PROBLEMATICHE DI CERTIFICAZIONE DEL SOFTWARE SAFETY-CRITICAL:*
II - Progetto, sviluppo e valutazione

Fabio A. Schreiber e Angelo Morzenti
Dipartimento di Elettronica e Informazione
Politecnico di Milano
{schreibe, morzenti}@elet.polimi.i

Pubblicato su ***Ingegneria Ferroviaria,*** Anno LIV, n°. 11, novembre 1999
pp. 775-804

* Ricerca parzialmente finanziata dal MURST, progetto "Scienza e tecnica per la qualità della mobilità locale"

4.4 Architettura software

4.4.1 Posizione della norma EN 50128 sull'architettura software

L'architettura software è trattata nel capitolo 9 di EN 50128. Gli obiettivi di questo capitolo del progetto di normativa sono:

- sviluppare un'architettura software che raggiunga i requisiti della specifica dei requisiti software;
- identificare e valutare l'importanza delle interazioni hardware/software per la safety;
- prepararsi al testing di integrazione.

I documenti prodotti in questa fase saranno la specifica dell'architettura software e la parte iniziale del piano di integrazione software che verrà completato nella fase di progetto. *La specifica dell'architettura software identifica la strategia di sviluppo, integrazione, verifica, convalida e manutenzione del software*. Essa dev'essere chiara, precisa, inequivocabile, verificabile, testabile, manutenibile e realizzabile. Inoltre si deve poter mettere facilmente in corrispondenza con la specifica dei requisiti software. Queste caratteristiche - tipiche di un linguaggio di specifica formale - non potranno essere raggiunte pienamente usando un linguaggio naturale. Pertanto anche se la normativa non dice nulla in proposito, *la specifica dell'architettura software dovrebbe essere scritta con lo stesso formalismo usato nella specifica dei requisiti*, creando così una corrispondenza facile e immediata fra le due e raggiungendo gli obiettivi di chiarezza, inequivocabilità e verificabilità.

Secondo EN 50128, il metodo di progetto scelto dovrà avere caratteristiche di modularità, information hiding ed encapsulation. Inoltre la specifica dell'architettura software includerà strategie di progetto software sia per la *fault tolerance* che per la *fault avoidance* e la giustificazione della scelta fatta tra le suddette tecniche. Le tabelle normative corrispondenti ai capitoli 9 e 10, contenute nell'allegato A non suppliscono molto alla genericità dei capitoli cui fanno riferimento in quanto le combinazioni possibili di tecniche, già di per sè molto generiche, sono numerose e, perciò, lasciano agli sviluppatori una libertà d'azione pressoché totale. Ad esempio, per i livelli di integrità software 3 e 4 le combinazioni di tecniche approvate sono:

1-	programmazione difensiva	programmazione con N versioni	codici di rilevamento degli errori
2-	"	"	programmazione con pre- e post- asserzioni
3-	"	"	memorizzazione dei casi eseguiti
4-	"	codici di rilevamento degli errori	programmazione con pre- e post- asserzioni
5-	"	"	memorizzazione dei casi eseguiti

Della *programmazione difensiva* si parlerà nel prossimo capitolo, ma è importante sottolineare che nel progetto di normativa europea EN 50128 si dà a questo termine un significato molto lato, facendogli comprendere sia tecniche di fault detection, sia software error tolerant.

Per il livello di integrità software 2, oltre a una o più delle tecniche sopra riportate, è possibile usare ulteriori metodi che sono: *recovery block, fault detection & diagnosis, tecniche safety bag*.

E' importante notare che questi metodi non sono in realtà completamente distinti e distinguibili gli uni dagli altri, ma hanno ampie zone di sovrapposizione (e quindi pregi e difetti comuni).

Ad esempio l'allegato informativo B spiega che per tecniche di fault detection non s'intende altro che programmazione multiversione, safety bag e programmazione con pre- e post- asserzioni, la quale costituisce, a sua volta, la tecnica di programmazione del test di accettazione dei recovery block [114].

4.4.2 Fault tolerance

La "*tolleranza ai guasti*" (fault tolerance) è, insieme alla "*prevenzione dei guasti*" (fault avoidance) e alla "*rimozione dei guasti*" (fault removal), una delle principali tecniche per aumentare la reliability di un sistema sia hardware sia software [1].

Si dice che un sistema è "tollerante ai guasti" rispetto a una certa classe di difetti/guasti se ha la capacità di svolgere il suo compito in presenza di difetti di quella classe.

La prima introduzione di un qualche meccanismo di tolleranza ai guasti, con la duplicazione o triplicazione di sottosistemi hardware, è avvenuta nei computer della prima generazione [5]: basti pensare che EDVAC, un computer progettato nel 1949, usava delle unità aritmetico/logiche (ALU) duplicate per rilevare gli errori nei calcoli e che la tolleranza ai guasti hardware fu sempre più impiegata negli anni 50 anche a causa della scarsissima reliability dell'hardware - basato su tubi termoionici - che era usato allora [118]. Ancor oggi, nonostante che i componenti hardware abbiano raggiunto una reliability ben maggiore, la tolleranza ai guasti hardware è essenziale, specie negli impianti di segnalazione ferroviaria [40], e in applicazioni critiche per la sicurezza in genere, come mostrano anche i più recenti progetti europei come ad esempio GUARDS (Generic Upgradable Architeture for Real Time Dependable Systems) [15].

Probabilmente si continuerà ad usare la tolleranza ai guasti hardware ancora per molto tempo, poiché la reliability dell'hardware subisce, nel tempo, un progressivo deterioramento e *questa tecnica si è rivelata particolarmente efficace nel contrastare i fallimenti di sistema, dovuti a guasti casuali (e quindi statisticamente indipendenti) dei componenti a causa dell'usura.*

Gli studi sull'applicabilità del concetto di tolleranza ai guasti al software sono iniziati molto più tardi, intorno ai primi anni 70, quando si è pensato e tentato di applicare questa tecnica, oltre che alla eliminazione delle conseguenze di difetti fisici dell'hardware anche alla eliminazione o almeno mitigazione degli effetti (spesso ben più gravi) di errori di progetto sia hardware che software in sistemi computerizzati [5].

La fault tolerance secondo Johnson [77] e Storey [118] si ottiene con la ridondanza, cioè con l'aggiunta al sistema di risorse classificabili come:

- *ridondanza hardware*: uso di componenti hardware replicati come processori, memoria aggiuntiva, alimentatori, ecc.;

- *ridondanza informativa*: uso di codici di rilevamento e correzione degli errori, anche se questa tecnica è trascurata da quasi tutti gli altri autori che si occupano di fault tolerance [5, 6, 3] ecc.;

- *ridondanza temporale*: si consente di utilizzare tempo aggiuntivo cosicché quando si rileva un errore si può ritentare la computazione fallita: in pratica la ridondanza temporale consiste nella ripetizione più volte del programma a partire da un "rollback point" distinguendo due casi:

    1. la(e) nuova(e) esecuzione(i) è(sono) identica(he) alla precedente(i) nel quel caso vengono "tollerati" solo gli errori transitori [5, 77] e in caso di errori permanenti il sistema viene bloccato in uno stato safe;

    2. la(e) nuova(e) esecuzione(i) è(sono) fatta(e) usando una funzione "*encoding on the operands*" nel qual caso possono essere rilevati (e tollerati) anche gli errori permanenti [77], anche se Avizienis e Kelly [5, 6] preferiscono tollerare gli errori permanenti con l'esecuzione successiva di diverse versioni del software e quest'orientamento sembra prevalente, in quanto condiviso anche dagli altri sostenitori della fault tolerance come emerge da [19] e da [1];

- *ridondanza software*: si realizza con due tecniche principali: programmi multiversione e recovery block. che verranno approfonditi in seguito. Diversi autori [3, 31, 118] distinguono tra:

    - *ridondanza statica* nella quale i difetti sono mascherati in modo che i loro effetti non emergano all'esterno, ad esempio con la pratica di accettare come output solo il risultato della maggioranza dei componenti (hardware o software) senza necessità di nessuna azione per realizzare la fault tolerance;

    - *ridondanza dinamica* nella quale vengono rilevati i difetti/errori e vengono attuate sul sistema particolari azioni per annullarne gli effetti (di solito riconfigurando il sistema per rimuovere l'influenza del componente difettoso);

    - *ridondanza ibrida* nella quale si adotta una combinazione delle precedenti tecniche, ad esempio si possono "mascherare" i difetti per impedire che gli errori si propaghino nel sistema e poi rilevarli e riconfigurare il sistema per rimuovere i componenti difettosi.

Una classificazione della fault tolerance in base a dove si trovano i difetti è quella proposta in [66]:

- *tolleranza dei difetti hardware*: tecniche per "tollerare" i guasti hardware, evitando i fallimenti hardware e, così, continuando a far funzionare il sistema almeno in maniera ridotta;

- *tolleranza dei difetti software*: tecniche per “tollerare” i difetti software sia che essi siano transitori, di implementazione o di progetto, evitando i fallimenti software;
- *tolleranza dei difetti/guasti di sistema*: tecniche per “tollerare” i difetti o i guasti nell'ambiente, come ad esempio gli errori degli utenti, e nelle parti del sistema non computerizzate come ad esempio i sensori.

Questa distinzione, che potrebbe sembrare un esercizio puramente accademico, secondo Heimerdinger e Weinstock [66] ha una sua importanza nella realizzazione di sistemi reali, in quanto ad esempio i provvedimenti per garantire la tolleranza dei difetti di sistema si oppongono efficacemente ai difetti di progetto hardware.

I guasti/difetti possono essere *indipendenti* o *dipendenti*. I guasti indipendenti provocano fallimenti casuali in componenti individuali [122] (tipici di questa categoria sono i guasti hardware dovuti all'usura) e sono molto ben “tollerati” con la ridondanza anche senza diversità [5].

I guasti/difetti dipendenti colpiscono simultaneamente o concorrentemente entità multiple e sono uniti da relazioni di dipendenza.

Secondo Villemeur i guasti possono essere tra loro dipendenti per tre diversi motivi [122]:

1. *iniziatori di modo comune*: eventi interni o esterni al sistema che possono innescare un incidente, producendo simultaneamente fallimenti multipli ai componenti;
2. *dipendenze fra sistemi elementari*: eventi o cause di fallimento che creano interdipendenze fra eventi indesiderabili in numerosi sistemi elementari come ad esempio:
    - *dipendenze funzionali*: sono dipendenze fra funzioni di sistemi elementari causate dal progetto stesso dell'impianto;
    - *dipendenze dovute ad equipaggiamenti condivisi*;
    - *interazioni fisiche*: sono eventi che aumentano la probabilità di fallimenti di sistema multipli, anche se non sono iniziatori di causa comune;
    - *dipendenze dovute all'interazione umana*: sono causate dall'azione dell'uomo a tutti gli stadi del ciclo di vita del sistema elementare.
3. *dipendenze fra componenti*: eventi o cause di fallimento che provocano una dipendenza fra eventi indesiderabili che colpiscono numerosi componenti.

Queste classificazioni hanno una loro utilità nell'economia di questo lavoro, in quanto *si vuole analizzare in modo chiaro e compiuto con quali difetti/guasti la fault tolerance (e in particolare la software fault tolerance) è efficace e con quali è poco utile, utilizzando soltanto dati oggettivi.*

Infatti su questo argomento si sente un gran bisogno di chiarezza, perché la letteratura è spesso ambigua o addirittura ingannevole [82]. I sostenitori della software fault tolerance usano spesso espressioni troppo generiche, che, forse per l'ambiguità insita in tutti i termini dei linguaggi naturali, finiscono per far sopravvalutare al lettore la portata della tecnica (si vedano ad esempio [5, 6, 1]) mentre gli oppositori hanno a volte spunti eccessivamente polemici (si veda ad esempio [82]) che infastidiscono il lettore e gli impediscono di apprezzare la solidità degli argomenti. Inoltre i sostenitori della fault tolerance dimenticano di sottolineare che il suo utilizzo comporta aggiungere complessità al sistema e che tale complessità rende il sistema meno affidabile [104].

4.4.3 Fault tolerance con ridondanza senza diversità

Un sistema non fault tolerant è caratterizzato da una sola esecuzione (1 Time 1T) di un programma (1 Software 1S) su un canale hardware (1H), quindi può essere descritto con la notazione: 1T/1H/1S [5].

Nel caso di ridondanza temporale N-pla (con $N \geq 2$) ho un sistema NT/1H/1S che svolge N esecuzioni consecutive dello stesso programma sullo stesso hardware usando N repliche identiche del programma e dei dati ad esso associati: così facendo si riescono a tollerare gli errori transitori (come ad esempio quelli dovuti a momentanei disturbi elettromagnetici o a particelle alfa che colpiscono i chip di memoria a semiconduttore), prendendo come validi gli output solo se sono tutti o almeno la maggioranza uguali; ad esempio nell'Airbus 330/340, dove vi è una ridondanza tripla, l'esecuzione ha successo se gli output di due computer su tre sono uguali, mentre l'output del terzo computer, se non congruente, viene scartato automaticamente [19].

Questo tipo di ridondanza temporale ha il difetto di fallire nel caso di errori permanenti (quali errori di codifica); esiste anche una versione di ridondanza temporale proposta da [77], che, però, non è stata recepita né dal mondo accademico né da quello industriale. Questi, come mostrano [5, 1, 6] e i progetti industriali (quali Airbus 330/340, ELEKTRA ecc.), preferiscono tollerare gli errori permanenti (del software) con la ridondanza e diversità software di cui si parlerà in seguito nei paragrafi sulla software fault tolerance.

Una pratica comune, nel far fronte a guasti di sistema, è quella di caricare da una memoria stabile ed eseguire una nuova copia del programma e dei dati sullo stesso hardware realizzando un sistema 2T/1H/2S [5].

Nei cosiddetti “tandem computer” si realizza, invece, un sistema 2T/2H/2S nel quale si esegue un'altra copia del programma che si trova in un canale hardware in standby [116]. Naturalmente ci dev'essere un programma di riferimento in una memoria stabile, con il quale si realizzano le nuove copie del programma.

Un'architettura fault tolerant, usata in progetti di grande importanza, è la cosiddetta architettura a canale multiplo, nella quale N copie del programma di riferimento vengono eseguite una sola volta in maniera concorrente su N canali hardware (1T/NH/NS) [5], realizzando un sistema con configurazione ridondante N-modulare in grado di tollerare ben (N-1)/2 fallimenti dei moduli senza produrre un fallimento di sistema [118]. Un esempio significativo di quest'architettura mostrato sia in [5] sia in [118] sono i computer di bordo dello Space-Shuttle. Naturalmente si può aggiungere ridondanza temporale all'architettura a canale multiplo; infatti il caso forse più generale di ridondanza (senza diversità) è un'architettura del tipo NT/YH/ZS, nella quale Z copie del programma vengono eseguite N volte con Y canali hardware, con la condizione $N \cdot Y \geq Z$ [5].

L'algoritmo di decisione che sceglierà quale risultato della computazione mandare in output dev'essere molto affidabile e perciò spesso viene implementato N volte, una per ogni computazione in cui si usa il risultato della decisione [5]. Il criterio seguito dall'algoritmo di decisione è solitamente quello di scegliere un risultato appartenente al sottoinsieme maggioritario di risultati uguali, ma a volte può anche essere quello di accettare il primo risultato che supera un test di accettazione.

La ridondanza a computazione multipla senza diversità è molto efficace con i guasti/difetti fisici (visto che è assai improbabile che più componenti hardware abbiano contemporaneamente lo stesso guasto) e con i difetti transitori. Purtroppo quest'architettura non ha nessuna efficacia contro i difetti multipli, che sono presenti in tutte le identiche copie ridondanti dell'hardware e del software, quali ad esempio gli errori di progetto che si traducono in errori permanenti, o i fallimenti di modo comune (per esempio dovuti ad un sistema comune di alimentazione difettoso) [19].

4.4.4 Fault tolerance con ridondanza con diversità

Molti autori [5, 1, 6, 19] propongono di applicare la fault tolerance anche agli errori permanenti e in particolare a quelli di progetto, e tale approccio è stato recepito anche in diversi progetti industriali come il sistema elettronico ELEKTRA sviluppato in Austria [19].

Gli errori permanenti o sistematici rimangono indefinitamente nel sistema finché non vengono corretti e sono, ad esempio, per l'hardware gli errori di specifica e di progetto e per il software gli errori di specifica (che sono i più pericolosi e numerosi [88, 122, 26, 76, 110]), quelli di codifica, l'uso di variabili non inizializzate, gli errori logici nei calcoli ecc..

Le espressioni “errori permanenti” o “errori di progetto”, come tutte le frasi espresse in linguaggio naturale, si prestano ad interpretazioni diverse.

Gli errori permanenti o sistematici software, che sono certamente i più pericolosi, difficili e costosi da eliminare (specie se a si tratta di errori di specifica), spesso sono sottovalutati perché sembrano essere errori casuali, in quanto si manifestano quando sono innescati da una particolare sequenza di input e le sequenze di input sono casuali, o almeno non completamente predicibli [107]. **Altre volte gli errori permanenti sembrano essere errori intermittenti, poiché i loro effetti appaiono solo certe volte** come nel caso di un errore di sincronizzazione software che è permanente poiché la sua codifica è sempre presente, ma l'esecuzione del codice provocherà un errore solo qualche volta, dipendendo da considerazioni temporali [118].

Pertanto gli errori permanenti sono difficili da localizzare e, a volte, scambiandoli per errori casuali (transitori o intermittenti) può accadere che si tenti di eliminarne o almeno mitigarne gli effetti con misure del tutto inadeguate come si è detto parlando della software fault tolerance per gli errori di specifica.

Secondo Avizienis [5] un metodo potenzialmente efficace per evitare i fallimenti causati da errori di progetto in sistemi a computazione multipla è usare N elementi software o/e hardware progettati indipendentemente invece che identiche copie generate da un unico progetto. Si pensa che implementazioni multiple di una specifica adottino progetti diversi e pertanto contengano difetti o errori diversi [82]; infatti l'ipotesi fondamentale della fault tolerance è che sistemi progettati indipendentemente abbiano fallimenti indipendenti [32].

Come già detto, lo sviluppo della software fault tolerance e molto posteriore a quello della hardware fault tolerance ed è stato caratterizzato da grosse polemiche e contrapposizioni insanabili tra autori anche molto noti, con la conseguenza che vi sono industrie dello stesso settore come l'Airbus che usano quest'architettura in maniera estesa ed altre come la Boeing che la rifiutano completamente.

Le principali tecniche di software fault tolerance sono *recovery block* e *N-version programming software* o software multiversione.

4.4.5 Recovery block

Questa tecnica è una tecnica di "riparazione dell'errore all'indietro (deduttiva)" [81, 118]. Il nome è dovuto al fatto che un *recovery block* è un blocco di software che viene eseguito solo se in esso non è stato rilevato un errore, altrimenti viene ripristinato lo stato esente da errori precedente l'esecuzione e viene eseguito un altro recovery block. In pratica prima di eseguire un recovery block si salva lo stato corrente della macchina in modo che esso possa essere ripristinato se necessario [82].

L'architettura di un sistema computerizzato che adotta la tecnica dei recovery block secondo Avizienis [5] è NT/1H/NdS (dove NdS sta per "N diversi software" cioè N programmi sviluppati indipendentemente per soddisfare la stessa specifica o almeno un sottoinsieme di essa).

Ci sono N programmi progettati in maniera diversa e alternativi tra loro che usano uno stesso canale hardware e che vengono eseguiti in sequenza (naturalmente ripartendo sempre dallo stato esente da errori salvato prima dell'esecuzione del primo recovery block) finché un algoritmo di decisione non seleziona il primo insieme di risultati che "passa" il test per il rilevamento degli errori. Come già accennato, non è detto che tutte le implementazioni del programma soddisfino l'intera specifica: alcune soddisferanno solo un sottoinsieme di essa. Le varie implementazioni sono classificate in base alla completezza e alla qualità [82]. La prima implementazione si dice primaria, le altre alternative. Naturalmente è anche possibile usare una sola versione del programma, avendo così N recovery block uguali, ma, così facendo, si rileveranno solo gli errori transitori e non quelli di codifica e di progetto, che invece dovrebbero essere rilevati usando N recovery block diversi. Quest'ultima è un'affermazione plausibile anche secondo il parere di [32] (sia pure in relazione al software multiversione), ma non suffragata da prove adeguate e spesso sconfessata dall'osservazione empirica di programmi reali come vari software per le ricerche geologiche che avevano i medesimi fallimenti a causa dell'uso degli stessi COTS (Component Off The Shelf, cioè software preconfezionato).

