


# Appunti di Progettazione Hardware 2

Matteo Gianello

16 luglio 2014

Quest'opera è stata rilasciata con licenza Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 4.0 Internazionale. Per leggere una copia della licenza visita il sito web <http://creativecommons.org/licenses/by-nc-sa/4.0/>. .

# Indice

<b>1</b>	<b>Test di circuiti digitali</b>	<b>2</b>
1.1	I modelli di guasto . . . . .	3
1.1.1	Single Stuck-at fault . . . . .	4
1.1.2	Transistor faults . . . . .	5
1.2	Fault simulation . . . . .	6
1.2.1	Simulazione seriale . . . . .	7
1.2.2	Simulazione parallela . . . . .	7
1.2.3	Ssimulazione deduttiva . . . . .	8
1.2.4	Simulazione concorrente . . . . .	8
1.2.5	Simulazione di vettori parallela . . . . .	8
1.2.6	Fault samplig . . . . .	9
1.3	Generazione dei test nei circuiti combinatorio . . . . .	9
1.3.1	Algoritmo esaustivo . . . . .	10
1.3.2	Generazione casuale . . . . .	10
1.3.3	Path Sensitization . . . . .	11
1.3.4	Algoritmo D . . . . .	13
1.3.5	Algoritmo PODEM . . . . .	16
1.4	Generazione dei test nei circuiti sequenziali . . . . .	16
1.4.1	Metodo Contest . . . . .	18
1.4.2	Algoritmo genetico . . . . .	18
1.4.3	Metodo spettrale . . . . .	18
<b>2</b>	<b>Modelling</b>	<b>20</b>
2.1	Modelli di computazione . . . . .	21
2.1.1	Composizione di macchine a stati . . . . .	23
2.1.2	Composizione in cascata . . . . .	25
2.1.3	Composizione gerarchica . . . . .	28
2.2	Attori . . . . .	28
2.3	Dataflow model . . . . .	30
2.4	Multitasking . . . . .	30
<b>3</b>	<b>Flusso di progetto nei sistemi eterogenei</b>	<b>31</b>

# 1 Test di circuiti digitali

Quando si vuole realizzare un nuovo prodotto hardware come uno smartphone o semplicemente anche un nuovo modulo wifi si parte sempre dalle richieste del cliente e da queste si passa alla stesura di un documento di *specifica*, alla *progettazione* del sistema ed infine alla *verifica* che il sistema progettato è conforme alla specifica. Dopo queste fasi iniziali però si deve tradurre il progetto in uno hardware che si possa costruire in modo semplice ed efficiente. Il passo finale prima di poter commercializzare il prodotto è quello del collaudo in quanto, in molti casi possono sorgere problemi di fabbricazione e, pur essendo il progetto corretto il sistema finale non funzionerà. Lo scopo di questo capitolo allora è quello di capire quali sono le tecniche migliori per far collaudare un nuovo sistema. Primo fra tutti è quello di pensare al collaudo durante la fase di progettazione e prevedere un modo per rendere facile i test dopo aver prodotto il sistema. Il modo più semplice per testare un circuito è quello di prelevare il componente terminato e inserire una serie di valori in ingresso e comparare i valori all'uscita per verificare che il risultato sia corretto come mostrato in Figura 1.

Questo meccanismo può essere applicato a qualsiasi componente del circuito ma questa tecnica presenta diversi problemi, prima tra tutte il costo delle apparecchiature necessarie per l'esecuzione dei test; ad esempio per un ATE a 1024 pin che lavora ad una frequenza che varia tra  $0.5$  e  $1\text{ GHz}$  il costo è dato dal costo base più un certo costo per ogni pin.

$$\$1.2M + 1024 \times \$3000 = \$4,27M$$

Per tale apparecchiatura si dovrà inoltre prevedere un costo dovuto all'ammortamento, alla manutenzione e alla sua operatività che per il nostro esempio possiamo quantificare come:

$$\begin{aligned} &= \text{Ammortamento} + \text{Manutenzione} + \text{Operatività} \\ &= \$0.854M + \$0.085M + \$0.5M \\ &= \$1.439M/\text{anno} \end{aligned}$$

Tale costo si riflette sul prodotto finale, infatti, supponendo che la macchina lavori 24/7 avremmo che il costo al secondo della macchina è pari a:

$$\begin{aligned} &= \$1.439M / (365 \times 24 \times 3600) \\ &= 4.5\text{cent}/\text{sec} \end{aligned}$$

Questo significa che ogni secondo di test costa  $4.5\text{ cent}$  per questo motivo è necessario che il test dia risultati chiari ma che venga svolto nel minor tempo possibile.

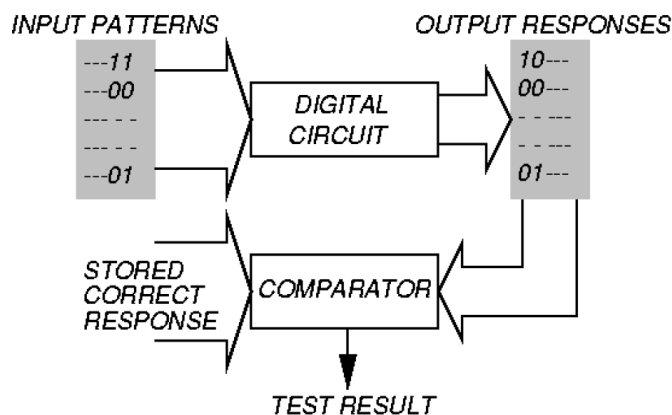


Figura 1: Collaudo di un sistema hardware

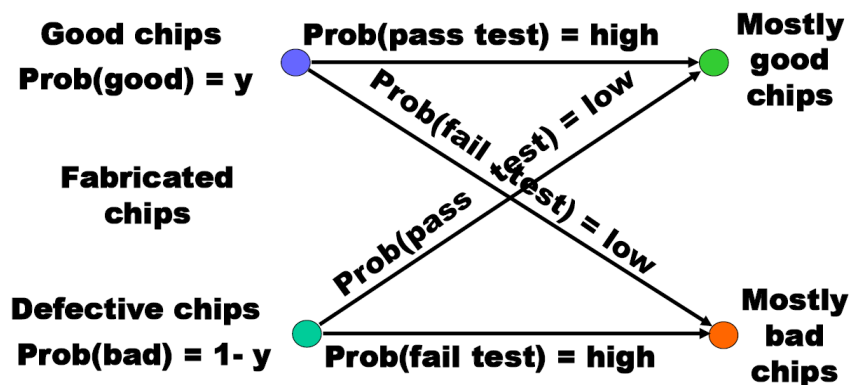


Figura 2: Probabilità di individuazione di componenti guasti

Un altro problema che si può presentare è la granularità con la quale si effettuano i test, infatti più il livello al quale si effettua il test è basso meno costoso sarà la riparazione, ad esempio se il livello al quale si scopre il guasto è a livello di transistor il costo della sostituzione del transistor è quantificabile a  $1$ , se il guasto viene individuato quando il transistor è montato su di una board allora il costo sale a  $10$  se tale board è montata nel sistema prima di scoprire il guasto allora il costo di quel guasto sale a  $100$  e così via; mano a mano che si sale di livello il costo di un guasto in termini di individuazione e riparazione diventa dieci volte più grande. Per questo motivo quando si effettuano i test le probabilità di individuare un componente difettoso devono essere le più alte possibili anche a costo di scartare qualche componente che non sia difettoso come mostrato in Figura 2.

### 1.1 I modelli di guasto

L'individuazione dei difetti di fabbricazione è diventata perciò una parte fondamentale della fase di realizzazione di un sistema hardware, e come tale si è evoluta nel tempo passando da considerare i *difetti di fabbricazione* come eventi sfortunati, a considerarli successivamente dei *guasti* fino ad arrivare ai *modelli di guasto* che si prefiggono lo scopo di creare un modello per ogni livello di astrazione per individuare i punti cruciali del circuito nel quale effettuare i test. I guasti possono essere di diverso tipo e dovuti a diversi fattori, essi possono essere:

- Difetti di processo:
  - Contatti mancanti
  - Capacità parassite
  - Ossidazione dei contatti
- Difetti materiali:
  - Difetti di carico (rottture o imperfezione nei cristalli)
  - Impurità sulle superfici
- Guasti dipendenti dal tempo:
  - Rotture elettriche
  - Elettromigrazione

Defect classes	Occurrence frequency (%)
<b>Shorts</b>	<b>51</b>
<b>Opens</b>	<b>1</b>
<b>Missing components</b>	<b>6</b>
<b>Wrong components</b>	<b>13</b>
<b>Reversed components</b>	<b>6</b>
<b>Bent leads</b>	<b>8</b>
<b>Analog specifications</b>	<b>5</b>
<b>Digital logic</b>	<b>5</b>
<b>Performance (timing)</b>	<b>5</b>

Figura 3: Percentuale dei guasti ad una PCB

- Guasti da imballaggio

In Figura 3 sono mostrati i principali guasti che si presentano su una PCB con le rispettive percentuali.

Esistono diversi modelli di guasto i più comuni ed utilizzati sono:

- Single stuck-at faults
- Transistor open and short faults
- Memory faults
- PLA faults
- Functional faults
- Delay faults
- Analog faults

### 1.1.1 Single Stuck-at fault

Il primo modello che analizziamo è il *Single stuck-at* in questo tipo di modello si assume che una linea del circuito rimanga bloccata ad un livello sia esso alto o basso. Il *single stuck-at* ha tre caratteristiche che lo contraddistinguono, la prima è che solamente una linea si guasta, la seconda è che la linea guasta si blocca permanentemente ad un valore 0 o 1, ed infine, la linea che si blocca può essere sia all'ingresso che all'uscita di una porta. Un esempio di circuito XOR analizzato con modello di guasto a *single stuck-at* è mostrato in Figura 4 dove sono indicati con un pallino verde i 12 punti di guasto mentre i possibili guasti sono 24 (due valori per ogni punto). Questo modello seppur semplice è supportato dall'evidenza empirica infatti è semplice da usare, può individuare anche altri tipi di guasti ed inoltre si presta ad essere modellizzato.

Questo modello può essere ampliato tramite la nozione di *equivalenza di guasto* e quella di *collasso dei guasti*. Si ha un'*equivalenza tra guasti* quando due guasti  $f_1$  e  $f_2$  se tutti i test che individuano il guasto  $f_1$  individuano anche il guasto  $f_2$ , se due guasti sono equivalenti le loro funzioni di guasto sono *identiche*. Il *collasso di guasti* si ha quando un circuito logico può essere diviso in un sottoinsieme equivalente dove tutti i guasti del sottoinsieme sono equivalenti, il collasso dei guasti avviene prelevando un guasto da ogni sottoinsieme. In Figura 5 sono mostrate le regole di equivalenza per le diverse porte logiche.

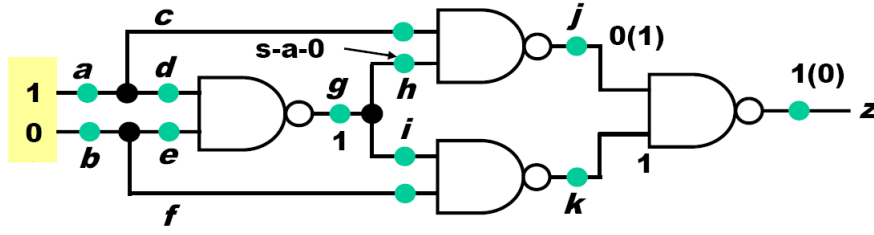


Figura 4: Esempio di modello di guasto *single stuck-at*

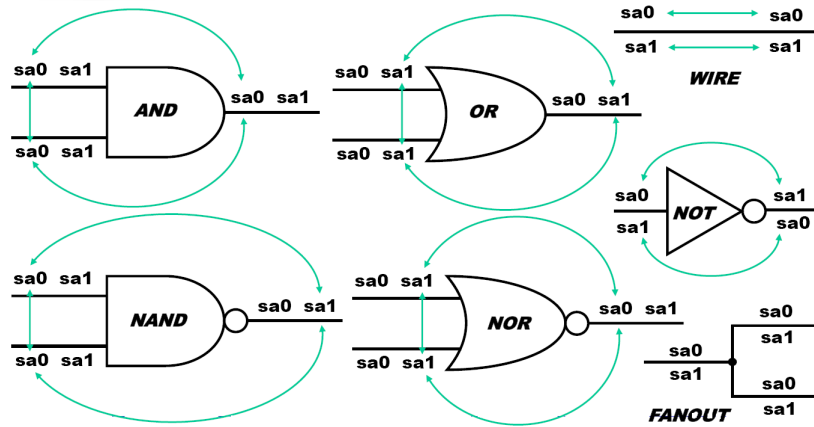


Figura 5: Equivalenze tra le diverse porte logiche

Un altro concetto che vogliamo introdurre è quello della *dominanza* tra guasti, se tutti i test che individuano  $f_1$  individuano anche un secondo guasto  $f_2$  allora si dice che  $f_2$  domina  $f_1$ . Gli ingressi primari e i bracci dei *fanout* di un circuito combinatorio sono chiamati *checkpoints*; un teorema afferma che se un insieme di test individua i guasti su tutti i checkpoint di un circuito combinatorio allora quello stesso insieme di test individua tutti i guasti del circuito. Il modello a *multiple stuck-at* prevede, rispetto al modello a singolo stuck-at, che più linee contemporaneamente possano essere bloccate in una determinata combinazione di guasto. Il numero totale di combinazioni di guasti in un circuito con  $k$  punti di guasto è di  $3^k - 1$ .

