

Corso di Informatica Industriale

Appunti su Macchine a Stati Finiti e Sistemi Real-Time

Alessandro Fantechi
parzialmente basati sugli appunti di Claudio Guida

20 marzo 2007

Indice

1	Introduzione	2
2	Comportamento a stati finiti di un sistema embedded	2
2.1	Richiami su automi a stati finiti riconoscitori di linguaggi	2
2.2	Grammatiche	3
2.3	Linguaggi non regolari	5
2.4	Classificazione di Chomsky	5
2.5	Macchine a stati finiti	7
2.6	Implementazione di automi a stati finiti	8
2.7	Implementazione di macchine a stati finiti	9
2.8	Event-driven systems	10
3	Real Time Systems	11
3.1	Temporizzazione dell'algoritmo di controllo	11
3.2	Multitasking	14
3.3	Multitasking	14
3.4	Real-time Scheduling	14
3.5	Modello di processo semplice	15
3.6	Esecuzione ciclica	15
3.7	Scheduling basato sui processi	17
3.7.1	Modalità di scheduling	17
3.7.2	Prelazione e non-prelazione	17
3.7.3	FPS e priorità rate-monotonic	17
3.8	Test di scheduling basati sull'utilizzo	17
3.8.1	Test di utilizzo per schemi EDF	20
3.9	Processi sporadici e aperiodici	20
3.9.1	Processi forti (hard) e deboli (soft)	21
3.10	Interazione tra i processi e blocking	21
3.11	Protocolli priority ceiling	22
3.11.1	Protocolli ceiling, mutua esclusione e deadlock	23
4	Codici rilevatori di errore	23
4.1	Codici di parità	24
4.2	Codifica m-of-n	27
4.3	Duplicazione	27
4.4	Checksum	27
4.5	Codici ciclici	28
4.6	Codici aritmetici	29

1 Introduzione

Per sistema computerizzato *embedded* si intende un sistema composto da uno o più processori che fanno fisicamente parte di un sistema elettronico o meccanico e che ne controllano il funzionamento. L'utente del sistema elettronico o meccanico spesso non percepisce neppure la presenza di un processore con una funzione specifica. Il processore in un sistema *embedded* ha la funzione di monitorare (attraverso dei *sensori*) alcune grandezze fisiche e di calcolare la risposta alla loro variazione, in modo da poter agire direttamente (attraverso opportuni *attuatori*) sui dispositivi fisici circostanti variandone alcune altre grandezze fisiche. Troviamo esempi di sistemi *embedded* nella vita di ogni giorno: dalla lavatrice alla bilancia elettronica, dal televisore al lettore di CD, usiamo quotidianamente dei processori senza poterne accedere le caratteristiche funzionali che si è abituati a considerare quando accendiamo un PC. Un'automobile di classe ha al suo interno varie decine di computer, da quello ormai onnipresente della centralina che ha sostituito il carburatore, al sistema di dispiegamento degli air-bag, ai più sofisticati dispositivi come cambio sequenziale o *steer-by-wire*.

2 Comportamento a stati finiti di un sistema *embedded*

Il processore in un sistema *embedded* ha la funzione di monitorare (attraverso dei *sensori*) alcune grandezze fisiche e di calcolare la risposta alla loro variazione, in modo da poter agire direttamente (attraverso opportuni *attuatori*) sui dispositivi fisici circostanti variandone alcune altre grandezze fisiche. L'algoritmo che sovrintende al monitoraggio dei sensori e al comando degli attuatori prende il nome di *algoritmo di controllo*. L'algoritmo di controllo può essere in prima analisi classificato in due categorie principali, cioè quella degli algoritmi di controllo continui, e quella degli algoritmi di controllo discreti - non nascondendoci che in pratica si utilizzano anche algoritmi che ne ereditano entrambe le caratteristiche, detti pertanto *ibridi*.

Negli algoritmi di controllo continui, almeno in via teorica, le grandezze fisiche sono considerate come una funzione continua del tempo e il comando degli attuatori avviene attraverso una funzione continua del tempo: ad ogni istante temporale vi è una precisa relazione tra i valori di input e quelli di output del sistema di controllo. La disciplina dell'*automatica* si occupa della definizione di algoritmi di controllo continui il cui scopo principale è mantenere all'interno di ben definiti confini di stabilità il sistema al variare delle condizioni esterne, spesso mediante opportuni meccanismi di retroazione (*feedback*). Non approfondiremo nel seguito (se non per quanto riguarda gli aspetti *real-time*) questa disciplina, per la quale rimandiamo ai testi del settore.

Negli algoritmi di controllo discreti, invece, si considerano solo dei particolari istanti di tempo in cui l'algoritmo deve calcolare la risposta a stimoli provenienti dal sistema circostante; tali istanti di tempo possono essere dettati dall'occorrere di specifici eventi, oppure dal verificarsi di specifiche condizioni temporali (come, ad es., un *time-out*).

Possiamo considerare come semplice esempio di sistema con un algoritmo di controllo discreto la classica macchina distributrice del caffè da ufficio. La macchina è normalmente in uno stato di attesa (*idle*). Appena si preme un pulsante, o si inserisce una moneta, la macchina reagisce opportunamente, passando in uno stato diverso, ad es., se le operazioni svolte sulla macchinetta sono compatibili

2.1 Richiami su automi a stati finiti riconoscitori di linguaggi

Dato un insieme V di simboli, detto *alfabeto*, si denota con V^* l'insieme di tutte le parole finite composte di simboli di V . Useremo le lettere minuscole a, b, c, \dots come simboli di V , e v, w per indicare elementi di V^* . La parola vuota, indicata con λ , appartiene a V^* .

Si dice linguaggio un qualsiasi sottoinsieme (finito o infinito) di V^* . Un automa a stati finiti A sull'alfabeto V è una quadrupla (S, s_0, S_f, t) dove:

S è un insieme finito non vuoto di *stati*;

$s_0 \in S$ viene detto *stato iniziale*;

$S_f \subseteq S$ è l'insieme degli *stati finali*;

$t : S \times V \rightarrow S$ è la *funzione di transizione* tra gli stati. Quando $t(s, a) = s'$ si scrive anche $s \xrightarrow{a} s'$

Sulla base della funzione di transizione t si definisce, per transitività, la relazione di raggiungibilità \rightarrow^* : $S \times V^* \times S$: $s \xrightarrow{w} s'$ sse $w = a_1 a_2 \dots a_n$ e $\exists s_1, \dots, s_{n-1} : s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_{n-1} \xrightarrow{a_n} s'$

Un automa a stati finiti può essere rappresentato graficamente come in figura 1. Si dice che un linguaggio $L \subseteq V^*$ viene riconosciuto da un automa A se:

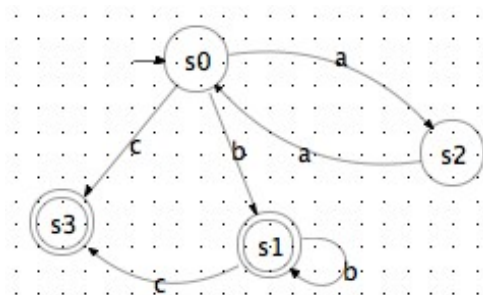


Figura 1: Esempio di automa a stati finiti

$w \in L$ sse $s_0 \xrightarrow{w} *s$, con $s \in S_f$.

Ad esempio, l'automata riportato in figura 1 riconosce il linguaggio formato da tutte le parole formate da una sequenza di un numero dispari di a , o da una sequenza (anche vuota) di un numero pari di a seguita da una sequenza (anche vuota) di b terminata da un simbolo c .

Per denotare il linguaggio generato da tale automa si può usare l'espressione regolare $a(aa)^* + (aa)^*b^*c$, dove il simbolo di somma indica l'unione di due linguaggi, e la stella indica la ripetizione dell'espressione alla cui destra è applicata. La notazione delle espressioni regolari, in varie varianti, viene spesso utilizzata per definire problemi di ricerca di stringhe in un testo.

2.2 Grammatiche

Un modo alternativo per definire un linguaggio è definendo la grammatica del linguaggio.

Una grammatica è una quadrupla $G = \langle V, N, S, P \rangle$:

- V è un insieme, detto vocabolario, i cui elementi sono detti simboli terminali.
- N è un insieme, disgiunto da V , i cui elementi sono detti categorie sintattiche.
- S è un elemento di N , detto simbolo iniziale.

- P è una relazione di N su $(N^* \cup V^*)$, ovvero un sottoinsieme del prodotto cartesiano $N \times (N^* \cup V^*)$, ovvero un insieme di coppie $\langle n, \eta \rangle$, dove n è un elemento di N , e η è una sequenza di elementi che appartengono a N o a V . Una tale coppia è detta produzione ed è comunemente denotata come $n \rightarrow \eta$.

- Derivazione diretta: siano *pre* e *post* elementi di $(N \cup V)^*$. Sia α_0 un elemento di N , e sia infine α_1 un elemento in $(N \cup V)^*$. Si dice che *pre* α_1 *post* deriva-direttamente da *pre* α_0 *post*, e si scrive *pre* α_0 *post* \rightarrow_1 *pre* α_1 *post* se P contiene la produzione $\alpha_0 \rightarrow \alpha_1$.
- Derivazione indiretta: siano α_0 e α_N elementi in $(N \cup V)^*$. Si dice che α_N deriva da α_0 , e si scrive $\alpha_0 \rightarrow \alpha_N$ se esiste una sequenza di elementi $\alpha_i \in (N \cup V)^*$ tali che per $\forall i \in [1, N]$ $\alpha_{i-1} \rightarrow \alpha_i$.
- Linguaggio definito da una grammatica: il linguaggio L_G definito dalla grammatica $G = \langle V, N, S, P \rangle$ sul vocabolario V è l'insieme degli elementi di V^* che derivano dal simbolo iniziale S attraverso le categorie sintattiche N e le produzioni P .

Si ricorda che una forma in cui vengono spesso scritte le grammatiche di un linguaggio di programmazione è la Backus-Naur-Form (BNF).

La grammatica per il linguaggio riconosciuto dall'automata di fig. 1 è la seguente:

$$\begin{aligned} S_0 &\leftarrow a S_2 \mid b S_1 \mid c S_3 \\ S_1 &\leftarrow b S_1 \mid c S_3 \\ S_2 &\leftarrow a S_0 \end{aligned}$$

La corrispondenza tra simboli non terminali della grammatica e gli stati dell'automata è evidente. Notiamo che le produzioni di questa grammatica sono tali che a membro sinistro troviamo solo un simbolo non terminale e che a membro destro troviamo o soli simboli terminali, o un simbolo non terminale preceduto da simboli terminali. Una grammatica con tutte produzioni di questo tipo prende il nome di grammatica regolare, e il linguaggio da essa derivato può essere riconosciuto da un automata a stati finiti: è infatti evidente la corrispondenza tra simboli non terminali della grammatica e stati dell'automata. I linguaggi derivabili da grammatiche regolari e riconosciuti da automi a stati finiti si dicono linguaggi regolari.

2.3 Linguaggi non regolari

Non tutti i linguaggi sono regolari. Prendiamo ad esempio il comune linguaggio delle formule matematiche. In esso troviamo l'uso delle parentesi, con la regola che ad ogni parentesi aperta deve corrispondere una parentesi chiusa. Un automa riconoscitore di tale linguaggio dovrebbe perciò essere in grado di contare quante occorrenze di parentesi aperte vi siano prima di trovare le parentesi chiuse. L'automata dovrà perciò cambiare stato ad ogni nuova parentesi aperta. A meno che non vogliamo limitare arbitrariamente il numero delle parentesi, ottenendo perciò un linguaggio finito, il numero degli stati dell'automata sarà infinito. Si può pensare comunque che tale linguaggio sia riconoscibile semplicemente aggiungendo un contatore illimitato ad un automata a stati finiti. Se però consideriamo un linguaggio dove le parentesi siano di vario tipo (ad es. le formule con parentesi tonde, quadre e graffe, o un linguaggio di programmazione dove oltre alle parentesi tonde ho le parentesi **begin end**), avere un contatore non basta a memorizzare la posizione relativa delle parentesi incontrate. La forma più semplice di riconoscitore di un linguaggio di questo tipo è il cosiddetto automata a pila (pushdown automaton), cioè un automata a stati finiti a cui si associa una pila (stack) in cui si inseriscono le parentesi aperte man mano che si incontrano nella scansione del testo. Quando si incontra una parentesi chiusa, si toglie la corrispondente parentesi dalla pila, dove si deve trovare nella posizione di testa se la parola letta fa parte del linguaggio (ovvero le parentesi sono correttamente bilanciate. Al termine della scansione della stringa in ingresso, la pila deve essere vuota, altrimenti la parola letta non appartiene al linguaggio).

Possiamo notare che in un automata a pila abbiamo aggiunto un elemento di memoria (la pila, appunto) di dimensioni illimitate. Mentre un automata a stati finiti occupa sempre una quantità di memoria finita, riuscendo a riconoscere parole di lunghezza non limitata, imporre una limitazione alla grandezza della pila significa limitare la lunghezza delle parole riconosciute (nel nostro caso il numero delle parentesi nella formula).

