Affidabilità

Alberto Gianoli

Univ. Ferrara - Corso di Laurea in Informatica

Dove?

- Pressmann
- Sommerville: qualcosa dai cap. 9, 20, 23, 24

Affidabilità

- E' 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 pesano che esso fornisca i servizi che essi richiedono

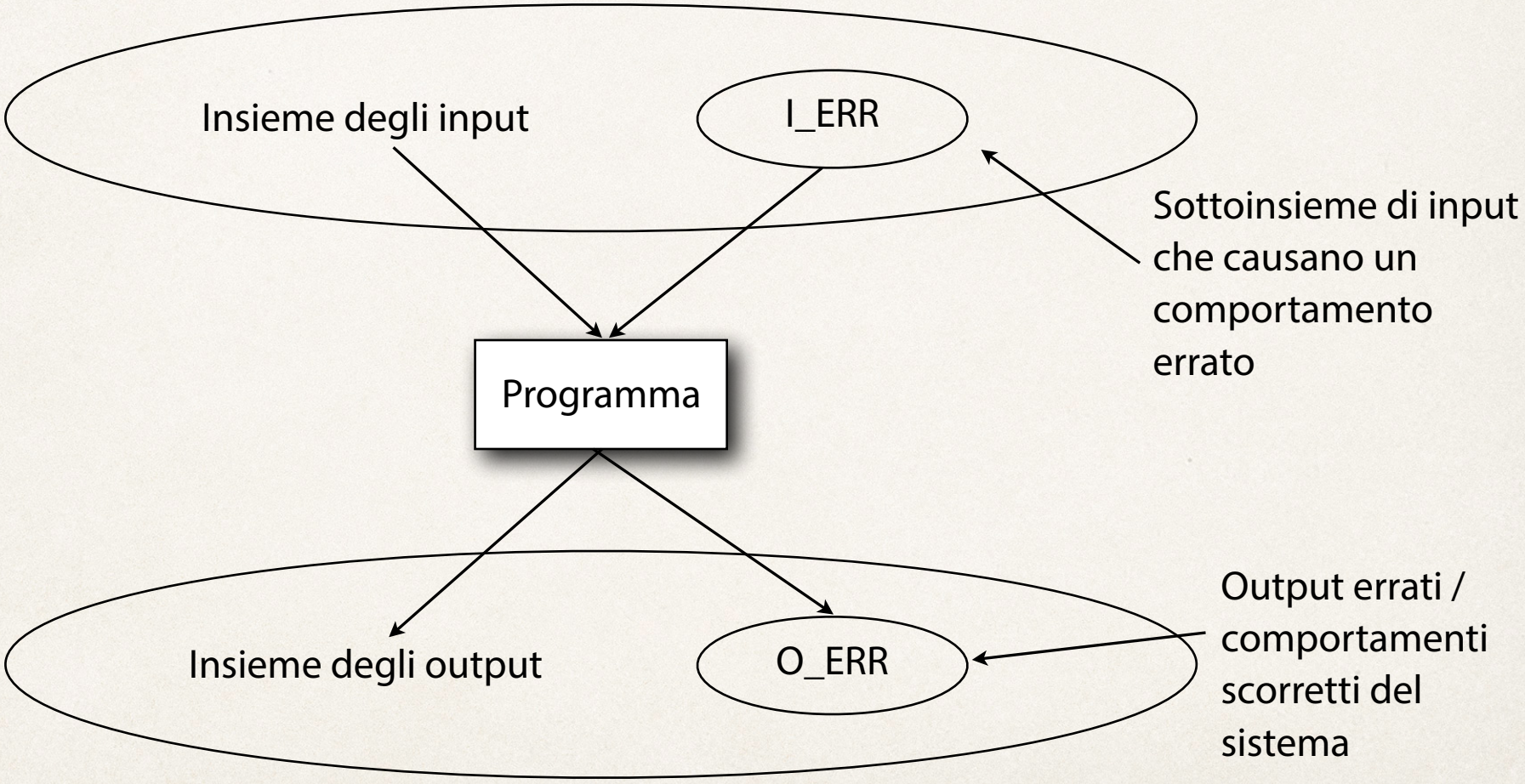
Affidabilità del sw

- Difficilmente si può definirla in modo oggettivo
  - 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 hw o che compie sempre lo stesso insieme limitato di azioni; non è significativo specificare l'affidabilità di un sistema interattivo che viene usato in modi diversi, a meno di non specificare questi modi
- Bisogna considerare le conseguenze dei fallimenti
  - non tutti i fallimenti sono ugualmente gravi: un sistema è percepito inaffidabile se si verifica un fallimento grave

Fallimenti e fault

- Un fallimento corrisponde a un comportamento a 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: solo quando la componente errata del sistema viene utilizzata

