UNIVERSITÀ DEGLI STUDI DI FIRENZE
FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica

Appunti del Corso di Informatica Industriale 2

Umberto Monile

Indice

1. Modellazione Markoviana degli Attributi di Dependability

1.1 Introduzione

Definizione di *Dependability*: proprietà di un sistema di essere adeguato alla dipendenza da parte di un essere umano, o di una collettività, senza il pericolo di rischi inaccettabili.

La dependability può essere definita come la credibilità di un sistema di calcolo, ovvero il grado di fiducia che può essere ragionevolmente riposto nei servizi che esso offre, dove: il **servizio** offerto da un sistema di calcolo è rappresentato dal suo comportamento così come viene percepito dagli utenti; **l'utente**, umano o fisico, rappresenta un sistema distinto che interagisce con il sistema di calcolo.

La dependability di un sistema di calcolo include gli **attributi** di affidabilità, disponibilità, sicurezza e protezione:

(*Reliability*, *Availability*, *Safety*, *Security*)

- **Reliability** (affidabilità): sistema pronto ad essere usato.
- **Availability** (disponibilità): sistema che fornisce continuità di servizio.
- **Safety** (sicurezza): assenza di conseguenze catastrofiche.
- **Security** (protezione): assenza di intrusioni.

In molti casi, la sicurezza (security) intesa come protezione da guasti intenzionali, non è considerata fondamentale. Viene quindi eliminato l'attributo Security e viene aggiunto *Maintainability* ottenendo quindi:

RAMS = (**R**eliability, **A**vailability, **M**aintainability, **S**afety)

La **manutenibilità** misura la facilità con la quale un sistema può essere riparato una volta manifestatosi il fallimento.

Le minacce (**impedimenti**) che possono violare la dependability di un sistema sono classificate in:

- guasti (*fault*): è il difetto fisico o malfunzionamento che avviene in qualche componente.
- errori (*error*): è il comportamento scorretto causato dal fault.
- fallimento (*failure*): è l'incapacità del sistema a realizzare il suo servizio.

Le **tecniche** (mezzi) utilizzate per difendere un sistema da tali minacce sono le seguenti:

- **Prevenzione dai guasti** (*fault prevention*): come possono essere prevenute le occorrenze di guasti;
- **Tolleranza ai guasti** (*fault tolerance*): come garantire un servizio che si mantenga conforme alle specifiche, nonostante i guasti;
- **Eliminazione del guasto** (*fault removal*): come ridurre l'occorrenza (numero, gravità) dei guasti;
- **Predizione di guasti** (*fault forecasting*): come stimare il numero, la frequenza di incidenza, presente e futura, e le conseguenze dei guasti.

1.2 Valutazione quantitativa degli attributi RAMS con il Metodo Markoviano

Gli Attributi RAMS possono essere valutati quantitativamente. Una valutazione quantitativa permette di fissare dei requisiti, e di verificare se gli attributi RAMS ottenuti per un sistema rispettano i requisiti imposti, inoltre può aiutare la ricerca di soluzioni nel caso che i requisiti non siano soddisfatti. La valutazione degli attributi può essere fatta tramite diversi approcci analitici, quelli più comunemente utilizzati sono : la modellizzazione combinatoria e quella markoviana, noi analizzeremo solo la seconda.
I processi di Markov si basano sul concetto di stato e su quello di transizione. Lo stato rappresenta tutto ciò che deve essere conosciuto per descrivere il sistema in un dato istante. Nel caso di modelli di dependability uno stato rappresenta una possibile configurazione di entità sane ed entità guaste. Le transizioni descrivono i passaggi di stato al verificarsi di un evento: guasto di una nuova entità, riparazione di una entità guasta. Le transizioni di stato sono caratterizzate da probabilità, come la probabilità di guasto di una entità, la probabilità di riparazione e il fattore di copertura.

1.2.1 Affidabilità

L'affidabilità (reliability) di un sistema è la misura del tempo continuativo in cui viene fornito un servizio corretto.

Definizione di Distribuzione dell'Affidabilità:
La distribuzione dell'affidabilità R(t) esprime la probabilità che un componente, correttamente funzionante al tempo t_0, sia correttamente funzionante al tempo t.

$$R(t) = e^{-\lambda \cdot t}$$

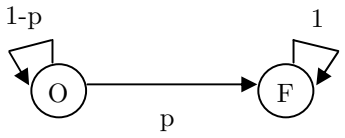
Il tasso di fallimento è denotato dal simbolo λ. In generale il tasso di fallimento (failure rate) di un sistema è il numero di fallimenti nell'unità di tempo.

Definiamo **MTTF** (*Mean Time To Failure*) come il tempo medio tra l'avvio e il guasto; è il reciproco del tasso di fallimento:

$$MTTF = \frac{1}{\lambda}$$

Prendiamo la seguente catena di Markov composta da due stati: Op (operativo) ed F (guasto)

A questo modello associamo la probabilità di passare da uno stato all'altro, inoltre deve essere rispettata la proprietà della Catena di Markov ovvero che la somma delle probabilità di tutti gli archi uscenti dal nodo è pari a 1



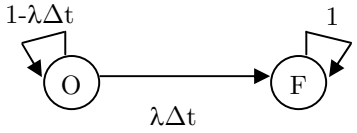
La probabilità che al tempo t + Δt il sistema si trovi in OP dato che era in OP in t è data da:

$$P_{OP} = \frac{R(t + \Delta t)}{R(t)} = \frac{e^{-\lambda(t+\Delta t)}}{e^{-\lambda t}}$$

La probabilità che da OP si passi in F è :

$$p = 1 - \frac{R(t+\Delta t)}{R(t)} = 1 - \frac{e^{-\lambda(t+\Delta t)}}{e^{-\lambda t}} = 1 - e^{-\lambda \Delta t} = 1 - 1 + (-\lambda \cdot \Delta t) + (-\lambda \cdot \Delta t)^2 + \ldots\ldots\ldots = \lambda \cdot \Delta t$$

Il sistema può essere modellato come segue:



$$R(t) = e^{-\lambda t}$$

Questo appena visto è un modello semplice, ora vediamo un modello per la determinazione dell'affidabilità di un sistema costituito da entità interconnesse in modo TMR (*Triple Modular Redundancy*).

Un sistema con ridondanza modulare tripla (TMR) è costituito da tre moduli uguali che lavorano in parallelo sugli stessi dati d'ingresso e le cui uscite sono confrontate da un elemento di comparazione (voter), la cui uscita è pari alla maggioranza degli ingressi.

Ipotesi: V (voter) è esente da guasti

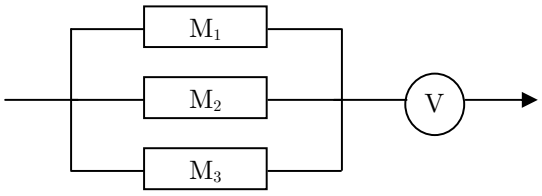
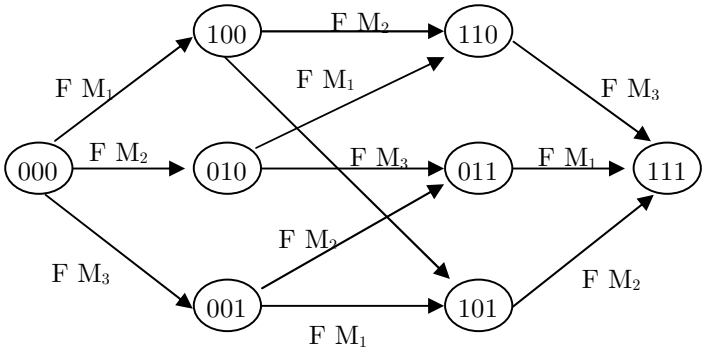


Figura 1: TMR (ridondanza modulare tripla)

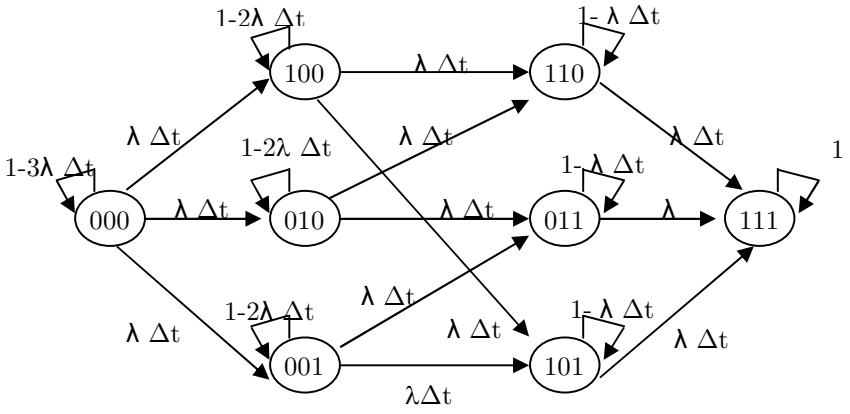
Si può numerare ogni stato con dei numeri binari, usando una variabile booleana per ogni modulo.



Come si può osservare dallo stato (0,0,0) sono possibili tre transizioni dovute al guasto del primo, del secondo o del terzo modulo. Si può osservare che ogni singola transizione è causata da un singolo evento.
Si può affermare che:

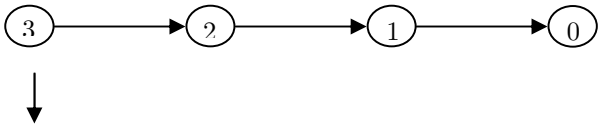
- Per un modello TMR, tutti gli stati con due variabili a 1 sono stati di guasti.
- Per un sistema in parallelo (NO TMR) l'unico stato di guasto è l'ultimo stato, quindi il sistema funziona se c'è almeno un valore 0.
- Per un sistema in serie l'unico stato funzionante è il primo (000).

Per trasformare questo modello in un modello Markoviano bisogna aggiungere le probabilità.
Per un modello TMR si considerano tutti i moduli con lo stesso tasso di fallimento (stesso MTTF e stesso λ)



Questa è una catena Markoviana che ha un numero di stati esponenziale rispetto al numero di moduli, infatti, dato n (numero di moduli) il numero totale di stati possibili è 2^n.

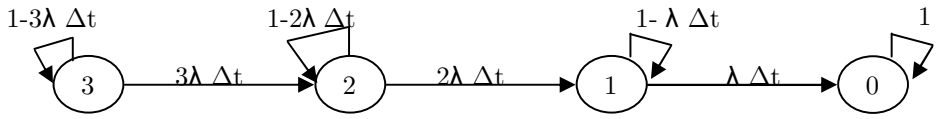
In questo modello è evidente il problema dell'esplosione dello spazio degli stati, questo succede quando ho oggetti equivalenti che possono cambiare indipendentemente dagli altri.
E' possibile semplificare il modello e ridurre così il numero di stati, l'idea che sta alla base di questa "compressione" è la seguente: l'importante non è sapere quale modulo è guasto ma solo se uno dei moduli lo è; come è facile intuire una delle conseguenze è la perdita di informazione. La semplificazione è sempre possibile se i moduli sono equivalenti ed hanno lo stesso λ,



Il numero degli stati è lineare ed è uguale a n+1

moduli funzionanti

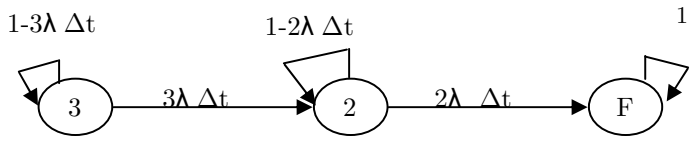
la catena Markoviana della figura sovrastante diventa:



Questo appena visto è un **modello astratto** valido per ogni sistema che abbia tre moduli uguali (TMR, serie, parallelo).

- **Sistema TMR** abbiamo: 3,2 Stati OP
  1,0 Stati F
- **Sistema Serie** abbiamo: 3 Stato OP
  2,1,0 Stati F
- **Sistema Parallelo** abbiamo: 3,2,1 Stati OP
  0 Stati F

Se consideriamo il caso TMR possiamo fare un'ulteriore astrazione possiamo riunire gli stati 1 e 0 in F:



Un modello Markoviano di questo tipo si risolve calcolando le probabilità di essere nei vari stati al tempo t: $P_f(t)$, $P_2(t)$, $P_3(t)$.
Conoscendo le tre probabilità possiamo calcolarci l'affidabilità del sistema con la seguente regola $R(t) = P_3(t) + P_2(t)$ ed inoltre sappiamo che:

$$Q(t) = 1 - R(t) = P_f(t)$$
$$P_f(t) + P_3(t) + P_2(t) = 1$$

Per calcolare le tre probabilità posso condurmi alle seguenti equazioni che derivano direttamente dalla catena di Markov:

$$P_3(t + \Delta t) = (1 - 3\lambda\Delta t) \cdot P_3(t)$$
$$P_2(t + \Delta t) = (1 - 2\lambda\Delta t) \cdot P_2(t) + (3\lambda\Delta t) \cdot P_3(t)$$
$$P_f(t + \Delta t) = P_f(t) + (2\lambda\Delta t) \cdot P_2(t)$$

Inoltre è possibile scrivere il sistema in forma matriciale:

$$\begin{bmatrix} P_3(t+\Delta t) \\ P_2(t+\Delta t) \\ P_f(t+\Delta t) \end{bmatrix} = \begin{bmatrix} 1-3\lambda\Delta t & 0 & 0 \\ 3\lambda\Delta t & 1-2\lambda\Delta t & 0 \\ 0 & 2\lambda\Delta t & 1 \end{bmatrix} \cdot \begin{bmatrix} P_3(t) \\ P_2(t) \\ P_f(t) \end{bmatrix}$$

↓

Matrice di Raggiungibilità (A) rappresenta il grafo in forma matriciale; la somma degli elementi in colonna è 1

In forma vettoriale:

$$P(t + \Delta t) = A \cdot P(t)$$

posso scrivere:

$$\frac{P_3(t+\Delta t)-P_3(t)}{\Delta t} = -3\lambda P_3(t)$$
$$\frac{P_2(t+\Delta t)-P_2(t)}{\Delta t} = -3\lambda P_3(t) - 2\lambda P_2(t)$$
$$\frac{P_f(t+\Delta t)-P_f(t)}{\Delta t} = -2\lambda P_2(t)$$

Facendo il passaggio a limite otteniamo:

$$\frac{dP_3}{dt} = -3\lambda P_3(t)$$
$$\frac{dP_2}{dt} = -3\lambda P_3(t) - 2\lambda P_2(t)$$
$$\frac{dP_f}{dt} = -2\lambda P_2(t)$$

Risolvendo il sistema di disequazioni differenziali otteniamo le seguenti soluzioni:

$$P_3(t) = e^{-3\lambda t}$$
$$P_2(t) = 3 \cdot e^{-2\lambda t} - 3 \cdot e^{-3\lambda t}$$
$$P_f(t) = 1 - 3 \cdot e^{-2\lambda t} + 2 \cdot e^{-3\lambda t}$$

A questo punto possiamo calcolarci la distribuzione dell'affidabilità R(t)

$$R(t) = P_3(t) + P_2(t) = 1 - P_f(t) = 3 \cdot e^{-2\lambda t} - 2 \cdot e^{-3\lambda t}$$

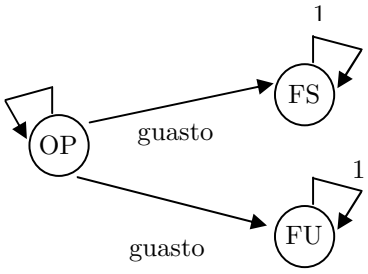
Questo risultato coincide con la formula generale dell'affidabilità di un TMR indipendente dal modello dei guasti usati:

$$R_{TMP}(t) = 3 \cdot R^2(t) - 2 \cdot R^3(t)$$

questo risultato vale secondo l'assunzione che le probabilità siano esponenziali.

1.2.2 Safety

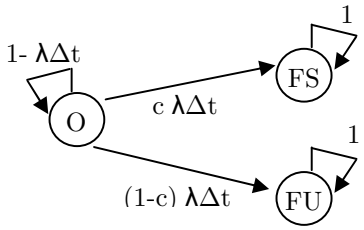
Definizione di Distribuzione della Safety:
La distribuzione S(t) esprime la probabilità che un sistema sia funzionante o sia in uno stato sicuro (*fail-safe*) al tempo t.
Il sistema sarà safet se, a fronte di un guasto, si porta in uno stato di fallimento NON catastrofico.



Fail-Safe (fallimento benigno)

Fail-UnSafe (fallimento catastrofico)

La **misura di copertura (c)** è la capacità dei meccanismi preposti di rilevare i guasti (in termini percentuali) e di portarsi quindi in uno stato FS, comunque c'è sempre una probabilità che il guasto non venga rilevato.



Analogamente a prima

$$\begin{bmatrix} P_{OP}(t+\Delta t) \\ P_{FS}(t+\Delta t) \\ P_{FU}(t+\Delta t) \end{bmatrix} = \begin{bmatrix} 1-\lambda\Delta t & 0 & 0 \\ c\lambda\Delta t & 1 & 0 \\ (1-c)\lambda\Delta t & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_{OP}(t) \\ P_{FS}(t) \\ P_{FU}(t) \end{bmatrix}$$

segue che:

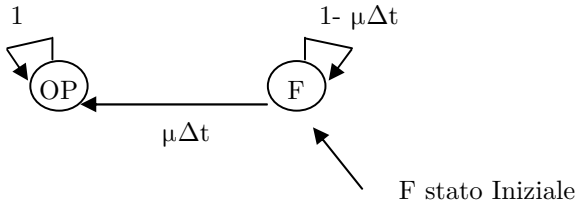
$$P_{FS}(t) = c - c \cdot e^{-2\lambda t}$$
$$P_{FU}(t) = (1-c) - (1-c) \cdot e^{-2\lambda t}$$
$$P_{OP}(t) = e^{-\lambda t}$$

Dunque

$$S(t) = P_{OP}(t) + P_{FS}(t) = c + (1-c) \cdot e^{-\lambda t}$$

1.2.3 Manutenibilità

Definizione di Distribuzione della manutenibilità:
La distribuzione M(t) esprime la probabilità che un sistema guasto venga ripristinato entro un prefissato periodo di tempo t.



Il tasso di riparazione μ è definito attraverso la relazione:

$$\mu = \frac{1}{MTTR}$$

La distribuzione M(t) è:

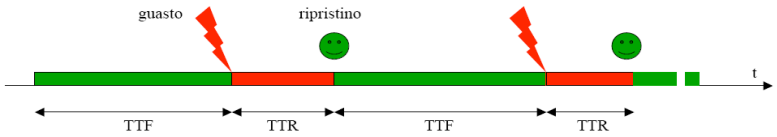
$$M(t) = P_{OP}(t) = 1 - e^{-\mu \cdot t}$$

La probabilità di essere nello stato F al tempo t:

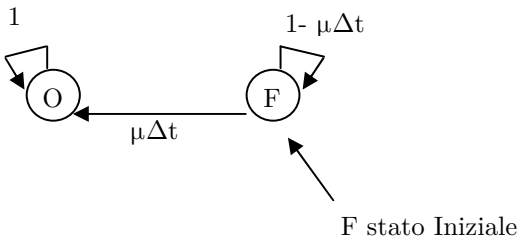
$$P_F(t) = e^{-\mu \cdot t}$$

1.2.4 Disponibilità

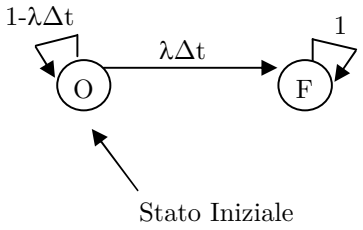
Definizione di Distribuzione della disponibilità:
La distribuzione A(t) esprime la probabilità che il sistema sia funzionante al tempo t in un periodo in cui possono alternarsi periodi di guasto (fallimento) e periodi di corretto funzionamento



Sia λ il tasso di fallimento e sia μ il tasso di riparazione, partendo dal modello di Manutenibilità e di Affidabilità.

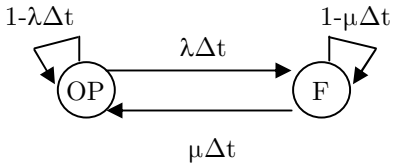


Manutenibilità



Affidabilità

Unendoli in un unico modello simmetrico e ciclico si ottiene:



A questo punto dobbiamo calcolarci le probabilità P_{OP} e P_{FS}:

$$\begin{bmatrix} P_{OP}(t+\Delta t) \\ P_{FS}(t+\Delta t) \end{bmatrix} = \begin{bmatrix} 1-\lambda\cdot\Delta t & \mu\cdot\Delta t \\ \lambda\cdot\Delta t & 1-\mu\cdot\Delta t \end{bmatrix} \cdot \begin{bmatrix} P_{OP}(t) \\ P_{FS}(t) \end{bmatrix}$$

$$P_{OP}(t) = A(t) = \frac{\mu}{(\lambda+\mu)} + \frac{\lambda}{(\lambda+\mu)} \cdot e^{-(\lambda+\mu)\cdot t}$$

$$P_{F}(t) = \frac{\lambda}{(\lambda+\mu)} - \frac{\lambda}{(\lambda+\mu)} \cdot e^{-(\lambda+\mu)\cdot t}$$

Definiamo **MTBF** (*Mean Time Between Failures*) il tempo medio tra due fallimenti (nel frattempo vi è quindi stata una riparazione):

$$MTBF = MTTF + MTTR$$

Calcoliamo la disponibilità a regime:

$$A(\infty) = \frac{\mu}{(\lambda+\mu)} = \frac{\frac{1}{MTTR}}{\frac{1}{MTTF}+\frac{1}{MTTR}} = \frac{MTTF}{MTTR+MTTF} = \frac{MTTF}{MTBF}$$

Ponendo $\mu=0$ riconduciamo il modello a quello dell'affidabilità; dunque la disponibilità a regime sarà $A(\infty)=0$ e $P_{OP}(t) = e^{-\lambda\cdot t}$ come visto prima nel modello dell'affidabilità.

2. Ridondanza

2.1 Duplicazione e Confronto

Duplicazione e Confronto è detta anche architettura 2 su 2: rappresenta l'architettura di ridondanza attiva più semplice.

- Si basa sulla duplicazione del master e sull'aggiunta di un comparatore che confronta le uscite delle due macchine.
- Le due macchine ricevono gli stessi input e nel caso sia rilevata una differenza (e quindi un errore) la segnala all'esterno.

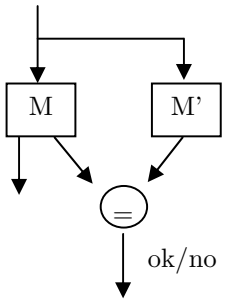
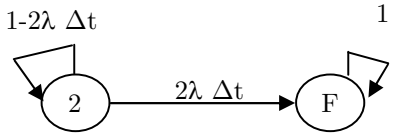


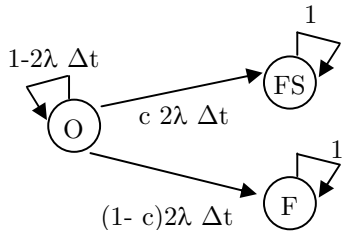
Figura 2: Architettura 2 su 2

Il modello markoviano (nel caso consideriamo l'affidabilità) corrispondente è uguale a quello visto per una singola macchina (par 1.2.1) ma con probabilità di fallimento doppia perché i moduli sono uguali e ugualmente necessari.



$$R(t) = e^{-2\lambda t}$$

Se consideriamo la Safety otteniamo:



Indichiamo con c la probabilità che un guasto porti a risultati diversi.
Possiamo affermare che:

- vado in FS nel caso in cui ho moduli (M ed M') guasti uscite diverse.
- vado in FU nel caso in cui ho moduli guasti uscite uguali.

