LABORATORIO PUBBLICO-PRIVATO COSMIC

SVEVIA

AUTORIZZATO E FINANZIATO DAL MINISTERO DELL'UNIVERSITÀ E DELLA RICERCA

APPROVATO CON DECRETO DEL DIRETTORE GENERALE DEL MIUR PROT.660/RIC. DEL 08/10/2012

SU DOMANDA PON02_00669 – PROGETTO PON02_00485_3487758.

D1.1 STUDIO ED ANALISI DELLE TECNICHE E DEGLI STRUMENTI DI V&V

AUTORI E APPROVAZIONI

Nome / Cognome	Partner
M. FICCO	CINI
R. NATELLA	CINI
R. PIETRANTUONO	CINI
F. SCIPPACERCOLA	DIETI
M. SORRENTINO	SESM
G.CARROZZA	SESM

Approvato da	Partner
A. JARRE	SELEX

INDICE

1 INTRODUZIONE

I sistemi software mission-critical sono sistemi il cui fallimento o malfunzionamento può condurre a perdite catastrofiche in termini economici, di danni per l'ambiente o anche di vite umane. I sistemi mission-critical sono adottati in un numero crescente di aree, che spaziano dai sistemi bancari a quelli di e-commerce, dagli impianti aeronautici a quelli ferroviari, dagli scenari automotive fino a quelli di health-care. Per cui, la Verifica e Validazione (V&V) dei sistemi mission-critical rappresenta un fattore chiave per garantire la qualità e la affidabilità.

1.1 SCOPO DEL DOCUMENTO

Questo documento ha lo scopo di sintetizzare la vasta letteratura inerente alle politiche e alle tecniche di V&V per sistemi software con stringenti requisiti di affidabilità. Inoltre, questo documento considera l'uso degli strumenti di V&V nel contesto della realtà aziendale di SELEX-ES.

1.2 STRUTTURA DEL DOCUMENTO

La Sezione 2 discute gli approcci quantitativi che sono stati proposti per gestire il processo di V&V, mostrando i principali approcci analitici ed empirici per focalizzare e dimensionare le attività di V&V. La Sezione 3 illustra invece le varie tecniche esistenti per la V&V, includendo tecniche di revisione del codice, l'analisi del codice, il testing, e il debugging. Infine, la Sezione 3 analizza l'adozione e le caratteristiche delle tecniche e dei tool utilizzati dai team di sviluppo e di verifica all'interno dell'azienda SELEX-ES, con lo scopo di evidenziare le principali necessità e le opportunità aperte per il progetto SVEVIA.

2 POLITICHE DI GESTIONE DELLE RISORSE DI V&V

La corretta pianificazione delle attività e delle risorse di V&V rappresenta un elemento chiave per ottenere un efficiente ed efficace processo di V&V. A questo scopo, esistono diversi approcci, illustrati in questa sezione, che permettono di focalizzare e adattare la durata delle attività di V&V, con lo scopo di ottenere dei prefissati obiettivi di affidabilità. Nel seguito, si illustra inizialmente il problema dell'allocazione delle risorse, poi gli approcci modellistici di base per analizzare la reliability dei sistemi, ed infine i principali modelli per relazionare le attività di V&V con l'affidabilità dei sistemi.

2.1 IL PROBLEMA DELL'ALLOCAZIONE DELLE RISORSE DI V&V

Nei sistemi mission-critical, un alto livello di affidabilità rappresenta un requisito cruciale da soddisfare. Tuttavia, le attività richieste per raggiungere tali livelli di affidabilità richiedono ingenti costi. In particolare, è stato provato che la maggior parte dei costi di sviluppo sono dovuti al processo di V&V e di manutenzione, in particolare per i sistemi di grandi dimensioni.

In più domini applicativi critici, il livello di affidabilità da raggiungere (assieme ad altri obiettivi di qualità) è imposto da standard specifici del dominio, come quello CENELEC, nel campo applicativo ferroviario, IEC 61508, per sistemi elettrici/elettronici, oppure DO-178B per i sistemi avionici. Al fine di raggiungere i requisiti di affidabilità, questi standard suggeriscono sia potenziali tecniche che possono essere adottate, sia processi di sviluppo, ma trascurano le problematiche relativi ai costi e all'efficienza. Le linee guida che loro forniscono sono abbastanza generali, come è evidenziato da [1], poiché il loro scopo non è quello di definire quali tecniche una azienda debba usare o quale sia il loro impatto sui costi di produzione. Dunque, esiste un divario tra ciò che loro suggeriscono e le strategie che sono concretamente adoperate.

Questo pone serie difficoltà alle aziende, che da una parte sono vincolate a raggiungere predefiniti livelli di affidabilità, mentre dall'altra devono consegnare sistemi in tempi e costi contenuti.

Al fine di costruire piani efficaci, le risorse per la verifica dovrebbero essere allocate in maniera conveniente alle varie parti del sistema, per esempio dedicando la maggior parte delle risorse alle parti che possono mettere a repentaglio gli obiettivi di qualità. Questo richiede che gli ingegneri identifichino correttamente i componenti e i sottosistemi più critici nell'architettura software dal punto di vista dell'affidabilità. Tuttavia non è facile identificare le parti di un sistema complesso che sono le maggiori responsabili dell'inaffidabilità, perciò l'allocazione delle risorse al testing è perlopiù basata sul giudizio degli ingegneri. Gli ingegneri tendono a volte a giudicare come "maggiormente critici" quei componenti che sono i più complessi, oppure quelli che sono i più usati e assegnano ad essi la maggior parte delle risorse di testing, facendo scelte di allocazione errate e dunque sprecando risorse preziose.

Inoltre, anche supponendo di distribuire le risorse in maniera corretta, è comunque critico scegliere le tecniche da adottare in un piano di verifica: nel contesto di sistemi altamente affidabili, l'impatto dei differenti tipi di tecniche di verifica sull'affidabilità finale del sistema dovrebbe essere tenuta in conto e usata come guida per la realizzazione del piano. È da notare che identificare più malfunzionamenti non implica migliorare l'affidabilità: incrementare l'affidabilità significa rimuovere quei malfunzionamenti che si

manifestano più frequentemente a tempo di esecuzione. Una selezione delle tecniche di verifica orientata all'affidabilità dovrebbe considerare l'influenza che queste hanno sul sistema sottoposto a verifica: applicare una tecnica di verifica a due sistemi software distinti conduce, in generale, a risultati differenti. Pertanto gli ingegneri devono conoscere come scegliere una giusta combinazione di tecniche di verifica che meglio si adatti al software in sviluppo.

2.2 MODELLI ANALITICI DI RELIABILITY

La reliability è una misura della continuità dell'erogare un servizio in maniera corretta. È uno degli attributi del concetto di affidabilità [2] che è particolarmente rilevante per i sistemi critici, dove non possono essere tollerate interruzioni di servizio. Differentemente dalla availability, che si concentra su fallimenti in dati istanti temporali, la reliability enfatizza l'occorrenza di eventi non desiderabili in uno specificato intervallo temporale, spesso chiamato Mission Time. Un sistema scarsamente available potrebbe essere altamente reliable in un determinato mission time, a seconda di quando i fallimenti si verificano. Ad esempio, un sistema potrebbe fallire molte volte in un anno, ma mai nell'intervallo di tempo che è cruciale per il servizio che fornisce il sistema, cioè fallisce sempre in istanti in cui c'è tolleranza ai malfunzionamenti.

La reliability può essere valutata usando differenti approcci, generalmente classificati in due categorie: model-based e measurements-based. Gli approcci model-based sono largamente utilizzati per la valutazione dell'affidabilità di sistemi hardware/software complessi. Essi sono basati sulla costruzione di un modello che è una conveniente astrazione di un sistema, con un adeguato livello di dettaglio per rappresentare gli aspetti di interesse per la valutazione. Il grado di accuratezza di un modello dipende dall'abilità del formalismo associato nel catturare le caratteristiche del sistema.

I modelli permettono di analizzare in maniera appropriata l'architettura di un sistema, di valutare differenti configurazioni, di scovare i colli di bottiglia delle performance/reliability, di fare predizioni e di comparare scelte progettuali senza doverle prima implementare. Queste possono essere oggetto di analisi o simulazione. I modelli analitici sono classificati in combinatoriali e state-based. I modelli nella prima categoria rappresentano la struttura di un sistema in termini di connessioni logiche di componenti funzionanti (falliti) in modo da ottenere il funzionamento (fallimento) del sistema. I modelli a spazio di stati rappresentano il comportamento di un sistema in termini di stati raggiungibili e possibili transizioni di stato.

I modelli possono essere risolti in forma chiusa in maniera analitica oppure numericamente a seconda della loro complessità computazionale. Quando una soluzione analitica non è disponibile (o computazionalmente proibitiva), la simulazione rappresenta una alternativa percorribile.

2.2.1 MODELLI COMBINATORIALI

I modelli combinatoriali hanno una notazione semplice e intuitiva, sono facili da disegnare e manipolare e possono essere efficientemente analizzati grazie a tecniche combinatoriali. Il sistema è tipicamente suddiviso in un insieme di moduli non intersecanti, ognuno associato a una relativa probabilità di funzionamento, P_i, o una probabilità espressa in funzione del tempo, ad esempio $R_i(t)$. L'obiettivo è derivare il valore complessivo P_{sys} (o la funzione $R_{sys}(t)$) che rappresenta la probabilità che il sistema

sopravviva (fino l'istante t). Questi modelli tipicamente enumerano tutti gli stati del sistema utilizzando tecniche di calcolo combinatoriale per semplificare il processo.

Nonostante i loro vantaggi (analisi semplice e intuitiva), i modelli combinatoriali soffrono di un limitato potere espressivo, dovuto principalmente alle assunzioni in cui si pongono: (i) i fallimenti dei moduli sono indipendenti (cioè, indipendenza statistica degli eventi), (ii) dopo che un modulo è fallito, si assume sempre che fornisca un risultato non valido, (iii) dopo che il sistema è entrato nello stato di fallimento, gli altri fallimenti non possono ripristinare il funzionamento del sistema.

Esempi di modelli combinatoriali sono i Reliability Block Diagram (RBD) [4] [5] e i Fault Tree (FT) [6].

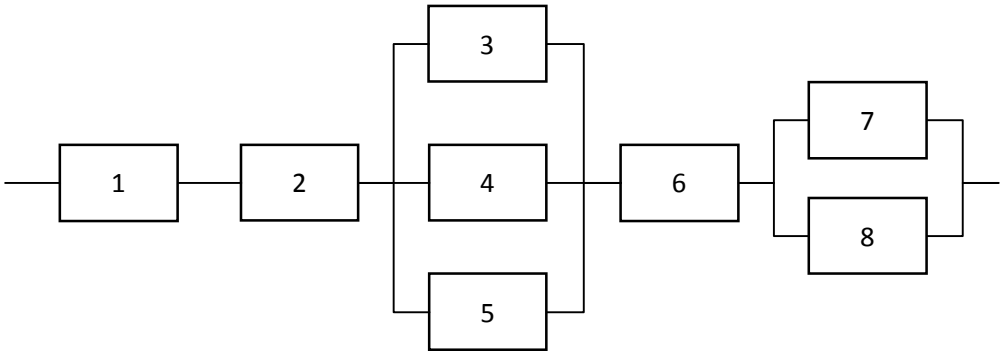


Figura 2-1: Reliability Block Diagram (RBD). Il fallimento del sistema è legato alla eq. booleana 1+2+(345)+6+(78)

I RBD (**Figura 2-1**) usano blocchi logici per collegare uno stato di un sistema complesso agli stati dei suoi componenti. Un blocco, rappresentante un componente, può essere visto come un interruttore che è chiuso quando il blocco funziona mentre è aperto quando il blocco è guasto. Il sistema è funzionante se almeno un percorso attraversato da tutti interruttori chiusi è presente dal punto di input a quello di output del diagramma. I blocchi possono essere connessi in serie (per rappresentare componenti che sono tutti richiesti per il funzionamento del sistema), in parallelo (per rappresentare blocchi di cui almeno un componente è richiesto), in una struttura k-of-n (quando almeno k su n componenti sono richiesti). La struttura complessiva può essere composta da tutti questi tipi di connessioni e conduce sia a RBD in serie-parallelo (che possono essere risolti da semplici riduzioni serie-parallelo) oppure a RBD non in serie-parallelo (che possono essere risolti per enumerazione, fattorizzazione, conditioning oppure tramite binary decision diagram (BDD)).

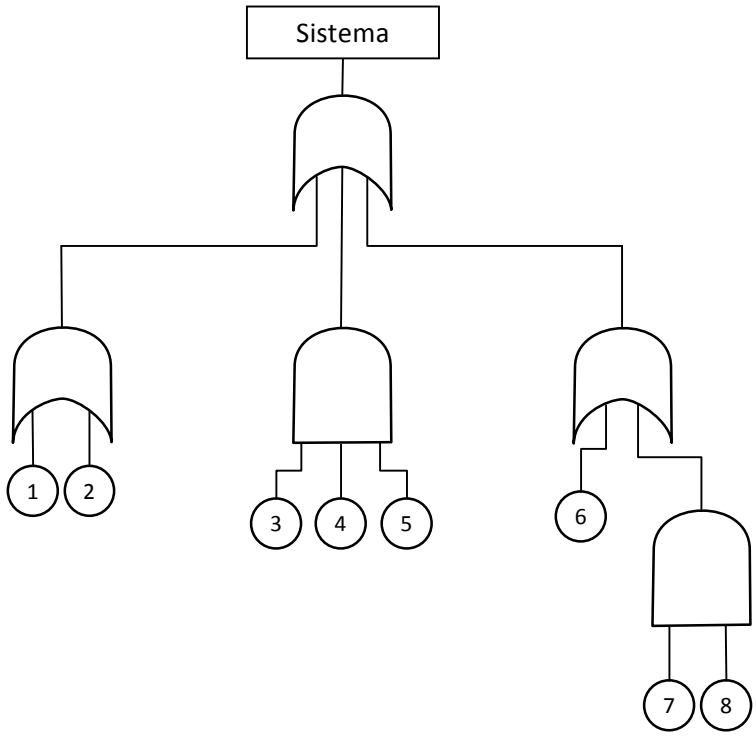


Figura 2-2: Fault-Tree (FT). Il fallimento del sistema è legato alla equazione booleana (1+2)+(345)+(6+78)

I fault tree (Figura 2-2) legano combinazioni di eventi elementari al fallimento del sistema. Essi sono una rappresentazione grafica in cui i componenti sono interconnessi tra loro attraverso porte logiche in una struttura ad albero. Il fallimento di un componente o di un sottosistema, cioè il verificarsi di un evento elementare, causa il variarsi del corrispondente input della porta logica nel valore VERO; quando l'output della porta logica in cima assume valore VERO, il sistema è considerato fallito. Nelle versioni di base, i fault tree utilizzano porte logiche AND per collegare componenti in parallelo, OR per collegare componenti in serie, e $(n-k+1)$ di n porte logiche per i componenti k-of-n. Estensioni ai fault-tree includono altre porte logiche, come la porta NOT, EXOR, Priority AND, cold spare gate, functional dependency gate and sequence enforcing gate. La facilità dei fault tree e la loro facile rappresentazione grafica sono la ragione principale del loro successo che è confermato dal loro uso estensivo per la modellazione di sistemi complessi reali.

Per migliorare il loro potere espressivo, più estensioni al formalismo dei fault-tree sono state proposte in letteratura, come i Dynamic Fault Tree (DFT) [7], Parametric Fault Tree (PFT) [8], e i Repairable Fault Tree (RFT) [9].

2.2.2 Modelli a spazi di stati

Quando l'accuratezza di un modello combinatoriale non è sufficiente per catturare le caratteristiche del sistema da modellare, si possono prendere in considerazione i modelli a spazio di stati.

I modelli in questa categoria possiedono un più ampio potere espressivo e maggiore flessibilità rispetto i modelli combinatoriali, ma l'analisi dello spazio degli stati potrebbe risultare computazionalmente impraticabile. Questo dipende dal numero degli stati nel modello, poiché lo spazio degli stati cresce esponenzialmente col numero dei componenti nel sistema. Il problema dell'esplosione degli stati ha scatenato parecchi studi e ha permesso di conseguire risultati significativi basati su due approcci principali: *largeness avoidance* e *largeness tolerance*. Le tecniche di largeness avoidance cercano di prevenire la generazione di modelli con un elevato numero di stati, mentre le tecniche di largeness tolerance forniscono il supporto pratico di modellazione per facilitare la generazione e la soluzione di modelli con un elevato numero di stati.

Il formalismo base per i modelli a spazio di stati sono i modelli markoviani. Un processo di Markov è un processo stocastico il cui comportamento dinamico è tale che le distribuzioni di probabilità per le sue future evoluzioni dipendono solamente dallo stato presente e non da come il processo sia giunto in quello stato [3]. Quando lo spazio degli stati è discreto, cioè l'insieme di tutti i possibili valori che possono essere assunti dalle variabili aleatorie del processo è discreto, il processo di Markov è conosciuto come una catena di Markov. Le catene di Markov sono un blocco fondamentale per l'analisi a spazio di stati.

Essi sono stati adoperati per modellare un ampio numero di sistemi, che spaziano dai sistemi di rete, ai protocolli di componenti hardware, ai sistemi software/hardware, alle applicazioni software, ai sistemi cluster complessi, e sono stati usati per l'analisi di qualsiasi tipo di attributo in relazione all'affidabilità e alle performance (reliability, availability, performability, survivability).

I modelli markoviani possono essere classificati in: catene di Markov a tempo discreto (DTMC), quando il modello adotta un indice discreto t, di solito rappresentante il tempo; catene di Markov a tempo continuo (CTMC), quando t è un indice continuo; e modelli di Markov con reward, quando la catena di Markov include inoltre un punteggio di reward (o peso) collegato allo stato in modo da fornire misurazioni aggiuntive, per esempio il numero atteso di reward accumulate in un intervallo di interesse.

Il modello è tipicamente rappresentato graficamente da un grafo a stati-transizioni (Figura 2-3), che mostra gli stati del sistema (i nodi) e le transizioni tra di esse (gli archi) etichettati dal valore di probabilità delle transizioni.

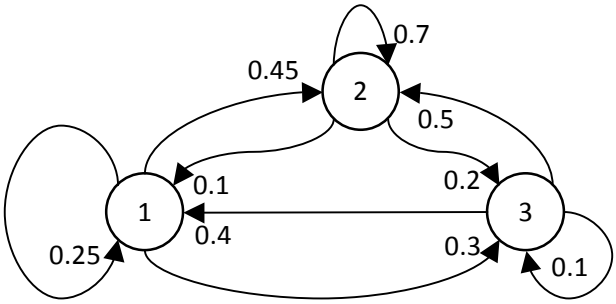


Figura 2-3: rappresentazione grafica di una catena di Markov a tre stati.

Quando la proprietà markoviana di memoryless non vale, cioè quando essa non descrive in maniera accurata il sistema modellato, sono impiegati modelli non-markoviani (come ad esempio i processi semi-Markov o i modelli generativi di Markov) oppure modelli di Markov non omogenei, laddove altre distribuzioni sono permesse. L'accuratezza che questi modelli aggiungono è pagata in termini di complessità di gestione, parametrizzazione e soluzione.

L'analisi markoviana consiste di alcuni passi elementari da compiere: astrazione del sistema fisico, costruzione del modello di Markov, impostazione di una soluzione di equazioni differenziali ordinarie (per la soluzione del transitorio) o di equazioni lineari (per la soluzione dello stato a regime). Quando il numero di stati è elevato, l'analisi markoviana diventa tediosa, complessa e soggetta ad errori. Con l'incremento delle dimensioni dei sistemi, tale problema ha condotto, alla fine degli anni del '80, all'introduzione di nuovi formalismi e strumenti. Uno di questi ha avuto particolare successo, a causa della propria abilità di rappresentare in maniera concisa un sistema complesso in maniera intuitiva: le Stochastic Petri Nets (SPN). Una delle caratteristiche fondamentali delle SPN è che c'è una corrispondenza diretta tra loro e le CTMC, che permettono ai progettisti di modellare i propri sistemi tramite le più intuitive SPN e di tradurle automaticamente in CTMC da essere risolte (tramite strumenti come SPNP, DSPNExpress, GreatSPN, e SHARPE). Similmente le stochastic reward nets (SRN), che sono una estensione delle SPN con l'aggiunta del reward, possono essere tradotti in modelli di Markov con reward.

Motivati dalla loro potenza espressiva (la loro rappresentazione grafica è particolarmente adeguata a modellare architetture parallele, programmi concorrenti, problemi di sincronizzazione e sistemi multiprocessore) e dalla loro capacità di essere risolti tramite modelli markoviani, i ricercatori hanno sviluppato ulteriori varianti delle reti di Petri stocastiche, appropriati per particolari esigenze applicative o per specifici metodi risolutivi, come le Generalized SPN, le Stochastic Activity Networks (SAN) e reti di Petri colorate (CPN).

2.2.3 MODELLI IBRIDI

I modelli non a spazio di stati (per esempio, i RBD e i FT) sono indubbiamente efficienti per la specifica e per l'analisi, ma l'assunzione di indipendenza su cui si basano può essere troppo restrittiva per parecchie situazioni pratiche. D'altra parte, i modelli markoviani forniscono l'abilità di modellare sistemi che vìolino questa assunzione, al costo dell'esplosione dello spazio degli stati. Per far fronte a questo problema un numero di approcci di modellazione è stato proposto per evitare la generazione di numerosi stati (per esempio, il già citato approccio di largeness avoidance). Questi sono ottenuti tramite i modelli ibridi. L'idea fondamentale è comporre/decomporre gerarchicamente il sistema e costruire i modelli in maniera analoga: i metodi a spazio di stati sono usati per quelle parti del sistema che richiedono di modellare le dipendenze, laddove metodi combinatoriali sono usati per le parti che possono essere considerate indipendenti.

Parecchi sono i lavori di ricerca pubblicati che tentano di combinare i vantaggi dei metodi di analisi combinatoriale con i vantaggi dei modelli basati sugli spazi di stati: tipicamente, in questi lavori, pochi componenti sono isolati in sottosistemi/componenti, e trattati con i metodi a spazi di stati, e poi sono combinati in strutture come FT, sfruttando le tecniche di analisi combinatoriale a livello di sistema

complessivo. In particolare, tali studi hanno anche incoraggiato proposte di estensioni del formalismo originale del FT, in modo da esprimere dipendenze in un formalismo simile a quello dei FT.

Esempi di lavori che adottano modelli ibridi sono [10][11][12][13][14][15][16]. Per esempio, gli autori in [12] hanno modellato una configurazione di una applicazione server SIP su WebSpere suddividendo il sistema complessivo in sottomodelli interagenti catturando il loro comportamento nel fallimento e nel ripristino. Una analisi dell'affidabilità del servizio che adotta un modello gerarchico è presentata in [11]; la modellazione gerarchica si fa corrispondere all'architettura fisica e logica di un sistema grid-service e fa uso di modelli di Markov, della teoria delle code e della teoria dei grafi per modellare e valutare l'affidabilità del grid-service.

2.2.4 Approcci basati su misurazioni

I modelli sono estensivamente usati per la valutazione degli attributi di affidabilità, specialmente per l'analisi della reliability. Comunque, essi possono non essere accurati sufficientemente quando i valori dei parametri di ingresso non sono rappresentativi del comportamento reale del sistema.

L'approccio basato sulle misurazioni può consentire risultati più accurati: esso è basato su dati reali operazionali (dal sistema o un suo prototipo) e sull'uso di tecniche di inferenza statistica. È una opzione attraente per valutare un sistema esistente o un prototipo e costituisce una strada efficace per ottenere una caratterizzazione dettagliata del comportamento del sistema in presenza di fault. Comunque, poiché sono necessari dati reali, non è sempre possibile applicare questo approccio, perché i dati possono non essere disponibili. Inoltre, affidarsi solamente all'approccio basato sulle misure non conduce a una comprensione sulle complesse dipendenze tra i componenti e non permette l'analisi di sistema da un più generale punto di vista. È spesso più conveniente fare misure sul livello di singoli componenti/sottosistemi piuttosto che sul sistema in generale [17], e quindi combinarli in un modello del sistema.

Una panoramica degli approcci sperimentali per la valutazione della dependability è in [18]. Nonostante la maggior parte degli articoli utilizza sia l'approccio basato su modelli o sulle misurazioni, alcuni lavori utilizzano un approccio combinato [19][20]. Un sistema di monitoring online che combina entrambi gli approcci per una valutazione della disponibilità di un sistema è in [21][22]. Modelli che siano parametrizzati da dati sperimentali sono inoltre usati in [23] per l'analisi del software aging, in cui le catene di Markov sono usate per modellare sia gli stati del sistema che gli stati del workload, e le probabilità di transizione e le stime dei tempi di soggiorno sono state ottenute usando dati sperimentali reali.

2.3 Modelli per la allocazione delle risorse

Gli approcci per allocare le risorse ai moduli/componenti/sottosistemi di un sistema software hanno lo scopo di distribuire le risorse disponibili per la verifica in una maniera efficace in termini economici. Il modo principale per affrontare questo problema è usare un modello: tipicamente l'obiettivo è quello di ottenere una qualche forma di predizione (per esempio contenuto di fault) dalle caratteristiche di un sistema che sono quindi usate per assegnare in maniera opportuna le risorse. Nota che le risorse che identificano lo sforzo di testing possono essere intese come mesi/uomo, numero di casi di test, tempo di CPU allocato per

il testing, e altre metriche. Come descritto in seguito, le funzioni di Testing Effort (TEF) possono essere usate per descrivere la relazione tra sforzo-di-testing e tempo-di-testing, e possono essere adeguatamente integrate in modelli.

Il compito di identificare le parti di un sistema che dovrebbero ricevere maggiore o minore attenzione nella fase di verifica dipendono principalmente da quali obiettivi specifici la verifica deve servire. Tipicamente la verifica ha l'obiettivo di identificare quanti più fault è possibile nel software. Per cui i criteri per distribuire gli sforzi dovrebbero riflettere questo bisogno: allocare maggiori risorse sui moduli-componenti-sottosistemi software che si attende contengano più fault. Verso tale obiettivo, molta ricerca è stata prodotta nel passato riguardo i modelli di fault-proneness, ovvero quei modelli abili a stimare il contenuto di fault nei moduli software basandosi su numerose metriche del software.

Meno comunemente, l'obiettivo della verifica è migliorare la reliability del sistema. Nota che identificare più fault non implica migliorare la reliability. Il miglioramento della reliability richiede la rimozione di quei fault che accadono a tempo operazionale in maniera più frequente. Una sessione di testing può pertanto rimuovere meno fault rispetto ad un'altra ma può consegnare un sistema con una maggiore reliability.

In questo caso, i criteri per allocare gli sforzi non cercano i moduli con un più alto contenuto di fault. Piuttosto quei moduli che più impattano sulla reliability che il sistema deve raggiungere, cioè quelli che sono più critici secondo la prospettiva dell'affidabilità. Questo non è banale: l'area di ricerca più vicina a questi bisogni è quella che affronta il ben noto problema della *Reliability Allocation*.

Approcci qualitativi per identificare i componenti critici di una architettura, basati sul giudizio degli ingegneri, non forniscono una informazione quantitativa sufficiente per allocare adeguatamente le risorse di testing e quantificare la reliability del sistema finale. Inoltre, questi possono anche condurre a una valutazione erronea (per esempio, un componente meno affidabile e raramente usato può impattare sulla reliability totale meno rispetto ad un componente più affidabile ma usato di frequente). L'allocazione basata su un ragionamento quantitativo è essenziale per rispondere a domanda come:

- Quanto un componente costituisce un rischio per il sistema?
- Componenti con lo stesso livello di rischio impattano ugualmente sulla reliability del sistema?
- Qual è l'impatto di un cambiamento della reliability di un componente sulla reliability del sistema?

I più importanti approcci non quantitativi non possono rispondere al seguente quesito:

- Qual è l'affidabilità che ogni componente deve raggiungere al fine di assicurare un livello minimo di reliability del sistema, e a quale costo?

La soluzione principe per rispondere a questi problemi è utilizzare dei modelli, che consentano di effettuare analisi e/o simulazioni di quanto i vari profili di allocazione delle risorse impattano sulla reliability del sistema finale: i Software Reliability Growth Model consentono di predire le risorse del testing che servono per raggiungere un desiderato livello di difettosità residua; i modelli di Fault-proneness, stimano invece il numero di fault che i componenti all'interno del sistema posseggono. Questi modelli sono da

utilizzare a seconda degli obiettivi del processo di verifica e sono oggetto di approfondimento nei successivi paragrafi.

2.3.1 SOFTWARE RELIABILITY GROWTH MODEL

Mezzi utili per progettare la suddivisione delle risorse nel testing sono i Software Reliability Growth Model (SRGM). Un SRGM è un modello che scrive quanto cresce l'affidabilità di un software che è soggetto a miglioramento durante un testing tramite identificazione e rimozione di fault.

Indipendentemente dal formalismo di modellazione che può essere adottato, gli approcci di analisi della reliability possono essere distinti rispetto al loro obiettivo. Una prima classe di modelli, a cui ci riferiamo come *modelli black-box*, hanno l'obiettivo di valutare come la reliability migliori durante il testing e vari dopo la consegna; una seconda classe di modelli si concentra perlopiù sul comprendere le relazioni tra i componenti del sistema e la loro influenza sulla reliability del sistema. Ci si riferisce ad essi come modelli basati sull'architettura (*architecture-based models*).

2.3.1.1 BLACK-BOX RELIABILITY MODELS

I modelli classici di reliability (black-box) sono tipicamente calibrati tramite i failure data raccolti durante il testing, più specificamente facendo il fitting dei tempi di inter-fallimento, dunque osservando la variazione dell'intensità di fallimento (il numero di fallimenti sull'unità di tempo) rispetto al tempo di testing. La forma della curva dell'intensità di fallimento distingue l'ampia varietà di SRGM presenti in letteratura (per esempio [24][25][48][49][50][51]).

Parecchi SRGM di questo tipo sono stati proposti in letteratura. Tipicamente i modelli usati adottano un processo di Poisson non-omogeneo (NHPP) per modellare la crescita della reliability di un software durante la fase di testing. I modelli NHPP sono distinti dalla forma della propria funzione valore medio $(t) = \mathrm{E}[N(t)]$, dove $N(t)$ è il numero di occorrenze di fallimento nell'intervallo di tempo $(0, t]$, oppure equivalente al suo integrale $\int_0^t m(x)\mathrm{d}x$, chiamato intensità di fallimento. Uno dei modelli di maggior successo è stato proposto nel 1979 da Goel e Okumoto (GO) [24]. Esso assume una funzione esponenziale di intensità dei fallimento $\lambda(t) = b\big(a - m(t)\big) = abe^{-bt}$, dove a è il numero atteso di fault da essere identificati se il testing è eseguito in maniera indefinita, b si interpreta come il tasso di occorrenza di fallimenti per fault (che, nel contesto, è chiamato hazard rate $h(t)$). Consentendo $h(t)$ ad non essere costante, altri modelli sono stati sviluppati. Una versione generalizzata del modello GO si deriva utilizzando la distribuzione di Weibull [25], $h(t) = bct^{c-1}$. Gokhale e Trivedi [49] proposto una distribuzione log-logistic per $h(t)$ col fine di catturare il comportamento crescente o decrescente del tasso di occorrenza di fallimento per fault.

Molti altri modelli sono stati proposti, che inoltre catturano l'attività di debugging o il comportamento infinite-failure [26] (cioè i modelli che assumono che un numero infinito di fault è identificato in un testing infinito), e vari strumenti sono stati sviluppati per eseguire il fitting e la parametrizzazione dei modelli più adeguati per un dato insieme di dati di fallimento (come SREPT, SMERFS, SoRel e CASRE).

I SRGM sono tipicamente usati per rispondere a domande come “per quanto tempo fare il test di un software”, oppure “quanti fault è probabile siano rimasti”; questo uso system-level è il più comune. Meno lavori usano il SRGM component-level, in particolare per ottimizzare la distribuzione delle risorse tra i componenti di un sistema soddisfacendo un obiettivo di affidabilità. Tra questi, Yamada et al. [52], si sono posti il problema dell'allocazione ottimale delle risorse nel testing di modulo assumendo gli stessi SRGM per tutti i moduli coinvolti e formulando due varianti del problema di ottimizzazione. In questo e in successivi articoli, i SRGM includono nella formulazione ciò che è noto come Testing Effort Function (TEF). La dimensione temporale su cui si valutata la crescita dell'affidabilità è espressa come tempo del calendario, tempo orario, tempo di esecuzione CPU, numero di test-run, o misure simili: ma, in generale, le risorse di testing non variano linearmente con il tempo e i TEF descrivono relazioni non lineari [53][54][55].

Lyu et al. [56][57] si pongono lo stesso problema, e propongono un modello di ottimizzazione con una funzione di costo basata sui ben noti modelli SRGM. Essi includono l'uso di un fattore di coverage per ogni componente, per tenere conto delle possibilità che un fallimento può essere tollerato. Altri lavori, di Lyu e Huang, formulano lo stesso problema con poche variazioni [58][59]. Il costo, assieme alla funzione di risorse del testing, è considerato anche in lavori successivi degli stessi autori [60][61]. Gli autori in [62] inoltre provano ad allocare i tempi ottimali di testing ai compoenenti, con i SRGM limitati a modelli iper-geometrici (S-shaped). Il lavoro in [63] considera inoltre implicitamente l'architettura del software, tenendo in conto dell'utilizzo di ogni componente con un fattore che sia assunto noto.

In ogni caso, l'uso dei SRGM implica un insieme di assunzioni sul processo di testing che sono facilmente violati nei processi di testing reali. Esempi sono: la riparazione perfetta o immediata, tempi di inter-fallimento dipendenti, report di difetti duplicati, nessun cambiamento del codice durante il testing, equiprobabilità in ogni unità temporale di trovare un fallimento (per esempio anche durante le feste e le ferie) [65][66]. D'altra parte gli autori in [65][66] affermano che si sa che i SRGM forniscono buoni risultati anche quando i dati parzialmente violano le assunzioni del modello.

2.3.1.2 ARCHITECTURE-BASED RELIABILITY MODELS

Tradizionalmente, gli approcci per analizzare la reliability del software sono black-box, cioè loro trattano il software applicativo come un intero monolitico modellando le proprie interazioni con l'ambiente esterno. L'approccio black-box ignora informazioni riguardo la struttura interna dell'applicazione e trascura le relazioni tra i componenti del sistema. Esso è basato su (i) la raccolta di dati riguardo il fallimento durante il testing, e (ii) sulla calibrazione del modello di reliability growth del software (SRGM) tramite tali dati. Questo modello è quindi usato per la predizione della fase operazionale (per predire la prossima occorrenza di fallimento basato sul trend osservato durante il testing) e/o nella fase di testing di successive release di sistema per determinare quando interrompersi col testing. Tuttavia, il problema principale di questi modelli rimane l'incapacità di considerare lo stato interno dei componenti e le loro interazioni.

Per superare queste limitazioni, sono stati proposti approcci architecture-based. Questo tipo di modelli ha guadagnato importanza sin dall'avvento dei sistemi component-based e object-oriented, quando il bisogno di considerare la struttura interna del software per caratterizzare correttamente al propria reliability è diventata importante. Ciò ha condotto a un interesse crescente nell'analisi della reliability e nelle

performance architecture-based [27][28][29][30]. I modelli architecture-based possono essere categorizzati come segue [31]:

- I modelli state-based usano un control flow graph (CFG) per rappresentare l'architettura software; loro assumono che il trasferimento del controllo tra i componenti ha una proprietà markoviana, e modellano l'architettura come una catena di Markov tempo-discreto (DTMC) o una catena di Markov tempo-continuo (CTMC) o un processo semi-Markov (SMP).
- I modelli basati su cammini (path-based) calcolano la reliability del sistema considerando tutti i possibili cammini di esecuzione di un programma.
- I modelli additivi, dove la reliability dei componenti è modellata tramite processi di Poisson non omogenei (NHPP) e l'intensità di fallimento del sistema è calcolata come la somma della intensità di fallimento dei singoli componenti.

I modelli state-based possono essere ulteriormente categorizzati in modelli compositi o gerarchici [32]. Nei primi, l'architettura del software e il comportamento di fallimento del software sono combinati nello stesso modello, mentre l'approccio gerarchico separatamente risolve il modello architetturale e quindi sovrappone il comportamento di fallimento dei componenti sulla soluzione. Nonostante i modelli gerarchici forniscono una approssimazione della soluzione del modello composito, essi sono più flessibili e computazionalmente trattabili. In un modello composito, invece, valutare differenti alternative architetturali o gli effetti del cambiamento del comportamento di un singolo componente è computazionalmente costoso. Inoltre, non come i modelli gerarchici, essi sono anche soggetti al problema della stiffness [33]. Per superare la differenza tra l'accuratezza tra i modelli compositi e gerarchici, Gokhale e Trivedi [34] hanno incluso gli effetti architetturali di secondo ordine nei modelli gerarchici.

2.3.2 MODELLI EMPIRICI DI FAULT-PRONENESS

I modelli di fault-proneness permettono di sfruttare dati storici col fine di identificare, tramite analisi empiriche, i moduli software più inclini (prone) a contenere fault. Questo consente di meglio allocare le risorse sui moduli software. Tali modelli hanno recentemente riguadagnato attenzione, a causa dell'ampia disponibilità di dati in grandi database di bug-tracking e nei sistemi di controllo delle versioni.

Per quanto riguarda i modelli di fault-proneness, numerosi studi nel passato hanno correlato metriche del software ai fault osservati in un dato numero di campioni (per esempio programmi software) [35]. Le metriche del codice sono state ampiamente usate nel passato sia per misurare la qualità del codice in accordo ad alcuni valori di riferimento, e sia come predittori dei difetti nei moduli software: tecniche statistiche sono solitamente adottate in questi articoli per costruire modelli di regressione che consentono di predire il contenuto di fault a partire dai valori delle metriche.

Le metriche possono essere divise in metriche method-level, class-level, file-level e component-level [67]. Gli esempi delle più comuni metriche method-level sono le metriche di McCabe (ad esempio, la complessità ciclomatica di McCabe, McCabe LoC) [68], e le metriche di Halstead (ad esempio, il volume, la lunghezza e la difficoltà) [69]. Le metriche class-level sono più recenti, e si riferiscono al paradigma orientato agli oggetti. Esempi sono la suite di metriche di Chidamber-Kemer (CK) [36] proposta nel 1994, la MOOD

(metriche per la progettazione object-oriented) [70], QMOOD (metriche di qualità per la progettazione object-oriented) [71], e le suite di metriche di L&K (Lorenz and Kidd) [72].

Comunque, la suite di metriche CK è più popolare di altre e sono perlopiù usate se le metriche class-level sono applicate. Esse sono: Coupling_Between_Objects (CBO), Depth_Of_Inheritance_Tree (DIT), Lack_Of_Cohesion_Of_Methods (LCOM), Num_Of_Children (NOC), Response_ For_Class (RFC), and Weighted_Method_Per_Class (WMC). WMC è il numero di metodi che sono collocati in ogni classe. DIT è la distanza del più lungo cammino da una classe alla radice nell'albero dell'ereditarietà. EFC è il numero di metodi che possono essere eseguiti per rispondere a un messaggio. NOC è il numero di classi che sono diretti discendenti per ogni classe. CBO è il numero di classi non legate per ereditarietà a con cui una classe è accoppiata. LCOM è in relazione con la proporzione dell'accesso degli attributi. Stando a più studi software sulla fault-prediction [73], CBO, WMC, e le RFC sono le metriche più significative per la predizione dei fault.

Le metriche per file sorgente sono inoltre sempre più usate per la fault prediction, come in [74][75]: alcune di queste metriche sono il numero di linee di codice per file, il numero di linee commentate di codice per file, il numero di cambiamenti per file.

Molta letteratura ha avuto l'obiettivo di trovare il miglior insieme di metriche che fosse capace di predire la fault proneness di moduli software. Altri lavori sono sull'identificare le relazioni tra vari tipi di metriche software e la defect proneness in un programma. Le prime ricerche miravano a definire metriche che fossero adeguate per misurare la complessita di un modulo software e, di contro, la propria probabilità di essere faulty (come le metriche di McCabe e di Halstead). Gli approcci di fault prediction sono quindi evoluti con l'adozione di algoritmi e tecniche di machine learning e di data mining, col fine di stabilire una relazione più accurata tra gli insiemi di metriche del software e i fault, tramite l'uso di classificatori e modelli di regressione.

In [36] e [37] le metriche Object-oriented sono state proposte come predittori della densità dei fault. Lo studio in [38] mostra una panoramica di otto studi empirici che dimostrano come le metriche OO sono significativamente correlate con i fault. Tale lavoro ha validato empiricamente tre metriche OO adeguate per predire la qualità del software in termini di fault-proneness: le CK, le MOOD, e le QMOOD.

Ulteriori studi che usano metriche sono [39][40] che investigano metriche di progetto capaci di predire i moduli più inclini ai fallimenti. In [41], gli autori hanno usato un insieme di 11 metriche e un approccio basato sugli alberi di regressione per predire i moduli più faulty. In [42] gli autori indagano metriche per predire la quantità di fault post-release in cinque progetti software Microsoft di grandi dimensioni. Essi hanno adottato tecniche statistiche note, come l'Analisi a Componenti Principali (PCA), per trasformare l'insieme originale di metriche in un insieme di variabili incorrelate con lo scopo di evitare problemi di multicollinearità (cioè una sovrabbondante stima della varianza dovuta a una forte correlazione tra i predittori). Gli autori suggeriscono anche una procedura per costruire potenti modelli di regressione.

