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

Tesi di Laurea Magistrale in Ingegneria Informatica

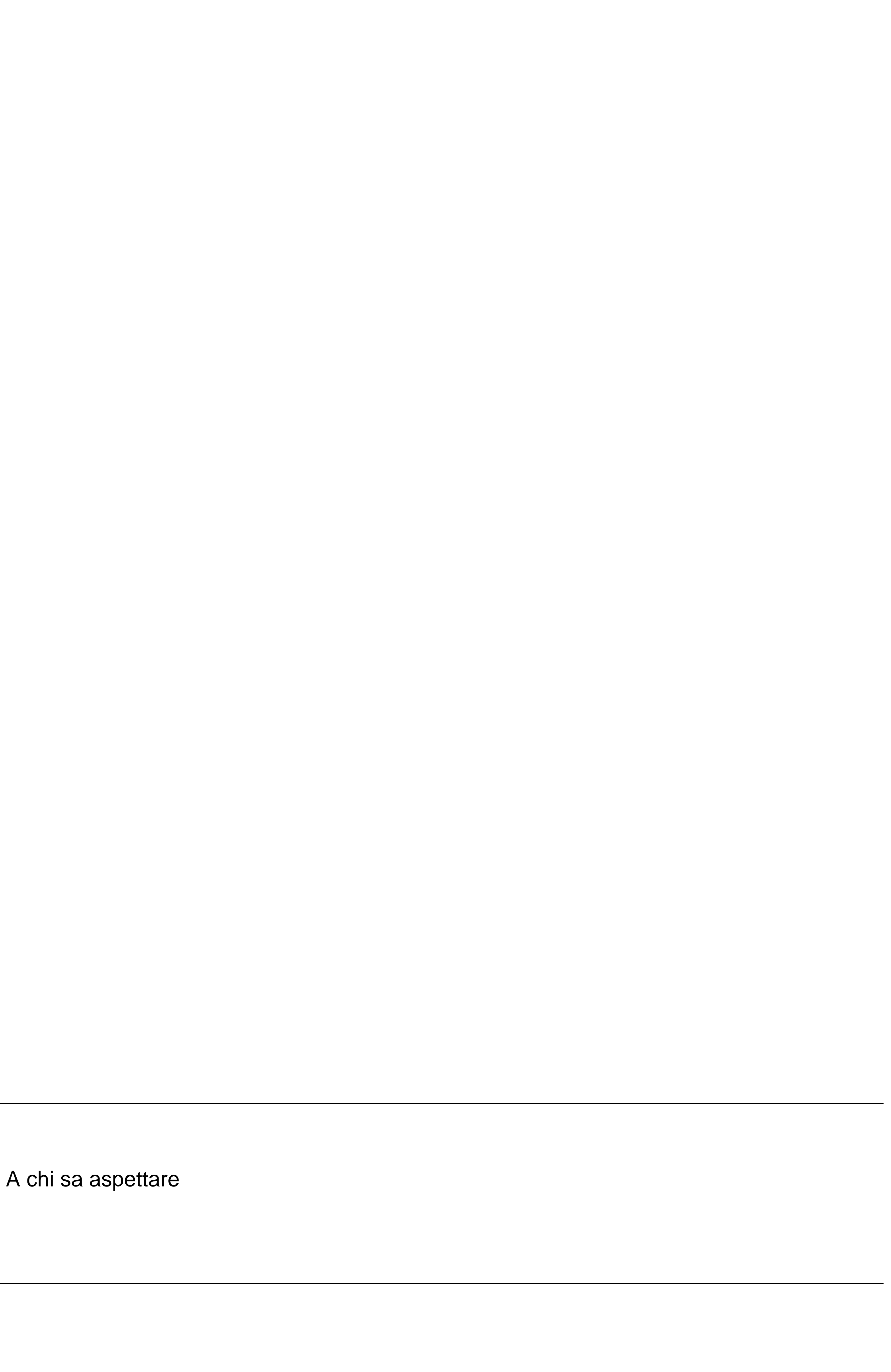
Una tecnica ibrida manuale/sistematica per la generazione di casi di test per applicazioni Android

Anno Accademico 2014/2015

relatore
Ch.mo Prof Porfirio Tramontana

correlatore
Ing. Nicola Amatucci

candidato
Francesco Giovanni Sanges
matr. M63000309



A chi sa aspettare

Indice

Indice III
Introduzione 5
Capitolo 1: La piattaforma Android 8
1.1 Cosa è Android 8
1.1.1 Dalvik Virtual Machine (DVM) e Android Runtime (ART) 9
1.1.2 Releases 10
1.1.3 L'ambiente di sviluppo 12
1.2 L'architettura 14
1.2.1 Linux Kernel 14
1.2.2 Hardware Abstraction Layer 15
1.2.3 Libraries 15
1.2.4 Android Runtime 16
1.2.5 Application Framework 16
1.2.6 Applications 17
1.3 Le app 17
1.3.1 I pacchetti APK 18
Capitolo 2: Testing 19
2.1 Attività di verifica e convalida 19
2.2 Test 20
2.2.1 Test Automation 21
2.2.2 Generazione dei test automatici basati su sessioni utente 22
2.2.3 Testing dell'interfaccia utente 23
2.3 Testing in Android 24
2.3.1 Robotium e Robotium Recorder 26
2.3.2 Monkey e MonkeyRunner 27
2.3.3 Android Ripper 27
2.3.4 Emma 27
Capitolo 3: Android Ripper 28
3.1 GUI testing di applicazioni Android 28
3.2 Ripper 29
3.2.1 Input e Output 30
3.2.2 Architettura 31
3.2.2 Funzionamento 33
Capitolo 4: Android Ripper "Ibrido" 40
4.1 Perché una tecnica ibrida 40
4.1.1 Componente sistematica 42
4.1.2 Componente manuale 42
4.2 La soluzione proposta: Ripper Ibrido 43

4.2.1 Architettura ... 43
4.2.2 Processo di funzionamento ... 44
4.3 Ambiente di sviluppo ... 47
4.4 Input ... 48
4.4.1 Robotium Recorder ... 48
4.5 Tool di conversione della classe di test ... 50
4.6 Funzionamento ... 54
4.6.1 Messaggi ... 62
4.7 Output ... 68
4.7.1 Output fase 1 ... 68
4.7.2 Output fase 2 ... 70
Capitolo 5: Efficacia Ripper Ibrido ... 73
5.1 Confronto Ripper Ibrido vs Ripper Originale ... 73
5.2 Applicazioni selezionate per il test ... 74
5.3 Risultati ... 75
5.3.1 SimplyDo ... 76
5.3.2 TippyTipper ... 78
5.3.3 Trolly ... 80
5.3.4 MunchLife ... 82
5.3.5 FillUp ... 84
Conclusioni e Sviluppi Futuri ... 86
Appendice A1: Guida all'installazione e uso del Ripper Ibrido ... 87
A1.1 Prerequisiti ... 87
A1.2 Android Ripper Installer ... 89
A1.3 Android Ripper Driver ... 89
A1.3.1 Fase 1 ... 90
A1.3.2 Fase 2 ... 90
Appendice A2: Esempio d'uso del Ripper Ibrido ... 91
A2.1 Prerequisiti ... 91
A2.2 Registrazione con Robotium Recorder ... 93
A2.3 Android Ripper Installer ... 97
A2.4 Android Ripper Driver ... 99
Bibliografia ... 100

Introduzione

Il panorama attuale delle piattaforme di tipo mobile vede principalmente la presenza di tre grandi competitors, ciascuno dei quali con un proprio sistema: Apple con iOS, Microsoft con Windows Phone e Google con Android.

Ogni piattaforma ha le proprie peculiarità che ne hanno decretato il successo e l'acquisizione di quote di mercato a vantaggio delle altre.

La naturale evoluzione della tecnologia dei terminali ha consentito di avere sempre più funzionalità con una user experience notevolmente migliorata. Si pensi ad esempio all'aggiunta di nuovi sensori, all'incremento delle capacità computazionali e di storage, alla capacità di eseguire più applicazioni contemporaneamente, all'incremento di reattività di risposta dei moderni dispositivi.

Se da un lato il progresso hardware ha consentito questo sviluppo, d'altra parte il sistema operativo, "cuore" dei dispositivi, deve essere in grado di sfruttare questo progresso. Un nuovo sensore o l'aumento di potenza elaborativa è praticamente inutile se il software non è in grado di utilizzarlo.

Non solo il sistema operativo, ma l'intero stack di strumenti software deve essere in grado di supportare al meglio la crescita e lo sviluppo della piattaforma. Si pensi alla possibilità di rendere "semplice" la creazione di applicazioni mobili garantita dal Software Development Kit (SDK), ossia l'insieme di strumenti messi a disposizione agli

sviluppatori, all'accesso a un market store di applicazioni fornitissimo, alla possibilità di accedere a tutte le funzionalità del sistema senza vincoli.

Android in particolare ha fatto dell'open source la sua filosofia, in netto contrasto con la politica "chiusa" di Apple, ed è questo uno dei motivi per la sua rapida diffusione e adozione da parte di moltissimi vendors.

Naturalmente anche altre caratteristiche hanno incrementato la sua crescita; le principali saranno analizzate nel Capitolo 1 di questo lavoro di tesi.

In particolare verranno presentati i punti cardine della piattaforma, l'architettura, la struttura delle applicazioni.

Una panoramica del mondo Android è necessaria per l'introduzione alle tipologie di testing che è possibile effettuare sulle applicazioni mobili, presentate nel Capitolo 2.

In generale l'attività di verifica e validazione punta a mostrare che il software è conforme alle sue specifiche e soddisfa le aspettative del cliente. Si parla di testing quando l'approccio alla verifica è di tipo sperimentale, attraverso l'osservazione dinamica del prodotto, del suo comportamento in esecuzione. In particolare ci concentreremo sull'interfaccia grafica (GUI: Graphical User Interface) delle applicazioni e verranno presentati ed esaminati gli strumenti utilizzati in Android per perseguire tale scopo.

La trattazione lascia spazio, nel Capitolo 3, al software Android Ripper, sviluppato all'Università di Napoli, che consente di esplorare automaticamente la GUI di un'applicazione mobile in maniera casuale e sistematica rilevando i crash dovuti ad eccezioni non gestite, di misurare la copertura del codice sorgente, sollevando lo sviluppatore dagli oneri richiesti a un'attività dispendiosa in termini di costi e tempi quale è l'attività di testing.

Tale software è stato preso come base di partenza per la creazione di un Ripper "ibrido", capace cioè di funzionare in prima istanza in maniera automatica e in seconda istanza di eseguire test case utente registrati con lo strumento Robotium Recorder congiuntamente alle capacità di esplorazione del Ripper originale. I test case utente sono così utilizzati in modo da giungere in "stati" dell'applicazione potenzialmente non esplorati durante la

prima fase. Per arrivare a tale traguardo il Ripper originale è stato profondamente modificato e corredato di strumenti per la corretta esecuzione e analisi dei risultati.

Lo sviluppo di tale software e la sua modalità di funzionamento sono oggetto del Capitolo 4, mentre i benefici ottenuti rispetto al software originale sono oggetto del Capitolo 5.

Capitolo 1: La piattaforma Android

In questo capitolo verranno presentati gli elementi costitutivi della piattaforma Android, le release del sistema, l'ambiente di sviluppo, gli elementi costituenti l'architettura e infine i blocchi costitutivi delle applicazioni mobili (le app).

1.1 Cosa è Android

Android è una piattaforma software che include un sistema operativo di base, un insieme di strumenti e di librerie che consente la realizzazione di applicazioni mobili.
E' caratterizzato dalla struttura open source, il suo stesso codice è ora disponibile sotto licenza Apache 2.0, quindi liberamente estendibile e modificabile e dall'adozione del kernel Linux (versione 2.6), progressivamente aggiornato nel corso delle release.
La piattaforma fornisce un SDK in grado di facilitare lo sviluppo delle applicazioni; il linguaggio principalmente utilizzato per la loro creazione è Java. E' comunque possibile ricorrere all'Android Native Development ovvero un insieme di strumenti che consente di implementare parti dell'applicazione usando codice nativo come C e C++.
Java era ed è un linguaggio affermato e che conta una nutrita schiera di sviluppatori quindi la sua adozione ha contribuito al successo della piattaforma. La scelta di Java ha però un risvolto in contrasto con quella che è la natura open di Android. I dispositivi che intendono adottare la Virtual Machine (VM) devono pagare una royalty.
La strategia adottata è stata quella di usare una propria VM, la Dalvik Virtual Machine (DVM). Si è attinto a piene mani dalle librerie standard di Java (versione di riferimento J2SE) ad esclusione delle componenti riguardanti la grafica come le Abstract Window

Toolkit (AWT) e le Swing. La definizione dell'interfaccia grafica è infatti un aspetto fondamentale nell'architettura di Android [1].

1.1.1 Dalvik Virtual Machine (DVM) e Android Runtime (ART)

La DVM è una VM ottimizzata per l'esecuzione di applicazioni in ambienti con risorse limitate. Non esegue bytecode standard, ma un altro linguaggio chiamato DEX (Dalvik EXecutable) in grado di ottimizzare lo spazio occupato attraverso una gestione della struttura delle informazioni che evita parti ridondanti. I file ".dex" sono ottenuti a partire da file ".class" di bytecode Java dati dalla compilazione del codice.

La DVM gestisce i thread con alcune limitazioni, non gestisce le eccezioni e ha un'architettura a registri; permette inoltre una efficace esecuzione di più processi in contemporanea [1]. Dalla versione di Android 2.2 include un compilatore Just In Time per migliorare le prestazione della macchina virtuale.

Dalla versione 5.0 (Lollipop), la DVM è ufficialmente sostituita da ART.

Il nuovo runtime system software è stato introdotto, in realtà, in modalità preview nella versione 4.4 KitKat. La virtual machine Dalvik è basata su tecnologia JIT (just-in-time): ogni app quindi viene compilata solo in parte dallo sviluppatore e sarà poi di volta in volta compito della DVM eseguire il codice e compilarlo definitivamente in linguaggio macchina in tempo reale, per ogni esecuzione dell'app stessa. Questo ovviamente incide sulle prestazioni anche se permette una maggiore versatilità nello sviluppo di applicazione che girano su più piattaforme.

ART invece è basata su tecnologia AOT (ahead-of-time) che esegue l'intera compilazione del codice durante l'installazione dell'app e non durante l'esecuzione stessa del software. Il vantaggio è in termini di tempo in fase di esecuzione al costo di un incremento temporale in fase di installazione.

Sono stati inoltre migliorati i report diagnostici relativi a eccezioni e crash e la gestione del garbage collector (GC).

1.1.2 Releases

Android nasce nel 2003 da una startup californiana, Android Inc., acquisita nel 2005 da Google. L'evoluzione della piattaforma è tuttavia curata da un consorzio denominato Open Handset Alliance (http://www.openhandsetalliance.com), nato nel 2007. Del gruppo, oltre a Google, fanno parte numerose società, tra cui Intel, Motorola, Samsung, Sony-Ericsson, Texas Instruments, LG e molti altri. La licenza scelta dalla Open Handset Alliance è la Open Source Apache License 2.0, che permette ai diversi vendor di costruire su Android le proprie estensioni anche proprietarie senza legami che ne potrebbero limitare l'utilizzo.

Dal 2008 gli aggiornamenti di Android per migliorarne le prestazioni e per eliminare i bug delle precedenti versioni sono stati molti. Ogni aggiornamento o release, segue un ordine alfabetico e una precisa convenzione per i nomi [1].

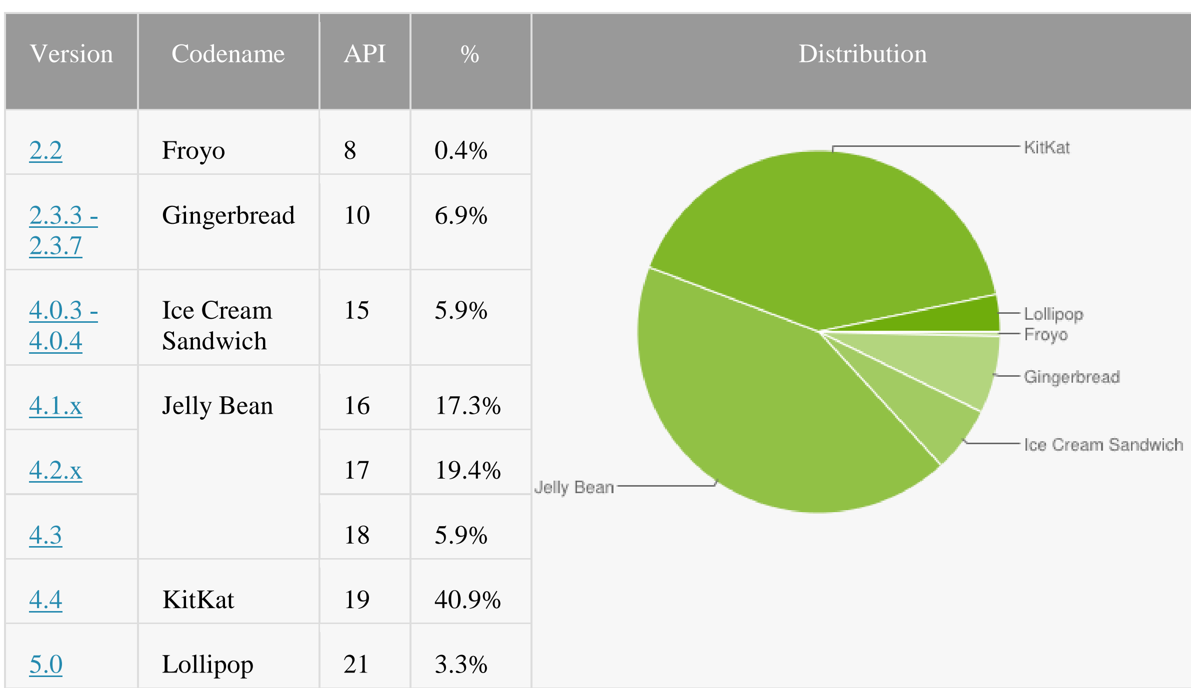
Di seguito una breve cronistoria delle versioni succedutesi, corredata delle principali novità introdotte e il livello di API (Application Program Interface). Il numero di versione cambia sia perché le API sono modificate sia per correzione di bug. Il livello di API determinerà quali dispositivi potranno eseguire l'applicazione.

1.0 – 1.1 API 1 - 2	Nel 2007 la prima versione del Software Development Kit (SDK), che ha permesso agli sviluppatori di esplorare la nuova piattaforma. Nel 2008 è stato rilasciato il sorgente in open source. La versione 1.0 è stata affinata fino al rilascio della versione 1.1. Erano già presenti la finestra di notifica a scomparsa, il supporto per i widget nella home sebbene non creabili dagli sviluppatori, uno scarno Market Store.
1.5 Cupcake API 3	Febbraio 2009. La versione 1.5 dell'SDK porta numerose novità come l'introduzione della tastiera virtuale, la possibilità di aggiungere widget utente alla home del dispositivo, miglioramenti alla clipboard, una rudimentale interfaccia della fotocamera.
1.6 Donut API 4	Settembre 2009. Aggiunta di ricerca vocale e testuale per i contenuti presenti in locale e sul Web. Introdotta la sintesi vocale e le gesture.

2.0 a 2.1 Eclair API 5 a 7	Ottobre 2009. Aggiunte numerose funzionalità per la fotocamera, il supporto al multi-touch e ai live wallpaper. Miglioramento delle prestazioni.
2.2 Froyo API 8	Maggio 2010. Miglioramento delle prestazioni dovuto ad una migliore gestione delle risorse hardware (compilazione just in time). Tethering USB e Wi-Fi. Miglioramento del browser di sistema. Supporto alla tecnologia Adobe Flash. Migliorie apportate a gran parte delle applicazioni di sistema e della sicurezza.
2.3 Gingerbread API 9 - 10	Dicembre 2010. User Interface aggiornata. Aggiunto il supporto agli schermi XL. Supporto nativo al SIP VoIP e alla tecnologia NFC. Tastiera riprogettata. Aggiunta l'app Download Manager, per la gestione unificata di tutti i download. Supporto nativo a sensori come giroscopio e barometro. Migliorata la gestione energetica.
3.0 a 3.2 Honeycomb API 11 a 13	Febbraio 2011. Versione ottimizzata per tablet. Introdotta nuova interfaccia utente ("Holo"). Aggiunta la barra di sistema (pulsanti software Home, Indietro, Task manager) e la Action Bar (fornisce accesso ad opzioni che variano in base al contesto). Accelerazione hardware e supporto per processori multi-core. Possibilità di criptare tutti i dati personali. Ampliato il supporto hardware.
4.0 Ice Cream Sandwich API 14 - 15	Ottobre 2011. Interfaccia utente completamente riprogettata: prestazioni migliorate, pulsanti virtuali al posto di quelli hardware (per i dispositivi che ne sono privi), launcher personalizzabile, nuovo font di sistema (Roboto). Aggiornate tutte le app di sistema per sfruttare le nuove API. Fotocamera migliorata. App "Contatti" integrata con i social network. Android Beam (scambio di dati tramite NFC). Wi-Fi Direct.
4.1 a 4.3 Jelly Bean API 16 a 18	Luglio 2012. Riconoscimento del tocco migliorato, ottimizzato l'utilizzo della CPU, migliorate la digitazione del testo, il riconoscimento e la sintesi vocale, ottimizzata l’applicazione fotocamera, gesture avanzate

	per le notifiche, nuovo servizio Google Now, Google Play Store aggiornato, abbandono al supporto di Adobe Flash.
4.4 KitKat API 19	Ottobre 2013. Rinnovata l'interfaccia grafica e introdotto il full screen completo. Supporto per nuovi tipi di sensore. Migliorata efficienza energetica. Introduzione di ART, attivabile dalle opzioni sviluppatore. Ottimizzato il funzionamento del sistema sui dispositivi con poca RAM.
5.0 – 5.1 Lollipop API 21 - 22	Novembre 2014. Interfaccia utente rinnovata (Material Design). Nuovo multitasking. Miglioramenti gestione audio, prestazioni grafiche e sicurezza. Eliminazione della runtime Dalvik in favore di ART. Supporto nativo ai 64 bit. Aggiunto il multi-utente e la modalità ospite.

La diffusione delle versioni aggiornata a Marzo 2015:



Version	Codename	API	%
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	6.9%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	5.9%
4.1.x	Jelly Bean	16	17.3%
4.2.x		17	19.4%
4.3		18	5.9%
4.4	KitKat	19	40.9%
5.0	Lollipop	21	3.3%

Immagine 1.1: Diffusione versioni Android

1.1.3 L'ambiente di sviluppo

Android Software Development Kit è il contenitore di tutti gli strumenti fondamentali per lo creazione di applicazioni Android in Java. Comprende: strumenti a supporto dello

sviluppo, emulatore, documentazione, esempi, strumenti di utilità.

La composizione non è immutabile ma viene gestita tramite l'SDK Manager. Grazie al Manager, il programmatore potrà profilare le piattaforme e gli strumenti presenti nel SDK nella maniera più congeniale al proprio lavoro.

L'AVD (Android Virtual Device) Manager consente invece la gestione dei device virtuali (Virtual Device, VD) su cui effettuare i propri esperimenti. E' possibile lavorare direttamente su un dispositivo reale connesso tramite USB alla macchina su cui risiede l'SDK.

Tra i tool forniti a corredo citiamo [5]:

- "android" (development tool) che gestisce gli AVD, i progetti e i componenti installati con l'SDK
- il tool di debug "adb" (Android Debug Bridge) per la comunicazione con l'emulatore o il terminale fisico. Fornisce inoltre l'accesso alla shell del device per l'esecuzione di comandi avanzati
- "emulator", un emulatore che permette di testare l'applicazione in modo sufficientemente accurato prima del deploy sul dispositivo reale
- "logcat" (platform tool) per collezionare e esaminare l'output per il debug
- il DDMS (Dalvik Debug Monitor Server) monitora il comportamento della macchina virtuale e consente di emulare eventi come ad esempio la ricezione di chiamate, sms o di variare le coordinate GPS

Per favorire lo sviluppo, l'SDK costituisce un insieme di strumenti stand-alone, lasciando allo sviluppatore la possibilità di integrarlo con i più diffusi IDE (Integrated Development Environment) in circolazione.

