Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Tesis de Licenciatura en Ciencias de la Computación

Testing automático y análisis de cobertura de aplicaciones Android

Fernando G. Paulovsky
fernandopaulovsky@gmail.com
L.U. 880/05

Directores: Diego Garbervetsky, Esteban Pavese

Noviembre de 2013

Resumen

En la actualidad, el mercado de aplicaciones móviles crece día a día, con varios fabricantes y desarrolladores compitiendo continuamente en la mejora de sus plataformas, y ampliando la oferta de aplicaciones. Por otra parte, el bajo costo de estas plataformas hace que su presencia sea ubicua, y sus aplicaciones cubran varios espectros. Bajo este escenario, asegurar el correcto funcionamiento de las aplicaciones de forma previa a su puesta en el mercado es crucial. Sin embargo, las características propias de estas aplicaciones (desarrollo bajo distintos *frameworks*, alta reactividad, interacción en base a interfaces táctiles, modelo de navegación de la aplicación) hacen que las técnicas clásicas de verificación y validación no sean directamente aplicables. En esta tésis nos proponemos desarrollar una estrategia de verificación basada en testing de este tipo de aplicaciones, enfocándonos en la plataforma *Android*. Además, presentaremos una noción de cobertura apropiada para el proceso de verificación.

Agradecimientos

Agradezco a todos mis maestros, no sólo a los de la universidad sino también a los que me formaron desde que comencé mis estudios. Estoy seguro de que aprendí de ellos no sólo los diferentes temas que dictaron en sus cursos, sino mucho cosas más. Agradezco a cada uno de los profesores o ayudantes de la Universidad de Buenos Aires que me ayudaron a transitar esta carrera hasta el dia de hoy. Particularmente, quiero agradecer a Diego y Esteban, mis directores de tesis, que siempre estuvieron para ayudarme y brindarme su conocimiento y sus consejos en lo que hiciera falta. Han tenido una enorme paciencia que nunca deje de destacar. Agradezco tambien a Guido y Charly, los jurados de esta tesis, que con sus correcciones hicieron que éste fuera un mejor trabajo.

Sin duda alguna nada de lo que hago puede ser posible sin el apoyo y la ayuda constante de mi gente. Es mi gente la que hace que toda situación en mi vida sea mas llevadera, divertida y feliz. Es mi gente la que me conoce, me forma día a día y hace de mi una mejor persona. Por lo tanto no me queda mas que agradecerles, ya que es gracias a ellos, por su consejo, paciencia, compañía y amor, que hoy puedo estar terminando una carrera que siempre fue un gran desafío para mí. Este no es un logro personal, es un logro compartido con todas estas personas, mi familia y mis amigos, que hacen de mi quien soy.

Gracias a Tere, Aisac, Bobe, Zeide, Vero G, Ammi, Mert-Ammi, Matienz, Guille, Vero, Fran, Aida, Rosa, Guiyane, Sari, Emi, Dani, Marce, Vane, Ari, Vale, Nico, Diego, Ari B, Piru, Pablo, Loli, Katu, Feli, Benja, Eze, Mazal, a la gran cantidad de amigos que la vida me dio y particularmente a mis viejos (Ricardo y Patri) y hermanos (Darryl y Maishel), por convertirme en la persona que soy, acompañarme siempre, guiarme en el camino del estudio y el trabajo y ayudarme a superarme día a día.

Gracias Vani por ayudarme en absolutamente todo, hacerme mejor cada día, estar siempre conmigo, bancarme en cada momento, y por sobre todas las cosas por hacerme feliz.

Índice general

1. Introducción **12**

1.1. Objetivo 13

1.2. Estructura de la tesis 13

2. Las aplicaciones *Android* **15**

2.1. Estructura de las aplicaciones *Android* 15

2.2. El framework de testing 19

3. Construcción **21**

3.1. Motivación del modelo 21

3.2. Modelo formal 24

4. Implementación **29**

4.1. Arquitectura y componentes 29

4.2. Configuración previa 31

4.3. Análisis de cobertura y mapa de la interfaz de usuario 33

4.4. Limitaciones 35

4.5. Detalles de implementación 36

4.6. Requerimientos técnicos e instalación 41

4.7. Utilización 42

5. Resultados **45**

5.1. Android Calculator 45

5.2. Notepad 47

5.3. Notepad con errores 52

5.4. ContactManager 53

5.5. TippyTipper 55

5.6. Taskos 57

5.7. Conclusiones 58

6. Trabajo relacionado **59**

7. Conclusiones **61**

7.1. Trabajo a futuro 61

Bibliografía **63**

Glosario

actividad Representa una pantalla con su correspondiente interfaz de usuario.

Android Debug Bridge Es un sistema que utiliza la línea de comandos y permite la comunicación entre una computadora y un emulador o un dispositivo móvil físico que corra *Android*. Adopta una arquitectura cliente-servidor que incluye tres componentes: Un cliente que corre en una computadora en un ambiente de desarrollo, el cual se invoca a través de la línea de comandos; Un server que corre en segundo plano en la misma computadora, y un daemon, que corre como un proceso en segundo plano en el emulador o dispositivo físico.

androidManifest.xml Es un archivo que toda aplicación Android debe declarar en su directory raíz. Contiene información esencial acerca de la aplicación que el sistema Android debe conocer antes de poder ejecutarla. Describe el nombre de la aplicación, sus componentes, los intents que puede manejar cada actividad, permisos de ejecución y acceso entre otras cosas.

APK Un archivo con extensión .apk (Application PacKage File) es un paquete para el sistema operativo Android. Este formato es una variante del formato JAR de Java y se usa para distribuir e instalar componentes.

apktool Es una herramienta que permite realizar ingeniería reversa de aplicaciones *Android* cerradas. Permite decodificar sus recursos hasta casi su forma original y re-compilarlas luego de realizar modificaciones.

aplicación de instrumentación Android Es una aplicación *Android* que utiliza herramientas provistas por el framework de instrumentación de la plataforma y explicita acciones para ser ejecutadas sobre las diferentes vistas de la interfáz gráfica de otra aplicación. Ésta última aplicación es considerada la aplicación bajo prueba. Las acciones se explicitan como tests *Android* que derivan de JUNIT tests.

caso de test Un caso de prueba (test case) es un conjunto de condiciones o variables bajo las cuales la persona que prueba el software determinará si una aplicacion, un sistema de software o uno de sus componentes funciona de acuerdo a su especificación. El mecanismo para determinar si un elemento de software pasa o no dicha prueba es conocido como "oráculo". Un oráculo puede ser un requerimiento funcional, un caso de uso o una heurística.

cobertura La cobertura de un proceso de testing establece la medida en que el proceso de testing ejercita el software bajo prueba. En el caso de testing white-box, una medida de cobertura posible es la cobertura de código, que mide el grado en que

el código fuente de un componente de software es ejercitado por un conjunto de casos de prueba.

emulador Es un programa que simula un dispositivo móvil virtual que corre en una computadora. Permite desarrollar y testear aplicaciones *Android* sin la necesidad de utilizar un dispositivo real.

framework de instrumentación de Android Es un conjunto de herramientas de software que permite, entre otras cosas, el testeo de una aplicación *Android* basado en la interacción mediante su interfaz gráfica.

intent Es una descripción abstracta de una operación a realizar. Su principal uso es el de iniciar actividades.

jar Es un tipo de archivo que empaqueta recursos y aplicaciones escritas en el lenguaje Java. Está comprimido mediante el formato zip y cambiada su extension a .jar que significa tarro en inglés.

jarsigner Es una herramienta que permite tanto firmar digitalmente archivos jar como chequear la integridad de sus firmas y contenido.

Robotium Es un framework de testing de código abierto que permite la escritura de casos de test de caja negra para aplicaciones *Android*.

testing black-box El testing black-box (caja negra) es una metodología de testing de software que examina la funcionalidad de un componente de software sin tener en cuenta su funcionamiento interno. El software es estudiado desde el punto de vista de las entradas que recibe y las salidas o respuestas que produce. En esta modalidad de testing, es central la forma en que el software estudiado interactúa con el medio que le rodea.

testing de regresion El testing de regresión es el proceso de testing de software que trata de descubrir nuevos errores en un sistema, ya sea en los aspectos funcionales como en los no funcionales, luego de la introducción de cambios en el mismo tales como mejoras, actualizaciones o cambios en sus configuraciones.

testing gray-box El testing gray-box (caja gris) es una metodología de testing que examina la funcionalidad de un componente de software desde el punto de vista de las entradas que recibe y las salidas o respuestas que produce, pero aprovechándose de la ventaja de contar con cierto grado de conocimiento de la estructura interna del mismo. Por ello, es una mezcla entre black-box y white-box.

testing white-box El testing white-box (caja blanca) es una metodología de testing de software que examina la estructura interna del componente bajo prueba y utiliza este conocimiento con el fin generar los casos de prueba correspondientes.

vista Es un elemento gráfico que controla un espacio rectangular particular dentro de la ventana de una actividad, y puede responder a las acciones del usuario. Ejemplos de vistas pueden ser botones, listas, grillas y cajas de texto, entre otros, que disparan determinadas acciones cuando un usuario las toca o interactúa con ellas a través de algún gesto táctil.

zipalign Es una herramienta que introduce ciertas optimizaciones a los archivos apk. Entre otras cosas hace que toda la informacion incluída en el apk, como imágenes y otros archivos, esten alineados de a 4 bytes, lo que produce una reducción de la memoria RAM consumida por los dispositivos o emuladores al correr la aplicación correspondiente.

Capítulo 1

Introducción

El testing de software *black-box* (testeo de caja negra) es una actividad primordial durante el desarrollo de un proyecto de software, y de sus diferentes versiones una de las más utilizadas es el testing automático. En este caso, el software es ejercitado mediante distintas heurísticas de exploración a fin de encontrar fallas básicas en la implementación tales como la presencia de excepciones o errores sin manejar. De tal manera, la propiedad a testear resulta ser la ausencia de este tipo de errores. El *testing* automático de aplicaciones con fuerte uso de interfaces gráficas (*GUIs*) introduce problemas aún más complejos, dado que los puntos de entrada a ejercitar no son fácilmente detectables. En particular, uno de los dominios donde las interfaces gráficas son absolutamente predominantes es el de las aplicaciones para smartphones y otros dispositivos táctiles. Existen distintos avances en este sentido ([11], [12], [13], [14], [15], [16]), aunque no hay demasiados trabajos específicamente dedicados a las aplicaciones móviles. Además, otro punto de interés es que tampoco existe demasiado avance en técnicas que permitan, al menos, estimar el grado de cobertura del proceso de testing sobre éste tipo de aplicaciones. Cabe recordar que la medida de cobertura es un objetivo clásico a lograr, ya que altas coberturas usualmente muestran correlación con la detección de errores en el software bajo análisis.

Por otra parte, el uso de diversos *frameworks* durante el desarrollo de aplicaciones móviles fuerza al desarrollador al uso de las metáforas impuestas por estos *frameworks*, siendo muy común el de subdividir las aplicaciones en páginas y/o actividades. Si bien tal metáfora es útil, puede ser difícil para el desarrollador mantener su concepción del software alineada con estas metáforas. De esta forma, una herramienta que permita mapear entre diversos componentes de software y estas metáforas puede ser de utilidad: tal herramienta podría proveer al desarrollador una abstracción del software desarrollado, que le permita validar si se alinea con la solución especificada. Adicionalmente, este mapeo puede demostrar ser útil para introducir una nueva medida de cobertura del proceso de *testing* mencionado anteriormente.

En particular, el sistema *Android* [1] es un sistema reactivo, y como tal, mantiene una continua interacción con su entorno, respondiendo ante los estímulos externos, generados por el usuario o las diversas aplicaciones que se ejecutan, en función de su estado interno. Esto causa que su comportamiento sea complejo de analizar y muy sujeto a errores.

1.1. Objetivo

El objetivo de este trabajo es proveer técnicas automatizadas con el fin de facilitar el proceso de testing automático de aplicaciones de la plataforma móvil *Android* [1] mediante un fuerte uso de sus interfaces gráficas (*GUIs*). Para ello, se presentarán tanto la arquitectura básica de las aplicaciones de la plataforma como las herramientas de testeo que la misma provee incluyendo sus limitaciones.

Luego, a partir de este conocimiento, se presentará un algoritmo de testing, y su correspondiente implementación llamada *ATG* (Android Test Generator) que es el principal aporte del presente documento, que deberá ser capaz de tomar el instalador de una aplicación (APK - application package file), instalarlo en un dispositivo (o en un emulador), ejecutarlo con el fin de generar trazas de manera aleatoria, conocer si durante la ejecución de dichas trazas se produjeron excepciones o errores sin manejar, y retornarlas de manera de que sirvan como tests de regresión.

Así mismo, se presentará un cálculo de cobertura basado en la interfaz gráfica de las aplicaciones bajo prueba, con el fin de conocer el grado de alcance de las trazas generadas, y un grafo que permita visualizar de manera gráfica los diversos componentes de las misas, sus interrelaciones y su participación en las trazas generadas.

1.2. Estructura de la tesis

En el capítulo 2 presentamos la estructura de las aplicaciones *Android* junto con los diferentes componentes del *framework* de testing de la plataforma.

En el capítulo 3 damos un ejemplo motivador para nuestra herramienta e introducimos los algoritmos utilizados en ésta, los aspectos formales que los fundamentan, y establecemos una notación clara para los mismos.

En el capítulo 4 introducimos la arquitectura y detalles de implementación de nuestra herramienta, *Android Test Generator*. Por otro lado, se enumeran y explican todas las acciones que la herramienta realiza, más allá de la ejecución propiamente dicha de los algoritmos mencionados. Así mismo, se introducen las diferentes limitaciones encontradas a la hora de la implementación, y ciertas decisiones por las cuales algunos de sus efectos pudieron ser mitigados.

En el capítulo 5 introducimos los aspectos involucrados en la validación de la hipótesis del presente trabajo a partir del desarrollo de casos de estudio, que nos permiten analizar la utilidad de nuestra herramienta, así como también la escalabilidad de la misma.

En el capítulo 6 mencionamos brevemente otros trabajos relacionados con el nuestro.

Finalmente, en el capítulo 7 presentamos las conclusiones de esta tésis y discutimos algunas alternativas de trabajo a futuro.

Capítulo 2

Las aplicaciones *Android*

En este capítulo se presentan tanto la estructura básica de las aplicaciones de la plataforma *Android*, es decir, sus componentes y la manera en que se interrelacionan, como las distintas herramientas disponibles en la plataforma que posibilitan el testeo de aplicaciones basado en la interacción mediante sus interfaces gráficas (GUIs).

2.1. Estructura de las aplicaciones *Android*

Las aplicaciones *Android* se escriben nativamente en *Java*. Las herramientas de su SDK (*Software Development Kit*, kit de desarollo de software) [1] compilan el código, junto con ciertos recursos y datos particulares de la aplicación, en un paquete *Android* cuya extensión es APK. Dicho paquete es considerado una aplicación.

Los componentes de las aplicaciones son los bloques de construcción esenciales de una aplicación *Android*. Cada uno es un punto diferente mediante el cual el sistema puede acceder a las aplicaciones, y aunque no todos los componentes son en realidad puntos de acceso a nivel del usuario y algunos dependen de los otros, cada uno desempeña un rol específico.

Existen cuatro tipos de componentes. Cada uno sirve para un propósito distinto, por lo que tienen diferentes ciclos de vida que definen la manera en que los mismos son creados y destruídos.

A continuación se explican los diferentes componentes:

- Actividades: Una actividad representa una sola pantalla con una interfaz de usuario. Por ejemplo, una aplicación de e-mail puede tener una actividad que muestra una lista de nuevos e-mails, otra para redactar un nuevo e-mail y otra para leerlos. A pesar de que estas actividades trabajan de manera conjunta cada una es independiente del resto, por lo que otra aplicación puede iniciar cualquiera de estas actividades sin pasar por el resto de las actividades de la misma aplicación (si dicha aplicación lo permite). Por ejemplo, una aplicación que utilice la cámara de fotos podría iniciar la actividad para redactar un nuevo e-mail, con el fin de compartir una foto. Las actividades se implementan como subclases de la clase android.app.Activity.
- Servicios: Un servicio es un componente que corre en segundo plano (*background*) con el fin de realizar operaciones de larga duración o para realizar trabajo para procesos remotos. Un servicio no provee una interfaz de usuario.

- Proveedores de contenidos: Un proveedor de contenido administra información que puede ser eventualmente compartida por varias aplicaciones. Por ejemplo, el sistema posee un proveedor de contenido que administra la información concerniente a los contactos, de manera de que diversas aplicaciones con los permisos correspondientes puedan leer o escribir información acerca de una persona particular.
- Receptores *broadcast*: Un receptor *broadcast* es un componente que responde anuncios generales a nivel del sistema. Muchos de estos anuncios se originan en el mismo sistema, como por ejemplo el anuncio de que la pantalla se apagó, el nivel de batería es bajo, o que se sacó una foto con la camara, pero también pueden ser generados por aplicaciones, por ejemplo, con el fin de informar al resto de las aplicaciones que se dispone de cierto nuevo dato o de que se terminó de realizar alguna operación.

Un aspecto único en el diseño del sistema *Android* es que cualquier aplicación puede iniciar componentes de otras aplicaciones, por lo que éstas no tienen un único punto de entrada. Para que una aplicación inicie un componente de otra, la primera debe enviar un mensaje al sistema indicando su intención de uso y es éste último el que inicia el componente en cuestión. La intención (*intent*) contiene entre otras cosas la acción a realizar, y puede también contener una categoría. Los mensajes de intención de uso son instancias de la clase android.content.Intent.

