Università degli studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Progettazione e realizzazione di un tool per la generazione automatica di casi di test

Tesi in Ingegneria del software II

Relatori

Chiar.mo Prof. Andrea De Lucia

Dott. Dario Di Nucci

Candidato

Simone Scalabrino

Matr: 0522500268

Anno Accademico 2014/2015

Università degli studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Progettazione e realizzazione di un tool per la generazione automatica di casi di test

Tesi in Ingegneria del software II

Relatori

Chiar.mo Prof. Andrea De Lucia

Dott. Dario Di Nucci

Candidato

Simone Scalabrino

Matr: 0522500268

Anno Accademico 2014/2015

Πα΄ντα αριθμο΄ς εστι

Pitagora

Indice

Capitolo 1

Introduzione

La qualità di un prodotto software non è limitata all'assenza di problemi. Il termine problema, usato in questo contesto, ha un significato assai generico, si potrebbe definire come qualsiasi caratteristica o comportamento che abbia come conseguenza un discostamento dai requisiti o dalla specifica del programma.

Per avere un maggior livello di dettaglio, occorre fare una distinzione tra malfunzionamenti (failure), difetti (bug o fault), vulnerabilità ed errori del software. Un malfunzionamento è un comportamento del software difforme da quanto previsto dalla specifica. In pratica, si verifica quando, in assenza di malfunzionamenti della piattaforma (hardware e software di base), il sistema non fa quello che l'utente si aspetta. Un difetto è una sequenza di istruzioni, sorgenti o eseguibili, che, quando eseguita con particolari dati in input, genera un malfunzionamento. In pratica, si ha un malfunzionamento solo quando viene eseguito il codice che contiene il difetto, e solo se i dati di input sono tali da evidenziare l'errore. Per esempio, se invece di scrivere $a \geq 0$, il programmatore ha erroneamente scritto $a > 0$ in una istruzione, si può avere un malfunzionamento solo quando viene eseguita quell'istruzione mentre la variabile a vale zero.

Un errore ha, nella maggior parte dei casi, radici umane: una distrazione in fase di codifica può introdurre un difetto nel programma che genera malfunzionamenti. Un errore è l'origine di un difetto.

Lo scopo del testing è di rilevare quanto prima i difetti, al fine di minimizzare la probabilità che il software rilasciato abbia dei malfunzionamenti nella normale operatività. Tale probabilità non può essere portata a zero, in quanto le possibili combinazioni di input valide sono enormi e non possono essere riprodotte in un tempo ragionevole; tuttavia una buona fase di testing può rendere la probabilità di malfunzionamenti abbastanza bassa da essere accettabile per utente. Per rilevare il maggior numero possibile di difetti, nel testing si sollecita il software in modo che sia eseguita la maggior quantità possibile di codice con svariati dati di input. Le tecniche di testing si possono classificare in molti modi. I principali sono i seguenti:

- per livello di conoscenza delle funzionalità interne: testing funzionale, testing strutturale.
- per fase di sviluppo: Alfa Testing, Beta Testing.
- per grado di automazione: manuale, semi-automatizzato, completamente automatizzato.
- per granularità: Test di Unità, Test di Integrazione, Test di Sistema.

1.1 Contesto

Il software fa parte degli artefatti ingegneristici più variabili e complessi. I requisiti di qualità di un software in un ambiente possono essere molto diversi e incompatibili con quelli di un ambiente diverso o di un diverso dominio di applicazione. Si noti, inoltre, che la struttura del software evolve e spesso si deteriora con la crescita del sistema: la non linearità intrinseca dei sistemi software e la distribuzione irregolare dei guasti ne complica verifica. Il costo di verifica del software spesso rappresenta più della metà del costo complessivo di sviluppo e manutenzione. Esistono tecnologie di sviluppo avanzate e potenti strumenti di supporto in grado di ridurre la frequenza di alcune classi di errori, ma l'eliminazione totale degli errori e la produzione di software senza difetti sono obiettivi ancora lontani. In molti casi, i nuovi approcci di sviluppo introducono nuovi tipi di difetti, che possono essere

più difficili da rilevare e rimuovere rispetto a quelli classici. Ad esempio, lo sviluppo orientato agli oggetti introduce nuovi problemi dovuti all'uso del polimorfismo, del binding dinamico e dello stato privato che sono assenti o meno evidenti nel software procedurale. La varietà di problemi e la ricchezza degli approcci rendono difficile la pianificazione e la scelta del giusto mix di tecniche per raggiungere il livello di qualità richiesto all'interno dei vincoli di costo. Si noti inoltre che non esistono ricette fisse per risolvere questo problema. Anche gli specialisti più esperti non hanno soluzioni predefinite, ma hanno bisogno di progettare una soluzione che si adatti al problema, ai requisiti, e all'ambiente di sviluppo.

La fase di **Verifica e Validazione (V & V)** del software riveste un'importanza fondamentale all'interno del ciclo di sviluppo del software al fine di garantire la qualità del prodotto finale. Essa, infatti, permette di valutare il livello di qualità raggiunto dall'applicazione sviluppata verificandone la corrispondenza alle specifiche iniziali, rilevando gli errori e permettendone la correzione. Verifica e validazione rispondono rispettivamente alle domande:

- stiamo realizzando correttamente il prodotto?
- stiamo realizzando il prodotto corretto?

Obiettivo della verifica è il controllo di qualità delle attività svolte durante una fase dello sviluppo; obiettivo della validazione è il controllo di qualità del prodotto rispetto ai requisiti del committente.

È possibile interpretare il processo di sviluppo come una successione di lavorazioni di prodotti intermedi; la più stringente delle visioni è il ciclo di vita a cascata. In questa prospettiva, un'attività di verifica è il controllo che il prodotto ottenuto al termine di una fase sia congruente con il semilavorato avuto come punto di partenza di quella fase. Per esempio, nella realizzazione di un modulo, è una tipica verifica il controllo che le specifiche del modulo siano state rispettate sia come interfaccia che come funzionalità. La validazione è un'attività di controllo mirata a confrontare il risultato di una fase del processo di sviluppo con i requisiti del prodotto (tipicamente con quanto stabilito dal contratto o, meglio, dal documento di analisi dei requisiti).

Un comune esempio di validazione è il controllo che il prodotto finito abbia funzionalità e prestazioni conformi con quelle stabilite all'inizio del processo di sviluppo. La validazione è un'attività normalmente prevista sul prodotto finito. Tuttavia, limitandosi ad aspetti particolari, è comunque possibile effettuare delle operazioni di validazione anche durante il processo di sviluppo. Ad esempio, l'architettura può essere validata con i requisiti; in questo caso la validazione è una preventiva assicurazione contro possibili errori di interpretazione dei requisiti.

Verifica e validazione sono attività che si sovrappongono a quelle tradizionali e concretamente produttive del processo di sviluppo e con le quali devono integrarsi nel più efficace possibile dei modi. La scoperta di un errore durante lo sviluppo è sicuramente un guadagno per la qualità del prodotto, ma è anche un evento la cui gestione va prevista. L'organizzazione del processo di sviluppo deve prevedere e pianificare ogni attività di verifica e validazione, mediando fra la parallelizzazione e la serializzazione delle attività di sviluppo e di quelle di controllo: nel primo caso si tende a minimizzare i ritardi, nel secondo si cerca di risparmiare sulle risorse e di evitare gli sprechi.

1.2 Motivazioni e obiettivi

È evidente, da quanto detto finora, che il testing è un processo molto costoso. In questo contesto, una delle attività più onerose del testing è la progettazione, ovvero la fase in cui si iniziano a definire quelli che poi diventeranno, nella fase di specifica, i casi di test. Un caso di test ha l'obiettivo di esercitare il sistema su uno specifico input.

Esistono in letteratura varie tecniche che permettono di generare in maniera automatica i casi di test, riducendo notevolmente i costi del testing. L'obiettivo è quello di testare il sistema, coprendo quanto più possibile determinati elementi, definiti in base al criterio di copertura scelto. Uno dei problemi che sorge quando si parla di generazione automatica dei casi di test è il *problema del costo dell'oracolo*: se, da un lato, la generazione dei dati di test è automatizzabile, la definizione dell'oracolo, nella maggior parte dei casi reali, resta un compito che dev'essere portato a termine dagli umani.

In questo contesto, l'obiettivo principale di questo lavoro è la definizione di una nuova metodologia per la generazione automatica di casi di test. Si vuole affrontare, inoltre, il problema dell'oracolo: al fine di ridurre il costo dell'oracolo, si è voluto fare in modo di ridurre il numero totale di casi di test generati, senza sacrificare la copertura.

Un altro obiettivo è la definizione di un nuovo tool per la generazione automatica di casi di test che implementi la nuova metodologia.

1.3 Risultati raggiunti

Il risultato principale di questo lavoro è l'introduzione di un nuovo approccio per la generazione automatica di casi di test, basato sui cammini linearmente indipendenti di McCabe [41]. Si è realizzato, inoltre, un nuovo tool per la generazione automatica di casi di test denominato OCELOT, ispirato ad altri lavori presenti in letteratura [44].

Il nuovo approccio consente di raggiungere un livello di copertura elevato, cercando, allo stesso tempo, di minimizzare il numero di casi di test, riducendo il costo dell'oracolo [25].

L'efficacia e l'efficienza della metodologia sono state valutate attraverso un caso di studio, confrontando la nuova metodologia con quella basata sulla generazione casuale dei casi di test. Dai risultati ottenuti si evince che la prima tecnica permette di ottenere livelli di copertura più altri rispetti alla seconda (mediamente del 7.1%), generando un numero di casi di test estremamente ridotto (54,9% in meno).

1.4 Organizzazione della Tesi

A questo capitolo introduttivo seguirà una panoramica sullo stato dell'arte per quanto riguarda il testing e, in particolare, il search-based software testing.

Nel capitolo 3 è descritta la nuova metodologia introdotta, basata sui cammini linearmente indipendenti di McCabe.

Nel capitolo 4 è presentato il tool (OCELOT), la sua architettura e il processo di esecuzione.

Nel capitolo 5 è presentato il caso di studio e la sperimentazione condotta, con i relativi risultati raggiunti.

Nel capitolo 6, infine, saranno tratte le conclusioni e si descriveranno gli sviluppi futuri di questo lavoro.

Capitolo 2

Stato dell'arte

Come osservato da Dijkstra, il testing può rivelare la presenza di malfunzionamenti in un programma, ma non dimostrarne l'assenza. Questo perché esistono dei problemi indecidibili che impediscono di dimostrare la correttezza di un programma (si veda il problema della fermata [61]). I metodi di progettazione dei test definiscono tecniche e criteri per ottenere buone approssimazioni del test ideale che garantirebbe la correttezza di un programma. In un processo di produzione software un obiettivo importante è quello di ottenere il miglior rapporto possibile fra la qualità del prodotto e il tempo e le risorse impiegate nelle attività di testing. Ottenere, quindi, buone metodologie e strategie per la conduzione dei test assume una importanza cruciale.

La progettazione dei test è l'attività di definizione di un insieme di casi di test. Un caso di test (test case) è un insieme di condizioni o variabili sotto le quali un tester determina se un sistema risponde correttamente o meno. In un caso di test sono specificati i dati in input, i risultati attesi ed una descrizione dell'ambiente di esecuzione. Si può determinare se un sistema ha superato un test attraverso il cosiddetto **oracolo**: questo indica il risultato atteso di un determinato caso di test, quindi come ci si aspetta che il sistema reagisca ad un determinato stimolo. Un insieme di casi di test è detto *test suite*.

Per la progettazione dei casi di test esistono due approcci: *black-box* e

white-box.

Testing Black-box Attraverso il testing funzionale si vogliono individuare i casi di test a partire dalle specifiche. Essendo basato sulle specifiche del programma e non sulle parti interne del codice, il testing funzionale è anche definito *black-box*. Il testing funzionale è tipicamente la tecnica di base per la progettazione dei casi di test, per diversi motivi. In prima istanza, il testing funzionale può (e dovrebbe) iniziare come parte del processo di specifica dei requisiti, e continuare attraverso ogni livello di progettazione delle specifiche e dell'interfaccia. Questa tecnica è l'unica che può essere applicata in maniera così ampia e precoce. Le tecniche di testing funzionale possono essere applicate a qualsiasi specifica del programma e ad ogni livello di granularità: dal testing di unità a quello di sistema. Il testing funzionale è efficace nella ricerca di alcune classi di problemi che di solito sfuggono alle tecniche white-box.

Testing White-box La struttura del software stesso è una preziosa fonte di informazioni per la selezione dei casi di test e per determinare se una test suite è sufficientemente accurata. A partire da una rappresentazione alternativa del codice sorgente, come *Control Flow Graph* o un altro modello, il testing strutturale consiste nel definire casi di test che riescano a coprire una certa quantità di elementi del modello scelto. Il testing può rivelare un difetto solo quando l'esecuzione dell'elemento difettoso causa un errore. Sulla base di questa semplice osservazione, un programma non è stato adeguatamente testato se alcuni dei suoi elementi non sono stati eseguiti. I criteri di testing strutturale sono definiti per particolari classi di elementi.

Come si può desumere da quanto detto finora, il testing è un processo altamente costoso. Proprio per questo, molto sforzo è stato fatto al fine di automatizzare quanto possibile alcune delle attività del testing. In particolare, si possono individuare due problemi principali che si è tentato di risolvere in letteratura:

- Generazione automatica dei casi di test

- Ottimizzazione della test suite

Nel primo caso, il problema consiste nel trovare una metodologia sistematica di generazione di casi di test; nel secondo caso, si vuole far sì che l'insieme di casi di test sia facilmente gestibile durante la fase di evoluzione del software. Gli obiettivi in gioco sono contrastanti: da un lato si vuole generare un insieme di casi di test che permetta di esercitare in maniera adeguata il sistema; dall'altro, però, si vuole che questo insieme sia piuttosto contenuto, in modo da ridurne il costo di esecuzione e manutenzione.

Per quanto riguarda la generazione automatica dei casi di test, si possono distinguere due categorie di tecniche utilizzate in letteratura: analisi statica e analisi dinamica.

Le tecniche di analisi statica sono basate sull'analisi della struttura interna del programma, senza eseguire il programma stesso. Ad esempio, l'esecuzione simbolica [32] [33] [13] consiste nell'assegnare valori simbolici alle variabili al fine di ricavare, attraverso metodi matematici, le condizioni necessarie per attraversare percorsi specifici nel programma [13] [34]. Anche se sono stati ottenuti risultati promettenti, queste tecniche hanno molti problemi [30]: la scalabilità, i predicati non lineari, i tipi di dati non primitivi, cicli, array, e così via. Tutto ciò che dipende dalla dinamicità di un programma non può essere in nessun modo affrontata dall'esecuzione simbolica e dalle tecniche di analisi statica in generale.

Le tecniche di analisi dinamica, al contrario, consistono nell'analizzare il comportamento del programma durante la sua esecuzione con un dato di input. Con la tecnica di generazione dei casi di test strutturali dinamici il programma in prova viene generalmente instrumentato ed eseguito con alcuni input ed i risultati dell'esecuzione vengono analizzati al termine dell'esecuzione. Duran e Ntafos [16] hanno indagato l'uso di controlli casuali come approccio dinamico. La tecnica prevede l'esecuzione del programma con input casuali e il controllo degli elementi eseguiti / coperti nella struttura del programma. La tecnica proposta non ha ottime prestazioni in quanto a numero di casi di test generati; in molti casi, tuttavia, riesce a raggiungere buoni livelli di copertura. Pertanto è generalmente utilizzato come riferimento per valutare le prestazioni di altri algoritmi di ricerca. Un modo per rendere

più veloce e più affidabile la generazione automatica di casi di test è quello di utilizzare algoritmi di ricerca. In letteratura, sono stati utilizzati diversi algoritmi [26], come ad esempio gli algoritmi genetici [31] [68] [58], la programmazione genetica [14] [38], il simulated annealing [10] e l'hill climbing [39].

In questo capitolo si procederà con un'analisi dettagliata delle tecniche dinamiche, più in particolare delle tecniche search-based, dopo aver fatto una breve introduzione sugli algoritmi di ricerca (meta-euristiche) più utilizzate.

2.1 Algoritmi search-based

Esistono molti casi in cui bisogna risolvere problemi di ottimizzazione in spazi continui. In alcuni di questi è possibile utilizzare algoritmi che permettono di raggiungere un risultato esatto. In altri casi, tuttavia, ci si trova davanti a problemi che non consentono l'applicazione di questi algoritmi. Essendo le soluzioni in uno spazio continuo, provare tutte le possibilità fino a trovare la migliore è impensabile. In questi contesti si utilizzano, generalmente, le cosiddette **meta-euristiche**, ovvero degli algoritmi in grado di ridurre in maniera significativa lo spazio di ricerca attraverso diverse assunzioni. Questi algoritmi non garantiscono il raggiungimento di una soluzione esatta, e potrebbero bloccarsi in un ottimo locale. In particolare, è possibile distinguere le meta-euristiche in due categorie: locali e globali.

2.1.1 Ricerca locale

La ricerca locale è una meta-euristica utile a risolvere problemi di ottimizzazione computazionalmente complessi. La ricerca locale può essere utilizzata per risolvere problemi che possono essere formulati come ricerca di una soluzione (tra un insieme di soluzioni candidate) che massimizzi (o minimizzi) un determinato criterio. Si definisce, in maniera preventiva, uno spazio delle soluzioni e gli algoritmi di ricerca locale applicano cambiamenti locali, spostandosi da una soluzione all'altra, finché non trovano una soluzione che

sembra essere ottimale. Uno degli algoritmi di ricerca locale più utilizzati è l'hill climbing.

Hill climbing L'algoritmo è molto semplice nella sua formulazione basilare (algoritmo 1). Si parte da una soluzione iniziale e ci si sposta nella direzione che permette di aumentare maggiormente la funzione obiettivo finché non si arriva ad un ottimo (ovvero una soluzione tale che la funzione obiettivo calcolata su tutti i vicini è minore rispetto alla funzione calcolata sulla soluzione trovata).

Data: Problema
Result: nodo
$nodo = CreaNodo(StatoIniziale(Problema))$;
$vicino = \arg\max_{n \in Successori(nodo)} Obj(n)$;
while $Valore(vicino) \geq Valore(nodo)$ **do**
    $nodo = vicino$;
    $vicino = \arg\max_{n \in Successori(nodo)} Obj(n)$;
end

Algoritmo 1: Algoritmo Hill climbing basilare, relativo ad un problema di massimizzazione

Un problema dell'hill climbing (e di tutti gli algoritmi di ricerca locale in generale) è che non è robusto in caso di presenza di ottimi locali. Questi algoritmi, infatti, possono facilmente trovare soluzioni che non sono ottimali, ma sub-ottimali. Per questa ragione esistono varianti dell'hill climbing che riducono il rischio di trovare ottimi locali, come, ad esempio, l'hill climbing stocastico o l'hill climbing con riavvii casuali [55]. Aggiungendo una componente casuale all'algoritmo si riesce a mitigare il problema precedentemente evidenziato [55].

2.1.2 Ricerca globale

La ricerca globale comprende diverse meta-euristiche che, a differenza della ricerca locale, si concentrano su tutto lo spazio delle soluzioni, esplorandolo sia in ampiezza sia in profondità, al fine di trovare soluzioni ottime e di evitare,

per quanto possibile, gli ottimi locali. Esistono diverse meta-euristiche, tuttavia le più diffuse ed utilizzate sono simulated annealing e algoritmi genetici. Verranno di seguito approfonditi questi ultimi.

Algoritmi genetici Gli algoritmi genetici sono basati sulla teoria evoluzionistica di Darwin. L'idea di base degli algoritmi genetici è quella di simulare i processi naturali necessari per l'evoluzione, in particolar modo quelli che seguono i principi della sopravvivenza degli essere viventi più adatti. Essi rappresentano un utilizzo intelligente di una ricerca casuale in uno spazio di ricerca definito per risolvere un problema.

Holland introdusse per primo gli algoritmi genetici [23]. L'algoritmo genetico di Holland è un metodo per passare da una popolazione iniziale di cromosomi ad una nuova, più adatta all'ambiente, usando il meccanismo di selezione naturale e gli operatori genetici di crossover, mutazione ed inversione (attualmente poco utilizzata).

Un algoritmo genetico tipicamente parte con una popolazione di soluzioni (cromosomi) scelte a caso e, attraverso un processo di ricombinazione e operazioni di mutazione, gradualmente evolve la soluzione avvicinandola alla soluzione ottima. L'algoritmo, in ogni caso, non garantisce il raggiungimento di una soluzione ottima. Il primo passo è la *selezione* delle soluzioni nella popolazione corrente che saranno utilizzate come genitori nella generazione successiva di soluzioni. La selezione richiede che le soluzioni vengano valutate in base al fitness: soluzioni che sono più vicine alla soluzione ottima sono giudicate con un valore di fitness più alto rispetto alle altre. Si usa una funzione di fitness (equivalente dalla funzione obiettivo in altri contesti) per valutare le soluzioni. Alla fine del processo, alcune soluzioni verranno mantenute, altre verranno scartate, in base al valore di fitness ottenuto. La ragione di questa scelta è che si assume che una buona soluzione sia composta da buoni componenti (geni). Scegliere, come soluzioni, genitori con questi geni, aumenta la possibilità che i loro figli ereditino tali geni e, di conseguenza, siano più vicini all'obiettivo. Anche se la selezione è direzionata verso le soluzioni migliori, i membri peggiori della popolazione hanno ancora la possibilità di essere selezionati come genitori, seppur con probabilità minore; anche una

soluzione con fitness basso, infatti, può avere dei geni che possono portare beneficio all'intera popolazione.

