UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

Tesi di laurea in Ingegneria del Software

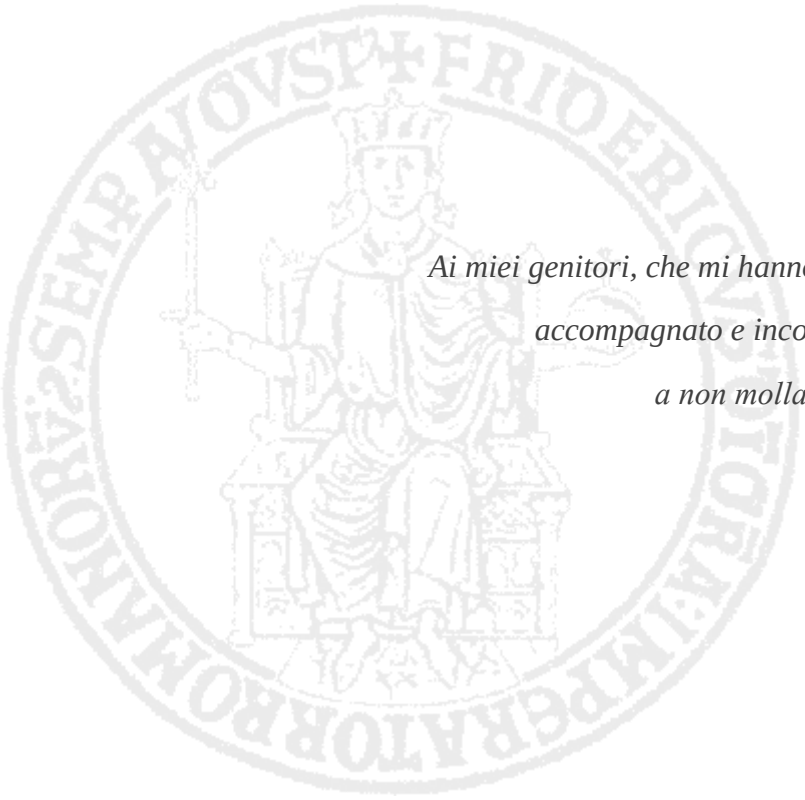
TECNICHE DI RIDUZIONE DELLE TEST SUITE PER APPLICAZIONI ANDROID

Anno Accademico 2013/2014

Relatore
Ch.mo Prof. Porfirio Tramontana

Correlatore
Ing. Nicola Amatucci

Candidato
Giuseppe Graziuso
Matr. 534/777



Ai miei genitori, che mi hanno sempre

accompagnato e incoraggiato

a non mollare mai…

Grazie!

Indice

Introduzione

I dispositivi mobili, al giorno d'oggi, rivestono un ruolo sempre più importante sia nel lavoro aziendale sia nella nostra vita privata, permettendoci di compiere molte operazioni e svolgere dei compiti che, fino a qualche anno fa, erano eseguibili solo attraverso un normale Personal Computer. Infatti l'evoluzione tecnologica che ha accompagnato proprio lo sviluppo dei Personal Computer, ha coinvolto anche tali dispositivi, trasformandoli da semplici organizer "da tasca" a veri e propri terminali ricchi di funzionalità, discreta potenza di calcolo e soprattutto di connettività. Esistono diversi tipologie di dispositivi mobili: Pocket PC, Smarthphone, Tablet PC e Ultra Mobile PC. Per poter sfruttare al meglio le potenzialità di tali dispositivi sono stati introdotti e sviluppati diversi sistemi operativi per dispositivi mobili quali: Android, Symbian Os, Windows Phone, etc. Ma uno dei punti di forza è stato sicuramente la possibilità di espandere le funzionalità di tali dispositivi tramite applicazioni per dispositivi mobili (o semplicemente mobile apps), che sono state sviluppate già circa dieci anni fa, ma la loro diffusione è aumentata in maniera esponenziale a partire dall'apertura della piattaforma Apple App Store avvenuta nel 2008, la quale nel 2013 ha raggiunto lo storico traguardo dei cinquanta miliardi di download [1]. Successivamente sono comparse altre piattaforme digitali destinate a fornire software per dispositivi mobili; alcune gestite dal proprietario del sistema operativo del dispositivo, altre da terze parti, tra cui organizzazioni commerciali come Amazon. Tali piattaforme hanno avuto un grandissimo successo, soprattutto per la loro facilità di utilizzo ed infatti

con esse gli utenti possono scegliere e scaricare moltissime applicazioni disponibili, sia gratuitamente che a pagamento. La richiesta di apps è in continuo aumento. Inizialmente le applicazioni mobili erano state offerte per la produttività, il recupero e la gestione di informazioni, compresa la posta elettronica, il calendario, contatti e informazioni meteo. Tuttavia, la domanda pubblica e la disponibilità di strumenti hardware sempre più potenti ha spinto l'espansione anche in altre categorie, come i giochi per cellulari, GPS e servizi di localizzazione, bancario, acquisto di biglietti e recentemente applicazioni mediche mobili.

Naturalmente il consumatore non si limita a richiedere delle specifiche apps per eseguire determinate operazioni o per un determinato utilizzo, ma richiede anche che queste apps funzionino correttamente, cioè richiede delle applicazioni di alta qualità ed è proprio questo uno degli obiettivi primari, a cui si dedica maggior impegno e particolare attenzione durante lo sviluppo di questi software. In particolare una fase importantissima a garantire tale risultato è quella del testing e della sua automazione. Per soddisfare i requisiti di qualità il testing deve avvalersi di adeguati strumenti, tecniche e strategie per realizzare test efficaci, efficienti e comprensibili.

I dati e i sondaggi relativi all'utilizzo delle mobile apps nel mondo degli affari e in quello aziendale fa prevedere che la richiesta non può che continuare ad aumentare. Ormai queste applicazioni sono indispensabili poiché forniscono un mezzo, per poter comunicare e condividere file con i colleghi di tutto il mondo, per poter accedere a postazioni remote; strumenti quindi che migliorano e semplificano il lavoro e la produttività. La maggior parte delle applicazioni mobile sono software di piccoli dimensioni; ideate, progettate e sviluppate da un piccolo team. Il team lavora in tempi brevi adottando opportuni strumenti di sviluppo. Poi ci sono le applicazioni mobile più grandi e più complesse la cui realizzazione richiede più tempo e specifiche tecniche dell'Ingegneria del Software.

Attualmente, ma in realtà da più di qualche anno, Android è il sistema operativo per dispositivi mobili più diffuso e utilizzato [2]; i suoi punti di forza sono soprattutto il fatto che si tratta di un sistema operativo Open Source e la sua sconfinata compatibilità con un'infinità di dispositivi e sistemi. Inoltre attraverso indagini di marketing è stato possibile constatare che i consumatori che utilizzano il sistema operativo Android su un loro

dispositivo, ad esempio uno Smartphone, se devono acquistare un ulteriore dispositivo, ad esempio un Tablet, sono più propensi ad acquistare un dispositivo che abbia lo stesso sistema operativo. Questo perché il consumatore per entrambi i dispositivi potrà usare la stessa interfaccia, e non solo, ma potrà condividere anche la stessa esperienza di utilizzo, così come anche applicazioni, giochi ed impostazioni, mirando quindi a una semplicità di utilizzo, a un sistema già noto e alla compatibilità.

Naturalmente obiettivo degli sviluppatori delle applicazioni Android per confermare e ampliare il successo della loro piattaforma è garantire: l'efficacia, l'efficienza e l'affidabilità delle loro apps.

Capitolo 1

Android e applicazioni mobile per Android

1.1 Un po' di storia

Android inizialmente fu sviluppato nel 2003 da una piccola azienda californiana di nome Android, Inc., fondata da Andy Rubin, Rich Miner, Nick Sears e Chris White. Il principale ideatore fu Rubin il cui obiettivo era quello di sviluppare "*dispositivi cellulari più consapevoli della posizione e delle preferenze dei loro proprietari*". Inoltre la sua idea fu quella di creare un sistema operativo aperto, basato su Linux, conforme agli standard, con un'interfaccia semplice e funzionale, che mettesse a disposizione degli sviluppatori strumenti efficaci per la creazione di applicazioni. E soprattutto, a differenza di tutti gli altri dispositivi sul mercato, il suo utilizzo doveva essere gratuito.

Nel 2005 l'azienda venne acquistata dalla Google Inc. diventando la Google Mobile Division. Nello stesso anno l'ormai celebre azienda costituì un'alleanza con: i maggiori operatori telefonici come Vodafone, Telecom Italia, T-Mobile; produttori di dispositivi mobili come Motorola, HTC, Samsung; produttori di semiconduttori come Intel e Nvidia. Il loro scopo era quello di definire uno standard per l'open source dei dispositivi mobili. Nel 2007 l'Open Handset Alliance (OHA) viene istituita ufficialmente e presenta la prima versione di Android. Qualche giorno dopo verrà rilasciato anche il primo Software Development Kit (SDK) per gli sviluppatori, che include: gli strumenti di sviluppo, le librerie, un emulatore del dispositivo, la documentazione, tutorial e altro. Al momento del

lancio, Android presentava poche applicazioni: una rubrica e un calendario sincronizzati con Gmail, un browser basato su Webkit, e poco altro. Ciò nonostante in pochi anni si è affermato come uno dei sistemi operativi per dispositivi mobili più diffuso al mondo e a favorire questo oltre alle sue caratteristiche è stata la scelta di un sistema "aperto", open source appunto, dando la possibilità a moltissimi programmatori di poter sviluppare applicazioni per tale sistema [3]. La diffusione di tale sistema su scala mondiale è stata accompagnata dalla necessità di rendere il sistema e i software che lo accompagnano (le mobile apps appunto) più stabili, affidabili, efficienti, e più sicuri. E per ottenere questo è stato necessario lo sviluppo di appositi strumenti a supporto della fase di testing del software. Per capire in che modo eseguire il testing delle applicazioni e quali strumenti sono necessari per eseguire correttamente la fase di testing è utile conoscere l'architettura del sistema operativo Android e capire in che modo è strutturata e funziona una generica applicazione.

1.2 L'architettura di Android

Android [4] non è una piattaforma hardware, ma un ambiente software progettato appositamente, inizialmente per i telefonini, ma più in generale per dispositivi mobili. Android è uno stack software, cioè un set di sottosistemi software, basato su kernel Linux e che è composto da applicazioni Java eseguite su uno speciale framework, anch'esso basato su Java e orientato agli oggetti, a sua volta eseguito su un nucleo costituito da librerie Java eseguite tramite una macchina virtuale, Dalvik. I vari elementi architetturali possono essere riassunti nei seguenti punti:

- A livello più basso abbiamo il cuore del sistema, basato sul kernel di Linux (versione 2.6). Esso agisce da layer di astrazione fra l'hardware sottostante (che comprende il GPS, la fotocamera, il touchscreen, il wifi) e il resto dello stack software. Tale livello di astrazione è importante perché i vari produttori di telefonini possono già intervenire a questo livello per personalizzare i driver di comunicazione dei loro dispositivi e anche perché i livelli sovrastanti non si accorgono dei cambiamenti hardware,

permettendo quindi al programmatore e all'utente un'esperienza indipendente dal device;

- Al livello superiore abbiamo tutta una serie di librerie che consentono di gestire diversi elementi, come ad esempio: la grafica 2D e 3D (OpenGL ES), il database (SQLite) il browser integrato (WebKit);
- L'ambiente di runtime include un set di librerie base (Core Libraries) che fornisce la maggior parte delle funzionalità e una macchina virtuale (Dalvik Virtual Machine) progettata per operare su hardware non performanti; insieme costituiscono la piattaforma di sviluppo di Android;
- Al penultimo livello abbiamo l'Application Framework, che mette a disposizione degli sviluppatori tutta una serie di componenti con cui è possibile realizzare un'applicazione. Ci sono gestori per le telefonate, per le risorse, per le applicazioni installate e molto altro ancora;
- Infine a livello più alto risiedono le applicazioni utente, cioè le applicazioni base del dispositivo mobile.

La figura seguente mostra appunto tali componenti del sistema operativo Android.

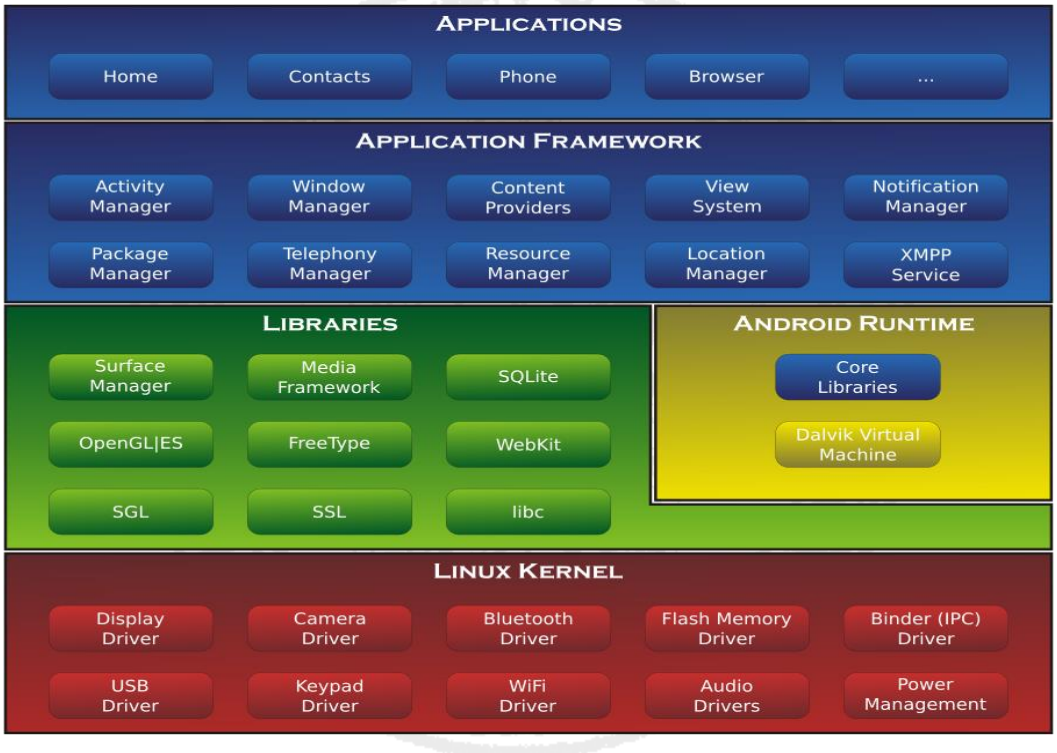


Figura 1.1: Architettura di Android

1.3 Le applicazioni di Android

Uno degli aspetti centrali di Android è che un'applicazione può far uso di elementi di altre applicazioni (se quest'ultime lo permettono). Infatti diversamente da molti altri sistemi operativi, le applicazioni Android non hanno un unico entry point, ma sono composte da componenti che il sistema può istanziare e avviare secondo le necessità. Questi componenti sono di quattro tipi:

- Activity;
- Service;
- Broadcast Receiver;
- Content Provider;

Il più importante tra questi quattro tipi di componenti è l'Activity.

1.3.1 Activity

L'*Activity* [5] è il componente di Android incaricato di visualizzare l'interfaccia utente e di gestire l'iterazione dell'utente con tale interfaccia. Essa rappresenta una singola schermata di un'applicazione; quindi non è errato pensare ad un'*Activity* come la schermata che è visualizzata in un determinato momento sul dispositivo. Solitamente un'applicazione di Android ha più *Activity*; generalmente ha almeno un'*Activity* (la *main Activity*), quella principale, che è quella visualizzata quando viene lanciata l'applicazione e tale *Activity*, (ma in realtà qualsiasi altra *Activity*) può chiamare a sua volte altre *Activity* in seguito allo scatenarsi di eventi, che possono essere generati dall'iterazione dell'utente, dal sistema o dall'ambiente software a seguito di eventi hardware. Ogni *Activity* può avviarne altre per eseguire specifiche azioni, il sistema conserva l'*Activity* precedente per poterla poi riprendere una volta che l'*Activity* chiamata è terminata. In pratica possiamo considerare, in un caso semplice, un'**Activity chiamante** (che può essere la *main Activity*) che chiameremo **Activity1** e l'**Activity chiamata** che chiameremo **Activity2**. Quando l'**Activity1** chiama l'**Activity2**, il sistema provvede a memorizzare l'**Activity1** in una pila

detta "*Back Stack*" per non perderla e riprenderla successivamente, questa *Activity* passa così nello stato di stop o pausa; invece l'**Activity2** viene inserita in cima alla pila e diventa l'*Activity* in esecuzione.

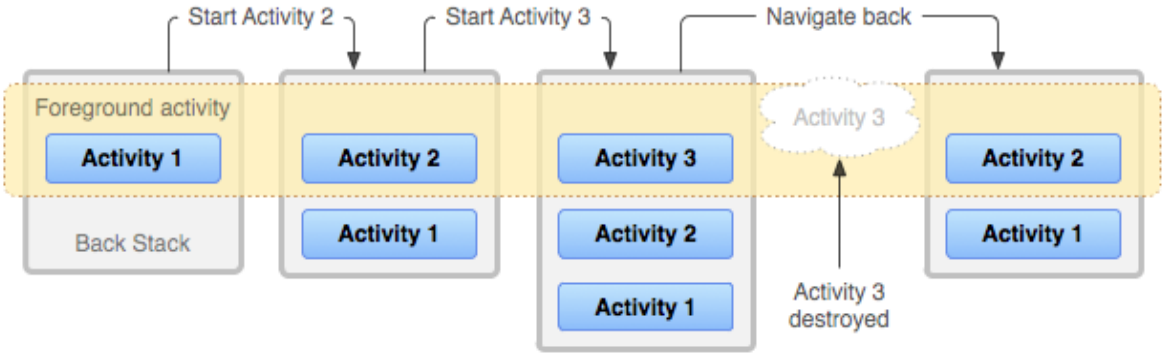


Figura 1.2: Back Stack

Naturalmente questo procedimento può essere iterato considerando casi più complessi con più *Activity*. L'**Activity1** ritornerà nello stato di esecuzione solo quando l'**Activity2** termina, anche se in realtà l'utente può tornare all'*Activity* precedente tramite la pressione del tasto **BACK** del dispositivo, motivo per il quale tale pila è indicata col nome di "*Back Stack*"; tale operazione inoltre provvede a distruggere l'evento in esecuzione (**Activity2**). Quando un'**Activity** viene distrutta essa viene eliminata dalla memoria del dispositivo e non potrà più essere ripristinata. Un'altra operazione particolare può essere causata dalla pressione del tasto **HOME**, mediante la quale si ritorna sempre alla schermata iniziale del dispositivo, però l'*Activity* in esecuzione non viene distrutta, ma viene solo sospesa e può essere ripresa in qualsiasi momento.

E' importante sottolineare che, sebbene le *Activity* lavorano insieme e offrono un'interfaccia uniforme, ognuna di esse è indipendente dalle altre. Ogni *Activity* è visualizzata all'interno di una *window*. Tipicamente la *window* occupa l'intera schermata, ma può essere anche più piccola o apparire sopra altre *window*.

1.3.2 Service

Il *Service* [6] è il componente dell'applicazione che si occupa di effettuare operazioni di lunga durata, che non prevede iterazioni con l'utente, infatti non ha alcuna interfaccia grafica. E' un componente molto semplice che ha due principali proprietà:

- Offre all'applicazione un metodo per dire al sistema che devono essere eseguite delle operazioni in background;
- Consente all'applicazione di mettere a disposizione delle operazioni per altre applicazioni.

1.3.3 Broadcast Receiver

Il *Broadcast Receiver* [7] è un componente che ha il compito di ricevere e rispondere alla ricezione di un evento, che di solito si presenta sotto forma di annuncio broadcast. Ci sono diversi tipi di annunci broadcast, quelli generati dal sistema (ad esempio l'annuncio che una foto è stata scattata o che la batteria è scarica), ma anche quelli generati e inviati dalle applicazioni stesse. Esso non è dotato di un'interfaccia grafica anche se però può creare notifiche nella barra di stato.

1.3.4 Content Provider

Il *Content Provider* [8] è il componente predisposto ad immagazzinare e a reperire i dati condivisi tra più applicazioni. I dati salvati nel file system, in un database o sulla rete possono essere consultati da qualsiasi applicazione fornita dei permessi necessari. Android fornisce dei *Content Provider* per i tipi di dati più comuni (audio, video, immagini, etc.). Se un programmatore vuole condividere i dati di una propria applicazione dovrà: o creare un proprio Content Provider o aggiungere i suoi dati a un Content Provider già esistente. Ogni Provider immagazzina i dati in modo diverso, ma tutti implementano un'interfaccia comune, per effettuare la query, per aggiungere, modificare, rimuovere dati e restituire i risultati.

1.3.5 Le Interazioni in Android

I componenti Activity, Service e Broadcast Receiver utilizzano come unica forma di comunicazione lo scambio asincrono di *Intent* [9]. Un *Intent* è un oggetto della classe *android.content.Intent* che consente la comunicazione in run-time tra le componenti della stessa applicazione o di diverse applicazioni. Gli *Intent* posso essere divisi in due classi:

- **Gli Intent espliciti**, in questo caso l'*Intent* consente il passaggio da un Activity ad un'altra indicando semplicemente e in maniera univoca il nome dell'Activity a cui si vuole passare e che si vuole eseguire. Questo tipo di *Intent* sono utilizzati di solito per lo scambio di informazioni interno all'applicazione.
- **Gli Intent impliciti**, che non indicano esplicitamente l'Activity da eseguire; in questo caso l'*Intent* indica solo i dati da elaborare e eventualmente quale operazione eseguire, ma non indica alcun nome. Sarà compito del sistema Android trovare il miglior componente per occuparsi dell'*Intent* e riesce a trovarlo comparando l'*Intent* con gli Intent filter dei vari componenti.

1.3.6 Multi-Threading in Android

Uno dei principali obiettivi di uno sviluppatore di applicazioni Android è quello di realizzare delle applicazioni "reattive" [10]. Con tale aggettivo si intende proprio la capacità dell'applicazione di reagire agli input dell'utente nella maniera più veloce possibile. In questo senso Android è un sistema molto rigido e considera la reattività del software un punto fondamentale. Infatti se Android rileva per un'applicazione un tempo di risposta intorno ai cinque secondi, considera subito quella applicazione come non reattiva e il sistema provvede a visualizzare il messaggio noto come **ANR**, "*Application Not Responding*". Generalmente questa situazione produce una finestra di dialogo che chiede all'utente se vuole aspettare la risposta dell'applicazione o se desidera chiuderla.

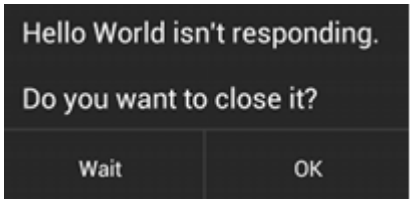


Figura 1.3: "Application Not Responding"

Chiaramente, nessun utente gradisce la comparsa di questa finestra mentre usa l'applicazione, per questo è fondamentale che essa venga progettata, considerando anche l'ottimizzazione dei tempi di risposta.

In realtà la comparsa di questa finestra ha un significato un po' ambiguo, nel senso che per il sistema l'applicazione di riferimento, probabilmente, si è bloccata, e dovrebbe essere terminata, ma in realtà può essere che è in attesa di una risposta. Per poter garantire la reattività dell'applicazione l'idea è stata quella di adottare un sistema Multi-Threading in grado di svolgere più operazioni in parallelo e ottimizzare l'utilizzo delle limitate risorse disponibili su un dispositivo mobile. L'interfaccia utente in Android lavora interamente su un thread principale, detto *Main Thread* o *UI-Thread*, mentre tutti gli altri threads non possono accedere all'interfaccia utente e operano in background. Questi threads "secondari" sono detti *Worker Threads*, essi si occupano di eseguire le operazioni più lente e quelle che sono soggette a tempi di esecuzione lunghi e/o variabili e indipendenti dall'applicazione.

In questi termini Android adotta due regole fondamentali:

1. Non bloccare mai la *Main Thread* con operazioni lunghe. Lo *UI-Thread* deve occuparsi prevalentemente della gestione dei messaggi del sistema riguardanti l'interfaccia utente.
2. Un *Worker Thread* deve solo svolgere lavoro in background e non modificare mai l'interfaccia utente direttamente.

Queste due regole sottolineano l'importanza di avere tanto un thread principale che threads alternativi, ma anche meccanismi opportuni che consentono un'ottima e rapida comunicazione tra di loro.

1.4 Sviluppare applicazioni per Android

Per iniziare a programmare in Android e creare applicazioni è necessario utilizzare alcuni strumenti software:

- Un *JDK* (*Java Development Kit*), che è il kit usato per la normale programmazione Java (è importante però sottolineare che Java non è l'unico linguaggio di programmazione usabile sotto Android, ma di sicuro è quello più comune e più supportato);
- Un *IDE* (*Integrated Development Environment*), un ambiente di sviluppo che include tutti gli strumenti necessari al programmatore (Eclipse o Android Studio).

In realtà, in questo senso gli sviluppatori Android sono particolarmente avvantaggiati. Infatti Android fornisce un **SDK** (*Software Development Kit*) [11], che contiene tutti gli strumenti necessari per lo sviluppo e la realizzazione di applicazioni. Tale "pacchetto" è disponibile per i sistemi operativi più diffusi quali Windows, Linux e Mac; e non necessita di installazione. L'**SDK** comprende molti strumenti tra cui: un emulatore/simulatore, strumenti a supporto dello sviluppo (per la compilazione, il debug e il deploying), documentazione, programmi di supporto la cui composizione può cambiare nel tempo; inoltre essi vengono gestiti attraverso il programma Android SDK Manager, che può essere eseguito sia in Eclipse che in Android Studio. Attraverso il Manager il programmatore può configurare la piattaforma come meglio crede, nel modo più congeniale possibile al proprio lavoro. L'SDK contiene anche un Driver USB che permette di collegare un dispositivo mobile alla macchina su cui è installato l'SDK in modo da poter testare e usare l'applicazione creata. Come accennato questo pacchetto è sempre in continua evoluzione, pertanto si consiglia di utilizzare sempre l'ultima versione disponibile, perché le nuove release contengono nuove funzionalità, che magari possono essere proprio quelle che ci interessano e che vogliamo utilizzare.

1.4.1 Risorse di un'applicazione

Un'applicazione Android non è costituita soltanto da codice, ma essa necessita anche di altre risorse [12], che in alcuni casi sono elementi quali immagini, video, audio usati per definire l'aspetto visivo dell'applicazione, in altri casi si tratta di stringhe, numeri o documenti. Per gestire al meglio tali risorse a disposizione dell'applicazione, l'idea è stata quella di inserirle all'interno di una sottocartella del progetto denominata *res*. Inoltre questa cartella contiene a sua volta delle sotto cartelle in cui le varie risorse sono divise per tipologia.

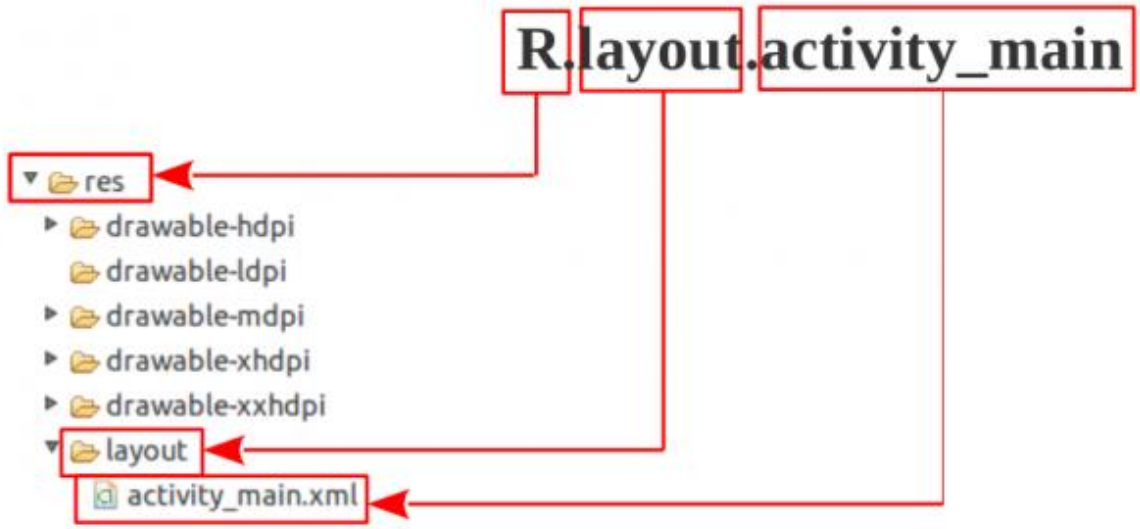


Figura 1.4: R.layout.activity_main

Le più usate sono:

- *values* che contiene colori, stringhe e dimensioni, parametri che possono essere usati per definire altre risorse;
- *drawable* che contiene immagini nei formati più comuni;
- *layout* che contiene elementi dell'architettura grafica dell'interfaccia utente.