2.2 Duplicazione Riconfigurabile

Duplicazione riconfigurabile è detta anche architettura 1 su 2:

- Una macchina Master e una Slave.
- La macchina slave subentra a quella master in caso di guasto del master
- La rilevazione dell'errore avviene mediante codifica dell'errore o time-out: il fallimento è un crash della macchina, sempre rilevabile (assunzione sui guasti)

La duplicazione Riconfigurabile con:

- riserva Fredda (*cold spare*) lo slave viene tenuto normalmente spento.
- riserva Calda (*hot spare*) lo slave funziona in parallelo al master, ricevendo gli stessi input; i suoi output non vengono forniti all'esterno.

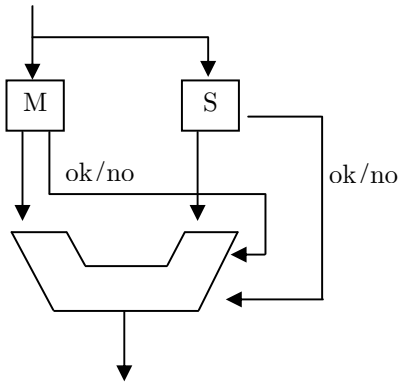
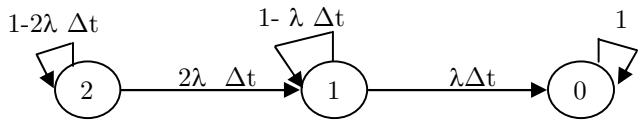
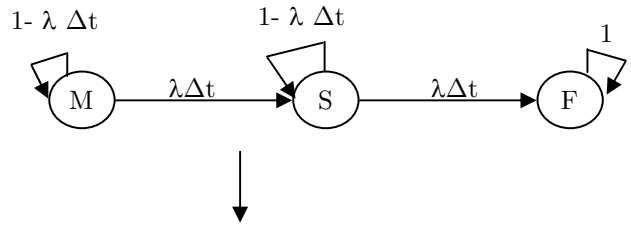


Figura 3: Architettura 1 su 2

Consideriamo il caso in cui vogliamo determinare l'affidabilità; il modello di Markov corrispondente nel caso di riserva calda (Hot Spare) è:



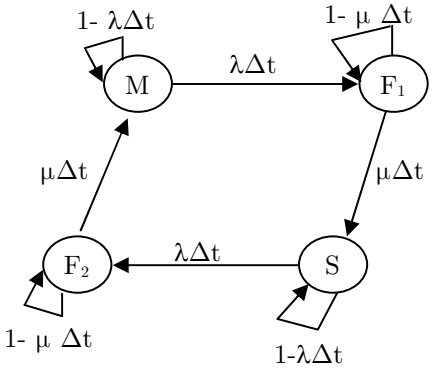
Nel caso riserva Fredda (Cold Spare) ho un sistema che ha un elemento in funzione per volta



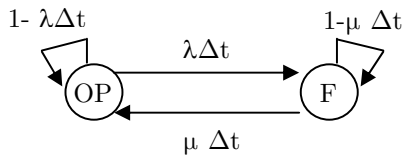
Il 2 è scomparso perché lo slave è spento quando il master funziona

In questo caso non è stato tenuto conto del set-up dello slave (è compreso nell'arco del guasto da M ad S). Infatti che per un dato periodo di tempo il sistema non è disponibile non interessa ai fini dell'affidabilità.

Se si considerano i tempi di Set-up la catena markoviana diventa:



Dove F_1 e F_2 sono rispettivamente il fallimento del Master M e dello Slave S. Si presuppone che nel tempo in cui lo slave funziona il master sia stato riparato e che lo slave non si guasti prima che questo sia avvenuto. Con questo modello si ha anche la possibilità di determinare la disponibilità:



2.3 Modello NMR

Un sistema con ridondanza modulare N (NMR – *N Modular Redundancy*) è costituito da N entità uguali che adempiono alle stesse funzioni e le cui uscite sono confrontate da un elemento di comparazione (*voter*), la cui uscita è pari alla maggioranza degli ingressi. Il votatore non distingue tra guasti TRANSITORI e PERMANENTI.

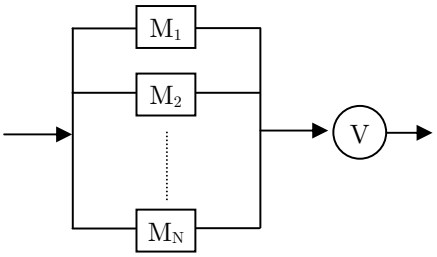


Figura 4: NMR (N Modular Redundancy)

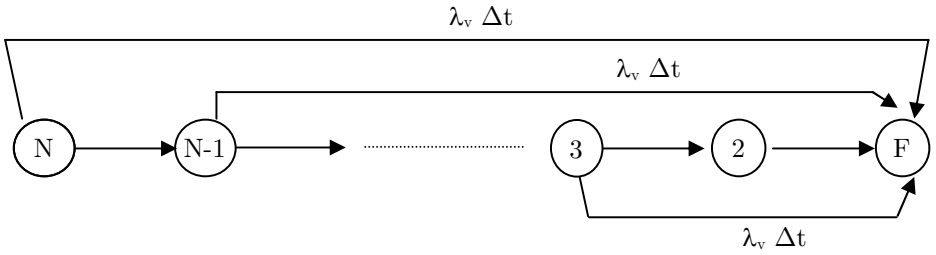
La scelta di N deve essere dispari e deve essere un valore maggiore di 3. Il vantaggio di utilizzare N > 3 moduli risiede nel fatto che più di un singolo guasto può essere tollerato. Nel caso di N pari per esempio 8 entità funzionanti se ne considerano solo 7.

Il sistema con Voter consente il mascheramento sino a $\frac{N-1}{2}$ guasti (*fault masking*).

La catena di Markov che rappresenta l'NMR (hp: V non guasto) è:

Se non ci sono (N+1)/2 unità concordanti a formare una maggioranza il sistema fallisce.

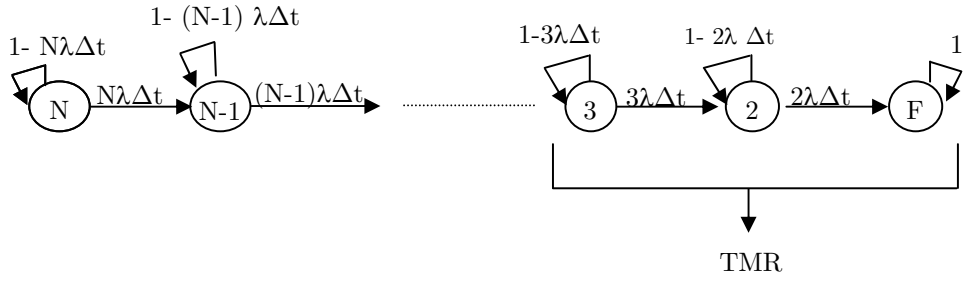
Se si guasta il votatore il sistema è complessivamente guasto:



λ_v è il tasso di guasto del votatore
L'affidabilità è data da: $R(t) = e^{-\lambda_v \cdot \Delta t} \cdot R_{NMR}(t)$

2.3.1 NMR Riconfigurabile

Nei sistemi N-modular redundancy (NMR) con voting, la possibilità di fare *fault masking* diviene sempre minore man mano che alcune copie si guastano, e, in alcuni casi, è anche possibile che il voto delle copie guaste superi in maggioranza quello delle copie corrette. Se le copie guaste fossero escluse dalla votazione il NMR potrebbe invece continuare a lavorare correttamente, la catena Markoviana che rappresenta questo scenario è la seguente (hp: V non guasto):



Un approccio utilizzato a tale scopo è la *hybrid redundancy*, che rimpiazza il modulo guasto con un modulo di riserva non utilizzato.
La ridondanza ibrida è implementata tramite la combinazione di M moduli attivi e di S riserve (fredde o calde) per un totale di N moduli dove N=M+S.

In questa soluzione viene adottata sia una tecnica di mascheramento dei guasti (fault masking) per prevenire gli errori attraverso il voting, sia una tecnica di diagnosi dei guasti stessi con relative azioni di **riconfigurazione** per rimpiazzare e isolare il componente guasto.

Vediamo un esempio di *hybrid redundancy (hot spare)*:

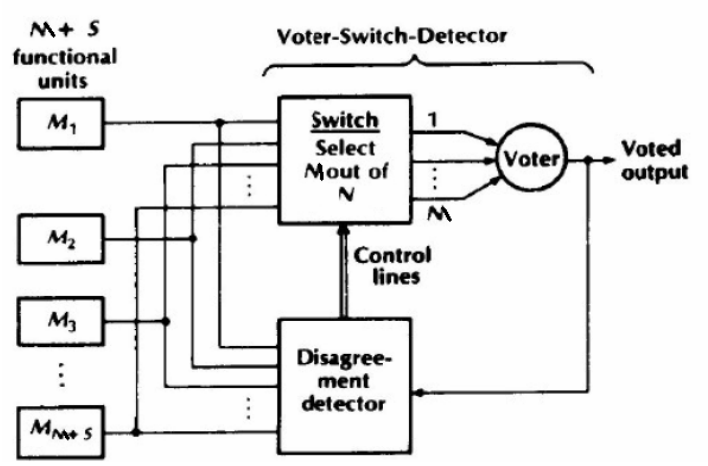


Figura 5: *Hybrid redundancy*

Il disagreement detector riceve tutte le uscite dalle unità e l'uscita del sistema e comunica al selettore quali sono le unità che hanno dato una risposta scorretta e che devono essere escluse, indicando inoltre esplicitamente o implicitamente quali unità tra quelle di riserva debbano essere utilizzate al posto delle unità guaste. Una memoria nel disagreement detector o nel selettore permette di tenere traccia di quali sono le unità rotte che non devono essere prese in considerazione.
In assenza di memoria i guasti temporanei vengono tollerati dalla votazione. In caso di presenza di memoria i guasti temporanei vengono trattati come guasti permanenti.

2.3.1.1 Self purging

Una tecnica più efficiente per la realizzazione dello switch, prevede che l'uscita del voter sia ricollegata ai moduli che partecipano alla votazione. Sono gli stessi moduli ad accorgersi che il loro output (uscita binaria) non corrisponde con quello fornito dal voter, e quindi, automaticamente si escludono dalla votazione. Chiaramente in questa soluzione vanno affrontati problemi di sincronia che solitamente si risolvono tramite un delayed clock. Questa soluzione è generalmente più efficiente della hybrid redundancy.

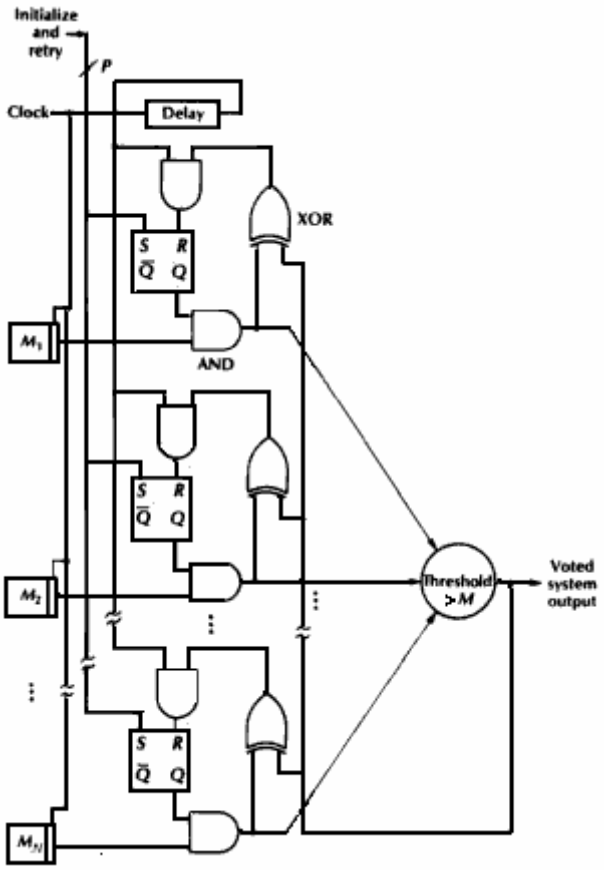


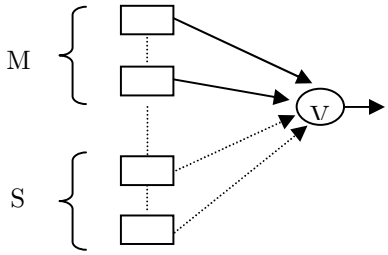
Figura 6: Self purging

Questa struttura non gestisce i guasti temporanei; infatti quando un'unità sbaglia viene esclusa fino a che non sia stato asserito il segnale di reset, che concede una nuova possibilità a tutte le unità.

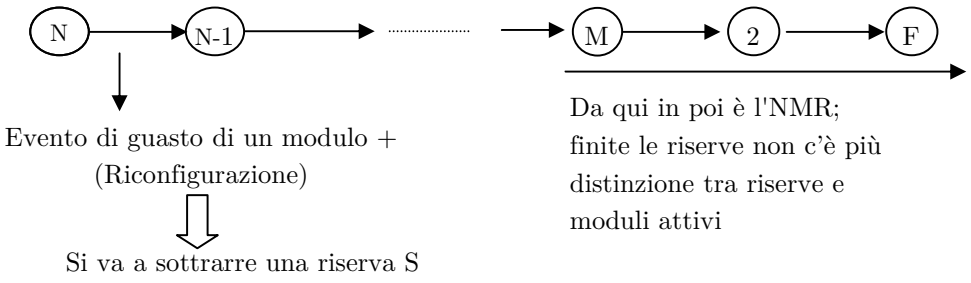
2.3.1.2 Modello Markoviano di un NMR Riconfigurabile

Definiamo un modello markoviano per un NMR Riconfigurabile che **non considera la possibilità di guasti Temporanei**:

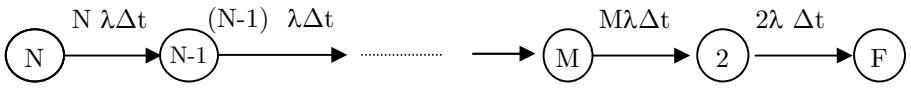
- N= moduli
- M= moduli Attivi
- S = riserve che subentrano in caso di guasti permanenti ai moduli attivi
- N=M+S



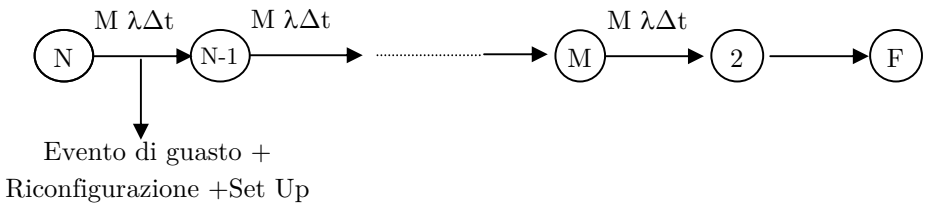
Il modello Markoviano corrispondente è il seguente:



Nel caso in cui consideriamo le riserve S *hot spare*, le quali subentrano in caso di guasti permanenti, otteniamo il seguente modello Markoviano:



Nel caso in cui le riserve S sono *cold spare* (non si guastano perché sono spente):



3. Rilevazione & Correzione d'Errore

3.1 Introduzione

Definizione: Sia W insieme delle parole di n bit. Si definisce **Codice C** un sottoinsieme di W.

Definizione: Presi w1, w2 $\in$C la **distanza di *Hamming*** d(w_1,w_2) è il numero di bit diversi tra w1, w2, cioè rappresenta il numero di bit da invertire per trasformare una parola di codice nell'altra.

Definizione: La distanza di Hamming su C è la minima distanza di Hamming che si misura confrontando due a due tutte le parole del codice:

$$d(C) = \min\left\{d(w_1, w_2) \middle| w_1, w_2 \in C, w_1 \neq w_2\right\}$$

Un errore singolo su una parola trasforma w_1 in una w_1' tale che d(w_1,w_1')=1, dove w_1 è la parola trasmessa e w_1' è la parola ricevuta.
Supponiamo che l'errore venga fatto sul k-esimo bit:

$e_k = 00.....1...0 \quad \rightarrow \quad w_1' = w_1 + e_k$

K-esimo elemento

XOR bit a bit
Somma Modulo 2 bit a bit $\rightarrow$ Somma vettoriale in modulo 2

Per correggere l'errore sulla parola ricevuta basta fare:

$$w_1 = w_1' - e_k$$

oppure:

$$w_1 = w_1' + e_k$$

che sono uguali essendo operazioni in modulo 2, ma si deve conoscere e_k

3.2 Codici di parità

La codifica più semplice consiste nel controllo della parità. Esistono molte variazioni rispetto all'idea che sta alla base della codifica. Esistono due tipi di parità:

1. parità pari
2. parità dispari.

a seconda del numero, pari o dispari, dei bit posti a 1 nelle parole di codice.
Come si può immaginare, in una codifica con parità pari, il cambiamento di un singolo bit provoca un numero dispari di 1 e, viceversa, in una codifica a parità dispari, la variazione di un singolo bit provoca un numero pari di 1.

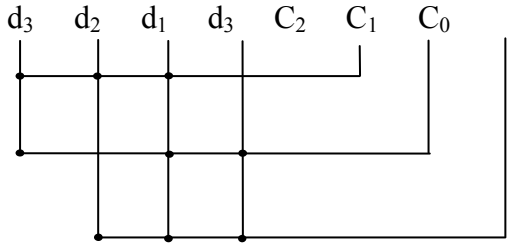
La codifica di parità viene eseguita aggiungendo alla parola un bit con valore dipendente dal tipo di parità scelta, in modo da avere un numero di bit a 1 pari o dispari a seconda della parità in uso. Si noti che, poiché la distanza da un code-word valido e l'altro, in entrambe le tipologie è pari a 2, questa tecnica è in grado di rilevare gli errori su singolo bit, ma non è in grado di correggerli. Dato il modo in cui viene costruita la code-word: aggiunta del bit di controllo, la codifica di parità è una codifica separabile.
Un'applicazione comune che utilizza questa tecnica è rappresentata dalle memorie dei computer. La parola, prima di essere scritta in memoria, viene codificata, cioè viene aggiunto il bit di parità, quindi viene memorizzata.
Al momento della lettura viene controllato il bit di parità per verificare che non ci siano stati errori. In caso di mancata verifica della parità l'utente viene avvertito.

3.3 Codici Overlapping Parity

Una particolare applicazione dell'utilizzo ridondato dei bit di parità si ha in codici che permettono una correzione dell'errore. I codici *Overlapping Parity* permettono, oltre ad una **rivelazione** degli errori singoli, anche alla loro localizzazione in modo che, mediante un'operazione di complemento, l'errore può essere **corretto**.

Il concetto di base di questa tecnica consiste in due fasi:

1. **fase di codifica**: vengono generati i bit di parità da associare alla parola da trasmettere; più precisamente nel caso si voglia inviare una parola di 4 bit i bit di parità si calcolano nel seguente modo:



$$C_0 = (d_1 \text{ xor } d_2) \text{ xor } d_0$$
$$C_1 = (d_0 \text{ xor } d_2) \text{ xor } d_0$$
$$C_2 = (d_0 \text{ xor } d_1) \text{ xor } d_0$$

   La parola trasmessa sarà $d_3\ d_2\ d_1\ d_0\ C_2\ C_1\ C_0$.

2. **fase di decodifica**: i bit di parità vengono nuovamente generati nello stesso identico modo con cui avviene nella fase di codifica e poi vengono confrontati attraverso uno xor con quelli memorizzati. Il risultato del confronto permette quindi la correzione dell'eventuale errore, sempre riferendosi al caso di 4 bit di parola e 3 di parità ecco cosa si ottiene:

$$S_0 = ((d_2 \text{ xor } d_1) \text{ xor } d_0) \text{ xor } C_0 = C_0 \text{ xor } C_0$$
$$S_1 = ((d_3 \text{ xor } d_1) \text{ xor } d_0) \text{ xor } C_1 = C_1 \text{ xor } C_1$$
$$S_2 = ((d_3 \text{ xor } d_2) \text{ xor } d_1) \text{ xor } C_2 = C_2 \text{ xor } C_2$$

   La combinazione di $S_2\ S_1\ S_0$ ci fornisce la **Sindrome**, che può servire da indicazione di quale sia il bit sbagliato.

La seguente tabella mostra tutte le possibili combinazioni dei bit S_2 S_1 S_0 e il bit coinvolto nell'errore:

S_2	S_1	S_0	Errore
0	0	0	Nessuno
1	1	0	d_3
1	0	1	d_2
1	1	1	d_1
0	1	1	d_0
1	0	0	C_2
0	1	0	C_1
0	0	1	C_0

Tabella 1: Tutte le possibili combinazioni della Sindrome

Per capire come è stata costruita la tabella consideriamo il caso in cui la sindrome S_2 S_1 S_0 vale [011] si ha che l'errore è su d_0 questo perché il termine d_0 appare solo in S_1 e in S_0 quindi se S_1 e S_0 valgono 1 e S_2 vale 0 vuol dire che il bit sbagliato è d_0, e così vale per tutte le altre combinazioni.

Con 4 bit di dati e con 3 bit di check per un totale di 7 bit vado a coprire tutti gli errori singoli infatti con 3 bit di sindrome ottengo 2^3 situazioni di errore; nello specifico ottengo 7 possibili errori singoli, più il caso di nessun errore.

Si noti che, in questo caso, la "spesa" per la copertura su tutti e quattro i bit è molto alta: 3−bit di parità per 4−bit di informazione cioè il 75% di ridondanza in più.
Comunque, all'aumentare dei bit di informazione, la percentuale di ridondanza diminuisce. In particolare è possibile stabilire il legame tra i bit di informazione da proteggere e i bit di parità. Sia m il numero di bit di informazione e k il numero di bit di parità. Le combinazioni uniche che devono essere contemplate sono almeno m + k cioè dobbiamo proteggere m−bit di informazione e k−bit di parità. Inoltre dobbiamo contemplare anche la combinazione in cui non si sia verificato un errore. In definitiva dobbiamo contemplare almeno m + k + 1 combinazioni uniche. Per cui la relazione tra k ed m è data dalla seguente relazione:

$$2^k \geq m + k + 1$$

Per Rilevare gli errori doppi? → 1. una situazione di non errore 1
2. m + k errori singoli.
3. $\frac{(m+k)\cdot(m+k-1)}{2}$ errori doppi

dovrà essere $2^k \geq m+k+1+\frac{(m+k)\cdot(m+k-1)}{2}$ se questo vale e scegliendo un k adeguato posso correggere anche gli errori doppi.

3.3.1 Overlapping Parity modificato

Usando una variante dell'overlapping parity è ancora possibile rilevare e correggere gli errori singoli e in più permette di rilevare gli errori doppi. L'idea è semplicemente quella di usare la overlapping parity così come è stato appena definito aggiungendo un bit di ridondanza C_3 che non è altro che il bit di parità pari di tutti gli altri presenti nella parola codificata.

$$C_3 = d_3 \text{ xor } d_2 \text{ xor } d_1 \text{ xor } d_0 \text{ xor } C_0 \text{ xor } C_2 \text{ xor } C_1$$

In questo modo si conosce a priori la parità della parola trasmessa. Quindi, nel caso di parità pari, la parola trasmessa avrà sempre un numero pari di "1"

In fase di decodifica i bit di parità vengono nuovamente generati nello stesso identico modo con cui avviene nella fase di codifica e poi vengono confrontati attraverso uno xor con quelli memorizzati. Inoltre il coefficiente S_3 non sarà altro che lo xor tra tutti i bit della parola ricevuta:

$$S_0 = ((d_2 \text{ xor } d_1) \text{ xor } d_0) \text{ xor } C_0 = C_0 \text{ xor } C_0$$
$$S_1 = ((d_3 \text{ xor } d_1) \text{ xor } d_0) \text{ xor } C_1 = C_1 \text{ xor } C_1$$
$$S_2 = ((d_3 \text{ xor } d_2) \text{ xor } d_1) \text{ xor } C_2 = C_2 \text{ xor } C_2$$
$$S_3 = d_3 \text{ xor } d_2 \text{ xor } d_1 \text{ xor } C_3 \text{ xor } C_2 \text{ xor } C_1$$

Nel caso di errori singoli la parità della parola ricevuta cambia e la sindrome mi restituisce il bit errato, mentre nel caso di errore doppio la parità non cambia.

