UNIVERSITÀ DEGLI STUDI DI FIRENZE
FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica

Appunti del Corso di Informatica Industriale 2

Umberto Monile

4.10 Guasti software su un sistema distribuito

Un guasto software in un sistema distribuito si ripercuote su tutti i nodi; per tollerare guasti software si fa ricorso alla diversità (ho repliche di HW diversi su cui posso far funzionare software diverso).
Il software usato da questi sistemi sarà composto da:

1. Algoritmi Funzionali.
2. Meccanismi di Tolleranza ai guasti: quest'ultimi permettono di ottenere la consistenza anche in presenza di guasti HW o guasti agli algoritmi funzionali.

L'architettura di un software di comunicazione distribuito tra più nodi è così definito:

- Livello Applicativo (Algoritmi funzionali)
- *Middleware* (Meccanismi di Tolleranza)
- Software di Base (Comunicazione)

Mentre sul livello applicativo i guasti software possono essere tollerati per **diversità**, per gli altri due livelli le cose cambiano; più precisamente il middleware non deve contenere guasti, mentre la comunicazione deve rispettare le assunzioni di guasto che sono state fatte.

Per poter fare la **diversità** a livello applicativo, è necessario fare più versioni dello stesso algoritmo (per esempio comprarlo n volte o installarlo in n sistemi operativi differenti) e questo ha un costo, quindi si preferisce avere il livello applicativo senza guasti è per verificare tale assenza lo si fa tramite:

- Verifica Formale.
- *Testing* (non è esaustivo ma molto usato in ambito industriale).

Anche sul software di base si fa la verifica oppure la prova sul campo (*proven in use*).

COTS (***C**ommercial **O**ff-**t**he-**S**helf component)*, si riferisce a componenti hardware e software disponibili sul mercato per l'acquisto da parte di aziende di sviluppo interessate a utilizzarli nei loro progetti, in questo modo evitano alle aziende acquirenti di fare tutti i processi di verifica in quanto sono già stati fatti dalle case produttrici.

Il processo di verifica del software viene scelto in base al costo e all'utilità, comunque ci sono delle normative da rispettare.

Definizione di Integrità della Sicurezza: Il grado di fiducia assegnato al sistema per svolgere soddisfacentemente le funzioni di sicurezza richieste in tutte le condizioni fissate e all'interno di un fissato periodo di tempo

SIL (Safety Integrity Level): Insieme di livelli discreti utilizzati per specificare i requisiti di integrità della sicurezza da assegnare ai sistemi:

- SIL 1-4: al valore 4 è associato il livello di integrità più alto mentre al valore 1 è associato il più basso.

Nelle normative EN 50128, IEC 1508 è definito anche il livello 0 per indicare che non ci sono requisiti di Sicurezza. Un sistema con SIL 0 è detto sistema non critico altrimenti è detto critico. Il SIL 0 significa che un guasto su qualche componente non ha effetto sulla sicurezza, mentre SIL 4 il guasto ha effetto immediato e diretto sulla sicurezza.

Esempio di descrizione qualitativa del SIL (normativa Def-STAN 00-56)
"*… claim limits shall be determined for each Safety Integrity Level that give the inimum failure rate that can be claimed for a function or component of that level…*"

Claim Limits

Safety Integrity Level	**Minumum failure rate**
S4	Remote
S3	Occasional
S2	Probable
S1	Frequent

Una volta definito il SIL del sistema, tramite il *SIL apportonment* si definisce il SIL dei vari componenti, per esempio se il sistema ha un SIL4 ed uso la diversità, i vari moduli software possono avere un SIL più basso ad es 3 o 2.

Si possono avere due tipi di interazione negativa nel caso in cui un oggetto di livello di SIL più basso interagisce con uno di SIL più alto (si può indicare così $il(A) < il(B)$):

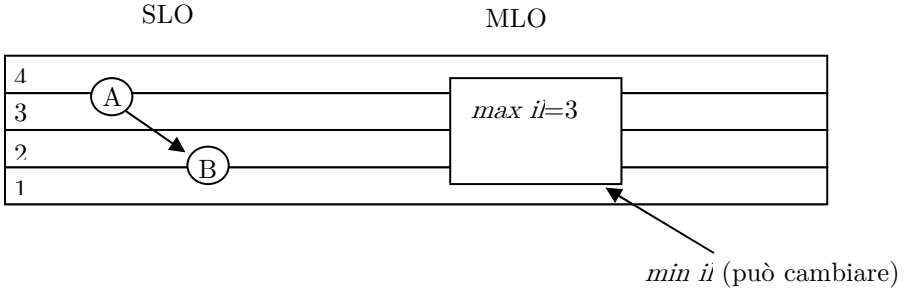
1. A blocca B vediamo come:
   - tramite meccanismi di priorità: un S.O real time da priorità ad A.
   - A impedisce a B di ricevere messaggi (DoS).
2. A invia informazioni scorrette a B o scrive scorrettamente sulle variabili di B.

Per evitare il verificarsi di queste due interazioni negative si adotta l'architettura "*Service Oriented*": questa architettura tende alla coesistenza di moduli (oggetti) con SIL diversi in un sistema secondo la *Multi-Level Integrity Policy.*

Supponiamo di avere una funzione Object-Oriented: si hanno degli oggetti ai quali è associato un livello di integrità. Questi oggetti possono avere un solo livello di integrità o più infatti si parlerà di :

1. **SLO (*single level object*):** ogni oggetto ha un solo livello di integrità che rimane costante nel tempo ed è pari a quello prescritto dalla normativa di riferimento. Un oggetto può ricevere dati (essere invocato) solo da oggetti con un livello di integrità (il) superiore e non può inviare dati (invocare) a oggetti con livello di integrità (il) superiore.
2. **MLO(*multiple level object*):** un oggetto può con livello di integrità variabile tra un *max il* che è il livello di integrità al quale sono stati validati ed un *min il* che è il più basso livello di integrità a cui possono scendere per ricevere dati da un oggetto con livello di integrità inferiore, riassumendo MLO[*max il*, *il*, *min il*] dove:
   - *max il* è il livello intrinseco (costante);
   - *il* livello attuale;
   - *min il* è il livello minimo raggiungibile durante l'esecuzione.

Esempio:



Dato un livello di integrità le normative dicono quale deve essere il tipo di test da fare perché il software sia integro (ad esempio nel campo avionico si ha la copertura del testing a livello MCDC (*Modified Condition Decision Coverage*)).
Le normative sono rispettate se (consideriamo il caso di due oggetti SLO A e B):

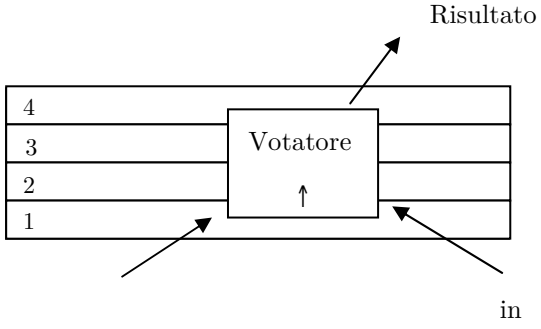
- A read B (A←B) $il(A) \leq il(B)$
- A write B (A→B) $il(A) \leq il(B)$
- A r&w B (A↔B) $il(A) = il(B)$

Vediamo nel dettaglio cosa succede quando si instaura una comunicazione tra componenti di tipo diverso o meglio quale sono le normative da rispettare considerando condizioni, effetti all'invocazione e gli effetti al ritorno dell'invocazione:

	Condizioni	**A&B (SLO)**	**A SLO & B (MLO)**
	A read B A write B A r&w B	il(A) ≤ il(B) il(A) ≥ il(B) il(A)= il(B)	il(A) ≤ il(B) il(A) ≥ il(B) il(A)= il(B)
Effetti Invocazione	A read B A write B A r&w B		min il(B) = il(A); il(B):= maxil(B) il(B) := min(il(A), max il(B)) min il(B); il(B):= il(A)
Effetti Ritorno	A read B A write B A r&w B		

	Condizioni	**A (MLO) & B (SLO)**	**A & B (MLO)**
	A read B A write B A r&w B	min il(A) ≤ il(B) il(A) ≥ il(B) min il(A) ≤ il(B) ≤ il(A)	min il(A) ≤ max il(B) true min il(A) ≤ max il(B)
Effetti Invocazione	A read B A write B A r&w B		min il(B)=min il(A); il(B)=maxil(B) il(B)= min(il(A), max il(B)) min il(B)=min il(A); il(B)=min(il(A), max il(B))
Effetti Ritorno	A read B A write B A r&w B	il(A) := min(il(A),il(B)) il(A) := min(il(A),il(B)) il(A) := min(il(A),il(B))	il(A) := min(il(A),il(B)) il(A) := min(il(A),il(B)) il(A) := min(il(A),il(B))

Esistono dei *validation object* che alzano il livello di integrità (*il*) dei dati, ad esempio il votatore. In generale se si rispettano le regole in tabella si può far coesistere più oggetti diversi a diversi livelli di integrità.



5. Logica

5.1 Logica delle proposizioni

Sintassi:

$$\varphi := \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \text{true} \mid \text{false} \mid p$$

dove p è una proposizione che può valere true o false.

Questa definizione è ridondante perché ad esempio:

- $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
- false = ¬true

A questo punto possiamo ridefinire la sintassi solo con gli operatori primitivi:

$$\varphi := \neg\varphi \mid \varphi \wedge \varphi \mid \text{true} \mid p$$

Semantica:
Data una formula φ definiamo una funzione S(φ) dove S: **Φ**→{true, false} con **Φ** l' insieme delle formule, per fare questo serve una funzione di valutazione:
si chiama funzione di valutazione una funzione che va dall'insieme P nell'insieme {true, false}

$$V : P \rightarrow \{\text{true, false}\}.$$

- S(false) = false.
- S(P) = V(P).
- $S(\neg\varphi) = \neg S(\varphi) = \begin{cases} \text{false} & \text{se } S(\varphi) = \text{true} \\ \text{true} & \text{se } S(\varphi) = \text{false} \end{cases}$
- $S(\varphi_1 \wedge \varphi_2) = S(\varphi_1) \wedge S(\varphi_2) = \begin{cases} \text{false} & \text{altrimenti} \\ \text{true} & \text{se } S(\varphi_1) = \text{true e } S(\varphi_2) = \text{true} \end{cases}$

La naturale estensione di questa logica è il calcolo dei predicati del 1° ordine.

