

# Comparison Among Automatic Android Application Exploration Tools

---

Michael Stiven Osorio Riaño

*June, 2020*

Version: 1.0



Bogotá, Colombia



Department of Systems and Computing Engineering  
The Software Design Lab

Undergraduate Thesis

# **Comparison Among Automatic Android Application Exploration Tools**

Michael Stiven Osorio Riaño

*Advisor*      **Ph.D. Mario Linares Vásquez**  
Department of Systems and Computing Engineering  
University of Los Andes, Bogotá, Colombia

June, 2020

**Michael Stiven Osorio Riaño**

*Comparison Among Automatic Android Application Exploration Tools*

Undergraduate Thesis, June, 2020

Advisor: Ph.D. Mario Linares Vásquez

**University of Los Andes**

*The Software Design Lab*

Department of Systems and Computing Engineering

Cra 1 # 18A - 12

111711 and Bogotá, Colombia

# Abstract

La complejidad y el rápido crecimiento del mercado de aplicaciones móviles, hace necesario que los desarrolladores deban automatizar tareas de pruebas y control de calidad sobre los productos de software. El presente documento presenta un enfoque de generación automática de pruebas, basado en el análisis de multi-modelos: estructuras con información sobre una aplicación móvil y su contexto que permiten definir casos de uso de las aplicaciones sin intervención humana. De forma automática se crean pruebas basadas en *Espresso*, una herramienta enfocada en los desarrolladores, que les permite integrar sus pruebas: los ladrillos que permiten construir entornos con integración continua.



# Índice general

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Objectives . . . . .	2
1.3	Expected Results . . . . .	2
1.4	Reached Results . . . . .	2
<b>2</b>	<b>Trabajo relacionado y herramientas existentes</b>	<b>3</b>
2.1	Generación automática de pruebas . . . . .	3
2.1.1	Barista . . . . .	3
2.1.2	MobiGUITAR . . . . .	3
2.1.3	Firebase Test Lab . . . . .	3
2.1.4	Sapienz . . . . .	4
<b>3</b>	<b>RIP Tests Generator</b>	<b>5</b>
3.1	Solución propuesta . . . . .	5
3.2	Arquitectura de la solución . . . . .	5
3.3	Requisitos técnicos para la generación automática de pruebas . . . . .	7
3.4	Generación automática de una prueba . . . . .	7
<b>4</b>	<b>Conclusiones</b>	<b>11</b>
4.1	Objetivos cumplidos . . . . .	11
4.2	Limitaciones del desarrollo . . . . .	11
4.3	Trabajo futuro . . . . .	12





# Introduction

## 1.1 Motivation

Las aplicaciones móviles se han convertido en una gran parte de nuestras vidas. De acuerdo a Statista [**MobileStatista**], el número de aplicaciones disponibles en Google Play Store para el primer cuarto de 2018 fue de 3.8 millones. En 2017 fueron descargadas 178.1 billones de aplicaciones en el mundo. A medida que crece este mercado, también lo hace la necesidad de mejorar cada etapa del proceso de desarrollo para entregar un producto de calidad de manera eficiente. Una de estas es la etapa de pruebas. Mediante esta los desarrolladores pueden asegurar la calidad del software y por consiguiente la satisfacción del cliente.

En el desarrollo de aplicaciones móviles, es especialmente crítico asegurar que el producto está libre de errores. Hay una gran variedad de alternativas presentadas a los usuarios, y esta facilidad de adquisición hace para ellos más fácil tomar la decisión de cambiar entre ellas. Es por esto que un error en la aplicación puede traducirse en una gran pérdida de clientes.

Las pruebas en aplicaciones móviles se enfrentan a una variedad de retos, uno de estos es la des-fragmentación de plataformas móviles. Esto se refiere al hecho de que hay un gran número de combinaciones de dispositivos con sus sistemas operativos. Por lo tanto, la tarea de realizar pruebas en una aplicación móvil se hace aún más complicada. No solo tiene el tester que asegurar el correcto funcionamiento de la aplicación en un SO (Android, iOS), sino que también se deben tener en cuenta las diferentes modificaciones que son realizadas por cada OEM (Manufacturador original del equipo). Dado que este problema tiene mayor impacto para el caso de Android, el proyecto se basará en dicho sistema operativo. Por otro lado, las aplicaciones móviles, en comparación con las de escritorio, están sujetas a muchas más variables que pueden influenciar su correcto funcionamiento. Los recursos disponibles en un dispositivo móvil son mucho más limitados que los que se encuentran en un computador. Además, se deben tener en cuenta sensores, batería, otras aplicaciones corriendo en el dispositivo, etc.

