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

Generación aleatoria de casos de test Espresso para Android

Tesis presentada para optar por el título de
Doctor de la Universidad de Buenos Aires
en el área Ciencias de la Computación

Iván Arcuschin Moreno

Director de tesis: Dr. Juan Pablo Galeotti
Consejero de estudios: Dr. Hernán Melgratti
Lugar de trabajo: Laboratorio de Fundamentos y Herramientas para la Ingeniería de Software (LaFHIS), Departamento de Computación (DC) e Instituto UBA-CONICET de Ciencias de la Computación (ICC), Facultad de Ciencias Exactas y Naturales (FCEyN), Universidad de Buenos Aires (UBA).

Ciudad Autónoma de Buenos Aires, Argentina, 2023

Generación aleatoria de casos de test Espresso para Android

Resumen

El *testing* es una parte integral del proceso de desarrollo de aplicaciones Android: al correr casos de *test* regularmente en sus aplicaciones, los desarrolladores pueden verificar el correcto comportamiento y la usabilidad antes de poner las aplicaciones a disposición del público. Espresso es un *framework* de *testing* que permite a los desarrolladores escribir casos de *test* de interfaz de usuario (UI) Android concisos, confiables y legibles, y es el único *framework* de *testing* de UI con una amplia adopción entre los desarrolladores de aplicaciones. Se han propuesto varias herramientas de generación automática de *tests* para ayudar a los desarrolladores en la tarea de *testing*. Sin embargo, muchas de estas herramientas no producen casos de *test* ejecutables, solo informan errores, y de las que generan *tests*, sólo algunas admiten el formato Espresso.

Esta tesis se centra en mejorar la generación de casos de *test* Espresso para aplicaciones Android. Comenzamos realizando un estudio empírico que compara la efectividad de distintos algoritmos evolutivos y muestra que dichos algoritmos no son adecuados para generar casos de *test* Android, siendo muchas veces superados por algoritmos puramente aleatorios. A continuación, analizamos los desafíos de generar casos de *test* en formato Espresso, utilizando un enfoque basado en traducción que aprovecha la salida de herramientas de *testing* automático existentes. Encontramos que uno de los principales desafíos es la falta de propiedades únicas para identificar de manera inequívoca *widgets* específicos en la UI. Esto se agrava debido a que muchas herramientas utilizan el *Servicio de Accesibilidad* de Android, que puede devolver información inconsistente.

Finalmente, esta tesis presenta una técnica para generar casos de *test* Espresso que son ampliamente más confiables que los generados utilizando el enfoque basado en traducción según una evaluación experimental en 1.035 apps Android. Esta técnica incluye algoritmos novedosos para generar *View Matchers* de Espresso que seleccionan de manera concisa *widgets* de Android, y para crear *View Assertions* de Espresso que sirven para *tests* de regresión. Utiliza además el *framework* Espresso directamente para obtener información e interactuar con la aplicación bajo *test*.

Palabras claves: Android, Espresso, generación de casos de *test*, oráculos de *test*, algoritmos basados en búsqueda.

Random Espresso Test Case Generation for Android

Abstract

Testing is an integral part of the Android application development process: by running regularly testing their apps, developers can verify correct behavior and usability before making the applications available to the public. Espresso is a testing framework that allows developers to write concise, reliable, and readable Android user interface (UI) test cases and is the only UI testing framework with widespread adoption among application developers. Several automated test generation tools have been proposed to assist developers in the testing task. However, many of these tools only report errors and do not produce executable test cases, and of those that generate tests, only some support the Espresso format.

This thesis focuses on improving the generation of Espresso test cases for Android applications. We begin by conducting an empirical study comparing the effectiveness of different evolutionary algorithms and show that such algorithms are not suitable for generating Android test cases, often being surpassed by purely random algorithms. Next, we analyze the challenges of generating Espresso test cases, using a translation-based approach that leverages the output of existing automated testing tools. We find that one of the main challenges is the lack of unique properties to unequivocally identify specific *widgets* in the UI. This is exacerbated by the fact that many tools use the Android *Accessibility Service*, which can return inconsistent information.

Finally, this thesis presents a technique for generating Espresso test cases that are significantly more reliable than those generated using the translation-based approach, according to an experimental evaluation on 1,035 Android apps. This technique includes novel algorithms for generating Espresso *View Matchers* that concisely select Android *widgets* and for creating Espresso *View Assertions* used in regression tests. It also directly utilizes the Espresso framework to gather information and interact with the application under test.

Keywords: Android, Espresso, test generation, test oracles, search-based software testing.

Agradecimientos

Quisiera arrancar agradeciendo profundamente a mi director, **Juan Pablo Galeotti** (JP), por la infinita paciencia, dedicación y apoyo que me brindó durante todos estos años. No sólo me guió en el desarrollo de esta tesis, sino que también me ayudó a crecer como investigador y como persona.

Luego, quiero agradecer al **LaFHIS**, el Laboratorio de Fundamentos y Herramientas para la Ingeniería de Software, por haberme recibido con los brazos abierto y por haberme dado la oportunidad de estudiar y trabajar en un ambiente de investigación tan cálido y amigable. El LaFHIS fue mi segunda casa durante mi doctorado, y estoy muy agradecido por eso. En particular, quiero agradecer a los profesores y doctorandos del LaFHIS, todas personas excelentes con las que compartí muchas momentos: **Victor Braberman**, **Hernán Gagliardi**, **Diego Garverbetsky**, **Javier Godoy**, **Carlos Lopez Pombo**, **Daniela Marottoli**, **Hernán Melgratti**, **Leandro Nahabeddian**, **Christian Roldán**, **Alexis Soifer**, **Sebastián Uchitel**, **Florencia Zanollo**. Agradezco especialmente a mi compañero de escritorio, **Agustín Martinez Suñé**, por haber compartido conmigo tantas horas de laburo y charlas, en las que ponderamos la máxima ciencia y máxima verdad. Las risas no faltaron.

Por supuesto, el LaFHIS no existiría sin el **DC**, el Departamento de Computación de la Facultad de Ciencias Exactas y Naturales, y el **ICC**, el Instituto de Ciencias de la Computación. Quiero agradecer a todas las personas que componen el DC e ICC por generar un ambiente tan estimulante y enriquecedor en el que hacer investigación. Al mismo tiempo, quiero agradecer a la **UBA**, la Universidad de Buenos Aires, por haberme dado la oportunidad de estudiar en una universidad pública, gratuita y de calidad, y al **CONICET**, por haberme otorgado una beca doctoral que me permitió dedicarme a la investigación durante estos años.

Quiero agradecer también a mis estudiantes de licenciatura, que me ayudaron a crecer como docente y como persona: **Christian Ciccaroni**, **Gustavo Giráldez**, **Nicolas Walter**, **Gustavo Arribas**, y **Lisandro Di Meo**. Fue un placer dirigirlos en sus respectivas tesis de licenciatura.

A los jurados de esta tesis, **Alessandra Gorla**, **Renzo Degiovanni** y **Marcelo D'Amorim**, por el tiempo dedicado a evaluar este trabajo, y por sus valiosos comentarios y sugerencias que me ayudaron a mejorar esta tesis.

Por último, quiero agradecer a mis personas más queridas, que siempre me apoyaron incondicionalmente, acompañandome durante todo el recorrido y dándome fuerzas para seguir adelante. Sin duda, esta tesis no hubiera sido posible sin ellos: mi **mamá**, mi **papá**, mi **hermano**, y mi compañera, **Caro**.

A todos ellos, muchas gracias.

Índice general

1

Introducción

1.1. Motivación

El ***testing* de software** es un proceso integral y sistemático que se realiza para evaluar la calidad de un producto de software [1]. Este proceso consiste en ejecutar diferentes *casos de test*, también conocido como casos de prueba o *test cases* en inglés, con el fin de descubrir y reportar problemas. Cada **caso de *test*** es un documento que describe un escenario específico a ser probado en el software que se está analizando. Dicho documento proporciona instrucciones claras y detalladas sobre cómo ejecutar una prueba y qué resultados se esperan obtener. Mediante el proceso de *testing* podemos verificar si el software cumple con los requisitos especificados, detectar defectos o errores y asegurar que el sistema funcione correctamente (incluyendo a veces la usabilidad o seguridad del software).

A medida que el software adquiere una importancia cada vez mayor en nuestra vida diaria, también aumenta el uso de **dispositivos móviles** como teléfonos inteligentes (*smartphones*) y tabletas (*tablets*). Se estima que las tecnologías móviles son utilizadas actualmente por dos tercios de la población mundial [2]. Además, los usuarios móviles consumen universalmente más minutos digitales por persona, más del doble en la gran mayoría de países y regiones [3]. En este contexto, los *smartphones* se han convertido en la plataforma dominante para el consumo de tiempo móvil, en términos de minutos totales en cada mercado. Alrededor del 80% de todo el tiempo móvil [3] se dedica al uso de aplicaciones (comúnmente conocidas como "*apps*"): programas especialmente diseñados para ser ejecutados en dispositivos móviles. Al mes de Diciembre de 2023, existen alrededor de 2.5 millones de aplicaciones ANDROID disponibles en la tienda de aplicaciones Google Play Store [4].

Los desarrolladores de **aplicaciones móviles** pueden recurrir a diversas herramientas, *frameworks* y servicios para desarrollar y mejorar la calidad de sus *apps* [5]. Sin embargo, sigue siendo un hecho que los errores se cuelan en el software implementado, lo cual puede disminuir significativamente la reputación tanto de los desarrolladores como de las empresas. Existen numerosos estudios sobre cómo los desarrolladores realizan *testing* de software de propósito general [6]. Sin embargo, las particularidades de las aplicaciones móviles hacen que se diferencien de otros tipos de software como los programas de escritorio, páginas web, etc. [7, 8].

Por lo tanto, la llegada de las aplicaciones móviles ha traído consigo un nuevo ecosistema donde las herramientas de *testing* tradicionales no siempre son aplica-

bles [9–11]: se deben admitir interacciones de usuario complejas (e.g., *swipe*, *zoom-in*, etc.) [12]; las aplicaciones deben tener en cuenta dispositivos con recursos limitados (e.g., fuente de alimentación limitada, capacidad de procesamiento inferior) [13]; los desarrolladores deben tener en cuenta un número cada vez mayor de dispositivos y versiones de sistemas operativos [14]; las aplicaciones suelen seguir una estrategia de lanzamiento basada en el tiempo semanal o quincenal, lo que crea restricciones críticas en las tareas de *testing* [15]. Además, las pruebas manuales no son un enfoque rentable para garantizar la calidad del software y deben ser reemplazadas por técnicas automatizadas [16].

Diversos *frameworks* han sido creados en los últimos años para ayudar a los desarrolladores ANDROID a escribir casos de *test* y automatizar su ejecución. JUnit es un *framework* muy popular para *testing* unitario en JAVA. En proyectos de ANDROID, los casos de *test* escritos puramente en JUnit solo se pueden utilizar para analizar clases que no interactúan con la plataforma de ANDROID. Para realizar *testing* de los componentes de la interfaz gráfica (en inglés, *Graphical User Interface*, abreviado GUI o UI) de una aplicación (e.g., pantallas y *widgets*), los desarrolladores deben utilizar un *framework* basado en instrumentación. Este tipo de *frameworks* utilizan la Instrumentación de ANDROID [17] para inspeccionar e interactuar con las pantallas que se están navegando y requieren que los *tests* sean ejecutados en un emulador o dispositivo ANDROID. Algunos *frameworks* conocidos para realizar *testing* de UI ANDROID son Robotium [18], Appium [19], Calabash [20], MonkeyRunner [21], UIAutomator [22] y ESPRESSO [23]. De estos *frameworks*, Robotium y Calabash están obsoletos y ya no reciben mantenimiento. El *framework* Robolectric [24] requiere una mención especial ya que permite realizar *testing* unitario de componentes de la UI utilizando un entorno ANDROID simulado dentro de una JVM JAVA, sin necesidad de un emulador o dispositivo.

De los *frameworks* mencionados anteriormente, **Espresso** [23] es el único *framework* de *testing* de UI con una amplia adopción entre los desarrolladores de *apps* ANDROID [25]. Este *framework* permite escribir casos de *test* de UI concisos, confiables y legibles. Su popularidad está dada por su inclusión en el Kit de Desarrollo de Software (SDK) ANDROID, su capacidad para mitigar la inestabilidad (*flakiness*), y la simplicidad en la creación y mantenimiento de casos de *test*.

Aunque existen varios *frameworks* para automatizar la ejecución de casos de *test* en ANDROID, la creación de estos *tests* sigue siendo principalmente una tarea manual [25]. Esto implica que el *testing*, a pesar de ser efectivo para detección de fallas y defectos, todavía es una tarea costosa que requiere de mucho tiempo y esfuerzo, y propensa a errores [26, 27].

En los últimos años se han propuesto distintas herramientas de **generación automática de *tests*** para apps ANDROID, con el fin de ayudar a los desarrolladores en esta tarea [28–32]. Sin embargo, dichas herramientas tienen limitaciones que las hacen poco atractivas para los desarrolladores. En primer lugar, muchas de estas herramientas no producen casos de *test*, solo informan errores (en inglés, *crashes*). Aunque esto es ciertamente información valiosa, no apoya el objetivo de construir *tests* de regresión fuertes. En segundo lugar, incluso cuando las herramientas producen

tests, estos son a menudo poco confiables debido a la naturaleza volátil de los *widgets* de UI en ANDROID, lo que lleva a errores durante su re-ejecución. En tercer lugar, los *tests* generados no son fáciles de leer o mantener, ya que rara vez utilizan el elegante *framework* ESPRESSO. Finalmente, los *tests* generados no contienen *aserciones*, que son un prerequisito para hacer que los *tests* sean útiles durante el *testing* de regresión.

1.2. Objetivo y contribuciones de la tesis

La pregunta central que nos hacemos en esta tesis es:

¿Cómo podemos mejorar la generación automática de casos de *test* para aplicaciones Android?

La tesis comienza explorando las limitaciones y los factores de impacto de los enfoques existentes para realizar *testing* automático en aplicaciones ANDROID. Este análisis inicial destaca la falta de soporte para el formato ESPRESSO en las herramientas de generación de casos de *test* más avanzadas, que a menudo requieren ingeniería inversa de los casos de *test*. Esto motiva la necesidad de un enfoque directo y confiable para la generación de casos de *test* ESPRESSO. Además, el estudio inicial de esta tesis aporta evidencia empírica de que muchos de los algoritmos basados en búsqueda utilizados en el estado del arte para generar *tests* unitarios (e.g., algoritmos evolutivos como NSGA-II [33]) no son adecuados para generar casos de *test* en el contexto de ANDROID.

A continuación, la tesis presenta un estudio empírico evaluando la factibilidad de adaptar las herramientas existentes de generación de casos de *test* ANDROID para generar casos de *test* en formato ESPRESSO. Aunque los resultados de dicho estudio son prometedores, también revelan desafíos importantes que deben abordarse para que la generación de casos de *test* ESPRESSO sea posible.

En base a estos aprendizajes, la tesis presenta finalmente un método novedoso para generar casos de *test* ESPRESSO, utilizando directamente el *framework* ESPRESSO para interactuar con la aplicación bajo *test*. Este método nos permite generar *selectores* claros y concisos para los *widgets* de la UI. Además, a diferencia de los enfoques existentes, este método permite agregar *aserciones* a los casos de *test* generados, lo que los hace más útiles para el *testing* de regresión. La evaluación experimental de esta técnica en 1.035 apps ANDROID muestra mejoras significativas en la efectividad de la técnica propuesta con respecto a los enfoques existentes en la literatura.

1.3. Artículos publicados relevantes

Los resultados presentados en esta tesis fueron publicados en los siguientes artículos:

- I. Arcuschin, L. Di Meo, M. Auer, J. Galeotti, G. Fraser.
  Brewing Up Reliability: Espresso Test Generation for Android Apps

Aceptado para ICST 2024: International Conference on Software Testing, Verification and Validation.

- I. Arcuschin, C. Ciccaroni, J. Galeotti, J.M. Rojas.
  On the feasibility and challenges of synthesizing executable Espresso tests
  AST 2022: International Conference on Automation of Software Test.
  doi: 10.1145/3524481.3527234
- I. Arcuschin, J. Galeotti, D. Garbervetsky.
  An Empirical Study on How Sapienz Achieves Coverage and Crash Detection
  Journal of Software: Evolution and Process.
  doi: 10.1002/smr.2411
- I. Arcuschin, J. Galeotti, D. Garbervetsky.
  Algorithm or Representation? An empirical study on how SAPIENZ achieves coverage
  AST 2020: International Conference on Automation of Software Test.
  doi: 10.1145/3387903.3389307
- I. Arcuschin.
  Search-Based Test Generation for Android Apps
  Doctoral Symposium at ICSE 2020: International Conference on Software Engineering.
  doi: 10.1145/3377812.3381389

1.4. Estructura de la tesis

Este trabajo está organizado en 7 capítulos:

Capítulo 1. Introducción. Se presenta el contexto de la tesis, el problema a resolver y el objetivo de la misma.

Capítulo 2. Preliminares. Se presentan los conceptos y herramientas necesarios para comprender el trabajo realizado.

Capítulo 3. Estado del arte. Se presentan los trabajos relacionados con la tesis.

Capítulo 4. Evaluación algoritmos. Se presenta una evaluación empírica de varios algoritmos de búsqueda utilizados para generación de casos de *test* de *apps* Android.

Capítulo 5. Desafíos y oportunidades. Se presentan los desafíos y oportunidades en la generación de casos de *test* Espresso.

Capítulo 6. Técnica propuesta. Se presenta la técnica propuesta y su evaluación experimental para la generación de casos de *test* Espresso.

Capítulo 7. Conclusiones y trabajo futuro. Se presentan las conclusiones del trabajo realizado y se proponen líneas de trabajo futuro.

2

Preliminares

2.1. Testing de software

2.1.1. Errores en el software

Como mencionamos en el Capítulo 1, el *testing* de software es un proceso de evaluación de un sistema, programa o aplicación para detectar diferencias entre el comportamiento esperado y el implementado. Utilizaremos la siguiente terminología para referirnos a los distintos tipos de desviaciones que un software puede tener con respecto a su comportamiento esperado. Un **error** es una desviación no buscada ni intencional de lo que es correcto, esperado o verdadero. Un **defecto** es un error en el código del programa, específicamente uno que puede crear un infección (y conducir a una falla). Una **infección** es un error en el estado del programa, específicamente uno que puede llevar a una falla. Una **falla** es un error externamente visible en el comportamiento del programa. Luego, en la práctica, el objetivo del *testing* de software es detectar fallas en el software bajo análisis.

2.1.2. Caso de test

Un caso de *test* (a veces traducido literalmente como "caso de prueba") es un documento detallando un escenario específico que debe ser probado en el software bajo análisis. Este documento puede ser tanto una especificación de los pasos a seguir para probar una funcionalidad como una especificación de los requisitos que debe cumplir el software. Puede estar escrito en lenguaje natural o en un lenguaje de programación (e.g., Java), de forma que pueda ser ejecutado automáticamente. En cualquiera de sus variantes, los casos de *tests* suelen estar compuestos de los siguientes elementos:

- **Configuración**: una etapa de preparación (en inglés, *setup*) requerida antes de ejercitar el *test*. Por ejemplo, la creación de un objeto o la inicialización de una variable.
- **Ejercitación**: el código (o pasos) que ejercita la funcionalidad bajo *test*. Por ejemplo, la invocación de un método.
- **Chequeo**: código (o pasos) para verificar la respuesta del programa bajo análisis contra el resultado esperado (oráculo). Por ejemplo, la comparación de un valor

contra otro.

Es interesante detenerse en el concepto de **oráculo**, ya que es un elemento fundamental en el *testing* de software. Un oráculo es un mecanismo que permite determinar si el resultado de un *test* es correcto o no, es decir, si el software bajo análisis se comporta como se espera. Los oráculos pueden ser escritos manualmente por un desarrollador o pueden ser generados automáticamente a partir de una especificación formal del software.

Al mismo tiempo, los oráculos se pueden dividir en *explícitos* o *implícitos*. Los oráculos explícitos son aquellos que están basados en especificaciones o criterios predefinidos y claros. Estos oráculos se utilizan cuando se dispone de información detallada sobre cómo se debe comportar el software en diferentes situaciones. Por ejemplo, si se espera que una función aritmética devuelva el resultado correcto según una fórmula matemática conocida, el oráculo explícito sería simplemente aplicar esa fórmula para verificar si el resultado obtenido es el esperado. Los oráculos implícitos son menos obvios y se basan en información menos explícita o detallada sobre el comportamiento esperado del software. Estos oráculos se utilizan cuando no es posible o práctico establecer especificaciones detalladas para todas las situaciones posibles. En cambio, se basan en el conocimiento del dominio del problema, la experiencia del desarrollador y las expectativas generales sobre el comportamiento razonable del software. Algunos ejemplos de oráculo implícito son:

- Comparar la salida de un programa con la salida de una versión anterior del mismo programa (*testing* de regresión). Si el programa se modificó recientemente, es posible que no se disponga de una especificación formal de cómo debería cambiar la salida, pero se puede utilizar la versión anterior como oráculo para verificar si el comportamiento del programa es correcto.
- Verificar que el programa no produce *crashes*. Los *crashes* son situaciones dónde el software deja de funcionar de manera inesperada y abrupta, llevando al sistema operativo a cerrar dicho programa o terminarlo de forma automática. En este caso, el oráculo es simplemente la ausencia de *crashes*.

Los oráculos implícitos pueden ser valiosos cuando las especificaciones son incompletas o inexistentes, pero también pueden ser más subjetivos y propensos a errores si no se apoyan en un buen conocimiento del dominio y una experiencia sólida del desarrollador. En general, es importante utilizar una combinación de oráculos explícitos e implícitos para mejorar la efectividad del *testing* de software y garantizar que se detecten tanto errores obvios como sutiles en el software.

Por último, un ***test suite***, también conocido como conjunto de casos de *test*, es un grupo o colección organizada de casos de *test* relacionados. En otras palabras, un *test suite* agrupa varios casos de *test* con un propósito común, como probar una funcionalidad específica del software, realizar *tests* de regresión o validar un conjunto de requisitos.

2.1.3. Testing manual vs. automático

El ***testing* manual** se refiere a la ejecución de casos de *test* de forma manual, sin la ayuda de herramientas automatizadas. El desarrollador, o analista de calidad, lleva a cabo el procedimiento descripto en uno o varios *tests*, ejecutando manualmente las acciones necesarias para verificar el comportamiento del software. Algunas características del *testing* manual incluyen:

- Interacción humana: Los desarrolladores interactúan directamente con la aplicación, probando su funcionalidad, navegación y usabilidad.
- Exploración: Los desarrolladores pueden aplicar su experiencia y conocimiento para explorar diferentes escenarios y buscar posibles problemas o defectos.
- Flexibilidad: El *testing* manual permite a los desarrolladores adaptarse rápidamente a cambios en el software y realizar casos de *test* ad hoc según sea necesario.
- Casos de *test* no repetibles: Las casos de *test* manuales pueden ser difíciles de repetir exactamente de la misma manera, lo que puede afectar la consistencia y la precisión de los resultados.

El ***testing* automático** implica el uso de herramientas y scripts automatizados para ejecutar casos de *test*. Estos scripts se utilizan para simular acciones y verificar los resultados esperados. Algunos aspectos importantes del *testing* automático son:

- Eficiencia: Las casos de *test* automáticos se pueden ejecutar de manera más rápida y eficiente que los manuales.
- Repetibilidad: Los scripts automatizados se pueden ejecutar una y otra vez de manera consistente, lo que facilita la detección de defectos y problemas recurrentes.
- Regresión: El *testing* automático es especialmente útil para detectar regresiones, es decir, problemas que surgen después de realizar cambios en el software.
- Escalabilidad: Las casos de *test* automáticos son ideales para aplicaciones complejas o grandes conjuntos de datos, ya que pueden realizar tareas repetitivas y exhaustivas de manera más efectiva que los casos de *test* manuales.

La **integración continua**, en el contexto del *testing*, es una práctica de desarrollo de software en la que se ejecutan *tests* de forma automatizada y regularmente a medida que se agregan cambios al código fuente del proyecto. El objetivo principal de la integración continua es detectar rápidamente cualquier problema o error en el código y garantizar que todas las partes del software se integren correctamente.

2.1.4. Tipos y niveles de testing

Existen diversos tipos de *tests* que se pueden utilizar para ejercitar un software. Si miramos los distintos aspectos de un programa que se pueden analizar, encontramos las siguientes categorías (entre otras):

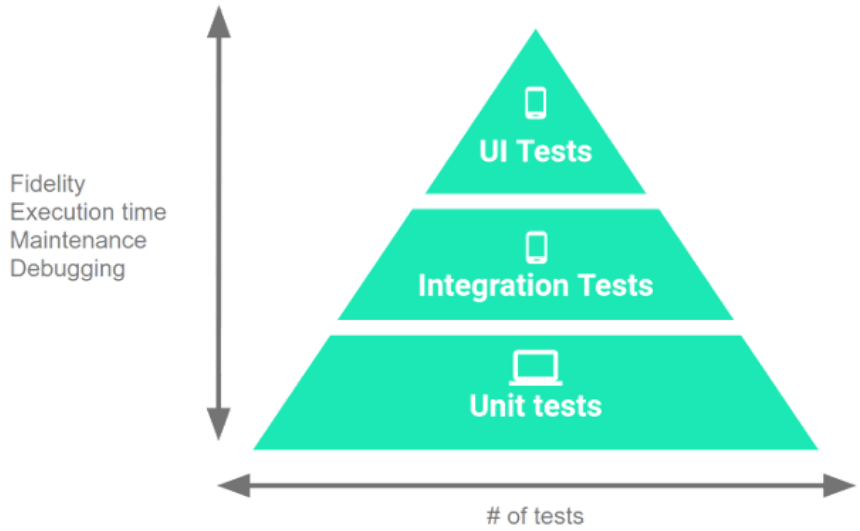


Figura 2.1: Pirámide de *tests* con las tres categorías principales que deben ser incluidas en el *test suite* de una aplicación ANDROID, como se ilustra en la documentación oficial de Prácticas recomendadas para desarrolladores ANDROID [34].

- *Tests* funcionales: ¿hace mi software lo que se supone que debe hacer?
- *Tests* de rendimiento (*performance*): ¿lo hace de manera rápida y eficiente?
- *Tests* de accesibilidad: ¿funciona bien con los servicios de accesibilidad?
- *Tests* de compatibilidad: ¿funciona bien en cada dispositivo y nivel de API?

Por otro lado, si nos concentramos en el nivel de detalle y fidelidad con el que se analiza el software, encontramos los siguientes categorías (entre otras):

- Casos de ***test* unitarios** o de unidad: se centran en la verificación de unidades de código individuales para asegurar que funcionen correctamente.
- Casos de ***test* de integración**: se realizan para evaluar la interacción entre diferentes componentes o módulos del software. El objetivo es asegurar que los componentes se integren adecuadamente y funcionen como se espera.
- Casos de ***test* de sistema**: se llevan a cabo en el sistema completo para verificar que cumple con los requisitos especificados. Esto incluye casos de *test* de funcionalidad, rendimiento, seguridad y usabilidad.

Estos categorías se pueden agrupar en niveles de una pirámide. La Figura 2.1 ilustra este concepto para el contexto de aplicaciones ANDROID. A medida que se "sube" en la pirámide, los *tests* se vuelven más lentos y frágiles, pero también más confiables y representativos de cómo los usuarios interactúan con la aplicación. Es decir, en los niveles más altos de la pirámide los *tests* se vuelven más caros de escribir y mantener, pero también más valiosos. La recomendación general en la práctica es que el *test suite* de un software incluya una combinación de *tests* de cada categoría de la pirámide. Particularmente, se recomienda que el *test suite* incluya 70% de *tests* unitarios, 20% de *tests* de integración y 10% de *tests* de sistema. Obviamente, esta

proporción es sólo una guía y debe adaptarse según los casos particulares de cada proyecto.

2.1.5. Criterios de adecuación

En el contexto de *testing*, los **criterios de adecuación** son un conjunto de reglas o directrices utilizadas para determinar si se ha realizado una cobertura suficiente y exhaustiva de las funcionalidades y características del software bajo análisis durante la ejecución de los *tests*. Estos criterios establecen métricas y condiciones que deben cumplirse para considerar que un conjunto de casos de *test* proporciona una cobertura adecuada. En otras palabras, nos permiten evaluar si un *test* suite es "suficientemente bueno". Entonces, un criterio de adecuación es un predicado que es verdadero o falso para un programa y *test* suite dado. Usualmente es expresado en forma de una regla de cobertura. Por ejemplo, "cada sentencia del programa debe ser ejecutada al menos una vez".

En definitiva, el objetivo de los criterios de adecuación es garantizar que los *tests* cubran una amplia gama de escenarios y situaciones, con el fin de identificar y detectar la mayor cantidad posible de errores o defectos en el software. Al utilizar los criterios de adecuación, los desarrolladores pueden determinar si se han ejercitado adecuadamente los diferentes aspectos del software.

A grandes rasgos, los criterios de adecuación se pueden clasificar de acuerdo a si contamos o no con el código fuente del software bajo análisis. Los criterios de adecuación que se evalúan sin el código fuente se denominan de **caja negra** (*black-box testing*), ya que se basan en la especificación del software y no en su implementación. Por el contrario, aquellos que se evalúan con el código fuente se denominan de **caja blanca** (*white-box testing*), ya que utilizan el conocimiento de la implementación del software. Ejemplos de criterios de adecuación de caja negra incluyen:

- **Partición de equivalencias** (*equivalence partitioning*): Este criterio se basa en la partición de los datos de entrada en clases de equivalencia, de forma que cada clase sea tratada de la misma manera por el software. Se verifica que cada clase de equivalencia sea probada al menos una vez.
- **Análisis de valores límite** (*boundary value analysis*): Este criterio se basa en los valores límite de las clases de equivalencia definidas en el criterio anterior. Se verifica que los valores límite sean probados al menos una vez.

Ejemplos de criterios de adecuación de caja blanca incluyen:

- **Cobertura de sentencias** (*statement coverage*): Este criterio busca cubrir todas las sentencias o líneas de código del software. Se verifica que cada sentencia haya sido ejecutada al menos una vez. La intuición de este criterio es que un defecto en una sentencia sólo puede ser detectado si la sentencia es ejecutada. Entonces, la métrica para cobertura de sentencias se calcula como "sentencias ejecutadas"dividido por "sentencias totales".
- **Cobertura de rama** (*branch coverage*): Este criterio se enfoca en cubrir todas las posibles decisiones o bifurcaciones que se encuentran en el código. Se busca

que cada rama del flujo de control del programa bajo análisis sea ejecutada al menos una vez.

- **Cobertura de condición** (*condition coverage*): Este criterio se centra en cubrir todas las condiciones lógicas y evaluaciones booleanas dentro del software. Se verifica que todas las combinaciones de resultados de las condiciones sean probadas al menos una vez.
- **Cobertura de camino** (*path coverage*): Este criterio busca cubrir todos los posibles caminos de ejecución del software. Se busca que cada camino, desde la entrada hasta la salida, sea ejecutado al menos una vez.

Independiente del tipo de criterio, la cobertura de un *test* es usualmente computada automáticamente mientras el programa es ejecutado. Este cómputo requiere la *instrumentación* del programa en cuestión. Luego de la ejecución del programa, una herramienta de cobertura analiza y resume los resultados.

2.1.6. Mutation testing

El **Mutation *Testing***, o *testing* de mutación, es un criterio de adecuación especial utilizado para evaluar la efectividad de un *test suite* en la detección de errores [35]. A diferencia de otros criterios de adecuación que se centran en evaluar la cobertura lograda por los *tests*, como la cobertura de sentencias o la cobertura de ramas, *mutation testing* se enfoca en evaluar la capacidad de los *tests* para detectar cambios o mutaciones artificiales realizados en el código fuente.

El proceso de *mutation testing* implica introducir modificaciones deliberadas en el código original, creando así versiones mutantes. Estas mutaciones pueden incluir cambios como la inversión de operadores, la eliminación o inserción de instrucciones, la alteración de constantes, entre otros. Cada mutación genera una versión alterada del código fuente. Una vez que se han creado las mutaciones, se ejecuta el *test suite* sobre cada mutante individualmente. Si un *test* falla al ejecutarse sobre un mutante, se considera que el *test* ha sido efectivo en detectar la mutación y se dice que la mutación ha sido "eliminada" (*killed*, en inglés). Por el contrario, si una mutación no es detectada por ningún *test*, se considera que el *test suite* ha fallado en detectar la mutación y la mutación "sobrevive".

Entonces, *mutation testing* evalúa la calidad del *test suite* identificando si existen mutaciones que no son detectadas por ningún *test*. Si una mutación sobrevive al *test suite*, eso indica una debilidad en los *tests*, ya que no ha logrado identificar una alteración en el código que potencialmente podría ser un error. El resultado del *mutation testing* se puede expresar en términos de la tasa de mutantes eliminados, que representa el porcentaje de mutaciones que fueron detectadas y "eliminadas" por los *tests*. Una alta tasa de mutantes eliminados indica un *test suite* más efectivo, ya que ha logrado detectar y reaccionar a los cambios en el código.

Notar que no todas las mutaciones pueden llegar a ser útiles para *mutation testing*. En particular, las mutaciones *equivalentes* son aquellas que no alteran el comportamiento observable del programa, por lo que no pueden ser detectadas por los

tests. Por ejemplo, la eliminación de una sentencia que no tiene efecto en el programa o la inversión de un operador en una condición lógica que no altera el resultado de la evaluación. Este tipo de mutaciones son problemáticas, ya que pueden generar falsos positivos en el *mutation testing*. Aunque sería ideal identificar y eliminar estas mutaciones, en la práctica es difícil distinguir las mutaciones equivalentes de las no equivalentes, y de hecho el problema es indecidible [36, 37].

En definitiva, el *mutation testing* es un enfoque riguroso y exigente, ya que implica la generación de múltiples mutaciones y la ejecución de los *tests* sobre cada mutante individualmente, lo que puede ser computacionalmente costoso. Sin embargo, proporciona una evaluación más completa y realista de la calidad de los *tests* en términos de su capacidad para detectar errores potenciales en el código. Es importante destacar que el *mutation testing* no reemplaza otros criterios de cobertura, como la cobertura de sentencias o ramas. En cambio, se utiliza en conjunto con estos criterios para obtener una evaluación más completa de la efectividad de los *tests* y mejorar la calidad del proceso de *testing*.

2.1.7. Generación automática de casos de test

La **generación automática de casos de *test*** es un enfoque en el que se utilizan herramientas y técnicas automatizadas para crear casos de *test* de manera automática, en lugar de crearlos manualmente. Consiste en desarrollar algoritmos y procesos que generen automáticamente casos de *test*, utilizando modelos, reglas predefinidas, heurísticas u otras técnicas de generación. El primer artículo de esta índole fue publicado por el Coronel Richard L. Sauder en el año 1962 [38], presentando una técnica para generar datos de *test* para programas escritos en el lenguaje de programación COBOL.

La generación automática de casos de *test* puede implicar diferentes niveles de automatización, desde la generación completa de los casos de *test* hasta la generación parcial o sugerencia de casos de *test* que luego son revisados y modificados por desarrolladores. Algunos de los beneficios de generar casos de *test* automáticamente son:

- Ahorro de tiempo y esfuerzo: La generación automática puede producir una gran cantidad de casos de *test* en poco tiempo, lo que acelera el proceso de *tests* y libera a los desarrolladores para que se centren en tareas más críticas.
- Cobertura exhaustiva: La generación automática de casos de *test* puede ayudar a lograr una cobertura exhaustiva del software al explorar diferentes combinaciones de entradas, escenarios y caminos de ejecución. Los algoritmos y enfoques automáticos pueden abordar situaciones que podrían pasarse por alto en el diseño manual de casos de *test*, lo que mejora la calidad y la eficacia de los *tests*.
- Detección temprana de defectos: Al cubrir una amplia gama de escenarios y combinaciones de datos, los casos de *test* generados automáticamente pueden revelar problemas y errores que podrían pasar desapercibidos en los *tests*

manuales, permitiendo detectar defectos tempranamente en el ciclo de desarrollo de software.

- Escalabilidad: La generación automática de casos de *test* es especialmente útil en proyectos grandes y complejos, donde la cantidad de casos de *test* requeridos puede ser considerable. Los enfoques automáticos permiten generar una gran cantidad de casos de *test* sin incurrir en la misma cantidad de tiempo y recursos necesarios para crearlos manualmente.

Es importante destacar que, si bien la generación automática de casos de *test* puede ser beneficiosa, no reemplaza completamente los *tests* manuales y la experiencia humana. Ambos enfoques tienen sus fortalezas y se complementan entre sí para lograr una cobertura de *tests* completa y confiable.

2.2. Aplicaciones Android

2.2.1. Sistema operativo

Android [39] es un sistema operativo móvil basado en Linux y desarrollado por Google. Fue lanzado inicialmente en 2008 y se ha convertido en uno de los sistemas operativos móviles más populares y utilizados en el mundo. A continuación mencionamos algunas de las características salientes de esta plataforma.

Arquitectura. Android se basa en una arquitectura de capas [40] (ilustrado en la Figura 2.2). En la parte inferior, se encuentra el kernel de Linux, que proporciona funciones de bajo nivel como la administración de memoria, la gestión de procesos y el control de dispositivos. Encima del kernel, se encuentra la capa de abstracción de hardware (*Hardware Abstraction Layer* o HAL, en inglés), que proporciona una interfaz de programación de aplicaciones (*Application Program Interface* o API, en inglés) para que el sistema operativo interactúe con el hardware del dispositivo. La capa de servicios nativos proporciona servicios adicionales, como la administración de la energía y las notificaciones. Finalmente, la capa de aplicaciones es donde se ejecutan las aplicaciones y se muestra la interfaz de usuario.

Interfaz de usuario. La interfaz de usuario (*User Interface* o UI, en inglés) de Android se basa en el paradigma de diseño Material Design, que utiliza elementos visuales, animaciones y efectos para proporcionar una experiencia de usuario atractiva y coherente [41]. Los elementos principales de la interfaz de usuario de Android incluyen la barra de estado, la barra de navegación y los elementos de la pantalla de inicio, como los íconos de aplicaciones y los *widgets*. Los usuarios interactúan con las aplicaciones a través de gestos táctiles y también pueden utilizar botones físicos en el dispositivo, como los botones de inicio y retroceso.

Aplicaciones. En Android, las aplicaciones son componentes clave. Los desarrolladores pueden crear aplicaciones utilizando el kit de desarrollo de software de

Android [42, 43] (*Software Development Kit* o SDK, en inglés) y distribuirlas a través de la tienda de aplicaciones oficial de Android, Google Play Store [44]. Esta tienda *online* es una plataforma donde los usuarios pueden descargar aplicaciones, juegos, libros, música y más. También existen otras tiendas de aplicaciones alternativas donde los desarrolladores pueden distribuir sus aplicaciones.

Personalización y fragmentación. Android ofrece una alta capacidad de personalización tanto para los fabricantes de dispositivos como para los usuarios. Los fabricantes de dispositivos pueden personalizar la interfaz de usuario, agregar aplicaciones preinstaladas y modificar el aspecto y la funcionalidad del sistema operativo para adaptarse a sus necesidades. Los usuarios pueden personalizar sus dispositivos mediante la instalación de aplicaciones de terceros, personalización de *widgets* y ajustes de configuración.

Sin embargo, la personalización y la amplia gama de dispositivos Android también han llevado a un desafío conocido como "fragmentación". Debido a las diferencias de hardware [45] y las versiones del sistema operativo [46], las aplicaciones pueden comportarse de manera diferente en diferentes dispositivos. Los desarrolladores deben tener en cuenta esta fragmentación y probar sus aplicaciones en una variedad de dispositivos para garantizar una experiencia consistente para los usuarios.

2.2.2. Desarrollo de aplicaciones

El desarrollo de aplicaciones Android es el proceso de crear aplicaciones móviles para dispositivos que funcionan con el sistema operativo Android. El entorno de desarrollo recomendado es Android Studio [48], que es el IDE oficial para el desarrollo de aplicaciones Android. Android Studio proporciona una interfaz de usuario intuitiva y una amplia gama de herramientas para ayudar a desarrollar y depurar aplicaciones Android. Las Figuras 2.3 y 2.4 muestran algunas de las características de Android Studio.

Los principales lenguajes de programación utilizados para el desarrollo de aplicaciones Android son Java y Kotlin. Java ha sido el lenguaje tradicionalmente utilizado en Android, pero Kotlin ha ganado popularidad en los últimos años debido a que lleva a código muy conciso y tiene chequeo de tipos explícito para valores *null*. Ambos lenguajes son oficialmente admitidos por la plataforma Android.

Las interfaces de usuario de las aplicaciones Android pueden ser definidas tanto en archivos XML como programáticamente, ya sea usando Java o Kotlin. En cualquiera de los casos, los desarrolladores deben definir los elementos de UI que tendrá cada interfaz en la aplicación, como los botones y los campos de texto, y especificar sus atributos, como el color y el tamaño [49]. El uso de archivos XML era hasta hace unos años una práctica estándar entre desarrolladores Android y recomendada oficialmente por la plataforma Android. Sin embargo, en los últimos años, el uso de Kotlin para definir las interfaces de usuario ha ganado popularidad, ya que permite definir las interfaces de usuario de forma más concisa y con menos código que Java. Esto se logra mediante el uso del *framework Jetpack Compose* [50],

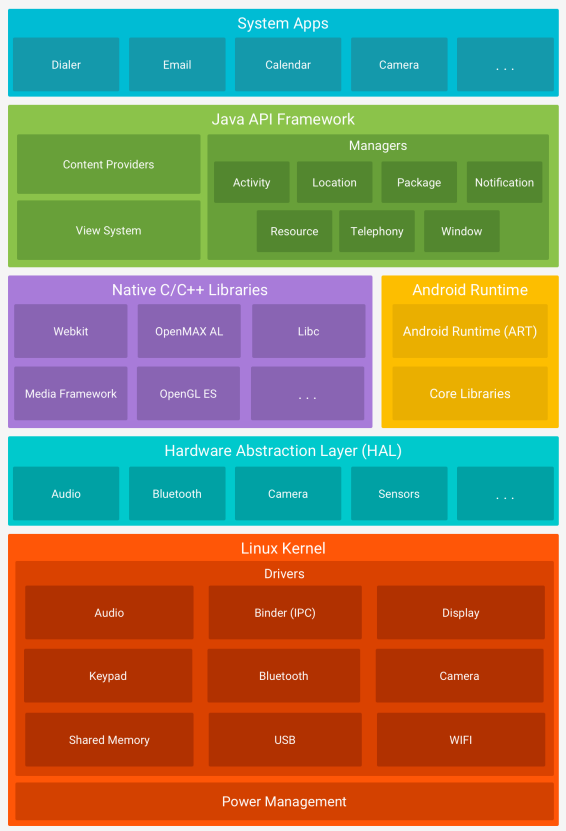


Figura 2.2: La arquitectura del sistema operativo ANDROID, como se muestra en la documentación oficial [47].

que permite definir las interfaces de usuario de forma declarativa, en lugar de imperativa. *Jetpack Compose* se basa en el *framework* declarativo *React* [51] y es soportado oficialmente por la plataforma ANDROID.

2.2.3. Compilación de aplicaciones

El sistema de compilación de ANDROID es el encargado de compilar los recursos y el código fuente de la aplicación, para luego empaquetarlos en archivos de formato *ANDROID Application Package* (APK) o *ANDROID App Bundle* (AAB) [52]. Estos dos formatos de archivo son los formatos de paquete de aplicaciones ANDROID que se utilizan para distribuir e instalar aplicaciones y módulos en dispositivos ANDROID. Sin embargo, la compilación de un proyecto también puede producir otros tipos de archivos, como bibliotecas de archivos de aplicaciones ANDROID (AAR) o archivos de bibliotecas de JAVA (JAR).

La plataforma ANDROID utiliza GRADLE [53] para automatizar y gestionar el proceso de compilación. GRADLE es un conjunto de herramientas de compilación avanzadas que permite definir configuraciones de compilación personalizadas y flexibles, por ejemplo, para establecer diferentes versiones de un programa. Cada configuración declara el código fuente y recursos a utilizar, permitiendo reutilizar las partes comunes

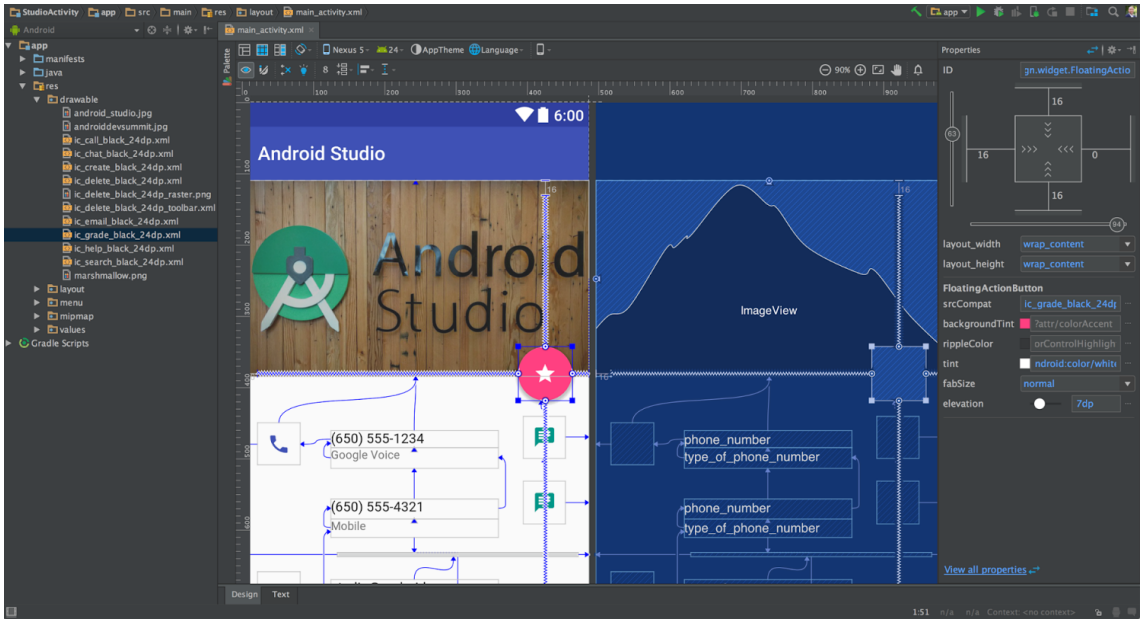


Figura 2.3: Captura de pantalla de ANDROID Studio: diseño de UI.

a todas las versiones del programa.

El *plugin* GRADLE de ANDROID proporciona procesos y configuraciones que son específicas para la compilación y *testing* de aplicaciones ANDROID. Tanto GRADLE como el *plugin* GRADLE de ANDROID se pueden utilizar de forma independiente a ANDROID Studio. Esto significa que un desarrollador puede compilar sus aplicaciones ANDROID desde dentro de ANDROID Studio, desde la línea de comandos de su máquina o en máquinas donde ANDROID Studio no está instalado, como servidores de integración continua.

2.2.4. Componentes de una aplicación

Las aplicaciones ANDROID están compuestas por una serie de componentes interconectados [54]. Los componentes principales son:

Actividades. A diferencia de los programas de escritorio tradicionales que se inician con un método principal (`main`), las aplicaciones ANDROID se inician a través de componentes llamados *actividades* (*activities*, en inglés) [55], que representan las pantallas con las que el usuario puede interactuar. Entonces, la experiencia de un usuario con una aplicación móvil difiere de la versión de escritorio, ya que la interacción del usuario con la app no siempre comienza en el mismo lugar. Por ejemplo, si un usuario abre una app de correo electrónico desde la pantalla principal, es posible que vea una lista de correos electrónicos recibidos. Por el contrario, si el usuario usa una app de redes sociales que luego inicia la app de correo electrónico, es posible que acceda directamente a la pantalla de redacción.