### 1.1.2 Transistor faults

Esistono diversi modelli di guasto che riguardano i transistor che fanno riferimento a dei comportamenti anomali di questi. I modelli che noi vedremo sono:

**Stuck-open:** nel quale un singolo transistor è modellizzato come un circuito aperto

**Stuck-short:** nel quale il transistor viene trattato come un cortocircuito

Nel caso dello *stuck-open* sono necessari due vettori per individuare il guasto in quanto il sistema non è in grado di cambiare lo stato precedente del circuito perciò per poterlo testare prima bisogna portare lo stato dell'uscita in uno stato noto e poi cercare di invertire questo stato; in caso di guasto questo cambio risulta impossibile. Per quanto riguarda il caso di *stuck-short* invece per identificare i guasti è necessario andare a misurare la *corrente di dispersione*  $I_{DDQ}$  del circuito in quanto in un circuito stabile nel quale sono esauriti i transistori non dovrebbe esserci un passaggio di corrente; tuttavia con le nuove tecnologie le correnti di quiescenza sono diventate sempre minori e quindi difficili da individuare.

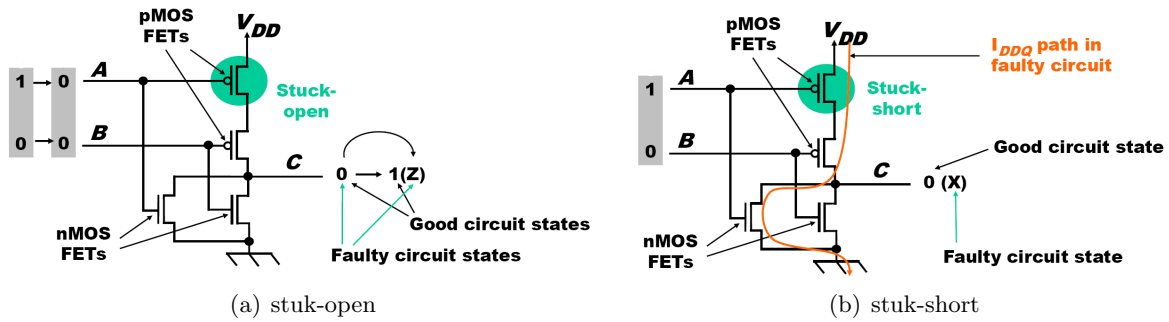


Figura 6: Esempio di stuck-open e stuck-short

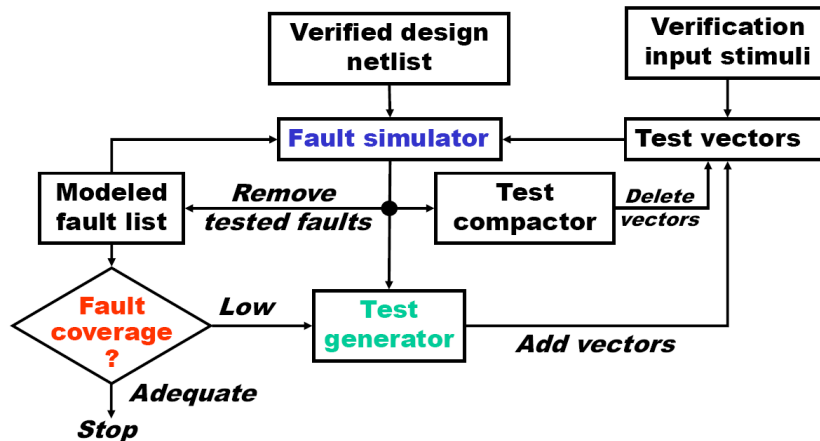


Figura 7: Diagramma che mostra il procedimento e l'utilità della *fault simulator*

## 1.2 Fault simulation

Dopo aver capito che cos'è un modello di guasto possiamo ora a capire come sfruttarlo all'interno di una simulazione. Per effettuare una simulazione si parte sempre da un circuito, nella maggior parte dei casi questo è espresso a livello logico, da una sequenza di vettori di test e da un modello di guasto; questo per determinare la copertura dei guasti individuati dall'insieme dei vettori di test, ma anche gli eventuali guasti non coperti per migliorare ed ottimizzare i collaudi post-produzione. In alcuni casi la simulazione è utile anche per identificare quelle aree di test che risultano difficili da testare in modo che i progettisti possano valutare la possibilità di aggiungere della logica per aumentarne la testabilità; questo tipo di progettazione è anche chiamata *Design for Test (DFT)*. In Figura 7 vediamo quali sono le diverse utilità della simulazione dei guasti sia in termini di copertura dei guasti (*fault coverage*) sia in termini di ottimizzazione dei test (*test compactor*). Un'altra funzionalità per cui la simulazione dei guasti è utilizzata è quella di *timing* ovvero capire i tempi di transitorio che il circuito ha in presenza di guasti, in quanto potrebbe essere che il circuito ottenga un valore corretto ma che il tempo del transitorio sia maggiore di quello di clock e quindi non riuscirei ad analizzarlo. Questo tipo di simulazione è stata introdotta solamente negli ultimi anni quando le frequenze sono aumentate considerevolmente. Gli algoritmi per la simulazione dei guasti sono diversi con diversi gradi di difficoltà, queste simulazioni sono:

- Seriale
- Paralela

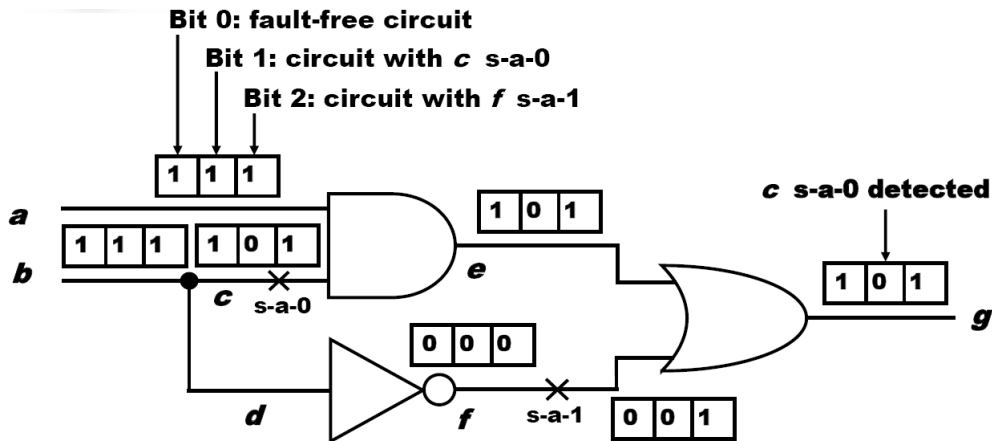


Figura 8: Esempio di simulazione parallela con processore a 3 bit

- Deduttiva
- Concorrente
- e molti altri ancora

### 1.2.1 Simulazione seriale

Il primo tipo di simulazione che andiamo ad analizzare è quella che sfrutta un algoritmo di tipo *seriale* in quanto è quello più semplice.

Si parte da una simulazione del circuito senza guasti in modo da registrare le uscite del circuito. A questo punto si effettua un'*iniezione del guasto* ovvero si modifica il modello per far sì che sia presente il guasto. A questo punto si riapplica il vettore di test al circuito e se la risposta differisce dal caso senza guasto sono in grado di affermare che il vettore di test analizzato identifica il guasto appena testato. Questa operazione viene ripetuta per tutti i guasti nella lista dei guasti, alla fine di tutta la simulazione posso determinare quali guasti sono coperti da quel vettore di test.

Per simulare un guasto durante una simulazione abbiamo detto che dobbiamo fare una *iniezione di guasto* questa tecnica però può essere implementata in due modi, il primo è passare di volta in volta al simulatore un circuito diverso dove è stato introdotto il guasto oppure passare al simulatore un circuito simile a quello da testare ma dove nei punti di guasto sono stati inseriti dei *multiplexer* e selezionare i punti di guasto tramite i valori di controllo del multiplexer.

Lo svantaggio principale di questa tecnica è che richiede un notevole numero di simulazioni e le computazioni risultano ripetitive. Per risolvere tale problema si può sfruttare un certo grado di parallelizzazione.

### 1.2.2 Simulazione parallela

Nella simulazione parallela si utilizza una caratteristica dei processori, ovvero la possibilità di eseguire un AND o un OR di un'intera parola (32 o 64 bit) in un unico ciclo di clock. L'idea è quella di compattare più di una simulazione durante un'unica computazione, sfruttando tale tecnica si possono testare  $w - 1$  (è necessario simulare anche il circuito senza guasti) guasti in un'unica simulazione. Un esempio di questa tecnica è mostrato in Figura 8 dove si fa l'esempio di un processore a 3 bit e si testano due guasti con un'unica simulazione.



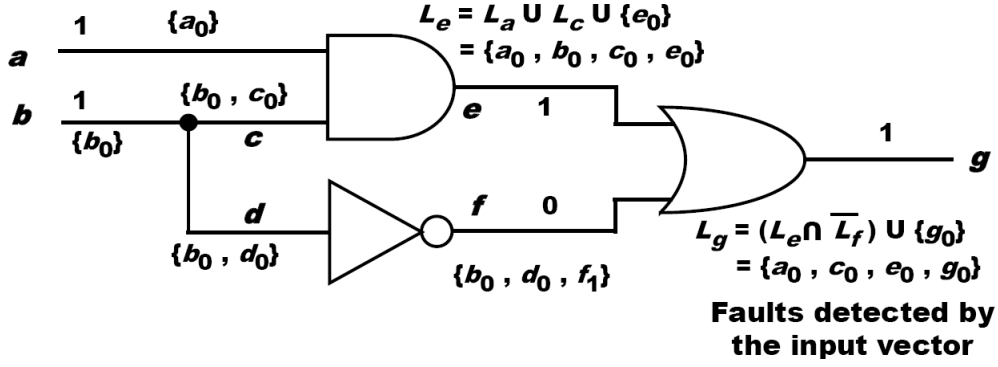


Figura 9: Esempio di simulazione deduttiva

### 1.2.3 Simulazione deduttiva

Nel caso di simulazione parallela si sfrutta la possibilità di parallelizzare i calcoli, tuttavia non si agisce sul fatto di riutilizzare le precedenti computazioni per quelle parti di circuito che non vengono toccate da guasti. La simulazione deduttiva, invece, sfrutta proprio questa caratteristica. In questo caso si analizza il circuito con la presenza di tutti i guasti ma considerando un guasto alla volta. Tramite questa tecnica si riesce a calcolare l'insieme dei guasti individuabili tramite un determinato vettore di test utilizzando un'unica simulazione.