Por consiguiente, al tener una gran variedad de retos en este campo, proponemos abordar el problema mediante el uso de multi-modelos para generar estas prue-

bas. Basados en la extracción de modelos propuesta en Automated Extraction of Augmented Models for Android Apps (RIP tool) [LinanAutomatedApps] y CEL [Linares-Vasquez2017ContinuousTesting], proponemos una herramienta para generar tests de manera automática que tiene en cuenta los tres modelos (contexto, dominio y GUI). Al tener en consideración la información proporcionada por estos modelos, se generarán tests más fieles a la realidad, asegurando mayor cobertura en casos de prueba.

## 1.2 Objectives

- Generar casos de prueba de 'esquina' basados en las probables causas de error encontradas con los modelos.
- Generar casos de pruebas basados en los cambios contextuales de la aplicación (conectividad, batería, etc).
- Proveer al usuario con una visualización de la ejecución de las pruebas a través de una aplicación web.

## 1.3 Expected Results

Al finalizar el proyecto, será posible para el tester generar varios casos de prueba de manera automatizada, considerando los posibles factores que pueden influenciar la aplicación móvil. Esto reducirá el costo de tener que probar la aplicación de manera manual o programando los casos de prueba.

## 1.4 Reached Results

Se construyó una herramienta que permite al desarrollador generar casos de prueba que se pueden reproducir con el framework Espresso. Estos casos tienen en cuenta el modelo GUI extraído por RIP junto con el modelo de dominio y los cambios contextuales producidos en la aplicación. Se decidió usar Espresso debido a que es la herramienta oficial propuesta por Google, además de estar ampliamente documentado para su fácil modificación por parte del desarrollador/usuario.

## Trabajo relacionado y herramientas existentes

### 2.1 Generación automática de pruebas

Para la generación de pruebas existen diversas herramientas que abarcan el problema de distintas maneras. A continuación se presenta un análisis de las herramientas que se acercan más a nuestra propuesta, sin embargo difieren en que solo se basan en el modelo de GUI.

#### 2.1.1 Barista

Barista [Fazzini2017] es una herramienta *record and replay* en la cual el tester realiza una prueba de manera manual sobre la GUI de la aplicación. Después, un caso de prueba es codificado para poder ser reproducido en otros dispositivos múltiples veces. A pesar de que este enfoque provee una manera fácil de generar un caso de prueba, el tester se ve obligado a realizar manualmente cada prueba al menos una vez interactuando con la GUI.

#### 2.1.2 MobiGUITAR

A diferencia de Barista, MobiGUITAR [MobiGUITAR] realiza la interacción y extracción de eventos de la aplicación de manera automática usando GUI Ripping. Mediante esta técnica, se genera un modelo de máquina de estado que representa todos los eventos de la aplicación. A pesar de automatizar el paso de creación de la prueba, esta herramienta no tiene en cuenta los cambios contextuales del dispositivo.

#### 2.1.3 Firebase Test Lab

Este servicio [firebase] en la nube ofrecido por Google, permite al usuario subir el apk de la aplicación y correr varios tipos de pruebas en la aplicación. Sin embargo,

similar a las herramientas anteriores, se basa en exploración de GUI. Además, posee un costo adicional si se desea probar en un mayor número de dispositivos.

#### 2.1.4 Sapienz

Sapienz[**sapienz**] utiliza un enfoque multi-objetivo para la exploración y generación de casos de prueba. Esto hace que se obtengan tests mas cortos que maximizan su cobertura. Una ventaja de esta herramienta es que se pueden generar pruebas solo con el apk de la aplicación. Sin embargo no se tienen en cuenta los cambios contextuales de la aplicación.

# RIP Tests Generator

## 3.1 Solución propuesta

Como fue mencionado anteriormente, uno de los principales problemas a la hora de generar casos de prueba, es poder tener en cuenta los diferentes cambios de contexto y los tipos de entidades encontrados en la aplicación. Por esto, la generación de tests de esta herramienta está basada en los diferentes modelos encontrados con la herramienta **RIP**[**LinanAutomatedApps**]. Concretamente se tienen definidos tres modelos: GUI, contexto y dominio, los cuales se basan en lo propuesto en CEL[**Linares-Vasquez2017ContinuousTesting**].

El modelo GUI es extraído mediante la técnica GUI ripping, en donde se realiza una exploración de la interfaz de la aplicación, generando un diagrama de estados que representan las actividades y eventos de transición entre ellas. Para el modelo de dominio, se realiza un análisis de los componentes encontrados en los XML's de la aplicación, para así describir los datos que pueden ser introducidos o producidos. Finalmente, mediante el diagrama de contexto se conocen las condiciones externas que podrían afectar el correcto funcionamiento de la aplicación. Información como sensores, uso de batería, o conectividad es recolectada en este modelo.

