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

View Ranking: Un algoritmo model-based para generación automática de tests de aplicaciones móviles

Tesis de Licenciatura en Ciencias de la Computación

Lisandro Diaz Di Meo

Director: Iván Arcuschin Moreno

Buenos Aires, 2024

VIEW RANKING: UN ALGORITMO MODEL-BASED PARA GENERACIÓN AUTOMÁTICA DE TESTS DE APLICACIONES MÓVILES

Las aplicaciones móviles ya están integradas al ecosistema de nuestras actividades del día a día. Por ende, es crucial que estas estén testeadas para proveer un funcionamiento esperado. No obstante, es poco común que los equipos de desarrollo inviertan tiempo en la escritura de tests, y la garantía de funcionalidad queda en manos de pruebas manuales hechas por los *QA's*. Las herramientas de generación automática de tests tratan de atacar esta problemática, pero suelen tener el problema de que no son muy conocidas por la comunidad, o bien son difíciles de integrar en el flujo de desarrollo. Otro enfoque común es el desarrollo de herramientas que faciliten la escritura de casos de tests, para que tanto desarrolladores como QA's pueda automatizar los flujos de una aplicación con poco esfuerzo. *Maestro* es una de las herramientas que surgió en los últimos años para este último, que permite (entre otras funcionalidades) escribir casos de test para aplicaciones móviles tanto de iOS como Android. Se plantea en este trabajo extender Maestro para que pueda generar casos de test automáticamente, y así ofrecer a los desarrolladores un punto de partida para sus pruebas. A su vez, se introduce *View Ranking*, un algoritmo *model based* cuyo objetivo es explorar todas las acciones disponibles de una aplicación aprovechando el modelo que se puede construir en base a las relaciones entre las mismas. Luego, presentamos una comparación exhaustiva de View Ranking contra *Random*.

Palabras claves: Model-based, Test-generation, Android, iOS, View Ranking, Search-based.

AGRADECIMIENTOS

A mis padres y hermano, Silvana, Ariel, y Santiago, que siempre me apoyaron en todo con mucho amor, y sin ellos no estaría acá.

A mis amigos, que siempre estuvieron ahí para mí, y me hicieron reir cuando más lo necesitaba.

A Sofía, por darme motivación siempre, y acompañarme en todo momento.

A mis compañeros de cursada y profesores, que me ayudaron a entender esta hermosa carrera.

A mi director Iván, por toda la paciencia y por ayudarme a ser mejor profesional.

Al LaFHIS, por darme un lugar de aprendizaje y recibirme con mucha generosidad.

Y a la Universidad Pública, que hoy me siento muy orgulloso de representar.

A los dos que más disfruté ver jugar, Wildo y Lionel Messi.

Índice general

1. INTRODUCCIÓN

Las aplicaciones para smartphones ya son parte de nuestra vida hace varios años. El uso intensivo de estas generó, entre otras consecuencias, una exigencia más alta en la calidad que se espera de las mismas. Por eso, cuando una aplicación empieza a fallar, las quejas repercuten directamente en la calificación de la misma, perjudicando a la marca y al equipo de desarrollo.

Por otro lado, y en contramano con las exigencias de calidad, es común que en los equipos de desarrollo exista un rechazo a la escritura de casos de tests, porque no solo deben lidiar con escribir tests unitarios para cubrir las funcionalidades del código, sino que también deben probar que la GUI (*Graphical User Interface*) se comporte como se espera escribiendo *UI Tests*. Esta disyuntiva pone en jaque a los desarrolladores, ya que deben priorizar entre entregar *features* al público, o garantizar la integridad de la aplicación. Generalmente lo que termina sucediendo es que se derivan los tests de la GUI al equipo de *QA's* (Quality Assurance).

Aunque esta práctica logra quitar parte del trabajo al desarrollador, los QA's suelen únicamente realizar *Manual GUI Testing*, que consiste en probar casos descritos por las reglas de negocio a mano (por ejemplo, manualmente abrir la aplicación y verificar que ciertas reglas se cumplen). Es decir, *UI Tests* manuales. Esto es alejado de lo ideal porque termina siendo bastante lento, propenso a errores, y además no es algo que podamos integrar a un *pipeline* de desarrollo.

Si bien algunos proyectos terminan escribiendo tests de GUI, no suele ser algo muy exhaustivo. Esto se debe a que el proceso es lento, costoso, y requiere de cierto conocimiento técnico. Suele pasar que para abaratar costos, los QA's no sepan usar herramientas de tests, y se limiten a conocer las reglas de negocio para probar la aplicación manualmente.

Los tests manuales además de ser poco confiables, suelen dejar de lado situaciones excepcionales. Esto puede deberse a limitaciones de tiempo, recursos, por error del propio tester, o porque la cantidad de casos posibles a probar no es manejable. La causa de esta última es la inherente complejidad de una aplicación, que puede generar un gran número de estados a examinar, lo que añade un desafío difícil (o imposible) de atacar de manera manual.

Para tratar este problema es ampliamente estudiada la **generación automática de tests de GUI**. Esta práctica consiste en aplicar heurísticas de exploración sobre la AUT (*application under test*). Sapienz [14] y MATE [6], por ejemplo, ofrecen varios algoritmos genéticos para lograr dicho objetivo. A diferencia de estos, nosotros haremos un acercamiento *model-based*.

Las estrategias *model-based* buscan describir parcialmente la AUT y derivar *test suites* del mismo. Uno de los desafíos fundamentales es que nuestra abstracción del modelo logre capturar apropiadamente el espacio de comportamiento de cada estado. Profundizaremos esto en la sección 3.1.

En este trabajo presentamos View Ranking, un algoritmo *model-based* diseñado para maximizar la ejecución de acciones de una aplicación móvil. Aunque fue inicialmente mencionado durante una charla en *ISSTA 2020* [1], no estaba acompañado de una formalización concreta, ni fue publicado posteriormente. Por lo tanto, una de las contribuciones principales de esta tesis ha sido su formalización e implementación. Para lograr esto tomamos decisiones de diseño que exploraremos en detalle en la sección de Implementación 3. Esta fue realizada extendiendo *Maestro* [16], una herramienta para la escritura de test cases ejecutables tanto en iOS como Android, con una amplia aceptación de la comunidad.

Luego, en la fase de Evaluación 4, comparamos View Ranking contra *Random* (ejecución aleatoria de acciones) mediante un análisis cuantitativo en el cual buscamos medir la mejora por sobre este último. Sumado a esto, realizamos un análisis cualitativo de las diferencias existentes de los modelos producidos para Android y iOS.

Finalmente, presentamos conclusiones y planteamos algunas preguntas que podrían ser abordadas en trabajos futuros.

En resumen, el objetivo de esta tesis es presentar un algoritmo model based para generar **UI Tests Suites** para iOS y Android en una herramienta *production-ready*, como así realizar una

evaluación exhaustiva de sus capacidades.

2. PRELIMINARES

2.1. Dispositivos Móviles

Los dispositivos móviles son terminales de cómputo portátiles. El término está asociado comúnmente a los *smartphones*, aunque también se refiere a *tablets* o *smart watches*. Estos dispositivos están equipados con una pantalla táctil y típicamente incluyen algunos botones para funciones básicas como control de volumen, bloqueo de pantalla y acceso a la cámara. Además, tienen la capacidad de conectarse a Internet a través de WiFi o redes móviles, entre otras características. El crecimiento del poder de cómputo de estos dispositivos hizo que una gran parte de personas abandone las computadoras convencionales [8], y utilicen al propio smartphone para las actividades que estas realizaban (escribir correos electrónicos, armar hojas de cálculo, navegar por internet, etc).

2.2. Android

Android es un sistema operativo móvil basado en Linux. Originalmente lanzado en 2008 y pensado para teléfonos táctiles. Hoy en día es habitual encontrarlo en una gran variedad de otros dispositivos, como por ejemplo: smart watches, televisores, terminales de cobro centralizadas. Al día de la fecha se estima que más del 70 % de las personas con smartphones utiliza Android [21].

El código fuente de Android es de libre acceso bajo la iniciativa de *Android Open Source Project* [11]. Esto permite una flexibilidad muy grande para su modificación, y un acceso más directo a su implementación. Esta es una de las razones por las que en el contexto de generación automática de tests es donde más artículos pueden hallarse.

2.3. iOS

iOS es el sistema operativo móvil desarrollado por *Apple Inc.* basado en *Darwin*. También fue pensado para teléfonos táctiles, y hoy en día también está presente en otros dispositivos, pero por su naturaleza de ser *código cerrado*, solo puede ser utilizado por dispositivos de Apple. Esta falta de apertura limita la capacidad de aplicar ciertas técnicas de análisis de software que sí son viables en Android, como el análisis a nivel de byte-code, instrumentación, instalación de aplicaciones externas, modificación del sistema operativo, entre otras. Esto se traduce en que la cantidad de trabajos y herramientas relacionadas con la generación automática de tests para Android es significativamente mayor que para iOS.

2.4. Aplicaciones Móviles

Las aplicaciones móviles son los programas que podemos ejecutar en estos dispositivos (análogos a los programas de las computadoras). Los móviles en sí ya traen consigo un conjunto de aplicaciones que apuntan a las funcionalidades básicas de un teléfono (calendario, agenda telefónica, alarma, etc). A su vez, ambos sistemas operativos cuentan con una *tienda de aplicaciones*, en la que el usuario puede descargar o comprar aplicaciones

hechas por terceros. Estas últimas son las que suelen generar un impacto más grande en el uso de los smartphones.

2.5. Testing

El testing de software es la práctica de analizar un programa mediante ciertas pruebas en pos de mejorar la calidad del mismo, ya sea para corroborar que su comportamiento es el esperado o para identificar errores.

Es un proceso sumamente importante, ya que apunta a crear confianza sobre el software en cuestion. Sin embargo, se debe tener en cuenta que "el testing de software solo puede mostrar la presencia de errores, nunca su ausencia" [5]. Es decir, una prueba exitosa no demuestra la *correctitud* del programa, sino que simplemente que para esas pruebas **no falló**. Esto se debe a que para asegurar un comportamiento esperado, deberíamos probar todos los inputs o estados posibles para dicho programa. En sí el proceso consiste en escribir *test cases* donde probamos características de nuestro programa.

Una de las dificultades del testing es que usualmente los casos a probar son demasiados –*e.g.* de orden exponencial–, lo que hace inviable probarlos a todos. Ante la inherente complejidad de enfrentarse a un número potencialmente exponencial de casos a probar, resulta crucial adoptar una estrategia efectiva al escribir test cases. Esto no solo permite optimizar la cobertura de casos, sino también reducir los costos en términos de tiempo y recursos. Una técnica útil consiste en analizar el programa para identificar conjuntos de inputs equivalentes. Por ejemplo, para probar del método `add` del Listing 2.1 el hecho de que la suma de dos números positivos mantiene la positividad, no hace falta probar todos los pares de `numberA` y `numberB` positivos. Esencialmente se trata de dividir el conjunto de inputs posibles en subconjuntos que reflejen comportamientos similares. Para lograr esto, es fundamental tener un profundo entendimiento de las reglas de negocio subyacentes. Otra estrategia válida es la generación automática de test cases, en la cual nos enfocamos en este trabajo (Ver 2.6). En última instancia, la combinación de estas metodologías puede garantizar una cobertura exhaustiva de los posibles escenarios del programa a testear, dentro de los límites de recursos disponibles.