Un altro aspetto critico, oltre alla mancanza di prove per l'ipotesi di fallimenti indipendenti per programmi progettati indipendentemente, che sta alla base di tutta la software fault tolerance con diversità, è il *test di accettazione* che di solito, ma non sempre, è lo stesso per l'implementazione primaria e per tutte le alternative. E' anche possibile avere test di accettazione separati per ogni alternativa e ciò è auspicabile quando le alternative successive forniscono progressivamente sempre meno funzionalità perché ad esempio sono più semplici per ridurre la complessità o per avere maggiore performance [82]. Il test di accettazione dovrebbe essere al tempo stesso semplice e completo, ma ciò è molto difficile al di fuori di casi rari, come mostrano le esperienze compiute su programmi Pascal degli studenti di Leveson [114]. Test di accettazione completi sono spesso complessi e perciò hanno elevata probabilità di contenere a loro volta errori di progetto che portano o a far passare un output errato, riducendo così la reliability del sistema, o a non far passare un output corretto, riducendo la disponibilità del sistema. *Un sistema software con recovery blocks con test di accettazione scorretti può avere chiaramente una reliability addirittura inferiore a quella dello stesso sistema non fault tolerant.*

I recovery block sono stati oggetto di minori studi (e minori confutazioni) rispetto al software multiversione, anche se uno studio su un progetto di larga scala durato tre anni svolto da Anderson e Barrett ha mostrato che *sono in grado di aumentare la reliability di un ordine di grandezza* [82, 1]. Probabilmente anche la relativa scarsità di studi su questa tecnica ha portato i compilatori di EN 50128 a raccomandarne l'uso solo per livelli di integrità software bassi (1 e 2).

Un sistema con recovery block software è naturalmente più complesso rispetto ad uno non fault tolerant, anche perché per avere una performance accettabile in termini di velocità di esecuzione bisogna rendere più complesso anche l'hardware aggiungendo componenti speciali a ciò finalizzati, detti *recovery cache* [1], e, come ammettono anche Barrett e altri, in [19], l'affidabilità è inversamente proporzionale alla complessità.

4.4.6 software multiversione (o N-version programming)

Questa tecnica è una tecnica di “riparazione dell'errore all’avanti (induttiva)” [81, 118]. L'architettura di un sistema con N versioni (diverse) del programma, di norma, è 1T/NH/NdS [5] (con NdS = “N diversi software”). N (con $N \geq 2$) programmi progettati e implementati indipendentemente da una stessa specifica vengono eseguiti contemporaneamente su N canali hardware ed i risultati sono inviati ad un algoritmo di decisione che determina un singolo risultato da mandare in output, solitamente il valore prodotto dalla maggioranza delle versioni del programma. Naturalmente gli errori sono “tollerati” solo se la maggioranza dei programmi produce valori uguali (cioè compresi in un certo intervallo).

In molte applicazioni un controllore esterno ai programmi (di solito un altro programma) ripara o disconnette le versioni che hanno fallito prima dell'esecuzione successiva [82]. Si può utilizzare anche un algoritmo di decisione che verifica i valori intermedi prodotti dalle varie versioni del programma nei cosiddetti punti di verifica incrociata [5, 82], trovando così i programmi con errori prima della fine dell'esecuzione.

Per la “buona riuscita” di questo metodo è molto importante che le diverse versioni della specifica iniziale siano progettate in maniera indipendente, cioè da team indipendenti, separati fisicamente e magari anche provenienti da culture diverse, con strumenti e linguaggi di programmazione diversi.

Un approccio promettente, usato nel sistema ferroviario ELEKTRA, sviluppato da Alcatel Austria e del centro di ricerca austriaco Seiberdorf, è stato usare un linguaggio di programmazione procedurale per una versione ed uno non procedurale per un'altra [19]. Dopo che i team hanno sviluppato in maniera del tutto indipendente le versioni del programma e le hanno testate individualmente, spesso le versioni sono soggette a testing usando verifiche di comparazione nelle quali esse sono considerate corrette se i loro output sono concordi [82]. Questa pratica è molto rapida e relativamente poco costosa, poiché può essere automatizzata, pertanto alcuni (sostenitori del software multiversione) hanno suggerito di usare solo questo tipo di testing comparativo per testare il software multiversione, ma Leveson [88] ha giustamente criticato questa proposta poiché, così facendo, è sicuro che gli errori che non verranno "tollerati" non saranno neppure rilevati dal testing. Naturalmente gli errori/difetti, trovati col testing comparativo, sono corretti dai vari team in maniera indipendente.

[5] auspica che *la multiversione venga applicata anche alla specifica*, scrivendola in due linguaggi formali in maniera indipendente e poi effettuando un confronto. Ciò, probabilmente, potrebbe limitare la scorretta traduzione dei requisiti imposti dal cliente nella specifica formale, ma a questo scopo, sembra che, alla luce dei più recenti sviluppi che si sono avvalsi di specifiche formali [51], sia più proficuo far imparare ai tecnici dell'azienda cliente e a tutti coloro che partecipano in qualche modo al progetto il metodo formale in uso. Certamente *combinare due tecniche formali una operazionale e l'altra descrittiva è utilissimo per verificare in due modi differenti le proprietà di un sistema*; attualmente sono in corso studi al politecnico di Milano per integrare le reti di Petri temporizzate e TRIO a livello teorico oltre che operativo [101], ma con una filosofia molto diversa rispetto a quella che sta alla base della programmazione multiversione.

L'uso combinato di diversi linguaggi di specifica formali e non formali alla luce di quanto emerge da [6] è da evitarsi, in quanto, su 100 test eseguiti, in oltre 20 o non c'era un output maggioritario o l'output maggioritario non era quello corretto.

4.4.7 Valutazione della software fault tolerance

A guardare la mole di articoli scritti a favore e contro la software fault tolerance (soprattutto per quanto riguarda la modalità multiversione), si potrebbe pensare che questo argomento sia stato studiato in maniera completa ed esaustiva, ma in realtà non è così; infatti gli studi sperimentali che hanno esplorato questa tecnica sono molto pochi [114] e vi è un'assenza totale di studi che mettano a confronto la software fault tolerance con altre tecniche per aumentare la reliability in sistemi safety critical reali e di una certa consistenza.

Di norma la scelta di usare o meno la software fault tolerance è dettata dalle convinzioni personali dei progettisti e viene effettuato soltanto un mero confronto fra la reliability del sistema fault tolerant realizzato e quella di un ipotetico sistema analogo nel quale non si sia usata nessuna tecnica per aumentare la reliability. Nella valutazione si dovrebbero utilizzare unicamente dati provenienti da studi svolti su programmi sviluppati da team altamente professionali per sistemi safety critical e non, ma comunque di una certa complessità.

Diversi autori [82, 83, 114, 88] sostengono apertamente, basandosi sui risultati dei propri esperimenti, che l'ipotesi di indipendenza statistica dei fallimenti di programmi sviluppati indipendentemente non è valida.

Knight e Leveson, in [83], sottopongono un programma scritto in 27 versioni a un milione di input verificando che gli errori coincidenti sono in numero molto maggiore di quello predetto dal modello, che assumeva l'indipendenza statistica degli errori. Inoltre, come ribadito anche in [114], nel 35% dei casi con software sviluppato in tre versioni e sottoposto a un milione di casi di test gli errori non venivano rilevati col meccanismo di "votazione di maggioranza" tipico della programmazione multiversione. Ciò non è poi così sorprendente, in quanto le persone tendono a commettere errori nelle parti più difficili del problema e quando trattano casi non standard o casi limite nello spazio di input, non commettono errori a casaccio [88].

Butler e Finelli, in [32], dimostrano che l'ipotesi che versioni di un programma sviluppate separatamente falliscano indipendentemente non è dimostrabile (né confutabile) attraverso sperimentazione nella zona dell'ultrareliability (cioè per livello di integrità software 4), ed afferma che le argomentazioni, peraltro molto soggettive, in suo favore non possono certamente giustificare che essa venga presa come un assioma.

Avizienis e Kelly [6] affermano che, con la software fault tolerance, è possibile ridurre i costi di sviluppo eliminando gran parte dell'attività di convalida e verifica; ciò tuttavia è confutato dai risultati sperimentali di Shimeall e Leveson [114] che, eseguendo otto versioni di un programma con diecimila insiemi di dati generati casualmente, hanno mostrato che gli errori rilevati ed eliminati con tecniche di fault elimination non erano gli stessi "tollerati", facendo così arguire agli autori che testing e fault tolerance dovrebbero essere considerati complementari e non in competizione sia pure con qualche perplessità, visto che *su 103 errori che hanno portato a fallimenti coincidenti delle varie versioni soltanto 24 venivano rilevati dal testing*.

Anche i risultati di questo esperimento devono essere considerati molto parziali; essi tuttavia mostrano chiaramente che *la combinazione di fault tolerance e testing lascia spazio a un gran numero di errori di sfociare in fallimenti*, quindi ci vuole qualcosa in più e, forse, dal momento che la fault tolerance in questo esperimento ha tollerato solo il 35% dei difetti software, il notevole costo aggiuntivo necessario per usare questa tecnica non è neppure giustificato.

Più redditizio sarebbe investire di più nella specifica e nella verifica formali che nella fault tolerance; infatti Shimeall e Leveson, in [114], facendo riferimento ad un esperimento svolto da Brunelle ed Eckhardt, osservano che, se si usano tre versioni di un programma di cui solo una verificata formalmente, si nota che *le due versioni non verificate formalmente spesso portano con il loro voto a risposte sbagliate mascherando l'output giusto dell'altra*.

Brilliant e altri [17] notano che algoritmi anche matematicamente corretti a volte producono risultati diversi dovuti a instabilità numerica e ciò porta una difficoltà notevole nello stabilire i criteri in base ai quali gli output delle varie versioni possono considerarsi concordi o meno.

Anderson e altri [1] valutano *l'incremento di spesa ottenuto adottando la software fault* *tolerance (in questo caso i recovery blocks) intorno al 60%,* ma probabilmente questo valore va incrementato alla luce del fatto che il testing non può essere ridotto ed il suo costo è direttamente proporzionale alla complessità del sistema.

Secondo Leveson [88], *con N versioni di software i costi vengono moltiplicati per un fattore maggiore di N* perchè non solo il costo di sviluppo sarà N volte, ma anche le spese di manutenzione aumenteranno N volte ed aumenterà anche la spesa per la specifica. Inoltre la complessità aggiunta grazie alla coesistenza di più versioni dello stesso programma e dei programmi per gestirla - come test di accettazione, algoritmo di decisione ecc - renderà il sistema più soggetto agli errori. Ad esempio i fallimenti del sistema di controllo digitale durante i primi testing di volo dello Space Shuttle erano dovuti ad errori nel sistema gestione della ridondanza [88]. Inoltre, per avere *prestazioni accettabili in termini di velocità di esecuzione* in progetti di un certo spessore, va ribadito che *occorre aumentare anche la complessità dell'hardware* con l'aggiunta di nuovi componenti per velocizzare le operazioni del software fault tolerant [1];in tal modo aumentano ancor più i costi di sviluppo, di verifica e di manutenzione nonchè i problemi cui si deve far fronte, visto che occorre studiare la reazione del software e del sistema ad eventuali guasti in sistemi hardware più complessi e quindi più difficili da verificare.

Eckhardt e Lee, in [44], hanno dimostrato matematicamente che ci sono dei limiti ai possibili incrementi di reliability del software ottenibili con il software multiversione.

Nei *sistemi real-time* la fault tolerance è ancora più problematica, in quanto la *ridondanza temporale (v. 4.4.3) non può essere indefinita* e ciò, specie nei casi dove non si trova una maggioranza tra i risultati dei programmi, può essere molto problematico. La reale diversità di progetto nel software in tempo reale è minima, in quanto i progetti dei "team indipendenti", per rendere possibile alle versioni di votare in un ambiente real time (o di poter confrontare i risultati intermedi), sono sovraspecificati e con moltissimi vincoli per cui si arriva al paradosso che la safety dipende da una qualità (la diversità) che in realtà non esiste o è minima [88].

L'ultimo problema (ma il primo per importanza) è che versioni multiple di un programma, scritte dalla stessa specifica dei requisiti, sono efficaci solo con gli errori di codifica (e talvolta solo con un limitato sottoinsieme di questi) mentre esperienze tratte dagli incidenti ci mostrano che i maggiori problemi per la safety - sia in termini di numero sia di gravità delle conseguenze - nascono da errori nella specifica dei requisiti software, soprattutto da fraintendimenti dell'operazione richiesta al software e qualsiasi tipo di ridondanza non fa altro che duplicare questi fraintendimenti [88].

Gli errori di specifica sono problemi intrattabili con la fault tolerancee pertanto devono essere trattati con tecniche di fault avoidance come l'uso di linguaggi di specifica formali [118]. Anche Avizienis [5] e le varie esperienze di Knight e Leveson mostrano come nella programmazione multiversione le specifiche debbano essere ancor più complete e chiare di quanto di solito sia necessario, pertanto il ricorso ad un linguaggio di specifica formale sarà comunque necessario.

Riassumendo, i motivi che, nell'attuale stato dell'arte, spingono ad essere scettici sul software fault tolerant sono:

- eccessivo costo;

- scarsa efficacia (mostrata sperimentalmente);
- aumento di complessità con interazione tra molti programmi e, come nota [107], anche l'interazione tra programmi non safety critical e programmi safety critical può avere effetti imprevedibili per cui conviene semplificare al massimo le funzioni safety critical ed addirittura allocarle in computer separati da quelli che svolgono funzioni non safety critical;
- diminuzione della velocità di esecuzione che può essere fatale in un sistema real time;
- sostanziale inesistenza di diversità e perciò inutilità nei sistemi real time;
- inattendibilità dell'ipotesi di indipendenza statistica dei fallimenti di programmi sviluppati indipendentemente da una medesima specifica [45, 17, 18, 83, , 85, 82, 114, 88] che porta all'impossibilità di fare stime della reliability nella zona dell'ultrareliability [32];
- impossibilità di "tollerare" gli errori di specifica e quindi possibilità di evitare al massimo un terzo dei fallimenti software (e solo quelli meno pericolosi).

Le (poche) esperienze di successo nell'implementazione di software safety critical con la programmazione multiversione rilevate in campo ferroviario sono due: il sistema ELEKTRA in Austria [19] e gli "interlocking and train control systems" delle ferrovie svedesi. Il successo di questi progetti, però non è certamente dovuto interamente all'applicazione della programmazione multiversione anzi, per quanto riguarda il progetto svedese, nel quale al software vengono affidate molte funzioni safety critical vitali, è doveroso ricordare che **fu specificato formalmente** e che ABB Signal (il costruttore) attribuisce gran parte del merito del suo successo (ed i notevoli risparmi in tempo e denaro ottenuti) all'uso di Prover [118].

Certamente l'uso della software fault tolerance può portare ad un aumento della reliability rispetto ad un analogo progetto che non adotta nessuna tecnica per valorizzare questa qualità, ma le esperienze in campo industriale e accademico dimostrano che si possono avere incrementi più significativi, e soprattutto con maggiori ricadute positive per la safety ed a minor costo con altre tecniche.

5 LA NORMA EUROPEA EN 50128 E IL PROGETTO E LA GESTIONE OPERATIVA

5.1 Progetto e sviluppo software

5.1.1 Posizione della proposta di normativa EN 50128 sul progetto e lo sviluppo software

La norma europea EN 50128, nel capitolo 10 riguardante il progetto e lo sviluppo del software, si pone come obiettivo di creare software di un definito livello di integrità, a partire dalla specifica dei requisiti e dalla specifica dell'architettura. Essa inoltre richiede esplicitamente che il software prodotto sia facilmente analizzabile, verificabile e testabile, nonché di selezionare un insieme di strumenti, compresi linguaggi di programmazione e compilatori, che assista la verifica, la convalida, la valutazione e la manutenzione.

In questa fase i documenti ricevuti in input saranno:

- specifica dei requisiti software;
- specifica dell'architettura software;
- piano di integrazione software;
- piano di assicurazione della qualità software.

I documenti in output saranno:

- specifica del progetto software;
- specifica dei test sul progetto software;
- specifica del progetto dei moduli software;
- specifica dei test sui moduli software;
- piano di integrazione software;
- codice sorgente e documentazione di supporto;
- rapporto sui test sui moduli software;
- rapporto di integrazione software.

Se la specifica dei requisiti software contiene sia funzioni safety-critical sia funzioni non safety-critical la specifica del progetto software indicherà come dev'essere ottenuta l'indipendenza fra queste funzioni.

Distinguere parti safety-critical e non di un medesimo software è una tesi molto cara a Leveson (reperibile in quasi tutti i suoi articoli citati in bibliografia) e non priva di suggestione, ma in pratica di difficile realizzazione, in quanto dall'esperienza del gruppo di Parnas risulta che i fallimenti anche in parti non importanti del codice possono avere ripercussioni inaspettate ovunque [107]. Secondo la visione di Parnas tutte le parti di un prodotto software safety-critical devono essere considerate safety-critical, pertanto le funzioni safety-critical e quelle non safety-critical devono essere separate fisicamente allocandole in software e computer diversi. La posizione di Parnas è condivisibile in quanto *è molto difficile ottenere una "separazione" funzionale di parti di uno stesso codice, cui applicare un trattamento di progetto e di verifica diverso ed ancora più difficile è dimostrare l'indipendenza tra funzioni codificate dallo stesso programma.* Invece è molto più facile costruire programmi che implementano le funzioni safety-critical, separati fisicamente dai programmi che svolgono le altre funzioni. Così facendo si otterranno dei programmi più piccoli e quindi meno complessi e più semplici da verificare.

La clausola 10.4.2 impone che la misura e la complessità del software prodotto devono essere ridotte al minimo e ciò appare perfettamente in linea (o almeno più facilmente realizzabile) con la tecnica suggerita da Parnas.

Il capitolo 10 poi suggerisce di applicare una tecnica di programmazione modulare, fornisce un lungo elenco di documenti cartacei che dovranno essere allegati e dei contenuti "minimi" delle linee di commento del codice sorgente, il quale, accogliendo le imposizioni della norma, almeno nella parte "esplicativa", non potrà certamente essere breve.

Comunque questo capitolo, come i precedenti, fornisce un numero di alternative così ampio, da finire per non imporre (a parte la mole di documentazione cartacea) praticamente nulla e lasciare piena libertà agli sviluppatori. Ad esempio imporre che i test sul software possano essere svolti su un computer, su un processore che emula la macchina obiettivo, o sulla macchina stessa, equivale a non imporre nulla. Allo stesso modo, richiedere che il linguaggio scelto abbia caratteristiche che si armonizzano col metodo di progetto, lascia una libertà di scelta pressoché totale.

La tabella normativa allegata al capitolo 10 non viene poi molto in aiuto ai gestori dei progetti, perché anche essa lascia la medesima varietà di scelta che era contenuta nella tabella allegata al capitolo sulla specifica dei requisiti.

La tabella riguardante il progetto e lo sviluppo software impone che, per il livello di integrità software tre o quattro, l'insieme delle tecniche scelte debba includere una delle seguenti:

- metodi formali;
- metodi semiformali;
- metodologia strutturata, per esempio JSD, MASCOT, SADT, SDL, SSADM e Yourdon.

Una delle suddette tecniche (scelta liberamente dagli sviluppatori) dovrà essere abbinata o con un traduttore convalidato o con un traduttore già provato in uso. Inoltre, per tutti i livelli integrità software, sarà obbligatorio adottare un approccio modulare e, per i livelli 3 e 4, registrare e analizzare i dati e servirsi di standard di codifica e di progetto, quali:

- evitare la presenza di oggetti dinamici;
- evitare la presenza di variabili dinamiche;
- limitare l'uso dei puntatori;
- limitare l'uso della ricorsione;
- evitare salti incondizionati.

Gli sviluppatori potranno poi scegliere (a loro discrezione) un insieme appropriato di ulteriori tecniche fra le numerose non obbligatorie contenute nella tabella "normativa" allegata al capitolo 10. Tali tecniche non obbligatorie includono, ad esempio: uso di linguaggi di programmazione fortemente tipizzati, programmazione ad oggetti, programmi analizzabili (cioè con moduli piccoli e che si prestano bene all'analisi statica), uso di sottoinsiemi dei linguaggi di programmazione, varie tecniche di testing ecc..

Per quanto riguarda i linguaggi di programmazione, per i livelli di integrità software 1 e 2, sono altamente raccomandati ADA, modula-2 e Pascal, mentre per i livelli di integrità software 3 e 4, sono raccomandati, nella stessa misura, ADA, modula-2, Pascal e Fortran 77. Sembra tuttavia molto difficile, se non impossibile, coniugare l'obbligo di utilizzo di programmazione modulare con l'uso del Pascal, che notoriamente (almeno nella sua forma standard ISO) non supporta questa tecnica di programmazione, e con l'uso del Fortran 77, che ha nella mancanza di modularità una delle principali cause del suo abbandono.