Una vez se tiene la información sobre los diversos modelos, se propone crear **RIP Tests Generator**, una herramienta que genere de manera automática los archivos de prueba usando Espresso. Este es el framework sugerido por Google con el fin de escribir pruebas de para aplicaciones Android.

## 3.2 Arquitectura de la solución

**RIP Tests Generator** genera tests después de que **RIP** termina la ejecución, lo que permite el desacoplamiento de las dos herramientas. En este caso, **RIP** [**LinanAutomatedApps**] puede mejorar o cambiar sus algoritmos de exploración dinámica sobre aplicaciones móviles, manteniendo la compatibilidad con **RIP Tests Generator**. La figura 3.1 presenta el resumen del proceso de generación de tests a partir de los archivos obtenidos mediante la ejecución de RIP.



**Fig. 3.1:** Proceso de generación automática de tests

Test Generator es un componente acoplado con RIP que es ejecutado cuando se termina la exploración de la aplicación. Para esto, se creó la clase `TestCaseBuilder.java` que toma los datos proporcionados en el archivo `tree.json` y traduce los eventos a comandos de espresso.

Para la generación de pruebas se definieron los siguientes eventos, basados en los generados por RIP:

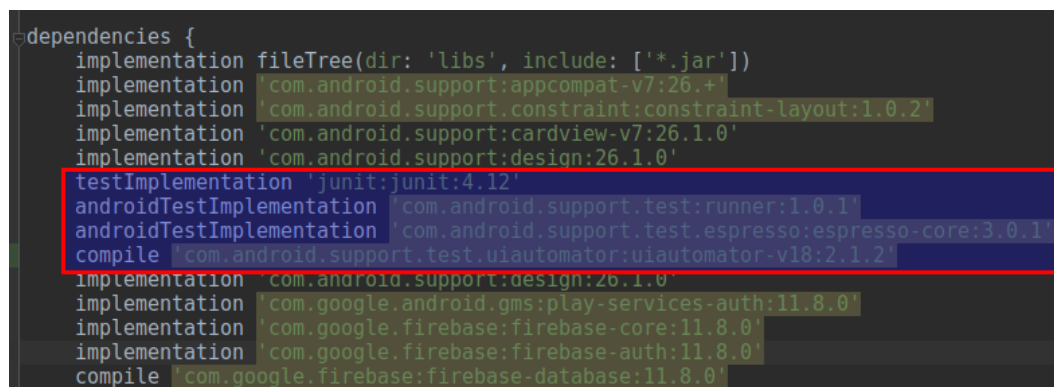
**TAP** : Cuando se tiene esta acción se traduce a un evento de hacer click en el botón o campo con el id proporcionado. Para implementar esta acción en espresso, se utiliza el comando `onView(withId(id)).perform(click())`.

**INPUT** : Al detectar un evento de campo de texto, con ayuda del modelo de dominio se detecta el tipo de dato a ingresar. Para esto se tiene la posibilidad de crear texto de manera aleatoria en un rango de caracteres normal o con una gran cantidad de estos con el fin de probar los limites de estos campos. Esta acción es implementada mediante el comando `onView(withId(id)).perform(replaceText(randomtext"),closeSoftKeyboard());`

### 3.3 Requisitos técnicos para la generación automática de pruebas

Con el fin de que el desarrollador no tenga ningún inconveniente usando la herramienta, se definieron los siguientes requisitos técnicos:

1. Al estar integrada con **RIP**[**LinanAutomatedApps**], se debe tener instalado ADB y correr el software en dispositivos rooteados o emuladores.
2. El desarrollador debe tener acceso al código fuente de la aplicación, ya que los archivos de prueba generados solo pueden ser ejecutados si son empaquetados junto al código fuente.
3. Es necesario hacer los cambios pertinentes en el build.gradle del proyecto para importar las dependencias utilizadas en los casos de prueba. Un ejemplo de esto es encontrado en la figura 3.2.



```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:appcompat-v7:26.+'
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'
    implementation 'com.android.support:cardview-v7:26.1.0'
    implementation 'com.android.support:design:26.1.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.1'
    compile 'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'
    implementation 'com.android.support:design:26.1.0'
    implementation 'com.google.android.gms:play-services-auth:11.8.0'
    implementation 'com.google.firebase:firebase-core:11.8.0'
    implementation 'com.google.firebase:firebase-auth:11.8.0'
    compile 'com.google.firebase:firebase-database:11.8.0'
```

**Fig. 3.2:** Ejemplo de archivo build.gradle con las dependencias necesarias para la ejecución de las pruebas

### 3.4 Generación automática de una prueba

Para generar una prueba se hace uso del archivo tree.json **RIP**, el cual contiene un grafo donde las actividades corresponden a los nodos y los links a los eventos que disparan una transición entre dos estados. Cada nodo contiene la información del modelo de dominio con el tipo de datos que se encuentran. Además, se tienen los diferentes indicadores de contexto en cada estado, donde se pueden ver datos como batería, conectividad de internet, etc.