En los test cases se suelen escribir *assertions* (aserciones) para verificar condiciones que presenta el programa bajo test. De esta manera, ante un comportamiento no esperado en el test, la ejecución del caso se interrumpirá si una aserción no se cumple, notificando que hubo un error en la prueba.

Al producto final de escribir varios test cases se lo conoce como *test suite*. Generalmente esta es la que se termina ejecutando para correr los test cases en conjunto. Si todos los tests finalizan existosamente, entonces se considera que la suite es exitosa.

A su vez, en el mundo del testing podemos encontrar diferentes tipos de tests. Cada uno de ellos enfocado en aspectos específicos del sistema bajo test, con el objetivo de proporcionar diferentes garantías de calidad.

2.5.1. Unit Tests

El *Unit Testing* (prueba unitaria) es el tipo de test más elemental. El propio nombre nos dice que apunta a testear "unidades", es decir, componentes dentro de nuestro sistema que puedan ser aislados. Por lo general, son métodos, funciones, o una clase. El propio desarrollador suele escribirlos para corroborar mínimo funcionamiento del código. En el

Listing 2.1 podemos observar cómo se realiza un test case del método `add`, agregando una aserción para corroborar que se cumple con el resultado esperado.

```
def add(numberA: int, numberB: int) -> int:
    return numberA + numberB

class AddTest :
    def test_sum_is_properly_done():
        expected = 42 # Resultado esperado
        actual = add(40,2) # Resultado obtenido
        assertEquals(
            expected,
            actual
        ) # Metodo que chequea la condicion esperada
```

Listing 2.1: Caso de test unitario en Python

2.5.2. Integration Tests

El *integration testing* busca analizar el comportamiento de un módulo entero, o cómo un conjunto de módulos interactúan entre sí. Un módulo suele englobar varios componentes de nuestro sistema. Cuenta con diferentes dinámicas para ser aplicado (top-down, bottom-up, sandwich, big-bang, etc.), pero la filosofía es integrar los diferentes componentes de un módulo –o módulos– para detectar errores en la interacción entre estos, o verificar que estos se comportan como es esperado.

Por ejemplo, podríamos tener dos módulos: uno encargado de la gestión de usuarios, y otro encargado de la gestión de productos. En un integration test buscaríamos que estos dos módulos interactúen correctamente entre sí.

2.5.3. End-to-End Tests

Los tests end-to-end analizan flujos enteros del sistema, simulando un escenario del mundo real realizado por un usuario. En el contexto del desarrollo de aplicaciones móviles, estos son representados por los *UI Tests*.

En un UI Test se detallan las acciones a realizar sobre la pantalla, como también aserciones para verificar el estado de la GUI, o de algún componente de esta. Se suelen organizar en flujos que ofrece la aplicación. Tomando como ejemplo una aplicación de compras online, un UI Test podría describir los escenarios que ocurren desde la selección del producto, hasta una vez efectuado el cobro.

Es poco común la escritura de estos casos de tests. Como hemos mencionado en la introducción (Ver 1), se realiza una práctica similar mediante el *Manual Testing*. Básicamente una persona realiza los flujos de manera manual y verifica que el comportamiento es el esperado.

2.6. Generación automática de tests

La generación automática de casos de test busca automatizar el proceso mencionado en 2.5 a través de diferentes estrategias.

Los intentos de automatizar el testing encuentran sus orígenes a inicios de los 60's, particularmente con el paper de *Richard L. Sauder* [20], donde muestra cómo producir

datos de prueba para un programa en COBOL. Algunos años después Hanford [9] implementaría un mecanismo basado en una gramática libre de contexto para generar test cases de manera aleatoria (una estrategia que al día de hoy se suele seguir utlizando).

Con el paso del tiempo, el estado del arte progresó, y las heurísticas para generar tests cases fueron evolucionando y adaptándose a diferentes escenarios. Hoy en día es común que las empresas que necesitan entregar software **confiable** (con fuerte impacto social y civil) implementen en su pipeline de desarrollo alguna herramienta de generación de casos de test, como *Meta* –antiguamente *Facebook*– con Sapienz [14][7].

2.6.1. Tipos de generación automática de tests

La generación de casos de tests pueden tener dos naturalezas: *estática* o *dinámica*.

- **Estática:** se analiza el código del programa a testear para comprender su estructura y comportamiento. Luego, con algún tipo de criterio, se genera una test suite. Notar que no hace falta ejecutar el programa en ningún momento para generarla.
- **Dinámica:** los casos de test son generados mientras el programa está siendo ejecutado. En este caso se suele analizar el estado actual del programa, entradas, salidas, entre otros datos relevantes.

2.6.2. Proceso de Generación

El proceso de generación suele contar con las siguientes variables para la exploración:

- Presupuesto de búsqueda: Es la unidad de tiempo asociada al proceso de generación de casos de test. Normalmente no vamos a lograr que los casos de test generados cubran todas las condiciones que exijamos para ellos, por ende es natural pensar en prefijar un límite de tiempo para que el algoritmo explore al programa y no quede ejecutando indefinidamente.
- Condición de corte: Análogamente podemos definir un corte para la exploración en base a algún criterio (cubrimiento de líneas, cantidad de casos generados, etc). Algo para considerar en el caso de usar solo condición de corte (es decir, sin presupuesto de búsqueda) es que esta siempre debe ser alcanzada. Caso contrario, el algoritmo puede quedar explorando el programa indefinidamente.
- Función objetivo: Una manera de guiar la generación de casos de test es mediante una *función objetivo* que nos dé cierto feedback de la exploración. Por ejemplo, si nuestro objetivo es utilizar la mayor cantidad posible de acciones que ofrece una aplicación, la función objetivo asociada es la cantidad de acciones ejecutadas sobre las descubiertas.

2.6.3. Problemáticas de la generación de casos de test

Anteriormente se ha mencionado que un test solo puede evidenciar errores, o predicar correctamente sobre un subconjunto de casos (Ver 2.5). Pero ambas situaciones introducen el *problema del oráculo* [10]. Un oráculo es un mecanismo que nos define cuándo un test ha fallado o no. Cuando se escribe `assertEquals(expected, actual)` estamos definiendo un oráculo porque estamos pidiendo que el resultado de una ejecución concreta (`actual`)

sea igual a un valor provisto (`expected`). Si bien existen casos que evidencian un defecto implícito en el programa que nos permiten determinar un caso de test fallido –por ejemplo, un cierre inesperado del mismo–, en el caso general no existe manera trivial ni automática de definir cuál es el comportamiento esperado o erróneo de un programa. Por lo tanto, es responsabilidad del programador proporcionar oráculos que se basen en el conocimiento de los valores esperados de la ejecución.

Por otro lado, uno de los conflictos que tienen las aplicaciones móviles es que suelen estar atadas a comportamientos no determinísticos. Es decir, generar dos test suites para una aplicación bajo las mismas condiciones puede producir dos suites distintas. Peor aún, pueden generar un test case durante la fase de exploración que al re-ejecutarlo falle. Esto se debe a que las aplicaciones pueden introducir comportamientos como los detallados en el Listing 2.2. Básicamente ese código nos dice que la aplicación puede mostrar un elemento visual de manera aleatoria. Si se da el caso que nuestro generador de tests interactúa con dicho elemento, entonces el test case producido fallará el 50 % de las veces. A este fenómeno se lo conoce como *flakiness*, y es inevitable ya que depende de la propia naturaleza de las aplicaciones.

```
1
2  void showDiscountsPopUp(){
3      /**
4       * El contenido de este if se ejecuta de manera aleatoria un 50% de las
        veces
5       * y no lo podemos controlar
6       **/
7      if ( shouldShowDiscountPopUp() ) {
8          displayDiscountsPopUp();
9      }
10 }
11
12 boolean shouldShowDiscountPopUp() {
13     return Random.nextBoolean();
14 }
```

Listing 2.2: Una aplicación puede mostrar contenido visual de manera aleatoria

2.7. Model Based Testing

Model Based Testing es una técnica de *automated testing* en la que se utiliza un modelo del *System Under Test* (SUT) para derivar test cases del mismo. El modelo suele ser un grafo dirigido –o un *autómata de estados finitos*–, donde los nodos representan estados abstractos del sistema, y las aristas las transiciones que pueden ocurrir entre estos estados. De esta manera el modelo *abstrae* el comportamiento de nuestro sistema. Con esto, en lugar de crear test cases manualmente, podemos hacer un recorrido del grafo que capture algún escenario que querramos probar de nuestro sistema. Finalmente a este recorrido (que no es más que un listado de transiciones entre nodos) lo podemos pensar como un *test case abstracto*, que luego lo podemos *mappear* en un test case concreto (es decir, ejecutable contra nuestro sistema real). En la Figura 2.1 se muestra un ejemplo de cómo realizar este mapping.

Las ventajas que ofrece esta técnica son:

- **Test cases correctos respecto al modelo:** Los test cases generados son correctos

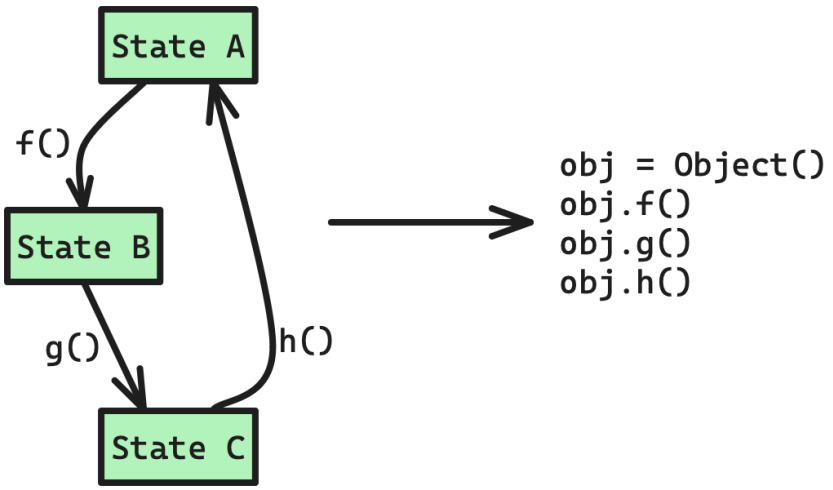

Fig. 2.1: Un sistema con tres estados y tres transiciones. Haciendo el recorrido de $\{(A \to B), (B \to C), (C \to A)\}$ podemos mapearlo al test case concreto de la derecha.

con respecto al modelo, dado que al representar recorridos en el grafo, los hace válidos por su construcción.

- **Generación automática de tests:** El grafo al modelar el comportamiento de nuestro sistema nos permite desarrollar mecanismos para recorrerlo, y así automatizar la generación de test cases.

No obstante, construir estos modelos manualmente no es factible en el caso general, puesto que los sistemas comúnmente cuentan con una gran cantidad de estados, o directamente no son determinísticos.

Para solucionar eso, se suele buscar una manera de generar el modelo de manera automática. Es decir, explorar el sistema y construir el grafo dinámicamente. Esta construcción dinámica nos permite sacar conclusiones de los estados parciales, para así extender el grafo inteligentemente. En nuestro caso, nuestro SUT es una aplicación móvil, y el modelo va a estar basado en las acciones sobre la GUI.

2.8. Graphical User Interface

La *Graphical User Interface* (GUI) es una interfaz que permite a los usuarios interactuar con dispositivos electrónicos utilizando elementos gráficos como botones, tarjetas, imágenes, menús, etc.

