Politecnico di Milano
V Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

E-SWEAM: una metodologia per l'analisi di guasto all'interno di applicazioni eseguite su sistemi dedicati

Relatore: Prof.ssa Cristiana BOLCHINI
Correlatore: Ing. Antonio Rosario MIELE

Tesi di Laurea di:
Stefano GUIDOBALDI Matr. 734841

Anno Accademico 2010-2011

Ringraziamenti

Sommario

Con l'utilizzo sempre più frequente di sistemi dedicati in molti ambiti e contesti della vita di tutti i giorni, sono cresciute le esigenze di affidabilità in modo significativo poiché questi sistemi sono sempre più soggetti a *soft errors* causati dall'ambiente nel quale operano anche a causa del processo produttivo con il quale vengono realizzati. Per ovviare a questo problema, i sistemi sono sottoposti a test specifici per testarne l'affidabilità e la suscettibilità ai guasti. Le metodologie di valutazione presenti in letteratura, però, sono carenti per quanto riguarda l'analisi accurata della propagazione degli errori, e non investigano, in modo approfondito, le relazioni che sussistono tra i guasti nell'hardware e gli errori nel software.

Questo lavoro di tesi è frutto di uno studio critico delle metodologie di analisi di affidabilità di sistemi digitali hw/sw. Con questo studio, abbiamo individuato dei limiti sostanziali negli approcci passati che ci hanno portato alla formulazione di una nuova metodologia chiamata *Enhanced Software Error Analysis Methodology* (E-SWEAM). Da queste premesse, lo sviluppo di E-SWEAM è coinciso, fin dall'inizio, con la volontà di proporre un metodo efficace nello studio delle classi di fallimento e nell'analisi delle criticità di un sistema dedicato, siano esse rappresentate da componenti hardware particolarmente suscettibili ai guasti, o siano esse rappresentate da funzionalità software critiche per la corretta esecuzione dell'applicazione.

L'obiettivo di questa tesi, quindi, è di presentare la metodologia che dà il titolo all'elaborato, unitamente a dei casi di studio specifici, nei quali essa è stata applicata, con lo scopo di validarne le ipotesi di partenza. E-SWEAM sfrutta la

tecnica di iniezione di guasti per valutare le proprietà del sistema e per modellarne in modo preciso le tipologie di fallimento. Inoltre, nei casi di studio proposti, abbiamo sfruttato le proprietà di analisi accurata dal punto di vista del software per valutare le proprietà di affidabilità dei sistemi in analisi.

I risultati di questo lavoro sottolineano le potenzialità della metodologia proposta. E-SWEAM è in grado di estrarre, dai modelli simulati, informazioni che in precedenza non era possibile ottenere; queste informazioni comprendono, ad esempio, l'analisi del flusso di chiamate a funzione oppure il numero di errori sui dati in uscita da ogni funzione, ed esse sono facilmente associabili ai guasti iniettati a livello architetturale.

Indice

Indice IX

Elenco delle figure XIII

Elenco delle tabelle XVIII

1 Introduzione **1**

2 Stato dell'Arte **7**

2.1 Definizioni preliminari 8

2.2 Panoramica sugli ambienti di iniezione di guasti 9

2.2.1 *Physical Fault Injection* 10

2.2.2 *Software Implemented Fault Injection* 12

2.2.2.1 *Xception* 13

2.2.2.2 *EXFI* 14

2.2.2.3 *GOOFI* 15

2.2.2.4 OCD-FI 16

2.2.3 *Simulation-based Fault Injection* 17

2.2.3.1 SyFI 20

2.2.3.2 ReSP 22

2.2.4 *Emulation-based Fault Injection* 23

2.2.5 Considerazioni finali sulle strategie e sugli ambienti di iniezione di guasti 26

2.3 Tecniche di analisi dell'affidabilità dei sistemi dedicati 27

2.3.1 FMEA 28

2.3.2 Metodologia FMEA per la certificazione di un circuito in conformità alla norma IEC61508 30
2.3.3 AVF 33
2.3.4 PVF e TVF 35
2.3.5 PRASE 36
2.3.6 Un modello di propagazione di guasti intermittenti all'interno di programmi 38
2.3.7 SWAT 39
2.4 Motivazioni del lavoro proposto 41

3 E-SWEAM 45

3.1 Panoramica generale di E-SWEAM 45
3.1.1 Visione di insieme delle fasi della metodologia E-SWEAM 49
3.1.2 Introduzione al *Framework* proposto per E-SWEAM . . . 55
3.1.3 Analisi Statica dell'Applicazione 57
3.1.4 Analisi Integrale del Sistema 61
3.1.5 Esecuzione *Golden* e salvataggio delle tracce 63
3.1.6 *Monitoring*, Classificazione e Campagne di Iniezione dei guasti mirate 66
3.2 Dettagli dell'implementazione di E-SWEAM 74
3.2.1 *SystemC* e *Transaction-level Modeling* 74
3.2.2 Reflective Simulation Platform 76
3.3 Strategie di iniezione dei guasti basate sulla simulazione 79
3.3.1 Iniezione nello stato interno 80
3.3.2 Iniezione nella comunicazione 81
3.4 Descrizione dei moduli sviluppati in ReSP per il supporto a E-SWEAM 82
3.4.1 *Function Profiler* 83
3.4.2 *Checker Tool* 84

3.4.3 *Delta Monitor* e Sonde 86

4 Casi di Studio **89**

4.1 Caso di studio *Edge Detector Triple Modular Redundancy* 90

4.1.1 Caratterizzazione del sistema 90

4.1.2 Definizione delle strategie di *monitoring* e classificazione . 93

4.1.3 Classificazione dei fallimenti 96

4.1.4 Analisi dei risultati della campagna di iniezione dei guasti 104

4.2 Caso di studio *Anti-lock Braking System* 109

4.2.1 Caratterizzazione del sistema 109

4.2.2 Definizione delle strategie di *monitoring* e classificazione . 115

4.2.3 Classificazione finale dei fallimenti 117

4.2.4 Analisi dei risultati della campagna di iniezione di guasti . 128

5 Conclusioni e Sviluppi Futuri **139**

Bibliografia **149**

A Risultati della campagna di iniezione di guasti nel caso di studio EDTMR **151**

B Risultati della campagna di iniezione di guasti nel caso di studio dell'ABS **155**

Elenco delle figure

2.1 Architettura del *Framework* di Xception. 13
2.2 Panoramica dell'architettura *On-Chip Debug Fault Injection*. . . . 17
2.3 *Framework* di SyFI. 22
2.4 Panoramica del *framework* per la validazione di FMEA a livello SoC. 32
2.5 Esempio di DDG presentato in [39]. 39

3.1 Esempio di sistema hw/sw utilizzato per E-SWEAM. 46
3.2 Flusso di esecuzione della Metodologia E-SWEAM proposta. . . . 51
3.3 Insieme di attività che compongono l'analisi architetturale del sistema. 52
3.4 Panoramica del *framework* associato a E-SWEAM. 56
3.5 Operazioni che compongono l'attività di Analisi Statica durante la Fase Preliminare di E-SWEAM. 59
3.6 Un semplice esempio di un *Execution Flow Graph*. 64
3.7 Flusso di operazioni dell'attività di Analisi delle Tracce. 65
3.8 Classificazione base intermedia e finale supportata da E-SWEAM. 70
3.9 Esempio del diagramma a stati generato dal classificatore: caso base. 71
3.10 Attività di *monitoring*, classificazione ed iniezione dei guasti e loro legame. 73
3.11 Livelli in cima alla libreria standard C++ che compongono il linguaggio *SystemC* ([2]), compreso TLM. 75

3.12 Vista d'insieme dell'architettura di ReSP, con enfasi sui collegamenti tra i diversi moduli. 77
3.13 Dimostrazione della proprietà riflessiva della piattaforma ReSP. . 78

4.1 Modello del sistema utilizzato per il caso di studio EDTMR. . . . 91
4.2 Immagine in ingresso e risultato in uscita dell'applicazione EDTMR. 92
4.3 *Execution Flow Graph* del sistema EDTMR. 94
4.4 Classificazione iniziale degli stati del sistema EDTMR. 97
4.5 Classificazione dei fallimenti causati da errori sui dati. 98
4.6 Classificazione dei fallimenti causati da errori sul flusso di controllo dell'esecuzione. 100
4.7 Caso particolare di errori sui dati causato dal guasto dello *Stack Pointer*. 101
4.8 Classificazione completa di tutti i fallimenti del caso di studio EDTMR. 103
4.9 Modello del sistema utilizzato per il caso di studio ABS. 110
4.10 Sezione della traccia salvata dall'esecuzione *Golden* di ABS. . . . 112
4.11 *Execution Flow Graph* generato durante il caso di studio ABS. . . 114
4.12 Diagramma a stati finale dell'evoluzione del sistema ABS. 119
4.13 Modello del classificatore per il caso di studio dell'ABS - prima fase. 120
4.14 ABS - Diagramma a stati frutto della rifinitura della classe intermedia di errori latenti. 121
4.15 ABS - Diagramma a stati frutto della rifinitura della classe intermedia di errori sul controllo. 122
4.16 Rifinitura della classe di fallimento intermedia *Thread Synchronization Error*. 124
4.17 ABS - Diagramma a stati che lega gli errori di controllo con gli errori sui dati. 125
4.18 ABS - Diagramma a stati frutto della rifinitura della classe intermedia di errori sui dati. 126

4.19 Diagramma completo degli stati intermedi e delle classi di fallimento del caso di studio ABS. 129

Elenco delle tabelle

3.1 Esempio di *Functions Descriptor Table*: Funzioni. 60

3.2 Esempio di *Functions Descriptor Table*: Parametri di una funzione. 61

4.1 Percentuali di immagini scorrette per ogni funzionalità guastata. . 105

4.2 Risultati delle iniezioni nelle celle di memoria del sistema EDTMR. 106

4.3 Legenda per le sigle dei fallimenti contenuti in Tabella 4.4. 131

4.4 Tipologie di fallimenti suddivisi per funzionalità e *task* corrotti. . 132

4.5 Differenze percentuali di lancio di eccezioni tra variabili passate per riferimento e variabili passate per valore. 134

4.6 Risultati delle iniezioni nella memoria effettuate in istanti temporali *random* durante l'intera esecuzione dell'applicazione. 137

A.1 Risultati delle iniezioni nei registri #0 e #1 del primo processore, contenenti gli indirizzi delle immagini. 151

A.2 Risultati delle iniezioni nei registri #0 e #1 dei processori secondari. 152

A.3 Risultati delle iniezioni nei registri #2 e #3 del primo processore, contenenti le dimensioni dell'immagine, passate per valore. 152

A.4 Risultati delle iniezioni nei registri #2 e #3 dei processori secondari. 152

A.5 Risultati delle iniezioni nel registro #11 del primo processore, contenente il *Link Register*. 153

A.6 Risultati delle iniezioni nel registro #11 dei processori secondari. . 153

A.7 Risultati delle iniezioni nel registro #13 del primo processore, contenente lo *Stack Pointer*. 153

A.8 Risultati delle iniezioni nel registro #13 dei processori secondari. . 154

A.9 Risultati delle iniezioni nel registro #15 del primo processore, contenente il *Program Counter*. 154

A.10 Risultati delle iniezioni nel registro #15 dei processori secondari. . 154

B.1 Risultati delle iniezioni nel registro #0 del primo processore durante il *task* di acquisizione dei dati eseguito dal *thread* principale. . . 155

B.2 Risultati delle iniezioni nel registro #1 del primo processore durante il *task* di acquisizione dei dati eseguito dal *thread* principale. . . 156

B.3 Risultati delle iniezioni nel registro #0 del secondo processore durante il *task* di acquisizione dei dati eseguito dal *thread* secondario. 156

B.4 Risultati delle iniezioni nel registro #0 del primo processore durante il *task* di elaborazione dei dati eseguito dal *thread* principale. . . 157

B.5 Risultati delle iniezioni nel registro #1 del primo processore durante il *task* di elaborazione dei dati eseguito dal *thread* principale. . . 157

B.6 Risultati delle iniezioni nel registro #0 del secondo processore durante il *task* di elaborazione dei dati eseguito dal *thread* secondario. 158

B.7 Risultati delle iniezioni nei registri #0-#3 del primo processore durante il *task* di attuazione eseguito dal *thread* principale. . . . 159

B.8 Risultati delle iniezioni nel registro #11 del primo processore durante il *task* di acquisizione dei dati eseguito dal *thread* principale. 160

B.9 Risultati delle iniezioni nel registro #11 del secondo processore durante il *task* di acquisizione dei dati eseguito dal *thread* secondario. 160

B.10 Risultati delle iniezioni nel registro #11 del primo processore durante il *task* di elaborazione dei dati eseguito dal *thread* principale. 161

B.11 Risultati delle iniezioni nel registro #11 del secondo processore durante il *task* di elaborazione dei dati eseguito dal *thread* secondario. 161

B.12 Risultati delle iniezioni nel registro #11 del primo processore durante il *task* di attuazione eseguito dal *thread* principale. 162

B.13 Risultati delle iniezioni nei registri #12 e #13 del primo processore durante il *task* di acquisizione dei dati eseguito dal *thread* principale. 162

B.14 Risultati delle iniezioni nei registri #12 e# 13 del secondo processore durante il *task* di acquisizione dei dati eseguito dal *thread* secondario. 163

B.15 Risultati delle iniezioni nei registri #12 e #13 del primo processore durante il *task* di elaborazione dei dati eseguito dal *thread* principale. 163

B.16 Risultati delle iniezioni nei registri #12 e #13 del secondo processore durante il *task* di elaborazione dei dati eseguito dal *thread* secondario. 163

B.17 Risultati delle iniezioni nei registri #12 e #13 del primo processore durante il *task* di attuazione eseguito dal *thread* principale. . . . 164

B.18 Risultati delle iniezioni nei registri #14 e #15 del primo processore durante il *task* di acquisizione dei dati eseguito dal *thread* principale. 164

B.19 Risultati delle iniezioni nei registri #14 e #15 del secondo processore durante il *task* di acquisizione dei dati eseguito dal *thread* secondario. 165

B.20 Risultati delle iniezioni nei registri #14 e #15 del primo processore durante il *task* di elaborazione dei dati eseguito dal *thread* principale. 165

B.21 Risultati delle iniezioni nei registri #14 e #15 del secondo processore durante il *task* di elaborazione dei dati eseguito dal *thread* secondario. 166

B.22 Risultati delle iniezioni nei registri #14 e #15 del primo processore durante il *task* di attuazione eseguito dal *thread* principale. . . . 166

Capitolo 1

Introduzione

Al giorno d'oggi garantire il corretto funzionamento dei sistemi digitali è divenuto una necessità molto più rilevante rispetto al passato, a causa dei notevoli progressi tecnologici nel processo produttivo dei circuiti integrati, che hanno portato alla realizzazione di dispositivi sempre più suscettibili a guasti di natura transitoria, comunemente chiamati *soft errors*. Questi fenomeno, particolarmente pericolosi per applicazioni critiche, assumono ancora più importanza quando si considera l'impiego diffuso dei circuiti digitali dedicati (o *embedded*) nei sistemi e nelle apparecchiature che ci circondano nella vita di tutti i giorni. Di conseguenza, l'affidabilità ha assunto un ruolo di primo piano nella progettazione dei sistemi dedicati alla pari degli aspetti classici, quali le prestazioni o il consumo di potenza.

I sistemi *embedded* vengono spesso impiegati in contesti *safety-critical*, ma, sempre più spesso, il loro utilizzo viene esteso anche ad applicazioni di uso comune. La pervasività raggiunta da questa famiglia di dispositivi richiede che essi siano in grado di reagire in modo corretto alla presenza di un errore nel proprio stato interno, poiché un loro fallimento potrebbe rappresentare un potenziale pericolo per le persone o l'ambiente nel quale essi operano. Dispositivi *automotive*, aerospaziali e ferroviari, infrastrutture di rete, sistemi industriali, elettronica di consumo e applicazioni per la domotica sono solo alcuni esempi dell'uso intensivo, fatto oggigiorno, di sistemi dedicati. Per la maggior parte di questi scenari sono richiesti meccanismi per l'autodiagnosi dei guasti o per l'irrobustimento dell'ar-

chitettura, meccanismi che sono necessari per garantire proprietà di affidabilità adeguate al contesto operativo di riferimento.

L'affidabilità non è una problematica del tutto nuova; al contrario, è stata ampiamente affrontata in letteratura, dove è possibile trovare un grande numero di tecniche per l'introduzione di proprietà di individuazione e/o tolleranza ai guasti. Purtroppo, la progettazione dei sistemi dedicati sta diventando un'attività sempre più complicata, e gli approcci, come quelli proposti in letteratura, non effettuando un'analisi sistematica di diverse strategie di irrobustimento, non risultano efficaci, poiché rischiano di portare a soluzioni costose e con caratteristiche non ottimali. Al contrario, c'è la necessità di considerare gli aspetti relativi all'affidabilità durante tutto il flusso di progetto adottando un approccio che integri tutti i diversi aspetti, e che quindi guidi le varie scelte progettuali sfruttando la sinergia sia dei parametri classici di progetto sia di quelli strettamente legati all'affidabilità.

Le metodologie presenti in letteratura si servono della tecnica di iniezione di guasti per validare le analisi effettuate. L'iniezione di guasti, in questo tipo di scenario, ricopre un ruolo di primaria importanza, consentendo, a chi si occupa dell'analisi, di studiare il comportamento del sistema in presenza di un guasto al suo interno. Per ottenere dati che abbiano rilevanza statistica, i singoli esperimenti sono raggruppati in campagne di iniezione di guasti.

Una campagna di iniezione dei guasti richiede che il progettista selezioni un sottoinsieme di tutte le possibili locazioni nelle quali potrebbe essere iniettato un guasto, che selezioni un istante di tempo arbitrario nel quale effettuare l'iniezione, e, infine, che stabilisca una finestra temporale per monitorare il comportamento del sistema, che deve essere confrontato con la corretta evoluzione dello stesso in assenza di guasto. Lo scopo di questo confronto è di stilare un resoconto specifico delle proprietà di affidabilità del sistema in analisi; per esempio, in caso di sistemi tolleranti ai guasti, il resoconto può avere come risultato "nessun effetto" oppure "fallimento". Analisi statistiche ricavate dal resoconto finale di una campagna

possono essere sufficienti per soddisfare i requisiti di calcolo di percentuale di copertura dei guasti, un parametro che viene spesso preso in considerazione quando è necessario validare l'affidabilità di un sistema. La percentuale di copertura dei guasti ha il compito di segnalare la percentuale di guasti iniettati che ha causato un errore sulle uscite del sistema.

Questo approccio, però, risulta limitato nei contesti descritti precedentemente. In situazioni come queste, infatti, il progettista ha bisogno di un quadro chiaro e specifico delle relazioni che legano i guasti con gli errori, in modo da realizzare come lo stato del sistema sia influenzato dal guasto, oppure in modo da investigare per quale motivo un guasto produce, o meno, un errore sulle uscite.

Per questo motivo, specialmente nelle prime fasi del flusso di progettazione di un sistema dedicato, l'attività di monitoraggio viene effettuata non solo sulle uscite del sistema, ma anche sugli elementi interni di memoria. In questo scenario, può essere considerata un'ulteriore classe che descrive il comportamento del sistema, chiamata latente, che definisce le situazioni nelle quali, alla fine della finestra temporale di osservazione considerata, le uscite del sistema sono corrette, ma lo stato interno è corrotto, cosa che può causare il verificarsi di un errore in un istante temporale successivo al termine della finestra di osservazione. Sfortunatamente, questo approccio si limita, ancora una volta, alla valutazione della presenza di uno stato corrotto e, opzionalmente, del modulo hardware compromesso; nessuna informazione aggiuntiva è disponibile sullo stato dell'applicazione eseguita dal sistema dedicato. Questo tipo di informazione, invece, considerando la complessità dei moderni sistemi dedicati, e considerando che sono spesso dotati di architetture multiprocessore e di acceleratori hardware, è fondamentale per analizzare in che modo l'applicazione è fallita, causando la generazione di un risultato finale diverso. In particolare, è interessante studiare quali funzionalità del sistema sono state compromesse a causa di un guasto, e quali conseguenze possono avere, sul comportamento del sistema, simili problematiche.

Per queste ragioni, riteniamo sia necessario un nuovo approccio per l'analisi dei risultati di campagne di iniezione di guasti, in grado di fornire i mezzi per investigare gli effetti di un guasto non solo sulle uscite, ma anche nello stato del sistema durante l'intera esecuzione dell'applicazione. Inoltre, è necessario che lo studio del fallimento sia compiuto dal punto di vista del comportamento anomalo osservato a livello del software. Crediamo, infatti, che, quando si considera la complessità dei moderni sistemi dedicati, l'analisi degli effetti nelle diverse fasi dell'applicazione sia fondamentale per capire le funzionalità più critiche, oppure per identificare le cause del fallimento delle tecniche di irrobustimento applicate. Ad esempio, nei sistemi di *safety*, assicurare il corretto funzionamento del dispositivo in presenza di un guasto è un requisito necessario. Sistemi di questo tipo sono dotati di funzionalità aggiuntive che devono garantire che, in presenza di un errore, l'esecuzione dell'applicazione non sia compromessa e consenta al sistema di produrre un risultato finale corretto. Vogliamo, quindi, concentrarci sull'analisi della proprietà di affidabilità di simili sistemi, al fine di garantirne l'adeguata robustezza.

Il nostro scopo è proporre una nuova metodologia e relativo ambiente che consenta di effettuare una classificazione dei guasti e un'analisi della propagazione degli errori a tutti i livelli di astrazione, a supporto dello studio del legame guasto-errore sia dal punto di vista architetturale che, soprattutto, dal punto di vista applicativo. Questo ambiente è stato sviluppato sulla base di un motore per l'iniezione di guasti allo stato dell'arte. Il *framework* è basato su un meccanismo simile a quello di un *debugger*, ed è in grado di interpretare dati grezzi, estratti dall'architettura, dal punto di vista dell'applicazione; è, inoltre, basato su una strategia di monitoraggio che sfrutta un insieme di condizioni sullo stato del sistema, definite dall'utente, in grado di offrire una classificazione specifica dei fallimenti a supporto dell'analisi di criticità delle proprietà di affidabilità del sistema. Presenteremo un'implementazione di questo ambiente, che lavora su spe-

cifiche *SystemC Transaction-Level Modeling*, e descriveremo nuovi meccanismi e strategie di analisi per offrire una valutazione dettagliata e personalizzata degli effetti di *soft errors* all'interno di sistemi *embedded*.

Questo lavoro di tesi è organizzato nel modo seguente.

Nel Capitolo 2 vengono presentate le strategie per l'iniezione di guasti all'interno di sistemi digitali e alcuni approcci per l'analisi di affidabilità presenti in letteratura. Le strategie di iniezione che hanno più punti in comune con il nostro lavoro (*Software Implemented Fault Injection* e *Simulation-based Fault Injection*) sono affiancate dalla presentazione di alcuni ambienti che le utilizzano per lo svolgimento di campagne di iniezione di guasti. Il capitolo si conclude con la discussione delle motivazioni che ci hanno portato allo sviluppo di questo lavoro, che sono derivate dall'analisi dei limiti dei lavori presentati nelle sezioni precedenti.

Nel Capitolo 3 viene descritta in modo dettagliato la metodologia di analisi approfondita per sistemi dedicati oggetto del lavoro presentato in questo elaborato, chiamata *Enhanced Software Error Analysis Methodology* (E-SWEAM). La metodologia viene dapprima presentata da un punto di vista teorico; viene, quindi, descritto il *framework* a supporto della metodologia. Viene fatto, in seguito, un breve accenno al livello di astrazione per la specifica del sistema simulato, e viene quindi presentata la piattaforma che abbiamo scelto per lo svolgimento dei casi di studio a supporto della validazione di E-SWEAM. Viene discusso, inoltre, un breve approfondimento su quelle che sono le tecniche utilizzate per l'iniezione di guasti all'interno dell'architettura. Il capitolo si chiude con la descrizione degli strumenti, progettati ed implementati all'interno della piattaforma di simulazione ed iniezione, per la caratterizzazione e la classificazione del comportamento del sistema analizzato.

Nel Capitolo 4 sono presentati due casi di studio, *EdgeDetectorTripleModularRedundancy* (EDTMR) e *Anti-lock Braking System* (ABS), a supporto della validazione della metodologia. Per entrambi sono descritti, nel dettaglio, i passi

che hanno portato alla valutazione finale delle criticità. I casi di studio si riferiscono a due sistemi dedicati molto diversi tra loro. In particolare, il primo è un sistema dedicato all'elaborazione di immagini, mentre il secondo è un sistema di *safety time-critical*. EDTMR è stato irrobustito dal punto di vista software, attraverso l'implementazione, a livello di funzione, di un meccanismo di *Triple Modular Redundancy*. Il meccanismo, unito all'esecuzione replicata dell'applicazione in tre diversi *thread*, ha lo scopo di garantire che l'immagine restituita dal sistema sia corretta anche in presenza di guasto in un processore. ABS, invece, gode di proprietà di individuazione e correzione degli errori grazie all'irrobustimento del codice dell'applicazione e dell'implementazione di una tecnica basata sul meccanismo di *Rollback &Replay*. Lo scopo di questi due casi di studio è verificare l'efficacia, attraverso E-SWEAM, dei meccanismi di affidabilità introdotti, e fornire un quadro completo dei possibili modi di fallimento, dal punto di vista del software, e verificare il tipo di impatto che essi hanno sull'intero sistema.

Nel Capitolo 5 vengono tratte le conclusioni finali sul lavoro svolto, con un accenno a quelli che possono essere gli sviluppi futuri della nostra metodologia nell'ambito di analisi dell'affidabilità di sistemi dedicati.

Negli Appendici A e B, infine, sono presentati, nel dettaglio, i risultati delle campagne di iniezione di guasti effettuate durante lo svolgimento dei casi di studi proposti. In particolare, sono mostrate le percentuali e le statistiche dei fallimenti osservati durante la fase di classificazione della metodologia applicata ai sistemi EDTMR e ABS.

Capitolo 2

Stato dell'Arte

In questo capitolo vengono presentati i concetti preliminari necessari alla comprensione degli argomenti discussi in questo lavoro di tesi.

Il capitolo è strutturato in quattro sezioni: nella prima sezione, saranno introdotte brevemente alcune definizioni che riguardano l'ambito di affidabilità dei sistemi digitali; successivamente, verranno discusse le principali strategie di iniezione di guasti, corredate da esempi di ambienti proposti che le utilizzano; nella terza sezione, la trattazione sarà rivolta all'analisi di metodologie per lo studio di affidabilità dei sistemi proposte in letteratura. Da queste discussioni emergeranno i limiti e le carenze dei lavori passati, riassunte alla fine del capitolo, che ci hanno spinto alla definizione di un nuovo approccio e che ci hanno quindi motivato alla stesura di questo lavoro.

Per concludere l'introduzione al capitolo di stato dell'arte, è doveroso sottolineare che le metodologie presentate nella terza sezione del capitolo si servono di alcune delle strategie presentate nella sezione di panoramica sugli ambienti di iniezione. Per questo motivo, verranno prima discusse le strategie di iniezione per poi introdurre le metodologie che ne fanno uso per la validazione della fase di analisi.

2.1 Definizioni preliminari

L'analisi di affidabilità ricopre, al giorno d'oggi, un'importanza fondamentale nell'ambito della progettazione di sistemi dedicati, o sistemi *embedded*. Questo tipo di analisi ha l'obiettivo di individuare le cause che possono portare alla deviazione dal comportamento atteso del sistema in seguito all'occorrenza di un guasto. In generale, è possibile classificare i problemi legati al non corretto funzionamento di un sistema in:

- **guasto**: è un difetto, un'imperfezione o un degrado fisico che avviene all'interno di un determinato componente hardware;
- **errore**: è una deviazione dalla corretta esecuzione e rappresenta la manifestazione di un guasto;
- **fallimento**: rappresenta la condizione di effettiva deviazione del comportamento del sistema dal risultato atteso o desiderato.

Quando un guasto causa un cambiamento scorretto nello stato del sistema, avviene un errore. Nonostante il guasto sia localizzato in un punto ben preciso del sistema, si possono generare errori multipli che si propagano all'interno dell'intero sistema. Quando i meccanismi di rilevazione e di tolleranza ai guasti individuano un errore, possono intraprendere delle azioni correttive per gestire il guasto e contenere gli errori. Altrimenti, esiste il rischio che il sistema subisca un malfunzionamento e che ciò porti ad un fallimento.

Un guasto può manifestarsi, all'interno di un componente, durante le fasi di realizzazione, che comprendono la produzione, l'assemblaggio e l'installazione, oppure durante il suo normale utilizzo, in qualsiasi momento del ciclo di vita. I tipi di guasti fisici possono essere classificati in base alla loro durata in:

- **Guasti Permanenti**: causati da un danno irreversibile del componente. Solitamente, l'unica soluzione, in caso di occorrenza questo tipo di guasti, è la sostituzione del componente o la riparazione.

- **Guasti Transitori**: causati dalle condizioni ambientali nelle quali il sistema viene utilizzato, che possono presentare fenomeni di fasci di radiazioni o di interferenze elettromagnetiche. Questi guasti difficilmente causano danni a lunga durata al componente che affliggono; possono però indurre il sistema in uno stato erroneo. Secondo diversi studi, essi hanno una probabilità di accadere molto più elevata rispetto ai guasti di tipo permanente.

- **Guasti Intermittenti**: causati da hardware instabile o da stati variabili dello hardware. Possono essere gestiti attraverso la sostituzione o la riprogettazione.

In questo lavoro di tesi, l'attenzione si è focalizzata sull'individuazione e la gestione di guasti di tipo transitorio, molto più frequenti e difficili da diagnosticare rispetto agli altri tipi di guasto.

2.2 Panoramica sugli ambienti di iniezione di guasti

L'iniezione di guasti, o *Fault Injection* (FI), è definita in [45] come la tecnica di validazione dell'affidabilità di sistemi tolleranti ai guasti, e consiste nell'esecuzione di esperimenti controllati dove viene osservato il comportamento del sistema a fronte dell'introduzione, all'interno di esso, di uno o più guasti.

L'iniezione di guasti mira a determinare se la risposta del sistema, in presenza di un insieme definito di guasti, corrisponde alle specifiche. Per fare ciò sono necessarie campagne estese di iniezione di guasti, cioè una serie di singoli esperimenti di iniezione per i quali vengono raccolti e analizzati i risultati finali, in modo da avere a disposizione una serie di dati che abbiano valenza statistica. Di norma, i guasti vengono iniettati in stati o locazioni del sistema, definiti da un'iniziale analisi architetturale. Si richiede, al progettista che si occupa dell'esecuzione della campagna e dell'analisi dei risultati, la conoscenza approfondita del modello

del sistema e la capacità di ideare casi di test mirati a evidenziare i punti critici dell'architettura.

Molti approcci per l'analisi di affidabilità dei sistemi si servono di questa strategia per la validazione dei risultati ottenuti con analisi statiche di criticità oppure studiare con accuratezza la propagazione degli errori all'interno del sistema. Per iniettare guasti all'interno di un prototipo o di un modello di un sistema sono state sviluppate diverse tecniche, che possono essere raggruppate in quattro grandi categorie:

- *Physical Fault Injection*: l'implementazione finale del circuito viene esposta a radiazioni ionizzanti, interferenze elettromagnetiche, etc.;
- *Software Implemented Fault Injection*: ha l'obiettivo di riprodurre a livello software gli errori che si sarebbero verificati se si fosse guastato l'hardware sottostante;
- *Simulation-based Fault Injection*: consiste nell'iniezione di guasti in modelli di sistemi ad alto livello, simulati su computer;
- *Emulation-based Fault Injection*: una soluzione alternativa alla simulazione che riduce la durata degli esperimenti attraverso l'emulazione del sistema, ad esempio mediante l'utilizzo di un dispositivo FPGA.

Nelle prossime sottosezioni saranno discusse, nel dettaglio, le strategie appena presentate. Per ognuna di esse, a parte *Physical Fault Injection* che è oramai adottata solamente per la validazione finale di un sistema sul quale è stata effettuata un'analisi preliminare di affidabilità, saranno presentati alcuni ambienti, proposti negli anni precedenti, che cercano di sfruttarne le peculiarità.

2.2.1 *Physical Fault Injection*

Physical Fault Injection (PFI) prevede lo studio degli effetti di un guasto indotto dall'esposizione del sistema a effetti esterni oppure indotto da componenti inseriti

all'interno di esso. Perché PFI possa essere utilizzata, è necessario avere a disposizione un prototipo assemblato e funzionante del sistema, cosa che già di per sé rappresenta una limitazione poiché implica che PFI non può essere sfruttata nelle fasi iniziali di progettazione, quando i costi di un'eventuale modifica del sistema sono maggiormente contenuti.

In base al tipo di guasti e alla loro locazione, PFI si divide in due categorie:

- PFI con contatto, nel caso in cui l'iniettore di guasti ha un contatto diretto col sistema, causando cambiamenti di voltaggio o di corrente esternamente al *chip* considerato; esempi di questa categoria sono metodi che si basano sull'utilizzo di sonde a livello *pin* oppure sull'inserimento di *socket* tra il componente considerato ed il circuito stampato;

- PFI senza contatto: è la più efficace tra le due tecniche, e prevede che l'iniettore di guasti non abbia diretto contatto fisico col sistema. Una fonte esterna si occupa della generazione di fenomeni fisici, come radiazioni ionizzanti o interferenze elettromagnetiche, che causano correnti spurie all'interno del *chip*.

Questa strategia è più adatta all'iniezione di guasti di tipo transitorio, poiché l'introduzione di guasti permanenti, ad esempio di tipo *stuck-at* o *bridging*, è molto più complicata soprattutto nel caso di PFI senza contatto, vista la scarsa controllabilità che si ha sul circuito in analisi. Il fatto di dover inserire (internamente o esternamente) una strumentazione di test per iniettare guasti e monitorarne gli effetti richiede tempo per la messa a punto del sistema, che può essere, però, risparmiato in fase di esecuzione degli esperimenti, molto più rapidi rispetto a strategie che sfruttano la simulazione o l'emulazione di interi modelli complessi.

Nonostante le buone prestazioni, PFI offre una scarsa controllabilità in termini delle locazioni nelle quali iniettare i guasti e, soprattutto, presenta rischi legati al danneggiamento irreparabile del sistema digitale in analisi.

2.2.2 *Software Implemented Fault Injection*

Tradizionalmente, *Software Implemented Fault Injection* (SWIFI) concerne la modifica del software eseguito all'interno del sistema in analisi con lo scopo di simulare l'insorgere di un guasto nell'hardware sottostante. Possono essere iniettati, in questo modo, tutti i tipi di guasti, da quelli nei registri o nella memoria, a perdite di pacchetti a livello di rete, a condizioni o *flag* errati. SWIFI è orientata ai dettagli dell'implementazione, e si rivolge allo stato del programma così come alle sue comunicazione e alle sue interazioni. Gli errori implementabili all'interno dei programmi possono essere di diverso tipo: da letture errate, a messaggi ripetuti, ad accessi errati ai dischi, fino a errori di sincronizzazione.

I vantaggi principali di questa strategia risiedono nella sua assoluta non intrusività rispetto all'hardware, nell'osservazione degli effetti accurata a livello del software sovrastante l'architettura, nell'espandibilità, relativamente semplice, a nuovi modelli di guasto e nella mancanza di strumenti accessori che andrebbero introdotti nel sistema in analisi.

In compenso, generalmente, per sfruttarla è necessario modificare l'applicazione eseguita o utilizzare sistemi operativi *ad-hoc* che implementino l'iniezione dei guasti, modificando, di fatto, il comportamento che l'applicazione mostrerebbe nel sistema reale. Tra gli altri svantaggi è possibile, inoltre, sottolineare: insieme ridotto di istanti validi per l'iniezione, in particolare solo a livello di istruzione *assembly*, una granularità che in alcuni casi può non essere sufficientemente fine; impossibilità di iniezione di guasti in locazioni inaccessibili dal software, come, ad esempio, i registri privilegiati del processore; infine, difficoltà elevate nella modellazione di guasti permanenti.

I metodi SWIFI possono essere categorizzati in base al momento nel quale sono iniettati i guasti: a *compile-time* o a *run-time*. Nel primo caso, è sufficiente modificare il codice per iniettare un guasto durante l'esecuzione, un'operazione relativamente semplice e che non causa perturbazioni eccessive, ma decisamente

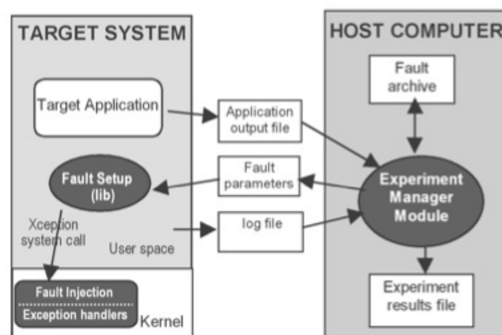


Figura 2.1: *Architettura del Framework di Xception.*

poco duttile. Nel secondo caso, invece sono necessari meccanismi che facciano scattare l'iniezione a *run-time*, come ad esempio: *timeout*, *exception/trap* oppure inserimento di codice, tecnica diversa dall'iniezione a *compile-time* perché non prevede la modifica del codice sorgente, ma il solo inserimento di codice a *run-time* che ha il compito di iniettare il guasto.

In seguito sono presentati quattro esempi di ambienti che adottano la strategia di iniezione SWIFI.

2.2.2.1 *Xception*

Progettato all'Università di Coimbra in Portogallo, [13], utilizza funzionalità avanzate di monitoraggio delle prestazioni attraverso l'uso di sonde, presenti ormai nella maggior parte dei moderni processori, per iniettare guasti. Sfrutta, inoltre, le eccezioni del processore per attivare i guasti. Non richiede modifiche nel software né la specifica di ulteriori *trap*. L'iniettore di guasti è implementato come un gestore delle eccezioni e richiede la sola modifica del vettore di gestione delle *interrupt*. I guasti sono attivati tramite l'accesso a determinati indirizzi. Questo rende gli esperimenti facilmente riproducibili. Xception usa una maschera quando un guasto viene iniettato in una locazione del sistema. La maschera viene confrontata con memoria/registri/dati e i bit che sono posti a 1 sulla maschera sono cambiati nel sistema usando operazioni a livello di bit come *stuck-at-zero*, *stuck-at-one* e *bridging*.

In Figura 2.1 è mostrata l'architettura del *framework* presentato. Xception considera solamente gli errori che attivano delle eccezioni all'interno del sistema. Questa classe di errori è solo una delle possibili classi di fallimento che devono essere considerate quando si effettua l'analisi di affidabilità di un sistema sfruttando la tecnica SWIFI. La classificazione ternaria proposta (*No-error/Detected/Undetected*) è limitante e non sufficientemente dettagliata per descrivere le modalità di fallimento del sistema da un punto di vista del software.

2.2.2.2 *EXFI*

EXFI è un sistema di iniezione di guasti per circuiti *embedded* basati su microprocessori sviluppato al Politecnico di Torino ([11]). Il kernel del sistema EXFI è basato sul *trace exception mode* disponibile in molti microprocessori attuali. Durante gli esperimenti di iniezione di guasti, la routine di gestione della traccia dell'eccezione è incaricata di calcolare l'istante dell'iniezione, di eseguire fisicamente l'iniezione e di attivare possibili condizioni di timeout. Lo strumento è in grado di iniettare singoli *bit-flip* sia nell'immagine di memoria del processore (dati e istruzioni) sia nei registri dedicati agli utenti. L'approccio può essere facilmente esteso per supportare diversi modelli di guasti permanenti, come ad esempio guasti *stuck-at, couple, bridging*, etc. Le caratteristiche principali di EXFI sono il costo ridotto (non richiede hardware aggiuntivo), l'elevata velocità, la necessità di non introdurre funzionalità aggiuntive al sistema operativo, la flessibilità (supporta diversi tipi di guasto) e l'elevata portabilità (può essere facilmente migrato per operare su sistemi differenti).

Un particolare interessante di EXFI è che sfrutta l'analisi delle tracce per implementare un meccanismo che ha il compito di fermare l'esecuzione in presenza di un *loop*. In questo particolare ambiente, una tecnica come l'analisi delle tracce può rivelarsi molto utile se sfruttata in modo adeguato. La metodologia presentata nel Capitolo 3, ad esempio, si serve di questa tecnica per confrontare gli stati intermedi dell'applicazione e dell'architettura durante l'intera esecuzione del programma.

Gli autori di EXFI, invece, al di là dell'utilizzo nel meccanismo per la gestione dei *loop*, non hanno valorizzato sufficientemente le potenzialità dell'analisi delle tracce.

2.2.2.3 *GOOFI*

Generic Object-Oriented Fault Injection ([6]) (GOOFI) è stato sviluppato al dipartimento di Ingegneria Informatica dell'Università Tecnologica Chalmers in Svezia. Il più grande obiettivo che gli autori si sono posti è stato di fornire un ambiente di iniezione dei guasti facilmente usabile e personalizzabile, con un'interfaccia grafica pratica e fornito di un'architettura sottostante di base. L'utente che utilizza GOOFI può adattarlo alle proprie necessità modificando i componenti dell'architettura a disposizione e specificando quali tipi di guasti iniettare all'interno.

