Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica per il Management

TECNICHE PER LO SVILUPPO DI SISTEMI SOFTWARE AD ALTA AFFIDABILITÀ

Tesi di Laurea in Algoritmi e Strutture Dati

Relatore:
Chiar.mo Prof.
MORENO MARZOLLA

Presentata da:
CARLO BARALDI

Sessione II
Anno Accademico 2010/2011

Ai miei genitori, Nadia e Silvano.

Sommario

Al giorno d'oggi è sempre maggiore la richiesta di sistemi software affidabili, in grado cioè di adempiere alle proprie funzioni nel modo corretto anche qualora si dovessero verificare dei problemi interni o esterni al sistema. I sistemi informatici complessi sono sempre più chiamati a svolgere compiti altamente critici, che coinvolgono la sicurezza e l'incolumità delle persone. Si pensi per esempio ai sistemi di controllo di aerei, delle centrali nucleari, dei sistemi ferroviari o anche solo ai sistemi di transazioni monetarie. É importante per questi sistemi adottare tecniche in grado di mantenere il sistema in uno stato correttamente funzionante ed evitare che la fornitura del servizio possa essere interrotta dal verificarsi di errori. Tali tecniche possono essere implementate sia a livello hardware che a livello software. È tuttavia interessante notare la differenza che intercorre tra i due livelli: l'hardware può danneggiarsi e quindi la semplice ridondanza dei componenti è in grado di far fronte ad un eventuale malfunzionamento di uno di questi. Il software invece non può rompersi e quindi la semplice ridondanza dei moduli software non farebbe altro che replicare il problema. Il livello software necessita dunque di tecniche di tolleranza ai guasti basate su un qualche tipo di diversità nella realizzazione della ridondanza. Lo scopo di questa tesi è appunto quello di approfondire le varie tipologie di diversità utilizzabili a livello software e il funzionamento delle tecniche appartenenti a queste tipologie.
Nel Capitolo 1 introdurremo quindi il concetto di affidabilità, dandone una definizione generale e focalizzandoci sul significato di questa in riferimento al software. Nel Capitolo 2 tratteremo il concetto di Dependability, ovvero la capacità di un sistema di essere considerato attendibile, evidenziando quali sono le minacce che possono impedire l'ottenimento di tale capacità e quali

sono gli strumenti che permettono di conquistarla. Nel Capitolo 3 approfondiremo il concetto di tolleranza ai guasti, indicando i principi con cui viene realizzato, le strategie con cui cerca di rilevare i guasti (mantenendo la macchina in uno stato funzionante) e i problemi connessi all'uso di questo strumento. Nel Capitolo 4 presenteremo una rassegna delle tecniche utilizzabili per implementare la tolleranza ai guasti all'interno dei sistemi, dividendole per categorie ed indicando per ogni tecnica la descrizione del funzionamento, lo schema e lo pseudocodice che la rappresentano. Conclusa la rassegna delle tecniche, nel Capitolo 5 affronteremo il discorso dei costi, sia monetari che computazionali, che l'implementazione delle tecniche di tolleranza ai guasti comporta. Nel Capitolo 6 concluderemo il lavoro presentando, come caso di studio, la descrizione del sistema di controllo software del velivolo d'uso civile Airbus A-320.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione all'affidabilità

Al termine "affidabilità" possono essere dati molteplici significati [1]:

- Disciplina tecnico-scientifica, intesa come insieme di concetti, teorie e modelli che hanno lo scopo di determinare e descrivere il comportamento degli oggetti nel tempo.
- Attività pratico-organizzativa, volta ad ottenere l'affidabilità degli oggetti avvalendosi di tecniche specifiche e operando nelle varie fasi del ciclo di vita del prodotto.
- Proprietà di un oggetto, intesa come la capacità di questo di rispondere a specifiche richieste per un certo periodo di tempo e in determinate condizioni.

L'affidabilità acquisisce sempre più importanza con il passare degli anni esplicitandosi in numerose branche di applicazione che pur essendo molto diverse tra loro presentano comunque la comune concezione dell'affidabilità come qualità del sistema. Tale qualità rappresenta la misura della probabilità che l'oggetto considerato, indifferentemente che sia un sistema complesso o un semplice modulo software, non presenti deviazioni dal comportamento descritto nelle sue specifiche. L'affidabilità è quindi un disciplina ad ampio raggio, in continua evoluzione e strettamente collegata ad altri concetti qualitativi quali: manutenibilità, disponibilità e sicurezza.

1.1 Operazioni sull'affidabilità

Le operazioni principali che possono essere eseguite sull'affidabilità, intesa come qualità di un oggetto, sono: calcolare, verificare e costruire.
Calcolare e prevedere l'affidabilità è possibile attraverso l'uso di distribuzioni di probabilità. I metodi per il calcolo dell'affidabilità sono molteplici: si va dal più semplice metodo "combinatorio", che tuttavia è possibile solo se si riesce ad attribuire un valore di affidabilità ad ogni componente elementare dell'oggetto, a modelli più complessi che presentano una più ampia generalità e rendono possibile la misurazione qualora la complessità di calcolo divenisse proibitiva anche avvalendosi dell'utilizzo di calcolatori.
Verificare l'affidabilità del sistema è possibile tramite due approcci differenti: il primo si limita a registrare ed osservare il comportamento del sistema durante il suo funzionamento e più ampio sarà il numero delle osservazioni effettuate tanto più precisa sarà la stima ottenuta. È intuibile, tuttavia, come il risultato ottenuto da queste osservazioni comporti un certo grado di incertezza poiché si è vincolati ad un campione limitato di osservazioni. Il secondo approccio si basa sulla tecnica delle prove accelerate; prove in cui il sistema viene volutamente sollecitato e portato in condizioni di stress per osservare in un tempo minore la risposta del sistema a tali sollecitazioni.
La costruzione dell'affidabilità può essere realizzata sia durante la fase di progettazione del sistema, sia nella fase di produzione dello stesso, attraverso l'utilizzo di particolari tecniche volte al conferimento di un giusto grado di affidabilità agli oggetti trattati.
Queste tecniche possono essere o di valenza generale, come ad esempio l'utilizzo della ridondanza, oppure specifiche. E' importante sottolineare che il risultato di queste tecniche è trasformare il sistema in un progetto probabilistico dove le varie parti costituiscono le variabili aleatorie ed è quindi possibile calcolare il grado di incertezza e ridurlo.

1.2 L'affidabilità nel software

Il software può essere considerato come l'insieme di programmi e moduli composti a loro volta da un insiemi di istruzioni e dati, su cui si basano gli elaboratori per eseguire le operazioni. Il software è quindi un prodotto che, per raggiungere una certa qualità di realizzazione, può richiedere l'implementazione di determinate strutture atte a fornire al sistema la proprietà di affidabilità. In questo senso possiamo definire l'affidabilità software come:

> *La misura di quanto un sistema sia in grado di comportarsi secondo ciò che è stabilito nelle sue specifiche*[2].

Lo studio dell'affidabilità si propone quindi di descrivere e misurare la capacità del sistema di adempiere le funzioni e i compiti per i quali il sistema stesso è stato progettato e costruito; tale valutazione serve proprio a definire il grado di "fiducia" che possiamo avere nei confronti del "buon funzionamento" del sistema. Possiamo in tal senso ulteriormente definire l'affidabilità come*:*

> *La probabilità che il sistema svolga con continuità le funzioni richieste in determinate condizioni sia operative che ambientali.*

Tuttavia, data la variabilità e l'aleatorietà del sistema e dell'ambiente in cui opera, la corretta esecuzione delle funzioni richieste, a priori, non è totalmente prevedibile. Per poter discutere di affidabilità di un sistema in un'accezione più tecnica del termine, bisogna considerare sia le connessioni dei vari componenti che lo costituiscono (e relative competenze), sia il ruolo che il sistema ricopre nell'ambiente in cui è inserito; difatti, un malfunzionamento può scaturire sia da problemi interni, quali circostanze casuali o fenomeni di invecchiamento, sia da interferenze esterne, come sollecitazioni ambientali o passaggio di informazioni non prevedibili. Per tale motivo lo studio dell'affidabilità di un sistema è strettamente legato al calcolo dell'affidabilità dei suoi componenti e della loro interazione con l'ambiente esterno.

Inoltre la crescente complessità dell'architettura interna ed esterna dei sistemi software e il crescente numero di agenti con cui questi sistemi entrano

in rapporto inducono un aumento della probabilità di commettere errori durante la realizzazione del software; ed è oltre modo difficile, proprio a causa di tale aumento di complessità, verificare la presenza o meno di questi errori nell'applicativo. Più la complessità del sistema cresce più diviene facile commettere errori di realizzazione. Partendo dalla consapevolezza di quanto sia difficile ottenere un sistema totalmente privo di difetti, consideriamo allora "affidabile" non il software strettamente privo di difetti di realizzazione ma bensì il software che piuttosto tende a questo stato di perfezione, anche senza magari raggiungerlo pienamente, e che è in grado di procedere nel proprio funzionamento anche qualora si dovessero verificare degli errori sia di natura interna sia di natura ambientale. Questo è possibile grazie all'utilizzo di determinate tecniche di tolleranza ai guasti che tratteremo nel dettaglio nei prossimi capitoli.

Capitolo 2

Dependability

Negli ultimi decenni l'affidabilità dei sistemi informatici è diventata un aspetto sempre più importante, dibattuto in diversi ambiti e discipline. Questo in seguito al profondo inserimento dei sistemi informatici all'interno della nostra società e alla grande espansione a livello di utilizzo che hanno avuto. I sistemi informatici forniscono servizi sempre più sofisticati e per questo richiedono standard qualitativi sempre più elevati. Per questo motivo nel corso della produzione di un sistema software è importante focalizzarsi su quelli che sono gli aspetti che determinano il successo di un prodotto come ad esempio: le prestazioni, intese come tempo necessario ad eseguire le operazioni, o la robustezza, intesa come capacità del sistema di comportarsi ragionevolmente a fronte del verificarsi di situazioni impreviste.

In questo capitolo daremo quindi una definizione generale del concetto di Dependability, fondamentale per la comprensione e la trattazione dell'argomento "Tolleranza ai guasti" che vedremo nel dettaglio successivamente.

2.1 Il Concetto Generale di Dependability

Il concetto di Dependability[3] consta sostanzialmente di 3 parti (vedi Figura 2.1):

1. gli attributi (attributes).
2. le minacce (threats).
3. i mezzi con cui l'affidabilità del sistema è ottenuta (means).

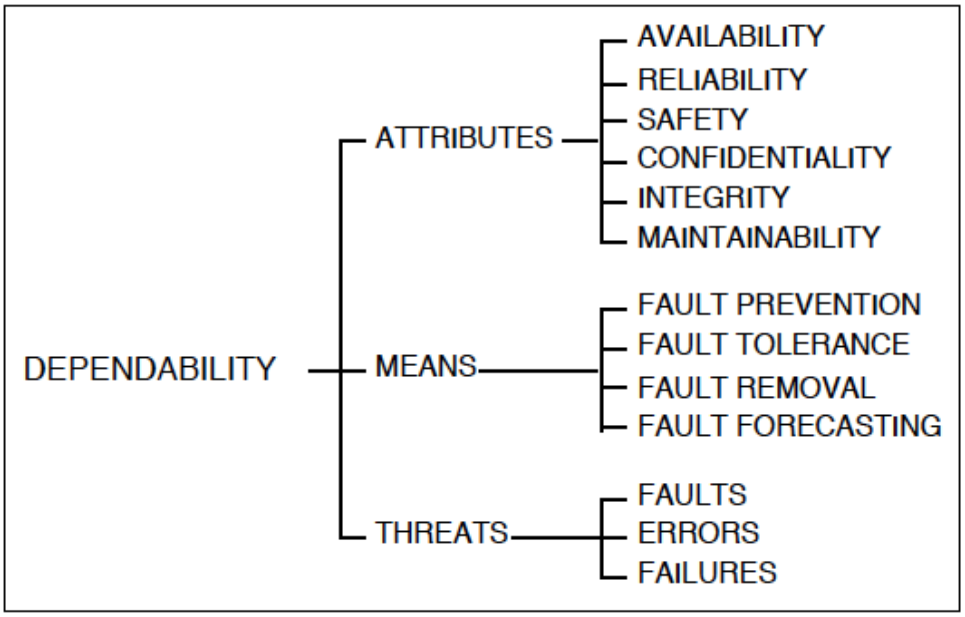

Figura 2.1: Schema ad albero del concetto di Dependability[3]

La Dependability di un sistema software viene definita dallo stesso autore Laprie come:

> *La capacità del sistema di fornire un servizio che può essere considerato attendibile dall'utilizzatore.*

La funzione che il sistema è chiamato a svolgere è definita dalle sue stesse specifiche e si dice quindi che il servizio fornito è corretto se il sistema rispetta

tali specifiche. Qualora il servizio offerto dal sistema si discosti dalle specifiche dello stesso vorrebbe dire che si è verificato un fallimento del sistema, con conseguente interruzione dell'attività. Vedremo nel dettaglio il significato e la natura del fenomeno del fallimento nel proseguo del capitolo. È stato necessario introdurlo per poter dare una ulteriore definizione del concetto di Dependability:

> *La Dependability è la capacità del sistema di evitare frequenti e gravi fallimenti che comporterebbero lunghe interruzioni del servizio rendendo il sistema stesso inutilizzabile dall'utente.*

2.2 Gli attributi della Dependability

La Dependability è un concetto che comprende all'interno del suo significato i seguenti attributi:

- Disponibilità (Availability): disponibilità del servizio corretto.
- Affidabilità (Reliability): continuità del servizio nel tempo.
- Confidenzialità (Confidentiality): riservatezza delle informazioni.
- Integrità (Integrity): assenza di alterazioni indesiderate dello stato del sistema.
- Sicurezza (Safety): assenza di gravi conseguenze sull'utente e l'ambiente.
- Manutenibilità (Mantainability): predisposizione a subire riparazioni e modifiche.

A seconda di quale sia l'intento che sta dietro la produzione del software è necessario dare maggiore enfasi ad un attributo piuttosto che ad un altro. Nell'assegnare i diversi pesi agli attributi si deve tener presente di alcune importanti proprietà: la *disponibilità* del servizio è sempre necessaria, mentre *l'affidabilità*, la *sicurezza* e la *riservatezza* sono necessarie in relazione alla funzione che il sistema è chiamato a svolgere.

Si può quindi notare come esista una stretta dipendenza tra i diversi attributi: *l'integrità* è un requisito indispensabile per la *disponibilità*, per *l'affidabilità* e per la *sicurezza*, mentre il requisito di *disponibilità* è correlato a quelli di *affidabilità* e *manutenibilità*. Vale la pena precisare la differenza tra i due attributi *disponibilità* ed *affidabilità*: il primo fa riferimento al fatto che il sistema è chiamato a svolgere le proprie funzioni per cui è stato realizzato fornendo il servizio in modo corretto, mentre il secondo riguarda la probabilità che un dispositivo assolva alla funzione richiesta per un tempo prefissato e in determinate condizioni. Da tenere in considerazione è il fatto che, al fine di rappresentare diversi aspetti del sistema in esame, possono essere definiti ulteriori attributi non presenti nella classificazione sopra riportata ottenibili dalla commistione degli attributi sopra citati.

2.2.1 Parametri di Misurazione degli Attributi

Vediamo ora alcuni parametri importanti nella misurazione degli attributi della Dependability:

- t è la variabile *tempo*.
- La disponibilità *A(t)* è la misura della continuità con cui un sistema svolge correttamente le specifiche richieste. Considerando la variabile tempo t possiamo quindi dire che la disponibilità rappresenta la probabilità che al tempo t il sistema sia funzionante.
- L'affidabilità *R(t)* è la probabilità che un dispositivo funzioni correttamente per un lasso di tempo t .
- La durata di vita del sistema *Tg (TTF, Time to Fault)*.
- La probabilità di guasto *F(t)* al tempo *t,* detta anche curva dell'inaffidabilità, rappresenta la funzione di ripartizione della variabile *Tg (tempo al guasto)*. Ricordiamo che la funzione di ripartizione, detta anche funzione di distribuzione cumulativa (CDF), di una variabile casuale X è la funzione che associa a ciascun valore di x la probabilità che la variabile X assuma valori minori o uguali ad x.

- Il tempo di riparazione *Tr (TTR, Time to Repair)* è la quantità di tempo che serve per ripristinare il sistema nel caso si verifichi un errore.

- M(t) rappresenta la funzione di ripartizione della variabile *Tr*.

- MTTR (Mean Time To Repair) è la media della variabile *Tr*, cioè il tempo medio di riparazione.

- MTTF (Mean Time To Fail) è il valore atteso della variabile aleatoria *Tg*.

- MTBF (Mean Time Between Failures) è il tempo medio tra due guasti consecutivi. Questo indice vale solo nel caso in cui si stiano studiando "unità riparabili" con la distribuzione dei tempi al guasto che quindi non varia dopo un intervento di ripristino.

In Figura 2.2 è possibile osservare l'andamento delle due curve F(t) e R(t). Si può notare che la funzione R(t) è monotona decrescente con R(0)=1 in t=0 (il dispositivo funziona) mentre R(t) $\rightarrow$0 per t$\rightarrow\infty$. Altra importante osservazione è la relazione che intercorre tra le due curve del grafico per cui R(t) + F(t) = 1. Risulta poi che la probabilità di guasto definita su un intervallo è data da R(t1) – R(t2). Nella Tabella 2.1 si riassumono invece le variabili Tg e Tr e le misure a loro connesse.
Possiamo quindi notare la differenza tra le funzioni disponibilità e affidabilità: la disponibilità *A(t)* rappresenta la probabilità di trovare il sistema funzionante in un dato istante di tempo, mentre l' affidabilità *R(t)* rappresenta la probabilità che nell'intervallo (0, t) il sistema sia attivo e funzionante. Concludendo, è importante notare che l'affidabilità, a differenza della disponibilità, può anche non esssere percepita dall'utente. Questo perché un sistema può avere un grado di affidabilità dei componenti basso pur possedendo un livello di disponibilità elevato, a patto che il MTTR sia sufficientemente ridotto.

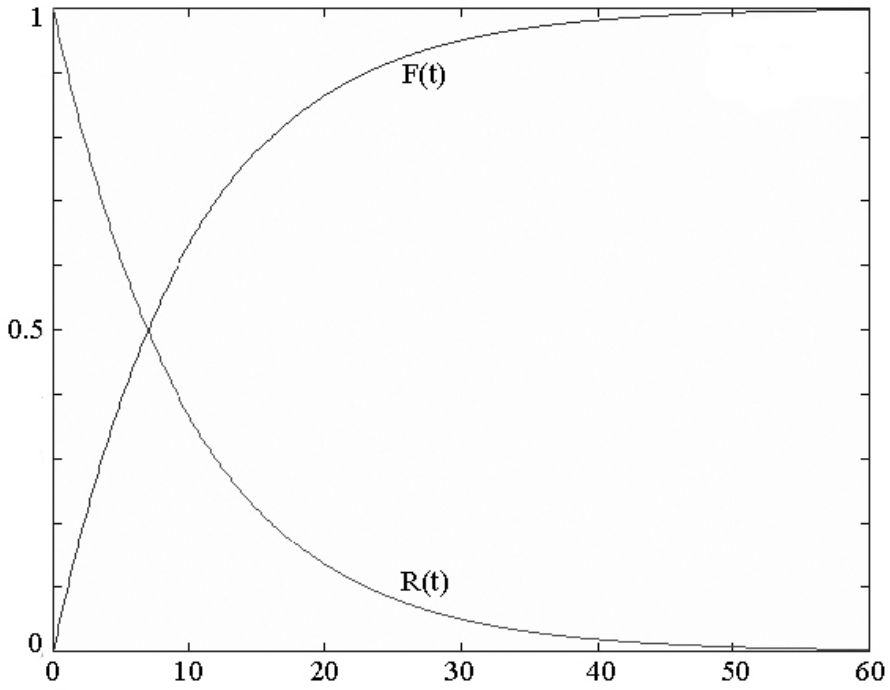

Figura 2.2: Le curve F(t) e R(t)

Variabile aleatoria	Descrizione	CDF	Media
Tg	Tempo al guasto	F(t)	MTTF
Tr	Tempo di riparazione	M(t)	MTTR

Tabella 2.1: Le variabili Tg e Tr

2.3 Le minacce alla Dependability

Il ciclo di vita del sistema può essere diviso sostanzialmente in due parti: la fase di sviluppo e la fase di utilizzo. La fase di sviluppo include tutte quelle attività che vanno dalla presentazione dell'idea iniziale del sistema alla decisione che il sistema sviluppato è pronto per essere messo in servizio. Durante tutta questa fase di progettazione può capitare che vengano inseriti all'interno del sistema dei guasti di progettazione (development fault). La fase di utilizzo inizia quando il sistema è messo in opera e l'utente finale può

beneficiare del servizio offerto. In questa fase gli stati in cui può venirsi a trovare il sistema sono tre: il primo consiste nel corretto funzionamento, il secondo identifica un fallimento del sistema con conseguente interruzione di servizio (outage), mentre il terzo consiste nello stop forzato del sistema da parte di un utente autorizzato (service shutdown). Vengono poi definite le operazione di ripristino (restoration), cioè l'attività di riportare il sistema ad uno stato funzionante in seguito ad un fallimento, e di manutenzione (maintenance) che comprende gli interventi di riparazione effettuati sul sistema e anche tutte quelle semplici modifiche che vengono apportate per migliorarne il servizio. La manutenzione può essere intrapresa sia successivamente alla fase di utilizzo, sia contemporaneamente.

Introduciamo ora tre importanti concetti: il *fallimento del sistema (failure)* che consiste nella non corrispondenza dei servizi offerti dal sistema con le specifiche definite in fase di sviluppo, l'*errore (error)* che è la parte dello stato del sistema che causa un fallimento e il *guasto (fault)* che è la causa dell'errore. Il guasto viene definito *attivo* se origina un errore, altrimenti *dormiente*.

2.3.1 Guasti

In ambito informatico il guasto viene definito come una condizione di anomalia del sistema hardware o software. Le cause che possono indurre un guasto possono avere diversa origine e natura: possono essere difetti di progettazione del sistema, oppure problemi di interazione del sistema con l'ambiente circostante, oppure ancora avarie della componentistica hardware. Relativamente ai guasti del sistema è possibile utilizzare la classificazione fornita da Avizienis [4], secondo il quale è possibile identificare 16 tipologie di guasti software. Queste 16 tipologie sono a loro volta classificate in 8 sottoclassi chiamate classi di guasto elementari (Figura 2.4).

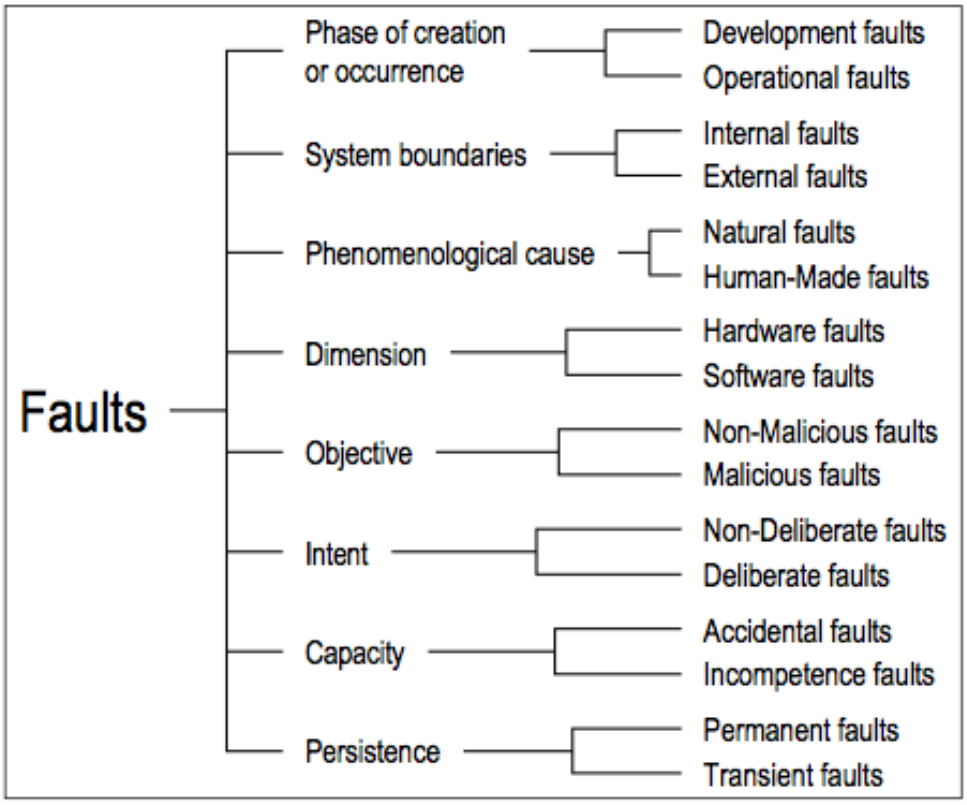

Figura 2.3: Classi di guasto elementari[4]

I criteri di classificazione sono i seguenti:

1. Il momento della vita del software in cui si origina il guasto.
   Guasti di progetto: guasti che vengono generati durante lo sviluppo del software o nella fase di manutenzione.
   Guasti operativi: guasti che vengono generati durante l'utilizzo del sistema.

2. La posizione del guasto.
   Guasti interni: guasti che vengono generati all'interno del sistema.
   Guasti esterni: guasti che vengono generati all'esterno del sistema e si propagano all'interno dello stesso causa interfacciamento o comunicazione.

3. La natura del guasto.
   Guasti naturali: guasti causati da fenomeni ambientali senza la responsabilità umana.
   Guasti causati dall'uomo: guasti causati dall'azione dell'uomo.

4. Il livello in cui il guasto si trova.
   Guasti hardware: guasti che affliggono l'hardware.
   Guasti software: guasti che affliggono il software, come ad esempio programmi o dati.

5. Il motivo della presenza del guasto.
   Guasti maliziosi: guasti introdotti nel sistema con l'intento di causare danni al sistema.
   Guasti non maliziosi: guasti introdotti nel sistema senza volontà.

6. L'intento di chi ha introdotto il guasto all'interno del sistema.
   Guasti intenzionali: guasti intenzionali inseriti con la volontà di arrecare danno.
   Guasti non intenzionali: guasti introdotti senza consapevolezza.

7. Le capacità della persona che ha introdotto l'errore.
   Guasti accidentali: guasti inseriti accidentalmente.
   Guasti dovuti ad incompetenza: guasti inseriti a causa dell'inadeguatezza ed incompetenza del programmatore autorizzato.

8. La persistenza nel tempo dell'errore.
   Guasti permanenti: guasti che si presumono essere costanti nel tempo.
   Guasti transienti: guasti la cui presenza è temporalmente limitata.

Combinando tutti i possibili criteri potremmo ottenere 256 diverse classi di guasto combinato. Tuttavia ciò non è possibile poiché alcuni criteri non sono compatibili tra loro; vengono quindi individuate come possibili 31 classi di guasto combinato (vedi Figura 2.4).

