Copyright © 2004 Moreno Marzolla

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 2.5 Italy License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.5/it/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Affidabilità

- L'affidabilità è definita come *la probabilità che il sistema funzioni senza errori per un dato intervallo di tempo, in un dato ambiente e per un determinato scopo*
- Questo vuol dire diverse cose in base al sistema. all'ambiente e allo scopo.
- Inoltre, utenti diversi percepiscono una diversa affidabilità del sistema
- Informalmente, l'affidabilità misura "quanto bene" gli utenti del sistema pensano che esso fornisca i servizi che essi richiedono

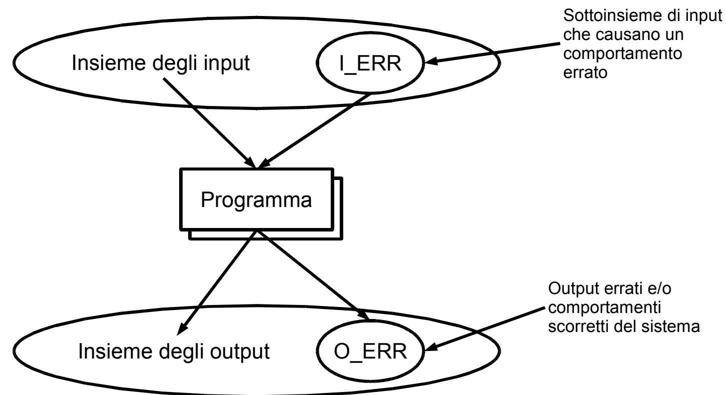
Affidabilità del software

- Difficilmente può essere definita oggettivamente
  - Misure di affidabilità che sono citate fuori da ogni contesto non sono significative
- Richiede il profilo di utilizzo del sistema per essere definita
  - Il profilo di utilizzo definisce il pattern di utilizzo "tipico" del sistema
  - Può essere ragionevole quantificare l'affidabilità di un sistema embedded che controlla sempre lo stesso hardware; non è significativo specificare l'affidabilità di un sistema interattivo che viene usato in modi differenti—a meno che non si specifichino questi modi di utilizzo
- Bisogna considerare le conseguenze dei fallimenti
  - Non tutti i fallimenti del sistema sono ugualmente gravi. Il sistema viene percepito come inaffidabile se si verificano fallimenti gravi

Fallimenti e *fault*

- Un fallimento corrisponde ad un comportamento run-time inaspettato (ed errato) osservato da un utente del sistema
- Un *fault* è una caratteristica statica del software che causa il fallimento
- I fault non necessariamente causano fallimenti. Lo fanno solo quando la componente errata del sistema viene utilizzata

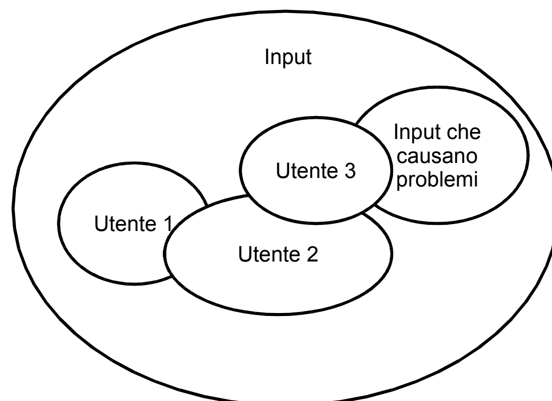
Mappatura input/output



Migliorare l'affidabilità

- L'affidabilità migliora quando si rimuovono i fault che compaiono nelle parti del sistema che sono più frequentemente usate
- Rimuovere l'x% di tutti i fault dal software non necessariamente causa un miglioramento dell'x% dell'affidabilità complessiva
- In uno studio, la rimozione del 60% dei difetti del software ha migliorato l'affidabilità del 3%
- E' importante rimuovere i difetti che causano le conseguenze più serie

Percezione dell'affidabilità



Affidabilità e metodi formali

- L'uso di metodi formali per lo sviluppo del sistema può portare ad un sistema maggiormente affidabilità, poiché consentono di dimostrare che il sistema si comporta in modo conforme ai suoi requisiti
- Lo sviluppo di una specifica formale obbliga una analisi dettagliata dei requisiti del sistema che può portare a scoprire anomalie ed omissioni
- Tuttavia, i metodi formali possono anche *non* migliorare l'affidabilità del sistema
  - Le specifiche potrebbero non riflettere i requisiti reali degli utenti
  - Le dimostrazioni di correttezza potrebbero contenere errori
  - Le dimostrazioni di correttezza potrebbero fare delle assunzioni implicite sull'ambiente in cui il sistema opera che non sono corrette