Un linguaggio riconoscibile da un automata a pila si dice *libero da contesto* (context-free) e la grammatica che lo genera grammatica libera da contesto (context-free grammar - CFG). Un esempio di CFG è la seguente semplice grammatica per un linguaggio di espressioni:

$$Expr ::= Const \mid Expr + Expr \mid Expr * Expr \mid (Expr)$$

in cui vi sono produzioni in cui appaiono due simboli non terminali. Si può notare come le grammatiche dei comuni linguaggi di programmazione contengano spesso produzioni di questo tipo.

Un linguaggio caratteristico della classe dei linguaggi liberi da contesto è il linguaggio $a^n b^n$, che può appunto essere considerato il più semplice linguaggio di parentesi, dove a è la parentesi aperta e b è la parentesi chiusa.

2.4 Classificazione di Chomsky

Una classe più ampia di linguaggi è quella dei *linguaggi dipendenti da contesto*: nelle grammatiche che li generano infatti vi possono essere produzioni in cui la parte sinistra non è costituita da un solo simbolo non terminale, ma definisce il *contesto* nel quale occorre trovare un simbolo nonterminale per poterlo sostituire. In generale, un linguaggio dipendente dal contesto non può essere riconosciuto da un automata a pila, ma occorre definire uno specifico algoritmo di riconoscimento (si dice che è riconoscibile da una Macchina di Turing, perchè occorre un generico esecutore di algoritmi). Un linguaggio caratteristico della classe dei linguaggi dipendenti da contesto è il linguaggio $a^n b^n c^n$.

In realtà anche la classe dei linguaggi generabili da grammatiche dipendenti dal contesto è una sottoclasse della classe dei linguaggi generabili da generiche *grammatiche a struttura di frase*, per le quali il problema dell'appartenenza di una parola al linguaggio è semidecidibile. Se chiamiamo, facendo riferimento alla classificazione di Chomsky, L_0 questa classe più ampia di linguaggi, L_1 la classe dei linguaggi dipendenti da contesto, L_2 quella dei linguaggi liberi da contesto, L_3 quella dei linguaggi regolari, L_{FIN} quella dei linguaggi finiti, si ha: $L_{FIN} \subset L_3 \subset L_2 \subset L_1 \subset L_0$.

Non è nostra intenzione approfondire ulteriormente questi temi, per i quali si rimanda ai testi di base della teoria dei linguaggi formali, ma ricordiamo che al crescere della classe dei linguaggi diminuisce la possibilità di decidere particolari problemi (equivalenza di due linguaggi, inclusione di due linguaggi, ecc...). Qui basta ricordare che se abbiamo a disposizione una quantità di memoria limitata è possibile solo riconoscere linguaggi regolari, altrimenti dovremo mettere delle limitazioni alla lunghezza delle parole riconoscibili di un linguaggio di classe superiore.

Questa classificazione trova riscontro non soltanto nel riconoscimento di linguaggi, ma più in generale per distinguere classi di algoritmi e quindi problemi che possono essere da essi risolti. Possiamo cioè considerare problemi che possono essere risolti nella loro forma più generale tramite algoritmi che necessitano di una memoria limitata a priori, e problemi che nella loro forma più generale possono essere risolti solo se non si pongono limiti alla memoria utilizzata: solo se si limita la dimensione del problema si può definire un limite alla memoria utilizzata.

Un comune esempio di un programma che ha bisogno di una memoria teoricamente illimitata è un compilatore: poiché in un qualsiasi linguaggio di programmazione vi sono strutture parentetiche, il problema della sua traduzione è intrinsecamente context-free. In pratica, per tradurre un programma C in codice oggetto è necessario, in generale, mantenere in memoria informazioni sui costrutti incontrati in precedenza, durante la scansione di tutto il programma, e queste informazioni possono crescere, senza limiti, al crescere della dimensione del programma. La memoria necessaria al compilatore, nel caso fosse insufficiente quella disponibile in RAM, può essere riservata su disco mediante il meccanismo della memoria virtuale, ma prima o poi incontreremo un limite fisico: questo limite definirà la dimensione massima del programma C effettivamente compilabile. Si noti che per la traduzione da un linguaggio assembler al linguaggio oggetto, sfruttando la corrispondenza uno ad uno tra istruzioni assembler e istruzioni macchina, e mancando in assembler strutture parentetiche, può essere definita a priori la quantità di memoria necessaria (se ignoriamo la tabella dei simboli, la cui dimensione in genere cresce con la dimensione del programma, ma che possiamo considerare fissata prima dell'esecuzione della seconda passata dell'assemblatore).

Un computer general-purpose deve prevedere la possibilità di trattare problemi non limitati a priori, e da qui nasce la necessità di avere memoria su disco, eventualmente estendibile, e meccanismi di memoria virtuale. In un sistema embedded, invece, le necessità di memoria possono essere molto ridotte: l'algoritmo di controllo di un sistema di solito richiede poca memoria sulla storia precedente del sistema, e quindi le necessità di memoria possono essere definite a priori: l'utilizzo di processori con poca memoria e senza dischi né possibilità di estensione è la norma nei sistemi embedded di piccola o media complessità. Ad esempio, la macchina distributrice del caffè deve memorizzare i comandi digitati dall'ultimo utente, e le quantità di prodotti e di monete contenuti nella macchina: in totale poche variabili intere. Il fatto che la sequenza di comandi in ingresso sia potenzialmente di lunghezza infinita non incide sulla quantità di memoria necessaria.

Così come per gli automi, in generale la limitazione a priori sulla memoria utilizzata da un programma permette la decidibilità di problemi quali l'equivalenza o la terminazione, non decidibili su programmi generici. Cioè è possibile definire un algoritmo che decide se due programmi eseguono la stessa funzione, o se un programma termina, entrambi problemi indecidibili per programmi generici. La complessità computazionale è in genere proibitiva, ma recentemente sono state introdotte tecniche innovative per ridurre tale complessità e queste possibilità di *verifica formale* di un programma stanno trovando un sempre maggiore utilizzo in quei settori dove la correttezza del software è di vitale importanza.

La limitazione a priori della memoria consente in generale un'analisi più semplice del comportamento del programma, ed è quindi consigliata o addirittura imposta (ad es. attraverso la proibizione dell'uso della allocazione dinamica - la `malloc` in C) da normative dei settori applicativi *safety-critical*, quali avionico o ferroviario.

Se la memoria di un sistema è limitata, questa memoria può assumere al massimo un numero di configurazioni fissato (2 elevato al numero di bit complessivo della memoria), e quindi posso sempre modellare il sistema come una *macchina a stati finiti*, in cui la configurazione di memoria iniziale (al reset) costituisce lo stato iniziale della macchina, e ogni configurazione di memoria raggiungibile dal sistema a partire da quella rappresenta uno dei suoi possibili stati. Una modifica alla memoria costituisce una transizione da uno stato ad un altro.

Figura 2: Macchina di Mealy

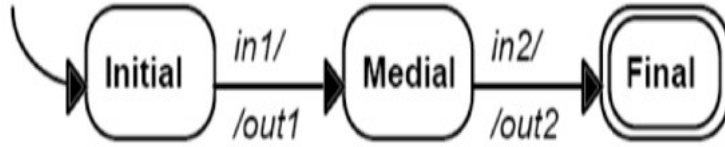
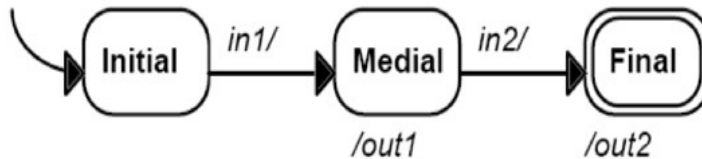


Figura 3: Macchina di Moore



2.5 Macchine a stati finiti

Le macchine a stati finiti rappresentano una maniera efficace di esprimere il comportamento complessivo di un sistema. Essere in uno stato significa che il sistema risponde solo a un sottoinsieme degli ingressi consentiti, produce solo un sottoinsieme delle possibili risposte e transisce direttamente solo verso un sottoinsieme di tutti i possibili stati.

Le due principali e ormai classiche interpretazioni delle macchine a stati finiti sono le macchine di Mealy e le macchine di Moore, che differiscono tra loro nella produzione delle risposte. In una macchina di Mealy (Figura 2) le risposte sono associate alle transizioni di stato. Quindi una macchina di Mealy può essere definita come una sestupla (S, s_0, In, Out, t, o) dove:

S è un insieme finito non vuoto di *stati*;

$s_0 \in S$ viene detto *stato iniziale*;

In è l'insieme dei possibili ingressi, detti anche *trigger*;

Out è l'insieme delle possibili risposte in uscita;

$t : S \times In \rightarrow S$ è la *funzione di transizione* tra gli stati;

$o : S \times In \rightarrow Out$ è la *funzione di uscita*.

In una macchina di Moore (Figura 2.2) le risposte sono associate agli stati invece che alle transizioni. La risposta in uscita di una macchina di Moore dipende solo dallo stato corrente, mentre l'uscita di una macchina di Mealy dipende sia dallo stato corrente che dall'ingresso ricevuto. Quindi una macchina di Moore può anch'essa essere definita come una sestupla (S, s_0, In, Out, t, o) dove l'unica differenza sta nella funzione di uscita:

$o : S \rightarrow Out$.

Le macchine di Mealy e di Moore sono matematicamente equivalenti, cioè una può sempre essere trasformata nell'altra. In generale una macchina di Mealy richiede un numero inferiore di stati per rappresentare lo stesso sistema, perchè può usare transizioni differenti (con un diverso trigger) verso lo stesso stato e può eseguire azioni differenti; una macchina di Moore deve invece usare più stati per rappresentare condizioni nelle quali vengono eseguite azioni diverse.

Le macchine a stati finiti possono essere utilizzate non solo per definire direttamente il comportamento effettivo di un sistema, utilizzando come ingressi informazioni provenienti da sensori e come uscite le informazioni da fornire agli attuatori, ma più in generale per modellare a più alto livello le fasi in cui un sistema si può trovare, lasciando ad algoritmi specifici di ogni fase la funzionalità effettiva del sistema. Un esempio di questo tipo è fornito in Figura 4 che rappresenta come macchina a stati le fasi in cui un sistema di rilevamento di oggetti all'interno di un'area


```

        break;
    case 2:
        switch(c)
            { case 'a': stato =0; break;
              case 'b': stato =4; break;
              case 'c': stato =4; break; }
        break;
    case 3: stato = 4; break;
    case 4: break;
}
getch(&c);
}
if (stato == 2 || stato == 3)
    printf("la parola appartiene la linguaggio")
else
    printf("la parola non appartiene la linguaggio")

```

Nella seconda invece si usa una matrice che definisce la relazione di transizione, ovvero restituisce il prossimo stato a partire dallo stato attuale e dal simbolo letto.

```

int mat [4][2];
char c;
int stato = 0; int i;
getch(&c);
while(c != eol)
{ switch(c)
    { case 'a': i =0; break;
      case 'b': i =1; break;
      case 'c': i =2; break; }
  stato = mat[stato][i];
}
getch(&c);
}
if (stato == 2 || stato == 3)
    printf("la parola appartiene la linguaggio")
else
    printf("la parola non appartiene la linguaggio")

```

Si noti che tale implementazione è generica, cioè permette di realizzare un qualsiasi automa sull'alfabeto $\{a, b, c\}$ fornendone la matrice di transizione. La matrice `mat` per l'automata in questione è la seguente:

	a	b	c
0	2	1	3
1	4	1	3
2	0	4	4
3	4	4	4
4	4	4	4

2.7 Implementazione di macchine a stati finiti

L'implementazione di una macchina a stati finiti non è dissimile da quella di un automa precedentemente vista, e può utilizzare sia il metodo basato su uno switch che quello tabellare, che usa una matrice (in particolare, una matrice per il prossimo stato e una per l'uscita nel caso di una macchina di Mealy, o una matrice per il prossimo stato e un vettore per l'uscita nel caso di una macchina di Moore. Possiamo astrarre da questa scelta individuando due funzioni `next` e

output; assumendo anche una procedura che ritorna il valore di input letto, una implementazione generica può avere la seguente forma per una macchina di Mealy:

```
int c;
int stato = 0; int i;
getch(&c);
while(TRUE)
  {input(&c);
   stato = next(stato, c);
   output(stato);
  }
```

e la seguente forma per una macchina di Moore:

```
int c;
int stato = 0; int i;
getch(&c);
while(TRUE)
  {input(&c);
   output(stato,c);
   stato = next(stato, c);
  }
```

Si noti che la principale differenza con l'implementazione di un automa riconoscitore di linguaggi sta nell'utilizzo di un ciclo infinito; in effetti un sistema embedded che deve controllare in continuazione un sistema o processo fisico spesso non prevede la terminazione. Comunque, nel caso sia necessaria, la terminazione si può comunque modellare con uno stato da cui non si può uscire.