A Maggio 2013 è stato inoltre presentato da parte di Google un IDE, Android Studio, basato su IntelliJ IDEA, pensato appositamente per questo ambiente e che ingloba quindi l'SDK. Le sue principali caratteristiche sono l'utilizzo di Gradle come strumento di build automation, atto quindi ad accompagnare lo sviluppatore nelle fasi di build, sviluppo, test e pubblicazione della propria app; la disponibilità di un gran numero di template per la

realizzazione di applicazioni già in linea con i più comuni pattern progettuali; un editor grafico per la realizzazione di layout dotato di uno strumento di anteprima in grado di mostrare l'aspetto finale dell'interfaccia che si sta realizzando in una molteplicità di configurazioni (tablet e smartphone di vario tipo).

Sito di riferimento http://developer.android.com/sdk/index.html

1.2 L'architettura

Android comprende tutto lo stack degli strumenti per la creazione di applicazioni mobili, tra cui un sistema operativo, un insieme di librerie native per le funzionalità core della piattaforma, una implementazione della VM e un insieme di librerie Java. Si tratta di una architettura a layer, benché questi si compenetrano, dove i livelli inferiori offrono servizi ai livelli superiori offrendo un più alto grado di astrazione [1].

L'immagine illustra la suddivisione tra i layer e le componenti di ognuno di essi:

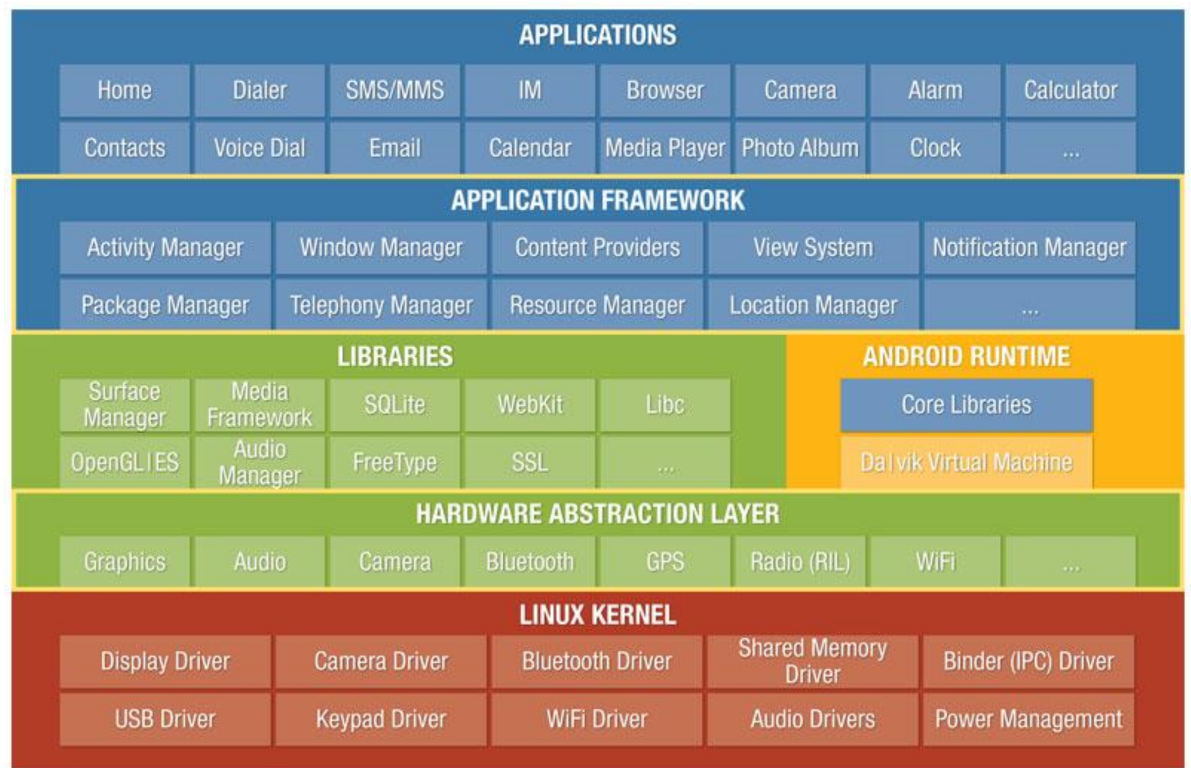


Immagine 1.2: Architettura di Android

1.2.1 Linux Kernel

Il layer di più basso livello è rappresentato dal kernel Linux nella versione 2.6. Garantisce gli strumenti di basso livello per l'astrazione dell'hardware sottostante attraverso la

definizione di diversi driver ad esempio per la gestione del display, della connessione Wi-Fi, del Bluetooth.

E' da notare anche la presenza di un driver dedicato alla gestione della comunicazione tra processi diversi (IPC); la sua importanza è fondamentale per far comunicare componenti diversi in un ambiente in cui ciascuna applicazione viene eseguita all'interno di un proprio processo.

Fornisce, oltre all'astrazione hardware, gestione della memoria, impostazioni di sicurezza, gestione del consumo.

1.2.2 Hardware Abstraction Layer

L'accesso ai driver dei dispositivi normalmente non è molto standardizzato quindi Android effettua un'astrazione di ciascuno di essi con una libreria nativa condivisa. La libreria è un oggetto condiviso aderente a un'interfaccia comune che supporta ogni principale driver di hardware. Questo comporta che ogni produttore deve implementare una libreria comune e astrarre le complicazioni relative al design del suo specifico dispositivo.

1.2.3 Libraries

Le librerie native sono scritte in C / C++. Il loro scopo principale è supportare il layer Android Application Framework. Alcune di queste sono create appositamente per Android, altre sono prelevate dalla community open source per completare il sistema operativo.

- Surface Manager ha la responsabilità di gestire le view, ovvero ciò da cui un'interfaccia grafica è composta. Il suo compito è infatti di coordinare le diverse finestre che le applicazioni vogliono visualizzare sullo schermo.
- Open GL ES per la grafica 3D; permette di accedere alle funzionalità di un acceleratore grafico integrato; SGL per la grafica 2D
- Media Framework per la gestione dei CODEC per i vari formati di acquisizione e riproduzione audio e video

- Freetype, motore di rendering dei font
- SQLLite, implementa un database SQL che il framework rende disponibile alle stesse applicazioni
- Webkit, browser engine
- SSL, per la gestione dei Secure Socket Layer
- Libc, implementazione della libreria standard C libc ottimizzate per dispositivi embedded.

1.2.4 Android Runtime

L'Android Runtime si compone delle Core Libraries e della Dalvik Virtual Machine, quest'ultima già trattata in precedenza.

Le Core Libraries forniscono molte delle funzionalità delle analoghe librerie disponibili per il linguaggio di programmazione Java. Per le applicazioni, in fase di compilazione avremo bisogno del file "android.jar", contenente le classi di sistema relative alla versione Android su cui si basa l'applicazione, per la creazione del bytecode Java, mentre in esecuzione il device metterà a disposizione la versione "dex" del runtime che costituisce appunto la core library.

1.2.5 Application Framework

Il framework è un ambiente che offre numerose librerie e servizi a supporto degli sviluppatori:

- Activity Manager gestisce il ciclo di vita di un'activity, ossia un elemento assimilabile a una schermata attraverso il quale l'utente interagisce con l'applicazione
- Package Manager gestisce il ciclo di vita delle applicazioni
- Window manager gestisce le finestre delle diverse applicazioni presenti sul device
- Telephony Manager gestisce l'interazione con i servizi di chiamata e di gestione dei messaggi
- Content Provider gestisce la condivisione di informazioni tra i vari processi
- Resource Manager gestisce tutto l'insieme di file di cui è composta l'applicazione

come ad esempio menu e immagini. Anche le risorse sono ottimizzate nel processo di creazione dell'applicazione

- View System gestisce il rendering dei componenti e degli eventi associati
- Location Manager fornisce le API per il supporto alla localizzazione
- Notification Manager si occupa della gestione delle notifiche (ad esempio accensione LED, vibrazione)

1.2.6 Applications

Al livello più alto sono presenti le applicazioni utente. E' interessante osservare che le funzionalità base del sistema come il telefono sono considerate alla stregua delle altre applicazioni installate dall'utente.

1.3 Le app

I quattro elementi principali per la costruzione di un'applicazione sono:

- Activity. Le attività sono quei blocchi di un'applicazione che interagiscono con l'utente utilizzando lo schermo ed i dispositivi di input messi a disposizione dallo smartphone. Comunemente fanno uso di componenti grafici già pronti, come quelli presenti nel pacchetto android.widget. Un'applicazione può avere più activity, il cui ciclo di vita (cinque stati possibili: starting, running, stopped, paused, destroyed) è gestito dall'Activity Manager, servizio presente nell'Application Framework, come visto in precedenza. L'Activity Manager è responsabile della creazione, distruzione e gestione delle activity.
- Service. Un servizio è eseguito in background e non interagisce direttamente con l'utente. Può svolgere le stesse azioni dell'activity ma senza alcuna interfaccia utente. I service sono utili per quelle azioni da eseguire per un certo periodo di tempo, a prescindere da quanto sia mostrato sullo schermo. I servizi hanno un ciclo di vita più semplice rispetto alle activity, infatti un servizio può essere solo avviato o interrotto
- Broadcast Receiver. Rappresentano l'implementazione di un meccanismo publish/subscribe a livello di sistema. Il receiver è un codice che viene attivato solo in

concomitanza di un evento, al quale il receiver stesso è iscritto. Lo stesso sistema esegue il broadcast di eventi continuamente. Ad esempio eventi quali la ricezione di un messaggio o di una chiamata sono trasmessi e potrebbero attivare diversi receiver scatenando diverse azioni come avviare ad esempio un'activity o un service

- Content Provider. Sono interfacce per la condivisione di dati tra le applicazioni e costituiscono pertanto un canale di comunicazione tra le differenti applicazioni installate nel sistema. Questo meccanismo è necessario perché ciascuna applicazione è eseguita in una propria sandbox, isolata dalle altre per motivi di sicurezza

Un'applicazione è costituita da uno o più di questi elementi [1].

1.3.1 I pacchetti APK

Un progetto Android si compone di molti elementi. Oltre ai file sorgenti suddivisi in package, sono presenti librerie, il già citato file "android.jar", file generati a seguito della compilazione come "R.java" che contiene un insieme di puntatori atti a identificare le risorse dell'app in fase di runtime, le classi compilate e racchiuse nel file "classes.dex", le risorse quali le componenti audiovisive, i layout, i file xml organizzati in cartelle (drawable, layout, values, xml), file di configurazione e il file "AndroidManifest.xml". Quest'ultimo contiene delle informazioni quali ad esempio il nome del package java dell'applicazione che costituisce un suo identificatore univoco, i componenti come le activity e i servizi, la dichiarazione dei permessi richiesti dall'app, il livello minimo di API richiesto, la versione e altre informazioni necessarie per la sua corretta esecuzione.

Le applicazioni Android sono distribuite sotto forma di file APK (Android Package). Al loro interno vengono raccolti e ottimizzati alcuni degli elementi sopra citati, quali il file AndroidManifest.xml, il file classes.dex, le risorse, le eventuali librerie native, la firma che certifica l'autore. Ottenuto il file APK è possibile installare l'applicazione su un terminale reale o emulato.

Capitolo 2: Testing

Filo conduttore di questo capitolo è l'attività di verifica e convalida, con particolare riferimento all'approccio dinamico, ossia alla fase di test. Saranno presentati le tipologie e le tecniche usate per lo svolgimento di tale attività e l'uso di tecnologie per automatizzarla. La trattazione lascerà spazio all'analisi dell'Android Testing Framework e al framework Robotium per il testing "black box" dell'interfaccia grafica delle applicazioni mobili in Android. Infine verranno presentati alcuni strumenti utilizzati in tale piattaforma.

2.1 Attività di verifica e convalida

Durante e dopo il processo di implementazione, un sistema software deve essere verificato per assicurarsi che sia conforme alle sue specifiche e fornisca le funzionalità offerte attese dai clienti. Si parla in tal caso di "verifica" e "convalida".

La verifica si confà alla domanda "stiamo costruendo il prodotto giusto?" e il suo ruolo è quello di controllare che il software sia conforme alle sue specifiche, ossia deve soddisfare i requisiti funzionali e non funzionali specificati.

La convalida si confà alla domanda "stiamo costruendo il prodotto nel modo giusto?" e si prefigge come scopo quello di assicurare che il sistema software rispetti le attese del cliente.

Queste due attività si svolgono dopo ogni stadio del processo software: cominciano con la revisione dei requisiti e continuano attraverso la revisione dei progetti e l'ispezione del codice fino al test del prodotto.

Nel processo di verifica e convalida ci sono due approcci complementari al controllo e

all'analisi del sistema:

- ispezione del software, tecnica statica dove non è necessaria l'esecuzione, in cui sono analizzati il documento dei requisiti, i diagrammi di progettazione e ispezionato il codice sorgente del programma. E' possibile l'uso di analizzatori statici automatici.
- test del software, tecnica dinamica in quanto richiede l'esecuzione del programma in esame attraverso i casi di test sottomessi. Si esaminano gli output e il suo comportamento operativo.

Come annunciato in precedenza, ci concentreremo esclusivamente sulla tecnica dinamica, il test [2].

2.2 Test

Le tecniche statiche di ispezione possono controllare solo la corrispondenza tra un programma e le sue specifiche, non possono dimostrare l'affidabilità o le prestazioni. Il test consiste invece nell'eseguire il programma utilizzando dati simili a quelli reali e, in base agli output ottenuti, verificare il corretto funzionamento o la presenza di anomalie. Due sono i tipi di test:

- test di convalida per verificare il rispetto dei requisiti, le prestazioni e l'affidabilità
- test dei difetti per verificare appunto la presenza dei difetti

In presenza di errori scoperti dal test, l'attività di debugging dovrebbe localizzare e correggere questi difetti. La successiva convalida del sistema prende il nome di test di regressione, ovvero sono eseguiti nuovamente i test precedenti e si verifica che le modifiche non abbiano introdotto nuovi errori.

Essendo costoso il processo di verifica e convalida, esso dovrebbe essere attentamente pianificato.

Il test può riguardare:

- i singoli componenti (ad esempio funzioni, metodi, oggetti, classi di oggetti)
- l'intero sistema; richiede l'integrazione di due o più componenti che implementano le funzionalità del sistema. Spesso è distinto in:

- test di integrazione che ha lo scopo di trovare i difetti; il team di test ha accesso al codice sorgente e, in presenza di errori, viene identificato il componente da sottoporre a debugging
- test della release o di funzionalità, in cui viene testata una versione del sistema da rilasciare agli utenti; la tipologia è "black box" cioè è verificato solo il corretto funzionamento o meno, disinteressandosi della struttura interna. Si contrappone al test di tipo "white box" in cui si utilizza la conoscenza della struttura del programma per progettare test che lo coprano interamente

Nell'ambito del test di sistema, la progettazione dei test case è quella parte di tale attività in cui si progettano gli input e gli output attesi, con l'obiettivo di creare dei test case in grado di evidenziare difetti del sistema e di convalidare i requisiti [2][9].

2.2.1 Test Automation

Per "test automation" si intende l'implementazione e l'uso di tecnologie software per la costruzione e l'esecuzione parziale o totale di casi di test ripetibili e consistenti, con l'obiettivo di ridurre i tempi ed i costi del software testing.

È un rafforzamento delle metodologie e dei processi utilizzati nel test manuale, anche se non li sostituisce del tutto, poiché alcune verifiche (ad esempio verifiche sul layout grafico e sull'usabilità del prodotto) restano più efficaci se eseguite manualmente.

Casi tipici di utilizzo di test automation riguardano i test funzionali di un prodotto, che spesso rendono necessario ripetere più volte lo stesso test con configurazioni differenti o con differenti valori dei dati in ingresso. La necessità di molteplici ripetizioni con le stesse configurazioni si verifica anche a fronte di nuovi rilasci del software, al fine di verificare che le modifiche apportate non abbiano introdotto delle regressioni (regression test).

Automatizzare il caso di test, in modo da essere eseguito senza l'intervento umano, ha un certo costo iniziale che viene ripagato nel tempo a seguito delle numerose riesecuzioni dei casi di test.

Evita inoltre procedure noiose, ripetitive e potenzialmente soggette a errori umani.

Può però essere necessaria una fase di valutazione dell'efficacia dei casi di test e una fase

di riduzione dei casi di test ridondanti [9].

I test case possono essere generati automaticamente a partire ad esempio da:

- documentazione di analisi (specifica dei requisiti)
- documentazione di progetto (Model based testing)
- analisi statica del codice sorgente
- osservazione di esecuzioni reali dell'applicazione (user session testing)
- interazione casuale con l'applicazione (Monkey testing). Il Monkey testing è la specializzazione del Random testing nel quale gli input sono gli eventi. Nel Random vengono generate sequenze casuali di input, e può essere utilizzato per cercare possibili situazioni di crash e eccezioni non gestite

Per l'esecuzione automatica dei casi di test la soluzione più efficace è quella di scrivere codice con framework XUnit che consente di eseguire test black box e white box. JUnit ad esempio è un insieme di classi Java che l'utente estende per creare un ambiente automatico di test. Le classi di test sono separate da quelle originali in modo da non influenzare il comportamento del sistema.

In generale un test JUnit è un metodo i cui statements convalidano il funzionamento di una parte dell'applicazione sotto test. I metodi sono organizzate in classi. Più classi creano una suite di test [10].

2.2.2 Generazione dei test automatici basati su sessioni utente

Come enunciato nel paragrafo precedente, uno dei modi in cui è possibile generare automaticamente dei casi di test è partire dall'analisi delle sessioni utente (User Session), ovvero delle sequenze dei valori di input immessi e di output ottenuti in utilizzi reali del software.

In pratica, vengono installati strumenti che siano in grado di mantenere un log di tutte le interazioni che avvengono tra gli utenti (alpha o beta tester) dell'applicazione da testare e l'applicazione stessa (fase di Capture); a partire da tali dati vengono formalizzati casi di test che replichino le interazioni "catturate" (fase di Replay).

In tal modo è possibile ottenere casi di test che siano rappresentativi dei reali utilizzi

dell'applicazione da parte dei suoi utenti.

Il vincolo è dato dall'esistenza del prodotto software completo [9].

Vantaggi:

- lo sforzo per la generazione dei casi di test è molto limitato
- è possibile che gli utenti scoprano funzionamenti anomali che i progettisti non avevano previsto
- la riesecuzione dei casi di test può essere completamente automatizzata

Svantaggi:

- Scarsa efficienza. Con molta probabilità, per perseguire l'obiettivo dei test, sarà necessario un numero considerevole di test case, spesso con parti in comune. Si possono applicare a tal riguardo delle tecniche di minimizzazione basata sulla copertura, in cui si fissa un obiettivo di copertura, si valuta il grado di copertura di ogni caso di test e si esegue un algoritmo che estragga il più piccolo insieme di test che massimizzi la copertura
- Ridotta efficacia perché molti casi limite potrebbero non essere testati
- Per ottenere esecuzioni significative è probabile che sia necessario raccogliere sessioni per un tempo prolungato
- Può essere difficile riutilizzare i casi di test prodotti, a seguito di un intervento di manutenzione sul software o a causa della natura della registrazione stessa

Nel corso del lavoro di tesi si è fatto largo uso di test automatici basati su sessioni utente registrati e riprodotti con il supporto di strumenti descritti nel prosieguo del capitolo. L'utente, in particolare, ha interagito direttamente con l'applicazione in esame attraverso la sua interfaccia grafica, sollecitata attraverso eventi.

2.2.3 Testing dell'interfaccia utente

Le interfacce utente rappresentano un caso di sistema ad eventi, il quale si trova in uno stato stabile fino al momento in cui interviene un evento utente, che innesca l'esecuzione di una porzione di codice abilitata alla sua gestione.

I test, solitamente di tipo "black box", di tali interfacce sono progettati considerando le

possibili interazioni eseguibili dall'utente; è pertanto necessario un modello descrittivo di tutte le possibili interazioni.

Tipicamente è utilizzato il modello di macchina a stati (FSM acronimo di Finite State Machine), dove lo "stato" rappresenta l'insieme degli elementi costitutivi dell'interfaccia (ad esempio finestre, widget, layout) corrispondente a uno stato dell'automa. Gli eventi sull'interfaccia come la selezione di un elemento in una lista, sono modellati come ingressi che innescano transizioni nell'automa. La sequenza di input corrisponde a una sequenza di stati visitati.

Ottenere l'FSM che descrive l'interfaccia su cui effettuare i test case spesso non è semplice. Essa può essere prodotta in fase di sviluppo dell'applicazione oppure ricostruita tramite Reverse Engineering a partire dall'interfaccia stessa [9].

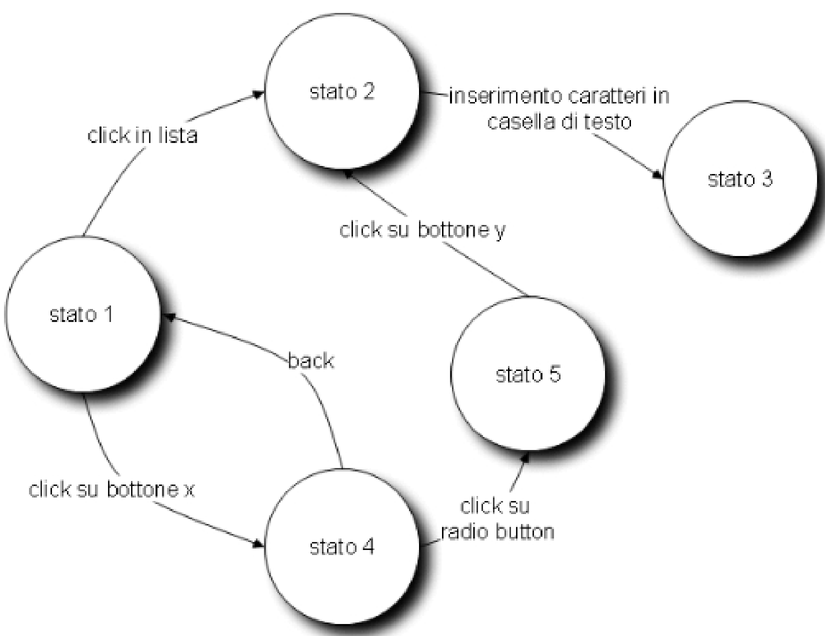


Figura 2.1: esempio di GUI modellata con FSM

2.3 Testing in Android

L'Android Testing Framework è una parte dell'ambiente di sviluppo che fornisce gli strumenti per testare tutti gli aspetti dell'applicazione mobile.

Tali strumenti prevedono come input il progetto dell'app (AUT, acronimo di Application

Under Test) dal quale prelevano le informazioni per creare automaticamente i file e le cartelle del progetto di test.

Il diagramma seguente illustra il framework:

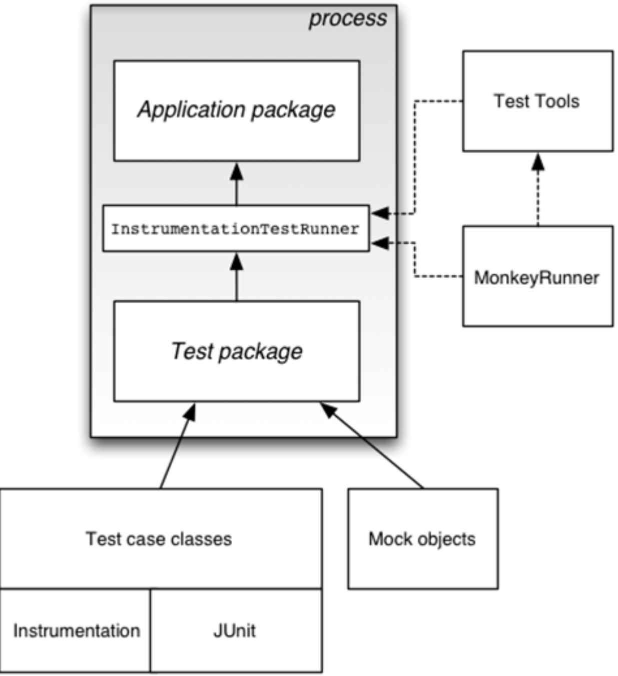


Figura 2.2: Android Testing Framework

Le suite di test Android sono basate su JUnit, esteso opportunamente per testare le parti dell'applicazione che fanno uso delle API Android e gestire al meglio il ciclo di vita delle componenti.

