UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria

Corso di Studi in Ingegneria Informatica

Elaborato finale in **Ingegneria del software**

Modelli e tecniche per il testing delle GUI di sistemi software

Anno Accademico 2011-2012

Candidato:

Dario Scotto di Uccio

matr. N46/000642

Indice

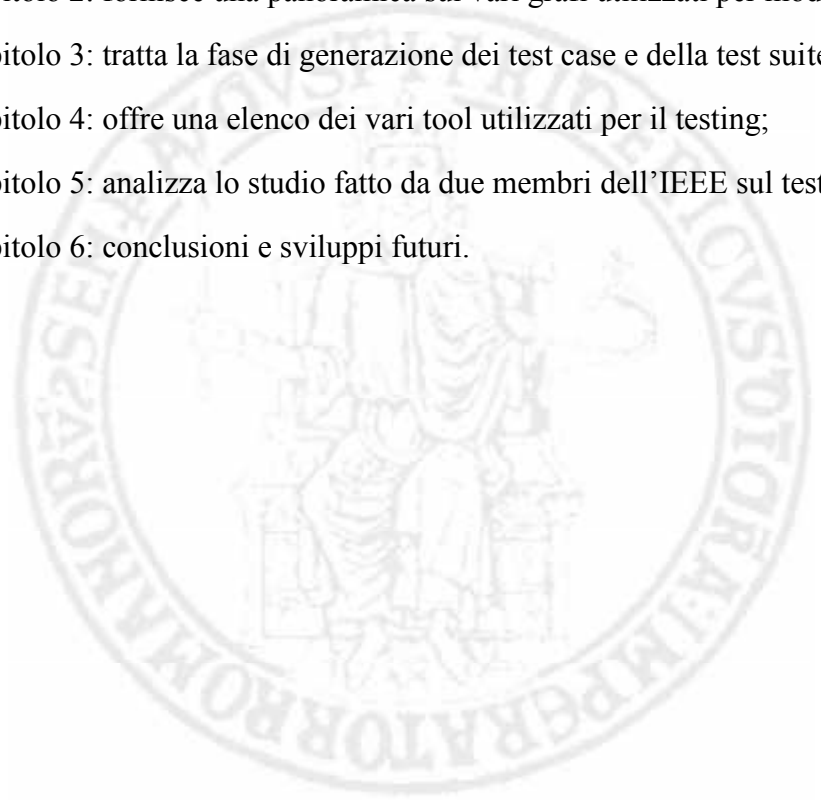
Introduzione

Nell'era moderna quasi tutti i software ed i sistemi operativi, sono dotati di una GUI (dall'inglese Graphical User Interface). La GUI è un tipo d'interfaccia utente che ci permette di interagire con tali ambienti. Nel caso dei sistemi operativi si parla di "Desktop environment", o semplicemente "Desktop". L'introduzione delle GUI ha rappresentato una svolta epocale per gli ambienti software. Infatti, prima della creazione delle stesse, qualsiasi interazione da parte dell'utente sia con i sistemi operativi, sia con le applicazioni, avveniva attraverso l'interfaccia a "linea di comando", la quale era composta esclusivamente da comandi testuali inseriti attraverso la tastiera. Il primo ambiente di lavoro ad essere dotato di un puntatore manovrato attraverso il mouse, fu progettato nei laboratori Xerox, e commercializzato per la prima volta nel 1981 con lo **Xerox Star**, il cui prezzo base si aggirava intorno ai 75.000 dollari. In seguito, lo sviluppo delle interfacce grafiche è stato ripreso anche dall'Apple nel 1983, ed ha portato alla creazione dell'**Apple Lisa.** La prima interfaccia grafica a colori è stata introdotta dall'**Atari** nel 1985. Le interfacce grafiche, hanno acquistato man mano sempre maggiore diffusione, divenendo parte integrante anche di altri sistemi operativi come Windows e Linux. Quest'ultimo tuttavia ha preferito non abbandonare l'interfaccia a linea di comando. Infatti, la shell grafica è posta al di sopra di quella testuale, ma non la sostituisce. L'importanza dell'introduzione delle GUI, è dovuta al fatto che grazie a loro, ora anche gli utenti meno esperti sono in grado di utilizzare un computer. Tutto ciò ha fatto in modo che, oggigiorno,

all'interno di ogni casa ci sia almeno un computer. Per questo motivo lo sviluppo della GUI nel processo di creazione del software, occupa una parte sempre crescente in termini di costi e quantità di lavoro. Una volta sviluppata, la GUI deve essere adeguatamente testata. L'obiettivo di questa tesi è di fornire al lettore, una panoramica sulle varie fasi di sviluppo del testing delle GUI e sulle tecniche utilizzate. In particolare, verrà focalizzata l'attenzione sui vari modelli di rappresentazione del software come ad esempio, il modello della macchina a stati finiti, oppure l'event semantic interaction graph. Successivamente, verrà trattata la fase di scelta dei test case da eseguire. Verranno inoltre presentati alcuni tool di testing, scelti tra i più utilizzati. Infine, analizzeremo lo studio di due membri dell'IEEE, che hanno mostrato la maggiore efficacia del testing eseguito in maniera automatica rispetto a quello manuale, in termini di rilevazione degli errori e copertura del sistema.

La tesi è strutturata come segue:

- Capitolo 1: un'introduzione sulla struttura e sul testing di una GUI;
- Capitolo 2: fornisce una panoramica sui vari grafi utilizzati per modellare la GUI;
- Capitolo 3: tratta la fase di generazione dei test case e della test suite;
- Capitolo 4: offre una elenco dei vari tool utilizzati per il testing;
- Capitolo 5: analizza lo studio fatto da due membri dell'IEEE sul testing automatico;
- Capitolo 6: conclusioni e sviluppi futuri.



Capitolo 1

Struttura e testing di una GUI

Gli elementi fondamentali delle GUI sono le finestre e i relativi widget, i quali, possono essere dotati di campi da settare e legati agli eventi che possono essere scatenati dall'utente. Una GUI contiene due tipi di finestre: le **Modal windows** e le **Modeless windows**. Le prime, una volta invocate, monopolizzano le interazioni con la GUI, focalizzando l'attenzione dell'utente sulle funzioni al suo interno, finché non vengono chiuse. Al contrario, le seconde non restringono l'attenzione dell'utente. Se durante l'esecuzione, una modal window ne richiama un'altra, la chiamante è detta "genitore". Nell'ingegneria del software, il GUI testing è un procedimento usato per testare l'interfaccia grafica del sistema al fine di assicurarsi che corrisponda alle specifiche dello stesso. Tali procedure sono tipicamente eseguite in modalità "black box". Lo sviluppo del test si basa sull'esecuzione di una varietà di test case. Per crearne uno efficace, gli sviluppatori devono essere sicuri di coprire tutte le funzionalità del sistema e l'intera GUI. Le difficoltà che s'incontrano nel cercare di soddisfare questi task sono trattare con la dimensione del dominio di appartenenza del software, nonché trattare con le sequenze di eventi e stati presenti nel sistema. Un'ulteriore difficoltà è introdotta dal testing di regressione, che rappresenta la riesecuzione di un test, una volta effettuata una correzione. Rispetto ad una **CLI** (dall'inglese Comand Line Interface), una GUI ha un numero maggiore di operazioni da testare. Ad esempio, in un programma semplice come il WordPad ci sono 325 possibili operazioni all'interno della GUI. Molte tecniche di testing