Le risorse vengono compilate in un formato binario e sono indicizzate da un numero intero, un ID univoco, con cui è possibile identificarle ed accedervi.

1.4.2 Il Layout di un'App Android

Abbiamo visto che in generale un'applicazione è costituita da più Activity. Un'Activity è caratterizzata da un aspetto grafico. La struttura grafica di un'Activity prende il nome di *layout* [13]. In Android, un layout viene progettato e realizzato o in XML, tramite file, che descrive proprio l'aspetto desiderato, o usando gli IDE che offrono strumenti grafici, come i widget, le windows, per disegnare i layout con un approccio drag-and-drop. Esistono diversi tipi di layout, definiti nel framework di Android e i più comuni sono:

- **LinearLayout** che contiene una serie di elementi che distribuisce in maniera sequenziale, o dall'alto al basso o da sinistra a destra;
- **TableLayout** che contiene un insieme di elementi che inserisce in una tabella, distribuendoli in maniera regolare (per righe e per colonne);
- **RelativeLayout** che è più flessibile e tende a disporre gli elementi in maniera meno strutturata, infatti gli elementi vengono posizionati in relazione al loro contenitore e funzione.

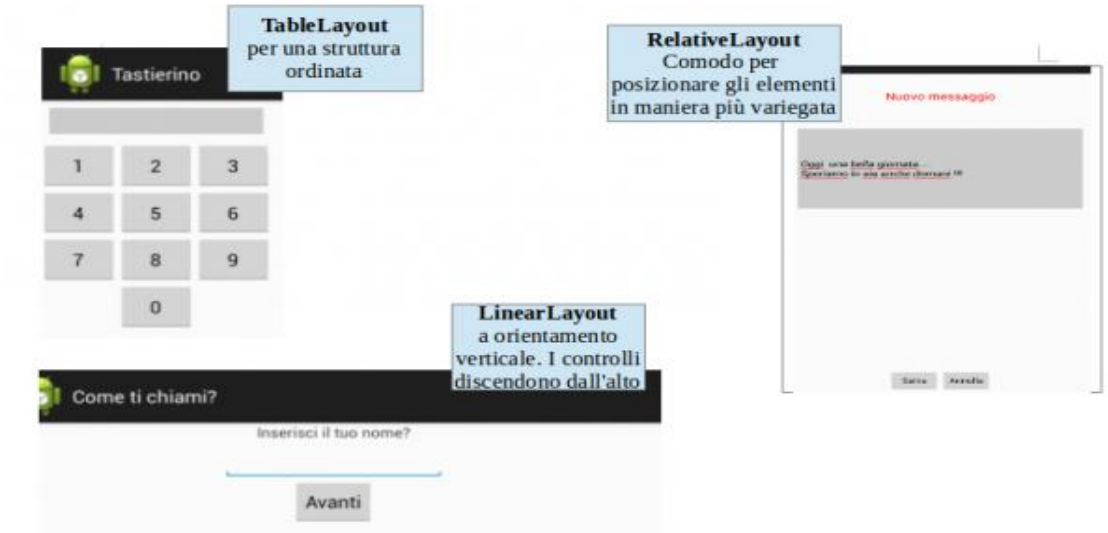


Figura 1.5: Tipi di Layout

1.4.3 L'Interfaccia Grafica Utente

L'*Interfaccia Grafica Utente*, comunemente indicata con la sigla **GUI** [14], acronimo di *Graphical User Interface*, di un'applicazione Android, rappresenta tutto ciò che un utente

può visualizzare e con cui può interagire sul proprio dispositivo mobile. Essa è costituita principalmente da due tipi di oggetti: le *View* e le *ViewGroup*. Le *ViewGroup* sono tipici contenitori di *View*. Le *View* sono tutti gli elementi che compaiono in un'interfaccia e svolgono principalmente due funzionalità:

1. Descrivono un aspetto grafico;
2. Gestiscono eventi, dovuti all'interazione dell'utente.

Tra le varie tipologie di *View* esistenti, sicuramente le più importanti sono i *layout* di cui abbiamo già parlato, e i *widgets*.

1.4.4 I widgets

I *widgets* [15] sono elementi grafici che costituiscono l'interfaccia utente, in particolare sono quegli elementi attraverso cui un utente, può effettivamente interagire con un'applicazione, controllandone o modificandone il flusso di esecuzione. Tra i widgets più comuni in Android abbiamo:

- **TextView**, usato per rappresentare un testo fisso;
- **EditText** usato per permettere l'inserimento di testo;
- **Button** è un pulsante, nel caso più comune questo elemento consente di gestire l'evento click, per attivare una qualche reazione dell'Activity;
- **CheckBox** che definisce un flag che può essere attivato o disattivato;

Ciascun widget è caratterizzato da un ID e da una serie di caratteristiche, definite da un nome accompagnato da un valore, che variano da tipo a tipo.

1.4.5 L'emulatore

Uno degli strumenti più importanti inclusi nell'SDK è l'emulatore virtuale di dispositivi mobili [16], che permette di sviluppare, simulare e testare le applicazioni anche se non siamo in possesso di un dispositivo fisico. Tale emulatore riproduce tutte le caratteristiche software e hardware di un dispositivo mobile, ad eccezione solo di poche, come: le chiamate telefoniche o il touchscreen, quest'ultimo però può essere simulato usando il

mouse. Presenta tutti i tasti tipici di un normale dispositivo mobile, anch'essi gestiti con il mouse, per generare e gestire gli eventi che caratterizzano un'applicazione. Inoltre nel simulatore è presente anche una tastiera QWERTY, con cui è possibile comporre un testo (naturalmente per inserire un testo è possibile usare anche la tastiera della propria macchina). Per consentire di modellare e testare le applicazioni, l'emulatore usa le configurazioni **Android Virtual Device** (**AVD**). Un **AVD** è una macchina virtuale che rappresenta proprio il dispositivo emulato. Essa esegue un sistema Android completo, e include una serie di applicazioni preinstallate a cui è possibile accedere.

Figura 1.6: Android Virtual Device

Quando viene creato un AVD è possibile definire alcuni parametri hardware del dispositivo (dimensione del display, numero di webcam, dimensione della RAM, dimensione della SD, etc.), permettendo quindi di creare più configurazioni e per testare le applicazioni su più dispositivi diversi; è possibile inoltre eseguire tali dispositivi singolarmente oppure contemporaneamente.

1.5 L'AndroidManifest.xml

Per realizzare un'applicazione mobile, un ulteriore elemento importante, che possiamo dire costituisce il cuore dell'applicazione, è il file *AndroidManifest.xml* [17]. Questo file è molto importante perché oltre ad elencare gli elementi che costituiscono l'applicazione, le Activity e i Service (rappresentati come dei nodi), definisce i diritti e i permessi (rappresentati come attributi) per determinare come questi elementi interagiscono tra di loro. Ogni processo ha delle regole ben precise che deve seguire, come: i file di un applicativo non possono essere scritti da altri applicativi, o un processo non può accedere alla memoria di un altro processo; appunto per far in modo che l'applicazione funzioni correttamente. Tra i principali permessi abbiamo:

- **READ_CONTACTS** che consente di leggere i dati dei contatti dell'utente;
- **WRITE_CONTACTS** che consente di scrivere i dati dei contatti dell'utente;
- **RECIVE_SMS** che consente di monitorare l'arrivo di SMS;
- **INTERNET** che consente di utilizzare la connessione Internet;
- **ACCESS_FINE_LOCATION** che consente di utilizzare il GPS;

All'interno di questo file inoltre è definito il nome del package, che identifica in maniera univoca l'applicazione, sono anche definite le classi di tipo Instrumentation necessarie per testare l'applicazione ed è indicata anche la versione minima di Android usata per l'app. Ogni applicazione deve includere il file *AndroidManifest.xml* nella directory principale del progetto.

1.6 L'Android Development Tool

Un ulteriore strumento, fornito insieme all'SDK, molto utile per creare applicazioni Android è l'*Android Development Tool* (**ADT**). Infatti esso contiene tutte le classi e le librerie utili per creare un'applicazione. L'ADT è un plugin che ottimizza Eclipse (ambiente di sviluppo) per Android, cioè fa in modo che supporti la creazione e il debugging delle applicazioni. Usando questo plugin è possibile creare applicazioni in maniera molto semplice, infatti quando si crea un nuovo progetto, verranno subito create

tutte le directory e i file necessari. Inoltre questo plugin aggiunge ad Eclipse l'interfacciamento con il Dalvik Debug Monitor Server (DDMS), che consente di controllare lo stato del dispositivo collegato, sia reale che virtuale.

1.7 Il processo di compilazione

Abbiamo visto che usando Eclipse più il plugin ADT [18] è possibile realizzare un progetto Android: tale progetto attraverso la fase di compilazione viene trasformato in un file Android Package con estensione *apk.*

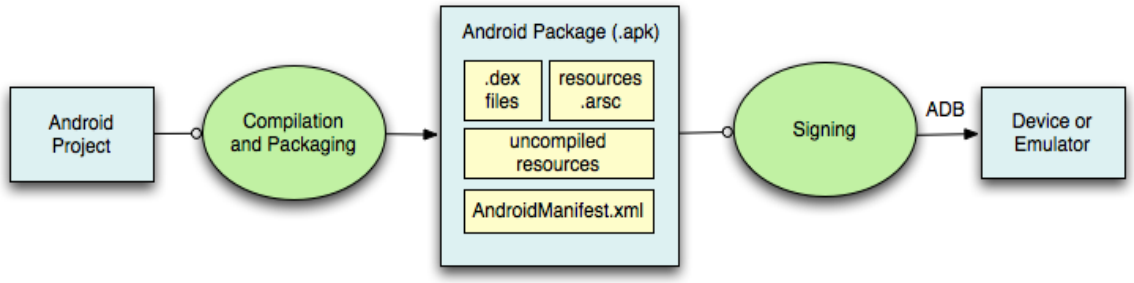


Figura 1.7: L'Android Package

Questo tipo di file è una variante del formato JAR, ed è un archivio che contiene tutte le informazioni per gestire ed eseguire un'applicazione. Esso in Android corrisponde ad un file eseguibile, analogo ad un file con estensione *exe* in Windows. Ed è proprio tramite questo file che è possibile installare l'applicazione sul dispositivo mobile.

In fase di compilazione l'*Android Asset Packaging Tool* (**AAPT**) legge i file XML e genera la classe *R.java*, mentre invece l'*Android Interface Definition Language* (**AIDL**) converte i file con estensione *aidl* in interfacce Java.

Il codice, la classe R.java e le interfacce vengono poi inviate al compilatore Java che genera il file Java con estensione *class.* A questo punto il tool *dex* converte i file in file con estensione *dex* che sono eseguibili dal Dalvik; convertendo quindi il codice da java in byte code. Successivamente il tool *Apkbuilder* riceve le risorse compilate, altre risorse e i file .dex e li comprime in un unico pacchetto, l'Android Package appunto, che si presenta

come un unico file con estensione apk.

Dopodiché tale file viene passato al tool *Jarsigner* che permette di inserire una firma nel pacchetto. La firma è utile a garantire la fiducia dell'applicazione e serve per poter pubblicare l'applicazione. Infine il pacchetto APK viene ottimizzato con il tool *Zipalign*, che provvede a fare in modo che quando l'applicazione verrà installata e eseguita sul dispositivo, non occupi troppa memoria.

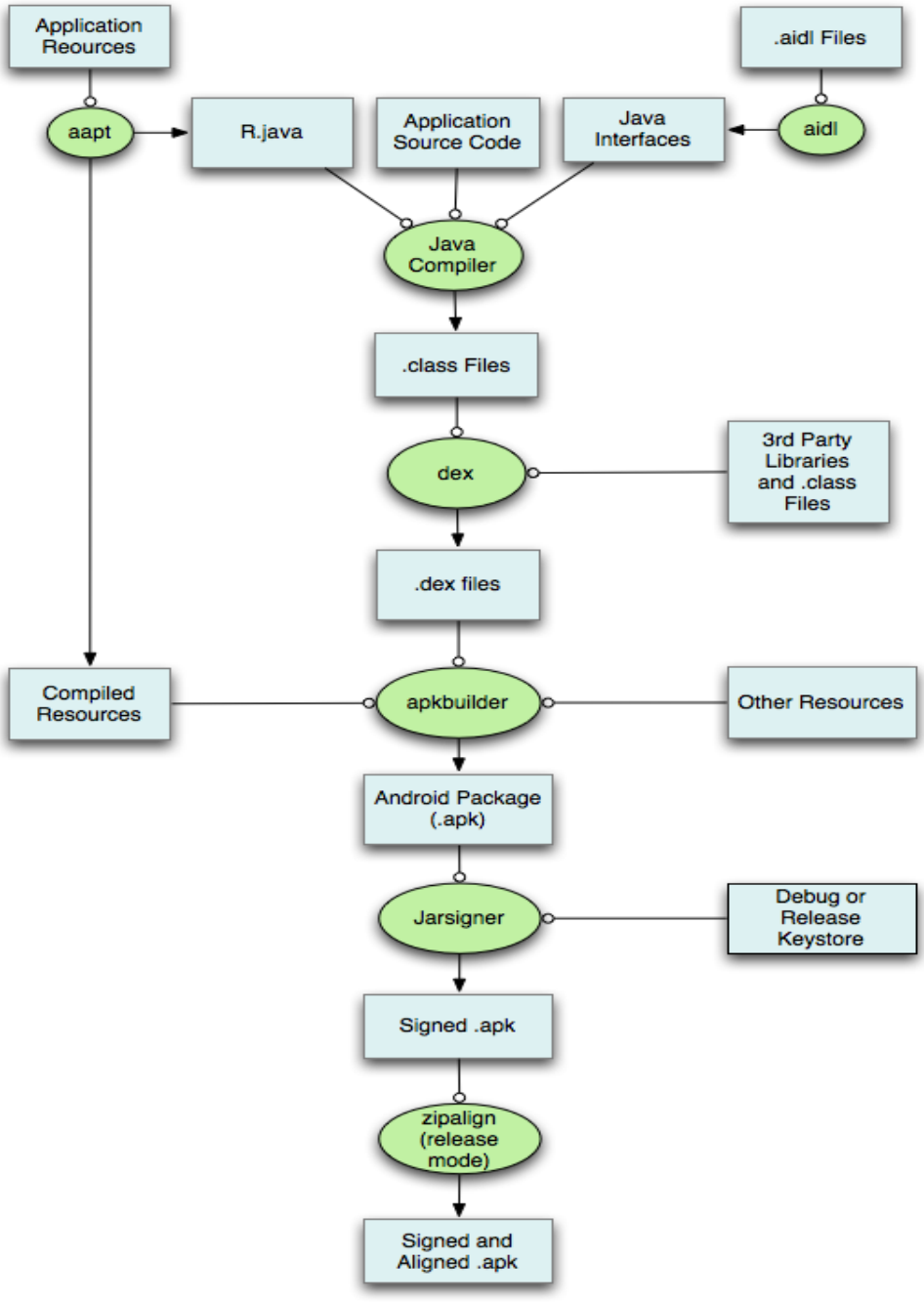


Figura 1.8: Il processo di compilazione

1.8 Google Play Store

La distribuzione delle applicazioni mobili avviene attraverso appositi negozi virtuali online. Google Play detto anche Google Play Store è il negozio ufficiale per dispositivi Android [19].

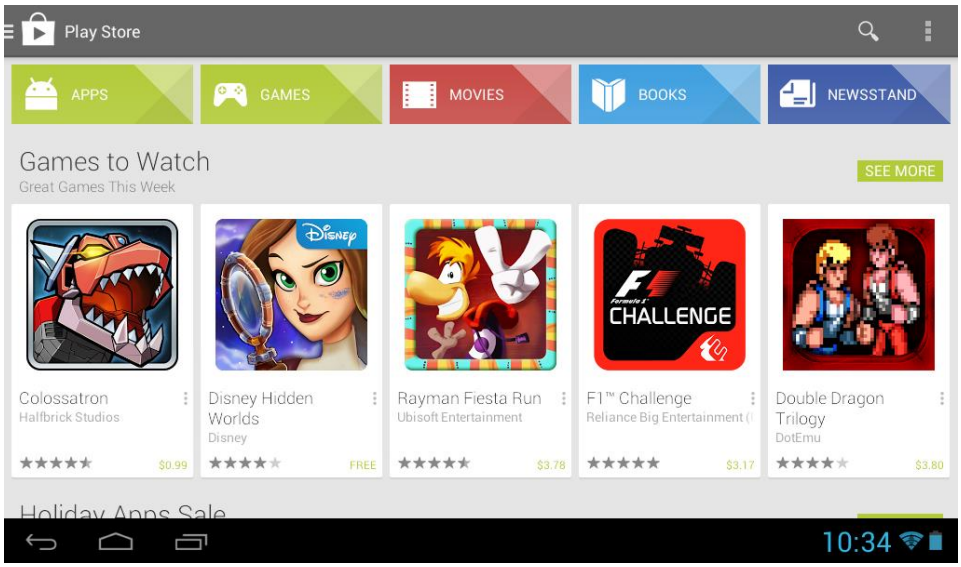


Figura 1.9: Google Play Store

E' possibile accedere a tale negozio sia attraverso il web che tramite l'apposita app di solito preinstallata sul dispositivo e che rimanda proprio al sito di Google Play.

Attraverso tale negozio gli utenti possono navigare tra milioni di applicazioni divise per tipologia: applicazioni, brani musicali, pellicole cinematografiche, libri e riviste; e possono scegliere e scaricare quelle che più preferiscono e che sono adatte ai loro dispositivi. Inoltre è possibile scegliere tra applicazioni gratuite e quelle a pagamento.

Inizialmente tale negozio era conosciuto come Android Market, quando però trattava solo applicazioni.

Capitolo 2
Android Testing

2.1 Test delle Applicazioni Android

Una delle fasi più importanti, durante l'implementazione di un'applicazione Android, dopo che questa è stata realizzata, riguarda il testing dell'applicazione stessa. L'obiettivo è quello di verificare il corretto funzionamento dell'app e individuarne l'affidabilità, l'efficacia, l'efficienza, la comprensibilità, la completezza e la complessità. E' possibile testare l'applicazione sull'emulatore, strumento molto comodo perché fornisce dei feedback riguardo all'andamento del test. Ma è importante sottolineare che questo non è sufficiente, poiché l'Android Virtual Device non implementa tutte le caratteristiche hardware di un dispositivo mobile Android, come la fotocamera, il touchscreen, i sensori di inclinazione e altro ancora. Pertanto, in tutti i casi in cui l'applicazione ha bisogno di questi elementi hardware, è necessario eseguire il debug dell'applicazione anche su un dispositivo fisico reale. Per capire come testare un'applicazione Android, ricordiamo che essa è composta da un lato client e da una o più risorse lato server. Ci occuperemo dello studio delle problematiche relative al lato client, che costituisce proprio l'app. L'app Android è un'applicazione interattiva caratterizzata da transizioni di stato dell'interfaccia, dovute al verificarsi di eventi. Gli eventi possono essere eventi utente, quindi legati all'interazione con i widgets della GUI, oppure eventi di sistema, legati ai segnali o a messaggi inviati dai componenti del dispositivo mobile. Durante la fase di testing, l'idea è

quella di prevedere questi eventi e verificare che l'applicazione si comporti nel modo previsto dopo che un dato evento si è verificato. Tra le due tipologie di eventi quelli che suscitano particolare attenzione sono gli eventi di sistema. Infatti tali eventi si possono verificare in qualsiasi momento durante l'esecuzione di un'applicazione, e in maniera non prevista. Basti pensare all'interruzione dell'applicazione dovuta al sopraggiungere di una chiamata telefonica. Inoltre in questo caso l'utente si aspetta di poter riprendere l'applicazione dal punto in cui l'aveva lasciata, una volta terminata la chiamata. Gestire e valutare tale fenomeno in fase di testing non è semplice. Naturalmente è importante valutare attentamente anche gli eventi utente, che in generale sono prevedibili, ma si possono verificare situazioni in cui tali eventi vengano eseguiti in maniera asincrona rispetto al flusso dell'applicazione generando scenari non previsti in fase di implementazione. Una buona strategia per il test con Android include i seguenti passi:

- Unit Test;
- Test di integrazione;
- Operational Test;
- Test del sistema.

Per Unit Test si intende l'attività di testing eseguita su singole unità del software, per verificarne il comportamento. In un'applicazione Android tali unità possono essere: Activity, Services, Broadcast Receiver, Content Provider, oppure delle specifiche funzioni o semplici classi Java. Per questo tipo di test è necessario definire un driver e uno stub, dove il driver rappresenta un'unità chiamante, e lo stub l'unità chiamata. Il Test di integrazione rappresenta una estensione del Test Unit, infatti in questo caso l'elemento da testare è dato dalla combinazione di due o più unità, quindi si testano combinazioni di elementi che costituiscono il software. Il Test Operativo invece consente di valutare un'applicazione da un punto di vista delle sue funzionalità. Infine il Test del sistema consente di controllare l'interazione tra le varie componenti hardware e software del sistema. La piattaforma Android offre diversi strumenti per eseguire test tra cui: un framework JUnit, l'Instrumentation, MonkeyRunner e Monkey [20]; a cui va aggiunto un ulteriore strumento detto Robotium.

2.1.1 Android testing framework

Android include un testing framework integrato che consente di testare tutti gli aspetti di un'applicazione, tale framework è una estensione di quello JUnit. L'elemento base è il **Test Case JUnit** (o i casi di test JUnit) che altro non è che una classe Java che contiene i metodi di test, opportunamente definiti per coprire ogni possibile situazione. E' possibile scrivere sia test JUnit classici sia test dedicati alla piattaforma Android. Inoltre è buona norma raggruppare Test Case omogenei in una **Test Suite**, questo perché eseguendo la Suite, verranno eseguiti tutti i test in essa contenuta, e inoltre in questo modo il codice del test risulta più ordinato. L'esecuzione di un test può portare a tre risultati:

- *Successo* (pass), vuol dire che il test è andato a buon fine;
- *Fallimento* (failure), vuol dire che il test è stato eseguito, ma l'esecuzione del test non ha prodotto il risultato atteso;
- *Errore* (error), indica una situazione imprevista e che qualcosa è andato storto.

Inoltre l'Android testing framework è esteso da classi che forniscono metodi per poter creare e gestire i componenti e controllarne il ciclo di vita.

In JUnit, per eseguire le classi di test si utilizza un *test runner*; in Android invece è necessario usare degli strumenti, dei tool, che permettono di caricare i test package e l'applicazione sotto test, dopodiché si usa un altro tool per controllare un *test runner* specifico di Android (l'Instrumentation TestRunner).

2.1.2 Instrumentation

Il collegamento con l'applicazione sotto test avviene attraverso l'Instrumentation Framework. L'Android Instrumentation è un insieme di metodi ("hooks") che permettono di "agganciarsi" al sistema Android. Questi agganci permettono di controllare le componenti Android indipendentemente dal loro normale ciclo di vita. Normalmente un Activity esegue il ciclo di vita determinato dal sistema o dal programmatore della app. Per esempio il ciclo di vita viene iniziato da un Intent, poi viene chiamato il metodo onCreate(), che consente di avviare l'applicazione. Quando l'utente lancia un'altra

applicazione, viene chiamato il metodo onPause(), per mettere l'applicazione in uno stato di standby; in fine se l'Activity chiama il metodo finish(), viene chiamato il metodo onDestroy() per chiudere l'applicazione definitivamente. Android non permette di chiamare direttamente queste call-back, ma in fase di test sarebbe molto utile poter usare tali metodi per poter controllare il ciclo di vita di un'Activity, forzando ad esempio un suo componente in uno stato differente da quello previsto dal suo normale ciclo di vita, cioè nello stato che ci interessa. Questo però è possibile attraverso l'Instrumentation, infatti possiamo simulare il ciclo di vita delle Activity e forzare la chiamata alle call-back per testarne il funzionamento. Per esempio una funzione molto utile, che fa parte proprio dell'Instrumentation API è getActivity(), che consente di ottenere informazioni relativamente ad un'Activity in esecuzione e eseguire uno dei qualsiasi metodi ad essa associati.

2.1.3 MonkeyRunner e Monkey

L'SDK di Android offre due strumenti utili per il testing: *MonkeyRunner* e l'Application Exerciser Monkey, più semplicemente conosciuto come *Monkey*. *MonkeyRunner* è un tool che consente di controllare un emulatore o un dispositivo Android. Con questo strumento è possibile scrivere un programma, in linguaggio Python, che consente di installare un'applicazione sul dispositivo, inviare comandi alla GUI e consente di memorizzare gli screenshot dell'interfaccia, che rappresentano lo stato dell'applicazione. In fase di testing *MonkeyRunner* consente:

- Il controllo multiplo del dispositivo;
- Di eseguire test automatici (test funzionale);
- Di eseguire test di regressione, per testare la stabilità di un'applicazione.

E' importante non confondere *MonkeyRunner* con *Monkey*, quest'ultimo è un programma eseguito sul dispositivo, fisico o virtuale, e genera flussi pseudo casuali di eventi ed input, testando quindi l'applicazione in maniera casuale e stressante. Entrambi questi strumenti consentono di automatizzare il testing delle applicazioni Android.

2.1.4 Robotium

Robotium [21] è un test framework open source, che estende il test framework di Android, usato per scrivere robusti e potenti Test Case per applicazioni Android, che ricoprono tutti o quasi gli scenari possibili; facilitando quindi il lavoro del programmatore/sviluppatore.
E' molto efficace soprattutto per eseguire testing Black Box ed è in continua evoluzione. Tuttavia *Robotium* presenta un limite nel GUI testing. In particolare esso riconosce solo gli elementi che sono effettivamente visibili nell'interfaccia, ignorando tutti gli altri (non visibili). Pertanto se un test viene eseguito su dispositivi diversi, che hanno display con risoluzioni diverse, può capitare che tale test dia esiti diversi, poiché possono essere considerati elementi diversi. Naturalmente questo è un problema. Per fare in modo che vengano considerati tutti gli elementi, anche quelli non visibili, magari appartenenti ad una lista, è necessario eseguire la funzione *solo.scrollDownList(0)* prima dell'esecuzione del test; tale funzione consente di scorrere tutti gli elementi della lista, in modo che nessuno venga ignorato.

2.2 GUI Testing

Oggi giorno quasi tutti i software e i sistemi operativi sono dotati di una GUI. L'introduzione della GUI ha rappresentato una vera svolta nello sviluppo di software, infatti in passato qualsiasi interazione avveniva tramite linea di comando. L'importanza dell'introduzione della GUI sta soprattutto nel fatto che attraverso l'interfaccia grafica qualsiasi utente può interagire con un dispositivo in maniera molto semplice ed intuitiva. Pertanto l'interfaccia grafica è diventata uno degli elementi più importanti da considerare durante la creazione e lo sviluppo di un software. L'obiettivo è quello di creare un'interfaccia intuitiva, ma che permette all'utente di usare tutte le funzionalità di un dispositivo e di accedere alle sue componenti. Una volta sviluppata, la GUI dovrà essere testata; inoltre bisogna testarla anche ogni qual volta verrà aggiornata, o perché è stato scoperto un bug o perché sono state aggiunte nuove funzionalità, richiedendo quindi la frequente esecuzione di test di regressione. E' importante inoltre testare la GUI non come

componente a se, ma proprio come parte integrante del software, perché altrimenti potremmo non ottenere tutte le informazioni sul suo reale funzionamento nell'ambiente finale. Nell'ingegneria del software il GUI Testing [22] è un processo per testare l'interfaccia utente di un sistema o di un'applicazione per rilevare che questa funzioni correttamente. Lo sviluppo del test consiste nel realizzare e eseguire tutta una serie di casi di test con cui si cerca di ricoprire tutte le funzionalità del software. La difficoltà nel realizzare questo test sta soprattutto nel trattare con sequenze di eventi e stati dell'applicazione. L'idea è quella di effettuare tutta una serie di tasks e di controllare e confrontare i loro risultati con quelli che ci si aspetta, eventualmente poi ripetere questi casi di test con input diversi.

