LABORATORIO PUBBLICO-PRIVATO COSMIC

PROGETTO SVEVIA

"METODOLOGIE E TECNICHE INNOVATIVE PER LA VERIFICA E VALIDAZIONE DEL SOFTWARE PER SISTEMI COMPLESSI NEAR-REALTIME"

AUTORIZZATO E FINANZIATO DAL MINISTERO DELL'UNIVERSITÀ E DELLA RICERCA
DECRETO DEL DIRETTORE GENERALE MIUR PROT.660/RIC. DEL 08/10/2012
SU DOMANDA PON02_00669 – PROGETTO PON02_00485_3487758

DELIVERABLE D2.2

DEFINIZIONE DI MODELLI DI ALLOCAZIONE DELLE RISORSE DI V&V PER SISTEMI COMPLESSI NEAR-REALTIME

AUTORI E APPROVAZIONI

Nome / Cognome	Partner
R. PIETRANTUONO	CINI
G. CARROZZA	SESM
M. CINQUE	CINI
D. COTRONEO	DIETI
S. RUSSO	CINI
V. VITTORINI	CINI
A. CILARDO	CINI
P. TRAMONTANA	CINI
R. NATELLA	CINI
L. DE SIMONE	CINI
U. GIORDANO	CINI
A. MAZZEO	DIETI
M. LOFFREDA	SESM
A.R. FASOLINO	CINI
A. PERON	DIETI
E.D. CAPUANO	CINI

Approvato da	Partner
S. RUSSO	CINI
M. LOFFREDA	SESM
A. JARRE	SELEX ES
D. COTRONEO	DIETI

INDICE

INDICE DELLE FIGURE

INDICE DELLE TABELLE

1 INTRODUZIONE

L'evoluzione dei sistemi di elaborazione, attuata nell'ultimo ventennio, ha portato alla nascita di sistemi ed applicazioni sempre più funzionali e performanti, a cui non sempre è corrisposto un eguale miglioramento in termini di affidabilità. Ciò è particolarmente rilevante nell'ambito dei sistemi cosiddetti "*dependable*[1]", ossia quei sistemi impiegati in contesti in cui è necessario garantire il rispetto di requisiti quali, ad esempio, la sicurezza e la disponibilità. Appartengono a questa categoria i sistemi cosiddetti "critici per missione[2]", oggetto del progetto SVEVIA, i cui malfunzionamenti possono comportare pesanti conseguenze in termini di sicurezza delle persone, dell'ambiente e/o di beni materiali, così come gravi perdite economiche. Un sistema siffatto è sottoposto a stringenti requisiti di affidabilità, il cui livello è commisurato a quanto critica sia la sua applicazione [2]. Quanto più un sistema è complesso e critico, tanto più è oneroso verificare che esso soddisfi i livelli di qualità richiesti; in questi casi, le attività di V&V possono incidere per più del 50% sui costi totali del progetto [3]. Questo è l'effetto, da un lato, della carenza di metodologie e strumenti a supporto del miglioramento dell'affidabilità dei sistemi critici; dall'altro, dell'incapacità da parte di molte realtà d'impresa, di scegliere, adottare e pianificare strategie di V&V adeguate alla propria *mission*. Ciò è quello che il progetto SVEVIA si propone di supportare.

Ad oggi, molte delle attività nell'ambito di un processo di V&V sono spesso condotte con un approccio poco oggettivo e ripetibile. Ad esempio, scelte strategiche sulla pianificazione delle attività di V&V, o sull'opportunità di adottare tecniche che non siano puramente di test funzionale, frequentemente sono operate sulla base di considerazioni fatte sui requisiti, o su caratteristiche del software, o sulla disponibilità di personale competente, oppure basandosi sulla sola esperienza dei tester.

In contesti industriali altamente specializzati, come nel caso di SELEX ES (proponente industriale di questo progetto), è importante assicurarsi che decisioni e criteri per la conduzione delle attività di V&V non siano lasciate ad "opinioni" generiche e soggettive, che per natura possono portare a valutazioni inesatte o addirittura totalmente sbagliate. Ad esempio, un test manager non potrà affermare con certezza che un componente sia qualitativamente superiore ad un altro, o meno soggetto ad errori (destinando, conseguentemente, più *effort* alla relativa attività di V&V) semplicemente basandosi sull'intuizione.

Da ciò nasce la necessità di un approccio sistematico al problema della corretta pianificazione del processo di test, e più in generale di V&V, grazie al quale determinare – ancor prima che abbiano inizio le attività o durante le attività stesse - quante risorse spendere per migliorare la qualità ciascun modulo software. A tal fine è indispensabile dapprima identificare tecniche di V&V adeguate alle caratteristiche e ai requisiti dei sistemi che si realizzano e, successivamente, razionalizzare i processi decisionali di allocazione delle risorse, così da consentire l'avviamento di iniziative che, seppur costose, potrebbero produrre dei benefici significativi, limitando lo spreco di risorse per attività non necessarie. Dunque, il primo passo verso questa direzione è selezionare tecniche di V&V che consentano di raggiungere più rapidamente gli obiettivi di

[1] Fiducia nel corretto funzionamento del sistema [1].
[2] Esempi classici sono i sistemi per il controllo del traffico aereo, così come sistemi per il controllo di processi industriali, prodotti dalla proponente industriale SELEX ES.

qualità desiderati per un sistema e poi operare un'attenta pianificazione, prestando particolare attenzione al *trade-off* tra l'*effort* da profondere ed il livello di qualità atteso.

1.1 Scopo Del Documento

L'obiettivo del documento è illustrare alcuni criteri adottati in SVEVIA per supportare il processo decisionale, che guideranno i *team* di V&V nella pianificazione delle attività nell'ambito dei sistemi critici, oggetto del progetto SVEVIA (cfr. con D1.2: "Sviluppo ed analisi dei processi, delle metriche e degli standard aziendali per lo sviluppo software", che descrive il contesto industriale della proponente SELEX ES).

In particolare, saranno dapprima introdotte brevemente le tecniche di V&V identificate nell'ambito della metodologia SVEVIA; successivamente, il focus si sposterà sulle tecniche di ottimizzazione della "*effort allocation*", il cui obiettivo è supportare il team nella scelta della pianificazione ottimale delle attività di V&V, onde valutare i conseguenti costi e benefici. Per mostrare le potenzialità di tali criteri, ne sarà illustrata l'applicazione preliminare a sottosistemi utilizzati in domini critici, quali quello militare e navale, nel contesto industriale di SELEX ES.

1.2 Struttura Del Documento

Il documento si articola in ulteriori cinque sezioni. La sezione 2 introduce le tecniche di V&V che sono identificate nel progetto SVEVIA come strategie rilevanti per il miglioramento della qualità dei sistemi considerati. La sezione 3 richiama possibili metodi esistenti in letteratura per la pianificazione delle attività di V&V, utilizzabili in stadi diversi del processo. Tale sezione fornirà il background necessario per le sezioni successive. Le sezioni 4, 5 e 6 illustreranno tre metodi di pianificazione che definiscono nuovi criteri e modelli per il supporto alle decisioni nelle fasi di V&V. In particolare: per l'allocazione dinamica delle risorse di test (sezione 4), per l'identificazione dei moduli critici di un sistema, per la conseguente prioritizzazione delle attività di V&V (sezione 5) e per l'ottimizzazione delle attività di "*code sanitization*", a valle dell'applicazione di tecniche di analisi statica automatica (sezione 6).

1.3 ACRONIMI E TERMINOLOGIA

Acronimo	descrizione
V&V	Verification & Validation
SRGM	*Software Reliability Growth Models*
Mvf	Mean Value Function
AIC	Akaike's Information Criterion
GoF	Goodness-of-fit
LoC	Line of code
TEF	Testing Effort Function
MTTF	Mean Time To Failure
MTTR	Mean Time to Repair
MTBF	Mean Time Between Failure
KS	Kolmogorov- Smirnov
CSCI	Computer Software Configuration Item
BOH	Bohrbug
MAN	Mandelbug
UNK	Unknown
Pr	Precision
Re	Recall
F	F-measure
FP	False Positive
FN	False Nagative
SVM	Support Vector Machines
SVR	Support Vector Regression
R^2	Coefficiente di determinazione
Adj – R^2	Adjusted R^2

Definizioni

Defect: È una imperfezione o l'assenza di qualche proprietà del prodotto, che lo rende non rispondente ai requisiti ed alle specifiche per cui è stato realizzato, tanto da far nascere l'esigenza di ripararlo o addirittura sostituirlo (*standard IEEE 1044-2009, 2009*). In accordo a tale standard, un Fault è una manifestazione di un errore (inteso come "*mistake, misconception, misunderstanding dello sviluppatore*") nel software; esso è un sottotipo del supertipo *defect.* Un difetto è un *fault* quando viene rilevato durante l'esecuzione del software (causando un fallimento).

In questo documento i termini *defect* e *fault* sono stati utilizzati come sinonimi, in quanto si assume che abbiano la stessa accezione.

***Dependablility*:** Può essere definita come l'attendibilità di un sistema, cioè il grado di fiducia che può essere ragionevolmente riposto nei servizi che esso offre. È la garanzia di funzionamento di un sistema di calcolo, che include attributi di affidabilità, disponibilità, sicurezza e protezione.

***Machine learning*:** Branca dell'Intelligenza Artificiale, che si occupa dello sviluppo di algoritmi e tecniche che consentono ai *computer* di apprendere, mediante degli esempi, la realtà circostante.

Metrica: Una misura quantitativa del grado in cui un Sistema, un *component* o un processo possiede un dato attributo (*IEEE 729-1993, 1993*) e (*IEEE 610.12*)

2 ATTIVITÀ DI V&V NELL'AMBITO DEL PROGETTO SVEVIA

Durante le attività di studio del presente progetto, i cui risultati sono esposti nel documento di progetto "D1.1: Studio ed analisi delle tecniche e degli strumenti di V&V", sono state passate in rassegna le più importanti tecniche e gli strumenti associati di V&V del software, nonché le principali politiche di gestione delle risorse.

Nel seguito sono riportate brevemente alcune delle tecniche di test (funzionale e non funzionale), di analisi (statica e dinamica) e criteri di utilizzo ritenuti rilevanti per i sistemi SELEX ES, su cui ci si soffermerà in questa fase del progetto.

Oltre all'usuale test funzionale (che potrà essere basato su specifiche testuali o su modelli), i requisiti di criticità dei sistemi considerati nel progetto richiedono la verifica di requisiti non funzionali. Delle tecniche studiate, sono state ritenute rilevanti due tipologie di test, che mirano, rispettivamente, a migliorare l'affidabilità del sistema e la sua capacità di tollerare eventuali guasti non rilevati durante il test. Esse sono il "*reliaiblity testing*" ed il "*fault-injection testing*". Dal punto di vista dell'analisi, invece, ci si focalizzerà sull'utilizzo di tecniche di analisi statica automatica (ASA).

Il *reliability testing* mira ad ottenere una alta affidabilità operazionale, intesa come la "probabilità di non fallire durante la fase operazionale" – per tale motivo, talvolta viene denominato "*operational testing*", o, più generalmente, "*operational profile based testing*". Di conseguenza, questa tipologia di test necessita di particolari accorgimenti nella generazione dei casi di test. Il test operazionale è basato sull'idea che, poiché durante l'esecuzione del sistema alcune funzionalità saranno più esercitate di altre, su di esse dovrà essere effettuato un test più approfondito per massimizzare la reliability in fase operazionale. Ciò non necessariamente corrisponde a scovare e rimuovere più difetti. Ed infatti, dati due prodotti software A e B, con A contenente più difetti di B, se si riscontra che a tempo di esecuzione i difetti di A hanno una frequenza di attivazione molto bassa rispetto ai ridotti difetti di B (che hanno frequenza di attivazione maggiore), si potrà constatare che in fase operazionale A è più "affidabile" di B.

Dunque, l'idea è quella di cercare nel sistema i difetti che a tempo di esecuzione hanno un maggiore impatto sull'affidabilità (ci si riferisce alla "*size*" di un fallimento come il numero di input che, attivando il difetto, causano il fallimento del sistema). Il test operazionale può essere utilizzato sia come metodo di "stima" dell'affidabilità (ossia, un test condotto senza rimuovere, in prima istanza, i difetti, ma solo osservando l'esito dei singoli casi di test) o per migliorare l'affidabilità (ossia, rimuovendo i difetti man mano che vengono rilevati – nel qual caso la stima dell'affidabilità è meno accurata).

Per condurre tale test è necessario pertanto avere una stima del profilo di utilizzo del sistema, il cosiddetto profilo operazionale, che rappresenta la caratterizzazione quantitativa di come verrà usato il software [31]. Esso è costruito assegnando delle probabilità di occorrenza alle funzionalità del sistema, oppure a gruppi di funzionalità (sintesi in Figura 1).

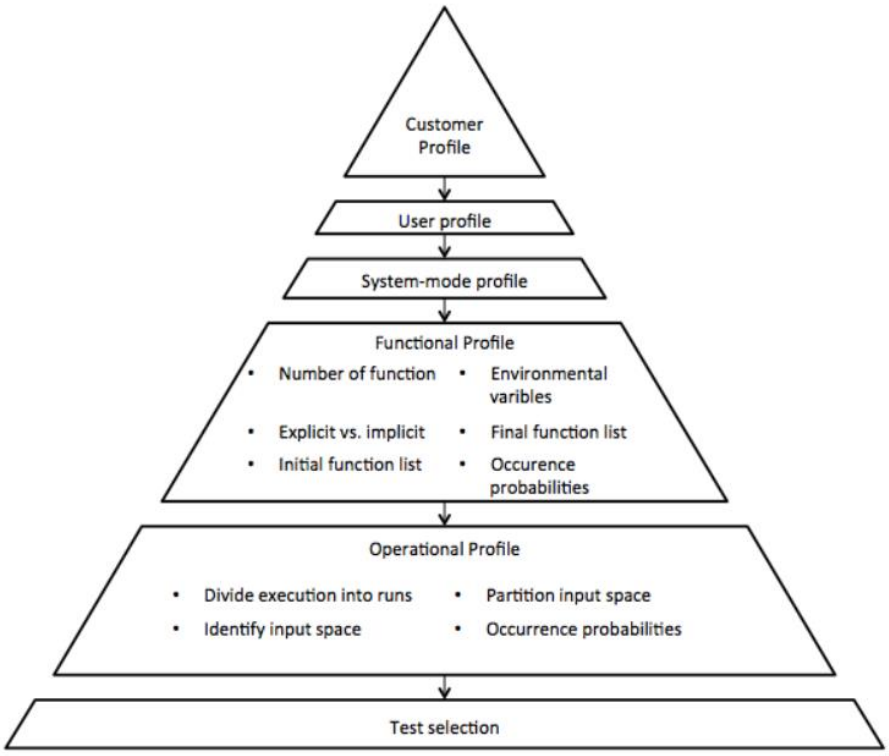


Figura 1. Processo di definizione di un profilo operativo

Il test operazionale è storicamente considerato il metodo più promettente per migliorare e/o stimare l'affidabilità di un sistema, poiché è l'unico metodo di test messo a punto per trovare fallimenti con probabilità corrispondente alla loro frequenza operazionale. Tuttavia, l'utilizzo di questa tecnica fa insorgere alcune difficoltà, la prima delle quali risiede nel presupposto fondamentale di conoscere il profilo operazionale. Derivare un profilo operazionale corretto può essere complicato, poiché questa attività dipende fortemente dalla qualità delle informazioni che si hanno a disposizione; inoltre, è altrettanto difficile controllare se un profilo operazionale è rispondente al profilo reale e, nel caso, stabilire di quanto si discosti da esso.

La seconda difficoltà legata al test operazionale è relativa all'efficacia della tecnica nel lungo periodo. In particolare, mentre i difetti con frequenza di occorrenza maggiore vengono rilevati nella fase iniziale del test, una volta rimossi, i rimanenti difetti con bassa probabilità di occorrenza sono difficilmente rilevabili con il test operazionale. Infatti, poiché questo "emula" l'utilizzo reale del sistema, è ragionevole pensare che un difetto con bassa probabilità di occorrenza abbia una bassa probabilità di essere rilevato durante il test operazionale: ciò costituisce un limite per la *detection* di difetti "rari", che possono essere oggetto di interesse nei sistemi critici.

Per superare tali criticità, in SVEVIA l'applicazione del test operazionale prevede degli aggiustamenti per la generazione dei casi di test: l'idea è quella di combinare, quando il numero di fallimenti rilevati scende sotto una prefissata soglia (ulteriori dettagli in [34]), input a frequenza alta con input a frequenza bassa e simulare nella distribuzione del profilo operazionale un errore casuale gaussiano a media nulla, in modo da fornire un *range* di risultati "robusti" rispetto ad un dato errore nella stima del profilo.

Il *fault-injection testing* è una tecnica che prevede l'inserimento deliberato di guasti artificiali (in modo simile al *mutation testing*, ma con finalità diverse) – che possono essere iniettati a livello di codice sorgente o a livello binario – al fine di valutare la reazione del sistema, in termini di capacità di tollerare eventuali guasti. I guasti sono tipicamente distinti per tipologia (identificata da "*fault operators*") e la loro iniezione è eseguita da tool automatici, che identificano anche le locazioni adatte in cui iniettare un certo tipo di guasti (ulteriori dettagli sulle tecniche esistenti in [35]). Il *fault-injection testing* è di fondamentale importanza per i sistemi critici.

A valle di una sessione di *fault injection* su di un sistema, il *tester* ottiene una misura (nonché il tipo) dei guasti tollerati rispetto a quelli iniettati; questa misura è espressa in termini di "copertura".

Questa tecnica è stata spesso applicata per emulare guasti hardware via software (ad esempio un *bit flip* che emula un difetto transiente di una memoria); tuttavia, nell'ambito di SVEVIA, l'interesse è sull'uso della per emulare guasti software, ad esempio possibili errori di programmazione commessi dai programmatori, come in [35]. Nell'ambito del progetto si utilizzerà la "*software*" *fault injection*, sfruttando le competenze del CINI (laboratorio di Napoli) che, oltre ad essere uno dei partner di progetto, è tra i promotori di questa tecnologia.

Infine, riguardo le tecniche di analisi, per i sistemi SELEX ES è stata ritenuta di particolare importanza l'applicazione di tecniche di *analisi statica automatica* (ASA), che rappresenta un mezzo per implementare in modo efficiente politiche di *early detection*.

L'ASA si basa su l'esplorazione di tutti i possibili percorsi di esecuzione in un programma in fase di compilazione, per verificare se sono rispettate alcune proprietà di interesse (ad esempio, le regole di codifica, convenzioni di programmazione). La sua applicazione è molto utile per identificare errori di programmazione (come buffer *overflow*, riferimento a puntatori nulli, l'uso di variabili non inizializzate) che possono sfuggire sia ai compilatori sia al test. L'ASA è comunemente utilizzata da grandi aziende che producono enormi quantità di software, come Microsoft e Nasa, ottenendo ottimi risultati [36][37].

L'analisi statica può essere utilizzata per verificare se il codice soddisfa requisiti di sicurezza, affidabilità, prestazioni e manutenibilità. È in grado di fornire una base per produrre codice solido, esponendo errori strutturali o impedendo l'occorrenza di intere classi di difetti. Vi sono diversi strumenti di ASA, che identificano specifici tipi di anomalie (ad esempio, la violazione di proprietà) tramite *pattern-based analysis*, *data* e *control-flow analysis, path flow analysis* (maggiori dettagli su queste tecniche sono nel *Deliverable* di progetto "D1.1: Studio ed analisi delle tecniche e degli strumenti di V&V"). Un uso comune delle tecniche di ASA è controllare il codice sorgente rispetto a *pattern* noti che causano difetti o, più in

generale, che ne penalizzano la qualità. Tali strumenti consentono di verificare la conformità a regole standard di codifica (ad esempio JSF, MISRA, ecc.). Esempi di regole sono: *Avoid conditions that always evaluate to the same value, Avoid use before initialization, Avoid null pointer dereferencing, Ensure resources are deallocated, Avoid overflow due to reading a not zero terminated string,* etc..