Internamente las GUI's son representadas como árboles, en los que cada nodo puede tener cero o más hijos. Cada nodo va acompañado de datos (texto, contenido descriptivo, etc) y metadatos (tamaño, clase, id, tipo, etc). Esto es así para que el sistema de renderización subyacente pueda dibujar correctamente los elementos en pantalla. A estos elementos también se los conoce como *widgets* o *views* (vistas).

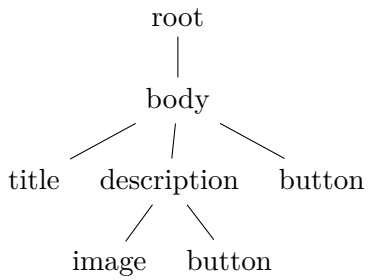

Fig. 2.2: Representación interna de la GUI de 2.3 en forma de árbol. También conocido como jerarquía de vistas.

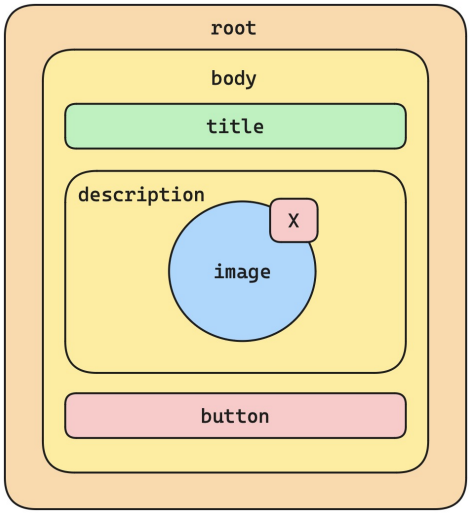

Fig. 2.3: Representación visual de la GUI 2.2.

2.8.1. Acciones sobre la GUI

Cuando hablamos de acciones sobre la GUI nos referimos a las operaciones o interacciones que nos ofrece cada elemento gráfico proporcionado por la interfaz, como clickear un botón, arrastrar una imagen, escribir en un campo de texto, entre otras.

En los UI tests se describen acciones sobre los elementos de la GUI. Para llevar a cabo esto, las herramientas específicas de este propósito (*e.g.* Espresso [12]) requieren que les propocionemos la acción a ejecutar, y una especificación unívoca del elemento con el que desamos interactuar. Esta especificación es esencial porque es la que les permite localizar al elemento dentro de la GUI. Por ejemplo, si se desea interactuar con el botón de la Figura 2.3 con el texto "button", es necesario proporcionar los datos que lo diferencien de los demás elementos.

Introducimos aquí el concepto de *Selector*, una entidad necesaria para elegir un elemento de la GUI y posteriormente llevar a cabo una acción sobre él. En esencia describen las propiedades de un elemento de la GUI. Por ejemplo `Selector(withId: 'id', withText: 'text', ...)` describe que hay un elemento con identificador 'id' y texto 'text'. Con estos selectores es posible decirle al sistema que busque al elemento que cumpla dichas propiedades, y ejecute una acción.

2.8.2. Problemas al armar selectores

El uso de selectores introduce el problema de *desambiguar vistas* [18], porque no siempre un elemento tiene atributos únicos en una GUI. De hecho, en algunos casos resulta imposible distinguir una vista particular.

Si bien es posible definir un criterio de orden para solucionar una situación como la que presenta la Figura 2.4 (es decir, utilizar alguna noción de orden en el árbol), no termina siendo evidente para quien lee un test case el hecho de interactúar con el n-ésimo elemento que tenga el texto `text`. El motivo de esto es que, si bien el árbol puede presentar cierto orden, visualmente puede ser otro, porque el sistema de reenderización se termina encargando de ubicar los elementos en la GUI.

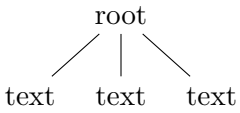

Fig. 2.4: Es imposible definir un selector para cualquiera de los hijos de la raíz, a menos que se use una noción de orden entre los mismos.

Más adelante (Ver 3.3) profundizaremos en nuestro enfoque para este problema.

2.9. Maestro

Maestro es una herramienta para escribir casos de test en formato *YAML*. Este formato de test se abstrae del sistema operativo donde la app está ejecutando, permitiendo que un mismo caso de test funcione tanto en Android como iOS. Maestro llama a estos archivos *Flow files*. En el Listing 2.3 se muestra un ejemplo de un test case de Maestro.

Esta herramienta hoy en día es bastante aceptada por varias empresas y desarrolladores (al día de la fecha registra +4500 *stars* en su repositorio de GitHub [17]), cuenta con una comunidad muy activa, y que sigue buenas prácticas de desarrollo (*i.e.* tests regresivos). Por estas razones se eligió Maestro para la implementación de *View Ranking*.

Notar que Maestro no usa el código fuente de las aplicaciones a testear, ya que solo utiliza información que provee la GUI.

```
appId: com.apple.MobileAddressBook
---
- launchApp
- tapOn:
    withText: "John Appleseed"
- tapOn:
    withText: "Edit"
- tapOn:
    withText: "Add phone"
- inputText: "123123"
- tapOn:
    withText: "Done"
```

Listing 2.3: Flow file sobre la aplicación de contactos de iOS

A grandes rasgos lo que se describe en 2.3 es:

- `appId: com.apple.MobileAddessBook`: En la cabecera de los flowfiles es necesario indicar el *PackageName* de la aplicación a testear. El PackageName es el identificador de la aplicación.
- `launchApp`: Le pide al sistema operativo que ejecute la aplicación con el PackageName proporcionado.
- `tapOn (selector)`: Envía un evento de click/tap al elemento que cumpla con dicho selector.
  - `withText`: Selector que busca elementos con el texto proporcionado.
- `inputText (text)`: Envía un evento de ingreso de texto con el contenido proporcionado.

Maestro ofrece varias acciones para interactuar con los elementos de la GUI o con el propio dispositivo. A estas acciones las llama *Commands* (comandos). Estas pueden requerir (o no) de un selector para así encontrar al elemento, e interactuar con él.

2.9.1. Selectors

Los Selectors (implementación de selectores de Maestro), como se ha mencionado antes (Ver 2.8.1), son objetos que predican sobre las propiedades de un elemento en la GUI. Dicho elemento puede existir como no. Es posible pensar a los selectores como un objeto que describe un filtro sobre la GUI.

Maestro permite‘’ construir a sus selectores con varios atributos. Algunos de estos son:

- textRegex: describe el contenido textual de un elemento, como así el contenido textual *descriptivo* (visible para herramientas de accesibilidad).
- idRegex: describe un identificador del elemento. Es normal (especialemente en Android) anotar a los elementos con algún identificador para poder referenciarlos en el código y operar sobre ellos.
- containsChild: describe a una vista que tenga a un determinado hijo directo.
- containsDescendants: describe a una vista que tenga como descendientes a un conjunto de elementos.

Por simplicidad a la hora de desambiguar vistas, se ha decidido de utilizar solo los atributos de *textRegex*, *idRegex*, y *containsChild*.

2.9.2. Abstracción sobre las vistas de Android y iOS

Cada sistema operativo tiene su propia implementación de la representación interna de los elementos de la GUI. Es decir, los datos y metadatos existentes en un nodo no son los mismos en Android que en iOS. Esto es un problema, ya que si queremos escribir selectores que sirvan para ambos se debería mantener cierta homogeneidad. Maestro unifica ambas representaciones mediante una clase llamada `TreeNode`. Esta clase cuenta con atributos que capturan características de un elemento visual, como si es clickeable, si está activado, como así un listado de sus nodos hijos. Notar que esto define una entidad recursiva que representa a un árbol, similar al de la Figura 2.2.

2.10. Arquitectura de Maestro

La arquitectura de Maestro se basa principalmente en los componentes de la Figura 2.5. Una manera intuitiva de entender la arquitectura de Maestro es justamente pensar en el concepto de una *Orquesta*. En una Orquesta hay instrumentos (*Driver*'s), y es necesario de músicos (*Maestro*) para que pueda tocarlos.

- **Orchestra:** La clase *Orchestra* es la más *high level*, ya que es –casi– el punto de entrada para interactuar con las funcionalidades de Maestro. No contiene lógica sobre los dispositivos, pero sabe interpetar los Commands, solicitar su ejecución, y comenzar un Flow File. Depende directamente de la clase *Maestro*, recibiendo los atributos de los comandos a ejecutar.

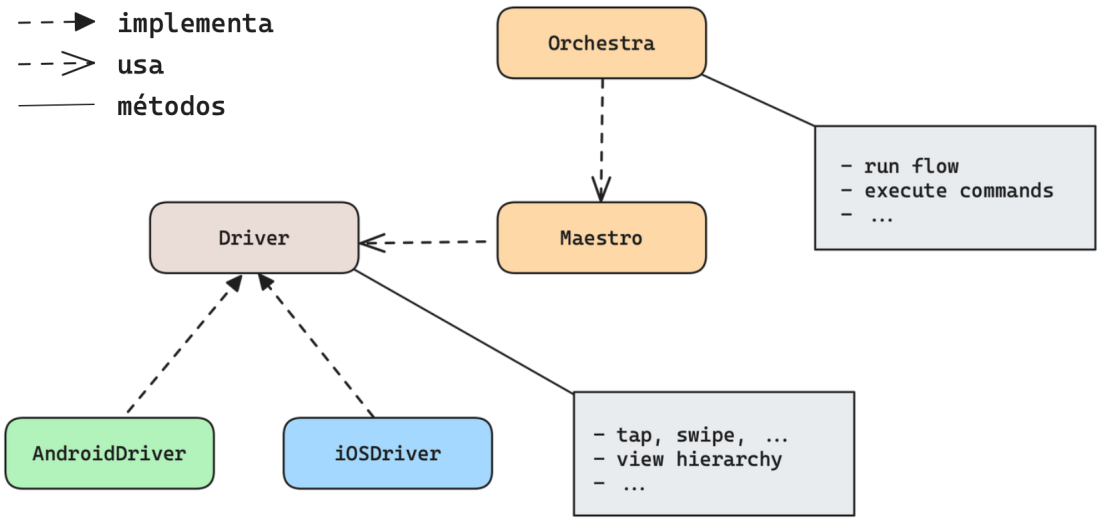

Fig. 2.5: Diagrama de los componentes principales de la arquitectura de Maestro.

- **Maestro:** La clase Maestro es un proxy entre Orchestra y el device, ya que recibe los atributos de los commands, y sabe cómo colaborar con el device para lograr la ejecución real de la acción.
- **Driver:** Es la abstracción de un device y de lo que se espera del mismo. Como es esperable, cada implementación maneja detalles específicos de la plataforma, pero al manejar ambas el mismo protocolo se logra unificar su uso.

3. IMPLEMENTACIÓN

De la misma manera que el equipo de desarrollo de Maestro construyó la clase *Orchestra*, seguiremos esa misma línea creando una clase llamada *TestGenerationOrchestra*. Esta principalmente va a recibir una instancia de *Maestro*, un *PackageName* de la aplicación a explorar, y los componentes necesarios para explorar la aplicación: *ActionHasher*, *HierarchyAnalyzer*, *DisambiguationRules*, *Strategy*. Finalmente creamos al *TestSuiteGenerator*, una clase dedicada a utilizar iterativamente al TestGenerationOrchestra para así producir una test suite. En este capítulo comentaremos a alto nivel cómo está implementado cada componente principal del generador de test suites. Para examinar más detenidamente la herramienta, su código fuente está accesible en un repositorio público en GitHub [15].