S_3	S_2	S_1	S_0	**Effetti**
0	0	0	0	Nessun Errore
1	1	0	1	d_2 sbagliato
1	..	..	..	d.. sbagliato
0	1	1	1	errore doppio

Se $S_3 = 0$ e S_2 , S_1 , $S_0 \neq 0$, vuol dire che la parità della parola non è cambiata e quindi siamo in presenza di un errore doppio, mentre nell'altro caso avremo un errore singolo perchè la parità della parola è cambiata $S_3 = 1$.

3.4 Codici di Hamming

Il codice di Hamming è un codice correttore lineare (paragrafo 3.5) che prende il nome dal suo inventore Richard Hamming. Un codice di Hamming che codifica k bit di data in n bit, aggiungendo n-k bit di parità (detti anche check bits) si indica con la sigla Hamming(n,k). Il codice di Hamming può rivelare e correggere gli errori di singolo bit.

Per i codici di Hamming vengono definite due matrici di Hamming: la matrice generatrice del codice G e la matrice di controllo di parità (Parity Check Matrix) H. La matrice G(k,n) ha dimensione (n x k) dove le n righe che sono date da tutte le possibili sequenze di bit eccetto la sequenza zero, la matrice PCM H(n,k) ha dimensione (n-k x n) dove le n colonne sono date dalle sequenze di (n-k) bit, eccetto la sequenza zero.

La **codifica** della parola da trasmettere è ricavata moltiplicando il vettore di k bit (parola informativa), per la matrice G di dimensione (n x k) nell'aritmetica modulo 2, che produce quindi un vettore di n bit.

La **decodifica** della parola ricevuta avviene analogamente mediante la sua moltiplicazione per una matrice H di dimensione (n-k $\times$ n), che produce un vettore lungo n-k detto sindrome. Se questo vettore è nullo non si rivelano errori, altrimenti il vettore indica la posizione dell'errore.

La codifica overlapped parity vista nel paragrafo 3.3 può essere studiata all'interno della teoria dei codici di Hamming, nel caso della decodifica possiamo ricavarci la sindrome attraverso H nel seguente modo:

$$\begin{array}{c} \\ s_2 \\ s_1 \\ s_0 \end{array}\begin{array}{c} \begin{array}{ccccccc} C_2 & d_2 & C_0 & d_0 & d_3 & C_1 & d_1 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array} \cdot \begin{bmatrix} C_2 \\ d_2 \\ C_0 \\ d_0 \\ C_2 \\ C_1 \\ d_1 \end{bmatrix} = \begin{bmatrix} S_2 & S_1 & S_0 \end{bmatrix}$$

La matrice H è costruita in base all'ordine della parola trasmessa dove le colonne sono etichettate con i nomi dei bit della parola ricevuta e riportando per ogni colonna i bit di sindrome corrispondenti ad identificarli (Tabella 1).

Se una **parola W è ricevuta correttamente** risulta che H.W = 0; nel caso di una parola **affetta da un errore** (W'=W+e_i), risulta che :

$$H \,.\, W' = H \cdot W \;+\; H \cdot\; e_i = 0 + (H \cdot\; e_i) = S$$

dove (H · W) è zero perchè W una parola di codice; S è la sindrome con errore e_i, e sta ad indicare la colonna i-esima di H, in questo modo S codifica direttamente la posizione del bit errato.

Esempio: Errore in ricezione, con S = [101]. La parola trasmessa è della forma $d_3\ d_2\ d_1\ d_0\ C_2\ C_1\ C_0$.

$$\begin{array}{c} \begin{array}{ccccccc} d_3 & d_2 & d_1 & d_0 & C_2 & C_1 & C_0 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array} \cdot \begin{bmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_2 & S_1 & S_0 \end{bmatrix}$$

Se S =[101] mi identifica la seconda colonna di H è più precisamente mi codifica direttamente il bit errato della parola ricevuta è il secondo (d_2)
I codici di Hamming rientrano nella categoria dei codici lineari.

3.5 Codici lineari

3.5.1 Introduzione

Definizione: Dato un alfabeto B={0,1} un codice è lineare se ∃ una matrice PCM H su B di dimensione $(n - k \times n)$ dove n è la lunghezza di w ∈ C e k è la lunghezza della parola originale r, e C è lo spazio nullo su H, cioè :

$$C = \left\{ x \;\middle|\; Hx^T = 0 \right\} \quad \leftarrow \text{Insieme delle parole corrette}$$

Esempio: prendiamo H dove n=7=2^3-1, la matrice di Hamming risulta

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

E' vero in generale che H è formata da tutti i vettori di m bit tranne il vettore 0 se n=2^m-1; il numero delle possibili parole che costituisce il codice C è dato da 2^{2^m-1} dove n=2^m-1 è la lunghezza della parola.

Definizione: C è un sottospazio lineare di B^n; dato che è lineare allora vale:

- se x,y ∈ C allora x + y ∈ C
- se x∈ C, a∈ B allora a . x ∈ C
- ∅ ∈ C

Definizione: La definizione di codice lineare è estendibile anche quando si ha un alfabeto B che è un campo finito:

- B è un insieme finito di elementi (es. {0,1})
- sono definite le operazioni di somma e prodotto e B è chiuso rispetto a tali operazioni (es. {+:XOR, ·:AND)
- ∅ ∈ B, cioè l'elemento neutro ∈ B

Ci limitiamo per ora al caso B={0,1}, ma quanto detto può essere fatto valere per qualsiasi campo finito B.

Definizione: Dato una parola x∈ C, si definisce **Peso di Hamming** di x e si indica con w(x) il numero delle cifre di x non nulle (non è altro che il numero di 1).

Definizione: Il Peso di Hamming del Codice C, w(C) è dato da:

$$w(C) = \min\{w(x) | x \in C, x \neq 0\}$$

Teorema: Il peso di Hamming del Codice C, w(C) è anche uguale al minimo numero delle colonne linearmente dipendenti di H.

dimostrazione: sia H una matrice di Hamming sia r il minimo numero di colonne linearmente dipendenti in posizione $i_1,...i_r$. Per definizione ∃ dei valori $a_1,...a_r$ tali che $a_1c_{i1}+...+ a_rc_{ir} = 0$ quindi ∃ una parola x: w(x) = r:

$$x = \begin{cases} a_i & \forall i = i_1,...i_r \\ 0 & \text{altrimenti} \end{cases}$$

se moltiplico x per la matrice ottengo

$$H \cdot x^T = a_1 \cdot c_1 + a_2 \cdot c_2 + ... + a_r \cdot c_r = 0$$

Inoltre r è anche il peso minimo delle parole di codice non nulle.

Teorema: Il peso del codice è uguale alla distanza di Hamming perché il codice è lineare:

$$w(C) = d(C) = \min_{x,y \in C} d(x, y)$$

dimostrazione: scegliamo una coppia x', y' a distanza minima, d(x',y') è il numero di simboli che cambiano da una parola all'altra:

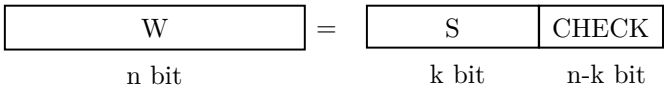
x' - y' → 010101 + → nell'aritmetica modulo 2 "+" e "-" si equivalgono
000110 =
010011 d(x',y')

$$d(C) = \min_{x,y \in C} d(x, y) = d(x', y') = w(x'+y') \geq w(c)$$
$$d(C) = \min_{x,y \in C} d(x, y) \leq \min(d(x,0)) = \min(w(x)) = w(c)$$

Per correggere "t" errori deve valere $d(c) \geq 2t+1$, quindi devono esistere almeno 2t+1 colonne linearmente dipendenti

3.5.2 Codifica & Decodifica

Data S la parola originale di k bit, come si ottiene la parola W da trasmettere?



Sia H=[A | I] dove I è la matrice identità di dimensione $(n-k)\times(n-k)$ allora posso scrivere una matrice di codifica G = [I | A^T] dove I è la matrice identità $k \times k$.

La **codifica** della parola S è eseguita moltiplicando S per G:

$$S \cdot G = W$$

In **decodifica** per verificare se la parola ricevuta sia corretta o meno si moltiplica H per la trasposta della parola ricevuta W^T:

$$(H \cdot W^T)$$

se il risultato ottenuto è zero la parola è corretta in caso contrario il risultato mi restituisce la sindrome .

Esempio:

$$H = \left[A \mid I\right] = \left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

$$G = \left[I \mid A^T\right] = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right]$$

Data la parola S =[0 1 0 1] abbiamo che la parola trasmessa è:

$$S \cdot G = w = \left[0 \;\; 1 \;\; 0 \;\; 1 \mid 1 \;\; 0 \;\; 1\right]$$

Se in ricezione non si verifica errore ottengo: $H \cdot w^T = 0 = \left[0 \;\; 0 \;\; 0\right]$

Questo è un codice di Hamming separabile (7,4) dove n=7 e k=4

3.6 Codici aritmetici

Definizione: Sia C(.) la funzione che mappa le parole informative in parole codificate, si dice che un codice è aritmetico se è vero che:

$$c(x \text{ op } y) = c(x) \text{ op' } c(y)$$

in cui op è un'operazione tra le parole (op' può essere uguale ad op ma non è detto)

3.6.1 Codice AN

Un classico esempio di codice aritmetico è il codice AN, questo tipo di codifica consiste nel moltiplicare ciascuna parola dei dati per una costante A. Questa codifica è invariante alle operazioni di addizione e di sottrazione, ma non alle operazioni di moltiplicazione e divisione.

In **codifica** la parola da trasmettere è ottenuta moltiplicando la parola informativa per la costante A.

In **decodifica** la verifica che la parola ricevuta w sia valida viene eseguita dividendo w per A e verificando che sia divisibile quindi per A, cioè che resto(w,A)= 0.

La scelta della costante A comporta il numero di extra-bit che verranno utilizzati e la capacità di rilevare gli errori, perché al crescere di A crescono il numero di bit necessari per la codifica e cresce la distanza di Hamming che sappiamo essere determinante per la rilevazione e correzione degli errori.
Quindi la scelta di A è un'operazione critica, infatti deve essere scelta in modo tale che non sia $A = 2^p$. Per capire le ragioni di questa limitazione dobbiamo ricordare che in base binaria la moltiplicazione per 2^p si traduce in una operazione di shift aritmetico verso sinistra del numero per cui il numero $a_{n-1}, \ldots, a_0$ diventerebbe $a_{n-1}, \ldots, a_0, 0, \ldots, 0$ con esattamente 2^p zeri.
La rappresentazione decimale quindi è data da:

$$a_{n-1}2^{\,p+n-1} + \cdots + a_0 2^p + (0)2^{\,p-1} + \cdots + (0)2^{\,0}$$

Questo numero è divisibile per 2^p. Si può notare inoltre che in caso di un'alterazione di uno solo dei coefficienti, il numero rimane comunque divisibile per 2^p. Quindi se $A = 2^p$ la codifica non è in grado di rilevare gli errori su

singolo bit. Se si sceglie il valore A come $A = 2^p - 1$ con $p \geq 2$ la codifica prende il nome di *low-cost residue*. In particolare scegliendo $A = 2^p - 1$ con $p \geq 2$ il numero di extra-bit per la codifica low-cost è pari a p.

Sia w la parola trasmessa se la parola è corretta si ottiene resto(w / A) = 0. Supponiamo che la parola ricevuta sia r sia diversa da w poiché il bit i-esimo è stato modificato quindi $r = w \pm 2^i$

$$\text{resto } (r / A) = S \quad \text{con} \quad S \neq 0 \quad \text{poiché A è dispari}$$

$$\text{resto } (r / A) = \text{resto}(\overset{0}{\cancel{w / A}}) \pm \text{resto}(2^i / A) \rightarrow S = \pm \text{resto } (2^i / A)$$

dove S rappresenta la sindrome e 2^i è l'errore sul bit i-esimo. Tale resto può assumere A valori diversi: da 0 (che indica assenza di errori) a $A - 1$, potrebbe quindi essere possibile localizzare errori singoli su una parola di $A - 1$ bit.

Esempio: Sia dato A = 3; ottengo ricordando che $S = \pm \text{resto}(2^i / A)$

Errore sul bit i-esimo	S	Effetti
0	0	Nessun Errore
2^0	1	Errore sul bit in posizione 0
2^1	2	Errore sul bit in posizione 1
2^2	1	Non distingue l'errore

Se ho un parola piccola di 2 bit, con A=3, posso correggere l'errore singolo.

Usare A significa aggiungere $\sup(\log_2 A)$ bit ai k bit di dati, quindi deve essere $A - 1 > \sup(\log_2 A) + k$. Occorrono quindi valori di A piuttosto alti, inoltre A deve essere primo con le potenze del due, cioè deve essere dispari.

A	k	n
13	2	6
19	4	9
23	6	11
29	9	14
37	12	18
…	…	…
101	42	50
…	…	…

Dove:

- k indica il numero di bit della parola di dati che posso correggere.
- n è il numero di bit totali della parola da trasmettere.

Da notare che nessuno degli A riportati nella tabella è della forma 2^p-1, dunque non è un "low cost residue".

3.7 Codici CRC (Controllo di Ridondanza Ciclico)

I codici CRC (*Cyclic Redundancy Check codes*) sono tutti codici a rilevazione di errore. Passiamo adesso all'introduzione del concetto di codice ciclico, che sarà fondamentale nella trattazione dei codici CRC:

Definizione: un codice a blocco binario lineare C(n, k) si dice ciclico se è invariante rispetto all'operatore di shift ciclico ρ, ovvero se per ogni parola di codice w $\in$ C, ρ (w) $\in$ C

Esempio 1 Il codice di Hamming C(7, 4) e' un codice ciclico:
Le 16 parole di codice sono: C = {0000000, 1000101, 0100111, 1100010, 0010110, 1010011, 0110001, 1110100, 0001011, 1001110, 0101100, 1101001, 0011101, 1011000, 0111010, 1111111}.

Indicando con la freccia la trasformazione indotta da ρ si ha:

- 0000000 → 0000000 (parola di partenza)
- 1000101 → 1100010 → 0110001 → 1011000 → 0101100 → 0010110 → 0001011 →1000101 (parola di partenza)
- 0100111 → 1010011 → 1101001 → 1110100 → 0111010 → 0011101 → 1001110 →0100111 (parola di partenza)
- 1111111 → 1111111 (parola di partenza)

E' comune rappresentare ogni parola w di un codice ciclico:

$$w = (c_{n-1}, c_{n-2}, \ldots, c_2, c_1, c_0) \in C$$

mediante un polinomio nell'indeterminata x a coefficienti in Z_2:

$$w(x) = c_0 + c_1\, x + c_2\, x^2 + \ldots + c_{n-1}\, x^{n-1}$$

Esempio 2: Dato il codice di Hamming C(7, 4) visto in precedenza si ha, ad esempio:

- 0000000 → $w_1(x) = 0$
- 1000101 → $w_2(x) = 1+x^2+x^6$
- 1111111 → $w_3(x) = 1+x+x^2+x^3+x^4+x^5+x^6$

Per i codici ciclici vale il seguente teorema fondamentale:
Teorema: dato un codice binario ciclico C(n, k):

- esiste un unico polinomio generatore G(x) di grado z = n − k che divide $1 + x^n$ (dove $1+x^n$ è un polinomio di n+1 cifre 10……01).

- I 2^k polinomi di grado non maggiore di n-1 del codice si ottengono come prodotto $w(x) = U(x) \cdot G(x)$, dove U(x) sono tutti i 2^k polinomi di grado minore o uguale a (k − 1) e G(X) è il polinomio generatore.

Esempio 3: Dato il codice di Hamming C(7, 4), si ha $G(x) = 1 + x + x^3$.
Per le parole dell'esempio precedente si ha (da ricordare che la somma è in modulo 2):

- per $w(x) = 0$, allora $U(x) = 0$
- per $w(x) = 1 + x^2 + x^6$, allora $U(x) = 1 + x + x^3$
- per $w(x) = 1 + x + x^2 + x^5$, allora $U(x) = 1 + x^2$
- per $w(x) = x + x^5 + x^6$, allora $U(x) = x + x^2 + x^3$

3.7.1 Codifica & Decodifica

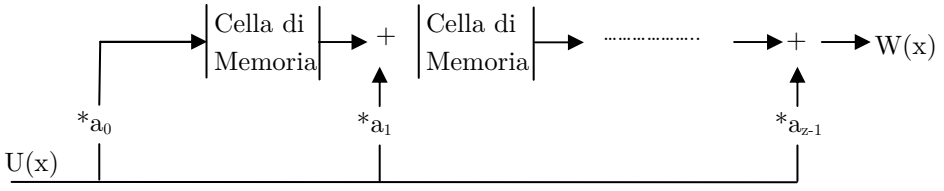
La **codifica** di una parola informativa W(x) è eseguita moltiplicando il polinomio associato U(x) di grado k-1 per il polinomio generatore G(x) di grado n-k:

$$W(x) = U(x) \cdot G(x)$$

Questo si può effettuare mediante un semplice circuito shift register retroazionato con z-1 celle, con le connessioni di retroazione presenti solo in corrispondenza dei coefficienti non nulli di G(x).

Tale shift register è un caso particolare di Macchina Lineare (Registro di spostamento Generalizzato).

$$W(x) = U(x).\ G(x) \text{ dove } G(x) = a_{z-1}\ .\ x^{z-1} + a_{z-2}\ .\ x^{z-2} + a_1\ .\ x + a_0$$



Dato che la parola U(x) data in ingresso viene modificata all'interno della macchina (non la ritrova più in uscita), il codice NON è SEPARABILE. Per ottenere un codice separabile, siccome vogliamo che i bit di ridondanza siano mantenuti separati da quelli del messaggio, shiftiamo U(x) a sinistra di z bit:

$$U(x) \cdot x^z$$

In fase di codifica per ottenere questa separabilità possiamo effettuare la divisione:

$$\frac{U(x) \cdot x^z}{G(x)} = Q(x) + \frac{R(x)}{G(x)}$$

dove G(x) è un polinomio di grado n-k, Q(x) è il quoziente di grado k-1 e R(x) è il polinomio resto di grado k-1.

La divisione è l'ordinaria divisione di un polinomio per un altro; la particolarità risiede nel fatto che i coefficienti di dividendo e di divisore sono binari e che l'aritmetica viene svolta modulo 2.

E' possibile formulare la seguente importante osservazione:
Dato il grado del polinomio generatore, il grado del polinomio resto R(x) è al più uguale a $z - 1$, conseguentemente R(x) può essere sempre rappresentato con z coefficienti (binari), ponendo uguali a 0 i coefficienti dei termini mancanti.

Se moltiplico entrambi i membri per G(x) ottengo:

$$U(x) \cdot x^z = Q(x) \cdot G(x) + R(x)$$

per cui:

$$U(x) \cdot x^z + R(x) = Q(x) \cdot G(x)$$

In questo modo si ottiene il polinomio da trasmettere, che può essere definito come:

$$W(x) = U(x) \cdot x^z + R(x)$$

dove:

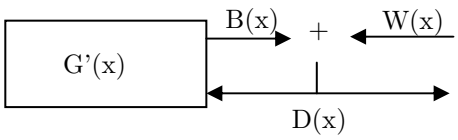
- U(x) polinomio di grado k-1 (k bit);
- x^z grado z (con z = n-k): la moltiplicazione mi produce lo spostamento a sinistra di z bit;
- R(x) polinomio di grado z-1 (z bit);
- W(x) polinomio di grado n-1 = $k - 1 + z$ (n bit).

Come si vede dalle formule precedenti è stato costruito un polinomio W(x) che è sempre divisibile per G(x) (la divisione produce un resto nullo). Il controllo dell'errore avviene nel ricevitore effettuando la divisione del polinomio ricevuto per il polinomio generatore. Nel caso di assenza di errori il resto di tale divisione dovrà essere nullo, in caso contrario il resto sarà diverso da zero.

La **decodifica** di una parola di codice ricevuta viene eseguita dividendo la parola ricevuta per il polinomio generatore:

$$U(x) = \frac{W(x)}{G(x)}$$

Il resto della divisione deve essere zero, altrimenti la parola ricevuta non è di codice e in questo caso il resto costituisce la sindrome. In particolare si utilizza la seguente macchina:



La macchina prende in ingresso la parola ricevuta W(x) ed esegue la divisione con G(x) = 1+ G'(x)

Dalla macchina possiamo ricavarci:

$$\begin{aligned} B(x) &= D(x) \cdot G'(x) \\ D(x) &= B(x) + W(x) \\ B(x) &= W(X) + D(x) \end{aligned}$$

segue che:

$$\begin{aligned} W(x) + D(x) &= D(x) \cdot G'(x) \\ D(x) \cdot (1 + G'(x)) &= W(x) \\ D(x) &= \frac{W(x)}{1 + G'(x)} \end{aligned}$$

Otteniamo l'uscita della macchina D(x).

Voglio che la parola ricevuta W(x) sia divisibile per G(x) ed il resto sia uguale a zero ottenendo D(x) della forma:

$$D(x) = \frac{W(x)}{G(x)} \cdot x^z = Q(x) \cdot x^z$$

Se W(x) non è divisibile per G(x) ottengo:

$$D(x) = Q(x) \cdot x^z + R(x)$$

dove Q(x) è il quoziente (k bit) ed R(x) è il resto (n-k bit)

Vediamo adesso più in dettaglio come questi concetti vengano utilizzati dal punto di vista sia dell'emettitore del messaggio da proteggere che del ricevitore.

3.7.2 L'emettitore

Ottenuto il resto R(x), l'entità emittente inserisce i coefficienti di questo polinomio in un apposito campo della stringa di bit (campo CRC), che deve quindi avere lunghezza z .
Nella stringa di dati emessa trovano quindi posto le cifre binarie da proteggere (in numero uguale a k) e le cifre CRC (in numero uguale a z): in totale k + z cifre binarie, che sono rappresentative di un polinomio W(x) di grado k +z −1 che costituiscono una parola di codice:

$$W(x) = x^z\, U(x) + R(x)$$

Tenendo conto che, per definizione:

$$x^z\, U(x) = Q(x)G(x) + R(x)$$

e poiché addizione e sottrazione modulo 2 si equivalgono, si ottiene:

$$W(x) = Q(x)G(x)$$

cioè la stringa emessa (rappresentativa del polinomio W(x)) è divisibile per il polinomio generatore G(x).

Si conclude che tutte le parole di codice sono divisibili per il polinomio generatore, e tutti i polinomi divisibili per G(x) sono parole di codice.

3.7.3 Il ricevitore

L'entità ricevente esegue, con il polinomio generatore, l'operazione di divisione effettuata in emissione; in questo caso opera però sul polinomio rappresentato dalle k + z cifre binarie ricevute.
Se nella trasmissione si sono verificati degli errori in ricezione si ottiene un polinomio:

$$W'(x) \neq W(x)$$

che può essere espresso come:

$$W'(x) = W(x) + E(x)$$

E(x) rappresenta il polinomio degli errori: ogni errore nella stringa di bit corrisponde ad un coefficiente non nullo in E(x); allora le cifre binarie ricevute rappresentano il polinomio W(x) + E(x) ,dove l'addizione è svolta modulo 2.
Ogni bit 1 in E(x) corrisponde ad un bit che è stato invertito e quindi a un errore isolato. Se ci sono n bit 1 in E(x), sono avvenuti n errori di un singolo bit; un singolo errore a raffica (burst, nel seguito useremo i due termini in maniera equivalente) di lunghezza n è caratterizzato in E(x) da un 1 iniziale, una mescolanza di 0 e 1, e un 1 finale per un complesso di n coefficienti binari:

$$E(x) = x^i(x^{n-1} + \ldots + 1)$$

dove i determina quanto la raffica è lontana dall'estremità destra della stringa di bit.
Il ricevitore calcola il resto della divisione di W(x) + E(x) per G(x); le modalità sono le stesse utilizzate nell'emettitore, ne segue che:

$$R\left(\frac{W'(x)}{G(x)}\right) = R\left(\frac{W(x)+E(x)}{G(x)}\right) = R\left(\frac{W(x)}{G(x)}\right) + R\left(\frac{E(x)}{G(x)}\right) = 0 + R\left(\frac{E(x)}{G(x)}\right) = R\left(\frac{E(x)}{G(x)}\right)$$

quindi si ha:

$$R\left(\frac{E(x)+W(x)}{G(x)}\right) = R\left(\frac{E(x)}{G(x)}\right)$$

Conseguentemente la regola applicata dal ricevitore è la seguente: se il resto della divisione (W(x) + E(x))/G(x) è nullo, la stringa di bit ricevuta è assunta senza errori; in caso contrario, si sono verificati uno o più errori nel corso del trasferimento.