GOOFI è altamente portabile tra differenti piattaforme ospiti, dato che è stato implementato utilizzando il linguaggio di programmazione Java e un database compatibile in SQL. Inoltre, per incrementare la flessibilità e la manutenibilità dello strumento, è stato scelto un approccio orientato agli oggetti. L'attuale versione di GOOFI supporta SWIFI *pre-runtime* e *Scan Chain Implemented Fault Injection* (SCIFI). Quest'ultima strategia prevede l'iniezione di guasti attraverso la logica di test integrata, cioè le cosiddette *scan-chains* interne e periferiche, presenti nella maggior parte dei moderni circuiti VLSI. Ciò garantisce che i guasti possano essere iniettati a livello *pin* o negli stati interni dei componenti del circuito integrato, così come viene garantita l'osservabilità dello stato interno del sistema. In SWIFI *pre-runtime*, invece, i guasti sono iniettati nel programma e nelle aree dati del sistema (memorie) prima dell'inizio dell'esecuzione vera e propria. GOOFI è in grado di iniettare guasti transitori singoli o multipli.

GOOFI è dotato di un modulo base per l'analisi dei risultati delle campagne in grado di estrarre informazioni dai singoli esperimenti effettuati. Il modulo in questione, comunque, non è in grado di effettuare analisi approfondite del modello guasto. I dati contenuti nel database, relativi allo stato del sistema, sono

processati per ottenere metriche di affidabilità. Il tipo di metriche dipende dal tipo di sistema in analisi. I risultati che si possono ottenere dalla fase di analisi coinvolgono le modalità di rilevazione dell'errore, che può essere individuato dal sistema, osservato attraverso il *framework* di GOOFI, oppure non essere rilevato con nessuna delle tecniche a disposizione.

2.2.2.4 OCD-FI

Fidalgo et al. propongono un'architettura avanzata *On-Chip Debug* (OCD) che ha l'obiettivo di supportare la verifica di meccanismi di tolleranza ai guasti di microprocessori attraverso campagne di iniezione dei guasti ([20]). Questo *debugger* su chip avanzato (chiamato OCD-FI per sottolineare la presenza di un iniettore di guasti al suo interno) fornisce un meccanismo efficiente di iniezione dei guasti unito a proprietà dinamiche.

Molti microprocessori moderni supportano l'accesso al proprio stato interno attraverso una circuiteria dedicata di debug. Visto che questo modulo è integrato e può accedere agli stati interni in modo trasparente e non intrusivo, gli autori hanno pensato di sfruttarlo per iniettare guasti direttamente all'interno dell'hardware e osservare i risultati. I loro esperimenti hanno confermato che l'inserimento di guasti all'interno dello spazio di memoria e sfruttare l'analisi delle tracce di ogni istruzione del processore è possibile ed è un metodo efficiente per analizzare il flusso di programma, prima e dopo l'attivazione di un guasto.

In Figura 2.2 è mostrata l'architettura proposta. Lo standard utilizzato definisce un insieme minimo di dispositivi di *debugging*, una porta per l'interfacciamento e il protocollo di comunicazione. Il modulo *Message Queuing and Management* implementa il controllore dell'OCD. Il modulo *Run Control & Trace* riceve i comandi dal MQM e dal RWA e restituisce le tracce e i segnali del processore. Infine, il modulo *Read & Write Access* è usato per accedere ai registri dell'OCD e alle risorse della CPU (memorie e registri). Questa architettura viene emulata attra-

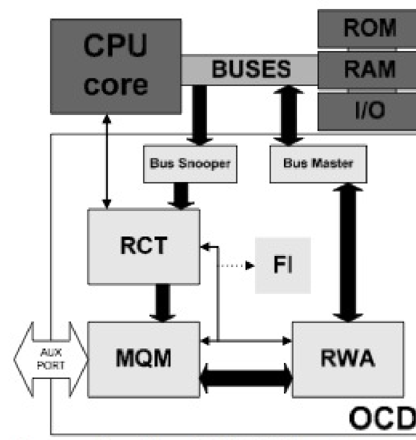


Figura 2.2: *Panoramica dell'architettura On-Chip Debug Fault Injection.*

verso una FPGA connessa ad un PC che si occupa della gestione delle campagne di iniezione.

Questo lavoro ha offerto delle idee interessanti sia per quanto riguarda l'approccio di analisi del sistema con la tecnica di *debugging*, sia per la novità della simulazione di un sistema reale nel quale si sfrutta un componente presente nei moderni processori per effettuare l'iniezione, favorendo prestazioni molto elevate. Il limite più evidente, però, è rappresentato dal tipo di analisi effettuata, di tipo architetturale, che non considera il livello applicazione soprastante.

2.2.3 *Simulation-based Fault Injection*

Simulation-based Fault Injection (SFI) prevede la definizione di un modello simulato del sistema in analisi. La simulazione del modello viene sviluppata utilizzando specifiche scritte con linguaggi di descrizione dell'hardware (*Hardware Description Languages*, HDL), come ad esempio VHDL, Verilog e SystemC. I guasti sono iniettati nel modello ad alto livello e attivati da un determinato vettore di ingresso.

Un esperimento di iniezione elementare consiste ad una simulazione eseguita sul sistema durante la quale un numero qualsiasi di guasti viene iniettato in locazioni singole o multiple dell'architettura e in uno o più istanti temporali. Una

campagna di iniezione dei guasti consiste nell'esecuzione di una sequenza di singoli esperimenti di iniezione elementari.

Per implementare SFI in modo efficace, sono state proposte, in passato, diverse tecniche, basate su due diversi approcci:

- approccio basato sulla modifica del codice della specifica HDL: il sistema viene modificato introducendo componenti atti all'iniezione dei guasti chiamati *saboteurs* oppure modificando le descrizioni esistenti dei componenti nella specifica HDL, generando una descrizione modificata chiamata *mutante*. I *saboteurs* sono componenti inattivi durante la normale esecuzione del sistema. Quando vengono attivati, però, hanno il compito di alterare i valori o le tempistiche di uno o più segnali ai quali sono associati. I *saboteurs*, comunque, sono trattati in modo più ampio in Sezione 3.3 poiché sono presenti nel *framework* di iniezione di guasti utilizzato nella metodologia presentata in questo lavoro di tesi. Un mutante, invece, è un modello che contiene codice "dormiente" all'interno della normale descrizione del componente. Questo codice viene attivato quando si inietta un guasto, alterando l'operazione logica del componente stesso. Con questo metodo è possibile iniettare ad ogni livello di astrazione, ma l'utilizzo di mutanti richiede la sostituzione dei modelli esistenti con quelli modificati attraverso l'inserimento di codice dormiente. Nonostante il vantaggio della totale indipendenza dal simulatore adottato, le prestazioni sono potenzialmente molto basse a causa della possibile ricompilazione necessaria per ogni guasto;

- approccio basato sui comandi della simulazione modificati: i comandi di simulazione sono i comandi integrati nei simulatori, e permettono l'iniezione di un guasto e l'osservazione degli effetti. Questo approccio garantisce, in generale, prestazioni migliori, in quanto non richiede la modifica del codice che descrive il sistema, ma può essere adottato solo quando il codice dello strumento di simulazione è disponibile e facilmente modificabile, proprio

come nel *framework* presentato in Sezione 3.2.2. Possono essere identificate due tecniche basate sull'uso dei comandi del simulatore:

- manipolazione dei segnali: i guasti sono iniettati alterando il valore dei segnali usati per la comunicazione tra i componenti che compongono il modello ad alto livello. Questa tecnica si può implementare disconnettendo il segnale dal trasmettitore e forzandone il valore;
- manipolazione delle variabili: i guasti sono iniettati nei modelli comportamentali alterando i valori delle variabili definite nei processi del modello.

Riassumendo, i mutanti offrono la maggiore capacità di modellazione dei guasti tra le tecniche di iniezione proposte, mentre i *saboteurs* sono in genere meno potenti. La manipolazione dei segnali è adatta all'implementazione di semplici modelli di guasto e la manipolazione di variabili offre un modo semplice per iniettare guasti comportamentali.

L'*overhead* di tempo richiesto dalla manipolazione di segnali e variabili e dovuto solamente al controllo necessario all'iniezione dei guasti, poiché la simulazione deve essere interrotta e ripresa per ogni guasto iniettato. Al contrario, l'*overhead* generato dall'utilizzo di mutanti e *saboteurs* dipende dal numero di eventi addizionali generati, dall'ammontare di codice da eseguire per ogni evento e dalla complessità del controllo dell'iniezione.

Tra i vantaggi di SFI, sottolineiamo: supporto a tutti i livelli di astrazione di un sistema; pieno controllo dei modelli di guasto e dei meccanismi di iniezione; gli esperimenti di iniezione dei guasti sono effettuati su un sistema che esegue la propria applicazione *as-is*, senza modifiche, come ad esempio può richiedere, invece, SWIFI; osservabilità e controllabilità ottime: lavorando ad un livello di dettaglio sufficiente, è possibile corrompere qualsiasi segnale/valore si desidera nel modo che si desidera, ed è possibile osservare gli effetti di questa azione malgrado la

locazione del guasto; possibilità, infine, di modellare guasti transitori, permanenti e, come vedremo in Sezione 2.3.6, anche intermittenti.

Tra gli svantaggi, invece, bisogna principalmente sottolineare i costi in termini di tempo necessario alle simulazioni, sia per il *setup* del sistema, nel caso in cui sia necessario scrivere da zero le specifiche dei componenti in HDL, sia per l'esecuzione delle campagne di guasto. SFI, infatti, ha come requisito la disponibilità delle specifiche HDL dei modelli hardware, specifiche che non sempre sono disponibili.

Per concludere il discorso su SFI, vengono mostrati due ambienti che implementano questa strategia. ReSP, in particolare, è il *framework* che è stato scelto per lo sviluppo della metodologia oggetto di questa tesi. SyFI, invece, è una soluzione che presenta scelte innovative e che ci sembra giusto, quindi, trattare.

2.2.3.1 SyFI

Sviluppata alla Korea Aerospace University, *SystemC kernel-based Fault Injection* (SyFI) ([23]) è una piattaforma per la simulazione di modelli e per l'iniezione di guasti che punta alle prestazioni e all'assoluta non intrusività all'interno del sistema in analisi.

SyFI si basa su un kernel SystemC che è in grado di fornire tutti i servizi necessari all'iniezione di guasti. Dato che il codice sorgente non viene modificato, tutti gli esperimenti possono essere condotti in modo più veloce ed efficiente. L'ambiente SyFI ha lo scopo di valutare l'affidabilità del SoC considerato, e, nel farlo, esegue tre operazioni:

- **Setup**: la prima fase utilizza tre diversi input, che sono il modello del sistema in analisi, il software eseguito su di esso e il profilo della campagna di iniezione dei guasti. Il profilo viene descritto all'interno di un file che contiene cinque attributi: tipo di guasto, istante e locazione di iniezione, modello del guasto e intervallo. Per guasti permanenti e transitori, SyFI supporta istanti e locazioni sia deterministiche che random;

- **Esecuzione**: una volta caratterizzato il sistema e il tipo di campagna, è possibile eseguire le iniezioni. SyFI crea un file *Value Change Dump* (VCD) che contiene la "manifestazione" dei guasti iniettati. Queste informazioni sono passate all'analizzatore dei log per l'ultima fase. Per ottenere risultati statisticamente accettabili, sono necessarie lunghe campagne con decine di migliaia di simulazioni;

- **Analisi**: l'ultima fase viene svolta dall'analizzatore di *log*. Esso si occupa, nell'ordine, di confrontare ognuno dei file VCD e dei file di *dump* della memoria della simulazione in corso con quelli *golden*, cioè della simulazione senza iniezione di guasti, precedentemente generati; determina il tipo di fallimento in base a una classificazione determinata, che comprende, ad esempio, *Silent Data Corruption* (SDC) oppure *Detected Unrecoverable Error* (DEC); calcola, infine, la percentuale di guasti individuati all'interno del modello.

In Figura 2.3 vengono mostrate nel dettaglio le tre fasi all'interno dell'ambiente di SyFI. L'immagine mostra un ambiente per l'iniezione di guasti dotato di elementi per la caratterizzazione (o setup) del modello e per l'analisi. Idealmente, questo ambiente rappresenta una base adatta per lo sviluppo della metodologia presentata nel Capitolo 3. Le caratteristiche che mancano a questa soluzione, e che ci hanno poi fatto propendere per utilizzare la piattaforma di iniezione descritta nella prossima *embedded*, sono la mancanza di scelta dell'applicazione da eseguire sull'architettura simulata (è possibile eseguire solo *benchmark* predefiniti) e la mancanza di un *framework* flessibile per svolgere un'analisi del sistema accurata e che tenga conto dell'applicazione che viene eseguita.

Nonostante la presenza di un modulo (analizzatore di *log*) che ha il compito di confrontare le tracce della simulazione del modello guasto rispetto alla simulazione in assenza di guasto, SyFI soffre la mancanza di un vero e proprio *framework* che si occupi di effettuare un analisi approfondita dello stato interno del sistema. Per questo motivo, durante la scelta dell'ambiente ideale da utilizzare per raggiungere

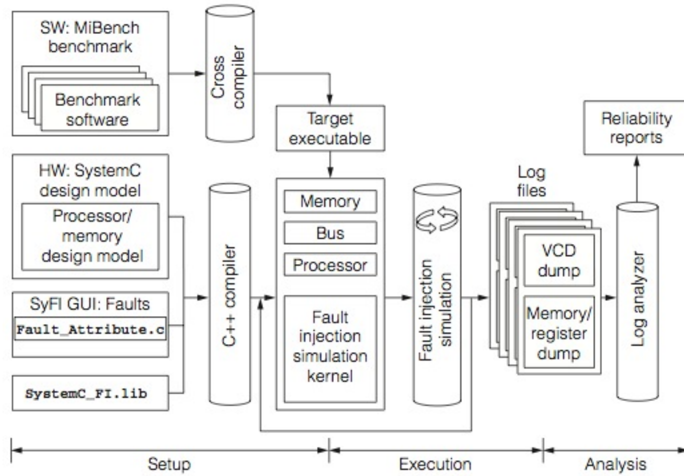


Figura 2.3: *Framework di SyFI.*

gli obiettivi che ci siamo posti nello sviluppo del nostro approccio, non abbiamo preso in considerazione questa piattaforma.

2.2.3.2 ReSP

Reflective Simulation Platform (ReSP), presentato in [9, 10] e discusso in modo più approfondito in Sezione 3.2.2, è una piattaforma di simulazione hw/sw basata su specifiche di sistema descritte attraverso il linguaggio SystemC ([2]) e le librerie Transaction-Level Modeling (TLM, [12]). Il nucleo del simulatore è il *kernel* SystemC, che fornisce i mezzi per simulare le specifiche SystemC TLM. Al di sopra di esso, è stata implementata la piattaforma di simulazione scritta in *Python*, per sfruttare le capacità riflessive del linguaggio; queste capacità permettono un accesso non intrusivo a *run-time* della struttura interna dei componenti del modello. Questa caratteristica, unita al fatto che *Python* è un linguaggio di scripting, consente la composizione a *run-time* del sistema (non è necessario compilare file addizionali per i componenti) e la gestione dinamica del sistema in analisi (osservando e modificando gli stati interni dei componenti).

ReSP è stato esteso con un motore per l'iniezione dei guasti per ottenere un ambiente ideale all'iniezione e all'analisi di suscettibilità. Lo strumento offre, inol-

tre, mezzi per automatizzare l'estrazione di possibili locazioni nelle quali iniettare guasti, la generazione di liste di guasto e l'esecuzione degli esperimenti. Le attività base di iniezione prevedono la corruzione dello stato di determinati componenti oppure dei dati trasmessi tra essi.

ReSP ha le caratteristiche necessarie per essere utilizzato come *framework* per l'applicazione della metodologia presentata nel Capitolo 3, poiché consente un'analisi a livello sia hardware che software e fornisce una piattaforma, per la simulazione e l'iniezione, potente e flessibile.

2.2.4 *Emulation-based Fault Injection*

Emulation-based Fault Injection (EFI) è considerata come il modo migliore per ridurre il tempo necessario all'esecuzione di un'intera campagna di iniezione dei guasti su un sistema complesso con un gran numero di vettori di test, contesto nel quale SFI mostra i propri limiti in modo evidente.

EFI utilizza una piattaforma hardware programmabile, come ad esempio una *Field Programmable Gate Array* (FPGA), per ospitare il sistema in analisi. La *netlist* del circuito viene generata a partire dalle risorse interne della FPGA, e viene poi scaricata direttamente all'interno della scheda, dove l'emulazione del circuito può essere eseguita a velocità molto maggiori rispetto alle piattaforme di simulazione descritte precedentemente. Sfortunatamente, gli emulatori di sistemi soffrono di osservabilità dello stato e controllabilità dell'iniezione limitate. Nell'implementazione più semplice di EFI, il circuito in analisi può essere controllato solo attraverso i propri segnali di input primari e il comportamento può essere solamente osservato sui segnali di output primari.

A causa di questa limitazione, sono state proposte soluzioni differenti per aumentare la controllabilità e l'osservabilità degli emulatori. Molte di esse prevedono la modifica del circuito in analisi attraverso l'introduzione di ulteriore hardware che ha il compito di cambiare lo stato dei registri interni durante la campagna di iniezione dei guasti e, opzionalmente, è in grado di tracciare la

propagazione dei guasti. I metodi che sfruttano questo approccio sono basati sull'"instrumentazione" del circuito, cioè sull'introduzione di nuovo hardware come abbiamo appena discusso. L'instrumentazione del circuito è un metodo efficace per aumentare la controllabilità degli emulatori e garantisce prestazioni molto migliori, soprattutto rispetto alle strategie basate su simulazione. Questa soluzione, però, è fortemente intrusiva, poiché richiede la modifica della specifica HDL del sistema che deve essere emulato, generando un modello per i test che è diverso rispetto a come era stato progettato in origine.

Un altro approccio per aumentare la controllabilità e l'osservabilità dei sistemi che si basano su EFI consiste nello sfruttamento delle proprietà di riconfigurazione totale o parziale delle piattaforme di emulazione. Questo approccio, chiamato *run-time reconfiguration emulation-based fault injection* (RTR in breve), invece di iniettare guasti attraverso l'uso di logica addizionale inserita nel circuito, consente di utilizzare le proprietà di riconfigurazione per iniettare guasti. Questo significa che devono essere effettuate riconfigurazioni a *run-time* dell'emulatore per ogni guasto iniettato; nonostante ciò, questo evita l'impiego di tempo per generare le versioni instrumentate. La modifica della configurazione necessaria alla riconfigurazione è un procedimento molto semplice se confrontato, ad esempio, con la modifica della descrizione HDL del circuito. Inoltre, il tempo di riconfigurazione speso globalmente dall'emulatore hardware può essere ridotto attraverso una parziale riconfigurazione preliminare svolta prima dell'inizio dell'iniezione. L'iniziale descrizione ad alto livello viene sintetizzata, posizionata e collegata e viene generato un file di configurazione che corrisponde al circuito in esame senza nessun elemento addizionale. Il file generato viene inserito nell'emulatore e la campagna di iniezione inizia dall'esecuzione di un carico di lavoro predefinito (chiamato anche *test bench*) sul prototipo implementato. Il risultato dell'esecuzione viene usato in seguito come riferimento per l'analisi degli effetti dei guasti. Poi, lo stesso carico di lavoro viene rieseguito tante volte quante sono i guasti da iniettare.

RTR sceglie di iniettare i guasti ad un livello "basso", direttamente nell'hardware riconfigurabile, attraverso la modifica del design precedentemente implementato sull'emulatore. Così, ogni iniezione di guasti può essere realizzata senza necessità di cambiare la descrizione iniziale e senza aggiungere altro hardware. Nonostante la riconfigurazione necessaria per ogni guasto da iniettare, i risparmi in termini di tempo, rispetto alle altre tecniche, sono notevoli, soprattutto nei confronti di SFI.

In conclusione, presentiamo i principali vantaggi di questa strategia: il tempo di esecuzione degli esperimenti di iniezione è inferiore se confrontato con le tecniche SFI; particolarmente interessante in un contesto di progettazione di SoC, dato che può portare ad analisi di affidabilità efficienti e a basso costo, prima che essi siano implementati in un circuito; il tempo di simulazione può essere ulteriormente ridotto implementando parzialmente o totalmente la generazione dei *pattern* di input all'interno dell'FPGA, visto che sono noti quando il circuito da analizzare viene sintetizzato.

I principali svantaggi, invece, sono descritti nel seguito: la descrizione iniziale del circuito deve essere sintetizzabile e ottimizzabile, per evitare la necessità di un emulatore troppo grande o troppo costoso e per ridurre i tempi di esecuzione delle campagne; il costo di un generico hardware per l'emulazione è elevato, così come la complessità di una FPGA dedicata; l'emulazione può essere utilizzata solo per l'analisi funzionale delle conseguenze di un guasto, non per gli impatti in termini temporali che esso può avere sul sistema.

EFI è una strategia trattata in modo decisamente ampio in letteratura. Nel contesto di sviluppo della metodologia proposta in questa tesi, però, le innovazioni e le scelte attuate in questo ambito sono di relativo interesse, poiché l'approccio che abbiamo sviluppato è supportato dalla strategia di *Simulation-based Fault Injection*. Rimandiamo il lettore alla visione, a scopo di approfondimento, dei seguenti articoli, che discutono l'applicabilità e le potenzialità dell'iniezione basata

su emulazione: [5], [16], [17], [29], [36].

2.2.5 Considerazioni finali sulle strategie e sugli ambienti di iniezione di guasti

In questa prima parte del capitolo di stato dell'arte, abbiamo fornito al lettore una panoramica sulle strategie di iniezione dei guasti che sono sfruttate per la verifica dell'affidabilità di sistemi, reali o simulati; accanto ad esse, abbiamo fornito esempi progettati per utilizzare le strategie descritte. In questa sezione, vogliamo discutere le principali limitazioni che caratterizzano i lavori proposti.

Quello che manca, in generale, agli ambienti per l'iniezione dei guasti trattati, è un *framework* o, quantomeno, un modulo, all'interno dell'ambiente di iniezione, in grado di svolgere un'analisi approfondita degli effetti di un guasto sul sistema considerato.

Nelle Sezioni 2.2.2.1 e 2.2.2.3, nell'ambito della strategia SWIFI, gli ambienti presentati integrano un semplice modulo di analisi che si limita, nel caso di Xception, a stabilire se il guasto ha causato la chiamata a un'eccezione all'interno dell'applicazione; nel caso di GOOFI, invece, la strategia di analisi prevede il calcolo separato degli errori individuati direttamente dal sistema e quelli individuati dal *framework* di GOOFI. Questa suddivisione risulta essere interessante, poiché si è in grado di stabilire quali errori sono automaticamente gestiti dal sistema considerato e quali, invece, vengono osservati dallo strumento che si occupa del monitoraggio. OCD-FI, invece, introdotto in Sezione 2.2.2.4, presenta una strategia per l'iniezione di guasti che sfrutta il modulo di *debugging* presente all'interno del processore nel sistema simulato. L'iniezione e l'analisi sono entrambe effettuate a livello dell'architettura, mentre sarebbe interessante sfruttare una tecnica simile al *debugging* per effettuare l'analisi a livello di applicazione. In generale, gli ambienti analizzati che adottano la strategia SWIFI, valutano l'impatto del guasto sull'applicazione eseguita nel sistema, in termini di segnalazione di un er-

rore da parte dell'applicazione stessa, ma non si occupano di analizzare in modo accurato in che modo l'errore viene propagato all'interno del livello software.

Passando alla strategia di iniezione basata su simulazione (SFI), oltre all'introduzione alla piattaforma che è stata scelta come ambiente di supporto allo sviluppo di E-SWEAM, abbiamo analizzato due ambienti che presentano caratteristiche degne di nota. *SystemC kernel-based Fault Injection*, presentato in Sezione 2.2.3.1, è dotato di un modulo per l'analisi che si occupa di analizzare le tracce dell'esecuzione dell'applicazione, ma si limita al confronto con la controparte *Golden* senza scendere nel dettaglio.

Gli ambienti che implementano EFI, infine, presentano soluzioni interessanti dal punto di vista delle prestazioni durante l'esecuzione degli esperimenti di iniezione, ma, in generale, non sono dotati di strumenti efficaci per l'analisi approfondita del comportamento del circuito emulato. Queste considerazioni, unite al fatto che gli ambienti basati su EFI richiedono l'utilizzo di una piattaforma complessa per l'emulazione, che richiede un dispositivo (ad esempio, una FPGA) collegato ad un computer ospite, ci hanno fatto propendere per la scelta di una strategia basata su simulazione.

2.3 Tecniche di analisi dell'affidabilità dei sistemi dedicati

La seconda parte del capitolo di stato dell'arte è dedicata alla trattazione di alcuni metodi di analisi di affidabilità dei sistemi dedicati presenti in letteratura. Vengono introdotte, nell'ordine: *Failure Modes & Effects Analysis* (FMEA), *Architectural Vulnerability Factor* (AVF), insieme alle due varianti *Program Vulnerability Factor* (PVF) e *Thread Vulnerability Factor* (TVF), una metodologia di analisi che sfrutta l'approccio PVF (PRASE) e, infine, una metodologia di diagnosi di guasto che nonostante non rientri nell'ambito delle metodologie di analisi che consideriamo, contiene delle idee interessanti che, per una maggiore chiarezza,

vogliamo trattare.

2.3.1 FMEA

Failure Modes & Effects Analysis (FMEA) è una metodologia utilizzata per analizzare le modalità di guasto o di difetto di un sistema. Le classi di fallimento potenziali sono classificate in base alla gravità che hanno sul comportamento del sistema e in base alla frequenza con la quale possono presentarsi. L'applicazione di FMEA a un sistema dedicato aiuta il progettista a identificare le possibili classi di fallimento in base all'esperienza passata con prodotti simili, consentendo l'identificazione di quelle che possono essere le classi più critiche con il minimo sforzo e con l'impiego di un numero limitato di risorse, riducendo, in questo modo, tempi e costi di progettazione. FMEA viene ampiamente utilizzata dalle industrie durante diverse fasi del ciclo di vita di un prodotto. Generalmente, comunque, l'analisi di sistema che viene effettuata da FMEA è statica e viene eseguita preventivamente, basandosi, quindi, su considerazioni teoriche e non sperimentali.

In [14], gli autori presentano questa metodologia applicata a sistemi basati su *System-on-Chip* (SoC) modellati con SystemC TLM, con lo scopo di valutare il rischio e l'impatto dei fallimenti sul comportamento dell'architettura in analisi, sfruttando la tecnica di SFI per validare l'intero approccio. Come motore di simulazione, è stato scelto *CoWare Platform Architect*.

La premessa con la quale introducono il loro lavoro consiste nella constatazione che, a causa della complessità crescente e dei costi elevati nei quali possono incorrere i produttori di sistemi *embedded*, sia necessario avere a disposizione una metodologia che possa essere adottata su modelli di sistemi non ancora realizzati e fisicamente "testabili", per poter fornire una valutazione efficace dell'affidabilità del componente e non incorrere nei costi che si dovrebbero sostenere in caso di una riprogettazione totale. Per questo motivo, e per ottenere prestazioni accettabili durante le simulazioni, gli autori hanno deciso di "alzare" il livello al quale viene

simulato il sistema a livello *Transaction-Level Modeling*. Queste considerazioni fondamentali sono state di ispirazione, in parte, anche per il nostro lavoro.

In questo studio viene proposto un modello di rischio che combina FMEA con la tecnica di iniezione dei guasti, allo scopo di identificare e valutare i potenziali comportamenti anomali di un SoC modellato con SystemC TLM, e di misurare i gradi di rischio delle conseguenze risultanti dai possibili fallimenti. Se i requisiti di affidabilità non vengono soddisfatti, i risultati di FMEA saranno sfruttati per sviluppare un processo di riduzione del rischio efficiente.

Il modello di rischio proposto si basa sul calcolo e sulla valutazione empirica di parametri come le probabilità di fallimento e il costo dell'impatto di una classe di fallimento sul sistema. Lo scopo di questo modello è il calcolo del *Risk Priority Number* (RPN), cioè del valore di soglia che specifica il tipo di azione correttiva da intraprendere nei confronti del componente in esame (del quale un calcolo alternativo interessante è presente in [34]). RPN viene solitamente calcolato nel modo seguente:

$$RPN = Severity * Occurrence * Detection$$

Per ottenerlo sono quindi necessari i parametri relativi alla frequenza, al costo che il fallimento causa sul componente e alla probabilità di individuazione. Gli autori caratterizzano, quindi, il sistema in analisi per estrarre queste informazioni, e si servono dell'iniezione di guasti per ottenere valori statisticamente validi.

La classificazione dei fallimenti proposta in questo approccio è suddivisa in quattro classi:

- *Fatal Failure* (FF): crash del sistema o *hang* dell'applicazione;
- *Silent Data Corruption* (SDC)
- *Correct Data / Incorrect Time* (CD/IT)
- *Deadlock* (DL)

In realtà, viene lasciato spazio ad altre classi di fallimento, ma gli autori si concentrano su queste quattro perché ritengono che siano quelle che, con più probabilità, possono affliggere il loro sistema.

Lo studio proposto in [14] ha alcuni punti di contatto con il nostro lavoro. In particolare, abbiamo trovato molto interessanti la scelta di lavorare ad un livello di astrazione alto (TLM) e la scelta di classificare il comportamento del sistema in base al comportamento dell'applicazione che viene eseguita. Questi concetti sono ripresi in Sezione 3.1 ed ampliati. La metodologia presentata in questo articolo mostra i propri limiti nel tipo di analisi di sistema che viene applicata. Durante la fase iniziale di analisi statica del modello, vengono definite le classi di fallimento. L'iniezione di guasti viene utilizzata per capire la frequenza con cui esse si manifestano. In questo modo, gli autori raccolgono dati statistici relativi alla probabilità di guasto di ogni componente che testano, e utilizzano questi dati per fare l'analisi di criticità che si ottiene dal calcolo degli RPN e della percentuale di guasti individuati. La classificazione, in sostanza, viene generata sulla base dei soli risultati finali nei quali il sistema si porta. Non esistono strumenti o strategie per l'analisi dello stato interno intermedio dei componenti dell'architettura in analisi.

2.3.2 Metodologia FMEA per la certificazione di un circuito in conformità alla norma IEC61508

In [30], Mariani et al. presentano una metodologia basata su FMEA per la validazione, in conformità con lo standard IEC61508 ([18, 8]), di un circuito descritto a livello *System-on-Chip* (SoC). La metodologia utilizza un simulatore di livello *Register Transfer Level* (RTL) e un iniettore di guasti. Il simulatore è dotato di uno strumento in grado di analizzare nel dettaglio il circuito in analisi, per estrarre informazioni dettagliate a livello RTL e permettere l'iniezione in zone sensibili dell'architettura.

Il principale contributo dell'articolo consiste nella presentazione di un approccio sistematico, supportato dal *framework* utilizzato per la simulazione e l'iniezione, per l'estrazione di zone sensibili del circuito direttamente dalla descrizione RTL dello stesso e l'applicazione di FMEA.

Il primo passo prevede l'estrazione di tutte le possibili zone sensibili, o *Sensible Zones* (SZ), dalla descrizione RTL del circuito. Una SZ è un punto di fallimento del SoC dove uno o più guasti possono portare alla deviazione dal comportamento atteso. Definizioni valide per SZ possono essere: elementi di memoria (registri, *flip-flop*, etc.), input e output primari del SoC, entità logiche mappate su un elemento di memoria (ad esempio, condizione errata in un'istruzione condizionale), oppure interi sotto-blocchi del circuito, scelti per prendere in considerazione un determinato sotto-insieme di output, ad esempio.

Un altro elemento importante per l'utilizzo di FMEA è rappresentato dai punti di osservazione, o *Observation Points* (OP), che sono le locazioni del circuito nelle quali si misurano gli effetti di un fallimento di una SZ. OP validi possono essere: SZ a valle di una determinata SZ, un'uscita primaria (nella maggior parte dei casi), oppure una funzione primaria del SoC (se l'analisi viene effettuata ad un livello di astrazione più alto).

Le classi di fallimento, in inglese *Failure Modes* (FM), considerati per FMEA possono essere di due tipi: direttamente collegate a guasti fisici, come ad esempio un *bit-flip* causato da un guasto nella circuiteria di un elemento di memoria, oppure possono essere la somma di un determinato numero di eventi di guasto, come ad esempio guasti multipli che colpiscono lo stesso elemento di memoria considerato come SZ.

Per ogni *gate* e registro del circuito estratto dalla descrizione RTL, partendo dal parametro base *Failure in Time*, viene calcolata la frequenza di fallimento. Questo dato viene inserito in un foglio di calcolo, fornito con la metodologia, al quale devono essere forniti altri parametri fondamentali, come i fattori *Safe*

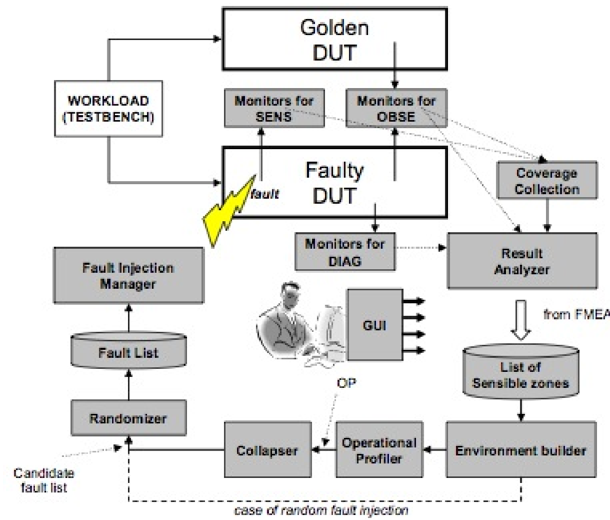


Figura 2.4: *Panoramica del framework per la validazione di FMEA a livello SoC.*

fraction e *Dangerous fraction* dei possibili fallimenti per ogni SZ. Tutti questi dati sono necessari all'esecuzione di FMEA sul modello del circuito.

Per concludere il discorso, vogliamo presentare brevemente il *framework*, utilizzato nell'articolo, per la validazione di FMEA e necessario all'iniezione di guasti nel circuito simulato, come richiesto dallo standard IEC61508. Il *framework* è mostrato in Figura 2.4.

Lo strumento è composto da:

- *Environment Builder* (EB): questo blocco estrae da FMEA le informazioni, relative al circuito, per l'esecuzione della campagna di iniezione;

- *Operational Profiler, Collapser and Randomiser*: partendo dalle informazioni estratte dall'EB, questo blocco estrae le informazioni relative al profilo operazionale, in inglese *Operational Profile* (OP), di un dato carico di lavoro. OP contiene una collezione di dati che riguardano tutte le attività del sistema in assenza di guasto, e il suo scopo è di caratterizzare al meglio il sistema nel suo complesso, per essere in grado di assicurare che solo i guasti che produrranno un errore siano selezionati durante il processo di generazione della lista di guasti;

- *Fault Injection Manager*: questo blocco si occupa dell'esecuzione della campagna di iniezione dei guasti e della raccolta dei risultati;
- *Result Analyzer*: questo strumento colleziona i risultati della campagna e riempie in automatico un foglio di calcolo necessario alla validazione di FMEA. Inoltre, si occupa anche di estrarre una tabella degli effetti per ogni SZ, cioè una tabella degli OP che hanno mostrato una deviazione rispetto alla simulazione in assenza di guasto;
- *Monitors and Coverage Collection*: questo blocco, composto da un insieme di monitor istanziati dall'EB, genera e raccoglie le informazioni necessarie al calcolo dei parametri come la percentuale di copertura dei guasti.

Il lavoro presentato in questo articolo offre delle soluzioni interessanti per l'analisi approfondita a livello architetturale di un circuito. In particolare, sono risultate utili anche nel nostro lavoro le definizioni di SZ e di OP. Inoltre, il *framework* qui presentato rispecchia i requisiti necessari al simulatore che erano richiesti per lo sviluppo della nostra metodologia. In generale, quindi, l'unico aspetto critico di questo approccio rappresenta il livello di analisi scelto, solamente architetturale. Sono presenti cenni, all'interno dell'articolo, relativi ad un'analisi a livello applicativo, ma l'analisi e la certificazione finale riguardano solo il circuito a livello SoC.

2.3.3 AVF

In [32], Mukherjee et al. introducono il concetto di *Architectural Vulnerability Factor* (AVF) come la probabilità che un guasto all'interno di una struttura di un componente causi un errore. Per calcolare questa quantità, è necessario conoscere la percentuale di errori che possono avvenire in quella determinata struttura. Lo studio si concentra sul confronto degli AVF di diverse strutture di un processore.

Per generare una stima valida delle percentuali di errore, è necessario comprendere quali siano le strutture più suscettibili all'iniezione di un guasto. Non tutte,

infatti, influiscono sul risultato finale del programma eseguito. Una stima basata solo sulla percentuale di guasto di un componente è oltremodo pessimistica, in quanto solo una percentuale di questi guasti si traduce in un errore del sistema. Ad esempio, il guasto di un bit nel *branch predictor* non influisce sulle istruzioni eseguite. AVF viene quindi definito in modo più preciso come la probabilità che un guasto in una struttura del processore risulti in un errore visibile nell'output del programma. Dagli studi effettuati, se il *branch predictor* ha un AVF dello 0%, un guasto nel *program counter* ha un AVF che si avvicina al 100%. Per le altre strutture, AVF oscilla tra questi due valori.

L'articolo propone una stima dell'AVF utilizzando un approccio che individua il sottoinsieme di bit di stato del processore richiesti per un'esecuzione architetturalmente corretta, in inglese *architectural correct execution* (ACE). Ogni guasto in una cella o registro che contenga uno di questi bit, chiamati ACE bits, può causare un errore visibile nell'output finale del programma, in assenza di tecniche per la correzione degli errori. Al contrario, un guasto che affligge solo *un-ACE bits* non causa un errore. L'ipotesi di partenza è che ogni cella o registro abbia la stessa percentuale di guasto e che l'AVF di una struttura sia la media degli AVF delle celle che la compongono.

La metodologia per il calcolo di AVF consente di analizzare in modo molto approfondito lo stato interno dell'architettura durante l'esecuzione. In particolare, questo approccio di analisi è in grado di individuare i componenti più suscettibili ai guasti presenti nell'architettura considerata. Il tipo di analisi effettuata ha lo scopo di segnalare la vulnerabilità di ogni componente del processore, in riferimento al tipo di errore che un guasto all'interno del componente causa nell'output del programma eseguito. L'analisi, quindi, è unicamente a livello architetturale, con l'applicazione che serve unicamente da discriminante per stabilire se un guasto causa un errore oppure no. Sarebbe interessante poter considerare il parametro di AVF in relazione al tipo di errore che si manifesta nel software, per consenti-

re di svolgere un'analisi approfondita di propagazione degli effetti all'interno del sistema.

2.3.4 PVF e TVF

In [42, 35] vengono presentati i concetti di *Program Vulnerability Factor* (PVF) e di *Thread Vulnerability Factor* (TVF), che cercano di sfruttare i meccanismi ideati per il calcolo di AVF, appena discusso, per analizzare il sistema a livello dell'applicazione che viene eseguita. Nel caso del TVF, si considerano applicazioni complesse che eseguono in parallelo su più *threads*.

Sridharan e Kaeli propongono l'utilizzo di tracce PVF per accelerare la stima dell'AVF. La vulnerabilità di un sistema ai guasti transitori può essere quantificata misurando l'AVF di ogni struttura del sistema. La *System Vulnerability Stack* (SVS) è un metodo per il calcolo della vulnerabilità di un sistema a diverse classi di guasto, e osserva che un sistema è composto da livelli indipendenti che interagiscono attraverso interfacce ben definite. SVS suddivide la vulnerabilità nei singoli componenti di ogni livello del sistema e, per esempio, definisce l'*Hardware Vulnerability Factor* (HVF) come il livello di vulnerabilità del microprocessore, il PVF come il livello di vulnerabilità del software. Il mascheramento di un guasto può così essere quantificato all'interno di un livello analizzando le sue interfacce: un guasto che non si propaga fino alle interfacce può essere considerato mascherato. Nell'articolo si propone la stima, durante la fase di progettazione di un sistema, del suo AVF attraverso l'utilizzo delle tracce PVF ottenute dall'esecuzione di programmi *benchmark*. Le tracce salvate sono solo quelle segnalate da una prima analisi ACE.

Nonostante l'interessante sfruttamento delle tracce di programma per il calcolo dell'AVF, questo metodo viene ridimensionato nella sua efficacia dal livello di granularità scelto. Le tracce, infatti, sono a granularità di byte. Lo studio, quindi, offre vantaggi in termini di velocità nel calcolo dell'AVF, ma lo strumento in più a disposizione (le tracce) non viene utilizzato per un'analisi più approfondita, ad

esempio a livello del flusso di programma. In questo modo si sarebbero potute ottenere informazioni dettagliate sul comportamento dell'applicazione all'interno di un contesto nel quale l'hardware sottostante è malfunzionante a causa di un guasto.

Lo stesso discorso si può estendere all'articolo che tratta lo studio del TVF. A livello applicativo, si va ancora più in profondità analizzando il comportamento del software suddiviso in *threads* ed eseguito su più processori, per valutare l'impatto che esecuzioni parallele possono avere su ogni struttura dell'architettura. In realtà, però, mancano informazioni interessanti che si sarebbero potute ottenere semplicemente analizzando l'esecuzione a run-time del flusso di chiamate a funzione piuttosto che di ogni singola istruzione *assembly*.

2.3.5 PRASE

In [44], Xu et al. discutono l'applicabilità di un approccio, chiamato *Program Reliability Analysis with Soft Errors* (PRASE), per l'analisi dell'affidabilità di un programma sotto effetto di errori transitori. L'analisi è basata su semplici nozioni di teoria della probabilità e sul codice *assembly* del programma, e mira a determinare la generazione e la propagazione degli errori. Per quantificare la vulnerabilità del programma, gli autori sfruttano il fattore PVF precedentemente descritto in Sezione 2.3.4.