Altra cosa che appare evidente è come, una volta sia disponibile la definizione della macchina a stati, ad es. mediante la tabella di transizione, il processo di scrittura del codice sia totalmente meccanico e possa essere automatizzato. In effetti strumenti di descrizione di Macchine a stati finiti quali Stateflow di Matlab permettono di generare codice direttamente dalla specifica grafica delle macchine a stati.

2.8 Event-driven systems

Un sistema è detto reattivo o event-driven quando, a seguito della ricezione di eventi provenienti dall'esterno, risponde con una reazione. Molti casi di sistemi embedded possono essere visti come sistemi event-driven: un evento può essere notificato ad un sistema principalmente in due modi, che possono essere definiti sincrono o asincrono: vale a dire con la variazione del valore di una interfaccia in ingresso (quindi di una variabile, che può corrispondere ad un registro di ingresso), oppure attraverso una segnalazione asincrona quale un'interruzione. Nel primo caso il sistema può ciclicamente verificare il valore della variabile in ingresso, con un algoritmo di questo tipo:

```
void main()
{ while(TRUE)
  if (input variato)
    {reazione all'input}
}
```

Se vi sono più variabili in ingresso, vi sarà un if per ciascuna variabile.

Un'implementazione di questo tipo si dice con *attesa attiva* perchè il processore attende la variazione degli ingressi senza compiere alcuna operazione. Si noti che la reazione ad un input può essere realizzata con una macchina a stati finiti, in cui la chiamata alla funzione di transizione `next` dell'implementazione vista sopra incapsuli il test sulla variabile di input, e in cui una variabile .

Un'altra possibilità di realizzazione di sistemi event driven è attraverso le interruzioni. Un evento viene associato ad una interruzione, e la relativa routine di interruzione implementa la reazione all'evento. In questo caso il processore può essere semplicemente impegnato in una attesa attiva delle interruzioni con un semplice ciclo infinito `while(TRUE)`;

Anche in questo caso le routine di interruzione possono realizzare le funzioni di transizione e di output di una macchina a stati, purchè venga mantenuto lo stato corrente in una variabile globale di stato, che deve essere inizializzata per denotare lo stato iniziale prima che il processore entri in attesa attiva. Comunque, in entrambe le soluzioni la struttura dell'algoritmo di controllo è comunque costituita da una fase di inizializzazione seguita da un ciclo infinito. All'interno del ciclo, eventualmente tramite le routine delle interruzioni, si leggono gli ingressi e si compiono le opportune azioni sulle uscite: in un sistema embedded, si parla per queste due fasi di lettura dei sensori e di comando degli attuatori.

Si noti invece che le due implementazioni di un sistema event-driven si comportano diversamente da un punto di vista temporale, soprattutto quando vi siano più eventi da trattare. Infatti se gli eventi non si sovrappongono temporalmente, nella soluzione con interruzioni l'evento viene servito immediatamente, mentre nella soluzione in cui un ciclo testa periodicamente tutte le variabili di ingresso (questo metodo viene anche chiamato *polling*) l'evento viene servito solo quando il processore arriva a testare la relativa variabile di ingresso. Questo differente trattamento ha quindi effetto sul *tempo di risposta* all'evento e se ne deve quindi tener conto quando il tempo di risposta è un importante requisito. Sistemi con requisiti sui tempi di risposta si dicono sistemi in tempo reale (*real-time*) ed esigono una trattazione specifica.

3 Real Time Systems

Una prima classificazione dei sistemi real-time è quella tra *hard* e *soft* real-time:

Soft realtime:

I vincoli sui tempi di risposta non sono stringenti, perchè in qualche caso si può non rispettarli.

Esempio: i programmi di visualizzazione di un DVD devono estrarre i dati e processarli in modo da rispettare i vincoli di un frame ogni 50-esimo di secondo. Ma in casi di computazioni particolarmente complesse (ad es. compensazione di errori, o impegno della CPU in un altro compito, se i vincoli non sono rispettati si produce solamente un peggioramento della qualità dell'immagine.

Hard realtime:

I vincoli sui tempi di risposta sono stringenti, perchè se non vengono rispettati il sistema di controllo è inutile, o addirittura pericoloso.

Esempio: programmi di controllo delle superfici di volo di un aereo.

Data l'attenzione che poniamo in questo contesto sui sistemi embedded, riteniamo utile concentrarci sui sistemi hard real-time.

3.1 Temporizzazione dell'algoritmo di controllo

Abbiamo già visto come nel caso di algoritmi di controllo di tipo discreto (affidati cioè, ad esempio, a macchina a stati finiti) la struttura tipica dell'algoritmo sia un ciclo infinito che alterna la lettura dei sensori al calcolo delle risposte e del prossimo stato, al comando attuatori.

Molti sistemi embedded trovano applicazione in sostituzione di sistemi di controllo analogici preesistenti, dove il controllo viene invece effettuato in modo continuo: le grandezze fornite dai sensori vengono elaborate in modo analogico per fornire comandi agli attuatori in ogni istante, spesso in quello che si chiama anello di controreazione (feedback loop). Si rimanda ai testi di controlli automatici per la teoria del controllo continuo. Nel controllo digitale di questi sistemi, si deve però far ricorso ad una discretizzazione temporale del controllo, vale a dire le grandezze misurate dai sensori possono essere rilevate solo ad istanti di tempo discreti, e così pure i comandi agli attuatori possono essere inviati solo ad istanti di tempo discreti.

Il teorema del campionamento (erroneamente detto di **Nyquist-Shannon**) definisce la frequenza minima di **campionamento** di un **segnale**, necessaria per evitare **distorsioni** dello stesso.

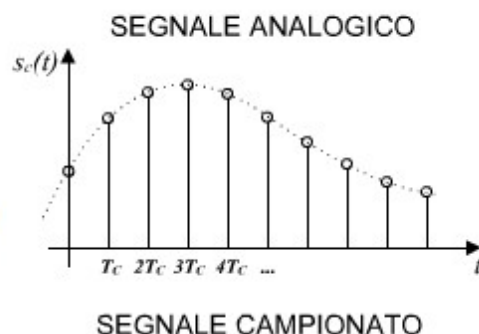
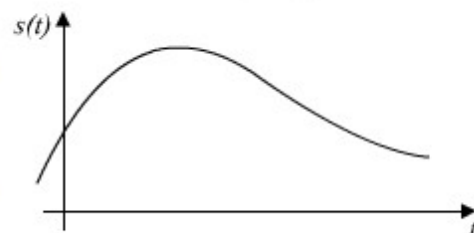
Dato un segnale, con **larghezza di banda** finita e nota, la **frequenza** minima di campionamento di tale segnale deve essere almeno il doppio della sua massima frequenza.

Il teorema, comparso per la prima volta nel 1949 in un articolo di **C. E. Shannon**, dovrebbe chiamarsi Whittaker-Nyquist-Kotelnikov-Shannon, secondo l'ordine cronologico di chi ne dimostrò versioni via via più generalizzate.

Il campionamento è un passo del processo di **conversione analogico-digitale** di un segnale. Consiste nel prelievo di campioni (*samples*) da un segnale analogico e continuo nel tempo ogni Δt secondi.

Δt è l'*intervallo di campionamento*, mentre $F_s = \frac{1}{\Delta t}$ è la *frequenza di campionamento*. Il risultato è un segnale analogico in tempo discreto. Tale segnale sarà in seguito quantizzato, codificato e quindi reso accessibile a qualsiasi elaboratore digitale.

In pratica il teorema del campionamento pone un vincolo per la progettazione di apparati di conversione analogico-digitale: se si ha a disposizione un campionatore che lavora a frequenza F_s , è necessario mandargli in ingresso un segnale a banda limitata da $\frac{F_s}{2}$.



Un segnale $f(t)$ a banda limitata da f_M può essere univocamente ricostruito dai suoi campioni $f(n\Delta t)$ ($n \in \mathbb{N}$) presi a frequenza $F_s = \frac{1}{\Delta t}$, se $F_s \geq 2f_M$.

Il teorema del campionamento di Nyquist-Shannon (spesso detto di Shannon), di cui riportiamo nel riquadro la definizione ripresa da Wikipedia, garantisce che se un segnale viene campionato ad una sufficiente frequenza, l'insieme dei campioni basta a ricostruire il segnale. Si può perciò compiere una discretizzazione delle misure continue ottenute da sensori analogici, su cui si eseguirà una computazione discreta che darà in uscita una serie discreta di valori verso gli attuatori, che di fatto non li distingueranno da una variazione continua del valore, sempre grazie al teorema del campionamento. Si definisce perciò per un dato sistema controllato la frequenza tipica di controllo, che sarà maggiore della frequenza minima di campionamento dei sensori. Tale frequenza f definisce il periodo $T = 1/f$ dell'algoritmo di controllo.

Quindi in un sistema di controllo real-time, sia basato su un controllo continuo discretizzato, sia basato su un controllo discreto con macchina a stati, si può definire il periodo tipico dell'algoritmo di controllo. Ad ogni periodo viene eseguito il codice dell'algoritmo, che di solito è costituito dalla lettura dei sensori, dal calcolo delle uscite e dal comando degli attuatori. Il codice ripetuto ogni periodo viene detto *task* (compito). Più avanti risulterà evidente la somiglianza tra i *task* e i processi di un sistema operativo: per questo motivo spesso *task* e *processo* saranno considerati sinonimi.

Per rendere il periodo di un processo dipendente dal tempo è possibile sincronizzare l'esecuzione del codice del *task* con un evento di timer. Il codice seguente mostra un loop sincronizzato con un timer.

```

main(){
    /* inizializzazione */
    while(TRUE){
        while(!Timer){
            /* task code */
        }
    }
}

```

In questo caso, `Timer` sarà una variabile, spesso un registro dedicato a questo scopo, che viene automaticamente messa a 1 quando scade il periodo, e rimessa a 0 poco dopo. Si noti la similarità con la prima implementazione proposta un sistema event-driven: in questo caso l'evento è lo scadere del periodo. Similmente, si può ipotizzare una soluzione con interruzioni. Un comune modo di realizzare questo meccanismo è un registro che viene automaticamente decrementato ad ogni ciclo di clock, e che genera un'interruzione quando il suo valore arriva a zero. Quindi l'inizializzazione del sistema dovrà settare il registro al valore del periodo (misurato in cicli di clock); la scadenza del periodo darà perciò un'interruzione, e la routine di gestione dell'interruzione sarà il codice del task. La prima istruzione della routine di interruzione dovrà però settare di nuovo il registro timer al valore del periodo. Si noti che questa soluzione richiede uno specifico supporto hardware, che peraltro è fornito in molti microcontrollori o processori industriali disponibili sul mercato.

Un altro metodo per sincronizzare il task con il tempo reale è di far riferimento al clock di sistema. Se suppongo di avere a disposizione una funzione di sistema che ritorna il valore attuale del clock (di un tipo `time` che supporremo definito), possiamo in ogni ciclo controllare se il tempo attuale ha raggiunto il valore dato dal periodo sommato al tempo memorizzato all'inizio del periodo:

```

main(){ time t;
    /* inizializzazione */
    t = getclock();
    while(TRUE){
        if(getclock() + Periodo == t)
        {
            t = getclock();
            /* task code */
        }
    }
}

```

Si noti che per il corretto funzionamento di tale soluzione occorre che il clock sia una funzione monotona crescente del tempo fisico: questa proprietà non è sempre garantita dal clock di un sistema operativo convenzionale, che permette aggiustamenti all'indietro del clock sia da parte dell'utente, sia come azione propria del sistema (ad es. in occasione del cambio di ora da legale a solare).

In tutte le soluzioni che abbiamo visto, abbiamo in realtà supposto che l'esecuzione del codice del task termini sempre prima dello scadere del periodo: se questo non avviene, il task non garantisce il controllo corretto del sistema. Occorre pertanto assicurarsi che il tempo effettivo di esecuzione del codice del task sia sempre limitato superiormente da un valore minore del periodo. Se questo non accade, è necessario velocizzare l'esecuzione del codice, o tramite l'utilizzo di un processore più veloce, di ottimizzazione del codice, o di algoritmi più veloci. Infatti il tempo effettivo di esecuzione dipende da tutti questi fattori.

L'analisi del tempo di esecuzione di un task deve necessariamente considerare il caso peggiore: il Worst Case Execution Time (WCET) dà così un limite superiore al tempo di esecuzione del task, tenendo conto che:

- Il calcolo del WCET, da effettuare andando a vedere quante (e quali) istruzioni vengono effettivamente effettuate dal task nel caso peggiore, è in generale indecidibile, per la possibile presenza di cicli illimitati all'interno del codice del task
- se si considerano programmi ristretti di cui si conosce un limite superiore al numero di cicli. il calcolo è semplice per architetture del processore semplici o vecchie ,
- molto complesso per architetture moderne con pipeline, cache, memoria virtuale, etc.