sono state progettate a partire da quelle utilizzate per le CLI. Una di queste comporta l'uso di un Planning system, che è una tecnica appartenente al dominio dell'AI. Un'altra tecnica ripresa da quelle del testing delle CLI è Capture/Playback. Questa si basa sulla cattura e l'immagazzinamento di varie istantanee dello schermo raccolte durante l'esecuzione del testing di sistema. Tali istantanee vengono poi confrontate con degli screen di riferimento. La rilevazione delle differenze può essere problematica in quanto ci potrebbero essere delle differenze di stile, come ad esempio una diversa collocazione degli oggetti sullo schermo o diversi colori per le finestre, pur mantenendo il medesimo contenuto di base. Per risolvere questo problema, si è pensato di operare direttamente sulle interazioni tra la GUI e il sistema sottostante, eliminando in tal modo l'insidia delle differenze stilistiche.

In questo modo vengono catturati i flussi di eventi, i quali devono poi essere filtrati, datosi che non tutti sono rilevanti. Per migliorare la qualità del testing, si possono utilizzare delle tecniche automatiche finalizzate all'eliminazione degli errori degli utenti. I maggiori benefici derivanti dai test automatizzati sono: a) un maggiore livello di copertura – b) una maggiore affidabilità – c) cicli di test ridotti – d) la possibilità di effettuare test multi utente senza costi aggiuntivi. Tutto questo, porta ad un aumento della qualità del prodotto, che si traduce in una maggiore richiesta sul mercato.

1.1 Testing automatico

Le tecniche di generazione di test case automatici (le ATCG), stanno riscuotendo grandi consensi, e ciò a causa della loro riduzione dei costi del testing seguito da una maggiore qualità. Un tipico approccio usato per queste tecniche, è quello di creare un modello astratto dell'applicazione sotto test, come ad esempio il modello a stati, ed utilizzarlo poi per generare i test case. Alcuni ricercatori hanno riscontrato che questi task possono essere aiutati dallo sfruttamento dei risultati dei test case già esistenti. Di conseguenza hanno sviluppato delle tecniche automatiche basate sul feedback. Queste tecniche richiedono un insieme di test, creati manualmente o automaticamente, ed eseguiti sul software. I risultati di tali test, sono usati per migliorare il modello preliminare e generare automaticamente

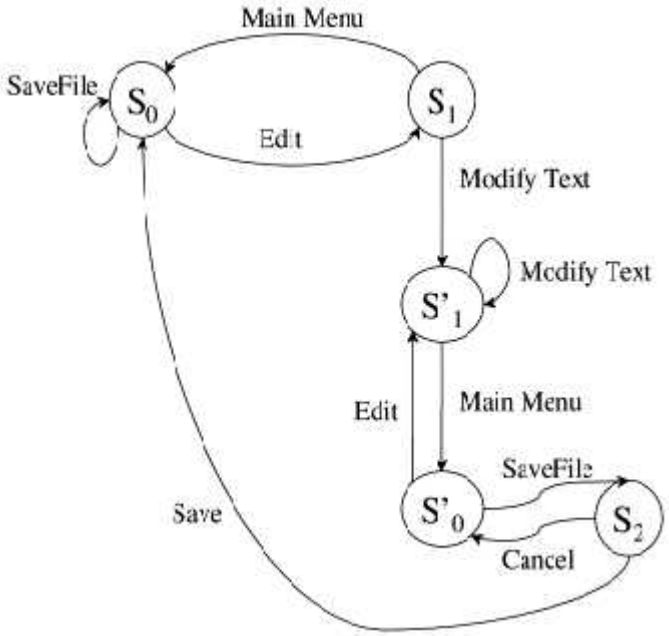
altri test case. La natura dei risultati dipende dall'obiettivo del test. Ad esempio, i risultati di un test di copertura, sono utilizzati per generare dei test che aumentano la copertura del sistema. Alcune tecniche automatiche ricavano i feedback dagli stati del sistema a tempo di esecuzione. Tale approccio è diffuso per varie motivi di utilità. Il primo è che i test case completamente automatici della GUI (basati sui modelli esistenti), ricoprono in maniera esaustiva le interazioni "**two-way**" tra gli eventi. Quest'ultimi sono definiti **Smoke test.** Questo tipo di test è eseguito in fase preliminare, e serve a identificare errori semplici ma che possono mettere a rischio il rilascio di una release del software. Un sottoinsieme di test case, che copre le funzionalità più importanti di un componente o di un intero sistema, viene selezionato ed eseguito, per accertare che tali funzioni siano attuate correttamente. Lo scopo è di determinare lo stato dell'applicazione. In quanto se essa risulta non funzionante, ulteriori test sono inutili. Questa procedura può essere catalogata sia come test funzionale, sia come test di unità. La seconda ragione è che i tool esistenti sono utili per il monitoraggio e l'archiviazione degli stati raccolti a tempo di esecuzione. Infine, la GUI rappresenta circa la metà del codice del software, quindi un testing di qualità migliore diminuisce la possibilità di errori ed aumenta la vendibilità del prodotto. Un altro tipo di test automatico è il **Monkey Test**. Esso si basa sull'immissione di input casuali, in modo da poter coprire ogni possibile interazione da parte dell'utente con l'interfaccia. Vi sono vari tipi di Monkey test: **Smart Monkey Testing, Brilliant Monkey Testing, Dumb Monkey Testing**. Nel primo gli input sono generati da distribuzioni di probabilità che riflettono le statistiche reali dell'utilizzo previsto. Ci sono diversi livelli di testing, cioè, un test richiede un vettore di almeno cinque componenti, nel quale troviamo gli ingressi. Nel livello più semplice ogni ingresso è considerato indipendente dagli altri. Nel livello più alto viene presa in considerazione la correlazione tra gli ingressi. In ogni caso però ogni ingresso è considerato come un singolo evento. Il Brilliant Monkey testing, invece, si basa sul modello della macchina a stati. In questo caso non solo la sequenza di input, ma anche la sequenza di stati a cui gli input portano è scelta basandosi su modelli probabilistici. Infine, per quanto riguarda il Dumb Monkey Testing, gli ingressi sono generati a partire da una distribuzione uniforme senza tener conto delle statistiche di utilizzo effettivo. Un test

suite generato automaticamente, come ad esempio uno smoke test, utilizza un modello "Event-interaction-graph (EIG)", che rappresenta tutte le possibili sequenze di eventi che possono essere eseguiti all'interno della GUI. Durante l'esecuzione del test, vengono raccolti i widget, i quali successivamente sono utilizzati per identificare automaticamente una relazione "Event Semantic interaction (ESI)" tra coppie di eventi. Queste relazioni mostrano come un evento è legato ad un altro, in termini di come ne modifica il comportamento a tempo di esecuzione. Il comportamento a run-time è valutato in termini di proprietà dei widget. Le relazioni ESI sono utilizzate per costruire automaticamente un nuovo modello, "Event Semantic interaction graph (ESIG)". Siccome il test suite è generato a partire dall'EIG (un modello strutturale), e le relazioni ESI sono ottenute in termini di esecuzione di eventi (un'attività dinamica), l'EISG cattura alcuni aspetti sia strutturali sia dinamici della GUI. Inoltre, l'EISG è utilizzato per generare automaticamente nuovi test case. Questi hanno un'importante proprietà, ogni evento ha una relazione ESI con l'evento successivo.