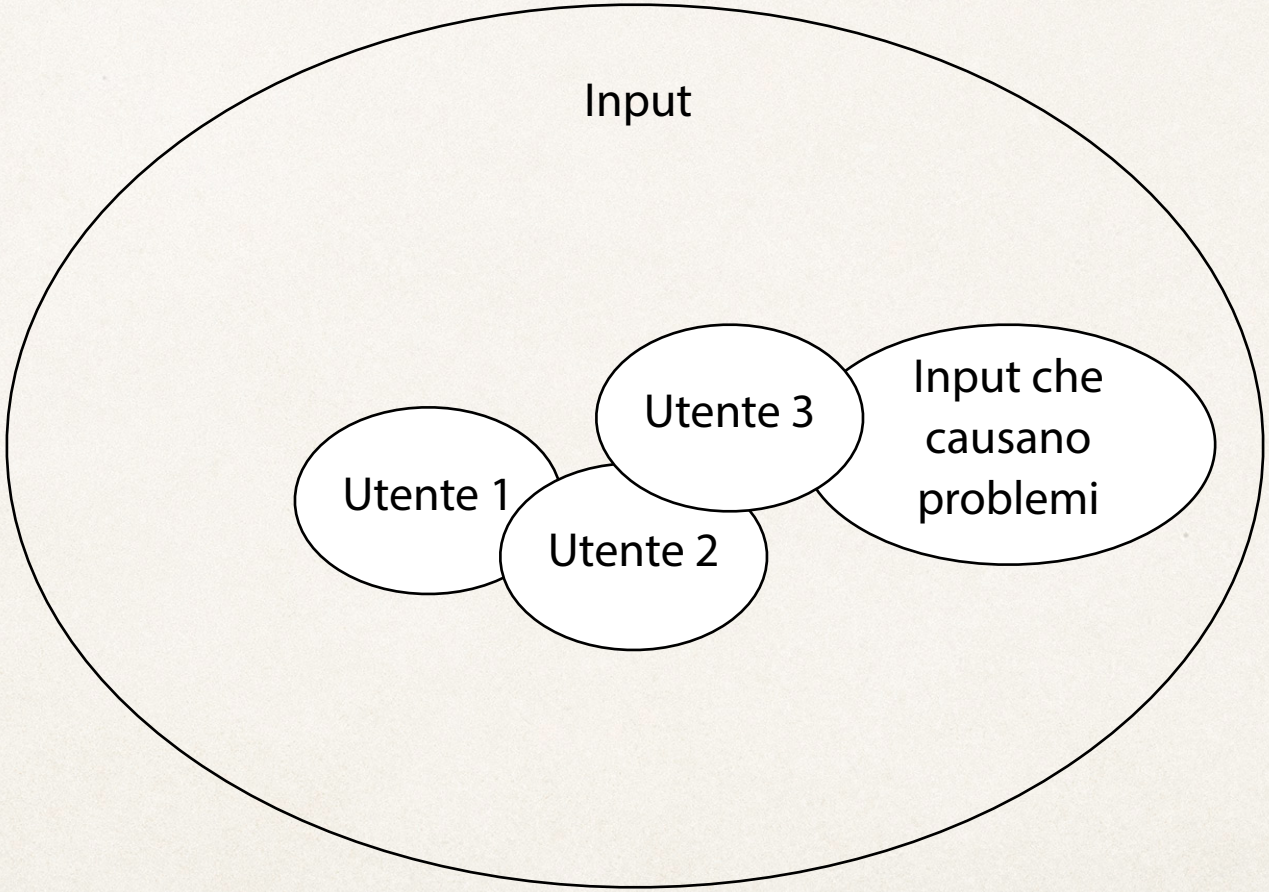
Mappatura input/output



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 affidabile, perché 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 si 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 usare del codice ridondante per effettuare controlli a tempo di esecuzione. Tutto questo può causare rallentamenti
- Però
  - spesso l'affidabilità ci interessa più dell'efficienza
  - computer più veloci aumentano le aspettative degli utenti in termini di affidabilità
  - sistemi inaffidabili molto semplicemente non vengono usati
  - sistemi inaffidabili possono essere difficili da migliorare
  - i costi connessi a perdite di dati sono molto alti

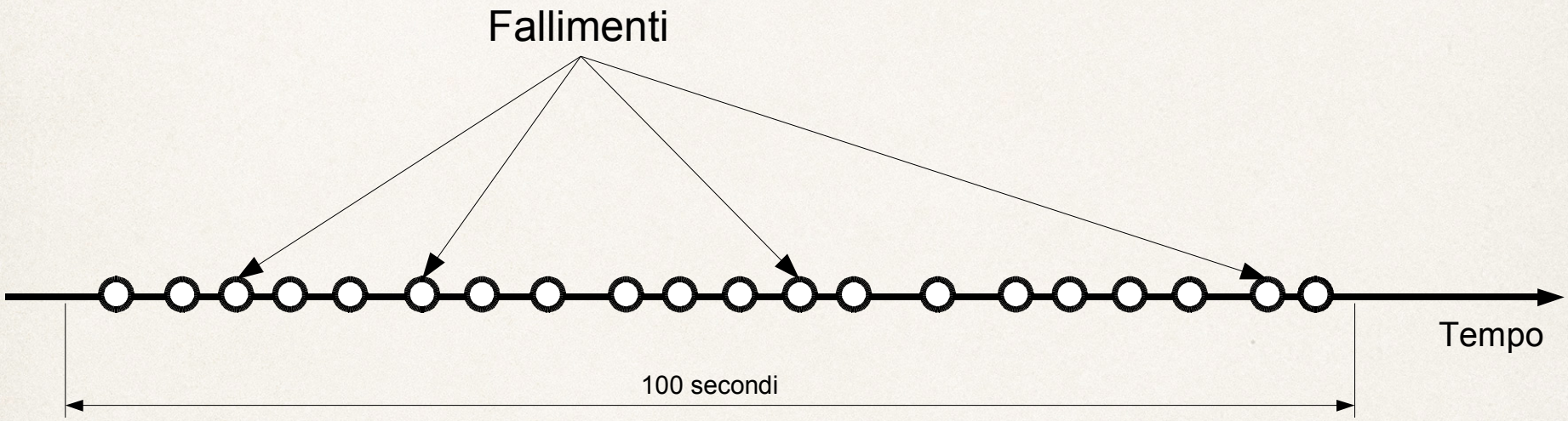
Metriche per l'affidabilità

- L'affidabilità dell'hw non è la stessa cosa dell'affidabilità del sw
  - componenti hw guaste possono essere riparati o sostituiti con altre componenti identiche; il disegno complessivo dell'hw solitamente è corretto, i difetti sono semplicemente causati da logorio
- Fallimenti del sw sono causati spesso da problemi di progettazione
  - in più, nel caso di alcuni fallimenti sw, il sistema continua a funzionare più o meno correttamente

Metriche

- *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 importante per sistemi critici o non stop
- *Rate of fault occurence* (ROCOF)
  Tasso di occorrenza dei fallimenti
  - frequenza con cui accadono comportamenti anomali
  - ROCOF=0.02 significa che ci si aspetta 2 fallimenti ogni 100 unità di tempo di funzionamento (occorre specificare l'unità di misura: giorni? anni?)
  - metrica importante per sistemi operativi

Esempio



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

Metriche

- *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

Esempio

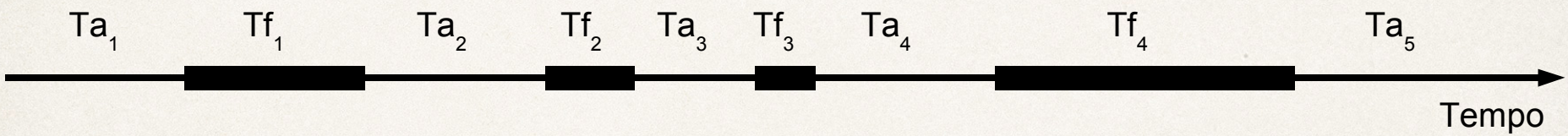
- $MTTF = \sum(Ta_i)/N$
- $AVAIL = \sum(Ta_i) \ / \ (\sum(Ta_i) + \sum(Tf_i))$

MTTF = MTBF? dipende dal tempo per riparare sistema

MTBF = MTTF + MTTR

mean time to repair

AVAIL = MTTF / (MTTF + MTTR)



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 raramente sono 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: in tutte le specifiche di affidabilità legate al codice sorgente non ha senso dire "non più di N fallimenti ogni 1000 righe 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 causa 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 ogni 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: sportello Bancomat