Il secondo passo consiste nel combinare i le soluzioni selezionate nella fase precedente e nel generare una nuova popolazione. Questo meccanismo imita la riproduzione degli esseri viventi. Per fare ciò, si usa un operatore di *crossover*, attraverso il quale, per ogni coppia di cromosomi, si scelgono dei punti di divisione e si scambiano alcuni geni.

Il terzo passo, infine, consiste nell'aggiungere una componente casuale all'algoritmo, che permetta di evitare, in parte, che l'algoritmo converga verso un ottimo locale. In questa fase si applica un operatore di *mutazione* che, con una certa probabilità (generalmente bassa) modifica alcuni geni casualmente. A questo punto il ciclo riparte dalla selezione, in modo da poter generare una nuova popolazione ed evolvere le soluzioni.

Evoluzione, selezione, ricombinazione e mutazione possono essere applicate molte volte in un algoritmo genetico; esse, quindi, devono essere il più semplici possibile. La decisione principale durante lo sviluppo di un algoritmo genetico riguarda la rappresentazione delle soluzioni ed il metodo di valutazione. Altre decisioni riguardano la taglia della popolazione, la frequenza di ricombinazione ed il tipo di sostituzione della popolazione. Gli algoritmi genetici hanno trovato un vasto campo di utilizzo. Il loro successo dipende, essenzialmente, dalla loro semplicità e, allo stesso tempo, dalla loro capacità di scoprire buone soluzioni in tempi rapidi per problemi di grandi dimensioni. Gli algoritmi genetici sono utili ed efficienti quando:

- Lo spazio di ricerca è grande, complesso e di difficile comprensione
- La conoscenza del dominio è scarsa

L'algoritmo in sé è, in definitiva, molto semplice e lineare (si veda l'algoritmo 2). La definizione della rappresentazione (e, in alcuni casi, degli operatori) è, invece, un compito che dipende strettamente dal problema specifico da risolvere.

Data: $Popolazione_0, Fitness, Tempo$
Result: $\arg\max_{s \in S} Fitness(s)$
$S = Popolazione_0$;
while $\exists s \in S$ *t.c.* $Fitness(s)$ *è abbastanza alta O* $tempo() \geq Tempo$ **do**
    $S = Seleziona(S)$;
    $S = Crossover(S)$;
    $S = Muta(S)$;
end

Algoritmo 2: Algoritmo genetico basilare, relativo ad un problema di massimizzazione

2.2 Generazione di casi di test con algoritmi search-based

La Search Based Software Engineering [24] ha l'obiettivo di ri-formulare alcuni problemi tipici dell'ingegneria del software come problemi di ottimizzazione, quindi di ricerca. In questo modo è possibile trovare soluzione ai problemi in maniera automatica attraverso meta-euristiche come quelle descritte nella sezione 2.1. In questa macro-area di ricerca si possono distinguere varie sotto-categorie, in base alla fase di sviluppo presa in considerazione:

- Project Management: è possibile prendere alcune decisioni in maniera automatica, riducendo l'effort del project manager (ad esempio, la definizione dello schedule) [2] [3] [20].
- Design: si cerca di definire un'architettura ottimale, che massimizzi la coesione e minimizzi l'accoppiamento delle componenti [52].
- Testing: si possono affrontare problemi tipici del testing (generalmente NP-C o NP-Hard) per trovare soluzioni sub-ottimali.
- Manutenzione: si utilizza la programmazione genetica per modificare direttamente il codice al fine di risolvere un problema [48] [64].
- Ottimizzazione: si ottimizza il codice, come nel caso precedente, attraverso la programmazione genetica [37].

In seguito verrà fatta una panoramica sullo stato dell'arte per quanto riguarda l'area del Search Based Software Testing.

Il problema della generazione automatica dei casi di test consiste nel generare, dato un programma P, un insieme di dati di test $\{ TC_1, ..., TC_n \}$ tale da massimizzare la copertura di un determinato elemento su P. I dati di test possono essere espressi come una lista di parametri, variabili d'ambiente o, in generale, come l'input da dare a P per eseguire un singolo caso di test. Per definire i casi di test è necessario specificare l'oracolo, ovvero il risultato atteso da P dato un input TC_i: in alcuni casi, se si hanno a disposizione pre e post condizioni del programma da testare, è possibile automatizzare la generazione dell'oracolo [25]; nella maggior parte dei casi reali, tuttavia, queste tecniche hanno poco spazio per essere applicate [25], e si demanda ad operatori umani il compito di determinare gli oracoli a partire dalle specifiche.

Gli approcci proposti in letteratura possono essere divisi nelle categorie mostrate in figura 2.1. Nel lavoro di Watkins [62] si prova a raggiungere la copertura totale dei percorsi. La funzione obiettivo penalizza gli individui che coprono percorsi già coperti. Il problema di questo e di tutti gli approcci cosiddetti "coverage-oriented" è che falliscono nel guidare la ricerca verso obiettivi nuovi, non ancora coperti.

In seguito si approfondiranno gli approcci *structure-oriented*: questi, attraverso la filosofia "divide et impera", permettono di raggiungere la copertura totale. Si procede con una ricerca distinta per ogni elemento richiesto dal criterio di copertura, fino a coprirli tutti (quando è possibile). In genere, gli approcci structure-oriented possono essere distinti in approcci *control-oriented*, approcci *branch-distance-oriented* e approcci misti (figura 2.1).

2.2.1 Approcci control-oriented

Negli approcci *control-oriented* la funzione obiettivo è in genere definita in termini di nodi che devono essere eseguiti al fine di testare l'elemento del programma desiderato. Jones et al. [31] hanno considerato come funzione obiettivo la differenza tra il numero di istruzioni veramente eseguite e di quelle

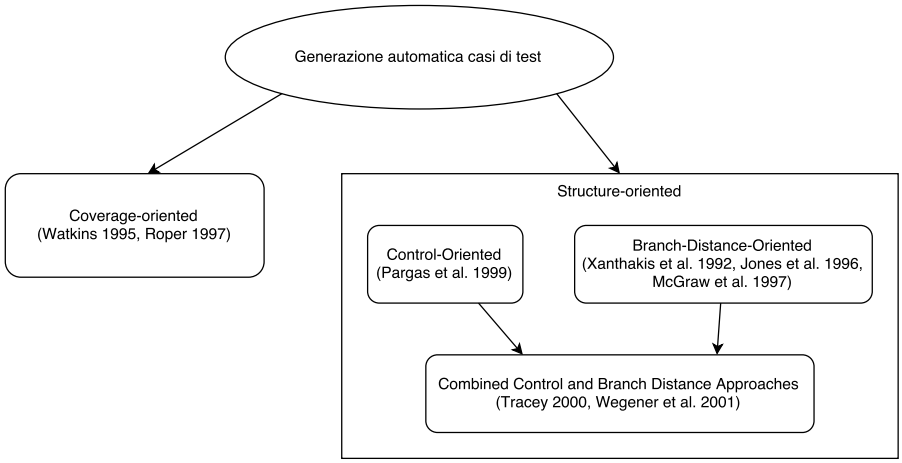


Figura 2.1: Classificazione delle tecniche per la generazione automatica di casi di test.

desiderate. Pargas et al. [51] hanno invece considerato una funzione obiettivo che misuri il numero di elementi del programma eseguiti, intesi come quelli eseguiti con una specifica dati di input. Così, hanno usato la *statement* e la *branch coverage* analizzando la struttura del Control Flow Graph durante l'esecuzione del programma. Come sottolineato da McMinn [43], la misura utilizzata da Pargas et al. è equivalente al numero di rami critici evitati con successo dai dati di input. Un ramo critico per un dato nodo di destinazione è un arco del Control Flow Graph che porta il percorso di esecuzione lontano dal nodo di destinazione. Se il flusso di controllo viene fatto scendere in un ramo critico, non vi è alcuna possibilità che il target sia raggiunto. Uno svantaggio delle funzioni obiettivo degli approcci control-oriented consiste nel non aver alcuna guida durante il processo di ricerca [31] [51]. Tutti i dati di ingresso che eseguono gli stessi nodi nel Control Flow Graph sono valutati come uguali dalla funzione obiettivo.

2.2.2 Approcci branch-distance-oriented

Negli approcci *branch-distance-oriented* la funzione obiettivo è tipicamente definita in termini di predicati del programma. Tracey et al. [60] [59] hanno

proposto una serie di funzioni obiettivo per i predicati relazionali ispirati alle funzioni obiettivo originali definite da Korel [35]. In generale, per ogni condizione nel Control Flow Graph la funzione di *branch distance* prende in input un insieme di valori di ingresso I, e valuta le funzioni obiettivo sulla base dei valori correnti di I. Questo tipo di funzione di fitness va bene per espressioni contenenti numeri e valori booleani. Per altri tipi di espressioni, ad esempio un confronto di uguaglianza tra oggetti o stringhe, sono state definite in letteratura funzioni di *branch distance* più complesse [45] [6] [1].

Infine, dato un determinato percorso obiettivo e un insieme di valori di ingresso I, la funzione obiettivo finale è misurata come la somma di tutti i valori di *branch distance* incontrati durante l'attraversamento del percorso dovuto all'esecuzione del programma con i dati di ingresso I. Data la *branch distance* come funzione di fitness, un algoritmo di ricerca viene eseguito per trovare i dati di input che soddisfino un elemento target nel Control Flow Graph almeno una volta. Il processo di ricerca continua fino a quando tutti gli elementi sono attraversati, secondo un criterio di copertura specifico, ovvero quando i valori di copertura massimi non possono essere ulteriormente migliorati. Xanthakis et al. [65] hanno usato gli algoritmi genetici per generare casi di test per i percorsi nel Control Flow Graph non coperti dalla ricerca casuale. In questo lavoro la selezione del percorso da considerare durante il processo di ricerca viene eseguita dagli sviluppatori. Jones et al. [31] hanno risolto il problema della selezione di un percorso specifico considerando la *branch distance* e calcolando la funzione di fitness come la *branch distance* del branch desiderato. Uno dei principali problemi degli approcci *branch-distance-oriented* riguarda la presenza del problema della bandiera nel caso in cui le condizioni abbiano un valore booleano. Per questo tipo di predicato non esiste una una guida (gradiente) per gli algoritmi di ricerca poiché la *branch distance* può assumere solo due valori differenti. Per alleviare il problema della bandiera diversi autori hanno suggerito di utilizzare trasformazioni [5] o funzioni di fitness più sofisticate [9].

La tabella 2.1 mostra come è possibile calcolare la *branch distance* in base al tipo di predicato.

Predicato	**Distanza**
a == b	se $a = b$, 0; altrimenti $abs(a - b)$
a != b	se $a \neq b$, 0; altrimenti K
a < b	se $a < b$, 0; altrimenti $a - b + K$
a <= b	se $a \leq b$, 0; altrimenti $a - b + K$
a > b	se $a > b$, 0; altrimenti $b - a + K$
a >= b	se $a \geq b$, 0; altrimenti $b - a + K$
a (booleano)	se a è vero, 0; altrimenti K
a || b	$\min bd(a), bd(b)$
a && b	$bd(a) + bd(b)$
!a	si propaga la negazione nell'espressione a

Tabella 2.1: Calcolo della branch distance per vari tipi di predicato

2.2.3 Approcci misti

Per superare le limitazioni dei metodi precedenti, Wegener et al. [63] hanno proposto un approccio combinato che unisce gli approcci *branch-distance-oriented* e *control-oriented* in una sola funzione obiettivo. L'approccio risultante è stato ampiamente utilizzato in letteratura per la generazione di casi di test [5] [58] [21]. La funzione obiettivo è definita unendo due componenti: *branch distance* e *approach level*.

La *branch distance* indica, dato un nodo-condizione del Control Flow Graph, quanto dev'essere modificato l'input affinché quella condizione sia valutata come negata. Se, ad esempio, dato un input $\{X_1, X_2, ...X_n\}$ una determinata condizione viene valutata come vera, la *branch distance* indica quanto devono essere modificate le variabili $\{X_1, X_2, ..., X_n\}$ affinché la condizione diventi falsa.

Il concetto di *approach level* dipende maggiormente dal caso specifico, ma, in generale, indica una misura approssimativa di quanto il percorso eseguito sia distante dall'obiettivo definito.

Wegener et al. [63] definiscono quattro tipi diversi di funzioni-obiettivo per quattro casi specifici, vale a dire:

- Copertura di un singolo nodo
- Copertura di un singolo percorso
- Copertura di una sequenza di nodi

- Copertura di una sequenza di nodi-percorso

2.2.3.1 Funzione di normalizzazione

La *branch distance* è generalmente normalizzata prima di essere utilizzata insieme all'*approach level* come funzione obiettivo. Questo perché la *branch distance* può assumere valori arbitrariamente alti, rendendo trascurabile, nella maggior parte dei casi, l'*approach level*, sebbene questo sia un indicatore ben più importante nel determinare quanto si è distanti dalla soluzione ricercata [4].

Una caratteristica desiderabile della *branch distance* normalizzata è che assuma valori compresi tra 0 (incluso) e 1 (escluso). Generalmente si esclude il valore 1 poiché una soluzione che ha *approach level* migliore rispetto ad un'altra, per quanto quest'ultima possa avere una *branch distance* buona non dev'essere mai preferita, quindi la somma dei due indicatori deve essere tale che il primo valore sia minore del secondo.

Dato che il valore massimo che può assumere la *branch distance* non è noto a-priori generalmente, è necessario utilizzare funzioni di normalizzazione non standard. In letteratura una delle funzioni più utilizzate è stata definita da Baresel [7]:

$$\omega_0(x) = 1 - \alpha^x$$

dove α è una costante maggiore di 1 e x indica il valore da normalizzare. Un buon valore per la costante α è 1.001 [7].

Arcuri [4] propone una nuova funzione di normalizzazione, ω_1, che ha un costo computazionale minore ed è meno soggetta ad errori di arrotondamento. La funzione è definita come segue:

$$\omega_1(x) = \frac{x}{x + \beta}$$

dove β è una costante maggiore di 0. Arcuri [4] indica che un buon valore per β è 1, e dimostra che la funzione ω_1 permette di ottenere, empiricamente,

risultati migliori, almeno nel caso in cui il criterio di copertura è costituito dai branch.

2.2.4 Tool esistenti

Nonostante il crescente interesse nel SBST e l'introduzione di differenti approcci per la generazione di casi di test, gli strumenti a disposizione dei ricercatori per eseguire testing strutturale basato sulla ricerca rimangono pochi. In letteratura sono descritti vari tool, quali TESTGEN [34], QUEST [11] e GADGET [46]. Questi strumenti sono datati, e, inoltre, non sono disponibili né il codice sorgente né gli eseguibili. Si tratta di tool creati allo scopo specifico di effettuare casi di studio per gli articoli in cui sono introdotti. Uno dei tool più citati è IGUANA [44], un tool per la generazione automatica di casi di test che utilizza algoritmi genetici. La funzione di fitness è calcolata attraverso la combinazione di *branch distance* e *approach level*. L'*approach level* è calcolato come il numero di nodi con strutture di controllo che devono essere soddisfatte per eseguire il target in esame. Con *branch distance* invece si esprime quanto è distante il vettore in input dal soddisfare la condizione del predicato nel quale il flusso di controllo diverge dalla direzione desiderata che conduce al target. Ovviamente per diversi tipi di predicati sono previsti differenti modi per calcolare questa distanza. La figura 2.2 mostra un modello del funzionamento di IGUANA.

Il tool necessita di un'infrastruttura di base che permetta il parsing del codice, con la relativa estrazione di un *Control Dependency Graph* e del Control Flow Graph, usati per capire quali sono i nodi delle strutture di controllo da attraversare per raggiungere i vari target individuali. Altro step fondamentale è quello dell'instrumentazione del codice. Tale processo va a rimpiazzare ogni condizione su un nodo condizione con una chiamata ad una funzione `node`. Tale funzione necessita di due parametri: il primo indica l'ID del nodo nel control flow graph mentre il secondo rappresenta la condizione booleana, mantenendo dunque il comportamento atteso. Per tracciare l'esecuzione di un nodo condizione e calcolare i valori necessari per le funzioni di fitness ci si avvale del processo di instrumentazione. Se il flusso di controllo diverge

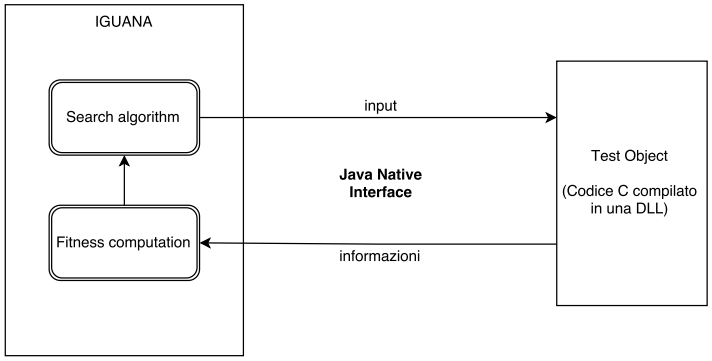


Figura 2.2: Modello di funzionamento del tool IGUANA

dal target selezionato vengono calcolate le metriche necessarie a definire la funzione di fitness, ovvero *approach level* e *branch distance*.

Un tool più recente è stato introdotto da Lakhotia et al. [36]. AUSTIN è uno strumento per la generazione di dati di test strutturati per i programmi C. AUSTIN utilizza tre algoritmi di ricerca: ricerca casuale, ricerca hill climbing, e ricerca hill climbing con esecuzione simbolica. Ad esempio, combina un semplice algoritmo hill climbing con valori d'input di tipo intero e in virgola mobile con un insieme di regole per input di tipo puntatore. È stato concepito come uno strumento di test di unità per i programmi C che considera ogni unità come una funzione da testare insieme a tutte le funzioni raggiungibili da tale funzione. AUSTIN può essere utilizzato per generare un insieme di dati di input per una data funzione che permettano di raggiungere un certo livello di copertura. La ricerca è guidata dalle funzioni obiettivo introdotte da Wegener et al. [63].

2.3 Ottimizzazione di test suite

In generale, la risoluzione di problemi di ottimizzazione di test suite prevede:

- la scelta di un criterio di test

- l'utilizzo di una tecnica di ottimizzazione per selezionare/ordinare i casi di test sulla base del criterio scelto

Ad esempio, criteri ampiamente utilizzati sono la copertura del codice [54], le modifiche subite dal programma [57], il costo di esecuzione [18], le informazioni storiche sui guasti [66] e così via. Hemmati et al. [28] suggeriscono di usare la diversità dei casi di test come criterio per ottimizzare la selezione dei casi di test. La combinazione di più criteri, spesso contrastanti, può essere più utile rispetto ai singoli criteri [8] [29] [66] [67] [56] per il testing di regressione. In seguito alla scelta dei criteri in un paradigma multi-criterio, deve essere utilizzato un algoritmo di ottimizzazione per selezionare/ordinare i casi di test. La ricerca esaustiva dei sottoinsiemi di test suite è praticamente impossibile. Per affrontare il problema sono stati utilizzati, quindi, algoritmi di approssimazione.

Iniziative mirate a ridurre il costo del testing di regressione includono minimizzazione della test suite, selezione dei casi di test, e prioritizzazione dei casi di test. Questi approcci sono generalmente definiti come approcci di ottimizzazione di test suite per il testing di regressione.

L'obiettivo del problema di **minimizzazione della test suite** consiste nel ridurre la dimensione dell'insieme dei casi test eliminando i casi di test ridondanti rispetto ai criteri di copertura.

La **prioritizzazione dei casi di test** ha lo scopo di ordinare i casi di test per ottimizzare alcune proprietà. Essa comporta l'esecuzione dei casi di test in un dato ordine e termina il processo di testing quando viene soddisfatta una condizione definita dallo sviluppatore [68].

La **selezione dei casi di test** ha l'obiettivo di selezionare un sottoinsieme della test suite iniziale per testare le modifiche apportate al software, ovvero per verificare se le parti non modificate di un programma funzionano correttamente dopo che varie modifiche hanno coinvolto altre parti.

Minimizzazione della test-suite Stabilito un criterio di copertura, si indichi con $Cov(S) = \bigcup_{T_i \in S} T_i$ l'insieme di elementi del Control Flow Graph coperti grazie ad una generica test suite S. Data una test suite $S_1 = \{T_1, ..., T_n\}$, una test suite *minimale* S_M di S_1 è tale che:

1. $S_M \subseteq S_1$
2. $Cov(S_M) = Cov(S_1)$
3. $\nexists S_{M2} \neq S_M \subseteq S_1$ t.c. $|S_{M2}| < |S_M|$.