Capitolo 2

Modelli del sistema

Per analizzare le sequenze di eventi si possono utilizzare varie tecniche**: a) modello della macchina a stati – b) AI planning – c) algoritmi genetici – d) modelli probabilistici – e) diagrammi dell'architettura**. Tutte queste tecniche possono essere usate per generare diversi tipi di test case in diversi domini e sono tutti basasti su modelli creati manualmente. I modelli sono delle astrazioni del comportamento del software da una particolare prospettiva. Ci possono essere differenti livelli di astrazione: stati astratti, stati della GUI, variabili di stato interne, percorsi di funzioni. Il modello più utilizzato è quello della macchina a stati. Nell'immagine a sinistra abbiamo un esempio tale modello. Di solito è rappresentato come un diagramma di transizione degli stati, e descrive il comportamento del software in termini di stati, siano essi astratti o concreti. Ad una sequenza di input corrisponde una sequenza di transizioni di stato. L'insieme di queste transizioni rappresenta un cammino

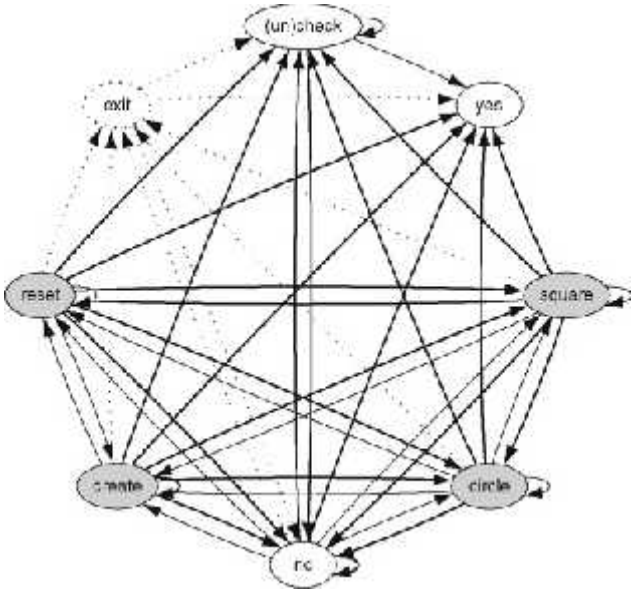


all'interno del modello. Un modello a stati può essere ricavato sia in fase di sviluppo dell'applicazione, sia utilizzando un Reverse Engineering partendo dall'UI implementata. Alcuni tipi dei seguenti modelli sono utilizzati per il testing del software**: finite state machine model (FSM), UML diagram based model, Markov chains**. Alcune estensioni dell'**FSM** utilizzano variabili per rappresentare dei contesti, in aggiunta agli stati. Ad esempio **l'extended finite state machine (EFSM)**, fa uso di uno stato di dati e degli input per la transizione da uno stato all'altro. Questo tipo di modello è utilizzato da un tool chiamato **TestMaster** per generare dei test case che attraversano tutti i percorsi dallo stato iniziale a quello finale. Molti ricercatori utilizzano il modello della macchina a stati per il testing degli "**event driven software (EDS)** ". Per questo tipo di software, i test case sono sequenze di eventi. Gli EDS sono modellati in termini di stati e gli eventi rappresentano le transizioni tra gli stati. Alcuni algoritmi attraversano questi modelli per generare le sequenze di eventi. Ad esempio Campbell ha utilizzato il modello della macchina a stati per testare i sistemi reattivi orientati agli oggetti. Gli stati, che in questo caso sono oggetti, sono modellati in termini di valori d'istanze di variabili. Le transizioni sono ottenute dall'invocazione dei metodi. I test case sono sequenze di chiamate di metodi e, sono generate dall'attraversamento del modello.

2.1 Event flow graph e Event interaction graph

Per ridurre il lavoro manuale, sono state sviluppate delle nuove tecniche sistematiche basate sui modelli delle GUI. Queste tecniche si basano su due tipi di grafo, in particolare: "l**'event flow graph (EFG)** " e "**l'event interaction graph (EIG)** ". Un event flow graph contiene tutti gli eventi ai quali è possibile accedere in un determinato istante. Quando i componenti modeless della GUI sono attivi, è possibile accedere ai loro eventi in contemporanea con quelli dei componenti modal. Invece, un event interaction graph è ottenuto automaticamente usando un algoritmo di reverse engineering sulla GUI. Al suo interno sono rappresentati solo due tipi di eventi: terminazione e interazioni di sistema. Gli eventi di terminazione chiudono le modal windows. Gli altri non manipolano la struttura

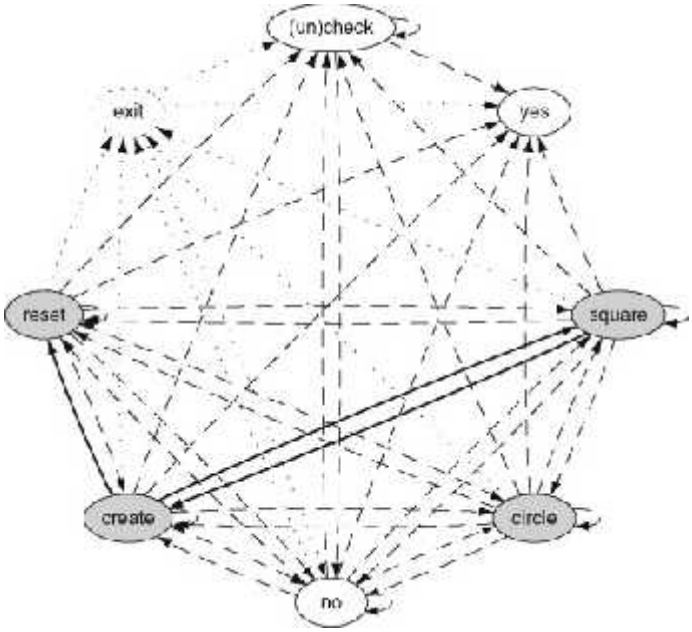
della GUI. Gli archi orientati tra i nodi codificano l'esecuzione dei percorsi. Analizziamolo ora nel dettaglio. Come mostra la figura, i nodi sono gli eventi e gli archi orientati rappresentano relazioni di sequenzialità tra essi. Ogni cammino, rappresenta una sequenza di eventi, e corrisponde a un possibile test case. Il susseguirsi di eventi può creare un percorso estremamente lungo, se non illimitato, quindi la copertura fornita dall'EIG è relativa al grado di profondità al quale ci si arresta. Siccome i nodi dell'EIG non rappresentano eventi di apertura o chiusura di menu o finestre, la sequenza ottenuta non può essere eseguita. A tempo di esecuzione sono generati automaticamente altri eventi per raggiungere quelli dell'EIG e formare così un test case eseguibile. Per permettere una terminazione pulita dell'applicazione, ogni test case è fornito di un evento che chiude tutte le modal windows prima che il test case termini.



