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

Mejorando la generación de casos de test Espresso para aplicaciones Android

Tesis de Licenciatura en Ciencias de la Computación

Christian Ariel Ciccaroni

Director: Iván Arcuschin Moreno

Buenos Aires, 2020

RESÚMEN

En las aplicaciones móviles se debe testear no solo los objetos de negocio y los objetos que interactúan con el sistema operativo, sino que también la interfaz de usuario(UI). Esta clase de tests son conocidos como *tests de UI* y su objetivo es dar garantías de que la interfaz de la aplicación se comporte como se espera ante las interacciones realizadas en cada test sobre los distintos *widgets* que componen las pantallas.

Realizar tests correctos y completos de una vista puede llevar más tiempo que desarrollar la vista en si, por lo que en la práctica muchas veces no son escritos. En el caso particular de Android, el principal framework utilizado para este fin es Espresso, y la escritura de los tests es responsabilidad del desarrollador, ya que se requiere conocimiento técnico de Android y del código de la aplicación a testear.

MATE es una herramienta capaz de interactuar con los widgets de la pantalla de una aplicación utilizando el sistema de accesibilidad de Android, sin necesidad de interactuar con un humano. Una vez finalizada las exploraciones se genera un archivo con las acciones realizadas en cada paso y los widgets sobre los que se realiza cada acción.

ETG es un prototipo académico que se ocupa de generar casos de prueba Espresso recibiendo como input la descripción de una serie de acciones realizadas en los widgets de una aplicación. Si bien el input puede tener cualquier procedencia, en esta tesis se utiliza el output de *MATE* ya que dicha herramienta realiza acciones sobre widgets particulares al igual que Espresso. El objetivo de esta herramienta es lograr documentar casos funcionales de una navegación correcta sobre una aplicación y, además, servir como punto de partida para la escritura de nuevos casos de tests.

Se trabajó en mejorar *ETG* para conseguir casos de tests fieles al input recibido. Se realizaron pruebas sobre aplicaciones creadas específicamente para la búsqueda de casos problemáticos donde la herramienta no se comporte correctamente y los casos generados presenten errores. Luego, se evaluó el rendimiento de la herramienta en una aplicación industrial de gran tamaño y, finalmente, se definen ideas y posibilidades de uso en la industria.

Palabras claves: Android, Espresso, Testing, UI, Automático.

Índice general

1. INTRODUCCIÓN

1.1. Motivación

A pesar de su creciente popularidad, las aplicaciones móviles, comunmente denominadas apps, tienden a tener defectos que luego se manifiestan como fallas o crashes al usuario final. El desarrollo de tests para aplicaciones móviles se dividen en 2 grandes grupos: tests de unidad y tests de interfaz.

Los tests de unidad verifican que los objetos de negocio se comporten como se espera mientras que los tests de interfaz verifican que la interfaz de la aplicación responda como se espera ante ciertos estímulos. Sin embargo, la escritura de tests suele ser dejada para el final del desarrollo y, dado que la escritura de los mismos es un proceso costoso en tiempo, en la práctica no siempre se realizan o se escriben a medias produciendo así tests incompletos o incluso incorrectos. Los tests de interfaz suelen ser los últimos que se escriben y son los más complejos de escribir, si acaso se escriben.[1]

Múltiples herramientas han sido propuestas en años recientes para realizar testing automático de aplicaciones Android[2, 3]. Monkey[4], es una herramienta popular entre los desarrolladores para testing de apps basada en exploración puramente aleatoria. Lamentablemente, Monkey no permite al usuario re-ejecutar secuencias de eventos(i.e., casos de test) lo cual puede ser crítico para entender la causa de un crash. De manera similar, muchas otras herramientas para testear automáticamente aplicaciones Android no generan casos de test en un formato legible y re-ejecutable. Por ejemplo, Sapienz[5] devuelve una secuencia de acciones atómicas que sólo pueden ser re-ejecutadas por una máquina, *MATE*[6] genera un reporte de accesibilidad, mientras que Dynodroid[7] y Stoat[8] sólo reportan los crashes encontrados. Este hecho complica la adopción de dichas herramientas por parte de los desarrolladores que desean preservar los casos de test interesantes y poder correrlos de manera periódica (i.e, en un pipeline de integración continua). De hecho, tener casos de test legibles y re-ejecutables permitiría a los desarrolladores utilizarlos como inspiración o punto de partida para crear casos de test completos o modificar tests existentes.

Existen herramientas que intentan ayudar a la generación de estos tests como Espresso Test Recorder[9], un grabador de casos de tests para android, que permite grabar las acciones que hace un usuario en pantalla para generar casos de Espresso. Sin embargo, es muy limitada las acciones que ofrece realizar sobre la pantalla por lo qué solamente sirve como una guía o punto de partida para el desarrollo completo de un test.

Es en este contexto que entra *ETG*, un prototipo académico que intenta generar casos de test de Espresso[10] a partir de un input de exploración de una aplicación en particular. Actualmente, trabaja recibiendo el output de una extensión de *MATE* que escribe la exploración realizada sobre una app en un formato JSON. Gracias a esta herramienta, se podrían documentar casos funcionales de una navegación correcta de la aplicación, para poder correr en regresiones y asegurar, al menos parcialmente, el correcto funcionamiento de una aplicación, incluso luego de que se desarrollen cambios sobre la misma.

1.2. Preliminares

En esta sección veremos una introducción a los conceptos básicos del desarrollo de aplicaciones móviles en Android y su framework de desarrollo; así como también introduciremos las herramientas *MATE* y *ETG*.

1.2.1. Android

Android es un sistema operativo desarrollado por Google y basado en Linux, diseñado para dispositivos móviles con pantalla táctil. Posee un framework de desarrollo basado en extensión de clases que representan diversos componentes del sistema para que un desarrollador los extienda y agregue el comportamiento deseado. A continuación revisaremos los principales aspectos del framework.

Activity

Un *activity* es el punto de entrada de interacción con el usuario y representa una pantalla individual con una interfaz de usuario. Es la encargada de crear una ventana en una aplicación y notificar al desarrollador para que pueda agregarle el contenido deseado, según el estado actual de la aplicación. Toda *activity* tiene un ciclo de vida, desde que es creada y visualizada por el usuario; hasta que es destruida por ser abandonada y no requerida por el usuario. Se trata de la unidad base de construcción de una aplicación, ya que una navegación puede ser construida como un flujo de activities.

UI

Es la interfaz de usuario. Se refiere a lo que el usuario ve en pantalla y como puede interactuar con ella.

View

Es la unidad básica de construcción de una UI, tambien se les llama *widgets*. Cualquier componente que se muestre en pantalla es una *view*. En Android, las vistas son una estructura jerárquica en forma de árbol y, dependiendo el tipo, pueden tener otra vista como único padre o varias vistas como hijas. Cualquier tipo de *view* que exista extiende de la clase *View*. Toda *view* posee un ID para identificarla, pero el mismo no necesariamente es único. Es perfectamente normal, por ejemplo, que todas las vistas de una lista tengan el mismo ID, pues es la misma vista que muestra información distinta.

Existen distintos tipos de vistas, las más básicas que son relevantes para el contenido de esta tesis son:

- *Button:* Botón clickeable por el usuario
- *TextView:* Vista que contiene un texto que se muestra al usuario
- *EditText:* Vista que contiene un texto editable por el usuario
- *ViewGroup:* Es un tipo de vista que puede contener vistas hijas. Ejemplos relevantes de este tipo de vista son:

- *ListView:* Vista usada para mostrar listas. Para cada item de la lista utiliza una vista con la información del mismo, lo que resulta ser poco eficiente para listas extensas ya que maneja en memoria una cantidad de vistas muy grandes cuando solo necesita mostrar una cantidad acotada según el tamaño de la pantalla.
- *Recyclerview:* Es también una vista usada para mostrar listas y el reemplazo de *ListView*. A diferencia de su predecesora, funciona utilizando la cantidad de vistas necesarias según el tamaño de la pantalla y, a medida que el usuario realiza un scroll sobre la lista, se re-usan las mismas vistas cambiando la información que muestran.
- *FrameLayout, LinearLayout, ConstraintLayout*: Son los *ViewGroups* básicos utilizados para construir vistas y, a su vez, también son vistas. Generalmente suelen ser los elementos root y dentro de ellos se organizan las distintas vistas que pueden contener. La diferencia esencial entre estas vistas radica en como se pueden organizar sus hijos.
  - FrameLayout: Generalmente se usan cuando es necesario un container de una vista que se agregará de forma dinámica, aunque puede contener varios hijos. Si se agrega mas de un hijo, los mismos se superpondrán centrados en su padre uno sobre el otro.
  - LinearLayout: Como su nombre lo indica, es un contenedor lineal de vistas. Los hijos que se le agreguen estaran organizados linealmente. Puede ser de manera vertical o horizontal.
  - ConstraintLayout: Es el layout con mayor poder expresivo, permite definir restricciones sobre la ubicación relativa o absoluta de cada hijo respecto a su padre y sus hermanos.

