcal{Z}$ (che non ha nemmeno elementi minimali che non contengono m). Quindi:

$$m = \min \mathcal{Z}$$

Quindi manca da dimostrare che m sia un punto fisso.
Ripartiamo da:

$$f(m) \subseteq m$$

avendo una relazione d'ordine tra due sottoinsiemi di A. MA avendo che f è monotona posso dire che:

$$f(f(m)) \subseteq f(m)$$

e quindi $f(m) \in \mathcal{Z}$. Ma si ha anche che $m \subseteq f(m)$, avendo che $m = \min \mathcal{Z}$. Possiamo quindi concludere che:

$$m = f(m)$$

e quindi è un punto fisso.
Concludendo abbiamo visto che m è minimo ed è un punto fisso.
Il ragionamento si può fare analogo e speculare per il massimo punto fisso.
Riascoltare dimostrazione □

Esempio 73. *Riprendendo l'esempio 72, avendo una funzione monotona avendo dominio e codominio che sono reticoli completi (un reticolo finito è sempre completo), so sicuramente che ci sono un punto fisso massimo e uno minimo.*

Questo teorema è poco utile dal punto di vista algoritmico, dovendo costruire $\mathcal{Z}$ che può essere infinito.
Per lo più il model checking si fa su insiemi finiti ma con un numero di stati magari enorme e quindi analizzare tutti i sottoinsiemi è poco pratico. Si ha quindi un altro teorema.

Teorema 18 (di Kleene)**.** *Sia $f : 2^A \to 2^A$ una funzione monotona. Tale funzione è* ***continua*** *se, con $X_i \subseteq 2^A$:*

$$X_1 \subseteq X_2 \subseteq \cdots \subseteq X_i \subseteq \cdots$$

e si prendano le immagini:

$$f(X_1) \subseteq f(X_2) \subseteq \cdots \subseteq f(X_i) \subseteq \cdots$$

Possiamo dire che:

$$f\left(\bigcup X_i\right) = \bigcup f(X_i)$$

Questa proprietà non è detto che valga se la funzione è solo monotona ma deve essere anche continua e in tal caso si ha che:

- *il minimo punto fisso di f (che esiste per il teorema precedente) può essere calcolato con:*

  $$f(\emptyset), \qquad f(f(\emptyset)), \qquad f(f(f(\emptyset))), \qquad \cdots$$

  calcolando lo "step" successivo fino ad arrivare ad un punto fisso arrivando prima o poi al minimo punto fisso

- *il massimo punto fisso di f (che esiste per il teorema precedente) può essere calcolato con:*

$$f(A), \qquad f(f(A)), \qquad f(f(f(A))), \qquad \cdots$$

arrivando prima o poi al massimo punto fisso (occhio che A è quello di 2^A)

Esempio 74. *Siano $A = 2^{\mathbb{N}}$, $S \subseteq \mathbb{N}$ e $F(S) = S \cup \{2, 7\}$.*
*Applico f all'insieme vuoto e trovo $\{2, 7\}$ che non è punto fisso e quindi proseguo. Applico poi a $\{2, 7\}$ e ritrovo $\{2, 7\}$ che quindi è un punto fisso ed è il minimo. Per il massimo al primo step arrivo a dire che $\mathbb{N}$ è il massimo punto fisso (**non chiaro perché parte da $\mathbb{N}$ e non da $2^{\mathbb{N}}$**).*

Esempio 75. ***Su slide disegni dell'esempio.***
Si supponga di avere un modello di Kripke e si vuole risolvere il model checking globale per $\mathbf{F}\, p$.
Aggiungiamo tutti gli stati in cui in un solo step arriviamo sicuramente in uno stato dove vale p.
Ora rifacciamo aggiungendo gli stati dove con uno step arriviamo sicuramente in uno stato dove vale p o dove vale che allo step successivo sicuramente vale p. Procedo aumentando di volta in volta l'insieme "buono" fino a che non possiamo aggiungere nulla. Abbiamo trovato un minimo punto fisso dell'algoritmo.
Valuto ora $\mathbf{G}\, p$ e ragiono all'inverso di come fatto prima escludendo di volta in volta gli stati in cui non vale $\mathbf{G}\, p$ (quindi l'insieme di partenza, diciamo, è quello in cui vale $\neg p$). Di volta in volta escludo gli stati che mi portano ad avere $\neg p$ più gli eventuali stati già inclusi nell'insieme in uno step precedente fino a che non ho più passi possibili. Si arriva a trovare il massimo punto fisso.