In generale un test JUnit è un metodo i cui statement testano una parte dell'AUT. I metodi sono organizzati in classi chiamate test case o test suite, eseguite da un test runner. In modo analogo, in Android i test sono raccolti in classi, a loro volta organizzati in package ed eseguite da uno specifico test runner (InstrumentationTestRunner).

In Android è necessario pertanto creare un progetto di test separato dal progetto dell'applicazione con gli opportuni riferimenti ad esso. Questo lavoro è facilitato dagli strumenti presenti nell'SDK che, a partire dall'AUT fornita come input, si occuperanno di creare e settare in modo automatico le informazioni del progetto di test. In particolare verrà impostato l'InstrumentationTestRunner come esecutore dei test, il nome del package sarà il nome del package dell'AUT con il suffisso ".test" per sottolineare la relazione

esistente e prevenire conflitti nel sistema, infine verranno create tutte le directory e i file necessari, tra cui il file AndroidManifest.xml.

Il framework di testing eseguirà gli elementi dell'applicazione, ad esempio le activity, come sua parte, utilizzando funzionalità di instrumentazione per poterla monitorare.

L'instrumentazione in Android consiste in un insieme di metodi di controllo o "ganci" nel sistema Android. Questi ganci controllano un componente Android in maniera indipendente rispetto al suo nomale ciclo di vita. Normalmente, tale componente viene eseguito in un ciclo di vita determinato dal sistema. Ad esempio all'avvio di un'activity di un'applicazione viene richiamato il metodo "onCreate()" seguito dalla chiamata del metodo "onResume()"; se l'utente avvia un'altra applicazione, viene richiamato il metodo "onPause()". E' possibile richiamare questi metodi di callback solo attraverso l'instrumentazione: ad esempio con il metodo "getActivity()" è possibile ottenere un riferimento all'activity in esecuzione e eseguire uno qualsiasi dei metodi che essa definisce [4].

Per ulteriori approfondimenti si rimanda al link della documentazione ufficiale: http://developer.android.com/tools/testing/testing_android.html

2.3.1 Robotium e Robotium Recorder

Robotium è un framework che consente, semplificandolo, automatizzandolo e velocizzandolo, il testing black box dell'interfaccia grafica dell'AUT, richiedendo una minima conoscenza del suo funzionamento e offrendo delle API per garantire l'interazione con essa.

I test case prodotti hanno la proprietà di essere facilmente leggibili e modificabili.

Il funzionamento di Robotium è basato sull'utilizzo di un oggetto denominato "solo"

Solo solo = new Solo(getInstrumentation(),getActivity());

Tramite esso è possibile interrogare e modificare i widget della UI, eventualmente anche senza conoscerne l'identificativo [6].

Robotium Recorder è invece uno strumento, basato su Robotium, in grado di registrare tutte le interazioni che l'utente effettua sull'AUT (fase di recording) e in seguito di

rieseguirle (fase di Replay) [7]. E' stato utilizzato per la generazione di test automatici basati su sessioni utente e sarà descritto in maniera più approfondita nel capitolo 4.

2.3.2 Monkey e MonkeyRunner

Monkey è un'utility fornita con l'android SDK, che è in grado di generare flussi di eventi utente pseudocasuali e eventi di sistema, registrando gli eventuali crash.

Monkeyrunner è un API che consente la scrittura di programmi in grado di controllare un dispositivo Android dall'esterno. Ad esempio è possibile scrivere un programma Python che installa un'applicazione, esegue casi di test, invia eventi, salva screenshot.

2.3.3 Android Ripper

Software per l'esplorazione dell'interfaccia utente di un'applicazione Android. La sua trattazione sarà oggetto del capitolo 3.

2.3.4 Emma

Emma è un tool di instrumentazione del codice sorgente che consente di valutare l'effettiva copertura del codice ottenuta a seguito dell'esecuzione di un insieme di casi di test. Permette di conoscere quante e quali parti del codice siano state effettivamente attraversate durante l'esecuzione di un test, in modo da ottenere una stima quantitativa della sua efficacia. Genera report e metriche in formato testo, xml e html.

Capitolo 3: Android Ripper

In questo capitolo verrà introdotto il software Android Ripper, capace di testare in maniera sistematica e casuale un'applicazione Android attraverso la sua interfaccia. Verrà introdotta l'architettura, i principali blocchi costitutivi, le interazioni tra essi e l'algoritmo su cui basa il suo funzionamento.

3.1 GUI testing di applicazioni Android

L'interfaccia grafica di un'applicazione Android si compone tipicamente di schermate composte da widget.

Essa rappresenta un sistema ad eventi, in cui le interazioni con i widget, ad esempio click, longclick, gesture, rappresentano gli eventi che se, eseguiti, avviano un "gestore" in grado di servirli correttamente; ad esempio un click su un bottone di un'activity può causare l'avvio di una nuova activity.

Si possono modellare le interazioni e le schermate con una macchina a stati, in cui uno "stato" dell'interfaccia utente corrisponde a uno stato dell'automa.

Come vedremo nel prosieguo del capitolo, uno degli output del Ripper (file Activities.xml) racchiude al suo interno tutti gli stati visitati a seguito delle interazioni effettuate. Ciascuna sezione del file rappresenta una descrizione dell'activity comprensiva di tutti i widget.

Il passaggio da uno stato a un altro avviene a seguito di una sollecitazione da parte dell'utente.

Le tecniche di GUI testing possono essere divise in tre categorie:

- Tecniche Model Based, in cui esiste una descrizione dell'applicazione sotto test sufficiente alla generazione automatica dei test case per la verifica del software
- Tecniche Random Testing, in cui gli eventi sono generati in maniera casuale
- Tecniche Model Learning, in cui l'applicazione viene esplorata senza una conoscenza pregressa della sua struttura seguendo una strategia di navigazione

Il Ripper è capace di adottare sia la tecnica Random che Model Learning specificandola tramite opzione da riga di comando [3].

3.2 Ripper

Il Ripper è un software capace di eseguire automaticamente test in maniera casuale e sistematica su un'applicazione Android. Nell'ambito di questo lavoro di tesi ci occuperemo della sola sezione sistematica.

Introduciamo alcuni dei concetti su cui si basa il suo funzionamento [3]:

- Activity State: rappresenta l'Activity correntemente visualizzata (running activity), in termini dei suoi parametri e dei widget in essa contenuti; i widget saranno a loro volta descritti in termini di uno o più attributi
- Evento: un azione utente eseguita su uno dei widget della GUI in grado di determinare il passaggio da una schermata ad un'altra
- Input: un'interazione con la GUI dell'applicazione sotto test che non provoca cambiamenti di stato
- Azione: la sequenza ordinata di un evento e di tutti gli input che lo precedono. Essa rappresenta le operazioni necessarie ad indurre un passaggio di stato nella GUI dell'applicazione sotto test
- Task: coppia (Azione, GUI State) che rappresenta un azione eseguita sull'interfaccia
- Plan: è l'insieme dei task che descrivono le possibili azioni con le quali è possibile interagire per innescare ulteriori transizioni e proseguire l'esplorazione
- Trace: un rapporto sull'esecuzione di un task. Per ogni azione componente il task, il trace dovrà contenere lo stato dell'Activity all'inizio, la sequenza degli input, l'evento

e lo stato dell'Activity alla fine della transizione

- Sessione: l'insieme di tutte le operazioni eseguite dal Ripper, cominciando con la prima inizializzazione fino alla generazione del file di output

3.2.1 Input e Output

In input il Ripper prevede l'immissione in appositi file ("ripper.properties" e "systematic.properties") di alcune informazioni riguardanti l'app da testare (il percorso, il package principale, l'activity principale) e l'SDK (la porta e il nome dell'AVD su cui verrà eseguito il test); inoltre è possibile specificare alcune configurazioni relative a elementi costitutivi del Ripper come il planner e il configurator descritti nel prosieguo del capitolo. Di seguito è riportato un esempio dei file sopra citati:

```
AUT_PATH = C:/Users/Testing/AUT/TippyTipper
TEST_SUITE_PATH = %PWD%/AndroidRipper
SERVICE_APK_PATH = %PWD%
AVD_NAME = avd_ripper
AVD_PORT = 5554
```

File ripper.properties

```
aut_package = net.mandaria.tippytipper
aut_main_activity = net.mandaria.tippytipper.TippyTipperApplication
avd_name = avd_ripper
avd_port = 5554
coverage=1
ping_failure_threshold=1
planner=it.unina.android.ripper.planner.HandlerBasedPlanner
comparator=it.unina.android.ripper.comparator.GenericComparator
comparator_configuration=CustomWidgetSimpleComparator
```

File systematic.properties

Al termine del processo di ripping otteniamo i seguenti file:

File e directory	Esempio	Contenuto informativo
File ".bin"	| current_ActivityStateList.bin | current_TaskList.bin | |	Lista degli stati e lista dei task sono dei file di elaborazione intermedi usati e aggiornati durante l'esecuzione.
Directory "coverage"	+---coverage | coverage00000_ec.ec | coverage00001.ec | coverage00001_ec.ec | […] | coverage00033.ec | coverage00033_ec.ec	File di coverage generate con Emma. Necessari per il calcolo della copertura ottenuta
Directory "junit"	+---junit | junit-log-00000.xml | junit-log-00001.xml | […] | junit-log-00033.xml	Risultati dell'esecuzione dei casi di test in formato xml
Directory "logcat"	+---logcat | logcat_5554_0.txt | logcat_5554_1.txt | […] | logcat_5554_33.txt | |	File di logcat ottenuti col meccanismo messo a disposizione dall'SDK. Rappresentano il tracciamento cronologico delle operazioni con la relativa visualizzazione sullo standard output del sistema.
Directory "model"	+---model | activities.xml | log_0.xml | log_1.xml | […] | log_34.xml	File "activities.xml" contiene la lista degli stati esplorati durante l'esecuzione. I file di log contengono informazioni riguardanti gli eventi eseguiti.

3.2.2 Architettura

Di seguito sono riportati i principali componenti del software [3]:

- Engine è il controllore centralizzato che racchiude la business logic del Ripper. Gestisce il processo di ripping, supervisiona il flusso di esecuzione e garantisce lo scambio di messaggi fra le componenti
- Scheduler (o dispatcher) è il componente che decide l'ordine di esecuzione dei task generati per l'esplorazione dell'applicazione sotto test. Esso si occupa di memorizzare i task in attesa di essere eseguiti in un'opportuna struttura dati e fornisce all'Engine, di volta in volta, il task da eseguire
- Robot è il componente che si occupa dell'interfacciamento con l'applicazione, ed esegue i task pianificati per esplorarla, sfruttando la classe Instrumentation messa a

disposizione dal Testing Framework di Android. Esso agisce riproducendo le interazioni che un utente reale eserciterebbe per svolgere il compito descritto nel task da eseguire

- Extractor ha la responsabilità di estrarre le informazioni che determinano lo stato dell'applicazione, ed in particolare l'aspetto dell'interfaccia grafica dell'Activity attiva nel momento in cui il componente svolge il suo compito. A seguito dell'elaborazione, l'extractor mette a disposizione del Ripper una Activity Description, una descrizione dell'istanza di interfaccia corrente
- Abstractor crea e fornisce al Ripper un modello esportabile dell'interfaccia correntemente visualizzata dall'applicazione, al fine di rendere possibile la costruzione di un modello dell'applicazione la cui validità si estenda oltre la durata della singola sessione di ripping
- Strategy si occupa di:
    - confrontare lo stato corrente dell'applicazione e gli stati già visitati in precedenza. In caso di equivalenza lo stato corrente non necessita di ulteriore esplorazione
    - prendere le decisioni che guidano il flusso di esecuzione del Ripper in base al risultato del confronto fra stati, ai dati forniti dall'abstractor, alla descrizione dell'ultimo task eseguito ed eventualmente ad ulteriori informazioni interne al componente, quali il tempo di esecuzione del software ed il livello di profondità raggiunto
- Planner ha il compito di generare il plan di esplorazione dell'applicazione a partire dall'activity corrente. I nuovi task saranno generati in base all'analisi del risultato dell'esecuzione del task che ha portato l'applicazione in questo stato, e solo se di tale stato è stata richiesta l'esplorazione da parte dello strategy. Il planner esamina la struttura dell'istanza di interfaccia visualizzata e seleziona quei widget che, in base a regole prestabilite o definite dal tester, sono ritenuti in grado di innescare una transizione di stato nell'applicazione

- Persistence Manager provvede a tutte le operazioni da e verso le memorie di massa (lo storage interno o la scheda SD del dispositivo)

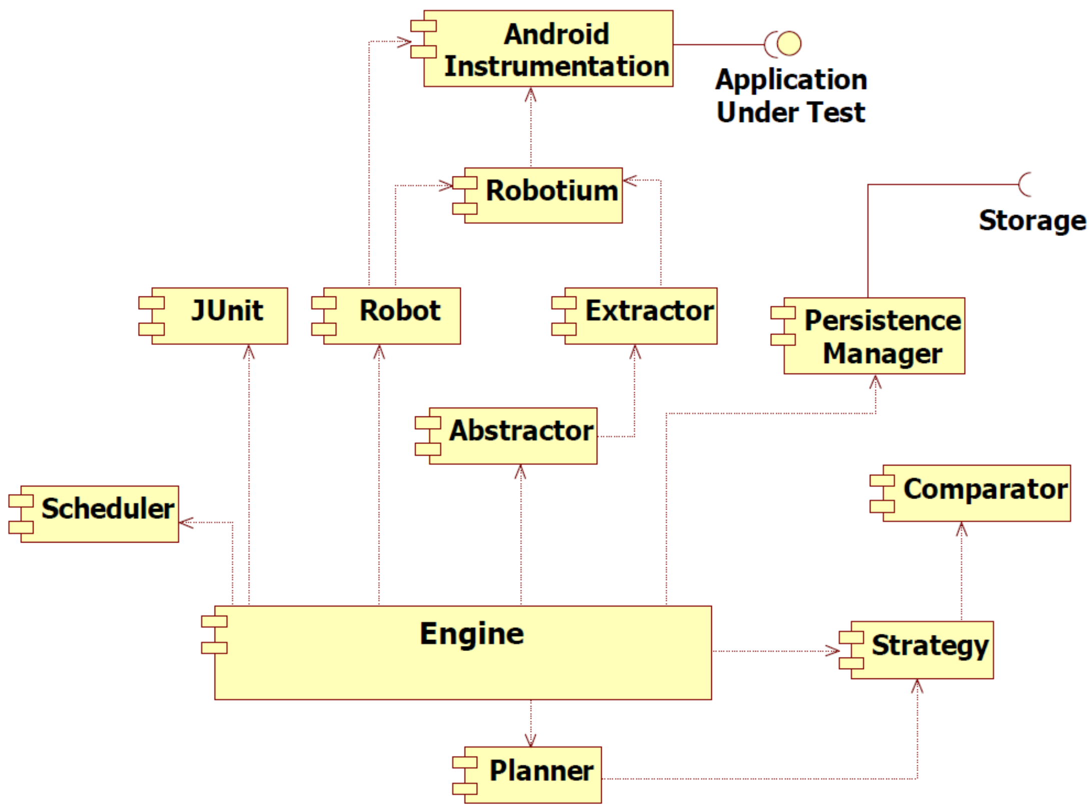


Figura 3.1: Architettura del Ripper

3.2.2 Funzionamento

Il processo su cui si basa il Ripper è diviso logicamente in due passi distinti:

1. installazione
2. esecuzione

L'installazione avviene attraverso il comando:

java –jar AndroidRipperInstaller.jar

Questo comando comporta la modifica automatica di alcuni file dell'applicazione fornita come input, la quale è compilata e installata nell'AVD, insieme alle componenti del Ripper.

A un alto livello di astrazione il Ripper può essere visto come l'insieme di quattro blocchi principali. Il primo di questi blocchi (l'installer) riguarda la funzione di installazione

appena descritta.

I blocchi sono illustrati nel seguente schema:

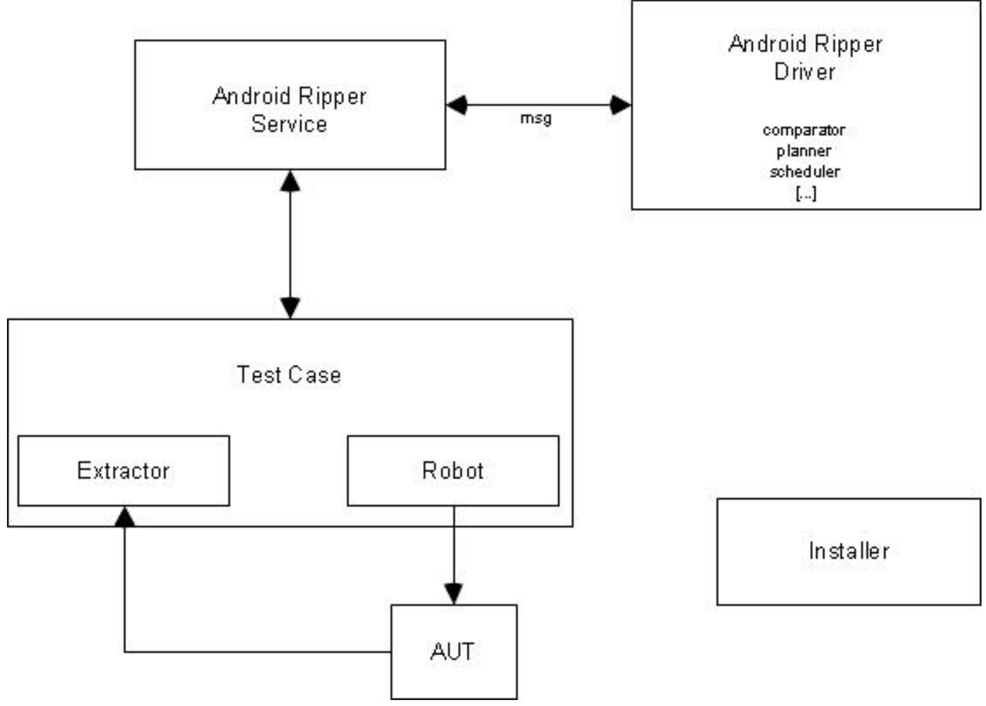


Figura 3.2: Diagramma concettuale delle componenti del Ripper

Per l'esecuzione è necessario specificare il tipo di test (casuale o sistematico) attraverso il comando:

java -jar AndroidRipper.jar s systematic.properties (sistematico)

java -jar AndroidRipper.jar r random.properties (random)

Il componente Driver rappresenta la "mente" del processo di ripping, comandando la sezione Test Case (il "braccio") attraverso scambio di messaggi. Il "Test Case" esegue le operazioni interagendo con l'applicazione. Per ogni task eseguito, viene estratta un'activity description (compito dell'extractor) e spedita tramite scambio di messaggi al componente "Driver" il quale agirà secondo un algoritmo riportato di seguito. Un esempio di messaggio è la richiesta di descrizione della GUI corrente richiesta dal Driver. Al messaggio di richiesta corrisponde un messaggio di risposta contenente l'xml corrispondente alla descrizione. Anche le operazioni come la richiesta di copertura e la chiusura del Ripper e i relativi "ack" sono effettuati previo invio di messaggi.

E' bene ricordare che tutte le operazioni verranno eseguite in maniera completamente

automatica, senza il supporto dell'utente.

L'esecuzione, a sua volta, si suddivide a livello logico in tre fasi principali [3]:

- Inizializzazione, durante la quale le varie componenti del Ripper vengono istanziate ed inizializzate
- Setup, in cui il Ripper si interfaccia all'applicazione da esplorare, ne esamina lo stato iniziale, salvandolo nel file "Activities.xml" di cui forniamo un estratto:

```
<?xml version="1.0"?><states>
<activity class="kdk.android.simplydo.SimplyDoActivity" id="a1" keypress="FALSE"
longkeypress="FALSE" menu="TRUE" name="SimplyDoActivity" tab_activity="0"
title="Simply Do" uid="1">
  <listener class="SensorListener" present="FALSE"/>
  <listener class="OrientationListener" present="FALSE"/>
  <listener class="SensorEventListener" present="FALSE"/>
  <listener class="LocationListener" present="FALSE"/>
  <widget ancestor_id="-1" ancestor_type="root" class="android.widget.LinearLayout"
count="2" enabled="TRUE" id="-1" index="0" name="" p_id="-1" p_name=""
p_type="com.android.internal.policy.impl.PhoneWindow.DecorView"
simple_type="linearLayout" visible="TRUE">
    <listener class="AnimationListener" present="FALSE"/>
    <listener class="OnKeyListener" present="FALSE"/>
    <listener class="OnHierarchyChangeListener" present="FALSE"/>
    <listener class="OnClickListener" present="FALSE"/>
    <listener class="OnFocusChangeListener" present="FALSE"/>
    <listener class="OnLongClickListener" present="FALSE"/>
  </widget>
  […]
</states>
```

In base allo stato salvato, viene generata la lista di eventi scatenabili sull'activity creando una lista di task (TaskList), inizialmente vuota.
Lo stato dell'activity iniziale, opportunamente elaborato, viene memorizzato nella lista degli stati visitati per successivi confronti, ed inviato al Planner per la compilazione di un piano di esplorazione contenente l'elenco dei primi task generati, che andranno successivamente eseguiti nella fase di esplorazione

```
Algoritmo    Setup
Input: androidApplication = the application under test
  robot.bindApplication(androidApplication)
  baseActivity ← robot.getCurrentActivity()
  activityDescription ← extractor.describe(baseActivity)
  activityState ← abstractor.abstract(activityDescription)
  strategy.storeVisitedState(activityState)
  plan ← planner.createPlan(activityState)
  scheduler.addPlan(plan)
```

Figura 3.3: Algoritmo relativo alla fase di Setup

- Esplorazione: sono eseguiti i task estratti dallo scheduler, sono elaborati i risultati e generati eventualmente nuovi task

```
Algoritmo    Exploration
  while scheduler.hasMoreTasks() do
    task ← scheduler.getNextTask()
    strategy.setCurrentTask(task)
    robot.process(task)
    currentActivity ← robot.getCurrentActivity()
    activityDescription ← extractor.describe(currentActivity)
    activityState ← abstractor.abstract(activityDescription)
    isStateNew ← strategy.compareState(activityState)
    if isStateNew then
      strategy.storeVisitedState(activityState)
    end if
    if strategy.transitionOccurred() then
      trace ← abstractor.createTrace (task, activityState)
      persistence.addTrace(trace)
      if strategy.explorationNeeded() then
        plan ← planner.createPlan(activityState)
        scheduler.addPlan(plan)
      end if
      if strategy.sessionTermination() then
        close the session
        break the loop
      end if
    end if
  end while
```

Figura 3.4: Algoritmo relativo alla fase di Esplorazione

Inizialmente, l'Extractor e l'Abstractor forniscono una descrizione dell'activity corrente, ovvero quella ottenuta al termine dell'esecuzione del task. Tale descrizione viene poi inviata allo Strategy per effettuarne la comparazione con gli stati visitati in precedenza; se lo stato corrente non equivale a nessuno di quelli presenti nella activity list, allora dovrà essere aggiunto. A questo punto, lo Strategy individua se una transizione ha effettivamente avuto luogo alla fine del task eseguito. Se non vi è stata transizione, l'esecuzione riprende dall'inizio del ciclo con l'estrazione di un nuovo

task. In caso contrario, l'esecuzione prosegue con la generazione del trace che descrive l'esecuzione del task appena terminato. Esso viene poi passato al Persistence Manager per essere memorizzato su disco. L'insieme di questi trace costituirà l'output della sessione, dal quale sarà in seguito possibile estrarre un modello a stati dell'applicazione sotto test. Lo Strategy dovrà ora decidere se lo stato corrente è passibile di ulteriore esplorazione. Nel caso più semplice, tale decisione equivale all'esito del confronto appena effettuato: lo stato verrà esplorato se e solo se non è mai stato esplorato in precedenza. Infine, si controlla se uno dei criteri di terminazione è verificato. In tal caso, la sessione viene chiusa e si esce dal ciclo di iterazione. Altrimenti si procede con l'esecuzione di un nuovo task, se disponibile.