Las vistas generalmente son descriptas en formato XML, en el Listing 1.1 podemos ver un ejemplo de una vista de login, la cuál se compone de un ViewGroup(*ConstraintLayout*) y 3 childs: un widget para mostrar texto(*TextView*), otro para input de texto(*EditText*) y un boton(*Button*). Cada vista posee atributos propios que permite definir sus propiedades, como ser ancho, alto, texto a mostrar, posición relativa del widget respecto a otro, etc. En la Figura 1.1 se encuentra la vista resultante de dicho XML.

Listing 1.1: Ejemplo de XML describiendo una vista en Android

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <androidx.constraintlayout.widget.ConstraintLayout
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      xmlns:app="http://schemas.android.com/apk/res-auto"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      android:padding="24dp">
8
9      <TextView
10         android:id="@+id/title"
11         android:layout_width="wrap_content"
12         android:layout_height="wrap_content"
13         style="@style/TextAppearance.AppCompat.Title"
14         android:text="Login"
15         app:layout_constraintTop_toTopOf="parent"
16         app:layout_constraintStart_toStartOf="parent" />
17
```

```
18      <EditText
19          android:id="@+id/input"
20          android:layout_width="0dp"
21          android:layout_height="wrap_content"
22          android:layout_marginTop="24dp"
23          android:hint="usuario"
24          app:layout_constraintTop_toBottomOf="@id/title"
25          app:layout_constraintStart_toStartOf="@id/title"
26          app:layout_constraintEnd_toEndOf="parent" />
27
28      <Button
29          android:id="@+id/enterButton"
30          android:padding="16dp"
31          android:layout_width="wrap_content"
32          android:layout_height="wrap_content"
33          style="@style/Widget.AppCompat.Button.Colored"
34          android:layout_marginTop="40dp"
35          android:text="Ingresa"
36          app:layout_constraintEnd_toEndOf="parent"
37          app:layout_constraintTop_toBottomOf="@id/input" />
38
39  </androidx.constraintlayout.widget.ConstraintLayout>
```

Fig. 1.1: Ejemplo de vista

También es posible, por ejemplo, agregar vistas hijo a un *ViewGroup* de manera programática, como se oberva en el listing Listing 1.2.

Listing 1.2: Agregado de un *ImageView* como hijo a un *ViewGroup* por código

```
1 View someView = findViewById(R.id.someView);// finds the required view
2 someView.addView(new ImageView());//creates an ImageView and add it to someView
```

Emulador

Un emulador permite emular un teléfono corriendo Android en una computadora, sin necesidad de una unidad física.

ADB

Android Debug Bridge, es una herramienta de línea de comandos que permite comunicarse, debugear y enviar comandos a cualquier dispositvo o emulador que corra Android.

Permite acceso al shell de unix del sistema operativo.

Sistema de accesibilidad

El sistema de accesibilidad de Android es una API del sistema operativo que permite conseguir información del contenido que se está mostrando en pantalla en un momento dado. Mediante esta API es posible conseguir para cada widget mostrado en pantalla

- ID
- Clase del widget
- Información del widget
  - Si es clickeable
  - Childs y parents
  - Texto, en caso de tenerlo
  - Posición en la pantalla
- Acciones que pueden realizarse sobre el ítem

1.2.2. Clasificación de tests

La responsabilidad de un test es verificar los requerimientos funcionales de un conjunto de objetos y la interacción entre ellos. A grandes rasgos, existen 2 tipos de tests:

- Caja blanca: Son tests que se realizan con conocimiento de la implementación y estructura interna de los objetos que se encuentran bajo prueba. Por lo que se puede no solo asegurar el resultado luego de un input dado, sino que también es posible probar detalles de implementación y verificar que las interacciones entre objetos sea la deseada.
- Caja negra: Son tests que se realizan sin conocer detalles de implementación, se ve la unidad a testear como una caja negra que produce un output para un input dado.

1.2.3. Tipos de test en Android

En Android podemos clasificar los tests según los componentes con lo que interactúan los objetos a testear.

- Tests no instrumentados: Son tests de unidad que permite testear de manera ágil objetos que no interactuan con el sistema operativo. Los tests son ejecutados en una maquina virtual java del mismo ordenador que corre los tests. No es necesario un dispositivo o emulador con Android.
- Tests instrumentados: Son tests de unidad que deben ejecutarse en dispositivos físicos o en emuladores ya que interactúan con las APIs de Android. Permite realizar tests de objetos que interactúan de alguna manera con el sistema operativo, por ejemplo permite comenzar activities como parte del test, interactuar con ellas, o incluso utilizar recursos del sistema como las API de bluetooth.

1.2.4. Mock

En la jerga, un *mock* es un objeto que puede reemplazar a otro y se comporta de la manera en que se lo configura. Por ejemplo, es posible tener un *mock* de un generador de números aleatorios para realizar un test que siempre generé el número *42*. Esto permite tener consistencia en los tests y obtener el mismo resultado en cualquier ejecución del mismo. Se dice que algo está *mockeado* si se lo reemplaza por un *mock*.

1.2.5. Espresso

Es un Framework utilizado para realizar tests de UI de caja blanca en Android. Dado que estos tests permiten probar la interfaz de una aplicación, los mismos deben ser tests instrumentados para interactuar con el sistema operativo. Para implementar estos tests es necesario tener conocimiento sobre la aplicación, sus activities y sus vistas, por lo que debe ser realizado por desarrolladores. El framework permite seleccionar vistas en la pantalla actual y realizar acciones o aserciones sobre las mismas a través de una API.

Existen 4 conceptos fundamentales en este framework:

- *ViewInteraction:* Es la abstracción del framework para realizar acciones y aserciones sobre vistas. Toda interacción con una vista se crea utilizando un *matcher* que permite definir unívocamente a la vista sobre la cuál se va a realizar la interacción.
- *ViewMatcher:* Es una colección de sentencias que permite describir una vista. Un *ViewInteraction* es creado a partir de un *ViewMatcher*. En caso de ejecutarse exitósamente encontrará la vista con la que se quiere interactuar, mientras que generará un error si no es posible encontrar una vista que cumpla con la descripción.
- *ViewAction:* Especifica una acción sobre una vista, como ser un click, un swipe, ingresar un texto, etc.
- *ViewAssertion:* Especifica una aserción sobre una vista, por ejemplo: El texto que contiene, si está visible o no, etc.

Los tests instrumentados son métodos anotados como tests pertenecientes a clases ubicadas dentro del directorio *src/androidTest*. Cada test es anotado con *@Test* para indicar al framework que dicho método es un test. Existen, además, las anotaciones *@Before* y *@After* que indican métodos que deben correrse antes y después de cada test de la clase. Se utilizan para el setup y el teardown de cada test. Las clases son anotados con *@RunWith*, para especificar el Runner que se encargará de correr los tests instrumentados. Se puede ver un ejemplo en el Listing 1.3

Listing 1.3: Ejemplo de test Espresso

```
1  private int someStateVariable;
2
3  @Before
4  public void setUp() {
5      someStateVariable = 0;
6  }
7
8  @After
9  public void teardown() {
```