Definizione 88. *Definiamo **automi di Büchi** come automi finiti che riconoscono parole infinite su un alfabeto Σ. Si indicano con la quadrupla:*

$$B = (Q, q_0, \delta, F)$$

con:

- *Q insiemi di stati, detti locations*
- *$q_0 \in Q$ stato iniziale*
- *$\delta \subseteq Q \times \Sigma \times Q$ relazione di transizione*

- $F \subseteq Q$ *insieme di stati accettanti (non si può dire finali in quanto le stringhe sono infinite)*

Presa quindi una parola infinita:

$$w = a_0 a_1 \cdots$$

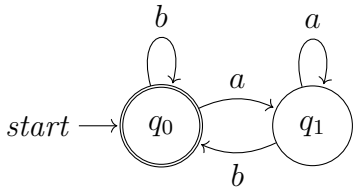
essa è accettata da B *se la sequenza di stati corrispondente* $q_0 q_1 \cdots$ *passa infinite volte per almeno uno stato di* F*. In altri termini mi fermo se uno stato non ha un arco uscente che porta alla lettera successiva altrimenti mi sposto in quello stato ma non serve che mi fermo in uno stato accettante ma è sufficiente passarci infinite volte.*
Diciamo che $L(B)$ *è il linguaggio, ovvero l'insieme di tutte le parole accettate dalla'automa.*

Teorema 19. *Il problema:*

$$L(B) = \emptyset$$

è ***decidibile****.*

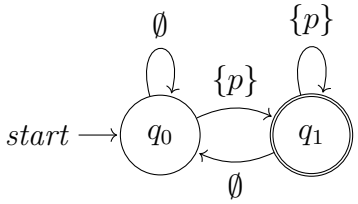
Esempio 76. *Sia:*



Si ha:

- $w_1 = bbbbbbbb \cdots$ *che è accettata*
- $w_2 = bbaaabbbb \cdots$ *che è accettata*
- $w_3 = babababab \cdots$ *che è accettata*
- $w_4 = baabbbaaa \cdots$ *che non è accettata*

Esempio 77. *Studiamo la formula* $\mathbf{GF}\, p$ *e sia:*



Immaginiamo qualche computazione:

- $w_1 = \emptyset\{p\}\{p\}\emptyset\{p\}\emptyset\emptyset\cdots$ *che non è accettata*
- $q_2 = \emptyset\{p\}\emptyset\{p\}\emptyset\{p\}\emptyset\cdots$ *che è accettata*

Possiamo associare ad una computazione su un modello di Kripke una parola che viene eseguita su un certo automa di di Büchi.
Un automa di Büchi corrisponde quindi ad una formula PLTL, ma bisogna notare che l'insieme degli stati dell'automa non ha una relazione diretta con il modello di Kripke.
Vediamo quindi l'algoritmo per vedere se una formula α è vera in (M, q_0):

- costruiamo l'automa che verifica quando α non è verificata $B_{\neg\alpha}$ (non dimenticando i discorsi fatti in merito alla negazione in PLTL)
- trasformiamo M in un automa etichettato come $B_{\neg\alpha}$ ovvero in un automa etichettato da insiemi diproposizioni atomiche
- calcoliamo il prodotto sincrono dei due automi, che chiamiamo PS. Dati due automi definiti sullo stesso alfabeto il prodotto sincrono è un automa che corrisponde ad eseguire in parallelo i due automi in modo che procedano con le stesse etichette. Gli stati sono quindi il risultato del prodotto cartesiano tra gli stati dei due automi
- se $L(PS) = \emptyset$ allora $M, q_0 \vDash \alpha$, in quando nel modello di Kripke M non c'è nessun cammino infinito nel quale non è verificata α, che quindi è valida nel modello di Kripke. Altrimenti si potrebbe eseguire un cammino in M in cui non sarebbe soddisfatto α (si genera in caso anche un controesempio in cui la formula α è violata)

Passiamo ora cercare un algoritmo per CTL.