Il principio è quello di individuare tutti i possibili guasti di una determinata linea  $k$  e inserirli in una lista  $L_k$  e a questo punto applicare delle regole insiemistiche delle varie porte. La lista dei guasti che si ottengono all'uscita è l'insieme dei guasti individuabili dal vettore di test utilizzato. Lo svantaggio di questa tecnica è che si applica solamente a circuiti logici booleani e non a quelli con *flip-flop* o altri componenti. In Figura 9 vediamo un esempio di questa tecnica, all'uscita della porta AND abbiamo che l'insieme dei guasti rilevabili sono :

$$L_e = L_a \cup L_c \cup \{e_0\} = \{a_0, b_0, c_0, e_0\}$$

Questo perchè entrambe le linee di ingresso mostrano valori non dominanti nel caso invece della porta OR abbiamo che all'uscita i guasti individuati sono quelli dell'ingresso  $e$  meno quelli dell'ingresso  $f$  in quanto sull'ingresso  $f$  è presente un valore non dominante.

### 1.2.4 Simulazione concorrente

La simulazione concorrente è molto simile a quella deduttiva, tuttavia in questo caso si parte da una simulazione priva di guasti e via via si iniettano i diversi guasti ricalcolando solamente le porte che subiscono una variazione dopo l'iniezione. Questo sistema è molto più rapido dei precedenti, tuttavia il fatto di memorizzare lo stato delle porte che cambiano comporta un grande consumo di memoria.

In Figura 10 vediamo un esempio di tale tecnica, si nota subito come l'ultima porta or debba tener conto di tutti i guast.

### 1.2.5 Simulazione di vettori parallela

Questo tipo di simulazione è molto simile a quella parallela ma in questo caso non si testano diversi guasti con lo stesso vettore bensì si testa un guasto diversi vettori di test. Questa tecnica è utile soltanto se il circuito è privo di memoria. In caso contrario tale tecnica è inapplicabile.

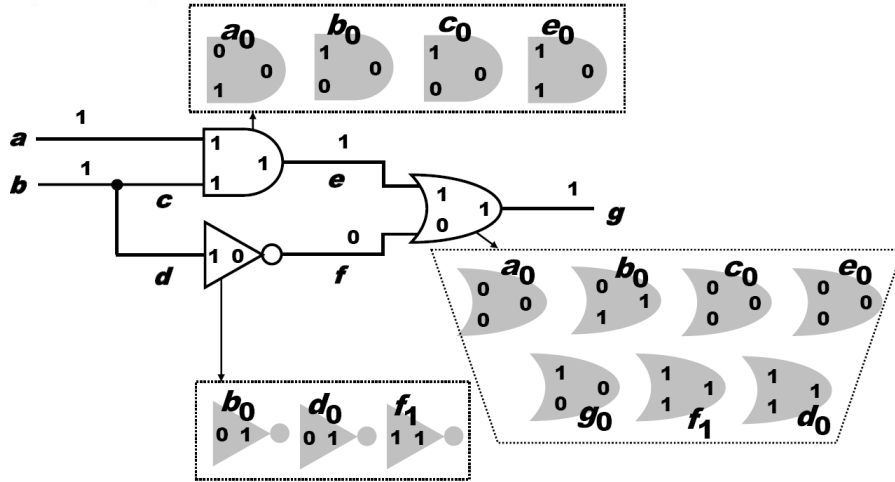


Figura 10: Esempio di simulazione concorrente

### 1.2.6 Fault samplig

In molti casi è impossibile testare tutti i guasti, tuttavia è possibile ottenere una buona copertura utilizzando il *fault samplig*. Si tratta di scegliere un sottoinsieme casuale dei guasti ed effettuare la simulazione su questi guasti per calcolarne la copertura. La misura di copertura dei guasti su questo sottoinsieme ci dà una misura sulla copertura totale dei guasti. Questo meccanismo permette di risparmiare molto tempo nelle simulazioni tuttavia può portare a non testare alcuni guasti.

## 1.3 Generazione dei test nei circuiti combinatorio

In questo paragrafo analizzeremo come creare i vettori di test che serviranno poi sia per la simulazione che per il collaudo del sistema. In molti casi questi vettori sono generalmente gli stessi che il progettista usa per verificare la corretta funzionalità del sistema rispetto alle specifiche. Tuttavia in molti casi questi vettori non sono sufficienti e a volte necessari per valutare tutti i guasti. Prendiamo ad esempio il caso in Figura 11, in questo caso tenendo conto solo della funzionalità e volendo testare completamente il circuito dovremmo utilizzare  $2^{64}$  valori per il primo ingresso moltiplicati per  $2^{64}$  valori per il secondo ingresso per l'ingresso di riporto per un totale di  $2^{129}$  vettori di test. Nel caso in cui prendiamo in considerazione la struttura del circuito abbiamo che gli stuck-at per la parte di somma di un bit sono 10 mentre per la parte di riporto (non presente in Figura) comprende 17 stuck-at questo significa per ogni singolo bit i guasti da testare sono 27 che moltiplicati per i 64 bit del sommatore fanno un totale di 1728 guasti; considerando il caso peggiore in cui sia necessario un vettore di test per ogni guasto abbiamo che sono necessari  $1728 \ll 2^{129}$  vettori.

Come abbiamo visto conoscere la struttura del circuito ci aiuta a ridurre i vettori di test, un altro fattore che aiuta nel ridurre la complessità delle simulazioni è l'utilizzo di un algoritmo per la generazione dei test che sia *completo* ovvero per ogni guasto l'algoritmo è in grado di stabilire se questo è testabile o meno.

Un guasto *non testabile* è un guasto che, pur essendo presente nel circuito, non influenza il comportamento di tale circuito ovvero non ne compromette la funzionalità, questo tipo di guasto è anche detto *ridondante* ma solo nei circuiti combinatorio e indica la presenza di hardware non necessario.

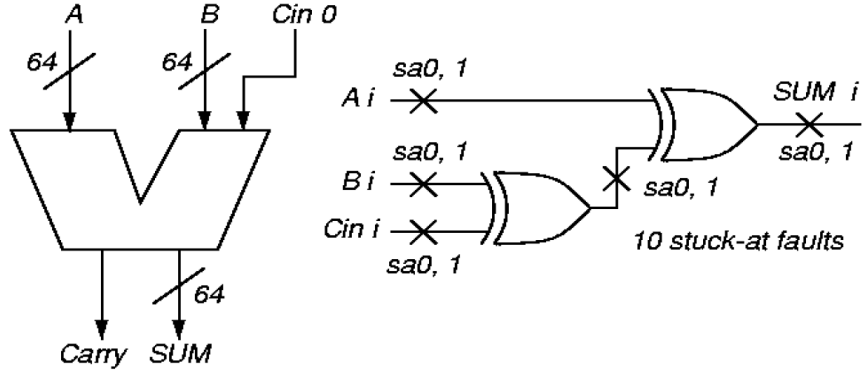


Figura 11: Schema funzionale e strutturale di un full-adder con riporto

Symbol	Meaning	Good Machine	Failing Machine	
$D$	1/0	1	0	Algebra di Roth
$\overline{D}$	0/1	0	1	
0	0/0	0	0	
1	1/1	1	1	
$X$	$X/X$	$X$	$X$	
$G0$	0/ $X$	0	$X$	Algebra di Muth
$G1$	1/ $X$	1	$X$	
$F0$	$X/0$	$X$	0	
$F1$	$X/1$	$X$	1	

Tabella 1: Notazioni delle algebre di Roth e di Muth

Per lo studio della generazione di test dobbiamo prima introdurre alcuni strumenti tra questi le algebre di *Roth* e di *Muth* espresse in Tabella 1. Queste algebre permettono di simulare due macchine simultaneamente, il primo valore indica una macchina che funziona correttamente il secondo valore una macchina in errore, inoltre queste algebre sono di facile rappresentazione e risoluzione. Per trovare i vettori di test dobbiamo però fare delle premesse, il valore nel punto di guasto deve essere differente rispetto a quello dello stuck-at (deve essere presente una  $D$  o una  $\overline{D}$ ), il guasto deve essere propagato all'uscita (una  $D$  o una  $\overline{D}$  devono essere presenti sull'uscita). Analizziamo ora alcuni dei principali algoritmi di generazione dei test.

### 1.3.1 Algoritmo esaustivo

Nel caso di un circuito con  $n$  input si generano tutti i  $2^n$  possibili vettori di test. Questo algoritmo è applicabile solo per un numero  $n \leq 15$  di input altrimenti il numero di vettori risulta troppo grande. Tuttavia questo meccanismo può essere utilizzato su alcuni circuiti per effettuare dei test *built-in* su una parte del circuito.

### 1.3.2 Generazione casuale

In questo caso i vettori di test sono generati casualmente. Questo meccanismo è utile soprattutto nel caso di circuiti aritmetici e fornisce dei risultati buoni con una copertura dei guasti che si aggira tra il 60% e l'80%. Questa tecnica è utilizzata anche in combinazione con altre tecniche per aumentare il grado di copertura.

### 1.3.3 Path Sensitization

La *Path sensitization* è un algoritmo per l'individuazione dei vettori di test molto semplice e di facile applicazione. Esso si compone di tre passi:

1. **Fault Sensitization:** durante questo passo si innesta il guasto e a questo punto tramite l'algebra di Roth si inserisce una  $D$  o una  $\overline{D}$  nel punto di guasto (Figura 12(a))
2. **Fault Propagation:** durante questo passo si propaga il guasto verso l'uscita scegliendo uno o più percorsi come mostrato in Figura 12(b) e Figura 12(c)
3. **Line Justification:** a questo punto si impongono sulle linee restanti dei valori utili per completare il circuito. Nel caso in cui sia possibile (Figura 12(d)) ottengo il vettore di test, nel caso non sia possibile (Figura 12(b) e 12(c)) scarto il circuito

Analizziamo più in dettaglio l'esempio, in Figura 12(a) vediamo come venga iniettato un guasto con stuck-at 0 ciò significa che l'ingresso  $B$  deve essere inizializzato a 1 altrimenti sarebbe impossibile individuare il guasto, inoltre secondo l'algebra di Roth un passio da  $1 \leftarrow 0$  va indicato da una  $D$ .

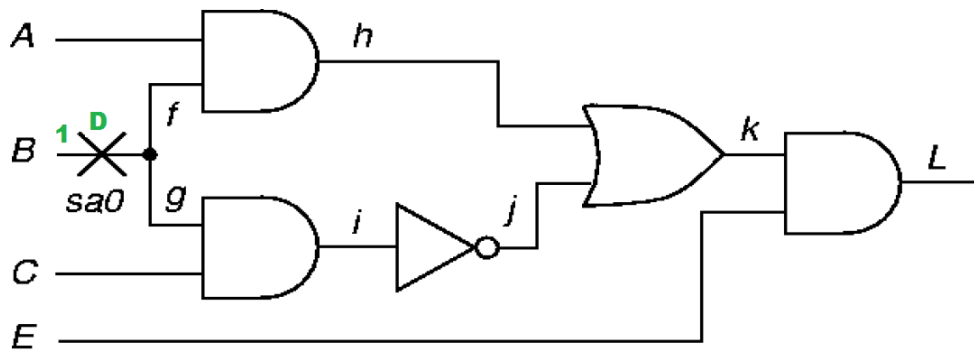
In Figura 12(b) il guasto viene propagato lungo il percorso  $f-h-k-L$  tale propagazione è indicata in blu, dopo aver propagato il guasto fino all'uscita abbiamo cercato di andare a ritroso per riempire le linee rimaste vuote (numeri in rosso). In questo caso, per la porta AND con ingressi  $k$  ed  $E$  ad  $E$  abbiamo assegnato il valore 1, alla porta AND con ingressi  $A$  e  $B$  all'ingresso libero abbiamo assegnato il valore 1, infine, alla porta OR il cui ingresso  $j$  era ancora libero abbiamo assegnato il valore 0 e lo abbiamo propagato all'indietro fino ad avere in posizione  $i$  il valore 1, tuttavia, questo valore è incompatibile con qualsiasi valore che avremmo potuto assegnare all'ingresso  $C$ ; questo porta a scartare il percorso.

In Figura 12(c) il guasto viene propagato parallelamente sia lungo il percorso  $f-h-k-L$  sia lungo il percorso  $g-i-j-k-L$ , in questo caso abbiamo che durante la propagazione del guasto il guasto scompare (valore 1 in posizione  $k$ ) tale fenomeno è chiamato *scomparsa della D-frontiera*.