Affidabilità ed efficienza

- Man mano che aumenta l'affidabilità, l'efficienza tende a diminuire
- Per rendere un sistema maggiormente affidabile può essere necessario inserire codice ridondante per effettuare controlli a tempo di esecuzione. Tutto ciò può causare rallentamenti
- Però...
  - Spesso l'affidabilità è più importante dell'efficienza
  - I computer sono veloci ed economici. Macchine più veloci aumentano le aspettative degli utenti in termini di affidabilità
  - I sistemi inaffidabili semplicemente non vengono usati
  - I sistemi inaffidabili possono essere difficili da migliorare
  - I costi connessi a perdite di dati sono molto alti. L'affidabilità serve!

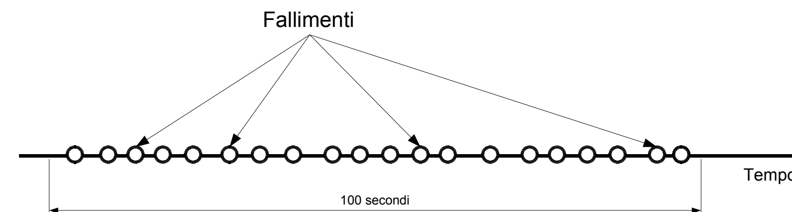
Metriche per l'affidabilità

- L'affidabilità dell'hardware non è la stessa cosa dell'affidabilità del software
  - Componenti hardware guaste possono essere riparate o sostituite con altre componenti identiche. Il disegno complessivo dell'hardware di solito e corretto, e i difetti sono semplicemente causati da logorio
- Fallimenti del software sono causati spesso da problemi di design.
  - In più, nel caso di alcuni fallimenti software, il sistema continua a funzionare più o meno correttamente

Metriche / 1

- *Probability of failure on demand* (POFOD)
  Più o meno "Probabilità di fallimento per richiesta”
  - Probabilità che si verifichi un fallimento quando viene fatta una richiesta al sistema
  - POFOD = 0.001 significa che una richiesta ogni 1000 causa un fallimento
  - Metrica rilevante per sistemi critici o non-stop
- *Rate of fault occurrence* (ROCOF)
  Tasso di occorrenza dei fallimenti
  - Frequenza con cui accadono comportamenti anomali
  - ROCOF=0.02 significa che ci si aspettano 2 fallimenti ogni 100 unità di tempo di funzionamento del sistema (occorre specificare l'unità di misura: 2 ogni 100 secondi? 100 giorni? 100 anni?)
  - Metrica rilevante per sistemi operativi

Esempio



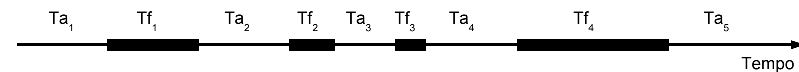
POFOD = 4 / 20 = 0.2
ROCOF = 4 / 100s = 0.04

Metriche / 2

- *Mean time to failure* (MTTF)
  Tempo medio tra fallimenti
  - Misura il tempo che passa tra due fallimenti consecutivi
  - MTTF=500 significa che il tempo medio tra due fallimenti consecutivi è di 500 unità di tempo
  - Rilevante per sistemi in cui le transazioni possono durare a lungo
- *Availability* (AVAIL)
  Disponibilità
  - Misura quanto è probabile che il sistema sia operativo in un dato istante. Tiene in considerazione i tempi di riparazione/riavvio
  - Availability di 0.998 significa che il sistema è disponibile per 998 unità di tempo su 1000
  - Metrica rilevante per sistemi che devono funzionare in continuazione, ad es. sistemi telefonici

Esempio

- MTTF = somma(Ta_i) / n
- AVAIL = somma(Ta_i) /
  (somma (Ta_i) + somma (Tf_i))



Misurare l'affidabilità

- Misurare il numero di fallimenti dato un numero N di input dati al sistema
  - Da questo si calcola POFOD
- Misurare il tempo che intercorre tra due fallimenti consecutivi
  - Da questo si calcola ROCOF e MTTF
- Misurare il tempo necessario per far ripartire il sistema dopo un fallimento
  - Serve per calcolare AVAIL

