Scuola di Ingegneria

Dipartimento di
Ingegneria dell'Informazione

Corso di Laurea in
Ingegneria Informatica

Ruolo della copertura MC/DC nel testing strutturale secondo lo standard DO-178

MC/DC coverage role in structural testing according to DO-178 standard

Relatore
Alessandro Fantechi

Candidato
Tommaso Gualtierotti

Anno Accademico 2021/2022

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Nel 2019 hanno preso un aereo circa 4.5 miliardi di persone, con circa 42 milioni di voli. Numeri che si sono più dimezzati nei due anni successivi a causa della pandemia ma che sono destinati a risalire. Nel 2021 sono stati effettuati 146 lanci spaziali, di cui 32 solamente da un'unica agenzia privata ovvero SpaceX. Al contempo giornalmente volano centinaia di aerei a scopo militare. Ed il trend è vertiginosamente in salita, dal momento che negli anni 70 hanno volato solamente circa 310 milioni di passeggeri, mentre riguardo ai lanci spaziali sebbene durante la corsa allo spazio contesa tra Stati Uniti e URSS si siano visti numeri simili, (infatti il record precedente di 136 lanci risaliva proprio al 1967, in vista delle preparazioni alla corsa alla Luna, dopo che il presidente Kennedy nel Settembre del 1962 annunciò la volonta di atterrare sulla Luna con umani entro la fine del decennio.); dopo il 1990 i numeri di lanci spaziali subirono un calo drastico, fino a toccare quota 54 lanci nel 2004. Ad oggi il trend è fortemente in risalita dal momento che i prezzi riguardanti i lanci spaziali sono calati fortemente e quindi la possibilità di lanciare satelliti nello spazio ad oggi è alla portata anche di aziende molto più piccole, se non anche di alcune università. Anche in ambito militare il numero di velivoli impiegati è sempre più in aumento, infatti solo gli Stati Uniti contano più di 5000 velivoli impiegati a scopo militare, di cui più di 2000 Caccia.

Tutti i mezzi aerospaziali appena citati sono di fatto un concentrato delle migliori tecnologie odierne di cui dispone l'essere umano. Infatti ad oggi la maggior parte, se non tutti i mezzi che volano nei nostri cieli e fuori dal nostro pianeta sono supportati in modo massivo da elettronica e informatica, che subentrano a supporto ed a servizio dell'uomo. Il XX secolo è stato il secolo che ha dato piede anche allo sviluppo dei primi processori e dei primi computer, che sono stati poi affiancati insieme all'elettronica di contorno nella gestione di mezzi aerospaziali.

Difatti tutti questi mezzi sono gestiti da software: ci sono molti esempi, tra più e meno famosi possono essere ricordati per esempio:

- Il programma Apollo, con il quale il 21 Luglio 1969, gli Stati Uniti e la NASA hanno portato il primo essere umano sulla Luna. Di fatti la NASA si appoggiò all'MIT di Boston nello sviluppo del Software di controllo dell'AGC (Apollo Guidance Controller) il quale gestiva sia il modulo di comando che il modulo lunare (Eagle) nelle loro funzioni. AGC fu sviluppato completamente in linguaggio Assembly.

- Il Joint Strike Fighter è un programma avviato dagli Stati Uniti per la produzione di un caccia multiruolo, da dare in forza allo US Army, prodotto dalla Lockheed Martin la quale affidò lo sviluppo del software di controllo del primo aeromobile a Bjarne Stroustrup, famoso nel mondo informatico come l'inventore del linguaggio C++.

- Falcon 9 di SpaceX è un vettore spaziale, il primo vettore spaziale della storia ad aver riutilizzato il primo stadio, infatti Falcon 9 implementa il sistema VTOL (Vertical Take-Off and Landing), ossia è in grado di atterrare in modo autonomo in rientro dallo spazio.

- Boeing 737 MAX, uno degli ultimi gioielli di casa Boeing, spesso al centro delle discussioni negli ultimi anni dopo i due incidenti in meno di un anno che hanno portato alla morte di 189 persone con il volo Lion Air 610 e 157 vittime con il volo Ethiopian Airlines 302. È stato confermato che entrambi gli incidenti sono stati causati da un software proprietario

di Boeing (MCAS), il quale a causa del guasto di un sensore non opportunamente ridondato, non ha lasciato possibilità ai piloti di portare in salvo l'aereo. Questo è uno degli esempi lampanti che rinforza la decisione di Agenzie addette alla sicurezza di introdurre degli standard sullo sviluppo di software per applicazioni aerospaziali (cfr. DO-178).

Molti altri mezzi aerospaziali potrebbero essere approfonditi, ma ci limitiamo alla sola citazione di alcuni:

- STS (Space Transportation System, meglio conosciuto come Space Shuttle),
- SLS (Space Launch System, utilizzato ad oggi nel programma lunare Artemis),
- Starship,
- New Shepard,
- F-22 Raptor,
- F-35,
- Eurofigther Typhoon,
- NH-90 (primo elicottero al mondo ad essere completamente Fly-by-Wire),
- Airbus A320.

Essendo che i software di controllo di volo (conosciuti come computer di volo, o computer di bordo), sono responsabili del corretto funzionamente del mezzo, sono quindi anche direttamente responsabili delle vite umane che sono a contatto con il mezzo (chi usa quindi il mezzo, ma anche chi rischia che la caduta di uno questi mezzi possa provocare danni fisici).

Per definizione un sistema critico è un sistema il cui ipotetico fallimento produca risultati catastrofici e quindi non accettabili. Gli effetti catastrofici non riguardano esclusivamente la vita umana, infatti generalmente si ha una distinzione in due tipologie di sistemi critici:

▷ **Sistema critico per la sicurezza**, il cui fallimento può provocare:

- ferimento o persino la perdita di vite umane;
- gravi danni ambientali;
- perdita di strutture o mezzi di valore.

▷ **Sistema critico per la missione**, il cui fallimento può causare:

- danni economici gravi ed irreparabili;
- fallimento o perdita sostanziale di efficacia di una missione.

Capitolo 2

Critical Software Development in Aerospace and DO-178 standard

La qualità del software è un tema molto caldo al giorno d'oggi, soprattutto nei campi ove un qualsiasi fallimento potrebbe portare al verificarsi di eventi catastrofici. Il campo aerospaziale è sicuramente uno di questi, perciò sono stati elaborati diversi approcci per certificare un software applicato in ambito aerospaziale. L'approccio più utilizzato e quello in atto da più tempo è sicuramente riferirsi allo standard DO-178. La principale differenza riscontrata nello sviluppo di software certificato per applicazioni aerospaziali rispetto allo sviluppo di software commerciale comune, è sicuramente il maggior costo in termini di tempo, risorse e personale dovuto ovviamente alla criticità dell'applicazione.
Secondo una ricerca del Real-Time Systems Group della Vienna University of Techonology[14], il costo[1] per lo sviluppo del software può essere diviso come segue nella figura 2.1

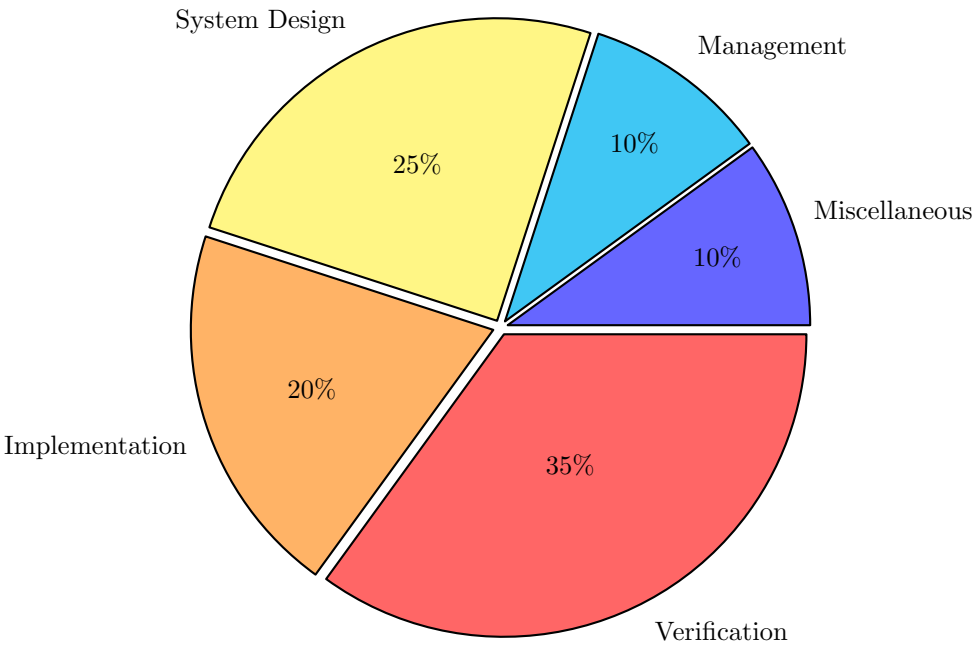

Figura 2.1: Costo del processo di sviluppo

A differenza di quello che si potrebbe pensare di primo acchito, la fase di implementazione del codice non è quella che richiede uno sforzo maggiore, ma al contrario le due fasi che richiedono maggiori risorse sono la fase di **Design and Requirement Analysis** e la fase di **Verifica** del sistema. Questo perché? Il principale motivo di questa suddivisione è riconducibile al fatto che

[1] si intende costo sia di risorse che di tempo e personale

la fase di Design è la fase in cui se si commettono degli errori di progettazione, questi errori si propagheranno per tutto il progetto, ed ovviamente questo è uno dei problemi più gravi che si possano verificare. La fase di verifica invece, richiede anch'essa uno sforzo notevole poiché è una delle fasi che stabiliscono quando il software possa essere pronto o meno per la distribuzione. Si può facilmente notare che una fase di verifica condotta in breve tempo e in modo approssimativo potrebbe non evidenziare eventuali problemi presenti, ed essere quindi un grave pericolo.