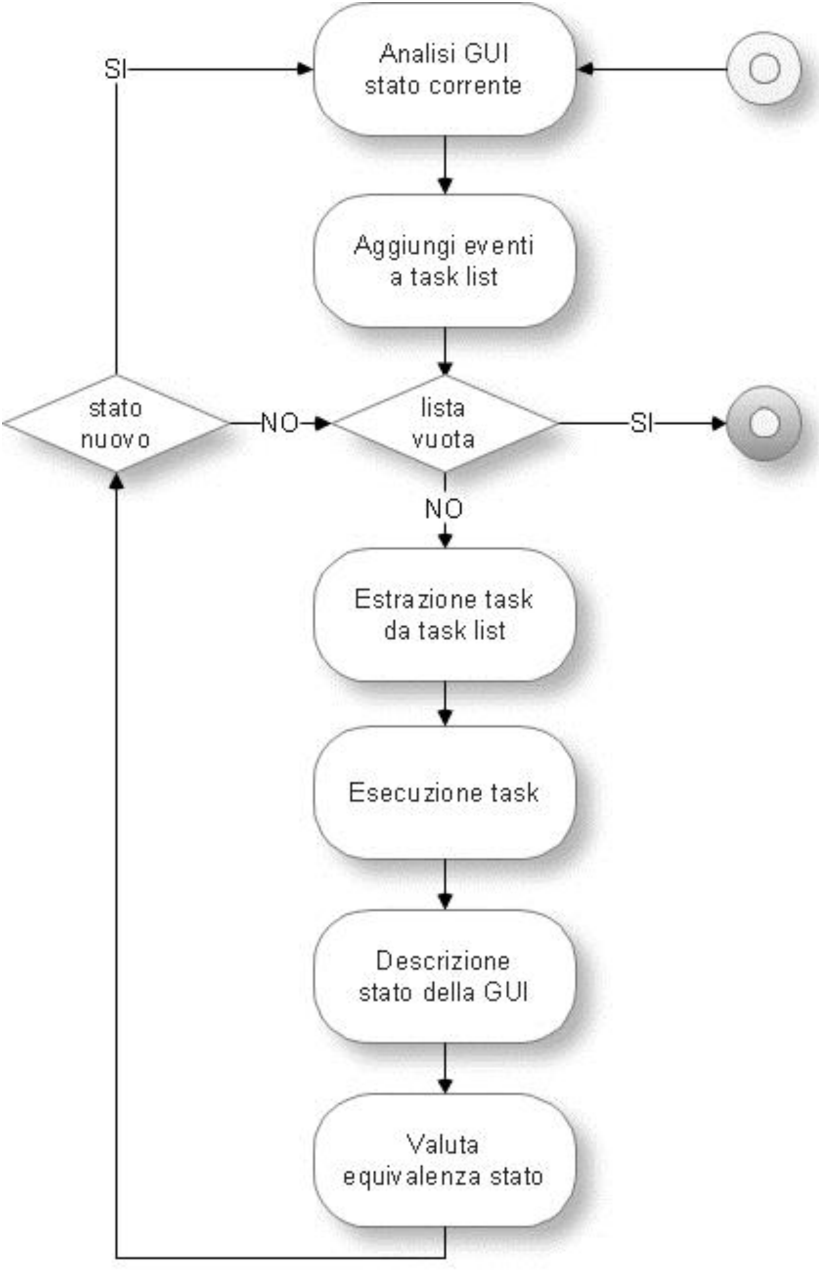


Figura 3.5: Activity Diagram Ripper

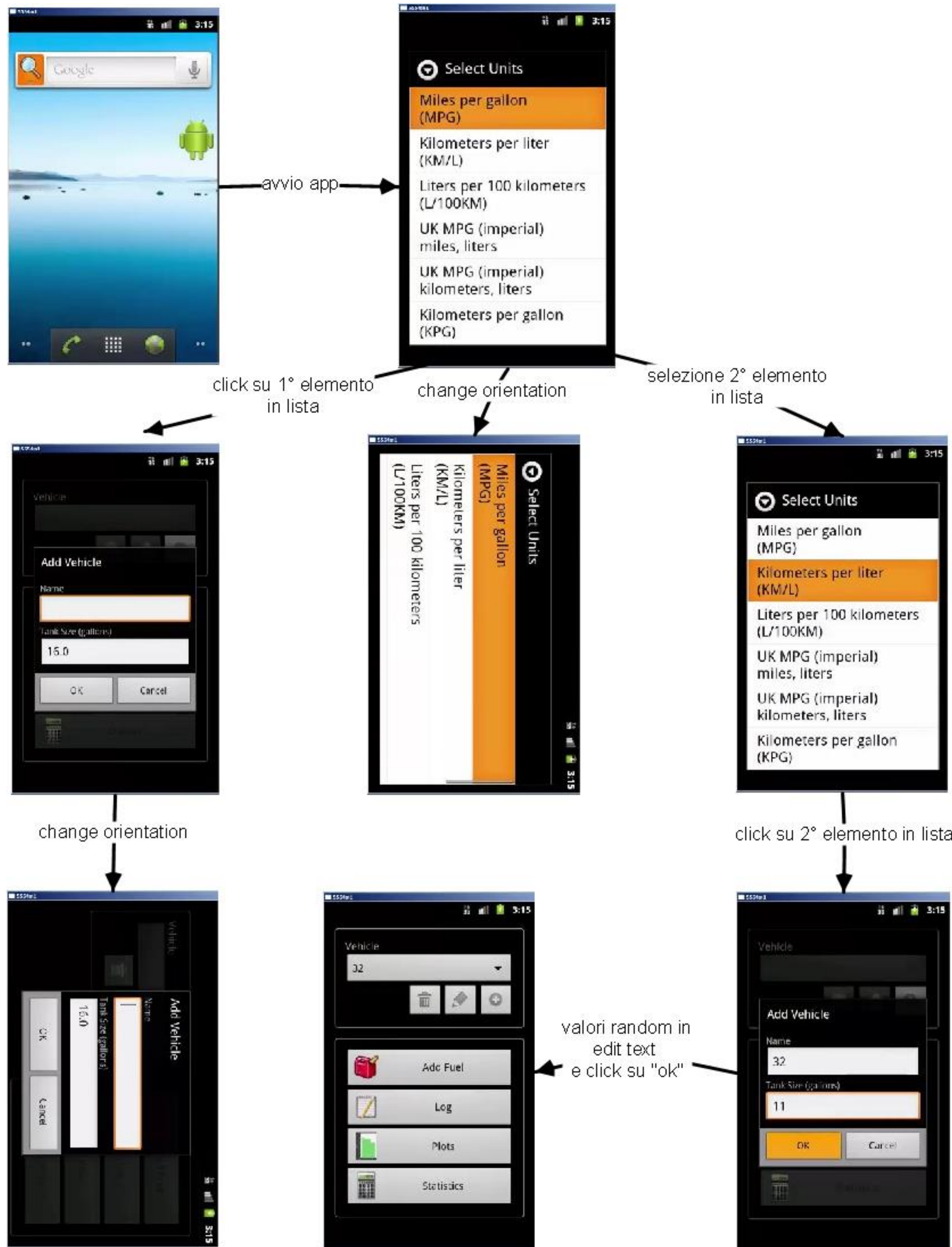


Figura 3.6: Esempio processo di ripping

Capitolo 4: Android Ripper "Ibrido"

Il capitolo 3 ha mostrato il principio di funzionamento, l'input fornito e l'output corrispondente, l'architettura del Ripper esistente (da ora in poi chiameremo Ripper originale). Il presente capitolo mostra invece come si è realizzato un fork di tale software (che chiameremo Ripper Ibrido), comprendente una importante serie di modifiche che ne hanno cambiato radicalmente la modalità e le fasi di esecuzione. Parliamo di Ripper "ibrido" perché il software originale era completamente automatico mentre questo ingloba sia la parte automatica sia la parte "manuale" intesa come supporto a sessioni utente registrate.

Restano invariate le caratteristiche che sovraintendono le operazioni di estrazione e confronto delle activity description, le operazioni di pianificazione, schedulazione ed esecuzione dei task, ossia l'ossatura dell'architettura originale.

Modificata invece in maniera radicale la logica di funzionamento (l'algoritmo di esecuzione) e la procedura di elaborazione degli output. Uno degli output è rappresentato dai file di coverage, i quali sono stati analizzati in maniera automatica attraverso dei file batch per la generazione dei report in formato html, testo e xml.

Inoltre è stato creato un tool ad-hoc per il supporto di sessioni utente registrate con il software Robotium Recorder, fornite come input aggiuntivo rispetto al Ripper originale.

4.1 Perché una tecnica ibrida

Nei capitoli precedenti sono state presentate le differenti tecniche con cui è possibile testare un'applicazione mobile attraverso la sua GUI: model based, random, model

learning (ML), user based session. Ciascuna tecnica ha le sue caratteristiche peculiari e un determinato campo di applicabilità. A tal proposito, la tecnica model based richiede come vincolo l'esistenza di un modello, di una descrizione dell'applicazione per generare i test, mentre la tecnica random potrebbe richiedere un tempo elevato per avere una copertura del codice ritenuta sufficiente. Considerando tali limitazioni, si è focalizzata l'attenzione sulla tecnica model learning e su quella basata su sessioni utente. Ciascuna di esse ha punti deboli e vantaggi, alcuni mutuamente esclusivi, come visto in precedenza in questo lavoro di tesi. Si pensi al fatto che le sessioni di test user based richiedano appunto risorse umane impegnate nell'attività di test, essendo una tecnica manuale, benché coadiuvate da software che consentono una certa automazione minimizzando lo sforzo richiesto, mentre la tecnica model learning è totalmente automatica. Inoltre le sessioni utente potrebbero non coprire interamente l'applicazione (esplorazione non esaustiva), mentre ML cerca di perseguire questo scopo ma al tempo stesso non garantisce che una specifica porzione dell'app sia effettivamente testata e quindi verificare il comportamento a seguito di eventi e input ben precisi.

Da queste considerazioni si è pensato di unire i vantaggi delle due tecniche ML e user based ottenendo pertanto una tecnica "ibrida" manuale/sistematica.

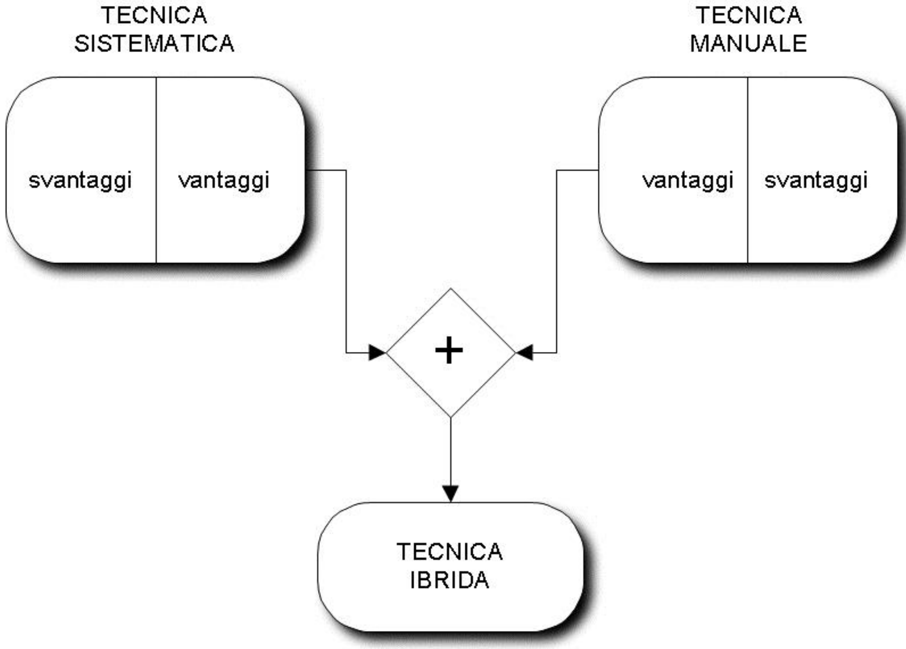


Figura 4.1: Tecnica ibrida

4.1.1 Componente sistematica

La base di partenza per la componente sistematica è il Ripper, descritto nel capitolo 3. La ragione di tale scelta è da ricercare nella flessibilità offerta da tale software che si presta a modifiche perché suddiviso in blocchi funzionali ben identificati. Inoltre, si è beneficiato del supporto ottenuto da parte dei ricercatori dell'Università di Napoli che attualmente lavorano al suo sviluppo e alla sua evoluzione.

Un altro software capace di esplorare in modo sistematico un'app Android è ad esempio A3E Automatic Android App Explorer (sito di riferimento: http://spruce.cs.ucr.edu/a3e/index.html).

4.1.2 Componente manuale

Le sessioni di test utente sono supportate da software che consentono di registrare in primo luogo il test e in secondo luogo offrono la possibilità di rieseguirlo (fasi di Capture e Replay). Evitano pertanto la scrittura a mano del test, attività tediosa e potenzialmente soggetta a errori. Sono state valutate due soluzioni commerciali: TestDroid Recorder (sito di riferimento: http://testdroid.com/) e Robotium Recorder (sito di riferimento: http://robotium.com/products/robotium-recorder).

Entrambe le soluzioni sono basate sul framework Robotium, consentono di testare un'app sia partendo dall'apk che dal codice sorgente e forniscono in output un file ben strutturato di tipo Robotium Junit facile da leggere e modificare.

TestDroid Recorder utilizza una libreria che estende le funzionalità di Robotium denominata “ExtSolo” mentre Robotium Recorder utilizza la libreria standard del framework.

Dato che il Ripper fa uso di quest'ultima e inoltre TestDroid Recorder non consente di registrare le interazioni basate sul long click, risultando meno usabile, si è scelto di utilizzare Robotium Recorder come base per ottenere i test case utente, elaborati opportunamente come vedremo nel prosieguo del capitolo.

4.2 La soluzione proposta: Ripper Ibrido

Il Ripper Ibrido nasce dalla volontà di unire l'approccio sistematico legato al Ripper originale con l'approccio manuale relativo all'esecuzione di test case utente registrati tramite Robotium Recorder concretizzando il concetto di creare una tecnica ibrida, come descritto in precedenza.

4.2.1 Architettura

A livello architetturale, gli elementi che costituivano il Ripper originale restano invariati nel Ripper Ibrido. Ad essi si affiancano però alcuni elementi nuovi relativi all'elaborazione ed esecuzione del test case utente.

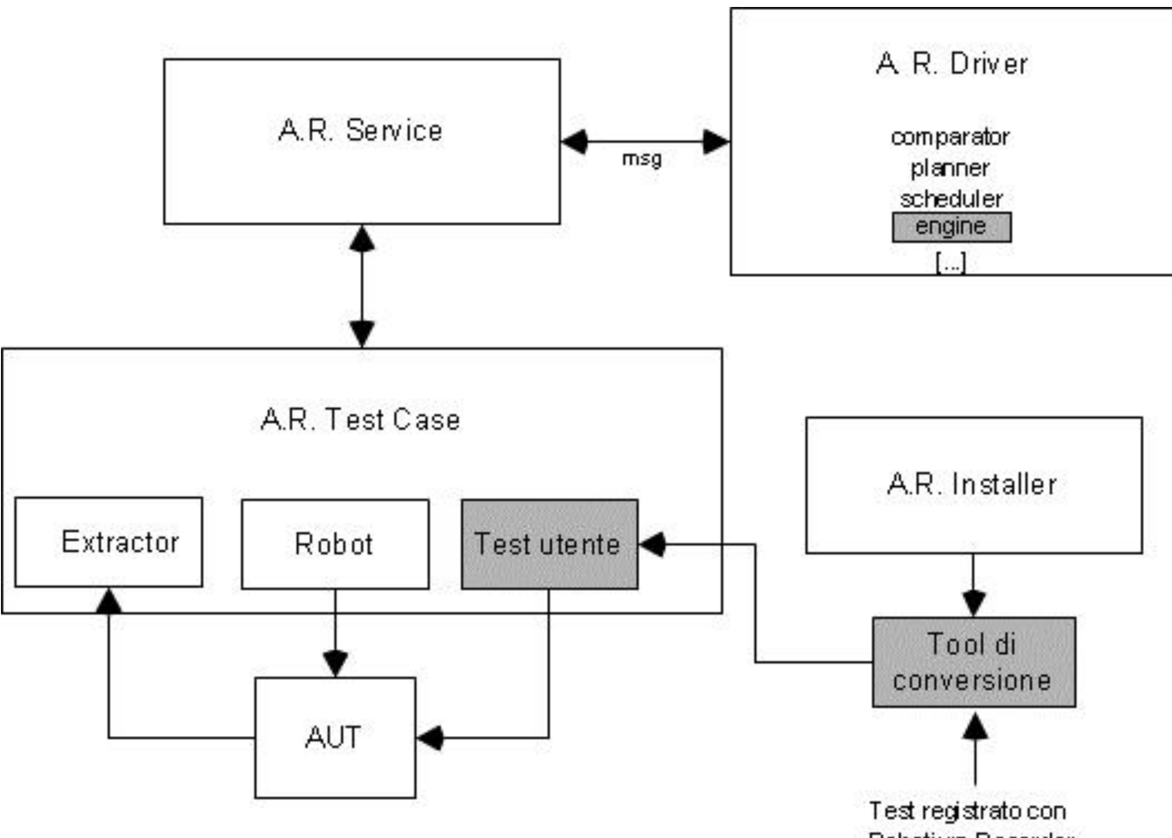


Figura 4.2: Componenti del Ripper Ibrido

In particolare notiamo in figura il tool di conversione che ha come input il test case registrato con Robotium Recorder e un componente deputato all'esecuzione di tale test

dopo opportune elaborazioni (Test utente nella figura). Il tool di conversione realizza tali elaborazioni; esso è richiamato dal componente “Installer” durante la procedura di installazione dello stesso Ripper. Fornisce in output una classe con determinate caratteristiche, la quale sarà richiamata dal componente “Test Case” attraverso particolari messaggi creati a tale scopo, in una ben precisa fase del processo di funzionamento, come illustrato in seguito.

Benché i componenti del Ripper Ibrido non differiscano dal Ripper originale, il processo di funzionamento, gestito dal “motore” del software (engine), risulta essere totalmente differente e notevolmente più complicato rispetto a quest’ultimo.

4.2.2 Processo di funzionamento

Il processo di funzionamento globale è sintetizzato nella figura seguente:

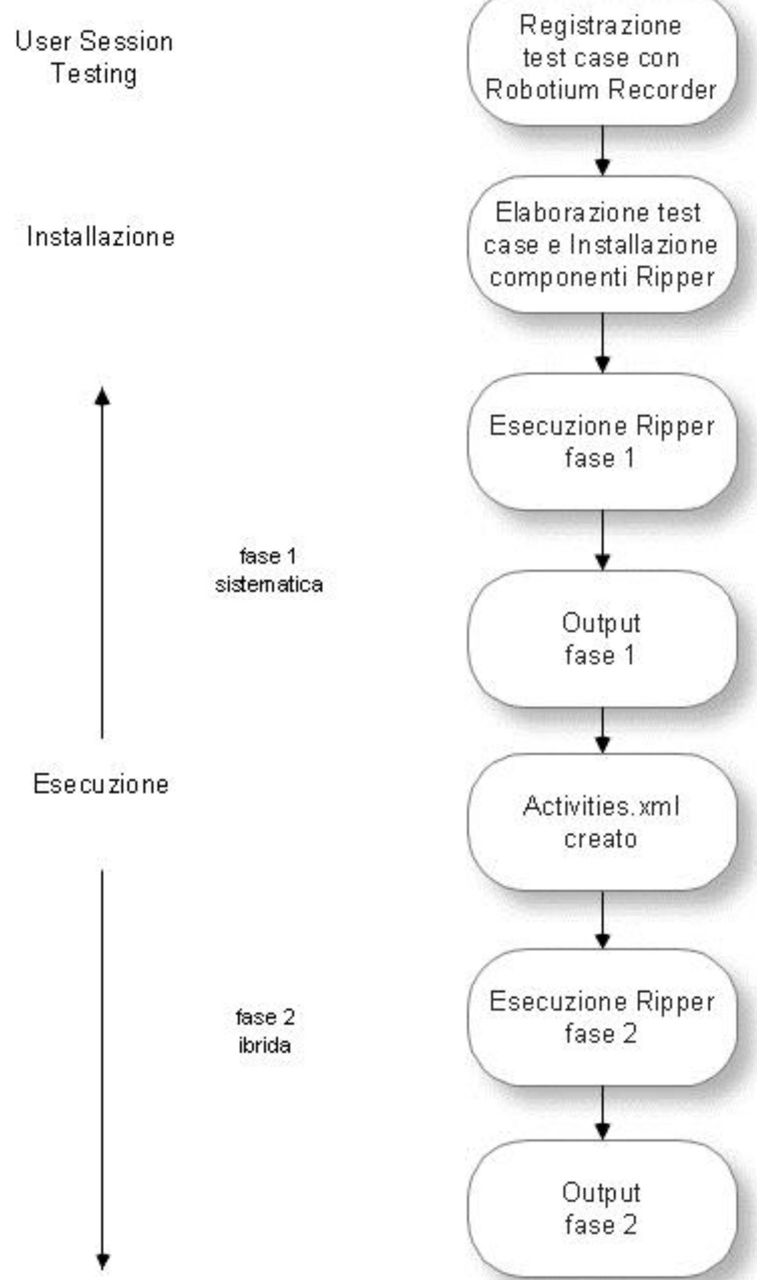


Figura 4.3: Processo di funzionamento Ripper Ibrido

In figura possiamo identificare tre sezioni. La prima sezione riguarda la registrazione del test case con Robotium Recorder; è l'unica sezione che richiede il supporto dell'utente, le altre saranno svolte in maniera automatica.

La seconda sezione riguarda l'installazione e l'elaborazione automatica della classe di test secondo una procedura discussa in seguito.

Le terza sezione racchiude il funzionamento del Ripper Ibrido, diviso in due fasi.

La prima fase corrisponde all'esecuzione del Ripper originale e consente di effettuare una prima esplorazione dell'applicazione, con conseguente collezione degli stati trovati. Usando la terminologia introdotta nel capitolo 3, uno stato è un'activity description, una descrizione dell'istanza di interfaccia corrente. In particolare uno degli output di tale fase è il file "Activities.xml" che contiene appunto gli stati e rappresenta l'elemento in base al quale sono diversificate le due fasi.

Nella prima fase l'esplorazione è totalmente sistematica, a differenza della seconda, il cuore della tecnica ibrida. Proprio in quest'ultima fase è previsto l'utilizzo del test registrato. La classe di test non è utilizzata direttamente ma deve essere elaborata in modo da essere eseguito in maniera congiunta al Ripper. La registrazione rappresenta un unico test case ottenuto attraverso l'interazione con la GUI dell'applicazione. E' pertanto composta da tutti gli eventi e input effettuati sui singoli widget. Come visto nel capitolo 3, si parla di "evento" quando l'interazione causa un cambiamento di stato; in caso contrario si parla semplicemente di "input".

Dopo aver analizzato tutte le possibili interazioni con i widget previsti dal framework Robotium (come ad esempio click su menu, pressione tasto back, click in caselle di testo) sono stati selezionati tra queste gli eventi, elencati successivamente. Il singolo test case registrato, pertanto, è una collezione di statement corrispondenti ad input o a eventi. Gli statement considerati come eventi saranno presi in considerazione per suddividere automaticamente il singolo test case iniziale in un insieme di test case. Si individuano pertanto nel test case tante sezioni o "tagli" quanti sono gli statement corrispondenti ai soli eventi. La struttura dei test case risultanti verrà analizzata nei paragrafi successivi; l'importante è sottolineare che la suddivisione nasce dalla necessità di riconsiderare le sequenze più o meno lunghe di test generate dagli utenti, come composte da tanti test aventi una precondizione: racchiudere tutti gli statement precedenti al taglio considerato a cui si aggiunge come statement finale un evento (statement di taglio). In base a questa caratteristica, l'esecuzione di ogni test case conduce sicuramente a uno stato a partire dal quale si valuta l'eventuale copertura. Solo quando lo stato in cui si è giunti non è

equivalente a nessuno degli stati già scoperti, l'idea è di avviare una nuova esplorazione sistematica, ma a partire da quello specifico stato trovato, incrementando in tal caso la copertura rispetto a quella già ottenuta e potenzialmente scoprendo automaticamente nuovi stati.