```
10        System.out.println("someStateVariable has the value " + someStateVariable);
11    }
12
13    @Test
14    public void widgetTextIsHello() {
15        //finds the button view on screen
16        ViewInteraction buttonView = onView(withId(R.id.buttonView);
17        //performs click on button
18        buttonView.perform(click())
19        //asserts that button is displayed
20        buttonView.check(matches(isDisplayed()))
21    }
```

Un *test Espresso* es una serie de *acciones* y *aserciones* sobre vistas seleccionadas.

1.2.6. MATE

Se trata de una herramienta para realizar testing automatizado de aplicaciones Android. Tiene la finalidad de probar que tan *accessibility friendly* es una aplicación, para esto explora la misma utilizando el sistema de accesibilidad de Android y corre chequeos para medir que tan accesible es para usuarios con capacidades disminuidas. Por ejemplo uno de los chequeos evalúa y reporta las vistas que no tienen *content description*, un campo usado para describir el contenido de, por ejemplo, imágenes.

MATE tiene visibilidad sobre los widgets que se muestran en pantalla y realiza las acciones en base a los widgets, concepto muy cercano a Espresso. Es por esto que elegimos esta herramienta para realizar las exploraciones, extendiendo la misma para permitir exportar en formato JSON las acciones realizadas en cada widgets durante la exploración.

El framework tiene una arquitectura tipo cliente-servidor,

Servidor

Encargado de enviar los comandos *ADB* al dispositivo en el que se corre los tests.

Cliente

Corre un test instrumentado sobre un emulador o dispositivo y explora la aplicación. Se conecta al servidor para ejecutar los comandos de *ADB*. Sus principales responsabilidades son:

- **Representación de los widgtes en pantalla:** Para conseguir información del contenido de la pantalla en cada momento, *MATE* utiliza el sistema de accesibilidad de Android que expone información de la pantalla con el objeto *AccessibilityNodeInfo*. Este objeto representa un nodo de una ventana y las acciones que es posible realizar sobre el mismo. Desde el punto de vista del sistema de accesibilidad de Android, el contenido de una pantalla es representado mediante un árbol de estos nodos.

  Iterando cada uno de estos nodos es posible obtener información de cada uno de los widgets mostrados en pantalla

- **Listar las acciones posibles:** Cada *AccessibilityNodeInfo* expone la información necesaria para determinar qué acciones se puede realizar sobre el mismo, nos permite saber si un widget es clickeable, scrolleable, editable, etc.

- **Elegir widget y acción a realizar:** De acuerdo al algoritmo que se esté ejecutando, *MATE* elige con qué widget interactuar y que acción realizar sobre el mismo.
- **Realizar la acción elegida sobre el widget elegido:** Una vez determinado el widget y acción a realizar, *MATE* se encarga de ejecutarlo en el dispositivo. Este se logra utilizando la librería de automatización *UiAutomator[11].*
- **Exportar exploración:** La exploración termina una vez alcanzado un timeout especificado y la exploración se envía al servidor para su almacenamiento en formato JSON, detallando la siguiente información:
  - Si la exploración detectó un crash en la aplicación.
  - Acciones realizadas.
    - Tipo de acción realizada: swipe, click, long click, tipeado de texto, etc.
    - Widget sobre el que se realizó la acción, ubicación en pantalla y atributos que permiten saber el estado del mismo.
    - childs y parents del widget sobre el que se realizó la acción.

1.2.7. Espresso Test Recorder

(*ETR*) es una herramienta desarrollada por Google que permite grabar acciones realizadas por un humano sobre un emulador o dispositivo corriendo android, especificar unas pocas aserciones luego de cada acción y finalmente genera un caso de test de Espresso. Sin embargo, esta herramienta no es completa ya que los casos generados deben ser retocados por el programador, sirven como base y guía para la generación de un test.

1.2.8. Espresso Test Generator

ETG es un generador de casos de test de Espresso, basado de Espresso Test Recorder.

Input

El input de la aplicación es un archivo en formato JSON que describe acciones realizadas sobre widgets en pantalla. Además, para cada acción realizada se incluye información de todos los *widget* padre e hijos del receptor de la misma.

Output

Se genera un archivo *java* que incluye un test de Espresso que intenta realizar cada acción descripta en el input de la aplicación.

Dado que la definición de un *ViewMatcher* para una vista puede resultar ambigua, existe la posibilidad de configurar *ETG* para que ejecute cada acción dentro de un *try-catch*. De esta manera, se logra evitar la finalización del test por este error y continuar la ejecución del mismo.

Template

Un *template* un esqueleto de código con *placeHolders* que pueden ser reemplazados cada vez que se lo utiliza, es la base de las clases de test generados por la herramienta. Define, entre otras cosas, el nombre de la clase de test, el nombre del test y métodos comunes usados.

Heurística de prunning

Ya sea por tener expresiones ambiguas, incorrectas o inconsistentes, existen muchos casos de tests generados por *ETG* que fallan. Para intentar mitigar esta situación y conseguir tests exitosos, se implementa una heurística de punto fijo para remover acciones fallidas buscando la convergencia a un test case sin errores. La base de esta heurística es ejecutar el caso de test generado, removiendo las acciones que fallan en cada ejecución, hasta que el mismo sea exitoso.

2. DESARROLLO

Se realizaron pruebas de concepto de *ETG* utilizando aplicaciones focalizadas en cierto tipo de pantallas, para buscar casos donde no fuera posible generar tests fieles a la exploración realizada por *MATE*. Se desarrollaron 2 aplicaciones de muestra para este fin:

- **ABC-APP:** Aplicación con solo una lista de ítems, donde cada ítem muestra una letra del abecedario. Al hacer click sobre cualquier ítem se muestra un diálogo indicando que letra se clickeó. Se pueden ver en la Figura 2.1 y Figura 2.2 las pantallas de esta app.
- **Movies & Tv!:** Aplicación que muestra en pantalla listas horizontales con información de películas y series. Al realizar click en un ítem de la lista se accede al detalle de la misma en una nueva pantalla. También, es posible expandir cualquiera de las listas horizontales viendo en una nueva pantalla una grilla con todas películas o series relacionadas. Se pueden ver en la Figura 2.3, Figura 2.4, Figura 2.5 y Figura 2.6 las pantallas de esta app.

Luego, se atacaron las distintas limitaciones encontradas al querer realizar tests para estas aplicaciones y, finalmente, se probó la herramienta en una aplicación del mundo real para analizar su potencial. Por cuestiones de privacidad esta aplicación quedará anónima.

- **Payment Processing S.A.:** Aplicación industrial relacionada con el pago mediante tarjetas. La misma consta de más de 30 pantallas, pero se limitó el acceso a algunas de ellas por lo que solo 25 serán accesibles. Este recorte se debe a que hay pantallas que interactuan con dispositivos bluetooth pero no es posible realizar esto utilizando un emulador. Esta aplicación tendrá un carácter anónimo en el desarrollo de esta tesis.

  Más adelante se discutirán los cambios y adaptaciones realizados en la app.

2.1. Adaptación de ETG

Se realizaron algunos cambios y configuración mínimas en *ETG* para el desarrollo de esta tesis:

- Como buscamos minimizar los errores que se presentan, se modificó *ETG* para que no aplique su heurística de prunning, ya que queremos saber exactamente qué acciones fallan, por qué fallan y corregir la generación para que no fallen.
- El input de la herramienta será el resultado de una exploración realizada por *MATE*.
- Para tener una métrica de cuantas acciones fallan en un test creado, se agregó un contador de acciones fallidas sobre el *template*. Esto funciona en conjunto a la posibilidad de ejecutar cada acción dentro de un *try-catch* pues el contador se incrementa en 1 cada vez que una excepción es generada dentro del mismo. Además, se agregó un *teardown* en el test para logear la cantidad de errores y poder visualizarlos.

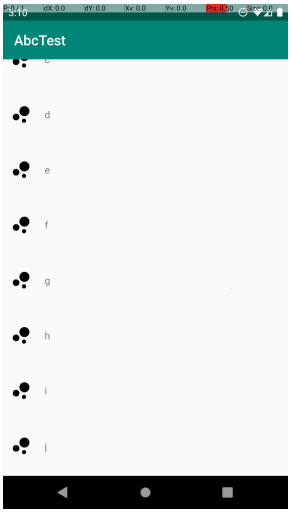

Fig. 2.1: Lista de items en ABC-APP

Fig. 2.2: Dialogo al clickear un item en ABC-APP

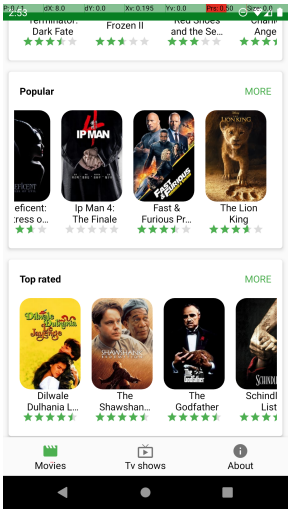

Fig. 2.3: Vista de peliculas en Movies & Tv!

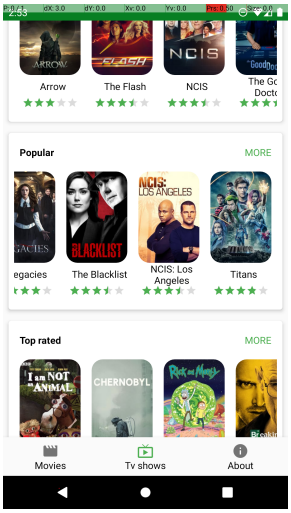

Fig. 2.4: Vista de series en Movies & Tv!

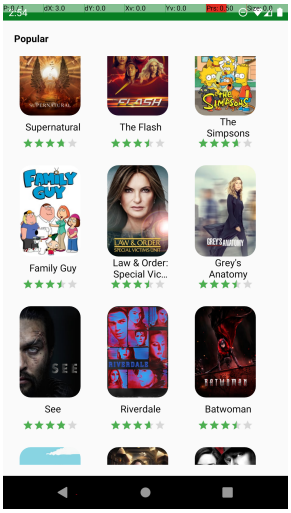

Fig. 2.5: Vista de todas las fuentes Movies & Tv!

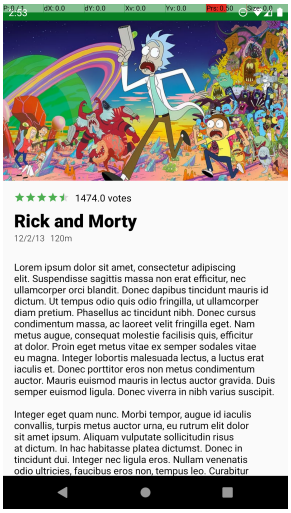

Fig. 2.6: Detalle de un ítem en Movies & Tv!

2.2. Problemas encontrados

A continuación, se verán los problemas que se encontraron, el motivo por el cual sucedían y la solución aplicada para mitigarlos.

2.2.1. Gesto swipe

Un *swipe* es el gesto que se realiza sobre la pantalla con el dedo para desplazar el contenido de la misma, y puede ser tanto vertical como horizontal.

Utilizando la app ABC-APP, que posee una larga lista de ítems, se descubrió rápidamente una gran cantidad de fallas en los tests generados cuando se pretendía buscar vistas con las que interactuar luego de haber realizado 1 o más gestos de *swipe*. Los *ViewMatcher* fallaban en ubicar vistas dado que las mismas no se encontraban visibles en pantalla. Esto sucedía debido a que el *swipe* realizado en *ETG* no era fiel al realizado por *MATE*.

La forma en la que un gesto *swipe* se realiza sobre un widget determinado en cada una de las herramientas es la siguiente:

- En *MATE* hay 2 posibles casos:
  - Es posible definir la clase del *widget* receptor de la acción. En este caso se calcula el centro del mismo y, desde allí, se realiza el gesto en la dirección correspondiente en una distancia de 300 píxeles.
  - No es posible definir la clase del *widget* receptor de la acción. En este caso se desliza desde el centro de la pantalla en la dirección correspondiente hasta los bordes de la misma.
- En *ETG*, se utiliza el método swipeUp, swipeDown, swipeLeft o swipeRight de Espresso según el caso. El funcionamiento de estos métodos es el siguiente:
  - Ubicar el centro del *widget* receptor de la acción.

- Realizar el gesto *swipe* en la dirección correspondiente hasta alcanzar borde del mismo. Esto implica que la distancia recorrida por el gesto es un 50 % del ancho o alto del *widget* en cuestión.

Claramente, un swipe de *MATE* y un swipe de *ETG* realizaban un recorrido completamente diferente.

Para lograr fidelidad entre el swipe realizado por cualquier herramienta y el swipe realizado por *ETG*, es necesario conocer las coordenadas origen y destino exactas de la pantalla sobre las que se realizó el *swipe*.

Cambios en MATE

Se agregó un objeto **Swipe** al modelo que representa la acción junto a sus coordenadas origen y destino, para así disponer de esta información al momento de generar los test cases en *ETG*. Esta información también se agregó al output de la exploración.

Cambios en ETG

En lugar de utilizar los métodos que provee Espresso para realizar un *swipe* se creó un *swipe custom*(Listing 2.1). El mismo hereda de la clase *GeneralSwipeAction* provista por la API de Espresso y puede utilizar las coordenadas origen y destino de la acción para realizar el gesto.

Listing 2.1: ETG swipe action fast