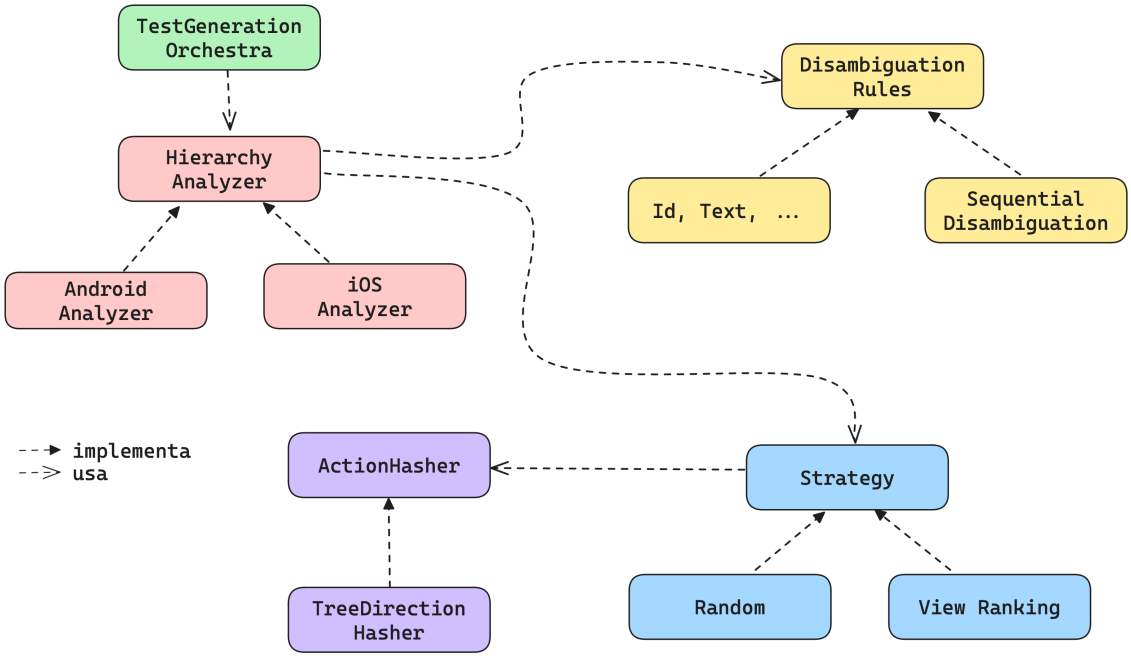

Fig. 3.1: Versión simplificada de la arquitectura del generador de test suites[1]

3.1. Construcción del modelo

Como se ha mencionado en secciones anteriores (Ver 2.7), al hacer model based testing es necesario definir qué significado se le da a los nodos y aristas.

En esta tesis decidimos representar a las acciones como nodos. A su vez, un nodo 'a' tiene una arista a otro 'b' cuando al ejecutar a la acción representada por 'a', la acción 'b' está disponible para ser ejecutada. Si además guardamos si una acción fue ejecutada

[1] Se excluye al TestSuiteGenerator para una mejor visualización. Este simplemente usa al TestGenerationOrchestra.

o no, logramos que con esta representación sea posible recorrer el grafo en búsqueda de acciones sin uso. La motivación de esta representación está fundada principalmente en buscar maximizar la ejecución de acciones sin uso.

Para lograr que cada acción de una aplicación sea representada por un nodo, es necesario asignarle un identificador a cada una. Para ello utilizamos un criterio similar al usado por Baek et al. [4] tomando la secuencia de índices desde la raíz hasta la vista asociada al selector (ver Figura 3.2). La vista asociada al selector es la vista donde se realiza la acción de *tap*. Luego, esa secuencia se codifica utilizando el método `hashCode` de *Kotlin* (el lenguaje en el que Maestro está implementado). Para las acciones que no tienen selector (ya que no se ejecutan sobre ningún elemento en sí) como *BackPress* y *VerticalScroll*, les tomamos como identificador la composición del árbol. Es decir, dejamos únicamente los índices de cada nodo en el árbol, y codificamos el árbol (también con el método `hashCode`).

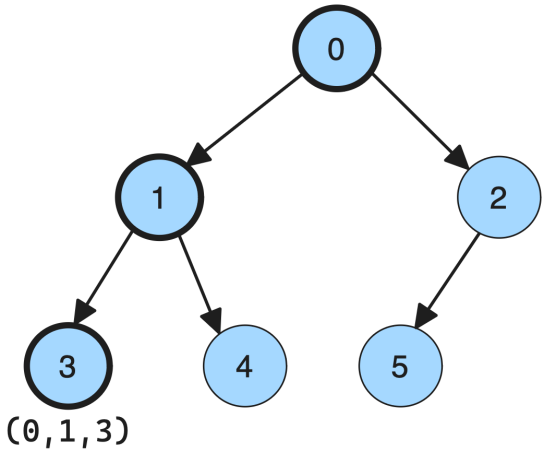

Fig. 3.2: Sea el nodo "3" la vista con el selector asociada, la secuencia de índices es (0,1,3). Esta secuencia se codifica posteriormente haciendo `hashCode("(0,1,3)")`

Sin embargo, este criterio presenta las siguientes limitaciones que podemos observar en la figura 3.3:

- Entre 'Screen A' y 'Screen C' podría tener la misma secuencia de índices para llegar al elemento con texto "Accept", teniendo así una granularidad muy laxa (permito identificar dos elementos distintos como iguales).
- Entre 'Screen A' y 'Screen B', suponiendo que en Screen B el contenido es *scrolleable* (es decir, se mueve verticalmente), la secuencia de índices en Screen B se altera, por ejemplo, si hago un scroll hacia arriba, provocando que el elemento con texto "Accept" ahora esté más cerca de la raíz, teniendo una granularidad más rígida (permito identificar dos elementos iguales como distintos).

Las acciones que vamos a ejecutar son: *Tap on Element*, *BackPress*[2], y *VerticalScroll*. En esta primera implementación se decide excluir las acciones de ingreso de texto *Input Text* por razones de simplicidad. Por un lado, porque las acciones de input en Maestro

[2] Al momento de realizar este trabajo, Maestro solo soporta esta acción para Android

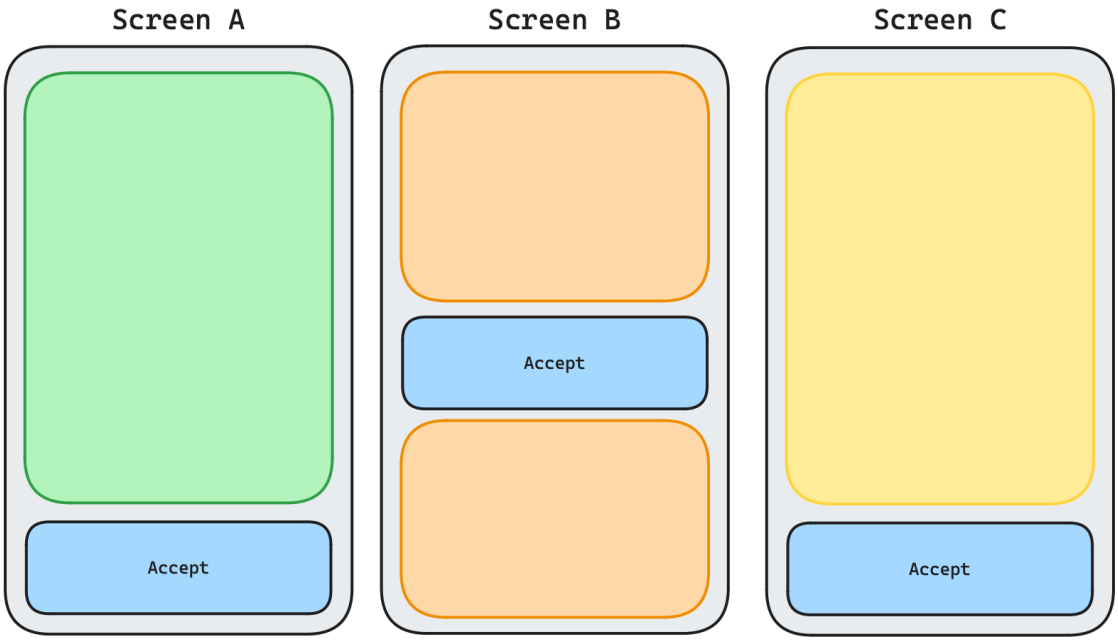

Fig. 3.3: A pesar de que todas las pantallas sean distintas, el criterio falla ante estos casos patológicos.

no son atómicas (es requerido primero presionar algún elemento que abra el teclado, y posteriormente hacer el ingreso), y esto dificulta el hecho de cómo identificarlas, pues tendríamos que mantener un registro de cuál fue la que abrió el teclado. Por otro lado, buscamos evitar los problemas que menciona Nadia Alshahwan en su charla [1] sobre la implementación de View Ranking en Sapienz, que resumidamente es que esta acción requiere de algunas heurísticas extra para funcionar correctamente.

El componente que realiza esta tarea lo nombramos *Action Hasher*. En el Algoritmo 1 describimos su funcionamiento en términos generales.

Algorithm 1: Algoritmo de hashing de acciones

Input : Un comando L_C, una vista V asociada a $L_{Comando}$, y el árbol de vistas R
Output : Un `string` S que representa a L_C

if *type*(L_C) = *Tap* **do**
    `// La acción es de tap sobre` V`, buscamos su camino desde la raiz de` R `a` V
    pathFromRootToV ← pathToView(R, V)
    return hashPath(pathFromRootToV)
else
    `// La acción no es de tap, la vista asociada es el árbol en sí`
    return hashActionOverTree(R, L_C)

3.2. Grafo de representación

Para la representación concreta del grafo utilizamos **dot**, un lenguaje descriptivo en texto plano que se utiliza para definir la estructura y el diseño de grafos de manera textual, permitiendo representar relaciones entre nodos y aristas de forma clara y concisa. Esta herramienta también permite convertir estos archivos de texto en imágenes, como podemos observar en la figura 3.4.

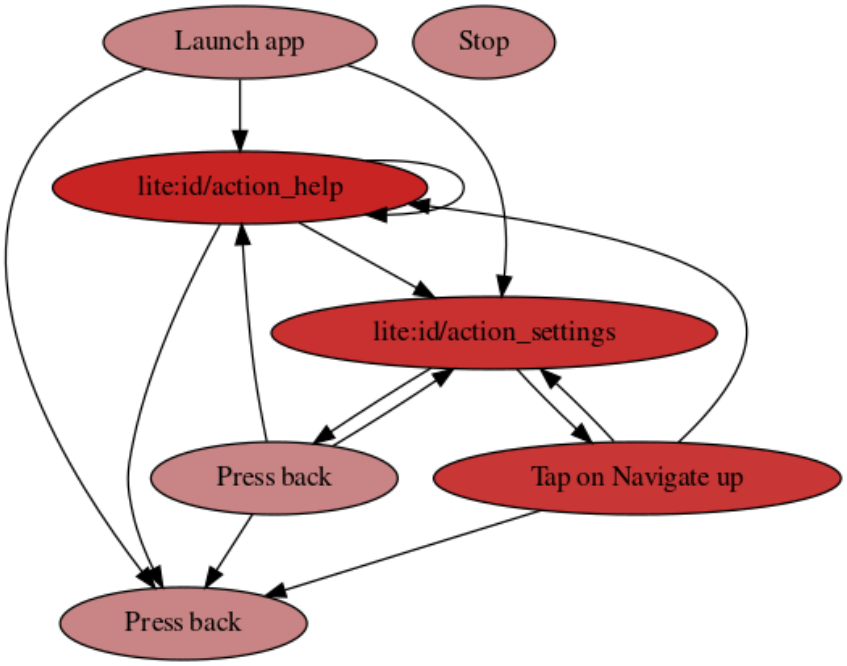

Fig. 3.4: Un grafo de una exploración. A medida que el rojo se oscurece, significa que han sido más usadas.