Il problema in questione è un caso specifico del problema noto come "Set Cover": l'insieme universo è $U = Cov(S_1)$ e si ha una collezione di sottoinsiemi di S_1 tra cui scegliere il più piccolo tale che $Cov(S_M) = \bigcup_{T_i \in S_M} T_i = Cov(S_1) = U$. Per questo motivo, il problema della minimizzazione della test suite è **NP-Completo**.

Selezione dei casi di test Stabilito un criterio di copertura, si indichi con $Cov(S) = \bigcup_{T_i \in S} T_i$ l'insieme di elementi del Control Flow Graph coperti grazie ad una generica test suite S e con $Cost(S) = \sum_{T_i \in S} Cost(T_i)$ il tempo di esecuzione totale della test suite. Data una test suite $S_1 = \{T_1, ..., T_n\}$, il problema della selezione dei casi di test consiste nel trovare un insieme di casi di test S_S tale che:

- $S_S \subseteq S_1$
- $\nexists S_{S1} \neq S_S \subseteq S_1$ t.c. $Cov(S_{S1}) > Cov(S_S) \wedge Cost(S_{S1}) < Cost(S_S)$

Il problema della selezione dei casi di test è un caso specifico del problema "Weighted Set Cover": l'insieme universo è $U = Cov(S_1)$ e si ha una collezione di sottoinsiemi di S_1 tra cui scegliere quello a costo minimo tale che $Cov(S_M) = \bigcup_{T_i \in S_M} T_i = Cov(S_1) = U$. Per questo motivo, il problema della selezione dei casi di test è **NP-Completo**.

Prioritizzazione dei casi di test Stabilito un criterio di copertura, si indichi con $Cov(S) = \bigcup_{T_i \in S} T_i$ l'insieme di elementi del Control Flow Graph coperti grazie ad una generica test suite S e con $Fault(S) = \sum_{T_i \in S} Fault(T_i)$ il numero di fault individuati dalla test suite. È opportuno precisare che il numero di fault che un caso di test può individuare è, ovviamente, non noto finché non si procede alla sua esecuzione: per questo motivo si utilizzano delle metriche e dei dati che sono generalmente correlati al numero di errori

che può individuare un caso di test, come il numero di fault individuati in passato. Data una test suite $S_1 = \{T_1, ..., T_n\}$, il problema della prioritizzazione dei casi di test consiste nel trovare un'ordinamento di S_1 che permetta di individuare il prima possibile il maggior numero di difetti. In alcuni casi è valutato anche il costo di esecuzione tra gli obiettivi della prioritizzazione (eseguire per primi i casi di test che individuano più difetti in meno tempo).

Algoritmi approssimati per l'ottimizzazione Come è stato sottolineato da Yoo e Harman [66] [67], la minimizzazione della test suite, la selezione dei casi di test e la prioritizzazione dei casi di test sono problemi fortemente legati. Gli algoritmi descritti in letteratura, dunque, sono molto spesso comuni ad entrambi i problemi.

La dimensione della test suite può essere ridotta cancellando i casi di test ridondanti rispetto ad alcuni criteri di copertura [27] [53]. Diverse euristiche sono state applicate per affrontare questo problema [27] [47] [12]. Harrold et al. [27] propongono l'uso dell'algoritmo greedy tradizionale per il problema del *set cover*. Questo algoritmo inizia con un sottoinsieme vuoto della suite di test e iterativamente aggiunge il caso di test che fornisce la maggiore copertura tra i casi di test rimanenti. Il processo continua fino a quando non viene raggiunta la massima copertura (algoritmo 3).

Considerando che il problema del *set cover* è il duale a quello dell'*hitting set minimo* [22], Chen e Lau [12] hanno proposto una variante dell'algoritmo greedy nota per essere un'euristica efficace per il *Set Covering Problem*. Anche Offutt et al. hanno trattato il problema di minimizzazione suite di test come *Set Covering Problem* [47], proponendo diverse varianti dell'approccio greedy utilizzando criteri di ordinamento diversi dall'approccio originale [47]. Tuttavia, un confronto empirico tra i diversi approcci greedy ha suggerito che nessuna delle tecniche è in grado di superare l'altra in maniera significativa.

Altri lavori basati sugli approcci greedy hanno considerato altri criteri di copertura rispetto a quelli strutturali utilizzati da Harrold et al. [27], Offutt et al. [47], e Chen et al. [12]. Ad esempio, Marrè e Bertolino [40] hanno formulato il problema della test suite minimization come il problema di trovare un insieme minimo ricoprente sul grafo delle decisioni. McMaster

e Memon [42] hanno proposto una tecnica di minimizzazione della suite di test basata alla copertura call-stack. Black et al. [8] hanno considerato un approccio che tenga conto di due criteri:

- La copertura del codice
- L'esperienza passata nel rilevare i guasti

Nel loro studio hanno combinato i due obiettivi applicando un approccio di somma pesata, e utilizzando la programmazione lineare intera per trovare sottoinsiemi, quindi riducendo il problema multi-obiettivo a un problema a singolo obiettivo.

In alcuni casi, sia minimizzazione sia prioritizzazione possono essere affrontati con algoritmi multi-obiettivo, grazie ai quali si vuole selezionare un sottoinsieme efficiente secondo Pareto della suite di test, sulla base di più criteri di testing [66][67]. Sampath et al. [56] hanno fornito un'analisi sul vantaggio della combinazione di più criteri per il testing di regressione, mostrando che i criteri combinati spesso superano i criteri individuali che li costituiscono.

Yoo e Harman [66][67] hanno considerato prima due e poi tre criteri contrastanti: la copertura del codice e il tempo di esecuzione nella formulazione a due obiettivi e hanno aggiunto le informazioni storiche sugli errori come terzo criterio nella formulazione a tre obiettivi. Essi hanno inoltre valutato diversi algoritmi di ottimizzazione per la ricerca di sottoinsiemi di test suite ottimali secondo Pareto: un algoritmo greedy addizionale e un algoritmo genetico multi obiettivo chiamato NSGA-II [15]. Il confronto empirico tra gli algoritmi genetici multi-obiettivo e gli algoritmi greedy non ha evidenziato un chiaro vincitore, e in alcuni casi gli algoritmi greedy hanno mostrato di trovare soluzioni migliori [66]. Inoltre, la combinazione tra questi due tipi di algoritmi non è sempre utile per raggiungere risultati migliori[67].

Panichella et al. [49] hanno dimostrato che l'ottimalità degli algoritmi genetici multi-obiettivo può essere notevolmente migliorata diversificando le soluzioni generate durante il processo di ricerca. In particolare, hanno introdotto un nuovo algoritmo, detto DIV-GA (DIVersity based Genetic Algorithm), che aumenta la diversità, iniettando nuovi individui ortogonali durante

il processo di ricerca. I risultati ottenuti su undici programmi hanno dimostrano che DIV-GA supera sia gli algoritmi greedy che gli algoritmi genetici multi-obiettivo tradizionali dal punto di vista dell'ottimalità. Inoltre, le soluzioni fornite da DIV-GA sono in grado di rilevare più errori rispetto agli altri algoritmi, mantenendo lo stesso costo di esecuzione del testing.

Data: S_1
Result: S_m
$C_1 = coverage(S_1)$;
$C_m = \varnothing$;
$S_m = \varnothing$;
while $C_m \neq C_1$ **do**
    $T_b = \arg\max_{T \in S_1 - S_m} coverage(\{T\})$;
    $S_m = S_m \cup T_b$;
    $C_m = coverage(S_m)$;
end

Algoritmo 3: Algoritmo greedy per l'ottimizzazione della test suite

Capitolo 3

Metodologia

L'obiettivo che si vuole raggiungere nella generazione automatica di casi di test *white-box* è la definizione, in maniera automatica, dei dati da dare in input ad un determinato programma P che si vuole testare, per far sì che si possa raggiungere la massima copertura possibile, in base ad un criterio di copertura stabilito a-priori. Gli approcci più utilizzati in letteratura sono quelli *structure-oriented*, che consistono nel dividere il programma da testare in modo tale da individuare diversi obiettivi intermedi da coprire al fine di raggiungere la copertura totale. Gli obiettivi intermedi possono essere, dunque, archi del *Control Flow Graph* (*branch coverage*), linee di codice (*statement coverage*), condizioni (*condition coverage*) e così via.

Per raggiungere questo scopo, è necessario, quindi, definire:

- Il criterio di selezione degli obiettivi da coprire.
- L'algoritmo di ricerca da utilizzare al fine di trovare un input che permetta di coprire un obiettivo intermedio.

In questo capitolo verrà presentata una nuova metodologia per la generazione automatica di casi di test. La principale novità introdotta consiste nell'utilizzo dei cammini linearmente indipendenti [41] come obiettivi intermedi da coprire al fine di raggiungere la copertura totale.

3.1 Introduzione

3.1.1 Rappresentazioni di un programma

Si consideri una funzione o una procedura da testare. Questa può essere rappresentata in vari modi. Innanzitutto può essere codificata in un linguaggio di programmazione: attraverso questa rappresentazione, la funzione/procedura può essere compresa dai programmatori. Nel momento in cui si vuole compilare il programma, questo viene rappresentato in diversi modi alternativi, affinché lo si possa tradurre in maniera corretta nel linguaggio oggetto, compreso dalla macchina. Un particolare modulo del compilatore, il parser, ha lo scopo di controllare la correttezza sintattica del codice sorgente e di creare un *Abstract Syntax Tree* (o AST), il quale costituisce una seconda rappresentazione utile del programma.

La radice dell'*Abstract Syntax Tree* indica l'intera procedura, ogni nodo intermedio indica un costrutto del programma (quindi una condizione, un ciclo e così via) mentre i nodi foglia indicano generalmente le variabili, i letterali e così via. La struttura dell'*Abstract Syntax Tree* dipende strettamente dal linguaggio di programmazione e dal parser. È possibile, infine, trasformare l'*Abstract Syntax Tree* in altre forme, attraverso cui è possibile fare analisi di tipo dinamico: la rappresentazione che sarà utilizzata per la generazione dei casi di test, in particolare, è quella del *Control Flow Graph* (o CFG). Un *Control Flow Graph* è un grafo che ha almeno un nodo iniziale e un nodo finale. Dal nodo iniziale parte soltanto un arco che va alla prima istruzione del programma. Generalmente ogni nodo rappresenta un'istruzione (o un blocco di istruzioni, nel caso in cui queste siano consecutive e si può inferire staticamente che verranno eseguite una dopo l'altra). Da ogni nodo parte un arco per ogni possibile strada che si può prendere in fase di esecuzione: un nodo condizionale, ad esempio, avrà due archi uscenti, uno che verrà percorso nel caso in cui la condizione risulta essere falsa, l'altro nel caso in cui è valutata come vera. Tutti i nodi che rappresentano istruzioni finali (dopo le quali non verrà eseguita nessun'altra istruzione), avranno un arco uscente che porta al nodo finale del *Control Flow Graph*.

La figura 3.3 mostra un esempio completo di come si passa dal codice sorgente, all'*Abstract Syntax Tree* e, successivamente, al *Control Flow Graph*. L'*Abstract Syntax Tree* (figura 3.1b) ha un nodo per ogni istruzione, per ogni espressione, e per ogni sotto-espressione, fino alle foglie, che contengono gli elementi che non possono essere ulteriormente scomposti, quindi identificatori, letterali e così via. Il *Control Flow Graph*, invece, (figura 3.1c), ha un nodo per ogni per ogni istruzione, mentre gli archi rappresentano come si può passare da un'istruzione all'altra.

Una singola esecuzione del programma, relativa ad uno specifico input, può essere rappresentata, in modo alternativo, come un percorso sul *Control Flow Graph*, che parte dal nodo iniziale e termina nel nodo finale. Al contrario, non tutti i percorsi sul *Control Flow Graph* rappresentano possibili esecuzioni del programma: è possibile, infatti, che un determinato percorso sia *infeasible*, quindi non percorribile. Questo può capitare nel caso in cui ci siano condizioni contraddittorie, insoddisfacibili o situazioni simili.

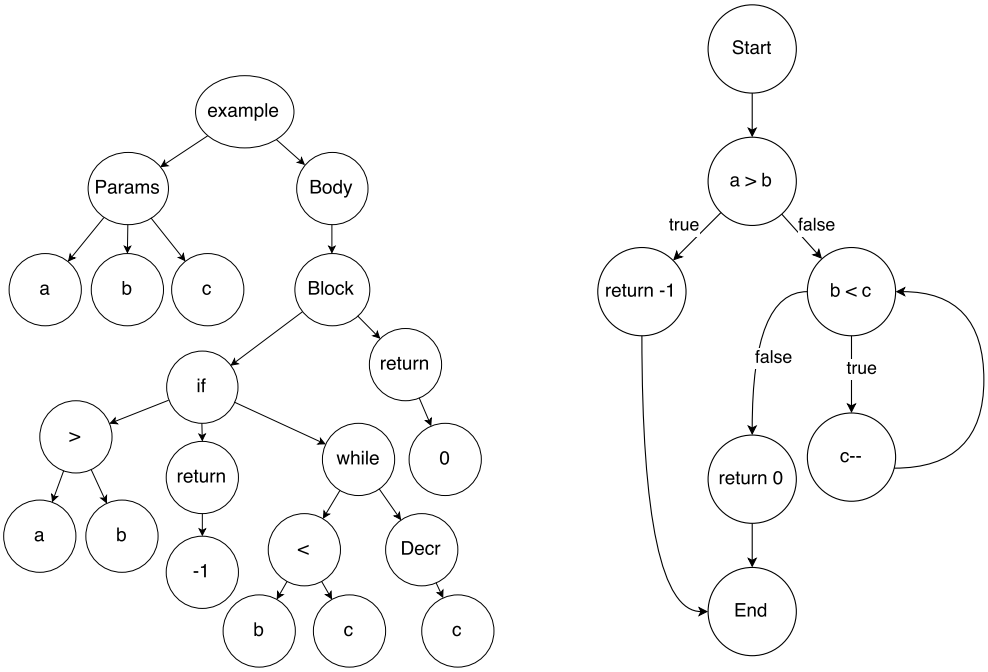
Si consideri l'esempio in figura 3.2. Il percorso $\langle (a < 10), (a > 10), (a{+}{+}) \rangle$ è, banalmente, non percorribile, perché la *path condition*, ovvero la condizione necessaria affinché sia percorso quel determinato cammino, è che la variabile a sia, allo stesso tempo, maggiore di 10 e minore di 10.

Costruzione del Control Flow Graph Si può costruire il *Control Flow Graph* in maniera ricorsiva attraverso la visita in profondità dell'*Abstract Syntax Tree*. Si parte dalle istruzioni semplici, quindi istruzioni di assegnazione, istruzioni di salto (`break`, `continue`, e così via) e si generano i primi nodi del *Control Flow Graph*. In maniera ricorsiva, quindi, finita la visita dei figli $n_1, n_2, ..., n_k$ di un dato nodo N dell'*Abstract Syntax Tree*, e costruiti i sottografi del *Control Flow Graph* $G_1, ..., G_k$ relativi a quei nodi specifici, è possibile procedere con la visita del nodo N stesso e costruire il sottografo G, sfruttando i sottografi $G_1, ..., G_k$ dei figli.

Ad esempio, si consideri lo snippet in figura 3.3. In primo luogo vengono creati i sotto-grafi relativi alle righe 2 (sottografo A) e 4 (sottografo B). Trattandosi di istruzioni semplici, i sotto-grafi A e B sono composti da un solo

```
1  int example(int a, int b, int c) {
2    if (a > b)
3      return -1;
4    else
5      while (b < c)
6        c--;
7
8    return 0;
9  }
10
```

(a)



(b) (c)

Figura 3.1: Esempio completo di passaggio dal codice (a), all'Abstract Syntax Tree (b) e, infine, al Control Flow Graph (c)

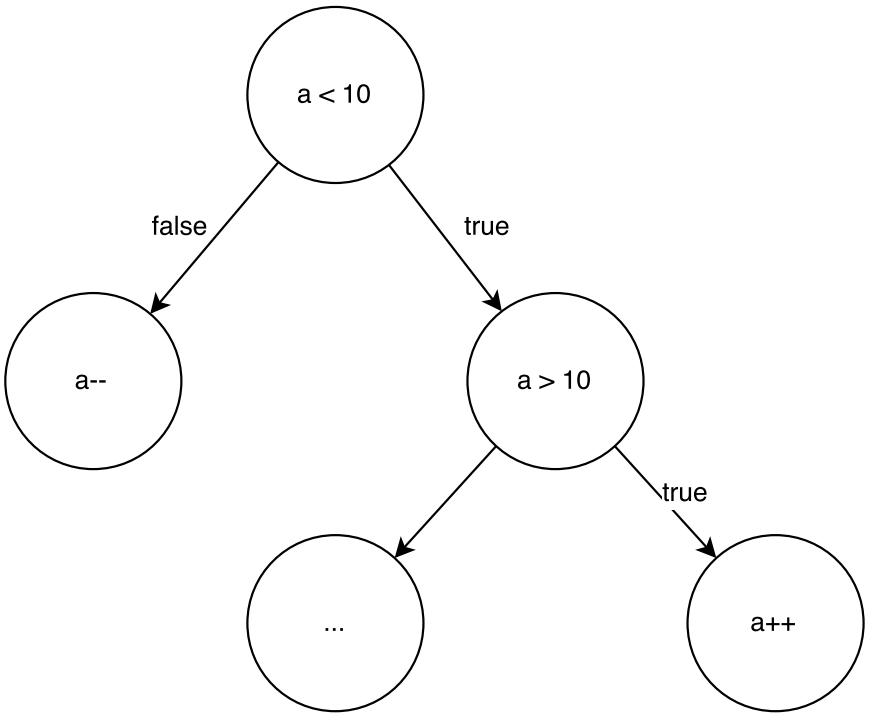


Figura 3.2: Porzione di *Control Flow Graph* con un percorso infeasible.

```
1 if (a > 10) {
2   b = 11;
3 } else {
4   b = 10;
5 }
```

Figura 3.3: Esempio di codice di cui si vuole calcolare il Control Flow Graph.

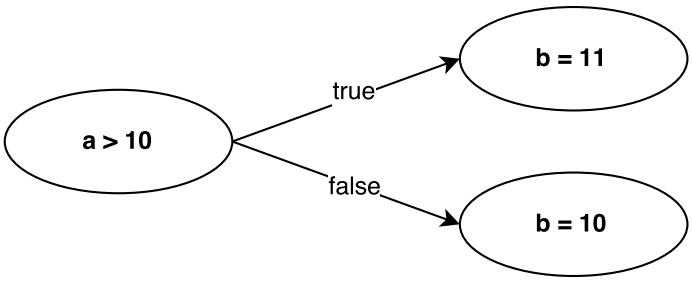


Figura 3.4: Control Flow Graph finale.

nodo ciascuno. Successivamente viene eseguita la visita del nodo dell'*Abstract Syntax Tree* relativo all'istruzione condizionale. Durante la visita, vengono creati gli archi dal nodo condizione (a > 10) ai nodi iniziali dei sottografi A e B precedentemente creati. La figura 3.4 mostra il risultato finale.

Il *Control Flow Graph* di un programma ha due nodi speciali: il nodo iniziale e il nodo finale. Dal nodo iniziale parte un solo arco, che porta alla prima istruzione del programma. Alla fine della generazione del *Control Flow Graph* di un programma, dev'essere aggiunto, a tutti i nodi che non hanno archi uscenti, un arco che porta al nodo finale. Tutti i cammini che rappresentano percorsi di esecuzione su un *Control Flow Graph* devono partire dal nodo iniziale e terminare nel nodo finale.

3.1.2 Obiettivi intermedi

Come accennato precedentemente, l'obiettivo della generazione automatica di casi di test è massimizzare la copertura in base ad un determinato criterio. Alcuni esempi di criteri di copertura possono essere *statement coverage* (copertura di tutte le linee di codice), *branch coverage* (copertura di tutti gli archi del *Control Flow Graph*), *path coverage* (copertura di tutti i cammini sul *Control Flow Graph*) e così via.

Generalmente il criterio di copertura più usato è la *branch coverage*, perché da un lato implica la *statement coverage*, dall'altro fa sì che tutte le condizioni siano valutate in tutti i modi possibili. Per questo, il criterio di copertura preso in considerazione nella metodologia presentata è, appunto, la *branch coverage*.

L'approccio *structure-oriented* (si veda la sezione 2.2 per un approfondimento) prevede la divisione del problema del raggiungimento della copertura totale in vari problemi più piccoli, seguendo l'approccio *divide et impera*: nel caso della copertura dei branch, si può dividere il problema della copertura di tutti gli archi del grafo in vari problemi, ognuno dei quali consiste nel coprire un singolo arco. In questo modo è possibile, alla fine, raggiungere la *branch coverage* totale.

Una problematica tipica della selezione dei casi di test attraverso un approccio *stucture-oriented* è la selezione degli obiettivi intermedi, attraverso la copertura dei quali si intende raggiungere un determinato livello di copertura totale. Un obiettivo intermedio da testare consiste in un qualsiasi elemento o porzione del *Control Flow Graph* o di un altro modello di rappresentazione del programma. Può trattarsi, ad esempio, di un nodo, di un arco, di un intero percorso, o anche di una combinazione di questi elementi.