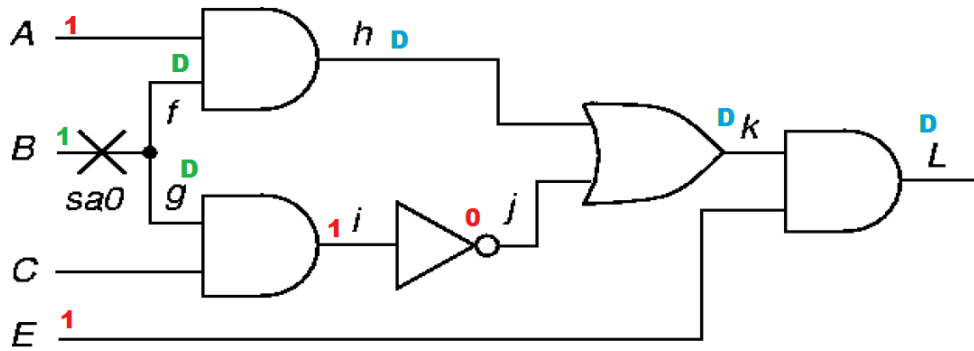
Infine, in Figura 12(d) vediamo la propagazione del guasto lungo il percorso  $g-i-j-k-L$  e vediamo come sia possibile sia portare il guasto fino all'uscita sia effettuare la *Line Justification* ottenendo così il vettore di test per il guasto analizzato. Tale vettore corrisponde al vettore 0-1-1-1 rispettivamente per gli ingressi  $A-B-C-E$ .

Abbiamo visto come questo meccanismo sia molto semplice tuttavia non è molto efficiente, in caso di circuito con *no\_pi* numero di ingressi primari e *no\_ff* numero di flip-flop del circuito e volendo simulare  $n$  porte logiche abbiamo che la complessità è pari a :

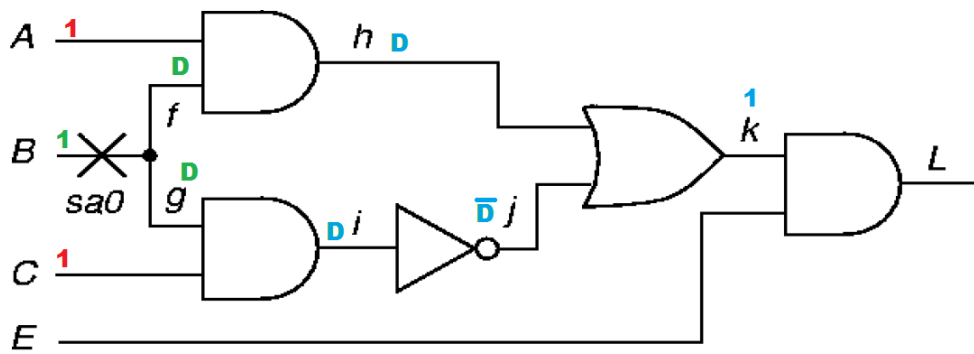
$$O(n \times 2^{no\_pi} \text{ times } 4^{no\_ff})$$



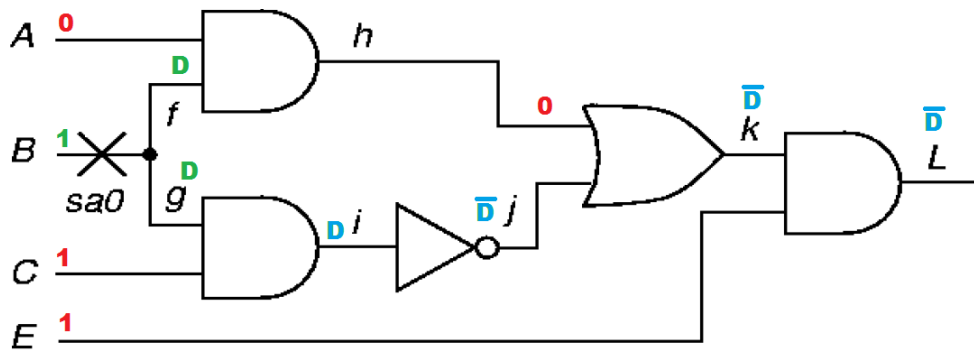
(a) Fault Sensitization



(b) Fault propagation sul percorso f-h-k-L



(c) Fault propagation parallela su f-h-k-L e g-i-j-k-L



(d) Fault propagation su g-i-j-k-L

Figura 12: Esempio di *Path Sensitization*

$a \backslash b$	0	1	X	D	$\bar{D}$
0	0	0	0	0	0
1	0	1	X	D	$\bar{D}$
X	0	X	X	X	X
D	0	D	X	D	0
$\bar{D}$	0	$\bar{D}$	X	0	$\bar{D}$

Figura 13: Forward implication table per una porta AND

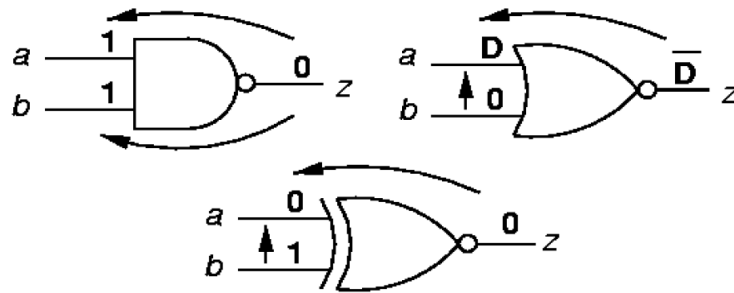


Figura 14: Esempi di backward implication

#### 1.3.4 Algoritmo D

Prima di iniziare a parlare di questo algoritmo dobbiamo introdurre alcuni concetti. Il primo concetto che introduciamo è quello del *forward implication* la quale afferma che dati degli ingressi di porte logiche i quali sono etichettati in modo significativo è possibile determinare univocamente le uscite. Ad esempio per una porta AND la tabella della forward implication è mostrata in Figura 13.

Un'altra definizione è quella della *backward implication* molto simile alla forward solo che in questo caso i valori noti sono quelli dell'output e a volte di uno degli ingressi e si riesce a risalire alle uscite, un esempio è mostrato in Figura 14.

Durante la fase di decisione sui valori da assegnare agli ingressi può capitare che in alcuni punti possano essere possibili più scelte come ad esempio nel caso di una porta AND la cui uscita vale 0. Per tenere traccia di queste possibilità e delle decisioni prese si utilizza un *implication stack* ovvero una tabella nella quale si tengono conto dei vari ingressi, del valore ad esso assegnati e se quel valore è l'unico possibile. Da questa tabella è possibile creare un *albero delle decisioni* per tenere traccia di quali percorsi sono esplorabili, quali sono stati esplorati e quali no. Un esempio di albero delle decisioni è mostrato in Figura 15. Questo tipo di algoritmo sfrutta il meccanismo del *Branch and Bound* per trovare una soluzione, ad ogni livello dell'albero l'algoritmo seleziona uno dei possibili input escludendo al tempo stesso porzioni dell'albero per restringere le decisioni possibili in quanto un'esplorazione completa dell'albero è impraticabile.

L'algoritmo *D* è il primo algoritmo completo per la creazione di vettori di test, esso sfrutta oltre alle precedenti nozioni anche quella del *D-Cube* e del *D-Calculus*. Il *D-Cube* non è altro che il collassamento della tavola di verità del circuito utilizzando i valori dell'algebra di Roth.

L'algoritmo si compone di cinque passi

- Si numerano tutte le linee del circuito in modo incrementale dagli ingressi alle uscite.

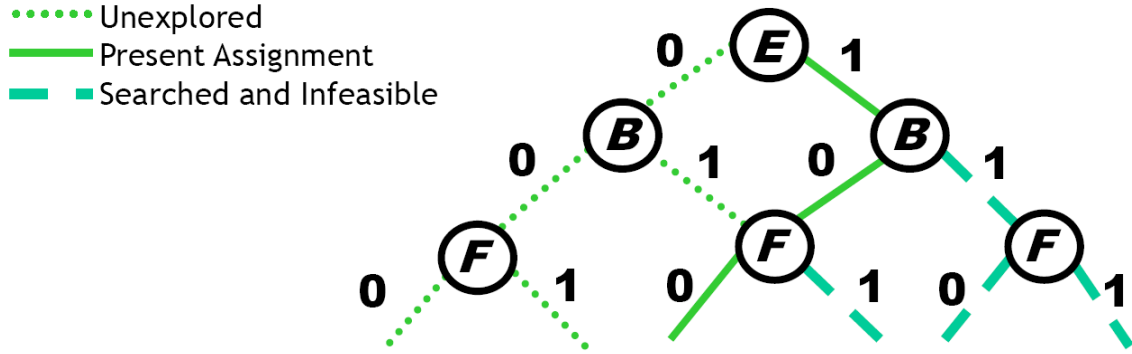


Figura 15: Esempio di albero delle decisioni

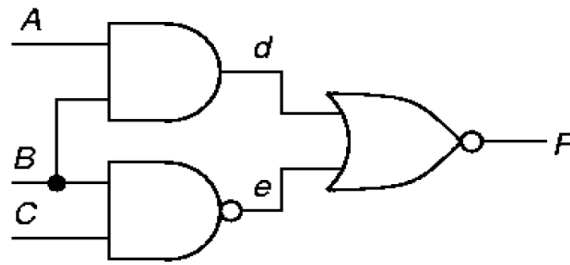


Figura 16: Circuito di esempio per l'algoritmo D

- Si seleziona una delle primitive 0 che indicano il guasto dal *D-Cube*
- Si effettua il D-drive ovvero la propagazione del guasto
- Si verifica la consistenza
- Si ripete l'operazione del D-drive

Un esempio è quello in Figura 16 mentre la Tabella 2 è la tabella D-Cube di copertura. Volendo analizzare uno *stuck-at 0* nel punto *d* i passi sono tre e sono mostrati in Tabella 3.

Uno *stuck-at 0* in *d* significa che in quel punto abbiamo una *D* utilizzando la backward implication otteniamo che sugli ingressi *A* e *B* abbiamo due 1. Sfruttando la seconda parte della Tabella 2 abbiamo che per un valore *D* sulla linea *d* abbiamo due possibilità (ultima e terzultima riga) prendiamo la prima delle due possibilità e questo ci dà il valore della linea *e* ovvero 0. A questo punto per avere 0 sulla linea *e* ed avendo già l'ingresso *B* fissato al valore 1 l'unica soluzione possibile è che all'ingresso di *C* abbiamo il valore 1. Per un ulteriore esempio più esaustivo si rimanda alle slide del corso e al video dell'insegnante.

A	B	C	d	e	F
1	1		1		
0			0		
	0		0		
	1	1		0	
	0			1	
		0		1	
				1	0
			1		0
			0	0	1
$D$	1		$D$		
1	$D$		$D$		
$D$	$D$		$D$		
	$D$	1		$\overline{D}$	
	1	$D$		$\overline{D}$	
	$D$	$D$		$\overline{D}$	
			$D$	0	$\overline{D}$
			0	$D$	$\overline{D}$
			$D$	$D$	$\overline{D}$

Tabella 2: Esempio di D-Cube di propagazione

Step	A	B	C	d	e	F	Cube type
1	1	1		$D$			Primitive D-cube of Failure
2				$D$	0	$\overline{D}$	Propagazione D-Cube
3		1	1		0		Backward implication

Tabella 3: Passi per la generazione del test



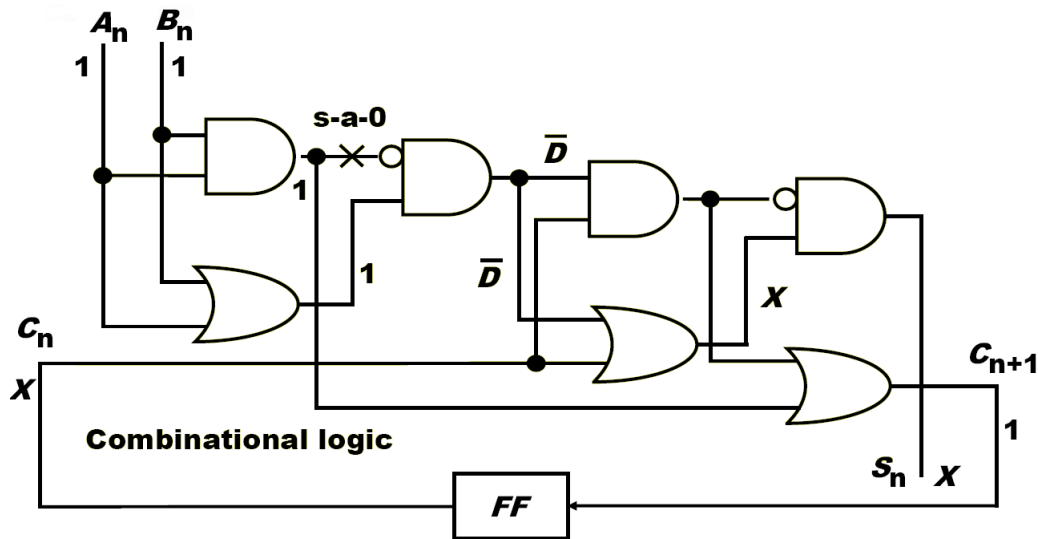


Figura 17: Esempio di circuito sequenziale

### 1.3.5 Algoritmo PODEM

L'algoritmo PODEM introdotto nel 1981 introdusse il concetto che l'albero delle decisioni non doveva essere espanso a tutte le linee del circuito ma solamente agli ingressi primari, inoltre esso non andava ad analizzare tramite backward e forward implication se il vettore esisteva ma semplicemente controlla di volta in volta l'esistenza della *D-Frontiera* ed infine utilizzava il *backtracing*.