Ci sono poi studi che usano ogni sorta di metrica, come gli autori di [67], che affermano che quelle più usate e affidabili sono le metriche method-level (essi stimano che più del 64% dei lavori che hanno

ispezionato adottano metriche method-level), seguite da le metriche class-level, tra cui le più comuni sono quella della complessità ciclomatica di McCabe, le linee di codice, le metriche di Halstead, e le metriche object-oriented CK.

Dunque, dal momento che numerosi studi passati garantiscono che tali metriche sono sufficienti per avere buoni risultati, un modo pratico da applicare è partire da esse e successivamente aggiungere altre metriche specifiche per la compagnia o per le caratteristiche del prodotto/processo per raffinare l'accuratezza della predizione. Ad esempio, il lavoro in [43] utilizza la regressione logistica per mettere in relazione le misure del software di fault-proneness con classi di prodotti software omogenei. Gli autori in [44] estendono poi questo studio riportando una analisi empirica di validità di modelli multivariati per predire fault-proneness software tra differenti applicazioni. Varie tecniche statistiche sono state analizzate da Khoshgoftaar, che ha applicato la analisi in discriminanti con Munson [45] e la regressione logistica con Allen, Halstead, Trio, and Flass [46].

In parecchi casi, le metriche comuni forniscono buoni risultati predittivi anche tra differenti prodotti. Comunque, è ancora abbastanza difficile affermare che un dato modello di regressione o un insieme di modelli di regressione è abbastanza generale da essere usato anche con prodotti molto differenti, come anche è discusso in [42]. D'altra parte, essi sono indubbiamente utili all'interno di uno sviluppo aziendale di una determinata classe di sistemi, e che in tale processo raccoglie dati riguardo i fault.

Alcuni articoli si concentrano infine su come trasferire modelli di predizione tra differenti progetti e aziende [47]. Gli svantaggi di tale approccio sono: i) il bisogno di avere una base di conoscenza estesa da cui la relazione empirica tra metriche e difetti può essere derivata; ii) la necessità di possedere il codice sorgente del prodotto, come per la maggior parte degli approcci, per estrarre le metriche; iii) la mancanza di una formulazione di uno schema esplicito di allocazione; questi metodi tipicamente non predicono il numero di difetti, ma la probabilità di un modulo di essere difettoso e/o il ranking di più o meno moduli critici, tale da consentire una allocazione delle risorse del testing “relativa” e non assoluta; iv) la predizione metric-based è una informazione statica che non prende in considerazione dati addizionali di testing online che possono correggere la predizione basata sulle osservazioni reali.

3 TECNICHE DI VERIFICA E VALIDAZIONE

Numerose tecniche di verifica e validazione sono state proposte con lo scopo di supportare l'identificazione dei difetti all'interno di programmi. Esse spaziano dalle tecniche di revisione del codice (attraverso uno studio sistematico del codice da parte degli ingegneri), alla tecniche di analisi del codice (che analizzano le proprietà di un programma, o una sua rappresentazione, attraverso algoritmi automatici), alla tecniche di testing dinamico (che mirano a trovare i difetti esercitando il sistema con degli input ed analizzandone gli output). Inoltre, tali tecniche sono supportate da tecniche di debugging, che a partire da un malfunzionamento rilevato permettono di identificare la causa radice e dove correggere il software.

3.1 CODE READING, CODE INSPECTIONS O REVIEWS, WALKTHROUGH

Una delle principali tecniche di analisi statica del codice è il ***Code reading*** (o *Desk checking*) [76]. Essa è basata sull'analisi informale del codice, condotta "a mano" – in inglese *desk check*. La tecnica del *Code reading* consiste in un'attenta lettura del codice che serve ad individuare errori e/o discrepanze con il progetto. Il lettore effettua mentalmente una pseudo-esecuzione del codice e dei processi di astrazione che lo conducono a verificare la correttezza del codice rispetto alle specifiche e il rispetto degli standard adottati.

E' possibile eseguire *Code reading* usando diverse strategie:

- una tecnica basata su checklists, utilizzate per garantire la qualità del *software engineering*, per verificare la conformità dei processi, del codice di standardizzazione e di prevenzione degli errori;
- una tecnica guidata dai Casi d'Uso;
- una tecnica basata su un ordine sistematico.

I tipici errori identificabili con questa tecnica sono: nomi di identificatori errati, errato innesto di strutture di controllo, loop infiniti, inversione di predicati, commenti non consistenti con il codice, incorretto accesso ad array o altre strutture dati, incoerenza tra tipi di dati coinvolti in una istruzione, incoerenza tra parametri formali ed effettivi in chiamate a subroutine, inefficienza dell'algoritmo, non strutturazione del codice, codice morto.

La ***Code Inspection*** o ***Review***, invece, è un esame formale e sistematico del codice sorgente per rilevare gli errori [76]. Il processo di esame del codice avviene attraverso delle riunioni formali in cui il codice software è presentato ai responsabili di progetto e agli utenti per il commento di approvazione. Prima di fornire alcun commento, il gruppo di ispezione controlla il codice sorgente per gli errori. In genere, questa squadra è composta da un Moderatore, un Reader, un Registratore, e un Autore.

- *Moderatore*: Conduce le riunioni di ispezione, verifica gli errori e assicura che il processo di ispezione è seguito.
- *Reader*: Parafrasa il funzionamento del codice software.
- *Registratore*: Tiene memoria di ogni errore nel codice del software. Questo libera gli altri membri del team dal compito di pensare in modo approfondito sul codice.

- *Autore*: Osserva il processo di ispezione del codice in modo silenzioso e aiuta solo se esplicitamente richiesto. Il ruolo dell'autore è quello di capire gli errori trovati nel codice.

Come detto sopra, durante il processo di ispezione del codice il *Reader* parafrasa il significato di piccole sezioni di codice, cioè traduce la sezione di codice da un linguaggio informatico a una lingua comunemente parlata. Il processo di ispezione viene effettuato per verificare se l'attuazione del codice software è fatto secondo i requisiti utente o meno.

In genere, per effettuare l'ispezione del codice sono eseguiti un certo numero di passi. Essi sono:

- *Planning*: Dopo che il codice viene compilato senza errori e messaggi di "warning", l'autore presenta i risultati al moderatore che è responsabile della formazione del team di controllo. Dopo che il gruppo di ispezione si è formato, il moderatore distribuisce le liste, nonché altri documenti correlati per ogni membro del team. Il moderatore pianifica le riunioni di controllo in coordinamento con i membri del team.
- *Overview*: Questo è un passaggio facoltativo ed è necessario solo quando i membri del team di ispezione non sono a conoscenza del funzionamento del progetto. Per mettere a conoscenza i membri del team, l'autore fornisce dettagli sul codice.
- *Preparation*: Ogni membro del team di ispezione esamina singolarmente il codice ed la relativa documentazione. È usata una checklist per assicurare che ogni area del problema sia selezionata e ispezionata. Ogni membro del gruppo di ispezione tiene una copia di questa check list, in cui sono menzionati tutti i settori problematici.
- *Inspection meeting*: Questa è una riunione che avviene con tutti i membri del team per rivedere il codice software. Il moderatore discute il codice in esame con i membri del team di ispezione.

Esistono due check list per la registrazione del risultato sul controllo del codice, vale a dire la *code inspection check list* e la *inspection error list*. La code inspection check list contiene un riepilogo di tutti gli errori di vario tipo presenti nel codice software. Questa lista di controllo è utilizzata per comprendere l'efficacia del processo di ispezione. La *inspection error list* fornisce i dettagli di ogni errore che richiede rielaborazione. Si noti che questo elenco contiene solo i dettagli di quegli errori che richiedono l'intero processo di codifica da ripetere.

Tutti gli errori nella *code inspection check list* sono classificati come maggiore o minore. Un errore è maggiore se si traduce in problemi e successivamente ne viene a conoscenza dell'utente. D'altro canto, piccoli errori sono errori di ortografia e di non conformità agli standard di scrittura del codice. La classificazione degli errori è utile quando il software deve essere consegnato all'utente e c'è poco tempo per rivedere tutti gli errori presenti nel codice software.

A conclusione della riunione di ispezione si decide se il codice deve essere accettato nella forma attuale o rimandato per la rielaborazione. Nel caso in cui il codice software ha bisogno di rielaborazione, l'autore fa tutte le correzioni proposte e lo compila. Viene poi inviato al moderatore. Il moderatore controlla il codice che è stato rielaborato. Se il moderatore è completamente soddisfatto del codice software, l'ispezione diventa formalmente completa e il processo di verifica del codice software inizia.

Un ultima tecnica per effettuare la revisione statica del codice è il ***Walkthrough***. Esso è un'analisi informale del codice svolta da vari partecipanti i quali '*operano come il computer*', cioè scelgono alcuni casi di test e simulano l'esecuzione del codice a mano (cioè, si attraversa – *walkthrough* - il codice). Il *Walkthrough* ha lo scopo di trovare e denunciare dei difetti senza però che il revisore si sostituisca al progettista o al programmatore, a cui comunque rimane il compito di scoprire l'errore e correggere il codice. Nel *Walkthrough*, l'organizzazione della riunione è simile a quella che avviene nel *Code Inspection*: ci sono dai 3 ai 5 partecipanti, le riunioni sono brevi, al massimo di 120 minuti, e c'è più attenzione sulla ricerca dei difetti piuttosto che sulla correzione, quindi si deve far attenzione a non "criminalizzare" il programmatore (autore del difetto).

Prima che venga effettuato il *Walkthrough*, l'autore deve distribuire tutta la documentazione che verrà consegnata e illustrata ad ogni persona presente. Ad esempio, se il *Walkthrough* è fatto attraverso la presentazione di diapositive, copie delle diapositive dovrebbero essere inviate per posta elettronica ai partecipanti alla riunione. Durante la riunione, l'autore dovrebbe sollecitare il feedback da parte del pubblico. Questa è l'occasione per scambiare idee nuove o alternative, e verificare che ogni persona capisca il documento che viene presentato. L'autore, attraverso la documentazione, deve assicurarsi che il prodotto è stato presentato nel modo più chiaro possibile.

Le seguenti linee guida possono aiutare un autore durante una riunione sul *Walkthrough*:

- Verificare che tutti siano presenti nel caso in cui si ha bisogno di riesaminare il prodotto. Questo potrebbe includere gli utenti, i soggetti interessati, i conduttori di ingegneria, manager e altre persone interessate.
- Verificare che tutti i presenti capiscano lo scopo della riunione riguardante il *Walkthrough*, e come il materiale sta per essere presentato.
- Descrivere ogni sezione della documentazione che deve essere coperta con il *Walkthrough*.
- Presentare il materiale in ogni sezione, assicurando che tutti i presenti capiscano il materiale da presentare.
- Portare alla discussione per identificare tutte le sezioni o materiali mancanti.
- Documentare tutte i problemi sollevati dai partecipanti al *Walkthrough*.

Dopo l'incontro, l'autore deve lavorare individualmente con i partecipanti che potrebbero fornire ulteriori informazioni o suggerimenti. Il documento deve quindi essere aggiornato per tener conto di eventuali altri problemi sollevati.

Anche il *Walkthrough*, come la *Code Inspection*, può essere organizzato nell'ambito dello sviluppo del processo software nel seguente modo: il gruppo di revisione e quello di codifica dovrebbero essere diversi e l'attività di verifica si dovrebbe concludere con un rapporto da includere nella documentazione del processo di sviluppo. La principale differenza fra *Code Inspection* e *Walkthrough* sta invece nel tipo di errori che la tecnica si prefigge di scoprire.

Il *Walkthrough* è inteso come un'esecuzione simulata del codice e porta generalmente a scoprire difetti relativi a problemi algoritmici. Il controllo statico è tendenzialmente identificato come un'attività manuale.

In realtà molti controlli statici possono essere eseguiti con l'ausilio di analizzatori di codice automatici molto più efficienti e rigorosi di qualsiasi professionista. Perciò, in base alle caratteristiche del linguaggio di programmazione usato, si dovrà ricorrere a controlli manuali – ad esempio per mezzo di ispezioni mirate – solo per individuare le categorie di difetti per cui non sono disponibili strumenti automatici.

3.2 ANALISI DI MODELLI STRUTTURALI DEL CODICE

L'analisi del codice a livello strutturale può suddividersi in due fasi, analisi di flusso ed analisi delle dipendenze, di cui la prima è propedeutica alla seconda. L'*analisi di flusso* esamina il flusso di controllo e dei dati durante l'esecuzione del programma; l'*analisi delle dipendenze* identifica le relazioni tra le computazioni costituenti il programma, che impongono vincoli sull'ordine delle loro esecuzioni.

3.2.1 ANALISI E RAPPRESENTAZIONE DEL FLUSSO

L'*Analisi di Flusso*, come detto all'inizio, esamina il flusso di controllo e dei dati durante l'esecuzione del programma. Essa può essere suddivisa in Analisi del flusso di controllo ed Analisi del flusso dei dati (la prima propedeutica alla seconda). L'*Analisi del flusso di controllo* (*control flow analysis*) determina la struttura di controllo del programma, cioè l'insieme di tutte le possibili sequenze di esecuzione; l'*Analisi del flusso dei dati* (*dataflow analysis*) determina il flusso dei dati attraverso la struttura di controllo.

3.2.1.1 ANALISI E RAPPRESENTAZIONE DEL FLUSSO DI CONTROLLO

L'*Analisi del flusso di controllo* è generalmente dedicata alla verifica della correttezza del codice. Il flusso di controllo del programma è rappresentato da un G*rafo di Flusso di Controllo* (o *Control flow Graph - CfG*).

La costruzione di questa struttura si rende necessaria per linguaggi con salti arbitrari nel flusso di controllo, causati ad esempio da costrutti di salto incondizionato; talvolta, in presenza di assunzioni molto restrittive sul flusso di controllo, possono essere sviluppati algoritmi di analisi dataflow che non richiedono la costruzione di grafi di flusso di controllo.

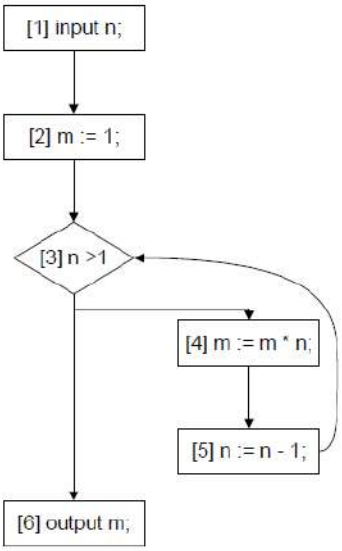
Il Grafo di Flusso viene esaminato per identificare ramificazioni del flusso di controllo e verificare l'esistenza di eventuali anomalie quali codice irraggiungibile e non strutturato.

Un *Grafo di Flusso* è definito [76] da una tripla $G = (N, A, s)$ tale che:

- (N, A) è un grafo orientato;
- $s \in N$ è il nodo iniziale;
- tutti i nodi del grafo sono raggiungibili da s.

Associando un'appropriata semantica agli oggetti componenti un Grafo di Flusso, esso può rappresentare il Flusso di Controllo (*Control Flow Graph*, *CFG*) di un programma (o di una procedura). Un *Control Flow Graph* è una rappresentazione di tutti i cammini che possono essere percorsi durante l'esecuzione di un programma. In modo più formale, possiamo definire un *Control Flow Graph* come un grafo orientato i cui nodi rappresentano computazioni, mentre gli archi indicano come il flusso di esecuzione possa spostarsi da una computazione all'altra. Ogni nodo di un *CFG* prende il nome di *basic block* e rappresenta una sequenza di zero o più istruzioni del programma, con un singolo punto di ingresso (la prima istruzione che viene eseguita) ed un unico punto di uscita (l'ultima istruzione eseguita). Quando il flusso di controllo dell'applicazione raggiunge un *basic block*, provoca l'esecuzione della prima istruzione del blocco; le istruzioni successive sono eseguite in ordine, senza possibili interruzioni o salti al di fuori del blocco finché l'ultima istruzione non è stata raggiunta.

Il Grafo di Flusso è, quindi, una struttura adatta a rappresentare la struttura di controllo del programma ad un livello di astrazione variabile, in quanto è possibile associare ad ogni nodo una singola istruzione, un'insieme di istruzioni, o perfino una singola operazione.




```
[ input n; ]1
[ m:= 1; ]2
[ while n>1 do ]3
      [ m:= m * n; ]4
      [ n:= n - 1; ]5
[ output m; ]6
```

Figura 3-1: Esempio di Grafo di Flusso di Controllo (con blocco base costituito da una istruzione).

Una nozione molto importante, per l'identificazione dei cicli nel programma e la definizione del concetto di dipendenza di controllo, è la *relazione di dominanza* tra i nodi di un CFG, di cui quindi diamo la definizione.

Relazione di Dominanza. Dato un grafo di flusso $G = (N, A, s)$ e una coppia di nodi n_i, $n_k \in N$ (non necessariamente distinti), allora n_i *domina* n_k se ogni cammino in G dal nodo iniziale a n_k contiene n_i.

Inoltre, n_i *domina propriamente* n_k se $n_i \neq n_k$ e n_i domina n_k, e n_i *domina direttamente* n_k se (a) n_i domina propriamente n_k, e (b) se n_n domina propriamente n_k, e $n_n \neq n_i$, allora n_n domina n_i.

Sono importanti le seguenti proprietà della relazione di dominanza sui nodi di un *CFG*:

- l'insieme dei dominatori del nodo iniziale *s* è costituito dal solo *s*;
- *s* domina tutti i nodi del grafo;
- la relazione di dominanza costituisce un ordinamento parziale riflessivo sui nodi del grafo;
- ogni nodo del grafo (tranne *s*) ha un unico dominatore diretto;
- l'insieme dei dominatori di un nodo è ordinato linearmente dalla relazione di dominanza.

Un modo possibile per rappresentare le relazioni di dominanza tra i nodi di un grafo orientato è costituito dall'*albero dei dominatori*. La sua radice è il nodo iniziale, e ciascun nodo domina soltanto i suoi discendenti nell'albero.

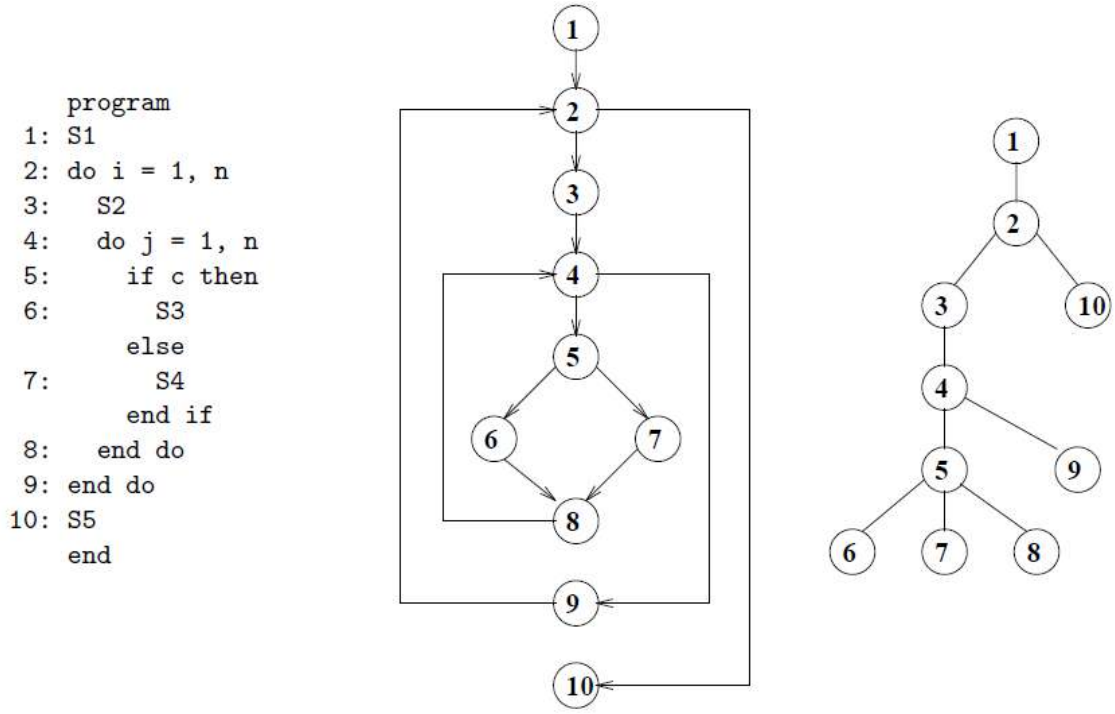


Figura 3-2: Esempio di Grafo di Flusso di Controllo e di albero dei dominatori.

Cicli. Nell'ambito dell'*Analisi Control Flow*, un concetto fondamentale è quello di ciclo (*loop*). La definizione formale di loop non è univoca, ma vi sono essenzialmente tre tipi di definizioni possibili: una basata sul concetto di *regione fortemente connessa* di un grafo di flusso, la seconda basata sul concetto di *intervallo* in un grafo di flusso e la terza sul concetto di *dipendenza di controllo*.

La rappresentazione del flusso di controllo mediante un grafo permette di caratterizzare facilmente un ciclo presente nella struttura di controllo di un programma, come un cammino non banale $\{n_1, n_2, \ldots, n_m\}$, tale che $n_1 = n_m$. L'identificazione del concetto di loop con quello di ciclo non permette però di ottenere la caratteristica di strutturazione nella definizione del loop: si vuole cioè una definizione che caratterizzi due loop come o disgiunti, o propriamente innestati, cioè l'uno contenuto interamente nell'altro. E' possibile ottenere tale caratteristica utilizzando, per la definizione di loop, il concetto di *regione fortemente connessa*. Dato un grafo G, si definisce *regione fortemente connessa* di G un sottografo G' di G tale che, per ogni coppia (n_1, n_2) di nodi di G', esiste un cammino non banale da n_1 a n_2 e viceversa. Dato un grafo di flusso $G = (N, A, s)$, un *loop*, con punto di ingresso s', è una regione fortemente connessa di G, indicata con $G' = (N', A', s')$, tale che, per ogni arco $(n, n') \in A$ con $n' \in N'$, si abbia: $n' = s'$ o $n \in N'$. In altri termini, un loop è dunque una regione fortemente connessa del grafo di flusso i cui nodi possono essere raggiunti, dall'esterno di tale regione, solo passando per il suo punto d'ingresso s' (non vi sono "salti" all'interno di un loop). Ne consegue quindi che il punto d'ingresso domina tutti i nodi del loop, ed è unico. Questa caratteristica della definizione di loop fornisce un metodo per la stesura di programmi aventi flusso di controllo strutturato. La generalizzazione di questa caratteristica conduce alla definizione di *grafo di flusso riducibile*.

Grafo Riducibile. Un grafo di flusso $G = (N, A, s)$ è detto *riducibile* [77] se è possibile dividere l'insieme A dei suoi archi in due sottoinsiemi disgiunti, A_1, A_2, tali che G, privato del secondo insieme di archi A_2, formi un sottografo di flusso aciclico $G' = (N, A_1, s)$, ed A_2 consista di soli archi il cui nodo destinazione domini il nodo origine.

Gli archi dei due insiemi A_1, A_2 sono spesso chiamati, rispettivamente, *forward edges* e *back edges*. Questa rappresentazione costituisce il modello base del flusso di controllo strutturato: i CFG di programmi sviluppati utilizzando solo costrutti di controllo strutturati, come cicli `do`, `while`, `repeat` e istruzioni condizionali, sono riducibili, mentre programmi sviluppati utilizzando costrutti di controllo non strutturati, come `goto`, i cui CFG siano riducibili possono essere trasformati in programmi strutturati.

La proprietà di riducibilità è importante non solo perché caratterizza i grafi di flusso dei programmi strutturati, ma anche perché l'efficienza di molti algoritmi per l'analisi dataflow intraprocedurale dipende fortemente da tale proprietà [77].

3.2.1.2 Analisi e Rappresentazione del Flusso dei Dati

L'analisi del flusso dei dati (*dataflow*) può essere descritta [76] come il processo di pre-esecuzione, cioè di esame e collezione di informazioni riguardo alla modifica e all'uso di definite *"quantità"* in un programma, e al modo in cui gli effetti di tali modifiche si propagano attraverso la struttura di controllo del programma. Tali quantità rappresentano i valori assunti dalle variabili del programma, ma le metodologie di tale analisi possono essere applicate a svariati problemi, detti *problemi dataflow*. Si tratta di un'attività intrinsecamente dinamica, ma alcuni aspetti possono essere analizzati staticamente.

L'analisi statica è legata alle operazioni che possono essere eseguite su una variabile:

- *definizione*, in cui alla variabile viene assegnato un valore;
- *uso*, dove il valore della variabile è usato in un'espressione o un predicato;
- *annullamento*, in cui, al termine di un'istruzione il valore associato alla variabile non è più significativo.

Ad esempio, nell'espressione $a := b + c$ la variabile a è definita, mentre b e c sono usate. La definizione di una variabile, così come l'annullamento, cancella l'effetto di una precedente definizione della stessa variabile, ovvero ad essa è associato il nuovo valore derivante dalla nuova definizione (o il valore nullo).

Una corretta sequenza di operazioni prevede che:

- L'uso di una variabile x deve essere sempre preceduto da una definizione della stessa variabile x, senza annullamenti intermedi;
- Un uso non preceduto da una definizione può corrispondere al potenziale uso di un valore non determinato.

Una definizione di una variabile *x* deve essere sempre seguita da un uso della variabile *x*, prima di un'altra definizione o di un annullamento della stessa variabile *x*. Se una definizione non seguita da un uso, corrisponde all'assegnamento di un valore non utilizzato e quindi potenzialmente inutile.

Quando la struttura di controllo del programma è ovvia e derivabile dalle caratteristiche sintattiche, non è necessaria la costruzione del *CFG* per la risoluzione di molti problemi *dataflow*. In alcuni casi, attraverso l'analisi *dataflow*, si ottiene una rappresentazione più accurata del flusso di controllo, quindi una iniziale assunzione conservativa sul flusso di controllo può permettere all'analisi *data flow* di procedere, e i suoi risultati possono poi essere applicati per raffinare l'analisi *control flow*. La derivazione delle relazioni di dominanza tra i nodi di un grafo di flusso è un tipico problema *dataflow*. La caratteristica dell'analisi *dataflow* è la propagazione dell'informazione raccolta localmente attraverso la struttura di controllo del programma.

La specifica dell'analisi è data mediante un insieme di equazioni che legano l'informazione che si sta analizzando ai punti del programma, ovvero ai nodi del grafo. L'informazione può essere propagata in avanti (*forward analysis)* e all'indietro (*backward analysis*). Ogni blocco di istruzioni del programma, la cui dimensione è scelta in base alla velocità/precisione della analisi desiderata, corrisponde ad un nodo del grafo.

Una metodologia che risolve gran parte dei problemi *dataflow* in maniera uniforme è l'astrazione sotto forma di "sistemi monotoni" per l'analisi *dataflow* (*Monotone Data Flow Systems*). Queste strutture sono considerate il modello formale più generale ed appropriato per la rappresentazione e la risoluzione dei problemi data flow fino ad ora sviluppato.

Prima di presentare alcuni tipici problemi dataflow e introdurre una metodologia generale di risoluzione di tali problemi c'è da ricordare brevemente un approccio alternativo all'analisi *data flow*, detto *Analisi degli Intervalli*. Esso è basato sulla riduzione di un grafo di flusso riducibile, attraverso una sequenza di trasformazioni sul grafo, in un grafo di un solo nodo.

La propagazione dell'analisi *data flow* è effettuata in base alla riduzione del grafo, raccogliendo innanzitutto l'informazione relativa a sezioni del programma crescenti e poi propagando l'informazione, in una fase inversa alla riduzione, fino ai blocchi base individuali del grafo di flusso. Per analizzare alcuni tipici problemi *data flow* assumiamo che:

- il CFG, *CFG* = (*N*; *A*; *s*), del programma sia costruito;
- l'informazione data flow locale a ciascun nodo del grafo sia disponibile;
- l'insieme delle variabili del programma sia costituito da sole variabili semplici;
- *Var* sia l'insieme delle variabili nel programma.

Generalmente, in un compilatore la fase di ottimizzazione trasforma i programmi in modo tale da renderli più efficienti senza, però, cambiarne l'output. Ciò viene fatto compiendo diverse operazioni, come ad esempio eliminazione delle espressioni comuni (*common – subexpression elimination*: se un'espressione è calcolata più di una volta, elimina tutti i calcoli tranne uno), eliminazione del codice morto (*dead – code*

elimination: eliminazione dei calcoli il cui risultato non verrà mai utilizzato), ripiegamento delle costanti (*constant folding*: se gli operandi di un'espressione sono costanti, esegui il calcolo durante la compilazione). Tali ottimizzazioni vengono effettuate sul linguaggio intermedio, per essere indipendenti dalla piattaforma. Le ottimizzazioni possono essere:

- *Interprocedurali*, anche dette ottimizzazioni globali, sono applicate all'interno del programma;
- *Intraprocedurali*, sono applicate ad una sola subroutine. Sono più semplici da attuare.

Per ottenere le prestazioni migliori, entrambi i tipi di ottimizzazione devono essere applicati. L'applicazione delle prestazioni si divide in due fasi, eseguite in alternanza fino a quando i miglioramenti di prestazione ottenuti diventano trascurabili:

1. Analisi statica del programma
2. Applicazione di trasformazioni al codice

Nel seguito sono descritte, a titolo di esempio, due tipologie di analisi statica del codice:

- Reaching Definitions Analysis (Analisi di tipo forward);
- Liveness Analysis (Analisi di tipo backward).

3.2.1.2.1 Reaching Definitions

Dato un punto del programma (punto d'ingresso del nodo $n \in N$) si vuole determinare l'insieme $RD(n)$ delle definizioni di variabili che sono attuali (ancora assegnate) quando l'elaborazione del programma raggiunge quel punto.

Per *definizione di una variabile* si intende l'istruzione *S* che può modificare il valore della variabile. Nel caso di un'analisi intra-procedurale, questo tipo di istruzioni sono le istruzioni di definizione di una variabile, cioè le istruzioni che vedono la variabile come elemento sinistro di una operazione di assegnazione. A questo proposito, un cammino nel grafo è detto *libero da definizioni* (*definition free*) per la variabile *v*, se *v* non è definita in alcuno dei nodi sul cammino in esame. Una definizione *S* di una variabile *v* in un nodo *n* è detta *esposta verso l'esterno* se è l'ultima definizione di *v* in *n*.

XDEFS(*n*) rappresenta l'insieme delle definizioni esposte verso l'esterno del nodo *n*. Se il nodo *n* in un *CFG* rappresenta un istruzione semplice, l'insieme *XDEFS*(*n*) equivale all'insieme *DEFS*(*n*) di tutte le variabili definite in *n*. E' da notare che questo insieme per istruzioni semplici, in genere ha un solo elemento oppure è l'insieme vuoto.

Si dice che una definizione *S* di una variabile *v* in un nodo n, esposta verso l'esterno, *raggiunge* il nodo *n'*, se esiste un cammino da *n* al punto di ingresso di *n'* libero da definizioni per *v*.

Dati due nodi *n* ed *n'* ed un cammino sul *CFG* che li connette, supponendo che in *n* esiste un assegnamento di una variabile *v*, qualsiasi altro assegnamento (definizione) di *C* in un nodo n_k sul cammino in esame, tra *n* ed *n'*, *uccide* tutte le definizioni (o assegnamento) di *v* che precedono n_k sul cammino. Quindi un punto del programma può "uccidere" una definizione di una variabile: se il comando associato a quel punto del

programma è un assegnamento (una definizione) di una variabile, vengono uccisi gli altri assegnamenti alla stessa variabile fatti in precedenza.

KILL(*n*) rappresenta l'insieme di tutte le uccisioni di variabili nel nodo *n*.

Inoltre, una definizione di una variabile *v* nel nodo *n*, su un cammino tra *n* ed *n'*, si dice *preservata* in un nodo n_p sul cammino se sul cammino fino a n_p non ci sono nodi dove la definizione di *v* viene uccisa e se lo stesso nodo n_p non uccide definizioni di *v*. Con *PRESERVED*(*n*) si indica l'insieme di tutte le definizioni preservate da un nodo *n*.

Un punto del programma può "generare" nuove definizioni, cioè in un punto del programma ci possono essere nuovi comandi di assegnamento di una variabile.

Ad esempio, nel caso del Control Flow Graph in Figura 3-3, sia la definizione di *m* del punto 2 che quella del punto 4 possono raggiungere il punto 6. Nel punto 5 non può arrivare la definizione di *m* del punto 2. In questo caso l'analisi sarà di tipo: (i) *Possible*, cioè ad ogni punto associo le definizioni che possono raggiungere quel punto; (ii) *Forward*, cioè propago i comandi di assegnamento dall'alto verso il basso fino a che non incontro un altro assegnamento.

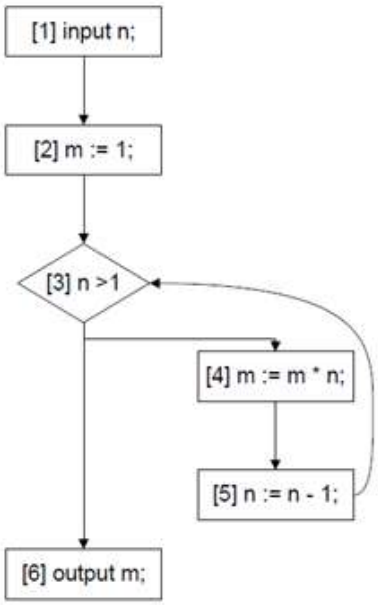


Figura 3-3: Esempio di Flusso di Controllo con assegnazioni.

Quindi i comandi che hanno un impatto significativo sulle *Reaching Definitions* sono solo quelli di assegnamento (che uccidono o generano).

La proprietà di interesse può essere rappresentata da insiemi di coppie

$$\{(v,n)|v \in Var; n \in N\}$$

associato ad ogni nodo del *CFG*, dove *v* è una variabile del programma e *n* è un punto del programma o nodo del grafo.

Se la coppia (v,n) viene associata ad un nodo *m* del programma, cioè ad un punto *m* del programma, questo vuol dire che la definizione (o assegnamento) di *v* nel nodo (o nel punto) *n* può raggiungere il nodo (il punto) *m*.

Il problema delle *Reaching Definitions* può essere espresso mediante un sistema di equazioni ricorsive.

Inizialmente, si assume che ogni variabile del programma non è ancora stata assegnata in alcun nodo e quindi:

$$\tau = (v, ?), \forall n \in N, \forall v \in Var$$

Percorrendo il grafo di flusso, ad ogni punto del programma dovremmo togliere le definizioni "uccise" e aggiungere quella "generate".

Le equazioni ricorsive che definiscono il problema delle *Reaching Definitions* sono le seguenti:

Per ogni punto *n* del programma

$$RD_{entry}(n) = \begin{cases} \tau = (v, ?) & \text{se } n \text{ è un punto iniziale} \\ \bigcup_{m \in pred[n]} RD_{exit}(m) \end{cases}$$

$$RD_{exit}(m) = gen_{RD}(m) \cup (RD_{entry}(m) - kill_{RD}(m))$$

La seconda espressione rappresenta gli assegnamenti generati in *m* sommati alla differenza tra gli assegnamenti che arrivano in *m* e gli assegnamenti uccisi.

Le *Reaching Definitions* relative al Grafo di Flusso in Figura 3-3 sono:

$$RD_{entry}(1) = \{(n, ?), (m, ?)\}$$
$$RD_{exit}(1) = \{(n, ?), (m, ?)\}$$
$$RD_{entry}(2) = \{(n, ?), (m, ?)\}$$
$$RD_{exit}(2) = \{(n, ?), (m, 2)\}$$
$$RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$$
$$RD_{exit}(3) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$$
$$RD_{entry}(4) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$$
$$RD_{exit}(4) = \{(n, ?), (n, 5), (m, 4)\}$$
$$RD_{entry}(5) = \{(n, ?), (n, 5), (m, 4)\}$$
$$RD_{exit}(5) = \{(n, 5), (m, 4)\}$$
$$RD_{entry}(6) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$$
$$RD_{exit}(6) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$$

dove vediamo che *n* può essere definito dall'input o nel punto 5; *m* in 2 o in 4.

E' possibile definire un algoritmo iterativo per la soluzione del problema della costruzione delle *reaching definitions*. L'algoritmo in esame prende come input il *CFG* e le informazioni sulle definizioni e gli usi di ogni variabile in ogni istruzione del programma da analizzare e costruisce iterativamente l'insieme delle *Reaching Definitions* per ogni variabile.

Input: grafo di controllo del programma
Output: coppia *RD*

Inizializzazione: **foreach** $n \in N$ do
  $RD_{entry}(n) = \emptyset$
end
$RD_{entry}(1) = \tau\{(v, ?), \forall v \in Var\}$

Iterazione: **while** ci sono aggiornamenti della RD_{exit} per almeno una variabile **do**
  foreach $n \in N$ **do**
    calcola $\bigcup_{m \in pred[n]}(gen_{RD}(m) \cup (RD_{entry}(m) - kill_{RD}(m))$
  end
end

Nella soluzione del problema della *Reaching Definitions*, si può notare come si analizzi il *CFG* dal solo nodo radice e si procede avanzando via via verso nodi del *CFG* relativi ad istruzioni successive. Man mano che si avanza nell'analisi del *CFG*, per ogni nodo, inoltre, si considerano le informazioni relative ai nodi che lo precedono nel *CFG*.

3.2.1.2.2 LIVENESS ANALYSIS

La *Liveness Analysis* ha il compito di analizzare la *vita* delle variabili. Si può dire che una variabile è *viva* se verrà usata in futuro, mentre è *morta* se il prossimo comando in cui compare è un assegnamento in cui la variabile stessa non compare a destra dell'uguale. Questo è quindi un'analisi di tipo *backward*.

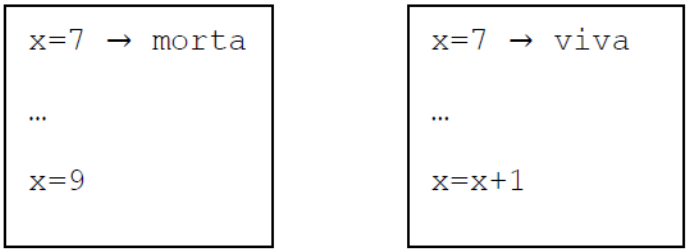


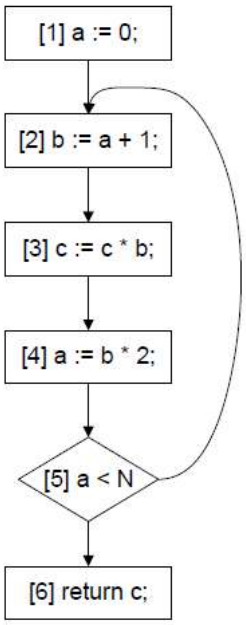
Figura 3-4. Liveness di variabili.

Il problema della *Liveness*, risulta quello di determinare, $\forall n \in N$, l'insieme *LV*(*n*) delle variabili che sono "*vive*" all'uscita del nodo *n*.

I comandi in cui una variabile viene usata generano vita, quelli di assegnamento (o definizione) uccidono. In modo più formale, una variabile *v* è detta *viva* all'uscita di un nodo *n* se esiste un nodo *n'* ∈ *N* ed un cammino *c* da *n* ad *n'* tale che:

- *c* sia libero da definizioni per *v*
- in *n'* è presente un uso di *v* esposto verso l'esterno.

Un uso *S* di una variabile *v* in un nodo *n* è detto *esposto verso l'esterno* se *n* non contiene alcuna definizione di *v* prima di *S*. *XUSES*(*n*) è l'insieme delle variabili con usi esposti verso l'esterno nel nodo *n*.




```
a = 0;
do{
  b= a+1;
  c+=b;
  a=b*2;
}
while (a<N);
return c;
```

Figura 3-5. Esempio di liveness analysis.

Come possiamo vedere in Figura 3-5, l'istruzione *4* fa morire la variabile *a*, mentre la variabile *b* viene usata e quindi è viva nell'arco *3* → *4*. Nell'istruzione *3* non viene assegnato nulla a *b* che quindi è viva nell'arco *2* → *3*. Nell'istruzione *2*, infine, *b* viene assegnata e dunque muore.

In genere, un grafo di flusso ha archi uscenti (*out-edges*) che portano a nodi successori, ed archi entranti (*in-edges*) che provengono da nodi predecessori. *pred*[*n*] e *succ*[*n*] denotano rispettivamente i nodi predecessori e successori del nodo *n*. Un assegnamento ad una variabile definisce tale variabile, quindi

$$def[n] = \{\text{insieme di variabili definite nel nodo } n\}$$

L'occorrenza di una variabile nell'espressione a destra dell'operazione di assegnamento usa tale variabile, cioè

$use[n]$ = {insiemi di variabili che sono usate nel nodo n}

Facendo riferimento alla Figura 3-5, si ottiene:

- $succ[5] = \{6, 2\}$
- $pred[2] = \{1, 5\}$
- $def[3] = \{c\}$
- $use[3] = \{b, c\}$

Una variabile è *live su un arco* se esiste un cammino orientato da tale arco verso un nodo in cui è usata (*use*) quella variabile e che non attraversa nodi dove quella variabile è definita (*def*).

Una variabile è *live-in* in un nodo se è *live* su ognuno dei suoi *in-edges* (archi entranti), mentre è *live-out* in un nodo se è *live* su almeno uno dei suoi *out-edges* (archi uscenti).

Definiamo il *live range* di *b* come $\{2 \rightarrow 3, 3 \rightarrow 4\}$. La variabile *a* è *live* negli archi $\{1 \rightarrow 2, 5 \rightarrow 2, 4 \rightarrow 5\}$ ed è *morta* negli archi $\{2 \rightarrow 3, 3 \rightarrow 4\}$.