Si può pensare di fare in modo che ogni caso di test sia composto semplicemente da:

- Una chiamata alla funzione da testare
- Un controllo che il risultato sia conforme all'oracolo (asserzione)

In questo modo ogni caso di test permette di coprire un singolo cammino del *Control Flow Graph* che parte dal nodo iniziale e termina nel nodo finale. Il percorso di esecuzione relativo ad uno specifico input del programma da testare potrebbe coprire o meno l'obiettivo intermedio. Se l'obiettivo definito non è un percorso, bensì un singolo nodo/arco, il percorso di esecuzione coprirà in maniera collaterale altri obiettivi.

3.1.3 Algoritmo di ricerca

Dato un particolare obiettivo intermedio da coprire, è necessario cercare l'input da dare al programma da testare che permetta di eseguire esattamente quell'elemento del *Control Flow Graph*. Un modo per fare ciò è attraverso gli algoritmi genetici. In questo caso è necessario definire:

- La rappresentazione delle soluzioni
- Gli operatori da utilizzare, quindi:
  - Operatore di selezione
  - Operatore di crossover
  - Operatore di mutazione

Rappresentazione delle soluzioni Si può pensare di ammettere, in una fase iniziale, soltanto programmi aventi parametri di tipo numerico. In questo modo, infatti, è possibile semplificare la rappresentazione delle soluzioni: un cromosoma avrà la forma di una collezione (array) di numeri reali (decimali a doppia precisione, o *double*). Anche se i parametri numerici possono essere di vari tipi (interi, interi non segnati, byte e così via), è sempre possibile tradurre un reale in qualsiasi altro tipo numerico, essendo il più generico (oltre ad essere il tipo numerico di base che richiede il maggior numero di byte in memoria). La tabella 3.1 mostra alcuni esempi di dichiarazioni di funzioni e la rappresentazione di una soluzione per ognuna di esse.

Funzione	**Esempio di cromosoma**
test1(int a, int b, int c)	$\langle 12, 32, 342 \rangle$
test2(double a, int b)	$\langle 122.343, 1 \rangle$
test3(char a, char b, char c, double d)	$\langle 0, 23, 4, 2.123 \rangle$
test4()	$\langle \rangle$

Tabella 3.1: Alcune dichiarazioni di funzioni ed esempi di cromosomi.

Operatori Si è scelto come operatore di *selezione* il cosiddetto "Tournament selection". Questo metodo consiste nel prendere, ad ogni iterazione, k individui della popolazione (tournament) e nell'assegnare ad ognuno di essi una probabilità di essere scelti pari a $p * (1 - p)^i$, dove i indica la posizione dell'individuo nel ranking di ogni singolo tournament (deciso in base al valore della funzione di fitness) e p indica la probabilità che sia scelto il primo individuo (quindi il migliore). Generalmente si assegna a p un valore inferiore ad 1, per ridurre il rischio di incorrere in ottimi locali.

Per quanto riguarda l'operatore di *crossover*, si può scegliere se dividere il cromosoma in un singolo punto oppure in più punti, al fine di combinare le soluzioni intermedie. Si è optato per la prima soluzione, la quale da un lato semplifica la realizzazione, dall'altro è una delle più utilizzate. Con una probabilità generalmente bassa, i cromosomi selezionati e incrociati possono subire una mutazione. Generalmente, dato lo spazio delle soluzioni, la mutazione permette ad un cromosoma di trasformarsi in un secondo cromosoma simile a quello originale. In particolare si utilizzerà un operatore di *mutazione* polinomiale, che modifichi la probabilità di mutazione in base al numero di iterazioni effettuate dall'algoritmo (generazioni). La probabilità tenderà a decrescere all'aumentare del numero di generazioni.

La figura 3.5 mostra un esempio di come sono applicati gli operatori di selezione, crossover e mutazione a una popolazione di sei individui.

Nuovo operatore di mutazione Associato all'operatore di mutazione polinomiale, sarà presente un secondo operatore. Questo muterà con una probabilità molto bassa ogni singolo gene del cromosoma in una delle costanti

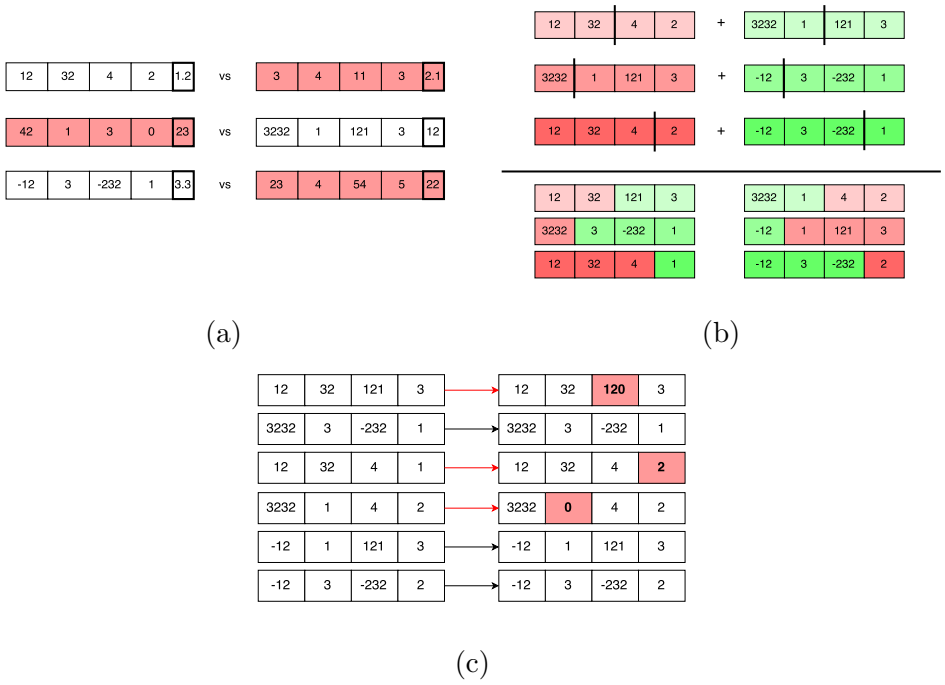


Figura 3.5: Operatori dell'algoritmo genetico. La figura (a) mostra l'operatore di selezione: in rosso i cromosomi selezionati in ogni *tournament*, nell'ultimo riquadro di ogni cromosoma il valore di fitness calcolato. La figura (b) mostra l'operatore di crossover: nella parte superiore, i cromosomi selezionati, con i relativi punti di crossover (linee nere verticali); in basso la nuova popolazione risultante dopo lo scambio delle parti. Il colore di sfondo indica il cromosoma di partenza. Nella figura (c) è mostrato un esempio di operatore di mutazione: le frecce rosse indicano i cromosomi che hanno subito mutazione e le celle con lo sfondo rosso i geni mutati.

```
1 if (a == 0.0)
2   a++;
3 else
4   a -= 12;
5
```

Figura 3.6: Esempio in cui è difficile coprire un branch

numeriche presenti nel programma da testare. Si consideri, ad esempio, il codice in figura 3.6. Sebbene si tratti un tipo intelligente di algoritmo casuale, un algoritmo genetico genera i cromosomi in maniera casuale, e sempre in maniera parzialmente casuale li accoppia e li muta. Per questo motivo, se si considera a come un parametro di tipo reale, sarà molto improbabile che l'algoritmo assegni ad a esattamente il valore 0. In questo caso l'operatore di mutazione potrebbe modificare qualche cromosoma in modo che a abbia esattamente il valore 0. Dato che il programma contiene anche la costante numerica 12, esiste una probabilità che l'operatore muti il cromosoma in modo che a assuma anche questo valore, il quale, al contrario del precedente, è inutile.

La probabilità di mutazione dovrebbe essere molto piccola, poiché questa tecnica da un lato potrebbe facilitare la copertura di un determinato obiettivo, ma dall'altro potrebbe introdurre rumore, se sono presenti nel programma costanti numeriche inutili, come 12 nell'esempio precedente. Questo avrebbe un effetto negativo e genererebbe cromosomi con valore di fitness basso.

3.1.4 Cammini linearmente indipendenti

La complessità ciclomatica di McCabe nasce dalla necessità di disporre di uno strumento di valutazione quantitativa da usare sui moduli di un sistema. Nel 1976 egli propose, dunque, una tecnica matematica ad hoc per misurare la complessità di un programma, basata sull'utilizzo del grafo del flusso di controllo dello stesso.

La misura di complessità permette di misurare e controllare il numero dei cammini all'interno di un programma. È semplice notare come, tuttavia, possa bastare un qualsiasi arco del grafo per generare un numero infinito di

cammini. Proprio per questo motivo, la misura di complessità sviluppata da McCabe [41] viene definita in termini di cammini-base che, combinati tra loro, permettono di generare qualunque tipo di percorso.

Una serie di definizioni e teoremi sulla teoria dei grafi sono necessari per comprendere al meglio la tecnica proposta. È utile specificare, innanzitutto, il concetto di connettività. Dato un grafo $G = (V, E)$, due vertici u, v si dicono *connessi* se esiste un cammino con estremi v e u. Se tale cammino non esiste, v e u sono detti sconnessi.

Per $i \in \{1, ..., k\}$ (k classi di equivalenza) sono definibili i sottografi $G_i = (V_i, E_i)$ come i sottografi massimali che contengono tutti gli elementi connessi tra loro, che prendono il nome di *componenti connesse* di G, la cui cardinalità spesso viene indicata con $\gamma(G)$.

Definizione 1. *Il numero ciclomatico $V(G)$ di un grafo G con n vertici, e archi e p componenti connesse è pari a:*

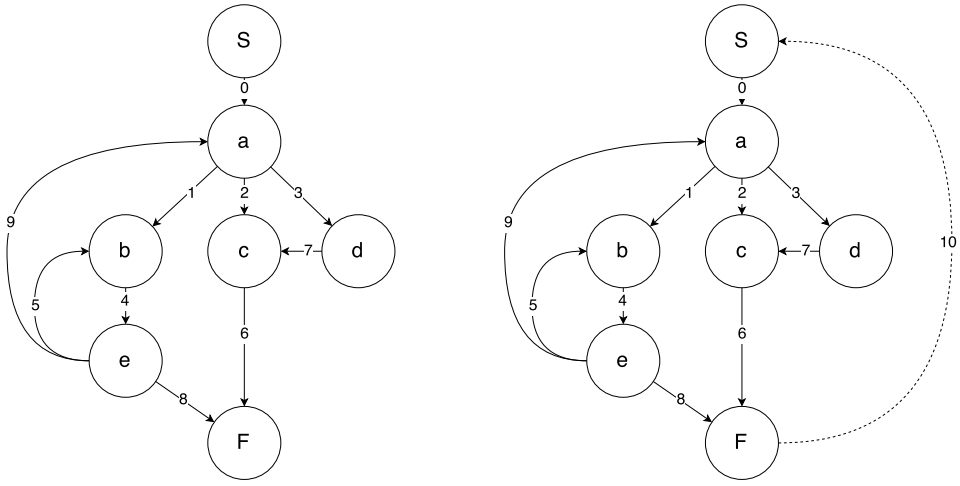
$$V(G) = e - n + p$$

Teorema 1. *In un grafo G fortemente connesso, il numero ciclomatico è uguale al numero massimo di cammini linearmente indipendenti.*

Il teorema 1 appena enunciato può essere applicato come segue. Dato il *Control Flow Graph* di un programma, è assumibile che ogni nodo possa essere raggiunto dal nodo di entrata e che ogni nodo è in grado di raggiungere il nodo di uscita. Ad esempio, si consideri il *Control Flow Graph* in figura 3.7a con un nodo iniziale S e un nodo finale F.

Si immagini che il nodo di uscita F sia collegato con un arco al nodo di entrata S. Tale arco corrisponde al numero 10, tratteggiato in figura 3.7b.

Con questo stratagemma il grafo è diventato *fortemente connesso*: esiste un cammino che collega ogni coppia di nodi distinti. Il teorema 1 diventa, quindi, applicabile a tutti i *Control Flow Graph*. Il numero massimo di cammini linearmente indipendenti nel grafo G è 9-6+2 = 5. Un esempio di circuiti indipendenti che si possono scegliere in G è il seguente:



(a) (b)

Figura 3.7: Control Flow Graph originale (a) e modificato con un arco che torna dal nodo finale al nodo iniziale (b)

$$B_1 = \{\langle S, a, b, e, F, S\rangle, \langle b, e, b\rangle, \langle a, b, e, a\rangle, \langle S, a, c, F, S\rangle, \langle S, a, d, c, F, S\rangle\}$$

Ne consegue che B_1 forma una *base* per costruire tutti i cammini di G e ogni cammino può essere espresso tramite una combinazione lineare dei percorsi appartenenti a B_1. Ad esempio, il cammino $\langle S, a, b, e, a, (b, e)^3, F\rangle$ è esprimibile nel seguente modo:

$$\langle a, b, e, a\rangle + 2\langle b, e, b\rangle + \langle S, a, b, e, F, S, a\rangle$$

Ad ogni elemento della base B_1 si può associare un vettore come specificato nella tabella 3.2

Ad esempio il path $\langle S, a, b, e, a, b, e, b, e, b, e, F, S\rangle$ corrisponde al vettore $[1, 2, 0, 0, 4, 2, 0, 0, 1, 1, 1]$.

Sempre in accordo con il teorema 1 esposto in precedenza, è possibile scegliere un set base di circuiti tale che ognuno di essi corrisponda esatta-

Percorsi	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\langle S, a, b, e, F, S\rangle$	1	1	0	0	1	0	0	0	1	0	1
$\langle b, e, b\rangle$	0	0	0	0	1	1	0	0	0	0	0
$\langle a, b, e, a\rangle$	0	1	0	0	1	0	0	0	0	1	0
$\langle S, a, c, F, S\rangle$	1	0	1	0	0	0	1	0	0	0	1
$\langle S, a, d, c, F, S\rangle$	1	0	0	1	0	0	1	1	0	0	1

Tabella 3.2: Vettori relativi ai percorsi linearmente indipendenti

mente ad un cammino sul *Control Flow Graph* che parte dal nodo iniziale e termina nel nodo finale. Il seguente set B_2 corrisponde ai percorsi base del programma rappresentato nel *Control Flow Graph*:

$B_2 = \{\langle S, a, b, e, F\rangle, \langle S, a, b, e, a, b, e, F\rangle, \langle S, a, b, e, b, e, F\rangle, \langle S, a, c, F\rangle, \langle S, a, d, c, F\rangle\}$

Essendo anche B_2 una base, una combinazione lineare dei cammini in B_2 genererà qualsiasi cammino possibile nel *Control Flow Graph*. Ad esempio, $\langle S, a, b, e, a, b, e, b, e, b, e, F\rangle = 2\langle S, a, b, e, b, e, F\rangle - \langle S, a, b, e, F\rangle$

Sia G un grafo di controllo di flusso di un programma e $V(G)$ la sua complessità ciclomatica, è possibile esporre brevemente alcune proprietà:

- $V(G) \geq 1$
- $V(G)$ corrisponde al numero massimo di cammini linearmente indipendenti in G
- L'inserimento e l'eliminazione di istruzioni funzionali in G non influenza $V(G)$
- G ha un solo cammino se e solo se $V(G) = 1$
- L'inserimento di un nuovo arco in G incrementa $V(G)$ di un'unità
- $V(G)$ è dipendente solamente dalla struttura decisionale di G

3.2 Generazione dei casi di test

L'approccio proposto si divide in 3 macro-passaggi, che saranno dettagliati in seguito:

- Generazione di casi di test per ogni cammino linearmente indipendente
- Generazione di casi di test per ogni arco non coperto
- Minimizzazione della test suite

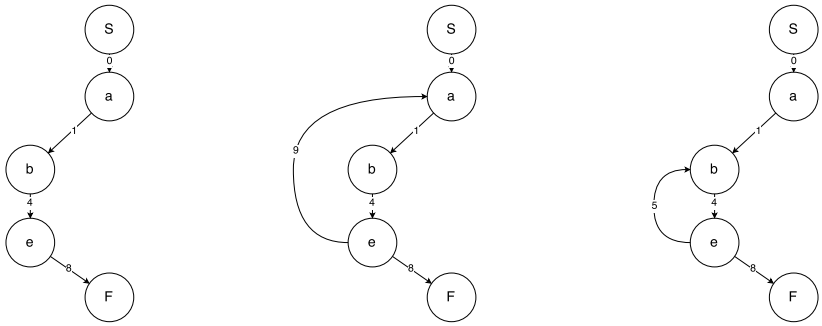
3.2.1 Copertura dei cammini linearmente indipendenti

Per prima cosa si vuole raggiungere la copertura totale dei branch del *Control Flow Graph* di un programma P tentando di eseguire esattamente i cammini linearmente indipendenti del *Control Flow Graph*. L'esecuzione dei cammini linearmente indipendenti, infatti, come visto nella sezione 3.1.4, garantisce la copertura totale dei branch.

La figura 3.8, ad esempio, mostra i cammini linearmente indipendenti sul *Control Flow Graph* dell'esempio 3.7. In questo caso basterebbe avere 5 casi di test distinti per raggiungere la completa *branch coverage*.

In primo luogo bisogna generare tutti i cammini a partire dal grafo. L'operazione è possibile grazie ad all'algoritmo delle baseline successive (algoritmo 4): si sceglie il cammino che contiene il maggior numero di punti di decisione, quindi di vertici del grafo con *outdegree* maggiore di uno. Questo cammino, detto *baseline*, è il primo percorso. A partire dalla *baseline*, si sceglie un punto di decisione e si segue un arco alternativo, in modo da includere il maggior numero di punti di decisione. A questo punto si modifica l'ultimo percorso trovato finché non si sono presi in considerazione tutti i punti di decisione; in questo caso si ritorna alla *baseline* e si cambia un'altra decisione. Tutto ciò finché non si analizzano tutti i punti di decisione del *Control Flow Graph*.

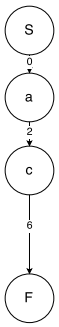
Una volta generati tutti i cammini linearmente indipendenti, si utilizza un algoritmo genetico (descritto nei dettagli nella sezione 3.1.3) al fine di trovare l'input che permetta di coprire il percorso desiderato. Più in particolare, si



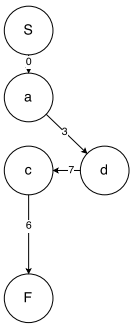
(a)

(b)

(c)



(d)



(e)

Figura 3.8: Rappresentazione grafica dei cammini linearmente indipendenti del grafo in figura 3.7a. Per ognuno dei grafi in figura, il cammino linearmente indipendente è quello che parte da S, finisce in F e attraversa almeno una volta e al massimo due volte ogni arco.

Data: CFG
Result: $\{P_0, ..., P_c\}$
P_0 = percorso con maggior numero di punti di decisione;
$C_0 = P_0$;
$i = 1$;
while *Non tutti i punti di decisione esaminati* **do**
  while *È possibile trovare altri percorsi a partire da* C_{i-1} **do**
    $P_i = C_{i-1}$;
    $C_i = P_i$;
    Cambia una decisione in P_i in modo da massimizzare il numero di punti di decisione in P_i;
    $i = i + 1$;
  end
  $C_{i-1} = P_0$;
end

Algoritmo 4: Algoritmo per il calcolo dei cammini linearmente indipendenti usando la tecnica delle baseline successive.

utilizza la funzione di fitness definita da Wegener et al. [63] per la copertura di percorsi. Questa funzione consiste di due parti: *approach level* e *branch distance*.

Al fine di definire le due misure, è necessaria una definizione preliminare.

Definizione 2. *Si definisce* ***percorso di esecuzione*** *(o percorso eseguito) come il percorso sul* Control Flow Graph $e_e = \langle E_1, E_2, ..., E_m \rangle$ *relativo ad un dato input. L'insieme di nodi eseguiti* $e_n = \{N_1, ..., N_{m+1}\}$ *è tale che:*

- N_1 *equivale al nodo di partenza del* Control Flow Graph.
- N_i *(con* $i \in \{2, ..., m\}$*) è tale che, dati i due archi* E_{i-1}, E_i *del percorso eseguito,* $E_{i-1} = (N_{i-1}, N_i)$ *e* $E_i = (N_i, N_{i+1})$
- N_{m+1} *equivale al nodo di fine del* Control Flow Graph.

Approach level L'*approach level* è calcolato come il numero di nodi appartenenti al percorso obiettivo ma non a quello di esecuzione. Formalmente,

si consideri il percorso obiettivo $t_e = \langle T_1, ..., T_k \rangle$. L'*approach level* è calcolato come:

$$approach\_level(e_e, t_e) = |t_e - e_e|$$

Per rendere la funzione più precisa, è opportuno considerare solo le sezioni in cui i due percorsi divergono [63]: se i percorsi si ricongiungono, gli archi percorsi da quel punto in poi (finché non divergono di nuovo, eventualmente) non devono essere sommati per il calcolo dell'*approach level*.