- Ogni sportello viene usato 300 volte al giorno
- La banca ha 1000 sportelli Bancomat
- Dopo 2 anni il sistema viene sostituito
- Ogni sportello gestisce circa 200000 transazioni durante la sua vita (300 transazioni/giorno x 365 giorni x 2 anni)
- Sono 300000 transazioni in totale (tutti gli sportelli) per ogni 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. Sw must be restarted to correct failure	ROCOF 1 occurrence / 1000 days
Transient, non-corrupting	The magnetic data cannot be read on an undamaged card which is input	POFOD 1 in 1000 transaction
Transient, corrupting	A pattern of transaction across the network causes database corruption	Unquantifiable! Should never happen in the lifetime of the system

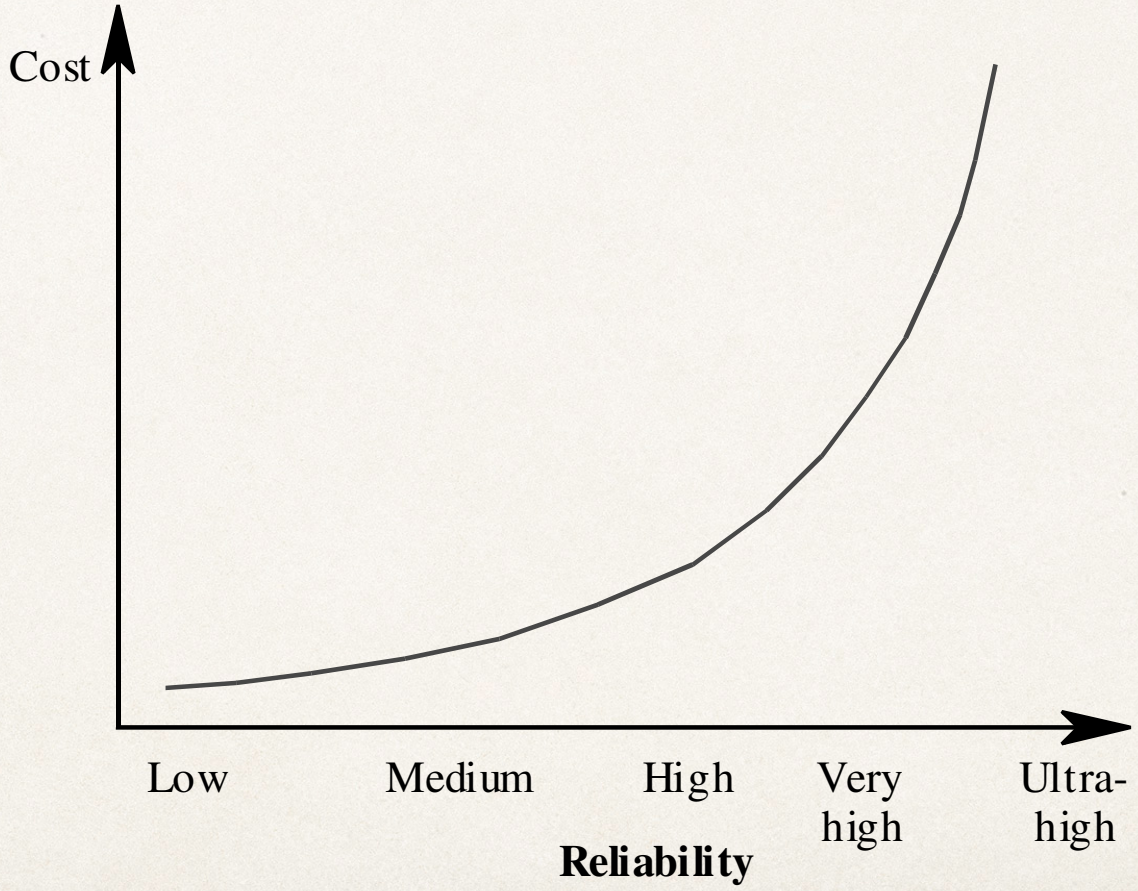
Validare le specifiche dell'affidabilità

- E' impossibile validare empiricamente specifiche di affidabilità molto stringenti
- "No database corruptions" significa (di solito) POFOD minore di 1 su 200 milioni
  - se per eseguire una transazione si impiega 1 sec, 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

- ….. quindi???
- Ottenere affidabilità molto alta è costoso. Spesso può essere più conveniente (dal punto di vista economico) accettare possibili fallimenti e pagarne le conseguenze
  - però…. se vi fate una reputazione di produrre sistemi inaffidabili, anche pagando le conseguenze, il mercato non vi cercherà
  - per certi tipi di sistemi (software per contabilità per esempio) una affidabilità modesta è adeguata

Costi per migliorare l'affidabilità



Test statistici

- Usiamo il numero di difetti che vengono scoperti durante la fase di test per fare una predizione sull'affidabilità
  - ma in questo caso i casi di provo devono essere scelti in accordo con il profilo operazionale del sistema, non per individuare il maggior numero di errori
- Bisogna specificare un livello di affidabilità accettabile: il software deve essere migliorato finché il livello di affidabilità desiderato viene raggiunto
- Vediamo come...

Procedura

1)Determinare il profilo operazionale del sistema

- cioè quale 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

- Un insieme di dati di input la cui frequenza si accorda con un uso normale del sistema
- Possono essere dati reali raccolti da un sistema esistente o (più spesso) dipendono da assunzioni fatte riguardo l'uso
- Se possibile dovrebbe essere generato automaticamente
  - può essere difficile per sistemi interattivi
  - può essere difficile per sistemi nuovi o innovativi: non si sa come verranno usati di preciso
  - più facile trovare gli input normali, meno facile individuare gli input improbabili

Difficoltà di generazione del profilo operazionale

- Incertezze intrinseche
  - soprattutto per sistemi nuovi, mai usati in passato; è un problema minore nel caso di sistemi che rimpiazzano altri obsoleti
- Costi elevati
  - dipende fortemente dal tipo di informazioni che sono raccolte e dal modo in cui sono raccolte
- Incertezza statistica quando si richiede alta affidabilità
  - è difficile stimare il livello di confidenza del profilo operazionale
  - i pattern di utilizzo del sistema possono cambiare col tempo