L'informazione di *Liveness* si può calcolare nel modo seguente:

1. Se una variabile è in *use*[*n*], allora è *live-in* nel nodo *n*; quindi se un comando usa una variabile, tale variabile è viva all'entrata di quel comando.
2. Se una variabile è *live-in* nel nodo *n*, allora è *live-out* per tutti i nodi in *pred*[*n*].
3. Se una variabile è *live-out* nel nodo *n*, e non appartiene a *def*[*n*], allora la varibile è anche *live-in* nel nodo *n*; quindi se un comando ha bisogno del valore di *a* alla fine del comando *n* e *n* non fornisce quel valore, allora il valore di *a* è necessario anche all'entrata di *n*.

Possiamo ora rappresentare il problema delle "*live variables*" mediante il seguente insieme di equazioni:

$$LV_{exit}(n) = \begin{cases} \emptyset & \text{se } n \text{ è un punto finale} \\ \bigcup_{m \in succ[n]} LV_{entry}(m) & \end{cases}$$

Punto di partenza per risolvere la mutua ricorsione

$$LV_{entry}(m) = \underbrace{gen_{LV}(m) \cup (LV_{exit}(m) - kill_{LV}(m))}$$

utilizziamo informazioni in uscita per recuperare informazioni in entrata: *analisi backward*

dove:

$$gen_{LV}(m) = XUSES(m)$$

$$kill_{LV}(m) = XDEFS(m)$$

Dalla descrizione del problema, è infatti chiaro che, in un cammino tra due nodi *n* ed *n'*, ogni definizione uccide tutte le definizioni precedenti, visto che i valori della variabile in esame visibili nei nodi precedenti la definizione, non sono più visibili dopo una sua ridefinizione. Quando invece si usa il valore di una variabile, questo è indice del fatto che l'ultima definizione è ancora visibile nel punto in cui se ne fa uso.

L'analisi dataflow per la *liveness* comincia dalla fine del *CFG* e procede all'indietro lungo i cammini del *CFG* (*analisi backward*).

Come per le *Reaching Definitions*, è possibile definire un algoritmo iterativo anche per la soluzione del problema della *Liveness Analysis*. L'algoritmo in esame prende come input il *CFG* e le informazioni sulle definizioni e gli usi di ogni variabile in ogni istruzione del programma da analizzare e costruisce iterativamente l'insieme delle *Liveness* per ogni variabile.

Input: *CFG*; *use*/*def* per ogni nodo
Output: *live-in* e *live-out* per ogni variabile/nodo

Inizializzazione: Considera tutte le varibili non vive in tutti i nodi. Si utilizzano 2 vettori per memorizzare le informazioni di *live-in* e *live-out*: *in* e *out*.

foreach $n \in N$ **do**
  $in[n] = \{\}$;
  $out[n] = \{\}$
end

Iterazione: **repeat**
  foreach $n \in N$ **do**
    $in'[n] = in[n]$; $out'[n] = out[n]$; $in[n] = use[n] \cup (out[n] \backslash def[n])$;
    $out[n] = \cup_{m \in succ[n]} in[m]$;
  end
until $\forall n$: $in[n] == in'[n]$ && $out[n] == in'[n]$;

3.2.1.2.3 Catene Definizione – Uso e Uso – Definizione

Dato un nodo $n \in N$ e una variabile *v* per cui esiste un uso esposto verso l'esterno in *n* ($v \in XUSES(n)$), si vuole determinare l'insieme *UD*(*n*, *v*) delle definizioni *S'* di *v* che raggiungono *n* ($S' \in RD(n)$). Tale insieme è detto *catena uso – definizione* per il blocco *n* e la variabile *v*.

Dato un nodo *n* e una variabile *v*, viva all'uscita di *n* ($v \in LV(n)$), per cui esiste una definizione *S* esposta verso l'esterno in *n* ($S \in XDEFS(n)$), si vuole determinare l'insieme *DU*(*n, v*) degli usi $S' \in n'$ di *v* che definiscono *v* come variabile viva ($v \in XUSES(n')$). Tale insieme è detto *catene definizione-uso* per il blocco *n* e la variabile *v*.

In termini meno formali, la catena *UD*(*S*; *v*) è una lista di tutte le definizioni di *v* che possono raggiungere l'uso *S* di *v*, mentre la catena *DU*(*S*; *v*) è una lista di tutti gli usi di *v* che possono essere raggiunti dalla definizione *S* di *v*.

Le catene *UD* e *DU* trovano applicazione in svariate ottimizzazioni del codice, come l'eliminazione di codice morto, la propagazione di costanti (che è esso stesso un problema data flow), la "*scalar forward substitution*" e la "*induction variable substitution*", come pure nell'analisi *data dependance*.

Ad esempio:

- se *DU*(*S*; *v*) = ∅, allora l'esecuzione della definizione *S* di *v* non ha alcun effetto semantico, e *S* può essere eliminato come *codice morto*.
- una soluzione al problema della propagazione di costanti può essere trovata utilizzando le catene *UD* e *DU*: se si mostra che ogni definizione *S'* in *UD*(*S*; *v*) assegna lo stesso valore costante *c* a *v*, allora si può sostituire la costante *c* a *v* nell'uso *S*; analogamente, un valore costante assegnato a *v* nella definizione *S* può essere propagato a tutti gli usi connessi con quell'istruzione, utilizzando *DU*(*S*; *v*).
- per calcolare le catene *DU* è possibile utilizzare le informazioni riguardo le *reaching definitions*; a tal scopo, per ogni uso di una variabile *v*, bisogna mantenere informazioni (in una lista) di tutte le definizioni di *v* che raggiungono quell'uso.
- un modo per rappresentare i risultati di un'analisi di *liveness*, è proprio quello di utilizzare le catene *DU*, che mantengono, per ogni definizione, una lista di tutti i possibili usi della definizione.

La caratteristica essenziale dei problemi data flow è che la loro risoluzione implica la propagazione dell'informazione data flow associata ai nodi del grafo attraverso il grafo di flusso di controllo. Il processo di propagazione può essere implementato attraverso un algoritmo iterativo, che termina quando la propagazione non modifica l'informazione data flow associata ad alcun nodo del grafo. La direzione di propagazione dell'informazione data flow distingue i vari problemi data flow.

Ad esempio:

- per la *Reaching Definitions* la propagazione avviene dai predecessori verso i successori del grafo di flusso;
- per il problema delle *Live Variables* la propagazione avviene dai successori verso i predecessori del grafo.

Quando l'informazione si propaga nello stesso verso del flusso di controllo, si parla di problemi di tipo *top-down* (o *forward*), mentre quando l'informazione si propaga in verso contrario i problemi vengono detti di tipo *bottom-up* (o *backward*).

La direzione di propagazione va tenuta in conto per stabilire l'ordine di visita dei nodi negli algoritmi iterativi, in quanto gli algoritmi risulteranno convergere in modo efficace solo se l'ordine di visita coincide con la direzione di propagazione. La relazione di dominanza fornisce un modo efficace per controllare che l'ordine di visita rispetti la direzione di propagazione.

Un'altra caratteristica che distingue i problemi data flow è la maniera in cui l'informazione proveniente dai predecessori nella direzione di propagazione viene aggregata per produrre l'informazione locale al nodo in esame.

Per i problemi presi in esame, quali *RD* e *LV*, le corrispondenti equazioni mostrano che l'operazione di aggregazione è l'unione. Ciò corrisponde a risolvere *Problemi di Esistenza*, cioè:

- verificare se esiste un cammino che raggiunge il nodo in esame, e che verifichi la proprietà in esame, è un problema *possible*.
- i problemi per i quali l'operazione è l'intersezione vengono detti *all problems*, perchè l'operazione corrisponde a verificare la proprietà in esame per tutti i cammini che raggiungono il nodo in esame (problemi *definite*).

3.2.2 Analisi e Rappresentazione delle Dipendenze

Una *dipendenza* è un vincolo che ha una rilevanza semantica sull'ordine di esecuzione di due computazioni. Le dipendenze sorgono come risultato di due effetti diversi. Si ha una dipendenza tra due statements (o istruzioni) ogni volta che una variabile che compare in uno statement può assumere un valore errato se viene invertito l'ordine di esecuzione dei due statements. Dipendenze di questo tipo vengono dette *data dependences* (o dipendenze dei dati). Inoltre, si può avere una dipendenza tra uno *statement* e un *predicato*, il cui valore controlla l'esecuzione dello statement. Dipendenze di questo tipo sono dette *control dependences* (o dipendenze di controllo).

Il compito principale dell'analisi delle dipendenze è quello di prevedere e rappresentare la *"struttura di dipendenza"* del programma, che è determinata dalla struttura delle operazioni di riferimento in memoria (lettura e scrittura) e dalla struttura di controllo del codice. La struttura di dipendenza determina l'ordinamento necessario nell'esecuzione degli statements del programma; essa rappresenta, tra l'altro, il potenziale parallelismo presente nel codice. La struttura di dipendenza prevista a tempo di compilazione è di solito un'approssimazione della vera struttura di dipendenza e deve essere conservativa, nel senso che deve includere tutti i vincoli della struttura effettiva.

3.2.2.1 Control Dependence

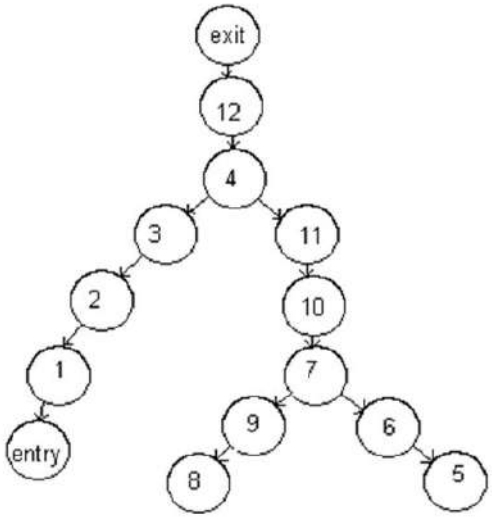
Le relazioni di flusso di controllo tra gli statements di un programma sono oggetto dell'analisi *control flow* e sono rappresentate con il *Control Flow Graph.* Esse non rappresentano, in maniera esplicita, le condizioni essenziali che controllano l'ordine di esecuzione degli statements; ciò è dovuto al fatto che l'analisi *control flow*, e quindi il *CFG*, tiene conto della sequenza testuale degli statements nel codice, che non coincide con l'ordinamento nella loro esecuzione, dettato dalle condizioni di controllo. L'*analisi control dependence*, quindi, riassume le condizioni essenziali che controllano l'esecuzione degli statements, senza considerare il loro ordinamento testuale. Il concetto di dipendenza di controllo è definito in termini del control flow graph, attraverso la definizione di *grafo a singola uscita*, e della relazione di *post-dominanza*.

Un *Grafo di flusso a singola uscita* è definito da una quadrupla $G = (N, A, s, t)$ tale che (N, A, s) è un grafo di flusso, $t \in N$ è il nodo finale di G, ed esiste un cammino da ogni nodo di G a t.

Così come un *CFG* con punti di ingresso multipli può essere facilmente convertito in un *CFG* con un singolo punto d'ingresso (aggiungendo un nodo iniziale globale, collegato con un arco per ogni singolo punto d'ingresso) allo stesso modo un *CFG* con punti d'uscita multipli può essere convertito in un *CFG* a singola uscita. Invece, mentre i nodi che non sono raggiungibili dal nodo iniziale possono sempre essere rimossi, ciò non è possibile per i nodi che non raggiungono il nodo finale. Un esempio può essere quello di programmi con un loop infinito. In tal caso, l'analisi sulla control dependence non può essere applicata all'intero programma, ma deve essere diviso in regioni singolo ingresso-singola uscita, e l'analisi può essere applicata a ciascuna regione.

Dato un *grafo di flusso a singola uscita* $G = (N, A, s, t)$, ed una coppia di nodi $x, y \in N$, allora x *post-domina* y se $x \neq y$ ed ogni cammino da y a t contiene x.

È da notare che questa definizione di post-dominanza non include il nodo iniziale, e che un nodo non può post-dominare se stesso.




```
1.    read (n)
2.    i = 1
3.    sum  = 0
4.    while i <= n do
5.       sum = 0
6.       j = 1
7          while j <= i do
8.           sum = sum + j
9.           j = j + 1
           endwhile
10         write (sum, i)
11.        i = i + 1
      endwhile
12.   write (sum, i)
```

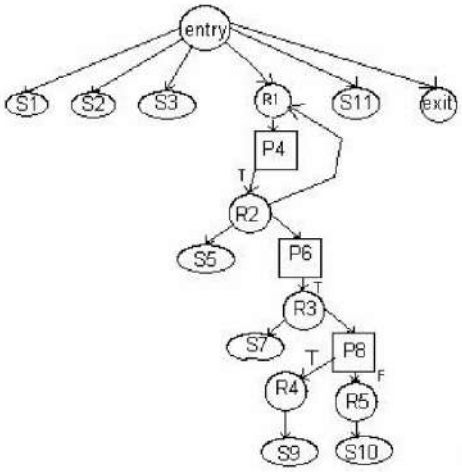
Figura 3-6. Albero di Post-Dominanza.

Dato un grafo di flusso a singola uscita $G = (N, A, s, t)$, ed una coppia di nodi $x, y \in N$, allora y è *control dependent* da x se

- esiste un cammino non banale da x a y ogni nodo del quale (tranne x e y) sia post-dominato da y,
- x è non post-dominato da y.

Meno formalmente: se y è *control dependent* da x allora il nodo x deve avere due uscite: seguendo uno dei due cammini il nodo y sarà in ogni caso incontrato (e quindi lo statement y sarà in ogni caso eseguito), mentre seguendo l'altro cammino il nodo y potrà non essere incontrato. Dalla definizione segue chiaramente che un nodo che ha una singola uscita, quindi un singolo successore, non può essere sorgente di una dipendenza di controllo. Possono essere quindi sorgenti di dipendenze di controllo solo gli statements di controllo del linguaggio in oggetto: il predicato dello statement di controllo determina le condizioni per l'esecuzione degli statements dipendenti da esso.

La relazione di *control dependence* tra statement del programma viene rappresentata mediante il *Control Dependence Graph* (introdotto in [79] come sottografo del *Program Dependence Graph*): i nodi di tale grafo sono gli statements del programma e gli archi rappresentano le relazioni di *control dependence* tra essi; gli archi possono essere etichettati, con dei valori che dipendono dai valori che può assumere il predicato di controllo del nodo sorgente della dipendenza: ad esempio valori `true`, `false` per predicati booleani, o valori interi per statements di controllo multibranch, come il `case`.




```
S1.    read i
S2.    sum = 0
S3.    done = 0
P4.    while i <= 5 and !done do
S5.       read j
P6.       if j >= 0 then
S7.          sum = sum + j
P8.          if sum > 100 then
S9.             done = 1
               else
S10.               i = i + 1
           endif
       endwhile
S11.   print sum
```

Figura 3-7. Control Dependence Graph.

La costruzione del *Control Dependence Graph* presuppone la costruzione dell'albero dei post-dominatori, e la disponibilità del *Control Flow Graph*. A questo punto, diamo la definizione di *loop* basata sulla proprietà di *control dependence.* A tale scopo, facciamo notare che uno statement *x* può essere *control dependent* da se stesso: infatti la seconda condizione vale sempre quando $x \equiv y$, e la prima condizione può, ad esempio, essere soddisfatta da un cammino costituito da un unico edge, da *x* a se stesso.

Loop: Un *Loop* è una regione fortemente connessa del Control Dependence Graph. La relazione tra questa definizione di loop e quella basata sul flusso di controllo, è la seguente. Gli statement costituenti un loop secondo la definizione basata sulla control dependence sono un sottoinsieme degli statement costituenti un loop secondo la definizione basata sul control flow: essi sono gli statement di controllo che determinano un'uscita dal loop. Gli altri statement del control flow loop che non appartengono al control dependence loop, sono in ogni caso collegati a questi ultimi mediante cammini di control dependence. Essi corrispondono intuitivamente al corpo del loop. Un vantaggio della definizione basata sulla control dependence è che con essa i loop innestati sono rappresentati come insiemi di nodi distinti (regioni fortemente connesse), con un edge di dipendenza di controllo tra il loop esterno e ciascun loop interno immediato.

3.2.2.2 Data Dependence

Informalmente, uno statement S_2 è *data dependent* da uno statement S_1 se in entrambi gli statements vi è un riferimento (definizione o uso) alla stessa variabile, con almeno una definizione, che impone un vincolo di ordinamento nell'esecuzione dei due statements.

In particolare, la dipendenza può essere [80]:

- *true*, se la variabile viene definita dallo statement che precede l'altro nell'ordinamento, e viene usata dallo statement che segue;
- *anti*, se la variabile viene usata dal primo statement e definita dal secondo;
- *output*, se la variabile è definita da entrambi gli statement.

Gli ultimi due tipi di dipendenza sono anche detti *false dipendenze*, in quanto sorgono dalla principale caratteristica dei linguaggi imperativi di poter esprimere il riuso di locazioni di memoria attraverso il riassegnamento di variabili. Esse possono essere eliminate mediante l'introduzione di nuove variabili.

Denominati *DEF*(*S*) ed *USE*(*S*) gli elementi (variabili scalari o elementi di array) rispettivamente definiti ed usati dallo statement *S*, si ha la seguente definizione:

Lo statement S_2 è *data dependent* dallo statement S_1 se:

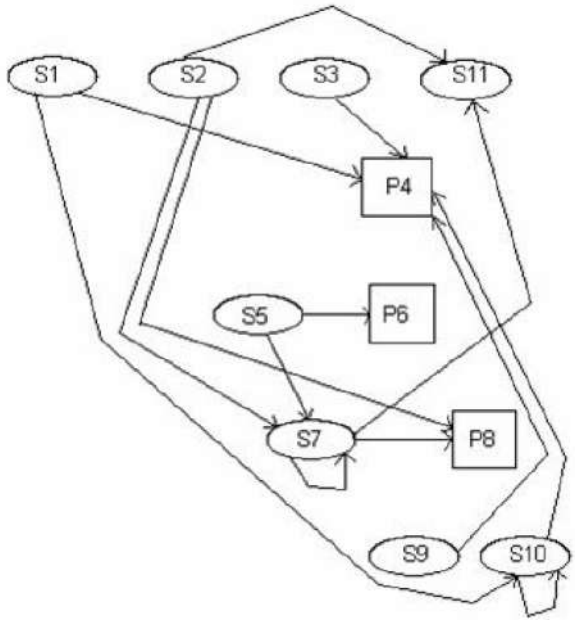
- esiste una variabile *v* tale che:

$$v \in DEF(S_1) \cap USE(S_2) \quad \text{(true dependence)}$$
$$v \in USE(S_1) \cap DEF(S_2) \quad \text{(anti dependence)}$$
$$v \in DEF(S_1) \cap DEF(S_2) \quad \text{(output dependence)}$$

- esiste un cammino nel *CFG* da S_1 a S_2, libero da definizioni per *v*.

Le dipendenze sui dati tra statements vengono rappresentate attraverso il *Data Dependence Graph*, i cui nodi rappresentano gli statement e i cui archi, orientati, rappresentano i vincoli di dipendenza.

Gli archi di dipendenza sono di solito etichettati con i simboli δ^t, δ^a e δ^o, che denotano rispettivamente le dipendenze true, anti ed output.




```
S1.    read i
S2.    sum = 0
S3.    done = 0
P4.    while i <= 5 and !done do
S5.       read j
P6.       if j >= 0 then
S7.          sum = sum + j
P8.          if sum > 100 then
S9.             done = 1
             else
S10.            i = i + 1
          endif
       endwhile
S11.   print sum
```

Figura 3-8. Data Dependence Graph.

Dalla definizione che è stata data, il problema della determinazione delle *data dependences* sembrerebbe riconducibile ad un problema dataflow. Ciò è vero se l'analisi è limitata alle variabili scalari, ed in tal caso la determinazione delle dipendenze *true* coincide con la determinazione delle catene uso-definizione, mentre la determinazione delle dipendenze *anti* coincide con la determinazione delle catene definizione-uso. L'analisi dataflow risulta inadeguata se si considerano i singoli elementi degli array come singole variabili.

Nell'analisi dataflow, i riferimenti ad elementi di array sono considerati come riferimenti a tutto l'array, e ciò comporta un vincolo eccessivo, soprattutto nella determinazione delle dipendenze.

La principale differenza tra l'a*nalisi dataflow* e l'*analisi data dependence* sta nella capacità di quest'ultima di tener conto delle dipendenze tra i singoli elementi degli array; ciò implica, oltre a una maggiore precisione nell'analisi delle dipendenze tra gli statements, anche la capacità di prefigurare i pattern con cui le locazioni di memoria sono accedute dalle operazioni di riferimento in memoria.

L'*analisi data dependance* si concentra principalmente sulle variabili con indici e ciò implica la necessità di rappresentare in maniera esplicita gli effetti della presenza dei loop sulle dipendenze. Quando si verifica un loop, si hanno differenti esecuzioni dello stesso statement. È quindi opportuno definire queste dipendenze.

In presenza di un loop, si ha una dipendenza di tipo *loop carried* se l'elemento che determina la dipendenza è definito (dall'istanza di uno statement) in una delle iterazioni del loop, e usato in un'iterazione successiva (da un'istanza di un'altro statement, o anche da una differente istanza dello stesso statement). Se la definizione e l'uso avvengono durante la medesima iterazione del loop, la dipendenza è detta di tipo *loop independent*.

In presenza di dipendenze *loop carried*, si potrà avere il caso in cui lo statement sorgente di una *true dependence* segue, secondo l'ordine lessicale, lo statement destinazione (*lexically backward dependence*), o il caso in cui uno statement dipende da se stesso.

Nel caso in cui la variabile oggetto della dipendenza *loop carried* sia uno scalare, e quindi invariante rispetto all'indice del loop, la sua rappresentazione non crea problemi: infatti le dipendenze fra le istanze dei due statement coinvolti (o dell'unico) si ripetono allo stesso modo per tutte le iterazioni del loop, e quindi le dipendenze tra le varie istanze possono essere rappresentate, in maniera riassuntiva, come un'unica relazione di dipendenza tra i due statement.

Nel caso delle variabili con indici (dove gli indici sono espressioni dipendenti dall'indice del loop) la situazione è più complessa, perché gli elementi oggetto della dipendenza (gli elementi dell'array) dipendono dalle iterazioni del loop. È quindi conveniente rappresentare esplicitamente le dipendenze tra istanze differenti di statement, e quindi tra iterazioni successive di un loop.

Un loop innestato, composto da d loop normalizzati definisce uno *Spazio delle Iterazioni* Z^d [81], in cui ciascuna iterazione del loop è rappresentata da un punto $\vec{p} = (p_1, \ldots, p_d)$; p_i è il valore dell'i-mo indice del loop nest.

Le dipendenze loop carried possono essere rappresentate nello spazio delle iterazioni mediante "vettori delle dipendenze". Un *vettore delle dipendenze* (denotato con $\vec{d} = (d_1, \ldots, d_2)$ in un loop n-innestato) definisce un arco orientato tra due punti $\vec{p}_1$ e $\vec{p}_2$ dello spazio delle iterazioni, se esiste una dipendenza (loop carried) tra le istanze, corrispondenti alle iterazioni $\vec{p}_1$ e $\vec{p}_2$, di due qualsiasi statement del corpo del loop. L'orientazione dell'arco è dalla sorgente alla destinazione della dipendenza.

L'insieme dei punti che rappresentano il loop nest nello spazio delle iterazioni, considerati come nodi, e dei vettori delle dipendenze, considerati come archi, costituisce il *Grafo delle Dipendenze tra le iterazioni del loop*. Ogni ordinamento topologico del grafo costituisce un ordine legale di esecuzioni delle iterazioni del loop, in quanto tutte le dipendenze sono soddisfatte.

Per le espressioni indice delle variabili coinvolte nelle dipendenze, poste determinate condizioni, è possibile determinare in modo esatto le iterazioni del loop tra cui si presentano le dipendenze: in tal caso, i vettori delle dipendenze sono vettori propriamente detti, cioè $d_i \in Z$, e vengono denominati *vettori distanza*.

Si noti che, siccome un'iterazione può dipendere soltanto da un'iterazione eseguita precedentemente a questa, non successivamente, un vettore distanza valido deve essere *lessicograficamente positivo*, (cioè, il suo primo elemento diverso da zero deve essere positivo): se il suo primo elemento non nullo fosse negativo, ciò indicherebbe una dipendenza su un'iterazione futura.

In generale, non è sempre possibile determinare, a tempo di compilazione, i vettori distanza; ma vi è abbastanza informazione per caratterizzare parzialmente la dipendenza. Nel caso generale, il vettore della dipendenza $\vec{d} = (d_1; \ldots; d_n)$ può essere quindi costituito da intervalli (eventualmente infiniti) invece che

semplicemente da interi: $d_i \equiv [d_i^{min}, d_i^{max}]$. Esso quindi rappresenta un insieme di vettori distanza, e quindi un insieme di coppie di iterazioni che possono essere in relazione di dipendenza.

In molti casi è possibile stabilire soltanto la *direzione* della dipendenza, cioè il segno delle componenti del vettore dipendenza; possono aversi i seguenti casi:

- il segno positivo, o direzione "*forward*" (simbolo <), che equivale all'intervallo [1, +∞]. La direzione forward (<) indica che la dipendenza attraversa la frontiera dell'iterazione "in avanti".
- Il segno negativo, o direzione "*backward*" (simbolo >), che equivale all'intervallo [-∞, -1]. La direzione backward (>) indica che la dipendenza attraversa la frontiera dell'iterazione "all'indietro": tale caso può presentarsi solo se vi è una direzione forward in un loop più esterno del loop nest.
- Il segno di uguaglianza, o direzione "nulla" (simbolo =), che equivale al valore nullo per la componente del vettore dipendenza. La direzione nulla indica che la dipendenza non attraversa la frontiera della dipendenza.
- Il segno $\pm$, o mancanza di informazione sulla direzione (simbolo $*$), che equivale all'intervallo [-∞, +∞].

Sono possibili combinazioni delle prime tre direzioni (simboli $\geq$, $\leq$, $\neq$).

Per rappresentare questo tipo incompleto di informazione è stato quindi introdotto il *vettore delle direzioni*, di dimensionalità uguale al nesting del loop, le cui componenti sono costituite dai simboli che indicano una direzione ($d_i \in \{<, >;=;\geq, \leq, \neq; *\}$). Per quanto riguarda le componenti per le quali è possibile determinare con esattezza la dipendenza ($d_i \in Z$), esse possono anche essere rappresentate, nel vettore delle direzioni, mediante tale numero (si ha così un ibrido vettore distanza - vettore direzione); la direzione corrisponderà naturalmente al segno dell' intero d_i. Così come per i vettori distanza, anche i vettori direzione, per rappresentare dipendenze legali, debbono essere lessicograficamente positivi: cioè la prima direzione non-uguale deve essere una direzione forward (ad es., il vettore (=;>) non è plausibile).

La caratterizzazione delle dipendenze tra le iterazioni mediante il vettore delle direzioni permette di riassumere in maniera efficace alcune proprietà delle relazioni di dipendenza. Ad esempio caratterizziamo il concetto di profondità o livello della dipendenza in un loop nest. Dato un loop d-innestato, e numerati i loop componenti dal più esterno al più interno, la *profondità* (o *livello*) *di una dipendenza* tra le iterazioni del loop è definita come il livello di nesting del loop più esterno che deve essere eseguito serialmente per soddisfare la condizione di dipendenza. Tale loop è detto il *portatore* (carrier) *della dipendenza*. Una dipendenza loop indipendent ha, per definizione, profondità infinita.

3.2.2.3 Program Dependence Graph

Fino ad ora sono state introdotte delle strutture atte a rappresentare, separatamente, le dipendenze sul controllo (*Control Dependence Graph*) e sui dati (il *Data Dependence Graph*).

La rappresentazione separata della dipendenze sul controllo e sui dati viene fuori dal fatto che è possibile, mediante una ristrutturazione del codice detta *if-conversion* [82], convertire la *control dependence* in *data dependence*. Quindi l'analisi delle dipendenze e l'analisi del flusso di controllo possono essere ottenute in

maniera indipendente, e così anche le rispettive rappresentazioni. Questo, però, avviene al costo di una profonda ristrutturazione del programma, che, mantiene la semantica, ma distrugge l'informazione sulla struttura di controllo, cruciale nell'ambito della trasformazione del codice e della sua comprensione.

Una rappresentazione del programma che unisce i risultati ottenuti dalla analisi di flusso e dalla analisi delle dipendenze, è quella fornita dal *Program Dependence Graph*, che rende esplicite entrambe le dipendenze, di dati e di controllo, per ciascuna operazione in un programma.

Quindi, indicati con *CDG* = (*N*, E_c, *s*) e *DDG* = (*N*, E_d, *s*) rispettivamente il *Control Dependence Graph* ed il *Data Dependence Graph* di un programma (dove *N* rappresenta in entrambi i casi, l'insieme delle operazioni del programma), il *Program Dependence Graph* è il grafo *PDG* = (*N*, $E_{c,} \cup E_d$, *s*) unione dei due grafi.

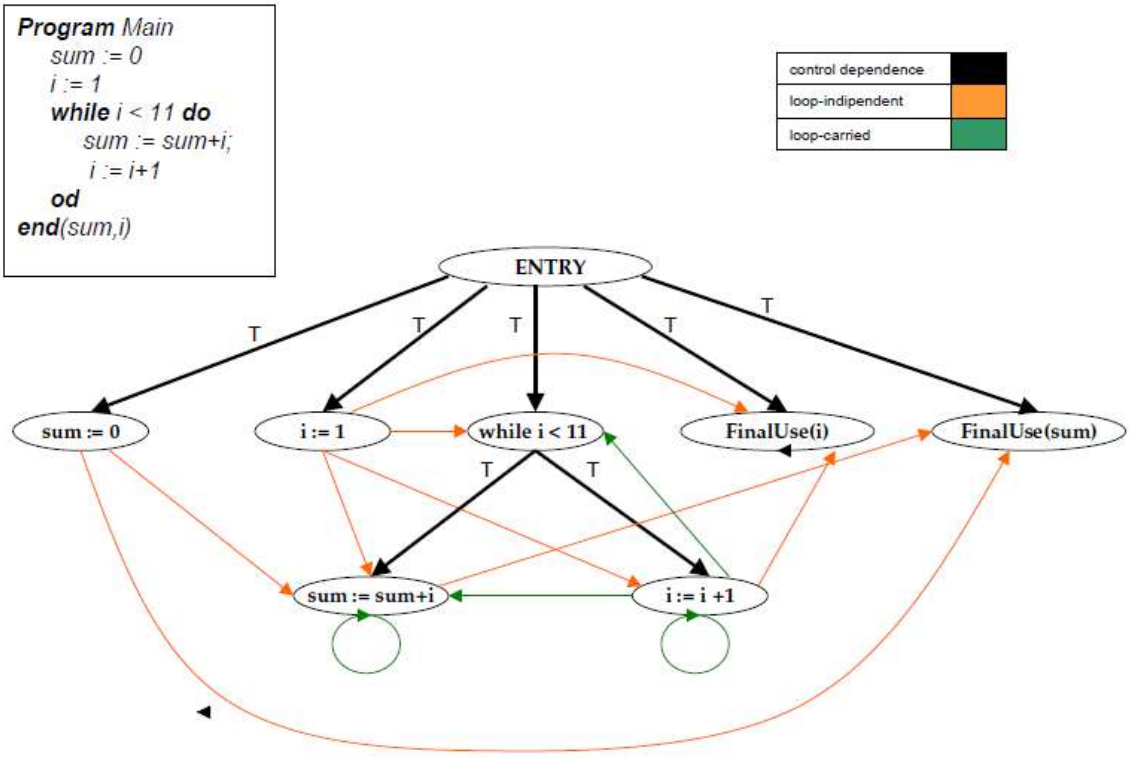


Figura 3-9. Program Dependence Graph.

Esso è quindi un grafo orientato i cui nodi rappresentano le operazioni del programma, e i cui archi rappresentano le dipendenze di controllo e di dati tra essi.

Gli archi di un *Control Dependence Graph* sono etichettati con il branch della dipendenza, mentre gli archi di *Data Dependence Graph* sono etichettati in base al tipo di dipendenza (*true*, *anti* o *output*; inoltre, ortogonalmente: *loop independent* o *loop carried*).

Gli archi di dipendenza loop carried vengono inoltre etichettati con il livello, ed in alcuni casi con un puntatore al nodo che rappresenta il loop carrier della dipendenza. Gli archi di data dependence possono inoltre essere etichettati con i rispettivi vettori delle dipendenze (vettori distanza o vettori direzione, a

seconda dell'accuratezza dell'analisi effettuata). Un *PDG* può essere un *multigrafo*, cioè può contenere archi multipli che connettono la stessa coppia di nodi. In tal caso, tali archi vengono distinti dal loro tipo (data o control) e/o dalle loro etichette.

3.2.2.4 Analisi e Rappresentazione Intraprocedurale e Interprocedurale

Come abbiamo visto, l'analisi dataflow è una collezione di tecniche tramite cui si possono derivare staticamente informazioni globali riguardanti il flusso dei dati all'interno del programma.

Questo tipo di analisi viene tradizionalmente utilizzata nelle ultime due fasi di un compilatore (ottimizzazione del codice e generazione di questo) ed è divenuta parte integrante di altri strumenti come *debuggers* e *program slicers*. In base al "grado di risoluzione" dell'analisi del flusso, essa può essere classificata, in analisi *intraprocedurale* e *interprocedurale*.

L'analisi *intraprocedurale* considera il flusso dei dati all'interno di una sola procedura e assume alcune approssimazioni riguardo le definizioni e gli usi dei parametri passati (per indirizzo) e delle variabili globali. L'analisi *interprocedurale* considera anche chiamate a procedure e funzioni, analizza dunque le informazioni riguardanti il punto di entrata nel corpo della procedura chiamata, il flusso che corrisponde all'uscita dal corpo della procedura e il ritorno del controllo alla procedura chiamante.

L'analisi control flow interprocedurale costruisce, a partire dalla costruzione del grafo di flusso di controllo, un grafo di chiamata (*Call Graph*), rappresentante le relazioni di chiamata fra le procedure. L'analisi dataflow interprocedurale prende in considerazione il passaggio di parametri fra procedure, e i problemi dell'*aliasing* e dei *side effects*. Una variante del call graph, il *Program Summary Graph*, permette l'analisi dataflow interprocedurale di tipo flow sensitive (viene preso in considerazione il flusso di controllo interno alle procedure). Un estensione del *Program Summary Graph*, l'*Interprocedural Flow Graph*, permette la definizione di coppie definizione-uso interprocedurali. Un'estensione del *Program Dependence Graph*, il *System Dependence Graph*, tiene conto anche dell'analisi interprocedurale.

3.3 PROGRAM SLICING

Negli anni, con lo sviluppo di sistemi software avanzati si è presentata la necessità di trovare delle tecniche di ausilio alla tradizionale attività di *debugging* e *testing*.

Una tecnica, denominata *slicing*, viene introdotta per tale scopo e nasce per consentire ai programmatori, durante queste fasi, di astrarre in modo automatico dall'insieme delle istruzioni del programma un suo sottoinsieme contenente le sole istruzioni che contribuiscono a determinare il valore di una o più variabili in corrispondenza di un determinato punto del codice.

Secondo *Weiser*, una *program slice S* è una "*fetta*" del programma principale che potenzialmente determina il valore di un insieme di variabili in un particolare punto del codice sorgente.

Grazie a questa definizione sono state presentate molte implementazioni e tecniche ausiliarie per superare i problemi intrinseci nell'analisi statica del codice sorgente al fine di ottenere una *slice* che, risultasse corretta rispetto ad alcuni parametri, e che nello stesso tempo contenesse solamente le istruzioni realmente influenti.

Dividere i programmi in moduli è sempre stato un approccio molto usato per far fronte alla crescente complessità dei sistemi. Anche la tecnica del *Divide et Impera* è diventata, con la programmazione *Object Oriented*, un rimedio efficace per lo sviluppo e la manutenzione di software, evidenziando l'importanza del concentrare la propria attenzione su singoli componenti in modo da realizzare un buon prodotto. Ricavare una *slice* significa infatti ridurre un programma alle sole istruzioni che influenzano il valore delle variabili in un determinato punto del codice. Tale punto viene denominato *slicing criterion*.

Dalla definizione originale di questa tecnica, basata unicamente sull'analisi statica del codice sorgente (*static slicing*), sono state intraprese numerose ricerche per superare i limiti di questo metodo. Da un lato si è cercato di superare le problematiche riguardanti alcuni particolari costrutti dei linguaggi di programmazione, come procedure, operatori di controllo del flusso, puntatori, array, etc., estendendo la tecnica ad approcci basati sul paradigma ad oggetti; dall'altro invece si è cercato di utilizzare ulteriori informazioni sull'input di una particolare esecuzione al fine di ridurre dinamicamente il codice (*dynamic slicing*) per poi effettuare considerazioni analoghe al caso statico.

In generale, il *program slicing* è in grado di assistere gli sviluppatori in numerose attività che vanno ben oltre il debugging del codice sorgente. Per esempio, gli addetti alla manutenzione di un programma dopo il suo rilascio si trovano di fronte a problemi simili a quelli dei program integrator. Essi, infatti, devono capire a fondo il software esistente ed integrare le eventuali modifiche senza che queste abbiano un impatto negativo sulle parti di codice non modificate (*Program Integration*). Per fare ciò è possibile utilizzare un tipo particolare di *slice*, chiamata *decomposition slice*, in grado di catturare tutte le istruzioni che hanno a che fare con una variabile e indipendente dalla posizione nel programma. Una *decomposition slice* può essere particolarmente utile quando i programmatori sanno che una determinata variabile dovrà cambiare valore nel corso dell'esecuzione del programma.

A volte, gli addetti al mantenimento di un programma si trovano anche a dover ritestare il software dopo averlo modificato (fase di *Testing*). Questo processo porta ad eseguire il programma in un gran numero di casi di test, anche per la più piccola delle modifiche. Esistono, infatti, alcuni algoritmi che usano il *program slicing* per determinare quali componenti di un programma siano stati influenzati transitivamente da una modifica ad un'istruzione *p*.

Infine, effettuare il *debug* dei programmi è sempre stato considerato un compito difficile. Spesso la parte più difficile del *debugging* sta proprio nello scoprire il bug. Ciò è dovuto al fatto che nella porzione di codice che viene esaminata possono esserci molte istruzioni e non tutte hanno necessariamente effetto sull'istruzione che effettivamente crea il problema. Uno strumento che calcola le *program slice* può essere un aiuto fondamentale per chi deve effettuare il *debugging* di un programma, in quanto permette al programmatore di concentrarsi solo sulle istruzioni che contribuiscono ad un malfunzionamento, rendendo più efficiente l'identificazione del punto di fault. Il *dynamic slicing* risulta utile per effettuare il *debugging* legato ad uno specifico test case poiché mette in luce tutte le possibili istruzioni che certamente influiscono su una variabile.

3.3.1 Static Slicing

Una volta illustrate le potenzialità che la tecnica dello *slicing* ha nel ciclo di sviluppo del software, andiamo a trattare l'approccio di tipo statico. Weiser definisce il processo di elaborazione della *slice* usando solo informazioni statiche, senza fare alcuna assunzione sull'input. La *program slice S* è un programma eseguibile e ridotto, ottenuto da un altro programma *Q*, rimuovendo parti di codice, in maniera da mantenere comunque una consistenza funzionale rispetto allo *slicing criterion* (*i*; *W*), dove *i* è un'istruzione di *Q* da osservare, mentre *W* è un sottoinsieme delle variabili di *Q* su cui focalizzare l'analisi. Lo *slicing criterion*, per un particolare programma, definisce una sorta di finestra di osservazione del comportamento del codice stesso.

La *program slice* possiede quindi idealmente due caratteristiche fondamentali: è minimale ed eseguibile. Minimale poichè contiene il minor numero di linee di codice tali da rappresentare tutti i punti di interesse per lo *slicing criterion*, cioè non deve esistere un'altra *slice*, per lo stesso criterio (*i*; *W*), con un minor numero di istruzioni; questa proprietà è importante in quanto una *slice*, relativamente breve, focalizza l'attenzione dello sviluppatore e del tester sulle parti fondamentali dell'algoritmo, rimuovendo quelle parti che generano confusione e ridondanza.

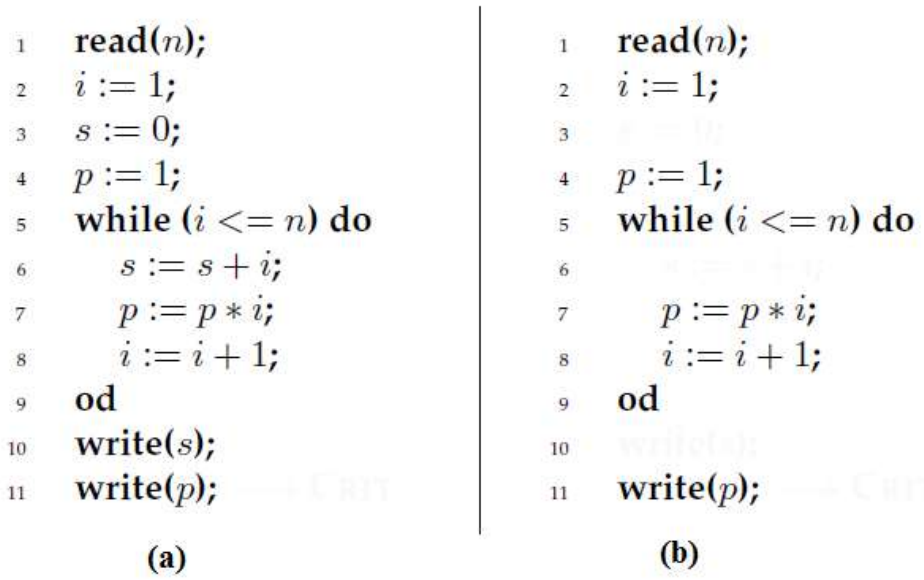


Figura 3-10. Slice statico: (a) programma d'esempio. (b) slice statico del programma.