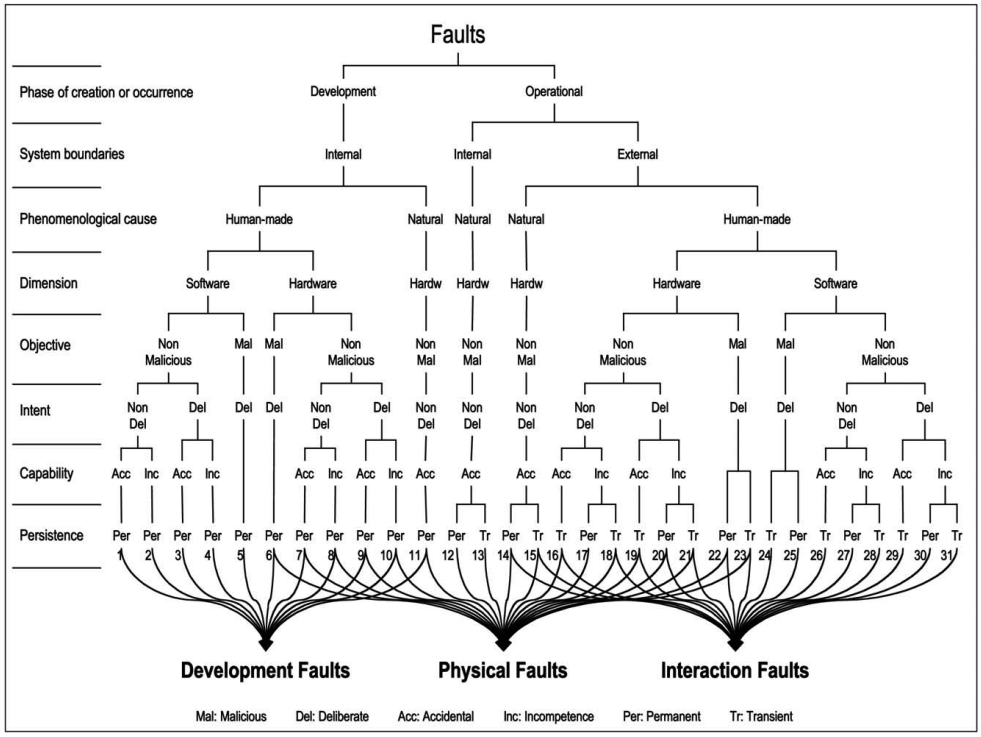

Figura 2.4: Classi di guasto combinato[4]

Le classi di guasto combinato vengono quindi riassunte in 3 grandi gruppi[5]:
Guasti di sviluppo che comprendono tutte quelle classi di guasto che si verificano in fase di sviluppo del sistema.
Guasti fisici che comprendono quelle classi di guasto che affliggono l' hardware.
Guasti di interazione che comprendono quelle classi di guasto relative guasti esterni al sistema.

I guasti possono poi essere classificati in base alla loro persistenza nel sistema:
Guasti permanenti: sono guasti che si verificano in modo continuato.
Guasti transienti: sono guasti che si verificano temporaneamente, in seguito ad una singola circostanza di errore, e non hanno bisogno di intervento di manutenzione poiché svaniscono da soli.

Guasti intermittenti: sono guasti che si verificano occasionalmente e in diversi intervalli di tempo.

La differenza sostanziale tra guasti intermittenti e transienti è che i primi solitamente sono problemi che affliggono l'hardware e che possono quindi essere facilmente risolti con la riparazione della parte fisica del sistema, mentre invece i guasti transienti, o temporanei, proprio per la loro non ripetitività, sono più difficili da individuare e per questo sono spesso causa di fallimento per un sistema software. É importante secondo Avizienis partire dal presupposto che le cause che portano ad un guasto interno al sistema software sono il più delle volte per loro natura permanenti quindi, per fare un esempio, un errore di progettazione dell'applicazione porterà sempre al medesimo guasto del sistema fino a quando non si interverrà con opportune modifiche.

Solid Fault – Elusive Fault

Un'ulteriore classificazione dei guasti può essere data in base al tempo di scoperta del guasto: la maggior parte degli errori insiti nel codice sono scoperti ed eliminati nelle prime fasi di realizzazione del programma. Questi tipi di guasti sono definiti *guasti solidi (hard faults*), e solitamente, proprio a causa della loro determinabilità e riproducibilità, sono eliminati prima della messa in opera del sistema. I guasti solidi vengono anche chiamati Bohrbugs, in riferimento al determinismo del modello atomico di Bohr[6].
Tuttavia, può succedere che il guasto si verifichi a causa di una combinazione di fattori tali che il suo accadimento possa essere definito "non prevedibile". Per questo motivo individuare il problema prima della messa in opera del sistema può diventare complicato e dubbio, ed è possibile che l'eliminazione del difetto non avvenga. Questi errori vengono chiamati *guasti sfuggenti (elusive faults*), oppure "Heisenbugs", dal principio di indeterminazione di Heisenberg; in pratica se si cerca di trovare un guasto col un tradizionale debugger, lo strumento altererà l'ambiente a tal punto da non far presentare il guasto.

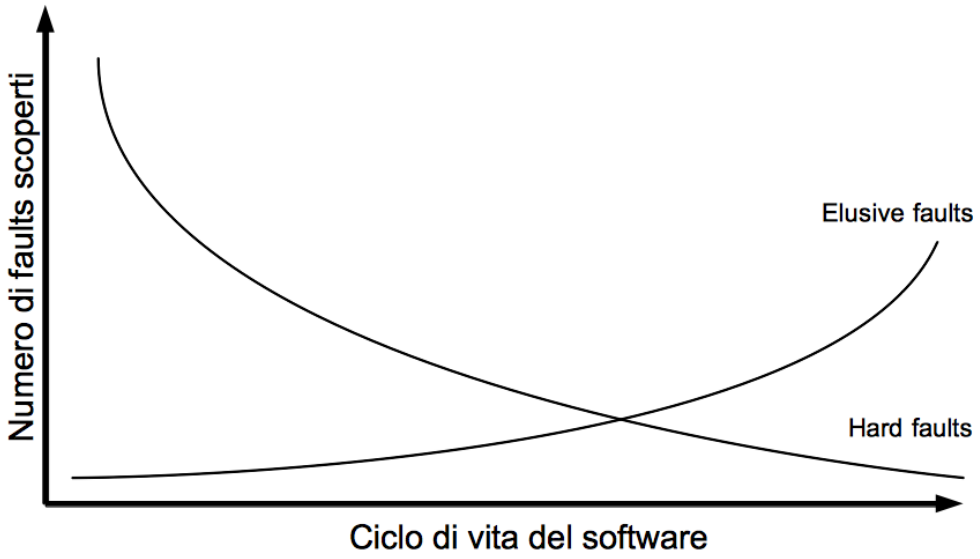

Figura 2.5: Curve dei guasti solidi e dei guasti sfuggenti.

Nella Figura 2.5 è possibile osservare l'andamento delle due curve che rappresentano la quantità di guasti scoperti nelle varie fasi di progettazione del software. Come si può vedere dal grafico i *guasti solidi* hanno un andamento decrescente. Questo perché la loro possibile identificazione fa sì che vengano corretti sia nelle prime fasi di progettazione che nelle fasi di testing pre-rilascio. Al contrario la curva che rappresenta l'andamento dei *guasti sfuggenti* è una curva crescente con il massimo raggiunto nella fase di utilizzo del sistema; questo poiché è proprio in questa fase che il software giunge al massimo della complessità e viene fatto lavorare in tutte le possibili condizioni, dando magari origine ad una contingenza di fattori che provocano l'insorgenza dell'errore.

È intuibile come sia più facilmente possibile porre rimedio ai *guasti solidi* piuttosto che ai *guasti sfuggenti*. Questo proprio perché i *guasti solidi* sono caratterizzati da una natura determinabile che li rende identificabili ed eliminabili, mentre i *guasti sfuggenti* tendono maggiormente a passare inosservati o peggio ancora ad insorgere in un momento successivo al rilascio del

software.

2.3.2 Fallimento

Un *fallimento* è definito come l'evento che si verifica quando il sistema fornisce un servizio non in linea con quanto richiestogli. I fallimenti possono verificarsi con diverse modalità e differiscono tra loro per la gravità del disservizio arrecato. Le tipologie di fallimento sono rappresentate in Figura 2.6. Ciò che è importante tenere presente è che un fallimento va sempre rapportato con quella che è la funzione del sistema e non con quanto scritto nelle specifiche poiché potrebbe accadere che il sistema, pur soddisfacendo le specifiche iniziali, non soddisfi la volontà del cliente. Questo significa che vi è stato un errore nella stesura delle specifiche di progetto e tale errore è riconoscibile solo dopo la sua comparsa, quindi a sistema già in funzione.

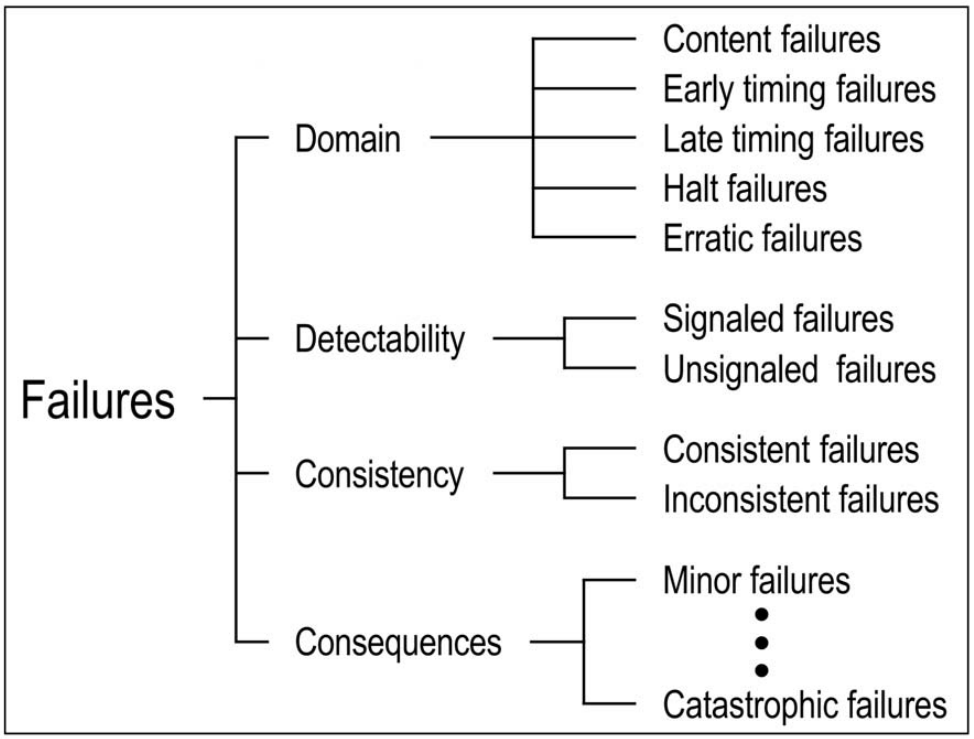

Figura 2.6: Schema delle tipologie di fallimento[4]

Le varie modalità con cui un sistema può incorrere in un fallimento sono detti modi di fallimento e vengono categorizzati secondo quattro punti di vista:

- **Dominio (Domain)**: Bisogna distinguere il caso in cui il fallimento sia dovuto al fatto che le informazioni inviate all'interfaccia del sistema si discostano dalla normale funzione del sistema, oppure al fatto che il tempo di arrivo o di durata dell'erogazione delle informazioni si discosta da quanto definito nelle specifiche. Quando il fallimento del sistema è causato da entrambi i casi allora il sistema può cadere in uno stato di fallimento dove si verifica un blocco totale del sistema che non ritorna alcuna informazione all'interfaccia. Nello stato di blocco totale può anche accadere che alcune informazioni vengano inviate all'interfaccia ma queste sono comunque erronee.

- **Controllabilità (Detectability)**: si riferisce alla proprietà del sistema di segnalare all'utente lo stato di errore. A volte i sistemi sono progettati per fallire seguendo modalità precise. Questo avviene tramite meccanismi di rilevazione della correttezza del servizio erogato. In questo modo vengono generati fallimenti controllati, detti *fallimenti segnalati*, che il sistema controlla attuando contromisure come può essere il riavvio controllato del sistema.

- **Consistenza (Consistency)**: Quando il sistema ha due o più utenti definiamo il fallimento consistente se tutti gli utenti hanno la medesima percezione dell'errore, altrimenti incosistente se qualche utente percepisce diversamente la scorrettezza del servizio.

- **Conseguenze (Conseguences)**: i fallimenti vengono classificati secondo la gravità delle conseguenze, questa classificazione prevede una suddivisione sostanziale tra *fallimenti minori,* che hanno conseguenze accettabili e i costi di manutenzione minimi, e *fallimenti catastrofici*, dove il costo delle conseguenze del fallimento è di gran lunga maggiore rispetto al guadagno che si trarrebbe dall'erogazione corretta del servizio.

2.3.3 Errori

Si definisce l'errore come la parte dello stato totale del sistema che può portare ad un fallimento, questo avviene quando l'errore è in grado di discostare il servizio del sistema dal corretto funzionamento. Con "stato totale del sistema" si intende l'insieme di tutti gli stati dei componenti che combinati identificano il sistema totale.
Questo fa sì che l'errore inizialmente sia limitato ad un singolo componente (o più di uno) e che questo si traduca in fallimento del sistema solo quando questo errore va a manifestarsi non più solo a livello di singola componente ma dell'intero sistema. Fino a questo momento infatti l'errore rimane limitato allo stato del componente interno e non influenza lo stato totale del sistema.
La causa di un errore è ciò che in precedenza abbiamo definito guasto. Un errore si dice *errore rilevato* se questo viene segnalato da un messaggio di errore o da qualsiasi altra forma di segnalazione, altriementi viene detto *errore latente*. Non abbiamo una classificazione propria degli errori, questo perché si usa descriverli sulla base della classificazione delle tipologie di fallimento. Una eventuale classificazione può essere data in base alla tipologia del danno causato: se singolo, per esempio, nel senso che affligge un solo componente oppure se multiplo, come può accadere in seguito ad una scarica elettromagnetica in grado di coinvolgere più componenti contemporaneamente.

È possibile evitare che un errore porti ad un fallimento implementando nel sistema una qualche forma di ridondanza. Vedremo nel dettaglio queste tecniche nel quarto capitolo. É altresì possibile non arrivare ad un fallimento qualora la parte dello stato affetta da errore non venga mai utilizzata dal sistema o qualora questa venga corretta prima della sua attivazione.

2.3.4 Propagazione del Guasto

Dopo aver visto cosa sono i guasti, gli errori e i fallimenti, soffermiamoci ora meglio sulla relazione che lega queste tre minacce per il sistema software. Tale relazione è detta *processo di propagazione* e permette che la manifesta-

zione del guasto raggiunga l'interfaccia utente. Vediamo ora i passaggi di tale processo:

1. Un guasto come detto è attivo quando da origine ad un errore, altrimenti è detto inattivo. Un guasto attivo può essere un guasto già insito nel sistema che viene attivato dal processo di calcolo, oppure da un sistema esterno che interfacciandosi col sistema in esame fa sì che il guasto inattivo si attivi.

2. La propagazione dell'errore all'interno del componente è causata dal processo di calcolo del sistema. In questo modo il sistema transita da uno stato di corretto funzionamento ad uno stato improprio, che identifichiamo con l'errore, e ricorsivamente questo errore può essere trasformato in altri errori. Questo è reso possibile dal fatto che l'errore interno ad un componente, raggiungendo l'interfaccia di questo, può propagarsi agli altri componenti connessi che usufruiscono del servizio. Se invece l'errore non si propaga ad altri componenti allora è detto latente.

3. La propagazione dell'errore tra i vari componenti fa sì che il servizio devi dal normale funzionamento e non risulti più corretto, generando un fallimento. Un fallimento interno ad un componente può propagarsi ad altri componenti e causare quindi un fallimento a livello di sistema.

Risulta in questo modo chiaro il nesso causale che esiste tra i tre concetti. Ciò che dobbiamo tenere presente è come il fallimento di anche solo una parte dello stato del sistema possa causarne, attraverso la propagazione, il totale fallimento. La Figura 2.7 illustra i passaggi del processo di propagazione del guasto.

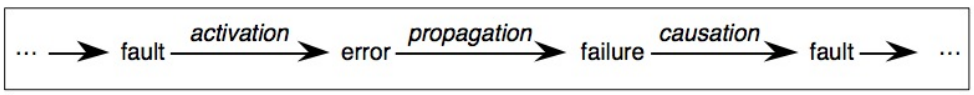

Figura 2.7: Processo di propagazione del guasto[3]

2.4 I mezzi per ottenere la Dependability

Un sistema può innanzi tutto essere classificato sulla base del fatto che sia o meno tollerante ai guasti. Identifichiamo quindi due classificazioni: *sistemi tolleranti ai guasti* e i *sistemi non tolleranti ai guasti*.
I sistemi non tolleranti ai guasti sono quei sistemi dove si cerca esclusivamente di rendere meno probabile possibile il verificarsi dei fallimenti. Si opera sia a livello hardware che software. Sostanzialmente a livello hardware si cercherà di utilizzare componenti altamente affidabili, mentre a livello software si tenderà piuttosto a soffermarsi sulle fasi di testing agevolando l'individuazione dei guasti. Avendo già accennato a quanto sia difficile evitare che un sistema possa essere distribuito con la certezza di essere privo di guasti, possiamo immaginare come questi sistemi, pur avendo superato tutte le fasi di testing, non possano considerarsi assolutamente immuni da fallimenti. Questo poiché, non essendo il sistema tollerante ai guasti, l'attivarsi anche di un solo guasto potrebbe causare il fallimento di tutto il sistema.
I sistemi tolleranti ai guasti sono invece sistemi pensati e progettati al fine, più che di evitare l'insorgenza di errori, di ridurre e contenere le conseguenze del verificarsi di un errore. In questo modo è possibile evitare che il sistema pur essendo affetto da guasti degeneri in fallimento, finendo per diventare inutilizzabile.

Dire che questi sistemi devono essere progettati in un determinato modo fa capire come la gestione del processo di sviluppo software rivesta un ruolo importante per raggiungere l'obbiettivo di costruire sistemi affidabili. Il *processo di sviluppo software* è identificato da una serie di passi che bisogna necessariamente svolgere per ottenere risultati di qualità. Le attività fondamentali che compongono lo sviluppo del software sono:

1. *Analisi*. Consiste nello studio preliminare del prodotto che deve essere realizzato, nel senso che deve essere definito il problema che si vuole risolvere. Questa attività può essere scomposta in sotto-attività tra cui l'analisi dei requisiti del prodotto. Il documento risultante da questa

attività è il cosiddetto *documento di specifica*, contenente appunto le specifiche del prodotto software.

2. *Progettazione.* Consiste nella definizione della struttura generale del prodotto software sulla base dei requisiti espressi nella fase precedente. Se nella fase precedente si è definito il problema da risolvere, in questa fase viene invece definito come il problema deve essere risolto.

3. *Implementazione.* Consiste nella realizzazione vera e propria del prodotto software sulla base della struttura definita nel passo precedente. All'interno di questa attività vengono poi definite alcune sotto-attività come: lo sviluppo separato dei moduli che compongono il software e l'integrazione di questi nel medesimo prodotto.

4. *Collaudo.* Consiste nel verificare quanto il software realizzato rispetti le specifiche definite nell'attività di *Analisi*. Anche in questo caso l'attività può essere scomposta nell'attività di verifica dei singoli moduli e nella successiva verifica dell'interazione tra questi nel programma completo. É principalmente in questa fase che si tenta di individuare e risolvere i guasti inseriti nel programma al momento dell'implementazione. L'attività di collaudo termina con la validazione del prodotto.

5. *Rilascio.* Consiste nel rilascio del programma una volta che è stato validato e nella successiva messa in opera.

6. *Manutenzione.* Consiste nelle operazioni di modifica al programma apportate successivamente al rilascio dello stesso. Tali operazioni sono volte a correggere errori individuati nel codice o ad estendere le funzionalità del sistema.

L'organizzazione di queste attività identifica i vari modelli di sviluppo software. Lo studio e la progettazione di questi modelli è uno degli obbiettivi dell'Ingegneria del Software. Alla base di questa disciplina c'è la considerazione del software come risultato di un processo di sviluppo suddiviso in diverse attività. L'organizzazione di tali attività identifica il ciclo di vita del

software.

Sulla base degli approcci di progettazione del sistema vengono quindi identificate dallo stesso Laprie [3] quattro metodologie volte ad aumentare la Dependability di un sistema e che si distinguono tra loro per la fase del ciclo di vita del software in cui vengono adottate:

- **Prevenzione dei guasti (fault prevention)**: cerca di prevenire l'inserimento dei guasti nel sistema e viene quindi utilizzata nelle attività di analisi e progettazione.
- **Tolleranza ai guasti (fault tolerance)**: mantiene il sistema funzionante anche in presenza di guasti e deve essere adottata nelle fasi di progettazione e implementazione del programma.
- **Rimozione dei guasti (fault removal)**: cerca di eliminare il più possibile i guasti presenti nel sistema e viene utilizzata nelle attività di collaudo e manutenzione del software.
- **Previsione dei guasti (fault forecasting)**: cerca di stimare la presenza dei guasti all'interno del sistema.

Le prime tre metodologie cercano di conferire al sistema un certo grado di affidabilità, mentre l'ultima è volta a stimare l'affidabilità del sistema. La scelta di una di queste tecniche non esclude la possibilità di implementare anche le altre. Inoltre la scelta delle metodologie influenza inevitabilmente il livello di Dependability del sistema risultante, e viceversa le aspettative che si hanno sul grado di Dependability condizionano certamente la scelta delle tecniche da utilizzare tra quelle sopra elencate.

2.4.1 Prevenzione dei guasti

Il metodo denominato *prevenzione dei guasti* ha come obbiettivo la corretta organizzazione delle attività precedentemente esposte e si avvale dell'utilizzo di tecniche come la modularizzazione, l' uso di linguaggi fortemente tipizzati, l'utilizzo di componenti affidabili, ecc. .

La *prevenzione dei gusti* pone molta importanza nelle fasi iniziali del processo di sviluppo del software, in quanto queste sono fasi cruciali per la corretta realizzazione del sistema. E' infatti nella prima fase del processo di sviluppo del software che vengono definiti i requisiti del programma; è quindi importante, attraverso l'utilizzo di una metodologia formale, definire in modo completo e non ambiguo cosa il sistema è tenuto a fare e come deve farlo. Il formalismo nella descrizione del sistema software è indispensabile affinché lo stesso possegga i requisiti di affidabilità e sicurezza. Lo sviluppatore deve sempre avere uno schema logico dettagliato del software che andrà a realizzare; per questo è necessario adottare tecniche di tracciamento dei requisiti come può essere lo sviluppo di una check list per la definizione di tutte le attività, verifiche e risultati.
L'importanza della definizione delle specifiche del prodotto è anche motivo della maggiore insidiosità degli errori che possono essere commessi in questa fase; infatti un errore di descrizione del comportamento che il software deve seguire comporta inevitabilmente un guasto, oltretutto difficile da risolvere in quanto insito nella definizione stessa del sistema.
Allo stesso modo bisogna fare attenzione alle attività di progettazione e disegno del software, in quanto sono le fasi in cui si definiscono i moduli, le interfacce che li collegano e la struttura complessiva dell'applicazione. Per questo motivo è necessario utilizzare una precisa metodologia di disegno condivisa da tutto il team di sviluppo consapevole dell'importanza di questo strumento che se trascurato o approssimato porta ad errori progettuali.
Nella realtà queste prime fasi, pur essendo fondamentali per la corretta realizzazione del software, sono spesso trascurate: o gli sviluppatori concentrano la propria attenzione direttamente nella fase di implementazione, oppure nel team di sviluppo potrebbe mancare personale con competenza manageriale per progetti simili, oppure ancora il committente potrebbe anteporre la puntuale consegna del lavoro alla cura dell'affidabilità del sistema.
La gestione dell'affidabilità del progetto non può né essere lasciata alla volontà dei singoli sviluppatori né alle richieste del committente, deve essere invece curata fin dall'inizio tramite l'utilizzo di metodologie adeguate e l'inserimento nel team di figure esperte e competenti.

Questo poiché vi è la necessità che ogni singola attività del processo di sviluppo software venga correttamente eseguita in modo tale da realizzare un prodotto completo e corretto, che non tralasci l'aspetto dell'affidabilità. In sostanza la prevenzione dei guasti consiste nel miglioramento del processo di sviluppo del software.

2.4.2 Tolleranza ai guasti

Le tecniche di *tolleranza ai guasti* puntato a ridurre i danni provocati dall'attivazione di guasti e in generale dal verificarsi di errori. In questo modo è possibile evitare che il sistema incorra in fallimenti, in modo tale da permetterne la prosecuzione delle funzionalità. Vedremo nel dettaglio questo punto nel prossimo capitolo.

2.4.3 Rimozione dei guasti

Le tecniche di *rimozione dei guasti* puntano a scoprire i guasti insiti all'interno del sistema e quindi correggerli. Le fasi di applicazione di queste tecniche possono essere la fase di sviluppo del sistema oppure direttamente la fase d'uso.

Durante la fase di sviluppo la tecnica prevede tre passaggi:

- Nel primo passaggio si verifica che il sistema sia effettivamente conforme alle sue specifiche. Questa fase viene detta *verifica (verification).*
- Nel secondo passaggio, qualora il sistema non corrisponda alle specifiche, si individuano i guasti e la loro natura. Questa fase è chiamata *diagnosi (diagnosis).*
- Nel terzo passaggio si punta a correggere gli errori trovati nella fase precedente. Questa fase è chiamata *validazione (validation).*

Nel primo passaggio la verifica che viene effettuata può essere *statica* o *dinamica*. La verifica statica si avvale dell'utilizzo di modelli e astrazioni e

consiste nell'ispezionare i requisiti, la progettazione e il codice dell'applicazione al fine di individuare problemi, o inadeguatezza alle specifiche. Il sistema quindi non deve essere in esecuzione. La verifica dinamica controlla invece i moduli del sistema o il sistema nella sua interezza. Il sistema in questo caso deve essere in esecuzione; gli vengono passati in ingresso dei dati e viene osservato se il comportamento è corretto.

Applicare invece le tecniche di *rimozione dei guasti* durante la fase d'uso del sistema consiste nel procedere con le azioni di manutenzione. La manutenzione può essere preventiva o correttiva. Preventiva qualora le correzioni venissero apportate al fine di prevenire l'errore, correttiva quando invece l'errore si è già manifestato.

2.4.4 Previsione dei guasti

La *previsione dei guasti* è il mezzo per valutare il comportamento futuro del sistema e stimare quindi il livello di affidabilità che il sistema è in grado di fornire nel rispetto delle sue specifiche.

L'approccio con cui può essere eseguita questa tecnica può essere di due tipi: *qualitativo* o *quantitativo*. L'approccio quantitativo cerca di identificare e classificare i fallimenti che possono interessare il sistema, definendone inoltre la combinazione di eventi che li hanno generati e le conseguenze che comportano. L' approccio quantitativo cerca invece di stimare direttamente il grado di Dependability del sistema valutando il limite in cui alcuni requisiti sono soddisfatti.
Per fare questo l'approccio quantitativo può adottare due tecniche: la modellazione del sistema oppure la misurazione diretta. La modellazione del sistema consiste nella costruzione di modelli che rappresentino il sistema, nella validazione di questi e infine nell'esecuzione delle misurazione della Dependability di questi modelli. I modelli probabilistici più semplici sono quelli basati sul calcolo combinatorio, che permettono di ottenere facilmente stime della Dependability di sistemi semplici, mentre diventano meno attendibili

quando vengono applicati a sistemi con interazioni inter-componenti complesse. A fronte quindi di un costo di attuazione contenuto, si ha purtroppo una scarsa accuratezza delle stime. La misurazione diretta consiste in uno studio del sistema da un punto di vista sperimentale, testando direttamente il sistema ed ottenendo misure campionarie. Tuttavia la misurazione diretta del sistema soffre del problema dell'ammissibilità del risultato ottenuto in quanto, essendo frutto di una serie limitata di osservazioni, non può essere preso per sicuramente vero. Per ridurre il grado di incertezza bisogna aumentare il numero delle osservazioni ma questo non è sempre possibile, poiché i costi di realizzazione di tali osservazioni crescono e i tempi di ottenimento di un risultato diventano sempre più lunghi.

La Fault Forecasting deve trovare il giusto compromesso tra accuratezza dei risultati ottenuti e costi sostenuti per effettuare tali misurazioni. Nel caso di sistemi complessi è necessario utilizzare sia l'approccio qualitativo che quantitativo per ottenere risultati soddisfacenti e affidabili.