5.2 Logica dei predicati del primo ordine

Sintassi:

$$\varphi := \neg\varphi \mid \varphi \wedge \varphi \mid \text{true} \mid p(\text{exp}, \ldots\ldots, \text{exp})$$

Con exp:=x | $f(\text{exp}_1, \ldots\ldots, \text{exp}_n)$ dove x è una variabile e $f : D^n \rightarrow D$.

In questo caso p non è una proposizione ma è il predicato:

$$P : D^n \rightarrow \{\text{true, false}\}$$

Inoltre possiamo introdurre un **quantificatore** tra gli operatori logici: quantificatori $\forall$ ed $\exists$, denominati rispettivamente il quantificatore universale e il quantificatore esistenziale.

Esempio: $\exists x : \varphi$ (esiste x per cui φ è vera).

lega la variabile x variabile x è libera

La negazione di una formula universale è una formula esistenziale e viceversa:

$$\neg\forall x = \exists x$$
$$\neg\exists x = \forall x$$

Inoltre posso definire:

- false = ¬true.
- $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\forall x : \varphi = \neg\exists x : \neg\ \varphi$

Semantica:

Per definire la semantica dobbiamo introdurre la struttura di interpretazione $\boldsymbol{\sigma}$=(D, V, $\boldsymbol{\xi}$, $\boldsymbol{\pi}$) dove:

- **D** è il dominio di interpretazione delle variabili.
- **V** è la funzione di valutazione delle variabili.
- $\boldsymbol{\xi}$ è la funzione di interpretazione delle funzioni $\boldsymbol{\xi} : F \rightarrow (D^n \rightarrow D)$.
- $\boldsymbol{\pi}$ è la funzione di interpretazione dei predicati $\boldsymbol{\pi} : P \rightarrow (D^n \rightarrow \{\text{true, false}\})$

A questo punto definiamo la semantica come :

- S': exp→ D
- $S : \varphi \rightarrow \{\text{true, false}\}$

In particolare:

- S'(x) = V(x) valutazione di x
- S'(f(exp_1, ….., exp_n)) = ξ(f)(S'(e_1),…….,S'(e_n))
- S(true) = true
- S($\neg\varphi$) = $\neg$S(φ)
- S($\varphi_1 \wedge \varphi_2$) = S($\varphi_1$) $\wedge$ S(φ_2)
- S(p(e_1, ….., e_n)) = π(p)(S'(e_1), …., S(e_2))
- S($\exists$x : φ) = ….
  poiché $\exists$x : φ lega le occorrenze di x in φ, x non ha valore in V, infatti solo le x libere hanno un valore in V, allora si può concludere dicendo:

$$S(\exists x \cdot \varphi) = \begin{cases} \text{se } \exists y \in D : S(\varphi)_{[V/V \cup [x \to y]]} = \text{true} \quad \text{allora} \quad \text{true} \\ \\ \text{altrimenti false} \end{cases}$$

Possiamo ridefinire la semantica attraverso la relazione di soddisfacibilità (|=):

$$\boldsymbol{\sigma} \models \varphi \;=\; \begin{cases} \boldsymbol{\sigma} \text{ soddisfa } \varphi \\ \\ \boldsymbol{\sigma} \text{ è un modello per } \varphi \text{ quando } \varphi \text{ è vera} \end{cases}$$

Ottenendo così:

- $\sigma \models$ true sempre
- $\sigma \models \neg\varphi$ se solo se $\neg\sigma \models \neg\varphi$
- $\sigma \models \varphi_1 \wedge \varphi_2$ se solo se $\sigma \models \varphi_1 \wedge \sigma \models \varphi_2$
- $\sigma \models$ p(e_1, ….., e_n) se solo se $\sigma \models \pi$(p)(S'(e_1), …., S(e_2))=true
- $\sigma \models \exists$x : φ sse $\exists\sigma$' : $\sigma' \models \varphi \wedge \sigma' = (D, V, \xi, \pi) \wedge V' = V \cup [x \to y] \wedge y \in D$

Esempio: Consideriamo la formula: $\exists$x : p(x_1, f(x_1, x))
Consideriamo:

- p come >
- f come +
- D = $\mathbb{N}^+ \setminus \{0\}$

Dunque ho $\exists$x : $x_1 > x_1 + x$ = false sempre, dunque la struttura presa in considerazione non è un modello per φ.

Se invece consideriamo:

- p come >
- f come +
- D = $\mathbb{Z}$

Dunque ho $\exists$x : $x_1 > x_1 + x$ = true, dunque la struttura presa in considerazione è un modello per φ.

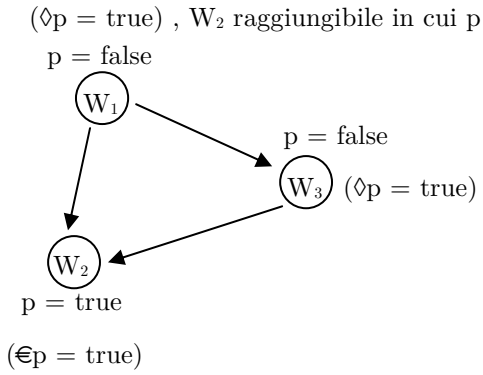
5.3 Logica Modale

Nell'ambito della logica formale, si indica come **logica modale** una qualsiasi logica in cui è possibile esprimere il "modo" in cui una proposizione è vera o falsa (calcolo proposizioni). Generalmente la logica modale si occupa dei concetti di **possibilità** e **necessità**, infatti, si ha che p è necessariamente vera (p) o p è possibilmente vera (◊p).
La possibilità o necessità si riferisce all'esistenza di "mondi", comunque raggiungibili, in cui p possa o debba essere vera:

- p significa che in tutti i mondi raggiungibili p è vera.
- ◊p significa che ∃ un mondo raggiungibile in cui p è vera.

Proprietà della dualità: ¬p = ¬◊p

Esempio:



Sintassi:

$$\varphi := \neg\varphi \mid \varphi \wedge \varphi \mid \text{true} \mid \quad \varphi \mid p$$

Semantica:
Per definire la semantica dobbiamo introdurre la struttura di interpretazione **σ** = (W, R, V) dove:

- **W** è l'insieme dei mondi.
- **R** è la relazione di raggiungibilità tra mondi R ⊆ W×W.
- **V** è la funzione di valutazione delle proposizioni V : P×W→{true, false}

In particolare consideriamo w_0 il mondo di partenza :

- $\sigma,w_0 \models$ true sempre
- $\sigma,w_0 \models \neg\varphi$ se solo se $\sigma \neq \varphi$
- $\sigma,w_0 \models \varphi_1 \wedge \varphi_2$ se solo se $(\sigma \models \varphi_1) \wedge (\sigma \models \varphi_2)$
- $\sigma,w_0 \models p$ se solo se $V(p, w_o)$=true
- $\sigma,w_0 \models \varphi$ sse $\forall w \in W$: $(w_o, w) \in R$, $V(\varphi, w)$ = true cioè $(\sigma,w \models \varphi)$

Spesso si trascura il riferimento alla struttura di interpretazione σ scrivendo $(w_0 \models_\sigma)$ invece di $(\sigma,w_0 \models)$.

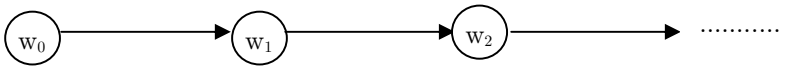
Le proprietà della relazione di raggiungibilità considerate inducono degli assiomi di equivalenza tra formule della logica:

1. **Transitiva**: $w_0 R w_1$, $w_1 R w_2 \Rightarrow w_0 R w_2$:
   i) $\varphi \Rightarrow \varphi$
   ii) $\Diamond\Diamond\varphi \Rightarrow \Diamond\varphi$

2. **Riflessiva**: $\forall w \in W$, $w R w$:
   i) $\varphi \Rightarrow \Diamond\varphi$
   ii) $\varphi \Rightarrow \varphi$
   iii) $\Diamond\varphi' \Rightarrow \Diamond\Diamond\varphi'$
   iv) $\varphi' \Rightarrow \varphi'$

3. **Riflessiva e Transitiva** (assorbimento delle modalità):
   $\varphi = \varphi$ $(\varphi \Rightarrow \varphi$ e $\varphi \Rightarrow \varphi)$
   $\Diamond\varphi = \Diamond\Diamond\varphi$ $(\Diamond\varphi \Rightarrow \Diamond\Diamond\varphi$ e $\Diamond\Diamond\varphi \Rightarrow \Diamond\varphi)$

CASO 1: Supponiamo che W sia infinito e numerabile, con la relazione di raggiungibilità della forma $R = \bigcup_{i=0}^{\infty} \{w_i, w_{i+1}\}$, che non è altro che una catena lineare:

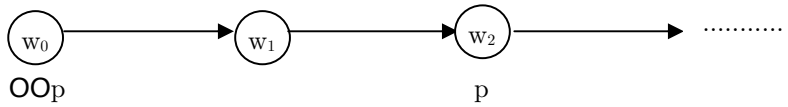


Allora vale sia $\varphi \Rightarrow \Diamond\varphi$ (se φ è vera in tutti i mondi raggiungibili allora esiste un mondo in cui φ è vera) che $\Diamond\varphi \Rightarrow \varphi$ (se esiste un mondo in cui φ è vera allora φ è vera in tutti i mondi).
Dunque le due modalità si equivalgono, cioè $(\varphi \Rightarrow \Diamond\varphi)$ e $(\Diamond\varphi \Rightarrow \varphi)$ quindi posso affermare che $\varphi = \Diamond\varphi$.

In questo caso si usa spesso un unico simbolo O, che si legge anche "next" perché Op indica che p è vera nel prossimo mondo raggiungibile.