Branch distance Si definisca D_{NN} come il sottoinsieme dei nodi del *Control Flow Graph* tale che $\forall NN \in D_{NN}, t_e \ni E_\alpha = (NN, N_\alpha)$ e $e_e \ni E_\beta = (NN, N_\beta)$ con $E_\alpha \neq E_\beta$, ovvero quei nodi in corrispondenza dei quali i percorsi t_e e e_e divergono. Questi nodi sono sicuramente condizionali (altrimenti i percorsi non divergerebbero in quei punti).

La *branch distance* relativa al percorso di esecuzione $e_e = \langle E_1, ..., E_m \rangle$ e al percorso obiettivo $t_e = \langle T_1, ..., T_k \rangle$ è data dalla formula:

$$branch\_distance(e_e, t_e) = \sum_{NN \in S_{NN}} bd(NN)$$

Dove la funzione *bd* indica la *branch distance* basilare del nodo condizionale NN.

Funzione obiettivo La funzione obiettivo è definita come:

$$o(e_e, T) = approach\_level(e_e, T) + normalize(branch\_distance(e_e, T))$$

In questo caso è stata utilizzata la funzione definita da Arcuri [4] per normalizzare la *branch distance*. Dato un valore assoluto di *branch distance* x, la funzione di normalizzazione è definita come:

$$normalize(x) = \frac{x}{x+1}$$

Si consideri l'esempio in figura 3.9. In giallo è evidenziato il percorso di esecuzione per $a = 0$ e $b = 0$, in rosso il percorso di obiettivo. In questo

caso il numero totale di nodi che sono nel percorso obiettivo ma non in quello di esecuzione è 2 (i nodi sono `a++` e `b++`), quindi il valore dell'*approach level* è, appunto, 2. Per calcolare la *branch distance*, bisogna considerare i nodi in cui i due percorsi divergono, quindi, in questo caso, `a==0` e `b==0`. Si somma, quindi, la *branch distance* relativa a questi due nodi (indicata sugli archi) e il risultato è $2.3 + 12 = 14.3$. Infine, per calcolare il valore della funzione obiettivo, si normalizza la *branch distance*, quindi $normalized\_branch\_distance = \frac{14.3}{14.3+1} \approx 0.9346$, da cui il valore finale è $o(e_e, T) \approx 2 + 0.9346 = 2.9346$.

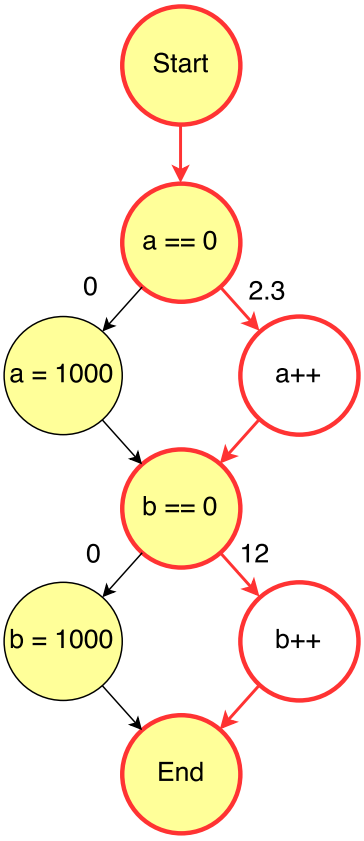


Figura 3.9: Esempio per il calcolo della funzione di fitness nel caso in cui l'obiettivo è un percorso. I nodi con lo sfondo giallo fanno parte del cammino di esecuzione, mentre quelli cerchiati di rosso rappresentano il percorso obiettivo. Sugli archi sono presenti le *branch distance* calcolate in base alla condizione del nodo padre.

3.2.2 Copertura degli archi non coperti

Come accennato in precedenza, qualora si riuscissero a selezionare dati di test che permettono di coprire esattamente i percorsi linearmente indipendenti, si riuscirebbe ad avere una *branch coverage* del 100%. Tuttavia esistono dei problemi teorici e pratici che quasi sempre impediscono il raggiungimento di questo obiettivo ideale:

- I percorsi potrebbero essere molto difficili da coprire
- I percorsi potrebbero essere *infeasible*

Nel primo caso il problema è pratico, poiché si è di fronte a percorsi per i quali gli algoritmi di ricerca hanno difficoltà a trovare i dati di input. Questa problematica è presente per la copertura di un singolo branch, e naturalmente si amplifica quando l'obiettivo è coprire un intero percorso (quindi molti branch contemporaneamente).

Il secondo problema è ancora più insidioso: se nel primo caso è possibile aumentare il tempo da dare agli algoritmi di ricerca per avere più probabilità di trovare i dati di input, è impossibile trovare dati di input che permettano di coprire un percorso *infeasible*. In definitiva, possono sorgere diversi problemi che impediscono all'algoritmo di ricerca di riuscire a trovare i dati di test per coprire alcuni dei percorsi. Il risultato finale può essere il raggiungimento di un livello di copertura inferiore rispetto a quello desiderato.

Per questo motivo, una volta terminata la ricerca dei dati di test relativi ai percorsi linearmente indipendenti, si passa alla ricerca dei dati di test che permettano di coprire, uno alla volta, i singoli branch non ancora coperti nel *Control Flow Graph*. In questa fase può capitare che, nel tentativo di coprire un singolo branch, si riescano a coprire anche altri branch. La copertura di alcuni di questi può essere desunta staticamente grazie ad un *dominator tree*, un'altra rappresentazione di un programma, mentre la copertura di altri dipende dalla specifica esecuzione. In definitiva non si effettuerà una ricerca separata per quei branch già coperti collateralmente grazie ad altri dati di test.

Anche in questo caso si utilizza l'algoritmo genetico definito nella sezione 3.1.3 al fine di trovare l'input che permette di coprire i singoli branch obiettivo non ancora coperti. Si utilizza la funzione di fitness definita da Wegener et al. [63] e utilizzata in vari lavori successivi [25] [50] per la copertura di singoli archi. Anche in questo caso la funzione obiettivo è composta da *branch distance* e *approach level*, calcolati però in maniera diversa.

Approach level L'*approach level* è calcolato come la distanza minima tra un qualsiasi nodo per cui è passato il controllo e il nodo da cui parte l'arco obiettivo. Più formalmente, dato l'insieme di nodi eseguiti $e_n = \{N_1, ..., N_{m+1}\}$, si indichi con T il nodo da cui parte l'arco obiettivo; l'*approach level* si definisce in questo modo:

$$approach\_level(e_e, T) = \min_{i=1}^{m+1} distance(N_i, T)$$

Branch distance Si definisca $NN = \arg\min_{i=1}^{m+1} distance(N_i, T)$ come il nodo eseguito appartenente a $e_n = \{N_1, ..., N_{m+1}\}$ più vicino al nodo (T) da cui parte l'arco obiettivo. È possibile enunciare il seguente teorema:

Teorema 2. *Si possono avere solo due casi:*

- $NN = T$
- NN *è un nodo condizionale*

Dimostrazione. Si ammetta, per assurdo, che NN non sia un nodo condizionale ed che sia diverso da T. Non essendo un nodo condizionale, si può considerare il nodo NN_1 successore del nodo NN nel percorso, poiché il nodo NN avrà soltanto un arco uscente. Dato che $NN \neq T$, NN_1 è sicuramente più vicino a T rispetto a NN; ma questo negherebbe l'ipotesi secondo cui NN è il nodo più vicino a T. □

La *branch distance* relativa al percorso di esecuzione $e_e = \langle E_1, ..., E_m \rangle$ e al nodo obiettivo T equivale alla *branch distance* del nodo NN, oppure è uguale alla *branch distance* relativa all'arco obiettivo se $NN = T$. Se l'arco obiettivo è attraversato, banalmente la *branch distance* varrà 0.

Funzione obiettivo Come nel caso descritto in precedenza, la funzione obiettivo è definita come:

$$o(e_e, T) = approach\_level(e_e, T) + normalize(branch\_distance(e_e, T))$$

dove la funzione *normalize* normalizza la *branch distance* in modo che assuma valori nell'intervallo $[0, 1)$. Si utilizza qui la stessa funzione di normalizzazione definita in 3.2.1

Si consideri l'esempio in figura 3.10. In giallo è evidenziato il percorso di esecuzione per l'input $a = 0$, $b = 0$, in rosso il branch obiettivo. Si prende il nodo del percorso di esecuzione più vicino al padre del branch obiettivo, quindi, in questo caso `a==0`, e si calcola la distanza tra i due, che nell'esempio è 1. La *branch distance* è esattamente quella del nodo `a==0`, quindi 2.3. Infine, per calcolare il valore della funzione obiettivo, si normalizza la *branch distance*, quindi $normalized\_branch\_distance = \frac{2.3}{2.3+1} \approx 0.6970$, da cui il valore finale è $o(e_e, T) \approx 1 + 0.6970 = 1.6970$.

3.2.3 Ottimizzazione della test suite

È possibile che i percorsi linearmente indipendenti non costituiscano un insieme minimale di casi di test che permettono di avere la massima *branch coverage*: inoltre, anche nei casi in cui costituiscono un insieme minimale, è possibile che alcuni percorsi non siano coperti e, quindi, che si ricorra alla tecnica che permette di coprire singoli branch. Per questo motivo, una volta terminata la fase di generazione dei dati di test, si passa alla minimizzazione della test suite generata. Questo processo consente di eliminare alcuni casi di test che danno un contributo nullo, ovvero che, rispetto agli altri, non coprono nessun arco aggiuntivo. La minimizzazione, quindi, permette di mantenere costante la copertura pur riducendo il numero di casi di test.

Le tecniche di minimizzazione della test suite sono descritte in maniera più approfondita nella sezione 2.3. In questo caso si è scelto di utilizzare l'al-

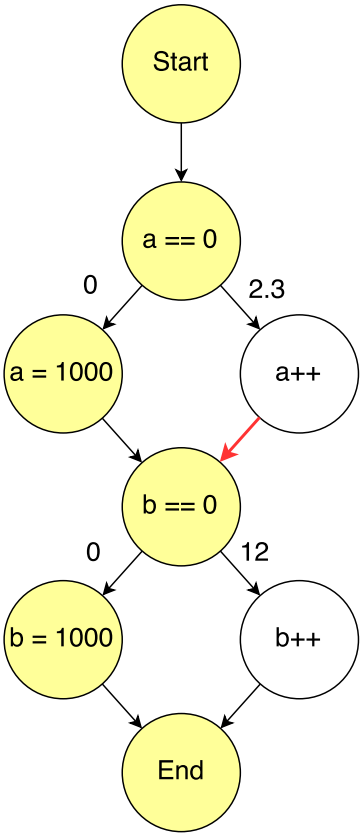


Figura 3.10: Esempio per il calcolo della funzione di fitness nel caso in cui l'obiettivo è un branch. I nodi con lo sfondo giallo fanno parte del cammino di esecuzione; in rosso il branch obiettivo. Sugli archi sono presenti le *branch distance* calcolate in base alla condizione del nodo padre.

goritmo greedy addizionale (algoritmo 5), che coniuga un'estrema semplicità ad ottimi risultati.

L'algoritmo prende in input la test suite da minimizzare e, ad ogni iterazione, aggiunge all'insieme che rappresenta la test suite minimizzata il caso di test che, aggiunto alla test suite temporanea, massimizza la copertura temporanea. Il ciclo si ferma nel momento in cui si raggiunge la stessa copertura della test suite originale.

Data: S_1
Result: S_m
$C_1 = Cov(S_1)$;
$C_m = \varnothing$;
$S_m = \varnothing$;
while $C_m \neq C_1$ **do**
    $T_b = \arg\max_{T \in S_1} Cov(S_m \cup \{T\})$;
    $S_m = S_m \cup \{T_b\}$;
    $C_m = Cov(S_m)$;
end

Algoritmo 5: Algoritmo greedy addizionale

Si consideri l'esempio in figura 3.11. Per semplicità, si assuma di voler raggiungere la *statement coverage* totale e che i casi di test generati dai primi passi dell'approccio siano $S_0 = \{ T_1, T_2, T_3, T_4, T_5, T_6 \}$, i quali usano come parametro `a`, rispettivamente, -2, 0, 1, 4, 45, 23. Innanzitutto è necessario generare una matrice (come quella in tabella 3.3) che associ casi di test ed elementi coperti (in questo caso, istruzioni). L'algoritmo parte considerando uno dei casi di test tra T_1, T_4, T_5 e T_6: questi, infatti, coprono un maggior numero di elementi rispetto ai restanti due. Ammettiamo che aggiunga alla test suite minimizzata il caso di test T_0.

Alla seconda iterazione, i casi di test T_4, T_5 e T_6 avranno copertura addizionale pari a 0, poiché non coprono alcun elemento in più rispetto a T_0, scelto in precedenza. La scelta, in questo caso, potrà ricadere indifferentemente su T_2 o T_3, perché entrambi hanno copertura addizionale rispetto a T_0 pari a 1. Supponiamo che sia scelto T_2.

Alla terza iterazione, infine, verrà scelto T_3, l'unico con copertura addizionale pari a 1. A questo punto è raggiunta la copertura originale, quindi l'algoritmo può fermarsi. La test suite minimizzata sarà $S_M = \{T_0, T_1, T_2\}$.

```
1  int factorial(int a) {
2    if (a == 0)
3      return 0;
4
5    if (a == 1)
6      return 0;
7
8    int result;
9    for (result = a; a > 1; a--)
10     result *= a;
11
12   return result;
13 }
14
```

Figura 3.11: Funzione per il calcolo del fattoriale.

	T_1	T_2	T_3	T_4	T_5	T_6
if (a == 0)	X	X	X	X	X	X
return 0;		X				
if (a == 1)	X		X	X	X	X
int result;			X			
for (...)	X			X	X	X
result *= a;	X			X	X	X
return result;	X			X	X	X

Tabella 3.3: Matrice di copertura di alcuni casi di test individuati. Il criterio di copertura è statement coverage.

Capitolo 4

Architettura e implementazione del tool

I tool descritti nella sezione 2.2.4 hanno diversi problemi: alcuni non sono pubblicamente disponibili, altri sono disponibili, ma solo come eseguibili (non è fornito, quindi, il codice sorgente). Altri, infine, sono open-source, ma sono stati sviluppato in linguaggi poco diffusi (si veda Austin, sviluppato in OCaml).

Per queste ragioni si è ritenuto opportuno implementare un nuovo tool a supporto degli studi futuri. Il tool, denominato **OCELOT**, è stato progettato seguendo, a grandi linee, l'approccio utilizzato per IGUANA [44]. OCELOT è in grado di generare, a partire da una funzione C, una test suite che riesca a coprire il maggior numero di branch possibili.

Come accennato, OCELOT è implementato in Java, soprattutto per la disponibilità di un gran numero di componenti open-source riutilizzabili (CDT, jMetal [17], etc.).

In questo capitolo saranno descritti dettagliatamente le componenti riutilizzate principali, i sotto-sistemi del tool e il processo di esecuzione dello stesso.

4.1 Componenti riutilizzate

Innanzitutto si è proceduto con una rassegna preventiva delle componenti (librerie o framework) che potevano essere utili nello sviluppo del tool. In particolare sono state utilizzate componenti per l'analisi del codice sorgente C (CDT), per l'esecuzione di meta-euristiche (jMetal [17]) e altre librerie utili (jGrapht per la gestione dei grafi, Apache Commons Maths per le funzioni matematiche avanzate).

4.1.1 CDT

Il framework CDT (C/C++ Development Tooling) di Eclipse offre numerose classi in grado di facilitare l'analisi statica di codice sorgente C. In particolare sono state utilizzate le seguenti utilità:

- Meccanismo di estrazione dell'*Abstract Syntax Tree* di un programma e relativo meccanismo per la visita
- Risoluzione dei tipi delle variabili (binding)
- Visitor per la traduzione di un *Abstract Syntax Tree* in codice sorgente C

Innanzitutto è stata utilizzata la classe `GCCLanguage` per estrarre, dato un file contenente codice C e le cartelle che conterranno i file da includere (header), l'*Abstract Syntax Tree* relativo ad una specifica funzione di cui si vogliono generare i casi di test. Questa rappresentazione è necessaria, perché consente di analizzare molto facilmente il programma. In particolare, in OCELOT, si visita l'*Abstract Syntax Tree* sia per generare il *Control Flow Graph* del programma, sia per instrumentare il codice. Per fare ciò, sono stati definiti dei visitor specifici, estendendo la classe `IASTVisitor` e implementando alcuni metodi che saranno richiamati automaticamente quando si visitano determinate porzioni dell'albero. Nella tabella 4.1 sono illustrati i metodi principali che possono essere implementati in un visitor.

All'interno del visitor che permette di instrumentare il codice, è stato necessario analizzare i tipi delle diverse variabili utilizzate in ogni istruzione.

Nome metodo	Descrizione
visit(IASTTranslationUnit)	Visita del nodo radice dell'albero, che rappresenta il programma nel suo insieme.
visit(IASTDeclaration)	Visita di una dichiarazione, sia essa di funzione, di variabile, di tipo e così via.
visit(IASTStatement)	Visita di una riga di codice, che generalmente si conclude con un `;` o con un `}`. Comprende tutte le possibili istruzioni, quindi assegnazioni, strutture di controllo e così via.
visit(IASTExpression)	Visita di una espressione, ovvero di una porzione di istruzione. Un'espressione può essere, ad esempio, la parte destra di un'assegnazione, la condizione di un `if` e così via.

Tabella 4.1: Metodi principali della classe ASTVisitor

CDT comprende alcune utilità che permettono di facilitare questo compito, poiché, quando genera l'*Abstract Syntax Tree*, oltre a fare un'analisi sintattica, procede anche ad una parziale analisi semantica del codice: tra le altre cose, CDT effettua il *binding* dei tipi delle variabili.

Un'ultima classe utilizzata di CDT è `ASTWriter`. Questa classe permette di trasformare un *Abstract Syntax Tree* in codice sorgente C. Questo è molto utile, poiché nella fase di instrumentazione è stato sufficiente modificare l'albero, e non definire un modo per scrivere direttamente il C. È sufficiente chiamare il metodo `write` di un'istanza di `ASTWriter` per assolvere a questo compito.

4.1.2 jMetal

Il termine jMetal si riferisce all'acronimo *Metaheuristic Algorithms in Java* [17]. Si tratta di un framework Java per la risoluzione di problemi di ottimizzazione mono/multi-obiettivo attraverso l'utilizzo delle più svariate meta-euristiche. jMetal fornisce un ricco set di classi utilizzabili per l'implementazione della maggior parte delle tecniche di ricerca presenti in letteratura. I diversi algoritmi implementati condividono le stesse componenti base, quali, ad esempio, operatori genetici vari. Si è scelto di utilizzare jMetal va-

lutando una serie di caratteristiche del framework stesso, che corrispondono esattamente agli obiettivi di design che ci si è posti. In particolare i motivi sono riassumibili nei seguenti punti:

- **Semplicità e facilità d'uso**: le classi base fornite da jMetal (`Solution`, `SolutionSet`, `Variable`, e così via) sono semplici ed intuitive e, di conseguenza, facili da capire e da utilizzare. Inoltre, il framework include l'implementazione di numerose meta-euristiche.

- **Flessibilità**: è semplice, per ogni algoritmo, impostare lo stesso problema per l'esecuzione attraverso differenti parametri, più o meno dipendenti dal problema da risolvere.

- **Estendibilità**: l'aggiunta di nuovi algoritmi, nuovi operatori e nuovi tipo di problemi è relativamente semplice e veloce. Ad esempio, tutti i problemi derivano dalla classe `Problem`. Ciò permette facilmente la creazione di un nuovo problema semplicemente definendo i metodi specifici per quella classe.

L'architettura base di jMetal si basa su un `Algorithm` che ha lo scopo di risolvere un `Problem` attraverso l'utilizzo di una (o più) `SolutionSet` e un insieme di `Operator`. Rispetto alla terminologia relativa agli algoritmi genetici, si può assimilare la popolazione a un `SolutionSet`, mentre una soluzione corrisponde a una `Solution`.

Codifica della soluzione La codifica della soluzione al problema che si sta tentando di risolvere rappresenta un primo aspetto da valutare quando ci si approccia all'utilizzo di una determinata euristica. Ovviamente la rappresentazione dipende fortemente dal problema e ha un impatto anche sulle operazioni che possono essere applicate.

Una `Solution` è composta da un insieme di `Variable`, le quali possono essere di tipo differente (binarie, reali, interi e così via), e da un array nel quale sono conservati i valori di fitness. Ad ogni `Solution` è associato un tipo (`SolutionType`). Questo meccanismo permette di definire

i tipi di una `Solution` e di crearne altre attraverso l'utilizzo del metodo `createVariables`.

Una codifica delle soluzioni di questo tipo si rivela estremamente estendibile alla creazione di nuove rappresentazioni più complesse e particolari (anche attraverso la combinazione di differenti tipi) che possono essere richieste da alcune tipologie di problemi.