2.2 Event semantic interaction graph

Le tecniche che utilizzano il feedback si basano sull'identificazione di un set di eventi, che devono essere testati in interazioni multi way. Si può avere un'identificazione approssimativa attraverso l'analisi dei feedback ottenuti dagli stati di una GUI presi a run-time in un test suite iniziale. Attraverso uno smoke test suite è abbastanza facile testare tutte le interazioni two way tra tutte le coppie di eventi. Le modal windows creano delle situazioni speciali dovute alla presenza di eventi di terminazione. Questo perché, le azioni di un utente all'interno di una modal windows, non provocano l'immediato cambio di stato. Infatti, le azioni hanno effetto solo dopo che l'evento di terminazione **TERM** è stato eseguito. Quindi, ogni stato viene memorizzato solo dopo l'esecuzione di tale evento.

Problemi simili sorgono quando due eventi si trovano in due modal windows differenti, oppure quando uno si trova in una modal window e l'atro in una modeless window, o anche quando uno si trova in una modal windows e l'altro nella parent window.



All'interno dell'ESIG, come mostra la figura, i nodi rappresentano gli eventi, un arco orientato da un nodo all'altro, mostra che esiste una relazione tra gli eventi dei nodi. I test suite basati sull'ESIG sono in grado di rilevare un numero maggiore di errori rispetto agli altri. In particolare, questo confronto si basa su tre parametri: raggiungibilità, manifestazione e numero di test case. La raggiungibilità è definita come la copertura delle istruzioni all'interno delle quali è presente un errore. La manifestazione è definita come il difetto che porta a un fallimento visibile, in modo che possa essere identificato. Questi due parametri sono necessari per il processo d'individuazione degli errori. La maggiore efficacia nel rilevare gli errori, è dovuta alla natura delle relazioni ESI che si basano sugli stati della GUI e quindi sull'output del software. Infatti, eseguendo test che si focalizzano esclusivamente sugli eventi ESI, aumentano le probabilità che un errore sia individuato.

Capitolo 3

Generazione dei test case

Una volta costruiti i modelli, si passa alla generazione dei test case. Esistono diverse tecniche per la generazione dei test case per le GUI, e tutte utilizzano un modello del software. A partire dal modello esistono alcun tecniche usate per la generazione dei test case: **a) la tecnica basata sugli stati – b) AI planning – c) algoritmi genetici – d) il GUI ripping**. La tecnica basata sugli stati, si fonda appunto sul modello della macchina a stati, dove ad ogni evento scatenato dall'utente, corrisponde una transizione di stato. Uno dei problemi nella generazione dei test case è dato dalla presenza di un grande numero di possibili stati ed eventi presenti nella GUI. Sono stati fatti diversi tentativi per risolvere questo problema. Ad esempio, Shehady, utilizzò una "**Variable Finite State Machine (VFSM)** ", che introduceva all'interno di un FSM delle variabili globali, che potevano assumere un numero finito di valori durante l'esecuzione di un test case. Il valore di ogni variabile era usato per determinare il prossimo stato e la risposta ad un evento. Tra le tecniche utilizzare per ovviare al problema, troviamo anche l'AI planning. Questa tecnica si basa su quattro parametri: stato iniziale, stato finale, un insieme di operatori e di oggetti su cui operare. In questo caso, un tester crea manualmente una descrizione della GUI. La quale sottoforma di pianificatore delle operazioni, è utilizzata per modellare le precondizioni e le post condizioni legate ai ogni evento della GUI. I test case sono generati automaticamente, a partire dalla coppia stato iniziale e finale, invocando un pianificatore che cerca un percorso dallo stato iniziale a quello finale. Un'altra tecnica molto utilizzata è

quella che sfrutta gli algoritmi genetici. I test case che sono generati usando tali algoritmi simulano le azioni di un utente principiante. Questi algoritmi sono chiamati così perché, ispirandosi alla selezione naturale, durante ogni iterazione del programma esegue una selezione tra le possibili soluzioni. Quest'approccio prevede la presenza di un utente esperto che genera manualmente una sequenza di eventi, poi si utilizza la tecnica genetica per generare delle sequenze più lunghe. Si sfrutta l'assunzione che un utente esperto per lo svolgimento di un task utilizzi un percorso diretto, mentre un principiante ne utilizza uno indiretto più lungo. Infine, abbiamo il GUI ripping. Questi è un processo dinamico, utilizzato per esplorare automaticamente la GUI, analizzando i widget che compongono l'interfaccia. Esso ci permette, inoltre, attraverso un algoritmo esplorativo automatico guidato dagli eventi scatenati sulle interfacce, di ottenere uno schema ad albero della GUI. L'algoritmo attraversa tutta la struttura, e una volta che ha aperto tutte le finestre, estrae i widget, le proprietà e i valori degli oggetti della GUI. Questi, una volta estratti, vengono verificati e infine utilizzati per generare dei test case automatici.

3.1 Generazione dei test case basati sul feedback

I feedback sono informazioni che si ottengono durante l'esecuzione dei test, e utilizzate per la generazione di nuovi test case. Questa procedura è chiamata, generazione di test dinamica. In questa tecnica il codice sorgente del software è utilizzato per ottenere i feedback dall'esecuzione. L'intera procedura parte eseguendo un test iniziale. I feedback sono memorizzati e analizzati, i risultati sono utilizzati per valutare secondo alcuni criteri quanto siano simili ai risultati desiderati. Vengono poi apportate delle modifiche al modello per cercare di raggiungere tali risultati. A questo punto si esegue un nuovo test case e si riconfrontano i risultati. La procedura termina quando le condizioni sono soddisfatte. Molti ricercatori hanno utilizzato lo stesso principio per la generazione di test dinamici. Xie, ad esempio, ha sviluppato un framework che utilizza i feedback sottoforma di stati oggetto e astrazioni operazionali, cioè un rapporto sugli stati del programma a run-time. Questo frame work integra la generazione dei test basati sulle specifiche formali con

le specifiche dinamiche. La generazione dei test, basata sulle specifiche formali, rappresenta il comportamento desiderato del programma. Siccome tali specifiche sono difficili da ottenere, le specifiche dinamiche cercano di dedurle direttamente dall'esecuzione del software attraverso le astrazioni operazionali. Invece, Pacheco ha generato un'unità di test casuale incorporando i feedback dagli input dei test eseguiti in precedenza. Egli costruì in modo incrementale un insieme di ingressi, selezionando casualmente le funzioni da testare.