La componente manuale viene pertanto sfruttata per raggiungere porzioni dell'applicazione inesplorate (sfruttando uno dei vantaggi della tecnica manuale) lasciando poi il campo all'esplorazione metodica (vantaggio questo della ML), per eseguire un'attività di test dell'applicazione più approfondita rispetto a quella ottenibile dalle tecniche manuale e sistematica considerate singolarmente.

La descrizione di dettaglio del processo di generazione e modifica della classe di test, della creazione del codice, della descrizione delle due fasi di esecuzione e quindi dell'intera realizzazione del Ripper Ibrido saranno presentate nei paragrafi seguenti di questo capitolo.

4.3 Ambiente di sviluppo

Per lo sviluppo del software è stato utilizzato Eclipse Luna (https://projects.eclipse.org/releases/luna) con Android Development Toolkit versione 23.0.4.1468518.

Per le registrazioni di test utente, Robotium Recorder versione 2.1.25

Per la verifica del funzionamento del Ripper Ibrido, il sistema è stato settato utilizzando le impostazioni riportate nella sezione "HowTo - Guida all'installazione e uso del Ripper Ibrido".

Il software Android Ripper originale è stato prelevato dal seguente link: https://github.com/reverse-unina/AndroidRipper

Nell'archivio sono presenti due sezioni, i file sorgenti e di release. I sorgenti sono divisi in quattro progetti:

- AndroidRipper
- AndroidRipperInstaller

- AndroidRipperNewDriver
- AndroidRipperService

Le classi interessate dalle modifiche per la versione ibrida sono le seguenti:

- AndroidRipper
    - RipperTestCase.java nel package it.unina.android.ripper
- AndroidRipperInstaller
    - GuiRipperInstaller.java nel package it.unina.android.ripper.installer.console
    - Actions.java nel package it.unina.android.ripper.autoandroidlib
- AndroidRipperNewDriver
    - Actions.java nel package it.unina.android.ripper.autoandroidlib
    - AbstractDriver.java nel package it.unina.android.ripper.driver
    - SystematicDriver.java nel package it.unina.android.ripper.driver.systematic
    - RipperServiceSocket.java nel package it.unina.android.ripper.net
    - Message.java e MessageType.java nel package it.unina.android.ripper.net della cartella "shared"

4.4 Input

Nella realizzazione del Ripper Ibrido si è scelta la linea di limitare, per quanto possibile, le operazioni aggiuntive richieste all'utente rispetto alla versione originale. Il requisito è stato soddisfatto usando gli stessi file di configurazione ("ripper.properties" e "systematic.properties") con l'aggiunta di una riga relativa al percorso del file di test utente creato attraverso l'uso dello strumento Robotium Recorder.

La precondizione da rispettare è relativa all'esecuzione di una sessione di test attraverso questo tool.

4.4.1 Robotium Recorder

Robotium Recorder è uno strumento, basato su Robotium, in grado di registrare tutte le interazioni che l'utente effettua sulla GUI dell'AUT ottenendo un test case rieseguibile e ben leggibile (per maggiore chiarezza, ciascuna azione è corredata automaticamente di

commento) [7].

Per il suo funzionamento si rimanda alla pagina ufficiale http://robotium.com/products/robotium-recorder e all'appendice A2 sezione "A2.2 Registrazione con Robotium Recorder" di questo lavoro di tesi.

L'aspetto importante da rilevare è l'output elaborato: un progetto di test separato dall'applicazione a cui fa riferimento, che prevede al suo interno una classe per ogni test case effettuato. Un esempio è il seguente:

```
package kdk.android.simplydo.test;

import kdk.android.simplydo.SimplyDoActivity;
import com.robotium.solo.*;
import android.test.ActivityInstrumentationTestCase2;


public class SimplyDoActivityTest_JK extends ActivityInstrumentationTestCase2<SimplyDoActivity> {
		private Solo solo;

	public SimplyDoActivityTest_JK() {
		super(SimplyDoActivity.class);
	}

	public void setUp() throws Exception {
		super.setUp();
		solo = new Solo(getInstrumentation());
		getActivity();
	}

	@Override
	public void tearDown() throws Exception {
		solo.finishOpenedActivities();
		super.tearDown();
	}

	public void testRun() {
		// Wait for activity: 'kdk.android.simplydo.SimplyDoActivity'
		solo.waitForActivity(kdk.android.simplydo.SimplyDoActivity.class, 2000);
		// Enter the text: 'kkk'
		solo.clearEditText((android.widget.EditText)
		solo.getView(kdk.android.simplydo.R.id.AddListEditText));
		solo.enterText((android.widget.EditText)
		solo.getView(kdk.android.simplydo.R.id.AddListEditText), "kkk");
		[…]
		}

}
```

Le sezioni da evidenziare sono:

- "package kdk.android.simplydo.test". Tipicamente Robotium Recorder suggerisce

come best practice di usare come package della classe di test quello dell'AUT con l'aggiunta del suffisso ".test"

- "extends ActivityInstrumentationTestCase2". ActivityInstrumentationTestCase2 è una classe per il testing delle activity che ha metodi protected "setUp()", "tearDown()" e "runTest ()"
- metodi "setUp()" e "tearDown()" per le fasi di inizializzazione e chiusura di ciascun test. Nel setUp() c'è l'inizializzazione dell'oggetto "Solo"
- metodo "runTest ()" deputato all'esecuzione del test

Dall'analisi della struttura del file fornito come input, si è pensato di ricavare automaticamente un file contenente le principali informazioni di quest'ultimo, modificandolo però nella struttura, per asservire allo scopo del Ripper ibrido, come descritto nel successivo paragrafo.

4.5 Tool di conversione della classe di test

La classe di test registrata con Robotium Recorder, prima di essere eseguita dal Ripper, necessita di una profonda rielaborazione. E' stato per tale scopo creato un tool di conversione (i cui sorgenti sono presenti nella directory "Tool_Conversione_RoboRecoTC") richiamato automaticamente nella fase iniziale del Ripper (tool_jk.jar sito nella directory "Release\AndroidRipperInstaller"), il quale effettua le seguenti operazioni:

- verifica la presenza della classe di test specificata dall'utente nel file "ripper.properties"
- crea a partire da essa una nuova classe di test (RobotiumTest.java)
- sposta la classe ottenuta in una directory ben precisa dell'Installer del Ripper (\Release\AndroidRipperInstaller\AndroidRipper\src\it\unina\android\ripper)

Vediamo nel dettaglio un estratto del file generato (RobotiumTest.java) a partire da quello illustrato nel precedente paragrafo:

```
package it.unina.android.ripper;
```

```
import kdk.android.simplydo.SimplyDoActivity;
import com.robotium.solo.*;
import android.test.ActivityInstrumentationTestCase2;


public class RobotiumTest extends ActivityInstrumentationTestCase2<SimplyDoActivity> {
	private Solo solo;

	public RobotiumTest() {
		super(SimplyDoActivity.class);
	}

	public void setUp() throws Exception {
		super.setUp();
		solo = new Solo(getInstrumentation());
		getActivity();
	}

	@Override
	public void tearDown() throws Exception {
		solo.finishOpenedActivities();
		super.tearDown();
	}

	//il numero tc = al numero tagli; da restituire al driver
	public static final int num_tc = 45;

	public void testRun1(Solo solo){
		this.solo = solo;
		// Wait for activity: 'kdk.android.simplydo.SimplyDoActivity'
		solo.waitForActivity(kdk.android.simplydo.SimplyDoActivity.class, 2000);
		solo.sleep(1000);
	}

	public void testRun2(Solo solo){
		this.solo = solo;
		// Wait for activity: 'kdk.android.simplydo.SimplyDoActivity'
		solo.waitForActivity(kdk.android.simplydo.SimplyDoActivity.class, 2000);
		// Enter the text: 'kkk'
		solo.clearEditText((android.widget.EditText)
		solo.getView(kdk.android.simplydo.R.id.AddListEditText));
		solo.enterText((android.widget.EditText)
		solo.getView(kdk.android.simplydo.R.id.AddListEditText), "kkk");
		// Click on Add
		solo.clickOnView(solo.getView(kdk.android.simplydo.R.id.AddListButton));
		solo.sleep(1000);
	}
	[…]
	public void testRun45(Solo solo){
	[…]
	}

}
```

La prima differenza riguarda il package, modificato per funzionare con il Ripper e per garantire la corretta esecuzione dei metodi tramite reflection. Il costruttore, i metodi “setUp()” e “tearDown()” restano invariati. Il contenuto del metodo testRun() presente in

precedenza viene ora suddiviso in diverse sezioni. Il numero di sezioni o “tagli” è variabile rispetto a ciascun test case; il valore è salvato in una variabile di tipo intero, nell’esempio proposto è pari a 45 (*public static final int num_tc = 45;*) cioè dalla registrazione fornita sono stati individuati 45 possibili sequenze di operazioni ciascuna delle quali potenzialmente potrebbe scoprire nuovi stati.

I tagli sono effettuati in corrispondenza di specifici eventi, ad esempio il click su un widget. Per stato si intende una descrizione dell’activity composta da tutti gli elementi grafici in essa presenti. La classe che effettua tale operazione è “Tool_JK.java”. Si riporta la porzione di codice di tale classe relativa all’identificazione dei tagli:

```
// se necessario  modificare pattern per taglio
if (line.contains("solo.clickOnView") ||
line.contains("solo.goBack") ||
line.contains("solo.waitForActivity") ||
line.contains("solo.clickInList")) {
//System.out.println("trovato statement taglio su linea: "+ lineNum);
          occorrenze += 1; // nuovo statement utile

  […]
```

Analizziamo le stringhe riconosciute e gli statement corrrispondenti riconosciuti come eventi in grado di determinare eventuali cambiamenti di stato nella seguente tabella:

Stringhe riconosciute	Statement corrispondenti http://robotium.googlecode.com/svn/doc/com/robotium/solo/Solo.html	Azione corrispondente
solo.clickOnView	clickOnView(android.view.View view) clickOnView(android.view.View view, boolean immediately)	Clicks the specified View. Clicks the specified View.
solo.goBack	goBack() goBackToActivity(String name)	Simulates pressing the hardware back key. Returns to an Activity

		matching the specified name.
solo.waitForActivity	waitForActivity(Class<? extends android.app.Activity> activityClass) waitForActivity(Class<? extends android.app.Activity> activityClass, int timeout) waitForActivity(String name) waitForActivity(String name, int timeout)	Waits for an Activity matching the specified class. Waits for an Activity matching the specified class. Waits for an Activity matching the specified class.
solo.clickInList	clickInList(int line) clickInList(int line, int index)	Clicks the specified list line and returns an ArrayList of the TextView objects that the list line is displaying. Clicks the specified list line in the ListView matching the specified index and returns an ArrayList of the TextView objects that the list line is displaying.

Una volta identificati i tagli, si procede alla composizione dei metodi testRun. Ciascun metodo ha la seguente struttura:

- nome: "testRunX(Solo solo)" dove X è un numero progressivo da 1 al numero di tagli. Il parametro "solo" di tipo "Solo" è necessario per l'invocazione del metodo

stesso tramite reflection

- corpo del metodo: testRunX ha come corpo tutte le istruzioni dall'inizio del metodo testRun originale fino all'istruzione identificata come taglio X

Dalla registrazione originale racchiusa in un solo test case nel metodo "testRun()" si passa pertanto alla creazione di una molteplicità di test case, ciascuno dei quali comprende i passi del precedente, a cui aggiunge uno o più statement.

Si noti la presenza dell'istruzione "solo.sleep(1000)" al termine di ciascun metodo "testRunX" (e assente nel metodo originario "testRun()") necessario al Ripper per la corretta estrazione dell'activity description dopo l'esecuzione delle operazioni. Il valore "1000" rappresentano i millisecondi di attesa. Tale valore può essere modificato a seconda della complessità della GUI dell'applicazione da testare; nel caso questa sia particolarmente ricca potrebbe essere necessario un aumento di tale valore. Negli esperimenti condotti su alcune app tale valore è stato impostato tra i 500ms e i 1000ms. Il codice corrispondente a tale modifica nella classe Tool_JK.java è il seguente

```
scrivo_stringa_a_file("\t"+"\t"+"solo.sleep(1000);"+"\n"+"\t"+"}" +"\n\n");
```

4.6 Funzionamento

Il Ripper originale prevede una singola esecuzione, al termine della quale occorre salvare i dati di output prodotti. Il Ripper Ibrido prevede ora una esecuzione singola ma composta da due fasi distinte, senza sovrascrittura dei file prodotti tra una fase e l'altra, sempre nel rispetto del requisito di non richiedere alcuna operazione da parte dell'utilizzatore del software.

La discriminante tra le due fasi è la presenza del file "Activities.xml". Nel caso questo sia presente nella cartella di output "model", il Ripper comprende che si tratta della seconda fase, scatenando una serie di operazioni aggiuntive rispetto a quelle eseguite nella prima fase.

Vediamo nel dettaglio le due fasi:

1. 1° fase. Prevede la procedura di installazione attraverso il comando "*java -jar*

AndroidRipperInstaller.jar“, cui segue l’elaborazione vera e propria attraverso il comando “*java -jar AndroidRipper.jar s systematic.properties*”. Questa prima fase è esattamente la stessa del Ripper originale consentendo pertanto di generare gli stessi output (i file di coverage, log, e Activities.xml) modificati nel nome per evitare sovrascritture. Questa prima fase consente di esplorare automaticamente la GUI dell’applicazione, collezionando una serie di activity description nel file “Activities.xml”. L’algoritmo di esecuzione esamina di volta in volta gli stati scoperti. Per ciascuno stato nuovo verrà generata una task list di eventi che andranno a sollecitare lo stato stesso. Il processo terminerà quando saranno esaminate tutte le activity scoperte e la task list sarà vuota:

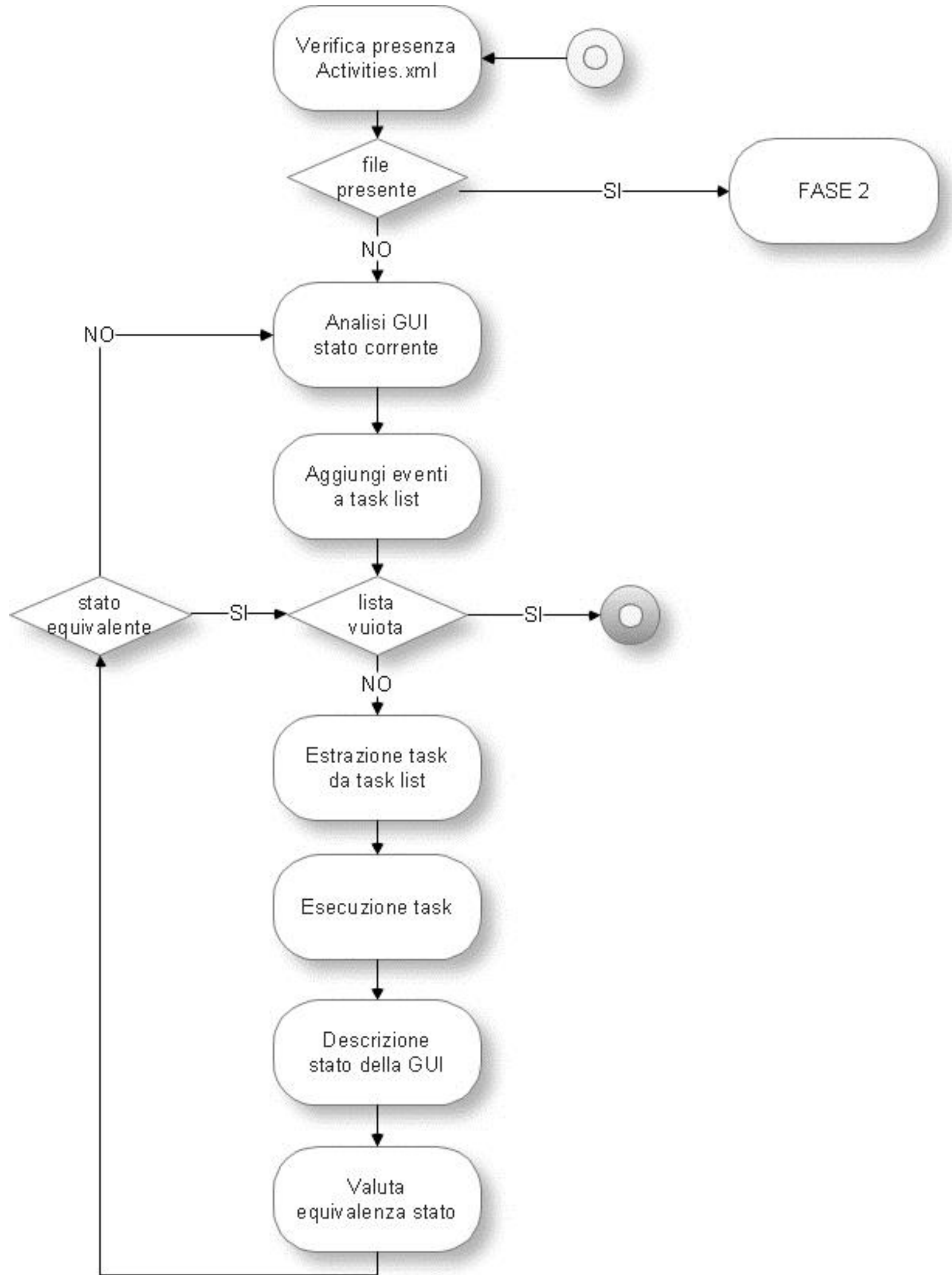


Figura 4.4: Activity Diagram Ripper Ibrido, 1° fase

Il diagramma illustra come la prima fase abbia esattamente le stesse attività del Ripper originale se si esclude il controllo dell'esistenza di Activities.xml. Questo risponde a un altro requisito di sviluppo, rispettare cioè la sequenza di passi di base del Ripper originale, producendo gli stessi output (come vedremo in seguito, gli

output sono praticamente identici tranne nella denominazione degli stessi). La macroattività "FASE 2" riguarda l'activity diagram relativa appunto alla seconda fase, discussa nel seguito.

2. 2° fase. Al termine della fase 1, otteniamo in output gli stessi file del Ripper originale. Per ottenere tale risultato sono state settate delle condizioni booleane e inseriti statement di controllo in più parti del codice.

   La fase 2 vede una nuova esecuzione del Driver attraverso lo stesso comando "*java -jar AndroidRipper.jar s systematic.properties*". Sebbene il comando sia il medesimo della fase 1, il comportamento atteso è differente. In questo caso, infatti, il Ripper vede l'esistenza del file "Activities.xml" creato nella prima fase ed effettua una serie di operazioni totalmente differente. L'activity diagram seguente sintetizza queste operazioni;

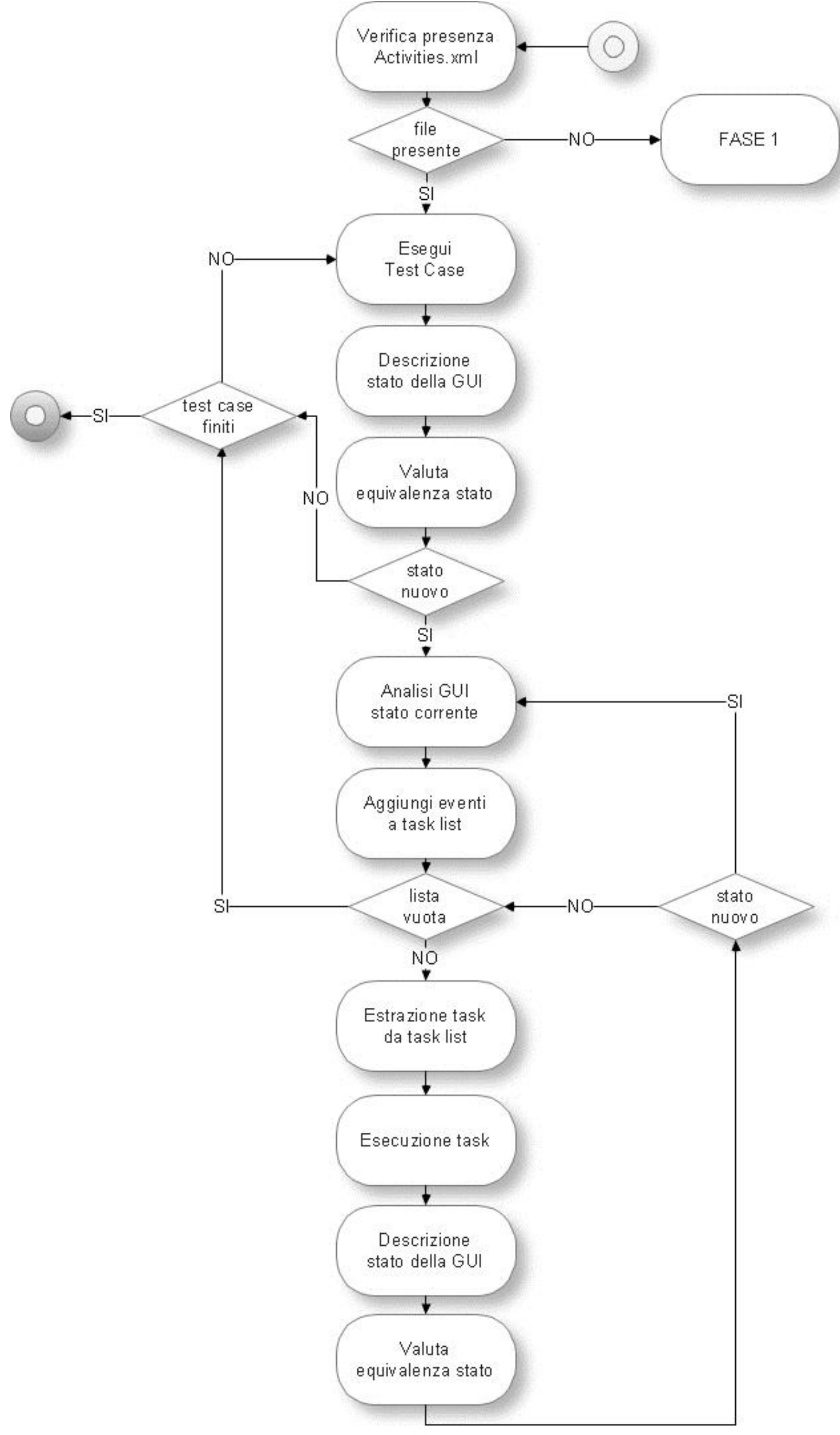


Figura 4.5: Activity Diagram Ripper Ibrido, 2° fase

La macro-attività "FASE1" rappresenta l'activity diagram il cui esploso è stato presentato in precedenza.

Nel diagramma è possibile identificare due cicli innestati. Il ciclo esterno governa l'esecuzione dei test case e ha come criterio di terminazione il raggiungimento del numero totale di questi, previsto dalla registrazione. Il ciclo interno ha come criterio di terminazione la task list vuota come visto nella fase 1, in quanto rappresenta il processo di esplorazione del Ripper originale, innescato tante volte quanti sono gli stati nuovi scoperti dai metodi della classe di test.