2.1 DO-178

I documenti DO-178 sono considerati lo standard per lo sviluppo (da parte delle società sviluppatrici) del software e per la certificazione (da parte di FAA e EASA) in ambito aerospaziale. Queste norme hanno introdotto un cambio radicale nell'approccio allo sviluppo e nella verifica del software critico. Infatti, se fino al 1981 non vi erano presenti norme che regolassero la procedura di sviluppo del software critico, da quell'anno e soprattutto dal 1992 con la versione DO-178B dello standard ci fu una svolta epocale nella regolazione dello sviluppo. Infatti solamente un anno dopo, nel 1992, la FAA approvò lo standard DO-178B come documento ufficiale a cui riferirsi per lo sviluppo.

2.1.1 DO-178 Development Process

Lo standard DO-178 prevede un ciclo di vita del software articolato in sei fasi:

1. Pianificazione
2. Analisi dei Requisiti
3. Design e Scrittura del codice
4. Verifica e Testing
5. Controllo della qualità
6. Certificazione

2.1.2 DO-178 DAL (Design Assurance Levels)

Nella tabella 2.1 sono indicati 5 livelli software definiti dallo standard DO-178, utilizzati per classificare ogni software secondo l'impatto che può avere sull'aeromobile in caso di malfunzionamento.

Livello	Gravità	Descrizione	Tasso Fallimento
A	Catastrofico	Guasti che possono creare un disastro aereo e la potenziale morte di tutte le persone a bordo.	10^{-9}
B	Critico	Guasti che possono causare una riduzione delle performance o della sicurezza dell'aeromobile. Guasti che possono portare alla morte o al grave ferimento delle persone a bordo.	10^{-7}
C	Maggiore	Guasti che possono ridurre performance e sicurezza dell'aeromobile o un significativo aumento del carico di lavoro richiesto ai piloti. Guasti che possono portare al ferimento di persone a bordo	10^{-5}
D	Marginale	Guasti che hanno un impatto sul carico di lavoro richiesto ai piloti. Guasti che possono provocare disagi alle persone a bordo.	10^{-3}
E	Trascurabile	Guasti che non hanno nessuna ricaduta su passeggeri e equipaggio.	Non definito

Tabella 2.1: Livelli software definiti dallo standard DO-178

Capitolo 3

Coverage

≪Our goal, then, should be to provide enough testing to ensure that the probability of failure due to hibernating bugs is low enough to accept. 'Enough' implies judgement.≫ (Beizer, Boris, 1983)

3.1 Copertura Strutturale

La copertura si riferisce alla misura in cui, una determinata attività di verifica, ha soddisfatto i suoi obiettivi. Sebbene il termine copertura solitamente sia applicato erroneamente solo alla misura di copertura in fase di testing, il termine copertura può essere esteso ed applicato a tutte le fasi di verifica.

La copertura è una misura, e non quindi un metodo oppure un test. Perciò la dicitura "Testing MC/DC" è errata, poiché MC/DC è una misura di copertura e non un modo di fare test.

Solitamente con **Copertura Strutturale** si intende misurare la percentuale di codice eseguita dai casi di test. Sono presenti varie tipologie di copertura strutturale, riguardanti anche costrutti diversi della programmazione che sono presentati nella tabella 3.1 qui di seguito, e sono poi approfonditi più nel dettaglio di seguito.

3.1.1 Definizioni operative

- **Condizione**: Una condizione è un'espressione booleana irriducibile (ovvero non può essere scomposta in espressioni booleane più semplici). Con espressione booleana irriducibile si intende sia una variabile booleana, sia un espressione data dall'utilizzo di operatori di confronto come $\geq, \leq, ==, \neq, >, <$.

- **Decisione**: Una decisione è un'espressione booleana che può essere composta da una o più condizioni.

3.1.2 Misure di Copertura Strutturale basate sul controllo del flusso

Secondo le sezioni 6.4.4.2 e 6.4.4.3 dello standard DO-178B/ED-12B gli obiettivi della copertura strutturale sono:

1. Mostrare che il grado di verifica di una certa parte di codice è consono a quello richiesto per il relativo livello software (cfr. sez. 3.2).

2. Dimostrare l'assenza di funzioni indesiderate.

3. Stabilire la completezza del testing basato sui requisiti.

Criterio di Copertura	Statement Coverage	Decision Coverage	Condition Coverage	Condition Decision Coverage	MC/DC	MCC
Ogni punto di entrata e di uscita del programma è stato invocato almeno una volta		•	•	•	•	•
Ogni istruzione del programma è stata invocata almeno una volta	•					
Ogni decisione del proramma ha assunto tutti i possibili valori almeno una volta		•		•	•	•
Ogni condizione in una decisione ha assunto tutti i possibili valori almeno una volta			•	•	•	•
Ogni condizione in una decisione è stata mostrata influenzare indipendentemente il risultato della decisione					•	• [1]
Ogni combinazione di valori di una condizione all'interno di una decisione è stata invocata almeno una volta						•

Tabella 3.1: Misure di copertura strutturale

3.1.3 Statement Coverage

Per ottenere la statement coverage, ogni istruzione del programma deve essere eseguita almeno una volta. L'ottenimento di questo tipo di copertura dimostra che tutte le istruzioni del programma sono raggiungibili. Questo tipo di copertura è ritenuto molto debole poiché non copre l'analisi delle espressioni logiche, e non capta quindi eventuali errori causati da condizioni logiche errate. Tuttavia questa tipologia di copertura fa parte di quelle richieste dallo standard DO-178, e solitamente è considerata come un requisito minimo da ottenere.
Consideriamo il seguente pezzo di codice in linguaggio C:

```
if (x > 0 && y == 0)
    z /= x;
if (z == 2 || y > 1)
    ++z;
```

Scegliendo x = 4, y = 0, z = 8 possiamo notare che il corpo di entrambi i costrutti if viene eseguito. Abbiamo così una statement coverage del 100%.

3.1.4 Decision Coverage

Per ottenere la decision coverage, sono richiesti per una condizione solamente due casi di test: uno che faccia assumere valore **True** alla decisione, ed uno che faccia assumere valore **False** alla decisione. Nonostante la Decision Coverage sia più forte della Statement Coverage anche questa copertura è ritenuta abbastanza debole, poiché potrebbe non fare distinzioni tra due diverse espressioni a seconda dei casi di test considerati.
Consideriamo il seguente pezzo di codice in linguaggio C:

```
if (A && B)
```

Scegliendo il caso A = T, B = T e il caso A = T, B = F possiamo notare che la condizione viene ad assumere sia valore True nel primo caso, sia valore False nel secondo caso. Abbiamo così una decisione coverage del 100%.

3.1.5 Condition Coverage

Questa tipologia di copertura richiede che ogni condizione all'interno di una decisione assuma tutti i possibili valori, non necessita quindi che anche la decisione assuma tutti i possibili valori.
Consideriamo nuovamente il pezzo di codice del caso precedente:

```
if (A && B)
```

Scegliendo il caso A = T, B = F ed il caso A = F, B = T possiamo notare che le due variabili assumono tutti i possibili valori, raggiungendo così una condition coverage del 100%.
Possiamo notare però che la decisione non assume valore True, perciò il ramo *then* dell'if non viene eseguito.

3.1.6 Condition/Decision Coverage

Questa tipologia di copertura invece, come si può intuire dal nome, combina le richieste della decision e della condition coverage. Si richiede quindi che tutte le condizioni all'interno della decisione assumano tutti i possibili valori e che al contempo anche la decisione assuma tutti i possibili valori.
Consideriamo nuovamente il pezzo di codice del caso precedente:

```
if (A && B)
```

Scegliendo il caso A = T, B = T ed il caso A = F, B = F possiamo notare che la decisione assume entrambi i possibili valori e che anche le condizioni (le variabili booleane) assumono sia valore True che valore False.