Per migliorare il testing sono state introdotte delle tecniche di automazione, che consentono di ottenere dei test più:

- **affidabili**;
- **efficaci**;
- **con cicli ridotti**.

Con le tecniche di generazione dei casi di test automatici si cerca innanzitutto di creare un modello astratto (come il Finite State Machine Model o l'Event Flow Graph) dell'applicazione sotto test, in particolare per individuare eventi, stati, transizioni e interazioni, poi tale modello viene usato per generare i casi di test. Possibili approcci al GUI Testing possono essere:

- **Model-Based:** in cui esiste un modello non molto sofisticato, che descrive l'applicazione, in particolare le interfacce, a partire dal quale si ottengono i casi di test;
- **Random Testing:** in cui non esiste un modello descrittivo dell'applicazione, pertanto tale applicazione viene testata generando input ed eventi casuali sulle interfacce;
- **Ripper Based:** in cui l'applicazione viene esplorata in maniera metodica cercando di individuare situazioni non gestite; inoltre questa tecnica consente di costruire un modello che è possibile usare per elaborazioni future.

2.2.1 Modello Macchina a Stati Finiti (FSM)

Uno dei modelli astratti più usati, per descrivere un'applicazione, è il modello della Macchina a Stati Finiti (**Finite State Machine Model**) [23].

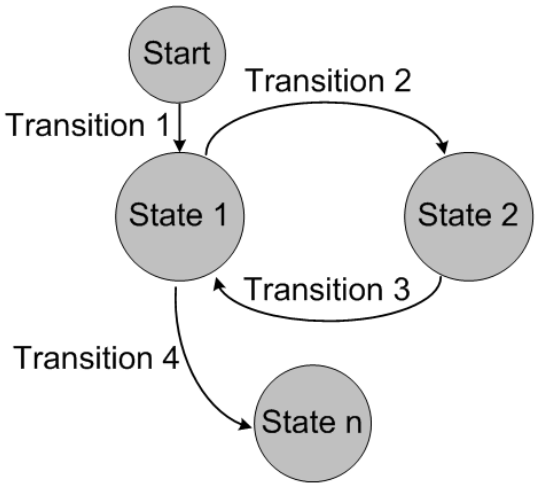


Figura 2.1: Modello Macchina a Stati Finiti

E' rappresentato come un diagramma che contiene nodi e archi, dove ogni nodo rappresenta un particolare stato dell'interfaccia, mentre gli archi rappresentano le transizioni. Ciascuna transizione è scatenata da un evento che di solito è dovuto all'interazione dell'utente con la GUI. Questo modello descrive quindi il comportamento dell'applicazione in termini di stati.

2.2.2 Event Flow Graph (EFG)

Un altro modello astratto usato per descrivere un'applicazione è l'**Event Flow Graph** (EFG) [24]. Questo modello descrive il comportamento dell'applicazione in termini di eventi, infatti consente di individuare e rappresentare tutti gli eventi ai quali è possibile accedere a partire dall'interfaccia utente. Ogni nodo del diagramma rappresenta un evento, invece ogni arco rappresenta una traslazione, che non è intesa più come interazione con l'interfaccia scatenata da un evento, ma è intesa come relazione tra eventi; cioè indica se un evento può scatenare un altro evento.

Ogni cammino rappresenta una sequenza di eventi e corrisponde quindi a un possibile caso di test.

2.2.3 GUI Ripping Tool e GUI-Tree

Il *GUI Ripping Tool* [25] è uno strumento che implementa un processo dinamico, simile al Web Crawler, che consente di esplorare automaticamente la GUI di un'applicazione.

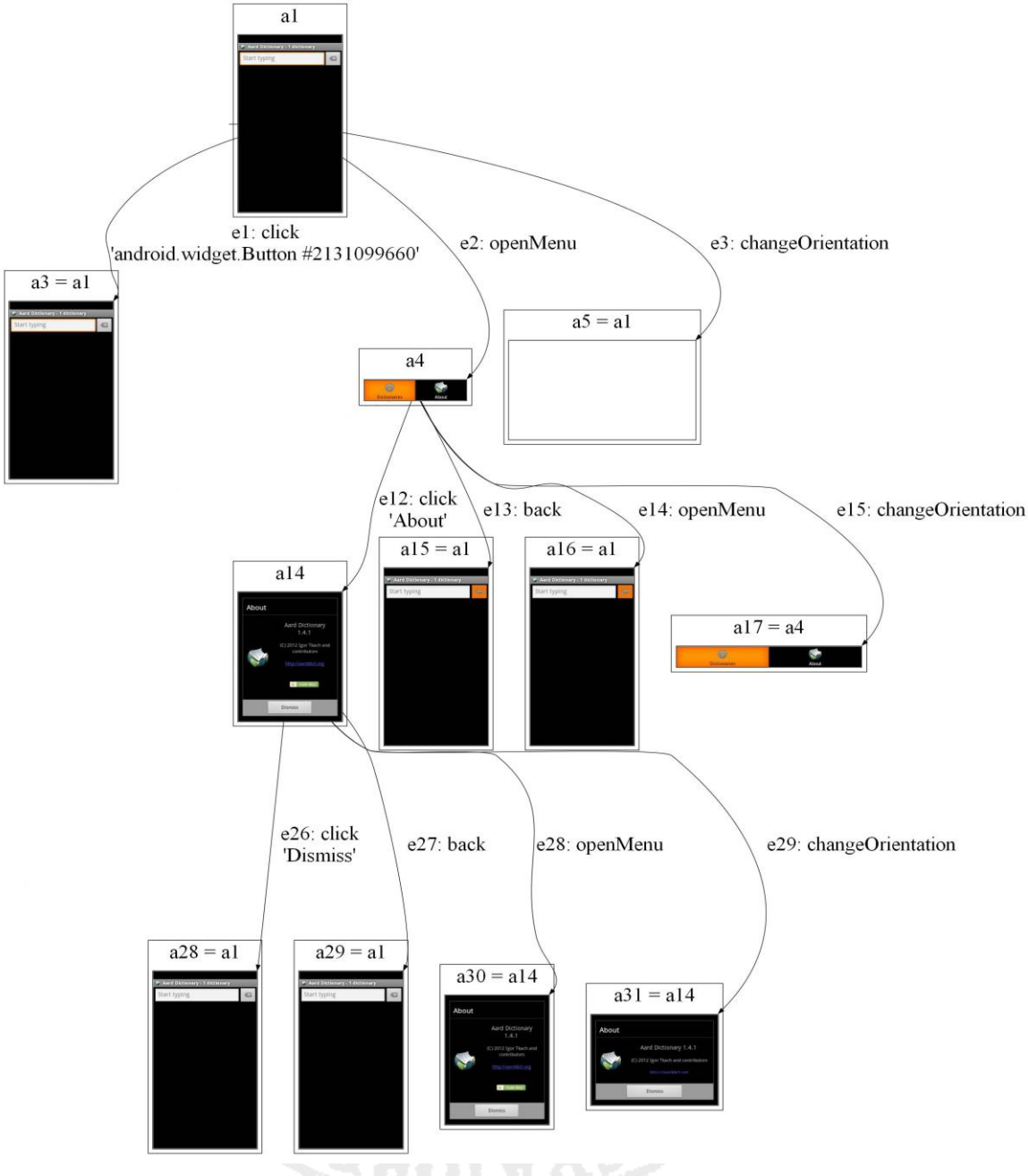


Figura 2.2: Esempio di *GUI-Tree* dell'app Aard Dictionary (rappresentazione parziale)

Con questo tool è possibile ottenere informazioni riguardo allo stato del software sotto test, ricordando che il comportamento di tale software dipende dagli eventi scatenati dall'utente e che vanno a modificare il suo stato e quindi l'interfaccia. In particolare è possibile individuare e analizzare le interfacce che compongo l'applicazione, analizzando i widgets che le compongono e gli eventi che portano a tali interfacce. Inoltre è possibile ottenere, sempre attraverso il tool, il modello astratto che descrive l'applicazione detto *GUI-Tree.* Esso rappresenta un albero della struttura di un'applicazione in termini di GUI. Ogni "nodo" rappresenta un'interfaccia dell'applicazione, uno stato; mentre invece ogni "arco" rappresenta una transizione, scatenata da un evento utente, che implica il passaggio da uno stato ad un altro. In linea generale il *GUI-Tree* che questo tool è in grado di definire è illimitato perché l'algoritmo di GUI Ripping adottato consente di valutare ogni possibile stato dell'app e tutti i possibili eventi scatenabili che possono portare a quello stato, più quelli che possono essere scatenati a partire da tale stato.

Questo approccio consente di realizzare sicuramente un test molto **efficace**, in termini di copertura del codice sorgente, poiché con esso si vanno a considerare tutte le "situazioni" possibili dell'applicazione, tutti i possibili stati e tutti i possibili percorsi; ma meno **efficiente** perché normalmente si vanno a valutare anche stati già percorsi e meno **comprensibile**, infatti ogni percorso tende ad essere molto lungo. Per risolvere questo problema è necessario definire e utilizzare:

- *un criterio di terminazione*, con cui il tester può definire, ad esempio, che lo scatenarsi di eventi termina quando si verifica una certa condizione, imponendo quindi un limite di profondità;
- *un criterio di esplorazione,* basato sull'uguaglianza imponendo appunto che anche se uno stato potrebbe essere esplorato, non lo sarà, poiché uguale ad uno stato già controllato;

Per tenere conto delle varie interfacce ne viene creata un'istanza, cioè si considerano gli elementi in essa contenuti, ricordando che due interfacce sono considerate equivalenti se hanno lo stesso nome e lo stesso numero di widgets.

Concludiamo ricordando che il tool offre anche altre due funzionalità:

- la prima è la funzione di ripristino, tale funzione consente di salvare i risultati del test e di riprendere l'esecuzione del test se si verifica un errore o malfunzionamento dell'AVD;
- la seconda riguarda il salvataggio di tutti gli screenshot dell'emulatore; queste immagini sono utili per capire cosa fa il Ripper, quali elementi va ad esplorare.

2.2.4 L'Algoritmo di GUI Ripping

L'algoritmo di GUI Ripping è un algoritmo di esplorazione automatica delle GUI di un'applicazione, guidato dagli eventi che sono scatenati sulle interfacce di un'applicazione Android. E' un algoritmo euristico, che tende ad individuare tutti i possibili stati/interfacce che caratterizzano l'applicazione sotto test e gli eventi associati.
L'algoritmo richiede l'esistenza di una task list in cui vengono salvati gli eventi scatenabili e che sono stati individuati durante la fase di esplorazione. Inoltre per evitare ricerche illimitate, viene fatto un confronto tra stati, confrontando le interfacce. Per fare questo, per ogni GUI, viene memorizzata un'opportuna "immagine". Le operazioni fondamentali dell'algoritmo possono essere descritte nel seguente modo:

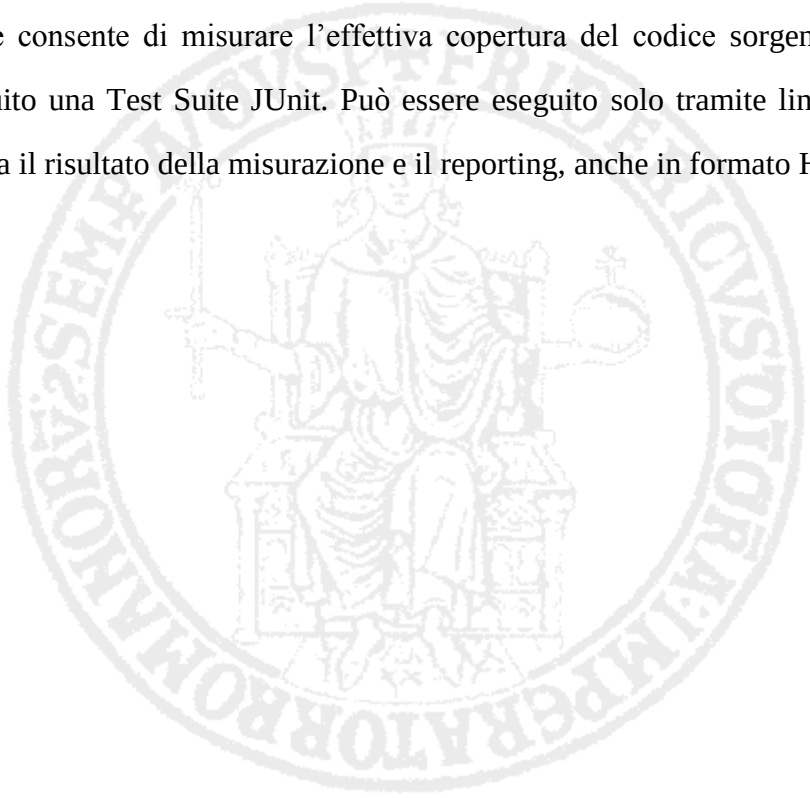
1. Il Ripper va ad analizzare l'interfaccia grafica dell'applicazione e cerca di individuare tutti gli eventi che è possibile esercitare a partire da questa interfaccia e se li trova, li inserisce nella task list;
2. Poi controlla la task list, se questa è vuota il Ripper esce, se invece la lista non è vuota, viene estratto il primo elemento della lista e viene eseguito;
3. Il nuovo evento, in genere, dovrebbe portare a un nuovo stato della GUI. A questo punto la presunta nuova GUI viene confrontata con gli stati già controllati e memorizzati precedentemente, e se essa è effettivamente nuova allora si ripetono le operazioni di esplorazione degli eventi ritornando al punto 1; se invece essa non è nuova si ritorna al punto 2.

2.2.5 Configurazione del GUI Ripping Tool

Per poter usare correttamente il tool, il tester deve definire alcuni parametri. Innanzitutto deve definire i parametri che consentono di individuare l'applicazione da testare, come il **nome della Main Activity**, o il **nome del Package**; è possibile verificare tali parametri consultando il file *AndroidManifest.xml*. Inoltre il tester può definire quali eventi dell'applicazione possono o devono essere scatenati; e può anche gestire gli eventi esterni, quelli di sistema, come il tasto "*BACK"* o "*MENU"* e la rotazione del dispositivo.
Come accennato precedentemente il tester può anche definire la strategia da adottare per il test, definendo il criterio di terminazione, impostando quindi un limite di profondità, e può anche configurare un criterio di esplorazione/equivalenza, evitando di ricontrollare interfacce simili. Infine il tester ha anche la possibilità di definire un tempo di esecuzione massimo del test, espresso in millisecondi.

2.2.6 EMMA Toolkit

Un ulteriore strumento che viene usato in fase di test è il tool ***EMMA*** [26]. E' un tool open source che consente di misurare l'effettiva copertura del codice sorgente ottenuta dopo aver eseguito una Test Suite JUnit. Può essere eseguito solo tramite linea di comando e rappresenta il risultato della misurazione e il reporting, anche in formato HTML.



Capitolo 3

Tecniche per ottimizzare il testing

3.1 Test Automation

Per ottimizzare il testing di un software, di un'applicazione, esistono diverse tecniche, ciascuna delle quali di solito, tende a migliorare una specifica caratteristica: l'efficacia, l'efficienza o la durata del test. Una delle tecniche usata è quella dell'automazione del test; è importante applicare tale tecnica ad un meccanismo efficiente, altrimenti non si avrà alcun beneficio, come ricorda Bill Gates: "*La prima regola di ogni tecnologia è che l'automazione applicata ad un'operazione efficiente ne aumenterà l'efficienza. La seconda è che l'automazione applicata ad un'operazione inefficiente ne aumenterà l'inefficienza*".
Con il termine Test Automation facciamo proprio riferimento a tutte le tecniche che ci permettono di automatizzare alcune attività del processo di testing. In generale gli elementi su cui si opera sono:

- Generazione dei casi di test;
- Preparazione ed esecuzione del test;
- Valutazione.

L'automazione del test è importante soprattutto quando si ha a che fare con una Test Suite che contiene molti Test Cases, necessari per ottenere un test efficace. Infatti in questo caso eseguire manualmente tutti i casi di test può essere una operazione molto onerosa. I vantaggi che di solito si ottengono con questa strategia, sono legati all'efficienza del test,

in particolare abbiamo:

- la riduzione dei costi del testing;
- la riduzione del tempo speso per il test;
- una maggiore affidabilità del test (non c'è il rischio di errore umano);
- aumento (eventualmente) del coverage del test.

Nei nostri test per applicazioni Android abbiamo adottato questa tecnica per usufruire dei suoi vantaggi e abbiamo ottenuto questo risultato utilizzando appositi strumenti quali: il GUI Ripping Tool, l'Android testing framework, l'ADB e Robotium.

3.2 Tecniche di Riduzione del Test di Applicazioni Android

La tecnica di automazione del test è quella che di solito viene usata; ma per migliorare ulteriormente un test possono essere adottati altri approcci, come le tecniche di riduzione, che consentono di migliorare l'efficienza dei test, mantenendo più o meno l'efficacia inalterata. Abbiamo focalizzato la nostra attenzione proprio su questo tipo di tecniche, infatti l'obiettivo della mia tesi è proprio quello di realizzare delle nuove metodologie per definire delle Test Suite "ridotte" a supporto del processo di testing per applicazioni Android e di confrontare i risultati ottenuti con quelli ottenuti nel caso in cui è stata usata una Test Suite completa; per valutare la bontà di questo metodo.

Partendo dai risultati ottenuti dall'esecuzione di test su applicazioni Android fatte in precedenza, usando la tecnica del GUI-Ripper, e dai due file, *guitree.xml* ed *activities.xml*, che descrivono l'applicazione e i test fatti su di essa; lo scopo del mio lavoro è stato quello di cercare un metodo per ottimizzare le Test Suite e i risultati ottenuti nei test precedenti e sviluppare, quindi, un tool che implementasse tale metodo. In particolare l'obiettivo è stato quello di ottenere dei test più **efficienti** a parità o meno di **efficacia** e comprensibilità. Ove per quantificare:

- L'**efficacia** abbiamo indicato la percentuale di linee di codice sorgente coperto, il Max Coverage;

- L'**efficienza** abbiamo considerato il valore dato dal rapporto tra l'efficacia ottenuta per eseguire il test e il numero di eventi totali per raggiungere il valore di massima copertura, infatti essa è intesa come lo sforzo che si fa per eseguire il test e per raggiungere quel valore di copertura.

L'idea è stata quella di adottare una tecnica di riduzione delle Test Suite che permettesse di ottenere Test Suite più piccole, con meno Test Cases, eliminando quelli che possiamo considerare "inutili" o "superflui" rispetto ad una strategia ben definita

3.2.1 Tecnica di Riduzione per Stati e Tecnica di Riduzione per Eventi

Per valutare l'utilità della riduzione, nel nostro caso abbiamo adottato due strategie. Nel primo caso abbiamo ridotto la Test Suite focalizzando l'attenzione sulle Activity, definendo "*il sottoinsieme minimo di trace che copre il massimo quantitativo di Activity*". Nel secondo caso, invece, abbiamo ridotto la Test Suite, focalizzando l'attenzione sugli Events, estraendo "*il sottoinsieme minimo di trace che copre il massimo quantitativo di Events*". In questo modo abbiamo ottenuto due tecniche di riduzione:

- **Una Riduzione per Stati**;
- **Una Riduzione per Eventi**;

E' importante sottolineare che è possibile realizzare diversi meccanismi di riduzione, adottando strategie diverse.

3.2.2 Costruzione delle matrici di copertura

In entrambi i casi il punto di partenza è stato quello di analizzare i file *guitree.xml* ed *activities.xml*, che naturalmente si presentano come degli "alberi" con diversi nodi e archi. In particolare nel file *activities.xml* sono descritti i vari componenti che costituiscono ciascuna interfaccia grafica dell'applicazione in un dato momento; nel file *guitree.xml* sono invece contenuti gli elementi, quali eventi, Activity, transitions, che definiscono i vari Test Cases, che andremo a considerare e che sono eseguiti nel caso di test "completo", per testare l'applicazione Android. In pratica il *guitree.xml* rappresenta l'esecuzione del testing

ed è il file usato per generare la Test Suite Junit. Attraverso questa analisi abbiamo ricavato informazioni relative agli stati e agli eventi ricoperti da ciascuna trace. Inoltre per semplificare il nostro lavoro e le operazioni sulle trace, che costituiscono la Test Suite, abbiamo inserito i dati e i parametri che ci interessano delle trace in due diverse matrici:

- una matrice del tipo trace\Activity;
- una matrice del tipo trace\Events.

Nella prima matrice abbiamo che il generico elemento m(i, j) è pari a 1 se il Test Case i-esimo copre la j-esima Activity, altrimenti tale elemento è uguale a 0. In maniera del tutto analoga abbiamo ragionato per la seconda matrice, infatti abbiamo che il generico elemento m(i, j) è pari a 1 se il Test Case i-esimo copre il j-esimo evento. In questo modo abbiamo ottenuto, per esempio, delle matrici di copertura simili a quelle illustrate nelle seguenti immagini.

	a1	a2	a3	...	aN
T1	0	1	0	...	0
T2	0	0	0	...	1
T3	0	0	1	...	0
T4	1	0	0	...	0
...	...	...	...	...	...
TM	0	1	1	...	0

Figura 3.1: Esempio matrice Trace\Activity

	e0	e1	e2	...	eN
T1	1	0	0	...	0
T2	0	1	0	...	0
T3	0	1	1	...	0
T4	0	0	1	...	0
...	...	...	...	...	...
TM	0	0	1	...	1

Figura 3.2: Esempio matrice Trace\Events

Costruite le due matrici di copertura, le abbiamo poi usate separatamente per trovare il sottoinsieme con il minimo numero di trace che copre il massimo numero di Activity e il sottoinsieme con il minimo numero di trace che copre il massimo numero di Events. Per realizzare le due riduzioni e ricavare i due sottoinsiemi, abbiamo usato l'algoritmo di riduzione di una matrice, basato sul "*metodo delle righe dominate e delle colonne dominanti*".

3.2.3 Algoritmo di riduzione

L'algoritmo di riduzione [27] consente di individuare la forma minima di una funzione: è anche noto come "*metodo di copertura minima*" di una funzione e si presta con estrema facilità alla matrice di copertura che abbiamo definito.

Il primo passo per la ricerca della copertura minima di una matrice di copertura è quello di individuare il suo **nucleo** e quindi i suoi **primi implicanti essenziali**. Questo si ottiene facilmente dalla matrice di copertura: Se infatti in una colonna j della matrice esiste un solo 1, per una certa riga i, cioè risulta che $m_{ij} = 1$ e $m_{kj} = 0 \; \forall \kappa \neq i$ allora per definizione la riga i-esima è un **implicante essenziale** e pertanto va selezionata per la forma minima, si individua così il nucleo della funzione o della matrice. Cioè dopo questo primo passaggio il **nucleo** conterrà solo **primi implicanti essenziali**. Per far riferimento ad un esempio specifico partiamo dalle matrici illustrate precedentemente.

	a1	a2	a3	...	aN
T1	0	1	0	...	0
T2	0	0	0	...	1
T3	0	0	1	...	0
T4	1	0	0	...	0
...	...	...	...	...	...
TM	0	1	1	...	0

Nucleo	T2, T4

Figura 3.3: Primi implicanti essenziali Matrice Trace\Activity

	e0	e1	e2	...	eN
T1	1	0	0	...	0
T2	0	1	0	...	0
T3	0	1	1	...	0
T4	0	0	1	...	0
...	...	...	...	...	...
TM	0	0	1	...	1

Nucleo	T1, TM

Figura 3.4: Primi implicanti essenziali Matrice Trace\Events

In questo caso possiamo facilmente individuare i primi implicanti essenziali, evidenziati in arancione, mentre in rosso sono indicati i **mintermini**. Nella prima matrice i primi implicanti essenziali corrispondono alle trace T2 e T4; tali trace saranno "prese" per determinare il nucleo della matrice. Nella seconda matrice invece i primi implicanti

essenziali corrispondono alle trace T1 e TM; anche in questo caso tali trace vengono "prese" per definire il nucleo della matrice.

Il secondo passo del processo di minimizzazione consiste nel determinare la forma minima della matrice, si cerca cioè di individuare gli implicanti che, pur non essendo essenziali, devono essere comunque aggiunti al nucleo per trovare una forma che copra tutti i **mintermini**. Una volta trovato il nucleo, come descritto al primo passo, si possono eliminare dalla matrice le righe corrispondenti agli implicanti essenziali e tutte le colonne da queste già ricoperte (nelle due figure precedenti tali righe e colonne sono rappresentate in grigio); riducendo il problema di copertura ad uno più semplice, su una matrice più piccola. Per individuare gli **implicanti non essenziali o secondari**, si utilizza poi, *l'algoritmo delle righe e delle colonne dominanti*. A tal proposito risulta utile la seguente definizione: "Si dice che la riga (o colonna) *i* domina la riga (o colonna) *k* se risulta che $m_{ij} = 1$ ($m_{ji} = 1$) per tutti i valori di j per cui si ha $m_{kj} = 1$ ($m_{jk} = 1$)". In altri termini la riga (colonna) dominante contiene tutti gli 1 della riga (o colonna) dominata ed eventualmente qualcuno in più. In realtà però noi abbiamo ragionato in modo leggermente diverso considerando *l'algoritmo delle righe dominate e delle colonne dominanti;* eliminando dalla matrice le righe dominate e le colonne dominanti:

	a2	a3	...
T1	1	0	...
T3	0	1	...
...	...	...	...
TM	1	1	...

Nucleo	T2, T4

Figura 3.5: riduzione matrice Trace\Activity

	e1	e2	...
T2	1	0	...
T3	1	1	...
T4	0	1	...
...	...	...	...

Nucleo	T1, TM

Figura 3.6: riduzione matrice Trace\Events

A titolo esemplificativo continuiamo a ragionare sulle "nostre" matrici rappresentate nelle figure precedenti (Figura 3.5 e Figura 3.6). Le due matrici sono ottenute per riduzione dalle due matrici di partenza. In questo caso abbiamo che nella prima matrice è possibile individuare che la riga TM è dominante rispetto alle righe T1 e T3; pertanto tali righe possono essere eliminate. Allo stesso modo nella seconda matrice abbiamo che la riga T3 domina T2 e T4, quindi queste ultime possono essere eliminate.

Questo algoritmo si basa sul seguente Teorema: "*Se si eliminano le righe dominate e le colonne dominanti, da una matrice di copertura, se ne ricava una equivalente, che rappresenta cioè il medesimo problema di copertura*". Quindi una volta che dalla matrice sono state eliminate righe e colonne per dominanza, può accadere che una riga (o più di una) assuma la proprietà di coprire da sola una colonna, tale riga è un implicante ed è detta "**implicante essenziale secondario**".

Tale implicante sarà scelto come nuovo e ulteriore elemento del nucleo. A questo punto sarà quindi possibile ripetere nuovamente le operazioni descritte in precedenza per ridurre ancora la matrice, se possibile.

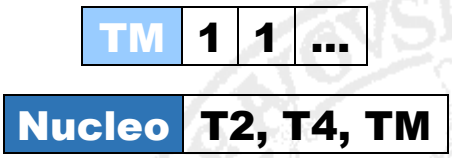


Figura 3.7: ultimo passo della riduzione della matrice Trace\Activity

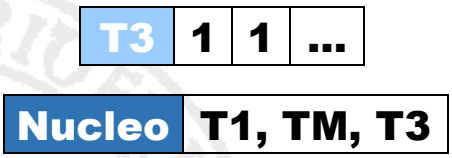


Figura 3.8: ultimo passo della riduzione della matrice Trace\Events

Facendo ancora riferimento al nostro esempio, applicando l'algoritmo delle righe dominate e delle colonne dominanti alla nostre matrici otteniamo che della prima matrice rimane solo la riga TM, della seconda rimane solo la riga T3; entrambe sono primi implicanti secondari, e pertanto saranno considerate all'interno del nucleo della matrice a cui appartengono.