```
1  @NotNull
2  private ViewAction getSwipeAction(final int fromX, final int fromY, final int toX, final int
       toY) {
3      return ViewActions.actionWithAssertions(
4                      new GeneralSwipeAction(
5                              Swipe.FAST,
6                              new CoordinatesProvider() {
7                                  @Override
8                                  public float [] calculateCoordinates(View view) {
9                                      float [] coordinates = {fromX, fromY};
10                                     return coordinates;
11                                 }
12                             },
13                             new CoordinatesProvider() {
14                                 @Override
15                                 public float [] calculateCoordinates(View view) {
16                                     float [] coordinates = {toX, toY};
17                                     return coordinates;
18                                 }
19                             },
20                             Press.FINGER));
21     }
```

Sin embargo, se observó que esto no era suficiente para ser fiel al *swipe* realizado. Las acciones que realiza Espresso tienen como origen y destino el recorrido realizado por el dedo durante el desplazamiento pero una vez levantado el mismo el comportamiento de android es seguir desplazando ligeramente la pantalla. Este desplazamiento adicional genera una perdida de fidelidad entre las herramientas. Se observó que cuando la acción se realiza

como un *swipe rápido*, el movimiento que se realiza más allá del deseado es realmente significativo, pero si se realiza lentamente es prácticamente nulo. Por este motivo se ajustó la acción *swipe* cambiando la velocidad de *FAST* por *SLOW*, como se ve en el Listing 2.2.

Listing 2.2: ETG swipe action slow

```
@NotNull
private ViewAction getSwipeAction(final int fromX, final int fromY, final int toX, final int
    toY) {
  return ViewActions.actionWithAssertions(
                new GeneralSwipeAction(
                        Swipe.SLOW,
                        new CoordinatesProvider() {
                            @Override
                            public float [] calculateCoordinates(View view) {
                                float [] coordinates = {fromX, fromY};
                                return coordinates;
                            }
                        },
                        new CoordinatesProvider() {
                            @Override
                            public float [] calculateCoordinates(View view) {
                                float [] coordinates = {toX, toY};
                                return coordinates;
                            }
                        },
                        Press.FINGER));
  }
```

Además, se cambió la vista receptora del *swipe*. Como el gesto se realiza en coordenadas específicas de la pantalla, no tiene sentido realizarlo sobre un *widget* particular. El receptor de cualquier *swipe* realizado por *ETG* es ahora el widget root de la vista mostrada en pantalla.

Luego de cada *swipe* realizado fue necesario agregar un ligero delay en la ejecución del test, pues Espresso pasaba a ejecutar la siguiente acción del test cuando el *swipe* realizado en la acción anterior aún no había concluido el movimiento generado por la inercia. Esto se mitigó realizando un sleep de *1500ms* luego del desplazamiento realizado.

Finalmente, se puede ver en el Listing 2.3 un ejemplo de las sentencias generadas por *ETG* para realizar un *swipe*. Las coordenadas usadas para el método getSwipeActions son las coordenadas del gesto informado en el input la herramienta.

Listing 2.3: ETG swipe example

```
ViewInteraction root = onView(isRoot());

try {
    root.perform(getSwipeAction(540, 897, 540, 1794));
} catch (Exception e) {
    System.out.println(buildPerformExceptionMessage(e, 1));
}

waitToScrollEnd();
```

2.2.2. Clicks

Un *click* es el gesto de presionar una vista y soltarla rápidamente.

Utilizando la app ABC-APP se descubrió que fallaba el método *click()* de Espresso en vistas que se encontraban parcialmente visibles. Se puede ver en la Figura 2.1 un ejemplo donde la acción *click* fallaría para una vista parcialmente visible como lo es el item con la letra **c**, pero es perfectamente válido realizar la acción en dicha vista, tanto para un usuario humano como para un explorador automático.

Cambios en MATE

No se realizaron cambios en *MATE*.

Cambios en ETG

En lugar de utilizar los métodos que provee Espresso para realizar un *click* sobre un *widget* se creó un click custom. Sin embargo, la clase *GeneralClickAction* que provee la API de Espresso para realizar esta acción tiene prefijada la restricción de que la vista tiene que estar visible en al menos un 90 %. No existe documentación al respecto de esta decisión, y se puede constatar que removiendo esta restricción la acción resulta exitosa sin inconvenientes. Se estima que es una decisión de diseño, ya que en un caso de test escrito por un humano se puede manejar la vista correctamente para que esté en un estado donde la vista se encuentra completamente visible. Es por este motivo que se creó un nuevo *template* de una nueva clase con nombre *ClickWithoutVisibilityConstraint*, la misma es una copia de *GeneralClickAction* sin la restricción de visibilidad. La diferencia entre estas clases se encuentra únicamente en el método *getConstraints*. Ver el Listing 2.4 y el Listing 2.5.

Listing 2.4: getConstraint en GeneralClickAction

```
1 public Matcher<View> getConstraints() {
2     Matcher<View> standardConstraint = isDisplayingAtLeast(90);
3     if (rollbackAction.isPresent()) {
4         return allOf(standardConstraint, rollbackAction.get().getConstraints());
5     } else {
6         return standardConstraint;
7     }
8 }
```

Listing 2.5: getConstraint en ClickWithoutVisibilityConstraint