Si nota che sono non rivelabili le configurazioni di errore per le quali il relativo polinomio E(x) contiene G(x) come fattore.

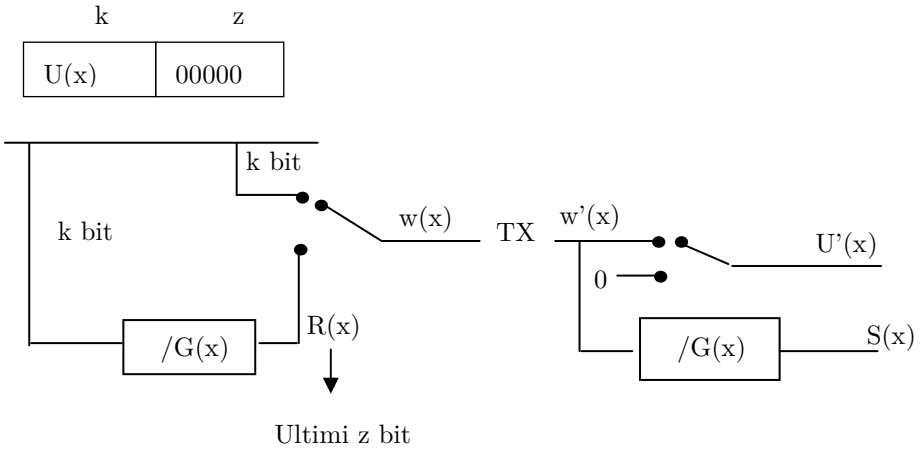


Figura 7: Emettitore e Ricevitore

Discutiamo adesso il comportamento di un codice CRC a seconda dei vari tipi di errore.

3.7.4 Rilevazione degli Errori

Un codice CRC, in cui il polinomio generatore contiene x + 1 come fattore primo, è in grado di rivelare:

- Errori Singoli
- Errori Dispari.
- Errori Doppi.
- Errori di Burst.

Errori Singoli

Sia E(x) il polinomio d'errore e G(x) il polinomio generatore si ha che la sindrome è il resto della divisione:

$$\text{resto}\left(\frac{w(x)+E(x)}{G(x)}\right) = \text{resto}\left(\frac{w(x)}{G(x)}\right) + \text{resto}\left(\frac{E(x)}{G(x)}\right) = 0 + \text{resto}\left(\frac{E(x)}{G(x)}\right)$$

Se il $\text{resto}\left(\frac{E(x)}{G(x)}\right) \neq 0 \Rightarrow$ rilevo l'errore E(x)

Il polinomio d'errore, nel caso di errore singolo, assume la forma $E(x) = x^j$ con $j=\{0,1,\ldots,n-1\}$ dove l'esponente indica la posizione del bit errato.
Per rilevare l'errore deve valere che E(x) non sia divisibile per G(x) $\forall$ j il che accade quando G(X) ha almeno due coefficienti non nulli (contiene perlomeno due 1).

Errori Dispari

Il polinomio E(x) nel caso di errori dispari avrà la seguente forma:

$$E(x) = x^{i+k+j} + x^{i+k} + x^i$$

cioè E(x) contiene un numero dispari di termini, il polinomio G(x) dovrà allora contenere il polinomio x + 1 come fattore primo, cioè:

$$G(x) = (x + 1)Y(x)$$

infatti non esiste alcun polinomio con numero dispari di termini che abbia (x+1) come fattore.
Riformulando in maniera rigorosa le seguenti osservazioni, possiamo affermare che: un codice ciclico C(n, k) con polinomio generatore G(x) che contiene (1 + x) come fattore è in grado di rivelare tutti i vettori di errore di peso dispari.

Infatti, un codice di parità C(n, n−1) è un codice ciclico, generato dal polinomio G(x) = 1 + x. Di conseguenza, se:

$$G(x) = f(x)(1 + x)$$

le parole del codice C sono del tipo:

$$c(x) = U(x)G(x) = U(x)f(x)(1 + x) = U(x)(1 + x)$$

E' quindi un sottocodice del codice di parità e contiene perciò solo parole di codice di peso pari. Tutti i vettori di errore di peso dispari vengono quindi sicuramente rivelati (il resto della divisione per G(x) non è zero).

Un'altra spiegazione alternativa è la seguente: se E(x) ha peso dispari non è divisibile per (1+x) perché non si annulla ponendo x = 1. Di conseguenza E(x) non è divisibile per G(x) ed ha sindrome non nulla.

Errori Doppi

Se ho un errore doppio il polinomio di errore assume la seguente forma:

$$E(x) = x^j + x^i = x^i \cdot (1 + x^{j-i}) = x^i \cdot (1 + x^h)$$

Il termine x^i non dà contributo significativo, mentre ci interessa vedere quando il termine $(1 + x^h)$ sia divisibile per G(x).

Non esiste alcuna regola generale che ci permetta di assicurare un resto diverso da zero, ma notando che h rappresenta la distanza tra i due errori, possiamo affermare di poter riuscire a rilevare due errori solo se sono posti ad una distanza inferiore ad h.

Proprietà: Un codice ciclico C(n, k) con polinomio generatore G(x) è in grado di rivelare vettori di errore di peso due se G(x) non divide $1+x^i$ con $i < n$..

Per progettare codici ciclici con questa proprietà si possono usare le proprietà dei polinomi primitivi: sono polinomi irriducibili di grado l tali che dividono $1+x^j$ per $j = 2^l - 1$ e mai per j inferiori (i polinomi primitivi sono noti e tabulati).

Ad esempio, un polinomio di grado 6 primitivo divide $1+x^{63}$, quindi va bene per rivelare errori doppi per $n \leq 62$.
Un polinomio di grado 7 primitivo divide $1 + x^{128}$, e va bene per $n \leq 127$.

Errori Burst

Gli errori di burst sono errori che interessano i bit che si trovano in un certo intervallo finito della parola ricevuta. L'esempio tipico è quello di un problema temporaneo sul canale di comunicazione: tanti errori ma tutti concentrati in una parte delimitata della parola.
Il polinomio E(x) nel caso di errori a burst avrà la seguente forma:

$$E(x) = x^{i} \cdot B_{b-1}(x)$$

Il termine x^i indica la posizione iniziale del burst all'interno della trama, mentre il termine B(x) (di grado b-1) rappresenta i bit sbagliati all'interno del burst e indica anche la lunghezza del burst stesso, essendo B di grado b-1 avremo che la lunghezza del burst è di b bit:

$$B_{b-1}(x) = a_{b-1} \cdot x^{b-1} + \cdots a_1 \cdot x + a_0$$

per la nostra definizione di burst, sicuramente $a_{b-1} = a_0 = 1$, mentre gli altri bit interni al burst (da a_{b-2} a a_1) possono essere uguali a 1 oppure a 0.

Dobbiamo verificare che il resto del secondo termine sia differente da zero:

$$\text{resto}\left(\frac{B_{b-1}(x)}{G(x)}\right) \neq 0$$

Se il grado di B(x) è minore al grado di G(x), allora sicuramente il resto della loro divisione sarà diverso da zero, e quindi rileveremo l'errore.

Se il grado di B(x) è uguale al grado di G(x), allora per avere resto nullo occorrerebbe che tutti i bit dei due polinomi fossero uguali, ma questo è assai poco probabile in quanto il burst è casuale. La probabilità che i due polinomi siano uguali è infatti soltanto $1/2^{r-1}$.

Nel caso in cui il grado di B(x) sia maggiore del grado di G(x) , la probabilità di non rilevare l'errore è pari a $1/2^{r}$.

3.7.5 Applicazioni Pratiche

Polinomi generatori

Sono standard i seguenti polinomi generatori:

$$G(x) = x^{16} + x^{12} + x^5 + 1$$

e:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + x^2 + x + 1$$

Entrambi sono divisibili per x+1 e quindi danno luogo a codici CRC con le proprietà descritte nel paragrafo precedente.

Codici CRC in applicazioni pratiche

Vediamo due applicazioni pratiche di codici CRC:

1. Standard CCSDS
2. Standard UMTS.

Standard CCSDS:

Lo standard CCSDS per la telemetria da spazio e da satellite utilizza un codice CRC con polinomio generatore:

$$G(x) = x^{16} + x^{12} + x^5 + 1$$

Questo polinomio genera dei codici CRC del tipo C(k + 16, k). Questi 16 bit di ridondanza consentono di rivelare sicuramente:

- tutti i burst di errore di lunghezza $l < 16$
- tutti i vettori di errore di peso dispari ((1 + x) divide G(x));

Standard UMTS

Lo standard UMTS per la terza generazione di telefonia cellulare prevede l'uso, a seconda del contesto, di 4 diversi codici CRC caratterizzati da ridondanza r pari a 8, 12, 16 e 24 bit, con polinomi generatori rispettivamente:

$$G_8(x) = 1 + x + x^3 + x^4 + x^7 + x^8$$
$$G_{12}(x) = 1 + x + x^2 + x^3 + x^{11} + x^{12}$$
$$G_{16}(x) = 1 + x^5 + x^{12} + x^{16}$$
$$G_{24}(x) = 1 + x + x^5 + x^6 + x^{23} + x^{24}$$

3.7.6 Codici CRC correttori

Fino a qui abbiamo parlato dei codici CRC come rilevatori di errori, vediamo cosa si può dire sulla correzione.
I codici CRC riescono a correggere soltanto errori singoli, necessità di una maggiore ridondanza rispetto alla semplice rilevazione. La correzione è possibile in particolare se si utilizza un codice CRC per il quale si realizza una corrispondenza uno a uno tra la sindrome, cioè il polinomio risultante da resto(E(x)/G(x)) e il polinomio di errore E(x).

$$S(x) = \text{resto}\left(\frac{w(x)}{G(x)}\right) = \text{resto}\left(\frac{E(x)}{G(x)}\right)$$

Localizzo l'errore solo se :

$$\forall E(x), E'(x) \in C : E(x) \neq E'(x), E(x) \neq 0, E'(x) \neq 0$$

Devo trovare G(x) che mi verifica quella proprièt̀a.

$$\text{resto}\left(\frac{E(x)}{G(x)}\right) \neq \text{resto}\left(\frac{E\ '(x)}{G(x)}\right)$$

Prendiamo ad esempio un codice di Hamming C(7,4) dove le parole sono codificate su 7 bit a partire da parole informative di 4 bit, e prendiamo $G(x) = 1 + x + x^3$.

Supponiamo di avere :

$E(x) = x^6$ (errore su settimo bit 1000000) $\Rightarrow \text{resto}\left(\frac{E(x)}{G(x)}\right) = 101 = x^2 + 1$

$E(x) = x^5$ (errore su sesto bit 0100000) $\Rightarrow \text{resto}\left(\frac{E(x)}{G(x)}\right) = 111 = x^2 + x + 1$

$E(x) = x^4$ (errore su quinto bit 0010000) $\Rightarrow \text{resto}\left(\frac{E(x)}{G(x)}\right) = 110 = x^2 + x$

$E(x) = x^3$ (errore su quarto bit 0001000) $\Rightarrow \text{resto}\left(\frac{E(x)}{G(x)}\right) = 011 = x + 1$

$E(x) = x^2$ (errore su terzo bit 0000100) $\Rightarrow \text{resto}\left(\frac{E(x)}{G(x)}\right) = 100 = x^2$

$E(x) = x$ (errore su secondo bit 0000010) $\Rightarrow$ resto$\left(\frac{E(x)}{G(x)}\right) = 010 = x$

$E(x) = 1$ (errore su primo bit 0000001) $\Rightarrow$ resto$\left(\frac{E(x)}{G(x)}\right) = 001 = 1$

Ho sette resti diversi per i sette errori singoli dunque $G(x) = 1 + x + x^3$ è un codice **CRC correttore** per errori singoli. Quindi in questo caso ho una corrispondenza biunivoca tra sindrome e il polinomio errore.

3.8 Codici di Solomon

Al fine di capire i principi di codifica e decodifica dei **codici non binari**, come *Reed-Solomon*, è necessario introdurre il concetto dei campi finiti e campi di *Galois* (GF).

3.8.1 Introduzione

Definizione di Gruppo:
Un gruppo (G, $\cdot$) è un insieme di elementi tra i quali è definita un'operazione binaria che chiamiamo prodotto che soddisfa le seguenti proprietà:

- Chiusura: se a,b $\in$ G allora a $\cdot$b $\in$ G.
- Proprietà Associativa: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- Esistenza dell'elemento neutro: $\exists e \in G : a \cdot e = a \quad \forall a \in G$
- Esistenza inverso: $\forall a \in G \quad \exists b \in G : a \cdot b = e$

Definizione: Un gruppo si dice Abeliano se vale anche la proprietà commutativa:

$$a \cdot b = b \cdot a$$

Definizione di Campo:
Un campo (F, +, $\cdot$) è un'insieme di elementi su cui sono definite due operazioni binarie dette somma e prodotto tali che:

- L'insieme F è un gruppo abeliano rispetto alla somma;
- L'insieme F è un gruppo abeliano rispetto al prodotto se non si considera l'elemento neutro (F-{0});
- La moltiplicazione è distributiva rispetto alla somma, cioè :

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

Esempi semplici di campi sono gli insiemi $\mathbb{R}$(reali), $\mathbb{Q}$(razionali), $\mathbb{C}$(complessi) rispetto alle operazioni di somma e prodotto.

Definizione di Campo di Galois:
E' un campo finito, denotato $GF(2^m)$ con 2^m numero degli elementi contenuti nel campo:

$$GF(2^m) = \left\{ 0, \alpha^0, \alpha^1, \alpha^2, \cdots, \alpha^{2^m-2} \right\}$$

Ogni elemento diverso da zero in $GF(2^m)$ può essere rappresentato da una potenza di α, ove α è un particolare elemento di $GF(2^m)$. Si può definire un insieme di infiniti elementi, F, a partire dagli elementi $\{0,1, \alpha\}$ e generato progressivamente moltiplicando l'ultimo valore per α, ottenendo:

$$F = \left\{ 0,1,\alpha^1,\alpha^2,\cdots,\alpha^j,\cdots \right\} = \left\{ 0,\alpha^0,\alpha^1,\alpha^2,\cdots,\alpha^j,\cdots \right\}$$

Per ottenere un insieme finito di $GF(2^m)$ da F, si impone la condizione che F debba contenere solo 2^m elementi e sia chiuso rispetto all'operazione di moltiplicazione.

$$\alpha^{2^m-1} - 1 = 0 \tag{4.1}$$

oppure:

$$\alpha^{2^m-1} = 1 = \alpha^0 \tag{4.2}$$

Usando questo vincolo polinomiale, ogni elemento che ha una potenza maggiore o uguale a 2^m-1 può essere ricondotto a un elemento con una potenza minore di $2^m - 1$, come segue:

$$\alpha^{2^m+n-1} = \alpha^{2^m-1} \cdot \alpha^n = \alpha^n, \quad n \geq 0$$

Le equazioni (4.1) e (4.2) possono essere usate per creare la sequenza finita F* dalla sequenza infinita F:

$$F = \left\{ 0,1,\alpha^1,\alpha^2,\cdots,\alpha^{2^m-2},\alpha^{2^m-1},\alpha^{2^m},\cdots \right\} = \left\{ 0,\alpha^0,\alpha^1,\alpha^2,\cdots,\alpha^{2^m-2},\alpha^0,\alpha^1,\alpha^2,\cdots \right\}$$

Quindi:

$$GF(2^m) = \left\{ 0,\alpha^0,\alpha^1,\alpha^2,\cdots,\alpha^{2^m-2} \right\}$$

3.8.2 L'operazione di somma nel campo esteso GF(2^m)

Ognuno dei 2^m elementi del campo finito, GF(2^m), può essere rappresentato come un polinomio di grado minore o uguale $(m-1)$. Indichiamo quindi ogni elemento diverso da zero di GF(2^m) come un polinomio, $a_i(x)$, dove almeno uno dei coefficienti di $a_i(x)$ è diverso da zero:

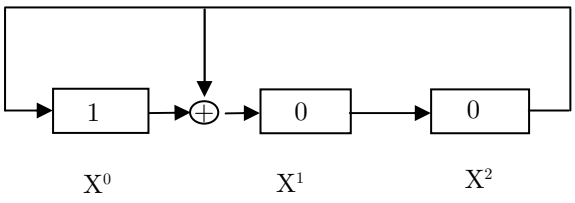
$$\alpha^i = a_i(x) = a_{i,m-1} \cdot x^{m-1} + \ldots + a_{i,1} \cdot x + a_{i,0} \quad \forall i = 0,1,2 \ldots, 2^m - 2 \qquad (4.3)$$

Consideriamo il caso di $m = 3$: il campo finito si indica con GF(2^3). La tabella sottostante mostra la mappatura dei sette elementi $\{\alpha^i\}$ e l'elemento nullo, in termini degli elementi base $\{x^0, x^1, x^2\}$ descritti dall'equazione (4.3):

	X^2	X^1	X^0
0	0	0	0
α^0	0	0	1
α^1	0	1	0
α^2	1	0	0
α^3	0	1	1
α^4	1	1	0
α^5	1	1	1
α^6	1	0	1

Il polinomio α^1 sarà uguale a $\alpha^1 = x^1$
Il polinomio α^2 sarà uguale a $\alpha^2 = x^2$
Il polinomio α^3 sarà uguale a $\alpha^3 = x^1 + 1$
E così via.

I coefficienti α possono essere ricavati attraverso la seguente macchina:



Uno dei vantaggi dell'uso degli elementi $\{\alpha^i\}$ al posto degli elementi non binari è la notazione compatta che facilita la rappresentazione matematica della codifica e decodifica non binaria.
L'addizione di due elementi del campo finito è definita come la somma modulo-2 di ciascuno dei coefficienti del polinomio (della stessa potenza):

$$\alpha^i + \alpha^j = (a_{i,0} + a_{j,0}) + (a_{i,1} + a_{j,1})x + (a_{i,m-1} + a_{j,m-1})x^{m-1}$$

Vediamo la tabella della somma:

+	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	0	α^3	α^6	α^1	α^5	α^4	α^2
α^1	α^3	0	α^4	α^0	α^2	α^6	α^5
α^2	α^6	α^4	0	α^5	α^1	α^3	α^0
α^3	α^1	α^0	α^5	0	α^6	α^2	α^4
α^4	α^5	α^2	α^1	α^6	0	α^0	α^3
α^5	α^4	α^6	α^3	α^2	α^0	0	α
α^6	α^2	α^5	α^0	α^4	α^3	α	0

3.8.3 I polinomi primitivi

E' possibile vedere un Campo di Galois come un oggetto in cui i valori sono generati da un Polinomio Primitivo.

Definizione di irriducibilità: Un polinomio p(x) di grado m su un certo dominio GF(2^m) (Campo di Galois) è irriducibile quando il polinomio p(x) non è divisibile per alcun polinomio di grado maggiore di 0 e minore di m.

Definizione di primitivo: Un polinomio p(x) di grado m irriducibile si dice primitivo se il minimo intero positivo n per il quale divide x^n+1 è $n = 2^m-1$

Esempio:

- $1+x+x^3$ è primitivo perché è irriducibile e divide x^7+1, $(2^3-1)=7$, e non divide x^n+1 $\forall n$ tale che $1 \leq n < 7$. Quindi $1 + x + x^3$ è un polinomio primitivo.

- $1+x+x^4$ è primitivo perché è irriducibile e divide $x^{15}+1$, e non divide x^n+1 $\forall n$ t.c $1 \leq n < 15$. Quindi $1 + x + x^4$ è un polinomio primitivo.

3.8.4 Codifica

Convenzionalmente un codice Reed-Solomon si indica con la sigla RS(n,k) ed esiste per valori di n e k tali che:

$$0 < k < n < 2^m + 2$$

dove k è il numero di simboli da codificare e n è il numero totale di simboli in un blocco di codifica; normalmente:

$$(n, k) = (2^m - 1, 2^m - 1 - 2t)$$

dove t è la capacità correttiva del codice e n - k = 2t è il numero di simboli di parità.
Indichiamo il messaggio da trasmettere con il polinomio di grado k − 1:

$$m(x) = m_0\, x^0 + m_1 x + \ldots + m_{k-1} x^{k-1}$$

I cui coefficienti m_0, m_1, . . ., m_{k-1} appartengono al campo finito $GF(p^r)$ (generalmente p è scelto uguale a 2).
Consideriamo un primo metodo di codifica e decodifica.
Il metodo di codifica consiste nel valutare il polinomio

$$m(x) = m_0\, x^0 + m_1 x + \ldots + m_{k-1} x^{k-1}$$

in n = k + 2t punti distinti non nulli e trasmettere tali valori. Notare che $n \leq p^r - 1$ (numero di elementi non nulli nel gruppo), altrimenti non abbiamo sufficienti punti in cui valutare m(x).

Il metodo di decodifica si basa sull'interpolazione polinomiale. Per evitare confusione dimentichiamo il fatto che i coefficienti non derivano da $GF(p^r)$ ma che appartengano a un qualche campo astratto. Supponiamo di avere k valutazioni del polinomio:

$$m(x_0) = y_0, \ldots, m(x_{k-1}) = y_{k-1}$$

Vogliamo ricostruire il polinomio di grado k - 1, interpolando questi punti.
Probabilmente il modo più semplice è il metodo di *Lagrange*:

$$m(x) = \sum_{i=0}^{k-1} y_i \prod_{j=0.j\neq i}^{k-1} \frac{x - x_j}{x_i - x_j}$$

Notare che:

$$\prod_{j=0.j\neq i}^{k-1} \frac{x - x_j}{x_i - x_j} = \begin{cases} 1 \text{ se } x = x_i \\ 0 \text{ se } x = x_j \wedge i \neq j \end{cases}$$

Si può verificare facilmente che tale polinomio interpola y_0, y_1, . . ., y_{k-1}.

C'è un altro approccio per trovare i coefficienti del polinomio. L'interpolazione può essere vista come la soluzione di un sistema di k equazioni in k incognite:

$$\begin{cases} m_0 + m_1 \cdot x_0 + \cdots + m_{k-1} \cdot x_0^{k-1} = y_0 \\ m_0 + m_1 \cdot x_1 + \cdots + m_{k-1} \cdot x_1^{k-1} = y_1 \\ \vdots \\ m_0 + m_1 \cdot x_{k-1} + \cdots + m_{k-1} \cdot x_{k-1}^{k-1} = y_{k-1} \end{cases}$$

Supponiamo ora di avere n = k + 2t valutazioni del polinomio y_0,….y_n, posso quindi scrivere un sistema di n equazioni. Qualunque sottosistema di k equazioni darà lo stesso risultato, ovvero vi saranno $\binom{n}{k}$ soluzioni coincidenti.
Se consideriamo la possibilità di introdurre errori nelle n valutazioni, allora avremo la non unicità delle soluzioni. Se si ammette un massimo di t errori avremo però un sottoinsieme di almeno k+t equazioni soddisfatte della stessa ed unica k-upla (m_0, m_1, . . ., m_{k-1}). La decodifica consiste quindi nella ricerca di tale sottoinsieme; per la sua natura combinatoria, questo metodo è troppo oneroso dal punto di vista della complessità computazionale.