Definizione 89. *Fissiamo $M = (Q, T, I)$ e sia α una formula. Si definisce* ***estensione di*** *α:*

$$[[\alpha]] = \{q \in Q |\, M, q \vDash \alpha\}$$

Se $\alpha = \top$ estensione è l'insieme di tutti gli stati, se è $\bot$ è l'insieme vuoto. Se la formula è una proposizione atomica allora l'estensione è l'insieme degli stati dove vale la proposizione atomica. Se ho un connettivo tra due formule già estese segue la logica dell'operatore (esempio se ho l'and in un certo stato

è valido sse valgono entrambe le proposizioni).
Passiamo a CTL.
Sia $\alpha \equiv \mathbf{AF}\,\beta$*. Si definisce la funzione:*

$$f_\alpha : 2^Q \to 2^Q$$

ovvero una funzione che prende un sottoinsieme di stati e lo trasforma in un sottoinsieme di stati. Si ha che $\forall H \subseteq Q$*:*

$$f_\alpha(H) = [[\beta]] \cup \{q \in Q | \forall (q, q') \in T : q' \in H\}$$

Si noti che $f_a(\emptyset) = [[\beta]]$*.*
Si dimostra che $[[\alpha]]$ *è il minimo punto fisso di* f_α*.*
Ogni volta si aggiungono nuovi stati.
Posso usare lo stesso algoritmo per vedere se α *vale in un certo stato, osservando se alla fine lo stato che ci interessa è contenuto nell'estensione.*
Sia ora $\alpha \equiv \mathbf{EG}\,\beta$*. Si definisce la funzione:*

$$g_\alpha : 2^Q \to 2^Q$$

Si ha che $\forall H \subseteq Q$*:*

$$g_\alpha(H) = [[\beta]] \cap \{q \in Q | \exists (q, q') \in T : q' \in H\}$$

Si noti che $g_\alpha(Q) = [[\beta]]$*.*
Si dimostra che $[[\alpha]]$ *è il massimo punto fisso di* g_α*.*
Ogni volta si tolgono stati.
In generale si nota che il "per ogni" e l' "esiste" compaiono nella formula.
Per l'operatore next è sufficiente guardare il passo successivo quindi non serve un ragionamento così complesso mentre per l'until si fa un ragionamento simile al future, essendo nella pratica una sua forma rafforzata (poi si cerca, come per il future, il minimo punto fisso).

Possiamo quindi trovare un algoritmo per CTL che sarà ricorsivo.

Definizione 90. *Definiamo* μ ***calcolo*** *come un linguaggio logico che permette di definire formule ricorsive.*

Si supponga di avere solo l'operatore temporale next e i quantificatori. Cerchiamo di esprimere, solo con il next, la proprietà $\mathbf{EF}\,\alpha$. Si ha quindi una sorta di srotolamento:

$$\mathbf{EF}\,\alpha \equiv \alpha \vee \mathbf{EX}\,\alpha \vee \mathbf{EXEX}\,\alpha \vee \cdots$$

che è una formula infinita ma con una struttura ben definita, dal secondo termine si inizia sempre con **EX**. Raccogliamo:

$$\mathbf{EF}\,\alpha \equiv \alpha \vee \mathbf{EX}(\,\alpha \vee \mathbf{EX}\,\alpha \vee \cdots)$$

ma quindi tra parentesi si ha in pratica la formula completa di $\mathbf{EF}\,\alpha$ e quindi posso dire che:

$$\mathbf{EF}\,\alpha \equiv \alpha \vee \mathbf{EX}\,(\mathbf{EF}\,\alpha)$$

Fisso quindi una notazione per scrivere questa cosa:

$$\mu Y.(\alpha \vee \mathbf{EX}\,Y)$$

Dove la μ indica tradizionalmente il minimo punto fisso, Quindi la formula si legge "minimo punto fisso della funzione $(\alpha \vee \mathbf{EX}\,Y)$ dove Y è l'oggetto che cerchiamo di definire". È quindi una forma compatta di scrivere la formula infinita.
Passo a $\mathbf{AG}\alpha$:

$$\mathbf{AG}\,\alpha \equiv \alpha \wedge \mathbf{AX}\,\alpha \wedge \mathbf{AXAX}\,\alpha \wedge \cdots$$

e quindi, raccogliendo:

$$\mathbf{AG}\,\alpha \equiv \alpha \wedge \mathbf{AX}\,(\mathbf{AG}\alpha)$$

ovvero:

$$\nu Y.(\alpha \wedge \mathbf{AX}\,Y)$$

Possiamo generalizzare fino ad avere un linguaggio completo, una nuova logica, detto appunto **calcolo** μ.