Per i livelli di integrità software 3 e 4, il progetto di normativa europea sembrerebbe aver accolto il pressante invito di molti eminenti studiosi, quali Cullyer, Neumann e

Storey, di non servirsi assolutamente del linguaggio C per il software safety-critical, in quanto il C è “non raccomandato”, ma è così solo in parte perché, in realtà, nello standard si accetta e si “raccomanda” l'uso di un non meglio identificato “sottoinsieme di C” con standard di codifica. In pratica è tranqillamente possibile realizzare del software safety-critical con il linguaggio C a patto di “limitare” (ma fino a che punto?) l'uso di puntatori e ricorsione ed evitare l'uso di variabili e di strutture dinamiche.

5.1.2 Principali tecniche di progetto

Vi sono sostanzialmente due metodologie di progettazione del software safety critical: il progetto “*tradizionale*” e il progetto *per raffinamenti successivi* (a volte indicato anche come progetto per raffinamento formale). Con l'espressione progetto tradizionale si intendono tutte le tecniche di progetto che non fanno uso dei metodi formali. Certamente l'aggettivo “tradizionale” può essere giudicato poco appropriato a tecniche, nate anche recentemente, ma esso sembra preferibile all'aggettivo “informale”, in quanto il progetto di un software safety-critical, anche se non svolto con i raffinamenti successivi della specifica formale, dovrà comunque raggiungere livelli di qualità notevoli ed essere sottoposto a verifica formale per accertarsi che il progetto sia conforme alle specifiche, se non si vuole incorrere nel rischio di usare software inaffidabile per funzioni di sicurezza. Questa suddivisione rispecchia le due visioni, lontane sia temporalmente sia concettualmente, di Floyd (cioè dare significati ai programmi) ed Abrial (cioè dare programmi ai significati), anche se poi in realtà questi due grandi ricercatori hanno creato degli slogan molto suggestivi ed accattivanti per tecniche che erano già presenti prima della pubblicazione dei loro libri. Ad esempio, la tecnica di progettazione e sviluppo per raffinamenti successivi della specifica formale (in modo semiautomatico) è già ben presente in TRIO [54] prima della nascita del metodo B e parecchi anni prima della pubblicazione di “The B-method” (cfr. anche [80]).

L'industria comunemente accetta ed applica, come efficienti e sicure, strategie come progettazione modulare, information hiding e programmazione strutturata. La *progettazione modulare* consiste nel dividere il progetto in componenti facilmente gestibili e comprensibili, distinte, ma connesse tramite ben definite interfacce. I benefici dell'approccio modulare sono più che evidenti; in pratica si suddivide il programma in moduli di dimensioni inferiori, che sono più semplici da progettare, testare e verificare. Inoltre è più facile dividere il lavoro fra più team di sviluppo (o fra diversi programmatori in caso di software di dimensioni modeste) ed è più semplice modificare parti del software, intervenendo solo sui singoli moduli. La facile modificabilità aumenta le possibilità di riuso del software, una prospettiva molto allettante (e molto ricercata) soprattutto nel software gestionale per ovvi motivi economici.

Il *riuso del software* safety-critical presenta, invece, non pochi problemi, poiché esso, di solito, è tagliato su misura per il sistema che deve controllare, quindi il suo riutilizzo, in applicazioni critiche per la sicurezza anche molto simili, *può avere conseguenze disastrose, se non vengono rilevate differenze anche minime fra i sistemi da controllare o proteggere*. Un esempio molto noto è quello del fallimento della missione Ariane-5, a causa del riutilizzo del software già usato con successo per Ariane

4, senza tenere in debita considerazione la differenza tra i due vettori. La prima parte della traiettoria di Ariane 5 differiva infatti da quella di Ariane 4, tanto da produrre una velocità orizzontale molto più alta, cosicché proprio la velocità orizzontale troppo elevata ha provocato un errore sugli operandi, durante una conversione di dati, errore che non era protetto [4].

Pertanto il riutilizzo di software safety-critical, anche se molto caldeggiato dall'industria (per ovvi motivi) e da diversi progetti di Standard (come ad esempio IEC 61508), va messo in atto solo dopo un'attenta analisi e raffronto dei requisiti dei sistemi in considerazione ovviamente specificati con linguaggi formali, perché il raffronto sia, veramente affidabile ed indipendente da fattori legati alla capacità di giudizio di “esperti”, la quale, in molti casi, ha provocato veri e propri disastri.

L'information hiding permette di avere variabili la cui manipolazione è fortemente disciplinata. Ciò fa sì che tali variabili non abbiano influenza sugli altri moduli cosicché si riduce la complessità, l'accoppiamento fra moduli e quindi la probabilità di errore. L'information hiding insieme alla programmazione modulare, poi, dovrebbe evitare (o almeno ridurre) la propagazione di condizioni anormali delle variabili e di errori. Con l'applicazione dell'information hiding gli sviluppatori ed i revisori di un modulo non conoscono (o comunque conoscono solo in minima parte) la struttura e le caratteristiche interne degli altri moduli. Un esempio di successo di applicazione di questa tecnica è il sistema operativo Unix che deriva gran parte della sua flessibilità dal fatto di nascondere la differenza fra file e dispositivi. Parnas e altri [107] esprimono delle riserve su un uso esagerato di questa tecnica che, ad esempio, può portare a delle difficoltà nelle modifiche e contrasta col principio ingegneristico generale, che considera l'esame completo del lavoro degli altri come una parte importante del controllo di qualità di un progetto. Bisognerebbe che tutti gli oggetti che hanno buona probabilità di essere modificati insieme fossero “nascosti” nello stesso modulo, quindi per applicare con profitto questa tecnica è necessario fare delle ipotesi sulla probabilità dei vari tipi di cambiamento.

L'information hiding è certamente, insieme alla programmazione modulare, uno degli strumenti principali per ottenere l'isolamento delle parti safety-critical del software tanto auspicato in [87], ma criticato da Parnas e altri [107] che sostengono che questo risultato si può ottenere soltanto separando fisicamente il software safety-critical da tutti gli altri programmi. Questa diversità di vedute tra due scuole di pensiero ha radici concettuali profonde. Parnas ed i suoi seguaci ritengono che *safety e reliability non devono essere considerate come qualità separate del software*, mentre Leveson e i suoi allievi ritengono che *safety e reliability siano qualità separate e spesso in alternativa*.

La *programmazione strutturata* è quella che include nella sua sintassi costrutti come "while…do", "if…then...else", "repeat...until" e cicli finiti. Oggi questo tipo di programmazione e i numerosi linguaggi che la supportano si sono affermati nel mondo industriale perché notevolmente più efficienti non solo in termini di impegno del team di sviluppo, ma, con i moderni compilatori, anche in termini di tempo di esecuzione. Inoltre i programmi strutturati hanno maggiore portabilità, in quanto non hanno problemi di adattabilità alla macchina, e sono più sicuri, in quanto più comprensibili, quindi più facilmente revisionabili.

Secondo Bologna e altri le proprietà di un progetto importanti per la safety sono [10, 105]:

- uso di *programmazione modulare*;
- *interfaccia esterna*: assicurare che specificati eventi esterni siano fronteggiati in modo che le funzioni di safety continuino ad operare senza interruzione;
- *interfacce alle componenti di safety*: separare le componenti software con implicazioni per la safety da quelle senza implicazioni per la safety in modo da assicurare che le operazioni delle seconde non interferiscono con quelle delle prime. Su questo punto vediamo che il settore nucleare mantiene una posizione ambigua rispetto sia a Parnas che a Leveson, in quanto questa frase può essere liberamente interpretata a favore dell'uno o dell'altro approccio;
- *integrità di interfaccia*: assicurare la correttezza delle interfacce fra componenti software cosicché non si verifichino errori, come ad esempio dati non validi;
- *integrità dei dati*: assicurare che il software operi in stati definiti, minimizzando l'occorrenza di errori che potrebbero portare il software ad una transizione ad uno stato indefinito o permettere che sia svolta una funzione non voluta;
- *controllo di flusso*: assicurare il corretto continuo operare delle componenti con implicazioni per la safety;
- *gestione degli errori*: assicurare che il software sia robusto e capace di riprendersi da un errore seguendo una strategia ben definita.

Oggi si sono affermati, nel mondo industriale, i metodi di progettazione ed i linguaggi di programmazione orientati agli oggetti, mentre i vecchi metodi di decomposizione algoritmica, che ponevano la loro attenzione soprattutto sull'ordine degli eventi sono stati in gran parte abbandonati.

Un progetto orientato agli oggetti mette in evidenza gli oggetti su cui si opera o che causano le azioni volute. Da un progetto orientato agli oggetti è molto più facile produrre un programma orientato agli oggetti, in quanto ci sarà una più diretta corrispondenza fra gli oggetti del progetto e gli oggetti dell'implementazione. Comunque, un programma orientato agli oggetti non è necessariamente prodotto da un progetto orientato agli oggetti, né un progetto orientato agli oggetti genera necessariamente un programma orientato agli oggetti.

Va notato che la distinzione fra progetto tradizionale e progetto per raffinamenti successivi non è, naturalmente, l'unica possibile: c'è anche chi, come Yau e Tsai [126] considera innovativo il progetto orientato agli oggetti e tradizionali tutti gli altri tipi di progetto. Si preferisce non usare questo tipo di distinzione, in quanto un progetto orientato agli oggetti può essere svolto sia in modo formale sia in modo informale. In particolare alcuni linguaggi di specifica formale, come ad esempio TRIO+ (la versione di TRIO orientata agli oggetti), OBJ e B, si prestano particolarmente bene alla specifica e al progetto di programmi orientati agli oggetti, sia pure con opportune distinzioni. TRIO+ è molto adatto per la specifica ed il progetto di software real-time orientato agli oggetti, invece OBJ ha dato buona prova di sè nella specifica di programmi orientati agli oggetti non real-time (in quanto non supporta il tempo reale). Per quanto riguarda B, il metodo si applica adeguatamente alla specifica e progettazione di software orientato agli oggetti, ma non è ancora ben chiaro se e quando potrà trattare anche applicazioni concorrenti con requisiti real-time stringenti dato che, a tutt'oggi, i suoi strumenti non sono ancora adeguati al tempo reale.

Comunque, indipendentemente dalla scelta di progetto formale o informale, secondo Yau e Tsai [126] la progettazione orientata agli oggetti ha dei vantaggi sugli altri metodi di progetto in termini di:

- produttività, grazie all'astrazione che permette la costruzione di componenti riutilizzabili;
- manutenibilità, grazie alla pratica di nascondere l'informazione dentro i vari componenti software (senza che essa sia disponibile agli altri componenti);
- sicurezza dei programmi e integrità dei dati, grazie ai domini di protezione che definiscono i diritti di accesso e le operazioni possibili per uno specifico utente;
- maggiore capacità di adattarsi ai sistemi in tempo reale.

Due notazioni, tra la varie utilizzabili in fase di progetto, particolarmente efficaci e con un elevato grado di complementarità sono: TDN (Textual Design Notation) e GDN (Graphical Design Notation). TDN prende spunto dal linguaggio Ada, con opportune integrazioni, ad esempio per quanto riguarda la progettazione ad oggetti. Questa notazione è un ausilio molto valido per la modularizzazione in quanto consente di descrivere i vari moduli e le loro interrelazioni, cioè le risorse altrui utilizzate da un modulo e quelle da esso rese disponibili ad altri [53].

GDN è una notazione grafica, che si può affiancare a quella testuale, basata sulla rappresentazione a grafo aciclico per descrivere le interrelazioni tra i moduli. Si disegna un grafo nel quale ogni nodo rappresenta un modulo, che naturalmente ne può contenere altri, e gli archi orientati entranti o uscenti da esso rappresentano, in modo molto immediato, le sue relazioni con gli altri moduli: un arco uscente dal modulo A ed entrante nel (orientato verso il) modulo B significa che A USA B [53].

5.1.3 Progetto e sviluppo per raffinamenti successivi

Questo metodo consiste nel raffinamento di una specifica software fino ad arrivare alla codifica verificando passo dopo passo la coerenza di ogni successivo raffinamento con quello precedente, che ha un più elevato livello di astrazione. In pratica bisogna riformulare successivamente le variabili e le operazioni della specifica di un modulo (che nel metodo B, ad esempio, è rappresentata con una macchina astratta), fino ad arrivare all'implementazione del modulo nel linguaggio di programmazione scelto.

Le figure 1 e 2 illustrano il meccanismo del raffinamento per il metodo B e per TRIO.

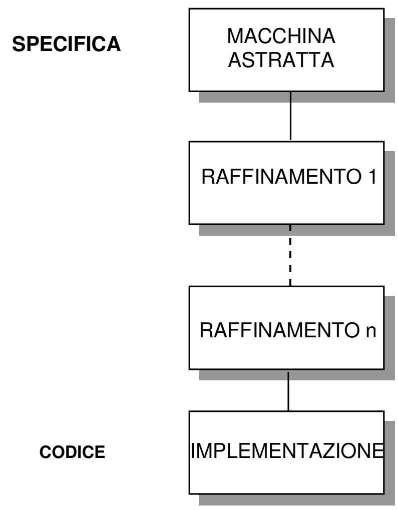

Fig. 5.1 meccanismo di raffinamento utilizzato nel metodo B

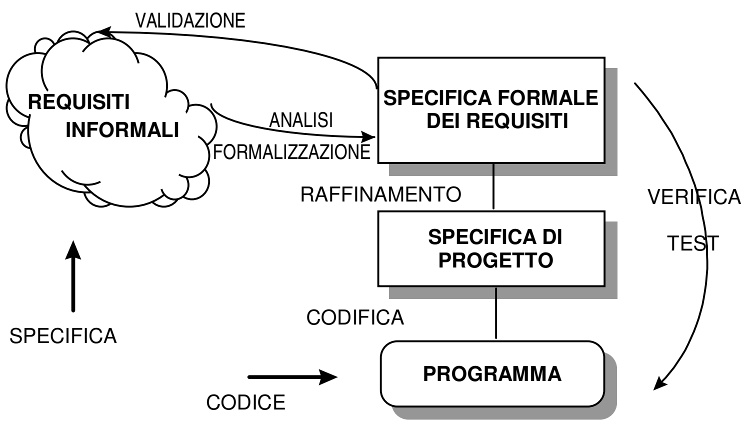

Fig. 5.2 ciclo di sviluppo del software con TRIO [54]

L'aspetto più importante, per l'applicazione di questa tecnica, è che i vari raffinamenti devono mantenere le stesse operazioni che sono state definite nella specifica. Per ogni raffinamento è necessario *provare* che il comportamento di un'operazione è compatibile con l'operazione che viene dettagliata.

Così facendo la codifica di un modulo viene a corrispondere effettivamente alla sua specifica, in quanto si passa dalla specifica all'implementazione, attraverso una serie di trasformazioni (che con TRIO ed altri linguaggi di specifica formale possono essere svolte in maniera automatica o semiautomatica), che aggiungono dettagli operazionali e implementativi, senza cambiare le proprietà fondamentali descritte nella specifica, fino a giungere ad un vero e proprio programma [98].

Nell'applicazione concreta della tecnica del raffinamento formale bisogna svolgere le seguenti operazioni:

- trasformare progressivamente le strutture dei dati astratti (insiemi, relazioni, funzioni etc.) in dati concreti (variabili scalari, vettori, matrici etc.);

- eliminare gradualmente il livello di non determinismo;
- sostituire le strutture astratte (parallelismo, scelta) con strutture concrete (sequenze, condizioni, cicli).

Purtroppo *il processo di raffinamento di una specifica non è del tutto automatico*, quindi ha bisogno di personale esperto e preparato. Ogni specifica può avere diversi possibili raffinamenti e nella scelta fra essi possono giocare un ruolo importante diversi fattori come ad esempio la sicurezza o le possibili prestazioni, comunque bisogna avere un'idea di ciò cui si mira, poiché il processo di raffinamento ha bisogno di una guida esterna.

Naturalmente alcuni particolari variano nel processo di raffinamento a seconda del formalismo adottato. *Nel metodo B il progetto completo è realizzato usando il meccanismo di composizione e decomposizione di più macchine astratte*. Una macchina astratta contiene variabili e operazioni. Essa permette di incapsulare le variabili, mentre le operazioni regolano l'accesso alle variabili e la loro manipolazione. Quando il livello di complessità del raffinamento di una macchina astratta è troppo elevato, si effettua una sua decomposizione in parti più semplici (altre macchine astratte). L'implementazione, che rappresenta il livello finale di raffinamento della macchina astratta, può realizzare le specifiche di una o più macchine astratte. Si costruisce un progetto con architettura stratificata dove, nei livelli più bassi, si trovano le macchine astratte di più basso livello, che a volte possono essere ricavate dalle librerie di macchine astratte fornite insieme agli strumenti del metodo B, le quali forniscono le strutture dati classiche; altre volte devono essere costruite dall'utente, che comunque ha a disposizione l'ausilio di un generatore automatico di lemmi intermedi da dimostrare e di un "dimostratore" automatico in grado di svolgere almeno l'80% delle prove, per verificare la consistenza dei raffinamenti successivi con le macchine astratte originarie.

Utilizzando uno sviluppo per raffinamenti successivi il tempo ed i costi impiegati sono inferiori rispetto ad uno sviluppo tradizionale nel quale la verifica formale viene svolta dopo la codifica [80]. In uno sviluppo tradizionale è molto difficile dimostrare che il codice scritto in un linguaggio di alto livello corrisponde ai requisiti di safety e a quelli funzionali*, invece con uno sviluppo per raffinamenti successivi può essere fatto più facilmente, verificando il progetto ad ogni passo;* infatti, usando lo stesso linguaggio di specifica formale per ogni livello del progetto, è possibile ragionare rigorosamente sul progetto, prima di passare alla codifica. Infine, appena il codice è pronto, è possibile verificare formalmente, in modo rapido ed efficace, che la codifica corrisponde alla sua specifica.

In realtà esiste anche un terzo metodo di progetto e sviluppo, che si pone a mezza strada tra quello tradizionale (con verifica formale dopo la codifica) e quello per raffinamenti successivi: lo *sviluppo con verifica in parallelo*. Tuttavia tale metodo è scarsamente citato nella letteratura [80] ed ancora più scarsamente usato poichè, se non c'è una buona intesa tra i due gruppi, si crea un solco profondo tra sistema e "sua" specifica formale, per cui si finisce per effettuare una semplice verifica formale dopo la codifica, più che una verifica in parallelo.

5.1.4 L'implementazione

Anche l'implementazione dà molti problemi agli sviluppatori del software. Da più parti si reclama che *per ottenere software con un certo livello di integrità, bisognerebbe che tutti gli strumenti software utilizzati a partire dai sistemi operativi, dai compilatori e dai linker fino ai sistemi di gestione di database fossero dello stesso livello di integrità software*, ma ciò, come notano Cullyer e Storey [41, 118], non avviene mai, in quanto questi strumenti essenziali non sono praticamente mai certificati per un livello di integrità adeguato.

Se poi bisogna sviluppare del software safety-critical in tempo reale, la situazione diventa ancora più difficile, poiché i sistemi operativi più noti come MS-DOS, Unix e Windows non sono real-time, dato che non impiegano nè osservano obblighi temporali. Inoltre, dagli studi di Cullyer e Storey [41, 39] emerge che gli strumenti in commercio utilizzati su una workstation forniscono soltanto una visione parziale del comportamento temporale del software e che essi possono essere usati soltanto in modo molto limitato quando il software è integrato sull'hardware obiettivo.

I linguaggi di programmazione, che i maggiori esponenti del mondo accademico vorrebbero fossero usati per il software safety-critical, sono Modula-2 e Ada. Secondo Neumann [104] bisogna scegliere linguaggi che richiedono essi stessi una certa disciplina di programmazione come C++, Modula-2+ e Modula-3, però nessuno di essi è l'ideale. C++, anche se contiene alcune caratteristiche moderne, è un linguaggio di basso livello, mentre Modula-3, anche se ha un grande potenziale, è ancora immaturo in termini di compilatori ed altri strumenti. Ada, secondo Neumann [104] è da escludere, in quanto presenta serie difficoltà, anche se esso, come osserva Storey [118], è obbligatorio per il software critico per le missioni spaziali dell'agenzia spaziale europea.

L'uso di linguaggi di programmazione di basso livello induce all'errore più dell'uso di linguaggi di programmazione di alto livello per programmi di pari complessità, perciò un linguaggio di basso livello scarsamente tipizzato come C andrebbe escluso a priori.