La mayoría de las apps contienen varias pantallas, lo cual significa que incluyen

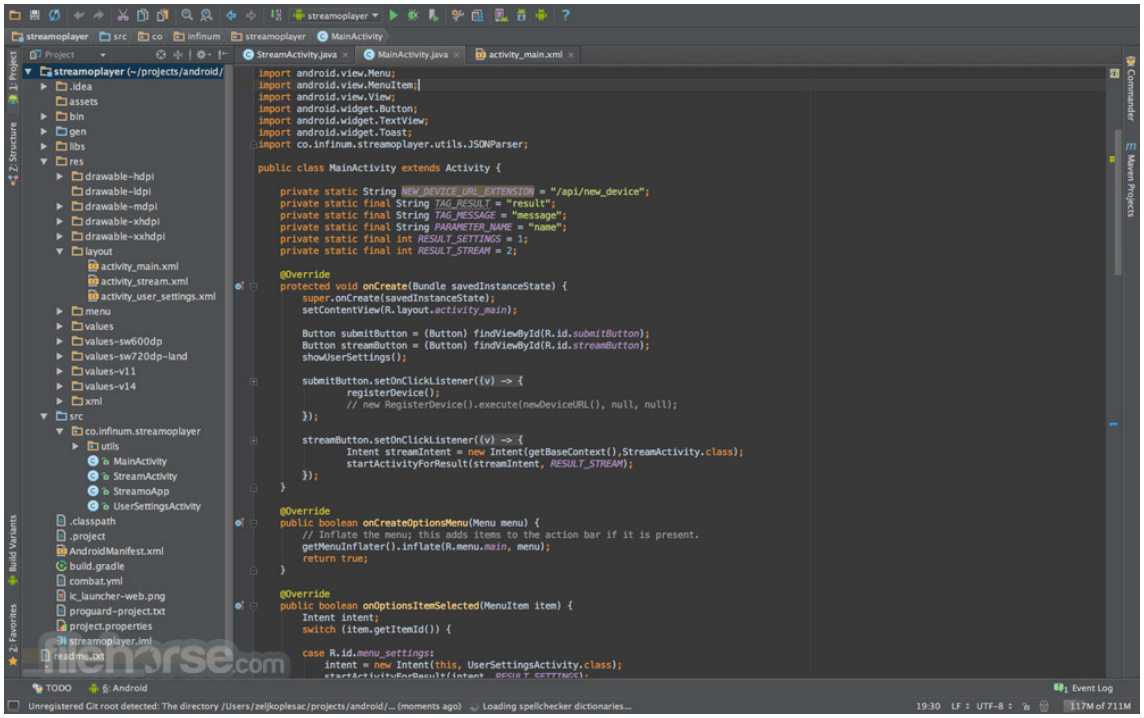


Figura 2.4: Captura de pantalla de ANDROID Studio: programando en KOTLIN.

varias actividades. Para cada una de estas, los desarrolladores deben crear una clase JAVA correspondiente que herede de la clase `Activity`, provista por el SDK de ANDROID. Cada actividad define su propia interfaz de usuario, con elementos como botones, campos de texto, imágenes y otros componentes visuales con los que el usuario puede interactuar [56]. Por lo general, los desarrolladores especifican una actividad como *principal*, que es la primera pantalla que aparece cuando el usuario inicia la app.

Las actividades cuentan con un ciclo de vida definido que incluye una serie de estados y transiciones [57] (ilustrado en la Figura 2.5). Cada uno de estas transiciones se corresponde con un método que los desarrolladores pueden sobrescribir para realizar acciones específicas en cada momento del ciclo de vida de la actividad. Algunos de estos métodos son:

- `onCreate()`: Se llama cuando se crea la actividad por primera vez. Aquí se realiza la inicialización de la actividad, como la creación de la interfaz de usuario y la vinculación de datos.
- `onStart()`: Indica que la actividad se está volviendo visible para el usuario. En este punto, la actividad está a punto de ser mostrada en pantalla.
- `onResume()`: La actividad está en primer plano y el usuario puede interactuar con ella. Aquí se realizan tareas como la recuperación de recursos o la actualización de la interfaz de usuario.
- `onPause()`: Se llama cuando la actividad pierde el foco, generalmente debido a que otra actividad se vuelve visible. En este estado, la actividad aún es visible,

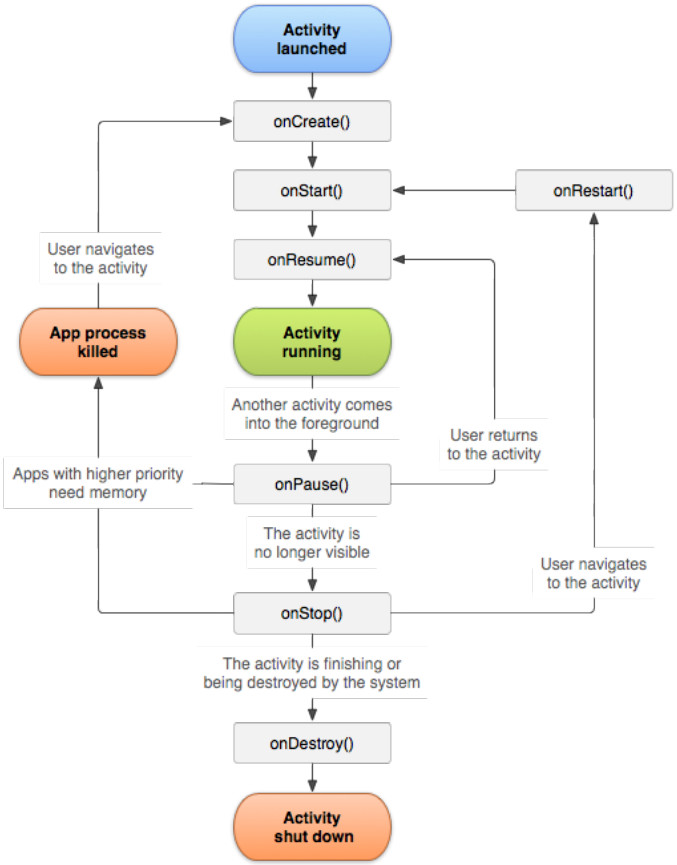


Figura 2.5: Una ilustración simplificada del ciclo de vida de una actividad en la plataforma Android, como se muestra en la documentación oficial [57].

pero no está en primer plano.

- `onStop()`: La actividad ya no es visible para el usuario y se detiene. Aquí se pueden realizar tareas de limpieza o liberación de recursos.
- `onDestroy()`: Se llama cuando la actividad está siendo destruida. En este punto, se liberan todos los recursos utilizados por la actividad.

Finalmente, las actividades pueden comunicarse entre sí mediante la transferencia de datos. Esto se puede lograr utilizando intenciones (*intents* en inglés) para iniciar una nueva actividad y pasar información a través de extras o mediante el uso de métodos como `startActivityForResult()` y `onActivityResult()` para recibir resultados de actividades secundarias.

Fragmentos. Los fragmentos representan una parte modular de la UI y pueden combinarse o reutilizarse en múltiples actividades [58], permitiendo dividir la UI de una aplicación en partes más pequeñas y manejables. Por ejemplo, un fragmento que muestra una lista de elementos puede utilizarse en diferentes actividades para mostrar la lista en diferentes contextos.

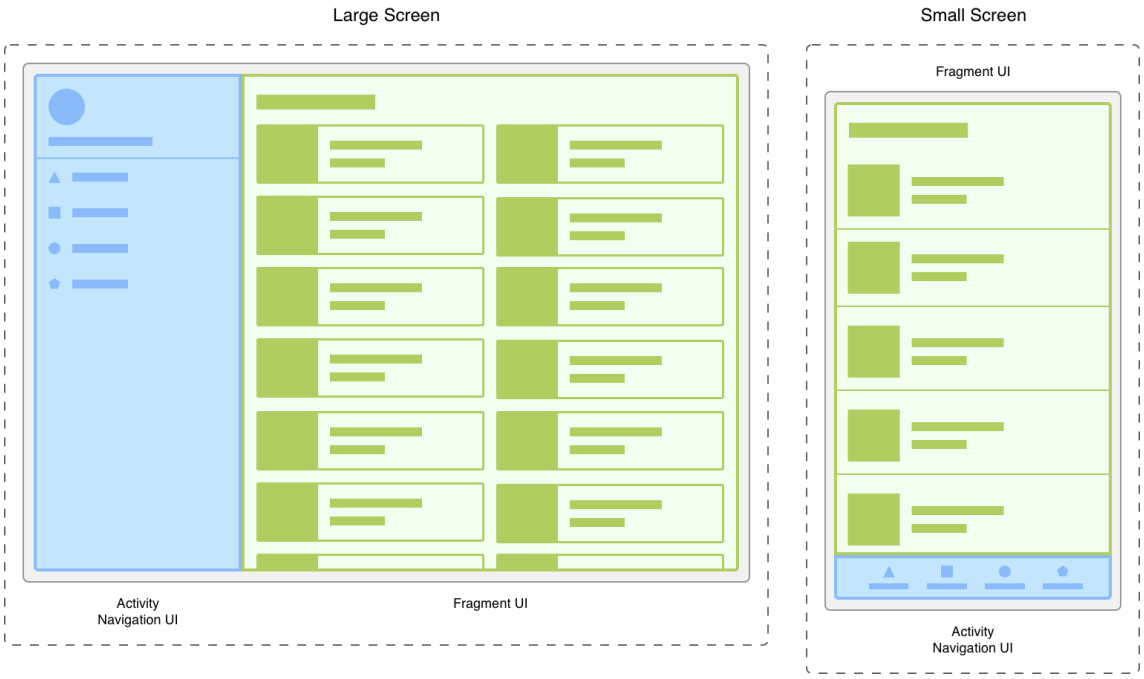


Figura 2.6: Ejemplo ilustrativo de la ventaja de utilizar fragmentos para diseñar aplicaciones ANDROID, como se muestra en la documentación oficial [58].

Los fragmentos son especialmente útiles para el diseño adaptable, ya que permiten reorganizar y cambiar la apariencia de la UI en función de las dimensiones y orientación de la pantalla. Se pueden utilizar diferentes fragmentos para mostrar diferentes vistas según el tamaño de la pantalla o la orientación del dispositivo, lo que proporciona una experiencia de usuario más consistente y optimizada para cada dispositivo (ilustrado en la Figura 2.6)

Al igual que las actividades, los fragmentos tienen su propio ciclo de vida. Esto significa que los fragmentos tienen métodos como `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()` y `onDestroy()` que se asemejan a los métodos de las actividades. Sin embargo, los fragmentos tienen también algunos métodos adicionales, como `onCreateView()` y `onDestroyView()`, que están relacionados con la creación y destrucción de la vista del fragmento.

Servicios. Los servicios en ANDROID son componentes que permiten ejecutar operaciones de larga duración en segundo plano, sin una UI visible [59]. Dichos componentes son utilizados para realizar tareas que no requieren interacción directa con el usuario y pueden continuar ejecutándose incluso cuando la aplicación se encuentra en segundo plano o el dispositivo se bloquea. Por lo tanto, son útiles para tareas que no requieren una visualización activa, pero que siguen siendo esenciales para el funcionamiento de la aplicación. Los servicios tienen su propio ciclo de vida similar al de las actividades, aunque con algunas diferencias. Los métodos del ciclo de vida de un servicio incluyen:

- `onCreate()`: Se llama cuando se crea el servicio y se realiza la inicialización necesaria.
- `onStartCommand()`: Se llama cuando se inicia el servicio. Aquí es donde se inicia la tarea en segundo plano.
- `onDestroy()`: Se llama cuando el servicio se detiene o se destruye. Aquí es donde se liberan los recursos y se finalizan las tareas en curso.

Los servicios pueden comunicarse con otras actividades, fragmentos o servicios a través de mecanismos como la transmisión de intenciones (*intents*) o el uso de enlaces de servicios (*service bindings*). Esto permite que los componentes de la aplicación interactúen con el servicio, por ejemplo, para enviar comandos o recibir actualizaciones de estado.

Además de los servicios en segundo plano, ANDROID también admite servicios en primer plano. Un servicio en primer plano muestra una notificación permanente en la barra de notificaciones, lo que indica al usuario que el servicio está en ejecución. Los servicios en primer plano se utilizan para tareas que requieren una mayor prioridad y deben mantenerse en ejecución incluso ante escenarios de escasa memoria.

Proveedores de contenido. Los proveedores de contenido (*content providers*, en inglés) en ANDROID son componentes que permiten acceder y compartir datos entre diferentes aplicaciones [60, 61]. Dichos componentes actúan como una interfaz para acceder a una base de datos, archivos u otro origen de datos y proporcionan métodos para realizar operaciones *CRUD* (crear, leer, actualizar y eliminar) en esos datos. Es decir, facilitan la separación de los datos y su lógica de acceso, permitiendo que diferentes aplicaciones accedan a los datos de manera segura y controlada.

Los proveedores de contenido permiten además establecer permisos de acceso para garantizar la seguridad y privacidad de los datos. Las aplicaciones deben tener los permisos adecuados (definidos en el archivo de manifiesto de la aplicación) para acceder a los proveedores de contenido y manipular los datos. Esto ayuda a prevenir el acceso no autorizado a los datos y mantener la integridad de los mismos.

ANDROID proporciona varios proveedores de contenido predefinidos, como el proveedor de contactos (para acceder a la información de contactos del dispositivo), el proveedor de media (para acceder a archivos multimedia como imágenes o videos) y el proveedor de SMS (para acceder a los mensajes de texto). Además, los desarrolladores también pueden crear sus propios proveedores de contenido personalizados para exponer y compartir datos específicos de sus aplicaciones.

Receptores de difusión. Los receptores de difusión (*broadcast receivers*, en inglés) son componentes fundamentales en el desarrollo de aplicaciones ANDROID que permiten a las aplicaciones recibir y responder a eventos o mensajes del sistema y de otras aplicaciones [62]. Los receptores de difusión actúan como "oyentes" que esperan y responden a las difusiones (*broadcasts*) enviadas por otros procesos. Concretamente, una difusión es un mensaje que se envía a través del sistema para notificar sobre

eventos o cambios, como la conexión de un cargador, el inicio de una llamada telefónica o la instalación de una nueva aplicación.

Los desarrolladores pueden registrar un receptor de difusión de dos formas: de manera dinámica o de manera declarativa. El registro dinámico se realiza mediante el código en tiempo de ejecución, mientras que el registro declarativo se realiza a través del archivo de manifiesto de la aplicación. El registro declarativo permite que el receptor de difusión se inicie automáticamente cuando se recibe una difusión específica, incluso si la aplicación no está en ejecución en ese momento.

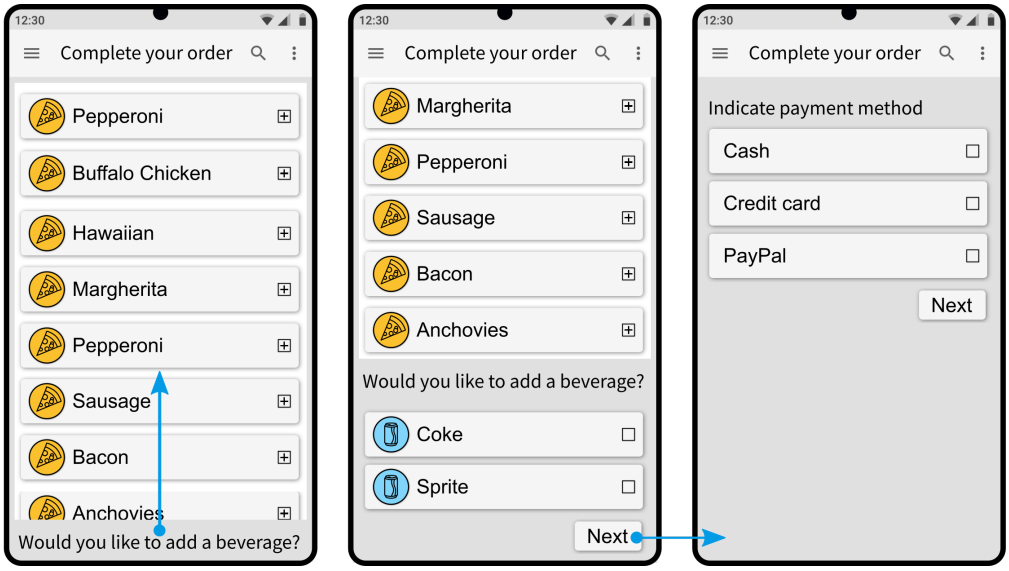
Los receptores de difusión pueden utilizar filtros de intenciones (*intents*) para especificar qué difusiones desean recibir. Un filtro de intenciones define una combinación de acciones, categorías y datos que deben coincidir con la difusión para que el receptor la reciba. Esto permite que la aplicación seleccione qué difusiones son relevantes para ella y responda únicamente a esas difusiones específicas.

Una vez que un receptor de difusión recibe una difusión, puede realizar acciones en respuesta a ella. Esto puede incluir iniciar una actividad, iniciar un servicio, mostrar una notificación, modificar datos, reproducir un sonido, entre otras acciones. La respuesta del receptor de difusión puede variar según la naturaleza del evento y los requisitos de la aplicación. Los receptores de difusión tienen un ciclo de vida breve y se ejecutan de manera asíncrona. Cuando se recibe una difusión, se invoca el método `onReceive()` del receptor de difusión para manejarla. Una vez que el método `onReceive()` finaliza su ejecución, el receptor de difusión se considera inactivo y se libera de la memoria.

2.2.5. Archivo de manifiesto

Las aplicaciones ANDROID deben contener un archivo de manifiesto (*manifest*, en inglés) que describa la estructura y las características de la aplicación [63]. Este archivo se denomina `AndroidManifest.xml` y se encuentra en el directorio raíz del proyecto de la aplicación. El sistema operativo ANDROID utiliza este archivo para comprender la estructura y las funcionalidades de la aplicación, asegurando una ejecución adecuada e interacción con el sistema operativo del dispositivo y otras aplicaciones.

El manifiesto de una app declara por ejemplo su "nombre de paquete" (*package names*, en inglés). Dichos nombres son elegidos por los desarrolladores y suelen seguir una estructura jerárquica, generalmente en notación de dominio inverso (e.g., `com.example.myapp`). Los nombres de paquete juegan un papel importante en la implementación de aplicaciones, ya que sirven luego para identificar de forma única a cada aplicación en la plataforma ANDROID y en la tienda Google Play Store. El archivo `AndroidManifest.xml` incluye además detalles como el código de versión, los permisos necesarios para acceder a varias características del dispositivo, las actividades, los servicios y los receptores utilizados dentro de la aplicación.



(a) Ejemplo de UI Android. El usuario *desliza* hacia arriba para revelar el resto de la lista, y luego hace *click* en el botón "Next" para revelar la segunda página del formulario.

```
<CardView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ImageView android:id="@+id/pizzaIcon"
        android:contentDescription="@string/pizza_icon" />
    <TextView android:id="@+id/pizzaName"
        android:text="@string/pepperoni" />
    <ImageButton android:id="@+id/addItem"
        android:contentDescription="@string/add_item_button" />
</CardView>
```

(b) Ejemplo de código XML para la fila "Pepperoni" en la lista de la Figura 2.7a.

Figura 2.7: Ejemplo de UI Android y código XML asociado.

2.2.6. Interfaz de usuario

Como se mencionó, la UI de una actividad Android se puede definir tanto mediante código como mediante archivos XML (como se muestra en la Figura 2.7b). En ambos casos, los desarrolladores deben proporcionar una *Jerarquía de Vistas* (*View Hierarchy*, en inglés) que será interpretada por el sistema operativo Android para representar la pantalla. Luego, los usuarios interactúan con la aplicación móvil a través de acciones en las vistas visibles. Por ejemplo, al hacer *click* en el `ImageButton` "Agregar elemento" (es decir, el botón ⊞) en las filas de la Figura 2.7a. Entre otras, algunas de las acciones estándar que pueden ser realizadas sobre las vistas son *click*, *long click* y *scrolling*.

Una Jerarquía de Vistas consiste de una estructura similar a un árbol, con una

vista raíz en la parte superior y vistas secundarias que se ramifican. La clase *View* es el bloque de construcción básico de las UI de ANDROID, ocupando un espacio rectangular en la pantalla y encargándose tanto del dibujo como del manejo de eventos. Sirve como base para elementos interactivos, como botones y campos de texto (conocidos informalmente como *widgets*). Los nodos internos en la Jerarquía de Vistas son *Grupos de Vistas*, que son vistas que pueden contener otras vistas. Estos grupos, que sirven como contenedores invisibles para vistas y administran su posicionamiento, se crean utilizando la subclase *ViewGroup*.

Las vistas pueden tener atributos o propiedades que definen su apariencia y comportamiento. Por ejemplo, el atributo `android:text` define el texto que se muestra en un `TextView`, mientras que el atributo `android:hint` define el texto que se muestra en un `EditText` cuando está vacío. Estos atributos se pueden definir tanto en el código como en los archivos XML.

Un atributo clave de las vistas es el `android:id`, que define un identificador para la vista (en inglés, *resource id*). Este identificador se utiliza para referirse a la vista desde el código de la aplicación, por ejemplo, para modificar su contenido o comportamiento. Además, el identificador se utiliza para referirse a la vista desde los archivos XML, por ejemplo, para definir relaciones de jerarquía entre vistas. Es importante destacar que las vistas pueden no tener un identificador definido, y que estos identificadores no son necesariamente únicos dentro de la aplicación, o incluso dentro de la actividad. Es decir, es posible que dos vistas diferentes tengan el mismo identificador en una jerarquía de vistas.

3

Frameworks y herramientas de testing para Android

3.1. Frameworks de testing

Al igual que con otros tipos de software, el *testing* es una parte integral del proceso de desarrollo de aplicaciones Android [64]. Esta tarea permite verificar la corrección, el comportamiento funcional y la usabilidad de una aplicación antes de lanzarla públicamente. Los desarrolladores pueden realizar *testing* manual de su aplicación navegando a través de ella. Pueden utilizar diferentes dispositivos y emuladores, cambiar el idioma del sistema y tratar de generar todos los errores de usuario o recorrer todos los flujos de usuario. Sin embargo, el *testing* manual no escala bien y puede ser fácil pasar por alto regresiones en el comportamiento de la aplicación. El *testing* automatizado implica el uso de herramientas que ejecutan los *tests* automáticamente, lo que es más rápido, repetible y generalmente brinda información más útil sobre la aplicación antes en el proceso de desarrollo.

Varios *frameworks* de *testing* han sido creados para ayudar a los desarrolladores de Android a escribir casos de *test* y automatizar su ejecución. Para la creación de *test* unitarios encontramos: JUnit [65] y Robolectric [24]. Por otro lado, para creación de casos de *test* de UI tenemos: AndroidViewClient [66], Appium [19], Calabash [20], Compose [67], Espresso [23], MonkeyRunner [21], Python-UIAutomator [68], Robotium [18], y UIAutomator [22].

JUnit [65] es un *framework* ampliamente utilizado para realizar casos de *test* unitarios en Java. En proyectos de Android, los casos de *test* de JUnit puros solo se pueden utilizar para ejercitar clases que no interactúan con el sistema operativo de Android. Para ejercitar los componentes de la interfaz gráfica de una aplicación, como actividades y vistas, un desarrollador necesita escribir *tests instrumentados*, las cuales se ejecutan en dispositivos Android, ya sean físicos o emulados [69]. Por lo tanto, pueden aprovechar las APIs del sistema operativo Android.

Los *tests* instrumentados (también conocidos como *tests* de *instrumentación*) brindan mayor fidelidad que los *tests* unitarios, aunque se ejecutan más lentamente. Estos *tests* se inicializan en un entorno especial que les brinda acceso a una instancia de la clase `Instrumentation` [17], lo que permite a los desarrolladores monitorear todas las interacciones del sistema Android con la aplicación bajo *test* (*Application Under Test* o AUT, en inglés). La clase `Instrumentation` es la clase base para

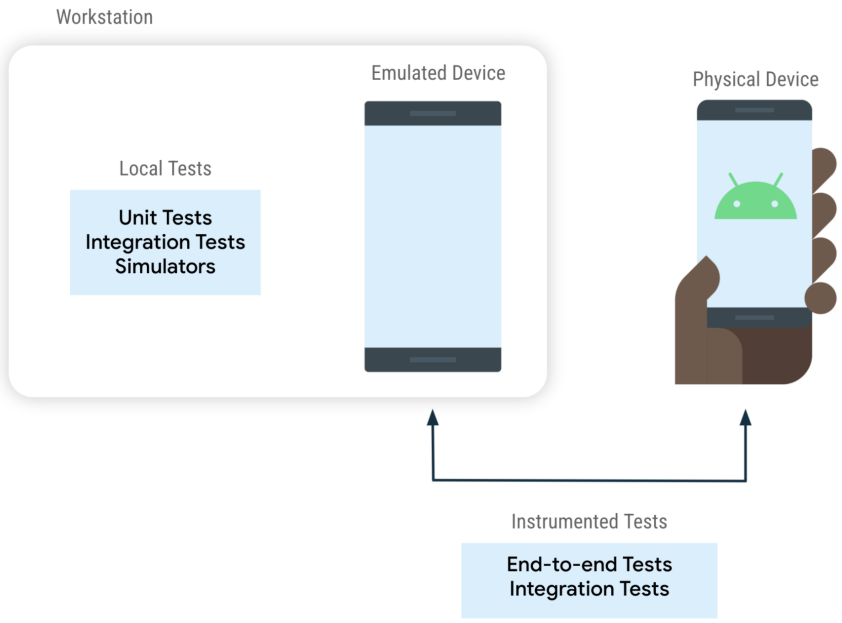


Figura 3.1: Ilustración de los distintos tipos de *tests* disponibles para la plataforma ANDROID, como se muestra en la documentación oficial [64].

implementar código de instrumentación de aplicaciones. Cuando se ejecuta con la instrumentación activada, esta clase es instanciada antes que cualquier código de la aplicación, lo que permite monitorear toda la interacción del sistema con la aplicación. Para ejecutar los *tests* instrumentados, los desarrolladores deben utilizar la clase `AndroidJUnitRunner` [70]. Esta clase es un *test runner* de JUNIT que permite ejecutar *tests* instrumentados en dispositivos ANDROID, incluidos aquellos *tests* que utilizan los *frameworks* ESPRESSO [23], UIAUTOMATOR [22] y COMPOSE [67]. La Figura 3.1 ilustra los distintos tipos de *tests* visualmente.

ROBOLECTRIC [24] es un *framework* open-source diseñado para realizar *tests* unitarios de aplicaciones ANDROID. A diferencia de los *tests* instrumentados tradicionales que se ejecutan en dispositivos físicos o emulados, ROBOLECTRIC permite a los desarrolladores ejecutar *tests* directamente en la Máquina Virtual de JAVA (*JAVA Virtual Machine* o JVM, en inglés). Esto mejora significativamente la velocidad de *testing* y permite obtener una respuesta rápida durante el proceso de desarrollo. El *framework* proporciona un entorno ANDROID simulado, lo que permite que los *tests* interactúen con componentes ANDROID como actividades, vistas y fragmentos sin necesidad de un dispositivo real. ROBOLECTRIC puede manejar la carga de recursos, la gestión del ciclo de vida y la simulación de eventos, convirtiéndolo en una herramienta poderosa para realizar *testing* de la lógica de la aplicación ANDROID en aislamiento. Su capacidad para imitar el entorno de ejecución ANDROID lo convierte en una opción eficiente y valiosa para los desarrolladores que buscan garantizar la confiabilidad y estabilidad de sus aplicaciones ANDROID.

El *framework* de *testing* UIAUTOMATOR [22] es una herramienta para realizar *testing* de aplicaciones ANDROID, desarrollada por Google. Permite a los desarrolla-

dores crear casos de *test* automatizados que interactúan con la interfaz de usuario. UIAutomator está diseñado para ejercitar las aplicaciones en diferentes dispositivos y versiones de Android. Con su capacidad para realizar varias acciones de interfaz de usuario como tocar botones, ingresar texto y deslizar, el *framework* puede simular interacciones de usuario de manera efectiva. Además, puede acceder a elementos y vistas fuera de la aplicación bajo *test*, lo que permite un *testing* exhaustivo de las interacciones entre diferentes aplicaciones.

PythonUIAutomator [68] es un *framework* open-source Python que permite a los desarrolladores escribir casos de *test* en este lenguaje de programación para aplicaciones Android utilizando UIAutomator.

Robotium [18] es un *framework* open-source diseñado para el *testing* automatizado de aplicaciones Android. Proporciona una API que permite a los desarrolladores escribir casos de *test* robustos y eficientes para sus aplicaciones Android. Este *framework* es particularmente útil para el *testing* de caja negra, ya que permite el *testing* de aplicaciones sin requerir acceso al código fuente.

Calabash [20] es un *framework* open-source utilizado para *testing* automatizado de aplicaciones Android e iOS, desarrollado por *Xamarin*. Este *framework* está integrado con Cucumber [71], una herramienta de *testing* BDD (*Behavior-Driven Development*, en inglés), que permite escribir escenarios de *test* en sintaxis Gherkin [72]. Gherkin utiliza construcciones de lenguaje natural, lo que lo hace más accesible para los miembros del equipo de desarrollo no técnicos. Calabash también proporciona una API Ruby para interactuar con la interfaz de usuario de la aplicación bajo *test*.

Espresso [23] es un *framework* de *testing* ampliamente utilizado y diseñado específicamente para automatizar el *testing* de aplicaciones Android [25]. Desarrollado por Google, Espresso ofrece una API robusta y fácil de usar que simplifica la creación y ejecución de *tests* de interfaz de usuario. Permite a los desarrolladores simular interacciones de usuario, como toques, deslizamientos y entradas de texto, y luego verificar las respuestas y el comportamiento de la aplicación. Las capacidades de sincronización de Espresso garantizan que los *tests* esperen a que los elementos de la interfaz de usuario estén listos antes de realizar acciones, lo que resulta en *tests* confiables y estables. Su sintaxis concisa y expresiva permite a los desarrolladores escribir *scripts* de *test* legibles, lo que promueve un mantenimiento y una colaboración más fáciles dentro del equipo de desarrollo.

Appium [19] es un *framework* open-source para *testing* de aplicaciones móviles en Android e iOS. Permite a los desarrolladores escribir casos de *test* para aplicaciones nativas, híbridas y web usando una variedad de lenguajes de programación como Java, Python, Javascript, Ruby y más. Estos casos de *test* pueden ser ejecutados en múltiples plataformas, como Android e iOS, lo que permite la reutilización de código. Appium está construido sobre el protocolo WebDriver [73], que proporciona una interfaz común para interactuar con aplicaciones web y móviles. Funciona mediante la interacción con la aplicación utilizando los *frameworks* de automatización nativos proporcionados por cada plataforma (UIAutomator [74, 75] o Espresso [76] para Android y XCTest [77] para iOS), lo que garantiza resultados de *test* precisos y

confiables.

MonkeyRunner [21] es un *framework* de *testing* basado en Python desarrollado por Google para automatizar el *testing* de aplicaciones Android. Permite a los desarrolladores escribir *scripts* para interactuar con aplicaciones en dispositivos reales o emuladores, realizando tareas como tocar, deslizar y escribir texto. El *framework* ofrece funcionalidad básica de grabación y reproducción y permite el control de aplicaciones, como iniciar y detener aplicaciones o instalar APKs. Si bien es fácil de usar y flexible, es menos especializado para *testing* de interfaz de usuario en comparación con Espresso o UIAutomator, que Google promueve para soluciones de *testing* de aplicaciones Android más completas.

AndroidViewClient [66] es un *framework* open-source para automatizar el *testing* de aplicaciones Android. Esta herramienta fue desarrollada originalmente como una extensión de MonkeyRunner, pero últimamente evolucionó como una herramienta puramente Python que automatiza o simplifica la creación de *scripts* de *test*. Proporciona una API Python que permite a los desarrolladores e ingenieros de *testing* interactuar con dispositivos Android y emuladores, inspeccionar sus elementos de interfaz de usuario y realizar acciones automatizadas sobre ellos. El *framework* funciona conectándose al dispositivo a través del *Android Debug Bridge* (ADB) y capturando la jerarquía de la interfaz de usuario del dispositivo, que luego se puede recorrer y manipular programáticamente.

El *framework* de *testing* Compose [67] se utiliza para verificar el comportamiento correcto de las interfaces de usuario definidas utilizando el *framework Jetpack Compose*. Compose proporciona un conjunto de APIs de *testing* para encontrar elementos, verificar sus atributos y realizar acciones de usuario. El *testing* de una interfaz de usuario creada con *Jetpack Compose* es diferente al *testing* de una interfaz de usuario basada en vistas (*Views*). Una vista ocupa un espacio rectangular y tiene propiedades, como identificadores, posición, margen, relleno, etc. En el paradigma del *framework Jetpack Compose*, solo algunos componentes emiten una interfaz de usuario en la jerarquía de la interfaz de usuario, por lo que se necesita un enfoque diferente para encontrar elementos de la interfaz de usuario. Los *tests* de interfaz de usuario de Compose utilizan la semántica para interactuar con la jerarquía de la interfaz de usuario. La semántica, como su nombre lo indica, da significado a una pieza de la interfaz de usuario. En este contexto, una "pieza de la interfaz de usuario" (o elemento) puede significar cualquier cosa, desde un solo componente hasta una pantalla completa. El árbol de semántica se genera junto con la jerarquía de la interfaz de usuario y la describe.

Los *frameworks* Calabash, Robotium, y MonkeyRunner están "*deprecados*" y ya no reciben mantenimiento. Por otro lado, es importante mencionar que la herramienta UIAutomator utiliza internamente para su implementación el *Servicio de Accesibilidad* de Android [78] para obtener información sobre la interfaz de usuario y realizar acciones de *click* y *scroll* [79–81]. Sin embargo, el objetivo principal de la API del *Servicio de Accesibilidad* es ayudar a los usuarios con discapacidades a utilizar dispositivos y aplicaciones Android, y no está diseñada para fines de *testing*. En particular, puede devolver información inconsistente [82], como el *nombre*

de clase impreciso de las vistas, informar incorrectamente el *hint* (i.e., sugerencia) de un campo como entrada de *texto*, falta de propiedades de *descripción de contenido* (*content description*, en inglés) y proporcionar textos con mayúsculas incorrectas.

Por lo tanto, todas las herramientas o *frameworks* que que utilicen directamente el *Servicio de Accesibilidad* para automatizar el *testing* de aplicaciones ANDROID, o que lo utilicen indirectamente a través de UIAUTOMATOR, pueden verse afectados por estos problemas. De los *frameworks* mencionados arriba, tanto PYTHONUIAUTOMATOR como ANDROIDVIEWCLIENT utilizan UIAUTOMATOR en su implementación. El *framework* ESPRESSO no tiene este problema porque utiliza directamente la clase de *Instrumentation* de ANDROID para interactuar con la aplicación bajo *test* [83].

3.2. Herramientas del estado del arte

Diversas herramientas y técnicas han sido propuestas a lo largo de los últimos años para realizar *testing* automático de aplicaciones ANDROID [28–32].

MONKEY [84] es una herramienta de *testing* automático para aplicaciones ANDROID desarrollada por Google. Esta herramienta es muy popular en entornos industriales para probar aplicaciones ANDROID, en parte porque forma parte del kit de herramientas de desarrollo de ANDROID y no requiere ningún esfuerzo adicional de instalación. Además, es compatible con diferentes versiones de la plataforma ANDROID. MONKEY implementa la estrategia aleatoria más básica, generando secuencias pseudoaleatorias de eventos de UI (por ejemplo, *clicks*, *taps* y gestos en la interfaz de usuario) y eventos de nivel de sistema (por ejemplo, cambiar el volumen). Los usuarios de esta herramienta deben especificar la cantidad de eventos que desean que MONKEY genere. Una vez que se alcanza este límite máximo, o se encuentra un *crash*, MONKEY se detiene.

DYNODROID [85] también se basa en la exploración aleatoria, pero tiene varias características que hacen que su exploración sea más eficiente en comparación con MONKEY. En primer lugar, esta herramienta genera solo eventos del sistema que son relevantes para la aplicación. DYNODROID obtiene esta información monitoreando cuándo una aplicación registra un receptor de difusión dentro de la plataforma ANDROID. Por esta razón, necesita instrumentar el sistema operativo ANDROID. La estrategia de generación de eventos aleatorios de DYNODROID es más inteligente que la que implementa MONKEY. Puede seleccionar los eventos que se han utilizado menos frecuentemente (estrategia de "Frecuencia") y puede tener en cuenta el contexto (estrategia de "*BiasedRandom*"), es decir, los eventos que son relevantes en más contextos se utilizaran con mayor frecuencia. Una mejora adicional de DYNODROID es la capacidad de permitir a los usuarios proporcionar manualmente *input* (por ejemplo, para autenticación) cuando la exploración se detiene.

SAPIENZ [86, 87] es una herramienta basada en algoritmos evolutivos para generación de casos de *test* de UI ANDROID. Utiliza un algoritmo genético multiobjetivo [33] para evolucionar secuencias de *inputs* generadas como semillas y buscar *test suites* optimizados que contengan secuencias de *input* cortas mientras maximizan la cobertura de código y la detección de fallas. Se aprovechan secuencias de *input*

predefinidas (llamadas *motif genes*) para complementar la exploración aleatoria y proporcionar ejercicios locales para diferentes tipos de elementos de UI. Los recursos de texto dentro de las aplicaciones se extraen como semillas para las entradas de texto. Mao et al. [86] han mostrado que SAPIENZ puede superar tanto a MONKEY como a DYNODROID en términos de cobertura de código y detección de fallas. Este éxito llevó a su posterior implementación en producción en Meta [88] (previamente, Facebook) donde actualmente se utiliza para generar automáticamente casos de *test* de UI para aplicaciones ANDROID [89].

STOAT [90] es una herramienta de *testing* de UI ANDROID basada en modelos estocásticos. Utiliza una exploración dinámica para construir y evolucionar un modelo de estados-transiciones probabilístico de la UI, del cual se generan los casos de *test*. Específicamente, STOAT opera en dos fases: (1) Dada una aplicación como entrada, utiliza análisis dinámico mejorado por una estrategia de exploración de UI ponderada y análisis estático para producir un modelo estocástico de las interacciones de la UI en la aplicación; y (2) adapta el muestreo de Gibbs [91] para mutar/refinar de forma iterativa el modelo estocástico. Luego, STOAT guía la generación de *tests* a partir de los modelos mutados para lograr una alta cobertura de código y modelo y exhibir secuencias diversas. Durante el *testing*, se inyectan eventos a nivel de sistema de forma aleatoria para mejorar aún más la efectividad de los *tests*. Ting Su et al. [90] han mostrado que STOAT puede superar a SAPIENZ.

APE [92] es una herramienta de *testing* de UI basada en modelos. A diferencia de herramientas anteriores basadas en modelos como STOAT, que utilizan criterios de abstracción de UI estáticos, APE utiliza la información en tiempo de ejecución para evolucionar dinámicamente su criterio de abstracción mediante un árbol de decisiones, lo que permite equilibrar de manera efectiva el tamaño y la precisión del modelo. Específicamente, con este modelo refinado de forma dinámica, APE genera eventos de UI mediante una estrategia de exploración de estado aleatoria y *greedy deep-first*. Además, APE también utiliza internamente MONKEY para emitir ocasionalmente eventos de UI aleatorios y eventos del sistema para evitar quedarse atrapado en estados locales.

HUMANOID [93] es una de las primeras herramientas de *testing* para ANDROID basada en aprendizaje profundo. El componente principal de esta herramienta es un modelo de red neuronal profunda que predice qué elementos de la UI actual es más probable que sean interactuados por los usuarios y cómo interactuar con ellos. Dicho modelo fue entrenado con un gran conjunto de datos de interacciones humanas obtenidas mediante *crowdsourcing*. Se espera que HUMANOID dirija la exploración de la UI hacia estados importantes más rápidamente, ya que prioriza los elementos de la UI según su importancia y significado, como lo haría un humano.

COMBODROID [94] es una herramienta de *testing* basada en modelos. Su idea principal es generar secuencias de eventos largas y significativas al combinar varios casos de *test* cortos e independientes, para explorar estados profundos de la aplicación. COMBODROID obtiene estos casos de *test* ya sea de humanos o los genera automáticamente a partir de un modelo de UI construido mediante exploración dinámica. Luego analiza el flujo de datos y las relaciones de transición de UI entre

los casos de *test* obtenidos, y los combina (es decir, concatenando los casos de *test* en órdenes específicas) para generar casos de *test* finales. Además, funciona en un bucle de retroalimentación, es decir, generando casos de *test* adicionales cuando los *tests* anteriores alcanzan nuevos estados de la aplicación.

TimeMachine [95] es una herramienta de *testing* basada en estados. A diferencia de herramientas anteriores como Sapienz y Stoat, que evolucionan secuencias de eventos para maximizar la cobertura de código, TimeMachine, en cambio, evoluciona una población de estados que pueden ser capturados al descubrirse y reanudados cuando sea necesario para encontrar errores profundos. Durante la ejecución de los *tests*, TimeMachine toma capturas instantáneas de los estados interesantes y los agrega al corpus de estados. Esto le permite luego retroceder a estados más prometedores cuando la exploración no pueda alcanzar nuevos estados interesantes. Su singularidad radica en la capacidad de capturar y reanudar un estado de la aplicación específica para realizar casos de *test* adicionales mediante el uso de una máquina virtual donde es ejecutado el emulador Android.

Q-testing [96] es una herramienta de *testing* basada en aprendizaje por refuerzo. Utiliza una red neuronal entrenada para comparar las pantallas de la UI. Si una pantalla es similar a alguna de las UI exploradas previamente, el comparador otorgará una pequeña recompensa. De lo contrario, el comparador otorgará una gran recompensa. Estas recompensas se utilizan y se actualizan de forma iterativa para guiar los *tests* y cubrir más funcionalidades de las aplicaciones.

MATE [97] es una herramienta para generar secuencias de interacciones, basada en información de la UI. Aunque originalmente fue diseñada para encontrar problemas de accesibilidad (por ejemplo, una descripción de contenido faltante para un componente visible), recientemente se ha ampliado para estudiar diferentes algoritmos evolutivos para la generación de casos de *test* [98].

Todas las herramientas mencionadas arriba representan internamente las interacciones con la aplicación Android utilizando diferentes formatos. Por ejemplo, Sapienz representa las interacciones como secuencias de *acciones atómicas* (*click*, *click* largo, etc.) sobre coordenadas específicas $\langle x,y \rangle$ de la pantalla. Sin embargo, muchas de ellas utilizan la herramienta UIAutomator [22] para encontrar eventos de UI disponibles y ejecutarlos, y así representan las interacciones como secuencias de acciones sobre *widgets* en la UI (llamadas *acciones de widget*). Algunas de las herramientas en este grupo son: Stoat, Dynodroid, MATE, Ape, ComboDroid, TimeMachine y Q-testing. Nos referimos a ellas como herramientas de *testing* de Android *basadas en widgets*.

3.3. Creación de casos de test Espresso

Espresso [23] es un *framework* para escribir casos de *test* de UI para aplicaciones Android. Esta basado en *Instrumentation*, lo que permite a los desarrolladores escribir *tests* instrumentados que se ejecutan en dispositivos Android. Este *framework*, lanzado en octubre de 2013, cuenta con el respaldo oficial del ecosistema Android y forma parte del repositorio de *testing* AndroidX [99].

```java
@Test
public void clickButtonTest() {
    // Build a matcher to find a specific button in the UI.
    Matcher<View> matcher = allOf(
        withId(R.id.button),
        withParent(withId(R.id.layout))
    );
    // Build an interaction to perform actions/assertions
    // on the button.
    ViewInteraction viewInteraction = onView(matcher);
    // Build an action to click the button and execute it.
    ViewAction action = click();
    viewInteraction.perform(action);
    // Build an assertion to check the text of the button
    // and execute it.
    ViewAssertion assertion = matches(withText("Clicked"));
    viewInteraction.check(assertion);
}
```

Figura 3.2: Ejemplo de un caso de *test* Espresso. Este *test* construye un *View Matcher* de Espresso para seleccionar una vista que tiene, al mismo tiempo, un identificador de recurso "`R.id.button`" y un padre con identificador de recurso "`R.id.layout`". Luego hace *click* en la vista seleccionada y verifica que su texto sea "`Clicked`".

Espresso proporciona una API que permite a los desarrolladores simular de manera programática las interacciones del usuario con la AUT [100]. Esta API está organizada en tres categorías: ***View Matchers*** para identificar y seleccionar una vista dentro de la Jerarquía de Vistas actual, ***View Actions*** para realizar acciones en la vista seleccionada y ***View Assertions*** para verificar el estado de la vista seleccionada. El *framework* permite a los desarrolladores especificar qué actividad Android debe iniciarse al comienzo de cada *test* utilizando una anotación `@Rule`. Además, Espresso mejora la confiabilidad de los *tests* al ejecutar las acciones pendientes solo cuando la AUT está inactiva.

La Figura 3.2 muestra un ejemplo de caso de *test* en formato Espresso. El *test* selecciona la vista que cumple al mismo tiempo con el identificador de recurso "`R.id.button`" y un padre con el identificador de recurso "`R.id.layout`". Luego, hace *click* en la vista seleccionada y verifica que su texto sea "`Clicked`". Crear un caso de *test* Espresso implica entonces escribir:

- *View Matchers* que localicen correctamente las vistas objetivo. Estos *matchers* deben ser específicos. Si no se encuentra la vista o varias vistas cumplen los criterios en una pantalla determinada, la ejecución del *test* fallará. Por ejemplo, un *matcher* que localiza un `TextView` por su texto fallará si el texto no es único en la interfaz de usuario.
- *View Actions* adecuadas para las vistas seleccionadas. Por ejemplo, una acción

```
type= raw events
count= -1
speed= 1.0
start data >>
DispatchTrackball(-1,56979,2,2.0,3.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchPointer(43114,43246,2,621.3782,1169.5244,0.0,0.0,0,1.0,1.0,0,0)
DispatchTrackball(-1,56895,2,1.0,-1.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchTrackball(-1,44282,2,1.0,2.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchPointer(57453,57454,2,559.66693,638.5824,0.0,0.0,0,1.0,1.0,0,0)
DispatchTrackball(-1,44668,2,4.0,0.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchPointer(44124,44124,0,501.0,580.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchTrackball(-1,44172,2,-4.0,0.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchPointer(43251,43252,2,284.27853,1022.2573,0.0,0.0,0,1.0,1.0,0,0)
DispatchKey(57293,57293,0,19,0,0,-1,0)
DispatchPointer(57058,57058,0,669.0,886.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchPointer(43251,43253,2,293.40747,1060.9349,0.0,0.0,0,1.0,1.0,0,0)
DispatchKey(56844,56844,0,23,0,0,-1,0)
DispatchTrackball(-1,57331,2,-5.0,-4.0,0.0,0.0,0,1.0,1.0,0,0)
DispatchKey(44628,44628,1,203,0,0,-1,0)
DispatchKey(57450,57450,1,103,0,0,-1,0)
```

Figura 3.3: Ejemplo de caso de *test* generado por SAPIENZ.

de *click* fallará si se ejecuta en una vista que no se muestra en la pantalla.

- *View Assertions* que verifiquen propiedades apropiadas para las vistas seleccionadas. Por ejemplo, intentar verificar el texto mostrado de un `ImageView` provocará una excepción.

El soporte oficial de ESPRESSO, junto con la capacidad del *framework* de proporcionar *tests* de interfaz de usuario concisos y confiables, lo ha convertido en el *framework* de *tests* más popular entre los desarrolladores [25], con un aumento constante en su adopción en los últimos años. Por otro lado, UIAUTOMATOR, ROBOTIUM y APPIUM son utilizados por muy pocos proyectos, mientras que ANDROIDVIEWCLIENT, CALABASH, MONKEYRUNNER y PYTHONUIAUTOMATOR no son utilizados en absoluto.