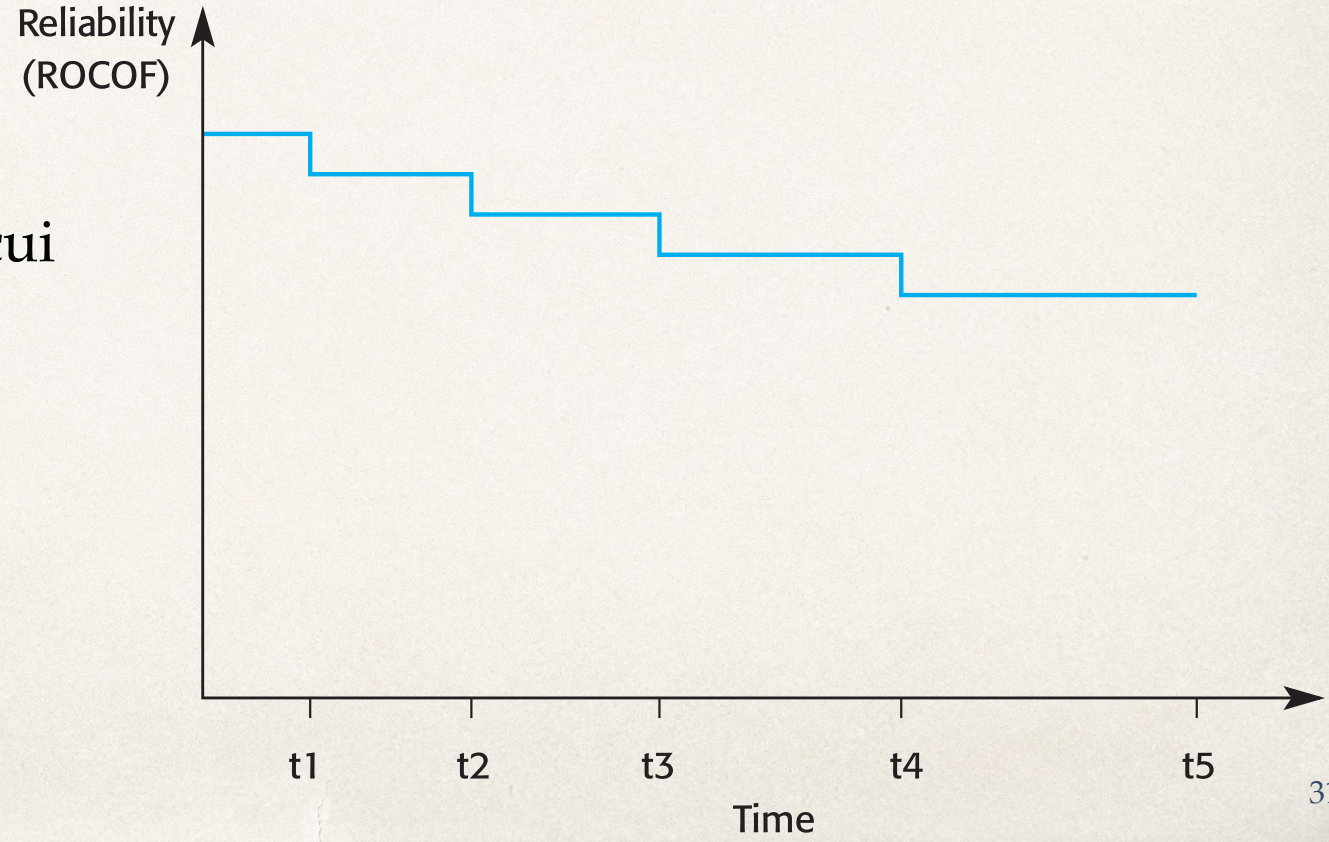
Modelli di crescita dell'affidabilità

- 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 attuali
  - occorrono strumenti statistici per far misure significative

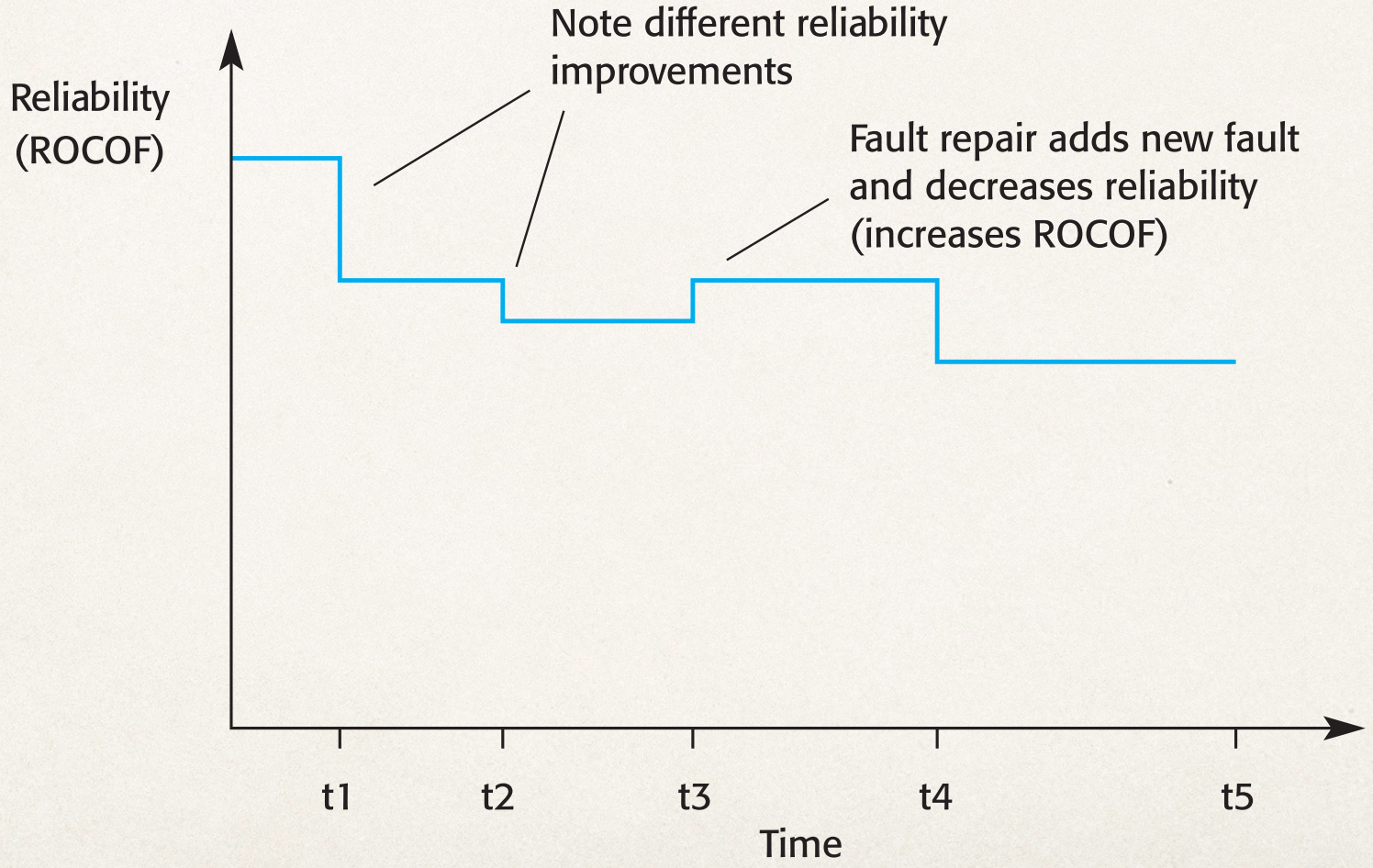
Modelli di crescita dell'affidabilità: la teoria