Capitolo 3

La tolleranza ai guasti

Dalla trattazione svolta nel capitolo precedente si è compreso come la tecnica di *prevenzione dei guasti* sia volta a risolvere quegli errori deterministici e replicabili che abbiamo definito, sempre nel Capitolo 2, *guasti solidi* e che possono essere rimossi tramite rigorose fasi di testing e debugging. Tuttavia, l'utilizzo della sola tecnica di prevenzione non può assicurare che il software sia totalmente privo di guasti[7], soprattuto nel caso di sistemi di elevata complessità. Bisogna, quindi, considerare possibile che una qualche parte del software possa contenere guasti nascosti che potrebbero attivarsi nella fase operativa dello stesso. Per questo motivo i progettisti devono valutare l'impatto che potrebbe avere l'eventuale verificarsi di un guasto in una specifica parte del programma e in un momento successivo al rilascio del prodotto, adottando quindi le precauzioni appropriate. I guasti di cui stiamo parlando, che non vengono corretti nelle fasi di testing e debugging e che rimangono nel software al momento del rilascio, sono quegli errori che abbiamo definito nel Capitolo 2 come *guasti sfuggenti*, chiamati così appunto perché eludono i controlli della tecnica di *prevenzione dei guasti*.
L'attività di sviluppo del software è poi spesso condizionata, oltre che dalle problematiche di progettazione del sistema più volte menzionate, anche da vincoli economici e di mercato che obbligano a ridurre i tempi di sviluppo e produzione, costringendo i progettisti e gli sviluppatori a tempistiche che non permettono il corretto svolgimento di fasi importanti quali testing e verifica.

Ciò aumenta ulteriormente la difficoltà di garantire l'assenza di difetti nel software.

Per questo motivo, oltre all'implementazione di tecniche per la prevenzione dei guasti, è necessario adottare un *disegno (design)* per l'applicazione in grado di gestire l'eventuale verificarsi di un errore in un momento successivo al suo rilascio. Tale design è fornito dalle tecniche di tolleranza ai guasti, le quali rimangono l'unica possibilità per poter produrre software affidabili. Le tecniche di tolleranza ai guasti sono volte a rilevare i guasti (ed eventualmente ad eliminarli) mantenendo il sistema in uno stato funzionante, evitando che lo stesso degeneri in fallimenti che comprometterebbero la corretta fornitura del servizio. La tolleranza ai guasti deve essere implementata nelle fasi di progettazione del sistema, onde evitare di accorgersi dell'inadeguatezza del software solo successivamente alla sua realizzazione, momento nel quale diventa difficile, se non impossibile, attuare modifiche al fine di migliorare l'affidabilità del sistema[8].
Quanto detto è fondamentale soprattutto in presenza di sistemi software che presentano specifiche di affidabilità di alto livello.

Esistono tecniche di tolleranza ai guasti realizzate appositamente per il livello hardware, altre realizzate per il livello software. e altre ancora che coinvolgo entrambi i livelli.
A livello hardware possiamo accennare alcune possibili implementazioni:

- **Ridondanza di alimentazione**: l'adozione di sistemi di alimentazione supplementare è utile nel caso in cui venga a mancare l'alimentazione, magari in seguito ad un guasto dell'alimentatore principale oppure ad una mancanza di tensione. Grazie a questa tecnica il sistema è in grado di continuare a funzionare senza interruzione di servizio.

- **Sistemi hardware multiprocessore**: la progettazione di sistemi hardware di questo tipo, definiti SMP (multiprocessore simmetrico), permette di utilizzare contemporaneamente i diversi processori sfrut-

tando la potenza del calcolo distribuito ed evitando il blocco del sistema qualora uno dei processori dovesse guastarsi.

- **Ridondanza dell'informazione**: questa tecnica è alla base del sistema RAID (Redundant Array of Independent Disks) e combina un insieme di supporti per il salvataggio dati al fine di distribuire e replicare le informazioni; il sistema è così dotato di maggiore robustezza, di una più sicura integrità dei dati, e di un aumento di prestazioni.

A il livello software invece, per comprendere l'importanza delle tecniche di tolleranza ai guasti, bisogna tener conto della pratica comune, da qualche decennio a questa parte, di spostare gli aspetti di controllo e gestione del sistema dal livello hardware al livello software. Per questo motivo, affinché i sistemi possano fornire il proprio servizio nel rispetto delle specifiche, è necessario sia adottare tecniche di tolleranza ai guasti a livello hardware sia anche, e soprattutto, implementare tecniche di affidabilità a livello software. L'utilizzo di queste tecniche permette di rilevare e gestire gli errori mantenendo l'esecuzione del sistema coerente alle specifiche dello stesso.

I sistemi software di prossima generazione sono sistemi dall'architettura hardware e software sempre più complessa e per questo motivo necessitano di tecniche per il controllo della correttezza dell'esecuzione. Pertanto la tolleranza ai guasti è diventata una componente imprescindibile per lo sviluppo di questi sistemi. Un esempio sono i *sistemi* embedded che sono progettati per eseguire una determinata funzione e sono spesso basati su una struttura hardware apposita e non riprogrammabile dall'utente finale. Gran parte di questi sistemi è progettata al fine di eseguire un numero definito di operazioni, nel rispetto di determinati vincoli di prestazione minimi. L'hardware stesso è progettato al fine di minimizzare le componenti necessarie e massimizzare le prestazioni ottenibili. Ciò che fa sì che il *sistema embedded* richieda un determinato livello di affidabilità è il fatto che questi sistemi siano il più delle volte progettati per rimanere costantemente attivi per un periodo di tempo relativamente lungo e senza l'intervento umano. Per questo motivo vengono dotati di determinati meccanismi di tolleranza ai guasti che gli permettano

di essere autosufficienti e di recuperare uno stato di operatività qualora si verificasse un errore interno.
Più in generale si può evidenziare l'importanza dell'affidabilità di un sistema software prendendo in considerazione i sistemi cosiddetti *real-time*, quei sistemi, cioè, che richiedono che l'esecuzione del calcolo avvenga in un tempo definito affinché l'operazione possa considerarsi corretta. La necessità di ottenere i risultati dei calcoli entro un tempo limite è data dal fatto che questi sistemi hanno bisogno di eseguire determinate operazioni a velocità predefinite. Un altro requisito dei *sistemi real-time* è la continuità operativa, intesa come capacità di fornire il servizio per lunghi periodi senza alcuna interruzione. Per fare un esempio di sistemi complessi in grado di reagire ad un evento in real-time si pensi al sistema di controllo di volo di un aereo o al sistema di controllo di una centrale nucleare. Le tecniche di tolleranza ai guasti sono difatti utilizzate nei settori aerospaziale, energia nucleare, sanità, telecomunicazioni e industria dei trasporti di terra, e tanti altri.

Cogliere la necessità dell'introduzione di queste tecniche significa prima di tutto comprendere la natura stessa del problema che si intende risolvere. Ciò che queste tecniche mirano a correggere sono errori il più delle volte derivanti da un'errata fase di progettazione del software, che ha fatto sì che un errore umano divenisse un guasto interno del software[9]. Questi errori umani sono estremamente comuni per il semplice motivo che, come già detto, negli ultimi tempi si tende a concentrare la complessità di un sistema hardware-software nella sola parte software. É stato stimato che il 60-90% degli errori che si verificano in un sistema provengono dal livello software[10].

Definiamo quindi la tolleranza ai guasti di un prodotto software come la capacità dello stesso di rilevare e porre rimedio ad un errore che è avvenuto o che sta accadendo, minimizzandone le conseguenze; la tolleranza ai guasti serve ad evitare che un guasto in un componente o sotto-sistema porti ad un malfunzionamento dell'intero sistema.

3.1 Principi di progettazione

I principi di progettazione di un sistema tollerante ai guasti possono essere così elencati[11]:

3.1.1 Ridondanza

La ridondanza è uno strumento fondamentale per l'ottenimento dell'affidabilità del sistema dal momento che, qualora una parte del sistema non dovesse funzionare correttamente, è necessario vi sia una parte di riserva predisposta a svolgere la medesima funzione.
La ridondanza è comunemente usata a livello hardware tramite la replicazione dei componenti. Su questi componenti possono essere quindi eseguite parallelamente copie dello stesso programma. Tuttavia questa configurazione fornisce tolleranza a guasti solo di origine hardware. Infatti, per quanto riguarda il software, la semplice replicazione non è in grado di fornire una buona tolleranza ai guasti. Questo è dovuto alla differenza sostanziale che intercorre tra hardware e software: l'hardware si può danneggiare e quindi la replicazione del componente è in grado di risolvere il problema, il software invece non può rompersi. Spesso, come già detto, i guasti nel software sono causati da errori di progettazione e design, che sarebbero presenti in tutte le copie del programma qualora questo venisse semplicemente replicato[12].
Vedremo nel prossimo capitolo che la soluzione a questo problema consiste nella diversificazione delle copie del programma, quindi nel non utilizzare una semplice replicazione dello stesso ma replicare piuttosto il servizio offerto. Questo significa anche che i moduli software, ora non più uguali, possono produrre risultati diversi. Si deve quindi adottare un *meccanismo di decisione (Decision Mechanism)*, che nel proseguo della trattazione indicheremo anche con DM, che decida quali risultati ritenere accettabili e soprattuto quale output finale restituire e se restituirne uno.

La progettazione della ridondanza software è strettamente collegata alla struttura hardware adottata. Le configurazioni possibili (vedi Figura 3.1) sono principalmente tre: la prima soluzione prevede le diverse varianti imple-

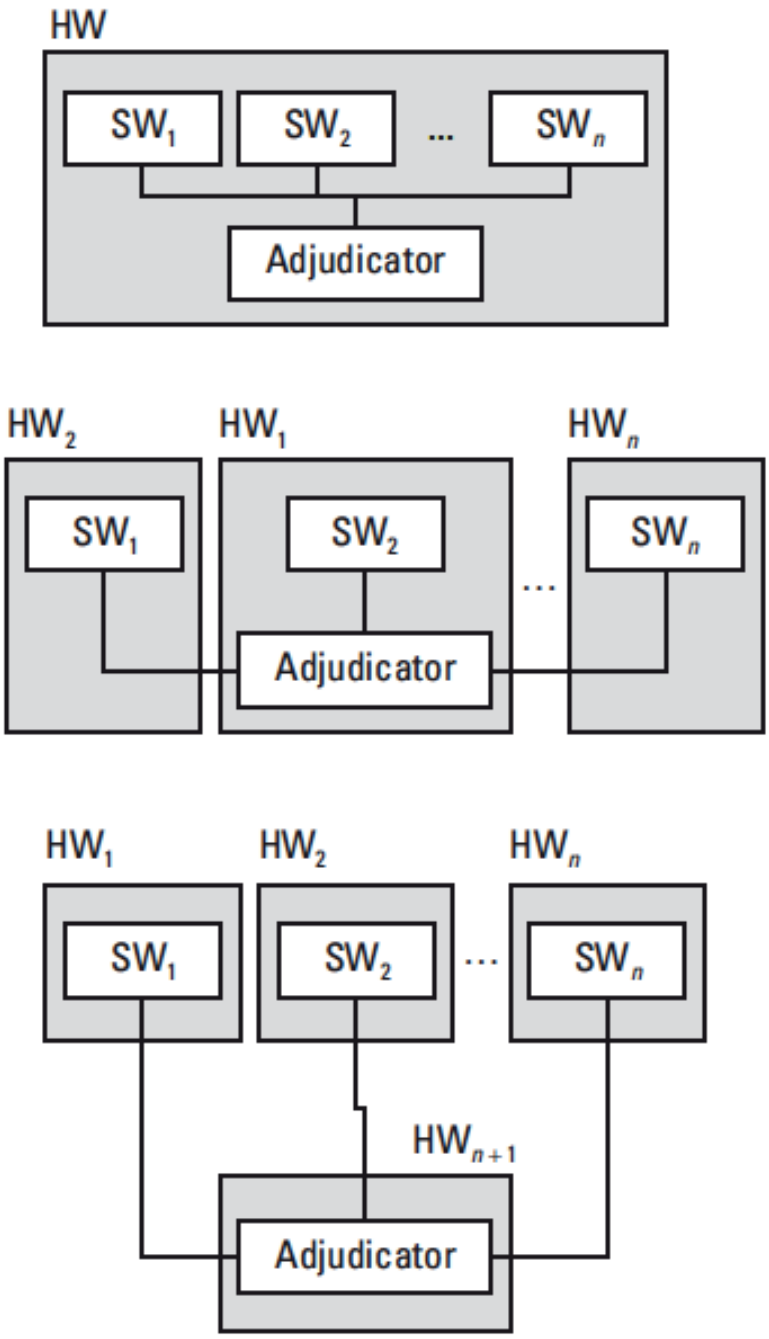

Figura 3.1: Le possibili configurazioni della ridondanza[11]

mentate su un unico componente hardware, la seconda soluzione prevede un componente hardware dedicato ad ogni modulo software e il DM collocato su uno di questi componenti, mentre la terza dedica un componente hardware separato anche al DM.

Il ricorso alla ridondanza determina quindi un aumento del livello di disponibilità del sistema a fronte tuttavia di una conseguente crescita del costo finale del sistema dovuto all'incremento del numero di moduli e dispositivi implementati e relativi costi di riparazione e manutenzione. Tratteremo nel dettaglio i costi della ridondanza nel Capitolo 5.

3.1.2 Isolamento dei guasti

Un sistema affidabile deve essere in grado di isolare i guasti in modo tale da evitare che uno solo di questi possa generare una serie di guasti multipli in cascata con conseguente alterazione del funzionamento del sistema. Un guasto individuato in un'unità dovrà quindi rimanere isolato a quella unità, evitando che si propaghi agli altri componenti. I guasti in grado di coinvolgere diverse unità, determinandone un malfunzionamento e rendendo di conseguenza inefficace la ridondanza, sono definiti in letteratura *guasti di modo comune*.

3.1.3 Rilevamento e notifica dei guasti

Una volta realizzato un sistema ridondante in grado di isolare i guasti alle unità dove si sono verificati rimane la necessità di intervenire rapidamente apportando le giuste modifiche volte a correggere il guasto, in modo tale che il sistema continui ad essere ridondante e che il *tempo medio di riparazione MTTR* (parametro visto nel Capitolo 2) rimanga contenuto. Partendo dal presupposto che il sistema possa contenere guasti latenti che potrebbero in un futuro attivarsi arrecando danni al sistema, è giusto adottare meccanismi di tolleranza ai guasti che non si limitino alla sola rilevazione dei guasti attivi ma che invece tentino anche di rilevare guasti nascosti nel sistema.

A livello software il rilevamento è attuato tramite i cosiddetti "test di accettabilità" dei risultati. Vi sono diverse modalità di controllo, le più diffuse sono:

- **Controllo di replicazione,** con cui si verifica che i risultati prodotti dai moduli siano coerenti. Si parte dal presupposto che i moduli vengano sviluppati in maniera autonoma e che quindi tendano a guastarsi indipendentemente. Si suppone in questo caso che non esistano errori di modo comune.

- **Controllo del tempo**, basato sull'utilizzo di un timer, che ha il compito di fermare il processo in esecuzione qualora questo non ritornasse un risultato entro un tempo definito.

- **Controllo all'indietro**, con cui si verifica la validità degli output utilizzando questi come dati in ingresso per la funzione inversa che permette di riottenere gli input iniziali. La non corrispondenza identifica un errore nel calcolo del risultato.

- **Controllo di consistenza**, con cui si controlla , attraverso l'utilizzo della ridondanza e di funzioni di verifica, la ragionevolezza e l'integrità del risultato. Basandosi quindi sulla conoscenza delle caratteristiche che devono possedere i risultati delle elaborazioni, il sistema è in grado di rilevare eventuali guasti ed errori. Per quanto riguarda il controllo della ragionevolezza del risultato un esempio può essere un risultato che per specifica non può mai essere superiore ad un certo valore; dovesse accadere il contrario il sistema è quindi in grado di determinare che il modulo che ha prodotto tale risultato è mal funzionante. Per quanto riguarda invece il controllo dell'integrità vengono prodotte delle varianti dei dati di input, chiamate codifiche, che sono ottenute allegando informazioni aggiuntive al dato originale. Le codifiche ottenute sono validate tramite l'utilizzo di codici correttori in grado di individuare eventuali variazioni sui bit del dato.

- **Controllo delle capacità**, che permette di testare il funzionamento dei componenti hardware del sistema passando loro determinati input

e chiedendo loro di eseguire operazioni di cui si conosce a priori il risultato corretto. Un altro possibile test è volto a verificare la corretta comunicazione tra i processori. Questo controllo viene eseguito richiedendo periodicamente agli stessi di scambiarsi informazioni tra loro, verificando il corretto invio/ricezione dei dati.

3.2 Strategie di tolleranza ai guasti software

Le strategie di tolleranza ai guasti livello software hanno sostanzialmente due obbiettivi (vedi Figura 3.2): il *rilevamento dell'errore (error detection)* e il *recupero della stabilità del sistema (recovery).*

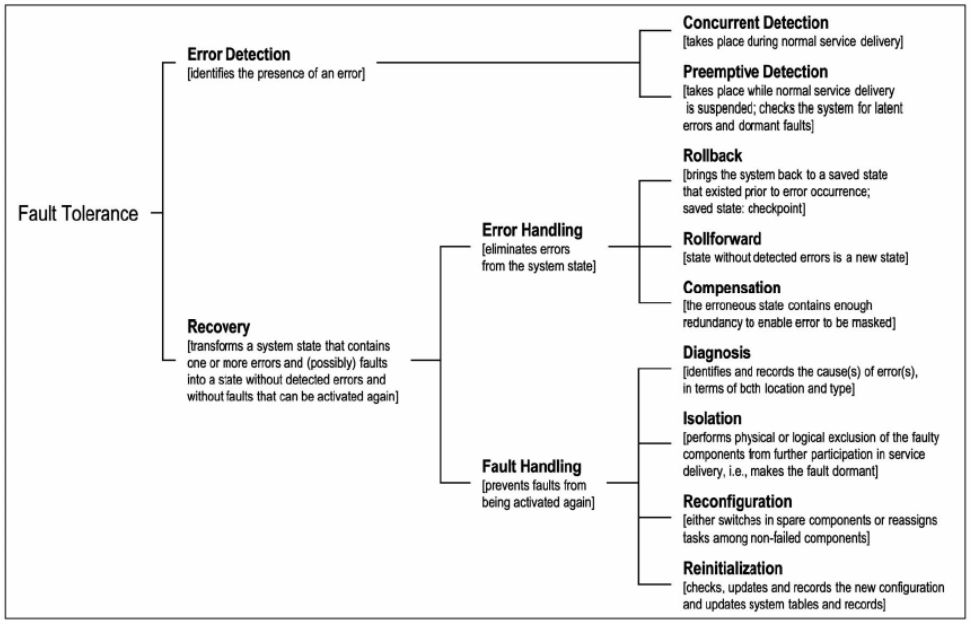

Figura 3.2: Le strategie di tolleranza ai guasti [4].

3.2.1 Rilevamento dell'errore

In questa prima fase si tenta di individuare gli eventuali errori presenti nel sistema e si procede quindi alla loro segnalazione. Laprie[4] propone due

possibili strategie di rilevamento degli errori:

1. La *rilevazione concorrente (Concurrent Detection)* che si svolge nel corso del normale funzionamento del sistema e dunque durante la fornitura dei servizi.

2. La *rilevazione preventiva (Preemptive Detection)* che si svolge nel corso dei periodi di sospensione del servizio e serve all'individuazione di possibili guasti latenti nel sistema.

3.2.2 Rimozione dell'errore

La seconda fase consiste in un insieme di operazione volte a ripristinare il corretto funzionamento del sistema mediante l'eliminazione dell'errore dal sistema stesso (*error handling*).
Le modalità con cui si può operare l'error handling sono tre:

1. **Recupero all'indietro** (Backward recovery): il sistema è riportato all'ultimo stato stabile in cui permaneva prima del riconoscimento dell'errore. Questo è reso possibile dal salvataggio di un insieme di stati stabili in cui si è trovato precedentemente il sistema. Questi stati salvati sono chiamati checkpoints. Il recupero all'indietro è indicato quando il sistema è affetto per lo più da guasti transienti ed è quindi possibile che, riportando il sistema ad uno stato passato ma stabile, poi il guasto non rigeneri nuovamente l'errore.

2. **Recupero in avanti** (Forward recovery): il sistema, che si trova in uno stato erroneo, è portato in un nuovo stato stabile privo di errori. Per far questo può essere necessaria una lunga serie di trasformazioni dello stato corrente del sistema.

3. **Compensazione**: questa modalità permette di correggere immediatamente lo stato del sistema. Il suo impiego è possibile quando il sistema possiede sufficienti componenti ridondanti che gli permettano di mascherare gli errori. Questo requisito comporta maggiore complessità della fase di progettazione e costi superiori di realizzazione.

L'utilizzo di una modalità non esclude la possibilità di poterne utilizzare congiuntamente un'altra. Vedremo infatti nel prossimo capitolo come alcune tecniche di tolleranza ai guasti tentino prima un recupero all'indietro, nella speranza che l'errore non si ripresenti con l'esecuzione successiva. Qualora l'errore si verifichi di nuovo allora procedono con la ricerca di uno stato stabile per il sistema.

3.2.3 Rimozione dei guasti

La seconda fase consiste in un insieme di operazione volte ad evitare che il guasto che ha generato l'errore possa riattivarsi provocando ulteriori errori (*fault handling*).
L'operazione di fault handling prevede invece le seguenti fasi:

1. **Diagnosi** (Diagnosis), che consiste nell'individuare la localizzazione dell'errore e il suo istante di occorrenza.

2. **Isolamento** (Isolation), durante la quale si esclude dal sistema il componente guasto in modo che non possa danneggiare la fornitura di altri servizi.

3. **Riconfigurazione** (Reconfiguration), ovvero si assegnano ai componenti integri i task in esecuzione sul componente guasto al momento in cui si è verificato l'errore. Questo è necessario per mantenere il livello di performability del sistema.

4. **Reinizzializzazione** (Reinitialization), si realizza il ripristino del funzionamento del sistema attraverso un aggiornamento delle informazioni alterate dalle precedenti operazioni.

3.3 I problemi delle tecniche di tolleranza ai guasti

Sviluppare sistemi altamente affidabili non è semplice. In questo sottocapitolo vedremo appunto alcuni dei problemi che si possono incontrare

implementando le tecniche di fault tolerance.
Sarebbe infatti sbagliato pensare che i vantaggi che l'utilizzo di queste tecniche portano in realtà non comportino anche dei problemi e dei costi. Vengono infatti individuati tre principali problemi che affliggono le tecniche di tolleranza ai guasti: il errori simili (*similar errors*) , il problema del confronto coerente (*the consistent comparison problem*), e il problema dell'effetto domino (*the domino effect*).

3.3.1 Errori simili

Abbiamo detto più volte che i guasti livello software gestiti dalle tecniche di tolleranza ai guasti sono guasti che spesso derivano da errori di progettazione o disegno dell'applicativo. Abbiamo anche detto che questi errori di progettazione non possono essere scoperti tramite la semplice replicazione del software poiché questa non farebbe altro che replicare anche l'errore nei vari moduli ridondanti. Per questo motivo c'è bisogno di diversità nella realizzazione dei moduli software.

Tuttavia può accadere che versioni del programma realizzate indipendentemente portino in qualche misura al medesimo errore[13]. Questo fenomeno e la difficoltà di riconoscere un errore durante l'esecuzione del sistema sono la causa del problema degli *errori simili*. Infatti la scelta del risultato finale può essere fatta in base ad un meccanismo di decisione di tipo maggioritario, per cui il risultato ritornato più volte dalle singole varianti del programma viene considerato il risultato corretto. Accade quindi che, qualora la maggioranza dei risultati fossero errati e simili tra loro, allora questi verrebbero considerati erroneamente corretti e ritornati come risultato finale.
Il motivo di questo errore è da ricercare nel meccanismo di verifica della validità dei risultati ottenuti dalle varianti. L'algoritmo che decide se un risultato è valido o meno tiene infatti conto di un margine di varianza del risultato. Due o più risultati che sono approssimativamente uguali nel limite di tale margine vengono definiti risultati simili e al di là che questi siano corretti o meno, se sono la maggioranza, l'algoritmo di decisione li considera

dei risultati corretti. Se questi risultati simili sono sbagliati allora vengono definiti errori simili.

La Figura 3.3 riporta un esempio di errori simili: consideriamo r1 e r2 due risultati errati, mentre il risultato r1 corretto. Quello che accade è che vengono considerati corretti i risultati r1 e r2 poiché il meccanismo di decisione è di tipo maggioritario e i due risultati r1 e r2 differiscono tra loro nel limite del margine tollerato; r1 e r2 vengono quindi definiti errori simili.

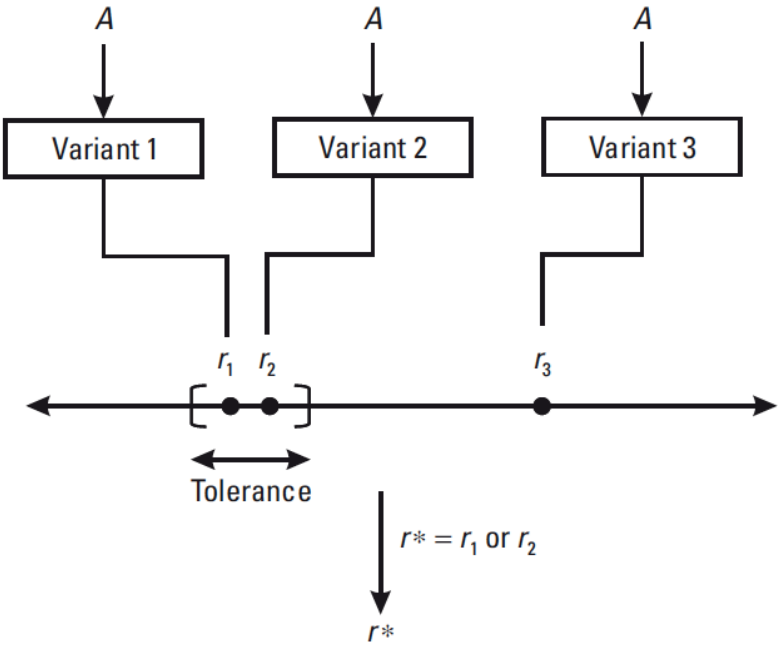

Figura 3.3: Esempio di errori simili.[17]

3.3.2 Il problema del confronto coerente

Il problema del confronto coerente (*consistent comparison problem, CCP*) è causato dalle limitazioni dell'aritmetica a precisione finita e dalle differenti scelte implementative che possono essere prese nelle diverse varianti del programma[14]. Per questo motivo non è possibile garantire che le varianti, pur rispettando le specifiche, arrivino al medesimo risultato. Possono invece

portare a risultati totalmente differenti tra loro.
Il problema in questo caso, a meno di un evidente errore interno ad un modulo, non è da imputare alle varianti ma alle specifiche del programma che evidentemente non definiscono in modo soddisfacente il livello di dettaglio con cui devono essere eseguiti i calcoli.
Le tecniche di tolleranza ai guasti, visto che il problema del CCP non è causato da guasti al software, non considerano queste discordanze come errori e quindi non eliminano nessun risultato ottenuto dalle varianti. Potrebbe quindi accadere che l'algoritmo di decisione del risultato ottimo non riesca a produrre una soluzione poiché i risultati, seppur corretti, differisco più del margine tollerato. In questo modo il CCP genera un fallimento che non sarebbe invece accaduto qualora non fosse stata implementata la tecnica di tolleranza ai guasti, poiché il singolo programma produce comunque un risultato, seppur approssimato.

3.3.3 Effetto domino

Il cosiddetto effetto domino (*domino effect*) [15] affligge quelle tecniche di tolleranza ai guasti basate sul ripristino all'indietro dell'ultimo stato stabile (backward recovery). Consiste nella propagazione del fallimento che si genera tra due o più processi comunicanti tra loro quando uno di questi rileva un fallimento del software (vedi Figura 3.4). Quando si implementa la tolleranza ai guasti in un sistema composto da processi comunicanti tra loro non si può trattare singolarmente ogni processo ed applicare la singola tecnica di tolleranza ai guasti ad ognuno di questi in modo separato. Così facendo si avrebbe ogni processo con il proprio meccanismo di rilevazione dell'errore e con il proprio punto di ripristino. In questa configurazione potrebbe quindi accadere che un processo, qualora rilevasse un errore, procederebbe a recuperare l'ultimo stato stabile registrato, comunicando agli altri processi il suo ripristino. Gli altri processi a loro volta, trovandosi in uno stato instabile, recupererebbero il loro ultimo punto di ripristino, generando una catena di ripristino stati che terminerebbe solo quando il sistema raggiunge un totale stato stabile, che tuttavia potrebbe essere lo stato iniziale. Questo compor-

terebbe la perdita dell'intera computazione svolta fino a quell'istante. Tutto ciò è dovuto ad una cattiva coordinazione della comunicazione tra i processi.