Cada aplicación debe definir un archivo llamado AndroidManifest.xml en donde se declaran, entre otras cosas, los componentes pertenecientes a la misma, y las acciones asociadas a los mismos, de manera de que cuando el sistema recibe un intent con una determinada acción inicia el componente correspondiente.

De los cuatro tipos de componentes pertenecientes a la plataforma, esta tésis se centrará en las actividades, ya que son los únicos componentes que manejan y tienen interacción con la interfaz de usuario. Teniendo esto en cuenta, de ahora en adelante en este documento entenderemos a una aplicación *Android* como un conjunto de una o más actividades.

Ahora bien, cuando un usuario inicia una aplicación desde el menú principal, ¿cuál de todas las actividades pertenecientes a la aplicación debe iniciarse? En este caso, el sistema inicia la actividad que esté asociada a la acción android.intent.action.MAIN y a la categoría android.app.category.LAUNCHER al mismo tiempo, por lo que éstos no pueden estar asociados a más de una actividad en la misma aplicación. La asociación de la acción y la categoría se realiza en base a un *intent*.

Una vez que la aplicación se encuentra ejecutando, la actividad activa puede iniciar otras actividades. Cada vez que una actividad inicia, la anterior suspende su ejecución, pero el sistema la almacena en una pila, de manera de que si se presiona el botón *back*, se pueda regresar a la actividad anterior.

La interfaz de usuario de una actividad está compuesta por una jerarquía de vistas (cada una de éstas es una instancia de una clase derivada de android.view.View). Cada vista controla un espacio rectangular particular dentro de la ventana de la actividad, y puede responder a las acciones del usuario. Ejemplos de vistas pueden ser botones, listas, grillas y cajas de texto, entre otros, que disparan determinadas acciones cuando un usuario las toca o interactúa con ellas a través de algún gesto táctil.

La disposición gráfica de las vistas en la interfaz de una actividad puede ser definida

en el código fuente de la actividad, o también mediante ciertos archivos XML ubicados en el directorio res/layout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" ... >
    <EditText android:id="@+id/title"
        android:maxLines="1" ... />
    <Button android:id="@+id/ok"
        android:text="@string/button_ok" ... />
</LinearLayout>
```

Figura 2.1: XML que define la estructura gráfica de la actividad TitleEditor, de la aplicación com.example.android.Notepad

Un elemento visual que es muy utilizado en la navegación de las aplicaciones es el menú, que se muestra al pulsar la tecla física de menú presente en los distintos dispositivos. *Android* maneja a los ítems del menú de manera diferente que el resto de las vistas. De hecho, los ítems no son vistas. El sistema mantiene jerarquías de clases separadas para manejar las vistas y los ítems del menú. Éstos últimos no derivan de android.view.View sino que implementan la interfaz android.view.MenuItem. Por otro lado, éstos definen las opciones a mostrar en el menú pero no son las vistas que se grafican en el mismo, es decir, el usuario no pulsa sobre un android.view.MenuItem, sino sobre una vista que se grafica al momento de mostrar el menú, y que posee la información definida en éste. Dicha información puede ser el texto a mostrar, un ícono relacionado y una determinada acción a realizar al momento de ser seleccionado.

Los ítems del menú, al igual que las vistas, pueden ser definidos en el código fuente de las actividades, o en archivos XML ubicados en el directorio res/menu.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- This is our one standard application action (creating a new note). -->
    <item android:id="@+id/menu_add"
          android:icon="@drawable/ic_menu_compose"
          android:alphabeticShortcut='a'
          android:title="@string/menu_add" />
</menu>
```

Figura 2.2: XML que define los ítems del menú de la actividad NotesList, de la aplicación com.example.android.Notepad

Tanto las vistas como los ítems del menú pueden contener un identificador alfanumérico único global a toda la aplicación, que hace posible su acceso y manipulación de manera directa. Éste puede ser definido también en los archivos de configuración XML utilizando el atributo android:id="@+id/". En la figura 2.1 y la figura 2.2 puede observarse su utilización. Por ejemplo, el único ítem del menú definido en la figura 2.2 tiene a *menu_add* como identificador global.

Otro aspecto a considerar de las aplicaciones *Android* es que el texto de la interfaz gráfica está separada de la misma. Es decir, las cadenas de texto que se visualizan en la interfaces gráficas no se encuentran definidas en los mismos archivos que definen a éstas últimas, sino que se definen en archivos XML ubicados en el directorio res/values. Esta separación hace posible las traducciones de una aplicación a diferentes lenguajes.

Un punto que cabe destacar en cuanto a la navegación de las aplicaciones se basa en el funcionamiento del botón físico *back* (que en las últimas versiones del SDK de *Android* se vio reemplazado por nuevos controles táctiles ilustrados en las pantallas de los dispositivos tales como el ActionBar). La idea es que siempre que se oprima este botón se retorne a la actividad previa, o se salga de la aplicación si no existe tal actividad.

Para ejemplificar los aspectos de las aplicaciones *Android* recientemente explicados, tomaremos como ejemplo a la aplicación Notepad (com.example.android.Notepad), que es una aplicación de ejemplo que viene incluída en el SDK [1] de *Android*.

Esta aplicación se compone de 4 actividades:

- NotesList: Muestra la lista de las notas existentes. La vista principal de esta actividad es una *android.widget.ListView*. Desde esta actividad podremos llegar a la actividad *NoteEditor*, pulsando sobre la única opción del menú, *Add Note*, o pulsando sobre un ítem de la lista de notas. También podremos acceder a *TitleEditor* a través del menú contextual que se muestra al realizar un click largo sobre alguna nota.
- NoteEditor: Permite escribir o modificar una nota. Está compuesta por una vista de tipo *com.example.android.notepad.NoteEditor$LinedEditText*. Desde esta actividad podremos regresar a *NotesList* o acceder a *TitleEditor* si pulsamos sobre *Edit title*.
- TitleEditor: Es la actividad que permite editar el título de una nota. Está compuesta por vistas de *tipo android.widget.EditText* y *android.widget.Button*.
- NotesLiveFolder: Esta actividad no tiene una interfaz gráfica definida, ya que se usa para manejar el proveedor de contenidos asociado a las notas, por ende no hay forma de acceder visualmente a esta actividad a través de gestos táctiles en otras actividades.

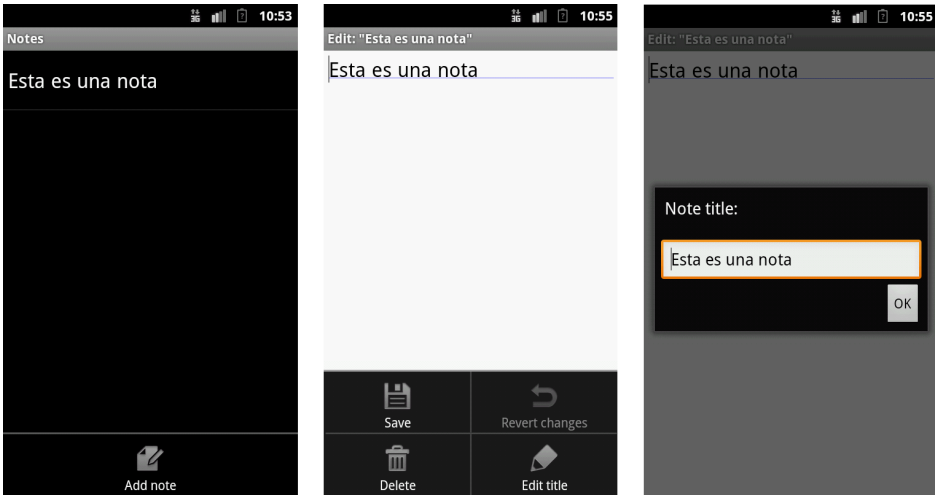


Figura 2.3: Las actividades NotesList, NoteEditor, TitleEditor de la aplicación com.example.android.Notepad

2.2. El framework de testing

El *framework* de testing nativo de *Android* [2] provee la arquitectura y las herramientas necesarias para testear los componentes de una aplicación en diferentes niveles: desde el testing de unidad de un componente específico, hasta el testing de una actividad manejado por su interfaz gráfica. Dicha arquitectura basa su funcionamiento en el *framework* de testing JUnit 3 [3]. Además se provee un generador de eventos de usuario pseudo-aleatorios que simulan ser clicks, toques y otros gestos, llamado Application Exerciser Monkey [4], que permite realizar pruebas de *stress* sobre aplicaciones.

Ahora bien, para poder escribir tests que interactúen con una aplicación por medio de su interfaz gráfica, que es a lo que estamos abocados en este trabajo, debemos utilizar las herramientas provistas en el *framework* de instrumentación de *Android*, ya que es el único capaz de enviar eventos a dicha interfaz.

Ya hemos visto la estructura básica de las aplicaciones, por lo que sabemos que nuestros tests deberán basarse en acciones que tengan impacto sobre las vistas de las distintas actividades incluídas en éstas. El framework de instrumentación de Android provee el soporte mediante el cual podemos realizar este tipo de tests. Supongamos que nos encontramos en el caso de querer testear una aplicación A en base a su interfaz gráfica. En primer lugar, debemos instalar A en un dispositivo o emulador, y luego a través del *framework* de instrumentación de *Android*, podremos generar una aplicación de instrumentación que contenga los tests que queremos ejecutar en la aplicación bajo prueba A. Por lo tanto tendremos que instalar 2 aplicaciones. En primer lugar, A, la aplicación bajo prueba, y en segundo lugar la aplicación de instrumentación que hace referencia a A. Ésta última es un tipo de aplicación especial, que carece de interfaz gráfica y sólo contiene el código de los tests que se quieren ejecutar sobre A.

Para hacer posible la generación de las aplicaciones de instrumentación, el *framework* correspondiente provee la clase *android.test.ActivityInstrumentationTestCase2*, entre otras, pero en el caso de nuestra herramienta, ésta es la clase que más se adapta a nuestros requerimientos. Ésta tiene ciertas limitaciones importantes. En primer lugar, y tal como se menciona en su documentación, dicha clase permite realizar el testing funcional de una sóla actividad. En nuestro caso, este hecho representa una gran limitación, ya que deseamos poder generar tests que no se restrinjan sólo a una actividad, sino que contengan acciones que impacten sobre las diferentes actividades de la aplicación bajo prueba. En segundo lugar, podríamos llegar a manipular vistas a las cuales el usuario no tiene acceso. Por ejemplo, supongamos que tenemos una vista de tipo *ScrollView* (una vista cuyas dimensiones son mayores que la dimension visible de la pantalla e incluye barras de desplazamiento de manera de poder visualizar el contenido oculto) que incluye distintas vistas que ocupan un tamaño mayor al del alto de la pantalla. En este caso, ciertas vistas sólo serán accesibles en el momento en que el usuario mueva hacia abajo la barra de desplazamiento vertical, pero la clase mencionada anteriormente no provee este tipo de soporte, hecho por el cual podríamos llegar a generar tests que representen ejecuciones de la aplicación bajo prueba que no fueran posibles en la realidad.

Capítulo 3

Construcción

Tal como se mencionó anteriormente, nuestro objetivo es construir una herramienta que dada una aplicación *Android* la ejecute, en un dispositivo real o en un emulador, con el fin de generar trazas de manera aleatoria, conozca si durante la ejecución de dichas trazas se produjeron excepciones o errores sin manejar, y las retorne de manera de que sirvan como tests de regresión. Es importante destacar que nuestra herramienta no retornará sólo una traza, sino que retornará todas las posibles trazas generadas de manera aleatoria acotadas por un tamaño dado. En este capítulo introduciremos los algoritmos utilizados, los aspectos formales que los fundamentan, y estableceremos una notación clara para los mismos.

3.1. Motivación del modelo

El sistema *Android* [1] es un sistema reactivo, y como tal, mantiene una continua interacción con su entorno, respondiendo ante los estímulos externos, generados por el usuario o las diversas aplicaciones que se ejecutan, en función de su estado interno. Lo que nos proponemos hacer es simular la interacción que un usuario real podría tener con una aplicación ejecutando en dicho sistema, y a partir de esto poder encontrar excepciones o errores sin manejar en la misma. El usuario interactúa con una aplicación a través de diferentes acciones táctiles (ingresar texto, cliquear, deslizar, etc...) que éste puede realizar sobre la pantalla del dispositivo que la ejecuta. Entenderemos a una **traza como una secuencia de acciones de usuario sobre las vistas** de las diferentes actividades de una aplicación.

Con el fin de modelar el conjunto de posibles trazas de una aplicación, tomemos como ejemplo a AndroidCalculator (`https://robotium.googlecode.com/files/AndroidCalculator-V1_0.apk`). Esta es una aplicación de prueba que pertenece al *framework* de testing Robotium [5]. Esta aplicación se compone de una única actividad y ésta a su vez se compone de diversas vistas, siendo las principales 2 EditText y 1 Button (tal como se puede visualizar en la figura 3.1).

A continuación se listan algunas trazas de dicha aplicación. Denotaremos a la aplicación de una acción sobre una vista como $\langle Accion, \langle IdVista, TipoVista\rangle\rangle$:

- $[start, \langle enterText, \langle 0, EdiText\rangle\rangle, \langle enterText, \langle 1, EditText\rangle\rangle, \langle click, \langle 2, Button\rangle\rangle]$
- $[start, \langle enterText, \langle 0, EdiText\rangle\rangle, \langle click, \langle 2, Button\rangle\rangle]$
- $[start, \langle click, \langle 2, Button\rangle\rangle]$

Tal como vemos en las primeras dos trazas, diferentes trazas pueden contener un inicio común. Con el objetivo de reutilizar dicha información común en la generación de todas las posibles trazas acotadas en tamaño (en base a la cantidad de acciones de usuario), modelaremos el conjunto de trazas en forma de árbol. Se debe tener en cuenta que cada traza es una secuencia de acciones de usuario y no un árbol de acciones. Solo utilizamos un arbol para representar un conjunto de trazas acotado en altura y/o anchura, fusionando prefijos de acciones comunes entre las diferentes trazas de manera de reutilizarlos en la generación de las siguientes. Este hecho es solo una mejora en el proceso de generación, ya que otro algoritmo podría generar trazas al azar y luego compararlas con las ya generadas a fin de almacenarlas si es que no habían sido previamente generadas. En el ejemplo en cuestión, una posible representación arbórea de las trazas de la aplicación esta definida en la figura 3.2.

En un principio, la primera traza puede ser generada al azar (traza principal), pero las subsiguientes comenzarán desde el inicio (start), o bien serán ramificaciones de trazas ya existentes. En dicho árbol pueden observarse en color verde la traza principal, y en amarillo y en azul dos trazas "hijas" o ramificaciones.

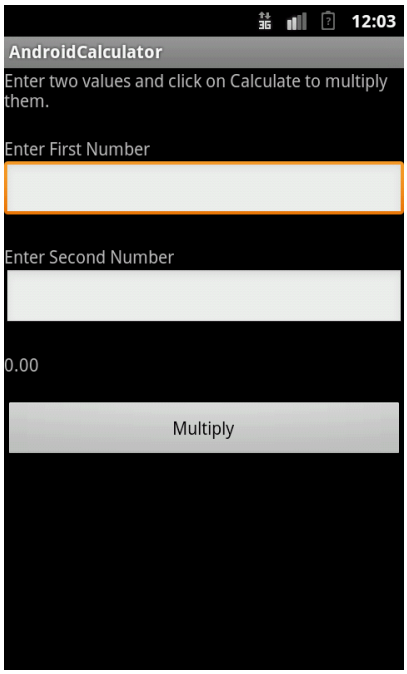


Figura 3.1: La única actividad de la aplicación AndroidCalculator

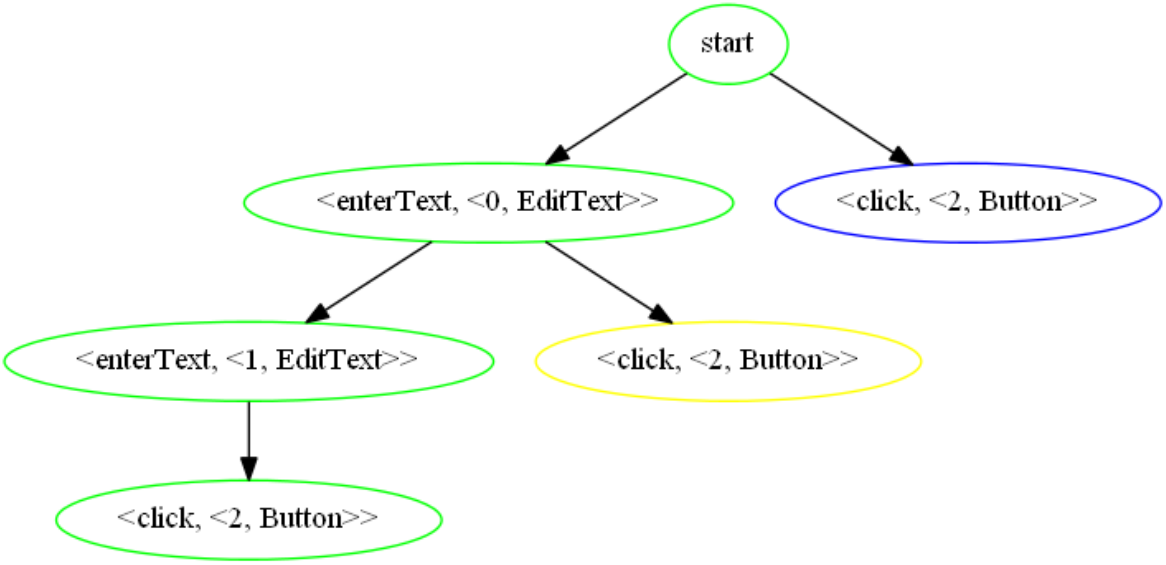


Figura 3.2: Una sección del árbol de trazas de la aplicación AndroidCalculator

3.2. Modelo formal

Cada actividad de una aplicación está compuesta por vistas de diferentes tipos. Definiremos $ViewType$ como el tipo de la vista, donde $ViewType \in \{Button, ImageButton, TextView, ListView, GridView, EditText, ...\}$.