- Modello a passo lineare: semplice ma non rispecchia la realtà
- L'affidabilità non migliora necessariamente in quanto anche i cambiamenti possono introdurre nuovi errori

- La velocità di crescita diminuisce col tempo, come la frequenza con cui si trovano nuovi errori



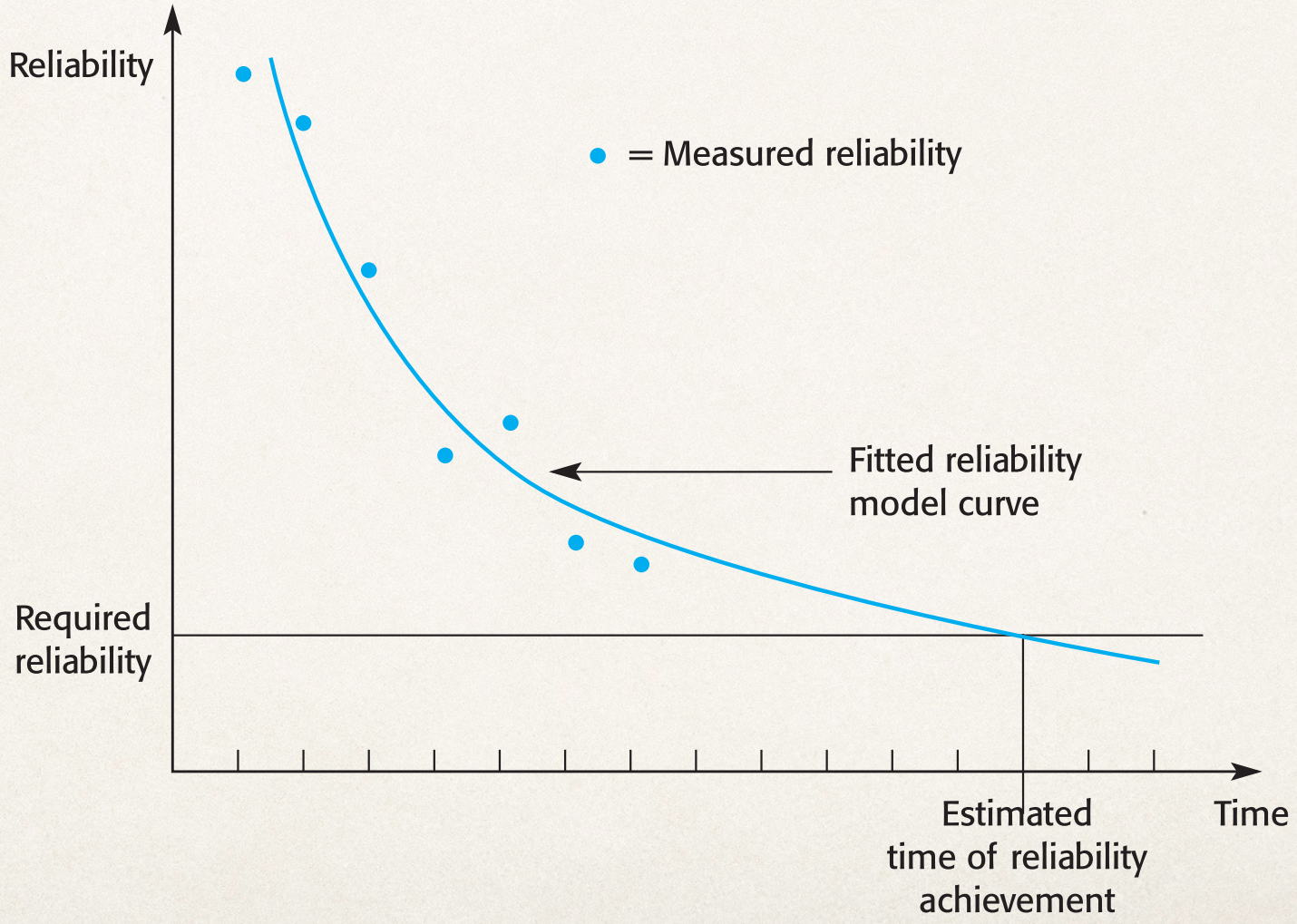
Modelli di crescita dell'affidabilità: la realtà



Quale modello usare

- Esistono molti modelli e nessuno è applicabile a una qualsiasi situazione
- L'affidabilità dovrebbe essere misurata e i dati osservati dovrebbero essere rapportati ai diversi modelli
- In questo modo il modello che si adatta meglio può essere usato per fare predizioni