- richiede l'analisi del codice macchina
- esistono comunque strumenti automatici per il calcolo del WCET.

Alternativa: misurazione dei tempi effettivi di esecuzione del codice (attendibilità solo statistica).

3.2 Multitasking

D'altra parte, nel caso che il WCET sia molto minore del periodo, la disponibilità di potenza di calcolo esuberante rispetto alle necessità date dal periodo di un task di controllo porta naturalmente alla tendenza ad unificare varie funzioni di controllo su uno stesso processore. Si parla in questo caso di real time multitasking: più task, ognuno con le sue caratteristiche funzionali e temporali, vengono eseguiti dallo stesso processore, e quindi sorge il problema del corretto sequenziamento (scheduling) dei task in modo da rispettare tutti i vincoli temporali esistenti.

3.3 Multitasking

Al fine di trattare correttamente questa problematica, occorre introdurre ancora altra terminologia.

Esistono due parametri che caratterizzano un processo real-time: il periodo T_p e il tempo di esecuzione T_t . Tramite questi due parametri è possibile calcolare il carico sulla CPU per l'esecuzione del processo in questione:

$$L = \frac{T_t}{T_p} \quad (1)$$

Dall'equazione 1 si nota che aumentando il periodo T_p , o diminuendo il carico sulla CPU diminuisce. In questo modo si rende possibile eseguire altri processi.

I tasks di un sistema real-time si distinguono in *periodici* e *aperiodici*, a seconda che debano essere eseguiti periodicamente o no.

Tasks aperiodici che richiedono il processore in modo imprevedibile sono detti *sporadici* (se c'è una separazione minima tra due invocazioni). Questi task sono quelli che gestiscono eventi (vedi Sistemi event-driven), e che in genere devono rispettare requisiti relativamente ai tempi di risposta.

I rimanenti tasks aperiodici (tipicamente task di gestione o di diagnostica) sono anche detti task di background perchè in genere vengono eseguiti quando non vi sia necessità di eseguire task periodici o sporadici.

Ad ogni istante, ad un task (periodico o sporadico) può essere associata una deadline, ovvero l'istante di tempo in cui la sua esecuzione deve essere conclusa. Per i task periodici la deadline è calcolata sulla base del loro periodo: siccome devono essere eseguiti compiutamente allo scadere del periodo, la deadline è data dall'istante di tempo in cui scade il prossimo periodo. Per i task sporadici la deadline è calcolata sulla base dei tempi di risposta richiesti, ovvero la deadline è pari al tempo di rilascio più il tempo di risposta richiesto. Per questi task, valgono le considerazioni implementative fatte per i sistemi event-driven, quindi si può ad esempio pensare che la condizione di rilascio del task venga periodicamente testata: se T è il periodo di questo test, e T_r è il tempo di risposta richiesto, e w il WCET del task, deve essere $T + w \leq T_r$. Per questo motivo i task sporadici possono venir trattati come task periodici, come faremo nel seguito, salvo poi studiare in dettaglio i tempi di risposta effettivamente raggiungibili (cosa che non faremo in questo ambito).

La distinzione tra sistemi soft e hard real-time può essere descritta in termini di deadline come segue: in un sistema hard real-time il non rispetto di una singola deadline può provocare malfunzionamenti anche gravi del sistema, mentre in un sistema soft real-time tale evento diminuisce la qualità del servizio offerto dal sistema, possibilmente senza abbassarla sotto una data soglia di accettabilità.

Ad ogni task viene associata una priorità, che permette di guidare la scelta di quali task eseguire per primi. La priorità è un numero intero. In questo testo useremo un valore più alto per i task a maggiore priorità, ma occorre tener presente che altri testi, e alcuni sistemi operativi real time, indicano una maggiore priorità con un valore più basso. Tipicamente, i processi di background avranno priorità minore dei processi periodici, in modo da rendere disponibile il processore a garantire il rispetto delle deadline, e i task sporadici avranno maggiore priorità di quelli periodici, per

garantire il rispetto dei tempi di risposta richiesti. La priorità potrebbe essere assegnata anche sulla base della criticità del task, vale a dire dalla gravità del possibile mancato rispetto della deadline da parte di quel task. Ma in generale, in un sistema real-time la priorità deve essere assegnata in modo da garantire il rispetto delle deadline, e come vedremo potrà anche variare dinamicamente.

3.4 Real-time Scheduling

Uno schema di scheduling sostanzialmente consiste in due caratteristiche:

- ordinamento delle risorse – in particolare della CPU – rispetto al loro utilizzo mediante un algoritmo.
- previsione del comportamento del sistema nel caso peggiore.

La previsione, ad esempio, può essere utilizzata per confermare che le richieste temporali del sistema siano soddisfatte. Uno schema di scheduling può essere statico, se le decisioni – predizioni – vengono prese prima dell'esecuzione dei processi, oppure dinamico, se le decisioni vengono prese durante la loro esecuzione. In questo paragrafo ci occuperemo degli schemi statici.

3.5 Modello di processo semplice

La previsione del comportamento di un sistema molto complesso può essere difficile. Per questo motivo è necessario imporre delle restrizioni alla struttura dei programmi concorrenti. Il modello base, che vedremo in questo paragrafo, presenta le seguenti caratteristiche:

- si assume che l'applicazione sia composta da un insieme fisso di processi,
- ogni processo è considerato periodico con periodo conosciuto,
- i processi sono indipendenti
- tutti costi sono ignorati, ad esempio il cambio di contesto,
- tutti i processi devono terminare nel periodo, cioè devono essere completati prima di una nuova esecuzione,
- ogni processo ha un tempo di esecuzione nel caso peggiore fisso.

La condizione di indipendenza tra i vari processi permette di assumere che ad un certo istante tali processi verranno eseguiti tutti nello stesso istante. Questo istante rappresenta il momento di massimo carico sul processore e viene chiamato *critical instant*. Nella tabella 1 sono elencate alcune notazioni standard per la descrizione delle caratteristiche dei processi.

Tabella 1: Notazioni standard per le caratteristiche dei processi

Simbolo	Descrizione
B	tempo di blocco del processo nel caso peggiore
C	tempo di computazione del processo nel caso peggiore (WCET)
D	deadline del processo
I	tempo di interferenza del processo
J	tempo di avvio del processo (jitter)
N	numero dei processi nel sistema
P	Priorità assegnata al processo
R	tempo di risposta del processo nel caso peggiore
T	tempo minimo tra ciascuna esecuzione del processo (periodo del processo)
U	utilizzo del processo $U = C/I$
$a - z$	nome del processo

3.6 Esecuzione ciclica

Dato un insieme fisso di processi periodici è possibile stendere uno schema di scheduling la cui esecuzione ripetuta provochi il funzionamento dei processi alla loro frequenza. Uno schema di questo tipo può essere visto come una tabella di procedure dove ciascuna procedura rappresenta una parte di codice del processo. L'intera tabella viene chiamata *major cycle*, mentre ciascuna procedura in essa viene chiamata *minor cycle*. Ogni minor cycle ha lunghezza fissata. Quindi quattro minor cycle da 25 ms compongono un major cycle da 100 ms. Un clock genera un'interruzione ogni 25 ms dando modo allo scheduler di passare da una procedura all'altra. Nella tabella 2 è mostrato un insieme di processi che deve essere implementato mediante un major cycle di quattro slot. Una possibile implementazione è quella fornita dal seguente codice:

```
loop
  wait_interrupt;
  procedure_a;
  procedure_b;
  procedure_c;
  wait_interrupt;
  procedure_a;
  procedure_b;
  procedure_d;
  procedure_e;
  wait_interrupt;
  procedure_a;
  procedure_b;
  procedure_c;
  wait_interrupt;
  procedure_a;
  procedure_b;
  procedure_d;
  wait_interrupt;
end loop
```

Tabella 2: Insieme di processi

Processo	Periodo T	Computazione C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Nonostante la semplicità, questo esempio illustra alcune caratteristiche importanti di questo approccio:

- al momento dell'esecuzione non esiste nessun processo vero e proprio: ogni minor cycle è solo una sequenza di chiamate a procedure,
- le procedure condividono un'area di indirizzamento comune. Questo permette il passaggio di dati. Si noti che non è necessario proteggere tale area – ad esempio mediante un semaforo – in quanto l'accesso concorrente a più procedure non è consentito.
- Ogni periodo di un processo deve essere un multiplo del tempo di un minor cycle.

Questa ultima caratteristica rappresenta uno degli svantaggi maggiori dell'esecuzione ciclica, assieme ai seguenti:

- difficoltà di incorporare processi sporadici,

- difficoltà di incorporare processi con periodo lungo. Infatti il periodo del major cycle è il periodo più lungo che può essere utilizzato senza l'intervento di scheduling secondario,
- la difficoltà di costruire effettivamente il ciclo di esecuzione,
- un processo con un tempo di computazione variabile deve essere suddiviso in parti fisse. Questo significa che il codice nelle procedure potrebbe venir tagliato il che potrebbe portare ad errori.

Se è possibile costruire un ciclo di esecuzione non si ha ulteriore necessità di test di scheduling in quanto la correttezza di questo schema è dimostrata dalla sua costruibilità. Comunque, per sistemi ad alto utilizzo, la costruzione di un ciclo di esecuzione diventa problematica. Per comprendere quanto appena detto si noti che il problema della creazione di un ciclo di esecuzione può essere paragonato con il classico problema dei recipienti: dobbiamo mettere degli oggetti di varia grandezza (problema unidimensionale) nel minor numero di recipienti in modo tale che nessun recipiente risulti "strapieno". Il problema dei recipienti è conosciuto come problema di complessità NP . Per questo motivo non è pensabile l'utilizzo di schemi di complessità NP in caso di problemi di grandezza variabile. E' necessario dunque utilizzare dei metodi ottimizzati.

3.7 Scheduling basato sui processi

Mediante la costruzione di un ciclo di esecuzione non è possibile utilizzare una differente sequenza di chiamate a procedura durante l'esecuzione del ciclo. Questo tipo di schema di scheduling non supporta il concetto di processo. Un'alternativa consiste nel mantenere il concetto di processo e di supportarne la sua esecuzione direttamente. Quanto appena detto può essere eseguito mediante l'esame di una o più variabili di scheduling. A questo punto un processo può trovarsi in diversi stati. Si supponga che i processi non comunichino tra loro, allora gli stati in cui un processo può trovarsi sono i seguenti:

- eseguibile
- sospeso in attesa di un evento temporale – utile per processi periodici,
- sospeso in attesa di un evento non temporale – appropriato per processi sporadici.

3.7.1 Modalità di scheduling

Esistono varie modalità di scheduling. Il *Fixed-Priority Scheduling* (FPS) il quale consiste nell'assegnare una priorità fissa prima dell'esecuzione ai processi e ordinare l'esecuzione secondo le priorità assegnate. In un sistema real-time, ad esempio, la priorità è data dai requisiti sul tempo. Un altro tipo di schema di scheduling è il cosiddetto *Earliest-Deadline First* (EDF). In questo schema l'ordine di esecuzione è determinato dal tempo rimanente alla conclusione del processo. Quindi il processo che viene scelto per l'esecuzione è il processo più vicino al termine. Poiché la scelta del processo da eseguire viene fatta in run-time questo tipo di schema è dinamico.

3.7.2 Prelazione e non-prelazione

In un modello con priorità, può accadere che durante l'esecuzione di un processo, un processo con priorità più elevata arrivi alla coda dei processi. In un caso di sistema con prelazione, il processo con priorità minore, viene solitamente interrotto per permettere al processo con priorità maggiore di essere eseguito. Differentemente, in un sistema senza prelazione, il processo con priorità minore viene prima completato. Tra le due situazioni estreme: con prelazione e senza prelazione, esistono alternative intermedie. Ad esempio, esistono sistemi che, in presenza di processi con priorità maggiore rispetto a quella del processo in esecuzione, concedono un periodo limitato di tempo al processo con priorità minore, scaduto il quale tale processo viene sospeso a vantaggio del processo con priorità maggiore. Sistemi di questo tipo vengono chiamati *deferred preemption* oppure *cooperative dispatching*.

3.7.3 FPS e priorità rate-monotonic

Esiste un semplice e ottimale metodo di assegnamento per la priorità chiamato *rate-monotonic* il quale consiste nell'assegnare una priorità rispetto al periodo del processo. In particolare si assegna una alta priorità ai processi con breve periodo, quindi, dati due processi i e j rispettivamente con periodo T_i e T_j abbiamo che se $T_i < T_j$ allora

$P_i > P_j$. Si noti che se un insieme di processi può essere organizzato secondo lo scheduling FPS allora tale insieme può anche essere organizzato secondo lo schema rate-monotonic. Data la costruzione delle priorità si noti che i valori più grandi sono relativi ai processi con priorità maggiore. La tabella 3 mostra un esempio di insieme di processi e di assegnazione della priorità.