Operatori Le differenti tecniche meta-euristiche sono accomunate dal fatto che esse tentano di arrivare a una soluzione al problema tramite la modifica o la generazione di una soluzione a partire da una esistente attraverso l'applicazione di una serie di operatori. In jMetal, ogni operazione che ha effetto su una soluzione eredita dalla classe `Operator`. jMetal, oltre a permettere la definizione di nuovi operatori ah hoc, contiene un numero abbastanza ampio di operatori predefiniti. Sono elencate le categorie principali di operatori:

- `Selection`: questo tipo di operatore viene utilizzato per le operazioni di selezione a partire da una popolazione (tipico degli algoritmi evolutivi). Un esempio di operatore di selezione è il *binary tournament*.
- `Crossover`: implementano gli operatori di ricombinazione e di *crossover* classici della teoria degli algoritmi evolutivi. Ad esempio, jMetal fornisce l'implementazione del *simulated binary crossover* (SBX crossover) e il *two-point crossover*.
- `Mutation`: rappresentano gli operatori di mutazione degli algoritmi evolutivi. Esempi di operatori di mutazione già forniti da jMetal sono il *polynomial mutation* e il *bit-flip mutation*.
- `LocalSearch`: rappresentano procedure per la ricerca locale.

Ogni operatore contiene i metodi `setParameter` e `getParameter`, utilizzati per aggiungere o accedere a specifici parametri riguardo un operatore (un classico esempio di parametro è la probabilità con la quale verrà applicato un operatore di *crossover* o di mutazione).

Problemi Come accennato in precedenza, in jMetal ogni problema deriva dalla classe `Problem`. Questa classe contiene due metodi base: `evaluate` e `evaluateContraints`. Entrambi i metodi ricevono una `Solution`, la quale rappresenta una soluzione candidata alla risoluzione del problema. Il primo metodo si occupa di valutare tale soluzione, mentre il secondo va a controllare se sono stati violati dei vincoli alla soluzione stessa. Tutti i problemi che vengono definiti devono necessariamente definire il metodo `evaluate`, mentre solamente i problemi che presentano dei vincoli hanno bisogno di ridefinire il metodo `evaluateContraints`.

Algoritmi L'ultima classe da analizzare è `Algorithm`. Questa rappresenta una classe astratta da cui dovranno ereditare tutte le meta-euristiche incluse nel framework, nonché tutte quelle che verranno definite successivamente. In particolare, per ogni approccio dovrà essere implementato il metodo `execute`, il quale sarà il metodo effettivamente richiamato nel momento in cui l'algoritmo dovrà essere eseguito. Questo metodo restituisce un risultato sotto forma di un `SolutionSet`.

4.2 Componenti ed architettura del sistema

Una prima fase necessaria per la realizzazione del tool è consistita nel progettare accuratamente il sistema da realizzare. Questa attività è importante, perché permette di trovare in anticipo e risolvere, se possibile, eventuali problematiche critiche. In particolare, è stato decomposto il sistema intero in vari sotto-sistemi che potranno essere realizzati in maniera relativamente isolata.

Il sistema sarà composto da 9 sotto-sistemi fondamentali, come mostrato nella figura 4.1.

In seguito saranno descritte in maniera più dettagliata tutte le parti di OCELOT.

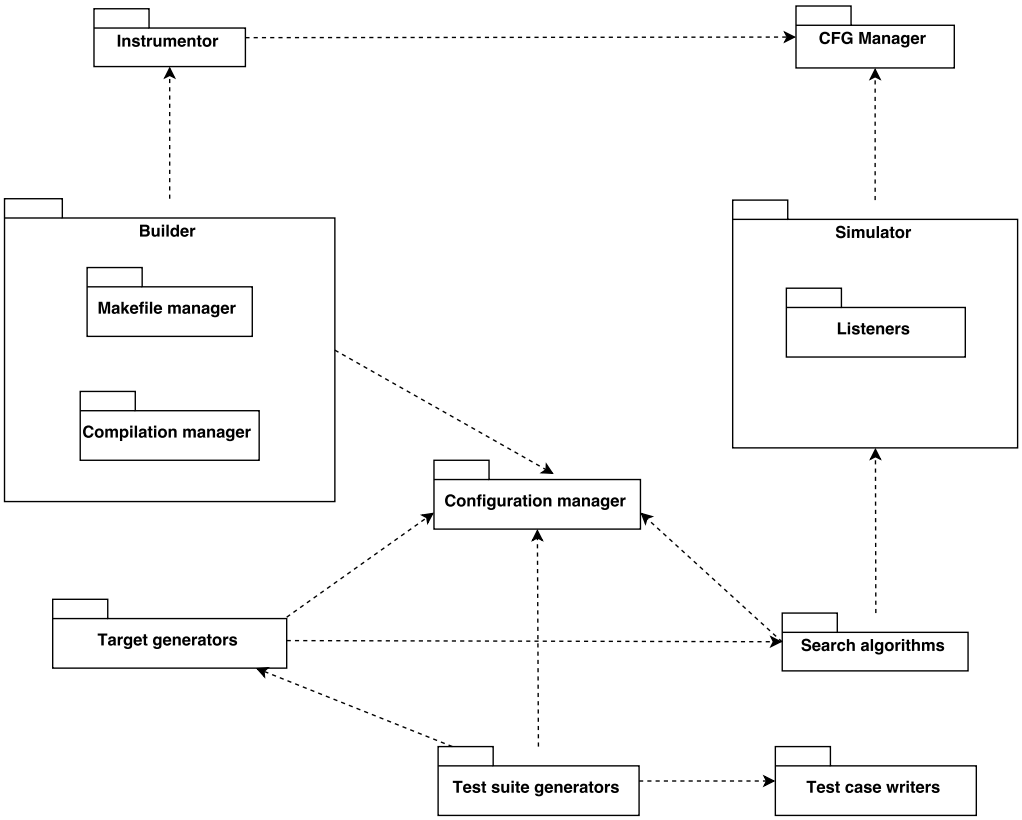


Figura 4.1: Package-diagram di OCELOT

4.2.1 Creazione del Control Flow Graph

Una delle componenti basilari è il generatore del *Control Flow Graph* del programma in input. Il pacchetto comprende varie classi, tra cui, quelle più importanti, sono:

- `CFGVisitor`: *visitor* dell'albero sintattico generato da CDT che permette di generare il *Control Flow Graph* di una funzione
- `CFG`: rappresentazione interna in OCELOT di un *Control Flow Graph*. Questa classe utilizza la libreria jGrapht per la gestione del grafo alla base del *Control Flow Graph*.

Il *Control Flow Graph* è costruito in maniera incrementale: si esegue una visita in profondità dell'albero sintattico, quindi si costruisce dei sotto-grafi a partire dalle istruzioni più semplici. Ogni sotto-grafo è identificato da:

- Una lista di nodi di input, che rappresentano i nodi di ingresso del sotto-grafo
- Una lista di nodi di output, che rappresentano i nodi di uscita del sotto-grafo
- Liste supplementari di nodi case (per gli `switch`), `break` (per i cicli e gli `switch`) e `continue` (per i cicli)

Si consideri di nuovo l'esempio in figura 3.3. Come descritto più dettagliatamente nella sezione 3.1, in primo luogo vengono creati i sotto-grafi relativi alle righe 2 e 4. Le liste di input/output saranno composte esattamente da quell'unico nodo (la tabella 4.2 mostra il risultato parziale ottenuto).

Lista	**Sotto-grafo A**	**Sotto-grafo B**
Input	(b = 11)	(b = 10)
Output	(b = 11)	(b = 10)

Tabella 4.2: Le liste relative ai primi due sotto-grafi creati a partire dalle istruzioni "b=11" e "b=10".

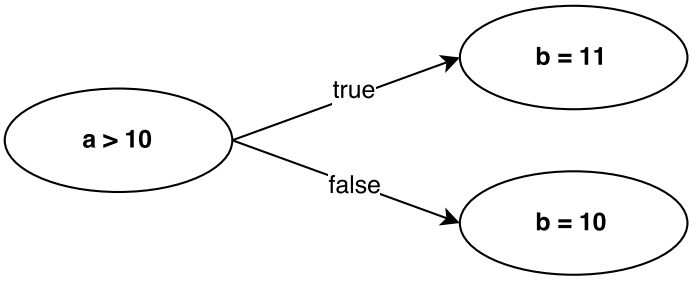


Figura 4.2: Control Flow Graph finale.

La tabella 4.3 mostra il risultato della visita all'istruzione condizionale: vengono creati gli archi dal nodo condizione (a > 10) agli input di **A** (b = 11) e **B** (b = 10).

Lista	**Sotto-grafo C**
Input	(a > 10)
Output	Output di **A** e di **B** [(b = 11), (b = 10)]

Tabella 4.3: Le liste relative al sotto-grafo creato a partire dall'intera istruzione "if", dati i due sotto-grafi A e B.

Il sotto-grafo risultante può essere utilizzato da un eventuale padre come riportato nella figura 4.2. Gli archi del grafo hanno un'etichetta simbolica che ne identifica il tipo. Questi si distinguono in:

- `Flow`: Arco che parte da un nodo con *outdegree* pari a 1. È un arco che viene sicuramente percorso, se il *visitor* si trova nel suo nodo di origine.

- `True`: Arco che parte da un nodo condizionale (che ha sempre *outdegree* pari a 2). Il nodo di origine deve avere anche un arco uscente etichettato `False`.

- `False`: Arco che parte da un nodo condizionale (che ha sempre *outdegree* pari a 2). Il nodo di origine deve avere anche un arco uscente etichettato `True`.

- `Case`: Arco che parte da un nodo relativo ad un'istruzione di tipo `switch`. Il nodo di origine può avere un *outdegree* variabile (maggiore di 0), e ha solo archi uscenti etichettati con `Case`. L'etichetta contiene anche le informazioni relative all'espressione `Case` del nodo di arrivo.

4.2.2 Instrumentazione del codice

La componente più critica di OCELOT è quella che ha il compito di instrumentare il codice sorgente in modo tale da:

- Registrare tutti gli eventi (archi non banali seguiti nel *Control Flow Graph* per un dato input)

- Calcolare i valori delle *branch distance* (si veda la sezione 2.2.3 per maggiori dettagli)

La complessità di tale compito è data dal dover considerare tutti i casi possibili, le varie combinazioni di tipi utilizzati e così via.

In primo luogo sono state definite alcune funzioni di base (in C), che sono utilizzate dal codice instrumentato:

- `_f_ocelot_trace`: Aggiunge un evento alla lista.

- `_f_ocelot_XXX_numeric`: Calcola la distanza per una determinata espressione di confronto (al posto di XXX; es: uguale, maggiore, maggiore o uguale, etc.) tra due espressioni di tipo numerico.

- `_f_ocelot_XXX_pointer`: Calcola la distanza per una determinata espressione di confronto (al posto di XXX; es: uguale, maggiore, maggiore o uguale, etc.) tra due espressioni di tipo "puntatore a numerico".

Un evento registrato nella lista contiene i seguenti campi:

- Scelta effettuata: Può assumere i valori 0 (`false`) o 1 (`true`)
- Distanza da `true`: *Branch distance* rispetto al branch etichettato `True` del *Control Flow Graph*
- Distanza da `false`: *Branch distance* rispetto al branch etichettato `False` del *Control Flow Graph*

Per le istruzioni `switch` si registrano n eventi consecutivi, dove n è il numero di istruzioni `case` presenti nello `switch`.

Dato l'albero sintattico della funzione da instrumentare, si visitano soltanto i nodi "interessanti" e vi si innesta la funzione di tracciamento. Questa prende in input tre parametri, ovvero i campi dell'evento da registrare (quindi scelta, distanza da `true` e distanza da `false`). Relativamente al primo parametro, si copia il contenuto del nodo condizione dell'istruzione. Per gli altri due, invece, si inseriscono le chiamate alle funzioni che calcolano le distanze relativa all'espressione originale (distanza da `true`) e all'espressione negata (distanza da `false`). Si modifica, quindi, l'istruzione condizionale, estraendone le componenti e utilizzandole come parametri della funzione che calcola le distanze.

Si consideri il frammento in figura 4.3. Il risultato dell'instrumentazione è presentato in figura 4.4. Nel codice instrumentato vengono modificate soltanto le espressioni condizionali (con l'eccezione dell'istruzione `switch`, la quale richiede un'instrumentazione più complessa). La funzione di tracciamento è definita in modo da restituire sempre il primo parametro passato (quindi la condizione originale), affinché il codice instrumentato si comporti come l'originale.

Nella tabella 4.4 sono elencate le funzioni native principali di OCELOT, attraverso le quali si calcolano le distanze e si tracciano gli eventi. Come si può notare, non sono presenti le funzioni che calcolano le distanze in presenza di operatori "minore" e "minore o uguale", dato che queste possono essere calcolate a partire dalle distanze per "maggiore" e "maggiore o uguale" negate.

Nome funzione	Descrizione
`_f_ocelot_trace`	Traccia un evento e le relative distanze.
`_f_ocelot_trace_case`	Traccia un evento relativo ad una singola istruzione case. Registra la distanza dal branch true e un flag che indica se questo arco è stato percorso.
`_f_ocelot_eq_numeric`	Calcola la distanza dal soddisfacimento della condizione “a == b”.
`_f_ocelot_gt_numeric`	Calcola la distanza dal soddisfacimento della condizione “a > b”.
`_f_ocelot_ge_numeric`	Calcola la distanza dal soddisfacimento della condizione “a >= b”.
`_f_ocelot_neq_numeric`	Calcola la distanza dal soddisfacimento della condizione “a != b”.
`_f_ocelot_and_numeric`	Calcola la distanza dal soddisfacimento della condizione “a && b”.
`_f_ocelot_or_numeric`	Calcola la distanza dal soddisfacimento della condizione “a || b”.

Tabella 4.4: Funzioni principali aggiunte al codice sorgente durante l’instrumentazione al fine di registrare gli eventi. Sono riportate solo le funzioni che agiscono su operatori numerici (per le distanze); le funzioni che agiscono su variabili di tipo “puntatore ad intero” sono analoghe, ma hanno il suffisso “pointer” al posto di “numeric”.

```
if (a == 10) {
  b = 11;
} else {
  b = 10;
}
```

Figura 4.3: Snippet di partenza.

```
if (
  _f_ocelot_trace(
    a == 10,
    _f_ocelot_eq_numeric(a, 10),
    _f_ocelot_neq_numeric(a, 10)
  )
) {
  b = 11;
} else {
  b = 10;
}
```

Figura 4.4: Risultato dell'instrumentazione.

4.2.3 Compilazione della libreria

Avendo a disposizione il codice instrumentato, è possibile compilare la libreria C, che sarà utilizzata in Java attraverso Java Native Interface (JNI). Al fine di far funzionare le chiamate da Java alla libreria nativa, sono state definite delle funzioni apposite.

La componente che si occupa della *build* è divisa in due parti:

- Makefile generator: si occupa della generazione automatica del *makefile* che sarà utilizzato per fare il *build*.
- Compilation manager: definisce alcune funzioni utili, dipendenti dal compilatore C utilizzato, e si occupa di fare il *build*.

La problematica principale della compilazione è il supporto multi-piattaforma. Si è risolto questo problema grazie all'utilizzo del Bridge Design Pattern (in figura 4.5). Il Bridge è un design pattern strutturale, attraverso il quale è possibile disaccoppiare un'astrazione dalla sua implementazione, così

che le due parti possano evolvere indipendentemente l'una dall'altra. Nel caso specifico, si è definita una classe astratta `JNIMakefileGenerator`, la quale contiene una serie di metodi astratti che devono necessariamente essere implementati dalle sottoclassi. Sono state definite successivamente delle classi che implementano i generatori di *makefile* dei principali sistemi operativi in circolazione, quindi `LinuxMakefileGenerator`, `MacOSXMakefileGenerator` e `WindowsMakefileGenerator`. Quando si procede alla *build* della libreria, un'istanza concreta della classe `Builder` si occupa di eseguire tutte le operazioni, quindi, tra le altre cose, di generare il *makefile* specifico a partire da un istanza concreta di `JNIMakefileGenerator` fornita dal chiamante (in questo caso, la classe eseguibile `Build`).

La sezione 4.3.1 contiene una descrizione dettagliata del processo di *build*.

4.2.4 Simulazione dell'esecuzione

La libreria generata, contenente il codice instrumentato, potrà essere chiamata da Java, come accennato precedentemente, attraverso JNI. Grazie ad una singola chiamata è possibile eseguire la funzione da testare e registrare gli eventi non banali. Questo è possibile in quanto è stata definita una componente che, data la lista degli eventi registrati e dato il *Control Flow Graph* della funzione, è in grado di simulare sul grafo l'esecuzione del programma. In questo modo è possibile effettuare tutti i calcoli necessari, senza dover modificare ulteriormente il codice sorgente C.

Per rendere il sistema facilmente estendibile è stato utilizzato un *Observer Design Pattern* (si veda la figura 4.6): qualora si dovessero aggiungere nuovi approcci sarà sufficiente implementare l'interfaccia `SimulationListener`. Il simulatore notificherà a tutti i *listener* associati i seguenti eventi:

- Visita di un nodo.
- Visita di un arco di tipo `Flow`.
- Visita di un arco di tipo `True`/`False`.
- Visita di un arco di tipo Case.

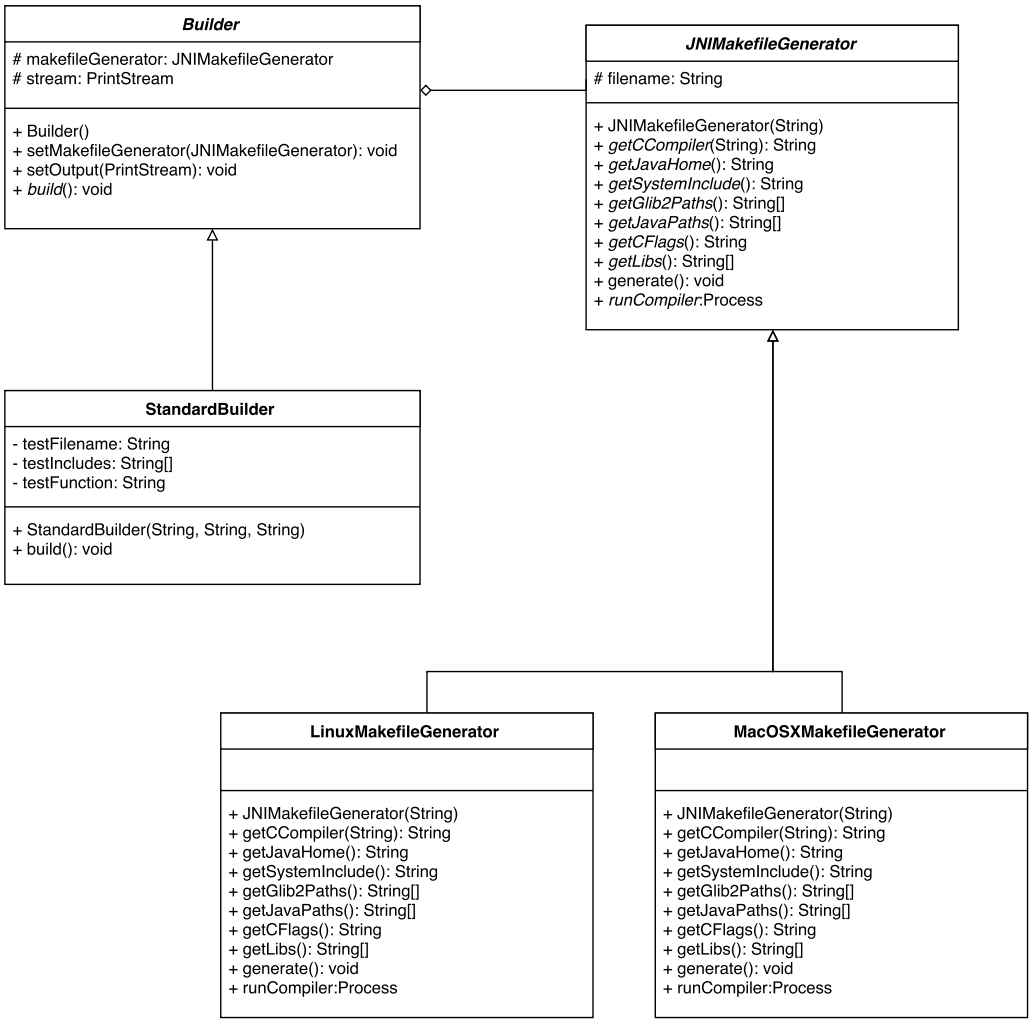


Figura 4.5: Bridge design pattern.