Conseguenze dei fallimenti

- Le misure di affidabilità non tengono in considerazione le conseguenze dei fallimenti
- Fallimenti transienti possono non avere conseguenze gravi, ma altri tipi di guasti possono causare perdita di dati o interruzione del servizio
- Può essere necessario identificare diverse classi di fallimenti e usare le metriche appropriate per ciascuna classe

Specificare i requisiti di affidabilità

- I requisiti di affidabilità sono raramente espressi in modo quantitativo e verificabile
- Per verificare i requisiti di affidabilità occorre definire un profilo operazionale come parte del collaudo
  - Profilo operazionale = in che modo il sistema verrà “tipicamente” utilizzato
- L'affidabilità è dinamica: tutte le specifiche di affidabilità legate al codice sorgente non hanno senso.
  - "Non più di N fallimenti ogni 1000 linee di codice"

Classificare i fallimenti

Transienti	Si verificano solo con certi input
Permanenti	Si verificano per tutti i possibili input
Recuperabili	Il sistema può ripartire senza interventi esterni
Non recuperabili	L'operatore deve intervenire per far ripartire il sistema
Non corruttivi	Il fallimento non causano corruzione di dati o dello stato del sistema
Corruttivi	Il fallimento corrompe dati e/o lo stato del sistema

Passi per creare una specifica di affidabilità

- Per ciascun sottosistema, analizzare le conseguenze dei possibili fallimenti
- Partizionare i fallimenti così individuati nelle classi appropriate
- Per ciascun requisito di affidabilità, e considerando le classi di fallimenti individuati in precedenza, definire quantitativamente usando le metriche appropriate

Esempio: sportelli Bancomat

- Ciascuno sportello è usato 300 volte al giorno
- La banca ha 1000 sportelli Bancomat
- Dopo 2 anni il sistema viene sostituito
- Ciascuno sportello gestisce circa 200000 transazioni durante la sua vita (300 transazioni/giorno * 365 giorni * 2 anni)
- Sono 300000 transazioni in tutto per ciascun giorno

Esempio di specifica dei fallimenti

Failure class	Example	Reliability metric
Permanent, non-corrupting.	The system fails to operate with any card which is input. Software must be restarted to correct failure.	ROCOF 1 occurrence/1000 days
Transient, non-corrupting	The magnetic stripe data cannot be read on an undamaged card which is input.	POFOD 1 in 1000 transactions
Transient, corrupting	A pattern of transactions across the network causes database corruption.	Unquantifiable! Should never happen in the lifetime of the system

Validare le specifiche dell'affidabilità

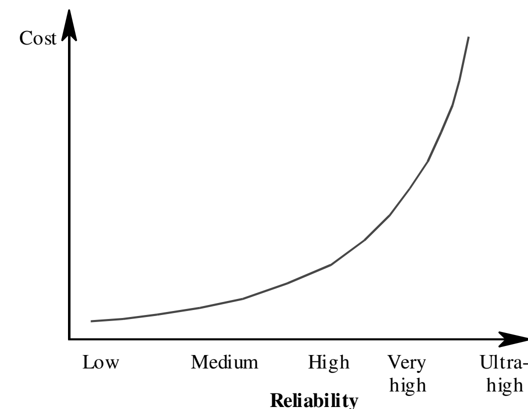
- E' impossibile validare empiricamente specifiche di affidabilità molto stringenti
- "No database corruptions" significa POFOD di meno di 1 su 200 milioni
  - Se per eseguire una transazione si impiega 1 secondo, simulare tutte le transazioni di un giorno richiede 300000 secondi (3,5 giorni)
  - Testare la specifica significa che la simulazione del sistema dura di più del suo ciclo di vita!!

L'"economia" dell'affidabilità

- **...E quindi?**

  Ottenere affidabilità molto alta è costoso. Spesso può essere più conveniente (dal punto di vista economico) tollerare possibili fallimenti e pagarne le conseguenze
  - Però dipende... Se vi fate la reputazione di produrre sistemi inaffidabili, anche se ne pagate le conseguenze nessuno si rivolgerà più a voi!
  - Invece, per altri tipi di sistemi (software per contabilità) una affidabilità modesta è adeguata