3.4. Análisis del output de las herramientas de testing existentes

Para entender si las herramientas de *testing* automático de ANDROID también se ajustan a estas tendencias, examinamos 101 artículos de la literatura de *testing* de ANDROID y categorizamos cada uno según el tipo de salida proporcionada.

El resultado de este análisis fue que solo unas pocas herramientas se enfocan en el *framework* ESPRESSO como formato de salida. Una gran parte de ellas no proporcionan una salida ejecutable, como informes de fallos. La mayoría de las herramientas que generan casos de *test* ejecutables tienden a utilizar un formato personalizado, porque utilizan un motor personalizado para ejecutarlos, o el *framework* ROBOTIUM, que está deprecado y ya no recibe mantenimiento. La lista completa de los artículos analizados se puede encontrar en la Tabla 3.1, así como un resumen visual de la categorización realizada.

MONKEY [84], considerada como una de las herramientas más populares en la práctica, solo informa sobre excepciones en tiempo de ejecución no capturadas (i.e.,

Tabla 3.1: Herramientas de *testing* automático para ANDROID, categorizadas por el tipo de salida.

Tipo de salida	Herramientas y artículos	Cantidad
Salida personalizada	SAPIENZ [86], CRASHSCOPE [101], MAMBA [102], SSDA [103], GAT [104], ATT [105], VALERA [106], Ermuth et al. [107], IntentDroid [108], iMPAcT [109], CRAXDroid [110], APSET [111], PBGT [112], Sasnauskas et al. [113], Appstrument [114], Smartdroid [115], AimDroid [116], Polariz [87], ACTEve [117], JPF-ANDROID [118], SWIFTHAND [119], DRUN [120], MonkeyLab [121], CuriousDroid [122], Mirzaei et al. [123]	25
***Tests* en formato Robotium**	EVODROID [124], GUIRIPPER [125], A^{3}E [126], ORBIT [127], ALARic [128], QUANTUM [12], TrimDroid [129], MoTiF [130], Zhang et al. [131], Farto et al. [132], Guo et al. [133], Yang et al. [134], LEAKDROID [135], Mahmood et al. [136], LAND [137], ATG [138], AMOGA [139], SIG-Droid [140], AndroidRipper [125], AGRippin [141], ACRT [142], A^{2}T^2 [143]	22
Salida no ejecutable	MONKEY [84], APE [92], STOAT [90], DYNODROID [85], HUMANOID [93], MATE [97], Xdroid [144], MobiCoMonkey [145], Synthesi-SE [146], CrawlDroid [147], AppSPIN [148], Kraken [149], ParaAim [150], IntentFuzzer [151], Pretect [152], DiagDroid [153], AppAudit [154], DroidBot [155], Data Loss Detector [156], DroidFuzzer [157], TIMEMACHINE [95], VANARSena [158]	22
***Tests* en formato UIAutomator**	PUMA [159], WCTester [160], DroidMate [161] Sentinel [162], AppCheck [163], Mimic [164], Q-TESTING [96], Mobolic [165], T+ [166], Fusion [167],	10
***Tests* en formato Espresso**	Rohella et al. [168], COBWEB [169], RacerDroid [170], ETR [171], Barista [172], Coppola et al. [173], DiffDroid [174], AppTestMigrator [175]	8
***Tests* en formato JUnit**	MobiGUITAR [176], Shahriar et al. [177], Avancini et al. [178], Franke et al. [179], CTGen [180], Hu et al. [7]	6
***Tests* en formato Appium**	Autodroid [181], Ali et al. [182], ACAT [183], JazzDroid [184]	4
***Tests* en formato MonkeyRunner**	EHBDroid [185], Banerjee et al. [186], AppDoctor [187]	3
***Tests* en formato Robolectric**	CATE [188], RoboLIFT [189]	2
***Tests* en formato Calabash**	Griebe et al. [190]	1

crashes) durante una exploración aleatoria. Además, MONKEY no permite a los usuarios reproducir las secuencias de eventos generadas (es decir, no produce casos de *test*), lo cual puede ser fundamental para comprender la causa de un fallo.

De manera similar, otras herramientas para realizar *testing* automático de aplicaciones ANDROID (como DYNODROID [85], SAPIENZ [86], STOAT [90], MATE [97], etc.) no generan casos de *test* en un formato legible y reproducible. Por ejemplo, SAPIENZ [86] produce una secuencia de acciones "atómicas" destinadas a ser utilizadas por la misma herramienta (es decir, para volver a ejecutar casos fallidos, ilustrada en la Figura 3.3), MATE [97] genera un informe de accesibilidad, mientras que DYNODROID [85] y STOAT [90] solo proporcionan informes de fallos. Incluso herramientas recientes como APE [92], HUMANOID [93], TIMEMACHINE [95], Q-TESTING [96] y COMBODROID [94] solo generan informes de cobertura y fallos.

En resumen, independientemente de la estrategia de generación de casos de *test* subyacente, una suposición común implícita de muchas herramientas existentes es que los desarrolladores solo se preocupan por encontrar *crashes* en la aplicación bajo *test*, y no por generar casos de *test* reproducibles. A continuación mencionamos las herramientas de *testing* para ANDROID existentes en la literatura que sí generan casos de *test* ESPRESSO.

Espresso Test Recorder (ETR) [171] proporciona a los desarrolladores una herramienta de *grabación y reproducción* que facilita la tarea de escribir *tests* de ANDROID en formato ESPRESSO. ETR está integrado directamente en ANDROID STUDIO, el IDE oficial para el desarrollo de ANDROID. Sin embargo, ETR no es un enfoque automático, ya que el desarrollador es responsable de proporcionar las acciones de la UI que se deben realizar en el *test*. En otras palabras, el proceso de ETR requiere que los desarrolladores interactúen manualmente con la aplicación bajo *test*. Estas interacciones se capturan y procesan para generar el código correspondiente. A través de este proceso, ETR ayuda a los desarrolladores a generar de manera interactiva una *test* ejecutable en formato ESPRESSO.

La idea clave de ETR es utilizar el *debugger* de JAVA para establecer puntos de interrupción (*breakpoints*, en inglés) dentro del *bytecode* del sistema operativo ANDROID en tiempo de ejecución. Esto permite que la herramienta capture fielmente la amplia gama de eventos que pueden ocurrir en la UI (por ejemplo, *click*, *long click*, menús, etc.). Cuando se alcanza un punto de interrupción, los datos relevantes se almacenan en una *representación intermedia* y la aplicación continúa ejecutándose normalmente. Cada acción consta entonces de un tipo (*click*, *long click*, etc.), parámetros (por ejemplo, el texto al escribir en un campo) y el *widget* sobre el cual se realizó la acción (por ejemplo, un botón con una cierta identificación).

Dado que ETR requiere intervención manual constante, el uso de esta herramienta en la práctica está limitado por la cantidad de tiempo y esfuerzo que los desarrolladores estén dispuestos a invertir en interactuar con la herramienta para grabar *tests*. Además, aunque ETR podría escuchar las interacciones de la UI que se produzcan automáticamente (por ejemplo, mediante una herramienta), solo puede iniciarse y controlarse desde la IDE de ANDROID STUDIO. En otras palabras, ETR no puede ejecutarse como una herramienta independiente, por lo que la integración con

otras herramientas para la generación automatizada de entradas de Android está restringida.

Barista [172] es una herramienta de grabación y reproducción similar a ETR que ofrece un mejor soporte para definir oráculos y *View Matchers*. También mejora el soporte para ejecutar los casos de *test* generados en múltiples dispositivos y versiones de Android. Al igual que ETR, Barista está integrado en Android Studio y requiere interacción manual con la aplicación bajo *test* (AUT).

DiffDroid [174] es una herramienta para encontrar automáticamente inconsistencias en aplicaciones Android al ser ejecutadas en distintas versiones del sistema operativo y tamaños de pantalla. Extiende Monkey para generar secuencias de interacciones y codificarlas en un formato interno de la herramienta, mientras que guarda simultáneamente las Jerarquías de Vista de las pantallas visitadas durante la exploración.

AppTestMigrator [175] es una herramienta para migrar casos de *test* en formato Espresso entre aplicaciones de la misma categoría (por ejemplo, aplicaciones bancarias). Aprovecha las similitudes entre las UI para migrar automáticamente los *tests* existentes de una aplicación a otra.

Coppola et al. [173] presentan un enfoque para traducir scripts de *test* de herramientas basadas en interfaces visuales a casos de *test* Espresso. Rohella et al. [168] mencionan brevemente que su herramienta produce *tests* Espresso, sin proporcionar más detalles. COBWEB [169] utiliza Robolectric [24] como representación interna de los *tests* y los transforma en *tests* Espresso al final. No proporcionan detalles específicos sobre cómo se realiza esta transformación. RacerDroid [170] modifica el *framework* de Espresso para controlar el envío de eventos. No está claro si los casos de *test* Espresso generados por RacerDroid pueden ejecutarse fuera del *framework* modificado.

4

Evaluación empírica de algoritmos basados en búsqueda

En este capítulo presentamos un estudio empírico para analizar la efectividad de distintos algoritmos basados en búsqueda (*search-based algorithms* [191], en inglés) para la generación de casos de *test* para aplicaciones ANDROID. Los resultados de este estudio guían luego nuestra elección de algoritmo para la generación de casos de *test* en formato ESPRESSO en el resto de la tesis. El estudio se realiza utilizando como base la herramienta SAPIENZ [86] para generación automática de casos de *test* para aplicaciones ANDROID. Las conclusiones y aprendizajes que sacamos de este estudio son presentados en la Sección 4.6. Este capítulo fue publicado como artículo de conferencia [192], y luego extendido a artículo de revista [193].

4.1. Introducción

SAPIENZ [86] es una herramienta de generación automática de casos de *test* para aplicaciones ANDROID que ha demostrado superar a varias herramientas de vanguardia como DYNODROID [85] y MONKEY [84]. En los últimos años, una versión modificada de la herramienta se ha implementado en la compañía de software Meta [89] (previamente, Facebook). Esencialmente, el enfoque de SAPIENZ presentado en Mao et al. [86] se distingue de otras herramientas de testing ANDROID anteriores debido a estas dos características:

i) Un algoritmo evolutivo multiobjetivo (NSGA-II [33]) que genera secuencias de eventos, maximizando simultáneamente la cobertura de sentencias y la detección de fallas, mientras minimiza la longitud de los casos de *test*.

ii) La representación de casos de *test* como secuencias de acciones *atómicas* y *motifs*.

Donde una acción *atómica* es un evento en la UI que no se puede descomponer más (por ejemplo, presionar una tecla, tocar la pantalla en una determinada coordenada, etc.). Por otro lado, las acciones *motif* son "eventos" compuestos (es decir, una secuencia de acciones *atómicas*) que representan un patrón de uso en la aplicación.

Dado que estas características (es decir, el algoritmo NSGA-II combinado con acciones *motif*) fueron presentadas simultáneamente, nos gustaría estudiar el impacto de cada una de ellas por separado. Además, Mao et al. [86] solo realizaron

comparaciones entre su herramienta y otras. Este tipo de comparaciones no son deseables ya que puede haber factores derivados de detalles de implementación que alteren los resultados.

En particular, nos interesa comparar diferentes elecciones de algoritmos evolutivos para la generación de casos de *test* ANDROID. Sell et al. [98] presentaron un estudio comparando diferentes algoritmos para la generación de casos de *test* ANDROID, pero estos algoritmos se evaluaron en la herramienta de testing MATE [97]. Entre otras diferencias, MATE difiere de SAPIENZ en que utiliza una representación basada en *widgets* de los individuos. Los *widgets* son componentes interactivos en una interfaz de usuario ANDROID (como botones, campos de texto, etc.). En cambio, los individuos de SAPIENZ se basan en secuencias de acciones que no dependen de *widgets* (es decir, las acciones *atómicas* y *motif* mencionadas anteriormente solo utilizan coordenadas de pantalla). Esta diferencia afecta cómo se realizan los operadores evolutivos (como el cruce y la mutación). Por lo tanto, nos gustaría estudiar cómo pueden comportarse diferentes algoritmos evolutivos al implementarlos en la herramienta SAPIENZ. También se ha demostrado que (al menos para la generación de casos de *test* unitarios), debido a funciones de *fitness* que generan espacios de búsqueda planos y problemas de búsqueda a menudo simples, *Random Search* [194] (*Búsqueda Aleatoria*) puede tener un rendimiento tan bueno como los algoritmos evolutivos e incluso superarlos en ocasiones [195]. Por esta razón, también nos gustaría estudiar la efectividad del algoritmo *Random Search* para la generación de *tests* ANDROID.

Luego, en este capítulo, nuestro objetivo es obtener más información sobre los efectos de las características principales de SAPIENZ: la elección del algoritmo multiobjetivo NSGA-II y la representación de los individuos utilizando acciones *motif*. Específicamente, las contribuciones de este capítulo son las siguientes:

- **Diseño del experimento:** Presentamos un estudio empírico que compara la eficacia en términos de cobertura de sentencias y detección de fallas de 9 algoritmos diferentes para la generación de casos de *test* ANDROID (*Random Search*, *Random Search* con acciones *motif*, *Standard GA*, *Monotonic GA*, *Steady state GA*, $(\mu + \lambda)$ EA, (μ, λ) EA, NSGA-II y NSGA-II con acciones *motif*) utilizando un total de 38 sujetos experimentales (8 aplicaciones de código abierto y 30 aplicaciones populares de código cerrado). Este estudio (Sección 4.4) involucra algoritmos con y sin acciones *motif*, así como un enfoque de *Random Search* que servirá como punto de referencia para la comparación. El tiempo total de ejecución fue de 202 días.
- **Resultados del experimento:** Presentamos los resultados de nuestro estudio empírico (Secciones 4.4.2 a 4.4.4), que involucran un total de 2430 puntos de datos.

Nuestro estudio empírico arroja los siguientes hallazgos:

- Tanto NSGA-II como *Random Search* mejoran su cobertura cuando los casos de *test* incluyen acciones *motif*.
- Entre todos los algoritmos evolutivos considerados en nuestro estudio, NSGA-II

es el que logra la mayor cobertura de sentencias. Sorprendentemente, NSGA-II no se distingue con significancia estadística de *Random Search*.

- Tanto NSGA-II como *Random Search* mejoran *marginalmente* la detección de fallas cuando los casos de *test* incluyen acciones *motif*. Sin embargo, esta diferencia no es estadísticamente significativa.
- Ya sea que se utilicen acciones *motif* o no, NSGA-II y *Random Search* detectan un número similar de fallas.

En resumen, nuestro experimento proporciona evidencia de que las mejoras en el rendimiento de SAPIENZ se deben más a la representación de los casos de *test* que incluyen acciones *motif* que al uso de un enfoque evolutivo específico.

4.2. Algoritmos basados en búsqueda

Los **algoritmos basados en búsqueda** (*search-based algorithms* [191], en inglés), también conocidos como algoritmos de optimización basados en búsqueda o algoritmos de búsqueda heurística, son técnicas de *inteligencia artificial* que utilizan estrategias de búsqueda para encontrar soluciones óptimas o cercanas a óptimas para problemas complejos. Estos algoritmos se basan en el principio de explorar sistemáticamente un espacio de búsqueda en busca de soluciones que cumplan ciertos criterios de calidad.

Los algoritmos basados en búsqueda comienzan desde un punto inicial y, mediante la aplicación de operadores de búsqueda, exploran iterativamente el espacio de búsqueda en busca de soluciones mejores. Los operadores de búsqueda modifican las soluciones actuales para generar nuevas soluciones que puedan ser evaluadas y comparadas. Otra característica clave de estos algoritmos es el uso de una función de evaluación, también conocida como función de *fitness* o función objetivo. Esta función asigna un valor numérico a cada solución candidata en función de su calidad o ajuste con respecto al problema en cuestión. El objetivo del algoritmo de búsqueda es encontrar la solución con el mejor valor de la función de *fitness*. En el contexto de la generación de *test suites*, las funciones de *fitness* se basan típicamente en criterios de cobertura estructural, como la cobertura de sentencias o la cobertura de ramas (*branches*).

Existen varios tipos de algoritmos basados en búsqueda, incluyendo:

- Búsqueda ciega: Estos algoritmos exploran el espacio de búsqueda sin utilizar información específica sobre el problema. Ejemplos de algoritmos de búsqueda ciega incluyen la búsqueda en anchura (*Breadth-First Search* o BFS, en inglés), la búsqueda en profundidad (*Depth-First Search* o DFS, en inglés) y la búsqueda de costo uniforme.
- Búsqueda heurística: Estos algoritmos utilizan información adicional o conocimiento experto sobre el problema para guiar la búsqueda hacia soluciones prometedoras. Algunos ejemplos populares son el algoritmo A* y la búsqueda *greedy*.

Algoritmo 1: *Random Search*

Input : Condición de parada C, Función de *fitness* δ, Tamaño de población p_s, Función de selección s_f
Output : Población de individuos optimizados P

```
1 P ⟵ {}
2 while ¬C do
3 |   N_P ⟵ GenerateRandomPopulation(p_s)
4 |   PerformFitnessEvaluation(δ, N_P)
5 |   P ⟵ Selection(s_f, P ∪ N_P)
6 return P
```

- Algoritmos genéticos: Estos algoritmos se inspiran en la evolución biológica y utilizan conceptos como la selección natural, la reproducción y la mutación para buscar soluciones óptimas. Los algoritmos genéticos son especialmente útiles en problemas de optimización combinatoria.
- Algoritmos de enjambre: Estos algoritmos están inspirados en el comportamiento colectivo de los enjambres de animales, como las colonias de hormigas o las bandadas de aves. Los individuos en el enjambre colaboran y se comunican entre sí para buscar soluciones óptimas. Ejemplos de algoritmos de enjambre incluyen el algoritmo de optimización por colonia de hormigas (ACO) y el algoritmo de enjambre de partículas (PSO).

Los algoritmos basados en búsqueda se aplican en una amplia gama de problemas, como la optimización de funciones matemáticas, la planificación de rutas, la asignación de recursos, el diseño de circuitos, la programación de horarios y muchos otros. Estos algoritmos son especialmente útiles cuando los problemas son demasiado complejos o el espacio de búsqueda es demasiado grande para ser explorado exhaustivamente.

4.2.1. Random Search

Random Search [194] (ver Algoritmo 1) es uno de los algoritmos basados en búsqueda más simples. Consiste en muestrear repetidamente candidatos del espacio de búsqueda. Una vez que se agota el presupuesto, se devuelve el individuo muestreado más apto, es decir, con mejor resultado para la función de *fitness*. Debido a su simplicidad, es muy útil como base para estudiar la efectividad de cualquier otra técnica propuesta. Para la generación de *tests* unitarios, se ha demostrado que el desempeño de *Random Search* a menudo es tan efectivo como el de otros algoritmos evolutivos, e incluso puede superarlos si el programa bajo análisis es lo suficientemente simple [195].

4.2.2. Algoritmos genéticos

Los **algoritmos genéticos** (GA) son técnicas de optimización y búsqueda inspiradas en la teoría de la evolución biológica y la genética. La idea detrás de

Algoritmo 2: *Standard GA*

Input : Condición de parada C, Función de *fitness* δ, Tamaño de población p_s, Función de selección s_f, Función de cruce c_f, Probabilidad de cruce c_p, Función de mutación m_f, Probabilidad de mutación m_p

Output : Población de individuos optimizados P

```
1  P ⟵ GenerateRandomPopulation(p_s)
2  PerformFitnessEvaluation(δ, P)
3  while ¬C do
4      N_P ⟵ {} ∪ Elitism(P)
5      while |N_P| < p_s do
6          p_1, p_2 ⟵ Selection(s_f, P)
7          o_1, o_2 ⟵ Crossover(c_f, c_p, p_1, p_2)
8          Mutation(m_f, m_p, o_1)
9          Mutation(m_f, m_p, o_2)
10         N_P ⟵ N_P ∪ {o_1, o_2}
11     P ⟵ N_P
12     PerformFitnessEvaluation(δ, P)
13 return P
```

los algoritmos genéticos es simular el proceso de selección natural y evolución para encontrar soluciones óptimas o cercanas a óptimas. Estos algoritmos trabajan con una población inicial de soluciones candidatas, que se representa como una colección de cromosomas o cadenas de genes. En este contexto, un cromosoma es una representación de una solución potencial al problema, mientras que un gen es una parte de un cromosoma que codifica una característica específica de la solución.

A lo largo de las generaciones, los algoritmos genéticos aplican operadores genéticos como selección, cruce (*crossover*) y mutación para generar nuevas soluciones y mejorar gradualmente la calidad de la población. Estos operadores se inspiran en los procesos biológicos de la evolución:

- Selección: Los cromosomas más aptos, es decir, aquellos que tienen un mejor valor de *fitness*, tienen más probabilidades de ser seleccionados para reproducirse y transmitir sus características a la siguiente generación.
- Cruce (*crossover*): Se seleccionan dos cromosomas padres y se intercambian segmentos de sus genes para generar uno o más descendientes. El cruce permite combinar características prometedoras de diferentes soluciones en la población.
- Mutación: Se introduce un cambio aleatorio en uno o más genes de un cromosoma para explorar nuevas soluciones y evitar la convergencia prematura hacia un óptimo local. La mutación permite la introducción de variabilidad en la población.

Después de aplicar los operadores genéticos, se evalúa la *fitness* de las nuevas soluciones generadas y se selecciona la próxima generación de acuerdo con su rendi-

Algoritmo 3: *Monotonic GA*

Input : Condición de parada C, Función de *fitness* δ, Tamaño de población p_s, Función de selección s_f, Función de cruce c_f, Probabilidad de cruce c_p, Función de mutación m_f, Probabilidad de mutación m_p

Output : Población de individuos optimizados P

```
1  P ⟵ GenerateRandomPopulation(p_s)
2  PerformFitnessEvaluation(δ, P)
3  while ¬C do
4      N_P ⟵ {} ∪ Elitism(P)
5      while |N_P| < p_s do
6          p_1, p_2 ⟵ Selection(s_f, P)
7          o_1, o_2 ⟵ Crossover(c_f, c_p, p_1, p_2)
8          Mutation(m_f, m_p, o_1)
9          Mutation(m_f, m_p, o_2)
10         PerformFitnessEvaluation(δ, o_1)
11         PerformFitnessEvaluation(δ, o_2)
12         if Best(o_1, o_2) is better than Best(p_1, p_2) do
13             N_P ⟵ N_P ∪ {o_1, o_2}
14         else
15             N_P ⟵ N_P ∪ {p_1, p_2}
16     P ⟵ N_P
17 return P
```

miento. Este proceso se repite durante un número determinado de generaciones o hasta que se alcance un criterio de terminación, como la convergencia de la *fitness* o el agotamiento del tiempo de ejecución. Es importante notar que los algoritmos genéticos típicamente inician el proceso con una población inicial aleatoria. Sin embargo, en algunos casos, se puede utilizar un enfoque de inicialización más inteligente para generar una población inicial de soluciones de alta calidad.

En este estudio utilizaremos el Algoritmo Genético Estándar (*Standard GA*, en inglés) descrito por Campos et al. [196] (ver Algoritmo 2). El algoritmo comienza generando una población inicial aleatoria de tamaño p_s. Luego, la población se evalúa utilizando una función de *fitness* δ. Cada iteración (es decir, "generación") del algoritmo consiste en construir una nueva población y evaluar cada individuo en ella. La nueva población se crea eligiendo repetidamente un par de individuos de la población actual y luego recombinándolos en dos nuevos individuos. La selección se realiza mediante una estrategia s_f, como selección basada en *ranking*, elitismo o torneo. La recombinación se realiza mediante una función de cruce c_f, como de un solo punto o de múltiples puntos, con una probabilidad c_p. Antes de insertar la descendencia en la nueva población, se aplica la mutación de forma independiente en ambos, con una probabilidad m_p. Esta probabilidad suele ser $\frac{1}{n}$, donde n es el número de genes en un cromosoma. Esto asegura que, en promedio, al menos un gen

Algoritmo 4: *Stady-state GA*

Input : Condición de parada C, Función de *fitness* δ, Tamaño de población p_s, Función de selección s_f, Función de cruce c_f, Probabilidad de cruce c_p, Función de mutación m_f, Probabilidad de mutación m_p

Output : Población de individuos optimizados P

```
1  P ⟵ GenerateRandomPopulation(p_s)
2  PerformFitnessEvaluation(δ, P)
3  while ¬C do
4      p_1, p_2 ⟵ Selection(s_f, P)
5      o_1, o_2 ⟵ Crossover(c_f, c_p, p_1, p_2)
6      Mutation(m_f, m_p, o_1)
7      Mutation(m_f, m_p, o_2)
8      PerformFitnessEvaluation(δ, o_1)
9      PerformFitnessEvaluation(δ, o_2)
10     if Best(o_1, o_2) is better than Best(p_1, p_2) do
11         P ⟵ P \ {p_1, p_2} ∪ {o_1, o_2}
12     else
13         P ⟵ P \ {o_1, o_2} ∪ {p_1, p_2}
14 return P
```

mute en cada descendencia, manteniendo la diversidad en la población.

Existen muchas variantes de *Standard GA* que buscan mejorar la efectividad. En particular, consideramos las siguientes alternativas:

- **Monotonic GA:** Similar a *Standard GA*, pero solo incluye ya sea la mejor descendencia o el mejor padre en la siguiente población (ver Algoritmo 3). Esto asegura que el valor de *fitness* alcanzado no disminuya a medida que evoluciona la población.
- **Steady-state GA:** Este algoritmo utiliza la misma estrategia de reemplazo que el *Monotonic GA*, pero en lugar de crear una nueva población en cada generación, la descendencia reemplaza a los padres en la población actual inmediatamente después de que son mutados y evaluados (ver Algoritmo 4).

4.2.3. Algoritmos evolutivos

Los términos "algoritmos genéticos" y "algoritmos evolutivos" a menudo se utilizan indistintamente porque ambos se basan en los principios de la evolución biológica y la selección natural. Sin embargo, hay una distinción sutil entre ellos. Los algoritmos genéticos son una subclase específica de los algoritmos evolutivos que se enfocan en la representación de soluciones como cromosomas y utilizan operadores genéticos como selección, cruce y mutación. Por otro lado, los algoritmos evolutivos son un término más amplio que incluye otras variantes como la programación evolutiva, las estrategias de evolución diferencial y la programación genética. La diferencia

Algoritmo 5: Algoritmo evolutivo $(\mu + \lambda)$

Input : Condición de parada C, Función de aptitud δ, Tamaño de población μ, Tamaño de descendencia λ, Función de selección s_f, Función de mutación m_f, Probabilidad de mutación m_p

Output : Población de individuos optimizados P

```
1  P ⟵ GenerateRandomPopulation(μ)
2  PerformFitnessEvaluation(δ, P)
3  while ¬C do
4      O ⟵ {}
5      forall p ∈ P do
6          for i ⟵ 1, λ/μ do
7              o ⟵ Mutation(m_f, m_p, p)
8              O ⟵ O ∪ {o}
9      PerformFitnessEvaluation(δ, O)
10     P ⟵ select best μ individuals from P ∪ O
11 return P
```

clave radica en la representación de soluciones y los operadores utilizados, ya que los algoritmos genéticos se centran en cromosomas y operadores genéticos específicos, mientras que los algoritmos evolutivos pueden usar diferentes estructuras y operadores evolutivos.

El Algoritmo Evolutivo $(\mu + \lambda)$ [197] es un algoritmo basado en mutación [198] (ver Algoritmo 5). En este caso, μ representa el tamaño de los padres y λ el tamaño de la descendencia. Para cada individuo en la población actual, se aplica la mutación de manera independiente en cada gen con una probabilidad de $\frac{1}{n}$. Después de la mutación, se seleccionan los mejores μ individuos de una combinación de padres y descendencia para formar la nueva población. Por lo tanto, los padres solo serán reemplazados si se encuentra una descendencia mejor.

Una variante de esto es el Algoritmo Evolutivo (μ, λ), donde los padres se descartan y los nuevos μ individuos se seleccionan solo de la descendencia (ver Algoritmo 6).

4.2.4. Representación de individuos

La representación de los individuos en Sapienz sigue los principios de generación *Whole Test Suite* (WTS) [199]. WTS evoluciona *test suites* completos para un sólo criterio de cobertura al mismo tiempo (por ejemplo, sentencias en el programa que se está analizando). En WTS, cada individuo es un *test suite* (es decir, un conjunto de casos de *test*). Cada caso de *test* puede ser visto como un "cromosoma" del individuo. Luego, cada uno de estos cromosomas se representará como una secuencia de genes (eventos de *test*).

En nuestro contexto, y para hacer comparaciones justas entre Sapienz y otros algoritmos, estos genes consistirán en una combinación de genes *atómicos* y *motif*.

Algoritmo 6: Algoritmo evolutivo (μ, λ)

Input : Condición de parada C, Función de aptitud δ, Tamaño de población μ, Tamaño de descendencia λ, Función de selección s_f, Función de mutación m_f, Probabilidad de mutación m_p

Output : Población de individuos optimizados P

```
1  P ⟵ GenerateRandomPopulation(μ)
2  PerformFitnessEvaluation(δ, P)
3  while ¬C do
4      O ⟵ {}
5      forall p ∈ P do
6          for i ⟵ 1, λ/μ do
7              o ⟵ Mutation(m_f, m_p, p)
8              O ⟵ O ∪ {o}
9      PerformFitnessEvaluation(δ, O)
10     P ⟵ select best μ individuals from O
11 return P
```

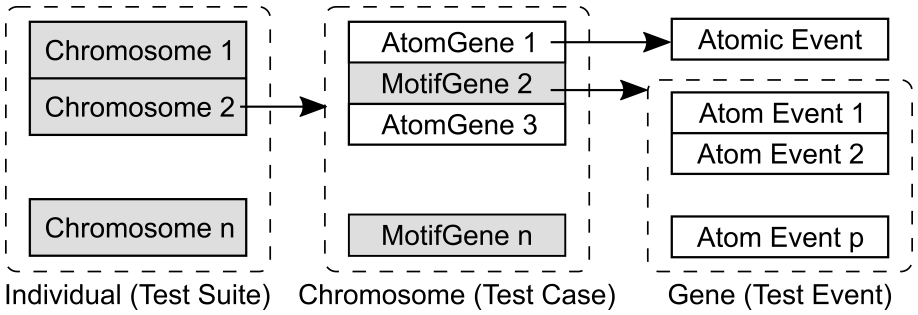


Figura 4.1: Representación de individuos en algoritmos evolutivos en el trabajo de Mao et al. [86]

Según la definición de Mao et al. [86], un **gen atómico** es un evento que no puede ser descompuesto aún más (por ejemplo, presionar una tecla, tocar la pantalla en una determinada coordenada, etc.), mientras que un **gen motif** se interpreta como una secuencia de eventos *atómicos* $\langle e_1, \ldots, e_p \rangle$. Esta representación se muestra en la Figura 4.1.

Cada gen *motif* definido representa un patrón de uso en la aplicación. Estos patrones siguen el comportamiento común del usuario, como llenar todos los campos de texto en la pantalla actual y luego hacer *click* en un botón. Por lo tanto, las acciones *motif* se basan en la información de la UI disponible en la pantalla actual.

4.2.5. Algoritmos multiobjetivo

En muchos casos, es deseable optimizar los casos de *test* generados hacia múltiples objetivos de optimización (posiblemente conflictivos). Un mecanismo sencillo

Algoritmo 7: Algoritmo evolutivo multiobjetivo NSGA-II

Input : Condición de parada C, Función de *fitness* multiobjetivo δ, Función de selección s_f, Función de mutación m_f, Probabilidad de mutación m_p

Output : Población de individuos optimizados P

```
1  P ⟵ GenerateRandomPopulation(μ)
2  PerformFitnessEvaluation(δ, P)
3  while ¬C do
4      O ⟵ {}
5      forall p ∈ P do
6          o ⟵ Mutation(m_f, m_p, p)
7          O ⟵ O ∪ {o}
8      F ⟵ sortNonDominated(P ∪ O, |P|)
9      P' ⟵ {}
10     for each front F in F do
11         if |P'| ≥ |P| do
12             break
13         calculate crowding distance for F
14         for each individual f in F do
15             P' ⟵ P' ∪ {f}
16     P' ⟵ sorted(P', ≺_c)
17     P ⟵ P'[0 : |P|]
18 return P
```

para combinar múltiples objetivos de cobertura es a través de una combinación lineal ponderada [200]. Sin embargo, una combinación lineal requiere objetivos de optimización no conflictivos, como alta cobertura de sentencias y alto puntaje de mutación [201]. Por ejemplo, un desarrollador desearía obtener un *test suite* con mayor cobertura de sentencias y menor longitud de casos de *test*, con tal de facilitar el *debugging* y mantenimiento de dichos *tests*. Es fácil ver que estos objetivos son conflictivos: aumentar la longitud de los *tests* puede llevar a una mayor cobertura de sentencias, mientras que reducir la longitud de los *tests* puede disminuir la cobertura. Los algoritmos evolutivos multiobjetivo se centran especialmente en abordar varios objetivos (posiblemente conflictivos) simultáneamente.

4.3. El enfoque de Sapienz

Sapienz [86] es una técnica de generación de *tests* multiobjetivo para Android que tiene como objetivo maximizar la cobertura de código y la detección de fallas, al mismo tiempo que minimiza la longitud de las secuencias de *tests* que revelan fallos.

Para lidiar con los objetivos conflictivos (es decir, maximizar la cobertura mien-

tras se minimiza la longitud de los *tests*), SAPIENZ utiliza el algoritmo evolutivo multiobjetivo NSGA-II [33] (ver Algoritmo 7), que es ampliamente utilizado en la literatura de ingeniería de software basada en búsqueda (*Search-based software engineering*, SBSE, en inglés) [202]. Este algoritmo utiliza un ordenamiento rápido no dominado con un operador de selección que crea un grupo de apareamiento al combinar las poblaciones de padres e hijos, y selecciona las mejores N soluciones según la *fitness* y la dispersión. Durante el proceso de selección, todos los objetivos se combinan utilizando un enfoque basado en búsqueda óptima de Pareto [203]. Formalmente, se dice que un individuo x está *dominado* por otro individuo y ($x \prec y$) según una función de *fitness* si y solo si x es parcialmente menor que y:

$$\forall\ i = 1, \ldots, n,\ f_i(x) \leq f_i(y) \quad \wedge \quad \exists\ i = 1, \ldots, n : f_i(x) < f_i(y)$$

Entonces, un conjunto de Pareto óptimo consiste en todos los individuos *no dominados* (pertenecientes a todas las soluciones S):

$$P^* \triangleq \{x \in S \mid \nexists\ y \in S, x \prec y\}$$

Por lo tanto, una solución al problema de optimización multiobjetivo no es un único punto en el espacio de búsqueda (como en la generación de WTS), sino una familia de puntos. En la práctica, esto significa que los individuos con secuencias de *tests* más largas no se descartan cuando son los únicos que encuentran fallos, ni cuando son necesarios para lograr una mayor cobertura de código. Así, mediante el uso de la optimalidad de Pareto, SAPIENZ reemplaza progresivamente las secuencias más largas con secuencias de *tests* más cortas cuando son igualmente buenas.

En SAPIENZ, un individuo es un *test suite*. Como se mencionó anteriormente, cada individuo consta de varios cromosomas (casos de *test*) y cada cromosoma contiene múltiples genes (eventos de *test*), que consisten en una combinación aleatoria de genes *atómicos* y genes *motif*. Las secuencias de eventos en los casos de *test* son generadas y ejecutadas por un componente especial llamado MOTIFCORE. Este componente combina *tests* aleatorios y exploración sistemática, correspondiendo a los dos tipos de genes que SAPIENZ admite: genes *atómicos* y genes *motif*. El comportamiento de cada gen *motif* depende de la información de la UI disponible en el momento de su ejecución. Estos genes se utilizan para realizar patrones comunes de interacción del usuario durante la exploración, como completar todos los campos de texto en la pantalla y hacer *click* en los botones ejecutables. El algoritmo completo de MOTIFCORE se muestra en el Algoritmo 8.

4.4. Estudio empírico

En este capítulo de la tesis nos interesa investigar cómo el algoritmo evolutivo y la representación de individuos en SAPIENZ afectan al rendimiento general de la generación de *tests*. Por lo tanto, planteamos las siguientes preguntas de investigación (*Research Questions*, RQ):

RQ1: (Representación) *¿Cuál es la contribución de los genes* motif *en la* ***cobertura*** *lograda por SAPIENZ?*

Algoritmo 8: Simplificación de la estrategia de exploración MotifCore presentada por Mao et al. [86]

```
Input  : Aplicación bajo test A, secuencia de tests T = ⟨E_1, E_2, ..., E_n⟩,
         strings estáticos S, Modelo de UI M
Output : Modelo de UI actualizado M
1  for EACH EVENT E IN T do
2      if E IS AN ATOMIC EVENT do
3          EXECUTE ATOMIC EVENT E AND UPDATE M
4      if E IS A MOTIF EVENT do
5          currentActivity ⟵ EXTRACTCURRENTACTIVITY(A)
6          uiElementSet ⟵ EXTRACTUIELEMENT(currentActivity)
7          for EACH ELEMENT w IN uiElementSet do
8              if w IS EditText widget do
9                  SEED STRING s ∈ S INTO w
10             else
11                 EXERCISE w ACCORDING TO MOTIF PATTERNS IN E
12             UPDATE M
13 return M
```

RQ2: (Algoritmo) *¿Cuál es la contribución del algoritmo evolutivo NSGA-II en la* ***cobertura*** *lograda por* Sapienz*?*

RQ3: (Representación) *¿Cuál es la contribución de los genes* motif *en la* ***detección de fallas*** *lograda por* Sapienz*?*

RQ4: (Algoritmo) *¿Cuál es la contribución del algoritmo evolutivo NSGA-II en la* ***detección de fallas*** *lograda por* Sapienz*?*

RQ5: (Apps Industriales) *¿Cómo se comparan los resultados en aplicaciones de código abierto con las de* ***código cerrado*** *del mundo real?*

Para responder a estas preguntas, realizamos un estudio empírico. En la siguiente subsección, describimos la configuración experimental.

4.4.1. Configuración experimental

Realizamos dos estudios para responder a las preguntas de investigación anteriores: el estudio "A" aborda las preguntas RQ1 a RQ4 y el estudio "B" aborda la pregunta RQ5. En ambos casos, intentamos imitar lo mejor posible la configuración experimental utilizada en el Estudio #2 presentado por Mao et al. [86].

Selección de sujetos de prueba. Para el estudio "A", elegimos utilizar los 10 sujetos ya utilizados en el Estudio #2 de Mao et al. [86]. Estos sujetos fueron seleccionados al azar por los autores de Sapienz del repositorio F-Droid de aplicaciones

ANDROID de código abierto. Se descartaron dos sujetos preliminares (`BabyCare` y `Hydrate`) debido a la falta de código fuente (`BabyCare`) y a dependencias que ya no son compatibles con la versión de la plataforma ANDROID utilizada en nuestro estudio (`Hydrate`). Los 8 sujetos de código abierto restantes utilizados en nuestro estudio se muestran en la Tabla 4.1.

Para el estudio "B", seleccionamos al azar 30 aplicaciones de código cerrado de la Google Play Store. Estas aplicaciones fueron elegidas después de un meticuloso proceso de selección. El grupo de selección comenzó con 531 aplicaciones: 200 tomadas del *ranking* general de las 200 mejores aplicaciones gratuitas, y 331 de la unión de todos los *rankings* de las 50 mejores aplicaciones gratuitas en cualquier categoría específica (descartando las que ya estaban presentes en las primeras 200 aplicaciones). De estas 531 aplicaciones, solo 275 son compatibles con ANDROID KitKat (API 19) y dispositivos x86. Lo primero es necesario porque es la última versión de ANDROID compatible con el prototipo de SAPIENZ disponible públicamente en GitHub. Lo segundo es necesario para ejecutar las aplicaciones en emuladores. De esas 275 aplicaciones, seleccionamos al azar 30 adecuadas para la instrumentación utilizando la herramienta ELLA-customized [204]. Vale la pena señalar que estas son aplicaciones reales de terceros que pueden utilizar técnicas de ofuscación y anti-depuración, y podrían ser más difíciles de instrumentar. Los 30 sujetos de código cerrado se muestran en la Tabla 4.2.

No elegimos ninguna de las aplicaciones presentadas en los estudios #1 y #3 de Mao et al. [86]. Aunque las 68 aplicaciones del Estudio #1 de Mao et al. [86] podrían haber sido utilizadas en nuestro Estudio "A", preferimos utilizar las 10 aplicaciones presentadas en el Estudio #2 de Mao et al. [86], ya que ya habían sido utilizadas en un estudio con análisis estadístico. En cuanto a las 1,000 aplicaciones de Google Play Store utilizadas en el Estudio #3 de Mao et al. [86], sus nombres no están presentes en el artículo, por lo que no pudimos descargarlas.

Nuestra lógica detrás de la selección de sujetos para los estudios "A" y "B" tiene como objetivo minimizar posibles amenazas a la validez externa. Por un lado, decidimos tener un estudio que utiliza 8 sujetos de código abierto. Dado que el software de código abierto no siempre es el mejor representante de los sujetos del mundo real, optamos por tener un mayor número de repeticiones (30 por combinación de sujeto y algoritmo) para obtener una mejor significancia estadística. Por otro lado, decidimos tener otro estudio (es decir, el estudio "B") que utiliza sujetos de código cerrado. Dado que estos sujetos se tomaron directamente de los *rankings* de las mejores aplicaciones gratuitas en la Google Play Store, podemos asumir que son buenos representantes de los sujetos populares del mundo real. Por lo tanto, en este caso, nos conformamos con una sola repetición por combinación de sujeto y algoritmo.

Implementación. Como explicamos en la sección Sección 4.3, SAPIENZ [86] implementa el algoritmo evolutivo multiobjetivo NSGA-II que incluye una representación de los individuos como secuencias de genes tanto *atómicos* como *motif*. Extendimos la

Tabla 4.1: Aplicaciones de código abierto utilizadas en el estudio "A".

App	Descripción	Versión	Fecha	LOC
Arity	Calculadora científica	1.27	2012-02-11	2.821
BookWorm	Gestor de colección de libros	1.0.18	2011-05-04	7.589
DroidSat	Visor de satélite	2.52	2015-01-11	15.149
FillUp	Calcular el consumo de combustible	1.7.2	2015-03-10	10.400
JustSit	Temporizador de meditación	0.3.3	2012-07-26	728
Kanji	Reconocimiento de caracteres	1.0	2012-10-30	200.154
L9Droid	Ficción interactiva	0.6	2015-01-06	18.040
Maniana	Lista de tareas	1.26	2013-06-28	20.263

versión más reciente disponible públicamente de SAPIENZ en GitHub [205], agregando los algoritmos descritos en la sección Sección 4.2. Estos algoritmos se implementaron utilizando una función de *fitness* simple de un solo objetivo: la *fitness* de un individuo se asigna según la cobertura de sentencias acumulada lograda durante la ejecución. Aunque es posible implementar una variante multiobjetivo, decidimos utilizar una función de *fitness* de un solo objetivo, ya que una parte importante de este estudio empírico consiste en validar si el uso de un algoritmo evolutivo multiobjetivo (por ejemplo, NSGA-II) mejora realmente la efectividad o si algoritmos evolutivos más simples de un solo objetivo podrían lograr resultados similares.

Según indicaron los autores, la versión actual disponible de SAPIENZ en GitHub [205] se considera "desactualizada y ya no es compatible", y la última actividad en el historial de versiones se registró en mayo de 2016. Debido a esto, mejoramos la última versión de SAPIENZ solucionando algunos problemas como la gestión adecuada del presupuesto de tiempo, el manejo de los tiempos de espera al enviar comandos a los emuladores y la recuperación de un bloqueo del emulador. Algunos de los problemas en la implementación original que tuvimos que resolver fueron:

- La inicialización de los emuladores utilizaba parámetros obsoletos que ya no eran compatibles.
- La falta de gestión del presupuesto de tiempo y de un tiempo de espera confiable para los comandos al emulador.
- La falta de un manejo adecuado de los dispositivos para garantizar que el fallo de uno o más dispositivos al ejecutar un caso de *test* no detenga toda la experimentación.
- Falta de información en tiempo de ejecución sobre la eficiencia de cada algoritmo.
- Por último, los comandos utilizados en el código fuente impedían extender fácilmente la herramienta para utilizar otros algoritmos basados en búsqueda.

Además del algoritmo original utilizado por SAPIENZ, hemos considerado, y luego implementado, otros 8 algoritmos en nuestra extensión de la herramienta. Específicamente, *Random Search* (con y sin genes *motif*), *Standard GA*, *Monotonic GA*, *Steady-state GA*, $\mu + \lambda$ EA, (μ,λ) EA y NSGA-II (es decir, SAPIENZ sin genes

Tabla 4.2: Aplicaciones de código cerrado utilizadas en el estudio "B".

App	Descripción	Versión	Fecha	Descargas
AccuWeather	Alertas meteorológicas e información de pronóstico en vivo	5.8.6-free	2021-07-02	100.000.000+
Bitmoji	Crea tu propio emoji personal	10.47.177	2021-05-31	100.000.000+
Body Temperature Records	Aplicación para el seguimiento de la fiebre	1.0	2021-07-15	10.000+
Calorie Counter by Lose It	Contador de calorías y diario de alimentos	7.1.1	2021-07-07	10.000.000+
ClipKey	Gestor de portapapeles	1.3.2	2017-05-07	500.000+
Craigslist	Clasificados *online*	1.14.2	2021-04-13	1.000.000+
CT Prepares	Información de emergencia para residentes de Connecticut	1.0.3	2016-11-01	5.000+
Daily Mail Online	Aplicación de noticias	4.0.1	2021-08-02	5.000.000+
Dark Horse Comics	Leer cómics	1.3.17	2020-12-18	1.000.000+
Durham Bus Tracker	Ubicación en tiempo real de autobuses escolares	1.6.0	2019-12-13	50.000+
Ecobee	Gestor de dispositivos para el hogar inteligente	3.2.2	2021-07-06	1.000.000+
Google Translate	Aplicación de traducción de texto	4.4.0.RC01.-104701208	2021-06-01	1.000.000.000+
Indeed Job Search	Aplicación de búsqueda de empleo	4.6	2021-06-02	100.000.000+
Interval Timer	Temporizador minimalista	1.2.6	2021-05-24	5.000.000+
Mobills	Planificador de presupuesto y seguimiento de finanzas	5.3.7	2021-08-23	5.000.000+
Move to iOS	Aplicación para migrar a iOS	3.1.2	2021-05-18	50.000.000+
My Baby Firework	Exhibición de fuegos artificiales con sonido	2.116.5	2019-12-16	1.000.000+
My Baby Piano	Exhibición de piano con sonido	2.146.9	2019-12-16	5.000.000+
MyChart	Aplicación para el almacenamiento de información médica	5.1	2021-05-17	10.000.000+
Namshi	Compras de moda y belleza *online*	3.0.4	2021-08-12	10.000.000+
Pandora	Reproducción de música, radio y podcasts	8.7.1	2021-05-20	100.000.000+
Remote for Roku by Codematics	Aplicación para controlar Smart TV	1.29	2021-06-15	1.000.000+
Roku	Control remoto oficial de Roku	3.6.0.2281118	2021-05-08	10.000.000+
Snapseed	Un editor de fotos profesional	2.19.0.-201907232	2020-04-14	100.000.000+
SoftList	Lista de compras	2.5.0	2021-05-26	1.000.000+
SwingU	GPS de golf y tarjeta de puntuación	5.0.62	2021-06-28	1.000.000+
USA Today	Aplicación de noticias	3.1.3	2021-06-17	5.000.000+
VINDecoded	Informe de verificación y historial de VIN	8.3.0.0	2021-05-28	500.000+
Weather by WeatherBug	Mapa de radar en vivo y pronóstico del tiempo	5.4.4.38	2021-06-30	10.000.000+
Yelp	Encuentra comida, entrega y servicios cercanos	9.33.0	2021-06-03	50.000.000+

motif). Todos los algoritmos implementados y la mejora de la implementación de SAPIENZ están disponibles públicamente en GitHub [206]. Para este capítulo, hemos considerado un subconjunto de los algoritmos estudiados por Campos et al [207]. Descartamos los algoritmos de búsqueda multiobjetivos (es decir, MOSA [208] y DynaMOSA [209]), ya que estos algoritmos fueron diseñados originalmente para trabajar con criterios de cobertura de *approach level* y *branch distance*, que no son proporcionados por las herramientas EMMA [210] ni ELLA-customized [204] que utilizamos para recopilar información de cobertura.