L'algoritmo di PODEM si svolge in 6 passi:

1. Si assegna un valore binario ad uno degli ingressi ancora libero.
2. Si determinano le implicazioni su tutti gli altri ingressi
3. Il vettore di test è stato generato? Se sì *fatto*.
4. Se è possibile assegnare valori ad altri ingressi tornare al punto 1
5. Esistono combinazioni di valori per gli ingressi non ancora testati? Se no il guasto è *non testabile*.
6. Cercare una combinazione di valori per gli ingressi tramite obiettivi e backtrace.

Per un esempio completo si rimanda alle slide del corso.

## 1.4 Generazione dei test nei circuiti sequenziali

Passiamo ora alla generazione dei test per circuiti sequenziali. Nei circuiti sequenziali sono presenti degli elementi di memoria il cui valore non è noto e perciò è necessario introdurre più vettori di test in sequenza che permettano di portare questi elementi in uno stato conosciuto e successivamente testare i guasti. Un esempio di questa tecnica è mostrato in Figura 17 dove vediamo che dopo aver inserito agli ingressi il vettore *1-1* non riusciamo ad individuare il valore dell'uscita ma possiamo inizializzare il flip-flop ad un valore noto. A questo punto è possibile applicare un secondo vettore di test per testare il guasto. Per effettuare il test dei circuiti sequen-

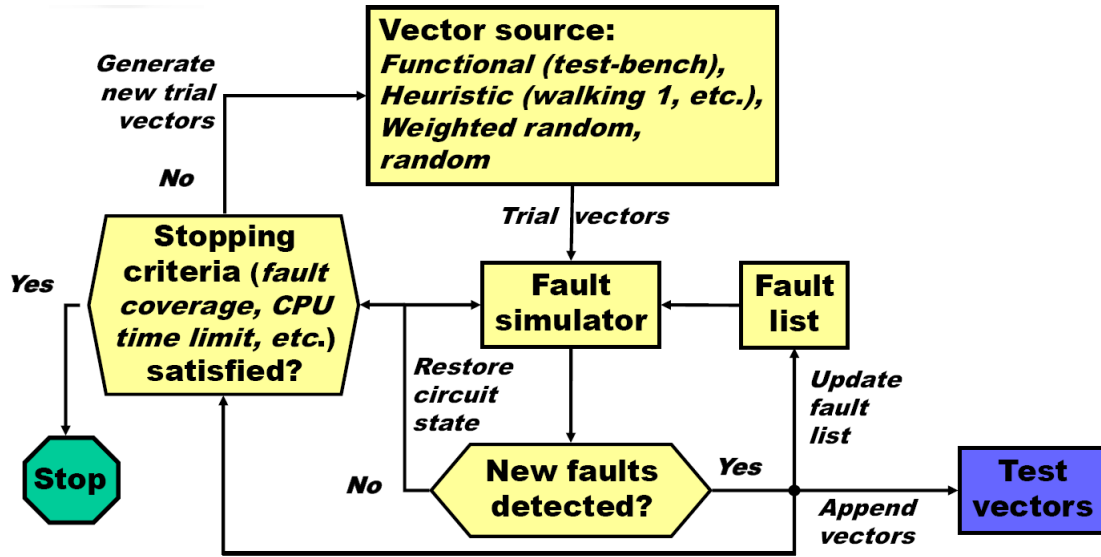


Figura 18: Generazione dei vettori di test tramite simulazione

ziali come abbiamo detto è necessario applicare più vettori, ma per farlo dobbiamo applicare la *time-frame expansion* che in pratica non fa altro che replicare *n-volte* il circuito da testare (dove *n* il numero di vettori di test) inserisce il guasto in ogni circuito ed infine genera i vettori di test utilizzando sia l'algebra di Roth che quella di Muth. Questo meccanismo permette di esplicitare il tempo tuttavia non si riesce a sapere a priori il numero di repliche necessarie.

Il meccanismo di risoluzione è molto simile ai precedenti:

- Si seleziona un'uscita sulla quale esporre il guasto.
- Si posiziona il valore logico, 1/0 o 0/1 in base al tipo di guasto e al numero di inversioni.
- Si giustificano le linee partendo dalle uscite e andando a ritroso verso gli ingressi.
- Se non è possibile effettuare la giustificazione si seleziona una nuova uscita tramite la derivabilità.
- Se tale tecnica fallisce per tutte le uscite raggiungibili allora il guasto è non testabile.
- Se i valori 1/0 o 0/1 non sono giustificabili ma i valori 1/X o 0/X lo sono allora il guasto diventa *potentially detectable*.

Possiamo fare una distinzione tra i circuiti sequenziali, possiamo avere circuiti *cycle-free* ovvero senza cicli nei quali non vi è alcun feedback tra i diversi flip-flop, in questo caso la generazione dei vettori è molto più semplice e richiede al massimo un'espansione pari a  $1 + d_{seq}$  dove  $d_{seq}$  è il grado di dipendenza dei ff. Nel caso di circuiti con *cicli* le cose si complicano ed è necessaria un'espansione di  $9^{N_{ff}}$  dove  $N_{ff}$  è il numero di flip flop.

Analizziamo ora i diversi algoritmi per la generazione dei test nei circuiti sequenziali, questi algoritmi non sono algoritmi esatti come quelli utilizzati per i circuiti combinatori ma sfruttano la simulazione per trovare dei risultati. L'utilizzo della simulazione è dovuto al fatto che le sequenze da inizializzare sono molto lunghe ed è impossibile utilizzare un albero delle decisioni, inoltre i modelli dei circuiti sono limitati e l'espansione porta a raggiungere tali limiti molto in fretta; infine problemi di complessità e di corsa alle risorse rende il metodo *time-frame* inutilizzabile.

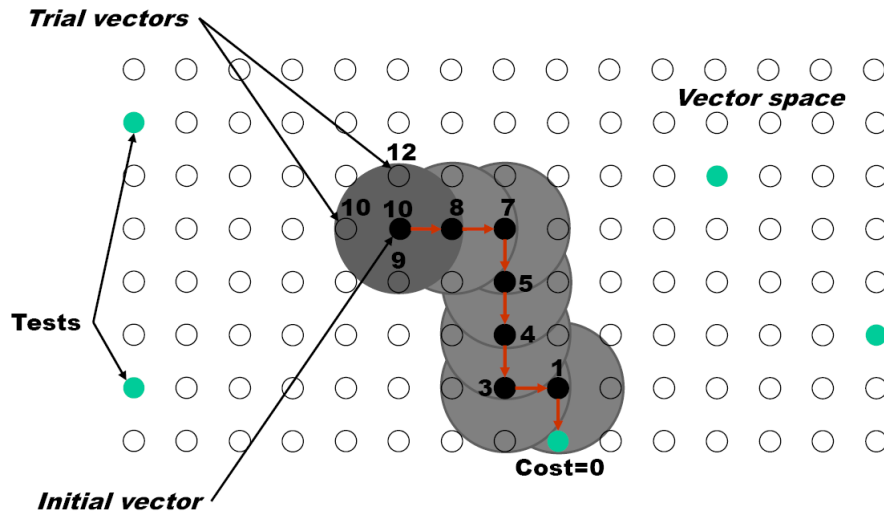


Figura 19: Esempio di ricerca tramite funzione obiettivo

I metodi basati sulla simulazione invece sfruttano le già collaudate tecnologie di simulazione dei guasti inoltre i modelli sono già stati creati e verificati, le metodologie che si possono utilizzare sono diverse. Un diagramma che riassume tale metodologia è mostrato in Figura 18.

#### 1.4.1 Metodo Contest

In questo tipo di simulazione vengono testati più vettori contemporaneamente e vengono via via migliorati tramite l'utilizzo di una *funzione di costo*. Tale metodo si suddivide in tre fasi:

- Una fase *iniziale* nella quale non si tiene conto di nessun guasto e la funzione di costo viene calcolata con il circuito allo stato originale
- Fase *concorrente* vengono introdotti tutti i guasti, la funzione di costo viene calcolata in modo concorrente su più simulazioni.
- Fase *singolo guasto*, in questa fase si analizza un solo guasto alla volta utilizzando i valori della simulazione.

La funzione di costo definisce l'obiettivo che si vuole perseguire come l'inizializzazione dei vettori o la testabilità di un guasto, la funzione di costo si annulla quando i vettori trovati rispettano l'obiettivo. Un esempio molto banale del meccanismo è mostrato in Figura 19.

#### 1.4.2 Algoritmo genetico

L'algoritmo genetico afferma che una *popolazione* migliora con ogni *generazione*. Nel caso di generazione di vettori di test la popolazione è per l'appunto un insieme di vettori, il fattore di miglioramento è una qualità misurabile come una funzione di costo. Al funzione di rigenerazione invece viene trovata in modo euristico considerando i vettori più *promettenti* e trasformandoli.

#### 1.4.3 Metodo spettrale

Il *metodo spettrale* si può considerare un'evoluzione di quello genetico. Si parte da un insieme casuale di vettori di test si effettua un compattamento eliminando quelli non necessari o quelli

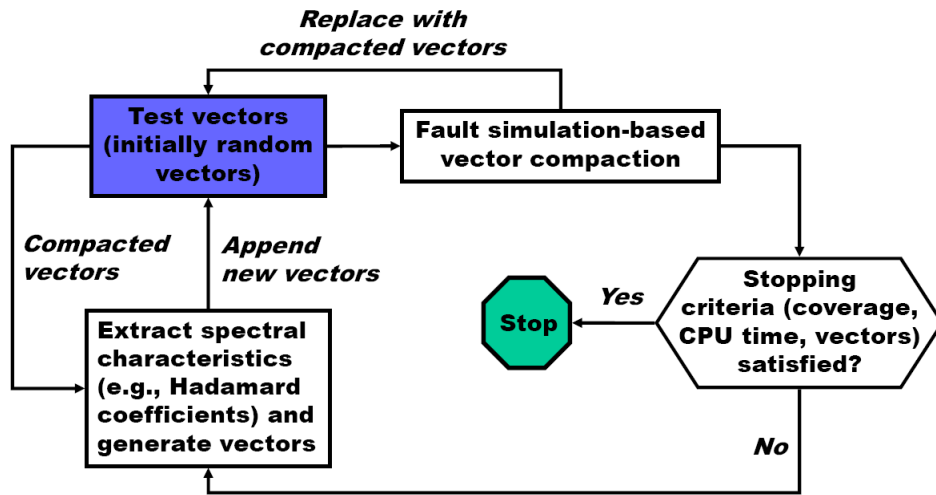


Figura 20: Schema di riepilogazione del meccanismo spettrale

ridondanti, dopo di che si simulano i diversi vettori. Lo schema di tale meccanismo è mostrato in Figura 20.