3.2 Sviluppo di un test suite

Un test suite è una collezione di test case utilizzati per testare il software, esso mostra un insieme di comportamenti del programma. Inoltre, contiene informazioni dettagliate dei test case e sulla configurazione del sistema sotto test. La copertura degli eventi e degli archi dell'EIG, oltre al numero di test case, occupa un ruolo fondamentale per stabilire l'efficacia di un test suite nell'individuazione di errori. Per esempio, un test suite che ricopre poche linee di codice individuerà pochi errori. Per giudicare l'efficacia di un test suite, bisogna che abbia la stessa copertura di eventi, di archi e numero di test case di un test suite basato sull'ESIG, per poter fare poi un confronto. Un altro problema, da considerare durante lo sviluppo di un test suite, è la copertura di determinate linee di codice. A causa dei vari livelli di astrazione tra gli eventi della GUI e il codice, bisognerebbe esaminare il codice sorgente, le relazioni tra gli eventi e il codice sottostante, per poi distribuire gli eventi nei vari test case per assicurarsi di coprire il codice specifico. Questo processo, però, non è automatico e quindi , la sua realizzazione può portare a una grande mole di lavoro. Siccome i criteri di cui sopra possono essere soddisfatti da un gran numero di test suite, con efficacia variabile, il processo di generazione e confronto dei test suite ha bisogno di essere ripetuto diverse volte. Esistono alcune tecniche che si basano proprio sulla copertura del codice. Gli obiettivi di tali tecniche sono la copertura di uno specifico percorso e la copertura delle condizioni e decisioni. Per generare un test case che segua un percorso assegnato, si utilizzano degli approcci iterativi. Uno di questi, è

l'approccio della **discesa a gradini**, il quale corregge gradualmente un test case affinché esegua il percorso stabilito. Gli algoritmi genetici sono poi stati utilizzati per generare automaticamente dei test case che soddisfino i criteri delle condizioni e delle decisioni. Ogni condizione deve essere vera per almeno un test case e falsa per almeno un test case. Si definisce una funzione per ogni ramo di decisione, che sono utilizzate per valutare la validità per ogni test case.

3.3 L'oracolo

L'oracolo è un meccanismo, fondamentale nello sviluppo del testing, che determina se l'esecuzione dell'applicazione durante un test case è andata a buon fine. Il fallimento di un test può essere dovuto a un crash oppure a dei comportamenti indesiderati da parte dell'applicazione. Infatti, l'oracolo confronta i comportamenti dell'applicazione durante un test con quelli desiderati ricavati dalle descrizioni della GUI. Ci sono vari modi per creare tali descrizioni. Il primo consiste nell'usare le specifiche formali della GUI che sono poi usate per creare automaticamente l'oracolo. Il secondo è usare un tool di cattura e riproduzione per creare manualmente delle asserzioni sull'oracolo del test, e utilizzare tali asserzioni come oracolo per il test delle altre versioni del software. Il terzo metodo consiste nello sviluppo di un oracolo da una versione del software detta "Golden", ed usarlo per testare una versione dell'applicazione all'interno della quale sono presenti errori. I primi due approcci, richiedono una mole di lavoro molto elevata, al contrario il terzo metodo può essere eseguito in maniera del tutto automatica. In un test basato sul modello della macchina a stati, l'oracolo confronta gli stati evidenziati dal test con quelli presenti nel modello. Invece, in un test che si basa sull'EIG, partendo da un evento, l'oracolo confronta quello successivo all'interno del test con quello presente nel grafo. Uno dei tool utilizzati per i confronti è **jDiffChaser,** il quale automatizza l'individuazione di differenze di comportamento tra le esecuzioni. Si possono memorizzare e riprodurre scenari di due diverse versioni della stessa applicazione in maniera sequenziale o parallela. Alla fine della comparazione, viene compilato un rapporto dettagliato delle differenze, il tutto comprensivo di immagini.

Capitolo 4
Tool di testing

Alcuni tool di testing delle GUI sono stati progettati per uno specifico ambiente, altri si adattano a qualsiasi sistema operativo e dominio di applicazione. Nel corso di questo capitolo verrà fornita una panoramica su alcuni di questi tool e su i rispettivi domini applicativi. Iniziamo dalle applicazioni web. Qui troviamo i tool: **Appperfect e Tellurium.** Il primo permette di distribuire il test su più macchine e simulare le condizioni di funzionamento del mondo reale, ed è utilizzato maggiormente per network applications, applicazioni client-server e altre applicazioni multi-livello. Tellurium, invece, è una tecnica di testing automatica per applicazioni web che utilizza gli oggetti per incapsulare elementi dell'UI . Un modulo dell'UI è semplicemente un oggetto formato da altri oggetti nidificati. I test sono scritti in **Groovy** oppure in **Java.** Le sue funzioni principali sono: test operanti sui moduli dell'UI, mappatura degli oggetti da localizzare e linguaggio di dominio specifico (DLS) per la definizione dei moduli dell'UI. Questi sono utilizzati per definire il codice HTML trovato nella pagina, utilizzando poi il DLS riesce a identificare gli oggetti presenti nella pagina e le loro relazioni. L'algoritmo **Santa** (chiamato così perché è stato realizzato durante il periodo natalizio del 2009) utilizzato nelle versioni 0.7.0 e successive, aumenta la robustezza dei test localizzando tutti i moduli dell'UI a tempo di esecuzione. Nella precedente versione 0.6.0, i riscontri basati sui moduli dell'UI erano generati a tempo di esecuzione. Poi però erano passati al **Selenium** che si occupava dell'individuazione di ogni singolo elemento. Selenium è un software che fornisce tool di