Il linguaggio C, inoltre, andrebbe evitato per i seguenti motivi:

- **il grandissimo uso di puntatori, che rende impossibile dimostrare la safety nelle operazioni di accesso e di assegnamento;**
- **l' ampio uso di strutture dati dinamiche, notoriamente disastrose per i requisiti temporali dei programmi;**
- **la mancanza di una solida semantica;**
- **la scarsa immediatezza notazionale che rende di scarsa efficacia le revisioni sul codice.**

Secondo Storey [118], C++, anche se migliore del suo progenitore, mantiene molti dei problemi intrinseci di C . L'aspetto paradossale è che, non solo C è il linguaggio più ampiamente usato nel mondo industriale (un vero e proprio standard di fatto, come osservano Bellini e Guidi in [12]), ma che, secondo lo studio di Cullyer e Storey [41], è anche il linguaggio con cui sono stati realizzati gli strumenti per il testing del software safety-critical.

Tutti i linguaggi menzionati hanno delle difficoltà con il tempo reale, che vengono in parte risolte con l'uso di loro sottoinsiemi.

Per la realizzazione di software safety-critical real-time l'uso di un sottoinsieme sicuro di un linguaggio di programmazione diventa inderogabile se si vogliono raggiungere i requisiti temporali del programma.

Burns e Wellings sostengono che *non si dovrebbero usare le strutture dati dinamiche*, come ad esempio le liste dinamicamente concatenate o gli alberi, in quanto il tempo di accesso a tali dati è impredicibile [31]. Inoltre bisognerebbe *evitare l'uso della ricorsione e dei cicli nei quali il numero di iterazioni non è limitato in maniera certa*, poiché, in tali casi, non si può predire il tempo di esecuzione. Infine è necessario *non servirsi dell'allocazione dinamica della memoria* a causa dell'incertezza se la memoria richiesta sarà ottenibile.

Tutte queste caratteristiche di implementazione sono state pienamente recepite dal progetto di standard EN 50128, quindi sembra chiaro che i suoi compilatori si rendono perfettamente conto che il software safety-critical ferroviario è anche real-time; non si capisce allora perché non richiedano anche l'uso di linguaggi di specifica formale che supportano il tempo reale: addirittura l'elenco, sia pure solo esemplificativo, dei metodi formali utilizzabili per la specifica dei requisiti e per il progetto non ne contempla neppure uno che abbia questa caratteristica.

L'implementazione del software di sistemi complessi pone il problema di come gestire questa complessità intrinseca. Burns e Wellings [31] propongono di risolvere questo problema principalmente con la *decomposizione e l'astrazione*, due tecniche ampiamente presenti nei linguaggi di programmazione modulari, come il Modula-2. La decomposizione si realizza suddividendo un programma in moduli, che possono essere compilati e verificati separatamente. La separazione dei moduli può essere aumentata grazie all'information hiding, di cui si è già parlato, che rende alcune variabili "visibili" solo per alcuni moduli. L'astrazione, nel contesto dei moduli, implica la rimozione dei dettagli non necessari dalla rappresentazione di un particolare componente di sistema, favorendo così il riutilizzo dei moduli.

I moduli del Modula-2 (come i packages di Ada) si prestano particolarmente bene a contenere al loro interno "*tipi di dati astratti*" che sono considerati il modo più sicuro per implementare un sottosistema. I tipi di dati astratti sono particolari costrutti di alcuni linguaggi di programmazione, che permettono una notevole astrazione sui dati; infatti si specifica un tipo di dati con una sua propria interfaccia e si "incapsula" insieme alle operazioni per manipolarlo e osservarlo.

La cosiddetta incapsulazione, che è tra l'altro una delle caratteristiche più apprezzate dei linguaggi orientati agli oggetti, consiste proprio nella protezione dalla manomissione delle strutture dati interne ad un modulo e dei loro meccanismi interni, regolamentandone l'accesso attraverso opportune operazioni che costituiscono l'interfaccia del tipo di dato astratto.

Oggi l'implementazione viene sempre più svolta con linguaggi orientati agli oggetti. Questa prassi è considerata positivamente da Neumann [104] in quanto *la programmazione orientata agli oggetti combina quattro principi di buona ingegneria: astrazione, incapsulazione, polimorfismo, ed ereditarietà.*

Le proprietà di astrazione e incapsulazione, ben presenti nel Modula-2, non bastano a classificare un linguaggio di programmazione nella categoria di quelli orientati agli oggetti. Secondo alcuni autori [53, 16] la caratteristica più qualificante, oltre alle precedenti, di un linguaggio orientato agli oggetti, è *l'ereditarietà*. I linguaggi di programmazione che hanno le proprietà di incapsulazione e astrazione (come Modula-2 e Ada), ma non di ereditarietà si definiscono "basati sugli oggetti" non "orientati agli oggetti" [16]. Secondo Burns e Wellings [31] un linguaggio di programmazione con un buon grado di astrazione, per potersi definire orientato di oggetti, deve avere le seguenti caratteristiche:

- *ereditarietà*: le classi/tipi possono essere derivate/i da altre/i classi/tipi e le classi derivate ereditano gli attributi e le operazioni delle classi genitrici;
- *inizializzazione automatica degli oggetti*: l'inizializzazione di un oggetto viene svolta automaticamente, come conseguenza della dichiarazione dell'oggetto, grazie a un programma di costruzione, mentre in un linguaggio di programmazione "basato sugli oggetti" come Modula-2 l'inizializzazione di un tipo di dato astratto dev'essere fatta esplicitamente durante la codifica;
- *polimorfismo*: il nome di un'operazione si riferisce a diverse operazioni e il tipo dello specifico oggetto processato determina quale delle operazioni è richiesta.

Strigini [119] sostiene che, grazie alle caratteristiche sopra indicate, un programmatore può creare astrazioni dei dati e usarle come tipi in modo conveniente e sicuro.

Certamente la programmazione orientata agli oggetti ha alcuni vantaggi innegabili come:

- *riutilizzabilità* dei moduli [53];
- *scarsa inclinazione agli errori* [119];
- *buona comprensibilità*, grazie all'assenza di dati globali [16];
- *struttura modulare* che facilita il lavoro di diversi team di programmatori.

Tuttavia a parere di Bihari e Gopinath [16] vi sono seri motivi per cui la programmazione orientata agli oggetti non è stata usata (e non dovrebbe essere usata) per i sistemi in tempo reale in maniera ampia come per i sistemi di altra natura. Secondo gli autori le motivazioni principali sono:

- i linguaggi orientati agli oggetti esistenti hanno caratteristiche temporali impredicibili;
- si ritiene che i sistemi orientati agli oggetti siano inefficienti, a causa della gestione dinamica della memoria e dei vincoli dinamici.

Anche Neumann [104] sostiene apertamente che C++, Modula-2+ e Modula-3 hanno serie deficienze nelle applicazioni real-time.

Una critica molto puntuale ed efficace all'uso dei linguaggi di programmazione orientati agli oggetti, nei sistemi safety-critical, è quella portata da Rosen [111], il quale sostiene che *l'ereditarietà non consente una reale incapsulazione, poiché per capire una classe si deve capire l'implementazione delle sue classi progenitrici, e se si cambia l'implementazione di una classe questo influenzerà tutte le sue classi discendenti.* Inoltre un difetto a ciò collegato è che man mano che un progetto si sviluppa il grafo originale dell'ereditarietà diventa distorto rispetto alla realtà, quindi inadeguato e fuorviante e sorge il bisogno di una riformulazione della struttura dell'ereditarietà. Naturalmente si può svolgere una tale riformulazione, ma ciò influenza tutto lavoro fatto fino ad allora, quindi è necessario porvi molta attenzione.

Infine alcuni autori [36, 111] sostengono che un progetto orientato agli oggetti si serve esclusivamente della tecnica bottom-up (e non di quella top-down). A onor del vero l'ultima obiezione è quella meno mirata, poiché le tecniche di information hiding sono per loro stessa natura bottom-up [53], quindi tale critica potrebbe essere rivolta a tutti i linguaggi che usano queste tecniche, compreso Ada che è ritenuto proprio da Rosen [111] il linguaggio migliore per i sistemi safety-critical. Inoltre i *sistemi software di grosse dimensioni* (come sono normalmente quelli ferroviari: Meteor di Ratp è costituito da 90.000 linee di codice) *non possono essere progettati che con una metodologia bottom-up* o perlomeno alternando tecniche top-down e bottom-up (cfr. [53]).

La critica all'uso dell'ereditarietà nei sistemi safety-critical è di più difficile confutazione, perciò Bihari e Gopinath [16] sostengono che l'ereditarietà può non essere usata in tali sistemi, preservando gli altri benefici della programmazione orientata agli oggetti.

Tenendo conto delle osservazioni viste fino ad ora, **i lingaggi di programmazione orientati agli oggetti attualmente esistenti non dovrebbero essere usati per la realizzazione di software safety-critical in tempo reale sia per le loro deficienze nell'affrontare il real-time [104] sia per i loro problemi legati all'ereditarietà.**

Le caratteristiche che *un linguaggio di programmazione deve assolutamente avere* per questi sistemi *sono incapsulazione e modularità*, come notato da Barnard [7] nello studio di un sistema di segnalazione ferroviaria; infatti queste caratteristiche sono sufficienti a far fronte al continuo bisogno di adattamenti delle applicazioni industriali, che devono avere la possibilità di adattarsi in tempi brevi e con facilità a nuovi mercati (con ad esempio regole di segnalazione diverse), a nuove interfacce con l'utente e soprattutto devono essere in grado di progredire nel tempo, aumentando le proprie funzionalità, qualora ve ne sia bisogno.

Pertanto, *un linguaggio di programmazione particolarmente adatto ai sistemi safety-critical potrebbe essere Modula-2 o meglio un suo sottoinsieme, privo di puntatori, variabili e strutture dati dinamiche, ricorsione*.

Un commento particolare merita la ricorsione; il suo uso infatti produce un codice elegante e semplice da analizzare, ma richiede una "forma mentis" che si acquista solo con un certo sforzo. L'equivalente formulazione iterativa, in genere più intuitiva, deve tuttavia, nei sistemi safety-critical, essere accompagnata da una stretta limitazione al numero di iterazioni, cosa che, d'altra parte, può essere fatta senza troppi danni vista la relativa semplicità di molti algoritmi di controllo.

Il Modula-2 ha un'elevata leggibilità, che facilita l'analisi statica dei programmi e, in particolare, le revisioni svolte dagli operatori.

Inoltre la semantica del Modula-2 è molto ben definita (è addirittura specificata formalmente, anche se purtroppo in VDM) per cui non ci possono essere ambiguità o incertezze sul significato nelle sue istruzioni, invece il C, secondo Storey [118] non ha neppure una solida definizione della sua semantica.

Storey [118] fa notare che nel linguaggio C alcuni aspetti non sono neppure definiti, come ad esempio l'azione da svolgersi in caso di overflow di un intero, e C++, anche se migliore del suo progenitore, mantiene molti dei problemi intrinseci di C. La norma EN 50128, tra i possibili linguaggi di programmazione, propone l'uso di un sottoinsieme (sicuro) di C, ma *questa proposta è priva di fondamento, in quanto non esistono sottoinsiemi di C o C++ adatti al software safety-critical, in quanto è impossibile verificare efficacemente i puntatori in questi linguaggi [118].*

Il Modula-2 è stato usato con successo nella realizzazione di alcuni progetti di software ferroviario di protezione e controllo, come ad esempio il software delle metropolitane di Parigi e Calcutta. Alcuni autori [118, 37] auspicano che il Modula-2 diventi *il linguaggio* utilizzato per l'implementazione del software safety-critical. Infine molti esponenti di primo piano del mondo accademico internazionale, come ad esempio Abrial, Cullyer, Goodenough, Mandrioli, Storey, Wichmann e Wirth, apprezzano molto questo linguaggio di programmazione. Purtroppo è anche vero che lo scarso utilizzo nel mondo industriale rappresenta un handicap non trascurabile; infatti la società che commercializza gli strumenti di supporto al metodo B, nonostante successi ottenuti con il Modula-2, ora ha approntato un traduttore delle specifiche B in C, C++ e Ada per evidenti motivi di mercato.

Alternative al Modula-2 potrebbero essere SPARK (un sottoinsieme di Ada), che è anche esso uno dei pochi linguaggi di programmazione con una semantica specificata formalmente, oppure Real-Time Euclid, che è stato realizzato proprio per applicazioni safety-critical in tempo reale e che è apprezzato da Neumann [104]. Purtroppo, come riconosce lo stesso Neumann, un linguaggio come Real-Time Euclid, realizzato appositamente per un mercato ancora piccolo come quello del software safety-critical, ha non pochi difetti quali: notevole costo di compilatori e strumenti di supporto, difficoltà di reperire programmatori, scarsità di investimenti, scarso uso che ne limita i possibili miglioramenti ottenuti grazie ai suggerimenti degli utenti. Queste limitazioni poi finiscono inevitabilmente per comportare una scarsa affidabilità dei compilatori e degli strumenti di supporto.

5.2 Verifica del software

5.2.1 Posizione della norma europea EN 50128 sulla verifica del software

La verifica del software è trattata nel capitolo 11 della norma EN 50128, con l'obiettivo di testare e valutare i prodotti di ogni fase del ciclo di sviluppo, per assicurare correttezza e consistenza rispetto ai prodotti e agli standard forniti come input a quella fase.

In input ci saranno tutti i documenti prodotti nella fase in esame e in quelle precedenti. I documenti prodotti in output (una volta terminati sistema e verifica) saranno:

- piano di verifica del software;
- rapporto sulla verifica dei requisiti del software;
- rapporto sulla verifica del progetto software;
- rapporto sulla verifica dei moduli software;

- rapporto sulla verifica del codice sorgente;
- rapporto sui test sui moduli software;
- rapporto sull'integrazione del software;
- rapporto sull'integrazione software/hardware.

Il piano di verifica del software sarà compilato allo scopo di dirigere le attività di verifica e naturalmente crescerà gradualmente, via via che si sviluppa il software e quindi diventano più chiare le necessità di verifica.

La clausola 11.4.4, certamente una delle più importanti (se non la più importante) del capitolo, stabilisce che il piano di verifica del software conterrà:

1. la selezione di strategie e tecniche di verifica;
2. la selezione ed utilizzazione dei dispositivi per il test del software;
3. la selezione e documentazione delle attività di verifica;
4. la valutazione dei risultati di verifica raggiunti;
5. la valutazione dei requisiti di reliability;
6. il ruolo e le responsabilità di coloro che sono coinvolti nel processo di verifica;
7. il grado di copertura dei test che dev'essere ottenuto.

La verifica dovrà eventualmente essere svolta da un organismo indipendente a seconda del livello di integrità software di ciò che si sta esaminando: in particolare, per i livelli di integrità software 3 e 4 si dovrà avere una verifica indipendente.

Ovviamente, come nello spirito di tutta la normativa, alla fine di ogni attività dovrà essere stilato un rapporto dettagliato.

Per quanto riguarda la verifica dei requisiti software, si stabilisce che dev'essere *verificata la conformità della specifica dei requisiti software alla specifica dei requisiti di sistema e alla specifica dei requisiti di safety di sistema,* rilevando le incompatibilità fra esse, *ma senza dare indicazioni concrete su come effettuare questa verifica*, che, come già detto nei paragrafi riguardanti la specifica dei requisiti, potrebbe essere svolta in maniera semplice e soprattutto sicura (grazie alla possibilità di effettuare delle vere e proprie dimostrazioni di consistenza), se tutti i requisiti, sia quelli software sia quelli di sistema e di safety, fossero specificati formalmente con lo stesso linguaggio.

Allo stesso modo si chiede di *verificare l'adeguatezza della specifica dell'architettura software e della specifica del progetto software nel soddisfare i requisiti software, nonchè l'adeguatezza della specifica del progetto dei moduli nel soddisfare la specifica del progetto software*, senza offrire nessuna indicazione veramente utile.

Dopo che è stata effettuata l'implementazione, si dovrà procedere *alla verifica del codice*, ogni modulo dovrà essere testato in base alla specifica dei test sui moduli software, dovrà essere verificata l'integrazione tra i moduli e infine si dovranno svolgere dei test in base alla specifica dei test sui requisiti software, i quali dovrebbero mostrare che sono stati correttamente raggiunti tutti i requisiti contenuti nella specifica dei requisiti software.

Gli articoli del capitolo 11 sono certamente troppo vaghi, poco incisivi, molto verbosi, e più interessati alla forma che alla sostanza delle verifiche, a giudicare dalla pedante elencazione dei vari rapporti, documenti e annessi cartacei da produrre con la verifica, senza nel contempo fornire veri suggerimenti su come svolgere la verifica, ma ciò che veramente sconcerta sono le tabelle *normative* allegate a questo capitolo.

In base a queste tabelle, per i livelli di integrità software 1 e 2, la verifica dovrà essere svolta per mezzo di prova formale oppure testing, invece per i livelli di integrità software 3 o 4, le combinazioni di tecniche da usare sono: 1 e 4 (cioè prova formale e testing) oppure 3 e 4 (cioè analisi statica e testing).

Dare la possibilità di *escludere le prove formali dal processo di verifica è in netto contrasto con la prassi adottata nei più recenti sviluppi (di successo) di software ferroviario safety-critical*, nei quali la verifica formale ha svolto il ruolo principale nei processi di verifica, convalida e certificazione [51]. A ciò si aggiunga che *nessuna delle tecniche di analisi statica, giudicate (e sperimentate) più efficaci* dagli esperti del mondo accademico e di quello industriale, quali Software Fault Tree Analysis o Path Specification Analysis, *è stata inclusa tra quelle imposte dalla tabella normativa riguardante l'analisi statica*. In compenso vengono "altamente raccomandati" oppure semplicementi "raccomandati" metodi considerati di scarsissima (o nessuna) utilità, come ad esempio la Sneak Circuit Analysis, che, come si capisce anche dal nome, è efficace con l'hardware, ma non con il software.

5.2.2 L'analisi statica

Analisi statica, testing e verifica formale vengono trattate separatamente per ottenere una maggiore chiarezza di esposizione, ma in realtà traggono grandi benefici dal loro uso in contemporanea, si supportano e si complementano a vicenda e, soprattutto analisi statica e testing (oltre naturalmente alla verifica formale), ottengono risultati migliori e più rapidi in presenza di specifica formale e di progetto formale del software. Le tecniche di analisi statica sviluppate in questi anni sono molte, ed ognuna di esse ha sostenitori e detrattori, anche se, in realtà, sono stati compiuti pochi studi sugli effetti di miglioramento della safety del software esaminato con questi metodi.