```
1 public Matcher<View> getConstraints() {
2     Matcher<View> standardConstraint = isDisplayed();
3     if (rollbackAction.isPresent()) {
4         return allOf(standardConstraint, rollbackAction.get().getConstraints());
5     } else {
6         return standardConstraint;
7     }
8 }
```

En el Listing 2.6 puede verse un caso de test que realiza un *click* en una vista cuya clase es *Button*, con texto *OK!* y que se encuentra visible en pantalla. La instanciación

de la clase *ClickWithoutVisibilityConstraint* para realizar el *click* se la puede observar en el Listing 2.7.

Listing 2.6: Ejemplo de click

```
1   ViewInteraction android_widget_Button =
2       onView(allOf(classOrSuperClassesName(is("android.widget.Button")),
3                    withText(equalToIgnoringCase("OK!")),
4                    isDisplayed())) ;
5
6   try {
7       android_widget_Button.perform(getClickAction());
8   } catch (Exception e) {
9       System.out.println(buildPerformExceptionMessage(e, 6));
10  }
```

Listing 2.7: Creación de acción Click

```
1 @NotNull
2 private ClickWithoutVisibilityConstraint getClickAction() {
3     return new ClickWithoutVisibilityConstraint(
4         Tap.SINGLE,
5         GeneralLocation.VISIBLE_CENTER,
6         Press.FINGER,
7         InputDevice.SOURCE_UNKNOWN,
8         MotionEvent.BUTTON_PRIMARY);
9 }
```

En el caso del LongClick sucede lo mismo y la solución es completamente análoga. Se creó la clase *LongClickWithoutVisibilityConstraint*.

2.2.3. Back Pressed

Se detectó que durante la exploración de *MATE* existían casos donde se elige presionar el botón *Back* de Android, pero él mismo fallaba sin advertencia, por lo que la exploración seguía y generaba un output donde se presionaba el botón de *Back* sin efecto real. Además, se constató que si se reintentaba la acción *back* suficientes veces luego de haber fallado la misma funcionaba.

Este es un error de la librería UI-Automator y hacía que fuese imposible generar casos fieles con *ETG* cuando un *back* no surte efecto en *MATE* pero el mismo es parte del output generado.

Cambios en MATE

Para mitigar este error en UI-Automator, se agregó un total de 10 reintentos para el evento *Back* si la acción falla. Y en caso caso de que no se pueda realizar exitosamente se tira una excepción, comenzando así una nueva exploración.

Cambios en ETG

No se realizaron cambios en *ETG*.

2.2.4. Vistas no válidas

La vista que una aplicación decide mostrar en pantalla es agregada como hijo directo de otra vista que pertenece al sistema operativo. Se detectó que *MATE* realiza acciones sobre vistas del sistema operativo que no son accesibles desde la aplicación ni desde Espresso, por lo que cualquier test que intente realizar una acción sobre alguna de estas vistas fallaría.

Cambios en MATE

Se eliminan del output de *MATE* aquellos widgets pertenecientes al sistema operativo y que no pertenecen a la aplicación bajo análisis. Los mismos se identifican por su *id*, que comienzan con el prefijo **android:id**

Cambios en ETG

No se realizaron cambios en *ETG*.

2.2.5. Receptor de acción

Cuando un usuario hace una acción en una vista generalmente es una hoja, por ejemplo un botón. Más aún, en la práctica un usuario hace *click* en un punto dado de la pantalla y la librería UI-Automator, que utiliza *MATE*, realiza los clicks en una coordenada específica. *MATE* escoge las coordenadas en las que se realizará en *click* como el centro del *widget* que con el que se va a interactuar. Espresso, en cambio, cuando realiza un *click* en un vista no utiliza coordenadas, sino el conocimiento del *widget* sobre el cual se quiere hacer *click*. Actualmente, *ETG* genera un caso de test que realiza un *click* en el *widget* informado en su input.

Esto trae el siguiente escenario problemático cuando *MATE* decide realizar un click, por ejemplo, en la vista root de la pantalla:

- Se calcula las coordenadas del centro del *widget*. Las mismas son el centro de la pantalla.
- La librería UI-Automator realiza el *click* en el centro de la pantalla.
- Si existe casualmente una vista, por ejemplo un botón, en el centro de la pantalla entonces la misma recibe el *click* y generá la consecuencia del mismo. Caso contrario el *click* no tiene efecto.
- *ETG* genera un caso de test que busca la vista root de la pantalla y realiza un click sobre ella. Sin importar si hubiera o no botón en el centro de la pantalla este click no genera ningún resultado.

El resultado de este escenario es que en *MATE* se pudo haber realizado un *click* manejado por una vista y que en el test case generado por *ETG* se realiza un *click* que no hace nada, lo cual claramente resulta en un test que no se comporta como el input lo especifica.

Para mitigar este problema se decidió realizar un refinamiento del *widget* seleccionado para realizar la acción en base a las coordenadas.

Cambios en MATE

No se realizaron cambios en *MATE*.

Cambios en ETG

Dentro del output de *MATE* se encuentran las coordenadas donde se realizó la acción y, también, todos los padres e hijos del *widget* seleccionado para recibir la acción junto a las coordenadas que ocupa cada uno de estos en pantalla. Con está información, más la coordenada donde se realizó la acción, se puede buscar la vista qué verdaderamente recibe la acción en pantalla. El Listing 2.8 muestra el refinamiento realizado que busca la vista mas cercana a una hoja que cumple que las coordenadas donde se realizó la acción caiga dentro de su área.

Listing 2.8: Refinamiento de widget receptor de acción

```
1   public Widget getReceiverOfClickInCoordinates(int x, int y){
2       for(Widget child: children){
3           Widget receiver = child.getReceiverOfClickInCoordinates(x, y);
4           if (receiver != null) return receiver;
5       }
6
7       if (receivesClickOnCoordinates(x, y)) return this;
8       else return null;
9   }
10
11
12  private boolean receivesClickOnCoordinates(int x, int y){
13      return isInRange(x, x1, x2) && isInRange(y, y1, y2);
14  }
15
16
17  private boolean isInRange(int c, int c1, int c2){
18      return (c1 <= c && c <= c2) || (c2 <= c && c <= c1);
19  }
```

Este refinamiento es solo realizado cuando la acción no se trata de un *swipe*, ya que en ese caso el receptor del gesto es siempre la vista root.

2.2.6. Ambigüedades

En Android es muy común definir vistas y luego reutilizarlas en distintos lugares. Esto permite que se pueda introducir una vista dentro de otra repetidas veces con la misma o distinta información. El caso más común donde se puede encontrar una ambiguedad es al realizar un *click* en un ítem de una *RecyclerView* con items repetidos o similares(ver Sección 2.3). Esto es un problema porque al generar un *ViewMatcher* de Espresso el mismo puede no ser suficientemente preciso y dar un resultado ambiguo, creando fallas en las acciones del caso de test.

RecyclerViewActions[12] se trata de un conjunto de métodos que permiten interactuar con las *RecyclerView* en Espresso, pero tienen el problema de que requiere información de la posición de los items dentro de la lista; lo cual no es posible conseguir, al menos actualmente, desde un explorador como *MATE*.

Para mejorar la precisión y disminuir estas ambigüedades se agregó información al *ViewMatcher* de Espresso tanto de los padres, como de los hijos de la vista que se quiere describir. De esta forma, se disminuye la aparición de ambiguedades debido a estos motivos.

Cambios en MATE

En el output generado se agregaron los padres del *widget* receptor de cada acción. La información de los hijos ya se encontraba presente.

Cambios en ETG

Luego de refinar el receptor de la acción como se vió en la Subsección 2.2.5 y de armar la sentencia para elegir la vista, se agrega información de padres e hijos del *widget* utilizando los siguientes matchers que ofrece Espresso.

- *hasDescendant*: Método de Espresso que recibe un *ViewMatcher* como parámetro que describe una vista. Por ejemplo, el Listing 2.9 muestra la creación de un *ViewMatcher* que busca una vista con id *item* y que tiene algún descendiente con id *description*.

Listing 2.9: hasDescendant matcher

```
1 onView(allOf(withId(R.id.item), hasDescendant(withId(R.id.description))));
```

- *isDescendantOfA*: Método de Espresso que recibe un *ViewMatcher* como parámetro que describe una vista. Por ejemplo, el Listing 2.10 muestra la creación de un *ViewMatcher* que busca una vista con id *item* y que tiene algún padre con id *list*.

Listing 2.10: isDescendantOfA matcher