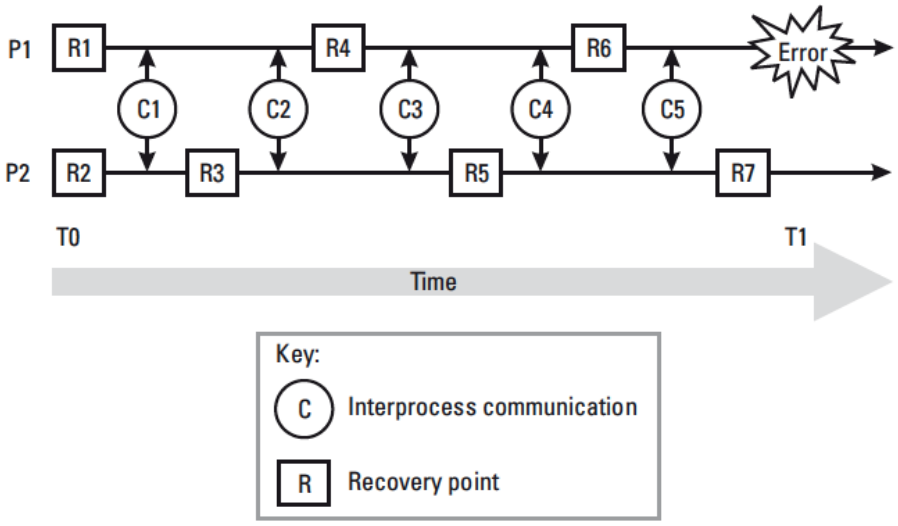

Figura 3.4: L'effetto domino[17]

Capitolo 4

Tecniche di tolleranza ai guasti

Vediamo in questo capitolo quali sono le tradizionali tecniche di tolleranza ai guasti.
Vale la pena innanzitutto sottolineare come, inizialmente, le uniche tecniche di tolleranza ai guasti fossero implementate a livello hardware e solo successivamente, quando il software divenne la componente principale del sistema, si intuì la necessità di dover dotare anche questo livello di misure di affidabilità. Si presentò quindi il problema di come realizzare queste tecniche; eseguire infatti una semplice trasposizione da un livello hardware ad un livello software delle tecniche già esistenti non sarebbe stata la soluzione adatta, poiché le stesse erano state progettate per far fronte a problemi di natura principalmente componentistica-ambientale e non software.
Nonostante questa fondamentale differenza, le tecniche di tolleranza ai guasti di livello software *(software fault tolerance, SFT)* realizzate per far fronte alle problematiche software, rimangono tutt'oggi molto simili alle tecniche hardware e sono spesso chiamate a rilevare e gestire anche guasti di livello hardware. Ciò che distingue la tolleranza ai guasti di livello software dalla tolleranza ai guasti di livello hardware è la necessitàdi ottenere non una indipendenza fisica tra le componenti hardware ma bensì una indipendenza logica tra i moduli software.

Possiamo quindi dare una prima classificazione delle tecniche dividendole

in due macrocategorie [11]:

- **Tecniche di rilevamento**. Tecniche atte principalmente al rilevamento dell'errore (sotto forma di input imprevisto o non corretto) ed al suo confinamento tramite opportune strategie. Segue poi la gestione delle eccezioni (siano esse interne, di interfaccia o dovute a malfunzionamento) e relativa notifica e propagazione. In queste tecniche non viene fatto uso della ridondanza.

- **Tecniche strutturate**. Tecniche fondate sull'utilizzo della ridondanza al fine di rilevare gli errori, correggerli o mascherarli. La correzione, come visto nel capitolo precedente, può essere realizzata nei seguenti modi: tramite recupero di uno stato di ripristino (checkpoint) salvato precedentemente all'ultima esecuzione; tramite trasformazioni dello stato del sistema atte al raggiungimento di uno stato corretto; tramite mascheramento ottenuto con il sistema del voting in grado di produrre un risultato finale corretto partendo da un insieme di risultati sia corretti che errati. I moduli software ridondanti possono essere eseguiti in modo distribuito su più componenti hardware oppure su una singola macchina. Possono inoltre essere eseguiti in parallelo oppure sequenzialmente.

In seguito procederemo alla trattazione delle tecniche appartenenti alla seconda categoria, di cui è possibile fornire una ulteriore classificazione sulla base dell'approccio utilizzato dalle tecniche al fine di contrastare il verificarsi di errori:

- **Versioni multiple del programma**. Si basa sul concetto di diversità di disegno *(Design Diversity)*, per cui vengono realizzate differenti copie del programma, implementate indipendentemente da team di sviluppo diversi ma programmate a svolgere la stessa funzione. Le principali tecniche che appartengono a questa categoria e che vedremo nel dettaglio in seguito sono: Recovery Blocks, N-Version Programming, Distributed Recovery Blocks, N Self-checking Programming, Consensus Recovery Block e Acceptance Voting.

- **Rappresentazioni multiple dei dati**. Le tecniche di tolleranza ai guasti basate sulla diversità della rappresentazione dei dati *(Data Diversity)* fanno uso di differenti formati di input dei dati per verificare la correttezza dei risultati. Delle tecniche appartenenti a questa categoria tratteremo le tecniche chiamate Retry Blocks e N-Copy Programming.

- **Esecuzione ripetuta in momenti diversi**. Si basa sul concetto di diversità temporale *(Temporal Diversity)* e consiste nell'eseguire il programma in momenti temporalmente distinti, oppure eseguendolo sulla base di input ottenuti in momenti differenti. Questo presuppone la presenza di guasti transienti che in un dato istante possono produrre un risultato errato mentre in un altro istante potrebbero non essere più presenti.

- **Riesecuzione in un ambiente diverso**. Si basa sul concetto di diversità ambientale *(Environment Diversity)* e presuppone che la riesecuzione programmata del software in un ambiente diverso, inteso come stato del sistema, possa evitare il verificarsi di un numero di errori tale da portare il sistema al fallimento. Di questa categoria vedremo la tecnica chiamata Rejuvenation.

Per comprendere appieno il funzionamento delle tecniche che andremo ad esaminare, è necessario prima trattare il concetto di *meccanismo di decisione (Decision Mechanism, DM)*, anche detto *aggiudicatore (Adjudicator)*. Il meccanismo di decisione, all'interno della tecnica, viene utilizzato per determinare se uno o più risultati posso essere considerati corretti o meno. Vedremo infatti che, quando uno o più moduli software concludono l'esecuzione, può essere previsto il controllo di accettabilità o di validità dei risultati ottenuti.
L'algoritmo implementato nel meccanismo di decisione può utilizzare diversi criteri di scelta, noi ne vedremo sostanzialmente due tipi: il *voter* e il *test di accettabilità (Acceptance Test)*. Il voter riceve in ingresso due o più risultati e determina il risultato corretto, ammesso che riesca ad individuarlo. Sostanzialmente quello che viene effettuato è una comparazione tra i risultati.

Qualora i risultati da controllare fossero solo due allora il voter prende il nome di *comparatore* e il risultato finale viene individuato se i due risultati sono concordi tra loro. Nel caso invece di 3 o più risultati, il voter prende il nome di *voter a maggioranza* e seleziona come presunto risultato corretto il risultato che compare più volte tra quelli ottenuti. Il test di accettazione controlla invece, sulla base di particolari condizioni definite dallo sviluppatore, la validità di un singolo risultato. Consiste in un algoritmo che definisce unicamente se accettare o meno il risultato e che necessita di essere semplice, efficace, altamente affidabile e che si attenga alle specifiche del prodotto. Per questi motivi un test di accettabilità risulta di difficile realizzazione e non sempre è in grado di riconoscere risultati erronei. É un strumento necessario per le tecniche auto-validanti come ad esempio: Recovery Blocks, Consensus Recovery Block, Distributed Recovery Block, Acceptance Voting, Retry Blocks.

4.1 Design Diversity

Il concetto di *diversità di disegno (Design Diversity)* si basa sul presupposto che la sola ridondanza software, a fronte di errori di progettazione e implementazione, non sia sufficiente a garantire la tolleranza ai guasti. Per tale motivo, nel realizzare le copie del programma, la ridondanza deve essere affiancata dalla *design diversity*, che prevede non una semplice replicazione del programma ma la realizzazione, ad opera di team di sviluppo indipendenti, di un insieme di implementazioni diverse tra loro chiamate tuttavia a svolgere la medesima funzione. Le tecniche appartenenti alla categoria della *diversità del disegno* consistono nella realizzazione di *varianti* del programma principale*,* anche dette *moduli,* in grado di fornire il medesimo servizio, ma sviluppate separatamente e quindi con un design e un'implementazione differenti [16]. Il presupposto fondamentale è che lo sviluppo indipendente dei moduli ridondanti possa diminuire la probabilità che si verifichi lo stesso tipo di malfunzionamento nei diversi moduli. Quindi si presume che se i moduli incorrono in errore, a seguito di guasti verificatesi al loro interno, lo

facciano in maniera indipendente l'uno dall'altro.

Il processo di definizione della *design diversity* ha inizio con la scrittura delle specifiche del prodotto. Le specifiche dichiarano i requisiti funzionali del software, i dati che devono essere trattati e quali decisioni devono essere prese su questi dati. Ogni variante viene sviluppata a partire da queste specifiche. Il calcolo consiste poi nell'esecuzione parallela o sequenziale di tutte le *varianti (moduli)* realizzate. Ogni variante restituisce un risultato ed è quindi necessario un mezzo per decidere quali di questi risultati ritenere corretti, dato che tali risultati possono non essere concordi tra loro. I risultati vengono quindi esaminati da un *meccanismo di decisione (Decision Mechanism)* che scegliendo quali dei risultati ritenere attendibili determina di conseguenza quali sono i moduli software che continueranno a fare parte del sistema. La Figura 4.1 rappresenta lo schema che descrive il funzionamento generale della *design diversity.*

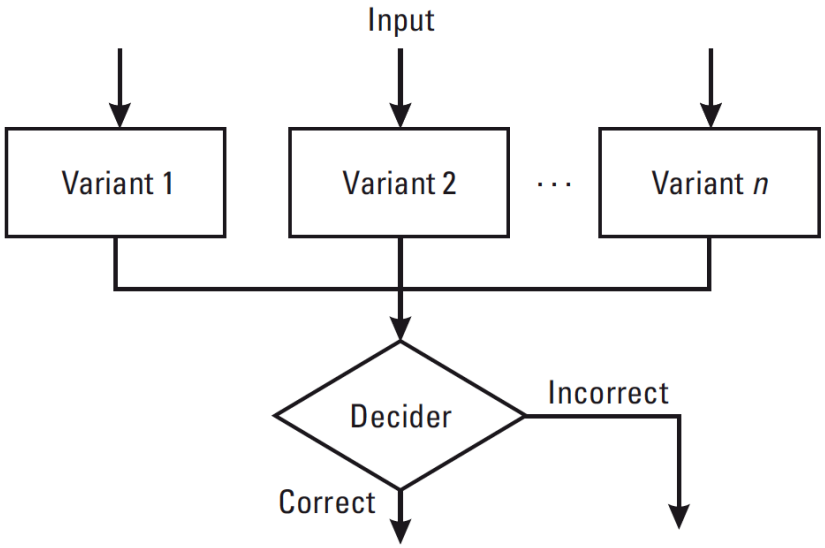

Figura 4.1: Schema generale della design diversity[12]

Le tecniche facenti parte della categoria *design diversity* sono certamente quelle più utilizzate. Questo perché permettono di ottenere tolleranza ai guasti guadagnando inoltre ulteriori vantaggi, ad esempio:

- Fasi di testing e validazione del software più veloci e meno costose, non essendo più indispensabile ottenere la rimozione totale degli errori ed è quindi possibile rilasciare il software in minor tempo.

- Ottenimento di un livello di affidabilità non raggiungibile con altre tecniche e con costi di realizzazione contenuti (sviluppo codice ridondante e utilizzo di strumenti di sviluppo adeguati) soprattutto in previsione di un minor numero di interventi di manutenzione successivi. Diversi studi hanno evidenziato che l'introduzione di una nuova variante non duplica il costo del sistema una bensì una spesa corrispondente al 70-80 percento del costo del singolo programma. Questo perché, seppur per definizione le implementazioni sono realizzate separatamente, vi sono comunque fasi comuni alle varianti, come ad esempio la stesura dei requisiti e il testing, che non richiedono la duplicazione.

Nell'implementare le tecniche di *design diversity* bisogna considerare due aspetti. Il primo è quello di determinare a quale grado di dettaglio bisogna decomporre il sistema in moduli diversificati e per fare questo bisogna considerare la dimensione media dei componenti; infatti, i componenti di piccole dimensioni sono generalmente poco complessi e sono quindi gestibili attraverso meccanismi decisionali più semplici rispetto a quelli dei moduli di grandi dimensioni (che solitamente presentano una complessità maggiore). Il secondo aspetto è quello di definire i livelli del sistema da coinvolgere nella diversificazione, scegliendo tra livello solo software oppure software-hardware [17].

Vediamo ora nel dettaglio le tecniche di tolleranza ai guasti appartenenti alla categoria *design diversity*.

4.1.1 Recovery Blocks (RcB)

La tecnica *Recovery Blocks* è stata una delle prime tecniche di tolleranza ai guasti basate sul concetto di *design diversity*. Fu presentata per la prima volta nel 1974 e successivamente implementata nel 1975 da B. Randell [15].

In questa tecnica la scelta del risultato corretto è realizzata sulla base della validazione dei risultati effettuata da un meccanismo di decisione di tipo *test di accettabilità (Acceptance Test, AT)*. Queste validazioni vengono eseguite durante l'esecuzione stessa del programma, per questo motivo viene considerata una tecnica dinamica. La tecnica RcB utilizza inoltre il meccanismo del recupero all'indietro (backward recovery) che consiste nel ripristino dell'ultimo stato stabile qualora si verificasse un errore.
La logica alla base della tecnica RcB è la seguente: essendo le varianti del programma sviluppate indipendentemente ed essendoci diversi modi per implementare lo stesso algoritmo, ne risulta che questi diversi moduli/blocchi presentino inevitabilmente caratteristiche tecniche diverse tra loro, come ad esempio l'efficienza della gestione della memoria, i tempi di esecuzione, ecc. L'RcB raccoglie tutti i moduli e li ordina in una classifica, ponendo nella posizione più alta il modulo con l'efficienza (presunta) maggiore. Questo blocco viene definito *blocco principale (primary try block)*. I moduli restanti vengono inseriti nella classifica progressivamente dopo il modulo principale in base al grado di efficienza a loro attribuito e sono poi messi in relazione al modulo principale come *blocchi alternativi (alternate try block)*. In questo modo abbiamo ottenuto l'ordinamento dei moduli in base alla loro efficienza. I moduli sono poi eseguiti uno alla volta, dal modulo più efficiente al modulo meno efficiente, finché il risultato prodotto da uno dei moduli non supera la validazione del *test di accettabilità*.
Lo schema base della tecnica RcB comprende: il controller che gestisce l'esecuzione generale, i blocchi (il principale e i secondari) che consistono nelle varianti di implementazione e il *test di accettabilità*. Alcune implementazioni dell'RcB, soprattutto quelle utilizzate nei sistemi real-time, includono l'utilizzo di un *watchdog timer, WDT* [18]. Lo schema di riferimento della tecnica RcB è rappresentato nella Figura 4.2.

Lo pseudocodice che rappresenta la tecnica RcB è invece rappresentato dall'Algoritmo 4.1. Dallo pseudocodice vediamo come la tecnica RcB tenti prima di soddisfare il *test di accettabilità AT* eseguendo il modulo principale. Se durante l'esecuzione del blocco principale viene sollevata un'eccezione, oppure

il risultato prodotto non supera l'AT, allora vengono eseguiti, nell'ordine definito dalla classifica, i moduli secondari finché uno dei risultati non soddisfa l'AT. Questo avviene solo se esistono altri moduli secondari non ancora eseguiti. Nel caso in cui nessuno dei risultati soddisfi l'AT allora viene sollevata un'eccezione.

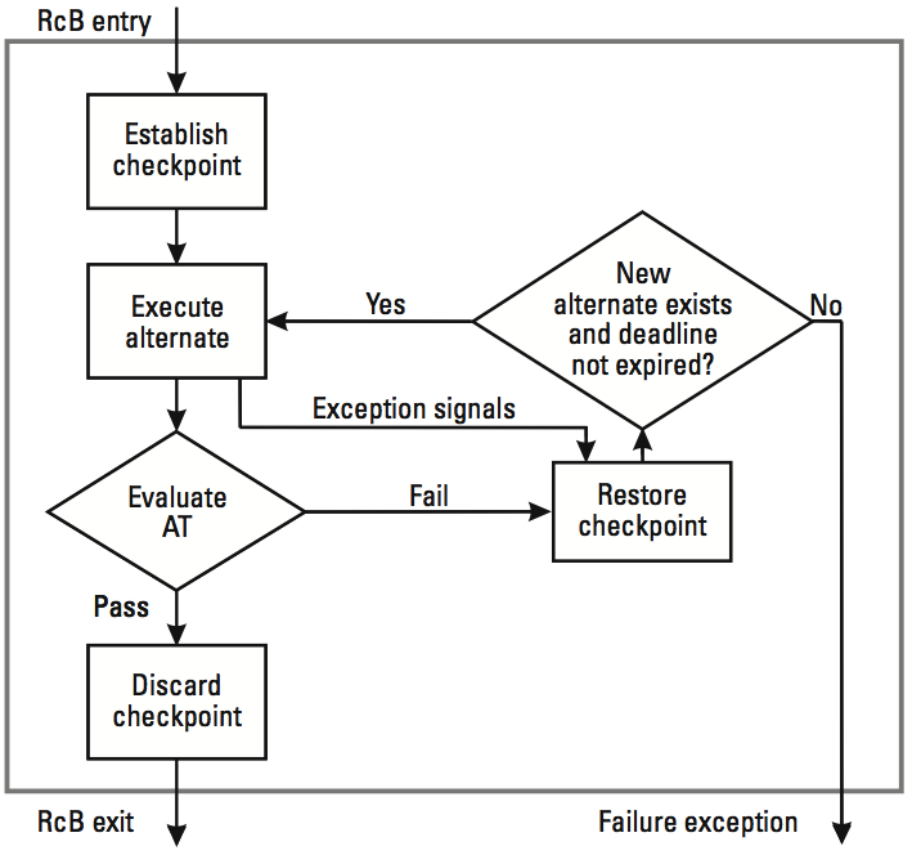

Figura 4.2: Funzionamento della tecnica RcB[17]

Algoritmo 4.1 Pseudocodice della tecnica RcB

```
ensure Acceptance Test
by Primary Alternate
else by Alternate 2
else by Alternate 3 .
..
else by Alternate n
else failure exception
```

Considerazione per la tecnica RcB

Sono state proposte diverse modifiche alla tecnica dell'RcB. La prima è l'introduzione di un timer per controllare il tempo di esecuzione del programma. Se prima dello scadere del timer nessuna delle varianti ha prodotto un risultato valido allora l'esecuzione generale dela tecnica viene interrotta e viene sollevata un'eccezione.
Una seconda modifica della tecnica prevede invece l'adozione di un *contatore del numero di esecuzioni dei blocchi alternativi (alternate routine execution counter).* Questo contatore viene incrementato ogni qual volta il modulo principale non supera il test di accettabilità e l'esecuzione viene quindi passata ad un blocco alternativo. Il contatore viene quindi utilizzato per stabilire il numero di volte che devono essere utilizzati direttamente i blocchi alternativi invece del modulo principale. In questo modo è possibile sapere per quante esecuzioni consecutive il modulo principale non sarà utilizzato e sfruttare quindi questo intervallo di inattività per apportare modifiche allo stesso.
Una terza possibile modifica consiste nell'adottare un *test di accettabilità* più evoluto che comprenda test multipli. Può per esempio essere eseguito un test prima dell'esecuzione del blocco principale al fine di controllare se i dati di input ai moduli sono validi o se sono in un formato particolare che richiede il passaggio dell'esecuzione direttamente ad un blocco alternativo.

Per quanto riguarda i problemi che interessano la tecnica RcB possiamo dire che essendo una tecnica appartenente alla categoria *design diversity* ed essendo basata sull'utilizzo del recupero all'indietro, soffre di conseguenza dei problemi che derivano da queste due metodologie: gli errori simili, il problema del confronto coerente e il problema dell'effetto domino. Problemi che abbiamo affrontato nel capitolo precedente. La tecnica RcB presenta poi dei problemi specifici legati alla logica stessa della tecnica. L'RcB esegue infatti le varianti del programma in modo sequenziale e il tempo necessario per l'ottenimento di un risultato finale comporta quindi dei costi elevati. In generale avremo una esecuzione di modulo, una esecuzione dell'AT e un ripristino dello stato, per ogni risultato che non supera il test di accettabilità. Nel caso

migliore si ha la validazione del primo risultato ottenuto, con un costo rappresentato dalla sola esecuzione del modulo principale e del test di accettabilità; nel caso peggiore invece, cioè nel caso in cui nessun risultato superi il test, il costo totale del processo è definito dalla sequenza modulo-AT-ripristino eseguita un numero di volte pari al numero delle varianti implementate. È evidente come il verificarsi del caso peggiore sia inaccettabile nei sistemi real-time.
Altro aspetto problematico per la tecnica RcB, e per tutte le altre tecniche che ne fanno uso, è la validità del test di accettabilità. Svilupparne uno infatti non è semplice e gran parte della validità del test dipende dalla completezza delle specifiche del software. Un semplice esempio di test di accettabilità può essere quello applicato all'algoritmo di ordinamento di numeri interi. Un possibile test per verificare che non vi siano gravi errori nel codice consiste nel controllare la somma dei numeri da ordinare prima e dopo l'ordinamento. Se la somma non corrisponde significa che si è verificato un errore. Tuttavia, come è facile intuire, questo controllo non evita che possa essere effettuato un ordinamento errato. Questo vuol dire che l'implementazione dell'AT può non essere in grado di catturare l'errore, con conseguente propagazione di quest'ultimo verso gli altri moduli e il rischio di portare al fallimento il sistema.

4.1.2 N-Version Programming (NVP)

La tecnica *N-Version Programming* è l'altra principale tecnica di tolleranza ai guasti di tipo *design diversity*. Fu proposta nel 1972 da Elmendorf [19] e implementata per la prima volta nel 1977 da Avizienis [20].
A differenza della tecnica RcB, la tecnica N-Version Programming viene definita una tecnica statica in quanto i risultati vengono controllati solo una volta terminata l'esecuzione di tutti i moduli. Il controllo è realizzato da un meccanismo di decisione di tipo *voter*. Tale meccanismo esamina i risultati delle singole varianti e identifica il risultato corretto calcolando la maggioranza dei risultati concordanti tra quelli ottenuti. L'esecuzione dei moduli può avvenire sequenzialmente sullo stesso componente hardware o in modalità

distribuita. L'esecuzione parallela presuppone la presenza di un numero sufficiente di componenti hardware da poter eseguire contemporaneamente su ognuno di questi un modulo diverso. Lo schema di riferimento della tecnica N-Version Programming è rappresentato nella Figura 4.3.

Lo pseudocodice che rappresenta la tecnica N-Version Programming è invece rappresentato dall'Algoritmo 4.2. Lo pseudocodice definisce l'esecuzione concorrente delle N varianti, queste esecuzioni ritornano N risultati che vengono passati al meccanismo di decisione di tipo *voter* il quale valuterà se è possibile identificare un risultato migliore tra quelli analizzati. Dallo pseudocodice si può vedere che nel caso in cui un risultato corretto viene individuato il meccanismo di decisione ritorna TRUE e viene quindi ottenuto il risultato. Qualora invece il DM non fosse in grado di individuare un risultato migliore, ritornando FALSE, allora viene sollevata un'eccezione.

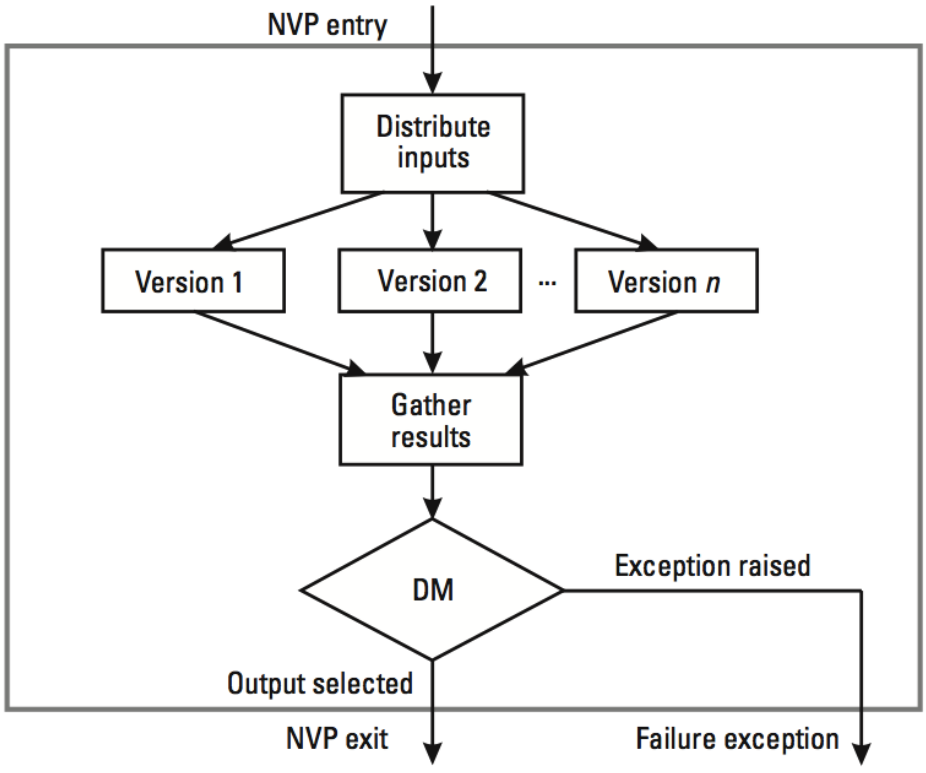

Figura 4.3: Funzionamento della tecnica NVP[17]

Algoritmo 4.2 Pseudocodice della tecnica NVP

```
run Version 1, Version 2, ..., Version n
if (Decision Mechanism (Result1, Result2,...,Result n))
   return Result
else failure exception
```

Considerazione per la tecnica NVP

Ci sono diverse logiche utilizzabili per il meccanismo di decisione (DM). Alcune logiche prevedono semplicemente un meccanismo differente da quello maggioritario del *voter*. Altre sono invece basate su un processo di voting differente. La logica alla base di queste varianti è la seguente: invece di attendere che ogni modulo concluda l'esecuzione per poi raccogliere tutti i risultati e valutarne la validità, si potrebbe invece procedere al confronto appena sono generati almeno due risultati. Se questi sono concordi allora il meccanismo di voting presume che rappresentino il risultato corretto e può quindi concludere il processo decisionale, se invece non sono concordi si attende il terzo risultato applicando poi il solito meccanismo maggioritario. Si procede così finché non si ottiene un risultato in maggioranza rispetto agli altri. Questa modifica è interessante poiché permette di ottenere un risultato anche senza aver valutato tutti i risultati di tutte le varianti. Difatti una volta deciso l'output possono essere interrotti i processi dei restanti moduli ancora in esecuzione con conseguente risparmio di tempo per il processo decisionale.

Per quanto riguarda i problemi che interessano la tecnica NVP possiamo dire che essendo una tecnica di tipo *design diversity* che utilizza il meccanismo del recupero in avanti eredita sia i benefici che i problemi correlati ad essi. Il fatto quindi che l'NVP si basi sull'implementazione di moduli differenti con lo stesso scopo per ottenere la tolleranza ai guasti, non esclude comunque che possano verificarsi guasti comuni nelle varianti indipendenti, negando il principio stesso della indipendenza dei guasti della *design diversity*. Può anche anche accadere che vengano commessi errori nella progettazione del meccanismo di decisione: in presenza di guasti può non essere quindi in grado di eliminare i risultati erronei o, al contrario, può respingere risultati corretti.