Bologna e altri, in [10], propongono i seguenti metodi di analisi statica:

- *analisi dei valori limite*: si studiano i valori limite dei parametri di input e delle variabili interne del programma come, ad esempio, divisori nulli, stringhe vuote, puntatori non assegnati. Si valutano le possibili conseguenze dell'inserimento di tali valori limite. Di solito questo tipo di analisi fornisce anche le basi (gli input) per l'elaborazione di test sulla performance del software in presenza dei valori limite trovati ed esaminati proprio con tale tecnica statica;
- *analisi del cammino / analisi del flusso di controllo*: si svolge un'analisi del software per rilevare strutture del programma di bassa qualità e non corrette. Si rappresenta il programma con un grafo orientato che individua i possibili cammini al suo interno, si esaminano le condizioni necessarie perché venga seguito un certo cammino e si opera una progressiva semplificazione del grafo. Questo metodo permette di identificare problemi quali cicli infiniti, parti di codice inaccessibili e strutture di bassa qualità;
- *analisi del flusso dei dati*: l'analisi viene svolta per mezzo di un diagramma che rappresenta il flusso dei dati attraverso il programma. Tale diagramma mostra separatamente ogni trasformazione che avviene sui dati, mentre l'informazione passa

dall'input all'output. Si esamina (e si verifica) la correttezza delle operazioni che vengono svolte e dei flussi di dati, effettuando un confronto fra il flusso di dati attuale e quello richiesto;

- *software sneak analysis (o circuit sneak analysis)*: si svolge un tipo di analisi che, a onor del vero, è più appropriata per l'hardware, per il quale è nata, che per il software. Gli "sneak circuits" (o circuiti infidi) sono condizioni latenti progettate e sviluppate inavvertitamente in un sistema, le quali, in certe circostanze, possono portare ad un malfunzionamento. Nell'hardware essi sono veri e propri cammini fisici. In pratica il programma viene rappresentato tramite dei diagrammi ad albero, servendosi dei simboli grafici dei circuiti. Questi alberi vengono poi analizzati usando una lista di verifica standard di possibili tipi di errore. Leveson [88] critica pesantemente questa tecnica, annoverandola tra quelle che servono solo a buttare via delle risorse, in quanto esistono rappresentazioni del software molto migliori, e la maggior parte di questo tipo di analisi potrebbe essere svolta direttamente sul codice. Certamente, come rimarcato da Leveson e Shimeall [88, 114], le tecniche di analisi che usano strumenti software standard (in questo caso le liste di verifica) e che cercano di ricondurre il software a modelli caratteristici non trovano molti errori;

- *ispezioni di Fagan*: questa tecnica consiste nell'esame e nella revisione di tutto il processo di sviluppo a partire dalla specifica per arrivare fino alla codifica. Questo metodo trova grande ausilio in una specifica e in un progetto formali, poiché comporta una discussione tra diversi esperti, fra cui progettisti e verificatori (indipendenti per i livelli di integrità software 3 e 4) e, se non è ben chiaro - soprattutto ai verificatori - quello che il software deve fare, questo, come altri tipi di revisione, sarà di scarsa efficacia. Secondo Taylor [120] *questo metodo, pur richiedendo un lavoro molto intenso, trova addirittura il 90% degli errori, fra cui anche molti errori di specifica*;

- *revisioni del codice*: in questo tipo di analisi, uno degli sviluppatori dovrebbe far compiere una "visita guidata" del codice ai verificatori e convincerli della correttezza del programma. Per questo tipo di revisione del codice (come per quelle del progetto) la specifica formale è di grande aiuto per evitare che gli stessi fraintendimenti della specifica fatti dagli sviluppatori vengano commessi anche dai verificatori, cosa molto probabile in assenza di specifica formale visto che uno dei progettisti fa da guida alla revisione;

- *simulazione simbolica*: questo metodo si propone di verificare la corrispondenza tra codice sorgente e sua specifica. Il programma viene eseguito con l'aiuto di un simulatore, ma, al posto dei dati di input, si usano variabili algebriche. Le istruzioni di assegnamento non producono valori numerici, ma espressioni algebriche per le variabili prese in considerazione. Le formule risultanti possono poi essere confrontate con quelle predette dalla specifica. Come risultato dell'analisi si ha un insieme di formule per ogni possibile cammino attraverso il software. *Una analisi completa di un programma (anche dei più semplici) è quasi sempre impossibile*, anche usando gli strumenti a disposizione, perciò è necessario scegliere solo alcuni cammini d'interesse. Purtroppo quando i verificatori operano delle scelte (specie se il loro grado di indipendenza non è troppo elevato) c'è il rischio, come osservato causticamente da Leveson [88] che facciano delle scelte opportunistiche dettate dalla maggiore o minore facilità di verifica; in pratica “che per evitare di cercare il classico ago nel pagliaio (con un'analisi completa) cerchino l'ago dove c'è più luce invece che dov'è caduto”;

- *cammino predicato / analisi della specifica*: il concetto di base di questo metodo è che ogni cammino nel programma deve corrispondere ad un caso nella specifica. “Combinando tutte le condizioni logiche per seguire un certo cammino (predicato del cammino) diventa possibile verificare che le condizioni corrispondono alla specifica” come spiegato da Taylor [120] che giudica questa tecnica di grande efficacia;
- *software Failure Modes and Effects Analysis*: questo metodo, che può essere considerato sia una tecnica di analisi statica sia una tecnica di analisi degli azzardi software, fu ideato da Reifer [110] ed è stato recentemente rilanciato da Goddard [55]. La software FMEA non è contemplata in [10] e, in realtà, ha sempre dato risultati piuttosto scarsi, al punto che molti autori (v. ad esempio [122, 88, 118]) sostengono l'inapplicabilità della FMEA al software. Essa però è molto caldeggiata dagli esperti dell'hardware e dell'analisi degli azzardi, la cui opinione, pur trascurando completamente le peculiarità del software, è molto influente. La software FMEA è una tecnica di analisi induttiva svolta generalmente già dopo il completamento del progetto software e poi di nuovo in modo iterativo, seguendo le modifiche al progetto e alla codifica dovute al rilevamento di errori durante la verifica o a mutamenti di funzionalità. Si prende ogni unità software, si identificano tutti i suoi modi di fallimento e si valuta il loro effetto sugli altri componenti software e hardware, nonché sull'intero sistema. Inoltre ogni modo di fallimento è classificato in base alla sua severità. L'analisi identifica tutti i modi nei quali un fallimento del software può influenzare il sistema. Naturalmente, come d'altra parte succede anche con FMEA, questo metodo riesce a considerare soltanto i fallimenti provocati da un singolo errore e non quelli provocati da più errori/difetti software combinati;
- *software Fault Tree Analysis*: questa tecnica è giudicata da Leveson [88] l'unica efficace fra quelle di analisi statica per la verifica e valutazione della safety. Inoltre essa è considerata in maniera molto positiva anche daTaylor, Bologna e altri [120, 10]. Certamente i suoi numerosi punti in comune con la verifica formale ne aumentano molto l'efficacia, soprattutto quando essa è *combinata con tecniche di analisi degli azzardi di alta qualità e di specifica formale*, che permettono di rilevare rispettivamente (e prontamente) gli azzardi, che costituiscono la radice degli alberi costruiti con questo metodo, e le precondizioni foglia, quando l'azzardo è provocato da un errore di specifica. La caratteristica forse più interessante di questo metodo è proprio *la possibilità reale di scoprire degli errori di specifica*, in quanto esso parte dai risultati dell'analisi degli azzardi e non dalla specifica, come avviene nella maggior parte delle tecniche di verifica. Un software Fault Tree può avere diversi gradi di formalità. L'analisi può essere svolta semplicemente come una revisione informale orientata alla safety, o più formalmente come una dimostrazione assiomatica di proprietà di safety. Naturalmente la seconda alternativa è di gran lunga la più efficace, la più pienamente supportata da strumenti e non è pesantemente dipendente dal talento (o dall'errore) umano, pertanto essa verrà

analizzata estensivamente. La software Fault Tree Analysis deriva la simbologia grafica (e il nome) dalla Fault Tree Analysis degli azzardi, ma non il tipo di ragionamento che invece è lo stesso usato nella verifica formale assiomatica deduttiva da Hoare [70] e soprattutto da Dijkstra [43]. Si prende in considerazione un output del programma che può creare azzardo, lo si pone alla radice dell'albero e si cerca all'indietro un percorso nel software che può portare a tale output. Se si trova una contraddizione logica nell'albero dei difetti software, per la quale non si può raggiungere l'evento radice (che, di solito, è un'azione di controllo non sicura), allora si può escludere il verificarsi dell'azzardo, in quanto esso è contrario alla logica specificata nel codice. Se al contrario esiste un percorso logico, che conduce attraverso il programma fino alla radice, allora le foglie dell'albero (che sono le cause primarie, o condizioni di input, dell'output pericoloso) rappresentano le precondizioni che possono arrecare danno alla safety del programma o del modulo di programma analizzato. *Se viene trovato un percorso che conduce all'output non sicuro esso va eliminato, senza tentare di operare una quantificazione probabilistica della sua possibilità di occorrenza*, come avviene invece nell'analisi dell'albero dei guasti dell'hardware.

La software Fault Tree Analysis è un'applicazione grafica della verifica assiomatica, dove le postcondizioni descrivono le condizioni che creano azzardo invece che le condizioni di correttezza [88]. Per evitare alberi troppo grandi, ingestibili anche con gli strumenti a disposizione, l'analisi lungo un ramo si ferma appena si scopre che la precondizione immediatamente a monte (che rappresenta la “weakest precondition”) di un'istruzione, necessaria per il verificarsi dell'azzardo in considerazione, è falsa quindi la postcondizione non potrà mai essere soddisfatta. Al contrario tutte le volte che la precondizione immediatamente a monte è vera, bisogna continuare il percorso su quel ramo per verificare se è possibile arrivare a condizioni di input, a causa delle quali il programma può provocare l'azzardo, quindi è intrinsecamente privo di safety. La software fault tree analysis si può applicare sia a programmi scritti con linguaggi di alto livello, come dimostrato da [64] e da [38], quest'ultimo con specifico riferimento a software safety critical scritto in Ada, sia a linguaggi di basso livello come l'Assembler, come ribadito in molteplici occasioni dagli ideatori di questa tecnica, J. R. Taylor e N. Leveson.

Secondo Bologna e altri [10] *la software Fault Tree Analysis può rilevare tutti gli errori sia di specifica sia di implementazione, a patto che sia svolta in modo formale, e che sia basata su un'analisi integrata dell'albero dei guasti per il sistema sul quale lavora il software*. Taylor [120] sostiene che l'analisi può essere svolta automaticamente in pochi giorni anche per un sistema con più di un milione di linee di codice, anche se ammette che i programmi non dovrebbero essere troppo complessi a livello di cicli, ricorsione, e calcoli per evitare che gli alberi dei guasti diventino eccessivamente grandi. Infatti con alberi troppo grandi diventa difficile organizzare e verificare i risultati. Sono stati sviluppati molteplici strumenti per automatizzare questo metodo, ma bisogna tenere conto del fatto che in presenza di cicli, come anche un semplice *while...do*, è necessario l'intervento umano, come dimostrato da Friedman e Voas [50]. Una condizione necessaria per svolgere un'analisi di buona qualità è poi la specifica formale dei requisiti software. In caso di specifica informale può non essere chiaro ai verificatori *se un azzardo è dovuto a una specifica errata o a un suo* *fraintendimento durante il progetto o la codifica*, pertanto c'è il rischio che, si elimini un percorso che conduce a un azzardo, ma se ne lascino altri con la stessa meta, se non ci si è accorti che l'errore è proprio nella specifica. Questo rischio è tutt'altro che improbabile, poiché gli errori e fraintendimenti di specifica rappresentano la classe più pericolosa e più numerosa. Inoltre un'analisi completa di tutti i percorsi di un programma complesso è spesso impossibile a causa dell'eccessiva ampiezza e ramificazione dell'albero, ragion per cui si può credere di aver raggiunto le foglie o cause primarie troppo presto e non prendere nella dovuta considerazione alcuni percorsi lunghi e tortuosi che portano fino alla specifica. Da queste considerazioni nasce l'ovvia conseguenza che si cerchi di eliminare un percorso che porta ad un azzardo, agendo a livello di semplice codifica, cosa più che naturale visto che, come detto nei capitoli precedenti, agire a livello di specifica costa oltre 200 volte di più (quando il codice è già pronto). Se tuttavia l'azzardo è causato da una specifica errata e ambigua, si possono lasciare inalterati altri cammini, che arrivano, seguendo altri rami, dall'azzardo alla medesima specifica sbagliata. D'altra parte non è raro che accada che gli sviluppatori prima e i verificatori dopo fraintendano una specifica non formale con gravi conseguenze.

Infine bisogna notare che *il successo (o l'insuccesso) di questo metodo dipende dalla qualità dell'analisi degli azzardi* effettuata, dal momento che i percorsi che portano ad azzardi non identificati non verranno neppure presi in considerazione (non si può far crescere un albero senza radice) e, come già detto nel capitolo sulla maieutica dei requisiti, le tecniche attualmente in uso sono incomplete e andrebbero integrate.

5.2.3 Il testing e i modelli per la valutazione della reliability

Il testing consiste nella verifica dinamica, cioè tramite esecuzione, della correttezza di un software; si esegue il programma senza conoscerne la struttura e con dati scelti in base alla specifica (nel caso del black box testing), oppure si esegue il programma con dati di test scelti accuratamente in base alla struttura del codice oltre che alla specifica (nel caso di white box testing). "Un programma *P* è considerato una funzione da un insieme di dati *D*, il dominio della funzione, in un insieme di dati *R*, il codominio. Il risultato *P(d)* ottenuto eseguendo il programma *P* sul dato d'ingresso *d*, con *d* appartenente a *D*, è corretto se soddisfa le specifiche, non corretto se diverso dal risultato previsto dalle specifiche del programma *P*" [53].

Il testing rappresenta il modo più comune di quantificare la reliability intesa come la misura della probabilità che difetti/guasti nel software causino un fallimento durante uno specificato periodo di attività e in uno specificato ambiente d'uso. La reliability può essere rappresentata con diversi modelli matematici, che forniscono informazione quantitativa sulla probabilità di corretta (o scorretta) esecuzione di un programma in base alla "storia" dei fallimenti del software rilevati durante il testing. Nel seguito si analizzeranno i modelli più comunemente usati per una valutazione della reliability:

1. **Modelli basati sulla "semina" dei difetti:** in questi modelli si semina un certo numero di difetti/errori in un programma che si assume abbia un numero sconosciuto di errori. Si testa il software e si contano gli errori seminati e non seminati che vengono rilevati. Da questa misura si ottiene una stima degli errori

presenti nel programma prima della semina e quindi una stima della reliability. Questi modelli sono indipendenti dal tempo. In questa classe il più conosciuto è probabilmente il *modello di Mills*. La procedura adottata è simile a quella utilizzata per stimare il numero di pesci presenti in uno stagno o il numero di animali selvaggi presenti su un certo territorio [129].

2. **Modelli basati sul dominio di input:** in questi modelli si genera un insieme di casi di test da una distribuzione di input, che si assume che sia rappresentativa dell'utilizzo operativo del programma sotto esame. Il dominio di input viene diviso in un insieme di classi di equivalenza. Di solito si associa ogni classe di equivalenza a un cammino del programma. Si ottiene una stima della reliability del software grazie ai fallimenti che vengono rilevati durante l'esecuzione dei casi di test presi come campione del dominio di input. Anche questa classe di modelli, come la precedente, è indipendente dal tempo. Si prendono in considerazione il modello di Nelson e il modello di Bastani e Ramamoorthy, certamente tra i più utilizzati [129, 30].

   - *Modello di Nelson*: si misura la reliability di un programma eseguendolo con un campione di n input, i quali vengono scelti casualmente dall'insieme dominio di input E, con E = (E_i: i = 1, ... ,N) ed E_i = insieme di valori dei dati necessari per un'esecuzione del programma. Si fa la scelta casuale del campione di n input in accordo con la distribuzione di probabilità P_i. L'insieme (P_i: i = 1, ... ,N) è il profilo operativo o semplicemente la distribuzione di input dell'utente. Sia n_e il numero di input che hanno provocato dei fallimenti durante l'esecuzione, allora una stima della reliability del programma è data da:

     $$R = 1 - n_e/n. \qquad (1)$$

     Con questo modello si può verificare il caso che l'insieme dei dati di test, usato durante il testing, non sia rappresentativo dell'uso operativo che ci attende. I modelli di questo tipo ipotizzano che ci sia una conoscenza pregressa della distribuzione di utilizzo operativo, che è realmente possibile solo se ci sono sistemi simili già in uso da qualche tempo, cosa non sempre probabile nei sistemi safety-critical, che hanno, in ogni caso delle peculiarità notevoli e un mercato ancora relativamente piccolo.

     Modello di Bastani e Ramamoorty: è usato soprattutto per la valutazione della reliability di software per il controllo di sistemi safety-critical in tempo reale, in ambito industriale [9]. In questi sistemi non dovrebbe essere rilevato nessun fallimento durante la fase di stima della reliability, in modo che essa risulti pari a uno. Pertanto, la misura più importante è la fiducia nella stima della reliability. Il modello fornisce una *stima della probabilità che il programma sia corretto per tutti i possibili input, se è corretto per uno specificato insieme di input*. L'ipotesi basilare è che il superamento di ogni caso di test fornisce almeno qualche informazione stocastica circa il comportamento del programma per altri punti, che sono vicini al punto preso in esame nel test. Questo modello permette di usare qualsiasi strategia di selezione di casi di test. Quindi è possibile minimizzare il testing, scegliendo casi di test che operano proprio sui costrutti più inclini agli errori.

3. **Modelli basati sui tempi intercorsi fra i fallimenti:** in questi modelli, che sono tra i primi usati per la valutazione della reliability del software, si assume che il tempo intercorso fra il fallimento (i-1)-mo e il fallimento i-mo segua una distribuzione i cui parametri dipendono dal numero di difetti che rimangono nel programma durante questo intervallo. Le stime dei parametri si ricavano dai valori osservati per quanto riguarda i tempi che intercorrono fra i fallimenti, mentre le stime della reliability del software, tempo medio per il prossimo fallimento ecc. si ricavano dal modello usato. Dopo che viene rilevato il difetto del software, che ha provocato il fallimento, questo viene rimosso. Tra questi modelli, i più importanti per diffusione sono: il modello di Jelinski e Moranda, il modello di Shick e Wolverton, il modello di Goel e Okumoto (o del debugging imperfetto) e il modello bayesiano di Littlewood e Verral.

   - *Modello di Jelinski e Moranda*: è forse il modello più comunemente usato. Si assume che:

     a. ci siano N difetti/errori software all'inizio del testing;

     b. ogni difetto sia indipendente dagli altri e abbia uguale probabilità di provocare un fallimento durante il testing;

     c. ogni difetto rilevato sia rimosso in un tempo trascurabile e non siano introdotti nuovi difetti durante quest'operazione;

     d. Il tasso di fallimento software sia proporzionale al numero di difetti del programma;

     sotto queste ipotesi, il tasso di fallimento software è dato da:

     $Z(Ti) = \theta * [N - (i - 1)]$

     dove θ è una costante di proporzionalità, T_i è il tempo intercorso tra il fallimento (i-1)-mo e il fallimento i-mo. Il tasso di fallimento software diminuisce proporzionalmente a θ ogni volta che si rimuove un difetto.

   - *Modello di Schick e Wolverton*: si basa sulle stesse ipotesi a, b, c, d del modello di Jelinski e Moranda con in più l'ipotesi che il tasso di fallimento software (o funzione d'azzardo) sia proporzionale, oltre che al numero di difetti del programma, anche al tempo intercorso fra il fallimento (i-1)-mo e quello i-mo, cioè a Ti.

   - *Modello di Goel e Okumoto* (o del debugging imperfetto): nasce dalla constatazione che, in realtà, non sempre i difetti vengono rimossi con certezza, come supposto nei modelli precedenti. In questo modello, si considera il numero di difetti presenti nel sistema al tempo t, Y(t), un processo di Markov le cui probabilità di transizione sono governate dalla probabilità di debugging imperfetto. I tempi che intercorrono tra le transizioni di Y(t) si considerano distribuiti esponenzialmente con tassi che dipendono dal numero di guasti presenti al momento nel sistema.

- *Modello Bayesiano di Littlewood e Verrall*: questo modello [92, 89, 91] segue un'ottica diversa dai precedenti, in quanto non considera la reliability del software in termini di numero di difetti/errori nel programma. Si assume che i tempi che intercorrono fra due fallimenti successivi seguano una distribuzione esponenziale, ma il parametro di questa distribuzione viene considerato come una variabile casuale con una distribuzione gamma. Le intensità di fallimento (o tassi di fallimento) software formano una sequenza stocasticamente decrescente.

4. **Modelli basati sul conteggio dei fallimenti o dei difetti**: in questi modelli ci interessa di più il numero di difetti o di fallimenti in intervalli di tempo specificati piuttosto che i tempi che intercorrono fra i fallimenti. Appena i difetti sono rimossi dal sistema, ci si aspetta che il numero di fallimenti rilevati per unità di tempo decresca. Il tempo che si considera in questi modelli può essere il tempo vero e proprio, ma anche il numero di casi di test eseguiti o qualche altra misura rilevante. In questo contesto, gli intervalli di tempo possono essere fissati a priori e si considera il numero di fallimenti rilevati in un intervallo come una variabile casuale. Si suppone che i conteggi dei fallimenti seguano un processo stocastico conosciuto con un tasso di fallimento dipendente dal tempo, nel discreto o nel continuo. Si possono stimare i parametri del tasso di fallimento dei valori osservati dai conteggi dei fallimenti o dai tempi di fallimento. L'idea che sta alla base della maggior parte di questi modelli è quella della distribuzione di Poisson, che si è dimostrata eccellente in molti campi di applicazione. Per meglio capire le caratteristiche di questa classe di modelli ne verranno esaminati in dettaglio alcuni:

   - *modello di Goel e Okumoto* (o del processo non omogeneo di Poisson): Goel e Okumoto [56] assumono che un programma sia soggetto a fallimenti che si verificano in istanti di tempo casuali e sono causati da difetti presenti nel software. Anche questo è un modello di crescita della reliability come il modello bayesiano di Littlewood - Verrall e tutti i modelli basati sui tempi della classe precedente. La decisione di non trattare tutti insieme i modelli di crescita della reliability, come viene fatto comunemente in molti testi, è dovuta ad un raggruppamento per omogeneità di ipotesi che apparirà ancora più chiaro in seguito.