Es importante destacar que el componente MOTIFCORE de SAPIENZ (que maneja los genes *motif*) no se modificó. En otras palabras, los genes *atómicos* y *motif* utilizados en este estudio son los mismos que propusieron Mao et al. [86]. Para generar individuos sin genes *motif*, simplemente filtramos el contenido de los casos de *test* generados por el componente MOTIFCORE para dejar solo los genes *atómicos*.

Selección de parámetros. Los parámetros se seleccionaron siguiendo las elecciones realizadas en el Estudio #2 de Mao et al. [86]. Para la función de cruce, se utilizó el operador de cruce uniforme. Para la función de mutación, se utilizó una combinación de mezcla y cruce de un punto. Las probabilidades de cruce y mutación se establecieron en 0,7 y 0,3, respectivamente. La función de selección utilizada para el algoritmo NSGA-II fue la misma que la descrita por sus autores originales [33]. La función de selección utilizada para los algoritmos de un solo objetivo fue la selección de ruleta. El número máximo de generaciones se estableció en 100, aunque ninguno de los algoritmos evolutivos alcanzó esta cantidad de generaciones. El tamaño de la población para cada generación fue de 50 individuos, mientras que los individuos estaban compuestos por 5 casos de *test*. La longitud inicial de cada caso de *test* se seleccionó al azar entre 20 y 500 eventos. Todos estos parámetros se mantuvieron constantes en todas las ejecuciones. Optamos por mantener los parámetros constantes para garantizar que la comparación entre los algoritmos sea justa, ya que ajustar los parámetros para cada algoritmo podría cambiar su efectividad [211, 212].

Procedimiento del Experimento. Todos los casos de *test* se generaron en ANDROID KitKat (API 19) porque es la última versión de ANDROID compatible con el prototipo SAPIENZ disponible públicamente en GitHub. Todas las técnicas son completamente automatizadas y no se proporcionó intervención manual (por ejemplo, inicio de sesión) durante la ejecución de los experimentos.

Para cada una de estas ejecuciones, establecimos un límite de tiempo máximo de 2 horas. Aumentamos de forma conservadora el límite de tiempo original de 1 hora utilizado en Mao et al. [86] al doble para mitigar cualquier diferencia en el emulador o el hardware. Esto no representa una amenaza para los enfoques evolutivos, ya que un mayor límite de tiempo permite más evaluaciones de *fitness* (es decir, más generaciones).

Para el Estudio "A", ejecutamos todos los algoritmos de generación de *tests* en la plataforma de computo en la nube de Microsoft Azure [213]. El tipo de máquina virtual elegida fue `D16s_v3`, que cuenta con 16 núcleos y 64 GB de RAM. El sistema

operativo instalado en estas máquinas virtuales fue Ubuntu 14.04. En cada máquina de 16 núcleos, se lanzaron 16 emuladores de ANDROID. Estos emuladores se utilizan todos al mismo tiempo al generar casos de *test* para una aplicación.

Como se mencionó anteriormente, para el Estudio "A" decidimos realizar 30 repeticiones en cada sujeto de código abierto para obtener resultados estadísticamente significativos. Por lo tanto, el tiempo total de experimentación para el Estudio "A" fue de 9 *algoritmos* $\times 8$ *aplicaciones* $\times 30$ *repeticiones* $\times 2$ *horas cada una* $= 4.320$ *horas* (es decir, 180 días en una máquina de 16 núcleos). Si consideramos el tiempo invertido en la emulación, esto representa 4.320 *horas* $\times$ 16 *emuladores* $=$ 69,120 *horas* (es decir, 2.880 días).

Por otro lado, el Estudio "B" se ejecutó en una computadora de escritorio con 8 núcleos y 32 GB de RAM que ejecuta Ubuntu 18.04. En esta máquina, solo se lanzaron 8 emuladores de ANDROID para cada ejecución. A diferencia del Estudio "A", solo se realizó una repetición para cada combinación de algoritmo y aplicación. Por lo tanto, el tiempo total de experimentación para los sujetos de código cerrado fue de 9 *algoritmos* $\times 30$ *aplicaciones* $\times 1$ *repetición* $\times 2$ *horas cada una* $= 540$ *horas* (es decir, 22,5 días en una máquina de 8 núcleos). Si consideramos el tiempo invertido en la emulación, esto representa 540 *horas* $\times$ 8 *emuladores* $= 4.320$ *horas* (es decir, 180 días).

En el prototipo SAPIENZ disponible en GitHub, el componente MOTIFCORE se utiliza tanto para generar nuevas secuencias de *test* aleatorias como para ejecutar una secuencia de *test* dada. Este último se requiere para obtener la cobertura de sentencias de un *test suite*. Sin embargo, encontramos que este componente tiene un defecto conocido que dificulta obtener el valor de *fitness* correcto al generar nuevas secuencias de *test*. Por lo tanto, fue necesario volver a ejecutar los *tests* generados aleatoriamente para obtener su valor de *fitness* correcto. En consecuencia, para no penalizar aquellos enfoques que dependen en gran medida de la generación aleatoria de casos de *test* (como *Random Search*), no consumimos ningún límite de tiempo durante la generación de casos de *test* aleatorios con el componente MOTIFCORE.

Análisis del Experimento. Para el Estudio "A", se obtuvo automáticamente la cobertura de sentencias para los *test suites* generados utilizando la herramienta EMMA [210]. Para el Estudio "B", en cambio, se utilizó la cobertura de métodos proporcionada automáticamente por la herramienta ELLA-customized [204]. Esta herramienta es una versión mejorada (por ejemplo, agrega soporte *multi-dex*) de la herramienta original ELLA [214]. Fue desarrollada y utilizada por primera vez en el artículo de Wang et al. [30]. Para los algoritmos genéticos de objetivo único (es decir, *Standard GA*, *Monotonic GA*, *Steady-state GA*, $\mu + \lambda$ EA y (μ,λ) EA), informamos la cobertura de sentencias lograda por el mejor individuo (es decir, el que tiene la cobertura más alta) en la última generación. Para *Random Search*, informamos la cobertura más alta lograda por cualquier individuo muestreado al azar. Para el algoritmo NSGA-II multiobjetivo, dado que la solución no es un solo individuo, sino un conjunto de individuos (es decir, la frontera de Pareto-óptimo), informamos la cobertura de sentencias más alta lograda por un individuo en la

frontera de Pareto-óptimo.

El número de fallas detectadas para un determinado *test suite* se calculó como el número de fallas únicas encontradas durante su ejecución. Esto es importante ya que la misma falla puede surgir de diferentes secuencias de *tests*. Según la definición de Mao et al. [86], una falla se considera única cuando su *stacktrace* difiere de todos los demás. También excluimos todas las fallas que no fueron causadas por fallas de los sujetos (por ejemplo, aquellas causadas por el sistema ANDROID).

Para el análisis estadístico, seguimos los mismos procedimientos que Panichella et al. [215] y Campos et al. [196] para comparar diferentes algoritmos aleatorizados en un conjunto de sujetos. Específicamente, aplicamos el *test* de Friedman [216] con un nivel de significancia $\alpha = 0{,}05$.

El *test* de Friedman es un *test* no paramétrico para el análisis de múltiples problemas y difiere de los *tests* tradicionales de significancia (por ejemplo, el *test* de Wilcoxon) ya que calcula la clasificación entre algoritmos sobre múltiples problemas independientes, es decir, aplicaciones ANDROID en nuestro caso. Un valor p significativo indica que la hipótesis nula (es decir, ningún algoritmo en el torneo tiene un rendimiento significativamente diferente de los demás) debe ser rechazada a favor de la hipótesis alternativa (es decir, el rendimiento de los algoritmos es significativamente diferente entre sí). Si se rechaza la hipótesis nula, utilizamos el *test* de Conover post hoc para comparaciones múltiples por pares. Este *test* se utiliza para detectar pares de algoritmos que son significativamente diferentes. Finalmente, los valores p obtenidos con el *test* post hoc se ajustan con el procedimiento de Holm-Bonferroni para corregir el nivel de significancia estadística ($\alpha = 0{,}05$) en caso de comparaciones múltiples.

En los casos en los que deseamos obtener una comparación más detallada entre dos algoritmos para un determinado sujeto, utilizamos el U-*test* de Wilcoxon-Mann-Whitney para determinar si existe una diferencia estadísticamente significativa y el tamaño del efecto $\hat{A}_{12}$ de Vargha-Delaney para medir esta diferencia (si la hay).

4.4.2. Resultados del estudio "A": cobertura (RQ1 y RQ2)

La Tabla 4.3 resume los resultados del experimento descrito en la sección anterior. Reportamos la cobertura general de sentencias y el *ranking* de cada algoritmo, obtenidos mediante el *test* de Friedman basado en su rendimiento promedio. La Tabla 4.3 también muestra la desviación estándar y los intervalos de confianza (CI) utilizando el método de *bootstrap* al nivel de significancia del 95% de la cobertura de sentencias obtenida.

Entre todos los algoritmos evaluados, NSGA-II + genes *motif* (es decir, SAPIENZ) es el que logra la mayor cobertura general de sentencias (47%) y CI. El valor p obtenido del *test* de Friedman es $1{,}97e-09$. Esto significa que podemos rechazar la hipótesis nula del *test* de Friedman (es decir, que al menos un algoritmo difiere del resto). La Tabla 4.4 muestra el *ranking* obtenido por cada algoritmo para cada aplicación. Por ejemplo, para el sujeto `Arity`, NSGA-II logró la mejor cobertura de sentencias, mientras que (μ, λ) EA obtuvo el peor rendimiento en términos de esa

Tabla 4.3: Resumen de los resultados de cobertura para el estudio "A": Cobertura general, desviación estándar y *ranking* de cada algoritmo basado en su rendimiento promedio, que es estadísticamente significativo según el *test* de Friedman (valor $p < 0{,}0001$, los datos completos están disponibles en la Tabla 4.4). Para los valores de cobertura promedio, también informamos intervalos de confianza (CI) utilizando el método de *bootstrapping* con un nivel de significancia del 95%.

Algoritmo	Media de Ranking	Desviación Estándar de Ranking	Cobertura Promedio General	Desviación Estándar de Cobertura	CI
NSGA-II + MG (Sapienz)	1,25	0,71	47,87	17,22	[45,68, 50,06]
Random Search + MG	2,12	0,35	46,95	17,65	[44,71, 49,23]
NSGA-II	3,00	1,41	44,07	18,71	[41,70, 46,47]
Random Search	4,19	0,37	43,78	18,57	[41,46, 46,16]
$(\mu + \lambda)$ EA	5,12	1,36	41,79	17,79	[39,47, 44,06]
Monotonic GA	5,56	0,82	40,70	17,62	[38,46, 42,91]
(μ, λ) EA	7,62	1,06	37,23	17,85	[35,00, 39,51]
Standard GA	8,00	0,76	34,49	19,43	[32,05, 36,96]
Steady State GA	8,12	0,83	33,23	19,80	[30,70, 35,74]

métrica.

La Tabla 4.5 muestra los valores p obtenidos por el *test* de Conover para la comparación entre pares. Estos valores p indican si existe o no una diferencia estadística para cada par de algoritmos. Por ejemplo, aunque la Tabla 4.3 muestra que $(\mu + \lambda)$ EA obtuvo una clasificación más alta que *Monotonic GA*, la Tabla 4.5 indica que el valor p obtenido para el *test* de Conover es mucho mayor que 0,05. Por lo tanto, no hay suficiente evidencia para afirmar con confianza estadística que el promedio de $(\mu + \lambda)$ EA es diferente al de *Monotonic GA*. La Figura 4.2 muestra visualmente la cobertura general de sentencias lograda por cada algoritmo.

RQ1: ¿Cuál es la contribución de los genes *motif* en la cobertura de Sapienz? Para responder a esta pregunta, realizamos un torneo entre NSGA-II con genes *motif* (es decir, Sapienz) y NSGA-II. Además, para comprender si los genes *motif* contribuyen a obtener mayores beneficios (incluso sin utilizar un algoritmo evolutivo), también incluimos en el torneo *Random Search* y *Random Search* con genes *motif*.

La Tabla 4.6 resume los resultados del torneo de enfrentamientos uno a uno. Dado un sujeto particular, se considera que el Algoritmo X es mejor que el Algoritmo Y para ese sujeto si el resultado del U-*test* de Wilcoxon-Mann-Whitney da un valor p menor a 0,05 (es decir, podemos decir con confianza estadística que el Algoritmo X es diferente del Algoritmo Y) y el tamaño de efecto de Vargha-Delaney $\hat{A}_{12}$ es mayor que 0,5. Coloquialmente, esto significa que el Algoritmo X tiene un rendimiento significativamente mejor en un mayor número de comparaciones que el Algoritmo Y. Si el valor p de U-*test* es mayor o igual a 0,05, no podemos concluir que el Algoritmo

Tabla 4.4: Ranking completo de la cobertura lograda por los algoritmos para cada sujeto en el estudio "A".

	$(\mu + \lambda)$ EA	(μ, λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG	Random Search	Random S. + MG	Standard GA	Steady State GA
Arity	5,00	9,00	6,00	1,00	3,00	4,00	2,00	8,00	7,00
BookWorm	3,00	8,00	4,00	6,00	1,00	5,00	2,00	7,00	9,00
DroidSat	8,00	6,00	4,50	2,00	1,00	4,50	3,00	8,00	8,00
FillUp	5,00	7,00	6,00	3,00	1,00	4,00	2,00	8,00	9,00
JustSit	5,00	7,00	6,00	3,00	1,00	4,00	2,00	9,00	8,00
Kanji	5,00	8,00	6,00	3,00	1,00	4,00	2,00	7,00	9,00
L9Droid	5,00	7,00	6,00	3,00	1,00	4,00	2,00	9,00	8,00
Maniana	5,00	9,00	6,00	3,00	1,00	4,00	2,00	8,00	7,00
Mean	5,12	7,62	5,56	3,00	1,25	4,19	2,12	8,00	8,12

X sea ni mejor ni peor que el Algoritmo Y. Las posiciones en el torneo se deciden clasificando las diferencias entre las columnas "Mejor que" y "Peor que". Por ejemplo, SAPIENZ tiene una diferencia de $16 - 0 = 16$, mientras que *Random Search* con genes *motif* tiene una diferencia de $10 - 3 = 7$, lo que significa que la primera debería estar más alta en el torneo que la última.

Podemos ver que la primera posición se asigna a SAPIENZ con un rendimiento significativamente mejor en 16 de 24 comparaciones y un tamaño de efecto promedio de 0,86. Además, en las otras 8 de 24 comparaciones, SAPIENZ no es significativamente peor. Sorprendentemente, la segunda posición es para *Random Search* con genes *motif*. Este algoritmo tiene un rendimiento significativamente mejor en 10 de 24 comparaciones y un tamaño de efecto promedio de 0,90.

En general, se puede observar en la Tabla 4.6 una clara mejora en la cobertura de sentencias en aquellos algoritmos que incluyen genes *motif* en comparación con sus contrapartes sin genes *motif*. En términos de significancia estadística, la Tabla 4.5 muestra que hay suficiente evidencia estadística para sostener que hay una diferencia entre ambos: NSGA-II y NSGA-II con genes *motif* (es decir, SAPIENZ) y entre *Random Search* y *Random Search* con genes *motif*.

Conjeturamos que este incremento se debe a que los genes *motif* utilizan una representación más compacta que los genes *atómicos*. En otras palabras, un patrón de usuario complejo se puede representar con una secuencia de N eventos *atómicos* o un solo gen *motif*. Esto significa que, siempre y cuando ese gen particular siga propagándose a través de las generaciones, el patrón sobrevivirá en la población. Por otro lado, si ese mismo patrón se dispersa en varias docenas de eventos, sería

Tabla 4.5: Resultados del *test* post hoc de Conover para el análisis por pares de la cobertura alcanzada en el Estudio "A". Un valor p menor que 0,05 para los algoritmos X e Y significa que hay suficiente evidencia para afirmar que son diferentes con significancia estadística.

	$(\mu+\lambda)$ EA	(μ,λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG	Random Search	Random S. + MG	Standard GA
(μ,λ) EA	$< 0{,}05$	-	-	-	-	-	-	-
Monotonic GA	1,000	$< 0{,}05$	-	-	-	-	-	-
NSGA-II	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	-	-	-	-	-
NSGA-II + MG	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	-	-	-	-
Random Search	0,410	$< 0{,}05$	0,058	0,141	$< 0{,}05$	-	-	-
Random S. + MG	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	0,462	0,462	$< 0{,}05$	-	-
Standard GA	$< 0{,}05$	1,000	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	-
Steady State GA	$< 0{,}05$	1,000	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	1,000

más fácil que los operadores de recombinación y mutación lo rompan y pierdan sus beneficios. En resumen, una representación más compacta de los casos de *test* podría ayudar a reducir el espacio de búsqueda y lograr una mayor cobertura de sentencias.

RQ1: Los genes **motif** tienen un impacto **significativo** en la cobertura de SAPIENZ. De hecho, tanto NSGA-II como **Random Search** mejoran su cobertura cuando los casos de *test* incluyen genes **motif**.

RQ2: ¿Cuál es la contribución del algoritmo evolutivo NSGA-II en la cobertura de SAPIENZ? Para responder a esta pregunta, realizamos un torneo de enfrentamientos uno a uno entre NSGA-II y los algoritmos evolutivos presentados en la Sección 4.2. Como referencia para la comparación, también incluimos *Random Search* en el torneo.

La Tabla 4.7 resume los resultados del torneo de enfrentamientos uno a uno. De todos los algoritmos evolutivos evaluados, NSGA-II es el que logra la mayor cobertura general de sentencias (44%). También es significativamente mejor que los demás algoritmos en 36 de 48 comparaciones, y solo es peor en 2 de 48. Un valor promedio de $\hat{A}_{12}$ de 0,88 significa que en las comparaciones en las que NSGA-II es significativamente mejor que otro algoritmo, obtiene una cobertura de sentencias más alta en el 88% de las repeticiones.

Este resultado es consistente con otros estudios, como el realizado por Campos et al. [196] para la generación de casos de *test* unitarios en JAVA, donde los algoritmos multiobjetivo como MOSA (una variación de NSGA-II) y DynaMOSA (una variación posterior de MOSA) mostraron una cobertura más alta en comparación con los algoritmos de búsqueda de objetivo único.

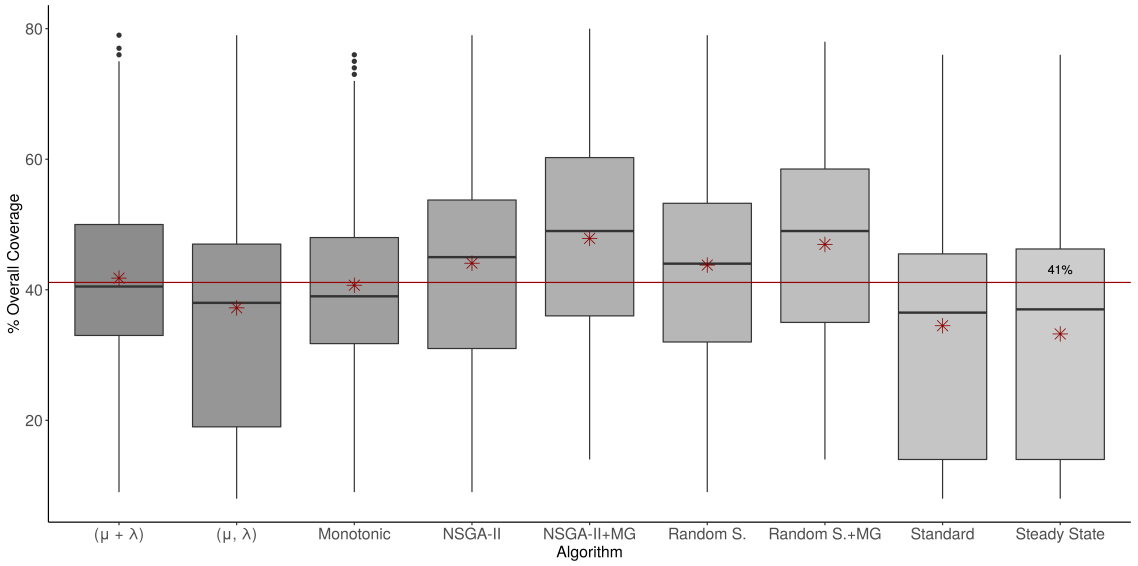


Figura 4.2: Cobertura general alcanzada por cada algoritmo en el estudio "A". La línea media de cada *boxplot* representa la mediana, los círculos negros representan los valores atípicos, el símbolo $\star$ muestra la media, y la línea roja representa la media de todas las coberturas (41%).

Vale la pena señalar que *Random Search* obtuvo el segundo mejor lugar en este torneo de enfrentamientos uno a uno. Además, el valor p obtenido del *test* de Conover al comparar NSGA-II vs *Random Search* es mayor que 0,05. Esto significa que no hay suficiente evidencia para rechazar la hipótesis nula (es decir, que NSGA-II es diferente de *Random Search*). Para comprender mejor la magnitud de esta diferencia entre *Random Search* y los algoritmos evolutivos, realizamos una comparación más detallada y luego calculamos el tamaño de efecto promedio. La Tabla 4.8 muestra los resultados de esta comparación. La Figura 4.3 muestra los resultados visualmente. Como podemos ver, NSGA-II es el único algoritmo que logra un tamaño de efecto promedio mayor que 0,5, pero no hay significancia estadística.

En otras palabras, *Random Search* es al menos tan bueno como NSGA-II, *Monotonic GA* y $(\mu + \lambda)$ EA. Para todos los demás algoritmos evolutivos, *Random Search* es mejor con significancia estadística. En resumen, esto significa que los algoritmos evolutivos no están contribuyendo sustancialmente a lograr una mejor cobertura de sentencias en la generación de *tests* ANDROID. Este resultado también es consistente con el estudio presentado por Sell et al. [98], donde los algoritmos de búsqueda de objetivo único y multiobjetivo no tienen un rendimiento mejor que los algoritmos puramente aleatorios, y a veces incluso tienen un rendimiento ligeramente peor.

Para optimizar una población hacia un objetivo dado, los algoritmos evolutivos requieren evolucionar tantas generaciones como sea posible. El costo de una evaluación de *fitness* afecta directamente la cantidad de generaciones que el algoritmo evolutivo puede alcanzar. En particular, para la generación de *tests* ANDROID, con el fin de obtener la cobertura de sentencias para un individuo dado, los enfoques evolutivos necesitan: enviar el caso de *test* a un dispositivo/emulador, iniciar la aplicación,

Tabla 4.6: Comparación de cobertura lograda por algoritmos con y sin genes *motif* en el estudio "A". "Mejor que" y "Peor que" indican el número de comparaciones para las cuales el mejor algoritmo evolutivo es estadísticamente significativo (es decir, el valor p del test de Wilcoxon-Mann-Whitney es menor a 0,05) mejor y peor, respectivamente. Las columnas $\hat{A}_{12}$ dan el tamaño promedio del efecto.

Algoritmo	Posición del Torneo	Cobertura Promedio General	Mejor		Peor	
			que	$\hat{A}_{12}$	que	$\hat{A}_{12}$
NSGA-II + MG	1,00	47,87	16/24	0,86	0/24	-
NSGA-II	3,00	44,07	1/24	0,66	10/24	0,07
Random S. + MG	2,00	46,95	10/24	0,90	3/24	0,32
Random Search	4,00	43,78	0/24	-	14/24	0,14

Tabla 4.7: Comparación de cobertura lograda por algoritmos evolutivos y *Random Search* en el estudio "A". "Mejor que" y "Peor que" indican el número de comparaciones para las cuales el mejor algoritmo evolutivo es estadísticamente significativo (es decir, el valor p del test de Wilcoxon-Mann-Whitney es menor a 0,05) mejor y peor, respectivamente. Las columnas $\hat{A}_{12}$ dan el tamaño promedio del efecto.

Algoritmo	Posición del Torneo	Cobertura Promedio General	Mejor		Peor	
			que	$\hat{A}_{12}$	que	$\hat{A}_{12}$
NSGA-II	1,00	44,07	36/48	0,88	2/48	0,31
Random Search	2,00	43,78	33/48	0,90	1/48	0,34
Standard GA	6,50	34,49	1/48	0,73	30/48	0,15
Monotonic GA	4,00	40,70	17/48	0,78	14/48	0,14
Steady State GA	6,50	33,23	1/48	0,68	30/48	0,09
$(\mu + \lambda)$ EA	3,00	41,79	24/48	0,79	13/48	0,21
(μ, λ) EA	5,00	37,23	5/48	0,83	27/48	0,18

ejecutar el caso de *test*, recopilar la información de *fitness* y extraerla del dispositivo/emulador. En [98], Sell et al. sugieren que los altos costos de ejecución dificultan cualquier evolución significativa para los algoritmos de búsqueda. En nuestro estudio, observamos que la evaluación de *fitness* puede llevar hasta 60 segundos para un caso de *test*, dependiendo de su longitud. En general, esto resultó en aproximadamente 30 generaciones en promedio para cada algoritmo evolutivo. Con una población de 50 individuos, el número máximo de evaluaciones de *fitness* logradas dentro del presupuesto de tiempo de 2 horas fue en promedio $50 \times 30 = 1.500$ evaluaciones de *fitness*. Resultados similares también se pueden encontrar en el trabajo de Vogel et al. [217], donde los autores informan tiempos de ejecución de 101 minutos en promedio, y hasta 5 horas, para ejecutar 10 generaciones de SAPIENZ en una aplicación (usando 10 emuladores de ANDROID).

Finalmente, la Tabla 4.5 indica que no hay suficiente evidencia para sostener

Tabla 4.8: Comparación de cobertura lograda por algoritmos evolutivos y *Random Search* en el estudio "A". Los efectos estadísticamente significativos se muestran en negrita.

Algoritmo	Promedio Cobertura General	Random Search	
		$\hat{A}_{12}$	p
NSGA-II	44,07	0,56	0,141
Random Search	43,78	-	-
Standard GA	34,49	**0,12**	**< 0,05**
Monotonic GA	40,70	0,23	0,058
Steady State GA	33,23	**0,08**	**< 0,05**
$(\mu + \lambda)$ EA	41,79	0,30	0,410
(μ, λ) EA	37,23	**0,15**	**< 0,05**

Tabla 4.9: Resumen de los resultados de detección de *crashes* para el estudio "A": Número general de *crashes*, desviación estándar y *ranking* de cada algoritmo basado en su rendimiento promedio, que **no** es estadísticamente significativo según el *test* de Friedman (valor p mayor a 0,05, los datos completos están disponibles en la Tabla 4.10). Para los valores de número de *crashes* promedio, también informamos intervalos de confianza (CI) utilizando el método de *bootstrapping* con un nivel de significancia del 95%.

Algoritmo	Media de Ranking	Desviación Estándar de Ranking	Crashes Promedio General	Desviación Estándar de Crashes	CI
NSGA-II + MG	1,75	0,93	1,20	0,78	[1,11, 1,30]
Random Search	2,56	1,18	1,10	0,67	[1,01, 1,18]
Random S. + MG	2,62	1,16	1,16	0,70	[1,07, 1,24]
NSGA-II	3,06	0,73	1,07	0,71	[0,98, 1,16]

con significancia estadística que NSGA-II con genes *motif* (es decir, SAPIENZ) es diferente de *Random Search* con genes *motif*.

RQ2: El algoritmo evolutivo NSGA-II tiene un impacto **marginal** en la cobertura de SAPIENZ. Aunque NSGA-II es mejor que los otros algoritmos evolutivos considerados, **Random Search** es al menos tan bueno como NSGA-II.

4.4.3. Resultados del estudio "A": detección de *crashes* (RQ3 y RQ4)

Para responder a las preguntas de investigación relacionadas con el rendimiento de la detección de *crashes*, aplicamos un análisis similar al utilizado para la cobertura de sentencias. Dado que los AE de un solo objetivo solo buscan mejorar la cobertura de la población, y se demostró que NSGA-II los supera en ese objetivo (ver Sección 4.4.2), en estas preguntas de investigación solo comparamos los 4 mejores algoritmos de la

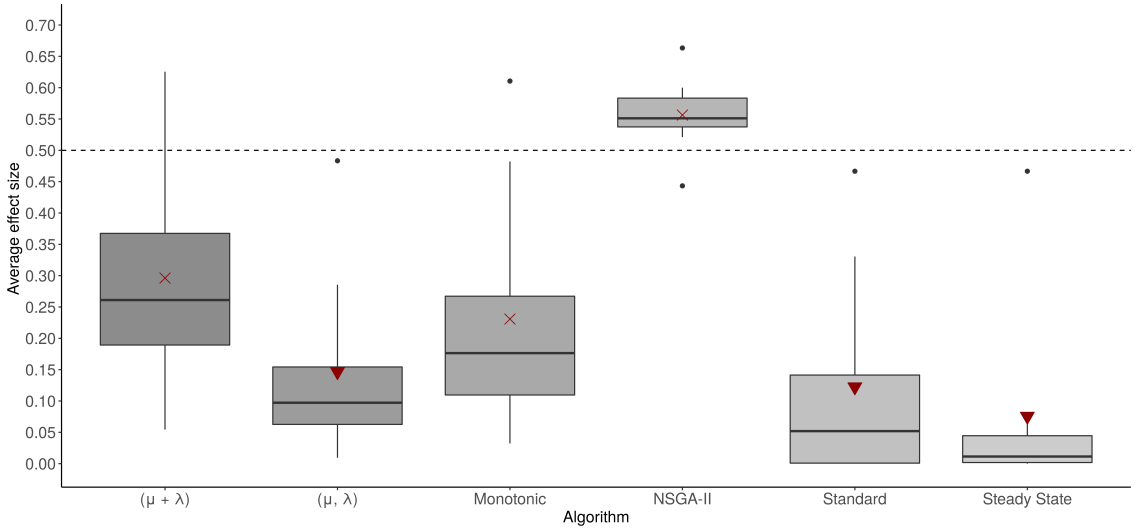


Figura 4.3: Tamaño del efecto $\hat{A}_{12}$ de la cobertura alcanzada por algoritmo evolutivo (EA) X contra *Random Search* en el estudio "A". La línea central de cada *boxplot* marca la mediana, los círculos negros representan los valores atípicos, ▲ representa la media de un tamaño de efecto significativo mayor que 0,5 (es decir, EA X tiene un rendimiento significativamente mejor que *Random Search*), ▼ la media de un tamaño de efecto significativo menor que 0,5 (es decir, EA X tiene un rendimiento significativamente peor que *Random Search*), × representa la media de un tamaño de efecto no significativo.

Tabla 4.3: NSGA-II y *Random Search*, ambos con y sin genes *motif*.

La Tabla 4.9 resume los resultados del experimento en cuanto al número de *crashes* únicos detectados. Entre todos los algoritmos evaluados, NSGA-II + genes *motif* (es decir, SAPIENZ) es el que logra la mayor cantidad de *crashes* detectados en general (1,2 en promedio) y CI. El valor p obtenido del test de Friedman es 0,188. Esto significa que **no** podemos rechazar la hipótesis nula del test de Friedman (es decir, no hay suficiente confianza estadística para afirmar que al menos un algoritmo difiere del resto).

La Tabla 4.10 muestra el *ranking* obtenido por cada algoritmo para cada aplicación. La Figura 4.4 muestra visualmente la cantidad de *crashes* en promedio detectada por cada algoritmo.

RQ3: ¿Cuál es la contribución de los genes *motif* en la detección de *crashes* de SAPIENZ? Para responder a esta pregunta, realizamos un torneo por pares entre NSGA-II y *Random Search*, ambos con y sin genes *motif*. La Tabla 4.11 resume los resultados del torneo por pares.

La primera posición se asigna a SAPIENZ con un rendimiento significativamente mejor en 7 de 24 comparaciones y un tamaño de efecto promedio de 0,63. Además, en las otras 17 de las 24 comparaciones, SAPIENZ no es significativamente peor. La segunda posición corresponde a *Random Search* con genes *motif*. Este algoritmo tiene un rendimiento significativamente mejor en 4 de 24 comparaciones y un tamaño

Tabla 4.10: Ranking completo de número de *crashes* detectados por los algoritmos para cada sujeto en el estudio "A".

	NSGA-II	NSGA-II + MG	Random Search	Random S. + MG
Arity	3,50	1,00	2,00	3,50
BookWorm	3,00	1,50	4,00	1,50
DroidSat	4,00	3,00	1,00	2,00
FillUp	2,00	3,00	4,00	1,00
JustSit	4,00	1,00	3,00	2,00
Kanji	3,00	1,00	3,00	3,00
L9Droid	2,50	1,00	2,50	4,00
Maniana	2,50	2,50	1,00	4,00
Mean	3,06	1,75	2,56	2,62

Tabla 4.11: Comparación de detección de *crashes* lograda por NSGA-II y *Random Search* (con y sin genes *motif*) en el estudio "A". "Mejor que" y "Peor que" indican el número de comparaciones para las cuales el mejor algoritmo evolutivo es estadísticamente significativo (es decir, el valor p del test de Wilcoxon-Mann-Whitney es menor a 0,05) mejor y peor, respectivamente. Las columnas $\hat{A}_{12}$ dan el tamaño promedio del efecto.

Algoritmo	Posición del Torneo	*Crashes* Promedio General	Mejor		Peor	
			que	$\hat{A}_{12}$	que	$\hat{A}_{12}$
NSGA-II + MG	1,00	1,20	7/24	0,63	0/24	-
NSGA-II	3,00	1,07	0/24	-	4/24	0,34
Random S. + MG	2,00	1,16	4/24	0,63	2/24	0,40
Random Search	4,00	1,10	1/24	0,63	6/24	0,37

de efecto promedio de 0,63. Solo es significativamente peor en 2 de 24 comparaciones.

En general, podemos observar en la Tabla 4.11 una mejora marginal en la detección de *crashes* en aquellos algoritmos que incluyen genes *motif* en comparación con sus contrapartes sin genes *motif*. Sin embargo, el hecho de que tanto Sapienz como *Random Search* con genes *motif* tengan un tamaño de efecto promedio de 0,63 en las pocas comparaciones en las que son significativamente diferentes significa que la magnitud de esta diferencia es bastante pequeña (un tamaño de efecto promedio de 0,5 significa que no hay diferencias).

También es importante tener en cuenta que, en términos de detección de *crashes*, no podemos rechazar la hipótesis nula del test de Friedman (el valor p es mayor que 0,05), por lo que no podemos afirmar que haya algoritmos estadísticamente diferentes entre sí.

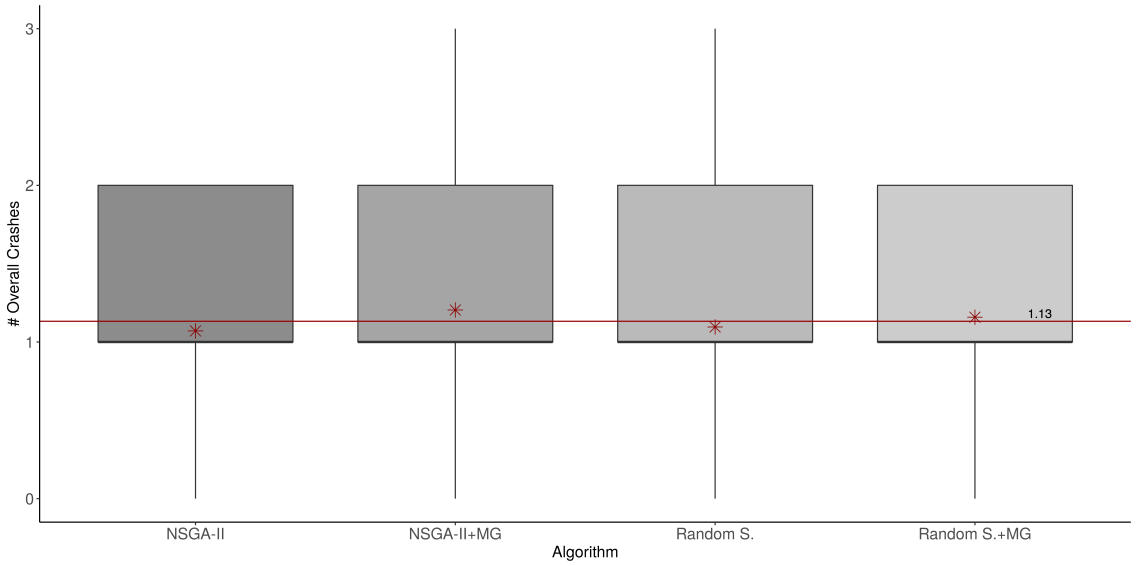


Figura 4.4: Número total de *crashes* detectados por cada algoritmo en el estudio "A". La línea media de cada *boxplot* representa la mediana, los círculos negros representan los valores atípicos, el símbolo ⋆ muestra la media, y la línea roja representa la media de todos los *crashes* (1,13).

RQ3: Los genes *motif* tienen un impacto **marginal** en la detección de *crashes* de SAPIENZ. Aunque tanto NSGA-II como **Random Search** mejoran su detección de *crashes* cuando los casos de *test* incluyen genes *motif*, esta diferencia no es estadísticamente significativa.

RQ4: ¿Cuál es la contribución del algoritmo evolutivo NSGA-II en la detección de *crashes* de SAPIENZ? Para responder a esta pregunta, utilizamos el mismo torneo por pares presentado para RQ3 (Tabla 4.11). Nuevamente, debido a que el valor p del test de Friedman es mayor a 0,05, no podemos rechazar la hipótesis nula y afirmar que haya algoritmos estadísticamente diferentes entre sí.

Aunque podemos observar en la Tabla 4.11 una diferencia muy pequeña en la detección de *crashes* al comparar NSGA-II vs. *Random Search* y SAPIENZ vs. *Random Search* con genes *motif*, no está claro si podemos afirmar que NSGA-II tiene un impacto *marginal* o no. Si observamos los algoritmos sin genes *motif*, NSGA-II no tiene un rendimiento significativamente mejor en ninguna de las 24 comparaciones, mientras que *Random Search* tiene un rendimiento significativamente mejor en solo 1 de esas comparaciones. Al mismo tiempo, *Random Search* tiene un rendimiento significativamente peor en 6 de las 24 comparaciones, mientras que NSGA-II solo en 4.

RQ4: El algoritmo evolutivo NSGA-II tiene un impacto **marginal** o **nulo** en la detección de *crashes* de SAPIENZ. Ya sea utilizando genes *motif* o no, NSGA-II tiene un rendimiento similar al de **Random Search**.

Tabla 4.12: Resumen de los resultados de cobertura para el estudio "B": Cobertura general, desviación estándar y *ranking* de cada algoritmo basado en su rendimiento promedio, que es estadísticamente significativo según el *test* de Friedman (valor $p < 0{,}0001$). Para los valores de cobertura promedio, también informamos intervalos de confianza (CI) utilizando el método de *bootstrapping* con un nivel de significancia del 95%.

Algoritmo	Media de Ranking	Desviación Estándar de Ranking	Cobertura Promedio General	Desviación Estándar de Cobertura	CI
NSGA-II + MG	2,63	1,54	19,07	11,43	[15,05, 23,17]
Random S. + MG	3,47	2,15	18,90	11,56	[14,87, 22,94]
$(\mu + \lambda)$ EA	4,65	1,41	18,00	11,48	[14,00, 21,93]
NSGA-II	4,83	1,33	17,90	11,65	[13,72, 22,00]
Standard GA	5,12	1,32	17,63	11,51	[13,59, 21,67]
Steady State GA	5,73	1,68	17,60	11,40	[13,70, 21,57]
Random Search	5,75	1,82	17,37	11,48	[13,37, 21,37]
Monotonic GA	5,92	1,52	17,50	11,53	[13,42, 21,56]
(μ, λ) EA	6,90	1,67	16,67	11,43	[12,63, 20,71]

4.4.4. RQ5: ¿Cómo se comparan los resultados en aplicaciones de código abierto con aplicaciones de código cerrado del mundo real?

Para responder a esta pregunta, realizamos un análisis estadístico similar al presentado para el Estudio "A". Dado que solo tenemos una repetición para cada combinación de sujeto de código cerrado y algoritmo, no podemos realizar un torneo por pares entre algoritmos. Es decir, no hay suficientes puntos de datos para realizar un *U-test* de Wilcoxon-Mann-Whitney entre dos algoritmos para un sujeto dado. Sin embargo, todavía podemos realizar el *test* de Friedman, calcular su clasificación y realizar el *test* de Conover posterior. Con estos *tests* estadísticos, podemos concluir si hay alguna diferencia en la eficacia entre los algoritmos considerados.

Las Tablas 4.12 y 4.13 resumen los resultados del *test* de Friedman en términos de cobertura lograda y número de fallas únicas detectadas, respectivamente. La Figura 4.5 muestra visualmente la cobertura de método general lograda por cada algoritmo. Como se puede observar, la media general de cobertura para el Estudio "B" es considerablemente menor en comparación con los resultados del Estudio "A" (18% frente a 41%). Una posible explicación de estos resultados podría ser que las aplicaciones de código cerrado del mundo real son más complejas que las de código abierto, o que su funcionalidad completa generalmente está bloqueada detrás de pantallas de inicio de sesión o registro.

El valor p obtenido del *test* de Friedman para la cobertura lograda es $5{,}78e-16$, y para el número de fallas únicas detectadas es $2{,}49e-06$. Esto significa que en ambos casos podemos rechazar la hipótesis nula del *test* de Friedman (es decir, hay al menos un algoritmo que difiere del resto). En términos de cobertura, este resultado coincide con el del Estudio "A" (Sección 4.4.2). Sin embargo, en términos de fallas, este

Tabla 4.13: Resumen de los resultados de detección de *crashes* para el estudio "B": Número general de *crashes*, desviación estándar y *ranking* de cada algoritmo basado en su rendimiento promedio, que es estadísticamente significativo según el *test* de Friedman (valor $p < 0{,}0001$). Para los valores de número de *crashes* promedio, también informamos intervalos de confianza (CI) utilizando el método de *bootstrapping* con un nivel de significancia del 95%.

Algoritmo	Media de Ranking	Desviación Estándar de Ranking	Crashes Promedio General	Desviación Estándar de Crashes	CI
Random S. + MG	4,13	1,61	0,27	0,52	[0,08, 0,46]
Random Search	4,65	1,09	0,13	0,35	[0,01, 0,25]
NSGA-II	4,95	0,74	0,07	0,25	[−0,02, 0,16]
NSGA-II + MG	5,10	0,55	0,03	0,18	[−0,03, 0,10]
$(\mu + \lambda)$ EA	5,23	0,50	0,00	0,00	[0,00, 0,00]
(μ, λ) EA	5,23	0,50	0,00	0,00	[0,00, 0,00]
Monotonic GA	5,23	0,50	0,00	0,00	[0,00, 0,00]
Standard GA	5,23	0,50	0,00	0,00	[0,00, 0,00]
Steady State GA	5,23	0,50	0,00	0,00	[0,00, 0,00]

resultado difiere de los resultados observados en el Estudio "A". Sorprendentemente, hay al menos un algoritmo destacado en el Estudio "B", mientras que en el Estudio "A" no teníamos suficiente evidencia estadística para afirmar alguna diferencia. Sin embargo, el número total de fallas encontradas en el Estudio "B" es menor que el número total de fallas en el Estudio "A". Esto es de alguna manera esperado, ya que las aplicaciones de código cerrado populares son más robustas y maduras que las aplicaciones de código abierto utilizadas en el Estudio "A".

En cuanto a RQ1, en ambos estudios (es decir, "A" y "B") observamos un comportamiento similar. NSGA-II con genes *motif* (es decir, SAPIENZ) se encuentra por encima de NSGA-II en la clasificación de Friedman (Tabla 4.12), y lo mismo ocurre al comparar *Random Search* con genes *motif* con *Random Search*. En términos de significancia estadística, la Tabla 4.14 muestra que hay suficiente evidencia estadística para afirmar que hay una diferencia entre cada algoritmo en ambos pares. En otras palabras, los algoritmos que incluyen genes *motif* tienen una mayor cobertura que sus contrapartes sin genes *motif*, y esta diferencia es estadísticamente significativa.

Conclusiones del Estudio "B" para RQ1: Los genes *motif* tienen un impacto *significativo* en la cobertura de SAPIENZ. De hecho, tanto NSGA-II como *Random Search* mejoran su cobertura cuando los casos de *test* incluyen genes *motif*.

Para RQ2, nuestras conclusiones en el Estudio "B" son similares a las del Estudio "A". A partir de los datos recopilados en el Estudio "A", concluimos anteriormente que NSGA-II tiene un impacto *marginal* en la cobertura de SAPIENZ. En el Estudio "A", NSGA-II fue mejor que los otros algoritmos, pero al mismo tiempo *Random Search* fue al menos tan bueno como NSGA-II. Ahora, en el Estudio "B", los resultados se invierten. Por un lado, NSGA-II parece ser estadísticamente mejor que *Random Search*, con o sin genes *motif*. Por otro, no solo $(\mu + \lambda)$ EA logró una posición más alta que NSGA-II en la clasificación de Friedman (Tabla 4.12), sino que tampoco hay

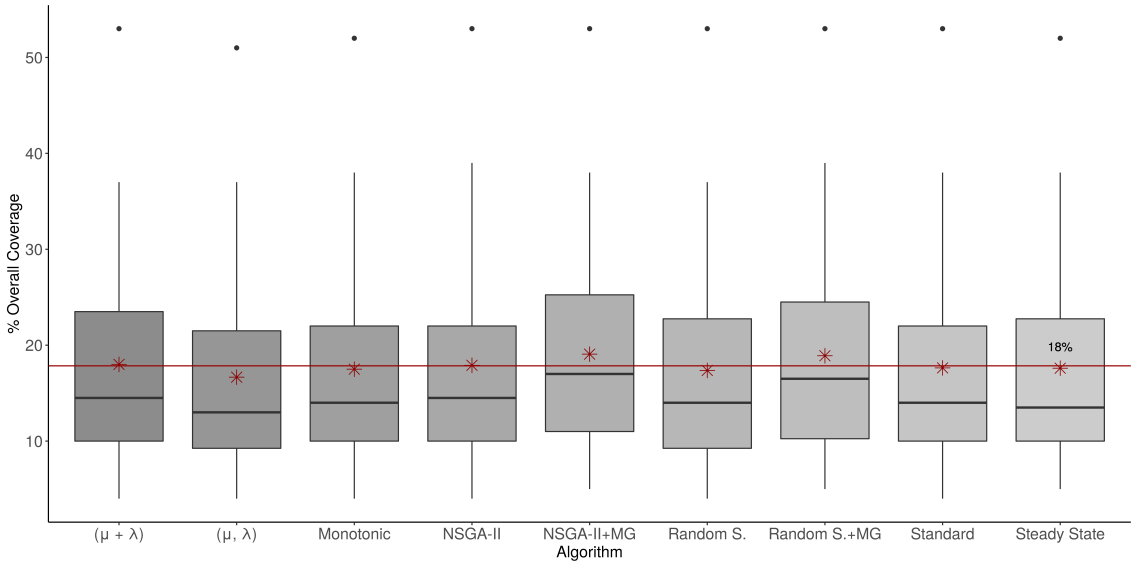


Figura 4.5: Cobertura general lograda por cada algoritmo en el Estudio "B". La línea media de cada *boxplot* representa la mediana, los círculos negros representan los valores atípicos, el símbolo ⋆ muestra la media y la línea roja representa la media de todas las coberturas (18%).

suficiente evidencia estadística para afirmar que son diferentes. Además, *Standard GA*, que se clasificó por debajo de NSGA-II, tampoco es estadísticamente diferente.