3.1.7 Multiple Condition Coverage

Per ottenere invece questa tipologia di copertura è necessario che una decisione assuma tutte le possibili combinazioni. Quindi essendo le decisioni composte da condizioni booleane, supponendo sia **n** il numero di condizioni, sono richiesti precisamente $\mathbf{2^n}$ casi di test. Si parla quindi di testing esaustivo.
Consideriamo nuovamente il pezzo di codice del caso precedente:

```
if (A && B)
```

In questo caso come ci indica la definizione, dobbiamo selezionare tutte le possibili combinazioni. Quindi poiché abbiamo 2 variabili avremo 4 diversi casi di test che sono:

1. A = T, B = T

2. A = F, B = T
3. A = T, B = F
4. A = F, B = F

3.2 Criterio di copertura richiesto dal DO-178 per specifico Livello Software

Criterio di Copertura	**Livello Software**
Statement Coverage	A, B, C
Decision Coverage	A, B
MC/DC	A

Tabella 3.2: Criterio di Copertura richiesto dal DO-178 per specifico livello software

Capitolo 4

Copertura MC/DC

4.1 Definizione di copertura MC/DC

Modified Condition/Decision Coverage (MC/DC) è una misura di copertura originariamente definita nello standard DO-178B. Principalmente MC/DC dice che l'insieme dei casi di test devono mostrare che ogni condizione all'interno di una decisione influenza in maniera indipendente il risultato della decisione. In termini tecnici devono essere soddisfatte le seguenti richieste:

1. Ogni punto di entrata e uscita nel programma è stato invocato almeno una volta,
2. ogni condizione in una decisione nel programma deve aver assunto tutti i possibili valori almeno una volta,
3. ogni decisione nel programma ha assunto tutti i possibili valori almeno una volta,
4. ogni condizione è stata mostrata influenzare in modo indipendente il risultato della decisione della quale fa parte:
   - 4.1 variando il valore della condizione in esame mentre vengono fissati i valori di tutte le altre condizioni della decisione.
   - 4.2 variando il valore della condizione in esame mentre vengono fissati i valori delle **sole** condizioni che possono influenzare il risultato della decisione.

Se sono soddisfatti i precedenti requisiti si può notare che sono necessari n + 1 casi di test, dove n è il numero di condizioni all'interno della decisione presa in esame.

4.2 Definizioni operative di copertura MC/DC

4.2.1 Unique-Cause MCDC

Unique-Cause MCDC richiede che la variazione del valore di un'unica condizione modifichi il risultato della decisione. Questo deve essere valido per tutte le condizioni della decisione. Qualora la decisione contenga condizioni strettamente accoppiate (cfr. sez. 4.3) non è possibile ottenere un set di test che ottenga copertura MC/DC al 100%.

4.2.2 Masking MCDC

Una condizione si dice mascherata se, modificando il suo valore, la modifica non influenza il risultato di una decisione a causa della struttura della decisione e del valore di altre condizioni. Si può quindi generare un insieme di test sfruttando questa proprietà.
Consideriamo la seguente espressione in linguaggio C:

```
A || (B && C)
```

Test	A B C	Risultato	Masking			Unique-Cause		
			A	B	C	A	B	C
1	F F F	F	5,6,7					
2	F F T	F	5,6,7	4		6	4	
3	F T F	F	5,6,7		4	7		4
4	F T T	T		2	3		2	3
5	T F F	T	1,2,3					
6	T F T	T	1,2,3			2		
7	T T F	T	1,2,3			3		
8	T T T	T						

Tabella 4.1: Tabella delle coppie per Masking MC/DC e Unique-Cause MC/DC

Dalla tabella 4.1 si può evincere la differenza nella creazione di un insieme di test per ottenere la copertura MC/DC nel caso Masking e nel caso Unique-Cause. Concentriamoci nella scelta dei casi di test per la condizione A, poiché in questo specifico caso le condizioni B e C hanno una unica possibile coppia di test in entrambi i casi; a differenza invece della condizione A che nel caso Masking ha una più ampia possibilità di scelta rispetto al caso Unique-Cause. Infatti, mentre nel caso Unique-Cause sono indivudabili solamente due coppie di test, nel caso Masking è presente un numero maggiore di possibili coppie; in più si nota che mentre le coppie individuate nel caso Unique-Cause sono presenti anche nel caso Masking, il viceversa non si verifica. La differenza principale si presenta nel fatto che nel caso Masking sono presenti coppie che tra loro variano anche più di una condizione alla volta (ricordiamo che nel caso Unique-Cause solamente una condizione alla volta può cambiare, ed è proprio la condizione che vogliamo mostrare influenzare indipendentemente il risultato), infatti nel caso Masking l'unico vincolo è il fatto che la condizione sotto test non sia mascherata. Per fare ciò quindi si ricorre a fissare il valore delle restanti condizioni all'interno dell'espressione per fare in modo che la variazione della sola condizione sotto test influenzi il risultato. Avendo quindi questo esclusivo vincolo è possibile variare il valore di più condizioni alla volta. Si nota inoltre che per testare la condizione nel caso Masking, si può creare una qualsiasi combinazione tra un elemento qualsiasi dell'insieme {1, 2, 3} ed un elemento qualsiasi dell'insieme {5, 6, 7}. Per finire si vede che è possibile creare due insiemi di test che permettano di ottenere una copertura del 100% MC/DC in entrambi i casi (e.g. {2, 3, 4, 6} oppure {2, 3, 4, 7}), tuttavia nel caso Masking c'è una maggiore possibilità di scelta.

4.2.3 Unique-Cause + Masking MCDC

Questa definizione combina le due definizioni precedenti. Più nello specifico, Masking MC/DC viene utilizzata solamente sulle condizioni strettamente accoppiate mentre su tutte le altre condizioni disaccoppiate viene fatto utilizzo della definizione Unique-Cause.

4.3 Due diverse interpretazioni della copertura MC/DC

È necessario fare una precisazione riguardo all'affermazione effettuata nella sezione 4.1 riguardo al numero di test necessari per ottenere la copertura MC/DC. Per fare ciò mostriamo due diverse interpretazioni della copertura MC/DC in relazione alle condizioni fortemente accoppiate, dando prima una definizione di condizione accoppiata.
Le condizioni accoppiate possono essere di due tipi:

- **Fortemente** accoppiate: se al variare di una condizione corrisponde la variazione di tutte le altre condizioni.
- **Debolmente** accoppiate: se alla variazione di una condizione non corrisponde sempre la variazione di tutte le altre condizioni, ma solo talvolta.

Le condizioni accoppiate rendono alcuni test infattibili, ed ovviamente questo può rendere più difficile ottenere la copertura MC/DC, risultando quindi o in una maggior difficoltà nel trovare le coppie di test per condizione oppure in una quantità di test necessari maggiore di n + 1.
Se consideriamo per esempio l'espressione in linguaggio C:

```
x > 0 && x < 10
```

si può notare che le due condizioni sono debolmente accoppiate poiché variando il valore di x da 5 a 10, varia la seconda condizione (che assume valore False) ma non varia la prima condizione che continua ad avere valore True. Mentre per esempio variando il valore di x da 0 a 10, entrambe le condizioni variano ed assumono il valore opposto. Le condizioni debolmente accoppiate spesso non sono un problema poiché o i test infattibili non concorrono alla determinazione di un insieme di test per ottenere la copertura MC/DC, o esiste un insieme di test alternativo oppure i test affetti dall'accoppiamento delle condizioni sono comunque fattibili anche se più difficili da costruire. Per esempio, il caso di test (False, False) per la decisione precedente è infattibile (infatti non esiste un valore di x che permetta ad entrambe le condizioni di assumere valore False), tuttavia non è un problema poiché questo caso di test non concorre alla determinazione di un insieme di test per il raggiungimento della copertura MC/DC. Per quanto riguarda invece le condizioni **fortemente** accoppiate, illustriamo qui di seguito due diverse interpretazioni della copertura MC/DC.

4.3.1 Weak MC/DC

La copertura *weak* MC/DC suggerisce di trattare le condizioni fortemente accoppiate come singole condizioni. Quindi, per esempio, consideriamo la seguente espressione in linguaggio C:

```
(A && !B) || (!A && B)
```

In questo caso, questa espressione sarebbe vista come avente solamente due condizioni, A e B, mentre sappiamo che dal punto di vista sintattico questa decisione è formata da quattro condizioni. Sfortunatamente l'interpretazione weak può portare ad un insieme di test debole che non è in grado di garantire la decision coverage, quindi nonostante possa garantire la copertura MC/DC, staremmo violando la gerarchia di sussunzione, che definiremo successivamente.