Il metodo usato è, invece, il metodo sistematico. Invece di n valutazioni di m(x), inviamo k coefficienti di m(x) (che sono i coefficienti del messaggio originale) insieme a un insieme di 2t simboli di parità (simboli di check). Il motivo è migliorare in efficienza nel caso comune in cui la trasmissione è priva di errori: in questo caso, vorremmo limitare l'elaborazione computazionale per verificare l'assenza di errori, e quindi leggere semplicemente il messaggio senza decodifica.
Supponiamo che i 2t simboli di controllo della parità siano valutazioni di m(x) in 2t punti distinti (non nulli) x_0, x_1, . . ., x_{2t-1}.

Si vuole ricostruire m(x) dati k coefficienti e 2t valutazioni. Si possono scrivere 2t equazioni lineari:

$$\begin{aligned} &m_0 + m_1 \cdot x_0 + \cdots + m_{k-1} \cdot x_0^{k-1} = y_0 \\ &m_0 + m_1 \cdot x_1 + \cdots + m_{k-1} \cdot x_1^{k-1} = y_1 \\ &\vdots \\ &m_0 + m_1 \cdot x_{k-1} + \cdots + m_{k-1} \cdot x_{2t-1}^{k-1} = y_{2t-1} \end{aligned}$$

Analogamente a prima, ci deve essere un insieme consistente di k + t simboli. I coefficienti di m(x) appartengono al campo finito $GF(p^r)$.
Sia m(x) un polinomio primitivo di $GF(p^r)$ e sia g(x) il polinomio generatore così definito:

$$g(x) = (x - \alpha^1)(x - \alpha^2) \ldots (x - \alpha^{2t})$$

Useremo g(x) per mappare i messaggi di k simboli sui 2t simboli di parità.
Definiamo:

$$b(x) = (x^{2t} \cdot m(x)) \cdot \mod\{g(x)\}$$

$$b(x) = \text{resto}\left(\frac{m(x) \cdot x^{2t}}{g(x)}\right)$$

Quindi esiste q(x) tale che:

$$x^{2t} \cdot m(x) = q(x)g(x) + b(x)$$

Da cui si può definire il polinomio che rappresenta la codifica:

$$c(x) = x^{2t} \cdot m(x) - b(x) = x^{2t} \cdot m(x) + b(x)$$

Ciò significa che il codice è della forma:

$$m_{k-1}, \ldots, m_0, b_{2t-1}, \ldots, b_0$$

dove m sono i k simboli e b sono i 2t simboli di check.
Si è quindi costruito il polinomio della parola da trasmettere c(x) :

$$c(x) = x^{2t} \cdot m(x) - b(x) = [q(x)g(x) + b(x)] - b(x) = q(x)g(x)$$

dove c(x) è multiplo di g(x).

Per implementare la codifica sistematica per il codice RS(n,k) si utilizza il circuito LFSR. Per esempio nella Figura 8 è raffigurato un circuito LFSR per un codice RS(7,3) descritto da $g(x) = \alpha^3 + \alpha^1 x + \alpha^0 x^2 + \alpha^3 x^3 + x^4$ (in grado di correggere fino a due simboli scorretti).

Si può facilmente verificare che i termini moltiplicatori (da sinistra a destra) corrispondono ai coefficienti del polinomio g(x) (in ordine crescente). Ovviamente le parole di codice (non zero) sono costituite da $2m - 1 = 7$ simboli e ogni simbolo è fatto di $m = 3$ bit.

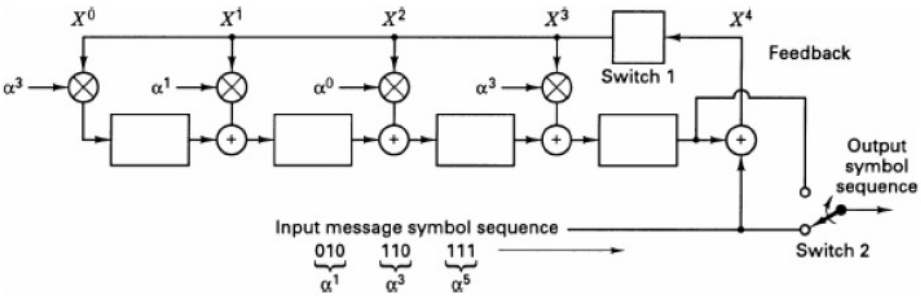


Figura 8: Codificatore LFSR per il codice RS(7,3)

Il codice è non binario, perciò ogni stadio dello shift register contiene 3 bit. Nel caso di codici binari, i coefficienti indicati con g1, g2, . . . sono binari. Quindi, essi hanno valori 1 e 0: indicano perciò presenza o assenza di connessione elettrica nel LFSR. In figura, ogni coefficiente è specificato con 3 bit e quindi può assumere 8 valori.

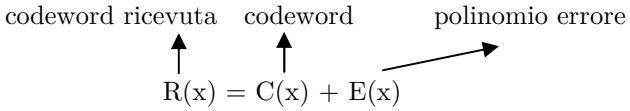
L'operazione non binaria implementata dall'encoder in Figura 8 è fatta in modo analogo al caso binario e i passi possono essere descritti come di seguito:

1. L'interruttore 1 (switch 1) è chiuso durante i primi k clock per permettere lo shift dei simboli del messaggio negli stadi dello shift register.
2. L'interruttore 2 (switch 2) è in posizione inferiore per i primi k clock, per permettere il simultaneo trasferimento dei simboli del messaggio in un registro di uscita (non rappresentato in Figura 8).
3. Dopo il trasferimento del k-esimo simbolo sul registro di uscita, l'interruttore 1 è aperto e l'interruttore 2 è spostato nella posizione superiore.
4. I rimanenti n - k clock permettono lo spostamento dei simboli di parità contenuti nello shift register verso il registro di uscita.

5. Il numero totale di clock è pari a n e il registro di uscita contiene la codeword $p(x) + x^{n-k}m(x)$, dove $p(x)$ rappresenta i simboli di parità e $m(x)$ i simboli del messaggio in forma polinomiale.

3.8.5 Decodifica

Indichiamo la *codeword* ricevuta con il polinomio ricevuto $R(x)$; esso può essere scomposto come:



Sia $E(x) = E_0 + E_1x + \ldots + E_{n-1}x^{n-1}$ l'espansione del polinomio errore.
Assumiamo che ci siamo al più t errori. Quindi al più t dei coefficienti E_i sono non nulli. Si può assumere senza perdita di generalità che esattamente s dei coefficienti sono non nulli: non è difficile estendere la trattazione seguente al caso in cui ci sono meno di t errori.
Siano $j_1, j_2, \ldots, j_s$ le posizioni degli errori, dove $j_i = 0, 1, \ldots, n-1$.

Definizione: La locazione dell'errore X_i è definita come $X_i = \alpha_{ji}$.

Quindi, se α è il generatore di $GF(p^r)$, dato un qualunque X_i, è possibile determinare l'unico valore j_i tale che $X_i = \alpha_{ji}$ attraverso il logaritmo discreto di X_i usando la base α. Quindi le locazioni degli errori $X_1, X_2, \ldots, X_t$ sono un altro modo di rappresentare gli indici degli errori. Occorre notare che il logaritmo discreto è un'operazione complessa: in realtà alla fine non verrà utilizzato. Sono state introdotte le locazioni degli errori X_i (trattazione del tutto equivalente) perchè si dimostra che sono più convenienti computazionalmente rispetto agli indici espliciti.

Definizione: L'ampiezza dell'errore Y_i è definita come E_{ji} , cioè è il coefficiente non nullo, corrispondente, nel polinomio errore.

La Figura 9 illustra **l'architettura di decodifica**. In ingresso è presente la codeword ricevuta R(x) e in uscita la codeword corretta C(x), assumendo al più t errori. In seguito sono esaminati in particolare i vari livelli.

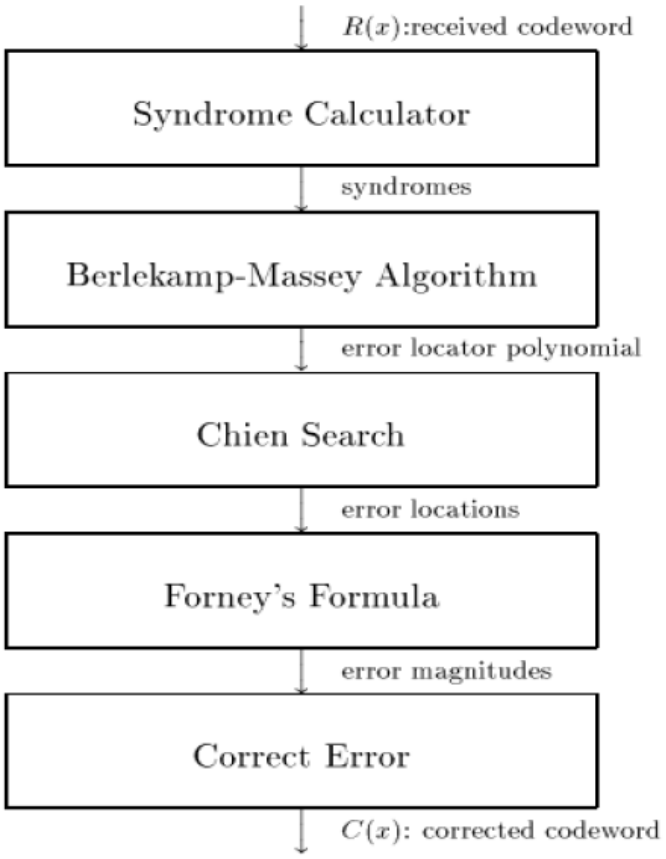


Figura 9: Architettura di decodifica

Calcolo della sindrome.

Il primo passo nella decodifica di un messaggio ricevuto R(x) è **calcolare la sindrome.**

Definizione La sindrome s_n, $\forall\ 1 \leq n \leq\ 2t$, del messaggio ricevuto è il polinomio R(x) valutato in α^n, cioè $s_n = R(\alpha^n)$.

Ora $C(\alpha^n) = 0$, $s_n = R(\alpha^n) = E(\alpha^n)\ \forall\ 1 \leq n \leq\ 2t$. Quindi le 2t sindromi permettono di avere 2t distinte valutazioni di E(x). Ricordiamo che E(x) è un polinomio con al più t coefficienti non nulli. Andremo a calcolare E(x) attraverso queste valutazioni.
Vale la seguente relazione:

$$s_n = R(\alpha^n) = \sum_{i=1}^{t} Y_i \cdot \alpha^{n_{ji}} = \sum_{i=1}^{t} Y_i \cdot x_i^n$$

Si può adesso definire il polinomio sindrome:

$$s(z) = \sum_{i=1}^{\infty} s_i \cdot z^i$$

Notare che s(z) ha un insieme infinito di termini e la relazione $s_n = R(\alpha^n) = E(\alpha^n)$ permette solo di calcolare i primi 2t coefficienti.

Algoritmo di Berlekamp-Massey

Avendo calcolato la sindrome, dobbiamo ora trovare il polinomio error locator tramite l'algoritmo di *Berlekamp-Massey*, non trattato a lezione in quanto troppo complesso.

Definizione: Il polinomio error locator $\sigma(z)$ è definito come:

$$\sigma(z) = \prod_{i=1}^{t} (1 - X_i z)$$

Notare che le radici di $\sigma(z)$ sono precisamente il reciproco delle locazioni degli errori, cioè $1/X_1, 1/X_2, \ldots, 1/X_t$;

Ricerca di Chien

Avendo risolto $\sigma(z)$, vogliamo ora calcolare le locazioni degli errori X_i.
Per definizione, il polinomio error locator :

$$\sigma(z) = \prod_{i=1}^{t} (1 - X_i z)$$

Le radici di $\sigma(z)$ sono precisamente il reciproco delle locazioni degli errori, cioè $1/X_1$, $1/X_2$, . . ., $1/X_t$; quindi calcoleremo le radici di $\sigma(z)$ e ne prenderemo il reciproco. In realtà tale approccio risulta inefficiente.
Il metodo di *Chien* prevedere di numerare semplicemente gli elementi del campo finito e verificare se il polinomio si annulla in ciascuno di essi.

L'algoritmo di Chien si può così riassumere:

1. Sia α un elemento generatore di $GF(p^r)$.
2. Inizializzare X_i all'insieme vuoto.
3. $\forall$ n = 1, 2, . . . se $\sigma(\alpha^n) = 0$, aggiungi α^{-n} all'insieme X_i.

Se $\sigma(\alpha^n) = 0$, allora α^n è radice di $\sigma(z)$, cioè α^{-n} è una locazione di errore. Notare che l'algoritmo di Chien fornisce gli indici in cui si verificano gli errori j_1, j_2, . . ., j_s direttamente. Quando inseriamo α^{-n} all'insieme X_i, si può aggiungere anche n a j_n.

Si può calcolare $\sigma(\alpha^{n+1})$ da $\sigma(\alpha^n)$ velocemente, infatti:

$$\sigma(z) = 1 + \sigma_1 z + \sigma_2 z^2 + \sigma_3 z^3 + \ldots + \sigma_t z^t$$

Quindi

$$\sigma(\alpha^n) = 1 + \sigma_1 \alpha^n + \sigma_2 \alpha^{2n} + \sigma_3 \alpha^{3n} + \ldots + \sigma_t \alpha^{tn}$$

$$\sigma(\alpha^{n+1}) = 1 + \sigma_1 \alpha^{n+1} + \sigma_2 \alpha^{2n+2} + \sigma_3 \alpha^{3n+3} + \ldots + \sigma_t \alpha^{tn+t}$$

Perciò il termine i-esimo di $\sigma(\alpha^{n+1})$, può essere calcolato dal termine i-esimo di $\sigma(\alpha^n)$ attraverso il prodotto per α^n.

Formula di Forney

Adesso rimangono da calcolare Y_i ovvero l'ampiezze dell'errore. La formula di *Forney* ci dice che se valutiamo $\omega(z)$ in X_n^{-1} abbiamo:

$$\omega(X_n^{-1}) = \sigma(X_n^{-1}) + \sum_{i=1}^{t} X_n^{-1} X_i Y_i \prod_{j=1, j\neq i}^{t} (1 - X_i X_n^{-1})$$
$$= Y_i \prod_{j=1, j\neq i}^{t} (1 - X_i X_n^{-1})$$

Dove si è sfruttato il fatto che $\sigma(X_n^{-1}) = 0$
Si può ora calcolare Y_n:

$$Y_n = \frac{\omega(X_n^{-1})}{\prod_{j=1, j\neq i}^{t} (1 - X_i X_n^{-1})}$$
$$= \frac{X_n^t \cdot \omega(X_n^{-1})}{X_n^t \cdot \prod_{j=1, j\neq i}^{t} (X_n \cdot X_j)}$$
$$= \frac{\varpi(X_n)}{X_n \cdot \prod_{j=1, j\neq i}^{t} (X_n \cdot X_j)}$$

dove $\varpi(z) = z^t \omega(1/z)$.

Quindi, l'algoritmo di Chien restituisce gli indici degli errori, la formula di Forney l'ampiezza dell'errore. Abbiamo così la soluzione del polinomio errore E(x). Si può adesso calcolare, per completare la decodifica, la codeword C(x) usando la formula $C(x) = R(x) - E(x)$.

3.8.6 Applicazioni dei Codici di Solomon

CD

La codifica Reed-Solomon è ampiamente usata nei sistemi di archiviazione di massa per correggere errori burst, associati a difetti del supporto.
La codifica R-S è un componente fondamentale nei compact disc (CD) e nei dispositivi ottici simili. Nei CD-audio sono presenti due livelli di codifica R-S su GF(28), separati da un *cross-interleaver* chiamato *Cross-Interleaved Reed Solomon Coding* (CIRC), come mostrato in Figura 10. La prima codifica è RS(28,24), la seconda è RS(32,28), ciò comporta una ridondanza $\mathrm{R} = \frac{32}{24} = \frac{4}{3}$.
Il codice più esterno può correggere fino a errori di 2 byte su blocchi di 32 byte, errori maggiori vengono segnalati. Grazie al deinterlacciamento, la cancellazione di uno blocco da 28 byte più interno diventa un singolo errore in ognuno dei 28 blocchi più esterni, quindi correggibile.
Con questi accorgimenti sono possibili correzioni di errori fino a 4000 bits consecutivi, corrispondenti ad un graffio sulla superficie di 2.5 mm.

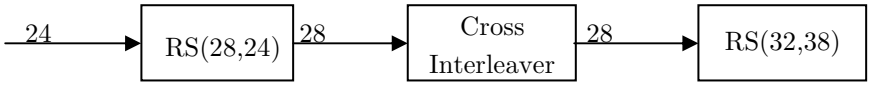


Figura 10: Codifica RS nei CD-audio

Per i CD-dati, per cui è richiesto un livello di protezione superiore, si premette un ulteriore livello di codifica con un codice rivelatore (CRC) con 4 byte di parità, ed il prodotto di due R-S su GF(256) accorciati RS(26,24) e RS(45,43).

DVD

La tecnologia DVD si basa su una maggiore densità dei dati rispetto alla tecnologia CD. Questo significa che la correzione dell'errore deve essere più accurata di quella usata per i CD. La necessità principale è correggere, o meglio impedire, i burst errors. Per ottenere questo risultato, i dati vengono posizionati in una matrice di 192 righe e 172 colonne, successivamente vengono aggiunti 16 byte di PO-parity ad ogni colonna e 10 byte di PI-parity ad ognuna delle 208 righe (192 di codice e 16 di PO), così da formare un Reed-Solomon Product Code da 208 righe e 182 colonne, visibile nella Figura 11.

Questo codice riesce a correggere al massimo 5 byte errati in ogni riga e 8 in ogni colonna.

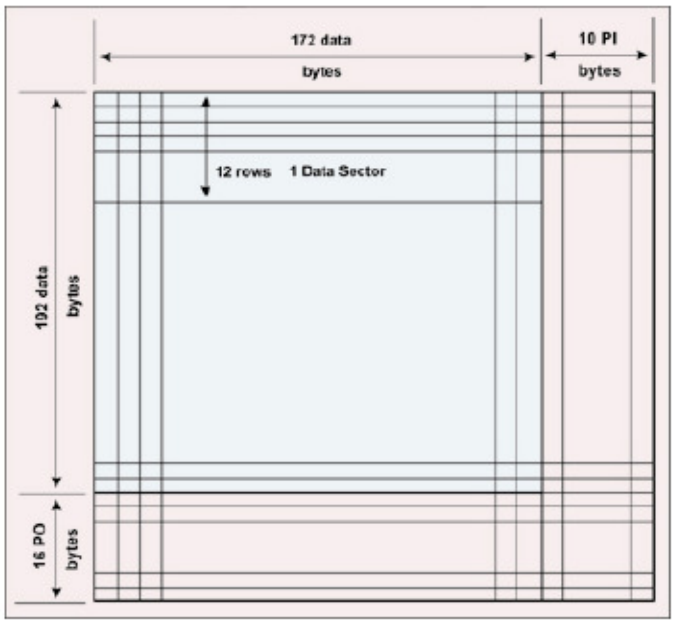


Figura 11: DVD *ECC block Product Code* RS(208,192)*RS(182,172)

Infine le righe PO sono interlacciate con le righe dati in rapporto 1:12 e ogni blocco ECC interlacciato è diviso in 16 settori di registrazione, evidenziati in Figura 12. In questo modo ogni settore di registrazione contiene i dati originali, più le informazioni di correzione PI/PO, formando un blocco di 2366 byte.

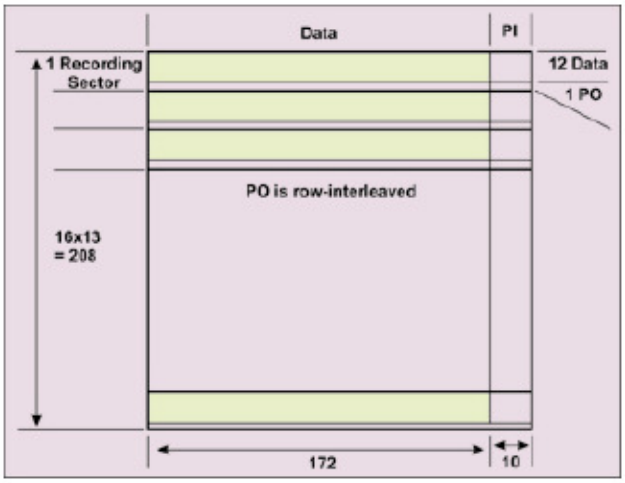


Figura 12: DVD: *Recording Sector*

4. Algoritmi distribuiti

4.1 Introduzione

Esistono due tecniche di *Recovery*:

1. ***Forward error recovery***: il sistema viene riportato ad uno stato noto come "*safe*" (il *reset* è un caso particolare: stato iniziale)
2. ***Backward error recovery***: il sistema viene riportato ad uno stato "corretto" in cui è già transitato.

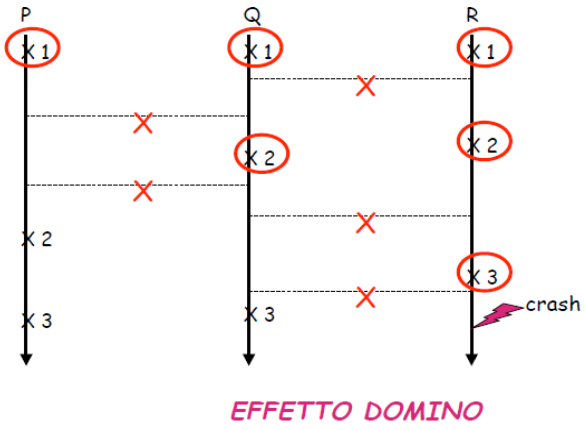
Il reset è un caso particolare delle procedure di recovery (recupero) a fronte di un guasto di una macchina singola. In caso di guasto (crash) la macchina viene fatta ripartire, ma nel caso di computazioni "*stateful*" dovremo essere in grado di ripristinarne lo stato.

Nel *Backward* occorre perciò memorizzare lo stato di tanto in tanto su memoria non volatile (Checkpoint), in una memoria stabile.
La granularità del checkpointing dipende dai valori di disponibilità attesi (quanta computazione possiamo permetterci di perdere).

Il lavoro svolto tra un checkpoint e il successivo (insieme di operazioni), viene considerato, per quanto riguarda i guasti, **un'azione atomica**, che gode della proprietà tutto o niente. Questo concetto è molto simile a quello di transizione su un data base (DB).
Un'azione atomica deve soddisfare alcune proprietà, note come ACID *property* (da Atomicità, Consistenza, Isolamento, Durabilità):

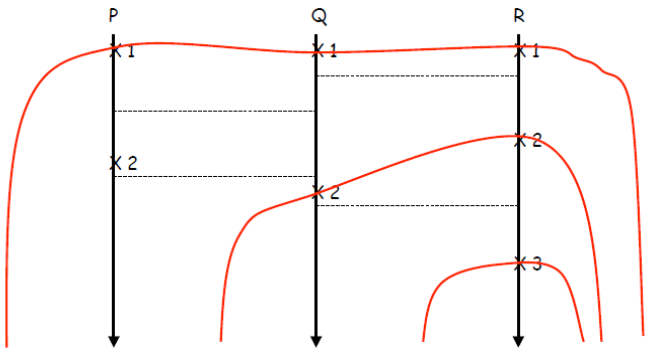
- **atomicità**: un'azione atomica è indivisibile nella sua esecuzione e la sua esecuzione deve essere o totale o nulla, non sono ammesse esecuzioni intermedie;
- **coerenza**: un'azione atomica non viola gli invarianti del Sistema (contraddizioni)
- **isolamento**: ogni azione atomica deve essere eseguita in modo isolato e indipendente dalle altre azione atomiche, l'eventuale fallimento di una azione atomica non deve interferire con le altre azione atomiche in esecuzione;
- **durabilità**: dopo che una azione atomica è stata eseguita nella sua interezza i suoi effetti vengono registrati in maniera permanente.

Vediamo un esempio di Checkpointing in ambito distribuito



L'inconveniente, come mostrato in figura è l'Effetto-domino. L'effetto domino provoca il ripristino dello stato di ciascun nodo al proprio valore iniziale, perdendo quindi tutti i risultati della computazione prodotti fino alla rivelazione dell'errore.