Nell'articolo, vengono considerati i soli errori transitori che affliggono strutture di salvataggio dei dati, come memorie, *cache*, registri. Questo tipo di errori vengono detti *raw soft errors* (RSE). Dato che un RSE si riferisce ad una cella a singolo bit, l'analisi viene svolta in base alla codifica del codice *assembly* del programma. Nel lavoro proposto, viene adottato un *instruction set* basato su MIPS 32. Per semplicità, si assume che ogni istruzione viene eseguita in un ciclo ed è isolata rispetto all'hardware sottostante. L'approccio PRASE è composto da diverse fasi eseguite strettamente in ordine, che sono: un'analisi probabilistica delle generazione di RSE, un'analisi della propagazione di RSE, un'ottimizzazione

che sfrutta l'analisi dei blocchi basici e infine un metodo per calcolare l'affidabilità totale del programma.

Dato che l'analisi di propagazione degli errori istruzione per istruzione diventa molto complessa per programmi di certe dimensioni e, dall'altra parte, osservare la propagazione degli errori in ogni istruzione è un approccio limitato che non consente di capire in che modo il fallimento impatta sull'esecuzione, gli autori propongono un metodo di ottimizzazione basato sull'analisi della propagazione degli errori all'interno dei blocchi basici di istruzioni. I blocchi basici possono essere associati ad un *control flow graph* (CFG) composto da nodi che rappresentano i blocchi basici e da archi che rappresentano tutti i possibili flussi di esecuzione. Attraverso questo strumento, l'analisi statica effettuata da PRASE risulta più efficace e meno complessa, e consente di definire in modo più accurato l'impatto degli errori sul flusso di esecuzione. Ad ogni blocco basico di istruzioni del CFG è associato un *data dependency graph* (DDG) che modella l'utilizzo delle variabili all'interno del blocco basico stesso.

Gli autori si servono di un ambiente di simulazione per iniettare guasti all'interno di alcune applicazioni *benchmark* e confrontare i risultati ottenuti con PRASE con quelli ottenuti dall'esecuzione della campagna di iniezione.

Al di là delle considerazioni sulla validità del metodo proposto che, dai risultati degli esperimenti di iniezione dei guasti proposti nell'articolo, si comporta bene con alcune applicazioni, mentre fornisce delle stime molto diverse per altre, il contributo di questo articolo allo sviluppo di E-SWEAM consiste nell'utilizzo dei CFG e dei DDG per la modellazione del comportamento dell'applicazione eseguita. Gli autori non chiariscono, però, come vengano costruiti sia i CFG che i DDG per le applicazioni sulle quali hanno testato il proprio metodo. Per applicazioni complesse, la costruzione dei due grafi può richiedere molto tempo e notevoli risorse computazionali. Inoltre, questi modelli sono complessi da implementare su applicazioni reali.

Nel Capitolo 3, presenteremo il nostro strumento per la modellazione delle applicazioni che si ispira ai CFG e ai DDG qui presentati. Mostreremo, inoltre, un metodo efficace e non troppo dispendioso per generarlo a partire dalle tracce salvate durante l'esecuzione dell'applicazione.

2.3.6 Un modello di propagazione di guasti intermittenti all'interno di programmi

Rashid et al. ([39]) presentano una nuova metodologia che sfrutta un simulatore di livello microarchitetturale per valutare l'impatto di guasti di tipo intermittente a livello di applicazione.

Per creare un modello adatto alla rappresentazione di guasti intermittenti, gli autori si sono focalizzati su un insieme di guasti che affliggono istruzioni consecutive durante l'esecuzione di un programma. In questo modello, il guasto intermittente può essere rappresentato dall'istruzione in esecuzione quando il guasto viene iniettato e dal numero di istruzioni durante le quali il guasto persiste. Il modello rappresenta diversi tipi di guasto, come ad esempio errori di decodifica delle istruzioni, errori dell'ALU e errori delle unità di *load/store*. Si ipotizza, inoltre, che le memorie e la logica di controllo del processore siano coerenti e affidabili. Negli studi effettuati per l'articolo, l'impatto di guasti intermittenti, quando non ha un effetto benigno (cioè di nessun errore) sull'applicazione, causa nella maggior parte dei casi un *crash* della stessa. Per questo è stato anche creato un modello del tipo di crash che può affliggere l'applicazione, suddividendo i casi nei quali l'esecuzione continua da quelli nei quali si verifica un evento di tipo *fail/stop*.

Per studiare la propagazione dell'errore all'interno dell'applicazione, gli autori hanno sfruttato uno strumento che modella a livello di istruzione le dipendenze del programma, chiamata *Dynamic Dependencies Graph* (DDG); il DDG, del quale mostriamo un esempio in Figura 2.5, è un grafo diretto aciclico che modella le dipendenze sui dati tra i *task* di un programma, e si può ricavare dalla traccia delle istruzioni eseguite dall'applicazione in assenza di guasto. Questo modello è

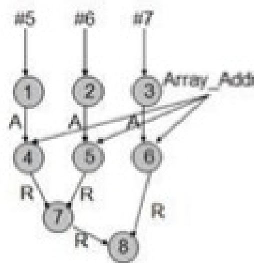


Figura 2.5: *Esempio di DDG presentato in [39].*

stato sfruttato per ridurre il numero di campagne di iniezione dei guasti necessarie all'ottenimento di dati statistici significativi, che si sono rivelate molto dispendiose in termini di potenza di elaborazione e di tempo.

Il lavoro proposto in [39] ha offerto spunti significativi per la stesura della nostra metodologia. In particolare, l'analisi a livello applicazione ottenuta con l'ausilio del DDG è un buon punto di partenza per il tipo di analisi approfondita che abbiamo cercato di offrire con E-SWEAM (si veda 3.1.5). Oltre a ciò, gli autori offrono una prospettiva interessante concentrando il proprio lavoro sullo studio dell'impatto di guasti intermittenti, che non sono mai stati studiati in modo approfondito a causa della difficoltà intrinseca della loro modellizzazione, a livello di applicazione. Lo studio si limita, però, a rilevare che questo tipo di guasti, se non mascherati a livello applicazione, possono causare un *crash* della stessa o fenomeni di *silent data corruption*, senza entrare nello specifico delle conseguenze che questi comportamenti anomali hanno effettivamente sul sistema.

2.3.7 SWAT

Per concludere il discorso relativo agli approcci per l'analisi, a livello software, di sistemi complessi, è doveroso citare gli articoli di Li, Ramachandran et al. ([27, 26, 25, 22, 24]) che presentano la metodologia *SoftWare Anomaly Treatment* (SWAT) per sistemi PC basati su architetture *multi-core*; SWAT sfrutta l'iniezione dei

guasti a livello micro-architetturale per effettuare la diagnostica dei componenti guasti all'interno dell'architettura.

Questi articoli descrivono un approccio che ha lo scopo di diagnosticare, all'interno di un sistema composto da un'architettura multiprocessore, il tipo di guasto occorso a livello hardware. Per ipotesi, un *core* all'interno dell'architettura sarà sempre funzionante e privo di guasti (questa assunzione è necessaria per avere un elemento *Golden* da confrontare con gli elementi nei quali verranno iniettati i guasti). L'applicazione *benchmark* eseguita nel sistema è dotata di diversi *checkpoint*. SWAT sfrutta la tecnica di *rollback/replay* per interrompere la simulazione nel momento in cui viene individuato un sintomo software di errore. I monitor software del *framework* sono in grado di rilevare i sintomi di un malfunzionamento, rappresentati da chiamate a *trap* hardware, da chiamate a eccezioni software, oppure da *deadlock* dell'applicazione. Una volta rilevato il sintomo, viene effettuato un *rollback* dell'esecuzione fino al *checkpoint* precedente. La fase di *replay* viene eseguita sul *core* privo di guasti. Questa riesecuzione è in grado di diagnosticare il tipo di guasto avvenuto nel sistema (transitorio, permanente, o software).

SWAT ha subito diverse modifiche che ne hanno incrementato le potenzialità, come ad esempio l'introduzione di un iniettore di livello *gate* e l'utilizzo di un modulo che si occupa di analizzare le tracce di esecuzione delle istruzioni di ogni processore. Queste modifiche hanno esteso le potenzialità della metodologia, permettendo di diagnosticare, nel caso di guasto permanente, quale parte del processore si è guastata, oppure permettendo di diagnosticare quale *core* è guasto senza necessità di avere, per ipotesi, un *core* privo di guasti.

Potenzialmente, SWAT è interessante perché sfrutta tecniche avanzate di analisi delle tracce e classificazione accurata degli errori a livello software. La metodologia, però, si limita alla diagnosi del guasto hardware. Sarebbe stato più interessante sfruttare le informazioni ottenute ai due livelli hardware e software per caratterizzare il sistema nel suo complesso ed effettuare un'analisi di propagazione

dei guasti all'interno dell'applicazione.

2.4 Motivazioni del lavoro proposto

In questo capitolo sono state presentate le metodologie per l'analisi di sistemi e le principali strategie di iniezione dei guasti, corredate dalla descrizione di alcuni ambienti che le utilizzano, presenti in letteratura. Per ognuna di esse si è cercato di evidenziare quelli che possono essere i limiti o gli aspetti che non sono stati trattati con sufficiente chiarezza. Dall'analisi di queste proposte si è cominciato a delineare il quadro completo della metodologia presentata nel Capitolo 3, nata con lo scopo di sopperire alle mancanze più significative degli approcci finora presentati.

In generale, una delle più grandi limitazioni delle metodologie appena descritte consiste nel fatto che esse sono applicabili in contesti specifici e con componenti/applicazioni specifici, e difficilmente possono essere estese per risultare efficaci anche in contesti per i quali non sono state ideate. Partendo da questo presupposto, la nostra metodologia è stata sviluppata con l'obiettivo di risultare applicabile in tutti i sistemi *embedded* per i quali un progettista deve verificare le proprietà di affidabilità, senza limitazioni dovute all'hardware utilizzato o al software eseguito.

Un altro problema, riscontrabile nella maggior parte dei lavori analizzati, è il livello di astrazione considerato per la fase di analisi. A un livello troppo basso, infatti, come ad esempio a livello architetturale o addirittura circuitale, non sono disponibili informazioni sufficienti per modellare il comportamento del software eseguito all'interno del sistema, La stessa cosa vale per quelle metodologie che valutano la propagazione dell'errore all'interno delle singole istruzioni *assembly*. Un approccio simile limita il tipo di deduzioni che si possono cogliere dall'osservazione di un fallimento. Un valore diverso in un registro o un parametro erroneo in un'operazione *assembly* non hanno la potenza espressiva necessaria a spiegare quali possono essere le ripercussioni, in termini di comportamento dell'applicazione, che

si verificano sul sistema. Per questo motivo, è importante affiancare, ad un'analisi architetturale approfondita, necessaria anche per lo svolgimento delle campagne di iniezione di guasti, un'analisi software che sia in grado di definire il comportamento dei programmi in assenza di guasto, per poi confrontare, durante le iniezioni, quali cambiamenti avvengono. Nello sviluppo della nostra metodologia, abbiamo posto l'enfasi sull'analisi applicativa dei sistemi in analisi, in particolare dal punto di vista dei *task* eseguiti. Da essi è infatti possibile dedurre innanzitutto se il tipo di errore propagato nel software riguarda il flusso di controllo dell'applicazione o i dati presenti all'interno di essa, e inoltre consente di comprendere in modo accurato in che modo il sistema è fallito e a causa di che tipo di guasto.

Molto importante, poi, a nostro avviso, un'analisi non solo delle uscite, ma anche degli stati interni del sistema, che permetta di comprendere se l'errore si è propagato ma è stato mascherato. Questo viene fatto in alcuni dei lavori analizzati, ritenendo più importante l'analisi dello stato interno rispetto all'analisi delle uscite. Nessuno di essi si spinge, però, nella direzione di associare, a questo tipo di analisi, un'analisi approfondita a livello software.

Infine, l'ultima limitazione riscontrabile riguarda il tipo di classificazione che viene più comunemente adottata. La sola osservazione delle uscite o dello stato del sistema confrontata con le tracce salvate durante l'esecuzione in assenza di guasti non è in grado di fornire un quadro completo della propagazione di un errore fino alla manifestazione di un comportamento anomalo. I risultati osservabili sulle uscite sono infatti riconducibili a più configurazioni errate degli stati interni dei componenti. Nel nostro lavoro abbiamo quindi cercato di definire con maggiore accuratezza le classi di fallimento finali, ereditate da altre metodologie (ad es. FMEA in [14]). Per fare ciò, abbiamo introdotto un nuovo tipo di classificazione, detta intermedia, che ha lo scopo di modellare l'evoluzione del sistema, nel quale vengono effettuati gli esperimenti, in ogni istante. La classificazione intermedia è strettamente legata all'analisi dello stato interno del sistema, e consente di "pre-

dire", per certi versi, le modalità nelle quali il sistema è destinato a fallire. In questo modo, non appena si ottiene una classificazione intermedia, in alcuni casi è possibile interrompere la simulazione perché è possibile già stabilire come proseguirebbe. Ciò consente un'analisi molto più accurata rispetto a quelle proposte in questo capitolo, e consente di far risparmiare tempo di simulazione agli utenti.

Nel prossimo capitolo viene presentata la metodologia oggetto di questo lavoro di tesi. La progettazione e lo sviluppo di E-SWEAM derivano dalle mancanze che sono state descritte in quest'ultima sezione.

Capitolo 3

La metodologia proposta: E-SWEAM

In questo capitolo viene presentata la metodologia *Enhanced Software Error Analysis*, dandone prima una descrizione generale insieme a un modello di *framework* che possa supportarne le diverse fasi, e passando in seguito a un'analisi più approfondita delle singole attività che la compongono.

In seguito, sono presentati i dettagli dell'implementazione della metodologia, con un accenno al linguaggio di descrizione dell'hardware *SystemC* e all'approccio ad alto livello TLM. Inoltre, verrà approfondito il discorso su *Reflective Simulation Platform* (ReSP), la piattaforma di simulazione scelta per questo lavoro e già introdotta brevemente in Sezione 2.2.3.2. In particolare, verranno presentate le strategie di iniezione supportate dalla piattaforma.

Il capitolo si conclude con la descrizione dei moduli che abbiamo implementato in ReSP ed utilizzato per i casi di studio presentati nel Capitolo 4

3.1 Panoramica generale di E-SWEAM

Enhanced Software Error Analysis Methodology (E-SWEAM) è stata definita e sviluppata con lo scopo di fornire, ai progettisti di sistemi realizzati con proprietà di tolleranza ai guasti, un nuovo approccio che consenta la verifica delle proprietà di affidabilità di un sistema *embedded* in tutte le fasi della sua progettazione e rea-

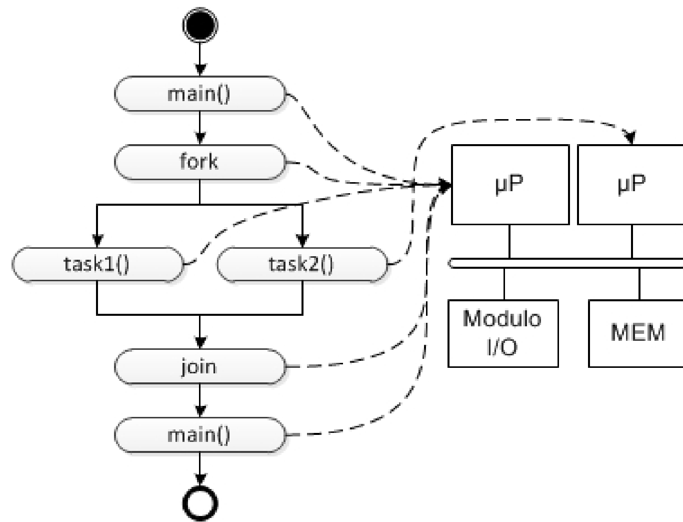


Figura 3.1: *Esempio di sistema hw/sw utilizzato per E-SWEAM.*

lizzazione. Lo scenario specifico in cui ci collochiamo riguarda l'analisi di sistema nelle prime fasi di sviluppo, quando non sono ancora disponibili prototipi sui quali poter effettuare una serie adeguata di test. Il sistema in analisi viene modellato e simulato, e da questo modello vengono ricavate le informazioni necessarie allo svolgimento delle attività di E-SWEAM.

L'approccio che proponiamo, però, è flessibile e può essere riutilizzato in altri passi del flusso di progetto, quando può essere necessaria una rifinitura delle classi di fallimento, in modo da comprendere, nel dettaglio, i motivi di comportamenti anomali derivanti da guasti all'interno del sottosistema hardware.

E-SWEAM si rivolge principalmente alla caratterizzazione e all'analisi di sistemi multiprocessore che eseguono software suddiviso in *task* ed eseguito in parallelo su più *core*. Un esempio di questo scenario è mostrato in Figura 3.1. Per effettuare un'analisi che sia così approfondita, E-SWEAM si serve della tecnica di iniezione dei guasti con lo scopo di verificare le condizioni che sono state definite a partire dalle informazioni ricavate dall'architettura e dall'applicazione.

Il sistema, in termini dell'architettura hardware e dell'applicazione software, viene simulato all'interno di un ambiente chiuso e, a partire dalla descrizione dei singoli componenti hw e delle funzionalità eseguite, vengono ricavate le informa-

zioni necessarie all'applicazione di E-SWEAM. La metodologia è composta da un innovativo *framework* per l'analisi di affidabilità logicamente posto al di sopra di un iniettore di guasti allo stato dell'arte, come quelli descritti in Sezione 2.2. I requisiti che l'iniettore deve soddisfare sono una buona osservabilità dello stato interno del sistema durante le simulazioni (non solo dei risultati finali dell'esecuzione) e la controllabilità completa delle iniezioni e della relativa evoluzione del sistema guastato. La controllabilità è fondamentale, nel nostro approccio, poiché ci interessa monitorare il comportamento del sistema durante ogni istante degli esperimenti, ed eventualmente siamo interessati all'interruzione della simulazione quando sono raggiunte determinate condizioni sufficienti a stabilire lo stato finale nel quale il sistema si porterà. Questa considerazione consente, oltretutto, un risparmio notevole in termini di tempi necessari allo svolgimento degli esperimenti di iniezione dei guasti.

È necessario, vista la complessità dei moderni sistemi *embedded*, analizzare l'impatto degli errori sulle funzionalità svolte per identificare le cause di un fallimento, soprattutto nel caso in cui il sistema sia già stato irrobustito durante fasi di sviluppo precedenti. Gli obiettivi principali di questa metodologia sono:

- fornire un'analisi accurata delle relazioni che sussistono tra i guasti a livello hardware e gli errori che vengono generati a livello software nell'applicazione eseguita;

- fornire i mezzi per monitorare gli effetti dei guasti non solo sugli output del sistema, ma anche negli stati interni dei componenti durante la completa esecuzione dell'applicazione;

- fornire un metodo per effettuare una classificazione automatizzata e personalizzabile dei guasti, che tenga conto degli stadi intermedi nei quali il sistema può trovarsi;

- fornire un metodo per l'analisi della propagazione degli errori in base al tipo di applicazione considerata e al contesto di riferimento nel quale il sistema *embedded* si trova ad operare.

Come accennato prima, per studiare in modo accurato le reazioni del sistema alla presenza di un guasto, si è reso necessario l'utilizzo di un motore per l'iniezione dei guasti basato sulla simulazione, per sfruttare le caratteristiche di osservabilità e controllabilità presentate nel Capitolo 2 quando abbiamo discusso la strategia di iniezione *Simulation-based Fault Injection*. Le attività che compongono E-SWEAM costituiscono, idealmente, una fase di analisi aggiuntiva posta al di sopra di un ambiente nel quale realizzare campagne estese di iniezione dei guasti, simile a quelli che sono stati discussi nel capitolo precedente.

Il motore di iniezione dei guasti sfrutta un meccanismo simile a quello di un *debugger*, ed è in grado di interpretare dati, ricavati dal livello architetturale, a livello dell'applicazione. Attraverso una strategia di *monitoring*, cioè di analisi dello stato di un componente hw/sw, che utilizza un insieme di condizioni di verifica definite dall'utente, è possibile fornire una classificazione del sistema nel suo insieme, che sfrutti le informazioni ricavate attraverso questo meccanismo. La novità del nostro meccanismo di *monitoring* consiste nel fatto che esso è basato su condizioni che possono essere specificate sia a livello architetturale che a livello applicativo, e inoltre è possibile definire strategie *ad-hoc* in base al tipo di funzionalità esposte dall'applicazione.

L'ambiente di iniezione, così definito, deve essere in grado, come abbiamo detto, di fornire un'ottima osservabilità del modello e deve garantire una completa controllabilità della simulazione, per poter ottenere una classificazione dello stato interno "fino a un determinato momento". Questo significa che, durante l'esecuzione della simulazione, in ogni istante, le condizioni specificate per il *monitoring* devono essere verificate. La verifica di tali condizioni deve consentire, al modulo che si occupa di monitorare la simulazione, di capire in quale stato si trova il siste-

ma, e quali possono essere le conseguenze di tale stato. Ad esempio, se osservo il passaggio di un valore errato in ingresso ad una funzione, riconosco che il sistema si sta portando in uno stato nel quale l'output di quella funzione, che utilizza il valore per effettuare dei calcoli, sarà erroneo. Questa considerazione può portare a diverse situazioni: l'output è erroneo ma rimane circoscritto a quella funzione perché non viene più utilizzato dall'applicazione, oppure si propaga corrompendo le successive istruzioni che lo utilizzano, oppure ancora causa la chiamata a una funzione che, nel flusso del programma in assenza di guasto, non era prevista (come una funzione di gestione di un errore in un'applicazione di *safety*).

Infine, un altro requisito che deve essere garantito da questo ambiente di simulazione ed iniezione è il tempo richiesto ad eseguire le campagne di iniezione dei guasti. Vogliamo, infatti, che gli esperimenti compiuti su modelli, che possono essere molto complessi, non abbiano basse prestazioni. In questo, E-SWEAM cerca di venire incontro all'utente dando la possibilità di svolgere campagne mirate, consentendo l'iniezione di guasti dove vi è la certezza di poter corrompere lo stato del sistema e osservare un errore all'interno dell'applicazione; questa strategia è stata scelta per evitare l'esecuzione di numerose simulazioni dove decine di migliaia di guasti sono inseriti, in modo random, nel sistema, per ottenere un valore di percentuale di copertura dei guasti il più alto possibile, durante l'intera campagna di iniezioni.

Dopo questa doverosa introduzione alle specifiche del lavoro che stiamo presentando, è necessario discutere, più nel dettaglio, le singole fasi che compongono la metodologia.

3.1.1 Visione di insieme delle fasi della metodologia E-SWEAM

In Figura 3.2 viene mostrato il flusso di attività che compongono E-SWEAM. La metodologia richiede che in input siano disponibili le descrizioni dei modelli dell'architettura e dell'applicazione che compongono il sistema *embedded* che andrà

analizzato. Su questi modelli, infatti, si svilupperà tutto il lavoro suddiviso nelle fasi che presentiamo, e che comprendono:

- Caratterizzazione del Sistema: la fase preliminare di E-SWEAM ha lo scopo di definire, in modo approfondito, il comportamento del sistema in assenza di guasto. Questa fase è composta da:
  - Analisi Statica e Analisi Funzionale: questi due passi fanno parte di una macro-attività chiamata più generalmente Analisi dell'Applicazione. L'attività di Analisi Statica ha lo scopo di estrarre le informazioni che riguardano il flusso di esecuzione dell'applicazione. L'Analisi Funzionale, invece, è necessaria per capire di quali proprietà è dotata l'applicazione in esame, come, ad esempio, proprietà di *fault detection* oppure di *error avoidance*;
  - Analisi dell'Architettura: attività, mostrata in Figura 3.3, che ha lo scopo di elencare tutte le zone sensibili, in inglese *Sensible Zones* (SZ, [30]) dell'architettura, come registri del processore e celle di memoria utilizzate dall'applicazione. Le SZ sono le candidate ideali all'iniezione dei guasti che verrà svolta nel seguito, ed è necessario averle tutte a disposizione per essere in grado di definire quali di esse possono essere sfruttate con più efficacia durante le esecuzioni delle campagne;
  - Integrazione delle Informazioni: dalle informazioni ricavate dall'Analisi Statica e dall'Analisi Architetturale, questa attività estrae le SZ significative dell'architettura, e inoltre segnala i possibili punti di osservazione, in inglese *Observation Points* (si veda sempre [30]), all'interno del sistema, che rappresentano i punti nei quali osservare la propagazione di un errore, e si trovano a "valle" della SZ nella quale si è iniettato un guasto. Gli OP possono essere le SZ stesse, in alcuni casi;

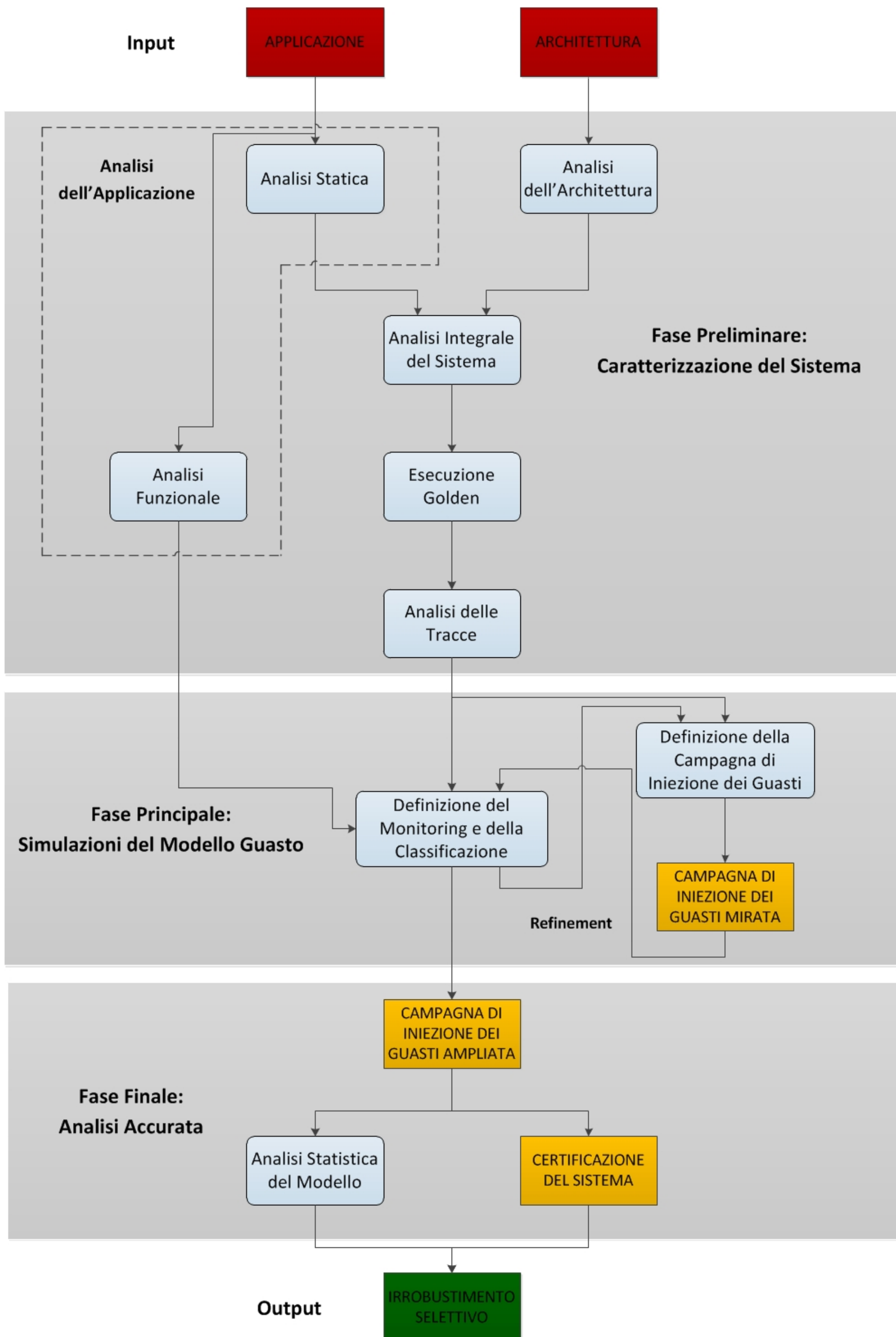


Figura 3.2: *Flusso di esecuzione della Metodologia E-SWEAM proposta.*

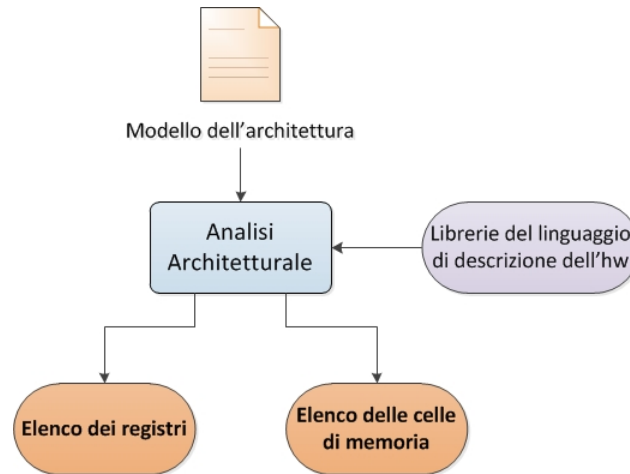


Figura 3.3: *Insieme di attività che compongono l'analisi architetturale del sistema.*

- Esecuzione *Golden*: viene eseguita una prima simulazione del sistema in condizioni di assenza di guasto, con lo scopo di ottenere le tracce relative all'applicazione, in particolare sul flusso di controllo e sui dati, necessarie per l'attività successiva. Questa attività ha anche lo scopo di salvare le informazioni di *Liveness* delle SZ considerate. Questo dato viene utilizzato durante la campagna di iniezione dei guasti, per capire in quale istante iniettare un guasto che possa effettivamente corrompere lo stato del sistema e che non venga, ad esempio, mascherato dalla sovrascrittura immediata della SZ stessa;
- Analisi delle Tracce: per fornire una caratterizzazione accurata a livello software, la metodologia E-SWEAM definisce un'attività di Analisi delle Tracce preliminare alle fasi di simulazione del modello guasto. Le tracce salvate dall'esecuzione *Golden* consentono di costruire un grafo, detto *Execution Flow Graph* (EFG) (per la creazione del quale abbiamo preso spunto da [21, 19, 44, 23]) in grado di modellare, in modo preciso, il comportamento dell'applicazione, informazione strettamente necessaria al tipo di analisi proposta dalla nostra metodologia.

• Simulazioni di Guasto: la fase principale di E-SWEAM è composta da una

serie di attività, ripetute più volte, che servono a rifinire in modo quanto più completo possibile la classificazione finale e intermedia dei fallimenti. La fase è composta da:

  - Definizione del *monitoring* e della Classificazione: una prima definizione è possibile con i dati ricavati dall'analisi delle tracce salvate. E' possibile, però, rifinire le definizioni in base ai risultati ottenuti da una prima serie di esperimenti di iniezione di guasti;
  - Definizione della Campagna di Iniezione dei guasti: una prima campagna di iniezione dei guasti viene eseguita attraverso i dati ricavati dall'Analisi delle Tracce. È possibile, in questo modo, osservare un'iniziale propagazione degli errori, e definire condizioni più specifiche da monitorare per ottenere una classificazione più completa.

- Analisi Accurata: una volta ottenuto un numero di classi di fallimento consono al sistema da analizzare, e una volta che queste stesse classi riescono a descrivere in modo appropriato la propagazione di un errore all'interno dell'applicazione, è possibile eseguire una campagna di iniezione dei guasti più ampia che possa fornire una serie di dati sui quali effettuare analisi statistiche. Una campagna di questo tipo è in grado, infatti, di determinare i componenti più critici dell'architettura e dell'applicazione e fornire una stima di quantità, come la *Safe Failure Fraction* e la *Detection Coverage* (descritte in [14, 30]), necessarie alla certificazione del sistema rispetto a standard internazionali come lo IEC 61508 (presentato in [8, 18]).

In output alla metodologia, siamo in grado di ottenere un'analisi approfondita di quelle che sono le criticità presenti all'interno del sistema, sia da un punto di vista architetturale sia dal punto di vista dell'applicazione. Le criticità, infatti, possono essere rappresentate sia da componenti hardware maggiormente suscettibili all'iniezione di un guasto, sia da funzionalità software che, in presenza di

un errore, rischiano di compromettere l'esecuzione di tutto il programma. Queste informazioni possono rivelarsi molto utili in una fase, successiva all'applicazione di E-SWEAM, nella quale si cerca di irrobustire selettivamente i singoli componenti hw/sw critici, per migliorare, cercando di non ricorrere a tecniche che richiedono *overhead* elevati di area occupata e di consumi, il comportamento del sistema a fronte di un guasto.

Lo scopo finale del l'esecuzione di una o più campagne di iniezione dei guasti, quindi, è di individuare quei componenti, all'interno del sistema, che sono più suscettibili a fallimenti critici, a causa di un guasto transitorio. I sistemi analizzati, come già accennato, sono dotati di proprietà di individuazione dei guasti o di tolleranza agli errori. I meccanismi adottati per garantire queste proprietà possono essere di diverso tipo, e vanno da soluzioni hardware, come *Triple Modular Redundancy*, a soluzioni software, come l'esecuzione parallela degli stessi calcoli per garantire la correttezza di un'elaborazione, fino a soluzioni miste, che possono prevedere, ad esempio in contesti *time-critical*, l'adozione di *timer* all'interno dell'architettura, che siano in grado di monitorare la lunghezza dell'esecuzione dell'applicazione per garantire che essa venga terminata entro vincoli di tempo precisi. In generale queste soluzioni vengono adottate sempre più spesso, non solo all'interno di sistemi nei quali la *safety* è un requisito fondamentale, ma anche in applicazioni di uso più comune per le quali si vuole garantire la correttezza di esecuzione.

Prima di descrivere nel dettaglio le attività più complesse e articolate di E-SWEAM, per chiarire tutti gli aspetti che le riguardano, è necessario introdurre le caratteristiche e i componenti di un *framework* adatto a supportare lo svolgimento delle fasi della metodologia. La trattazione che ne faremo ha lo scopo di definire gli strumenti necessari che verrano sfruttati all'interno di E-SWEAM.

3.1.2 Introduzione al *Framework* proposto per E-SWEAM

Il *framework* ideato per supportare le attività di E-SWEAM viene mostrato nel dettaglio in Figura 3.4. Esso è composto da una piattaforma di simulazione che ha il compito di fornire un ambiente chiuso nel quale poter istanziare il modello del sistema in analisi. La piattaforma è dotata di un motore per l'iniezione dei guasti integrato, che si occupa di accedere all'architettura del modello, elaborare le possibili locazioni di iniezione, e infine iniettare il guasto nella locazione prescelta. Al di sopra di questa piattaforma, abbiamo collocato il *framework* vero e proprio che si occupa di della parte di analisi del sistema, prima e dopo l'iniezione. Il *framework* ha quindi il compito, inizialmente, di caratterizzare il modello del sistema in assenza di guasto. Da queste informazioni, è possibile specificare il tipo di campagna di iniezione che si vuole eseguire ed è possibile impostare i monitor e le sonde per osservare la simulazione di guasto. Idealmente, infatti, il flusso di informazioni si dipana dal sistema di test, viene elaborato dall'interprete dell'applicazione, in inglese *Application-Level Interpreter* (ALI), che trasferisce i dati sullo stato dell'applicazione al "caratterizzatore" del modello in assenza di guasto (*Golden*) chiamato *Application Tracer* (AT). Quest'ultimo si occupa di comunicare, al motore per l'iniezione, ai monitor e alle sonde, i dati sul sistema *Golden* nella sua interezza. A questo punto, possibile eseguire gli esperimenti di simulazione e rifinire la classificazione iniziale ottenuta dall'AT.

Il primo elemento del *framework* presentato è il simulatore del modello hw/sw. Questa piattaforma deve essere tanto funzionale, per poter simulare, in modo semplice ed efficace, sistemi *embedded* complessi, dotati di architetture multi-processore e acceleratori hardware, quanto flessibile, per permettere, sia all'utente, sia al motore di iniezione di guasti, la manipolazione degli stati interni dei propri componenti. Il simulatore è in grado di modellare il sistema di test attraverso la descrizione ad alto livello dello stesso, che si può ottenere attraverso l'utilizzo di linguaggi per la specifica di sistemi, come VHDL. Per simulare un sistema

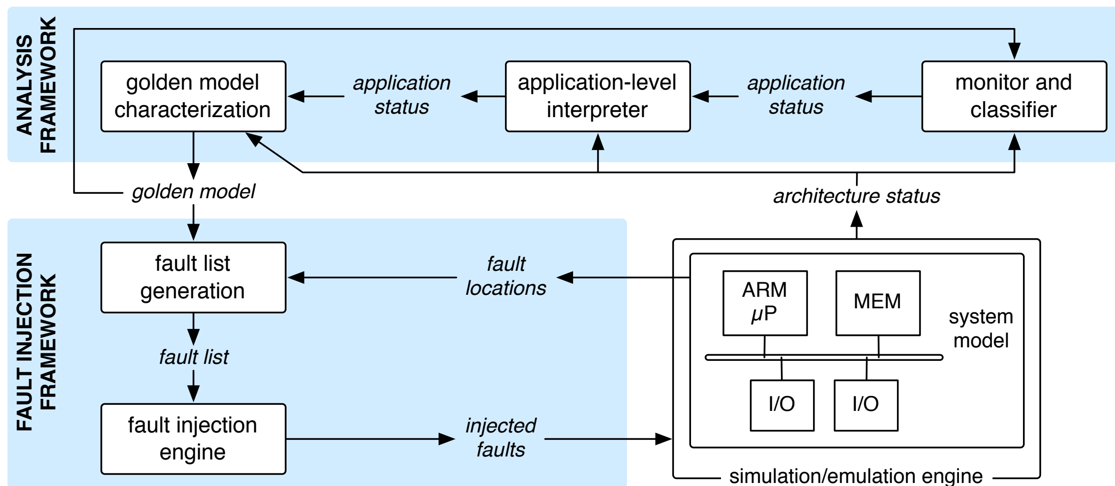


Figura 3.4: *Panoramica del framework associato a E-SWEAM.*

embedded, è necessario che al di sopra del modello dell'architettura venga eseguita un'applicazione.

Le informazioni sullo stato dell'architettura sono trasferite all'ALI, all'AT, ai monitor e alle sonde. L'ALI è un modulo specifico, simile a un *debugger*, che effettua due diverse operazioni: è in grado di effettuare l'analisi e la caratterizzazione statica dell'applicazione, visto il livello di astrazione a cui è in grado di agire, e, durante le simulazioni dell'esecuzione del sistema, è in grado di interpretare lo stato interno dell'architettura e valutare l'esecuzione dell'applicazione.

La prima operazione produce un output che viene direttamente passato all'AT, che si occupa di effettuare una prima simulazione del sistema in assenza di guasto. Questa attività ha lo scopo di caratterizzare il modello e di salvare le tracce di esecuzione dell'applicazione. La caratterizzazione è necessaria al motore di iniezione per generare le liste dei guasti iniettati. In particolare, sono richieste le locazioni (SZ) e gli istanti temporali nei quali si vuole inserire un guasto per analizzare il comportamento del sistema. Le tracce dell'esecuzione dell'applicazione sono passate, invece, ai monitor e alle sonde. Questi componenti hanno il compito di verificare delle condizioni, sono specificate dall'utente, che il sistema deve sod-

disfare. La verifica di tali condizioni è il discriminante per comprendere in quale classe di fallimento, intermedia o finale, si trova il sistema. Le condizioni possono essere di livello applicativo o architetturale, e possono comprendere, ad esempio, il controllo degli indirizzi di lettura/scrittura in memoria, il controllo di valori all'interno di registri, o il controllo dell'esecuzione delle funzioni nell'applicazione.

Stabilite le condizioni, sarà di nuovo l'ALI l'incaricato ad analizzare le informazioni ricavate da monitor e sonde per interpretare lo stato interno dell'architettura e dell'applicazione.

Una volta effettuata la caratterizzazione del modello *golden*, stabilite le condizioni e impostato l'ALI per verificare determinate condizioni, è possibile eseguire le iniezioni di guasti. Il motore avrà già provveduto a generare, a questo punto, una lista di che tipi di guasti iniettare, in quali SZ e in quali istanti temporali. Si tratta, quindi, di accedere allo stato interno dell'architettura (per questo serve un simulatore flessibile e dotato di ottime proprietà di osservabilità e controllabilità del modello) e corromperlo. Se i monitor sono stati impostati in modo efficace, l'ALI avrà a disposizione una quantità di dati sufficiente per classificare i fallimenti del sistema in modo adeguato. Altrimenti, è possibile rieseguire questa fase della metodologia, raffinando ulteriormente le condizioni che andranno specificate.

L'introduzione al *framework* ha consentito di descrivere, da un punto di vista astratto, gli strumenti dei quali necessitiamo e che devono essere associati alle attività descritte nella metodologia. Ora che abbiamo a disposizione questi strumenti, è possibile entrare nel dettaglio di E-SWEAM descrivendo in modo più significativo le fasi principali della metodologia.