4.3.2 Strong MC/DC

Un'interpretazione alternativa inidcata come *strong* MC/DC suggerisce di trattare ogni condizione come una entità distinta, ma richiede di trovare dei casi di test in cui la variazione di un'istanza di una variabile abbia effetto sul risultato finale e contemporaneamente mascheri l'effetto di tutte le altre istanze della variabile. Consideriamo, per esempio, la seguente espressione in linguaggio C:

```
(A && B) || (!A && C)
```

Possiamo notare che tenendo fissato il valore di C a False, siamo sicuri che la seconda istanza della variabile A sia mascherata e che quindi solamente la prima istanza di A contribuisca alla determinazione del risultato. Similmente fissando il valore di B a False, ci assicuriamo che la prima istanza di A venga mascherata e possiamo quindi essere certi che solamente la seconda istanza di A concorra al risultato.
Qui di seguito mostriamo la tabella delle coppie[1] per questa espressione:

[1] La tabella delle coppie è una tabella che contiene le coppie di test che permettono di mostrare l'indipendenza di una condizione dal resto dell'espressione. Se per ogni condizione sono presenti coppie di test che ne mostrino l'indipendenza allora si può ottenere una copertura MC/DC del 100% secondo la definizione Unique-Cause

Caso di Test	A B C	Risultato	A_1	B	A_2	C
$\overline{\mathbf{0}}$	T T T	T		2		
$\overline{\mathbf{1}}$	T T F	T	5	3		
$\overline{\mathbf{2}}$	T F T	F		0	6	
$\overline{\mathbf{3}}$	T F F	F		1		
$\overline{\mathbf{4}}$	F T T	T				5
$\overline{\mathbf{5}}$	F T F	F	1			4
$\overline{\mathbf{6}}$	F F T	T			2	7
$\overline{\mathbf{7}}$	F F F	F				6

Tabella 4.2: Tabella delle coppie per (A **and** B) **or** (**not** A **and** C)

Come si può intuire dal nome, la *strong* MC/DC è più difficile da ottenere rispetto alla *weak* MC/DC. In certi casi potrebbero essere necessari più di N + 1 casi di test, ma, se un'espressione ammette un insieme di test per la copertura MC/DC questo ha dimensione sicuramente inferiore o uguale a 2N.
Esistono tuttavia espressioni con condizioni fortemente accoppiate che non sono testabili sotto l'interpretazione *strong* della copertura MC/DC. Consideriamo per esempio la seguente espressione in linguaggio C:

```
(A || B) && (B || (A ^ C))
```

Qui di seguito mostriamo la tabella delle coppie per questa espressione:

Caso di Test	A B C	Risultato	A_1	B	B_2	A_2	C
$\overline{\mathbf{0}}$	T T T	T			2		
$\overline{\mathbf{1}}$	T T F	T					
$\overline{\mathbf{2}}$	T F T	F			0		3
$\overline{\mathbf{3}}$	T F F	T					2
$\overline{\mathbf{4}}$	F T T	T		6			
$\overline{\mathbf{5}}$	F T F	T					
$\overline{\mathbf{6}}$	F F T	F		4			
$\overline{\mathbf{7}}$	F F F	F					

Tabella 4.3: Tabella delle coppie per (A **or** B) **and** (B **or** (A **xor** C))

Si può notare dalla tabella delle coppie soprastante che non esiste alcun caso di test che permette di mostrare l'indipendenza di nessuna delle due istante di A all'interno dell'espressione. Tuttavia spesso queste espressioni possono essere riscritte in espressioni semplificate che possono essere testate sotto questa interpretazione, ciononostante non si sa se determinate espressioni possano essere sempre riscritte in una forma semplificata.

4.4 MC/DC con logica di cortocircuito

Quando utilizziamo operatori **and** ed **or** standard, entrambi gli operandi dell'espressione vengono valutati. Esistono però alcuni linguaggi di programmazione (tra cui Ada, C, C++ fra quelli più utilizzati in ambito di sviluppo di software critico) che permettono di utilizzare questi operatori in *cortocircuito*, ovvero, vengono valutati solo quegli operandi che concorrono alla determinazione del risultato dell'espressione booleana. Due banali esempi con gli operatori **and** e **or** sono mostrati qui di seguito.
Siano A e B due variabili booleane, che possono assumere valore **True** o **False**.

- A **and** B: ricordando come funziona l'operatore logico *and*, si può notare che una variabile tra A e B che assuma valore **False**, forza il risultato dell'espressione a **False**, rendendo quindi il valore dell'altra variabile non necessario a determinare il valore dell'espressione.

- A **or** B: stessa situazione del caso precedente, anche se in questo caso è il valore **True** a permettere di non valutare il valore dell'altra variabile.

L'utilizzo di operatori di cortocircuito ha un effetto rilevante sulla copertura MC/DC, ovvero viene suddiviso l'insieme di tutti i possibili casi di test in un insieme di classi di casi di test equivalenti, dove due casi di test che appartengano alla stessa classe, eseguiranno lo stesso cammino nell'object code.
Consideriamo le seguenti espressioni in linguaggio Ada:

A and then (B or C)

Test	A B C	Risultato
1	T T T	T
2	T T F	T
3	T F T	T
4	T F F	F
5	F××	F

Tabella 4.4: Operatore **and** di cortocircuito

A and (B or C)

Test	A B C	Risultato
1	T T T	T
2	T T F	T
3	T F T	T
4	T F F	F
5	F T T	F
6	F T F	F
7	F F T	F
8	F F F	F

Tabella 4.5: Operatore **and** standard

Osservando le tabelle 4.4 e 4.5, grazie anche all'aiuto dei colori, possiamo riscontrare l'effetto dell'utilizzo di operatori di cortocircuito. Infatti come precedentemente detto, gli operatori di corto circuito suddividono l'insieme dei casi di test in un insieme di classi equivalenti. In questo caso possiamo notare che il caso di test numero 5 nella tabella 4.4, ovvero quella relativa all'espressione contenente l'operatore *and* di corto circuito, racchiude all'interno i casi di test dal numero 5 al numero 8 presenti nella tabella 4.5, ovvero quella relativa all'espressione con l'utilizzo di operatori logici standard.

Bisogna tuttavia prestare attenzione al linguaggio che stiamo utilizzando ed anche a come questa proprietà è stata definita. Infatti sebbene nella maggioranza dei casi l'operando che si trova a sinistra nell'espressione venga valutato per primo, in altri casi potrebbe essere valutato in modo diverso; tutto ciò dipende quindi dalla definizione di tale proprietà all'interno del linguaggio.

4.5 Probabilità di rilevazione degli errori

≪Program testing can be used to show the presence of bugs, but never to show their absence!≫(Edsger W. Dijkstra)

Una fase di testing ideale dovrebbe essere in grado di rilevare qualsiasi errore presente nel sistema ma realisticamente questo non è possibile. Il nostro obiettivo nella pratica quindi diventa:

Generare un insieme di casi di test più piccolo possibile in grado di rilevare più errori possibili.

In [1], differenti misure di copertura vengono comparate e viene stilata una gerarchia di sussunzione[2], come possiamo vedere in figura 4.1. Viene affermato che MC/DC è più sensibile agli errori nella codifica di un singolo operando rispetto a Decision e Condition/Decision coverage.

[2]Sussunzione: date due misure di copertura A e B, la misura A si dice sussumere la misura B se e solo se ogni insieme di test che soddisfa A, soddisfa anche B.

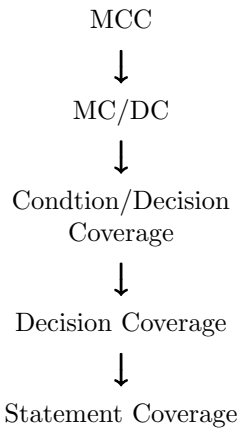

Figura 4.1: Gerarchia di Sussunzione per misure di copertura sul controllo del flusso

Inoltre la **probabilità** di rilevare un errore è data come funzione del numero di test eseguiti. Dato un insieme di M test distinti, la **probabilità** $P_{(N,M)}$ di rilevare un errore in una **implementazione errata** di un espressione booleana con N condizioni è data da:

$$P_{(N,M)} = 1 - \left[\frac{2^{(2^N - M)} - 1}{2^{2^N}} \right] \tag{4.1}$$

L'andamento della funzione sopra definita è mostrato in figura 4.2.

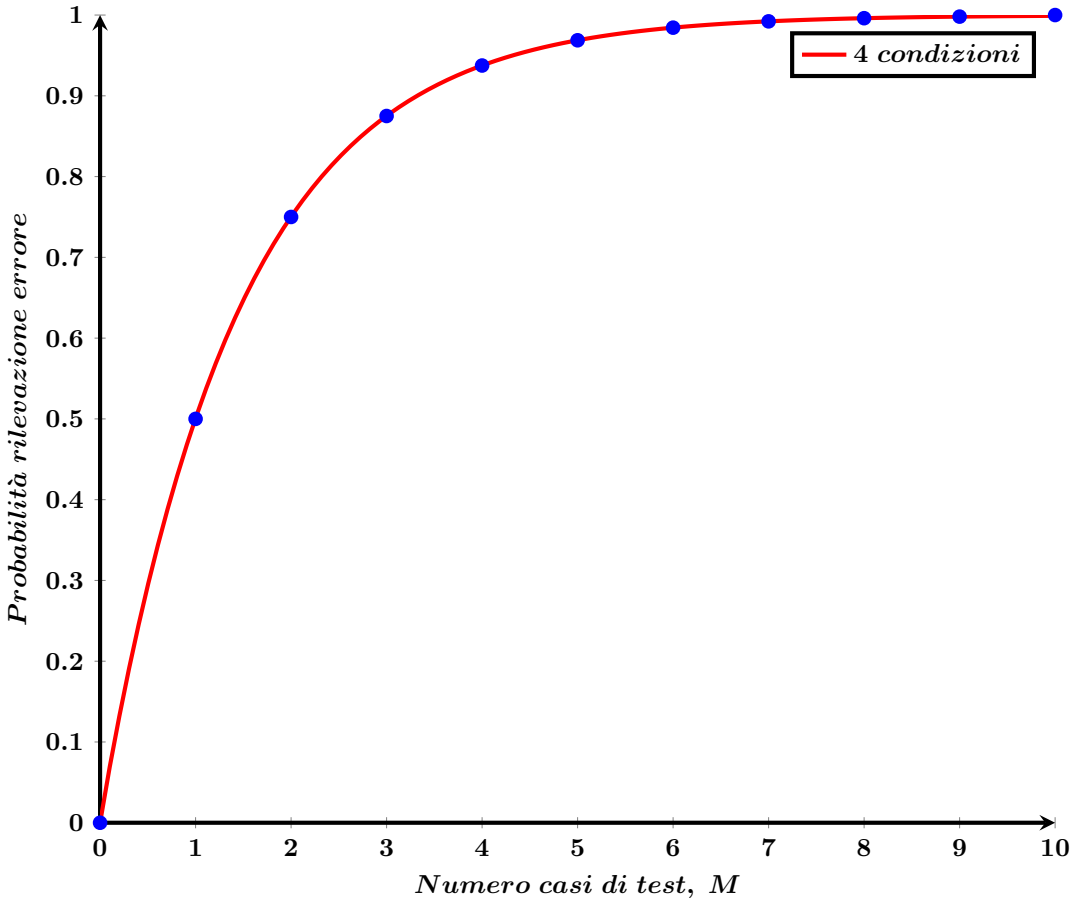

Figura 4.2: Probabilità di rilevazione di un errore con MC/DC su decisione con 4 condizioni (N = 4)