   - *Modello di Goel del processo di Poisson non omogeneo generalizzato*: i modelli basati sui tempi che intercorrono tra i fallimenti e quelli basati sul conteggio dei guasti assumono quasi sempre che un programma manifesti un tasso di fallimento decrescente col procedere del testing, quindi che la qualità del software migliori sempre più, grazie all'individuazione e alla rimozione dei difetti, ma ciò non è sempre vero. Si è notato che in molte situazioni reali, il tasso di fallimento prima aumenta poi diminuisce col procedere dell'attività di testing. Questa situazione di incremento seguito da decremento del tasso di fallimento è stata ulteriormente elaborata da Goel.

   - *Modello di Musa o del tempo di esecuzione*: Musa [102] fa le stesse ipotesi presenti nel modello di Jelinski e Moranda con l'eccezione che il processo modellato è il numero di fallimenti in specificati intervalli di esecuzione. Il tasso di fallimento software è dato da:

$z(\tau) = \phi * f * (N - n_C)$

dove τ è il tempo impiegato nell'esecuzione del programma fino all'istante attuale, f è la frequenza lineare di esecuzione (tasso medio di esecuzione di un istruzione del programma diviso per il numero di istruzioni), ϕ è una costante proporzionale all'esposizione ai difetti e mette in rapporto questa con la frequenza di esecuzione lineare, e n_C è il numero di difetti/errori corretti nell'intervallo $(0, \tau)$. *Questo modello mette in risalto la dipendenza del tasso di fallimento software dal tempo di esecuzione*.

- *Modello esponenziale di Shooman*: Shooman [115] elabora un modello che non si discosta molto da quello di Jelinski e Moranda. Il tasso di fallimento software (o funzione d'azzardo) è rappresentato da:

  $z(t) = k * [N/I - n_C(\tau)]$

  dove t è il tempo operativo del sistema a partire dalla sua attivazione, I è il numero totale di istruzioni del programma, τ è il tempo di debugging, $n_C(\tau)$ è il numero di difetti corretti durante τ, normalizzato rispetto a I, e k è una costante di proporzionalità.

- *Modello di Poisson generalizzato*: Musa e altri descrivono questo modello tra le diverse varianti del modello di Goel-Okumoto [97].

- *Modello di Musa e Okumoto* (o *modello di Poisson logaritmico*): questo modello, nato nel 1984 e descritto in [97], supera il principale difetto del modello di Musa, che era quello di assumere che tutti i difetti contribuissero ugualmente al tasso di fallimento del programma in esame. Qui invece si sottolinea come ogni difetto presente nel software contribuisca in modo diverso, con un suo proprio tasso di fallimento, al tasso di fallimento del programma. Prima viene scoperto un difetto, più alto è il suo tasso di fallimento, mentre i difetti/errori rilevati successivamente hanno un tasso di fallimento inferiore.

  Questo modello può essere considerato un'evoluzione di una variazione, elaborata da Moranda, del modello di Jelinski e Moranda, quindi in ultima analisi un'evoluzione di quest'ultimo. Inoltre esso è un modello di crescita della reliability, quindi è chiaro che i possibili raggruppamenti sono molteplici e tutti in qualche misura giustificabili e al tempo stesso in parte arbitrari. La classificazione adottata in questo lavoro si basa sull'omogeneità delle ipotesi dei modelli appartenenti a ciascun gruppo.

Una classificazione alternativa, molto in uso, raggruppa in un unico insieme tutti i modelli di crescita della reliability, riunendo modelli spesso strettamente imparentati, ma che, a volte, hanno alla base ipotesi diverse. Comunque sono modelli di crescita della reliability il modello Bayesiano di Littlewood e Verrall, il modello di Jelinski e Moranda, il modello di Musa e Okumoto, il modello di Poisson generalizzato, il modello di Stick e Wolverton, il modello esponenziale di Shooman, il modello di Musa o del tempo di esecuzione, il modello di Goel e Okumoto. Altre possibili classificazioni potrebbero raggruppare tutti i modelli derivati da un comune progenitore, oppure quelli dipendenti dal tempo o quelli indipendenti.

Si esaminano, ora, le ipotesi che accomunano i vari gruppi e ne determinano in larga misura il campo di applicazione.

Le ipotesi chiave su cui poggiano i modelli basati sulla semina dei difetti sono:

- *distribuzione casuale nel software dei difetti seminati;*
- *uguale probabilità di essere rilevati dei difetti originali e di quelli seminati.*

Le ipotesi chiave su cui poggiano i modelli basati sul dominio di input sono:

- *distribuzione conosciuta del profilo di input;*
- *utilizzo di testing casuale;*
- *possibilità di dividere il dominio di input in classi equivalenti.*

Le ipotesi chiave su cui poggiano i modelli basati sui tempi che intercorrono tra i fallimenti sono:

- *indipendenza dei tempi tra i fallimenti;*
- *uguale probabilità di osservazione di ogni difetto;*
- *indipendenza dei difetti l'uno dall'altro;*
- *rimozione dei difetti che hanno provocato un fallimento dopo che esso si è verificato;*
- *rimozione perfetta dei difetti, cioè non devono essere introdotti nuovi difetti durante la correzione.*

Le ipotesi chiave su cui poggiano i modelli basati sul conteggio dei difetti sono:

- *indipendenza l'uno dall'altro degli intervalli di testing;*
- *testing omogeneo negli intervalli;*
- *indipendenza l'uno dall'altro dei numeri di difetti trovati in intervalli che non si sovrappongono.*

L'elencazione delle ipotesi sulle quali si basa ciascun gruppo analizzato non è un puro esercizio accademico, ma serve per poter operare una rapida sintesi delle aree e delle fasi di applicazione dei modelli esaminati.

Durante il **testing dei singoli moduli**, *non può essere applicato alcun modello basato sui tempi che intercorrono tra i fallimenti, in quanto viene pesantemente violata l'ipotesi che i tempi che intercorrono tra i fallimenti siano indipendenti*; infatti la strategia utilizzata per testare un singolo modulo non può essere il testing casuale**.**

I modelli basati sulla semina dei difetti di solito sono applicabili, anche se è preferibile che chi esegue i test sia una persona diversa dal programmatore.

I modelli basati sul dominio di input sono applicabili, ma, a volte, possono non essere utili, se i casi di test generati dal dominio di input del modulo in esame non sono un campione rappresentativo della distribuzione di utilizzo operativa.

Nel **testing d'integrazione** bisogna distinguere il caso in cui i casi di test possono essere generati casualmente da una distribuzione di input da quello in cui essi possono essere generati in modo deterministico usando una strategia per i test (ad esempio analisi dei valori al limite, o testing dei cammini). *Nel primo caso i modelli basati sui tempi fra i fallimenti e quindi basati sul conteggio dei difetti sono i più applicabili*, però sono applicabili anche la maggior parte degli altri modelli sia pure con qualche restrizione.

I *modelli basati sul dominio di input possono essere usati*, ma dovrebbero utilizzare una distribuzione dei dati di test che sia statisticamente equivalente alla distribuzione operativa reale.

I *modelli basati sulla semina dei difetti possono essere usati*, in quanto, grazie alla natura casuale del processo di generazione dei test, si può considerare ragionevolmente rispettata l'ipotesi di uguale probabilità di rilevamento dei difetti seminati e di quelli originali.

In caso di **testing deterministico,** *i modelli basati sui tempi che intercorrono fra i fallimenti, a volte, non sono appropriati* a causa del mancato rispetto della ipotesi di indipendenza dei tempi fra i fallimenti, che si verifica facilmente ad esempio nell'analisi dei valori al limite.

I *modelli basati sul conteggio dei difetti sono applicabili* se gli insiemi dei casi di test sono indipendenti l'uno dall'altro, anche se i test vengono scelti in maniera deterministica in un certo insieme. Questo succede perché in tali modelli si assume che il tasso di fallimento del sistema software diminuisca con l'esecuzione di un insieme di casi di test e non ad ogni fallimento.

Nel **testing di accettazione** *seminare dei difetti non è molto pratico* e gli errori scoperti di solito non sono immediatamente corretti a causa della notevole onerosità in termini sia economici sia temporali di rimuovere un errore trovato in fase così avanzata. Pertanto, in genere, *sono applicabili solo i modelli basati sul conteggio dei difetti e quelli basati sul dominio di input.*

Durante *la vita operativa del software*, questo, di solito, viene monitorato con *modelli basati sul conteggio degli errori,* in quanto gli input forniti dall'utente di solito non hanno distribuzione casuale a causa, ad esempio, della routine del lavoro, che porta ad usare spesso la stessa funzione o lo stesso cammino. Inoltre, nei sistemi in tempo reale, gli input spesso hanno una correlazione non nulla.

Littlewood e Strigini [91] sostengono che i modelli di crescita della reliability siano la tecnica più soddisfacente per la valutazione dell'affidabilità e della reliability del software. Inoltre affermano che, fra questi, il modello più maturo sia quello bayesiano di Littlewood e Verrall.

Purtroppo accanto a questo giudizio estremamente positivo ve ne sono molti pesantemente negativi. Butler e Finelli [32] mostrano con alcuni esempi come questi modelli non risolvano affatto il problema del testing del software, neppure facendo l'ipotesi (indimostrata e indimostrabile) che il modello si possa applicare anche nella regione dell'ultrareliability (cioè per livello d'integrità software 4). Anche Milner [95] giunge alla conclusione che questi modelli non possono essere applicati alla dimostrazione degli alti livelli di reliability necessari per il software safety critical. Gray [57] mostra come alcuni aspetti di difficile quantificazione, che vengono trascurati nei modelli di crescita della reliability del software, non possono, in realtà, essere ignorati quando vogliamo quantificare livelli di reliability molto elevati; perciò questi modelli sono inadeguati per il software safety-critical. Infine Hamlet e Taylor [62] trovano ancora ulteriori complicazioni e difetti, e Keiller e Miller [79] sostengono che ci sono almeno due ragioni per le quali i modelli di crescita della reliability non possono essere applicati al software safety-critical, il quale necessita di un livello di reliability estremamente alto. Il tempo richiesto perché la reliability aumenti fino ad un livello accettabile tenderà ad essere estremamente alto. Inoltre la reliability necessaria per un livello d'integrità software 4 non può essere garantita statisticamente a priori.

A causa di queste e di molte altre critiche al testing, che sono state registrate nel capitolo 2, alcuni autori, anche recentemente, ad esempio in seno alla "comunità dei sostenitori del metodo B", hanno sostenuto che, per quanto riguarda il software, bisognerebbe eliminare del tutto o quasi questa tecnica "classica", sostituendola soprattutto con prove formali. Per quanto riguarda il metodo B in particolare, una risposta a coloro che pensavano di eliminare il testing da questo metodo di sviluppo ed una riaffermazione della sua necessità viene fornita in [24]. *In realtà il testing ha ancora un suo ruolo, sia pure accanto ad altre tecniche*, indipendentemente dal metodo con cui si sviluppa il software, poiché *soltanto esso può rilevare le eventuali discrepanze tra il modello usato e la situazione reale*, può verificare la validità dell'interfaccia del software con l'hardware e con l'utente, cosa che i metodi formali non sono in grado di fare, almeno al momento attuale, e che non è di secondaria importanza vista la rilevanza dell'errore umano negli incidenti rilevata da vari autori [112], ed è almeno in parte necessario per *verificare la conformità del software con la specifica dei requisiti*. Inoltre il testing si rivela più efficace di qualsiasi altro metodo su quelle parti del software dove è possibile effettuarlo in maniera esaustiva e non è improbabile trovare qualche sottodominio dello spazio di input dove ciò è possibile [10], quando si usa un modello per la valutazione della reliability basato sul dominio di input.

Certamente, però, come puntualizzato in [99], per verificare il software bisognerà usare anche *l'hybrid white box analysis* e non solo il *black box testing*, poichè, a causa del numero di input e di cammini possibili per programmi non banali, usare solo la seconda alternativa sarebbe come cercare il classico ago in un pagliaio.

Il testing e i metodi formali si complementano e si supportano a vicenda. Con l'ausilio della specifica formale del software si può aumentare molto l'efficacia del testing, grazie alla sua combinazione con l'analisi di testabilità, come mostrato da Voas e Miller [123]. Inoltre *si può arrivare alla completa automatizzazione dell'attività di testing, rendendola indipendente dagli errori umani del verificatore e velocizzandola, grazie alla possibilità di ricavare un oracolo automatico dalla specifica formale,* come fatto da Parnas e Peters in [106], o in maniera ancora migliore (specie se abbiamo a che fare con sistemi in tempo reale) è possibile generare test cases (che includono sia gli input che gli output del sistema) direttamente dalla specifica formale TRIO [130].

Il messaggio più importante di Voas e Miller è, per loro stessa ammissione, che, se si vuole veramente migliorare la reliability, bisogna concentrare i propri sforzi sul rendere la specifica dei requisiti più chiara e più completa, specificando meglio ciò che deve accadere anche a livello di computazioni intermedie. *Uno dei metodi (se non l'unico valido) per fare ciò, è certamente l'adozione di un linguaggio di specifica formale;*

infatti, così, è possibile individuare prontamente e senza possibilità di errori o fraintendimenti il dominio e il codominio di funzioni software specificate formalmente e, dal rapporto fra le cardinalità di questi, ricavare il *DRR (Domain Range Ratio).*

Il *DRR (= (cardinalità dominio) / (cardinalità codominio))* è un parametro di grande importanza per la valutazione della testabilità delle funzioni contenute nei moduli, e da questa della testabilità dei moduli stessi e dell'intero programma, poiché Voas e Miller sostengono di aver dedotto dai numerosi studi effettuati una regola empirica in base alla quale *più grande è il DRR di una funzione o di un modulo e maggiore sarà la quantità di testing necessaria per evitare il rischio di errori non rilevati* [123].

In presenza di specifica formale il DRR è disponibile in maniera certa, in quanto questo tipo di specifica delle funzioni non può dare adito a fraintendimenti per sua stessa natura, e in una fase molto precoce. *Ciò permette di sapere, prima di avere iniziato progetto e codifica quali sono le parti del software, che potranno meglio essere verificate col testing e quali quelle in cui sarà meglio usare soprattutto altre tecniche.* In verità la testabilità è inversamente proporzionale oltre che al DRR, che è correlato alla perdita di informazione implicita che si verifica durante l'esecuzione di un programma, anche alla perdita di informazione esplicita, ma quest'ultima potrà essere evitata con semplici accorgimenti di progetto e implementazione.

Friedman, Voas e Miller [50, 123] propongono l'uso *dell'analisi di sensitività automatizzata in combinazione col testing* per poter quantificare livelli notevoli di reliability. L'analisi di sensitività è una nuova tecnica empirica che cerca di quantificare la possibilità che dei difetti rimangano nascosti e le dimensioni minime dei difetti. Si esegue ripetutamente il programma originale e le mutazioni del suo codice sorgente e degli stati dei dati con l'ausilio di un apposito strumento, PiSCES Software Analysis Toolkit. Purtroppo questa tecnica si basa su due ipotesi indimostrabili, abbastanza poco verosimili e spesso contraddette dalle osservazioni empiriche. Le due ipotesi sono:

1. il difetto esiste solo in una singola locazione, non distribuito in tutto il programma (ipotesi di difetto semplice);
2. il programma contiene un singolo difetto/errore, non tanti difetti distribuiti (ipotesi di difetto singolo).

Ora, se, come rilevato da molti autori, "ci sono molte prove che programmi scritti da team indipendenti per un certo compito presentano errori strettamente correlati" [107], risulta difficilmente credibile che il medesimo team non commetta lo stesso errore in più locazioni in software non banale; da ciò è evidente che l'ipotesi di difetto semplice è una forzatura**.** L'ipotesi di difetto singolo, poi, è ampiamente contraddetta dall'osservazione empirica di qualsiasi programma (anche piuttosto banale). Il favore con cui questa tecnica è stata accolta da alcune frange del mondo industriale, è dovuto principalmente alle *suggestive considerazioni di convenienza economica* fatte proprio dai suoi autori. L'analisi di sensitività ha la possibilità di essere totalmente automatizzata, quindi è molto economica rispetto al testing che, invece, ha (o meglio aveva) la necessità di avere, comunque, un oracolo umano, che è molto costoso e il cui costo è tendenzialmente in aumento.

Oggi, se abbiamo a disposizione una specifica formale del software scritta in un linguaggio basato sulla logica del primo ordine, è possibile automatizzare *completamente il testing, ricavando dalla specifica formale un oracolo automatizzato o, se si ha a disposizione una specifica realizzata in TRIO è possibile ricavare da essa in modo semiautomatico dei test cases.*

Parnas e Peters [106] mostrano come sia possibile generare, a partire dalla specifica formale, delle espressioni tabulari e da queste un oracolo per i test completamente automatizzato, aumentando quindi molto la velocità e l'accuratezza della valutazione dei test. E' già stato realizzato un prototipo di strumento, TOG, che genera un oracolo sotto forma di procedure, scritte in linguaggio C. L'uso del linguaggio C è dovuto al fatto che gli esempi, selezionati per illustrare questo metodo, erano realizzati in tale linguaggio, ma si potrebbe utilizzare qualsiasi altro linguaggio.

Con questa tecnica è possibile migliorare notevolmente la qualità, poiché non ci saranno più errori da parte dell'oracolo, la quantità e la copertura del testing, vista la notevole velocità, peraltro ulteriormente incrementabile, dell'oracolo generato e il suo costo basso e tendenzialmente in decremento rispetto a quello di un oracolo umano.

Le limitazioni del metodo sono poche: esso, ad esempio, è difficilmente applicabile a programmi che usano allocazione dinamica della memoria, ma come si è visto nel capitolo precedente l'allocazione dinamica della memoria andrebbe comunque evitata nei sistemi safety-critical, perciò questa limitazione è, in realtà, solo apparente. Inoltre il metodo è stato applicato anche a sistemi con questa caratteristica e si è rivelato efficace anche se l'oracolo si è dimostrato più lento.

Di contro questa tecnica è applicabile sia al testing in grande che a quello in piccolo; infatti essa è applicabile alla verifica della consistenza fra specifica e codice, ai test dei moduli, alla realizzazione di tipi di dati astratti autoverificanti, al cosiddetto testing in situ, che consiste in chiamate ai programmi oracolo, da parte di componenti particolarmente critici, durante l'attività operativa del software safety-critical, rilevando e segnalando prontamente eventuali fallimenti di questi componenti. Naturalmente, in applicazioni come il testing in situ, la performance del programma oracolo diventa vitale, perciò aumenta d'importanza (ma forse sarebbe meglio dire diventa inderogabile) l'evitare l'uso di strutture, come l'allocazione dinamica della memoria, che deteriorano la performance dell'oracolo.

Questa tecnica, come tutto il testing, non può far fronte ad errori di specifica, che perciò andranno evitati con un'analisi degli azzardi di ottima qualità e con l'uso di tecniche di analisi statica come la software fault tree analysis, che riescono a rilevare anche alcuni errori di specifica.

L'analisi della testabilità e la generazione di programmi oracolo sono due tecniche con le quali la specifica formale può aumentare direttamente l'efficacia del testing, ma esistono anche metodi, per così dire, mediati per raggiungere il medesimo scopo.

Come si è visto nel paragrafo precedente, la specifica formale incrementa molto l'efficacia delle tecniche di analisi statica, queste, a loro volta, possono, così, meglio individuare cammini particolarmente critici all'interno del programma. Tali cammini (e soprattutto gli eventuali input che li innescano), poi, possono essere testati in maniera intensa utilizzando un modello per la valutazione della reliability come quello di Bastani e Ramamoorthy, che permette di servirsi di qualsiasi strategia per la selezione dei test case e, quindi, di concentrare il testing dove si vuole e di mettere alla prova i costrutti più inclini agli errori.

5.2.4 La verifica formale

La verifica formale, svolta dopo la codifica, è certamente, come insegna Kemmerer [80], più impegnativa in termini di sforzo, di preparazione e di esperienza richiesti e più onerosa sia in termini di costo economico vero e proprio che di tempo necessario rispetto allo sviluppo per raffinamenti successivi della specifica formale; comunque va anch'essa esaminata, in quanto l'ampia libertà d'azione, concessa da norme come EN 50128, potrebbe permettere tranquillamente di verificare formalmente un software safety-critical, che è stato specificato e sviluppato in modo. Inoltre la cosiddetta “after the fact verification” è ancora in uso sia pure con portata ridotta, rispetto a quella immaginata in [49]. Floyd e Hoare [49, 70]) volevano verificare la correttezza (o la scorrettezza) di interi programmi mediante le prove formali, invece, oggi, si tende a compiere prove di safety, che sono molto più limitate e trattabili, per verificare se il programma (o più verosimilmente la parte di esso) sotto esame può contribuire ad un azzardo per la safety.