Vale ancora la dualità $\neg p = \neg\Diamond p \Rightarrow \mathsf{O}\neg p = \neg\mathsf{O}p$

Esempio:



OOP è vero in w_0 se p è vera in w_2

CASO 2: Consideriamo il caso di W infinita e numerabile con $R = \bigcup_{i=0}^{\infty}\{w_i, w_{i+1}\}$ **riflessiva e transitiva** (chiusura transitiva), non vale più la proprietà $\Diamond\varphi = \varphi$, mentre vale :

- $\varphi \Rightarrow \varphi$
- $\varphi \Rightarrow \Diamond\varphi$
- $\varphi \Rightarrow \Diamond\varphi$

Per la proprietà riflessiva e transitiva abbiamo:

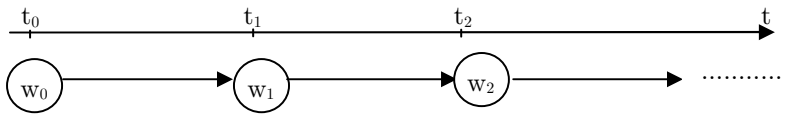
- $\varphi = \varphi$
- $\Diamond\varphi = \Diamond\Diamond\varphi$

Dove:

- φ: significa che $\forall$ mondo prossimo (incluso se stesso) in cui è vera φ.
- $\Diamond\varphi$: significa che $\exists$ un mondo (incluso se stesso) in cui è vera φ.

5.4 Logica Temporale

Supponiamo di considerare un dominio temporale; l'insieme dei mondi è il cosiddetto tempo discreto dove ogni mondo è un istante di tempo. Gli istanti di tempo t_0,t_1,....t_n sono chiamati STATI.
Il sistema evolve attraverso vari stati quindi una sua evoluzione è una successione di stati (o traccia o storia):



Nella logica temporale:

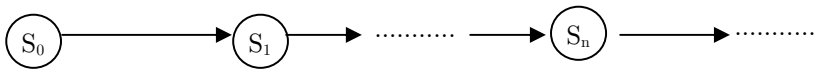
- $\Box$ diventa **ALWAYS** (sempre); quindi $\Box\varphi$ indica che φ è vero d'ora in poi: $t_0 \models_\sigma \Box\varphi$ se solo se $\forall t \geq t_0$ il tempo $t \models \varphi$
- $\Diamond$ diventa **EVENTUALLY** (prima o poi) quindi $\Diamond\varphi$ indica che φ sarà vero prima o poi: $t_0 \models_\sigma \Diamond\varphi$ sse $\exists t \geq t_0$ il tempo $t \models \varphi$
- L'operatore $\bigcirc$ **NEXT** non subisce cambiamenti, quindi $\bigcirc\varphi$ indica che φ sarà vero nell'istante successivo.

Logica temporale si divide nelle seguenti categorie:

1. Ramificata o Lineare.
2. Numero finito o Numero Infinito di stati.
3. Tempo passato o Tempo futuro.
4. Tempo continuo o tempo discreto.

5.4.1 Logica temporale Lineare con numero infinito di stati (LTL)

Dominio temporale costituito da una sequenza infinita e discreta di *stati*, dove $\forall i \in \mathbb{N}$, S_i è in relazione S_{i+1} e si indica S_i R S_{i+1} .



La logica LTL definita dai seguenti **operatori temporali:**

1. L'operatore **Always** φ (detto anche *Globally* $\mathsf{G}\varphi$) indica che φ è vero d'ora in poi.
2. L'operatore **Eventually** $\Diamond\varphi$ (detto anche *Future* $\mathsf{F}\varphi$) indica che φ sarà vero prima o poi.
3. L'operatore **Next** $\mathsf{X}\varphi$ indica che φ sarà vero nell'istante successivo, da notare che il suo duale è ancora se stesso: $\mathsf{O}\varphi = \neg\mathsf{O}\neg\varphi$.
4. L'operatore **Until** $\varphi_1 \mathsf{U} \varphi_2$: esiste uno stato futuro in cui vale φ_2, e in tutti gli stati precedenti vale φ_1.
5. L'operatore **Precede** $\varphi_1 \mathsf{P} \varphi_2 = \neg(\neg\varphi_1 \mathsf{U} \varphi_2)$ sta ad indicare che φ_1 precede φ_2: non è possibile che esista uno stato futuro in cui vale φ_2, preceduto da stati in cui non vale φ_1.
6. L'operatore U**nless** $\varphi_1 \mathsf{W} \varphi_2 = (\varphi_1 \mathsf{U} \varphi_2) \vee \varphi_1$, sta ad indicare che φ_1 è sempre vera a meno che non diventi vera φ_2

Sintassi:

$$\varphi := \neg\ \varphi \mid \varphi \wedge \varphi \mid p \mid \varphi\ \mathsf{U}\ \varphi \mid \mathsf{O}\varphi$$

Attraverso l'operatore U è possibile definire l'operatore **eventually** ($\Diamond$) nel seguente modo: $\Diamond\varphi$ = true U φ.
Si può dire che :

- $\Diamond$ è l' operatore derivato da U.
- $\Box$ è l'operatore derivato da $\Diamond$, quindi da U infatti: $\varphi \equiv \neg\Diamond\neg\varphi$

Semantica:
Un modello LTL è una quintupla ξ = (S, S_0, R, V, P) dove:

- S è insieme infinito di stati,
- S_0 è lo stato iniziale ($S_0 \in$ S)
- R è la relazione : $\forall i \in \mathbb{N}$, S_i R S_{i+1}
- V : P$\times$S $\rightarrow$ {true, false}
- P sono le preposizioni atomiche.

Fissato ξ, la relazione di soddisfacibilità $S \models_\xi \varphi$ (ξ soddisfa φ in S) è definita come (per semplicità di notazione omettiamo ξ):

- $S \models p$, $p \in P$ se solo se V(p, S)=true.
- $S \models \neg\varphi$, se solo se $\neg S \models \varphi$
- $S \models \varphi_1 \wedge \varphi_2$, se solo se $(S \models \varphi_1) \wedge (S \models \varphi_2)$
- $S \models \Box\varphi$, se solo se $\overline{R}$ = chiusura transitiva di R, $\forall$S': S R $\overline{S'}$, S'$\models \varphi$
- $S \models \Diamond\varphi$, se solo se $\overline{R}$ = chiusura transitiva di R, $\exists$S': S R $\overline{S'}$, S'$\models \varphi$

Nel caso più specifico, per i singoli stati S_i:

- $S_i \models p$, $p \in P$ se solo se V(p, S)=true
- $S_i \models \neg\varphi$, se solo se $\neg S \models \varphi$
- $S_i \models \varphi_1 \wedge \varphi_2$, se solo se $(s \models \varphi_1) \wedge (S \models \varphi_2)$
- $S_i \models \Box\varphi$, se solo se $\forall k \geq i$: $S_k \models \varphi$
- $S_i \models \Diamond\varphi$, se solo se $\exists k \geq i : S_k \models \varphi$
- $S_i \models \mathsf{O}\varphi$, se solo se $S_{i+1} \models \varphi$
- $S_i \models \varphi_1 \mathsf{U} \varphi_2$, se solo se $\exists k \geq i : S_k \models \varphi_2$ e $\forall i \leq k' \leq k : S_{k'} \models \varphi_1$

Alcune tipiche formule LTL che esprimono proprietà interessanti possono essere le seguenti:

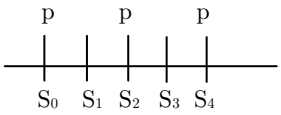
- $y > n+1$, **proprietà Invariante** (è sempre vero).
- $\Diamond p$, **proprietà *Liveness*** (prima o poi succede qualcosa).
- $\neg$bad, **proprietà *Safety*** (non è mai vero che qualcosa è scorretto).
- (request $\Rightarrow \Diamond$reply), **proprietà *Liveness*** (se faccio una richiesta prima o poi risponde).
- $\Diamond p$ (GFp), **proprietà *Infinitely often*** (p è vera infinitamente spesso).
- $\Diamond\ p$ (FGp), **proprietà *Eventually always*** (prima o poi arrivo in uno stato tale che da li in poi risulta sempre vero p).

L' **infinitely often** è usato per definire la proprietà di *Fairness:*

$$\Diamond \text{request}_k \Rightarrow \quad \Diamond \text{replay}_k$$

se richiedo infinitamente spesso, infinitamente spesso ottengo risposte.

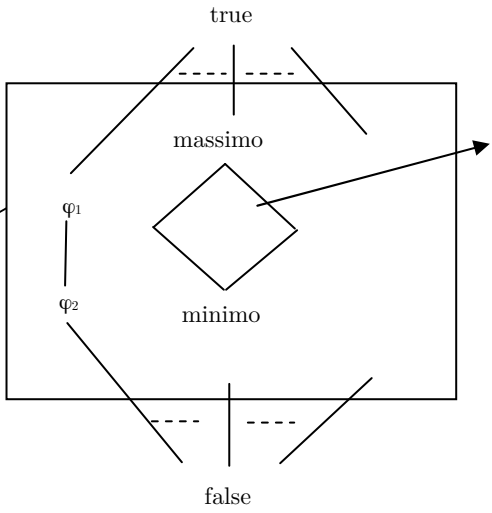
Esempio: p è vera in ogni stato pari



Formule che esprimono la condizione:

- $(p \wedge OOp)$ non va bene, perché vale anche per i dispari.
- $(p \wedge O\neg p \wedge OOp)$ non va bene, è sempre falsa.
- $X = p \wedge OOX$ questa formula va intuitivamente bene, ma non è una formula della logica, è un'equazione sulla logica, a cui bisogna dare soluzione.

Definizione: Se x = f(x) allora x è un **Punto Fisso** di f.



Sottoreticolo: insieme delle formule che sono punti fissi di f, che ha un massimo e un min.