Una cosa importante a cui fare attenzione è la *probabilità relativamente bassa* di rilevare errori con solo ***due*** casi di test, come è invece normalmente richiesto in caso di condition coverage o decision/condition coverage.
Riguardo la funzione si può notare che:

1. alla crescita di $\boldsymbol{M}$, la probabilità di rilevazione tende ad 1;
2. alla crescita di $\boldsymbol{N}$, la funzione converge rapidamente a $\mathbf{1 - 1/2}^{\boldsymbol{M}}$.

Guardando quindi il punto 2 dell'elenco soprastante si può supporre che verosimilmente all'aumentare di $\boldsymbol{N}$, la probabilità di rilevare un errore in una decisione con $\boldsymbol{N}$ condizioni con $\boldsymbol{N + 1}$ casi di test cresce. Ciò appena detto potrebbe sembrare controintuitivo, ma si vede che all'aumentare di $\boldsymbol{N}$ cresce ovviamente anche il numero di casi di test; mentre la sensibilità all'errore rimane abbastanza stabile.

4.5.1 Esempio di copertura MC/DC

Consideriamo la seguente espressione booleana in linguaggio C:

```
A && (B || C)
```

Caso di Test	A B C	A && (B ‖ C)	(A && B) ‖ C	(A ‖ B) ⊕ C
$\overline{\mathbf{0}}$	0 0 0	0	0	0
$\overline{\mathbf{1}}$	0 0 1	0	1	1
$\overline{\mathbf{2}}$	0 1 0	0	0	0
$\overline{\mathbf{3}}$	0 1 1	0	1	1
$\overline{\mathbf{4}}$	1 0 0	0	0	0
$\overline{\mathbf{5}}$	1 0 1	1	1	1
$\overline{\mathbf{6}}$	1 1 0	1	1	1
$\overline{\mathbf{7}}$	1 1 1	1	1	0

Tabella 4.6: Tabella di verità per l'espressione dell'esempio precedente

L'insieme di test che ci permetta di raggiungere il 100% della copertura MC/DC può essere costruito nel modo seguente:
Ricordando che è necessario mostrare l'indipendenza di ogni condizione, possiamo notare[3] dalla tabella 4.6 le seguenti coppie di casi di test indipendenti:

A : ($\overline{\mathbf{1}}$,$\overline{\mathbf{5}}$), ($\overline{\mathbf{2}}$,$\overline{\mathbf{6}}$) e ($\overline{\mathbf{3}}$,$\overline{\mathbf{7}}$).

B : ($\overline{\mathbf{4}}$,$\overline{\mathbf{6}}$).

C : ($\overline{\mathbf{4}}$,$\overline{\mathbf{5}}$).

Perciò abbiamo un insieme di test per MC/DC che consiste nei casi $\overline{\mathbf{4}}$, $\overline{\mathbf{5}}$, $\overline{\mathbf{6}}$ ed in aggiunta uno a scelta tra $\overline{\mathbf{1}}$ e $\overline{\mathbf{2}}$.[4]

4.5.2 Controesempio

In contrasto all'assunzione precedente, ovvero l'equazione (4.1) e la gerarchia di sussunzione mostrata in figura 4.1, è possibilie costruire un esempio che mostri che la probabilità di rilevazione dell'errore è pressoché nulla per una decisione con una espressione complessa.
Consideriamo per esempio l'espressione in linguaggio C:

[3] Ricordiamo che per mostrare l'indipendenza bisogna mantenere fissi i valori delle condizioni non considerate, e, variando il valore della variabile di interesse, il valore dell'espressione deve cambiare.
[4] ricordiamo infatti che per n condizioni abbiamo bisogno di almeno $n + 1$ casi di test.

```
(A && B) || C
```

Facendo riferimento alla tabella 4.6, un insieme di test per ottenere copertura MC/DC del 100% è composta dai casi $\overline{\mathbf{2}}$, $\overline{\mathbf{4}}$, $\overline{\mathbf{6}}$ più uno a scelta tra $\overline{\mathbf{1}}$, $\overline{\mathbf{3}}$, $\overline{\mathbf{5}}$. A questo punto, mutando semplicemente l'operatore logico OR in un XOR, l'unico risultato dell'espressione che cambia è quello del caso $\overline{\mathbf{7}}$. Così facendo, un eventuale errore che si verifichi nel caso $\overline{\mathbf{7}}$ non viene rilevato da un insieme di test per la copertura MC/DC poiché possiamo facilmente notare che il caso $\overline{\mathbf{7}}$ è l'unico che non concorre alla formazione dell'insieme di test per l'espressione sopra definita. Può inoltre essere mostrato che un insieme di test per l'ottenimento della decision coverage avrebbe rilevato l'errore con una probabilità del 20%.[5]

4.6 MC/DC vs. MCC

Nei sistemi avionici, è comune trovare espressioni Booleane molto complesse. Il numero di condizioni può facilmente superare la decina in software di livello A. La tabella 4.7 mostra il numero di espressioni booleane con n condizioni per tutte la logica presente in 5 sistemi software di Livello A a bordo di due differenti aeromobili nel 1995. Si può notare che sono presenti addirittura 2 espressioni che hanno un numero di condizioni variabile tra 36 e 76. La figura 4.3 mostra l'andamento delle funzioni che descrivono il numero di casi di test necessari, rispetto al numero di condizioni, per i due criteri più completi, ovvero Multiple Condition Coverage (MCC) e Modified Condition/Decision Coverage (MC/DC). Si nota facilmente che MCC avendo una funzione esponenziale all'aumentare del numero di condizioni rende impraticabile la conduzione completa della campagna di test. Questo mostra quindi l'efficacia e l'utilità di un criterio come MC/DC che permette di condurre una campagna di test completa mantenendo dei limiti ragionevoli, nello specifico si ha un andamento lineare, quindi un numero di casi di test totali che aumenta in modo lineare rispetto al numero di condizioni. Riferendosi quindi ai valori della tabella 4.7 si può vedere che nel caso in cui n = 76, $\mathbf{\#\{MCC\} = 2^{76} \approx 7 \times 10^{22}}$, mentre $\mathbf{\#\{MC/DC\} = 2 \times 76 = 152}$. La differenza è netta, ci sono circa 20 ordini di grandezza differenti. Supponendo per esempio che sia necessario 1 millisecondo per la conduzione di 1 caso di test, il tempo di esecuzione di MCC su 76 condizioni è circa $\mathbf{10^{19}}$ secondi, ovvero un tempo maggiore dell'età dell'universo ad oggi. Si parla quindi di un tempo infinitamente grande. Usando invece il criterio MC/DC si otterrebbe un tempo di esecuzione circa pari a 0.15 secondi. Si tratta quindi di una differenza inestimabile.