Analizziamo nel dettaglio le attività del processo di ripping, riprendendo il diagramma comprendente le componenti logiche del Ripper presentato in precedenza:

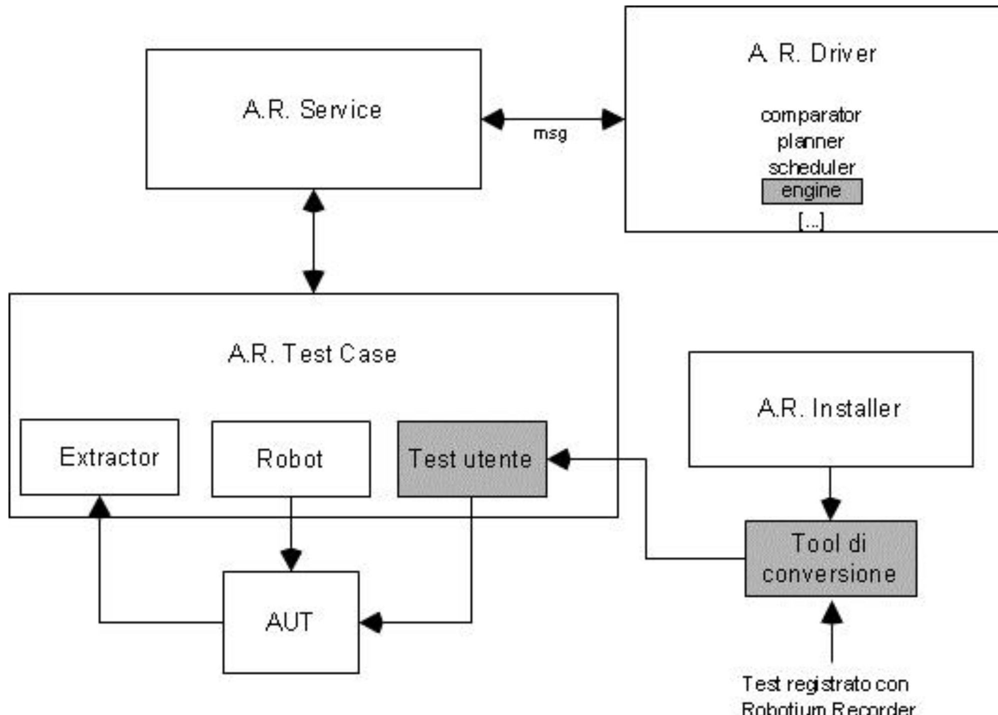


Figura 4.6: Componenti del Ripper Ibrido

Nella seconda fase, il "Driver" invia un messaggio, per conoscere il numero totale di test case utente che rappresenterà il criterio di terminazione del ciclo esterno e quindi della intera fase 2 (terminati i test case, l'esecuzione terminerà). Fin quando tale numero non è raggiunto, il driver invia un messaggio, appositamente creato per

l'esecuzione dei test case utente, al componente "Test Case", il quale, esegue le operazioni e invia in risposta l'activity description sempre incapsulata in un messaggio.

Il Driver, ottenuta l'activity description, la confronta con tutte quelle prelevate dal file "Activities.xml". Nel caso sia già presente, si passa direttamente al successivo test, altrimenti viene eseguito il processo di ripping ma a partire dal nuovo stato scoperto.

Supponiamo di eseguire il test case i-esimo. Al termine dell'esecuzione viene estratta l'activity description corrispondente e confrontata con quelle già presenti. Consideriamo nel dettaglio i due casi:

- stato equivalente. Nel caso in cui lo stato estratto a seguito dell'i-esimo test case risulta essere equivalente a uno degli stati già visitati, il ciclo interno non viene eseguito e si considera direttamente il test case successivo
- stato non equivalente. L'activity description viene salvata nel file Activities.xml ed inizia nuovamente il processo di ripping ma a partire dallo stato nuovo scoperto. L'applicazione sarà forzata in questo stato dal driver, previo scambio di un ulteriore messaggio, prima della pianificazione dei task e della conseguente esecuzione degli stessi. E' possibile che, a partire dallo stato considerato, il Ripper scopra nuovi stati in maniera automatica. Terminato il processo di ripping, si passerà al test case successivo

La classe che racchiude il principio di funzionamento è "SystematicDriver.java", profondamente modificata nel Ripper Ibrido. La figura seguente illustra la composizione di tale classe e le relazioni con alcune delle classi con cui interagisce:

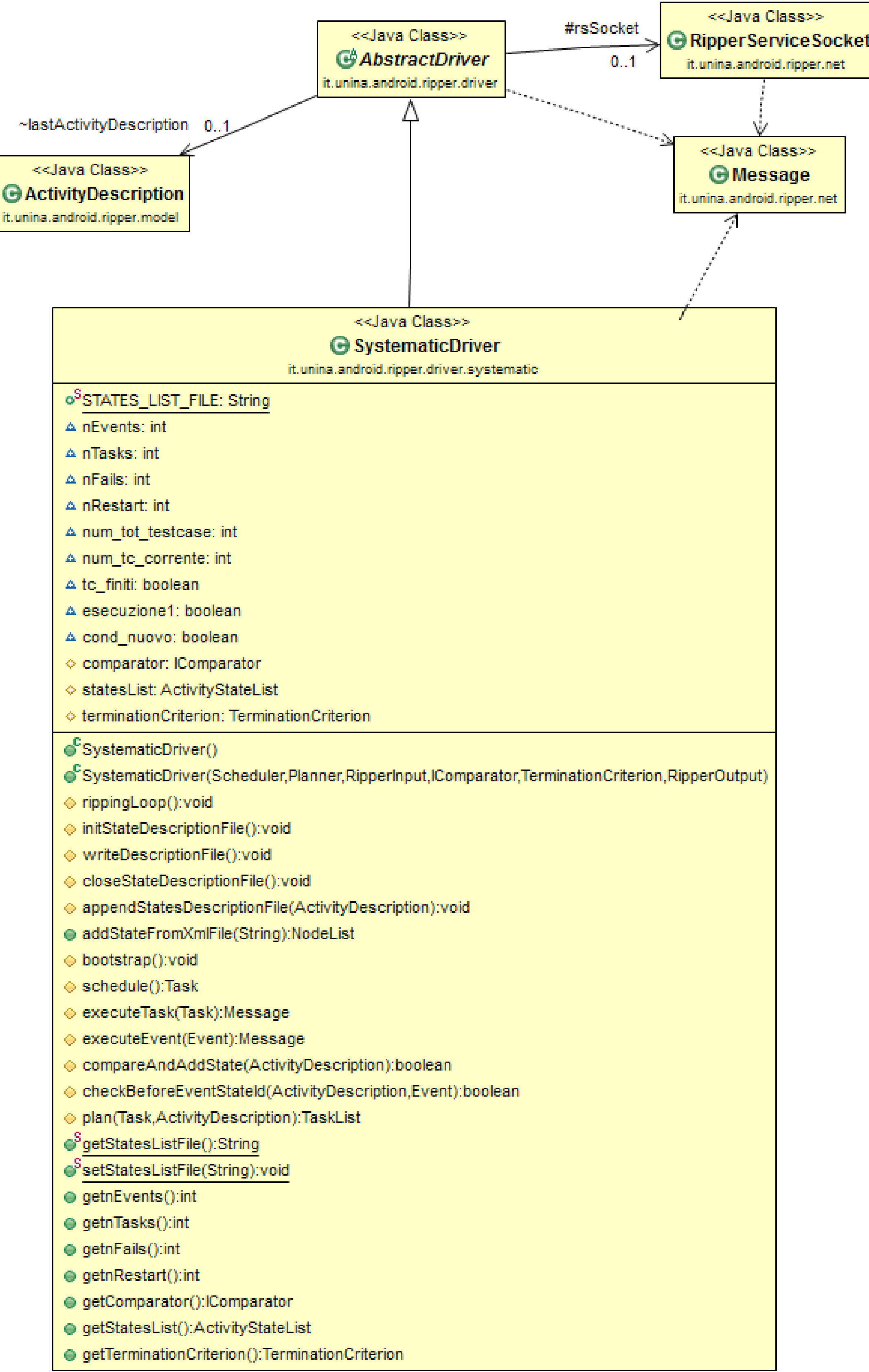


Figura 4.7: Class Diagram SystematicDriver

In particolare nel metodo "*protected void rippingLoop()*" due cicli "do-while" innestati racchiudono il processo di ripping:

```
// ciclo su controllo test_case finiti
do {
      // loop ripper
       do {
       [. . .]
           } while (running && this.terminationCriterion.check() == false);
                // condizioni di uscita ripper
                closeStateDescriptionFile();
                //controllo su raggiungimento num totale di test case
                if (num_tc_corrente < num_tot_testcase && esecuzione1 == false) {
                      bootstrap = false; // devo rientrare nel bootstrap perchè
                                        // num_tc_corrente<num_tot_testcase
                      System.out.println("Test utente non terminati");
                      } else {
                            tc_finiti = false; //test utente terminati
                            System.out.println("Test utente terminati");
                            break; // esci dal loop; testcase finiti
                      }
       } while (tc_finiti == true); // loop fin quanto i test case non sono
       terminati
```

Il ciclo interno (*while (running && this.terminationCriterion.check() == false);*) ha come condizione di terminazione la task list vuota dopo aver esaminato tutti gli stati trovati, come visto nella fase 1.

Il ciclo esterno (*while (tc_finiti == true);*) ha come criterio di terminazione il raggiungimento del numero dei test previsti, basato sulla variabile di controllo booleana "tc_finiti", impostata inizialmente a "true". Sarà impostata a "false", con conseguente uscita dal ciclo, quando è stato raggiunto il numero totale di test case oppure nel caso di fase 1 (quest'ultima condizione è identificata nel codice con la variabile booleana "esecuzione1" impostata a "true").

4.6.1 Messaggi

Come visto nel paragrafo precedente, il processo di ripping è governato da uno scambio di messaggi tra componente Driver e componente Test Case. In particolare, la figura seguente illustra tale scambio relativamente alla seconda fase del Ripper, per la quale sono stati creati tre nuovi messaggi oltre a quelli esistenti:

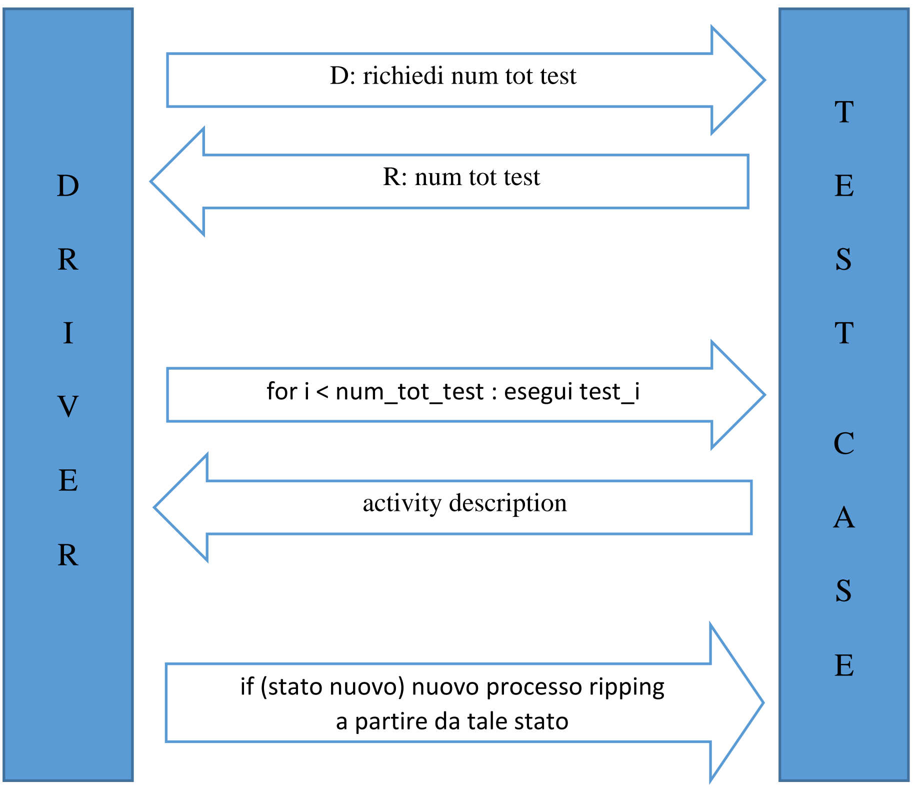


Figura 4.8: scambio messaggi tra componenti del Ripper Ibrido, fase 2

Nello specifico, nella classe "MessageType.java" abbiamo la stringa identificativa dei tre messaggi, rispettivamente per eseguire il test case i-esimo, per conoscere il numero totale di test case e infine per forzare l'applicazione all'eventuale stato nuovo trovato.

```
public static final String TEST_UTENTE_MESSAGE = "TEST";
public static final String NUM_TOT_TEST_CASE = "NUMTC";
public static final String ESEGUI_TEST_CASE_PER_DRIVER = "ETC";
```

I messaggi non sono altro che degli HashMap contenenti delle stringhe, difatti la classe "Message.java" estende la classe HashMap:

```
public class Message extends HashMap<String, String>{
[. . . ]

// messaggio per exec test case
public static Message getRunMessage(String i)
{
```

```
		Message msg = new Message(MessageType.TEST_UTENTE_MESSAGE);

		msg.addParameter("test", i);
		return msg;
	}
	// messaggio per num tot test case
	public static Message getNumTestCaseMessage()
	{
	return new Message(MessageType.NUM_TOT_TEST_CASE);
	}
	// messaggio per forzare stato
	public static Message getTestCasePerDriver(String string)
	{
		Message msg = new Message(MessageType.ESEGUI_TEST_CASE_PER_DRIVER);
		msg.addParameter("runner", string);
		return msg;
	}
	// messaggio per forzare stato
	public static Message getTestCasePerDriver()
	{
		Message msg = new Message(MessageType.ESEGUI_TEST_CASE_PER_DRIVER);
		return new Message(MessageType.ESEGUI_TEST_CASE_PER_DRIVER);
		}
		[. . .]
}
```

I valori restituiti, ossia il numero totale di test case e l'activity description sono anch'essi incapsulati in messaggi inviati dal componente Test Case e letti dal Driver. La classe che regola lo scambio nel Driver (invio - ricezione) è "RipperServiceSocket.java" dal quale sono estratti i seguenti metodi:

```
	//Invia messaggio per esecuzione test case i-esimo
	public String runTestCaseUtente(int MAX_RETRY,int i) throws
	SocketException
	{
	Message runMSGdecr = null;
	this.sendMessage(Message.getRunMessage(Integer.toString(i)));
	int describeCnt = 0;
	do {
		//ricezione messaggio
		runMSGdecr = this.readMessage_utente(true);
		if (runMSGdecr != null &&
		runMSGdecr.getType().equals(MessageType.DESCRIBE_MESSAGE) == false)
		{
			System.out.println("Message != runMSGdecr -> " +
			runMSGdecr.getType());
			continue;
		}
		if (runMSGdecr != null && runMSGdecr.getParameterValue("wait") !=
		null)
		{
			try { Thread.sleep(1000); } catch(Throwable tr) {}
```

```
                this.sendMessage(Message.getRunMessage(Integer.toString(i)));
                continue;
            }
            else if (runMSGdecr != null && runMSGdecr.getParameterValue("xml")
            != null)
            {
                String xml = runMSGdecr.getParameterValue("xml");
                if (xml != null && xml.length() > 45)
                {
                    return xml;
                }
                this.sendMessage(Message.getRunMessage(Integer.toString(i)));
                if (xml != null)
                    System.out.println("xml != null" +xml);
            }
            else
            {
                try { Thread.sleep(1000); } catch(Throwable tr) {}
                if (describeCnt++ > MAX_RETRY)
                    throw new RuntimeException("JK describeCnt overflow");
                //System.out.println("Describe retry " + describeCnt);
                //this.sendMessage(Message.getDescribeMessage());
                continue;
            }
        } while(true);

        }


        // richiede numero totale di test case da eseguire
        public String richiediNumTestCase() throws SocketException
        {
            Message NumTestCaseMSG = null;
            this.sendMessage(Message.getNumTestCaseMessage());
            // invio messaggio richiesta numero testCase

            String numero_test_case_rx = null;
            NumTestCaseMSG = this.readMessage(1000, true);
            // leggo numero di test case previsti
            System.out.println("Lettura Message richiediNumTestCase -> " +
            NumTestCaseMSG );
                if (NumTestCaseMSG != null &&
                NumTestCaseMSG.getParameterValue("num") != null)
                {
                    numero_test_case_rx =
                    NumTestCaseMSG.getParameterValue("num");

                }
                System.out.println("Valore numero_test_case ricevuto -> " +
                numero_test_case_rx );
                return numero_test_case_rx;
        }


        /*Invio messsaggio per forzare stato iniziale del ripper;
        Il test case i-esimo che ha scoperto un nuovo stato diviene punto di
        partenza del ripper*/
        public String eseguiTestCase_perDriver(int i) throws SocketException
```

```
{
Message eseguiTestCaseMSG = null;
this.sendMessage(Message.getTestCasePerDriver(Integer.toString(i)));
do{
       String esegui_test_case = null;
              //System.out.println("RSS/MessageSend
              eseguiTestCase_perDriver -> " +
              Message.getTestCasePerDriver(Integer.toString(i)));

              eseguiTestCaseMSG = this.readMessage(1000, false);
              // attende che venga eseguito test case
              if (eseguiTestCaseMSG != null &&
              eseguiTestCaseMSG.getParameterValue("runner") != null)
              {
                     esegui_test_case =
                     eseguiTestCaseMSG.getParameterValue("runner");
                     //System.out.println("RSS/MessageRead
                     eseguiTestCase_perDriver -> " + eseguiTestCaseMSG);
                     //System.out.println("RSS/MessageRead
                     eseguiTestCase_perDriver parametro-value -> " +
                     esegui_test_case);
                     return esegui_test_case;
              }else{
                     try { Thread.sleep(500); } catch(Throwable tr) {}
                     //System.out.println("RSS/MessageRead
                     eseguiTestCase_perDriver Else -> ");
                     continue;
              }
}while(true);

}
```

La componente Test Case resta in attesa dei messaggi. A seconda della tipologia ricevuta (Message Type) esegue specifiche operazioni. A titolo di esempio si riporta un estratto del codice della classe "RipperTestCase.java" relativo a due tipi di messaggi (TEST_UTENTE_MESSAGE e DESCRIBE_MESSAGE):

```
// Messaggio ricevuto dal driver per eseguire i test case utente;
// fornisce al driver l'activity description ottenuta al termine
dell'esecuzione // del test case
else if (msg.isTypeOf(MessageType.TEST_UTENTE_MESSAGE))
{
Log.v(TAG, "MSG_TEST_UTENTE_MESSAGE_ricevuto : " +
msg.getParameterValue("test")) ;

       String parametroRx=msg.getParameterValue("test");
       Log.v(TAG, "Parametro_rx : " + parametroRx) ;
       // indice del test case da eseguire
       //int valore_parametro_test = Integer.parseInt(parametroRx);
       // oggetto "solo" passato come argomento ai metodi della classe di
       test
       solo = new Solo(getInstrumentation(),getActivity());
       try {
```

```
            Class<?> cls =
            Class.forName("it.unina.android.ripper.RobotiumTest");
            Object obj = cls.newInstance();
            String nome_metodo="testRun"+parametroRx;
            System.out.println("Eseguo : "+nome_metodo);
            Method mtdS = cls.getMethod(nome_metodo, solo.getClass());
            mtdS.setAccessible(true);
            mtdS.invoke(obj,solo);
    [. . .]

    else if (msg.isTypeOf(MessageType.DESCRIBE_MESSAGE))
    {
    try
    {
            String processName = getForegroundApp2();
            Log.v(TAG, "DSC : " + processName);
            if(processName.equals(Configuration.PACKAGE_NAME) == false)
            {
                    Log.v(TAG, "DSC : wait process name");
                    Message retMsg = Message.getDescribeMessage();
                    retMsg.addParameter("wait","wait");
                    retMsg.addParameter("index", msg.get("index"));
                    mService.send(message);
            }
            else
            {
                    Activity activity = getActivity();
                    if (activity != null)
                    {
                            OutputAbstract o = new XMLOutput();
                            ActivityDescription ad = extractor.extract();
                            o.addActivityDescription( ad );
                            Message retMsg = Message.getDescribeMessage();
                            retMsg.addParameter("index", msg.get("index"));
                            String s = o.output();
                            retMsg.addParameter("xml", s);
                            mService.send(retMsg);
                    }
                    else
                    {
                    Log.v(TAG, "DSC : wait activity null");
                    Message retMsg = Message.getDescribeMessage();
                    retMsg.addParameter("wait","wait");
                    retMsg.addParameter("index", msg.get("index"));
                    mService.send(message);
                    }
    [. . .]
```

Si noti nel codice il riferimento alla classe "RobotiumTest" (`it.unina.android.ripper.RobotiumTest`) che rappresenta la classe ottenuta a partire dal test case utente registrata con Robotium Recorder. Si accede ai suoi metodi tramite reflection.

Il riconoscimento del tipo di messaggio DESCRIBE innesca invece l'estrazione

dell'activity description corrente (`ActivityDescription ad = extractor.extract();`) e il suo invio tramite messaggio alla componente Driver.

4.7 Output

Analizziamo ora l'output prodotto nelle due fasi. L'AUT è SimplyDo, una semplice applicazione il cui codice è disponibile in rete.

4.7.1 Output fase 1

Requisito:

Gli output della prima fase del Ripper ibrido devono essere equivalenti (in termini di struttura, numero file, contenuto informativo) a quelli del Ripper originale.

Analisi struttura output:

<table>
<tr><th colspan="2">Ripper originale</th><th colspan="2">Ripper Ibrido Fase 1</th><th>Esito</th></tr>
<tr><td>Elenco del percorso delle cartelle</td><td>Numero file</td><td>Elenco del percorso delle cartelle</td><td>Numero file</td><td></td></tr>
<tr><td>|
current_ActivityStateList.bin
| current_TaskList.bin
|</td><td>2</td><td>|
current_ActivityStateList.bin
| current_TaskList.bin
|</td><td>2</td><td></td></tr>
<tr><td>+---coverage
| coverage00000_ec.ec
| coverage00001.ec
| coverage00001_ec.ec
| […]
| coverage00033.ec
| coverage00033_ec.ec
|</td><td>67</td><td>+---coverage
| coverage00000_ec.ec
| coverage00001.ec
| coverage00001_ec.ec
| […]
| coverage00033.ec
| coverage00033_ec.ec
|</td><td>67</td><td></td></tr>
<tr><td>+---junit
| junit-log-00000.xml
| junit-log-00001.xml
| […]
| junit-log-00033.xml
|</td><td>34</td><td>+---junit
| junit-log-00000.xml
| junit-log-00001.xml
| […]
| junit-log-00033.xml
|</td><td>34</td><td></td></tr>
<tr><td>+---logcat
| logcat_5554_0.txt
| logcat_5554_1.txt
| […]
| logcat_5554_33.txt
|</td><td>34</td><td>+---logcat
| logcat_5554_0.txt
| logcat_5554_1.txt
| […]
| logcat_5554_33.txt
|</td><td>34</td><td></td></tr>
<tr><td>+---model
| activities.xml
| log_0.xml
| log_1.xml
| […]
| log_34.xml</td><td>35+1</td><td>+---model
| activities.xml
| log_test_0_0.xml
| log_test_0_1.xml
| […]
| log_test_0_34.xml</td><td>35+1</td><td></td></tr>
</table>

Contenuto informativo output:

File	Tipo confronto	Risultato	Esito
bin	Confronto non rilevante		
Coverage	Sono stati generati i due report di Emma usando i file di coverage dei due Ripper. Di seguito sono riportati i report	I due report presentano le stesse percentuali di copertura	✓
Junit	E' stato usato un file batch per il controllo delle differenze di tutti i file di log junit	I log differiscono solo per il campo "time" e "timestamp"	✓
Logcat	Confronto non rilevante		
Model: activities.xml log.xml	I file activities.xml e tutti i log.xml sono stati confrontati usando un file batch per il controllo delle differenze	I file differiscono solo per gli id assegnati ai widgets dal Ripper	✓

Esempio di confronto tra report di Emma relativi ai due Ripper (nessuna differenza)

File report Ripper Ibrido:	File report Ripper Originale:	Esito
[EMMA v2.0.5312 report, generated Sun Mar 22 18:42:21 CET 2015] --- OVERALL COVERAGE SUMMARY: [class, %] 75% (42/56)! [method, %] 41% (105/256)! [block, %] 34% (1915/5553)! [line, %] 32% (414,5/1291)! [name] all classes OVERALL STATS SUMMARY: total packages: 1 total classes: 56	[EMMA v2.0.5312 report, generated Sun Mar 22 18:41:25 CET 2015] --- OVERALL COVERAGE SUMMARY: [class, %] 75% (42/56)! [method, %] 41% (105/256)! [block, %] 34% (1915/5553)! [line, %] 32% (414,5/1291)! [name] all classes OVERALL STATS SUMMARY: total packages: 1 total classes: 56	✓

total methods: 256 total executable files: 17 total executable lines: 1291 COVERAGE BREAKDOWN BY PACKAGE: [class, %] 75% (42/56)! [method, %] 41% (105/256)! [block, %] 34% (1915/5553)!	total methods: 256 total executable files: 17 total executable lines: 1291 COVERAGE BREAKDOWN BY PACKAGE: [class, %] 75% (42/56)! [method, %] 41% (105/256)! [block, %] 34% (1915/5553)!	