Il corretto utilizzo e la possibilità di sfruttare al massimo le potenzialità sia delle tecniche di test sia dei risultati della analisi statica automatica è strettamente legato al costo per condurle ed alla capacità di pianificare l'utilizzo di risorse (in termini di ore/uomo) per ottimizzare il risultato. Di seguito, dopo una sezione introduttiva sui metodi per la pianificazione, sono descritti tre metodi per la allocazione ottimale di risorse di test, per l'identificazione di moduli critici, per la conseguente prioritizzazione di attività di V&V e per la allocazione ottima delle risorse dedicate alla risoluzione di problemi rilevati da strumenti di ASA.

3 PIANIFICAZIONE DELLE ATTIVITÀ DI V&V

Prendere decisioni strategiche richiede che un'organizzazione sia capace di "prevedere" ed anticipare i cambiamenti futuri, così da investire più proficuamente sugli obiettivi maggiormente critici che intende raggiungere.

I modelli di pianificazione strategica illustrati in questo documento nascono dallo studio dello stato dell'arte dei più significativi criteri di V&V e dalle tecniche di ottimizzazione della ricerca operativa. Essi consentono di ottenere delle stime quantitative delle relazioni che intercorrono tra costo di test e qualità che tale attività produce, facendo leva sulle caratteristiche del *software*, su modelli empirici, nonché sull'esperienza e sul *know-how* maturati all'interno dell'organizzazione che li adotta.

Ciò che verrà discusso di seguito rappresenta un ulteriore tassello che contribuirà a dimostrare che l'adozione di metodologie e tecniche innovative di V&V agevolino una organizzazione ad assumere nei confronti del mercato un atteggiamento sia anticipativo sia proattivo. La caratteristica di anticipare i cambiamenti, adattandosi e, laddove possibile, influenzando il cambiamento a proprio favore, costituisce una delle più importanti risorse di un'organizzazione.

Il principale scopo della V&V di un sistema software è quello di esporre il maggior numero possibile di "fault" che esso presenta; tale attività però non garantisce che al termine il sistema sia più affidabile. Pertanto, il problema dell'allocazione delle risorse di test è quello di identificare le parti del sistema alle quali destinare un maggior numero di risorse durante le attività di *testing*, con l'obiettivo di ottimizzare il rapporto tra costi sostenuti e qualità raggiunta.

Esso può essere affrontato focalizzandosi sulla "difettosità" dei moduli (componente o sotto-sistema) *software*, oppure sul miglioramento del livello di reliability del sistema.

Il primo approccio di cui sono un esempio i modelli basati sui SRGM (*Software Reliability Growth Models*), consiste nell'allocare maggiori risorse a quei componenti i cui fallimenti possono avere un maggiore impatto sulla *reliability* complessiva del sistema, poiché ci si aspetta che l'affidabilità aumenti proporzionalmente alla quantità di risorse di *testing* allocate sui componenti più impattanti. Questa tecnica si focalizza sul rilevamento e rimozione dei *fault* la cui frequenza attesa in fase operazionale è più alta.

Il secondo approccio, consente di concentrare gli sforzi ("*effort*") di test su moduli *software* che ci si aspetta contengano più "*fault*"; ad esempio, nei modelli "*fault-proneness*", che utilizzano la "*fault-prediction*", la difettosità del modulo viene valutata in base a numerose metriche di processo e/o di prodotto.

I metodi basati sul miglioramento della *reliability* adoperano modelli predittivi che offrono una misura del numero di risorse di *testing* da allocare ai componenti che hanno un maggiore impatto sull'affidabilità, ciò al fine di garantire che il sistema raggiunga un fissato livello di *reliability*. Dunque, prima di adoperare il criterio, abbiamo bisogno di reperire informazioni sull'uso dei singoli componenti (ad esempio il numero medio di visite al componente, il tempo medio di esecuzione per ciascun modulo, la frequenza di

fallimento) e, naturalmente, il livello di affidabilità da raggiungere. Pertanto, è consigliabile utilizzare questi metodi in una fase avanzata del processo di V&V, o quando si dispone di numerosi dati storici relativi al processo di test o di uso operazionale. Si noti che, quando si dispone di informazioni sufficienti, i modelli basati sul miglioramento della reliability possono suggerire una allocazione delle risorse più accurata rispetto ai modelli di *fault-proneness*.

I metodi basati sulla *fault-proneness* adoperano dei modelli per stimare la difettosità di un componente *software* (spesso con output binario del tipo "modulo con difetti" o "modulo senza difetti"). Pertanto, per utilizzare questo metodo è importante disporre di informazioni tratte dalle metriche di prodotto, dal numero di *fault* per ciascun modulo analizzato, nonché dal numero di *fault* ivi rilevato in passato, eventualmente classificati per tipo.

Mentre nei metodi basati sul miglioramento della reliability i dati sono legati al processo di test con cui i difetti vengono esposti, in quelli basati sulla *fault-proneness* il tipo di informazione rappresenta la qualità dal punto di vista dell'implementazione. In entrambi i casi, l'obiettivo è quello di migliorare il *trade-off* tra costo di applicazione delle tecniche di V&V e qualità predetta (utilizzando metriche di implementazione oppure metriche di test). Saranno nel seguito brevemente riprese le due categorie di metodi così come affrontate in letteratura (maggiori dettagli sono nel *Deliverable* di progetto: "D1.1: Studio ed analisi delle tecniche e degli strumenti di V&V").

3.1 *Software Reliability Growth Models*

La "*Software Reliability Engineering*" (SRE) è una branca dell'ingegneria incentrata sull'analisi dell'affidabilità di un sistema *software* [15]. I principi della disciplina SRE possono essere applicati a tutte le fasi del ciclo di vita di un *software*: nelle fasi iniziali (al fine di valutarne preliminarmente l'affidabilità); nel corso del processo (allo scopo misurare e monitorare le procedure di individuazione dei *bug*, utilizzando modelli statistici crescita dall'affidabilità); nelle fasi finali (per effettuare prove di verifica[3] dell'affidabilità).

I modelli "*Software Reliability Growth Model*[4]" (SRGM), descrivono il modo in cui cresce l'affidabilità di un sistema software durante il processo di *testing*. Gli SRGM sono adottati con finalità distinte, quali:

- ottenere una predizione[5] sul livello di *reliability* che il sistema *software* raggiungerà dopo un certo tempo o *effort* di test;

[3] Di solito i piani di prova per certificare l'affidabilità del software vengono realizzati su richiesta del cliente.
[4] In letteratura sono stati presentati molti modelli SRGM, come: Exponential SRM [16], Log Normal SRM [17], Log Logistic SRM [18], Gamma SRM[19], Truncated Logistic SRM [20], Truncated Extreme-Value Min e Max SRMG[21].Una descrizione più esaustiva è presente nel Deliverable D1.1: "Studio ed analisi delle tecniche e degli strumenti di V&V".
[5] Le previsioni realizzate dagli SRGM si basano sulla disponibilità delle informazioni sul sistema in esame; di conseguenza, la bontà della stima dipende dalla quantità e dalla qualità dei dati utilizzati nel modello. Nelle fasi inziali

- dopo aver fissato un obiettivo di reliability R, stimare l'*effort* da allocare al *testing* per raggiungere il livello di affidabilità R e migliorare il processo di *decision making* relativo alla pianificazione ottimale del rilascio.

Storicamente, questi modelli sono stati adoperati per ottenere misure di affidabilità di un sistema *software* nel suo complesso; tuttavia, recentemente, essi sono stati sfruttati anche per ottimizzare l'allocazione delle risorse di *testing* ai singoli componenti. L'adozione degli SRGM forniscono stime tanto più accurate quanto più le informazioni di cui si dispone sono significative.

Nel caso di utilizzo a livello di componente, se supponiamo che una campagna di test abbia evidenziato un numero rilevante di anomalie, il modello assegnato al componente *software* in esame potrà essere calibrato su tali informazioni. Grazie all'SRGM si potrà ottenere una previsione del numero di anomalie residue che potrebbero essere ancora rilevate, nonché una stima del *test effort* (costi/tempi) necessario a portare a termine il processo.

Le stime ottenute consentono ad un'organizzazione di misurare il tempo necessario al rilascio di un sistema o componente e, di conseguenza, di scegliere una soluzione che rappresenti un valido compromesso tra numero di difetti residui e le risorse da allocare alla relativa attività di *testing*.

Queste considerazioni possono tornare estremamente utili quando vi sono vincoli contrattuali al rilascio di un sistema o componente *software*. Ad esempio, prima della consegna del prodotto, l'ente produttore potrebbe dover garantire che il prodotto *software* presenti un numero di *bug* residui inferiore ad una soglia massima che il Cliente è disposto a tollerare. Pertanto, l'organizzazione sarebbe obbligata a proseguire con le attività di *testing* sul sistema o componente, sino al raggiungimento del livello di soglia fissato. In questi casi, atteso che siano verificate determinate assunzioni sul processo di test, i modelli di crescita dell'affidabilità potrebbero aiutare il team di V&V a determinare il tempo di *testing* ancora necessario per il rilascio della *release*. Per lo stesso motivo, le stime ottenute mediante gli SRGM possono agevolare le scelte di una organizzazione anche quando il vincolo contrattuale è la data del rilascio del prodotto *software*.

I modelli di crescita dell'affidabilità del *software* possono essere applicati anche in fase di analisi e valutazione dei risultati di test. In tal caso, essi consentono di ottenere una misura dello scostamento tra i risultati reali e quelli attesi (calcolati preliminarmente attraverso i modelli).

---

del ciclo di vita di un sistema, soprattutto se non è mai stato realizzato prima, le informazioni disponibili potrebbero essere poche, quindi sarà necessario operare delle semplificazioni nel modello. In una fase già più avanzata del CVS, quando saranno disponibili più dati significativi, l'applicazione di questi modelli possono offrire una stima del livello di *reliability* raggiunto dal sistema subito dopo una sessione di test. Si deduce, quindi, che la stima effettuata su quest'ultima informazione potrà essere più precisa

Un modello SRGM viene costruito a partire da informazioni, dette "*failure data*[6]", raccolte durante la fase di *testing*. Successivamente, viene operato il *fitting* dei tempi di inter-fallimento con modelli pensati per rappresentare il processo di test, che consente di osservare come varia il numero dei fallimenti nel tempo di *testing*; pertanto, l'intensità dei fallimenti determina l'andamento della curva di *fitting*. Ogni modello SRGM presentato in letteratura è caratterizzato da una particolare curva dell'intensità dei fallimenti [22].

In particolare, gran parte degli SRGM modellano la crescita della *reliability* di un sistema *software* come un Non-Homogeneous Poisson Process[7] (NHPP) [23][32].

Un processo stocastico NHP è caratterizzato dalla propria "*mean- value function*" (mvf), che può essere calcolata come:

$$m(t) = E[N(t)]$$

in cui N(t)[8] è pari al numero di fallimenti osservati nel *software* nell'intervallo temporale di *testing* (0, t]; oppure come "*failure intensity*", cioè:

$$\int_0^t m(x)dx$$

Tipicamente, le *mvf* sono descritte mediante distribuzioni di probabilità, riportate in Tabella 1.

[6] Si noti che, una volta raccolti, i dati vengono utilizzati per definire una distribuzione empirica, di cui viene verificata la rappresentatività eseguendo il *fitting* con una distribuzione teorica; questa verifica è nota come di "*goodness of fit tests*". Tra le tecniche più utilizzate allo scopo vi sono i test di Kolmogorov-Smirnov e quelli di *Akaike's Information Criterion* (AIC). I primi selezionano il modello confrontando la funzione di ripartizione empirica (dedotta dal dato raccolto) con quella teorica (dedotta dal modello). Il criterio AIC sfrutta la distanza KL e permette di selezionare il modello la cui somma degli errori al quadrato è più piccola.

[7] È poco realistico pensare che il tasso di guasti che il processo di test riesce a rilevare sia costante nel tempo. I processi di Poisson non omogenei (detti anche a tasso non costante) consentono di modellare un maggior numero di "eventi" in modo più realistico.

[8] Per maggiori dettagli fare riferimento al § A.1 dell'Appendice.

Tabella 1 Esempi di SRGM

COVERAGE FUNCTION	PARAMETERS	m(t)	Failure Occurrence Rate per Fault
Exponential	a, g	$a(1 - e^{-gt})$	Constant
Weibull	a, g, ϒ	$a(1 - e^{-gt^{\Upsilon}})$	Increasing/Decreasing
S shaped	a, g	$a[1 - (1 + gt)e^{-gt}]$	Increasing
Log - Logistic	a, ω, κ	$a\frac{(\omega t)^{\kappa}}{1+(\omega t)^{\kappa}}$	Inverse Bath Tub

L'intensità di fallimento *λ(t)*, che rappresenta il numero di fallimenti nell'unità di tempo, è pari alla derivata della *mvf*.

Alcuni studi sperimentali hanno dimostrato che, sotto le ipotesi che i *fault* siano distribuiti uniformemente all'interno del sistema o componente *software* e un potenziale *fault* venga riparato contestualmente alla sua rilevazione, è possibile ricavare una relazione tra la funzione valore medio e la "*coverage function*", cioè:

$$m(t) = a \cdot c(t)$$

dove:

- a rappresenta il numero di *fault* residui;
- $c(t)$ è il fattore di copertura, che indica l'esito dell'intervento di riparazione di un guasto al tempo t.

Questa equazione mostra chiaramente che l'intensità di fallimento dipende non solo dal numero di difetti residui, ma anche dal *rate* con cui i *fault* potenziali sono riparati [24].

La formulazione dei modelli di crescita dell'affidabilità può includere il concetto *di Testing Effort Function* (TEF), cioè di funzioni che descrivono l'andamento non lineare delle risorse di test spese nel tempo [25][26][27].

Tra gli SRGM proposti in letteratura, il più noto è il Goel-Okumoto *model*:

$$\lambda(t) = b\big(a - m(t)\big) = abe^{-bt}$$

dove la *failure intensity*[9] λ(t) è espressa attraverso una funzione esponenziale, in cui:

- a rappresenta il numero atteso di *fault* che il *testing* riuscirebbe a rilevare se non fosse fissato un limite temporale;
- *b* rappresenta lo "*hazard rate*"[10], una misura molto utile per stimare la *reliability* finale del *software*, che nella distribuzione esponenziale è costante (esso rappresenta il *rate* di fallimento per *fault* residuo).

Un modello che generalizza il modello esponenziale è ottenuto tramite la distribuzione di Weibull, che viene utilizzata molto frequentemente per modellare i tempi di guasto dei sistemi con *hazard rate* non costante nel tempo, ma tali che:

$$h(t) = \alpha\beta(\alpha t)^{\beta-1}$$

dove:

- $t > 0$ rappresenta il tempo di attesa del guasto;
- $\alpha > 0$ è un fattore di scala;
- $\beta > 0$, che è il parametro più significativo, definisce la curva di distribuzione di Weibull (rappresentata in Figura 2).
-

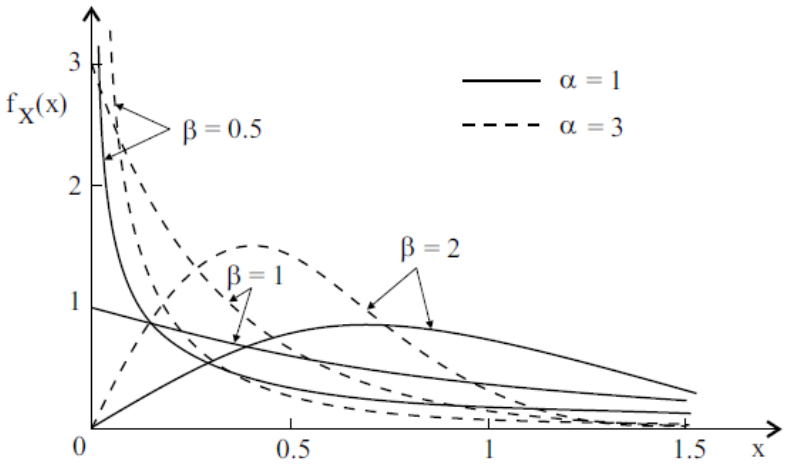


Figura 2 Distribuzioni di Weibull W(α, β), al variare di α e β

[9] Per maggiori dettagli fare riferimento al § A.1 dell'Appendice.
[10] L'*hazard rate*, h(t), indica la probabilità condizionata che un sistema (o un suo componente) sopravvissuto sino al tempo t, si deteriori nell'intervallo di tempo (t, t + Δt]. Tale valore dipende esclusivamente dalle caratteristiche intrinseche del sistema (architettura e tecnologia)

Un altro modello significativo, per la potente capacità predittiva [28], è quello presentato da Gokhale e Trivedi: l'SRGM Log-logistic, in cui l'*hazard rate* viene espresso attraverso una distribuzione log – logistic. Esso è in grado di catturare l'aumento o la diminuzione del tasso di occorrenza per *fault*, che ben rappresenta la casistica in cui, nella fase iniziale del test, il tester non ha ancora acquisito familiarità col sistema.

Gli SRGM, essendo modelli, sono utilizzabili purché certe assunzioni siano almeno in parte verificate. La Tabella 2 [30] riassume le ipotesi sotto le quali sono stati considerati i modelli SRGM.

Chiaramente, nei contesti industriali, specie quando si lavora con sistemi complessi, non è sempre possibile operare sotto le ipotesi sopra citate. Tuttavia, questo potrebbe non costituire un problema, poiché è stato dimostrato che, anche quando alcune di queste assunzioni viene violata, i modelli SRGM possono continuare ad offrire predizioni molto soddisfacenti [30].

I modelli SRGM possono essere utilizzati, come menzionato, per il problema dell'allocazione delle risorse di *testing*. Dato un sistema *software* e fissato un livello minimo di *reliability* che esso dovrà garantire, il "*reliability allocation problem*" può essere risolto per mezzo di un modello. La soluzione ottima del problema è quella che restituisce la migliore allocazione delle risorse sui componenti del sistema, nel rispetto del vincolo fissato sulla *reliability* complessiva del sistema o sul *budget*.

Tabella 2 NHPP SRGM - Ipotesi Principali [30]

Faults are repaired immediately when they are discovered
Fault repair is perfect
Faults are independent of each other
No new product features were introduced during test
The test script remained constant over the stability test period
All reliability improvements are caused by solving the open stability faults
An NHPP can describe the fault detection and removal process
Each unit of test execution time is equally likely to find a failure
Testing follows an operational profile

Dunque, sotto le ipotesi che:

- un sistema *software* sia costituito da N componenti, a ciascuno dei quali è assegnato un preciso *slot* temporale di *testing*, detto tempo di *testing*, indicato con T_i (con i = 1, …, N);

a) la *reliability* di ciascun componente del sistema aumenti mano a mano che prosegue la relativa attività di *testing*, cioè (indicata con λ la *failure intensity*[11]*):*

$$T = f(\lambda)$$

i modelli di allocazione ottima delle risorse di *testing* possono essere risolti mediante la formulazione di un opportuno problema di ottimizzazione. Ad esempio, se si suppone di voler minimizzare il tempo di *testing* totale per raggiungere un obiettivo di *reliability* prefissato, la soluzione dell'allocazione ottima delle risorse di *testing* coincide con la soluzione del problema di ottimizzazione rappresentato come:

$$Minimize\ T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f_i(\lambda_i)$$

$$\text{s. a} \qquad E[R] = \left[\prod_{i=1}^{n-1} R_i\right] R_n \times K \geq R_{min}$$

dove:

- $f_i(\lambda_i)$ indica la stima, ottenuta invertendo la funzione SRGM, del tempi di *testing* da spendere sul componente *i* per ottenere una *failure intensitiy* pari a (λ_i). La somma, pertanto, rappresenta il tempo di *testing* totale da minimizzare.
- $E[R]$ è la *reliability* del sistema, vincolata ad essere maggiore del livello minimo prefissato R_{min};
- K rappresenta le reliability di ciascun componente, legate alle variabili decisionali R_i del modello.