	Numero di condizioni, n									
	1	2	3	4	5	6-10	11-15	16-20	21-35	36-76
numero di espressioni booleane con n condizioni	16491	2262	685	391	131	219	35	36	4	2

Tabella 4.7: Numero di espressioni booleane con n condizioni in Software di Livello A di 5 sistemi avionici differenti

[5] si ha quindi un controesempio anche per ciò che viene espresso nella figura 4.1

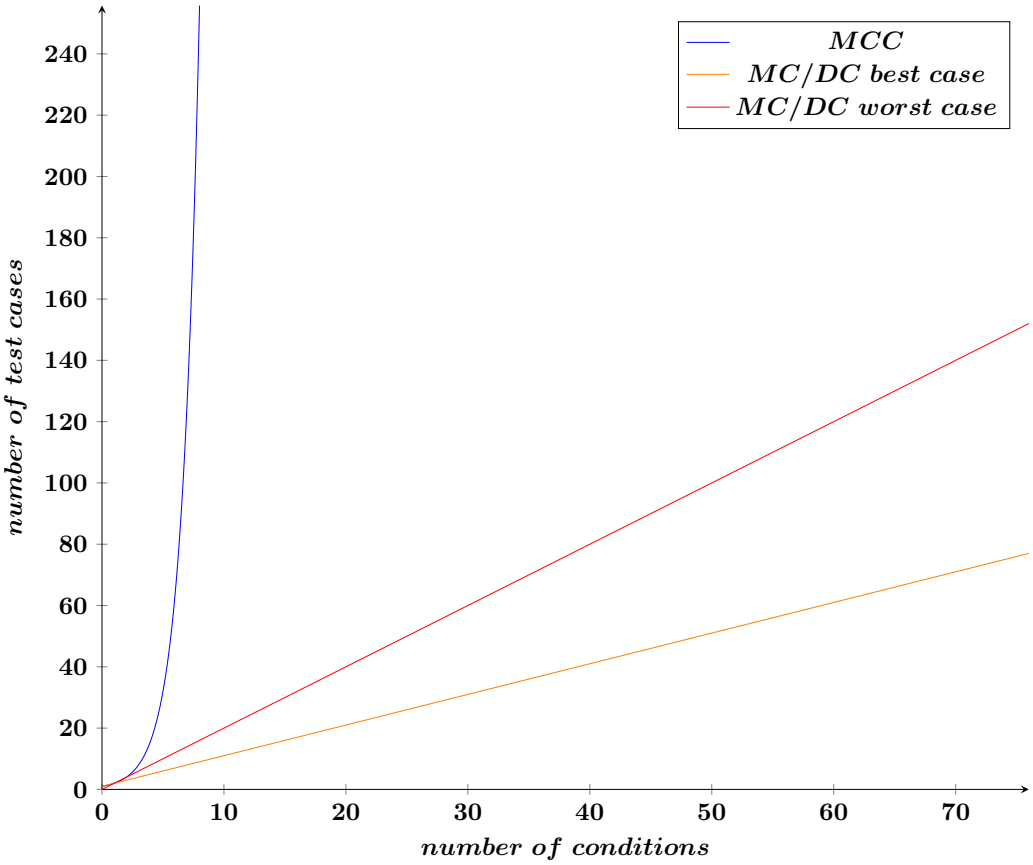

Figura 4.3: Numero di casi di test, in base al numero di condizioni per MCC ed MC/DC

Capitolo 5

Confronto di due Code Coverage tools

Abbiamo già parlato nella sezione 3.1 di cosa sia la copertura del codice, e di quale utilità abbia, soprattutto nell'ambito che stiamo trattando. In caso di programmi di ingente dimensione, si può intuire che la fase di verifica, in questo caso nello specifico la fase di analisi della copertura, può diventare un'operazione altamente dispendiosa, soprattutto in termini di tempo e di conseguenza anche in termini economici. Perciò, sono disponibili sul mercato degli strumenti che aiutino, o che sostituiscano l'addetto alla verifica. Questi strumenti sono di grande utilità, poiché se la fase di verifica fosse sostenuta interamente in modo manuale, o comunque interamente da una persona fisica senza l'appoggio di una macchina, sarebbe molto prona agli errori. Questi strumenti sono quindi utili e fondamentali ad oggi nel ciclo di sviluppo di software safety critical. È necessario tuttavia che questi strumenti siano qualificati, ovvero che siano soddisfatti dei requisiti che attestino che il dato strumento sia conforme ad un certo standard. Nel nostro caso lo standard DO-178, il quale fornisce un insieme di requisiti da soddisfare per poter qualificare un determinato strumento.
Qui di seguito introduciamo due diversi strumenti per l'analisi della copertura del codice secondo molteplici criteri. Partiremo con l'introdurre singolarmente entrambi i tools, parlando delle caratteristiche principali. Continueremo poi introducendo un piccolo esperimento che condurremo su un codice sorgente di esempio, analizzando, se presenti, le eventuali differenze tra i due tools. In realtà nell'ambito di questo lavoro di tesi sono stati presi in considerazione altri tools di questo tipo, che sono poi stati scartati per diversi motivi. quindi prima di tutto elenchiamo i tools considerati e spieghiamo brevemente il motivo della scelta di alcuni piuttosto che altri.

- **RapiCover** (L'obiettivo dell'esperimento sarà anche confrontare i tools sulla semplicità di utilizzo, ma la prova gratuita di questo strumento veniva fornita solamente previa videoconferenza per illustrare l'utilizzo del tool, quindi il tool non è stato selezionato assumendone una bassa semplicità di utilizzo.)

- **Reactis** (La prova gratuita di 30 giorni viene fornita su richiesta e il programma presenta tutte le caratteristiche utili al nostro esperimento, per questi motivi è stato selezionato.)

- **MCDC** (Questo tool è reperibile su GitHub, essendo un progetto open source. Il problema che ha portato alla rinuncia a questo tool è il fatto che il tool non venisse incontro alle necessità dell'esperimento poiché non calcola la copertura del codice dato un insieme di test eseguito sul codice, ma piuttosto data la decisione booleana da testare, il tool fornisce un insieme minimo di test per raggiungere la copertura massima.)

- **COEMS MC/DC** (Questo strumento nonostante sia disponibile solamente per scopi accademici, può essere eseguito solamente su calcolatori con processore Intel. Essendo i test stati condotti su un calcolatore con processore AMD si è dovuto scartare in partenza questo strumento.)

- ▷ **VectorCast** (Di questo strumento, sebbene avesse tutte le caratteristiche in regola per la conduzione dell'esperimento, non è stata fornita la prova gratuita dall'azienda produttrice.)

- ▷ **Testwell CTC++** (Questo strumento viene fornito su richiesta per uso accademico, previa sottoscrizione di un form che deve essere compilato da un docente universitario. La prova dura 30 giorni e lo strumento presenta tutte le caratteristiche necessarie per poter condurre l'esperimento. Per questi motivi lo strumento è stato selezionato.)

- ▷ **PARASOFT C/C++ TEST** (Questo strumento nonostante avesse tutte le carte in regola per l'esperimento non è stato selezionato poiché alla richiesta di ricevere una prova gratuita, non è seguita alcuna risposta.)

- ▷ **CANTATA** (Questo programma ha riscontrato lo stesso problema incontrato con il primo programma dell'elenco, ovvero RapiCover.)

5.1 Reactis for C

Reactis for C, è uno strumento automatizzato di debugging e di generazione di test che permette di individuare eventuali bugs in programmi sviluppati in linguaggio C.
Reactis è composto di tre strumenti principali: un tester, un simulatore ed un validatore. Il tester è responsabile della generazione automatica di una suite di test, il simulatore invece simula l'esecuzione passo passo della suite di test sul programma evidenziando, tramite diversi criteri, il codice eseguito; infine il validatore ha il compito di produrre una suite di test che viola dei requisiti definiti dall'utente. Nel nostro caso utilizzeremo solamente il validatore, simulando l'esecuzione su un insieme di test definito manualmente e controlleremo la copertura del codice secondo i criteri di nostro interesse. L'utilizzo è semplice e consiste nell'individuare quella che è definita la funzione di ingresso, ovvero la funzione che viene eseguita non appena entra in funzione il simulatore, e definire una lista di parametri di input e di output. Fatto ciò si può caricare la suite di test ed eseguire il programma, alla fine dell'esecuzione si può mostrare un report che evidenzia i risultati di copertura sui principali criteri.

5.2 Testwell CTC++

Testwell CTC++ è uno strumento che permette di misurare la copertura del codice tramite suite di test definiti dall'utente. Questo tool si compone di due principali strumenti, un preprocessore ed un postprocessore. Il preprocessore è responsabile di compilare i vari file sorgente e linkarli in un eseguibile che verrà poi eseguito e sul quale verranno condotti i test. Più nello specifico il preprocessore quando invocato da linea di comando prende in ingresso dei parametri per stabilire il criterio di copertura sul quale vogliamo analizzare il codice[1]. Questo strumento tuttavia funziona in modo diverso dal precedente, infatti qui non esiste una funzione di ingresso, ma la funzione di ingresso è il sempre presente main() di tutti i programmi in linguaggio C. Infatti il programma viene eseguito normalmente come ad ogni utilizzo, finita l'esecuzione del programma, Testwell CTC++ genera dei file che contengono informazioni riguardo all'esecuzione. Ripetute esecuzioni del programma con input diversi o suite di test diverse, non sovrascrivono questi file ma anzi li aggiornano. Quando poi si vuole effettivamente avere un riscontro sull'esecuzione del codice entra in gioco il postprocessore il quale viene istruito sul criterio di copertura che vogliamo visualizzare nel report di esecuzione. Fatto ciò si può con un comando apposito mostrare in formato HTML il contenuto del file di report che abbiamo generato con il postprocessore. Da notare anche che questo strumento permette di misurare la copertura sia su host che su target e sono inoltre disponibili molteplici compilatori e cross-compilatori che permettono di verificare programmi su tutte le principali architetture (vedi ARM e x86).