3.1.3 Analisi Statica dell'Applicazione

Quando si considera un'architettura basata su processore, l'applicazione eseguita viene implementata per la maggior parte nel software. L'attività di analisi statica ha lo scopo di analizzare il codice sorgente e l'eseguibile, per poter fornire una caratterizzazione in termini delle funzioni ivi contenute, insieme ai loro *inputs* e

outputs. Il livello di granularità scelto per quest'analisi (la chiamata a funzione) è motivato dalle seguenti considerazioni:

- da un punto di vista metodologico, siamo interessati a svolgere un'analisi a livello di granularità delle funzioni chiamate, assumendo, ragionevolmente, che le funzionalità eseguite dall'applicazione siano opportunamente strutturate in singole funzioni (*task*, in inglese) all'interno del codice sorgente;
- l'approccio simile al *debugging* che viene qui presentato può essere, inoltre, applicato a un livello di granularità più fine che concerne la singola istruzione all'interno del codice sorgente.

L'output di questa fase è una tabella dei descrittori delle funzioni, in inglese *Functions Descriptor Table* (FDT), che contiene un descrittore per ogni funzione implementata all'interno del codice.

Le operazioni eseguite durante l'attività di Analisi Statica sono mostrate in Figura 3.5. Per prima cosa, il codice sorgente viene analizzato per estrarre i prototipi delle funzioni contenute. Per ogni prototipo, viene aggiunta una riga alla FDT, nella quale viene specificato il nome della funzione, una lista dei parametri formali e il valore restituito; inoltre, per ogni parametro, la riga identifica il tipo di dato, la modalità con la quale viene passato alla funzione (per valore o per riferimento) e, per ogni parametro passato per riferimento, nel caso esso sia un *array*, la dimensione occupata in memoria, valore che deve essere passato manualmente dall'utente.

In parallelo, vengono estratte informazioni addizionali dal file eseguibile: l'eseguibile viene disassemblato per analizzarne l'organizzazione all'interno della memoria; in particolare, vengono ricavati gli indirizzi di entrata e di uscita dello spazio di memoria contenente il codice di ogni funzione, che sono aggiunti alla riga relativa all'interno della FDT.

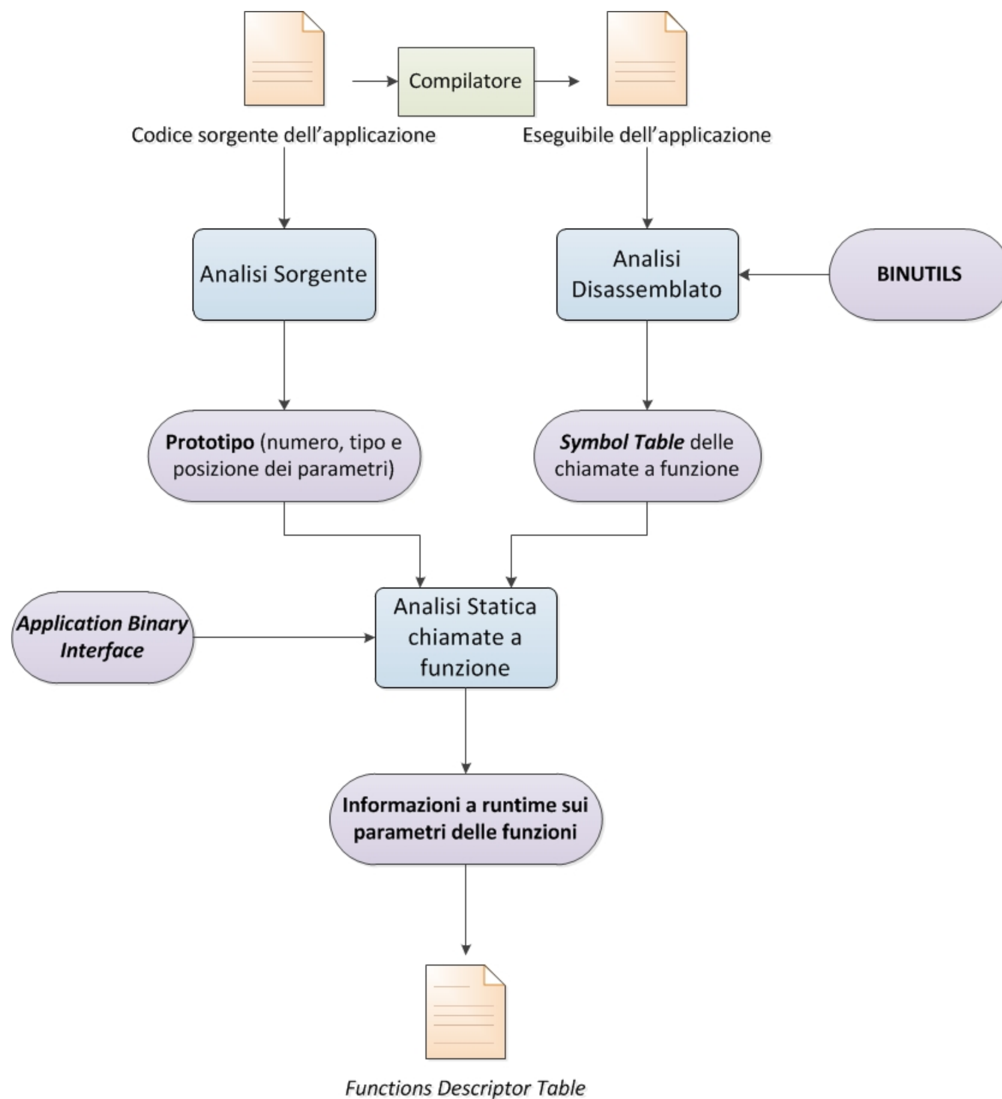


Figura 3.5: *Operazioni che compongono l'attività di Analisi Statica durante la Fase Preliminare di E-SWEAM.*

ID Funzione	Indirizzo di entrata	Indirizzo di uscita	Numero di Parametri	Valore Restituito
Func1	0x00fde0	0x00fdea	5	0xffb41742
Func2	0x00fee4	0x00ffda	2	0
Func3	0x0aff00	0x0cb44	3	52

Tabella 3.1: *Esempio di Functions Descriptor Table: Funzioni.*

Infine, durante l'ultimo passo, ogni riga delle FDT viene analizzata seguendo la convenzione per le chiamate a funzione definita all'interno dell'*Application Binary Interface* (ABI) del processore considerato. L'ABI descrive le interfacce di basso livello tra ogni tipo di applicazione e il sistema operativo sul quale viene eseguita. Essa ha il compito di definire dettagli quali il tipo di dato di ogni variabile, la sua dimensione e l'allineamento (cioè come avviene l'accesso ai dati in memoria); stabilisce, inoltre, il numero di chiamate di sistema presenti e definisce come esse devono essere eseguite dall'applicazione; infine, definisce la cosiddetta "convenzione" di chiamata a funzione, che gestisce come vengono passati gli argomenti di ogni funzione e come vengono ricavati i valori di ritorno.

L'ABI, all'interno della metodologia, viene utilizzata per determinare quali registri o indirizzi di memoria conterranno i parametri passati, e per determinare il registro o l'indirizzo nel quale sarà disponibile il valore di ritorno della funzione stessa; nella FDT vengono quindi aggiunte anche queste informazioni. È importante notare che gli indirizzi di memoria sono specificati relativamente alla posizione dello *Stack Pointer*.

In Tabella 3.1 e in Tabella 3.2 possiamo osservare una FDT con tre diverse funzioni, ognuna con un numero diverso di parametri, la prima delle quali restituisce un indirizzo (un puntatore). I parametri nell'esempio sono passati il primo per valore, il secondo per riferimento.

ID Funzione	Numero del Parametro	Tipo	Modo-Dimensione	Ubicazione
Func1	1	int	Valore	R0
Func1	2	char	Riferimento-256	R1
Func1	3	int	Valore	R2
Func1	4	float	Valore	R3
Func1	5	char	Riferimento-5760	SP+0x0

Tabella 3.2: *Esempio di Functions Descriptor Table: Parametri di una funzione.*

3.1.4 Analisi Integrale del Sistema

Prima di eseguire la simulazione in assenza di guasto, le informazioni ottenute dalle fasi di analisi dell'architettura e di analisi dell'applicazione vengono sfruttate per una fase intermedia, chiamata analisi integrale del sistema, che ha lo scopo di evidenziare le zone sensibili e i punti di osservazione del sistema, che saranno necessarie anche durante le fasi di definizione di *monitoring*, classificazione e pianificazione della campagna di iniezione dei guasti. Come abbiamo discusso precedentemente, quando abbiamo presentato il *framework*, l'ALI svolge una seconda operazione che riguarda la valutazione dell'esecuzione dell'applicazione sia in assenza di guasto sia quando vengono effettuati gli esperimenti di iniezione dei guasti. La fase di analisi statica ha come output una tabella che contiene le informazioni dettagliate delle chiamate a funzione effettuate dall'applicazione, e questo tipo di informazione è proprio ciò che serve all'ALI per interpretare in modo efficace ciò che avviene a livello software.

Una volta popolata, quindi, la FDT viene usata, durante le simulazioni, dall'ALI, che monitora ogni istruzione eseguita da ogni processore nell'architettura, e controlla gli indirizzi di memoria rispetto a quelli contenuti nella FDT. L'ALI può identificare tre diversi eventi:

- *Function Enter*, quando una funzione viene chiamata;
- *Function Exit*, quando l'esecuzione della funzione è terminata;

- *Function Re-enter*, nel momento in cui il controllo viene restituito al chiamante;

Questi eventi possono essere considerati come una sorta di “checkpoint”: nel momento in cui viene rilevato un evento di entrata nella funzione, l’ALI ricava anche i parametri che vengono passati accedendo i registri o gli indirizzi di memoria specificati nella FDT. È importante notare che, per ogni parametro passato per riferimento, la posizione identificata conterrà un puntatore, cioè l’indirizzo nel quale sono salvati i dati; quindi, per accedere ai dati. è necessario dereferenziare il puntatore.

Similmente, durante l’evento di uscita da una funzione, l’ALI ricava il valore di ritorno e, ancora, il valore dei parametri passati per riferimento; infatti, è presumibile che, all’interno della funzione, essi vengano modificati. Quando questi parametri verranno usati nel seguito dell’applicazione, cresce la probabilità che un errore, all’interno di questi dati, venga propagato a funzioni successive. Tutti questi dati vengono poi passati all’*Application Tracer* (AT) e al Classificatore.

Nel caso in cui il sistema contenga anche degli acceleratori *hardware*, la FDT dovrà contenere una riga per ognuno di essi. Se l’invocazione di un acceleratore è contenuta in una funzione del *software*, allora esso può essere trattato nel modo appena descritto; altrimenti, sarà necessario caratterizzarlo in un modo alternativo. Sfortunatamente, dato che non esiste uno standard per le interfacce tra il *software* eseguito sui processori e gli acceleratori *hardware*, questa fase non può essere completamente automatizzata, ma richiede l’interazione del progettista. Noi assumiamo che un generico acceleratore *hardware* sia un componente *slave* per il quale i registri di controllo e i *buffers* dei dati sono “mappati” nello spazio di indirizzamento della memoria. Quindi, durante la prima operazione svolta dall’ALI di analisi statica dell’applicazione, l’utente dovrà occuparsi di caratterizzare le funzionalità eseguite analizzando gli indirizzi di memoria contenuti nei registri di controllo, il comando trasmesso di attivazione dell’acceleratore (se

richiesto), e le celle di memoria che contengono gli input e gli output. Poi, durante la simulazione, l'ALI avrà il compito di monitorare il *bus* di sistema, attraverso una sonda appositamente localizzata, per individuare la richiesta dell'operazione o la sua terminazione. In più, se dovesse verificarsi uno di questi due eventi, il modulo si occupa di ricavare i dati di input o di output direttamente dai *buffer* dell'acceleratore.

3.1.5 Esecuzione *Golden* e salvataggio delle tracce

Una prima simulazione dell'applicazione in assenza di guasto può essere eseguita non appena l'*Application-level Interpreter* (ALI) ha elaborato la FDT generata durante i passi precedenti. Questa esecuzione, chiamata *Golden Execution*, viene effettuata per salvare le tracce di esecuzione dell'applicazione e i dati di *Liveness* dei registri e delle celle di memoria. In particolare, l'esecuzione *Golden* ha lo scopo di salvare gli output dell'applicazione, ma anche alcuni dati intermedi ricavati a determinati *checkpoint*. I dati intermedi, in particolare, sono fondamentali per ricavare la classificazione intermedia (o diagnosi parziale) del fallimento del sistema.

La metodologia proposta introduce un'ulteriore caratterizzazione a livello applicazione: questa attività viene svolta dall'*Application Tracer* (AT), collegato all'ALI, che è in grado di disegnare il cosiddetto *Execution Flow Graph* (EFG); l'EFG è un grafo direzionato aciclico che mostra il flusso di esecuzione dei *task* dell'applicazione durante l'esecuzione *Golden*, come mostrato in Figura 3.6. Il grafo mostra gli eventi di *Function Enter, Re-enter* ed *Exit* dell'applicazione. Se ad un *checkpoint* è associata la traccia salvata di un dato, essa viene mostrata sull'arco che va dal *checkpoint* all'evento successivo. Ad esempio, in uscita da una funzione potrebbe essere necessario tenere traccia dei valori contenuti all'interno di un *array*. In questo caso, sull'arco tra una *Function Exit* e la *Function Re-enter* successiva viene segnalato il nome del file che contiene la traccia considerata.

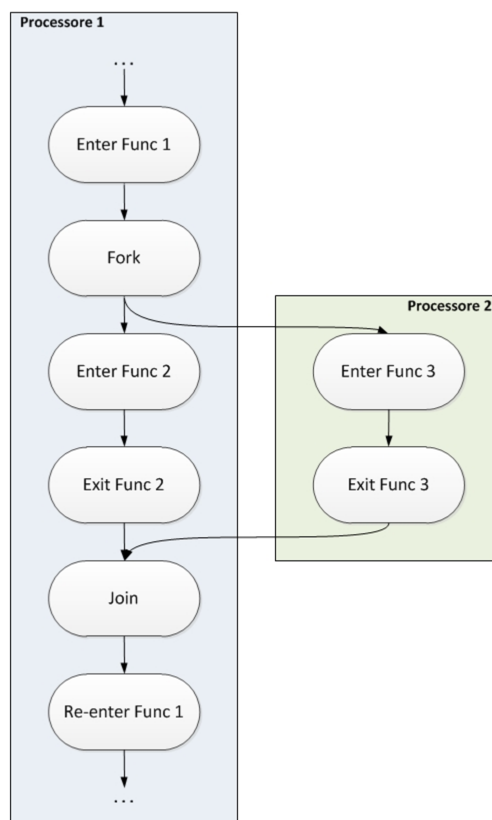


Figura 3.6: *Un semplice esempio di un Execution Flow Graph.*

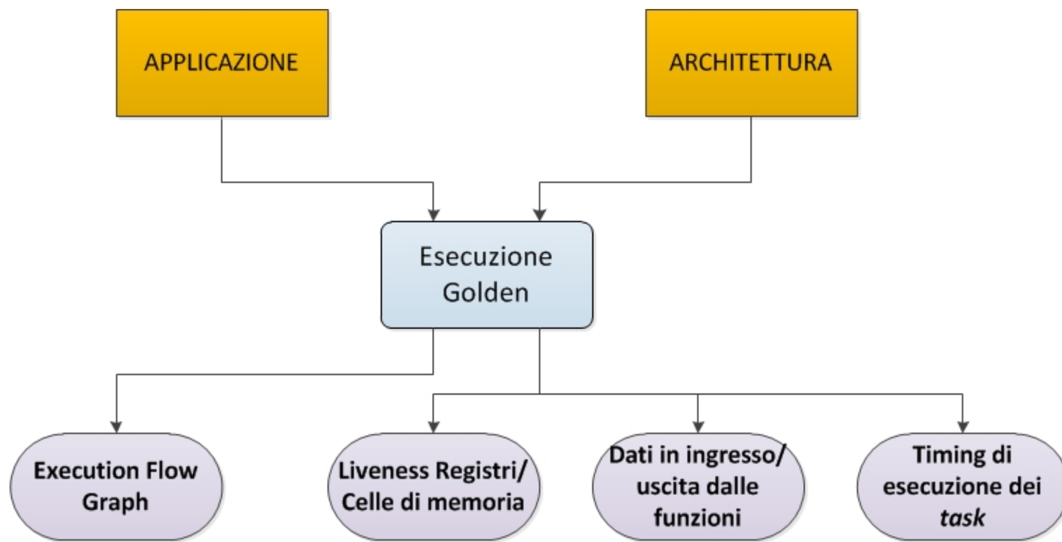


Figura 3.7: *Flusso di operazioni dell'attività di Analisi delle Tracce.*

Ogni evento è caratterizzato in termini dei parametri che vengono passati e dell'istante temporale durante il quale viene eseguito; questo tipo di informazioni sono disponibili grazie all'Analisi delle Tracce. In Figura 3.7 è mostrata questa attività, sottolineando come tutti i dati ottenuti sono direttamente ricavabili dalle descrizioni dell'architettura e dell'applicazione sulle quali sono state svolte le attività di Analisi Statica e di Analisi dell'Architettura.

Inoltre, dato che il sistema può contenere un'architettura multi-processore oppure diversi acceleratori *hardware*, viene utilizzato un modello di applicazione parallelo, che sia in grado di evidenziare i *task* eseguiti da ogni processore e l'ordine nel quale essi avvengono. Ogni arco, inoltre, contiene informazioni riguardo al tipo di dati disponibili durante un determinato evento. Ad esempio, è possibile sapere se in uscita da una funzione esiste un parametro di interesse e dove andarlo a reperire. È importante notare che l'EFG ottenuto può assumere una dimensione considerevole nel caso di sistemi complessi. Per questa ragione, data l'organizzazione gerarchica delle chiamate a funzione, è possibile filtrare in modo semplice l'EFG in modo da considerare solo le funzioni di interesse dell'applicazione considerata, ignorandone altre come le chiamate di sistema.

L'analisi delle tracce dal punto di vista architetturale, infine, ha come obiettivo quello di fornire i dati necessari alla valutazione della *Liveness* dei registri del processore e delle celle di memoria. Queste informazioni diventano fondamentali per la generazione di una lista di guasti durante la definizione della campagna di iniezione dei guasti, in modo da essere consapevoli delle funzionalità che si andranno a corrompere. Infatti, rispetto agli approcci classici, ogni guasto che verrà simulato viene accuratamente caratterizzato in termini della locazione, all'interno dell'architettura, e dell'istante temporale durante il quale si vuole iniettare il guasto, ma anche in termini della funzionalità che si vuole corrompere.

3.1.6 *Monitoring*, Classificazione e Campagne di Iniezione dei guasti mirate

La fase principale della metodologia E-SWEAM consiste nell'esecuzione delle attività di definizione del *monitoring*, della classificazione finale e intermedia e nello svolgimento di una o più campagne di iniezione dei guasti mirate che siano in grado di fornire informazioni specifiche alla rifinitura delle condizioni specificate per il *monitoring* e delle classi definite.

Gli strumenti necessari a questa fase sono stati presentati nella sezione relativa al *framework*, e sono: i monitor, un classificatore dei fallimenti implementato all'interno dell'*Application-level Interpreter* e l'iniettore di guasti che si occupa dell'esecuzione delle campagne.

Per l'importanza che assumono nella buona riuscita della classificazione, questi strumenti devono necessariamente essere flessibili e devono poter essere sfruttati in modo sinergico tra di loro, in modo che le informazioni ricavate da uno di essi possano essere sfruttate da un altro per modellare l'evoluzione del comportamento del sistema in analisi.

Per quanto riguarda la definizione del *monitoring*, possono essere specificati due diversi tipi di condizioni:

- Condizioni a livello applicativo: il classificatore accede allo stato dell'applicazione attraverso l'ALI. Il progettista può quindi definire condizioni sul flusso di esecuzione e sui dati restituiti/scambiati. In particolare, un sottoinsieme di eventi di entrata, uscita e rientro selezionati dalla FDT viene utilizzato come una sorta di lista di *checkpoints*, ai quali il progettista può associare determinate condizioni. A questo livello, possono essere monitorati e valutati più aspetti funzionali, così come è possibile esprimere condizioni più astratte;

- Condizioni a livello architetturale: l'obiettivo è di osservare direttamente lo stato dell'architettura del sistema. Il progettista può richiedere il *monitoring* di:

  - una differenza tra il valore *Golden* e il valore "guasto" di un registro o di una locazione di memoria;
  - una particolare condizione sul valore *faulty* (ad esempio inserendo una soglia tra il valore corretto e quello scorretto).

È importante notare che, utilizzando queste condizioni, il classificatore è in grado di effettuare non una comparazione cieca tra il sistema guasto e quello *Golden*, come invece fanno i gli ambienti presentati in Sezione 2.2, ma effettua, invece, un'analisi intelligente, fino a un determinato istante, dello stato dei componenti e del flusso di esecuzione dell'applicazione, per stabilire in quale classe il sistema si è portato o potrebbe finire.

Per definire in modo accurato la strategia di classificazione, ipotizziamo che il classificatore sia in grado di effettuare un'operazione continua, di tipo diagnostico, di raccolta di informazioni sullo stato interno del sistema; in questo modo, è in grado di restituire, in ogni istante (evento dell'applicazione, come *Function Enter/Exit*), come *output* una risposta unica sull'andamento degli effetti del guasto, durante l'intera finestra di simulazione. Il classificatore riceve in *input* informa-

zioni sia di livello applicativo e architetturale sullo stato del sistema in analisi, sia sulla caratterizzazione del sistema in assenza di guasto, come è stata definita in Sezione 3.1.5. L'*output* del modulo è, per ogni simulazione, la diagnosi del fallimento fino a quell'istante, nel rispetto del tipo di guasto preso in considerazione. Infatti, le informazioni architetturali in input al Classificatore hanno lo scopo di segnalare ad esso che tipo di elemento dell'architettura si vuole compromettere. Esempi di questa operazione sono la corruzione del registro contente il *Program Counter* o lo *Stack Pointer* del processore. Da questo dato, il classificatore è in grado di determinare il tipo di errore che con maggiore probabilità verrà propagato nel software (ad esempio, un errore di controllo).

Dato che la classificazione dell'esperimento dipende dallo scenario applicativo considerato, la metodologia offre la flessibilità necessaria a definire classi personalizzate per fornire una tassonomia degli effetti, causati dal guasto iniettato, sullo stato del sistema. Queste classi possono essere partizionate in due insiemi distinti: classificazione finale e classificazione intermedia. Le classi del primo insieme rappresentano lo stato il finale nel quale il sistema si porta al termine della simulazione, direttamente osservabile sulle uscite. Questo sottoinsieme di classi può essere ereditato da quelli che sono già adottati da *framework* di iniezione dei guasti esistenti.

Possibili esempi di classificazione finale sono:

- **silent / detected / failure**: mostrata in Figura **??** c), descrive gli stati raggiungibili quando sia l'architettura che l'applicazione godono di proprietà di affidabilità;

- **no-effect / critical / not critical**: utilizzata sia per guasti *silent* che osservabili e mostrata in Figura **??** d), distingue l'impatto effettivo del guasto sul comportamento dell'intero sistema e sulle uscite, individuando i guasti che necessitano di tecniche di mitigazione;

- **no-effect / safe / dangerous**: mostrata in Figura **??** e) e utilizzata in FMEA e FMEDA (si vedano [14, 41]) in conformità con lo standard EN/IEC 61508, ha lo scopo di valutare le proprietà del sistema rispetto alla *safety*.

Il secondo insieme di classi descrive i vari passi intermedi nell'evoluzione dello stato del sistema afflitto da un guasto. Queste classi sono anch'esse specificate in accordo con i possibili effetti del guasto sul sistema e sulle proprietà esposte, per esempio in termini di meccanismi orientati all'affidabilità. Alcuni lavori del passato propongono l'adozione di una classe *latent* che rappresenta la sola divergenza tra lo stato interno del sistema rispetto all'esecuzione *Golden*; purtroppo, però, questa classe non fornisce dettagli ulteriori dal punto di vista dell'applicazione, specialmente in scenari basati su architetture a multi-processore. La metodologia proposta fa un passo in avanti, supportando la definizione di un numeroso insieme di classi che sono in grado di garantire una classificazione intermedia molto accurata e specifica per il sistema in analisi. Il più semplice sottoinsieme di classi intermedie è rappresentato dalle classi *control error*, quando un evento non previsto modifica il flusso di esecuzione dell'EFG, oppure *data error*, definito in generale come errore non di controllo, ma che riguarda la scorrettezza dei dati considerati. Questo insieme può inoltre essere ulteriormente specificato in accordo con le proprietà che il progettista intende verificare. Generici errori di controllo possono essere ulteriormente raffinati come errori nell'esecuzione di una funzione, come errori di sincronizzazione tra i *thread* oppure come errori nel ritorno al flusso di controllo che ha effettuato la chiamata all'ultima funzione. Questa distinzione può essere effettuata anche per la classe di errori sui dati, che può essere ampliata, in base all'applicazione, in classi più specifiche. Un esempio della classificazione finale e intermedia di base, supportata in E-SWEAM, è mostrato in Figura 3.8.

L'insieme di classi definito può essere organizzato in una diagramma a stati, per permettere un'analisi dell'evoluzione del sistema e una conseguente diagnosi parziale dei fallimenti che risultino più efficaci. Più precisamente, le classi

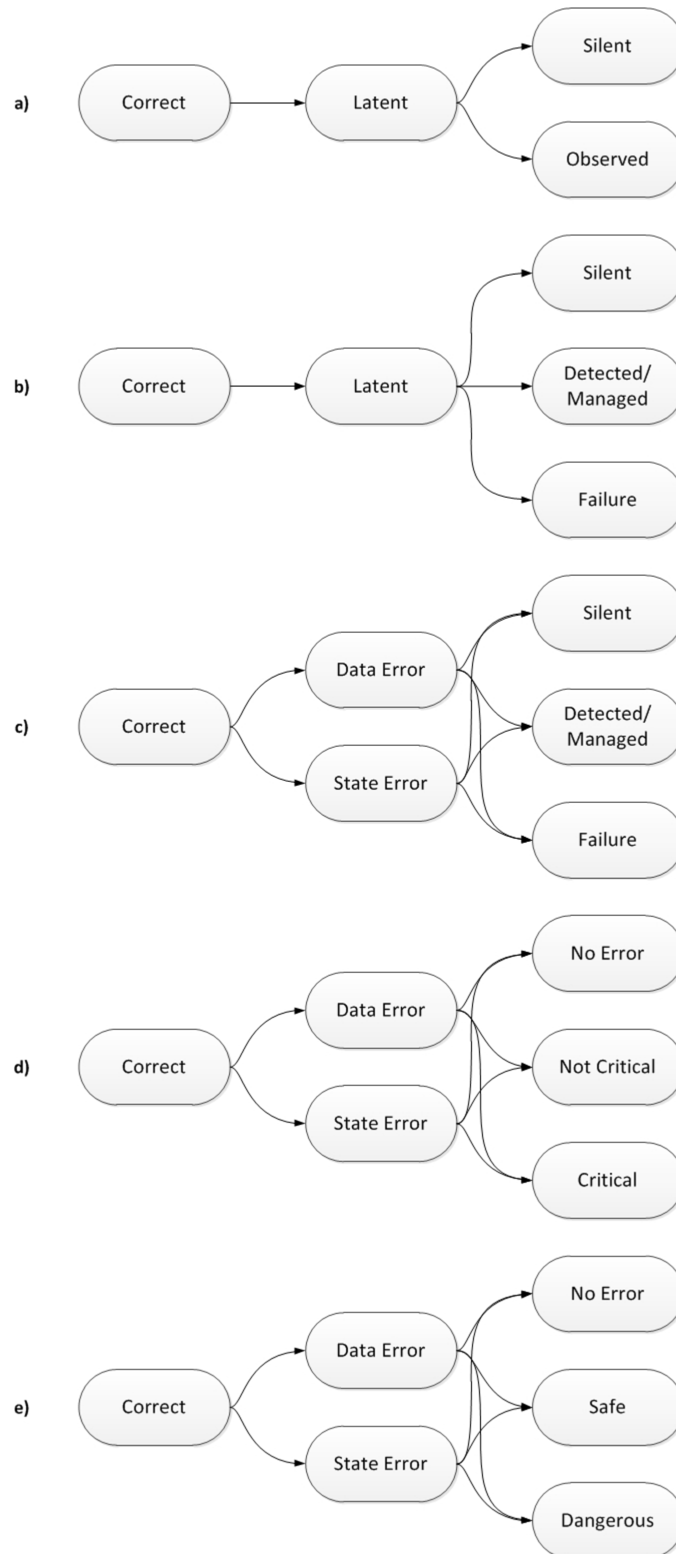


Figura 3.8: *Classificazione base intermedia e finale supportata da E-SWEAM.*

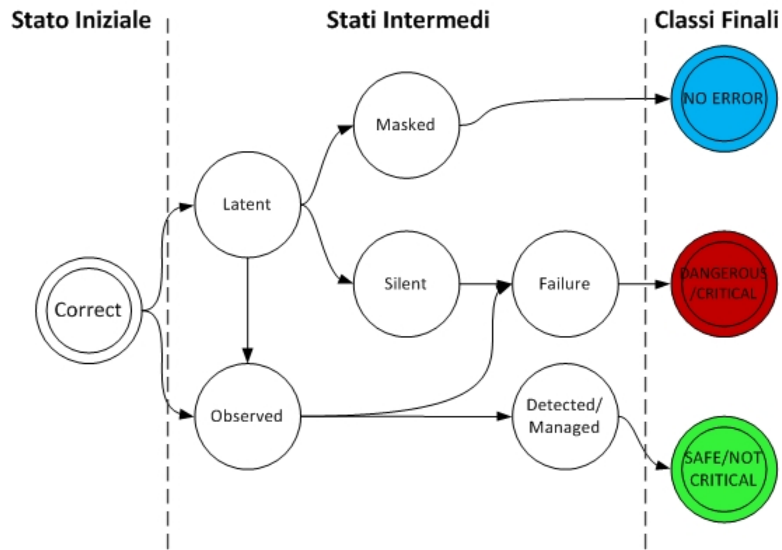


Figura 3.9: *Esempio del diagramma a stati generato dal classificatore: caso base.*

di fallimento possono essere connesse da archi (transizioni) che rappresentano l'evoluzione del sistema da uno stato interno ad un altro.

Il classificatore è in grado di costruire il diagramma a stati dell'evoluzione del sistema seguendo un *pattern* che viene mostrato in Figura 3.9.

Lo stato del sistema è corretto fino all'istante immediatamente precedente all'iniezione del guasto all'interno dello stato interno di un componente. Una volta effettuata l'iniezione, a livello *software* possono accadere due eventi: il guasto rimane *latent* in quanto non produce effetti visibili dal punto di vista dell'applicazione, oppure può esserne immediatamente osservato l'effetto sui dati o sul flusso di controllo (*observed*), in termini dello stato interno che viene alterato. Alternativamente, l'errore può rimanere *latent* per un ciclo dell'applicazione oppure nel *task* all'interno del quale dovrebbe essere osservato, e diventare *observed* nel ciclo successivo oppure nel *task* successivo, evento segnalato dall'arco tra lo stato *latent* e quello *observed*.

Un errore di tipo *latent*, che non provoca effetti osservabili sull'applicazione, può avere due conseguenze sul sistema: essere mascherato dall'architettura o dall'applicazione, oppure rimanere *silent* per la durata della finestra di osservazione

o fino al termine della simulazione. Nel primo caso, il sistema finale si porta in uno stato considerato privo di errori e le uscite mostrano il risultato corretto. Al contrario, nel secondo caso, i monitor non riescono ad individuare l'anomalia all'interno del sistema, ma le uscite risulteranno corrotte. In generale, questo tipo di casistica è la più pericolosa nell'ambito dell'analisi di affidabilità, e viene comunemente identificata con il termine *Silent Data Corruption*.

Infine, un errore osservato attraverso le condizioni specificate nei monitor può avere anch'esso due conseguenze: risultare in un comportamento anomalo che può corrompere l'*output* dell'applicazione oppure può essere correttamente gestito a livello di sistema (ad esempio, causando un'eccezione) e non causare un fallimento.

Esiste, infine, la possibilità, invero abbastanza remota, che un guasto classificato come *silent* provochi la reazione del sistema ad esso, portando il sistema nello stato *detected/managed* senza che i monitor siano in grado di rilevarlo. Vista la rarità dell'evento, abbiamo preferito non includere l'arco relativo nel *pattern* base. Per ipotesi, i monitor che vengono inseriti all'interno del sistema in analisi sono sempre in grado di rilevare un guasto che viene individuato dal sistema stesso. In caso contrario, sarebbe presente una falla significativa nella strategia di *monitoring* adottata e nelle condizioni specificate.

Per ogni sistema considerato, le classi intermedie sulle quali poter specificare condizioni più stringenti o più accurate sono *observed*, *failure* e *detected/managed*. Infatti, ogni progettista può decidere quali eventi o quali valori compromettono la simulazione in modo irreversibile o sono da considerare accettabili al fine di ottenere un risultato finale che non sia compromesso.

Per essere in grado di prendere queste decisioni, è necessario eseguire delle campagne di iniezione dei guasti mirate, sfruttando le SZ ricavate dalla caratterizzazione preliminare del sistema, in grado di fornire una quantità di dati sufficiente a giustificare le ipotesi fatte durante l'attività di definizione della classificazione. Solo testando il sistema in modo appropriato, infatti, si è in grado di giustificare

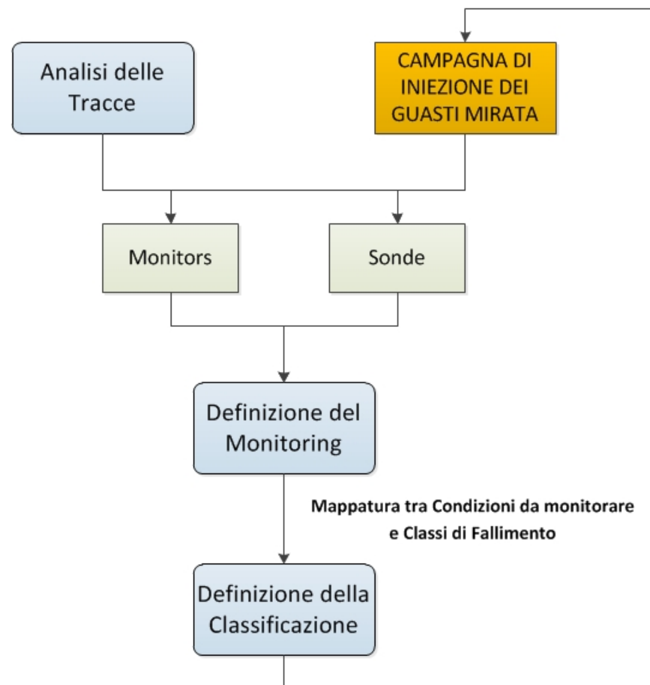


Figura 3.10: *Attività di monitoring, classificazione ed iniezione dei guasti e loro legame.*

le scelte fatte nella stesura del diagramma a stati e delle condizioni da monitorare. In Figura 3.10 sono mostrate le attività appena descritte e il flusso logico che le lega.

In generale, almeno una prima campagna di iniezione dei guasti deve essere effettuata per la verifica della bontà della classificazione proposta. Con buona probabilità, però, l'esecuzione di una prima campagna consente di determinare degli eventi che non erano stati inizialmente previsti oppure che si sono verificati in condizioni nei quali non ce li si aspettava. Proprio per questo motivo, una campagna di iniezione condotta in modo mirato e intelligente è la chiave per ottenere una classificazione dei fallimenti quanto più minuziosa e significativa.

Infine, è importante sottolineare che non esiste un numero di cicli predefinito da eseguire, e le condizioni di equilibrio devono essere decise in base al buon senso e all'esperienza del progettista che sta verificando il sistema. In generale, come riportato nel Capitolo 4 nei casi di studio che abbiamo esaminato, tre ripetizioni sono un numero sufficiente per ottenere una classificazione veritiera dei

comportamenti anomali.

3.2 Dettagli dell'implementazione di E-SWEAM

Questa sezione offre una breve presentazione dei linguaggi e degli strumenti utilizzati per l'implementazione della metodologia presentata in Sezione 3.1. Verrà prima introdotto il linguaggio *SystemC* insieme all'approccio *Transaction-level Modeling*, per poi discutere la piattaforma adottata per lo sviluppo dell'ambiente di analisi ed iniezione dei guasti descritto.

3.2.1 *SystemC* e *Transaction-level Modeling*

SystemC ([2]) è un linguaggio di modellazione che garantisce un approccio unificato per la specifica e la simulazione di sistemi complessi composti da componenti *hardware* e *software*. Dato che fornisce la possibilità di modellare un sistema a diversi livelli di astrazione, supporta un raffinamento iterativo di modelli ad alto livello a livelli di astrazione più bassi, offrendo, così, uno strumento di modellazione per tutte le diverse attività della metodologia di *co-design* HW/SW, come ad esempio la modellazione a livello di sistema, l'esplorazione dell'architettura, il partizionamento HW/SW, la modellazione delle *performance*, lo sviluppo del *software* e la verifica funzionale.

SystemC, come si può vedere in Figura 3.11, è composto da una libreria C++ che mette a disposizione i costrutti necessari alla creazione dell'architettura di un sistema, incluse la concorrenza temporale *hardware* e comportamenti reattivi che mancano nello standard C++; inoltre, la libreria contiene un kernel di simulazione basata su eventi; in questo modo, la specifica *SystemC* può essere compilata ed eseguita come un normale programma C++, raggiungendo velocità di simulazione ragguardevoli.

SystemC, quindi, permette di modellare quegli aspetti peculiari dello hardware che non sono presenti nei linguaggi di programmazione per software, come ad

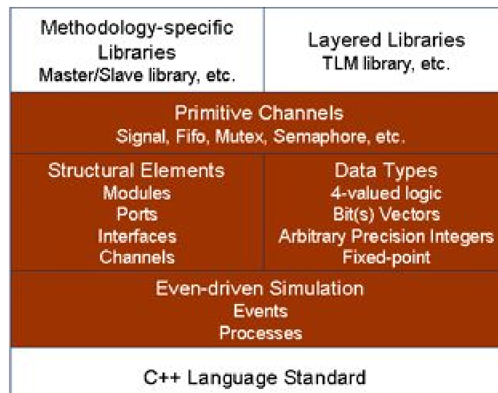


Figura 3.11: *Livelli in cima alla libreria standard C++ che compongono il linguaggio SystemC ([2]), compreso TLM.*

esempio: la nozione di tempo, la nozione di concorrenza, i tipi di dati per la modellazione hardware, le interfacce e la gerarchia delle descrizioni hardware, etc.

Transaction-Level Modeling (TLM, in alto a destra in Figura 3.11, [12]) è stato introdotto negli ultimi anni come uno stile di progettazione per descrizioni ad alto livello e la sua principale implementazione è lo standard OSCI all'interno della libreria *SystemC*. La maggiore peculiarità è che i dettagli della comunicazione tra i moduli sono separati dai dettagli dell'implementazione delle unità funzionali. Gli aspetti computazionali sono descritti in modo algoritmico in C++, mentre i meccanismi di comunicazione sono modellati sulla base di interfacce ad alto livello, basate su chiamate a funzione senza nessun dettaglio di livello basso (*pin*); lo standard OSCI TLM support due differenti livelli di astrazione per quanto riguarda gli aspetti della comunicazione. Una transazione, rappresentata da una chiamata a funzione che simula una *Remote Procedure Call*, modella un trasferimento di dati in TLM. Difatti, a livello transazione l'enfasi è posta maggiormente sulle funzionalità di trasferimento dei dati rispetto all'attuale implementazione (cioè il tipo di dati trasmessi piuttosto che il protocollo utilizzato). Come risultato, una transazione in TLM garantisce elevate velocità di simulazione e le descrizioni dei modelli sono semplificate, dato che non contengono dettagli di basso livello.

Un sistema descritto in *SystemC* TLM, quindi, consente la specifica di siste-

mi molto complessi in modo semplice e quasi immediato, visto che lo scopo del progettista deve protendersi ai soli dettagli delle interfacce tra i diversi moduli e ai dati che vengono scambiate tra di essi. Inoltre, i tempi di simulazione sono accettabili e non si discostano molto da quelli di un sistema reale. *SystemC* non introduce, infatti, un *overhead* consistente rispetto ad un programma scritto in C++ base.

Come spiegato prima, *SystemC* e TLM supportano la modellazione di un sistema a diversi livelli di astrazione. Nel lavoro di E-SWEAM, prendiamo in considerazione uno solo di essi, del quale andremo a descrivere brevemente le principali caratteristiche e gli aspetti di elaborazione e comunicazione. Il livello *Loosely Timed* è il secondo livello di astrazione definito all'interno della libreria TLM (il primo è l'*Approximately Timed*, mentre gli altri due di livello più basso sono già descritti in *SystemC*). L'elaborazione è modellata algoritmicamente, mentre è adottata un'interfaccia di trasporto bloccante per implementare la comunicazione, imitando un puro meccanismo di *Remote Procedure Calls*. I componenti considerati implementano la funzione *transport* che il costruttore chiama quando effettua una richiesta; durante la transazione, il *payload* contenente i dati relativi alla richiesta viene trasmesso e, se necessario, viene aggiornato dal componente considerato e ritrasmesso attraverso la funzione di *return*. Dato che la chiamata è bloccante, essa restituisce il controllo dell'esecuzione al costruttore quando la richiesta è completata.