```
1 onView(allOf(withId(R.id.item), isDescendantOfA(withId(R.id.list))));
```

Para cada hijo de la vista con la que se quiere interactuar se utiliza el matcher *hasDescendant*, utilizando para describirlo su id, content description y texto, si los tuviera.

Se utiliza el matcher *isDescendantOfA* para describir al padre de la vista que se quiere matchear, utilizando para describirlo su id, content description, texto y recursivamente la información de su parent, si lo tuviera.

Se puede ver un ejemplo completo en el Listing 2.11, donde se especifica un matcher que busca una vista que cumpla:

- Tiene id *see_more_view*.
- Tiene algún descendiente con id *category* cuyo texto es *Popular*.
- Tiene algún parent cuyo id es *container* el cual tiene algún parent cuyo id es *mainView*.

Listing 2.11: ejemplo de matcher completo

```
onView(
    allOf(
        withId(R.id.see_more_view),
        hasDescendant(
            allOf(
                withId(R.id.category),
                withText(equalToIgnoringCase("Popular"))
            )
        ),
        isDescendantOfA(
            withId(R.id.container),
            isDescendantOfA(
                withId(R.id.mainView)
            )
        )
    )
);
```

2.3. Casos patólogicos

Dentro del universo de las listas implementadas con *Recyclerview* existen 2 familias de vistas donde la herramienta encuentra una limitación para desambiguar acciones actualmente:

- Existe al menos 1 ítem repetido en la lista.
- Cada ítem de la lista cumple que existe un *widget* cuya jerarquía no aporta información diferencial respecto a otros ítems

2.3.1. Elementos repetidos

Esta familia es las mas simple aunque menos frecuente de encontrar. Se trata de aquéllas listas que poseen por lo menos 2 ítems exactamente iguales con el mismo contenido. En este caso es imposible desambiguar la situación para la herramienta dado que no se tiene información sobre la posición del item en la lista. En la Figura 2.7 se observa un ejemplo donde la vistas que contienen al item **b** son exactamente iguales, y son solo diferenciables por su posición en la lista.

2.3.2. Existencia de widget con jerarquía ambigua

Esta familia de casos es la más frecuente. Ocurre cuando las vistas de los ítems de la lista son diferentes entre si al instanciarse, pero contienen un elemento cuyos hijos, si los tiene, y padres no permiten generar un matcher no ambiguo; pues todos los elementos de la lista machean con la misma condición.

Un claro ejemplo donde se puede observar esta situación es en una vista como muestra el Listing 2.12. La vista descripta por este XML puede verse en la Figura 2.8. En este caso tenemos un típico ítem de una lista con una imagen a la izquierda y un texto a la derecha.

Listing 2.12: Layout de ejemplo

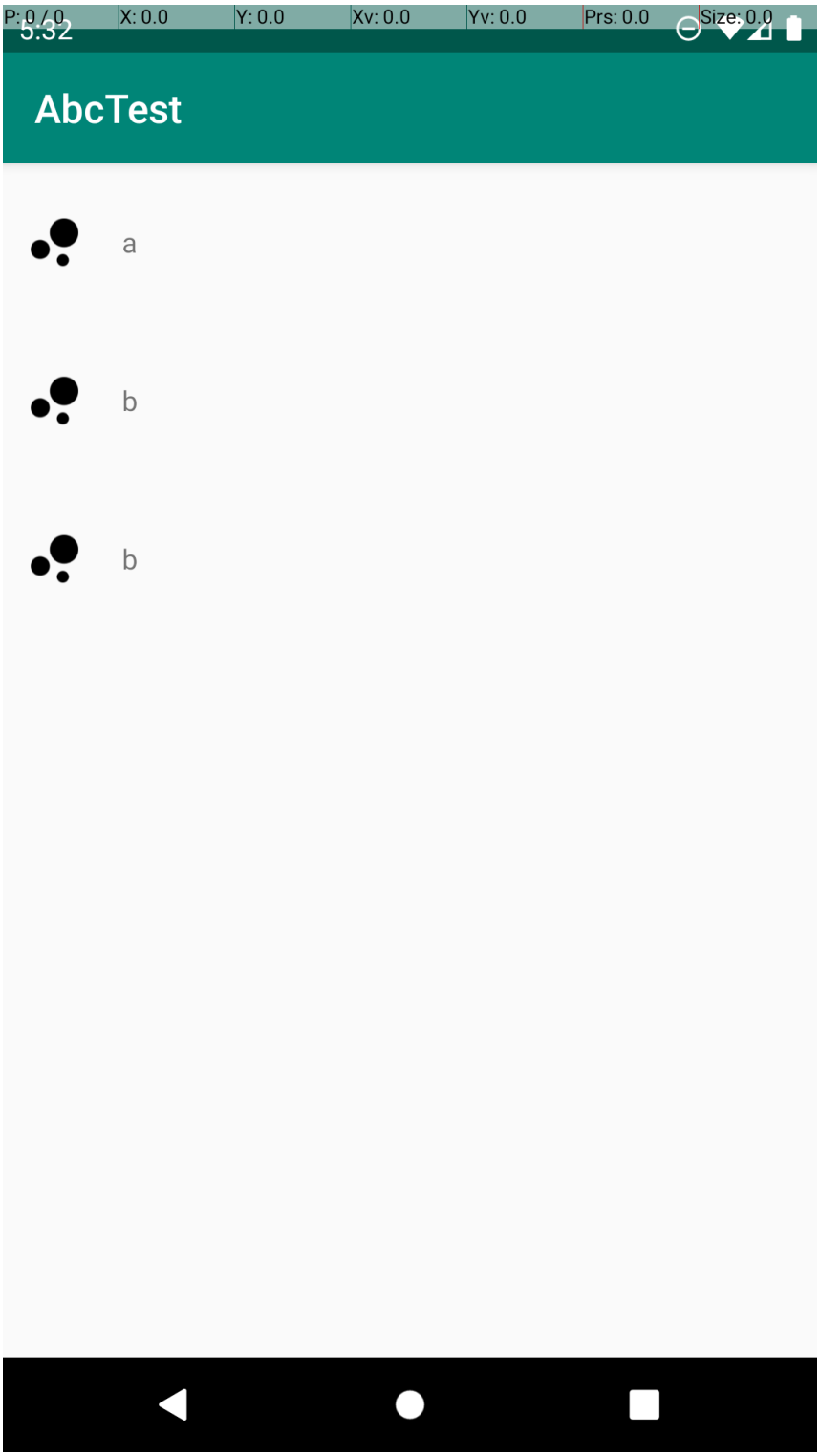

Fig. 2.7: Lista con el item **b** repetido

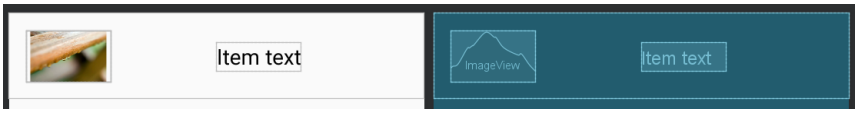

Fig. 2.8: Vista de ejemplo

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="80dp">

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        tools:text ="Item text"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@id/image"
        app:layout_constraintTop_toTopOf="@id/image"
        app:layout_constraintBottom_toBottomOf="@id/image" />

    <ImageView
        android:id="@+id/image"
        android:layout_width="80dp"
        android:layout_height="0dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        tools:srcCompat="@tools:sample/backgrounds/scenic" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Si la herramienta generadora de acciones realiza un *click* sobre el *TextView* puede ser posible realizar un *ViewMatcher* suficientemente preciso dado el texto distintivo que puede tener este *widget*. Pero si la herramienta realiza una acción sobre el *ImageView* entonces no es posible realizarlo. Esto sucede porque este *widget* no tiene hijos que aporten información distintiva, en si mismo no aporta información distintiva y sus ancestros tampoco pueden aportar información distintiva; todos los items de la lista cumplirían con cualquier matcher que se pueda generar mirando solo el imageView, sus childs y sus ancestros; dado el que el unico widget distintivo de la lista es el *TextView*, pero su información no se encuentra presente en la información suministrada actualmente a la herramienta.

Este caso hoy en día es muy frecuente debido al widget *ConstraintLayout* que permite crear vistas complejas con una layout de altura 1, consiguiendo así una jerarquía plana.

3. EVALUACIÓN

En cada aplicación, se ejecutó *MATE* para generar un output que luego se intentaría reproducir con un caso de test de Espresso generado con *ETG*. En todos los casos se utilizó el algoritmo de exploración aleatoria con la excepción de la aplicación Payment Processing S.A., más adelante se explicará el motivo.

- El algoritmo utilizado es *RandomWalkActivityCoverage*
- Para realizar cada acción, se elige el *widget* con el cúal interactuar de manera aleatoria.
- Se realizan 50 acciones en la exploración.
- Una vez finalizada la exploración de 50 acciones se procede a realizar una nueva *mejorando* la exploración previa.
- La *mejora* de la ultima exploración es aleatoria, por lo que no necesariamente es una mejora propiamente dicho. Se elige un numero aleatorio n y se guardan las primeras n acciones de la exploración previa.
- La nueva exploración procederá a realizar los primeros n pasos de la misma forma que la anterior, y los últimos serán elegidos aleatoriamente.
- Una vez alcanzado un timeout de 20 minutos la exploración finaliza con la ultima exploración completa.

Dado que el output de *MATE* luego de las modificaciones en esta tesis es compatible con la versión inicial de *ETG*, fue posible correr la herramienta antes y después de los cambios con el mismo input, para poder corroborar si existió alguna mejora real en la calidad de los casos generados.

3.1. Evaluación de ABC-APP

Con esta aplicación se buscaron falencias en:

- **Swipe**: El problema se encuentra detallado en la Subsección 2.2.1
- **Listas**: No se encontraron grandes problemas en las listas de esta aplicación, debido a que la misma no tiene repetidos. Si se encontraron problemas al realizar un *click* en ítems que no se encontraban suficientemente visibles. Dicho problema se encuentra detallado en la Subsección 2.2.2

	Antes	Despues
Acciones exitosas	20 %	96 %

En el test generado por la versión inicial de *ETG* fallaron 40 acciones sobre un total de 50; mientras que el test generado por la versión final solo fallaron 2 acciones.

3.1.1. Análisis de errores

En el Listing 3.1 se observa las 2 acciones que fallaron, las cuales se corresponden con:

- Realizar un *click* en el ítem de la lista con la letra *l*
- Realizar un *click* en el botón con texto *OK!*.

La acción previa a la primera falla se trata de un *swipe* el cual mueve la pantalla pero no lo suficiente para que el item con la letra *l* quede visible. Esto causa que se produzca un error al intentar matchear con dicho ítem, pues el mismo no se encuentra en pantalla. En la Figura 3.1 y Figura 3.2 se evidencia el resultado del *swipe*. El segundo fallo se produce como consecuencia del primero, dado que se quiere presionar el botón *OK!* de un dialogo que no fue mostrado en pantalla.

Listing 3.1: Sentencias fallidas en test de ABC-APP