Sobre cada vista de un determinado tipo se pueden realizar diferentes tipos de acciones. Definiremos $ActionType$ como el tipo de acción que se puede realizar sobre una vista, donde $ActionType \in \{click, longClick, swipe, enterText, copy, paste, goBack, ...\}$.

Una vista es identificable unívocamente dentro de una misma actividad. Por lo cual definimos a una vista como $View$ donde éste sirve como identificador único. Para fines prácticos $View$ podría ser un Int, $String$, o cualquier tipo que sirva como identificador.

Definimos $Type : View \rightarrow ViewType$ como una función de mapeo que dado un $View$ permite conocer su $ViewType$.

Definimos $Actions : ViewType \rightarrow \{ActionType\}$ como una función de mapeo que mapea un $ViewType$ con sus respectivos $ActionType$. Es decir, esta función permite conocer el conjunto de las acciones que se pueden realizar sobre una vista de un tipo dado.

Una actividad es identificable unívocamente dentro de una misma aplicación. Por lo cual definimos a una actividad como $Activity$ donde éste sirve como identificador único. Para fines prácticos $Activity$ podría ser un Int, $String$, o cualquier tipo que sirva como identificador.

Definimos $Views : Activity \rightarrow \{View\}$ como una función de mapeo que dado un $Activity$ permite conocer sus respectivas $View$.

La aplicación de una acción sobre una vista particular se define como $Step = \langle action, view \rangle$, donde:

- $action : ActionType$ es el tipo de acción que se realiza.
- $view : View$ es la vista sobre la cual se realiza la acción.

Con el fin de destacar el comienzo de cada traza "hija" o ramificación, se introducen los puntos de ramificación o $BranchPoints$. Definiremos $BranchPoint$ como un renombre de Int.

En base a anteriores definiciones, definiremos a una traza como $Trace = \langle Steps, Branchpoints, ChildTraces, error \rangle$, donde:

- $Steps : [Step]$ es la lista de pasos ejecutados en la traza.
- $BranchPoints : \{BranchPoint\}$ es el conjunto de posiciones de comienzo de las ramificaciones.
- $ChildTraces : \{Trace\}$ es el conjunto de ramificaciones de la traza.
- $error : BOOL$ indica si la traza terminó su ejecución con error.

Definimos a una aplicación como $Application = \langle Activities, currentActivity \rangle$, donde:

- $Activities : \{Activity\}$ es el conjunto de actividades de la aplicación.
- $currentActivity : Activity$ es la actividad actual.

El siguiente algoritmo construye una traza a medida de que va explorando la interfaz gráfica de la aplicación. Cada acción sobre una vista se modela con un *Step*. Se debe tener en cuenta de que cada acción puede desencadenar cambios en la interfaz gráfica, por lo que en cada iteración se la debe volver a escanear. Tanto la acción a ejecutar como la vista en la cual se ejecuta son seleccionadas al azar basándose en un *random seed*. De esta manera se garantiza que la construcción de trazas se realiza de manera aleatoria. En cada paso, a medida de que se recorre la aplicación, si se encuentra más de una acción a realizar o más de una vista en donde realizarla, el algoritmo genera un nuevo *BranchPoint*, para indicar que en dicho punto puede comenzar una nueva ramificación. En caso de encontrar un error en la ejecución de determinado *Step*, el algoritmo setea a la traza generada como fallida y retorna. Se debe tener en cuenta que cada acción contiene todos los parámetros necesarios para su ejecución.

```
1:  EXPLORE(in app : Application, in maxTraceSize : Int, inout trace : Trace)
2:      for i in [1..maxTraceSize] do
3:          # Se obtienen las vistas de la actividad actual
4:          currentViews ← Views(app.currentActivity)
5:          if currentViews = ∅
6:              # Al no haber vistas se retorna
7:              return
8:
9:          # Se selecciona una vista y una acción sobre la misma
10:         action ← nil
11:         while action = nil
12:             # Se obtiene una vista al azar
13:             view ← v ∈ currentViews
14:             # Se obtienen las acciones que se pueden ejecutar sobre la vista seleccionada
15:             runnableActions ← Actions(Type(view))
16:             # Se obtiene una accion al azar
17:             action ← a ∈ runnableActions
18:
19:         # Se genera un nuevo step
20:         currentStep ← ⟨action, view⟩
21:         pos ← |trace.Steps|
22:         # Se agrega el nuevo step al final de la lista de steps de la traza actual
23:         add(trace.Steps, currentStep)
24:
25:         if #(runnableActions) > 1 or #(currentViews) > 1
26:             # Se almacena un nuevo BranchPoint en caso de poder realizar
27:             # mas de una acción o de haber mas una vista para utilizar
28:             trace.BranchPoints ← trace.BranchPoints ∪ {pos}
29:
30:         # Se ejecuta el step recién creado
31:         RUNSTEP (app, currentStep, failed)
32:         if failed
33:             # Se retorna en caso de que la ejecucion del step falle
34:             trace.error ← true
35:             return
```

Algoritmo 3.1: Construye una traza de manera aleatoria a medida de que recorre la interfaz gráfica

A continuación se declara la aridad de un procedimiento necesario para interactuar directamente con el dispositivo (o emulador) que ejecuta la aplicación. El mismo debe ejecutar el *Step* especificado, indicar si su ejecución fue satisfactoria o se detectó un

error, y eventualmente actualizar la actividad actual (app.currentActivity).

`RUNSTEP`(in $app : Application$, in $step : Step$, out $failed : Bool$)

El algoritmo 3.2 es el algoritmo principal de nuestra herramienta. El mismo genera de manera aleatoria y exhaustiva todas las posibles trazas de la aplicación acotadas en tamaño (definido en base a su cantidad de *Steps*). Debido a su construcción el mismo no retorna trazas inválidas, es decir, trazas que no puedan ser obtenidas mediante la ejecución real de la aplicación, ya que de encontrar un error, la traza generada no se sigue explorando. En primer lugar, se llama a EXPLORE de manera de generar aleatoriamente una traza de tamaño $maxTraceSize$ como máximo. Ésta será considerada la traza principal. Luego, se procederá a generar las ramificaciones en base a los $BranchPoints$ encontrados. Como se explicó anteriormente un $BranchPoint$ define el comienzo de una ramificación. Llamaremos traza "padre" a una traza que contiene al menos un $BranchPoint$, y traza "hija", sub-traza o ramificación a las trazas que comienzan en dichos $BranchPoints$. Por lo tanto, para generar una ramificación, debemos primero ejecutar los *Step* de su traza "padre" previos al correspondiente $BranchPoint$, en segundo lugar debemos seleccionar un *Step* que no haya sido ejecutado en la posición definida en el $BranchPoint$ y no pertenezca a la traza "padre" o a las ramificaciones pertenecientes al mismo $BranchPoint$, y por último, seguir ejecutando desde allí, de manera aleatoria, hasta que el tamaño de la misma sea $maxTraceSize$ o se encuentre un error.

Con el fin de que todas las trazas pertenezcan al mismo árbol, introduciremos un nuevo tipo de $Step = start$.

El algoritmo 3.3 retorna, en caso de existir, un *Step* válido en la aplicación, que no haya sido ejecutado en la posición definida en el $BranchPoint$ especificado y no pertenezca a la traza "padre" o a las ramificaciones pertenecientes al mismo $BranchPoint$.

Notar que la terminación de nuestro algoritmo radica en la ejercitación de todos los $BranchPoints$ encontrados. Como cada uno de estos es un punto de ramificación, al generar todas las posibles ramificaciones para cada uno de éstos, se termina completando el árbol de trazas. Que el algoritmo 3.3 ya no pueda retornar un *Step* no ejecutado en el $BranchPoint$ dado indica que las ramificaciones correspondientes ya fueron generadas, razón por la cual se procede a eliminar dicho $BranchPoint$. Una vez eliminados todos los $BranchPoints$ encontrados el algoritmo termina.

```
1:  TEST(in app : Application, in maxTraceSize : Int, out traces : {Trace})
2:          # Se inicializan el conjunto de trazas y la traza principal
3:          traces ← {}
4:          startTrace ← ⟨[start] , {} , {} , false⟩
5:
6:          # Se construye la traza principal de manera aleatoria
7:          EXPLORE(app, maxTraceSize, startTrace)
8:          traces ← traces ∪ {startTrace}
9:
10:         # Se computa el conjunto de trazas incompletas
11:         incompleteTraces ← {t ∈ traces | t.BranchPoints ≠ ∅}
12:
13:         while incompleteTraces ≠ ∅ do
14:                 # Se selecciona una traza incompleta
15:                 trace ← t ∈ incompleteTraces
16:                 traces ← traces − {trace}
17:                 # Se selecciona un BranchPoint
18:                 branch ← b ∈ trace.BranchPoints
19:
20:                 # Se ejecutan los Steps de la traza hasta el BranchPoint seleccionado
21:                 for each step in trace.Steps [0..branch) do
22:                         RUNSTEP(app, step, failed)
23:
24:                 # Se busca un Step no ejecutado en la posición definida en el BranchPoint
25:                 FINDNEWSTEP(app, trace, branch, newStep, found)
26:
27:                 if !found
28:                         # Al no encontrar un nuevo Step se elimina el BranchPoint seleccionado
29:                         trace.BranchPoints ← trace.BranchPoints − {branch}
30:                 else
31:                         # Si existe un nuevo Step lo ejecutamos
32:                         RUNSTEP(app, newStep, failed)
33:
34:                         # Como existe un nuevo Step, existe una nueva ramificación
35:                         # Se la crea tomando como base los Steps de su traza padre.
36:                         steps ← [trace.Steps_0...trace.Steps_branch − 1]
37:                         steps_branch ← newStep
38:                         childTrace ← ⟨steps, {} , {} , failed⟩
39:                         trace.ChildTraces ← trace.ChildTraces ∪ {childTrace}
40:
41:                         if !failed
42:                                 # Se sigue construyendo la ramificación desde el BranchPoint
43:                                 EXPLORE(app, maxTraceSize − branch, childTrace)
44:
45:                         # Se agrega la ramificación al conjunto de trazas
46:                         traces ← traces ∪ {childTrace}
47:
48:                 # Se agrega la traza modificada al conjunto de trazas
49:                 traces ← traces ∪ {trace}
50:                 # Se computa el conjunto de trazas incompletas
51:                 incompleteTraces ← {t ∈ traces | t.BranchPoints ≠ ∅}
```

Algoritmo 3.2: Algoritmo principal. Genera todas las trazas de una aplicación de manera exhaustiva (acotadas en tamaño)

```
1:  FINDNEWSTEP(in app : Application, in trace : Trace, in branch : BranchPoint, out step : Step,
    out found : Bool)
2:          # Se obtiene el Step de la traza padre ejecutado en el BranchPoint
3:          parentTraceStep ← trace.Steps_branch
4:          # Se obtienen los Steps de las ramificaciones de trace ejecutados en el mismo BranchPoint
5:          childTracesSteps ← {t.Steps_branch | t ∈ trace.ChildTraces}
6:          # Se obtiene el conjunto de todos los Steps ejecutados en el mismo BranchPoint
7:          executedSteps ← childTracesSteps ∪ {parentTraceStep}
8:
9:          # Se obtiene el conjunto de todos los posibles Steps a ejecutar en la actividad actual
10:         allSteps ← {⟨a, v⟩ | v ∈ Views(app.currentActivity) ∧ a ∈ Actions(Type(v))}
11:
12:         # Se obtiene el conjunto de todos los Steps todavía no ejecutados
13:         newSteps ← allSteps − executedSteps
14:
15:         # En caso de existir un nuevo Step, se retorna
16:         if newSteps ≠ ∅
17:                 found ← true
18:                 step ← s ∈ newSteps
19:         else
20:                 found ← false
```

Algoritmo 3.3: Si existe, retorna un *Step* no ejecutado en el *BranchPoint* dado, teniendo en cuenta las trazas ya generadas

Capítulo 4

Implementación

En este capítulo se introducen la arquitectura y detalles de implementación de nuestra herramienta *Android Test Generator* (de ahora en más *ATG*), que genera casos de test de una aplicación *Android* en base a la interacción mediante su interfaz gráfica, e implementa los algoritmos especificados en el capítulo anterior. Por otro lado, se enumeran y explican todas las acciones que la herramienta realiza, más allá de la ejecución propiamente dicha de los algortimos mencionados, con el fin de poder computarlos. Así mismo, se introducen las diferentes limitaciones encontradas a la hora de la implementación, y ciertas decisiones por las cuales algunos de sus efectos pudieron ser mitigados.

4.1. Arquitectura y componentes

La arquitectura básica de *ATG* se describe en la figura 4.1. La misma se divide en dos grandes secciones. Una de ellas se centra en los componentes incluídos en un entorno Java, mientras que la otra describe los componentes que corren sobre el entorno *Android*, ya sea en un dispositivo real o un emulador.

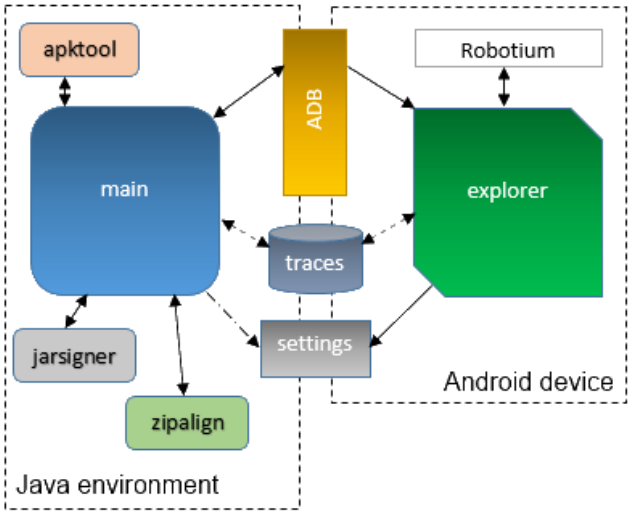


Figura 4.1: Arquitectura de *ATG*

El componente *explorer* es una aplicación de instrumentación Android. La misma contiene la implementación de los algoritmos $EXPLORE$, $FINDNEWSTEP$ y gran parte de $TEST$ definidos en la sección anterior, es decir, se encarga de gran parte de la exploración y generación de los casos de test de la aplicación bajo prueba. Debido a que las aplicaciones de instrumentación se componen de tests *Android* (que derivan de tests *JUnit*), los algoritmos de la aplicación *explorer* son a su vez casos de test y se encuentran codificados y se ejecutan como tales. Dicho de otro modo, al ejecutarse los casos de test incluídos en *explorer* se generan los casos de test de la aplicación bajo prueba.

Como se explicó anteriormente, el *framework* de instrumentación nativo de *Android* tiene ciertas limitaciones que atentan contra la realización y el desempeño de nuestra herramienta. Es por ello que decidimos incluir *Robotium* [5], un *framework* de testing que corre por sobre el *framework* de instrumentación nativo de *Android*, y que si bien mantiene ciertas limitaciones, resuelve otras que hacen posible el funcionamiento de nuestra herramienta. Entre éstas últimas se encuentra la posibilidad de testear la aplicación bajo prueba a través de múltiples actividades (tal como lo hace un usuario a la hora de utilizarla). Por otro lado, provee el acceso y manipulación de las vistas de las diferentes actividades, permitiendo obtener las que un usuario final puede visualizar en particular, lo que aporta a la idea de generar tests que sean factibles en un entorno real.

El *framework* de instrumentación de *Android* provee *ADB* (Android Debug Bridge) [6], una pieza clave para el funcionamiento de *ATG*, más específicamente, para permitir la interacción entre un dispositivo (o emulador) y un programa externo. En nuestro caso, *main*, la aplicación Java de *ATG*, utiliza *ADB* para enviar y recibir información y comandos desde y hacia el dispositivo en el que se corre la aplicación bajo prueba.

Por otro lado, *main* y *explorer* comparten ciertos archivos comunes tales como "settings", un XML en donde el usuario define sus configuraciones para la generación, y "traces", una base de datos sqlite [7] en donde se van almacenando los resultados.

El componente *main* es un recurso ejecutable Java, con el que el usuario final interactúa, que se encarga de tomar todos los parámetros y de realizar los pasos previos necesarios para dar lugar a la generación de los casos de test sobre la aplicación bajo prueba. Por otro lado, también es el componente encargado de controlar el proceso de generación propiamente dicho.

Apktool [8] es una herramienta que permite decodificar y compilar apks. El componente *main* lo utiliza para obtener el AndroidManifest.xml, que especifica las actividades que componen a la aplicación, los archivos XML que definen interfases gráficas (res/layout), cadenas de texto (res/values), y otra información como los identificadores globales de la aplicación bajo prueba. Esta información es útil tanto para testear la aplicación como a la hora de realizar un análisis de cobertura.

Jarsigner [9] es una herramienta que se utiliza para firmar las aplicaciones *Android*, de manera de que pasen las validaciones de seguridad de la plataforma y puedan correr en los distintos dispositivos o emuladores. Es utilizada por *main* para firmar tanto la aplicación bajo prueba como la aplicación de instrumentación, ya que ambas deben correr en los dispositivos.