Tabella 3: Esempio di priorità rate-monotonic

Processo	Periodo	Priorità
a	25	5
b	60	2
c	50	3
d	40	4
e	100	1

3.8 Test di scheduling basati sull'utilizzo

In questa sezione verrà mostrato un metodo di testing per la correttezza del modello di scheduling FPS, che, sebbene non sia molto preciso, viene spesso utilizzato per la sua semplicità. E' stato dimostrato che un insieme di processi ordinato secondo lo schema FPS, considerando solo l'utilizzo, è possibile formulare una condizione – test – per la *schedulability*. Se la condizione seguente viene verificata, tutti gli N processi appartenenti all'insieme dei processi potranno soddisfare i loro requisiti in termine di deadline:

$$\sum_{i=1}^N \frac{C_i}{T_i} \leq N \left(2^{1/N} - 1 \right) \quad (2)$$

La tabella 4 mostra il limite di utilizzo – espresso in percentuale – per un insieme di processi con N piccolo. Al crescere di N l'utilizzo tende asintoticamente al 69.3% per cui ogni insieme di processi con un requisito di utilizzo minore del 69.3% viene sempre accettato in uno schema con prelazione a priorità di tipo rate-monotonic. Vediamo

Tabella 4: Limite di utilizzo

N	Utilizzo
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

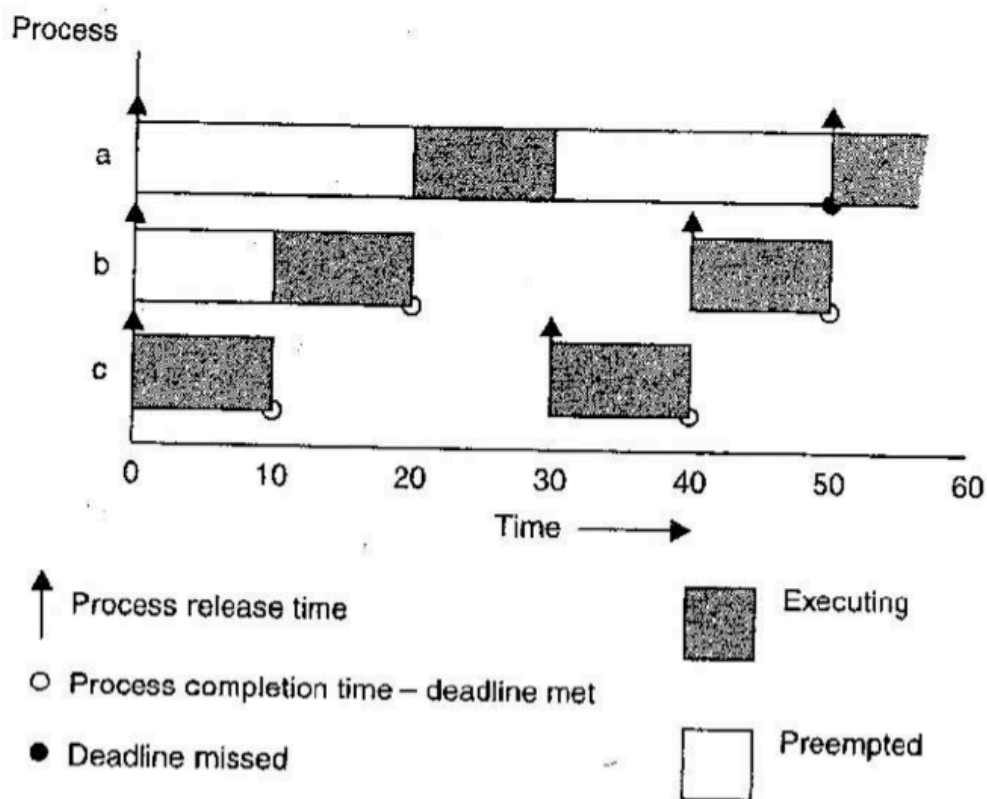
alcuni esempi di applicabilità del test proposto. Dal momento che i tempi T , C ecc. sono della stessa unità di misura il test è applicabile. La tabella 5 mostra un insieme di tre processi. Come si può notare la combinazione di utilizzo

Tabella 5: Insieme di tre processi

Processo	Periodo T	Computazione C	Priorità P	Utilizzo U
a	50	12	1	24%
b	40	10	2	25%
c	30	10	3	33%

dei tre processi è pari all'82% (somma dei singoli utilizzi). Dalla tabella 4 si nota che questo valore di utilizzo è

Figura 5: Andamento temporale dell'insieme di processi



superiore al valore di soglia per tre processi (cioè 78%). Questo insieme di processi fallisce quindi il test di utilizzo. Per comprendere il motivo si osservi la figura 5 che mostra l'andamento temporale dell'esecuzione dei tre processi. Si osservi che una deadline viene persa. Il secondo esempio viene mostrato nella tabella 6. Per questo insieme l'utilizzo totale vale 77.5% che è al di sotto della soglia per tre processi. Se disegnassimo il diagramma del tempo per questo insieme tutte le deadline verrebbero raggiunte. Sebbene il processo di costruzione del diagramma temporale sia

Tabella 6: Insieme di tre processi

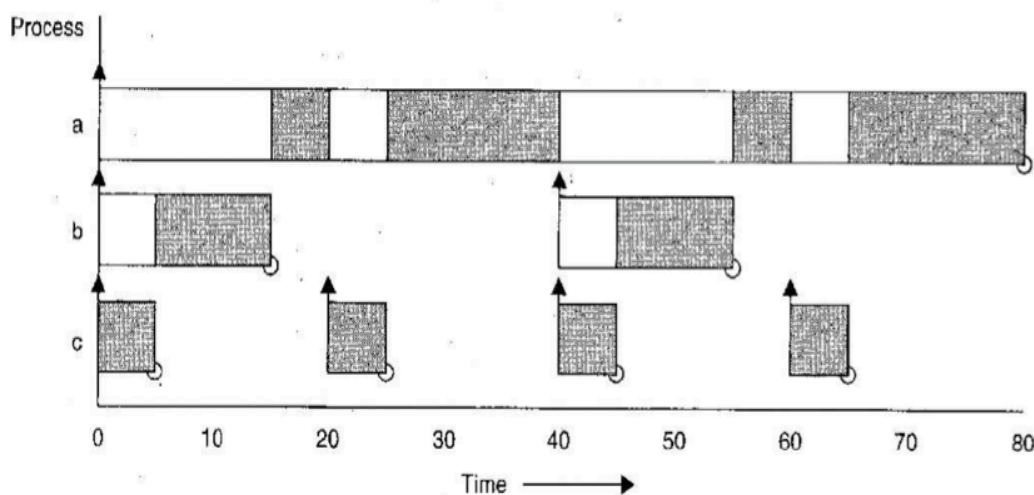
Processo	Periodo T	Computazione C	Priorità P	Utilizzo U
a	80	32	12	40%
b	40	5	2	12.5%
c	16	4	3	25%

abbastanza oneroso, viene considerato uno strumento utile per testare la possibilità di eseguire uno scheduling su un insieme di processi. La domanda che dobbiamo porci a questo punto è la seguente: quanto dobbiamo estendere la linea temporale per essere sicuri di poter affermare che il futuro non riservi sorprese? Nel caso di insiemi di processi che condividono il tempo di avvio, cioè che condividono un critical instant, si può dimostrare che una linea temporale lunga quanto il periodo maggiore è sufficiente. In questo modo se le deadline vengono raggiunte in questo periodo di tempo si può essere sicuri che lo saranno anche nel futuro. Un ultimo esempio di insieme di processi è mostrato nella tabella 7. Come si può notare l'utilizzo complessivo è del 100%. Questo insieme fallisce chiaramente il test. Dall'esame della figura 6 si nota però che il comportamento run-time di questo insieme è comunque corretta e tutte le deadline vengono raggiunte. Quindi si può affermare che il test è *sufficiente*, ma non *necessario*. Per cui se un

Tabella 7: Insieme di tre processi

Processo	Periodo T	Computazione C	Priorità P	Utilizzo U
a	80	40	1	50%
b	40	10	2	25%
c	20	5	3	25%

Figura 6: Esempio di insieme che fallisce il test, ma ha un run-time corretto



insieme di processi passa il test si può dire che sicuramente il sistema rispetta i requisiti di tempo dei processi, mentre se l'insieme fallisce il test può fallire – o no – l'esecuzione run-time. Il test presentato in questo paragrafo restituisce una risposta semplice di tipo “sì/no”, ma non fornisce nessuna informazione sulla risposta temporale dei processi. Un tipo di test che fornisce tali informazioni verrà esaminato più avanti.

3.8.1 Test di utilizzo per schemi EDF

E' possibile utilizzare un test di utilizzo simile a quello precedentemente presentato anche per gli schemi EDF. La condizione che deve essere rispettata quindi è la seguente:

$$\sum_{i=1}^N \frac{C_i}{T_i} \leq 1 \quad (3)$$

Il test per uno schema EDF è quindi più semplice in quanto qualsiasi combinazione di utilizzo è consentita. Questo significa che per qualunque insieme di processi le deadline vengono sempre raggiunte. In questo senso EDF è superiore in qualità rispetto a FPS per il quale invece l'insieme di processi deve rispettare un certo limite di utilizzo. Detto questo potremmo chiederci il perché EDF non sia lo schema comunemente utilizzato. La risposta a questa domanda deriva dal fatto che FPS presenta dei vantaggi rispetto a EDF:

- FPS è più facile da implementare, in quanto il parametro di scheduling per FPS – la priorità – è un parametro statico, mentre EDF è uno schema dinamico per cui necessita di un sistema più complesso durante il run-time,
- è più semplice inserire processi senza deadline in FPS (assegnando semplicemente la priorità),
- i deadline non sono l'unico parametro importante. Inoltre è possibile tener conto di vari parametri nell'assegnare la priorità.

- nelle situazioni critiche (come situazioni di guasto) il comportamento dello schema FPS è molto più prevedibile rispetto allo schema EDF. Infatti nello schema FPS i processi che perdono le loro deadline per primi sono quelli a minor priorità, mentre nello schema EDF potrebbe verificarsi una reazione domino nella quale molti processi perdono le proprie deadline.

3.9 Processi sporadici e aperiodici

Per espandere il semplice modello presentato nella prima sezione di questo capitolo introduciamo i processi sporadici, o aperiodici. Con l'introduzione di questo tipo di processi, il periodo T deve essere considerato come un tempo minimo – o media – del tempo di arrivo. Ad esempio, per un processo sporadico con $T = 20$ ms è garantito che non arrivi più di una volta in un intervallo di 20 ms. In realtà la frequenza di esecuzione è molto minore di una volta per intervallo. Comunque il test del tempo di risposta assicura che la massima frequenza può essere sostenuta, ovviamente se il test viene superato correttamente. Un altro requisito da considerare quando vengono inclusi processi sporadici consiste nella ridefinizione dei deadline. Il modello semplice visto in precedenza prevedeva che il deadline di un processo – cioè il termine – coincidesse con il proprio periodo, cioè $D = T$. Nel caso di processi sporadici questo non può essere più assunto come corretto. Spesso i processi sporadici vengono invocati come gestori di errore o di situazioni critiche per cui la loro esecuzione, oltre ad essere urgente, deve essere anche rapida. Il modello semplice per cui deve essere ulteriormente esteso. Più precisamente deve poter distinguere tra T e D e permettere $D < T$.

3.9.1 Processi forti (hard) e deboli (soft)

Per i processi sporadici devono essere definite la frequenza media e la frequenza massima di arrivo. Sfortunatamente, in molti casi il caso peggiore è molto più grande rispetto al caso medio. Questo comporta un basso utilizzo del processore. Come linee guida possiamo seguire quindi le seguenti due regole:

1. tutti i processi dovrebbero essere gestiti con scheduling utilizzando il tempo medio di esecuzione e il tempo medio di arrivo,
2. tutti i processi fortemente real-time dovrebbero essere gestiti con scheduling utilizzando il tempo di esecuzione nel caso peggiore ed il tempo di arrivo sempre nel caso peggiore.

Una conseguenza della regola 1 consiste nel fatto che si potrebbero verificare situazioni nelle quali non tutte le deadline dei processi vengono rispettate. Questa situazione viene chiamata sovraccarico transitorio (*transient overload*). La regola 2 invece assicura che nessun processo fortemente real-time perda le proprie deadline. Nel caso in cui la regola 2 desse luogo ad un eccessivo basso utilizzo del processore durante la “normale esecuzione” sarebbe necessario intervenire per cercare di ridurre il tempo di esecuzione del caso peggiore o il tempo di arrivo.