3.2.2 Reflective Simulation Platform

Avendo scelto di appoggiarci ad un linguaggio che potesse garantire tempi di simulazione accettabili e semplicità di modellazione dei componenti, il passo successivo è stato scegliere una piattaforma in grado di effettuare un'analisi non intrusiva del sistema e che nel contempo avesse a disposizione un motore di iniezione dei guasti di livello architetturale per sfruttare la tecnica di iniezione dei guasti necessaria

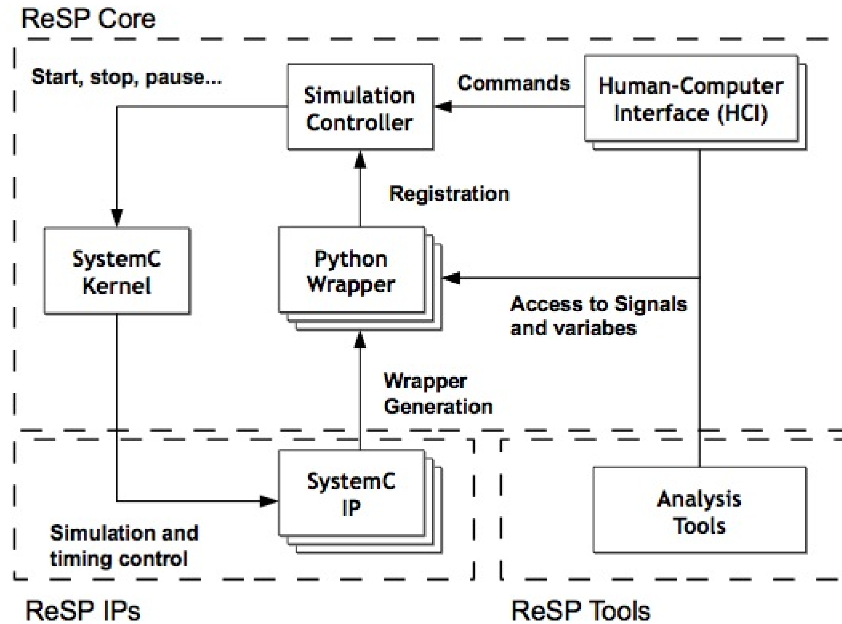


Figura 3.12: *Vista d'insieme dell'architettura di ReSP, con enfasi sui collegamenti tra i diversi moduli.*

a validare le assunzioni fatte durante le fasi di caratterizzazione del sistema e di definizione delle classi di fallimento.

Reflective Simulation Platform (ReSP, [9, 10]) è una piattaforma di simulazione per specifiche di sistema basata sui linguaggi *SystemC* e *Python* ([3]), particolarmente adatta alla descrizione di sistemi multi-processore a livello transazione. Lo scopo di ReSP è fornire al progettista un modo semplice per specificare l'architettura di un sistema seguendo una metodologia di design basata sui componenti con un riutilizzo sistematico di moduli *Intellectual Properties* (IP), per simulare una data configurazione ed effettuare analisi di sistema parzialmente automatizzabili.

In Figura 3.12 viene mostrata l'architettura della piattaforma. Il cuore del simulatore è il *kernel SystemC* che offre le funzionalità fondamentali per simulare specifiche in *SystemC*. In cima al *kernel*, la piattaforma di simulazione è stata costruita in *Python* con lo scopo di sfruttare le proprietà riflessive del linguaggio. La riflessione, anche detta introspezione, è una proprietà offerta da alcuni linguaggi

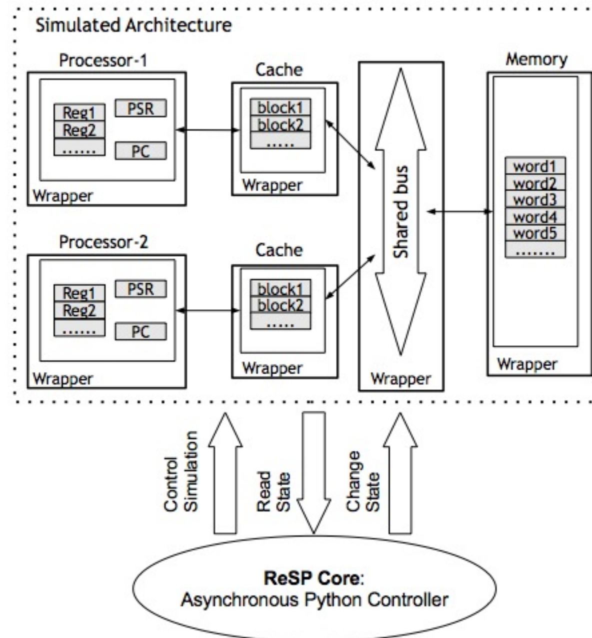


Figura 3.13: *Dimostrazione della proprietà riflessiva della piattaforma ReSP.*

moderni come, appunto, *Python* o *Java*, che consente di osservare, accedere e modificare la struttura del sistema a *runtime*, come mostrato in Figura 3.13; attraverso questa proprietà, infatti, un programma può accedere, a *runtime*, a variabili e funzioni di un oggetto senza possedere una conoscenza a priori della descrizione della classe alla quale appartiene. In questo modo, la piattaforma offre la possiblità di osservare la struttura interna del modello di un componente *SystemC* senza la necessità di modificarne il codice. Questa proprietà, unita al fatto che *Python* è un linguaggio di *scripting*, permette la composizione a *runtime* (non è necessario scrivere e compilare files della struttura del sistema) e la gestione dinamica dell'architettura in analisi. L'interoperabilità tra il *kernel SystemC* e le descrizioni dei componenti con i moduli di *Python* sono rese possibili attraverso l'utilizzo di un insieme di *wrappers Python* generati automaticamente attraverso uno strumento specifico descritto in [9].

È possibile accedere al *kernel SystemC* dal livello di gestione attraverso un controllore di simulazione che offre un insieme di funzionalità per facilitare l'i-

stanziamento del sistema. Il progettista può comandare la piattaforma attraverso un interfaccia testuale che è un'estensione della *console Python*. Il controllore di simulazione lavora sia in modo interattivo che *batch*. Per il nostro lavoro, è più interessante il primo modo di utilizzo, che offre funzionalità come come `pause` e `resume` asincrone, senza pericolo di perdita di consistenza.

ReSP è dotato di una libreria di componenti IP base, come ad esempio processori, *bus* e memorie; è comunque supportato un approccio per la progettazione di nuovi componenti. In particolare, la libreria è integrabile con qualsiasi componente *SystemC* (grazie anche alla generazione automatica di *wrapper* garantita da *Python*). La specifica di un sistema consiste nell'inizializzazione dei suoi componenti e delle loro interconnessioni solitamente contenuta in un file di configurazione (in *Python*). La compatibilità tra le numerose interfacce dei componenti è garantita dal fatto che essi sono specificati con un approccio TLM a livello di astrazione *Loosely Timed*.

3.3 Strategie di iniezione dei guasti basate sulla simulazione

Il contenuto di questa sezione viene ampiamente discusso da Miele in [31] nel capitolo 4. In questo contesto verranno discusse solamente le strategie utilizzate per la stesura della metodologia E-SWEAM. Rimando alla tesi citata per ulteriori nozioni sulle strategie esistenti per l'iniezione di guasti a livello di simulazione.

Il requisito fondamentale per le strategie di iniezione dei guasti a livello di simulazione è che esse siano il meno intrusive possibile rispetto ai modelli IP e che siano indipendenti dal livello di astrazione della loro descrizione. Infatti, uno degli obiettivi dell'approccio sviluppato per questa metodologia è la definizione di strategie di iniezione che non richiedano al progettista di modificare la specifica *SystemC* per svolgere l'analisi di affidabilità; il motore di iniezione dei guasti deve iniettare il guasto direttamente nel modello nominale in *SystemC*. Le motivazioni

alla base di un approccio così trasparente risiedono nella necessità di non interferire con la descrizione nominale: le modifiche per introdurre la strumentazione necessaria all'iniezione può richiedere tempo ed essere subordinata ad una specifica implementazione; inoltre, può portare ad errori nel codice e, quando si considera l'approccio di progettazione basato su piattaforma supportato da ReSP, è anche possibile che il codice sorgente di quel componente non sia disponibile, avendo a disposizione solo un modello compilato all'interno del *repository* del simulatore.

Nello stesso modo, i componenti possono essere specificati a differenti livelli di astrazione (dalla descrizione a livello RTL - bassa - a quella transazionale - alta - utilizzata in E-SWEAM); è quindi importante definire strategie che possano essere adottate in tutte le situazioni, indipendentemente dal livello di astrazione per poter funzionare in contesti diversi, nei quali può essere necessaria la verifica del comportamento di un componente a basso livello. Infine, l'approccio di gestione dinamica del sistema fornito dalla piattaforma considerata consente di gestire iniezioni a *runtime* manualmente.

Per queste ragioni, sono state definite strategie basate sui comandi della simulazione e su *saboteurs* per l'iniezione di guasti nello stato interno dei componenti e nelle comunicazioni tra essi. Nel primo caso, il valore di un segnale o di una variabile viene manipolato a *runtime* dalla console della simulazione, manualmente o in modo automatizzato, attraverso l'uso di *macro* e *script*. Nel secondo, invece, viene inserito un modulo specifico di iniezione dei guasti, il *saboteur*, appunto, sull'interconnessione tra due componenti. Il *saboteur* viene comandato attraverso un controllo in *input* per attivare l'iniezione del guasto.

3.3.1 Iniezione nello stato interno

L'iniezione nello stato interno è stata realizzata seguendo l'approccio dei comandi della simulazione, attraverso la riflessione garantita da *Python*. Anche se *SystemC* non fornisce strumenti avanzati per questo tipo di meccanismi di iniezione, le capacità di introspezione offrono la possibilità di accedere e modificare l'attributo

desiderato del componente tra due cicli di *clock* (o, più generalmente, tra due *delta cycles*); in questo modo viene garantita la possibilità di iniettare un guasto ed osservare l'errore propagato. La modifica degli attributi viene effettuata in base alla "maschera di corruzione" estratta da un insieme di maschere standard, come ad esempio il classico *bit-flip* o un generico cambiamento di valore.

La strategia definita può essere adottata per l'iniezione di guasti multipli, intermittenti o permanenti semplicemente modificando degli specifici attributi. Nell'ultimo caso, essa consiste nell'aggiornamento ripetuto della locazione del valore guasto.

Il vantaggio principale di questo approccio è che esso è stato implementato senza modificare il *kernel SystemC*, garantendo la possibilità di aggiornare l'ambiente con nuove versioni del linguaggio senza per questo dover aggiornare gli strumenti di iniezione dei guasti. L'unica limitazione, invece, consiste nel fatto che i *wrappers Python* consentono l'accesso ai soli attributi pubblici; per questo motivo, tutti gli attributi di oggetti C++ considerati possibili locazioni di guasto devono essere dichiarati `public`, o in alternativa, ne devono essere forniti i metodi `get` e `set`.

3.3.2 Iniezione nella comunicazione

L'approccio precedente è valido anche per la comunicazione nel caso vengano utilizzati dei segnali. Infatti, i segnali sono modellati come oggetti C++ e possono essere modificati nello stesso modo degli altri componenti. Quando vengono utilizzate interfacce di comunicazione ad alto livello, però, l'approccio precedente non è più applicabile, poiché la comunicazione tra i componenti viene implementata attraverso transazioni che sfruttano chiamate a funzioni. In questo caso, la strategia per l'iniezione di guasti nelle comunicazioni è stata implementata attraverso l'utilizzo di *saboteurs* inseriti tra due componenti.

I *saboteurs* implementati supportano una lista di possibili corruzioni che possono essere applicate e possono essere sfruttati per simulare effetti permanenti,

come ad esempio la corruzione di tutte le transazioni dopo un determinato istante temporale.

Il vantaggio del nostro approccio è la possibilità di introdurre il componente in modo trasparente durante l'instanziamento e la composizione del sistema. Infatti, proprio perché ReSP supporta l'instanziamento e la composizione a *runtime* senza richiedere alcun file *SystemC* che contenga la specifica del sistema, è solo necessario includere il codice del *saboteur* nel *repository* di ReSP e poi instanziarlo e connetterlo sulla linea desiderata. È importante sottolineare che la limitazione comune dei *saboteurs* dipende dalla loro implementazione non indipendente dalla specifica del sistema, ma strettamente legata alle interfacce dei componenti ai quali dovrebbe essere collegati. Quando si utilizza l'approccio TLM, però, tutte le interfacce sono standard; è così possibile definire un unico *saboteur* utilizzabile con tutti i componenti.

3.4 Descrizione dei moduli sviluppati in ReSP per il supporto a E-SWEAM

La possibilità di aggiungere componenti alla piattaforma scelta unita alla proprietà di introspezione garantita da *Python* consente, al progettista, di specificare nuovi componenti che vadano a migliorare le capacità di analisi del sistema fornite dall'ambiente. Il disaccoppiamento tra il simulatore stesso e i modelli *SystemC* permette l'accesso e la modifica ai componenti senza dover conoscere a priori come essi sono fatti, e inoltre consente di non modificare il codice del simulatore nel caso alcuni modelli siano diversi. In questo modo, per quanto riguarda questo lavoro di tesi, è stato possibile integrare componenti per l'analisi di sistema a livello software mirata alla verifica della proprietà non funzionale di affidabilità.

In Sezione 3.1.2 abbiamo descritto le caratteristiche che i moduli del *framework* proposto dovevano avere. Ora vediamo nel dettaglio i componenti introdotti in ReSP e utilizzati per E-SWEAM.

3.4.1 *Function Profiler*

Il *Function Profiler* (FP) è un modulo collegato al multiprocessore del sistema che unisce alcune funzionalità proprie dell'ALI e dell'AT descritte in Sezione 3.1.3 e in Sezione 3.1.5 e ha il compito di estrarre il profilo operazionale di un dato carico di lavoro. Il profilo operazionale è una collezione di informazioni che riguardano tutte le attività rilevanti del sistema in assenza di guasto; queste informazioni possono comprendere, ad esempio, le tracce di esecuzione dell'applicazione, le attività di lettura/scrittura associate ai registri del processore, agli indirizzi di memoria o ai dati scambiati sul *bus*.

Per quanto riguarda l'utilizzo di FP all'interno di E-SWEAM, il modulo ha il compito di stampare una traccia, per ogni processore, dell'esecuzione degli eventi di *Function Enter/Exit/Re-enter* di ogni funzione eseguita all'interno dell'applicazione. La traccia contiene, per ogni evento, l'instante temporale nel quale viene eseguito e, per le funzioni contenute nella FDT, della quale abbiamo parlato in Sezione 3.1.3, anche i valori o gli indirizzi dei parametri passati al momento della chiamata.

Se, all'interno di una funzione della FDT, esistono parametri passati per valore o riferimento per i quali è stato specificato il salvataggio dei dati durante il *checkpoint*, il FP si occupa di creare, per ognuno di essi, un file nel quale vengono salvati i valori del parametro. Facciamo un esempio: una struttura *array* viene passata, per riferimento, ad una funzione che ha il compito di riempirlo di dati; se nella FDT è stato specificato che il contenuto dell'*array* deve essere salvato, durante l'evento di *Function Exit* dalla funzione alla quale esso è stato passato, il FP verifica l'indirizzo in memoria nel quale si trova l'*array*, vi accede, e salva in un file il contenuto delle celle di memoria che contengono i valori dell'*array*, a seconda della dimensione specificata all'interno della FDT. Questi file fanno parte delle "tracce" che verranno utilizzate successivamente durante le simulazioni di guasto per confrontare i valori *Golden* con i valori ottenuti dall'esperimento di

iniezione.

Al termine di questo processo, il FP unisce le tracce degli eventi di ogni processore in un unico file, in ordine cronologico. La traccia così ottenuta viene utilizzata per la generazione dell'EFG descritto in Sezione 3.1.5, al quale sono eventualmente aggiunti, se esistono, riferimenti ai file che contengono i dati salvati dai parametri *checkpointed*.

FP ha il compito, inoltre, di salvare la traccia delle letture/scritture effettuate su registri, celle di memoria e indirizzi del *bus*. Queste tracce sono necessarie all'attività di analisi di *Liveness*, attraverso la quale sono ricavate le informazioni che devono essere passate all'iniettore per la generazione di liste di guasti efficaci. Sfruttando le proprietà di ReSP, questa operazione risulta molto semplice: è solamente necessario modificare leggermente il modello *SystemC* del processore, della memoria o del bus per ottenere la stampa di tutte le operazioni associate a registri, celle o indirizzi. FP elabora questi dati e restituisce le informazioni sulla *Liveness* di ognuno di essi.

Lo scopo del FP è di far comprendere meglio la situazione nella quale il sistema o l'applicazione verranno usati, per poi analizzare le informazioni raccolte per garantire che solo i guasti che produrranno un errore osservabile siano selezionati durante il processo di generazione delle liste di guasto. In questo modo, la lista generata è compatta e non banale.

3.4.2 *Checker Tool*

Il modulo più importante per l'implementazione di E-SWEAM è il *Checker Tool* (CT). CT unisce le potenzialità dell'ALI a un classificatore dei fallimenti che segnala la classe di fallimento nella quale si trova il sistema attraverso la verifica di condizioni a livello applicativo.

CT svolge due compiti fortemente legati tra loro: monitora i'applicazione durante l'esperimento di iniezione di guasti e stabilisce la classe di fallimento del sistema durante o al termine dell'esperimento.

Il primo compito di CT consiste, similmente a quello che viene fatto da FP, nell'analisi dell'esecuzione dell'applicazione durante l'iniezione di un guasto. CT è in grado di rilevare gli eventi di *Function Enter/Exit/Re-enter*, e quindi di valutare il flusso di esecuzione dell'applicazione in presenza di errori. Per ogni evento possono essere specificate delle precise condizioni che servono a verificare la presenza di un errore sul flusso di controllo oppure un errore sui dati. Queste condizioni possono essere semplici, come ad esempio l'inserimento di un *flag* che indichi l'ingresso/uscita da una funzione, oppure possono essere più complicate, e ad esempio possono includere l'analisi delle tracce salvate durante l'esecuzione *Golden*.

Le condizioni specificate garantiscono che la classificazione dello stato del sistema sia incrementale e, in caso di raggiungimento di una determinata condizione, che venga interrotto l'esperimento. Ad ogni evento al quale è associata una condizione, infatti, tale condizione viene verificata. In caso di soddisfacimento, il classificatore, che abbiamo visto poter essere rappresentato attraverso un diagramma a stati, segnala che il sistema si è portato in un nuovo stato. Se uno di questi stati è uno stato finale, la simulazione viene interrotta e si può iniziare la fase di analisi. Se, al contrario, lo stato è intermedio, la simulazione prosegue fino al proprio termine oppure fino al raggiungimento di uno stato detto finale o, in modo più appropriato, terminatore. Se il sistema raggiunge questo tipo di stato, significa che è possibile determinare con accuratezza le conseguenze dell'errore che si è propagato, e non è necessario proseguire l'esperimento per raccogliere altri dati.

Il CT, al termine dell'esperimento, sia esso terminato normalmente o interrotto dal CT stesso, produce un'uscita contenente lo stato finale del sistema, una breve descrizione e gli eventuali errori che sono stati osservati. Da questo output, è possibile stabilire con accuratezza tutto ciò che è successo all'interno del sistema durante un dato esperimento di iniezione dei guasti, ed è inoltre possibile

utilizzare l'output del CT per ricavare dati statistici utili a calcolare, ad esempio, i parametri di percentuale di copertura dei guasti o la frequenza di fallimento di un determinato *task* dell'applicazione.

3.4.3 *Delta Monitor* e Sonde

Il *Delta Monitor* (DM) è un modulo di livello architetturale che ha il compito di osservare lo stato del componente al quale è associato ogni *delta* temporale, che è rappresentato da un ciclo di *scheduling*.

All'interno di DM possono essere specificate condizioni che i componenti osservati devono rispettare. Essendo un modulo di livello architetturale, le condizioni che è possibile specificare sono limitate e legate al valore che viene letto dal DM. E' possibile, quindi, osservare un valore scorretto (tramite il confronto con il valore letto durante l'esecuzione *Golden*) oppure è possibile impostare una soglia all'interno della quale dovrà essere compreso il valore osservato.

Il vantaggio di DM è la frequenza con la quale verifica che le condizioni specificate siano soddisfatte. Nel momento in cui una condizione viene violata, DM può far interrompere la simulazione e segnalare, al CT, il motivo. In questo modo è possibile osservare in modo parallelo l'esecuzione della funzione e lo stato interno dei componenti per ottenere una classificazione che tenga conto di ogni aspetto del sistema durante gli esperimenti.

Le Sonde, in modo simile al DM, sono collegate alle interconnessioni e, per ogni transazione che passa attraverso di esse, si occupano di osservare l'indirizzo, il tipo di transazione (richiesta o risposta) e i dati contenuti. Si possono collegare in modo molto semplice (grazie a TLM) alle interfacce dei componenti che devono monitorare, e fanno una sorta di *sniffing* controllato del traffico sull'interconnessione. Anche per le sonde è possibile specificare delle condizioni che devono essere verificate. Le condizioni possono comprendere condizioni sui valori, simili a quelle descritte per DM, oppure condizioni sugli indirizzi: è possibile interrompere la

simulazione o, in generale, segnalare l'accesso a un indirizzo di memoria che può compromettere l'esecuzione della simulazione.

In questo capitolo abbiamo presentato la metodologia proposta in questa tesi. Nel prossimo capitolo si illustra l'applicazione di quest'ultima a due casi di studio significativi, analizzandone pregi, difetti e discutendo le eventuali scelte di irrobustimento ulteriore.

Capitolo 4

Casi di Studio per la Validazione della Metodologia

In questo capitolo sono presentati i casi di studio per la validazione della metodologia E-SWEAM descritta nel Capitolo 3. Verranno mostrate le tecniche e le soluzioni proposte in precedenza, applicandole a sistemi *embedded* modellati e simulati con ReSP. Lo scopo di questo capitolo è mostrare quali informazioni siamo in grado di ottenere in più rispetto ai metodi classici di analisi di affidabilità e vogliamo, inoltre, testare l'efficacia delle proprietà di affidabilità dei sistemi in analisi.

L'attenzione, nella trattazione dei casi di studio e dei risultati ottenuti, sarà rivolta ad evidenziare come queste informazioni aggiuntive, che E-SWEAM è in grado di estratte dal modello del sistema simulato, ci consentano di modellare con precisione i fallimenti che avvengono all'interno dei sistemi in analisi, così da comprendere quali componenti o quali *task* dell'applicazione necessitano dell'applicazione di tecniche di mitigazione degli errori oppure di irrobustimento ulteriore.

Per mostrare quanto la metodologia sia valida anche in ambiti completamente diversi tra loro, le applicazioni scelte sono un programma per l'elaborazione di un'immagine che si affida a periferiche per la ricezione e la trasmissione dell'immagine stessa (*EdgeDetectorParallel*) e un programma per la modellazione del

comportamento di un modulo *Anti-lock Braking System* (*SimpleABS* presente, ormai, in tutte le centraline di vetture disponibili sul mercato.

4.1 Caso di studio *Edge Detector Triple Modular Redundancy*

Il caso di studio *Edge Detector Triple Modular Redundancy* (EDTMR), presentato per la validazione della metodologia E-SWEAM, consiste in un sistema hw/sw per l'elaborazione di immagini, che svolge le attività di rilevazione dei bordi e loro sovrapposizione sull'immagine originale. Il sistema è irrobustito attraverso un'implementazione software della tecnica *Triple Modular Redundancy* (TMR, [28]).

Verrà fornita, brevemente, la caratterizzazione del sistema descritto, e saranno discusse, in modo approfondito, le strategie di *monitoring* e classificazione applicate. L'obiettivo è fare una valutazione accurata della tecnica di irrobustimento mediante un'analisi accurata del comportamento interno del sistema.

4.1.1 Caratterizzazione del sistema

Il sistema *embedded* utilizzato per l'analisi è composto da un'architettura multiprocessore basata su processori ARM9, e si serve di un dispositivo esterno che ha il compito di acquisire e trasmettere l'immagine elaborata. In Figura 4.1 è mostrata una panoramica dell'hardware e del software considerati.

Il sistema, come premesso, è stato irrobustito a livello software attraverso l'utilizzo della tecnica TMR. TMR è un particolare tipo di ridondanza N-modulare, in cui tre diversi sistemi eseguono un processo. I tre risultati vengono sottoposti ad un sistema di *voting* per produrre un unico risultato corretto. Se uno dei tre sistemi fallisce, gli altri due sistemi sono sufficienti a garantire che il risultato prodotto dalla fase di *voting* sia corretto. Il fallimento del *voter*, in compenso, comporta il fallimento dell'intero sistema. Per questo motivo, in un buon sistema

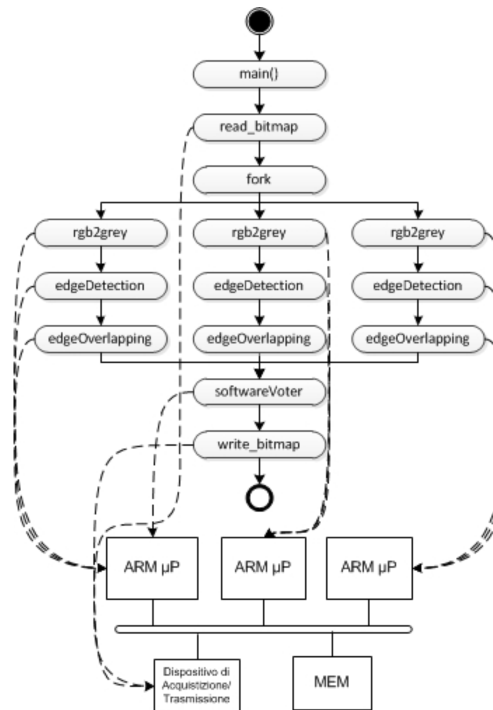


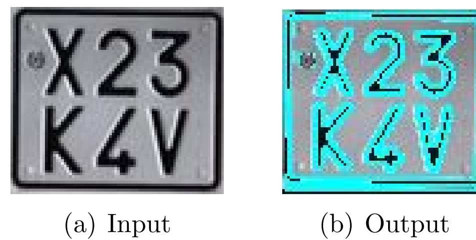
Figura 4.1: *Modello del sistema utilizzato per il caso di studio EDTMR.*

TMR il *voter* deve consistere in un componente molto più affidabile dei tre sistemi a monte.

Nel nostro caso, sui 3 processori presenti nell'architettura vengono eseguiti i *task* replicati che compongono l'applicazione e fanno parte della strategia di ridondanza utilizzata da TMR. Il sistema di *voting*, in EDTMR, è stato implementato a livello software, e consiste in una funzione che riceve in ingresso le tre immagini, le confronta pixel per pixel e seleziona il pixel che risulta uguale per almeno due tra le tre immagini in ingresso. In questo modo, si vuole prevenire un errore, sull'immagine elaborata, causato da un guasto all'interno di uno dei processori.

L'applicazione *multithread*, come mostrato in Figura 4.1, oltre al *voter*, è composta dai tre *task* che hanno il compito di restituire l'immagine digitalizzata e dai *task* di acquisizione e trasmissione dell'immagine. In particolare:

- `read_bitmap`: acquisisce l'immagine dalla periferica e la rende disponibile all'applicazione EDTMR;



(a) Input (b) Output

Figura 4.2: *Immagine in ingresso e risultato in uscita dell'applicazione EDTMR.*

- `rgb2grey`: converte l'immagine RGB in un'immagine a diverse intensità di grigio, eliminando le informazioni sulla tinta e la saturazione del colore e mantenendo la stessa luminosità;

- `edgeDetection`: marca i punti dell'immagine in cui l'intensità luminosa cambia bruscamente. Per farlo, utilizza l'operatore di Sobel, un algoritmo apposito per la tecnica di riconoscimento dei contorni;

- `edgeOverlapping`: elabora l'immagine restituita dalla funzione *edgeDetection* ottenendo un'immagine nella quale i contorni sono evidenziati e facilmente riconoscibili;

- `write_bitmap`: scrive l'immagine ottenuta in un file *bitmap*.

In Figura 4.2 sono mostrate le immagini in ingresso e in uscita al sistema, nel caso di un'esecuzione in assenza di guasto.

Le attività di analisi statica e analisi funzionale, effettuate sull'applicazione EDTMR, ci hanno permesso di individuare quali sono le funzionalità che possiamo considerare zone sensibili del sistema. In particolare, data la natura dell'applicazione, che svolge 3 funzioni in serie su immagini intermedie, ci interessa avere la possibilità di confrontare i dati intermedi in uscita da ogni funzione, durante le simulazioni di guasto. Il modulo di *Function Profiling*, quindi, è stato impostato per salvare le immagini intermedie, nel momento in cui rileva l'evento di uscita da una funzione di elaborazione dell'immagine.

L'attività di esecuzione *Golden*, nella quale abbiamo simulato l'esecuzione in assenza di guasto dei *task* descritti in precedenza e abbiamo salvato le tracce di esecuzione dell'applicazione, ha fornito l'*Execution Flow Graph* mostrato in Figura 4.3. Il grafo descrive con accuratezza gli eventi che compongono l'applicazione EDTMR, mostrando le transizioni tra una funzione e l'altra su ognuno dei tre processori.

4.1.2 Definizione delle strategie di *monitoring* e classificazione

Lo scopo di questo caso di studio è duplice: vogliamo sfruttare la metodologia E-SWEAM per definire nel dettaglio i modi di fallimento del sistema e le loro cause, e vogliamo studiare in quali situazioni l'implementazione di TMR che abbiamo proposto non è sufficiente a garantire un esecuzione corretta dell'applicazione. Il sistema deve essere studiato attentamente in presenza di un guasto al suo interno, per comprendere quali condizioni e quali stati bisogna far emergere nella classificazione dei fallimenti.

Abbiamo quindi specificato, nel *Checker Tool* (CT) di base in Sezione 3.4.2, alcune condizioni specifiche rispetto al contesto e al sistema considerato:

- *flag*: gli eventi principali dell'applicazione, rappresentati dagli ingressi e dalle uscite dei *task* di elaborazione dell'immagine, sono associati a dei *flag* che vengono sollevati nel momento in cui l'interprete di livello applicativo rileva quel determinato evento. In questo modo è facile verificare che il flusso di controllo dell'esecuzione *Golden* sia rispettato anche dalle simulazioni in presenza di guasto. Ad esempio, se un *thread* non esegue la funzione `edgeDetection`, CT segnala questo evento scorretto mostrando che i *flag* di entrata e di uscita dalla funzione `edgeDetection` non sono sollevati. Queste informazioni sono necessarie al classificatore per ricostruire il flusso di controllo dell'applicazione in presenza di errori;

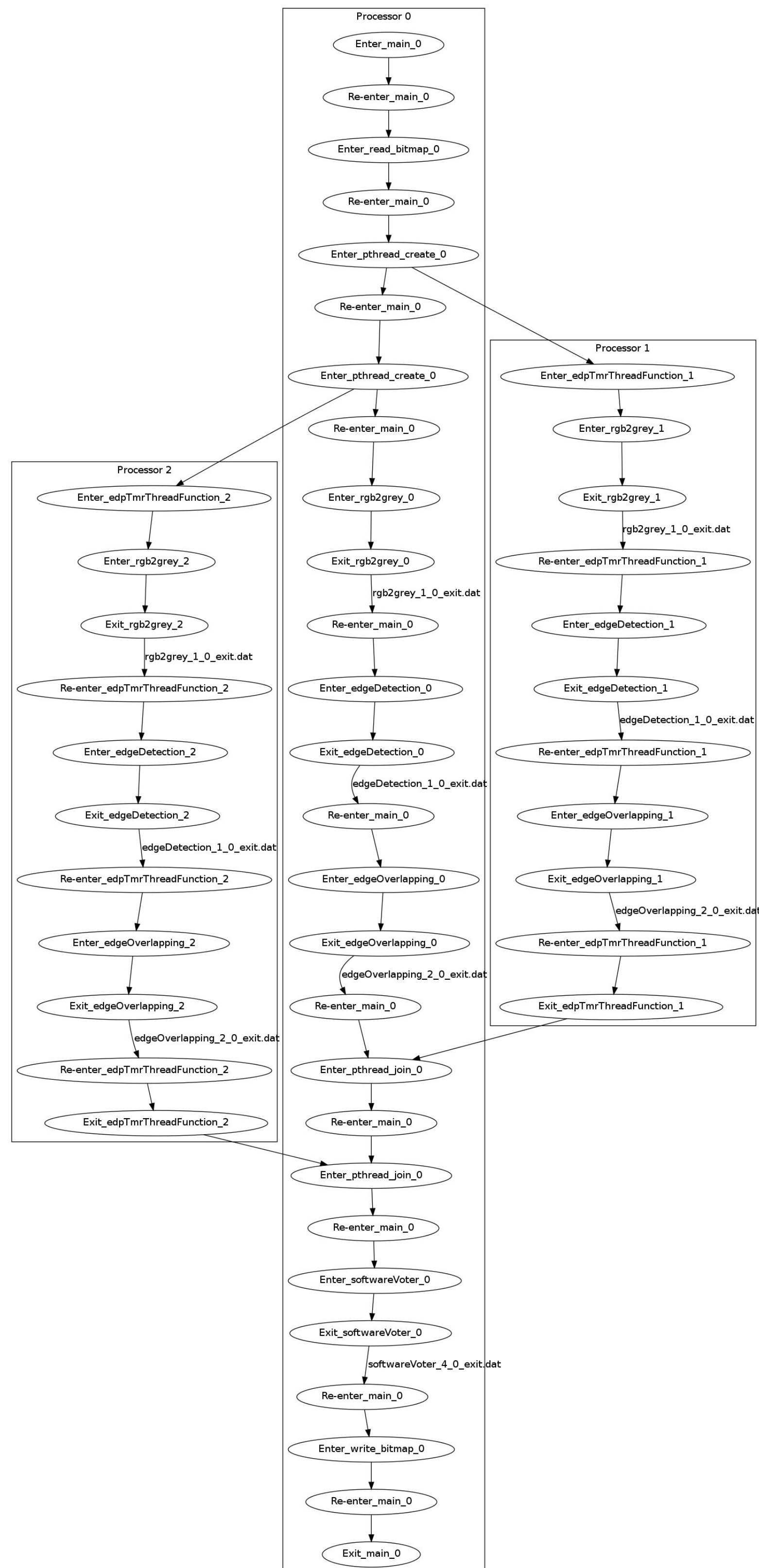


Figura 4.3: *Execution Flow Graph del sistema EDTMR.*

- condizioni sulle immagini in uscita da ogni *task*: nell'EFG di Figura 4.3 sono evidenziati i file contenenti le immagini intermedie elaborate da ciascun *task* di ogni processore. Le tracce dell'immagine *Golden* sono utilizzate dal CT, che le confronta con i risultati intermedi di ogni *task* durante le simulazioni in presenza di guasto. Il confronto avviene pixel a pixel, in modo da catturare la percentuale di immagine errata restituita da un *task*. Su questa percentuale è possibile specificare una soglia, superata la quale l'immagine non è più accettabile. In questo caso, il CT segnala la presenza di un'immagine corrotta in uno dei *thread* dell'applicazione. Questa condizione non pregiudica lo svolgimento dell'applicazione, poiché TMR garantisce il suo compito anche in presenza di un'immagine errata. Nel caso CT rilevi, però, che anche una seconda immagine è stata corrotta, ad esempio da un errore che si è propagato nella memoria e ha modificato i pixel di una delle immagini intermedie utilizzata da un altro processore, la simulazione viene interrotta, poiché TMR è compromesso e non più in grado di garantire un risultato un'azione di *recovery* sulle immagini che osserva;

- *log* delle chiamate a funzione: un *log* aggiuntivo, contenente tutte le chiamate a funzione effettuate da ogni *thread*, viene utilizzato nel caso i *flag* specificati non siano sufficienti a definire l'esatta causa di fallimento dell'applicazione. Dal confronto tra i *flag* sollevati e il contenuto del *log*, il progettista è in grado di comprendere le cause che hanno portato il sistema a fallire. In generale, il *log* è fondamentale nell'interpretazione degli errori sul controllo avvenuti all'interno dell'applicazione.

Delle modifiche attuate nel CT fanno parte anche le condizioni per la classificazione dello stato, fallimentare o meno, del sistema. Il classificatore, come introdotto in Sezione 3.1.6, è un modulo che analizza le informazioni ottenute dalle condizioni impostate durante le simulazioni e mostra lo stato finale del sistema al termine della simulazione. Nel caso di EDTMR, la classificazione finale del sis-

tema dipende dal tipo di risultato ottenuto in uscita. I risultati dell'applicazione sono di due tipi: immagine corretta (o accettabile) e immagine non accettabile.

Le problematiche che siamo interessati ad investigare, però, riguardano il modo in cui il sistema giunge in uno dei due stati finali. In particolare, ci interessa capire perché la tecnica di irrobustimento TMR applicata non è stata in grado di mitigare o, meglio, correggere, la presenza di un errore all'interno di una funzione dell'applicazione. Tratteremo la classificazione proposta nella prossima sezione.

4.1.3 Classificazione dei fallimenti

La classificazione finale (*Correct/Incorrect*) che si può ottenere dall'osservazione dell'immagine in uscita dall'applicazione EDTMR è figlia delle evoluzioni dello stato interno del sistema durante gli esperimenti di iniezione. In questa sezione presentiamo i risultati dell'investigazione del comportamento di EDTMR attraverso l'uso dei diagrammi a stati discussi in Sezione 3.1.6. In Figura 4.4 è mostrato un primo schema di classificazione ricavato dalla caratterizzazione del sistema.

Vogliamo sottolineare che i diagrammi dei fallimenti presentati in questa sezione descrivono il comportamento generale del sistema durante esperimenti di iniezione negli elementi di memoria, che siano essi registri del processore o celle di memoria. In particolare, abbiamo costruito il diagramma basandoci sulle informazioni estratte dagli esperimenti di iniezione nel processore, e abbiamo verificato che le classi evidenziate sono valide anche per gli esperimenti di iniezione della memoria. Nella sezione si parlerà di guasti all'interno dei registri del processore, ma le considerazioni fatte possono essere estese a fallimenti dovuti a guasti in memoria.

Le classi intermedie sono state estratte basandosi sulle informazioni disponibili a livello architettura, applicazione e sulle condizioni specificate nel CT, durante il passo precedente. In particolare, consideriamo individuati (*Detected*) gli errori dei quali si accorge il sistema stesso, e ai quali può reagire, ad esempio con il sollevamento di un'eccezione nel momento in cui viene richiesto un'indirizzo non

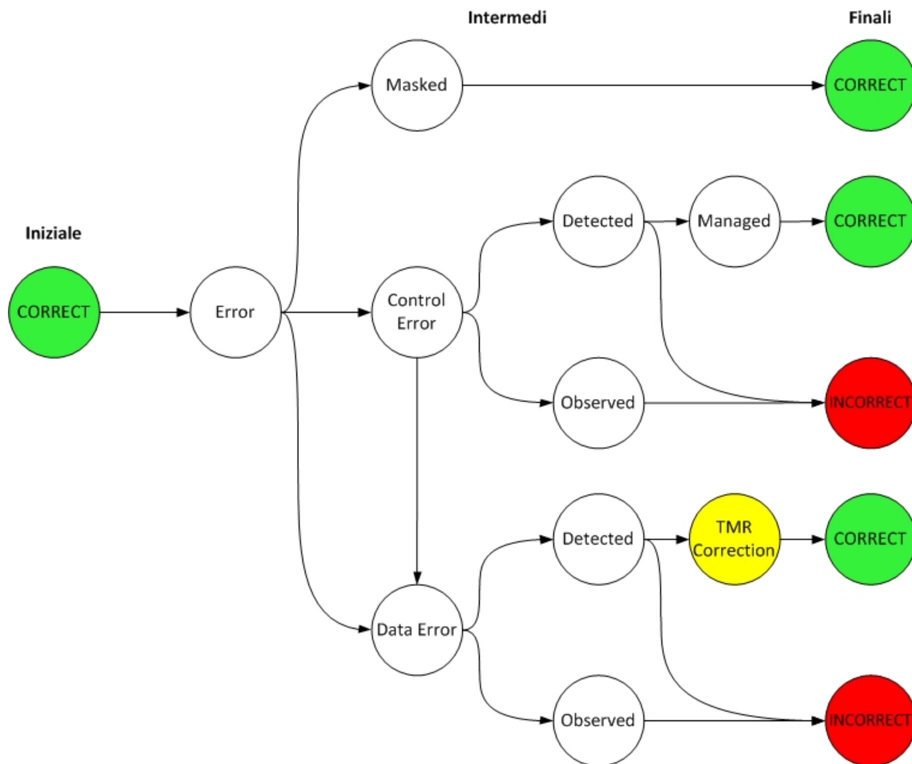


Figura 4.4: *Classificazione iniziale degli stati del sistema EDTMR.*

presente in memoria. La classe intermedia di errori individuati distingue tra un errore sul controllo e un errore sui dati. Nel primo caso, rientrano quei casi descritti in precedenza, gestiti attraverso il lancio di un'eccezione causata da una *interrupt* hardware o software. Questa azione provoca la terminazione anticipata dell'applicazione, che non è in grado di portare a termine l'elaborazione dell'immagine o il *task* di votazione, e quindi non restituisce un'immagine corretta in uscita. Nel secondo caso, invece, un errore sui dati correttamente individuato porta all'intervento della funzione `softwareVoter` discussa in precedenza, che ha il compito di scartare l'immagine che presenta dei pixel sbagliati e restituire l'immagine ottenuta dall'elaborazione degli altri due *thread*. Lo stato *Masked* rappresenta il caso in cui un *thread* è in grado di mascherare l'effetto di un guasto al proprio interno, risultando nell'esecuzione corretta dell'applicazione e nella restituzione di un'immagine senza pixel errati.