Conclusiones del Estudio "B" para RQ2: El algoritmo evolutivo NSGA-II tiene un impacto *marginal* en la cobertura de SAPIENZ. Aunque este algoritmo parece mejorar la cobertura en comparación con *Random Search*, no es estadísticamente diferente de algunos de los otros algoritmos (es decir, $(\mu + \lambda)$ EA y *Standard GA*).

Con respecto a RQ3, la respuesta para el Estudio "A" y el Estudio "B" son diferentes. Aunque *Random Search* con genes *motif* está por encima de *Random Search* en la clasificación de Friedman (Tabla 4.13), lo mismo no ocurre para NSGA-II: la versión sin genes *motif* ocupa el primer lugar en la clasificación. En términos de significancia estadística, la Tabla 4.15 muestra que hay suficiente evidencia estadística para afirmar que solo hay una diferencia entre *Random Search* con y sin genes *motif*. No se puede decir lo mismo de NSGA-II vs. NSGA-II con genes *motif*: parecen ser estadísticamente iguales.

Conclusiones del Estudio "B" para RQ3: Los genes *motif* tienen un impacto *marginal* o *nulo* en la detección de fallas de SAPIENZ. Solo *Random Search* mejora la detección de fallas cuando los casos de *test* incluyen genes *motif*. Lo mismo no ocurre con NSGA-II.

En cuanto a RQ4, el Estudio "B" tiene un resultado similar al Estudio "A". Vale la pena destacar que ninguno de los algoritmos evolutivos de objetivo único logró detectar alguna falla durante la ejecución. En ese sentido, NSGA-II y *Random Search* no solo están por encima del resto de los algoritmos en la clasificación de Friedman (Tabla 4.13), sino que también lograron detectar al menos algunas fallas. En términos de significancia estadística, la Tabla 4.15 muestra que solo hay una

Tabla 4.14: Resultados del *test* post hoc de Conover para el análisis por pares de la cobertura alcanzada en el Estudio "B". Un valor p menor que 0,05 para los algoritmos X e Y significa que hay suficiente evidencia para afirmar que son diferentes con significancia estadística.

	$(\mu+\lambda)$ EA	(μ,λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG	Random Search	Random S. + MG	Standard GA
(μ,λ) EA	$< 0{,}05$	-	-	-	-	-	-	-
Monotonic GA	$< 0{,}05$	$< 0{,}05$	-	-	-	-	-	-
NSGA-II	1,000	$< 0{,}05$	$< 0{,}05$	-	-	-	-	-
NSGA-II + MG	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	-	-	-	-
Random Search	$< 0{,}05$	$< 0{,}05$	1,000	$< 0{,}05$	$< 0{,}05$	-	-	-
Random S. + MG	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	-	-
Standard GA	0,376	$< 0{,}05$	$< 0{,}05$	1,000	$< 0{,}05$	0,094	$< 0{,}05$	-
Steady State GA	$< 0{,}05$	$< 0{,}05$	1,000	$< 0{,}05$	$< 0{,}05$	1,000	$< 0{,}05$	0,099

diferencia estadística entre *Random Search* y NSGA-II cuando tienen genes *motif*. Es decir, NSGA-II no es estadísticamente diferente de *Random Search*. Además, NSGA-II tampoco es estadísticamente diferente de los otros algoritmos.

Conclusiones del Estudio "B" para RQ4: El algoritmo evolutivo NSGA-II tiene un impacto *marginal* o *nulo* en la detección de fallas de SAPIENZ. Aunque NSGA-II logró detectar algunas fallas, no es estadísticamente diferente de los otros algoritmos evolutivos. Además, *Random Search* es al menos tan bueno como NSGA-II.

En resumen, para el Estudio "B", al igual que en el Estudio "A", los genes *motif* siguen ayudando a lograr una mayor cobertura. Además, *Random Search* con genes *motif* sigue siendo al menos tan bueno como NSGA-II con genes *motif* en términos de cobertura y detección de fallas. El cambio principal proviene del hecho de que NSGA-II no es estadísticamente mejor que otros algoritmos en el Estudio "B" cuando analizamos la cobertura lograda. Tampoco está claro que los genes *motif* ayuden a detectar un mayor número de fallas únicas. Asociamos esta ligera diferencia en los resultados relacionados con la detección de fallas al hecho de que el Estudio "B" utiliza aplicaciones de ANDROID de código cerrado del mundo real que podrían tener menos fallas (o fallas más difíciles de encontrar).

RQ5: Los resultados de las RQ 1-2 se mantienen en su mayoría. Los genes **motif** siguen teniendo un impacto **significativo** en la cobertura de SAPIENZ. Sin embargo, NSGA-II podría no ser tan superior a otros algoritmos como creíamos en la RQ2. Los resultados de las RQ 3-4 tienen algunos cambios leves debido a la dificultad de encontrar fallas en aplicaciones populares de código cerrado. A partir de estos resultados, no está tan claro que los genes **motif** ayuden a detectar más fallas.

Tabla 4.15: Resultados del *test* post hoc de Conover para el análisis por pares del número de *crashes* alcanzado en el Estudio "B". Un valor p menor que 0,05 para los algoritmos X e Y significa que hay suficiente evidencia para afirmar que son diferentes con significancia estadística.

	$(\mu + \lambda)$ EA	(μ, λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG	Random Search	Random S. + MG	Standard GA
(μ, λ) EA	1,000	-	-	-	-	-	-	-
Monotonic GA	1,000	1,000	-	-	-	-	-	-
NSGA-II	0,450	0,450	0,450	-	-	-	-	-
NSGA-II + MG	1,000	1,000	1,000	1,000	-	-	-	-
Random Search	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	0,328	$< 0{,}05$	-	-	-
Random S. + MG	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	$< 0{,}05$	-	-
Standard GA	1,000	1,000	1,000	0,450	1,000	$< 0{,}05$	$< 0{,}05$	-
Steady State GA	1,000	1,000	1,000	0,450	1,000	$< 0{,}05$	$< 0{,}05$	1,000

4.4.5. Amenazas a la validez

Las amenazas a la validez (*threats to validity* en inglés) interna pueden surgir de cómo se llevó a cabo el estudio empírico. Dado que todos los algoritmos estudiados se ven afectados por el no determinismo, para el estudio "A" ejecutamos 30 repeticiones de cada experimento con diferentes semillas aleatorias y seguimos rigurosos procedimientos estadísticos para evaluar los resultados. Para evitar posibles factores de confusión al comparar diferentes algoritmos, todos se implementaron en la misma herramienta. Dado que el ajuste de parámetros puede afectar el rendimiento de los algoritmos, utilizamos los mismos valores predeterminados para todos los parámetros en todos los experimentos. Estos valores se eligieron en base al artículo que presenta Sapienz [86]. Utilizamos la selección por ruleta como función de selección para los algoritmos de optimización de un solo objetivo. Aunque la función de selección por *ranking* se prefiere para evitar la convergencia prematura [212], el número de generaciones realizadas por los algoritmos de un solo objetivo en nuestro estudio fue 30 en promedio, lo que mitiga esta posible amenaza.

Otra amenaza a la validez interna podría surgir del hecho de que utilizamos en nuestros experimentos una versión de Sapienz que podría ser diferente de la que se encuentra actualmente en desarrollo en un entorno industrial (es decir, Meta). Elegimos usar esa versión (aunque marcada como "obsoleta y sin soporte") porque sigue siendo la última versión disponible públicamente utilizada para la evaluación por Mao et al. [86].

Medimos el éxito de cada algoritmo en términos de cobertura de sentencias o métodos. Si bien una mayor cobertura es un objetivo deseable para la generación de *tests*, solo es un indicador de la meta más importante de detección de fallas. Por lo tanto, existe una amenaza a la validez de construcción causada por cómo

determinamos qué algoritmo es mejor. En la versión de revista de la publicación asociada ampliamos el trabajo anterior de conferencia [192] al informar cuántos fallos puede detectar cada algoritmo. No obstante, creemos que este criterio de adecuación de *tests* sigue siendo un indicador razonable de la efectividad de diferentes algoritmos basados en la búsqueda.

Las amenazas a la validez externa provienen del hecho de que, debido al gran número de experimentos, solo utilizamos 8 sujetos como estudios de caso para el estudio "A", lo cual aún llevó mucho tiempo incluso al utilizar un clúster de computadoras. Para evitar sesgos de selección, decidimos incluir *solamente* aquellas aplicaciones que han sido utilizadas previamente en un análisis estadístico de SAPIENZ (es decir, Estudio #2 en Mao et al. [86]). Para el estudio "A", en lugar de incluir nuevos sujetos de evaluación, optamos por favorecer un mayor número de repeticiones (30 por combinación de sujeto y algoritmo) para obtener una mejor significancia estadística.

Este trabajo incluyó también un estudio (específicamente, Estudio "B") que utilizó aplicaciones ANDROID de código cerrado del mundo real y populares tomadas directamente de Google Play Store como sujetos. Al hacer esto, nuestro objetivo fue evaluar cómo se comparan los resultados observados en aplicaciones de código abierto con aplicaciones ANDROID de código cerrado del mundo real que son más representativas de las aplicaciones de la industriales. Es importante tener en cuenta que, si bien nos gustaría tener cientos de sujetos, como es el caso en algunos estudios empíricos de generación de *tests* unitarias de JAVA (por ejemplo, Shamshiri et al. [195], Campos et al. [196]), los tiempos de ejecución de la evaluación de *tests* ANDROID hacen que sea muy lento y, por lo tanto, costoso. A modo de comparación, Shamshiri et al. [195] y Campos et al. [196] utilizan 978 y 346 sujetos de JAVA respectivamente, pero su presupuesto de tiempo asignado para cada ejecución es de solo 2 y 1 minuto, respectivamente, mientras que nuestro presupuesto de tiempo es de *2 horas*. Sin embargo, otra selección de sujetos podría llevar a conclusiones diferentes.

Para la selección de algoritmos, consideramos los algoritmos estudiados en [207]. Algunos de los algoritmos de optimización multiobjetivo (por ejemplo, MOSA y DynaMOSA [209]) tuvieron que ser excluidos del estudio ya que no fueron diseñados para trabajar exclusivamente con la cobertura de sentencias proporcionada por EMMA. Aunque incluimos un algoritmo de optimización multiobjetivo (es decir, NSGA-II), incluir otros algoritmos de optimización multiobjetivo también podría llevar a conclusiones diferentes.

4.5. Trabajo relacionado

Sell et al. [98] también presentan un estudio comparativo de diferentes algoritmos para la generación de *tests* en ANDROID, pero estos algoritmos son evaluados en la herramienta de *testing* MATE [97]. Como hemos mencionado anteriormente, MATE utiliza una representación basada en *widgets* para los individuos, mientras que SAPIENZ no lo hace. Esto significa que los operadores evolutivos como el cruce y la mutación son diferentes entre ambas herramientas y pueden influir en los resultados

obtenidos. Además, algunos algoritmos genéticos y evolutivos clásicos en nuestro estudio no están incluidos en el trabajo de Sell et al. [98], a saber: *Monotonic GA*, *Steady-State GA*, $(\mu + \lambda)$ EA, (μ, λ) EA. Finalmente, el trabajo de Sell et al. [98] utiliza principalmente casos de *test* en lugar de *test suites* y no estudia el efecto de los genes de *motif*.

Vogel et al. [218] realizan un análisis del espacio de búsqueda de la función de *fitness* de SAPIENZ utilizando 5 aplicaciones y 5 repeticiones. En su análisis, observan una falta de diversidad en los *test suites* evolucionados y estancamiento de la búsqueda después de 25 generaciones. Luego proponen SAPIENZ ^{div}, una extensión de SAPIENZ que integra mecanismos que promueven la diversidad. Este nuevo algoritmo se evalúa frente a la versión original en 34 aplicaciones con 30 repeticiones. La evaluación mostró que SAPIENZ ^{div} es capaz de lograr resultados mejores o similares a SAPIENZ en términos de cobertura y detección de fallos. Sin embargo, SAPIENZ ^{div} tiende a producir secuencias de *test* más largas y tiene un tiempo de ejecución significativamente mayor en comparación con el algoritmo original.

Pilgun et al. [219] presentan una nueva herramienta llamada ACVTool (Android Code coVerage Tool) para instrumentar y medir la cobertura de código de aplicaciones de código cerrado a nivel de clase, método e instrucción. Demuestran el valor práctico de ACVTool al integrarlo con SAPIENZ y realizar un experimento a gran escala donde comparan diferentes herramientas de cobertura (JACOCO y ELLA) y granularidades (actividad, método, instrumentación y sin cobertura en absoluto). En este estudio, evaluaron el prototipo de ACVTool en un total de 832 aplicaciones de código cerrado de la muestra Google Play Store del conjunto de datos AndroZoo [220] y 446 aplicaciones de código abierto del repositorio F-DROID. Informan que ACVTool logró instrumentar correctamente el 96.9% de las aplicaciones en sus experimentos. Además, sus hallazgos sugieren que incluso al realizar varias repeticiones, una única métrica de cobertura no es capaz de encontrar todos los fallos que fueron detectados por otras métricas. Por lo tanto, ninguna granularidad de cobertura supera claramente a las demás en este aspecto.

Guo et al. [221] realizan un estudio cualitativo de la cobertura de actividad, método y línea reportada por MONKEY [84] y STOAT [90] en 70 aplicaciones de código abierto de ANDROID (una repetición cada una). Informan que el código no explorado se debe principalmente a la falta de conocimiento de dependencias sobre los eventos requeridos y el estado de los *widgets* de la aplicación (es decir, qué acciones provocan una transición de pantalla y cuál es el estado específico del *widget* necesario para lograr la transición). En el mismo trabajo, proponen GESDA, una herramienta que aprovecha el análisis estático de dependencias para construir un modelo de transición de páginas de GUI que captura los *widgets* con devoluciones de llamada en una pantalla y los *widgets* cuyo estado influye en una devolución de llamada. Evalúan esta herramienta en las mismas 70 aplicaciones y muestran que puede superar tanto a MONKEY como a STOAT.

Choudhary et al. [28] comparan varias herramientas de generación de *tests* para ANDROID en un número considerable de aplicaciones de código abierto. Las herramientas consideradas en su estudio son: MONKEY [84] y DYNODROID [85],

EvoDroid [124], GUIRipper [125], PUMA [159], A^{3}E [126], SwiftHand [119], JPF-Android [118] y ACTEve [117]. Aunque es una cantidad impresionante de trabajo empírico, no se centran en las contribuciones específicas del algoritmo subyacente utilizado por cada una de las técnicas.

Wang et al. [30] también comparan varias técnicas de vanguardia en aplicaciones industriales. Las herramientas evaluadas en su estudio son: Monkey [84], WCTester [29, 160], Sapienz, Stoat [90], DroidBot [155] y A^{3}E. El estudio no alcanza una confianza estadística: solo utilizan unas pocas repeticiones para compensar la naturaleza aleatoria de los algoritmos utilizados por las herramientas.

En cuanto a la comparación de diferentes algoritmos evolutivos, la mayoría de los trabajos existentes se centran en el contexto de la generación de *tests* unitarios en Java. Campos et al. [196] realizaron un estudio empírico comparando múltiples algoritmos evolutivos (incluyendo algunos multiobjetivo) y dos enfoques aleatorios para la generación de *tests* unitarios en Java. El estudio se aplicó a una selección de clases de código abierto no triviales. Demuestran que la elección del algoritmo puede tener una influencia sustancial en el rendimiento de la optimización de WTS (*Whole Test Suite*). Panichella et al. [222] también realizaron un estudio empírico con diferentes algoritmos evolutivos para la generación de *tests* unitarios en Java y confirmaron varios de los hallazgos en Campos et al. [196]. Arcuri et al. [212] realizan un estudio exhaustivo sobre la configuración de parámetros para algoritmos basados en búsqueda. Los resultados se analizan estadísticamente en el contexto de la generación de datos de *test* para programas en Java utilizando la herramienta EvoSuite. Sus resultados muestran que la configuración de parámetros tiene un impacto en el rendimiento de un algoritmo de búsqueda. Sin embargo, también muestran que no es fácil encontrar configuraciones que superen significativamente los valores "predeterminados" sugeridos en la literatura. Nuestro trabajo también analiza algoritmos evolutivos y aleatorios basados en búsqueda, pero en el contexto de aplicaciones Android, prestando especial atención al efecto de utilizar genes de *motif*.

4.6. Conclusiones

En este capítulo buscamos estudiar cómo las principales características de Sapienz (específicamente, el algoritmo NSGA-II y la representación de los individuos mediante genes *motif*) impactan en su efectividad. Este estudio empírico fue llevado a cabo tanto en aplicaciones utilizadas previamente en la literatura relacionada como aplicaciones populares del mundo real, con código fuente cerrado, tomadas de Google Play Store.

Nuestro estudio muestra que, en el caso de la generación de *tests* para Android y en términos de cobertura, el algoritmo evolutivo multiobjetivo NSGA-II no presenta una mejora clara en comparación con otros algoritmos. En el Estudio "A", NSGA-II no es distinguible con confianza estadística de *Random Search*, mientras que en el Estudio "B" tiene un rendimiento similar a otros algoritmos evolutivos. Estos resultados generan dudas sobre la efectividad real del algoritmo NSGA-II para

la generación de *tests* en ANDROID. En cuanto al impacto de los genes *motif*, nuestros resultados experimentales proporcionan evidencia que muestra que NSGA-II y *Random Search* obtuvieron mejores resultados estadísticos cuando se incluyeron los genes *motif*. Ambos hallazgos sugieren que la mejora de SAPIENZ en términos de cobertura se debe más a la adición de genes *motif* que a la elección de un algoritmo evolutivo multiobjetivo en particular. Por lo tanto, las comparaciones intra-herramienta (como las realizadas en este artículo y en Sell et al. [98]) deberían ser preferibles a las comparaciones entre herramientas (como la realizada por Mao et al. [86]) siempre que sea posible. En otras palabras, diferentes técnicas deben compararse utilizando la misma herramienta, con el objetivo de evitar la mezcla de factores que causen cambios en la efectividad del *test suite*.

En cuanto a la detección de fallas, sorprendentemente, no observamos diferencias significativas entre los algoritmos estudiados, ya sea que incluyan o no una representación de genes *motif*. Sin embargo, observamos que el número de fallas detectadas en los sujetos es bastante pequeño, lo que podría dificultar el análisis de la capacidad de cada herramienta.

En definitiva, el estudio realizado en este capítulo muestra que los algoritmos puramente aleatorios, como *Random Search*, son una alternativa válida, y atractiva, para la generación de casos de *test* en ANDROID. Estos algoritmos son simples de implementar y prácticos. En contra parte, los algoritmos evolutivos son más complejos y requieren mayor tiempo de ejecución debido a la necesidad de evaluar la *fitness* de cada individuo en la población, produciendo a la larga iguales o peores resultados que los algoritmos puramente aleatorios.

5

Traducción de casos de test al formato Espresso

En este capítulo presentamos un estudio empírico para analizar los desafíos y oportunidades de utilizar herramientas existentes de generación automática de casos de *test* ANDROID para producir *tests* en formato ESPRESSO. Particularmente, analizamos la factibilidad de traducir automáticamente secuencias de acciones sobre *widgets* en *tests* ESPRESSO ejecutables. Los resultados de este estudio guían luego el desarrollo de nuestra técnica de generación de *tests* ESPRESSO en el Capítulo 6. El estudio se realiza utilizando como base la herramienta MATE [97]. Las conclusiones y aprendizajes que sacamos de este estudio son presentadas en las Secciones 5.5 y 5.6. Este capítulo fue publicado como artículo de conferencia [82].

5.1. Introducción

Como mencionamos previamente, muchas de las herramientas existentes para realizar *testing* automático de aplicaciones ANDROID no generan casos de *test* en un formato que motive a los desarrolladores a leer y modificar dichos *tests* más adelante. Por lo tanto, su adopción por parte de los desarrolladores que desean preservar *tests* valiosos y volver a ejecutarlos periódicamente, por ejemplo, en integración continua, se ve obstaculizada. Los desarrolladores se sienten cómodos al escribir casos de *test* en el formato ESPRESSO [25]. Especulamos que generar casos de *test* en ESPRESSO permitiría a los desarrolladores preservar los *tests* generados en sus repositorios de código, refactorizarlos si es necesario o reutilizarlos como inspiración o puntos de partida para ampliar sus *test suites*.

En este trabajo, exploramos la viabilidad de aprovechar o extender herramientas existentes en lugar de desarrollar nuevas desde cero. Presentamos un estudio empírico sobre los desafíos de generar automáticamente *test suites* ESPRESSO a partir de secuencias de interacciones con *widgets*. Con este fin, extendemos la herramienta de *testing* MATE [97] para traducir automáticamente secuencias de acciones de *widgets* en *tests* ESPRESSO ejecutables. Esta extensión consta de dos partes. En primer lugar, refactorizamos MATE para generar la descripción de las secuencias de acciones de *widgets* en formato JSON. En segundo lugar, implementamos un sintetizador de código ESPRESSO para traducir este documento JSON en *tests* ESPRESSO legibles y

reproducibles que se ajusten al comportamiento original de la forma más fiel posible. Las contribuciones de este capítulo son:

- Una extensión de la herramienta de *testing* MATE [97] para sintetizar automáticamente *tests* Espresso.
- Un estudio empírico centrado en la *correctitud* de los casos de *test* Espresso sintetizados. Esto incluye un análisis cuantitativo y cualitativo extenso para asegurar que se preserve la semántica de las secuencias originales.
- Un análisis y discusión de los desafíos y limitaciones de generar automáticamente *tests* Espresso a partir de generadores de *tests* basados en *widgets*. Con el fin de comprender la utilidad de herramientas como la propuesta para los desarrolladores y la comunidad de Android, también presentamos los comentarios recopilados de *Pull Requests* enviados a proyectos de código abierto y de un cuestionario respondido por desarrolladores de una aplicación industrial.

Nuestro estudio empírico proporciona los siguientes aprendizajes:

- La creación de *tests* Espresso es difícil, principalmente debido a la falta de propiedades únicas para identificar de manera inequívoca *widgets* específicos en la interfaz de usuario. Este problema se agrava en algunos casos debido a la definición incompleta o ambigua de los componentes y diseños de la GUI.
- Es importante que la herramienta de generación de *tests* utilizada para la exploración ejecute las acciones en vistas que sean alcanzables para Espresso. Además, para aliviar el problema de identificar *widgets* en la interfaz de usuario, la información proporcionada por la herramienta de *testing* debe ser abundante y precisa.
- Los *tests* Espresso sintetizados son útiles para proyectos con pocos o sin casos de *test* y pueden servir como punto de partida para crear nuevos casos de *test*. También proporcionan una forma rápida y sencilla de aumentar la cobertura del proyecto. Sin embargo, estos *tests* Espresso sintetizados deben incluir una descripción clara de la intención de cada caso de *test*. Además, siempre que sea posible, los casos de *test* deben enfocarse en un escenario de usuario específico.

5.2. Trabajo relacionado

En esta sección mencionamos las diferencias entre el estudio empírico propuesto y aquellos existentes en la literatura (ver Sección 3.4). El estudio empírico presentado en el paper de la herramienta Barista [172] se enfoca principalmente en comparar Barista con otras herramientas de *grabación y reproducción*, como ETR. En cambio, el estudio presentado en este capítulo se centra en evaluar si una herramienta existente de generación automática de *tests* puede ampliarse para producir *tests* Espresso que conserven la semántica original lograda durante la exploración.

En ese sentido, el estudio de Barista sobre la preservación semántica es una comparación manual de los *tests* Espresso generados con casos de *test* escritos

por desarrolladores en lenguaje natural. Esta distinción es importante, ya que las secuencias de interacciones generadas manualmente por los desarrolladores para una herramienta de *grabación y reproducción* pueden diferir de las generadas automáticamente por una herramienta. Sin embargo, algunos de los desafíos presentados en el trabajo de BARISTA [172] son similares a los presentados en este capítulo. Por ejemplo, la necesidad de traducir la secuencia de interacciones con la interfaz de usuario a un formato analizable (ellos eligen un formato de traza personalizado, nosotros elegimos un esquema JSON). También se presenta la dificultad de construir los *View Matchers*. Por lo tanto, en este capítulo aprovechamos las soluciones y heurísticas descritas por ellos para construir nuestro prototipo. El código fuente de BARISTA ya no está disponible públicamente, por lo que no pudimos examinarlo.

La implementación de DIFFDROID [174] demuestra que también es posible extender las herramientas de generación automática de *tests* que no se basan en *widgets* para sintetizar *tests* ESPRESSO. A pesar de ello, el estudio empírico de DIFFDROID se enfoca en evaluar si la herramienta detecta inconsistencias entre plataformas con un número limitado de falsos positivos. No se estudia si los *tests* ESPRESSO sintetizados son semánticamente equivalentes o no a las secuencias originales producidas por MONKEY.

Aunque el estudio empírico de AppTestMigrator [175] proporciona información sobre la precisión de la herramienta en la migración de *tests*, estos casos de *test* fueron escritos por desarrolladores y, como se mencionó anteriormente, pueden diferir de los generados automáticamente.

La motivación del trabajo de Coppola et al. [173] presenta similitudes con el nuestro, pero los enfoques son diferentes. Por ejemplo, su implementación requiere que la aplicación bajo *test* sea instrumentada para complementar los registros de la herramienta basada en imágenes. Su evaluación experimental abarca 60 *tests* creados manualmente por los autores y no aborda si se preserva la semántica de los scripts iniciales de *tests* basadas en imágenes. Sin embargo, en general, su trabajo se alinea con el nuestro al reconocer la necesidad de generar *tests* re-ejecutables ESPRESSO.

Aunque tanto AppTestMigrator como el trabajo de Coppola et al. se enfocan en el formato ESPRESSO, están principalmente centrados en migrar casos de *test* y no en generarlos desde cero. Además, sus estudios empíricos se basan en casos de *test* escritos manualmente y, por lo tanto, pueden diferir de los generados automáticamente.

Cabe destacar que ninguna de las herramientas para generación de *tests* ESPRESSO mencionadas en la Sección 3.4 presenta una evaluación sobre la confiabilidad de los casos de *test* generados. Además, vale la pena mencionar que algunos de los desafíos y limitaciones presentados en este capítulo también han sido mencionados en trabajos relacionados [171–174, 223]. Sin embargo, en la mayoría de estos trabajos, los casos de *test* son escritos manualmente por un participante humano del experimento o un desarrollador externo (por ejemplo, proyectos obtenidos de GitHub). En los pocos casos en que se utilizan casos de *test* generados automáticamente, los desafíos y limitaciones de sintetizar *tests* ESPRESSO no son presentados y analizados completamente. En contraste, este capítulo proporciona un estudio empírico que evalúa la viabilidad y los desafíos de sintetizar *tests* ESPRESSO a partir de casos de

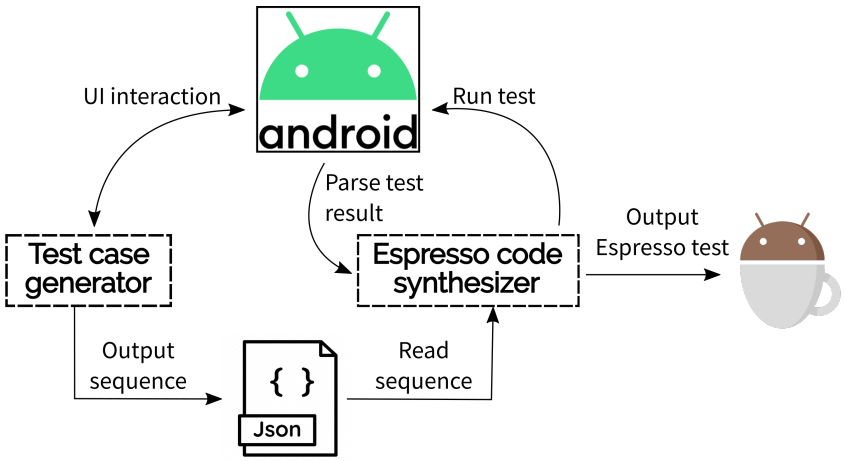


Figura 5.1: Arquitectura del prototipo utilizado para traducción de casos de *test*.

```
[
  {
    "actionType": "CLICK",
    "parameters": "",
    "widget": {
      "classname": "android.widget.TextView",
      "resourceID": "com.myapp:id/button",
      ...widget attributes...
      "parent": {
        "classname": "android.widget.LinearLayout",
        "resourceID": "com.myapp:id/activity_layout"
      },
      "children": []
    },
  }
]
```

(a) Ejemplo de descriptor de acción JSON.

```
@Test
public void clickButtonTest() {
    onView(allOf(
            withId(R.id.button),
            withParent(withId(R.id.activity_layout))
        )).perform(click());
}
```

(b) Ejemplo de un *test* Espresso generado automáticamente.

Figura 5.2: Ejemplo del *input* JSON (a) y el *test* Espresso generado (b).

test basados en *widgets* generados automáticamente. Finalmente, en este capítulo realizamos además un análisis cuantitativo y cualitativo de los resultados. Dicho análisis sistemático no solo presenta una explicación detallada de los desafíos y limitaciones, sino que también muestra cuán generalizados son cada uno de estos problemas.

5.3. Implementación

La arquitectura general de nuestro prototipo se muestra en la Figura 5.1. Este prototipo consta de dos partes principales. Primero, una herramienta de *testing* basada en *widgets* que proporciona un archivo JSON describiendo las secuencias de interacciones sobre la UI de la aplicación bajo *test*. Por ejemplo, si la herramienta de *testing* utiliza un algoritmo evolutivo, estas secuencias pueden describir a los individuos en la última población. Segundo, el módulo "*generador de código Espresso*" que traduce la información en el JSON a un *test suite* Espresso.

Consideramos utilizar MATE, Dynodroid y Stoat como generadores de casos

Algoritmo 9: Traducción de secuencias de acciones de *widgets* en un *test suite* Espresso

```
   Input  : Archivo JSON f
   Output : Conjunto de tests S
1  Suite_E ← {}
2  WActionSeqs ← ParseWidgetActionSequences(f)
3  for WActionSeq ∈ WActionSeqs do
       // Traducir una secuencia de acciones de widgets
4      Test_E ← []
5      for WAction ∈ WActionSeq do
           // Traducir una sola acción de widget
6          TestCode ← BuildViewMatcher(WAction)
7          TestCode.append(BuildViewAction(WAction))
8          Test_E.append(TestCode)
9      if Test_E is not empty do
10         Suite_E ⟵ Suite_E ∪ {Test_E}
11 return Suite_E
```

de *test* subyacentes para el estudio empírico, ya que todos ellos son herramientas modernas basadas en *widgets*. Elegimos MATE [97] porque es la herramienta más moderna de las tres, está en desarrollo activo, teníamos experiencia previa utilizándola y, lo que es más importante, ya se había aplicado a una lista seleccionada de aplicaciones que contenían *tests* Espresso. No obstante, cualquier otra herramienta basada en *widgets* también puede ser utilizada, siempre que se extienda para producir el documento JSON requerido. Sostenemos que el esfuerzo de extender una herramienta basada en *widgets* de esta manera es asumible; a modo de referencia, la implementación de esta extensión en MATE requirió un total de solo 55 líneas de código.

Extendimos MATE para producir el archivo JSON como muestra el diagrama en la Figura 5.1. Este documento describe las secuencias de acciones de *widgets* que constituirán cada caso de *test* Espresso. La Figura 5.2a muestra un ejemplo de la salida JSON. Este esquema es muy simple y sigue la representación propuesta por Negara et al. [171] para ETR. Para cada acción de *widget*, se recopila el tipo de acción, así como cualquier parámetro adicional necesario (por ejemplo, el texto al escribir en un campo de texto). Cada acción de *widget* también hace referencia al *widget* objetivo, sus padres, hijos y atributos específicos si los hubiera (por ejemplo, el texto mostrado). Estos campos son útiles cuando se necesita desambiguar los *widgets*. Se puede encontrar una descripción completa del esquema JSON (que incluye los tipos de acción proporcionados por MATE) en un repositorio público de GitHub [224].

El Algoritmo 9 describe cómo el módulo "*generador de código Espresso*" traduce secuencias de acciones de *widgets* en un *test suite* Espresso. En primer lugar, el archivo `JSON` se analiza para obtener las secuencias de acciones de *widgets* (Línea 2).

A continuación, se construye un *test suite* Espresso traduciendo cada secuencia de acciones de *widgets* en una llamada a `perform` de Espresso que preserve idealmente el significado semántico. Es decir, para cada acción de *widget* (por ejemplo, la que se muestra en la Figura 5.2a), el generador de código produce la declaración de Espresso correspondiente para activar la misma acción utilizando la API de Espresso. Esto se logra mediante la síntesis consecutiva del código *View Matcher* adecuado para seleccionar el *widget* y la correspondiente operación `perform` en el *widget* seleccionado (Línea 6 a Línea 8). Por ejemplo, si la acción de *widget* es un *click*, entonces la operación de `perform` apropiada sería `perform(click())`. La Figura 5.2b muestra un ejemplo en el que la invocación `onView()` selecciona el *widget* objetivo en el que se ejecuta `perform(click())`. Notar que el algoritmo no agrega ninguna *aserción de vista* (*View Assertion*). La implementación no agrega tales aserciones ya que no son estrictamente necesarias para reproducir el comportamiento de la secuencia de acciones de *widgets*. Finalmente, el *test suite* Espresso resultante se guarda como un único archivo Java en el directorio de *tests* especificado por el desarrollador.

Desafíos de implementación. Enfrentamos varios desafíos durante la implementación del prototipo. Para mitigarlos, nos basamos en las heurísticas propuestas por la documentación oficial de Espresso [100] y trabajos del estado del arte. Específicamente, utilizamos las heurísticas propuestas en trabajos relacionados para mitigar el problema de la *desambiguación de widgets* (es decir, [171, 172, 174]). Este problema surge porque el método `onView()` de Espresso, utilizado para seleccionar el *widget* objetivo de una acción o aserción, toma un *View Matcher* que se espera que coincida con un único *widget* dentro de la Jerarquía de Vistas actual.

A menudo, el *widget* deseado tiene un identificador de recurso único que se puede utilizar para seleccionarlo de manera inequívoca (es decir, utilizando el `withId` *matcher* de Espresso). Sin embargo, hay muchos casos legítimos en los que el *widget* objetivo puede no tener un identificador de recurso o el identificador puede no ser único. En este último escenario, el intento de utilizar el `withId` *matcher* generará una excepción de `AmbiguousViewMatcherException`. Por ejemplo, en las listas, los elementos suelen compartir el mismo identificador de recurso cuando se generan dinámicamente (una práctica común en las aplicaciones de Android). Este escenario está ilustrado en la Figura 2.7, donde todos los iconos de la lista tienen el mismo identificador de recurso "*pizzaIcon*".

Algunas de las heurísticas automáticas implementadas para reducir los criterios de coincidencia y mitigar este problema fueron: utilizar el identificador de recurso del padre, como se muestra en la Figura 5.2b; utilizar el identificador de recurso de los hijos (usando los *matchers* `hasDescendant` y `withId`); utilizar el texto de la vista si no está vacío (*matcher* `withText`), o de lo contrario su descripción de contenido (*matcher* `withContentDescription`); utilizar el nombre de la clase de la vista (*matcher* `withClassName`).

Acciones de deslizamiento. Los deslizamientos (*swipe*, en inglés) son gestos en los que el usuario de la aplicación mueve uno o más dedos por la pantalla en una dirección y con una velocidad específicas. El comportamiento específico de estos gestos puede variar según la implementación específica de la UI. Por ejemplo, estos gestos se pueden utilizar para abrir un panel de menú (deslizamiento horizontal corto desde el borde izquierdo), para actualizar una pantalla (deslizamiento vertical desde el borde superior) o para desplazar una lista de elementos. En este último caso, la velocidad del deslizamiento altera la cantidad de elementos desplazados: cuanto más rápido se realiza el deslizamiento, más elementos se saltan. La primera pantalla en la Figura 2.7a ejemplifica esto: según la longitud y la velocidad de la acción de deslizamiento hacia arriba, es posible que no lleguemos al botón "*Next*" al final de la página.

Por lo tanto, es importante que las acciones de deslizamiento utilizadas en los casos de *test* Espresso sintetizados se ajusten a las acciones registradas por la herramienta de generación subyacente. De lo contrario, una secuencia válida de acciones de *widgets* puede dar lugar a un escenario inviable al ejecutar el caso de *test* Espresso. Esto puede suceder, por ejemplo, si la siguiente acción después del desplazamiento debe realizarse en un elemento específico de una lista. Si las acciones de desplazamiento registradas y sintetizadas difieren, es posible que el elemento no sea visible en el *test* Espresso, lo que provocaría el fallo de todas las acciones restantes en el *test*.

Para mitigar este problema, agregamos parámetros especiales para la acción de deslizamiento en el archivo intermedio JSON. Estos son: las coordenadas iniciales y finales $\langle x, y \rangle$ del deslizamiento y la duración del deslizamiento. Luego, en el *test* Espresso, ejecutamos la acción de deslizamiento en el *widget* raíz de la Jerarquía de Vista, utilizando las coordenadas y la velocidad definidas en el JSON. Como resultado, tanto la secuencia registrada de acciones de *widgets* como la ejecución de el *test* Espresso se comportan lo más parecido posible. También agregamos una pequeña demora después de cada deslizamiento en los casos de *test* Espresso para asegurarnos de que no haya animaciones restantes al ejecutar la siguiente acción.

Reproducción de las ejecuciones de MATE. Para permitir que el módulo "*Sintetizador de código Espresso*" vuelva a ejecutar la secuencia de acciones de *widgets* registradas por MATE con el mismo comportamiento, tuvimos que implementar las siguientes tácticas:

- Detección automática de la actividad de inicio definida para cada aplicación bajo *test*. Esto asegura que durante la ejecución tanto de MATE como del módulo "*Sintetizador de código Espresso*", las secuencias siempre comiencen desde la misma actividad.
- Modificamos MATE para que la aplicación se reinstale antes de generar un nuevo caso de *test*. De manera similar, durante la generación de los *tests* Espresso, reinstalamos la aplicación antes de ejecutar cada secuencia de acciones basadas en *widgets*. Esto fue necesario para asegurarnos de que cada ejecución comience

desde el mismo estado inicial, sin interferencias de las ejecuciones de casos de *test* anteriores. Por ejemplo, algunas aplicaciones definen una pantalla de inicio (*Splash*) que solo aparece la primera vez que se abre la aplicación. Si esto se captura en el JSON por MATE y no se borra la aplicación al ejecutar la generación de los *tests* Espresso, podríamos obtener un caso de *test* que falla porque espera que ciertas vistas en la pantalla de inicio estén disponibles.

- Modificamos MATE para otorgar automáticamente todos los permisos a la aplicación bajo *test* al instalarla. Esto asegura que MATE no se detenga en el diálogo de permisos de Android durante la exploración.
- Desactivamos las animaciones en el emulador antes de la exploración de MATE y durante la ejecución de los *tests*. Esto es necesario porque las animaciones causan retrasos (a veces no del todo deterministas) entre acciones, lo que a su vez hace que las vistas no estén visibles o listas en el momento de ejecutar la siguiente acción. Un ejemplo típico de esto es la animación de un panel de navegación: hasta que se abra por completo, no es posible ejecutar acciones en las vistas dentro del menú del panel. Desactivar las animaciones también es una recomendación oficial para evitar problemas de flacidez en los *tests* al utilizar el *framework* Espresso [225].

A lo largo de este trabajo nos enfrentamos a muchos problemas con la exploración realizada por MATE. La implementación del prototipo reveló varios problemas con la exploración realizada por MATE, lo que llevó a incongruencias durante la generación de los *tests* Espresso. Algunos de estos problemas surgieron únicamente del hecho de que MATE utiliza el *Servicio de Accesibilidad* proporcionado por Android para leer e interactuar con la UI bajo *test*:

- En varias ocasiones, en lugar de informar el nombre de clase exacto de una vista, el *Servicio de Accesibilidad* informa el nombre de una de sus superclases. Para mitigar esto, utilizamos un *matcher* personalizado llamado `classOrSuperClassesName` en los *tests* Espresso. Este *matcher* apunta a los *widgets* que tienen un cierto nombre de clase o cualquiera de sus clases padre con dicho nombre.
- Evitamos utilizar el *matcher* `withHint` de Espresso porque la información proporcionada por el *Servicio de Accesibilidad* es inconsistente: para algunas vistas, informa que tienen algún texto cuando en realidad es la sugerencia (*hint*, en inglés) la que se muestra. Esto hace imposible saber si el texto informado en el JSON para una vista es realmente el texto o la sugerencia. Para mitigar esto, utilizamos un *matcher* personalizado llamado `withTextOrHint` en los *tests* Espresso, que busca cualquier vista que tenga la cadena solicitada ya sea como texto o como sugerencia. Como nota aparte, tuvimos que desactivar el uso del *matcher* `withTextOrHint` para los *widgets Switch*, porque el *Servicio de Accesibilidad* informa que estos *widgets* tienen texto, cuando en realidad dicho texto no aparece en la UI usuario y, por lo tanto, Espresso no los reconoce.
- También evitamos utilizar el *matcher* `withContentDescription` de Espresso a menos que no se proporcione otra información (es decir, ni identificador de

recurso ni texto). Desafortunadamente, la información proporcionada por el *Servicio de Accesibilidad* no siempre coincide con la información recopilada por Espresso [226].

- Por último, la sensibilidad a las mayúsculas y minúsculas de los textos proporcionados por el *Servicio de Accesibilidad* también es problemática. A veces, estos textos no coinciden con los que se presentan en la interfaz de usuario, o tienen espacios en blanco adicionales alrededor. Implementamos un *matcher* personalizado llamado `equalToTrimmingAndIgnoringCase` que compara la cadena esperada con la que se encuentra en la vista, después de eliminar los espacios en blanco e ignorar las mayúsculas y minúsculas.

5.4. Estudio Empírico

Idealmente, todos los *tests* Espresso sintetizados deberían preservar la semántica de las secuencias de acciones de *widgets* originales a partir de las cuales se generaron. Sin embargo, como se discutió en la Sección 5.3, esto puede no ser siempre posible. Las siguientes preguntas de investigación (*Research Questions* o RQs, en inglés) tienen como objetivo revelar qué tan ocurrente es este problema. Un paquete de replicación de este estudio se puede encontrar *online* [227].

RQ1: *¿Los tests Espresso sintetizados replican la cobertura lograda por secuencias de acción de widget?*

RQ2: *¿Los tests Espresso sintetizados replican los estados de UI logrados por secuencias de acción de widget?*

RQ3: *¿Cuáles son las causas más comunes de falla en los tests Espresso sintetizados?*

5.4.1. Configuración experimental

Selección de sujetos de prueba. Utilizamos como sujetos experimentales las 12 aplicaciones de código abierto con configuración de Espresso recopiladas y utilizadas por Eler et al. [97]. Estas aplicaciones son adecuadas para nuestro estudio ya que MATE puede manejarlas y la mayoría de ellas tienen actividad reciente. Se muestran en la Tabla 5.1.

Configuramos cada sujeto de acuerdo con la estructura de su proyecto. Esto fue necesario ya que la mayoría de ellos tenían diferentes variantes de compilación (por ejemplo, *release*, *debug*, etc.) y producto definidas (que incluyen o no diferentes características o *features*). El módulo "*Sintetizador de código Espresso*" detecta automáticamente la versión de Espresso configurada en el sujeto (por ejemplo, utilizando bibliotecas AndroidX o Android Support), lo que evita problemas de dependencia durante la ejecución.

Implementación. Realizamos cambios menores en el código de MATE para utilizar su salida. En particular, agregamos un pequeño fragmento de código (55 LOC

Tabla 5.1: Sujetos utilizados en el estudio empírico. La última actividad fue inspeccionada el 10 de junio de 2020.

Sujeto	**Última actividad**	**#Actividades** ANDROID	**#*Tests* en el proyecto**
PoetAssistant	2019-10-16	7	152
Equate	2020-05-31	2	6
OneTimePad	2017-11-01	2	0
Orgzly	2020-05-12	13	460
MicroPinner	2019-08-31	1	23
OCReader	2020-06-09	2	17
HomeAssistant	2018-02-07	3	3
OmniNotes	2020-06-05	5	71
Kontalk	2020-04-21	14	59
KolabNotes	2020-03-20	9	0
ShoppingList	2019-03-19	8	7
MyExpenses	2020-06-06	13	521

utilizando la biblioteca JACKSON [228]) que guarda en un archivo JSON los objetos JAVA que representan la población final. También corregimos el tiempo de espera de exploración de MATE, ya que se quedaba ejecutándose varios segundos después de que el tiempo asignado había terminado. Mejoramos la detección de finalización del tiempo de ejecución de MATE utilizando un booleano atómico y verificando antes de cada muestreo de un nuevo individuo. Utilizamos el algoritmo de *Exploración Aleatoria* (*Random Exploration*, en inglés) de MATE para la generación de casos de *test* basados en *widgets*. La exploración aleatoria es un algoritmo bastante simple que ya se ha utilizado en MATE para la generación de casos de *test* ANDROID [98]. El algoritmo funciona muestreando continuamente el espacio de búsqueda hasta que se acaba el tiempo asignado. En cada iteración, se crea un individuo completamente nuevo. Si este nuevo individuo aumenta la cobertura global lograda hasta ese momento, se agrega a la población final. Por lo tanto, la población final consiste en todos los individuos que aumentaron la cobertura global durante la exploración. Vale la pena señalar que los experimentos fueron totalmente automatizados y no se proporcionó intervención manual (por ejemplo, inicios de sesión) durante la exploración de MATE.

Procedimiento del experimento. Los experimentos se ejecutaron en una PC con Ubuntu 18.04. La CPU era un Intel® Core™ i7-7700 @ 3.60GHz × 8 núcleos y la RAM era de 32 GB. Utilizamos emuladores ANDROID Pie (API 28). Para tener en cuenta la aleatoriedad del algoritmo elegido y mitigar el no-determinismo, ejecutamos los experimentos 5 veces en cada sujeto. Establecimos un tiempo máximo de 1 hora para cada ejecución de MATE. Aumentamos de forma conservadora el tiempo máximo original de 30 minutos utilizado por Eler et al. [97] para mitigar cualquier diferencia en el emulador o el hardware. No se estableció un tiempo máximo para

el módulo "*Sintetizador de código* Espresso" ya que la traducción en sí (Línea 5 a Línea 8 del Algoritmo 9) solo tarda unos segundos (despreciable en comparación con el tiempo de exploración).

Análisis del experimento. Obtuvimos la cobertura de sentencias para cada caso de *test* ejecutado utilizando la herramienta JaCoCo [229]. Verificamos manualmente que la cobertura lograda por la exploración de MATE y los *tests* Espresso correspondientes se informara correctamente en JaCoCo. Excluimos del análisis de JaCoCo algunas de las clases que podrían estar en el APK del sujeto pero que no fueron escritas directamente por sus desarrolladores (por ejemplo, bibliotecas Android, clases generadas automáticamente, etc.).

Para cada *test* Espresso generado, también recopilamos el número de llamadas al método *perform* que fallaron. Es decir, cuántas acciones en los casos de *test* originales no se tradujeron correctamente a Espresso. Para contar correctamente las acciones fallidas, agregamos un contador a cada caso de *test* y rodeamos cada acción con un *try-catch* en el que el contador se incrementa cada vez que se lanza una excepción.