Seguendo questo metodo si ricava il seguente algoritmo, caratterizzato dalle seguenti fasi:

1) Ricerca dei **primi implicanti** (**PI**) ed individuazione di quelli **essenziali**;
2) Inclusione della forma minima dei **PI essenziali**; loro eliminazione dalla matrice unitamente ai **mintermini** ricoperti;
3) Eliminazione delle righe dominate e delle colonne dominanti;
4) Individuazione dei **PI essenziali secondari** nella matrice ridotta;
5) Ripetizione dei passi 2, 3 e 4 finché è possibile;

L'algoritmo termina o perché si eliminano tutte le righe e le colonne della matrice o perché quest'ultima non possiede più né primi implicanti, né righe dominate e né colonne dominanti. Possiamo quindi parlare di una procedura "ciclica", al termine della quale siamo riusciti ad individuare per ciascuna matrice, il sottoinsieme che ci interessa. Questi due sottoinsiemi li abbiamo poi usati per costruire due file guitree.xml ridotti. Ho chiamato questi due file rispettivamente:

- **guitreeReductionState.xml** che è quello ottenuto dalla riduzione per stati;
- **guitreeReductionEvent.xml** che è quello ottenuto dalla riduzione per eventi;

A partire dai quali, abbiamo costruito le due Test Suite ridotte per testare l'applicazione.

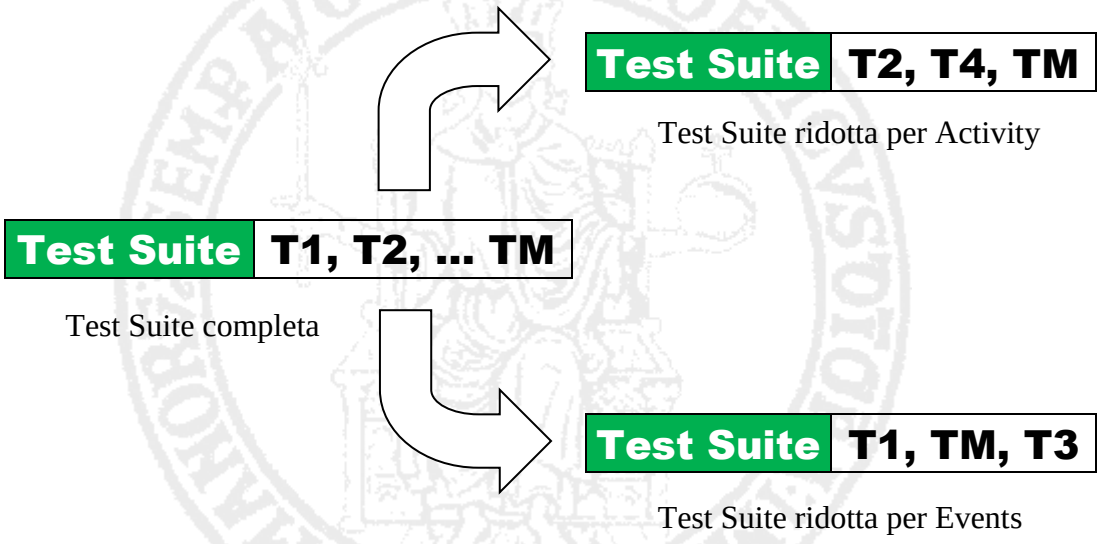


Figura 3.9: Esempio di riduzione di una Test Suite

Capitolo 4

Il Reduction_Tool

Il tool che ho realizzato è un tool che sostanzialmente riceve in ingresso, quindi va a leggere due file: il file *guitree.xml* e il file *activities.xml* di un'applicazione Android in modo da ottenere informazioni relativamente al testing automatizzato (trace che compongono il test, gli eventi che vengono eseguiti in quelle trace e gli stati che vengono incontrati una volta che questi eventi vengono scatenati, etc.) eseguito per quella applicazione, per individuarne eventuali crash o failure; esegue la riduzione e restituisce in uscita due file guitree "ridotti" con estensione XML: il **guitreeReductionStates.xml** che contiene *il sottoinsieme minimo di trace che copre il massimo numero di stati* e il **guitreeReductionEvents.xml** che contiene *il sottoinsieme minimo di trace che copre il massimo numero di eventi*, che sono i due file che ci interessano principalmente. Inoltre il tool restituisce in uscita anche due file di testo organizzati per righe e tabulazioni, **copyprintwriterCAM.txt** e **copyprintwriterCEM.txt**, che contengono una rappresentazione semplice delle matrici Trace\Stati e Trace\Eventi. Ho strutturato il tool in sette diverse classi: Essential_Prime_Implicants.java, Event.java, EventManagement.java, GenerateOutput.java, Matrix_id.java, ReductionAlgorithm.java e Reduction_Guitree.java. Ognuna delle quali è utile per implementare oggetti e funzioni, per ottenere il risultato che ci interessa. Ma andiamo ad esaminarle più in dettaglio.

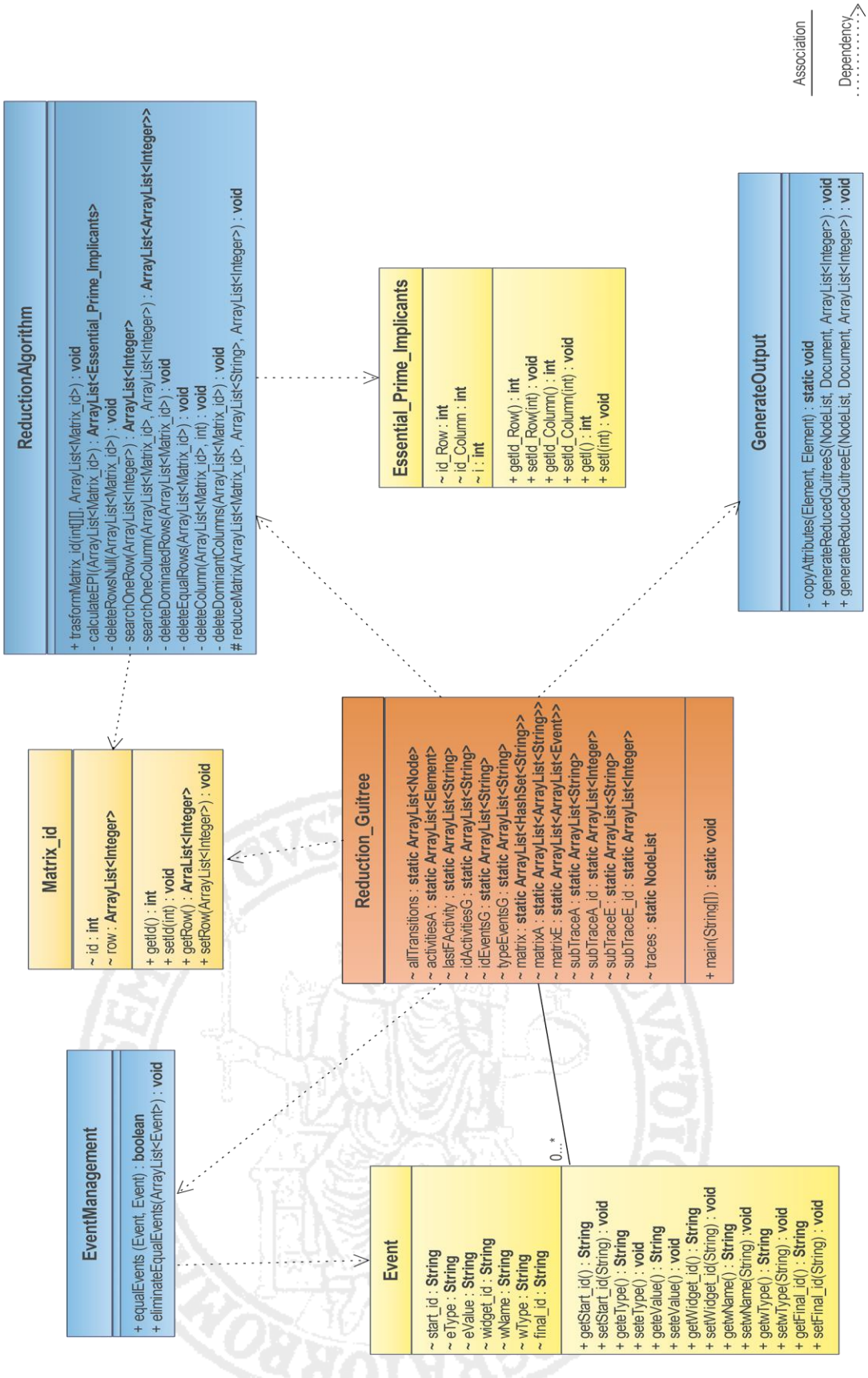


Figura 4.1: *Class Diagram* del Reduction_Tool

4.1 Classe Essential_Prime_Implicants.java

Ho definito questa classe perché ho ritenuto utile definire il tipo di oggetto **Primo Implicante Essenziale** (**PIE**), dovendo adottare un metodo che mi porta ad operare su matrici, per la cui riduzione ho bisogno proprio di individuare i PIE. Ho strutturato la classe in modo da definire al suo interno tre attributi: *id_Row*, *id_Column* ed *i*. Tali parametri mi permettono di:

- *id_Row* mi consente di tenere traccia dell'id della riga della matrice che costituisce il PIE;
- *id_Column* mi consente di tenere traccia dell'id della colonna della matrice a cui appartiene il mintermine del PIE;
- *i* è un intero con cui tengo traccia del valore assegnato al mintermine contenuto nella matrice, che nel nostro caso può essere o 1 o 0.

4.2 Classe Event.java

La classe Event.java definisce il tipo di oggetto **Event**. Poiché ogni evento è identificato:

- Dallo stato di partenza (START_ACTIVITY);
- Dal widget, con cui si interagisce per eseguire quell'evento;
- Dal tipo di evento ad esso associato;
- Dallo stato di arrivo o finale (FINAL_ACTIVITY);

Ho strutturato la classe definendo al suo interno sette attributi:

- Lo *start_id* con cui tengo traccia del valore dello stato di partenza della GUI, identificato dal suo ID;
- Il *widget_id* con cui tengo traccia dell'ID del widget associato all'evento, cioè il "tasto" che viene premuto sulla GUI per eseguire quell'evento;
- Il *wName* con cui memorizzo il nome identificativo associato al widget;
- Il *wType* con cui memorizzo il tipo del widget associato all'evento;
- L'*eType* il cui valore indica proprio il tipo dell'evento;

- L'*eValue* che indica il valore dell'evento;
- Il *final_id* con cui tengo traccia invece del valore dello stato di arrivo della GUI, identificato dal suo ID, una volta eseguito l'evento;

In particolare è da notare che per il widget ho definito tre tipi di attributi (*widget_id*, *wName*, *wType*) per gestire meglio tale parametro, infatti come vedremo nella classe EventManagement.java ci sono alcune situazioni in cui per identificare unicamente il widget di un evento non basta indicarne solo ID, ma sarà necessario indicarne anche il nome e il tipo. Discorso abbastanza analogo va fatto per valutare l'evento vero e proprio, infatti non sempre per distinguere due eventi basta indicarne il tipo (*eType*) ma bisogna considerare anche il valore ad essi assegnato (*eValue*).

4.3 Classe EventManagement.java

La classe EventManagement.java contiene due metodi, che ho definito e implementato, per poter meglio gestire gli eventi:

- Il metodo *equalEvents*, che come suggerisce il nome, è una funzione che permette di effettuare il confronto tra due oggetti di tipo **Event**;
- Il metodo *eliminateEqualEvents* che invece è una funzione che elimina gli eventi uguali, appartenenti ad uno stesso array.

La funzione *equalEvents* riceve in input due eventi (due oggetti di tipo **Event**) che va a confrontare e valuta se questi due elementi sono o meno uguali. Ricevuti i due eventi in ingresso e1 e e2, la funzione ne va a valutare innanzitutto il valore del *widget_id*, questo perché partendo dal file *guitree.xml* abbiamo notato che alcuni EVENT sono caratterizzati da un *widget_id* il cui valore è uguale a -1, che è quindi da ritenersi nullo. In questo caso il valore del *widget_id* diventa irrilevante per valutare la natura dell'evento, e quindi non bisogna più considerare il *widget_id* ma per il widget bisognerà considerare il nome (*wName*) e il tipo (*wType*). Pertanto ho costruito la funzione di modo che faccia due valutazioni diverse e cioè:

1) Se il valore del *widget_id* è diverso da -1 allora il confronto tra eventi può essere fatto normalmente e cioè confrontando rispettivamente i loro: *start_id*, *eType*, *eValue*, *widget_id* e *final_id*. Se questi valori sono uguali fra loro allora possiamo affermare che i due eventi sono uguali e pertanto la funzione restituisce una variabile booleana, ***response***, che assume valore uguale a **true**.

```
7    public boolean equalEvents(Event e1, Event e2){
8        boolean response = false;
9        if((e1.widget_id!="-1")&&(e2.widget_id!="-1")){
10           if((e1.start_id.equals(e2.start_id))&&
11                   (e1.eType.equals(e2.eType))&&
12                   (e1.eValue.equals(e2.eValue))&&
13                   (e1.widget_id.equals(e2.widget_id))&&
14                   (e1.final_id.equals(e2.final_id))){
15               response=true;
16           }
17       }
```

Figura 4.2: funzione *equalEvents*, prima parte

2) Se invece il valore del *widget_id* è uguale a -1 tale parametro viene ignorato e vengono considerati al suo posto altri due parametri per valutare il widget e cioè *wName* e *wType*. In questo caso quindi il confronto tra eventi è fatto confrontando rispettivamente i loro: *start_id*, *eType*, *eValue*, *wName*, *wType* e *final_id*; e se questi valori sono uguali fra loro allora diremo che i due eventi sono uguali, anche in questo caso la funzione restituirà una variabile booleana, ***response*** a cui sarà assegnato valore uguale a **true**.

```
19       else if((e1.widget_id.equals("-1"))&&(e2.widget_id.equals("-1"))){
20           if((e1.start_id.equals(e2.start_id))&&
21                   (e1.eType.equals(e2.eType))&&
22                   (e1.eValue.equals(e2.eValue))&&
23                   (e1.wName.equals(e2.wName))&&
24                   (e1.wType.equals(e2.wType))&&
25                   (e1.final_id.equals(e2.final_id))){
26               response=true;
27           }
28       }
```

Figura 4.3: funzione *equalEvents*, seconda parte

Se invece il confronto tra i due eventi non dà esito positivo, allora in tal caso significa che i due eventi sono diversi e la funzione restituisce sempre la variabile booleana ***response***, che però in tal caso assume valore pari a **false**.

L'altra funzione *eliminateEqualEvents* riceve in input un ArrayList di oggetti di tipo **Event** e in maniera ciclica va a confrontare tra loro gli eventi contenuti in questa lista: se riscontra che due eventi sono uguali, sfruttando la funzione *equalEvents*, allora per evitare ridondanze, nei calcoli, va ad eliminare il secondo evento dalla lista.

```
32      public void eliminateEqualEvents(ArrayList<Event> events){
33          for(int i=0;i<events.size();i++){
34              for(int j=i+1;j<events.size();j++){
35                  if(equalEvents(events.get(i), events.get(j))){
36                      events.remove(j);
37                  }
38              }
39          }
40      }
```

Figura 4.4: funzione *eliminateEqualEvents*

4.4 Classe GenerateOutput.java

La classe GenerateOutput.java contiene tre metodi che ho implementato per generare i file di output, **guitreeReductionStates.xml** e **guitreeReductionEvents.xml**, che mi serviranno per realizzare le Test Suite JUnit ridotte. Innanzitutto ho costruito la prima funzione *copyAttributes* che consente di copiare gli attributi di un elemento (**Element** *from*) ed assegnare gli stessi attributi a un altro elemento (**Element** *to*), con *from* e *to* che sono elementi che la funzione riceve in input. All'interno della funzione ho definito l'oggetto *attributes* di tipo **NamedNodeMap** e che rappresenta una lista non ordinata di nodi, in essa saranno contenuti tutti gli attributi da copiare. Poi ho fatto in modo che scorrendo ciclicamente tutti gli elementi contenuti in questa lista, man mano gli attributi di ogni elemento vengano assegnati a un nuovo elemento *node* di tipo **Attr** e che poi tali attributi, in particolare il NamespaceURI (*getNamespaceURI()*), il nome (*getName()*) e il valore (*getValue()*) siano assegnati all'elemento *to* tramite la funzione *setAttributeNS*.

```
27    private static void copyAttributes(Element from, Element to) {
28        NamedNodeMap attributes = from.getAttributes();
29        for (int i = 0; i < attributes.getLength(); i++) {
30            Attr node = (Attr) attributes.item(i);
31            to.setAttributeNS(node.getNamespaceURI(), node.getName(),
32                    node.getValue());
33        }
34    }
```

Figura 4.5: funzione *copyAttributes*

Le altre due funzioni, *generateReducedGuitreeS* e *generateReducedGuitreeE*, sono sostanzialmente simili, infatti eseguono le stesse operazioni, e differiscono solo per il tipo di file XML che generano e restituiscono in output; infatti la prima funzione genera il file **guitreeReductionStates.xml** e la seconda genera il file **guitreeReductionEvents.xml**. Le funzioni ricevono in ingresso: una lista di nodi *traces2*, che corrisponde alla lista che contiene tutti i nodi prelevati dal *guitree.xml* il cui TagName è "TRACE"; *docG*, che è il documento che rappresenta il file *guitree.xml*, sul quale vado ad operare e con cui posso accedere ai vari nodi, elementi e attributi del file *guitree.xml*; e un **ArrayList** di interi *array* che contiene l'id delle trace che formano il minimo sottoinsieme che ricopre il massimo numero di stati nella prima funzione o di eventi nella seconda funzione. All'interno delle funzioni ho, innanzitutto, convertito l'array di interi in un array di stringhe (*array50*), poiché i valori dei vari id delle tracce che andiamo a leggere dal file *guitree.xml* sono considerati proprio di tipo stringa. Poi sono passato a creare il file XML usando JAXP e in particolare il metodo e/o libreria DOM, creando quindi un **DocumentBuilderFactory** *factoryGR*, un **DocumentBuilder** *builderGR* e il **Document** *documetGR*. Dopodiché ho creato l'elemento radice, la *root*, del mio file XML e collegata allo stesso, ho definito i suoi attributi copiandoli dagli stessi elementi contenuti nel documento *docG*. Per copiare nel nuovo file ridotto solo le trace che mi interessano ho usato un metodo ciclico. In pratica partendo da *traces2* e da *array50*, ho fatto il confronto tra gli elementi di questi due insiemi, considerando in particolare il valore dell'attributo *id*; in questo modo quando trovo l'elemento di *array50* corrispondente in *traces2*, lo copio allegandolo alla *root*.

```
80      public void generateReducedGuitreeE (NodeList traces2, Document docG,
81              ArrayList<Integer>array) throws ParserConfigurationException,
82              SAXException, IOException, TransformerException{
83          ArrayList <String> array50= new ArrayList<String>(array.size());
84          for (Integer myInt : array) {
85              array50.add(String.valueOf(myInt));
86          }
87          DocumentBuilderFactory factoryGR=DocumentBuilderFactory.newInstance();
88          DocumentBuilder builderGR=factoryGR.newDocumentBuilder();
89          Document documentGR=builderGR.newDocument();
90
91          Element root=documentGR.createElement("SESSION");
92          documentGR.appendChild(root);
93          NodeList list=docG.getElementsByTagName("SESSION");
94          Node node=list.item(0);
95          Element eElement1=(Element)node;
96          copyAttributes(eElement1, root);
97          for(int i=0;i<traces2.getLength();i++){
98              Node nNode=traces2.item(i);
99              Element eElement=(Element) nNode;
100             for(int j=0;j<array.size();j++){
101                 if(array50.get(j).equals(eElement.getAttribute("id"))){
102                     root.appendChild(documentGR.importNode(nNode, true));
103                 }
104             }
105         }
```

Figura 4.6: funzione *generateReducedGuitreeE*, prima parte

Una volta creato il documento virtuale poi, ho definito il **TransformerFactory** e il **Transformer** per trasformare l'albero di nodi e ho copiato le proprietà del documento originario *guitree.xml* nel nuovo guitree ridotto. Ho infine creato il **DOMSource** *source* e l'oggetto di tipo **File** *XMLFileGR*, collegato i due elementi tramite uno **StreamResult** *result* in modo da copiare il contenuto della source nel file ridotto finale che mi interessa: o il **guitreeReductionStates.xml** o il **guitreeReductionEvents.xml**.

```
106         TransformerFactory transformerFactory = TransformerFactory.newInstance();
107         Transformer transformer=transformerFactory.newTransformer();
108
109         transformer.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "SESSION");
110         transformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, "guitree.dtd");
111         documentGR.setXmlStandalone(true);
112
113         DOMSource source=new DOMSource(documentGR);
114         File XMLFileGR=new File("guitreeReductionEvents.xml");
115         StreamResult result=new StreamResult(XMLFileGR);
116         transformer.transform(source, result);
117     }
118 }
```

Figura 4.7: funzione *generateReducedGuitreeE*, seconda parte

4.5 Classe Matrix_id.java

La classe Matrix_id.java definisce il tipo di oggetto Matrice. Essa contiene due attributi:

- L'*id* con cui tengo traccia del numero di riga, in maniera opportuna, infatti anche se andiamo ad eliminare le righe dalla matrice, applicando il metodo di riduzione, posso tenere conto del numero o dell'id delle righe che rimangono, e il valore di tale id naturalmente corrisponde al numero della traccia che quindi rimane e farà, eventualmente parte dell'insieme minimo;
- La *row*, un **ArrayList** di interi associato alla matrice; in pratica con questo attributo faccio riferimento, utilizzando opportunamente gli indici di riga della matrice, ad una riga specifica dell'oggetto di tipo **Matrix_id**; utile, come vedremo nella classe ReductionAlgorithm.java, per trovare la posizione esatta del PIE all'interno della riga, o anche per eliminare una riga specifica dalla matrice, per l'operazione di riduzione.

4.6 Classe ReductionAlgorithm.java

La classe ReductionAlgorithm.java contiene dieci metodi che ho opportunamente definito per implementare correttamente l'algoritmo che mi interessa e cioè quello che mi permette di ottenere la riduzione. Tale classe l'ho realizzata per strutturare meglio il tool. Le funzioni contenute in questa classe sono:

- La *transformMatrix_id*, questa funzione riceve in ingresso due matrici, *m1* che è un array bidimensionale quindi una matrice statica e *m2* che è invece un elemento del tipo **ArrayList** di oggetti di tipo **Matrix_id** pertanto è una matrice dinamica. L'obiettivo di tale funzione è quello di trasformare la matrice statica in una matrice dinamica in modo da poter operare su quest'ultima e poterla ridurre. In particolare la funzione va a leggere ciclicamente gli elementi riga per riga contenuti nella matrice *m1* e li copia in un array di interi che ho chiamato *help*, opportunamente inizializzato ad ogni ciclo. Poi definisce un elemento di supporto di tipo

Matrix_id, *support*, a cui vengono assegnati, sempre in maniera ciclica, i valori di *id* e *row* copiandoli da quelli estratti da *m1*, e infine tale oggetto *support* viene aggiunto nella matrice *m2*.

```
8    public void transformMatrix_id(int[][] m1,ArrayList<Matrix_id> m2){
9        for(int i=0;i<m1.length;i++){
10           ArrayList <Integer> help=new ArrayList<Integer>();
11           for(int j=0;j<m1[0].length;j++){
12               help.add(j, m1[i][j]);
13           }
14           Matrix_id support = new Matrix_id();
15           support.setId(i);
16           support.setRow(help);
17           m2.add(support);
18       }
19   }
```

Figura 4.8: funzione *transformMatrix_id*

- La *calculateEPI*, questa funzione calcola i **PIE** nella matrice. Essa riceve in ingresso la matrice dinamica (**ArrayList**<**Matrix_id**>) *m3* su cui deve operare e restituisce in output un array dinamico *aepi* che contiene elementi di tipo **Essential_Prime_Implicants**. In pratica analizza la matrice per righe e per colonne e se riscontra per quella colonna un elemento uguale a 1 incrementa il valore di un opportuno contatore (*counter*).

```
21   private static ArrayList<Essential_Prime_Implicants> calculateEPI
22   (ArrayList<Matrix_id> m3){
23       ArrayList<Essential_Prime_Implicants> aepi =
24                           new ArrayList<Essential_Prime_Implicants>();
25       for(int j=0;j<m3.get(0).getRow().size();j++){
26           int counter1=0;
27           int row=0;
28           for (int i=0;i<m3.size();i++){
29               if(m3.get(i).getRow().get(j)==1){
30                   counter1++;
31                   row=i;
32               }
33           }
```

Figura 4.9: funzione *calculateEPI*, prima parte

Se al termine della colonna risulta che il contatore è proprio uguale a 1, questo indica che ho proprio trovato un **PIE** e va a salvare tale elemento nella variabile *pi* del tipo

Essential_Prime_Implicants, memorizzandone il valore (*i*), la riga (*Id_Row*) e la colonna (*Id_Column*) di appartenenza.

```
34            if(counter1==1){
35                Essential_Prime_Implicants pi=new Essential_Prime_Implicants();
36                pi.setId_Row(m3.get(row).getId());
37                pi.setId_Column(j);
38                pi.setI(row);
39                aepi.add(pi);
40            }
41        }
42        return aepi;
43    }
```

Figura 4.10: funzione *calculateEPI*, seconda parte

- La *deleteRowsNull*, questa funzione riceve in ingresso la matrice dinamica (**ArrayList**<**Matrix_id**>) *m3* e la controlla per verificare se ci sono righe nulle: se è così elimina tali righe per semplificare la matrice poiché tali righe sono irrilevanti ai fini del calcolo dei **PIE**. Anche in questo caso la matrice viene esaminata per righe e per colonne in maniera ciclica, e tramite un contatore *counter0*, si memorizza il numero di 0 contenuti nella riga: se il valore del contatore è uguale al numero di elementi contenuti nella riga significa che la riga è nulla e pertanto la riga viene eliminata dalla matrice *m3*.

```
45    private static void deleteRowsNull (ArrayList<Matrix_id>m3){
46        for(int i=0;i<m3.size();i++){
47            int counter0=0;
48            int row=i;
49            for(int j=0;j<m3.get(i).getRow().size();j++){
50                if(m3.get(i).getRow().get(j)==0){
51                    counter0++;
52                }
53            }
54            if(counter0==m3.get(i).getRow().size()){
55                m3.remove(row);
56            }
57        }
58    }
```

Figura 4.11: funzione *deleteRowsNull*

- La *searchOneRow*, questa funzione riceve in ingresso una riga dinamica *row_1* e restituisce un **ArrayList** di interi *elements* che contiene le posizioni dei valori uguali a 1 nella riga (in pratica il numero della colonna a cui il valore 1 appartiene). La funzione esamina la riga e verifica in maniera ciclica il valore dei vari elementi in essa contenuti: se un valore (*index*) è uguale a 1 lo aggiunge all'array dinamico *elements*.

```
58   private static ArrayList<Integer> searchOneRow (ArrayList<Integer> row_1){
59       ArrayList<Integer> elements = new ArrayList<Integer>();
60       for (int i = 0; i< row_1.size(); i++){
61           int index = row_1.get(i);
62           if (index==1) elements.add(i);
63       }
64       return elements;
65   }
```