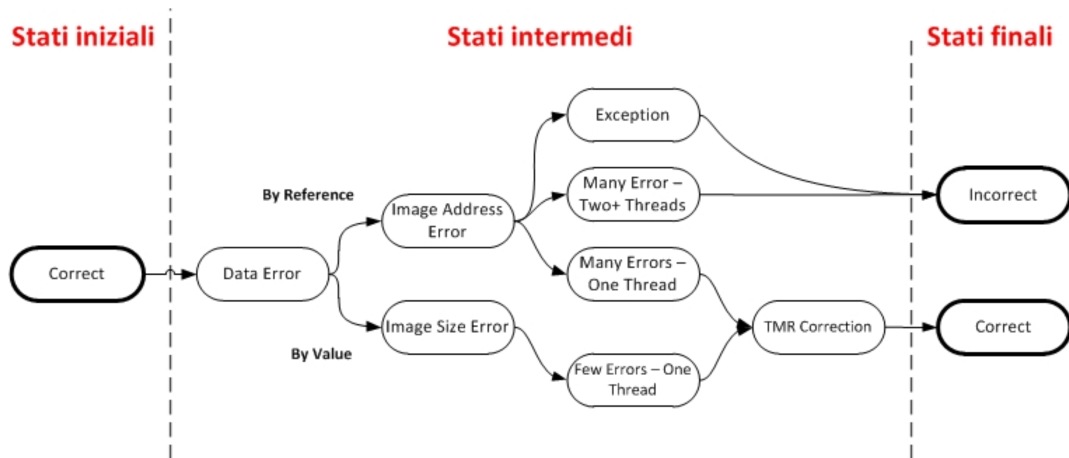


Figura 4.5: *Classificazione dei fallimenti causati da errori sui dati.*

Dal diagramma di Figura 4.4 è possibile notare immediatamente due cose, che verrano discusse in modo ampio nella sezione di analisi dei risultati delle campagne: solo gli errori sui dati prevedono una correzione attraverso la tecnica TMR, fatto, questo, che limita già in partenza l'efficacia di questa tecnica all'interno di un sistema complesso dove risulta molto critico il flusso di controllo dell'applicazione. In secondo luogo, allargando il discorso, non abbiamo applicato una strategia software efficace per la mitigazione degli errori che corrompono la corretta esecuzione dell'applicazione. Esiste uno stato intermedio *Managed* che rappresenta un caso particolare in cui il gestore di eccezioni del sistema effettua una riesecuzione dopo aver individuato un errore. Terremo in considerazione queste informazioni aggiuntive durante la discussione sui risultati ottenuti con la campagna di iniezione dei guasti.

In Figura 4.5 sono mostrati gli stati intermedi aggiuntivi ricavati dall'applicazione di E-SWEAM per quanto riguarda gli errori sui dati. L'analisi delle immagini intermedie, attraverso il nostro *framework* di analisi, ci consente di capire fino a che livello si è propagato un errore sui pixel di un'immagine. Abbiamo distinto due classi, che simboleggiano la percentuale di pixel corrotti all'interno dell'immagine restituita da ogni *task* dell'applicazione. Attraverso le condizioni specificate nel CT, siamo in grado di estrarre il numero di pixel errati dell'immagine. L'utente

può definire una soglia di accettabilità per la propria immagine. In questo modo è possibile stabilire se, nonostante alcuni errori sui pixel, l'immagine può comunque essere utilizzabile. Nel nostro particolare caso, questa condizione è relativamente utile, poiché sia che ci siano pochi pixel errati, sia che ce ne siano molti, il TMR necessita di due immagini corrette per effettuare il proprio compito. Il problema riguarda, semmai, molteplici errori su due delle tre immagini elaborate. Sottolineiamo che, nel caso di pochi errori su più immagini, il *voter* corre il rischio di restituire un'immagine contenente troppi errori per essere accettata. Questo evento, comunque, è un caso molto particolare, e non si è mai verificato durante i nostri esperimenti, anche se esiste una minima probabilità che avvenga.

Vengono distinti, per la loro diversa natura, gli errori che affliggono l'indirizzo di un'immagine (che è contenuta in un *array*) e gli errori che coinvolgono una variabile passata per valore. Nei *task* considerati, i parametri passati per valore, che abbiamo avuto modo di corrompere durante gli esperimenti di iniezione, contenevano le dimensioni, altezza e larghezza, dell'immagine. Gli indirizzi, invece, riguardano le immagini in ingresso ed in uscita ad una funzione. Per questo motivo, nel primo caso gli errori sui pixel si limitano a rimanere sotto la soglia che abbiamo imposto, e per questo motivo si parla di stato *few errors*, che in sé non rende l'immagine non accettabile. Al contrario, nel caso di corruzione di un indirizzo, l'immagine risulterà completamente errata, portando il sistema in uno stato dove vengono osservati *many errors*. Entrambi questi stati sono gestiti, se contenuti all'interno di un solo *thread*, dal meccanismo software TMR. L'unico caso in cui è possibile che un errore sui dati comprometta il funzionamento del meccanismo TMR è nel caso in cui dei pixel errati vengano propagati all'immagine elaborata da un secondo *thread*. Questo caso particolare è frutto della natura del sistema considerato, nel quale l'esecuzione dei *thread* non è adeguatamente separata, e nella quale la memoria non è dotata di meccanismi di protezione che assicurino che dati letti da un *thread* non siano sporcati da scritture eseguite da

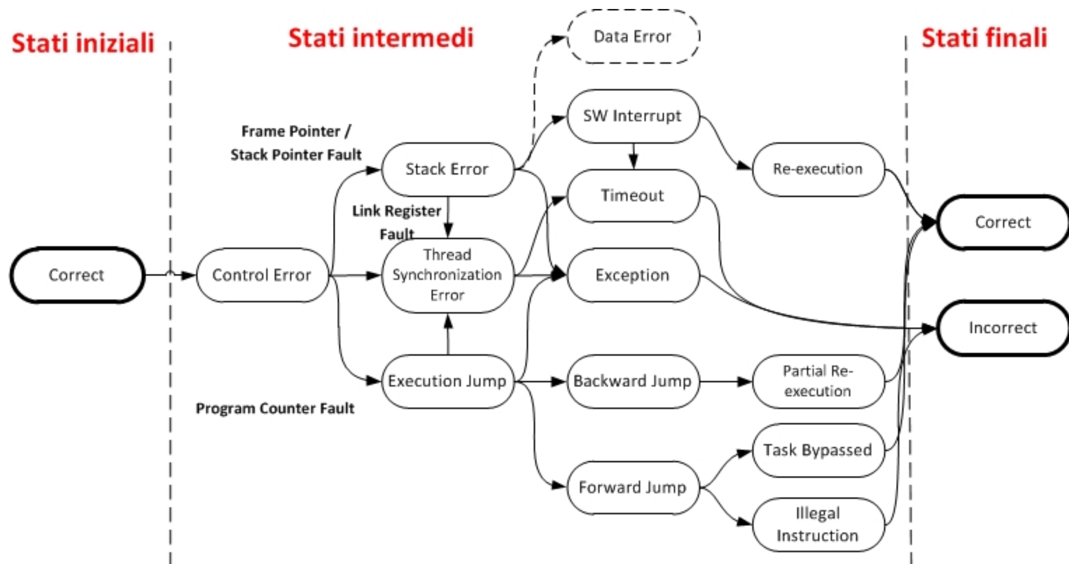


Figura 4.6: *Classificazione dei fallimenti causati da errori sul flusso di controllo dell'esecuzione.*

altri. Il corrompimento della memoria, causato dalla scrittura di dati errati da parte di un *thread* eseguito da un processore guasto, diffonde l'errore, che dovrebbe essere contenuto all'interno di un *thread*, ai *thread* concorrenti. Questa situazione porta il *voter* ad effettuare il proprio compito su almeno due immagini corrotte, con il conseguente rischio di restituire, in uscita, un'immagine scorretta.

In Figura 4.6, invece, sono mostrati gli stati intermedi relativi a errori sul flusso di controllo, e le loro relazioni con gli stati finali. Gli errori sul controllo sono causati da guasti che corrompono le funzionalità relative alla corretta esecuzione dell'applicazione, come i registri contenenti il *Program Counter* (PC), il *Link Register* (LR) e lo *Stack Pointer* (SP). Come si può vedere dal diagramma, a parte un caso particolare, gli stati intermedi del sistema indotti da un errore sul controllo portano inevitabilmente al fallimento dell'applicazione e alla restituzione di un'immagine scorretta.

In tutti i casi è possibile che il sistema stesso si accorga di un errore in questi registri lanciando un'eccezione. In questo caso, come detto in precedenza, l'esecuzione termina in modo anticipato, non consentendo al *voter* di selezionare l'immagine corretta da restituire in uscita. Nel caso di guasto di SP, poi, può

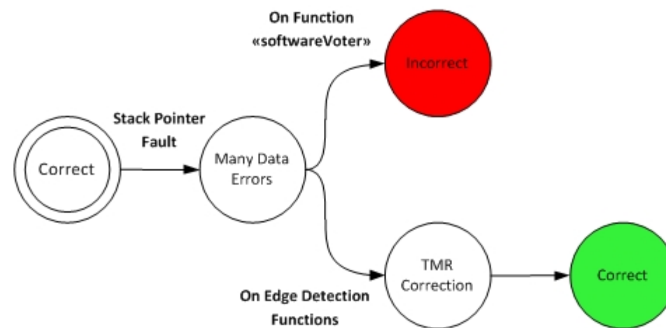


Figura 4.7: *Caso particolare di errori sui dati causato dal guasto dello Stack Pointer.*

succedere che il sistema lanci un'*interrupt* software per la gestione del guasto. In questo caso, l'*interrupt* può portare ad un errore che causa un *Timeout* all'interno dell'applicazione, che non riesce a portare a termine la gestione dell'eccezione. Il *Timeout* viene osservato attraverso la segnalazione di un *watchdog* all'interno dell'architettura: scaduto un determinato intervallo temporale, la simulazione viene interrotta perché il sistema non è in grado di portarla a termine. Altrimenti, la gestione della *interrupt* va a buon fine, l'applicazione viene rieseguita dall'inizio sul processore, il cui SP è stato guastato, e si riesce a eseguire la funzionalità di voto finale.

Infine, un guasto di SP può causare un errore sui dati nella funzione immediatamente successiva all'osservazione dell'errore. Questo caso particolare è mostrato in Figura 4.7. In particolare, questo fallimento può portare ad uno stato finale scorretto se il guasto nello SP avviene immediatamente prima della funzione di *voting*, caso nel quale l'immagine finale votata risulta scorretta. In tutti gli altri casi, invece, questo errore viene correttamente gestito dal meccanismo di TMR.

Nel caso di guasto al registro LR non mascherato, invece, sono possibili due sole evoluzioni dello stato: il lancio di un'eccezione oppure un errore di sincronizzazione tra i *thread* che porta ad un fallimento *Timeout*. LR, infatti, contiene l'indirizzo di ritorno alla funzione chiamante, e la conseguenza principale della corruzione di tale indirizzo è il mancato ritorno di una funzione al proprio chiamante. In questo caso, il *thread* rimane "appeso", non consentendo la corretta terminazione degli

altri due *thread*.

Il guasto di PC causa un “salto” nell’esecuzione del *thread*, che può essere in avanti o all’indietro. Un PC non valido, inoltre, può causare il blocco dell’esecuzione del *thread*, che genera un errore di sincronizzazione tra i *thread* e quindi uno stato *Timeout*. In generale, i salti che non bloccano l’esecuzione non sono dannosi per il risultato finale di EDTMR, poiché quello che può succedere è la non esecuzione o la riesecuzione di un determinato *task*. Nel primo caso, se un *task* viene saltato totalmente o in parte, saranno presenti errori sui dati in quel determinato *thread*, che saranno gestiti correttamente dalla funzione di *voting*. Nel secondo caso, invece, l’unico effetto riguarda il tempo di esecuzione, leggermente maggiore.

Possiamo riassumere gli stati iniziali, intermedi e finali appena descritti attraverso il diagramma presentato all’inizio della sezione, opportunamente decorato con i casi particolari di ogni stato intermedio. In Figura 4.8 viene mostrato il risultato.

Per una maggiore chiarezza, abbiamo collassato alcuni fallimenti intermedi in un unico stato. Ad esempio, lo stato *Exception* raggiungibile sia dagli stati di errori sui dati che dagli stati di errori sul controllo ovviamente si riferisce a eccezioni diverse. Lo stato di *Timeout* è è causato di un errore di sincronizzazione dei *thread*, che a sua volta può essere causato da un guasto nel registro contenente il *Link Register*, oppure da errori di controllo derivanti da guasti nei registri contenenti *Stack Pointer* o *Program Counter*. Infine, lo stato di di “correzione” effettuata dal TMR può portare a due risultati, in base al tipo di errore che si è verificato a monte di esso: l’immagine sarà corretta se si sono verificati errori sui pixel in un solo *thread*, oppure sarà scorretta se si saranno verificati errori su più di un *thread*.

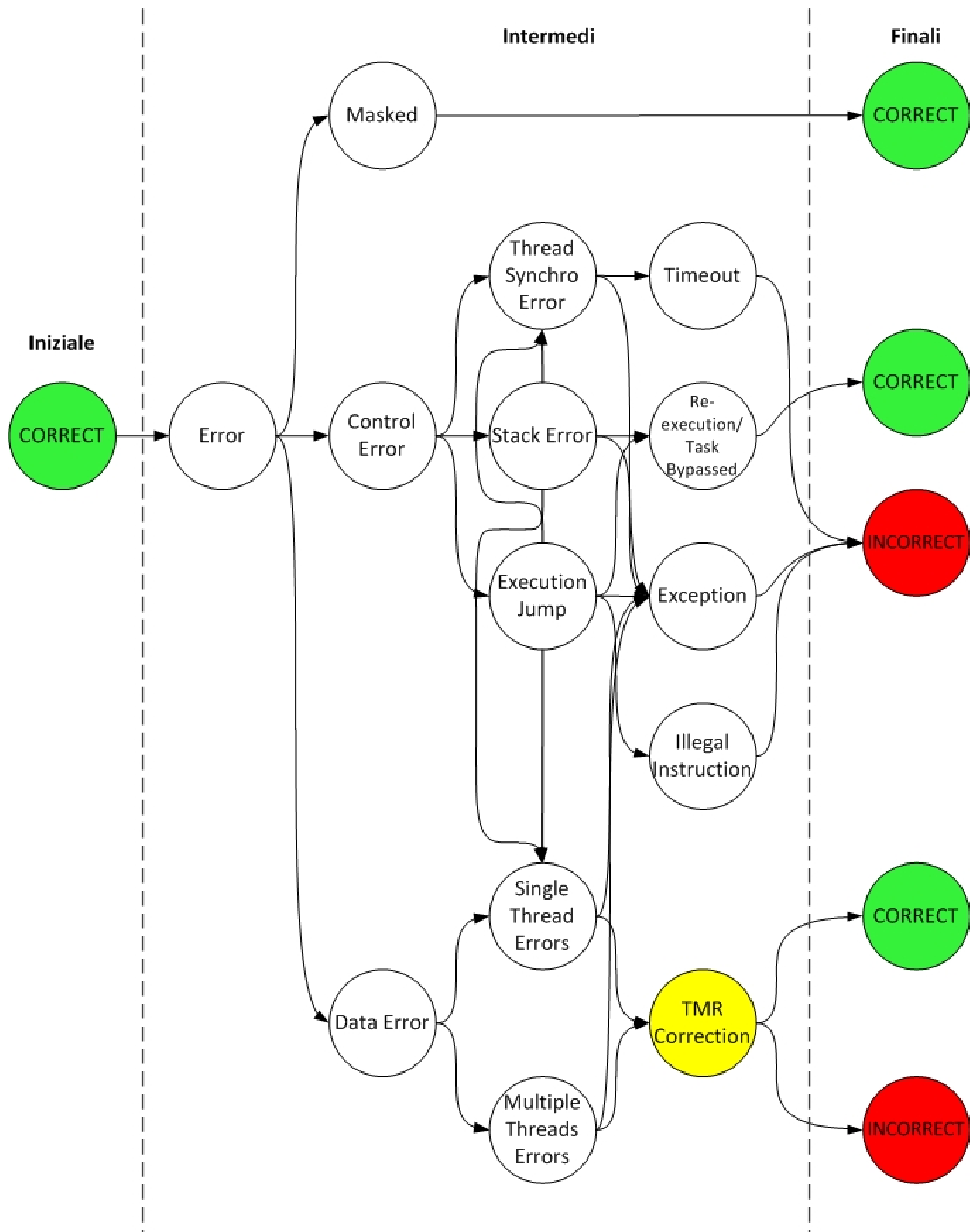


Figura 4.8: *Classificazione completa di tutti i fallimenti del caso di studio EDTMR.*

4.1.4 Analisi dei risultati della campagna di iniezione dei guasti

La campagna di iniezione dei guasti svolta per questo caso di studio è consistita nell'iniezione di 10000 guasti all'interno del processore, in zone sensibili diverse, estratte dall'analisi dell'architettura incrociata con l'analisi della *liveness* dei registri del processore, e nell'iniezione di 10000 guasti nelle celle di memoria. Anche in quest'ultimo caso, abbiamo calcolato la *liveness* delle celle per garantire una maggiore probabilità di attivazione degli errori in seguito all'iniezione di un guasto, cercando di fornire risultati più significativi rispetto ad una campagna di iniezioni a tappeto, che ha il rischio di restituire, per la maggior parte, risultati *Masked*. Il tempo totale di esecuzione, richiesto da questa campagna, è stato di circa 120 ore, dovute alla complessità dell'elaborazione dell'immagine, che richiede in media 30 secondi per ogni esperimento.

In Appendice A sono mostrate le tabelle complete relative alle percentuali di fallimenti riscontrati durante la campagna. Rimandiamo il lettore alla consultazione di tali tabelle per un'analisi completa dei risultati della campagna. In questa sezione, saranno discussi i risultati più interessanti, e saranno descritte le problematiche del sistema in esame, con particolare attenzione alla tecnica di irrobustimento scelta.

I dati più significativi, per quanto riguarda le iniezioni nel processore, riguardano le percentuali di immagini scorrette che si ottengono nei casi di errori sui dati e nei casi di errori sul flusso di controllo. In Tabella 4.1 sono mostrati tali valore: è possibile notare come le funzionalità più critiche, in questo caso di studio, siano rappresentate dai registri contenenti il *Link Register* e il *Program Counter*, mentre, nel caso di variabili passate per valore, circa nel 10% dei casi si ottiene un'immagine non accettabile a causa della terminazione anticipata dell'applicazione dovuta al lancio di un'eccezione. Sottolineiamo, inoltre, l'andamento dei fallimenti di tipo *Timeout*. Se, durante le iniezioni nello *Stack Pointer*, i casi di

Funzionalità	Thread Principale	Threads Secondari
Variabile - Riferimento	34,6%	38,7%
Variabile - Valore	12,4%	10,9%
Link Register	78,0%	73,4%
Stack Pointer	52,8%	56,2%
Program Counter	76,4%	75,5%

Tabella 4.1: *Percentuali di immagini scorrette per ogni funzionalità guastata.*

Timeout rilevati sono stati di circa il 10%, si è raggiunto il 25% nel caso di guasto del *Program Counter*, e con un valore intorno al 40% nel caso di guasto del *Link Register* nel *thread* principale dell'applicazione.

Il sistema EDTMR, quindi, è fortemente penalizzato nel caso di un errore sul flusso di controllo dell'applicazione. I frequenti casi di lancio di eccezioni o di errori di sincronizzazione tra i *thread*, che causano un *timeout*, non consentono al *voter*, implementato nel software, di svolgere il proprio lavoro in modo corretto. È significativa anche la percentuale di eccezioni sollevate nel caso di guasto in un registro contenente una variabile passata per riferimento. Circa nel 35% dei casi, infatti, l'eccezione sollevata causa la terminazione scorretta dell'applicazione, che non esegue la funzionalità di irrobustimento introdotta.

Dall'analisi dei dati relativi ai fallimenti causati da guasti nella memoria, presentati in Tabella 4.2, si evince come la memoria sia soggetta, anch'essa, a guasti che possono causare fallimenti critici. La percentuale maggiore di risultati consiste in guasti mascherati dal sistema, che non hanno ripercussioni sul risultato finale. Questo dato è giustificato dal fatto che, visto l'ordine di grandezza della memoria, esiste una probabilità molto inferiore che un guasto possa causare un errore osservabile a livello applicazione. Guasti delle celle di memoria, però, causano circa il 20% di errori sui dati. Tra essi, circa il 18% è rappresentato da singoli errori sui pixel dell'immagine, mentre solo il 2% ha coinvolto più pixel della stessa immagine. Il primo caso è giustificabile dal fatto che, con buona probabilità, un guasto in memoria causa un errore su un pixel di un'immagine intermedia elabo-

Iniezioni in memoria - EDTMR		
Correct	Masked	73,25%
	Few Errors	19,42%
	Many Errors	2,05%
	TOTALE	**94,72%**
Incorrect	Exception	4,71%
	Timeout	0,35%
	Application Abort	0,22%
	TOTALE	**5,28%**

Tabella 4.2: *Risultati delle iniezioni nelle celle di memoria del sistema EDTMR.*

rata dall'applicazione. Nel secondo caso, invece, potrebbe essere stato corrotto l'indirizzo di un immagine intermedia oppure potrebbero essersi propagati errori sui pixel all'interno delle stesse immagini intermedie. Tutti gli errori sui dati sono correttamente gestiti dal TMR implementato. Circa il 5% dei fallimenti, infine, fanno parte della tipologia di errori sul controllo. Tra di essi, la maggior parte è rappresentata da fallimenti causati dal lancio di un'eccezione, mentre sono stati rilevati sporadici casi di *Timeout* o terminazione anomala dell'applicazione.

I risultati della memoria sottolineano come anche questo elemento del sistema necessiti di irrobustimento ulteriore, per evitare che guasti delle celle causino il fallimento dell'intera applicazione, nonostante questa probabilità sia molto più bassa rispetto alla probabilità di fallimenti critici causati da guasti del processore.

Dai risultati presentati, emerge l'impossibilità del meccanismo TMR, implementato a livello software nel sistema, di gestire gli errori sul flusso di controllo dell'applicazione. Il *task* che effettua la votazione, eseguito dopo la terminazione delle elaborazioni delle immagini sui diversi *thread*, nei casi molto comuni di lancio di eccezioni da parte del sistema, oppure di esecuzione "appesa" a causa di un errore di sincronizzazione tra i *thread*, non può svolgere in modo corretto il proprio compito. Questo si traduce nell'impossibilità di giungere in uno stato finale che produca un risultato corretto, a causa della natura bloccante dei fallimenti

discussi. E-SWEAM, quindi, evidenzia come l'implementazione di TMR adottata fallisca proprio perché, nell'applicazione in esame, le tre repliche del programma di *edge detection* interagiscono tra di loro, e il fallimento di una delle tre esecuzioni si traduce in un fallimento globale del sistema. Una tecnica di irrobustimento che dipende, per la propria esecuzione, dalla correttezza dello stato del sistema a monte di essa non può garantire la correzione degli errori indotti da un guasto dei componenti hardware del sistema.

Per ovviare a questi problemi, sono possibili almeno due soluzioni. In particolare, la prima prevede il disaccoppiamento del *voter* dal sistema, in modo tale che non sia oggetto di errori di sincronizzazione a monte rispetto all'istante in cui viene richiesta la sua funzionalità, mentre la seconda prevede una strategia di gestione dei diversi *thread* in ambienti totalmente separati, così da prevenire fallimenti causati dal lancio di eccezioni che bloccano l'esecuzione dell'intera applicazione.

La prima modifica attuabile prevede l'introduzione, ad esempio, di un *voter* hardware al posto della funzionalità software adibita al confronto delle immagini in uscita dai tre diversi *thread*. Il *voter* hardware può essere impostato per cominciare la propria esecuzione allo scadere di un determinato istante temporale. In una situazione nella quale l'esecuzione è momentaneamente sospesa a causa di un *thread* che non riesce a ritornare correttamente, dagli esperimenti effettuati abbiamo constatato come le esecuzioni portate a termine dagli altri *thread* siano sempre corrette. In realtà, quindi, le immagini in ingresso necessarie al dispositivo di *voting* sono a disposizione del *voter* non appena l'elaborazione sul rispettivo *thread* è terminata. Il *voter* hardware, quindi, può ricevere in ingresso due immagini corrette e l'immagine scorretta frutto del *thread* eseguito sul processore guasto. In questo modo, è in grado di restituire in uscita un'immagine corretta nonostante il verificarsi di un errore sul controllo in grado di pregiudicare la corretta esecuzione dell'applicazione.

Questa leggera modifica è stata introdotta nel sistema EDTMR in analisi, con il risultato di verificare in quali casi gli errori di sincronizzazione tra i *thread* possano essere mitigati, consentendo la restituzione di un'immagine corretta in uscita. Dai risultati della campagna, emerge come il 14% circa di tutti i guasti iniettati nel processore siano risultati in un fallimento che ha portato l'applicazione in uno stato *Timeout*. Le analisi effettuate con la modifica del sistema, consistente nell'utilizzo di un *voter* slegato dall'applicazione EDTMR, ha permesso di verificare che circa il 96% di questi fallimenti in realtà possono essere corretti dall'utilizzo di questa soluzione. In questa maggioranza di casi, infatti, l'applicazione termina in uno stato nel quale due dei tre *thread* sono riusciti a portare a termine le proprie elaborazioni, e sono in grado di restituire immagini corrette, condizione sufficiente a garantire che il *voter* sia in grado di selezionare l'immagine non corrotta.

Ciò che la sola introduzione di un *voter* disaccoppiato, al posto del *task* software, non sarebbe in grado di gestire in modo corretto, è l'insorgere di una *interrupt*, hardware o software, che causa il lancio di un'eccezione da parte del sistema stesso. È necessario che ogni *thread* sia in grado di eseguire in modo indipendente, e che i dati intermedi siano salvati in proprie aree di memorie alle quali gli altri *thread* non possono avere accesso, così da evitare problemi come quello discusso, in cui un *thread* che lavora su un'immagine corrotta "sporca" l'area di memoria dal quale un altro *thread* legge l'immagine. In aggiunta a questo, è necessario implementare un meccanismo hardware di protezione della memoria, poiché, ad esempio, lo *stack* è salvato in essa. Queste soluzioni hanno lo scopo di irrobustire ulteriormente un sistema che, con l'implementazione attuale del meccanismo di TMR a livello software, è in grado di gestire in modo efficace errori sui dati, ma è prono alla maggior parte degli errori che coinvolgono il flusso di controllo dell'applicazione.

4.2 Caso di studio *Anti-lock Braking System*

In questa sezione viene presentato il caso di studio relativo all'*Anti-lock Braking System*. Similmente a quanto presentato prima, per il precedente caso di studio, sarà fornita una breve descrizione dell'architettura e del contesto nella quale opera.

Saranno, quindi, presentate le strategie per la caratterizzazione del sistema, per la definizione del *monitoring* e della classificazione, e saranno discussi i risultati finali ottenuti dall'applicazione di E-SWEAM.

4.2.1 Caratterizzazione del sistema

Il caso di studio dell'*Anti-lock Braking System* (ABS) analizza il comportamento di un sistema *embedded* che simula il comportamento del dispositivo di ABS, comune alla maggior parte dei veicoli su ruote in commercio. Il sistema, come si vede in Figura 4.9, è composto da due processori ARM9, che hanno il compito di elaborare i dati della vettura per regolare la frenata, dalla memoria e dal bus; inoltre, sono presenti dei sensori che si occupano di rilevare i dati sulla velocità del veicolo e sulla velocità di rotazione delle singole ruote e, infine, un attuatore, che ha il compito di regolare la forza frenante applicata sui freni, ed è controllato direttamente dai processori. Per comodità, i sensori e l'attuatore sono stati rappresentati con un singolo modulo.

Il sistema esegue un'applicazione *multi-thread*. La Figura 4.9 mostra i tre *task* nei quali è suddivisa l'applicazione. Quest'ultima è composta da un ciclo che esegue le tre funzionalità in sequenza, fino a che la macchina non è ferma, cioè finché la velocità non è pari a zero. Il primo *task* (suddiviso in tre diverse chiamate a funzione) rappresenta la richiesta e la ricezione dei dati dai sensori del veicolo, che trasmettono, nell'ordine: la velocità del veicolo, la forza frenante attualmente impiegata sui freni e la velocità di rotazione delle singole ruote. Dovendo impedire che la velocità di rotazione delle ruote sia ridotta a zero, l'applicazione, ad ogni ciclo, regola la forza frenante applicata sui freni. Per attuare la regolazione,

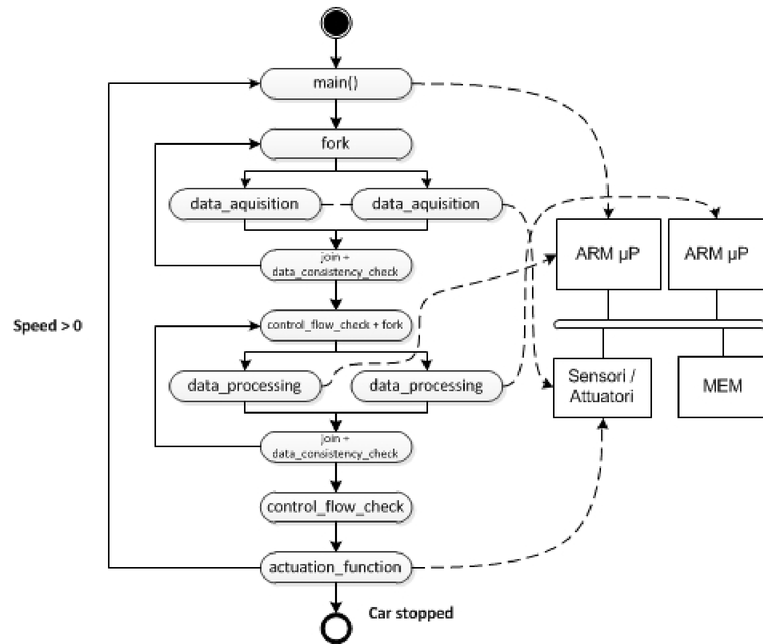


Figura 4.9: *Modello del sistema utilizzato per il caso di studio ABS.*

effettua dei calcoli sui parametri ricevuti dai sensori, che sono rappresentati dal *task* intermedio `data_processing` (in particolare, questo *task* è composto da due funzioni distinte). Infine, i dati calcolati durante quest'ultimo *task* sono trasmessi all'attuatore, che ha il compito di regolare la forza frenante.

Il modello viene classificato come sistema di *safety*. Per questa ragione, alcuni *task* sono stati duplicati ed eseguiti in parallelo e il codice è stato irrobustito in modo che possano venire garantite, anche in seguito ad un guasto, le funzionalità del dispositivo. Per irrobustire il codice sono state adottate le tecniche descritte da Rebaudengo et al. in [40]. L'approccio nell'articolo sfrutta diverse trasformazioni del codice che garantiscono sia l'individuazione di un guasto che, in alcuni casi, la correzione dell'errore.

In particolare, per garantire la correttezza dell'elaborazione sui dati, successivamente ad ogni funzione all'interno dei *task* che si occupano dell'acquisizione e dell'elaborazione dei dati stessi, è stato inserito un frammento di codice che ha lo scopo di confrontare i risultati ottenuti dalle funzioni eseguite in parallelo. Se viene individuata una discrepanza tra i dati in uscita dalle funzioni, viene chiamata

la funzione di *safety* `signal_data_error`, che segnala l'individuazione di un errore sui dati. Inoltre, la `signal_data_error` ha il compito di far rieseguire le funzioni parallele nelle quali si è verificato l'errore. Consideriamo, nei nostri casi di studio, solo guasti di tipo transitorio. Per questo motivo, la scelta di rieseguire la funzione che ha corrotto i dati riteniamo sia sufficiente a garantire la robustezza necessaria alla corretta gestione degli errori sui dati all'interno di questa applicazione.

Per garantire, invece, la correttezza dell'esecuzione dal punto di vista del flusso di controllo, è previsto uno schema a "firme". Ad ogni funzione viene passata, in ingresso, una variabile che la funzione deve firmare con il proprio identificatore (ad esempio, un numero). All'uscita dalla funzione, un frammento di codice ha il compito di verificare che la variabile sia firmata con l'identificatore corretto, cioè quello della funzione alla quale è appena stata passata. In questo modo è possibile capire se la funzione precedente è stata correttamente eseguita, e rilevare, quindi, eventuali anomalie nel flusso di esecuzione. In caso di errori, vista la criticità del sistema, la soluzione adottata prevede l'esecuzione di un "ciclo di frenata" alternativo, rappresentato dalla funzione di *safety* `emergency_brake`. Questa funzione acquisisce nuovamente i dati dai sensori, li rielabora e si occupa di trasmetterli, infine, all'attuatore. Al termine dell'esecuzione di `emergency_brake`, il controllo ritorna alle funzioni dell'ABS, che continuano l'esecuzione così com'era prevista.

Infine, è stato implementato un ultimo meccanismo di *safety* per le situazioni più critiche. In caso i due meccanismi precedenti falliscano nel portare il sistema in uno stato nel quale è possibile concludere l'esecuzione della frenata con successo, un meccanismo di *rollback & replay* (R&R), simile a quello descritto in Sezione 2.3.7, riporta il sistema in uno stato iniziale prima della propagazione dell'errore, e ha il compito di eseguire nuovamente la frenata. Avendo a che fare con un dispositivo di *safety* in un contesto *time-critical*, questo meccanismo necessita della presenza di un *timer* che sia in grado di monitorare il tempo di esecuzione dell'applicazione. In Sezione 4.2.2 vedremo nel dettaglio in che modo abbiamo

```
16587 Enter main 0
19133 Enter receive_car_sensor_data 0
19695 Exit receive_car_sensor_data 0
19916 Re-enter main 0
20312 Enter receive_brake_sensor_data 0
20874 Exit receive_brake_sensor_data 0
21095 Re-enter main 0
22839 Enter pthread_create 0
22882 Enter sensor_thread_function_wrapper 1
22922 Re-enter main 0
23322 Enter receive_wheel_sensor_data 0
24802 Enter receive_wheel_sensor_data 1
33122 Exit receive_wheel_sensor_data 0
33402 Re-enter main 0
33802 Enter pthread_join 0
```

Figura 4.10: *Sezione della traccia salvata dall'esecuzione Golden di ABS.*

implementato il *timer* a supporto del meccanismo di R&R.

Le attività di analisi dell'applicazione, dell'architettura e di caratterizzazione *Golden* permettono, attraverso il salvataggio delle tracce dell'esecuzione *Golden* delle funzioni, come quella mostrata in Figura 4.10, di costruire l'*Execution Flow Graph* mostrato in Figura 4.11.

L'EFG, generato dall'analisi delle tracce dell'applicazione, mostra un ciclo di esecuzione dell'applicazione ABS, a partire dalle funzioni che si occupano dell'acquisizione dei dati dai sensori, per passare alle funzioni di elaborazione, concludendo con la funzione di attuazione. Sono mostrate, nel grafo, le *fork* e le *join* relative alle funzioni che vengono eseguite in parallelo sui due processori. Nell'EFG è possibile, inoltre, osservare gli eventi di *Function Enter/Re-enter/Exit* associati all'applicazione ABS. Dal grafo è possibile stabilire il flusso corretto di esecuzione, che è fondamentale nella pianificazione delle strategie di *monitoring* e classificazione discusse nella prossima sezione. In questo particolare caso di studio, i dati intermedi *checkpointed* non sono sfruttabili come nel caso precedente di elaborazione di un'immagine, nel quale la strategia di *monitoring* prevedeva il confronto tra le immagini intermedie in assenza e in presenza di guasto. Nel caso di ABS, un solo errore sui dati o un'esecuzione della funzione `emergency_brake` causerebbero la divergenza nel confronto tra la traccia della simulazione di guasto

e la traccia *Golden*, rendendo inutile questa tecnica.

4.2.2 Definizione delle strategie di *monitoring* e classificazione

Partendo dalla caratterizzazione del sistema, dal contesto di *safety* considerato e dal *Checker Tool* (CT) presentato in Sezione 3.4.2, abbiamo definito strategie di *monitoring* e classificazione personalizzate per il sistema in esame.

Il CT di ABS, in particolare, è stato modificato con i seguenti meccanismi, per consentire l'osservazione di errori nel contesto considerato:

- *flag* che segnalano l'ultima funzione eseguita da un processore, prima della terminazione della simulazione;

- condizioni sulle chiamate alle funzioni di *safety*, che ci consentono di studiare le cause delle chiamate a tali funzioni. Ad esempio, è possibile comprendere se una funzione è stata effettivamente chiamata quando serviva o meno. Le funzioni di *safety* monitorate sono `signal_data_error` e `emergency_brake`, descritte in Sezione 4.2.1;

- un contatore nel quale vengono accumulate le chiamate alle funzioni di *safety* `signal_data_error`. Al contatore è associata una soglia. Se esso la supera, significa che la simulazione si trova in uno stato di fallimento irreversibile, poiché o i dati sono scorretti e la funzione di *safety* non è in grado di porre rimedio a questo errore, oppure il numero eccessivo di ripetizioni di determinati *task* allunga i tempi di esecuzione in modo da non consentire la terminazione della frenata nei tempi prestabiliti;

- il *timer* di sistema descritto precedentemente che, nel momento in cui CT osserva un errore, viene monitorato attivamente dal *framework*. Il tempo registrato dal *timer*, oltre a essere funzionale al meccanismo di R&R, è una

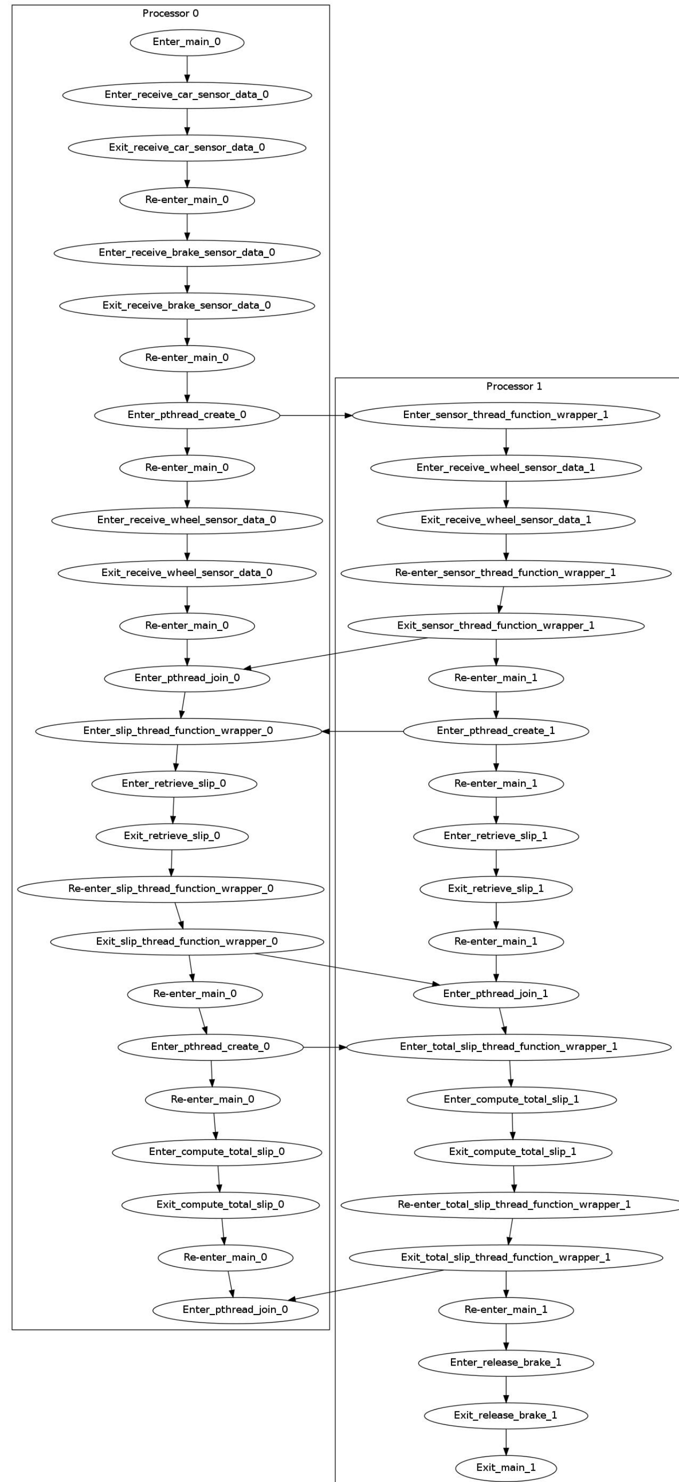


Figura 4.11: *Execution Flow Graph generato durante il caso di studio ABS.*

variabile utilizzata dal classificatore per stabilire la classe finale del sistema durante l'esperimento;

- un *log* contenente tutti gli eventi di *Function Enter/Exit/Re-enter* eseguiti durante la simulazione. Questo strumento viene utilizzato come una sorta di *debugger*: viene consultato quando il CT ha raccolto una serie di dati sull'esperimento che non sono sufficienti a stabilire la classe di fallimento all'interno della quale si trova il sistema. L'osservazione del flusso di eventi permette di analizzare il comportamento dell'applicazione e definire condizioni più stringenti per la classificazione del fallimento.

Il classificatore, in modo incrementale ad ogni evento osservato, classifica lo stato del sistema in base alle informazioni ottenute dalle condizioni appena descritte. In particolare, la verifica di alcune di esse consente di predire lo stato finale del sistema, e ci permette di interrompere preventivamente la simulazione e verificare la possibilità o meno dell'impiego del meccanismo di R&R. In seguito a quest'ultima verifica, è possibile effettuare la classificazione finale dell'esperimento, sulla base della classificazione intermedia ottenuta fino a quel momento.