Aprovechando la sintaxis que ofrece dot agregamos metadata como la cantidad de usos que tuvo cada acción, y así tener una vista general de cuánto se ejecuta cada nodo. A su vez guardamos una versión reducida de la descripción del comando que sirve como *label* del nodo. Internamente cada nodo tiene el `hashCode` específico asociado a la acción, y sus aristas apuntan a los vecinos que descubrió durante la exploración.

3.3. Reglas de desambiguación

Para solucionar el problema de la desambiguación de vistas, implementamos un conjunto de clases llamadas *Disambiguation Rules*. Estas tienen definido un criterio de desambiguación, y tratan de construir un *Selector* para una vista en particular. En caso de no lograrlo, se devuelve un *Selector* vacío.

Esta estrategia de desambiguación es *best-effort*, ya que no siempre es posible desambiguar una vista en un árbol particular (Ver 2.8.2). Las reglas que se definieron para desambiguar vistas son las siguientes:

- **WithUniqueId**: tanto iOS como Android ofrecen mecanismos para asignarles 'identificadores' a vistas particulares para poder referenciarlas desde el código o desde un

test. Dada una vista a desambiguar V, y un árbol R, esta regla retorna un *Selector* que la referencia únicamente si tiene un identificador único en R.

- **WithUniqueText**: la lógica es similar a la anterior, pero revisa si el texto o el texto de accesibilidad (que generalmente no es visible, sino que se usa para el sistema de accesibilidad) son únicos en R.
- **WithUniqueTextAndId**: esta regla mira unicidad tomando las 2-uplas (id, text) en el árbol R para la vista V.

En el Algoritmo 2 describimos el funcionamiento del componente *SequentialDisambiguationRule* (que podemos ver en la figura 3.1), que esencialmente aplica de manera secuencial las reglas mencionadas anteriormente.

Algorithm 2: Algoritmo de desambiguación de vistas

```
Input  : Árbol de vistas R, vista a desambiguar V ∈ R
Output : Selectors S_V asociado a V, o un selector vacío
rules ← [WithUniqueId, WithUniqueText, WithUniqueTextAndId]
for each rule ∈ rules do
    selector ← rule.apply(R, V)
    if selector.isNotEmpty() do
        return selector
return EmptySelector
```

3.4. Estrategias

Una estrategia es un mecanismo para elegir una acción de una lista de acciones. Necesariamente estas tienen que ser las disponibles en la pantalla, si no, no podrán ser ejecutadas.

3.4.1. Random

La idea de random, como ya se ha comentado, es muy directa. Del listado de acciones proporcionado, se elige alguna de manera aleatoria. Para hacer que esta elección sea determinística (en la medida de lo posible), es posible proporcionar una *seed*. Esto no siempre funciona debido a la naturaleza de las aplicaciones móviles (Ver 2.6.3).

Algorithm 3: Algoritmo de selección Random

```
Input  : Lista de comandos L_C
Output : Un comando C ∈ L_C
r ← selectActionRandomly(L_C)
return r
```

3.4.2. View Ranking

View Ranking busca maximizar model coverage. Para eso, internamente va construyendo el modelo mencionado en 3.1. El modelo inicia con un nodo de entrada que es

LaunchApp. Cada vez que recibe el listado de acciones, agrega aristas desde la acción ejecutada anteriormente a cada una de las acciones que recibe. Luego, rankea las acciones recibidas con tres criterios (en ese orden):

- **ActionWasUsed**: Primero **revisa si la acción fue ejecutada** o no. Al priorizar acciones no ejecutadas se asegura de ser lo más exhaustivo posible en cada pantalla.
- **PathLengthToUnusedAction**: Si la acción fue usada, busca en el grafo mediante un algoritmo de camino mínimo la distancia que tiene hasta una acción sin uso.
- **ActionPriority**: Finalmente, asignamos prioridad a cada tipo de acción. Decidimos que las acciones de tipo *Tap* tienen prioridad más alta. Luego le siguen las acciones de tipo *BackPress* (ir hacia atrás). Y por último las acciones de tipo *VerticalScroll*.

Este ranking produce 3-uplas donde la primera coordenada es 1 si la acción fue ejecutada, o 0 si no. La segunda coordenada es la distancia mínima a una acción sin uso, o infinito si no puede alcanzar a ninguna acción sin uso. Y la tercera coordenada es 3 si la acción es de tipo Tap, 2 si es de tipo BackPress, y 1 si es de tipo Scroll.

A modo de ejemplo, podríamos tener el siguiente par de 3-uplas: $v = (1, 2, 3), u = (1, 2, 1)$. En este caso, ambas fueron usadas, ambas tienen la misma distancia a una acción sin uso, pero la prioridad de v es mayor que la de u, por lo tanto $v > u$.

Una vez elegida la acción, aumentamos la cantidad de usos para dicha acción, y la retornamos.

Algorithm 4: Algoritmo de selección View Ranking

Input : Lista de comandos L_C
Output: Comando $C \in L_C \mid \forall \hat{C} \in L_C \; rank(C) \geq rank(\hat{C})$

```
// Actualizamos el modelo anterior con las acciones nuevas que
   recibimos
M_p ← previousModel()
M_u ← updateModelWithActions(M_p, L_C)
// Le asignamos un ranking a cada acción en base a ciertos criterios
   y elegimos la mejor
rankedActions ← rankBy(
    L_C,
    M_u,
    [ ActionWasUsed, ActionPriority, PathLengthToUnusedAction ]
)
bestAction ← selectBest(rankedActions)
// Incrementamos los usos de la acción seleccionada
updateUsagesForAction(M_u, bestAction)
return bestAction
```

3.5. Hierarchy Analyzer

El Hierarchy Analyzer examina la interfaz gráfica de usuario (GUI) para identificar las acciones disponibles, adaptándose a las particularidades de cada plataforma. Por esta

razón, en la Figura 3.1 se observan implementaciones específicas para Android e iOS. Este proceso, que está detallado de manera simplificada en el algoritmo 5, realiza la extracción de elementos de la GUI, la asignación de selectores según reglas de desambiguación, la evaluación de los metadatos de las vistas asociadas para determinar y la viabilidad de acciones de Tap, PressBack, y VerticalScroll. Luego, mediante una estrategia de selección, se elige una acción para ejecutar.

Como se comentó previamente (Ver 2.9.2) la representación de las vistas en Android y iOS difiere. Aunque Maestro intenta unificarlas, persisten ciertas particularidades que justifican el tratamiento separado de sus árboles. Esta diferenciación en los métodos de análisis permite obtener conclusiones más precisas sobre cada nodo, lo que a su vez aumenta el número de acciones disponibles. Por ejemplo, en Android es frecuente encontrar casos como el descrito en 3.1, donde el selector puede aplicarse a un nodo que no es clickable, pero su vista contenedora sí lo es. Para solventar esto, en el *Analyzer* de Android, si una vista es desambiguada pero no es clickeable, revisamos si su padre lo es. De ser así, construimos el selector *ContainsChild*, que recibe un selector. Este nuevo selector hace referencia a la vista que tenga a un hijo con el selector pasado por parámetro. Por construcción, este nuevo selector también está apropiadamente desambiguado.

Algorithm 5: Algoritmo para proveer una acción a ejecutar.

Input : Árbol de vistas R, estrategia de selección S, reglas de desambiguación D
Output : Un comando C

```
1 selectors ← disambiguateWith(R, D)
2 commands ← assignCommands(selectors)
3 commandToExecute ← S.selectOne(commands, R)
4 return commandToExecute
```

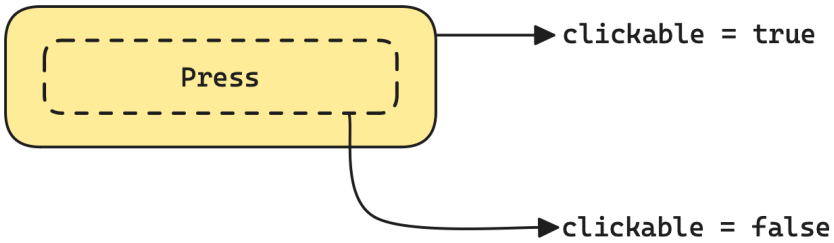

Fig. 3.5: Representación visual del listing 3.1

```
<node clickable="true" other="...">
    <node
        text="Press"
        id="button_press"
        clickable="false"
        className="..."
        checked="..."
        size="...">
    </node>
</node>
```

Listing 3.1: Un nodo con texto 'Press' no es clickeable, pero su vista contenedora lo es.

3.5.1. Generación de Test Suites

Para integrar estos componentes creamos la clase *TestGenerationOrchestra*. De la misma manera que la clase *Orchestra* (implementada por el equipo de Maestro) es la que se encarga de ejecutar las acciones de los test cases; TestGenerationOrchestra realiza la creación de un test case utilizando los componentes previamente mencionados. Esta requiere de los siguientes parámetros:

- packageName: identificador de la aplicación bajo test.
- maestro: análogamente a Orchestra, necesita de una instancia de la clase *Maestro* para poder interactuar con el dispositivo.
- testSize: tamaño del test case (cantidad de acciones) a generar.
- endIfOutsideApp: valor *booleano* para determinar si se desea finalizar el test case en caso de que se ejecute una acción que sale de la aplicación.
- hierarchyAnalyzer: una instancia de la clase HierarchyAnalyzer para analizar la GUI y en base a ello obtener una acción nueva para el test case.

Para la generación del test case, esta clase ofrece el método startGeneration, que dadas las configuraciones comentadas, comienza la generación del test case.

Por último, como buscamos generar test suites, desarrollamos la clase *TestSuiteGenerator*, que recibe parámetros desde la linea de comandos que utiliza para construir una instancia de TestGenerationOrchestra y configurar la generación, definiendo así los siguientes parámetros:

- device: el dispositivo donde se desea realizar la generación de la test suite. Es implícito si solo tenemos conectado un único dispositivo. Determina el Analyzer a utilizar dependiendo si este es Android o iOS.
- testSuiteSize: cantidad de test cases que queremos generar.
- strategy: estrategia de generación que deseamos utilizar en la generación (Random o View Ranking).
- seed: en 3.4.2 comentamos como *rankeamos* acciones, pero es común que sucedan empates. Para esos casos se selecciona alguna de manera aleatoria utilizando una *seed* especificada. También esta seed es la que se utiliza en la estrategia de Random para seleccionar una acción.