Otro de los pasos necesarios a la hora de correr una aplicación *Android* en un dispositivo es la ejecución de *zipalign* [10]. Esta herramienta realiza ciertas optimizaciones

de código y del paquete ya firmado que son necesarias para su posible ejecución. Nuevamente *main* hace uso de esta herramienta como paso previo a la ejecución de los algoritmos anteriormente mencionados.

4.2. Configuración previa

A continuación se enumeran los pasos más destacados realizados por *main*, luego de recibir los parámetros necesarios, con el fin de preparar el entorno para la generación de las trazas de la aplicación:

1. Inicia el servidor de *ADB*.
2. A través de *ADB* se obtiene la información de los distintos dispositivos conectados. En caso de no encontrarse ninguno, el programa finaliza su ejecución notificando esta situación. En caso de haber más de un dispositivo conectado *main* le solicita al usuario que especifique cual de éstos desea utilizar.
3. En caso de que corresponda, se cargan las configuraciones de usuario del correspondiente archivo XML.
4. Se decodifica el apk especificado a través de la utilización de apktool, con el fin de obtener su correspondiente *AndroidManifest.xml* y conocer información acerca de las actividades, vistas e ítems del menú de la aplicación bajo prueba.
5. Se obtiene el paquete y la actividad principal de la aplicación bajo prueba. Ésta actividad es el punto de entrada de la aplicación, es decir, la que se inicia cuando el usuario ejecuta la aplicación desde el menú principal, y está asociada a la acción *android.intent.action.MAIN* y a la categoría *android.app.category.LAUNCHER* en el *AndroidManifest.xml*. Esta información es realmente importante ya que es requerida por el *framework* de instrumentación para que la correspondiente aplicación de test logre vincularse con la aplicación bajo prueba.
6. Para que *explorer* y *main* puedan compartir información a través de la base de datos sqlite "traces" es necesario que la aplicación bajo prueba (*AUT*) tenga permisos de escritura en el sistema de archivos del dispositivo, ya que *explorer* corre en el entorno de la *AUT*. En caso de no tener dichos permisos, *main* modifica el *AndroidManifest.xml* de la *AUT* para otorgárselos.
7. Se configura *explorer* para que corra apuntando a la *AUT*.
8. Todas las aplicaciones *Android* deben estar firmadas digitalmente para poder ser ejecutadas en un dispositivo. Para que una aplicación de instrumentación pueda vincularse con su aplicación bajo prueba es necesario que ambas estén firmadas con la misma clave. Por lo tanto, se regenera el apk de la *AUT* con los cambios especificados, a través de la utilización de apktool y luego se la firma con jarsigner. De la misma manera, luego de compilar la aplicación de instrumentación con ANT, también se la firma con jarsigner.
9. Se ejecuta zipalign sobre el nuevo apk de la *AUT*. Este programa provee ciertas optimizaciones que hacen posible la ejecución de la aplicación en un dispositivo.
10. A través de *ADB* se instalan en el dispositivo tanto el apk resultante de la *AUT* como el de la aplicación de instrumentación *explorer*.
11. Se envía una copia de la base de datos "traces" al dispositivo, a través de *ADB*.

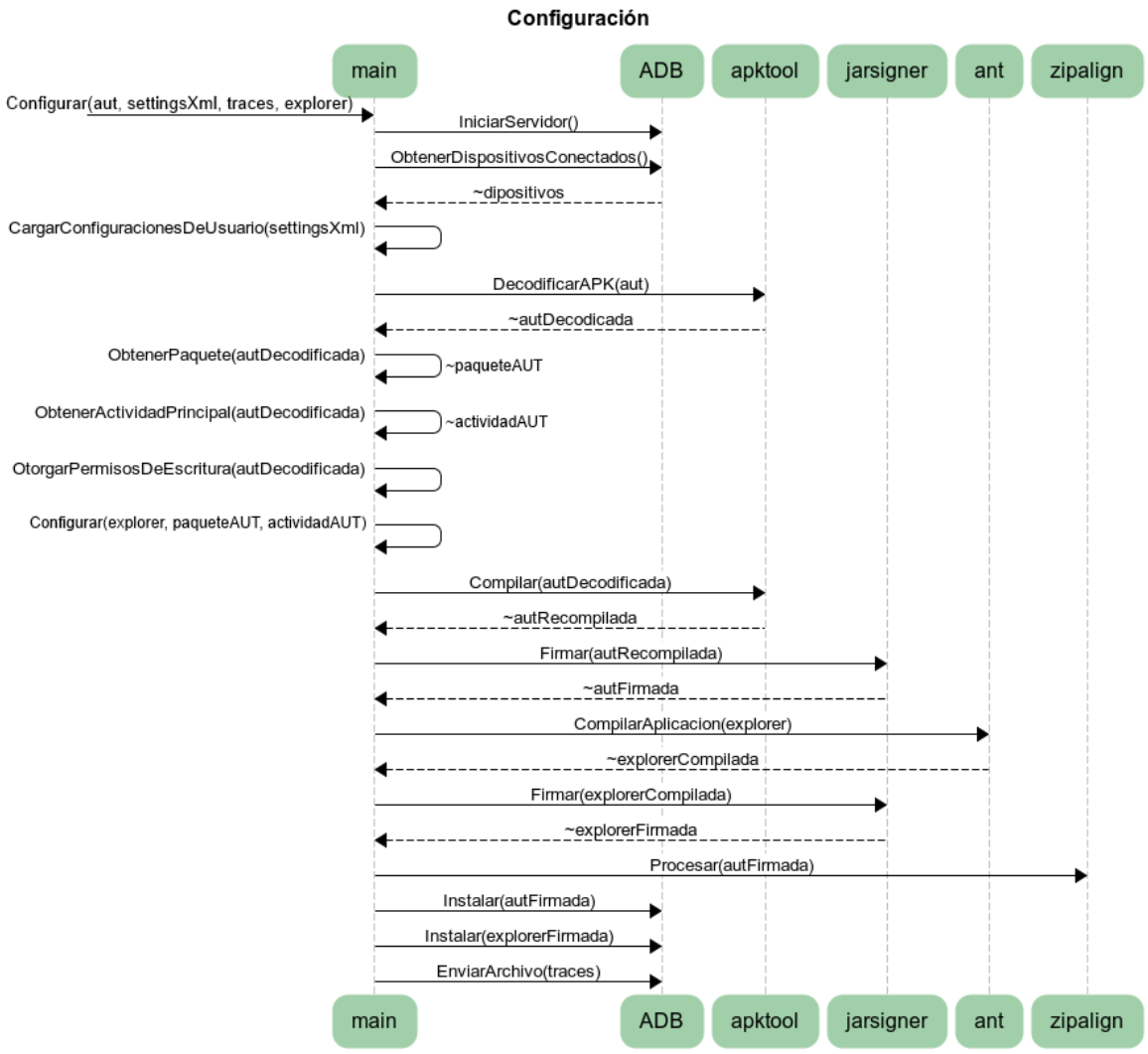


Figura 4.2: Preparación del entorno para la generación de trazas

Luego de realizar los pasos anteriores, *main* ejecuta a través de *ADB* el test *Android* incluido en *explorer*. Éste test toma a la aplicación bajo prueba recién instalada y ejecuta los algoritmos $TEST$, $EXPLORE$ y $FINDNEWSTEP$. Cada ejecución de *explorer* genera una nueva traza que se almacena en la base de datos "traces" junto con todos sus *BranchPoints*. Luego de la ejecución de *explorer*, *main* chequea si existen *BranchPoints* no completos, caso en el cual sigue vuelve a ejecutar *explorer*. La ejecución de *main* culmina al no encontrar *BranchPoints* incompletos, es decir, luego de haber explorado todo el árbol de trazas de la *AUT* (acotado en altura o anchura, según corresponda).

Una vez concluída la generación de trazas, *main* configura un nuevo proyecto de instrumentación de *Android*, de manera de que cada una de las trazas generadas pasa a ser un test que luego el usuario final puede correr o almacenar como test de regresión. Luego de la finalización de *main*, el proyecto resultante se encontrará en la carpeta de salida especificada por el usuario.

4.3. Análisis de cobertura y mapa de la interfaz de usuario

Con el fin de medir la eficiencia de los tests generados, introducimos en nuestra herramienta un análisis de cobertura. Ahora bien, debido a que nuestra herramienta no accede al código fuente de las aplicaciones bajo prueba, el análisis de cobertura propuesto se basa en los componentes gráficos (actividades, vistas e ítems del menú) de las mismas. En primer lugar se realiza un análisis estático de los recursos que definen la interfaz gráfica de la aplicación bajo prueba, y luego, en base a las trazas generadas, podemos conocer...

- Las actividades visitadas y no visitadas.
- Las vistas ejercitadas y no ejercitadas.
- Los ítems del menú ejercitados y no ejercitados.

Ahora bien, si sólo contáramos, por ejemplo, con las vistas visitadas no podríamos hacer análisis de cobertura alguno. Por ende, es necesario conocer cierta información de la aplicación bajo prueba antes de comenzar a generar las trazas. Al decodificar la aplicación bajo prueba, mediante la utilización de *apktool*, y a través de un análisis estático, podremos conocer cuales son sus actividades (definidas en el AndroidManifest.xml correspondiente), como están organizadas sus vistas (accediendo a los archivos ubicados en res/layout) y cuales son los ítems del menú con los que nos podremos encontrar en la aplicación (accediendo a los archivos ubicados en res/menu). Por lo que *main* antes de comenzar con la generación almacenará toda esta información acerca de la *AUT*, con el fin de poder luego concluir cuales de éstos recursos fueron visitados o no, y realizar el cálculo de cobertura correspondiente. Para este análisis identificamos a las actividades a través de la clase de la que son instancia, a las vistas en base a su identificador global, y a los ítems del menú en base su texto.

Debido a que en la gran mayoría de las veces las vistas de tipo *TextView* no permiten interacción, y en casi todas las aplicaciones hay una gran cantidad de éstas, se realizan dos cálculos de cobertura distintos para las vistas: En uno se las incluye y en el otro no. Ésto permite tener, por un lado, un cálculo general de todas las vistas, y por el otro uno incluyendo las vistas con las cuales se permite la interacción.

A continuación se encuentra el cálculo correspondiente al porcentaje de cobertura para las actividades de la aplicación bajo prueba.

$$\%\text{ de cobertura de actividades} = \frac{\#\,(\text{Actividades visitadas})}{\#\,(\text{Total de actividades})}$$

La cantidad total de actividades de la *AUT* se obtiene a partir de la inspección de los recursos XML incluidos en el APK de la AUT mediante el análisis estático inicial.

A continuación se encuentra el cálculo correspondiente al porcentaje de cobertura para las vistas de la aplicación bajo prueba.

$$\%\text{ de cobertura de vistas} = \frac{\#\,(\text{Vistas ejercitadas})}{\#\,(\text{Total de vistas})}$$

La cantidad total de vistas de la *AUT* se obtiene a partir del análisis estático inicial (las vistas definidas en los recursos XML), y a partir de las vistas obtenidas dinámicamente

durante el escaneo de la interfaz gráfica en el proceso de generación de trazas. Por lo tanto, se debe tener en cuenta que en la AUT pueden existir ciertas vistas que nuestra herramienta no logre encontrar, por lo que éstas no son tenidas en cuenta en el cálculo de cobertura correspondiente. Es decir, el conjunto total de las vistas de la AUT incluye a las vistas definidas en los recursos XML correspondientes y a las contenidas en las actividades visitadas.

A continuación se encuentra el cálculo correspondiente al porcentaje de cobertura para los ítems del menú de la aplicación bajo prueba.

$$\%\text{ de cobertura de ítems del menú} = \frac{\#\,(\text{Items ejercitados})}{\#\,(\text{Total de ítems del menú})}$$

La cantidad total de ítems del menú de la *AUT* se obtiene a partir del análisis estático inicial (los definidos en los recursos XML), y a partir de los ítems obtenidos dinámicamente durante el escaneo de la interfaz gráfica en el proceso de generación de trazas. Por lo tanto, se debe tener en cuenta que en la AUT pueden existir ciertos ítems del menú que nuestra herramienta no logre encontrar, por lo que éstos no son tenidos en cuenta en el cálculo de cobertura correspondiente. Es decir, el conjunto total de los ítems del menú de la AUT incluye a los ítems definidos en los recursos XML correspondientes y a los contenidos en las actividades visitadas.

Como agregado a éste análisis, nuestra herramienta genera también un mapa de la interfaz de usuario de la aplicación bajo prueba. El mismo permite visualizar de manera gráfica la composición de cada una de las actividades de la aplicación, es decir, las vistas que están incluídas en cada una de éstas, y brinda información acerca de las transiciones entre las mismas, es decir, las acciones sobre vistas que hacen que ocurra un cambio de actividad. Cabe destacar que en las aplicaciones de la plataforma *Android*, la relación entre las vistas y las actividades que las contienen no es fija y se realiza en tiempo de ejecución, hecho por el cual esta información debe ser recabada dinámicamente. Además de reunir la información recién mencionada, en el mapa generado se especifica si las actividades, vistas e ítems del menú fueron visitados o no, y si alguno de éstos es nuevo, es decir, si fue encontrado en el análisis dinámico, y no en el análisis estático inicial.

¿Por qué una vista puede no ser encontrada en el análisis estático inicial y luego ser encontrada mediante el análisis dinámico? Es muy común que mediante los XML que definen el layout de las vistas se definan vistas complejas. Un ejemplo de esto podría ser la definición de una vista que simule un *combobox* o *dropdownlist*. Ahora bien, el programador que definió dicha vista seguramente va a querer reutilizarla en las diferentes actividades de su aplicación y quizás más de una vez en cada una. En este caso, dicha vista personalizada contendrá un ID global en su definición, en el XML correspondiente, y al ser instanciada adquirirá un ID interno de manera de diferenciarla del resto de las instancias del mismo tipo de vista. Debido al hecho de que dicha vista no contiene un ID global estático, nuestro análisis estático la descarta. Luego, de manera dinámica y utilizando el ID que en este momento ya posee cada instancia de cualquier vista de la aplicación (ya sea global o interno), se almacena la ubicación de la vista en la aplicación, es decir, se especifica la actividad en la que está incluída.

La figura 4.3 muestra los resultados del cálculo de cobertura de la aplicación AndroidCalculator mencionada anteriormente. La misma posee una única actividad compuesta por 2 *EditText*, 1 *Button*, y algunos *TextViews*. En cuanto a éstos últimos se pueden ver la diferencia en el cálculo en base a si se los tiene en cuenta o no.

Actividades			
# Total	# Visitadas	# No visitadas	Cobertura
1	1	0	100%

Items del menú			
# Nuevos	# Visitados	# No visitados	Cobertura
0	0	0	100%

Vistas				
# Nuevas	# Visitadas	# No visitadas	Cobertura	Cobertura (sin contar TextViews)
2 TextViews	3 (2 EditText y 1 Button)	3 TextViews	50%	100%

Transiciones entre actividades
No posee

Figura 4.3: Cobertura de la aplicación AndroidCalculator

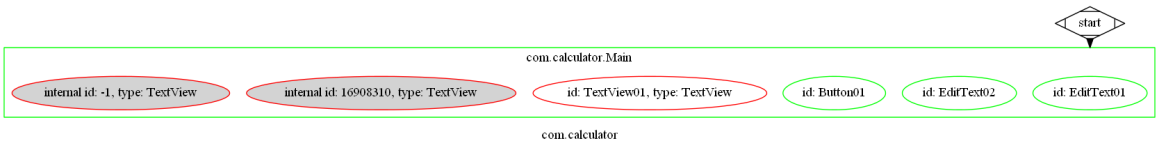


Figura 4.4: Mapa de la interfaz de usuario de AndroidCalculator

En la figura 4.4 se encuentra el mapa de la interfaz gráfica de AndroidCalculator. En la misma se muestran en verde los componentes visitados en los tests generados, y en rojo los no visitados. Por otro lado, el rectángulo grande hace referencia a una actividad, en este caso a *com.calculator.Main*, sú única actividad. Dentro de la misma pueden encontrarse todas las vistas que la componen. Los óvalos hacen referencia a vistas y los rectángulos pequeños a ítems del menú. Como puede verse en el mapa ésta aplicación carece de menú. Los componentes grisados son nuevos, es decir, fueron encontrados en el análisis dinámico pero no el estático. El identificador global único (id) para cada control se muestra, si lo tuviera, dentro de su correspondiente figura. Los componentes creados en tiempo de ejecución carecen de dicho identificador, por lo que su identificador interno (internal id, que es provisto por el entorno *Android*) se muestra en su lugar.

4.4. Limitaciones

En esta sección se encuentran las limitaciones más importantes que surgieron a la hora de implementar nuestra herramienta. La mayoría de ellas se encuentran asociadas al hecho de que tanto el *framework* de instrumentación de *Android* como *Robotium*, que está construído en base a éste, fueron creados para que los desarrolladores puedan realizar tests de sus propias aplicaciones y no en base al propósito de nuestra herramienta: testear cualquier aplicación en base a su interfaz gráfica.