Reticolo: insieme di tutte le formule φ

L' ordinamento dell'implicazione $(\varphi_1 \Rightarrow \varphi_2)$ costituisce un reticolo, in cui ho:

- $\text{false} \Rightarrow \varphi \quad \forall\varphi$ (minimo del reticolo)
- $\varphi \Rightarrow \text{true} \quad \forall\varphi$ (massimo del reticolo)

Definiamo:

1. Il minimo punto fisso all'interno del sottoreticolo è $\mu x \cdot f(x)$, vale che:
$$\mu x \cdot f(x) = \bigvee_{i=0}^{\infty} \cdot f^i(\text{false})$$

2. Il massimo punto fisso all'interno del sottoreticolo è $\nu x \cdot f(x)$, vale che:
$$\nu x \cdot f(x) = \bigwedge_{i=0}^{\infty} \cdot f^i(\text{true})$$

Calcoliamo min e max per la formula $x = p \wedge \mathsf{OO}x$
Ricordando che $x = f(x) = p \wedge \mathsf{OO}x$ quando x è un punto fisso di f

- $f^0(x) = p \wedge \mathsf{OO}x$
- $f^1(x) = p \wedge \mathsf{OO}f^0(x)$
- $f^2(x) = p \wedge \mathsf{OO}f^1(x)$
- :

Calcoliamo il minimo: $\mu x \cdot f(x)$:

- $x_0 = p \wedge \mathsf{OO}\text{false} = \text{false}$
- $x_1 = p \wedge \mathsf{OO}x_0 = \text{false}$
- $x_2 = p \wedge \mathsf{OO}x_1 = \text{false}$
- :

Andando avanti otteniamo sempre false, quindi si può concludere dicendo:

$$\mu x \cdot f(x) = \mu x \cdot p \wedge \mathsf{OO}x = \text{false}$$

Calcoliamo il massimo: $\nu x \cdot f(x)$:

- $x_0 = p \wedge \mathsf{OO}\text{true} =$ (dipende da p)
- $x_1 = p \wedge \mathsf{OO}(p \wedge \mathsf{OO}\text{true}) =$ (dipende da p)
- $x_2 = p \wedge \mathsf{OO}(p \wedge \mathsf{OO}(p \wedge \mathsf{OO}\text{true})) =$ (dipende da p)
- :

continuando cresce all'infinito; la formula che vogliamo è $\nu x \cdot p \wedge \mathsf{OO}x$ (massimo punto fisso) è una formula infinita, cioè non rappresentabile in modo finito con gli operatori della logica.

Definizione: $\mu x \cdot f(x) = \neg \nu x \neg f(x)$.

Possiamo avere una **logica temporale lineare con punto fisso** definito solo attraverso U,O e $\mu x \cdot f(x)$, oppure solo con O e $\mu x \cdot f(x)$ perché da questi due posso derivare U.

Esercizio: Vogliamo dimostrare la seguente uguaglianza :

$$\mu x\ \phi_2 \vee (\phi_2 \wedge \mathsf{O}x) = \phi_1\, \mathsf{U}\, \phi_2$$

dove $\phi_2 \vee (\phi_2 \wedge \mathsf{O}x) = f(x)$

$x_0 = \phi_2 \vee (\phi_1 \wedge \mathsf{O}\text{false}) = \phi_2 \longrightarrow$ ϕ_2 — È vera su ϕ_2

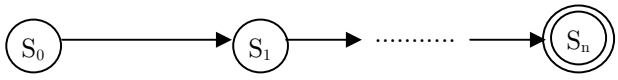
$x_1 = \phi_2 \vee (\phi_1 \wedge \mathsf{O}\phi_2) = \longrightarrow$ ϕ_2 — È vera su ϕ_2; $\phi_1\ \phi_2$ — È vera su ϕ_1 e al prossimo stato ϕ_2 è vera

$x_2 = \phi_2 \vee (\phi_1 \wedge \mathsf{O}(\phi_2 \vee (\phi_1 \wedge \mathsf{O}\phi_1))) = \longrightarrow$ ϕ_2; $\phi_1\ \phi_2$; $\phi_1\ \phi_1\ \phi_2$

Come si evince si ha sempre che $\phi_1\, \mathsf{U}\, \phi_2$

5.4.2 Logica temporale Lineare con numero finito di stati

Il dominio temporale costituito da una sequenza finita di *stati*, dove $\forall i \in \mathbb{N}$, S_i è in relazione S_{i+1} e si indica $S_i \, R \, S_{i+1}$.



Appare subito ovvio che non vale più che il duale dell'operatore O è ancora se stesso $\mathsf{O}p = \neg\mathsf{O}\neg p$, questo perché se valesse (usiamo provvisoriamente i simboli ' e ◊' per denotare il next e il suo duale):

- 'p $\forall$ stato successore vale p.
- ◊'φ $\exists$ stato successore per cui vale p.

Per la sequenza finita vale :

- $S_n \models$ 'p, $\forall$p (debole): non raggiungo nessun nodo in cui è vero quindi è soddisfatta.
- $S_n \not\models$ ◊'φ, $\exists$p (forte): non esiste alcun nodo raggiungibile in cui è vero quindi è soddisfatto.

Introduciamo due operatori NEXT: Forte O e DEBOLE $\mathsf{\odot}$

$\mathsf{O}\varphi = \neg\mathsf{\odot}\neg\varphi$ si può indicare anche nel seguente modo $\mathsf{X}\varphi = \neg\, \widetilde{\mathsf{X}}\, \neg\varphi$
Per un generico stato S_i ha che :

- $S_i \models \mathsf{O}\varphi$ se solo se $i < n$, $S_{i+1} \models \varphi$
- $S_i \models \mathsf{\odot}\varphi$ se solo se $(i < n, S_{i+1} \models \varphi) \vee (i = n)$

Inoltre:

- $\mathsf{\odot}$false = true all'ultimo stato.
- Ofalse = false.
- $S \models \mathsf{\odot}$false $\Leftrightarrow S = S_n$ (stato finale).

5.4.3 Logica temporale con Tempo passato

Consideriamo il caso di non avere un tempo iniziale, cioè ammettiamo di avere un passato, abbiamo nuovi operatori simmetrici a quelli appena visti:

- ■ p tutti gli stati precedenti è vera p.
- ⧫ p esiste uno stato precedente in cui è vera p.
- ● p nello stato precedente p è vera (PAST).

Introduciamo l'operatore **Since** S:

- φ_1 S φ_2 : dal momento in cui è stato vero φ_2 è vero φ_1 . Si evince che il Since è il corrispettivo al passato dell'Until.

Ai fini pratici, si può spesso esprimere la logica del passato attraverso la logica del presente.
Esempio: se accade p, allora è accaduto q si può esprimere come:

$$p \Rightarrow \blacklozenge q, \text{ dove } p \Rightarrow \blacklozenge q \text{ è equivalente a } \neg(\neg q \ \mathsf{U} \ p) \text{ cioè } q \ \mathsf{P} \ p.$$

5.4.4 Logica temporale Continua e tempo reale

Il tempo si è considerato finora discreto; nel caso di tempo continuo si perde il concetto di tempo prossimo (NEXT), possiamo solo dire che un tempo è maggiore dell'altro; si mantengono gli operatori ◊.

Inoltre vale che t R t' $\Leftrightarrow$ t $\leq$ t'

Questa logica è interessante quando si vuole quantificare puntualmente il tempo, cioè indicare nelle formule precisi istanti o intervalli di tempo. E' evidente la relazione di questo aspetto con i sistemi "real-time".

Nel discreto esprimere sotto forma di formula la seguente operazione: "se viene fatta una richiesta la risposta viene data dopo 3 unità di tempo " può essere fatto nel seguente modo:

$$(\text{request} \Rightarrow \mathsf{OOO}\text{replay})$$

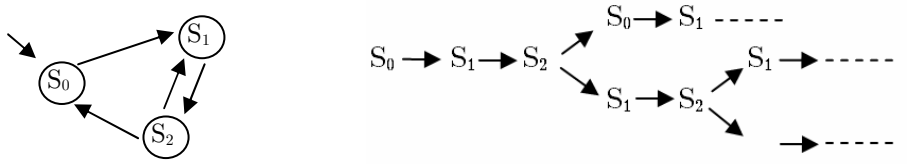
Mentre per un operazione del tipo “se viene fatta una richiesta la risposta viene data entro 3 unità di tempo”, si esprime nel seguente modo:

$$(\text{request} \Rightarrow \mathsf{O}\text{replay} \vee \mathsf{OO}\text{replay} \vee \mathsf{OOO}\text{replay})$$

E’ ovvio che voler indicare un istante di tempo preciso, diciamo t, richiede una specifica formula con t operatori di next, perdendo quindi di generalità. Sono state proposte varie logiche come la ITL (interval logic) e la RTL (real time temporal logic) che usano appositi operatori per indicare intervalli o istanti di tempo. Per esempio nella ITL l’operatore eventually si indica nel seguente modo $\Diamond_{(3\text{-}5)}$ p e sta a significare che p è vera nell’intervallo 3-5.
Questa classe di logiche temporali può essere definita sia in un dominio temporale discreto che continuo.

5.4.5 Logica temporale Ramificata (CTL)

Partendo da una macchina a stati posso ottenere un albero (una struttura ramificata), attraverso l’operazione di Unfolding “srotolamento” della macchina a stati:



Con la logica CTL è possibile esprimere formule logiche su cammini (ossia sequenze di transizioni di stato) a partire da un determinato stato iniziale. La caratteristica principale di questa logica è che ogni formula deve essere premessa da un quantificatore di cammino, per cui ogni formula viene sempre riferita all’insieme dei cammini a partire da uno stato iniziale. La logica CTL (*computation tree logic*) è definita dagli operatori temporali Xp (next), che significa che una certa proprietà p si verifica nell’istante successivo, Fp eventually, p si verifica in futuro, Gp always, p è sempre verificata, p U q (until) e p P q (precede). Questi operatori non possono però essere utilizzati liberamente: essi devono essere sempre prefissi da un quantificatore di cammino, si indica con A il **quantificatore universale** (per tutti i cammini) e con E il **quantificatore esistenziale** (esiste almeno un cammino).

Sintassi:
La logica CTL è definita per mezzo di due categorie sintattiche:

1. Formule di Stato (ϕ).
2. Formule di Cammino (ψ).

Formule di Stato:
Dato p $\in$ AP (insieme di proposizioni atomiche) si ha che:
$\phi := \neg\phi \mid \phi \wedge \phi \mid \text{true} \mid p \mid \mathsf{A}\psi \mid \mathsf{E}\psi$

Formule di Cammino
$\psi := \mathsf{X}\phi \mid \phi\ \mathsf{U}\ \phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi$

Esempio: true $\mathsf{U}\ \phi$ è una formula di cammino e vuol dire che prima o poi vale ϕ, questo non è altro che Eventually $\Rightarrow$ true $\mathsf{U}\ \phi = \mathsf{F}\phi$

Semantica:
Una **struttura di Kripke** M su un insieme di proposizioni atomiche AP è una 4−upla M = (S, S_0, $\rightarrow$, L) dove:

1. S è un insieme finito di stati.
2. $S_0 \in S$ è l'insieme degli stati iniziali.
3. $\rightarrow \subseteq S \times S$ è la relazione di raggiungibilità.
4. L : S $\rightarrow 2^{AP}$ è una funzione che assegna ad ogni stato un'etichetta che contiene le proposizioni atomiche vere in quello stato.