Los otros parámetros especificados en el listing 3.3 representan lo mismo que en la clase TestGenerationOrchestra. En los Algoritmos 7 y 6 detallamos las nociones generales de los métodos `generate()` y `startGeneration()` respectivamente. Como señalamos al principio del capítulo, el proceso de generación comienza con el método `generate()`, que utiliza iterativamente al método `startGeneration()` para producir los test cases de la suite. Con esto logramos que el modelo sea utilizado a lo largo de todo el proceso de la generación de la test suite.

Como comentario de implementación adicional, TestSuiteGenerator implementa un patrón *observer* para reaccionar a las actualizaciones del modelo, producto de la ejecución de acciones. Al finalizar la generación de la test suite, el output es la test suite compuesta por la cantidad de test cases especificados, y un archivo en formato **dot** que representa al grafo/modelo de la aplicación (Ver 3.2).

```
class TestGenerationOrchestra(
    maestro: Maestro,
    packageName: String,
    hierarchyAnalyzer: HierarchyAnalyzer,
    testSize: Int,
    endTestIfOutsideApp: Boolean
) {
    fun startGeneration(): List<Command>
}
```

Listing 3.2: Signatura del TestGenerationOrchestra

```
class TestSuiteGenerator(
    maestro: Maestro,
    device: Device,
    packageName: String,
    testSuiteSize: Int,
    testSize: Int,
    endTestIfOutsideApp: Boolean,
    strategy: String,
    seed: Long,
) {
    fun generate()
}
```

Listing 3.3: Signatura del TestSuiteGenerator

Algorithm 6: Algoritmo para generar un test case.

Input : Tamaño de test t_s, instancia de Maestro M, instancia de HierarchyAnalyzer H_A, *booleano* si debe finalizar el test case si se encuentra fuera de la aplicación e_t

Output : Listado de comandos C

```
1  openApplicationWithClearState() // Abrimos la aplicación limpiando la memoria
       de la misma, para que todos los tests comiencen de un mismo punto.
2  C ← [ ]
3  for 0 ≤ i ≤ t_s do
4      hierarchy ← M.viewHierarchy()
5      shouldEndTest ← e_t ∧ isOutsideApp(hierarchy)
6      if shouldEndTest do
7          endTest()
8      c ← H_A.fetchCommandFrom(hierarchy)
9      C.add(c)
10     executeCommand(c)
11     i++
12 return C
```

Algorithm 7: Algoritmo para generar una test suite.

Input : Instancia de Maestro M, instancia de Strategy S, instancia de Device D, tamaño de suite ts_s, tamaño de test t_s, PackageName P_n, *booleano* si debe finalizar el test case si se encuentra fuera de la aplicación e_t

Output : Test suite T, modelo de la aplicación en formato dot G_{dot}

```
1  observeModelUpdates() // Este método que observa las actualizaciones es
       asíncrono. Lanza en otro proceso el mecanismo del observer y continúa
       con la ejecución.
2  H_A ← analyzerFor(D, S) // Construímos el HierarchyAnalyzer para la
       estrategia y el dispositivo dado.
3  testGenerationOrchestra ← TestGenerationOrchestra(M, P_n, H_A, t_s, e_t)
4  T ← [ ]
5  for 0 ≤ i ≤ ts_s do
6      C ← testGenerationOrchestra.startGeneration()
7      testCase ← produceFlowFileFromCommands(C)
8      T.add(testCase)
9  G_dot ← buildDotModel()
10 return ⟨T, G_dot⟩
```

3.5.2. Ejemplo de uso

Para ilustrar el funcionamiento de la herramienta, proporcionamos un ejemplo práctico de cómo genera una test suite.

Seleccionamos los siguientes parámetros para la generación:

- Tamaño de Test Suite: 3.
- Tamaño de Test Case: 5.
- Estrategia: View Ranking.
- Aplicación: `com.google.android.contacts` (aplicación de contactos por defecto de Android).
- Dispositivo: Emulador de *Google Pixel*.
- Seed: Default.
- Finalizar test si se abandona la aplicación bajo test.

Para ello entonces debemos ejecutar (en la raiz del directorio del repositorio) el comando del Listing 3.4. Esto iniciará el proceso de generación, el cuál nos irá guiando qué decisiones está tomando el algoritmo (Figura 3.6). Al finalizar, guarda en el directorio `generated-flows/{packageName}` la test suite producida junto al archivo en formato `dot`.

```
./run-maestro generate com.android.contacts --testSize 5 --suiteSize 3 --
    endIfAppLeft --strategy viewranking
```

Listing 3.4: Ejemplo de cómo iniciar la generación de una test suite.

```yaml
appId: "com.google.android.contacts"
---
- launchApp:
    appId: "com.google.android.contacts"
    clearState: true
    stopApp: true
- waitForAnimationToEnd:
    timeout: 2000
- tapOn:
    id: "com.google.android.contacts:id/open_search_bar"
- tapOn:
    text: "Company"
- tapOn:
    text: "Phone contacts"
- tapOn:
    text: "More options"
- tapOn:
    text: "Open navigation drawer"
```

Listing 3.5: Ejemplo de un test case generado

```
- Executing command (0, Launch app "com.google.android.contacts" with clear state) ⌛
- Command (0,Launch app "com.google.android.contacts" with clear state) completed successfully ✅
- Executing command (1, Tap on "Company") ⌛
- Command (1,Tap on "Company") completed successfully ✅
- Executing command (2, Tap on id: com.google.android.contacts:id/touch_outside) ⌛
- Command (2,Tap on id: com.google.android.contacts:id/touch_outside) completed successfully ✅
- Executing command (3, Tap on "Phone contacts") ⌛
- Command (3,Tap on "Phone contacts") completed successfully ✅
- Executing command (4, Tap on Contains child: id: com.google.android.contacts:id/og_selected_account_disc_apd) ⌛
- Command (4,Tap on Contains child: id: com.google.android.contacts:id/og_selected_account_disc_apd) completed successfully ✅
- Executing command (5, Tap on id: com.google.android.contacts:id/my_account_chip) ⌛
- Command (5,Tap on id: com.google.android.contacts:id/my_account_chip) completed successfully ✅
- Previous action left the app. Perhaps a pop-up or something, attempting to go back with BackPress ⬅
- Adding a BackPress to dismiss an exit of the app ⬅...
- Brewing up Flow File 1 ☕...
- Executing command (0, Launch app "com.google.android.contacts" with clear state) ⌛
- Command (0,Launch app "com.google.android.contacts" with clear state) completed successfully ✅
- Executing command (1, Tap on id: com.google.android.contacts:id/open_search_bar) ⌛
- Command (1,Tap on id: com.google.android.contacts:id/open_search_bar) completed successfully ✅
- Executing command (2, Tap on "Company") ⌛
- Command (2,Tap on "Company") completed successfully ✅
- Executing command (3, Tap on "Phone contacts") ⌛
- Command (3,Tap on "Phone contacts") completed successfully ✅
- Executing command (4, Tap on "More options") ⌛
- Command (4,Tap on "More options") completed successfully ✅
- Executing command (5, Tap on "Open navigation drawer") ⌛
- Command (5,Tap on "Open navigation drawer") completed successfully ✅
- Brewing up Flow File 2 ☕...
- Executing command (0, Launch app "com.google.android.contacts" with clear state) ⌛
- Command (0,Launch app "com.google.android.contacts" with clear state) completed successfully ✅
- Executing command (1, Tap on id: com.google.android.contacts:id/selected_account_disc) ⌛
- Command (1,Tap on id: com.google.android.contacts:id/selected_account_disc) completed successfully ✅
- Executing command (2, Tap on id: com.google.android.contacts:id/og_header_close_button) ⌛
- Command (2,Tap on id: com.google.android.contacts:id/og_header_close_button) completed successfully ✅
- Executing command (3, Tap on Contains child: id: com.google.android.contacts:id/open_search_bar_text_view) ⌛
- Command (3,Tap on Contains child: id: com.google.android.contacts:id/open_search_bar_text_view) completed successfully ✅
- Executing command (4, Tap on "Company") ⌛
- Command (4,Tap on "Company") completed successfully ✅
- Executing command (5, Tap on id: com.google.android.contacts:id/open_search_view_mic_button) ⌛
- Command (5,Tap on id: com.google.android.contacts:id/open_search_view_mic_button) completed successfully ✅
- Previous action left the app. Perhaps a pop-up or something, attempting to go back with BackPress ⬅
- Adding a BackPress to dismiss an exit of the app ⬅...
- Brewing up Flow File 3 ☕...
- Building visual graph model 🌄
```

Fig. 3.6: Logs del proceso de generación.

4. EVALUACIÓN

En este capítulo, presentaremos las preguntas de investigación y el contexto en el que se llevó a cabo la experimentación para abordarlas. Además, describiremos detalladamente los resultados obtenidos, proporcionando explicaciones pertinentes. Finalmente, contestaremos a las preguntas de investigación en función de los resultados obtenidos.

4.1. Preguntas de investigación

En esta tesis proponemos comparar View Ranking contra Random, evaluando efectividad, y eficiencia. A su vez, se presenta un análisis cualitativo de los modelos generados tanto para iOS como Android, para así determinar limitaciones y puntos de mejora.

RQ1. Efectividad: Respecto al aporte de la heurística de View Ranking. ¿Cuánto aporta este algoritmo respecto al Model Coverage comparándolo con Random? ¿Y respecto al Code Coverage[1]?.

La naturaleza del algoritmo es, en esencia, ejecutar todas las acciones que descubre, por lo que es esperable observar una mejora en el Model Coverage. Pero no es claro qué sucede con el Code Coverage, ya que algunas funcionalidades internas solo podrían ejecutarse luego de realizar varias veces la misma acción (algo que el algoritmo trata de evitar). Cabe aclarar que el análisis de Code Coverage solo será realizado en Android, pues no tenemos manera de instrumentar las aplicaciones de iOS.

RQ2. Eficiencia: ¿Qué tan eficiente es View Ranking a lo largo de la exploración para obtener un Model Coverage con respecto a Random? ¿Es posible que ambas estrategias converjan al mismo Model Coverage?

Es deseable saber qué tan rápido View Ranking logra crear un buen modelo de la aplicación bajo test, para posteriormente hacer algún tipo de optimización de corte, y así evitar tiempo de cómputo en un modelo que ya no va a progresar. También es interesante saber qué tanto difiere del que va generando Random a lo largo de la exploración, para, por ejemplo, observar cuando cada modelo llega a su "máximo local".

RQ3. Android vs iOS: Respecto a los modelos generados por View Ranking para cada plataforma. ¿Cómo difieren estos? ¿Qué dificultades tiene cada uno? ¿Qué optimizaciones o mejoras podrían aplicarse para lograr un modelo más robusto?

Cada plataforma tiene su propia manera de representar árboles de vistas, por lo que es esperable que encontremos diferencias en los modelos generados, aunque estos se realicen sobre una misma aplicación. Haremos un análisis cualitativo de estos modelos, revisando en detalle estas diferencias.

4.2. Entorno de ejecución

La experimentación fue particionada en dos, siendo una ejecutada en una *MacBook Pro* con sistema operativo *macOS Sonoma 14*, con 16GB de memoria RAM, procesador *Apple M2 Pro*, y otra en una computadora de escritorio con sistema operativo *Ubuntu 22.04*, 24GB de memoria RAM, GPU *Radeon RX 570* 4GB, y un procesador *Intel i5-9600K*.

[1] En este trabajo tomaremos Line Coverage.

En cuanto a dispositivos donde se realizó la generación de los tests, para Android utilizamos un emulador *Pixel 2* con *API 26*. Para iOS seleccionamos un simulador iPhone 15 Pro Max con versión de iOS 17.