Altri problemi possono invece essere di natura strutturale. Se infatti la tecnica viene implementata, non su componenti distinti che lavorano in parallelo, ma bensì su un unico componente hardware (riducendosi alla semplice esecuzione sequenziale), allora bisogna attendere la conclusione di tutti i processi per proseguire con il calcolo del risultato finale. Pertanto, una possibile soluzione può essere l'utilizzo di un meccanismo di decisione in grado di produrre un output anche solo basandosi su 2 risultati. In questo modo i tempi, nel caso medio, vengono notevolmente ridotti rendendo la tecnica adottabile anche in sistemi real-time.

4.1.3 Distributed Recovery Blocks (DRB)

La tecnica *Distributed Recovery Block*, sviluppata da Kim [21], può considerarsi una combinazione tra la tecnica Recovery Blocks e l'esecuzione parallela-distribuita propria della tecnica N-Version Programming. Pur ereditando gran parte della logica dalla tecnica RcB, l'attenzione in questo caso è stata spostata sull'aspetto "tempo", distribuendo l'esecuzione su più componenti e adottando il meccanismo di recupero in avanti per riportare il sistema in uno stato stabile. Queste modifiche sono ottime soprattutto per un utilizzo della tecnica in applicazioni real-time. La tecnica consiste di due nodi semi-autonomi: un nodo chiamato nodo ombra (shadow node), copia dell'altro nodo chiamato nodo primario (primary node). Entrambi i nodi implementano al loro interno uno schema RcB. Questi due nodi sono in comunicazione tra loro e cooperano per trovare un unico risultato. L'AT implementato all'interno di ogni nodo viene in questa tecnica chiamato Logic and Time AT, in quanto alla sua funzione base di validazione del risultato viene aggiunto un timer che gli permette di sincronizzarsi con l'esecuzione dell'altro nodo. Lo schema di riferimento della tecnica Distributed Recovery Blocks è rappresentato nella Figura 4.4.

Lo pseudocodice che rappresenta la tecnica Distributed Recovery Blocks è invece rappresentato dall'Algoritmo 4.3. Dallo pseudocodice notiamo che la tecnica consiste nell'esecuzione parallela delle due tecniche Recovery Blocks

sui due nodi copia. Ogni nodo contiene due varianti dell'algoritmo di calcolo. Inizialmente nel nodo primario viene eseguito l'algoritmo principale mentre nel secondo nodo viene eseguito l'algoritmo alternativo. Una volta ottenuti i due risultati viene prima controllata la validità del risultato dell'algoritmo principale proveniente dal primo nodo. Se questo non supera il test allora si controlla il risultato dell'algoritmo alternativo del secondo nodo. Se nessuno dei due risultati supera l'AT allora viene rilanciata l'esecuzione parallela dei due nodi ma questa volta eseguendo l'algoritmo alternativo nel primo nodo e quello principale nel secondo nodo. Se anche i risultati di questi due processi non superano l'AT allora viene sollevata un'eccezione.

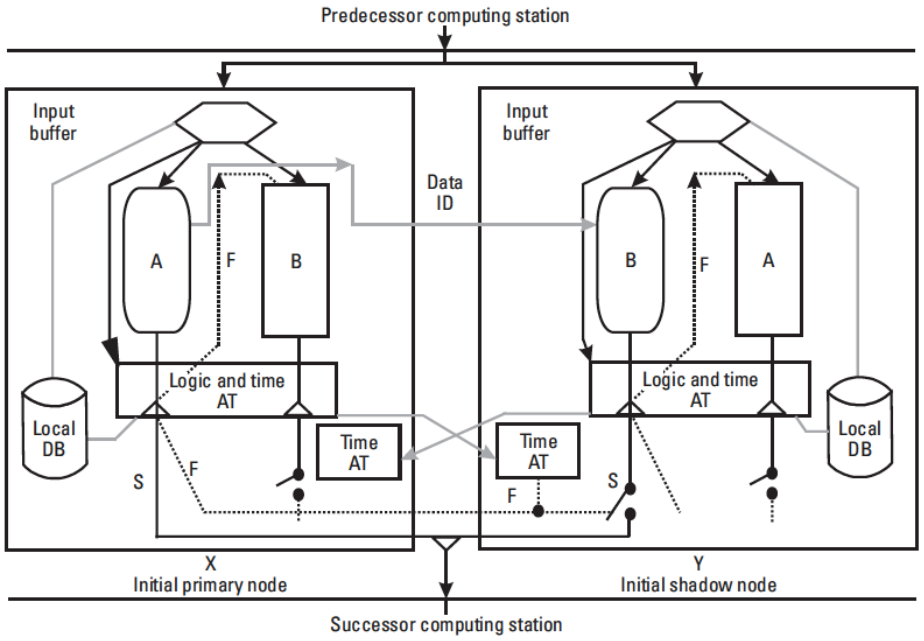

Figura 4.4: Funzionamento della tecnica DRB[17]

Algoritmo 4.3 Pseudocodice della tecnica DRB

```
run RcB1 on Node 1 (Initial Primary), RcB2 on Node 2 (Initial Shadow)
ensure AT on Node 1 or Node 2
    by Primary on Node 1 or Alternate on Node 2
    else by Alternate on Node 1 or Primary on Node 2
    else failure exception
```

Considerazioni per la tecnica DRB

La tecnica DRB si differenzia dalle tecniche precedenti in quanto, coniugando in un'unica struttura sia il *test di accettabilità* della tecnica Recovery Blocks sia il calcolo distribuito della tecnica N-Version Programming, permette di ottenere una tolleranza ai guasti sia di livello hardware sia di livello software. L'esecuzione parallela delle due varianti (primaria e secondaria) permette inoltre di elaborare il risultato in minor tempo. Questi aspetti rendono la tecnica DRB una valida alternativa per l'implementazione della tolleranza ai guasti in sistemi che richiedono alti livelli di affidabilità, come ad esempio i sistemi real-time.

Per quanto riguarda i problemi che interessano questa tecnica possiamo innanzitutto dire che eredita i problemi delle tecniche su cui si basa: *design diversity* e recupero in avanti. Allo stesso modo della tecnica Recovery Blocks, anche la tecnica DRB soffre del problema della qualità del test di accettabilità. Come abbiamo detto in precedenza, sviluppare un test di accettabilità efficiente è difficile poiché dipende fortemente dalle specifiche e può comunque essere a sua volta soggetto a guasti. Inoltre, essendo tale tecnica basata su una architettura multiprocessore, anche se il risultato dell'algoritmo principale del primo nodo passa il test di accettabilità, il costo totale dell'esecuzione comprenderà comunque anche il costo dell'esecuzione dell'altro algoritmo nell'altro nodo, oltre al costo del salvataggio del checkpoint e dell'esecuzione del test di accettabilità per entrambi i nodi. Questo costo rimane comunque relativamente basso per cui l'implementazione del DRB si presta bene per i sistemi real-time.

4.1.4 N Self-checking Programming (NSCP)

La tecnica *N Self-Checking Programming* fu sviluppata da Laprie [22] . È un tecnica di tipo dinamico e il suo aspetto fondamentale è la capacità del programma di auto-valutare il proprio operato nel corso dell'esecuzione. La logica su cui si basa questa tecnica è realizzata utilizzando un test di accettabilità su ogni risultato delle varianti e un comparatore su ogni coppia

di risultati. L'hardware a supporto della configurazione base di questa tecnica, con N varianti, consta di N componenti ognuno dei quali contiene una variante del programma. Questi componenti sono raggruppati in coppie e ognuna di queste coppie rappresenta una *unità auto-validante* (self-checking) della configurazione. Le N varianti, e quindi le N/2 unità, vengono eseguite in parallelo. All'interno dell'unità self-checking, una volta terminata l'esecuzione della coppia di varianti, i risultati delle due varianti vengono valutati singolarmente tramite un test di accettabilità. Se entrambi i risultati sono validati dall'AT, allora vengono confrontati dal meccanismo di decisione (che in questo caso consiste in un comparatore). Se sono concordi allora il comparatore produce il risultato dell'unità, che verrà poi confrontato con il risultato delle altre unità. Se invece l'esecuzione di una delle due varianti dovesse fallire, oppure uno dei risultati non dovesse essere validato, oppure ancora i risultati seppur validati non dovessero essere concordi tra di loro, allora i risultati delle due varianti verrebbero scartati e l'unità self-checking non produrrebbe alcun risultato. Si ha fallimento della tecnica NSCP nel caso in cui tutte le coppie di moduli producono risultati differenti e quindi ogni esecuzione delle *unità auto-validanti* fallisce. Lo schema che descrive la tecnica N Self-checking Programming è rappresentato nella Figura 4.5.

Lo pseudocodice che rappresenta la tecnica N Self-checking Programming nella versione a quattro varianti è rappresentato dall'Algoritmo 4.4. Notiamo dallo pseudocodice che le N varianti vengono eseguite a coppie e in modalità concorrente. Se nessun confronto tra i risultati delle varianti accoppiate ha esito positivo allora viene ritornato un errore. Se si verifica un errore in solo una delle due coppie allora viene considerato buono l'unico risultato ottenuto. Se, invece, entrambe le coppie hanno confronto interno positivo allora vengono confrontati i due risultati. Se questi sono concordi allora viene ritornato il risultato finale, se invece non corrispondono allora viene rilasciata un'eccezione.

Algoritmo 4.4 Pseudocodice della tecnica NSCP

```
run Variants 1 and 2 on Hardware Pair 1
run Variants 3 and 4 on Hardware Pair 2

compare Results 1 and 2
    if not (match)
        set NoMatch1
    else set Result Pair 1

compare Results 3 and 4
    if not (match)
        set NoMatch2
    else set Result Pair 2

if (NoMatch1) and not(NoMatch2)
    Result = Result Pair 2

else if (NoMatch2) and not (NoMatch1)
    Result =Result Pair 1

else if (NoMatch1) and (NoMatch2)
    raise exception

else if not (NoMatch1) and not (NoMatch2)
    then compare Result Pair 1 and 2
    if not (match)
        raise exception
    if (match)
        Result = Result Pair 1 or 2

return Result
```

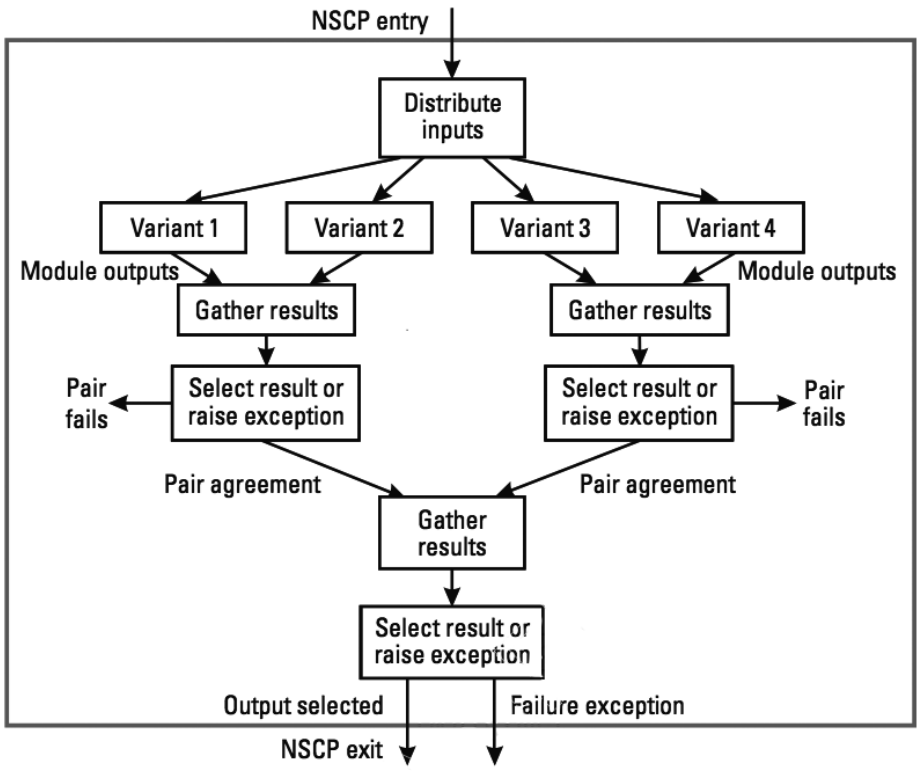

Figura 4.5: Funzionamento della tecnica NSCP[17]

Considerazioni per la tecnica NSCP

Come nel caso delle metodologie precedenti, essendo anche l'N Self-Checking Programming una tecnica appartenente alla categoria *design diversity* e *recupero in avanti*, eredita da queste i rispettivi problemi. I costi per il controllo di sincronizzazione tra le diverse coppie sono solitamente contenuti e raramente si hanno problemi di interruzioni di servizio da parte dei moduli o errori nella sincronizzazione dei risultati. Per questo motivo l'N Self-Checking Programming è una delle tecniche più utilizzate nei sistemi ad alta disponibilità e affidabilità.

4.1.5 Consensus Recovery Block (CRB)

La tecnica *Consensus Recovery Block* [23] rappresenta una ulteriore combinazione delle tecniche Recovery Blocks e N-Version Programming. Come

risultato si ha una tecnica al cui interno viene ridotta l'importanza del test di accettabilità della tecnica RcB ma allo stesso tempo in grado di gestire il problema che abbiamo definito problema dei *errori simili* (problema che affligge la tecnica NVP). La tecnica CRB richiede la progettazione e la realizzazione di N varianti dell'algoritmo che vengono poi classificate (come in Recovery Block) in base ad un grado di efficienza stimato non internamente alla tecnica.
Le N varianti vengono eseguite in parallelo e i risultati vengono raccolti dal meccanismo di decisione di tipo voter (come nel caso dell'NVP). Se il meccanismo di decisione, tipicamente maggioritario è in grado di individuare un risultato corretto, allora il processo termina. Se invece non viene individuato un risultato, l'esecuzione prosegue sulla linea della tecnica RcB. Viene infatti preso il risultato del modulo con posizione più alta nella classifica e viene sottoposto a test di accettabilità. Se questo risultato supera il test allora viene selezionato come risultato finale, altrimenti viene eseguita l'operazione di controllo dell'accettabilità sul risultato del modulo che occupa la seconda posizione in classifica e così via finché non viene individuato un risultato valido o non sono più disponibili risultati da valutare [24]. Il sistema incorre in fallimento qualora sia la parte RcB, sia la parte NVP della tecnica non sono in grado di produrre un risultato di output. Lo schema che descrive la tecnica N Self-checking Programming , dando per conosciute le architetture delle tecniche NVP e RcB, è rappresentato nella Figura 4.6.

Lo pseudocodice che rappresenta la tecnica Consensus Recovery Block è rappresentato dall'Algoritmo 4.5. Dallo pseudocodice è possibile notare come prima vengano eseguite tutte le N varianti del programma. I risultati sono poi sottoposti a comparazione tramite il meccanismo di decisione di tipo *voter*. Se il meccanismo di decisione ritorna TRUE, significa che ha individuato il risultato ottimo e quindi questo viene ritornato come output finale. Se invece il meccanismo di decisione ritorna FALSE, significa che il confronto non ha avuto esito positivo, quindi il controller generale passa alla fase RcB della tecnica che consiste nel verificare la validità di ogni singolo risultato sottoponendolo a test di accettabilità. Questo controllo viene iterato fino a

che non viene individuato un risultato accettabile. Se invece nessun risultato supera l'AT viene sollevata un'eccezione.

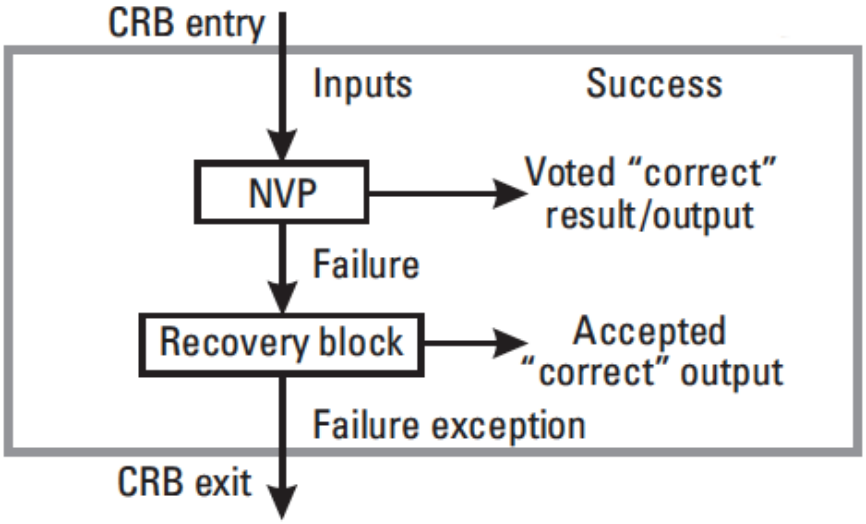

Figura 4.6: Funzionamento della tecnica CRB[17]

Algoritmo 4.5 Pseudocodice della tecnica CRB

```
run Ranked Variant 1, Ranked Variant 2, ..., Ranked Variant n
if (Decision Mechanism (Result 1, Result 2, ..., Result n))
    return Result
else
    ensure Acceptance Test
    by Ranked Variant 1 [Result]
    else by Ranked Variant 2 [Result] ...
    else by Ranked Variant n [Result]
    else raise failure exception
return Result
```

Considerazioni per la tecnica CRB

Essendo una tecnica di tipo *design diversity*, basata sul meccanismo del recupero in avanti, eredita da questi sia gli aspetti positivi che negativi. Uno svantaggio comune alle tecniche ibride è la complessità crescente del meccanismo di tolleranza ai guasti. La tecnica CRB deve infatti implementare entrambi i meccanismi di controllo delle due tecniche da cui deriva: il meccanismo di decisione appartenente alla tecnica NVP, e il test di accettabilità appartenente alla tecnica RcB. Al di là della difficoltà di realizzazione di questi due meccanismi rimane comunque il problema dell'aumento della complessità progettuale che comporta un maggior rischio di commettere errori di disegno e implementativi.
Particolari problemi propri della tecnica CRB non sono noti. I costi sostenuti per l'esecuzione della tecnica sono rappresentati dalla memoria dedicata all'esecuzione delle diverse varianti, dal tempo addizionale per la sincronizzazione, dal controllo effettuato tramite meccanismo di decisione e dall'esecuzione ripetuta dei test di accettabilità (qualora la prima fase NVP non producesse un risultato valido). Sono costi questi che rimangono comunque contenuti. Per questo motivo possiamo considerare la tecnica CRB indicata per quelle applicazioni che richiedono un alto grado di disponibilità e affidabilità.

4.1.6 Acceptance Voting (AV)

La tecnica *Acceptance Voting* fu proposta per la prima volta da Athavale [25]. Per ottenere la tolleranza ai guasti si utilizzano il *test di accettabilità* e il *voter* mentre la strategia di recupero consiste nel recupero in avanti. In questa configurazione tutte le varianti vengono eseguite in parallelo e ogni modulo possiede il proprio test di accettabilità. I risultati che superano il test vengono quindi passati al meccanismo di decisione di tipo voter il quale, visto che in questa configurazione può ricevere un numero variabile di risultati compreso tra 1 e N, deve implementare una meccanismo di voting dinamico, in modo tale da poter calcolare un output finale basandosi sui soli risultati ottenuti. Il sistema incorre in eccezione di fallimento se nemmeno uno dei test di accettabilità valida il risultato del modulo a cui è applicato

oppure se il meccanismo di decisione di tipo voter non è in grado di definire un output corretto. Lo schema che descrive la tecnica Acceptance Voting è rappresentato nella Figura 4.7.

Lo pseudocodice che rappresenta la tecnica Acceptance Voting è invece rappresentato dall'Algoritmo 4.6. Notiamo dallo pseudocodice che le N varianti implementate vengono eseguite subito in modalità concorrente. Ogni risultato viene verificato dal rispettivo test di accettabilità. Solitamente il test è semplicemente replicato, tuttavia è possibile fornirne una diversa implementazione per ogni modulo. I risultati che hanno superato il test (nello pseudocodice racchiusi tra le parentesi "[]") sono passati al voter che, sulla base del meccanismo di votazione a maggioranza, individua il risultato corretto. Il meccanismo di decisione ritorna FALSE se non individua l'output corretto e di conseguenza, non verificandosi la condizione dell'operatore if, viene sollevata un'eccezione. Si verifica un errore anche nel caso in cui nessun risultato dei moduli superi il proprio test di accettabilità. Viene invece prodotto automaticamente un output se un solo risultato supera il rispettivo test di accettabilità.

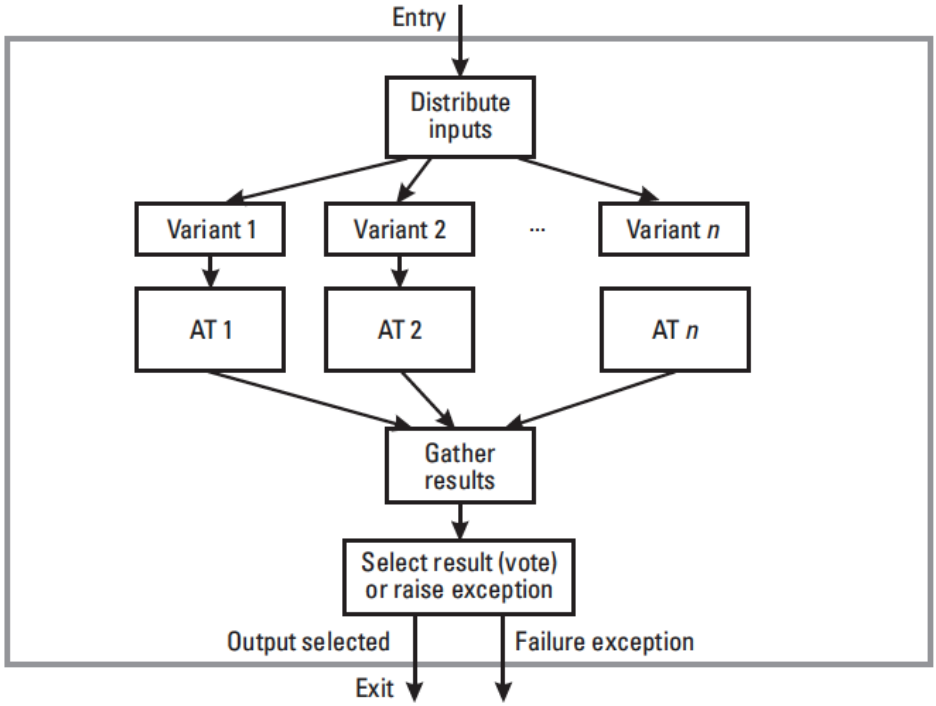

Figura 4.7: Funzionamento della tecnica AV[17]

Algoritmo 4.6 Pseudocodice della tecnica AV

```
run Variant 1, Variant 2, ..., Variant n
    ensure Acceptance Test 1 by Variant 1
    ensure Acceptance Test 2 by Variant 2
    ...
    ensure Acceptance Test n by Variant n
    [Result i, Result j, ..., Result m pass the AT]

if (Decision Mechanism (Result i, Result j, ..., Result m))
    return Result
else
    return failure exception
```

Considerazioni per la tecnica AV

Anche questa tecnica basandosi sulla tecnica design diversity e sul meccanismo di recupero in avanti, eredita da questi sia gli aspetti positivi che

negativi. Essendo poi una tecnica ibrida, come la Consensus Recovery Block, soffre della crescente complessità del meccanismo di tolleranza ai guasti. Implementando infatti sia il test di accettabilità, sia un meccanismo di accettazione, aumenta il rischio di incorrere in errori di progettazione e design. Sempre come la tecnica CRB soffre di lievi costi dovuti alla sincronizzazione dei risultati e alla ridondanza dei test di accettabilità. Rimangono tuttavia, anche in questo caso, costi di lieve entità e quindi la tecnica AV rimane una valida alternativa per l'implementazione della tolleranza ai guasti per i sistemi che necessitano di alta disponibilità.

4.1.7 N-version Programming TB-AT (NVP-TB-AT)

Di tutte le numerose varianti proposte per la tecnica N-Version Programming abbiamo già visto la tecnica Consensus Recovery Block (CRB) e la tecnica Acceptance Voting (AV). Un'altra la vedremo invece ora.
Questa tecnica è chiamata *N-Version Programming with tie-breaker and Acceptance Test (NVP-TB-AT)* e pur basandosi sulla tecnica NVP, presenta interessanti differenze che permettono di poterla considerare come una tecnica indipendente. Questa tecnica fu presentata da Ann Tai [28] come una tecnica in grado di valorizzare l'attributo della Dependability del sistema chiamato Performability (che abbiamo visto nel Capitolo 2). La tecnica NVP-TB-AT deriva dalla combinazione di altre due tecniche, entrambe modifiche della tecnica originale NVP: la tecnica NVP-TB e la tecnica NVP-AT; incorporandole entrambe è in grado di bilanciare efficientemente sia l'affidabilità che le performance del programma. Quando infatti la probabilità che si verifichino errori comuni ai moduli è bassa, la rapidità di calcolo fornita dal *tie-breaker* della tecnica NVP-TB, compensa le basse prestazioni causate dal *test di accettabilità*. Al contrario, quando la probabilità di difetti comuni è alta, il controllo supplementare fornito dal *test di accettabilità* riduce la probabilità che un errore non rilevato dal *tie-breaker* possa proseguire in output.

L'architettura su cui si basa la tecnica NVP-TB-AT è multiprocessore e le varianti del programma vengono eseguite sui diversi componenti. Uno di que-

sti componenti ospita anche l'esecuzione dell'algoritmo principale. Derivando dalla tecnica NVP è facile intuire come questa tecnica si basi sull'utilizzo di diverse versioni del programma, che vengono eseguite in modo concorrente. Si avvale poi dell'utilizzo di tre meccanismi decisionali: un *comparatore*, un *voter a maggioranza* e un *test di accettabilità*. Il controller dell'algoritmo non attende che tutti risultati delle varianti siano prodotti, anzi, una volta ricevuti i primi due risultati li invia subito al *comparatore*. Se i risultati corrispondono il meccanismo di confronto presume rappresentino il risultato corretto e li utilizza direttamente come output, mentre se non corrispondono allora il controller attende di ricevere dalle varianti l'ultimo risultato possibile. Una volta ricevuto il risultato, il controller utilizza il *voter a maggioranza* per individuare il risultato migliore. Questo risultato, prima di essere rilasciato come output, viene controllato con un *test di accettabilità*. Questo ulteriore controllo è volto ad evitare che risultati erronei, derivanti da guasti comuni alle varianti, possano essere valutati dal voter a maggioranza come risultato corretto e quindi forniti come output finale dell'esecuzione.
Lo schema che descrive la tecnica N-version Programming TB-AT è rappresentato nella Figura 4.8.

Lo pseudocodice che rappresenta la tecnica N-version Programming TB-AT nella versione a tre varianti è invece rappresentato dall'Algoritmo 4.7. Notiamo dallo pseudocodice che il medesimo input è passato alle varianti del programma. Il comparatore attende l'arrivo dei due risultati più veloci, li confronta e se corrispondono rilascia subito l'output. Altrimenti si attende l'arrivo del terzo risultato. Se il voter, in base ai tre risultati ricevuti, è in grado di individuare una maggioranza (in questo caso se due risultati su tre sono concordi) allora passa il risultato finale al test di accettabilità che ne controlla la correttezza. Se anche il test è soddisfatto allora viene rilasciato l'output finale. Se invece o il voter non è in grado di individuare un risultato migliore o il test di accettabilità non valida il risultato ricevuto, allora viene rilasciata un'eccezione.