In pratica è caduta anche l'unica sottile differenza che Leveson (v. anche [88]) rivendicava esistere fra il tipo di ragionamento della software fault tree analysis e quello delle prove formali assiomatiche, in quanto entrambe le tecniche hanno recepito che bisogna verificare la safety (che è la qualità più importante) del programma e non la sua correttezza, che, a volte, può sussistere anche in assenza di safety.

Una prova formale di safety parte da una specifica che è certamente sicura (safe). Come sottolinea Kemmerer in [80], una specifica formale o prima o dopo la codifica dovrà comunque essere realizzata, se si vogliono usare tecniche di verifica formale. Hoare, in [70], ha elaborato un metodo di verifica formale, basata sulla logica del primo ordine, che è alla base del ragionamento assiomatico-deduttivo di molti linguaggi di specifica formale, anche moderni, quali B. La stessa “logica di Hoare”, sia pure nelle sue numerose varianti, viene ancora utilizzata per la verifica di parti (safety-critical) di programma. La notazione introdotta da Hoare consiste in espressioni del tipo PRE(q)POST, che significa che, se il frammento di programma q, quando inizia la sua esecuzione, si trova in uno stato che soddisfa la precondizione PRE, allora quando q termina la sua esecuzione si troverà in uno stato che soddisfa la postcondizione POST.

Ogni costrutto di un linguaggio di programmazione ha una sua regola di prova. Dijkstra, in [43], ha elaborato il “calcolo dei programmi” un metodo strettamente imparentato con la logica di Hoare, il quale si basa sul concetto di “*weakest precondition*”. Questo concetto è di grande importanza, non solo per la sua applicazione nel metodo inventato da Dijkstra, ma anche perché è alla base di altre tecniche di analisi interessanti ed efficaci, come ad esempio la software fault tree analysis. La weakest precondition, rispetto a un frammento di programma p e a una postcondizione POST, è l'insieme di tutti gli stati che, quando viene eseguito p, termineranno e lasceranno lo stato della computazione in POST.

Provare formalmente la safety (o la correttezza delle parti che riguardano la safety) di un programma è un processo che, specie quando si tenta la verifica di un codice non specificato formalmente, può essere estremamente lungo e oneroso, perciò è necessario l'uso di strumenti di supporto come ad esempio i vari *prover* in commercio. Tra questi i più interessanti sono il già menzionato PVS, il Boyer-Moore theorem prover, che ha dato ottime prove di sé [27], e il SDVS (State Delta Verification System), sviluppato dall'Aerospace Corporation, che, soprattutto negli USA, è stato usato per la prova formale di molti programmi con quantità di software safety-critical dell'ordine di circa 1000 linee di codice sorgente. Questo strumento è particolarmente interessante per due motivi:

1. supporta i verificatori (umani) nel realizzare prove formali di correttezza, safety e sicurezza a livello sia di hardware sia di software;

2. si basa sulla logica temporale quindi, come essa, è in grado di trattare sistemi in tempo reale, sequenziali, concorrenti e distribuiti.

Naturalmente SDVS, come PVS, può essere usato anche per la specifica dei requisiti del software grazie al linguaggio "State Delta", che è una logica temporale. Il linguaggio prende il nome dalle sue formule di base che si chiamano appunto *state delta*. Una formula state delta, tramite costrutti tipici della logica temporale, rappresenta espressioni del tipo "se P è vero ora, allora Q sarà vero in futuro".

Un punto debole di *SDVS, come di molti altri strumenti, è che ha bisogno di assistenza da parte dell'operatore umano*, il quale deve svolgere un ragionamento induttivo, quando si esaminano cicli nei quali il numero di iterazioni dipende dai dati o è molto grande.

Leveson [88] sostiene che il software safety-critical deve essere progettato e sviluppato per essere verificabile. La verifica formale, come sottolinea Kemmerer in [80], è certamente più conveniente sia dal punto di vista temporale che da quello economico, realizzando un progetto ed uno sviluppo per raffinamenti successivi. Inoltre *una specifica formale dei requisiti è utile, come già detto, anche per migliorare l'efficacia, la velocità e la copertura del testing, nonchè per rendere più incisiva l'analisi statica.* I metodi formali più moderni come B e TRIO optano per la procedura per raffinamenti successivi realizzando software "corretto per costruzione". Anche volendo realizzare una verifica indipendente (cosa più che auspicabile) è meglio, che *verificatori indipendenti (che conoscono il metodo formale usato nel progetto) controllino la validità delle prove formali eseguite dagli sviluppatori,* come dimostrato dalle esperienze di realizzazione di software safety-critical ferroviario francesi [51]. Se la specifica è informale i verificatori indipendenti, nel tentare una verifica formale a partire dal codice con un'attività di reverse engineering, avranno possibilità di fraintendimento della specifica informale amplificate rispetto a quelle del personale che ha realizzato il sistema e avranno, comunque, l'onere di dover effettuare una sorta di specifica posteriore del programma con oneri e rischi amplificati.

Al contrario *la specifica formale dei requisiti è un'attività indispensabile, se si vuole realizzare una verifica veramente indipendente ed efficace*, come richiesto anche dalla norma EN 50128 per i livelli di integrità 3 e 4. Una specifica informale è, per sua stessa natura, ambigua e spesso incompleta e poco chiara.

Pertanto, *in caso di specifica informale, i verificatori esterni avranno certamente bisogno dell'aiuto dei progettisti per capire il sistema in esame, perdendo parte della loro obiettività, se faranno troppo affidamento sulla loro "guida", rilevando* *difficilmente i fraintendimenti della specifica fatti proprio da chi gli illustra il sistema e rischiando di sovrapporre i loro fraintendimenti a quelli dei progettisti con effetti disastrosi per la safety.*

5.3 Integrazione software/hardware

L'integrazione software/hardware è trattata nel capitolo 12 della norma europea EN 50128. Gli obiettivi di questo capitolo sono dimostrare che software e hardware interagiscono correttamente nello svolgere le funzioni loro assegnate e assicurarsi che la combinazione di software e hardware ottenuta con il progetto sia compatibile con la specifica dei requisiti di sistema relativi alla safety e con i requisiti derivanti dal livello d'integrità software.

In input ci saranno tutti i documenti prodotti nelle fasi precedenti e la documentazione sull'hardware. I documenti prodotti in output saranno il piano di integrazione software/hardware e il rapporto di integrazione software/hardware. Naturalmente il piano di integrazione software/hardware verrà soltanto completato in questa fase, mentre la sua stesura dovrà iniziare molto prima, nelle fasi iniziali del ciclo di sviluppo.

Il *piano di integrazione* software/hardware deve documentare la divisione del sistema in livelli di integrazione, i casi di test e i dati di test, i tipi di test che devono essere svolti, gli strumenti e il software di supporto ai test, e i criteri in base ai quali giudicare i risultati dei test.

Il *rapporto di integrazione* software/hardware conterrà i risultati dei test e illustrerà se gli obiettivi e i criteri del piano di integrazione software/hardware sono stati raggiunti.

In base alle tabelle normative allegate, le tecniche di verifica da adottare saranno: per i livelli di integrità software 1, 2, 3, 4, testing funzionale a scatola nera e testing di performance, mentre per il livello d'integrità 0 (assenza di implicazioni con la safety), sarà sufficiente il solo testing funzionale. Per quanto riguarda il testing funzionale in particolare, sono "altamente raccomandati", per tutti livelli di integrità software, l'analisi dei valori al limite e il testing basato sulla suddivisione del dominio di input, mentre sono soltanto "raccomandate" la simulazione di processo e, per i livelli di integrità 3 e 4, la prototipazione e l'esecuzione di test case tratti dai *diagrammi di cause e conseguenze* (combinazione di ETA e FTA, v. cap. 4.2.2). Sebbene il testing sia già stato trattato nel capitolo precedente, è utile ribadire e puntualizzare alcuni concetti. Queste tecniche traggono tutte grandi vantaggi non solo dalla specifica formale, ma anche dall'analisi statica, pertanto è sbagliato imporre, come fa la normativa in oggetto, di svolgere il testing funzionale a scatola nera, basandosi solo sulla specifica, specie se essa è informale.

Il testing basato sulla suddivisione del dominio di input trova grande supporto nella specifica formale, se si decide di definire le classi di equivalenza in cui suddividere il dominio di input in base alla specifica, in quanto non ci potranno essere possibilità di fraintendimento del reale dominio di input (e dei sottodomini ad esso associati). Se invece si sceglie la strategia alternativa di definire le classi di equivalenza in base alla struttura interna del programma, si otterrà *un ottimo (e indispensabile) supporto dall'analisi statica, la quale, come già detto, aumenta la sua efficacia in presenza di* *specifica formale, venendosi così a creare una sorta di circolo virtuoso con il risultato di una verifica più efficace.* La prima strategia, seppure in misura minore rispetto alla seconda, trova anch'essa un ottimo supporto nell'analisi statica come illustrato nel capitolo precedente.

L'analisi dei valori al limite, essendo basata su una divisione del dominio di input in classi e sull'esecuzione del software (in questa fase già implementato sull'hardware obiettivo) con i valori limite delle classi in ingresso, *trae, come la tecnica precedente, grandi benefici dalla specifica formale dei requisiti, sia software sia (e soprattutto) di sistema*. Ovviamente questo metodo ottiene un notevole ausilio dalla sua omonima tecnica statica, la quale individua proprio i valori limite che costituiscono l'input di questo metodo di testing.

La simulazione e soprattutto la prototipazione, di cui si è già parlato nel capitolo sulla specifica dei requisiti, rivestono un'importanza ben maggiore di quella riconosciuta ad esse dallo standard nella verifica dell'integrazione software/hardware di sistemi ferroviari, in quanto:

- *il testing* ha i molti limiti illustrati nei capitoli precedenti e per di più in questa fase, trattandosi di testing sul campo, è costosissimo;
- *la simulazione* ha dato ottimi risultati in molti casi come il software delle metropolitane di Parigi e di Calcutta, delle nuove linee ad alta velocità francesi e del progetto Meteor, un sistema ferroviario metropolitano ad automazione integrale;
- *la prototipazione*, poichè, a differenza della simulazione, rispetta anche i requisiti temporali, è l'ideale per i sistemi in tempo reale, in quanto è in grado di animare le specifiche formali (permettendo così di individuare eventuali errori) ed ha un costo infinitamente inferiore al testing sul campo.

Per quanto concerne l'esecuzione di test case ricavati dall'esame del *diagramma delle cause e delle conseguenze*, notiamo che *elaborare questo diagramma a livello di sistema completo è piuttosto arduo*, visto che esso assomma le limitazioni e i difetti della fault tree analysis e quelli della event tree analysis amplificandone alcuni, come i problemi di ingestibilità dovuti all’eccessiva crescita e ramificazione degli alberi. Pertanto non è chiaro come i verificatori possano svolgere questo compito conoscendo solo la specifica del sistema da costruire, come imposto dallo standard. Se, poi, i verificatori esterni usassero il diagramma fornito dai progettisti perderebbero parte della loro indipendenza e, comunque, non si potrebbe più parlare di testing a scatola nera, poichè un simile diagramma permette di conoscere moltissimo della struttura interna del sistema. Se i test case fossero forniti dai progettisti, i verificatori esterni perderebbero tutta la loro indipendenza e il testing diventerebbe inutile. Se infine i test case fossero elaborati da altri verificatori indipendenti e poi forniti a quelli che svolgono i test non si potrebbe, più comunque parlare di black box testing in quanto i casi di test sono stati ricavati almeno in parte dalla struttura del codice, a causa dell'applicazione anche al software della fault tree analysis.

L'inclusione dell'esecuzione di test case ricavati dai diagrammi cause conseguenze sembra un'ammissione implicita del fatto che, per la verifica del software e della sua integrazione con l’hardware, è necessario anche il white box testing.

Per quanto riguarda il *testing di performance*, sono “altamente raccomandati”, per tutti i livelli di integrità, l'analisi dei requisiti di performance e delle necessità di risposta temporale e di memoria. Inoltre lo stress testing è “raccomandato” per i livelli di integrità 1 e 2, e “altamente raccomandato” per i livelli 3 e 4.

Lo *stress testing* ha lo scopo di testare il dispositivo in esame applicando ad esso condizioni di carico di lavoro eccessivamente elevate, allo scopo di mostrare che il dispositivo può sopportare senza problemi dei sovraccarichi. Questa tecnica gioca un ruolo fondamentale nella valutazione della reliability dei componenti hardware. Può servire per valutare il comportamento del software a fronte di guasti hardware dovuti a sovraccarico, in quanto, come sostiene Leveson [88], i fallimenti software si verificano più frequentemente in presenza di guasti hardware. Barrett e altri [19] rilevano che bisognerebbe sviluppare sistemi software per applicazioni ferroviarie in grado di assistere l'operatore anche in condizioni al di fuori del normale, come sono certamente quelle che si verificano in presenza di guasti hardware, ma *lo stress testing, così come l'uso di reliability blocks o di modelli di Markov, è utilissimo a livello di verifica e di convalida dei dispositivi hardware, ma molto meno a livello di verifica dell'integrazione tra software e hardware.*

La reliability dei componenti hardware degrada con l'usura, perciò è probabile che, quando si effettua lo stress testing (su dispositivi nuovi), non si verifichino guasti ed il software non sia sottoposto a condizioni al di fuori del normale, che possono portare a comportamenti anomali a livello di sistema. Sarebbe molto più utile e meno costoso simulare condizioni di guasto hardware e verificare il comportamento del software e del sistema in condizioni anomale di funzionamento.

L'analisi dei requisiti di performance serve a stabilire se i requisiti di performance a livello di sistema integrato sono stati raggiunti. Questa tecnica di analisi, naturalmente, parte da un attento esame della specifica dei requisiti del software e del sistema sul quale esso è implementato, allo scopo di identificare tutti i requisiti che possono influenzare la performance del sistema. Poi si analizza ogni requisito di performance per individuare criteri di successo (raggiungimento del requisito), misure indicative del successo raggiunto, accuratezza di tali misure, stadi dello sviluppo in cui si possono svolgere queste misure, e raggiungibilità del requisito.

L'analisi delle *necessità di risposta temporale e di memoria* ha bisogno di stime dell'uso delle risorse e del tempo necessario allo svolgimento di ogni funzione di sistema. Tali stime possono essere ricavate o per raffronto con sistemi simili già esistenti o per prototipazione almeno dei sottosistemi con requisiti temporali critici. Il *confronto con sistemi simili*, come ci insegna l'esperienza fallimentare di Ariane 5 [4], *può essere adottato solo previa verifica della corrispondenza fra le specifiche formali del sistema già in funzione e di quello in esame. La prototipazione dei sistemi safety-critical in tempo reale è sempre necessaria, in quanto è il metodo più sicuro per verificare che il sistema raggiunga i requisiti temporali imposti dalla specifica; infatti la verifica del soddisfacimento dei requisiti temporali tramite testing ha pesanti limitazioni* già evidenziate nel capitolo 2, come l'inadeguatezza degli strumenti, attualmente in uso per il testing, per l'analisi delle proprietà temporali dei programmi, l'impossibilità per motivi economici, oltre che temporali, di svolgere una quantità sufficiente di testing sul campo sul sistema realizzato, vista l'enorme esosità di tali prove etc.

Le analisi sui requisiti imposte dalle ultime due tecniche esaminate, ovviamente, dovrebbero essere già state svolte anche dai progettisti al momento dell'elaborazione della specifica dei requisiti software, ma è certamente utile che vengano effettuate nuovamente da personale esterno (come imposto per i livelli di integrità software 3 e 4). *Comunque è necessario ribadire che tali analisi potranno essere svolte in modo rigoroso soltanto se il software e il sistema sono specificati formalmente. In caso contrario esse diventeranno semplici giudizi soggettivi espressi da esperti, senza la possibilità di effettuare delle vere e proprie prove di consistenza dei requisiti software con quelli di sistema e di completezza dei requisiti dei programmi rispetto a quelli assegnati al software dalla specifica dei requisiti funzionali e di safety a livello di sistema.*

5.4 Convalida del software

La convalida del software viene trattata nel capitolo 13 della norma europea EN 50128. *L'obiettivo del processo di convalida è dimostrare che il sistema è accettabile sia sotto l'aspetto della funzionalità sia sotto quello della safety.*

In input ci saranno la specifica dei requisiti di sistema e tutta la documentazione prodotta sull'hardware e sul software. I documenti prodotti in output saranno piano di convalida del software e rapporto sulla convalida del software. Il mezzo principale per svolgere la convalida dev'essere, secondo lo standard, il testing, mentre simulazione e uso di modelli possono essere usati, discrezionalmente, come supplemento al processo di convalida.

In questa fase entra in scena anche l'*assessor*, che è la persona o l'organizzazione indipendente cui è *affidata la verifica della validità dei requisiti di safety e la certificazione del loro raggiungimento.*

Il piano di convalida del software sarà sviluppato e valutato da un organismo indipendente, il quale dovrà accordarsi con l'assessor sulla portata e i contenuti di tale piano e sulla presenza dell'assessor durante il testing, per i livelli di integrità software 3 e 4. Il piano di convalida del software dovrà identificare i test case per ogni funzione affidata al software, e la giustificazione della strategia di convalida scelta.

I risultati dell'attività di convalida saranno registrati nel rapporto sulla convalida del software, che dichiarerà se il software ha superato l'esame di convalida o i motivi della sua bocciatura.

La tabella normativa allegata stabilisce che, per tutti i livelli di integrità software, una combinazione di tecniche approvata sarà: testing di performance e testing funzionale a scatola nera.

Esaminando il capitolo 13 e soprattutto i metodi suggeriti, sembra evidente che la norma europea dà in realtà *al termine convalida un significato ben più ristretto* rispetto a quello attribuito ad esso dalle fonti più autorevoli, riducendo l'attività di convalida ad una sorta di ulteriore verifica del raggiungimento dei requisiti imposti dalla specifica, in quanto è universalmente accettato che con il testing a scatola nera ed il testing sulla performance non si può ottenere nessun altro risultato.

“La convalida è l'attività volta ad esaminare se si sia costruito il sistema che risolve il problema applicativo che ha motivato la realizzazione” [53]. Il medesimo concetto, sia pur con definizioni più verbose, viene espresso dalla guida per gli ingegneri che progettano sistemi safety-critical elaborata dall’Hazards Forum [65], da Storey [118], da Leveson [88]. Bologna e altri [10] sostengono, in sostanziale accordo con Littlewood e Strigini [91], che i metodi da adottare per la convalida del software sono prove formali, testing, simulazione simbolica e le varie tecniche di analisi statica. Quindi è evidente che il processo di convalida è molto ampio e coinvolge le attività illustrate nei capitoli 13 e 14 rispettivamente sulla convalida e sulla valutazione del software, e molte altre ancora quali quelle di ulteriori analisi dei requisiti e degli azzardi.

D'altra parte *la verifica e la convalida svolte da organizzazioni pienamente indipendenti rappresentano le attività in base alle quali viene rilasciata (o meno) la certificazione del software*. Certamente è molto sentita la *necessità di agenzie veramente indipendenti che svolgano quest'attività*, come richiesto pressantemente da Parnas e altri e da Leveson [107, 88].

In particolare, Leveson sostiene che l'organizzazione che si occupa della verifica e della convalida di software safety-critical dev'essere indipendente da chi ha sviluppato il sistema sotto il profilo finanziario, tecnico e manageriale. Parnas e altri dicono a chiare lettere che è difficile verificare un proprio progetto in modo obiettivo e suggeriscono che la verifica e la convalida siano affidate ad un organismo interamente indipendente dall'azienda che ha sviluppato il sistema, paradossalmente suggeriscono un'azienda concorrente che tenterà di dimostrare col massimo impegno che il software non è valido, svolgendo così un attività meticolosa.

Dare alla modellazione e simulazione semplicemente un ruolo di eventuale supporto alla attività di convalida è in netto contrasto con le opinioni di diversi autori quali Storey, Bologna e altri [118, 10], e con la prassi dei più recenti sviluppi di software safety-critical in ambito ferroviario quali il software delle metropolitane di Parigi e Calcutta, delle nuove linee delle ferrovie francesi ad alta velocità del recentissimo progetto Meteor già menzionati precedentemente.

Gerhart e altri [51] riferiscono che, nel software della metropolitana parigina, la simulazione e la prototipazione hanno giocato un ruolo importante in combinazione con i metodi formali ed il testing. Oltre 600 scenari operativi sono stati elaborati e verificati per mezzo di automi a stati finiti. Inoltre la verifica e la convalida sono state considerate come parte dell'attività di certificazione, separando e scaricando così i loro costi da quelli di progetto e sviluppo e permettendo di rilevare (e correggere) gli errori in una fase precoce senza che fosse compromessa l'indipendenza del comitato incaricato della certificazione. *I metodi formali (soprattutto il metodo B) hanno avuto il ruolo più importante nella certificazione,* grazie anche al fatto che il comitato incaricato della certificazione, formato da 10 esperti fra membri delle ferrovie francesi, della società per l'alta velocità, dell'authority che controlla l'aviazione in Francia, del mondo accademico ed esperti di safety, ha accettato di apprendere il metodo formale utilizzato, cioè il metodo B, ed ha così potuto controllare pienamente tutte le fasi dello sviluppo formale.