Figura 4.12: funzione *searchOneRow*

- La *searchOneColumn*, questa funzione riceve in ingresso la matrice dinamica (**ArrayList**<**Matrix_id**>) *m3*, e l'**ArrayList** di interi *row* e restituisce una matrice dinamica di interi **ArrayList**<**ArrayList**<**Integer**>> *listElements*. In pratica la funzione esamina la matrice *m3* per colonna e se incontra un elemento della colonna uguale a 1 lo aggiunge a un array dinamico *support*; poi una volta che la colonna è terminata, inserisce *support* in una matrice, un **ArrayList** di **ArrayList** *listElements*. In questo modo tengo traccia della posizione dei valori uguali a 1 incontrati in ciascuna colonna della matrice.

```
67   private static ArrayList<ArrayList<Integer>> searchOneColumn
68                           (ArrayList<Matrix_id> m3,ArrayList <Integer>row){
69       ArrayList<ArrayList<Integer>>listElements =
70                                       new ArrayList<ArrayList<Integer>>();
71       for(int j=0;j<row.size();j++){
72           ArrayList<Integer> support = new ArrayList<Integer>();
73           for (int i = 0; i< m3.size(); i++){
74               int index=m3.get(i).getRow().get(j);
75               if (index==1) support.add(i);
76           }
77           listElements.add(support);
78       }
79       return listElements;
80   }
```

Figura 4.13: funzione *searchOneColumn*

- La *deleteDominatedRows*, questa funzione consente di eliminare le righe **dominate** presenti nella matrice dinamica *m3*, che la funzione riceve in input. Tale funzione sfrutta la funzione *searchOneRow*. In pratica di ogni riga *i* della matrice con *searchOneRow* calcolo il vettore *elements_i* che contiene le posizioni degli eventuali 1 presenti nella riga i-esima, poi itero tale processo anche su tutte le altre *k* righe della matrice *m3*, con $k = i+1$, calcolando così per ogni k-esima riga il vettore *elements_k*, che contiene le posizioni degli eventuali 1 presenti nella k-esima riga. A questo punto la funzione effettua un confronto tra *elements_i* ed *elements_k*: in particolare se l'array *elements_i* contiene l'array *elements_k* allora questo significa che la riga k-esima è dominata e quindi la posso eliminare dalla matrice; invece se si verifica l'inverso elimino la riga i-esima dalla matrice.

```
82      private static void deleteDominatedRows(ArrayList<Matrix_id> m3){
83          ArrayList<Integer> elements_i = new ArrayList<Integer>();
84          ArrayList<Integer> elements_k = new ArrayList<Integer>();
85          top : for(int i=0;i<m3.size();i++){
86              int index_i=i;
87              elements_i = searchOneRow(m3.get(index_i).getRow());
88              for(int k=i+1;k<m3.size();k++){
89                  int index_k=k;
90                  elements_k = searchOneRow(m3.get(index_k).getRow());
91                  if(elements_i.containsAll(elements_k)){
92                      m3.remove(index_k);
93                      k--;
94                      continue;
95                  }
96                  else if(elements_k.containsAll(elements_i)){
97                      m3.remove(index_i);
98                      i--;
99                      continue top;
100                 }
101             }
102         }
103     }
```

Figura 4.14: funzione *deleteDominatedRows*

- La *deleteEqualRows*, anche questa funzione riceve in ingresso la matrice dinamica *m3*, di elementi di tipo **Matrix_id**, sulla quale va ad operare. Tale funzione va ad esaminare in maniera ciclica riga per riga tutta la matrice e se trova due righe

uguali, ne elimina la seconda.

```
105    private static void deleteEqualRows (ArrayList<Matrix_id> m3){
106        for(int i=0;i<m3.size();i++){
107            for(int j=0;j<m3.get(0).getRow().size();j++){
108                for(int k=1+i;k<m3.size();k++){
109                    if(m3.get(i).getRow().equals(m3.get(k).getRow())){
110                        m3.remove(k);
111                    }
112                }
113            }
114        }
115    }
```

Figura 4.15: funzione *deleteEqualRows*

- La *deleteColumn*, questa funzione consente di eliminare una colonna specifica dalla matrice. Essa riceve in ingresso la matrice *m3* e il valore dell'indice della colonna che si vuole eliminare dalla matrice (*indexCol).* In maniera sempre ciclica si accede alla colonna individuata dal suo indice e si eliminano man mano tutti gli elementi che appartengono ad essa. Riducendo così la matrice di una colonna.

```
117    private static void deleteColumn (ArrayList<Matrix_id>m3, int indexCol){
118        for(int i=0;i<m3.size();i++){
119            m3.get(i).getRow().remove(indexCol);
120        }
121    }
```

Figura 4.16: funzione *deleteColumn*

- La *deleteDominantColumns*, questa funzione consente di eliminare le colonne **dominanti** dalla matrice dinamica *m3*, che si presenta come un ArrayList di oggetti di tipo **Matrix_id**, che la funzione riceve in ingresso. Tale funzione sfrutta la funzione *searchOneColumn*. In pratica con *SearchOneColumn* trovo un array di array, la matrice dinamica *listElements2*, che contiene le posizioni degli 1 riscontrati nelle colonne della matrice. Poi, in maniera ciclica, si va ad esaminare proprio *listElements2*, copio la riga m-esima in un array *elements_m* e la riga n-esima in un array *elements_n* con $n = m+1$, questo per fare il confronto. Se *elements_m* è uguale o contiene *elements_n*, significa che la colonna m-esima nella matrice *m3* è **dominante** e quindi la vado ad eliminare; inoltre vado ad eliminare

anche la riga m-esima corrispondente da *listElements2*. Se invece si verifica l'inverso vado ad eliminare la colonna n-esima da *m3* e la riga n-esima corrispondente da *listElements2*. Infine se invece la riga m-esima della matrice *listElements2* è vuota elimino la colonna m-esima dalla matrice *m3* e la rispettiva riga *m* da *listElements2*; questo perché tale colonna contiene tutti 0 e quindi è irrilevante ai fini del calcolo dei **PIE**.

```
123     private static void deleteDominantColumns (ArrayList<Matrix_id> m3){
124         ArrayList<ArrayList<Integer>>listElements2 =
125                                     new ArrayList<ArrayList<Integer>>();
126         ArrayList<Integer> elements_m = new ArrayList<Integer>();
127         ArrayList<Integer> elements_n = new ArrayList<Integer>();
128         for (int k=0;k<m3.size();k++){
129             listElements2=searchOneColumn (m3, m3.get(k).getRow());
130         }
131         top2 : for(int m=0;m<listElements2.size();m++){
132             elements_m=listElements2.get(m);
133             top3 :  for(int n=m+1;n<listElements2.size();n++){
134                 elements_n=listElements2.get(n);
135                 if(elements_m.containsAll(elements_n)){
136                     deleteColumn(m3, m);
137                     listElements2.remove(m);
138                     m--;
139                     continue top2;
140                 }
141                 else if(elements_n.containsAll(elements_m)){
142                     deleteColumn(m3, n);
143                     listElements2.remove(n);
144                     n--;
145                     continue top3;
146                 }
147                 else if(listElements2.get(m).isEmpty()){
148                     deleteColumn(m3, m);
149                     listElements2.remove(m);
150                     continue top3;
151                 }
152             }
153         }
154     }
```

Figura 4.17: funzione *deleteDominantColumns*

- La *reduceMatrix*, è la funzione che implementa l'algoritmo di **Quine-McCluskey**, l'algoritmo di riduzione, sfruttando e richiamando le funzioni che abbiamo descritto e che sono presenti nella classe *ReductionAlgorithm.java*. La funzione riceve in input: la matrice *m_Id* che è definita come **ArrayList** di **Matrix_id** ed è

quella sulla quale vado ad operare; e gli array *a1* e *a2* che conterranno i **PIE** che andremo man mano a trovare all'interno della matrice. All'interno della funzione ho definito: un array di **Essential_Prime_Implicants** *aepi* che conterrà i primi implicanti essenziali; un array di interi *list* che per ogni **PIE** contiene gli indici delle colonne coperte dalla riga che costituisce il **PIE**; un TreeSet di interi, *list2*, che uso per ordinare gli indici contenuti in *list* ed evitare ripetizioni; ed un ulteriore TreeSet *aepiOrd*, che uso per ordinare gli elementi dell'array *aepi*. Quest'ultima operazione l'ho fatta perché ho notato che l'ordine è importante per eliminare le colonne della matrice in maniera ordinata e corretta (dall'ultima alla prima, cioè da destra verso sinistra). A questo punto la funzione richiama la funzione *calculateEPI* e calcola così i **PIE** e li inserisce nell'array *aepi*. Poi passa a copiare i primi implicanti essenziali nei due array *a1* e *a2*, in maniera tale da rappresentarli in modo diverso (nel primo caso rappresentati dal valore preceduto dalla lettera "T", che sta per Trace, nel secondo caso rappresentati solo dal valore).

```
156    protected void reduceMatrix (ArrayList<Matrix_id> m_Id,
157                                 ArrayList <String> a1, ArrayList<Integer>a2){
158        do{
159            ArrayList<Essential_Prime_Implicants>aepi =
160                                  new ArrayList<Essential_Prime_Implicants>();
161            ArrayList<Integer> list=new ArrayList<Integer>();
162            TreeSet<Integer> list2=new TreeSet<Integer>();
163            TreeSet<Integer> aepiOrd=new TreeSet<Integer>();
164            int numRowsBefore=m_Id.size();
165            int numColumnsBefore=m_Id.get(0).getRow().size();
166            aepi=calculateEPI(m_Id);
167            for(Essential_Prime_Implicants pi : aepi){
168                a1.add("T"+pi.getId_Row());
169                a2.add(pi.getId_Row());
170                aepiOrd.add(pi.getI());
171            }
```

Figura 4.18: funzione *reduceMatrix*, prima parte

Dopodiché viene invocata la funzione *searchOneRow* per calcolare il valore di *list*, i cui elementi poi vengono copiati in *list2* (in modo da essere ordinati). Individuati gli indici delle colonne e delle righe a cui appartengono i **primi implicanti essenziali**, si passa in maniera ciclica ad eliminare tali colonne prima e tali righe

poi dalla matrice. Successivamente, per ottenere le operazioni desiderate, vengono invocate le altre funzioni in maniera ordinata: *deleteEqualRows*, *deleteDominatedRows*, *deleteDominantColumns*, *deleteRowsNull*.

```
172             ArrayList<Integer>aepiOrd2=new ArrayList<Integer>(aepiOrd);
173             int index=0;
174             for(int n=0;n<aepi.size();n++){
175                 Essential_Prime_Implicants pi1 = aepi.get(n);
176                 index = pi1.getI();
177                 list = searchOneRow(m_Id.get(index).getRow());
178                 for(int m=0;m<list.size();m++){
179                     list2.add(list.get(m));
180                 }
181             }
182             ArrayList<Integer> list3=new ArrayList<Integer>(list2);
183             for(int i=0;i<m_Id.size();i++){
184                 for(int k=list3.size()-1;k>=0;k--){
185                     int index1=0;
186                     index1=list3.get(k);
187                     m_Id.get(i).getRow().remove(index1);
188                 }
189             }
190             for(int n=aepiOrd2.size()-1;n>=0;n--){
191                 int index2 =0;
192                 index2=aepiOrd2.get(n);
193                 m_Id.remove(index2);
194             }
195             deleteEqualRows(m_Id);
196             deleteDominatedRows(m_Id);
197             deleteDominantColumns(m_Id);
198             deleteRowsNull(m_Id);
199             if(!m_Id.isEmpty()){
200                 int numRowsAfter=m_Id.size();
201                 int numColumnsAfter=m_Id.get(0).getRow().size();
202                 if((numColumnsBefore==numColumnsAfter)
203                                         &&(numRowsBefore==numRowsAfter))
204                     break;
205             }
206         }while(!m_Id.isEmpty());
207         for(int i=0;i<m_Id.size();i++){
208             a1.add("T"+m_Id.get(i).getId());
209             a2.add(m_Id.get(i).getId());
210         }
211     }
212
213 }
```

Figura 4.19: funzione *reduceMatrix*, seconda parte

Tutte queste operazioni poi vengono ripetute in maniera ciclica, tramite il processo do-while, ricordando che nei passi successivi in realtà i **nuovi primi implicanti** che vengono

trovati sono detti **secondari**; il processo continua finché la matrice non diventa vuota, oppure se non è vuota si controlla la sua dimensione e se dopo la riduzione questa non cambia significa che non può essere più ridotta e quindi si esce dal ciclo. Il tutto proprio come previsto e indicato dall'algoritmo di riduzione che abbiamo adottato. Usciti dal ciclo, poi come ultima operazione, verranno aggiunti in *a1* e *a2* anche i primi implicanti secondari trovati.

4.7 Classe Reduction_Guitree.java

La classe Reduction_Guitree.java è la classe principale del tool, ed è quella che contiene la main. All'interno della classe sono, innanzitutto, definiti tutta una serie di array dinamici (**ArrayList**), che contengono elementi di diverso tipo (**Node**, **Element**, **String**, etc.), sia monodimensionali che bidimensionali. Si tratta di costrutti che saranno utili o per contenere elementi su cui andremo a lavorare o semplicemente per memorizzare dei risultati e/o valori opportunamente calcolati, come vedremo più in dettaglio, man mano che li incontreremo durante la descrizione della classe. Il metodo che ho adottato per accedere ai file XML e più precisamente agli elementi contenuti in questi file, è quello che si basa sull'uso della libreria **DOM**, con cui i vari elementi presenti nel file, individuati da appositi TAG, sono visti come dei nodi che costituiscono una struttura ad albero, rappresentato da una classe che implementa l'interfaccia *org.w3c.dom.Node*. Questa interfaccia rappresenta un singolo nodo nell'albero del documento e fornisce metodi per gestire i vari nodi e l'albero stesso; con i nodi opportunamente legati tra di loro da una specifica relazione. Pertanto l'esplorazione del documento XML è fatta in maniera sequenziale partendo dal primo TAG, la root, e scendendo nelle varie ramificazioni. Ho preferito usare **DOM**, poiché con questa libreria il documento XML viene caricato completamente in memoria e così è stato possibile lavorare su di esso come meglio preferivo, avendo inoltre la possibilità di modificare e/o creare documenti XML; cosa che non sarebbe stata possibile con la libreria **SAX**. L'idea di fondo che seguiamo è la seguente: con JAXP per poter accedere ai nodi, agli elementi e agli attributi del documento

è necessario definire un **DocumentBuilderFactory**, un **DocumentBuilder** e da esso un **Document**. Inoltre dovendo operare su due file XML, *activities.xml* e *guitree.xml* ho provveduto a definire tali parametri per entrambi.

Per il primo file ho *factoryA*, *builderA* e *documentA*; per il secondo file ho *factoryG*, *builderG* e *documentG*. Partendo dal file *activities.xml*, innanzitutto ho provveduto a modificarlo facendo in modo che l'intestazione "<?xml version="1.0" encoding="UTF-8"?>", non venisse sempre ripetuta ad inizio di ogni Activity e ho aggiunto la *root*. Poi nella classe ho definito l'elemento *rootA* a cui ho assegnato come valore proprio il valore dell'elemento radice nel file. Poi ho definito una lista di nodi *childrenA* che contiene tutti i nodi figli della *root*. In maniera ciclica ho fatto in modo di accedere ai vari nodi della lista singolarmente (*childA*), e ho salvato il loro valore ELEMENT_NODE all'interno dell'array *activitiesA* che contiene oggetti di tipo **Element**.

```
139             File outNew = new File("activities_new.xml");
140             DocumentBuilderFactory factoryA=
141                                 DocumentBuilderFactory.newInstance();
142             DocumentBuilder builderA=factoryA.newDocumentBuilder();
143             Document documentA=builderA.parse(outNew);
144
145             Element rootA=documentA.getDocumentElement();
146             NodeList childrenA=rootA.getChildNodes();
147
148             for(int i=0;i<childrenA.getLength();i++){
149                 Node childA=childrenA.item(i);
150                 if(childA.getNodeType()==Node.ELEMENT_NODE){
151                     Element eElementA=(Element)childA;
152                     activitiesA.add(eElementA);
153                     NodeList descriptionA=childA.getChildNodes();
154                     for(int j=0;j<descriptionA.getLength();j++){
155                         NodeList widgetListA=
156                              descriptionA.item(j).getChildNodes();
157                         for(int k=0;k<widgetListA.getLength();k++){
158                             widgetListA.item(k);
159                         }
160                     }
161                 }
162             }
```

Figura 4.20: Algoritmo per accedere agli elementi del file *activities.xml*

Per accedere agli altri nodi "figli" ho definito la lista di nodi (**NodeList**) *descriptionA* che contiene tutti i nodi figli dei nodi *childA*, e la lista di nodi *widgetListA* che contiene tutti i

nodi figli dei nodi *descriptionA* e ho poi definito il nodo *widgetA* per poter accedere singolarmente ai nodi contenuti nella lista *widgetListA*. Ottenendo in questo modo un riferimento completo a tutti i nodi contenuti nel *documentA* e/o nel file *activities.xml*. Sono poi passato al file *guitree.xml*.

```
173     traces=documentG.getElementsByTagName("TRACE");
174     int dim=0;
175     int[][] completeA_Matrix = new int[traces.getLength()]
176                                             [traces.getLength()+3];
177     for (int i=0;i<traces.getLength();i++){
178         ArrayList<String> activitiesTrace = new ArrayList<String>();
179         ArrayList<String> idEventsTrace = new ArrayList<String>();
180         ArrayList<String> typeEventTrace = new ArrayList<String>();
181         ArrayList<Event> eventsTrace = new ArrayList<Event>();
182     Node trace=traces.item(i);
183     NodeList transitions=trace.getChildNodes();
184     for(int j=0;j<transitions.getLength();j++){
185         Event e=new Event(null, null, null, null, null, null, null);
186         Node transition=transitions.item(j);
187         if(transition.getNodeType()==Node.ELEMENT_NODE){
188             NodeList childrenTransition=transition.getChildNodes();
189             allTransitions.add(transition);
190             for(int k=0;k<childrenTransition.getLength();k++){
191             Node childTransition=childrenTransition.item(k);
192             if(childTransition.getNodeName().equals("START_ACTIVITY")){
193                 Node start_Activity=childTransition;
194                 Element eElementS=(Element)start_Activity;
195                         e.setStart_id(eElementS.getAttribute("id"));
196                         idActivitiesG.add(eElementS.getAttribute("id"));
197                         activitiesTrace.add(eElementS.getAttribute("id"));
198             }
199             else if(childTransition.getNodeName().equals("FINAL_ACTIVITY")){
200                 Node final_Activity=childTransition;
201                 Element eElementF=(Element)final_Activity;
202                     e.setFinal_id(eElementF.getAttribute("id"));
203                     lastFActivity.add(dim, eElementF.getAttribute("id"));
204                     dim=dim++;
205                     }
```

Figura 4.21: Algoritmo per accedere agli elementi del file *guitree.xml*, prima parte

Ho innanzitutto inizializzato la lista di nodi *traces*, definita precedentemente e che contiene tutti i nodi di documentG caratterizzati da un TagName uguale a "TRACE". A questo punto seguendo un procedimento abbastanza analogo a quello visto precedentemente per accedere ai nodi di documentA, ho definito il nodo *trace* per poter accedere singolarmente ai vari nodi contenuti nella lista *traces*. Per poter accedere ai nodi

più profondi ho poi definito:

- La lista di nodi *transitions*, che contiene i nodi figli dei nodi *trace*, identificati dal TagName TRANSITION; e il nodo *transition*, con cui indico il generico nodo appartenente a questa lista. Inoltre ho provveduto a memorizzare le varie *transition* nell'array *allTransitions* in modo da tenerne traccia;
- La lista di nodi *childrenTransition* che contiene i nodi figli dei nodi *transition*, e il nodo *childTransition* per accedere al singolo nodo appartenente alla lista.

Sono poi passato a controllare i nodi *childTransition* in quanto quelli che ci interessano per il nostro metodo sono quelli con TagName uguale a: "START_ACTIVITY", "FINAL_ACTIVITY" e "EVENT". In particolare se:

- Il nome del nodo (*NodeName*) è uguale a "START_ACTIVITY" in tal caso viene creato il nodo *start_Activity* e il suo attributo "*id*" viene copiato nell'array *activitiesTrace*, che quindi andrà a contenere gli start_Activity per ogni traccia. Inoltre il valore dell'*id* viene assegnato anche al campo *Start_id* dell'evento *e*;
- Il nome del nodo (*NodeName*) è uguale a "FINAL_ACTIVITY" in tal caso viene creato il nodo *final_Activity* e il suo attributo "*id*" viene copiato nell'array *lastFActivity*, che contiene solo l'ultimo final_Activity per ogni traccia. Inoltre il valore dell'*id* viene assegnato anche al campo *Final_id* delle evento *e*;
- Il nome del nodo (*NodeName*) è uguale a "EVENT" in tal caso viene creato il nodo *event* e i suoi attributi "*type*" e "*value*" vengono copiati nei campi *eType* e *eValue* dell'evento *e*. Inoltre in questo caso vengono creati la lista di nodi *events* che contiene i nodi figli di ciascun nodo *event* e il nodo *widget* per accedere ai vari elementi della lista singolarmente. Inoltre per ciascuno di questi nodi vengono copiati gli attributi "*id*", "*name*" e "*type*" nei rispettivi campi dell'evento *e*. I valori dei vari *id* riscontrati saranno memorizzai negli array *idEventsTrace*, che contiene gli *id* degli eventi per ogni singola traccia, ed *idEventsG* che contiene tutti gli *id* degli eventi riscontrati nel file; mentre i valori dei *type* saranno memorizzati in altri due array, *typeEventTrace*, che contiene i *type* degli eventi per ogni singola traccia, e *typeEventG* che invece contiene tutti i *type* degli eventi trovati nel file.

```
206             else if(childTransition.getNodeName().equals("EVENT")){
207                         Node event=childTransition;
208                         Element eElementEvent=(Element)event;
209                         e.seteType(eElementEvent.getAttribute("type"));
210                         e.seteValue(eElementEvent.getAttribute("value"));
211                         NodeList events=event.getChildNodes();
212                         for(int h=0;h<events.getLength();h++){
213                             Node widget=events.item(h);
214                             if(widget.getNodeType()==Node.ELEMENT_NODE){
215                                 Element eElementW=(Element)widget;
216                                 e.setWidget_id(eElementW.getAttribute("id"));
217                                 e.setwName(eElementW.getAttribute("name"));
218                                 e.setwType(eElementW.getAttribute("type"));
219                             }
220                         }
221                         idEventsTrace.add(eElementEvent.getAttribute("id"));
222                         idEventsG.add(eElementEvent.getAttribute("id"));
223                         typeEventTrace.add(eElementEvent.getAttribute("type"));
224                         typeEventsG.add(eElementEvent.getAttribute("type"));
225                             }
226                         }
227                         eventsTrace.add(e);
228                         eventsG.add(e);
229                     }
230                 }
231             idActivitiesG.add(lastFActivity.get(dim));
232             activitiesTrace.add(lastFActivity.get(dim));
233             HashSet <String> array7Hash = new HashSet <String>(activitiesTrace);
234             matrix.add(array7Hash);
235             matrixA.add(activitiesTrace);
236             matrixE.add(eventsTrace);
```

Figura 4.22: Algoritmo per accedere agli elementi del file *guitree.xml*, seconda parte

I vari eventi *e*, opportunamente creati durante queste operazioni, verranno inseriti di volta in volta, negli array *eventsTrace*, che contiene gli eventi per ogni traccia, ed *eventsG*, che contiene tutti gli eventi riscontrati nel file *guitree.xml*. Poi ho fatto in modo di riempire anche altri due array: l'*idActivitiesG* che alla fine del processo conterrà tutte le Activity che si trovano nel file *guitree.xml* e l'*activitiesTrace* che invece conterrà tutte le start_Activity più l'ultima final_Activity per ogni singola trace. Infine ho provveduto a inserire i vari Activity ed Event trovati in due matrici diverse *matrixA* e *matrixE*. A questo punto ho provveduto a creare un metodo ciclico che mi consente di riempire la matrice *completeA_Matrix*, che ho definito precedentemente come matrice statica e come array bidimensionale. In pratica sfruttando i dati raccolti negli array *activitiesTrace* ed *activitiesA* ho confrontato in maniera sequenziale gli elementi in essi contenuti e se l'm-

esimo elemento di *activitiesTrace* è uguale all'attributo *id* dell'l-esimo elemento di *activitiesA*, ho assegnato all'elemento, che occupa la posizione data dalla riga i e colonna n-1, valore pari a 1; mentre invece gli altri elementi manterranno il loro valore di inizializzazione, cioè 0. Inoltre sempre all'interno di questo ciclo ho gestito anche altri tre possibili stati che sono indicati rispettivamente come: *exit*, *crash* e *fail*.

```
239 for(int m=0;m<activitiesTrace.size();m++){
240 for(int l=0;l<activitiesA.size();l++){
241 if(activitiesTrace.get(m).equals(activitiesA.get(l).getAttribute("id"))){
242                 String stringNumber=activitiesTrace.get(m).substring(1);
243                 int n=Integer.parseInt(stringNumber);
244                 completeA_Matrix[i][n-1]=1;
245               }
246               else if(activitiesTrace.get(m).equals("exit")){
247                   completeA_Matrix[i][completeA_Matrix[0].length-3]=1;
248               }
249               else if(activitiesTrace.get(m).equals("crash")){
250                   completeA_Matrix[i][completeA_Matrix[0].length-2]=1;
251               }
252               else if(activitiesTrace.get(m).equals("fail")){
253                   completeA_Matrix[i][completeA_Matrix[0].length-1]=1;
254               }
255           }
256       }
```

Figura 4.23: Algoritmo che permette di riempire la matrice trace/stati completa

Infatti ho indicato che se il valore dell'm-esimo elemento dell'*activitiesTrace* è proprio uguale a uno di questi tre stati il valore dell'elemento corrispondente nella matrice, e cioè quello che si trova nella riga i-esima e che appartiene alla colonna corrispondente a tale stato, deve essere impostato a 1. Successivamente ho implementato altri due metodi: il primo che permette di stampare la matrice in un file di testo e il secondo che permette di stampare la matrice in un file XML. Nel primo ho innanzitutto creato il file di testo, *copyprintwriterCAM.txt*, poi ho creato il **BufferedWriter** *bufCAM* e il **PrintWriter** *printoutCAM*. Il primo realizza un buffer che consente di accedere al file ad esso associato e il secondo che mi consente proprio di scrivere nel file. Infine in maniera ciclica ho riempito questo file con i dati della matrice trace/stati, assegnando alle varie righe il nome dato da "trace + numero riga" e alle colonne il nome dato dal nome dell'Activity. Ho provveduto anche ad organizzare i dati per righe e tabulazioni.

```
292     FileWriter fileoutCAM = new FileWriter ("copyprintwriterCAM.txt");
293     BufferedWriter bufCAM = new BufferedWriter (fileoutCAM);
294     PrintWriter printoutCAM = new PrintWriter(bufCAM);
295     printoutCAM.print("\t");
296     for(int k=1;k<=traces.getLength();k++){
297         printoutCAM.print("\ta"+ k);
298     }
299     printoutCAM.print("\texit");
300     printoutCAM.print("\tcrash");
301     printoutCAM.print("\tfail");
302     printoutCAM.println("\n");
303     for(int i=0;i<traces.getLength();i++){
304         if (i<10){
305             printoutCAM.print("Trace "+i+"  : ");
306         }
307         else printoutCAM.print("Trace "+i+" : ");
308         for(int j=0;j<completeA_Matrix[0].length;j++){
309             printoutCAM.print("\t" + completeA_Matrix[i][j]+", ");
310         }
311         printoutCAM.println("\n");
312     }
313     printoutCAM.close();
```

Figura 4.24: Algoritmo per la stampa della matrice trace/stati in un file di testo

Nel secondo metodo ho utilizzato un nuovo documento, creato precedentemente, *documentM* che conterrà gli elementi della matrice che andrò a rappresentare in un file XML. Per prima cosa ho creato l'elemento radice del file XML, *rootM* che ho poi collegato a *documentM*. Dopo in maniera ciclica ho creato gli elementi *traceM*, che sono i nodi che rappresenteranno le trace contenute nel file *guitree.xml* e per ciascuno di essi ho creato i suoi attributi *attributeT* opportunamente definiti. Gli attributi poi vengono collegati ai rispettivi nodi *traceM* e tali nodi vengono collegati a loro volta alla *rootM*. Adottando sempre un metodo ciclico ho poi creato i nodi *activity*, e i loro attributi *attributeA*, settando opportunamente i loro valori, considerando anche i valori *exit*, *crash* e *fail*. Infine ho collegato gli attributi ai nodi *activity* e tali nodi al proprio nodo *tarceM*. In questo modo ho creato il documento virtuale che contiene l'albero di nodi del file XML. L'idea è stata quella di ricavare un documento virtuale nel quale per ogni traccia sono indicati solo gli Activity rilevanti e non tutti: cioè per ogni traccia sono indicati gli *id* degli Activity che per quella traccia assumono valore uguale a 1.

```
318     Element rootM=documentM.createElement("SESSION");
319     documentM.appendChild(rootM);
320     for(int i=0;i<traces.getLength();i++){
321         Element traceM=documentM.createElement("TRACE");
322         Attr attributeT=documentM.createAttribute("id");
323         String index2=Integer.toString(i);
324         attributeT.setValue(index2);
325         traceM.setAttributeNode(attributeT);
326         rootM.appendChild(traceM);
327         for(int j=0;j<completeA_Matrix[0].length;j++){
328             Element activity=documentM.createElement("ACTIVITY");
329             Attr attributeA=documentM.createAttribute("id");
330             if(j==completeA_Matrix[0].length-2){
331                 attributeA.setValue("exit");
332             }
333             else if(j==completeA_Matrix[0].length-1){
334                 attributeA.setValue("crash");
335             }
336             else if(j==completeA_Matrix[0].length){
337                 attributeA.setValue("fail");
338             }
339             else attributeA.setValue("a"+(j+1));
340             Attr valueA=documentM.createAttribute("value");
341             String s=Integer.toString(completeA_Matrix[i][j]);
342             valueA.setNodeValue(s);
343             if(s.equals("1")){
344                 activity.setAttributeNode(attributeA);
345                 activity.setAttributeNode(valueA);
346                 traceM.appendChild(activity);
347             }
348         }
349     }
```

Figura 4.25: Algoritmo per la creazione del documento virtuale *documentM*

A questo punto ho creato il **TransformerFactory**, il **Transformer** e la **DOMSource** per poter gestire *documentM* e poi ho creato il file *matrix_XML.xml* nel quale verrà copiato il contenuto di *documentM* ottenendo così il file XML desiderato. Sono poi passato a creare un'altra matrice statica, la matrice *supportedMatrix*, utilizzando un metodo analogo a quello visto per creare la matrice *completeA_Matrix*. Ho creato questa matrice per semplificare i calcoli di riduzione, infatti essa contiene tutte le righe ma meno colonne, poiché tiene traccia solo degli stati effettivamente riscontrati e coperti. In seguito ho provveduto a stampare anche tale matrice in un file di testo *copyprintwriterSM.txt*, adottando un metodo del tutto simile a quello descritto precedentemente per stampare la matrice *completeA_Matrix*.