Introduciamo il concetto di linee di recovery. Le line di recovery, come si può osservare dalla figura sottostante, attraversano le linee di vita dei vari nodi solo in corrispondenza dei checkpoint. Le linee di recovery indicano, dato un crash in un punto di una regione compreso tra due linee, a quale checkpoint deve essere ricondotto ciascun nodo.



L'azioni atomiche distribuite = lavoro svolto dai nodi tra due linee di recovery

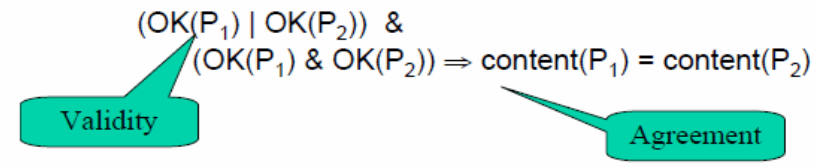
4.2 Memoria Stabile

Lo stato di ciascun nodo viene memorizzato su memoria stabile.
La memoria stabile è l'astrazione della memoria reale, con la proprietà di contenere le informazioni necessarie al recupero e di non essere soggetta ad alcun tipo di malfunzionamento.

```
type dati = .....;
      pagina = 0 .. Maxpag;
      risultato = (OK, BAD);
Procedure Read (P: pagina; var D: dati; var R:Risultato);
Procedure Write (P: pagina; D: dati);
```

Questa memoria paginata non viene influenzata da *crashes* del processore, eccetto la pagina su cui, eventualmente, si scriveva al momento del crash (procedura di *write* non atomica)

Ogni pagina stabile P_S è realizzata tramite due (o più) pagine reali P_1 e P_2 tali che non decadano mai contemporaneamente (appartengono a cilindri diversi dello stesso disco, o a dischi diversi)
Le procedure *read*-stabile, *write*-stabile, *restart* da-crash, garantiscono, per ogni pagina stabile P_S, che valga il seguente invariante:



- ***Validity:*** i dati replicati sono validi.
- ***Agreement:*** i dati replicati sono in accordo.

L'invariante vale sempre, tranne che durante l'esecuzione di una write-stabile.
Vediamo nel dettaglio le procedure di read-stabile, write-stabile, restart da-crash.

Read-stabile: legge una pagina, se la lettura non va a buon fine legge la pagina ridondante:

```
Procedure Read-stabile (Ps: pagina-stabile; var Ds: dati);
 begin
      Read(P1,d,Ris)*;
      if Ris = OK then Ds:=d;
      else begin
             Read(P2,d,Ris)*; Ds :=d;
      end;
 end;
```

2. Write-stabile: scrive una pagina e la legge per accertarsi che la scrittura sia avvenuta correttamente, procede nello stesso modo con la pagina ridondante. Il problema si ha se il *crash* avviene tra la fine della prima lettura e l'inizio della seconda scrittura:

```
Procedure Write-stabile (Ps: pagina-stabile; var Ds: dati);
 Begin
      repeat
        Write(P1, Ds); Read(P1,d,Ris);
      until Ris = OK and Ds = d;
      repeat
        Write(P2, Ds); Read(P2,d,Ris);
      until Ris = OK and Ds = d;
 end;
```

3. Restart da-crash: legge la prima e la seconda pagina, se la lettura ha successo passo alla seconda pagina, altrimenti copia la seconda pagina sulla prima e controllo l'esito della scrittura. Passo alla seconda pagina, se la lettura ha successo ed è consistente con la prima il recovery è terminato, altrimenti copia la pagina nella seconda e verifico il successo della scrittura.

```
Procedure restart-da-crash;
 Begin
      Read(P1,d1,Ris1);
      Read(P2,d2,Ris2);
      If Ris1 = BAD then
        repeat
          Write(P1, d2); Read(P1,d1,Ris1);
        until Ris1 = OK and d1 = d2;
      else if Ris2 = BAD or d1 != d2 then
       repeat
         Write(P2, d1); Read(P2,d2,Ris2);
   until Ris2 = OK and d1 = d2;
 end;
```

4.3 Two-Phase Commit Protocol

Il protocollo *two-phase commit protocol* (2PC) è un algoritmo distribuito che consente la gestione delle azioni atomiche in ambiente distribuito.
Con il protocollo di 2PC il *commit* dei dati avviene in due fasi. Nel prima fase un "coordinatore" della azione atomiche manda il messaggio di *PREPARE TO COMMIT* a tutti i partecipanti interessati all'azione atomica, i quali risponderanno positivamente inviando un messaggio di *AGREE* entro il timeout. Nella seconda fase il coordinatore invia il messaggio di *COMMIT* e tutti risponderanno con un *ACKNOWLEDGE*. Se qualcuno risponde negativamente inviando un messaggio di *DISAGREE* o non risponde, viene inviato il messaggio di *ABORT* da parte del coordinatore, e tutti risponderanno con un messaggio di ACKNOWLEDGE tornando alla situazione definita dagli ultimi checkpoint.

Nel 2PC il numero dei messaggi è 4 N, dove N è il numero dei partecipanti.
Questo protocollo è appropriato se:

- Esiste il meccanismo di broadcast (in questo caso il totale dei messaggi è 2N+2).
- Si vuol privilegiare il parallelismo tra i partecipanti.
- La struttura e il numero dei partecipanti variano dinamicamente.

4.4 Linear Two-Phase Commit Protocol

- Ad ogni partecipante è assegnato un numero progressivo.
- Ogni partecipante conosce il nome del successivo e l'ultimo sa di essere l'ultimo.
- I partecipanti comunicano, dal primo verso l'ultimo, di essere arrivati al punto di commit (fase 1).
- Se tutto va bene l'ultimo risponde OK e il messaggio fluisce in senso inverso (fase 2).
- Se durante la fase 1 qualcuno decide di abortire il messaggio "Abort" fluisce nei due sensi.
- Numero totale dei messaggi 2 N.

Questo protocollo è appropriato se:

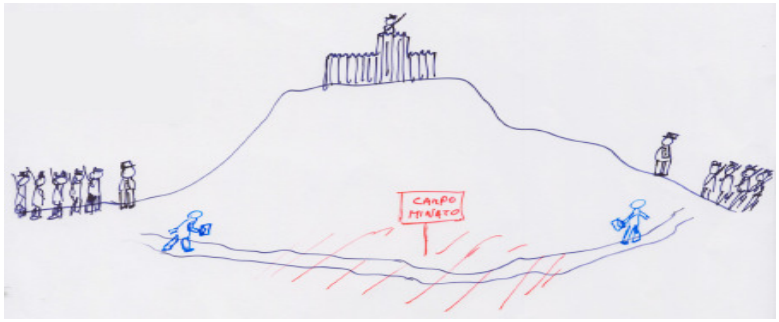
- Il meccanismo di comunicazione è costoso e in assenza di broadcast, oppure la tipologia della rete è ad anello
- Non è necessario prevedere concorrenza durante le fasi 1 e 2.
- La struttura dei partecipanti è conosciuta staticamente.

4.5 Considerazioni sugli Algoritmi distribuiti

Gli algoritmi distribuiti vengono utilizzati quando si voglia garantire la consistenza di dati in tutto o parzialmente replicati in vari server.
In generale, tali algoritmi garantiscono proprietà di Validity e Agreement a fronte di possibili guasti. Molti di questi algoritmi sono limitati dal **principio di incertezza**: un nodo che ha richiesto un'azione remota non può sempre determinare, in un tempo finito, se l'azione è stata eseguita o no.

4.6 Paradosso dei generali bizantini

Si immaginano diverse divisioni dell'esercito bizantino accampate al di fuori di una città nemica, ognuna delle quali con il proprio generale. I generali possono comunicare tra loro soltanto attraverso dei messaggeri e dopo aver osservato il nemico devono decidere un piano di attacco comune (protocollo per sincronizzare gli attacchi). Tuttavia, alcuni messaggi non arriveranno perché i messaggeri possono perdere la vita attraversando un campo minato (questo equivale ad avere un canale trasmissivo con disturbo).



E' facile da capire che non esiste un protocollo di durata massima fissata.
Dimostrazione: Supponiamo che esistano protocolli di questo tipo, e sia P quello più corto (in termini di numero di messaggi scambiati). Se l'ultimo messaggero salta su una mina, allora:

- era inutile, ma allora P non era il più corto;
- o era utile, ma allora P non funziona.

4.7 Consenso Distribuito tra processi asincroni

4.7.1 Introduzione

Definizione: un processo distribuito si dice corretto in un dato contesto se non va in crash in quel contesto (si considera il fatto che un processo può andare in crash ma non può inviare dati errati).

Un algoritmo di consenso distribuito deve soddisfare le seguenti proprietà:

1. **Validity:** se il sistema restituisce un valore v in uscita allora esiste almeno un processo che lo ha proposto.
2. **Agreement:** se due processi sono non *faulty* allora propongono valori concordanti.
3. **Termination:** un processo non faulty propone un valore entro un tempo massimo

Fischer, Lync e Paterson hanno affermato che è impossibile risolvere il consenso (cioè soddisfare le tre proprietà) se qualche processo fallisce.
Più precisamente, hanno provato che, sotto l'assunzione di Validità e Agreement, la possibilità che anche un singolo processo si gusta invalida la terminazione, cioè l'intero sistema fallisce. Così, mentre la proprietà di Validità e Agreement possono essere garantite attraverso appropriato algoritmo, al fine di rispettare la terminazione è necessario introdurre qualche forma di sincronia nel sistema.

Chandra e Toueg hanno proposto una soluzione a questo problema, aggiungendo dei "rilevatori di fallimento inaffidabili" (FDS), vale a dire, moduli che possono essere interrogati a livello locale per trovare se un altro processo a livello locale è attualmente sospettato di essere faulty (di essere andato in crash).

I rilevatori di fallimento $\Diamond\mathscr{S}$ garantiscono le seguenti due proprietà:

1. ***Eventual Weak Accuracy***: esiste un istante di tempo dopo il quale qualche processo corretto non è mai sospettato.
2. ***Strong Completeness***: prima o poi ogni processo faulty diviene permanentemente sospetto da ogni processo corretto.

4.7.2 Chandra Toueg

Chandra e Toueg [CT96] hanno proposto un algoritmo per risolvere il consenso in un modello **asincrono** con *crash-failures*, dove:

- i processi sono completamente interconnessi attraverso un canale di comunicazione bidirezionale *point-to-point* quasi-affidabile, in cui la consegna del messaggio è garantita solo se né il mittente né il ricevitore falliscono;
- il *broadcasting* è affidabile (Reliable broadcast), cioè i messaggi inviati in broadcasting prima o poi arrivano a tutti i processi non faulty.

Assunzioni di guasti:

1. solo una minoranza di processi può fallire.
2. la presenza dei FDS fornisce informazioni sui processi falliti.

Intuizione:

- L'algoritmo viene eseguito localmente da ogni processo e procede in cicli (rounds).
- Infatti si basa sul *rotating coordinator paradigm*: in ogni giro "r" un singolo processo sarà il coordinatore e gli altri saranno i partecipanti.
- Ognuno degli n processi conosce il valore "n" e conserva un local round counter; così, in ogni momento, può determinare localmente il coordinatore a quel giro .
- Per ogni giro, ciascun partecipante invia il suo valore attuale al coordinatore di questo giro.
- Il coordinatore sceglie uno tra i più recenti valori proposti che ha ricevuto e lo invia a tutti i partecipanti usando il meccanismo di Reliable Broadcast.
- Ciascun processo confronta il valore ricevuto con quello da lui calcolato e se concorda invia al coordinatore un ACK
- Se la maggioranza dei nodi ha inviato un ACK il coordinatore manda a tutti i processi corretti il risultato parziale della computazione da accettarsi come definitivo, a questo punto i processi possono decidere e uscire dall'algoritmo.

Poiché nessun presupposto sul tempo può essere fatto, è impossibile per un partecipante capire il motivo (rete lenta o si è schiantato coordinatore), che sta dietro alla non ricezione di un messaggio del coordinatore.

- In tal modo, ogni partecipante ha la possibilità di sospettare che il coordinatore abbia fallito.
- In questo caso, il partecipante invia un ACK negativo (NACK) e si sposta al seguente round, mantenendo il suo attuale valore.

4.7.2.1 Quasi-reliable point-to-point communication

I processi sono completamente interconnessi da canali di comunicazione bidirezionali che rappresentano il mezzo di comunicazione di base. Le primitive in pseudo codice sono abbastanza semplici:

- **Qpp send (sender, number, contents) to ricevitore :** denota l'invio quasi-affidabile di un messaggio da parte del processo mittente al processo ricevitore, dove number svolge il ruolo di numero di sequenza che permette di distinguere i messaggi provenienti dal mittente;
- **Qpp receive(sender, number, contents):** è una procedura di ricezione, estrae i dati pertinenti indicati formalmente dai parametri (sender, number, contents.)

La proprietà necessaria su queste operazioni è la seguente:
Quasi affidabilità: Se il processo mittente esegue "Qpp send(sender, number, contents) to ricevitore", e se sia il mittente e il ricevitore sono corretti, il ricevitore alla fine eseguirà "Qpp receive(s, n, c)" per ricevere i valori (sender, number, contents).

Formalmente il *Reliable Broadcast* è definito da due primitive:

- RBC broadcast(sender, contents): quando il processo invia in broadcast un messaggio del tipo (sender, contents)
- RBC deliver(sender, contents): quando il processo consegna un messaggio (sender, contents).

Trasmissione affidabile in broadcast (Reliable broadcast) è caratterizzata da tre seguenti proprietà:

- **Validità**: Se un processo corretto esegue RBC broadcast(sender, contenuto), alla fine esegue RBC deliver(s, c) per il messaggio (sender, contents).
- **Agreement** Se un processo corretto esegue RBC deliver(s, c) per un messaggio (sender, contents), alla fine tutti i processi corretti eseguono RBC deliver (s, c) per il messaggio (sender, contents).
- **Integrità uniforme** Per ogni messaggio (sender, contents), ogni processo esegue RBC deliver(s, c), al massimo una sola volta, e solo se alcuni processi precedentemente hanno eseguito un RBC broadcast(sender, contents).

4.7.2.2 Primitive del Consenso

Formalmente, nel problema del Consenso, i processi corretti propongono un valore e tutti i processi corretti devono decidere tra i valori proposti un valore comune. Il consenso è definito in termini di due primitive C_PROPOSE(v) e C_DECIDE(v). Quando un processo esegue C_PROPOSE(v), significa che il processo propone il valore v. Similarmente quando un processo esegue C_DECIDE(v) significa che il processo decide il valore v.
In sintesi, ogni esecuzione del consenso deve soddisfare le seguenti tre proprietà:

- **Validity:** Se un processo corretto decide un valore v (esegue C_DECIDE(V)), allora qualche processo deve avere proposto questo valore (eseguendo C_PROPOSE(V)).
- **Agreement:** Non è possibile che due processi corretti (cioè che eseguono C_DECIDE) decidono valori differenti.
- **Termination:** Ogni processo corretto alla fine decide un valore.

4.7.2.3 Algoritmo di Chandra e Toueg che risolve il consenso

In questa sezione viene proposto l'algoritmo di Chandra e Toueg che risolve il problema del Consenso usando un failure detector di tipo $\diamond\mathscr{S}$.

Questo algoritmo tollera al più di $\lceil (n/2)-1 \rceil$ processi guasti (in un sistema asincrono con n processi) e assume che $\lceil (n+1)/2 \rceil$ processi siano corretti. L'Eventual Weak Accuracy permette al failure detector di sospettare erroneamente qualsiasi processo corretto, solo dopo un certo istante di tempo qualche processo corretto non sarà più sospettato.
Questo algoritmo usa il paradigma rotating coordinator e procede in "round" asincroni.
Sia n il numero di processi, allora la funzione crd(r) restituisce il numero intero ((r - 1) mod n) che indica il coordinatore c durante il round r. Questa funzione è nota a tutti i processi, pertanto, ciascuno di essi è in grado di calcolare in qualsiasi momento l'identità del coordinatore di un ciclo. Ciascuno degli n processi memorizza una stima locale del valore decisionale cercato, insieme con uno *stamp* che dice in quale round il processo è giunto a credere in questa stima. Facciamo riferimento alla coppia (stima, stamp) con *belief*.

L'obiettivo di questo algoritmo è quello di raggiungere una situazione in cui la maggioranza dei processi è d'accordo sulla stessa stima. Per raggiungere una tale situazione, i coordinatori e partecipanti si scambiano le loro attuali stime, round dopo round, attraverso Qpp-messaggi.

Il ruolo di un coordinatore è: (i)raccogliere un numero sufficientemente grande di beliefs, (ii) per scoprire quello con lo stamp più alto (iii) e selezionarlo, e (iv) alla fine diffondere a tutti i partecipanti tramite un Reliable broadcast la sua selezione in modo che essi possano aggiornare le loro beliefs locali.

Analizziamo la struttura del loop dell'algoritmo in
Figura **13**.
Come detto prima, ogni processo passa attraverso quattro fasi per ciclo, vale a dire il ciclo di *while*:

- **Fase 1:** durante questa fase ogni processo invia la stima corrente del valore deciso, e lo stamp pari al valore dell'ultimo round in cui il valore stimato è stato aggiornato, al processo coordinatore c.

- **Fase 2:** tutti i processi, tranne il coordinatore passano immediatamente alla fase P3. Il processo coordinatore c durante questa fase raccoglie $\lceil (n+1)/2 \rceil$ valori, e di tali valori stimati seleziona quello con lo stamp più alto, questa operazione è eseguita attraverso la funzione best (Qpp:$P1^r$ ricevuto) , e lo invia a tutti i processi usando di nuovo Qpp-messanger. Il coordinatore passa alla fase P3.

- **Fase 3:** durante questa fase per ogni processo corretto ci sono due possibilità:
  - un generico processo p riceve la stima dal processo coordinatore c e invia un ack a c per indicare che ha adottato la proposta dal coordinatore come nuova stima; oppure
  - in seguito all'aver consultato il modulo locale di failure detector FD_p, p sospetta che c abbia fatto crash, e invia un nack a c.

- **Fase 4:** In questa fase, tutti i processi (tranne il coordinatore) procedono al turno (round) successivo attraverso while-loop. Il coordinatore aspetta fino a quando non riceve molte risposte di acknowledgment. La funzione ack(received Qpp: $P3^r$) seleziona solo messaggi di ack dal sottoinsieme di tutti i messaggi ricevuti ack e nack (received_Qpp.$P3^r$), Se la cardinalità di ack è almeno pari a $\lceil (n+1)/2 \rceil$ diciamo che il valore proposto da questa maggioranza è *locked* e il coordinatore “avvia una decisione", altrimenti il coordinatore fallisce questa occasione e finisce semplicemente questo round. In entrambi i casi, il coordinatore procede quindi al prossimo round attraverso il rientro nel *while-loop.*

```
PROCEDURE C_PROPOSE (v_i)
  state.counter ← 0
  state.belief.value ← v_i
  state.belief.stamp ← 0
  state.decision ← ⊥

  while state.decision = ⊥
  {  state.counter ← state.counter + 1
     r ← state.counter

     P1   Q_PP_SEND (i, (r, P1), state.belief) TO crd(r)

     P2   if      i = crd(r)
          then    { await  | received_Q_PP.P1^r | ≥ ⌈(n+1)/2⌉
                    for 1 ≤ j ≤ n
                    do  Q_PP_SEND (i, (r, P2), best(received_Q_PP.P1^r)) TO j
                  }

     P3   await  ( received_Q_PP.P2^r ≠ ∅  or  suspected(crd(r)) )
          if      received_Q_PP.P2^r = {(crd(r), (r, P2), (v, s))}
          then    { Q_PP_SEND (i, (r, P3), ack) TO crd(r)
                    state.belief ← (v, r)
                  }
          else      Q_PP_SEND (i, (r, P3), nack) TO crd(r)

     P4   if      i = crd(r)
          then    { await  | received_Q_PP.P3^r | ≥ ⌈(n+1)/2⌉
                    if     | ack(received_Q_PP.P3^r) | ≥ ⌈(n+1)/2⌉
                    then   RBC_BROADCAST(i, state.belief)
                  }
  }

when  RBC_DELIVER(j, d)
{  if     state.decision = ⊥
   then   state.decision ← d
          C_DECIDE (state.decision.value)
}
```

Figura 13 : Algoritmo che risolve il consenso con $\diamond\mathscr{S}$

L'algoritmo funziona correttamente se sono verificate correttamente le tre proprietà: Validity, Agreement, e Termination:

- **Validity** vale perché nessuna delle clausole dell'algoritmo inventa un valore. Tutti i valori che si verificano in un sistema sono stati inizialmente proposti da qualche processo.

- **Agreement** e **Terminazione** possono essere spiegati in modo intuitivo mediante l'esecuzione di tre epoche asincrone, ognuna delle quali può generare diversi round asincroni. In epoca 1 tutto è possibile, vale a dire, ogni valore iniziale potrebbe essere deciso. Nella seconda epoca un valore diventa "locked": nessun altro valore può essere deciso. Nella terza epoca i processi decidono il valore "locked".

4.8 Votazione Distribuita

Gli elementi che differiscono la votazione distribuita dal consenso distribuito sono i seguenti:

- Sincronismo.
- Modello dei guasti prevede la possibilità di generazione di risultati errati.

4.8.1 Byzantine Agreement

- Con questo nome si indica una famiglia di algoritmi il cui scopo è quello di raggiungere un consenso anche in presenza di guasti arbitrari (bizantini), ma introducendo assunzioni di sincronizzazione tra i nodi.
- L'algoritmo prevede l'esistenza di un unico trasmettitore e tutti i nodi ricevono lo stesso valore da lui inviato (è un modo per realizzare un broadcast reliable).
- L'algoritmo viene definito in modo che n ricevitori possano raggiungere un accordo sul valore che è stato trasmesso loro da un trasmettitore.
- Per compiere una votazione distribuita sugli n valori proposti da n nodi si usa l'algoritmo di **consistenza interattiva** (*interactive consistency*) che è composto da n algoritmi interlacciati di byzantine agreement in cui ognuno degli n nodi si comporta da trasmettitore rispetto agli altri n-1 nodi.

Definiamo l'algoritmo ZA, algoritmo che si basa sull'autenticazione, deriva dall'algoritmo Z [*Thambidurai and Park* 1988], ma prima di esporre il funzionamento introduciamo:

Modello dei guasti Ibridi
Il modello dei guasti che usiamo viene detto modello ibrido, le tre classi di guasti sono:

- **Guasti manifesti**, ovvero è possibile che un processo invii un valore da lui calcolato palesemente errato tanto che ogni altro nodo è in grado di rilevarlo autonomamente.
- **Guasti simmetrici**, ovvero guasti non rilevabili localmente dai nodi ai quali è inviato il medesimo valore errato.
- **Guasti arbitrari (bizantini),** ovvero è possibile che un processo trasmettitore invii valori diversi a processi diversi e che i processi non siano in grado di rilevare il guasto localmente.

Sotto questo modello dei guasti, quando un nodo mittente manda un valore v al nodo ricevitore non guasto, il valore ottenuto è:

- il valore v, nel caso in cui il mittente è non guasto;
- error, se il mittente ha un guasto manifesto.

Assunzioni:
L'algoritmo ZA richiede le seguenti assunzioni:

A1. Ogni messaggio inviato tra due processi non guasti è recapitato correttamente.

A2. Il ricevitore conosce il mittente di ogni messaggio.

A3. I nodi sono sincronizzati, quindi la mancata ricezione di un messaggio è rilevata dal ricevente.

A4. I messaggi ritrasmessi non possono essere corrotti.

Proprietà:
Le proprietà che l'algoritmo deve verificare sono:

- **Validity:** se un ricevitore p non è guasto, allora il valore che p associa al trasmettitore è:
  - il valore effettivamente inviato nel caso in cui il trasmettitore sia corretto o abbia un guasto simmetrico.
  - Il valore "error" se il trasmettitore ha un guasto manifesto.
  - Se il guasto è bizantino non si può fare alcuna assunzione.
- **Agreement:** dati due ricevitori p e q non fault i valori da loro associati al trasmettitore sono uguali (hanno la stessa percezione di ciò che il trasmettitore ha inviato).
- **Termination:** l'algoritmo termina in un tempo massimo determinato.