Trattandosi di un sistema che rientra nel contesto di *safety*, abbiamo scelto, come classi finali di fallimento del sistema, le classi definite dalla metodologia FMEA, descritta in Sezione 2.3.1 e in Sezione 2.3.2. Le classi di fallimento di FMEA sono: ***No-error***, nel caso in cui l'errore sia mascherato dal livello hardware o da quello software; ***Safe***, nel caso in cui venga osservato un errore dal *framework*, ma i meccanismi di *safety* introdotti nel codice riescano ad evitare la terminazione anomala o errata dell'esperimento; ***Dangerous***, nel caso in cui i meccanismi di *safety* siano compromessi dall'errore propagato nell'applicazione.

Per raggiungere uno stato finale *Dangerous*, quindi, è necessario che nessuno dei meccanismi di *safety* riesca nel compito di portare la simulazione in uno stato nel quale è in grado di terminare l'azione di frenata. Operando in un contesto *time-critical*, quando le tecniche di irrobustimento del codice hanno fallito, l'ultima

risorsa, per garantire un'esecuzione corretta, è rappresentata dal meccanismo di *Rollback & Replay*. Se il *timer* associato rileva un tempo di esecuzione che viola il vincolo temporale imposto, la simulazione sarà classificata come *Dangerous*. In tutti gli altri casi, R&R garantisce che un secondo tentativo di eseguire la frenata porti il sistema in uno stato *Safe*. Ancora una volta, un meccanismo di R&R riteniamo sia una scelta coerente con i guasti di tipo transitorio che stiamo studiando.

Alla luce di queste considerazioni sulle condizioni da monitorare e sulle classi finali che si possono ottenere dalle simulazioni di guasto, è necessario analizzare l'effettivo comportamento del sistema a fronte di una campagna di iniezione dei guasti sufficientemente estesa.

4.2.3 Classificazione finale dei fallimenti

In questa sezione viene mostrata l'evoluzione del sistema attraverso un diagramma a stati che comprende tutti gli stati intermedi nei quali l'applicazione può trovarsi. Saranno poi presentate tutte le classi di fallimento associate agli stati intermedi e finali. Inizialmente sarà presentato un diagramma contenente l'evoluzione finale del sistema in termini di stati intermedi che specificano le proprietà di individuazione/osservazione/*recovery* del sistema/*framework*. Verrano, poi, descritte nel dettaglio le tipologie di fallimenti associate a ciascun guasto e, alla fine della sezione, sarà mostrato il diagramma iniziale nel quale sono evidenziate le classi di fallimento aggiuntive che è stato possibile ottenere con l'analisi dal punto di vista dell'applicazione effettuata con E-SWEAM. I dettagli sui cambiamenti di stato visualizzati nei diagrammi non sono mostrati nelle immagini, poiché avrebbero reso la visualizzazione confusa, ma sono discussi in modo dettagliato nel testo.

Per ottenere il diagramma a stati finale dell'evoluzione del sistema presentato in Figura 4.12, è stato necessario studiare il comportamento del sistema durante l'esecuzione, effettuata in modo "assistito", di tre diverse campagne di iniezione dei guasti. Durante la prima campagna, infatti, sono emersi gli aspetti legati al

sistema considerato: abbiamo classificato con successo le eccezioni invocate da *interrupt* hardware, i *timeout* (che vengono segnalati da un *watchdog* come quello presente nell'architettura di EDP) e gli errori osservati dal *framework* associati agli eventi di chiamata delle funzioni di *safety*. Durante la seconda campagna, attraverso l'introduzione di meccanismi come il contatore e i *flag*, è stato possibile modellare con maggiore accuratezza il comportamento dell'applicazione, sottolineando che tipo di errore sui dati o sul controllo è stato rilevato. La terza campagna non ha fornito ulteriori risultati interessanti, ma ha anzi confermato che la classificazione definita nei due passi precedenti è sufficiente alla modellazione completa del comportamento del sistema.

Nel diagramma sono mostrati: lo stato iniziale corretto nell'istante prima dell'iniezione di un guasto, l'evoluzione del guasto in un errore di tipo latente, sul flusso di controllo o sui dati, l'effetto dell'errore sui meccanismi di individuazione del sistema e del *framework* e, infine, gli stati finali associati. In particolare, è interessante porre l'attenzione su alcuni degli stati intermedi evidenziati, e sulle loro relazioni.

Nel caso di un errore sul controllo, gli effetti di tale errore sul sistema possono essere di tre tipi: può essere correttamente individuato dai meccanismi di *safety* del sistema, che quindi invocano l'azione di *recovery* associata, può essere osservato attraverso i *monitor* del *framework*, oppure può non essere individuato, e a quel punto lo stato del sistema risulta corrotto senza che i meccanismi adottati siano in grado di rilevarlo. Quest'ultimo caso si traduce in un fallimento di tipo *silent* che causa un risultato errato in uscita, senza possibilità di intervento. Nel caso un errore sia solo individuato dal sistema (ad esempio nel caso di *Timeout*) o osservato dal *framework* (ad esempio, nel caso di *Application Abort*), possono succedere due cose: viene chiamata la funzione di *recovery* di R&R, oppure vengono attivate le funzionalità di *safety* relative all'individuazione di un errore sui dati, poiché la modifica del flusso di controllo relativa ha corrotto i dati elaborati

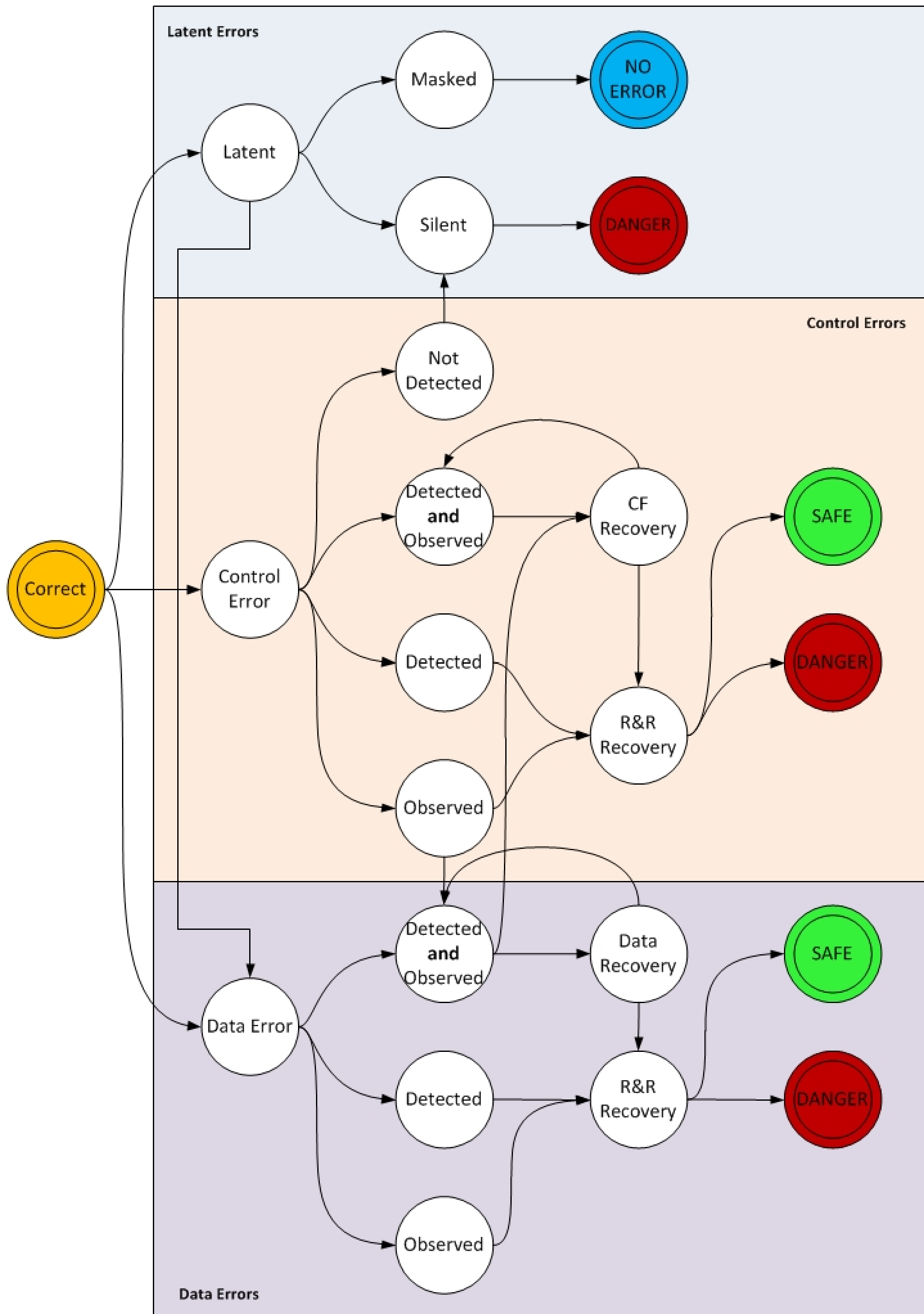


Figura 4.12: *Diagramma a stati finale dell'evoluzione del sistema ABS.*

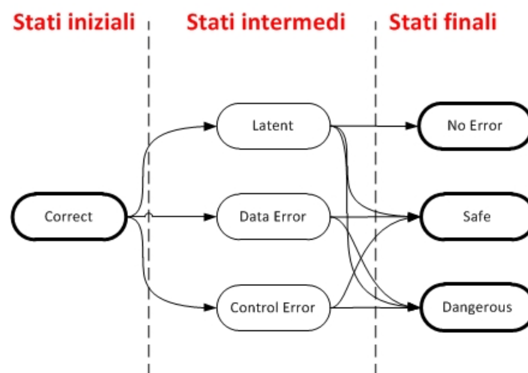


Figura 4.13: *Modello del classificatore per il caso di studio dell'ABS - prima fase.*

causando la chiamata a `signal_data_error`. Quest'ultima eventualità è associata al guasto del *Program Counter* dell'architettura. Per quanto riguarda gli errori sui dati, è possibile che essi siano correttamente individuati e gestiti dal sistema, oppure che siano solo osservati dal *framework*, causando il solo intervento finale del meccanismo di R&R. Nel caso di individuazione, sono possibili due cambiamenti di stato: l'errore sui dati costringe il sistema all'invocazione della funzione di *safety*, oppure, nel caso la variabile che contiene l'errore sia la variabile di controllo utilizzata dal meccanismo di *recovery* per gli errori sul flusso di controllo, viene effettuata una chiamata alla funzione di *recovery* per errori sul flusso di controllo, nonostante l'errore si sia, in origine, manifestato sui dati.

Nei prossimi diagrammi, sarà mostrata la classificazione approfondita ottenuta con E-SWEAM dallo studio accurato del comportamento del sistema in presenza di guasto. Il classificatore di base, impostato prima dell'esecuzione della prima campagna, è mostrato in Figura 4.13. Sono presenti le sole classi *latent*, *data error* e *control error*, con i riferimenti agli stati finali nei quali il sistema può finire. Le campagne di iniezione dei guasti svolte hanno permesso la rifinitura, così come previsto da E-SWEAM, delle classi di fallimento intermedie *data error* e *control error*. In generale, per errori sui dati consideriamo quegli errori che si propagano nell'applicazione e modificano il valore delle variabili passate in ingresso ad una funzione senza alterare il flusso di controllo.

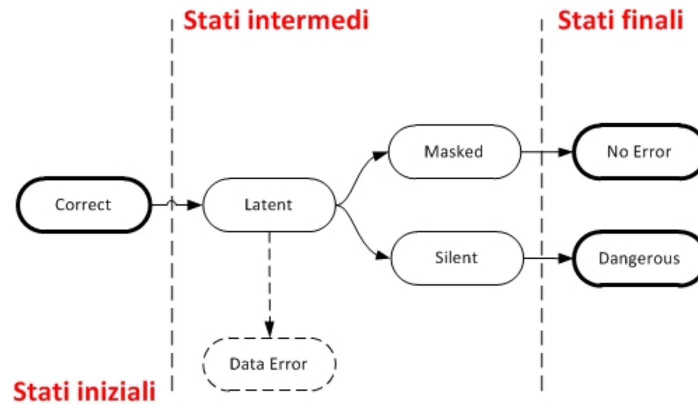


Figura 4.14: *ABS - Diagramma a stati frutto della rifinitura della classe intermedia di errori latenti.*

Nel diagramma di Figura 4.14 è possibile notare la suddivisione della classe di errori latenti in due macro-categorie: la classe *masked* e la classe *silent*. Gli errori che non hanno ripercussioni sull'esecuzione dell'applicazione, che non causano l'attivazione dei meccanismi di individuazione dell'errore del sistema (ad esempio, il gestore delle eccezioni) e che non vengono osservati dal *framework* di E-SWEAM si dice che sono mascherati, dall'applicazione o dall'architettura. Gli errori mascherati portano il sistema in uno stato di *No-error* (fra l'altro, sono gli unici a farlo in questo caso di studio), che significa che l'esecuzione dell'applicazione è proseguita senza deviazioni dal comportamento in assenza di guasto. Nell'applicazione di E-SWEAM, tramite l'analisi dell'architettura e l'estrazione dei dati di *Liveness* dei registri, la maggior parte dei guasti mascherati sono già stati scartati dalla lista di quelli che devono essere iniettati, perché sappiamo già che non causeranno un errore osservabile e non aggiungono nulla alla caratterizzazione del comportamento del sistema che vogliamo ottenere. Nonostante ciò, l'iniezione di alcuni guasti in determinati istanti e in determinate locazioni si traduce in un errore mascherato, che porta allo stato di *No-error*.

La classe di fallimento *silent* rappresenta una corruzione dello stato del sistema che non viene rilevata dal sistema stesso e non viene osservata dal *framework*. Si tratta della classe di fallimento più pericolosa, poiché non c'è modo di rile-

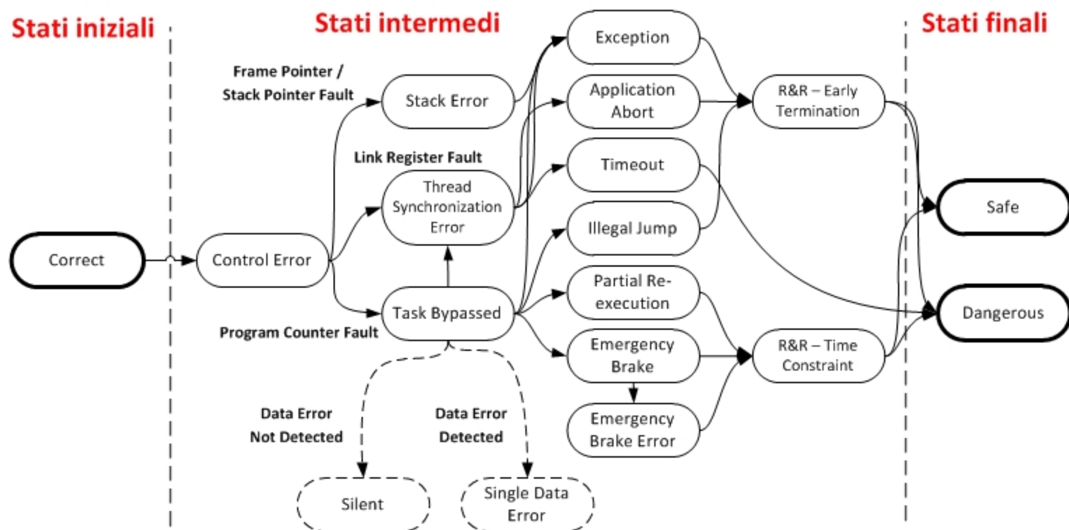


Figura 4.15: *ABS - Diagramma a stati frutto della rifinitura della classe intermedia di errori sul controllo.*

varla e l'esecuzione è destinata a proseguire fino alla fine, producendo risultati sbagliati. Nel caso di studio dell'ABS, dove abbiamo a disposizione un'architettura multi-processore e un'applicazione adeguatamente irrobustita, la casistica di questi errori è molto ridotta.

È importante sottolineare che si può ricadere in queste due classi solo al termine delle simulazioni, poiché, per classificare uno stato *masked*, è necessario che le uscite siano corrette, mentre è necessario che non lo siano per classificare uno stato *silent*.

Gli errori sul controllo (Figura 4.15) rappresentano una classe intermedia di fallimenti che coinvolge tutti i guasti che modificano il flusso di esecuzione dell'applicazione. Gli errori di controllo, per definizione, possono avere effetto anche sui dati, in particolare sulle variabili passate in ingresso alle funzioni che vengono modificate dalle funzioni stesse.

Nel diagramma sono rappresentati i possibili stati nei quali il sistema può trovarsi in ogni istante successivo all'iniezione di un guasto che ha lo scopo di causare un errore sul controllo. Tra i registri associati al flusso di controllo dell'esecuzione, abbiamo il registro #11 o *Link Register*, adibito a contenere l'indirizzo di ritorno

alla funzione chiamante, il registro #12 che contiene il *Frame Pointer*, il registro #13 che contiene lo *Stack Pointer*, il registro #14 e il registro #15 che contengono entrambi una copia del *Program Counter*. Un guasto all'interno di uno di questi registri, quindi, porterà, se non mascherato, alla manifestazione di un errore sul controllo.

Gli stati intermedi del sistema, nel momento in cui si osserva un errore sul controllo, possono essere di tre tipi: il *task* in esecuzione viene saltato, avviene un errore di sincronizzazione tra i *thread*, oppure, in generale, lo stato interno del sistema è corrotto. *Corrupted Internal State* è uno stato "ponte" che serve a identificare un errore di controllo nel sistema, correttamente osservato, causato dal guasto dei registri dedicati al *Frame Pointer* e allo *Stack Pointer*, che causano il sollevamento di eccezioni di tipo hardware oppure di tipo software. Le eccezioni di tipo hardware sono direttamente sollevate dall'architettura nel momento in cui si cerca effettuare un'operazione illegale, come ad esempio scrivere o leggere da un'indirizzo di memoria che non esiste. Le eccezioni di tipo software, invece, sono eccezioni sollevate dall'applicazione e, solitamente, sono invocate nel momento in cui il software rileva uno stato non corretto all'interno dell'architettura. Le eccezioni, di qualsiasi natura esse siano, portano il sistema in uno stato finale *Safe* se sollevate entro un limite di tempo utile rispetto al *timer* attivato dal *framework*. Al contrario, se vengono sollevate in ritardo, l'esecuzione dell'applicazione può risultare compromessa e portare il sistema in uno stato finale *Dangerous*.

In generale, gli errori di sincronizzazione tra i *thread*, in inglese *Thread Synchronization Errors* (TSE), sono causati da un guasto al *Link Register*. Per aumentare il grado di dettaglio relativo a questi fallimenti, nella seconda campagna di iniezione dei guasti abbiamo posto condizioni più stringenti che ci hanno permesso di modellare gli errori TSE come mostrato in Figura 4.16.

Nell'immagine è possibile distinguere tra gli errori che avvengono nel *thread* principale che ha effettuato la *fork* rispetto agli errori che avvengono nel *thread*

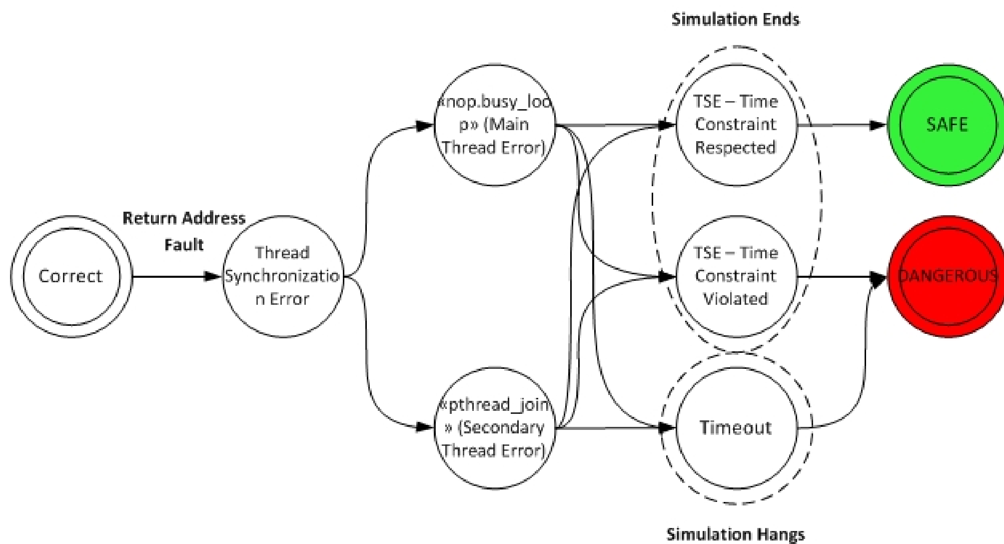


Figura 4.16: *Rifinitura della classe di fallimento intermedia Thread Synchronization Error.*

secondario. Nel primo caso, l'esecuzione del *thread* secondario rimane bloccata sulla funzione `._nop_busy_loop`, mentre, nel secondo caso, l'esecuzione del *thread* principale rimane correttamente in attesa della terminazione del secondo *thread* sulla funzione `pthread_join`. Gli errori TSE possono, nel caso pessimo, bloccare l'intera esecuzione dell'applicazione, causando un fallimento di tipo *Timeout*. *Timeout*, come viene anche mostrato nel diagramma a stati iniziale, è una condizione *Dangerous* perché non consente il corretto svolgimento dei *task* e non viene individuata dai meccanismi di gestione degli errori, portando ad un risultato finale sicuramente erroneo e con conseguenze rischiose. Ovviamente, un errore di TSE può portare anche al sollevamento di un'Eccezione, causata da chiamata ad un *interrupt* hardware o software. È importante sottolineare, quindi, come lo stato TSE sia uno stato intermedio, mentre gli stati *Timeout* e Eccezione sono stati finali, associati ad una classe di fallimento predefinita (*Safe/Dangerous*).

Infine, l'ultima tipologia di errori sul controllo rilevati in questo caso di studio, riguarda i task non correttamente eseguiti (*Task Bypassed* - TB) poiché saltati o interrotti a causa di un guasto nel *Program Counter* del processore. Lo stato TB può avere diverse conseguenze sul flusso di controllo dell'applicazione:

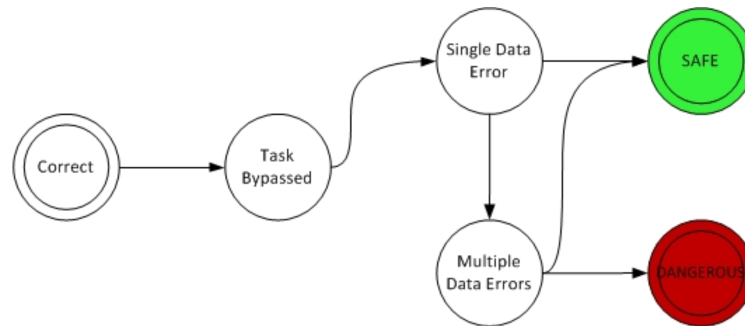


Figura 4.17: *ABS - Diagramma a stati che lega gli errori di controllo con gli errori sui dati.*

- può causare il sollevamento di un'eccezione, a causa, ad esempio, della richiesta di un'istruzione inesistente;
- può causare un *Timeout* nel caso in cui la non-esecuzione di un task causi un errore di sincronizzazione tra i *thread*;
- può causare una parziale ri-esecuzione di un *task*, che in generale non ha ripercussioni gravi sul flusso di controllo; può causare la chiamata alla funzione `emergency_brake`, che gestisce un errore sul controllo eseguendo un ciclo di frenata dall'iniziale lettura dei sensori;
- può causare un errore sui dati calcolati che si propaga fino a portare alla terminazione dell'esecuzione a causa dei troppi errori sui dati (come mostrato in Figura 4.17);
- infine, questo tipo di errore può non essere individuato/osservato, causando la transizione del sistema nello stato *Silent*, che porterà ad un risultato finale errato.

I casi di sollevamento di eccezione e *Timeout* sono già stati ampiamente discussi. Il caso di parziale ri-esecuzione di un *task* può avere conseguenze dannose, sull'esecuzione dell'applicazione, solo nel caso in cui la ri-esecuzione impieghi un tempo superiore al tempo limite impostato dal *timer*. In tutti gli altri casi, invece, la ri-esecuzione non causa fallimenti critici. Per quanto riguarda la chiamata

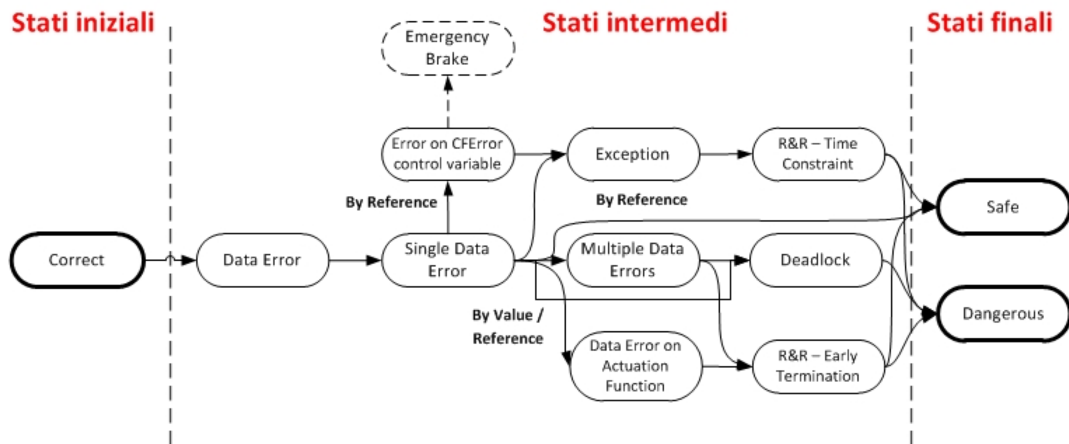


Figura 4.18: *ABS - Diagramma a stati frutto della rifinitura della classe intermedia di errori sui dati.*

alla funzione di emergenza, possiamo distinguere tre casi: il vincolo temporale del *timer* viene rispettato, oppure viene violato, oppure, infine, l'errore viene propagato all'interno della funzione di emergenza. Quest'ultimo caso è particolarmente difficile da gestire, poiché la funzione di emergenza non è dotata, al proprio interno, di meccanismi di gestione degli errori. Un errore che corrompe il flusso di controllo all'interno di essa, quindi, viene classificato come *Dangerous* perché non reversibile.

Il diagramma che mostra l'evoluzione del sistema a fronte di errori sui dati è mostrato in Figura 4.18. Gli errori sui dati coinvolgono la modifica scorretta dei valori delle variabili passate in ingresso ad una funzione, errori che possono facilmente propagarsi all'interno dell'applicazione causando errori negli stati intermedi e, soprattutto, sui risultati finali. Vista la criticità del sistema *embedded* ABS, è necessario che gli errori sui dati siano correttamente individuati e gestiti dall'applicazione.

In base al registro nel quale viene iniettato il guasto, è possibile distinguere tra errori sulle variabili passate per valore e su quelle passate per riferimento. Questa distinzione può risultare interessante per diversi motivi; ad esempio, un guasto in un registro che contiene l'indirizzo di una variabile passata per riferimento avrà

probabilità maggiori di causare il sollevamento di un'eccezione hardware causata dalla richiesta di un indirizzo non accessibile in memoria, perché inesistente oppure protetto. In generale, poi, oltre a questa prima distinzione, è possibile distinguere il tipo di errore sui dati in base alla funzionalità della variabile che viene corrotta, o anche in base al task all'interno della quale viene corrotta.

Un errore iniettato nei registri che contengono un parametro in ingresso ad una funzione, se non mascherato, causa un singolo errore sui dati. Da questo stato, il sistema può evolvere in cinque modi diversi: può essere sollevata un'eccezione hardware per le cause che abbiamo citato prima; l'errore rimane confinato alla funzione nella quale viene osservato, viene corretto dalla funzione che gestisce la consistenza dei dati, e al termine della simulazione è possibile classificare lo stato come *Safe* (è importante sottolineare che un singolo errore sui dati non causerà mai la violazione del vincolo temporale imposto dal *timer*, poiché il tempo necessario alla gestione del singolo errore è molto contenuto); l'errore si propaga causando errori multipli sui dati nelle funzioni successive; l'errore coinvolge la variabile adibita all'identificazione di errori sul controllo; infine, l'errore coinvolge la funzione di attuazione dell'applicazione.

Il singolo errore o errori multipli causati dal guasto di un'indirizzo di una variabile passata per riferimento possono causare il *deadlock* dell'applicazione, che continua la propria esecuzione con dati sbagliati finendo per non frenare il veicolo. Questa situazione è critica e non osservabile prima del termine della simulazione. Errori multipli sui dati, inoltre, se superano una soglia prestabilita, portano al sollevamento di un *flag* che interrompe la simulazione: in questo caso, se è avanzato tempo necessario per un *reset* e una ri-esecuzione, l'esperimento è considerato *Safe*; altrimenti, è considerato *Dangerous*. Un errore sui dati nella funzione di attuazione è particolarmente rischioso perché questa funzionalità non è irrobustita a livello di codice come le altre. Un errore nei parametri di ingresso di questa funzione può portare alla scrittura, sull'attuatore della pompa dei freni,

di un valore che può compromettere l'intera esecuzione. Per questo motivo, un errore sui dati della funzione di attuazione causa l'interruzione dell'esperimento e il *reset* dell'esecuzione.

Infine, se l'errore coinvolge la variabile che si occupa di gestire gli errori sul flusso di controllo (che, ricordiamo, è passata in ingresso ad ogni funzione), può essere sollevata un'eccezione oppure può essere invocata la funzionalità di frenata di emergenza che dovrebbe essere attivata solo quando effettivamente si verificano errori sul flusso di controllo. La chiamata di tale funzione ha le conseguenze descritte durante la trattazione degli errori sul controllo. Com'è possibile notare, quindi, errori sul controllo possono ripercuotersi sui dati dell'applicazione ma, nel caso specifico di questo caso di studio, un errore sui dati può modificare il flusso di controllo a causa delle funzionalità di gestione degli errori presenti nell'applicazione considerata.

In Figura 4.19 è mostrato il diagramma presentato all'inizio della sezione, decorato con le classi di fallimento estratte da questo caso di studio. Come premesso, questo diagramma risulta confuso, ma mostra in modo dettagliato tutti gli stati intermedi e le loro interazioni, per ogni tipo di errore.

4.2.4 Analisi dei risultati della campagna di iniezione di guasti

Per il caso di studio dell'ABS, sono stati iniettati, nei registri del processore, 25000 guasti, mentre nella memoria ne sono stati iniettati 10000. Il tempo complessivo necessario agli esperimenti è stato di circa 32 ore. Le locazioni nelle quali iniettare sono state scelte attraverso lo studio della *liveness* ottenuto durante la fase di caratterizzazione del sistema. In particolare, abbiamo estratto, sia per quanto riguarda i registri del processore che le celle di memoria, dei *log* contenenti gli intervalli temporali di *liveness* di questi componenti, in modo da concentrare l'iniezione in istanti nei quali la probabilità di osservare un errore sull'applicazione è maggiore di zero. Soprattutto per quanto riguarda i registri, il *log* contiene

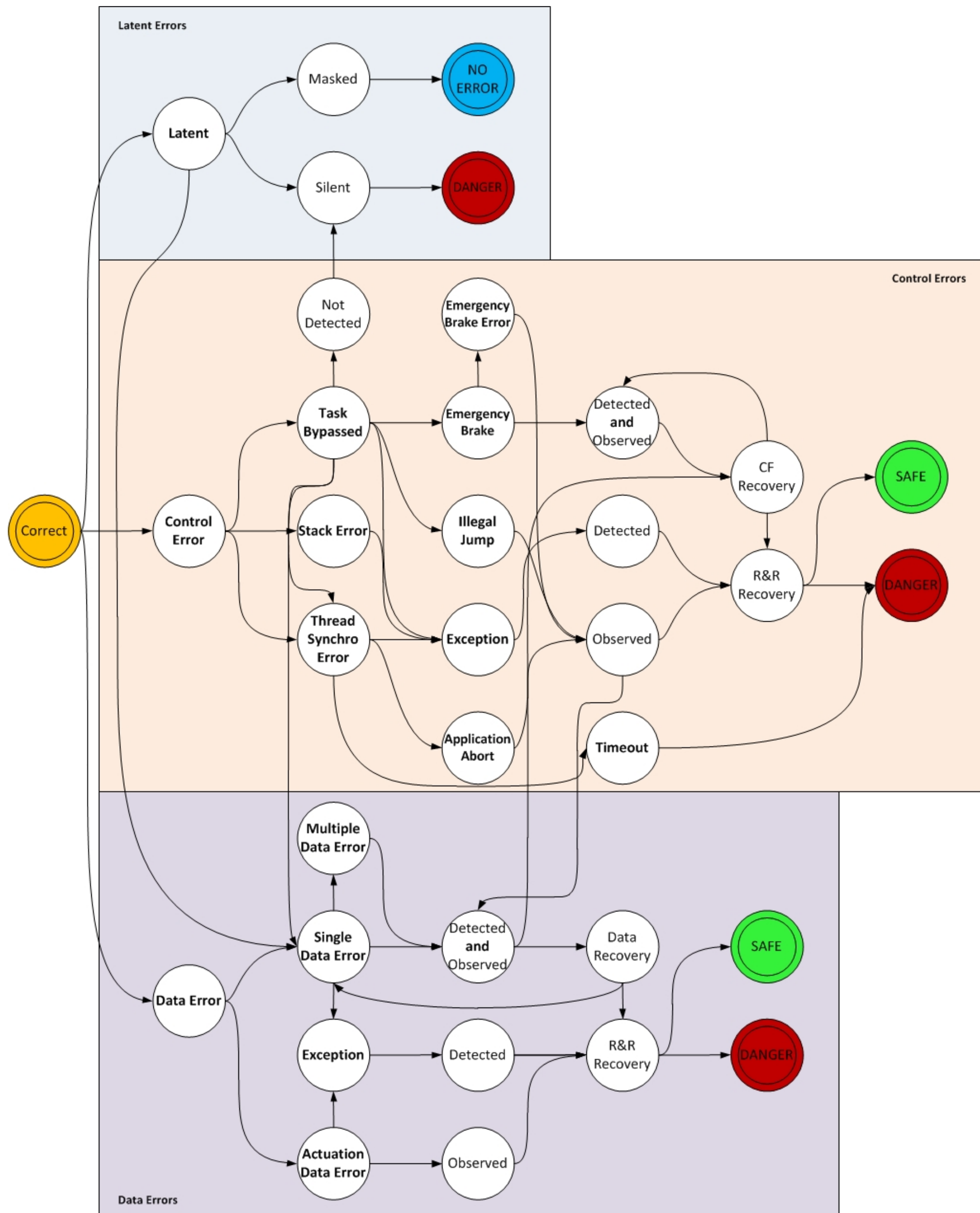


Figura 4.19: *Diagramma completo degli stati intermedi e delle classi di fallimento del caso di studio ABS.*

molti intervalli temporali. Abbiamo scelto, per ogni locazione, quattro diversi istanti temporali: dopo il primo ciclo dell'applicazione, dopo 20 cicli, a metà dell'esecuzione e a 10 cicli dalla fine, in modo da ottenere valori significativi e diversi tra loro. Per calcolare la *liveness* dei registri del processore, sono state analizzate nel dettaglio le tracce del processore che riguardano la lettura e la scrittura sui registri al proprio interno. Queste tracce sono state rielaborate e da esse siamo stati in grado di calcolare gli intervalli di *liveness*, cioè gli intervalli tra un'operazione di scrittura e l'ultima operazione di lettura successiva, all'interno dei quali un registro è considerato "vivo".

L'analisi dell'architettura e la caratterizzazione del sistema hanno evidenziato che i registri utilizzati dai processori, durante l'esecuzione dell'applicazione, sono i registri da #0 a #3 e i registri da #11 a #15. Mentre i primi, come sappiamo, sono registri liberi che contengono i primi quattro parametri di una funzione chiamata, i registri 11-15 sono registri dedicati, che contengono valori che dipendono dal tipo di architettura scelta. Per l'ARM9, questi registri contengono, nell'ordine: indirizzo di ritorno al chiamante all'uscita di una funzione, *Link Register*, *Frame Pointer*, *Stack Pointer* e *Program Counter*. In questo modo, per quanto riguarda la definizione di *monitoring* e classificazione, siamo in grado di sapere quali funzionalità dell'architettura stiamo corrompendo.

Com'è possibile notare, le classi di fallimento ottenute definiscono tutti i comportamenti assunti dall'applicazione a fronte degli errori all'interno di essa. Vogliamo sottolineare come l'applicazione di E-SWEAM abbia permesso una modellazione dell'evoluzione molto accurata, proprio dal punto di vista applicativo. Le scelte attuate per le strategie di *monitoring* e classificazione si rivelano corrette, consentendo, nel caso di rilevamento di un errore, la corretta individuazione. L'unico caso, trattandosi di un esperimento su 25000, quindi con un'incidenza dello 0,004%, in cui il *framework* non è stato in grado di fornire informazioni sulla propagazione dell'errore (il sistema si è quindi trovato in uno stato *Silent*

Sigla	**Significato**
MT	Main Thread
ST	Secondary Thread
AA	Application Abort
ADE	Actuation Data Error
DL	Deadlock
EB	Emergency Brake
EBE	Emergency Brake Error
HWEx	Hardware Exception
IJ	Illegal Jump
MDE	Multiple Data Errors
PRE	Partial Re-Execution
SDC	Silent Data Corruption
SDE	Single Data Error
SWEx	Software Exception
TB	Task Bypassed
TO	Timeout
TSE	Thread Synchronization Error

Tabella 4.3: *Legenda per le sigle dei fallimenti contenuti in Tabella 4.4.*

dopo la manifestazione di un errore sul controllo), è stato durante un esperimento di iniezione nel registro *Program Counter*, che ha fatto registrare un risultato in uscita errato senza che fosse possibile osservare stati intermedi corrotti. Questo si traduce in un'ottima capacità di analisi della metodologia, poiché siamo stati in grado di classificare praticamente tutte le classi di fallimento del sistema con le condizioni specificate nella fase precedente.

In Tabella 4.3 sono elencati i significati delle sigle presenti in Tabella 4.4, la quale mostra i fallimenti riscontrati nel caso di studio ABS, suddivisi per funzionalità guastata e *task* corrotto. Le sigle fanno riferimento ai fallimenti presentati precedentemente con i diagrammi, e legano tutti i fallimenti descritti al tipo di guasto che avviene all'interno dell'architettura e rispetto al *task* eseguito dal software. Queste tabelle sono uno strumento efficace nel mostrare il legame

Tabella 4.4: *Tipologie di fallimenti suddivisi per funzionalità e task corrotti.*

Proprietà Architettura		*Task* Applicazione				
Registro	**Funzionalità**	Acquisizione-MT	Acquisizione-ST	Elaborazione-MT	Elaborazione-ST	Attuazione
RB0..3	Parametri passati per valore	-	-	SDE, HWEx	-	ADE, HWEx
RB0..3	Parametri passati per riferimento	EB, SDE, HWEx, SWEx	EB, MDE, HWEx, DL, ADE	EB, HWEx, SWEx	EB, MDE, HWEx, DL	ADE, EB, HWEx, SWEx
RB11	*Link Register*	AA, TO, HWEx, SWEx	AA, TO, HWEx, SWEx	AA, TO, HWEx, SWEx	AA, TO, HWEx, SWEx	AA, TO, HWEx, SWEx
RB12	*Frame Pointer*	HWEx, SWEx	HWEx, SWEx	HWEx, SWEx	HWEx, SWEx	HWEx, SWEx
RB13	*Stack Pointer*	HWEx, SWEx	HWEx, SWEx	HWEx, SWEx	HWEx, SWEx	HWEx, SWEx
RB14-15	*Program Counter*	TO, EB, PRE, HWEx, SWEx	EB, IJ, SDE, MDE, HWEx, SWEx	TO, EB, PRE, HWEx, SWEx	EB, EBE, SDE, MDE, HWEx, SWEx	TO, EB, PRE, SDC, HWEx, SWEx

tra locazione del guasto, funzionalità corrotta ed errore osservato. Inoltre, sono utili alla comprensione delle tabelle mostrate in Appendice B contenenti i risultati dell'intera campagna effettuata per il sistema ABS. In questa sezione, infatti, saranno discussi solamente i valori più interessanti ottenuti dall'esecuzione degli esperimenti di iniezione.

HWEx e SWEX rappresentano eccezioni causate da chiamate a *interrupt* rispettivamente da parte dell'hardware e da parte del software. *Application Abort* (AA), *Illegal Jump* (IJ) ed *Emergency Brake Error* (EBE) fanno riferimento allo stesso tipo di fallimento, cioè la terminazione inattesa dell'applicazione senza segnalazione di errore. La classe è stata suddivisa in tre diversi fallimenti poiché le cause o le conseguenze possono differire. In particolare, AA è il caso generico di terminazione anomala che coinvolge un errore di sincronizzazione tra *thread*, e quindi la funzionalità di *Link Register* guasta. IJ si riferisce ad un salto inatteso causato dal guasto del *Program Counter* non correttamente gestito dall'applicazione, che termina. EBE, infine, rappresenta un caso di IJ nel quale l'applicazione "salta" all'interno della funzione di *recovery* implementata per gestire gli errori sul flusso di controllo. Dagli esperimenti è risultato che, in diversi casi, un simile salto ha provocato la terminazione anomala all'interno della funzione di *safety* che, in realtà, dall'analisi a livello applicazione effettuata, non avrebbe dovuto essere chiamata. *Partial Re-execution* è un caso di salto "all'indietro" dell'esecuzione che non causa errori, se non un maggiore tempo di esecuzione finale, che però, negli esperimenti eseguiti, non ha causato la violazione del vincolo temporale imposto dal *timer* del meccanismo di R&R. *Deadlock*, infine, rappresenta la situazione anomala di errore individuato ed esecuzione compromessa a causa della mancata frenata del veicolo poiché i parametri letti dai sensori sono errati.