Semantica per le Formule di Stato:

- S $\models$ true sempre.
- S $\models$ p, se solo se p $\in$ L(S).
- S $\models \phi_1 \wedge \phi_2 \Leftrightarrow (S \models \phi_1) \wedge (S \models \phi_2)$
- S $\models \neg\phi \Leftrightarrow \neg S \models \phi$
- S $\models \mathsf{A}\psi \Leftrightarrow \forall\pi \in$ Path(S), $\pi \models \psi$ dove π è un cammino e Path(S) restituisce tutti i cammini a partire da S
- S $\models \mathsf{E}\psi \Leftrightarrow \exists\pi \in$ Path(S), $\pi \models \psi$

Semantica per le Formule di Cammino:

- $\pi \models \mathsf{X}\phi \Leftrightarrow \pi = S_0, S_1, S_2, \ldots, S_n, \ldots \wedge S_1 \models \phi$
- $\pi \models \phi_1\ \mathsf{U}\ \phi_2 \Leftrightarrow \pi = S_0, S_1, S_2, \ldots, S_n, \ldots \wedge \exists i : S_i \models \phi_2 \wedge \forall k < i\ S_k \models \psi$
- $\pi \models \mathsf{F}\phi \Leftrightarrow \pi = S_0, S_1, S_2, \ldots, S_n, \ldots \exists i : S_i \models \phi$
- $\pi \models \mathsf{G}\phi \Leftrightarrow \pi = S_0, S_1, S_2, \ldots, S_n, \ldots \forall i : S_i \models \phi$

Ad esempio consideriamo le seguenti formule:

- EF(reply): esiste un cammino in cui ho una risposta.
- AF(reply): per ogni cammino ho una risposta.

Alcune tipiche formule CTL che esprimono proprietà interessanti possono essere le seguenti:

- $\neg\mathsf{AF}\neg$(reply) = EF(reply)
- $\neg\mathsf{EF}\neg$(reply) = AF(reply)
- $\neg\mathsf{EX}\neg$(reply) = AX(reply)
- AG(request $\Rightarrow$ AF (reply)): tutte le volte che c'è una richiesta il sistema risponde
- AG(request $\Rightarrow \mathsf{EF}$(reply)) almeno in un caso il sistema risponde.
- $\neg\mathsf{E}[\neg$p U q] p precede q
- $\mathsf{AG}(\mathsf{AF}\varphi)$: φ vale infinitamente spesso su ogni cammino
- $\mathsf{AG}(\mathsf{EF}\varphi)$: da ogni stato è possibile raggiungere lo stato φ

Come si può notare il *not* si applica solo alle formule di stato, che consentono la dualità.

5.4.6 CTL*

Come detto precedentemente la logica Temporale si divide in:

- **logica temporale lineare** (LTL), dove gli operatori sono forniti per la descrizione di eventi lungo un unico percorso.
- **logica temporale ramificata** (CTL) dove gli operatori temporali quantificano i percorsi che sono possibili da un dato stato.

La computation tree logic CTL* combina la logica temporale lineare e la logica temporale ramificata, in pratica rimuove la distinzione tra formula di stato e formula di cammino. Si può affermare che CTL* comprende sia la CTL che la LTL.

In questa logica un quantificatore di cammino può essere messo come prefisso in una asserzione composta da combinazioni arbitrarie di operatori temporali lineari

1. Quantificatore di cammino
   - A quantificatore universale “per tutti i cammini”
   - E quantificatore esistenziale “esiste almeno un cammino”.

2. Operatori temporali lineari
   - Xp (next)
   - Fp (Future)
   - Gp (Globally)
   - p U q (Until)

Sintassi:
La sintassi delle **formule di stato** è data dalle seguenti regole:

- Se p $\in$ AP, allora p è una formula di stato.
- Se f e g sono formule di stato, allora ($\neg$f) e (f $\vee$ g) sono formule di stato.
- Se f è una formula di cammino, allora E(f) è una formula di stato.

Due regole necessarie per specificare la sintassi delle **formule di cammino** sono:

- Se f è una formula di stato, allora f è anche una formula di cammino
- Se f e g sono formule di cammino, allora $\neg$f, (f $\vee$ g), Xf, e (f U g) sono formule di cammino

Semantica Formule di Stato:
Se f è una formula di stato, la notazione M,s |= f significa che f vale nello stato s nella struttura di Kripke M.

Assumiamo f_1 e f_2 siano formule di stato e g una formula di cammino. La relazione M,s |= f (omettiamo M) è definita induttivamente come segue :

- s |= true sempre.
- s |= p $\Leftrightarrow$ p $\in$ L(S).
- s |= $f_1 \wedge f_2 \Leftrightarrow$ (S |= f_1) $\wedge$ (S |= f_2).
- s |= $\neg f_1$ $\Leftrightarrow \neg$S |= f_1.
- s |= $\mathsf{A}(\psi)$ $\Leftrightarrow \forall \pi \in$ Path(S) π |= ψ dove π è un cammino e Path(S) restituisce tutti i cammini a partire da S.
- s |= $\mathsf{E}(\psi)$ $\Leftrightarrow \exists \pi \in$ Path(S) π |= ψ.

Semantica Formule di Cammino:
Se f è una formula di cammino, la notazione M, π |= f significa che f è valida lungo il cammino π nella struttura di Kripke M.

Assumiamo g_1 e g_2 siano formule di cammino e f una formula di stato. La relazione M, π|= f è definita induttivamente come segue :

- π |= f $\Leftrightarrow$ s è il primo stato di π e s|= f
- π |= $\neg g_1$ $\Leftrightarrow \neg \pi$ |= g_1
- π |= $g_1 \vee g_2 \Leftrightarrow$ (S |= g_1) o (S |= g_2)
- π |= $\mathsf{X}\ g_1$ $\Leftrightarrow \pi^1$ |= g_1
- π |= $g_1\ \mathsf{U}\ g_2 \Leftrightarrow$ esiste un $k \geq 0 : \pi^k$ |= g_2 e for $0 \leq j < k$, π^j |= g_1

CTL è un sottoinsieme limitato di CTL* che autorizza solo gli operatori di logica temporale ramificata, ciascuno degli operatori tempo lineari G, M, X e U deve essere immediatamente preceduta da un quantificatore di cammino (A, E).
Esempio: AF(EFp)

LTL consiste di formule Af se e solo se f è una "path formula" nella quale le sole "state formula" ammesse sono le proposizioni atomiche.
Esempio: A(FGp)

s |= f	se e solo se	s |= Af
LTL semantics		CTL* semantics

5.5 Comparazione tra LTL, CTL e CTL*

Le tre logiche LTL, CTL, CTL* hanno potere espressivo differente, per esempio:

- In CTL non esiste un equivalente alla formula LTL $\mathsf{A}(\mathrm{FG}\varphi)$ e $\mathsf{A}(\mathrm{GF}\varphi)$.
- In LTL non esiste un equivalente alla formula CTL $\mathsf{AG}(\mathsf{EF}\varphi)$.
- La disgiunzione $\mathsf{A}(\mathsf{FG}\mathrm{p}) \vee \mathsf{AG}(\mathsf{EF}\mathrm{p})$ è una formula CTL* che non si può esprimere né in LTL né in CTL.

Esempio: La proprietà FAIRNESS è esprimibile in LTL ma non in CTL perché non esiste una formula equivalente in CTL della formula $\mathsf{GF}\mathrm{p}$ (LTL).

La fairness in LTL è:
$\mathsf{GF}\mathrm{p} \Rightarrow \mathsf{GF}\mathrm{q}$: se è vero infinitamente spesso p, allora è vero infinitamente spesso q.

7. Automi di Büchi & Model Checking LTL

7.1 Problema del Model Checking

Dato un modello M, uno stato s ed una proprietà espressa con una formula LTL ϕ, determinare se

$$M, s \models \phi$$

$$\underset{\text{Decidibile}}{\text{Soddisfacibilità formule LTL}} \quad \Rightarrow \quad \underset{\text{decidibile}}{\text{Model Checking per LTL}}$$

Il Model Checking LTL è basato su una variante degli automi a stati finiti, chiamati Automi di Büchi

7.2 Automa a stati finiti

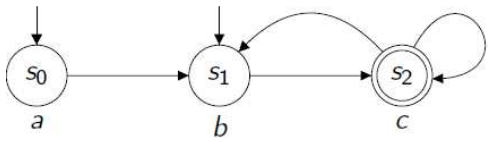
Definizione: Un Automa a stati finiti etichettato (LFSA) è una tupla $(\Sigma, S, S^0, \rho, F, L)$:

- Σ è un alfabeto di simboli.
- S è un insieme finito di stati.
- $S^0 \subseteq S$ è l'insieme di stati iniziali.
- $\rho : S \to S$ è la funzione di transizione di stato.
- $F \subseteq S$ è l'insieme di stati finali.
- $L : S \to \Sigma$ è la funzione di etichettatura degli stati.

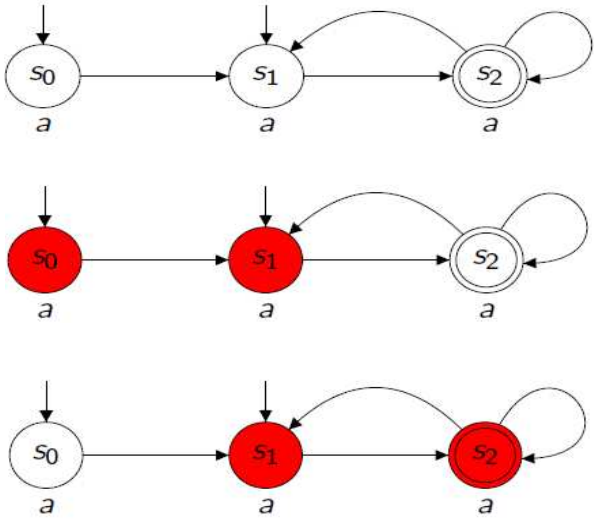
Definizione: Un LFSA è deterministico sse

1. $|\{s \in S^0 \mid L(s) = a\}| \leq 1 \qquad \forall a \in \Sigma$
2. $|\{s' \in \rho\ (s) \mid L(s') = a\}| \leq 1 \qquad \forall a \in \Sigma, \forall s \in S$

Esempi determinismo



Esempi di non determinismo



7.2.1 Linguaggio accettato

Definizione: Un run per un LFSA è una sequenza finita $\sigma = s_0\, s_1 \ldots s_n$ tale che $s_0 \in S^0$ e $s_i \to s_{i+1}$ $\forall 0 \leq i < n$