```
1  ViewInteraction android_widget_TextView10 = onView(allOf(withId(R.id.text),
       withText(equalToIgnoringCase("l")), isDescendantOfA(allOf(withId(R.id.item),
       isDescendantOfA(withId(R.id.list))))));
2
3  try {
4      android_widget_TextView10.perform(getClickAction());
5  } catch (Exception e) {
6      System.out.println(buildPerformExceptionMessage(e, 24));
7  }
8
9  ViewInteraction android_widget_Button10 = onView(allOf(withId(R.id.button),
       withText(equalToIgnoringCase("OK!")), isDisplayed(),
       isDescendantOfA(isDescendantOfA(allOf(withId(R.id.custom),
       isDescendantOfA(allOf(withId(R.id.customPanel),
       isDescendantOfA(withId(R.id.parentPanel)))))))));
10
11 try {
12     android_widget_Button10.perform(getClickAction());
13 } catch (Exception e) {
14     System.out.println(buildPerformExceptionMessage(e, 25));
15 }
```

Los errores suceden debido a que el *swipe* no resulta 100 % exácto con el que hace *MATE* y debido a esto los 6 *swipes* previos a intentar matchear la vista con item *l* acumulan un error que se nota en en este momento puntual dejando la *l* fuera de pantalla. Si bien la mejora respecto a la versión previa de *ETG* es realmente significativa, queda como trabajo futuro encontrar el motivo de este error. Se estima qué la razón de este error puede ser debido a qué la coordenada (0,0) para *MATE* es el angúlo superior izquierdo de la pantalla, teniendo en cuenta la *status bar* del sistema operativo, mientras que para etg dicha coordenada comienza en el angúlo superior izquierdo pero debajo de la *status bar*, lo que genera este error. Se entiende que es un error en la traducción a coordenadas de *MATE*.

3.2. Evaluación de Movies & Tv!

Con esta aplicación se buscaron falencias principalmente en listas. Se trata de una aplicación con múltiples listas cuyos ítems son, a su vez, listas y los ítems finales que se

Fig. 3.1: Antes del swipe

Fig. 3.2: Despues del swipe

muestran tienen una apariencia mas cercana a lo que podría ser una aplicación del mundo real. Las vistas de esta aplicación tienen las siguientes particularidades:

- Hay 2 pantallas que, a nivel vista, son iguales y contienen una lista vertical donde cada ítem contiene una lista horizontal. Una contiene información sobre distintos grupos de películas mientras que la otra muestra información sobre distintos grupos de series.
- Todas las pantallas utilizan las mismas vistas con los mismos *id*, excepto la del detalle de una pelicula o serie.
- Cada ítem de las listas horizontales contiene un *TextView*, una *RatingBar* y un *ImageView*. De cada una de estas vistas solo el texto es único por cada película o serie. Esto trae el caso patológico descrito en la Sección 2.3 de que si se hace el *click* en la imagen entonces no resulte posible escribir un matcher correcto, pues todas las imágenes en pantalla contienen el mismo *id*, no tiene vistas hijo y sus padres son los mismos.

Esta aplicación permitió encontrar los problemas detallados en la Subsección 2.2.4, Subsección 2.2.5, Subsección 2.2.6 y la Sección 2.3.

	Antes	Despues
Acciones exitosas	66 %	78 %

En el test generado por la versión inicial de *ETG* fallaron 17 acciones sobre un total de 50; mientras que el test generado por la versión final fallaron 11 acciones. Si bien existe una mejora, la misma no es realmente significativa debido a la gran cantidad de casos patólogicos que presenta esta aplicación.

3.2.1. Análisis de errores

Los 11 fallos que surgieron en la ejecución de test corresponden al caso patólogico ya mencionado: Se quiere realizar un *click* eligiendo una imagen, pero no se puede distinguir

cual pues todas los *ImageView* de las listas cumple la condición enunciada. Se puede ver el caso en el Listing 3.2

Listing 3.2: Sentencia fallida por ambigüedad en Movies & Tv!