Il frammento di codice (a) in Figura 3-10 mostra un esempio di programma che legge il valore di una variabile n, e poi calcola la somma ed il prodotto dei primi n numeri positivi. Il frammento (b) mostra invece una *slice* di questo programma rispetto al criterio (*11*, *p*); in questa figura è, infatti, possibile vedere che tutti i calcoli non rilevanti rispetto al valore finale della variabile *product* (*p*) sono stati *sliced away*. La *slice* sarà quindi costituita dalle istruzioni presenti nel programma precedente ad esclusione delle istruzioni alle righe 3, 6 e 10.

Le slice, quindi, sono ottenute calcolando insiemi consecutivi di istruzioni transitivamente rilevanti, sfruttando i *Data Dependence Graph* e *Control Dependence Graph*; in questo caso è utilizzata solo l'informazione disponibile staticamente ed è per questo che viene definito *slice statico*.

3.3.2 Dynamic Slicing

La nozione classica di *program slice* non tiene conto dei valori di input del programma; spesso però risulta particolarmente utile nel *debugging* tenere traccia del comportamento del programma nelle condizioni che hanno determinato un bug. Il problema viene affrontato proponendo la controparte dinamica del *program slicing classico*. Lo *slicing dinamico* è in grado di trovare tutte le istruzioni che hanno realmente influito sul valore di una variabile per una particolare esecuzione di un programma.

Per calcolare la *slice dinamica* si è preso spunto dall'algoritmo di Weiser per lo slicing statico. Le *slice dinamiche* sono calcolate a partire da una *execution history* (chiamata *trajectory*). La *execution history* è una lista delle istruzioni del programma registrate durante la sua esecuzione nell'ordine in cui esse sono state effettivamente eseguite. La *execution history* di un programma per un certo caso di test viene indicata con $x_1, x_2, \ldots, x_n$ in cui x_i indica la i-esima istruzione del programma. Nel caso in cui un'istruzione venga

eseguita più volte, per esempio se si trova all'interno di cicli, le diverse istanze vengono differenziate da un numero in apice.

Formalmente, un criterio di *slicing dinamico* è una coppia (*s*, *W*), dove *s* è un'istanza di istruzione di *Q*. Dato un *criterio di slicing T* ed un programma *Q*, possiamo quindi definire lo *slice S* per *Q* rispetto a *T* come l'insieme di tutte e sole le istruzioni di *Q* che hanno un qualche effetto sulle variabili osservate attraverso il criterio *T*.

```
1  read(n);
2  i:=1;
3  while (i<=n) do
4  begin
5    if (i mod 2 = 0)
6    then
7      x := 17;
8    else
9      x := 18;
10   i := i + 1;
11 end;
12 write(x);
```

(a)

```
1  read(n);
2  i:=1;
3  while (i<=n) do
4  begin
5    if (i mod 2 = 0)
6    then
7      x := 17;
8
9
10   i := i + 1;
11 end;
12 write(x);
```

(b)

Figura 3-11. Slice dinamico: (a) programma d'esempio. (b) slice dinamico del programma.

Il frammento di codice (a) in Figura 3-11 mostra un esempio di programma che legge il valore di una variabile n, e poi pone x = 17 se *i* è pari, altrimenti x = 18. Il frammento di codice (b) mostra invece una *slice* di questo programma rispetto al criterio (2, 8^1, x) dove 8^1 rappresenta la prima occorrenza nell'*execution history* del programma. Quello che si può notare per n=2 è che il loop è eseguito due volte e che entrambe le assegnazioni sono eseguite una volta.

Però per quello che riguarda lo *slicing dinamico*, il ramo ELSE dell'IF può essere omesso dato che l'assegnazione del valore 18 alla variabile X nella prima iterazione del loop è killed dall'assegnazione del valore 17 a X nella seconda iterazione. E' da notare che, nello slicing statico, non sarebbe stato possibile eliminare alcuna istruzione utilizzando il criterio (8,X).

3.4 ESECUZIONE SIMBOLICA DEL CODICE

Quando si ha a disposizione il codice sorgente di un programma, si possono utilizzare delle tecniche per generare automaticamente i casi di test. La generazione statica di casi di test si basa sull'analisi del programma senza richiedere un'esecuzione del programma stesso. La tecnica più utilizzata in questo ambito viene denominata *esecuzione simbolica*. Non si tratta di un'esecuzione vera e propria, ma indica il processo che permette l'assegnazione di vincoli e espressioni alle variabili durante l'attraversamento del codice del programma.

Questo procedimento permette di ricavare un sistema di disuguaglianze, in funzione delle variabili di ingresso, che descrive le condizioni necessarie per attraversare un dato cammino. Nella sua formulazione più generale, la soluzione di tale sistema risulta NP-completo. Tuttavia, se i vincoli sono lineari, possono essere utilizzate le tecniche di programmazione lineare [85].

L'*esecuzione simbolica* rappresenta una sintesi fra i metodi formali di controllo statico e controllo dinamico. L'esecuzione di un programma viene simulata; si rappresentano insiemi di possibili dati d'ingresso con simboli algebrici, piuttosto che con valori numerici reali. L'esecuzione delle istruzioni del programma, simulata attraverso la manipolazione delle espressioni algebriche d'ingresso, produce rappresentazioni simboliche degli stati possibili per il programma in esame. Si generano così dei modelli matematici che esplicitano l'effettivo legame funzionale fra ingressi e uscite, da confrontare con quello specificato per il programma.

Le principali difficoltà nell'applicazione dell'*esecuzione simbolica* nascono nella gestione delle chiamate a funzione (che richiederebbe di allargare l'analisi alla funzione invocata) e soprattutto la gestione di array e puntatori (perchè possono creare alias e quindi complicare notevolmente l'analisi statica del programma).

Il vantaggio sta nel non dover considerare le combinazioni possibili dei dati d'ingresso. I limiti sono invece legati agli elevati costi computazionali quando il software da controllare raggiunge dimensioni industriali.

Come esempio si consideri la seguente funzione, scritta per calcolare il valore assoluto del suo parametro:

```
public static int abs (int i){                    //L1
  int r;
  if (i<0)
    r = -i;                                       //L2
  else
    r = i;                                        //L3
                                                  //L4
  return r;
}
```

L'esecuzione simbolica simula l'esecuzione, associando a ogni stato un'espressione che ne descrive le proprietà, in funzione dei valori assunti dalle variabili. Nella tabella seguente è mostrata l'evoluzione di questa descrizione dello stato, dopo l'esecuzione dei comandi etichettati nel programma dato sopra.

Il simbolo ι (iota) rappresenta il valore iniziale della variabile `i`, ossia l'argomento del metodo. I simboli & and | indicano rispettivamente congiunzione e disgiunzione logica.

passi	stato	note
L1	$i = \iota$	è un intero
L2	$i = \iota \ \& \ (\iota < 0 \ \& \ r = -\iota)$	
L3	$i = \iota \ \& \ (0 \leq \iota \ \& \ r = \iota)$	
L4	$i = \iota \ \& \ ((\iota < 0 \ \& \ r = -\iota) \mid (0 \leq \iota \ \& \ r = \iota))$	I rami del condizionale si ricongiungono

Dal confronto dell'ultima espressione, che definisce il valore restituito dal metodo, con la specifica del metodo, si può decidere se il codice è corretto o meno.

Nel caso in cui si presentano esecuzioni simboliche con condizioni, alcuni statement sono eseguiti solo se gli input soddisfano determinate condizioni. Una *Path Condition* (*pc*), per un determinato statement, indica le condizioni che gli input devono soddisfare affinchè una esecuzione percorra un cammino lungo cui lo statement viene eseguito. Una *pc*, quindi, non è altro che un'espressione Booleana scritta in base agli input simbolici di un programma.

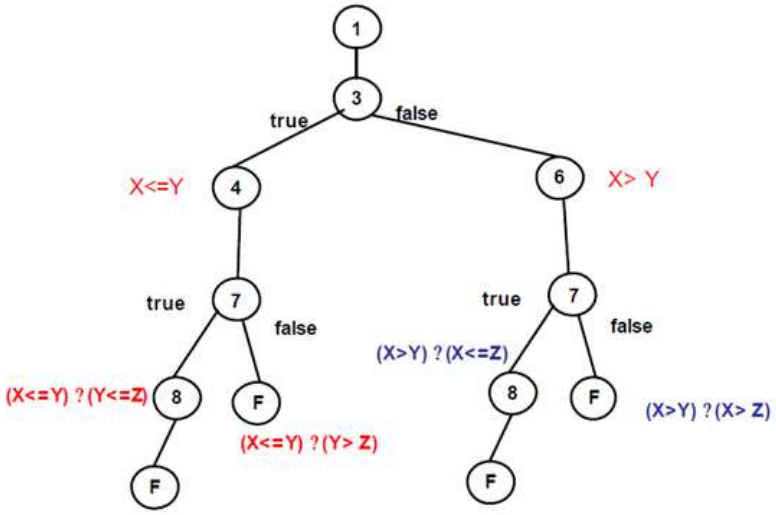
All'inizio dell'esecuzione simbolica la *pc* assume il valore vero [*pc := true*]. In seguito, per ogni condizione che si incontrerà lungo l'esecuzione, *pc* assumerà valori a seconda dei differenti casi relativi ai diversi cammini dell'esecuzione.

Dato il seguente programma di esempio, sono riportate le path condition per eseguire gli statement 4, 6, 8:

```
1. Function max (x, y, z: integer): integer;
2. Begin
3.       if x <= y then
4.           max=y;
5.       else
6.           max=x;
7.       if max <= z then
8.           max=z;
9. End
```

istruzione	pc	max	istruzione	pc	max
1	true	?	1	true	?
Case (x>y)			Case (x<=y)		
6	X>Y	X	4	X<= Y	Y
Case (max<=z)			Case (max<=z)		
8	(X>Y) ? (X<=Z)	Z	8	(X<=Y) ? (Y<=Z)	Z
Return questo valore per max			Return questo valore per max		
case (max>z)			case (max>z)		
8a	(X>Y) ? (X> Z)	X	8a	(X<=Y) ? (Y> Z)	Y
Return questo valore per max			Return questo valore per max		

Execution Tree



Ogni foglia dell'*execution tree* rappresenta un cammino che sarà percorso per certi valori di input. Le *pc* associate a due differenti foglie sono distinte; questo significa che ciascuna foglia dell'*execution tree* rappresenta un cammino che sarà percorso per la *pc* ad essa associata. Inoltre, non esistono esecuzioni per cui sono vere contemporaneamente più *pc*.

I percorsi che si possono eseguire su un *execution tree* si possono classificare in: *feasible path*, un cammino per il quale esiste un insieme di dati di ingresso che soddisfa la path condition, e *unfeasible path*, un cammino per il quale non esiste un insieme di dati di ingresso che soddisfa la path condition.

Il problema che si può presentare è quello di stabilire se esiste un punto del dominio di ingresso che rende soddisfatta la sua path condition. Per questo, si afferma che la determinazione della feasibility o della infeasibility di un cammino è indecidibile. Se si dovesse riuscire a dimostrare che ciascun predicato nella *pc* è dipendente linearmente dalle variabili di ingresso, allora il problema è risolvibile con algoritmi di programmazione lineare.

3.5 ABSTRACT INTERPRETATION

La *verifica formale* di un programma, o più in generale di un sistema informatico, consiste nel dimostrare che la sua semantica, cioè "cosa l'esecuzione del programma effettivamente fa", soddisfa le sue specifiche, che descrivono "ciò che le esecuzioni del programma sono tenute a fare".

L'*Abstract Interpretation*, formalizzata da P. Cousot e R. Cousot alla fine del 1970 in [86][87], fornisce l'idea che questa verifica formale può essere fatta ad un certo livello di astrazione in cui dettagli irrilevanti vengono ignorati. Ciò equivale a dimostrare che una semantica astratta soddisfa una specifica astratta. Essa rappresenta una teoria di approssimazione del *sound* della semantica dei programmi e può essere vista come una esecuzione parziale di un programma che acquisisce informazioni sulla sua semantica (ad esempio flusso di controllo e flusso dei dati) senza eseguire tutti calcoli. Un esempio di semantica astratta è la logica Hoare, mentre invece esempi di specifiche astratte sono l'invarianza e la correttezza parziale o totale.

Le *Astrazioni* sono rappresentate dal *sound*, quindi nessuna conclusione derivata dalla semantica astratta è sbagliata relativamente alla semantica concreta del programma e alle specifiche. Altrimenti detto, una prova che la semantica astratta soddisfa la specifica astratta implica che la semantica concreta soddisfa anche la specifica concreta. La logica di Hoare è un metodo di verifica del *sound*, non è debugging, dato che alcune esecuzioni non sono incluse, e non è model checking, dato che parti di alcune esecuzioni vengono escluse. Astrazioni deboli portano a falsi negativi, cioè il programma può affermare di essere corretto rispetto alla specifica che è scorretta. Le *Astrazioni* dovrebbero preferibilmente essere *complete*, cioè nessun aspetto della semantica, rilevante per la specifica, è lasciato fuori. Quindi, se la semantica concreta soddisfa la specifica concreta questo dovrebbe essere dimostrabile in astratto.

Le prove del programma sono indecidibili, e così gli strumenti automatici per ragionare su programmi sono tutti incompleti e devono quindi fallire su alcuni programmi. Ciò può essere conseguito consentendo allo strumento di non terminare, di essere alienato, o di essere incompleto (ad esempio strumenti di analisi statica possono produrre falsi allarmi). Astrazioni incomplete portano a falsi positivi o falsi allarmi (la specifica afferma di essere potenzialmente violata da alcune esecuzioni del programma, mentre non lo è).

Strumenti di terminazione e sound sono molto più difficili da progettare. Strumenti completi sono impossibili da progettare, a causa dell'indecidibilità. Strumenti di analisi statica che producono pochi o nessun falso allarme sono stati progettati e utilizzati in contesti industriali per specifiche famiglie di proprietà e programmi. In tutti i casi, l'*abstract interpretation* fornisce un metodo di costruzione sistematica basato sul ravvicinamento effettivo delle semantica concreta, che può essere (parzialmente) automatizzata e/o formalmente verificata.

L'*Abstract Interpretation* mira a:

- fornire una teoria di base coerente e concettuale per la comprensione in un framework unitario delle migliaia di idee, concetti, ragionamenti, metodi e strumenti per l'analisi del programma formale e la verifica;

- guidare la corretta progettazione formale di strumenti automatici per l'analisi del programma (calcolando una semantica astratta) e la verifica del programma (dimostrando che una semantica astratta soddisfa una specifica astratta).

La teoria dell'*Abstract Interpretation* studia la semantica (modelli formali di sistemi informatici), le astrazioni, la loro solidità e completezza.

L'*Abstract Interpretation* è utile in informatica per formalizzare ragionamenti che coinvolgono l'approssimazione della semantica di sistemi formali. La sua principale applicazione concreta è l'analisi statica formale, l'estrazione automatica di informazioni sulle possibili esecuzioni di programmi; tali analisi hanno due usi principali:

- all'interno di compilatori, per analizzare i programmi al fine di decidere se certe ottimizzazioni o trasformazioni sono applicabili;
- per il debugging o anche la certificazione dei programmi contro le classi di bug.

Un elenco di esempi tipici di applicazione è riportata di seguito.

- *Sintassi* - L'analisi delle proprietà delle grammatiche, come il *parsing*, sono interpretazioni astratte della semantica della grammatica operazionale e delle sue astrazioni [88].
- *Semantica* - La semantica dei programmi descrive le possibili esecuzioni runtime in tutti i possibili ambienti di esecuzione ad un certo livello di astrazione. La gerarchia della semantica può essere progettata dall'*abstract interpretation* [89][90].
- *Prove* - Prove formali di correttezza del programma implicano una astrazione da specifiche ignorando sempre alcuni aspetti dell'esecuzione del programma, che non devono essere presi in considerazione per la dimostrazione [91].
- *Analisi Statica* - L'Analisi statica è la determinazione automatica di proprietà astratte del programma [77]. Questa è l'applicazione motivante dietro l'introduzione della teoria dell'*abstract interpretation*.
- *Tipi* – La Tipizzazione statica e l'inferenza dei tipi [92] possono essere intese come interpretazione astratta di tipo runtime type checking fornendo così un metodo di progettazione "correct by construction".
- *Model-checking* - Il Model-checking verifica esaustivamente le proprietà temporali su un modello finito di sistemi informatici hardware e software. Questa astrazione di un sistema in un modello è spesso implicita. Il Model Checking astratto, come formalizzato dall'*abstract interpretation*, rende questa astrazione esplicita [93]. Dal un punto di vista dell'*abstract interpretation*, relativa al sistema per il suo modello, può essere *sound* sul modello ma *unsound* sul sistema. In tutti i casi le *abstracts interpretations* del sistema in un modello devono essere considerate. Tutti i modelli di transizione sono la semantica astratta, ma il viceversa non è vero.
- *Predicate abstraction* – Metodo che può essere utilizzato per ridurre qualsiasi analisi statica su un dominio astratto finito per computazioni di fixpoint booleano come eseguite da un model-checker utilizzando un theorem prover per ricavare automaticamente i trasformatori abstract coinvolti nella

definizione del fixpoint. La *Predicate abstraction* parametrica è una estensione di domini astratti infiniti [94].

- *Strong Preservation* - Il problema di modificare il model checking astratto al fine di ottenere una forte conservazione per qualche linguaggio di specifica, tra cui algoritmi di raffinamento della partizione, è un problema di completamento nella teoria dell'interpretazione astratta [95].
- *Program transformation* - Spesso richiedono una analisi statica del programma sorgente, come formalizzata dall'*abstract interpretation*. Inoltre, la trasformazione stessa, dal sorgente a programmi oggetto, comporta una perdita di informazioni sul programma originale o una limitazione dei possibili comportamenti di programma. Questa approssimazione può essere formalizzata dall'*abstract interpretation*. Ciò è stato semplificato dall'eliminazione di codice morto, slicing, valutazione parziale, o monitoraggio del programma [96].
- *Information hiding* - Nel software di sicurezza basato sul linguaggio, le informazioni devono essere nascoste per un intruso che può essere formalizzato come *abstract interpretation* della semantica del programma [97].
- *Code obfuscation* - L'obiettivo di offuscamento del codice è quello di evitare che gli utenti malintenzionati scoprono le proprietà del programma sorgente originale. Questo obiettivo può essere modellato con precisione dall'*abstract interpretation*, dove il nascondiglio di proprietà corrisponde ad astrarre la semantica [98].
- *Termination* - Un relazionale di analisi statica abstract-interpretation-based su un dominio astratto fondato può essere sistematicamente esteso ad una analisi di terminazione (da cui liveness analysis). L'analisi di terminazione può richiedere ipotesi probabilistiche [99] o ipotesi sulla equità di processi paralleli.

3.5.1 Argomenti di Abstract Interpretation

Abstract domains. Un dominio astratto è un algebra astratta, spesso implementata come un modulo di libreria che fornisce una descrizione delle proprietà del programma astratto e trasformatori di proprietà astratta che descrivono l'effetto operativo di istruzioni del programma e comandi in astratto. I domini astratti sono spesso reticoli completi, un'astrazione di powerset.

Domini astratti Finiti vs Infiniti. Il primo dominio astratto infinito è stato introdotto in [100]. Questo dominio astratto è stato poi utilizzato per dimostrare che, grazie ad ampliamenti, i domini astratti infiniti possono portare ad analisi statiche efficaci per un dato linguaggio di programmazione che sono strettamente più precise e altrettanto efficienti di qualsiasi altra utilizzando un dominio astratto finito.

Merge over all path (MOP) in analisi dataflow. La prima analisi dataflow in cui si è dimostrata corretta confrontando l'unione della astrazione lungo tutti i percorsi di esecuzione con una soluzione di fixpoint, usando funzioni di trasferimento (cioè trasformatore di proprietà astratto). Invece di confrontare una soluzione del problema di analisi statica con un altro, la teoria dell'*abstract interpretation* ha introdotto l'idea che la correttezza deve essere espressa con riferimento a una semantica formale. Una conseguenza è che gli algoritmi di analisi statica possono essere specificati

e progettati per approssimazione di una semantica del programma. Inoltre, la "fixpoint solution" ha mostrato di essere una astrazione della "soluzione MOP". Queste due soluzioni coincidono per interpretazioni astratte distributive (per i quali i trasformatori astratti preservano l'unione). Altrimenti, il dominio astratto utilizzato per la fixpoint solution può essere minimamente arricchita nel suo completamento disgiuntivo per ottenere esattamente la "MOP solution" in forma fixpoint (sul dominio astratto completato).

Galois connection. *Galois connection* può essere utilizzata quando il dominio astratto offre sempre una approssimazione più precisa di qualsiasi proprietà concreta. In effetti le Galois connections introdotte in [101] sono il semi-duale di quelle introdotte da Évariste Galois, e in modo da farne una composizione. Presentazioni equivalenti coinvolgono operatori di chiusura, ideali, congruenze, ecc. Una conseguenza interessante della esistenza della migliore astrazione delle proprietà concrete è l'esistenza di trasformatori più precisi, che forniscono le linee guida per il loro progetto automatico. Inoltre l'analisi statica è specificata da una semantica astratta espressa in forma di fixpoint, mentre un analizzatore statico è un algoritmo per il calcolo o sovrapprossimazione di questo fixpoint.

Moore family, operatori di chiusura. Se la corrispondenza tra le proprietà concrete e astratte è data da una Galois connection, allora l'immagine delle proprietà astratte nel concreto è una Moore family, che è l'immagine delle proprietà concrete da un operatore di chiusura. La formalizzazione dell'astrazione da operatori di chiusura è utile per astrarre dalla rappresentazione delle proprietà concrete. Altre formalizzazioni equivalenti dell'esistenza dell'astrazione migliore delle proprietà concrete comprendono principali ideali, complete congruenze unite, etc.

Moore completion. Se un dominio astratto non ha alcuna miglior approssimazione per alcune proprietà concrete il suo Moore completion è il dominio astratto più espressivo del dominio astratto originale che ha questa proprietà. Questo introduce una Galois connection.

Fixpoints (Punti noti). La maggior parte delle proprietà del programma possono essere espresse come fixpoints di proprietà monotone o estese dei trasformatori, una proprietà preservata da astrazione. Questo riduce l'analisi del programma all'approssimazione e alla verifica del fixpoint per il fixpoint check.

Le regole di inferenza. La maggior parte (concrete e astratte) delle proprietà del programma possono essere espresse utilizzando regole di inferenza, che anzi è equivalente a definizioni mediante fixpoint, equazioni, vincoli, ecc. Questo punto di vista ha il vantaggio di separare il progetto di una (abstract) semantica (o un'analisi statica) dalla sua presentazione formale.

Iterazione. Un metodo standard per la costruzione di fixpoint è l'*iterazione*. Questa può essere estesa a iterazioni transfinite per dimostrare il teorema del fixpoint di Tarski [103]. In pratica si può accelerare il calcolo del fixpoint utilizzando adeguate strategie di iterazione formalizzate come iterazioni caotiche/asincrone.

Combinazione di dominio astratta. La progettazione di astrazioni può essere fatta per parti, scegliendo astrazioni di base e poi componendo loro utilizzando combinatori di domini astratti. Questo fornisce una visione unificante sul progetto di dominio astratto. Esistono numerosi esempi di tali combinazioni di dominio astratto compresa la somma ridotta, il prodotto ridotto e la potenza ridotta [102].

Affinamento. L'arricchimento di un dominio astratto così deve essere espresso nelle proprietà di dominio raffinato che non possono essere espresse nel dominio originale.

Completamento. La rifinitura di un dominio astratto in un dominio più astratto che è abbastanza preciso per dimostrare una data specifica.

Reticolo di interpretazioni astratte. Una conseguenza della formalizzazione delle analisi statiche dalle connessioni di Galois è che possono essere organizzate in un reticolo completo secondo l'ordine parziale delle loro relative precisioni. Tutta la semantica e analisi sono presenti in questo reticolo formalmente definito.

3.6 THEOREM PROOF E SOFTWARE MODEL CHECKING

Una verifica si definisce *formale* se la proprietà di correttezza è una proposizione matematica precisa. Pertanto, formalmente, "verificare" corrisponde a decidere se sussiste o meno la relazione di soddisfacimento tra un modello e una formula:

$$M \models F$$

dove:

- *M* è un modello che descrive i possibili comportamenti del sistema.
- *F* è la proprietà o specifica che il sistema deve soddisfare.
- $\models$ è una relazione che deve sussistere tra *M* e *F*.

Ciò che permette di decidere se la relazione vale o meno è il *ragionamento logico*. Applicando tale ragionamento allo sviluppo di sistemi computerizzati è possibile parlare di "verifica automatica". L'obiettivo del *metodi formali* è quindi quello di stabilire la correttezza di un sistema con rigore matematico, esprimendo le specifiche in maniera precisa e definendo in modo chiaro quando una implementazione le soddisfa. Il loro potenziale ha portato ad un uso sempre crescente di questi metodi nell'ambito dei sistemi complessi.

I principali approcci alla *verifica formale* sono il *Theorem Proving* ed il *Model Checking*.

3.6.1 THEOREM PROVING

Il *Theorem Proving* è una verifica deduttiva che usa assiomi e regole specifiche per dimostrare la correttezza di un sistema. La proprietà da verificare viene espressa attraverso un teorema che viene quindi "provato" con l'aiuto di un dimostratore automatico (*theorem prover*). Con il *Theorem Proving* è possibile effettuare verifiche anche su sistemi a stati infiniti. Questa tecnica tratta lo sviluppo di software che mostra quale dichiarazione (o congettura) è una conseguenza logica di un insieme di istruzioni (assiomi e ipotesi).

La lingua in cui la congettura, le ipotesi, e gli assiomi (genericamente conosciuti come formule) sono scritti è una logica, spesso la logica classica del primo ordine, ma una logica non classica e, eventualmente, una logica di ordine superiore. Questi linguaggi consentono una dichiarazione formale precisa delle informazioni necessarie, che possono poi essere manipolate attraverso il *Theorem Proof*.

La formalità è la forza sottostante di questo approccio: non c'è ambiguità nella dichiarazione del problema, come spesso accade quando si utilizza un linguaggio naturale come l'inglese. Gli utenti devono descrivere a mano con precisione il problema, e questo processo di per sé può portare ad una più chiara comprensione del dominio del problema. Le prove (o dimostrazioni) prodotte descrivono come e perché la congettura segue assiomi e ipotesi, in modo che possono essere comprese e condivise da tutti.

Il *Theorem Proof* tratta software enormemente potenti, in grado di risolvere problemi estremamente difficili. A causa di questa capacità estrema, la loro applicazione e funzionamento talvolta hanno bisogno di

essere guidate da un esperto nel dominio di applicazione, al fine di risolvere i problemi in un tempo ragionevole.

Questi sistemi sono spesso utilizzati da esperti di dominio in modo interattivo. L'interazione può essere ad un livello molto dettagliato, in cui l'utente guida le inferenze effettuate dal sistema, o ad un livello molto più alto, dove l'utente determina lemmi intermedi da provare per dimostrare una congettura. C'è spesso un rapporto sinergico tra gli utenti e i sistemi stessi.

In genere, il sistema ha bisogno di una descrizione precisa del problema, scritto in una forma logica, per questo l'utente è costretto a riflettere attentamente sul problema al fine di produrre una formulazione appropriata, e quindi acquisisce una più profonda comprensione del problema.

Il sistema tenta di risolvere il problema, ma in caso di successo la prova (la dimostrazione) è un uscita utile, mentre in caso di insuccesso, l'utente può fornire una guida, o cercare di provare qualche risultato intermedio, o esaminare le formule per assicurare che il problema è descritto correttamente; in questo modo, quindi, il processo itera.

Questo approccio presenta degli svantaggi:

- È molto difficile da automatizzare (a volte impossibile).
- Richiede un continuo intervento da parte dell'utente.
- Necessita di molto tempo.
- Richiede un esperto per usare il *theorem prover*.

3.6.2 Software Model Checking

Il *Model Checking* è una tecnica per verificare la correttezza di sistemi concorrenti *a stati finiti*. Tale restrizione permette di svolgere la verifica in modo completamente automatico. Al contrario del testing, il *model checking*, garantisce la completa esplorazione dello spazio degli stati e, a differenza del *theorem proving*, non richiede un frequente intervento umano nella fase di verifica.

Ovviamente, affinché il risultato sia attendibile, è necessario che il modello del sistema non contenga errori e che la specifica sia consistente e correttamente formalizzata. Superati questi ostacoli, il problema della verifica si riduce semplicemente a un problema di ricerca sui grafi.

Il processo del *model checking* può essere suddiviso in tre fasi:

- *Modelling*
- *Specification*
- *Verification*

La fase di *Modelling* consiste nella costruzione di un modello astratto che rappresenti il sistema, trascurando tutti i dettagli non significativi al fine della verifica. Il livello di dettaglio dovrà essere sufficiente a cogliere solo gli aspetti rilevanti in modo da non rendere più complesso del necessario il processo di verifica finale. Per realizzare tale modello sono solitamente usati gli *automi a stati finiti*.

Vari tipi di automi a stati finiti, come *sistemi di transizione etichettati* (LTS), *strutture di Kripke* o *diagrammi di stato* UML sono ampiamente accettati come notazione formale chiara, semplice e sufficientemente astratta. Il *model checking*, per modellare i sistemi concorrenti, fa tipicamente uso di *LTS* e *strutture di Kripke*. In entrambi i casi si tratta di grafi orientati in cui i nodi rappresentano i possibili stati del sistema, definiti dai valori assunti dalle variabili di stato del sistema stesso, e gli archi rappresentano le transizioni da uno stato al suo successore.

A differenza degli LTS, dove gli archi vengono etichettati con le *azioni,* in una struttura di Kripke i nodi sono etichettati tramite *proposizioni atomiche (AP)*.

Una *struttura di Kripke M* è una tupla $M = (S, In, R, L, AP)$ dove:

- S è un insieme finito non vuoto detto *insieme degli stati di M.*
- $In \subseteq S$ è un insieme non vuoto di stati detto *insieme degli stati iniziali di M.*
- $R \subseteq S \times S$ è una *relazione di transizione totale*, ovvero, per ogni stato $s_0 \in S$ esiste uno stato $s_1 \in S$ tale che $R(s_0, s_1)$.
- $L : S \rightarrow 2^{AP}$ è una *funzione di labeling* che etichetta ogni stato con l'insieme AP delle proposizioni atomiche che sono vere in quello stato.

Sia $V = \{v_1, v_2, \ldots, v_n\}$ l'insieme delle variabili del sistema considerato. Uno stato del sistema s è dato dalla valutazione delle variabili v in s, ovvero, uno stato è una valutazione $s : V \rightarrow D$, dove D è il dominio dei valori che possono essere assunti dalle variabili.

La seconda fase di *Specification*, consiste nell'esprimere le proprietà che il sistema dovrebbe soddisfare durante il suo evolversi nel tempo. Ciò può essere fatto per mezzo di un formalismo logico. È molto comune l'uso della *logica temporale*. Essendo tale logica un'estensione della *logica proposizionale*, ci soffermeremo ad introdurne i concetti base.

La *logica proposizionale* è un linguaggio formale con una struttura sintattica semplice, basata fondamentalmente su proposizioni elementari e su connettivi logici. Ogni proposizione si riferisce a uno o più oggetti della realtà rappresentata e permette di descrivere o ragionare su quell'oggetto, utilizzando i due soli valori "Vero" e "Falso".

La *sintassi* si basa su due componenti principali ovvero un *alfabeto* e una *grammatica*. L'*alfabeto*, che definisce quali simboli possono essere usati all'interno del linguaggio, è costituito da:

- Un insieme numerabile di lettere, o simboli, proposizionali p, q, r, … dette *variabili atomiche*, *atomi proposizionali* o più semplicemente *atomi*.
- Un insieme di *connettivi* logici
- { ¬ (NOT), Λ (AND), V (OR), → (implicazione), ↔ (doppia implicazione) }.
- Un insieme di *simboli ausiliari*, ad esempio le parentesi "{}" e "()", che hanno per lo più lo scopo di rendere il linguaggio più chiaro ed evitare ambiguità.

La *grammatica* descrive come i simboli dell'alfabeto possono essere combinati per ottenere espressioni "sintatticamente corrette". Tali espressioni sono chiamate "*formule ben formate*" (*fbf*, oppure dall'inglese *wff*, "*well formed formulas*") e sono definite dalle seguenti regole:

1. Ogni simbolo di proposizione è una formula.
2. Se A è una formula lo è anche ¬A.
3. Se A e B sono formule, allora lo sono anche (A ∧ B), (A ∨ B), (A→B), (A↔B).
4. Nient'altro è una formula.

La *semantica* della logica proposizionale è definita tramite la nozione di *interpretazione*. Un'interpretazione è una funzione, detta anche *funzione di valutazione*, che assegna *valori di verità* ad ogni membro di un insieme di atomi proposizionali. Una volta definita tale funzione, sulla base dei valori assegnati e delle regole, è possibile stabilire il valore di verità di una qualsiasi formula proposizionale.

Un *valore di verità* è un elemento dell'insieme *D := { True, False }*.

Si definisce, invece, *funzione di valutazione* "*v*" una funzione che va dall'insieme *L* (delle formule ben formate) nell'insieme *D := { True, False }* (*v : L → D*) tale che per ogni coppia di *fbf* P e Q valgano le seguenti condizioni:

- *v*(¬P) = *true* se e solo se *v*(P) = *false* (e viceversa)
- *v*(P ∧ Q) = *true* se e solo se *v*(P) = *true* **e** *v*(Q) = *true*
- *v*(P ∨ Q) = *true* se e solo se *v*(P) = *true* **o** *v*(Q) = *true*
- *v*(P → Q) = *true* se e solo se *v*(P) = *false* **o** *v*(Q) = *true*
- *v*(P ↔ Q) = *true* se e solo se *v*(P) = *v*(Q)

Queste condizioni rispecchiano il significato che si vuole attribuire ai simboli associati ai connettivi logici e si possono riassumere mediante la seguente tabella di verità:

P	Q	¬P	P ∧ Q	P ∨ Q	P → Q	P ↔ Q
F	F	V	F	F	V	V
F	V	F	F	V	F	V
V	F	V	F	V	F	F
V	V	F	V	V	V	V

Torniamo ora alla *logica temporale*, che rappresenta il linguaggio formale maggiormente usato per esprimere le relazioni tra gli stati della struttura di Kripke. Le logiche temporali, introdotte nel *model*

checking, permettono di descrivere come evolvono gli stati senza menzionare esplicitamente il tempo, ma facendo riferimento alla sequenza con cui essi si presentano.

Una computazione del tipo $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ è l'oggetto di base del ragionamento. Partendo da uno stato s_0 ($s_0 \in In$) e collegando ad esso altri stati raggiungibili attraverso le relazioni R otteniamo una rappresentazione grafica dei comportamenti del sistema.

In questo caso, si possono individuare due famiglie principali: le *Logiche Temporali Lineari* (LTL) e le *Logiche Temporali Ramificate* (CTL).

Con la *logica temporale lineare* si possono descrivere le proprietà di tutti i possibili percorsi individuali che hanno origine dallo stato iniziale s_0, mentre in quella *ramificata* interessa analizzare l'*albero delle computazioni* a partire dal medesimo stato di partenza. Quindi, essendo obiettivo del *model checking* quello di stabilire se sussiste o meno la relazione $M \models \varphi$ (dove M è un sistema a transizioni e φ una formula), l'approccio lineare (LTL) consiste nel controllare che tutte le sequenze lineari che possono essere generate da M soddisfino la formula φ. Questo approccio viene naturale quando le proprietà da controllare sono espresse in termini di possibili esecuzioni del programma. Oltre alle variabili atomiche e ai connettivi definiti nella generica logica proposizionale, alla sintassi della logica temporale bisogna aggiungere degli *operatori* (detti appunto "*temporali*").

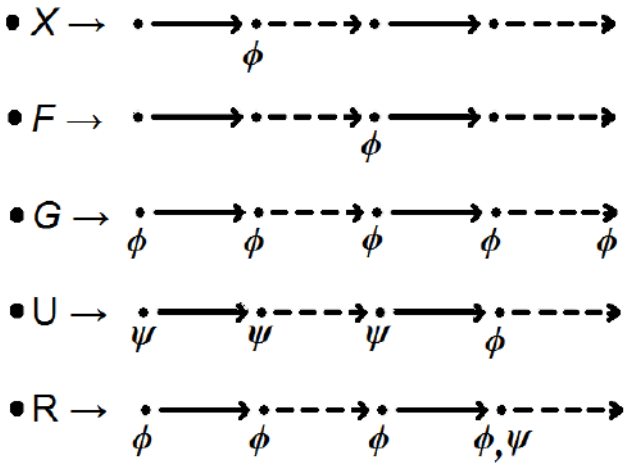
Siano ϕ e ψ due formule, si definiscono *operatori unari*:

- $X\,\phi$ (ne**x**t), richiede che ϕ valga nello stato immediatamente successivo al corrente.
- $F\,\phi$ (in the **f**uture), richiede che ϕ valga in un qualsiasi stato futuro del percorso considerato.
- $G\,\phi$ (**g**lobally), richiede che ϕ valga in tutti gli stati del percorso considerato.

E operatori binari:

- ψ U ϕ (**u**ntil), richiede che ϕ valga in qualche stato del percorso e che ψ sia verificata in ogni stato precedente.
- ψ R ϕ (**r**elease), richiede che ϕ valga fino allo stato in cui vale ψ per la prima volta. Ovviamente ψ potrebbe non essere mai verificata, implicando che ϕ valga per sempre.

Possiamo chiarire il concetto degli operatori temporali nella *logica LTL* osservando per ognuno di essi il diagramma esplicativo:

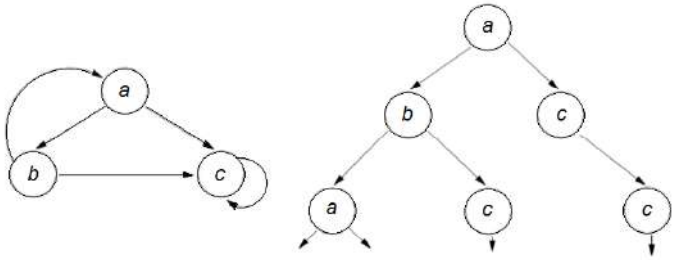


Un esempio di formula LTL potrebbe quindi essere:

$$G\,(\,p_1 \rightarrow X\,(\,\neg p_2\ U\ p_1\,)\,)$$

Ovvero: “In tutti gli stati del percorso considerato (*G*) il verificarsi di p_1 implica (→) che a partire dallo stato immediatamente successivo (*X*) p_2 non sia verificato (¬) finché (*U*) non sussiste di nuovo p_1”. Un tipico esempio di logica ramificata è invece la *logica CTL* (Computation Tree Logic). Essa permette di descrivere proprietà temporali riferendosi ad un *albero di computazione*, ossia ad un albero ottenuto “srotolando", a partire da uno stato scelto come radice, la struttura di Kripke del sistema in oggetto.

In pratica, l'albero di computazione rappresenta tutte le possibili evoluzioni del sistema a partire dallo stato scelto come iniziale:



Per effettuare questo tipo di descrizione, oltre che di *operatori temporali*, le formule CTL fanno uso di *quantificatori di percorso* che vengono usati, appunto, per associare le proprietà a tutti o solo ad alcuni dei percorsi che partono dal nodo considerato.

I quantificatori di percorso sono:

- **A** (per ogni percorso).
- **E** (esiste almeno un percorso) .

Un esempio di formula CTL potrebbe essere:

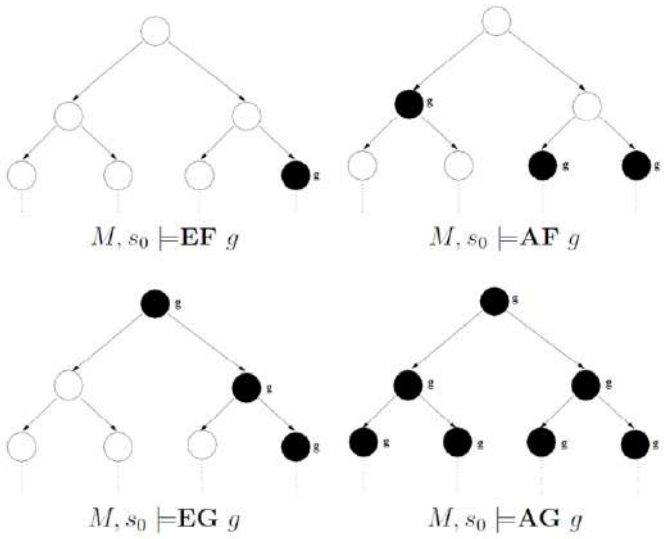
$$AG\,(\,\mathrm{p}_1 \rightarrow AX\,EF\,\mathrm{p}_2\,)$$

Ovvero: "In tutti gli stati di ogni percorso considerabile (*AG*) il verificarsi di p_1 implica (→) che per ogni percorso a partire dallo stato immediatamente successivo (*AX*) ne esiste almeno uno in cui, in un qualsiasi stato futuro (*EF*), sia verificato p_2".