Similmente, si può pensare di risolvere il problema duale: cioè, dato un certo *budget* di *testing* (vincolo del problema), adottare un criterio SRGM-*based* per stabilire l'allocazione delle risorse di testing che massimizzi la *reliability* (funzione obiettivo), o una sua funzione derivata (come, ad esempio, il numero atteso di guasti residui).

In ogni caso, la bontà della soluzione dipende dalla qualità e dalla quantità delle informazioni utilizzate nel modello. Ad esempio, quando non sono ancora disponibili informazioni sui componenti, il modello di ottimizzazione deve essere costruito su dati ottenuti analizzando l'architettura del sistema *software*. Di conseguenza, non è possibile differenziare i modelli SRGM per ogni componente, ma bisognerà operare delle semplificazioni, utilizzando per essi uno stesso SRGM (ad esempio, tale che $f_i(\lambda_i) = f(\lambda_j)$).

La soluzione (minima) del problema fornisce il livello di *reliability* che ogni componente dovrà mostrare affinché il sistema raggiunga il livello di affidabilità imposto. Supponendo, invece, che siano disponibili informazioni sulla complessità dei componenti *software* - reperite mediante analisi statica del codice sorgente - ed informazioni che riguardano le loro interazioni, si può ottenere una la soluzione più accurata (estesa). Come per il caso precedente, può essere usato lo stesso SRGM per tutti i componenti, i cui parametri varieranno a seconda del peso assegnato a ciascun componente $f_i(\lambda_i) = f\left(\frac{\lambda_i}{1+rWeight_i}\right)$.

[11] Per maggiori dettagli fare riferimento al § A.1 dell'Appendice.

La soluzione estesa del problema offre una stima dell'*effort* da profondere affinché ciascun componente raggiunga il livello di affidabilità fissato, mentre la soluzione ottima fornisce informazioni sul tempo di *testing* effettivo da allocare ad ogni componente del sistema *software*.

La selezione e la successiva raccolta delle informazioni che caratterizzano un sistema *software*, o i suoi componenti, sono attività cruciali per la scelta del modello da adottare. Il modo più sistematico per reperire informazioni utili in tal senso è quello di definire i profili operativi[12], già introdotti nella sezione 2, che offre una stima di come il sistema sarà utilizzato nella pratica.

Quando un *software* è nuovo, però, diventa difficile anticipare come sarà utilizzato dall'utente tipico e quindi sviluppare un profilo operativo accurato. Inoltre, col passar del tempo, l'uso del sistema può cambiare e di conseguenza anche i profili operativi che ne rappresentano le funzionalità. Quando non sono disponibili informazioni storiche relative all'utilizzo dei componenti ed al comportamento di un componente in passate installazioni (da cui potrebbe essere ricavato un SRGM), l'utilizzo di questi modelli potrebbe dare informazioni poco accurate. In tal caso, possono essere di supporto i modelli summenzionati di "*fault-proneness*" che richiedono altri tipi di informazioni.

Per contrastare questo problema, nell'ambito di SVEVIA è stata proposto un metodo di allocazione "dinamica" delle risorse di *testing* ai componenti *software*, basato su informazioni raccolte esclusivamente al momento del test dei componenti. Tale tecnica, illustrata in [33], si basa su:

- l'assegnazione a ciascun componente di un appropriato modello SRGM. Quest'ultimo viene valutato e raffinato dinamicamente, mano a mano che procede l'attività di test (senza quindi necessitare di dati storici);
- la minimizzazione della densità dei difetti attesa e/o la minimizzazione del numero di difetti residui attesi, oppure la massimizzazione della *reliability*.

Tale soluzione è proposta nel dettaglio nella sezione 4.

3.2 FAULT – PRONENESS

L'analisi *fault-proneness* consente di prevedere quali siano i componenti *software* più difettosi di un sistema, così da concentrare su di essi l'allocazione della maggior parte delle risorse di *testing*. Essa si avvale della tecnica di *fault-prediction*, che a sua volta sfrutta tecniche statistiche o tecniche

[12] Negli anni sono state sviluppate molte unità di misura per specificare i requisiti di affidabilità di un sistema, Tuttavia, per verificare che il sistema software soddisfi tali requisiti, occorre misurare l'affidabilità del sistema dal punto di vista dell'utente tipico. Le informazioni che rispecchiano l'uso pratico che verrà fatto del sistema verranno utilizzate come input nelle campagne di test. Tipiche informazioni raccolte sono numero: il numero medio di visite, tempo medio di esecuzione, frequenza di fallimento del sistema/componente.

dell'intelligenza artificiale per stimare, attraverso insiemi di metriche[13] usate come predittori, la tendenza a presentare *fault* da parte di un componente.

Le tecniche statistiche applicate per la *fault-prediction* danno luogo a modelli di regressione[14] e di *machine learning*[15], entrambi utilizzati per ottenere informazioni sulla correlazione tra metriche e *fault*.

In Figura 3 sono rappresentati le fasi che conducono all'allocazione delle risorse di V&V basata sulla *fault-prediction*.

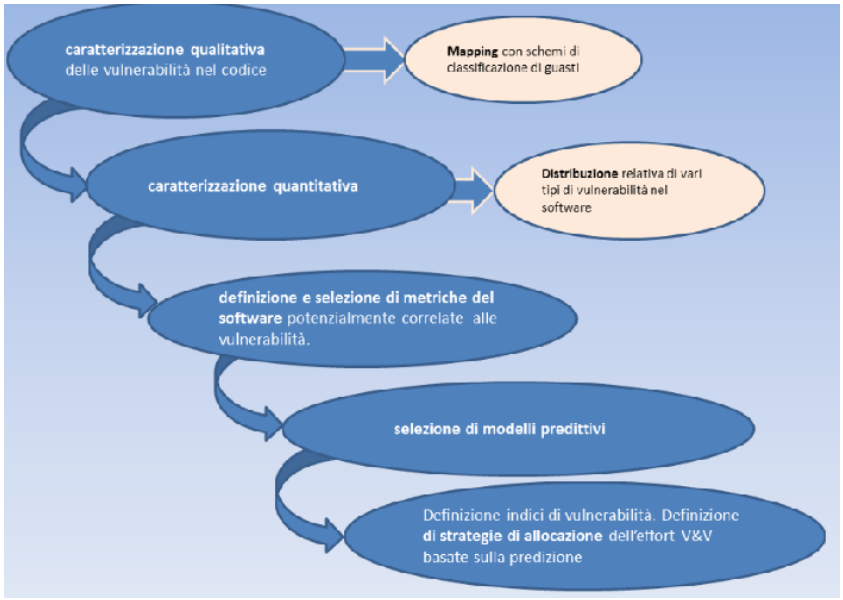


Figura 3 Approccio Fault - Proneness per l'allocazione delle risorse

[13] Tipicamente si tratta di metriche di prodotto, classificate in method-level, class-level, file-level, component- level [5]. In particolare, le più comuni:

- metriche *method-level* sono: le metriche di McCabe [6] e le quelle di Halstead [7];
- metriche *class-level* (O-O based) sono: le metriche di Chidamber-Kemer (CK) [8], le metriche MOOD [9], le QMOOD [10] e le suite di metriche di L&K (Lorenz and Kidd) [11].
- metriche *file-level* (sempre più usate per la *fault-prediction*), sono: il numero di linee di codice per file, il numero di linee commentate di codice per file, il numero di cambiamenti per file.

Tra le metriche class-level le più utilizzate per la fault-prediction sono: Coupling_Between_Object (CBO), Weighted_Method_Per_Class (WMC) e Response_For_Class (RFC).

[14] Il termine "regressione" (Galton 1888) assume oggi il significato di "relazione funzionale tra variabili ottenute con metodi statistici". Appartengono a questa categoria le tecniche *Linear Regression*, *Logistic Regression*.

[15] Il *Machine Learning* (Apprendimento Automatico) è un campo di ricerca multidisciplinare, che trae la sua motivazione storica dalla *Artificial Intelligence*, ma che ad oggi risulta sempre più legato a metodologie di indagine statistica dei dati. A questa categoria appartengono tecniche quali *Decision Tree, Bayesian Net, SVM*.

L'allocazione delle risorse di V&V basata sui *fault-proneness* inizia con la caratterizzazione delle vulnerabilità del codice, nella definizione e selezione delle metriche del *software* potenzialmente correlate alla presenza di vulnerabilità nel codice e nella rappresentazione di tali informazioni attraverso un modello statistico[16] (cfr. gli step 1, 2, 3, 4 di Figura 3).

In particolare, dato un modello di rappresentazione, la predizione consiste nell'identificare il valore che assumerà una grandezza misurabile in un istante di osservazione futuro. Questo valore è quello che meglio sintetizza il manifestarsi di un fenomeno all'istante temporale fissato, che nel nostro caso è la caratteristica "*faulty*" dei componenti software. La costruzione di un modello statistico serve a prevedere i valori assunti da una variabile dipendente (o anche "predetta") sulla base dei valori di una o più variabili indipendenti (dette anche "esplicative" o predittori). Pertanto, esso esprime se e quanto un effetto (variabile dipendente) sia correlato ad una o più cause (variabili indipendenti).

Nel caso in esame, i modelli statistici possono fornire una stima del numero di *fault* per ogni componente utilizzando come predittore un insieme di metriche, di cui sono noti i valori. La bontà della predizione dipende dalle variabili esplicative utilizzate; tipicamente viene scelto come ottimo il predittore che assicura il minor costo medio.

Le variabili dipendenti utilizzate nei modelli predittivi possono assumere valori reali oppure discreti. Nel primo caso, il dato previsto consiste nel numero di *fault* di ogni componente *software*; nel secondo caso, invece, ciò che viene previsto non è un valore numerico, bensì l'appartenenza o la non appartenenza di un *fault* ad una classe. Infatti, affinché la previsione possa essere svolta, è necessario rendere discreto l'insieme dei valori che può assumere la variabile dipendente, così da calibrare il modello su questi dati. Di conseguenza, la predizione consisterà nell'attribuzione dei campioni ai *range* dei valori ottenuti discretizzando il dominio di partenza. Sebbene predire il numero di *fault* possa sembrare un approccio migliore, la realtà dimostra che è complicato ottenere delle stime affidabili del *numero* di *fault*, data l'alta variabilità dei dati. Spesso, perciò, si preferisce adottare approcci che predicono la classe di appartenenza, tra un insieme di classi, di un determinato modulo (ad esempio, una classificazione binaria: "modulo con difetti", "modulo senza difetti", o con più classi a severità crescente).

Un ulteriore approccio, molto diffuso, consiste nell'operare una predizione su una variabile reale (numero di *fault*), utilizzando il numero assoluto solo per ordinare i moduli dal più difettoso a quello meno difettoso; dunque, in base alla lista dei moduli ordinati per "difettosità" (*ranking*) si stabilisce come allocare le risorse. Infine, la predizione può essere operata considerando i dati storici come classifiche (*ranking*) di moduli difettosi ed utilizzando tali classifiche (insieme alle metriche "predittori") per allenare i modelli che predicono direttamente il *ranking* (lo stesso output di prima, ma senza ricorrere all'artificio di usare il numero di *fault* per classificare il risultato).

[16] Il modello statistico è una riproduzione semplificata della realtà, ma che ne riflette gli aspetti essenziali eliminando quelli ritenuti secondari.

In base a quanto detto, potremmo riferirci ai modelli *fault-proneness* che forniscono una predizione della classe dei "*fault prone*" (modelli di classificazione), oppure a quelli (modelli di regressione) che offrono una predizione di un parametro quantitativo (cioè il numero di *bug* o direttamente il *ranking*).

Mentre esistono svariati modelli di predizione che si focalizzano sulla difettosità assoluta [5], non sono presenti modelli che, oltre a predire la difettosità, distinguono anche il tipo di difettosità presente in un sistema o componente. Infatti, alcuni *fault* che si manifestano durante la fase operativa di un sistema hanno un comportamento aleatorio rispetto al *pattern* di attivazione. Di conseguenza, né tali *fault*, né le cause che li hanno scatenati, sono facilmente rilevabili e riproducibili. In questi casi, l'uso di modelli di predizione che operano una tale distinzione può portare dei benefici.

Ad esempio, una possibile distinzione del tipo di *bug* che può essere adottata in un modello è quella proposta da Trivedi [39], che definisce come :

- "*Bohrbug*" difetti "deterministici", isolabili e facilmente riproducibili in fase di test;
- "*Mandelbug*" difetti apparentemente "non-deterministici", spesso latenti, e non facilmente riproducibili sistematicamente in fase di test, in quanto la loro attivazione dipende da fattori dell'ambiente di esecuzione.

Gli studi condotti sulla *fault-prediction* hanno evidenziato che, per essere realmente efficaci, i modelli predittivi devono essere costruiti sulle caratteristiche specifiche del sistema che si vuole analizzare[17], dunque, non esistono modelli predittivi generici, applicabili con successo a qualsiasi tipo di sistema [13].
Questa difficoltà può essere superata definendo modelli di "*fault prediction type-aware*", che sono in grado di predire sia il tipo che il numero di *fault* del componente in esame. Dunque, essi sfruttano sia le capacità dei modelli di classificazione che di quelli predittivi.

Data la doppia predizione, i modelli *fault prediction type-aware* possono essere più efficaci dei *fault prediction*. Infatti, la sola predizione della propensione ai guasti non fornisce alcuna indicazione sulla natura dei *fault*. La mancanza di queste informazioni può avere un impatto negativo sia sulla efficacia dell'allocazione delle risorse, sia sull'efficienza della tecnica di *V&V* adottata, le cui prestazioni dipendono dal tipo di guasti presenti nel sistema. Al contrario, l'adozione di tali modelli, può consentire di allocare le risorse di *testing* ai componenti *software* in modo più efficace, perché la distribuzione viene fatta sia sulla base del numero di *fault* sia sul tipo di *fault* predetti. Nella sezione 5 sarà presentato il metodo adottato nell'ambito del progetto SVEVIA per la definizione di un modello di predizione dei *fault* "*type-aware*".

In generale, i modelli di *fault-proneness* sono certamente degli strumenti di supporto validi allorquando si dispone di una quantità sufficiente di informazioni. Nel caso di organizzazioni che dispongono di ingenti

[17] La selezione delle informazioni specifiche del componente in esame viene anche detta *feature selection* [14].

quantità di informazioni - provenienti da sorgenti come sistemi di *bug-tracking*[18] e CVS, tali modelli costituiscono un modo naturale di sfruttare la conoscenza storica aziendale per effettuare previsioni sui comportamenti futuri dei propri sistemi. Inoltre, un vantaggio dei modelli di *fault proneness,* rispetto a quelli basati sugli SRGM, è che non necessitano di dati di test od operazionali, ma possono essere utilizzati a partire da informazioni sul codice sorgente, prima, cioè, di iniziare qualsiasi attività di esecuzione di test.

Tuttavia, i modelli *fault-proneness presentano* alcuni limiti*, quali:*

1. La necessità di disporre del codice sorgente[19] del componente dal quale estrarre le metriche;
2. L'impossibilità di definire uno schema di allocazione esplicito, infatti il metodo offre una possibile soluzione al problema dell'allocazione, che non è necessariamente quella ottima;
3. La natura statica della tecnica *metric-based*, ragion per cui la predizione non viene arricchita e/o corretta con informazioni ottenute osservando il sistema quando è in esecuzione;
4. Mancanza di un modello generico, applicabile a componenti eterogenei;
5. La necessità di una *knowledge base* per poter realizzare delle correlazioni significative tra le metriche ed i difetti.

Al fine di ridurre i costi delle attività di V&V e migliorare la *reliability* del proprio prodotto, un'organizzazione dovrebbe valutare attentamente le problematiche sopraccitate, così da stabilire se ed in quale momento del processo utilizzare la strategia.

[18] Tipicamente un sistema di *bug tracking* ha la capacità di estrarre ed inserire informazioni relative ai *bug* (id, *priority*, *resolution*, *status*, *comments*, *descriptions*, *changes*, *attachments*, ect.) in un *database*. Queste informazioni possono essere utilizzate per studi e analisi.

[19] Nelle tecniche di *fault/defect* prediction viene solitamente utilizzato un approccio *white-box*.

4 ALLOCAZIONE DELLE RISORSE DI TEST BASATA SU SRGM

Migliorare l'affidabilità di un sistema vuol dire migliorare la qualità dei suoi componenti, eppure spesso è difficile valutare la qualità iniziale dei singoli, così da scegliere la giusta allocazione delle risorse di *testing*.

Quello della allocazione delle risorse è un problema cruciale per i *test manager* coinvolti nel processo decisionale, in particolar modo quando il sistema in esame è complesso e critico, data l'ingente quantità di *testing* da operare e gli stringenti vincoli di tempo, di costi e di qualità a cui deve sottostare.

Nei contesti industriali quali SELEX ES questo problema è particolarmente sentito. Come descritto nei Deliverable precedenti del progetto SVEVIA (D1.2: "Sviluppo ed analisi dei processi, delle metriche e degli standard aziendali per lo sviluppo software" e D2.1: "Definizione di criteri, metriche, e scenari di validazione"), i sistemi prodotti da SELEX ES, tipicamente, sono realizzati con una architettura fortemente *component-based*. Ciascun *component*, denominato CSCI, ha caratteristiche proprie e svolge una attività specifica nel sistema, dunque può avere un impatto più o meno significativo sull'affidabilità del sistema complessivo. Questa proprietà, orientata al forte riuso e alla relativa indipendenza dei processi di sviluppo e di test dei singoli CSCI, può essere sfruttata dagli ingegneri per pianificare la distribuzione delle risorse di *testing*.

In mancanza di criteri specifici, l'allocazione delle risorse può essere operata mediante degli approcci più generici, come quello guidato dai requisiti, o dalla dimensione dei componenti che costituiscono il sistema *software*. Ancora, si può decidere di destinare un maggiore *effort* a quei sotto-sistemi/componenti *software* identificati dagli ingegneri come critici rispetto ai livelli di affidabilità richiesti al sistema.

Sebbene l'allocazione delle risorse debba far leva sulla conoscenza, sull'intuizione e sulla capacità di giudizio degli esperti coinvolti, in un sistema altamente complesso, di centinaia di migliaia di righe di codice o di dimensioni ancora maggiori, può essere difficile identificare i componenti che hanno un impatto maggiore sulla affidabilità complessiva del sistema.

È pratica comune utilizzare degli approcci qualitativi per identificare i componenti più critici di una architettura *software*, basati cioè sul giudizio degli esperti di dominio. Purtroppo, queste valutazioni sono spesso insufficienti e non consentono di pianificare opportunamente l'allocazione delle risorse di *testing*, né di quantificare il livello di *reliability* del sistema.

Il problema può essere meglio affrontato adoperando dei criteri ingegneristici che tengano conto delle considerazioni fatte sino ad ora, quali: il differente livello di qualità dei componenti *software*, l'impatto sulla affidabilità che ciascuno di essi può avere sul sistema, le capacità di valutazione dei singoli componenti e le conoscenze degli esperti del dominio applicativo ed i *tool* utilizzabili a supporto del processo decisionale.

A tale scopo è importante che i *test manager* siano in grado di stabilire quale debba essere il livello di affidabilità di un componente affinché il sistema complessivo raggiunga un livello accettabile di *reliability* ed a quale costo l'organizzazione possa ottenere questo risultato. Tale relazione può essere efficacemente descritta dagli SRGM.

Nel seguito viene descritta una soluzione basata sugli SRGM a livello di componente, in grado di stabilire dinamicamente (cioè durante l'esecuzione delle attività di test) come ri-allocare le risorse di test, al fine di raggiungere l'obiettivo di affidabilità fissato nel rispetto di un dato vincolo di costo. Tale metodo, nato dalla collaborazione nell'ambito del progetto collaborazione tra il mondo accademico e quello industriale, ha lo scopo di migliorare la capacità di *defect detection, a parità di costo di test,* utilizzando i modelli SRGM. Nello specifico, fissato un vincolo di costo/tempo di testing, il metodo di allocazione dinamica illustrato mira a minimizzare il numero dei difetti residui nel codice (*defect-based allocation*), oppure alla ridurre la *defect-density* (- *based allocation*). Nel prosieguo del presente capitolo viene descritto il metodo, l'algoritmo ed i risultati sperimentali ottenuti preliminarmente su alcuni sottosistemi di SELEX ES.

4.1.1 DESCRIZIONE DEL METODO

Dato un *budget* di *testing* iniziale, espresso in *man-week*, il metodo viene adottato per migliorare l'affidabilità del sistema attraverso l'allocazione basata sulla minimizzazione de:

- il numero di difetti residui nel codice; in tal caso si parla di *defect-based allocation*;
- la densità di difetti residui, operando la *defect density-based allocation*.

Tipicamente, un SRGM modella la crescita della *reliability* di un sistema *software* attraverso dei *Non-Homogeneous Poisson Process*, caratterizzati da:

a) λ(t), cioè l'intensità del fallimento;
b) *m(t)* (*mean value function*), che rappresenta il numero cumulativo dei difetti che ci si aspetta di riconoscere al tempo *t*. In particolare,

$$m(t) = a\,F(t)$$

dove:

- a è il numero di difetti totale atteso;
- $F(t)$ è una funzione di distribuzione che può assumere forme diverse, a seconda del processo di occorrenza dei guasti.

Per le funzioni *m(t)* e *λ(t)* vale:

$$N(t): m(t) = E[N(t)];$$

$$\frac{\mathrm{d}m(t)}{\mathrm{d}t} = \lambda(t)$$

Ciò fornisce indicazioni sia su quanti difetti siano stati rilevati nel corso del *testing*, sia una predizione del numero di difetti che il test potrà individuare entro il tempo *t*.

Riprendendo le finalità del metodo, le due misure che sarà possibile minimizzare sono:

- *E[Defect]*, cioè il numero previsto di difetti residui;
- *E[Density]*, cioè la densità attesa dei difetti residui, misurata come: $\frac{numero\ di\ defect}{KLoC}$

Le ipotesi su cui si basa il metodo proposto sono le seguenti:

- I componenti del sistema sono unità testabili e deployabili in modo indipendente gli uni dagli altri;
- Il *budget* disponibile per le attività di test è pari a *B*;
- La quantità di *testing effort* che il *test manager* allocherà sull'i-esimo componente è indicato come W_i *man weeks* (assunto che il numero di forza lavoro settimanale sia stato fissato a priori).

Il criterio si basa sull'idea di adoperare i modelli SRGM per predire iterativamente la capacità di *defect detection* del processo di *testing* effettuato su ogni componente del sistema, così da allocare maggiori risorse sui componenti i cui processi di *testing* mostrano, in un dato intervallo di tempo, una maggiore capacità di *detection*.

Poiché non esiste un SRGM generico, adattabile a qualsiasi contesto, per selezionare il modello più adeguato al caso in esame, la rappresentatività della distribuzione dei *failing testing data* ipotizzata è stata verificata eseguendo il *fit* (mediante l'algoritmo *Expectation – Maximization*[20]) con tutti i modelli SRGM riportati in Tabella 3.

Tabella 3 Modelli SRGM adottati nello studio condotto

COVERAGE FUNCTION	PARAMETERS	m(t)	Failure Occurrence Rate per Fault
Exponential	a, g	$a(1 - e^{-gt})$	Constant
Weibull	a, g, ϒ	$a(1 - e^{-gt^{\Upsilon}})$	Increasing/Decreasing
S shaped	a. g	$a[1 - (1 + gt)e^{-gt}]$	Increasing
Log – Logistic	a, ω, κ	$a\frac{(\omega t)^{\kappa}}{1+(\omega t)^{\kappa}}$	Inverse Bath Tub

[20] EM è un algoritmo iterativo molto utilizzato. Esso consente di calcolare stime di massima verosimiglianza di particolari parametri, nel caso di *dataset* incompleti.

I passi principali del metodo sono cinque, cioè:

1) ***Initialization***: Il *testing* dei componenti ha inizio al tempo t_0, istante in cui potrebbero non essere ancora disponibili i *testing data* necessari a costruire un SRGM (a meno che non siano disponibili da sessioni precedenti di utilizzo o di test dello stesso componente – una assunzione che tuttavia non è necessario fare). Di conseguenza, in mancanza di informazioni iniziali, il *budget* di *test* viene distribuito uniformemente tra tutti i componenti. Al contrario, quando sono già disponibili, i *testing data* possono fornire informazioni aggiuntive sui componenti a maggiore priorità, quindi possono essere utilizzati come punto di partenza per costruire il modello SRGM.
2) ***Start-up check***: durante il quale si verifica per ogni unità di tempo (che nel caso in esame è una settimana) se la procedura di *optimal allocation* possa essere applicata usando i dati disponibili sui difetti rilevati. In particolare, per ogni componente:
    a) adoperando l'algoritmo EM, viene effettuato il *fit* dei dati con tutti i modelli SRGM riportati in Tabella 3;
    b) Per valutare la bontà di adattamento di uno specifico modello rispetto al campione di dati, si esegue un test statistico di *goodness-of-fit* (GoF). Allo scopo viene utilizzato il Kolmogorov- Smirnov (KS) test col 90% di *confidence level*.
    c) Se l'esito del GoF è soddisfacente per almeno un modello SRGM[21], il componente viene etichettato come statisticamente valido.

    La fase di *start-up check* può essere ripetuta automaticamente ogni volta che un tester lo ritenga necessario. Questa fase può ritenersi conclusa non quando esiste almeno un SRGM valido per ciascun componente, ma anche quando solo un sottoinsieme dei componenti è statisticamente valido. In quest'ultimo caso, la *optimal allocation* viene applicata solo a tale sottoinsieme.

3) ***SRGM Selection***, in cui, dati gli N componenti di un sistema ed il relativo *set* di modelli SRGM, per ciascuno di essi viene calcolato il valore di AIC (*Akaike Information Criterion*[22]), al fine di selezionare lo SRGM più adeguato. In particolare, per ciascun SRGM che soddisfa il KS test, viene calcolato:

$$AIC = -2\ \log[L] + 2\phi$$

dove:

- ϕ è il numero di gradi di libertà del modello SRGM;

- L è il valore massimizzato della funzione di probabilità del modello stimato.

[21] Solitamente, il GoF rileverà che esiste più di un SRGM adeguato per ogni componente. Ciò viene gestito nel paso successivo.
[22] Lo Akaike Information Criterion (AIC) è una statistica che consente di selezionare il modello con la somma degli errori al quadrato più piccola (che corrisponde allo AIC più piccolo).

Il valore AIC di un SRGM esprime la perdita di informazione in cui si incorrerebbe adottando il modello. Pertanto, poiché l'obiettivo è quello di disporre di più informazioni possibili, una volta eseguito il *test*, per ciascun componente viene selezionato il modello SRGM che presenta il minor valore AIC, cioè quello su cui si ha una minore dispersione di informazioni.

Come innanzi accennato, può accadere – specie all'inizio del *testing* – che per uno o più componenti non siano disponibili i dati necessari e che, di conseguenza, nello *step* di *start-up check* non si possa risalire ad un SRGM statisticamente valido per ciascun componente. In tal caso, non essendo plausibile nella iterazione corrente trovare la *optimal allocation*, ad ognuno di questi componenti viene destinato un *testing effort* proporzionale alla capacità di *detection* mostrata, mentre sui restanti viene applicati la *optimal allocation*[23]. Così facendo, non essendo interrotto il *testing*, si potrà continuare a reperire dati utili per la costruzione di modelli SRGM adeguati;

4) ***Optimization***: Al termine del passo precedente, ai vari componenti sono associati SRGM caratterizzati ciascuno da una propria *mvf* (cfr. Tabella 3). Si noti che il parametro *a* delle *mean value function* dei modelli SRGM rappresenta il numero totale stimato dei *defect* del componente; in particolare, per l'i-esimo componente, tale valore viene indicato con EST_i.
Attraverso la *mvf* è possibile calcolare anche il numero atteso dei *defect* identificati dopo un certo tempo di testing *t*.
La differenza tra questi due valori fornisce una previsione sul numero dei difetti residui, sulla quale vengono costruite le funzioni obiettivo del metodo esposto.
Si ricordi che, definiti come:

- N, il numero dei componenti del sistema;

- EST_i, il numero atteso di *defect* nel componente i-esimo, stimato tramite il suo SRGM;

- *B*,* il budget iniziale

- W_i, il *testing effort* allocato sull' i-esimo componente (si noti che il *testing effort* totale non deve eccedere il budget B*);

- W_i^*, il *testing effort* che è stato già allocato sull' i-esimo componente;

- $m(W_i^* + W_i)$, il numero stimato dei *defect* che possono essere rimossi se l' i-esimo componente riceve un *testing effort* pari a $(W_i^* + W_i)$;

- $SIZE_i$, la dimensione in *KLoC* dell' i-esimo componente,

la prima funzione:

[23] Si noti che è stato osservato che dopo non più del 20% del tempo di testing totale, viene rilevato almeno un SRGM valido per ogni componente. Pertanto, si consiglia di iniziare la fase di start-up check dopo che sia trascorso il 10-15% del tempo di testing iniziale, in modo che tutti abbiano un SRGM al termine del passo 3.

$$min!\, E[Defect] = \sum_{i=1}^{N} (EST_i - m(W_i^* + W_i))$$

$$\text{s. a.} \qquad \sum W_i \leqslant B^*$$

è finalizzata alla identificazione della *defect-based allocation*; mentre la seconda:

$$min!\, E[Density] = \sum_{i=1}^{N} \left(\frac{EST_i - m(W_i^* + W_i)}{SIZE_i}\right)$$

$$\text{s. a.} \qquad \sum W_i \leqslant B^*$$

è rivolta alla *defect density–based allocation*.

5) ***Dynamic update***: il *testing effort* viene allocato al passo precedente; tuttavia la misura dell'allocazione viene aggiustata dinamicamente mano a mano che i dati disponibili diventano più cospicui. In questo modo vengono considerate anche le eventuali variazioni nel processo di *testing* che potrebbero invalidare la precedente soluzione. Tutti i passi precedenti vengono ripetuti ad intervalli di tempo stabiliti, in modo che le risorse di test vengano ri-allocate sui singoli componenti sulla base dei nuovi dati prodotti dall'avanzamento delle attività di V&V.

Per completezza, di seguito è riportata una versione semplificata dell'algoritmo di allocazione realizzato:

The dynamic allocation algorithm.

Input: *budget*, *nComp*, SIZE, *reallocStep*, *startCheck*;

//upper case variables are arrays; lower case are scalar

1. *i*=1; *t*=0; *ready*=false
2. while *i* ≤ *nComp* do
3. W*i =W*i = *budget/nComp;//Uniform Allocation*
4. end while
5. *// start testing*
6. while !*ready* do
7. *// do testing*
8. *t*++;
9. if (*t* ≥ *startCheck*)
10. D-DATA = readData();*// read defect data*
11. SRGM-MATRIX = fitData(D-DATA);
12. *ready*=checkSRGM(SRGM-MATRIX);
13. end if
14. end while
15. *toAllocate = updateBudget(budget, t); //for the next step*
16. while (*toAllocate* > 0) do
17. SRGM = selectSRGM(SRGM-MATRIX);
18. W = solveOpt(SRGM,D-DATA,SIZE,W*,*toAllocate*);
19. *//compute actual effort employed*
20. *r* = computeResidualEffort(W, *t*);
21. if (*r* !=0) W' = allocateResidualEffort(W, r);
22. else W'=W;
23. *// do testing with the allocated W* man-weeks*
24. *t = t+ reallocStep*; *//for the next step*
25. *toAllocate = updateBudget(budget, t); //for the next step*
26. if (*toAllocate* > 0)
27. D-DATA = readData();
28. SRGM-MATRIX= fitData(D-DATA);
29. end if

30. end while

31. end

Come sopra esposto, inizialmente l'algoritmo ripartisce equamente il *budget* tra tutti i componenti del sistema, per poi passare alla fase di *start-up checking*. Durante questa fase, i dati sui difetti rilevati nel *testing* di ciascun componente vengono inseriti nella matrice D-DATA; viene eseguito il *fit* con ogni SRGM mostrato in Tabella 3 ed il test KS. A questo punto, la matrice SRGM-MATRIX contiene tutti gli SRGM statisticamente validi per ciascun componente. Tale matrice si dirà *ready* (*ready* pari a *true*) quando tutti i componenti avranno almeno un SRMG valido; a tal proposito, attraverso la funzione *checkSRGM*, il metodo consente di gestire i casi in cui un *tester* voglia proseguire solo su un sotto-insieme dei componenti statisticamente validi.

Quando sono disponibili dati sufficienti, vengono eseguiti gli step 3), 4) e 5); quindi vengono allocati i *testing effort*, mentre le risorse rimanenti vengono riassegnate e ricalcolate in modo iterativo. Successivamente, per ciascun componente viene selezionato il modello SRGM più adeguato (in base al valore AIC), dal quale si ottengono le basi per la risoluzione del problema di ottimizzazione. Si noti che, una volta che il *tester* ha scelto se minimizzare il numero dei difetti residui o la *defect-density*, la funzione *solveOpt* (linea 18) provvede a trovare la soluzione del problema, dopodiché, l'algoritmo ricalcola l'ammontare delle risorse residue che potranno essere riallocate.

Queste operazioni vengono ripetute iterativamente, finché le risorse non sono state esaurite.

Lo studio esposto ha evidenziato che la frequenza di riallocazione delle risorse dipende principalmente da come varia nei componenti in esame la tendenza con cui si manifesta il rilevamento dei difetti. In particolare, per non incappare in inutili calcoli, è importante che tra due assegnazioni consecutive intercorra un lasso temporale sufficiente ad osservare delle variazioni significative nei *detection rate* rilevati. In generale, un *tester* può scegliere di riassegnare le risorse di *testing* ad intervalli temporali prefissati (ad esempio, ogni 4 settimane), ma anche ogni volta che lo ritiene necessario.

4.2 SPERIMENTAZIONE DEL METODO

Il metodo è stato sperimentato nell'ambito del progetto su un sotto-sistema di SELEX–ES, costituito da un insieme di 5 *Computer Software Configuration Item* (CSCI).

L'esperimento è stato condotto simulando allocazioni diverse sulla base di *testing data* raccolti durante attività reali di *testing* sui CSCI, effettuate tra il 2009 ed il 2012. Durante detto periodo, si è riscontrato un numero totale di *bug* pari a 1119, rilevati in 326 *man/week*.

Lo studio è consistito nel confronto tra differenti schemi di allocazione, al fine di dimostrare le maggiori potenzialità del metodo dinamico rispetto a quelli statici e tale comparazione è stata svolta in base al numero complessivo dei difetti rilevati.

In particolare, assumendo di adoperare lo stesso processo di *testing* e gli stessi *dataset*, per il confronto sono stati considerati gli schemi di:

1) Allocazione uniforme;
2) Allocazione *size-based*;
3) Allocazione *Defect – based*;
4) Allocazione *Density Defect – based*.

Inoltre, è stato assunto un *budget* iniziale pari a 150 *man/weeks*, da distribuire su 5 CSCI; la prima riallocazione è stata fissata dopo 8 settimane dall'inizio, mentre le osservazioni successive sono state effettuate ogni 4 settimane[24].

Come mostrato in Tabella 4, secondo lo schema di allocazione uniforme, il *budget* iniziale viene equamente distribuito tra i cinque CSCI, quindi su ciascuno di essi sono state allocate 30 man-week. In tal caso, osservando i risultati del processo di test reale e i difetti rilevati in ciascun CSCI dopo 30 man-week, il numero di difetti rilevato sarebbe stato pari a 488.

Tabella 4 Risultati dello schema di allocazione uniforme

	C1	C2	C3	C4	C5	Total det. defects	Residual man-weeks
Allocated man-weeks	30	30	30	30	30		
Detected defects per CSCI							
After 8 weeks	10	6	107	9	31	163	110
After 12 weeks	11	6	112	46	36	211	90
After 16 weeks	12	6	129	55	37	239	70
After 20 weeks	27	15	133	57	44	276	50
After 24 weeks	33	44	150	62	65	354	30
After 28 weeks	51	110	159	62	85	467	10
After 30 weeks	63	114	160	62	89	488	0

[24] La frequenza di riallocazione è stata impostata (a 8 e 4 settimane) dopo aver valutato attentamente la disponibilità di un numero adeguato di dati sui difetti rilevati.

Nello schema *size-based* (Tabella 5), su ciascun CSCI è stata allocata una quantità *testing effort* proporzionale alla dimensione del CSCI. Si noti che i CSCI ai quali è stata assegnata una quantità di risorse inferiore a 30 *man-weeks* hanno esaurito le proprie risorse prima dell'ultima osservazione; tuttavia, con questo schema sarebbero stati rilevati 523 *defect*, cioè 35 in più rispetto allo schema di allocazione uniforme.

Tabella 5 Risultati dello schema size-based

	C1	C2	C3	C4	C5		
Allocated man-weeks	27.46	38.78	15.61	41.87	26.29		
Detected defects per CSCI						**Total det. defects**	**Residual man-weeks**
After 8 weeks	10	6	107	9	31	163	110
After 12 weeks	11	6	112	46	36	211	90
After 16 weeks	12	6	114	55	37	224	70
After 20 weeks	27	15	114	57	44	257	50
After 24 weeks	39	54	114	76	65	348	30
After 28 weeks	51	148	114	94	68	475	10
After 30 weeks	51	162	114	128	68	523	0

A differenza degli approcci statici appena mostrati, gli schemi di allocazione *defect-based* e *defect density-based* hanno una natura dinamica. Di conseguenza, ogni osservazione coincide con una riallocazione delle risorse sui CSCI. In particolare, solo quando vengono rilevati nuovi difetti viene costruito un nuovo SRGM per un CSCI; esso fornisce il valore di EST_i, ossia di numero di difetti stimato. Quest'ultimo valore viene utilizzato per prevedere una stima sul numero di difetti rilevabili qualora sull'i-esimo componente venisse allocata la quantità di *testing effort* suggerita dal modello. Tale schema tiene conto anche della riallocazione del *testing effort* residuo sui componenti che mostrano un migliore *trend* di *defect detection*.

Come mostrato in Tabella 6, al termine dell'analisi si è potuto osservare che l'allocazione *defect–based* avrebbe consentito la *detection* di 552 *defect*, con un aumento del 13,11% rispetto allo schema uniforme e del 5,5% rispetto a quello *size-based*; mentre è stata stimata una *defect density* $\sum_i \frac{EST_i - Detected_i}{SIZE_i}$ pari a 0,5869 defect/KLoC.

Come riportato in Tabella 7, nel caso dell'allocazione *density-based* sarebbero stati rilevati 547 difetti, pari cioè al 12,09% in più rispetto allo schema uniforme e al 4,58% in più rispetto a quello *size-based*, mentre è stata stimata una *defect density* di 0,5572 *defect/KLoC*.

Tabella 6 Risultati della defect-based allocation

Iteration Time in weeks	Number of detected defects (DET) / Estimated defects (EST) / Predicted detections (m(W*+W))					Tot. Det. Def.	Man-weeks to (re-) allocate	Reallocation of man-weeks (#of allocated man-weeks)				
	C1	*C2*	*C3*	*C4*	*C5*			*C1*	*C2*	*C3*	*C4*	*C5*
0 weeks (t0)												
DET	0	0	0	0	0	0	150	30	30	30	30	30
8 weeks												
DET	10	6	107	9	31	163	110	5.93	5.93	52.41	5.99	39.72
EST	11.00	6.02	128.67	9.01	35.12							
m(W*+W)	10.99	6.00	128.66	9.00	35.11							
12 weeks												
DET	11	6	112	46	36	211	90	1.41	0	26.69	33.06	28.83
EST	11.00	6.02	118.97	1,105.58	39.04							
m(W*+W)	10.99	6.00	118.97	1,105.5	39.00							
16 weeks												
DET	11	6	132	55	40	244	70	0	0	38.45	1.69	29.86
EST	11.00	6.02	140.92	55.53	42.52							
m(W*+W)	10.99	6.00	140.90	55.52	42.50							
20 weeks												
DET	11	6	159	57	68	301	50	0	0	0	0	50
EST	11.00	6.02	171.62	62.69	355.80							
m(W*+W)	10.99	6.00	159.08	57.00	169.09							
24 weeks												
DET	11	6	159	57	239	469	30	0	0	0	0	30
EST	11.00	6.02	171.62	62.69	432.94							
m(W*+W)	10.99	6.00	159.08	57.00	331.13							
28 weeks												
DET	11	6	159	57	312	545	10	0	0	7.84	2.15	0
EST	11.00	6.02	171.62	62.69	316.43							
m(W*+W)	10.99	6.00	166.02	61.10	316.26							
30 weeks												
DET	11	6	161	62	312	552	0	-	-	-	-	-

Tabella 7 Risultati della density-based allocation

Iteration Time in weeks	Number of detected defects (DET) / Estimated defects (EST) / Predicted detections (m(W*+W))					Tot. Det. Def.	Man-weeks to (re-) allocate	Reallocation of man-weeks (#of allocated man-weeks)				
	C1	C2	C3	C4	C5			C1	C2	C3	C4	C5
0 weeks (t0)												
DET	0	0	0	0	0	0	150	30	30	30	30	30
8 weeks												
DET	10	6	107	9	31	163	110	17.72	17.72	37.94	17.72	18.89
EST	11.00	6.02	128.67	9.01	35.12							
m(W*+W)	11.00	6.01	128.66	9.00	35.11							
12 weeks												
DET	11	6	112	46	36	211	90	0.15	0.14	34.91	31.33	23.44
EST	11.00	6.02	118.97	1,105.58	39.04							
m(W*+W)	11.00	6.01	118.97	1,105.5	39.00							
16 weeks												
DET	11	6	133	55	40	245	70	0.87	0.84	38.74	0.85	28.69
EST	11.00	6.02	140.36	55.53	42.52							
m(W*+W)	11.00	6.01	140.34	55.52	42.50							
20 weeks												
DET	11	6	159	57	68	301	50	0	0	1.33	0	48.66
EST	11.00	6.02	171.62	62.75	355.80							
m(W*+W)	11.00	6.01	161.76	55.98	166.62							
24 weeks												
DET	11	6	160	57	189	423	30	0	0	0	0	30
EST	11.00	6.02	168.57	62.69	405.72							
m(W*+W)	11.00	6.01	160.01	55.98	329.67							
28 weeks												
DET	11	6	160	57	312	546	10	0	0	2.65	0	7.35
EST	11.00	6.02	168.57	62.75	337.03							
m(W*+W)	11.00	6.01	161.86	55.98	325.47							
30 weeks												
DET	11	6	161	57	312	547	0	-	-	-	-	-

Dagli esiti delle allocazioni *defect–based* e *density defect-based* (Tabella 6 e 7) si è osservato che fino alla ventiquattresima settimana i risultati prodotti dalle due allocazioni sono grosso modo simili, mentre successivamente – per le diverse allocazioni effettuate sui componenti C3, C4 e C5 - si osservano degli scostamenti. In particolare, lo schema *defect density-based* distribuisce in modo più bilanciato le risorse di

testing sui componenti, tuttavia il risultato finale si avvicina molto a quello ottenuto mediante lo schema di allocazione *defect-based*

La Figura 4 riassume i risultati finora illustrati: dal confronto tra i quattro schemi di allocazione discussi, si evince che quelli dinamici sono più performanti nel tempo.

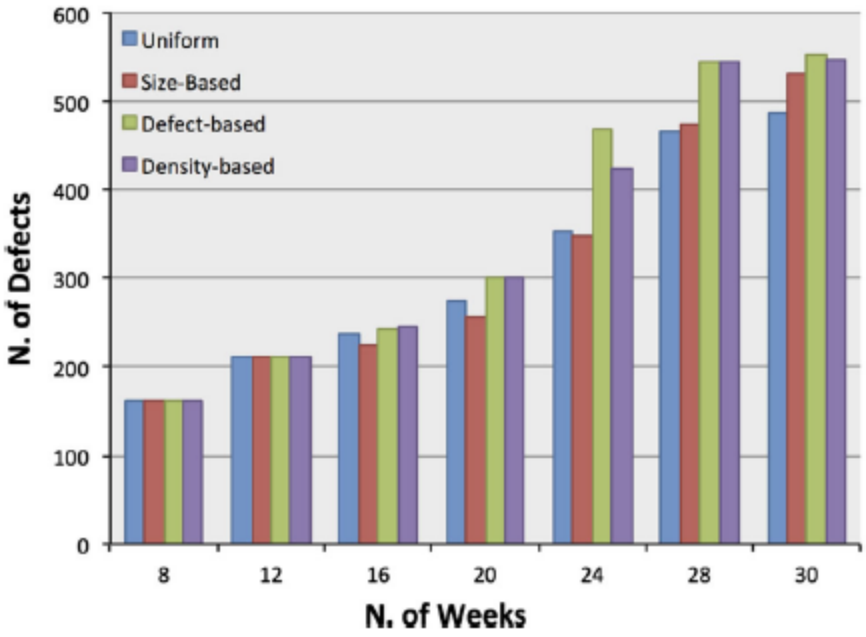


Figura 4 Confronto tra gli schemi di allocazione: il numero di defect rilevati durante il testing

Le assunzioni alla base dell'applicazione di modelli SRGM già discussi e riportati nuovamente in Tabella 8 per comodità, possono essere facilmente violati in un contesto reale. Ciò può dipendere da cambiamenti dell'ambiente in cui si opera, da fattori tecnici, umani e tecnologici. Il metodo di allocazione discusso è una soluzione robusta anche in queste circostanze, poiché è in grado di ricalcolare dinamicamente la distribuzione dell'*effort* di test ai sotto-sistemi/componenti in esame, consentendo di risparmiare risorse preziose che potranno essere così impiegate laddove serve.

Tabella 8 Assunzioni alla base dell'applicazione degli SRGM

Faults are repaired immediately when they are discovered
Fault repair is perfect
Faults are independent of each other
No new product features were introduced during test
The test script remained constant over the stability test period
All reliability improvements are caused by solving the open stability faults
An NHPP can describe the fault detection and removal process
Each unit of test execution time is equally likely to find a failure
Testing follows an operational profile

Rispetto agli studi effettuati nel passato, il valore aggiunto del metodo esposto è di aver mostrato concretamente come applicare un criterio SRGM-*based* in un contesto industriale reale, dove solitamente per il *test design* non vengono adottati approcci quantitativi. In tali contesti, la violazioni delle assunzioni può essere severa, ed avere un metodo che si "adatti" a tali variazioni consente di poter applicare un approccio quantitativo, preservando i vantaggi degli SRGM.

5 IDENTIFICAZIONE DI MODULI CRITICI TRAMITE UN MODELLO DI FAULT-PRONENESS

Le tecniche di *fault-proneness prediction* (semplicemente *fault prediction*) operano mediante la costruzione di modelli predittivi, che vengono addestrati con *dataset* reali che rappresentano alcune caratteristiche del sistema oggetto della *prediction*. I modelli predittivi vengono così adoperati allo scopo di indirizzare gli sforzi delle attività di V&V verso le parti critiche del sistema che mostrano una tendenza più spiccata a contenere *fault*.

Come discusso nella sezione 3, i modelli di *fault prediction* possono certamente fornire indicazioni utili a questo scopo. Tuttavia, sarebbe di estrema utilità ottenere una indicazione più a grana fine sulla criticità dei moduli, basata non solo sulla propensione di un modulo a contenere *fault*, ma anche rispetto alla "importanza" dei *fault* contenuti. Infatti, operare tale distinzione consentirebbe di allocare le risorse di *testing* ai componenti *software* in modo più efficace, potendo suggerire l'applicazione di tecniche diverse per identificare e rimuovere tipologie di *fault* diverse.
È acclarato che i *fault* sono i principali responsabili dei fallimenti del *software* e che uno degli strumenti principali per rimuovere i *fault* siano le tecniche di V&V. Tuttavia esistono *fault* più o meno "complessi" dal punto di vista della V&V: si considera complesso un *fault* la cui identificazione è complessa, poiché il relativo *pattern* di attivazione è difficilmente riproducibile in fase di V&V.

Una distinzione tipica, utile nel contesto dei sistemi critici, è la seguente: "*Bohrbug*" , "*Mandelbug*" ed "*Heisenbug*" [39]. I primi sono *fault* permanenti, che possono essere isolati e risolti durante il normale test funzionale e *debug*; data la natura deterministica, essi sono facilmente riproducibili, in quanto il *pattern* di attivazione e propagazione è poco complesso. I secondi hanno una natura apparentemente "non-deterministica", in quanto le relative condizioni di attivazione dipendono fortemente dall'ambiente di esecuzione, pertanto, esse sono complesse e difficili da riprodurre (ad esempio, *bug* attivati solo in corrispondenza di un particolare *scheduling dei* thread del sistema operativo). Di conseguenza, i *Mandelbug* sono difficilmente individuabili in fase di testing, e richiederebbero tecniche diverse dal semplice test funzionale. In particolare, i *pattern* di attivazione e/o di propagazione di un *Mandelbug* possono essere particolarmente complessi, poiché sono influenzati: dalla interazione tra lo stato interno dell'applicazione e le condizioni che si verificano nel "*system-internal environment*" (ad esempio, l'interazione con un *driver*); dai lunghi ritardi che potrebbero intercorrere tra l'attivazione e la manifestazione del *fault* all'interfaccia del sistema, che può inficiare l'individuazione della causa scatenante il fallimento.

Gli *Heisenbug,* visti in passato come il complemento dei *Bohrbug* [12], sono successivamente stati classificati come un sottoinsieme dei *Mandelbug* [39]; essi si riferiscono a guasti transienti, la cui manifestazione è temporanea. Ciò rende questi *bug* difficilmente osservabili, poiché, quando si tenta di riprodurli, non si manifestano nuovamente.

Un ulteriore sottoinsieme dei *Mandelbug*, che da alcuni anni sta assumendo una notevole rilevanza, è rappresentata dai "*Related Aging*[25] *Bug*", che si manifestano dopo lunghi periodi di esercizio del sistema, provocando un aumento del tasso di fallimento e/o un peggioramento delle prestazioni dello stesso.

Le comuni tecniche di test sono realmente efficaci quando un fallimento è riproducibile, quindi si rilevano molto utili nel caso di *Bohrbug*; tuttavia, possono rilevarsi inefficaci per la categoria dei *Mandelbug*. Per questi ultimi, possono essere più efficaci tecniche di verifica basate sull'analisi statica o dinamica del codice, oppure tecniche di *fault tolerance* a tempo operazionale.

Nell'ambito del progetto è stato sviluppato un metodo di *fault prediction* "*type-aware*" che distingue le due macro tipologie di *fault* sopra citate. A tale scopo sono stati definiti modelli di predizione per capire quanto fosse possibile predire la classe dei *bug* complessi (*Mandelbug)*, definendo metriche "opportune" (che si è ipotizzato fossero correlate all'una o all'altra tipologia di *fault*) ed utilizzando queste ultime con metriche tradizionali (come, ad esempio: la complessità ciclomatica, linee di codice, etc.).

Il metodo, descritto di seguito, è stato preliminarmente sperimentato su un sistema industriale di SELEX ES, nel dominio militare, con una dimensione pari a 1,3 MLoC e costituito da 23 moduli (dove per moduli si intendono sia CSCI, che i relativi sotto-componenti).

Lo studio è partito dalle informazioni relative ai problemi rilevati nel sistema in esame, raccolte tra il 2011 ed il 2012, durante il *test* di integrazione dello stesso. Per l'analisi, oltre a tali informazioni, memorizzate in insiemi di *problem report*[26] attraverso un sistema di *bug tracking*, sono state utilizzate anche le informazioni a corredo dei *report*. Attraverso queste ultime è stato possibile: analizzare la distribuzione dei *fault* nei diversi moduli, classificare la tipologia dei *fault* rilevati ed identificare i moduli coinvolti.

Ai fini dell'analisi, sono stati scartati i *report* relativi a problemi non imputabili a *software fault*, mentre i fallimenti attribuibili ad uno stesso *fault* sono stati raggruppati e considerati come un'unica istanza. È stata adottata la seguente distinzione, basata sulle condizioni di attivazione e sulla *error propagation* tratta dalle informazioni presenti nei *report*:

- *Bohrbug* (BOH);
- *Mandelbug* (MAN);
- *Unknown* (UNK), per indicare tutti le tipologie di difetti non attribuibili alle due classi precedenti.

Per poter classificare i fallimenti in *Mandelbug* o *Bohrbug*, all'interno di ciascun *report* sono state reperite le informazioni relative a:

- condizioni di attivazione, cioè i *trigger*;

[25] Il *software aging* è un fenomeno in cui si assiste al degradamento dello stato del software o del suo ambiente di esecuzione. Questo è causa della riduzione dell'affidabilità di un sistema *software*.
[26] Le informazioni contenute in un problem report in esame sono: *id, reporter, data di sottomissione, severity, priority, riproducibilità, passi per la riproduzione, categoria, versione, risoluzione, stato, sommario, descrizione, tipo.*

- *error propagation*, che fornisce un'indicazione di come il *bug* influenzi il sistema e come si propaghi attraverso di esso;
- *failure behavior*, cioè il comportamento che mostra il sistema al verificarsi del fallimento.

In base alla criticità delle condizioni di attivazione ed alla *error propagation*, si è poi provveduto a verificare se i *bug* considerati ricadessero nella categoria dei *Mandelbug*; al contrario, i *bug* per cui non è stata fornita tale evidenza sono stati identificati come *Bohrbug*, oppure come *Unknow*.

Ai fini dell'esperimento, sono stati ispezionati 463 *report*, ottenendo i seguenti risultati (Figura 5):

- 202 *problem* (43.63%) sono stati evidenziati come non classificabili (indicati quindi come NOC), quindi non sono stati considerati ai fini dell'analisi;
- 261 *problem* sono stati classificati come *Bohrbug* (78,93%) e *Mandelbug* (14,56%).

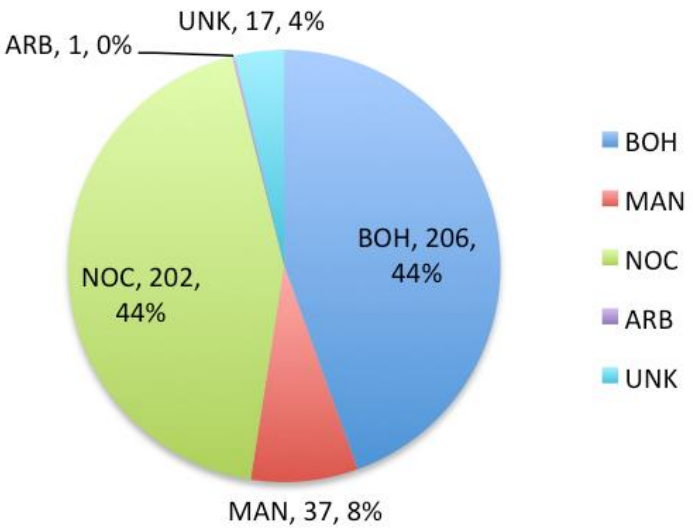


Figura 5 Risultati della classificazione

Come si può notare dalla Figura 6, la distribuzione di *Bohrbug* e *Mandelbug* è uniforme sui moduli considerati, tuttavia alcuni moduli hanno presentato un numero elevato di *Mandelbug* (C4, C6 e C7) rispetto ad altri che ne hanno mostrato solo uno (come C10, ad esempio).

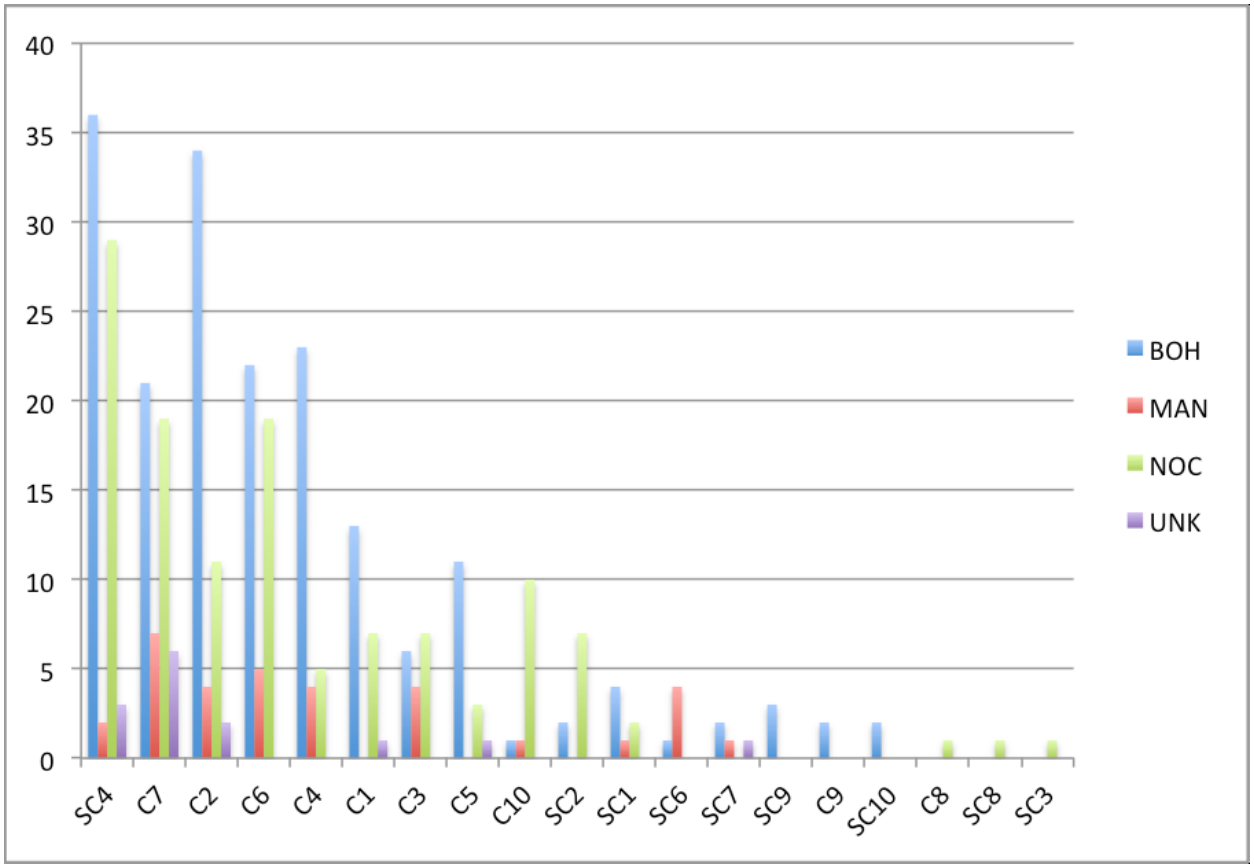


Figura 6 Distribuzione dei bug nei moduli

Questo risultato, in linea con il principio di Pareto, indica che la maggior parte (80%) dei *fault* è localizzata in piccole porzioni (20%) del codice del sistema oggetto dell'analisi.

A partire da questa evidenza, lo studio si è concentrato sulla definizione di una strategia di *fault prediction*, in grado di anticipare quali siano le parti critiche e maggiormente inclini a contenere *fault* di un sistema di tipo *Mandelbug* così da poter:

- Destinare un maggiore *effort* di *test* ai moduli con più *fault* attesi;
- Distinguere la tipologia di attività di V&V (ad esempio, test di stress e di robustezza o analisi statica per i Mandelbug) e/o pianificare adeguate strategie di *fault-tolerance* per i moduli propensi a contenere *Mandelbug.*

al fine di massimizzare il profitto dalle attività di V&V.

LE METRICHE INTRODOTTE NELLO STUDIO

Dopo aver classificato i *bug* presentati nei *report*, si è passati alla definizione di nuove metriche, che si sono dimostrate capaci di identificare con una certa accuratezza i moduli *Mandelbug-prone* (cioè più inclini a contenere *Mandelbug*). Come appena menzionato, oltre ad un insieme di metriche tradizionali (come la complessità di McCabe e le metriche di Halstead), nel lavoro sono state introdotte delle metriche, che si è ipotizzato fossero in relazione con la presenza di *Mandebug*. In particolare, sono state considerate delle metriche basate su fattori che possono causare indirettamente l'attivazione ed il manifestarsi di un *Mandelbug*, quali:

a) **Metriche basate sull'utilizzo costrutti *concurrency-related***, relative ai cosiddetti "*concurrency-related fault*", cioè derivati da problemi che sorgono a causa di errori nella programmazione concorrente[27];
b) **Metriche basate sull'utilizzo di librerie di I/O**, dato che molti *fault* sono attivati a causa delle interazioni con il *system internal environment* (specie con i *driver* dei dispositivi);
c) **Metriche basate sull'utilizzo di costrutti e regole di "Exception Handling"**, i *fault* relativi alla gestione delle eccezioni sono difficili da individuare anche durante il *debug*, dal momento che sono attivati in seguito ad eventi eccezionali e causati dalle interazioni.

Tute le metriche sono riportate in Tabella 9 (metriche tradizionali) ed in Tabella 10 (nuove metriche). Per ciascun modulo *software* (*file* e classi) è stato calcolato il valore medio di ciascuna metrica sopra citata, dando luogo ad un vettore di metriche.

[27] I problemi più frequenti legate alla programmazione concorrente riguardano ad esempio l'uso scorretto dei lock (deadlock) oppure dalla mancata sincronizzazione (è il caso delle *race condition*).

Tabella 9 Metriche di complessità del software tradizionali

TYPE	METRICS	DESCRIPTION
Program size	AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, CountDeclFunction, CountInput, CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountOutput, CountStmt, CountStmtDecl, CountStmtExe, CountPath, CountSemicolon, CountLineCodeExe, CountLineComment, RatioCommentToCode	Metrics related to the amount of lines of code, declarations, statements, and files
McCabe's cyclomatic complexity	AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, Cyclomatic, CyclomaticModified, CyclomaticStrict, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxNesting, SumCyclomatic, SumEssentialCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential	Metrics related to the control flow graph of functions and methods
Halstead metrics	Program Volume, Program Length, Program Vocabulary, Program Difficulty, Effort, N1, N2, n1, n2	Metrics based on operands and operators
Object-oriented metrics	CountClassBase, CountClassCoupled, CountClassDerived, CountDeclClass, CountDeclClassMethod, CountDeclClassVariable, CountDeclInstanceMethod, CountDeclMethod, MaxInheritanceTree, CountDeclMethodProtected, CountDeclMethodPublic, CountDeclMethodPrivate, CountDeclIstanceMethod, CountDeclMethodAll, CountDeclMethodDefault, CountDeclInstanceVariable, PercentLackOfCohesion	Metrics based on object-oriented constructs

Tabella 10 Le nuove metriche di complessità del software introdotte

NAME	DESCRIPTION	DOMAIN
CountTryBlocks	Number of try blocks	N
AtLeastOneTryBlock	At least one try block is present in the class	{0,1}
CountCatchBlocks	Number of catch blocks	N
StaticLibraryMethodCalls	Number of calls to static methods of the System package and of the Runtime.getRuntime() object	N
AtLeastOneStaticLibraryMethodCall	At least one call according to StaticLibraryMethodCalls is present in the class	{0,1}
UniqueStaticLibraryMethodCalls	Number of calls according to StaticLibraryMethodCalls, where calls to the same method are counted only one time	N
LibraryMethodCalls	Number of calls to methods of the java.io and java.net packages	N
AtLeastOneLibraryMethodCall	At least one call according to LibraryMethodCalls is present in the class	{0,1}
UniqueLibraryMethodCalls	Number of calls according to LibraryMethodCalls, where calls to the same method are counted only one time	N
ClassThread	The class is a descendant of the Thread class, or it implements the Runnable interface	{0,1}
CountSynchronizedMethods	Number of synchronized methods	N
CountSynchronizedBlocks	Number of synchronized blocks	N
CountLockCalls	Number of calls to instances of the Lock, Semaphore, ReentrantLock classes	N
CountWaitNotify	Number of calls to wait(), notify(), notifyAll(), await(), signal(), signalAll()	N
ImportIO	The file imports the java.io package	{0,1}
ImportNet	The file imports the java.net package	{0,1}
ImportConcurrent	The file imports the java.concurrent.util package	{0,1}

FAULT-PREDICTION CON MODELLI DI CLASSIFICAZIONE E MODELLI DI REGRESSIONE

Consideriamo due approcci per la predizione: la classificazione e la regressione. Si noti che in entrambi i casi un modulo *software* è stato rappresento attraverso un campione di dati riportanti i valori delle metriche menzionate per quel modulo.

Con la **classificazione**, la predizione consiste nello stabilire l'appartenenza dei moduli ad una ed una sola classe predefinita di *fault*.

Per questo caso, abbiamo **due scenari**:

1) Nel primo, i moduli sono stati ripartiti in due classi distinte (mutuamente esclusive): "*Mandelbug-prone*", per quei moduli che presentano almeno un *Mandelbug* e sui quali si può pensare di concentrare buona parte dell'*effort* di test; "*Mandelbug-free*", per quei moduli che non presentano *Mandelbug*;

2) Nel secondo, i moduli sono stati ripartiti in tre classi, vale a dire: "*No Mandelbug*", equivalente a "*Mandelbug-free*" del primo scenario; "*Exactly one Mandelbug*", che mostrano esattamente un *Mandelbug*; "*More than one Mandelbug*", che presentano più di un *Mandelbug*. Di fronte a tale scenario, un *team* di *test* ha la possibilità di decidere se distribuire maggiori risorse di *test* ai moduli della seconda e/o terza classe.

Affinché un algoritmo sia in grado di classificare nuovi campioni è necessario sottoporlo preliminarmente all'addestramento su un modello predittivo, adoperando campioni di dati noti (ad esempio, i dati di progetti precedenti o versioni precedenti del sistema). Successivamente, il modello addestrato viene utilizzato per classificare i campioni di dati non conosciuti. In particolare, potremmo considerare noti i dati rappresentati dalle informazioni raccolte su versioni precedenti del sistema in esame, mentre i nuovi campioni potrebbero essere i moduli in via di sviluppo, sui quali si vogliono intraprendere delle attività di V&V.

Nello studio descritto sono stati utilizzati cinque diversi algoritmi di classificazione per la fault-prediction: *Decision Trees*; *Support Vector Machines* (SVM); Reti Bayesiane; *Naive Bayes*; *Multinomial Logistic Regression*.

Per quanto riguarda la regressione, la predizione è finalizzata al *ranking* dei moduli, ordinati rispetto al numero atteso di *Mandelbug* predetto; dunque, tali modelli potrebbero essere sfruttati dai *tester* per identificare i moduli su cui concentrare maggiormente il proprio lavoro. Così come avviene per la classificazione, un modello di regressione è costruito sulla base di campioni di dati noti.
Sono stati utilizzati tre modelli di regressione: Regressione lineare; Alberi di regressione; *Support Vector Regression* (SVR).

In entrambi i casi, le metriche utilizzate possono avere un impatto significativo sull'efficacia della previsione. Di conseguenza, allo scopo di migliorare l'efficacia della *fault prediction* è stato incluso, prima dell'addestramento del classificatore o regressore, uno *step pre-processing* di *feature selection*. Le *feature* sono state selezionate mediante un algoritmo s*tep-wise regression*, che ad ogni iterazione valuta un sottoinsieme candidato di *feature*, basandosi sulla *goodness-of-fit* di un modello di regressione lineare multipla.
Tale algoritmo opera fino a quando è possibile migliorare il sottoinsieme di *feature*, aggiungendo e/o rimuovendo da esso un elemento per volta.

A valle di ciò, sono stati considerati due scenari: nel primo, la *fault prediction* viene eseguita a valle dello *step* di *feature selection*, nel secondo caso, invece, si fa a meno della *feature selection*.

EFFICACIA DELLA FAULT-PREDICTION CLASSIFICATION-BASED

Per valutare la bontà del classificatore **(predittore**) nel predire la *fault-proneness* è stato preliminarmente fissato un *training set* (rappresentato da dati noti) sul quale è stato addestrato il modello di classificazione. Successivamente, sono state stimate le capacità predittive del classificatore, valutando la correttezza della classificazione nel *test set* (dati non noti).

In particolare, la valutazione è stata effettuata adoperando l'approccio *k-fold cross-validation* (con k = 3), in cui, dopo aver ripetuto la *cross-validation* per 15 volte, è stata calcolata una media dei risultati. La correttezza è stata determinata mediante il confronto tra la classe prevista di appartenenza di ciascun campione del *test-set* con la classe reale.

Gli indicatori di *performance* utilizzati, che indicano la bontà di un modelli di classificazione, sono:

1) **Precision** (Pr), che rappresenta la percentuale di "*True Positive*" (TP) (ossia, classificazione corretta) rispetto al totale dei moduli classificati come "*Mandelbug-prone*" (si noti che con FP sono stati indicati i *false positive*, ossia erroneamente classificati come *Mandelbug-prone*):

$$Precision = \frac{TP}{TP+FP}$$

2) ***Recall*** (Re), cioè la percentuale di TP rispetto ai moduli sono effettivamente "*Mandelbug-prone*" (si noti che con FN sono stati indicati i *false negative*, ossia i moduli *Mandelbug-prone* non classificati come tali):

$$Recall = \frac{TP}{TP+FN}$$

3) **F-measure** (F): Media armonica tra *Precision* e la *Recall*:

$$F - measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

Si noti che un valore elevato della *Precision* e della *Recall*, indica un'occorrenza molto bassa di FP e FN. Di conseguenza, si può dedurre che il classificatore ha una buona capacità predittiva.

Migliorare la *Precision* riducendo i falsi positivi, si traduce spesso in una peggiore *Recall*, cioè in un aumento dei falsi negativi; pertanto va individuato il giusto *trade-off* tra queste due misure, sintetizzato dalla *F-measure*.

I RISULTATI DELLA CROSS-VALIDATION

In Tabella 12 sono illustrati i risultati della cross-validation effettuata nel caso in cui le classi target siano tre, cioè: "0 M*andelbugs*", "1 *Mandelbug*", "> 1 *Mandelbugs*". Nell'esempio mostrato, sono stati considerati quattro scenari distinti:

1. Utilizzo di metriche tradizionali, senza *feature-selection*;
2. Utilizzo di metriche tradizionali con *feature-selection;*
3. Utilizzo di metriche tradizionali e delle nuove metriche (sopra definite), senza *feature-selection*;
4. Utilizzo di metriche tradizionali, delle nuove metriche (sopra definite) con *feature-selection*.

Per ognuno di essi e per ciascuna classe *target* vengono riportati i valori della *Precision*, della *Recall*, della F-*measure* e la media pesata, calcolati con 5 algoritmi di classificazione diversi: *Decision Trees*; *Support Vector Machines* (SVM); Reti Bayesiane; *Naive Bayes*; *Multinomial Logistic Regression*.

La Tabella 11 mostra l'insieme di metriche selezionate attraverso la *feature-selection*, insieme alle misure di R^2 e Adj-R^2 che indicano la *goodnes-of-fit* del modello di regressione multipla lineare, che fa uso di tali *feature*.

Tabella 11 Feature Selection relativa alla classificazione a 3 classi

METRICS	Selected features	R^2	Adj-R^2
Traditional metrics	CountDeclMethodAll, CountClassCoupled, CountDeclClassVariable, RatioCommentToCode, CountLineBlank, CountLineCodeDecl	0.8784	0.7973
Traditional and proposed metrics	CountClassDerived, UniqueLibraryMethodCalls, CountSynchronizedBlocks, CountClassCoupled, CountTryBlocks, CountDeclClassVariable	0.8857	0.8095

Dai risultati riportati si può notare che:

- nella maggior parte dei casi, i valori relativi alle classi "0 *Mandelbug*" e "> 1 *Mandelbug*" variano tra 0,6 e 0,9. Ciò indica che i classificatori hanno buone capacità nell'identificare i moduli che sono *Mandelbug-prone*;
- i valori relativi alla classe "1 *Mandelbug*" sono molto più bassi, ciò indica una scarsa efficacia dei classificatori nell'identificare i moduli appartenenti a tale classe. Tuttavia, questi risultati vengono rilevati perché probabilmente la classe "1 *Mandelbug*" esibisce *feature* simili a quelle della classe "> 1 *Mandelbug*".

Inoltre, allo scopo di valutare se l'utilizzo di metriche tradizionali, delle nuove metriche e della *feature-selection* potesse realmente portare benefici nella *fault-prediction*, è stato effettuato un confronto tra le medie pesate della F-*measure*.

Tabella 12 Cross-validation relativo alla classificazione con 3 classi

Target class: number of Mandelbugs													
Classifier		0 Mandelbugs			1 Mandelbugs			> Mandelbugs			Weighted average		
		Pr	Re	F	Pr	Re	F	Pr	Re	F	Pr	Re	F
All metrics	Dec. Trees	0.772	0.637	0.689	0.283	0.333	0.280	0.779	0.750	0.746	0.682	0.608	0.627
	Mult. Logistic Reg	0.784	0.533	0.623	0.128	0.222	0.159	0.602	0.750	0.658	0.623	0.536	0.550
	Bayes Net.	0.702	0.726	0.707	0.109	0.156	0.126	0.950	0.800	0.858	0.653	0.638	0.636
	SVN	0.700	0.570	0.622	0.071	0.133	0.092	0.725	0.717	0.693	0.588	0.525	0.540
	Naive Bayes	0.612	0.904	0.729	0.000	0.000	0.000	0.722	0.467	0.557	0.525	0.625	0.549
Selected Metrics	Dec. Trees	0.683	0.644	0.660	0.072	0.089	0.079	0.771	0.967	0.857	0.590	0.621	0.600
	Mult. Logistic Reg	0.668	0.533	0.587	0.155	0.267	0.193	0.957	0.867	0.899	0.644	0.567	0.591
	Bayes Net.	0.711	0.970	0.820	0.022	0.022	0.022	1.000	0.833	0.876	0.654	0.758	0.684
	SVN	0.651	0.874	0.744	0.000	0.000	0.000	0.727	0.633	0.661	0.548	0.650	0.584
	Naive Bayes	0.682	0.793	0.731	0.000	0.000	0.000	0.822	0.650	0.709	0.589	0.608	0.588
All metrics, with novel metrics	Dec. Trees	0.740	0.615	0.662	0.231	0.267	0.222	0.762	0.717	0.721	0.650	0.575	0.595
	Mult. Logistic Reg	0.798	0.533	0.618	0.132	0.222	0.162	0.572	0.700	0.616	0.617	0.517	0.532
	Bayes Net.	0.682	0.719	0.695	0.096	0.111	0.099	0.906	0.800	0.842	0.628	0.625	0.620
	SVN	0.812	0.622	0.688	0.100	0.156	0.120	0.645	0.750	0.681	0.637	0.567	0.579
	Naive Bayes	0.632	0.978	0.767	0.000	0.000	0.000	0.867	0.467	0.603	0.572	0.667	0.582
Selected Metrics, with novel metrics	Dec. Trees	0.710	0.615	0.655	0.243	0.267	0.236	0.767	0.883	0.805	0.637	0.617	0.614
	Mult. Logistic Reg	0.724	0.548	0.616	0.165	0.333	0.216	0.879	0.717	0.775	0.658	0.550	0.581
	Bayes Net.	0.665	0.859	0.747	0.017	0.022	0.019	0.973	0.700	0.806	0.620	0.663	0.625
	SVN	0.778	0.993	0.871	0.000	0.000	0.000	0.793	0.867	0.819	0.636	0.775	0.695
	Naive Bayes	0.804	0.807	0.801	0.267	0.267	0.261	0.817	0.800	0.796	0.707	0.704	0.698

Per valutare se la bontà del miglior classificatore sia statisticamente significativa rispetto agli altri classificatori è stato eseguito un test d'ipotesi, adottando la procedura non parametrica "*Wilcoxon signed-rank*" al 90% di confidenza.
Dunque, nell'ultima colonna della Tabella 12, sono stati evidenziati in grassetto i risultati significativamente migliori (secondo l'esito del test d'ipotesi).
Un importante risultato è dato dal valore di F, che risulta maggiore quando si effettua la *feature-selection* e, in particolare, quando si adoperano:

- le Reti di Bayes e le metriche tradizionali;
- SVM e *Naive Bayes* sia con le metriche tradizionali che con quelle proposte.

Si noti, infine, che i migliori risultati sono stati ottenuti quando sono state adoperate le nuove metriche ed è stata effettuata la *feature selection*. Ciò, in accordo a quanto si voleva dimostrare, evidenzia che la presenza di informazioni sui costrutti come quelli di programmazione, di *exception handling* può fornire suggerimenti utili sul verificarsi di *Mandelbug*.
Analogamente al caso appena illustrato, in Tabella 13 e in Tabella 14 sono illustrati i risultati della *cross-validation* effettuata nel caso in cui le classi *target* siano due, cioè: "*Mandelbug-prone*" e "*Mandelbug-free*". Si può notare che nella maggior parte dei casi le misure relative a tali classi variano tra 0,6 e 1.

Tabella 13 Feature Selection con Classificazione a due classi

Metrics	Selected features	R2	Adj-R2
Traditional metrics	CountInput,N2,Vol,AvgEssential	0.8536	0.8004
Traditional and proposed metrics	CountInput, Essential, N2, Count-WaitNotify, Vol, CountClassBase, ImportIO	0.9843	0.9705

La scelta di utilizzare solo due classi *target* ha portato ad un miglioramento delle prestazioni di classificazione e a previsioni più affidabili, rispetto al caso precedente. Il miglior risultato si è ottenuto quando sono stati adottati SVM, le nuove metriche ed è stata operata la *feature-selection*. Infatti, i valori della F-*measure* media (pari a 0.941) e le delle metriche di *performance* (sempre maggiori di 0.9) hanno decretato questo tipo di previsione idonea per risolvere problemi relativi a scenari reali.

Tabella 14 Cross-validation relativa alla classificazione in 2 classi

Target class: number of Mandelbugs										
	Classifier				Mandelbug-prone			Weighted average		
					Pr	***Re***	***F***	***Pr***	***Re***	***F***
All metrics	Dec. Trees	0.698	0.652	0.665	0.589	0.629	0.598	0.651	0.642	**0.636**
	Mult. Logistic Reg	0.742	0.607	0.664	0.592	0.724	0.648	0.676	0.658	**0.657**
	Bayes Net.	0.701	0.748	0.706	0.676	0.571	0.593	0.690	0.671	**0.656**
	SVN	0.724	0.630	0.670	0.601	0.695	0.642	0.670	0.658	**0.658**
	Naive Bayes	0.654	0.659	0.650	0.548	0.533	0.533	0.608	0.604	0.599
Selected Metrics	Dec. Trees	0.882	0.770	0.819	0.754	0.867	0.804	0.826	0.813	**0.812**
	Mult. Logistic Reg	0.864	0.800	0.828	0.774	0.838	0.802	0.825	0.817	**0.817**
	Bayes Net.	0.833	0.807	0.815	0.776	0.790	0.778	0.808	0.800	**0.799**
	SVN	0.751	0.867	0.803	0.798	0.629	0.697	0.772	0.763	0.757
	Naive Bayes	0.810	0.815	0.808	0.763	0.743	0.746	0.789	0.783	**0.781**
All metrics, with novel metrics	Dec. Trees	0.680	0.622	0.643	0.549	0.600	0.565	0.622	0.613	0.609
	Mult. Logistic Reg	0.789	0.622	0.685	0.627	0.771	0.681	0.718	0.688	0.683
	Bayes Net.	0.633	0.622	0.622	0.549	0.543	0.537	0.596	0.588	0.585
	SVN	0.809	0.711	0.751	0.694	0.781	0.727	0.759	0.742	**0.741**
	Naive Bayes	0.702	0.770	0.730	0.672	0.571	0.609	0.689	0.683	0.677
Selected Metrics, with novel metrics	Dec. Trees	0.802	0.711	0.747	0.672	0.752	0.701	0.745	0.729	0.727
	Mult. Logistic Reg	0.956	0.926	0.938	0.916	0.943	0.927	0.938	0.933	**0.933**
	Bayes Net.	0.762	0.807	0.775	0.738	0.648	0.671	0.751	0.738	0.729
	SVN	0.976	0.919	0.943	0.917	0.971	0.940	0.950	0.942	**0.941**
	Naive Bayes	0.859	0.911	0.875	0.883	0.781	0.817	0.870	0.854	0.850

EFFICACIA DELLA FAULT-PREDICTION REGRESSION-BASED

Una seconda parte dello studio è stata dedicata a valutare l'efficacia della *fault prediction* quando si utilizza un modello di regressione.

L'obiettivo della sperimentazione è verificare l'accuratezza del *ranking* dei moduli, effettuata in base al valore stimato dei relativi *Mandelbug*; a tale scopo è stato utilizzando un modello di regressione.

In particolare, la precisione del *ranking* è stata valutata eseguendo più volte la *cross-validation* per l'addestramento dei modelli. Inoltre, il *ranking* predetto è stato confrontato con la classifica reale dei moduli, utilizzando il coefficiente di correlazione di *Pearson*.

Poiché, a partire dal *ranking*, un *tester* potrebbe decidere di concentrare gran parte delle risorse di *test* sui moduli che si trovano più in alto nella classifica fornita dal modello, nello studio è stata valutata anche l'accuratezza della *prediction* sui i moduli con maggiore *Mandelbug-prone*. Pertanto, sono stati valutati:

1) "***TOP-20%***", cioè il rapporto tra il numero di *Mandelbug* dei TOP- 20% "predetti" e il numero totale di *Mandelbug* del *test set*. Questo rappresenta la percentuale prevista di *Mandelbug* che è possibile rilevare quando ci si concentra sui moduli TOP-20% previsti, rispetto al *set* di *Mandelbug* presenti nel sistema in esame;
2) "***Normalized TOP-20%***", cioè il rapporto tra il numero di *Mandelbug* localizzati nei TOP-20% "predetti" ed il numero di *Mandelbug* nei TOP-20% "effettivi". Questo mette a confronto il numero di *Mandelbug* rilevabili focalizzandosi sui TOP-20% "predetti", con il numero di *Mandelbug* che potrebbero essere rilevati nel caso di previsione perfetta.

I risultati della *cross-validation* per i modelli di regressione sono mostrati in Tabella 15, in cui si può notare che i risultati migliori (secondo il test di *Wilcoxon signed-rank*) sono ancora una volta ottenuti quando si adoperano le metriche proposte e la *feature-selection*.

In particolare, quando viene adottato il modello di regressione lineare o SVR:

(i) Il valore atteso del coefficiente di correlazione del *ranking* previsto è superiore a 0,7;
(ii) **Quando ci si focalizza sui TOP-20% previsti, può essere rilevato circa il 60% dell'intero set di** *Mandelbug*;
(iii) Il modello di regressione identifica in media l'83% dei *Mandelbug* che potrebbero essere identificati in caso di perfetta previsione (come indicato dalla misura "Normalized TOP 20%").

Sulla scorta di questi risultati, si è potuto stabilire che la *fault prediction* può contribuire in modo significativo ad identificare il *set* di moduli in cui si trova la maggior parte di *Mandelbug*, migliorando così la pianificazione delle attività di V&V.

La sperimentazione ha evidenziato una presenza elevata di Mandelbug nel sistema complessivo (pari al 14,56%). Queste evidenze possono essere sfruttate per prendere delle decisioni in merito alle tecniche di *testing* più adeguate al caso in esame, oppure a favore di una progettazione *fault tolerant* spinta. Un importante elemento riguarda l'introduzione di nuove metriche, le quali, abbinate a quelle tradizionali, hanno dimostrato di migliorare notevolmente l'accuratezza della *fault prediction*.

Tabella 15 Cross validation in caso di regressione

	Regressor	Pearson	TOP 20%	Norm. coef. TOP 20%
All metrics	Linear Regr	0.241	**31%**	**46%**
	Regr. Trees	0.115	12%	19%
	SVR	0.352	**34%**	**50%**
Selected metrics	Linear Regr	0.516	38%	51%
	Regr. Trees	0.343	23%	33%
	SVR	0.617	**50%**	**74%**
All metrics with novel metrics	Linear Regr	0.227	**30%**	**45%**
	Regr. Trees	0.115	12%	19%
	SVR	0.326	**32%**	**50%**
Selected metrics with novel metrics	Linear Regr	0.770	**59%**	**83%**
	Regr. Trees	0.230	14%	22%
	SVR	0.707	**61%**	**83%**

6 OTTIMIZZARE LE RISORSE PER LA "*CODE SANITIZATION*"

Un'ulteriore esigenza legata all'utilizzo efficiente delle tecniche di V&V è legata alla analisi statica automatica (ASA). Come menzionato, gli strumenti di ASA sono di estrema utilità nella rilevazione di anomalie e di errori introdotti in fase di programmazione; di conseguenza, il loro utilizzo può apportare certamente un vantaggio in termini di qualità del codice prodotto.

Tuttavia, il problema principale delle tecniche di analisi è l'elevato numero di falsi positivi, ossia segnalazioni su difetti che il programma in realtà non contiene. Un caso estremo è riportato in [38], in cui gli autori parlano di oltre il 96% dei problemi di codifica segnalati, ma non relativi ad alcun difetto. Spesso, per superare questo problema, è possibile personalizzare gli strumenti, filtrando regole non pertinenti, in modo da ridurre, almeno in parte, il numero di falsi positivi.

La presenza di falsi positivi rende difficile riconoscere le violazioni delle regole più importanti da correggere, nonché i componenti più critici su cui focalizzare l'attenzione. Tuttavia, tale esigenza è particolarmente sentita nel contesto di SELEX ES, dove i prodotti *software* sono di grandi dimensioni, coprendo centinaia di componenti e milioni di linee di codice, ed in cui la corretta ripartizione degli sforzi per la correzioni delle violazioni ("*code sanitization*") può avere un impatto significativo sia in termini di aumento della qualità, sia in termini di riduzione dei costi.

Di seguito viene riportato un approccio, definito anch'esso nell'ambito del progetto, per l'allocazione ottimale delle risorse da dedicare al miglioramento del codice, basato sulla rimozione di violazioni di regole riportate negli strumenti di ASA.

L'idea di fondo è sfruttare i risultati dell'analisi statica, al fine di suggerire sia quanto sforzo dedicare al miglioramento della qualità del codice di un CSCI, sia quali attività svolgere - cioè, quali regole, tra quelle violate, valga la pena correggere.

L'analisi statica automatica è condotta tramite il *tool* Parasoft©, che analizza il codice rispetto a centinaia di regole. L'obiettivo è concentrarsi sui CSCI e sulle attività di rimozione delle violazioni di regole che hanno maggiore probabilità di ridurre il costo, nel rispetto di vincoli definiti dall'utente (ad esempio, il numero medio di violazioni nei CSCI deve essere inferiore a un valore critico).

FORMULAZIONE DEL MODELLO

I modelli sono parametrizzati e applicati separatamente alle prime 15 regole di tipo "bug detective" (BD) e di tipo "coding rules" **(CR**), che sono le due macro-classi di regole distinte dal *tool* di ASA impiegato – quindi si avrà un modello per le BD e uno per le CR. Denotiamo con v_i il numero totale di violazioni dell'i-esimo tipo, e con $v_{i,j}$ il numero di violazioni di tipo i scoperte nel CSCI j. Una violazione, se non rimossa, può portare a un difetto *software* in una fase successiva; d'altro canto, la sua rimozione richiede uno sforzo. Tale sforzo è, ovviamente, più basso dello sforzo per rimuovere un difetto. Assumiamo ogni tipo di violazione caratterizzato da una coppia < RE_i, w_i>, dove:

- RE_i (*Removal Effort*, sforzo di rimozione) è *l'effort* per unità necessario a rimuovere una violazione del tipo i-esimo;
- w_i è un fattore nell'intervallo [0,1] che pesa il tipo di violazione i rispetto all'impatto atteso sulla qualità del sistema che avrebbe una mancata rimozione della violazione.

I valori di RE_i sono definiti a partire dai dati storici sulla rimozione delle violazioni di tipo i osservate in passato. I pesi sono assegnati in base al giudizio di esperti (ingegneri SELEX ES), che tipicamente assegnano maggiore importanza alle violazioni che considerano più critiche (giudizio basato anche sui suggerimenti del forniti dal *tool* riguardo alla severità di ogni tipo di regola ed alle caratteristiche influenzate - come affidabilità, manutenibilità, sicurezza). Denotiamo con E_d lo sforzo per unità necessario a rimuovere un difetto nella fase di *field integration testing*, in cui i difetti *pre-release* vengono rimossi. Similmente ad RE_i, una stima di questo valore si ottiene dai dati storici sulle ore-uomo dedicate alle attività di rimozione dei difetti; chiaramente, $E_d \gg RE_i$. Abbiamo:

$$NRE_i = w_i * E_d$$

dove NRE_i è il *Non-Removal Effort* (sforzo di mancata rimozione), che rappresenta l'*effort* atteso per unità da profondere nel caso in cui la violazione di tipo i non venga rimossa. Questo valore è dato dallo sforzo unitario per la rimozione di un difetto moltiplicato per l'impatto della violazione i. Ovviamente, sarebbe molto costoso eliminare tutte le violazioni di tutti i tipi da tutti i CSCI. Il problema, quindi, è trovare il *trade-off* ottimale tra il numero e il tipo di violazioni da rimuovere (ovvero, determinare il costo della rimozione di violazioni) e il rischio connesso al non rimuoverle. Nel seguito, si denota con $x_{i,j}$ il numero di violazioni di tipo i che si decide di eliminare dal CSCI j; quindi, $x_{i,j}$ sono le variabili decisionali. La rimozione di violazioni deve essere effettuata entro vincoli di *budget* massimo e di qualità minima. A seconda di come sono espressi i vincoli di qualità, vengono definite tre diverse varianti del modello:

- *Target Average* (T-A): in questa variante, gli ingegneri vogliono che le violazioni nei CSCI siano al di sotto di uno specifico target, $\bar{v}_{max}$.
- *Target Average and Standard Deviation* (T-A & T-STD): in questa variante c'è una scelta più restrittiva della precedente, cioè che la deviazione standard sia inferiore a un *target* specificato dall'utente (std_{max}), oltre alla media.
- *Target Number* (T-N): gli ingegneri richiedono che un dato numero di CSCI, indicato con CN, presenti meno violazioni di un v_{max} definito dall'utente.

Per tutte le varianti, l'obiettivo è lo stesso: ridurre il costo totale, indicato con C, che è la somma dei costi su tutti i CSCI (C_j). Considerando le definizioni date, il costo su un singolo CSCI è dato da:

$$C_j = \sum_{i=1}^{m} x_{i,j} * RE_i + \sum_{i=1}^{m} [(v_{i,j} - x_{i,j}) * NRE_i]$$

Esso rappresenta il costo, per un CSCI j, della rimozione di $\sum_i x_{i,j}$ violazioni più il costo atteso della loro mancata rimozione. Segue il modello di ottimizzazione:

$$minimizza\ C = \sum_{j=1}^{n} E_j = \sum_{j=1}^{n}[\sum_{i=1}^{n}(x_{i,j} * RE_i) + \sum_{i=1}^{m}[(v_{i,j} - x_{i,j}) * (w_{i,j} * E_d)]]$$

$$\text{s. a} \quad 0 \leq x_{i,j} \leq v_{i,j}$$

e uno dei seguenti vincoli a seconda della variante:

$$\frac{1}{N}\sum_{j=1}^{n}\sum_{i=1}^{m}(v_{i,j} - x_{i,j}) \leq \bar{v}_{max} \qquad \text{(Modello T – A)}$$

$$\frac{1}{N}\sum_{j=1}^{n}\sum_{i=1}^{N}(v_{i,j} - x_{i,j}) \leq \bar{v}_{max} \qquad \text{(Modello T – A\&T – STD)}$$

$$STD\left[\sum_{i=1}^{m}(v_{i,j} - x_{i,j})\right] \leq std_{max}$$

$$\sum_{j=1}^{n} q_j \geq CN \qquad \text{(Modello T – N)}$$

Dove $q_j = \begin{cases} 1\ se\ \sum_{i=1}^{m} v_{i,j} - x_{i,j} > v_{min} \\ 0\ altrimenti \end{cases}$

Nella formulazione del modello, $j = 1 \ldots n-1$, n sono i CSCI, $x_{i,j}$ le variabili di decisione e STD denota la deviazione standard $STD[X] = \sqrt{\frac{\sum_i (x_i - \overline{x})^2}{n}}$.

APPLICAZIONE DEI MODELLI

L'output del modello suggerisce quante violazioni di ogni tipo dovrebbero essere rimosse da ogni CSCI al fine di raggiungere l'obiettivo desiderato con il minimo *effort*. Sono stati messi a confronto quattro scenari differenti su un insieme sperimentale di 156 CSCI appartenenti al dominio dell'*air traffic control*: le tre varianti tra di loro e con una strategia *Random* (R). Quest'ultima riflette la pratica corrente in cui ogni CSCI *manager* decide indipendentemente dagli altri quante violazioni rimuovere dal proprio CSCI, scegliendo arbitrariamente il tipo di violazione. Inoltre, allo scopo di avere un paragone equo con le altre strategie, è stato deciso di forzare una strategia *random* a consumare tutto il *budget* disponibile.

Dalla sperimentazione eseguita si è riscontrato che la soluzione calcolata dai modelli proposti è esatta; mentre, la soluzione fornita dall'allocazione *random* può essere diversa ad ogni esecuzione. Pertanto, è stato deciso di ripetere l'esecuzione della strategia *random* per 50 volte, in modo da ottenere risultati statisticamente significativi, poi è stata considerata la soluzione media in termini di costo totale.

PAREMETRIZZAZIONE DEL MODELLO

I costi di rimozione per tipo di violazione, RE_i, sono, nel nostro caso, stimati tramite la media dell'effort speso nel passato per rimuovere violazioni di ciascun tipo, approssimata dagli ingegneri SELEX. Per le regole BD, RE varia in media tra 1 e 5 minuti per violazione, a seconda del tipo di violazione;mentre, nel caso di regole CR, per le quali si evidenzia il supporto di *tool* per rimozione automatica, il tempo è inferiore al minuto. Inoltre, in questo esperimento, gli ingegneri SELEX hanno considerato *l'effort* di rimozione di un difetto, E_d, pari a 10 volte lo sforzo medio per rimuovere la violazione di una regola. I pesi sono stati assegnati come già descritto sopra, ovvero, tramite scale di severità. I parametri *RE* ed *NRE* per le regole BD e CR sono riportati in Tabella 16.

Tabella 16: Valori RE ed NRE (minuti)

BD Rules	RE	NRE	CR Rules	RE2	NRE3
1	5	22.6	1	0.1	0.01
2	4	18.8	2	0.5	1.0
3	3	21.6	3	0.1	1.25
4	3	13.6	4	0.05	1.42
5	4	8.8	5	0.1	1.53
6	4	8.8	6	0.2	0.88
7	2	16.8	7	0.1	1.12
8	5	18.4	8	0.3	0.77
9	2	28.0	9	0.2	0.22
10	3	25.8	10	0.1	0.33
11	1	5.2	11	0.1	0.33
12	1	7.6	12	0.05	0.55
13	1	3.2	13	0.1	0.44
14	2	11.6	14	0.1	0.66
15	2	8.8	15	0.2	1.25

Il *budget* per la rimozione delle violazioni è incluso nella funzione obiettivo come $\sum_j \sum_i x_{i,j} * RE_j$; è stato imposto che esso non sia superiore ad 800 ore-uomo per trattamento delle regole BD ed 800 per CR (ciò vuol dire che le soluzioni che non soddisfano i vincoli entro 800 ore-uomo vengono scartate; pertanto, queste necessitano di vincoli più rilassati sulla qualità raggiungibile, o di più ore-uomo). Gli ingegneri SELEX hanno imposto, in questo esperimento, i seguenti obiettivi di qualità per regole BD, per ognuno dei tre modelli: i) un numero di violazioni ridotto almeno del 40% rispetto alla media corrente; ii) una deviazione standard std_{max} ridotta di almeno il 75% rispetto a quella corrente, insieme a una media ridotta di almeno il 30%; iii) almeno il 70% dei CSCI con un numero di violazioni inferiore al 50% della media corrente. Per

quanto riguarda le regole CR: i) un numero medio di violazioni ridotto di almeno il 25% rispetto a quello corrente; ii) una deviazione standard ridotta di almeno il 30% e una media ridotta di almeno il 20%; iii) almeno il 70% dei CSCI con meno del 25% della media corrente.

METRICHE DI VALUTAZIONE

Dato lo stesso budget, il paragone è stato effettuato sulle seguenti metriche relative al caso random: il guadagno percentuale sul numero di violazioni di cui è stata suggerita la rimozione; la riduzione percentuale del costo totale stimato (TOT_{cost}, la somma del costo di rimozioni e non-rimozioni come descritta dalla funzione obiettivo); la riduzione percentuale del numero medio di violazioni tra i CSCI; il numero di CSCI con $\sum_i v_{i,j} < v_{max}$. Si noti che il costo di non-rimozione può essere considerato come misura di rischio: infatti, dal momento che il costo della rimozione di una violazione è dato come input (il *budget*, 800 ore-uomo, ed è sempre pienamente sfruttato dall'algoritmo), il costo rimanente è solo quello relativo alla mancata rimozione, che indica, indirettamente, il rischio connesso ad avere queste rimozioni residue. Un costo più alto indica che le violazioni residue hanno un impatto atteso maggiore.

RISULTATI

Le Tabella 17 e Tabella 18 riportano i risultati dell'esperimento in termini di guadagno o riduzione rispetto a media e deviazione standard iniziali, indicando in quale misura gli obiettivi sono stati raggiunti.

Tabella 17: Risultati dell'allocazione per le regole BD comparata con la soluzione random

Modello	% Guadagno su # Totale di Violazioni da Rimuovere	% di riduzione di TOT_{Cost}	% Riduzione del # Medio di Violazioni	% Riduzione STD del # di Violazioni	Numero (e %) di CSCI con $v < v_{max}$
T-A	**+16.12%**	**-32.95%**	**-14.94%** [-55.83%]	**-33.15%** [-57.68%]	**122 (78.20% dei CSCI)**
T-A & T-STD	**+16.14%**	**-29.13%**	**-14.96%** [-55.84%]	**-59.61%** [-75.43%]	**110 (70.51% dei CSCI)**
T-N	**+16.82%**	**-25.12%**	**-15.6%** [-56.17%]	**-57.65%** [-73.19%]	**110 (70.51% dei CSCI)**

Tabella 18: Risultati dell'allocazione per le regole CR comparata con la soluzione random

Modello	% Guadagno su # Totale di Violazioni da Rimuovere	% di riduzione di TOT_{Cost}	% Riduzione del # Medio di Violazioni	% Riduzione STD del # di Violazioni	Numero (e %) di CSCI con $v < v_{max}$
T-A	**+78.47%**	**-42.21%**	**-15.99%** [-30.22%]	**-14.50%** [-28.83%]	**117 (75%)**
T-A & T-STD	**+22.09%**	**-2.84%**	**-4.50%** [-20.67%]	**-21.65%** [-34.79%]	**110 (70.51%)**
T-N	**+22.66%**	**-2.34%**	**-4.62%** [-20.77%]	**-22.89%** [-35.82%]	**110 (70.51%)**

Esse mostrano che:

- Il modello T-A raggiunge l'obiettivo sul numero medio di violazioni; la soluzione fornita può anche soddisfare l'obiettivo ultimo sul numero di CSCI con $v < v_{max}$, anche se non richiesto. Anche la deviazione standard è ridotta considerevolmente, del 57% per BD e del 28% per CR rispetto a quella originale, e del 33% e del 54% rispetto alla soluzione casuale. Il numero di violazioni di cui è stata suggerita la rimozione è del 16% più alto della soluzione casuale (nel caso BD), ed è notevole nel caso CR, in cui si raggiunge un +78%. Il costo totale è ridotto del 32% e del 42% nei due casi. Questa soluzione permette di rimuovere molte più violazioni degli altri casi e, per di più, quelle che hanno un maggiore impatto sul costo.
- il modello TA & T-STD raggiunge i suoi obiettivi su media e deviazione standard, peraltro soddisfacendo anche il *target* sul numero di CSCI con $v < v_{max}$. Il guadagno in termini di STD è pagato in termini del numero di violazioni rimosse e costo totale, dove i guadagni sono molto meno consistenti del modello T-A. Per raggiungere l'obiettivo sulla *STD*, il modello T-A & T-STD si focalizza maggiormente sui pochi CSCI con molte violazioni, invece che sulle regole più impattanti.
- il modello T-N raggiunge il suo obiettivo soddisfacendo anche il *target* sulla media (non quello sulla *STD*). I risultati sono molto simili al modello della deviazione standard. Anche in questo caso, una grande riduzione è dovuta alla riduzione della *STD*.

Il modello T-A può selezionare i tipi migliori di violazioni da rimuovere, in quanto il suo obiettivo è raggiunto presto; pertanto, l'*effort* residuo è dedicato ad abbassare il valore di *effort* di non-rimozione selezionando i tipi di violazione più impattanti. Ovviamente, la scelta del *target* influenza l'esito, pertanto, l'utente può rilassare alcuni vincoli sulle soluzioni T-A & T-STD e T-N per ridurre il costo totale atteso. Ciò significa favorire una rimozione delle violazioni che tenga conto del tipo, piuttosto che una mera riduzione del

numero di violazioni. Al contrario, quando gli obiettivi sono raggiunti con ampio margine (come nel caso T-A), il *tester* può decidere di spendere meno *effort* e, così, ricalcolare una soluzione con un *budget* più basso (ad esempio, 600 ore-uomo). Si possono anche impiegare modelli multi-obiettivo.

In ogni caso, i risultati mostrano che i modelli sono strumenti utili per il ragionamento quantitativo e il supporto alle decisioni nella fase di *code sanitization*.

Ciò rappresenta un'indicazione molto utile per l'allocazione efficace degli sforzi di correzione su un insieme più piccolo di CSCI particolarmente critici. Inoltre, i modelli di ottimizzazione sviluppati rappresentano uno strumento per il ragionamento quantitativo e per il supporto alle decisioni in questa fase. Per esempio, il *tester* può decidere di investire meglio il *budget* disponibile per una rimozione *type-aware* delle violazioni, invece di selezionarle in maniera casuale. Oppure, può decidere di ottenere la stessa "qualità" di una rimozione *random*, riducendo però il costo complessivo, per esempio calcolando una soluzione ottimizzata con un *budget* più basso.

7 CONCLUSIONI

Il presente documento ha descritto i principali metodi che saranno utilizzati nell'ambito del progetto per l'applicazione di attività di V&V nel *framework* SVEVIA. Sono state brevemente introdotte le principali tecniche ritenute rilevanti per il miglioramento della qualità dei sistemi *mission-critical* sviluppati da SELEX ES, seguite da tre metodi di pianificazione per il loro utilizzo.

Tali metodi non sono alternativi e si riferiscono a fasi ed attività diverse del processo di V&V. Ciascuno di essi necessità di un certo tipo di informazione, diversa nei tre casi, e fornisce risultati di tipo diverso, seppur tutti legati all'ottimizzazione del *trade-off* tra costo e qualità. Nella situazione ideale, l'utilizzo di tutte e tre i metodi consentirebbe ai *test manager* di avere una visione chiara di come e dove spendere le risorse, attingendo a fonti di informazione eterogenee e rappresentative di diverse sfaccettature della qualità dell'artefatto. Ad esempio, nel caso degli SRGM, si riflette la qualità del processo di test dei singoli CSCI, ed in base a questo si allocano le risorse restanti; nel caso dei modelli di *fault proneness* si riflette la qualità interna del codice – e quindi dell'attività di implementazione - come indicatore della criticità dei CSCI, allocando le risorse corrispondentemente; infine, nel caso della ASA, la qualità è ancora relativa all'implementazione, ma le risorse da allocare si riferiscono non più al test (o ad altre attività di V&V), bensì al *fixing* dei problemi riscontrati.

In questo documento, sono stati altresì presentati risultati sperimentali. La loro valenza, per il prosieguo del progetto, è principalmente in relazione al fatto che la sperimentazione è stata condotta sui (sotto)sistemi stessi di SELEX ES. Nell'ambito della definizione ed implementazione del *framework*, tali risultati saranno pertanto considerati per il raffinamento delle tecniche implementate.

APPENDICE

CENNI SU RELIABILITY E MISURE

L'affidabilità ("*reliability*") di un sistema è la misura del tempo in cui esso riesce a fornire un servizio corretto in modo continuativo.
Supposto funzionante al tempo $\mathbf{t_0}$, la distribuzione dell'affidabilità $\mathbf{R(t)}$ di un sistema è definita come la probabilità condizionale che esso funzioni correttamente nell'intervallo [t0,t].
In particolare, sotto le ipotesi:

a) il sistema sia costituito da **N** componenti identici, ciascuno dei quali messo in funzione al tempo t_0;
b) $\boldsymbol{N_f(t)}$ sia il numero di componenti guasti al tempo t;
c) $\boldsymbol{N_0(t)}$ sia il numero di componenti funzionanti al tempo t,

si avrà che:

$$\boldsymbol{N_f(t) + N_0(t) = N}$$

Inoltre, supponendo che un componente guasto rimanga in questo stato per un periodo di tempo indefinito, è possibile esprimere la distribuzione di affidabilità dei restanti componenti come:

$$\boldsymbol{R(t)} = \frac{N_0(t)}{N} = \frac{N_0(t)}{N_f(t) + N_0(t)}$$

In cui R(t) esprime la probabilità che un componente, funzionante al tempo $\mathbf{t_0}$, funzioni correttamente al tempo $\boldsymbol{t}$.
Oltre alla *reliability*, è possibile rappresentare la distribuzione del tempo di fallimento del sistema, cioè la "*unreliability*", $F(t)$, come:

$$\boldsymbol{F(t)} = \frac{N_f(t)}{N}$$

Si noti che $\boldsymbol{R(t) + F(t) = 1}$.

Come mostrato in Figura 7, l'affidabilità di un componente varia a seconda del periodo di vita dello stesso, mentre l'inaffidabilità è una funzione cumulativa di guasto, espressa rispetto al numero totale di componenti operativi.

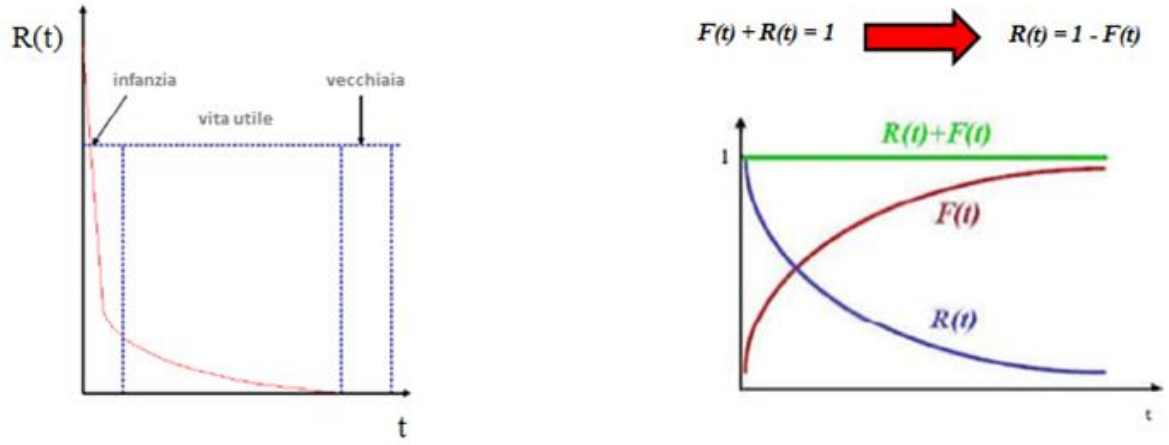
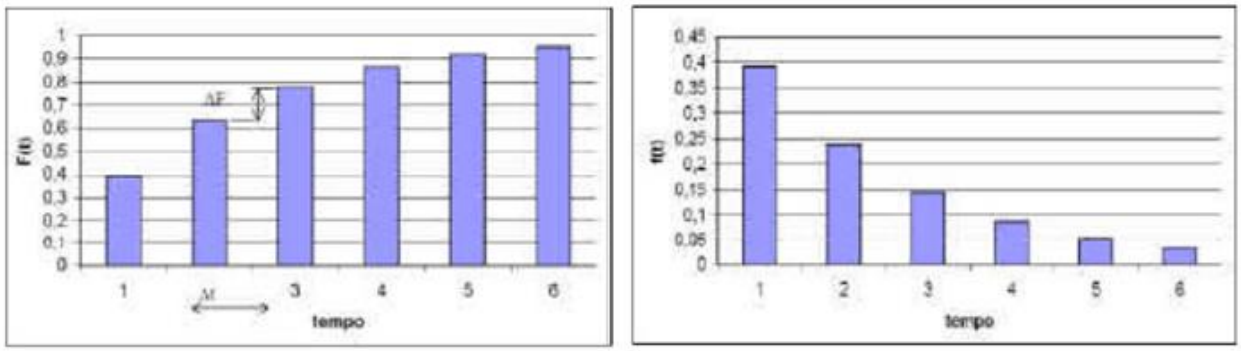


Figura 7 Affidabilità e Inaffidabilità

Un'importante misura (Figura 8), ottenuta osservando le variazioni di *F(t)* in intervalli discreti di ampiezza Δt, è la funzione densità di probabilità di guasto, *f(t)*, definita come:

$$\boldsymbol{f(t)} = \frac{\Delta F(t)}{\Delta t} = \frac{\Delta N_f(t)}{\Delta t} \cdot \frac{1}{N}$$



Probabilità di guasto *F(t)* **Densità di probabilità di guasto *f(t)***

Figura 8 Variazioni nel tempo di F(t) ed f(t)

Un'ulteriore parametro, accennato nel documento, è il tasso di guasto istantaneo λ(t) è la variazione relativa del numero di oggetti funzionanti (N_0), causata da un guasto, rispetto al tempo:

$$\boldsymbol{\lambda(t)} = f(t) \cdot \frac{N}{N_0(t)} = \left(\frac{dF(t)}{dt}\right) \cdot \frac{N}{N_0(t)} = \frac{f(t)}{R(t)}$$

Questo può essere calcolato sia sulla base di considerazioni di tipo statistico, sia sui parametri forniti dal costruttore, prestando però attenzione al periodo di vita del componente in esame
Una misura sintetica della affidabilità è il "*Mean Time To Failure*" (MTTF), che rappresenta il tempo medio che intercorre tra l'istante in cui il componente diventa operativo e l'istante in cui avviene un guasto:

$$\textbf{MTTF} = \int_0^{\infty} t \cdot f(t)dt$$

Si noti che, nel caso in cui il *failure rate* sia costante, $\textbf{MTTF} = \frac{1}{\lambda}$

Il principale parametro della disponibilità è "*Mean Time to Repair*" (MTTR), che esprime il tempo medio che intercorre tra l'istante in cui si verifica un guasto e l'istante di completamento della sua riparazione.

Dai due parametri sopra citati dipende un'importante misura sulla quale viene valutata la *reliability*: il "*Mean Time Between Failure*" (MTBF). Essa rappresenta il tempo medio che intercorre tra due fallimenti consecutivi rilevati nel sistema ed ha senso applicarla solo a componenti riparabili (Figura 9).

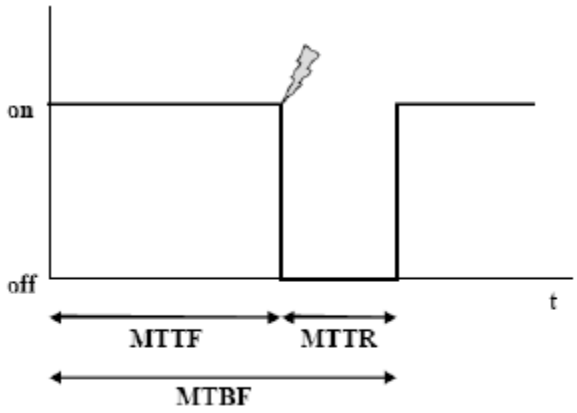


Figura 9 Relazione tra MTTF - MTTR - MTBF

RIFERIMENTI

N.	Riferimento / Descrizione
[1]	Avizienis A., Laprie J-C., Randell B., Landwehr C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: IEEE Transactions on Dependable and Secure Computing, Vol.1,2004, p.11-33
[2]	Optimal Testing Resource Allocation Models for Modular Software - Rani Rajan, IIT Kharagpur, Ravindra B. Misra, Ph.D., IIT Kharagpur
[3]	M Pezzè and M Young. Software Testing and Analysis: Process, Principles and Techniques. Wiley, 2008
[4]	David B.Brown, Saeed Maghsoodloo and William - H.Deason. "A Cost Model for Determining the Optimal Number of Software Test Cases", IEEE Transactions on Software Engineering, vol.15, no.2, Feb 1989
[5]	C. Catal, and B. Diri, A systematic Review of software fault prediction studies. Expert Systems with Applications, 36 (4), 2009
[6]	T. McCabe, A complexity measure, IEEE Transactions on Software Engineering, 2(4), 308-320, 1976
[7]	M. Halstead, Elements of Software Science, Elsevier Science, 1977
[8]	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.
[9]	F.B. Abreu, and R. Carapuca, Object---oriented software engineering: Measuring and controlling the development process. Fourth international conference on software quality, 1994
[10]	J. Bansiya, and C. Davis, A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, 28 (1), 4-17, 2002.
[11]	M. Lorenz, and J. Kidd, Object-oriented software metrics: A practical guide, Prentice Hall Inc, 1994
[12]	J. Gray, 1986. "Why do computers stop and what can be done about it?"
[13]	N. Nagappan, T. Ball, Zeller, Mining Metrics To Predict Component Failures, In The Proc. Of the 28th international conference on Software engineering (ICSE '06), 2006, 452–461.
[14]	*Isabelle Guyon, Andre´ Elisseeff, 2003. "An Introduction to Variable and Feature Selection"*
[15]	John D. Musa Software Reliability Engineering and Testing Courses]
[16]	Amrit L. Goel and Kazu Okumoto. Time-dependent error-detection rate model for

	software reliability and other performance measures. *Reliability, IEEE Transactions on*, R-28(3):206 -211, aug. 1979
[17]	R.E. Mullen. The lognormal distribution of software failure rates: application to software reliability growth modeling. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 134 –142, nov 1998
[18]	S.S. Gokhale and K.S. Trivedi. Log-logistic software reliability growth model. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 34 –41, nov 1998
[19]	Shigeru Yamada, Mitsuru Ohba, and Shunji Osaki. S-shaped reliability growth modeling for software error detection. *Reliability, IEEE Transactions on*, R-32(5):475–484, dec. 1983
[20]	H. Okamura, T. Dohi, and S. Osaki. Em algorithms for logistic software reliability models. In *Proc. 22nd IASTED International Conference on Software Engineering*, pages 263–268, 2004.
[21]	Koji Ohishi, Hiroyuki Okamura, and Tadashi Dohi. Gompertz Software Reliability Model: Estimation algorithm and empirical validation. *J. Syst. Softw.*, 82(3):535–543 (March 2009)
[22]	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)
[23]	Richard Lai and Mohit Garg. A detailed study of NHPPh Software Reliability Models. *Journal of Software*, 7(6) (2012)
[24]	W. Everett, Software Component Reliability Analysis, In: Proc. of the Symp. Application specific Systems and Software Engineering Technology (ASSET '99), pp. 204– 211 (1999)
[25]	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)
[26]	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)
[27]	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)
[28]	Analysis of an Inflection S-shaped Software Reliability Model Considering Log-logistic Testing-Effort and Imperfect Debugging N. Ahmada, M. G. M. Khanb and L. S. Rafi
[29]	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)
[30]	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)
[31]	J. Musa, "The Operational Profile in Software Reliability Engineering: An Overview," Proc. 3rd IEEE Int. Symp. on Software Reliability Engineering (ISSRE'92), pp. 140-154 (1992)
[32]	A. L. Goel, Software reliability Models: Assumptions, limitations, and applicability, IEEE transactions on software engineering, SE-11, 1411-1423 (1985)
[33]	G. Carrozza, R. Pietrantuono, S. Russo, Dynamic test planning: a study in an industrial context G, International Journal on Software Tools for Technology Transfer, (May 2014)
[34]	D. Cotroneo, R. Pietrantuono, S. Russo (2013). Combining Operational and DebugTesting for Improving Reliability. IEEE TRANSACTIONS ON RELIABILITY, vol. 62, p. 408-423, ISSN: 0018-9529, doi: 10.1109/TR.2013.2257051.
[35]	Natella, R.; Cotroneo, D.; Duraes, J.A.; Madeira, H.S., On Fault Representativeness of Software Fault Injection, Software Engineering, IEEE Transactions on , vol. 39, no.1, pp. 80–96, Jan. 2013
[36]	J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, “On the value of static analysis for fault detection in software,” Software Engineering, IEEE Transactions on, vol. 32, no. 4, pp. 240– 253, April 2006.
[37]	T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, “Defect prediction from static code features: current results, limitations, new approaches,” Automated Software Engineering, vol. 17, no. 4, pp. 375–407, 2010.
[38]	F.Wedyan,D.Alrmuny,andJ.Bieman,“Theeffectivenessofautomated static analysis tools for fault detection and refactoring prediction,” in Software Testing Verification and Validation, 2009. ICST ’09. Interna- tional Conference on, April 2009, pp. 141–150.
[39]	Michael Grottke and Kishor S. Trivedi. 2007. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. Computer 40, 2 (February 2007), 107-109. DOI=10.1109/MC.2007.55