4.3. Elección de sujetos

Para contestar **RQ1** y **RQ2** construimos un conjunto de sujetos –aplicaciones– tanto de Android como de iOS, con la salvedad mencionada de que la evaluación de Code Coverage solo será sobre el subconjunto de sujetos de Android.

Luego, para poder contestar para contestar **RQ3** seleccionamos aplicaciones *multiplataforma* (ejecutables en ambos sistemas) hechas en Kotlin Multiplatform (KMM) [13].

El listado detallado de los sujetos seleccionados se encuentra en el Anexo (6).

4.3.1. Android

Para Android seleccionamos 25 aplicaciones *open-source* de *F-Droid* [22]. Para ello se armó un *script* que itera sobre las diferentes categorías dentro de F-Droid, y extrae sus PackageName's. Luego tomamos un *sample* uniforme para tener una selección equivalente de cada categoría. Finalmente, para poder obtener Code Coverage de las mismas, se instrumentaron utilizando la herramienta *WallMauer* [3].

Previo a la experimentación se probó de manera manual que las aplicaciones seleccionadas puedan ser instaladas y que al abrirlas e interactuar brevemente no presenten *crashes*. Esto se hizo debido a que las apps de F-Droid a veces no funcionan, y además, instrumentarlas a nivel *byte-code* puede provocar que no puedan ser instaladas o siquiera ejecutar en el dispositivo.

4.3.2. iOS

Para iOS, no hemos encontrado un método directo y sencillo para instalar aplicaciones de terceros en el simulador nativo, excepto mediante la compilación del código fuente (proceso que consume bastante tiempo). Por eso, seleccionamos algunas de las aplicaciones pre-instaladas en el dispositivo para realizar las pruebas. Estas son: Calendar, Remainders, Messages, Files, Contacts, y AppleNews.

4.3.3. Android + iOS

Como los sujetos multiplataforma seleccionados son para un análisis cualitativo, tomamos solo dos del listado de *KMM Samples* [13]. Siendo estas: *Confetti*, y *KMPizza*.

4.4. Elección de parámetros

Tamaño de los test generados: para cada aplicación se generaron 20 casos de test con a lo sumo 15 acciones. Antes de fijar este número, analizamos variar tanto cantidad de tests generados, como su longitud de acciones. Creemos que este número mantiene una buena relación en cuanto a *tests-longitud*, manteniendo una interpretación viable de estos, y evitando un uso excesivo del tiempo de experimentación.

Estado de la aplicación: para mantener independencia entre cada caso de test, antes de generar cada test, limpiamos la memoria interna de la aplicación.

Límites de la exploración: es común que una acción nos lleve por fuera de la aplicación. Tomamos la decisión de dar por finalizado el caso de test en caso de ser así.

4.5. Proceso experimental

Para cada par $(App, Strategy)$ ejecutamos 3 veces la generación de casos de test. Esto se debe a la naturaleza de las apps que no suelen ser totalmente determinísticas, y así tener una imagen en promedio de cada proceso de generación.

Android: La generación de cada test de longitud 15 lleva a lo sumo 3 minutos. Por lo tanto, la experimentación realizó 25 apps $\times$ 20 test cases $\times$ 2 strategies $\times$ 3 attempts $\times$ 3 minutos $= 6,25$ días. Sumado a esto, para evaluar el CodeCoverage se deben re-ejecutar las test suites generadas, lo que casi duplica el tiempo anteriormente mencionado.

iOS: El cálculo aquí es similar, realizando así 6 apps $\times$ 20 test cases $\times$ 2 strategies $\times$ 3 attempts $\times$ 3 minutos $= 1,5$ días.
Como para iOS no hubo evaluación de Code Coverage, ese tiempo extra no es agregado en este caso.

4.5.1. Medición de Code Coverage

Para medir el Code Coverage (sobre las aplicaciones de Android instrumentadas), WallMauer ofrece un módulo para evaluar *branch coverage* y *line coverage*. Internamente guarda en la memoria del emulador información del código ejecutado, que posteriormente utiliza para realizar la evaluación.

Una vez generados los casos de tests, los re-ejecutamos, tomamos dicha información del código ejecutado, y guardamos en un **.csv** los resultados de la evaluación de Code Coverage.

4.5.2. Medición de Model Coverage

El Model Coverage que tomamos para este trabajo es:

$$\text{ModelCoverage}(S) = \frac{\#(\text{E}_S)}{\#(\text{D}_{Random} \bigcup \text{D}_{ViewRanking})}$$

donde S es alguna de las dos estrategias, E_S es el conjunto de nodos ejecutados por la estrategia S, y D_S es el conjunto de nodos descubiertos por la estrategia S para el proceso de generación de tests. Es decir, para medir qué tanto cubrimos el modelo, dividimos la cantidad de acciones ejecutadas, sobre la cantidad de nodos descubiertos por ambas estrategias. Tomamos la unión de los nodos de ambos modelos para crear una noción de modelo "global".

4.5.3. Medición de Flakiness

El *flakiness* es algo común en el testing de aplicaciones móviles [19]. Para medir qué tan flaky son los test generados, tomamos de manera aleatoria 10 aplicaciones junto a alguna de las 3 suites generadas para cada estrategia, y las re-ejecutamos 5 veces. Un test case será flaky si el resultado difere en alguna de las ejecuciones. El porcentaje de flakiness de una test suite será

$$\text{Flakiness(Suite)} = \frac{\#(\text{Flaky Tests})}{\#(\text{Tests})}.$$

4.5.4. Análisis de Efectividad para Model Coverage y Code Coverage

Para contestar **RQ1** analizamos los datos obtenidos en cuanto a Model y Code Coverage para ambas estrategias siguiendo las pautas para la ingenería de software experimental presentadas por Arcuri et al. [2]. Nuestras muestras serán $MC_{vr}, MC_r, CC_{vr}, CC_r$, donde MC es representa el Model Coverage, CC el Code Coverage, y vr, r View Ranking y Random respectivamente.

Para cada par (MC_{vr}, MC_r), (CC_{vr}, CC_r) utilizamos el test no paramétrico U de *Mann-Withney* para analizar la **hipótesis nula**: dada una generación al azar de tests para cada una de las estrategias, es igual de probable que el Coverage obtenido para una sea igual a la otra. Es decir, la hipótesis nula nos dice que no hay diferencia estadística entre la efectividad de las dos estrategias a nivel del tipo de Coverage seleccionado. Para determinar si se puede descartar o no la hipótesis nula vamos a tomar un nivel de significancia $\alpha = 0,05$ recomendado por Arcuri.

El test U de Mann-Whitney, al ser no paramétrico, no necesita que las muestras provengan de una distribución normal, a diferencia de pruebas paramétricas como el test t. Cuando se detecta una diferencia estadísticamente significativa entre dos variantes, empleamos la medida de efecto no paramétrica $\hat{A}_{12}$ de *Vargha y Delaney* para evaluar la magnitud de esta diferencia. Esta medida es fácil de interpretar en comparación con otras disponibles. Por ejemplo, si consideramos dos variantes $\mathcal{A}$ y $\mathcal{B}$, un valor de $\hat{A}_{12} = 0.7$ indica que uno tendría mejores resultados utilizando la variante A aproximadamente el 70 % del tiempo. Clasificamos las diferencias entre variantes como pequeñas, medianas y largas cuando el valor de $\hat{A}_{12}$ supera 0.56, 0.64 y 0.71 respectivamente.

La fórmula para calcular la medida de efecto $\hat{A}_{12}$ de *Vargha y Delaney* es

$$\hat{A}_{12} = ((R_1)/m - (m+1)/2)/n$$

Donde m es la cantidad de observaciones para la primer muestra de datos, y n la cantidad para la segunda. En nuestro caso, $n = m$, dado que tenemos la misma cantidad de mediciones de Model y Code Coverage para cada estrategia. R_1 es la suma del *ranking* del primer grupo bajo comparación.

4.5.5. Análisis de Eficiencia

Para contestar **RQ2** consideramos las siguientes métricas de eficiencia: por un lado la tasa de re-ejecución de acciones. Para ello anotamos por cada acción cuántas veces fue ejecutada. Luego tomamos

$$\text{Re-ExecutionRate}(S) = 1 - \frac{\#(E_S)}{\sum_{e \in E_S} \text{Usages(e)}}$$

De la misma manera que para la efectividad, haremos un análisis mediante el test U de *Mann-Whitney*, tomando el mismo nivel de significancia de α = 0,05, donde la hipótesis nula es que dadas dos test suites generadas por cada estrategia, la probabilidad de tener una tasa de re-ejecución menor es equivalente entre estas. En caso de rechazarse, mediremos el efecto también con la medida de *Vargha y Delaney*.

Por otro lado, llamamos *step* de exploración a cada acción que genera el proceso de generación. Como estamos generando 20 test cases con a lo sumo 15 acciones cada uno, como máximo tendremos 300 *steps*. En cada step medimos la cantidad de nodos ejecutados.

Con esto podemos armar la evolución que presenta cada estrategia respecto al total del modelo.

Es decir, en cada step podemos tomar

$$\text{ModelCoverageAt}(step_i, S) = \frac{\#\text{ExecutedNodesAt}(step_i, S)}{\#(\text{D}_{Random} \bigcup \text{D}_{ViewRanking})}$$

Esto nos permite evaluar la efectividad con la que cada estrategia mejora el Model Coverage.

Para ambas métricas tomaremos el promedio entre todas las aplicaciones.

4.5.6. Análisis Cualitativo

Por último, para contestar **RQ3**, analizaremos los modelos y test suites producidos para dos aplicaciones multiplataforma. Trataremos de encontrar patrones, diferencias, y problemas que estos podrían presentar.

4.6. Resultados

A continuación presentamos los resultados de los experimentos realizados en 4.5 para contestar las preguntas de investigación.

4.6.1. Efectividad

La tabla 4.1 muestra los *p–valores* obtenidos del test *U* de *Mann-Whitney*. Como conclusión de esto, se rechaza la hipótesis nula para el análisis de Model Coverage, pero no para el Code Coverage. Es decir, hay diferencia estadística entre el Model Coverage logrado por cada estrategia. En cambio, para Code Coverage, no hay evidencia suficiente que me permita afirmar lo mismo.

Coverage	*p–valores*
Model	0,0007
Code	0,765

Tab. 4.1: p–valores obtenidos del test *U* de *Mann-Whitney* para cada Coverage.

Luego, la medida de efecto $\hat{A}_{12}$ de *Vargha y Delaney* fue de 0,647 para el análisis de Model Coverage. Es decir, aproximadamente 65 % de las veces los test generados por View Ranking obtienen mejor Model Coverage que los generados por Random. Este número respalda el gráfico de la Figura 4.2, donde la mayoría de *buckets* estan del lado positivo, y con diferencias notables.

Respecto al análisis de Code Coverage, revisando *traces* generados notamos que en algunos casos hay código que solo es ejecutado cuando una acción es realizada varias veces (llegar al límite de un contador, por ejemplo). Esto es comportamiento que View Ranking trata de evitar, ya que prioriza ejecutar acciones que no ha utilizado. En 4.1 podemos observar que ambas *boxes* son similares. Queda pendiente un análisis más profundo para entender esta similitud, quizá dando pie a alguna optimización de View Ranking.