[1] si tenga conto del fatto che viene rispettata la gerarchia di sussunzione mostrata in figura 4.1, quindi se si instrumenta il codice per la copertura MCC, in automatico il programma calcolerà anche tutti i criteri di copertura sottostanti

5.3 Introduzione all'esperimento

L'esperimento che viene condotto qui di seguito, si basa su una funzione scritta in linguaggio C, che simula le varie modalità di un cruise control. Il codice di tale funzione è stato estrapolato dal codice sorgente disponibile come esempio fornito da Reactis. L'esperimento viene condotto su tre versioni differenti della funzione, denominate A, B e C. L'obiettivo è valutare le differenze tra i due strumenti e soprattutto capire il modo nel quale i due strumenti valutano le decisioni. Infatti come specificato in [4], uno dei temi che sono più al centro di discussione è il fatto che, **decisione** non è sinonimo di punto di branch (come può esserlo un costrutto if all'interno di un linguaggio), una decisione è una qualsiasi espressione Booleana, e perciò la copertura MC/DC si applica a tutte le decisioni presenti all'interno del codice sotto test, e non solo alle espressioni interne ad un costrutto condizionale. Qui di seguito viene mostrata una scaletta con le varie fasi di sperimentazione.

5.3.1 Fasi dell'esperimento

Nelle versioni A e B, le condizioni sono esplicitate all'interno di un costrutto if. L'unica differenza che interessa le versione A e B, è il fatto che la versione B accorpa due rami del costrutto if, i quali eseguono lo stesso statement, in uno unico mettendo in OR quelle che prima erano due decisioni indipendenti. La versione C invece calcola tutte le decisioni presenti nei vari rami del costrutto if della versione A come espressioni booleane e le usa all'interno di un costrutto if, costruito come quello presente nella versione B.

Qui di seguito sono indicati gli obiettivi dell'esperimento:

1. Se un insieme di test chiamato T_A ottiene la copertura MC/DC del 100% sulla versione A, si valuti se si ottiene la copertura MC/DC del 100% anche sulla versione B.

    1.1 In caso affermativo, vedere se è possibile togliere un caso di test per ottenere un nuovo insieme di test denominato T_B che non permetta di ottenere la copertura MC/DC del 100% sulla versione B.

    1.2 In caso contrario, aggiungere a T_A i casi di test necessari per ottenere T_B, il quale permetta di ottenere una copertura MC/DC del 100% sulla versione B.

2. Di norma T_A dovrebbe ottenere una copertura MC/DC del 100% anche sulla versione C, oltre al 100% della decision coverage. Si eliminino adesso dei casi di test da T_A per ottenere T_C, in modo che applicando T_C sulla versione A si ottenga una copertura MC/DC $<$ del 100%, ma si continui ad avere il 100% della decision coverage. Valutare adesso il grado di copertura MC/DC esercitando T_C sulla versione C.

5.3.2 Codice sotto esame

Nelle figure 5.1, 5.2, 5.3 mostriamo il codice delle tre varianti della funzione.

```
int g_dsMode = M_NOTINIT;

int Mode (int deactivate, int activate, int onOff, int set)
{
    if ( g_dsMode==M_OFF && onOff )
        g_dsMode = M_INIT;
    else if ( g_dsMode==M_NOTINIT || !onOff )
        g_dsMode = M_OFF;
    else if ( g_dsMode==M_INIT && (set && !deactivate) )
        g_dsMode = M_ACTIVE;
    else if ( g_dsMode==M_INACTIVE && activate )
        g_dsMode = M_ACTIVE;
    else if ( g_dsMode==M_ACTIVE && deactivate )
        g_dsMode = M_INACTIVE;

    return g_dsMode;
}
```

Figura 5.1: Versione A

```
int g_dsMode = M_NOTINIT;

int Mode (int deactivate, int activate, int onOff, int set)
{
    if ( g_dsMode==M_OFF && onOff )
        g_dsMode = M_INIT;

    else if ( g_dsMode==M_NOTINIT || !onOff )
        g_dsMode = M_OFF;

    else if ( (g_dsMode==M_INIT && (set && !deactivate) ) ||
                ( g_dsMode==M_INACTIVE && activate ) )

        g_dsMode = M_ACTIVE;
    else if ( g_dsMode==M_ACTIVE && deactivate )

        g_dsMode = M_INACTIVE;

    return g_dsMode;
}
```

Figura 5.2: Versione B

```
int g_dsMode = M_NOTINIT;

int Mode(int deactivate, int activate, int onOff, int set)
{
    int caseA = (g_dsMode==M_NOTINIT || !onOff );
    int caseB = (g_dsMode==M_OFF && onOff );
    int caseC = (g_dsMode==M_INIT && (set && !deactivate) );
    int caseD = (g_dsMode==M_INACTIVE && activate);
    int caseE = (g_dsMode==M_ACTIVE && deactivate);

    if (caseA)
        g_dsMode = M_OFF;

    else if (caseB)
        g_dsMode = M_INIT;

    else if (caseC || caseD)
        g_dsMode = M_ACTIVE;

    else if (caseE)
        g_dsMode = M_INACTIVE;

    return g_dsMode;
}
```

Figura 5.3: Versione C

5.3.3 Generazione dei casi di test

Sebbene Reactis permetta di generare un insieme di test in automatico, in questo caso l'insieme di test è stato generato manualmente, e nello specifico sono stati generati test set di dimensione minima. Il test set T_A è un insieme di dimensione 10, mentre il test set T_C è un insieme di dimensione 6.

Caso di Test	deactivate	activate	onOff	set
1	0	1	1	1
2	1	0	0	0
3	0	0	1	0
4	0	0	1	0
5	1	0	1	1
6	0	0	1	1
7	0	0	1	0
8	1	0	1	0
9	0	0	1	0
10	0	1	1	0

Tabella 5.1: T_A

Caso di Test	deactivate	activate	onOff	set
1	0	1	1	1
2	0	0	1	0
3	0	0	1	1
4	1	0	1	0
5	0	0	1	0
6	0	1	1	0

Tabella 5.2: T_C

5.4 Risultati dell'esperimento

I risultati, a differenza di come espresso negli obiettivi, non saranno espressi in percentuale ma saranno espressi come rapporto tra $\frac{\text{condizioni soddisfatte}}{\text{totale condizioni}}$. Questo perché, diverse versioni possono avere un numero diverso di condizioni, e quindi concettualmente non è corretto guardare solo la percentuale, poiché in caso di percentuale $\neq 100\%$, lo stesso numero di condizioni non soddisfatte in due versioni diverse della funzione potrebbero risultare in due percentuali diverse, rendendo l'interpretazione del risultato fuorviante. Per facilitare la lettura, sono stati evidenziati di colore verde i casi in cui la copertura risulta essere massima. Col colore arancione invece sono evidenziati i casi in cui la copertura passa dal massimo a un valore più basso. Per finire con il colore rosso invece i casi in cui la copertura non ha il valore massimo. Tuttavia la seguente tabella di colorazione viene riportata successivamente nella legenda sottostante.

Testwell CTC++ outcomes

	DC	MC/DC	MCC
T_A	12/12	23/23	28/28
T_C	12/12	18/23	23/28

Tabella 5.3: Versione A

	DC	MC/DC	MCC
T_A	10/10	21/21	25/29
T_C	10/10	16/21	20/29

Tabella 5.4: Versione B

	DC	MC/DC	MCC
T_A	10/10	33/33	39/39
T_C	10/10	28/33	34/39

Tabella 5.5: Versione C

Reactis outcomes

	DC	MC/DC	MCC
T_A	10/10	11/11	16/16
T_C	10/10	6/11	11/16

Tabella 5.6: Versione A

	DC	MC/DC	MCC
T_A	8/8	11/11	15/19
T_C	8/8	6/11	10/19

Tabella 5.7: Versione B

	DC	MC/DC	MCC
T_A	8/8	5/5	9/9
T_C	8/8	5/5	9/9

Tabella 5.8: Versione C

Legenda: **100% invariato** **100% diminuito** **<100% diminuito**

Come si può notare, a differenza degli obiettivi indicati nella sezione 5.3.1, nelle tabelle sovrastanti non sono riportati risultati eseguiti con l'insieme di test T_B. Questa scelta è stata presa per un semplice motivo: essendo T_A un insieme di test minimale, si può facilmente notare che alla rimozione casuale di uno qualsiasi dei casi di test, inevitabilmente la copertura è destinata a scendere. Bisogna però evidenziare il fatto che non tutti i casi di test se rimossi portano alla stessa copertura. Per esempio infatti, se viene rimosso il caso di test numero 6 di T_A, sulla versione A la copertura dal 100%, scende fino a 13/23 condizioni testate. Si nota quindi che rimuovendo un solo caso di test la copertura scende quasi del 50%. Ci sono casi poi in cui rimuovendo il caso di test il numero di condizioni testate scenda solamente a 22/23. Detto ciò, i risultati singoli per ogni situazione, seppur siano stati calcolati, non verranno riportati di seguito per non appesantire la lettura; ma al contrario ci concentreremo sulla discussione dei risultati sugli insiemi di test T_A e T_C.

5.4.1 Commento ed Analisi dei Risultati

In questa sezione confronteremo e commenteremo i risultati presentati nelle tabelle della sezione 5.4. I risultati saranno confrontati, sia tra le diverse versioni sullo stesso tool, sia tra stesse versioni e tools diversi.