Costi per migliorare l'affidabilità



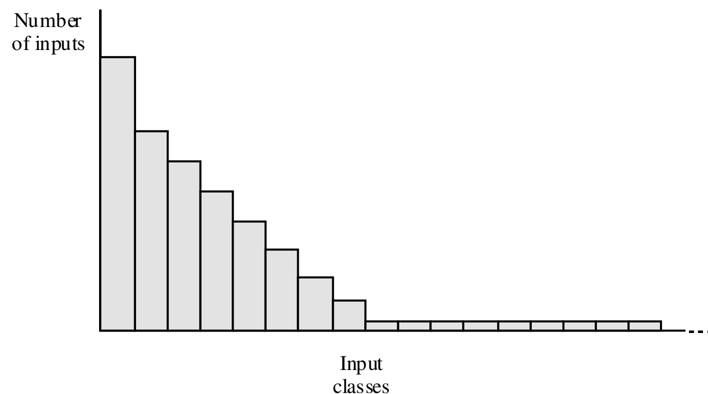
Test statistici

- Usiamo il numero di difetti (bug) che vengono scoperti durante la fase di test per predire l'affidabilità
  - Ma in questo caso, i casi di prova devono essere scelti in base al profilo operazionale del sistema, piuttosto che scelti in modo da individuare il maggior numero di errori
- Misurare il numero di errori scoperti consente di predire l'affidabilità del sistema
- Occorre specificare un livello di affidabilità accettabile; il software deve essere migliorato finché il livello di affidabilità desiderato viene raggiunto
- Vediamo ora in dettaglio...

Procedure per test statistici

1) Determinare il profilo operazionale del sistema
   - Cioè, quali tipi di input saranno più probabili
2) Generare i dati di test in base a questo profilo
3) Applicare i test, e misurare il tempo di esecuzione tra fallimenti
4) Valutare l'affidabilità dopo che un numero statisticamente significativo di test sono stati effettuati

Generazione del profilo operazionale



Generazione del profilo operazionale

- Dovrebbe essere generato automaticamente quando possibile
  - Può essere difficile in caso di sistemi interattivi
  - Può essere difficile per sistemi nuovi o innovativi, poiché è impossibile anticipare in che modo verranno usati
  - Può essere facile individuare gli input "normali", cioè quelli più frequenti. Meno facile è individuare gli input improbabili, proprio perché sono improbabili

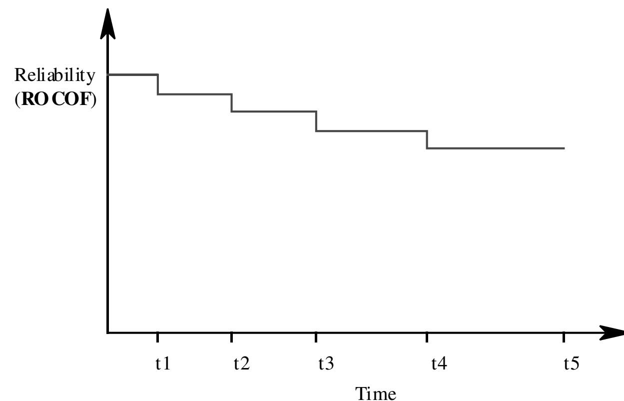
Difficoltà di generazione del profilo operazionale

- Incertezze intrinseche
  - Questo è particolarmente vero per sistemi nuovi che non sono mai stati usati in passato. L'incertezza è un problema minore nel caso di sistemi che ne rimpiazzano altri obsoleti
- Costi elevati
  - Dipende dal tipo di informazioni che sono raccolte, e dal modo in cui sono raccolte
- Incertezza statistica quando si richiede alta affidabilità
  - E' difficile stimare il livello di confidenza del profilo operazionale
  - I pattern di utilizzo del sistema possono cambiare col tempo

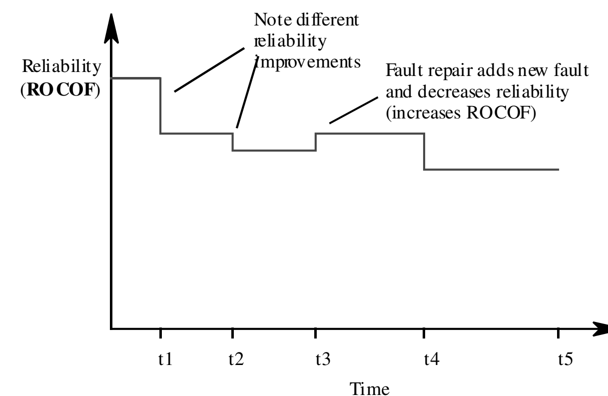
Modelli di crescita dell'affidabilità

- I modelli di crescita sono modelli matematici che rappresentano come l'affidabilità del sistema cambia man mano che i bug sono scoperti e risolti
  - Questi modelli estrapolano l'affidabilità in base ai dati correnti
  - Occorrono strumenti statistici per fare misure significative