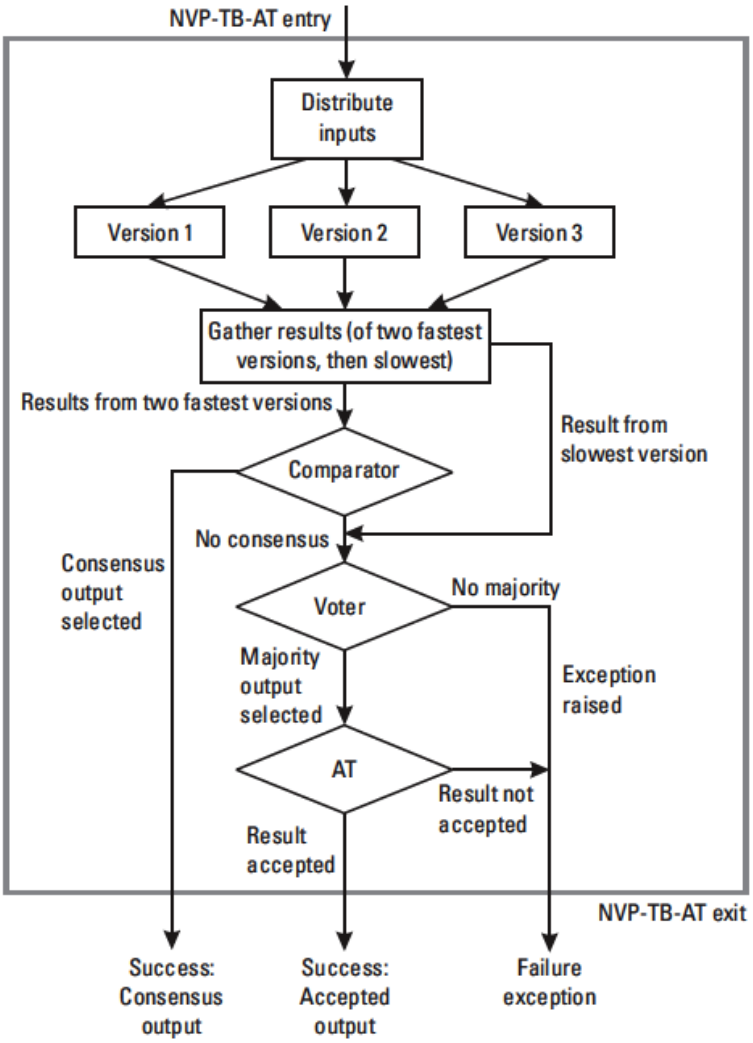

Figura 4.8: Funzionamento della tecnica NVP-TB-AT[17]

Algoritmo 4.7 Pseudocodice della tecnica NVP-TB-AT

```
run Variant 1, Variant 2, Variant 3
if (Comparator (Fastest Result 1, Fastest Result 2))
    return Result
else Wait (Last Result)
    if ( Voter (Fastest Result 1, Fastest Result 2, Last Result))
        if (Acceptance Test (Result))
            return Result
else failure exception
```

4.1.8 Self-Configuring Optimistic Programming (SCOP)

Nella trattazione delle tecniche di *design diversity* abbiamo visto l'importanza della ridondanza come tecnica per ottenere una serie di varianti dello stesso algoritmo e l'importanza di utilizzare i risultati prodotti da queste varianti per individuare un risultato finale ottimo. Uno dei problemi di queste tecniche era la necessità di avere a disposizione un numero sufficiente di componenti da far eseguire le varianti parallelamente.
Per quanto riguarda le tecniche di *data diversity* abbiamo visto invece l'utilizzo del test di accettabilità al fine di valutare la validità dei risultati ottenuti a partire dall'insieme di input realizzato tramite l'algoritmo di ri-espressione. Uno dei problemi di queste tecniche era il costo addizionale per ogni riesecuzione del programma con input diverso.

La tecnica *Self-Configuring Optimistic Programming* (SCOP) cerca di risolvere i problemi sopra riportati delle due categorie *data diversity* e *design diversity* facendo uso di una struttura dinamica in grado di adattarsi alle condizioni del sistema. La tecnica SCOP è sostanzialmente composta dai seguenti elementi: N varianti software, un meccanismo di aggiudicazione e un controller che gestisce la struttura dinamica del sistema.
L'esecuzione del programma è divisa in fasi. La fase iniziale sarà la fase "0". Può essere posto un limite massimo di fasi da eseguire in modo tale da simulare un timer. In ogni fase viene eseguito un sottoinsieme, definito *set*, di varianti. Il numero di varianti all'interno del *set* è definito come il minimo numero di varianti necessarie per soddisfare una determinata condizione (*delivery condition*) imposta dal controller. Prima di eseguire le varianti si controlla che ci siano componenti hardware disponibili ad ospitare le esecuzioni. Se non ci sono componenti disponibili allora l'esecuzione generale è momentaneamente sospesa in attesa di componenti che si liberino; se la quantità di componenti utilizzabili è maggiore di zero e minore del numero minimo di varianti necessarie a soddisfare la *delivery condition*, allora viene eseguito un numero di varianti uguale alla quantità di componenti disponibili. Un meccanismo di aggiudicazione di tipo *voter* controlla i risultati prodotti

dalle singole varianti: se trova una corrispondenza tra i risultati ottenuti allora produce il risultato finale, se invece non è in grado di definire un risultato ottimo allora viene avviata una nuova fase in cui verrà eseguito un altro set di varianti (dalla scelta delle varianti sono escluse quelle già precedentemente eseguite). Successivamente all'esecuzione del nuovo set, il meccanismo di aggiudicazione riesegue il controllo di corrispondenza sui nuovi risultati ottenuti più quelli precedentemente ricevuti. Questo processo viene iterato fino a quando o il meccanismo di aggiudicazione non è in grado di individuare un output corretto, oppure fino a quando non vengono esaurite le varianti a disposizione.
Lo schema che descrive la tecnica Self-Configuring Optimistic Programming è rappresentato nella Figura 4.9, dove:

- C Delivery condition.
- i Numero fase.
- N_p Numero componenti hardware disponibili.
- S_i Risultati fase i.
- T_d Timer.
- V_i Varianti selezionate nella fase i.

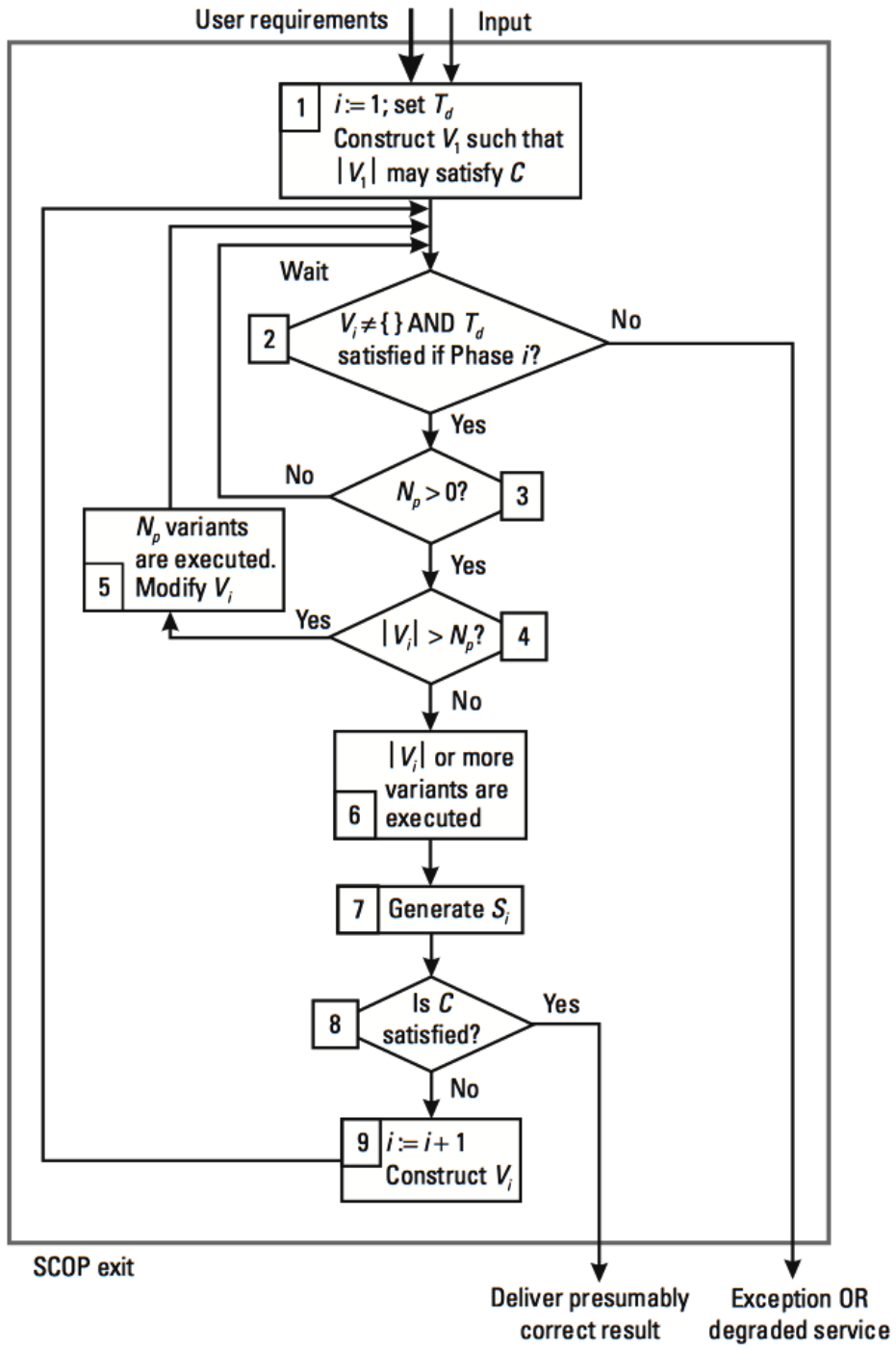

Figura 4.9: Funzionamento della tecnica SCOP[17]

Vediamo ora un piccolo esempio [29]. Prendiamo quindi:

- 3 varianti dello stesso programma.
- 2 componenti hardware a disposizione.
- Un meccanismo di decisione in grado di determinare il risultato migliore e una delivery condition che in questo caso sarà "2 su 3 risultati concordi".
- Un timer per limitare il tempo di esecuzione.

Lo schema risultante di questo esempio è riportato in Figura 4.10. Vediamo che in questo caso l'esecuzione generale è divisa in due fasi. Nelle prima fase vengono eseguite in parallelo le varianti 1 e 2. Una volta terminate le esecuzioni delle due varianti il meccanismo di decisione compara i risultati ottenuti. Supponiamo che questi non siano concordi (l'esecuzione sarebbe altrimenti terminata e l'output finale sarebbe rilasciato). Viene quindi avviata la seconda fase in cui viene eseguita la terza variante. Vengono poi comparati il risultato della terza variante e i risultati delle varianti precedenti. Se il meccanismo di decisione riconosce una corrispondenza tra il nuovo risultato e uno di quelli precedenti allora la delivery condition "2 su 3" è rispettata e viene ritornato il risultato finale, altrimenti viene ritornato errore.

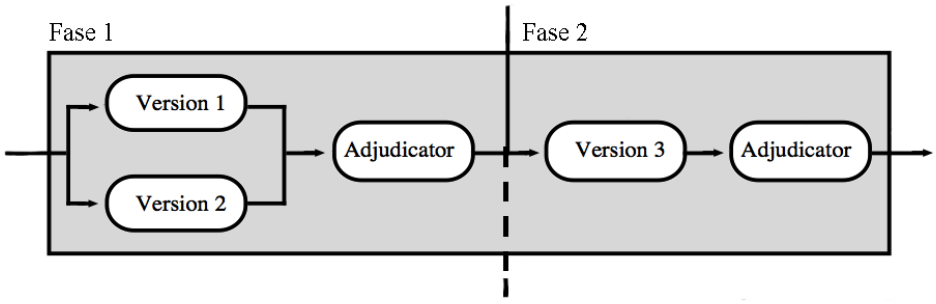

Figura 4.10: Esempio di funzionamento della tecnica SCOP[29]

Concludiamo la trattazione della tecnica *self-configuring optimistic programming* facendo notare l'importanza di uno dei casi limite del sistema: se

il numero di componenti hardware disponibile nella prima fase è superiore al numero di varianti del programma allora la logica della tecnica SCOP si riduce ad essere uguale alla logica della tecnica NVP.

4.2 Data Diversity

Per via di alcune limitazioni viste nella precedente trattazione delle tecniche appartenenti alla categoria *design diversity*, c'è stato bisogno di sviluppare altri metodi per l'implementazione della tolleranza ai guasti nei sistemi software. Questi metodi sono realizzati non per sostituire le tecniche di *design diversity* ma bensì per completarle [26].

A differenza delle tecniche di *design diversity* che utilizzano più varianti dello stesso programma, le tecniche di *data diversity* utilizzano una sola variante.
Quello che a volte accade è che il programma fallisca la propria esecuzione a causa di determinati valori di input. L'idea su cui si fonda questa tecnica è quindi quella di creare un insieme di input nello spazio dati del programma a partire da un unico dato di input, eseguire il medesimo modulo software su questo insieme di input, e usare un algoritmo di decisione per scegliere l'output risultante. L'approccio adottato dalle tecniche di *data diversity* è dunque quello di variare leggermente il dato di input, nel limite tollerato dal programma, in modo tale da evitare il fallimento dell'esecuzione.
Per ottenere l'insieme di input si usano algoritmi chiamati Data Re-expression Algorithms (DRA) che servono appunto ad ottenere una serie di dati logicamente equivalenti al dato originale. Le performance delle tecniche dipendono molto dalle performance degli algoritmi di ri-espressione del dato, che a loro volta dipendono molto dalla tipologia di applicazione in cui sono adottati. Lo sviluppo del DRA richiede una attenta analisi dei metodi di ri-espressione, valutando quale sia il più appropriato per ogni tipo di dato candidato alla riformulazione.

La tolleranza ai guasti sarà tanto più efficace quanto più l'algoritmo di ri-espressione sarà in grado, dato un input iniziale appartenente al *dominio di fallimento*, di ottenere una serie di nuovi input non appartenenti a tale dominio. Per dominio di fallimento si intende l'insieme degli input che, se utilizzati, causano un errore nel sistema [27]. È anche in questo caso preferibile un DRA semplice rispetto ad uno complesso poiché vi sono meno possibilità che contenga esso stesso guasti di progettazione. Poi non tutte le applicazioni possono adottare le tecniche di *data diversity*. Le ragioni di questa impossibilità posso essere varie: dal tipo di dato sui cui lavora l'applicazione che non può essere trattato con DRA, all'impossibilità di ottenere un algoritmo in grado di produrre input non appartenenti al dominio di fallimento.

Volendo dare una definizione formale del concetto di *data diversity* possiamo dire che l'algoritmo di ri-espressione dei dati R trasforma il dato originale di input x in un nuovo input y=R(x). Y è quindi funzione di x e in questo senso diciamo che approssima x o comunque contiene tutte le informazioni di x anche se sotto una forma diversa. Il programma P verrà poi eseguito sia sull'input x, ottenendo come risultato P(x), sia sull'input y, ottenendo P(y). La Figura 4.11 rappresenta lo schema che descrive il funzionamento generale delle tecniche di *data diversity*.

4.2.1 Retry Blocks (RtB)

Questa tecnica fu sviluppata da Amman and Knight [26]. La tecnica RtB è definita dinamica ed è considerata la tecnica complementare alla tecnica RcB, controparte nella categoria delle tecniche di *design diversity*. Per rendere il sistema tollerante ai guasti utilizza: test di accettabilità, recupero all'indietro, un data re-expression algorithm e opzionalmente un watchdog timer (WDT) che esegue un algoritmo di backup qualora l'algoritmo originale non dovesse produrre un risultato accettabile entro un determinato lasso

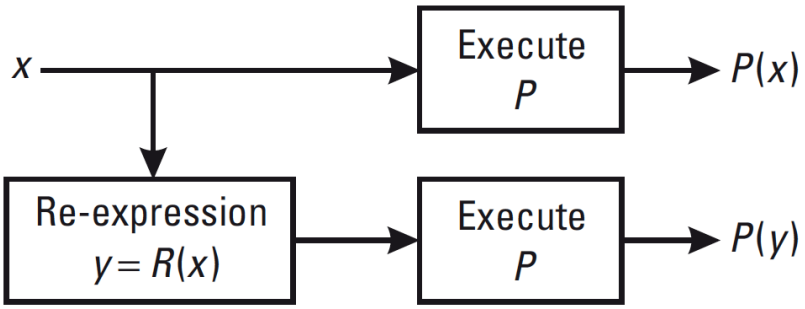

Figura 4.11: Schema generale della data diversity[12]

di tempo.

Per la comprensione della tecnica RtB partiamo dal presupposto che il sistema inizialmente si trovi in uno stato stabile, quindi privo di errori. Viene quindi avviato il controller dell'esecuzione generale della tecnica. Il programma è prima eseguito utilizzando come dato in ingresso l'input originale e il risultato ottenuto viene poi esaminato da un test di accettabilità che funziona allo stesso modo del test visto nella tecnica RcB. Se il risultato viene validato dal test di accettabilità allora la tecnica RtB termina la sua esecuzione, se invece il risultato non è validato allora l'input originale viene riformulato tramite l'algoritmo di data re-expression e il programma viene rieseguito utilizzando il nuovo input. Questo procedimento continua fintantoché non si ottiene un risultato validato dall'AT o fino a quando non scade il timer. Qualora si arrivasse allo scadere del timer senza aver ancora ottenuto un risultato validato dall'AT, verrebbe allora eseguito un algoritmo di backup, diverso dall'algoritmo originale, che a sua volta produrrebbe un risultato anch'esso da validare col test di accettabilità. Se anche questo risultato non dovesse passare il test allora viene sollevata un eccezione, altrimenti viene restituito il risultato e registrato un nuovo punto di ripristino del sistema. Lo schema che descrive la tecnica Retry Blocks è rappresentato nella Figura 4.12.

Lo pseudocodice che rappresenta la tecnica Retry Blocks è invece rappresentato dall'Algoritmo 4.8. Dallo pseudocodice notiamo come la tecnica RtB tenti prima di ottenere un risultato valido con il dato originale, e solo nel caso in cui questo non fosse possibile, ritenta l'esecuzione con dati riformulati utilizzando l'algoritmo DRA. A differenza delle tecniche di *design diversity* viene sempre usato lo stesso algoritmo di elaborazione. Vengono invece cambiati i dati di ingresso a tale algoritmo. Qualora non fosse ottenuto un risultato accettabile prima dello scadere del timer, si tenta l'ultima validazione con un algoritmo diverso, detto algoritmo di backup. Qualora anche il risultato di questo algoritmo fallisse la validazione del test di accettabilità, allora si verificherebbe un errore.

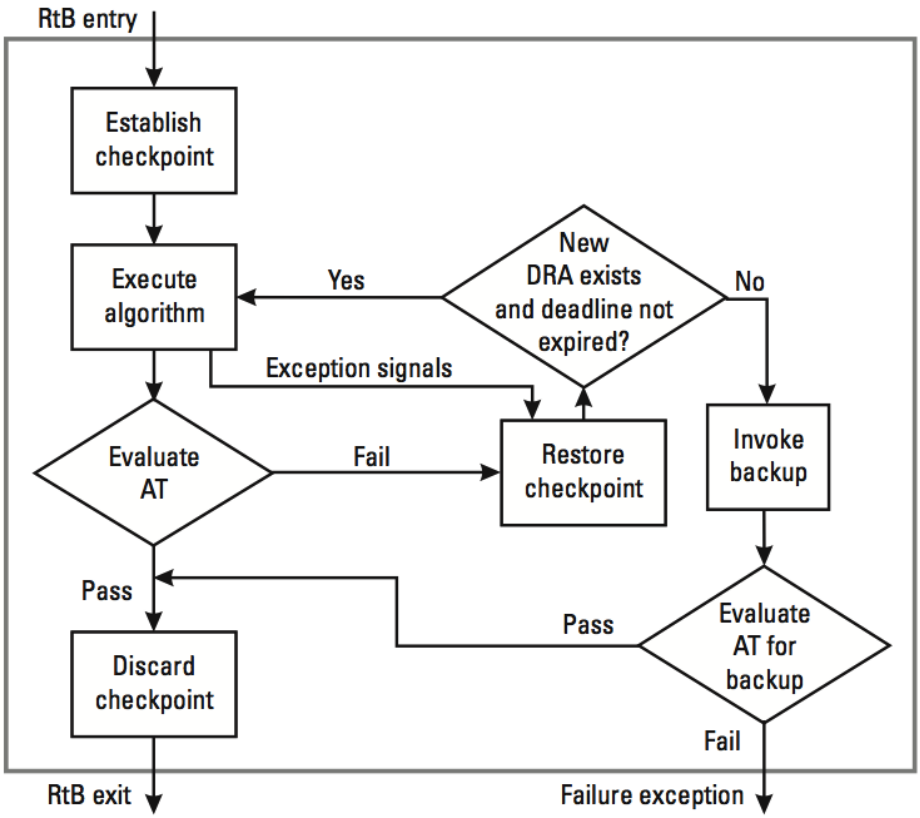

Figura 4.12: Funzionamento della tecnica RtB[17]

Algoritmo 4.8 Pseudocodice della tecnica RtB

```
ensure Acceptance Test
by Primary Algorithm(Original Input)
else by Primary Algorithm(Re-expressed Input)
else by Primary Algorithm(Re-expressed Input)
...
...
[WDT Deadline Violated]
else by Backup Algorithm(Original Input)
else failure exception
```

Considerazione per la tecnica RtB

Sono state proposte diverse modifiche all'algoritmo RtB. Una di questa prevede l'utilizzo di un cosiddetto *DRA execution counter*. Questa modifica consiste nell'introduzione di un contatore che, venendo incrementato ogni qual volta l'esecuzione fallisce, permette di tenere il conteggio delle volte in cui il programma fallisce con l'esecuzione sui dati originali. Il vantaggio di utilizzare il contatore così modificato è dato dalla possibilità di definire un numero limite di ri-espressioni dei dati di input. Avendo come limite il valore del contatore è quindi possibile non utilizzare il timer.
Tuttavia il non utilizzo di un *watchdog timer* pone il sistema a rischio fallimenti. Infatti errori di esecuzione interni al programma come ad esempio cicli infiniti, che prima potevano essere risolti con il timer, in questo caso non sarebbero più corretti.
Un'altra modifica possibile, da apportare alla tecnica base, è l'utilizzo non di un unico test di accettabilità del risultato ma bensì di una serie di test, come abbiamo visto per la tecnica RcB. Potrebbe essere implementato un test per valutare la validità dei dati passati in ingresso all'esecuzione, oppure un test dedicato esclusivamente all'esecuzione dell'algoritmo di backup. In quest'ultimo caso il test viene definito *acceptance test backup (ATB)*. Una ulteriore modifica della tecnica RtB riguarda le versioni di DRA. La tecnica RtB può essere infatti implementata sia usando un unico DRA, sia differenti DRA. Nel primo caso avremo un algoritmo che ogni volta riesprime l'input in modo diverso, nel secondo caso invece potremmo avere un algoritmo di

ri-espressione per ogni esecuzione secondaria.

Per quanto riguarda i problemi relativi alla tecnica Retry Blocks, possiamo dire che i problemi di tale tecnica sono molto simili a quelli della Recovery Block, essendo la corrispondente tecnica nella categoria *design diversity*. Il problema principale è quello della sequenzialità delle esecuzioni. Succede infatti che ogni volta che il programma è eseguito viene registrato un punto di ripristino e viene effettuato un controllo con test di accettabilità. Se il risultato supera il test allora l'esecuzione si arresta, se invece non supera il test si procede rieseguendo il programma ma con un input ri-espresso con DRA, quindi con un ulteriore salvataggio del punto di ripristino e un ulteriore controllo col test. Nel caso peggiore si ha che nessuna variante dei dati in input produce un risultato accettabile. Dato che non c'è parallelismo il costo di tutta l'esecuzione della tecnica sarà dato quindi dal singolo tempo di esecuzione del programma sommato al tempo di registrazione del punto di ripristino sommato al tempo di controllo della validità del risultato, il tutto moltiplicato per il numero di ri-espressioni effettuate. Oltretutto l'esecuzione concluderebbe senza ottenere un risultato corretto. Altro importante problema di questa tecnica è la temporanea indisponibilità del servizio. Difatti accade che quando un'esecuzione non produce un risultato valido si deve riportare il sistema all'ultimo stato stabile salvato. In questo lasso di tempo, definito recupero, il modulo non fornisce alcun servizio. È evidente come questi problemi siano inaccettabili per applicazioni ad alta disponibilità, e in generale per applicazioni di tipo real-time.

4.2.2 N-Copy Programming (NCP)

Anche questa tecnica fu sviluppata da Ammann e Knight [26]. A differenza della tecnica RcB non è dinamica ma bensì statica, per via del tipo di ridondanza utilizzato. La struttura hardware per eseguire questa tecnica è solitamente composta da N componenti, con i processi che vengono eseguiti contemporaneamente sui diversi componenti. I processi possono anche essere eseguiti in sequenza su un singolo calcolatore, ma è più frequente la prima

configurazione.
L'N-Copy Programming è da considerarsi complementare alla tecnica N-Version Programming vista nella categoria *data diversity*. Per ottenere un sistema tollerante ai guasti la tecnica NCP utilizza: un meccanismo di decisione di tipo voter, il meccanismo di recupero in avanti, un DRA singolo o multiplo e almeno 2 copie del programma. Il sistema utilizza il DRA per calcolare da 1 a N varianti del dato di input, che poi utilizza come dati in ingresso alle copie di programma eseguite in parallelo. Gli N risultati ottenuti vengono esaminati dal meccanismo di decisione di tipo voter, che seleziona il risultato migliore, ammesso che riesca ad individuarlo. Lo schema che descrive la tecnica N-Copy Programming è rappresentato nella Figura 4.13.

Lo pseudocodice che rappresenta la tecnica N-Copy Programming è invece rappresentato dall'Algoritmo 4.9. Notiamo dallo pseudocodice come prima venga invocato N volte l'algoritmo RDA in modo concorrente per ottenere gli N input diversi. Vengono quindi eseguite le N copie del programma sempre in modo concorrente, ognuna con in ingresso il rispettivo input prodotto nel passaggio precedente. Gli N risultati ottenuti vengono poi passati al meccanismo di decisione che se ritorna TRUE significa che ha individuato un risultato ottimo, se altrimenti ritorna FALSE significa che non è in grado di produrre un output ottimo e viene quindi sollevato un'eccezione.

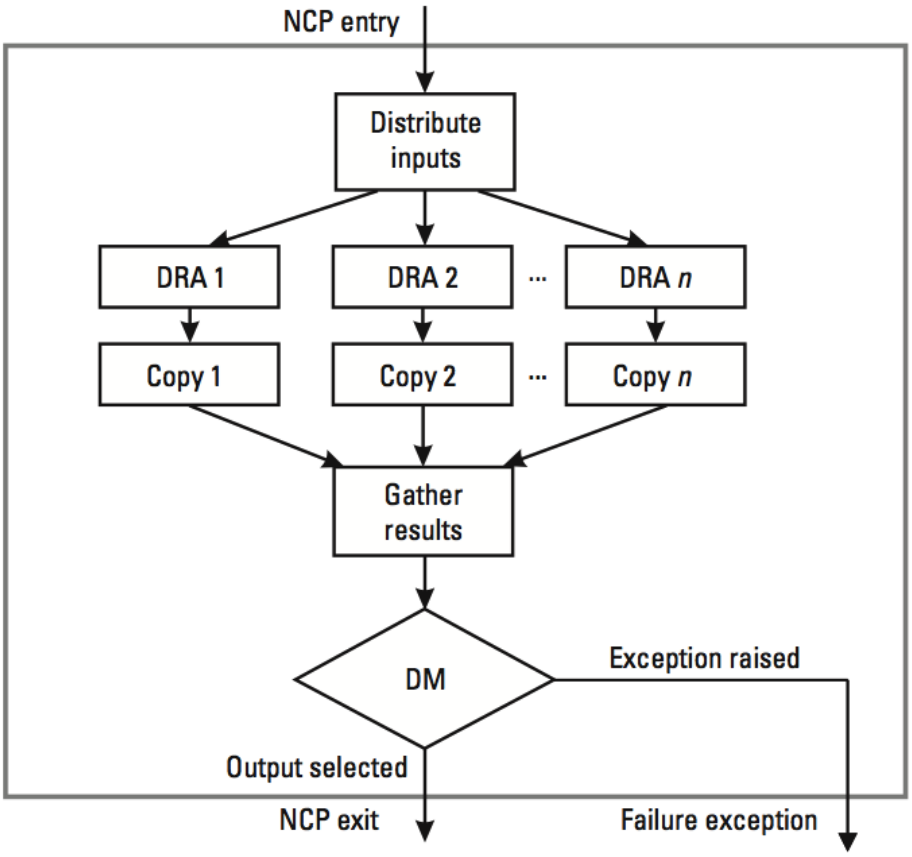

Figura 4.13: Funzionamento della tecnica NCP[17]

Algoritmo 4.9 Pseudocodice della tecnica NCP

```
run DRA 1, DRA 2, ..., DRA n
run Copy 1(result of DRA 1),
      Copy 2(result of DRA 2), ...,
      Copy n(result of DRA n)
if (Decision Mechanism (Result 1, Result 2, ...,Result n))
      return Result
else failure exception
```

Considerazioni per la tecnica NCP

Essendo la tecnica NCP basata sul concetto di *data diversity* e sul metodo di recupero in avanti, eredita da questi sia gli aspetti positivi sia quelli

negativi. Abbiamo poi detto che l'esecuzione della tecnica può avvenire sia in modalità concorrente, sia in modalità sequenziale su un unico componente. Nel primo caso il tempo totale necessario è pari al tempo per eseguire il DRA, sommato al tempo per eseguire il meccanismo di decisione, sommato al tempo impiegato dall'esecuzione del programma più lenta tra tutte quelle lanciate. Nel secondo caso invece i tempi di ogni singola esecuzione con input diverso si sommano tra di loro. É quindi preferibile la prima soluzione che permette un risparmio notevole di tempo.
Anche con la prima configurazione è tuttavia identificabile un problema: il meccanismo di decisione è costretto ad attendere che tutti i risultati delle singole esecuzioni siano disponibili. In questo modo è quindi vincolato al tempo di esecuzione del processo più lento. Una soluzione a questo problema è adottare un meccanismo di decisione in grado di lavorare sui risultati non appena almeno due di essi diventino disponibili. Questo gli permette l'identificazione di un output valido senza la necessità di aver ricevuto tutti i risultati. Una volta identificato il risultato ottimo è anche possibile interrompere i processi ancora in esecuzione.
Lo stesso meccanismo di decisione può poi contenere errori di progettazione, con la conseguente possibilità che vengano accettati risultati non validi e vengano invece rifiutati risultati corretti. Questo è un problema comune a tutte le tecniche basate sul meccanismo di decisione di tipo voting. Inoltre la qualità della tecnica dipende molto dalla validità del DRA implementato. Aspetto positivo per la tecnica NCP è che non essendo basata sul meccanismo di recupero all'indietro allora non è necessario interrompere la fornitura del servizio qualora un risultato non fosse validato. La tecnica N-Copy Programming si presta quindi ad essere una valida implementazione per la tolleranza ai guasti nei sistemi ad alta disponibilità.