3.10 Interazione tra i processi e blocking

Nella stesura dei punti che compongono il modello semplice di scheduling del primo paragrafo abbiamo supposto che i processi non interagiscano tra loro. Questa situazione però non è realistica e, in quasi tutti i sistemi, i processi hanno necessità di comunicare per il corretto funzionamento. Come è noto la comunicazione tra processi può avvenire tramite semafori, monitor, punti di incontro ecc. Utilizzando queste tecniche un processo può restare in attesa del verificarsi di un evento. L'esame del caso peggiore in una configurazione di processi intercomunicanti è più complessa in quanto è difficile determinare la vera situazione di caso peggiore, soprattutto in presenza di molte dipendenze tra i processi. In una situazione nella quale i processi comunicano tra loro può capitare che il modello della priorità venga minato. Infatti può accadere che un processo venga sospeso in favore di un processo a minor priorità in quanto quest'ultimo deve eseguire delle operazioni necessarie al proseguimento del primo processo. In una situazione ideale un processo ad alta priorità non dovrebbe attendere per un processo a bassa priorità. Quando questo accade si dice che il processo sospeso è bloccato (*blocked*). Ai fini del test per lo scheduling, il fenomeno di blocking deve essere limitato, misurabile e piccolo. Illustriamo un esempio estremo, cioè di priorità inversa. Supponiamo di avere quattro processi a , b , c e d con priorità assegnata in modo tale che il processo ad alta priorità sia d ed il processo a bassa priorità sia a . La tabella 8 mostra i dati relativi ai quattro processi. Si supponga inoltre che il processo d condivida con a la risorsa Q e condivida invece con c la risorsa V . Nella tabella è riportata anche la sequenza di esecuzione nella quale 'E' sta per singolo tick di esecuzione, 'Q' e 'V' stanno per singolo tick di esecuzione nel quale si accede alla sezione critica della risorsa Q e V). Il processo a inizia per primo, al tempo 0 ed entra nella sezione critica per

Tabella 8: Esempio di priorità invertita

Processo	Priorità	Sequenza di esecuzione	Tempo di avvio
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

la risorsa Q . Quindi il processore viene prelato in favore del processo c a sua volta sospeso a favore del processo d . Prima di essere sospeso però il processo c accede alla sezione critica della risorsa V . Il processo d quindi continua la sua esecuzione fino a quando non necessita di utilizzare la risorsa Q . Tale risorsa però è chiusa in quanto la sua sezione critica è stata acceduta in precedenza dal processo a e non ancora rilasciata. Il processo d deve essere quindi sospeso. Il processo c riprende il controllo del processore e termina il suo ciclo. A questo punto è il turno del processo b , nel frattempo avviato, ma sospeso al tempo 2 per problemi di priorità (b e c vengono avviati nello stesso momento, ma c ha priorità maggiore rispetto a b). Quando b termina, il processo a è finalmente in grado di riacquisire il controllo e rilasciare, al termine delle operazioni, la risorsa Q . Solo a questo punto il processo d può essere risvegliato e portato a conclusione. Dall'esame del ragionamento appena fatto si nota che il processo d soffre fortemente di una inversione di priorità. Infatti sicruamente un'attesa è necessaria per mantenere il rispetto della sezione critica e quindi per garantire l'integrità dei dati. Tale attesa è il tempo in cui il processo a utilizza la risorsa Q , ma la maggior parte di attesa che d deve sperimentare è dovuta ai processi b e c . Quest'ultima componente del tempo di risposta per d è assolutamente eccessiva e incide fortemente sulle operazioni di scheduling. Il risultato appena visto è dovuto allo schema di priorità fissa utilizzato. Per ovviare a questo problema viene utilizzato uno schema chiamato *priority inheritance* nel quale se un processo p deve attendere per il completamento delle operazioni di un processo q con priorità minore rispetto a p allora la priorità di q viene posta pari alla priorità di p . In questo modo q viene eseguito in maniera preferenziale rispetto agli altri processi. Nel caso dell'esempio appena proposto, ad a verrebbe assegnata la priorità di d in questo modo a può tranquillamente completare ed il tempo di risposta di d diminuisce fortemente. Il metodo appena indicato non è valido solo per un solo processo, ma può essere esteso. Se il processo d attendere per il processo c che a sua volta deve attendere per il processo b , si portano entrambi i processi b e c alla priorità di d . In questo modo la priorità dei processi cambia durante il funzionamento della macchina ed è meglio scegliere il giusto processo nel momento in cui si presenta la necessità di eseguire o rendere eseguibile un processo piuttosto che scegliere un processo da avviare da una lista ordinata secondo una priorità fissata.

3.11 Protocolli priority ceiling

La priorità inheritance – ereditata – fornisce un limite superiore al numero di blocchi (sospensioni) che un processo ad alta priorità può incontrare. Comunque questo limite può portare ad un calcolo inaccettabilmente pessimistico per quanto riguarda il caso peggiore. Questo è dovuto alla possibilità di catene di blocco che si sviluppano tra i processi, cioè il processo c è bloccato dal processo b il quale è a sua volta bloccato dal processo a e così via. Per cui, il blocking non deve essere solo minimizzato, ma devono essere eliminate anche le situazioni di deadlock. I protocolli *priority ceiling* risolvono questo tipo di problema. Esistono sostanzialmente due tipi di protocolli, quello originale *original ceiling priority protocol* – OCPP – ed una sua variante. Quando uno di questi due protocolli viene utilizzato su un singolo sistema:

- un processo ad alta priorità può essere bloccato al massimo una volta da un processo a bassa priorità,
- i deadlock sono prevenuti,
- i blocchi transitivi sono prevenuti (c bloccato da b il quale è bloccato da a ecc.),
- la mutua esclusione per gli accessi alle risorse è garantita dal protocollo stesso.

L'idea di base dei protocolli priority ceiling sta nell'utilizzare il concetto di risorsa e cioè della sezione critica. In particolare se un processo a sta utilizzando una risorsa e potrebbe quindi provocare il blocco multiplo di processi con priorità superiore, allora viene negata la possibilità di accedere ad altre risorse, che potrebbero bloccare il processo

b , da parte di tutti i processi tranne che da a . In questo modo un processo verrà ritardato non solo quando cerca di accedere ad una risorsa già in uso da un altro processo, ma anche quando tenta di accedere a risorse che potrebbero generare blocchi dei processi con priorità maggiore. Il protocollo originale prende la seguente forma:

1. ogni processo ha assegnata una priorità statica, ad esempio mediante deadline-monotonic,
2. ogni risorsa ha un valore di ceiling. Questo valore è il valore massimo della priorità dei processi che utilizzano tale risorsa: *tetto* (ceiling),
3. ogni processo ha una priorità dinamica che è il massimo della sua priorità statica e di quella che eredita mediante la sospensione dei processi con priorità maggiore,
4. un processo può “ritenere” (lock) una risorsa solo se la sua priorità dinamica è più alta del valore di ceiling delle risorse correntemente inaccessibili (locked), tranne quelle che esso stesso detiene.

3.11.1 Protocolli ceiling, mutua esclusione e deadlock

Sebbene gli algoritmi che definiscono i due tipi di protocolli siano basati sulla chiusura delle risorse, è da notare che i protocolli, rispetto ad altre primitive di sincronizzazione, implementano una mutua esclusione sulle risorse. In questo modo, se un processo sta utilizzando una risorsa, non può esistere nessun altro processo con priorità più alta del precedente. Così il processo può continuare l'esecuzione senza impedimenti o, nel caso di prelazione, il nuovo processo non potrà utilizzare tale risorsa. Infine i protocolli ceiling sono liberi dai deadlock in quanto sono una forma di prevenzione. Quando un processo mantiene l'accesso su una risorsa mentre sta chiedendo l'accesso ad un'altra, la seconda risorsa non deve avere un valore di ceiling inferiore alla prima. I protocolli ceiling sono una specie di prevenzione per i deadlock in quanto, quando un processo accede ad una risorsa la sua priorità viene alzata al ceiling della risorsa. In questo modo un altro processo di quelli che possono accedere a tale risorsa non potrà quindi utilizzarla in quanto la sua priorità è minore di quella del processo che correntemente la sta utilizzando.

4 Codici rilevatori di errore

La tecnica della ridondanza dell'informazione consiste nell'aggiunta di informazioni ripetute in modo tale da consentire il rilevamento, il mascheramento e a volte la tolleranza dei fallimenti. Una *codifica* – code – è una maniera di rappresentare l'informazione o i dati, secondo un insieme di regole predefinito. Un *code-word* è un insieme di simboli, digits se il sistema è numerico, utilizzato per rappresentare una porzione di dati secondo una codifica specificata. Un code-word viene chiamato *valido* se rispetta tutte le regole definite dalla codifica, altrimenti viene chiamato *non valido*. Il processo di *codifica* consiste nel prendere il dato originale e trovare il code-word corrispondente secondo la codifica scelta, cioè rappresentare il dato originale secondo le regole della codifica utilizzata. Il processo di *decodifica* consiste nel recuperare, a partire dal code-word in questione, il dato originale che esso rappresenta. Un ruolo di primaria importanza è rivestito dalle codifiche binarie. Può accadere però che se in un code-word binario uno dei suoi bit viene alterato, il code-word stesso rimanga sempre nell'insieme dei codici validi pur essendo incorretto per il dato originale che rappresenta. E' comunque possibile ovviare a questo problema progettando codifiche per le quali l'insieme dei code-word validi sia un sottoinsieme delle possibili combinazioni di zero e uno. In questo modo, se la codifica è stata progettata correttamente, l'alterazione di un singolo bit comporta, come risultato, un code-word che appartiene all'insieme dei codici illegali o non validi. Il concetto appena esposto è la base del *rilevamento degli errori mediante tecniche di codifica*. Il concetto di base della correzione dell'errore invece, consiste nel fatto che i code-word sono strutturati in un certo modo per cui è possibile recuperare il code-word corretto a partire da un code-word corrotto o errato.

Si noti che un errore può essere modellato da un vettore di m bit, dove m è il numero di bit della parola considerate: ad esempio, un errore singolo, che comporta cioè l'alterazione di un singolo bit può essere modellato da un vettore di errore costituito da zeri, fuori che nella posizione corrispondente al bit errato (es. 00000100 rappresenta l'errore del bit di posizione 2 su un byte). Data una code-word w , vista anch'essa come vettore, la somma vettoriale (cioè bit a bit) modulo 2 della code-word con il vettore dà la parola errata. Si ricorda che la somma modulo 2 è l'or esclusivo.

Un concetto importante per la caratterizzazione di una codifica risiede nella *distanza di Hamming* definita nel modo seguente: la distanza di Hamming tra due code-word è data dal numero delle posizioni dei bit nelle quali le due parole differiscono. Si definisce *distanza di una codifica* come la minima distanza di Hamming calcolata tra tutte le

possibili coppie di code-word della codifica. Si osservi che se due code-word hanno distanza di Hamming pari a 1 è sempre possibile, alterando un bit ad uno dei due, farlo coincidere con l'altro. Se invece due code-word hanno distanza di Hamming pari a 2, non è assolutamente possibile, a paritè da una delle due parole, recuperare l'altra alterando solo un bit in quanto i bit nei quali differiscono sono 2. Questo mostra come la distanza di Hamming sia utile ai fini del rilevamento degli errori. Se una codifica è caratterizzata da una distanza di Hamming pari a 2, tutti gli errori su singolo bit sono rilevabili. Allo stesso modo, se una codifica è caratterizzata da una distanza di Hamming pari a 3, tutti gli errori su singolo bit e tutti gli errori su doppio bit sarebbero rilevabili. Inoltre si noti che per una codifica con distanza di Hamming pari a 3, volendo, sarebbe possibile anche una correzione dell'errore su singolo bit in quanto il code-word risultante da un errore su singolo bit avrebbe una distanza di Hamming pari ad 1 rispetto ad uno solo dei code-word validi – cioè quello corretto – e almeno pari a 2 rispetto a tutti gli altri.

Un'altra caratterizzazione di una codifica può essere fatta sulla base della sua *ridondanza*, ovvero del costo aggiuntivo richiesto rispetto ad una codifica minima. La ridondanza può essere espressa come il rapporto tra il numero di bit della codifica e il numero di bit strettamente necessario a rappresentare tutte le informazioni codificate. La ridondanza può essere direttamente usata per determinare il costo della codifica. Di solito, una codifica con maggiore distanza di Hamming, e quindi migliore copertura di errori, ha ridondanza, e quindi costo maggiore.

D'altra parte, il costo della codifica può essere anche limitato dalla semplicità della codifica e decodifica; a questo proposito, un concetto importante consiste nella *separabilità*. Una codifica separabile consiste nel conservare intatta l'informazione codificata, alla quale vengono concatenate le informazioni di controllo. Quindi, l'informazione originale concatenata con le informazioni di controllo aggiuntive forma la code-word. In questo modo, in fase di decodifica, l'unica cosa da fare è eliminare le informazioni di controllo e mantenere solo quelle di interesse. D'altra parte una codifica *non separabile* non possiede la proprietà della separabilità. Per questo motivo il recupero dell'informazione originale deve essere eseguito mediante una decodifica più complicata.