Definizione 91. *Dato AP l'insieme delle proposizioni atomiche e siano α e β due formule. Si ha che la sintassi del linguaggio è:*

- *$\alpha \vee \beta$ e $\neg\alpha$ sono formule (forse anche le altre ottenute con connettivi logici (???))*
- ***AX**$\,\alpha$ e **AX**$\,\alpha$ sono formule*
- *$\mu Y.f(Y)$ è una formula è una formula, dove f è una formula nella quale compare Y (con restrizioni sulle negazioni)*
- *$\nu Y.f(Y)$ è una formula è una formula, dove f è una formula nella quale compare Y (con restrizioni sulle negazioni)*

Si ha quindi un modo molto libero di costruire formule.
La semantica del calcolo μ è definita sui modelli di Kripke attraverso gli operatori di punto fisso, usando idee simili a quelle usate per definire le estensioni di formule temporali.

Si nota che:

$$CTL^* \subset \text{calcolo } \mu$$

quindi tutte le formule di CTL^* sono esprimibili nel μ calcolo che quindi è più espressivo di tutte le logiche finora considerate (ed esprime anche cose che le altre non possono). La massima potenza espressiva si paga in complessità degli algoritmi e con una sorta di "oscurità" delle formule, che non sono facilmente leggibili ed interpretabili.
Vediamo qualche dettaglio sulla complessità temporale, dato un modello di Kripke M e una formula f, a parità di algoritmo canonico:

- PLTL ha complessità $O(|M| \cdot 2^{|f|})$
- CTL ha complessità $O(|M| \cdot |f|)$

In realtà la dimensione della formula rende difficile un confronto diretto in quanto una proprietà può avere una formula breve in LTL e una lunga in CTL (come abbiamo già visto). Anche il numero di stati, $|M|$, è da non trascurare in quanto può essere immenso, anche dal punto di vista della complessità spaziale. Si hanno quindi strategie per la trattazione efficiente, tra cui:

- rappresentazioni simboliche, tramite diagrammi di traduzione binari (Ordered Binary Decision Diagrams, OBDD)
- partial order reduction (tramite unfolding), spesso usato per le reti di Petri. SI hanno tecniche efficienti dal punto di vista spaziale
- traduzione in SAT, usando i SAT solver

Quindi, per concludere, per fare model checking una tecnica è ridurre il problema ad un modello di kripke.
Parliamo giusto un secondo di **fairness**.

Definizione 92. *Si dice che un'esecuzione è* ***unfair*** *se un evento rimane sempre abilitato da un certo istante in poi ma non scatta mai.*

Definizione 93. *Si definisce* ***fairness debole*** *se un evento che è abilitato prima o poi scatta o viene disabilitato.*

Bisogna quindi limitare la valutazione di una formula alla sua esecuzione **fair**, nella realtà è irrealistico pensare che qualcosa non scatti mai essendo sempre abilitata.

Definizione 94. *Si definisce* ***fairness forte*** *se ogni evento che viene abilitato infinite volte scatta infinite volte (a lungo termine senza curarsi degli aspetti quantitativi). Ovvero, in PLTL, se:*

$$\mathbf{GF}\,(\textit{evento è abilitato}) \rightarrow \mathbf{GF}\,(\textit{evento scatta})$$

Su slide vari esempi per la fairness.
Per il model checking si hanno diversi tool, tra cui:

- **Spin** per PLTL, tramite un linguaggio chiamato Promela
- **NuSMV** per PLTL e CTL
- **mCLR2** per il μ calcolo
- **TiNA** per PLTL e reti di Petri

Capitolo 8

Seminario su logiche quantistiche e sistemi concorrenti

Nota: queste note, prese durante il seminario del prof Bernardinello, tenuto l'11 Febbraio 2021, sono incomplete, parziali e probabilmente scorrette in determinati punti, soprattutto a causa del lag della chiamata.

Si parla di logica in relazione a sistemi concorrenti, superando il punto di vista non concorrente visto coi modelli di Kripke.
Partiamo da alcune considerazioni precedenti a quelle che riguardano direttamente la concorrenza.
Immaginando gli stati di un modello di Kripke, con un insieme di stati Q e un insieme di transizioni T, si ha una funzione di transizione:

$$I : Q \to 2^{AP}$$

con AP insieme delle proposizioni. Prendiamo un $p \in AP$ e prendiamo l'insieme degli stati nel modello dove è vero p, ugualmente facciamo con una certa $1 \in AP$. Tali insiemi di stati potrebbero avere un'intersezione non vuota e in tali stati vale $p \wedge q \in \mathbb{F}$, con $\mathbb{F}$ che rappresenta l'insieme delle formule proposizionali. La congiunzione logica è quindi l'intersezione tra insiemi. Ragionamenti analoghi si fanno per gli altri operatori logici. Qualunque sottoinsieme di stati corrisponde ad una certa proposizione e quindi, dato Q, costruiamo 2^Q e ogni elemento di questa famiglia è una proposizione. Può capitare che due proposizioni atomiche vengano associate allo stesso insieme di stati. Si dice che una proposizione è per definizione un sottoinsieme di stati e si parla di **dualità stati-proprietà** (con *proprietà* come stato associato alla proposizione). Quindi una **proprietà** è per definizione un insieme