L'algoritmo ZA permette la consistenza interattiva (I.C): se ho un processo guasto gli altri sono in grado di trovare comunque una stima corretta del valore grazie al meccanismo di votazione interna.

L'algoritmo ZA soddisfa le proprietà sotto le condizioni:

1. $n > a + s + m + 1$
2. $r \geq a$

dove:

- **n** numero di nodi;
- **a** numero di nodi arbitrariamente guasti;
- **s** numero di nodi simmetricamente guasti;
- **m** numero di nodi con guasti manifesti;
- **r** numero di rounds -1.

L'algoritmo termina sempre a causa dell'ipotesi di sincronia.

Vediamo nel dettaglio il funzionamento dell'**Algoritmo ZA:**
ZA(0)

1. Il trasmettitore manda il valore v ad ogni ricevitore.
2. Ogni ricevitore usa il valore ricevuto dal trasmettitore, o il valore "errore" in caso di guasti manifesti.

ZA(r) r >0

1. Il trasmettitore firma v e lo manda ad ogni ricevitore.
2. Sia v_p il valore ricevuto dal trasmettitore, o il valore "errore" in caso di guasti manifesti, ogni ricevitore p fa partire un algoritmo ZA(r-1) dove p diventa il trasmettitore e comunica il valore v_p verso tutti gli altri n-2 ricevitori.
3. Per ogni ricevitore p e q, sia v_p il valore che p riceve da q nel passo 2 di ZA(r-1) o il valore "errore" in caso di guasti manifesti; ogni ricevitore p esegue localmente la votazione a maggioranza sui v_q ricevuti (al più n-2).

Il numero di round nell'algoritmo ZA(r) è uguale a r+1.

Innanzitutto c'è un trasmettitore che trasmette il suo valore; se l'algoritmo si basa su un solo round (r=0), gli altri nodi ricevono e si accontentano.
Se l'algoritmo si basa su più round (r+1), gli altri nodi non si accontentano di ricevere dal trasmettitore, ma si proclamano trasmettitori facendo partire un algoritmo di ZA con r round (che è ZA(r-1)) indirizzato agli altri n-2 nodi (cioè non a se stesso e non al trasmettitore precedente).

Ad esempio, se r=1, quindi ci sono due round, il trasmettitore manda il suo valore. Ognuno degli altri nodi trasmette il valore a tutti gli altri n-2. I vari nodi, essendo uno ZA(0) ricevono questo(i) secondo(i) messaggio(i) e non ritrasmettono. Avendo però ricevuto n-1 messaggi (da tutti i nodi che non siano il trasmettitore) devono votarli per maggioranza, con le modalità descritte nel punto 3.
Quindi, il punto 2 serve a far partire un numero di round pari a r+1, il punto 3 (e qui sta forse la fonte maggiore delle possibili incomprensioni) serve a votare tutti i messaggi ricevuti dall'ultimo round effettuato: notare che questo passo non è ricorsivo, cioè la votazione viene fatta una volta sola, alla fine di tutti i round. Quindi, il messaggio originale ad ogni round viene ritrasmesso, e solo quando giunge alla destinazione dell'ultimo round non viene più ritrasmesso e viene votato.

Essenzialmente si vuole che il messaggio passi da più nodi possibili. L'autenticazione serve ad individuare il nodo guasto che ha spedito un messaggio corrotto.

A causa della velocità attuale dei clock non è possibile assumere la sincronia. Allo scopo sono nati diversi algoritmi distribuiti di sincronizzazione dei clock.

4.9 Algoritmi di Sincronizzazione in ambito distribuito

4.9.1 Formalismi e definizioni

Ogni nodo ha un proprio clock, cioè una funzione C(t) con un tempo reale t. C(t) ha un tasso massimo di deriva ρ. Siano t_1 e t_2 due istanti di tempo tali per cui:

$$t1 < t2 \qquad 1 - \rho < \frac{C(t_2) - C(t_1)}{t_2 - t_1} < 1 + \rho$$

Proprietà 1 (*Agreement* tra clock): Siano C1(t) e C2(t) due clock fisici non guasti e sia δ un valore reale fissato detto scarto. Si dice che i due clock sono considerati in accordo rispetto allo scarto δ se vale:

$$\forall t : \left| \; C_1(t) - C_2(t) \; \right| \leq \delta$$

Proprietà 2 (*Accurancy* di un clock): Siano γ, a, b variabili fissate dipendenti dalle condizioni iniziali di avvio del clock, si dice che un clock fisico C(t) è accurato se vale:

$$(1-\gamma)\cdot t + a < C(t) < (1-\gamma)\cdot t + b$$

Questo significa che date due rette $(1-\gamma)\cdot t + a$ e $(1-\gamma)\cdot t + b$ il clock C(t) si mantiene tra le due rette.

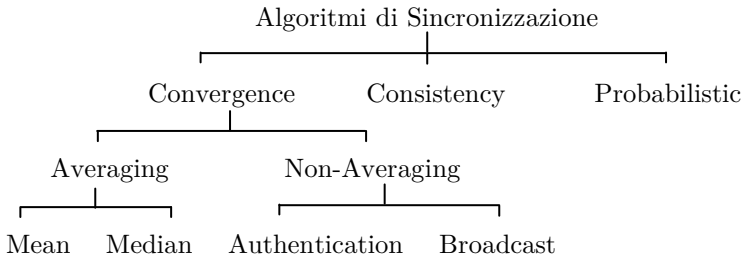
Definizione: un algoritmo di *clock synchronization* è un algoritmo che soddisfa le proprietà 1 e 2.

Un algoritmo di clock synchronization effettua una risincronizzazione ad intervalli regolari lunghi r. Sia ρ il tasso di deriva, si osserva facilmente che il massimo scostamento tollerato dalla condizione ideale è pari a (ρ . r) che definisce una limitazione superiore per tale scostamento.

4.9.2 Classificazione degli Algoritmi di Sincronizzazione

Ad oggi esistono svariati algoritmi che svolgono il problema della sincronizzazione in ambito distribuito. Ogni algoritmo differisce principalmente nel tipo di assunzioni che si fanno per il suo corretto funzionamento nel tipo di connessione che è richiesta fra i nodi, nella modalità di scambio dei messaggi e nelle prestazioni: traslazione temporale massima tra due clock non guasti, massimo clock guasti tollerati. Principalmente si dividono in 3 categorie:

- Algoritmi probabilistici.
- Algoritmi di consistenza.
- Algoritmi a convergenza.



Algoritmi probabilistici

Questo tipo di algoritmo è in grado di minimizzare la traslazione temporale massima tra due clock ad un valore piccolo a piacimento ma con una probabilità di perdita di sincronizzazione crescente.

Il problema di questi algoritmi è, sostanzialmente, che non garantiscono con probabilità unitaria il mantenimento della sincronizzazione, non vengono quindi considerati affidabili per sistemi *safety-critical*.

Algoritmi di consistenza

Questo tipo di algoritmo considera il valore del clock come un dato qualunque e vi applica un algoritmo a più round di consistenza interattiva (*interactive consistency*). Si tratta il problema della sincronizzazione come il problema dei generali Bizantini. Si vuole creare una condizione di accordo tra i nodi ovvero si vuole che se due nodi sono non guasti inviano allo stesso trasmettitore il medesimo valore. Inoltre se il trasmettente è non guasto il valore su cui si accordano i nodi non guasti è il valore privato del trasmittente.

Algoritmi a convergenza

Gli algoritmi a convergenza si dividono in:

1. Algoritmi a media (*Averaging*)
2. Algoritmi non a media (*Non-Averaging*)

1. Algoritmi a media (Averaging)

In questo tipo di algoritmi i processi intraprendono una procedura di risincronizzazione ogni R secondi. Contemporaneamente in una finestra di tempo centrata (o meno) sull'istante di risincronizzazione attendono i messaggi degli altri processi. Giunti alla fine della finestra di attesa calcolano il valore temporale logico a cui convergere con una funzione fault-toleran.

Vi sono due diversi algoritmi, tollerano al più m clock guasti su 3m+1 nodi, che calcolano tale funzioni in maniera diversa:

- **Mediana:** si eliminano m valori estremi più alti e più bassi e si calcola la media dei restanti.
- **Media:** si effettua la media tra i valori ricevuti a patto che non differiscano più di una costante δ dal valore locale.

Tali valori vengono sostituiti dal valore locale. In entrambi gli algoritmi viene data una stima nel caso pessimo per la massima traslazione temporale dei due clock.

Questi due algoritmi si basano sul fatto che ogni clock è in grado di conoscere non il valore locale degli altri clock ma la differenza tra essi, misurata appunto come ritardo (anticipo) tra il proprio istante di risincronizzazione e l'arrivo dei messaggi degli altri orologi. Entrambi gli algoritmi fanno l'assunzione della ρ-limitatezza dei clock coinvolti. Si considera tipicamente una rete di tipo broadcast. E' stato anche dimostrato che in assenza di un meccanismo di firma digitale dei messaggi scambiati 3m+1 nodi il minimo possibile per poter risolvere il problema della sincronizzazione in presenza di m clock bizantini.

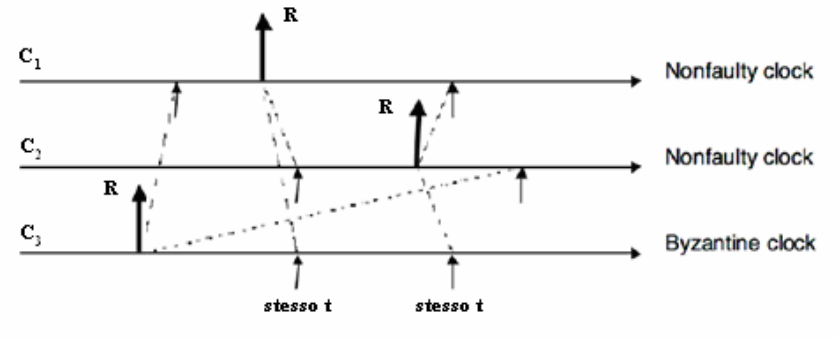


Figura 14: Desincronizzazione di due clock non guasti e la presenza di un clock con guasto bizantino.

2. Algoritmi non a media (*Non-Averaging*)

Come è stato detto precedentemente R è il periodo di risincronizzazione, ogni R secondi viene avviato il nuovo clock locale. In particolare, sia $C_i^{k-1}(t)$ il clock durante il k-1 esimo ciclo di risincronizzazione. Alla fine di tale ciclo si avrà che $C_i^{k-1}(t) = k \cdot R$. In questo istante il nodo invia in broadcast il messaggio firmato "round k" quanti sono i nodi. Tutti i nodi fanno la stessa cosa, quindi il nostro nodo appena ne ha raccolti m+1, incluso il proprio, imposta il suo valore locale $C_i^k(t) = k \cdot R + a$ dove:

- R è il periodo di risincronizzazione;
- k il round dell'algoritmo;
- *a* una costante tale che $C_i^k(t) > C_i^{k-1}(t)$.

Successivamente effettua un relay degli m+1 messaggi ricevuti "round k" agli altri nodi. Grazie all'uso di firme non alterabili durante i relay, questo algoritmo tollera t nodi guasti con 2t+1 nodi. Tuttavia il ritardo massimo tra due clock è maggiore del ritardo dovuto alla trasmissione sulla rete.

Il seguente algoritmo può essere descritto attraverso il seguente **pseudocodice**:

if $C^{k-1}(t) = k \cdot R$
  then manda in broadcast il messaggio (round k)
  //
if (m+1) messaggi ricevuti "round k"
  then $C_i^k(t) = k \cdot R + a$
    relay degli (m+1) messaggi a tutti

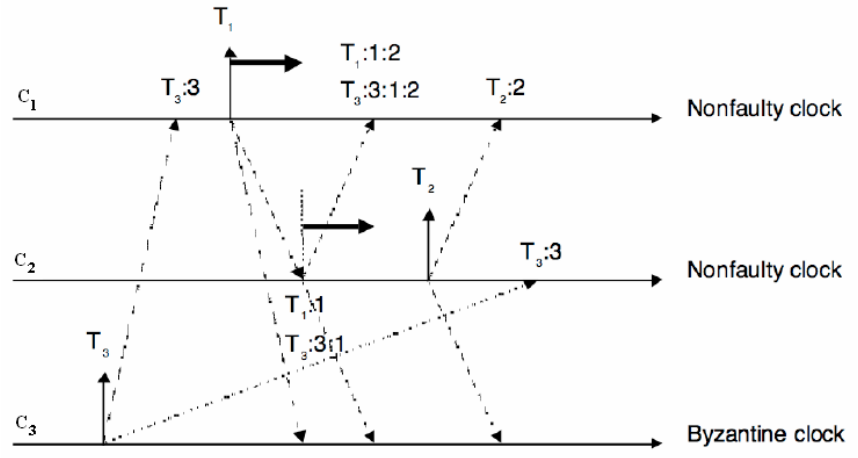


Figura 15: Tolleranza di un clock Bizantine in un sistema a 3 nodi

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 $il(A) < il(B)$):

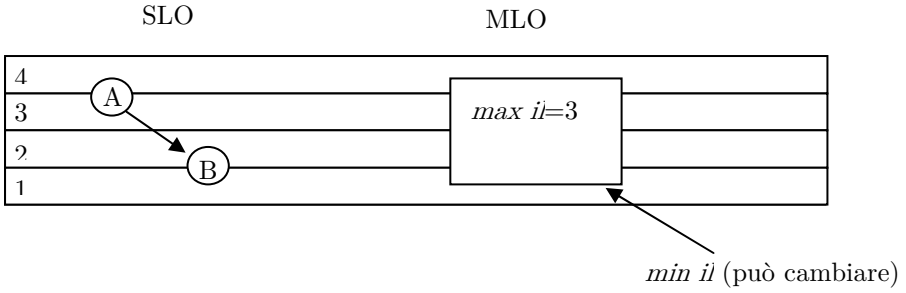
1. A blocca B vediamo come:
   - tramite meccanismi di priorità: un S.O real time da priorità ad A.
   - A impedisce a B di ricevere messaggi (DoS).
2. A invia informazioni scorrette a B o scrive scorrettamente sulle variabili di B.

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):

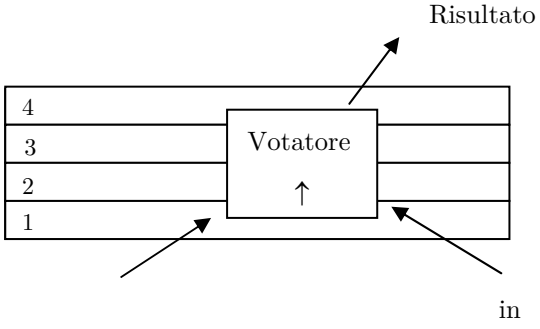
- A read B (A←B) $il(A) \leq il(B)$
- A write B (A→B) $il(A) \leq il(B)$
- A r&w B (A↔B) $il(A) = il(B)$

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:

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

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:

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

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($\exists x : \varphi$) = ….
  poiché $\exists x : \varphi$ 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:

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

Possiamo ridefinire la semantica attraverso la relazione di soddisfacibilità ($\models$):

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

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
- $\sigma \models \exists x : \varphi$ 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.

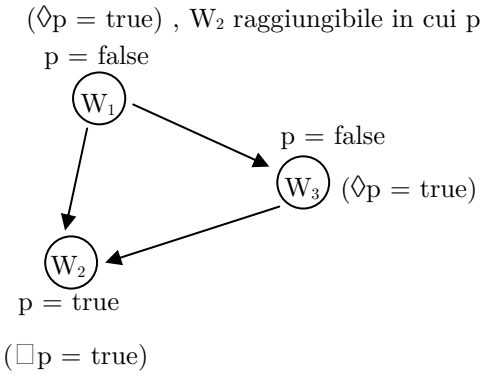
5.3 Logica Modale

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 ($\Box$p) o p è possibilmente vera ($\Diamond$p).
La possibilità o necessità si riferisce all'esistenza di "mondi", comunque raggiungibili, in cui p possa o debba essere vera:

- $\Box$p significa che in tutti i mondi raggiungibili p è vera.
- $\Diamond$p significa che $\exists$ un mondo raggiungibile in cui p è vera.

Proprietà della dualità: $\Box\neg p = \neg\Diamond p$

Esempio:



Sintassi:

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

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 $R \subseteq W \times W$.
- **V** è la funzione di valutazione delle proposizioni $V : P \times W \rightarrow \{\text{true, false}\}$

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
- $\sigma,w_0 \models \Box\varphi$ 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 $(\sigma,w_0 \models)$.

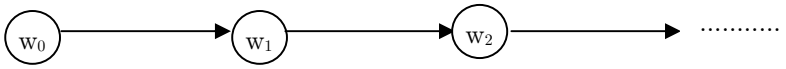
Le proprietà della relazione di raggiungibilità considerate inducono degli assiomi di equivalenza tra formule della logica:

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

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

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

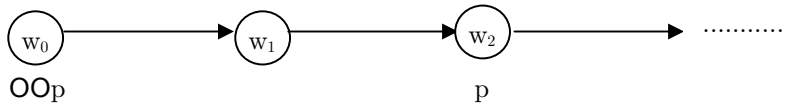


Allora vale sia $\Box\varphi \Rightarrow \Diamond\varphi$ (se φ è vera in tutti i mondi raggiungibili allora esiste un mondo in cui φ è vera) che $\Diamond\varphi \Rightarrow \Box\varphi$ (se esiste un mondo in cui φ è vera allora φ è vera in tutti i mondi).
Dunque le due modalità si equivalgono, cioè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

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 $\Box\neg p = \neg\Diamond p \Rightarrow \ O\neg p = \neg Op$

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 $\Diamond\varphi = \Box\varphi$, mentre vale :

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

Per la proprietà riflessiva e transitiva abbiamo:

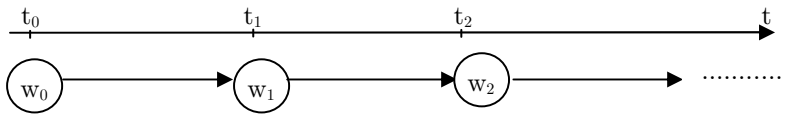
- $\Box\varphi = \Box\Box\varphi$
- $\Diamond\varphi = \Diamond\Diamond\varphi$

Dove:

- $\Box\varphi$: 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 $t_0, t_1, \ldots . t_n$ 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:

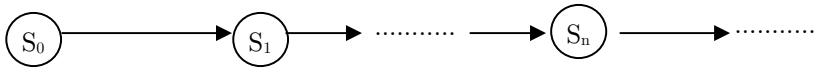
- $\Box$ diventa **ALWAYS** (sempre); quindi $\Box\varphi$ indica che φ è vero d'ora in poi: $t_0 \models_\sigma \Box\varphi$ se solo se $\forall t \geq t_0$ il tempo $t \models \varphi$
- $\Diamond$ diventa **EVENTUALLY** (prima o poi) quindi $\Diamond\varphi$ indica che φ sarà vero prima o poi: $t_0 \models_\sigma \Diamond\varphi$ sse $\exists t \geq t_0$ il tempo $t \models \varphi$
- L'operatore O **NEXT** non subisce cambiamenti, quindi $\mathsf{O}\varphi$ 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** $\Box\varphi$ (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** $\varphi_1 \mathsf{W} \varphi_2 = (\varphi_1 \mathsf{U} \varphi_2) \vee \Box\varphi_1$, 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×S → {true, false}
- P sono le preposizioni atomiche.

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

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

Nel caso più specifico, per i singoli stati S_i:

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

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

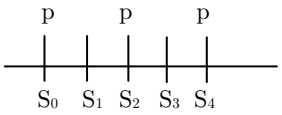
- $\Box y > n+1$, **proprietà Invariante** (è sempre vero).
- $\Diamond$p, **proprietà *Liveness*** (prima o poi succede qualcosa).
- $\Box\neg$bad, **proprietà *Safety*** (non è mai vero che qualcosa è scorretto).
- $\Box$ (request $\Rightarrow \Diamond$reply), **proprietà *Liveness*** (se faccio una richiesta prima o poi risponde).
- $\Box\Diamond$p (GFp), **proprietà *Infinitely often*** (p è vera infinitamente spesso).
- $\Diamond\Box$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:*

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

se richiedo infinitamente spesso, infinitamente spesso ottengo risposte.

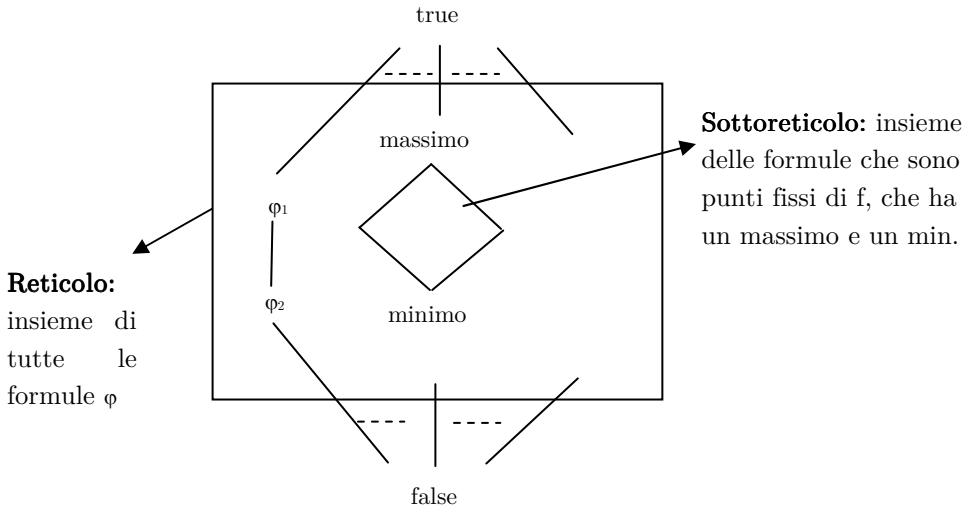
Esempio: p è vera in ogni stato pari



Formule che esprimono la condizione:

- $\Box(p \wedge \mathsf{OO}p)$ non va bene, perché vale anche per i dispari.
- $\Box(p \wedge \mathsf{O}\neg p \wedge \mathsf{OO}p)$ non va bene, è sempre falsa.
- $X = p \wedge \mathsf{OOX}$ 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.



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

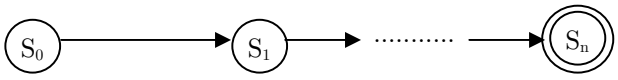
$\phi_1\ \phi_2$

$\phi_1\ \phi_1\ \phi_2$

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 ∀ stato successore vale p.
- ◊'φ ∃ stato successore per cui vale p.

Per la sequenza finita vale :

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

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

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

Inoltre:

- $\mathsf{\Theta}$false = true all'ultimo stato.
- Ofalse = false.
- $S \models \mathsf{\Theta}\text{false} \Leftrightarrow S = S_n$ (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' ⇔ t ≤ 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:

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

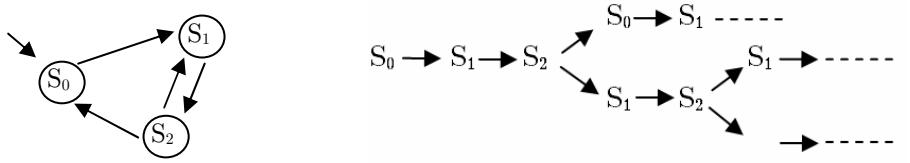
Mentre per un operazione del tipo “se viene fatta una richiesta la risposta viene data entro 3 unità di tempo”, si esprime nel seguente modo:

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

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

Semantica per le Formule di Cammino:

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

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 |= $\mathsf{X}\ g_1$ $\Leftrightarrow$ π^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.