Además, para comprender si los casos de *test* generados preservan las mismas semánticas que las secuencias de *tests* basadas en *widgets* originales, recopilamos capturas de pantalla durante su ejecución. En particular, tomamos una captura de pantalla antes de cada acción y una única captura de pantalla al final del *test*. De esta manera, registramos cualquier diferencia visible entre las ejecuciones de los *tests*. Para comparar las capturas de pantalla, realizamos una comparación píxel a píxel de cada imagen utilizando la herramienta `compare` del software de código abierto *imagemagick* [230]. La comparación se realizó con un factor de tolerancia del 20% [231] que ayuda a ignorar las diferencias menores entre las dos imágenes. Luego, utilizamos la métrica especial "*AE*" (abreviatura en inglés de "Error Absoluto") para contar el número real de píxeles que difieren, en el factor de tolerancia actual. Consideramos que dos capturas de pantalla son *diferentes* si la proporción de píxeles que difieren es mayor al 5%. En ese caso, la acción anterior a la captura de pantalla es un punto en la ejecución del *test* donde la secuencia original basada en *widgets* y el *test* Espresso sintetizado *divergieron*. El umbral y el factor de tolerancia elegidos se validaron manualmente utilizando varios ejemplos de capturas de pantalla tomadas de los sujetos en el experimento. También proporcionamos una validación adicional en la Sección 5.4.2.

5.4.2. Resultados

RQ1: ¿Los *tests* Espresso sintetizados replican la cobertura lograda por secuencias de acción de *widget*? La Figura 5.3 muestra para cada sujeto la cobertura alcanzada por MATE y por los *tests* Espresso sintetizados. Estos últimos logran una cobertura similar a las secuencias de acción de los *widgets* de MATE. Aunque las diferencias de cobertura son pequeñas, en la mayoría de los casos los *tests* Espresso sintetizados lograron una cobertura más baja que MATE. Conjeturamos que este comportamiento (que se repite en varios sujetos) se debe al

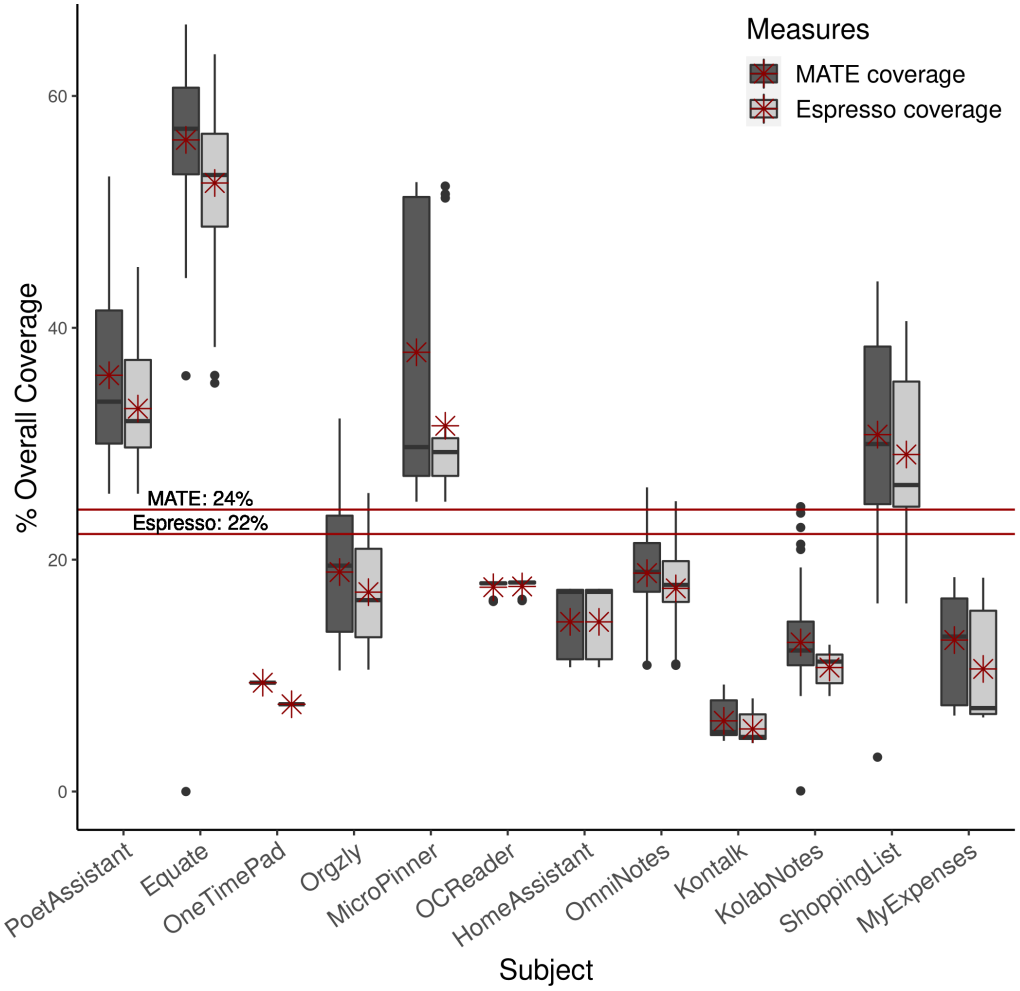


Figura 5.3: Cobertura alcanzada para cada sujeto. La línea media de cada *boxplot* marca la mediana, los círculos negros representan valores atípicos, el símbolo ⋆ muestra la media, y la línea roja representa la media de todas las coberturas (22% para los *tests* Espresso y 24% para MATE).

hecho de que hay al menos una acción fallida en cada *test suite* cuando se ejecuta. La Figura 5.4 muestra el porcentaje de acciones que fallaron al ejecutar los *tests* Espresso sintetizados. Si tomamos el total de acciones ejecutadas para todos los sujetos, encontramos que 30,19% de las acciones fallaron. Si hacemos un promedio por caso de *test*, encontramos que el 25,25% de las acciones fallaron.

RQ1: Los *tests* Espresso sintetizados preservan más del 90% de la cobertura de exploración basada en *widgets* (22% vs. 24%).

RQ2: ¿Los *tests* Espresso sintetizados replican los estados de UI logrados por secuencias de acción de *widget*? La Figura 5.4 también muestra el porcentaje de capturas de pantalla *divergentes* encontradas al ejecutar los *tests* Espresso sintetizados. Dado el alto número de capturas de pantalla divergentes

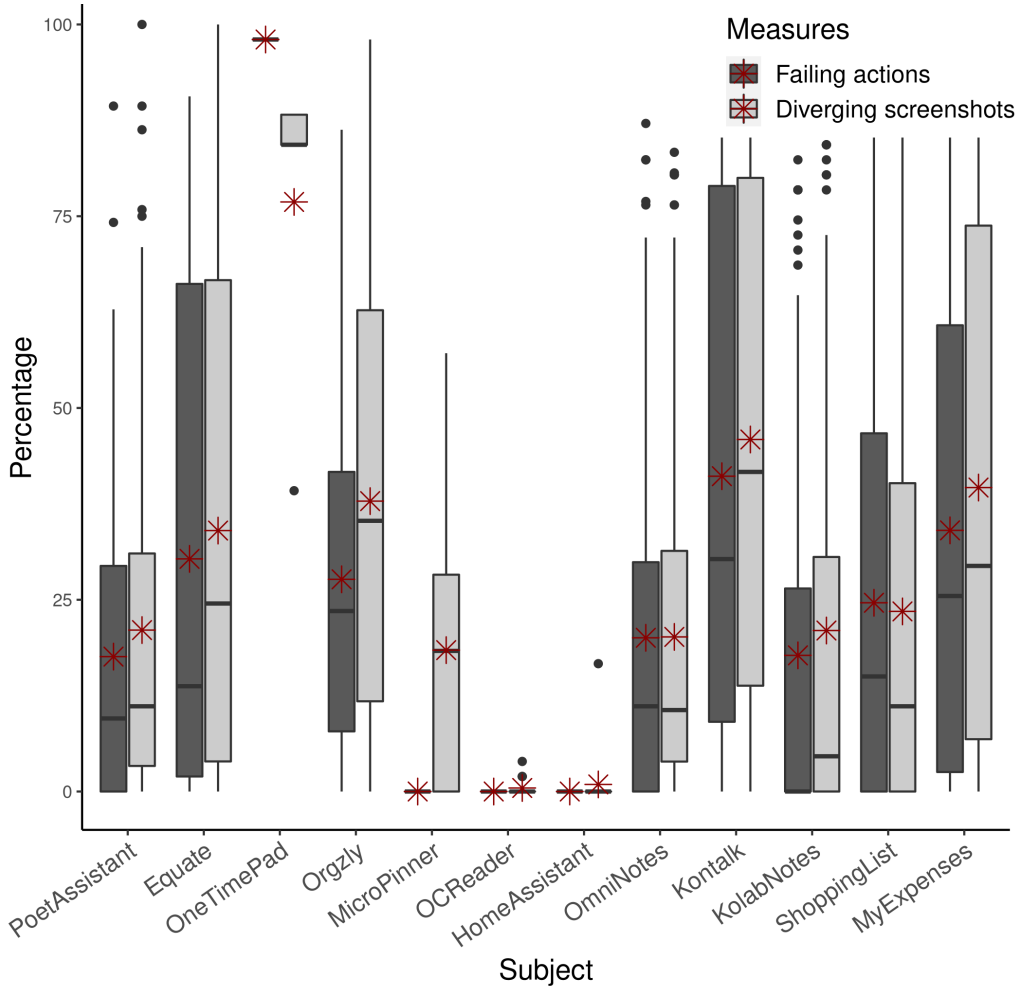


Figura 5.4: Acciones fallidas totales y capturas de pantalla divergentes encontradas al ejecutar los casos de *test* Espresso para cada sujeto. La línea media de cada *boxplot* marca la mediana, los círculos negros representan valores atípicos, y el símbolo ⋆ muestra la media.

encontradas (33,64% de capturas de pantalla divergentes al observar globalmente los sujetos y 29,18% en promedio por cada caso de *test*), decidimos realizar un análisis *cualitativo* más detallado de estas capturas de pantalla, para entender mejor si la comparación píxel a píxel estaba introduciendo errores en nuestros resultados. Para hacerlo, inspeccionamos manualmente más de la mitad de las capturas de pantalla registradas por la primera repetición de nuestro experimento (**3400** pares de capturas de pantalla, precisamente). Cada par contiene una captura de pantalla tomada durante la ejecución de MATE de la secuencia de acción basada en *widgets* y una captura de pantalla tomada durante la ejecución de su acción correspondiente de Espresso.

La Tabla 5.2 muestra el número total de capturas de pantalla inspeccionadas para cada sujeto y la Tabla 5.3 muestra cuántas de las capturas de pantalla inspeccionadas eran realmente divergentes o no. La comparación píxel a píxel pudo

Tabla 5.2: Número de pares de capturas de pantalla inspeccionadas manualmente para cada sujeto.

Sujeto	Pares totales en la 1ra repetición	Pares analizados
PoetAssistant	352	179 (50,85%)
Equate	782	432 (55,24%)
OneTimePad	51	51 (100,00%)
Orgzly	979	547 (55,87%)
MicroPinner	62	47 (75,81%)
OCReader	153	102 (66,67%)
HomeAssistant	49	30 (61,22%)
OmniNotes	985	528 (53,60%)
Kontalk	665	345 (51,88%)
KolabNotes	376	211 (56,12%)
ShoppingList	641	349 (54,45%)
MyExpenses	1130	579 (51,24%)
Total	**6225**	**3400 (54.62%)**

Tabla 5.3: Número de capturas de pantalla que fueron marcadas correctamente o incorrectamente como capturas de pantalla *divergentes* por la comparación píxel a píxel.

Estado marcado	Correctamente	Incorrectamente	**Total**
Divergentes	1257 (36,97%)	58 (1,71%)	**1315** (**38**,**68**%)
Iguales	1618 (47,59%)	467 (13,74%)	**2085** (**61**,**32**%)
Total	**2875** (**84**,**56**%)	**525** (**15**,**44**%)	3400

detectar correctamente el estado divergente del 84,56% de las capturas de pantalla analizadas. Del 15,44% restante, 58 (1,71%) capturas de pantalla fueron marcadas incorrectamente como divergentes y 467 (13,74%) fueron marcadas incorrectamente como la misma pantalla. Esto muestra que, aunque no es perfecta, la comparación píxel a píxel (junto con el factor de tolerancia y el umbral de divergencia elegidos) es capaz de determinar correctamente si dos estados de UI son iguales o no en un número significativo de casos.

En la Tabla 5.4 resumimos las causas observadas para las capturas de pantalla divergentes. La causa principal observada es "*Wrong UI State*", lo que significa que todas las capturas de pantalla que son diferentes son consecuencia de una divergencia previa en el *test*. Por ejemplo, una acción de *back* (ir "*hacía atrás*") que falla al comienzo de un caso de *test* produce que las acciones siguientes en el *test* comiencen desde un estado de UI diferente al que se pretendía en la secuencia original. Esto puede indicar cierta fragilidad en los *tests* de UI, que es un problema conocido que se ha estudiado en la literatura [223]. La siguiente causa observada es "*Off Timing*", que comprende todas las divergencias causadas por pequeñas diferencias de tiempo

Tabla 5.4: Detalles de causas para las capturas de pantalla *divergentes*.

Causa	Cantidad (%)
Wrong UI State	1246 (72,27%)
Off Timing	216 (12,53%)
MATE Exited	129 (7,48%)
Failed Action	93 (5,39%)
Flaky Test	40 (2,32%)
Total	**1724**

al ejecutar los casos de *test*, lo que llevó a que algunas acciones fallaran.

El término "*MATE Exited*" se refiere a un error específico en la implementación de MATE que causó que MATE continuara la exploración incluso después de haber salido de la aplicación bajo *test*. Este comportamiento defectuoso produjo capturas de pantalla repetidas de la pantalla de inicio del emulador y otras aplicaciones no relacionadas al final de algunas ejecuciones.

"*Failed Action*" representa todas las capturas de pantalla que divergieron debido a una acción que falló antes de tomar la captura de pantalla. Por último, "*Flaky Test*" se refiere a todas las capturas de pantalla que divergieron debido a que los casos de *test* generados por MATE son inestables (*flaky*). Por ejemplo, la ejecución de una acción que depende de la fecha, como establecer una alarma, produce un *test* inestable (*flaky*). Otro caso parecido se da en algunos sujetos que obtienen información de servicios web y el resultado de esta información cambia con el tiempo (por ejemplo, las tasas de cambio de divisas). Cabe destacar que en este caso las secuencias de MATE en sí no se pueden replicar, ya que la inestabilidad (*flakiness*) se debe al comportamiento de la aplicación bajo *test*.

RQ2: Los *tests* Espresso sintetizados logran estados de UI similares en aproximadamente dos tercios de las veces. En los casos en que difieren, generalmente se debe a una acción fallida al comienzo de un *test*.

RQ3: ¿Cuáles son las causas más comunes de falla en los *tests* Espresso sintetizados? Dado el alto número de pares de capturas de pantalla no coincidentes que fueron causados por una divergencia previa en el *test*, decidimos estudiar también la primera acción fallida de esos *tests*. En total, inspeccionamos manualmente 94 *tests* (los mismos considerados en nuestro análisis manual de las capturas de pantalla). De esos 94, 31 no contenían ninguna acción fallida. La Tabla 5.5 muestra las causas de la primera acción fallida para los 63 *tests* restantes. Agrupamos las causas según el componente de la arquitectura que fue responsable (es decir, el componente que debería modificarse para evitar el problema).

El módulo de "*sintetizador de código Espresso*" es responsable del 38,10% de las acciones fallidas analizadas. Entre las razones de esto podemos enumerar las siguientes. "*JSON Schema*" se refiere a todos los casos en los que una acción fallida

podría solucionarse mejorando el esquema JSON con información adicional que ya está disponible en MATE. "*Swipe Difference*" se refiere a las acciones fallidas que ocurrieron debido a pequeñas diferencias en la ubicación de la pantalla entre los *tests* Espresso sintetizados y las secuencias de *test* originales basadas en *widget*. Estos cambios hicieron que algunas acciones no estuvieran disponibles o presentes en la pantalla al momento de la re-ejecución. "*Ambiguous Matcher*" representa las acciones que fallaron debido a cualquiera de las heurísticas en el módulo de "*sintetizador de código Espresso*" para construir el *View Matcher*.

El *framework* Espresso es responsable del 30,16% de las acciones fallidas analizadas. "*View Not Displayed*" se refiere a cualquier acción que falló debido a que la vista objetivo no se mostraba en la pantalla, lo cual es una limitación común de Espresso. "*Framework Limitation*" representa otras limitaciones conocidas del *framework* Espresso, por ejemplo, cuando se trabaja con vistas personalizadas. "*Subject Not Supported*" representa las acciones que fallaron debido a un detalle de implementación específico en el sujeto.

Después de una inspección más detallada, encontramos que los errores en la categoría de "*View Not Displayed*" se agravaban por el comportamiento de MATE. Específicamente, MATE a veces realiza acciones en vistas que no están *mostradas* en la pantalla (es decir, vistas que no son directamente visibles para el usuario). Un caso típico de esto es una pantalla con un formulario largo: las vistas en la parte inferior son "visibles", pero no se muestran en la pantalla. La primera pantalla en la Figura 2.7a ejemplifica este caso: las opciones de bebida y el botón "*Next*" son visibles en la interfaz de usuario pero no se muestran actualmente al usuario. Este comportamiento de MATE no coincide con la forma en que funcionan las acciones predeterminadas de Espresso, que asume que las vistas objetivo siempre se encuentran *mostradas* al usuario.

MATE es responsable del 15,87% de las acciones fallidas analizadas. "*Wrong Info*" se refiere a acciones que fallaron debido a que MATE proporcionó información errónea en el JSON. Por ejemplo, las acciones realizadas por MATE en elementos de la UI por *fuera* de la aplicación (como en la barra superior) interrumpen la compilación de los *tests* Espresso correspondientes, ya que los identificadores de recurso de esas vistas no están definidos dentro de la aplicación objetivo.

"*Accessibility Service*" representa las limitaciones conocidas del *Servicio de Accesibilidad* utilizado por MATE para recopilar información de la pantalla. Algunas de estas limitaciones llevaron a incongruencias durante la generación de *tests* Espresso: nombre de clase impreciso de las vistas, informar incorrectamente el "*hint*" de un campo como texto escrito por el usuario, falta de propiedades de "descripción de contenido" y proporcionar textos con mayúsculas y minúsculas incorrectas.

Por último, la *inestabilidad* (*flakiness*) general en algunos sujetos representa el 15,87% de las acciones fallidas.

RQ3: Las causas más comunes de falla son: falta de información en el esquema JSON, acciones realizadas en *widgets* que no se muestran al usuario, información errónea de MATE y *flakiness* general de los sujetos.

Tabla 5.5: Análisis de las causas y componentes asociados para las acciones fallidas.

Componente	Causa	Cantidad (%)	Total (%)
Espresso *code synthesizer*	*JSON Schema*	16 (25,40%)	24 (38,10%)
	Swipe Difference	5 (7,94%)	
	Ambiguous Matcher	3 (4,76%)	
Espresso *framework*	*View Not Displayed*	12 (19,05%)	19 (30,16%)
	Framework Limitation	6 (9,52%)	
	Subject Not Supported	1 (1,59%)	
MATE	*Wrong Info*	9 (14,29%)	10 (15,87%)
	Accessibility Service	1 (1,59%)	
Subject	*Flakiness*	10 (15,87%)	10 (15,87%)
Total		**63**	

5.4.3. Amenazas a la validez

Las amenazas a la validez interna pueden surgir de cómo se llevó a cabo el estudio empírico. Para la selección del algoritmo, consideramos aquellos estudiados en [98]. Decidimos utilizar el algoritmo de mejor rendimiento en ese estudio, es decir, la Exploración Aleatoria (*Random Exploration*, en inglés). Esto es consistente con nuestros resultados obtenidos en el Capítulo 4. Dado que el algoritmo de Exploración Aleatoria se ve afectado por el no determinismo, realizamos 5 repeticiones en cada sujeto con diferentes semillas aleatorias. Para evitar posibles factores de confusión al comparar diferentes algoritmos, todos se implementaron en la misma herramienta. Además, el ajuste de parámetros puede afectar el rendimiento de los algoritmos, por lo que utilizamos los mismos valores predeterminados para todos los parámetros en los experimentos. Estos valores se eligieron tomando como guía el trabajo de Sell et al. [98].

Las amenazas a la validez externa provienen del hecho de que solo utilizamos 12 sujetos de código abierto como sujetos de prueba. Para evitar sesgos de selección en los sujetos de código abierto, decidimos explícitamente incluir solo aquellas aplicaciones que ya habían sido utilizadas previamente en el estudio empírico de MATE [97], y que ya habían sido configuradas para ejecutar *tests* Espresso. No obstante, es importante tener en cuenta que otra selección de sujetos podría dar lugar a conclusiones diferentes.

5.5. Análisis y discusión

En esta sección, discutimos los principales desafíos y limitaciones de la síntesis de *tests* Espresso y presentamos los aprendizajes de este estudio. Nos enfocamos

primero en los desafíos técnicos y luego analizamos las barreras existentes que podrían impedir que los desarrolladores adopten una herramienta como la implementada en su flujo de trabajo diario.

5.5.1. Desafíos y limitaciones

Los resultados de **RQ2** (Sección 5.4.2) muestran que los estados divergentes de la UI entre una secuencia de acciones basada en *widgets* y su correspondiente *test* Espresso se deben principalmente a acciones fallidas. El resto de las causas tuvieron un impacto menor durante el estudio: problemas de sincronización, sujetos inestables (*flaky*) y problemas en la herramienta de generación de *tests*.

Los resultados de **RQ3** (Sección 5.4.2) muestran las causas observadas para las acciones fallidas. *El principal desafío detectado es la dificultad de indicar correctamente al framework Espresso la vista en la cual debe realizar una acción.* Para hacerlo, un generador de casos de *test* Espresso (o el desarrollador de la app) debe construir correctamente un *View Matcher* correspondiente. Dado que el *framework* Espresso requiere que se realice una acción en una vista identificada de manera inequívoca, este *View Matcher* debe construirse con gran precisión. Si no se hace correctamente, el *framework* Espresso lanzará una excepción.

Los *View Matchers* se pueden construir utilizando identificadores de recursos, que no siempre son únicos, y muchas otras propiedades de las vistas (por ejemplo, texto, vistas hijo, vistas padre, etc.) que pueden ayudar a reducir la búsqueda. Por lo tanto, *es de suma importancia que la información proporcionada por la herramienta subyacente de generación de tests sea abundante y precisa.* Cualquier información faltante o incorrecta tiene una alta probabilidad de causar una acción fallida.

También encontramos que *es importante que la herramienta de generación de tests que realiza la exploración ejecute las acciones en vistas que sean accesibles para Espresso.* Por ejemplo, las acciones realizadas en vistas que no están *mostradas* para el usuario (es decir, en pantalla) no funcionarán con las acciones predeterminadas de Espresso, y las acciones realizadas en vistas *fuera* de la aplicación no funcionarán en absoluto. Para mitigar esto, creamos acciones personalizadas de Espresso (por ejemplo, para *click* y *click* largo) que solo requieren que la vista de destino tenga un estado de visibilidad "visible" (es decir, visible pero no necesariamente *mostrada en pantalla*) y que tenga un ancho y alto mayor que cero (necesario porque en algunas aplicaciones las vistas se vuelven "invisibles" estableciendo su tamaño en cero).

Sin embargo, adoptar acciones personalizadas debe verse como un compromiso en lugar de una solución definitiva, ya que en algunas aplicaciones el uso de las acciones predeterminadas es correcto. Por ejemplo, la aplicación `OmniNotes` tiene una pantalla de inicio con diferentes páginas, que se pueden navegar deslizando hacia la derecha y hacia la izquierda. Dado que cada página de la pantalla tiene similitudes, los desarrolladores eligieron reutilizar el mismo botón "*Next*" en todas ellas. Como todos estos botones "*Next*" son visibles (pero no *mostrados*) al mismo tiempo, al usar las acciones personalizadas introducimos una `AmbiguosMatcherException` en algunos de los casos de *test* generados. En este caso, está perfectamente bien usar

las acciones predeterminadas, ya que solo se *muestra* uno de estos botones "*Next*" a la vez. Esto se ilustra en la segunda y tercera pantallas de Figura 2.7a, donde dos páginas diferentes del mismo formulario tienen el mismo botón "*Next*". Decidimos habilitar el uso de acciones personalizadas de manera predeterminada, ya que en general parecía mejorar los casos de *test*.

Entonces, la limitación mencionada anteriormente es un problema fundamental que surge de la semántica de las acciones en MATE y Espresso. En otras palabras, si MATE hace *click* en vistas que no están *mostradas* y tratamos de evitar esto en Espresso (es decir, mediante el uso de acciones personalizadas), podríamos provocar excepciones de *matchers* ambiguos en algunos casos. Por otro lado, si usamos las acciones predeterminadas de Espresso (con la restricción de estar *mostradas*), entonces cada vez que MATE hace *click* en una vista fuera de la pantalla, el caso de *test* Espresso correspondiente fallará.

Por último, como se mencionó anteriormente, algunos problemas son independientes del método elegido para sintetizar *tests* Espresso. Por lo general, la inestabilidad (*flakiness*) del sujeto puede generar secuencias de acciones de *widgets* que no son reproducibles. Además, las limitaciones del *framework* Espresso impiden que funcione con vistas personalizadas sin esfuerzos adicionales.

En resumen, la creación de *tests* Espresso es difícil debido a la falta de propiedades únicas para identificar inequívocamente los elementos en los diseños. Este problema se agrava en algunos casos debido a la definición incompleta o ambigua de los componentes y diseños de la interfaz de usuario. A continuación, comentamos las limitaciones técnicas particulares encontradas durante el estudio empírico que también pueden ser de interés para otros investigadores.

Limitaciones en la desambiguación de *widgets*. A pesar de las estrategias implementadas para mitigar el problema de la ambigüedad de los *widgets*, algunos casos pueden no ser posibles de desambiguar:

- *Imágenes con jerarquía idéntica pero contenido diferente.* Esto puede ocurrir, por ejemplo, si dos imágenes diferentes están una al lado de la otra. En este caso, no es posible construir un *View Matcher* que distinga ambas por el contenido (ni el *Servicio de Accesibilidad* utilizado por MATE proporciona esa información).
- *Widgets con jerarquía y contenido idénticos, pero con hermanos diferentes en la jerarquía.* Por ejemplo, la segunda pantalla en la Figura 2.7a muestra una lista corta de opciones de bebidas en la que cada fila tiene el mismo *checkbox*, pero todas las filas tienen un texto diferente. Podríamos querer ejercitar un *checkbox* específico (por ejemplo, el de la fila de "Coca-Cola"), pero si solo observamos la jerarquía de padres e hijos, como se propone en nuestro esquema JSON, no hay forma de distinguir uno del otro. Esto podría resolverse si agregamos hermanos al esquema JSON o si proporcionamos todo el árbol de jerarquía de la interfaz de usuario en el JSON.
- *Widgets con jerarquía, contenido y hermanos idénticos.* Por ejemplo, si tenemos filas duplicadas en una lista (idénticas incluido el contenido, como se muestra

en la fila de "Pepperoni" en la primera pantalla de la Figura 2.7a), entonces la única forma de desambiguarlas es utilizando su posición en la lista, pero esta información no es proporcionada por el *Servicio de Accesibilidad* y, por lo tanto, no se incluye en la salida de MATE.

Falta de transformaciones de *testing*. El concepto de *transformación de testing* [232] (*testability transformations*, en inglés) se refiere a cualquier modificación aplicada a un programa para hacer que el proceso de *testing* sea más efectivo. En ese sentido, el proceso de sintetizar automáticamente *tests* Espresso puede beneficiarse de transformaciones de *testing* personalizadas. Estas transformaciones pueden ayudar, por ejemplo, con el problema mencionado en el párrafo anterior, asegurando que todos los *widgets* tengan un identificador de recurso único. Un ejemplo de dicha transformación es presentado por Coppola et al. [173], pero es necesario un análisis adicional para medir su efectividad.

Otro escenario en el que las transformaciones de *testing* pueden ser útiles es cuando una aplicación bloquea el *thread* principal (es decir, el *thread* de la UI). Dado que las acciones ejecutadas por Espresso también se ejecutan en el *thread* principal, cualquier bloqueo allí impide que Espresso funcione. En particular, esto ocurrió en nuestro estudio con el sujeto `OneTimePad`. Esta aplicación permite a los usuarios almacenar contraseñas de un solo uso, protegidas por una contraseña maestra. Al ingresar, la aplicación le pide al usuario que ingrese la contraseña maestra. Sin embargo, para hacerlo, la aplicación abre un diálogo separado del *thread* principal y bloquea este último con un *busy loop* rodeado por un *try-catch*. El *busy loop* solo se libera cuando el diálogo se cierra, lanzando una excepción en el *thread* principal. Mientras el *thread* principal está bloqueado, Espresso no puede enviar ninguna acción a la UI de la aplicación. Aunque esta es una limitación de Espresso, creemos que el comportamiento implementado por `OneTimePad` no es estándar y no debería afectar a un rango más amplio de aplicaciones. Es interesante destacar que MATE funciona para `OneTimePad`, ya que se ejecuta como un proceso independiente en el dispositivo y, por lo tanto, en un *thread* diferente.

Limitaciones de ingeniería vs. desafíos de investigación. Hasta ahora, hemos presentado todos los desafíos y limitaciones juntos. No obstante, es útil también separar estos problemas entre aquellos incidentales y los fundamentales. Las limitaciones incidentales son aquellas que se pueden reformular como un problema de ingeniería. En otras palabras, son limitaciones encontradas como producto de la implementación elegida para el prototipo o la configuración experimental. Identificamos lo siguiente:

- Asegurar que la información recopilada durante la generación de *tests* sea correcta, para construir luego los *View Matchers* correctamente.
- Proporcionar toda la información disponible durante la generación de *tests* al *generador de código Espresso*.
- Encontrar formas de recopilar información sobre vistas personalizadas, imágenes y listas. Este problema también requeriría la implementación de *View Matchers*

personalizados.

- Minimizar errores debido a diferencias en deslizamientos y problemas de sincronización para evitar inestabilidad (*flakiness*) en general.

A modo de ejemplo, el primer elemento mencionado anteriormente podría ser facilitado mediante la mejora de MATE de manera que no realice acciones fuera de la aplicación bajo *test*, y que las realizadas dentro sean solo en vistas accesibles para Espresso. Además, sería de gran ayuda reemplazar el *Servicio de Accesibilidad* utilizado en MATE por otra fuente de información de la interfaz de usuario. Los desafíos fundamentales pueden ser reformulados como un problema de investigación. Es decir, se deben diseñar algoritmos o técnicas especiales para encontrar formas de resolver o mitigarlos. Consideramos los siguientes problemas como tales:

- Información Android insuficiente para identificar inequívocamente los elementos en los diseños de la interfaz de usuario.
- Inestabilidad (*flakiness*) del sujeto debido a estados de la UI que dependen de factores externos (por ejemplo, la hora o servicios web).
- Además, falta de *transformaciones de testing* que se puedan aplicar a las aplicaciones para mitigar ambos problemas mencionados anteriormente.

5.5.2. Adopción y utilidad

Para comprender la utilidad y las limitaciones de los *tests* Espresso sintetizados para los desarrolladores, evaluamos la efectividad del prototipo implementado en las 12 aplicaciones de código abierto previamente estudiadas, además de una aplicación industrial. Enviamos los *tests* Espresso sintetizados como *Pull Requests* para las aplicaciones de código abierto y aquí informamos sobre sus tasas de aceptación. Finalmente, proporcionamos el prototipo a un socio industrial para su evaluación. Realizamos una entrevista escrita con dos desarrolladores del equipo de Android de dicha empresa. Pedimos a estos desarrolladores que resumieran los aspectos positivos y negativos del prototipo, así como sus comentarios para mejorar la utilidad de los *tests* generados.

Proyectos de código abierto. Enviamos un *Pull Request* con un solo caso de *test* a cada repositorio de código público (por ejemplo, en GitHub) de los proyectos utilizados en este capítulo (ver Tabla 5.1). Para evitar cualquier posible sesgo, seguimos un estricto proceso de selección para decidir qué *test* enviar.

En primer lugar, comenzamos considerando solo aquellos *tests* Espresso sintetizados en la primera repetición de nuestro estudio empírico. Como las repeticiones son independientes, esto no afecta la representatividad del resultado. En segundo lugar, si el caso de *test* Espresso sintetizado con la mayor cobertura para un proyecto aumentaba la cobertura general del proyecto en un 5% o más, se seleccionaba para enviar. Cuando ningún caso de *test* lograba cumplir este requisito, verificábamos si existían *tests* Espresso que cubrieran actividades faltantes en la cobertura del *test*

suite existente del proyecto. Si ese era el caso, seleccionábamos aquel que brindara el mayor aumento de cobertura al proyecto. Si ningún caso de *test* lograba aumentar la cobertura en más del 5% ni cubría ninguna actividad nueva, decidíamos no enviar ningún *Pull Request*.

Cada *Pull Request* contenía el caso de *test* seleccionado, dos archivos JAVA auxiliares documentados, dependencias externas si fuera necesario, y una breve explicación destacando las contribuciones (es decir, el aumento general de cobertura y las nuevas pantallas cubiertas). Para los proyectos con *guías de contribución* explícitas, revisamos manualmente el código enviado para asegurarnos de que se adhiriera a ellas.

Finalmente, seleccionamos 8 de 12 proyectos para enviar *Pull Requests*. De los 8 *Pull Requests* enviados, recibimos respuestas con comentarios para la mitad de ellos (`ShoppingList`, `MicroPinner`, `KolabNotes` y `OmniNotes`). `KolabNotes` y `OmniNotes` fusionaron el *Pull Request* en su *test suite* y agradecieron la contribución. El desarrollador de `OmniNotes` incluso dijo: "*Great thanks! Code coverage is always the best way to contribute!*". El *Pull Request* enviado a `HomeAssistant` se cerró sin comentarios, y poco después el desarrollador cerró y *archivó* el proyecto. Nuestra hipótesis es que, en esta etapa inicial, *la salida del prototipo es principalmente útil para proyectos activos sin casos de test o con algunos casos de test pero baja cobertura*.

Proyecto industrial. Para evaluar la utilidad del prototipo y su rendimiento, nos comunicamos con un socio industrial. Este socio industrial tiene una aplicación móvil desarrollada para el manejo de pagos con tarjeta de crédito. La aplicación tiene más de 30 pantallas diferentes y más de 100.000 líneas de código, utilizando tanto JAVA como KOTLIN como lenguajes de programación.

En cuanto a su base de código de *tests*, el proyecto solo tiene *tests* unitarios (alrededor de 100). Aunque no miden la cobertura, sospechan que es bastante baja. El proyecto no tiene *tests* de integración ni a nivel de sistema, y tampoco utiliza el *framework* ESPRESSO.

Pudimos mostrar el prototipo a dos desarrolladores del equipo ANDROID en la empresa. Les pedimos que lo probaran y respondieran una serie de preguntas posteriormente. La evaluación se realizó de forma remota y asíncrona debido a la pandemia COVID-19 en curso en ese momento. Las preguntas se enviaron por correo electrónico. En su evaluación, los desarrolladores desactivaron el acceso a 5 de las 30 pantallas que componen la app. Esta limitación se impuso porque dichas pantallas interactúan con dispositivos Bluetooth que no están disponibles al utilizar un emulador.

Después de probar el prototipo y examinar los *tests* ESPRESSO sintetizados, informaron los siguientes aspectos positivos. En primer lugar, *para proyectos que no tienen o tienen pocos tests a nivel de sistema, proporciona una manera rápida y sencilla de aumentar su cobertura*. En otras palabras, los *tests* generados se ejecutaron con éxito y *también sirven como punto de partida para crear nuevos casos de test*. En segundo lugar, aunque no hay garantía, la Exploración Aleatoria permite que la herramienta encuentre errores ocultos que de otra manera podrían llegar a los

usuarios finales.

También destacaron las siguientes desventajas. En primer lugar, *los tests Espresso sintetizados no proporcionan una descripción clara de cuáles son los objetivos de cada caso de test*. En segundo lugar, debido al uso de la Exploración Aleatoria de MATE, los casos de *test* tienden a volverse largos (es decir, hasta 50 acciones). Estos *tests* largos se vuelven muy frágiles cuando se tiene una aplicación grande que está en constante crecimiento. *Siempre que sea posible, los casos de test deben enfocarse en un escenario de usuario específico*. Además, la Exploración Aleatoria puede causar una exploración desigual de la aplicación. Del mismo modo, estos *tests* generados aleatoriamente tienen algunas acciones redundantes que podrían eliminarse. En tercer lugar, los *tests* generados en una configuración de dispositivo deben ejecutarse en la misma configuración. Esto se debe a que las UI de Android cambian su apariencia según el tamaño, la densidad y la orientación de la pantalla.

Una vez completado el cuestionario, les preguntamos si considerarían incorporar los *tests* generados a su base de código. Respondieron que primero necesitarían tener una descripción clara para cada caso de *test*. Pero si se proporcionara eso, dijeron que agregarían todos los *tests* razonables sin hacer ningún cambio en ellos. También comentaron que al agregar esos *tests*, la cobertura del proyecto aumentaría. Es importante mencionar aquí que el problema de sumarización de *tests* es un desafío de investigación abierto y no se limita a la generación de *tests* de Android [233].

Finalmente, les preguntamos qué cambios sugerirían para mejorar la *usabilidad* (es decir, facilidad de uso por parte de los desarrolladores), la *adopción* (es decir, incorporar la herramienta al flujo de trabajo) y la *integración* (es decir, el uso regular de la herramienta) del prototipo actual. En términos de *usabilidad*, propusieron: mejorar los mensajes de error en los casos de *test*, agregar descripciones de los casos de *test*, y agregar la posibilidad de guiar la exploración de forma semiautomática (es decir, permitir el *feedback* humano durante la exploración). En términos de *adopción* e *integración*, mencionaron que es muy importante proporcionar *plugins* de integración para las herramientas que ya se utilizan a diario, por ejemplo, el entorno de desarrollo integrado Android Studio. Además, en caso de que no sea posible solucionar el problema de tener que ejecutar los *tests* en el mismo dispositivo en el que se generaron, sería bueno que el prototipo pudiera explorar casos de *test* en varios dispositivos al mismo tiempo.

5.6. Conclusiones

En este capítulo realizamos un estudio empírico para evaluar la factibilidad y los desafíos de sintetizar automáticamente *tests* de UI Espresso a partir de secuencias de acciones basadas en *widgets*. El estudio se llevó a cabo en varias aplicaciones de código abierto y un proyecto industrial. Para comprender mejor los desafíos y limitaciones de la creación automática de *tests* Espresso, el estudio fue seguido por un análisis cuantitativo y cualitativo de los resultados.

Estos resultados experimentales muestran que existen problemas fundamentales con el enfoque basado en traducción para generar *tests* Espresso. En particular, la

creación de *tests* Espresso es difícil, principalmente debido a la falta de propiedades únicas para identificar de manera inequívoca *widgets* específicos en la UI. Este problema se ve agravado por el uso directo o indirecto que muchas herramientas hacen del *Servicio de Accesibilidad* de Android, que puede devolver información inconsistente.

Aunque el enfoque basado en traducción no resultó ser prometedor, y nos lleva a explorar formas alternativas de generar *tests* Espresso, nuestra experiencia modificando MATE fue positiva, con lo que creemos que utilizar una herramienta existente como base para diseñar un prototipo experimental puede ser una buena estrategia para futuros trabajos. El prototipo utilizado para el estudio es de código abierto y se puede encontrar en GitHub [234].

6

Generación de casos de test en formato Espresso

En este capítulo presentamos una técnica para generar casos de *test* Espresso utilizando directamente el *framework* Espresso para interactuar con la aplicación bajo *test*. Este método permite generar *selectores* claros y concisos para los *widgets* de la UI. Además, permite agregar *aserciones* a los casos de *test* generados, lo que los hace más útiles para el *testing* de regresión. Presentamos una evaluación experimental de la técnica en 1.035 apps Android, mostrando mejoras significativas en la confiabilidad de los *tests* generados con respecto a los enfoques existentes en la literatura basados en traducción. Este capítulo fue aceptado para publicación como artículo de conferencia [235].

6.1. Introducción

Como se mencionó en el Capítulo 5, una posibilidad para generar casos de *test* Espresso es realizar ingeniería inversa de la salida de herramientas existentes, traduciéndola al formato Espresso [82]. Sin embargo, este enfoque de traducción se ve limitado por la ausencia de propiedades únicas que permitan identificar de manera inequívoca los *widgets* de la UI. Este problema se agrava debido a que muchas herramientas de generación de *tests* utilizan la API del *Servicio de Accesibilidad* de Android [78] para recopilar información del estado de la pantalla, lo que puede resultar en inconsistencias [82] como nombres de clase de vistas inexactos, propiedades de descripción de contenido faltantes o mayúsculas y minúsculas incorrectas. Estos desafíos impiden la generación de *tests* Espresso confiables, es decir, *tests* que no fallan erróneamente al volver a ejecutarse.

En este capítulo, proponemos una técnica novedosa para superar estas dificultades: en lugar de depender del *Servicio de Accesibilidad* de Android, presentamos una nueva representación de casos de *test* que utiliza directamente el *framework* Espresso para interactuar con una aplicación bajo *test* mediante la definición de *View Matchers* de Espresso que seleccionan de manera concisa los *widgets* de Android. Esto tiene varias ventajas: primero, el uso del formato Espresso coincide con lo que los desarrolladores desean; segundo, este enfoque aumenta la confiabilidad de los *tests* generados; tercero, a diferencia de enfoques anteriores de generación de *tests*

Android, esto permite agregar aserciones utilizando el mecanismo de *View Assertions* de Espresso. Implementamos este enfoque en EspressoMaker, una extensión de la herramienta de generación de *tests* MATE [97]. En detalle, las contribuciones de este capítulo son las siguientes:

- Presentamos un enfoque para generar *tests* Espresso utilizando *View Matchers* de Espresso.
- Introducimos un enfoque para generar aserciones de regresión utilizando el mecanismo de *View Assertions* de Espresso.
- Proporcionamos una implementación de ambos enfoques propuestos en EspressoMaker, una extensión de la herramienta MATE para generación de *tests* Android.
- Estudiamos empíricamente EspressoMaker en 1.035 aplicaciones Android en términos de confiabilidad y detección de fallas.

Los experimentos demuestran que el enfoque propuesto aumenta la confiabilidad: el 74,33% de los *tests* generados con EspressoMaker pueden ejecutarse de manera confiable, mientras que el enfoque tradicional basado en traducción solo lleva a un 24,11% de pruebas re-ejecutables. El estudio también revela que las aserciones Espresso generadas por EspressoMaker son estadísticamente mejores para detectar fallos que los *tests* sin aserciones.

La implementación de EspressoMaker es de código abierto y se puede encontrar en GitHub [236, 237]. Además, proporcionamos un paquete de replicación disponible públicamente [238] que contiene el código fuente de EspressoMaker, los scripts utilizados para ejecutar los experimentos y los datos crudos recopilados durante el estudio empírico.

6.2. Enfoque de EspressoMaker

6.2.1. Generando casos de test con MATE

Aunque en principio la pregunta de cómo generar casos de *test* Espresso es ortogonal a la pregunta de qué enfoque o herramienta de generación de *tests* utilizar, una prueba de concepto requiere adaptar un generador de *tests* concreto para utilizar las API de Espresso en lugar del *Servicio de Accesibilidad* de Android. Al igual que en el Capítulo 5, elegimos la herramienta de generación de *tests* MATE [97] para generar secuencias de interacciones sobre las aplicaciones Android, y la adaptamos para utilizar Espresso.

Internamente, cada caso de *test* en MATE se representa como una secuencia de acciones sobre vistas disponibles de la aplicación bajo *test*. MATE genera una representación XML de los casos de *test* y un informe sobre el número de casos de *test* ejecutados, *crashes* encontrados y cobertura alcanzada. Para generar casos de *test*, MATE (al igual que otros generadores de *tests* Android como Stoat o Dynodroid) requiere realizar las siguientes tareas:

Tabla 6.1: View Actions de Espresso utilizadas en EspressoMaker.

View Action	**Descripción**
`pressBack()`	Hace *click* en el botón "atrás".
`clearText()`	Borra el texto en la vista.
`click()`	Hace *click* en la vista (es decir, toque simple).
`pressKey(ENTER)`	Presiona la tecla de hardware "*enter*".
`longClick()`	Hace *click* prolongado en la vista (es decir, *click* largo).
`pressMenuKey()`	Presiona la tecla de hardware "*menú*".
`pressKey(SEARCH)`	Presiona la tecla de hardware "*búsqueda*".
`swipeDown()`	Realiza un deslizamiento de arriba hacia abajo en el centro horizontal de la vista.
`swipeLeft()`	Realiza un deslizamiento de derecha a izquierda en el centro vertical de la vista.
`swipeRight()`	Realiza un deslizamiento de izquierda a derecha en el centro vertical de la vista.
`swipeUp()`	Realiza un deslizamiento de abajo hacia arriba en el centro horizontal de la vista.
`typeText(String)`	Selecciona la vista (haciendo *click* en ella) y escribe un texto proporcionado en la vista.

- Dado el estado actual de la aplicación, obtener la lista de todas las acciones disponibles.
- Dada una acción seleccionada de las disponibles en la aplicación, ejecutar dicha acción.
- (Opcionalmente) Luego de ejecutar una acción, contrastar y verificar que el estado actual de la aplicación sea el esperado.

Todas estas tareas se realizan de manera independiente al algoritmo subyacente para generar los casos de *test*. En otras palabras, es posible identificar estas tareas en generadores de *tests* basados en búsquedas (por ejemplo, MATE), generadores de *tests* basados en modelos (por ejemplo, Stoat) o generadores probabilísticos (por ejemplo, Dynodroid).

6.2.2. Generando *tests* Espresso

Una tarea fundamental en la generación automatizada de *tests* es recolectar adecuadamente las acciones disponibles en el estado actual de la aplicación bajo *test*. Dada la pantalla actual, nos referiremos por "*información de pantalla*" a los siguientes elementos:

- Jerarquía de Vista de la pantalla actual, que también incluye las propiedades de cada vista, como el identificador de recurso, texto, y descripción del contenido.
- Las acciones Espresso disponibles para cada vista en la Jerarquía de Vista.

- El paquete y el nombre de la actividad de la aplicación que se muestra en la pantalla. Esto es necesario para saber cuándo un generador de *tests* ha "*abandonado*" la aplicación bajo *test* (por ejemplo, apretando el botón de *back*).

Desafortunadamente, el *framework* Espresso no proporciona una forma directa de obtener los dos primeros elementos. Sin embargo, aún podemos acceder a la Jerarquía de Vista utilizando "*reflection*" sobre las clases internas del *framework* Espresso [83]. En programación, "*reflection*" refiere a la habilidad de un programa para examinar y manipular su propia estructura, clases, métodos, y atributos en tiempo de ejecución [239]. Provee una forma de inspeccionar y modificar elementos de código dinámicamente, sin conocer sus nombres en tiempo de compilación. También permite acceder a métodos o clases privadas, que no son accesibles desde fuera de la clase. Más detalles de la implementación de EspressoMaker se pueden encontrar en la Sección 6.3.

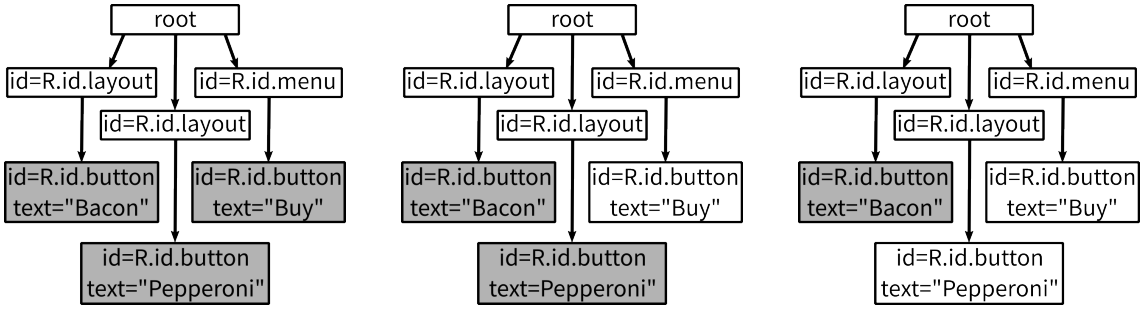
Para obtener las acciones disponibles para una vista dada, iteramos sobre todas las acciones Espresso enumeradas en la Tabla 6.1. Filtramos esta lista comprobando si cada acción se puede realizar en la vista, llamando al método `getConstraints()` en la acción. Este método permite que las *View Actions* especifiquen restricciones que las vistas deben cumplir para considerarse como objetivos para esa acción. Por ejemplo, la acción `click()` requiere que la vista se muestre en la pantalla (es decir, `isDisplayed()`).