Una limitación importante del *framework* de instrumentación radica en que si por alguna razón, al testear una aplicación, se sale de la misma, por ejemplo por realizar un *back* (mediante la tecla física del dispositivo) o presionando un botón de *salir*, el framework no se da por enterado de este hecho, manteniendo una referencia a la

actividad actual (a la última ejecutada antes de salir). Para superar dicha limitación, nuestra herramienta, antes de que ejecutar cualquier acción, realiza una llamada al sistema de manera de obtener las tareas que se están ejecutando. De éstas, toma la primera actividad, que es la que tiene el foco, y chequea si dicha actividad pertenece al paquete de la aplicación bajo prueba. En dicho caso, ATG concluye que la AUT se sigue ejecutando y continúa con la generación de la traza, y en caso contrario ATG finaliza la generación. Esta solución no es ideal, ya que depende tanto del tiempo de espera entre cada par de acciones, como del tiempo que le insume al dispositivo (o emulador) salir de la aplicación, pero contribuye a bajar el número de falsos positivos (se considera positivo la ocurrencia de un error sin manejar o excepción) encontrados por nuestra herramienta. En base a esta limitación, surgió la idea de que la herramienta permitiera configurar las acciones a realizar con el fin de generar trazas (y eventualmente tests). El usuario final puede configurar *ATG* de manera de que no se utilice el botón físico *back*, haciendo que la cantidad de trazas fallidas generadas por ésta limitación baje considerablemente.

Otra limitación inherente al *framework* de instrumentación radica en que si en el handler *onCreate* de la actividad inicial se crea una nueva actividad que toma el control de la aplicación, el *framework* de instrumentación de *Android* no podrá tomar el control de la ejecución y fallará por *timeout*. Por otro lado, el *framework* finaliza los tests *Android* que llaman a otras aplicaciones. Por ejemplo, si una aplicación hace uso de la aplicación Camera (cámara de fotos), al correr un test sobre la primera, dicho *framework* lo finalizará al momento de abrir la cámara.

En cuanto a *Robotium*, si bien soluciona ciertos problemas y es fácil a la hora de permitir la interacción con los dispositivos, tiene ciertas limitaciones importantes. Por ejemplo, no provee método alguno para conocer cuales son los ítems de menú existentes en una actividad. Por esta razón, nuestra herramienta utiliza mecanismos de reflection con el fin de obtener esta información, ya que el menú es una parte esencial de la interfaz gráfica. Por otro lado, *Robotium* provee soporte limitado para diferentes componentes gráficos de *Android*, y muchas veces es requerida la introducción de *sleeps* en transiciones entre actividades.

Por otro lado, un aspecto limitante en relación a la generación de tests radica en que los datos a ingresar por las distintas acciones (por ejemplo, la cadena que se ingresa en las diferentes cajas de texto de la AUT) son fijos por acción. Con el objetivo de mitigar dicha limitación, nuestra herramienta posibilita la introducción de nuevas acciones por parte del usuario (por ejemplo, se podría generar una nueva acción que ingrese distintas cadenas - o cadenas aleatorias - en las diferentes cajas de texto de la AUT).

4.5. Detalles de implementación

Como se mencionó anteriormente las trazas generadas son almacenadas en la base de datos sqlite *traces* compartida entre *main* y *explorer*. A continuación se encuentra su modelo relacional.
$TRACE(id, traceId, position, failed)$
$STEP(id, traceId, methodName, position, viewId)$
$STEP_PARAMETER(stepId, value, type)$
$BRANCH_POINT(id, traceId, position, tracesCount, completed)$
$UI_ACTIVITY(name, timesVisited)$

$UI_MENU_ITEM(id, title, static, skip)$
$UI_VIEW(id, type, static, skip)$
$UI_ACTIVITY_MENU_ITEM(activityName, itemTitle, timesVisited)$
$UI_ACTIVITY_VIEW(activityName, viewId, timesVisited)$
$UI_ACTIVITY_TRANSITION(toActivityName, fromActivityName, stepId)$
$SETTING(key, value)$

restricciones:

$TRACE.traceId$ está en $TRACE.id$ o es nil
$STEP.traceId$ está en $TRACE.id$
$STEP.viewId$ está en $UI_VIEW.id$
$BRANCH_POINT.traceId$ está en $TRACE.id$
$UI_ACTIVITY_MENU_ITEM.activityName$ está en $UI_ACTIVITY.name$
$UI_ACTIVITY_MENU_ITEM.itemTitle$ está en $UI_MENU_ITEM.title$
$UI_ACTIVITY_VIEW.activityName$ está en $UI_ACTIVITY.name$
$UI_ACTIVITY_VIEW.viewId$ está en $UI_VIEW.id$
$UI_ACTIVITY_TRANSITION.toActivityName$ está en $UI_ACTIVITY.name$
$UI_ACTIVITY_TRANSITION.fromActivityName$ está en $UI_ACTIVITY.name$
$UI_ACTIVITY_TRANSITION.stepId$ está en $STEP.id$

A continuación se encuentran los diagramas de clases principales de nuestra herramienta. El primero incluye el modelo de vistas, el segundo el modelo de acciones (no se detallan todas las acciones soportadas), y el tercero el modelo de trazas incluyendo la interacción con el dispositivo.

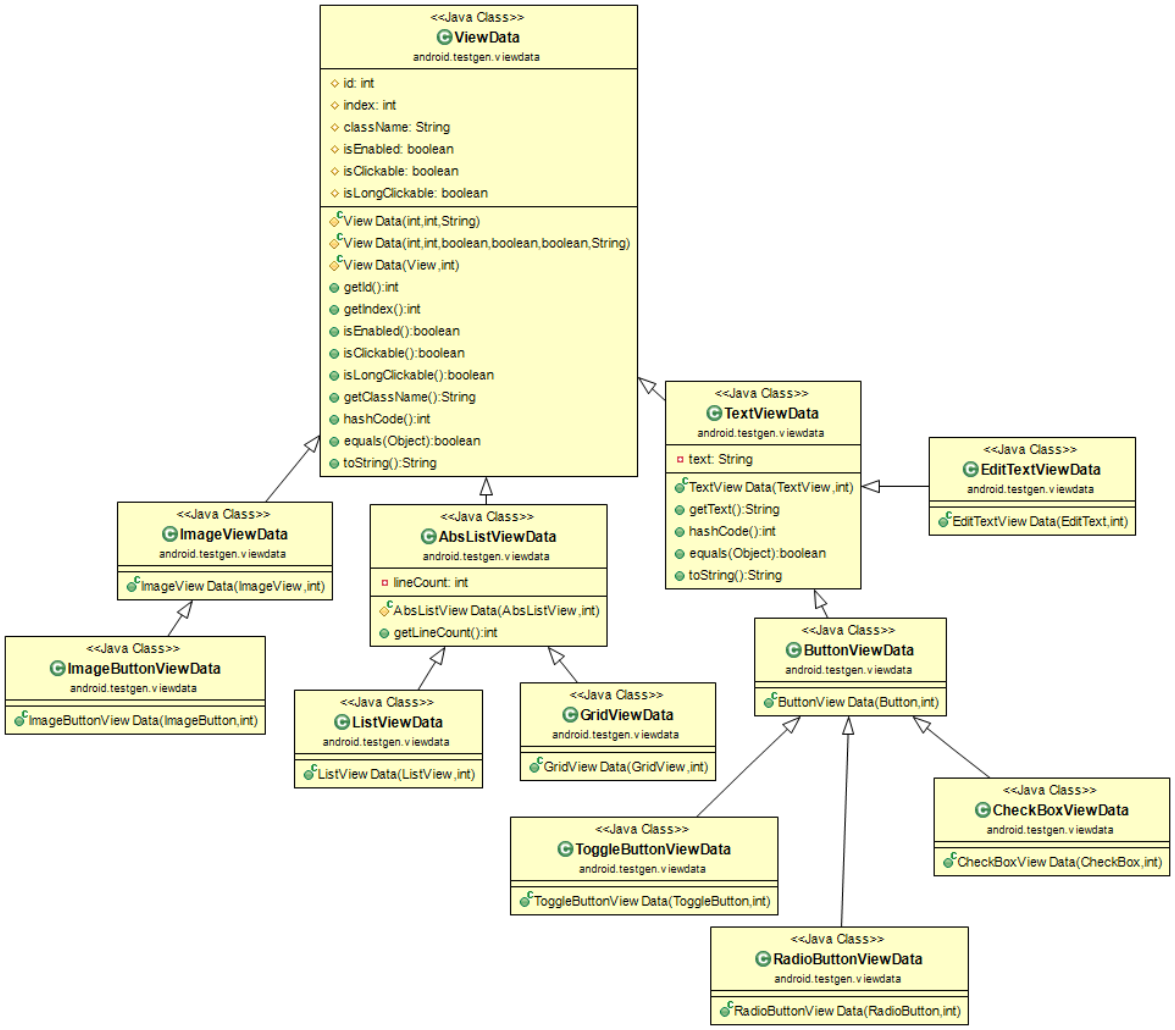


Figura 4.5: Modelo de vistas. Otras vistas se modelan con otras subclases de *ViewData* tales como *BackButtonViewData*, *DatePickerViewData*, *MenuButtonViewData*, o *MenuItemViewData*.

Cada vista de *Android* se encuentra modelada con su correspondiente subclase de ViewData. Debido a que se las vistas de una aplicación se obtienen y manipulan a través de *Robotium*, las vistas soportadas son las que éste *framework* soporta.

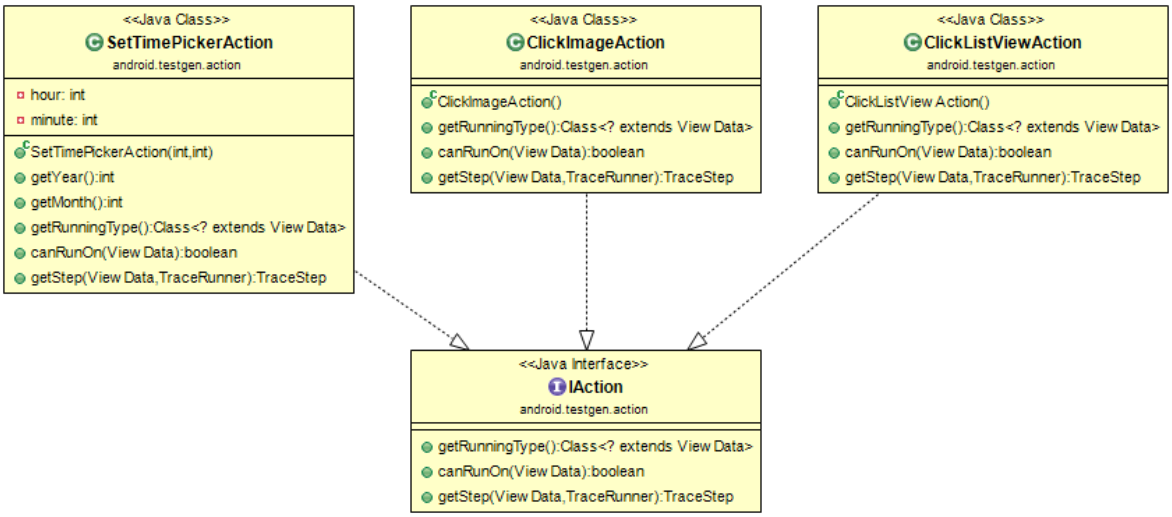


Figura 4.6: Modelo de acciones

Toda acción que se desee soportar debe implementar la interfaz *IAction*. Las acciones son las encargadas de retornar el *Step* (paso de ejecución) correspondiente a través de la colaboración con una instancia de *TraceRunner*.

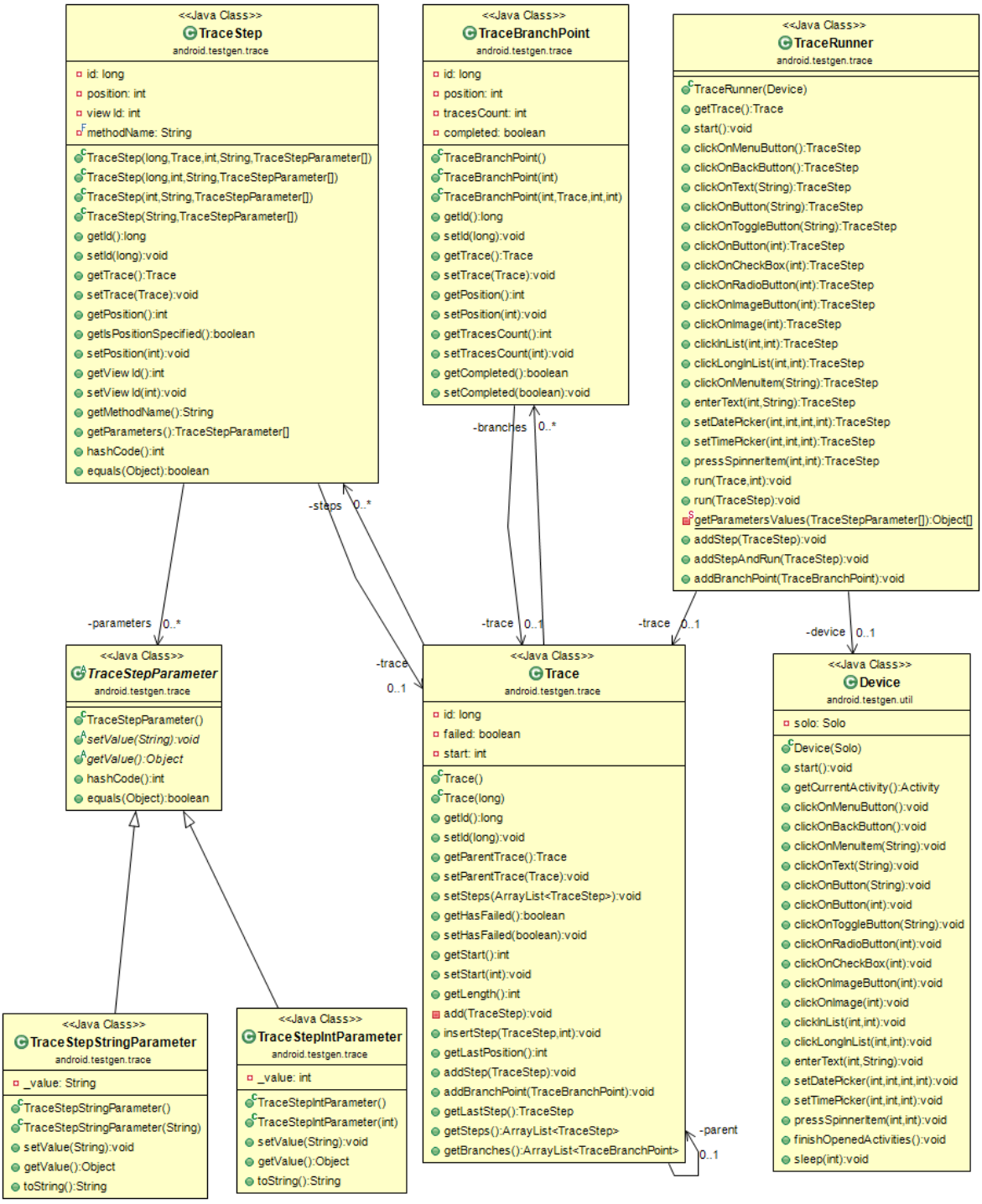


Figura 4.7: Modelo de trazas

Una instancia de *TraceRunner* ejecuta una *Trace* en base a sus *Step*, a través de la colaboración con una instancia de *Device* que se encarga de la interacción con un dispositivo.

Un aspecto importante a destacar es que cada nueva traza generada debe comenzar desde el inicio de la aplicación, por lo que luego de generada una traza cualquiera y como paso previo a la generación de la siguiente, debemos reiniciar el estado de la aplicación. Con este objetivo, decidimos desinstalar y re-instalar la misma luego de la generación de una traza. Ahora bien, debido a que ciertas aplicaciones requieren un seteo inicial por parte del usuario, luego de su instalación (por ejemplo completar un proceso de login), tuvimos la necesidad de implementar un proceso que así lo permita. *ATG* permite, a través del parámetro -settings, especificar la ruta a un archivo XML que contenga la serie de pasos de inicialización que se ejecutarán previos a la generación de cada traza. Por otro lado, en este archivo es posible también definir tanto las acciones a incluir como las que se desea excluir de la generación.

```
<settings>
   <setup-steps>
      <step action="clickOnButton">
         <args>
            <arg type="String" value="Start"/>
         </args>
      </step>
      <step action="clickOnButton">
         <args>
            <arg type="int" value="0"/>
         </args>
      </step>
   </setup-steps>
   <actions run-all="true">
      <skip>
         <action classname="android.testgen.action.ClickBackButtonAction" />
      </skip>
   </actions>
   <skip-views>
      <view id="View1"/>
   </skip-views>
</settings>
```

Figura 4.8: Un ejemplo de un archivo de configuracion

En la figura 4.8 pueden visualizarse los pasos correspondientes a la inicalización de una aplicación dentro del tag $setup - steps$. Cada *step* define una acción que deriva en la ejecución de un método sobre una instancia de *Device* (que es un *wrapper* de la clase *Solo* de *Robotium* y permite la interacción con el dispositivo) y sus parámetros correspondientes. Por otro lado, podemos ver que se definen las acciones que serán incluídas en la generación (en este caso todas), y las que serán excluídas de la misma (en este caso la generación no incluirá a la acción *back*). La herramienta también brinda la posibilidad de no realizar acciones sobre ciertas vistas particulares. En este caso la vista con identificador global $View1$, no será tenida en cuenta en la generación. Este hecho es de utilidad cuando se tiene el conocimiento de que la ejecución de acciones sobre alguna vista particular de nuestra aplicación será perjudicial para el testing de la misma. Por ejemplo, si tenemos una vista que al tocarla llama a otra aplicación lo que terminará en la finalización de nuestro test.