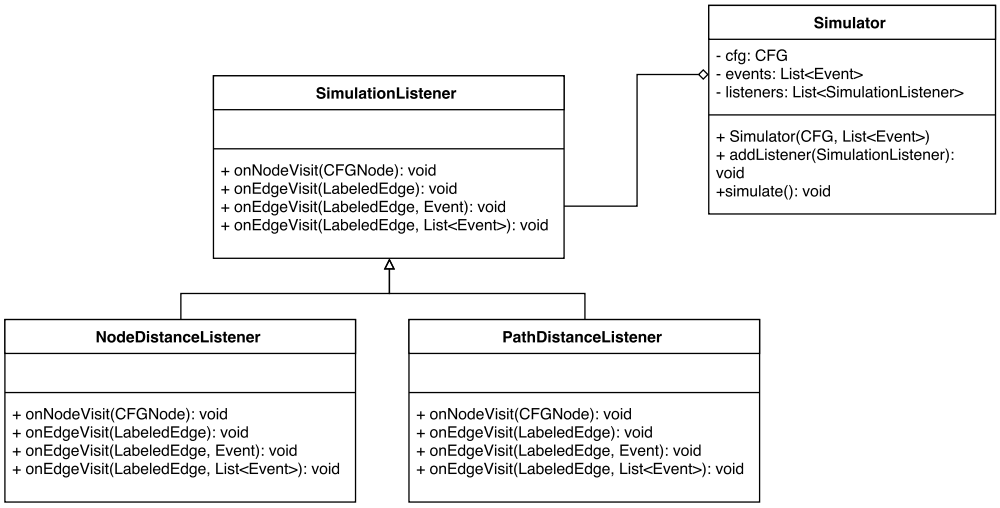


Figura 4.6: Observer design pattern instanziato per il simulatore.

Sono stati definiti alcuni *listener* utili. In particolare:

- `EdgeDistanceListener`: Calcola *branch distance* e *approach level* tra il percorso di esecuzione e l'arco obiettivo.
- `NodeDistanceListener`: Calcola *branch distance* e *approach level* tra il percorso di esecuzione e il nodo obiettivo.
- `PathDistanceListener`: Calcola *branch distance* e *approach level* tra il percorso di esecuzione e il percorso obiettivo.
- `CoverageCalculatorListener`: Calcola la copertura di uno o più percorsi di esecuzione.
- `NodePrinterListener`: Stampa sulla console tutti i nodi visitati (utilizzato soprattutto per il debugging).

4.2.5 Ricerca delle soluzioni

Tutto il sistema finora descritto viene sfruttato dalla componente che si occupa di trovare un insieme di parametri da dare in input alla funzione al fine di coprire un determinato obiettivo. Come detto in precedenza, in letteratura sono stati preferiti algoritmi di ricerca globale, quali, ad esempio, *simulated annealing* e, soprattutto, algoritmi genetici (si veda la sezione 2.1). Per questo motivo si è deciso di implementare, come prima tecnica, proprio la ricerca attraverso algoritmi genetici. In ogni caso, il sistema è strutturato in modo tale da poter essere facilmente esteso con altri algoritmi di ricerca.

Nel caso specifico della generazione di casi di test con approccio goal-oriented, la funzione obiettivo (o di fitness) da minimizzare è quella che indica la distanza da un determinato elemento del *Control Flow Graph* (il target). Ad esempio, si può voler coprire un determinato nodo del *Control Flow Graph*: in questo caso la funzione obiettivo avrà un valore che riflette la distanza del path di esecuzione da quel nodo obiettivo. Se la distanza è 0, il nodo è stato coperto. Per questo specifico problema si sono codificate le soluzioni come vettori di numeri reali. Ogni elemento del vettore rappresenta un parametro da passare alla funzione da testare per valutare la funzione obiettivo.

Come detto in precedenza, si è sfruttato il framework JMetal [17] al fine di implementare più velocemente l'algoritmo genetico. In particolare è stato sufficiente estendere tre classi di JMetal:

- `Experiment`: funge da classe principale, instanzia `Settings` e `Problem` e permette di eseguire l'algoritmo.
- `Settings`: instanzia la specifica versione dell'algoritmo genetico da utilizzare, imposta i parametri e gli operatori.
- `Problem`: esegue la funzione nativa C con i parametri relativi ad una specifica soluzione, esegue la simulazione sul *Control Flow Graph* e, attraverso uno dei listener descritti in precedenza, calcola la funzione di fitness relativa alla soluzione.

4.2.6 Selezione degli obiettivi intermedi e generazione della test suite

Come analizzato in maniera approfondita nella sezione 2.2.3 e nel capitolo 3, è necessario selezionare gli obiettivi intermedi che permettono di raggiungere una certa copertura del *Control Flow Graph*. Per questo motivo, è necessaria la presenza di una componente che abbia il ruolo specifico di definire, dato il *Control Flow Graph*, gli obiettivi intermedi. Questa componente, tuttavia, può essere strettamente legata alla componente che genera i casi di test, poiché, come si vedrà in seguito, la selezione di un determinato caso di test può influire sulla selezione degli obiettivi successivi. Sono state implementate le seguenti metodologie per la selezione di obiettivi intermedi:

- Selezione naive di tutti gli archi
- Selezione intelligente di tutti gli archi
- Selezione dei cammini linearmente indipendenti

Nel primo caso, si selezionano prima tutti gli archi del *Control Flow Graph* e, in seguito, si cerca di coprire ogni singolo arco (algoritmo 6:

Data: E (archi da coprire)
Result: TS
$TS = \varnothing$;
foreach $e \in E$ **do**
    $t = search(e)$;
    $TS = TS \cup t$;
end

Algoritmo 6: Algoritmo naive

Nel secondo caso, si seleziona un arco iniziale, si cerca di coprirlo e, se si raggiunge questo obiettivo, si eliminano dall'insieme degli archi da coprire tutti gli archi coperti dal caso di test selezionato (algoritmo 7).

Infine, nel terzo caso, si selezionano tutti i cammini linearmente indipendenti, come previsto da parte della metodologia descritta nel capitolo 3 (algoritmo 8).

Data: E (archi da coprire)
Result: TS
$TE = \varnothing$;
$TS = \varnothing$;
foreach $e \in TE$ **do**
    $t = search(e)$;
    $TS = TS \cup t$;
    $TE = TE - Cov(t)$;
end

Algoritmo 7: Algoritmo intelligente

Data: $CFG = (V, E)$ (Control Flow Graph)
Result: TS
$BP = FindBasis(CFG)$;
$TS = \varnothing$;
foreach $p \in BP$ **do**
    $t = search(p)$;
    $TS = TS \cup t$;
end

Algoritmo 8: Algoritmo che utilizza i cammini linearmente indipendenti

La componente che genera i casi di test può utilizzare uno dei selettori di obiettivi singoli o combinarne di diversi al fine di migliorare la copertura finale. Ad esempio, la metodologia descritta nel capitolo 3 è stata implementata combinando i selettori 3 (cammini linearmente indipendenti) e 2 (selettore intelligente di archi), ricorrendo a quest'ultimo solo nel caso in cui il selettore dei cammini linearmente indipendenti non riesce a raggiungere la piena *branch coverage*. Questa componente ha anche la responsabilità di minimizzare, eventualmente, la test suite, riducendo così il numero totale di casi di test. Anche per la minimizzazione è stato utilizzato un Bridge design pattern per permettere di scegliere tra vari algoritmi diversi.

4.2.7 Scrittura dei casi di test

Una componente molto importante è quella che permette di scrivere in maniera automatizzata i casi di test. Si utilizza CDT al fine di creare un *Abstract Syntax Tree* che viene successivamente tradotto in codice C dalla stessa libre-

ria. Uno degli obiettivi di design principali è l'espandibilità del tool, quindi, anche in questo caso, si è voluto progettare questo sotto-sistema in modo tale da permettere, in futuro, di scegliere tra diversi framework per la definizione di casi di test in C. Si è scelto di utilizzare, dunque, un Bridge design pattern che permetta di scegliere tra una delle possibili implementazioni dello scrittore di casi di test modificando il meno possibile il codice relativo al chiamante. Ogni caso di test dovrà comporsi delle seguenti parti:

- Chiamata alla funzione da tesare, con i parametri necessari a coprire un obiettivo intermedio
- Controllo dell'oracolo (asserzioni)

Come analizzato in precedenza, allo stato dell'arte non esistono buone tecniche per la definizione automatica dell'oracolo, se non a partire dalle specifiche. Per questo motivo il modulo di scrittura inserisce asserzioni riguardanti sia il valore di ritorno della funzione, sia le variabili di tipo puntatore passate come parametri. Nelle asserzioni si assumerà che la funzione si comporti correttamente: sarà poi un operatore umano a controllare che questo sia vero e a modificare, eventualmente, le asserzioni.

4.2.8 Gestione della configurazione

La configurazione del tool è gestita attraverso una serie di parametri specificati in un file di configurazione. Tra questi sono presenti sia quelli relativi alla configurazione dell'algoritmo genetico, quali, ad esempio, la grandezza della popolazione iniziale, il numero massimo di valutazioni possibili e la probabilità dell'operazione di *crossover*, sia tutte le informazioni sul test object, sulla funzione da testare, sulle cartelle in cui sono presenti gli header file e così via. La figura 4.7 mostra un esempio di file di configurazione.

4.3 Processo di esecuzione

Il processo di esecuzione di OCELOT si compone di due fasi principali:

```
population.size:100
evaluations.max:50000
crossover.probability:0.8
mutation.probability:0.2
threads:1

test.debug:false
test.ui:false
experiment.output.folder:./outputs/
experiment.results.folder:./experiments/
experiment.results.print:true
experiment.runs:1

suite.generator:McCabe
suite.minimizer:AdditionalGreedy
suite.coverage:0.95
suite.writer:CUnit

test.basedir:testobject/
test.filename:test.c
test.includes:folderA/;folderB/

test.function:test_me
test.target:Flow,True,Flow
test.parameters.ranges:-1000:1000 -1000:1000 -1000:1000
```

Figura 4.7: Esempio di file di configurazione di OCELOT

- Build: Dopo aver individuato la funzione da testare dal file di configurazione, viene instrumentato il codice, in modo da registrare tutte le scelte fatte in corrispondenza delle strutture di controllo, vengono importate le librerie standard di OCELOT e il codice viene reso compatibile con JNI e compilato.

- Esecuzione: In questa fase viene carica la libreria precedentemente compilata e viene eseguito l'algoritmo genetico che permette di trovare i dati di test, ovvero i parametri di input della funzione, grazie ai quali è possibile raggiungere una determinata copertura.

4.3.1 Build

Come già accennato a grandi linee, la fase di *build* ha il compito di compilare la libreria che conterrà la funzione da testare. Come prima operazione, la classe `Build` carica tutti i parametri di configurazione già descritti instanziando la classe `ConfigManager`. Come step successivo viene invocato il metodo `instrument`. Tale metodo si avvale di due *visitor* al fine di instrumentare il codice:

- `InstrumentorVisitor`: è il *visitor* più critico, in quanto si occupa dell'instrumentazione vera e propria. Al suo interno sono stati implementati tutti i metodi `visit` necessari per la costruzione dell'albero sintattico della funzione in esame. In questo processo il codice viene instrumentato con le chiamate alle varie funzioni `_f_ocelot_` utilizzate per tracciare il flusso di esecuzione e le informazioni relative alle varie distanze.

- `MacroDefinerVisitor`: questo *visitor* si occupa di generare la macro contenente la chiamata necessaria per l'invocazione del target compilato ed instrumentato tramite JNI.

A questo punto tutto il codice C instrumentato viene salvato nella cartella che contiene le librerie C di OCELOT.

Lo step successivo riguarda la generazione del *makefile* specifico per il sistema operativo in questione. La classe astratta `JNIMakefileGenerator`

mette a disposizione il metodo generate() per la creazione del *makefile*. Come descritto più accuratamente nella sezione 4.2.3, questa classe è estesa per ogni sistema operativo supportato, in modo da fornire i parametri specifici per il sistema operativo trattato, tra cui, ad esempio, la cartella *HOME* di Java e il percorso della libreria "GLib2", che mette a disposizione una parte delle strutture dati utilizzate da OCELOT.

4.3.2 Esecuzione

La successiva fase nel processo di esecuzione di OCELOT è l'esecuzione di uno specifico generatore di test suite, indicato nel file di configurazione. Il generatore si fa carico di:

- Selezionare i target da coprire
- Lanciare l'algoritmo genetico per ogni target
- Minimizzare, eventualmente, la test suite generata

Attualmente vengono stampati sulla console i parametri trovati relativi ai vari casi di test generati, insieme a vari *benchmark* misurati (in particolare la copertura, la grandezza della test suite, il tempo impiegato per generarla). Una volta implementato il modulo per la scrittura dei casi di test, saranno passati a questo i casi di test "virtuali" generati in modo da trasformarli in casi di test compilabili ed eseguibili.

Capitolo 5

Caso di studio

In questo capitolo è descritta la sperimentazione condotta al fine di dimostrare la validità della nuova metodologia introdotta nel capitolo 3. Per tutti gli esperimenti è stato utilizzato il tool OCELOT, descritto dettagliatamente nel capitolo 4.

5.1 Delimitazione e contesto

L'obiettivo principale dello studio è dimostrare la validità della nuova metodologia, quindi che questa è in grado di raggiungere un livello di copertura dei branch elevata, superiore rispetto all'approccio casuale e, soprattutto, che grazie a questa è possibile ridurre considerevolmente il numero di casi di test totali. Benché la metodologia preveda tre macro-passi (generazione dei casi di test per i cammini linearmente indipendenti, generazione dei casi di test per i singoli branch non coperti e minimizzazione della test suite), in una prima fase si ignorerà la minimizzazione, per effettuare un confronto "alla pari" con l'algoritmo casuale.

La tabella 5.1 riporta i dettagli sulle funzioni utilizzate. Sono state scelte funzioni provenienti da progetti open-source (Gimp e Spice) in modo tale da poter coprire varie casistiche.

La funzione `gimp_rgb_to_hsl_int` permette di proiettare un colore dallo spazio RGB allo spazio HSL. Questa è difficile da coprire interamente, poiché

contiene `if` annidati e confronti con variabili che non sono quelle di input (ma derivano da esse). Questi ultimi possono dare problemi all'algoritmo genetico, perché, quando è necessario trovare l'input che permette di coprire il branch `false` relativo ad un confronto di uguaglianza (quindi si vuole cercare l'input che fa sì che gli operandi siano diversi), l'algoritmo non ha una guida, per come è definita la branch distance per l'operatore di disuguaglianza.

La funzione `gimp_rgb_to_hsv4` ha un obiettivo simile rispetto alla funzione `gimp_rgb_to_hsl_int`, quindi fa una proiezione dallo spazio RGB a quello HSV.

La funzione `cliparc` è la più grande tra quelle testate (sia in termini di complessità ciclomatica, sia di numero di branch). Nonostante il numero di parametri in input molto elevato (9), la funzione non presenta particolari criticità. Si noti che, all'interno del programma, è presente un *sanity check*: questi controlli servono a gestire situazioni in cui è presente un errore nel codice, e quindi, in linea teorica, non dovrebbero risultare mai veri. Escludendo i sanity check, la copertura massima raggiungibile è poco superiore al 95%.

La funzione `Csqrt` permette di calcolare la radice quadrata di un numero complesso. La funzione, apparentemente, non ha una complessità elevata; questa, tuttavia, presenta una criticità, ovvero un controllo che confronta la parte reale e la parte immaginaria del numero in input con il valore 0.

La funzione `triangle`, infine, è tipicamente usata in letteratura per testare gli approcci per la generazione automatica di casi di test. Questa permette di classificare un triangolo. Come nel caso precedente, la complessità non è molto elevata; ci sono, tuttavia, una serie di criticità (`if` annidati con controllo di uguaglianza, che possono essere molto difficili da soddisfare), confronti con numeri costanti e così via.

5.2 Pianificazione

Le domande che ci si è posti sono tre:

Funzione	Complessità ciclomatica	Branch	LOC
`gimp_rgb_to_hsl_int`	7	30	58
`cliparc`	32	107	136
`gimp_rgb_to_hsv4`	9	35	62
`Csqrt`	3	15	26
`triangle`	7	22	21

Tabella 5.1: Funzioni utilizzate per la sperimentazione

RQ1: *La nuova metodologia permette di avere una copertura maggiore rispetto alla generazione casuale di casi di test a parità di tempo?*

RQ2: *A parità di copertura raggiunta, utilizzando la nuova metodologia si riesce a generare un numero di casi di test inferiore rispetto alla generazione casuale di casi di test?*

RQ3: *Quanto contribuisce ogni macro-passo della metodologia proposta a raggiungere un determinato livello di copertura? Quanto contribuisce la minimizzazione alla riduzione dei casi di test?*

RQ1 e RQ2 Per rispondere alla prima e alla seconda domanda, si utilizzerà OCELOT per generare casi di test sia utilizzando il nuovo approccio, sia attraverso l'approccio casuale. Dato che entrambi gli approcci hanno una forte componente che dipende dal caso, saranno lanciati gli algoritmi di generazione automatica dei casi di test 10 volte e sarà riportata la media dei risultati ottenuti in quanto a copertura e numero di casi di test. Ogni volta che si eseguirà un algoritmo genetico al fine di cercare di coprire un obiettivo del *Control Flow Graph* nell'algoritmo relativo al nuovo approccio, si limiterà il numero totale di valutazioni a 20.000.

Nel caso specifico della **RQ1**, per far sì che la quantità di tempo sia uguale, si eseguirà preventivamente l'algoritmo relativo al nuovo approccio su ogni funzione da testare e si riporterà il tempo di esecuzione mediamente

impiegato per la generazione. Successivamente si userà quel tempo come limite massimo per l'algoritmo di generazione casuale sulle relative funzioni.

Nel caso della **RQ2**, si useranno i dati relativi alla copertura raccolti per rispondere alla **RQ1**: si userà come copertura-obiettivo la copertura media raggiunta dall'approccio Random. Si imposterà comunque un limite di casi di test per quest'ultimo di 1.000.000.

RQ3 Per rispondere alla terza domanda, si eseguira l'algoritmo relativo alla nuova metodologia su tutte le funzioni, registrando, per ogni caso di test generato, nell'ordine in cui vengono generati, la copertura (cumulativa) raggiunta. Si farà una distinzione tra la prima fase (ricerca dei casi di test per i cammini linearmente indipendenti) e la seconda fase (generazione dei casi di test per i singoli branch non coperti nella prima fase), al fine di evidenziare quanto ogni fase permette di aumentare la copertura. Infine, verrà utilizzato l'algoritmo greedy di minimizzazione della test suite previsto dalla metodologia (dettagliato nella sezione 3.2.3) e sarà riportato il numero di casi di test "inutili" generati dalla nuova metodologia. Per calcolare il numero di test superflui sarà sufficiente sottrarre la grandezza della test suite minimizzata alla grandezza della test suite non minimizzata.

5.3 Risultati

In questa sezione saranno presentati i risultati della sperimentazione, divisi per research question.

5.3.1 Livelli di copertura raggiunti (RQ1)

Il grafico in figura 5.1 mostra il livello di copertura sulle varie funzioni su cui è stata condotta la sperimentazione. Si può vedere chiaramente che il livello di copertura raggiunto dalla nuova metodologia è generalmente maggiore rispetto alla generazione di casi di test attraverso la metodologia casuale (in media del 7.1%). In alcuni casi, tuttavia (per la funzione `cliparc`), i livelli di copertura sono molto simili. Come si può immaginare, molto dipende dal

programma che si sta analizzando. In questo caso, come accennato precedentemente, il livello di copertura massimo raggiungibile è di poco superiore al 95%, ed entrambi gli approcci si avvicinano a questo risultato.

La ricerca casuale dà buoni risultati soprattutto se non ci sono condizioni particolarmente difficili da soddisfare all'interno della funzione da testare. In tutti gli altri casi, però, molti archi non possono essere coperti, se non con una probabilità molto bassa. Si consideri, ad esempio, il problema del triangolo: in questo caso, per testare l'arco che porta la funzione a dire che il triangolo è equilatero, è necessario che le tre variabili siano uguali. La probabilità che questo accada è $\frac{1}{|D|^2}$, dove D è il dominio di input. Se consideriamo gli interi da 0 a 999, la probabilità è pari a $\frac{1}{1000000}$, ovvero allo 0.0001%.

5.3.2 Dimensione della test suite (RQ2)

Il grafico in figura 5.2 mostra il numero di casi di test generati per le funzioni su cui è stata condotta la sperimentazione. È chiaro che, a parità di copertura, il numero di casi di test generati dall'approccio casuale è decisamente superiore: mediamente la metodologia proposta permette di ridurre il numero di casi di test totali del 54.9%. Questo risultato era atteso, poiché l'algoritmo casuale non fa una selezione dei casi di test utili/non utili, a differenza della nuova metodologia, la quale riduce decisamente il costo dell'oracolo.