6.2.3. Generación de *View Matchers* Espresso

Una vez que se obtienen todas las vistas y acciones disponibles, el generador de *tests* aún necesita crear *View Matchers* en Espresso para cada vista objetivo. Los *View Matchers* son cruciales ya que cumplen el doble propósito de permitir la ejecución de las *View Actions* y las *View Assertions*. Estos *View Matchers* deben seleccionar de manera inequívoca la vista objetivo. Un *View Matcher* es **inequívoco** si hay una y solo una vista en la Jerarquía de Vistas que coincide con él. Si el *View Matcher* no apunta a ninguna vista o apunta más de una vista, la ejecución del *test* fallará.

Construir este tipo de *View Matchers* para una vista objetivo puede no ser siempre factible debido a la falta de propiedades únicas para localizar de manera inequívoca la vista. Este problema se menciona en la literatura relacionada como *desambiguación de vistas* [82, 171, 172, 174]. Los *View Matchers* se definen utilizando identificadores de recursos junto con otras propiedades de la vista (como texto, vistas hijas y vistas padre, etc.). A menudo, la vista objetivo tiene un identificador de recurso único que se puede utilizar para ubicarla de manera inequívoca (es decir, usando el `withId` *matcher* de Espresso). Sin embargo, hay muchos escenarios legítimos en los que una vista objetivo puede no tener un identificador de recurso o este identificador puede no ser único. Por ejemplo, el escenario en el que las filas de una lista se representan utilizando un documento XML (ver la Figura 2.7b). En dicho escenario, todas las filas comparten el mismo identificador de recurso [240].

Dada una vista objetivo, nuestra técnica para generar un *View Matcher* inequívoco



(a) Al usar solo la restricción R_1, la conjunción ($C = R_1$) coincide con tres vistas diferentes en la jerarquía.

(b) Al agregar la restricción R_2 a la conjunción ($C = R_1 \wedge R_2$) se refina la selección a solo dos vistas diferentes.

(c) Finalmente, al agregar la restricción R_3 ($C = R_1 \wedge R_2 \wedge R_3$) se reduce la selección a solo una vista en la jerarquía.

Figura 6.1: Combinando iterativamente *Restricciones de Propiedad de Vista* para seleccionar una vista objetivo en la Jerarquía de Vistas. Cada caja representa una vista en la Jerarquía de Vistas. Las cajas grises indican las vistas que coinciden con la conjunción de restricciones, mientras que las cajas blancas indican las vistas que no coinciden. Las restricciones utilizadas en este ejemplo son: R_1 = *WITH_ID(v, R.id.button)*, R_2 = *WITH_ID(v.parent, R.id.layout)* y R_3 = *WITH_TEXT(v, "Bacon")*

en Espresso comienza identificando un conjunto de restricciones que solo son satisfechas por la vista objetivo. En otras palabras, estas restricciones no deben cumplirse por ninguna otra vista en la Jerarquía de Vistas. Por ejemplo, en la Figura 3.2, la vista objetivo v cumple con la restricción $v.resourceIdentifier == R.id.button$ al mismo tiempo que cumple con la restricción $v.parent.resourceIdentifier == R.id.layout$.

En cada una de estas restricciones, podemos encontrar los siguientes elementos: hay una vista a la que se puede acceder navegando la Jerarquía de Vistas a partir de la vista objetivo (por ejemplo, v y $v.parent$), y hay una propiedad de la vista navegada (por ejemplo, el identificador de recurso) que debe coincidir con un valor dado (por ejemplo, *R.id.button* y *R.id.layout*).

Definimos una **Restricción de Propiedad de Vista** (*View Property Constraint*, en inglés) como una restricción en una propiedad de una vista v utilizando un valor c. Por ejemplo, la restricción *WITH_ID(v, c)* se define como $v.resourceIdentifier == c$. Notar que $v.resourceIdentifier == R.id.button$ se puede reescribir entonces como *WITH_ID(v, R.id.button)*. Luego, el *View Matcher* en la Figura 3.2 para la vista objetivo v se puede expresar como una conjunción $R_1(v) \wedge R_2(v)$ de restricciones de propiedad de vista, tales que:

$$R_1(v) = WITH\_ID(v,\ R.id.button)$$
$$R_2(v) = WITH\_ID(v.parent,\ R.id.layout)$$

$R_1(v)$ indica que la vista objetivo debe tener el valor de *R.id.button* como identificador de recurso. Además, $R_2(v)$ indica que el padre de la vista objetivo debe

Algoritmo 10: Algoritmo para generar una conjunción inequívoca de *Restricciones de Propiedades de Vistas*

Input : Vista v, Árbol jerárquico de vistas t
Output : Conjunción de Restricciones de Propiedades de Vistas C

```
1  if v does not have parent do
2      return IS_ROOT(v)
3  C ← True
4  for View v' ∈ t do
5      path ← PathToView(v, v', t)
6      for Constraint P ∈ BASIC_CONSTRAINT_TYPES do
7          if type is valid for v' do
8              value ← GetValue(v', P)
9              C ← C ∧ P(path, value)
10             if C is unequivocal do
11                 return GetMinimalCombination(C)
12 return Null
```

tener el valor de *R.id.layout* como identificador de recurso. Aunque tanto $R_1(v)$ como $R_2(v)$ están parametrizados por la vista objetivo v, la vista en la que se verifica la restricción difiere en cada caso. En $R_1(v)$, la restricción se verifica en la vista objetivo v, mientras que en $R_2(v)$, la restricción se verifica en la vista padre de v. Por lo tanto, cuando se fija la vista objetivo (por ejemplo, un v dado), usamos directamente la ruta de "navegación" como el primer argumento de la restricción de propiedad de vista.

En resumen, las restricciones de propiedad de vista representan los bloques básicos que se utilizarán más adelante para construir el *View Matcher* Espresso. Si la conjunción de restricciones es *inequívoca*, habrá una y solo una vista en la jerarquía que cumpla con todas las restricciones simultáneamente. Por lo tanto, al agregar restricciones de manera iterativa a una conjunción, nuestro objetivo es construir una expresión que identifique de manera inequívoca una vista objetivo en la Jerarquía de Vistas. Esto se ilustra en la Figura 6.1.

El Algoritmo 10 describe los pasos para generar conjunciones de *Restricciones de Propiedad de Vista* inequívocas. Este algoritmo comienza verificando si la vista de entrada v es la raíz de la Jerarquía de Vistas t. En ese caso, simplemente retornamos la restricción `IS_ROOT`. De lo contrario, iteramos sobre todas las vistas en la Jerarquía de Vistas t (incluyendo v). Para cada vista v' en t, determinamos el camino que debe recorrerse en t desde v hasta v'. Por ejemplo, un camino de navegación podría indicar que para ir de v a v', necesitamos ir "hacia arriba" al padre de v y luego ir "hacia abajo" al primer hijo del padre.

Agregamos una restricción a C por cada *tipo básico de restricción* (nombre de recurso, identificador de recurso, texto, descripción del contenido y nombre de clase).

Cada restricción se agrega solo si la propiedad que verifica está definida para v'. Por ejemplo, una restricción de texto no se agregará para una vista sin texto como una imagen. El valor con el que las restricciones se compararán es el valor real observado para cada propiedad en v'. Finalmente, se devuelve C una vez que encontramos una conjunción *inequívoca*. Si no existe tal conjunción, devolvemos *Null*.

Una vez que se encuentra una conjunción de *Restricciones de Propiedad de Vista inequívoca*, la *minimizamos*. Una conjunción es *minimal* cuando eliminar cualquier restricción resulta en que coincida con más de una vista en la Jerarquía de Vistas (es decir, deja de ser inequívoca). Esta minimización se realiza mediante el uso de *Delta Debugging* [241]. La técnica de *Delta Debugging* consiste en eliminar iterativa e inteligentemente partes de un programa para encontrar un programa más pequeño que aún exhiba el mismo comportamiento. En nuestro caso, el programa es la conjunción de restricciones y el comportamiento deseado es que la conjunción sea inequívoca. Si bien este proceso de minimización no es necesario para que el *View Matcher* funcione correctamente, es útil para generar *View Matchers* más concisos y legibles. De otra forma, el *View Matcher* podría contener (potencialmente cientos de) restricciones redundantes que no son necesarias para seleccionar la vista objetivo.

Luego de hallar con éxito una conjunción de *Restricciones de Propiedad de Vista minimizada inequívoca*, la traducimos en un correspondiente *View Matcher* de Espresso. El Algoritmo 11 describe los pasos para realizar esta tarea. El algoritmo comienza creando un "*AllOf*" *View Matcher*: un *matcher* recursivo que selecciona una vista solo si coincide con todos los *matchers* internos especificados.

Para cada *Restricción de Propiedad de Vista* en la conjunción, la función `AppendInnerMatcher` (i.e., "agregar *matcher* interno") construye de forma recursiva el *View Matcher* de Espresso que satisface la ruta y el tipo de la restricción, y lo agrega donde sea necesario. Como ejemplo, consideremos la conjunción de *Restricciones de Propiedad de Vista* $R_1(v) \wedge R_2(v)$, tal que

$$R_1(v) = \mathit{WITH\_ID(v.parent,\ R.id.button)}$$
$$R_2(v) = \mathit{WITH\_TEXT(v.parent,\ "Bacon")}$$

Primero, el algoritmo ejecuta `AppendInnerMatcher` en R_1, produciendo el *View Matcher* parcial de Espresso:

```
allOf(withParent(withId(R.id.button)))
```

Luego, el algoritmo ejecuta `AppendInnerMatcher` en R_2, produciendo el *View Matcher* final de Espresso:

```
allOf(withParent(allOf(
    withId(R.id.button), withText("Bacon"))))
```

La Tabla 6.2 enumera todos los *View Matchers* de Espresso utilizados por el algoritmo.

Algoritmo 11: Generación de *View Matcher* Espresso

Input : Vista v, Conjunción de Restricciones de Propiedades de Vistas C
Output : *View Matcher* Espresso M para seleccionar v

Function *BuildEspressoViewMatcher*(v, C)
  $M \leftarrow$ AllOf([])
  for *Constraint* $c \in C$ **do**
    $type \leftarrow$ GetType(c)
    $path \leftarrow$ GetPath(c)
    AppendInnerMatcher$(M, v, type, path)$
  return M
Function *AppendInnerMatcher*$(M, v, type, path)$
  if *path is empty* **do**
    if $type =$ *WITH_RESOURCE_NAME* **do**
      $newMatcher \leftarrow$ WithResourceName(GetResourceName(v))
    if $type =$ *WITH_ID* **do**
      $newMatcher \leftarrow$ WithId(GetId(v))
    if $type =$ *WITH_TEXT* **do**
      $newMatcher \leftarrow$ WithText(GetText(v))
    if $type =$ *WITH_CONTENT_DESCRIPTION* **do**
      $newMatcher \leftarrow$ WithContentDescription(GetContDesc(v))
    if $type =$ *WITH_CLASS_NAME* **do**
      $newMatcher \leftarrow$ WithClassName(GetClassName(v))
    $M.matchers$.Append$(newMatcher)$
  else
    $nextStep \leftarrow$ Head$(path)$
    if $nextStep =$ *MOVE_TO_PARENT* **do**
      if $M.matchers$ *does not contain a* $WithParent$ *matcher* **do**
        $newMatcher \leftarrow$ WithParent()
        $M.matchers$.Append$(newMatcher)$
      else
        $newMatcher \leftarrow$ GetParentMatcher$(M.matchers)$
    else
      if $M.matchers$ *does not contain a* $WithChild$ *matcher* **do**
        $newMatcher \leftarrow$ WithChild()
        $M.matchers$.Append$(newMatcher)$
      else
        $newMatcher \leftarrow$ GetChildMatcher$(M.matchers)$
    $remainingPath \leftarrow$ Tail$(path)$
    $w \leftarrow$ GetViewAfterStep$(v, nextStep)$
    AddMatcherForPathAndType$(newMatcher, w,$
      $type, remainingPath)$
  return M

Algoritmo 12: Generación de *View Assertions* Espresso

```
   Input  : Caso de test tc, Nivel de aserción L
   Output : Lista de aserciones A
 1 A ← []
 2 if L == None do
 3     return A
 4 Start AUT
 5 lastScreen ← ∅
 6 for Action a ∈ tc do
 7     executeAction(a)
 8     currentScreen ← GetScreenInfo()
 9     if L == Full ∨ (L == SemiFull ∧ lastScreen == ∅) do
10         for View v ∈ GetAllViews(currentScreen) do
11             for attr ∈ getAttributes(v) do
12                 A.append(
13                     buildAttributeAssertion(v, attr))
14     else
15         screenDiff ← computeScreenDiff(currentScreen, lastScreen)
16         for View v ∈ GetCommonViews(screenDiff) do
17             for attr ∈ getAttributesDiff(screenDiff, v) do
18                 A.append(
19                     buildAttributeAssertion(v, attr))
20         for View v ∈ GetDisappearingViews(screenDiff) do
21             A.append(DoesNotExist(v))
22         for View v ∈ GetAppearingViews(screenDiff) do
23             A.append(IsDisplayed(v))
24             if L == SemiFull do
25                 for attr ∈ getAttributes(v) do
26                     A.append(
27                         buildAttributeAssertion(v, attr))
28     lastScreen ← currentScreen
29 return A
```

Tabla 6.2: View Matchers Espresso utilizados en EspressoMaker

View Matcher	**Descripción**
`isRoot()`	La vista es la raíz de la Jerarquía de Vistas.
`withId(Integer)`	La vista tiene un identificador específico.
`withResourceName(String)`	La vista tiene un nombre de recurso específico.
`withText(String)`	La vista tiene un texto específico.
`withContentDescription(String)`	La vista tiene una descripción de contenido específica.
`withClassName(String)`	La vista tiene un nombre de clase específico.
`allOf(ViewMatcher...)`	La vista coincide con *todos* los *matchers* especificados.
`withChild(ViewMatcher...)`	La vista coincide solo si una de las vistas hijo directas coincide con los *matchers* especificados.
`withParent(ViewMatcher...)`	La vista coincide solo si su vista padre coincide con los *matchers* especificados.

6.2.4. Generación de *View Assertions* Espresso

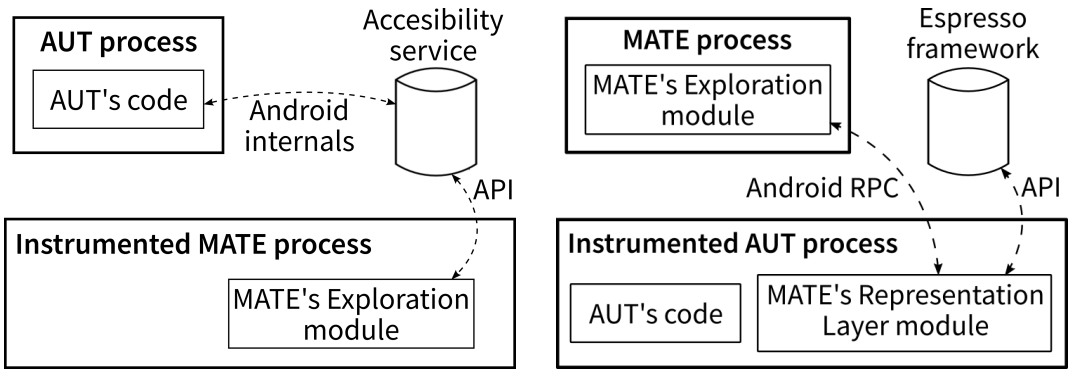
En el contexto de *testing*, una *aserción* es una declaración que verifica si una propiedad específica del estado del programa coincide con su valor esperado. Si el valor actual difiere, la aserción falla el caso de *test*. El *framework* Espresso utiliza aserciones para asegurar que una vista seleccionada se encuentra en el estado deseado. Por ejemplo, en la Figura 3.2 se muestra una aserción que chequea que un botón tiene el texto "*Clicked*" luego de haber sido presionado.

Presentamos un algoritmo de generación de aserciones para Espresso (Algoritmo 12) basado en el algoritmo de Xie [242], el cual consiste en registrar los valores de todos los atributos observables después de cada acción en el *test*, e insertar aserciones de regresión que capturen dichos valores. Es importante mencionar que el enfoque convencional de generar aserciones para *todos* los atributos de todas las vistas y luego filtrar las aserciones no relevantes mediante análisis de mutación [35] (*mutation analysis*, en inglés) no es viable para la generación de aserciones Espresso debido al alto número de vistas y atributos en una aplicación Android, y al costoso tiempo de ejecución de los *tests* Android, sumado a las múltiples ejecuciones requeridas.

Nuestro algoritmo propuesto (ver Algoritmo 12) ejecuta las acciones del *test* una a la vez, determinando en cada paso qué aserciones agregar al *test* basándose en la Jerarquía de Vista actual y la previa, utilizando los valores observados como valores esperados en las aserciones. Este algoritmo puede configurarse en cuatro niveles

Tabla 6.3: View Assertions de Espresso utilizadas en EspressoMaker.

View Assertion	**Descripción**
`doesNotExist()`	La vista no existe en la Jerarquía de Vistas.
`isDisplayed()`	La vista se muestra al menos un 90% en la pantalla.
`isEnabled()`	La vista está habilitada. La interpretación del estado habilitado varía según la subclase.
`isNotEnabled()`	La vista no está habilitada.
`isFocused()`	La vista está enfocada (es decir, el usuario interactúa directamente con la vista).
`isNotFocused()`	La vista no está enfocada.
`isFocusable()`	La vista es enfocable (es decir, puede recibir el enfoque).
`isNotFocusable()`	La vista no es enfocable.
`hasFocus()`	La vista tiene el enfoque. Esto significa que la vista o uno de sus descendientes está enfocado.
`doesNotHaveFocus()`	La vista no tiene el enfoque.
`isSelected()`	La vista está seleccionada. Esto no es lo mismo que el enfoque, se refiere al estado seleccionado en el contexto de una lista o similar (por ejemplo, la vista está resaltada).
`isNotSelected()`	La vista no está seleccionada.
`isChecked()`	La vista está marcada. Esto solo puede ser cierto para vistas "*checkable*" como un `CheckBox`.
`isNotChecked()`	La vista no está marcada.
`isClickable()`	La vista es clickeable (es decir, la vista acepta y reacciona a eventos de clic/tap).
`isNotClickable()`	La vista no es clickeable.
`hasLinks()`	La vista es un `TextView` y contiene URLs.
`hasContentDescription()`	La descripción de contenido de la vista no es nula (puede ser una cadena vacía).
`withText(String)`	La vista tiene un texto específico.
`hasErrorText(String)`	La vista tiene un texto de error específico.
`withContentDescription(String)`	La vista tiene una descripción de contenido específica.
`withHint(String)`	La vista tiene una pista (hint) específica.
`withAlpha(Float)`	La vista tiene una opacidad específica. Este es un valor de 0 a 1, donde 0 significa que la vista es completamente transparente y 1 significa que la vista es completamente opaca.
`hasChildCount(Integer)`	La vista tiene un número específico de hijos en la Jerarquía de Vistas. Este valor siempre es 0 si la vista no es un `ViewGroup`.
`withInputType(Integer)`	La vista es un objeto `Editable` y tiene un tipo específico de contenido básico (por ejemplo, texto, número, contraseñas, etc.).
`withParentIndex(Integer)`	La vista es un hijo con el índice específico en su padre.
`withVisibility(Enum)`	La vista tiene un estado de `Visibility` específico. Los valores posibles son *Visible*, *Invisible*, y *Gone*. *Invisible* significa que la vista no es visible, pero aún ocupa espacio en la pantalla. *Gone* significa que la vista es invisible y no ocupa ningún espacio.



(a) Vieja arquitectura de MATE.. *(b)* Nueva arquitectura de MATE.

Figura 6.2: Refactorización de los módulos de MATE. La nueva arquitectura permite a EspressoMaker acceder directamente al *framework* de Espresso y obtener la Jerarquía de Vista y las acciones disponibles para cualquier pantalla dada.

diferentes de generación de aserciones: *Full*, *SemiFull*, *DiffOnly*, y *None*.

- El nivel *Full* genera aserciones para todos los atributos de todas las vistas en la Jerarquía de Vistas (es decir, ignorando la Jerarquía de Vistas previa). Aunque esto usualmente resulta en un gran número de aserciones (posiblemente no relevantes), sirve como base para la comparación con los otros niveles.
- El nivel *DiffOnly* genera aserciones solo para atributos (como texto) que cambian en vistas comunes entre pantallas consecutivas. También agrega aserciones cuando las vistas aparecen (o desaparecen) de la UI. La razón detrás de este nivel es que las aserciones más relevantes son aquellas que verifican partes de la UI que se actualizan dinámicamente. Este enfoque reduce significativamente el número de aserciones generadas, pero también puede resultar en la omisión de aserciones para atributos que no cambian a lo largo de la ejecución de un caso de *test*.
- El nivel *SemiFull* tiene como objetivo mitigar este problema agregando aserciones para todos los atributos de vistas que aparecen en la UI (es decir, las vistas no presentes en la Jerarquía de Vistas previa), al mismo tiempo que mantiene las aserciones generadas en el nivel *DiffOnly*.
- Finalmente, el nivel *None* no genera ninguna aserción.

La Tabla 6.3 lista las *View Assertions* Espresso generadas por el algoritmo.

6.3. Implementación de EspressoMaker

Los algoritmos presentados en la Sección 6.2 fueron implementados como una extensión de MATE. Nos referimos a esta extensión como EspressoMaker a lo largo del capítulo. La implementación de EspressoMaker está disponible públicamente en GitHub [236, 237].

La implementación original de MATE utiliza un proceso *instrumentado* (ver Sección 3.1) para comunicarse con la API del *Servicio de Accesibilidad* de ANDROID y recopilar el estado actual de la UI de la aplicación bajo *test*. En esta configuración original, la aplicación bajo *test* se ejecuta como un proceso separado sin instrumentación. Esta configuración se muestra en la Figura 6.2a.

Sin embargo, para utilizar el *framework* de ESPRESSO, este último debe: (1) ejecutarse en el mismo proceso que la aplicación, y (2) ejecutarse como un proceso instrumentado de ANDROID. Ambos requisitos son necesarios para que ESPRESSO pueda acceder a la Jerarquía de Vista de la aplicación bajo *test* y ejecutar acciones sobre ella. Si ESPRESSO se ejecuta en un proceso instrumentado pero separado, no puede acceder a la Jerarquía de Vista de la aplicación bajo *test*. Si ESPRESSO se ejecuta en el mismo proceso que la aplicación pero sin instrumentación, no puede ejecutar acciones sobre la Jerarquía de Vista de la aplicación bajo *test*, debido a restricciones de seguridad de la plataforma ANDROID. Es importante además mencionar que los procesos instrumentados están pensados para interactuar con una sola aplicación objetivo. Esto significa que no es posible iniciar un proceso instrumentado desde otro también instrumentado si el nombre de paquete es distinto. Por último, si el código a ejecutar en el proceso instrumentado no se encuentra en el mismo APK que el código de la aplicación, ambos APK deben estar firmados con la misma firma.

Refactorizamos la arquitectura de MATE para incluir un nuevo módulo llamado *Capa de Representación* (*Representation Layer*, en inglés) que se ejecuta en el proceso instrumentado de la aplicación. Este módulo utiliza el *framework* de ESPRESSO internamente e informa a la capa de exploración de MATE sobre la Jerarquía de Vista y las acciones disponibles. Dado que ambos módulos se ejecutan ahora en procesos separados, dependen del Lenguaje de Definición de Interfaces de ANDROID (*Android Interface Definition Language* o AIDL, en inglés) [243] para la comunicación entre procesos mediante llamadas a procedimientos remotos (*Remote Procedure Call* o RPC, en inglés). Se muestra una ilustración general de la nueva arquitectura en la Figura 6.2b. Concretamente, hay entonces 3 APKs que son necesarios cuando utilizamos la nueva arquitectura de MATE:

- **APK de la aplicación bajo *test*:** contiene el código de la aplicación bajo *test*. Dependiendo del tipo de experimento, este APK puede estar modificado para recopilar información adicional como datos de cobertura al ejecutar la aplicación. Además, la firma de este APK debe coincidir con la del APK de MATE para el módulo de la capa de representación (se puede cambiar utilizando la herramienta `apksigner` provista por ANDROID).
- **APK de MATE para el módulo de exploración:** contiene el código de MATE encargado de la exploración. Tiene la definición de, por ejemplo, todos los algoritmos evolutivos que se pueden utilizar para explorar la aplicación bajo *test*.
- **APK de MATE para el módulo de la capa de representación:** contiene el código de MATE encargado de obtener la información de la Jerarquía de

```
// Create a new instance of the RootsOracle class
Class rootsOracleClass = Class.forName("androidx.test.
    espresso.base.RootsOracle");
Constructor<ActiveRootLister> rootsOracleConstructor =
    rootsOracleClass.getDeclaredConstructor(new Class[]{
    Looper.class});
rootsOracleConstructor.setAccessible(true);
ActiveRootLister sRootsOracle = rootsOracleConstructor.
    newInstance(Looper.getMainLooper());

// Call the method listActiveRoots
Method sRootsOracle_listActiveRoots = rootsOracleClass.
    getMethod("listActiveRoots");
List<Root> roots = (List) sRootsOracle_listActiveRoots.
    invoke(sRootsOracle);
```

Listado 6.1: Código JAVA para obtener la Jerarquía de Vista de la aplicación bajo *test*.

Algoritmo 13: Algoritmo para obtener las acciones disponibles en una pantalla.

Input : Jerárquía de vistas t
Output : Conjunto de *View Actions* A

$A \leftarrow \emptyset$
for *View* $v \in t$ **do**
  if *We can build a unequivocal View Matcher for* v **do**
    for *Action* $a \in DEFAULT\_ACTIONS$ **do**
      if $a.getConstraints().matches(v)$ **do**
        $A \leftarrow A \cup \text{VIEWACTION}(v, a)$
return A

Vista y las acciones disponibles para cualquier pantalla dada. Este APK se ejecuta como un proceso instrumentado de ANDROID. Es importante que el "`targetPackage`" [244] definido en el `AndroidManifest.xml` de este APK coincida con el nombre de paquete de la aplicación bajo *test*. En particular, al crear un APK utilizando el *ANDROID Gradle Plugin*, el nombre del "`targetPackage`" se define automáticamente como el mismo que el nombre de paquete del proyecto [245, 246], que podría no coincidir con el nombre de paquete de la aplicación bajo *test*. Por lo tanto, es necesario cambiar el nombre del "`targetPackage`" en el `AndroidManifest.xml` de este APK.

6.3.1. Capa de representación

Como mencionamos en la Sección 6.2.2, los principales desafíos técnicos que

debe resolver la capa de representación son: (1) obtener la Jerarquía de Vista de la aplicación bajo *test*, y (2) obtener las acciones disponibles para cualquier pantalla dada. Para obtener la Jerarquía de Vista, este módulo llama al método `listActiveRoots` de la clase `RootsOracle` [83] por "*reflection*". Este método es el mismo que utiliza ESPRESSO para obtener la Jerarquía de Vista de la aplicación bajo *test*. Es necesario usar "*reflection*" porque `RootsOracle` es una clase final en el *framework* de ESPRESSO y no puede ser instanciada directamente. El código utilizado se muestra en el Listado 6.1.

El método `listActiveRoots` devuelve una lista de objetos `Root`, que representan las raíces de las distintas Jerarquías de Vista que se muestran en pantalla. Cada objeto `Root` contiene una referencia a la Jerarquía de Vista correspondiente, que puede ser accedida mediante el método `getDecorView`. Es importante notar que puede haber más de una Jerarquía de Vista activa en un momento dado. Por ejemplo, si la aplicación bajo *test* muestra un *diálogo* (*dialog*, en inglés) o un *menú emergente* (*pop-up menu*, en inglés), la Jerarquía de Vista de la pantalla principal de la aplicación seguirá estando activa. Luego, esta lista de raíces es filtrada, quitando aquellas que no son visibles para el usuario, que se encuentran debajo de otra Jerarquía de Vista, o que pertenecen a otra aplicación que no es la aplicación bajo *test*. Si hubiera más de una Jerarquía de Vista activa para la aplicación bajo *test*, se selecciona la que esté más arriba en la jerarquía de ventanas de ANDROID (es decir, la que tenga el *z-index* más alto).

Una vez que obtenemos la Jerarquía de Vista de la aplicación bajo *test* sobre la que vamos a trabajar, podemos obtener las acciones disponibles. Este proceso se muestra en el Algoritmo 13. Para cada vista v en la Jerarquía de Vista, se intenta construir un *View Matcher* que sea inequívoco para v. Si se puede construir un *View Matcher* inequívoco, se itera sobre las acciones por defecto (ver Tabla 6.1) y se verifica si la vista objetivo cumple con las restricciones de la acción (por ejemplo, ESPRESSO no permite hacer *click* en un elemento de la UI que no está al menos 95% mostrado en la pantalla). Si la vista objetivo cumple con las restricciones de la acción, se agrega la acción al conjunto de acciones disponibles.

6.3.2. Módulo de exploración

El módulo de exploración de MATE es el encargado de, como su nombre lo indica, explorar la aplicación bajo *test*. Contiene las definiciones de los distintos algoritmos de exploración, y es el encargado de ejecutarlos. Este módulo define además un componente llamado MATE `Service`, que es el punto de entrada de MATE bajo la nueva arquitectura. La tarea principal de este servicio es de ejecutar el algoritmo de exploración seleccionado por el usuario, y de levantar y configurar el módulo de representación para que pueda comunicarse con el módulo de exploración cada vez que sea necesario.

6.4. Evaluación

Con el fin de investigar la dificultad de crear *tests* Espresso, así como la efectividad de la técnica propuesta, buscamos responder empíricamente las siguientes preguntas:

RQ1: (Motivación) *¿Son los identificadores de recursos suficientemente únicos para construir Espresso View Matchers?*

RQ2: (Confiabilidad) *¿Son los tests Espresso generados por EspressoMaker más confiables que los generados con un enfoque basado en traducción?*

RQ3: (Aserciones) *¿Son las aserciones Espresso generadas por EspressoMaker efectivas para detectar fallas?*

6.4.1. Configuración experimental

Selección de sujetos de prueba. Evaluamos EspressoMaker en un subconjunto del conjunto de datos de AndroZoo [220]. Inicialmente, descargamos 7.000 APKs seleccionados al azar y realizamos una verificación de salud de cada aplicación para excluir aquellas que no eran funcionales o incompatibles con nuestra configuración experimental. Esta verificación de salud consistió en ejecutar la herramienta Monkey [84] durante 1 minuto y verificar si se producían *crashes*. Además, eliminamos aquellas aplicaciones que no pudieron ser instrumentadas por la instrumentación de *bytecode* de MATE [247], también utilizada en EspressoMaker. Este proceso resultó en 1.035 APKs.

Procedimiento del experimento. Utilizamos la implementación de EspressoMaker para generar *tests* Espresso para las 1,035 aplicaciones en nuestro experimento. Elegimos el algoritmo de *Exploración Aleatoria* de MATE para generar casos de *test*. Aunque *Exploración Aleatoria* es un algoritmo bastante simple, se ha demostrado que es uno de los algoritmos más efectivos para la generación de pruebas en Android [98]. Limitamos el presupuesto de generación a 10 casos de *test* (es decir, individuos) por sujeto, con un máximo de 15 acciones (por ejemplo, *click*, *click* largo, escribir texto, etc.) permitidas para cada *test*. La *Exploración Aleatoria* funciona muestreando continuamente el espacio de búsqueda hasta que se agota el presupuesto o se alcanza el número máximo de acciones. La generación de un individuo también se detiene cuando el algoritmo de generación sale de la UI de la aplicación bajo *test*. En cada iteración, se crea un individuo completamente nuevo. Los experimentos fueron totalmente automatizados, sin intervención manual (por ejemplo, inicio de sesión) durante la ejecución de MATE. Los resultados para **RQ1** se basan en los estados de UI y acciones exploradas por EspressoMaker durante el proceso de generación de *tests*.

Para **RQ2**, utilizamos el enfoque basado en traducción propuesto en el Capítulo 5, también implementado en MATE. Comparamos la confiabilidad de los *tests* Espresso generadas por EspressoMaker con los *tests* generados por traducción. Al igual

que en el Capítulo 5, consideramos que un *test* es "confiable" si se puede ejecutar con éxito (es decir, no falla). Cabe destacar que los *tests* ESPRESSO generados por el enfoque basado en traducción no contienen aserciones.

Para **RQ3**, consideramos los APKs de AndroZoo utilizados en **RQ2** para los cuales pudimos generar al menos un *test* ESPRESSO exitoso (sin fallos) (926 aplicaciones). Utilizamos MUTAPK [248] para generar mutantes de estas aplicaciones. MUTAPK toma como entrada un APK y produce un conjunto de APK modificados aplicando un conjunto de operadores de mutación predefinidos. En nuestro experimento, limitamos los operadores de mutación solo a aquellos que tienen el potencial de cambiar la UI de la aplicación, ya que las aserciones en los *tests* ESPRESSO se basan en el estado de la UI. Descartamos los operadores de mutación que conducen a mutantes triviales (como el operador `ActivityNotDefined`, que solo puede provocar un *crash* en la AUT) y mutantes no observables (por ejemplo, operadores que no cambian la UI de la AUT, como `InvalidActivityName`). Para este proceso, inspeccionamos manualmente cada operador de mutación disponible en MUTAPK. Los operadores finales considerados en nuestro experimento fueron: `DifferentActivityIntentDefinition`, `WrongMainActivity`, `MissingPermissionManifest`, `WrongStringResource`, `InvalidURI`, `NullGPSLocation`, `InvalidDate`, `ViewComponentNotVisible`, y `InvalidViewFocus`. Dada la gran cantidad de mutantes (222 en promedio por aplicación), muestreamos aleatoriamente 10 mutantes por aplicación. También consideramos MUDROID [249] y DROIDMUTATOR [250] para la generación de mutantes. La implementación de MUDROID se encuentra obsoleta y no pudimos generar mutantes con ella. DROIDMUTATOR requiere el código fuente de las aplicaciones, pero el conjunto de datos AndroZoo solo contiene binarios.

Los experimentos se llevaron a cabo en un clúster compuesto por 12 nodos, cada uno equipado con 2 CPUs Intel Xeon E5-2650 v2 (8 núcleos cada una) y 128 GB de RAM. Seleccionamos ANDROID Lollipop (API 21) como la plataforma objetivo para nuestros experimentos. Para medir la significancia estadística, utilizamos el ampliamente reconocido U-test de Wilcoxon-Mann-Whitney y el tamaño del efecto $\hat{A}_{12}$ de Vargha-Delaney [251].

6.4.2. Amenazas a la validez

Las amenazas a la validez interna podrían surgir de cómo se llevó a cabo el estudio empírico. Nuestro esfuerzo de implementación fue considerable, por lo que comparamos diferentes herramientas y elegimos aquella que mejor se adaptaba a nuestras necesidades. Optamos por extender MATE para implementar ESPRESSO-MAKER debido a su madurez (mantenido activamente) y nuestra experiencia previa trabajando con él. Sin embargo, las técnicas presentadas en este capítulo no están atadas a MATE y pueden aplicarse a otras herramientas de generación de *tests* siempre que ejecuten los casos de *test* generados (por ejemplo, en un emulador o dispositivo). Utilizamos el algoritmo de *Exploración Aleatoria*, ya que un estudio previo muestra que logra los mejores resultados [98] (similar a los resultados del Capítulo 4). Notar que la representación de los *tests* es independiente del algoritmo

utilizado. Para evitar posibles factores de confusión al comparar EspressoMaker con el enfoque basado en traducción, ambos se implementaron en la misma herramienta (MATE). Dado que la configuración de parámetros puede afectar el rendimiento de los algoritmos [212], utilizamos los mismos valores predeterminados para todos los parámetros en todos los experimentos. Para **RQ3**, nuestro objetivo fue mitigar el sesgo utilizando la herramienta MutAPK para el análisis de mutaciones, y seleccionando aleatoriamente 10 mutantes para cada aplicación. Utilizamos el análisis de mutaciones como un indicador de detección de fallas. Aunque el análisis de mutaciones es una técnica bien establecida, los mutantes artificiales pueden no ser representativos de defectos reales.

Las amenazas a la validez externa pueden surgir de cómo se llevó a cabo la selección de sujetos. Para mitigar esta amenaza, descargamos aleatoriamente 7.000 APKs del conjunto de datos AndroZoo [220], excluyendo solo aquellos que eran no funcionales o que eran incompatibles con nuestra configuración experimental. Otra selección de sujetos podría dar resultados diferentes.

6.4.3. Resultados

RQ1: ¿Son los identificadores de recursos suficientemente únicos para construir Espresso *View Matchers*? EspressoMaker encontró 4.387.217 vistas mientras generaba *tests* Espresso para las 1.035 aplicaciones en nuestro experimento. De este total, solo el 61,90% tenía un identificador de recurso asignado por los desarrolladores, mientras que el 38,10% no lo tenía. La plataforma Android asigna un identificador de recurso predeterminado de -1 a las vistas que no tienen uno. Por otro lado, solo el 46,00% de las vistas tenían un identificador de recurso inequívoco. Esto significa que el 54,00% de las vistas necesitan un *View Matcher* no trivial para realizar una acción Espresso en ellas.

EspressoMaker pudo generar una combinación inequívoca de restricciones de propiedades para el 92,45% de todas las vistas encontradas. Para el restante 7,55%, inspeccionamos manualmente las aplicaciones afectadas y encontramos que estas fallas fueron causadas por *vistas gemelas*: vistas hermanas (es decir, hijas del mismo padre en la Jerarquía de Vista) con atributos idénticos, que no pueden ser desambiguadas agregando información adicional sobre sus padres (ya que tienen el mismo padre) o hijos (porque no tienen vistas hijos o porque los hijos también son idénticos). En principio, este problema podría resolverse utilizando el *View Matcher* `withParentIndex` de Espresso en el Algoritmo 11, permitiendo a EspressoMaker especificar el índice de un hijo en su vista padre. Dejamos esto como trabajo futuro.

RQ 1: Más de la mitad de las vistas en nuestros experimentos no tienen un identificador de recurso inequívoco, lo que requiere un *View Matcher* no trivial para realizar una acción de Espresso en ellas.

RQ2: ¿Son los *tests* Espresso generados por EspressoMaker más confiables que los generados con un enfoque basado en traducción? EspressoMaker generó un total de 11.049 *tests* Espresso, mientras que el enfoque basado

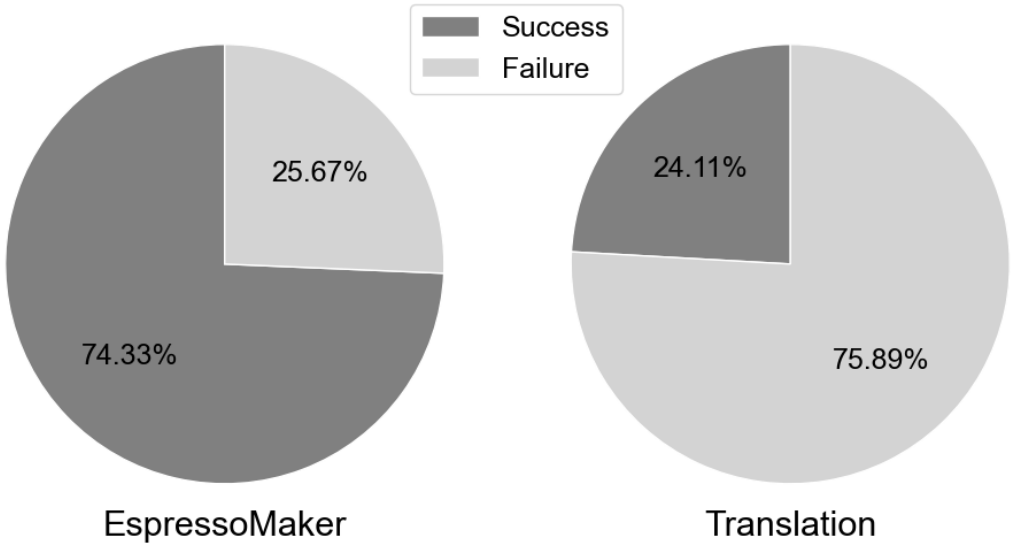


Figura 6.3: Porcentaje de casos de *test* ejecutados con éxito para **RQ2**.

en traducción generó 11.255. La diferencia en el número de casos de *test* se debió a un número menor de *timeouts* que ocurrieron durante la ejecución de EspressoMaker. Sin embargo, el total de *tests* generados por ambos enfoques sigue siendo muy similar, resaltando la robustez de nuestros resultados. La Figura 6.3 ilustra los resultados de la ejecución de *tests* para ambos enfoques. De los 11.049 casos de *test* Espresso generados por EspressoMaker, 8.213 (74,33%) se ejecutaron con éxito, mientras que solo 2.714 (24,11%) se ejecutaron con éxito para el enfoque basado en traducción. La diferencia es estadísticamente significativa según el resultado del U-*test* de Wilcoxon-Mann-Whitney para un $\alpha = 0{,}05$ y $p < 0{,}001$, y un tamaño de efecto de Vargha-Delaney *grande* de $\hat{A}_{12} = 0{,}82$.

RQ 2: Los *tests* Espresso generados por EspressoMaker son significativamente más confiables que los generados mediante traducción.

RQ3: ¿Son los *tests* Espresso generados por EspressoMaker más confiables que los generados con un enfoque basado en traducción? Comparamos los *scores* de mutación [252] (es decir, mutantes detectados sobre mutantes evaluados) de los casos de *test* Espresso con aserciones de nivel *Full* y sin aserciones. Dado que los *tests* inestables (*flaky*) pueden afectar los resultados del análisis de mutación, volvimos a ejecutar los *tests* 3 veces para identificar y eliminar dichos *tests*, resultando en 8.252 casos de *test* (no *flaky*).

La Figura 6.4 muestra los *scores* de mutación alcanzados por los *tests* Espresso utilizando niveles de aserción *Full* y *None*, en un total de 8.785 mutantes. Los casos de *tests* con aserciones *Full* lograron un *score* de mutación promedio del 19,09% (1.703 mutantes eliminados), mientras que aquellos sin aserciones lograron un 13,45% (1.202 mutantes eliminados). Esta diferencia es estadísticamente significativa ($p < 0{,}001$, $\hat{A}_{12} = 0{,}57$). Es importante notar que los mutantes detectados por casos de *test*

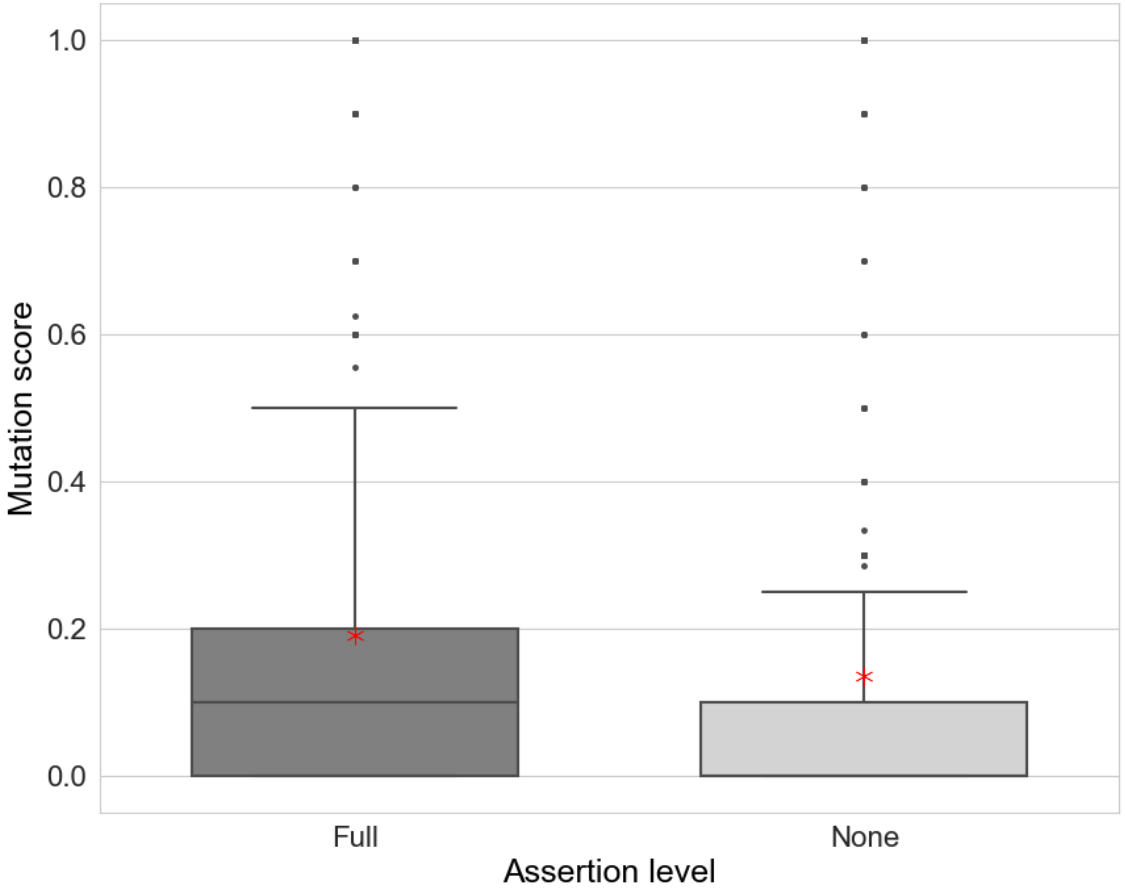


Figura 6.4: Scores de mutación para los niveles de aserciones *Full* (*Full*, en inglés) y *None* (*None*, en inglés).

sin aserciones se deben a fallas o cambios en la UI que generan errores en los *tests*. A diferencia de los *tests* unitarios, los *tests* de UI dependen de selectores, como los *View Matchers* de Espresso, para ubicar e interactuar con elementos de la UI. Estos selectores son sensibles a los cambios en la UI, generando errores cuando las condiciones especificadas ya no se cumplen.

Dado que el nivel de aserción *Full* puede generar un gran número de aserciones, también estudiamos la efectividad de los otros niveles de aserción restantes. Encontramos que *SemiFull* detectó el 89,08% de los mutantes eliminados por *Full*, mientras que *DiffOnly* y *None* detectaron el 71,29% y el 63,30%, respectivamente. Al mismo tiempo, mientras que los *tests* sin aserciones tienen una longitud fija de entre 1 a 15 sentencias, dependiendo del número de acciones realizadas por el *test*, el número promedio de aserciones aumenta exponencialmente por nivel de aserción: 29 para *DiffOnly*, 262 para *SemiFull*, y 1.157 para *Full*. Por lo tanto, los diferentes niveles de aserción introducen un compromiso entre la longitud de las *test suites* y su capacidad para detectar fallas. Dada la gran cantidad de aserciones generadas por *Full* (que pueden hacer que los *tests* sean difíciles de mantener) y la capacidad de detección de fallas de las aserciones *SemiFull* y *DiffOnly*, utilizar el nivel *SemiFull* parece ser una

buena elección práctica.

RQ 3: Los *tests* generados por EspressoMaker detectan significativamente más mutantes al incluir aserciones.