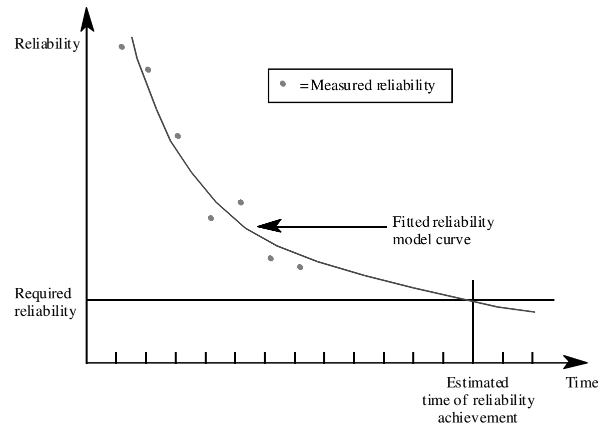
Modello di crescita dell'affidabilità... in teoria



Modello di crescita dell'affidabilità... in pratica



Predire l'affidabilità



Programmare l'affidabilità

- Chiaramente, gli utenti si aspettano che *tutto* il software sia totalmente affidabile.
  - Tuttavia, per applicazioni non critiche, gli utenti possono tollerare alcuni fallimenti sporadici
- Alcune applicazioni comunque richiedono una elevata affidabilità
- Domanda:

  Quali tecniche di programmazione possono essere usate per produrre software affidabile?

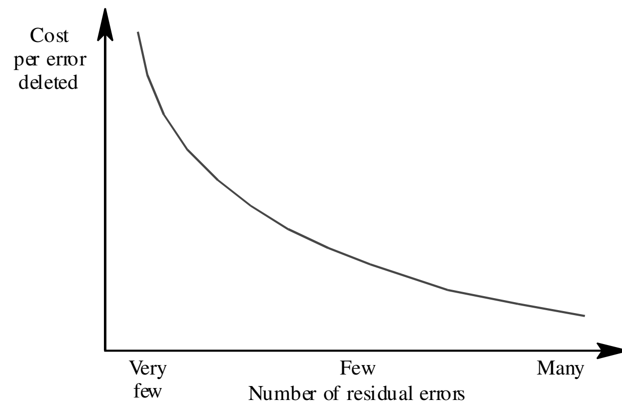
Come ottenere l'affidabilità

- *Fault avoidance*
  - Il sistema viene sviluppato in modo da non contenere errori
- *Fault detection*
  - Il processo di sviluppo del software è strutturato in modo tale che gli errori sono individuati e corretti prima di consegnare il sistema al cliente
  - Test, collaudi...
- *Fault tolerance*
  - Il software è scritto in modo tale che eventuali errori non necessariamente causano un fallimento totale del sistema

Fault Avoidance

- L'applicazione di principi rigorosi di ingegneria del software consente di sviluppare software (quasi) esente da errori
- Il software è esente da errori quando è conforme alle specifiche
  - **Non significa software che si comporta sempre correttamente**, dato che le specifiche stesse possono contenere errori
- Il costo di produrre software privo di errori è molto alto
  - Può risultare conveniente solo in certe situazioni
  - Può risultare economicamente più conveniente sopportare le conseguenze dei fallimenti

Costo di rimozione degli errori



Sviluppare software privo di errori

- Partire da una specifica precisa (possibilmente formale)
- Utilizzare *information hiding* e *incapsulamento dell'informazione*
- Sfruttare linguaggi di programmazione con tipizzazione stretta e controlli a tempi di esecuzione (run-time)
- Effettuare revisioni periodiche, in ogni fase del processo di sviluppo
- L'azienda produttrice del software deve essere interamente orientata alla qualità
- Test e collaudi *accurati* e *completi* sono indispensabili

Tecnica 1: Programmazione strutturata

- Esiste dagli anni '70
- Programmare senza `goto`
- do-while e repeat-until sono le uniche strutture iterative
- Applicare un approccio top-down
  - Spezzare elaborazioni complicate in funzioni e procedure più semplici
- La programmazione strutturata ha fornito il primo reale contributo e ha stimolato la discussione sulla programmazione
  - Non più (non solo?) un'arte, ma una disciplina che può essere resa rigorosa