Solo in un caso si può vedere che l'approccio casuale ha risultati migliori (funzione `gimp_rgb_to_hsl_int`): questo è dovuto al fatto che, in questo specifico caso, l'algoritmo casuale riesce a trovare molto velocemente i parametri per raggiungere una copertura basilare, poiché la maggior parte dei branch è molto semplice da coprire (a differenza della restante parte, che l'approccio casuale non riesce a coprire in tempi accettabili, come risulta evidente dalla figura 5.1)

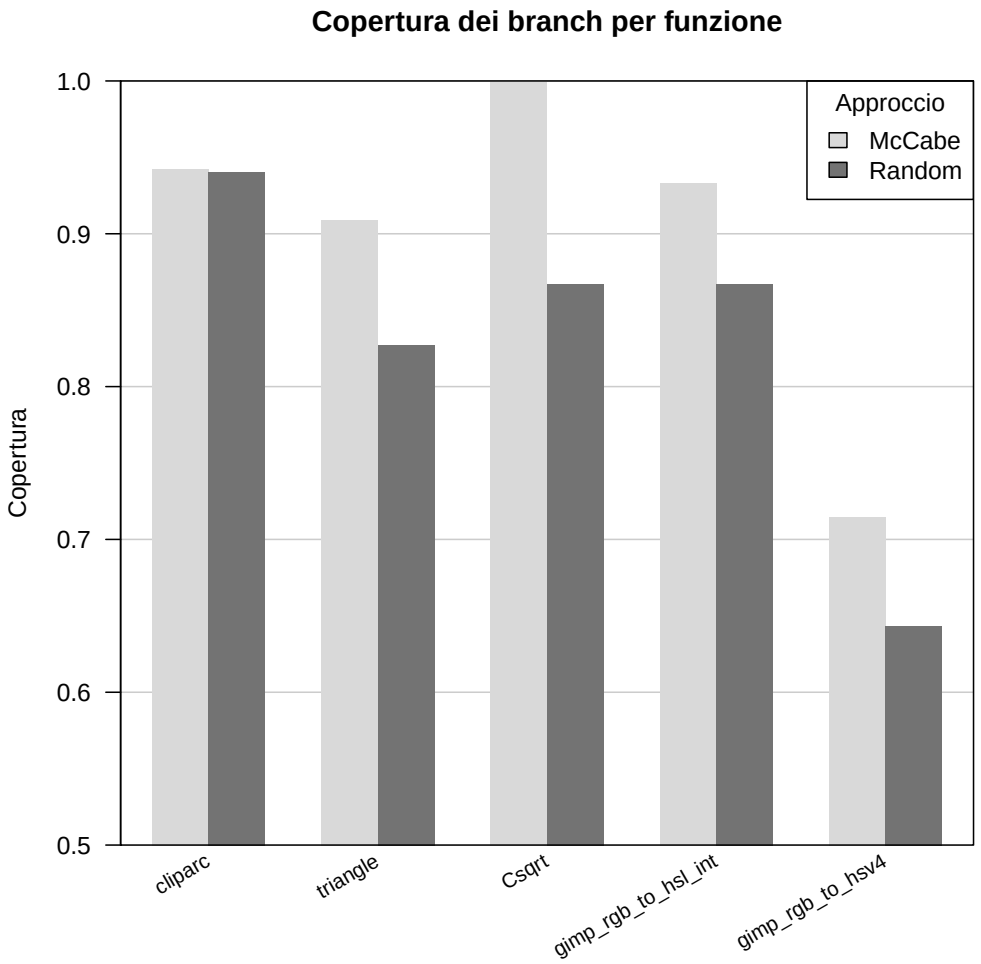


Figura 5.1: Branch coverage media utilizzando i due algoritmi di generazione automatica di casi di test.

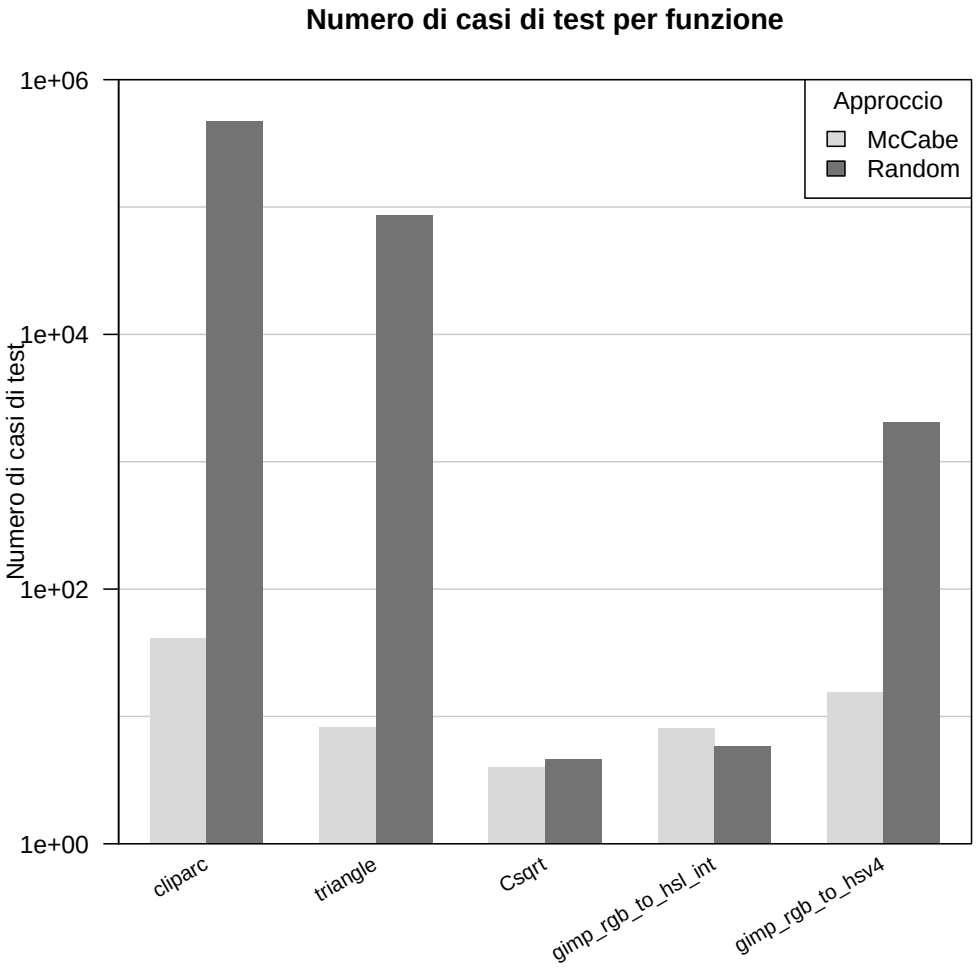


Figura 5.2: Numero medio di casi di test generati utilizzando i due algoritmi di generazione automatica di casi di test. Si è usata una scala logaritmica per mostrare al meglio i risultati.

	McCabe	Branch singoli	Minimizzazione
`cliparc`	28	43	13
`triangle`	8	12	5
`gimp_rgb_to_hsl_int`	8	12	4
`gimp_rgb_to_hsv4`	10	17	3
`Csqrt`	4	5	3

Tabella 5.2: Numero di casi di test ottenuti alla fine di ogni fase.

5.3.3 Contributo di ogni fase della nuova metodologia (RQ3)

I grafici in figura 5.3 mostrano la copertura complessiva del codice (sulle ordinate) al variare del numero di casi di test generati (sulle ascisse). La linea rossa verticale separa i casi di test generati a partire dai cammini linearmente indipendenti da quelli generati usando i singoli target. La selezione effettuata grazie ai cammini linearmente indipendenti permette di raggiungere un buon livello di copertura (mediamente il 93.8% della copertura totale raggiunta). Attraverso la selezione dei singoli obiettivi, tuttavia, si riesce a raffinare la copertura totale.

Per quanto riguarda il contributo della minimizzazione, la tabella 5.2 mostra il contributo di ogni passaggio nel modificare il numero totale di casi di test. La metodologia proposta genera una percentuale media di casi di test superflui del 63.4%. Questo indica che, sebbene si riesca a ridurre già a monte il numero di casi di test, la minimizzazione della test suite è un passaggio fondamentale al fine di ridurre il costo dell'oracolo. L'alta percentuale di riduzione del numero di casi di test è anche indice del fatto che si può fare ancora molto al fine generare un minor numero di casi di test e, quindi, ridurre (o distribuire in maniera migliore) il tempo totale di generazione.

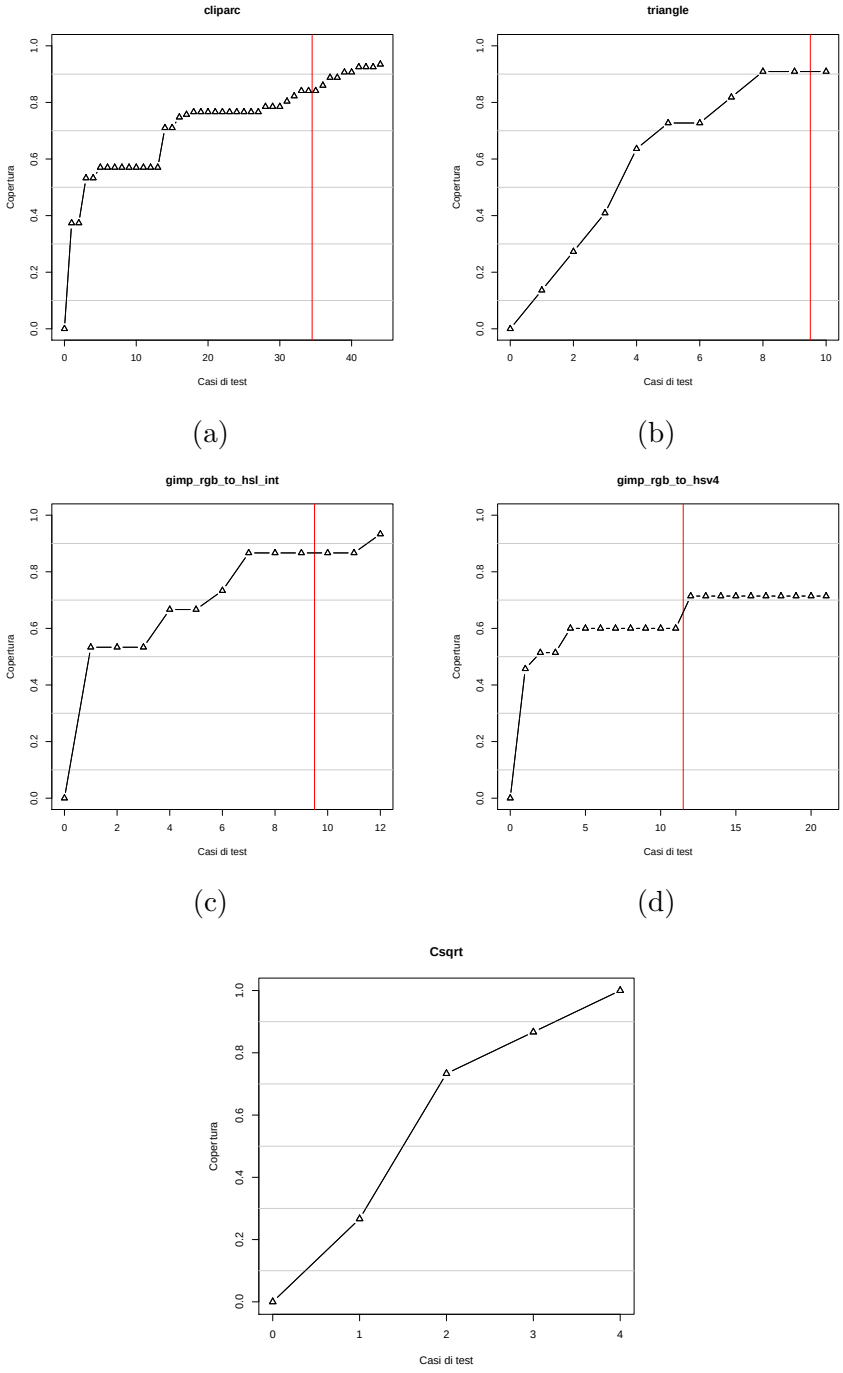


Figura 5.3: Copertura incrementale al variare del numero di casi di test generati. La linea rossa indica il punto in cui l'algoritmo inizia a considerare gli obiettivi singoli piuttosto che i cammini linearmente indipendenti.

Capitolo 6

Conclusioni e sviluppi futuri

In questo lavoro è stata presentata una nuova metodologia per la generazione automatica di casi di test, basata sui cammini linearmente indipendenti di McCabe [41]. Questa è composta da tre macro-passaggi principali:

- Ricerca dell'input che permette di coprire i cammini linearmente indipendenti di un dato programma
- Ricerca dell'input che permette di coprire i singoli archi non coperti nel passo 1
- Minimizzazione della test suite prodotta

L'ultimo passo, in particolare, ha lo scopo di ridurre il costo dell'oracolo [25], eliminando i casi di test che, in base al criterio di copertura scelto, sono superflui.

La metodologia è stata validata attraverso uno studio empirico: sono state scelte cinque funzioni C, tre delle quali provenienti da grandi progetti open-source, e sono state confrontate le performance del nuovo approccio con l'approccio casuale (che consiste nel generare i dati di test in maniera casuale).

Attraverso la nuova metodologia si raggiunge un livello di copertura superiore (in media del 7.1%) e si riesce a ridurre notevolmente il numero totale di casi di test (mediamente del 54.9%, pur non usando la minimizzazione).

Attraverso la minimizzazione, inoltre, è possibile ridurre in maniera significativa il numero totale di casi di test generati: in media il 63.4% dei casi di test generati sono superflui, quindi eliminati nell'ultimo passaggio.

Si è presentato, inoltre, un nuovo tool, OCELOT, sviluppato in Java, per la generazione automatica di casi di test per programmi C. Il tool implementa la metodologia proposta, ma è strutturato in modo tale da permetterne l'espansione.

Sono molti i possibili sviluppi futuri di questo lavoro. Innanzitutto è possibile provare ad utilizzare nuovi algoritmi di ricerca (in particolare il metodo delle variabili alternate, AVM, basato su hill climbing) per provare ad aumentare la copertura: è stato dimostrato, infatti, che spesso gli algoritmi genetici, seppur molto utilizzati in letteratura, non ottengono sempre i risultati migliori [19].

Data l'alta percentuale di casi di test superflui generati nei primi due passi della nuova metodologia, si può pensare di trovare un modo per ridurre questo numero: si potrebbe partire sempre dai cammini linearmente indipendenti (che sono una base a partire dalla quale è possibile generare tutti i cammini possibili sul Control Flow Graph) per cercare l'insieme più piccolo di cammini che permette di raggiungere la *branch coverage* totale.

Un altro sviluppo possibile potrebbe essere l'utilizzo dei cammini linearmente indipendenti per provare a risolvere il problema della bandiera, ponendo dei vincoli specifici che permettano di selezionare esattamente il percorso che comprende la modifica del flag e la condizione sul flag, in modo da dare una guida alla ricerca.

Bibliografia

[1] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[2] Giuliano Antoniol, Massimiliano Di Penta, and Mark Harman. A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 172–183. IEEE, 2004.

[3] Giulio Antoniol, Massimiliano Di Penta, and Mark Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 240–249. IEEE, 2005.

[4] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 205–214, 2010.

[5] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 108–118. ACM, 2004.

[6] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO*, volume 2, pages 1329–1336, 2002.

[7] André Baresel and Herr J Wegener. Automatisierung von strukturtests mit evolutionären algorithmen. *Master's thesis, Humboldt-Universität Berlin*, 2000.

[8] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, pages 106–115. IEEE Computer Society, 2004.

[9] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO*, volume 2, pages 1337–1342, 2002.

[10] Salah Bouktif, Houari Sahraoui, and Giuliano Antoniol. Simulated annealing for improving software quality prediction. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1893–1900. ACM, 2006.

[11] Kai H Chang, James H Cross II, W Homer Carlisle, and Shih-Sung Liao. A performance evaluation of heuristics-based test case generation methods for software branch coverage. *International Journal of Software Engineering and Knowledge Engineering*, 6(04):585–608, 1996.

[12] Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996.

[13] Lori Clarke et al. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[14] Ronan Cummins and Colm O Riordan. Term-weighting in information retrieval using genetic programming: A three stage process. *Frontiers in Artificial Intelligence and Applications*, 141:793, 2006.

[15] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *Lecture notes in computer science*, 1917:849–858, 2000.

[16] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, (4):438–444, 1984.

[17] Juan J Durillo and Antonio J Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011.

[18] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338. IEEE Computer Society, 2001.

[19] Robert Feldt and Simon Poulding. Broadening the search in search-based software testing: It need not be evolutionary.

[20] Filomena Ferrucci, Mark Harman, and Federica Sarro. Search-based software project management. In *Software Project Management in a Changing World*, pages 373–399. Springer, 2014.

[21] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, 2013.

[22] Michael R Garey and David S Johnson. Computers and intractability: a guide to the theory of np-completeness. 1979. *San Francisco, LA: Freeman*, 1979.

[23] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[24] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[25] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 182–191. IEEE, 2010.

[26] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer, 2012.

[27] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[28] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Reducing the cost of model-based testing through test case diversity. In *Testing Software and Systems*, pages 63–78. Springer, 2010.

[29] Dennis Jeffrey and Neelam Gupta. Test suite reduction with selective redundancy. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 549–558. IEEE, 2005.

[30] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[31] Bryan F Jones, H-H Sthamer, and David E Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[32] James C King. A new approach to program testing. In *ACM SIGPLAN Notices*, volume 10, pages 228–233. ACM, 1975.

[33] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[34] Bogdan Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990.

[35] Bogdan Korel. Automated test data generation for programs with procedures. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 209–215. ACM, 1996.

[36] Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of c programs. pages 112–125, 2013.

[37] William B Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, (99), 2013.

[38] Martin Lefley and Martin J Shepperd. Using genetic programming to improve software effort estimation based on general data sets. In *Genetic and Evolutionary Computation—GECCO 2003*, pages 2477–2487. Springer, 2003.

[39] Kiarash Mahdavi, Mark Harman, and Robert Mark Hierons. A multiple hill climbing approach to software module clustering. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 315–324. IEEE, 2003.

[40] Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *Software Engineering, IEEE Transactions on*, 29(11):974–984, 2003.

[41] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[42] Scott McMaster and Atif M Memon. Call-stack coverage for gui test suite reduction. *Software Engineering, IEEE Transactions on*, 34(1):99–115, 2008.

[43] Phil McMinn. Search-based software test data generation: a survey. pages 105–156, 2004.

[44] Phil McMinn. Iguana: Input generation using automated novel algorithms. a plug and play research tool. *Department of Computer Science, University of Sheffield, Tech. Rep. CS-07-14*, 2007.

[45] Phil McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

[46] Christoph C Michael, Gary E McGraw, Michael Schatz, Curtis C Walton, et al. Genetic algorithms for dynamic test data generation. In *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, pages 307–308. IEEE, 1997.

[47] A Jefferson Offutt, Jie Pan, and Jeffrey M Voas. Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw.* Citeseer, 1995.

[48] Mark O'Keeffe and Mel O Cinnéide. Search-based software maintenance. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10–pp. IEEE, 2006.

[49] Panichella. Improving multi-objective test case selection by injecting diversity in genetic algorithms. 2014.

[50] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *ICST*, pages 1–10, 2015.

[51] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4):263–282, 1999.

[52] Outi Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[53] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[54] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[55] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 1995.

[56] Smita Sampath, Renée Bryce, and Atif M Memon. A uniform representation of hybrid criteria for regression testing. *Software Engineering, IEEE Transactions on*, 39(10):1326–1344, 2013.

[57] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 97–106. ACM, 2002.

[58] Paolo Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, 2004.

[59] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 285–288. IEEE, 1998.

[60] Nigel Tracey, John Andrew Clark, and Keith Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *Proceedings of the IFIP International Workshop on Dependable Computing and Its Applications (DCIA).* York, 1998.

[61] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[62] Alison Lachut Watkins. The automatic generation of test data using genetic algorithms. In *Proceedings of the 4th Software Quality Conference*, volume 2, pages 300–309, 1995.

[63] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[64] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[65] S Xanthakis, C Ellis, C Skourlas, A Le Gall, S Katsikas, and K Karapoulios. Application of genetic algorithms to software testing. In *5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[66] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.

[67] Shin Yoo and Mark Harman. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689–701, 2010.

[68] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

Ringraziamenti

No one can whistle a symphony. It takes a whole orchestra to play it.

— HE Luccock

Difficilmente sarei riuscito a raggiungere questo traguardo senza l'aiuto, pratico e non solo, di tante persone.

Innanzitutto un grazie va al mio relatore, il Professor Andrea De Lucia, per l'estrema disponibilità e per la professionalità e precisione con cui mi ha seguito durante questo percorso e a Dario, per i continui consigli e le revisioni di questo lavoro.

Mi sembra doveroso, poi, dire grazie a Giovanni, Matteo e Carlo, i miei compagni di avventura all'Università di Salerno. Da un lato come gruppo ("Team Molise"), perché senza di loro, senza la loro preziosa collaborazione in tanti progetti, non sarei qui, ora, senza alcun dubbio; dall'altro come singoli: Giovanni, coautore di OCELOT e della metodologia, quindi fondamentale per questo lavoro; Carlo, per aver condiviso con me tutti i viaggi settimanali da Campobasso a Fisciano; Matteo, per la grinta e la determinazione che mi (e ci) ha sempre trasmesso.

È stato un privilegio poter lavorare con voi.

Un ringraziamento speciale a chi, da dietro le quinte, mi ha sempre incoraggiato e spinto a fare meglio, quindi alla mia famiglia, a Martina e a tutti i miei amici.

Infine ringrazio tutte quelle altre persone che, in un modo o nell'altro, hanno contribuito a rendere l'esperienza universitaria fantastica.