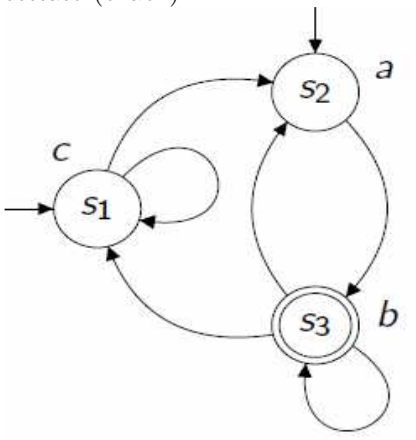
Definizione: Un run è detto ***accepting*** se $s_n \in F$

Definizione: Una parola $w = a_0\, a_1 \ldots a_n$ è accettata sse esiste un accepting run $\sigma = s_0\, s_1 \ldots s_n : L(s_i) = a_i$ $\forall 0 \leq i < n$ $(a_i \in \Sigma)$

Definizione: Se si indica con Σ^* l'insieme di tutte le possibili parole finite su Σ, il linguaggio accettato dall'LFSA A è:

$$\mathcal{L}(A) = \{\ w \in \Sigma^* \mid w \text{ è accettata da } A\ \}$$

Esempio linguaggio accettato $(c^*ab^+)^+$:



7.3 Automa di Büchi

Definizione di Automa di Büchi: Un Automa di Büchi etichettato (LBA) è un LFSA che accetta parole di lunghezza infinita invece che di lunghezza finita

Definizione: Un run per un LBA è una sequenza infinita $\sigma = s_0\ s_1\ \ldots$ tale che $s_0 \in S^0$ e $s_i \rightarrow s_{i+1}\ \forall i \geq 0$

Definizione: Sia $\lim(\sigma)$ l'insieme di stati che occorre in σ infinitamente spesso. Allora σ è un accepting run sse $\lim(\sigma) \cap F \neq \emptyset$

Poiché il numero di stati dell'automa è finito, mentre la sequenza σ ha lunghezza infinita, sicuramente $\lim(\sigma) \neq \emptyset$

7.3.1 Linguaggio accettato

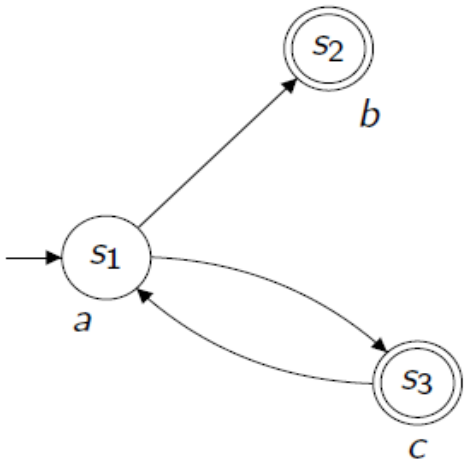
Definizione: Una parola $w = a_0\ a_1\ \ldots$ è accettata da un LBA sse esiste un accepting run

$$\sigma = s_0\ s_1\ \ldots \text{ tale che } L(s_i) = ai\ \forall i \geq 0\ (a_i \in \Sigma)$$

Definizione: Se si indica con Σ^ω l'insieme di tutte le possibili parole infinite su Σ, il linguaggio accettato dall'LBA A è:

$$\mathcal{L}_\omega(A) = \{\ w \in \Sigma^\omega \mid w \text{ è accettata da A }\}$$

Esempio: Linguaggio accettato $(a\ c)^{\omega}$



7.3.2 Equivalenza ed Espressività

Definizione: Due automi A_1 e A_2 sono equivalenti se accettano lo stesso linguaggio. Valgono in generale le seguenti considerazioni:

- $\mathcal{L}(A_1) = \mathcal{L}(A_2) \not\Rightarrow \mathcal{L}_{\omega}(A_1) = \mathcal{L}_{\omega}(A_2)$
- $\mathcal{L}_{\omega}(A_1) = \mathcal{L}_{\omega}(A_2) \not\Rightarrow \mathcal{L}(A_1) = \mathcal{L}(A_2)$

. . .

inoltre

- A LFSA non deterministico $\Rightarrow$ esiste A′ LFSA deterministico tale che $\mathcal{L}(A) = \mathcal{L}(A')$
- A LBA non deterministico $\not\Rightarrow$ esiste A′ LBA deterministico tale che $\mathcal{L}_{\omega}(A) = \mathcal{L}_{\omega}(A')$

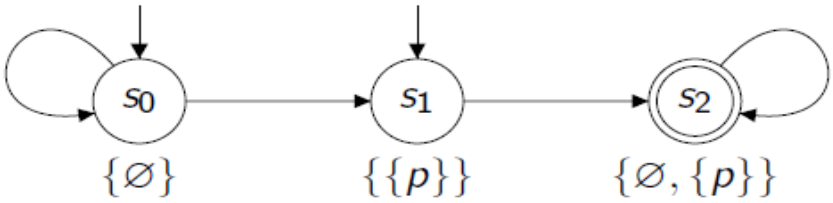
7.4 Model Checking LTL

7.4.1 Idee alla base del Model Checking LTL

- Modifichiamo la funzione di etichettatura degli stati in modo da considerare insiemi di simboli $L : S \to 2^{\Sigma}$
- Una parola $w = a_0\, a_1 \ldots$ è accettata da un LBA sse esiste un accepting run $\sigma = s_0\, s_1 \ldots$ tale che $a_i \in L(s_i)\ \forall\ i \geq 0$
- Consideriamo $\Sigma = 2^{AP}$, così gli stati saranno etichettati con insiemi di insiemi di proposizioni atomiche.

Si vuole associare ad ogni formula LTL ϕ un LBA il cui linguaggio accettato corrisponda alle sequenze di proposizioni atomiche che rendono valida ϕ

Esempio: corrispondente a Fp



7.4.2 Codifica delle formule proposizionali

Gli insiemi di insiemi di proposizioni atomiche codificano le formule proposizionali:

- Se $AP_1, \ldots, AP_n \subseteq AP$, allora ogni AP_i codifica la formula

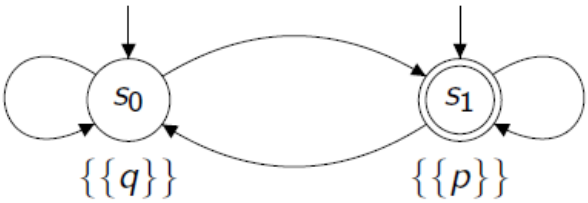
$$[\![AP_i]\!] = (\forall p \in AP_i \ : \ p) \wedge (\forall p \in AP - AP_i \ : \ \neg p)$$

- L'insieme $\{AP_{k_1}, \cdots, AP_{k_m}\}$, per $m \geq 1$ e $0 < k_j \leq n$ codifica

$$[\![AP_{k_1}]\!] \vee \ldots \vee [\![AP_{k_m}]\!]$$

Gli insiemi di insiemi di proposizioni atomiche non possono essere vuoti.

Esempio: codifica di una formula



AP = {p, q}
s_0: (q ∧ ¬p)
s_1: (p ∧ ¬q)

Formula LTL:
G[(q ∧ ¬p) U (p ∧ ¬q)]

Stati → formule proposizionali

Transizioni → comportamento temporale

} = Formule LTL

7.4.3 Model Checking

Model Checking (primo approccio)
Il metodo più semplice è quello di verificare che tutti i comportamenti del sistema siano desiderabili:

1. Costruire l'LBA per $\phi = A_\phi$
2. Costruire l'LBA per il modello del sistema $\Rightarrow A_{sys}$
3. Verificare se $\mathcal{L}_\omega (A_{sys}) \subseteq \mathcal{L}_\omega (A_\phi)$

Ma decidere l'inclusione tra i linguaggi accettati è un problema NP

Model Checking (secondo approccio)
Il problema di verificare l'inclusione tra linguaggi può essere aggirato poiché

$$\mathcal{L}_\omega (A_{sys}) \subseteq \mathcal{L}_\omega (A_\phi) \Leftrightarrow \mathcal{L}_\omega (A_{sys}) \cap \mathcal{L}_\omega (\overline{A_\phi}) = \varnothing$$

$\overline{A_\phi}$ è l'LBA complementare di A_ϕ e accetta il linguaggio $\Sigma^\omega \setminus \mathcal{L}_\omega (A_\phi)$

In generale la costruzione di $\overline{A_\phi}$ è quadraticamente esponenziale.

$$A_\phi \text{ ha n stati} \Rightarrow \overline{A_\phi} \text{ ha } c^{n^2} \text{ stati } (c > 1)$$

Model Checking (terzo approccio)
Infine, osservando che $\mathcal{L}_\omega (A_\phi) = \mathcal{L}_\omega (\overline{A_\phi})$ si arriva ad un metodo efficiente di model checking:

1. Costruire l'LBA per $\neg\phi = A_{\neg\phi}$
2. Costruire l'LBA per il modello del sistema $= A_{sys}$
3. Verificare se $\mathcal{L}(A_{sys}) \cap \mathcal{L}_\omega (A_{\neg\phi}) = \varnothing$

(i run in comune tra A_{sys} e $A_{\neg\phi}$ violano ϕ, quindi sono indesiderati)

Il terzo passo dell'algoritmo è decidibile in tempo lineare.

7.4.4 Complessità temporale Model Checking LTL

Se definiamo con S_{sys} l'insieme degli stati dell'LBA A_{sys} , si ha che nel caso peggiore la complessità temporale del model checking LTL è :

$$O\ (|S_{sys}\ |^2 \times 2^{|\Phi|})$$

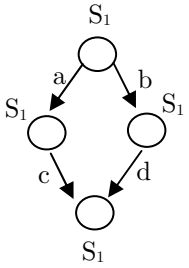
(il fattore $2^{|\Phi|}$ è legato alla costruzione del grafo)

Anche se la complessità è esponenziale nella lunghezza della formula LTL, nella pratica le formule sono abbastanza corte (max 2 o 3 operatori)

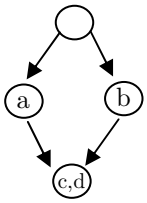
8. Algebre di Processi

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

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



LTS



KRIPKE

8.1 Labelled Transition System (LTS)

I sistemi di transizioni etichettate furono introdotti da *Keller* nel 1976 come un modello formale per descrivere programmi paralleli ed in seguito sono stati usati per dare una semantica operazionale strutturale ai linguaggi di programmazione. I sistemi di transizione sono quindi un modello relazionale astratto basato sulle nozioni primitive di stato e transizione. Molte proprietà dei sistemi concorrenti possono essere studiate tramite questa rappresentazione; bisogna essere in grado di conoscere la capacità di tali sistemi di compiere azioni appartenenti ad un insieme predeterminato Act, azioni che possono essere istantanee o durature. Ovviamente un sistema sequenziale può effettuare al più un'azione nello stesso istante, certe azioni però possono essere azioni di sincronizzazione di un processo con il sistema concorrente di cui fa parte oppure segnali inviati dal resto del sistema al processo; è ovvio che quest'ultimo tipo di azioni occorrono solamente nel caso in cui i processi cooperino. Per essere più precisi forniamo una definizione formale di LTS.