Costrutti problematici

- Numeri in virgola mobile
  - Inerentemente imprecisi. Errori di arrotondamento/approssimazione possono rendere confronti non validi
  - Se *a* e *b* sono double, MAI dire *if (a == b) {...*
- Puntatori
  - Possono puntare a locazioni di memoria non allocate. Puntatori diversi che referenziano la stessa area di memoria possono causare inconsistenze e rendere il programma difficile da leggere
- Allocazione dinamica della memoria
  - Memory overflow
- Parallelismo
  - Può causare computazioni errate o deadlock causate da interazioni inaspettate o scheduling particolari

Costrutti problematici / 2

- Ricorsione
  - Se la condizione di terminazione è sbagliata, causa stack overflow
- Interrupts
  - Rendono i programmi difficili da capire, perché sono in effetti equivalenti a dei goto da una qualsiasi zona del programma verso la routine di servizio delle interruzioni
- *Non stiamo dicendo che questi costrutti non devono essere usati!*
  - Ma se sono usati, occorre prestare **molta** attenzione...

Tecnica 2: Information hiding

- Le informazioni devono essere rese disponibili solo alle parti del programma che ne hanno effettivamente bisogno
  - Principio di origine militare
  - Si realizza incapsulando lo stato e le operazioni all'interno di oggetti
- Perché questo riduce le possibilità di fallimenti?
  - Perché si riduce la probabilità di modificare "accidentalmente" le informazioni
  - Le informazioni sono circondate da "firewall" che impediscono che i problemi si propaghino ad altre parti del sistema
  - Dato che tutte le informazioni sono localizzate (alta coesione), il programmatore ha meno probabilità di commettere errori, e i revisori hanno maggiore probabilità di identificarli

Oggetti e tipi di dati astratti (ADT)

- In C++, classi e oggetti
- Il nome del *tipo di dato astratto* è il nome della classe
- Le operazioni sul ADT sono rappresentate dai metodi
- La rappresentazione dell'ADT è contenuta nella parte protetta/privata

Esempio: una coda in C++

```
class Queue {
public:
	Queue () ;
	~Queue () ;
	void Put ( int x ) ; // adds an item to the queue
	int Remove () ; // this has side effect of changing the queue
	int Size( ) ;  // returns number of elements in the queue
private:
	int front, back ;
	int qvec [100] ;
} ;
```

Fault Tolerance

- In situazioni critiche, i sistemi software devono tollerare i fallimenti
  - Tollerare i fallimenti (*fault tolerance*) significa che il sistema deve continuare a funzionare a dispetto di errori software
- Anche se viene dimostrato in qualche modo che il sistema è totalmente esente da errori (*fault free*), è comunque bene che sia anche *fault tolerant*:
  - Errori nelle specifiche
  - Errori durante la fase di verifica
  - Ricordiamo: fault-free=il programma è conforme alle specifiche

Come si tollerano i fallimenti?

- Rilevare il guasto
  - Il sistema deve rendersi conto in qualche modo che è avvenuto un fallimento
- Valutare l'entità dei danni
  - Occorre rilevare le parti del sistema che sono affette dal guasto
- Recuperare il guasto
  - Il sistema deve ripristinare il proprio stato ad un valore stabile e corretto
- Riparare il guasto
  - Il sistema deve essere modificato in modo che il guasto non si ripeta in futuro. Spesso i guasti sono transitori, quindi possono ripresen-tarsi con bassissima probabilità. In tal caso, riparare il guasto può non essere conveniente

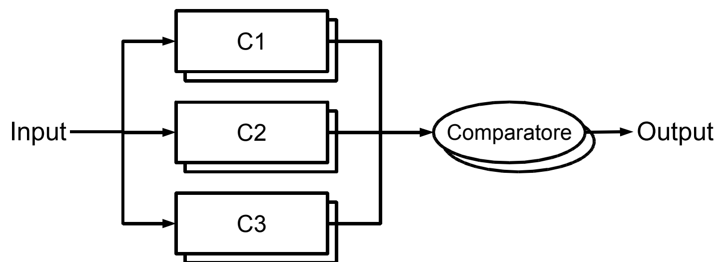
Recuperare dai fallimenti

- La maggior parte dei fallimenti sono di natura transiente e dipendono dagli specifici dati in ingresso. Il sistema può essere fatto ripartire dall'inizio
  - "L'applicazione ha eseguito una operazione non valida e verrà terminata"
- Se ciò è impossibile, il sistema può essere dinamicamente riconfigurato tramite sostituzione delle componenti difettose, senza causarne l'interruzione