## 2 Modelling

Al giorno d'oggi esistono una serie di computer il cui scopo principale non è quello di gestire informazioni, bensì quello di interagire con i sistemi fisici, questo tipo di computer è chiamato *sistema embedded*, alcuni esempi di sistemi embedded sono:

- controlli automotive
- avionica
- dispositivi medici
- controlli industriali
- dispositivi di gestione e conservazione dell'energia

Un sistema embedded è un sistema di calcolo ma non principalmente un computer integrato con un insieme di processi fisici tramite sensori, attuatori ecc. il quale deve essere *reattivo* ovvero deve rispettare dei vincoli temporali, è *eterogeneo* in quanto formato da componenti hardware, software, componenti di rete; ed infine è *distribuito* e *concorrente*. Un esempio di questo tipo di sistema lo troviamo sulle auto moderne dove sono presenti fino ad 80 computer denominate ECU (*electronic control units*), esse variano dal controllo del motore e della trasmissione fino al controllo audio e di climatizzazione ma anche i display e la strumentazione di bordo. Sono presenti più di 100 milioni di linee di codice e tutti i sistemi sono collegati tramite CAN bus e oltre 2 km di cavi. Tale parte incide sempre più sul costo di una automobile.

Una parte molto importante di un sistema embedded è il software che viene eseguito su di esso in quanto deve sfruttare risorse limitate mantenendo performance elevate. Ad esempio la corretta esecuzione di un programma in C, C#, Java etc. non tiene conto del tempo impiegato per l'esecuzione e ogni nostra astrazione è fondata su questa premessa; il tempo di esecuzione di un programma non è un fattore certo e ripetibile a meno che non prendiamo in considerazione una granularità molto grossa per l'analisi. Alcuni esempi che sfruttano l'irrelevanza del tempo nei sistemi normali sono il *garbage collector*, la *just-in-time* compilation, il *power managment* e molti altri.

In un sistema embedded invece bisogna tener conto delle risorse limitate come la poca memoria e la dimensione della parola più corta e la frequenza di clock bassa. Per tener conto di queste limitazioni ci si concentra sull'efficienza scrivendo software a basso livello (C, Assembly), si evitano sistemi operativi con molte funzionalità, si utilizzano architetture specializzate (DSP, Network) si utilizzano sistemi di interconnessione specializzati. Inoltre i sistemi embedded si differenziano da quelli *general-purpose* in quanto devono rispettare dei vincoli temporali (il più velocemente possibile non è abbastanza), la concorrenza è intrinseca, le specifiche dei processori sono specializzate per rispettare i vincoli temporali supportare le operazioni più comuni e utilizzare specifici tipi di dati ed infine i programmi devono poter essere eseguiti *per sempre* il reboot del sistema non è accettabile.

Il concetto di *real-time* è più complicato del concetto del *prima possibile*, in realtà esso indica dei limiti temporali che non possono essere valicati, oggi i programmatori di sistemi embedded costruiscono il sistema e successivamente lo testano per il timing. Un meccanismo basato sui modelli cerca di specificare il comportamento dinamico inclusi i vincoli temporali per poi *compilare* un'implementazione che rispecchi le specifiche.

In molti casi i sistemi operativi real-time (*RTOS*) sono utilizzati in maniere inappropriate, senza considerare particolari principi o funzionalità. Gli sviluppatori modificano le priorità fino a quando il prototipo non soddisfa i test. Tuttavia il sistema che ne risulta è fragile e un piccolo

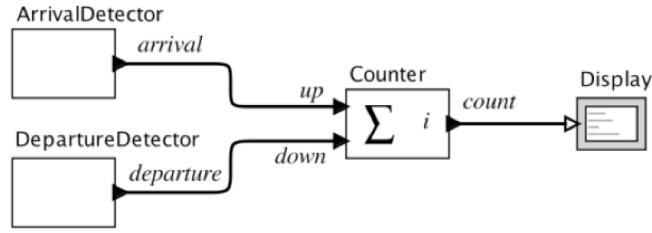


Figura 21: Schema di un contatore per un parcheggio

cambiamento nelle condizioni di operatività può causare grandi cambiamenti nel comportamento del sistema.

L'astrazione ha lo scopo di nascondere i dettagli dell'implementazione in modo da fornire una piattaforma sulla quale progettare. Un design basato su modelli può essere pensato a diversi livelli come strutturale, funzionale, orientato agli attori, ecc. questo permette uno sviluppo intuitivo del sistema tramite degli artifici che lo imitano. Un esempio è il *modello matematico* il quale rappresenta il sistema tramite un insieme di definizioni e formule matematiche. Creare un modello matematico di tutte le parti di un sistema può portare alla costruzione automatica del sistema come avviene per un compilatore.

Esistono diverse tecniche di modello ognuna adatta a rappresentare un aspetto diverso del sistema, ad esempio:

- **Macchine a stati:** utile per rappresentare decisioni logiche sequenziali.
- **Modelli basati sul flusso dei dati:** utili per esplorare il parallelismo e per il processo dei segnali.
- **Modelli ad eventi discreti:** nel quale viene esplicitato il tempo.
- **Modelli time-driven:** che studiano eventi periodici e azioni temporali

## 2.1 Modelli di computazione

Un *modello di computazione* è un assegnamento di semantica, ovvero di significato, alla sintassi definita dal modello. Il *MoC* definisce le regole per eseguire il modello, queste regole determinano come gli attori intervengono sulla computazione, aggiornano i loro stati interni e interagiscono con l'esterno. Infine il modello di computazione definisce come diversi componenti comunicano tra loro.

Prendiamo in considerazione l'esempio di Figura 21 nel quale viene mostrato un contatore per le auto che entrano ed escono da un parcheggio. In questo caso vediamo che il sistema è composto da:

**Segnali puri:**  $up, down : \mathbb{R} \rightarrow \{absent, present\}$

**Attori discreti:**  $Contatore : (\mathbb{R} \rightarrow \{absent, present\})^P \rightarrow (\mathbb{R} \rightarrow \{absent\} \cup \mathbb{N})_{-P} = \{up, down\}$

Definiamo *reazione* il fenomeno per cui per qualsiasi  $t \in \mathbb{R}$  dove  $up(t) \neq absent$  o  $down(t) \neq absent$  il *Contatore* reagisce ovvero produce un output in  $\mathbb{N}$  e modifica il suo stato interno. Per ogni  $t \in \mathbb{R}$  la porta  $p$  ha una valorizzazione ovvero un assegnamento dei valori che per la porta  $P = \{up, down\}$  corrisponde all'assegnamento sui due ingressi di uno dei due valori  $\{absent, present\}$ . Una reazione non è altro che una valorizzazione dell'output, in questo caso *count* assume uno dei valori nel set  $\{absent\} \cup \mathbb{N}$ .

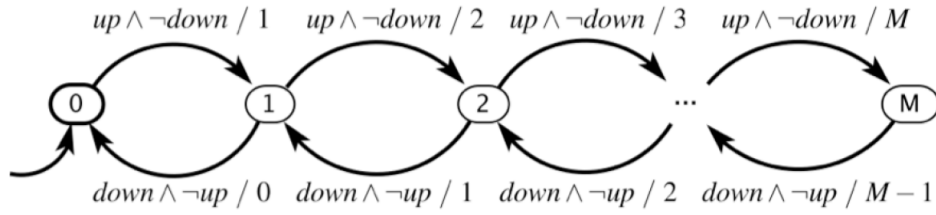


Figura 22: Macchina a stati finiti di un contatore per un parcheggio

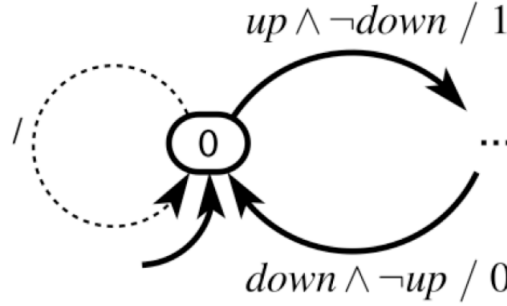


Figura 23: Esempio di transizione di default

Un'altra definizione che dobbiamo dare è quella dello *spazio degli stati* che non è altro che l'insieme degli stati in cui si può trovare il nostro contatore. Nel nostro caso il parcheggio ha un numero finito  $M$  di posti disponibili perciò lo spazio degli stati per il nostro contatore è

$$State = \{0, 1, 2, 3, \dots, M\}$$

Un modo per rappresentare il comportamento del nostro sistema è quello di utilizzare una *Macchina a Stati Finiti (FSM)* come quella di Figura 22 nella quale possiamo distinguere alcuni tratti caratteristici:

**Guardia:** insieme di valori che indicano i valori che devono essere presenti sugli input affinché avvenga un cambio di stato, per l'arco che va da 0 a 1 la guardia è:

$$up \wedge \neg down$$

**Stato iniziale:** è quello indicato da una freccia entrante in questo caso è lo stato indicato da 0

**Valore di uscita:** è il valore che viene prodotto durante un passaggio di stato in questo caso è il valore del contatore, nella macchina a stati finiti è indicato dal valore a destra della *guardia*.

Per definire una macchina a stati finiti formalmente sono necessari gli *stati*, gli *inputs*, gli *outputs*, gli *update* e lo *stato iniziale*.

Una *transizione di default* è una transizione che viene attivato solo quando le transizioni di default non sono attive e nessuna guardia è valutata come vera, un esempio di tale tipo di transizione è mostrata in Figura 23.

Una transizione *stuttering* è una transizione di default implicita che viene abilitata in assenza di input e che non produce output. La *ricettività* è una proprietà delle macchine a stati che afferma

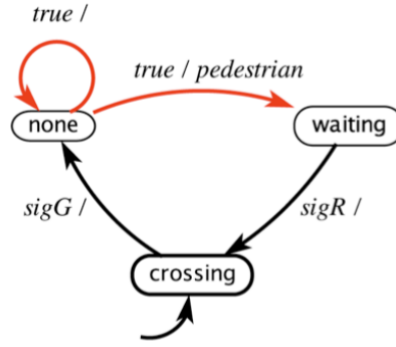


Figura 24: Esempio di macchina a stati non deterministica

che per qualsiasi valore in input almeno una transizione è abilitata, le strutture con transizioni di default fanno in modo che le FSM siano ricettive.

Il *determinismo* è una proprietà importante delle macchine a stati finite e afferma che in ogni stato per tutti i valori di input esattamente una transizione viene abilitata.

Il *comportamento* di una FSM è definito come una sequenza di transizioni non *stuttering*, una *traccia* è un insieme di input, stati e output di un comportamento; un *albero di computazione* è una rappresentazione grafica di tutte le possibili tracce. Tale proprietà fa sì che le FSM siano analizzabili formalmente.

Il *non determinismo* è la possibilità di avere, in una FSM, due possibili scelte per lo stesso valore di ingresso, un esempio di non determinismo è mostrato in Figura 24; la funzione di aggiornamento si trasforma in:

$$possibleUpdate : States \times Inputs \rightarrow 2^{States \times Outputs}$$

Il non determinismo è molto utile nella fase di modellazione del sistema in quanto permette di modellizzare aspetti ignoti riguardante l'ambiente esterno, nascondere dettagli di una specifica, infine il non determinismo è più compatto da rappresentare.

Le macchine a stati finiti permettono di rappresentare un sistema in modo tale che esso sia analizzato matematicamente e manipolato, modellare l'ambiente attorno al sistema, modellare cosa il sistema deve o non deve fare ed infine è un modo per verificare se il sistema rispetta le specifiche.

### 2.1.1 Composizione di macchine a stati

Esistono tre tipi di composizione nelle macchine a stati e queste sono:

- Composizione side-by-side.
- Composizione in cascata.
- Composizione gerarchica.

Un esempio di composizione *side-by-side* è mostrata in Figura 25, in questo caso due stati sono posizionati in parallelo, la questione principale è come la macchina reagisce agli input, le possibilità sono due, i due stati reagiscono *insieme* e allora si parla di composizione *sincrona*; in caso in cui invece gli stati reagissero in maniera *indipendente*, allora si parla di composizione *asincrona*.