Definizione: Un LTS è una quadrupla $(S, s_0, Act, \rightarrow)$ dove:

- S è un insieme di stati;
- s_0 è lo stato iniziale;
- Act è un insieme finito e non vuoto di azioni visibili;
- $\rightarrow \subseteq S \times Act \times S$ è la relazione di transizione tale che un'elemento $(s_0, a, s_1) \in \rightarrow$ se esiste la possibilità di passare da uno stato s_0 ad un altro s_1 tramite l'azione a $(s_0 \xrightarrow{a} s_1)$.

Un LTS può essere rappresentato tramite un grafo il cui nodo iniziale è lo stato iniziale s_0, le relazioni di transizione sono rappresentate dagli archi fra i nodi (archi etichettati con la azioni appartenenti ad Act), i nodi infine rappresentano gli stati appartenenti a S.

Definizione: Negli LTS, un cammino (o computazione) è una sequenza $(s_0, \alpha_0, s_1)\ (s_1, \alpha_1, s_2)\ (s_2, \alpha_2, s_3)\ \ldots$ dove ogni tripla $(s_i, \alpha_i, s_{i+1}) \in \rightarrow$
Un cammino può essere finito o infinito (in base al numero di transizioni che contiene) ed eventualmente può avere dei cicli al suo interno; possiamo inoltre trovare cammini nulli rappresentati da una sequenza vuota.

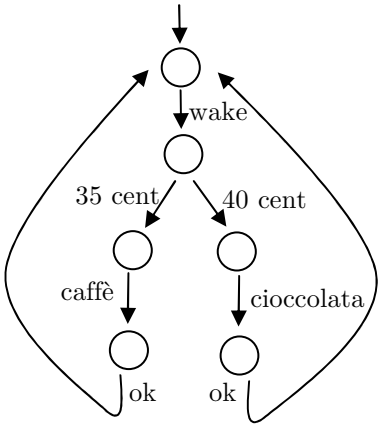
Esempio: Modellazione semplificata ad eventi della macchina del caffè di ingegneria.

Questa macchina permette di eseguire sequenze del tipo :

wake-35-caffè-ok-wake-40-cioccolata

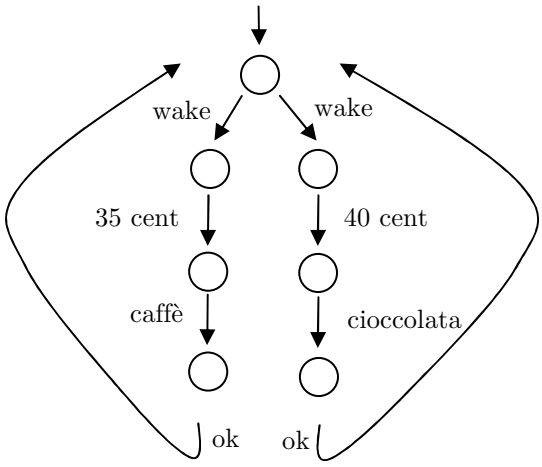
ma non:

wake-35-cioccolata



Con un modello del genere posso verificare il comportamento della macchina a fronte di sequenze ammesse e non ammesse. L'implementazione dovrà contenere tutte e sole le tracce definite nel modello

Supponiamo di avere la seguente implementazione:



L'implementazione contiene tutte le tracce del modello (*trace equivalence*) ma ha un non determinismo sullo stato iniziale infatti è la macchina e non l'utente a "decidere" se verrà presa una cioccolata o del caffè.
Da notare che la trace equivalence tra i due LTS non cattura la ramificazione: dunque anche se due macchine contengono le stesse tracce non è detto che siano capaci di soddisfare i test.

Uno dei vantaggi dell'LTS sta nella possibilità di costruire su di esso una **bisimulazione**.

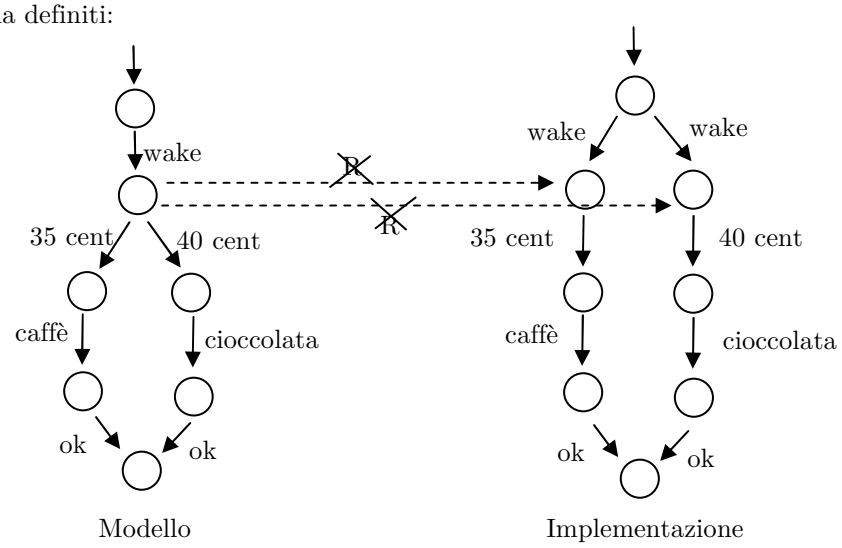
Definizione: R è una relazione di bisimulazione forte su S, S' si indica con S R S' (S e S' sono insieme degli stati dove $s_i \in S$ e $s_i' \in S'$) se $\forall s_i \in S$ e $\forall s_i' \in S'$:

$$SRS' \Leftrightarrow \begin{cases} s \xrightarrow{a} s_1 \Rightarrow \exists s_1' \in S' : s' \xrightarrow{a} s_1' \wedge\ s_1 R\ s_1' \\ s' \xrightarrow{a} s_1' \Rightarrow \exists s_1 \in S : s \xrightarrow{a} s_1 \wedge s_1 R\ s_1' \end{cases}$$

Definizione: $s \in S$ e $s' \in S'$ sono equivalenti ($s \sim s'$) se $\exists$ R di bisimulazione forte tra s e s'.

Definizione: Due LTS ($S, s_o, Act, \rightarrow$) e ($S', s_o', Act', \rightarrow'$) si dicono equivalenti (o bisimili) se $\exists$ R di bisimulazione forte tra S e S': $s_0 R\ s_0'$

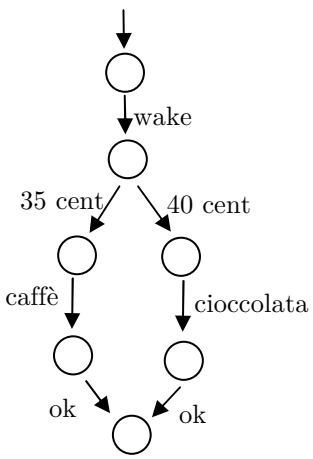
Valutiamo la relazione di bisimulazione sul modello e sull'implementazione prima definiti:



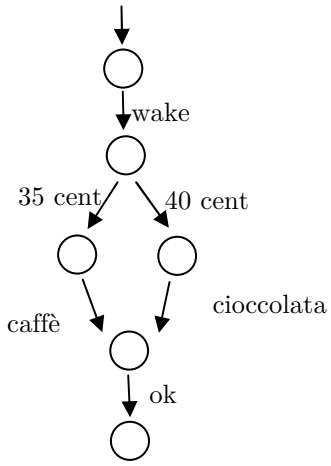
Modello Implementazione

In questo caso il modello ed l'implementazione non sono equivalenti per la bisimulazione.

Supponiamo invece di avere un'altra implementazione:



Modello



Implementazione

In questo caso il modello ed l'implementazione sono equivalenti per le tracce e per la bisimulazione.

Un'eventualità è che nell'implementazione vengono fatte azioni non osservabili, indicate con τ. Si ridefinisce LTS nel seguente modo:

$$(S, s_o, Act \cup \{\tau\}, \rightarrow)$$

Dove in questo caso $\rightarrow \subseteq S \times Act \cup \{\tau\} \times S$

Definizione: R è una relazione di bisimulazione debole su S, S' si indica con S R S' (S e S' sono insieme degli stati dove $s_i \in S$ e $s_i' \in S'$) se $\forall s_i \in S$ e $\forall s_i' \in S'$:

$$SRS' \Leftrightarrow \begin{cases} s \overset{a}{\Rightarrow} s_1 \Rightarrow \exists s_1' \in S' : s' \overset{a}{\Rightarrow} s_1' \wedge s_1 R\, s_1' \\ s' \overset{a}{\Rightarrow} s_1' \Rightarrow \exists s_1 \in S : s \overset{a}{\Rightarrow} s_1 \wedge s_1 R\, s_1' \end{cases}$$

(dove $s \overset{a}{\Rightarrow} s'$ se solo se $\exists$ $s_1, s_2, \ldots, s_n$ tale che $s \overset{\tau}{\rightarrow} s_1 \overset{\tau}{\rightarrow} s_2 \overset{\tau}{\rightarrow} \cdots \overset{\tau}{\rightarrow} s_n \overset{a}{\rightarrow} s'$).

Definizione: $s \in S$ e $s' \in S'$ sono observation equivalent ($s \approx s'$) se $\exists$ R di bisimulazione debole tra s e s'.