Tolleranza ai guasti nel caso dell'hardware

- Triple-modular redundancy (TMR)
- Ci sono tre componenti identiche che ricevono gli stessi input e devono generare gli stessi output
- Se uno degli output è diverso dagli altri due, viene ignorato e la componente che l'ha prodotto è dichiarata guasta
- Perché funziona?
  - Si fa l'assunzione che le componenti hardware falliscano a causa di usura, non perché è mal progettata
  - Si assume che sia estremamente improbabile (=impossibile) che più componenti falliscano contemporaneamente

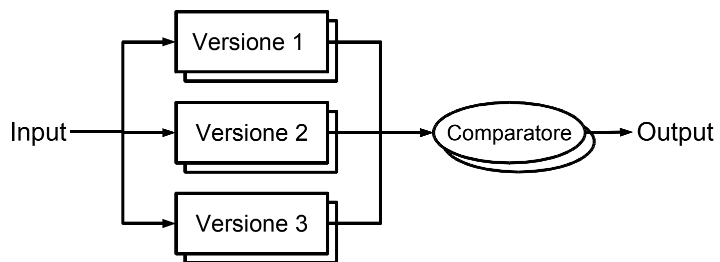
Affidabilità dell'hardware tramite TMR



Analogie software

- N-version programming
  - Implementare le stesse specifiche in modi diversi. Gli input sono passati a tutte le implementazioni contemporaneamente. L'output è selezionato in base a ciò che la maggioranza calcola. Questo approccio è usato, ad es., nell'Airbus 320.
  - Tener presente che è inefficace contro errori nelle specifiche
- Recovery blocks
  - Anche qui ci sono diverse implementazioni dello stesso sistema. Le versioni sono provate in sequenza, finché se ne trova una che fornisce la risposta corretta.
  - Il punto debole è che occorre avere un mezzo per capire quando una risposta è corretta (senza doverla ricalcolare!)

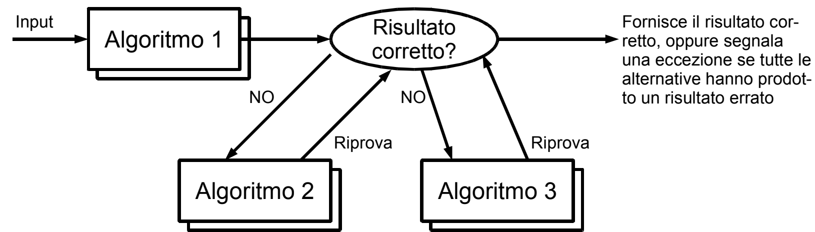
N-version programming



N-version programming

- Le diverse versioni devono essere sviluppate e implementate da team diversi
  - Questo per impedire che facciano gli stessi errori...
  - ...tuttavia ci sono evidenze empiriche che anche team diversi tendono a interpretare nello stesso modo errato le specifiche. Inoltre, team diversi tendono ad utilizzare gli stessi algoritmi e strutture dati

Recovery Blocks



Recovery blocks

- Occorre usare algoritmi diversi ad ogni tentativo allo sopo di ridurre la possibilità di aver introdotto errori comuni
- Il test di accettazione è difficile da implementare.
  - Non bisogna ricalcolare il risultato da zero: come facciamo ad essere sicuri che il valore calcolato dal test di accettazione sia quello giusto?
  - E' opportuno che la verifica del risultato sia fatta in modo più efficiente rispetto al calcolo dello stesso
- Analogamente ad N-version programming, anche questa strategia è suscettibile ad errori nelle specifiche

Gestione delle eccezioni

- Una eccezione è un errore causato da un evento inatteso
  - Interruzione dell'energia elettrica
  - Guasto di una componente
- Esistono meccanismi per la gestione delle eccezioni che evitano controlli continui di condizioni di eccezione
  - try { ... } catch