Infine, un altro aspetto desiderabile da una codifica è la sua *capacità diagnostica*, cioè la capacità non solo di rilevare la presenza di un errore (e quindi di un guasto che ha provocato l'errore) ma anche di localizzare il guasto, indicare cioè quale componente è guasto. In generale, per avere capacità diagnostica occorre maggiore ridondanza, e quindi costo maggiore. Inoltre, la capacità diagnostica può essere utilizzata non soltanto per guidare la manutenzione del sistema, ma direttamente per correggere l'errore: se la capacità diagnostica è tale da fornire il vettore di errore, sommandolo (bit a bit, in modulo 2 - cioè con una batteria di XOR) alla parola errata si ottiene la parola corretta. Un codice con tale capacità è quindi un *codice correttore di errore*.

4.1 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: parità pari e parità dispari. 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 ad 1, 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 si siano stati errori. In caso di mancata verifica della parità l'utente viene avvertito. La codifica/decodifica della parola, anche con questa semplice tecnica, comporta un hardware aggiuntivo, inoltre è necessario che la memoria abbia la capacità di contenere il bit aggiuntivo. L'hardware deve essere progettato quindi per la creazione e controllo dell'extra-bit. Si può notare, da questo piccolo esempio, che l'utilizzo delle tecniche di ridondanza, comporta sempre un hardware aggiuntivo. Si noti però che il controllo di parità con singolo bit può non rilevare alcuni errori su più bit, che invece sono piuttosto comuni nelle memorie. Solitamente una memoria è composta da chip individuali ciascuno contenente alcuni bit (un numero abbastanza comune è quattro). La codifica di parità con singolo bit può non essere in grado di rilevare errori derivanti da chip malfunzionanti. Sono state proposte quindi alcune varianti rispetto all'idea fondamentale, ne vedremo alcune.

Bit-per-word questa tecnica è stata appena discussa: aggiungere un bit di parità alla parola. Il principale svantaggio consiste nel fatto che alcuni errori possono non essere rilevabili. Ad esempio un fallimento completo, di un bus o dei buffer, che porta tutti i bit a 1 può essere rilevato da una parità dispari solo se il numero di bit è pari, lo

stesso fallimento viene rilevato da una parità pari solo se il numero di bit della parola, compreso il bit di parità, è dispari. D'altra parte il fallimento che porta tutti i bit a 0 non può essere mai rilevato dalla parità pari in quanto una parola che vale zero viene considerata con un numero pari di 1. Viceversa questo fallimento viene sempre rilevato dalla parità dispari.

Bit-per-byte questa tecnica consiste nell'utilizzare due bit di controllo e suddividere quindi la parola in due byte, un bit controlla un byte, l'altro bit controlla l'altro byte. Sebbene la tecnica si chiami bit-per-byte non è necessario che i gruppi siano esattamente di 8-bit. Per raggiungere completamente i vantaggi offerti da questa tecnica è necessario che il numero di bit di un gruppo sia pari e che un byte venga controllato con parità pari e l'altro con parità dispari. In questo modo entrambi gli errori con tutti i bit a 0 e a 1 vengono rilevati. Inoltre questa tecnica è in grado di rilevare tutti gli errori su due bit purché un errore avvenga in un byte e l'altro errore avvenga nell'altro. Lo svantaggio di entrambe le tecniche: bit-per-word e bit-per-byte consiste nel fatto che gli errori multipli non sono sempre rilevabili. Considerando che i bit che formano una parola sono spesso assegnati a chip separati, come abbiamo già detto, il fallimento di uno dei chip non viene identificato.

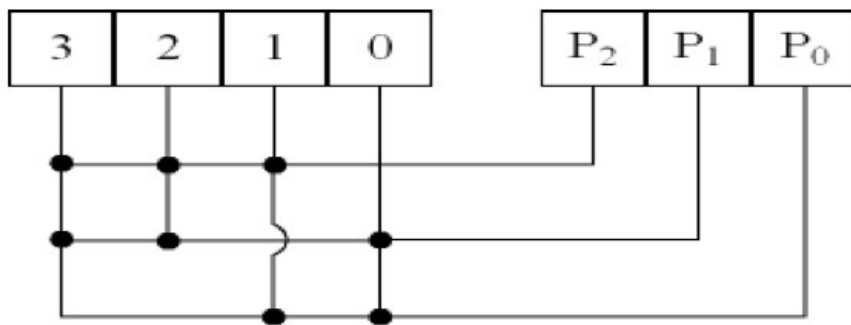
Bit-per-multiple-chip per risolvere questo problema viene utilizzato un numero di bit di parità pari al numero di bit contenuti in un chip. Ciascun bit viene associato ad un solo bit di ciascun chip in questo modo un bit di parità controlla un solo bit di ogni chip. Come si può facilmente notare questa tecnica di parità è in grado di rilevare gli errori dovuti al fallimento di un intero chip, fallimento denominato come *whole-chip failure mode*. Sebbene questa variante della codifica di parità sia in grado di rilevare il fallimento di un intero chip non è però in grado di localizzare il chip danneggiato.

Bit-per-chip questa variante è stata sviluppata per ovviare al problema appena accennato. Funziona nel modo seguente: ciascun bit di parità viene assegnato ad un singolo chip. In questo modo non solo viene rilevato il fallimento di singolo chip, ma dato che ogni bit di parità è associato ad un unico chip, il fallimento è anche localizzabile. Lo svantaggio primario è però lo stesso presentato dalla tecnica bit-per-word. Infatti il semplice controllo di parità su una parola non è in grado di rilevare errori su più bit, di conseguenza se all'interno di un singolo chip si verificano errori multipli, tali fallimenti potrebbero non venire rilevati.

Interlaced un altro tipo di organizzazione dei bit consiste nella parità interlacciata. È simile, in concetto, alla parità bit-per-multiple-chip. La differenza sostanziale consiste nel fatto che questo tipo di parità non tiene conto dell'organizzazione fisica dei bit come accade per la multiple-chip. La parola viene suddivisa in gruppi di uguale lunghezza di bit e i bit di parità vengono organizzati in modo tale che non sia possibile che due bit adiacenti della parola appartengano allo stesso gruppo. La parità interlacciata è utile, come si può immaginare, per il rilevamento degli errori sui bit adiacenti, tipici dei collegamenti a bus.

Overlapped parity L'ultimo tipo di parità che esaminiamo ha come idea di base che un bit appartenga a più di un gruppo di parità. Questo permette, oltre ad una rivelazione degli errori, anche la loro localizzazione in questo modo, mediante un'operazione di complemento, l'errore può essere corretto. Il concetto di base di questa tecnica consiste nel porre ciascun bit di parità in modo che controlli una combinazione unica dei bit della parola. Per comprendere meglio quanto appena detto si osservi la figura 7. Nella figura sono anche mostrati i bit coinvolti da un fallimento sui bit di parola e di parità. Questo tipo di parità può essere implementata mediante una serie di comparatori e un decoder con una circuiteria di controllo di parità. L'operazione di correzione dell'errore può essere fatta mediante il complemento del bit errato. Ad esempio, quando una parola viene memorizzata in una cella di memoria vengono costruiti i bit di parità aggiuntivi e memorizzati assieme alla parola. Durante l'operazione di lettura i bit di parità vengono nuovamente generati e confrontati con quelli memorizzati. Il risultato del confronto permette quindi la correzione dell'eventuale errore. Vediamo come funziona: si noti la tabella mostrata in figura. Tale tabella mostra i bit di parità coinvolti quando si verificano errori sui bit 3, 2, 1 ecc. Ora, si supponga di memorizzare la parola 1101. I bit di parità per questa parola sono i seguenti 100, per cui la parola complessiva è 1101100. Si supponga invece di leggere la parola 1001100. Come si può notare si è verificato un errore sul bit 2. Vediamo come può essere corretto. Come abbiamo detto quando si recupera una parola dalla memoria i bit di parità devono essere nuovamente generati. Stando alla parola letta i nuovi bit di parità sono i seguenti 010. Il confronto tra i vecchi bit di parità e i nuovi appena calcolati mostra due bit in disaccordo e più precisamente P_2 e P_1 . Dalla tabella in figura la configurazione dei bit affetti riferita alla riga P_2, P_1 si recupera il bit che causa questa combinazione – unica – e cioè il bit 2, proprio il bit che è effettivamente errato. A questo punto con un'operazione di complemento è possibile recuperare la parola originale. Si noti che,

Figura 7: Esempio di parità overlapped



BIT ERROR	COMBINATION
3	$P_2 P_1 P_0$
2	$P_2 P_1$
1	$P_2 P_0$
0	$P_1 P_0$
P_2	P_2
P_1	P_1
P_0	P_0

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:

$$2^k \geq m + k + 1 \quad (4)$$

4.2 Codifica m-of-n

Questa tecnica è concettualmente molto semplice: si utilizza un code-word di lunghezza n -bit nel quale si hanno esattamente m -bit a 1. In questo modo quando si verifica un errore il numero di bit a 1 nel code-word varia e diventa $m - 1$ o $m + 1$, in questo modo gli errori possono essere facilmente rilevati. Sebbene la tecnica sia semplice nell’idea che sta alla base, il processo di codifica e decodifica può risultare molto complesso. Per semplificare questi processi, data una parola con i -bit, si è soliti aggiungere altri i -bit e settare tali bit in modo da avere, nell’intera parola un numero pari ad i -bit a 1. In questo modo viene costruita una codifica di tipo i -of- $2i$. Lo svantaggio primario di una codifica i -of- $2i$ consiste nella ridondanza che infatti è del 100%. D’altra parte le operazioni di codifica e decodifica sono estremamente semplici in quanto la codifica i -of- $2i$ è separabile. Infatti l’operazione di codifica può essere eseguita contando il numero di bit a 1 contenuti nell’informazione e, dopo aver controllato in un’apposita tabella, completare gli i -bit di controllo. L’operazione di decodifica consiste semplicemente nel rimuovere gli i -bit aggiuntivi e mantenere l’informazione originale. La codifica m-of-n è una codifica con distanza di Hamming pari a 2 per cui tutti gli errori su singolo bit vengono rilevati. Si noti quindi che errori su doppi bit possono passare inosservati, ad esempio si pensi ad un errore su un bit che modifichi il suo valore da 0 a 1 ed un altro errore che modifichi il valore di un altro bit da 1 a 0. In questo caso il numero di bit a 1 rimarrebbe sempre m e il doppio errore non sarebbe rilevato. Si noti però che se gli errori multipli sono unidirezionali, e cioè tutti i bit coinvolti nell’errore cambiano da 0 a 1 o da 1 a 0, la codifica m-of-n è in grado di rilevarli. Per cui m-of-n è in grado di rilevare tutti gli errori su singolo bit e tutti gli errori multipli purché sia unidirezionali. La codifica m-of-n può essere applicata anche in modo più efficiente di quanto abbiamo visto, ma in questo caso la proprietà di separabilità viene persa.

4.3 Duplicazione

La tecnica di duplicazione consiste nel ripetere l’informazione originale. Il vantaggio primario di questa codifica consiste nella semplicità in quanto basta concatenare all’informazione originale una sua copia. Lo svantaggio principale invece consiste nella presenza di bit extra, precisamente, data una parola di i -bit, il code-word è formato quindi da $2i$ -bit. In caso di errore su singolo bit il rilevamento è facile in quanto le due metà del code-word non coincidono. La duplicazione è utilizzata in molte applicazioni, anche nelle memorie di sistema e nella trasmissione dei dati. Il problema principale però consiste nella diminuzione della velocità di trasferimento infatti dobbiamo trasmettere $2i$ -bit per recuperare un’informazione di soli i -bit. Una variazione della duplicazione standard consiste nella complementazione del duplicato. Questa variante è molto utile quando l’informazione originale deve essere processata dallo stesso hardware. Il vantaggio principale di tutte le varianti della duplicazione consiste nella semplicità. D’altra parte però la duplicazione comporta un 100% di ridondanza rispetto alle informazioni. Ad esempio nelle applicazioni di memoria, una parola deve essere scritta e letta due volte, nella trasmissione di dati, una parola deve essere trasmessa due volte. In aggiunta ad una ridondanza di hardware per la codifica, della ridondanza rispetto ai bit di informazione bisogna considerare anche un raddoppio di tempo.