4.3 Temporal Diversity

Le tecniche che abbiamo trattato nelle categorie *design diversity* e *data diversity* sono tutte tecniche che necessitano della ridondanza di informazione per poter eseguire le proprie strutture logiche. Le tecniche appartenenti alla categoria *temporal diversity* invece non necessitano della replicazione né dei componenti hardware né dei moduli software. Consideriamo quindi, in questo caso, la ridondanza come *ridondanza temporale* in quanto quello che sostanzialmente queste tecniche fanno è ripetere l'esecuzione dell'algoritmo in momenti diversi e verificare che i risultati ottenuti siano concordi. Se i risultati non sono concordi allora sappiamo che si è verificato un errore in una delle esecuzioni. Tale tecnica non può essere adottata all'interno dei sistemi di tipo real-time in quanto la ripetizione nel tempo dell'esecuzione va a svantaggio della rapidità di ottenimento di un risultato finale. È quindi utilizzata in quei sistemi dove il tempo non rappresenta un aspetto cruciale.

Una prima tipologia di *ridondanza temporale* consiste nella semplice riesecuzione in momenti differenti dell'algoritmo. Questo permette tuttavia di rilevare solo guasti di natura transiente. Ci si aspetta infatti che nelle esecuzioni successive ad una esecuzione errata, in cui cioè si è rilevato un risultato non corretto, i risultati ritornino ad essere corretti e concordi tra di loro.

La seconda tipologia prevede la riesecuzione con input diversi e questa modifica permette di rilevare anche guasti permanenti. La logica alla base di questo secondo tipo di ridondanza temporale è la seguente: al tempo T0 viene lanciata la prima esecuzione con in ingresso l'input originale che produce il risultato R1; successivamente, al tempo T1, viene lanciata una seconda esecuzione con l'input codificato in ingresso. Il risultato del calcolo viene poi decodificato ottenendo il risultato R2. Se R1 e R2 non corrispondono allora significa che si è verificato un errore. Lo schema che descrive il funzionamento della seconda tipologia di *ridondanza temporale* è rappresentato in Figura 4.14.

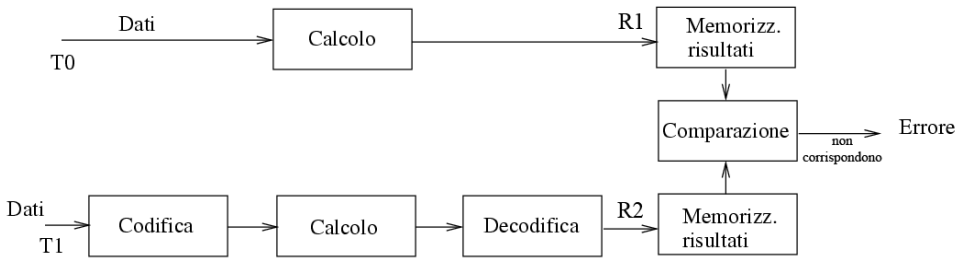

Figura 4.14: Schema della ridondanza temporale ad input diversi[12]

La codifica e la decodifica sono ottenute utilizzando o gli operatori di complementazione o gli shift logici [12].

Operatori di complementazione

Gli *operatori di complementazione* possono essere utilizzati se il sistema ha la proprietà di essere self-dual, il che significa che la funzione F applicata ad un input x produce come risultato F(x) = not(F(not(x))). Siamo quindi in un sistema in cui tramite l'operatore logico NOT si ottiene l'inversione del valore della variabile che può assumere solo uno tra i due valori 0 e 1. L'algoritmo viene eseguito a partire dall'input codificato. La decodifica applicata al risultato dell'algoritmo dovrebbe teoricamente produrre un risultato uguale al risultato ottenuto dall'esecuzione eseguita sull'input originale al tempo T0.

Shift logici

L'altra funzione che può essere utilizzata sono gli *operatori di shift*. Anche in questo caso al tempo T1 viene codificato l'input in ingresso all'algoritmo. Questa volta però la codifica consiste in uno shift a sinistra dei bit che compongo il dato e, di conseguenza, la decodifica in uno shift a destra. Qualora intervenga un guasto avremo che il bit errato sarà l'i-esimo nel risultato dell'esecuzione al tempo T0 e l' i-1-esimo nel risultato del calcolo al tempo T1. Anche in questo caso la comparazione di risultati ottenuti in momenti differenti permette di individuare eventuali guasti nel sistema.

4.4 Environment Diversity

L' Environment Diversity è uno tra gli approcci più recenti alla tolleranza ai guasti di livello software. L'idea alla base di questo approccio è che essendo gran parte dei guasti di natura transitoria allora una ri-esecuzione del sistema in uno stato differente può essere sufficiente a risolvere gran parte di questi errori.
Anche se questa concezione esiste da parecchi anni, solo ultimamente ha riguadagnato importanza. Questo perché, in passato, si considerava il software come un prodotto non soggetto ad usura. Questa idea col tempo si è rivelata errata e si è scoperto invece che l'usura del prodotto software è un fenomeno molto più rilevante di quanto si era pensato. Ciò vale per la gran parte dei prodotti software. Le cause di questa usura sono da ricercare in guasti di lieve entità che possono avere come risultato, per esempio, il mancato rilascio di determinate porzioni di memoria o di risorse di sistema. Questi tipi di guasti vengono definiti *guasti da invecchiamento del sistema (aging faults)* e sono guasti che rientrano nella più vasta categoria dei *guasti sfuggenti (elusive faults)*, essendo il più delle volte guasti indeterminabili. Sono caratterizzati da un lasso di tempo molto ampio che intercorre tra il guasto che origina la catena di errori e il fallimento del sistema, ed è proprio per questa sua caratteristica che risulta difficile risalire alla causa del guasto. I *guasti da invecchiamento* sono guasti temporanei che si verificano solitamente in seguito all'utilizzo continuo del sistema. L'accumulo nel tempo di questi guasti porta ad una notevole riduzione delle prestazioni del sistema che tende ad assumere un assetto fragile e a rischio di fallimento.

4.4.1 Ringiovanimento del software

La tecnica chiamata *ringiovanimento del software (software rejuvenation)* consiste nel riavviare periodicamente il sistema in modo tale da riportarlo al suo stato iniziale e far sì che tutte le risorse siano nuovamente sfruttabili. Questi riavvii programmati tuttavia comportano alcuni problemi. Il riavvio del sistema ha infatti come conseguenza il momentaneo disservizio dello stesso, che dura quanto il tempo necessario al sistema per ritornare operativo.

Questo lasso di tempo in cui il sistema è fuori servizio viene chiamato *downtime*. Nel considerare l'adozione della tecnica del ringiovanimento software bisogna quindi valutare quali siano i costi che un momentaneo disservizio del sistema comporta, tenendo presente che un downtime volontario è in grado di evitare eventuali downtime futuri involontari. Ci si potrebbe chiedere quale possa essere l'utilità di procurare un downtime per evitarne uno futuro. La risposta è semplice: essendo il downtime volontario una operazione programmata dallo sviluppatore o chi delegato, si presume che questa venga effettuata in determinati momenti di inattività del sistema senza interferire con la fornitura del servizio; il downtime involontario invece, non essendo prevedibile, potrebbe accadere in qualsiasi momento interrompendo la funzione del sistema.
Si considera pertanto il costo del downtime volontario minore del costo del downtime involontario. La tecnica di ringiovanimento del software deve quindi essere considerata una tecnica preventiva in grado di spezzare l'eventuale catena di guasti originata nel sistema.

Livello di ringiovanimento

Il ringiovanimento può avvenire sia ad un livello di programma sia ad un livello di processore. A livello di programma ci si limita a sospendere l'esecuzione del programma, che viene poi eliminato dallo stato del sistema, vengono reinizializzate le strutture dati e infine viene riavviato il programma. A livello di processore invece il riavvio coinvolge tutto il componente e per questo motivo è utile adottare questa tecnica solo qualora il sistema sia dotato di più processori in modo tale che in un determinato istante solo una parte di questi sia fuori servizio.

Tempi di ringiovanimento

Le tempistiche con cui eseguire il ringiovanimento del software possono essere regolari o basate su previsioni [7].
Nell'utilizzare il ringiovanimento del software basato sul riavvio ad intervalli di tempo regolari, si deve prima di tutto definire l'intervallo di tempo da

rispettare. Per fare questo c'è bisogno di studiare i costi e i benefici di un eventuale valore scelto. Ipotizzando che il costo del riavvio sia nullo mentre il beneficio consista nell'evitare il verificarsi di un qualsiasi guasto, è facile intuire che il caso migliore sarebbe rappresentato dal continuo riavvio del software. Tuttavia questo è impossibile. Difatti il costo del riavvio è sempre maggiore di zero e non può mai essere nullo. Vediamo quindi un semplice modello matematico che ci permetta di capire meglio quale sia l'intervallo di tempo migliore per eseguire il riavvio.
Consideriamo i seguenti valori:

- t è la variabile *tempo*.
- $N(t)$ è il numero di errori che si verificano nell'intervallo di tempo t senza ringiovanimento.
- Ce è il costo del verificarsi di un errore.
- Cr è il costo dell'esecuzione del ringiovanimento.
- P è il tempo tra un ringiovanimento e un altro.

Avremo quindi che il costo totale sostenuto nell'intervallo di tempo P, che è appunto l'inter-tempo tra un ringiovanimento e un altro, sarà dato da:

$$C_{ringiov}(P) = N(P) \cdot C_e + C_r \tag{4.1}$$

Quindi nell'unità di tempo il costo sostenuto sarà:

$$C_{tasso}(P) = \frac{C_{ringiov}(P)}{P} = \frac{N(P) \cdot C_e + C_r}{P} \tag{4.2}$$

Vediamo allora, sulla base di questa formula, due casi. Per il primo caso consideriamo il programma con un tasso di errore λ costante per tutto il corso della sua esecuzione. Abbiamo quindi che $N(P) = \lambda \cdot P$ se lo andiamo

ad inserire nella Formula (4.2) allora otteniamo $C_{tasso}(P) = \lambda \cdot C_e + C_r/P$. Vediamo che in questo caso per minimizzare il costo dell'unità di tempo si deve porre$P = \infty$, il che significa non eseguire mai la tecnica di ringiovanimento. Se infatti il tasso di errore è costante, l' eseguire il ringiovanimento del software comporta solo ulteriore costo.
Per il secondo caso proviamo allora a considerare un tasso di errore non costante ma che cresce esponenzialmente in relazione all'inter-tempo delle operazioni di ringiovanimento. Questo è dei due il caso che sicuramente rispecchia meglio la realtà. Prendiamo quindi $N(P) = \lambda \cdot P^n$ e andiamolo ad inserire nella Formula (4.2). Otteniamo allora $C_{tasso}(P) = \lambda \cdot P^{n-1} \cdot C_e + C_r/P$. Per minimizzare tale quantità poniamo la derivata uguale a 0 e quindi otteniamo che il valore ottimo di P non è più ∞ ma bensì:

$$P = \left(\frac{C_r}{(n-1) \cdot \lambda \cdot C_e} \right)^{\frac{1}{n}} \tag{4.3}$$

Da questa formula notiamo che P è espresso in relazione al rapporto C_r/C_e e in relazione al valore n. In questo caso per definire P bisogna quindi conoscere le quantità Cr , Ce e $N(t)$. Queste possono essere ottenute tramite l'applicazione della tecnica di *previsione dei guasti* vista nel capitolo precedente in grado appunto di stimare questi valori.

Il *ringiovanimento basato su previsioni* [30] è invece in grado di impostare dinamicamente l'inter-tempo tra un riavvio e l'altro monitorando il comportamento del sistema. Questa tecnica controlla varie caratteristiche del sistema come ad esempio la quantità di memoria allocata. Sulla base dei valori del sistema raccolti tenta di prevedere il verificarsi del successivo fallimento del sistema. La tecnica del ringiovanimento basato su previsioni interverrà quindi riavviando il sistema un istante prima del momento in cui è stato previsto il prossimo fallimento. Per fare questo è importante che il software che implementa la tecnica del ringiovanimento abbia accesso alle informazio-

ni del sistema che gli permettano di effettuare le previsioni. Se si tratta di un programma parte nativa del sistema allora l'accesso alle informazione è garantito, se si tratta invece di un programma applicato successivamente al sistema originale allora è vincolato ai privilegi di accesso che gli sono stati concessi e potrà quindi recuperare le informazioni necessarie solo attraverso le interfacce fornite dal sistema stesso. Linux per esempio fornisce comandi per accedere ad informazioni riguardo l'utilizzo del processore, la memoria primaria, i dispositivi di input/output, le interfacce di rete ecc. Una volta recuperati i dati il programma è quindi in grado di effettuare le proprie stime.

Capitolo 5

Costi della tolleranza ai guasti

Conferire al sistema affidabilità implementando le tecniche di tolleranza ai guasti comporta un prezzo da pagare in termini di tempo di esecuzione, occupazione di memoria e soprattutto costi di realizzazione del sistema.

Come abbiamo visto nel capitolo precedente, per poter gestire l'eventuale verificarsi di un errore è necessario ricorrere al concetto di diversità. Tale diversità può esplicarsi in numerose forme, come ad esempio la realizzazione di diverse varianti del programma oppure l'esecuzione del programma in diversi istanti di tempo. In qualsiasi caso l'introduzione della diversità comporta un costo che può consistere nell'adozione di un maggior numero di componenti qualora l'implementazione della diversità richiedesse l'esecuzione parallela delle varianti, oppure nell'aumento del tempo necessario per concludere il calcolo del risultato qualora fosse richiesta l'esecuzione sequenziale delle varianti. Questi aspetti vanno valutati nel momento in cui si decide di rendere una applicazione tollerante ai guasti. Bisogna comunque tenere presente che la natura stessa dell'applicazione può determinare l'impossibilità di adottare alcune tecniche di tolleranza ai guasti. Per fare un esempio prendiamo le applicazione real-time: essendo fondamentale la velocità di calcolo del risultato, è poco probabile che queste possano implementare tecniche di tolleranza ai guasti basate sull'esecuzione sequenziale delle variati come la tecnica RcB. La spesa da affrontare consisterà piuttosto nel dotare il sistema di un numero

di processori sufficiente da permettere l'esecuzione parallela delle varianti.

Bisogna poi considerare l'aspetto dei costi di realizzazione. É indubbio infatti che sviluppare differenti versioni dello stesso programma comporti una spesa maggiore. Tuttavia quantificare questo costo supplementare non è semplice e varia a seconda dei casi. Osservazioni dirette allo sviluppo di alcuni sistemi software [31] hanno evidenziato come il costo di realizzazione di un sistema con N varianti sia inferiore al costo di realizzazione di N sistemi composti di una variante sola. Questo è dato dal fatto che non tutte le parti di un sistema necessitano di essere tolleranti ai guasti; inoltre ci sono attività, come la definizione dei requisiti e la realizzazione dei test, che non necessitano di essere duplicate poiché sono comuni a tutte le varianti. Queste osservazioni tuttavia non ci danno una misura precisa di quanto l'adozione della tolleranza ai guasti possa incidere sul costo di realizzazione del sistema e quale sia il rapporto tra il costo di realizzazione del software tollerante ai guasti e il costo di realizzazione dello stesso software ma non tollerante ai guasti.

5.1 La distribuzione dei costi nel software non tollerante ai guasti

Laprie [22] ha esaminato i costi di implementazione delle tecniche di tolleranza ai guasti di tipo design diversity in alcuni sistemi e ha realizzato un modello in grado appunto di definire il rapporto tra i costi di realizzazione dello stesso sistema dotato e non dotato di tolleranza ai guasti.

Partendo dal presupposto che la diversità incide differentemente sulle attività che compongono il processo di sviluppo software, Laprie ha inizialmente osservato i costi delle diverse fasi del ciclo di vita del software. Considerando poi che la tolleranza ai guasti è utilizzata principalmente in applicazioni che richiedono alti livelli di affidabilità, ha quindi definito dei fattori moltipli-

cativi al fine di ottenere una distribuzione dei costi che meglio rappresenta questo tipo di applicazioni nella versione non tollerante ai guasti.
Lo schema che riassume i costi delle attività del ciclo di vita è rappresentato nella Tabella 5.1. Notiamo il fattore moltiplicativo 1.8 applicato alla fase *Verifica e Validazione* che esprime l'importanza di questa fase nello sviluppo di applicazione altamente affidabili. La distribuzione dei costi è espressa nelle ultime due colonne: nella colonna *development and maintenance* tenendo conto della ripartizione del peso della manutenzione sulle altre attività mentre nella colonna *development* la manutenzione non viene considerata. Queste ultime due colonne sono ottenute dai valori presenti nelle prime colonne *Life-Cycle Cost Breakdown* e *Multipliers for Critical Applications*.

Activities		Mnemonic	Life-Cycle Cost Breakdown [Zel 79]	Maintenance Cost Breakdown [Ram 84]			Multipliers for Critical Applications [Boe 81]	Cost Distribution	
				Corrective	Adaptive	Perfective		Development	Development and Maintenance
Development	Requirements	R	3 %				1.3	8 %	6 %
	Specification	S	3 %				1.3	8 %	7 %
	Design	D	5 %	20 %	25 %	55 %	1.3	13 %	14 %
	Implementation	I	7 %				1.3	19 %	19 %
	V & V	V	15 %				1.8	52 %	54 %
Maintenance			67 %	100 %				100 %	100 %
			100 %						

Tabella 5.1: I costi del ciclo di vita del software[32]

5.2 I costi del software tollerante ai guasti

Per determinare i costi del software tollerante ai guasti è necessario utilizzare fattori moltiplicativi in grado di esprimere il variare dei costi. Alcuni fattori esprimono l'aumento dei costi dovuto all'introduzione di punti di decisione e meccanismi di decisione (DM). Altri fattori sono in grado di esprimere il diminuire dei costi della fase di verifica e validazione dovuto alla comunan-

za di tale fase nelle diverse varianti.

Laprie ha definito tali fattori nel modo seguente:

- **r** è il fattore moltiplicatore associato ai punti di decisione ed è tale che $1 < r < 1.2$. I punti di decisione sono definiti espressamente nelle specifiche e definiscono quando (e su quali dati) le decisioni devono essere prese.

- **s** è il fattore moltiplicatore associato ai meccanismi di decisione (il meccanismo che controlla i dati delle varianti). Quando il *DM* consiste in un voter allora s è tale che $1 < s < 1.1$ (come nel caso delle tecniche NVP e NSCP), quando invece consiste in un test di accettazione allora è tale che $1 < s < 1.3$ (come nel caso della tecnica RcB).

- **u** è la quantità di attività di testing eseguita sull'insieme delle varianti ed è tale che $0.2 < u < 0.5$.

- **v** è la quantità di attività di testing eseguita su ogni singola variante ed è tale che $0.3 < v < 0.6$.

- **w** è il fattore di riduzione del costo della fase di testing condivisa dalle diverse varianti, ed è tale che $0.2 < w < 0.8$.

L'espressione con cui Laprie definisce il rapporto tra il costo del software tollerante ai guasti e il software non tollerante ai guasti è quindi la seguente:

$$C_{FT}/C_{NFT} = R+rsS+[Nr+(s-1)](D+I)+r\{us+(1-u)N[vw+(1-v)]\}V$$

dove N è il numero di varianti implementate nella versione tollerante ai guasti e R, S, D, I, V sono rispettivamente le percentuali di distribuzione dei costi delle attività viste nella Tabella 5.1.

Sulla base dei valori presentati nella Tabella 5.1 è quindi possibile, applicando la formula sopra indicata, ottenere i valori risultanti dal rapporto C_{FT}/C_{NFT}.

Nella Tabella 5.2 vediamo i valori ottenuti per alcune delle tecniche di design diversity che abbiamo visto nel Capitolo 4.

Number of faults tolerated	Fault Tolerance Method		N	$\left(\frac{C_{FT}}{C_{NFT}}\right)_{min}$	$\left(\frac{C_{FT}}{C_{NFT}}\right)_{max}$	$\left(\frac{C_{FT}}{C_{NFT}}\right)_{av}$	$\left(\frac{C_{FT}}{N\,C_{NFT}}\right)_{av}$
1	Recovery Blocks		2	1.33	2.17	1.75	.88
	N Self-Checking Programming	Acceptance Test					
		Comparison	4	2.24	3.77	3.01	.75
	N-Version Programming		3	1.78	2.71	2.25	.75
2	Recovery Blocks		3	1.78	2.96	2.37	.79
	N Self-Checking Programming	Acceptance Test					
		Comparison	6	3.71	5.54	4.63	.77
	N-Version Programming		4	2.24	3.77	3.01	.75

Tabella 5.2: Il rapporto tra i costi del software FT e del software NFT[32]

I risultati di questo modello ci permettono di affermare, per esempio, che sviluppare un software tollerante ai guasti basato sulla tecnica RcB a tre varianti comporta un costo pari al 79 % del costo da sostenere per sviluppare tre versioni indipendenti dello stesso software non tollerante ai guasti; L'implementazione della tecnica RcB a tre varianti comporta poi un costo pari a 2.37 volte il costo da sostenere per sviluppare un unico software non tollerante ai guasti. Questo modello ci permette quindi di conoscere qual è la spesa supplementare che si dovrebbe affrontare qualora si decidesse di implementare all'interno del software la tolleranza ai guasti utilizzando alcune delle tecniche appartenenti alla categoria design diversity.

Capitolo 6

Caso di studio: Fly-By-Wire

Il caso di studio scelto riguarda la trattazione della tolleranza ai guasti implementata nel sistema di controllo di volo denominato *Fly-By-Wire (FBW)*. Questo sistema venne introdotto alla fine degli anni '80 su alcuni aerei militari, tra cui il caccia General Dynamics F-16. Venne successivamente adottato su velivoli ad uso civile, primo fra tutti l' Airbus A-320.
Fino agli anni '90 i sistemi di controllo utilizzati negli aeromobili prevedevano una connessione meccanica diretta tra i comandi di controllo e gli organi di movimento. Con l'introduzione del sistema Fly-By-Wire i collegamenti meccanici furono sostituiti da connessioni di tipo informatico. L'Airbus A-320, in quanto primo velivolo ad utilizzare questo sistema, ha rappresentato un importantissimo salto tecnologico nella storia dei sistemi di controllo di volo[33].
Per spiegare il funzionamento del sistema Fly-By-Wire prenderemo quindi come riferimento proprio il sistema che equipaggia la famiglia di velivoli Airbus A-320/330/340. Vedremo successivamente l'implementazione dello stesso sistema di controllo ma in un altro aereo: il Boeing 777.

6.1 Airbus A-320

Il sistema FBW fu progettato per dotare i sistemi di controllo degli aerei di un elevato livello di affidabilità e sicurezza. Queste caratteristiche sono

prodotte dall'architettura degli elaboratori utilizzati, dall'implementazione delle tecniche di tolleranza ai guasti sia hardware che software, dal monitoraggio degli eventi del sistema e dalla protezione del sistema dall'azione di agenti esterni (ad esempio fulmini).
Prima degli anni '90 i velivoli tradizionali prevedevano una trasmissione meccanica dei comandi, impartiti dal pilota agli organi di controllo. Il sistema di controllo Fly-By-Wire (vedi Figura 6.1) prevede invece un controllo di tipo elettronico che si traduce, tramite opportuni dispositivi, in movimenti idraulico-meccanici. I comandi del pilota sono quindi interpretati dai computer di bordo che trasformano la manovra compiuta dal pilota in azioni meccaniche da far svolgere agli *attuatori*, organi responsabili del movimento fisico delle componenti dell'aereo.

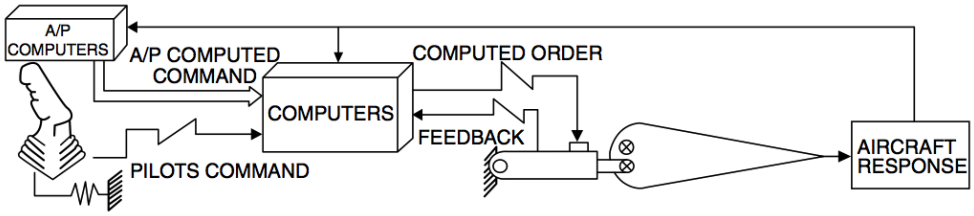

Figura 6.1: Sistema di controllo ibrido meccanico-elettronico[34]

L'adozione del sistema Fly-By-Wire fa sì che i movimenti fisici delle componenti non siano più una semplice trasposizione meccanica dei comandi impartiti dal pilota sulla cloche; cosa che sarebbe invece successa in tutti i sistemi di controllo di natura meccanica. Il sistema software permette infatti di interpretare i comandi ricevuti sulla base di regole di controllo dei movimenti che permettono di dotare il velivolo di una maggiore stabilità. Una brusca manovra compiuta dal pilota, per esempio, sarà effettuata dal sistema in misura tale da permettere il mantenimento della stabilità dell'aereo.

Hardware

La configurazione hardware del sistema FBW è composta sostanzialmente da un sistema di alimentazione, un circuito idraulico-meccanico e un sistema

informatico. Il sistema di alimentazione si occupa di fornire energia agli altri circuiti/sistemi ed è solitamente composto da un generatore attivo e uno di riserva. Qualora il primo generatore si guastasse entra subito in servizio il secondo. Il circuito idraulico è replicato in 3 sotto-circuiti identici tra loro, ognuno dei quali è in grado da solo di controllare tutto l'aereo permettendo il movimento delle parti meccaniche. Il sistema informatico fornisce invece il supporto all'esecuzione del software che controlla il velivolo. Prevede l'utilizzo di cinque computer tutti con pieno controllo delle funzionalità dell'aereo. Altri due computer sono invece adibiti alla funzione di pilota-automatico. In tutto si hanno quindi sette computer di bordo. Esistono due tipi di computer: il primo è chiamato ELAC (ELevator and Aileron Computers) e il secondo SEC (Spoiler and Elevator Computers). Questi computer sono progettati e realizzati indipendentemente da due diversi team di sviluppo in modo tale da conferire al sistema la capacità di tollerare i guasti di naturale progettuale. Ognuno di questi sette computer contiene al proprio interno due unità separate e speculari. Ogni unità può essere considerata un computer a se stante. La prima unità viene definita *canale di controllo (Command)*, mentre la seconda unità *canale di monitoraggio (Monitor)*. Il *canale di controllo* fornisce il servizio vero e proprio del sistema mentre il *canale di monitoraggio* serve a controllare la validità dell'esecuzione della prima unità. L'architettura dei computer è rappresentata in Figura 6.2 .