Predizione dell'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

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

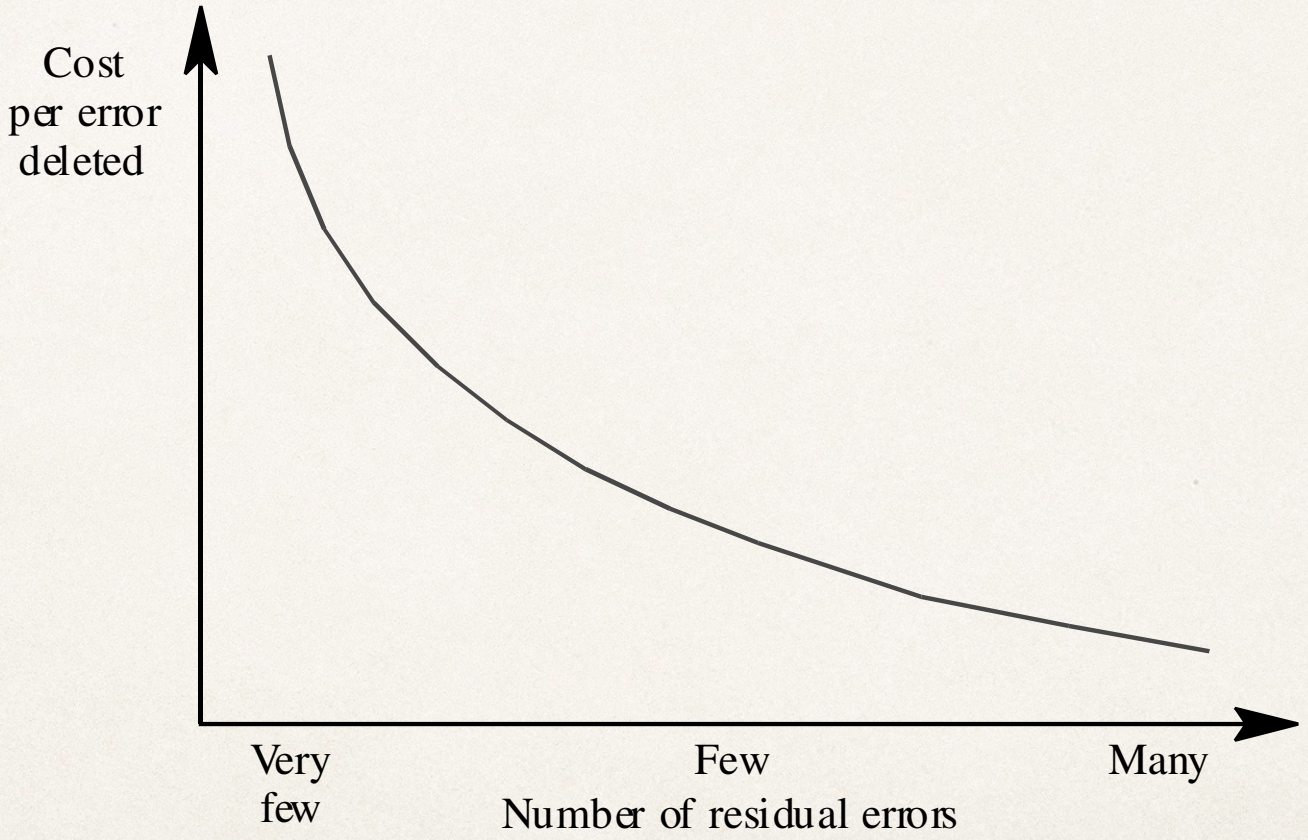
Come ottenere l'affidabilità

- 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 sw privo di errori è molto alto
  - conviene solo in alcune situazioni
  - può risultare economicamente più conveniente sopportare le conseguenze dei fallimenti

Costo di rimozione degli errori



Sviluppare sw privo di errori

- Partire da una specifica precisa, e possibilmente “formale”
- Utilizzare *information hiding* e *incapsulamento dell’informazione*
- Sfruttare linguaggi di programmazione con tipizzazione stretta e controlli a tempo di esecuzione
- Effettuare revisioni periodiche, in ogni fase del processo di sviluppo
- Chi produce il software deve essere interamente orientato alla qualità
- Test e collaudi accurati e completi sono indispensabili

Tecnica 1: programmazione strutturata

- Esiste dagli anni 70, abbiamo molta esperienza
- Non usare i `goto`
- `do-while` e `repeat-until` sono le uniche strutture iterative
- usare un approccio top-down
  - spezzare elaborazioni complicate in funzioni e procedure più semplici
- La programmazione strutturata è stata il primo reale contributo e la base di discussione sulla programmazione
  - non è solo un'arte, ma una disciplina ormai rigorosa

Punti problematici

- Numeri in virgola mobile
  - intrinsecamente imprecisi: errori di arrotondamento/approssimazione possono rendere i confronti non validi
  - se a e b sono double, NON dire if(a==b) {…
- Puntatori
  - possono puntare a memoria non allocata; puntatori diversi che referenziano la stessa area possono causare inconsistenze e rendere il programma illeggibile
- Allocazione dinamica della memoria
  - memory overflow
- Parallelismo
  - può causare computazioni errate o deadlock

Punti problematici

- Ricorsione
  - se sbagliate la condizione di terminazione avrete stack overflow
- Interrupts
  - rendono il programma difficile da capire perché sono equivalente a dei goto

- N.B.: non è che non dovete usarli, ma se li usate dovete essere molto attenti a cosa fate

Tecnica 2: information hiding

- Le informazioni devono essere rese disponibili solo alle parti del programma che ne hanno effettivamente bisogno
- si realizza incapsulando lo stato e le operazioni all'interno di oggetti
- Perché riduce le possibilità di fallimenti?
  - perché si riduce la probabilità di modificare "accidentalmente" le informazioni
  - è come mettere una cinta di mura attorno alle informazioni che impedisce il propagarsi dei problemi
  - dato che tutte le informazioni sono localizzate (alta coesione), il programmatore ha meno probabilità di commettere errori e il revisore ha maggiore probabilità di identificare gli errori

Oggetti e tipi di dati astratti (ADT)

- In Java, classi e oggetti
- Il nome del tipo di dato astratto (di solito) è il nome della classe
- Le operazioni sul ADT sono rappresentate dai metodi
- La rappresentazione dell'ADT è contenuta nella parte privata

Fault tolerance

- In situazioni critiche il sistema software deve tollerare i fallimenti
  - tollerare i fallimenti (fault tolerance) vuol dire che il sistema deve continuare a funzionare a dispetto di errori software
- Anche se in qualche modo viene dimostrato che il sistema è totalmente esente da errori (fault free), conviene comunque che sia anche fault tolerant
  - errori nelle specifiche
  - errori durante la fase di verifica
  - **N.B.** fault free == il programma è conforme alle specifiche

Come possiamo tollerare i fallimenti

- Rilevare il guasto
  - il sistema deve rendersi conto in qualche modo che è avvenuto un fallimento
- Valutare l'entità dei danni
  - bisogna rilevare quali parti del sistema sono affete 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 hanno bassissima probabilità di ripresentarsi; in questo caso potrebbe non essere conveniente riparare il guasto

Rilevare il guasto

- Di solito vuol dire accorgersi che siamo in uno stato del sistema "sbagliato", o che ci finiremo: di solito abbiamo dei valori di riferimento di alcune variabili per gli stati "giusti" e controlliamo lo stato attraverso queste variabili
- Rilevazione di guasto preventiva
  - la rilevazione comincia prima che il cambiamento di stato sia committed: se si rileva un errore, il cambiamento di stato non avviene
- Rilevazione di guasto retrospettiva
  - la rilevazione comincia dopo che il sistema ha cambiato stato; si usa se l'altro modo è troppo dispendioso o se è possibile che una sequenza incorretta di azioni corrette comporti uno stato sbagliato

Valutazione dei danni

- Analizzare il sistema per giudicare l'estensione della corruzione dei dati
- Viene fatto sulle parti del sistema che sono "fallite"
- Di solito si usano "funzioni di validità" che applicate ai vari elementi verificano se lo stato è consistente
  - checksums nella trasmissione dati
  - puntatori ridondanti per le strutture dati
  - watchdog timers per multiprogramming (essere sicuri che un certo processo risponda in tempo)

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"*: non vi ricorda nulla?
- Se non è possibile, il sistema può essere dinamicamente riconfigurato tramite sostituzione delle componenti difettose senza causarne l'interruzione