testing per diversi tipi di linguaggio. Un'altra categoria di tool è quella legata ad un particolare tipo di applicazioni. Ad esempio, ci sono dei tool che sono stati sviluppati appositamente per applicazioni Java. Tra questi abbiamo: **Abbot Java GUI Test Framework, Jemmy, Marathon** e **Pounder.** Il primo fornisce una generazione di eventi e la validazione per componenti delle GUI Java, in maniera automatica, migliorando le funzioni rudimentali fornite dalla classe java.awt.Robot. Questi può essere invocato direttamente dal codice Java oppure è accessibile senza programmazione utilizzando degli script. Questa tecnica permette di memorizzare ogni azione dell'utente all'interno di uno script che controlla la produzione e il testing degli eventi. Jemmy è utilizzato per la creazione di test automatici. I test sono scritti in Java e usano Jemmy come una normale libreria. A differenza degli altri tool, si concentra unicamente sulla stabilità del test e per questo può essere usato per applicazioni dinamiche di grandi dimensioni. Marathon invece, è uno strumento di testing utilizzato per le piattaforme Java nelle versioni 1.3 e successive. Per aiutare il test di regressione, memorizza eventi come ad esempio, le interazioni con l'applicazione. Questi sono poi convertiti in test validi, che possono essere in seguito riprodotti. Infine, abbiamo Pounder, anch'esso utilizzato per lo sviluppo di test case automatici. Questo tool consente agli sviluppatori di caricare, dinamicamente, record di script della GUI, per poi usarli in un Test Harness (che è un insieme formato da un software e dai dati ricavati dai test, usati per testare un programma in esecuzione sotto varie condizioni, monitorandone il comportamento e gli output). La differenza sostanziale rispetto agli altri è che permette di esaminare i risultati di un test eseguito in precedenza, pur mantenendo uno script separato dell'UI che può essere registrato nuovamente, se necessario. Esistono poi dei tool legati ad un particolare sistema operativo. Ad esempio, abbiamo: **Autonet** e **Dogtail**, **e**ntrambi legati al sistema operativo **Linux.** Autonet è una piattaforma di test per le GUI, interamente basato su CLI. Esso aiuta a organizzare i test case, i comandi di configurazione dei dispositivi, ed eseguire i comandi per controllare e memorizzare i risultati dei test. Dogtail invece, è un tool di testing open source scritto in **Python**, che come detto è basato su Linux ed è presente in distribuzioni come **Fedora**. Esso utilizza tecnologie di accessibilità, come ad esempio **AT-SPI** (**Assistive Technology**

Service Provider Interface), per comunicare con le applicazioni del desktop. Inoltre, utilizza l'accessibilità ai metadati per creare un modello in memoria degli elementi dell'interfaccia grafica delle applicazioni. Occupiamoci ora dei tool di testing che si adattano a vari ambienti. Tra questi abbiamo: **Cobra – Winldtp, Eclipse Jabula, Maveryx, QAliber, LoadRunner** e **Sikuli.** Cobra – Winldtp è un di tool di testing che utilizza librerie di accessibilità per attivare i widget dell'interfaccia utente. Genera una "Appmap" che individua quali widget, sono usati dall'applicazione. Può anche registrare vari test case osservando cosa un utente clicca. In seguito, utilizza l'appmap ed i test case immagazzinati per testare l'applicazione. È compatibile sia con Windows che con Linux, ed è utilizzata per testare programmi come: **Notepad, VMware Workstation, VMware Player, Mozilla, Openoffice** e applicazioni **GNOME.** Maveryx è un tool di testing automatico per le applicazioni Java su Windows, MAC e Linux, utilizzato anche nei testing regressivi. A differenza degli altri tool, non necessita di una GUI map per effettuare il test automatico. Gli oggetti della GUI sono riconosciuti e localizzati a tempo di esecuzione. Durante l'esecuzione di un test esegue una scansione dell'UI in cerca di oggetti e di controlli da manipolare. Durante la scansione sono prelevate una serie d'istantanee dall'UI, le quali sono poi utilizzate per riconoscere e localizzare gli oggetti. Con quest'approccio non c'è bisogno di aspettare l'inizio del test di scrittura dell'applicazione. In questo modo, i tester possono sviluppare gli script automatici nelle prime fasi del ciclo di vita del software, in parallelo con lo sviluppo dell'applicazione. Gli oggetti sono identificati univocamente attraverso una vasta gamma di algoritmi tra i quali: **Extract, Approximate matching** e **Fuzzy Matching**. Questi ci permettono di individuare la corrispondenza migliore tra l'oggetto sotto test e quello presente nello script della GUI, anche in caso di poche informazioni sull'oggetto su cui operare. Il Maveryx supporta testing Data-driven, punti di verifica e Keyword per incrementare la copertura del test e promuovere il riuso degli script. I keyword driven sono test automatici sviluppati come tabelle di dati. Ogni riga contiene la keyword che sarà usata come ingresso o uscita prevista. Maveryx inoltre genera automaticamente metriche e fornisce rapporti dettagliati sui risultati dell'esecuzione dei test. QAliber è un set di tool di testing automatico per

applicazioni web e desktop, nel sistema operativo Windows. Esso è composto a sua volta da due tool: **QAliber Test Developer** e **QAliber Test Builder**. La metodologia alla base di questi tool è che lo sviluppatore crea, utilizzando il Developer, piccoli Building blocks, come le unità di test, per coprire i test di sistema. Questi sono poi assemblati per creare un complesso scenario di test usando il Builder. Eclipse Jabula fornisce test automatici per le applicazioni Java e HTML. È utilizzato dagli sviluppatori che vogliono che i loro test automatici siano scritti dal punto di vista dell'utente. Garantisce la manutenibilità dei test a lungo termine. LoadRunner usa degli script per eseguire azioni come se fosse un utente reale. Esegue test di carico e stress test per ambienti Windows e Unix, e può essere utilizzato per misurare le risposte dei sistemi. Questo strumento può simulare migliaia di utenti virtuali per localizzare i "colli di bottiglia del sistema". In questo modo i problemi di prestazione possono essere affrontati prima del rilascio del software. Sikuli, infine, è un tool che automatizza il testing della GUI utilizzando delle istantanee dell'applicazione sotto test. Può essere utilizzato per il testing di una pagina web o di un software per sistemi operativi come Windows, MAC OX e Linux, ed anche per applicazione di smartphone con sistema operativo MAC o Android, attraverso un simulatore.

Capitolo 5

Studio condotto sul testing automatico

Andiamo ora ad analizzare degli studi condotti da **Xun Yuan**, Ingegnere del software specializzato in testing, alla Google di Kirkland, e da **Atif M. Memon**, professore associato dell'università del Maryland. L'argomento affrontato riguarda le tecniche di testing per le interazioni multi - way tra gli eventi di una GUI, eseguite in maniera automatica e basate sul feedback. All'interno di tale processo, gli script necessari per settare, eseguire e distruggere i test case sono stati implementati ed eseguiti senza l'intervento umano. Sono stati condotti due studi indipendenti sulle GUI di otto applicazioni Java, per valutare e comprendere questo nuovo approccio. Nel primo studio hanno utilizzato quattro applicazioni già testate. Esso ha dimostrato che le tecniche basate sul feedback migliorano quelle esistenti con piccoli costi aggiuntivi. Le relazioni ESI hanno identificato con successo interazioni complesse tra gestori di eventi della GUI che hanno portato a seri fallimenti. Fallimenti che non erano stati individuati dalle precedenti tecniche. Questi fallimenti sono consultabili online sul sito https://sourceforge.net/tracker/?func=detail&atid=535427&aid=153607&group_id=72728. Con questo studio, si è dimostrato che gli sviluppatori delle applicazioni non hanno individuato tali errori perché i loro tool di testing non erano in grado di testare automaticamente e complessivamente le applicazioni. Il secondo studio condotto dai due membri dell'IEEE, si basa sul testing di quattro applicazioni Java, all'interno delle quali sono stati "seminati" degli errori. Questo studio mostra che l'identificazione automatica

delle relazioni ESI tra eventi aiuta a identificare un numero maggiore di errori. Gli errori inseriti, sono stati sviluppati in modo da non poter essere rilevati dalle tecniche precedenti, ma solo da quelle nuove. La mancata localizzazione di alcuni errori è dovuta ai limiti dell'oracolo. Per altri invece, la localizzazione richiedeva una sequenza di eventi troppo lunga. Gli aspetti principali del loro studio sono:

- La generazione di test case automatici, basati sui modelli;
- La definizione di nuove relazioni tra gli eventi della GUI basati sulla loro esecuzione;
- L'utilizzo degli stati presi a tempo di esecuzione, per utilizzare un insieme di input più ampio e aumentare la localizzazione degli errori;
- La dimostrazione dell'efficacia del testing automatico basato sul feedback;
- La presentazione di prove empiriche, che legano le caratteristiche dei fallimenti con il tipo di test suite.

Entrambi gli studi si focalizzano su una classe di GUI che accetta eventi discreti scatenati da un singolo utente. Gli eventi sono deterministici, e quindi, il risultato è conosciuto. Una GUI appartenente a questa classe è formata da una serie di widget W, ogni widget w appartenente a W, ha una serie di proprietà $P_{w,}$ e ogni proprietà p, ha un valore univoco stabilito attraverso una funzione di settaggio dei valori V_p . Quindi lo stato di una GUI in qualsiasi istante è un insieme di valori w, p, v. Nel primo studio sono stati esaminati solo i fallimenti dei test dovuti ai crash. L'implementazione di un processo di crash testing include la creazione di un database con campi testuali per i valori. Siccome l'intero processo doveva essere completamente automatico, il database conteneva un'istanza per ogni tipo di valore, che poteva essere settato. Tali valori potevano essere ad esempio: "**negative number**", "**real number**", "**empty string**", "**exisisting file name**". Per questo studio i due membri dell'IEEE hanno scelto quattro applicazioni già testate e immesse in commercio. Queste sono: **CrosswordSage 0.3.5, FreeMind 0.8.0, GanttProject 2.0.1, JMSN 0.9.9b2**. Dopo aver scelto le applicazioni da studiare, hanno eseguito su di esse un tool di testing statico chiamato **FindBugs** per individuare i bug del sistema. Al termine dello studio i ricercatori verificarono che nessuno dei bug individuati dal testing

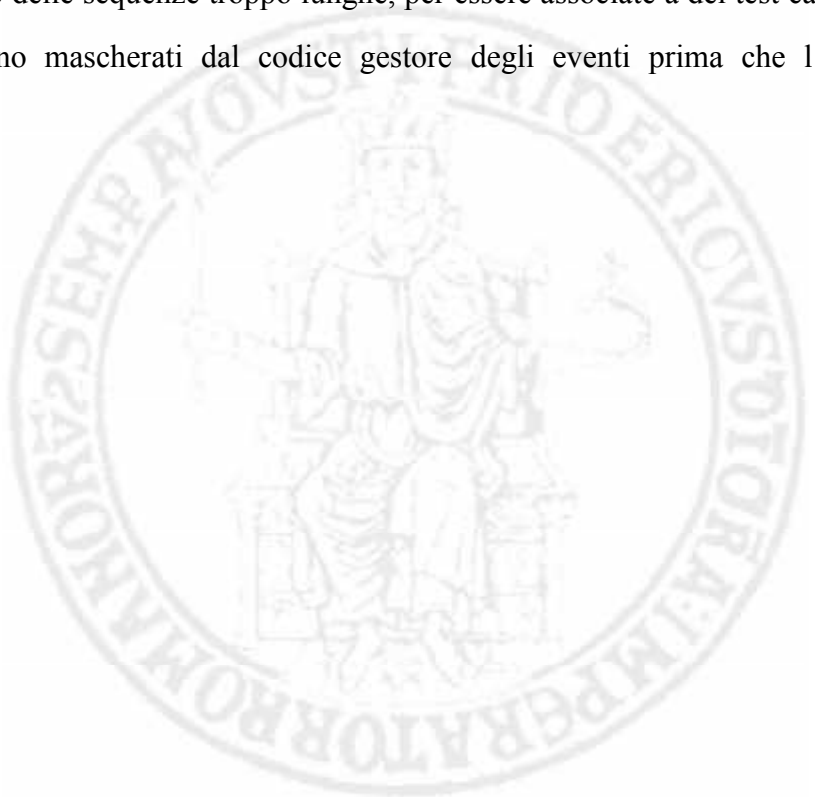
automatico era stato individuato in precedenza. Attraverso il reverse engineering, hanno ricavato l'EIG di ogni applicazione. L'intero processo di test è stato eseguito senza l'intervento umano generando un elevato numero di test case, per ogni applicazione. E per ognuna di esse ci sono stati dei test che hanno provocato dei crash. Una volta individuate le cause dei fallimenti, hanno eseguito un'operazione di correzione. Quest'operazione è stata necessaria per due motivi. Il primo è che questi errori avrebbero causato dei fallimenti anche nei test successivi rendendoli inutili. Il secondo motivo è che nella realtà un bug viene corretto, una volta localizzato. Siccome però la correzione di un bug può causare altri errori all'interno dell'applicazione, una volta sistemati i bug trovati vengono rieseguiti i test. Studiando le relazioni ESI di ogni applicazione, gli autori hanno mostrato che alcuni eventi, chiamati dominanti, partecipano ad un numero di relazioni molto maggiore rispetto agli altri. Il secondo studio condotto serve per compare le tecniche di testing basate sul feedback, con le altre tecniche. I termini di paragone erano: copertura degli eventi, degli archi e numero di test case. L'efficacia delle tecniche fu misurata in termini di fallimenti individuati. Le applicazioni che selezionarono per questo studio furono: **Paint, Present, SpreadSheet,** e **Word.** All'interno di queste applicazioni furono "seminati" alcuni errori. Questo richiese l'accesso al codice sorgente, al rapporto dei bug e alla documentazione sulla storia dello sviluppo. I tool utilizzati in questo studio sono stati implementati in Java. Per evitare l'interazione tra gli errori e per semplificare la mappatura dei fallimenti dell'applicazione, gli autori hanno creato diverse versioni di ogni applicazione. In ognuna di esse hanno inserito un solo errore. Un test case individuava un fallimento se c'era una discrepanza tra l'esecuzione della versione della GUI in esame e quella Golden. Il confronto fu eseguito attraverso l'utilizzo dell'oracolo. I valori delle proprietà di tutti i widget della GUI erano mostrati dopo ogni evento. Il processo di "semina" degli errori era formato da una fase iniziale, in cui scelsero dodici classi di errori conosciuti. Una volta scelte le classi, presero alcune istanze di ognuna e le introdussero nelle parti del codice sorgente coperte dallo smoke test. Questo garantiva che le istanze sarebbero state eseguite. Gli errori sono stati scelti in modo da somigliare a quelli che di solito occorrono nei programmi reali, dovuti ad errori commessi dagli sviluppatori oppure