Possiamo distinguere due tipi di formule:

1. Formule di stato
2. Formule di percorso

È possibile definire la semantica della logica temporale CTL facendo riferimento ad una generica struttura di Kripke *M = (S, In, R, L, AP).* Un percorso π in *M* è una sequenza di stati $s_0, s_1, s_2, s_3, s_4, \ldots$ tale che, per ogni $i \geq 0$, $(s_i, s_{i+1}) \in R$. Se *f* è una formula di stato, con $(M, s \models f)$ intendiamo dire che *f* è vera nello stato *s* della struttura M. Analogamente, se *g* è una formula di percorso, $(M,\pi \models g)$ indica che *g* è vera lungo il percorso π della struttura *M.* Utilizziamo anche in questo caso degli esempi grafici per comprendere meglio il significato di alcuni operatori temporali CTL:



Infine, la fase di ***Verification*** serve a constatare la correttezza del modello del sistema rispetto alle proprietà specificate. Questa fase è totalmente automatizzata e richiede l'intervento umano solo per l'analisi del risultato.

Nel caso in cui il sistema non soddisfi le specifiche, il *model checking* mostra la sequenza di stati che porta a violare una determinata proprietà. Tale sequenza è detta traccia di esecuzione (o *controesempio*). Dall'analisi della traccia si possono ricavare informazioni utili per apportare le necessarie correzioni al sistema in oggetto. Può accadere, però, che il processo termini prima di aver completato la verifica a causa di insufficienza di risorse. Infatti, quando il sistema è composto da più processi, il numero degli stati del sistema complessivo cresce in modo esponenziale con il numero dei processi coinvolti, perché lo spazio degli stati è dato dal prodotto cartesiano degli spazi di ogni singolo processo. Questo fenomeno, chiamato

"*State-Space Explosion*" (esplosione dello spazio degli stati), limita fortemente la capacità degli algoritmi di lavorare su esempi di sistemi di grosse dimensioni.

Una soluziona a tale problema è data dal *Model Checking Simbolico*. Questa tecnica consiste nel rappresentare lo spazio degli stati implicitamente (o *simbolicamente*) mediante un insieme di funzioni binarie, a loro volta rappresentate tramite speciali strutture dati chiamate *Binary Decision Diagrams* (*BDD*). L'idea alla base di questo approccio simbolico è quella di operare su dei gruppi piuttosto che sui singoli stati e sulle singole transizioni. L'identificazione dei sottoinsiemi dello spazio avviene in maniera semplice grazie alla codifica binaria dei singoli stati.

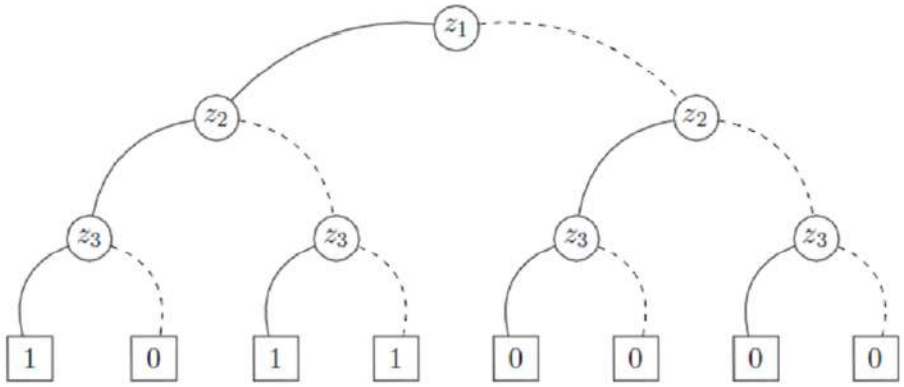
I *Binary Decision Diagrams* sono una versione compatta dei *Binary Decision Trees* (*BDT*). Essi si basano sull'idea di eliminare da essi i rami ridondanti. Un *BDT* è un albero binario in cui ogni nodo è associato ad una variabile ed ogni arco uscente è associato ad un valore di verità. Per determinare il valore di verità della formula rappresentata è sufficiente attraversare l'albero a partire dalla radice: dopo aver visitato un nodo, si percorre l'arco corrispondente al valore assegnato alla variabile associata ad esso, ripetendo l'operazione fino ad incontrare un nodo foglia. Ad esempio, siano z_1, z_2, z_3 delle variabili booleane. Immaginiamo di voler rappresentare la funzione:

$$f(z_1, z_2, z_3) = z_1 \wedge (\neg z_2 \vee z_3)$$

La sua tabella di verità sarà data dai seguenti valori:

$z_1 = 0 , z_2 = 0 , z_3 = 0 \rightarrow f = 0$
$z_1 = 0 , z_2 = 0 , z_3 = 1 \rightarrow f = 0$
$z_1 = 0 , z_2 = 1 , z_3 = 0 \rightarrow f = 0$
$z_1 = 0 , z_2 = 1 , z_3 = 1 \rightarrow f = 0$
$z_1 = 1 , z_2 = 0 , z_3 = 0 \rightarrow f = 1$
$z_1 = 1 , z_2 = 0 , z_3 = 1 \rightarrow f = 1$
$z_1 = 1 , z_2 = 1 , z_3 = 0 \rightarrow f = 0$
$z_1 = 1 , z_2 = 1 , z_3 = 1 \rightarrow f = 1$

Quindi il *BDT* corrispondente sarà:

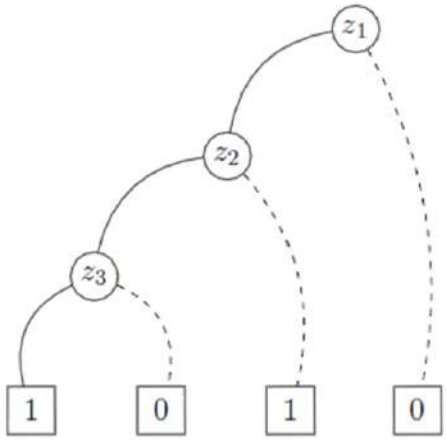


dove le linee continue rappresentano il caso $z = 1$, mentre quelle tratteggiate $z = 0$.

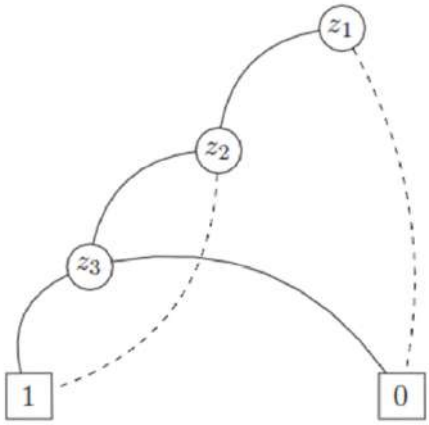
Un *BDD* tenta di eliminare le ridondanze presenti nel BDT seguendo due regole:

1. Eliminare i nodi i cui sotto-alberi sono isomorfi
2. Combinare sotto-alberi isomorfi in un unico albero

Nel nostro caso possiamo osservare che tutti i percorsi che partono dalla destra di z_1 portano allo stesso risultato. Inoltre, anche nel ramo sinistro c'è una ridondanza. Infatti alla destra di z_2, qualunque sia il valore di z_3, otteniamo sempre lo stesso risultato. Quindi, utilizzando la prima delle due regole appena citate, il nostro grafico diventa:



Applicando infine la seconda regola otteniamo:



Un *BDD* è detto "*ordinato*" (*OBDD*, *Ordered Binary Decision Diagram*) se più variabili compaiono nello stesso ordine in tutti i percorsi a partire dalla radice. Il nostro esempio rappresenta quindi un *OBDD* con il seguente ordinamento di variabili: *z1 < z2 < z3*.

In conclusione possiamo affermare che con i *BDD*, scegliendo il giusto ordinamento, si possono ottenere grosse riduzioni dello spazio degli stati, arrivando al punto di poter trattare sistemi fino a 10^{120} stati.

L'utilizzo della rappresentazione simbolica ha permesso che il *model checking* fosse applicato a sistemi di dimensioni maggiori rispetto a quanto poteva essere fatto utilizzando una rappresentazione esplicita.

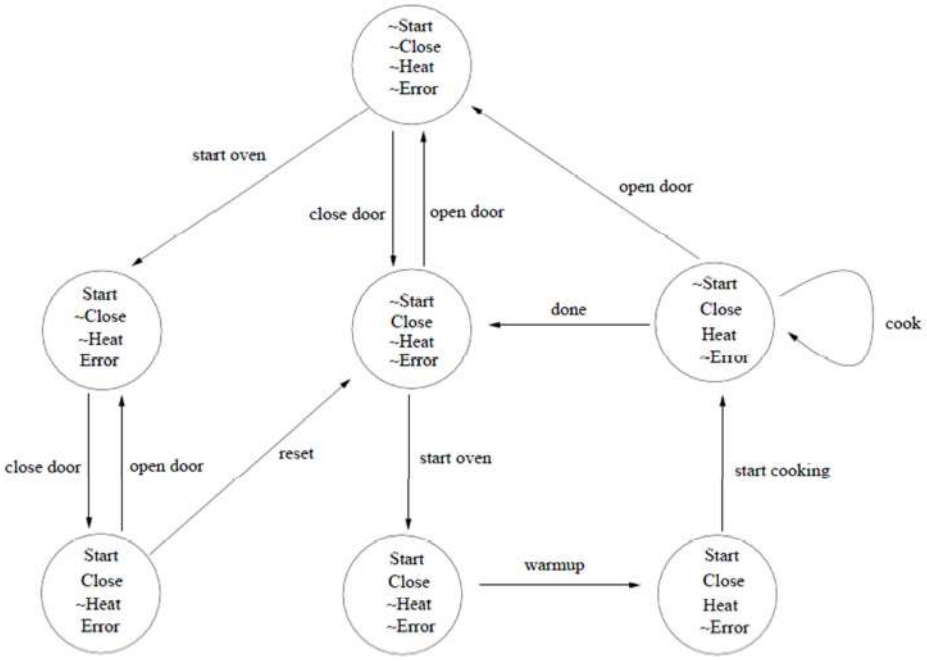
Quando il sistema da verificare è un componente hardware, per quanto possa essere grande, l'ipotesi di finitezza dello spazio degli stati risulta essere sempre soddisfatta grazie agli aspetti fisici del sistema. A volte, però, in particolare nei sistemi software, il numero degli stati può addirittura risultare infinito.

Con il passare del tempo, varie tecniche sono state studiate al fine di modellare sistemi a stati infiniti e ridurre il fenomeno dell'esplosione degli stati. Una possibilità è quella di generare e memorizzare solo lo spazio degli stati necessario per verificare la proprietà in questione. Due esempi di tecniche che si basano su questo principio sono il *Local Model Checking* e il *Bounded Model Checking* (*BMC*).

Nel *Local Model Checking* (noto anche come *on-the-fly model checking*) si generano, sulla base della struttura della formula, solo gli stati che effettivamente servono a verificare la formula. Nel *BMC*, invece, lo spazio degli stati viene generato fino ad una profondità fissata *k* e il risultato sarà quindi ternario: la formula è verificata, la formula non è verificata, la profondità non è sufficiente per verificare. Per questa ragione il *Bounded Model Checking* risulta più versatile per la ricerca degli errori piuttosto che per la verifica. Infatti, considerando percorsi di lunghezza *k*, non è possibile affermare nulla su quel che può accadere nelle successive evoluzioni del sistema.

Un'altra tecnica utilizzata per ridurre in maniera significativa il numero degli stati di un sistema è chiamata "*Partial Order Reduction*". Si basa su un'osservazione ben precisa: in caso di eventi indipendenti il concetto di ordinamento è privo di significato.

Analizziamo ora un semplice esempio di ricapitolazione che mostra le tre fasi del *model checking*. Immaginiamo di avere la seguente specifica informale sul funzionamento di un forno a microonde: "Per cuocere del cibo bisogna aprire lo sportello, inserire il cibo all'interno, chiudere lo sportello e premere il pulsante di avvio. Il forno si riscalderà per circa 30 secondi, dopodiché inizierà la cottura e a cottura ultimata si spegnerà automaticamente. Nel caso in cui venisse aperto lo sportello durante la cottura, il forno si bloccherà immediatamente e, analogamente, se di dovesse premere il pulsante di avvio mentre lo sportello è aperto, si verificherà un errore e il forno non avvierà la cottura. In quest'ultimo caso sarà necessario premere un pulsante di reset prima di un nuovo utilizzo". Possiamo individuare quattro proprietà principali nel sistema, che chiameremo "*Start*", "*Close*", "*Heat*", "*Error*". Etichettando ogni stato con l'insieme delle proposizioni atomiche (quelle false sono indicate con il simbolo "~"), otteniamo, al termine della fase di *modelling*, il seguente grafico:



Immaginiamo quindi, in seguito alla formalizzazione della specifica (***specification***), di ottenere delle formule in logica CTL che saranno poi oggetto dell'ultima fase, ovvero quella di verifica (***verification***). Prendiamone in considerazione due:

1. *AG* (Heat → Close). Il checker darà come positivo l'esito del controllo. Infatti, traducendo in linguaggio naturale la formula, essa indica che "ogni qual volta il forno si trovi in fase di cottura, lo sportello sarà chiuso".
2. *AG* (Start → *AF* Heat). Il checker questa volta darà esito negativo. La formula, infatti, indica che "il forno avvierà la cottura ogni qual volta verrà premuto il pulsante di avvio" e ciò non è conforme con il modello (né tantomeno con la specifica). Se la specifica fosse stata formalizzata in maniera corretta avremmo avuto *AG* ((Start Λ ¬Error) → *AF* Heat) ovvero "il forno avvierà la cottura ogni qual volta verrà premuto il pulsante di avvio *senza che questo generi errori*".

Applicare manualmente le tecniche di verifica formale a software e hardware di scala industriale è di fatto impraticabile. Le nuove generazioni di tecniche necessitano fortemente di ragionatori automatici e il cuore di tali ragionatori è rappresentato dalle *procedure di decisione*. Le *procedure di decisione* sono algoritmi in grado di ragionare sulla validità o soddisfacibilità di classi di formule e sono alla base di tutti i sistemi di analisi e verifica dei programmi, rendendoli più efficienti ed evitando continue interazioni tra il sistema e l'utente. Il successo del *model checking* è dovuto in larga parte agli sviluppi nel ragionamento sulla logica booleana (come per gli OBDD) ma l'analisi del software e, più in generale, di sistemi di grandi dimensioni o a stati infiniti, richiede l'uso di procedure di decisione ancora più efficaci ed espressive. Tali procedure sono, in generale, molto efficienti, ma risultano anche molto specializzate. Durante la verifica dei programmi le condizioni da verificare riguardano, però, combinazioni più o meno complesse di più domini, facendo così nascere l'esigenza di combinare le singole procedure al fine di ottenerne di nuove che abbiano migliori prestazioni rispetto a quelle di ogni singola tecnica o singolo algoritmo che li costituisce.

Mentre nella produzione dell'hardware il *model checking*, una volta superate le limitazioni di dimensione, ha trovato facile presa tra gli addetti, nel caso della produzione software la corrispondenza tra codice e modello non è così immediata. Quindi, fatta eccezione per i sistemi critici, in quest'ambito il testing viene ancora visto come la principale attività di verifica di correttezza del codice, pur non possedendo le doti di esaustività che costituiscono il principale vantaggio del *model checking*. Nella produzione software possiamo individuare due tendenze principali: la prima, che possiamo considerare più matura e diffusa, è situata all'interno dei metodi di sviluppo basati sulla modellazione formale dei requisiti (*Model Based Verification*); la seconda, ancora in via di sviluppo, è quella dell'applicazione diretta delle tecniche di verifica al codice prodotto (*Software Model Checking*). Nel primo approccio lo sviluppo del software avviene a partire dal modello del sistema, attraverso passi di raffinamento e traduzione. La definizione di modelli accurati, prima dello sviluppo del codice, ha effetti positivi sulla possibilità di rilevare errori in fase di progettazione, quando la loro correzione ha un costo molto minore. Inoltre può anche servire in maniera diretta per generare automaticamente il codice tramite appositi strumenti.

In caso di *Software Model Checking* l'approccio è completamente opposto: si parte dal codice, in qualunque modo esso sia stato sviluppato, e se ne ricava lo spazio degli stati.

Essenziale è in questo caso l'adozione di potenti tecniche di astrazione, che siano in grado di ridurre sostanzialmente la dimensione dello spazio degli stati, senza incidere sulla soddisfacibilità o meno delle proprietà da provare.

In conclusione, possiamo affermare che il *Model Checking* è decisamente più efficace del *testing* nel trovare errori. Negli ultimi vent'anni il *Model Checking* si è, infatti, imposto come la principale tecnica in grado di verificare la correttezza di un sistema. Il problema dello spazio degli stati e le difficoltà che emergono in fase di formalizzazione della specifica rappresentano però un ostacolo alla sua piena diffusione, specialmente in determinati settori della produzione software. L'attività di ricerca su questa tecnica è tuttora molto attiva e promette un suo più vasto utilizzo in vari ambiti dello sviluppo di sistemi informatici. Tra le direzioni di ricerca più seguite negli ultimi anni si possono citare quella sui sistemi *realtime*, in cui le proprietà di interesse quantificano con esattezza gli intervalli di tempo in cui azioni e reazioni del sistema debbano avvenire, e quella del *model checking probabilistico*, dove si vuole valutare quale sia la probabilità che un modello goda di una certa proprietà, aprendo così la strada all'utilizzo del *model checking* nella valutazione *quantitativa* (oltre che qualitativa) delle caratteristiche di un sistema.

3.7 TESTING

Il Testing (collaudo) è un processo di esecuzione del software allo scopo di scoprirne i malfunzionamenti. Il processo di testing è centrato sulla specifica e l'esecuzione di un insieme di casi di test (test case). Un ***Test Case*** è un insieme di input, condizioni di esecuzione, ed un criterio di pass/fail, mentre un ***Test Case Specification*** (specifica dei casi di test) è un requisito che deve essere soddisfatto da uno più casi di test. Una specifica dei casi di test parziale, che richiede qualche proprietà ritenuta importante per il test completo, è denominata ***Test obligation***. L'insieme dei casi di test è raggruppato in ciò che è comunemente nota come ***Test suite***. Una test suite può essere fatta da diversi insiemi per singoli moduli, sottosistemi o caratteristiche.

Allo scopo di valutare secondo il criterio pass/fail se il comportamento osservato coincide con quello atteso, è necessario definire un ***oracolo***, ossia una descrizione del comportamento atteso, che si applica ad una esecuzione del programma. Infine, la bontà dell'esecuzione di una test suite è misurata, in maniera aggregata, attraverso il soddisfacimento di uno o più ***criteri di adeguatezza***. Quest'ultimo è un predicato su una coppia <programma, test suite>, di solito espresso nella forma di una regola per derivare un insieme di *test obligation*. Il criterio è soddisfatto se ogni *test obligation* è soddisfatta da almeno un caso di test nella test suite. I criteri di adeguatezza sono tipicamente utilizzati per valutare il grado di copertura del testing, laddove un soddisfacimento completo del criterio di adeguatezza non è possibile.

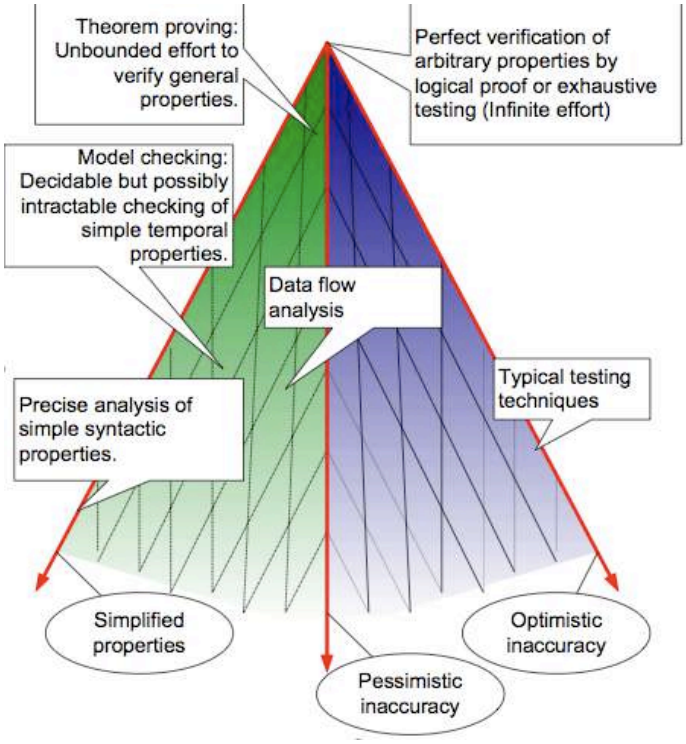
Un buon caso di prova (*test case*) è quello che ha una elevata probabilità di rilevare malfunzionamenti non ancora scoperti. Si dice in questo caso che il caso di test ha avuto successo.

La progettazione dei *test case* mira a definire casi di prova che scoprano sistematicamente differenti classi di malfunzionamenti con il minimo sforzo. Indirettamente il testing dimostra che il software rispetta le specifiche ed i requisiti fornisce un'indicazione sulla sua affidabilità e qualità globale ma non può dimostrare l'assenza di difetti, può solo dimostrare la loro presenza (**Tesi di Dijkstra**). La rilevazione di errori facilmente correggibili e/o il buon funzionamento del software possono indicare:

- affidabilità e qualità accettabili;
- inadeguatezza dei casi di prova (test-case).

Tali errori saranno poi rilevati dagli utenti e ciò comporta un aggravio dei costi rispetto a quelli, già alti, della loro rilevazione in fase di sviluppo.

E' importante rimarcare che Il testing è una delle due attività principali di verifica e validazione; l'altra grande famiglia di tecniche, trattata nella sezione precedente, è l'analisi. La differenza fondamentale tra le due è che mentre il testing mira rilavare malfunzionamenti, l'analisi mira verificare il soddisfacimento di proprietà (potenzialmente legate a difetti, e quindi futuri possibili malfunzionamenti).



La **Figura 3-12**, tratta da [117], mostra il trade-off tra le principali tecniche di V&V in termini di inaccuratezza. In essa, è evidente come la famiglia delle tecniche di testing ricada nella "inaccuratezza ottimisitca", ossia nelle tecniche che consentono di accettare programmi che non posseggono la proprietà analizzata (i.e., possiamo non rilevare tutte le violazioni). D'altro parte, le tecniche di analisi ricadono più nella inaccuratezza pessimistica: non è garantita l'accettazione del programma anche se possiede la proprietà analizzata (con la possibilità, dunque, di generare falsi allarmi).

La dimensione delle proprietà semplificate sta ad indicare invece una riduzione del grado di libertà per semplificare la proprietà da controllare.

Figura 3-12. Trade-off tra tecniche di V&V (tratta da [117])

Il processo di testing dovrebbe iniziare il prima possibile con la pianificazione dei test. Infatti, la generazione dei casi test nelle prime fasi ha diversi vantaggi: *i)* i test sono generati indipendentemente dal codice, quando le specifiche sono ancora ben impresse nella mente dell'analista; *ii)* la generazione dei casi di test può evidenziare inconsistenza e incompletezze delle specifiche corrispondenti; *iii)* i test possono essere usati come compendio delle specifiche dai programmatori. L'esecuzione dei casi di test richiede invece la presenza del codice e la possibilità di eseguirlo; pertanto la fase di esecuzione effettiva del test sul prodotto è prevista in uno stadio avanzato del processo di sviluppo.

Vi sono due approcci generali al testing, noti come test **Black Box,** e test **White Box.** Il primo si riferisce agli approcci che non richiedono la conoscenza del codice sorgente, ma si basano sulla specifica del prodotto o modulo da testare (in termini di requisiti funzionali/non-funzionali, di funzione logica svolta e di input/ouput, a seconda della granularità del modulo sotto test e dell'obiettivo del test stesso). L'approccio black box è molto utilizzato a livello di sistema, per rilevare, ignorando i dettagli interni, possibili malfunzionamenti nelle funzionalità specificate nei requisiti. Il criterio di adeguatezza in questi casi è pertanto tipicamente basato sulla *copertura dei requisiti*.

L'approccio *white box* invece si basa sulla conoscenza della struttura interna del codice sorgente; viene pertanto utilizzato più a livello di modulo, per testare l'operatività di singoli moduli. In questo caso, il criterio di adeguatezza si basa principalmente sulla *copertura di elementi del codice sorgente* (istruzioni, blocchi, condizioni, ecc.).

Le fasi principali del processo di testing sono comunemente raggruppate in: Test d'unità, Test d'integrazione, Test di sistema Test di accettazione (**Figura 3-13**).

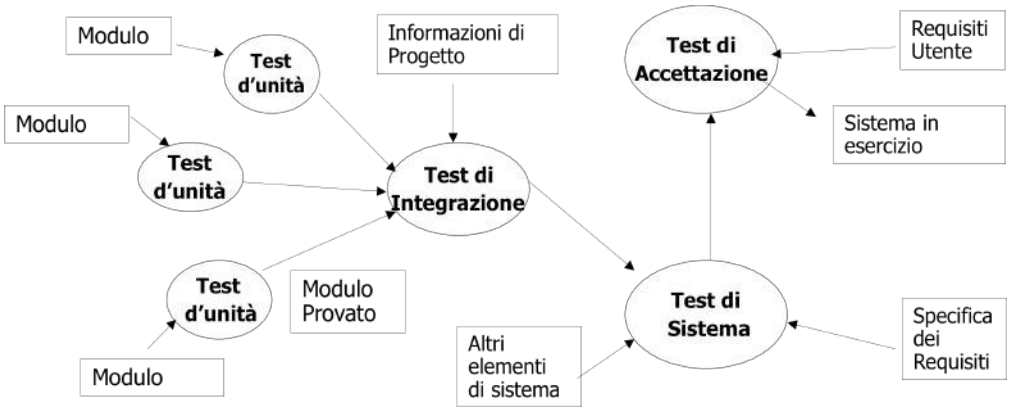


Figura 3-13. Fasi del processo di testing

Il **test di unità** verifica la più piccola unità di progettazione software: il modulo. Ciascun modulo viene verificato sulla base della documentazione relativa alla progettazione dettagliata. Spesso è effettuato subito dopo la codifica e ha necessità di utilizzare moduli pilota (*driver*) e filtri (*stub*) (**Figura 3-14**). Per sua natura, nel test di unità è diffuso l'uso di strategie white-box, anche se non in maniera esclusiva.

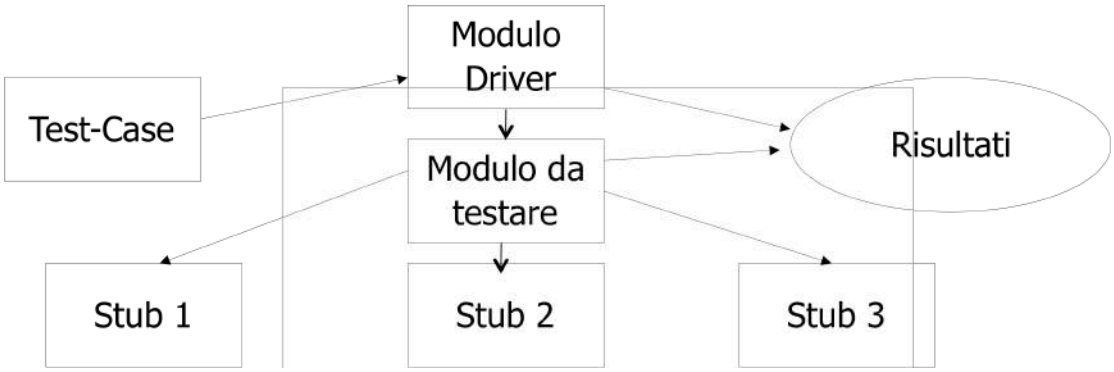


Figura 3-14. Test di unità

Il driver simula i moduli chiamanti; esso accetta i dati dei test-case, li passa al modulo e stampa i risultati. I moduli fittizi (*stub*) simulano i moduli gerarchicamente inferiori del modulo da testare, eseguono

eventualmente qualche manipolazione di dati, stampano o restituiscono i dati nel modulo da testare. *Driver* e *stub* sono onerosi in quanto devono essere realizzati ma non fanno parte del prodotto finale. Talvolta alcuni moduli non possono essere testati con il solo uso di *driver* e stub; in tal caso il test è posposto sino al test di integrazione. Il test di unità è più facile con moduli ad alta coesione, nel qual caso è necessario un minor numero di test-case, e gli errori sono rilevabili più facilmente.

Il **test di integrazione** mira a rilevare errori relativi all'interazione tra moduli. Esso considera i moduli testati singolarmente e costruisce in maniera incrementale la struttura del programma, aggiungendo uno o pochi moduli alla volta. Tale tipo di test è necessario per scovare errori dovuti alla comunicazione (reale) tra i moduli, ovvero a problemi nelle interfacce, a rilevare effetti collaterali negativi di un modulo su un altro, imprecisioni, accettabili localmente, ma che si amplificano in modo intollerabile, e problemi con l'utilizzo di dati globali. La strategia di integrazione può avvenire in modalità top-down o bottom-up. Nel primo caso, l'integrazione avviene seguendo la gerarchia di controllo iniziando dal *main* ed aggregando man mano i moduli subordinati, con modalità *depth-first* o *breadth-first* (**Figura 3-15**).

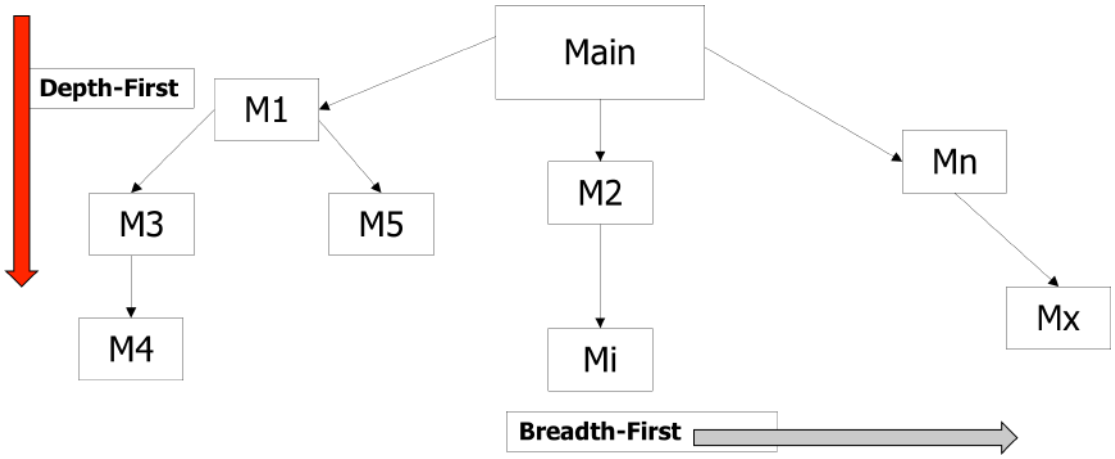


Figura 3-15. Test di integrazione top-down

Nell'integrazione top-down il processo avviene in 4 passi: il *main* è usato come driver del test e gli *stub* sono costituiti da tutti i moduli direttamente subordinati (gli *stub* sono sostituiti man mano con i moduli reali); dopo l'integrazione di ciascun modulo, si eseguono gli opportuni test; al completamento di ciascun test, un altro modulo reale viene aggiunto, sostituendolo allo *stub;* può essere infine condotto il test di regressione per accettarsi che non siano stati introdotti nuovi errori. Il processo si ripete fino alla completa integrazione del sistema. La modalità top-down presenta problemi di natura logistica: è richiesta l'elaborazione dei livelli bassi per poter testare in modo adeguato i livelli superiori (all'inizio ai livelli bassi vi sono ancora gli *stub* che potrebbero passare dati non significativi ai livelli superiori). Ciò può essere mitigato da una delle seguenti strategie: *i)* posporre molti test sino alla sostituzione degli *stub*; *ii)*costruire *stub* più realistici; *iii)* integrare bottom-up.

Nella strategia *bottom-up* l'integrazione inizia partendo dai moduli al livello più basso (quelli senza figli) e viene eliminata la necessità di *stub*. I passi sono i seguenti: i moduli di basso livello vengono aggregati in gruppi (cluster) che eseguono una specifica sotto-funzione software; viene realizzato un driver per coordinare l'esecuzione; si esegue il test del cluster; i driver sono sostituiti dai moduli reali che vengono aggregati nel programma. Un esempio è mostrato in **Figura 3-16**.

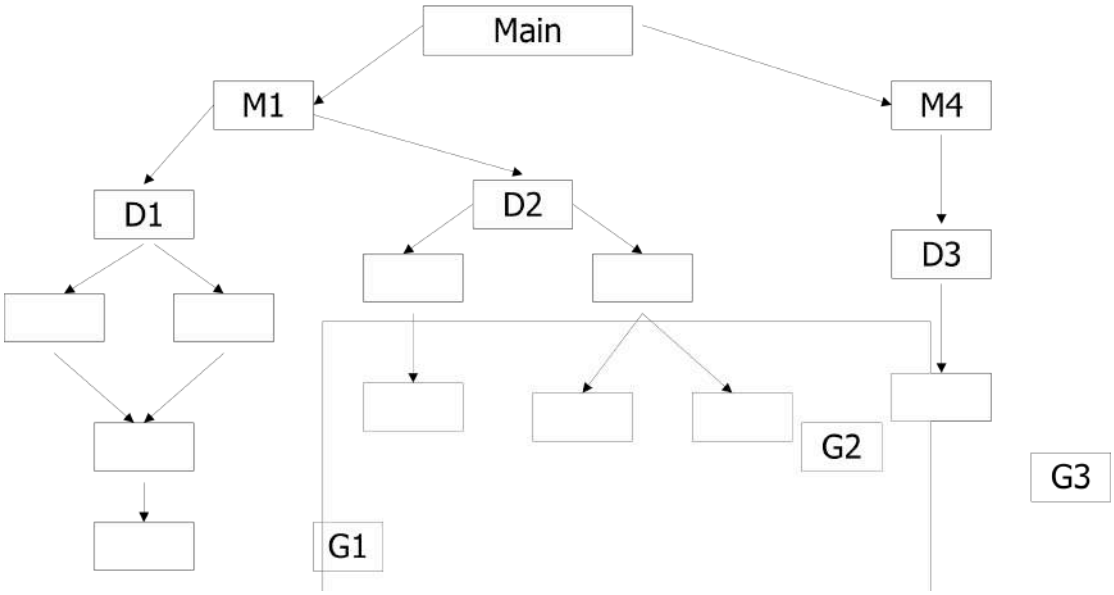


Figura 3-16. Test di integrazione bottom-up

Il limite della strategia bottom-up è che il programma non esiste fin quando non è aggregato l'ultimo modulo (il *main*); tuttavia ha il vantaggio di prevedere casi di test più semplici da progettare, e la mancanza di stub. Nella pratica, è molto diffuso l'uso di strategie miste, che integrano parte in modalità top-down e parte bottom-up, o partono da gruppi omogenei (ad esempio, integrano prima i moduli più critici).

Il **test di sistema** esercita il sistema software nella sua interezza. E' posto in essere quando viene tutti gli elementi del sistema sono integrati. Lo scopo del test è rilevare malfunamenti a livello di sistema, verificando che tutti gli elementi del sistema siano stati correttamente integrati e svolgano bene le funzioni loro assegnate. I casi di test per questa fase sono infatti tipicamente tratti dalla *specifica* dei requisiti utente. Inoltre, oer la sua natura, tale test non è condotto solo dall'ingegnere del software. Questo tipo di test è pertanto svolto secondo strategie black-box.

Infine, quando il prodotto ha passato la fase di test di sistema, l'ultimo passo è il **test di accettazione.** Quest'ultimo è condotto dall'utente finale per consentire la validazione di tutti i requisiti utente (che non necessariamente coincidono con i requisiti specificati; essi sono equivalenti solo se non vi sono stati errori di specifica). Questo test può richiedere molto tempo e portare alla rilevazione di errori cumulativi che non sono corretti e che possono portare al degrado del sistema.

Il test di accettazione (spesso noto anche come test di validazione), prevede due fasi: *i)* l'α-Test, condotto dal cliente o dall'utilizzatore finale, ma presso lo sviluppatore, ed avviene in ambiente controllato; *ii)* il β-Test, condotto dall'utente finale presso uno o più clienti; non è presente, generalmente, lo sviluppatore. Lo sviluppatore farà le opportune modifiche e poi rilascerà il prodotto.

Nel seguito, si andrà più nel dettaglio delle singole tecniche di test, ciascuna delle quali trova posto in una o più fasi tra quelle appena elencate. Esiste una vasta letteratura inerente le tecniche di test. Esse, infatti, sono tipicamente concepite per ottenere uno specifico obiettivo, ossia migliorare uno specifico attributo di qualità. Ciascuna di esse può essere meglio applicata in specifiche fasi del ciclo di sviluppo, su specifici tipi di sistemi e/o per specifiche dimensioni del sistema. Nel seguito viene fornita una breve descrizione delle tecniche di test più importanti nell'ambito dei sistemi *mission-critical*.

3.7.1 TESTING FUNZIONALE

Nel test funzionale, gli ingegneri derivano i casi di test a partire dalle specifiche, e collaudano il sistema rispetto ai suoi requisiti. Il termine "funzionale" si riferisce alla sorgente di informazione usata nella progettazione dei casi di test, non a cosa è testato. Più precisamente, questo tipo di test è riferito anche come "*specification-based testing*", mentre in maniera più impropria, come "*black-box test*" (poiché non si basa sul visionare il codice – tuttavia, black box racchiude diverse famiglie di tecniche, non solo quello funzionale). Il test funzionale rappresenta una tecnica fondamentale di test come illustrato in numerosi testi (e.g., [116][117][118]). Esso è tipicamente la tecnica di base, ed un punto di riferimento, nel progettare i casi di test. Vi sono vari criteri per definire i test funzionali. A partire dal dominio di input del programma, i casi di test possono essere ottenuti in maniera casuale (detto "random testing" [119]) attraverso una generazione a "forza bruta" a partire dalle specifiche, oppure, più spesso, in maniera sistematica, ossia formalizzando il processo di derivazione dei casi di test in passi elementari, e disaccoppiando i passi "human-intensive" da quelli automatizzabili [117]. In quest'ultimo caso, dato l'ingestibile numero di combinazioni degli input, la strategia di testing è determinata dal modo con cui i valori di input sono combinati: la maniera più comune è di usare il "principio di partizionamento", ossia separando lo spazio degli input in k *sottodomini* (i.e., classi), e prendendo dei valori di input da ognuno di essi. Nel "*partition testing*", gli elementi di ogni sottodominio si assumono essere strettamente correlati tra loro: per questo motivo i valori di input da un sottodominio sono scelti casualmente adottando una distribuzione di probabilità uniforme [120].

Vi sono diversi studi che comparano le prestazioni di un approccio sistematico, quale il *partition-based*, con il random testing. A seconda delle condizioni, i risultati sono discordanti; tuttavia un approccio sistematico al testing è tipicamente preferito, perché al tester è dato un maggior controllo sulla pianificazione ed esecuzione dei test.

Il test funzionale può avere due diversi scopi: stimare la proporzione di input che causano il fallimento (questo scopo supporta stime di *reliability,* che richiedono campioni casuali, non influenzati da chi li sceglie, per essere statistiche validi); cercare sistematicamente gli input che causano il fallimento, e rimuoverli (in questo caso, un campione random non è tipicamente sufficiente).

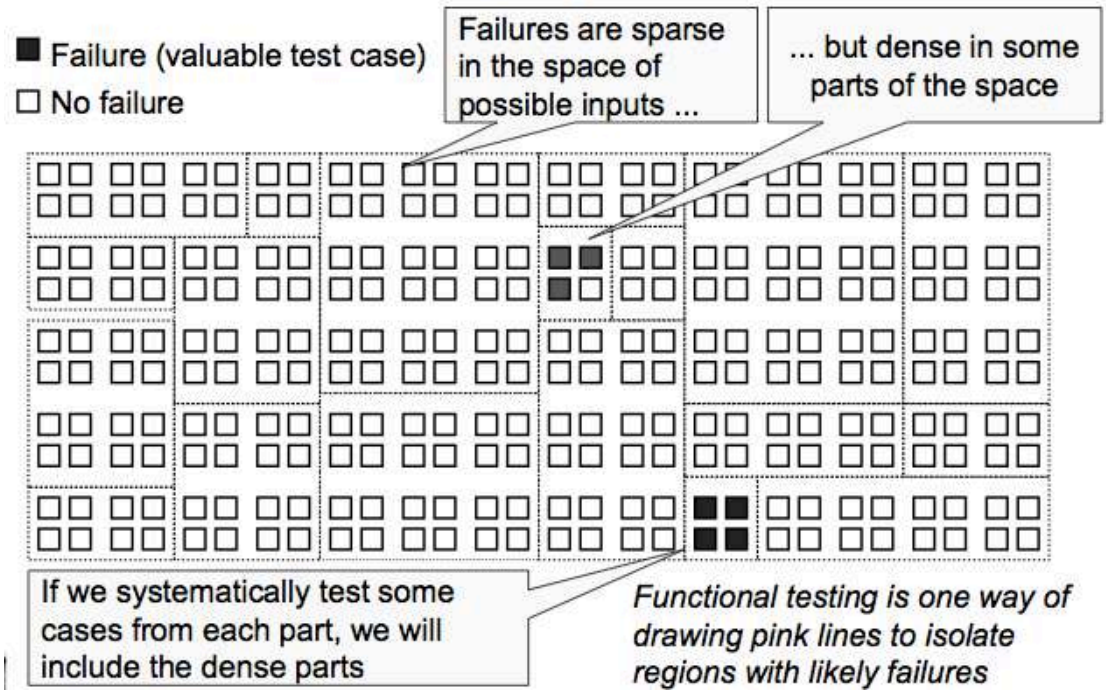


Figura 3-17. Il principio di partizionamento (tratta da [117])

Il principio di partizionamento consiste nello sfruttare conoscenza per scegliere campioni che più probabilmente includono regioni dello spazio degli input "speciali" o propense a causare fallimenti. Tali input sono sparsi nell'intero spazio degli input; ma è stato osservato che gli input che causano fallimento si addensano in alcune aree. Un partition testing perfetto richiede che lo spazio degli input sia separato in classi non sovrapposte i cui elementi (input) provocano tutti lo stesso comportamento (ad es., conducono tutti al fallimento). In questo modo è possibile selezionare un solo caso di test per classe e rilevare il fallimento. Tuttavia, questo è un approccio ideale, perché è complicato identificare tali classi. Nella pratica, l'approccio che tenta di approssimare questo metodo è noto come (Quasi*-)Partition Testing, che separa lo spazio degli input in classi la cui unione è l'intero spazio, ma in cui le classi possono sovrapporsi. Il caso desiderabile è che ogni fault porta a fallimenti che sono densi (facili da trovare) in alcune classi di input. Questo caso raramente è garantito; per cui ci si basa su euristiche basata sull'esperienza.