4.6. Requerimientos técnicos e instalación

A continuación se listan los requerimientos técnicos y los pasos para la instalación de *ATG*:

1. Instalar JDK 1.6.

2. Instalar *Android* SDK Tools (`http://developer.android.com/sdk/index.html`)
3. A través del Android SDK Manager, instalar Android SDK platform-tools (tools/platform-tools), Android 2.3.3 (API 10 es una de las más utilizadas actualmente), Extras/Android Support Library y Extras/Google USB Driver (driver usb para dispositivos Android).
4. En Windows abrir el Administrator de dispositivos y actualizar el driver usb del dispositivo utilizando el ubicado en android sdk/extras/google/usb_driver.
5. Setear Java/jdk1.6.0_xx/bin en la variable de entorno PATH (o donde se encuentre javac).
6. Activar el modo de depuracion usb en el dispositivo en donde atg correrá.
7. NOTA 1: No es necesario instalar los drivers USB si se va a correr en el emulador.
8. NOTA 2: Si corriendo en mac durante la ejecución no se llega a encontrar el archivo aapt, agregar el directorio platform-tools de la sdk de Android a la variable de entorno PATH. Para ello se debe ingresar export PATH=$PATH:‘pwd‘, en la línea de comandos.

4.7. Utilización

Para los parámetros que se definen luego, *ATG* genera la siguiente salida en el directorio especificado:

- atg-results: Directorio que incluye el proyecto de instrumentación de *Android* que contiene los tests generados.
- nombre_de_aplicacion.apk: El APK de la aplicación bajo prueba (donde nombre_de_aplicacion es el nombre del APK de la aplicación bajo prueba).
- install-aplicacion.apk: Un script de línea de comandos para instalar el APK de la aplicación bajo prueba en un dispositivo o emulador.
- uninstall-aplicacion.apk: Un script de línea de comandos para desinstalar el APK de la aplicación bajo prueba de un dispositivo o emulador. Éste y el anterior sirven para restablecer la aplicación bajo prueba a su estado inicial entre cada par de tests.
- paquete_aplicacion.dot: Es el archivo .dot con el que se puede llamar a graphviz y generar el mapa de la aplicación (donde paquete_aplicacion es el nombre del paquete de la aplicación bajo prueba).

Los parámetros de *ATG* son:

- apk: La ruta al APK de la aplicación a testear.
- sdk: La ruta a la SDK de *Android*.
- keystore: La ruta al almacén de claves (*keystore*) de desarollo.
- jarsigner: La ruta al programa jarsigner.
- settings: La ruta al archivo XML que define las configuraciones del usuario.
- maxlength (opcional - valor por defecto 5): La longitud máxima (en base a su cantidad de líneas) de los casos de tests a generar.

- maxtraces (opcional): La cantidad máxima de casos de test a generar.
- sleep (opcional): El tiempo que se debe esperar entre cada par de acciones (en milisegundos).
- out (opcional): La ruta donde poner los resultados.
- adbrestart: Especifica si se debe reiniciar el servidor adb entre la generación de una traza y la siguiente. Este parámetro responde a una diferencia en el funcionamiento de *ADB* entre los distintos sistemas operativos. En base a nuestra experiencia este parámetro sólo debería estar presente al correr la aplicación en Windows, mientras que en Linux y Mac no debería tener que utilizarse.

Capítulo 5

Resultados

En este capítulo introduciremos los aspectos involucrados en la validación de la hipótesis del presente trabajo a partir del desarrollo de casos de estudio. Para ello, presentaremos algunos ejemplos que nos servirán para analizar la utilidad de nuestra propuesta, así como también la escalabilidad de la misma. Principalmente nos interesa responder las siguientes preguntas.

- Utilidad
  - ¿Permite descubrir excepciones o errores sin manejar?
  - ¿Alcanza un grado de cobertura adecuado?
  - ¿Ayuda a identificar problemas en la navegabilidad de las aplicaciones?
- Performance
  - ¿Es nuestra implementación lo suficientemente robusta para manejar programas reales complejos?

Este capítulo incluye los resultados de las ejecuciones de nuestra herramienta sobre las siguientes aplicaciones tomadas como ejemplo: `AndroidCalculator`, `Notepad`, `ContactManager`, `TippyTipper`, `Taskos`. Las primeras 3 son aplicaciones de prueba del *framework*, mientras que el resto son aplicaciones reales que pueden ser descargadas desde el *Android Play Store*.

En las próximas secciones nos concentraremos en cada uno de los ejemplos por separado para realizar un breve análisis de los mismos.

5.1. Android Calculator

AndroidCalculator es una aplicación muy simple a la que se le ingresan dos números y luego de presionar un botón la misma retorna el resultado de su multiplicación. Con el fin de conseguir una mayor cobertura se deshabilitó la acción *back*. Por otro lado, la generación se acotó a las primeras 20 trazas de tamaño máximo 5.

Los tests de instrumentación incluídos en la figura 5.1 generados por nuestra herramienta sirven para hacernos dar cuenta de las siguientes anomalías:

- La aplicación falla si se pulsa el *Button* sin haber completado antes ambos *EditText*.

- La aplicación falla si se pulsa el *Button* luego de haber ingresado texto no numérico en los *EditText*.

```
public void testFail1() {
    device.enterText(1,"This is a test");
    device.clickOnMenuButton();
    device.clickOnButton("Multiply");
}

public void testFail2() {
    device.enterText(0,"This is a test");
    device.clickOnButton("Multiply");
}

public void testFail3() {
    device.clickOnButton("Multiply");
}

public void testOk4() {
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.enterText(1,"This is a test");
    device.enterText(1,"This is a test");
}
```

Figura 5.1: Algunos de los tests de *AndroidCalculator* generados por *ATG*

De la información proporcionada tanto por el el mapa de la intefaz gráfica (figura 4.4) como del análisis de cobertura de la aplicación (figura 4.3) sabemos que la misma se compone de una sola actividad. La cobertura de las actividades y de las vistas es del 100 %. Esta aplicación no contiene ítems del menú.

De las estadísticas finales sabemos que...

- El tiempo total de generación de las 20 trazas fue de aproximadamente 9 minutos.
- El tiempo mínimo de generación de una traza fue de 4 segundos.
- El tiempo máximo de generación de una traza fue de 20 segundos.
- El tiempo promedio de generación de una traza fue de 12 segundos.

Debemos tener en cuenta que la aplicación es re-instalada entre cada par de trazas y esto hace que el tiempo de generación total sea mayor que la suma de los tiempos de generación de cada traza.

Los resultados de esta generación se encuentran en resultados/atg out - calculator.

Debido a que el tamaño de esta aplicación es reducido, la misma nos servirá para mostrar que nuestra herramienta genera todas las posibles trazas acotadas en tamaño, siempre y cuando no se especifique el límite a la cantidad de trazas a generar (-maxtraces). En la figura 5.2 se encuentran los tests de dicha aplicación generados por nuestra herramienta donde el tamaño máximo de los mismos es 2. En este caso no se excluye ninguna acción.

Las acciones posibles sobre esta aplicación son *clickOnBackButton*, *clickOnMenuButton*, *enterText* en alguno de los 2 *EditText* y *clickOnButton* sobre el único botón presente en la misma. Si el tamaño de las trazas está acotado por

2, entonces la cantidad total de trazas posibles estará acotado por 25. Como se puede ver en la figura 5.2, la cantidad generada es de 18 tests (<25), y esto se debe a que existen ciertas trazas que no son válidas, ya que por ejemplo (como ya es sabido) la aplicación falla luego de ejecutar $clickOnButton('Multiply')$.

```
public void testFail1() {
    device.clickOnBackButton();
    device.clickOnBackButton();
}
public void testOk2() {
    device.enterText(0,"This is a test");
    device.enterText(1,"This is a test");
}
public void testOk3() {
    device.enterText(1,"This is a test");
    device.enterText(1,"This is a test");
}
public void testFail4() {
    device.clickOnButton("Multiply");
}
public void testFail5() {
    device.clickOnMenuButton();
    device.clickOnButton("Multiply");
}
public void testFail6() {
    device.clickOnBackButton();
    device.clickOnMenuButton();
}
public void testOk7() {
    device.enterText(0,"This is a test");
    device.enterText(0,"This is a test");
}
public void testFail8() {
    device.enterText(0,"This is a test");
    device.clickOnButton("Multiply");
}
public void testOk9() {
    device.enterText(0,"This is a test");
    device.clickOnMenuButton();
}
public void testOk10() {
    device.enterText(0,"This is a test");
    device.clickOnBackButton();
}
public void testOk11() {
    device.enterText(1,"This is a test");
    device.enterText(0,"This is a test");
}
public void testOk11() {
    device.enterText(1,"This is a test");
    device.enterText(0,"This is a test");
}
public void testFail12() {
    device.enterText(1,"This is a test");
    device.clickOnButton("Multiply");
}
public void testOk13() {
    device.enterText(1,"This is a test");
    device.clickOnMenuButton();
}
public void testOk14() {
    device.enterText(1,"This is a test");
    device.clickOnBackButton();
}
public void testOk15() {
    device.clickOnMenuButton();
    device.enterText(0,"This is a test");
}
public void testOk16() {
    device.clickOnMenuButton();
    device.enterText(1,"This is a test");
}
public void testOk17() {
    device.clickOnMenuButton();
    device.clickOnMenuButton();
}
public void testOk18() {
    device.clickOnMenuButton();
    device.clickOnBackButton();
}
```

Figura 5.2: Todos los posibles tests de *AndroidCalculator* con tamaño <= 2 generados por *ATG*

Nuevamente el grado de cobertura es del 100 % ya que alcanzamos a ejercitar todas las vistas de la misma.

Los resultados de esta generación se encuentran en resultados/atg out - calculator all.

5.2. Notepad

Tal como se mencionó anteriormente en la sección 2.1, esta aplicación es un típico editor de notas. Está compuesto por 3 actividades: Una lista de notas, un editor de notas, y una actividad que permite modificar el título de una nota.

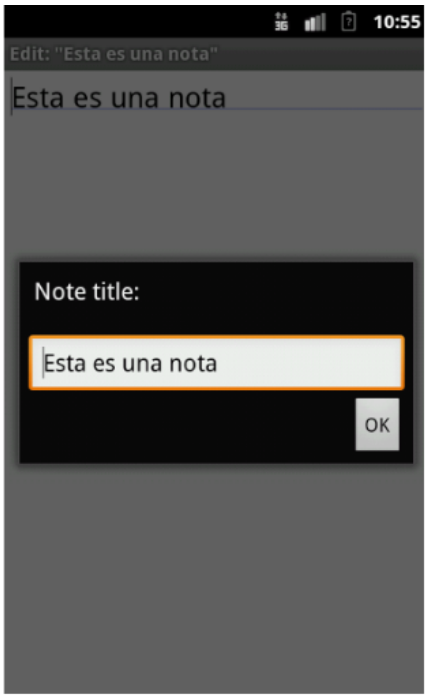
En la generación se deshabilitó la acción *back* y la misma se acotó a las primeras 20 trazas de tamaño máximo 40.

La figura 5.3 detalla las trazas resultantes e incluye el tiempo invertido en su generación (consumiendo un total de 49 minutos aproximadamente). Como se puede observar

```
Running testing apk in the device 39335AEF850D00EC...
Trace #1 succeeded (Elapsed time: 140 secs).
Trace #2 succeeded (Elapsed time: 139 secs).
Trace #3 succeeded (Elapsed time: 127 secs).
Trace #4 succeeded (Elapsed time: 119 secs).
Trace #5 failed (Elapsed time: 110 secs).
Trace #6 succeeded (Elapsed time: 131 secs).
Trace #7 succeeded (Elapsed time: 135 secs).
Trace #8 succeeded (Elapsed time: 131 secs).
Trace #9 succeeded (Elapsed time: 133 secs).
Trace #10 succeeded (Elapsed time: 121 secs).
Trace #11 failed (Elapsed time: 120 secs).
Trace #12 succeeded (Elapsed time: 100 secs).
Trace #13 succeeded (Elapsed time: 128 secs).
Trace #14 succeeded (Elapsed time: 132 secs).
Trace #15 succeeded (Elapsed time: 119 secs).
Trace #16 succeeded (Elapsed time: 116 secs).
Trace #17 succeeded (Elapsed time: 106 secs).
Trace #18 succeeded (Elapsed time: 130 secs).
Trace #19 failed (Elapsed time: 100 secs).
Trace #20 succeeded (Elapsed time: 120 secs).
Minimum trace time: 6 secs.
Maximum trace time: 140 secs.
Trace time average: 73 secs.
Total elapsed time: 2969.0 secs (49.0 mins).
```

Figura 5.3: Resultados de la generación sobre *Notepad*

las trazas 5, 11 y 19 fallaron mientras que el resto ejecutó satisfactoriamente. Si prestamos atención a los tests correspondientes a dichas trazas, llegaremos a la conclusión de que luego de pulsar la tecla de menú estando en la actividad que permite la edición del título de una nota (figura 5.4 a la izquierda), se pierde el foco sobre el cuadro de diálogo que contiene al *EditText* y al *Button* correspondientes, y al pulsar sobre éste último, la generación falla ya que no puede acceder al mismo. Esta situación puede ser corroborada mediante la ejecución de alguno de éstos tests o mediante la ejecución manual sobre un dispositivo. Si bien este hecho no es considerado una excepción o error sin manejar puede ser considerado un error en la usabildad de la aplicación ya que deberemos pulsar 2 veces sobre dicho botón para guardar los cambios.




```
public void testFail5() {
    device.clickOnMenuButton();
    device.clickOnMenuItem("Add note");
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuItem("Discard");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Add note");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Save");
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuItem("Add note");
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.enterText(0,"This is a test");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Save");
    device.clickInList(0,0);
    device.enterText(0,"This is a test");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Edit title");
    device.clickOnMenuButton();
    device.clickOnButton("OK");
}
```

Figura 5.4: Izquierda: Actividad *TitleEditor* de Notepad. Derecha: Test correspondiente a la traza 5

En la figura 5.5 se puede observar el análisis de cobertura de la aplicación. Cabe destacar que el análisis dinámico encontró nuevos ítems del menú y nuevas vistas que no estaban definidas en los archivos de layout correspondientes, hecho por el cual no pudieron ser descubiertos por el análisis estático previo. Si descartamos los *TextViews* (ya que no permiten interacción), la cobertura nuevamente es del 100 % para las vistas e ítems del menú y del 75 % para las actividades, pero se debe tener en cuenta de que no se podrá llegar al 100 % en cuanto a éstas ya que tal como se mencionó anteriormente la actividad NotesLiveFolder carece de interfaz gráfica por ende nunca podrá ser visitada a través de la misma.

Actividades			
# Total	# Visitadas	# No visitadas	Cobertura
4	3	1	75%

Items del menú			
# Nuevos	# Visitados	# No visitados	Cobertura
1 (Edit title)	9	0	100%

Detalle de ítems del menú	
Item	Encontrado en la actividad
Add Note	NotesList
Delete	NoteEditor
Delete	NotesList
Discard	NoteEditor
Edit title	NoteEditor
Edit title	NotesList
Open	NotesList
Revert changes	NoteEditor
Save	NoteEditor

Vistas				
# Nuevas	# Visitadas	# No visitadas	Cobertura	Cobertura (sin contar TextViews)
4	4	5	44%	100%

Detalle de vistas			
Id	Tipo	Encontrada en la actividad	Visitada?
note	EditText	NoteEditor	SI
interno (16908298)	ListView	NotesList	SI
interno (16908310)	TextView	NoteEditor	NO
interno (16908310)	TextView	NotesList	NO
interno (16908310)	TextView	TitleEditor	NO
interno (16908299)	TextView	NotesList	NO
interno (16908308)	TextView	NotesList	NO
title	EditText	TitleEditor	SI
ok	Button	TitleEditor	SI

Figura 5.5: Análisis de cobertura de *Notepad*

Tal como se mencionó anteriormente, nuestra herramienta almacena la información correspondiente a las transiciones entre las diferentes actividades de una aplicación. En este caso particular dicha información se encuentra en la figura 5.6.

En figura 5.7 se puede observar el mapa de la interfaz gráfica de la aplicación. El mismo incluye las diferentes actividades, transiciones entre las mismas, y vistas

Transiciones entre actividades		
Actividad origen	**Actividad destino**	**Accion**
NotesList	NoteEditor	clickOnMenuItem('Add Note')
NoteEditor	NotesList	clickOnMenuItem('Save')
NoteEditor	NotesList	clickOnMenuItem('Discard')
NotesList	NoteEditor	clickInList(view {id: 16908298, tipo: ListView})
NoteEditor	TitleEditor	clickOnMenuItem('Edit title')
NotesList	NoteEditor	clickOnMenuItem('Open')
NoteEditor	NotesList	clickOnMenuItem('Revert changes')
TitleEditor	NoteEditor	clickOnButton(view {id: ok, tipo: Button})
NoteEditor	NotesList	clickOnMenuItem('Delete')
NotesList	TitleEditor	clickOnMenuItem('Edit title')
TitleEditor	NotesList	enterText(view {id: title, tipo: EditText})

Figura 5.6: Transiciones entre las actividades de *Notepad*

e ítems del menú que las componen. Los rectángulos contenedores representan a las actividades, los rectángulos que aparecen dentro de éstas representan a los ítems del menú y los óvalos representan a las diferentes vistas. En color verde pueden observarse las actividades, vistas e ítems del menú visitados, en rojo los no visitados y en gris, los controles nuevos.

Los resultados de esta generación se encuentran en resultados/atg out - notepad.