5.5 Valutazione del software

La valutazione del software è trattata nel capitolo 14 della norma europea EN 50128. L'obiettivo di questa attività è *valutare che i processi e i risultati di ogni fase del ciclo di vita siano tali da permettere al software di raggiungere il livello d'integrità stabilito e di essere in grado di svolgere i compiti necessari per l'applicazione.*

In input ci saranno la specifica dei requisiti di safety di sistema, il piano di convalida del software e tutti i documenti inerenti l'hardware e il software.

L'output prodotto sarà il rapporto sulla valutazione del software.

L'attività di questa fase verrà svolta da un assessor, che, per i livelli di integrità software 3 e 4, dovrà essere indipendente dal team di progetto.

L'assessor deve valutare se il software sia in grado di svolgere i suoi compiti, se sia conforme ai requisiti di safety, se sia del livello d'integrità software richiesto, e se siano stati scelti e applicati metodi appropriati in ogni fase del ciclo di vita del software. Alla fine della valutazione l'assessor dovrà produrre il rapporto di valutazione del software che conterrà la dichiarazione del livello d'integrità ottenuto dal software in esame. In caso di valutazione negativa l'assessor compilerà comunque un rapporto di valutazione del software, contenente la sua opinione sulla causa della bocciatura e le azioni correttive necessarie. Durante il processo di valutazione dovranno essere considerate tutte le fasi del ciclo di vita del software a partire dall'assegnamento dei livelli di integrità software fino alla manutenzione. Le tecniche utilizzabili, come al solito, variano in base al livello di integrità software:

- per il livello 0 (assenza di implicazioni con la safety), sono “raccomandate” le checklists (v. 4.2.2), l'analisi statica del software (v. 4.6.2), i diagrammi cause conseguenze, la fault tree analysis (v. 4.2.2) e il collaudo sul campo;
- per i livelli 1 e 2 sono “raccomandate” le checklists, il testing (v. 4.6.3), i diagrammi cause conseguenze (v. 4.2.2), l'event tree analysis (v. 4.2.2), la fault tree analysis, la software errors effect analysis (più conosciuta con il nome di software failure modes and effects analysis, v. 4.6.2), la common cause failure analysis, i modelli di Markov, i reliability block, mentre sono “altamente raccomandate” l'analisi statica del software e il collaudo sul campo;
- per i livelli 3 e 4 sono “raccomandate” le checklists, l'event tree analysis, i diagrammi cause conseguenze, i modelli di Markov, e i reliability blocks, mentre sono “altamente raccomandate” l'analisi statica del software, il testing, la fault tree analysis, la software errors effect analysis, la common cause failure analysis, e il collaudo sul campo.

Tutte le summenzionate tecniche sono state ampiamente analizzate nei capitoli precedenti tranne i modelli di Markov, i reliability block e l'analisi dei fallimenti di causa comune.

I *modelli di Markov* vengono applicati soprattutto all'hardware di sistemi safety-critical, il cui comportamento varia col passare del tempo, cioè nel sistema in esame ci sono delle transizioni fra un insieme di stati. Questo metodo è *molto simile agli automi a* *stati finiti*, con l'aggiunta di informazione quantitativa riguardante la *probabilità di transizione tra gli stati*. Gli stati futuri si considerano indipendenti da tutti gli stati passati eccetto quello immediatamente precedente lo stato in considerazione. Si assume che il comportamento del sistema sia stazionario, cioè la probabilità che vi sia una transizione da uno stato ad un altro è indipendente dal tempo.

Da un punto di vista grafico, ogni *stato* è rappresentato da un'ellisse, con al centro una descrizione sintetica, sulla sinistra un numero identificativo, e sulla destra una lettera (S o F), che indica se lo stato è uno stato di successo o di fallimento. Ogni *transizione* è rappresentata con un arco orientato nella direzione della transizione stessa. Sopra ogni arco si trova un numero (numero di transizioni/h) o un'espressione simbolica, che rappresenta il tasso di transizione. Il *tasso di transizione* può essere il tasso di fallimento software (o hardware), quando si passa da uno stato in cui il software (o l'hardware) opera con successo ad uno in cui fallisce, o il tasso di riparazione hardware (o software), quando si passa da uno stato in cui l'hardware (o il software) sta fallendo ad uno in cui sta operando con successo.

Il tasso di transizione può anche essere rappresentato da un'espressione simbolica, composta con più parametri quali: tasso di fallimento, tasso di riparazione e copertura. La *copertura* è data da:

$C = C_r + C_i + C_R$

C_r = probabilità che una condizione di errore sia rilevata;

C_i = “ “ “ “ “ “ “ isolata;

C_R = “ “ “ “ “ “ “ riparata.

Il tasso di fallimento, il tasso di riparazione e la copertura rappresentano i parametri del modello. Quando si vuole valutare la disponibilità o altre caratteristiche del sistema modellato (p.e. l’affidabilità), si associa ad ogni parametro un valore stimato attraverso uno (o più) dei modelli di valutazione della reliability visti nel capitolo 4.6.3. Poichè vi possono essere delle incertezze sui valori dei tassi di fallimento o riparazione, si può studiare la *sensibilità* del modello al variare dei suoi parametri in un certo intervallo di incertezza sostituendo vari valori ai parametri.

Ci sono molti strumenti (quali SHARPE, SURE, ASSIST, MARKOV etc.), che calcolano la probabilità del sistema di trovarsi in ognuno degli stati esaminati.
La disponibilità del sistema è semplicemente la somma delle probabilità degli stati di successo del modello.

La figura 5.3 mostra un sistema ripristinabile rappresentato con questa notazione.

I modelli di Markov hanno dato buona prova con l'hardware (soprattutto con quello ridondante), ma *la loro applicazione al software presenta non poche difficoltà*, quali:

- la predizione dei parametri, che comporta una quantità irrealizzabile di testing per raggiungere tassi di fallimento del software accettabili;

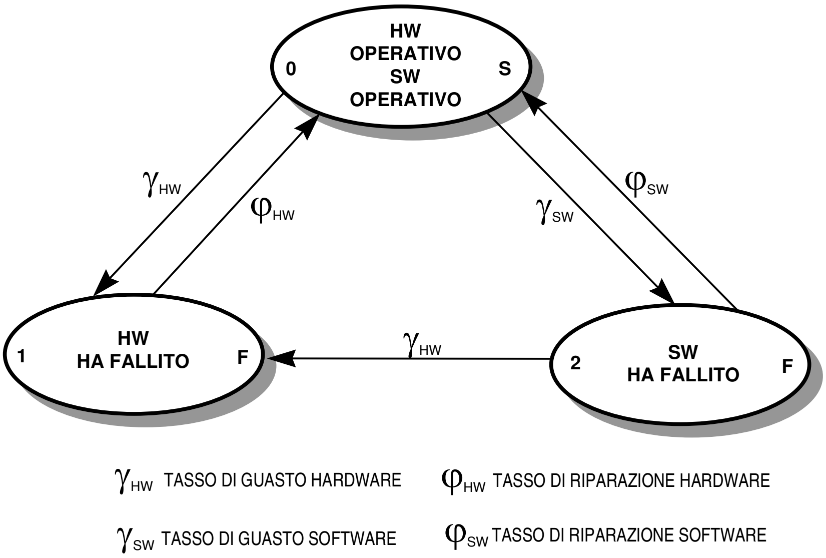

Fig. 5.3 Modello di Markov di sistema riparabile

- la sostanziale inefficacia dei meccanismi di fault tolerance per il software, che provoca l'irrealizzabilità (tranne che per una classe limitata di errori) di programmi che tornano automaticamente da uno stato di fallimento ad uno stato operativo;
- i problemi di esplosione dello spazio degli stati, tipici dei modelli che utilizzano automi a stati finiti.

In ultima analisi, applicando i modelli di Markov al software si combinano i difetti degli automi a stati finiti con quelli dei modelli di valutazione quantitativa della reliability.

I *reliability block* sono modelli che funzionano bene per la quantificazione dei tassi di fallimento e della reliability di sistemi che applicano all'hardware la fault tolerance.

Ad esempio se abbiamo un sistema con 2 componenti ridondanti con tassi di fallimento θ_1 e θ_2 e reliability R_1 e R_2, la reliability totale è data da:

$R = 1 - (1- R_1)(1 - R_2)$

Il tempo medio al fallimento è dato da:

$MTTF = 1/\theta_1 + 1/\theta_2 - 1/(\theta_1 + \theta_2)$

Il tasso di fallimento totale è funzione del tempo. Il diagramma con i reliability block di questo semplice esempio è illustrato nella figura 5.4.

Purtroppo le formule utilizzabili per l'hardware ridondante non si applicano minimamente al software, poiché, mentre se ci sono n componenti ridondanti nell'hardware, essi devono guastarsi tutti contemporaneamente per avere un fallimento di sistema e ciò è piuttosto difficile*, nel software i difetti più pericolosi e numerosi sono comuni a tutte le versioni del programma in esame* poiché sono dovuti ad errori di specifica o comunque al fatto che i programmatori, come notano Parnas e altri [107] tendono a commettere errori strettamente correlati (v. cap. 4.4).

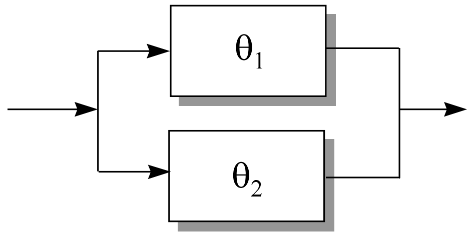

Fig. 5.4 Modello con reliability block

L'analisi dei fallimenti di modo comune è utile per individuare le cause di fallimenti simultanei di sistemi ridondanti, quindi è utile soprattutto dove la ridondanza è efficace cioè nell'hardware. Questa tecnica, come rilevano Bologna e altri [10] consiste di una lista di verifica con le possibili cause di fallimento simultaneo di sistemi ridondanti, per guidare la ricerca di modi comuni di fallimento nei sistemi in esame. L'analisi dei fallimenti di modo comune è di scarsa utilità per il software, vista la sostanziale inutilità della ridondanza e della programmazione multiversione illustrata nel cap. 4.4.

L'assessor viene nominato dall'Autorità che sovrintende alla safety per valutare l'idoneità del sistema al suo scopo e il raggiungimento dell'integrità safety richiesta, giudicando il lavoro dei team che si sono occupati del progetto e della convalida. *L'attività di questo organismo trae grande giovamento da una specifica formale dei requisiti di safety e funzionali di sistema, del software, e dell'architettura, e dal progetto e sviluppo per raffinamenti successivi a partire dalla specifica formale.* Ciò avviene in quanto soltanto se il team avrà a disposizione questi strumenti potrà certificare con sicurezza il rispetto dei requisiti di safety, che rappresentano l'input primario della sua attività, la quale, però, naturalmente, comporta anche la verifica della completezza ed adeguatezza dei requisiti di safety stessi.

6 MODUS OPERANDI OTTIMALE DELL'ORGANISMO DI CERTIFICAZIONE

Riassumendo in maniera sintetica le considerazioni più importanti dei capitoli precedenti si possono individuare le caratteristiche cui dovrebbe rispondere l'organismo di certificazione (e valutazione), un suo possibile metodo di lavoro, e le caratteristiche di progetto, in grado di assicurare il massimo livello di safety ottenibile all'attuale stato dell'arte.

L'organismo di certificazione dovrebbe essere composto da un team di esperti di comprovata esperienza, con un'ottima attitudine al lavoro di gruppo e competenze ben assortite nelle varie fasi dello sviluppo del software, nell'analisi degli azzardi, nei metodi formali e nelle varie tecniche di verifica.

Questo organismo dovrebbe essere totalmente indipendente e la sua attività principale (se non l'unica) dovrebbe essere proprio la certificazione dei sistemi safety-critical e del software in particolare; in pratica dovrebbe essere qualcosa di simile ai TUV tedeschi o almeno alle agenzie di certificazione che stanno nascendo nel Regno Unito come ad esempio ERA, o all'Institute of Railway Signalling Engineers.

Alla luce anche delle esperienze di successo avvenute all'estero, l'organismo di certificazione permanente dovrebbe essere affiancato da un comitato capeggiato dal Safety Assessor indipendente e composto da rappresentanti qualificati degli utenti/clienti del sistema e altre parti interessate (come ad esempio il Ministero dei trasporti per un sistema ferroviario), scelti anche all'esterno dell'azienda/ente cliente, ad esempio nel mondo accademico, allo scopo di dare maggiori garanzie di trasparenza e di corretto e non influenzabile svolgimento di questa delicatissima attività.

Come sottolineato da Storey [118], *la certificazione dei sistemi contenenti software deve comprendere un giudizio su gran parte sul processo di sviluppo* valutando sia i metodi utilizzati che la competenza dello staff di sviluppo. A onor del vero non esiste un metodo di certificazione unanimemente accettato per la preparazione del personale [103], pertanto, vista la buona dose di soggettività dell'argomento, ci si concentrerà principalmente sulle tecniche usate, tenendo anche conto che oggi "spesso sistemi software critici per la safety vengono realizzati da personale che non ha una preparazione che va oltre la scuola secondaria" [107], con effetti a volte disastrosi. Non sarebbe quindi una pretesa priva di fondamento richiedere che chi progetta e sviluppa programmi safety-critical sia in possesso di una laurea e di un'abilitazione adeguate anche se, naturalmente, deve essere affiancato da esperti del campo al quale il software verrà applicato.

A causa del notevole grado di interdisciplinarietà dei sistemi safety-critical, l'ideale sarebbe affidare la realizzazione del loro software di controllo e protezione ad ingegneri con una buona preparazione sia nel campo dell'informatica sia nelle altre discipline ingegneristiche, in modo da potere effettivamente raggiungere una proficua interazione tra esperti di settori diversi, un elemento strettamente necessario soprattutto nelle fasi letteralmente vitali dell'elicitazione e della specifica dei requisiti. Su tale strada sono già stati proposti dei curricula di studi, sia a livello di B.A. che di Master, con un lavoro di collaborazione internazionale tra diverse università [84]

Il processo di certificazione per raggiungere il massimo livello di assicurazione della safety dovrebbe procedere nel seguente modo:

1. *acquisizione di tutta la documentazione relativa al sistema in esame*, evitando il più possibile contatti tra team di certificazione e di sviluppo in modo che non si creino rapporti di influenza o peggio di sudditanza psicologica, che comprometterebbero il buon esito dell'attività di certificazione;

2. *verifica dell'elicitazione dei requisiti*, con controllo delle analisi effettuate dagli sviluppatori e, se ritenuto utile, ulteriori analisi mirate all'esclusione (ad esempio con l'uso dei metodi formali) del verificarsi di un azzardo piuttosto che alla quantificazione di una sua probabilità di occorrenza, sia pure giudicata irrilevante (v. cap. 4.2.1);

3. *verifica dell'idoneità ad evitare azzardi (e incidenti) dei requisiti di safety*. Qualora risulti che uno o più requisiti di safety sia(no) stato(i) trascurato(i), l'attività di certificazione verrà interrotta, mentre i progettisti dovranno modificare l'intero progetto e sviluppo (con costi enormi), evitando, con questo deterrente, che attività fondamentali siano svolte in maniera superficiale. La specifica formale dei requisiti di safety è indispensabile per capire realmente quali requisiti sono stati inclusi e

quali no. Verrebbero così evitati tutti i fraintendimenti e le contestazioni fra certificatori e sviluppatori consentendo all'organismo di certificazione di capire realmente la completezza dei requisiti di safety in una fase preliminare;

4. *verifica della conformità della specifica dei requisiti del software ai requisiti di safety e a quelli funzionali di sistema*. I requisiti del software come quelli di safety e funzionali di sistema dovranno essere inderogabilmente specificati con lo stesso linguaggio di specifica formale, per rendere quest'attività di verifica rapida e senza possibilità di errore(v. cap. 4.3.2). In caso di specifica informale questa verifica diventerebbe molto più lunga e costosa e soprattutto basata sul giudizio e l'esperienza di esseri umani, pesantemente inclini all'errore. Bisognerà evidenziare e registrare i requisiti temporali critici cui deve sottostare il sistema, in vista delle attività supplementari, necessarie per la verifica del raggiungimento di tali requisiti da parte del software;
5. *analisi di testabilità*, per vedere quali funzioni affidate al software sono più inclini a nascondere degli errori durante il testing, prediligendo per tali funzioni altre tecniche di verifica e concentrando su esse maggiori risorse di analisi statica di buon livello (come software fault tree analysis) e prove formali;
6. *verifica della prototipazione della specifica dei requisiti* allo scopo di individuare, prima del progetto e della codifica, eventuali errori di specifica, anche riguardanti l'allocazione dei requisiti temporali, e correggerli prontamente con una spesa centinaia di volte inferiore a quella necessaria per la loro individuazione e correzione dopo la codifica;
7. *verifica della consistenza tra la specifica formale dell'architettura software e quella dei requisiti software*. Avendo redatto la specifica dell'architettura con lo stesso formalismo usato per la specifica dei requisiti, questa analisi sarà facile, immediata e soprattutto sicura;
8. *controllo del progetto e dello sviluppo per raffinamenti successivi* con particolare attenzione alle funzioni con scarsa testabilità e a quelle che implementano requisiti temporali critici. Qualora il processo di sviluppo per raffinamenti successivi sia stato portato termine con strumenti non certificati per un livello d'integrità adeguato(come succede praticamente sempre oggi), verranno eseguite nuovamente "a mano" le prove formali delle parti di codice con scarsa testabilità e di quelle che allocano requisiti temporali critici, o si controlleranno e si verificheranno attentamente le prove formali effettuate dei progettisti. Qualora i progettisti abbiano optato per un progetto ed uno sviluppo "tradizionali", verranno verificate le prove formali da loro effettuate e ne verranno svolte di nuove, con conseguente lievitazione dei costi e i tempi dell'attività di certificazione (v. cap. 5.2.4);
9. *controllo dell'analisi statica del codice*. Esso non deve limitarsi ad un semplice controllo di quella effettuata dai progettisti, ma comprendere tecniche di provata efficacia come software fault tree analysis, ispezioni di Fagan e, se possibile, almeno una tecnica non utilizzata dai progettisti. In questa fase i certificatori individueranno anche i costrutti che non rispondono ad una buona pratica di programmazione (quali puntatori ecc., v. cap. 4.5.4) del software safety-critical, esaminando le giustificazioni addotte nella documentazione allegata ed eventualmente chiederanno

nuove motivazioni scritte e, se queste non saranno soddisfacenti, una modifica del codice. In questa fase si individueranno anche i cammini e i costrutti più critici;

10. *testing con oracolo automatico* ricavato dalla specifica formale, ponendo maggiore attenzione ai cammini e costrutti critici e alla testabilità delle funzioni software. Bisognerà testare particolarmente l'interfaccia con l'utente, simulando anche situazioni fuori del normale (come ad esempio guasto di componenti hardware), tenendo conto, per quanto possibile, del fatto che gli operatori si comportano diversamente in situazioni reali (di rischio);

11. *testing e collaudo sul campo*. Questo tipo di testing, che coinvolge il software implementato proprio nel suo hardware obiettivo e non in un emulatore, è molto efficace, ma incredibilmente più costoso di quello effettuato in laboratorio., Pertanto il suo uso va limitato al minimo, applicandolo soprattutto a supplemento della prototipazione, per la verifica del raggiungimento di requisiti real-time safety-critical, che sono i piu' penalizzati nei test di laboratorio e, in misura minore, per la verifica dell'interfaccia con l'hardware.

Come ultimo commento a conclusione di questo studio si vuole ribadire **l'importanza di una crescita culturale nel settore dell'applicazione dei metodi formali**, da ottenere con uno sforzo combinato tra enti accademici ed industriali. Qualsiasi ingegnere industriale (meccanico, elettrotecnico, ecc.) ha una buona familiarità con i metodi matematici dell'ingegneria classica, quali ad esempio le equazioni differenziali, anche piuttosto sofisticati. **I formalismi utilizzati dai metodi formali citati in questo lavoro sono diversi da quelli classici poichè derivano dalla Logica invece che dall'Analisi matematica, ma non sono per questo di comprensione più ardua**. Si può pertanto confidare che in un prossimo futuro la loro diffusione sarà altrettanto ampia.