Ho infine copiato la matrice statica *completeA_Matrix* in una matrice dinamica *matrix3* che si presta meglio alle operazioni di riduzione.

```
359     ArrayList<String> fromArrayListG2 =
360             new ArrayList<String>(fromArrayListG);
361     int [][] supportedMatrix =
362             new int [traces.getLength()][fromArrayListG2.size()];
363     for(int i=0;i<matrixA.size();i++){
364         for(int j=0;j<matrixA.get(i).size();j++){
365             for(int k=0;k<fromArrayListG2.size();k++){
366                 if(matrixA.get(i).get(j).
367                         equals(fromArrayListG2.get(k))){
368                     supportedMatrix[i][k]=1;
369                 }
370              }
371         }
372     }
```

Figura 4.26: Algoritmo che crea la matrice *supportedMatrix*

```
362     FileWriter fileoutSM = new FileWriter("copyprintwriterSM.txt");
363     BufferedWriter bufSM = new BufferedWriter(fileoutSM);
364     PrintWriter printoutSM = new PrintWriter(bufSM);
365     printoutSM.print("\t");
366     for(int k=0;k<activitiesA.size();k++){
367       printoutSM.print("\t"+activitiesA.get(k).getAttribute("id"));
368     }
369     for(int h=0;h<fromArrayListG2.size();h++){
370         if(fromArrayListG2.get(h).equals("exit"))
371             printoutSM.print("\texit");
372         if(fromArrayListG2.get(h).equals("crash"))
373             printoutSM.print("\tcrash");
374         if(fromArrayListG2.get(h).equals("fail"))
375             printoutSM.print("\tfail");
376     }
377     printoutSM.println("\n");
378     for(int i=0;i<traces.getLength();i++){
379         if(i<10){
380             printoutSM.print("Trace "+i+"  : ");
381         }
382         else printoutSM.print("Trace " +i+" :");
383         for(int j=0;j<supportedMatrix[0].length;j++){
384             printoutSM.print("\t" +supportedMatrix[i][j]+", ");
385         }
386         printoutSM.println("\n");
387     }
388     printoutSM.close();
```

Figura 4.27: Algoritmo che permette di stampare la *supportedMatrix* in un file di testo

Sono poi passato a valutare e calcolare la matrice Trace/Eventi. Per gestire gli eventi sono partito col considerare l'array *eventsG*, sul quale ho eseguito la funzione *eliminateEqualEvents*, in questo modo vado a considerare tutti gli eventi trovati nel file *guitree.xml* evitando che si ripetano (ciascun evento verrà considerato una sola volta). Poi sfruttando la funzione *equalEvents*, con un metodo ciclico ho provveduto a riempire la matrice *completeE_Matrix* definita precedentemente come array bidimensionale statico.

```
401     for(int n=0;n<matrixE.size();n++){
402         for(int m=0;m<matrixE.get(n).size();m++){
403             for(int k=0;k<eventsG.size();k++){
404                 if(new EventManagement().
405                    equalEvents(matrixE.get(n).get(m),eventsG.get(k))){
406                    completeE_Matrix[n][k]=1;
407                 }
408             }
409         }
410     }
```

Figura 4.28: Algoritmo che crea la matrice *completeE_Matrix*

Successivamente sono passato a stampare tale matrice all'interno di un file di testo, *copyprintwriterCEM.txt*, adottando per la stampa lo stesso metodo descritto e usato precedentemente per creare e riempire il file *copyprintwriterCAM.txt*; solo che in questo caso il nome assegnato alle colonne è dato dall'id dell'evento.

```
413     FileWriter fileoutMEC = new FileWriter ("copyprintwriterCEM.txt");
414     BufferedWriter bufMEC = new BufferedWriter (fileoutMEC);
415     PrintWriter printoutMEC = new PrintWriter(bufMEC);
416     printoutMEC.print("\t");
417     for(int k=0;k<eventsG.size();k++){
418         printoutMEC.print("\te"+ k);
419     }
420     printoutMEC.println("\n");
421     for(int i=0;i<traces.getLength();i++){
422         if (i<10){
423             printoutMEC.print("Trace "+i+"  : ");
424         }
425         else printoutMEC.print("Trace "+i+" : ");
426         for(int j=0;j<completeE_Matrix[0].length;j++){
427             printoutMEC.print("\t" + completeE_Matrix[i][j]+", ");
428         }
429         printoutMEC.println("\n");
430     }
431     printoutMEC.close();
```

Figura 4.29: Algoritmo che stampa la matrice *completeE_Matrix* in un file di testo

Creata la matrice statica e stampata, l'ho trasformata in una matrice dinamica, poiché, come accennato prima, quest'ultimo tipo di matrice si presta meglio alle operazioni di riduzione. Ho quindi richiamato la funzione *reduceMatrix* della classe ReductionAlgorithm.java, e l'ho eseguita:

1. Prima sulla matrice *matrix3* in modo da applicare il metodo di riduzione alla matrice Trace/Stati e ottenere così **il sottoinsieme minimo di trace che copre il massimo numero** di stati; sottoinsieme che viene memorizzato nell'array *subTraceA*;
2. Poi sulla matrice *matrix4* in modo da applicare il metodo di riduzione anche alla matrice Trace/Eventi e ottenere così anche il **sottoinsieme minimo di trace che copre il massimo numero di eventi**; sottoinsieme opportunamente memorizzato nell'array *subTraceE*.

Infine ho richiamato le funzioni *generateReducedGuitreeS* e *generateReducedGuitreeE* della classe GenerateOutput.java, in modo da generare i due file guitree "ridotti": *guitreeReductionStates.xml* e *guitreeReductionEvents.xml*, che sono appunto quelli che ci interessano per ricavare la Test Suite JUnit ridotta.

```
448 new ReductionAlgorithm().transformMatrix_id(completeE_Matrix, matrix4);
449 new ReductionAlgorithm().reduceMatrix(matrix3, subTraceA, subTraceA_id);
450 new ReductionAlgorithm().reduceMatrix(matrix4, subTraceE, subTraceE_id);
451 System.out.println();
452 System.out.println("The minimum subset of traces that"
453         + " covers the maximum amount of activity is: : ");
454 System.out.println(subTraceA);
455 System.out.println("The number of traces is : "+subTraceA.size());
456 System.out.println();
457 System.out.println("The minimum subset of traces that"
458         + "covers the maximum amount of events is : ");
459 System.out.println(subTraceE);
460 System.out.println("The number of traces is : "+subTraceE.size());
461 new GenerateOutput().generateReducedGuitreeS(traces, documentG, subTraceA_id);
462 new GenerateOutput().generateReducedGuitreeE(traces, documentG, subTraceE_id);
```

Figura 4.30: Ultima parte dell'Algoritmo di Riduzione

Concludo ricordando che all'interno di questa classe ho fatto anche in modo che venissero stampate nella console:

- Alcune informazioni utili ricavate dall'ispezione dei due file di partenza, *activities.xml* e *guitree.xml*, come: il numero di trace individuate nel file

guitree.xml (rappresentato dalla dimensione della lista *traces*), il numero di transizioni individuate sempre nel file *guitree.xml* (rappresentato dalla dimensione di *allTransitions*), il numero di stati riscontrati nel file *guitree.xml* (rappresentato dalla dimensione di *fromArrayListG*), il numero di stati riscontrati nel file *activities.xml* (rappresentato dalla dimensione di *id_activitiesA*) e il numero di eventi differenti riscontrati nel file *guitree.xml*, considerando solo il loro *id* (rappresentato dalla dimensione di *eventsG*);

- Il **sottoinsieme minimo di trace che copre il massimo numero di stati**, indicando quali e quante sono queste tracce (stampando il contenuto dell'array *subTraceA* e la sua dimensione);
- Il **sottoinsieme minimo di tracce che copre il massimo numero di eventi**, indicando quali e quante sono queste tracce (stampando quindi il contenuto dell'array *subTraceE* e la sua dimensione);

Il tutto per tenere traccia e visualizzare a schermo tutte le informazioni più importanti che ricavo man mano che analizzo i file ed eseguo l'operazione di riduzione, ma anche per poter poi fare un confronto con i risultati ottenuti dai file e dai test non ridotti, fatti in passato.

Capitolo 5

Sperimentazione, Testing e Analisi

Attraverso il mio tool, ampiamente descritto nel capitolo precedente, ottengo quindi due file guitree "ridotti". Questi due file sono quelli da cui partiamo per fare i nostri esperimenti, controlli, test e analisi. Ricordando che l'approccio che abbiamo adottato per il testing è quello **Sistematico**.

Il processo di testing si divide essenzialmente in tre passaggi:

1. Fase preliminare;
2. Fase di Ripping;
3. Fase di post-Elaborazione/Analisi;

Durante tale processo andiamo ad utilizzare diversi strumenti utili come: il GUI Ripping tool, l'AVD, l'ADB e la libreria EMMA, che ci aiutano nel testing e nelle nostre valutazioni. Nella prima fase il tester definisce e salva le precondizioni dell'applicazione, impostando opportunamente l'AVD. In particolare è necessario scegliere innanzitutto la versione del sistema operativo Android che si vuole utilizzare. La scelta viene fatta in funzione anche delle app che si vogliono testare, infatti analizzando il file *AndroidManifest.xml* dell'app, si può conoscere la versione "minima" del sistema richiesta da quell'applicazione affinché funzioni correttamente; naturalmente tale informazione viene rilasciata dagli sviluppatori. Inoltre la scelta degli sviluppatori è fatta anche considerando quella che è la piattaforma Android più diffusa, così da potersi rivolgere a più utenti e/o consumatori. Poi bisogna installare, sul dispositivo virtuale, l'app che si

vuole testare e creare un'immagine del sistema, detta immagine *Snapshot*, che definisce le condizioni che saranno ripristinate e richiamate ogni volta che il processo di ripping viene rieseguito. Inoltre si va a generare la nuova Test Suite JUnit ridotta a partire da uno dei due file *guitree* che abbiamo ottenuto precedentemente: Il **guitreeReductionStates.xml** o il **guitreeReductionEvents.xml**. Nella seconda fase, sfruttando la Test Suite JUnit generata, l'interfaccia dell'applicazione viene esplorata iterativamente e automaticamente, in modo da testare l'applicazione stessa. Inoltre usando l'ADB, per comunicare con l'emulatore, vengono generati anche un file di *log.txt*, che contiene tutti gli eventuali errori, failure o crash riscontrati e un file *test.txt* che invece contiene i dati di copertura. Nella terza e ultima fase abbiamo raccolto e analizzato i dati per confrontare le tre tecniche di testing:

1. Quella di **Base**;
2. Quella di **Riduzione per Stati**;
3. Quella di **Riduzione per Eventi**;

In particolare abbiamo puntato l'attenzione su alcuni parametri specifici per raggiungere i nostri obiettivi come:

- **Total Events:** che indica il numero di eventi totali eseguiti durante il test;
- **Different Events:** che indica il numero di eventi effettivamente differenti;
- **Max Coverage:** indica la percentuale di codice coperto dal test ed è usato per valutare l'**efficacia** del test;
- **Efficiency:** che è calcolata dal rapporto tra Max Coverage e Total Events ed è usata per valutare l'**efficienza** del test;
- **# of States:** che è il numero di stati coperti durante il test;
- **# of Traces:** che rappresenta il numero di casi di test, nella Test Suite;
- **Time:** per valutare il tempo impiegato per eseguire il test.

Nei nostri esperimenti abbiamo fatto riferimento a 14 applicazioni che andremo ora man mano ad esaminare in dettaglio. Sono state scelte tutte applicazioni Open Source così da poter avere accesso al codice sorgente e calcolarne la copertura; inoltre sono state scelte applicazioni per le quali sul sito sono riportati eventuali bug, in modo da notificare, eventualmente, nuovi bug, qualora ne fossero riscontrati.

5.1 Aard Dictionary 1.4.1

Aard Dictionary [28] (o semplicemente AardDict) è un dizionario, o più precisamente è un programma che cerca parole in maniera molto rapida, interagendo con diversi dizionari, quali ad esempio Wikipedia. E' in grado di ricercare parole in più dizionari e in diverse lingue, senza interruzione e funziona anche come lettore offline delle pagine di Wikipedia. Per l'esecuzione del test ripping abbiamo usato il dizionario Wikiquote in italiano.

Tabella 5.1: Valori test Sistematico – Aard Dictionary

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	217	49	44,25%	0,2%	13	49	63m
Riduzione per Stati	25	12	32,72%	1,3%	13	6	8m
Riduzione per Eventi	181	49	44,22%	0,24%	13	39	51m

Nella tabella 5.1 sono riportati i risultati dei test eseguiti. Nel caso base si ricopriva il **44,25%** del codice sorgente, eseguendo **in totale 217 eventi**, di cui **49** sono **differenti**, con un'**efficienza** pari allo **0,2%** circa, controllando **13 stati** ed eseguendo **49 trace** in **63 minuti**. Adottando la tecnica di riduzione per stati abbiamo che la Test Suite JUnit si riduce notevolmente in termini di trace, infatti da 49 si passa a solo **6 trace**. Eseguendo questa Test Suite si è coperto quasi il **33%** del codice sorgente, eseguendo **in totale 25 eventi**, di cui solo **12** sono **differenti**, con un'**efficienza** del test uguale a **1,3%** circa, ispezionando lo stesso numero di stati ma impiegando solo **8 minuti**. Infine, con la tecnica di riduzione per eventi, abbiamo ottenuto una Test Suite JUnit di **39 trace**. In questo caso viene coperto il **44,22%** del codice sorgente, che è un valore vicinissimo a quello che si ha con la tecnica di base e vengono controllati sempre **13 stati**. Il numero totale di eventi eseguiti per raggiungere il Max Coverage in questo caso è **181**, di cui come nel primo caso, **49** sono differenti. L'**efficienza** è leggermente migliorata rispetto al caso base ed

infatti è pari allo **0,24%**, inoltre il tempo di esecuzione è migliore, difatti la Test Suite viene eseguita in circa **51 minuti**.

Nel caso di AardDict possiamo dire che la riduzione per stati è la tecnica più efficiente e ci permette di eseguire il test in pochissimo tempo, ma risulta essere meno efficace delle altre, poiché ricopre soltanto il 33% del codice sorgente. La riduzione per eventi invece è migliore in termini di efficacia ed evidenzia che alcune transizioni sono ridondanti e inutili ai fini del calcolo del Max Coverage. Inoltre, questa riduzione risulta essere più efficiente rispetto al caso base in quanto permette di risparmiare circa 10 minuti. In definitiva per questa applicazione la tecnica di riduzione per stati è la più efficiente, mentre quella più equilibrata è quella della riduzione per eventi.

5.2 AlarmClock 1.7

AlarmClock [29] per Android è un'applicazione che consente di impostare e gestire un timer o una sveglia. Consente di memorizzare più sveglie o allarmi contemporaneamente e di impostare orari e/o giorni per la loro ripetizione.

Tabella 5.2: Valori test Sistematico – AlarmClock

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	876	244	66,56%	0,068%	19	251	295m
Riduzione per Stati	28	19	51,48%	1,84%	19	10	11m
Riduzione per Eventi	827	244	60,81%	0,073%	19	228	260m

Nella tabella 5.2 sono indicati i risultati dei test eseguiti. Con la tecnica base si ricopriva il **66,56%** del codice sorgente, esaminando **19 stati** ed eseguendo **876 eventi totali**, dei quali **244** sono realmente **diversi**, con un'**efficienza** pari allo **0,068%** circa e impiegando **295 minuti** per eseguire le **251 trace**. Sfruttando la tecnica di riduzione per stati abbiamo che la Test Suite JUnit è diventata molto più piccola, passando da 251 a soli **10 casi di test** e questo naturalmente ha enormemente influito sul tempo di esecuzione. Con questa Test Suite si è coperto poco più del **51%** del codice sorgente, ispezionando ancora **19 stati** ma eseguendo solo **28 eventi totali**, di cui **19 differenti**, con un'**efficienza** quindi pari all'**1,84%** circa, più grande di quella ottenuta precedentemente e impiegando solo **11 minuti**. Infine con la tecnica di riduzione per eventi è stato coperto il **60,81%** del codice sorgente. In questo caso sono stati controllati sempre **19 stati**, ma sono stati eseguiti **in totale 827 eventi**, di cui ancora **244 diversi**, con un'**efficienza** pari allo **0,073%**. Il test è durato circa **260 minuti** e anche in questo caso la Test Suite JUnit risulta essere ridotta, con i suoi **228 Test Cases**.

Per questa applicazione, la tecnica più efficiente è quella della riduzione per stati e ci

permette di risparmiare tantissimo sul tempo di esecuzione, ma è anche la meno efficace. Con la riduzione per eventi, si ottiene un'efficacia migliore rispetto alla riduzione per stati, ma comunque peggiore rispetto a quella ottenuta nel caso base. Invece l'efficienza è migliore rispetto al caso base ma peggiore rispetto alla riduzione per stati. Quindi per AlarmClock le due tecniche di riduzioni sono più efficienti, ma la tecnica più efficace rimane quella di base.

5.3 Android Level 1.9.4

Android Level [30] (o Bubble Level) è un'app che emula il funzionamento di una livella a bolla sul dispositivo mobile. La livella è uno strumento di misura usato per calcolare la pendenza di una superficie rispetto al piano orizzontale.

Tabella 5.3: Valori test Sistematico – Android Level

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	115	38	59,55%	0,52%	8	39	40m
Riduzione per Stati	19	7	56,44%	6,27%	8	3	3m
Riduzione per Eventi	100	38	58,55%	0,58%	8	32	34m

Nella tabella 5.3 sono riportati i valori dei parametri ottenuti dai test eseguiti. Nel caso base si copriva il **59,55%** del codice sorgente, eseguendo **in totale 115 eventi**, di cui **38** sono effettivamente **diversi**. Con un'**efficienza** dello **0,52%**, impiegando circa **40 minuti** per controllare gli **8 stati** ed eseguire i **39 Test Cases** individuati. Usando la tecnica di riduzione per stati abbiamo osservato che la Test Suite JUnit è notevolmente ridotta, infatti è costituita solo da **3 casi di test**. Con questa Test Suite si è coperto il **56,44%** del codice sorgente, ispezionando anche in questo caso **8 stati** ed eseguendo solo **19 eventi totali**, dei quali **7** sono **differenti,** con un'**efficienza** pari al **6,27%**, impiegando circa **3 minuti**. Questi risultati sono migliori, in efficienza, rispetto a quelli ottenuti con la tecnica base. Infine con la tecnica di riduzione per eventi si ha che la Test Suite copre il **58,55%** del codice sorgente, esaminando sempre gli **8 stati** ed eseguendo **100 eventi in totale** e come nel primo caso **38** di essi sono **differenti**; con un'**efficienza** dello **0,58%** circa e impiegando per lo più **34 minuti**.

Alla luce dei risultati mostrati, possiamo affermare che per il testing di questa applicazione la riduzione per stati è più efficiente e rapida delle altre, ma è anche meno efficace, anche

se non di molto. La riduzione per eventi è la soluzione più equilibrata in quanto è più efficace della riduzione per stati ma meno della tecnica base ed è meno efficiente della riduzione per stati ma più efficiente della tecnica di base. Mentre la tecnica di base rimane quella più efficace, ma questo è quello che in generale ci aspettavamo poiché con la riduzione andiamo a considerare meno eventi e transizioni.

5.4 Battery Circle 1.81

Battery Circle [31] è un'applicazione Android che permette di gestire il livello di carica della batteria del dispositivo mobile Android. Con questa app il livello di carica della batteria è indicato nella barra di stato e consente di visualizzare anche altri parametri legati alla batteria, come la temperatura e la durata; inoltre permette di gestire al meglio la carica rimasta.

Tabella 5.4: Valori test Sistematico – Battery Circle

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	68	26	91,24%	1,34%	5	26	27m
Riduzione per Stati	6	5	69,96%	11,66%	5	2	2m
Riduzione per Eventi	63	26	83,89%	1,33%	5	23	26m

Nella tabella 5.4 sono indicati i risultati ottenuti dai test fatti per questa applicazione. Con la tecnica base avevamo che si ricopriva il **91%** del codice sorgente, controllando **5 stati** ed eseguendo **68 eventi totali**, dei quali **26** sono effettivamente **differenti** tra loro. Con un'**efficienza** pari all'**1,34%** ed impiegando circa **27 minuti** per eseguire tutti i **26 casi di test**. Sfruttando la riduzione per stati abbiamo che la Test Suite JUnit è molto ridotta con soli **2 casi di test** e con essa si è coperto il **69%** del codice sorgente, analizzando ancora **5 stati**, ma eseguendo **in totale** solo **6 eventi**, di cui **5** sono **differenti**; con un'**efficienza** dell'**11,66%** e impiegando circa **2 minuti**. Infine con la tecnica di riduzione per eventi abbiamo ottenuto una Test Suite JUnit di **23 trace**. In questo caso è stato coperto quasi l'**84%** del codice sorgente, ispezionando sempre **5 stati** e svolgendo **63 eventi totali**, dei quali **26** sono realmente **differenti,** come nel primo caso. L'**efficienza** è stata pari all'**1,33%** e sono stati impiegati quasi **26 minuti** per eseguire il test.

Analizzando i risultati ottenuti, abbiamo che per questa app la tecnica meno utile è quella

della riduzione per eventi, perché ci dà risultati peggiori sia in termini di efficienza, che in termini di efficacia nonostante vengono considerati gli stessi stati, permettendo di risparmiare solo pochissimi minuti. La tecnica della riduzione per stati risulta essere la migliore da un punto di vista dell'efficienza e del tempo impiegato, ma è troppo poco efficace. Questo problema è legato alla strategia di riduzione adottata: infatti anche se vengono considerati gli stessi stati della tecnica base, si considerano meno eventi e questo ci porta a trascurare delle transizioni tra stati, facendoci ignorare che "*uno stesso stato può essere raggiunto anche attraverso sequenze diverse di eventi*". Per cui per Battery Circle la soluzione più efficace è quella della tecnica di base.

5.5 Marine Compass 1.2.4

Marine Compass [32] è un'applicazione Android, che consente di emulare, sul dispositivo Android, il funzionamento di una semplice bussola marina; inoltre anche muovendo il dispositivo la bussola rimarrà sempre parallela al pavimento.

Tabella 5.5: Valori test Sistematico – Marine Compass

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	10	6	88,57%	8,86%	2	6	6m
Riduzione per Stati	1	1	87,73%	87,73%	2	1	1m
Riduzione per Eventi	9	6	88,57%	9,84%	2	5	5m

Nella tabella 5.5 sono riportati i risultati dei test eseguiti per questa app. Partendo dalla tecnica base, avevamo che si ricopriva l'**88,57%** del codice sorgente, eseguendo **10 eventi in totale**, con **6** di questi **diversi**; con un'**efficienza** pari all'**8,86%**, impiegando **6 minuti** per controllare le **6 trace** ed esaminarne i **2 stati**. Usando la tecnica di riduzione per stati abbiamo ottenuto un risultato molto interessante: difatti in questo caso si è coperto l'**87,73%** del codice sorgente, valore molto vicino a quello ottenuto nel caso base; anche in questa circostanza vengono ispezionati **2 stati** ma si esegue **1 solo evento** per raggiungere il valore di massima copertura. L'**efficienza** è migliorata di molto, infatti è pari all'**87,73%** e il tempo di esecuzione del testing è notevolmente diminuito, difatti è stato quasi di **1 minuto**. E' interessante notare che la Test Suite JUnit in questo caso è composta da **1 sola trace**, pertanto è quel solo Test Case, o meglio è quell'unico evento, a coprire quasi tutto il codice sorgente. Infine adottando la tecnica di riduzione per eventi si ha che la Test Suite JUnit è ridotta di 1 sola trace. In questo caso è stata coperta la stessa percentuale di codice coperta nel caso base, cioè l'**88,57%**; ma per raggiungere tale valore sono stati eseguiti **in totale 9 eventi**, di cui ancora **6** sono **differenti**. L'**efficienza** è pari al

9,84% ed è migliore rispetto a quella ottenuta nel primo caso. Sono stati impiegati **5 minuti** per eseguire i **5 casi di test** e controllare i **2 stati** che sono rimasti sempre gli stessi. Per Marine Compass abbiamo che la tecnica migliore è quella della riduzione per eventi, poiché otteniamo un'efficacia simile a quella ottenuta con la tecnica di base, ma con un'efficienza migliore e poi il test viene fatto in minor tempo anche se vengono considerati gli stessi stati. D'altronde sono molto interessanti anche i risultati ottenuti con la tecnica di riduzione per stati, poiché abbiamo ottenuto un valore di copertura vicinissimo all'88% ma con un'efficienza maggiore rispetto a quella ottenuta con le altre due tecniche.

5.6 Notification Plus 1.1

Notification Plus [33] è un'applicazione che genera e invia notifiche, con cui richiama l'attenzione, sfruttando la vibrazione o la suoneria del telefono. E' molto utile nei dispositivi che non sono dotati di LED di notifica. Essa consente di gestire notifiche relative ai messaggi SMS, chiamate perse, o messaggi Gmail.