Dicho archivo es el que se utiliza como insumo para generar los casos de prueba mediante RIP Tests Generator. Gracias a la librería JavaPoet [**JavaPoet**] se genera

un archivo .java que contiene el código fuente de la prueba. Para esto, se tiene en cuenta la información obtenida del grafo generado por RIP y se hace una traducción de cada acción a ser ejecutada en la prueba.

Mediante el modelo GUI se pueden generar los eventos de transición. Por ejemplo, si la acción que produjo un cambio de estado es un tap en el botón con id 'buttonJugar', entonces esto será traducido como un `onView(withId(R.id.buttonJugar)).perform(click());`. Para generar esto, se tiene un método auxiliar que retorna la línea de código correspondiente. Al analizar el grafo, según el tipo de acción encontrada se genera una línea de código diferente. Este proceso se puede evidenciar en el ejemplo de la figura 3.3.

Respecto a la implementación de la generación de código con JavaPoet se debe mencionar que las librerías correctamente declaradas son importadas de manera automática en el archivo generado. Además, el paquete donde debe ser ubicado el archivo fuente se inserta de manera correcta. Finalmente, después de cada acción, se toma una captura de pantalla que será posteriormente extraída del dispositivo para generar el reporte de visualización web.

El algoritmo propuesto para la generación de la prueba es el siguiente:

1. Tomar la secuencia de eventos generada por RIP y guardarlos en una lista.
2. Realizar la acción de tomar una captura de pantalla antes de cada transición.
3. Según cada evento generar la línea de código correspondiente en el lenguaje espresso.
4. Si se tiene un cambio de contexto en el nodo, realizar la traducción correspondiente.
5. Con la ayuda de JavaPoet, generar los campos usados por Espresso. Como lo son la `ActivityTestRule`, para indicar en que actividad iniciar la aplicación, y los permisos de escritura y lectura para las capturas de pantalla.
6. Finalmente, para la correcta ejecución de la prueba, se genera un archivo ejecutable que se encarga de empaquetar el apk, correr las pruebas encontradas en el paquete test, y extraer las capturas de pantalla del dispositivo.

Un ejemplo de la prueba obtenida como resultado puede ser evidenciado en la figura 3.4



Archivo generado por RIP

```
"nodes": [
  {
    "screenCapture": ".\\generated\\testfinadebug\\2018-10-0517:18:05.601.png",
    "imageName": "2018-10-0517:18:05.601.png",
    "buttons": [
      "com.ppg.spunky_java:id\\BJugar",
      "com.ppg.spunky_java:id\\BCrear",
      "com.ppg.spunky_java:id\\BUnirse"
    ],
    "currentFocus": "com.ppg.spunky_java\\com.ppg.spunky_java.MainActivity\\r\\n",
    "clickedButtons": [
      "com.ppg.spunky_java:id\\BJugar",
      "com.ppg.spunky_java:id\\BUnirse",
      "com.ppg.spunky_java:id\\BCrear"
    ],
    "name": "(0) com.ppg.spunky_java\\com.ppg.spunky_java.MainActivity\\r\\n",
    "activityName": "MainActivity",
    "model": [
      {
        "field": "com.ppg.spunky_java:id\\BJugar",
        "name": "Jugar",
        "type": "BUTTON"
      },
      {
        "field": "com.ppg.spunky_java:id\\BCrear",
        "name": "Crear juego",
        "type": "BUTTON"
      },
      {
        "field": "com.ppg.spunky_java:id\\BUnirse",
        "name": "Unirse"
      }
    ]
  }
]
```