4.7.2 Output fase 2

Verifichiamo il funzionamento del Ripper nella fase 2, caratterizzata da un insieme di stati già esplorati e dall'esecuzione dei test case. Per facilitare la lettura saranno elencati gli output relativi alla sola fase 2, benché nelle directory saranno presenti i file di ambedue le fasi.

Ripper Ibrido Fase 2	Verifica correttezza	Esito
Elenco del percorso delle cartelle | current_ActivityStateList.bin | current_TaskList.bin |	File ".bin". File intermedi di elaborazione. Non importanti ai fini dell'analisi	✔
+---coverage | coverage_test_15_1.ec | coverage_test_15_1_ec.ec | coverage_test_15_2.ec | coverage_test_15_2_ec.ec | […] | coverage_test_15_4_ec.ec | coverage_test_20_1.ec | coverage_test_20_1_ec.ec | […] | coverage_test_20_22_ec.ec | […] |	File di coverage. Correttamente elaborati per fornire la copertura	✔
+---junit | junit-log-test_15_1.xml | […] | junit-log-test_15_4.xml | junit-log-test_20_1.xml | […] |	File junit. Correttamente elaborati	✔
+---logcat | logcat_5554_Exec2_0.txt | logcat_5554_Exec2_1.txt | […] | logcat_5554_Exec2_119.txt | +---model | activities.xml	File logcat. Non importanti ai fini dell'analisi	✔

| log_test_15_0.xml | log_test_15_1.xml | […] | log_test_15_4.xml | log_test_20_0.xml | […]	File Activities.xml. Contenuto informativo parziale rispetto a fase 1. In particolare non sono riportati i widget della GUI analizzata e le relative caratteristiche File log. Non importanti ai fini dell'analisi	

Per quanto riguarda la struttura dei file, è importante osservare in primo luogo che i file di coverage, quelli contenuti nella cartella junit, logcat e di log sono generati solo a seguito della scoperta di un nuovo stato.

In secondo luogo, notiamo la struttura dei nomi usati per tali file. Prendiamo come esempio "coverage_test_15_1.ec". Il primo numero ("15" nell'esempio) è il numero di test case scatenante la scoperta del nuovo stato; il secondo numero ("1" nell'esempio) rappresenta il numero di evento schedulato automaticamente dal Ripper a partire dal nuovo stato (ad esempio "back" o "change orientation").

L'analisi della correttezza dell'output ha invece rilevato una errata elaborazione del file Activities.xml per alcune applicazioni. In particolare gli stati nuovi venivano rilevati ma il processo di estrazione della descrizione non veniva completato correttamente. Il problema è stato individuato nel tempo necessario per il prelievo di tutte le informazioni riguardanti i widget costituenti la schermata al termine di ciascun test case. Come soluzione è stato impostato un tempo di 1000 millisecondi per consentire il corretto prelievo. A seguito di questa modifica, anche il file Activities.xml risulta essere correttamente costruito. Per la maggior parte delle applicazioni testate, questo valore si è dimostrato sufficiente. Nel caso di GUI particolarmente ricche è possibile innalzare tale valore come precedentemente illustrato.

Pre-modifica	Post-modifica
<?xml version="1.0"?><states> <activity class="kdk.android.simplydo.SimplyDoActivity" id="a56" keypress="FALSE" longkeypress="FALSE" menu="TRUE" name="SimplyDoActivity" tab_activity="0" title="Simply Do" uid="1"> <listener class="SensorListener" present="FALSE"/> <listener class="OrientationListener" present="FALSE"/> <listener class="SensorEventListener" present="FALSE"/> <listener class="LocationListener" present="FALSE"/> </activity>	<?xml version="1.0"?><states> <activity class="kdk.android.simplydo.SimplyDoActivity" id="a56" keypress="FALSE" longkeypress="FALSE" menu="TRUE" name="SimplyDoActivity" tab_activity="0" title="Simply Do" uid="1"> <listener class="SensorListener" present="FALSE"/> <listener class="OrientationListener" present="FALSE"/> <listener class="SensorEventListener" present="FALSE"/> <listener class="LocationListener" present="FALSE"/> <widget ancestor_id="-1" ancestor_type="root" class="android.widget.LinearLayout" count="2" enabled="TRUE" id="-1" index="0" name="" p_id="-1" p_name=""

<table>
<tr>
<td><activity class="kdk.android.simplydo.SimplyDoActivity" id="a57" keypress="FALSE" longkeypress="FALSE" menu="TRUE" name="SimplyDoActivity" tab_activity="0" title="Simply Do" uid="9">
<listener class="SensorListener" present="FALSE"/>
<listener class="OrientationListener" present="FALSE"/>
<listener class="SensorEventListener" present="FALSE"/>
<listener class="LocationListener" present="FALSE"/></td>
<td>p_type="com.android.internal.policy.impl.PhoneWindow.DecorView" simple_type="linearLayout" visible="TRUE">
<listener class="AnimationListener" present="FALSE"/>
<listener class="OnKeyListener" present="FALSE"/>
<listener class="OnHierarchyChangeListener" present="FALSE"/>
<listener class="OnClickListener" present="FALSE"/>
<listener class="OnFocusChangeListener" present="FALSE"/>
<listener class="OnLongClickListener" present="FALSE"/>
</widget>
<widget ancestor_id="-1" ancestor_type="root" class="android.widget.FrameLayout" count="1" enabled="TRUE" id="-1" index="1" name="" p_id="-1" p_name="" p_type="android.widget.LinearLayout" simple_type="" visible="TRUE">
<listener class="AnimationListener" present="FALSE"/>
<listener class="OnKeyListener" present="FALSE"/>
<listener class="OnHierarchyChangeListener" present="FALSE"/>
<listener class="OnClickListener" present="FALSE"/>
<listener class="OnFocusChangeListener" present="FALSE"/>
<listener class="OnLongClickListener" present="FALSE"/>
</widget>
[…]
</activity>
<activity class="kdk.android.simplydo.SimplyDoActivity" id="a57" keypress="FALSE" longkeypress="FALSE" menu="TRUE" name="SimplyDoActivity" tab_activity="0" title="Simply Do" uid="9">
<listener class="SensorListener" present="FALSE"/>
<listener class="OrientationListener" present="FALSE"/>
<listener class="SensorEventListener" present="FALSE"/>
<listener class="LocationListener" present="FALSE"/></td>
</tr>
</table>

Capitolo 5: Efficacia Ripper Ibrido

Il capitolo 5 illustra in generale i vantaggi e svantaggi derivanti dall'uso del Ripper Ibrido e la sua efficienza in termini di copertura rispetto al Ripper originale, effettuando il test su applicazioni mobili reali.

5.1 Confronto Ripper Ibrido vs Ripper Originale

Il Ripper Ibrido nasce come fork del Ripper originale, opportunamente esteso. Laddove il Ripper originale prevede una singola esecuzione, il Ripper Ibrido prevede una esecuzione basata su 2 fasi distinte. La prima fase del Ripper Ibrido corrisponde alla singola esecuzione del Ripper originale; la seconda fase amplia il testing dell'AUT attraverso il supporto a sessioni utente registrate con Robotium Recorder.

Quali sono i vantaggi e gli svantaggi?

Vantaggi:

- esser sicuri di esplorare una precisa porzione dell'applicazione che potrebbe non essere coperta dal Ripper originale. Lo sforzo per la generazione dei casi di test utente è limitato
- la riesecuzione dei casi di test è automatizzata
- è possibile che gli utenti scoprano funzionamenti anomali che i progettisti non avevano previsto
- dall'analisi degli output è possibile identificare direttamente i test case che hanno determinato la scoperta di stati nuovi
- aumento generale dell'efficacia perché all'esplorazione automatica si affianca

l'esplorazione manuale dei test case

- possibilità di testare casi limite

Svantaggi:

- per ottenere esecuzioni significative è probabile che sia necessario raccogliere sessioni per un tempo prolungato
- difficoltà a rieseguire i test a seguito di un intervento di manutenzione sul software o a causa della natura della registrazione stessa. Si pensi ad applicazioni che interagiscono con la data corrente oppure all'influenza del tempo di esecuzione
- i tempi potrebbero dilatarsi in maniera considerevole in caso di scoperta di nuovi stati. Per ogni stato nuovo parte un nuovo processo di ripping, benchè questo implichi comunque una maggiore esplorazione dell'applicazione
- efficienza variabile con i test case. Con molta probabilità, per perseguire l'obiettivo dei test, sarà necessario un numero considerevole di test case, spesso con parti in comune. Si possono applicare a tal riguardo delle tecniche di minimizzazione basata sulla copertura, in cui si fissa un obiettivo di copertura, si valuta il grado di copertura di ogni caso di test e si esegue un algoritmo che estragga il più piccolo insieme di test che massimizzi la copertura.

Questo procedimento è favorito dagli output ottenuti al termine della seconda fase, che indicano direttamente quali test hanno scoperto stati nuovi.

5.2 Applicazioni selezionate per il test

Per effettuare il confronto tra il Ripper originale e il Ripper Ibrido sono state utilizzate le seguenti applicazioni, il cui codice è facilmente reperibile in rete:

- SimplyDo versione 0.9.2
- TippyTipper versione 1.2
- Trolly versione 1.4
- MunchLife versione 1.4.4
- FillUp versione 1.7

	# executable files	# total classes	# total methods	# total executable lines
SimplyDo	15	46	246	1281
TippyTipper	13	42	225	999
Trolly	5	19	64	364
MunchLife	2	10	28	184
FillUp	57	105	669	3807

Per ciascuna applicazione sono state effettuate 5 registrazioni da altrettanti utenti: Developers, Alessia, Fede, Nic, PTramont.

5.3 Risultati

I confronti sono stati effettuati in base alla copertura ottenuta e agli stati scoperti da ciascun test case nel Ripper Ibrido rispetto al Ripper originale. Nella versione ibrida del software, la copertura ottenuta in fase 1 coincide con quella ottenuta dalla versione originale, mentre la copertura relativa alla fase 2 è ottenuta in seguito all'esplorazione di stati considerati non equivalenti a quelli scoperti nella fase 1.

La configurazione dei Ripper è la seguente:

Tecnica Sistematica, HandlerBased, criterio di equivalenza "Simple", esplorazione "Breadth".

Tutti i dati sono stati elaborati con Emma utilizzando un file batch appositamente creato; il dettaglio delle librerie utilizzate:

Ant-Version: Apache Ant 1.6.1

Created-By: JDK_1.4

Main-Class: emmarun

Specification-Title: emma

Specification-Vendor: (C) Vladimir Roubtsov

Implementation-Vendor: vlad on Windows NT:4.0:x86

Specification-Version: 2.0

Implementation-Version: 2.0.5312

5.3.1 SimplyDo

	Ripper originale
Numero stati	6
class, %	91%
method, %	43%
block, %	35%
line, %	32%

Copertura Ripper originale
Ripper originale
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Developers	Ripper Ibrido
Numero stati	15
class, %	96%
method, %	77%
block, %	74%
line, %	72%

Copertura Ibrido: Developers
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Alessia	Ripper Ibrido
Numero stati	13
class, %	96%
method, %	82%
block, %	78%
line, %	77%

Copertura Ibrido: Alessia
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Fede	Ripper Ibrido
Numero stati	17
class, %	96%
method, %	80%
block, %	78%
line, %	76%

User: Nic	Ripper Ibrido
Numero stati	18
class, %	96%
method, %	85%
block, %	85%
line, %	83%

User: PTramont	Ripper Ibrido
Numero stati	9
class, %	93%
method, %	55%
block, %	49%
line, %	47%

L'incremento medio delle linee di codice tra le 5 registrazioni è del 39%.

Il numero di stati nuovi scoperti varia tra 7 e 12, in aggiunta ai 6 scoperti dal Ripper originale.

5.3.2 TippyTipper

	Ripper originale
Numero stati	7
class, %	90%
method, %	65%
block, %	56%
line, %	57%

Copertura Ripper originale
Ripper originale

User: Developers	Ripper Ibrido
Numero stati	8
class, %	98%
method, %	86%
block, %	88%
line, %	86%

Copertura Ibrido: Developers
Fase 1 Fase 2

User: Alessia	Ripper Ibrido
Numero stati	8
class, %	98%
method, %	88%
block, %	89%
line, %	88%

Copertura Ibrido: Alessia
Fase 1 Fase 2

User: Fede	Ripper Ibrido
Numero stati	8
class, %	98%
method, %	86%
block, %	86%
line, %	86%

Copertura Ibrido: Fede

Fase 1 Fase 2

class, % method, % block, % line, %

0% 25% 50% 75% 100%

User: Nic	Ripper Ibrido
Numero stati	8
class, %	98%
method, %	87%
block, %	88%
line, %	87%

Copertura Ibrido: Nic

Fase 1 Fase 2

class, % method, % block, % line, %

0% 25% 50% 75% 100%

User: PTramont	Ripper Ibrido
Numero stati	8
class, %	98%
method, %	87%
block, %	88%
line, %	87%

Copertura Ibrido: PTramont

Fase 1 Fase 2

class, % method, % block, % line, %

0% 25% 50% 75% 100%

L'incremento medio delle linee di codice tra le 5 registrazioni è del 30%.

E' stato scoperto 1 nuovo stato, in aggiunta ai 7 scoperti dal Ripper originale.

5.3.3 Trolly

	Ripper originale
Numero stati	5
class, %	68%
method, %	47%
block, %	31%
line, %	34%

User: Developers	Ripper Ibrido
Numero stati	5
class, %	68%
method, %	47%
block, %	31%
line, %	34%

User: Alessia	Ripper Ibrido
Numero stati	6
class, %	79%
method, %	80%
block, %	67%
line, %	72%

User: Fede	Ripper Ibrido
Numero stati	5
class, %	68%
method, %	47%
block, %	31%
line, %	34%

User: Nic	Ripper Ibrido
Numero stati	6
class, %	79%
method, %	78%
block, %	66%
line, %	69%

User: PTramont	Ripper Ibrido
Numero stati	6
class, %	79%
method, %	75%
block, %	64%
line, %	68%

L'incremento medio delle linee di codice tra le 5 registrazioni è del 21%.

E' stato scoperto 1 nuovo stato in due registrazioni, in aggiunta ai 5 scoperti dal Ripper originale.

5.3.4 MunchLife

	Ripper originale
Numero stati	5
class, %	90%
method, %	93%
block, %	76%
line, %	80%

Copertura Ripper originale
Ripper originale
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Developers	Ripper Ibrido
Numero stati	5
class, %	90%
method, %	93%
block, %	76%
line, %	80%

Copertura Ibrido: Developers
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Alessia	Ripper Ibrido
Numero stati	5
class, %	90%
method, %	93%
block, %	76%
line, %	80%

Copertura Ibrido: Alessia
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Fede	Ripper Ibrido
Numero stati	5
class, %	90%
method, %	93%
block, %	76%
line, %	80%

Copertura Ibrido: Fede
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Nic	Ripper Ibrido
Numero stati	5
class, %	90%
method, %	93%
block, %	76%
line, %	80%

Copertura Ibrido: Nic
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: PTramont	Ripper Ibrido
Numero stati	5
class, %	90%
method, %	93%
block, %	76%
line, %	80%

Copertura Ibrido: PTramont
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

Non vi è stato nessun incremento della copertura, perché nessuna registrazione utente ha consentito di scoprire nuovi stati.

5.3.5 FillUp

	Ripper originale
Numero stati	7
class, %	22%
method, %	16%
block, %	11%
line, %	13%

Copertura Ripper originale
Ripper originale
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Developers	Ripper ibrido
Numero stati	18
class, %	78%
method, %	72%
block, %	66%
line, %	67%

Copertura Ibrido: Developers
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Alessia	Ripper ibrido
Numero stati	18
class, %	78%
method, %	72%
block, %	66%
line, %	67%

Copertura Ibrido: Alessia
Fase 1 Fase 2
class, %
method, %
block, %
line, %
0% 25% 50% 75% 100%

User: Fede	Ripper ibrido
Numero stati	17
class, %	74%
method, %	67%
block, %	62%
line, %	63%

Copertura Ibrido: Fede
Fase 1 Fase 2
class, % method, % block, % line, %
0% 25% 50% 75% 100%

User: Nic	Ripper ibrido
Numero stati	19
class, %	78%
method, %	73%
block, %	67%
line, %	68%

Copertura Ibrido: Nic
Fase 1 Fase 2
class, % method, % block, % line, %
0% 25% 50% 75% 100%

User: PTramont	Ripper ibrido
Numero stati	18
class, %	78%
method, %	72%
block, %	67%
line, %	67%

Copertura Ibrido: PTramont
Fase 1 Fase 2
class, % method, % block, % line, %
0% 25% 50% 75% 100%

L'incremento medio delle linee di codice tra le 5 registrazioni è del 46%.

Il numero di stati nuovi scoperti varia tra 5 e 12, in aggiunta ai 7 scoperti dal Ripper originale.

Conclusioni e Sviluppi Futuri

Lo sviluppo del Ripper Ibrido a partire dal Ripper originale ha consentito di unire i vantaggi di un'esplorazione esclusivamente automatica con una esplorazione manuale. In particolare si è ottenuto un incremento dell'efficacia in termini di copertura e la garanzia di testare porzioni ben identificate della GUI dell'applicazione, potenzialmente non esplorate dalla sola procedura automatica. Lo sforzo aggiuntivo richiesto all'utente rispetto al Ripper originale è confinato alla registrazione di test case con un tool facilmente eseguibile quale è Robotium Recorder.

Il software è stato eseguito su un numero esiguo di applicazioni e di registrazioni per ognuna di esse. Per verificare la bontà del progetto o la necessità di modificarlo, in futuro è possibile testarlo su un insieme maggiore di applicazioni anche con GUI particolarmente complesse.

Appendice A1: Guida all'installazione e uso del Ripper Ibrido

A1.1 Prerequisiti

Requisiti di sistema	
Sistema operativo	Windows XP o più recente, Linux, Mac OS X
JDK	Versione 1.7.25 o più recente
Android Tools	A causa di un bug, è necessario avere esattamente queste versioni: • Android SDK Tools ver. 22.3; • Android SDK Platform Tools ver. 19.0.2; • Android SDK Build Tools ver. 18.1.1 E' possibile scaricare l'installer per Windows, Linux e Mac OS X o il pacchetto compresso pronto all'uso per Windows, Linux e Max OS X. Se si scarica l'installer, eseguire AVD Manager e aggiornare Platform Tools, Build Tools, e Android Libraries alle versioni richieste.
Android Libraries	Versione 2.3.3 o più recenti
Ant	Versione 1.8.2 o più recente. Ant può essere scaricato qui
Note	Se l'applicazione dichiara nel Manifest.xml una versione di "targetSdkVersion" e "minSdkVersion" maggiori della versione della libreria del Ripper (10), è necessario installare le librerie richieste attraverso l'SDK Manager

Configurazione di sistema (variabili ambiente)	
ANDROID_HOME	settata al path dell'Android SDK (e.g. android-sdk)
JAVA_HOME	settata al path di Java SDK (e.g. "C:\Java\jdk1.7.0_25")
PATH	deve contenere: "bin" path di Ant "platform-tools" path di Android SDK "tool" path di Android SDK Per OS Windows aggiungere "aapt.exe" al path
Android Libraries	Versione 2.3.3 o più recenti
Ant	Versione 1.8.2 o più recente. Ant può essere scaricato qui

Creazione AVD	
Target	Android 2.3.3 - API Level 10 (o più recente)
Device Ram Size	512 MB o più
Max VM Heap	128 MB
Internal Storage	200 MB o più
SD Card Size	64 MB o più
Snapshot	attivato
Note	E' fortemente raccomandato creare un nuovo AVD per ogni esecuzione del Ripper

Registrazione utente con Robotium Recorder	
Versione	2.1.25 (o più recente)
Installazione & Uso	http://robotium.com/pages/installation http://robotium.com/pages/user-guide

Note	E' fortemente raccomandato, al termine della registrazione, cancellare dall'AVD i dati dell'applicazione e rieseguire il test Android JUnit prodotto per verificare il corretto comportamento. E' importante che l'esecuzione del Ripper e la registrazione con Robotium Recorder siano effettuate usando un AVD avente la stessa dimensione dello schermo (stesso device emulato)

A1.2 Android Ripper Installer

AZIONI	
Estrarre l'archivio Android Ripper Installer	
Editare il file "ripper.properties"	**AUT_PATH**: la directory radice dell'applicazione da testare (AUT). In Windows usare gli slash ("/"). Tale directory deve contenere il file "Manifest"
	TEST_RR_FILE: file registrato con Robotium Recorder comprensivo di path. In Windows usare gli slash "/" (e.g. C:/ActivityTest/src/com/ex/test/ActivityTest.java)
	AVD_NAME: nome dell'AVD
	AVD_PORT: porta dell'AVD (default: 5554)
Eseguire Android Ripper Installer	java -jar AndroidRipperInstaller.jar
Chiudere emulatore al termine	

A1.3 Android Ripper Driver

A1.3.1 Fase 1

AZIONI	
Estrarre l'archivio Android Ripper Driver	
Editare il file "ripper.properties"	**APP_PACKAGE**: il nome del package principale dell'AUT (e.g. com.example). Il nome del package può essere trovato nel file "Manifest" come valore dell'attributo "package" del tag "manifest"
	APP_MAIN_ACTIVITY: il nome della classe dell'activity principale comprensiva del package (e.g. com.example.ui.MainActivity) Il nome dell'activity può essere trovato nel file Manifest.xml come valore dell'attributo android:name di uno dei tag "activity"
	AVD_NAME: nome dell'AVD
	AVD_PORT: porta dell'AVD (default: 5554)
Eseguire Android Ripper Installer	java -jar AndroidRipper.jar s systematic.properties

A1.3.2 Fase 2

AZIONI	
Eseguire nuovamente Android Ripper Driver al termine della fase1	java -jar AndroidRipper.jar s systematic.properties

Per una nuova esecuzione, eliminare i file di output prodotti nella precedente.

Appendice A2: Esempio d'uso del Ripper Ibrido

Questa sezione illustra un caso d'uso reale del Ripper Ibrido, usando come AUT l'applicazione "FillUp".