Esempio: frigorifero

```
void Control_freezer ( const float Danger_temp)
{
   float Ambient_temp ;
   // try means exceptions will be handled in this block
   // Assume that Sensor, Temperature_dial and Pump are
   //objects which have been declared elsewhere
   try {
      while (true) {
         Ambient_temp = Sensor.Get_temperature () ;
         if (Ambient_temp > Temperature_dial.Setting () )
            if (Pump.Status () == off)
            {
               Pump.Switch (on) ;
               Wait (Cooling_time) ;
            }
            else
               if (Pump.Status () == on)
                Pump.Switch (off) ;
         if ( Ambient_temp > Danger_temp )
               throw Freezer_too_hot ( ) ;
      }  // end of while loop
   }  // end of exception handling try block
   // catch indicates the exception handling code.
   catch ( Freezer_too_hot )
      Alarm.Activate () ;
}
```

Programmazione "difensiva"

- Approccio allo sviluppo di software in cui si parte dall'assunzione che possono esistere errori non individuati nel software
- Pertanto, il programma contiene codice per individuare e recuperare dagli errori
  - I sistemi di tipi di alcuni linguaggi di programmazione fanno in modo da individuare errori potenziali a tempo di compilazione, o a tempo di esecuzione
  - Controlli agli accessi in memoria e meccanismi per la gestione delle eccezioni consentono di individuare altri tipi di errori a tempo di esecuzione
  - Ulteriori controlli possono essere inseriti sotto forma di asserzioni

Stimare i danni

- Supponiamo di accorgerci che c'è stato un errore. Come facciamo a renderci conto dell'estensione dell'errore?
  - Occorre valutare in quale misura lo stato interno del sistema è stato affetto dal fallimento
  - Questo viene realizzato tramite "funzioni di validità", il cui scopo è controllare che il valore delle variabili e delle strutture dati sia all'interno dell'intervallo di validità ammesso
- Tecniche
  - Checksum per controllare errori di trasmissione
  - Puntatori ridondanti per controllare l'integrità di strutture dati
  - Timer per controllare la terminazione dei processi, contro l'insorgere di cicli infiniti. Se il processo non risponde entro un certo tempo, si assume che ci siano dei problemi

Esempio: classe Array con controllo di consistenza

```
template <class elem> class Robust_array {
public:
	Robust_array (int size = 20) ;
	~Robust_array () ;
	void Assign ( int Index, elem Val) ;
	elem Eval  (int Index) ;
	// Damage assessment functions
	// Assess_damage takes a pointer to a  function as a parameter
	// It sets the corresponding element of Checks if a problem is
	// detected by the function Test
	void Assess_damage ( void (*Test ) (boolean*)) ;
	boolean Eval_state (int Index) ;
	boolean  Is_damaged () ;
private:
	elem* Vals ;
	boolean* Checks ;
} ;
```


Recuperare da un fallimento

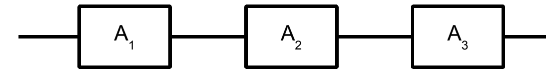
- Forward recovery
  - Riparare lo stato corrotto del sistema per trasformarlo in uno stato valido
  - Richiede conoscenze specifiche su cosa lo stato significa, e come può essere riparato.
    - Codici a correzione d'errore per individuare e correggere errori di trasmissione dati
    - Puntatori ridondanti (es. liste bidirezionali) per riparare filesystem danneggiati
- Backward recovery
  - Ripristinare lo stato del sistema in una condizione precedente stabile
  - E' molto più facile da implementare. Si eseguono snapshot periodici, e si rimpiazza lo stato corrotto con lo snapshot stabile più recente
  - L'idea delle transazioni nelle basi di dati

Analisi quantitativa dell'affidabilità

- Affidabilità = probabilità che il sistema funzioni correttamente in un dato istante
  - L'affidabilità (reliability) si indica spesso con *R*
- Sia *A* l'evento “il sistema funziona”
  $R = P(A)$ = P(“il sistema funziona”)

Sistemi in serie

- Il sistema funziona se tutte le componenti funzionano



- $R = P(A_1 \cap A_2 \cap A_3)$
  $= P(A_1) * P(A_2) * P(A_3)$
  $= \prod P(A_i)$

Sistemi in parallelo

- Il sistema funziona se almeno una delle componenti funziona
- R = P(“Almeno una delle comp. funziona”)
  = 1-P(“Nessuna comp. funziona”)
  = 1-P(“A1 non funziona”)*
  P(“A2 non funziona”)*
  P(“A3 non funziona”)
  $= 1-(1-P(A_1))*(1- P(A_2))*(1- P(A_3))$
  $= 1 - \prod (1-P(A_i))$