6. Model Checking CTL

6.1 L'algoritmo di Checking

Il problema del Model Checking si può definire come il problema, data una struttura di Kripke M e una formula f (che esprime una proprietà desiderata per M), di verificare se M soddisfa f (M |= f), cioè se M è un modello per f.
Un algoritmo che risolve il problema del Model Checking per la logica **CTL** è stato definito da Clarke, Emerson e Sistla; l'algoritmo etichetta ogni stato di M con le sottoformule di f che sono soddisfatte in quello stato, iniziando con le sottoformule di lunghezza 0 (i predicati atomici), per poi passare a quelle di lunghezza 1 (in cui cioè un solo operatore logico viene applicato ai predicati atomici), poi a quelle di lunghezza 2 (in cui cioè un operatore logico viene applicato a formule di lunghezza 1), ecc..
Di seguito riportiamo la versione semplificata di questo algoritmo di etichettamento per le sole formule base della logica, cui si possono ricondurre tutte le altre. Questo algoritmo assume che tutti gli stati siano già etichettati con le proposizioni atomiche che vi valgono; la formula da verificare è p_0, mentre S è lo spazio degli stati su cui verificarla:

for i =1 **to** length(p_0)
for each subformula p of p_0 of length i
case on the form of p
p = P, an atomic proposition /* nothing to do */
p = q and r: **for each** s in S
if q in L(s) **and** r in L(s) **then** add q and r to L(s)
end
p = ~q: **for each** s in S
if q in L(s) **then** add ~q to L(s)
end
p = EXq: **for each** s in S
if(for some successor t of s, q in L(t)) **then** add EXq to L(s)
end

```
    p = A[q U r]:  for each s in S
                    if r in L(s) then add A[q U r] to L(s)
                   end
                   for j  = 1 to card(S)
                      for each s in S
                          if q in L(s) and (for each successor t of s, A[q U r]
                           in L(t)) then add A[q U r] to L(s)
                      end
                   end
    p = E[q U r]:  for each s in S
                      if r in L(S) then add E[q U r] to L(s)
                   end
                   for  j  = 1 to card(S)
                       for each s in S
                           if q in L(s) and (for some successor t of s, E[q U r]
                           in L(t)) then add E[q U r] to L(s)
                        end
                   end

  end of case
end
end
```

Il risultato del model checking si evince dalle formule che etichettano lo stato iniziale: la formula è verificata se e solo se etichetta lo stato iniziale.

L'algoritmo ha complessità $|S| \cdot |T| \cdot |\varphi|$ dove:

- $|S|$ numero di stati.
- $|T|$ transizioni uscenti da ogni stato
- $|\varphi|$ lunghezza della formula

Tramite ottimizzazione si può arrivare a $\theta(|\varphi| \cdot |S|)$

Quindi CTL ha un algoritmo di model checking lineare con lo spazio degli stati. Questo è importante poiché nella pratica lo spazio degli stati tende ad esplodere esponenzialmente. Per risolvere tale problema si è passati dalla rappresentazione esplicita dello spazio degli stati ad una rappresentazione implicita (simbolica). Questo viene fatto tramite i binary decision diagrams (BDD).

Dato che CTL* ha più potere espressivo di CTL e LTL, perché non si usa?
L'algoritmo di model checking CTL* ha complessità:

$\theta\ ((\lvert\varphi\rvert \cdot \lvert S\rvert)^k)$	dove k è il grado di annidamento dei G e degli F ($\mathsf{AGF}\varphi$ ha annidamento pari a 2).

che confrontata con quella del CTL, ha al suo interno un fattore di esponenzialità che lo rende meno efficiente.

6.2 Algoritmo del Model Checking Simbolico.

L'algoritmo di *Symbolic Model Checking* utilizza gli *Ordered Binary Decision Diagram* (OBDD) come rappresentazione simbolica di transizioni e insiemi di stati.

6.2.1 Ordered Binary Decision Diagrams (OBDD)

Per definire l'OBDD dobbiamo innanzitutto definire il *Binary Decision Tree* (BDT) e il *Binary Decision Diagram* (BDD) infatti possiamo schematizzare nel seguente modo:

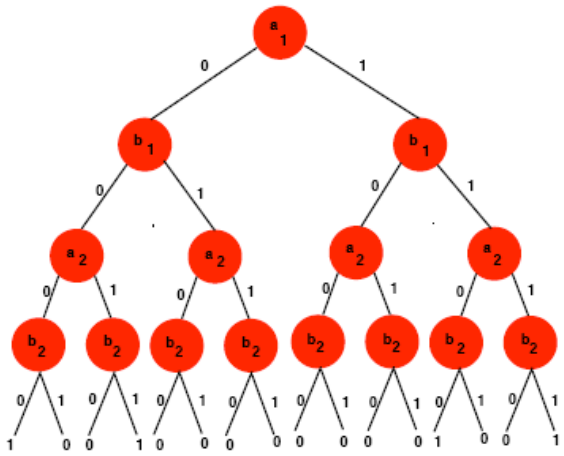
BDT → BDD → OBDD

Binary Decision Tree (BDT)
Una funzione booleana in più variabili si può rappresentare tramite un albero binario di decisione. I nodi non terminali fanno riferimento alle variabili (la stessa per tutti i nodi allo stesso livello). I livelli sono tanti quante sono le variabili. Ogni nodo non terminale ha due archi (low e hight) uscenti etichettati rispettivamente 1 e 0 (valori possibili della variabile che etichetta il nodo). Le foglie sono etichettate con 1 e 0. Il valore della funzione sui suoi input si ottiene attraverso i nodi di decisione dell'albero dalla radice fino alle foglie.

Esempio: Binary Decision Tree per un comparatore a due bit, data dalla seguente formula:

$$f(a_1, a_2, b_1, b_2)=(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$$

È mostrato nella figura sottostante:



Binary Decision Diagrams (BDD)

I BDT hanno dimensione esponenziale nel numero di variabili. Per compattare la rappresentazione dobbiamo eliminare le ridondanze:

- Combinare tutti i sottoalberi isomorfi.
- Eliminare tutti i nodi con figli isomorfi.

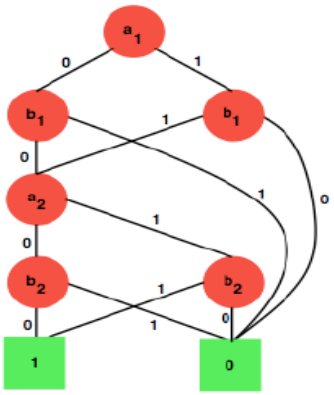
Si ottiene così un D.G.A (grafo diretto aciclico) chiamato BDD. A questo punto non rimane che trovare una rappresentazione canonica.

Ordered Binary Decision Diagrams (OBDD)

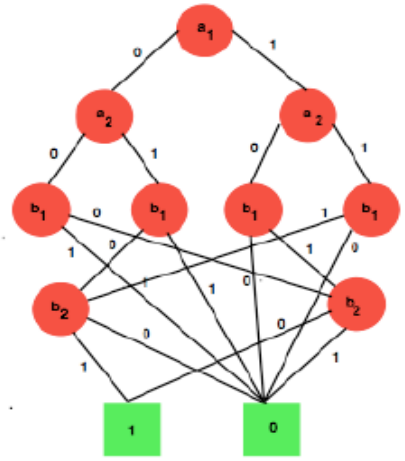
L'OBDD è una rappresentazione canonica di una funzione booleana, dove l'ordinamento delle variabili avviene nel seguente modo:

- In ogni cammino dalla radice alle foglie si mantiene lo stesso ordine, senza ripetizioni.
- Non tutte le variabili devono comparire lungo un cammino.
- Non devono esserci sottoalberi isomorfi o vertici ridondanti nel diagramma.

Se usiamo un ordine $a_1 < b_1 < a_2 < b_2$ per il comparatore, otteniamo il seguente OBDD:



Se usiamo un ordine $a_1 < a_2 < b_1 < b_2$ per il comparatore, otteniamo il seguente OBDD:



6.2.1.1 Logica operazionale OBDD:

- Negazione Logica: ¬f(a,b,c,d)
  Si sostituisce ciascuna foglia con la sua negazione.
- Congiunzione logica: f(a,b,c,d) ∧ g(a,b,c,d)
  - Si usa l'espansione di Shannon:

$$f \cdot g = \bar{a} \cdot \left(f\big|_{\bar{a}} \cdot g\big|_{\bar{a}} \right) + a \cdot \left(f\big|_{a} \cdot g\big|_{a} \right)$$

  - Divide il problema in due sottoproblemi. Questo significa costruire un albero la cui prima variabile è *a*, il ramo 0 porta all'OBDD del sottoproblema sinistro, il ramo 1 porta all'OBDD del sottoproblema destro. I sottoproblemi si risolvono ricorsivamente.

- Quantificatore Booleano: ∃a : f(a, b, c, d)
  - Per definizione

$$\exists a : f = \left(f\big|_{\bar{a}} \ \vee \ f\big|_{a} \right)$$

  - $f(a,b,c,d)\big|_{\bar{a}}$: sostituire tutti gli *a* nodi con il sotto-albero sinistro.
  - $f(a,b,c,d)\big|_{a}$: sostituire tutti gli *a* nodi nel sotto-albero destro.

Utilizzando le operazioni appena viste, si può costruire un OBDD per funzioni booleane complesse.

6.2.2 Rappresentazione delle transizioni con OBDD

Come detto prima il Symbolic Model Checking utilizza gli OBBD come rappresentazione simbolica di transizioni e insiemi di stati, ma come si rappresenta le transizioni di stato di un grafo con OBDD?
Si supponga che il comportamento del sistema è determinato dalle variabili booleane di stato: v_1,v_2, v_3,…….v_n

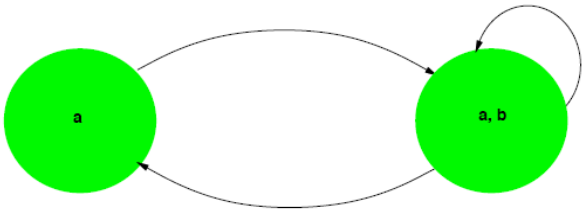
La relazione di transizione T sarà data come una formula booleana in termini di variabili di stato:

$$T(v_1,v_2,\ v_3,\ldots\ldots.v_n,\ v'_1,v'_2,\ v'_3,\ldots\ldots.v'_n)$$

dove v_1,v_2, v_3,…….v_n rappresenta lo stato attuale e v'_1,v'_2, v'_3,…….v'_n rappresenta il prossimo Stato.

A questo punto possiamo rappresentare T in OBDD.

Supponiamo di avere la seguente struttura di Kripke:



La relazione di transizione può essere rappresentata sotto forma di formule Booleane, nel seguente modo:

$$(a \wedge \neg b \ \wedge a' \wedge b') \vee (a \wedge b \ \wedge a' \wedge b') \vee (a \wedge b \ \wedge a' \wedge \neg b')$$

Poi si trasforma tale formula booleana in OBDD

Consideriamo la seguente formula f = EXp .

Introduciamo le variabili di stato e la relazione di transizione:

$$f(\overline{v}) = \exists \overline{v}' \; [T(\overline{v},\overline{v}') \wedge p(\overline{v}')]$$

A questo punto si calcola l'OBDD per la relazione prodotta sul lato destro della formula.

Ora consideriamo la formula f = EFp e valutiamo il punto fisso utilizzando OBDDs:

$$EFp = \mathrm{Lfp}\, U.p \vee EX\, U \qquad \text{dove Lfp (minimo punto fisso).}$$

Introducendo le variabili di stato e la relazione di transizione otteniamo:

$$EFp = \mathrm{Lfp}\, U.p(\overline{v}) \vee \exists \overline{v}' \wedge [T(\overline{v},\overline{v}') \wedge U(\overline{v}')]$$

A questo punto si calcola la sequenza:

$$U_0(\overline{v}),\ U_1(\overline{v}),\ U_2(\overline{v}),\ .....$$

fino alla sua convergenza.

La convergenza può essere rilevata dato che gli stati $U_i(\overline{v})$ sono rappresentati come OBDDs.

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

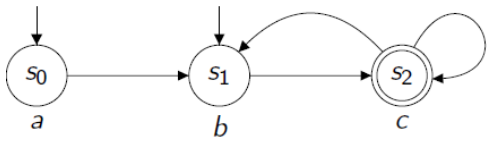
Definizione: Un Automa a stati finiti etichettato (LFSA) è una tupla $(\Sigma, S, S^0, \rho, F, L)$:

- 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.

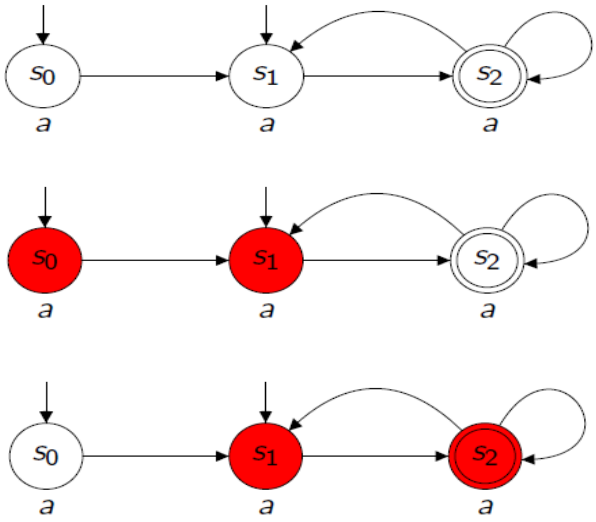
Definizione: Un LFSA è deterministico sse

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 $s_i \to s_{i+1}$ $\forall 0 \leq i < n$

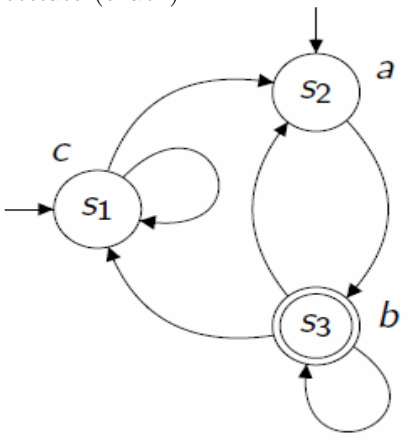
Definizione: Un run è detto ***accepting*** se $s_n \in F$

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 $\lim(\sigma) \cap F \neq \varnothing$

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

7.3.1 Linguaggio accettato

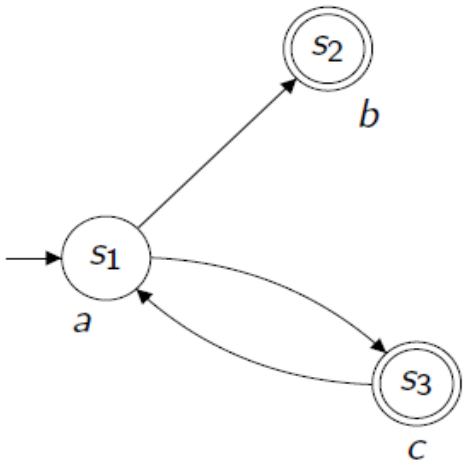
Definizione: Una parola $w = a_0\ a_1\ \ldots$ è accettata da un LBA sse esiste un accepting run

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

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 $\mathcal{L}_{\omega}(A) = \mathcal{L}_{\omega}(A')$

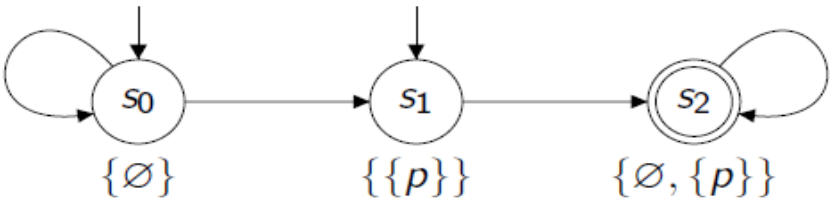
7.4 Model Checking LTL

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

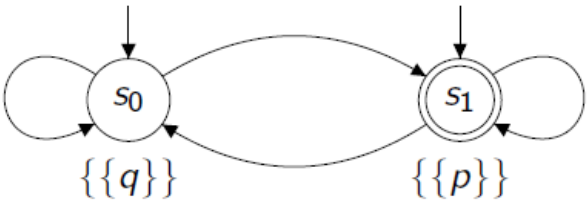
$$[\![AP_i]\!] = (\forall p \in AP_i \ : \ p) \wedge (\forall p \in AP - AP_i \ : \ \neg p)$$

- 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 $\wedge \neg$p)
s_1: (p $\wedge \neg$q)

Formula LTL:
G[(q $\wedge \neg$p) U (p $\wedge \neg$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 :

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

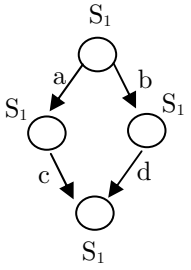
(il fattore $2^{|\phi|}$ è legato alla costruzione del grafo)

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

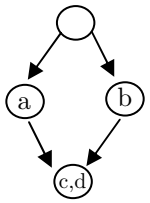
8. Algebre di Processi

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.

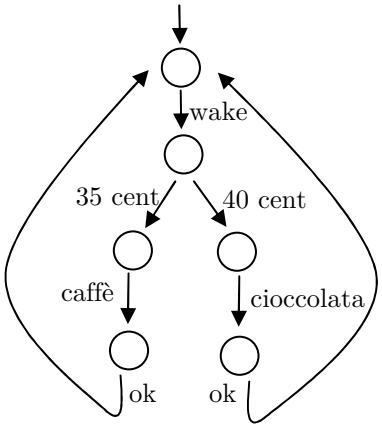
Esempio: Modellazione semplificata ad eventi della macchina del caffè di ingegneria.

Questa macchina permette di eseguire sequenze del tipo :

wake-35-caffè-ok-wake-40-cioccolata

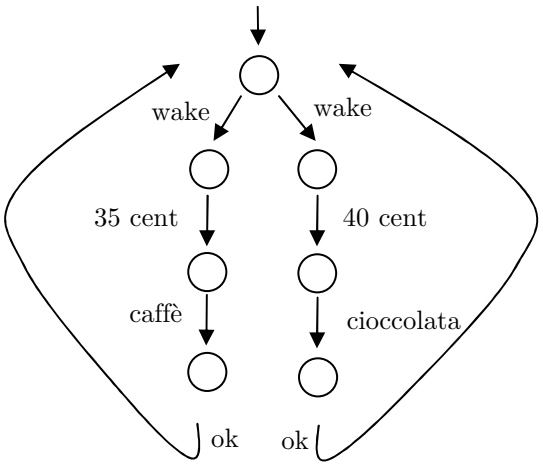
ma non:

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**.

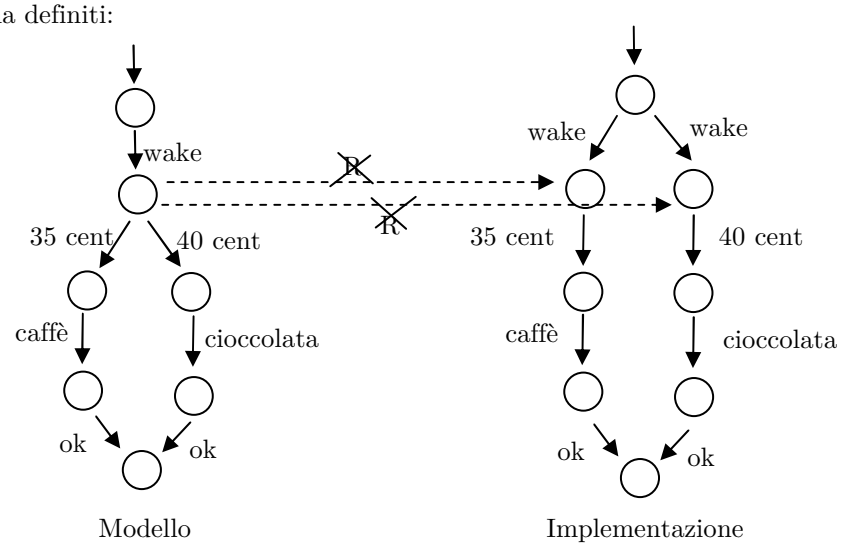
Definizione: R è una relazione di bisimulazione forte su S, S' si indica con S R S' (S e S' sono insieme degli stati dove $s_i \in S$ e $s_i' \in S'$) se $\forall s_i \in S$ e $\forall s_i' \in S'$:

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

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': $s_0 \, R \, s_0'$

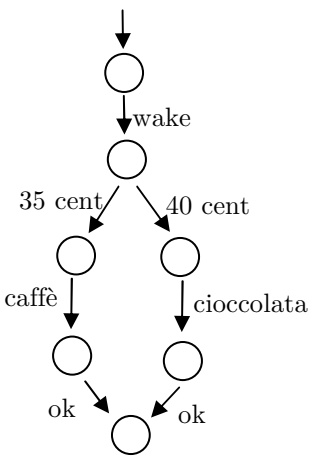
Valutiamo la relazione di bisimulazione sul modello e sull'implementazione prima definiti:



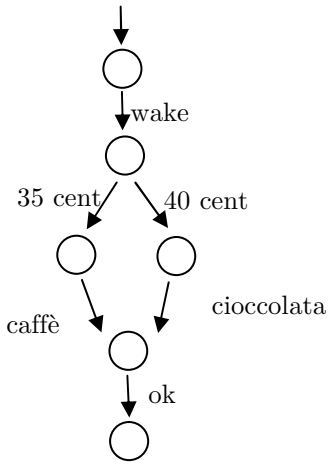
Modello Implementazione

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 $\cup$ { 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 $\subseteq$ Accs – { τ })
5. ***Relabelling***: P[f] (f: Accs $\rightarrow$ Accs)

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

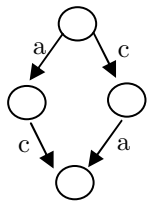
$$P ::= nil \mid a.P \mid P + Q \mid P \mid Q \mid P / L \mid P[f]$$

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

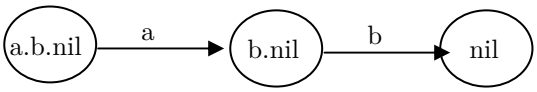
Con semantica operazionale di un'algebra di processi si intendono i passi di esecuzione che possono essere fatti da un processo. Questi comportamenti sono descritti dagli assiomi e dalle regole di transizione per gli operatori che nel CCS sono le seguenti:

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:

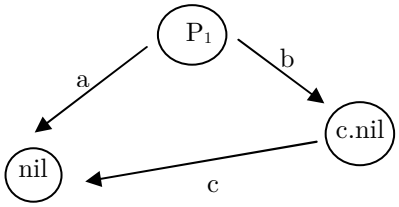
$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'}$$

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:

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

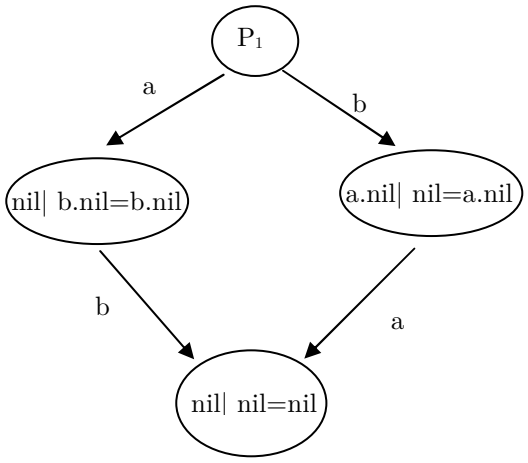
Se $P_1 \xrightarrow{a} P_1'$ allora $P_1 \mid P_2 \xrightarrow{a} P_1' \mid P_2$

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)



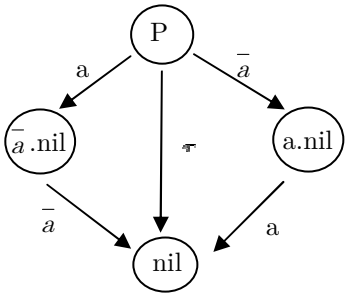
Da notare che le uguaglianze negli stati valgono perché valgono alcune regole di assorbimento:

- 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: