UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Ingegneria Informatica

Implementazione e sperimentazione di criteri di terminazione per processi di testing random

Anno Accademico 2015/2016

relatore
Ch.mo Prof. Porfirio Tramontana

correlatore
Ing. Nicola Amatucci

candidato
Andrea Messalino
matr. M63/000163

A tutti quelli che hanno creduto
e che crederanno nelle mie capacitá

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Cos'è il testing automatico

Com'è noto, il testing (o collaudo) del software è un'indagine attuata in ambito informatico per migliorare la qualità del software prodotto, rilevandone i difetti e contrastandone gli effetti indesiderati. Esso costituisce un passaggio obbligatorio nel processo di sviluppo del software, dal momento che si vuole essere il più possibile certi che il codice scritto non generi gravi fallimenti nel momento in cui venga eseguito. Per misurare adeguatamente la bontà del testing effettuato, uno dei criteri più utilizzati si basa sulla percentuale di linee di codice coperte dal processo collaudativo:[1] più se ne riescono a sollecitare, maggiore sarà l'affidabilità del test. Per conseguire tale scopo, si investono risorse ingenti in tal senso, ma sia per la loro limitatezza, sia per rendere meno ardua l'analisi complessiva (evitando compiti noiosi e ripetitivi), si cerca di automatizzare questo processo. Il testing automatico consiste proprio nell'impiego di speciali strumenti software creati apposita-

mente per collaudare i programmi che si vogliono controllare, senza ricorrere necessariamente a test di tipo manuale. Solitamente, infatti, le verifiche a mano dei prodotti software su predeterminate sequenze di istruzioni per scovare bug indesiderati o accertarsi che, a seguito di modifiche migliorative, le nuove versioni funzionino come dovrebbero, si rivelano estremamente tediose, nonché dispendiose in termini di tempo da riservare al testing; inoltre, con un test manuale non si è mai sicuri di aver coperto interamente il codice da testare, dato che non si possono conoscere a priori le sequenze da eseguire e/o i potenziali errori annidati all'interno.[2] Pertanto, automatizzare le procedure di testing del software è un metodo pratico per rendere più efficace e più efficiente l'attività di collaudo di qualsiasi applicativo: più efficace, perché consente di coprire capillarmente il codice sorgente, e più efficiente, perché si risparmia notevolmente tempo e fatica; in più, una volta implementato il metodo automatico più adatto alle circostanze, è possibile reiterarlo a piacere senza essere costretti a riprogettare da capo ogni sessione di test. Rimane, tuttavia, un problema di non poco conto: al di là della delegazione del testing a meccanismi di tipo automatico, è d'uopo stabilire un metodo affidabile per elencare gli eventi che s'intendono somministrare al software sotto esame, affinché si possa ottenere un collaudo dignitoso.

1.2 Cos'è il testing casuale

Per disbrigare brillantemente la determinazione delle sequenze di eventi da utilizzare nei test, anziché selezionarli a priori (rischiando, tra l'altro, di ometterne involontariamente qualcuno) si ricorre ad una tecnica concepi-

ta durante gli anni Settanta del XX secolo, ovvero il testing casuale. Esso consiste nella generazione automatica di input scelti in maniera aleatoria ed indipendente tra loro, i quali vengono poi immessi sull'interfaccia da testare per poterne registrare gli effetti ed il grado di copertura del codice sorgente, nonché rilevare errori e/o falle non previste durante lo sviluppo. Si tratta, dunque, di una tecnica poco dispendiosa e di facile utilizzo, dal momento che l'approccio con con cui viene collaudato un applicativo è totalmente demandato al sistema di elaborazione; su di essa sono state condotte diverse ricerche che ne hanno certificato un'efficacia migliore rispetto ad altri metodi di testing più tradizionali, e di cui sono state elaborate diverse varianti (che vedremo in dettaglio nel capitolo 2). Nonostante i vantaggi che offre, però, il testing casuale presenta anche qualche effetto collaterale: quello più rilevante è dato proprio dalla sua natura fortemente arbitraria, che non essendo predicibile potrebbe produrre sequenze simili tra loro per lunghi lassi di tempo, senza alcun miglioramento in termini di copertura del codice e/o di segnalazione di bug. Inoltre un'altra questione spinosa riguarda il criterio da adottare per stabilire quando un test casuale possa definirsi realmente concluso, dato che non si vogliono sciupare inutilmente le risorse a disposizione se in sostanza non si ottengono incrementi significativi a livello di collaudo (o, più banalmente, se i tempi di consegna del prodotto software sono particolarmente stringenti). In genere ci si ritrova a dover decidere su due potenziali alternative: o si interrompe brutalmente il test (ma in tal caso si rischia di non coprire adeguatamente il codice sotto esame) o lo si lascia proseguire finché non vengono raggiunti dei criteri prestabiliti (ma così facendo si potrebbe perdere troppo tempo ad attendere che tale obiettivo venga

effettivamente conseguito).[3] Dato che nessuna delle due ipotesi è perseguibile senza ripercussioni sulla bontà del processo di testing, nella pratica si adoperano dei criteri euristici che consentono di ottenere il giusto equilibrio tra la necessità di voler approfondire in dettaglio il codice sorgente e quella di contingentare i tempi di esecuzione dei vari test. Tra le possibili soluzioni adottabili per terminare anticipatamente i test vi sono l'interruzione dopo un tempo massimo di tipo arbitrario,[4][5] il raggiungimento di un valore fisso di copertura[6] e l'interruzione dopo il superamento di un valore di variazione massimo dall'ultimo incremento della percentuale di copertura.[7] In particolare, in ambiti in cui il testing casuale viene applicato su più sessioni di test eseguite in parallelo (ad esempio il cloud computing), sono state proposte ulteriori soluzioni che si basano sul cosiddetto "effetto saturazione", un fenomeno comune in tutti i metodi di testing[8] in cui la percentuale di copertura di un test, a mano a mano che essa progredisce, si assesta su un determinato valore che tende a rimanere costante.

Nel presente lavoro osserveremo come questi ultimi criteri di terminazione, congiuntamente ad altri presi in considerazione nel corso delle analisi effettuate, non solo individuino esattamente il punto di saturazione ma consentano anche di fermare con un buon grado di accettabilità i test compiuti su un campione di 12 applicazioni desktop con interfaccia Java. Prima, però, è bene illustrare in maniera più dettagliata lo stato dell'arte sulle tecniche più frequentemente impiegate nei testing casuali.

Capitolo 2

Stato dell'arte

2.1 Tecniche di testing casuale

Secondo quanto descritto nel 2015 da Mariani et al. nel corso dei loro studi sul testing automatico a scatola chiusa (*black-box automatic testing*),[9] il testing casuale è classificabile in quattro tipologie di tecniche: Pure Random Techniques (puramente casuali), Adaptive Random Techniques (di tipo adattativo), Guided Random Techniques (regolate da analisi preliminari) e Fuzz Techniques (con comportamenti imprevisti).

2.1.1 Pure Random Techniques

Le tecniche puramente casuali rappresentano il caso più semplice di generazione dei casi di test, poiché gli input vengono selezionati casualmente secondo una probabilità con distribuzione uniforme. Ciò comporta diversi vantaggi, tra cui bassi costi di implementazione, tempistica rapida ed equiprobabilità degli input estratti.[9] Uno dei strumenti che adoperano tale ti-

pologia sono i cosiddetti *monkey tools*, così denominati perché i loro input assomigliano al comportamento delle scimmie quando maneggiano un oggetto artificiale (su questo fenomeno è stato elaborato un teorema apposito, detto "delle scimmie infinite", il quale afferma che una o più scimmie in grado di battere i tasti di una macchina da scrivere per un tempo infinito "quasi sicuramente" riprodurranno le opere di un celebre scrittore, sia esso William Shakespeare o Isaac Asimov).[10] I *monkey tools* sono molto facili da realizzare e impiegati in molti contesti; tra i più noti si annovera l'*UI/Application Exerciser Monkey*,[1] un programma eseguibile su emulatori o dispositivi Android che crea input pseudo-casuali di eventi innescabili da un generico utente e che viene impiegato per il testing di applicazioni sviluppate per il sistema operativo mobile di Google.

2.1.2 Adaptive Random Techniques

Nelle tecniche adattative gli input vengono invece elaborati tenendo conto dei differenti pesi specifici tra le possibili sequenze generabili. Questa tipologia, analizzata per la prima volta da Chen et al. nel 2001,[11] consente di migliorare significativamente la ricerca di falle nascoste rispetto ai metodi puramente casuali, come dimostrato da diversi studi compiuti,[12][13] sebbene non manchino delle contro-analisi critiche che mettono in dubbio la sua reale efficacia su casi reali, come ad esempio quella condotta da Arcuri e Briand.[14]

[1] https://developer.android.com/studio/test/monkey.html

2.1.3 Guided Random Techniques

Nelle tecniche guidate, per la determinazione delle sequenze di input, si considerano anche le singole componenti del software da testare, ricavate tramite analisi statica oppure dinamica. Tra gli strumenti che adottano tale approccio, uno dei più popolari è Randoop (generatore di unità di test per applicazioni Java)[15], ma esistono tool con tecniche random guidate anche per applicazioni in mobilità, come ad esempio Android Ripper in ambiente Android.[16]

2.1.4 Fuzz Techniques

Nelle tecniche fuzz i test casuali vengono effettuati immettendo input non validi e/o non previsti e studiando il comportamento del sistema per scovare dei fallimenti che altrimenti non sarebbero rilevati con un insieme ordinario di input. Esempi recenti di prodotti software che prevedono l'impiego di tecniche fuzz sono SecFuzz (usato per la sperimentazione di protocolli di sicurezza)[17] e KEmuFuzzer (progettato per la sperimentazione di macchine virtuali).[18]

2.2 Valutazione delle prestazioni del testing casuale

In letteratura sono reperibili numerosi studi sulle prestazioni delle tecniche di testing casuale (tra cui quello descritto da Hutchins et al.[6]), svolti principalmente per saggiare l'affidabilità delle ipotesi proposte in ognuna di

esse e poterle comparare tra loro per stabilire su quale metrica (copertura del codice, individuazione di bug sconosciuti, ecc.) esse si rivelino più adatte. Se da un lato le ricerche effettuate hanno mostrato una certa correlazione tra il codice coperto e le falle scoperte (come osservato da Malaiya et al.[19]), di contro Wei et al. hanno confutato tale asserzione, ma limitatamente a programmi progettati e sviluppati in linguaggio Eiffel.[20] Inoltre sono stati elaborati modelli (sia teorici, sia empirici) per calcolare concretamente le prestazioni delle tecniche di testing casuali, come quello proposto da Arcuri et al., in cui, modellando l'analisi del software sotto esame secondo il cosiddetto "problema del raccoglitore di bigletti" (*Coupon collector's problem*)[21] anziché con probabilità uniforme (rivelatasi impraticabile dal punto di vista operativo), viene dimostrata la loro maggior efficacia nella copertura delle strutture di controllo e/o delle singole linee del codice sorgente rispetto al rilevamento di errori indesiderati; inoltre, viene fatta osservare l'esistenza di un valore minimo di test necessari per ottenere un buon grado di copertura, il raggiungimento di risultati migliori quando le sequenze da coprire sono equiprobabili e la scalabilità delle tecniche casuali rispetto a quelle partitive.

2.3 Criteri di terminazione del testing casuale

Come già accennato nel capitolo 1, di solito si tende a trovare un compromesso tra l'accuratezza delle prove effettuate e il tempo ad esse dedicato a causa della limitatezza delle risorse disponibili; per tale motivo sono sta-

ti concepiti alcuni criteri di terminazione per interrompere i test nel minor tempo possibile, ma conservando un livello accettabile delle analisi compiute. Un primo criterio abbastanza semplicistico consiste nel fissare un limite massimo di tempo,[22] di eventi[23] o di chiamate di funzioni[15] entro cui completare il testing casuale: se da un lato risulta la soluzione più diffusa, in quanto molto semplice da implementare, dall'altro non assicura per nulla che i risultati ottenuti siano affidabili, soprattutto se non si copre in maniera adeguata il codice sorgente a causa di fattori troppo stringenti. Di contro, l'uso di un criterio di terminazione basato sull'ottenimento di un determinato livello di copertura potrebbe comportare uno spreco di tempo indefinito nell'attesa che ciò accada (soprattutto in presenza di strutture di controllo difficilmente percorribili), pertanto viene adottato solo in casi particolari.[6] Un altro criterio abbastanza percorribile nella pratica prevede l'interruzione del test nel momento in cui viene superato un prestabilito numero di sessioni consecutive senza ulteriori miglioramenti nella copertura complessiva del codice.[7] Esistono, inoltre, criteri più complessi incentrati sui modelli statistici dell'incremento della copertura del codice (dove vengono impiegate diverse distribuzioni di probabilità: binomiale,[21] poissoniana,[24] bayesiana,[25] logaritmica,[26] ecc.) oppure su modelli di uso operazionale ricavati dalle catene di Markov[27] (che però si fondano sull'indipendenza statistica della copertura del codice, cosa che non può accadere in casi di test la cui metrica è fondata su una copertura strettamente legata alla sequenza di eventi che l'ha causata). In realtà, nel nostro lavoro di ricerca sui processi di testing casuale ci siamo concentrati in maniera approfondita su criteri di terminazione basati sul cosiddetto "effetto saturazione", ovvero sulla ridotta

capacità di scoperta di nuovi bug col passare del tempo di esecuzione di un test. Si tratta di un fenomeno documentato, tra gli altri, da Sherman[28] e che influisce sull'incremento della percentuale di copertura del codice da testare, il quale, dopo un determinato numero di eventi eseguiti (generalmente nell'ordine delle migliaia) diminuisce e/o si arresta su un determinato limite, detto punto di saturazione, in cui si satura completamente, poiché a mano a mano che si procede con la scelta casuale degli eventi aumenteranno le probabilità di ripescare e ripetere quelli già verificati, mentre diminuiranno le possibilità di coprire parti di codice ancora inesplorate; per tale ragione, insistere sul proseguimento dei test oltre il punto di saturazione non risulta affatto conveniente (specie se è stata adottata la copertura come metrica di riferimento), dato che non si otterrebbe alcun progresso sulla percentuale di copertura (se non dopo innumerevoli tentativi a vuoto e senza alcuna certezza che la copertura aumenti ulteriormente). A partire da tale punto sono stati stabiliti diversi criteri di terminazione dei test, che abbiamo messo a confronto e che illustreremo in dettaglio nel capitolo 3.

Capitolo 3

Criteri di terminazione

In questo capitolo verranno spiegate le considerazioni che ci hanno spinto ad adottare i criteri usati per le nostre ricerche, le finalità che ci siamo prefissati prima di intraprendere la fase sperimentale e con quale procedimento quest'ultima sia stata concepita.

3.1 Osservazioni sui criteri di terminazione

Come si è potuto intuire dal discorso introdotto nel capitolo 2, la scelta dei criteri non è affatto banale, poiché ognuna di esse può condurre a conclusioni totalmente differenti: se se ne considera uno troppo sbrigativo, i test verranno sì completati in tempi più brevi, ma potrebbero rilevare una copertura del codice piuttosto scarna, mentre un criterio troppo oneroso in termini temporali, viceversa, potrebbe registrare una copertura più capillare, ma con un maggior spreco di risorse; in casi estremi, si corre il rischio che il test non giunga mai a termine, costringendoci ad un intervento manuale per blocca-

re il processo, anche se un'evenienza del genere dipende dall'aleatorietà con cui vengono selezionate le sequenze di test. Infatti, dal momento che non si conoscono a priori quali siano le azioni intraprese dagli strumenti automatici impiegati per collaudare il software sotto esame, è impossibile predire con certezza come la copertura calcolata dal test evolva nel tempo; inoltre, se lo si prova a ripetere per più di una volta, la progressione con cui il software viene collaudato e coperto potrebbe cambiare sensibilmente proprio a causa del fatto che la serie di eventi non è mai la stessa, ma varia a seconda di quali vengono estratti passo dopo passo. Ciononostante, a prescindere da quali sequenze vengano prodotte dagli strumenti automatici, si può dimostrare facilmente che nel lungo periodo (ovvero dopo una massiccia dose di eventi somministrati) i test casuali mostrano i sintomi del già accennato "effetto saturazione", dato che all'aumentare delle linee di codice coperte ce ne saranno sempre alcune di esse che rimarranno irraggiungibili per i motivi più disparati: mancata estrazione dell'evento che "scatena" proprio quel ramo di codice, complessità di ramificazione delle strutture di controllo (ovvero "if-then-else", "switch-case", ecc.), precondizioni sussistenti sull'applicazione, e via discorrendo. Tale fenomeno costituisce un valido indicatore circa la bontà dei test compiuti: secondo le conclusioni tratte da Sherman[28], i test basati sull'osservazione della saturazione offrono diversi vantaggi, tra cui evitare l'uso di metriche di copertura che portano ad interruzioni premature, monitorare progressivamente i risultati in base al livello massimo ottenuto e trovare un giusto equilibrio tra risorse impiegate e codice coperto. Ad ogni modo, una volta fissato un criterio di terminazione, i test verranno interrotti nell'istante in cui esso avrà ottemperato i propri requisiti.

Osserviamo meglio con un esempio pratico quanto illustrato finora. In figura 3.1 è stato rappresentato in scala logaritmica l'andamento grafico di 8 sessioni di test (ognuna composta da 10000 eventi) su un'applicazione Java, in cui abbiamo marcato alcuni potenziali punti di arresto. Come si può

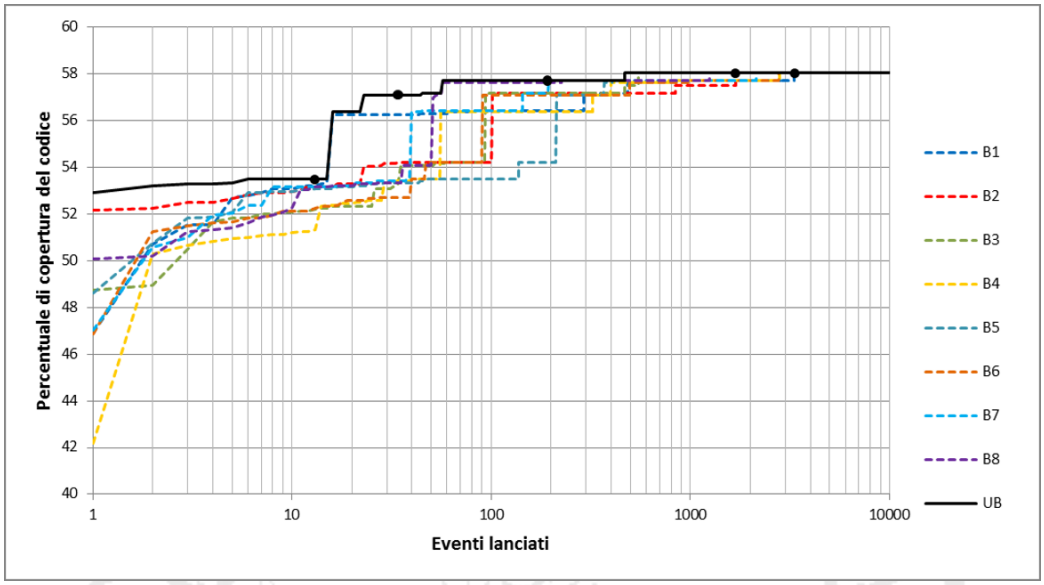


Figura 3.1: Grafico della copertura del codice calcolata su 8 sessioni di test

ben apprezzare, esse convergono prima o poi verso un determinato valore massimo di copertura (che in quest'esempio si attesta attorno al 58% del codice sorgente), ma con progressioni iniziali totalmente diverse l'una dall'altra: ciò è dovuto proprio al fatto che le sequenze impiegate per ciascuna sessione sono indipendenti tra loro per via della selezione casuale degli eventi da testare. Inoltre, possiamo constatare che, se bloccassimo il processo di testing prima che le sessioni convergano nel punto di saturazione, avremmo risultati differenti a seconda di quale sessione venga considerata; di contro, se proseguissimo i test oltre tale punto, si perderebbe ulteriore tempo nel ricercare un ipotetico valore di copertura superiore a quello registrato ma

difficilmente ottenibile, per cui sarebbe più ragionevole fermarsi in maniera opportuna prima che il processo continui senza miglioramenti. Per riscontrare meglio ciò, comunque, è sicuramente preferibile ipotizzare dei criteri di terminazione basati su confronti tra le coperture rilevate su più sessioni, così da individuare automaticamente un punto di interruzione nell'intervallo in cui si manifesta l'effetto saturazione. In una ricerca condotta nel 2015 da Amalfitano et al.[3] sono stati formulati 2 criteri per i quali i test venivano arrestati in altrettante occasioni: quando una delle sessioni eguagliava la copertura massima globale oppure quando a compiere il medesimo obiettivo erano tutte le sessioni. Tra i criteri impiegati in questa ricerca (che esamineremo a breve) abbiamo accolto anche tali ipotesi, raffrontandole con altre descritte in letteratura, tra cui quelle osservate da Arcuri e Fraser[4] e da Ciupa et al.[5], che si fondano su un limite temporale dei test proporzionale al numero di classi dell'applicazione in esame.

3.2 Criteri di terminazione adottati

Nel corso delle nostre sperimentazioni abbiamo contemplato 5 tipologie di criteri, in modo da mettere in evidenza quello più idoneo per l'interruzione automatica dei test. Prima di illustrarle singolarmente, però, definiamo la seguente notazione:

- ST_i: generica sessione composta da un numero di test casuali eseguiti in un periodo di tempo limitato (ad es. 60 secondi);

- $ST = \{ST_1, \ldots, ST_k\}$: insieme delle sessioni di test effettuate sull'applicazione sotto esame (con $k > 1$);
- $CC(ST_i)$: codice coperto dalla sessione ST_i, espresso in blocchi (ovvero con valori positivi interi) o linee di codice (ossia con valori positivi anche decimali);
- CS: dimensione del codice sorgente dell'applicazione, anch'esso espresso in blocchi o linee di codice;
- $PCC(ST_i)$: percentuale del codice coperto dalla sessione ST_i, descrivibile mediante l'equazione

$$PCC(ST_i) = \frac{CC(ST_i)}{CS} \times 100$$

- $UC(ST)$: unione delle coperture calcolate dalle singole sessioni appartenenti ad ST, esprimibile con l'equazione

$$UC(ST) = \bigcup_{i=1}^{k} ST_i, ST_i \in ST$$

- $PUC(ST)$: percentuale del codice coperto dall'unione delle singole sessioni appartenenti ad SS, esprimibile con l'equazione

$$PUC(ST) = \frac{UC(ST)}{CT} \times 100$$

A partire dalle suddette notazioni, possiamo definire i criteri adoperati nei test svolti:

- **CT1**: è verificato se e solo se tutti i valori percentuali di copertura delle sessioni ST_i contenute in ST convergono su quello calcolato nella loro unione $UC(ST)$. Prendendo in prestito la notazione booleana ed indicando coi simboli logici 1 e 0 rispettivamente le condizioni di vero e falso, possiamo esprimere il tutto come:

$$CT1 = 1 \iff PCC(ST_i) = PUC(ST) \forall ST_i \in ST$$

- **CT2**: è verificato se e solo se almeno una sessione di ST possiede un valore percentuale di copertura pari a quello dell'unione di tutte le sessioni. Poiché non potrà mai accadere che il valore di un qualunque ST_i superi quello di $UC(ST)$ (altrimenti sarebbe più grande della stessa unione, il che è impossibile), ma al massimo eguagliarlo, ne consegue che tale criterio verrà soddisfatto dalla sessione di ST che avrà raggiunto la maggior copertura rispetto alle altre fino a quel momento. Pertanto:

$$CT2 = 1 \iff \exists ST_i \in ST : PCC(ST_i) = PUC(ST)$$

dove $ST_i > ST_k, k \neq i$;

- **CT3**: è un criterio intermedio tra CT1 e CT2, poiché è verificato se e solo se almeno la metà delle sessioni di ST eguaglia il valore di copertura di $UC(ST)$. Nel caso in cui il numero delle sessioni k fosse dispari, si considera la parte intera superiore della suddetta quantità: ad esempio se $k = 7$, allora il criterio verrà soddisfatto se almeno $\lceil \frac{k}{2} \rceil = 4$ sessioni

di ST saranno pari a $PUC(ST)$. Riassumendo:

$$CT3 = 1 \iff$$

$$\exists STP : |STP| = \lceil \frac{k}{2} \rceil, PCC(ST_i) = PUC(ST) \forall ST_i \subseteq STP$$

dove STP è il sottoinsieme di ST contenente le sessioni che rispecchiano tale situazione di equivalenza con l'unione;

- **CT4**: è un criterio che prende in considerazione il rapporto tra il numero di eventi consecutivi in cui non è avvenuta alcuna variazione sul valore di copertura dell'unione ed il numero totale degli eventi già eseguiti. Esso si verifica se tale rapporto supera un determinato valore α compreso tra 0 e 1: ad esempio, se si desidera interrompere il processo di testing dopo che almeno per metà dei test totali non si siano registrati mutamenti sulla copertura calcolata sull'unione, allora $\alpha > 0,5$; ad ogni modo, per valutare ad ampio raggio il criterio e fissare un α che risulti più adatto alle esigenze richieste, negli esperimenti sono stati impiegati almeno 4 valori differenti di tale parametro (ben specificati nel paragrafo 5.1). In generale:

$$CT4 = 1 \iff \frac{TEST_{sv}}{TEST_{tot}} > \alpha$$

dove $TEST_{sv}$ rappresenta il numero di test consecutivi contati a partire dall'ultimo test effettuato, mentre $TEST_{tot}$ è il numero di test totali.

- **CT5**: è un criterio che interrompe il processo di testing al supera-

mento di un tempo massimo dato dal prodotto tra il numero di classi del software da testare e la durata temporale (in secondi o minuti) a loro riservata per l'esecuzione dei test. Ad esempio, secondo l'esperienza documentata da Arcuri e Fraser, ogni classe può essere testata sufficientemente per 2 minuti[4], mentre per Ciupa et al. sono stati necessari ben 90 minuti per classe per ottenere un test abbastanza predicibile;[5] ovviamente, tale valore deve essere soppesato in base al contesto in cui si sta operando. Tradotto in formule matematiche, il criterio è verificato secondo quanto segue:

$$CT5 = 1 \iff TIME > NCL \cdot TCL$$

dove $TIME$ è la durata complessiva del processo, NCL è il numero di classi testate e TCL è il tempo dedicato ad ogni classe.

3.3 Obiettivi

Dal momento che lo scopo fondamentale che vogliamo perseguire è l'idonea interruzione di test casuali automatici su alcune applicazioni Java open source facendo ricorso ai criteri poc'anzi illustrati, gli obiettivi finali da raggiungere sono i seguenti:

O1 valutare l'effettiva capacità di interruzione dei suddetti criteri;

O2 mettere a confronto, tramite l'analisi dei dati forniti dai test, le prestazioni dei vari criteri, osservandone in particolare i punti di terminazione e le percentuali di copertura.

Al fine di perseguire i suddetti obiettivi, è necessario saper progettare in maniera idonea gli esperimenti che si intendono mettere in piedi; nel prossimo paragrafo esamineremo più da vicino il processo di testing su cui essi poggiano.
A tali obiettivi sono state associate altrettante domande, a cui forniremo delle risposte nelle considerazioni che esprimeremo sui risultati ottenuti al termine degli esperimenti:

D1 I criteri prescelti sono sempre in grado di fermare il processo di testing con una percentuale di copertura corrispondente al livello di saturazione?

D2 Su quali fattori si differenziano maggiormente le prestazioni ottenute dai criteri sia in termini di copertura, sia in termini di tempo impiegato?

3.4 Processo di testing

Il processo messo a punto nel nostro lavoro può essere generalizzato secondo lo pseudocodice riportato in figura 3.2. Per com'è stato definito, esso è costituito dalla tupla *(app,nses,tses,precond,postcond,tc[])*, dove *app* è l'applicazione da testare, *nses* è il numero di sessioni (composte da vari cicli di test) in cui viene ripartito il carico delle azioni da testare, *tses* è la durata massima di ciascuna sessione, *precond* e *postcond* sono le eventuali precondizioni e postcondizioni da attribuire all'applicazione prima e dopo l'esecuzione dei test (onde evitare effetti collaterali come l'accesso incontrollato ai file in lettura e in scrittura, oppure per ignorare dipendenze da

file esterni; tali condizioni, però, potrebbero limitare la potenziale copertura massima percorribile, dato che si impedisce al processo di effettuare delle azioni corrispondenti a specifiche linee di codice) e *tc[]* è l'insieme dei criteri di terminazione impiegati per l'arresto automatico del processo.

```
1:  procedure TestProc(app,nses,tses,precond,postcond,tc[])
2:      cycle ← 0;
3:      loop START :
4:          cycle ← cycle + 1;
5:          for (i = 0; i < nses; i++) do
6:              setPrecond(app,precond);
7:              seed ← genRandSeed();
8:              cov ← executeTest(app,tses,seed);
9:              covses[i] ← mergeCovses(cov,covses[i]);
10:             covun ← mergeCovun(covses[i],covun);
11:             setPostcond(app,postcond);
12:         end for
13:         output[cycle] ← genOutput(cycle,covses[],covun);
14:         for (j = 0; j < lenght(tc[]); j++) do
15:             tc[j] ← FALSE;
16:             check ← evalTC(tcform[j],output[cycle]);
17:             if check = TRUE then tc[j] ← TRUE;
18:             end if
19:         end for
20:         tercond ← evalTerCond(tc[]);
21:         if tercond = TRUE then goto EXIT;
22:         else goto START;
23:         end if
24:         EXIT:
25:     end loop
26: end procedure
```

Figura 3.2: Algoritmo del processo di testing

La struttura del processo è di tipo iterativo: in pratica, dopo averlo avviato per la prima volta, si permane in uno stato di loop indefinito, da

cui si fuoriesce unicamente quando la condizione di terminazione che si vuole imporre viene evasa completamente. Inizialmente il contatore di cicli *cycle* viene regolato al valore 0, dopodiché si entra nel ciclo iterativo: al suo interno si attraversano obbligatoriamente 2 cicli for, uno per calcolare le percentuali di copertura e l'altro per verificare i criteri di terminazione. Nel primo, infatti, subito dopo aver assegnato le precondizioni con il metodo *setPrecond(app,precond)* e generato casualmente il valore numerico *seed* associato alla sequenza di eventi da testare, si esegue il test tramite *executeTest(app,tses,seed)*, ottenendo in output la percentuale di copertura *cov*; quest'ultima viene memorizzata nell'array *covses*, che contiene i valori delle percentuali di copertura delle singole sessioni, attraverso il metodo *mergeCovses(cov,covses[i])*, per poi procedere con l'operazione analoga *mergeCovun(covses[i],covun)* al calcolo della copertura data dall'unione delle sessioni (ovverosia *covun*) e giungere alla chiusura del ciclo for con l'inserimento delle postcondizioni in *setPostcond(app,postcond)*.
Prima di passare alla verifica dei criteri di terminazione, le cifre calcolate in precedenza vengono trasferite sull'array *output[]* con il metodo *genOutput(cycle,covses[],covun)* (pertanto ogni elemento reca in sé valori multipli, ovvero il numero corrente di cicli trascorsi e le coperture aggiornate delle sessioni e della loro unione). Espletata quest'azione, si entra nel secondo ciclo for: innanzitutto i criteri di terminazione vengono resettati al valore booleano *FALSE*, quindi si passa al controllo delle condizioni relative a ciascun criterio per mezzo di *evalTC(tcform[j],output[cycle])* (dove tcform[] rappresenta l'insieme delle formule dei criteri illustrate precedentemente), salvandone l'esito (costituito da un valore booleano) nella variabile *check*;

se quest'ultima risulta vera (ossia pari a *TRUE*), significa che il criterio corrente è stato soddisfatto e può essere segnato anch'esso come "vero". Al termine del controllo singolo, si accede infine a quello collettivo, dove i criteri che partecipano attivamente alla condizione di terminazione *tercond* vengono valutati in *evalTerCond(tc[])*; se l'esito della verifica è positivo (ovvero uguale a *TRUE*), allora si consente l'uscita dal ciclo di loop grazie all'istruzione di salto EXIT, altrimenti si ritorna al punto di partenza contrassegnato dall'etichetta START, ripetendo nuovamente tutti i passaggi descritti.

Ora che abbiamo mostrato il processo di testing nella sua interezza, non ci resta che concretizzare realmente le operazioni riportate nello pseudocodice. Per fare ciò, è necessario non solo reperire e/o creare *ex novo* gli strumenti adatti per analizzare e testare il software, ma bisogna anche saperli assemblare opportunamente per avere dei risultati apprezzabili; sarà questo l'argomento di cui ci occuperemo nel capitolo 4.

Capitolo 4

Software per il testing

In questo capitolo descriveremo nel dettaglio i 3 strumenti software impiegati per l'attuazione del processo di testing casuale su programmi Java: due di essi, ovvero Randoop ed EMMA, sono degli applicativi per il testing automatico liberamente disponibili ed ampiamente documentati in letteratura, mentre la terza è un file batch di nostra concezione per sistemi operativi Microsoft Windows che coniuga i succitati programmi in una serie di comandi eseguiti tramite l'interfaccia a riga di comando, generando in output dei file CVS (*Comma Separated Value*) contenenti i risultati del test. Per la loro valutazione rimandiamo il lettore al capitolo 5, dove discuteremo approfonditamente della sperimentazione effettuata.

4.1 Randoop

Randoop è un generatore di casi di unità di test per programmi sviluppati in Java (ad esempio compilati in archivi eseguibili con estensione .jar);

concepito dalle ricerche effettuate da Ernst e Pacheco nel 2007,[15][29] questo strumento è giunto (all'8 dicembre 2016) alla versione 3.0.8.[1] Il suo funzionamento è basato su una tecnica di testing casuale a retroazione (*feedback-directed random testing*):[15] a partire dalle classi del programma da testare (che devono essere indicate al tool tramite un file di testo in cui vanno elencati i loro percorsi all'interno dell'archivio `.jar`, senza però indicare l'estensione `.class`), esso crea delle sequenze casuali che successivamente sottopone al programma stesso, costruendosi di volta in volta delle asserzioni sul suo comportamento. In questo modo è possibile sia evidenziare eventuali falle e/o eccezioni dell'applicativo sotto esame, sia ottenere dei test di regressione per evitare l'immissione di errori nelle versioni revisionate. In figura 4.1 è stato riportato lo schema riassuntivo del funzionamento di Randoop, così com'era stato rappresentato dai suoi autori originari.[29]

Inoltre, il tool consente di specificare tramite parametri facoltativi sia il

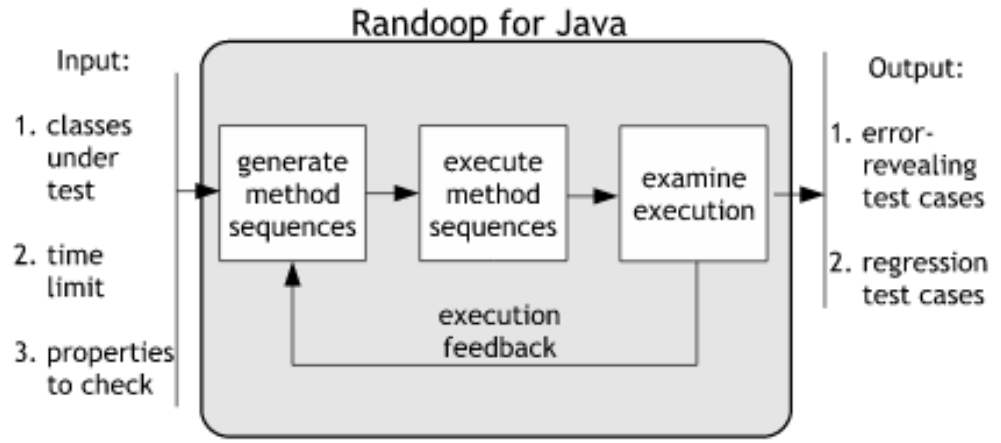


Figura 4.1: Schematizzazione del funzionamento di Randoop

[1] https://github.com/randoop/randoop/releases/

random seed (cioè un valore intero con cui estrarre la sequenza casuale di eventi; se non specificato, sarà pari a 0) sia un limite temporale (denominato *time limit*) entro cui concludere il test sulla singola sequenza (di default esso è fissato a 60 secondi). Per gli scopi che ci siamo prefissati, Randoop servirà unicamente per registrare le reazioni dei programmi da collaudare dopo avergli fornito tramite il file batch il *random seed* (che verrà sempre variato, dato che ci serve avere una nuova sequenza di eventi da testare) e il *time limit* (che, come vedremo, sarà regolato sui 20 secondi per sequenza), senza preoccuparci né del rilevamento di bug né dei test di regressione.

4.2 EMMA

EMMA è uno strumento impiegato per le analisi di copertura del codice sorgente di applicazioni Java. Sebbene l'ultimo aggiornamento risalga al mese di giugno del 2005[2], esso resta comunque un tool più che valido per misurare ed ottenere report sul codice coperto durante il processo di testing. EMMA consente di effettuare le analisi sul programma da testare sia in modalità "offline" (ovvero scindendo l'instrumentazione dell'applicazione sotto esame e la sua esecuzione) sia in modalità "on the fly" (cioé verificando gli eventi a tempo di esecuzione); noi adopereremo la prima opzione, in cui EMMA si attiverà nel momento in cui verrà lanciato Randoop, sia perché quest'ultimo ci viene già in soccorso nella simulazione degli eventi selezionati a caso, sia per aver un'unica instrumentazione uguale per tutte le sessioni di test avviate. Attraverso EMMA è possibile esplorare il codice sorgente secondo varie

[2] https://sourceforge.net/projects/emma/files/

tipologie: per classi, per metodi, per linee e per blocchi. Inoltre, grazie alla funzione di "merge" possiamo aggregare facilmente i risultati di sessioni differenti appartenenti ad un medesimo ciclo, ottenendo quindi la loro unione, che fungerà da pietra di paragone per la verifica dei criteri di terminazione.

4.3 File batch Starter

Affinché tutte le operazioni di svolgimento del processo di testing (dall'esecuzione di Randoop ed EMMA alla raccolta dei dati) venissero compiute senza eccessivi interventi manuali, si è provveduto alla creazione di un file batch, denominato semplicemente "`starter.bat`" (d'ora in avanti Starter), che permette al tester di far partire il processo di testing, attendere il suo completamento a seconda del/i criterio/i di terminazione prescelto (per il momento configurabile solo all'interno del codice sorgente, dato l'obiettivo a fini sperimentali di comparare più criteri e non uno soltanto) e visionare i dati raccolti nei file CSV generati in output e conservati in una cartella denominata secondo il seguente formato standard: `name-AAAA-MM-GG-HH-MM-SS`, dove `name` è il nome dell'applicazione testata, mentre `AAAA-MM-GG-HH-MM-SS` è la data (ordinata per anno, mese e giorno) e l'orario (ordinato per ore, minuti e secondi) in cui è stato lanciato il test. Inoltre, qualora il processo si sia interrotto volontariamente (ad es. chiudendo la finestra del prompt in cui lo si stava eseguendo) o involontariamente (ad es. per mancanza di corrente elettrica), il tester ha la facoltà di farlo ripartire dal punto in cui ci si è fermati sfruttando apposite variabili d'ingresso studiate per il ripristino. Ove richiesto, il tool supporta anche l'inserimento di insiemi di precondizioni

e postcondizioni del test sotto forma di file batch denominati rispettivamente `precond-name.bat` e `postcond-name.bat` (ovviamente, al posto di `name` va ricopiato il nome dell'applicazione). La sintassi di Starter è la seguente:

```
starter.bat name sessions tlimit alpha [flagresume]
[cycleresume] [mydatetimeresume] [valuesresume] [seedvalue]
```

Parametri d'ingresso

Illustriamo brevemente i parametri immissibili nel file batch:

- `name`: è il nome dell'applicazione che si vuole testare e deve coincidere esattamente con il nome del file .jar ad essa associato;
- `sessions`: è il numero di sessioni (ognuna composta da cicli di test) che si vuole avviare congiuntamente nel corso del test;
- `tlimit`: è il tempo limite (in secondi) da indicare a Randoop per l'interruzione del ciclo di test sulla sessione corrente;
- `alpha`: è il valore del rapporto tra il numero di eventi senza variazioni sulla copertura dell'unione ed il numero totale di eventi, così come osservato per la formulazione del criterio di terminazione CT4 nel paragrafo 3.2;
- `flagresume`: è una variabile facoltativa che, se pari a 1, abilita il ripristino di un test interrotto prematuramente;
- `cycleresume`: seconda variabile di ripristino che serve per indicare il numero di cicli finora compiuti;

- `mydatetimeresume`: terza variabile di ripristino, usata per segnalare il nome della cartella di output associata all'applicazione che si stava testando;

- `valuesresume`: è una stringa contenente i precedenti valori massimi di copertura tra le singole sessioni e nell'unione (sia a livello di blocchi sia a livello di linee di codice) e quelli di conteggio dei cicli senza variazioni di copertura che il tester deve immettere (recuperandoli manualmente dal file di output `parameters.txt` e separandoli col delimitatore ";") affinché si riprenda senza ripercussioni la verifica dei criteri di terminazione;

- `seedvalue`: variabile facoltativa che permette l'utilizzo di un seed fisso (che verrà incrementato di 1 dopo ogni ciclo) al posto di quello casuale generato di default dal file batch.

File di output

Al termine di almeno un ciclo di eventi, il tool creerà come output i seguenti file:

- `loccoverage.csv` raccoglie in sé i dati elaborati da EMMA circa la copertura del codice per tutte le sessioni, nonché i valori logici relativi ai criteri di terminazione, seppur formattati su una singola colonna. Tale scelta è giustificata dal modo in cui è stato progettato il file batch: siccome i calcoli sulle coperture e sulle condizioni di terminazione vengono svolti su cicli interni separati, non è tecnicamente possibile riportare

direttamente su una singola linea di testo i dati appartenenti ad un determinato ciclo di eventi. Si tratta, comunque, di un file testuale che funge da base per la costruzione della tabella dei dati vera e propria; inoltre, può all'occorrenza essere impiegato in fase di diagnostica per accertarsi che l'output sia quello atteso.

- `loccoveragecompact.csv` è identico al precedente `loccoverage.csv`, ma riarrangiato in maniera tale da riportare i dati incolonnati separatamente, così da poter essere facilmente consultabili su qualsiasi editor di fogli elettronici che supporta il formato CSV (Microsoft Office Excel, LibreOffice Calc, ecc.). Questo file rappresenta la versione finale della tabella dei dati, in quanto generato appositamente al termine dell'intero processo di testing. In figura 5.4 è stato inserito, a titolo esemplificativo, come appare la tabella interna al file se si apre quest'ultimo in Excel: su ogni riga sono riportati i valori di ciascun ciclo, mentre ad ogni colonna corrisponde uno specifico significato (se si osserva l'intestazione in alto, `CICLO` indica l'indice progressivo di ogni ciclo, i vari `B#` ed `L#` si riferiscono alle percentuali di copertura per la sessione `#` rispettivamente sui blocchi e sulle linee di codice, `UB` ed `UL` sono le percentuali sull'unione delle coperture delle sessioni rispettivamente sui blocchi e sulle linee di codice; sono inoltre presenti, sebbene non appaiano in figura, anche le singole colonne relative ai criteri di terminazione - anche qui sia sui blocchi sia sulle linee).

- `parameters.txt` è un file di testo in cui vengono memorizzati i valori delle variabili di ripristino, molto utile sia per il corretto riavvio del

test in caso di interruzione irregolare sia come diagnostica di controllo su tali valori. Questi ultimi sono raggruppati in 5 righe per ogni ciclo di eventi, pertanto le righe da 1 a 5 si riferiscono al primo ciclo, le righe da 6 a 10 al secondo, e via discorrendo.

- `seeds.txt` registra infine i valori di seed estratti durante il test, in modo da poterli riutilizzare per un'eventuale esecuzione futura.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	CICLO	B1	L1	B2	L2	B3	L3	B4	L4	B5	L5	B6	L6	B7	L7	B8	L8	UB	UL
2	1	1,979058	2,77444	1,21466	1,960677	12,2199	12,53413	1,895288	2,55598	11,13089	10,7373	1,089005	1,796832	11,13089	10,7373	0,125654	0,163845	13,2356	13,67559
3	2	31,34031	38,89132	1,21466	1,960677	12,55497	13,08028	31,13089	38,50901	11,25654	10,90115	12,2199	12,53413	40,10471	46,23703	0,125654	0,163845	42,08377	49,01147
4	3	42,08377	49,01147	2,104712	2,938285	12,55497	13,08028	41,87435	48,62916	13,02618	13,29328	12,2199	12,53413	40,10471	46,23703	11,25654	10,90115	42,08377	49,01147
5	4	42,08377	49,01147	13,2356	13,67559	12,55497	13,08028	41,87435	48,62916	13,02618	13,29328	12,2199	12,53413	40,23037	46,45003	11,3822	11,11415	42,08377	49,01147
6	5	42,08377	49,01147	13,2356	13,67559	12,55497	13,08028	42,08377	49,01147	13,02618	13,29328	12,42932	12,91644	41,87435	48,62916	41,87435	48,62916	42,08377	49,01147
7	6	42,08377	49,01147	13,2356	13,67559	12,55497	13,08028	42,08377	49,01147	13,02618	13,29328	13,10995	13,51174	42,08377	49,01147	41,87435	48,62916	42,08377	49,01147
8	7	42,08377	49,01147	13,2356	13,67559	12,55497	13,08028	42,08377	49,01147	13,02618	13,29328	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
9	8	42,08377	49,01147	42,08377	49,01147	12,55497	13,08028	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
10	9	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
11	10	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
12	11	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
13	12	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
14	13	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
15	14	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
16	15	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
17	16	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
18	17	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
19	18	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
20	19	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147
21	20	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	13,2356	13,67559	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147	42,08377	49,01147

Figura 4.2: Esempio di tabella di dati contenuta in `loccoveragecompact.csv`

Esempi d'uso

Se si vuole avviare per la prima volta un test sull'applicazione `javaapp.jar` con 8 sessioni, 20 secondi per evento testato ed $\alpha = \frac{4}{5}$, allora si dovrà inserire nel prompt dei comandi la seguente riga di testo:

```
starter.bat javaapp 8 20 ‘‘4;5’’
```

Se invece si desidera riprendere il test sul suddetto programma, nell'ipotesi che si siano già compiuti 61 cicli, la cartella di salvataggio abbia come etichetta `starter-2017-01-09_20-45-16`, i valori massimi di copertura nell'unione siano 1800 blocchi e 156,8 linee di codice e i contatori di variazione

immutata registrino 55 cicli sia per i blocchi sia per le linee, allora si digiterà questa riga di testo:

```
starter.bat javaapp 8 20 ‘‘4;5’’ 1 61
starter-2017-01-09_20-45-16 ’’1800;156,8;55;55’’
```

Si tenga comunque presente che in caso di ripristino di un test i parametri facoltativi, sebbene non siano obbligatori in linea teorica, vanno comunque compilati interamente (ad eccezione di `seedvalue`), onde evitare che ci siano alterazioni nei risultati prodotti in output; in più, se l'ultimo ciclo in esecuzione non è stato completato, il file `loccoverage.csv` deve essere ripulito dai risultati parziali memorizzati prima dell'interruzione (lasciando comunque una riga vuota alla fine del file), in maniera tale che il processo di compattazione per generare `loccoveragecompact.csv` non faccia slittare i valori tra le righe, rendendo infruibile la tabella dei risultati.

4.4 Codice sorgente di Starter

Per concludere degnamente questa documentazione sugli strumenti software poc'anzi illustrati, non ci resta che osservare più da vicino il codice sorgente del file batch Starter, così da riuscire a comprendere meglio il suo meccanismo interno, in cui vengono effettivamente coinvolti sia Randoop sia EMMA. Data la sua lunghezza e per agevolarne la lettura, suddivideremo il codice in 3 macroblocchi.

Macroblocco 1

Nella parte iniziale di Starter si trovano le istruzioni preparatorie che precedono l'avvio vero e proprio del test. Innanzitutto si procede a disattivare sul prompt la stampa a video dei comandi del file batch (linee 1-2), a preimpostare la generazione dei numeri random da impiegare come seed per le sequenze di eventi (linee 3-4) e a leggere i parametri d'ingresso descritti nel paragrafo precedente (linee 5-21). Quelli il cui valore è rappresentato da una stringa di caratteri racchiusa tra due virgolette ed intervallata da punti e virgola vengono scomposti per mezzo di un ciclo for al fine di separare le informazioni: da una parte il numeratore e il denominatore di α (linee 23-26), dall'altra i valori massimi di copertura tra le sessioni, i valori massimi di copertura dell'unione e i contatori di variazione immutata (linee 28-34). Nel passo successivo vengono dichiarati sia i percorsi di ubicazione degli archivi eseguibili .jar di EMMA (linee 36-37) e Randoop (linee 39-40) sia il nome della cartella che conterrà i file di output (linee 42-44). Fatto ciò, si passa ad instrumentare l'applicazione da testare attraverso EMMA (linee 46-47), imponendo come modalità di output (`-m`) l'opzione `fullcopy` (che consente il ricopiamento delle classi instrumentate senza sovrascrivere il codice sorgente), come destinazione (`-d`) la cartella di output `mydatetime` (che verrà generata proprio durante questo passaggio), come elementi da escludere nell'instrumentazione (`-ix`) le classi definite nel file di testo `myfilters.txt` e come output (`-out`) il file `coverage.em`, che racchiude in sè i metadati del programma instrumentato, che non dovranno essere nuovamente rigenerati e verranno impiegati sistematicamente per il calcolo delle coperture. Successi-

vamente si effettua la creazione preliminare di `loccoverage.csv`, inserendo al suo interno le intestazioni delle colonne della tabella finale dei dati (linee 49-65) ed infine si impostano ai valori iniziali alcuni parametri interni riguardanti i cicli di test, ovvero il contatore dei cicli `cycle` (linee 67-70), i registri dei valori massimi di copertura (linee 72-75), che serviranno per memorizzare i picchi di percentuale del codice coperto, e i contatori di variazione dell'ultimo valore dell'unione delle coperture (linee 77-79), che invece saranno utili per determinare in quale momento il criterio di terminazione T4 sarà verificato.

```
1  @echo off
2  REM preimpostazione per la generazione di numeri random
3  setlocal EnableDelayedExpansion
4  REM nome del test
5  set name=%1
6  REM numero di sessioni (macchine in esecuzione)
7  set sessions=%2
8  REM tempo limite massimo (in secondi) per ogni sessione
9  set tlimit=%3
10 REM valore di alpha (rapporto cicli senza
      variazioni/cicli totali)
11 set alpha=%~4%
12 REM flag abilitazione ripristino (abilitato solo se pari
      a 1)
13 set flagresume=%5
14 REM numero di cicli già effettuati
15 set cycleresume=%6
16 REM nome della cartella che li contiene
17 set mydatetimeresume=%7
18 REM valori di maxcoverB, maxcoverL, maxvalueB, maxvalueL
      , varcounterB e varcounterL prima dell'interruzione (
      separati da ;)
19 set valuesresume=%~8%
20 REM valore di seed fisso
21 set seedvalue=%9
22
23 for /f "tokens=1-2 delims=;" %%a in ("!alpha!") do (
```

```
24 set /A nmr = %%a
25 set /A dnm = %%b
26 )
27
28 for /f "tokens=1-5 delims=;," %%a in ("!valuesresume!")
      do (
29 if !flagresume!==1 set maxvalueB=%%a
30 if !flagresume!==1 set maxvalueL=%%b
31 if !flagresume!==1 set maxvalueLdec=%%c
32 if !flagresume!==1 set varcounterB=%%d
33 if !flagresume!==1 set varcounterL=%%e
34 )
35
36 REM percorso della cartella dove è ubicato il JAR di
      EMMA
37 set emmapath=".\emma-2.0.5312\lib\emma.jar"
38
39 REM percorso della cartella dove è ubicato Randoop
40 set RANDOOP_PATH="."
41
42 REM imposta denominazione cartella con nome dell'
      applicazione, data e ora di avvio del test
43 if !flagresume!==1 set mydatetime=%mydatetimeresume%
44 if not !flagresume!==1 for /f "tokens=1-6 delims=.,:/ "
      %%a in ("%date% %time%") do set mydatetime=%name%-%%c
      -%%b-%%a_%%d-%%e-%%f
45
46 REM instrumentazione dell'archivio jar e creazione della
       cartella per contenere gli output
47 java -noverify -cp %cycle% emma instr -m fullcopy -d
      %mydatetime% -cp ".\jars\%name%.jar" -ix @myfilters.
      txt -out %mydatetime%\coverage.em
48
49 REM scrivi intestazione della tabella dei risultati
      finali su file CSV
50 set /A sessionsplus = %sessions%+1
51 if not !flagresume!==1 for /L %%L in (1,1,%sessionsplus%
      ) do (
52 if %%L==1 @echo CICLO;>> %mydatetime%\loccoverage.csv
53 if not %%L==%sessionsplus% @echo B%%L;>> %mydatetime%\
      loccoverage.csv
54 if not %%L==%sessionsplus% @echo L%%L;>> %mydatetime%\
      loccoverage.csv
```

```
55 if %%L==%sessionsplus% @echo UB;>> %mydatetime%\
       loccoverage.csv
56 if %%L==%sessionsplus% @echo UL;>> %mydatetime%\
       loccoverage.csv
57 if %%L==%sessionsplus% @echo TB1;>> %mydatetime%\
       loccoverage.csv
58 if %%L==%sessionsplus% @echo TB2;>> %mydatetime%\
       loccoverage.csv
59 if %%L==%sessionsplus% @echo TB3;>> %mydatetime%\
       loccoverage.csv
60 if %%L==%sessionsplus% @echo TB4;>> %mydatetime%\
       loccoverage.csv
61 if %%L==%sessionsplus% @echo TL1;>> %mydatetime%\
       loccoverage.csv
62 if %%L==%sessionsplus% @echo TL2;>> %mydatetime%\
       loccoverage.csv
63 if %%L==%sessionsplus% @echo TL3;>> %mydatetime%\
       loccoverage.csv
64 if %%L==%sessionsplus% @echo TL4>> %mydatetime%\
       loccoverage.csv
65 )
66
67 REM setta il contatore dei cicli
68 if !flagresume!==1 set cycle=%cycleresume%
69 if not !flagresume!==1 set cycle=0
70 set /A cycleresumeplus = %cycleresume%+1
71
72 REM setta i parametri per registrare i valori massimi di
       copertura
73 if not !flagresume!==1 set maxvalueB=0
74 if not !flagresume!==1 set maxvalueL=0
75 if not !flagresume!==1 set maxvalueLdec=0
76
77 REM setta i parametri per i contatori di variazione dell
       'ultimo valore dell'unione delle coperture
78 if not !flagresume!==1 set varcounterB=0
79 if not !flagresume!==1 set varcounterL=0
80 REM FINE MACROBLOCCO 1
```

Macroblocco 2

Nella sezione intermedia del codice risiedono i comandi che fanno partire il test, effettuano il calcolo della copertura del codice e salvano i risultati nei file di output. Per cominciare, viene definita un'etichetta di loop `START:` (linea 81), che serve ad indicare il punto di ritorno per la riesecuzione di un nuovo ciclo di test qualora la condizione di terminazione non si fosse ancora verificata, compiendo pertanto un salto a ritroso. Subito dopo si incrementa di un'unità il contatore di ciclo (linea 82), ricopiando il suo nuovo valore nel file `loccoverage.csv` (linea 83), per poi procedere finalmente alla partenza del test attraverso un ciclo for (linee 84-85), che si concluderà solo quando tutte le sessioni avranno testato una sequenza di eventi cadauna. La prima istruzione lanciata (linee 87-88) riguarda l'eventuale caricamento delle precondizioni riportate in un file batch esterno (se non ve ne sono, il prompt mostrerà a video un innocuo messaggio d'errore che ne segnala l'assenza), dopodiché si estrae il numero random da assegnare al seed (linee 91-92), che per le caratteristiche intrinseche del linguaggio batch sarà un numero intero compreso tra 0 e 32767, a meno che non si sia optato per una cifra fissa (definito tramite `seedvalue`), che in tal caso verrà incrementata di 1 ad ogni giro (linea 93); il valore generato verrà inoltre salvato sul file testuale `seeds.txt` (linea 94).

A questo punto si attua l'istruzione fondamentale per il processo di testing, ovvero l'esecuzione di Randoop ad opera del comando `java` (linea 97), che comporterà anche l'attivazione implicita di EMMA; come si può notare, nelle opzioni abilitate vengono indicati vari fattori, ovvero le classi da testare

(elencate in un file di testo con lo stesso nome dell'applicazione; tale file è facilmente ricavabile aprendo l'archivio `.jar` in un archiviatore di dati come WinRAR e generando un report delle classi ivi contenute), il tempo limite per effettuare il test, il seed da impiegare per ottenere la sequenza degli eventi e la disattivazione dei test di regressione e del rilevamento degli errori. Poiché il file contenente i valori di copertura calcolati da EMMA (contraddistinto dall'estensione `.ec`) verrà salvato al di fuori della cartella di output, si provvede a ricopiarlo al suo interno (linea 98) e a cancellare il duplicato esterno (linea 99).

Ora che possediamo sia i metadati sia i valori di copertura dell'applicazione esaminata, i relativi file `.em` ed `.ec` vengono "uniti" (seppur conservandoli separatamente) grazie ad EMMA per ottenere dapprima un report in formato testuale (linee 101-102) e in seconda battuta un file unico `coverage%cycle%-%%M.es` (linee 104-105) che ci servirà per unire i risultati di diverse iterazioni che concorrono allo stesso test (anche se nei nostri esperimenti ci siamo limitati ad una sola iterazione). Difatti nel passo successivo si provvede a tale scopo fondendo i loro dati in un file collettivo `coverageunion%cycle%-%%M.es` (linee 107-112), e dopo un eventuale caricamento delle postcondizioni (linee 113-114) si procede allo stesso modo per ottenere un ulteriore file `coverageunion%%M.es` che rappresenta invece le coperture appartenenti alla medesima sessione (linee 116-121), da cui si estrapola un report testuale (linee 123-124) che conterranno le coperture totali della sessione corrente sia a livello di blocchi sia a livello di linee di codice, le quali verranno entrambe memorizzate nel file /textttloccoverage.csv (linee 126-132). Dopo aver generato il report anche per quanto

concerne `coverageunion%cycle%-%%M.es` (linee 134-135), il procedimento visto poc'anzi viene in un certo senso reiterato per pervenire l'unione delle coperture registrate su tutte le sessioni: pertando si effettua il "merge" del succitato file .es in un altro denominato `coverageunion.es` (linee 137-142), da cui ricavare il report sulla copertura di tale unione (linee 146-147) e ricopiarne i valori di blocchi e di linee coperte in `loccoverage.csv` (linee 149-155).

```
81  :START
82  set /a cycle+=1
83  @echo  %cycle%;>> %mydatetime%\loccoverage.csv
84  REM ciclo for per eseguire più sessioni consecutive
85  for /L %%M in (1,1,%sessions%) do (
86
87      REM setta precondizioni
88      call .\conds\precond-%name%.bat
89
90      REM imposta un valore random per il seed (se non
            impostato)
91      if not defined seedvalue set flagvalue=1
92      if !flagvalue!==1 set seedvalue=!RANDOM!
93      if not !flagvalue!==1 set /A seedvalue+=1
94      @echo !seedvalue! >> %mydatetime%\seeds.txt
95
96      REM esegui test con Randoop e salva le coperture
            calcolate da EMMA in un file .ec
97      java -noverify -cp "°%mydatetime%°\lib\°%name%°.jar"
            ;%cycle%;randoop-all-3.0.8.jar randoop.main.Main
            gentests --classlist=jars\%name%.txt --timelimit=
            !tlimit! --randomseed=!seedvalue!
            --no-error-revealing-tests=true
            --no-regression-tests=true
98      copy coverage.ec %mydatetime%\coverage%cycle%-%%M.ec
99      del coverage.ec
100
101     REM genera il report testuale attraverso i file .em
            e .ec
```

```
102     java -cp %cycle% emma report -r txt -Dreport.txt.out
           .file=%mydatetime%\coverage%cycle%-%%M.txt -in
           %mydatetime%\coverage.em,%mydatetime%\coverage
           %cycle%-%%M.ec
103
104     REM fondi i file .em e .ec in un unico file .es
105     java -cp %cycle% emma merge -in %mydatetime%\
           coverage.em,%mydatetime%\coverage%cycle%-%%M.ec
           -out %mydatetime%\coverage%cycle%-%%M.es
106
107     REM fusione dei risultati ottenuti in un unico file
           per ottenere la copertura degli eventi in M
108     if exist %mydatetime%\coverageunion%cycle%-%%M.es (
109       java -cp %cycle% emma merge -in %mydatetime%\
            coverage%cycle%-%%M.es,%mydatetime%\
            coverageunion%cycle%-%%M.es -out %mydatetime%\
            coverageunion%cycle%-%%M.es
110     ) else (
111       java -cp %cycle% emma merge -in %mydatetime%\
            coverage%cycle%-%%M.es -out %mydatetime%\
            coverageunion%cycle%-%%M.es
112     )
113     REM setta postcondizioni
114     call .\conds\postcond-%name%.bat
115
116     REM fusione dei risultati ottenuti in un unico file
           per ogni sessione per ottenere la copertura
           totale di ogni M
117     if exist %mydatetime%\coverageunion%%M.es (
118     java -cp %cycle% emma merge -in %mydatetime%\
           coverageunion%cycle%-%%M.es,%mydatetime%\
           coverageunion%%M.es -out %mydatetime%\
           coverageunion%%M.es
119     ) else (
120     java -cp %cycle% emma merge -in %mydatetime%\
           coverageunion%cycle%-%%M.es -out %mydatetime%\
           coverageunion%%M.es
121     )
122
123     REM genera il report della copertura degli eventi
124     java -cp %cycle% emma report -r txt -Dreport.txt.out
           .file=%mydatetime%\coverageunion%%M.txt -in
           %mydatetime%\coverageunion%%M.es
```

```
125
126     REM salva su file di testo il numero di blocchi e di
            linee coperti nell'unione delle sessioni in M
127     set /A count=0
128     for /F "skip=5 tokens=1-13 delims=(/)" %%a in (
            %mydatetime%\coverageunion%%M.txt) do (
129     set /A count+=1
130     if !count!==1 @echo =100*%%h/%%i;>> %mydatetime%\
            loccoverage.csv
131     if !count!==1 @echo =100*%%k/%%L;>> %mydatetime%\
            loccoverage.csv
132     )
133
134   REM genera il report della copertura degli eventi
135   java -cp %cycle% emma report -r txt -Dreport.txt.out.
        file=%mydatetime%\coverageunion%cycle%-%%M.txt -in
        %mydatetime%\coverageunion%cycle%-%%M.es
136
137     REM fusione dei risultati ottenuti in un unico file
            per ottenere la copertura totale sui vari M
138     if exist %mydatetime%\coverageunion.es (
139     java -cp %cycle% emma merge -in %mydatetime%\
            coverageunion%cycle%-%%M.es,%mydatetime%\
            coverageunion.es -out %mydatetime%\coverageunion.
            es
140     ) else (
141     java -cp %cycle% emma merge -in %mydatetime%\
            coverageunion%cycle%-%%M.es -out %mydatetime%\
            coverageunion.es
142     )
143
144 )
145
146 REM genera il report della copertura totale degli eventi
147 java -cp %cycle% emma report -r txt -Dreport.txt.out.
    file=%mydatetime%\coverageunion.txt -in %mydatetime%\
    coverageunion.es
148
149 REM salva su file CSV il numero di blocchi e di linee
    coperti nell'unione totale di tutti i vari M
150   set /A count=0
151   for /F "skip=5 tokens=1-13 delims=(/)" %%a in (
        %mydatetime%\coverageunion.txt) do (
```

```
152   set /A count+=1
153     if !count!==1 @echo =100*%%h/%%i;>> %mydatetime%\
          loccoverage.csv
154   if !count!==1 @echo =100*%%k/%%L;>> %mydatetime%\
        loccoverage.csv
155   )
156 REM FINE MACROBLOCCO 2
```

Macroblocco 3

Il blocco inferiore del codice sorgente di Starter è riservato alla verifica dei criteri di terminazione (escluso CT5, che verrà elaborato manualmente, saranno considerati solo i primi 4) tramite la lettura dei valori di copertura memorizzati precedentemente nel file `loccoverage.csv` affinché si scelga di rieseguire un nuovo ciclo, proseguendo in tal modo il processo di testing, oppure arrestandolo se la condizione imposta è stata soddisfatta. Prima di tutto si configurano alcune variabili tampone (linee 159-164), la cui utilità è quella di conservare i valori che vengono di volta in volta riesumati per poter effettuare i confronti, oltre ad azzerare i contatori di variazione immutata (linee 165-168) e i valori massimi di copertura tra le sessioni (linee 169-174). Finalmente entrano in gioco anche le variabili dei criteri di terminazione (linee 175-186), che risultano anch'esse sdoppiate per controllarne l'esito sia sui blocchi sia sulle linee di codice. Ad eccezione di TB1 e TL1 (regolate al valore logico 1), tutti i criteri sono impostati inizialmente al valore logico 0 per un motivo piuttosto semplice: mentre T2 e T3 necessitano che solo parte delle sessioni attive pervengano alle condizioni richieste e T4 opera su un rapporto derivante dal confronto tra i valori passati e presenti della copertura dell'unione, per cui tutti e tre impiegheranno un certo tempo per

passare alla condizione di "vero", T1 presuppone invece che tutte le sessioni abbiano lo stesso valore di copertura, per cui è sufficiente che un solo valore differisca dagli altri per rendere falsa tale condizione. Seguono infine ulteriori variabili di estrema importanza per il regolare funzionamento delle procedure di verifica: `checkskip` rappresenta il numero di righe di testo da saltare al momento della lettura dei valori contenuti in `loccoverage.csv` (linea 187), `halfsessions` corrisponde al valore k già citato nella formulazione di CT3 al paragrafo 3.2 (linea 188), `doublesessionsplus` e `doublesessionsplus2` servono per rispettare l'ordine di lettura e di confronto dei valori di copertura (linee 189-190), `nmrcycle` è un valore dato dal prodotto tra il numeratore di α e il numero di cicli eseguiti che verrà impiegato per la determinazione della condizione di vero per CT4 (linea 191), mentre `count` è un contatore che enumera quante righe di `loccoverage.csv` sono state lette (linea 192). Una volta allestito il tutto, si può procedere all'avvio di un altro ciclo for, da cui verranno emessi i verdetti sui 4 criteri di terminazione (linee 193-241) e si passerà (grazie al salto verso l'etichetta `:checkandcompact` dalla linea 240 alla 243) alla fase finale del processo di testing, vale a dire il compattamento dei dati e la valutazione della condizione di terminazione. Qui il contenuto del file `loccoverage.csv` verrà duplicato e riformattato in versione tabellare sul file gemello `loccoveragecompact.csv` mediante l'ennesimo ciclo for (linee 244-254), dopodiché si controllerà la condizione imposta per fuoriuscire dal ciclo di loop (linee 255-256): se risulta vera, si salterà all'etichetta di uscita `:goodbye` (linea 258), da cui con l'istruzione di salto alla fine del file batch `GOTO:eof` (linea 259) si porrà fine all'esecuzione del test, ma se al contrario risulta falsa si ritornerà indietro all'etichetta `:START` per proseguire i test e

compiere un nuovo ciclo (linea 257).

```
157 REM verifica i criteri di terminazione attraverso i
    valori salvati nel file CSV
158 REM set iniziale parametri
159 REM variabile tampone per memorizzare il valore della
    copertura di una sessione sui blocchi
160 set storeB=0
161 REM variabile tampone per memorizzare il valore intero
    della copertura di una sessione sulle linee di codice
162 set storeL=0
163 REM variabile tampone per memorizzare il valore decimale
     della copertura di una sessione sulle linee di
    codice
164 set storeLdec=0
165 REM contatore del numero di sessioni con valore di
    copertura sui blocchi uguale tra loro
166 set maxcounterB=0
167 REM contatore del numero di sessioni con valore di
    copertura sulle linee di codice uguale tra loro
168 set maxcounterL=0
169 REM variabile tampone per memorizzare il valore massimo
    della copertura tra le sessioni sui blocchi
170 set maxcoverB=0
171 REM variabile tampone per memorizzare il valore intero
    massimo della copertura tra le sessioni sulle linee
    di codice
172 set maxcoverL=0
173 REM variabile tampone per memorizzare il valore decimale
     massimo della copertura tra le sessioni sulle linee
    di codice
174 set maxcoverLdec=0
175 REM T1 indica se tutte le sessioni hanno lo stesso
    valore di copertura oppure no
176 set TB1=1
177 set TL1=1
178 REM T2 indica se la più grande copertura tra le sessioni
     è uguale all'unione delle coperture
179 set TB2=0
180 set TL2=0
181 REM T3 indica se almeno la metà delle sessioni hanno lo
    stesso valore dell'unione delle coperture
182 set TB3=0
```

```
183 set TL3=0
184 REM T4 indica se il rapporto tra il n. di cicli dall'
       ultima variazione e il n. di cicli trascorti è pari a
        1\2
185 set TB4=0
186 set TL4=0
187 set /A checkskip=((((!sessions!+5)*2)+1)*!cycle!)+1
188 set /A halfsessions=!sessions!/2
189 set /A doublesessionsplus=(!sessions!*2)+1
190 set /A doublesessionsplus2=(!sessions!*2)+2
191 set /A nmrcycle=!nmr!*!cycle!
192 set /A count=0
193   for /F "skip=%checkskip% tokens=1-3 delims=/,;*" %%a
         in (%mydatetime%\loccoverage.csv) do (
194   set /A count+=1
195   set /A modulus=!count! %% 2
196   if !modulus! EQU 0 if %%c GEQ 10 set /A dec = 0
197   if !modulus! EQU 0 if %%c LSS 10 set /A dec = %%c
198   if !count! LSS !doublesessionsplus! if !count! EQU 1
         set /A storeB = %%b
199   if !count! LSS !doublesessionsplus! if !count! EQU 2
         set /A storeL = %%b
200   if !count! LSS !doublesessionsplus! if !count! EQU 2
         set /A storeLdec = !dec!
201   if !count! LSS !doublesessionsplus! if !modulus! EQU 1
          if !storeB! NEQ %%b set /A TB1 = 0
202   if !count! LSS !doublesessionsplus! if !modulus! EQU 0
          if !storeL! NEQ %%b set /A TL1 = 0
203   if !count! LSS !doublesessionsplus! if !modulus! EQU 0
          if !storeLdec! NEQ !dec! set /A TL1 = 0
204   if !count! LSS !doublesessionsplus! if !modulus! EQU 1
          if %%b GEQ !maxcoverB! set /A maxcoverB = %%b
205   if !count! LSS !doublesessionsplus! if !modulus! EQU 0
          if %%b GTR !maxcoverL! set /A maxcoverLdec = !dec!
206   if !count! LSS !doublesessionsplus! if !modulus! EQU 0
          if %%b GTR !maxcoverL! set /A maxcoverL = %%b
207   if !count! LSS !doublesessionsplus! if !modulus! EQU 0
          if %%b EQU !maxcoverL! if !dec! GTR !maxcoverLdec!
          set /A maxcoverLdec = !dec!
208   if !count! LSS !doublesessionsplus! if !modulus! EQU 1
          if %%b EQU !maxvalueB! set /A maxcounterB+=1
209   if !count! LSS !doublesessionsplus! if !modulus! EQU 0
          if %%b EQU !maxvalueL! if !dec! EQU !maxvalueLdec!
```

```
        set /A maxcounterL+=1
210     if !count! EQU !doublesessionsplus!  if !maxcoverB!
        EQU %%b set /A TB2 = 1
211     if !count! EQU !doublesessionsplus2! if !maxcoverL!
        EQU %%b if !maxcoverLdec! EQU !dec! set /A TL2 = 1
212     if !count! EQU !doublesessionsplus!  if not !cycle!
        EQU %cycleresumeplus% if not %%b EQU !maxvalueB!
        set /A varcounterB=0
213     if !count! EQU !doublesessionsplus2! if not !cycle!
        EQU %cycleresumeplus% if not %%b EQU !maxvalueL!
        set /A varcounterL=0
214     if !count! EQU !doublesessionsplus2! if not !cycle!
        EQU %cycleresumeplus% if not !dec! EQU !
        maxvalueLdec! set /A varcounterL=0
215     if !count! EQU !doublesessionsplus!  if %%b EQU !
        maxvalueB! set /A varcounterB+=1
216     if !count! EQU !doublesessionsplus2! if %%b EQU !
        maxvalueL! if !dec! EQU !maxvalueLdec! set /A
        varcounterL+=1
217     if !count! EQU !doublesessionsplus!  set /A dnmvarcntB
        =!dnm!*!varcounterB!
218     if !count! EQU !doublesessionsplus2! set /A dnmvarcntL
        =!dnm!*!varcounterL!
219     if !count! EQU !doublesessionsplus!  if !dnmvarcntB!
        GTR !nmrcycle! set /A TB4 = 1
220     if !count! EQU !doublesessionsplus2! if !dnmvarcntL!
        GTR !nmrcycle! set /A TL4 = 1
221     if !count! EQU !doublesessionsplus!  if %%b GEQ !
        maxvalueB! set /A maxvalueB = %%b
222     if !count! EQU !doublesessionsplus2! if %%b GTR !
        maxvalueL! set /A maxvalueLdec = !dec!
223     if !count! EQU !doublesessionsplus2! if %%b GTR !
        maxvalueL! set /A maxvalueL = %%b
224     if !count! EQU !doublesessionsplus2! if %%b EQU !
        maxvalueL! if !dec! GTR !maxvalueLdec! set /A
        maxvalueLdec = !dec!
225     if !count! EQU !doublesessionsplus2! @echo maxcounterB
         = !maxcounterB!, maxcounterL = !maxcounterL! >>
        %mydatetime%\parameters.txt
226     if !count! EQU !doublesessionsplus2! @echo maxcoverB =
         !maxcoverB!, maxcoverL = !maxcoverL!,!maxcoverLdec
        ! >> %mydatetime%\parameters.txt
```

```
227     if !count! EQU !doublesessionsplus2! @echo maxvalueB =
          !maxvalueB!, maxvalueL = !maxvalueL!,!maxvalueLdec
        ! >> %mydatetime%\parameters.txt
228     if !count! EQU !doublesessionsplus2! @echo varcounterB
          = !varcounterB!, varcounterL = !varcounterL!,
        cycle = !cycle! >> %mydatetime%\parameters.txt
229     if !count! EQU !doublesessionsplus2! @echo dnmvarcntB
        = !dnmvarcntB!, dnmvarcntL = !dnmvarcntL!, nmrcycle
          = !nmrcycle! >> %mydatetime%\parameters.txt
230     if !count! EQU !doublesessionsplus! if !maxcounterB!
        GEQ !halfsessions! set /A TB3 = 1
231     if !count! EQU !doublesessionsplus! if !maxcounterL!
        GEQ !halfsessions! set /A TL3 = 1
232     if !count! EQU !doublesessionsplus!  @echo !TB1!; >>
        %mydatetime%\loccoverage.csv
233     if !count! EQU !doublesessionsplus!  @echo !TB2!; >>
        %mydatetime%\loccoverage.csv
234     if !count! EQU !doublesessionsplus!  @echo !TB3!; >>
        %mydatetime%\loccoverage.csv
235     if !count! EQU !doublesessionsplus!  @echo !TB4!; >>
        %mydatetime%\loccoverage.csv
236     if !count! EQU !doublesessionsplus2! @echo !TL1!; >>
        %mydatetime%\loccoverage.csv
237     if !count! EQU !doublesessionsplus2! @echo !TL2!; >>
        %mydatetime%\loccoverage.csv
238     if !count! EQU !doublesessionsplus2! @echo !TL3!; >>
        %mydatetime%\loccoverage.csv
239     if !count! EQU !doublesessionsplus2! @echo !TL4! >>
        %mydatetime%\loccoverage.csv
240     if !count! EQU !doublesessionsplus2! goto
        checkandcompact
241     )
242
243   :checkandcompact
244   REM riformatta il contenuto di loccoverage.csv per
        adattarlo a mo' di tabella
245   set /A cdvalue=(((!sessions!+5)*2)+1)
246   set /A countdown=!cdvalue!
247   set /A skiplines=!countdown!*!cycle!
248   if !cycle!==1 set SKIP="delims="
249   if not !cycle!==1 set SKIP="skip=%skiplines% delims="
250   for /F %SKIP% %%a in (%mydatetime%\loccoverage.csv) do (
251     set /A countdown-=1
```

```
252     set row=!row!%%a
253     if !countdown! EQU 0 (set /A countdown=!cdvalue!& echo
            .!row:~1!& set row=)
254 ) >> %mydatetime%\loccoveragecompact.csv
255 REM terminazione regolare se sono soddisfatti i criteri
        imposti nella condizione
256 if !TB1! EQU 1 if !TB2! EQU 1 if !TB3! EQU 1 if !TB4!
        EQU 1 goto goodbye
257 goto START
258 :goodbye
259 GOTO:eof
260 REM FINE MACROBLOCCO 3
```

Capitolo 5

Sperimentazione

Ora che i concetti e i meccanismi che si celano dietro le nostre ricerche sono stati sviscerati fino in fondo, è giunto il momento di mostrare i risultati degli esperimenti condotti al fine di rispondere in maniera soddisfacente alle domande sugli obiettivi che avevamo elencato nel paragrafo 3.3.

5.1 Impostazione degli esperimenti

La sperimentazione è stata condotta con l'impiego di un personal computer con sistema operativo Windows 7 a 32 bit e processore Intel Pentium Dual Core T3400. I test sono stati avviati tramite un file batch esterno, denominato `batchtest.bat`, al cui interno sono state riportate una o più chiamate a Starter secondo la sintassi descritta al paragrafo 4.3. Sebbene sia tecnicamente fattibile lanciare finestre multiple di Starter (così da eseguire allo stesso tempo più test su applicazioni differenti), è comunque preferibile non eccedere col numero di esecuzioni contemporanee a causa del continuo

cambio di contesto (*context switch*) del sistema operativo; al contrario si consiglia di eseguirne addirittura un'istanza alla volta per ottenere migliori prestazioni nei test. D'altronde, un eventuale parallelismo sarebbe stato giustificabile solo se si fosse operato su più elaboratori, oppure se i test fossero prevalentemente dei processi I/O bound anziché CPU bound, ma così non è, dato che l'unico I/O di una certa rilevanza è la scrittura dei file di copertura, pertanto non ci sono motivi validi per supportare l'esecuzione in parallelo avendo a disposizione un solo elaboratore.

La scelta per il software open source in linguaggio Java oggetto dei test automatici casuali è ricaduta sul gruppo dei 110 programmi di prova di Evosuite, noto anche come SF110[1] (già analizzati da Arcuri e Fraser nel 2014[30]), da cui è stato prelevato un campione di 11 applicazioni che fossero sia non troppo complesse sia supportate da Randoop (poiché non tutte quelle provate preliminarmente sono risultate gestibili a causa di particolari eccezioni sollevate che hanno causato l'aborto spontaneo della generazione delle unità di test, invalidando quindi l'intero processo); in aggiunta ad esse, inoltre, è stata collaudata anche un'applicazione avulsa dal suddetto gruppo, in modo da comparare la loro bontà rispetto ad un programma prelevato da un contesto differente. L'elenco completo del software provato è riportato integralmente nella tabella 5.1

Per ogni processo di testing si è deciso di eseguire 8 sessioni di test, con un tempo limite fissato a 20 secondi. Ciò è stato motivato da alcune indagini precedenti, in cui sono stati impiegati gruppi di 4, 8 e 12 sessioni,

[1] http://www.evosuite.org/experimental-data/sf110/
[2] Applicazione non appartenente al gruppo SF110

Applicazione	Nome	Descrizione	N. classi	N. metodi	N. blocchi	N. linee
APP1	Battlecry	Generatore casuale di brani musicali	15	130	9550	1831
APP2	dcParseArgs	Libreria Java per la gestione di `args[]`	6	21	654	110
APP3	FalseLight	Sistema di monitoraggio delle reti	14	32	1189	200
APP4	JCLO	Analizzatore di variabili in classi Java	4	43	1094	202
APP5	JGAAP	Framework grafico	23	95	3140	638
APP6	JIPA	Interprete Java	5	36	1488	228
APP7	Noen	Framework per analisi dinamiche	18	71	1353	-
APP8	Playgame[2]	Videogioco	8	87	3964	808
APP9	SAXPath	Strumento per applicazioni XPath	12	257	3115	955
APP10	Sugar	Strumento per programmatori Java	14	80	2089	407
APP11	Transmitter Locator	Rilevatore di trasmettitori	6	39	1097	238
APP12	Tullibee	Componente per il trading online	19	204	8402	1798

Tabella 5.1: Metriche delle 12 applicazioni sotto esame

da cui si è appurato che utilizzandone solo 4 i livelli di terminazione, così come i relativi punti, risultano più bassi del dovuto, mentre non sono state apprezzate sostanziali differenze tra i valori ricavati con 8 sessioni e quelli ottenuti con 12 (sebbene i tempi di terminazione siano maggiori impiegando 12 sessioni); per quanto riguarda il tempo limite massimo per lo svolgimento di un ciclo, invece, si è optato per un quantitativo temporale ristretto rispetto a quello di default imposto da Randoop (ovverosia 60 secondi) per avere un periodo di campionamento più basso su cui valutare meglio l'andamento della copertura. Inoltre, per quanto concerne il criterio CT4, sono stati presi in considerazione non uno, ma bensì 4 valori differenti di α, ovvero $0,5$ (50%), $0,6$ (60%), $0,7$ (70%) e $0,8$ (80%), per ampliare la gamma dei criteri previsti, sia per determinare un valore di tale parametro che si attagli meglio ai requisiti che ci siamo prefissati per decretare il criterio vincente. Dato che Starter è stato configurato per ammettere e memorizzare nella tabella finale di `loccoveragecompact.csv` un unico valore di α (che negli esperimenti è stato fissato a $0,8$), le varianti non calcolate in via diretta sono state ricavate operando manualmente sui dati di output tramite le formule matematiche disponibili su Microsoft Office

Excel. Nel codice sorgente descritto nel paragrafo 4.4, in particolare nel macroblocco 3, si può notare che la condizione di terminazione imposta nel file batch per gli esperimenti è quella di arrestare i test nel momento in cui tutti i criteri implementati giungano al valore logico di 1, così da essere sicuri di poter osservare i loro punti di terminazione: se se ne fosse scelto uno soltanto, infatti, avremmo corso il rischio di tralasciarne altri che avrebbero necessitato di più cicli di tempo prima di essere soddisfatti; ciò comporta, però, che il livello massimo di copertura desunto dal punto di terminazione dell'ultimo criterio commutato alla condizione di "vero" non sia il valore dell'effettiva saturazione, ma piuttosto un suo potenziale candidato di cui si suppone che sia ad esso equiparabile nell'ipotesi che la copertura globale non registri ulteriori incrementi oltre tale punto.

5.2 Analisi dei risultati

5.2.1 Obiettivo O1

Per capire se l'obiettivo O1 introdotto nel paragrafo 3.3 sia stato conseguito (e, di conseguenza, fornire una risposta alla domanda D1), dovremo confrontare per ogni singolo esperimento le percentuali di copertura registrate nei punti di terminazione dei criteri adottati con i livelli massimi di copertura (vale a dire le percentuali relative all'unione delle coperture delle 8 sessioni). Nella tabella 5.2 sono stati dunque riportati i risultati numerici degli esperimenti effettuati sulle 12 applicazioni campione, prendendo però

in considerazione soltanto i primi 4 tipi di criteri, dal momento che CT5 non era stato implementato in Starter ed è stato quindi analizzato separatamente: per ogni criterio è possibile leggere (da sinistra verso destra) il numero di cicli eseguiti, il tempo impiegato in secondi e le percentuali di terminazione relative alle coperture sui blocchi e sulle linee di codice; in basso a destra, invece, sono stati indicati i livelli massimi di copertura. Per brevità di notazione, CT1 è indicato con la sigla T1, CT2 con T2, CT3 con T3, CT4 ad $\alpha = 0,5$ con T50, CT4 ad $\alpha = 0,6$ con T60, CT4 ad $\alpha = 0,7$ con T70 e CT4 ad $\alpha = 0,8$ con T80. Le percentuali di terminazione coincidenti con i livelli massimi di copertura sono stati evidenziati in neretto, così da avere un rapido colpo d'occhio sui criteri che hanno rispettato con maggior frequenza tale requisito. I dati relativi alla tipologia di criterio CT5, la quale si fonda su un tempo finito per ogni classe costituente l'applicazione da testare, sono invece stati elencati nella tabella 5.3: affinché potesse essere comparata in maniera equa rispetto a quelli già analizzati, essa è stata declinata in tre criteri, denominati SC20, SC40 ed SC60, che prevedono rispettivamente 20, 40 e 60 secondi per classe, e a cui sono stati assegnati d'ufficio i valori registrati dall'unione delle sessioni per quanto concerne le percentuali di copertura.
In generale la copertura del codice non è stata particolarmente spinta, attestandosi il più delle volte tra il 40 e il 60% del totale e registrando un picco minimo del 12,9% in APP8 ed uno massimo del 89,9% in APP7 (caratterizzato, tra l'altro, anche per la mancanza di rilevamento delle coperture a livello di linee di codice a causa della sua particolare instrumentazione): ciò è dipeso sia dalla casualità degli eventi testati, sia dalla logica del codice sorgente, anche se c'è da tener presente che non è possibile affermare con

certezza che i livelli di copertura registrati siano effettivamente quelli massimi, dal momento che esiste comunque una remota possibilità di ampliamento del codice coperto se si fossero prolungati gli esperimenti oltre l'interruzione programmata su Starter. Al di là di tali osservazioni, dando uno sguardo alla tabella 5.2, in cui per ogni applicazione esaminata è stata riportata la differenza tra la copertura individuata dai 10 criteri adottati e il livello massimo di copertura, possiamo constatare che in ben 7 casi su 12, ovverosia su APP1, APP3, APP5, APP6, APP7, APP11 e APP12, tale valore è risultato uguale a 0 per tutti i criteri, il che significa che in quelle applicazioni si è sempre raggiunta la parità tra i due valori a prescindere dal criterio con cui s'intende interrompere il processo. Andando, invece, ad analizzare la tabella colonna per colonna, si scopre che solo il criterio T1 è stato in grado di arrestarsi con una copertura pari al livello massimo in tutti gli esperimenti; al contrario, gli altri criteri non sono riusciti a centrare sempre tale obiettivo, poiché T3 ed SC60 ce l'hanno fatta in 10 casi su 12, T80 ed SC40 in 9 su 12, T50, T60 e T70 in 8 su 12 e T2 ed SC20 in 7 su 12. Laddove la differenza non è stata nulla, però, il divario è stato però piuttosto contenuto, visto che sono emerse sostanzialmente divergenze che oscillano tra lo 0,3 e l'1 %, tranne in casi particolari, come la differenza del 10,2% riscontrata nella quasi totalità dei criteri applicati ad APP4, oppure quella del 4,6% rilevata sul criterio T50 in APP9. Pertanto, possiamo concludere che in generale i criteri prescelti per gli esperimenti effettuati in questo lavoro di ricerca consentono una terminazione accettabile con un livello di copertura uguale e/o pressoché vicino a quello massimo ottenuto globalmente; in particolare, da questo punto di vista, il criterio T1 è risultato essere quello più affidabile, sebbene tale pregio

(come rimarcheremo nel prossimo paragrafo) venga inficiato dal numero elevato di cicli da esso impiegato per giungere a compimento rispetto agli altri criteri.

5.2.2 Obiettivo O2

Per poter rispondere alla domanda D2, abbiamo tracciato l'avanzamento dei test in base ai dati sperimentali raccolti per capire esattamente come si siano evolute le coperture e in quali frangenti si siano verificate le condizioni di terminazione. Nelle figure 5.1, 5.2, 5.3 e 5.4 sono visibili i grafici relativi a ciascun esperimento, strutturati in scala logaritmica esattamente come nell'esempio illustrato anteriormente in figura 3.1: le linee intere nere rappresentano la copertura dell'unione delle sessioni, le linee tratteggiate colorate si riferiscono alle coperture delle singole sessioni, mentre i pallini neri segnalano i punti di terminazione dei 7 criteri adottati). Ricordiamo che sull'asse delle ordinate sono indicati i valori percentuali di copertura del codice dell'applicazione testata (segnalata nel titolo del grafico), mentre sull'asse delle ascisse sono indicati il numero di cicli eseguiti nel processo di testing. Per maggior comodità di lettura, in corrispondenza dei punti di terminazione sono state inserite delle linee aggiuntive verticali nella griglia interna e specificati sull'asse delle ascisse i valori dei cicli in cui un criterio è passato allo stato di "vero", indicandolo con la sua sigla nella parte alta del grafico.

Dai grafici si può evincere che gli esperimenti condotti hanno assunto in sintesi 2 tipologie di andamenti: il primo, in cui ricadono APP1, APP2,

APP3, APP5, APP6, APP7 e APP11, prevede una terminazione precoce del processo (ovvero al di sotto dei 100 cicli), con un avanzamento della copertura dell'unione delle sessioni molto rapida nella parte iniziale che si stabilizza e rimane costante fino alla fine; la seconda, invece, in cui possiamo annoverare APP4, APP9, APP10 ed APP12, è caratterizzata da un processo che si protrae a lungo (vale a dire per centinaia o addirittura migliaia di cicli) e che presenta una progressione della copertura rallentata e discontinua, la quale raggiunge una certa stabilità solo dopo molto tempo. Curiosamente, APP8 (ovvero l'applicazione non appartenente al gruppo SF110) costituisce un caso limite che si piazza a metà strada tra le suddette evoluzioni: fino al ciclo 11 la copertura unificata del programma testato ha mostrato una discreta costanza, senonché nel ciclo 12, a causa dello scatenamento di un evento particolare, quest'ultima ha subìto un incremento repentino che non è stato compensato altrettanto velocemente in tutte le sessioni, obbligando quindi il processo di testing ad attendere numerosi cicli prima che il criterio di terminazione T1 venisse rispettato.

Nonostante l'ordine cronologico di arresto dei vari criteri cambi sensibilmente da un esperimento all'altro, è possibile scorgere qualche dato significativo. Nella tabella 5.5 sono stati comparati i punti di terminazione di tutti i criteri (espressi attraverso i numeri di cicli trascorsi per giungere a compimento), marcando in neretto quelli in cui la copertura registrata nel corrispondente punto di terminazione ha coinciso con quella massima ed evidenziando per ogni applicazione testata quelli più veloci in rosso e quelli più lenti in blu. Da qui si può desumere che T1 è stato l'unico criterio che ha eguagliato la percentuale massima coperta in ogni esperimento (come già osservato

precedentemente), ma di contro è anche quello che ha sempre impiegato più cicli per essere soddisfatto (eccetto che in APP2, dove invece è stato T80 ad essere più tardivo); viceversa, T2 e T50 sono stati i criteri più rapidi nella terminazione, ma ciò non ha comportato un'adeguata affidabilità in termini di copertura, dal momento che il più delle volte non hanno raggiunto il livello massimo (in tal senso, il criterio T2 si è rivelato quello peggiore in assoluto, non pervenendo alla copertura massima in ben 5 casi su 12). Ovviamente, il criterio da privilegiare per l'interruzione del processo di testing dipende in grossa parte dalle priorità che si vogliono sostenere: se l'obiettivo fondamentale è l'ottenimento imprescindibile della copertura più capillare possibile senza badare troppo al fattore temporale, allora si opterà per quello che perverrà con maggior certezza a tale compito, vale a dire il già citato T1; se invece è richiesta obbligatoriamente una riduzione dei cicli di test, pur salvaguardando l'affidabilità sulla copertura massima, allora si dovranno valutare i criteri che più si avvicinano su entrambi gli aspetti, e da questo punto di vista, T3, SC60, T80 ed SC40 potrebbero essere dei buoni candidati, dal momento che sono quelli più affidabili subito dopo T1 e contengono di molto i cicli effettuati; se invece ci interessa fermare gli esperimenti il più rapidamente possibile, allora ci potremmo anche rivolgere a T2 e T50, ma con la consapevolezza di ricadere con buona probabilità in un livello di copertura poco affidabile. In estrema sintesi, i criteri osservati posseggono delle peculiarità molto contrastanti tra loro che ci conducono a risultati totalmente differenti a seconda del grado di raffinatezza del codice coperto, che in ogni caso sarà direttamente proporzionale al tempo trascorso.

App.	T1				T2			
	Cicli	Tempo (sec.)	% term. B	% term. L	Cicli	Tempo (sec.)	% term. B	% term. L
APP1	27	540	**42,1**	**49,0**	3	60	**42,1**	**49,0**
APP2	23	460	**55,0**	**67,0**	4	80	54,0	66,7
APP3	8	160	**26,0**	**24,3**	2	40	**26,0**	**24,3**
APP4	899	17980	**56,6**	**55,0**	11	220	46,3	46,3
APP5	88	1760	**50,1**	**50,7**	5	100	**50,1**	**50,7**
APP6	9	180	**53,9**	**46,8**	3	60	**53,9**	**46,8**
APP7	31	620	**89,9**	-	14	280	**89,9**	-
APP8	2306	46120	**12,9**	**16,4**	5	100	12,1	15,6
APP9	3326	66520	**58,0**	**60,5**	194	3880	57,7	60,3
APP10	376	7520	**40,7**	**43,3**	27	540	39,7	42,7
APP11	39	780	**47,3**	**49,6**	6	120	**47,3**	**49,6**
APP12	1566	31320	**44,7**	**30,1**	266	5320	**44,7**	**30,1**

App.	T3				T50			
	Cicli	Tempo (sec.)	% term. B	% term. L	Cicli	Tempo (sec.)	% term. B	% term. L
APP1	7	140	**42,1**	**49,0**	5	100	**42,1**	**49,0**
APP2	14	280	**55,0**	**67,0**	11	220	**55,0**	**67,0**
APP3	3	60	**26,0**	**24,3**	3	60	**26,0**	**24,3**
APP4	27	540	48,3	47,1	5	100	46,3	46,3
APP5	16	320	**50,1**	**50,7**	11	220	**50,1**	**50,7**
APP6	4	80	**53,9**	**46,8**	3	60	**53,9**	**46,8**
APP7	18	360	**89,9**	-	5	100	**89,9**	-
APP8	10	200	12,1	15,6	5	100	12,1	15,6
APP9	1683	33660	**58,0**	**60,5**	13	260	53,5	56,9
APP10	216	4320	**40,7**	**43,3**	19	380	39,7	42,7
APP11	12	240	**47,3**	**49,6**	5	100	**47,3**	**49,6**
APP12	426	8520	**44,7**	**30,1**	61	1220	**44,7**	**30,1**

App.	T60				T70			
	Cicli	Tempo (sec.)	% term. B	% term. L	Cicli	Tempo (sec.)	% term. B	% term. L
APP1	6	120	**42,1**	**49,0**	7	140	**42,1**	**49,0**
APP2	13	260	**55,0**	**67,0**	17	340	**55,0**	**67,0**
APP3	3	60	**26,0**	**24,3**	4	80	**26,0**	**24,3**
APP4	6	120	46,3	46,3	7	140	46,3	46,3
APP5	13	260	**50,1**	**50,7**	17	340	**50,1**	**50,7**
APP6	3	60	**53,9**	**46,8**	4	80	**53,9**	**46,8**
APP7	6	120	**89,9**	-	7	140	**89,9**	-
APP8	6	120	12,1	15,6	7	140	12,1	15,6
APP9	143	2860	57,7	60,3	191	3820	57,7	60,3
APP10	23	460	39,7	42,7	31	620	39,7	42,7
APP11	6	120	**47,3**	**49,6**	7	140	**47,3**	**49,6**
APP12	169	3380	**44,7**	**30,1**	225	4500	**44,7**	**30,1**

App.	T80				Livelli massimi	
	Cicli	Tempo (sec.)	% term. B	% term. L	% sat. B	% sat. L
APP1	11	220	**42,1**	**49,0**	42,1	49,0
APP2	26	520	**55,0**	**67,0**	55,0	67,0
APP3	6	120	**26,0**	**24,3**	26,0	24,3
APP4	11	220	46,3	46,3	56,6	55,0
APP5	26	520	**50,1**	**50,7**	50,1	50,7
APP6	6	120	**53,9**	**46,8**	53,9	46,8
APP7	11	220	**89,9**	-	89,9	-
APP8	11	220	12,1	15,6	12,9	16,4
APP9	286	5720	57,7	60,3	58,0	60,5
APP10	226	4520	**40,7**	**43,3**	40,7	43,3
APP11	11	220	**47,3**	**49,6**	47,3	49,6
APP12	351	7020	**44,7**	**30,1**	44,7	30,1

Tabella 5.2: Risultati degli esperimenti sulle 12 applicazioni in base ai criteri di terminazione implementati in Starter

App.	SC20			SC40			SC60		
	Tempo (sec.)	% term. B	% term. L	Tempo (sec.)	% term. B	% term. L	Tempo (sec.)	% term. B	% term. L
APP1	300	**42,1**	**49,0**	600	**42,1**	**49,0**	900	**42,1**	**49,0**
APP2	120	**55,0**	**67,0**	240	**55,0**	**67,0**	360	**55,0**	**67,0**
APP3	280	**26,0**	**24,3**	560	**26,0**	**24,3**	840	**26,0**	**24,3**
APP4	80	46,3	46,3	160	46,3	46,3	240	46,3	46,3
APP5	540	**50,1**	**50,7**	1080	**50,1**	**50,7**	1620	**50,1**	**50,7**
APP6	60	**53,9**	**46,8**	120	**53,9**	**46,8**	180	**53,9**	**46,8**
APP7	360	**89,9**	-	720	**89,9**	-	1080	**89,9**	-
APP8	140	12,1	15,6	280	**12,9**	**16,4**	420	**12,9**	**16,4**
APP9	340	56,4	59,2	680	57,1	59,7	1020	57,2	60,0
APP10	380	39,7	42,7	760	40,4	43,2	1140	**40,7**	**43,3**
APP11	120	**47,3**	**49,6**	240	**47,3**	**49,6**	360	**47,3**	**49,6**
APP12	420	44,1	29,2	840	**44,7**	**30,1**	1260	**44,7**	**30,1**

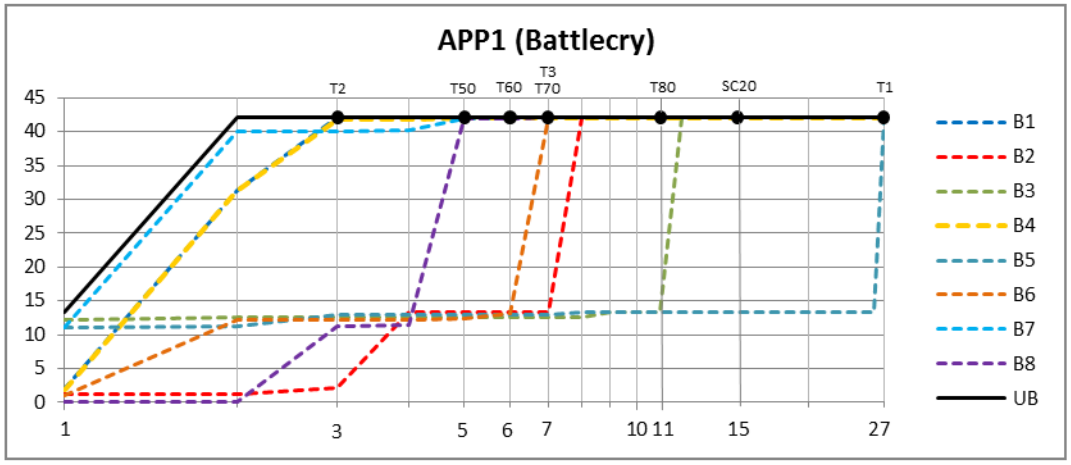
Tabella 5.3: Valori di terminazione (desumibili dai dati sperimentali) per i criteri di tipo CT5

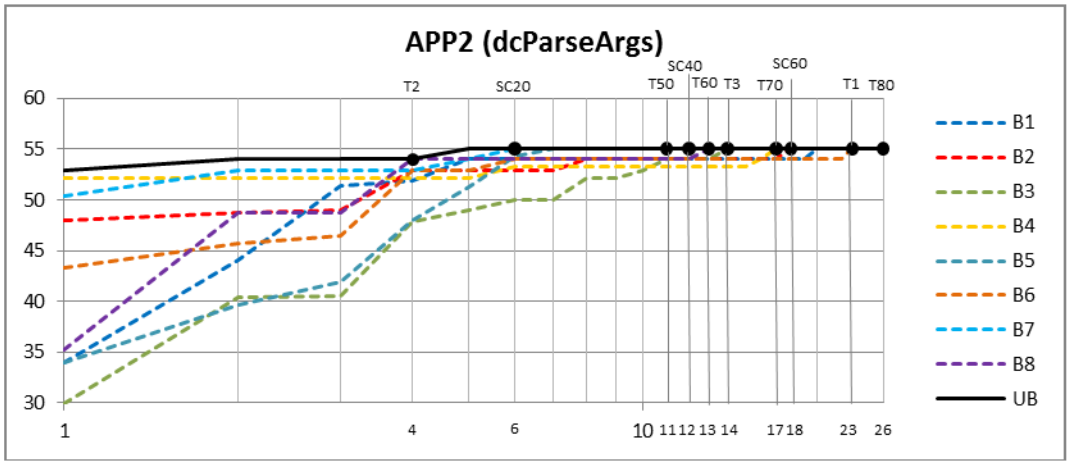
App.	T1	T2	T3	T50	T60	T70	T80	SC20	SC40	SC60
APP1	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
APP2	0,0	**1,1**	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
APP3	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
APP4	0,0	**10,2**	**8,3**	**10,2**	**10,2**	**10,2**	**10,2**	**10,2**	**10,2**	**10,2**
APP5	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
APP6	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
APP7	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
APP8	0,0	**0,8**	**0,8**	**0,8**	**0,8**	**0,8**	**0,8**	**0,8**	0,0	0,0
APP9	0,0	**0,3**	0,0	**4,6**	**0,3**	**0,3**	**0,3**	**1,7**	**0,9**	**0,9**
APP10	0,0	**1,0**	0,0	**1,0**	**1,0**	**1,0**	0,0	**1,0**	**0,3**	0,0
APP11	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
APP12	0,0	0,0	0,0	0,0	0,0	0,0	0,0	**0,6**	0,0	0,0

Tabella 5.4: Differenze tra le coperture individuate dai 10 criteri adottati ed i livelli massimi di copertura

App.	T1	T2	T3	T50	T60	T70	T80	SC20	SC40	SC60
APP1	**27**	**3**	**7**	**5**	**6**	**7**	**11**	**15**	**30**	**45**
APP2	**23**	4	**14**	**11**	**13**	**17**	**26**	**6**	**12**	**18**
APP3	**8**	**2**	**3**	**3**	**3**	**4**	**6**	**14**	**28**	**42**
APP4	**899**	11	27	5	6	7	11	4	8	12
APP5	**88**	**5**	**16**	**11**	**13**	**17**	**26**	**27**	**54**	**81**
APP6	**9**	**3**	**4**	**3**	**3**	**4**	**6**	**3**	**6**	**9**
APP7	**31**	**14**	**18**	**5**	**6**	**7**	**11**	**18**	**36**	**54**
APP8	**2306**	5	10	5	6	7	11	7	**14**	**21**
APP9	**3326**	194	**1683**	13	143	191	286	17	34	51
APP10	**376**	27	**216**	19	23	31	**226**	19	38	**57**
APP11	**39**	**6**	**12**	**5**	**6**	**7**	**11**	**6**	**12**	**18**
APP12	**1566**	**266**	**426**	**61**	**169**	**225**	**351**	21	**42**	**63**

Tabella 5.5: Confronto tra i punti di terminazione (in numero di cicli) per i 10 criteri analizzati





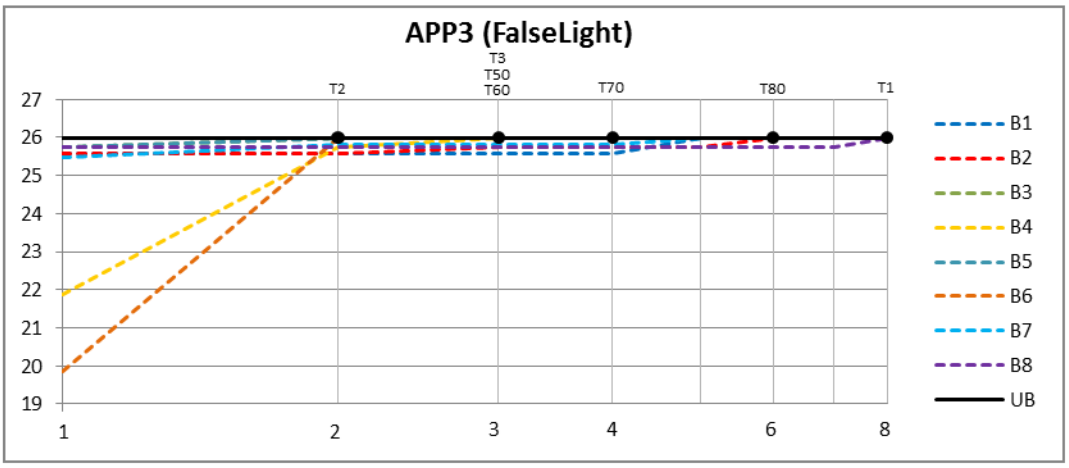
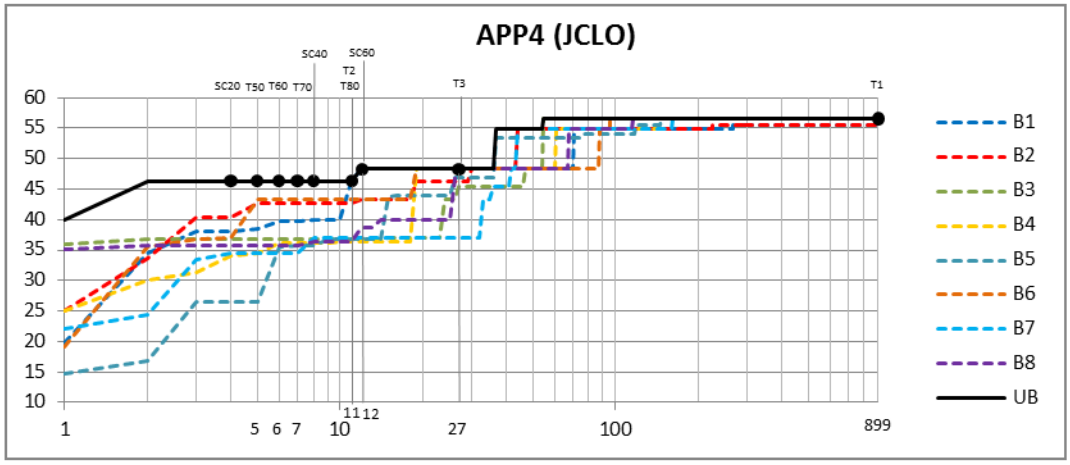
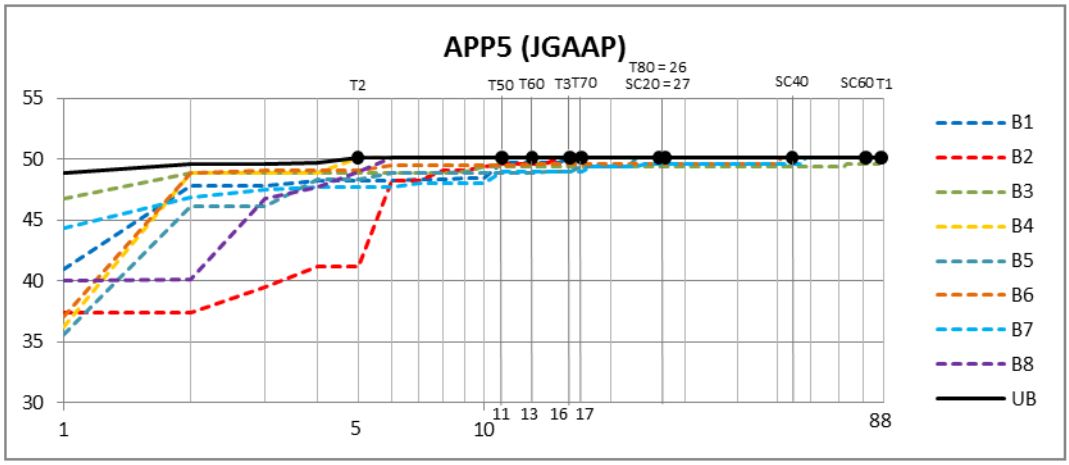


Figura 5.1: Grafici APP1, APP2 e APP3





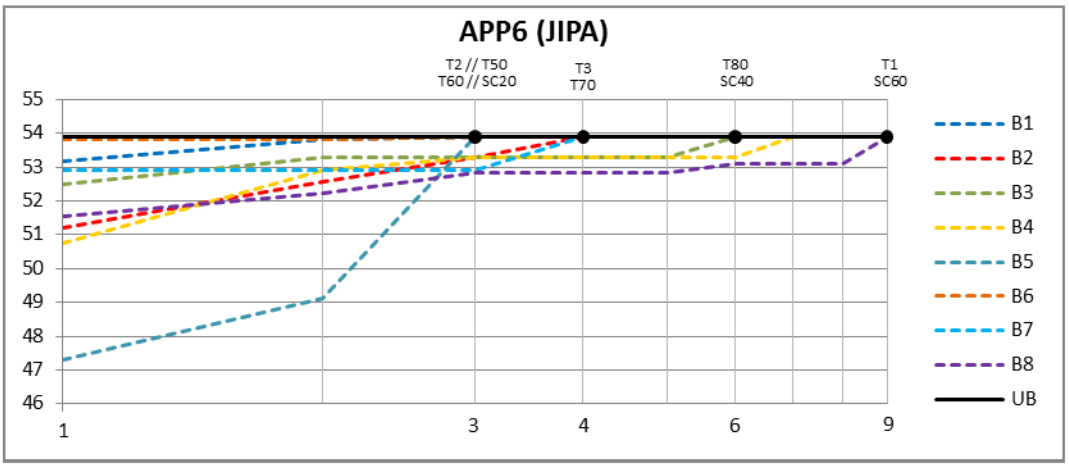
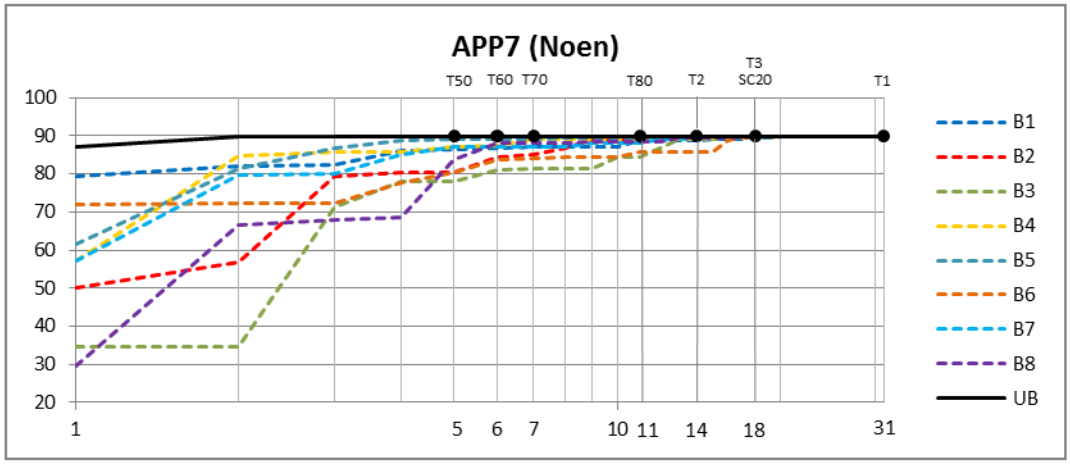
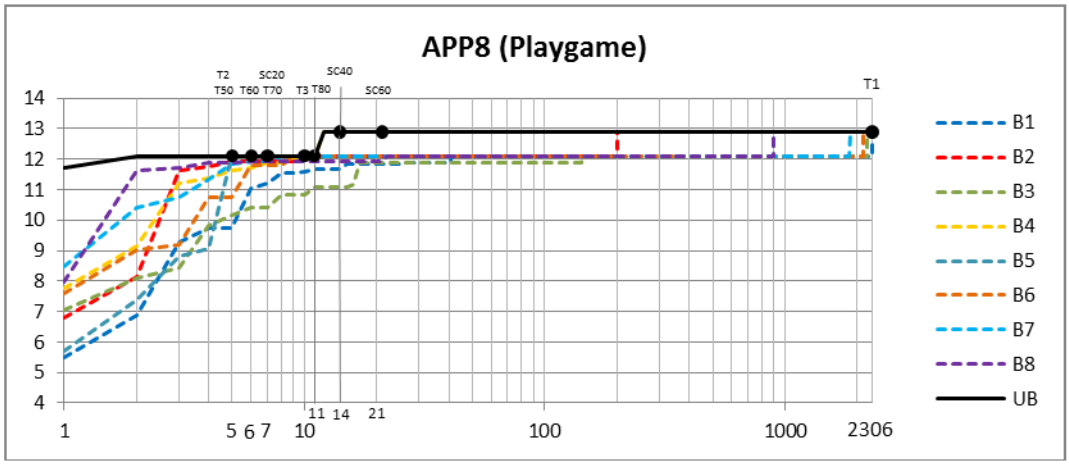


Figura 5.2: Grafici APP4, APP5 e APP6





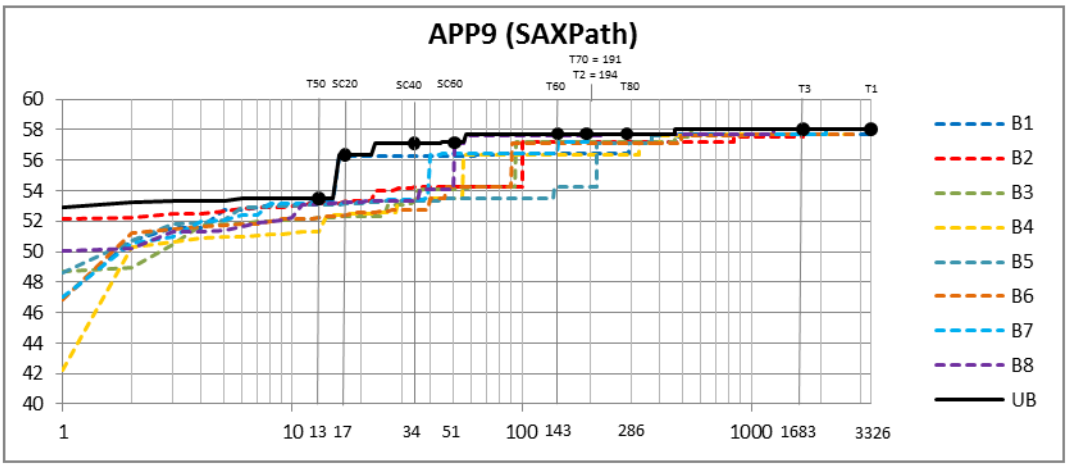
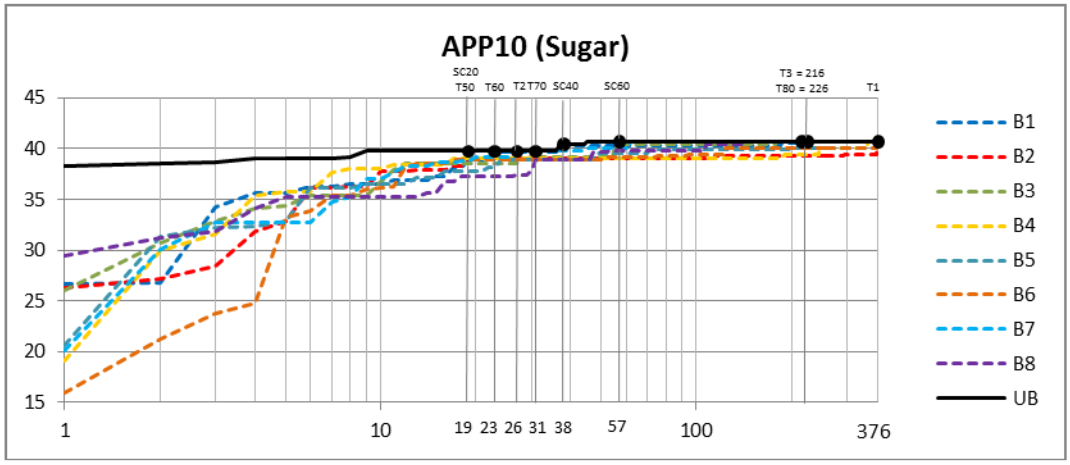
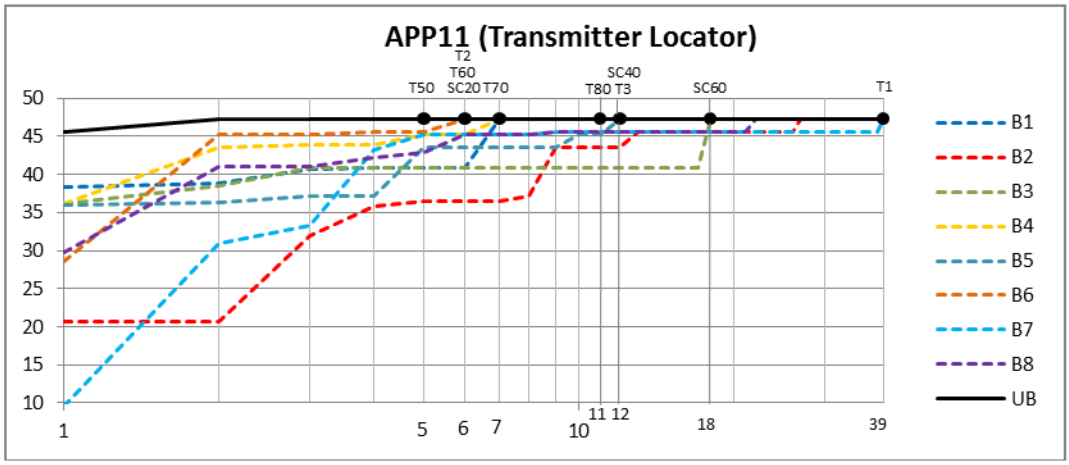


Figura 5.3: Grafici APP7, APP8 e APP9





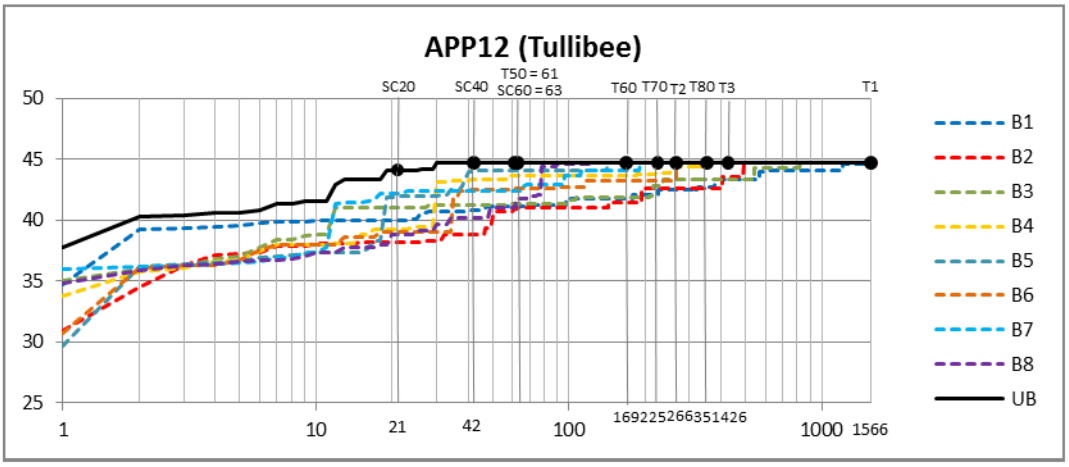


Figura 5.4: Grafici APP10, APP11 e APP12

Capitolo 6

Conclusioni

La ricerca di un criterio universale per la terminazione automatica del testing casuale sul campione di applicazioni Java analizzato si è rivelata più complessa del previsto. L'automatizzazione semplifica notevolmente le operazioni di collaudo del software, ma richiede una progettazione attenta e precisa delle procedure da applicare affinché si ottengano buone prestazioni sulle metriche di copertura del codice, contenendo contemporaneamente le risorse temporali disponibili. Attraverso la considerazione del fenomeno dell'"effetto saturazione" sono stati ipotizzati alcuni criteri che fossero in grado di interrompere il prima possibile i test senza compromettere la qualità del collaudo; per renderli applicabili, si è provveduto a strutturare un processo di testing, in seguito reso concreto in un file batch che, integrando alcuni strumenti ad hoc, ha consentito l'esecuzione e la raccolta dei dati sperimentali dei test, arrestandoli secondo le condizioni di terminazione imposte.

Nonostante sia sussistita una discreta incertezza sulla zona di effettiva saturazione, i risultati ottenuti hanno comunque messo in risalto la generale

capacità dei criteri di indicare dei potenziali punti di sospensione associati ad una percentuale di copertura del codice sorgente delle applicazioni testate abbastanza soddisfacente, seppur con modalità tra loro differenti e con esiti diversi in base alle caratteristiche intrinseche del codice scandagliato. Inoltre, dagli scenari osservati dalle tabelle e dai grafici dei dati si è cercato di capire meglio quale potesse essere la tipologia di criterio più adatta all'esigenza di ottenere il miglior collaudo col minor sforzo: CT1 si é dimostrato ottimo nella ricerca della potenziale copertura massima, ma pessimo nel contingentamento dei tempi di collaudo; CT2 si è posizionato invece all'estremo opposto, essendo efficace nella terminazione rapida, ma scarsamente affidabile nell'individuazione del punto più alto di copertura che si possa conseguire; CT3, essendo per definizione una via di mezzo tra CT1 e CT2, ha confermato tale peculiarità, piazzandosi a metà strada ed assumendo gli aspetti positivi dell'una e dell'altra (il che fa ben sperare per una sua potenziale candidatura); CT4 e CT5, attraverso i loro derivati (rispettivamente T50, T60, T70 e T80 per il primo e SC20, SC40 e SC60 per il secondo) si sono invece posti tra le fasce di collocamento delle tre tipologie poc'anzi citate, a seconda del valore che si intende attribuire ad α e a TCL (se basso sarà più vicino a CT2, se intermedio sarà simile a CT3, se alto sarà prossimo a CT1). In ogni caso, la risposta sulla preferenza di un criterio rispetto ad un altro non potrà mai essere univoca in senso stretto, data la forte influenza che il fattore di casualità degli eventi testabili esercita sul comportamento complessivo degli esperimenti condotti.

Capitolo 7

Sviluppi futuri

Al fine di migliorare il lavoro compiuto nella presente ricerca ed affinare lo studio sulle tecniche di testing casuali, nelle prossime indagini si prenderanno in considerazione le seguenti proposte:

- rendere più intuitivo ed efficiente il file batch Starter, perfezionando il suo codice sorgente ed implementando nuovi metodi e/o criteri; per conseguire tale scopo, si potrebbe riformulare parzialmente la struttura del processo di testing, oppure utilizzare altri strumenti di analisi al di fuori di Randoop ed EMMA, o ancora riprogettare Starter adottando un linguaggio di programmazione più flessibile, ma mantenendolo sempre disaccoppiato dal programma da testare;
- parallelizzare i test eseguiti nell'ambito del processo, ricorrendo all'utilizzo di molteplici calcolatori distribuiti su di una rete ed interconnessi tra loro;

- estendere la sperimentazione su campioni più significativi delle applicazioni sotto esame (in particolare quelli di dimensioni più considerevoli e/o con costrutti più complessi), fermo restando la compatibilità con gli strumenti di analisi di cui si dispone;

- rivedere le condizioni di terminazione da applicare agli esperimenti affinché si riescano a controverificare con maggior certezza i presunti livelli di saturazione misurati al termine degli esperimenti compiuti finora; in tal senso, si suggerisce di mantenere sì una condizione dettata dalla combinazione di uno o più criteri, ma una volta conseguita si prorogherà la conclusione del processo di testing con una quantità di cicli pari a metà della durata complessiva.

Bibliografia

[1] Khalid Alemerien and Kenneth Magel. Examining the effectiveness of testing coverage tools: An empirical study. http://www.sersc.org/journals/IJSEIA/vol8_no5_2014/12.pdf. ↑1

[2] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. On the effectiveness of manual and automatic unit test generation. In *Proceedings of the 2008 The Third International Conference on Software Engineering Advances*, ICSEA '08, pages 252–257, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3372-8. doi: 10.1109/ICSEA.2008.66. URL http://dx.doi.org/10.1109/ICSEA.2008.66. ↑2

[3] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. Exploiting the saturation effect in automatic random testing of android applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '15, pages 33–43, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2825041.2825046. ↑4, ↑14

[4] Andrea Arcuri and Gordon Fraser. Parameter tuning or default

values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 2013. ISSN 1382-3256. doi: 10.1007/s10664-013-9249-9. URL http://dx.doi.org/10.1007/s10664-013-9249-9. ↑4, ↑14, ↑18

[5] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In *Proceedings of the First International Conference on Software Testing, Verification and Validation (ICST'08)*, April 2008. URL http://se.inf.ethz.ch/old/people/ciupa/papers/icst08.pdf. ↑4, ↑14, ↑18

[6] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X. URL http://dl.acm.org/citation.cfm?id=257734.257766. ↑4, ↑7, ↑9

[7] Amrit L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Trans. Softw. Eng.*, 11(12):1411–1423, December 1985. ISSN 0098-5589. URL http://dx.doi.org/10.1109/TSE.1985.232177. ↑4, ↑9

[8] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*, chapter 13.4.3. 1996. ↑4

[9] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. Recent advances in automatic black-box testing. In *Advances in Computers*. Elsevier, 2015. URL http://dx.doi.org/10.1016/bs.adcom.2015.04.002. ↑5

[10] Noel Nyman. Using Monkey Test Tools. http://starcanada.techwelldev.com/sites/default/files/articles/Smzr1XDD2604filelistfilename1_0.pdf. ↑6

[11] Tsong Yueh Chen, T. H. Tse, and Yuen-Tak Yu. Proportional sampling strategy: a compendium and some insights. *Journal of Systems and Software*, 58(1):65–81, 2001. URL http://hub.hku.hk/bitstream/10722/48423/1/65653.pdf. ↑6

[12] Tsong Yueh Chen and Robert Merkel. An upper bound on software testing effectiveness. *ACM Trans. Softw. Eng. Methodol.*, 17(3):16:1–16:27, June 2008. ISSN 1049-331X. doi: 10.1145/1363102.1363107. URL http://doi.acm.org/10.1145/1363102.1363107. ↑6

[13] Huai Liu, Yansheng Lu, Tsong Yueh Chen, Xiaodong Xie, and Jing Yang. Adaptive random testing by exclusion through test profile. *Quality Software, International Conference on*, 00(undefined):92–101, 2010. ISSN 1550-6002. doi: doi.ieeecomputersociety.org/10.1109/QSIC.2010.61. ↑6

[14] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275, New York,

NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001452. URL http://doi.acm.org/10.1145/2001420.2001452. ↑6

[15] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.37. URL http://dx.doi.org/10.1109/ICSE.2007.37. ↑7, ↑9, ↑24

[16] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351717. URL http://doi.acm.org/10.1145/2351676.2351717. ↑7

[17] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Secfuzz: Fuzz-testing security protocols. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 1–7, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1822-8. URL http://dl.acm.org/citation.cfm?id=2663608.2663610. ↑7

[18] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, IS-

STA '10, pages 171–182, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. URL http://doi.acm.org/10.1145/1831708.1831730. ↑7

[19] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, 51 (4):420–426, Dec 2002. ISSN 0018-9529. doi: 10.1109/TR.2002.804489. ↑8

[20] Yi Wei, Bertrand Meyer, and Manuel Oriol. Empirical software engineering and verification. chapter Is Branch Coverage a Good Measure of Testing Effectiveness?, pages 194–212. Springer-Verlag, Berlin, Heidelberg, 2012. ISBN 978-3-642-25230-3. URL http://dl.acm.org/citation.cfm?id=2184075.2184080. ↑8

[21] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, March 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.121. ↑8, ↑9

[22] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 190–203, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2793-0. doi: 10.1145/2594368.2594377. URL http://doi.acm.org/10.1145/2594368.2594377. ↑9

[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491450. URL http://doi.acm.org/10.1145/2491411.2491450. ↑9

[24] A. Hajjar, T. Chen, I. Munn, A. Andrews, and M. Bjorkman. High quality behavioral verification using statistical stopping criteria. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 411–418, 2001. doi: 10.1109/DATE.2001.915057. ↑9

[25] M. Sahinoglu. An empirical bayesian stopping rule in testing and verification of behavioral models. *IEEE Transactions on Instrumentation and Measurement*, 52(5):1428–1443, Oct 2003. ISSN 0018-9456. doi: 10.1109/TIM.2003.818548. ↑9

[26] S. S. Gokhale and K. S. Trivedi. Log-logistic software reliability growth model. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, pages 34–41, Nov 1998. doi: 10.1109/HASE.1998.731593. ↑9

[27] James A. Whittaker and Michael G. Thomason. A markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, October 1994. ISSN 0098-5589. doi: 10.1109/32.328991. URL http://dx.doi.org/10.1109/32.328991. ↑9

[28] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. Saturation-based testing of concurrent programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 53–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595706. URL http://doi.acm.org/10.1145/1595696.1595706. ↑10, ↑12

[29] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297902. URL http://doi.acm.org/10.1145/1297846.1297902. ↑24

[30] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, December 2014. ISSN 1049-331X. doi: 10.1145/2685612. URL http://doi.acm.org/10.1145/2685612. ↑49

Ringraziamenti

Come tutte le cose della vita, dopo un inizio c'è inevitabilmente una fine: la mia carriera universitaria, protrattasi per 11 anni nella splendida cornice della città di Napoli, si conclude qui, con la consapevolezza di aver ricevuto un notevole bagaglio culturale che mi consentirà di affrontare al meglio i futuri obiettivi lavorativi. Se sono giunto sin qui, lo devo sia alla mia tenacia, sia al conforto di tutte le persone che ho avuto modo di conoscere nell'arco di quest'intensa esperienza di studio: è grazie a questi elementi se sono riuscito a sormontare le tante avversità che mi si sono parate dinanzi e a conseguire la tanto agognata laurea magistrale.

Ringrazio textit/in primis i relatori della presente tesi, vale a dire il professore Porfirio Tramontana e l'ingegnere Nicola Amatucci, che mi hanno seguito passo dopo passo nel percorso di tirocinio e che mi hanno aiutato in maniera egregia nel completamento di quest'ultimo tassello del mio iter alla "Federico II", fornendomi preziosi suggerimenti e dimostrandosi sempre disponibili nelle varie tappe di avvicininamento alla laurea.

Un sentito e doveroso ringraziamento va ai miei genitori, Pasquale e Maria Rosaria, che si accingono a festeggiare i loro primi 35 anni di matrimonio: entrambi non mi hanno mai fatto mancare nulla e si sono prodigati con

tutte le loro forze per portare avanti il mio desiderio di diventare ingegnere informatico. A loro devo solo eterna stima e gratitudine, augurandomi di ripagarli degnamente di tutti i sacrifici che hanno compiuto per me.

Non posso poi non ringraziare mia sorella Raffaella, che da un anno e mezzo sta portando avanti con successo la propria attività sartoriale e che ha contribuito personalmente al corredo per questo giorno speciale: che tu possa essere felice in ogni istante e raggiungere nuovi traguardi nel tuo lavoro!

Tornando in ambito universitario, desidero ringraziare tutti i colleghi (passati e presenti) con cui ho interagito sia di persona, sia a distanza tramite social network e forum specializzati (in particolare *Quelli di Informatica* ed *Ingegnerinforma*), scambiandoci consigli e suggerimenti: se ho potuto superare alcuni esami ostici, in parte lo devo anche a loro, a cui ho spesso e volentieri ricambiato il favore ricevuto.

Ringrazio calorosamente per la realizzazione di questa tesi l'intero staff di *Officina Studenti*, a cui mi sono rivolto sin da quando ero una semplice matricola nell'ormai remoto settembre 2005: si sono rivelati cortesi ed affabili ogni volta che mi sono recato da loro per acquistare un quaderno o stampare un blocco di appunti.

Rivolgo un grazie ai coinquilini che mi hanno accompagnato in questi ultimi 6 anni, da quelli "storici" (Alessandro, Gianluca, Mario e Peppino) con cui avevo condiviso il superamento della precedente laurea triennale e con cui ho trascorso piacevoli momenti durante la successiva scalata a quella magistrale, a quelli "recenti" (Domenico, Francesco, Peppe e Roberto) con cui ho stretto amicizia sin dal primo momento e che mi hanno supportato

fraternamente in questa volata finale: a loro auguro di avere successo in tutte le loro aspirazioni personali.

Il mio pensiero va, inoltre, a Don Danilo, che mi ha confortato spiritualmente nei momenti più tristi e che si è dimostrato un valente sacerdote sin dal suo insediamento a Conca dei Marini (di cui sono originario): che il Signore lo guidi e lo illumini nel suo delicato compito di "pastore di anime".

Per concludere, non mi resta che formulare un grazie a tutti i miei parenti, conoscenti ed amici che non sono riuscito ad elencare in queste poche righe, ma che sicuramente hanno espresso tutto il loro affetto nei miei confronti in ogni circostanza: è bello sapere di poter contare anche su di voi.