```
1 ViewInteraction android_widget_ImageView = onView(allOf(withId(R.id.imageView),
        isDescendantOfA(isDescendantOfA(allOf(withId(R.id.cardViewItemList),
        isDescendantOfA(allOf(withId(R.id.mediaList),
        isDescendantOfA(isDescendantOfA(withId(R.id.fragmentContainer)))))))));
2
3 try {
4     android_widget_ImageView.perform(getClickAction());
5 } catch (Exception e) {
6     System.out.println(buildPerformExceptionMessage(e, 4));
7 }
```

Generalmente luego de una acción fallida puede venir una nueva falla a consecuencia de la anterior, sin embargo en este caso eso no sucedió debido a la naturaleza de la aplicación, ya qué al ir al detalle de una película solo se puede realizar *swipes* o volver atrás, pero estas acciones también son validas y no fallarán en la pantalla principal.

Este tipo de aplicación requiere que los tests sean construidos con *RecyclerViewActions* para ser efectivos. Lamentablemente, como ya se mencionó, no es posible escribir tests con esta librería sin conocimiento sobre la totalidad de los ítems que se van a mostrar en cada lista.

3.3. Evaluación de Payment Processing S.A.

Se trata de una aplicación industrial de tamaño grande que permite el cobro con tarjetas de crédito y débito. La aplicación cuenta con secciones como:

- Login con usuario y contraseña.
- Recupero de contraseña.
- Pantalla principal con posibilidad de ingreso de montos a cobrar y lista de productos a vender.
- Pantalla con resumen de un carrito de la venta.
- Catálogo de productos y categorías.
- Pantalla de ajustes con visualización del perfil del usuario y configuraciones varias de la cuenta.
- Historial de ventas realizadas la posibilidad de filtrar la lista y ver el detalle de cada una.
- Historial de depósitos de dinero pendientes y pagados.
- Catálogo de clientes con la posibilidad de edición y visualización de cada uno de ellos.
- Sección de ayuda.

3.3.1. Adaptación

Para poder probar esta aplicación se realizaron cambios sobre la misma, ya que no se encuentra en condiciones de correr tests de interfaz en un 100 % de sus pantallas.

- Mocks: Todos los request de la aplicación debieron ser mockeados para garantizar un comportamiento consistente para cada corrida de los tests. Mediante un interceptor de requests HTTP se logró mockear los 21 requests distintos que la aplicación realiza en las pantallas habilitadas.
- Existe una sección de la aplicación que interactúa con dongles bluetooth. Dado que los tests se ejecutan en un emulador y no se dispone actualmente de la posibilidad de mockear estos dispositivos se decidió ocultar esta sección.
- Se removieron las acciones de la app que envían al usuario a una aplicación diferente, como ser un navegador o mail. Esto se realizó debido a que *MATE* finaliza la exploración si detecta que se abandonó la aplicación que se estaba explorando.
- Se eliminaron los hints de los *EditText* de la aplicación. El motivo de este cambio se debe a una limitación actual del sistema de accesibilidad de android, el cual utiliza *MATE* para explorar la aplicación. El objeto *AccessibilityNodeInfo* no distingue entre el texto y el hint de un *EditText*:
  - Si el *EditText* contiene texto informa el texto correcto.
  - Si el *EditText* no contiene texto ni hint informa un texto vacío.
  - Si el *EditText* no contiene texto pero tiene hint informa el hint como texto.

En versiones de Android inferiores a 8.0(Oreo) no existe forma de conseguir el hint mediante el sistema de accesibilidad. En dicha versión del sistema operativo se agregó el método *getHintText*[13] a *AccessibilityNodeInfo*, pero cuando el campo de texto está vacío sigue dando el hint como texto, motivo por el cual resulta imposible distinguir cuando lo que se está leyendo es el texto o el hint. Por ejemplo si el usuario escribe como texto el hint no es posible distinguirlo. Esto trae el problema de que no se pueden generar matchers debido a que *MATE* informara que se quiere accionar en una vista con un texto que la misma no posee. Este problema se trata de una limitación de *MATE* y no de *ETG*.

Para la evaluación de esta aplicación se realizaron 2 exploraciones diferentes por utilizando *MATE*.

3.3.2. Exploración aleatoria

Esta exploración de la aplicación se realizó con el algoritmo aleatorio de *MATE* sin modificaciones, al igual que en las demás aplicaciones. El test obtenido con esta exploración cubrió un total de 3 pantallas: **login, recupero de contraseña y pantalla principal(teclado e inventario)**.

	Antes	Despues
Acciones exitosas	20 %	52 %

En los tests generados por la versión inicial de *ETG* fallaron 40 acciones mientras que en la versión final fallaron 24 acciones, ambos casos sobre el total de 50 acciones.

3.3.3. Exploración guiada

Debido a la naturaleza aleatoria de la exploración realizada por *MATE* no es simple encontrar una ejecución del mismo que explore una cantidad importante de pantallas de la aplicación. Es por esto que se decidió realizar una exploración adicional en *MATE* guiada por la mano humana, para lograr una caso de exploración que recorra toda la aplicación:

- Los primeros 6 pasos de la exploración fueron elegidos intencionalmente para explorar la pantalla de recupero de contraseña y lograr un login exitoso.
- En pantallas donde realizar un swipe no aporta ninguna información relevantes se prohibió la elección de dicha acción.
- Luego de un total de 3 o 6 acciones aleatorias realizadas por *MATE* en cada pantalla se eligen las acciones necesarias para llevar la exploración a otra pantalla.

De esta forma se logró una exploración más completa de la aplicación pasando por 12 pantallas, a diferencia del primer caso aleatorio logrado que solo pasaba por 3.

	Antes	Despues
Acciones exitosas	62 %	82 %

En los tests generados por la versión inicial de *ETG* fallaron 19 acciones mientras que en la versión final fallaron 9 acciones, ambos casos sobre el total de 50 acciones.

3.3.4. Análisis de errores

Se pueden distinguir en ambos casos entre 2 grupos de fallos:

- Matchers ambiguos: Nuevamente se hace presente el caso de falla más frecuente, la ambigüedad al decidir por un matcher (Subsección 2.3.2). Sucede en la vista principal de la aplicación que contiene una lista de ítems y una tab inferior para elegir entre la lista de ítems y el teclado para ingresar montos.
- Consecuencia de una falla previa: Al fallar una acción que genera un cambio en la vista es de esperarse que las acciones siguientes fallen, pues dichos cambios en la vista no fueron fueron realizados.

En la siguiente tabla se puede ver el total de fallos por cada familia de errores.

	Ambigüedades	Consecuencias	Total
Aleatoria	17	7	24
Guiada	2	7	9

En el caso de la exploración guiada cabe destacar que una consecuencia de los fallos por ambigüedad fue que 2 pantallas que fueron exploradas por *MATE* no hayan sido exploradas en el test.

4. CONCLUSIONES

El trabajo realizado en esta tesis contribuye a mejorar la calidad de los tests generados por *ETG* aumentando la fidelidad entre el input y el output de la herramienta. Es por ello, que se trata de un paso adelante hacia una potencial integración de la misma con el ciclo de desarrollo de una aplicación productiva. Sin embargo, se observa aún una falta madurez para lograrlo ya que los tests generados aún no son claros y las acciones que se realizan en los mismos pueden resultar muy verbosas, lo cual dificulta entender lo que se está realizando. Por otro lado, queda también por resolver los casos patológicos y el correcto manejo de listas, que son muy comunes en las aplicaciones de hoy en día.

A futuro, y pensando en el uso industrial, veo muy favorable la posibilidad de integrarse con otra herramienta de exploración diferente a *MATE* para así dejar el armado de los tests de interfaz de una aplicación a un equipo de QA-Automation, sin perfil de desarrollador, que podría generar toda una suite de tests para correr regresiones como parte de un pipeline de integración continua. Adicionalmente, una herramienta de este estilo permitiría agregar información contextual al input de *ETG*, como por ejemplo posiciones de ítems en una lista, para mejorar así la interacción con ciertas vistas.

Pensando en esto, una excelente y necesaria feature para *ETG* sería que genere aserciones en los tests que crea para asegurar no solo que una exploración pueda seguir realizándose, sino que también la misma tenga siempre el mismo resultado. Esta clase de tests podría ahorrar una gran cantidad de horas hombre, especialmente en el testeo manual de aplicaciones, lo cual se traduce no solo en ahorro de tiempo, sino también en ahorro de dinero, ya que los equipos de QA-Manual ya no deberían realizar largas regresiones a mano, sino que solo probar los casos que fallan en el pipeline de la aplicación y las nuevas funcionalidades en busca de errores.

4.1. Trabajo futuro

- La gran mayoría de las aplicaciones de hoy en día tienen al menos una pantalla con una lista por lo que es importante lograr un correcto funcionamiento en estos casos. Sin embargo, pareciera ser una limitación hoy en día interactuar con elementos de una lista implementada con *RecyclerView* sin tener conocimiento sobre los items que se muestran.
- Investigar si es posible resolver los casos patológicos provocados por vistas ambiguas analizados en la Sección 2.3.
- Refinar la heurística definida en la Subsección 2.2.6, ya que la misma resulta ser muy verbosa ensuciando la legibilidad del código. Debe acotarse o refactorizarse para ser más legible.
- Se debe definir la coordenada (0,0) del input y adaptar las herramientas a tal estandar. Particularmente, esto afecta a los swipes debido a que el origen de las coordenadas para *MATE* y *ETG* es diferente.
- Mejorar la calidad y facilidad de lectura de los casos de test generados. Esto implica un refactor de los templates y forma de generación de código utilizada.

- Refactor general del código de *ETG*. El código es muy difícil de entender, utiliza condicionales para manejar casos borde en distintos lugares cambiando el flujo de generación del test y no son claras las responsabilidad de cada componente del mismo. Por ejemplo, la construcción de los matchers se encuentra hardcodeada de manera explícita en cada caso sin reutilizar código para construir correctamente los mismos. Esto genera que los cambios que se realizan deban hacerse a la defensiva sin tener la certeza de que la aplicación siga funcionando correctamente al cambiar una pequeña parte de la misma.

Bibliografía

[1] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In ESEM, pages 15-24. IEEE Computer Society, 2013.

[2] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for Android: Are we there yet? (E). In International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, pages 429-440, 2015.

[3] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An empirical study of Android test generation tools in industrial cases. In ASE. ACM, 2018.

[4] UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey.html`, 2019.

[5] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for Android applications. In ISSTA. ACM, 2016.

[6] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. Automated accessibility testing of mobile apps. In ICST. IEEE Computer Society, 2018.

[7] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In ESEC/SIGSOFT FSE, pages 224-234. ACM, 2013.

[8] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of Android apps. In ESEC/SIGSOFT FSE. ACM, 2017.

[9] Stas Negara, Naeem Esfahani and Raymond P. L. Buse. Practical Android Test Recording with Espresso Test Recorder. IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSESEIP), 2019.

[10] Espresso - *Android developers.* `https://developer.android.com/training/testing/espresso`

[11] UI Automator - *Android developers.* `https://developer.android.com/training/testing/ui-automator`

[12] RecyclerViewActions - *Android developers.* `https://developer.android.com/reference/androidx/test/espresso/contrib/RecyclerViewActions`

[13] AccessibilityNodeInfo- *Android developers.* `https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo#getHintText()`