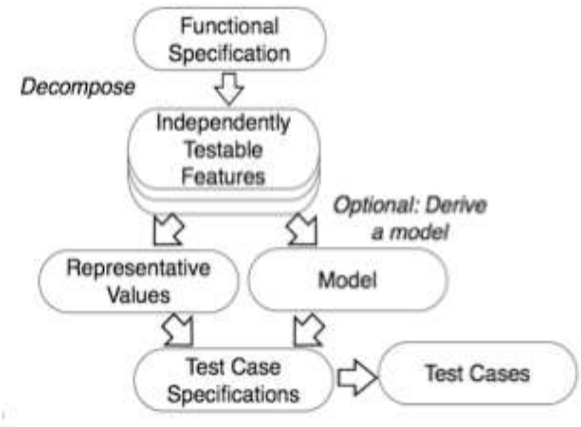
Il test funzionale usa la specifica (formale o informale) per partizionare lo spazio degli input; ad esempio, la specifica di un programma che calcola le "radici" di un'equazione di secondo grado (che riceve in input i tre parametri di un'equazione, *a,b,c*) suggerisce la divisione tra casi con zero, una e due radici reali.

In generale, si cerca di eseguire il test di ogni classe, e dei limiti tra le classi. Non c'è alcuna garanzia, ma l'esperienza suggerisce che i fault spesso si trovano a limite tra le categorie (ad esempio, nell'ipotetico programma "radice", il guasto potrebbe essere sollevato in casi limite, quali: a=0 e/o b=0 e/o c=0, o delta=0). Ciò che si f è dunque suddividere il dominio dei dati di input in classi di dati e si campiona da esse. Un test-case ideale rileva da solo una classe di errori (es. una scorretta elaborazione di tutti i dati di tipo

char), che altrimenti richiederebbe l'esecuzione di molti test, prima di intuire l'errore generico. Una classe rappresenta un insieme di stati validi o non validi per le condizioni d'ingresso. In generale una condizione di input può essere un valore specifico, un intervallo di valori, un insieme di valori, una condizione booleana.

Il test funzionale è ad oggi la tecnica base per progettare i casi di test, ed ha diversi vantaggi. Esso è utile nel raffinare le specifiche e stimare la testabilità prima della scrittura del codice; è efficiente nel trovare delle classi di fault che possono eludere altri approcci (e.g., *missing logic*); è ampiamente applicabile ad ogni descrizione del comportamento del programma che serve da specifica e ad ogni livello di granularità, dal test di sistema al test di modulo; è infine economico e tipicamente meno costoso da progettare ed eseguire rispetto ai casi di test strutturali (il codice del programma non è necessario, serve solo una descrizione del comportamento atteso; possono inoltre essere usate anche specifiche informali ed incomplete). Il test funzionale cerca di rilevare errori compresi nelle seguenti categorie:

- Funzioni incomplete o mancanti;
- Errori di interfaccia;
- Errori nelle strutture dati o negli accessi a data base esterni;
- Errori di presentazione;
- Errori di inizializzazione e terminazione.



Esso non è alternativo, ma complementare al test strutturale. La **Figura 3-18** mostra una possibile procedura [117] per un approccio sistematico al test funzionale.

Figura 3-18. Procedura per il test funzionale.

I passi principali sono i seguenti:

1. decomposizione della specifica in *caratteristiche testabili indipendentemente*;
2. Selezione dei *rappresentanti,* ossia valori rappresentativi di ogni input oppure comportamenti rappresentativi di un modello. Spesso, semplici trasformazioni input-output non descrivono un sistema. Vengono usati modelli nella specifica del programma, nella progettazione e, nella progettazione dei casi di test.
3. Specifica dei test, tipicamente tramite combinazioni di valori di input, o comportamenti del modello
4. Produzione ed esecuzione dei test effettivi.

Data una specifica ci possono essere più approcci per derivare i casi di test. Il test combinatoriale è una famiglia di approcci che consiste nella strutturazione (manuale) delle specifiche di test in un insieme di proprietà o attributi che possono essere sistematicamente variate in insiemi di valori. I due approcci più importanti sono il **category-partition testing ed il pairwise (o three-way, n-way) combinatorial testing.** Il testing "*category-partition*" permette di ridurre il numero di casi di test imponendo dei vincoli per escludere i casi di test potenzialmente ridondanti [121]. Gli approcci combinatori *n-way* per generare i casi di test permettono di combinare in maniera sistematica i valori di input e di ottenere un numero ridotto di casi di test. Il termine *combinatorial testing* viene spesso usato in riferimento a questa seconda categoria di tecniche. La scelta tra i due dipende dal tipo di dominio di input. Ad esempio, la presenza di molti vincoli nel domino di input suggerisce un metodo di partizionamento con vincoli (il summenzionato category-partition test); al contrario, valori di input con pochi o nessun vincolo suggerisce un approccio combinatoriale (come ad esempio il pairwise testing). Ancora, se le transizioni tra un insieme finito di stati sono facilmente identificabili, un approccio basato su FSA può essere opportuno. Specifiche descritte in un dato formato possono suggerire approcci corrispondenti (basate sullo stesso formalismo).

Il test combinatoriale ha il vantaggio di rendere i test automatizzabili. I test sono caratterizzati con attributi che possono essere combinati fra loro per ottenere delle combinazioni "rappresentative" per il test di una funzionalità. Lo spazio dei possibili valori dei singoli attributi può essere ridotto ad un subset. L'idea di base è identificazione dapprima gli attributi distinti nell'insieme dei dati, nel sistema, nella configurazione; successivamente si generano delle combinazioni degli attributi per il test .

Nel *category partition testing* si mira all'identificazione manuale di attributi che caratterizzano lo spazio di input. Gli attributi variano all'interno di un set di valori, e la riduzione delle combinazioni avviene tramite applicazione di vincoli agli input tali da eliminare test case potenzialmente ridondanti. La procedura prevista per il category-partition è la seguente:

1.**Decomposizione delle funzionalità che possono essere testate indipendentemente**

- Per ogni funzionalità si individuano:
- I parametri che la descrivono e gli elementi da cui dipende
- Per ogni parametro ed elemento si individuano un insieme di caratteristiche elementari, dette categorie

2.**Identificazione di valori rappresentativi**

Per ogni categoria si identifica classi di valori rappresentativi:

- *Normal values*
- *Special Values*
- *Boundary Values*
- *Error Values*

3.**Introduzione di vincoli sui valori individuati**

- Si impone che le combinazioni possono avere al più un errore;
- Si valuta se i valori sono compatibilmente combinati.

Nel pairwise testing le combinazionie degli input vengono opportunamente ridotte, considerando **coppie** dei valori degli attributi (a due a due). Si possono considerare anche combinazioni n-arie per n>2 (a tre a tre, a quattro a quattro, ecc.). L'idea alla base è che la maggior parte dei fallimenti software sono dovuti a combinazioni di pochi input (coppie, triple) e che quindi si possa coprire "la maggior" parte dei fault con pochi test. L'individuazione di coppie (triple) di valori di input può essere proibitiva se esistono diversi parametri del sistema, tuttavia si usano euristiche che semplificano la ricerca dei parametri.

3.7.2 TESTING STRUTTURALE

Quando la conoscenza del codice sorgente è impiegata per generare i casi di test, le tecniche di testing sono denominate "white-box". Poiché viene richiesto di conoscere la struttura del software, le tecniche in questa area sono spesso chiamate anche "structural testing". Il testing strutturale complementa il testing funzionale. Esso deriva i casi di test in base alla struttura del codice, con l'obiettivo di "coprire" la maggiore parte del codice del programma. Le entità del programma da coprire sono determinate dal criterio di adeguatezza dei test, che differenzia le diverse versioni del testing strutturale. La copertura può essere espressa in termini di istruzioni eseguite, rami nel grafo di flusso del programma, condizioni logiche, percorsi, o una combinazione di questi. Uno dei criteri di maggior successo è il criterio "condition/decision" (MC/DC), che è anche richiesto dagli standards di safety come il DO-178B. Il testing strutturale è spesso usato immediatamente dopo l'implementazione, in particolare per il testing di unità (ossia, testing di un singolo modulo software, come una funzione o una classe).

```
Repeat
  B0
if R1 then
   if R2 then
      if R3 then
           B1
      else
      B2
      endif
      if R4 then
           B3
      else
      B4
      endif
   else B5
  endif
endif
until R6
```

Oltre ai criteri basati sul flusso di controllo, un altro criterio ampiamente è il criterio "data flow". Il data flow testing genera i casi di test con l'obiettivo di coprire il flusso di dati tra la definizione di un dato e il suo utilizzo. Alcuni criteri sono la copertura di tutti i possibili percorsi "definition-use" (ossia, un percorso del CFG che va dalla definizione all'uso di una stessa variabile, in cui il valore non è ri-definito nel percorso), o la copertura di tutte le coppie "definition-use" (ossia, una coppia di definizione e uso di una qualche variabile, tale che almeno un percorso DU esiste dalla definizione all'uso).

Tra i criteri basati sul flusso di controllo, vi è il test basato sui cammini logici interni. In questo caso l'obiettivo è trovare test-case che esercitano insiemi specifici di condizioni e/o cicli. E' tipicamente Impossibile eseguire un testi esaustivo di tutti i cammini, che può essere enormemente elevato anche per piccoli programmi.

Nella figura a lato, ad esempio, se il ciclo viene eseguito max 20 volte si possono avere oltre 100.000 miliardi di cammini possibili, con 3.170 anni nell'ipotesi che ciascun test case sia elaborato in 1ms.

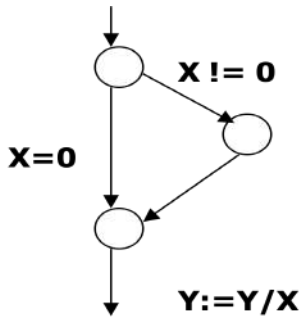
Per eseguire test strutturali, ci si basa tipicamente sulla rappresentazione tramite Control Flow Graph (CFG), introdotto nelle sezioni precedenti.

Il test strutturale prevede di creare dapprima la suite di test, misurare poi la coverage per poi identificare cosa manca. Nel caso ideale, i test-case vanno progettati in modo che tutti i cammini indipendenti all'interno di un modulo siano stati esercitati almeno una volta, che siano esercitate tutte le decisioni logiche nei casi vero e falso, che si eseguano tutti i cicli ai loro valori limite ed all'interno dei valori ammessi, e che siano esercitate tutte le strutture dati per assicurare la validità. E' ovvio che nella pratica solo alcuni di questi criteri possono essere soddisfatti, e tipicamente non al 100%. Le misure di copertura (*coverage*) raggiunte sono poi utilizzati come indicatori di come procede il test, e come criterio di completamento.

Sono di seguito riassunti brevemente i principali criteri di copertura utilizzati:

Statement-based testing

***Criterio di adeguatezza*:** Richiede che ogni istruzione sia eseguita almeno una volta



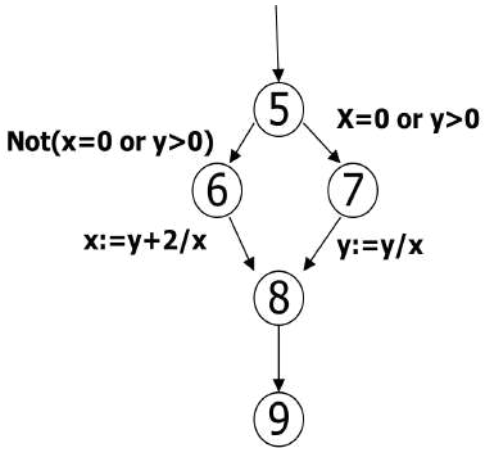
Esempio: if (x!=0) then S1
y=y/x

Misura di Coverage: (**#Statement Eseguiti) / (#Statement)**

E' importante notare che un test-case che assicuri la copertura degli statement non sempre comporta l'esecuzione del programma per tutti i valori delle decisioni (sia il ramo *true* sia quello false). Per questo obiettivo, va adoperato il secondo criterio.

Decision-based testing

Criterio di adeguatezza*:* Richiede che ogni arco del control-flow graph sia percorso ameno una volta. Un test dovrà contenere almeno 2 dati di test per ciascuna decisione, uno per ogni ramo uscente da un nodo decisione. *Il criterio di copertura delle decisioni include quello di copertura degli statement.*



Esempio: if (x=0 or y>0)
then y:=y/x
else x:=y+2/x

Misura di Coverage: (**#Archi Eseguiti) / (#Archi)**

Test-Case B= {(x=5,y=5),(x=-5,y=-5)}

Causa l'esecuzione di entrambi gli archi. Però può non rilevare il malfunzionamento dovuto a divisione per 0, infatti B non copre entrambi valori delle 2 condizioni in OR

Condition-based testing

Criterio di adeguatezza: Richiede che ogni singola condizione che compare nelle decisioni del programma valga sia vero che falso per diversi dati di test. Ad esempio, per l'istruzione "if (x=0 or y>0)", un possibile test-case è: C={(x=5,y=5), (x=0,y= -5)}. Infatti:

(x=5,y= 5)→ la condizione x=0 è False e y>0 è True

(x=0,y=-5)→ la condizione x=0 è True e y>0 è False

Misura di Coverage: (**#Valori di Verità presi da tutte le condizioni singole)/(2*#Condizioni singole).**

Un test che soddisfa il criterio di copertura delle condizioni non è detto che soddisfi quello delle decisioni. Nell'esempio, per C la condizione è sempre e solo vera. Per soddisfare entrambi è necessario il seguente criterio.

Decision/Condition-based testing

Criterio di adeguatezza: richiede che condizione e ogni decisione sia coperto da almeno un casi di test.

L'idea è cercare di coprire tutte le condizioni composte e tutti gli archi. Tuttavia, il criterio delle condizioni composto causa una crescita esponenziale del numero di casi di test. Un criterio che ha avuto particolare successo e che tenta di trovare il giusto trade-off tra numero di test e copertura è il seguente:

Modified condition-decision (MC/DC) coverage.

L'idea in questo caso è di testare combinazioni "rilevanti" di condizioni, evitando una crescita esponenziale. Rilevanti significa: ogni condizione singola che influenza indipendentemente l'esito di una decisione.

Questo criterio richiede per ogni condizione C due casi di test: i valori di tutte le condizioni valutate tranne C sono gli stessi. La condizione composta vale *true* per un caso di test e *false* per l'altro. In questo modo si ottiene una complessità lineare: N+1 casi di test per N condizioni singole. Questo criterio è richiesto dallo standard per i sistemi software nell'ambito avionico, il DO-178C. In **Figura 3-19** è riportato un esempio: i casi sottolineati influenzano indipendentemente l'output della decisione.

```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

Figura 3-19. Esempio di applicazione del MC/DC *testing* [117].

La copertura dell'MC/DC è "implicata" dal criterio delle condizioni composte e implica tutti gli altri. E' dunque più forte del criterio delle decisioni e degli *statament*, ma è un buon compromesso tra completezza e numero di casi di test (e per questo ampiamente usato).

Path-based testing

Criterio di adeguatezza: Richiede che ciascun cammino di un insieme di cammini di base venga eseguito almeno una volta

I test case effettuati sull'insieme di base garantiscono l'esecuzione di ciascuna istruzione almeno una volta.

La complessità ciclomatica di Mc Cabe del grafo definisce il numero dei cammini indipendenti nell'insieme di base di un programma, e rappresenta il limite superiore del numero di test da effettuare. La misura di copertura in questo caso è data dal: **(numero di cammini eseguiti)/(numero ciclomatico)**

3.7.3 MODEL-BASED TESTING

In aggiunta alle tecniche menzionate e alle loro numerose varianti, sono state proposte altre tecniche più specifiche e di nicchia. La diffusione dello sviluppo basato su modelli (Model-Based Development, MBD) ha favorito l'adozione su larga scale del Model-Based Testing (MBT). Il model-based testing è una tecnica in cui il comportamento a tempo di esecuzione di una "implementazione sotto test" è confrontato con una specifica formale del comportamento atteso, o modello. Questa tecnica, in quanto permette di generare casi di test in modo sistematico ed automatico, è applicata con successo per la verifica di sistemi di scala industriale, ed è supportata da molti tool commerciali. I modelli possono essere sviluppati usando molti formalismi (grammatiche, insiemi, stati), ma essi sono tipicamente definiti come automi a stati finiti (Finite State Automata, FSA) o simili, come le macchine a stati UML. I casi di test possono essere generati per coprire tutte le transizioni e/o tutti gli stati, fare cammini casuali, coprire prima i percorsi più brevi, e coprire prima i percorsi più probabili. Una panoramica dei criteri per generare i casi di test con approcci model-based è presentata in [122].

Un problema rilevante nel MBT è la minimizzazione/selezione dei casi di test, che è direttamente correlata alla scalabilità di questo approccio. Vi sono diverse classi di tecniche di selezioni nel MBT. La tecnica più semplice è il random testing [123], dove non c'è una scelta sistematica dei casi di test. Un trend rilevante nella selezione e prioritizzazione dei test è di massimizzare la copertura [124][125]. Nel MBT, la copertura è definita e misurata a livello del modello; essa può essere misurata dai casi di test astratti senza averli necessariaente eseguiti. La maggior parte delle tecniche di selezione basate su coverage sono riformulate in termini di un problema di ottimizzazione, dove l'obiettivo è di selezionare il miglior sottoinsieme di casi di test che ottengono una piena copertura. Per esempio, una tecnica presentata in [124] usa una ricerca "greedy" che seleziona il caso di test che copra la maggior parte degli statement non coperti fino a quel momento (tecnica della copertura addizionale). In modo simile, in [126] è stato usato un algoritmo genetico per massimizzare la copertura del sottoinsieme di casi di test selezionati. La selezione basata sulla similitudine dei casi di test (Similarity-based Test Case Selection, STCS) è una categoria di tecniche di selezione recentemente proposta [127][128][129] e che può essere applicata sia sul codice sia per il MBT. Essa seleziona i casi di test più "differenti" tra di loro rispetto a una misura di similitudine, che richiede di assegnare un valore di similitudine ad ogni coppia di casi di test e di minimizzare la media della similitudine tra coppie di casi di test [122]. L'idea alla base è di diversificare i casi di test selezionati, assumendo che più diversi sono i casi di test, maggiore è la loro capacità di rilevare i difetti. In [128] è stata condotta una simulazione su larga scala, basata su due casi di studio industriali, per studiare la relazione tra la rilevazione dei difetti e la distribuzione della similitudine tra casi di test. È stato osservato che la situazione ideale per le tecniche STCS è quando, in una test suite, (1) i casi di test che rilevano difetti distinti sono dissimili, e (2) i casi di test che rilevano uno stesso difetto sono simili.

3.7.4 TESTING NON-FUNZIONALE

Oltre alla copertura funzionale e strutturale, il testing deve essere rivolto a verificare i requisiti non-funzionali, in particolare nei sistemi mission-critical. A seconda dell'attributo di qualità che si considera, esiste un'ampia varietà di tecniche per migliorare (o anche semplicemente per valutare) un prodotto dal

punto di vista non-funzionale. Nel seguito, ci focalizziamo sulle tecniche più rilevanti per il contesto considerato.

3.7.4.1 RELIABILITY TESTING

Il testing orientato alla affidabilità (*reliability testing*) risale agli anni '80 ed ha una lunga storia. Quando il testing è utilizzato per quantificare (e in alcuni casi migliorare) la reliability, esso viene denominato testing operazionale (*operational testing*). Con esso, gli ingegneri mirano ad ottenere una alta affidabilità operazionale, intesa come la probabilità di non fallire durante la fase operazionale. Idealmente, il testing operazionale è storicamente considerato come l'approccio più promettente per migliorare la reliability, poiché è l'unica forma di testing pensata per trovare i fallimenti in base alla loro probabilità di occorrenza durante il reale utilizzo da parte dei suoi utenti. L'espressione "*statistical testing*" è anche usata per riferirsi a questa tecnica. Tuttavia, con "statistical testing" ci si riferisce più recentemente ad un approccio leggermente diverso, il cui obbiettivo è di soddisfare un criterio di adeguatezza espresso in termini di proprietà funzionali o strutturali (ad esempio, assicurare che ogni elemento strutturale è sollecitata il più frequentemente possibile [130]). Il testing operazionale è anche visto come un caso particolare del random testing con distribuzione di probabilità non uniforme [130][131][119]. Pezzé [117] si riferisce a quest'ultima accezione come "operational profile-based testing", o semplicemente testing operazionale.

Il testing operazionale è basato sull'idea che alcune funzionalità saranno più esercitate di altre durante l'esercizio del sistema, e che quindi meritano un testing più approfondito. Il profilo operazionale è costruito assegnando delle probabilità di occorrenza alle funzionalità del sistema, oppure a gruppi di funzionalità. Ciò è anche supportato da modelli analitici, come ad esempio le catene di Markov [132][133]. Tuttavia, l'accuratezza del profilo operazionale dipende fondamentalmente dalla accuratezza delle informazioni su come il sistema sarà utilizzato. Le principali fonti per costruire profili operazionali sono: (i) colloqui con esperti di dominio; (ii) dati dal campo sull'utilizzo del sistema (o sue versioni precedenti), che sono in alcuni casi disponibili all'interno di una organizzazione; (iii) dati riportati in letteratura [134].

I contesti più importanti in cui il testing operazionale è stato usato sono l'approccio Cleanroom [135], e il processo Software Reliability Engineering Test (SRET) proposto da John Musa [136].

Nella metodologia di sviluppo Cleanroom [135][137][138], il testing operazionale è adottato per rimuovere i possibili fallimenti con probabilità proporzionale alla loro severità, e il "debug testing" (ossia, approcci di testing funzionale/strutturale non orientati alla reliability) non viene effettuato. Il metodo Cleanroom identifica il testing operazionale come metodo per certificare il software rispetto ad un prefissato tempo medio tra fallimenti (MTTF) [139]. Esempi di applicazione di questo approccio sono riportati in [137] [140][141]. In modo simile, Musa ha proposto il SRET, in cui il testing operazionale è l'elemento principale del processo. Esso consiste di quattro passi: (i) definire il requisito di reliability, (ii) sviluppare i profili operazionali, (iii) predisporre il testing, e (iv) eseguire iterativamente i test analizzando via via i dati di fallimento [136]. Musa e i suoi colleghi presso l'AT&T hanno ottenuto risultati significativi a supporto del testing operazionali, in particolare una riduzione di un'ordine di grandezza dei problemi riportati dai clienti [142]. Ulteriori risultati a supporto del testing operazionale sono riportati in [143], ed in [144] dove il testing operazionale è usati in sistemi multi-modali.

Più recentemente, Chen et al. [145][146] hanno provato a migliorare l'efficacia del random testing uniforme tenendo conto del feedback generato dai casi di test che non rilevano difetti. Essi usano il concetto di adaptive random testing (ART), utilizzando profili di testing, diversi dai profili operazionali, in grado di distribuire i casi di test uniformemente sul codice. Il testing adattivo è anche l'approccio adottato in [148]. In [149] lo stesso approccio è usato per valutare la affidabilità del software, con l'obiettivo di minimizzare la varianza dello stimatore di affidabilità. Sebbene gli approcci citati promuovono il testing operazionale e orientato alla affidability, riportando buoni risultati, è tuttavia difficile, almeno ad oggi, trovare delle prove forti che il testing operazionale è sempre una pratica adatta.

3.7.4.2 TEST DI PERFORMANCE

Le prestazioni sono un attributo esterno basato sui requisiti utente, legato alla capacità del sistema di fornire la risposta richiesto entro determinati vincoli di tempo e di risorse. Le prestazioni sono principalmente valutate o attraverso approcci modellistici (ad esempio in fase di progettazione) o attraverso tecniche di performance testing. Le tecniche di modellazione delle prestazioni sono simili a quelle usate nell'analisi di affidabilità e disponibilità, essendo esse basate su formalismi di tipo Markoviano e su reti di code, ed hanno lo scopo di analizzare il possibile comportamento del sistema da un punto di vista architetturale, cambiando i parametri di interesse e migliorando la pianificazione e la allocazione delle risorse nel sistema.
L'altro tipo di approcci focalizzano il testing per valutare e migliorare le prestazioni. Nella pratica, la mancanza di test di questo tipo e di una opportuna loro pianificazione porta spesso a problemi di performance [150]; sono quindi necessarie strategie per supportare la scelta e la esecuzione di test di performance. In questa direzione, in [151] viene sottolineata l'importanza del progettare ed eseguire il test di performance sin dalle prime fasi di progettazione architetturali. Gli autori di questo studio propongono un approccio che supporta la selezione dei casi d'uso rilevanti dal progetto architetturale e l'esecuzione di questi test usando le prime versioni disponibili del software. Liu et *al.* propongono in [152] un approccio ibrido basato sul testing empirico e sulla modellazione basata su reti di code per predirre le prestazioni di applicazioni basate su componenti. Essi evidenziano l'importanza di isolare i problemi di prestazioni causati dalla logica di business da quelli dovuti all'infrastruttura middleware sottostante (i.e., il container) e all'ambiente operativo (e.g., hardware, OS, e DBMS). In [153] sono stati combinati il performance testing e un modello simulativo per predire problemi di performance in una applicazione di e-commerce, composta da una applicazione web di front-end, un application server come middle-tier, e un DBMS di back-end. Lo studio fornisce inoltre una serie di passi per supportare la diagnosi di rallentamenti di performance, e una procedura per la generazione ed esecuzione automatica di casi di test. In [154] è presentato un survey su ulteriori lavori basati su testing per valutare le prestazioni di sistemi basati su componenti. Gli autori hanno classificato gli approcci in base alla fase del ciclo di sviluppo in cui sono applicati, e discutono i benefici e i problemi di questi. Uno studio precedente [155] offre una panoramica di approcci basati sia su modelli sia su misure che mirano alla valutazione di performance di sistemi basati su componenti.

3.7.4.3 TEST DI ROBUSTEZZA

La robustezza di un sistema è definita come il grado con cui il sistema può operare correttamente in presenza di input eccezionali o condizioni ambientali stressanti [156]. Poiché la robustezza di un

componente software ha ripercussioni sulla affidabilità dell'intero sistema, la sua corretta valuazione è essenziale. Ciò è particolarmente importante per componenti di terze parti come ad esempio Off-The-Shelf (COTS), poiché essi sono spesso sviluppati senza considerare un contesto particolare e possono indurre malfunzionamenti una volta integrati in un nuovo sistema. Un modo comune per valutare la robustezza del software è attraverso il test di robustezza (robustness testing). Esso mira a valutare la capacità di un sistema di resistere e reagire ad input erronei ed eccezionali. Il robustness testing fu introdotta come tecnica automatica, che tratta il sistema o il componente da collaudare come una black-box. Le campagne sperimentali di robustness testing basate sul robustness failure rate (la percentuale di input errati gestiti in maniera non robusta) permettono la comparazione di componenti, e consentono di identificare e mitigare possibili input problematici.

La robustezza è valutata in presenza di input errati alle interfacce di un componente o sistema. Tali input errati possono essere causati guasti in altri componenti, o derivare da un utilizzo errato o malizioso del sistema da parte di utenti umani. In questa prospettiva, ci sono due categorie principali di approcci al robustness testing. Nella prima, i guasti possono essere iniettati in un componente A per collaudare la robustezza del componente B. Se e quando i guasti iniettati diventano attivi, c'è la possibilità che questi si manifestino come errori all'interfaccia tra il componente A ed il componente B, propagandosi verso il componente B che è oggetto del robustness testing. L'iniezione di guasti in A può essere ottenuta attraverso la mutazione del codice del componente. Questa tecnica è stata adottata in [157]; sebbene efficace, essa richiede che i guasti iniettati siano rappresentativi di guasti reali, e che i guasti iniettati si attivino e si propaghino all'interfaccia del componente B. La seconda categoria è di iniettare input errati direttamente all'interfaccia del target. In pratica, i parametri del servizio (ad esempio i parametri di una funzione) sono corrotti usando valori errati. Questo approccio non richiede di attendere l'attivazione dei guasti, poiché è sufficiente invocare il componente target con un valore già corrotto. In entrambi i casi, l'interfaccia del servizio è la locazione dell'errore. Per questa ragione il robustness testing è anche chiamato "interface error injection" [158].

Il robustness testing è stato adottato ampiamente e con successo alle interfacce dei sistemi operativi verso le applicazioni (system calls) e verso i device drivers [159][160]. Queste interfacce sono di interesse perché attraverso esse è possibile valutare la robustezza del sistema operative rispetto a comportamenti errati nelle applicazioni e nei driver, che si dimostrano essere particolarmente soggetti a malfunzionamenti [161].

I tipi di errori iniettati all'interfaccia di un servizio possono essere classificati in base a tre tipi di modelli di errore [162]:

- *Fuzzy*: gli errori sono scelti casualmente tra tutti i possibili valori del dominio di input del servizio. Per cui, gli esperimenti con questo tipo di errori tendono ad essere numerosi per potere ottenere una buona confidenza nei risultati finali.
- *Data-type based*: gli errori sono selezionati in base al tipo di parametri di ingresso. La selezione degli errori è condotta in base all'esperienza dell'ingegnere o con metodi sistematici (e.g., boundary analysis). In base al tipo di parametri, il numero di iniezioni può variare da uno a decine di casi per parametro.
- *Bit Flip*: gli errori sono permutazioni (invertendo un bit) dei parametri di ingresso del servizio. Questo modello deriva dai guasti hardware, che sono modellabili come "bit flips" [163]. È un approccio semplice da usare, ma richiede molti esperimenti a causa del numero di bit potenzialmente iniettabili per ciascun parametro di ingresso.

Gli errori possono essere iniettati in un preciso istante temporale (time-driven injection) o quando avvengono specifici eventi (event-driven injection). Il primo approccio assume che il sistema sia stato già portato in uno stato desiderato prima di effettuare il robustness test. Il secondo approccio inietta errori quando viene osservata una precisa sequenze di chiamate verso l'interfaccia del servizio.

BALLISTA [164][165] è un approccio, corredato da un tool, particolarmente noto e di successo. È stato il primo approccio per valutare ed effettuare il benchmark della robustezza di sistemi operativi commerciali rispetto alla interfaccia POSIX per le system call [166]. BALLISTA adotta un modello di errore basato sui tipi di dato, ossia, definisce un sottoinsieme di valori invalidi per ogni tipo di dato previsto dallo standard POSIX. Un caso di test consiste in un piccolo programma che invoca una chiamata di sistema target usando una combinazione di valori di input validi e invalidi. Gli esiti dei test sono classificati per severità, in base alla scala C.R.A.S.H.: c'è un fallimento Catastrofico quando il fallimento influisce su più task e sul sistema operativo stesso; i fallimenti Restart e Abort avvengono quando il task lanciato da BALLISTA è ucciso dal sistema operativo o è in stallo; i fallimenti Silent e Hindering avvengono quando la chiamata di sistema non ritorna un codice di errore, o ritorna un codice di errore errato. BALLISTA ha trovato numerosi input invalidi non correttamente gestiti (Restart e Abort), e rilevato alcuni fallimenti Catastrofici legati a puntatori errrati, overflow numerici, e overrun di file [164].

3.8 DEBUGGING

Lo sviluppo del software è una attività molto complessa, e come tale inevitabilmente soggetta ad errori; in particolare, sono da considerare due grandi classi di problemi all'interno del processo di sviluppo: quelli derivanti da una specifica dei requisiti inesatta, e dai difetti nell'implementare una corretta specifica dei requisiti. I primi vengono introdotti sin dall'inizio dello sviluppo e come tali sono estremamente dannosi, in quanto si ripercuotono su tutte le successive fasi sino a quando non emergono dall'inadeguatezza del sistema a realizzare alcuni compiti. I secondi vengono introdotti in fasi più avanzate, quindi di design o codifica, in seguito a fenomeni diversi, che vanno dalla scarsa conoscenza del linguaggio, ad assunzioni non verificate, alla schiacciante complessità del progetto, sino alla semplice distrazione. Tratteremo, quindi, di difetti, comunemente chiamati *bug*, che vengono introdotti in fase di codifica.

Un *bug* di un programma è un difetto o guasto che porta al malfunzionamento di esso, per esempio producendo un risultato inatteso o errato. Questo, quindi, può portare a conseguenze in certi casi particolarmente gravi, fino al punto di rendere vulnerabile ad attacchi informatici anche il computer che ospita il software. Causa di *bug* è spesso il codice sorgente scritto da un programmatore, ma può anche accadere che venga prodotto dal compilatore. Un programma che contiene un gran numero di *bug* che interferiscono con la sua funzionalità è detto *bacato* (in inglese "to be buggy").

Il *debugging***,** o semplicemente *debug*, indica quell'attività che individua la porzione di software affetta da bug rilevata a seguito dell'utilizzo del programma. L'attività di *debug* è una delle operazioni più importanti per la messa a punto di un programma, spesso estremamente difficile per la complessità dei software oggi in uso e delicata per il pericolo di introdurre nuovi errori o comportamenti difformi da quelli desiderati nel tentativo di correggere quelli per cui si è svolta l'attività di *debug*. Si deve infatti ricordare che una buona tecnica di programmazione deve ridurre il tempo passato a ricercare bug: programmare rapidamente solo per passare il restante tempo di sviluppo all'interno del debugger è raramente una tecnica vincente.

L'attività di *debugging* può presentare delle difficoltà. Tali difficoltà possono essere dovute ai seguenti motivi:

- il sintomo e la causa possono essere in parti "lontane" del software;
- il sintomo può scomparire solo temporaneamente, dopo la correzione di un altro bug;
- il sintomo può non essere causato da bug specifici, ma da problemi intrinseci all'ambiente di esecuzione, come ad esempio gli errori di arrotondamento;
- il sintomo può essere causato da un errore umano, non facilmente individuabile, come ad esempio la stessa variabile con due nomi diversi ma simili per errata digitazione;
- il sintomo può dipendere da problemi di temporizzazione e non di elaborazione;
- può essere difficile riprodurre esattamente gli input che hanno causato il malfunzionamento, come nei sistemi real time;
- il sintomo può essere 'intermittente';
- un solo difetto può causare più malfunzionamenti e un solo malfunzionamento può essere causato dall'azione combinata di più di un difetto.

Scopo del *debugging* è ridurre la distanza tra difetto e malfunzionamento mantenendo un'immagine dello stato del processo in esecuzione in corrispondenza dell'elaborazione di specifiche istruzioni.

Il *debugging* avviene attraverso l'utilizzo di uno strumento chiamato *debugger* (o correttore), il quale consente di eseguire *step by step* il codice dell'applicazione, ispezionando lo stack e lo heap durante il funzionamento di un programma, verificando i valori delle variabili, i valori dei parametri e lo stato dei componenti.

Durante l'operazione di *debugging* viene impostato il *breakpoint*, cioè il punto in cui si deve bloccare l'esecuzione del programma, e si comanda l'esecuzione del codice. Settare il *breakpoint* su una riga del programma comporta il blocco dell'esecuzione del programma immediatamente prima dell'esecuzione della riga.

In questo caso vengono utilizzati anche i *watchpoint* (o variabili di watch). Un *watch* in generale, è una semplice istruzione che inoltra il valore di una variabile verso un canale di output. L'inserimento di un *watch* (o *sonda*) è un operazione invasiva nel codice in quanto anche nel watch potrebbe annidarsi un difetto. In particolare l'inserimento di sonde potrebbe modificare il comportamento di un software concorrente.

I metodi principali di *debugging* sono:

- La Forza Bruta
- Il Ragionamento induttivo
- Il Ragionamento deduttivo
- Il Backtracking

Quello della *Forza Bruta* è il modo più inefficiente per eseguire il *debugging*. Diversi sono gli approcci possibili, come usare *storage dump*, cioè stampe dello stato della memoria in esadecimale o ottale, disseminare il codice di sonde per catturare quante più informazioni possibili e valutarle, e usare qualche strumento di debugging automatico, che permette di analizzare l'esecuzione del programma inserendo punti di break, osservazione di variabili, etc. Quest'ultimo approccio è largamente inefficace perché può produrre eccessive informazioni da comprendere. E' da precisare che la 'forza bruta' è una metodologia a cui ricorrere in *ultima ratio*, cioè quando altre tecniche hanno fallito.

Il processo di *Ragionamento Induttivo* è un metodo che fa un'analisi che va dal particolare al generale. Esso è basato sulla raccolta di dati ed indizi sul fallimento, la formulazione e la verifica di ipotesi sulle possibili cause, in modo iterativo (schematizzato in Figura 3-20).

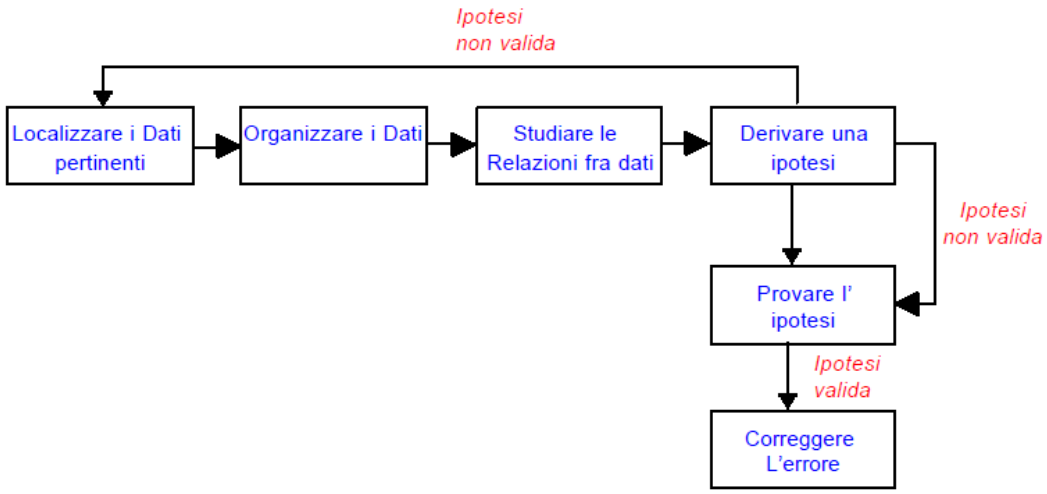


Figura 3-20. Debugging con ragionamento induttivo.

Il *debugging* eseguito utilizzando il *Ragionamento Deduttivo* avviene effettuando l'analisi che va dal generale al particolare. Si formulano varie ipotesi sulla causa dell'errore e si raccolgono dati per validarle o scartarle (schematizzato in Figura 3-21).

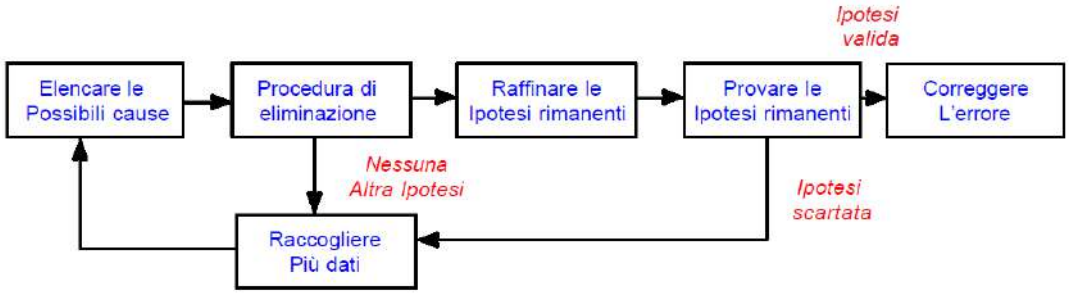


Figura 3-21. Debugging con ragionamento deduttivo.

Infine, nella metodologia di *debugging* per *BackTracking* si cerca di ripercorrere il codice "all'indietro" a partire dal punto dove si è verificato il malfunzionamento, che può essere un'istruzione di output o di eccezione. Analogamente alla tecnica delle Path Condition, però, diventa via via più difficile procedere all'indietro all'allargarsi del campo di possibilità.

Il *debugging* è un'attività estremamente intuitiva, che però deve essere operata nell'ambito dell'ambiente di sviluppo e di esecuzione del codice. Gli strumenti a supporto del debugging sono quindi convenientemente integrati nelle piattaforme di sviluppo (IDE), in modo da poter accedere ai dati del programma, anche durante la sua esecuzione, senza essere invasivi rispetto al codice. In assenza di ambienti di sviluppo, l'inserimento di codice di debugging invasivo rimane l'unica alternativa.

3.8.1 Statistical Debugging

Il *debugging statistico* è l'idea di utilizzare modelli statistici del programma che ha successo/insuccesso per rintracciare i *bugs*. Questi modelli statistici espongono relazioni tra specifici comportamenti del programma

e l'eventuale successo o fallimento di una esecuzione. Questo tipo di *debugging* formalizza e automatizza il processo di individuazione dei programmi che hanno un comportamento correlato con il fallimento, guidando in tal modo gli sviluppatori nella ricerca dei *bugs*.

L'idea di base del *statistical debugging* è identificare caratteristiche dell'esecuzione del programma che sono statisticamente correlate con i guasti. Questo approccio ci permette di concentrarci sulle anomalie: le differenze tra una moltitudine di impeccabili esecuzioni del programma e una singola esecuzione difettosa. Le singole esecuzioni garantiscono le proprietà che sono comuni a tutte le esecuzioni regolari; al contrario, l'esecuzione fallimentare prevede specifiche proprietà che riguardano il comportamento anomalo. Avendo queste proprietà a portata di mano, siamo in grado di esaminare le differenze, differenze che di solito sono in grado di suggerire la causa.

Un tipo di proprietà che può essere esaminata, per esempio, è la copertura del codice (*code coverage*) [104]: nell'esecuzione fallimentare, il codice difettoso deve essere eseguito per attivare il fallimento; inoltre, il codice che viene eseguito in una esecuzione difficile è più probabile che causi fallimento, cioè causi comportamento anomalo. Ulteriori proprietà includono i *function return values* (cioè, il comportamento erroneo della funzione legato ai fallimenti complessivi) [105], e *branches* (vale a dire, la valutazione sospettosa di rami condizionali) [106].

Per ottenere risultati significativi, gli approcci statistici devono considerare migliaia, o addirittura decine di migliaia di esecuzioni, sia ottenuti da una vasta suite di test, sia campionati sul campo. Ma anche avendo un gran numero di esecuzioni, i risultati possono essere ancora imprecisi.

L'approccio statistico migliore finora, il *CP model* di Zhang et al. In [107], riduce lo spazio di ricerca al 5% o a meno del 50% dei casi di test esaminati. Mentre il 5% inizialmente può sembrare impressionante, esso significa che ci sono ancora un enorme numero di linee di codice da esaminare. Inoltre, questi risultati sono ottenuti dalla cosiddetta suite Siemens [108], un benchmark utilizzato frequentemente per valutare gli approcci di localizzazione dei bug. La suite Siemens è "ottimistica" poiché i software inclusi sono relatiamente piccoli e contengono bugs "artificiali", e inoltre fornisce una ricchissima suite di test, per cui gli approcci statistici, quando applicati su questa suite, tendono ad esibire risultati "ottimistici" rispetto a software reali.

Uno svantaggio del *debugging statistico* è rappresentato dalla sua performance di runtime, in quanto ci possono volere delle ore per eseguire i test individualmente; ogni singolo test, infatti, viene eseguito in una nuova istanza della Java Virtual Machine (ovviamente questo dipende dallo strumento che è stato utilizzato per raccogliere i dati di copertura), inoltre, una grande quantità di dati di copertura devono essere raccolti e analizzati [109].

3.8.2 Delta Debugging

Zeller e Hildebrandt hanno ideato un metodo chiamato *delta debugging*, una tecnica che riduce sistematicamente le cause di guasto attraverso esperimenti automatizzati [110]. Cominciando con un insieme di possibili circostanze, il *delta debugging* isola, o semplifica, le circostanze che inducono al fallimento, le quali influenzano negativamente la funzione di un programma.

Il *Delta debugging* automatizza il metodo di debugging: l'idea di base è quella di stabilire per prima cosa un'ipotesi sul perché qualcosa non funziona; ad esempio, le chiamate di metodo specifiche con particolari argomenti devono essere chiamate in un certo ordine per produrre il problema. In seguito, si testa questa ipotesi, e si perfeziona o rifiuta a seconda del risultato del test. Applicato alle chiamate di metodo, fondamentalmente significa selezionare per prima una sequenza iniziale di chiamate di metodo, ed eseguire questa sequenza per verificare se il problema in questione si verifica. Se il problema è (ri) prodotto (cioè, il test ha esito negativo), si ridefinisce la sequenza: è possibile, ad esempio, rimuovere alcune chiamate di metodo. Successivamente, si dovrebbe eseguire nuovamente il test per valutare l'esito della ormai più piccola sequenza, che potrebbe essere allo stesso tempo una descrizione del guasto più precisa. In caso contrario, se il problema in questione non si verifica più (ad esempio, il test viene superato, perché è stato rimosso il metodo che attiva direttamente il fallimento), si dovrebbe rifiutare l'ipotesi considerata; per esempio, iniziando con una sequenza iniziale di chiamate di metodo.

Per automatizzare l'intero processo, il *delta debugging* in generale richiede solo un test che valuta un dato insieme di circostanze, nel nostro caso una data sequenza di chiamate di metodo. Tuttavia, non sarebbe sufficiente se tale test potrebbe distinguere solo tra due risultati di prova (che passa e fallisce). Invece, deve essere in grado di gestire una terzo esito, chiamato irrisolto.

Fondamentalmente, il *delta debugging* in alcuni casi implementa una ricerca binaria: si verifica la prima metà di un determinato insieme di circostanze e in funzione del risultato, continua sia con la prima metà, o procede con la seconda metà. Se la prima metà produce il guasto, la circostanza che induce il guasto deve essere inclusa. Altrimenti, se il guasto non viene riprodotto, tale circostanza deve trovarsi nella seconda metà.

Per far fronte a questo problema, il *delta debugging* in realtà estende la ricerca binaria suddividendo l'insieme non appena si verifica un risultato non risolto, e svolge quindi una ricerca ternaria, quaternaria, o in generale, n-aria, se necessario.

Figura 3-22 mostra la definizione formale dell'algoritmo di *delta debugging* attuale che minimizza un dato insieme di circostanze possibili. Per i dettagli e le altre definizioni, si può far riferimento al lavoro pubblicato da Zeller e Hildebrandt [110].

Let $\mathscr{C}$ be the set of all possible circumstances. Let $test : 2^{\mathscr{C}} \to \{\text{✘},\text{✔},\text{?}\}$ be a testing function that determines for a test case $c \subseteq \mathscr{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).
Now, let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ such that $c'_{\text{✘}} \subseteq c_{\text{✘}}, test(c'_{\text{✘}}) = \text{✘}$, and $c'_{\text{✘}}$ is 1-minimal.
The *Minimizing Delta Debugging algorithm ddmin(c)* is

$ddmin(c_{\text{✘}}) = ddmin_2(c_{\text{✘}}, 2)$ where

$$ddmin_2(c'_{\text{✘}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1,\ldots,n\} \cdot test(\Delta_i) = \text{✘} \\ ddmin_2(\nabla_i, max(n-1, 2)) & \text{if } \exists i \in \{1,\ldots,n\} \cdot test(\nabla_i) = \text{✘} \\ ddmin_2(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n)) & \text{if } n < |c'_{\text{✘}}| \\ c'_{\text{✘}} & \text{otherwise} \end{cases}$$

where $\nabla_i = c'_{\text{✘}} - \Delta_i, c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✘}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\text{✘}}) = \text{✘} \wedge n \leq |c'_{\text{✘}}|$.

Figura 3-22. Algoritmo di delta debugging.

Zeller e Hildebrandt hanno applicato l'algoritmo sull'input del programma: il *delta debugging* dovrebbe ripetere l'esecuzione del programma fallito mentre parti sistematicamente soppresse degli input, eventualmente hanno un input semplificato in cui ogni parte restante è rilevante per la produzione del fallimento. Purtroppo, applicando il *delta debugging* all'input comporta una serie di requisiti non banali. In primo luogo, abbiamo bisogno di un metodo per riprodurre automaticamente l'esecuzione. Questo è un compito che richiede tempo perché l'applicazione deve essere automatizzata mediante la simulazione di interazioni degli utenti. In secondo luogo, si ha bisogno di controllare l'input. In molti casi, gli input sono prodotti da altre parti o servizi esterni. In terzo luogo, abbiamo bisogno di conoscenze circa la struttura di input. Pur avendo la conoscenza di input testuale, esso può portare a miglioramenti delle prestazioni non avendo alcuna conoscenza circa la struttura dei dati non testuali; in questo modo si potrebbe rendere il *delta debugging* praticamente inutile.

3.8.3 Capture/Replay

Un campo di ricerca strettamente legato al debugging è il *capture/replay* (o acquisizione/riproduzione). In generale, gli approcci e gli strumenti di questo settore consentono agli sviluppatori in primo luogo di acquisire o, registrare le esecuzioni del programma, e riprodurre successivamente queste esecuzioni registrate per scopi di test e debugging. La maggior parte delle tecniche di capture/replay che sono state presentate rientrano in una delle due seguenti categorie. Essi rappresentano diverse strategie per raccogliere informazioni sufficienti al fine di essere in grado di riprodurre le esecuzioni registrate in precedenza:

Checkpoint. Questo strategia registra informazioni sullo stato del programma in punti ben definiti in una esecuzione del programma. Durante la riproduzione (o replay), l'esecuzione registrata può essere eseguita nuovamente a partire da questi *checkpoints*. Approcci che utilizzano i *checkpoints* per riprodurre (replay) includono il lavoro di Xu et al. [111], e RECRASH, uno strumento sviluppato da Artzi et al. [112]. RECRASH, per esempio, esegue un checkpoint ad ogni metodo di entrata e mantiene traccia dello stack. In questo modo, lo strumento consente allo sviluppatore di osservare una esecuzione in diversi stati prima del crash reale. Perché questa strategia cattura solo esecuzioni di programma in esecuzione in certi punti, può mancare informazioni che sono cruciali per riprodurre fedelmente il guasto originario.

Event based. In contrasto con la raccolta di informazioni riguardo lo stato del programma stesso, e in certi punti del tempo, questa strategia acquisisce le informazioni necessarie per la riproduzione come una sequenza di interazioni che portano ad un certo stato del programma; tipicamente, lo stato in cui il fallimento diventa evidente. In questo modo, questa strategia descrive come sia avvenuto lo stato difettoso, e quindi è in grado di rivelare in che modo lo stato difettoso del programma evolve. In genere, le interazioni acquisite vengono scritte come eventi in una sorta di traccia o di un file di log. Ad esempio, è possibile catturare le interazioni tra gli oggetti in forma di eventi, per mostrare come gli oggetti selezionati vengono costruiti e portati ad uno stato difettoso.

Alcuni tool di delta debugging proposti in letteratura sono:

JINSI [113][114] è un tool per il *delta debugging* che mira a ridurre al minimo le chiamate in arrivo ad un oggetto, introducendo e dimostrando la fattibilità del delta debug su chiamate di metodo. In precedenti implementazioni, JINSI richiedeva al programmatore di selezionare manualmente l'oggetto in questione, fornendo un suggerimento su dove il guasto potrebbe essere, e per fornire un predicato, che distingue un passaggio da una esecuzione fallimentare; inoltre, non è mai stato dimostrato in più di un unico esempio reale. Nella ultima versione, JINSI è diventato completamente automatico e non richiede nessuna interazione con il programmatore, e genera automaticamente predicati; esso funziona anche per i bug che non provocano crash, in cui è necessario un unico predicato iniziale per distinguere ciò che è previsto dal comportamento osservato. Inoltre, la precisione di JINSI è notevolmente incrementata compreso il dynamic slicing sia come filtro che come una guida strategica. Infine, JINSI è dimostrato su una vasta gamma di bug reali, scalando fino a problemi di notevole complessità.

SOBER [115] è un algoritmo di debugging statico, sviluppato dal *Departement of Computer Science – University of Illinois*. Esso localizza i bug del software senza alcuna conoscenza della semantica del programma. Si basa su predicati, cioè qualsiasi proposizione che restituisce un valore booleano, come ad esempio $idx < LENGTH$, $x!=0$ o $!empty(list)$.

A differenza degli approcci di debugging statistico esistenti, i quali selezionano i predicati correlati con gli errori di programma, SOBER analizza i predicati di un programma ad ogni esecuzione e in ognuna di esse, e ogni predicato può essere valutato più volte; inoltre, ogni valutazione può restituire sia il risultato di *vero* che di *falso*.

Per determinare il comportamento del programma durante un'esecuzione, viene utilizzato il concetto di *evaluation bias* definita come il numero di volte che un determinato predicato *P* viene valutato vero, rispetto a tutte le valutazioni. Indicando con n_t il numero di volte in cui *P* è vero, e con n_f il numero di volte in cui *P* è falso, si ha $\pi(P) = \frac{n_t}{n_t+n_f}$; da ciò si evince che $\pi(P)$ rappresenta la probabilità della *evaluation bias*, cioè per ogni predicato P, si indica con *X* la variabile aleatoria che rappresenta l'*evaluation bias* e, si denotano rispettivamente con $f_p(X|\theta_f)$ e $f_p(X|\theta_p)$, i due modelli per le esecuzioni fallite e per quelle andate a buon fine.

La rilevazione delle anomalie, si basa su quella che viene chiamata *fault relevance* di un predicato, che rappresenta quanto un determinato predicato sia legato al fallimento dell'esecuzione, in particolare, secondo la definizione, maggiore è la differenza fra $f_p(X|\vartheta_f)$ e $f_p(X|\vartheta_p)$ maggiore sarà il legame tra il predicato P e il fallimento.

Per quantificare tale differenza, viene utilizzata una funzione di similarità per confrontare i due modelli, costruita a partire dalla *evaluation bias* per *m* esecuzioni fallite, ed è definita come $L(P) = \frac{\sqrt{m}}{\sigma_P}\varphi(Z)$, dove σ_P è la radice quadrata della varianza di $f_p(X|\vartheta_f)$ e, $\varphi(Z)$ rappresenta la funzione di densità di probabilità della distribuzione normale standard.

Il punteggio per la stima della classifica, viene assegnato utilizzando una qualsiasi funzione monotona decrescente, nello specifico è stata scelta la funzione $s(P) = -\log(L(P))$; questo perché $\log(x)$ dà una buona misura della grandezza di *x* anche quando è molto prossimo allo zero.

Come per tutte le tecniche dinamiche, per poter analizzare le esecuzioni, è necessaria una fase preliminare di instrumentazione del codice sorgente, durante la quale vengono aggiunte alcune istruzioni al codice originale, che durante l'esecuzione, acquisiscono informazioni sul comportamento del software.