4.4 Checksum

La codifica checksum è un tipo di codifica separabile che viene maggiormente impiegata nelle situazioni in cui un blocco di informazioni deve essere trasferito da un punto ad un altro. La codifica checksum viene spesso utilizzata nei trasferimenti da dispositivi di memorizzazione di massa – come dischi – verso computer e nelle reti a commutazione di pacchetto. Esistono fondamentalmente quattro tipi di checksum. Durante la creazione delle informazioni originali viene costruita la checksum e aggiunta di seguito al blocco originale. In ricezione la checksum viene rigenerata e confrontata

con quella ricevuta. Fondamentalmente la codifica checksum consiste nella somma delle parole che compongono il blocco di dati da inviare. La differenza tra i vari tipi di checksum consiste nel metodo con cui questa somma viene eseguita. Il checksum più semplice consiste nel *single-precision*. Questa tipologia di checksum consiste nel sommare tra loro tutte le parole contenute nel blocco senza considerare l'eventuale riporto. In altre parole se il blocco è composto da parole di n -bit, il checksum è anch'esso costituito da n -bit anche se la somma eccede $2^n - 1$, cioè la somma viene eseguita in modulo n . Come ci si potrà immaginare la difficoltà primaria di questa tecnica consiste nel fatto che l'informazione, e quindi la capacità di rilevamento degli errori, ignorando la condizione di overflow, viene persa. Per risolvere questo problema viene spesso utilizzata la codifica *double-precision checksum* la quale consiste nell'eseguire la somma tra le coppie di parole che compongono il blocco non più modulo n , ma modulo $2n$. In altre parole considerare il checksum di lunghezza $2n$ -bit. La condizione di overflow è comunque possibile, ma questa volta si tratta di overflow su una somma di $2n$ -bit anziché di una somma su n -bit. La terza forma di checksum viene chiamata *Honeywell checksum*. Questo tipo di codifica consiste nel concatenare due parole alla volta del blocco ed eseguire il checksum su questa nuova struttura. Ad esempio se il blocco è costituito da k parole di n -bit, il nuovo blocco sarà costituito da $k/2$ parole costituite da $2n$ -bit. Questa variante permette di identificare gli errori che si verificano sempre sulla stessa posizione, cioè su una stessa colonna. L'ultima tipologia di checksum consiste nella *residue-checksum*. Questa tecnica è identica alla *single-precision checksum* tranne per il fatto che il bit di riporto nell'addizione del bit più significativo non viene ignorato, ma viene invece aggiunto al risultato. Il problema fondamentale con la codifica checksum consiste nel fatto che un errore, se pur rilevato, non può essere localizzato, questo comporta che tutto il blocco sul quale il controllo del checksum è fallito debba essere ricorretto.

4.5 Codici ciclici

La caratteristica dei codici ciclici è data dal fatto che lo shift con riporto di un code-word genera sempre un altro code-word. Un codice ciclico è univocamente, e completamente, identificato dal suo polinomio generatore $G(X)$ che è un polinomio di grado $(n - k)$ o superiore dove n rappresenta il numero di bit complessivo - bit di informazione più bit di controllo - e k è il numero di bit di cui è formata l'informazione originale. Un codice ciclico con polinomio generatore di grado $(n - k)$ viene chiamato codice $(n - k)$ e $(n - k)$ rappresenta la sua caratteristica. Un codice di questo tipo è in grado di rilevare errori sui singoli bit e tutti gli errori multipli e adiacenti con lunghezza non superiore a $(n - k)$ -bit. Supponendo che un code-word sia formato da n -bit possiamo affermare che ogni code-word è rappresentato da un polinomio $V(X)$. Supposto di avere un code-word del tipo $v = v_0, \dots, v_{n-1}$, il polinomio che lo rappresenta è dato da

$$V(X) = v_0 + v_1X + \dots + v_{n-1}X^{n-1} \quad (5)$$

Quindi, ogni code-word è rappresentato da un polinomio di grado $n - 1$ o inferiore chiamato *code-polynomial*. Quindi, dato il polinomio dei dati $D(X)$, ad esempio per la parola di 4-bit 1101 il polinomio dei dati è $D(X) = 1 + X + X^3$, il code-polynomial viene composto moltiplicando il polinomio dei dati con il polinomio generatore (che abbiamo già introdotto), per cui $V(X) = D(X)G(X)$. Si osservi però che ogni addizione richiesta durante la moltiplicazione dei due polinomi deve essere eseguita in modulo 2. Ad esempio, supposto di avere un polinomio generatore pari a $G(X) = 1 + X + X^3$, il polinomio di codifica è quindi $V(X) = D(X)G(X) = 1 + X^2 + X^6$, o meglio $V(X) = 1 + (0)X + (1)X^2 + (0)X^3 + (0)X^4 + (0)X^5 + (1)X^6$. In questo modo il code-word è dato dai coefficienti del code-polynomial, e in questo caso abbiamo $v = 1010001$. Vediamo come valutare se il code-word ricevuto sia valido o meno. Si supponga di aver ricevuto il code-word $r = r_0, \dots, r_{n-1}$, il polinomio del codice è $R(X)$. Poiché il polinomio $R(X)$ è stato ottenuto moltiplicando il polinomio dei dati per il polinomio generatore possiamo scrivere che $R(X) = D(X)G(X) + S(X)$, dove $S(X)$, dovrebbe essere nullo se $R(X)$ è un multiplo del polinomio generatore, cioè se $R(X)$ è un code-word valido. Un metodo per verificare se $R(X)$ sia un code-word valido consiste nel dividere $R(X)$ per $G(X)$ e controllare che il resto della divisione sia zero. In questo caso $R(X)$ è un codice valido. Il polinomio $S(X)$ viene chiamato *syndrome polynomial*. Lo svantaggio primario dei codici ciclici è rappresentato dal fatto che non sono separabili. Comunque sono state presentate procedure per la creazione di codici ciclici separabili. La generazione di un codice (n, k) può essere eseguita mediante la seguente procedura:

1. Moltiplicare il polinomio dei dati per X^{n-k} . Operazione che può essere eseguita mediante un semplice shifting di $D(X)$.
2. Dividere il polinomio ottenuto nel precedente punto per il polinomio generatore $G(X)$ per ottenere il resto $R(X)$,
3. Il code-polynomial viene creato nel seguente modo $V(X) = R(X) + X^{n-k}D(X)$. L'addizione tra $R(X)$ e $X^{n-k}D(X)$ può essere eseguita semplicemente concatenando il resto a $X^{n-k}D(X)$.

La validità del procedimento appena descritto può essere provata nel modo seguente. Si supponga di avere un arbitrario polinomio dei dati $D(X)$ ed un polinomio generatore $G(X)$. Il polinomio del codice viene generato come $V(X) = R(X) + X^{n-k}D(X)$ dove $R(X)$ è il resto ottenuto dalla divisione di $X^{n-k}D(X)$ con $G(X)$, in altre parole possiamo scrivere che

$$\frac{X^{n-k}D(X)}{G(X)} = Q(X) + \frac{R(X)}{G(X)} \quad (6)$$

dove $Q(X)$ è il quoziente della divisione. Moltiplicando entrambi i membri per $G(X)$ otteniamo:

$$X^{n-k}D(X) = Q(X)G(X) + R(X) \quad (7)$$

oppure:

$$X^{n-k}D(X) - R(X) = Q(X)G(X) \quad (8)$$

Si ricordi però che ogni addizione e sottrazione deve essere eseguita in modulo 2 per cui addizione e sottrazione sono identiche, per questo motivo possiamo scrivere:

$$X^{n-k}D(X) + R(X) = Q(X)G(X) = V(X) \quad (9)$$

in questo modo $V(X) = R(X) + X^{n-k}D(X)$ è un polinomio di codice che è esattamente un multiplo del polinomio generatore $G(X)$.

4.6 Codici aritmetici

Le codifiche aritmetiche sono utili quando si rende necessario controllare le operazioni aritmetiche come addizione, moltiplicazione, divisione ecc. L'idea di fondo è la seguente: gli operandi vengono codificati prima di eseguire su di essi l'operazione desiderata. Il risultato viene quindi controllato per verificare se il code-word risultante dall'operazione aritmetica è valido. Se il code-word risultante non è valido significa che si è verificata una condizione di errore. Una codifica aritmetica deve essere invariante rispetto ad un insieme di operazioni. In particolare una codifica aritmetica ha la seguente proprietà: $A(b * c) = A(b) * A(c)$ dove b e c rappresenta gli operandi, $*$ rappresenta una delle operazioni per le quali la codifica è invariante e $A(x)$ rappresenta la codifica aritmetica per l'operando x . In altre parole possiamo dire che una codifica aritmetica ha la seguente proprietà: l'operazione aritmetica su operandi codificati produce lo stesso code-word rispetto alla codifica aritmetica del risultato della normale operazione. Per caratterizzare una codifica aritmetica è necessario specificare il metodo di codifica e l'insieme delle operazioni per le quali la codifica è invariante. La codifica aritmetica più semplice è chiamata AN . Questo tipo di codifica consiste nel moltiplicare ciascuna parola dei dati, N , per una costante, A . Questa codifica è invariante alle operazioni di addizione e di sottrazione, ma non alle operazioni di moltiplicazione e divisione. Quindi la verifica che il code-word sia valido viene eseguita dividendo il code-word per A e verificando che sia divisibile quindi per A . La scelta della costante A comporta il numero di extra-bit che verranno utilizzati e la capacità di rilevare gli errori. Quindi la scelta di A è un'operazione critica. A deve essere scelta in modo tale che non sia $A = 2^x$. Per capire le ragioni di questa limitazione dobbiamo ricordare che in base binaria la moltiplicazione per 2^x si traduce in una operazione di shift aritmetico verso sinistra del numero per cui il numero a_{n-1}, \dots, a_0 diventerebbe $a_{n-1}, \dots, a_0, 0, \dots, 0$ con esattamente x zero. La rappresentazione decimale quindi è data da:

$$a_{n-1}2^{a+n-1} + \dots + a_02^a + (0)2^{a-1} + \dots + (0)2^0 \quad (10)$$

Questo numero è divisibile per 2^x . Si può notare inoltre che in caso di un'alterazione di uno solo dei coefficienti, il numero rimane comunque divisibile per 2^x . Quindi se $A = 2^x$ la codifica non è in grado di rilevare gli errori su singolo bit. Una codifica AN corretta è la $3N$. La codifica $3N$ comporta, per una parola di n -bit, $(n+2)$ -bit per rappresentare il code-word. L'implementazione della codifica $3N$ è relativamente semplice: si utilizza un adder per eseguire la somma $N + 2N$ dove la quantità $2N$ è facilmente ottenibile mediante shift binario verso sinistra di una posizione. Un'altra codifica aritmetica consiste nel *residuo*. È un tipo di codifica separabile che consiste nell'aggiungere al numero da codificare il residuo. Supposto che il numero da codificare sia N , scelto un modulo m chiamato anche *check-base*, possiamo scrivere $N = Im + r$ dove I è il quoziente intero tra N e m e r è il resto. Ad esempio supposto di dover codificare 14 avendo scelto un modulo pari a 3 possiamo scrivere che $14 = 3 * 4 + r$ da cui $r = 2$. Il residuo quindi è

pari a 2. Il numero di extra-bit necessari con questo tipo di codifica dipende dal modulo scelto, infatti il residuo non è mai maggiore del modulo: $0 \leq r < m$. La verifica di correttezza su un risultato è data dal calcolo del residuo del dato da verificare e dal suo confronto con il residuo concatenato con il dato. La codifica con residuo è invariante rispetto all'addizione. Quindi supposto di avere due numeri D_1 e D_2 , rispettivamente con i residui r_1 e r_2 , la somma verrà creata come $S = D_1 + D_2$ con residuo $r_s = r_1 + r_2$. Durante il controllo quindi si esegue il calcolo del residuo di S e lo si controlla con r_s . Se i due residui differiscono significa che si è verificato un errore. I residui vengono addizionati con un adder modulo- m . Il vantaggio primario di questa codifica consiste nel fatto che è separabile. Se il valore del modulo per il calcolo del residuo viene scelto in un modo particolare la codifica prende il nome di *low-cost residue*. In particolare scegliendo $m = 2^b - 1$ con $b \geq 2$ il numero di extra-bit per la codifica low-cost è pari a b . Il vantaggio primario di questa tecnica consiste nel fatto che il procedimento di codifica viene semplificato. Si ricordi infatti che per eseguire una codifica con residue dobbiamo calcolare un resto e quindi dobbiamo eseguire una divisione. La codifica low-cost permette di sostituire l'operazione di divisione con un'operazione di addizione. I bit dell'informazione vengono prima suddivisi in gruppi di b -bit, ciascun gruppo quindi viene addizionato all'altro mediante un addizione modulo- $(2^b - 1)$. Il risultato di questa operazione rappresenta il residue dell'informazione. Una variante alla codifica separabile con residuo consiste nella cosiddetta *residuo inverso* la quale consiste nel calcolare il residuo, come nella precedente versione, ma anziché aggiungere all'informazione tale residuo viene aggiunta la quantità $Q = m - r$, cioè l'inverso del residuo. Questa variante consente una migliore capacità di rilevamento degli errori "ripetuto". I fallimenti ripetuti sono quei fallimenti che si verificano ripetutamente prima che i dati vengano controllati. Un esempio tipico di sistema nel quale si verifica questo tipo di fallimenti è rappresentato da un sistema che esegue una moltiplicazione mediante ripetute addizioni. Se tale sistema presenta un malfunzionamento di qualsiasi tipo, il suo uso ripetuto causa proprio i fallimenti ripetuti, in inglese *repeated-use faults*. Questo tipo di fallimenti è particolarmente difficile da rilevare in quanto effetti del fallimento potrebbero cancellare gli effetti del fallimento verificatosi nel precedente utilizzo rendendo così il problema non rilevabile.