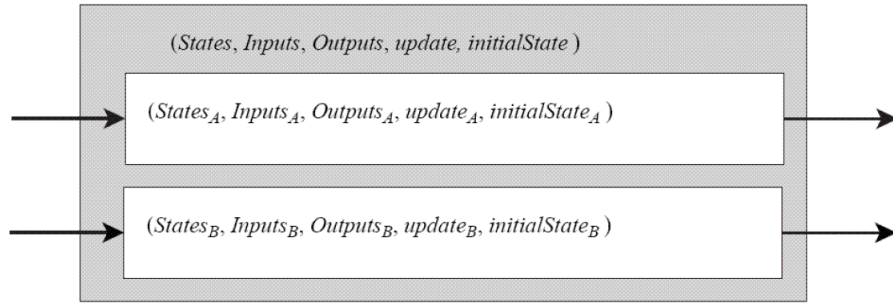


Figura 25: Esempio di composizione side-by-side

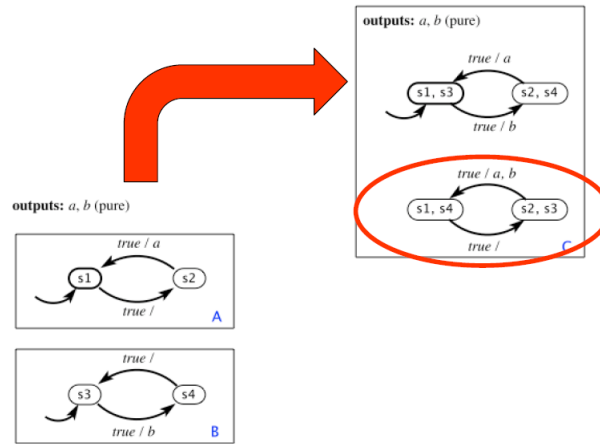


Figura 26: Esempio di composizione side-by side sincrona

**Composizione sincrona** In questo caso gli stati reagiscono insieme, la macchina a stati che si ottiene è formata dalla composizione degli stati precedenti:

$$S_C = S_A \times S_B$$

La macchina a stati che si ottiene è mostrata in Figura 26 vediamo che due stati non sono raggiungibili in quanto non vi è alcun modo in cui gli ingressi possano arrivare a quel valore. Nel caso di composizione sincrona la nuova macchina che si viene a creare può essere espressa come:

$$\begin{aligned}
\text{States} &= States_A \times States_B \\
\text{Inputs} &= Inputs_A \times Inputs_B \\
\text{Outputs} &= Outputs_A \times Outputs_B \\
\text{initialState} &= initialState_A \times initialState_B \\
((S_A(n+1), S_B(n+1)), (o_A(n), o_B(n))) &= update((S_A(n), S_B(n)), (i_A(n), i_B(n)))
\end{aligned}$$

dove

$$\begin{aligned}
(S_A(n+1), o_A(n)) &= update_A(S_A(n), i_A(n)) \text{ and} \\
(S_B(n+1), o_B(n)) &= update_B(S_B(n), i_B(n))
\end{aligned}$$

**Composizione asincrona** In questo caso gli stati reagiscono indipendentemente gli uni dagli altri, in questo caso tutti gli stati risultano raggiungibili come si vede dalla Figura 27 la

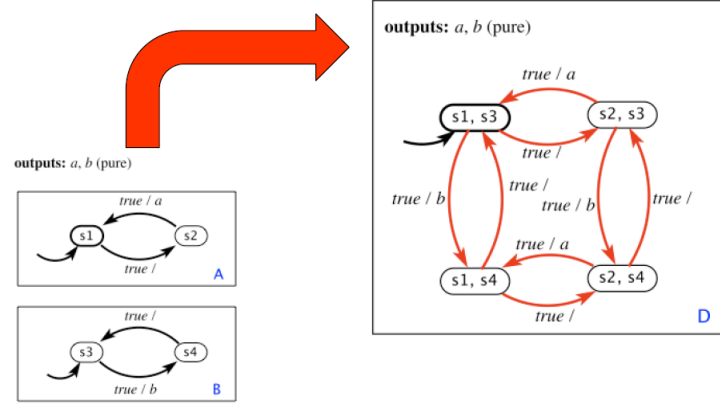


Figura 27: Esempio di composizione side-by-side asincrona

composizione degli stati è eseguita tramite la semantica intervallata ed è il risultato di

$$S_D = S_A \times S_B$$

Nel caso invece di composizione asincrona la macchina a stati finita viene descritta da:

$$\begin{aligned}
\text{States} &= \text{States}_A \times \text{States}_B \\
\text{Inputs} &= \text{Inputs}_A \times \text{Inputs}_B \\
\text{Outputs} &= \text{Outputs}_A \times \text{Outputs}_B \\
\text{initialState} &= \text{initialState}_A \times \text{initialState}_B \\
((S_A(n+1), S_B(n+1)), (o_A(n), o_B(n))) &= \text{update}((S_A(n), S_B(n)), (i_A(n), i_B(n)))
\end{aligned}$$

dove

$$\begin{aligned}
(S_A(n+1), o_A(n)) &= \text{update}_A(S_A(n), i_A(n)) \text{ and } S_B(n+1) = S_B(n) \text{ and } o_B = \text{absent or} \\
(S_B(n+1), o_B(n)) &= \text{update}_B(S_B(n), i_B(n)) \text{ and } S_A(n+1) = S_A(n) \text{ and } o_a = \text{absent}
\end{aligned}$$

### 2.1.2 Composizione in cascata

La *composizione in cascata* non è altro che il collegamento in serie di due macchine a stati come si vede dalla Figura 28 dove l'output della prima macchina a stati è connesso all'input della seconda macchina. Anche in questo caso possiamo parlare di composizione sincrona e asincrona. Analizziamo ora un esempio di un semaforo nel quale una macchina a stati descrive il comportamento del semaforo utilizzato per le automobili Figura 29 mentre una seconda macchina a stati descrive il comportamento del semaforo pedonale Figura 30. In Figura 31 vediamo la composizione delle due macchine a stati finiti nella quale però sono stati rimossi gli stati non raggiungibili. In caso di composizione *sincrona* la macchina reagisce simultaneamente nonostante sembra che esista una relazione causale tra le due. Ad esempio in questo caso sembrerebbe che la macchina agisca in modo sequenziale passando dal rosso al verde e viceversa ma bisogna tenere conto che in realtà gli stati che vengono raggiunti sono un insieme di stati e non vi è mai uno stato nel quale le due macchine non agiscono insieme.

Nel caso di composizione *asincrona* invece le uscite di una macchina entrano negli ingressi dell'altra in modo indipendente dai cambiamenti delle macchine. Un esempio di composizione asincrona è mostrata in Figura 32.

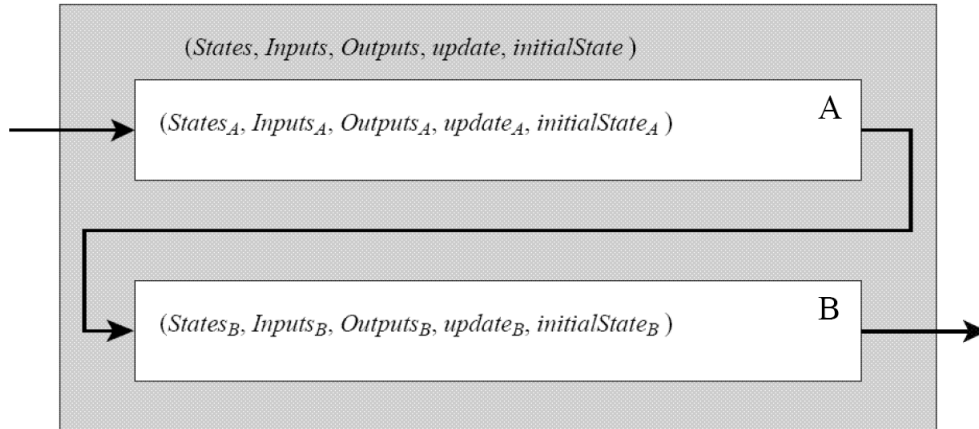


Figura 28: Esempio di composizione a cascata

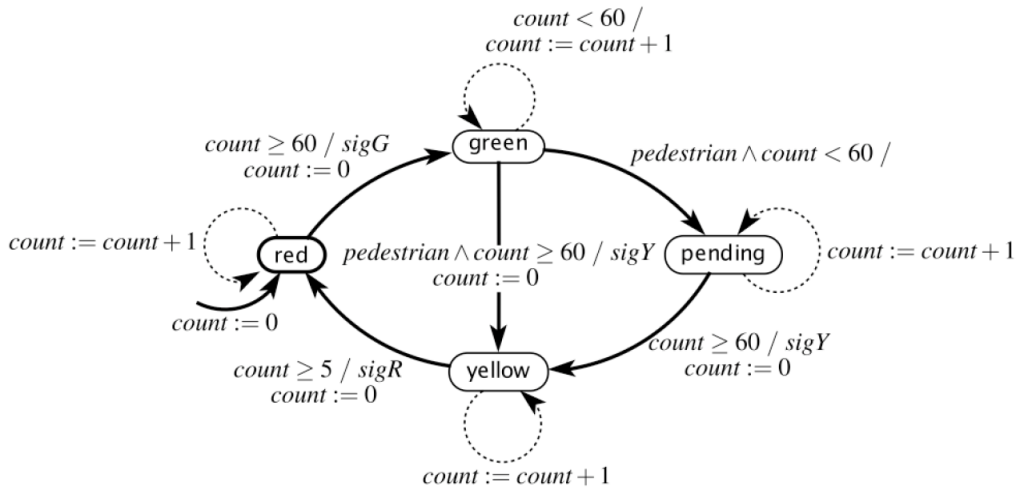


Figura 29: FSM che modella il comportamento di un semaforo per le auto

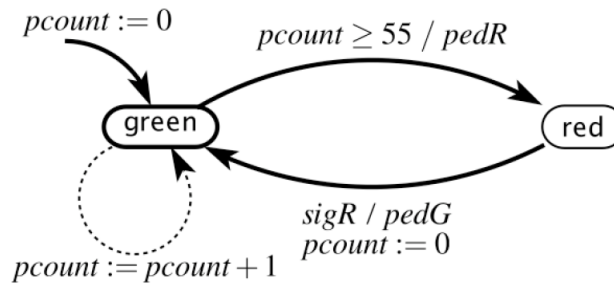


Figura 30: FSM che modella il comportamento di un semaforo per un attraversamento pedonale