Definizione: Due LTS (S, s_o, Act, $\rightarrow$) e (S', s_o', Act', $\rightarrow$') sono observation equivalent se $\exists$ R di bisimulazione debole tra S e S': s_0 R s_0'

Oltre all'equivalenza esiste il **preordine**, usato nel raffinamento dei modelli.
M_0 è il modello iniziale che viene via via raffinato nella versione M_1 M_2 …, ottenendo una sequenza:

$$M_0 \subseteq M_1 \subseteq M_2 \subseteq M_3 \ldots\ldots.$$

8.2 Il Calcolo CCS

In questo paragrafo introduciamo il CCS, cioè una delle algebre di processi più conosciute ed utilizzate nella teoria della concorrenza. Il Calcolo dei Sistemi di Comunicazione (*Calculus of Communicating Systems*) fu introdotto nel 1980 ad opera di *Robin Milner* per studiare le proprietà strutturali di sistemi composti. Questo calcolo ha una struttura semplice, ma molto efficiente nel modellamento di sistemi concorrenti. E' composto da un piccolo insieme di operatori con cui si costruisce un'ampia varietà di descrizioni di sistemi. I blocchi di base di queste descrizioni sono le azioni le quali rappresentano passi di esecuzione interna oppure interazioni potenziali con l'ambiente esterno (attraverso input e output). Le azioni visibili prendono il nome della porta su cui agiscono e se sono output vengono soprabarrati. In genere l'insieme di tutte le azioni di questo calcolo si indica con:

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

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

8.2.1 Struttura Sintattica di CCS

L'insieme delle azioni del CCS è definito da Accs = Act ∪ { τ } dove Act comprende tutte le azioni esterne (input e output) che possono essere svolte dal sistema, mentre τ rappresenta l'azione interna.
Gli operatori di base del CCS sono i seguenti (indichiamo con P e Q processi generici):

1. ***Action Prefix*:** a.P, azione che trasforma il processo in un altro.
2. **Scelta non deterministica:** P + Q
3. **Composizione parallela:** P | Q
4. **Restrizione:** P/Q (Q ⊆ Accs – { τ })
5. ***Relabelling*:** P[f] (f: Accs → Accs)

Schematizzando, i termini di questo calcolo sono generati dalla seguente Grammatica:

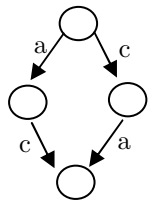
$$P ::= nil \mid a.P \mid P + Q \mid P \mid Q \mid P / L \mid P[f]$$

dove *nil* è il processo inattivo che non esegue alcun comportamento.

Esempio: la macchina vista prima può essere scritta come:

wake.(35.coffee.ok.nil+40.chocolate.ok.nile)

la composizione in parallelo sarà a.nil | c.nil (sintassi). Usualmente si dà alla composizione parallela una semantica “interleaving”:



(semantica)

Ciò che permette di passare dalla formula CCS alla macchina è la semantica operazionale.

8.2.2 Semantica Operazionale di CCS

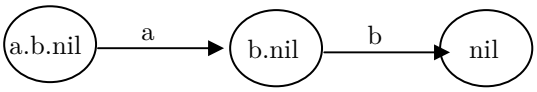
Con semantica operazionale di un'algebra di processi si intendono i passi di esecuzione che possono essere fatti da un processo. Questi comportamenti sono descritti dagli assiomi e dalle regole di transizione per gli operatori che nel CCS sono le seguenti:

1. Action Prefix

$$a \cdot P \xrightarrow{a} P$$

Significa che il processo a.P può sempre eseguire l'azione a e trasformarsi nel processo p (o meglio attivare il processo p)

Esmpio: a.b.nil



2. Scelta non deterministica:

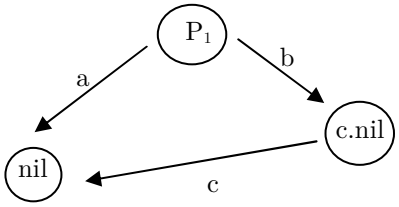
$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'}$$

Se $P_1 \xrightarrow{a} P_1'$ allora $P_1 + P_2 \xrightarrow{a} P_1'$

oppure

Se $P_2 \xrightarrow{a} P_2'$ allora $P_1 + P_2 \xrightarrow{a} P_2'$

Esempio: a.nil + b.c.nil (P_1=a.nil, P_2=b.c.nil)



3. Composizione parallela:

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 \mid P_2 \xrightarrow{a} P_1' \mid P_2} \qquad \text{(Interleaving Semantic)}$$

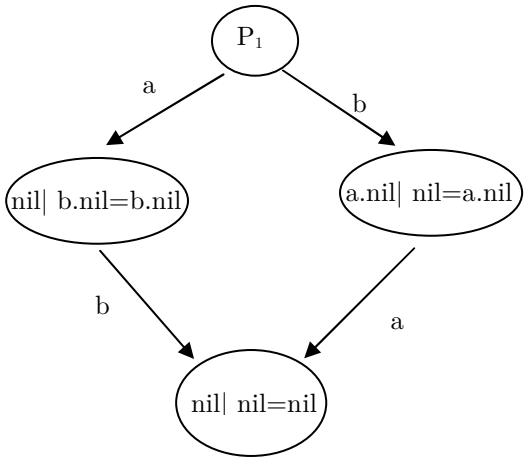
Se $P_1 \xrightarrow{a} P_1'$ allora $P_1 \mid P_2 \xrightarrow{a} P_1' \mid P_2$

Oppure:

$$\frac{P_2 \xrightarrow{a} P_2'}{P_1 \mid P_2 \xrightarrow{a} P_1 \mid P_2'} \qquad \text{(Interleaving Semantic)}$$

Se $P_2 \xrightarrow{a} P_2'$ allora $P_1 \mid P_2 \xrightarrow{a} P_1 \mid P_2'$

Esempio: a.nil|b.nil (P_1=a.nil e P_2=b.nil)



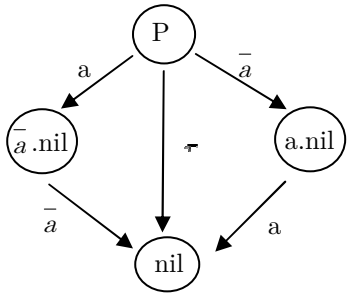
Da notare che le uguaglianze negli stati valgono perché valgono alcune regole di assorbimento:

- nil | p = p
- nil | p.nil = p.nil
- nil | nil = nil

Considerando $\mathrm{Accs} = \mathrm{Act} \cup \overline{\mathrm{Act}} \cup \{\tau\}$ dove $\overline{\mathrm{Act}}$ è l'insieme delle azioni negative, si definisce una terza regola del parallelo:

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

Esempio: a.nil | $\bar{a}$.nil



4. Restrizione

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 / L \xrightarrow{a} P_1' / L} \qquad (a, \bar{a} \notin L)$$

Se $P_1 \xrightarrow{a} P_1'$ allora $P_1 / L \xrightarrow{a} P_1' / L$

Esempio: (a.nil | $\bar{a}$.nil) / {a}
Esaminiamo la Restrizione partendo dal parallelismo si eliminano i due rami esterni, (a.nil | $\bar{a}$.nil) / {a} in questo modo restringo la possibilità di comunicazione di p sulla porta (gate) a: la nascondo sia in ingresso che in uscita

Possiamo dire che: (a.nil | $\bar{a}$.nil) / {a} ≈ nil

5. Relabelling

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1[f] \xrightarrow{f(a)} P_1'[f]}$$

Se $P_1 \xrightarrow{a} P_1'$ allora $P_1[f] \xrightarrow{f(a)} P_1'[f]$

8.3 HENNESSY MILNER LOGIC (HML)

Per poter fare il model checking su LTS c'è bisogno di definire una logica temporale definita su di essi (logica *ACTION BASED*), quella che andremo ad analizzare è la HML (*HENNESSY MILNER LOGIC*).

Sintassi:

$$\varphi := \neg\ \varphi \mid \varphi \wedge \varphi \mid \text{true} \mid [a]\varphi \mid \langle a\rangle\varphi$$

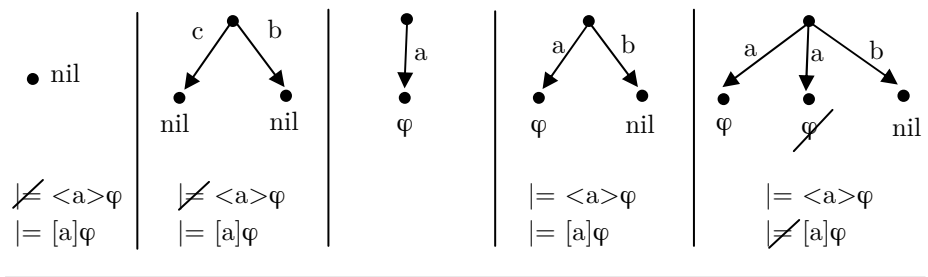
Analizziamo i due operatori:

- $\langle a\rangle\varphi$ $\exists$ un prossimo stato raggiunto con a, in cui vale φ.
- $[a]\varphi$ $\forall$ prossimo stato raggiunto con a, vale φ.

Semantica:

- S $\models \langle a\rangle\varphi$ se partendo da uno stato S è possibile raggiungere, attraverso un'azione "a" ,uno stato S' : S' $\models \varphi$
- S $\models [a]\varphi$ se per ogni stato S' raggiungibile da S, attraverso un'azione "a", vale S' $\models \varphi$

Esempi:



A Two counter machine in CCS

May 16, 2014

A *two counter machine* is a machine composed by two unlimited counters that can be incremented, decremented and checked for zero, and a Finite State Automaton (actually, a Transducer, since it can have outputs) that controls the operations over the two counters, defining the function computed by the machine. A two counter machine is known to be Turing equivalent, that is, any Turing machine can be expressed as a two counter machine.

A two counter machine can be expressed in CCS as follows:

$C = inc.(C[a\backslash b] \mid b.C) \setminus \{b\} + dec.\overline{a}.nil$

$Z = inc.(C[a\backslash b] \mid b.Z) \setminus \{b\} + tst.Z$

$Counter = Z$

$Counter1 = Counter[inc\backslash inc1, dec\backslash dec1, tst\backslash tst1]$

$Counter2 = Counter[inc\backslash inc2, dec\backslash dec2, tst\backslash tst2]$

$TwoCounterMachine = Counter1 \mid ASF \mid Counter2$

where ASF is the Finite State Automaton that defines the function of the two counter machine.

This implies that CCS is Turing equivalent. Notice that all the CCS operators are needed to define a two counter machine.