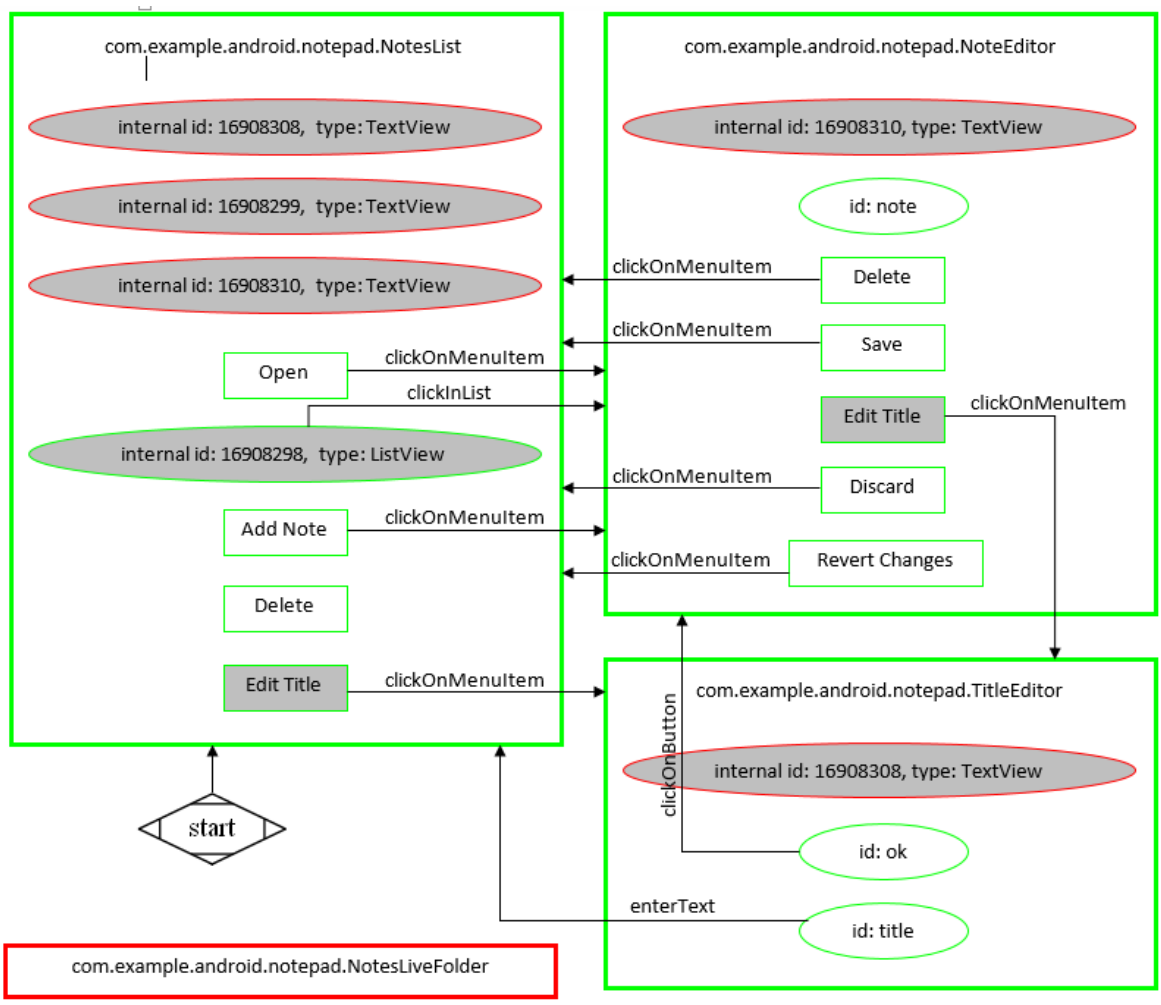


Figura 5.7: Mapa de la interfaz gráfica de *Notepad*

5.3. Notepad con errores

En este caso, se modificó el código fuente de la aplicación en cuestión y se le introdujeron los errores descriptos a continuación con el objetivo de conocer si nuestra herramienta es capaz de encontrarlos.

- En la actividad *NotesList* (lista de notas), luego de realizar un toque largo sobre alguna nota de lista, se abre un menú contextual con las opciones *Open*, *Delete* y *Edit Title*. Se introdujo un error no manejado en el manejador del evento que dispara la opción *Open*.
- En la actividad *NoteEditor* (editor de notas), se introdujo un error no manejado en el manejador del evento disparado por el ítem de menú *Discard*.

Luego de la introducción de dichos errores se ejecutó nuestra herramienta, nuevamente deshabilitando la acción *back* y acotando la generación de trazas a las primeras 20 siendo su tamaño máximo 40 pasos.

```
public void testFail1() {
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuItem("Add note");
    device.enterText(0,"This is a test");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Discard");
}
```

```
public void testFail19() {
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuItem("Add note");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Save");
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.clickOnMenuItem("Add note");
    device.enterText(0,"This is a test");
    device.enterText(0,"This is a test");
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.enterText(0,"This is a test");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Save");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Add note");
    device.clickOnMenuButton();
    device.clickOnMenuButton();
    device.enterText(0,"This is a test");
    device.enterText(0,"This is a test");
    device.clickOnMenuButton();
    device.clickOnMenuItem("Save");
    device.clickLongInList(0,0);
    device.clickOnMenuItem("Open");
}
```

Figura 5.8: Trazas 1 y 19 evidencian los errores introducidos

Como puede visualizarse en la figura 5.8 nuestra herramienta fue capaz de encontrar dichos errores.

5.4. ContactManager

Esta aplicación permite visualizar los contactos existentes en un dispositivo y agregar nuevos, por lo que se compone de 2 actividades, una para cada uno de éstos propósitos. Para ésta prueba, en la generación, se deshabilitó la acción *back* y la misma se acotó a las primeras 5 trazas de tamaño máximo 30.

```
Saving execution settings... <DONE>
+++ Found 2 activities... <DONE>
+++ Found 14 views... <DONE>
+++ Found 0 menu items... <DONE>
Pushing database into the device 39335AEF850D00EC... <DONE>

Please unlock 39335AEF850D00EC to start testing...
Press enter when ready...

Running testing apk in the device 39335AEF850D00EC...
Trace #1 succeeded (Elapsed time: 223 secs).
Trace #2 succeeded (Elapsed time: 171 secs).
Trace #3 succeeded (Elapsed time: 139 secs).
Trace #4 succeeded (Elapsed time: 119 secs).
Trace #5 succeeded (Elapsed time: 195 secs).
Minimum trace time: 119 secs.
Maximum trace time: 223 secs.
Trace time average: 171 secs.
Total elapsed time: 916.0 secs (15.0 mins).

Saving generated tests... <DONE>
Processing results...

Visited activities:
        com.example.android.contactmanager.ContactManager
        com.example.android.contactmanager.ContactAdder
+++ Total visited activities: 2.

Not visited activities:
+++ Total not visited activities: 0.
+++ Activities coverage: 100%.

New menu items found:
+++ Total new menu items: 0.

Visited menu items:
+++ Total visited menu items: 0.

Not visited menu items:
+++ Total not visited menu items: 0.
+++ Menu items coverage: 0%.
```

Figura 5.9: Últimos pasos del setup, resultados del análisis estático, trazas generadas y cobertura de actividades y menú de *ContactManager*

Como se ve en la figura 5.9, nuestra herramienta encontró las 2 actividades de la aplicación y no se encontró uso alguno de su menú. El tiempo total de generación de las trazas fue de 15 minutos aproximadamente.

```
++ Total not visited views: 8.
++ Total not visited views (not counting TextViews): 1.
+++ Views coverage: 55,56%.
+++ Views coverage (not counting TextViews): 90,91%.

Activity transitions:
        com.example.android.contactmanager.ContactManager->com.example.android.c
ontactmanager.ContactAdder: clickOnButton on view (id: addContactButton)
        com.example.android.contactmanager.ContactAdder->com.example.android.con
tactmanager.ContactManager: clickOnButton on view (id: contactSaveButton)
+++ Total transitions: 2.
```

Figura 5.10: Cálculo de cobertura de vistas y transiciones entre actividades de *ContactManager*

Nuestra herramienta ha alcanzado un alto grado de cobertura sobre las vistas (91 %). El mismo puede visualizarse en la figura 5.10 junto con las transiciones encontradas entre las actividades de la aplicación. De dicha figura se puede conocer que la única vista no visitada es una imagen, pero en este caso ninguna operación puede realizarse sobre la misma, por lo que podemos concluir que la cobertura fue del 100 %. Como se detalla, para acceder a la actividad *ContactAdder* debemos pulsar sobre el botón con id «addContactButton», y para regresar desde ésta última a *ContactManager* debemos pulsar sobre el botón con id «contactSaveButton».

Los resultados de esta generación se encuentran en resultados/atg out - contact manager.

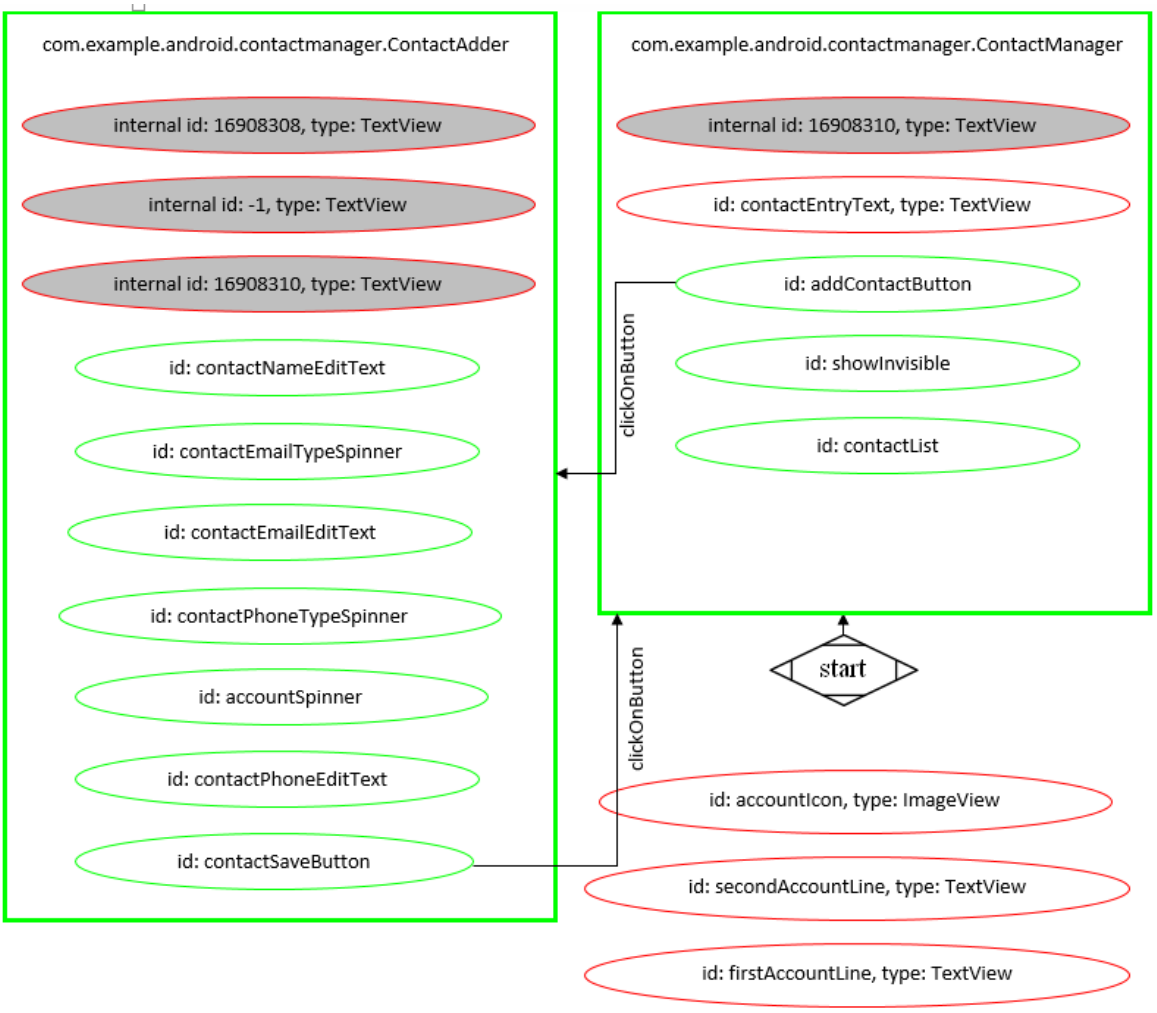


Figura 5.11: Mapa de la interfaz gráfica de *ContactManager*

5.5. TippyTipper

Esta aplicación permite calcular la propina en base a un monto ingresado, el porcentaje de propina que se desea dejar y la cantidad de personas por las cuales se quiere dividir el importe, entre otras configuraciones. Este ejemplo nos sirve para mostrar el desempeño de nuestra herramienta sobre una aplicación comercial de tamaño mediano, ya que la misma puede ser descargada desde el *Android Play Store*.

Para ésta prueba, en la generación, se deshabilitó la acción *back* y la misma se acotó a las primeras 10 trazas de tamaño máximo 40.

```
+++ Found 55 views... <DONE>
+++ Found 2 menu items... <DONE>
Pushing database into the device 39335AEF850D00EC... <DONE>

Please unlock 39335AEF850D00EC to start testing...
Press enter when ready...

Running testing apk in the device 39335AEF850D00EC...
Trace #1 succeeded (Elapsed time: 151 secs).
Trace #2 succeeded (Elapsed time: 335 secs).
Trace #3 succeeded (Elapsed time: 232 secs).
Trace #4 succeeded (Elapsed time: 349 secs).
Trace #5 succeeded (Elapsed time: 201 secs).
Trace #6 succeeded (Elapsed time: 199 secs).
Trace #7 succeeded (Elapsed time: 121 secs).
Trace #8 succeeded (Elapsed time: 260 secs).
Trace #9 failed (Elapsed time: 101 secs).
Trace #10 failed (Elapsed time: 33 secs).
Minimum trace time: 33 secs.
Maximum trace time: 349 secs.
Trace time average: 191 secs.
Total elapsed time: 2182.0 secs (36.0 mins).

Saving generated tests... <DONE>
Processing results...

Visited activities:
        net.mandaria.tippytipper.activities.TippyTipper
        net.mandaria.tippytipper.activities.Total
        net.mandaria.tippytipper.activities.SplitBill
        net.mandaria.tippytipper.activities.About
        net.mandaria.tippytipper.activities.Settings
+++ Total visited activities: 5.

Not visited activities:
+++ Total not visited activities: 0.
+++ Activities coverage: 100%.
```

Figura 5.12: Últimos pasos del setup, resultados del análisis estático, trazas generadas y cobertura de actividades de *TippyTipper*

Como se ve en la figura 5.12, *TippyTipper* se compone de 5 actividades ya que su cobertura es del 100 %, es decir, se visitaron todas las actividades de la aplicación. El tiempo total de generación de las trazas fue de 36 minutos aproximadamente, y de las 10 trazas generadas sólo las últimas dos fallaron. Como puede verse en los tests correspondientes a éstas últimas dos trazas, la falla se encuentra al ejecutar el *step device.clickOnButton*('+'), y está directamente relacionada con *Robotium* ya que cuando dicho *framework* busca las diferentes vistas de la aplicación en base al texto especificado (en este caso «+»), utiliza expresiones regulares y «+» es un caractér reservado, por lo que éstas trazas no representan errores en la aplicación bajo prueba y pueden ser considerados como falsos positivos.

```
ria.tippytipper.activities.Settings
++ Total not visited views: 32.
++ Total not visited views (not counting TextViews): 8.
+++ Views coverage: 43,86%.
+++ Views coverage (not counting TextViews): 75,76%.

Activity transitions:
        net.mandaria.tippytipper.activities.TippyTipper->net.mandaria.tippytippe
r.activities.About: clickOnMenuItem('About')
        net.mandaria.tippytipper.activities.TippyTipper->net.mandaria.tippytippe
r.activities.Settings: clickOnMenuItem('Settings')
        net.mandaria.tippytipper.activities.TippyTipper->net.mandaria.tippytippe
r.activities.Total: clickOnButton on view (id: btn_ok)
        net.mandaria.tippytipper.activities.Total->net.mandaria.tippytipper.acti
vities.Settings: clickOnMenuItem('Settings')
        net.mandaria.tippytipper.activities.Total->net.mandaria.tippytipper.acti
vities.SplitBill: clickOnButton on view (id: btn_SplitBill)
+++ Total transitions: 5.
```

Figura 5.13: Cálculo de cobertura de vistas y transiciones entre actividades de *TippyTipper*

La cobertura de vistas, que puede ser visualizada en la figura 5.13, es del 76 % aproximadamente, y no está nada mal visto y considerando que la aplicación bajo prueba es una aplicación comercial y que la generación se limitó a sólo 10 trazas. Existen ciertas vistas que no pudieron ser visitadas tanto por ser de tipo ViewStub (un tipo de vista especial que no tiene layout y suele ser invisible que sirve como origen de datos para otras vistas) o bien por no estar soportadas, como por ejemplo las de tipo *NumberPicker* (que es un tipo de vista especial definido por la aplicación). Por otro lado, las transiciones encontradas permiten acceder a todas las actividades de la aplicación.

Los resultados de esta generación se encuentran en resultados/atg out - tippy tipper.

5.6. Taskos

Esta aplicación permite administrar una lista de tareas incluyendo sus fechas y horas, prioridades y categorías, entre otras opciones. También permite compartir y sincronizar las tareas mediante diversos medios que pueden ser e-mails, mensajes de texto y otros. Este ejemplo nos sirve para mostrar el desempeño de nuestra herramienta sobre una aplicación comercial de gran tamaño. Esta aplicación también puede ser descargada desde el *Android Play Store*.