```
....
main(argc,argv)
{
 token  token_ptr;
 token_stream  stream_ptr;
 /* -- X45 System Initialization -- */
 X45_init_predicate_counters(186);
 if(X45_handle_cond_expr(0, (argc>2)))
 {
  fprintf(stdout,"The format is print_tokens filename(optional)\n");
  exit(1);
 }
 stream_ptr = open_token_stream(argv[1]);
 while(X45_handle_cond_expr(2, (!is_eof_token((token_ptr = get_token
       (stream_ptr))))))
 {
  print_token(token_ptr);
 }
 print_token(token_ptr);
 exit(0);
}
....
```

Figura 3-23. Esempio di codice instrumentato.

In particolare, SOBER, utilizza il sistema X45, che consiste in un set di funzioni inserite in corrispondenza di ogni predicato, utilizzate per tener traccia di ogni valutazione. Il sistema X45 produce, per ogni esecuzione, dei file traccia; questi file, contenenti le informazioni riguardanti le valutazioni dei predicati, vengono elaborati dall'algoritmo implementato in Matlab che effettua la classificazione dei vari predicati.

4 STRUMENTI DI SUPPORTO ALLA V&V NEL CONTESTO DI SELEX-ES

Gli strumenti software (tool) sono di fondamentale importanza per garantire un processo efficiente di V&V. Numerosi strumenti software sono tipicamente utilizzati per sostituire l'uomo nell'esecuzione di diverse attività di V&V, come ad esempio la esecuzione delle suite di test, la raccolta ed analisi degli esiti dei test, la generazione dei resoconti, la tracciabilità dei test rispetto agli altri artefatti software, il controllo di versione del codice dei test, e l'analisi automatica del codice volta a identificare tipologie di difetti ricorrenti e a verificare la conformità del codice rispetto alle regole di codifica dell'organizzazione.

Nell'ottica del progetto SVEVIA, è stata prevista una analisi preliminare sull'adozione di strumenti software nel contesto dell'azienda SELEX-ES, al fine di identificare le esigenze dell'azienda ed eventuali lacune negli strumenti utilizzati in essa. Dalla analisi è infatti emerso che, nonostante l'azienda conduca numerose attività di V&V dei propri sistemi, gli strumenti e le tecniche di V&V non sono sfruttati appieno. Non tutti i team utilizzano in maniera estesa degli strumenti per il testing, e di frequente si ricorre ad interagire con il System-Under-Test (SUT) manualmente mediante le interfacce utente da esso messe a disposizione. Inoltre, un'ulteriore problematica è la mancanza di condivisione delle tecniche e tecnologie utilizzate dai vari team.

Gli obiettivi dell'attività descritta in questa sezione sono lo studio e l'analisi degli strumenti a supporto della fase di validazione e verifica adottati nell'azienda. Lo scopo è stato quello di comprendere l'uso di tool software durante le varie fasi di V&V di moduli e sistemi software (unit testing, integration testing, analisi statica, etc.), e le necessità dell'azienda di cui tenere conto nell'ambito del progetto SVEVIA. Per perseguire tale scopo è stato seguito il seguente approccio. Sulla base dei requisiti di interesse per l'azienda, sono stati dapprima identificati i criteri da adottare per la valutazione degli strumenti software. Successivamente, gli strumenti considerati sono stati valutati in base a tali criteri, con la finalità di effettuare un *assessment* delle tecnologie adottate in SELEX-ES. Nella valutazione di tali strumenti si è tenuto conto delle loro caratteristiche tecniche e commerciali, della loro utilità nel contesto dei progetti dell'azienda e del loro impatto nei processi aziendali esistenti.

La valutazione degli strumenti è stata effettuata conducendo un sondaggio (*survey*) nell'azienda, che è stato trasmesso agli sviluppatori e ingegneri per chiedergli quali fossero i tool tipicamente utilizzati per il testing e quali fossero le loro caratteristiche. Il sondaggio è stato preparato sulla base dei criteri di valutazione precedentemente identificati. Nel seguito di questa sezione, sono prima descritti i criteri di valutazione ed il contenuto del sondaggio. Dopodiché, sono stati analizzati i risultati.

4.1 DEFINIZIONE DEL SURVEY

Sono stati definiti i criteri nell'ottica dei temi trattati nel progetto, e nell'ottica di evidenziare le necessità e le possibili limitazioni nell'uso di tool e nel processo di V&V dell'azienda. I tool di V&V sono stati analizzati secondo i seguenti criteri:

- **Features.** Sono le funzionalità fornite dal tool analizzato, quali:

  - Definizione dei test tramite linguaggi di scripting
  - Definizione dei test tramite record & replay
  - Definizione dei test mediante modelli (MDT)
  - Definizione di test casuali
  - Generazione automatica di codice per i test
  - Esecuzione automatica dei test
  - Determinazione automatica dell'esito dei test
  - Analisi automatica statica di regole di codifica
  - Analisi automatica statica di difetti di programmazione
  - Analisi dinamica
  - Ispezione manuale del codice
  - Test prestazionali
  - Test di robustezza
  - Pianificazione dei test
  - Reportistica
  - Tracciabilità
  - Bug tracking
  - Analisi della copertura del codice
  - Analisi delle metriche di complessità
- **Target.** Si riferisce a quale tipologia di SUT il tool può essere applicato, come ad esempio:
  - GUI
  - Applicazioni di rete
  - Applicazioni real-time
  - Applicazioni web
  - Applicazioni cloud-based
  - Indipendente dall'applicazione
- **Tecnologie.** Si riferisce alle tecnologie supportate dal tool, quali:
  - Linguaggio di programmazione supportato (es. Javs, C, C#, C++)
  - Sistema Operativo Supportato (es. Linux, Unix, Windows)
- **Aspetti commerciali.** Legate allo strumento che si dividono in:
  - Tipologia di licenza (Commerciale, Open-source, Proprietario)
  - Disponibilità di supporto agli utenti
  - Maturità del tool
- **Attinenza ai progetti dell'azienda.** È il criterio che indica quanto sia applicabile lo strumento analizzato ai progetti aziendali:
  - Inerenza ai progetti
  - Applicabilità ai progetti
  - Scalabilità in progetti complessi e di grosse dimensioni
  - Semplicità d'uso
  - Estendibilità

A partire da questi criteri, è stato preparato un sondaggio e divulgato al personale interno alle unità SELEX-ES di sviluppo software. Inoltre il sondaggio è stato esteso anche ai fornitori legati indirettamente a SELEX-ES e al progetto.

4.2 RISULTATI

Alla chiusura del sondaggio sono stati ottenuti 10 contributi relativi a 10 tool differenti. Il campione è risultato essere piccolo poiché è emerso che sono pochi i team, tra quelli intervistati, che fanno un uso sistematico di tool di V&V. I risultati ottenuti sono stati suddivisi in due categorie, denominate "tool interni" e "tool esterni". Il primo insieme di risultati si riferisce alle risposte ottenute da parte del personale SELEX-ES, mentre i secondi si riferiscono ad aziende esterne indirettamente legate all'azienda. Per permettere una corretta interpretazione dei risultati, i dati mostrati in questa sezione sono mostrati separatamente in base alla provenienza (tool interni o esterni). Per motivi di riservatezza, il nome dei tool analizzati sarà omesso nel seguito, riportando solo una breve descrizione delle caratteristiche di ogni tool.

Gli strumenti a supporto della V&V analizzati dal sondaggio sono:

- Sottomessi dagli interni
  1. Uno strumento sviluppato da SELEX-ES per il test di interfacce uomo-macchina, inizialmente sviluppato per il testing di un sistema del traffico aereo in fase di post integrazione. Collauda il sistema sia dal punto di vista funzionale sia per quanto riguarda le performance. I test sono a livello di scenario.
  2. Un tool sviluppato da SELEX-ES (precisamente dall'ex SELEX DATAMAT) per l'automatizzazione dei test. È fortemente utilizzato nei test di non-regressione.
  3. Un framework open-source per l'automazione di test funzionali e di regressione di applicazioni dotate di interfaccia grafica in Java. Gli elementi grafici vengono identificati direttamente a *runtime* durante l'esecuzione dei test script.
  4. Uno strumento open source indipendente dalla piattaforma per il testing funzionale per architetture SOA e Web Service.
  5. Un tool per lo sviluppo di test cases per il testing di applicazioni web.
  6. Uno strumento che integra l'analisi statica, flow analysis, revisione del codice, unit testing e component testing, verifica della copertura del codice sorgente, monitoraggio a runtime dell'applicazione e test di non regressione. Permette di verificare l'aderenza agli standard internazionali tra cui MISRA e DO-178B/C.
  7. Un tool, dedicato a software Java, che permette l'analisi statica, flow analysis, revisione automatizzata del codice, calcolo delle metriche, unit testing e component testing, verifica della copertura del codice sorgente, monitoraggio dell'applicazione a runtime e generazione di casi di test a partire dal "tracing" dell'utilizzo dell'applicazione in ambiente reale.
- Sottomessi dagli esterni
  1. Uno strumento di modellazione conforme alle regole dell'OMG incluso UML Testing Profile (UTP) e supporta l'esecuzione di testing di non-regressione.
  2. Una piattaforma web-based per gestire la qualità del software, e in particolare per monitoraggio di metriche software e per la rilevazione di violazioni a regole di programmazione.
  3. Uno strumento per il support al testing model-based, che supporta la generazione automatica dei test a partire da varie tipologie di modelli.

Di seguito sono mostrati i grafici che sintetizzano le risposte ottenute. Vengono mostrati due tipologie di grafici, a barre e a torta. Essi mostrano la frequenza con cui le funzionalità e caratteristiche considerate sono fornite dai tool.

In Figura 4-1 e in Figura 4-2 sono mostrate le funzionalità fornite dai tool:

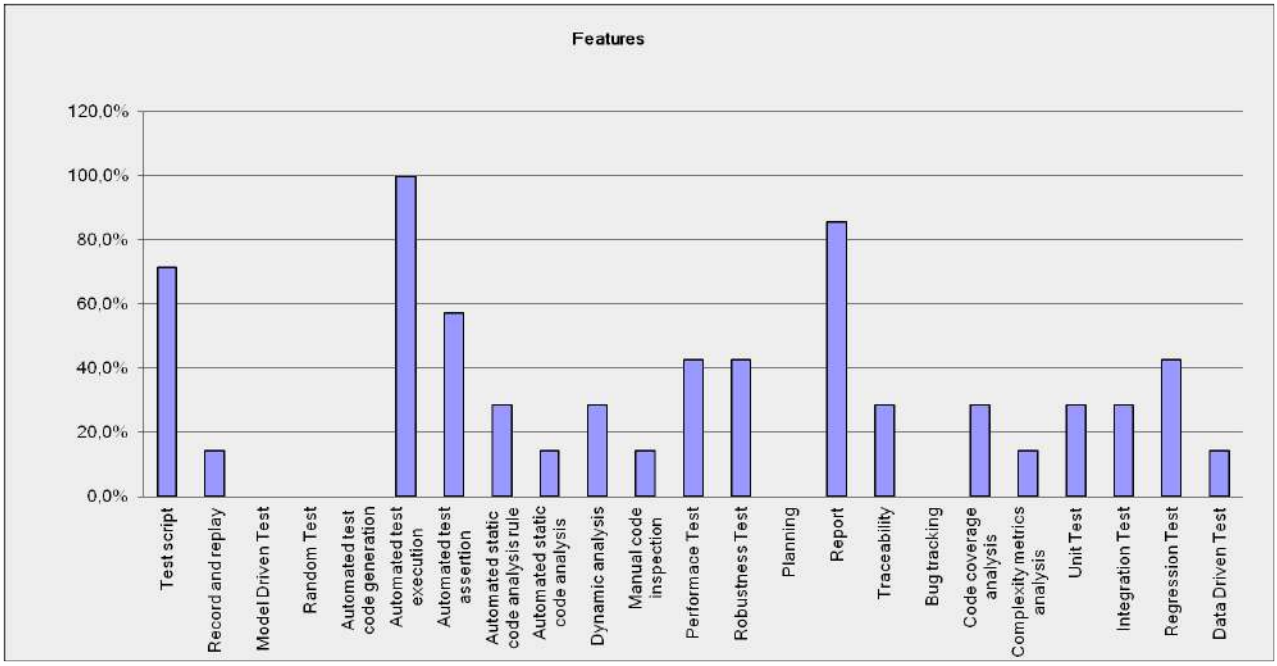


Figura 4-1. Features (interni)

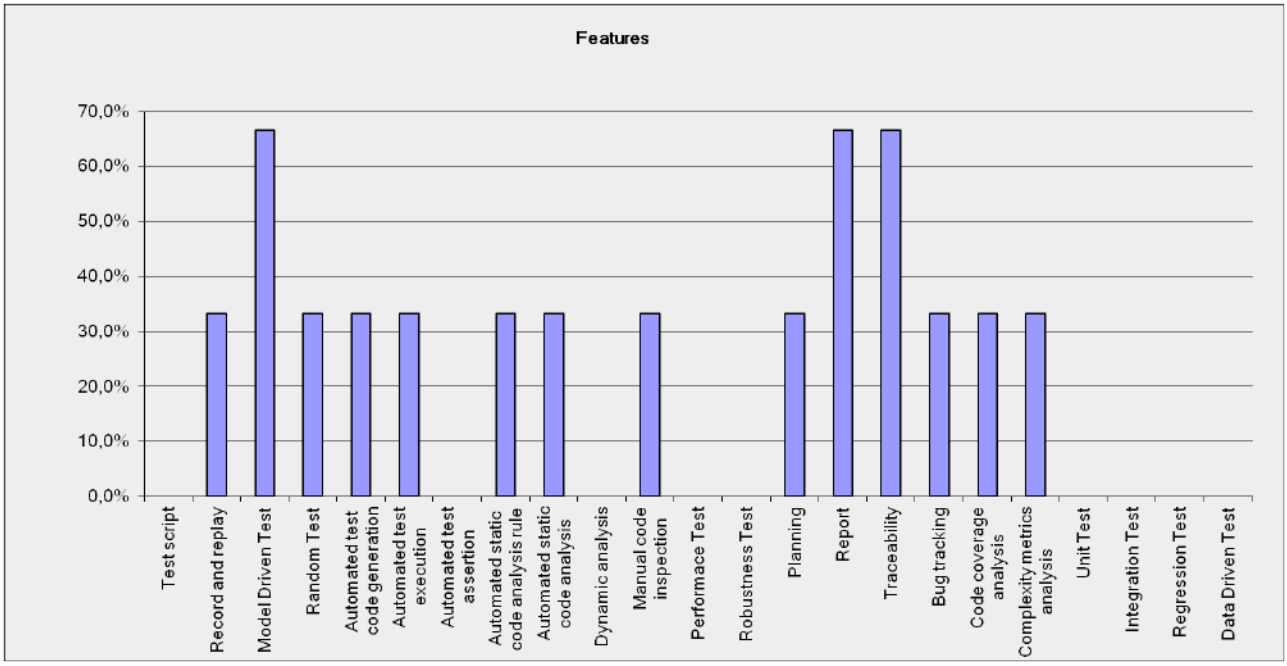


Figura 4-2. Features (esterni)

Analogamente a quanto fatto sopra, di seguito sono mostrati i grafici relativi alla tipologia di sistemi (SUT) a cui gli strumenti di testing possono essere applicati (Figura 4-3).

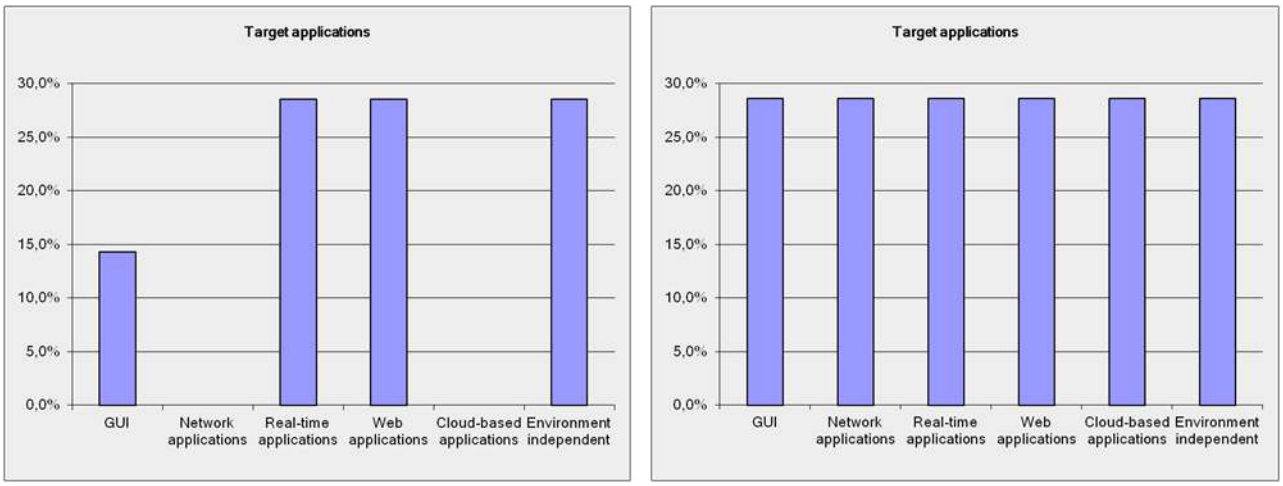


Figura 4-3. Tipologia di Sistemi applicabili (interni a destra ed esterni a sinistra).

Figura 4-4 e Figura 4-5 mostrano i linguaggi di programmazione a cui è possibile applicare un tool.

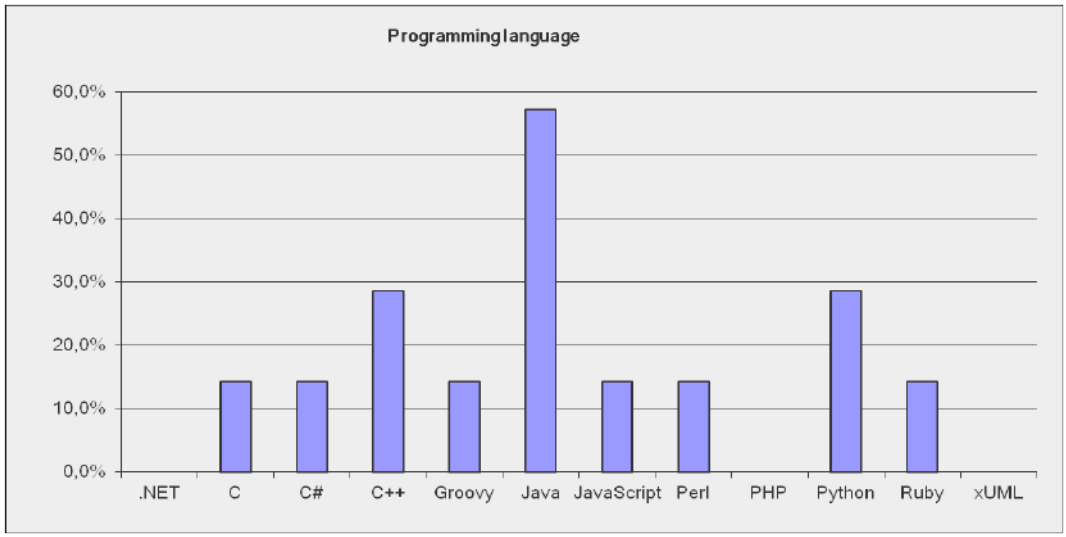


Figura 4-4. Linguaggi di programmazione supportati (interni).

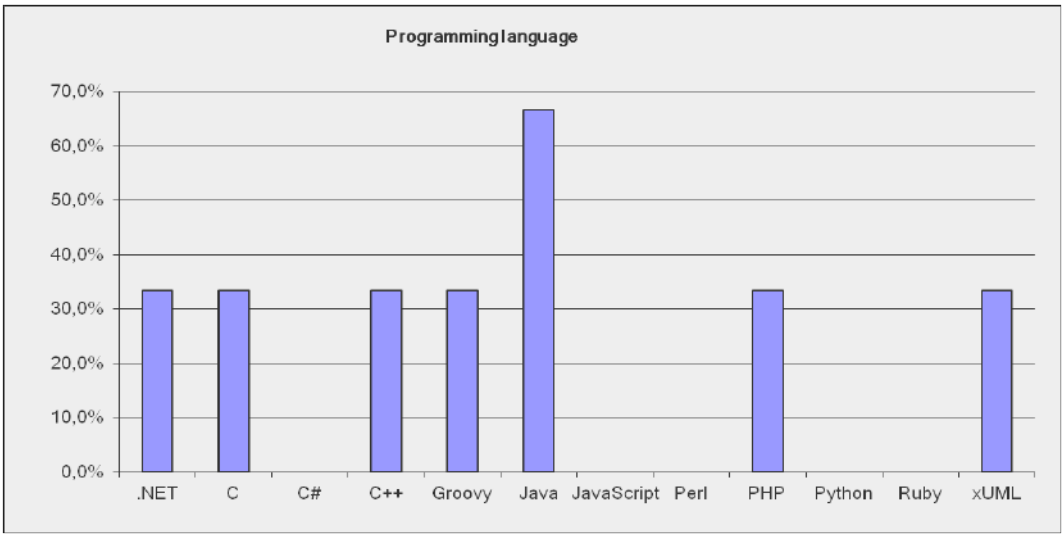


Figura 4-5. Linguaggi di programmazione supportati (esterni).

In Figura 4-6 sono mostrati i grafici che indicano la tipologia di sistemi operativi supportati dai tool.

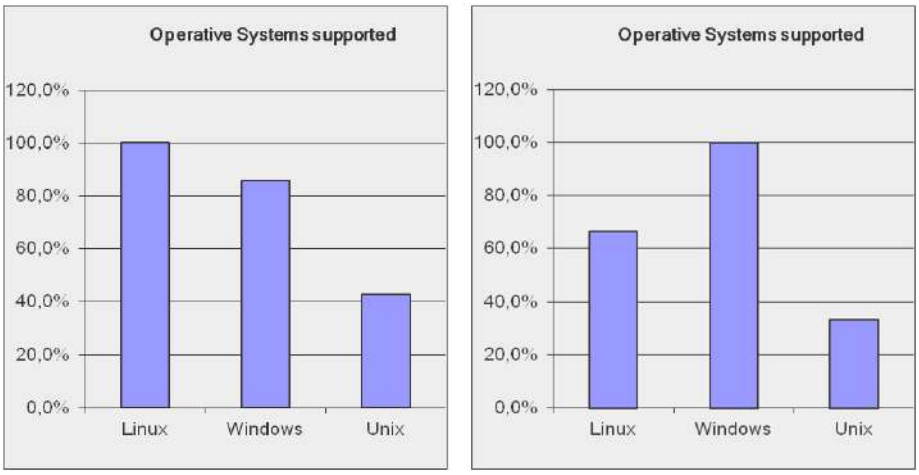


Figura 4-6. Sistemi operativi supportati (Interni a sinistra ed Esterni a destra).

Nei grafici mostrati in Figura 4-7, Figura 4-8, e Figura 4-9 sono mostrati rispettivamente la tipologia di licenza e di supporto di cui godono i tool analizzati ed infine la loro maturità.

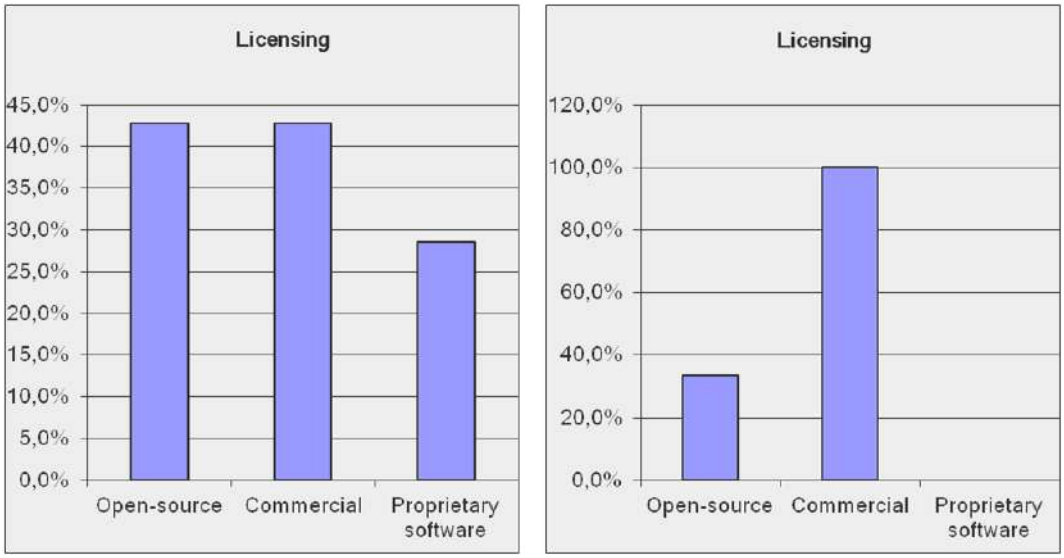


Figura 4-7. Tipologia di Licenza del tool (Interni a sinistra ed Esterni a destra).

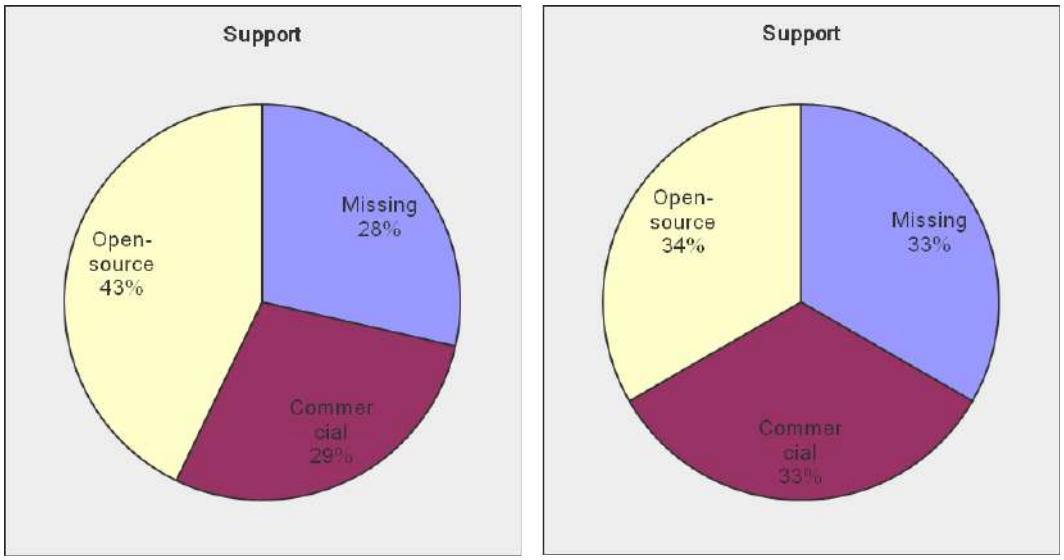


Figura 4-8. Tipologia di supporto esistente (interni a sinistra ed Esterni a destra).

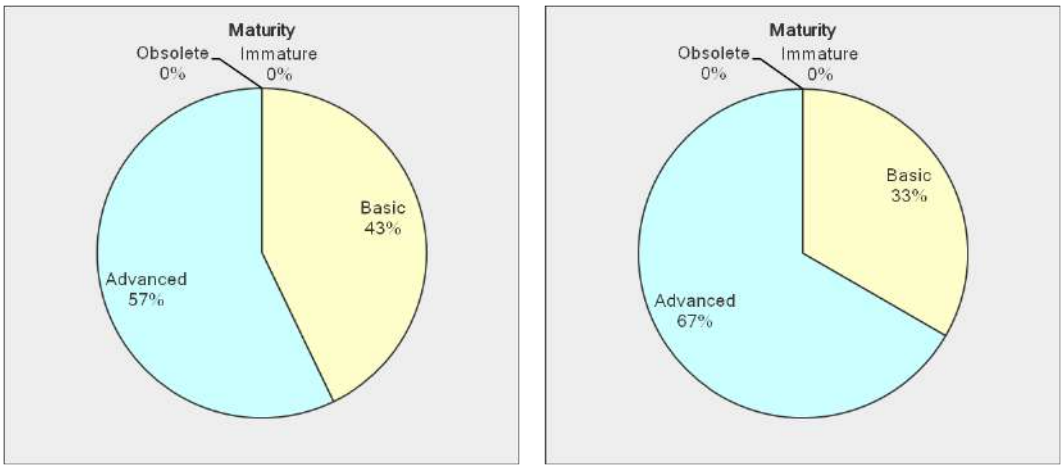


Figura 4-9. Maturità del Tool (Interni a sinistra ed Esterni a destra).

In Figura 4-11 e Figura 4-11 sono mostrati, rispettivamente, il grado di inerenza ai progetti aziendale e la difficoltà con cui essi possono essere applicati a tali progetti. Mentre in Figura 4-12 è mostrato il grado di scalabilità dei tool in sistemi complessi.

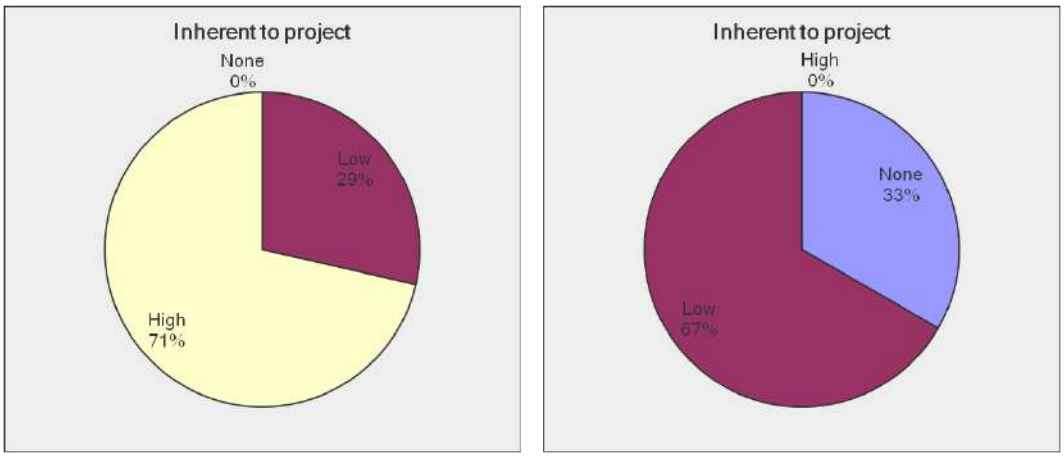


Figura 4-10. Inerenza con i Progetti Aziendali (Interi a sinistra ed Esterni a destra).

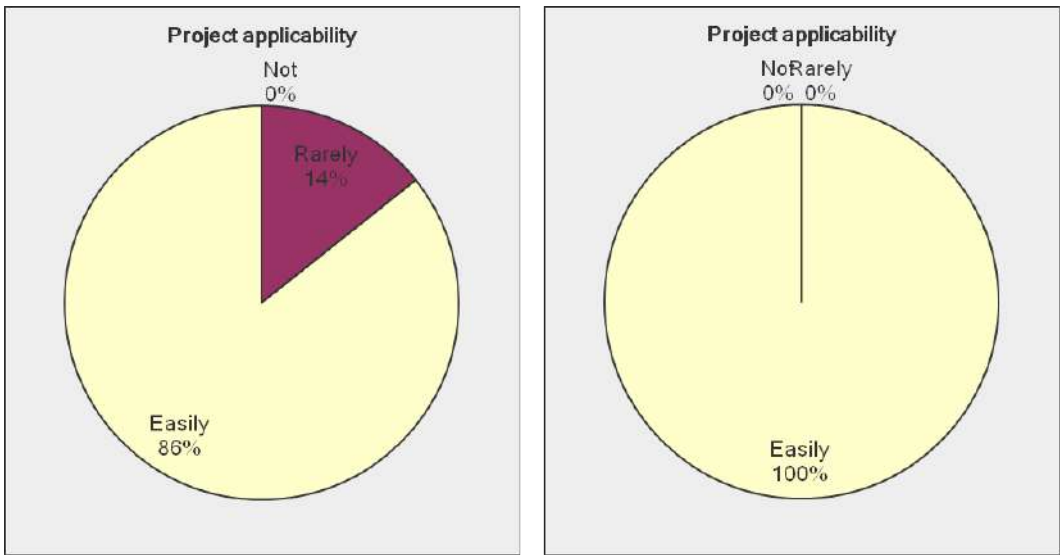


Figura 4-11. Applicabilità ai Progetti (Interni a sinistra ed Esterni a destra).

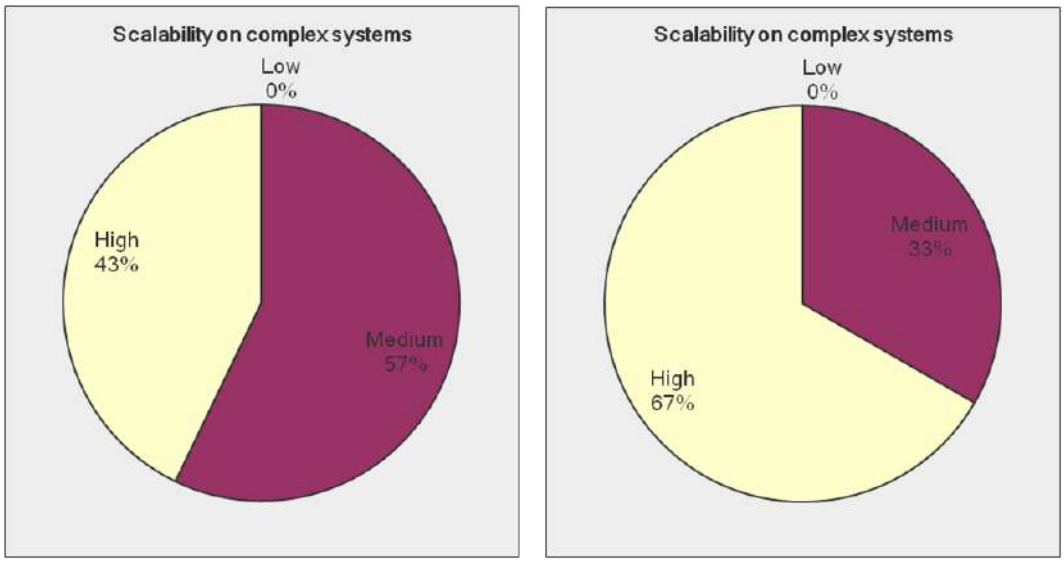


Figura 4-12. Scalabilità in sistemi complessi (Interni a sinistra ed Esterni a destra).

Infine in Figura 4-13 e Figura 4-14 sono mostrati i grafici relativi alla semplicità d'uso ed estensibilità dei tool, secondo la percezione degli sviluppatori.

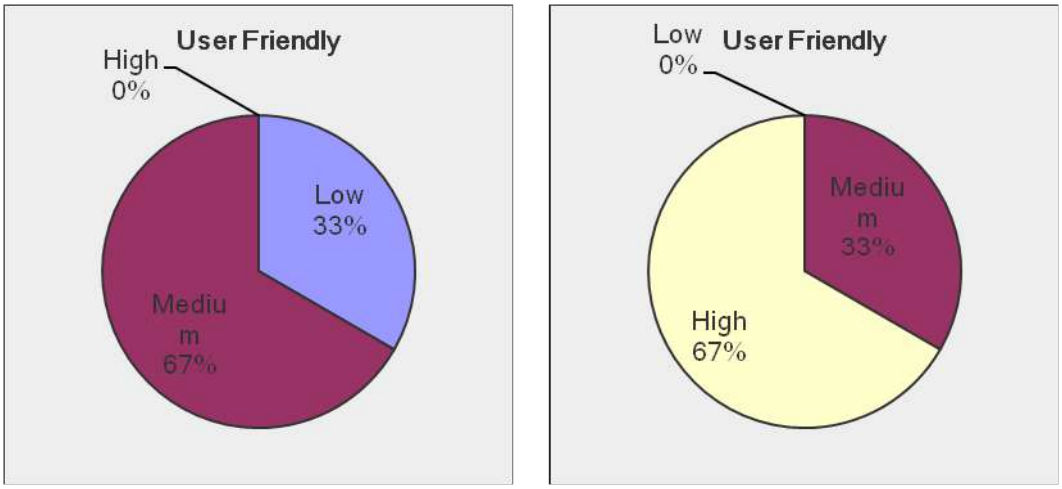


Figura 4-13. Usabilità del tool (interni a sinistra ed esterni a destra).

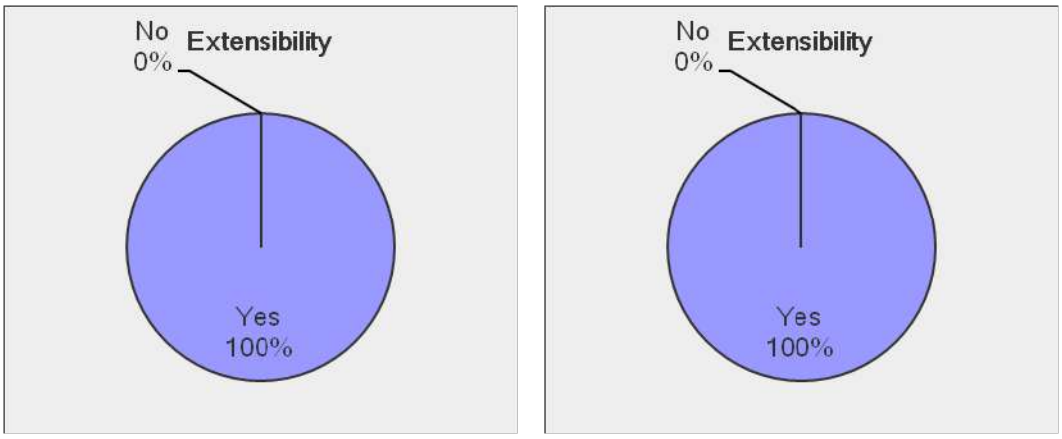


Figura 4-14. Estensibilità (interni a sinistra ed esterni a destra).

4.3 DISCUSSIONE

Dall'analisi dei risultati grezzi del sondaggio sono emerse alcune limitazioni nell'adozione di tool di V&V da parte dell'azienda. In particolare, dall'analisi delle funzionalità dei tool adottati "internamente" nell'azienda, è emerso che non vi sono soluzioni per la generazione automatica di test, quale ad esempio la generazione di test a partire da modelli del sistema (model-driven development). Infatti, le uniche soluzioni model-driven sono quelle sviluppate da “esterni”, ma non sono ancora pienamente utilizzate all’interno di SELEX-ES. Questo evidenzia il fatto che l’azienda sta guardando attivamente a queste soluzioni, ma esiste ancora un gap nella diffusione di tali tool, e che è necessario sviluppare un *know-how* interno per il loro proficuo utilizzo. Il punto di forza dei tool "interni" è risultato essere l'esecuzione automatica dei test (ad esempio ai fini dei test di regressione e dei test di qualifica) e la generazione dei report; tali aspetti sono elementi chiave nella V&V del software in grosse organizzazioni come SELEX-ES.

Un'altra funzionalità assente nei tool considerati (ad eccezione di uno dei tool utilizzati esternamente a SELEX-ES) è il supporto alla pianificazione (planning) delle attività di V&V. Questo è dovuto al fatto che la definizione delle attività di V&V è condotta senza il supporto di tool specializzati (che tengano conto, ad esempio, del costo e della copertura dei test, e dell'affidabilità stimata del software), ma viene fatta approcci classici di project management, senza tenere esplicitamente conto delle caratteristiche del software e degli esiti del test. Anche nel caso di dell'unico tool esterno in grado di supportare la pianificazione, esso fornisce la sola tracciabilità delle attività rispetto ai requisiti, senza supportare la scelta ottimale delle attività di testing. La definizione di attività di V&V tenendo conto della reliability, che è uno degli obiettivi principali del progetto SVEVIA, è potenzialmente utile a colmare questa limitazione.

Sia i tool adottati dall'azienda, sia quelli usati esternamente, sembrano fornire funzionalità per la raccolta delle metriche del software. Questo rappresenta un'opportunità per gli approcci di pianificazione che si basano sulle metriche del software. Questi tool forniscono la base per il supporto alla pianificazione da sviluppare nel progetto SVEVIA.

Infine, molti sviluppatori nell'azienda hanno enfatizzato la necessità di avere tool di testing scalabili, e che siano maturi ed estensibili, data la complessità e la dimensione dei sistemi sviluppati in SELEX-ES. Quindi questo è un requisito da tenere in grossa considerazione nello sviluppare politiche e tecniche per il planning.

5 RIFERIMENTI

Acronimo	Riferimento / Descrizione
[1]	K. Fowler: Mission-Critical and Safety-Critical Development, in IEEE Instrumentation and Measurement Magazine, Dec. 2004
[2]	A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing, 1(1):11-33, 2004.
[3]	K.S. Trivedi, "Probability and Statistics with Reliability, Queuing and Computer Science Applications," John Wiley and Sons, 2001.
[4]	R. A. Sahner, K. S. Trivedi, and A. Puliafito. Performance and Reliability analysis of Computer Systems; An Example-based Approach Using the SHARPE Software Package. Kluwer Academic Publisher, 1996.
[5]	A. P. Wood. Multistate block diagrams and fault trees. IEEE Transactions on Reliability, R-34:236-240, 1985.
[6]	W. G. Schneeweiss. Petri Nets for Reliability Modeling. LiLoLe Verlag, 1999.
[7]	J. Bechta-Dugan, S. J. Bavuso and M. A. Boyd, Dynamic fault-tree models for fault-tolerant computer systems, IEEE Transactions on Reliability, 41, pp.363-377, 1992.
[8]	A. Bobbio, G. Franceschinis, R. Gaeta and L. Portinale, Parametric Fault-Tree for the Dependability Analysis of Redundant Systems and its High Level Petri Net Semantics, IEEE Transactions Software Engineering, 29 (270-287) 2003.
[9]	D. Codetta Raiteri, G. Franceschini, M. Iacono and V. Vittorini, Repairable Fault Tree for the automatic evaluation of repair policies, in International Conference on Dependable Systems and Networks (DSN '04), (Florence, Italy), pp.659-668, IEEE Computer Society, 2004.
[10]	C.B.Almeida and K.Kanoun, Construction and Stepwise Refinement of Dependability Models, Performance Evaluation, vol. 56, 277-306, 2004.
[11]	Y. Dai, Y. Pan, X. Zou, A Hierarchical Modeling and Analysis for Grid Service Reliability, IEEE Trans. on Computers, vol. 56, 681-691, 2007.
[12]	Trivedi, K. Wang, D. Hunt, D.J. Rindos, A. Smith, W.E. Vashaw, B., Availability Modeling of SIP Protocol on IBM© WebSphere ©, Proc. of the 14th IEEE Pacific Rim Intl. Symposium on Dependable Computing, 2008, 323-330.
[13]	G. A. Hoffmann, K. S. Trivedi , M. Malek, A Best Practice Guide to Resource Forecasting for the Apache Webserver, Proc. of the 12th IEEE Pacific Rim Intl. Symposium on Dependable

	Computing, 2006,183-193.
[14]	W. E. Smith, K. S. Trivedi, L. A. Tomek, J. Ackaret, Availability analysis of blade server systems, Ibm Systems Journal, vol. 47, no. 4, 2008.
[15]	G. Ciardo and K. S. Trivedi, Decomposition Approach to Stochastic Reward Net Models, Performance Evaluation, vol. 18, 37-59, 1993.
[16]	D. Daly, W. H. Sanders, A connection formalism for the solution of large and stiff models, 34th Annual Simulation Symposium, 2001, 258-265.
[17]	Garzia, M.R., Assessing the Reliability of Windows Servers, Proc. of Dependable Systems and Networks, (DSN-2002).
[18]	Silva, G.J., Madeira, H., Experimental dependability evaluation. In Diab, H.B., Zomaya, A.Y., eds.: Dependable Computing Systems: Paradigms, Performance Issues, and Applications. Wiley (2005) 319-347.
[19]	D. Tang, R.K. Iyer, Dependability Measurement and Modeling of a Multicomputer System, IEEE Trans. on Computers, 42(1), 62-75, 1993
[20]	D.Long, A.Muir, R.Golding, A Longitudinal Survey of Internet Host Reliability, Proc. of the 14th Symposium on Reliable Distributed Systems.
[21]	Kesari Mishra, K.S. Trivedi, Model Based Approach for Autonomic Availability Management, Proc. of the Intl. Service Availability Symposium, Helsinki , Finlande, 2006, vol. 4328, 1-16
[22]	Haberkorn, M. Trivedi, K., Availability Monitor for a Software Based System, Proc. of the 10th IEEE High Assurance Systems Engineering Symposium, 2007. HASE '07, 21-328
[23]	Kalyanaraman Vaidyanathan and Kishor S. Trivedi. A comprehensive model for Software Rejuvenation. IEEE Transactions on Dependable and Secure Computing, 2(2):124-137, 2005.
[24]	A.L. Goel, K. Okumoto, A time dependent error detection rate model for software reliability and other performance measures, IEEE Transactions on Reliability, R-28(3) 206-211, 1979
[25]	A.L. Goel, Software reliability Models: Assumptions, limitations, and applicability, IEEE transactions on software engineering, SE-11, 1411-1423 (1985)
[26]	J.D. Musa, K. Okumoto, A logarithmic Poisson execution time models for software reliability mesaurement, Proc of the 7th Intl. Conference on Software Engineering, 1983, 230-237.
[27]	S.S. Gokhale, W. E.Wong, J.R. Horganc, K. S. Trivedi, An analytical approach to architecture-based software performance and reliability prediction, Performance Evaluation, vol. 58, issue 4, pp. 391-412, 2004.
[28]	K. Goseva-Popstojanova, A.P. Mathur, K.S. Trivedi, Comparison of architecture-based soft- ware reliability models, Proc. of the 12th International Symposium on Software Reliability

	Engineering (ISSRE '01), pp. 22-31, 2001.
[29]	S. Gokhale, M.R. Lyu, K.S. Trivedi, Incorporating fault debugging activities into software reliability models: A simulation approach, IEEE Trans. on Reliability, vol. 55, No. 2, pp. 281– 292, June 2006.
[30]	W.Wang, Y.Wu, M.H. Chen, An architecture-based software reliability model, Proc. of the Pacific Rim Dependability Symposium, 1999.
[31]	K. Goseva-Popstojanova and K. S. Trivedi, Architecture-based approach to reliability assessment of software systems, Performance Evaluation, vol. 45, issue 2-3, pp. 179-204, 2001.
[32]	S. Gokhale, K.S. Trivedi, Time/Structure Based Software Reliability Model, Annals of Software Engineering, vol. 8, pp. 85-121, 1999.
[33]	A. Reibman, K.S.Trivedi, Numerical transient analysis of Markov models, Computers & Operations Research, vol. 15, n. 1, pp. 19-36,1988.
[34]	S.S. Gokhale, K.S. Trivedi, Reliability prediction and sensitivity analysis based on software architecture, Proc. of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), p. 64, 2002.
[35]	N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, Brooks/Cole, 1998
[36]	S. R. Chidamber and C. F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, 476-493.
[37]	V. R. Basili, L. C. Briand, and W. L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering, 22(10), pp. 751-761, 1996.
[38]	R. Subramanyam and M. S. Krishnan, Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, IEEE Transactions on Software Engineering, Vol. 29, No. 4 (April 2003) 297-310