In entrambi i tipi di computer (ELAC e SEC) le due unità mantengono comunque la stessa architettura: un processore, una memoria principale, un circuito di input/output, un gruppo di continuità e il software dedicato. Nei modelli di Airbus A-330 e A-340 vengono usati altri computer: uno denominato *FCPC (flight control primary computers)* e uno chiamato *FCSC (flight control secondary computers)*. Per la maggior parte delle funzionalità equivalgono ai computer utilizzati sull'Airbus A-320.

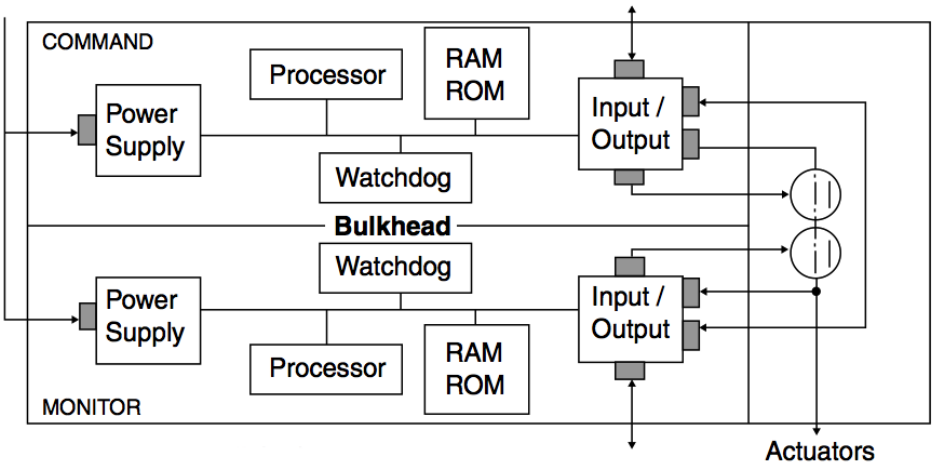

Figura 6.2: Architettura interna dei computer ELAC e SEC[34]

Software

Le specifiche del software da implementare all'interno del computer furono redatte dalla stessa Airbus che le fornì poi ai due team indipendenti. Una volta realizzati, i software vennero validati tramite opportuni tools per verificare l'effettiva conformità alle specifiche, che possono essere considerate una interfaccia tra i meccanismi fisici dell'aereo e i programmatori del software per comandarli.
In aggiunta alla diversità ottenuta tra i due computer, avendoli fatti sviluppare da due team diversi, si ha poi, all'interno del singolo computer, la diversità tra i due software che controllano il *canale di controllo* e il *canale di monitoraggio*. Come risultato si hanno quindi 4 varianti del software tutte differenti tra di loro, il che permette di minimizzare il rischio che errori comuni alle implementazioni degenerino in fallimenti di sistema.

In un dato momento l'esecuzione dei comandi è gestita da un solo computer dei cinque disponibili; quelli inattivi vengono definiti *hot spares*. La tolleranza ai guasti a livello software è fornita dall'utilizzo della tecnica N-Version Programming. Il risultato dell'esecuzione dell'unità *canale di controllo* viene infatti confrontato, tramite l'utilizzo di un *comparatore*, con il risultato

dell'unità *canale di monitoraggio*. Se i risultati corrispondono allora il primo risultato viene validato e utilizzato per compiere le operazioni richieste. Se invece i risultati non corrispondono allora significa che si è verificato un errore dovuto ad una qualche forma di guasto (hardware o software). È quindi richiesto ad un secondo computer di riattivarsi dallo stato di stand-by ed eseguire il calcolo. Questo processo di risveglio e passaggio di consegna dell'esecuzione è chiamato *riconfigurazione*. Cambiare computer qualora i due risultati non siano concordi permette di rimediare sia ad un eventuale guasto hardware sia ad un guasto software. Infatti la *riconfigurazione* prevede la scelta di un computer il cui produttore sia diverso da quello attualmente utilizzato. In questo modo viene effettuato un cambio sia hardware che software.
Un motivo della necessità di 4 computer sostitutivi è rappresentato dal problema dei guasti latenti. Questi infatti possono rimanere nascosti nei software di controllo dell'aereo per molto tempo e una volta che emergono c'è bisogno, soprattutto nel caso di sistemi real-time come questo, di avere a disposizione hardware e software alternativi in quantità sufficiente da poter rimediare all'errore verificatosi. Un esempio possono essere i guasti che interessano il *canale di monitoraggio*; errori che non possono essere scoperti fino a quando non si verifica un errore sul canale stesso.
Quando un errore viene rilevato e l'esecuzione è quindi passata ad un altro computer, bisogna decidere se mantenere attivo il computer su cui è avvenuto l'errore o se scollegarlo dal sistema. Viene quindi verificato se il computer ricommette il medesimo errore dopo un determinato lasso di tempo. Questo meccanismo si basa sul concetto di *temporal diversity*. Nel fare questo bisogna fare attenzione ad utilizzare un margine di tolleranza dell'errore e un tempo di riesecuzione, né troppo grandi né troppo piccoli. Scegliere, infatti, due valori troppo piccoli per queste due misure potrebbe portare ad escludere dal sistema un computer in realtà ancora funzionante, mentre invece scegliere due valori troppo grandi potrebbe impedire al sistema di rilevare un errore. Tale implementazione della tecnica di tolleranza ai guasti N-Version Programming a 2 varianti, unita al rigore del processo di progettazione del software e alla ridondanza hardware adottata, è quindi sufficiente a dotare il

sistema di un livello di affidabilità ottimo.

6.2 Boeing 777

Il sistema di controllo Fly-By-Wire, dopo essere stato utilizzato per la prima volta dalla Airbus, venne successivamente adottato anche dalla casa produttrice di aeromobili Boeing. Il primo velivolo della casa ad adottare questo sistema di controllo fu il Boeing 777. Per la realizzazione di questo modello si decise di apportare alcune modifiche sostanziali al sistema Fly-By-Wire, differenziandolo leggermente dall'architettura adottata nei modelli Airbus. Il funzionamento semplificato del sistema è così espresso [35]: i comandi del pilota vengono gestiti dal *controllo elettronico degli attuatori (ACE)* che li interpreta, li trasforma da segnale analogico a digitale e li invia ai *computer di volo principale (PFC)*. Il PFC, una volta ricevuti i comandi, li elabora e restituisce il risultato all'unità ACE che li riconverte da segnale digitale ad analogico e li invia agli attuatori che eseguono il movimento meccanico.
Sostanzialmente il sistema informatico di controllo è composto da 3 componenti: PFC, ACE e i bus di controllo che li collegano. Peculiarità del sistema di controllo Boeing, che lo differenzia dal sistema implementato nell'Airbus A-320, è che questi tre componenti sono triplicati nell'architettura.
Vediamo nel dettaglio il componente *computer di volo principale*. Il PFC è il computer che, come nel caso del computer ELAC e del computer SEC nell'Airbus A-320, fornisce il supporto al software di controllo del velivolo e si occupa di interpretare i comandi impartiti dal pilota, verificarne la validità e tradurli in operazioni reali da effettuare. Come detto il sistema è triplicato, quindi possiede tre computer PFC, i quali contengono a loro volta tre sotto-unità che possono essere considerate dei computer a sé stanti. In tutto sono quindi presenti nove computer. Ognuna delle sotto-unità è collegata al triplice bus di controllo, da cui riceve i dati in ingresso, ed è dotata del proprio generatore, del proprio processore e del proprio software di controllo. Da notare che questi (generatore, processore e software) sono diversi a seconda dell'unità. Anche in questo caso i software di ogni unità del PFC

sono stati sviluppati da 3 team di sviluppo differenti. In questo modo si è ottenuta diversità sia hardware che software. L'architettura del triplo PFC è rappresentata in Figura 6.3.

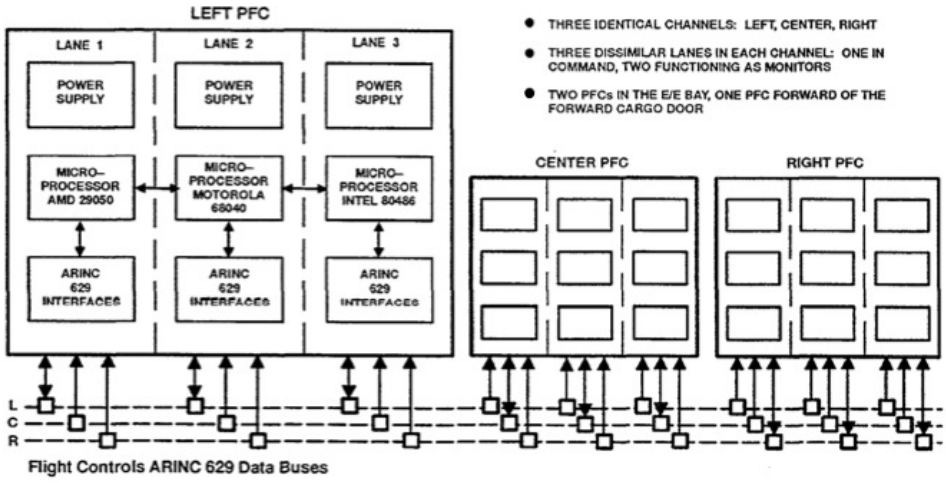

Figura 6.3: Architettura interna dei computer PFC[35]

Il sistema di controllo Fly-By-Wire realizzato dalla Boeing è quindi stato progettato per separare il più possibile i componenti tra di loro, per fornire una maggiore diversificazione rispetto al sistema FBW classico, e per diminuire il rischio di incorrere in errori comuni ai diversi computer.
La tolleranza ai guasti è quindi ottenuta utilizzando la tecnica N-Version Programming a tre varianti e su due livelli. All'interno di ogni PFC i risultati ottenuti dalle tre sotto-unità sono confrontati tramite un meccanismo di decisione che propone, a meno della rilevazione di un errore, il risultato del PFC. I risultati dei tre PFC vengono quindi raccolti da uno dei tre computer che li confronta e sempre tramite un meccanismo di decisioni definisce il risultato finale del calcolo che verrà quindi utilizzato dal sistema per effettuare l'operazione richiesta.

Capitolo 7

Conclusioni

In questo lavoro è stata fornita una rassegna delle più diffuse tecniche di tolleranza ai guasti applicabili a livello software. Oltre a quelle esaminate esistono altre tecniche più complesse sviluppate di recente. Tuttavia quelle trattate possono ritenersi il punto di partenza per molti degli sviluppi futuri. L'importanza di queste tecniche risiede nella possibilità data a progettisti e sviluppatori di produrre software in grado di tollerare sia guasti di progetto, derivanti da imperfezioni commesse nel corso dello sviluppo del sistema, sia guasti operazionali, dovuti al verificarsi di fenomeni casuali durante il normale funzionamento del sistema.
Nell'introdurre la trattazione è stato inizialmente presentato il concetto di affidabilità che abbiamo visto rientrare a far parte del più ampio concetto di Dependability. Tra le strategie utilizzate per ottenere la Dependability si è approfondita la tolleranza ai guasti. L'utilizzo di questa consiste generalmente nel definire l'insieme dei guasti che possono verificarsi e nell'adottare di conseguenza le opportune forme di ridondanza. Si è visto come sia necessario applicare una qualche forma di diversità all'utilizzo della ridondanza, in quanto a livello software la semplice replicazione non è sufficiente a garantire la rilevazione e la gestione dei guasti. Ci si è quindi soffermati sulla trattazione della diversità di disegno, della diversità della rappresentazione dei dati, della diversità temporale e della diversità ambientale. Tra queste la diversità di disegno è sicuramente la forma di diversità più comune. Spesso

le tecniche appartenenti a questa categoria sono ottenute da rielaborazioni o ottimizzazioni di quelle precedenti. Questo fa sì che vi possano essere alcune somiglianze tra le diverse tecniche.
Si conclude quindi la trattazione cercando di individuare, tra le principali tecniche di diversità di disegno, quelli che sono gli aspetti che le differenziano tra loro. Si fornisce infine uno schema riassuntivo di tutte le tecniche affrontate (vedi Tabella 7.1) ed alcune considerazioni finali.

7.1 Comparazione delle tecniche

Vediamo innanzitutto le differenze tra le tecniche RcB e NVP. La prima consiste nell'esecuzione sequenziale di N varianti, su un unico componente, fino a quando il risultato di una variante non soddisfa l'AT. La seconda consiste di N varianti eseguite parallelamente, su più componenti, i cui risultati sono confrontati da un voter che definisce il risultato corretto, ammesso che ciò sia possibile. La differenza sostanziale tra le due tecniche risiede nel meccanismo di decisione, la prima adotta un controllo di validità dei risultati mentre la seconda confronta i risultati ottenuti con un voter. La tecnica NVP può quindi soffrire del problema degli *errori simili* (Capitolo 4), mentre la tecnica RcB no. Allo stesso tempo però la tecnica NVP, eseguendo parallelamente le varianti su più componenti, riesce a controllare eventuali guasti hardware mentre la tecnica RcB no.

La terza tecnica vista è la tecnica DRB che consiste in due varianti replicate su due componenti distinti. Viene eseguita la prima variante sul primo componente e la seconda sull'altro componente. Si validano i risultati con un AT e, se validi, si confrontano con un comparatore. Se corrispondono allora si ottiene il risultato finale, altrimenti si riesegue il meccanismo utilizzando le altre due varianti non ancora utilizzate.
In questo modo la tecnica DRB, oltre al fatto che necessita di sole due varianti, è in grado di coniugare gli aspetti positivi delle tecniche RcB e NVP: i risultati vengono prima validati da un AT e successivamente confrontati

con un comparatore ed inoltre, essendo l'esecuzione distribuita e ripetuta, è possibile rilevare ed individuare eventuali guasti hardware.

La tecnica NSCP è invece un'evoluzione della tecnica NVP. Consiste di N componenti ognuno contenente una variante con relativo AT. I componenti sono raggruppati in coppie e le varianti sono eseguite in parallelo. In ogni coppia i due risultati ottenuti vengono prima validati con l'AT e, se validi, vengono poi confrontati dal comparatore che produce il risultato della coppia. Il meccanismo procede con la comparazione a due a due dei risultati. La caratteristica che differenzia questa tecnica dalle precedenti è la capacità dei componenti di auto-valutare il risultato della propria variante. In questo modo non vi è un unico punto di decisione e questo diminuisce la possibilità che si verifichino *errori comuni* (problema che interessa il voter della tecnica NVP semplice).

La tecnica CRB consiste di N varianti eseguite prima secondo la tecnica NVP e successivamente, qualora il voter non riuscisse ad individuare il risultato corretto, si valutano i risultati delle singole varianti come nella tecnica RcB. Questo permette di sfruttare la velocità di calcolo della tecnica NVP, senza però rinunciare alla più alta probabilità di ottenere un risultato utilizzando la singola validazione della tecnica RcB.

La tecnica AV consiste di N varianti eseguite parallelamente come nel caso della tecnica NVP. A differenza della tecnica NVP i risultati prima vengono validati tramite AT e, se validi, vengono inoltrati al voter che tenta di individuare il risultato corretto. La peculiarità della tecnica AV è quindi l'utilizzo di un meccanismo di decisione di tipo voter dinamico in quando deve essere in grado di valutare il risultato anche senza ricevere tutti gli N risultati. Questa tecnica viene utilizzata quando vi è una considerevole probabilità che il voter, basando il proprio calcolo su tutti gli N risultati, possa produrre un risultato errato.

La tecnica SCOP consiste di N varianti e un'esecuzione della tecnica divisa

in fasi. In ogni fase vengono eseguite parallelamente tante varianti quanti sono i componenti disponibili. Alla fine di ogni fase il voter controlla i risultati appena ottenuti e quelli delle fasi precedenti. La tecnica termina quando il voter è in grado di individuare un risultato corretto. La peculiarità di questa tecnica è quindi quella di riuscire ad adattare l'esecuzione parallela delle varianti alla capacità di calcolo momentanea del sistema.

7.2 Tabella riassuntiva

Categoria	Nome tecnica	Abbr.	Esecuzione	N. varianti : componenti	Mecc. di decisione	Strategia recupero	Sospensione del servizio
Design Diveristy	Recovery Blocks	RcB	Sequenziale	N : 1	AT	Indietro	Si, durante l' esecuzione delle varianti
	N-Version Programming	NVP	Parallela	N : N	Voter	Avanti	No
			Sequenziale	N : 1			Si, durante l' esecuzione delle varianti
	Distributed Recovery Blocks	DRB	Parallela	2 : 2	AT e comparatore	Avanti	No
	N Self-Checking Programming	NSCP	Parallela	N : N	AT e comparatore	Avanti	No
	Consesus Recovery Blocks	CRB	Parallela	N : N	Voter e AT	Avanti	No
	Acceptance Voting	AV	Parallela	N : N	AT e Voter	Avanti	No
	N-Version Programming TB-AT	NVP-TB-AT	Parallela	N : N	Comparatore voter e AT	Avanti	No
	Self-Configuring Optimistic Programming	SCOP	Parallela	N : variabile	Voter	Avanti	No, se N è minore del numero di comp. liberi
Data Diversity	Retry Blocks	RtB	Sequenziale	2 : 1	AT	Indietro	Si, durante l' esecuzione delle varianti
	N-Copy Programming	NCP	Parallela	1 : N	Voter	Avanti	No
			Sequenziale	1 : 1	Voter	Avanti	Si, durante l' esecuzione delle varianti
Temporal Diversity	Riesecuzione con diversità	-	Sequenziale	1 : 1	Confronto	Indietro	Si, durante la ripetizione delle esecuzioni
Environment Diversity	Rejuvenation	-	-	-	Tempo	Indietro	Si, il tempo necessario per riavviare il sistema

Tabella 7.1: Tabella riassuntiva delle tecniche

7.3 Considerazioni finali

Benché alcune delle tecniche viste abbiano ormai più di vent'anni, le tolleranza ai guasti non è ancora sufficientemente utilizzata nello sviluppo dei sistemi software da poter essere reputata un'abituale pratica di progettazione. I motivi di questo scarso utilizzo sono molteplici. Innanzitutto la tolleranza ai guasti viene considerata dai progettisti e dagli sviluppatori come un requisito non funzionale del sistema e questo fa sì che non venga opportunamente implementata nelle fasi di progettazione. Capita quindi di accorgersi dell'inadeguatezza del software solo in un momento successivo alla sua realizzazione, momento nel quale diventa difficile apportare modifiche al fine di migliorarne l'affidabilità.
Inoltre, come si è visto nel Capitolo 5, l'adozione di alcune tecniche di tolleranza ai guasti comporta una spesa aggiuntiva rappresentata sia dai costi di realizzazione delle varianti sia dai costi di acquisizione dell'hardware necessario. Non sempre nella realizzazione di un sistema è possibile affrontare questi costi supplementari. Deve quindi essere trovato un compromesso tra costo e affidabilità del prodotto, che in questo caso andrà a favore del primo ma a scapito della seconda. Questo compromesso definisce il diverso approccio con cui vengono intraprese le produzioni di sistemi software semplici e le produzioni di sistemi software critici.
Nella produzione di sistemi software semplici si tende a tralasciare l'aspetto "tolleranza ai guasti". Si preferisce magari dedicare più tempo all'attività di rimozione dei guasti oppure puntare maggiormente sulla successiva attività di manutenzione correttiva. Nella produzione di sistemi software critici invece, dove l'affidabilità è un vero e proprio requisito, vengono dedicate molte risorse alla realizzazione della tolleranza ai guasti. In sistemi complessi, come possono essere quelli del caso di studio del Capitolo 6, è facile comprendere che i costi supplementari sono più che giustificati, dato che ne va della sicurezza stessa delle persone.

Concludendo si può affermare che, per quanto i costi dell'hardware diminuiscano sempre più e gli strumenti a supporto dell'ottenimento dell'affida-

bilità nei sistemi software stiano aumentando, ottenere un elevato livello di affidabilità grazie all'utilizzo delle tecniche di tolleranza ai guasti rimane una realtà relegata principalmente ai soli sistemi software critici.

Ringraziamenti

Grazie a tutti coloro che mi hanno permesso, direttamente o indirettamente, di arrivare a questo piccolo (per me enorme) traguardo:
Al Prof. Marzolla per la disponibilità che mi ha dimostrato.
A Umbe e Fady, i miei compagni di avventura, per tutto quello che in questi tre anni mi avete dato e abbiamo passato insieme: l'amicizia, la solidarietà, le indimenticabili tirate di studio, gli interminabili progetti, le risate davanti a litri di birra e chili di patate al forno.
Agli amici di vecchia data, ai colleghi di lavoro del Mamamia, ai miei coinquilini, alla cricca sarda che mi aiuterà a conquistare Bologna e a tutti i nuovi amici che questa esperienza universitaria mi ha dato l'opportunità di conoscere.
A Francesca per avermi aiutato, confortato e sopportato ogni singolo giorno di questa faticosissima tesi.
A Barbara, Michele e al regalo più bello che mi avete mai fatto, Emanuele.
Ai miei nonni.
Infine... ai miei genitori, a cui ho dedicato la tesi, per avermi permesso di arrivare fin qui, per avermi sostenuto e compreso in questi anni per niente semplici, per avermi cresciuto senza mai negarmi nulla, per l'affetto che mi avete dimostrato quando ne ho avuto più bisogno e per aver sempre saputo che vi voglio bene anche se non ve l'ho mai detto.

Bibliografia

[1] Mattana G., *Qualità, Affidabilità, Certificazione. Strategie, tecniche e opportunità per il miglioramento dei prodotti, dei servizi, delle organizzazioni*, Milano, FrancoAngeli, 2002, pp. 81-130.

[2] Randell B., Lee P., Treleaven P. C., "Reliability Issues in Computing System Design", *ACM Comput. Surv. 10*, vol. 2, 1978, pp. 123-165.

[3] Avizienis A., Randell B., Laprie J.C. , "Fundamental Concepts of Computer System Dependability", in: *Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, IEEE Computer Society, Seoul, 2001, pp. 1-16.

[4] Avizienis A., Laprie J. C., Randell, B., Landwehr C., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Computer Society*, vol. 1, 2004, pp. 11-33

[5] Laprie J. C., "Dependable Computing: Concepts, Limits, Challenges", in: *Proceedings of the Twenty-Fifth international conference on Fault-tolerant computing*, IEEE Computer Society, 1995, pp. 42-54.

[6] Gray J., "Why Do Computers Stop and What Can Be Done About It?", in: *Symposium on Reliability in Distributed Software and Database Systems*, IEEE Computer Society, 1986, pp. 3-12.

[7] Koren I., Krishna C. M., *Fault-Tolerant Systems*, San Francisco, CA, Morgan Kaufmann Publishers, 2007, pp. 147-173.

[8] Zaipeng X., Hongyu S., Kewal S., "A Survey Of Software Fault Tolerance Techniques", University of Wisconsin-Madison, Department of Electrical and Computer Engineering, Madison, USA, 2003

[9] Storey N. R., *Safety Critical Computer Systems*, Addison-Wesley Longman Publishing Co., Boston, USA, 1996.

[10] Gray J. and Siewiorek D. P., "High-Availability Computer Systems", *IEEE Computer*, vol. 24 (9), 1991, pp. 39-48.

[11] Flammini F., Mazzocca N., "Introduzione ai Concetti e alla Modellistica dei Sistemi Dependable", Università "Federico II" di Napoli, dispense del corso di Sistemi di Elaborazione, 2005.

[12] Ciciani B., Quaglia F., "Sistemi affidabili ed in tempo reale", Università "Sapienza" di Roma, dispense del corso di Modellazione e Valutazione dei Sistemi di Elaborazione, 2006.

[13] Avizienis A., The N-Version Approach to Fault-Tolerant Software, IEEE Computer Society, Vol. SE-11 (12), 1985, pp. 1491-1501.

[14] Brilliant S. S., Knight J. C., Leveson N.G., The Consistent Comparison Problem in N-Version Software, IEEE Transactions on Software Engineering, Vol. 15 (11), 1989, pp. 1481-1485.

[15] Randell B., System Structure for Software Fault Tolerance, IEEE Transactions on Software Engineering, Vol. SE-1 (2), 1975, pp. 220-232.

[16] Avizienis A., Kelly J. P. J., "Fault Tolerance by Design Diversity: Concepts and Experimens", *IEEE Computer*, Vol. 17 (8), 1984, pp. 67–80.

[17] Pullum L. L., *Software Fault Tolerance Techniques and Implementation*, Artech House, Inc., Norwood, MA, 2001.

[18] Hecht H., "Fault Tolerant Software for Real-Time Applications", *ACM Computing Surveys*, Vol. 8 (4), 1976, pp. 391–407.

[19] Elmendorf W. R., "Fault-Tolerant Programming," in: *Proceedings of 2nd IEEE Int. Symp on Fault-Tolerant Computing*, Newton, MA, 1972, pp. 79–83.

[20] Avizienis A., Chen L. "On the Implementation of N-Versin Programming for Software Fault-Tolerance During Execution", in: *Proceedings of COMPSAC 77*, (First IEEE-CS International Computer Software and Application Conference), Chicago , 1977, pp. 149–155.

[21] Kim K. H., Welch H. O., Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults, IEEE Transactions on Computers, Vol. 38 (5), 1984, pp. 526-532.

[22] LAPRIE J. C., Arlat J., Beounes C., Kanoun K., "Definition and Analysis of Hardware and Software Fault Tolerant Architectures', *IEEE Computer*, Vol. 23 (7), 1990, pp. 39–51.

[23] Scott R. K., Gault J. W., McAllister D. F., The Consensus Recovery Block, in: *Proceedings Total Systems Reliability Symposium*, Gaithersburg, MD, 1983, pp. 74-85.

[24] Scott R. K., Gault J. W., McAllister D. F., "Fault-Tolerant SoFtware Reliability Modeling" , *IEEE Transactions on Software Engineering*, Vol. SE-13 (5), 1987, pp. 582-592.

[25] Athavale, A., Performance Evaluation of Hybrid Voting Schemes, M.S. Thesis, North Carolina State University, Department of Computer Science, 1989.

[26] Ammann P. E., Knight J. C., "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Transactions on Computers*, Vol. 37 (4), 1988, pp. 418-425.

[27] Cristian F., Exception Handling in: *Resilient Computing Systems*, T. Anderson, Vol. 2, New York, 1989, pp. 68-97.

[28] Tai A. T., Meyer J. F., and Avizienis A., Performability Enhancement of Fault-Tolerant Software, IEEE Transactions on Reliability, Vol. 42 (2), 1993, pp. 227-237.

[29] Chiaradonna S., Bondavalli A., Strigini L., "On Performability Modeling and Evaluation of Software Fault Tolerance Structures", in: *Proceedings of the First European Dependable Computing Conference on Dependable Computing*, Springer-Verlag, London, UK, 1994, pp. 97-114.

[30] Yurcik W., Doss D., "Achieving fault-tolerant software with rejuvenation and reconfiguration," *IEEE Software*, Vol. 18 (4), 2001, pp. 48-52.

[31] Hagelin, G., Ericsson Safety System for Railway Control, in: Software Diversity in Computerized Control Systems, Springer-Verlag, Vienna, Austria, 1988, pp. 11-21.

[32] Lyu M. R., Software Fault Tolerance, John Wiley & Sons, Bellcore, USA, 1995, pp. 47-75

[33] Briere D., Traverse P., "AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems", Fault-Tolerant Computing FTCS-23 Digest of Papers, 1993, pp. 616-623.

[34] Spitzer C. R., "The Avionics Handbook", Edizione II, New York, USA, CRC Press, 2001, cap. 12.

[35] Yeh, Y.C., "Triple-Triple Redundant 777 Primary Flight Computer", in: Aerospace Applications Conference, IEEE computer, vol.1, 1996, pp.293-307