Ejemplo de líneas de código generadas según caso



```
private static String clickButton2(String resId){
    String stm = "onView(withId(R.id."+resId+")).perform(click());\n\n";
    return stm;
}

private static String inputText(String resId){
    // onView(withId(R.id.editTextUnirse)).perform(replaceText("sdfsdf"),closeSoftKeyboard());
    Random rm = new Random();
    String input = "" + (char) (rm.nextInt( bound: 26) + 'A') + (char) (rm.nextInt( bound: 26) + 'a')
    + (char) (rm.nextInt( bound: 26) + 'a');

    String stm = "onView(withId(R.id."+resId+")).perform(replaceText(\""+input+"\"),closeSoftKeyboard());\n\n";
    return stm;
}
```



Código fuente final usando framework Espresso

```
takeScreenshot( name: "test"+i);
i++;
onView(withId(R.id.buttonJugar)).perform(click());

takeScreenshot( name: "test"+i);|
i++;
onView(withId(R.id.editTextApodo)).perform(replaceText( stringToBeSet: "Dau"),closeSoftKeyboard());
```

Fig. 3.3: Proceso de traducción y generación de prueba

```

@LargeTest
@RunWith(AndroidJUnit4.class)
public class RIPTest2 {
    @Rule
    public ActivityTestRule<MainActivity> mActivityTestRule2 = new
ActivityTestRule<MainActivity>(MainActivity.class);

    @Rule public GrantPermissionRule permissionRule =
GrantPermissionRule.grant(android.Manifest.permission.READ_EXTERNAL_STORAGE);
    @Rule public GrantPermissionRule permissionRule2 =
GrantPermissionRule.grant(Manifest.permission.WRITE_EXTERNAL_STORAGE);
    @Rule public GrantPermissionRule permissionRule3=
GrantPermissionRule.grant(Manifest.permission.WRITE_SECURE_SETTINGS);

    @Test
    public void mainActivityTest() {
        int i = 0;

        takeScreenshot("test"+i);
        i++;
        onView(withId(R.id.BUnirse)).perform(click());

        takeScreenshot("test"+i);
        i++;
        onView(withId(android.R.id.button1)).perform(click());

        onView(withId(R.id.editTextUnirse)).perform(typeText("Fnq"),closeSoftKeyboar

        takeScreenshot("test"+i);
        i++;
        onView(withId(R.id.buttonJugar)).perform(click());

        takeScreenshot("test"+i);
        i++;

        onView(withId(R.id.editTextApodo)).perform(replaceText("Dau"),closeSoftKeybo

```

**Fig. 3.4:** Prueba generada

## Conclusiones

Durante el desarrollo del proyecto se lograron la mayoría de los objetivos propuestos, y se construyó un generador automático de pruebas funcional. A continuación se presentan las conclusiones del proceso y el trabajo futuro.

### 4.1 Objetivos cumplidos

Gracias a la implementación de esta herramienta se obtuvo un producto inicial que genera pruebas basado en los modelos producidos por RIP. Durante el proceso se adquirieron conocimientos sobre exploración de aplicaciones móviles y generación de código java. Además, se logró el objetivo principal, integrar multi-modelos en la generación de pruebas. Se tuvieron en cuenta los 3 modelos, contexto, dominio y GUI, lo cual es el factor que diferencia la solución de otras ya existentes.

### 4.2 Limitaciones del desarrollo

- Como se mencionó anteriormente, la generación de pruebas está enfocada exclusivamente en aplicaciones Android. Esto porque es el sistema operativo donde se presentan mayores problemas de desfragmentación y ofrece la facilidad de explorar las aplicaciones mediante herramientas como ADB.
- Debido al alcance del proyecto, el desarrollador debe tener el código fuente de la aplicación para poder ejecutar la prueba
- La extracción del modelo de dominio está basada en los componentes básicos de Android, por lo tanto, los que sean personalizados o de otras librerías no serán reconocidos en dicho modelo.
- El desarrollador debe tener derechos de superusuario sobre el dispositivo o correr las pruebas en un emulador para realizar algunas acciones.

## 4.3 Trabajo futuro

El proyecto presenta varias oportunidades de mejora como lo son:

- Ejecución de la prueba sin tener acceso al código fuente.
- Extender los componentes que son detectados en los diferentes modelos.
- Tener una interfaz para el uso de la herramienta.
- Permitir al usuario ingresar valores personalizados para campos que lo requieran como los de usuario y contraseña.

## Índice de figuras

3.1	Proceso de generación automática de tests . . . . .	6
3.2	Ejemplo de archivo build.gradle con las dependencias necesarias para la ejecución de las pruebas . . . . .	7
3.3	Proceso de traducción y generación de prueba . . . . .	9
3.4	Prueba generada . . . . .	10

## Colophon

Este documento fue escrito en  $\text{\LaTeX}$  2<sub>ε</sub>. Usa el estilo *Clean Thesis* desarrollado por Ricardo Langner.