[39]	A. B. Binkley, Schach, S., Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In the proc. of the International Conference on Software Engineering, pp. 452 - 455, 1998.
[40]	N. Ohlsson, Alberg, H., Predicting fault-prone software modules in telephone switches, IEEE Transactions on Software Engineering, Vol. 22, No. 12, 1996, 886 - 894.
[41]	S.S. Gokhale, M.R Lyu, Regression Tree Modeling for the Prediction of Software Quality, in Proc. of the third ISSAT 1997, Anhaiem, CA.
[42]	N. Nagappan, T. Ball, A. Zeller, Mining Metrics to Predict Component Failures, In the Proc. of the 28th international conference on Software engineering (ICSE '06), 2006, 452–461.

[43]	G.Denaro, S. Morasca, M.Pezze', Deriving Models of Software Fault-Proneness, SEKE 2002.
[44]	G. Denaro, M.Pezze', An Empirical Evaluation of Fault-Proneness Models, in proc of the 24th International Conference on Software engineering, 241-251, 2002.
[45]	J. C. Munson and T. M. Khoshgoftaax. The detection of fault-prone programs. IEEE Transactions on Software Engineering, 18(5):423-33, May 1992.
[46]	T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass. Using process history to predict software quality. Computer, 31(4):66-72, Apr. 1998.
[47]	J. Nam, S. Jialin Pan, and S. Kim, S., Transer Defect Learning, Proceedings of the International conference on software engineering (2013)
[48]	S. Yamada, M. Ohba, and S. Osaki. S-Shaped Reliability Growth Modeling for Software Error Detection. IEEE Trans. on Reliability, R-32(5), pp. 475–485 (1983)
[49]	S. Gokhale, K. Trivedi, Log-logistic software reliability growth model, Proc. of the 3rd IEEE Intl. High Assurance Systems Engineering Symp., 1998
[50]	R.E. Mullen, The lognormal distribution of software failure rates: application to software reliability growth modeling. In: Proceedings 9th International Symposium on Software Reliability Engineering, pp. 134–142 (1998)
[51]	H. Okamura, T. Dohi, and S. Osaki, Em algorithms for logistic software reliability models. In Proc. 22nd IASTED International Conference on Software Engineering, pp. 263–268 (2004)
[52]	S. Yamada, T. Ichimori, and M. Nishiwaki, Optimal Allocation Policies for Testing-Resource Based on a Software Reliability Growth Model. International Journal of Mathematical and Computer Modeling, Vol. 22(10-12), 39. pp. 295-301 (1995)
[53]	C. Huang, S. Kuo, and M.R. Lyu, An Assessment of Testing-Effort Dependent Software Reliability Growth Models, IEEE Transactions On Reliability, vol. 56, no. 40. 2 (2007)
[54]	S. Yamada, H. Ohtera, and H. Narihisa, Software reliability growth models with testing effort. IEEE Transactions on Reliability, vol. R-35, pp. 19-23 (1986)
[55]	C. Huang, and M.R. Lyu, Optimal Release Time for Software Systems Considering Cost, Testing-Effort, and Test Efficiency. IEEE Transactions On Reliability, vol. 54, no. 4 (2005)
[56]	M.R. Lyu, S. Rangarajan and A.P.A. van Moorsel, Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development, IEEE Transactions on Reliability, vol. 51, no. 2 (2002)
[57]	M. R. Lyu, S. Rangarajan, A.P.A. van Moorsel, Optimization of Reliability Allocation and Testing Schedule for Software Systems, Proc. of the 8th International Symposium On Software Reliability Engineering (ISSRE '97), pp. 336-347 (1997)

[58]	C.Y. Huang, J.H.Lo, S.Y. Kuo, and M.R. Lyu, Optimal Allocation of Testing Resources for Modular Software Systems, Proceedings of the Thirteenth IEEE International Symposium on Software Reliability Engineering, pp. 129-138 (2002)
[59]	J.H. Lo, S.Y. Kuo, M.R. Lyu, and C.Y. Huang, Optimal Resource Allocation and Reliability Analysis for Component-based Software Applications. In: Proceedings of the 26th IEEE Annual International Computer Software and Applications Conference (COMPSAC), pp. 7–12 (2002)
[60]	C.Y. Huang, J.H. Lo, S.Y. Kuo, and M.R. Lyu, Optimal Allocation of Testing Resource Considering Cost, Reliability, and Testing-Effort. Tenth IEEE Pacific Rim International Symposium on Dependable Computing, pp. 103–112 (2004)
[61]	C.Y. Huang, and J.H. and Lo, Optimal Resource Allocation for Cost and Reliability of Modular Software Systems in the Testing Phase, Journal of Systems and Software, Vol. 79(5), pp. 653–664 (2006)
[62]	R.H. Hou, S.Y. Kuo, and Y.P. Chang, Efficient allocation of testing resources for software module testing based on the hyper-geometric distribution software reliability growth model, Proc. of the 7th International Symposium on Software Reliability Engineering (ISSRE '96), pp. 289-298, Oct./Nov. (1996)
[63]	W. Everett, Software Component Reliability Analysis, In: Proc. of the Symp. Application specific Systems and Software Engineering Technology (ASSET '99), pp. 204– 211 (1999)
[64]	R. Pietrantuono, S. Russo, K.S. Trivedi, Software Reliability and Testing Time Allocation: An Architecture-Based Approach, IEEE Transactions on Software Engineering, vol. 36, no. 3, pp. 323–337 (2010).
[65]	V. Almering, M. Van Genuchten, G. Cloudt, and P.J.M. Sonnemans, Using Software Reliability Growth Models in Practice, IEEE Software, vol. 24, no. 6, pp. 82-88, Nov./Dec. 2007.
[66]	C. Stringfellow and A. Amschler Andrews. 2002. An Empirical Method for Selecting Software Reliability Growth Models. Empirical Software Engineering, 7 (4) 319-343
[67]	C. Catal, and B. Diri, A systematic review of software fault prediction studies. Expert Systems with Applications , 36 (4), 2009.
[68]	T. McCabe, A complexity measure, IEEE Transactions on Software Engineering , 2 (4), 308-320, 1976.
[69]	M. Halstead, Elements of Software Science, Elsevier Science, 1977.
[70]	F.B. Abreu, and R. Carapuca, Object-oriented software engineering: Measuring and controlling the development process. Fourth international conference on software quality, 1994.
[71]	J. Bansiya, and C. Davis, A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, , 28 (1), 4-17, 2002.

[72]	M. Lorenz, and J. Kidd, Object-oriented software metrics: A practical guide, Prentice Hall Inc, 1994.
[73]	Y. Zhou, and H. Leung, H, Empirical analysis of object-oriented design metrics for predicting high and low severity faults, IEEE Transactions on Software Engineering, 32 (10), 771-789, 2006.
[74]	T. Khoshgoftaar, K. Gao, and R. Szabo, An application of zero-inflated poisson regression for software fault prediction, Twelfth international symposium on software reliability engineering, IEEE, 2001.
[75]	T. Ostrand, E. Weyuker, and R. Bell, Predicting the Location and Number of Faults in Large Software Systems, IEEE Transactions on Software Engineering , 31 (4), 340-355, 2005.
[76]	A. Stellman, J. Greene, Applied Software Project Management, O'Reilly Media, 2005
[77]	M.S Hecht, Flow Analysis of Computer Program, North-Holland, Amsterdam, 1977
[78]	A.V. Aho, R. Sethi and J.D.Ullman, Compilers: Principles, Tecniques and Tools, Addison-Wesley, Reading (Massachusetts), 1986
[79]	J. Ferrante, K.J. Ottenstein and J.D.Warren, “The Program Dependence Graph and its use in Optimization”, ACM Trans. Programming Languages and Systems, pp.319-349, 1987
[80]	D. Kuck, The Structure of Computers and Computations, vol. 1, Wiley, New York, 1978
[81]	M. Wolfe and U. Banerjee, “Data Dependence and its application to Parallel Processing”, International Journal of Parallel Programming, pp. 137-178, 1987
[82]	R. Allen, K. Kennedy, C. Porterfield and J. Warren, “Conversion of control dependence to data dependence”, Proc. 10th ACM Symposium on Principles of Programming Languages, 1983
[83]	Weiser, Mark. "Program slicing." Proc. 5th International Conference on Software Engineering, 1981
[84]	Gallagher, Keith Brian, and James R. Lyle. "Using program slicing in software maintenance." IEEE Transactions on Software Engineering, vol. 17, no. 8, pp. 751-761, 1991
[85]	G. Sierksma. Linear and Integer Programming: Theory and Practice. Monographs and Textbooks in Pure and Applied Mathematics. Marcel Dekker, 1996
[86]	P. Cousot, R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", Proc. 4th Symposium on Principles of Programming Languages, pp. 238-252, 1977
[87]	P. Cousot, R. Cousot. "Semantic analysis of communicating sequential processes", Proc. 7th International Colloquium on Automata, Languages and Programming, LNCS 85, pp. 119-133,

	1980
[88]	P. Cousot, R. Cousot. "Parsing as Abstract Interpretation of Grammar Semantics". Theoret. Comput. Sci., Vol. 290, n. 1, 2003, pp.531-544
[89]	P. Cousot, R. Cousot. "Inductive Definitions, Semantics and Abstract Interpretation". Proc. 19th Symposium on Principles of Programming Languages, 1992, pp. 83-94
[90]	P. Cousot. "Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation". Theoret. Comput. Sci., Vol. 277, n. 1-2, 2002, pp. 47-103
[91]	P. Cousot. "Partial Completeness of Abstract Fixpoint Checking, invited paper". Proc. 4th Int. Symp. on Abstraction, Reformulation, and Approximation, 2000, pp. 1-25
[92]	B. Pierce. "Types and Programming Languages". MIT Press, 2002
[93]	E. Clarke, O. Grumberg, D. Long. "Model Checking and Abstraction". ACM Transactions on Programming Languages and Systems, Vol. 16, n. 5, 1994, pp. 1512-1542
[94]	S. Graf, H. Saïdi. "Verifying Invariants Using Theorem Proving". Proc. 8th Int. Conf. Computer Aided Verification, pp. 196-207, 1996
[95]	F. Ranzato, F. Tapparo. "Strong Preservation of Temporal Fixpoint-Based Operators by Abstract Interpretation". Proc. 7th Int. Conf. Verification, Model Checking, and Abstract Interpretation, 2006 pp. 332-347
[96]	P. Cousot, R. Cousot. "Systematic Design of Program Transformation Frameworks by Abstract Interpretation". Proc. 29th Symposium on Principles of Programming Languages, 2002, pp. 178-190
[97]	R. Giacobazzi, E. Quintarelli. "Incompleteness, Counterexamples and Refinements in Abstract Model-Checking". Proc. 8th Int. Symp. Static Analysis, pp. 356-373, 2001
[98]	M. D. Preda, R. Giacobazzi. "Control Code Obfuscation by Abstract Interpretation", Proc. 3rd IEEE Int. Conf. Software Engineering and Formal Methods, 2005
[99]	D. Monniaux. "Abstract interpretation of probabilistic semantics", Proc. 7th Int. Symp. Static Analysis, pp. 322-339, 2000
[100]	P. Cousot, R. Cousot. "Static determination of dynamic properties of programs". Proc. 2nd Int. Symp. on Programming, 1976, pp. 106-130
[101]	P. Cousot, R. Cousot. "Compositional and Inductive Semantic Definitions in Fixpoint, Equational, Constraint, Closure-condition, Rule-based and Game-Theoretic Form". Proc. 7th Int. Conf. Computer Aided Verification, pp. 293-308, 1995.
[102]	G. Filè, R. Giacobazzi, F. Ranzato. "A Unifying View on Abstract Domain Design". ACM

	Computing Surveys, Vol. 28, n. 2, 1996, pp. 333-336
[103]	P. Cousot, R. Cousot. “Constructive versions of Tarski's fixed point theorems”. Pacific J. Math., Vol. 82, n. 1, 1979, pp. 43-57
[104]	J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization”, Proc. 24th International Conference on Software Engineering, 2002, pp. 467–477
[105]	A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, “Statistical debugging of sampled programs”, Advances in Neural Information Processing Systems 16, MIT Press, 2003, pp. 9–11
[106]	B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation”, Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005, pp. 15–26
[107]	Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, “Capturing propagation of infected program states”, Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2009, pp. 43–52
[108]	M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, “Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria”, Proc. 16th International Conference on Software Engineering, 1994, pp. 191–200
[109]	Martin Burger, Replaying and Isolating Failure-Inducing Program Interactions. PhD Thesis, Saarland University, 2011
[110]	A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input”, IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 183–200, 2002
[111]	G. Xu, A. Rountev, Y. Tang, and F. Qin, “Efficient checkpointing of Java software using context-sensitive capture and replay”, Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2007, pp. 85–94
[112]	S. Artzi, S. Kim, and M. D. Ernst, “ReCrash: making software failures reproducible by preserving object states”, Proc. 22nd European Conference on Object-Oriented Programming, 2008, pp. 542–565
[113]	M. Burger and A. Zeller, “Replaying and isolating failing multi-object interactions”, Proc. International Workshop on Dynamic Analysis, 2008, pp. 71–77
[114]	A. Orso, S. Joshi, M. Burger, and A. Zeller, “Isolating relevant component interactions with JINSI”, Proc. International Workshop on Dynamic Analysis, 2006, pp. 3–10
[115]	C. Liu, X. Yan, L. Fei, J. Han, S.P. Midkiff. "SOBER: Statistical Model-based Bug Localization", ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, 2005, pp. 286-295

[116]	Beizer, B., Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons Inc., 1995
[117]	Pezze, M., Young, M., Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons Inc., 2007
[118]	Lewis, W.E., 2008. Software Testing Continuous Quality Improvement, 3rd ed. Auerbach Publications, Taylor & Francis Group, Boca Raton, FL.
[119]	Duran, J.W., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Transactions on Software Engineering 10 (7), 438–444, 1984
[120]	Weyuker, E.J., Jeng, B., "Analyzing partition testing strategies". IEEE Transactions on Software Engineering, vol. 17, no. 7, pp. 703–711, 1991
[121]	Ostrand , T.J., Balcer, M.J., "The category-partition method for specifying and generating functional tests". Communications of the ACM, vol. 31, no. 6, pp. 676-686, 1988
[122]	Hemmati, H., Arcuri, A., Briand, L., "Achieving scalable model-based testing through test case diversity". ACM Transactions on Software Engineering and Methodologies, vol. 22, no. 1, 2013
[123]	Hamlet, R. 1994. Random testing. Encyclopedia of Software Engineering, Wiley, pp. 970-978, 1994
[124]	Jones, J.A. and Harrold, M.J., "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage". IEEE Transactions on Software Engineering, vol. 29, 195-209, 2003
[125]	Leon, D. and Podgurski, A., "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases". Proc. 14th IEEE International Symposium on Software Reliability Engineering, pp. 442-456, 2003
[126]	Ma, X.Y., Sheng, B.K. and Ye, C.Q. "Test-Suite Reduction Using Genetic Algorithm". Advanced Parallel Processing Technologies, pp. 253-262, 2005.
[127]	Cartaxo, E.G., Machado, P.D.L. and Neto, F., "On the use of a similarity function for test case selection in the context of model-based testing". Software Testing, Verification and Reliability, vol. 21, no. 2, pp.75-100, 2011.
[128]	Hemmati, H., Briand, L., Arcuri, A. and Ali, S., "An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study". Proc. 18th ACM International Symposium on Foundations of Software Engineering, pp. 267-276, 2010.

[129]	Ledru, Y., Petrenko, A. and Boroday, S., "Using String Distances for Test Case Prioritisation". Proc. 24th IEEE/ACM International Conference on Automated Software Engineering, pp. 510-514, 2009.
[130]	Poulding, S., Clark, J.A., "Efficient software verification: statistical testing using automated search". IEEE Transactions on Software Engineering, vol. 36, no. 6, pp. 763–777, 2010
[131]	Frankl, P.G., Hamlet, R.G., Littlewood, B., Strigini, L., "Evaluating testing methods by delivered reliability". IEEE Transactions Software Engineering, vol. 24, no. 8, pp. 586–601, 1998.
[132]	Whittaker, J.A., Thomason, M.G., "A Markov chain model for statistical software testing". IEEE Transactions on Software Engineering, vol. 20, no. 10, 1994.
[133]	Trammell, C., "Quantifying the reliability of software: statistical testing based on a usage model". Proc. 2nd IEEE International Software Engineering Standards Symposium, pp. 208–218, 1995
[134]	Pietrantuono, R., Russo, S., Trivedi, K.S., "Software reliability and testing time allocation: an architecture-based approach". IEEE Transactions on Software Engineering, vol. 36, no. 3, pp. 323–337, 2010
[135]	H. D. Mills, M. Dyer, and R. C. Linger, “Cleanroom software engineering,” IEEE Software, vol. 4, no. 5, pp. 19–24, 1987.
[136]	J. D. Musa, “Software-reliability-engineered testing,” Computer, vol. 29, no. 11, pp. 61–68, 1996.
[137]	R. W. Selby, V. R. Basili, and F. T. Baker, “Cleanroom software development: An empirical evaluation,” IEEE Trans. Software Eng., vol. 13, no. 9, pp. 1027–1037, 1987.
[138]	P. A. Currit, M. Dyer, and H. D. Mills, “Certifying the reliability of software,” IEEE Trans. Software Eng., vol. SE-12, no. 1, pp. 3–11, 1986.
[139]	R. H. Cobb and H. D. Mills, “Engineering software under statistical quality control,” IEEE Software, vol. 7, no. 6, pp. 44–54, 1990.
[140]	R. C. Linger and H. D. Mills, “A case study in cleanroom software engineering: The IBM Cobol structuring facility,” in Proc. Comput. Software Appl. Conf., pp. 10–17, 1988
[141]	J.H.Poore, H. D. Mills, S. L. Hopkins and J. A. Whittaker, "Cleanroom Reliability Manager: A Case Study Using Cleanroom With Box Structures ADL", Software Eng. Technol. Inc. Tech. Rep. CDRL 1880, 1990.

[142]	M. R. Lyu, Ed., Handbook of Software Reliability Engineering, IEEE Computer Society Press and McGraw-Hill, 1996.
[143]	L. Strigini and B. Littlewood, "Guidelines for Statistical Testing", Technical Report PASCON/WO6-CCN2/TN12, 1997
[144]	L. Madani, C. Oriat, I. Parissis, J. Bouchet, and L. Nigay, “Synchronous testing of multimodal systems: An operational profile-based approach,” in Proc. 16th IEEE Int. Symp. Software Rel. Eng., pp. 325–334, 2005
[145]	T.Y. Chen, F.-C. Kuo, and H. Liu, “Application of a failure driven test profile in random testing,” IEEE Trans. Rel., vol. 58, no. 1, pp. 179–192, 2009.
[146]	T.Y.Chen, F.-C.Kuo, and H.Liu, “Distributing test cases more evenly in adaptive random testing,” J. Syst. Software, vol. 81, no. 12, pp. 2146–2162, 2008.
[147]	K. Y. Cai, “Optimal software testing and adaptive software testing in the context of software cybernetics,” Inf. Software Technol., vol. 44, no. 14, pp. 841–855, 2002.
[148]	K. Y. Cai, B. Gu, H. Hu, and Y. C. Li, “Adaptive software testing with fixed-memory feedback,” J. Syst. Software, vol. 80, no. 8, pp. 1328–1348, 2007.
[149]	K. Y. Cai, Y. C. Li, and K. Liu, “Optimal and adaptive testing for software reliability assessment,” Inf. Software Technol., vol. 46, no. 15, pp. 989–1000, 2004.
[150]	Weyuker, E., and Vokolos, F., "Experience with performance testing of software systems: issues, an approach, and case study". IEEE Transactions on Software Engineering, vol. 26, no. 12, 1147–1156, 2000
[151]	Denaro, G., Polini, A., Emmerich, W., "Early performance testing of distributed software applications". Proc. 4th International Workshop on Software and Performance, 2004
[152]	Liu, Y., Gorton, I., Liu, A., Jiang, N., and Chen, A.S., "Designing a test suite for empirically-based middleware performance prediction". Proc. 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, 2002
[153]	A.Avritzer, and E.J.Weyuker, "The role of modeling in the performance testing of e-commerce applications. IEEE Transactions Software Engeneering" , vol. 30, no. 12, 1072–1083, 2004
[154]	Koziolek, H., "Performance Evaluation of Component-based Software Systems: A Survey". Performance Evaluation , 67 (8), 634– 658, 2010

[155]	Becker, S., Grunske, L., Mirandola, R., and Overhage, S.. "Performance prediction of component-based systems: A survey from an engineering perspective". Architecting Systems with Trustworthy Components, pp. 169-192, 2006
[156]	IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, IEEE CS, 1990.
[157]	Duraes, J., and Madeira, H., "Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation". Proc. Pacific Rim Intl. Symp. on Dependable Computing , pp. 201-209, 2002
[158]	Natella, R. (2011). Achieving Representative Faultloads in Software Fault Injection. PhD thesis, Università degli Studi di Napoli Federico II.
[159]	Johansson, A., Suri, N., and Murphy, B., "On the impact of injection triggers for OS robustness evaluation". Proc. International Symposium on Software Reliability Engineering, 2007.
[160]	Sarbu, C., Johansson, A., Suri, N., & Nagappan, N., "Profiling the operational behavior of OS device drivers". Empirical Software Engineering, vol. 15, no. 4, 2009
[161]	Cohn, M., Succeeding with Agile: Software Development Using Scrum. Addison-Wesley, 2009
[162]	Johansson, A., Suri, N., and Murphy, B., "On the selection of error model(s) for OS robustness evaluation". Proc. Intl. Conf. on Dependable Systems and Networks, pp. 502–511, 2007.
[163]	Hsueh, M.-C., Tsai, T., and Iyer, R. (1997). Fault injection techniques and tools. Computer , 30 (4), 75-82.
[164]	Koopman, P., and DeVale, J., "The exception handling effectiveness of POSIX operating systems". IEEE Trans. on Software Engineering , vol. 26, no. 9, 2002
[165]	Koopman, P., Sung, J., Dingman, C., Siewiorek, D., & Marz, T., "Comparing operating systems using robustness benchmarks". Proc. International Symposium on Reliable Distributed Systems, 1997
[166]	IEEE Standard for Information Technology-Portable Operating System Interface (POSIX). IEEE Std 1003.1b, IEEE CS, 1993