Dalla tabella è possibile evidenziare il legame tra funzionalità guastata e fallimento osservato, sottolineando quelle che sono le potenzialità di E-SWEAM nel dare una definizione molto dettagliata del legame guasto/errore. In particolare,

viene mostrato in quale modo il sistema fallisce relativamente alle locazioni nelle quali, durante la campagna, sono stati iniettati i guasti. I fallimenti qui descritti sono relativi allo stato interno del sistema e, in particolare, sono ricavabili, a parte le eccezioni causate da *interrupt* hardware, unicamente dall'applicazione analizzata. L'utilizzo di condizioni specifiche e, in alcuni casi, personalizzate rispetto al tipo di contesto di riferimento ci ha consentito di valutare accuratamente a livello software ciò che è in realtà corrotto a livello hardware.

Scendendo nel dettaglio, dai risultati della campagna è possibile ottenere le seguenti informazioni aggiuntive:

- la tecnica di irrobustimento del codice per quanto riguarda gli errori sui dati si è rivelata efficace. Infatti, osservando i risultati delle iniezioni nei registri contenenti i parametri passati in ingresso, è possibile notare come gli errori sui dati propagati siano correttamente gestiti dal sistema, che li individua, chiama la funzione di *safety* relativa e solo circa nel 7-8% dei casi si ritrova in uno stato *Dangerous*, causato dalla propagazione di errori multipli sui dati che non garantiscono tempo sufficiente al meccanismo di R&R per il *reset*.

- La differenza, in termini di numero di eccezioni sollevate, tra l'iniezione di un guasto all'interno di registri contenenti valori di variabili rispetto all'iniezione all'interno di registri contenenti l'indirizzo di una variabile è interessante. In quest'ultimo caso, infatti, ci aspettiamo che, vista la natura del parametro corrotto, le eccezioni (come, ad esempio, la `IllegalAddress`) siano più frequenti rispetto al primo caso. Il dato mostrato in Tabella 4.5 avvalora la nostra tesi, mostrando la differenza, in termini di eccezioni sollevate, dalla corruzione di un registro contenente un indirizzo e un registro contenente un valore, in due diversi *task*.

- La tecnica di irrobustimento del codice per quanto riguarda gli errori sul controllo, invece, risulta meno efficace. Le percentuali relative ai registri

RB0	Totale Exception Acquisizione - MT	17,2%
	Totale Exception Elaborazione - MT	1,4%

Tabella 4.5: *Differenze percentuali di lancio di eccezioni tra variabili passate per riferimento e variabili passate per valore.*

#14 e #15, contenenti il *Program Counter*, lo dimostrano: in media, circa il 2% degli errori di controllo in questi registri sono correttamente gestiti dalla funzione di *safety* che chiama la funzione `emergency_brake`. Paradossalmente, il meccanismo si rivela molto più efficace quando, a causa di un guasto, viene corrotto il valore della variabile passata in ingresso che gestisce il meccanismo di individuazione e gestione dell'errore sul flusso di controllo: le percentuali, in questo caso, superano il 50%. Questa conseguenza è, in realtà, un effetto collaterale dell'irrobustimento che abbiamo aggiunto.

- Il meccanismo di R&R si comporta bene quando si tratta di gestire fallimenti causati da errori sui dati. Le percentuali *Dangerous* relative ai guasti delle funzionalità di *Link Register* e *Program Counter*, invece, mostrano come questo meccanismo necessiti di un miglioramento per consentire che gli errori sul controllo non portino il sistema in uno stato nel quale le funzioni di *safety* introdotte non siano in grado di svolgere il proprio compito.

- I meccanismi di osservazione del *framework*, in generale, si sono rivelati utili nel svolgere il proprio compito. Gli errori sui dati, se individuati dal sistema, sono sempre rilevati dal *framework*. Inoltre, per quanto riguarda gli errori sul controllo, i meccanismi di *flag, timer* e il *log* delle chiamate a funzione permettono la modellazione di comportamenti non gestiti dal sistema stesso. Si vedano, per l'avallo di questa ipotesi, i dati relativi ai fallimenti di tipo *Application Abort*, *Emergency Brake Error* e *Illegal Jump*. Questi fallimenti non sarebbero stati identificabili senza l'analisi accurata svolta con l'ausilio della metodologia E-SWEAM;

- Le eccezioni simboleggiano la robustezza intrinseca del sistema prima dell'introduzione dei meccanismi di *safety* discussi. La percentuale di circa il 50% di eccezioni sollevate a causa dei guasti mostra come il sistema, in circa la metà dei casi, sia in grado di diagnosticare da solo la presenza di un errore. L'introduzione del meccanismo di R&R, in questo senso, permette di classificare la maggior parte dei casi di lancio di eccezione come stati finali *Safe*.

- I guasti iniettati nella memoria hanno fatto registrare, come previsto, un numero superiore alla media di *No-error*, poiché è più difficile iniettare in una locazione di memoria attiva che può causare la propagazione di un errore nell'applicazione. I guasti nella memoria hanno fatto comunque registrare una discreta percentuale di errori sia sui dati che sul flusso di controllo. In particolare, sono emersi stati *Single Data Error*, *Multiple Data Error* e *Actuation Data Error* nel primo caso, che hanno portato a terminazioni della simulazione nello stato *Safe*. Nel secondo caso, invece, si sono registrati dei *Timeout*, delle terminazioni anomale a causa di errori di sincronizzazione dei *thread* e chiamate alla funzione di *safety* `emergency_brake`. In Tabella 4.6 sono mostrate le percentuali di fallimenti relative ai guasti iniettati nella memoria. Va sottolineato che, dei 10000 guasti iniettati, solo l'11% circa ha fatto registrare un risultato *Dangerous* della simulazione. In particolare, quasi un terzo di questi sono causati da eccezioni, segnalate, in ritardo, dal sistema, situazione che provoca il fallimento del meccanismo di R&R. Percentuale significativa (circa del 25%) di errori sui dati, sia *Safe* che *Dangerous*. Questo risultato è prevedibile poiché è più facile corrompere una locazione di memoria contenente un dato elaborato dall'applicazione che una locazione contenente dati utiiizzati per la gestione del flusso di controllo. Vogliamo sottolineare un caso particolare di *Timeout* non dovuto ad un errore di sincronizzazione tra *thread* e nemmeno, a nostro parere, ad

No-error	Masked	**37,71%**
Safe	Hardware Exception	24,45%
	Single Data Error	10,67%
	Multiple Data Error	7,85%
	Actuation Data Error	3,24%
	Emergency Brake	2,92%
	Application Abort	1,51%
	TOTALE	**50,64%**
Dangerous	Hardware Exception	3,02%
	Timeout	2,55%
	Multiple Data Error	2,41%
	Actuation Data Error	2,04%
	Emergency Brake	1,33%
	Application Abort	0,30%
	TOTALE	**11,65%**

Tabella 4.6: *Risultati delle iniezioni nella memoria effettuate in istanti temporali random durante l'intera esecuzione dell'applicazione.*

un guasto del *Program Counter*. Durante una simulazione, l'applicazione ha eseguito correttamente ma impiegando molto più tempo (circa il 35% in più) del dovuto, causando la segnalazione di un *Timeout* ma mostrando risultati in uscita corretti. Come detto, si tratta comunque di un caso su 10000 iniezioni, con un'incidenza dello 0,01%.

In conclusione, il caso di studio dell'ABS ha dimostrato come l'applicazione di E-SWEAM consenta, oltre alla definizione di nuove classi di fallimento in grado di descrivere in modo accurato tutti i comportamenti del sistema, in relazione anche alle proprietà di *safety*. In questo contesto, ABS necessita di una differente strategia di gestione degli errori sul flusso di controllo dell'applicazione e di un meccanismo più evoluto di R&R, in grado di gestire in modo più efficace le situazioni rischiose.

In conclusione, abbiamo verificato, attraverso i casi di studio trattati, l'efficacia

di E-SWEAM nell'analizzare nel dettaglio il comportamento di sistemi complessi hw/sw soggetti a guasti hardware che causano la propagazione di errori nel software. Inoltre, abbiamo provato che le tecniche di affidabilità adottate soffrono di evidenti limitazioni, soprattutto per quanto riguarda la tolleranza a errori sul flusso di controllo delle applicazioni.

Nel prossimo capitolo, trarremo le conclusioni di questo lavoro di tesi, soffermandoci su quelli che possono essere gli sviluppi futuri di E-SWEAM.

Capitolo 5

Conclusioni e Sviluppi Futuri

In questa tesi è stata presentata una nuova metodologia per l'analisi delle proprietà di affidabilità di sistemi dedicati, chiamata *Enhanced Software Error Analysis Methodology* (E-SWEAM).

E-SWEAM è stata progettata e sviluppata con lo scopo di fornire un approccio per la caratterizzazione, l'analisi e la classificazione dei fallimenti derivanti da guasti all'interno di sistemi hw/sw complessi. Inoltre, E-SWEAM si occupa della valutazione delle proprietà di affidabilità del sistema, fornendo un'analisi delle criticità dei componenti hardware e delle funzioni software più suscettibili all'interno del dispositivo. In particolare, le problematiche di cui si occupa E-SWEAM riguardano sistemi che sono afflitti da guasti di tipo transitorio, i quali, al giorno d'oggi, rappresentano la classe di guasti più frequente nella quale possono incorrere i dispositivi durante la fase operativa. E-SWEAM nasce come una metodologia applicabile durante le prime fasi di sviluppo di un dispositivo *embedded*, poiché si serve di strumenti di simulazione in grado di consentire al progettista di modellare un sistema anche molto complesso, il quale può essere sottoposto ad adeguati test per valutarne la suscettibilità ai *soft errors*.

E-SWEAM è composta da tre fasi distinte, ognuna delle quali consiste in una serie di attività "atomiche". Ogni fase è dipendente dalle informazioni ottenute durante i passi precedenti. La prima fase consiste nella caratterizzazione del sistema in analisi. Le informazioni sull'architettura vengono ottenute dalle specifiche

in *Hardware Description Language* del modello, mentre le informazioni sull'applicazione sono estratte attraverso una fase di analisi statica del programma, che analizza il codice sorgente e il codice oggetto per ottenere le informazioni relative al flusso di esecuzione delle funzioni. In seguito, vengono salvate le tracce di esecuzione del programma durante una simulazione del modello in assenza di guasto.

Le tracce di esecuzione e le informazioni, architetturali e applicative, sono utilizzate per le seconda fase, che prevede la definizione delle strategie di *monitoring* e classificazione dei fallimenti del sistema. Le strategie sono personalizzabili in base ai dati ottenuti nella fase precedente. Una volta ottenuta una prima classificazione dei fallimenti, attraverso l'esecuzione di una o più campagne di guasto è possibile ottenere una rifinitura di tali classi e uno studio accurato dell'evoluzione del sistema a fronte di un guasto.

L'ultima fase prevede l'analisi dei dati ottenuti attraverso gli esperimenti di iniezione per evidenziare quelle che sono le criticità del modello simulato, in termini di componenti hardware oppure funzionalità software.

La metodologia è supportata da una piattaforma di simulazione, chiamata *Reflective Simulation Platform* (ReSP), in grado di simulare modelli di sistemi partendo dalle specifiche *SystemC TLM* fornite. ReSP, inoltre, mette a disposizione un motore per l'iniezione di guasti basato su simulazione. Il motore è basato sull'utilizzo dei comandi di simulazione modificati per l'iniezione di guasti all'interno degli stati interni del sistema, e si serve di moduli specifici, chiamati *saboteurs*, che hanno il compito di iniettare un guasto all'interno delle interconnessioni del circuito. In questo modo, è possibile studiare gli effetti di diversi modelli di guasto.

La metodologia sfrutta questa piattaforma in modo sinergico all'analisi dal punto di vista applicativo per consentire all'utente di studiare, in modo approfondito, l'evoluzione del sistema e la propagazione degli errori. Gli strumenti

permettono di salvare le tracce dell'esecuzione del software, in modo da avere a disposizione informazioni utili per la definizione di condizioni che si vuole monitorare durante gli esperimenti di iniezione di guasti. Quest'analisi approfondita a livello software è supportata dalla generazione di diagrammi a stati che modellano gli stati intermedi nei quali il sistema si trova in ogni istante della simulazione. Questi diagrammi sono uno strumento efficace per comprendere gli effetti di un guasto sulle proprietà di affidabilità del sistema.

Per validare le scelte compiute durante la progettazione di E-SWEAM, abbiamo applicato la metodologia a due casi di studio, molto diversi tra loro. Lo scopo, oltre a mostrare il grado di dettaglio dell'analisi che E-SWEAM è in grado di ottenere, è stato quello di testare le proprietà di affidabilità dei due sistemi in analisi, e mostrare le carenze delle scelte adottate.

Il primo caso di studio consiste nell'analisi di un'applicazione di elaborazione di immagine, chiamata *EdgeDetectorParallel* (EDP), irrobustita a livello software attraverso l'utilizzo di un meccanismo basato su *Triple Modular Redundancy*. EDTMR (questo il nome del sistema) è composto da un'architettura basata su tre processori, ed esegue un'applicazione *multi-thread* che ha il compito di elaborare un'immagine ricevuta in ingresso e restituirne una versione sulla quale è stata applicata la tecnica di rilevamento dei contorni. Ogni processore esegue gli stessi *task*. Al termine della loro esecuzione, il controllo ritorna al *thread* principale, che esegue una funzione, chiamata `softwareVoter`, che ha il compito di analizzare le tre immagini restituite dai *thread* e votare quella corretta, che verrà poi scritta su file. TMR, quindi, è implementato a livello software all'interno di un applicazione fortemente dipendente dalla corretta esecuzione di ognuno dei tre *thread*. Questa limitazione si traduce, durante gli esperimenti di iniezione di guasti, in una percentuale molto elevata di fallimenti critici dovuti a errori sul flusso di controllo dell'applicazione, che comportano la produzione, in un'uscita, di un'immagine corrotta e non utilizzabile. La campagna di iniezione di guasti ha sottolineato que-

sto aspetto del sistema, facendoci propendere per una soluzione alternativa che preveda l'utilizzo di un *voter* hardware che riceve le immagini dopo l'esecuzione separata dei singoli *thread*, così da garantire che, solo nel caso quasi impossibile che almeno due processori si guastino contemporaneamente, l'immagine restituita sia corrotta.

Il secondo caso di studio, invece, ha previsto l'analisi di un sistema che simula il dispositivo di *Anti-lock Braking System* (ABS) presente nei veicoli in commercio. ABS è un sistema di *safety time-critical*: lo scopo dell'applicazione, cioè far frenare in modo sicuro il veicolo sul quale è eseguita, è vincolato da un limite temporale ben preciso, superato il quale il sistema si trova in una condizione di rischio. Per questo motivo, ABS è stato irrobustito a più livelli: è dotato di un multiprocessore per l'esecuzione parallela e ridondante dell'elaborazione dei dati, è dotato di blocchi di codice che hanno il compito di rilevare e, eventualmente, correggere un errore nell'applicazione e, infine, si serve di un meccanismo di *safety*, basato su *timer*. I risultati degli esperimenti effettuati su questo sistema hanno rilevato come le tecniche di irrobustimento del codice e di esecuzione ridondante siano efficaci nella rilevazione e nella correzione degli errori che possono verificarsi all'interno del sistema, ma è necessario affiancare, al meccanismo basato su *timer*, un meccanismo di *safety* ulteriore, come ad esempio un circuito parallelo che effettui gli stessi calcoli, per garantire che, in caso di fallimento di uno dei due dispositivi, l'altro sia in grado di portare il sistema in uno stato sicuro.

Le potenzialità di E-SWEAM fin qui descritte possono essere sfruttate per l'analisi di qualsiasi tipo di sistema dedicato del quale si hanno a disposizione le specifiche *SystemC TLM*. In unione a questo fatto, il tipo di analisi eseguita con questa metodologia è in grado di valutare, in modo concreto e dettagliato, le proprietà di affidabilità dei sistemi in analisi, candidandosi ad essere, in un futuro, inserita nel flusso di progettazione di sistemi *embedded* con proprietà di affidabilità.

Il lavoro lascia aperti diversi spunti interessanti che possono rappresentare una valida motivazione per effettuare studi futuri nei confronti di questa metodologia.

Per prima cosa, E-SWEAM potrebbe essere sfruttata, data la sua flessibilità, anche in fasi più avanzate del flusso di progetto di sistemi dedicati. Ci siamo limitati allo studio di modelli attraverso le loro specifiche di alto livello, mentre sarebbe interessante applicare quanto trattato in questo elaborato anche nelle fasi successive del flusso di sviluppo, in modo da analizzare in modo approfondito il comportamento dei sistemi per i quali si vuole analizzare in modo esteso il comportamento in presenza di un guasto. E-SWEAM, infatti, risulta versatile nel definire classi di fallimento più specifiche e modellare l'evoluzione dello stato interno del sistema, magari per dispositivi già analizzati con altre metodologie, per confrontare le ulteriori informazioni che è possibile estrarre. Inoltre, il *framework* di analisi presentato può essere implementato come un livello al di sopra di diversi ambienti per l'iniezione di guasti. In particolare, l'interprete a livello applicazione è indipendente da ReSP e può essere facilmente adattato ad altri *framework*. Gli unici requisiti richiesti a eventuali altri motori per l'iniezione di guasti sono che l'iniettore garantisca buone proprietà di osservabilità rispetto al sistema in analisi, e un'elevata programmabilità, una proprietà fornita, solitamente, da ambienti il cui controllo è implementato a livello software. Proprio per questa ragione, gli iniettori di guasti che si candidano ad implementare il *framework* di E-SWEAM possono non necessariamente appartenere alla famiglia degli iniettori basati su simulazione ma, ad esempio, possono far parte di iniettori di guasti implementati a livello software (SWIFI), in grado di sfruttare le potenzialità dell'unità di *debugging*.

In secondo luogo, per quanto riguarda questo lavoro di tesi, durante lo sviluppo e i test effettuati nei casi di studio proposti, non ci siamo soffermati ad osservare le differenze che possono essere estratte da un'analisi effettuata attraverso l'iniezione di guasti sulle interconnessioni invece che negli elementi di memoria interni (regi-

stri del processore e celle di memoria). Dato che, in generale, E-SWEAM è utilizzabile con qualsiasi tipo di sistema dedicato, un'analisi di questo genere sarebbe interessante se effettuata, ad esempio, su sistemi basati su *Network-on-Chip*.

Bibliografia

[1] ARM Reference Site. http://infocenter.arm.com/help/index.jsp.

[2] Open SystemC Initiative. http://www.systemc.org.

[3] Python. http://www.python.org.

[4] Sobel Mask. http://www.cee.hw.ac.uk/hipr/html/sobel.html.

[5] M. A. Aguirre, J. N. Tombs, F. Muñoz, V. Baena, H. Guzmán, J. Nápoles, A. Torralba, A. Fernández-León, F. Tortosa-López, and D. Merodio. Selective Protection Analysis Using a SEU Emulator: Testing Protocol and Case Study Over the Leon2 Processor. *IEEE Transactions on Nuclear Science*, pages 951–956, 2007.

[6] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. GOOFI : Generic Object-OrientedFault Injection Tool. In *in Proceedings of International Conference on Dependable Systems and Networks (DSN 2001)*, 2001.

[7] ARM. *ARM Architecture Reference Manual.* ARM Limited, 2005.

[8] Ron Bell. Introduction to IEC 61508. *Health and Safety Executive*, 2005.

[9] Giovanni Beltrame, Cristiana Bolchini, Luca Fossati, Antonio Miele, and Donatella Sciuto. ReSP: A Non-Intrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. *IEEE*, 2008.

[10] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. ReSP: A Non-Intrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. *IEEE*, 2009.

[11] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. EXFI: A Low-Cost Fault Injection System for Embedded Microprocessor-Based Boards. *ACM Transactions on Design Automation of Electronic Systems*, pages 626–634, 1998.

[12] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proc. IEEE/ACM/IFIP Intl. Conf. on Hardware/software codesign and system synthesis*, pages 19–24, 2003.

[13] João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, pages 125–136, 1998.

[14] Yung-Yuan Chen, Chung-Hsien Hsu, and Kuen-Long Leu. SoC-Level Risk Assessment Using FMEA Approach in System Design with SystemC. *IEEE*, 2009.

[15] Jeffrey J. Cook and Craig Zilles. A Characterization of Instruction-level Error Derating and its Implications for Error Detection. *IEEE*, 2008.

[16] David de Andrés, Juan Carlos Ruiz, Daniel Gil, and Pedro Gil. Fault Emulation for Dependability Evaluation of VLSI Systems. *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, pages 422–431, 2008.

[17] Alireza Ejlali and Seyed Ghassem Miremadi. Error propagation analysis using FPGA-based SEU-fault injection. *Microelectronics Reliability*, pages 319–328, 2008.

[18] Exida. IEC 61508 Overview Report. Technical report, Exida, 2006.

[19] Fabrizio Ferrandi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Mapping and Scheduling of Parallel C Applications with Ant Colony Optimization onto Heterogeneous Reconfigurable MPSoCs. *IEEE*, 2010.

[20] André V. Fidalgo, Gustavo R. Alves, and José M. Ferreira. Real Time Fault Injection Using a Modified Debugging Infrastructure. In *12th IEEE International On-Line Testing Symposium*, 2006.

[21] Milind Girkar and Constantine D. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE*, 1992.

[22] Siva Kumar, Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems. 2009.

[23] Dongwoo Lee and Jongwhoa Na. A Novel Simulation Fault Injection Method for Dependability Analysis. *Design for Reliability at 32 nm and Beyond*, pages 50–60, 2009.

[24] Man-Lap Li, Pradeep Ramachandran, Ulya R. Karpuzcu, Siva Kumar, Sastry Hari, and Sarita V. Adve. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. 2009.

[25] Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. SWAT: An Error Resilient System. 2008.

[26] Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. 2008.

[27] Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. 2008.

[28] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal*, 1962.

[29] Celia Lòpez-Ongil, Mario Garcìa-Valderas, Marta Portela-Garcìa, and Luis Entrena. Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation. *IEEE Transaction on Nuclear Science*, pages 252–261, 2007.

[30] Riccardo Mariani, Gabriele Boschi, and Federico Colucci. Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508. *IEEE*, 2007.

[31] Antonio Rosario Miele. *A Methodology for the Design and the Analysis of Reliable Embedded Systems*. PhD thesis, Politecnico di Milano, 2009.

[32] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. *IEEE Computer Society*, 2003.

[33] Gustav Munkby and Sibylle Schupp. Improving Fault Injection of Soft Errors using Program Dependencies. *IEEE*, 2008.

[34] Yu-Mei Niu, Yu-Zhu He, Jian-Hong Li, and Xiao-Jun Zhao. The Optimization of RPN Criticality Analysis Method in FMECA. *IEEE*, 2009.

[35] Isil Oz, Haluk Rahmi Topcuoglu, Mahmut Kandemir, and Oguz Tosun. Quantifying Thread Vulnerability for Multicore Architectures. 2010.

[36] Andrea Pellegrini, Kypros Constantinides, Dan Zhang, Shobana Sudhakar, Valeria Bertacco, and Todd Austin. CrashTest: A Fast High-Fidelity FPGA-Based Resiliency Analysis Framework. 2008.

[37] Haapanen Pentti and Helminen Atte. *Failure Mode and Effects Analysis of Software-based Automation Systems*. Stuk, 2002.

[38] Jon Perez, Mikel Azkarate-askasua, and Antonio Perez. Codesign and Simulated Fault Injection of Safety-Critical Embedded Systems Using SystemC. *IEEE*, 2010.

[39] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Modeling the Propagation of Intermittent Hardware Faults in Programs. In *Pacific Rim International Symposium on Dependable Computing*, pages 19–26, 2010.

[40] M. Rebaudengo, M. Sonza Reorda, and M. Violante. A New Approach to Software-Implemented Fault Tolerance. *Journal of Electronic Testing*, 2002.

[41] John Peter Rooney. IEC 61508: An Opportunity for Reliability. In *Annual RELIABILITY and MAINTAINABILITY Symposium*, 2001.

[42] Vilas Sridharan and David R. Kaeli. Using PVF Traces to Accelerate AVF Modeling. 2010.

[43] Jeffrey Voas and Gary McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley and Sons, 1997.

[44] Jianjun Xu, Rui Shen, and Qingping Tan. PRASE: An Approach for Program Reliability Analysis with Soft Errors. In *14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 240–247, 2008.

[45] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology*, pages 171–186, 2003.

Appendice A

Risultati della campagna di iniezione di guasti nel caso di studio EDTMR

In questo appendice sono presentate le tabelle contenenti le percentuali di ogni fallimento osservato durante gli esperimenti di simulazione nel caso di studio dell'*Edge Detector Triple Modular Redundancy* (EDTMR), discusso nel Capitolo 4. Si rimanda il lettore, per ulteriori chiarimenti, all'analisi dei risultati effettuata in Sezione 4.1.4.

Thread Principale			
RB0-1	Indirizzo immagine	Masked	23,2%
		Many Errors	42,2%
		Correct	65,4%
		Exception	34,2%
		Many Errors - Dirty Memory	0,4%
		Incorrect	34,6%

Tabella A.1: *Risultati delle iniezioni nei registri #0 e #1 del primo processore, contenenti gli indirizzi delle immagini.*

Threads Secondari			
RB0-1	Indirizzo immagine	Masked	13,9%
		Many Errors	47,4%
		Correct	61,30%
		Exception	38,4%
		Many Errors - Dirty Memory	0,3%
		Incorrect	19,9%

Tabella A.2: *Risultati delle iniezioni nei registri #0 e #1 dei processori secondari.*

Thread Principale			
RB2-3	Dimensione Immagine	Masked	29,9%
		Few Errors	55,5%
		Few Errors - Dirty Memory	0,9%
		Many Errors	1,3%
		Correct	87,6%
		Exception	12,4%
		Incorrect	12,4%

Tabella A.3: *Risultati delle iniezioni nei registri #2 e #3 del primo processore, contenenti le dimensioni dell'immagine, passate per valore.*

Threads Secondari			
RB2-3	Dimensione Immagine	Masked	26,1%
		Few Errors	59,3%
		Few Errors - Dirty Memory	1,2%
		Many Errors	2,5%
		Correct	89,1%
		Exception	10,8%
		Many Errors - Dirty Memory	0,1%
		Incorrect	10,9%

Tabella A.4: *Risultati delle iniezioni nei registri #2 e #3 dei processori secondari.*

Thread Principale			
RB11	*Link Register*	Masked	22,0%
		Correct	22,0%
		Exception	38,1%
		TSR - Timeout	39,9%
		Incorrect	78,0%

Tabella A.5: *Risultati delle iniezioni nel registro #11 del primo processore, contenente il Link Register.*

Threads Secondari			
RB11	*Link Register*	Masked	26,6%
		Correct	26,6%
		Exception	40,1%
		TSR - Timeout	33,3%
		Incorrect	73,4%

Tabella A.6: *Risultati delle iniezioni nel registro #11 dei processori secondari.*

Thread Principale			
RB11	*Stack Pointer*	Masked	30,6%
		SW Interrupt	15,2%
		Many Errors	1,4%
		Correct	47,2%
		Exception	38,5%
		SW Interrupt - Timeout	8,5%
		Many Errors - Voter Func.	5,8%
		Incorrect	52,8%

Tabella A.7: *Risultati delle iniezioni nel registro #13 del primo processore, contenente lo Stack Pointer.*

Threads Secondari			
RB11	*Stack Pointer*	Masked	26,8%
		SW Interrupt	14,8%
		Many Errors	2,2%
		Correct	43,8%
		Exception	40,1%
		SW Interrupt - Timeout	11,4%
		Many Errors - Voter Func.	4,7%
		Incorrect	56,2%

Tabella A.8: *Risultati delle iniezioni nel registro #13 dei processori secondari.*

Thread Principale			
RB11	*Program Counter*	Masked	18,3%
		Context Change	1,3%
		Task Bypassed	1,0%
		Partial Re-execution	1,4%
		Many Errors	1,6%
		Correct	23,6%
		Exception	51,7%
		Timeout	24,7%
		Incorrect	76,4%

Tabella A.9: *Risultati delle iniezioni nel registro #15 del primo processore, contenente il Program Counter.*

Threads Secondari			
RB11	*Program Counter*	Masked	19,1%
		Context Change	2,1%
		Task Bypassed	0,9%
		Partial Re-execution	0,4%
		Many Errors	2,0%
		Correct	24,5%
		Exception	51,0%
		Timeout	24,5%
		Incorrect	75,5%

Tabella A.10: *Risultati delle iniezioni nel registro #15 dei processori secondari.*

Appendice B

Risultati della campagna di iniezione di guasti nel caso di studio dell'ABS

In questo appendice sono presentate le tabelle contenenti le percentuali di ogni fallimento del sistema ABS presentato nel Capitolo 4. Si rimanda il lettore, per ulteriori chiarimenti, all'analisi dei risultati effettuata in Sezione 4.2.3.

Acquisizione - Thread Principale - RB0		
No-error	Masked	**11,2%**
Safe	Single/Multiple Data Error	66,6%
	Hardware Exception	8,9%
	Software Exception	6,7%
	TOTALE	**82,2%**
Dangerous	Multiple Data Error	5,0%
	Hardware Exception	1,1%
	Software Exception	0,5%
	TOTALE	**6,6%**

Tabella B.1: *Risultati delle iniezioni nel registro #0 del primo processore durante il task di acquisizione dei dati eseguito dal thread principale.*

Acquisizione - Thread Principale - RB1		
No-error	Masked	**24,9%**
Safe	Emergency Brake	51,5%
	Hardware Exception	12,1%
	Emergency Brake	4,9%
	TOTALE	**68,5%**
Dangerous	Emergency Brake	2,3%
	Hardware Exception	3,3%
	Software Exception	1,0%
	TOTALE	**6,6%**

Tabella B.2: *Risultati delle iniezioni nel registro #1 del primo processore durante il task di acquisizione dei dati eseguito dal thread principale.*

Acquisizione - Thread Secondario - RB0		
No-error	Masked	**12,6%**
Safe	Single/Multiple Data Error	12,4%
	Hardware Exception	31,5%
	Emergency Brake	34,7%
	TOTALE	**78,6%**
Dangerous	Multiple Data Error	1,7%
	Hardware Exception	3,9%
	Deadlock	0,2%
	Emergency Brake	2,9%
	Actuation Data Error	0,1%
	TOTALE	**8,8%**

Tabella B.3: *Risultati delle iniezioni nel registro #0 del secondo processore durante il task di acquisizione dei dati eseguito dal thread secondario.*

Elaborazione - Thread Principale - RB0		
No-error	Masked	**14,9%**
Safe	Single/Multiple Data Error	79,2%
	Hardware Exception	0,8%
	TOTALE	**80,0%**
Dangerous	Multiple Data Error	4,5%
	Hardware Exception	0,6%
	TOTALE	**5,1%**

Tabella B.4: *Risultati delle iniezioni nel registro #0 del primo processore durante il task di elaborazione dei dati eseguito dal thread principale.*

Elaborazione - Thread Principale - RB1		
No-error	Masked	**21,9%**
Safe	Emergency Brake	55,8%
	Hardware Exception	12,7%
	Emergency Brake	4,5%
	TOTALE	**73,0%**
Dangerous	Emergency Brake	2,8%
	Hardware Exception	1,5%
	Software Exception	0,8%
	TOTALE	**5,1%**

Tabella B.5: *Risultati delle iniezioni nel registro #1 del primo processore durante il task di elaborazione dei dati eseguito dal thread principale.*

Elaborazione - Thread Secondario - RB0		
No-error	Masked	**13,1%**
Safe	Single/Multiple Data Error	22,1%
	Hardware Exception	29,3%
	Emergency Brake	28,2%
	TOTALE	**79,6%**
Dangerous	Multiple Data Error	1,3%
	Hardware Exception	2,5%
	Deadlock	0,5%
	Emergency Brake	3,0%
	TOTALE	**7,3%**

Tabella B.6: *Risultati delle iniezioni nel registro #0 del secondo processore durante il task di elaborazione dei dati eseguito dal thread secondario.*

Attuazione - Thread Principale - RB0		
No-error	Masked	**21,5%**
Safe	Actuation Data Error	63,4%
	Hardware Exception	8,7%
	TOTALE	**72,1%**
Dangerous	Actuation Data Error	5,1%
	Hardware Exception	1,3%
	TOTALE	**6,4%**
Attuazione - Thread Principale - RB1		
No-error	Masked	**22,6%**
Safe	Actuation Data Error	61,6%
	Hardware Exception	7,2%
	TOTALE	**68,8%**
Dangerous	Actuation Data Error	5,9%
	Hardware Exception	2,7%
	TOTALE	**8,6%**
Attuazione - Thread Principale - RB2		
No-error	Masked	**97,8%**
Safe	Hardware Exception	1,4%
	TOTALE	**1,4%**
Dangerous	Hardware Exception	0,8%
	TOTALE	**0,8%**
Attuazione - Thread Principale - RB3		
No-error	Masked	**37,7%**
Safe	Actuation Data Error	36,9%
	Hardware Exception	13,5%
	Software Exception	4,9%
	TOTALE	**72,1%**
Dangerous	Emergency Brake	2,2%
	Hardware Exception	4,5%
	Software Exception	0,3%
	TOTALE	**7,0%**

Tabella B.7: *Risultati delle iniezioni nei registri #0-#3 del primo processore durante il task di attuazione eseguito dal thread principale.*

Acquisizione - Thread Principale - RB11		
No-error	Masked	**3,1%**
Safe	Application Abort	3,6%
	Hardware Exception	29,5%
	Software Exception	28,5%
	TOTALE	**61,6%**
Dangerous	Application Abort	4,2%
	Hardware Exception	10,5%
	Software Exception	18,1%
	Timeout	2,5%
	TOTALE	**35,3%**

Tabella B.8: *Risultati delle iniezioni nel registro #11 del primo processore durante il task di acquisizione dei dati eseguito dal thread principale.*

Acquisizione - Thread Secondario - RB11		
No-error	Masked	**4,2%**
Safe	Application Abort	8,2%
	Hardware Exception	19,0%
	Software Exception	20,1%
	TOTALE	**48,5%**
Dangerous	Application Abort	7,3%
	Hardware Exception	11,5%
	Software Exception	19,0%
	Timeout	4,2%
	TOTALE	**48,5%**

Tabella B.9: *Risultati delle iniezioni nel registro #11 del secondo processore durante il task di acquisizione dei dati eseguito dal thread secondario.*

Elaborazione - Thread Principale - RB11		
No-error	Masked	**4,2%**
Safe	Application Abort	5,0%
	Hardware Exception	27,6%
	Software Exception	30,2%
	TOTALE	**62,8%**
Dangerous	Application Abort	5,4%
	Hardware Exception	9,9%
	Software Exception	15,6%
	Timeout	2,1%
	TOTALE	**33,0%**

Tabella B.10: *Risultati delle iniezioni nel registro #11 del primo processore durante il task di elaborazione dei dati eseguito dal thread principale.*

Elaborazione - Thread Secondario - RB11		
No-error	Masked	**3,5%**
Safe	Application Abort	9,7%
	Hardware Exception	19,8%
	Software Exception	22,4%
	TOTALE	**51,9%**
Dangerous	Application Abort	8,1%
	Hardware Exception	11,1%
	Software Exception	12,6%
	Timeout	12,8%
	TOTALE	**44,6%**

Tabella B.11: *Risultati delle iniezioni nel registro #11 del secondo processore durante il task di elaborazione dei dati eseguito dal thread secondario.*

Attuazione - Thread Principale - RB11		
No-error	Masked	**3,3%**
Safe	Application Abort	7,8%
	Hardware Exception	24,4%
	Software Exception	30,0%
	TOTALE	**62,2%**
Dangerous	Application Abort	7,1%
	Hardware Exception	9,5%
	Software Exception	13,9%
	Timeout	4,0%
	TOTALE	**34,5%**

Tabella B.12: *Risultati delle iniezioni nel registro #11 del primo processore durante il task di attuazione eseguito dal thread principale.*

Acquisizione - Thread Principale - RB12-13		
No-error	Masked	**44,7%**
Safe	Hardware Exception	30,1%
	Software Exception	15,4%
	TOTALE	**45,5%**
Dangerous	Hardware Exception	6,5%
	Software Exception	3,3%
	TOTALE	**9,8%**

Tabella B.13: *Risultati delle iniezioni nei registri #12 e #13 del primo processore durante il task di acquisizione dei dati eseguito dal thread principale.*

Acquisizione - Thread Secondario - RB12-13		
No-error	Masked	**41,2%**
Safe	Hardware Exception	27,8%
	Software Exception	22,1%
	TOTALE	**49,9%**
Dangerous	Hardware Exception	4,5%
	Software Exception	4,4%
	TOTALE	**8,9%**

Tabella B.14: *Risultati delle iniezioni nei registri #12 e# 13 del secondo processore durante il task di acquisizione dei dati eseguito dal thread secondario.*

Elaborazione - Thread Principale - RB12-13		
No-error	Masked	**42,5%**
Safe	Hardware Exception	28,8%
	Software Exception	18,8%
	TOTALE	**47,6%**
Dangerous	Hardware Exception	5,9%
	Software Exception	4,0%
	TOTALE	**9,9%**

Tabella B.15: *Risultati delle iniezioni nei registri #12 e #13 del primo processore durante il task di elaborazione dei dati eseguito dal thread principale.*

Elaborazione - Thread Secondario - RB12-13		
No-error	Masked	**39,5%**
Safe	Hardware Exception	24,5%
	Software Exception	27,5%
	TOTALE	**52,0%**
Dangerous	Hardware Exception	4,4%
	Software Exception	4,1%
	TOTALE	**8,5%**

Tabella B.16: *Risultati delle iniezioni nei registri #12 e #13 del secondo processore durante il task di elaborazione dei dati eseguito dal thread secondario.*

Attuazione- Thread Principale - RB12-13		
No-error	Masked	**46,8%**
Safe	Hardware Exception	27,9%
	Software Exception	16,8%
	TOTALE	**44,7%**
Dangerous	Hardware Exception	6,0%
	Software Exception	2,5%
	TOTALE	**8,5%**

Tabella B.17: *Risultati delle iniezioni nei registri #12 e #13 del primo processore durante il task di attuazione eseguito dal thread principale.*

Acquisizione - Thread Principale - RB14-15		
No-error	Masked	**9,6%**
Safe	Hardware Exception	29,3%
	Software Exception	24,9%
	Partial Re-execution	0,2%
	Emergency Brake	1,8%
	TOTALE	**56,2%**
Dangerous	Hardware Exception	15,9%
	Software Exception	15,1%
	Timeout	3,1%
	Emergency Brake	0,1%
	TOTALE	**34,2%**

Tabella B.18: *Risultati delle iniezioni nei registri #14 e #15 del primo processore durante il task di acquisizione dei dati eseguito dal thread principale.*

Acquisizione - Thread Secondario - RB14-15		
No-error	Masked	**4,4%**
Safe	Hardware Exception	24,6%
	Software Exception	21,8%
	Single/Multiple Data Error	3,4%
	Emergency Brake	1,5%
	TOTALE	**51,3%**
Dangerous	Hardware Exception	23,0%
	Software Exception	19,4%
	Multiple Data Error	0,9%
	Illegal Jump	0,7%
	Emergency Brake	0,3%
	TOTALE	**44,3%**

Tabella B.19: *Risultati delle iniezioni nei registri #14 e #15 del secondo processore durante il task di acquisizione dei dati eseguito dal thread secondario.*

Elaborazione - Thread Principale - RB14-15		
No-error	Masked	**7,5%**
Safe	Hardware Exception	27,2%
	Software Exception	24,4%
	Partial Re-execution	0,5%
	Emergency Brake	1,1%
	Illegal Jump	2,9%
	TOTALE	**56,1%**
Dangerous	Hardware Exception	15,8%
	Software Exception	11,8%
	Timeout	7,5%
	Emergency Brake	0,5%
	Illegal Jump	0,8%
	TOTALE	**34,2%**

Tabella B.20: *Risultati delle iniezioni nei registri #14 e #15 del primo processore durante il task di elaborazione dei dati eseguito dal thread principale.*

Elaborazione - Thread Secondario - RB14-15		
No-error	Masked	**5,3%**
Safe	Hardware Exception	25,5%
	Software Exception	19,4%
	Single/Multiple Data Error	4,2%
	Emergency Brake	0,8%
	TOTALE	**49,9%**
Dangerous	Hardware Exception	21,8%
	Software Exception	20,6%
	Multiple Data Error	0,9%
	Emergency Brake	0,1%
	Emergency Brake Error	1,4%
	TOTALE	**44,8%**

Tabella B.21: *Risultati delle iniezioni nei registri #14 e #15 del secondo processore durante il task di elaborazione dei dati eseguito dal thread secondario.*

Attuazione - Thread Principale - RB14-15		
No-error	Masked	**10,5%**
Safe	Hardware Exception	24,4%
	Software Exception	22,5%
	Partial Re-execution	0,1%
	Emergency Brake	2,5%
	Illegal Jump	3,3%
	TOTALE	**52,8%**
Dangerous	Hardware Exception	15,5%
	Software Exception	14,1%
	Timeout	6,6%
	Silent Data Corruption	0,1%
	Emergency Brake Error	0,4%
	TOTALE	**36,6%**

Tabella B.22: *Risultati delle iniezioni nei registri #14 e #15 del primo processore durante il task di attuazione eseguito dal thread principale.*