Para poder testear esta aplicación fue necesario generar una configuración de pasos iniciales, debido a que una vez que se ingresa a la aplicación por primera vez, se muestra su tutorial de uso, y tal como se mencionó anteriormente entre cada par de trazas la aplicación bajo prueba es re-instalada con el fin de volver a su estado inicial. Esta configuración se realiza mediante un archivo de *settings* especial cuya ruta se le otorga a nuestra herramienta a través del parámetro -settings. En este caso dicha configuración se encuentra en resultados/atg out - taskos/taskos - settings. Luego de ejecutar los pasos de inicialización, la herramienta comienza a generar las trazas correspondientes.

Para ésta prueba, en la generación, se deshabilitó la acción *back* y la misma se acotó a las primeras 100 trazas de tamaño máximo 50. Por otro lado, también se deshabilitó la vista con id «btnVoiceRecognition», ya que al ser tocada desencadena un intent que abre los componentes de reconocimiento de voz de *Google* y, debido a las limitaciones anteriormente mencionadas en la sección 4.4, este hecho provoca la finalización de los tests de instrumentación activos.

Debido su gran tamaño, la imágenes concernientes a los resultados obtenidos por nuestra herramienta testeando *Taskos* no se incluyen en su totalidad en este informe y pueden visualizarse en resultados/atg out - taskos/imagenes.

Luego de aproximadamente 5 horas de ejecución, *ATG* nos informa que *Taskos* posee 11 actividades y 8 de ellas pudieron ser visitadas, dando una cobertura del 72,73 %. Las 3 actividades no visitadas corresponden a diferentes *widgets* que provee la aplicación (2) y a una actividad que permite compartir una tarea con cualquier contacto incluído en el dispositivo, que por alguna razón nunca detecta conectividad a Internet. Por otro lado, se ejercitaron 13 de los 14 ítems del menú encontrados, otorgando una cobertura del 92,86 % en este rubro. El punto más bajo está relacionado a la cobertura de vistas, que fue del 41,98 %, y se debe a la imposibilidad de visitar las 3 actividades recién mencionadas. Se encontraron un total de 15 transiciones entre las diferentes actividades de la aplicación. No se encontraron trazas fallidas.

En la figura 5.14 puede observarse un gráfico del tiempo transcurrido en base a diferentes tamaños de traza. Los pequeños saltos (como el que se encuentra en 10 steps aprox.) indican que la vista inicialmente seleccionada por el algoritmo no pudo ser ejercitada, por lo que otra tuvo que ser seleccionada en su lugar (y eventualmente otra acción relacionada).

El mapa de la interfaz gráfica de la aplicación es demasiado grande y no entra en este informe, pero puede ser encontrado en resultados/atg out - taskos/com.taskos.png.

Los resultados de esta generación se encuentran en resultados/atg out - taskos.

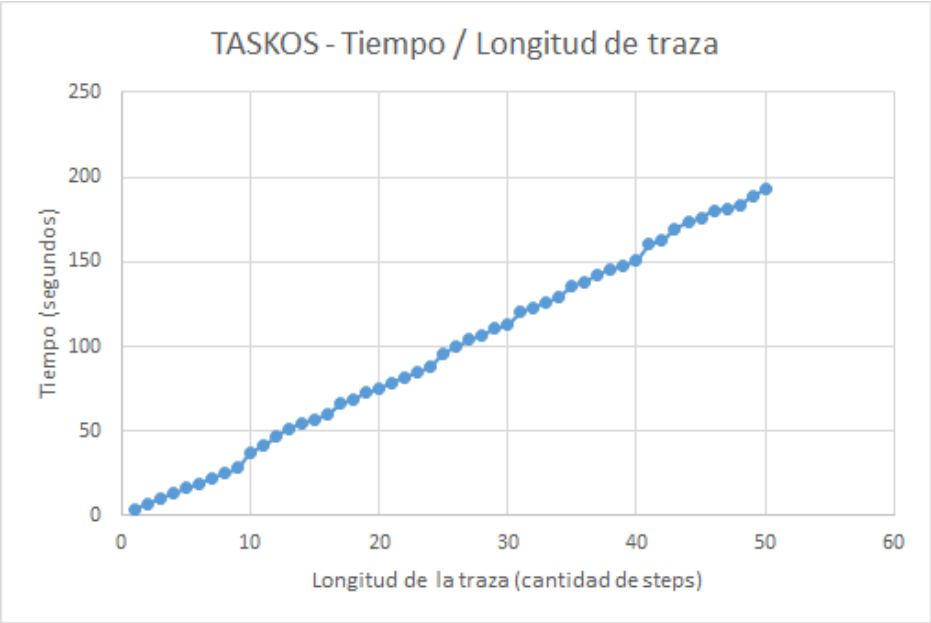


Figura 5.14: Gráfico de tiempo sobre longitud de traza

5.7. Conclusiones

En base a los resultados obtenidos en las diferentes corridas de nuestra herramienta sobre distintas aplicaciones, podemos concluir que *ATG* permite descubrir excepciones o errores sin manejar en las aplicaciones bajo prueba. Ésto se evidencia particularmente en los resultados obtenidos al correr nuestra herramienta sobre *AndroidCalculator* y sobre la versión modificada de *Notepad* a la cual se le introdujeron errores adrede.

Tal como se puede observar en la sección 5.2, *ATG* también permite identificar ciertos problemas en la navegabilidad / usabilidad de las aplicaciones a través de la inspección de las trazas generadas.

En cuanto al nivel de cobertura alcanzado, podemos concluir que depende directamente de la navegabilidad de las aplicaciones como de los tipos de vistas soportados por el *framework* utilizado para la interacción con los dispositivos/emuladores (en nuestro caso, *Robotium*). En este aspecto también se deben tener en cuenta los límites en la cantidad de trazas a generar y en su respectivo tamaño (ambos configurables en nuestra herramienta), ya que limitan el análisis y por lo tanto afectan a la cobertura.

En base a los resultados obtenidos con *Taskos* y *TippyTipper*, podemos concluir que nuestra herramienta puede soportar aplicaciones comerciales de tamaño real aunque su nivel de cobertura variará en base los aspectos recién mencionados.

Capítulo 6

Trabajo relacionado

Si bien el presente trabajo no se basa en ningún otro trabajo existente, mantiene ciertos puntos de contacto con éstos. En primer lugar, herramientas como Android GUITAR [11] u ORBIT [12], desempeñan un rol similar al de nuestra herramienta, en la generación de tests y en la búsqueda de excepciones o errores sin manejar en aplicaciones Android, interactuando con éstas en base a su interfaz gráfica. La diferencia entre éstas radica en que la primera realiza testing de tipo *black-box*, mientras que la segunda lo hace *gray-box*. En este aspecto, nuestro trabajo se asemeja más a la segunda herramienta, ya que de hecho ambas son de tipo *gray-box* y utilizan Robotium [5] como *crawler*, pero a diferencia de ésta, nuestra herramienta no analiza el código fuente de las aplicaciones bajo prueba como parte de su análisis estático sino que, como se mencionó anteriormente, analiza los archivos concernientes a los aspectos gráficos y visuales de las mismas, con el fin de conocer el grado de cobertura alcanzado en los tests generados, y no como base para su generación.

Otra diferencia que cabe destacar radica en que nuestra herramienta genera todas las posibles trazas acotadas en tamaño (definido en base a su cantidad de *Steps*) de la aplicación bajo prueba partiendo desde su estado inicial, en este caso, el estado al instalarla. En ORBIT [12], se trata a una misma aplicación (*com.example.android.Notepad*, que también fue analizada en el presente trabajo), como si fuera dos aplicaciones diferentes en base a su estado inicial, por ejemplo si posee una nota existente o no, por lo que dependiendo de su estado inicial obtendremos distintos resultados.

Otro aspecto similar entre Android GUITAR [11] (a través de su *plugin* Visualization Tool) y nuestra herramienta radica en que ambas permiten la generación de un mapa de la interfaz de usuario de las aplicaciones bajo prueba.

Nuestra herramienta pudo testear la mayoría de las aplicaciones analizadas en ORBIT [12], con el agregado de haber podido testear también aplicaciones que se pueden descargar directamente de la tienda *Android Play Store*, y a diferencia de dicho trabajo, las aplicaciones bajo prueba no tuvieron que sufrir modificaciones y/o simplificaciones.

Tal como se mencionó anteriormente, *Android* provee el Application Exerciser Monkey, un generador de eventos de usuario pseudo-aleatorios que simulan ser clicks, toques y otros gestos, pero a diferencia de nuestra herramienta, se aboca a realizar pruebas de *stress* y no a testear la *AUT* en base a su interfaz gráfica con el objetivo de conocer su composición y cubrirla con las trazas generadas.

Otra herramienta que promete realizar testeo de aplicaciones *Android* en base a su interfaz gráfica es DroidMate [13], pero la información relacionada a éste que puede

encontrase es limitada por lo que hasta el momento no puede realizarse comparación alguna con nuestra herramienta.

Existen otros tipos de trabajos relacionados a detectar violaciones a propiedades comunes como *deadlocks* y excepciones en *runtime*, pero que se diferencian del nuestro ya que requieren de la interacción con un usuario que conozca del funcionamiento y/o navegabilidad de la aplicación bajo prueba. El trabajo en [14] plantea la verificación de aplicaciones *Android* en base a Java PathFinder [17], uno de los *model checkers* más utilizados en la plataforma Java. Su herramienta, JPF-ANDROID, provee un modelo simplificado del *framework Android* en donde las aplicaciones pueden correr, y luego permite que el usuario final ingrese eventos de manera de manejar el flujo de la aplicación bajo prueba.

El trabajo incluído en [15] plantea una idea similar a la ya mencionada en [14], pero a diferencia de éste último, utiliza Java PathFinder junto con técnicas de ejecución simbólica con el fin de generar valores de entrada válidos para las diferentes aplicaciones bajo prueba. A diferencia de nuestro trabajo, ambos trabajos basan su funcionamiento en el análisis del código fuente de las *AUT*. Con el fin de interactuar con la *AUT*, en el caso de [15], se generan drivers que simulan los eventos ante los cuales la *AUT* reacciona, por ende no interactúa directamente con la interfaz gráfica tal como lo hace nuestra herramienta.

El trabajo en [16] utiliza un enfoque diferente a los recién mencionados, ya que se caracteriza por grabar y reproducir (*Record and Replay*) las distintas acciones y eventos de bajo nivel que suceden en los dispositivos, teniendo en cuenta también a los diferentes sensores que éstos poseen, a través del monitoreo de un *log* en donde impacta constantemente cualquier cambio de estado del mismo (obviamente incluyendo dichas acciones y eventos). Pero nuevamente requiere de la interacción con un usuario real que desecadene los diferentes eventos de las aplicaciones bajo prueba.

No es posible realizar una comparación empírica entre nuestro trabajo y el resto de los trabajos mencionados en este apartado en base al nivel de cobertura de código alcanzado, ya que dicho valor solo aplica a los trabajos que utilizan estrategias de testing white-box (y de hecho sólo un trabajo lo incluye), y el criterio de cobertura introducido en el presente trabajo no mantiene relación directa con el código fuente de la aplicación bajo prueba, sino con sus componentes gráficos. Por lo tanto, la única comparación empírica posible radica en el numero de excepciones, o errores sin manejar, encontrados por medio del testing de las mismas aplicaciones a traves de la ejecución de las diferentes herramientas introducidas en los presentes trabajos. Debido a que la instalación, configuración y ejecución de todas las herramientas en cuestión es un trabajo en sí mismo, lo proponemos como trabajo futuro.

Capítulo 7

Conclusiones

En este trabajo nos propusimos desarrollar una estrategia *gray-box* de *testing* automático de aplicaciones para dispositivos móviles pertenecientes a la plataforma *Android* mediante un fuerte uso de sus interfaces gráficas (*GUIs*). Para ello, explicamos la arquitectura básica de dichas aplicaciones, incluyendo sus componentes visuales, junto con los diferentes componentes que conforman tanto al *framework* de testing nativo de la plataforma, como a otras herramientas ajenas a ésta que son útiles a la hora de administrar los recursos de la misma.

Como producto de esto desarrollamos *Android Test Generator*, una herramienta capaz de tomar un *application package file* de una aplicación *Android* junto con otros parámetros, producir una serie de tests de instrumentación basados en la ejecución de trazas reales de la aplicación bajo prueba en un dispositivo o emulador a través de la interacción mediante su interfaz gráfica, y realizar un análisis de cobertura de la misma en base a las tests generados y un análisis estático previo. Dicha herramienta es también capaz de generar un mapa de la aplicación bajo prueba en base a su interfaz gráfica.

Cabe destacar que nuestra herramienta, si bien realiza testing de tipo *gray-box*, no analiza el código fuente de las aplicaciones bajo prueba con el objetivo de generar las trazas que determinarán los tests, sino que utiliza ciertos recursos asociados a aspectos gráficos de las mismas con el fin de proveer un análisis de cobertura más preciso.

En base a los resultados incluídos en el presente trabajo, y teniendo en cuenta las limitaciones encontradas en las distintas herramientas presentes en la plataforma, podemos concluir que nuestra herramienta sirve como una buena base para futuros desarrollos en el ámbito del *testing* de aplicaciones móviles. El nivel de cobertura alcanzado tanto en aplicaciones de prueba como en aplicaciones comerciales es alentador, y será aún mayor en la medida en que se puedan romper las limitaciones hasta hoy existentes.

7.1. Trabajo a futuro

En primer lugar, y como es sabido, la plataforma *Android* se encuentra en un estado de cambio constante dando lugar a nuevos componentes, vistas y recursos gráficos, por lo que este trabajo podrá ser actualizado para proveer el soporte correspondiente. Por otro lado, como ya se mencionó *Android Test Generator* depende de *Robotium*, por

lo que a medida de que éste evolucione brindando mejores servicios y acceso a nuevos componentes, nuestra herramienta se verá directamente beneficiada.

Los aspectos más importantes en los que se puede trabajar a futuro son los relacionados con la mitigación de las limitaciones descriptas en 4.4, y de todas ellas quizás la más importante sea la relacionada al cierre de la aplicación bajo prueba sin que el framework de instrumentación lo note, debido a que ésta es la causante de la mayor cantidad de falsos positivos en la detección de errores sin manejar de nuestra herramienta. Para ello, se deberían modificar componentes del *framework* de instrumentación nativo de la plataforma *Android*.

Una posible mejora de nuestra herramienta en la que se puede trabajar a futuro puede basarse en el almacenado de los tests generados como base para siguientes pruebas, de manera de que si por alguna razón la generación se corta durante el procesamiento, éste pueda ser reanudado.

Por otro lado, pueden considerarse otras mejoras en base a técnicas de grabar-y-reproducir (*record and replay*) con el objetivo de reemplazar el actual método de seteo inicial de pasos de ejecución de las aplicaciones bajo prueba.

Con el objetivo de conocer mayor información acerca de la aplicación bajo prueba, se podría realizar un análisis estático de su código de manera de, por ejemplo, conocer qué valores son aceptados en las distintas cajas de texto e inputs de la misma, hecho que permitiría testear a fondo las diferentes validaciones de entrada sin acudir a la aleatoriedad. Por otro lado, dicho análisis ayudaría a establecer las dimensiones de la *AUT* y aportaría información importante en relación al cálculo de cobertura.

Bibliografía

[1] Google Inc. and Open Handset Alliance, *Android SDK API Guides*. 2008. http://developer.android.com/guide/components/index.html

[2] Google Inc. and Open Handset Alliance, *Android Testing Fundamentals*. 2008. http://developer.android.com/tools/testing/index.html

[3] *JUnit*. http://junit.org/

[4] Google Inc. and Open Handset Alliance, *Application Exerciser Monkey*. http://developer.android.com/tools/help/monkey.html

[5] Renas Reda and Hugo Josefson, *Robotium*. 2011. https://code.google.com/p/robotium/

[6] Google Inc. and Open Handset Alliance, *ADB - Android Debug Bridge*. http://developer.android.com/tools/help/adb.html

[7] SQLite Consortium, *SQLite*. http://www.sqlite.org/

[8] Ryszard Wisniewski and Connor Tumbleson, *android-apktool*. https://code.google.com/p/android-apktool/

[9] Oracle, *jarsigner - JAR Signing and Verification Tool*. http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html

[10] Google Inc. and Open Handset Alliance, *zipalign*. http://developer.android.com/tools/help/zipalign.html

[11] Atif Memon, *Android GUITAR (Graphical User Interface Testing frAmeweoRk)*. 2011. http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR

[12] Wei Yang and Mukul R. Prasad and Tao Xie, *A gray-box approach for automated gui-model generation of mobile applications*. 2013. http://dx.doi.org/10.1007/978-3-642-37057-1_19

[13] Konrad Jamrozik and Florian Gross and Andreas Zeller, *DroidMate*. 2013. https://droidmate.org

[14] Heila van der Merwe and Brink van der Merwe and Willem Visser, *Verifying android applications using Java PathFinder*. 2012. http://dx.doi.org/10.1145/2382756.2382797

[15] Nariman Mirzaei and Sam Malek and Corina S. Pasareanu and Naeem Esfahani and Riyadh Mahmood, *Testing Android Apps Through Symbolic Execution*. 2012. http://dx.doi.org/10.1145/2382756.2382798

[16] Lorenzo Gomez and Iulian Neamtiu and Tanzirul Azim and Todd Millstein, *RERAN: Timing-and Touch-Sensitive Record and Replay for Android*. 2013. http://www.cs.ucr.edu/~neamtiu/pubs/icse13gomez.pdf

[17] Nasa Ames Research Center, *Java Pathfinder*. http://babelfish.arc.nasa.gov/trac/jpf