Confronto risultati Testwell CTC++

Come possiamo vedere nelle tabelle 5.3, 5.4 e 5.5, la differenza tra i due insiemi di test rispetta quelle che erano le aspettative. Infatti essendo T_A un insieme di test minimo, il quale però permette il raggiungimento del 100% della copertura MC/DC, se confrontato con T_C il

quale ha una dimensione inferiore a T_A, è normale aspettarsi che T_C non sia in grado di raggiungere la copertura MC/DC del 100%. Detto ciò si può notare che con entrambi gli insiemi di test, la decision coverage è coperta in tutte e tre le versioni della funzione, ciò sta a rimarcare la maggior debolezza di tale copertura in confronto alla copertura MC/DC. Si può inoltre intuire che il programma non fa distinzioni tra decisioni contenute all'interno di un costrutto if, oppure decisioni presentate come espressioni di assegnamento di una variabile. Questo si può affermare notando che l'insieme di test T_C, quando esercitato su una qualsiasi delle versioni, faccia sì che sempre 5 condizioni non siano testate. Questo porta quindi ad affermare che il programma si comporti quindi nel modo corretto secondo lo standard DO-178 e secondo [4], facendo notare che anche variando l'implementazione della funzione il risultato rimane il medesimo.

Confronto risultati Reactis for C

Osservando in questo caso le tabelle 5.6, 5.7, 5.8, si può notare che a livello di decision coverage la situazione è rimasta invariata rispetto all'esperimento condotto con Testwell CTC++. Si ottiene la massima copertura infatti su tutte le versioni con entrambi gli insiemi di test. Ancora una volta la copertura MC/DC sia sulla versione A che B rispecchia i risultati riscontrati nel caso precedente. Più precisamente ancora una volta precisamente 5 condizioni non vengono testate; si ha quindi lo stesso modus operandi del precedente. Passando alla versione C si può notare che a differenza delle altre due versioni, in questo caso la copertura MC/DC è del 100% oltre che con l'insieme di test T_A anche con T_C. Viene presentato quindi qui il fulcro dell'esperimento; si può infatti affermare che Reactis tratta in modo diverso le decisioni, ovvero, nella versione C il programma non riconosce le espressioni calcolate e poi assegnate a variabili, come decisioni. Si vede infatti che vengono individuate solamente 5 decisioni che sono precisamente quelle presenti nel costrutto if. Per concludere quindi Reactis non tiene conto delle decisioni presenti fuori dai costrutti if.
Una ulteriore curiosità a supporto dell'affermazione appena fatta è che si può notare che riconoscendo solamente 5 condizioni, le quali ovviamente non sono accoppiate in alcun modo, si ottiene il 100% della copertura MC/DC con T_C che ha dimensione pari a 6; che guarda caso è esattamente il famoso parametro n + 1, dove n in questo caso vale 5, che sono il numero di casi di test minimi necessari per ottenere la copertura MC/DC del 100%.

5.4.2 Confronto sulla facilità di utilizzo dei due tools

Parlando della semplicità di utilizzo dei due strumenti utilizzati per condurre l'esperimento, si può dire che entrambi gli strumenti non sono particolarmente difficili da utilizzare. Entrambi gli strumenti hanno un manuale per l'utente molto dettagliato che permette di utilizzare senza problemi il programma, e di essere in grado anche di capire ciò che sta facendo. La principale differenza tra i due programmi è il modo in cui viene presentata l'interfaccia utente, infatti Reactis mette a disposizione un'interfaccia utente grafica mentre Testwell mette a disposizione un'interfaccia a linea di comando. Entrambi i programmi sono anche forniti con un tutorial, che viene illustrato anche nel manuale, che permette di capire con più facilità i comandi e le funzionalità dei tool. Ricordiamo anche che il sottoprogramma sul quale abbiamo effettuato gli esperimenti è stato estratto dall'esempio fornito con Reactis. Concludendo quindi, per l'esperienza affrontata con l'esperimento, se uno ha una minima esperienza con il terminale, sono dell'avviso che Testwell sia leggermente più intuitivo e più semplice da utilizzare, sottolineando comunque il fatto che l'utilizzo di entrambi i programmi rimane molto semplice ed intuitivo.

5.5 Conclusioni sull'Esperimento

Stando ai risultati commentati nella sezione 5.4.1, si può mettere in risalto un fattore importante che dimostra la crucialità della qualifica di uno strumento secondo un determinato standard. Nel nostro caso infatti, tra i due strumenti utilizzati, solamente uno è qualificato secondo lo standard DO-178, ovvero Testwell CTC++. Ed infatti vediamo dai risultati che Reactis non si comporta come stabilito dallo standard in tutti i casi (cfr. tabelle 5.5 e 5.8). Questo dimostra quindi la necessità e l'importanza che uno strumento utilizzato come

supporto durante il ciclo di vita dello sviluppo di software safety critical debba essere qualificato rispettando quindi determinati requisiti.

Capitolo 6

Considerazioni Finali e possibili sviluppi futuri

Ricapitolando quindi in questo lavoro di tesi **abbiamo visto**:

- Cosa è ed a cosa serve lo standard DO-178.
- Cosa è la copertura strutturale e quali sono i principali criteri di misura della copertura.
- Cosa è MC/DC, come si definisce, quale è lo svantaggio principale, quali sono le possibili soluzioni al problema delle condizioni strettamente accoppiate, la probabilità di rilevazione di errori con la copertura MC/DC ed il confronto con MCC.
- È stato condotto un piccolo ma utile esperimento su due strumenti utilizzati per analizzare la copertura del codice, utilizzando un esempio comune e vedendo come gli strumenti si comportano nell'analisi delle decisioni, facendo infine un confronto in base a quanto espresso in [4].

Concludendo quindi si possono **proporre** ulteriori **ampliamenti del lavoro condotto** che seguono:

- Condurre un esperimento su del codice effettivamente in uso, quindi facente parte di un caso reale e di un progetto significativamente più grande.
- Effettuare una comparazione che comprenda un maggior numero di software, comparandoli su un maggior numero di caratteristiche (ricordiamo infatti che il poco tempo a disposizione non ha permesso di richiedere le prove degli strumenti in anticipo, e soprattutto non ha permesso di aspettare le risposte da parte delle aziende per molto tempo).
- Proprorre una collaborazione a un'azienda che sviluppi uno dei software citato in precedenza ed effettuare quindi un esperimento che occupi un tempo maggiore senza avere limitazioni sul periodo di prova del software.

Bibliografia

[1] Steven P. Miller John Joseph Chilenski. *Applicability of modified condition decision coverage to software testing*. Rapp. tecn. Software Engineering Journal, 1994.

[2] Nancy Leveson Arnaud Dupuy. ≪An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software≫. In: *DASC (Digital Aviation Systems Conference)*. 2000.

[3] John Joseph Chilenski. *AN INVESTIGATION OF THREE FORMS OF THE MODIFIED CONDITION DECISION COVERAGE (MCDC) CRITERION*. Rapp. tecn. FAA, 2001.

[4] Kelly J. Hayhurst et al. *A Practical Tutorial On Modified Condition/Decision Coverage*. Technical Memorandum (TM). NASA, 2001.

[5] Stacy Nelson. *Certification Processes For Safety-Critical And Mission-Critical Aerospace Software*. Rapp. tecn. NASA, 2003.

[6] P. V. Bhansali. *The MCDC Paradox*. Rapp. tecn. ACM SIGSOFT Software Engineering Notes, 2007.

[7] Raimund Kirner Susanne Kandl. *Error Detection Rate of MC/DC for a Case Study from the Automotive Domain*. Rapp. tecn. International Workshop on Software Technologies for Embedded e Ubiquitous Systems (SEUS), 2010.

[8] *What is safety-certifiable avionics hardware that meets Design Assurance Levels (DAL)?* 2016. URL: https://www.militaryaerospace.com/computers/article/16714656/what-is-safetycertifiable-avionics-hardware-that-meets-design-assurance-levels-dal.

[9] Andrea D'Urso. *32 lanci in un anno per SpaceX. Tutti i numeri di un record già infranto*. 2022. URL: https://www.astrospace.it/2022/07/23/32-lanci-in-un-anno-per-spacex-tutti-i-numeri-di-un-record-gia-infranto/.

[10] Stefano Piccin. *Nel 2021 sono stati lanciati più razzi orbitali che in qualsiasi anno precedente*. 2022. URL: https://www.astrospace.it/2022/01/04/nel-2021-sono-stati-lanciati-piu-razzi-orbitali-che-in-qualsiasi-anno-precedente/#:~:text=Per%20la%20prima%20volta%20il,135%20successi%20e%2011%20fallimenti)..

[11] P. V. Bhansali. *A Systematic Approach to Identifying a Safe Subset for Safety-Critical Software*.

[12] Vance Hilderman. *Avionics certification explained*. URL: https://afuzion.com/do-178-introduction/.

[13] *Visualising the global air travel industry*. URL: https://www.aljazeera.com/economy/2021/12/9/visualising-the-global-air-travel-industry-interactive#:~:text=In%202019%2C%20some%204.5%20billion,every%20day%2C%20according%20to%20FlightRadar24..

[14] Roland Wolfig. *Parameters for Efficient Software Certification*.