Tabella 5.6: Valori test Sistematico – Notification Plus

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	37	23	40,11%	1,08%	3	23	23m
Riduzione per Stati	2	2	34,69%	17,35%	3	2	2m
Riduzione per Eventi	35	23	40,11%	1,14%	3	21	21m

Nella tabella 5.6 sono riportati i valori dei parametri ricavati dai test eseguiti per questa app. Con la tecnica di base si ricopriva circa il **40,11%** del codice sorgente, ispezionando **3 stati** ed eseguendo **37 eventi totali,** dei quali **23** sono **differenti**; con un'**efficienza** quindi pari all'**1,08%**, impiegando circa **23 minuti** per eseguire i **23 casi di test**. Adottando la tecnica di riduzione per stati si è coperto circa il **34,69%** del codice sorgente, controllando anche questa volta **3 stati** ed eseguendo **in totale 2 eventi**, che sono anche differenti. L'**efficienza** è pari al **17,35%**, che è maggiore rispetto a quella ottenuta nel caso precedente. Il tempo di esecuzione è stato di circa **2 minuti**, necessario per eseguire i **2 casi di test**. Infine usando la tecnica di riduzione per eventi abbiamo che la Test Suite JUnit si riduce ed è composta da **21 casi di test**. Eseguendo questa Test Suite si è coperto il **40,11%** del codice sorgente, che è lo stesso valore di copertura raggiunto nel primo caso, esaminando sempre **3 stati** ed eseguendo **35 eventi totali**, di cui **23** sono **differenti**. L'**efficienza** è appena migliore di quella ottenuta nel caso base, infatti è pari all'**1,14%**, inoltre anche il tempo di esecuzione è migliore, poiché è stato di circa **21 minuti**.

Quindi confrontando le tre tecniche per Notification Plus, abbiamo che la tecnica di riduzione per stati è nettamente la più efficiente e rapida, ma risulta essere anche quella meno efficace. La soluzione migliore è data dalla tecnica di riduzione per eventi, poiché con essa otteniamo la stessa efficacia ottenuta con la tecnica base, ma con un'efficienza migliore, anche se di poco; inoltre vengono esaminati comunque gli stessi stati ma in un lasso di tempo inferiore. Anche per questa applicazione la riduzione per eventi evidenzia che alcuni eventi, considerati nel caso base, sono ridondanti e quindi eseguirli non influisce sul valore finale del Max Coverage.

5.7 Omnidroid 0.2.1

Omnidroid [34] è un'applicazione Android che consente agli utenti di automatizzare le funzionalità del sistema, in base a delle operazioni e/o intenti che si vogliono eseguire. Cioè consente di automatizzare il funzionamento del dispositivo in seguito al verificarsi di determinate condizioni e situazioni.

Tabella 5.7: Valori test Sistematico – Omnidroid

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	3739	589	58,01%	0,01%	97	624	905m
Riduzione per Stati	264	115	39,74%	0,15%	97	40	55m
Riduzione per Eventi	2986	589	47,96%	0,02%	97	493	660m

Nella tabella 5.7 sono indicati i risultati dei test eseguiti. Partendo dal caso base si copriva il **58,01%** del codice sorgente, ispezionando **97 stati** ed eseguendo **3739 eventi totali,** dei quali **589** sono **differenti**, con un'**efficienza** uguale allo **0,01%** circa; impiegando **905 minuti** per eseguire tutte le **624 trace**. Adottando la tecnica di riduzione per stati, la Test Suite JUnit è molto ridotta, siamo passati da 624 casi di test a **40**. Eseguendo questa Test Suite si è coperto quasi il **40%** del codice sorgente, valore inferiore a quello ricavato con la tecnica base, controllando comunque lo stesso numero di **stati** (**97**) ma eseguendo **264 eventi totali,** di cui **115 diversi**. Però come di solito è capitato anche negli altri test, con questa tecnica si ha un miglioramento dell'**efficienza** che è pari allo **0,15%** circa e del tempo di esecuzione che è stato solo di **55 minuti**. Anche con la tecnica di riduzione per eventi la Test Suite JUnit è abbastanza ridotta, infatti siamo passati da 624 Test Cases a **493** ed inoltre anche il valore di massima copertura è minore di quello ottenuto nel caso base, difatti è quasi del **48%**. Ma per coprire tale percentuale di codice sorgente sono stati eseguiti **2986 eventi**, di cui, come nel primo caso, **589** sono **diversi**. L'**efficienza** è

migliorata di pochissimo, infatti è pari allo **0,02%**. Sono stati ispezionati ancora **97 stati** e il test è durato circa **660 minuti**.

Per Omnidroid le due tecniche di riduzione risultano essere inefficaci, infatti in entrambi i casi il valore del Max Coverage raggiunto è inferiore a quello ottenuto con la tecnica di base, nonostante siano stati esaminati gli stessi stati. Ma queste due tecniche sono più efficienti di quella di base ed entrambe sono eseguite in minor tempo. E' importante sottolineare che per queste due tecniche abbiamo una percentuale di copertura inferiore poiché consideriamo meno eventi totali e quindi anche meno transizioni e meno situazioni in cui "*uno stato può essere raggiunto in diversi modi e attraverso diverse iterazioni e transizioni*". In conclusione quindi la tecnica più efficace per testare questa applicazione è la tecnica di base, quella più efficiente è quella della riduzione per stati.

5.8 Pedometer 1.4.0

Pedometer [35] è un'applicazione Android che consente di contare i passi che facciamo, calcola e visualizza la distanza percorsa, il ritmo, le calorie bruciate e la velocità, è un'app utile per il fitness. E' possibile inoltre impostare alcuni parametri come: la velocità, il ritmo e invia anche notifiche quando si deve andare più veloci o più lenti.

Tabella 5.8: Valori test Sistematico – Pedometer

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	148	48	65,54%	0,44%	9	48	51m
Riduzione per Stati	11	8	64,21%	5,84%	9	4	4m
Riduzione per Eventi	136	48	65,54%	0,48%	9	42	48m

Nella tabella 5.8 sono riportati i valori dei parametri ricavati dai test eseguiti. Nel caso della tecnica base si ricopriva circa il **65,54%** del codice sorgente, eseguendo **148 eventi totali**, dei quali **48** sono effettivamente **diversi**; con un'**efficienza** pari allo **0,44%** circa, controllando **9 stati** e impiegando circa **51 minuti** per eseguire un totale di **48 Test Cases**. Usando la tecnica di riduzione per stati abbiamo una riduzione estrema della Test Suite JUnit e del numero dei casi di test; infatti dalle 48 trace di partenza si passa a solo **4 casi di test**. Naturalmente questo farebbe pensare a una drastica riduzione anche del valore di copertura del codice sorgente, poiché si considerano meno eventi, ma in realtà non è così. Difatti in questo caso il valore del Max Coverage è stato del **64,21%** e il numero di eventi totali è stato pari a **11**, di cui **8** sono **differenti**. L'**efficienza** con cui è stato eseguito il test è maggiore, infatti è pari al **5,84%**. Si sono impiegati **4 minuti** per eseguire le **4 trace** totali, ispezionando sempre **9 stati**. Infine adottando la tecnica di riduzione per eventi abbiamo che anche in questo caso la Test Suite JUnit è stata ridotta passando da 48 a **42 casi di test** complessivi. Il valore di massima copertura raggiunto in questo caso è proprio

del **65,54%**, identico a quello raggiunto nel caso della tecnica di base. Sono stati eseguiti **136 eventi,** di cui, come nel primo caso **48** sono **differenti,** con un'**efficienza** pari allo **0,48%**, leggermente migliore a quella ottenuta nel caso base. Il test è durato circa **48 minuti**, necessari per eseguire i **42 casi di test** e controllare i **9 stati,** che non sono cambiati.

Nel caso di Pedometer possiamo dire che la riduzione è sicuramente un'ottima soluzione, in particolare con la riduzione per eventi otteniamo la stessa efficacia e una miglior efficienza per il testing, rispetto a quelle ottenute nel caso base; inoltre otteniamo tali risultati in minor tempo e eseguendo meno Test Cases, nonostante siano considerati lo stesso numero di stati; quindi tale tecnica è da preferirsi a quella di base. D'altro canto risultano essere molto interessanti anche i risultati ottenuti con la tecnica di riduzione per stati, poiché si ottiene un valore di Max Coverage vicinissimo al 65% con un'efficienza migliore, poiché si eseguono meno eventi e meno casi di test, ma si controllano gli stessi stati in un lasso di tempo decisamente inferiore. Quindi, anche per questa applicazione, abbiamo che la tecnica più efficiente in assoluto è quella della riduzione per stati, mentre quella più equilibrata è quella della riduzione per eventi.

5.9 Quick Setting 1.9.9.3

Quick Setting [36] è un'applicazione che consente di gestire rapidamente le impostazioni del sistema Android, quali la luminosità, il wifi, il bluetooth, il GPS, la torcia a LED, il mobile data etc. Il tutto con pochi e semplici touch.

Tabella 5.9: Valori test Sistematico – Quick Setting

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	869	239	39,67%	0,046%	21	254	299m
Riduzione per Stati	37	22	33,52%	0,906%	21	12	14m
Riduzione per Eventi	789	239	38,42%	0,049%	21	218	256m

Nella tabella 5.9 sono indicati i risultati dei test eseguiti. Con la Test Suite ottenuta usando la tecnica di base si ricopriva il **39,67%** del codice sorgente, ispezionando **21 stati**, eseguendo **869 eventi totali**, dei quali **239** sono effettivamente **diversi**; con un'**efficienza** pari allo **0,046%** e impiegando **299 minuti** per eseguire i **254 casi di test**. Con la tecnica di riduzione per stati la Test Suite JUnit è stata ampiamente ridotta, passando da 254 trace a solo **12 Test Cases**. Nonostante la forte riduzione, il valore di massima copertura non è diminuito di molto essendo del **33,52%**. In questo caso è stato ispezionato lo stesso numero di **stati** (**21**), ma eseguendo solo **37 eventi in totale**, di cui **22** sono **differenti**, con un'**efficienza** pari allo **0,906%** circa, migliore di quella del caso precedente e impiegando solo **14 minuti**. Infine con la tecnica di riduzione per eventi abbiamo ottenuto una Test Suite JUnit di **218 casi di test**. Con questa Test Suite è stato coperto il **38,42%** del codice sorgente, esaminando sempre **21 stati** ed eseguendo **789 eventi totali**, 80 in meno rispetto al caso base, di cui ancora **239 differenti**. L'**efficienza** è stata pari allo **0,049%** e il test è durato all'incirca **256 minuti**.

Per Quick Setting abbiamo che la soluzione più efficace è quella che usa la tecnica di base.

Ma le altre due soluzioni, che prevedono la riduzione, sono comunque molto interessanti, soprattutto se consideriamo il tempo di esecuzione e l'efficienza come fattori rilevanti per la nostra valutazione. Infatti è vero che in entrambi i casi il valore di massima copertura è inferiore, e questo è ancora una volta dovuto alla strategia di riduzione che abbiamo adottato, che ci porta a considerare meno eventi totali, rispetto al caso di partenza. Ma tali risultati sono ottenuti in meno tempo e con un'efficienza migliore, rispetto a quella ottenuta con la tecnica di base. In definitiva per questa applicazione la tecnica più efficiente è quella della riduzione per stati, quella più efficace è quella di base.

5.10 SimplyDo 0.9.2

L'applicazione SimplyDo [37] Android è un manager list, che consente appunto di gestire una lista di elementi. Affinché il test funzionasse è stato necessario creare all'interno dell'app, nell'AVD, un elemento (*Android Crawler*) e un sotto elemento (*Android Ripper*), poi abbiamo salvato tutto con una immagine di *Snapshot* dell'emulatore.

Tabella 5.10: Valori test Sistematico – SimplyDo

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	707	187	78,54%	0,11%	31	187	240m
Riduzione per Stati	51	32	58,91%	1,15%	31	13	18m
Riduzione per Eventi	619	187	78,25%	0,13%	31	157	214m

Nella tabella 5.10 sono rappresentati i risultati dei test eseguiti. Nel caso base si ricopriva circa il **78,54%** del codice sorgente, eseguendo **707 eventi in totale**, di cui **187** sono **diversi**; ottenendo così un'**efficienza** pari allo **0,11%**. In questo caso sono stati esaminati **31 stati** e sono stati eseguiti **187 casi di test** in **240 minuti**. Adottando la tecnica di riduzione per stati abbiamo ottenuto una consistente riduzione della Test Suite JUnit; infatti da 187 casi di test siamo passati a solo **13**. Anche in questo caso sono stati controllati **31 stati** totali, ma si è coperto il **58,91%** del codice sorgente, anche perché sono stati eseguiti meno **eventi totali** (**51**), di cui **32** sono **differenti**. L'**efficienza** è pari all'**1,15%**, più elevata di quella ottenuta nel caso precedente ed anche in termini di tempo si è avuto un notevole risparmio, infatti il testing è durato circa **18 minuti**. Infine con la tecnica di riduzione per eventi abbiamo ricavato una Test Suite JUnit di **157 trace**, 30 in meno rispetto al primo caso. Con questa Test Suite viene coperto il **78,25%** del codice sorgente, eseguendo **619 eventi totali,** dei quali, come nel caso base, **187** sono **differenti**, ispezionando sempre **31 stati**. L'**efficienza** è migliorata leggermente, infatti è pari allo

0,13% e il tempo di esecuzione è nettamente diminuito, difatti è pari a **214 minuti**, necessario per eseguire i **157 casi di test**.

Anche nel caso di SimplyDo possiamo dire che la riduzione è un'ottima soluzione per ottimizzare il testing. In particolare adottando la tecnica di riduzione per eventi, otteniamo per lo più la stessa efficacia e un piccolo miglioramento dell'efficienza, dovuto al fatto che consideriamo meno eventi totali; ma questo sottolinea anche come alcuni eventi che erano eseguiti nel caso base fossero inutili, poiché ridondanti, per calcolare il Max Coverage. Inoltre anche il tempo di esecuzione è decisamente inferiore, permettendo di risparmiare quasi 30 minuti, che è importantissimo; riuscendo comunque a controllare lo stesso numero di stati e di eventi differenti. Per quanto riguarda la tecnica di riduzione per stati è nettamente più efficiente e rapida delle altre due, ma è anche quella meno efficace, poiché è vero che vengono considerati gli stessi stati, ma non lo stesso numero di eventi.

5.11 Tic Tac Toe 1.0

Tic Tac Toe [38] è un'applicazione Android che permette di giocare al gioco del tris su un dispositivo Android gratuitamente. Questa app supporta sia la possibilità di giocare in due e sia quella di giocare contro il dispositivo stesso, quindi contro un avversario virtuale. La modalità contro l'intelligenza artificiale comprende tre livelli di difficoltà.

Tabella 5.11: Valori test Sistematico – Tic Tac Toe

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	168	68	78,49%	0,47%	10	68	74m
Riduzione per Stati	11	10	32,05%	2,91%	10	4	5m
Riduzione per Eventi	155	68	78,49%	0,51%	10	60	69m

Nella tabella 5.11 sono riportati i risultati dei test eseguiti per questa applicazione. Con la Test Suite della tecnica di base si ricopriva il **78,49%** circa del codice sorgente, analizzando **10 stati** e compiendo **168 eventi totali**, dei quali solo **68** sono **diversi**; ottenendo un'**efficienza** dello **0,47%** e impiegando più o meno **74 minuti**, per eseguire i **68 casi di test**. Sfruttando la tecnica di riduzione per stati, abbiamo che la Test Suite JUnit è molto ridotta, si passa da 68 a **4 trace**. Eseguendo questa Test Suite si è esaminato sempre lo stesso numero di **stati** (**10**), però si copre solo il **32,05%** del codice sorgente, svolgendo **11 eventi totali**, di cui **10** sono **diversi**; con un'**efficienza** pari al **2,91%** e impiegando quasi **5 minuti**. Infine usando la tecnica di riduzione per eventi, abbiamo ricavato una Test Suite JUnit di **60 casi di test**. In questo caso viene coperto il **78,49%** del codice sorgente, cioè lo stesso valore ottenuto con la tecnica di base, ispezionando ancora gli stessi **10 stati** e svolgendo **in totale 155 eventi,** dei quali **68** sono **diversi**. Rispetto al primo caso l'**efficienza** è un po' migliorata, infatti è pari allo **0,51%**, inoltre anche il tempo di esecuzione è migliore, difatti è stato di circa **69 minuti**.

Anche per questa applicazione la tecnica migliore per eseguire il testing è quella della riduzione per eventi. Infatti con essa si copre la stessa quantità massima di codice sorgente coperta nel caso base e sono esaminati gli stessi stati, ma tale risultato è ottenuto con un'efficienza migliore, in meno tempo ed eseguendo meno eventi; sottolineando ancora una volta che alcuni eventi sono ripetitivi e quindi inutili per il calcolo del Max Coverage. La riduzione per stati invece è la soluzione più efficiente e rapida, ma anche per questa app è quella meno efficace. Forse in questo caso lo è fin troppo, perché la minimizzazione è stata troppo eccessiva e ha portato a considerare troppi pochi eventi rispetto al caso base.

5.12 Tippy Tipper 1.2

Tippy Tipper [39] è un'applicazione che consente di utilizzare una calcolatrice "Tip", che permette di calcolare e valutare i parametri di una fattura. Inoltre permette di inserire una fattura personalizzata tramite tastiera, di definire una soglia massima per i valori da calcolare, di escludere l'aliquota fiscale etc.

Anche per questa applicazione abbiamo ricavato che la tecnica migliore, per eseguire il testing è quella della riduzione per eventi; ma andiamo per ordine.

Tabella 5.12: Valori test Sistematico – Tippy Tipper

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	220	81	75,33%	0,34%	10	81	95m
Riduzione per Stati	14	9	52,55%	3,75%	10	5	5m
Riduzione per Eventi	201	81	75,33%	0,37%	10	72	75m

Nella tabella 5.12 sono indicati i valori dei parametri ottenuti dall'esecuzione dei test. Con la tecnica di base si copriva il **75,33%** del codice sorgente, analizzando **10 stati** ed eseguendo **in totale 220 eventi,** dei quali **81** sono effettivamente **diversi**, con un'**efficienza** pari allo **0,34%**, impiegando circa **95 minuti** per eseguire gli **81 casi di test totali**. Utilizzando la tecnica di riduzione per stati la Test Suite JUnit è stata ridotta a soli **5 Test Cases**. Con questa Test Suite si è coperto il **52,55%** del codice sorgente, considerando ancora **10 stati** ed eseguendo solo **14 eventi totali**, di cui **9** sono **diversi**, con un'**efficienza** pari al **3,75%**, impiegando circa **5 minuti**. Infine anche usando la tecnica di riduzione per eventi abbiamo che la Test Suite JUnit è stata ridotta, siamo passati da 81 a **72 casi di test**. In questo caso è stato coperto il **75,33%** del codice sorgente, valore simile a quello ottenuto con la tecnica di base, controllando ancora **10 stati** ed eseguendo **in totale 201 eventi**, di cui **81 differenti** come nel primo caso. L'efficienza e il tempo di

esecuzione sono leggermente migliorati, infatti sono rispettivamente uguali allo **0,37%** ed a **75 minuti**.

Anche per Tippy Tipper risulta che l'utilizzo delle tecniche di riduzione è vantaggioso. In particolare la soluzione migliore, da tutti i punti di vista, è data dalla tecnica di riduzione per eventi, infatti abbiamo che l'efficacia è la stessa ricavata usando la tecnica base e in più anche gli stati analizzati sono gli stessi, ma l'efficienza è leggermente migliore e poi abbiamo risparmiato circa 20 minuti per eseguire il test. Sono interessanti anche i valori ottenuti adottando la tecnica di riduzione per stati. Con questa tecnica è vero che il valore di copertura raggiunto è minore, nonostante abbiamo considerato gli stessi stati, ma in compenso abbiamo un'efficienza maggiore rispetto alle altre due tecniche e inoltre il tempo impiegato per eseguire il testing è enormemente inferiore, parametro da non sottovalutare, spesso decisivo nella scelta della tecnica da usare.

5.13 Tomdroid 0.7.1

Tomdroid [40] è un'applicazione che permette di leggere e prendere appunti sfruttando il software TomBoy [41]. In particolare si è fatto in modo che tale applicazione fosse compatibile con il formato dei file e fosse in grado di sincronizzarsi con le note create con TomBoy.

Tabella 5.13: Valori test Sistematico – Tomdroid

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	875	182	36,19%	0,041%	27	184	235m
Riduzione per Stati	57	29	31,02%	0,544%	27	10	14m
Riduzione per Eventi	773	182	35,87%	0,046%	27	157	216m

Nella tabella 5.13 sono riportati i risultati dei test eseguiti. Con la Test Suite della tecnica di base è stato coperto il **36,19%** circa del codice sorgente e sono stati eseguiti **875 eventi totali**, di cui **182** sono **diversi**. L'**efficienza** è pari allo **0,041%** e il tempo di esecuzione del testing è stato di circa **235 minuti**, necessario per controllare tutti i **27 stati** ed eseguire i **184 casi di test**. Adottando la tecnica di riduzione per stati abbiamo ottenuto dei buoni risultati, considerando lo sforzo per ricavarli. Infatti in questo caso è stato coperto il **31,02%** del codice sorgente, ispezionando ancora **27 stati**, ma eseguendo solo **57 eventi totali**, di cui **29** sono **differenti**. Come si può notare la riduzione è stata notevole, difatti la Test Suite JUnit è molto ridotta, da 184 trace iniziali si è passati a solo **10 Test Cases**. L'**efficienza** è pari allo **0,544%** circa, ed è maggiore rispetto al caso precedente. Il test è durato circa **14 minuti**. Infine usando la tecnica di riduzione per eventi abbiamo ricavato che la Test Suite JUnit è ancora ridotta rispetto a quella di base, difatti si passa da 184 a **157 casi di test**. Con questa Test Suite è stato coperto il **35,87%** del codice sorgente, eseguendo **773 eventi totali,** di cui **182 differenti,** come nel primo caso. L'**efficienza** è

leggermente migliore rispetto al caso base, infatti è pari allo **0,046%**. Sono stati controllati ancora **27 stati** e il tempo per eseguire il test è stato di **216 minuti**, cioè un tempo inferiore di circa 20 minuti rispetto a quello ottenuto con la tecnica di base.

Nel caso di Tomdroid abbiamo che in termini di efficacia, la soluzione migliore è quella di base che permette di coprire il 36,19% del codice sorgente. Ma se consideriamo, come fattori di valutazione, anche l'efficienza e il tempo impiegato per eseguire il testing, allora diventano interessanti anche i risultati ottenuti con le tecniche di riduzione. Infatti con la tecnica di riduzione per stati è vero che la percentuale di copertura è solo del 31,02% ma l'efficienza è migliorata e il guadagno di tempo, per eseguire il test, è enorme. Però in quest'ottica la soluzione più equilibrata sembra essere quella ottenuta con la tecnica di riduzione per eventi, poiché, rispetto alla tecnica di base, si ha un guadagno sia in efficienza sia nel tempo di esecuzione del testing, con una percentuale di copertura solo lievemente inferiore.

5.14 Trolly 1.4

Trolly [42] è un'applicazione Android che consente di creare e gestire una lista della spesa, consentendo di aggiungere e/o eliminare elementi dalla lista; inoltre l'app ha dei filtri che permettono ad altre app di aggiungere automaticamente elementi alla lista.

Tabella 5.14: Valori test Sistematico – Trolly

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
Base	154	57	61,35%	0,39%	9	58	62m
Riduzione per Stati	10	8	40,36%	4,04%	9	4	4m
Riduzione per Eventi	139	57	59,15%	0,42%	9	50	55m

Nella tabella 5.14 sono indicati i risultati che abbiamo ricavato dai test eseguiti per questa app. Nel caso della tecnica base si ricopriva il **61,35%** del codice sorgente, controllando **9 stati** ed eseguendo **in totale 154 eventi,** dei quali solo **57** sono **diversi**; con un'**efficienza** dello **0,39%** e impiegando circa **62 minuti** per svolgere tutti i **58 casi di test**. Usando la tecnica di riduzione per stati abbiamo che la Test Suite JUnit è notevolmente ridotta, infatti è formata da solo **4 trace**. Eseguendo questa Test Suite si è coperto solo il **40%** del codice sorgente, considerando ancora **9 stati** ed eseguendo **10 eventi in totale**, di cui **8** sono effettivamente **differenti**; con un'**efficienza** pari al **4,04%**, superiore a quella ottenuta nel primo caso ed impiegando pochissimo tempo, circa **4 minuti**. Infine anche con la tecnica di riduzione per eventi la Test Suite JUnit è ridotta, da 58 casi di test passiamo a **50**. In questo caso sono controllati gli stessi **9 stati** della tecnica base e viene coperto il **59%** del codice sorgente, eseguendo **139 eventi totali**, con **57** di essi **differenti**. L'**efficienza** è pari allo **0,42%**, che è poco superiore a quella ottenuta con la tecnica di base e il tempo di esecuzione è stato di circa **55 minuti**.

Nel caso di Trolly le due tecniche di riduzione dal punto di vista dell'efficacia non sembrano essere particolarmente utili, infatti abbiamo che entrambe coprono una percentuale di codice sorgente inferiore a quella ottenuta nel caso base. Ma d'altronde questo era quello che in generale ci aspettavamo, poiché consideriamo meno eventi in totale. Quindi per questa applicazione la tecnica più efficace è quella di base. Però le due tecniche di riduzione sono più efficienti della prima tecnica: in particolare abbiamo che la tecnica più efficiente è quella della riduzione per stati, che è anche nettamente più rapida; ma è importante sottolineare che è pure quella meno efficace.