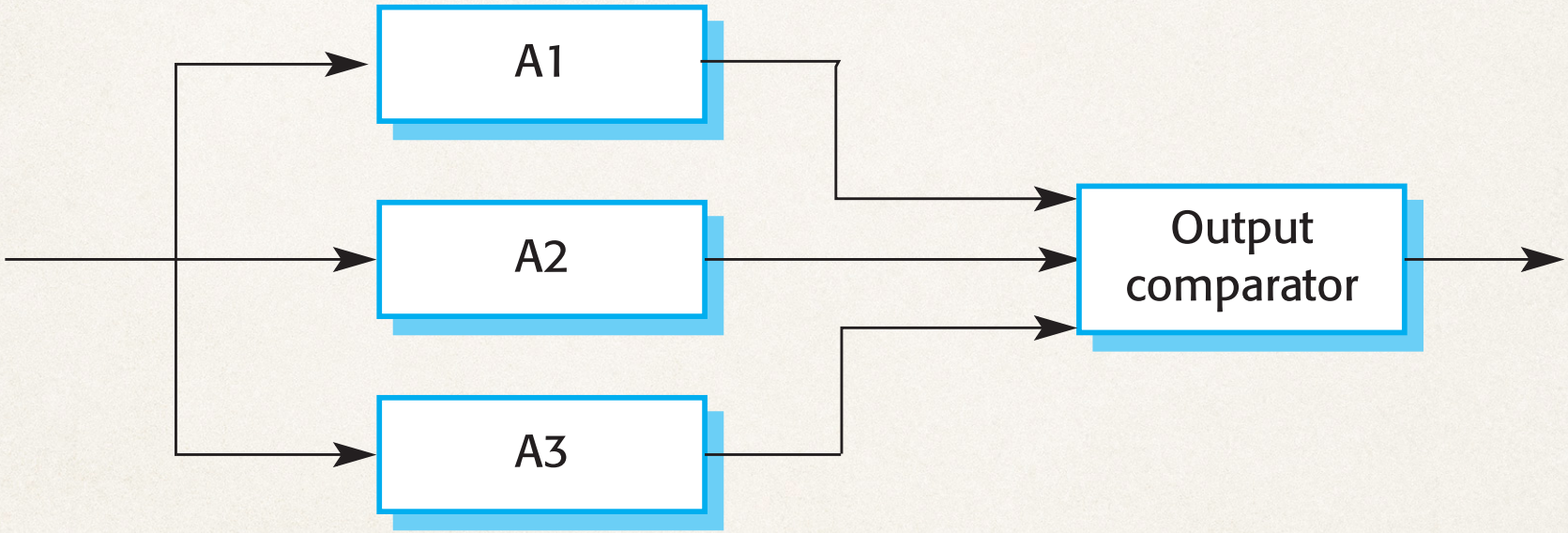
Recuperare dai fallimenti

- Recupero preventivo: solo se le informazioni di stato comprendono una ridondanza integrata
  - quando i dati codificati sono corrotti: una codifica ridondante permette di accorgersi degli errori e di ripararli
  - quando sono corrotte strutture collegate: una struttura dati con puntatori diretti e inversi può essere ricostruita se una quantità sufficiente di puntatori è intatta (vedi filesystem, db, ...)
- Recupero inverso: più semplice, ripristina lo stato sicuro dopo che è stato individuato un errore
  - p.e. le transazioni nei db contro modifiche non valide
  - nel caso di modifiche valide ma sbagliate si duplica periodicamente lo stato del sistema, in modo da poterlo ripristinare da una copia (checkpointing)

Tolleranza ai guasti per l'hardware

- Triple-modular redundancy (TMR)
- 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 viene dichiarata guasta
- Perché funziona?
  - si assume che le componenti hw falliscano causa usura, non perché progettate male
  - si assume che sia estremamente improbabile che più componenti falliscano contemporaneamente

Affidabilità dell'hardware mediante TMR



Architetture software fault tolerant

- Le due assunzioni fondamentali per il successo del TMR non si applicano al sw
  - non è possibile semplicemente replicare lo stesso modulo sw, in quanto entrambi avrebbero gli stessi fault di progettazione
  - questo vuol dire che inevitabilmente fallirebbero simultaneamente e allo stesso modo
- Per cui per il sw bisogna cambiare approccio

Diversità di progettazione

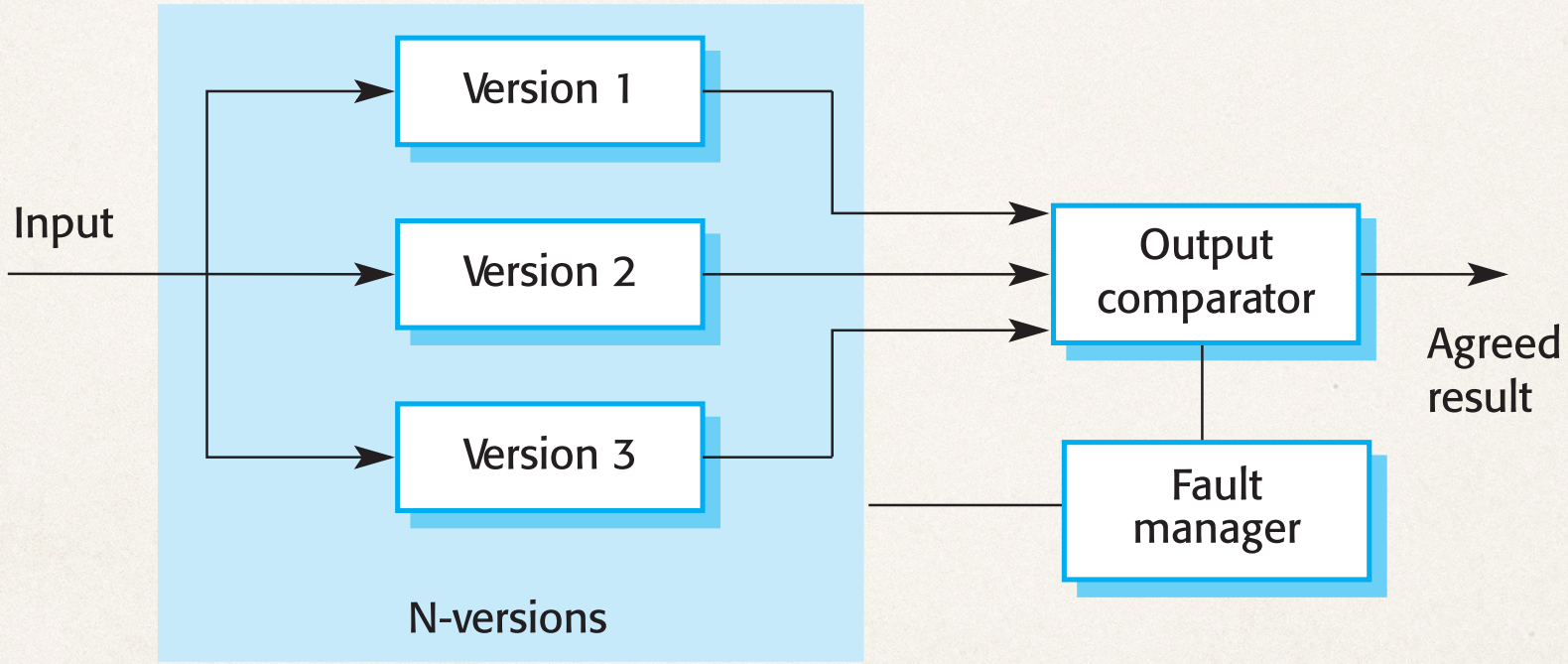
- Versioni diverse del sistema sono progettate e implementate in modi diversi. In questo modo avranno necessariamente modi diversi di fallire
- Approcci diversi alla progettazione (es. object oriented e procedurale)
  - implementazioni in linguaggi di programmazione diversi
  - uso di tools e ambienti di sviluppo diversi
  - uso di algoritmi diversi nell'implementazione