Para entender mejor los resultados del análisis de mutación, realizamos a continuación una inspección de dichos resultados en más detalle. En particular, los resultados de la **RQ3** muestran un *score* de mutación bastante bajo. Nótese que, para que un caso de *test* detecte un mutante, debe ocurrir lo siguiente: (1) la parte mutada del programa debe ejecutarse, (2) la ejecución de la falla debe cambiar el estado del programa y (3) el nuevo comportamiento exhibido por el mutante debe observarse ya sea con una aserción o con un *crash*.

Desafortunadamente, es común que los operadores de mutación en MutAPK trabajen a nivel de XML (por ejemplo, el operador `WrongStringResource`), que no se puede mapear directamente al código fuente. Esto dificulta medir exactamente cuántos mutantes fueron ejecutados (1), ya que no siempre está claro qué parte de la aplicación se ve afectada por la mutación y si la parte mutada es realmente ejecutada por los *tests*. En nuestros experimentos, el 85,34% de los mutantes fueron producidos por operadores de mutación XML. Analizamos el 14,66% restante de los mutantes producidos por operadores de mutación de código. Al comparar las trazas de ejecución de los *tests* Espresso generados y la ubicación del código binario modificado en los mutantes, encontramos que el código mutado de 65,07% de los mutantes no fue alcanzado por ninguno de los casos de *test* durante nuestros experimentos. Es importante destacar que nuestro experimento consistió únicamente en generar un conjunto comparativamente pequeño de *tests*, en lugar de configurar el experimento para maximizar completamente la cobertura de los *tests*, por lo que se espera una baja cobertura de mutantes. Sin embargo, los resultados muestran que en principio las aserciones generadas por EspressoMaker son efectivas, con lo que aumentar la cobertura de los *tests* conducirá a *scores* de mutación más altos. Además, aunque descartamos operadores de mutación que conducen a mutantes triviales o no observables, algunos mutantes generados aún pueden ser equivalentes a las aplicaciones originales; esto es válido en general siempre que se utilice análisis de mutación.

Para entender el impacto de (3), comparamos los mutantes creados por MutAPK contra las aserciones generadas por EspressoMaker. Los cinco tipos principales de mutantes (que comprenden el 98,11% del total) fueron: `WrongStringResource` (63,34%), `MissingPermissionManifest` (20,27%), `ViewComponentNotVisible` (5,10%), `InvalidViewFocus` (4,99%) y `DifferentActivityIntentDefinition` (4,41%). La Tabla 6.3 muestra los diferentes tipos de aserciones generadas por el nivel de aserción *Full*. La aserción `withText` es capaz de detectar mutantes `WrongStringResource`, la aserción `withVisibility` es capaz de detectar mutantes `ViewComponentNotVisible` y las aserciones `hasFocus`, `isFocused` y `isSelected` son capaces de detectar mutantes `InvalidViewFocus`. La aserción `isDisplayed` es capaz de detectar mutantes `MissingPermissionManifest` y `DifferentActivityIntentDefinition`, que pueden producir cambios de UI más grandes, así como mutantes `ViewComponentNotVisible`. Así, las aserciones generadas por Espresso-

Maker son adecuadas para detectar los cambios de UI introducidos por los mutantes de MutAPK. Dado que los operadores de mutación de MutAPK se derivan de una taxonomía de fallas reales en Android [253], es probable que las aserciones generadas por EspressoMaker también sean útiles para detectar fallas reales, pero son necesarios experimentos adicionales para confirmar esta hipótesis.

6.5. Conclusiones

En este capítulo, presentamos una técnica para generar casos de *test* Espresso confiables para aplicaciones Android. En particular, introducimos algoritmos novedosos para generar *View Matchers* de Espresso que seleccionan de manera concisa *widgets* de Android, y para crear *View Assertions* de Espresso que sirven para *tests* de regresión. Esta técnica fue implementada en EspressoMaker, una extensión de la herramienta de *testing* MATE para generación de *tests* Android. La implementación de EspressoMaker es de código abierto y está disponible públicamente [236, 237]. En resumen, este capítulo proporciona las siguientes ideas:

- El *framework* Espresso puede ser utilizado directamente dentro de una herramienta de generación de *tests* Android para recopilar las vistas y acciones disponibles en cualquier pantalla dada.
- Se pueden generar automáticamente *View Matchers* de Espresso concisos y confiables para seleccionar vistas en una interfaz de usuario, combinando iterativamente *Restricciones de Propiedades de Vistas*.
- Las *View Assertions* de Espresso para *tests* de regresión se pueden obtener al comparar las *View Properties* de la aplicación bajo *test* antes y después de la ejecución de una acción.

Realizamos un estudio empírico exhaustivo en 1.035 aplicaciones Android para comparar la confiabilidad de los *tests* Espresso generados por EspressoMaker con aquellos generadas por un enfoque basado en traducción. También medimos la eficacia de las *View Assertions* de Espresso generadas automáticamente en la detección de fallas. Los resultados de este estudio muestran que EspressoMaker genera *tests* Espresso significativamente más confiables a lo largo de muchas aplicaciones diferentes. También demuestran que EspressoMaker genera aserciones de Espresso que son estadísticamente mejores para detectar fallas que los *tests* sin aserciones. En particular, presentamos dos niveles de aserciones, *SemiFull* y *DiffOnly*, que reducen significativamente el número de aserciones, identificando aún una gran parte de los mutantes no equivalentes detectados. El código fuente, los *scripts* y los datos crudos recopilados durante el estudio empírico se pueden encontrar en un paquete de replicación disponible públicamente [238].

7

Conclusiones

En esta tesis nos propusimos mejorar la generación automática de casos de *test* en formato Espresso para aplicaciones Android. A continuación listamos las principales contribuciones de esta tesis.

En primer lugar, realizamos un estudio empírico de la herramienta Sapienz para entender sus fortalezas y debilidades. En este estudio encontramos que los algoritmos evolutivos (incluso en su versión multiobjetivo) no son adecuados para generar casos de *test* en el contexto de Android. En particular, no obtuvimos confianza estadística para distinguir este tipo de algoritmos de aquellos puramente aleatorios como *Random Search*. Esto se debe principalmente a que la ejecución de casos de *test* en Android es muy costosa en tiempo, y por lo tanto los algoritmos evolutivos no logran ejecutar suficientes para encontrar una solución de buena calidad.

También examinamos 101 artículos de la literatura de *testing* de Android y categorizamos cada uno según el tipo de salida proporcionada. El resultado de este análisis fue que solo unas pocas herramientas se enfocan en el *framework* Espresso como formato de salida. De hecho, una gran parte de ellas no proporcionan una salida ejecutable sino solamente informes de fallos. Las pocas herramientas que generan casos de *test* ejecutables tienden a utilizar un formato personalizado (por ejemplo, porque utilizan un motor personalizado para ejecutarlos) o el *framework* Robotium (que está deprecado y ya no recibe mantenimiento).

Luego, realizamos un estudio empírico analizando los desafíos y limitaciones para generar casos de *test* en formato Espresso. Para ello, utilizamos un enfoque basado en traducción que permite transformar la representación interna de las herramientas de *testing* automático existentes para aplicaciones Android a casos de *test* Espresso. Encontramos que la creación de *tests* Espresso es difícil, principalmente debido a la falta de propiedades únicas para identificar de manera inequívoca *widgets* específicos en la interfaz de usuario. Además, encontramos que la utilización del *Servicio de Accesibilidad* de Android puede llevar a errores en la generación de casos de *test* Espresso. Específicamente, puede devolver información inconsistente, como *nombre de clase* impreciso para las vistas, *hint* incorrecto en los campos de texto, falta de propiedades de *descripción de contenido* y proporcionar textos con mayúsculas incorrectas. Esto es problemático ya que muchas herramientas del estado del arte, o bien usan directamente el *Servicio de Accesibilidad*, o bien acceden a él indirectamente a través del *framework* UIAutomator.

Finalmente, la tesis presenta EspressoMaker, una herramienta innovadora para

generación automática de casos de *test* Espresso que combina los aprendizajes de todos los estudios empíricos realizados en esta tesis. Por un lado, EspressoMaker utiliza un algoritmo puramente aleatorio para generar casos de *test* Espresso que, como mostramos en el Capítulo 4, son más simples e igual de efectivos que los algoritmos evolutivos. Por otro lado, EspressoMaker utiliza directamente el *framework* Espresso para obtener información e interactuar con la aplicación bajo *test*, evitando así los problemas de utilizar el *Servicio de Accesibilidad* de Android que mostramos en el Capítulo 5. Al mismo tiempo, EspressoMaker incluye algoritmos novedosos para generar *View Matchers* de Espresso que seleccionan de manera concisa *widgets* de Android, y para crear *View Assertions* de Espresso que sirven para *tests* de regresión. Estos algoritmos, sumado a lo mencionado anteriormente, le permiten a EspressoMaker generar casos de *test* Espresso que son ampliamente más confiables que los generados utilizando el enfoque basado en traducción. La implementación de EspressoMaker es de código abierto y se encuentra disponible públicamente [236, 237].

Es importante destacar que EspressoMaker es la primera herramienta de generación automática de casos de *test* Android que produce casos de *test* en formato Espresso que son robustos e incluyen aserciones de Espresso para detectar fallas de regresión. Este aspecto la hace novedosa y la diferencia de otras herramientas de generación de *tests* Android existentes en la literatura. En definitiva, EspressoMaker constituye un avance significativo en el campo de la generación aleatoria de casos de *test* Espresso. El trabajo realizado contribuye a cerrar la brecha entre las herramientas existentes de *testing* automático y los desarrolladores Android que requieren *tests* en formato Espresso. Creemos que los aprendizajes de este trabajo son útiles para la comunidad de investigación de Android y aplicables a otras herramientas de generación de *tests*.

Como trabajo futuro, es fundamental desarrollar algoritmos que puedan ser más eficientes que los puramente aleatorios para la generación de casos de *test* Espresso. Si bien los algoritmos aleatorios son simples y efectivos, es posible que existan algoritmos más eficientes que puedan generar casos de *test* Espresso más rápido y con mayor cobertura de código. Por ejemplo, MIO [254] (*Many Independent Objective*) es un algoritmo no poblacional basado en búsqueda que se ha utilizado con éxito para generar casos de *test* en el contexto de APIs RESTful [255], donde la ejecución de casos de *test* también es muy costosa en tiempo. En esta misma línea de investigación, nos gustaría también analizar el uso de genes *motif* para *tests* Espresso, ya que han demostrado ser efectivos para aumentar la cobertura de los casos de *test* generados por Sapienz. Adicionalmente, sería interesante explorar la generación automática de aserciones Espresso mediante la detección dinámica de posibles *invariantes* de la UI en las aplicaciones bajo *test* [256]. Por último, es necesario analizar y mejorar la legibilidad de los *tests* Espresso generados. Hemos dado pasos iniciales en esta dirección al utilizar *Delta Debugging* para minimizar los *View Matchers* Espresso generados, pero se podría explorar otras técnicas como el uso de *Large Language Models* (LLM) para generar casos de prueba más legibles [257].

Bibliografía

[1] G. Myers, C. Sandler, T. Badgett, and a. O. M. C. Safari, *The Art of Software Testing, 3rd Edition*. John Wiley & Sons, 2011.

[2] "Global Mobile Trends 2023," https://data.gsmaintelligence.com/research/research/research-2023/global-mobile-trends-2023, (Accessed on 06/30/2023).

[3] "Global Digital Future in Focus 2018 - Comscore, Inc." https://www.comscore.com/Insights/Presentations-and-Whitepapers/2018/Global-Digital-Future-in-Focus-2018, (Accessed on 06/30/2023).

[4] "Number of Android applications on the Google Play store," https://www.appbrain.com/stats/number-of-android-apps, (Accessed on 06/30/2023).

[5] M. L. Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing," *CoRR*, vol. abs/1801.06267, 2018.

[6] N. C. Borle, M. Feghhi, E. Stroulia, R. Greiner, and A. Hindle, "Analyzing the effects of test driven development in GitHub," *Empir. Softw. Eng.*, vol. 23, no. 4, pp. 1931–1958, 2018.

[7] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *AST*. ACM, 2011, pp. 77–83.

[8] G. P. Picco, C. Julien, A. L. Murphy, M. Musolesi, and G. Roman, "Software engineering for mobility: reflecting on the past, peering into the future," in *FOSE*. ACM, 2014, pp. 13–28.

[9] K. Moran, M. L. Vásquez, and D. Poshyvanyk, "Automated GUI testing of Android apps: from research to practice," in *ICSE (Companion Volume)*. IEEE Computer Society, 2017, pp. 505–506.

[10] Y. Wang and Y. Alshboul, "Mobile security testing approaches and challenges," in *2015 First Conference on Mobile and Secure Services (MOBISECSERV)*, 2015, pp. 1–5.

[11] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian," in *ISSRE*. IEEE Computer Society, 2010, pp. 249–258.

[12] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps," in *ICST*. IEEE Computer Society, 2014, pp. 183–192.

[13] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *ICSE*. ACM, 2014, pp. 1013–1024.

[14] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: a case study of Android game apps," in *SIGSOFT FSE*. ACM, 2014, pp. 610–620.

[15] M. Nayebi, B. Adams, and G. Ruhe, "Release Practices for Mobile Apps - What do Users and Developers Think?" in *SANER*. IEEE Computer Society, 2016, pp. 552–562.

[16] H. Muccini, A. D. Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *AST*. IEEE Computer Society, 2012, pp. 29–35.

[17] "Instrumentation | Android Developers," https://developer.android.com/reference/android/app/Instrumentation.html, (Accessed on 06/07/2023).

[18] "RobotiumTech/robotium: Android UI Testing," https://github.com/RobotiumTech/robotium, (Accessed on 06/07/2023).

[19] "Appium: Mobile App Automation Made Awesome." http://appium.io/, (Accessed on 06/07/2023).

[20] "calabash/calabash-android: Automated Functional testing for Android using cucumber," https://github.com/calabash/calabash-android, (Accessed on 06/07/2023).

[21] "monkeyrunner | Android Developers," https://developer.android.com/studio/test/monkeyrunner, (Accessed on 06/07/2023).

[22] "Write automated tests with UI Automator | Android Developers," https://developer.android.com/training/testing/other-components/ui-automator, (Accessed on 06/07/2023).

[23] "Espresso | Android Developers," https://developer.android.com/training/testing/espresso, (Accessed on 06/07/2023).

[24] "Robolectric," http://robolectric.org/, (Accessed on 08/03/2020).

[25] L. Cruz, R. Abreu, and D. Lo, "To the attention of mobile software developers: guess what, test your app!" *Empirical Software Engineering*, vol. 24, no. 4, pp. 2438–2468, 2019.

[26] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real Challenges in Mobile App Development," in *ESEM*. IEEE Computer Society, 2013, pp. 15–24.

[27] K. S. Said, L. Nie, A. A. Ajibode, and X. Zhou, "GUI testing for mobile applications: objectives, approaches and challenges," in *Internetware*. ACM, 2020, pp. 51–60.

[28] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet? (E)," in *ASE*. IEEE Computer Society, 2015, pp. 429–440.

[29] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case," in *ICSE-SEIP*. IEEE Computer Society, 2017, pp. 253–262.

[30] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of Android test generation tools in industrial cases," in *ASE*. ACM, 2018, pp. 738–748.

[31] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated Testing of Android Apps: A Systematic Literature Review," *IEEE Trans. Reliab.*, vol. 68, no. 1, pp. 45–66, 2019.

[32] T. Su, J. Wang, and Z. Su, "Benchmarking automated GUI testing for Android against real-world bugs," in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 119–130.

[33] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.

[34] "Niveles de la pirámide de pruebas | Aspectos básicos de las pruebas | Desarrolladores de Android | Android Developers," https://developer.android.com/training/testing/fundamentals?hl, (Accessed on 12/22/2023).

[35] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *ISSTA*. ACM, 2010, pp. 147–158.

[36] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, pp. 31–45, 1982.

[37] A. J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Softw. Test. Verification Reliab.*, vol. 7, no. 3, pp. 165–192, 1997.

[38] R. L. Sauder, "A general test data generator for COBOL," in *Proceedings of the May 1-3, 1962, spring joint computer conference*. Association for Computing Machinery, 1962, pp. 317–323.

[39] "Android | Do More With Google on Android Phones & Devices," https://www.android.com/, (Accessed on 12/11/2023).

[40] "Platform architecture | Android Developers," https://developer.android.com/guide/platform, (Accessed on 07/16/2023).

[41] "Mobile | UI Design | Android Developers," https://developer.android.com/design/ui/mobile, (Accessed on 07/16/2023).

[42] "Developer guides | Android Developers," https://developer.android.com/guide, (Accessed on 07/16/2023).

[43] "Documentation | Android Developers," https://developer.android.com/docs, (Accessed on 07/16/2023).

[44] "Aplicaciones de Android en Google Play," https://play.google.com, (Accessed on 07/16/2023).

[45] "Distribution dashboard | Platform | Android Developers," https://developer.android.com/about/dashboards, (Accessed on 07/16/2023).

[46] "Android OS version market share over time," https://www.appbrain.com/stats/top-android-sdk-versions, (Accessed on 07/16/2023).

[47] "Platform architecture | Android Developers," https://developer.android.com/guide/platform, (Accessed on 12/22/2023).

[48] "Download Android Studio & App Tools - Android Developers," https://developer.android.com/studio, (Accessed on 07/16/2023).

[49] "Layouts | Android Developers," https://developer.android.com/develop/ui/views/layout/declaring-layout, (Accessed on 07/20/2023).

[50] "Jetpack Compose UI App Development Toolkit - Android Developers," https://developer.android.com/jetpack/compose, (Accessed on 07/20/2023).

[51] "React," https://react.dev/, (Accessed on 07/20/2023).

[52] "Android Gradle Plugin | Android Developers," https://developer.android.com/build, (Accessed on 07/24/2023).

[53] "Gradle Build Tool," https://gradle.org/, (Accessed on 07/24/2023).

[54] "Application fundamentals | Android Developers," https://developer.android.com/guide/components/fundamentals, (Accessed on 07/16/2023).

[55] "Introduction to activities | Android Developers," https://developer.android.com/guide/components/activities/intro-activities, (Accessed on 07/16/2023).

[56] "Material Components | Mobile | Android Developers," https://developer.android.com/design/ui/mobile/guides/components/material-overview, (Accessed on 07/16/2023).

[57] "The activity lifecycle | Android Developers," https://developer.android.com/guide/components/activities/activity-lifecycle, (Accessed on 07/16/2023).

[58] "Fragments | Android Developers," https://developer.android.com/guide/fragments, (Accessed on 07/16/2023).

[59] "Services overview | Android Developers," https://developer.android.com/guide/components/services, (Accessed on 07/16/2023).

[60] "Content providers | Android Developers," https://developer.android.com/guide/topics/providers/content-providers, (Accessed on 07/16/2023).

[61] "Content provider basics | Android Developers," https://developer.android.com/guide/topics/providers/content-provider-basics, (Accessed on 07/16/2023).

[62] "Broadcasts overview | Android Developers," https://developer.android.com/guide/components/broadcasts, (Accessed on 07/16/2023).

[63] "App manifest overview | Android Developers," https://developer.android.com/guide/topics/manifest/manifest-intro, (Accessed on 12/11/2023).

[64] "Fundamentals of testing Android apps | Android Developers," https://developer.android.com/training/testing/fundamentals, (Accessed on 07/19/2023).

[65] "JUnit framework for Java unit testing," https://junit.org, (Accessed on 06/07/2023).

[66] "dtmilano/AndroidViewClient: Android ViewServer and ADB client," https://github.com/dtmilano/AndroidViewClient, (Accessed on 07/20/2023).

[67] "Testing your Compose layout | Jetpack Compose | Android Developers," https://developer.android.com/jetpack/compose/testing, (Accessed on 07/21/2023).

[68] "xiaocong/uiautomator: Python wrapper of Android uiautomator test tool." https://github.com/xiaocong/uiautomator, (Accessed on 07/20/2023).

[69] "Build instrumented tests | Android Developers," https://developer.android.com/training/testing/instrumented-tests, (Accessed on 06/07/2023).

[70] "AndroidJUnitRunner | Android Developers," https://developer.android.com/training/testing/instrumented-tests/androidx-test-libraries/runner, (Accessed on 07/21/2023).

[71] "BDD Testing & Collaboration Tools for Teams | Cucumber," https://cucumber.io/, (Accessed on 07/20/2023).

[72] "Gherkin Reference - Cucumber Documentation," https://cucumber.io/docs/gherkin/reference/, (Accessed on 07/20/2023).

[73] "WebDriver," https://www.w3.org/TR/webdriver2/, (Accessed on 07/20/2023).

[74] "appium/appium-android-driver: Android Driver for Appium," https://github.com/appium/appium-android-driver, (Accessed on 07/21/2023).

[75] "appium/appium-uiautomator2-driver: Appium driver for Android UIAutomator2," https://github.com/appium/appium-uiautomator2-driver, (Accessed on 07/21/2023).

[76] "appium/appium-espresso-driver: Espresso integration for Appium," https://github.com/appium/appium-espresso-driver, (Accessed on 07/21/2023).

[77] "appium/appium-xcuitest-driver: Appium iOS driver, backed by Apple XCTest," https://github.com/appium/appium-xcuitest-driver, (Accessed on 07/21/2023).

[78] "AccessibilityService | Android Developers," https://developer.android.com/reference/android/accessibilityservice/AccessibilityService, (Accessed on 06/07/2023).

[79] "UiAutomation class | Android Open Source Project's mirror at GitHub," https://github.com/aosp-mirror/platform_frameworks_base/blob/main/core/java/android/app/UiAutomation.java, (Accessed on 07/21/2023).

[80] "UiAutomator's project | Android Open Source Project's mirror at GitHub," https://github.com/aosp-mirror/platform_frameworks_base/tree/main/cmds/uiautomator, (Accessed on 07/21/2023).

[81] "UiAutomator's QueryController | Android Open Source Project's mirror at GitHub," https://github.com/aosp-mirror/platform_frameworks_base/blob/main/cmds/uiautomator/library/core-src/com/android/uiautomator/core/QueryController.java, (Accessed on 07/21/2023).

[82] I. A. Moreno, J. P. Galeotti, C. Ciccaroni, and J. M. Rojas, "On the feasibility and challenges of synthesizing executable Espresso tests," in *AST@ICSE*. ACM/IEEE, 2022, pp. 92–102.

[83] "RootsOracle class | android/android-test | GitHub," https://github.com/android/android-test/blob/main/espresso/core/java/androidx/test/espresso/base/RootsOracle.java, (Accessed on 07/24/2023).

[84] "UI/Application Exerciser Monkey | Android Developers," https://developer.android.com/studio/test/other-testing-tools/monkey, (Accessed on 06/07/2023).

[85] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," in *ESEC/SIGSOFT FSE*. ACM, 2013, pp. 224–234.

[86] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," in *ISSTA*. ACM, 2016, pp. 94–105.

[87] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *ASE*. IEEE Computer Society, 2017, pp. 16–26.

[88] "F8 2018: Friction-Free Fault-Finding with Sapienz," https://developers.facebook.com/videos/f8-2018/friction-free-fault-finding-with-sapienz/, 2018.

[89] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying Search Based Software Engineering with Sapienz at Facebook," in *SSBSE*, vol. 11036. Springer, 2018, pp. 3–45.

[90] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *ESEC/SIGSOFT FSE*. ACM, 2017, pp. 245–256.

[91] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, "An Introduction to MCMC for Machine Learning," *Mach. Learn.*, vol. 50, no. 1-2, pp. 5–43, 2003.

[92] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of Android applications via model abstraction and refinement," in *ICSE*. IEEE / ACM, 2019, pp. 269–280.

[93] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing," in *ASE*. IEEE, 2019, pp. 1070–1073.

[94] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "ComboDroid: generating high-quality test inputs for Android apps via use case combinations," in *ICSE*. ACM, 2020, pp. 469–480.

[95] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of Android apps," in *ICSE*. ACM, 2020, pp. 481–492.

[96] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *ISSTA*. ACM, 2020, pp. 153–164.

[97] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, "Automated Accessibility Testing of Mobile Apps," in *ICST*. IEEE Computer Society, 2018, pp. 116–126.

[98] L. Sell, M. Auer, C. Frädrich, M. Gruber, P. Werli, and G. Fraser, "An Empirical Evaluation of Search Algorithms for App Testing," in *ICTSS*, vol. 11812. Springer, 2019.

[99] "AndroidX Test Releases | Android Developers," https://developer.android.com/jetpack/androidx/releases/test, (Accessed on 06/07/2023).

[100] "Espresso basics | Android Developers," https://developer.android.com/training/testing/espresso/basics, (Accessed on 07/19/2023).

[101] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes," in *ICST*. IEEE Computer Society, 2016, pp. 33–44.

[102] J. C. J. Keng, L. Jiang, T. K. Wee, and R. K. Balan, "Graph-aided directed testing of Android applications for checking runtime privacy behaviours," in *AST@ICSE*. ACM, 2016, pp. 57–63.

[103] Y. Hu and I. Neamtiu, "Fuzzy and cross-app replay for smartphone apps," in *AST@ICSE*. ACM, 2016, pp. 50–56.

[104] X. Wu, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Testing Android Apps via Guided Gesture Event Generation," in *APSEC*. IEEE Computer Society, 2016, pp. 201–208.

[105] Z. Meng, Y. Jiang, and C. Xu, "Facilitating Reusable and Scalable Automated Testing and Analysis for Android Apps," in *Internetware*. ACM, 2015, pp. 166–175.

[106] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for Android," in *OOPSLA*. ACM, 2015, pp. 349–366.

[107] M. Ermuth and M. Pradel, "Monkey see, monkey do: effective generation of GUI tests with inferred macro events," in *ISSTA*. ACM, 2016, pp. 82–93.

[108] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in Android," in *ISSTA*. ACM, 2015, pp. 118–128.

[109] I. C. Morgado and A. C. R. Paiva, "Testing Approach for Mobile Applications through Reverse Engineering of UI Patterns," in *ASE Workshops*. IEEE Computer Society, 2015, pp. 42–49.

[110] C. Yeh, H. Lu, C. Chen, K. K. Khor, and S. Huang, "CRAXDroid: Automatic Android System Testing by Selective Symbolic Execution," in *SERE (Companion)*. IEEE, 2014, pp. 140–148.

[111] S. Salva and S. R. Zafimiharisoa, "APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 2, pp. 201–221, 2015.

[112] P. Costa, A. C. R. Paiva, and M. Nabuco, "Pattern Based GUI Testing for Mobile Applications," in *QUATIC*. IEEE Computer Society, 2014, pp. 66–74.

[113] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *WODA+PERTEA@ISSTA*. ACM, 2014, pp. 1–5.

[114] V. Nandakumar, V. Ekambaram, and V. Sharma, "Appstrument - A Unified App Instrumentation and Automated Playback Framework for Testing Mobile Applications," in *MobiQuitous*, vol. 131. Springer, 2013, pp. 474–486.

[115] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: an automatic system for revealing UI-based trigger conditions in android applications," in *SPSM@CCS*. ACM, 2012, pp. 93–104.

[116] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lu, "AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications," in *ICSME*. IEEE Computer Society, 2017, pp. 103–114.

[117] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *SIGSOFT FSE*. ACM, 2012, p. 59.

[118] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using Java PathFinder," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[119] W. Choi, G. C. Necula, and K. Sen, "Guided GUI testing of android apps with minimal restart and approximate learning," in *OOPSLA*. ACM, 2013, pp. 623–640.

[120] Q. Sun, L. Xu, L. Chen, and W. Zhang, "Replaying Harmful Data Races in Android Apps," in *ISSRE Workshops*. IEEE Computer Society, 2016, pp. 160–166.

[121] M. L. Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios," in *MSR*. IEEE Computer Society, 2015, pp. 111–122.

[122] P. Carter, C. Mulliner, M. Lindorfer, W. K. Robertson, and E. Kirda, "CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes," in *Financial Cryptography*, vol. 9603. Springer, 2016, pp. 231–249.

[123] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[124] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: segmented evolutionary testing of Android apps," in *SIGSOFT FSE*. ACM, 2014, pp. 599–609.

[125] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *ASE*. ACM, 2012, pp. 258–261.

[126] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *OOPSLA*. ACM, 2013, pp. 641–660.

[127] W. Yang, M. R. Prasad, and T. Xie, "A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications," in *FASE*, vol. 7793. Springer, 2013, pp. 250–265.

[128] V. Riccio, D. Amalfitano, and A. R. Fasolino, "Is this the lifecycle we really want?: an automated black-box testing approach for Android activities," in *ISSTA/ECOOP Workshops*. ACM, 2018, pp. 68–77.

[129] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *ICSE*. ACM, 2016, pp. 559–570.

[130] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring," in *MOBILESoft*. ACM, 2016, pp. 88–99.

[131] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in Android applications," in *AST@ICSE*. ACM, 2016, pp. 64–70.

[132] G. de Cleva Farto and A. T. Endo, "Evaluating the Model-Based Testing Approach in the Context of Mobile Applications," in *CLEI Selected Papers*, vol. 314. Elsevier, 2014, pp. 3–21.

[133] C. Guo, J. Xu, H. Yang, Y. Zeng, and S. Xing, "An automated testing approach for inter-application security in Android," in *AST*. ACM, 2014, pp. 8–14.

[134] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in android applications," in *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*, 2013, pp. 1–6.

[135] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in Android applications," in *ISSRE*. IEEE Computer Society, 2013, pp. 411–420.

[136] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of Android applications on the cloud," in *AST*. IEEE Computer Society, 2012, pp. 22–28.

[137] J. Yan, T. Wu, J. Yan, and J. Zhang, "Widget-Sensitive and Back-Stack-Aware GUI Exploration for Testing Android Apps," in *QRS*. IEEE, 2017, pp. 42–53.

[138] F. Paulovsky, E. Pavese, and D. Garbervetsky, "High-Coverage Testing of Navigation Models in Android Applications," in *AST@ICSE*. IEEE Computer Society, 2017, pp. 52–58.

[139] I. A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, and A. Usman, "AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing," *IEEE Access*, vol. 7, pp. 17 158–17 173, 2019.

[140] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated system input generation for Android applications," in *ISSRE*. IEEE Computer Society, 2015, pp. 461–471.

[141] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "AGRippin: a novel search based testing technique for Android applications," in *DeMobile@SIGSOFT FSE*. ACM, 2015, pp. 5–12.

[142] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu, "Capture-Replay Testing for Android Applications," in *2014 International Symposium on Computer, Consumer and Control*, 2014, pp. 1129–1132.

[143] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI Crawling-Based Technique for Android Mobile Application Testing," in *ICST Workshops*. IEEE Computer Society, 2011, pp. 252–261.

[144] C. Cao, C. Meng, H. Ge, P. Yu, and X. Ma, "Xdroid: Testing Android Apps with Dependency Injection," in *COMPSAC (1)*. IEEE Computer Society, 2017, pp. 214–223.

[145] A. S. Ami, M. M. Hasan, M. R. Rahman, and K. Sakib, "Mobicomonkey: context testing of Android apps," in *MOBILESoft@ICSE*. ACM, 2018, pp. 76–79.

[146] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," in *ASE*. ACM, 2018, pp. 419–429.

[147] Y. Cao, G. Wu, W. Chen, and J. Wei, "CrawlDroid: Effective Model-based GUI Testing of Android Apps," in *Internetware*. ACM, 2018, pp. 19:1–19:6.

[148] W. Zhao, Z. Ding, M. Xia, and Z. Qi, "Systematically Testing and Diagnosing Responsiveness for Android Apps," in *ICSME*. IEEE, 2019, pp. 449–453.

[149] W. Ravelo-Méndez, C. Escobar-Velásquez, and M. Linares-Vásquez, "Kraken-Mobile: Cross-Device Interaction-Based Testing of Android Apps," in *ICSME*. IEEE, 2019, pp. 410–413.

[150] C. Cao, J. Deng, P. Yu, Z. Duan, and X. Ma, "ParaAim: Testing Android Applications Parallel at Activity Granularity," in *COMPSAC (1)*. IEEE, 2019, pp. 81–90.

[151] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: detecting capability leaks of android applications," in *AsiaCCS*. ACM, 2014, pp. 531–536.

[152] Y. Kang, Y. Zhou, M. Gao, Y. Sun, and M. R. Lyu, "Experience Report: Detecting Poor-Responsive UI in Android Applications," in *ISSRE*. IEEE Computer Society, 2016, pp. 490–501.

[153] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "DiagDroid: Android performance diagnosis via anatomizing asynchronous executions," in *SIGSOFT FSE*. ACM, 2016, pp. 410–421.

[154] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective Real-Time Android Application Auditing," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 899–914.

[155] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: a lightweight UI-guided test input generator for Android," in *ICSE (Companion Volume)*. IEEE Computer Society, 2017, pp. 23–26.

[156] O. Riganelli, S. P. Mottadelli, C. Rota, D. Micucci, and L. Mariani, "Data loss detector: automatically revealing data loss bugs in Android apps," in *ISSTA*. ACM, 2020, pp. 141–152.

[157] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag," in *MoMM*. ACM, 2013, p. 68.

[158] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *MobiSys*. ACM, 2014, pp. 190–203.

[159] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps," in *MobiSys*. ACM, 2014, pp. 204–217.

[160] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for Android: are we really there yet in an industrial case?" in *SIGSOFT FSE*. ACM, 2016, pp. 987–992.

[161] K. Jamrozik and A. Zeller, "DroidMate: a robust and extensible test generator for Android," in *MOBILESoft*. ACM, 2016, pp. 293–294.

[162] H. Wu, Y. Wang, and A. Rountev, "Sentinel: generating GUI tests for Android sensor leaks," in *AST@ICSE*. ACM, 2018, pp. 27–33.

[163] G. Wu, Y. Cao, W. Chen, J. Wei, H. Zhong, and T. Huang, "AppCheck: A Crowdsourced Testing Service for Android Applications," in *ICWS*. IEEE, 2017, pp. 253–260.

[164] T. Ki, C. M. Park, K. Dantu, S. Y. Ko, and L. Ziarek, "Mimic: UI compatibility testing system for Android apps," in *ICSE*. IEEE / ACM, 2019, pp. 246–256.

[165] Y. L. Arnatovich, L. Wang, M. N. Ngo, and C. Soh, "Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customated input generation," *Softw. Pract. Exp.*, vol. 48, no. 5, pp. 1107–1142, 2018.

[166] M. L. Vásquez, "Enabling Testing of Android Apps," in *ICSE (2)*. IEEE Computer Society, 2015, pp. 763–765.

[167] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "FUSION: a tool for facilitating and augmenting android bug reporting," in *ICSE (Companion Volume)*. ACM, 2016, pp. 609–612.

[168] A. Rohella and S. Takada, "Testing Android Applications Using Multi-Objective Evolutionary Algorithms with a Stopping Criteria," in *SEKE*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018, pp. 308–307.

[169] R. Jabbarvand, J. Lin, and S. Malek, "Search-based energy testing of Android," in *ICSE*. IEEE / ACM, 2019, pp. 1119–1130.

[170] H. Tang, G. Wu, J. Wei, and H. Zhong, "Generating test cases to expose concurrency bugs in Android applications," in *ASE*. ACM, 2016, pp. 648–653.

[171] S. Negara, N. Esfahani, and R. P. L. Buse, "Practical Android test recording with espresso test recorder," in *ICSE (SEIP)*. IEEE / ACM, 2019, pp. 193–202.

[172] M. Fazzini, E. N. de A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests," in *ICST*. IEEE Computer Society, 2017, pp. 149–160.

[173] R. Coppola, L. Ardito, M. Torchiano, and E. Alégroth, "Translation from Visual to Layout-based Android Test Cases: a Proof of Concept," in *ICST Workshops*. IEEE, 2020, pp. 74–83.

[174] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *ASE*. IEEE Computer Society, 2017, pp. 308–318.

[175] F. Behrang and A. Orso, "Test Migration Between Mobile Apps with Similar Functionality," in *ASE*. IEEE, 2019, pp. 54–65.

[176] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, 2015.

[177] H. Shahriar, S. North, and E. Mawangi, "Testing of Memory Leak in Android Applications," in *HASE*. IEEE Computer Society, 2014, pp. 176–183.

[178] A. Avancini and M. Ceccato, "Security testing of the communication among Android applications," in *AST*. IEEE Computer Society, 2013, pp. 57–63.

[179] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol, "Testing Conformance of Life Cycle Dependent Properties of Mobile Applications," in *ICST*. IEEE Computer Society, 2012, pp. 241–250.

[180] H. Q. Thang, D. Nguyen, N. Ha, T. Pham, P. Nguyen, and V. Tran, "A Combinatorial Technique for Mobile Applications Software Testing," in *KSE*. IEEE, 2019, pp. 1–6.

[181] D. Adamo, D. Nurmuradov, S. Piparia, and R. C. Bryce, "Combinatorial-based event sequence testing of Android applications," *Inf. Softw. Technol.*, vol. 99, pp. 98–117, 2018.

[182] A. Ali, H. A. Maghawry, and N. L. Badr, "Automated parallel GUI testing as a service for mobile applications," *J. Softw. Evol. Process.*, vol. 30, no. 10, 2018.

[183] A. Rosenfeld, O. Kardashov, and O. Zang, "Automation of Android applications functional testing using machine learning activities classification," in *MOBILESoft@ICSE*. ACM, 2018, pp. 122–132.

[184] W. Xiong, S. Chen, Y. Zhang, M. Xia, and Z. Qi, "Reproducible Interference-Aware Mobile Testing," in *ICSME*. IEEE Computer Society, 2018, pp. 36–47.

[185] W. Song, X. Qian, and J. Huang, "EHBDroid: beyond GUI testing for Android applications," in *ASE*. IEEE Computer Society, 2017, pp. 27–37.

[186] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *SIGSOFT FSE*. ACM, 2014, pp. 588–598.

[187] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with AppDoctor," in *EuroSys*. ACM, 2014, pp. 18:1–18:15.

[188] P. McAfee, M. W. Mkaouer, and D. E. Krutz, "CATE: Concolic Android Testing Using Java PathFinder for Android Applications," in *MOBILESoft@ICSE*. IEEE, 2017, pp. 213–214.

[189] A. Allevato and S. H. Edwards, "RoboLIFT: engaging CS2 students with testable, automatically evaluated android applications," in *SIGCSE*. ACM, 2012, pp. 547–552.

[190] T. Griebe and V. Gruhn, "A model-based approach to test automation for context-aware mobile applications," in *SAC*. ACM, 2014, pp. 420–427.

[191] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verification Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[192] I. A. Moreno, J. P. Galeotti, and D. Garbervetsky, "Algorithm or Representation?: An empirical study on how SAPIENZ achieves coverage," in *AST@ICSE*. ACM, 2020, pp. 61–70.

[193] I. A. Moreno, J. P. Galeotti, and D. Garbervetsky, "An Empirical Study on How Sapienz Achieves Coverage and Crash Detection," *J. Softw. Evol. Process.*, vol. 35, no. 4, 2023.

[194] D. C. Karnopp, "Random search techniques for optimization problems," *Autom.*, vol. 1, no. 2-3, pp. 111–121, 1963.

[195] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?" in *GECCO*. ACM, 2015, pp. 1367–1374.

[196] J. Campos, Y. Ge, N. M. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Inf. Softw. Technol.*, vol. 104, pp. 207–235, 2018.

[197] O. Sahin and B. Akay, "Comparisons of metaheuristic algorithms and fitness functions on software test data generation," *Appl. Soft Comput.*, vol. 49, pp. 1202–1214, 2016.

[198] A. Ter-Sarkisov and S. R. Marsland, "Convergence Properties of $(\mu + \lambda)$ Evolutionary Algorithms," in *AAAI*. AAAI Press, 2011.

[199] G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[200] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation," in *SSBSE*, vol. 9275. Springer, 2015, pp. 93–108.

[201] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empir. Softw. Eng.*, vol. 20, no. 3, pp. 783–812, 2015.

[202] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.

[203] C. M. Fonseca and P. J. Fleming, "Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization," in *ICGA*. Morgan Kaufmann, 1993, pp. 416–423.

[204] "ELLA-customized: An improved version of the ELLA tool," https://github.com/ms1995/ella-customized, 2020.

[205] "Rhapsod/sapienz: A Prototype of Sapienz (Out-of-date and no longer supported)," https://github.com/Rhapsod/sapienz, 2016.

[206] "Implementación y mejoras a la herramienta SAPIENZ," https://github.com/FlyingPumba/evolutiz, (Accessed on 10/24/2023).

[207] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri, "An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation," in *SSBSE*, vol. 10452. Springer, 2017, pp. 33–48.

[208] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating Branch Coverage as a Many-Objective Optimization Problem," in *ICST*. IEEE Computer Society, 2015, pp. 1–10.

[209] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Trans. Software Eng.*, vol. 44, no. 2, pp. 122–158, 2018.

[210] "EMMA: a free Java code coverage tool," http://emma.sourceforge.net/, 2020.

[211] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," in *SSBSE*, vol. 6956. Springer, 2011, pp. 33–47.

[212] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empir. Softw. Eng.*, vol. 18, no. 3, pp. 594–623, 2013.

[213] "Azure: Cloud Computing Service by Microsoft," https://azure.microsoft.com, (Accessed on 07/21/2023).

[214] "ELLA: A tool for binary instrumentation of Android apps," https://github.com/saswatanand/ella, 2020.

[215] A. Panichella and U. R. Molina, "Java Unit Testing Tool Competition - Fifth Round," in *SBST@ICSE*. IEEE, 2017, pp. 32–38.

[216] S. García, D. Molina, M. Lozano, and F. Herrera, "A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 Special Session on Real Parameter Optimization," *J. Heuristics*, vol. 15, no. 6, pp. 617–644, 2009.

[217] T. Vogel, C. Tran, and L. Grunske, "Does Diversity Improve the Test Suite Generation for Mobile Applications?" in *SSBSE*, vol. 11664. Springer, 2019, pp. 58–74.

[218] T. Vogel, C. Tran, and L. Grunske, "A comprehensive empirical evaluation of generating test suites for mobile applications with diversity," *Inf. Softw. Technol.*, vol. 130, p. 106436, 2021.

[219] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained Code Coverage Measurement in Automated Black-box Android Testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 23:1–23:35, 2020.

[220] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: collecting millions of Android apps for the research community," in *MSR*. ACM, 2016, pp. 468–471.

[221] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, "Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis," in *ICSME*. IEEE, 2020, pp. 557–568.

[222] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Inf. Softw. Technol.*, vol. 104, pp. 236–256, 2018.

[223] R. Coppola, M. Morisio, and M. Torchiano, "Mobile GUI Testing Fragility: A Study on Open-Source Android Applications," *IEEE Trans. Reliab.*, vol. 68, no. 1, pp. 67–90, 2019.

[224] "Full description of the JSON schema," https://github.com/FlyingPumba/etg/blob/master/schema.json, 2022.

[225] "Espresso setup instructions | Android Developers," https://developer.android.com/training/testing/espresso/setup#set-up-environment, (Accessed on 07/17/2023).

[226] "accessibility - Android TalkBack: Hint overwrites contentDescription - Stack Overflow," https://stackoverflow.com/questions/29889904/android-talkback-hint-overwrites-contentdescription, (Accessed on 07/25/2023).

[227] "Paquete de replicación para el artículo On the feasibility and challenges of synthesizing executable Espresso tests," https://github.com/FlyingPumba/etg-paper-replication-package, 2022.

[228] "Jackson JSON library," https://github.com/FasterXML/jackson, 2022.

[229] "JaCoCo coverage tool," https://www.eclemma.org/jacoco, 2022.

[230] "ImageMagick's compare program," https://imagemagick.org/Usage/compare/, 2022.

[231] "ImageMagick's Fuzz Factor," https://legacy.imagemagick.org/Usage/color_basics/#fuzz, 2022.

[232] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability Transformation," *IEEE Trans. Software Eng.*, vol. 30, no. 1, pp. 3–16, 2004.

[233] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *ESEC/SIGSOFT FSE*. ACM, 2015, pp. 107–118.

[234] "Implementación de código abierto del prototipo para el artículo On the feasibility and challenges of synthesizing executable Espresso tests," https://github.com/FlyingPumba/etg, 2022.

[235] I. A. Moreno, L. D. Meo, M. Auer, J. P. Galeotti, and G. Fraser, "Brewing Up Reliability: Espresso Test Generation for Android Apps," in *ICST*. IEEE Computer Society, 2024.

[236] "Implementación de código abierto de EspressoMaker (extensión de MATE)," https://github.com/FlyingPumba/EspressoMaker-mate, (Accessed on 10/24/2023).

[237] "Implementación de código abierto de EspressoMaker (extensión de MATE server)," https://github.com/FlyingPumba/EspressoMaker-mate-server, (Accessed on 10/24/2023).

[238] "Paquete de replicación para el artículo Brewing Up Reliability: Espresso Test Generation for Android Apps," https://zenodo.org/records/10038308, (Accessed on 06/07/2023).

[239] "Using Java Reflection," https://www.oracle.com/technical-resources/articles/java/javareflection.html, (Accessed on 07/22/2023).

[240] "Do Android's R.id need to be unique? - Stack Overflow," https://stackoverflow.com/questions/33326887/do-androids-r-id-need-to-be-unique/33327032#33327032, (Accessed on 07/22/2023).

[241] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[242] T. Xie, "Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking," in *ECOOP*, vol. 4067. Springer, 2006, pp. 380–403.

[243] "Android Interface Definition Language (AIDL) | Android Developers," https://developer.android.com/guide/components/aidl, (Accessed on 06/07/2023).

[244] "<instrumentation>| Android Developers," https://developer.android.com/guide/topics/manifest/instrumentation-element, (Accessed on 07/24/2023).

[245] "AndroidManifest.template | guptadeepanshu/agp-sources | GitHub," https://github.com/guptadeepanshu/agp-sources/blob/main/sources/com.android.tools.build/builder/com/android/builder/internal/AndroidManifest.template, (Accessed on 07/24/2023).

[246] "TestManifestGenerator.java | guptadeepanshu/agp-sources | GitHub," https://github.com/guptadeepanshu/agp-sources/blob/main/sources/com.android.tools.build/builder/com/android/builder/internal/TestManifestGenerator.java, (Accessed on 07/24/2023).

[247] M. Auer, I. A. Moreno, and G. Fraser, "WallMauer: Robust Code Coverage Instrumentation for Android Apps," in *AST@ICSE*. ACM/IEEE, 2024.

[248] C. Escobar-Velásquez, D. Riveros, and M. Linares-Vásquez, "MutAPK 2.0: a tool for reducing mutation testing effort of Android apps," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 1611–1615.

[249] "Yuan-W/muDroid: Mutation Testing tool for Android Integration Testing," https://github.com/Yuan-W/muDroid, (Accessed on 06/07/2023).

[250] J. Liu, X. Xiao, L. Xu, L. Dou, and A. Podgurski, "DroidMutator: an effective mutation analysis tool for Android applications," in *ICSE (Companion Volume)*. ACM, 2020, pp. 77–80.

[251] A. Arcuri and L. C. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verification Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.

[252] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[253] M. L. Vásquez, G. Bavota, M. Tufano, K. Moran, M. D. Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for Android apps," in *ESEC/SIGSOFT FSE*. ACM, 2017, pp. 233–244.

[254] A. Arcuri, "Many Independent Objective (MIO) Algorithm for Test Suite Generation," in *Search Based Software Engineering*. Springer International Publishing, 2017, p. 3–17.

[255] A. Arcuri, "RESTful API Automated Test Case Generation," in *QRS*. IEEE, 2017, pp. 9–20.

[256] J. C. Alonso, S. Segura, and A. Ruiz-Cortés, "AGORA: Automated Generation of Test Oracles for REST APIs," in *ISSTA*. ACM, 2023, pp. 1018–1030.

[257] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. Lou, and W. Chen, "CodeT: Code Generation with Generated Tests," in *ICLR*. OpenReview.net, 2023.