di *stati* e uno **stato** è per definizione un insieme di *proprietà*, per questo si parla di **dualità**.
Se immaginiamo di prendere la famiglia 2^Q e di avere un sottoinsieme di soli tre elementi, per esempio, posso disegnare un grafo con $X \to Y$ che indica che X è contenuto in Y. Parto quindi da $\emptyset$, poi i sottoinsiemi con un solo elemento, poi con due e il singolo con tre, con gli archi che indicano l'essere contenuto, ovvero $\subseteq$. Si ha quindi una relazione di ordine parziale ma si può vedere anche come *implicazione logica* $X \implies Y$, con $\emptyset$ che è la costante logica $\bot$ mentre l'insieme completo a $\top$. Si è creato un reticolo o meglio un'**algebra di bool**. Un algebra di bool è un reticolo con anche *join($\vee$), meet ($\wedge$), complemento/not ($\neg$).* Possiamo identificare la logica proposizionale classica con le algebre di bool.
Introduciamo nel discorso l'idea di sistema concorrente. Pensando ai *sistemi di transizione* abbiamo automi che fissano uno stato in un certo istante temporale t. Nei sistemi concorrenti il concetto di *stato globale* non è osservabile (per un discorso di velocità dei segnali e tempo che passa prima della ricezione). Si pensi che nella fisica non esiste un sistema di riferimento temporale unico e privilegiato e questo discorso è approfondito nella *teoria della relatività* (dove si introduce anche la velocità della luce come limite superiore di velocità).
Un sistema distribuito è quindi intrinsecamente concorrente e nessun componente può osservare lo stato globale, nemmeno un osservatore esterno, non si ha determinismo.
Per ovviare al problema Petri introduce le *reti di Petri*.
Discorso utile nelle reti di Petri è quello di condizione complementare, l'aggiunta di tali condizioni non cambia il comportamento della rete. La stessa cosa accade se aggiungo uno stato che è una **combinazione logica** di stati già presenti.
Portare avanti questa osservazione si ha la **saturazione della rete**, ovvero si aggiungono tutti gli stati che non modificano il comportamento della rete.
La logica delle condizioni delle reti di Petri non è la logica classica dell'algebra booleana. La teoria delle reti Petri nasce appunto dal fatto che le semantiche interleaving non sono adeguate per i sistemi concorrenti.
Si ha un **poset ortomodulare** (parzialmente ordinata) o **logica quantistica** come la *struttura logica degli stati locali*, dove appunto non valgono alcune proprietà delle algebre booleane.
La seguente parte formale è tratta dalle slide, purtroppo la discussione in merito è persa causa lag.
Formalmente prendiamo una rete:

$$N = (B, E, F, C)$$

si ha che, per $x, y \in B$:

$$x \leq y \iff \forall c \in C : x \in c \implies y \in c$$

Si prende quindi anche una rete:

$$N' = (B', E, F', C')$$

avendo che:

- $B \subseteq B'$
- $CG(N)$ è isomorfo $CG(N')$
- N' è *saturo* di stati locali

Teorema 20. *Date le premesse fatte su N' si ha che:*

$$\langle B', \leq, \bot, \top, (,)' \rangle$$

è un ***poset ortomodulare****.*

Definizione 95. *Preso:*

$$P = \langle P, \sqsubseteq, \bot, \top, (,)' \rangle$$

si definisce ***prime filter*** *$f \subseteq P$ tale che:*

- $x \in f, x \leq y \implies y \in f$
- $x, y \in f, x \$ y \implies x \wedge y \in f$

Ogni componente osserva le proprie condizioni locali in modo oggettivo e istantaneo ma dal punto di vista globale tutto l'insieme è formato all'unione delle algebre booleane dei singoli elementi. Non posso quindi usare sempre lo stato globale per ottenere informazioni utili, serve una *logica quantistica*, concetto nato nel tentativo di studiare sistemi di meccanica quantistica, dove si ha il fenomeno per il quale, rompendo i principi della meccanica classica, non si possono osservare simultaneamente due quantità fisiche diverse (questa caratteristica è simile a quella che si ottiene in un sistema distribuito, dove non posso osservare lo stato globale in un certo istante).