Analogie con il TMR

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 il più usato, per esempio in avionica (Airbus)
- Ovviamente se il problema sta nelle specifiche, non può farci nulla
- Le diverse versioni devono essere sviluppate e implementate da team diversi, per impedire che facciano gli stessi errori
- Però è empiricamente provato che anche team diversi possono interpretare nello stesso modo errato le specifiche; inoltre potrebbero utilizzare gli stessi algoritmi o strutture dati

Analogie con il TMR

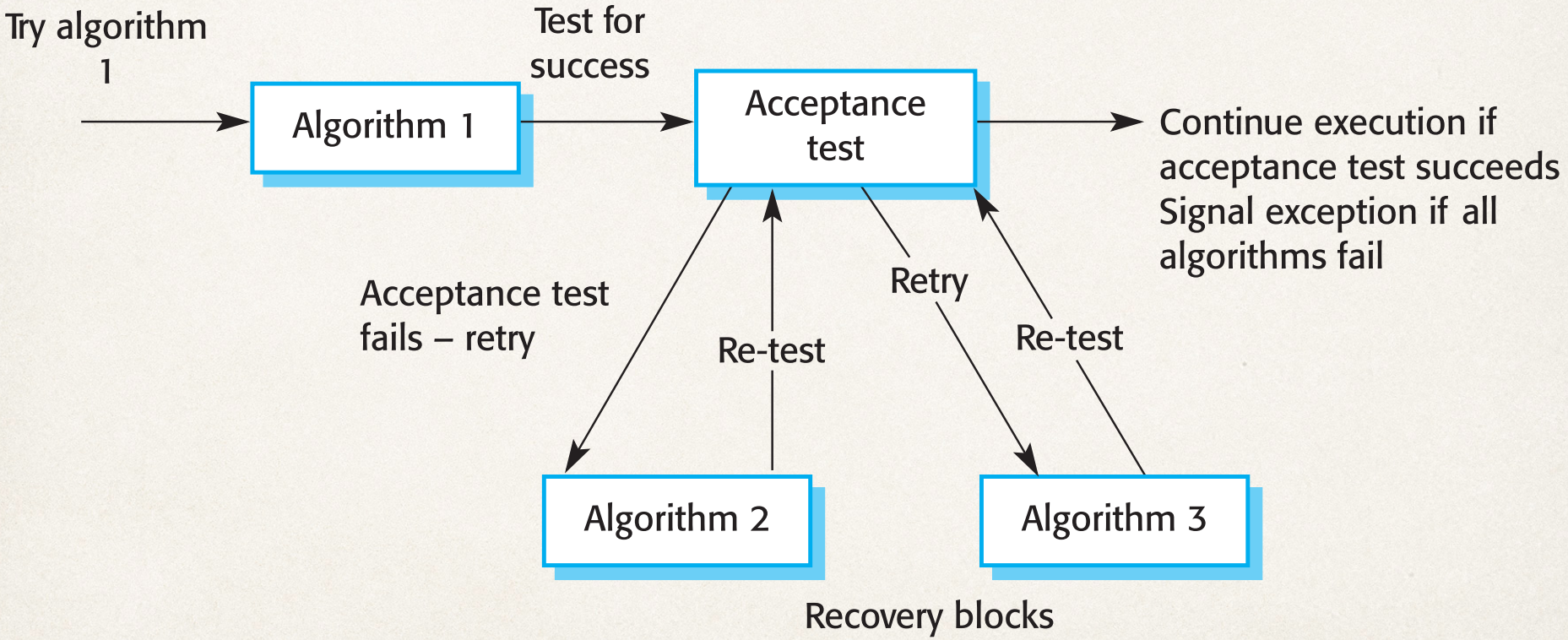


Analogie con il TMR

Recovery blocks

- Ogni componente ha un test per verificare se è stato eseguito con successo: in caso di fallimento un codice alternativo permette di ritornare indietro e di ripetere il calcolo; le implementazioni sono interpretazioni diverse della specifica e vengono eseguite in serie anziché in parallelo
- Gli algoritmi devono essere diversi a ogni tentativo per ridurre la possibilità di avere introdotto errori comuni
- Test di accettazione difficile da implementare
  - non bisogna ricalcolare il risultato da zero: siamo sicuri che il valore calcolato dal test sia giusto?
  - è opportuno che la verifica sia fatta in modo più efficiente rispetto al calcolo dello stesso
- Anche questo metodo dipende dalle correttezza delle specifiche

Analogie con il TMR



Problemi generali con la diversità di progettazione

- I team di sviluppo di solito non sono sufficientemente “diversi” culturalmente, e quindi tendono ad affrontare i problemi nello stesso modo
- Se c’è un errore nelle specifiche questo si rifletterà in tutte le implementazioni
- Per affrontare questo problema in alcuni casi vengono derivate indipendentemente varie specifiche del software partendo dalle stesse specifiche dell’utente
- … di solito evita errori nell’interpretazione ma non risolve il problema se le specifiche dell’utente sono sbagliate

Situazioni “a rischio di errori”

- Numeri floating-point
  - rischio errori di precisione che possono portare a errata comparazione di due valori
- Puntatori
  - se puntano alla zona di memoria sbagliata si può avere corruzione dei dati; e comunque gli alias possono rendere difficile la lettura (e la manutenzione) di un programma
- Allocazione dinamica della memoria
  - rischio memory overflow
- Parallelismo
  - interazioni non previste (e on volute) possono provocare errori random a run time
- Ricorsione
  - memory overflow

Situazioni “a rischio di errori”

- Interrupt
  - possono provocare la terminazione di una operazione critica e rendono il programma difficile da comprendere
- Ereditarietà
  - il codice non è localizzato: questo può risultare in comportamenti inaspettati quando si fanno modifiche e rende difficile la comprensione del codice
- Aliasing
  - usare più di un nome per riferirsi alla stessa variabile provoca confusione
- Unbounded arrays
  - controllare la dimensione degli array