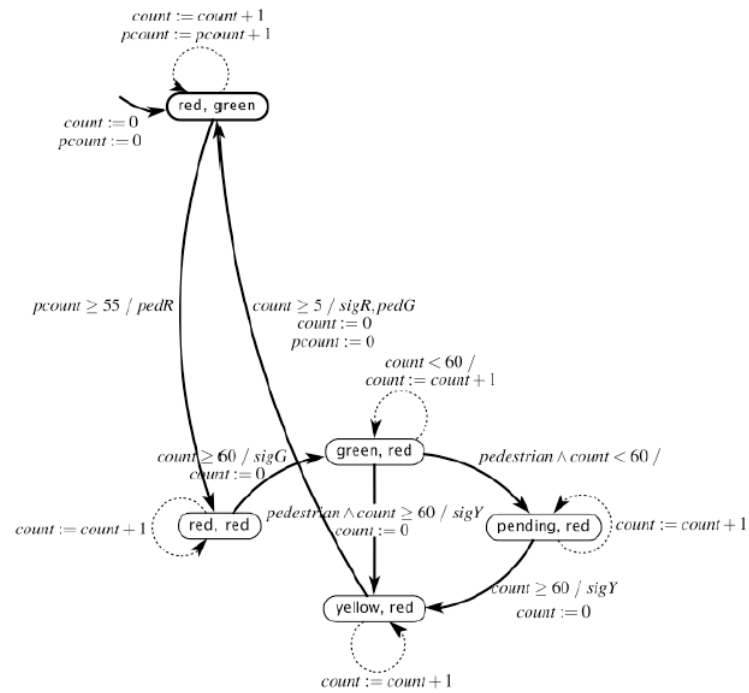


Figura 31: Composizione in cascata delle due FSM precedenti

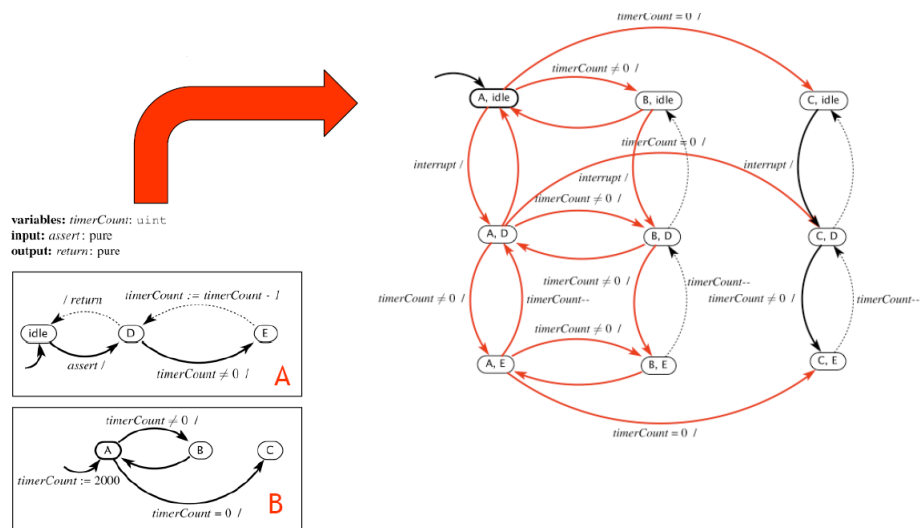


Figura 32: Esempio di composizione in cascata asincrona

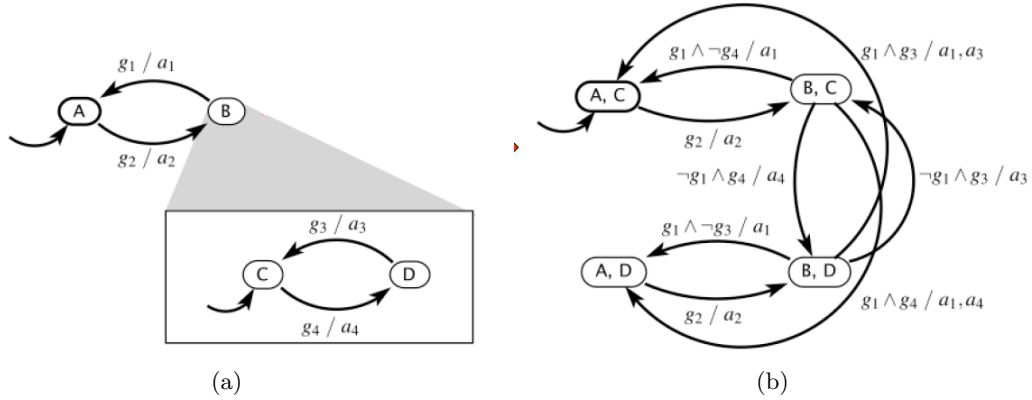


Figura 33: Esempio di FSM gerarchica

### 2.1.3 Composizione gerarchica

Quando parliamo di composizione gerarchica facciamo riferimento ad un particolare tipo di FSM nella quale uno stato è rappresentato da una seconda FSM come nel caso di Figura 33 in questo caso prima si raggiunge lo stato superiore e a quel punto si entra nello stato inferiore, nel caso entrambe le macchine producano un output questo non deve essere conflittuale.

A differenza degli altri casi il caso gerarchico permette diversi tipi di composizione. Ad esempio nel primo caso che analizziamo parliamo di *memoria della transizione* ad esempio nella nostra macchina a stati quando raggiungiamo lo stato B entriamo nella FSM composta dagli stati C e D a quel punto potremmo muoverci all'interno di quegli stati fino a quando non si presenti in ingresso il valore  $g_1$  che ci fa tornare allo stato A, nel caso di FSM con memoria una volta che torneremo nello stato B ripartiremo dall'ultimo stato in cui abbiamo lasciato la macchina, un esempio delle transizioni è mostrato di seguito

$$A \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} D \xrightarrow{g_1/a_1} A \xrightarrow{g_2/a_2} D \xrightarrow{g_3 \wedge g_1/a_3, a_1} A \dots$$

Esistono poi le FSM gerarchiche con reset come quella di Figura 34 in questo caso la transizione che entra nello stato gerarchico è indicata da una freccia vuota. Il reset fa in modo che ad ogni ingresso in B lo stato dal quale si parte sia sempre lo stato iniziale della sotto-FSM nel nostro esempio lo stato C. Le transizioni in questo caso sono:

$$A \xrightarrow{g_2/a_2} C \xrightarrow{g_4/a_4} D \xrightarrow{g_1/a_1} A \xrightarrow{g_2/a_2} C \xrightarrow{g_4 \wedge g_1/a_4, a_1} A \dots$$

L'ultima tipologia di composizione gerarchica è quella di Figura 35 ovvero una composizione gerarchica con *preemptive*; il preemptive è un meccanismo per cui la valutazione della guardia che riporta la macchina nello stato A è valutata prima di raggiungere lo stato corrente nella sottomacchina, nel caso la valutazione sia vera allora lo stato corrente non viene raggiunto.

## 2.2 Attori

Gli attori non sono altro che dei componenti nei quali input e output sono esposti come in Figura 36 in questo caso però si tratta di modelli a tempo continuo, anche per gli attori è possibile effettuare la composizione, la più comune è la composizione a *feedback*.

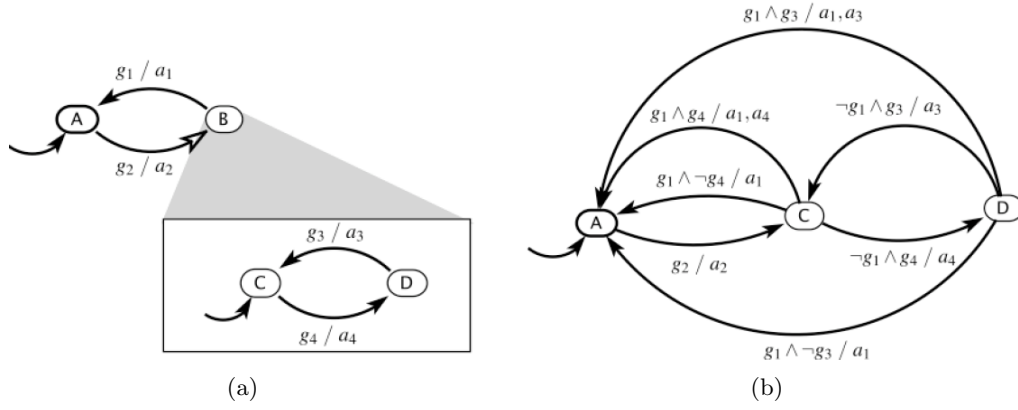


Figura 34: Esempio di FSM gerarchica con reset

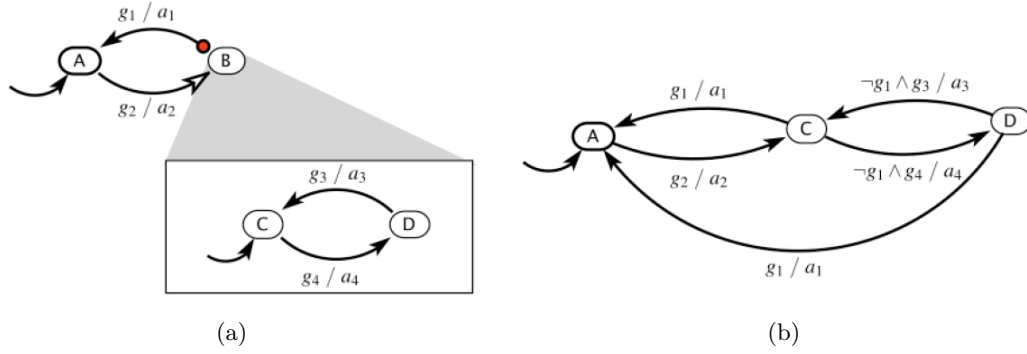


Figura 35: Esempio di FSM gerarchica con preemitive

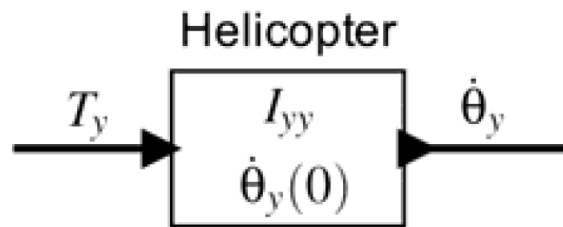


Figura 36: Esempio di attore

## 2.3 Dataflow model

...

## 2.4 Multitasking

Veramente volete ancora appunti sui *pthread*?

### 3 Flusso di progetto nei sistemi eterogenei

Come abbiamo visto i sistemi embedded sono sistemi specializzati progettati per svolgere una o poche funzioni, molto spesso con vincoli stringenti. Questi tipi di sistemi sono progettati come *SoC Multiprocessore* che sono diventati lo standard di fatto di questo tipo di sistemi. Molti dei sistemi embedded hanno dei vincoli stringenti per quanto riguarda le prestazioni, l'ottimizzazione delle applicazioni per sistemi embedded complessi è un problema difficile, che richiede progettisti con un alto livello di esperienza per identificare la soluzione migliore.

Solitamente si utilizzano delle piattaforme hardware di base, tuttavia i vincoli stringenti richiedono processori dedicati per accelerare specifiche funzioni.

Per definire il problema dobbiamo considerare numerosi aspetti:

**Job:** ovvero l'attività da eseguire e completare affinché il sistema soddisfi le specifiche.

**Punto di implementazione:** indica il modo nel quale svolgere un lavoro. Rappresenta una combinazione di *latenza* e *risorse necessarie*.

**Mapping:** assegna ogni *job* ad un possibile *punto di implementazione*, in modo da rispettare i vincoli imposti dalle risorse.

**Scheduling:** determina l'ordine di esecuzione di tutti i lavori.

**Obiettivo:** minimizzare il tempo di esecuzione complessivo dell'applicazione sull'architettura designata

Per quanto riguarda l'*obiettivo* di minimizzare il tempo di esecuzione possiamo intervenire in diversi modi, come quello di analizzare, valutare ed ottimizzare differenti alternative, ed una volta individuata una soluzione valutarne la qualità prima della sua implementazione.

La progettazione di sistemi eterogenei a multi processore richiede diversi fasi:

- Una fase di partizionamento dell'applicazione (*partitioning*)
- Una fase di assegnamento dei task ai diversi elementi architetturali (*mapping*)
- La determinazione dell'ordine di esecuzione dei task (*scheduling*)

La parte di *scheduling* e di *mapping* è un problema *NP-completo* inoltre il problema di un design e l'interfacciamento di componenti eterogenei può generare delle soluzioni non ammissibili.

Una rappresentazione imparziale e unificata di software ed hardware supporta la fase di progettazione ed analisi, permette una facile valutazione delle soluzioni individuate e inoltre permette la migrazione dei task tra la parte hardware e quella software. Tecniche di progettazione iterative permettono di valutare soluzioni differenti, aiuta a determinare l'implementazione migliore ed infine il partizionamento dei moduli permette un matching più preciso rispetto ai criteri di progettazione. Esistono inoltre dei metodi di parallelizzazione che possono valutare il grado di parallelismo a livello di :

- Istruzione
- Dati
- Task



Tuttavia si introducono problemi di dipendenza che devono essere soddisfatti.

il parallelismo dei dati è definito come il parallelismo ottenuto dall'esecuzione concorrente che utilizza porzioni di dati differenti. Il parallelismo dei task invece viene definito come il parallelismo ottenuto dalla computazione su diverse strutture dati. In molti casi si utilizza un misto delle due tecniche soprattutto in ambito scientifico.

Un *grafico dei task* è un grafico  $G = (T, E)$  nei quali i nodi rappresentano un gruppo di istruzioni e gli archi rappresentano le dipendenze, solitamente gli archi vengono indicati con il quantitativo di dati da trasferire in modo da considerare anche il ritardo di comunicazione. Nel caso di grafi ciclici adotteremo la rappresentazione *gerarchica dei grafi* nei quali i nodi son suddivisi in tre classi:

**semplici:** un task senza altri sotto task,

**composto:** quando un task ha altri sotto task associati,

**loop:** task che rappresentano un ciclo in cui il corpo è un sotto-grafo