5.15 Valutazioni sulle medie e confronto tra le tre tecniche

Tabella 5.14: Valori test Sistematico – Confronto tra i valori medi

Tecnica	Total Events	Different Events	Max Coverage	Efficiency	# of Traces	Time
Base	586	131	55,18%	0,006%	136	172m
Riduzione per Stati	38	20	42,13%	0,08%	8	10,4m
Riduzione per Eventi	501	131	51,26%	0,007%	114	142m

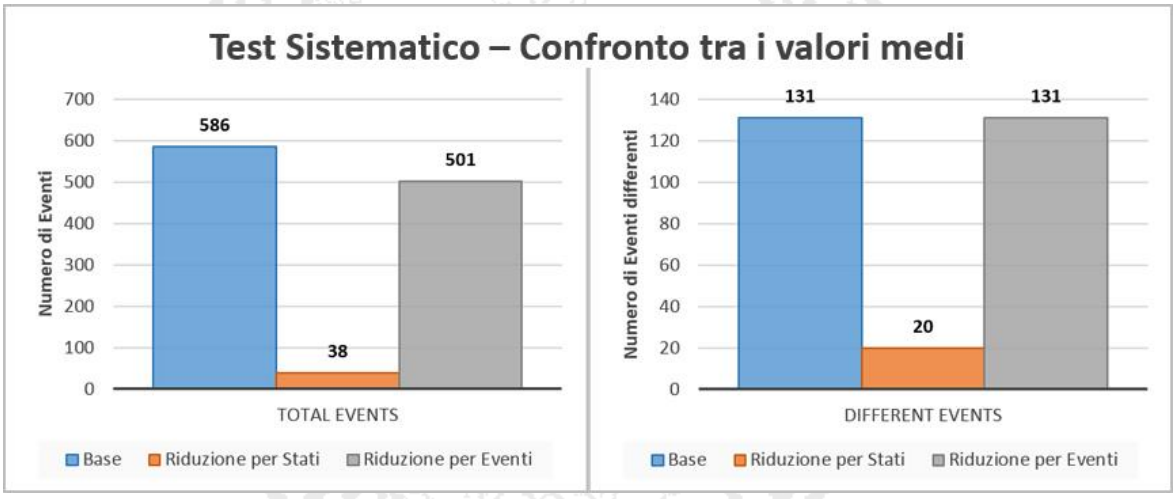


Figura 5.1: Istogramma Valori test Sistematico – Confronto tra i valori medi

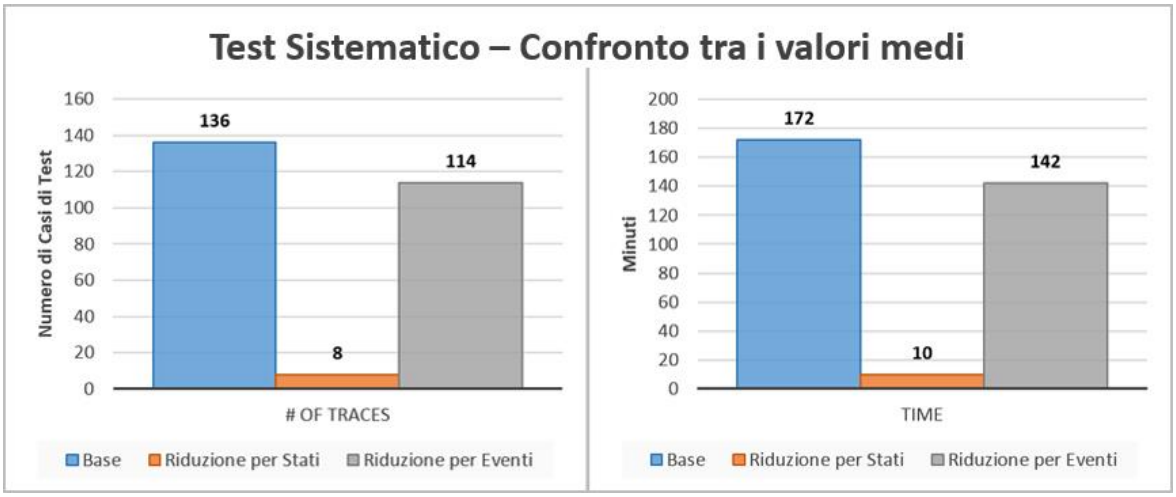


Figura 5.2: Istogramma Valori test Sistematico – Confronto tra i valori medi

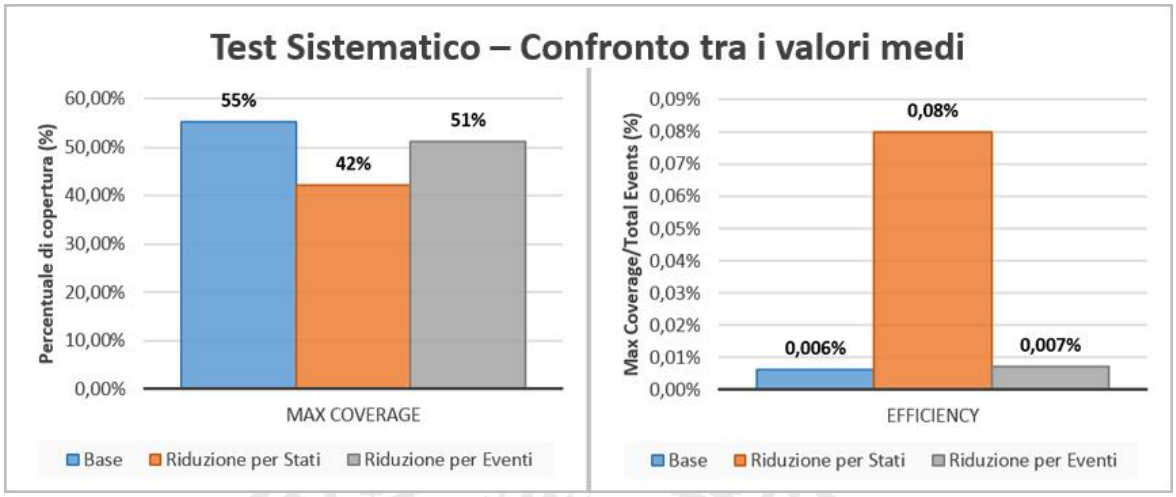


Figura 5.3: Istogramma Valori test Sistematico – Confronto tra i valori medi

Nella tabella 5.16 e nei tre istogrammi in figura 5.1, 5.2 e 5.3, sono indicati i valori che abbiamo ottenuto in media, per ogni parametro e per ogni tecnica/test eseguito. Considerando questi valori risulta che la tecnica più efficiente tra le tre è quella della riduzione per stati, perché ogni evento copre in media lo **0,08%** del codice sorgente, inoltre è anche la tecnica più veloce, infatti il tempo di esecuzione in media è pari a **10 minuti**. Però tale tecnica è anche quella meno efficace, poiché ci ha permesso di coprire in media solo il **42%** del codice sorgente, nonostante siano stati considerati gli stessi stati. Ma questo risultato è legato proprio alla strategia di riduzione per stati, a causa della quale abbiamo considerato in media meno casi di test, meno eventi totali e quindi meno eventi

differenti. La tecnica più efficace rimane quella di base, infatti con essa si è coperto mediamente il **55%** del codice sorgente, ma questo è normale, era ciò che ci aspettavamo, dato che in questo caso sono considerati in media più casi di test, più eventi totali e più eventi differenti. Però proprio per questo motivo tale tecnica è quella meno efficiente tra le tre, infatti ogni evento copre in media lo **0,006%** del codice sorgente e questo valore è più basso rispetto a quelli ottenuti negli altri due casi. Inoltre è anche la tecnica più lenta, difatti il tempo di esecuzione in media è pari a **172 minuti**. La tecnica di riduzione per eventi è quella più equilibrata. Infatti con questa tecnica si copre mediamente il **51%** del codice sorgente, che è maggiore di quello coperto in media con la tecnica di riduzione per stati, ma è leggermente minore di quello coperto con la tecnica di base. Poi si eseguono mediamente più eventi totali, più eventi diversi e più casi di test rispetto alla tecnica di riduzione per stati, ma meno Test Cases ed eventi rispetto alla tecnica di base. Inoltre il tempo di esecuzione è stato mediamente di **142 minuti**, che è maggiore di quello ottenuto in media con la tecnica di riduzione per stati, ma minore di quello ottenuto con la tecnica di base. Infine dai risultati ricaviamo che la riduzione per eventi è più efficiente della tecnica di base, ma meno efficiente di quella della riduzione per stati, infatti ogni evento copre in media lo **0,007%** del codice sorgente.

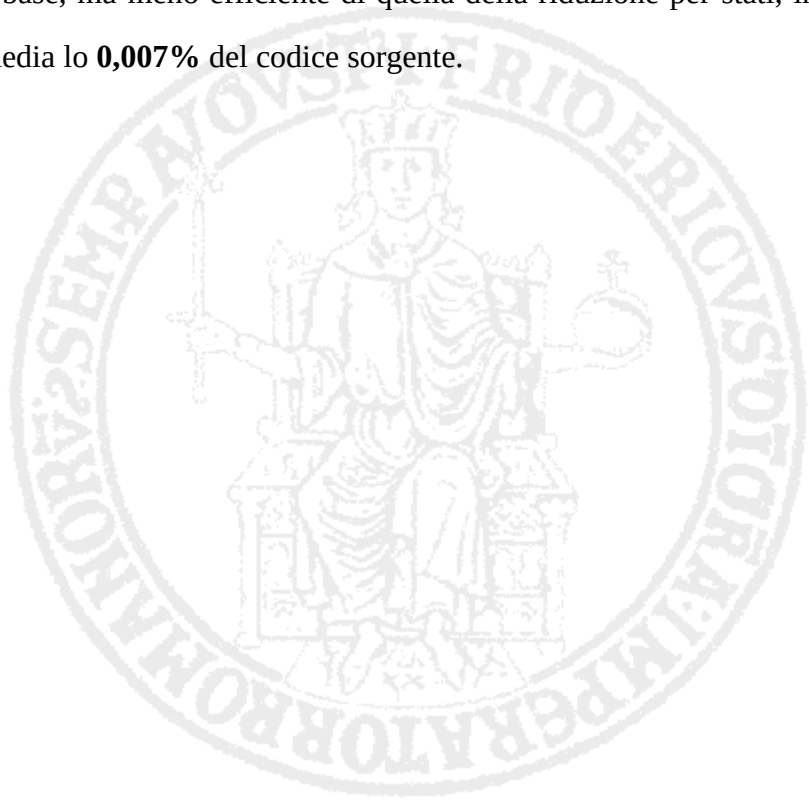


Tabella 5.43: Media dei valori test Sistematico – Tecnica Base

App	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
AardDict 1.4.1	217	49	44,25%	0,2%	13	49	63m
AlarmClock 1.7	876	244	66,56%	0,068%	19	251	295m
Android Level 1.9.4	115	38	59,55%	0,52%	8	39	40m
Battery-Circle 1.81	68	26	91,24%	1,34%	5	26	27m
Marine-Compass 1.2.4	10	6	88,57%	8,86%	2	6	6m
Notification Plus 1.1	37	23	40,11%	1,08%	3	23	23m
Omnidroid 0.2.1	3739	589	58,01%	0,01%	97	624	905m
Pedometer 1.4.1	148	48	65,54%	0,44%	9	48	51m
Quick-Setting 1.9.9.3	869	239	39,67%	0,046%	21	254	299m
SimplyDo 0.9.2	707	187	78,54%	0,11%	31	187	240m
Tic Tac Toe 1.0	168	68	78,49%	0,47%	10	68	74m
Tippy-Tipper 1.2	220	81	75,33%	0,34%	10	81	95m
Tomdroid 0.7.1	875	182	36,19%	0,041%	27	184	235m
Trolly 1.4	154	57	61,35%	0,39%	9	58	62m
AVERAGE	585,92	131,21	55,18%	0,006%	18,86	135,6	172m

Tabella 5.44: Media dei valori test Sistematico – Tecnica Riduzione per Stati

App	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
AardDict 1.4.1	25	12	32,72%	1,3%	13	6	8m
AlarmClock 1.7	28	19	51,48%	1,84%	19	10	11m
Android Level 1.9.4	19	7	56,44%	6,27%	8	3	3m
Battery-Circle 1.81	6	5	69,96%	11,66%	5	2	2m
Marine-Compass 1.2.4	1	1	87,73%	87,73%	2	1	1m
Notification Plus 1.1	2	2	34,69%	17,35%	3	2	2m
Omnidroid 0.2.1	264	115	39,74%	0,15%	97	40	55m
Pedometer 1.4.1	11	8	64,21%	5,84%	9	4	4m
Quick-Setting 1.9.9.3	37	22	33,52%	0,906%	21	12	14m
SimplyDo 0.9.2	51	32	58,91%	1,15%	31	13	18m
Tic Tac Toe 1.0	11	10	32,05%	2,91%	10	4	5m
Tippy-Tipper 1.2	14	9	52,55%	3,75%	10	5	5m
Tomdroid 0.7.1	57	29	31,02%	0,544%	27	10	14m
Trolly 1.4	10	8	40,36%	4,04%	9	4	4m
AVERAGE	37,57	19,92	42,13%	0,08%	18,86	8,3	10,4m

Tabella 5.45: Media dei valori test Sistematico – Tecnica Riduzione per Eventi

App	Total Events	Different Events	Max Coverage	Efficiency	# of States	# of Traces	Time
AardDict 1.4.1	181	49	44,22%	0,24%	13	39	51m
AlarmClock 1.7	827	244	60,81%	0,073%	19	228	260m
Android Level 1.9.4	100	38	58,55%	0,58%	8	32	34m
Battery-Circle 1.81	63	26	83,89%	1,33%	5	23	26m
Marine-Compass 1.2.4	9	6	88,57%	9,84%	2	5	5m
Notification Plus 1.1	35	23	40,11%	1,14%	3	21	21m
Omnidroid 0.2.1	2986	589	47,96%	0,02%	97	493	660m
Pedometer 1.4.1	136	48	65,54%	0,48%	9	42	48m
Quick-Setting 1.9.9.3	789	239	38,42%	0,049%	21	218	256m
SimplyDo 0.9.2	619	187	78,25%	0,13%	31	157	214m
Tic Tac Toe 1.0	155	68	78,49%	0,51%	10	60	69m
Tippy-Tipper 1.2	201	81	75,33%	0,37%	10	72	75m
Tomdroid 0.7.1	773	182	35,87%	0,046%	27	157	216m
Trolly 1.4	139	57	59,15%	0,42%	9	50	55m
AVERAGE	500,93	131,21	51,26%	0,007%	18,86	114,1	142m

Capitolo 6

Conclusioni e Sviluppi Futuri

Attraverso i nostri studi, abbiamo essenzialmente introdotto due tecniche di riduzione, la **Riduzione per Stati** e la **Riduzione per Eventi**, con l'obiettivo di usarle per ottenere delle Test Suite ridotte a partire da una Test Suite completa; cercando di ottimizzare ulteriormente il testing delle applicazioni Android. Per ottenere questo risultato, ho sviluppato il **Reduction_Tool**, che permette di ricavare due file guitree ridotti, a partire dai quali abbiamo ricavato le Test Suite JUnit ridotte, che poi abbiamo usato per eseguire i test per ogni applicazione e per valutare la bontà di queste riduzioni. Eseguendo e analizzando i test, abbiamo potuto osservare come l'**efficacia** e l'**efficienza** dipendono fortemente dalla strategia di minimizzazione adottata. Con le tecniche di riduzione ci aspettavamo di ottenere dei test meno efficaci ma più efficienti e questo è proprio quello che si è verificato; solo nel caso di *Battery Circle* si è verificato che la tecnica di base è più efficiente di quella della riduzione per eventi, ma comunque meno efficiente di quella della riduzione per stati.

Per confrontare in modo più specifico le due tecniche di riduzione con la tecnica di base, per capire se ci possiamo permettere queste tecniche di riduzione, abbiamo infine calcolato le differenze tra le medie per ciascun parametro che abbiamo considerato importante per i nostri obiettivi.

Tabella 6.1: Test Sistematico – Differenze tra le medie

Difference between Averages	Total Events	Different Events	Max Coverage	Efficiency	# of Traces	Time
Δ(OR-RED.ST) %	94%	85%	13%	-0,074%	94%	94%
Δ(OR-RED.EV) %	15%	0	4%	-0,001%	16%	18%

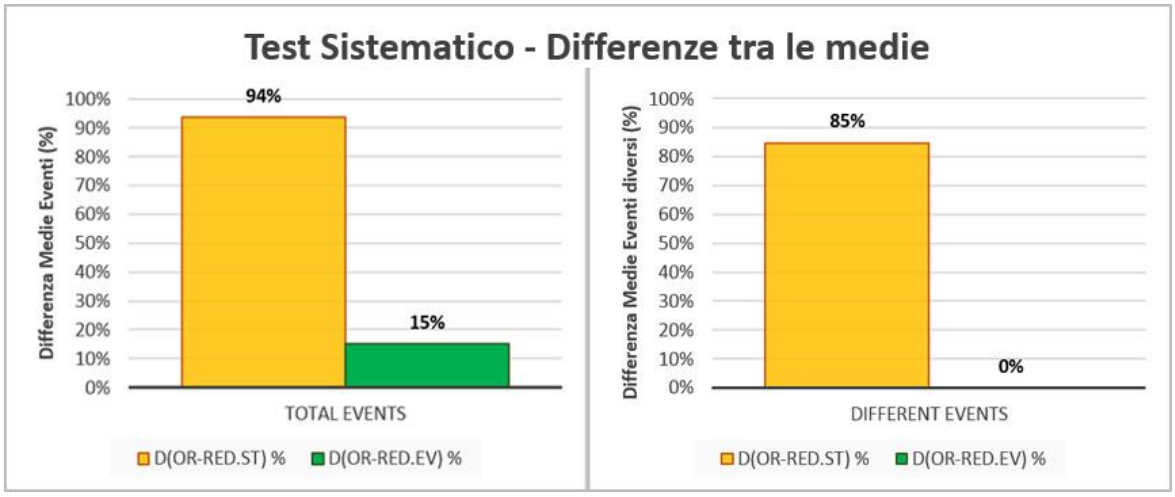


Figura 6.1: Istogramma Test Sistematico – Differenze tra le medie

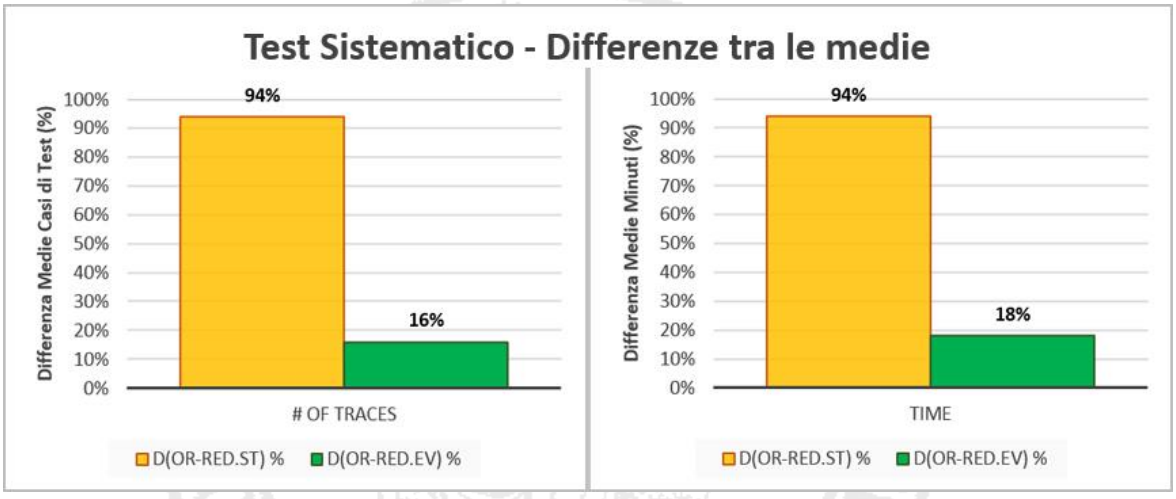
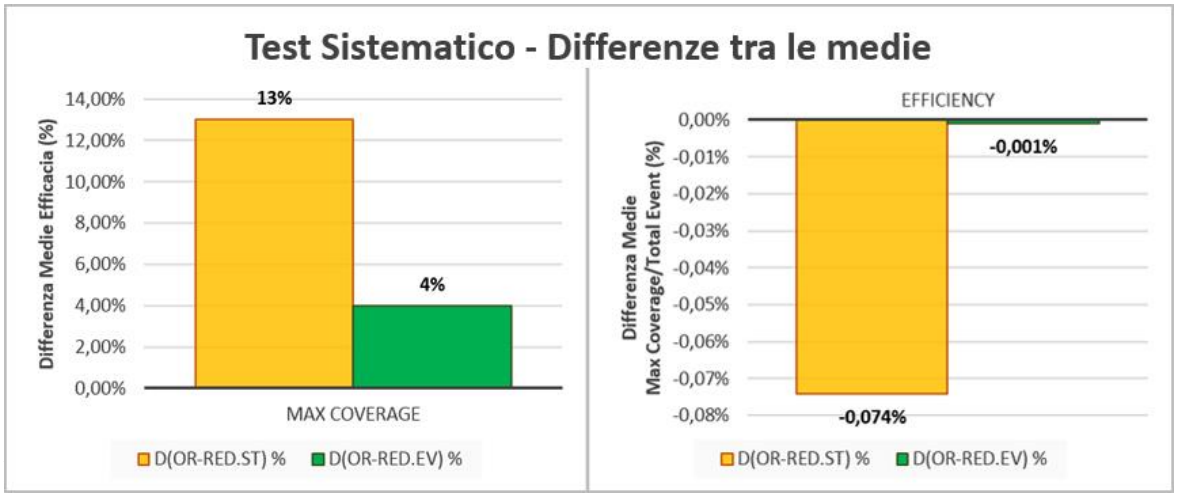


Figura 6.2: Istogramma Test Sistematico – Differenze tra le medie



igura 6.3: Istogramma Test Sistematico – Differenze tra le medie

Nella tabella 6.1 e negli istogrammi in figura 6.1, 6.2 e 6.3, sono indicati proprio i valori di queste differenze. Confrontando la tecnica di base con la tecnica di riduzione per stati si ricava che con la tecnica di base si copre mediamente il **13%** in più del codice sorgente, si eseguono in media il **94%** in più di eventi totali, l'**85%** in più di eventi differenti ed il **94%** in più di casi di test. Però la tecnica di riduzione per stati è mediamente più efficiente, poiché ogni evento copre in media lo **0,074%** in più del codice sorgente ed inoltre è anche più veloce, infatti il test con la tecnica di base è durato in media il **94%** in più rispetto a quello con la tecnica di riduzione per stati.

Quindi la tecnica di base è più efficace, quella della riduzione per stati è più efficiente.

Confrontando la tecnica di base con la tecnica di riduzione per eventi si ricava che la tecnica di base copre mediamente il **4%** in più del codice sorgente ed esegue in media il **15%** in più di eventi totali ed il **16%** in più di Test Cases. Invece il numero di eventi differenti che sono mediamente considerati nelle due tecniche sono gli stessi. Però la tecnica di riduzione per eventi è mediamente più efficiente, poiché ogni evento copre in media lo **0,001%** in più di codice sorgente rispetto alla tecnica di base ed inoltre è più veloce, infatti i test eseguiti con la tecnica di base sono durati mediamente il **18%** in più rispetto a quelli fatti con la tecnica di riduzione per eventi.

Quindi anche in questo caso la tecnica più efficace è quella di base, quella della riduzione per eventi è la più efficiente. Però con la tecnica di riduzione per eventi abbiamo ottenuto,

per alcune applicazioni come *AardDict*, *Marine Compass*, *Notification Plus*, *Pedometer*, *SimplyDo*, *Tic Tac Toe*, e *Tippy Tipper* dei risultati interessanti, poiché si ricopriva la stessa percentuale di codice sorgente coperta con la tecnica di base, ma eseguendo meno eventi totali, in minor tempo e quindi con un'efficienza migliore. Questo ha evidenziato che, per queste app, alcuni eventi sono inutili ai fini del calcolo del Max Coverage. In questi casi la tecnica di riduzione per eventi è da preferirsi a quella di base, poiché più efficiente a parità di efficacia.

Per migliorare queste tecniche di minimizzazione si potrebbero particolarizzare quelle che abbiamo visto. L'idea sarebbe quella di fare in modo che l'algoritmo di riduzione:

1. Nel definire **il sottoinsieme minimo di trace che copre il massimo numero di stati**, vada ad effettuare la riduzione, non considerando gli stati e gli eventi in maniera indipendente, ma considerando che "*uno stesso stato può essere raggiunto in diversi modi, attraverso diverse sequenze di eventi, diverse transizioni e/o iterazioni*".
2. Nel definire **il sottoinsieme minimo di trace che copre il massimo numero di eventi**, vada ad effettuare la riduzione considerando le relazioni tra gli eventi; cioè nel nostro caso abbiamo ridotto la Test Suite considerando che "*se una trace aveva gli stessi eventi di una seconda trace, allora la seconda trace viene eliminata dalla Test Suite*", senza considerare che, è vero che i due casi di test contengono gli stessi eventi, ma questi eventi potrebbero essere eseguiti con ordine diverso e quindi appartenere a sequenze diverse e portare a situazioni differenti.

Pertanto considerando questi criteri, si potrebbe sviluppare una tecnica di riduzione per sequenze, in cui invece di considerare gli stati e gli eventi, considero i sottostati di sequenze di eventi.

Oppure si potrebbero usare altre strategie per la riduzione, definendo nuove tecniche di minimizzazione. Ad esempio si potrebbe introdurre una tecnica di riduzione basata sulla copertura delle righe del codice sorgente; sicuramente in questo modo si potrebbero ottenere soluzioni migliori.

Ringraziamenti

Desidero ringraziare, tutte le persone che mi hanno sostenuto e che mi hanno aiutato a raggiungere questo importante traguardo della mia vita:

- il Prof. Porfirio Tramontana, che mi ha dato tantissimi consigli, mi ha spiegato moltissime cose, mi ha ascoltato e aiutato sia durante le fasi di sviluppo del tool, sia durante le fasi del testing…
- l'Ing. Nicola Amatucci che mi ha spiegato molte cose e chiarito i miei dubbi;
- il mio amico e Ing. Gennaro Imparato, con cui ho discusso di moltissimi argomenti, soprattutto legati al testing e che mi ha sempre incoraggiato;
- le persone che ho incontrato nel Lab 4.04;
- infine, ma non per ultima, la mia famiglia, i miei genitori, che mi son sempre stati vicini in questi anni e mi hanno sempre incoraggiato e sostenuto;

E ringrazio di cuore anche tutte le persone che hanno condiviso con me questi anni di Università…

Bibliografia

[1] Apple, *L'App Store di Apple raggiunge lo storico traguardo dei 50 miliardi di download*, 2013, http://www.apple.com/it/pr/library/2013/05/16Apples-App-Store-Marks-Historic-50-Billionth-Download.html.

[2] Andrea Bai, *Spartphone: Android e iOS dominano il mercato, ma il sistema operativo di Google batte la Mela*, 2014, http://www.businessmagazine.it/news/smartphone-android-e-ios-dominano-il-mercato-ma-il-sistema-operativo-di-google-batte-la-mela_50994.html.

[3] Felice Pescatore, *Storia di Android (Android History)*, 2013, http://www.storiainformatica.it/android.

[4] *Android Developers*, http://developer.android.com/index.html.

[5] *Activities*, http://developer.android.com/guide/components/activities.html.

[6] *Services*, http://developer.android.com/guide/components/services.html.

[7] *Broadcast Receiver*, http://developer.android.com/reference/android/content/BroadcastReceiver.html.

[8] *Content Provider*, http://developer.android.com/guide/topics/providers/content-providers.html.

[9] *Intent*, http://developer.android.com/reference/android/content/Intent.html.

[10] *Processes and Threads*, http://developer.android.com/guide/components/processes-and-threads.html.

[11] *SDK (Software Development Kit)*, https://developer.android.com/sdk/index.html?hl=i.

[12] *App Resources (Software Development Kit)*, http://developer.android.com/guide/topics/resources/index.html.

[13] *Layouts*, http://developer.android.com/guide/topics/ui/declaring-layout.html.

[14] *User Interface*, http://developer.android.com/guide/topics/ui/index.html.

[15] *Widgets*, http://developer.android.com/design/patterns/widgets.html;

[16] *Android Emulator* http://developer.android.com/tools/help/emulator.html.

[17] *App Manifest*, http://developer.android.com/guide/topics/manifest/manifest-intro.html.

[18] Marko Gargenta, *Learning Android*, O'Reilly, 2011.

[19] Chris Crum, *Updating Android Market to Google Play*, 2012, http://www.webpronews.com/updating-android-market-to-google-play-2012-03.

[20] *Testing Foundamentals*, http://developer.android.com/tools/testing/testing_android.html.

[21] *Robotium*, https://code.google.com/p/robotium/.

[22] *GUI-Testing*, http://www.appperfect.com/products/application-testing/app-test-gui-testing.html.

[23] C. Enrique Ortiz, *Managing the MIDIet Life-Cicle with a Finite State Machine*, 2004, http://www.oracle.com/technetwork/systems/fsm-156381.html.

[24] Atif Memon, *An event flow model of gui-based applications for testing*, Wiley InterScience, 2007.

[25] Domenico Amalfitano, Salvatore De Carmine, Anna Rita Fasolino, Porfirio Tramontana e Atif Memon, *Using GUI Ripping for Automated Testing of Android Apps*, 2012, http://www.slideshare.net/PorfirioTramontana/using-gui-ripping-for-automated-testing-of-android-apps.

[26] *EMMA: a free Java code coverage tool*, http://emma.sourceforge.net/.

[27] Fadini, Esposito, *Teoria e Progetto delle Reti Logiche*, Liguori Ed, seconda edizione, 1994.

[28] *Aard Dictionary*, http://aarddict.org/.

[29] *AlarmClock*, http://developer.android.com/reference/android/provider/AlarmClock.html.

[30] *Android Level*, https://play.google.com/store/apps/details?id=net.androgames.level.

[31] *Battery Circle*, https://play.google.com/store/apps/details?id=ch.blinkenlights.battery&hl=it.

[32] *Marine Compass*, http://www.pierrox.net/cmsms/open-source/marine-compass-2.html.

[33] *Notification Plus*, https://code.google.com/p/notification-plus/.

[34] *Omnidroid*, https://code.google.com/p/omnidroid/.

[35] *Pedometer*, https://code.google.com/p/pedometer/.

[36] *Quick Setting*, http://quick-settings.android.informer.com/1.9.9.3/.

[37] *SimplyDo*, https://code.google.com/p/simply-do/.

[38] *Tic Tac Toe*,

https://play.google.com/store/apps/details?id=com.optimesoftware.tictactoe.free.

[39] *Tippy Tipper*, https://code.google.com/p/tippytipper/.

[40] *Tomdroid*, https://launchpad.net/tomdroid.

[41] *TomBoy*, https://wiki.gnome.org/Apps/Tomboy.

[42] *Trolly*, https://code.google.com/p/trolly/.

.