A2.1 Prerequisiti

Configurazione di sistema	
Sistema operativo	Windows XP
JDK	jdk1.7.0_51
Android Tools	Scaricato il pacchetto compresso pronto all'uso per Windows.
Android Libraries	Versione 2.3.3
Ant	Versione 1.8.2
ANDROID_HOME	C:\Tirocinio\AndSDK\android-sdk
JAVA_HOME	C:\Programmi\Java\jdk1.7.0_51
PATH	C:\Tirocinio\apache-ant-1.8.2-bin\apache-ant-1.8.2\bin; C:\Tirocinio\AndSDK\android-sdk\platform-tools; C:\Tirocinio\AndSDK\android-sdk\tools;C:\Tirocinio\AndSDK\android-sdk\build-tools\18.1.1; C:\Tirocinio\AndSDK\android-sdk\tools; C:\Programmi\Java\jdk1.7.0_51\bin;

Creazione AVD	
Target	Android 2.3.3 - API Level 10
Device Ram Size	512 MB
Max VM Heap	128 MB
Internal Storage	200 MB
SD Card Size	64 MB
Snapshot	attivato
Altre opzioni	vedi figura

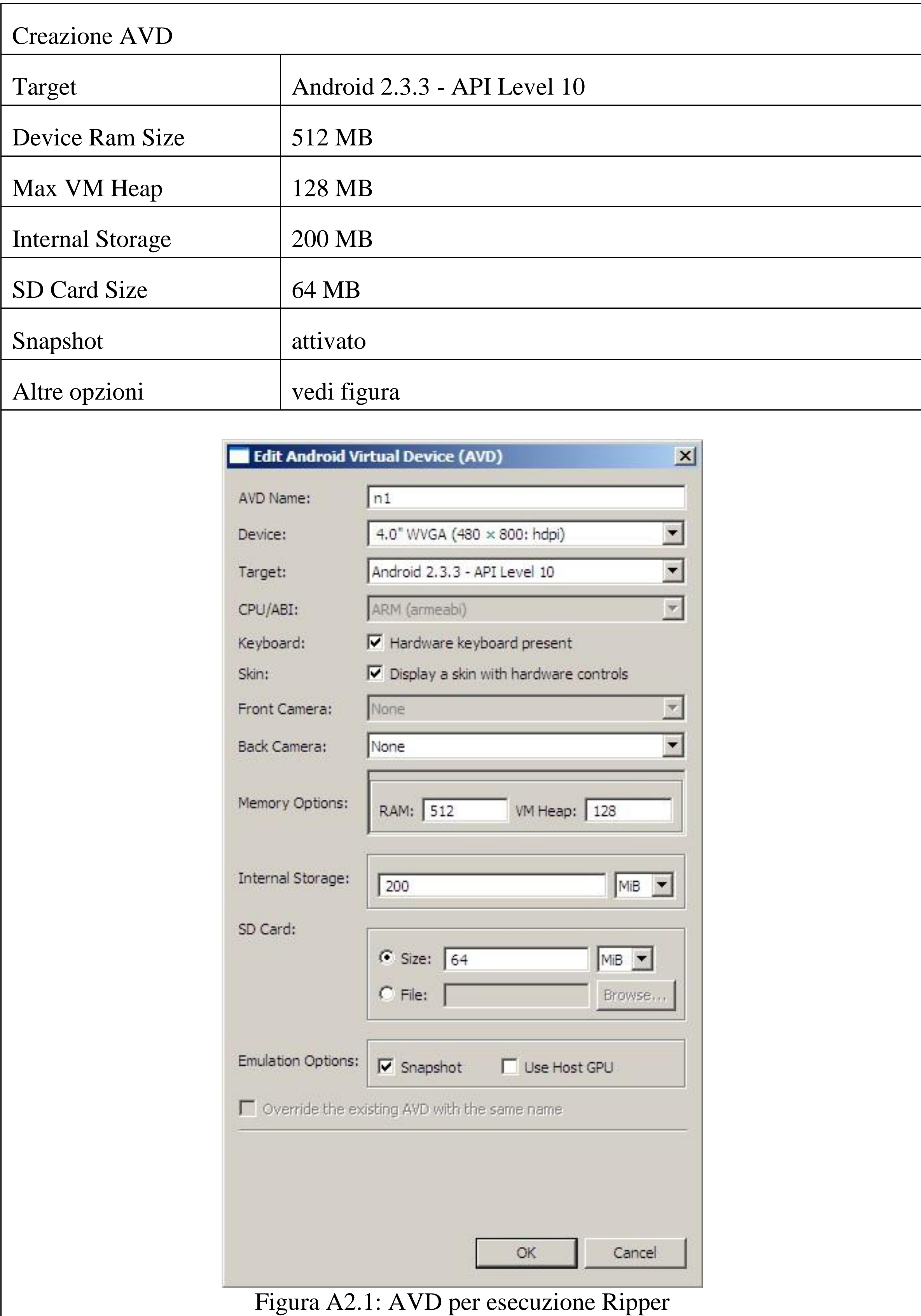


Figura A2.1: AVD per esecuzione Ripper

Registrazione utente con Robotium Recorder	
IDE	Eclipse - Version: Luna Service Release 1 (4.4.1)
Android Libraries	Versione 4.0.3 API Level 15
Robotium Recorder	Versione 2.1.25
Installazione & Uso	http://robotium.com/pages/installation http://robotium.com/pages/user-guide
Note	E' importante che l'esecuzione del Ripper e la registrazione con Robotium Recorder siano effettuate usando un AVD avente la stessa dimensione dello schermo (nell'esempio 4" WVGA (Nexus S) (480 x 800:hdpi))

A2.2 Registrazione con Robotium Recorder

Creare un AVD (unico requisito: API Level 15 o maggiore) e avviarlo

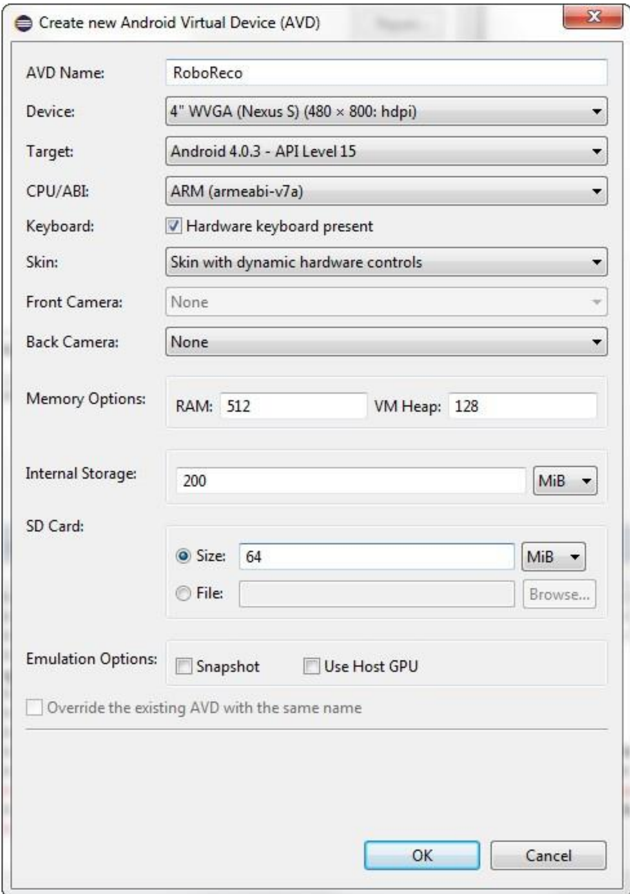


Figura A2.2 AVD per registrazione con Robotium Recorder

Da Eclipse: File > New > Other. Nella finestra selezionare "New Robotium Test" dalla sezione "Android - Robotium Recorder" e click su "Next".

Selezionare l'applicazione su cui effettuare la registrazione (es FillUp, il cui progetto è presente in Eclipse) e dare un nome al progetto di test (es FillUpTest).

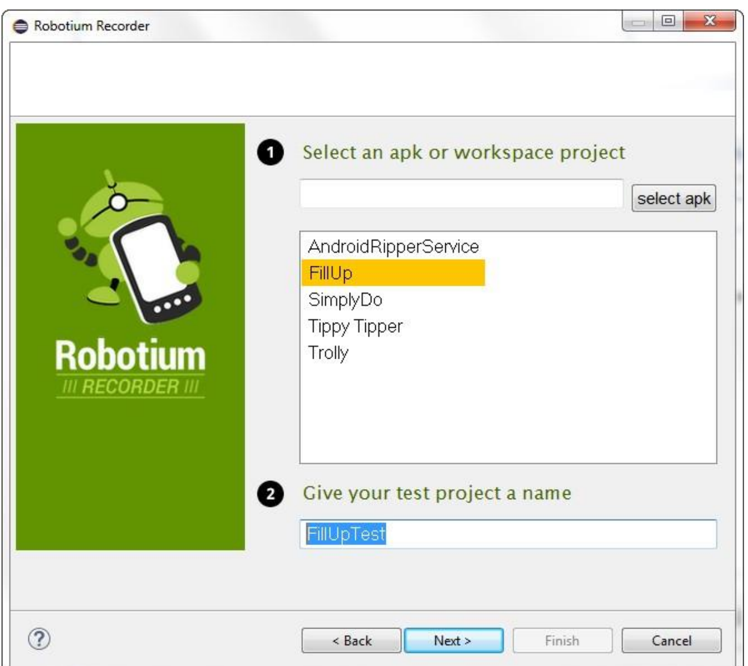


Figura A2.3: Fase di registrazione con Robotium Recorder

click su "Next":

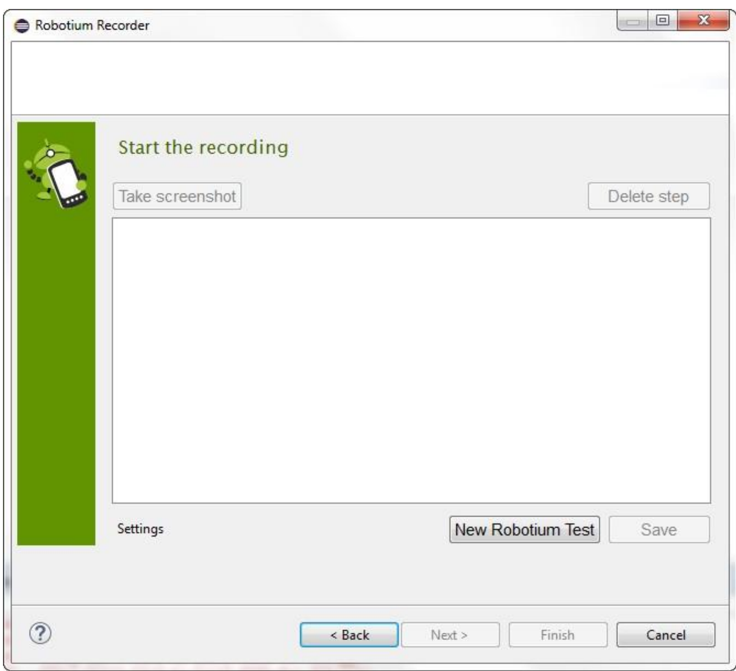


Figura A2.4: Fase di registrazione con Robotium Recorder

click su "New Robotium Test". Nell'AVD verrà eseguita l'applicazione (FillUp); esplorare l'applicazione:

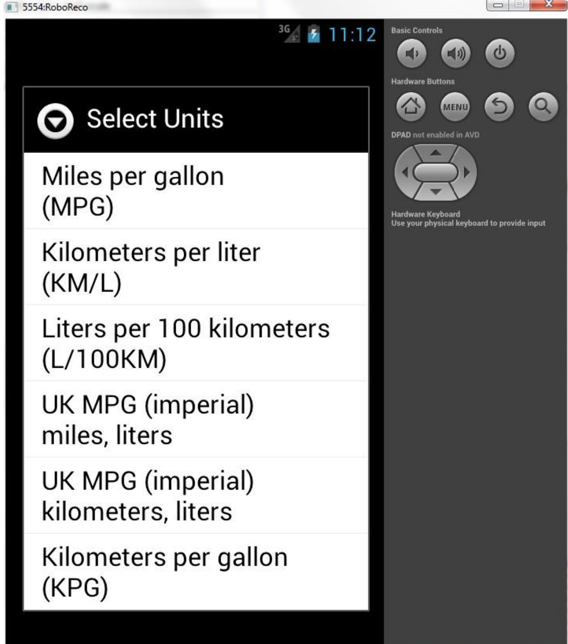


Figura A2.5: Avvio dell'applicazione nell'AVD

Nella finetra di Robotium Recorder verranno registrati tutti gli eventi scatenati nella GUI dell'app:

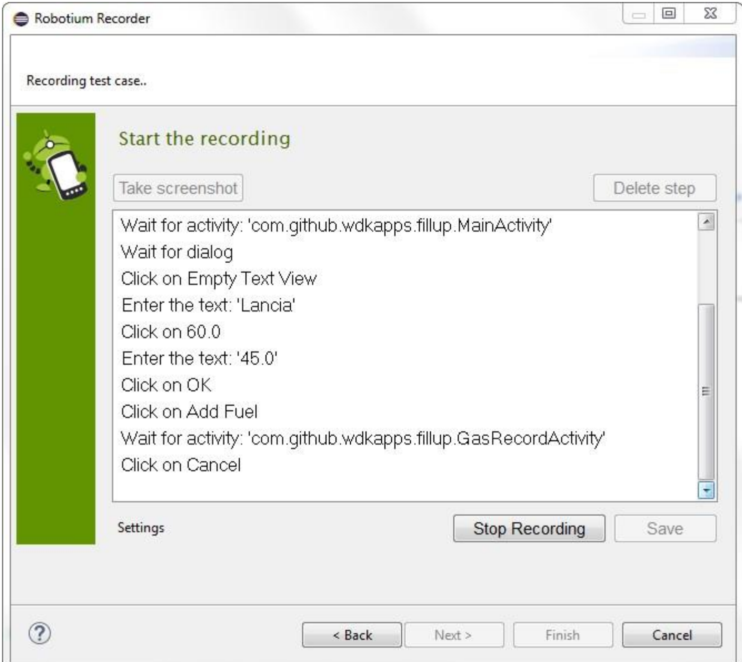


Figura A2.6: Fase di fine registrazione con Robotium Recorder

Al termine, click su Stop Recording e salvare (clic su "Save") il test case.

E' possibile salvare gli screenshot (Take screenshot) e cancellare i singoli eventi registrati

(Delete step).

Salvare assegnando un nome al test case (es: TestCase_FillUp).

In Eclipse è stato creato un nuovo progetto "FillUpTest". All'interno del package "com.github.wdkapps.fillup.test" è presente la classe "TestCase_FillUp.java" relative alal registrazione.

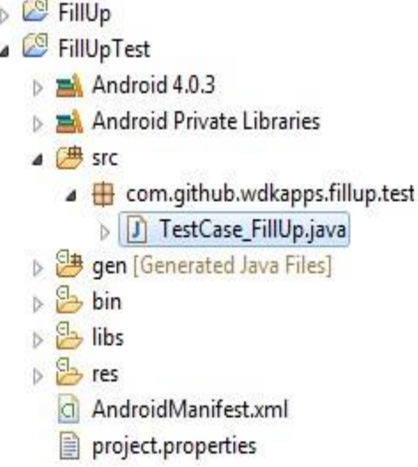


Figura A2.7: Creazione progetto di test in Eclipse

E' <u>fortemente raccomandato</u>, al termine della registrazione, rieseguire il test Android JUnit prodotto per verificare il corretto comportamento.

E' necessario a tal proposito cancellare dall'AVD i dati temporanei dell'app.

Nell'AVD click sul tasto "MENU" → Manage Apps e selezionare l'app (FillUp)

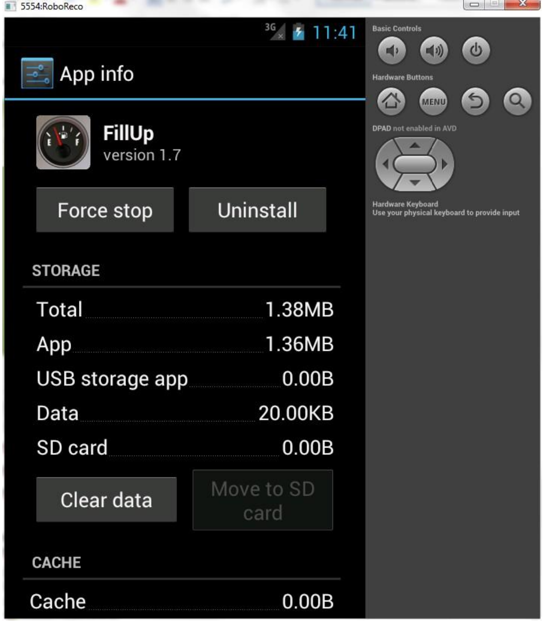


Figura A2.8: Cancellazione dati nell'app di test

click su "Clear data" e su "ok".

In Eclipse, fare clic col tasto destro sul test case (TestCase_FillUp) → Run As → Android JUnit Test. Si aprirà la finestra JUnit con l'esito del test:

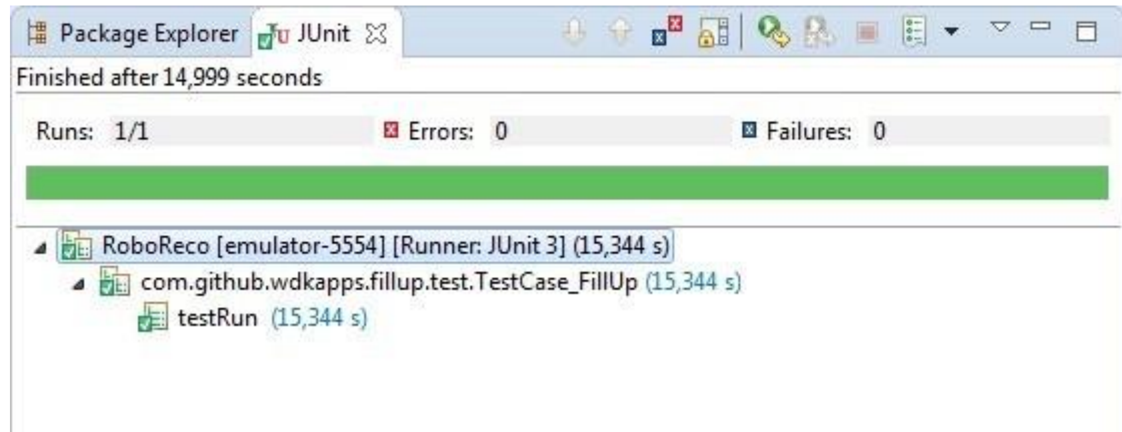


Figura A2.9: Esito esecuzione Android JUnit Test

Il test case (file "TestCase_FillUp.java") sarà fornito come input al Ripper

```java
package com.github.wdkapps.fillup.test;

import com.github.wdkapps.fillup.StartupActivity;

public class TestCase_FillUp extends ActivityInstrumentationTestCase2<StartupActivity> {
    private Solo solo;

    public TestCase_FillUp() {
        super(StartupActivity.class);
    }

    public void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation());
        getActivity();
    }

    @Override
    public void tearDown() throws Exception {
        solo.finishOpenedActivities();
        super.tearDown();
    }

    public void testRun() {
        // Wait for activity: 'com.github.wdkapps.fillup.StartupActivity'
        solo.waitForActivity(com.github.wdkapps.fillup.StartupActivity.class, 2000);
        // Wait for dialog
        solo.waitForDialogToOpen(5000);
        // Click on Kilometers per liter (KM/L)
        solo.clickOnView(solo.getView(android.R.id.text1, 1));
        // Wait for activity: 'com.github.wdkapps.fillup.MainActivity'
        assertTrue("com.github.wdkapps.fillup.MainActivity is not found!", solo.waitForActivity(com.github.wdkapps.fillup.MainActivity
        // Wait for dialog
        solo.waitForDialogToOpen(5000);
        // Click on Empty Text View
        solo.clickOnView(solo.getView(com.github.wdkapps.fillup.R.id.textVehicleName));
        // Enter the text: 'Lancia'
        solo.clearEditText((android.widget.EditText) solo.getView(com.github.wdkapps.fillup.R.id.textVehicleName));
        solo.enterText((android.widget.EditText) solo.getView(com.github.wdkapps.fillup.R.id.textVehicleName), "Lancia");
```

Figura A2.10: Classe ottenuta dalla registrazione

A2.3 Android Ripper Installer

Nella directory AndroidRipper_Ibrido\Release sono presente due directory:

- AndroidRipperInstaller

- AndroidRipperDriver

Spostiamoci nella directory "AndroidRipperInstaller"

Avviare l'AVD precedentemente creato (differente da quello usato per la registrazione).

Editare il file "ripper.properties":

AUT_PATH = C:/Tirocinio/AUT/FillUp

TEST_SUITE_PATH = %PWD%/AndroidRipper

SERVICE_APK_PATH = %PWD%

TEST_RR_FILE = C:/Tirocinio/AUT/TestCase_FillUp.java

AVD_NAME = n1

AVD_PORT = 5554

Eseguire Android Ripper Installer:

java –jar AndroidRipperInstaller.jar

Esito:

```
Android Ripper Installer
Using default configuration file 'ripper.properties'!
Starting installation...
[1430675594618] Booting Emulator!
[1430675594634] Waiting for Emulator...
emulator: warning: opening audio output failed
emulator: emulator window was out of view and was recentered
Emulator Online!
Emulator Booted!
[1430675719915] Emulator online!
[1430675719915] adb devices...
[1430675719962] Editing 'Configuration.java'
[1430675720040] Editing 'AndroidManifest.xml'
[1430675720040] Rielaboro il file di test...
[1430675720040] File caricato : C:/Tirocinio/AUT/TestCase_FillUp.java
[1430675720415] update project...
[1430675721478] Updated local.properties
[1430675721493] Updated file C:\Tirocinio\AUT\FillUp\proguard-project.txt
[1430675721493] It seems that there are sub-projects. If you want to update them
[1430675721493] please use the --subprojects parameter.
[1430675721509] update test project...
[1430675722493] Resolved location of main project to: C:\Tirocinio\AUT\FillUp
[1430675722540] Updated project.properties
[1430675722540] Updated local.properties
[1430675722540] Updated file
C:\Tirocinio\AndroidRipperIbrido\AndroidRipper_Ibrido\Release\AndroidRipperInstaller\AndroidRi
pper\proguard-project.txt
[1430675722556] Updated ant.properties
[1430675722571] compiling...
[1430675787165] install ripper-serivice apk...
[1430675791087]         pkg: /data/local/tmp/AndroidRipperService.apk
[1430675791087]
[1430675793368] Success
```

```
[1430675793368]
[1430675793571] adb shell mkdir...
[1430675793759] adb chmod 777...
[1430675793931] adb shell rm...
[1430675794134] rm failed for /data/data/com.github.wdkapps.fillup/files/*, No such file or directory
[1430675794134]
[1430675794212] Installation finished!
[1430675794212] Unlocking Emulator...
```

A2.4 Android Ripper Driver

Spostiamoci nella directory “AndroidRipperDriver”.

Editare il file “systematic.properties”:

```
aut_package = com.github.wdkapps.fillup
aut_main_activity = com.github.wdkapps.fillup.StartupActivity
avd_name = n1
avd_port = 5554
coverage=1
ping_failure_threshold=1
planner=it.unina.android.ripper.planner.HandlerBasedPlanner
comparator=it.unina.android.ripper.comparator.GenericComparator
comparator_configuration=CustomWidgetSimpleComparator
```

Eseguire Android Ripper Driver:

java -jar AndroidRipper.jar s systematic.properties

Nell’AVD partirà l’applicazione.

Al termine dell’esecuzione si otterranno i seguenti output:

directory: “coverage”, “junit”, “logcat”, “model”

file: “current_ActivityStateList.bin”; “current_TaskList.bin”

Eseguire nuovamente Android Ripper Driver al termine della fase 1:

java -jar AndroidRipper.jar s systematic.properties

Al termine dell’esecuzione si otterrà la stessa struttura di directory con file aggiornati.

Per una nuova esecuzione, eliminare i file di output prodotti nella precedente.

Bibliografia

[1] Massimo Carli, Android. Guida per lo sviluppatore, Apogeo, 2010

[2] Ian Sommerville, Ingegneria del software, Pearson, 2007

[3] Giuseppe di Maio, Realizzazione di tecniche parallele di generazione automatica di casi di test per applicazioni Android, Tesi di laurea Università degli studi di Napoli Federico II, 2013-2014

[4] Developer Android Testing, http://developer.android.com/tools/testing/testing_android.html, 05/2015

[5] Developer Android Tools, http://developer.android.com/tools/help/index.html, 05/2015

[6] Robotium, https://code.google.com/p/robotium/, 05/2015

[7] Robotium Recorder, https://code.google.com/p/robotium/, 05/2015

[8] Porfirio Tramontana, Slide Ingegneria del Software 2: Testing Automation, https://www.docenti.unina.it/downloadPub.do?tipoFile=md&id=426905, 05/2015

[9] Anna Rita Fasolino, Slide Ingegneria del Software 2: Automazione del testing e Analisi Mutazionale, http://www.federica.unina.it/ingegneria/ingegneria-software-ii/automazione-testing-analisi-mutazionale/30/, 05/2015

[10] JUnit Testing Framework, http://junit.org/, 05/2015