ad errori dovuti a input errati immessi dal'utente. Attraverso un processo di reverse engineering applicato alla versione originale del software, gli autori hanno ricavato l'EIG. All'aumentare di nodi ed archi presenti nel grafo corrispondeva un aumento del numero di test case da generare. L'EIG è stato usato per generare tutti i possibili test case two-way. Il numero di test generati è uguale al numero di archi, ed erano circa 80.000. L'esecuzione di questi test case è durato circa un giorno impiegando 50 macchine in parallelo. Successivamente i test two-way e gli stati della GUI furono utilizzati per ottenere tutti i possibili test di copertura three-way dell'ESIG. Visto il minor numero di nodi ed archi dell'ESIG, anche il numero di test relativo è minore.

	Paint	Present	Spreadsheet	Word
Fallimenti totali	263	265	234	244
Fallimenti individuati con i test Two-way basati sull'EIG	147	139	139	183
Fallimenti individuati con i test Three-way basati sull'ESIG	47	52	39	36

I fallimenti riportati nell'ultima riga della tabella, sono quelli che non erano stati individuati dai test two-way. Quindi i test basati sui modelli ESIG sono in grado di individuare un numero di fallimenti maggiore. Per individuare il test suite con maggiore efficacia nell'individuare gli errori, gli autori hanno generato 700 test suite per ogni applicazione e confrontato i loro risultati con quelli dell'ESIG. L'esecuzione di un test case può durare fino a due minuti. I test case totali dello studio erano 3.739.431, e ogni test case doveva essere eseguito su ciascuna versione del software. Usando 50 macchine in parallelo, ci sarebbero voluti alcuni anni per completare il lavoro. Per ovviare a questo problema utilizzarono processo meccanico randomizzato. In quest'approccio ogni comando era eseguito isolatamente e i test case erano assemblati concatenando i comandi in differenti permutazioni. Utilizzarono poi il modello EIG per generare solo sequenze eseguibili. Inoltre generarono i test case in lotti di lunghezza crescente, misurata in termini di numero di eventi dell'EIG. Ogni arco dell'EIG doveva essere coperto da almeno N test case di un lotto. Inoltre, ogni errore seminato doveva essere coperto da almeno M test case

dell'intera suite. La scelta di N e M è stata dettata dalle risorse a disposizione, visto che ogni test case doveva essere eseguito su ogni versione dell'applicazione. In particolare, N è stato scelto pari a 10 e M pari a 15. Anche con 50 macchine che lavoravano in parallelo, il processo è durato 4 mesi. I risultati furono immessi in delle matrici. In particolare, è stata creata la **matrice degli errori**, che riassume gli errori rilevati e i relativi test case, e la **matrice di copertura**, che riassume gli elementi coperti, come eventi o archi, per ogni test case. Grazie a queste matrici, si ottiene un test suite che offre una copertura adeguata di ogni applicazione. In seguito i due membri dell'IEEE hanno confrontato i risultati ottenuti con altri test suite che forniscono una copertura simile. Da questo confronto è emerso che il suite basato sull'ESIG, rileva un numero maggiore di errori. Tuttavia alcuni difetti non sono stati rilevati. Gli autori hanno esaminato questi errori uno a uno, cercando di portali alla luce manualmente. Questo li ha portati a capire alcune cose. La prima è che alcuni errori si manifestano, ma l'oracolo automatico non è in grado di esaminare le parti della GUI in cui si trovavano. La seconda è che alcuni errori causano fallimenti nelle parti della GUI non visibili e quindi non possono essere rilevati. La terza, è che alcuni errori richiedono delle sequenze troppo lunghe, per essere associate a dei test case. Infine, alcuni errori erano mascherati dal codice gestore degli eventi prima che l'oracolo potesse rilevarli.



Capitolo 6

Conclusioni e sviluppi futuri

Nel presente elaborato si è discusso sulle varie fasi di sviluppo de testing delle GUI. Precisando che esso occupa una parte sempre maggiore nello sviluppo di un software, in quanto l'interfaccia grafica è ciò che l'utente vede realmente e con cui interagisce. Quindi un'interfaccia grafica migliore, porta ad un apprezzamento maggiore del software da parte dell'utente. Questo si traduce, in termini economici, in un aumento delle vendite del prodotto. Siamo partiti dai vari grafi utilizzati per modellare il sistema, tutti basati sulle sequenze di eventi. In seguito abbiamo esaminato la fase di generazione dei vari test case, e ai relativi parametri di scelta. Successivamente è stato affrontato il problema della raccolta dei test case in test suite, scelti in base alla copertura del sistema offerta, al numero di errori individuato ed al numero di test case eseguiti. Poi, abbiamo analizzato alcuni tool di testing catalogati in base all'ambiente di utilizzo, osservando che alcuni di essi operano solo su applicazioni, od in sistemi operativi specifici. Altri tool si adattano a vari tipi di software ed a vari sistemi operativi. Infine, grazie allo studio di due membri dell'IEEE, abbiamo osservato come il testing eseguito utilizzando tecniche automatiche offra una copertura maggiore del sistema ed un maggior numero di errori rilevati. Possibili sviluppi futuri potrebbero essere incentrati sulla creazione di nuove tecniche automatiche, in grado di testare anche delle sequenze di eventi molto lunghe in un tempo non eccessivo. Inoltre, si può cercare di creare oracoli migliori, in grado di rilevare un numero sempre crescente di errori. È parere dello scrivente che il futuro del testing delle GUI risieda

proprio nell'evoluzione delle tecniche automatiche, in quanto, con costi aggiuntivi trascurabili, riducono la mole di lavoro degli sviluppatori e aumentano la qualità del software.

Bibliografia

[1] Kanglin Li, Mengqi Wu 2005 "Effective GUI Test Automation: Developing an Automatic GUI testing tool".

[2] Xun Yuan, Atif M Memon IEEE Transacion on software engineering, vol X, NO. X, 2010 "Generating Event Sequence-Based Test Cases Using GUI Run-Time State Feedback".

[3] http://www.appperfect.com/products/application-testing/app-test-gui-testing.html

[4] http://www.opensourcetesting.org/functional.php

[5] http://www.testingfaqs.org/t-gui.html

[6] http://www.webdesignerdepot.com/2009/03/operating-system-interface-design-between-1981-2009/

[7] http://www.guidebookgallery.org/