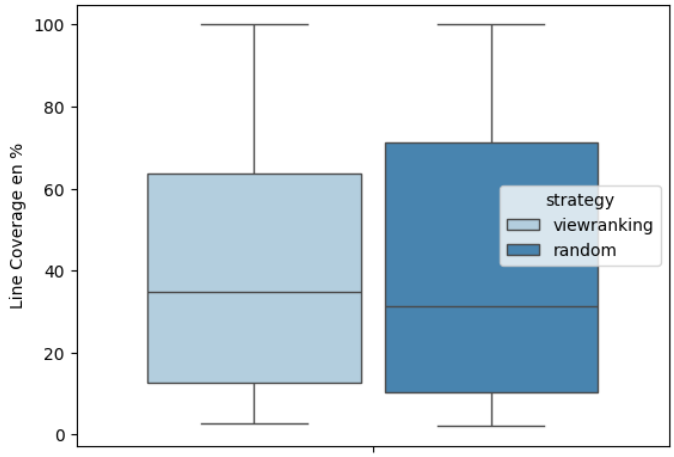

Fig. 4.1: Line Coverage obtenido por cada estrategia.

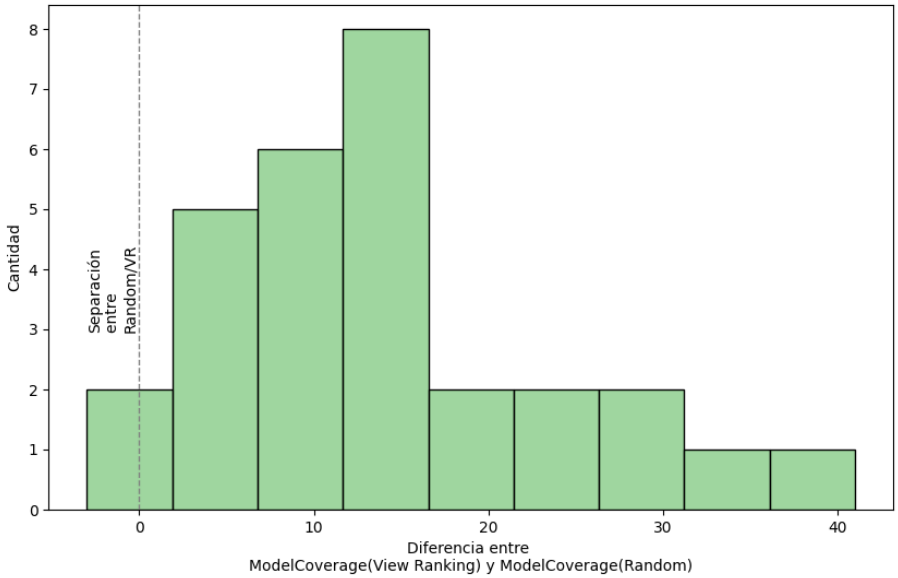

Fig. 4.2: Efectividad de View Ranking contra Random

4.6.2. Eficiencia

Obtuvimos un *p–valor* de 0,039, por lo que podemos rechazar la hipótesis nula. Podemos afirmar entonces que existe diferencia estadística entre la tasa de re-ejecucion de cada estrategia. Luego, la medida de efecto $\hat{A}_{12}$ dio 0,60. Por lo que un 60 % de las veces los test generados por View Ranking tienen una tasa de re-ejecución menor que Random.

Esto ayuda a entender el gráfico de la Figura 4.3, donde podemos ver cómo random se ameseta mucho más pronto, re-ejecutando acciones previamente utilizadas, mientras que View Ranking puede continuar aumentando el Model Coverage.

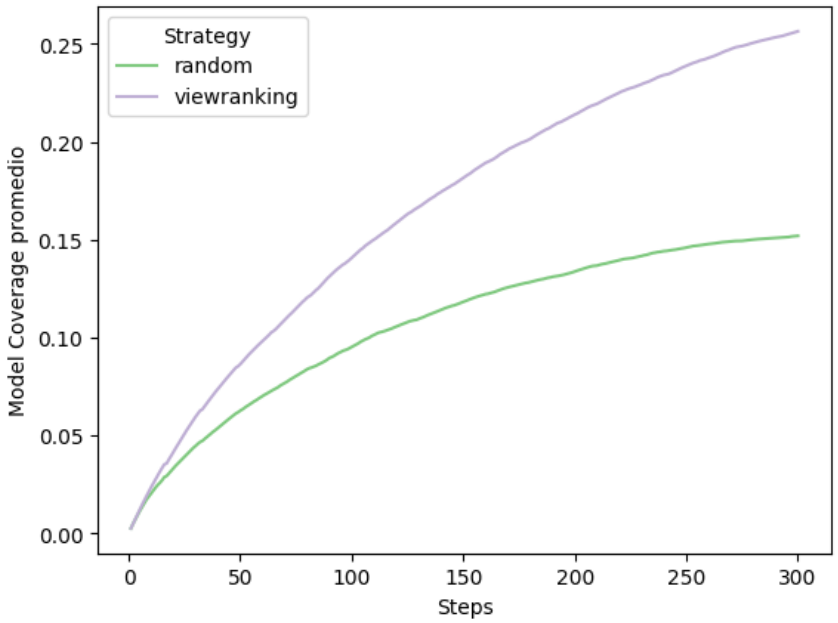

Fig. 4.3: Eficiencia promedio en cada paso por estrategia.

4.6.3. Flakiness

El chequeo de *flakiness* arrojó un porcentaje bajo, por lo que podemos concluir que las test suites generadas son, en general, re-ejecutables.

Strategy	Flakiness
Random	5.5 %
View Ranking	2.5 %

Tab. 4.2: Flakiness promedio calculado para cada estrategia.

Evidentemente esto depende de la aplicación a explorar, ya que algunas pueden tener una naturaleza muy dinámica, haciendo que los test generados rápidamente queden en desuso (por ejemplo, una aplicación de noticias que actualiza diariamente su contenido).

4.6.4. Análisis Cualitativo

Después de examinar el proceso de generación para ambas aplicaciones, se observa que en el caso de la aplicación Confetti, iOS presenta limitaciones para capturar acciones de la GUI, limitándose principalmente a realizar acciones de VerticalScroll. Por el lado de Android, este logra extraer acciones de la GUI, y termina produciendo un modelo apropiadamente.

Luego, en relación a la aplicación KMPizza, se observa que iOS produce un modelo excesivamente extenso debido a cómo interpeta qué vistas son clickeables. Como mencionamos previamente en la Sección 2.9.2, cada sistema representa independientemente sus vistas. En este caso, iOS parece indicar que un elemento que no tiene una acción asignada es clickeable, lo que resulta en la ejecución de acciones sin efecto alguno y, por ende, en

la generación de un modelo densamente interconectado (la imagen de este modelo era demasiado grande para incluirla en esta sección, por lo que fue trasladada al anexo 6.1). Esto conlleva a la creación de casos de prueba que carecen de utilidad, ya que muchas acciones no tienen impacto alguno. Por el contrario, en la Figura 4.4 se puede apreciar como Android logra generar un modelo más representativo de las acciones disponibles para el usuario.

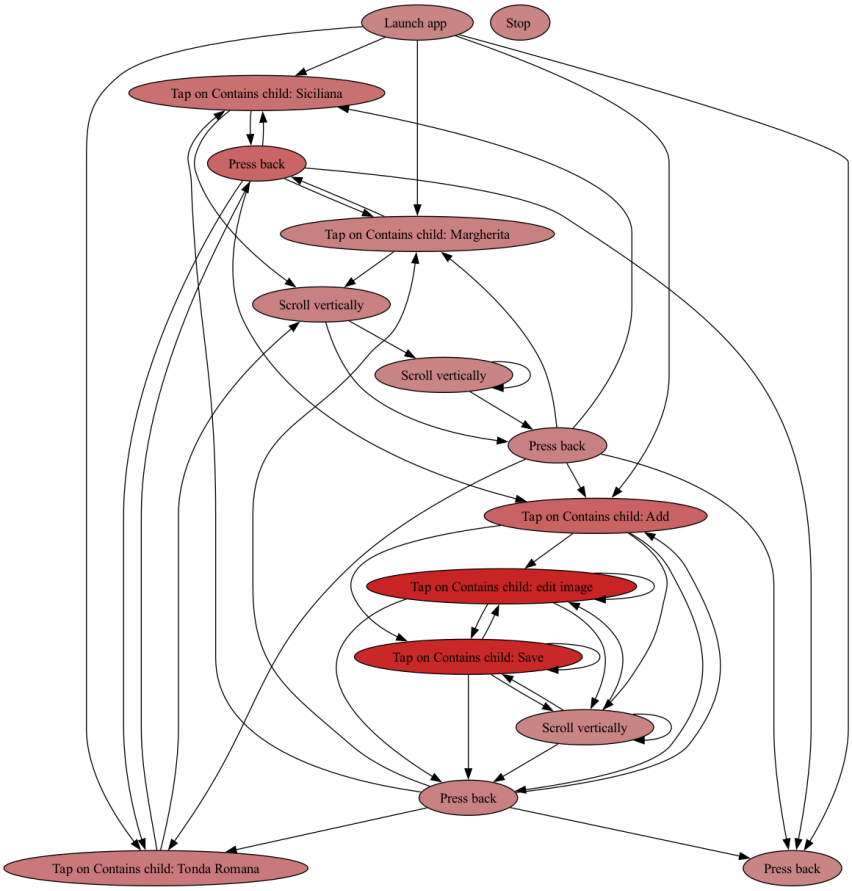

Fig. 4.4: Modelo generado de KMPizza para Android.

4.7. Respuestas a las preguntas de investigación

RQ1. Efectividad: en el análisis empírico realizado, podemos afirmar con una significancia de $\alpha = 0,05$ que la efectividad en cuanto al Model Coverage que logra View Ranking es superior a la estrategia Random. Además, podemos decir gracias a la medida de efecto $\hat{A}_{12}$ que caemos en el umbral de medida de efecto *mediana*.

Sin embargo, no pudimos decir lo mismo para Code Coverage, ya que para ambas estrategias se obtuvieron estadísticamente las mismas métricas.

Esto abre la puerta a pensar en optimizaciones sobre el algoritmo de View Ranking. Es probable que los comandos elegidos para realizar sean pocos, haciendo que el código "alcanzable" por estos comandos sea acotado.

Por otro lado, el Model Coverage obtenido muchas veces se ve perjudicado por acciones que luego no pueden volver a ser alcanzadas. Esto se debe a que acciones como *Scroll* o *Back* dependen del árbol actual, haciendo que la granularidad sea bastante fina. Podemos verificar revisando los grafos generados que las principales acciones que no termina siendo ejecutadas son las anteriormente mencionadas. Una posible optimización es mejorar el mecanismo de *Hashing* para estas acciones.

RQ2. Eficiencia: en el estudio empírico realizado, pudimos afirmar con una significancia de $\alpha = 0,05$ que View Ranking es más efectivo que Random (para la métrica específica que estamos midiendo de model coverage). Nuevamente, el umbral logrado entra en la categoría de medida de efecto *mediana*.

RQ3. Android vs iOS: durante el análisis cualitativo se ha identificado que iOS tiene dificultades para determinar cuándo un elemento es clickeable, provocando que los modelos generados sean poco indicativos (6.1).

5. CONCLUSIONES

En este trabajo presentamos View Ranking, un algoritmo model based para generar test cases de aplicaciones móviles. Para implementarlo elegimos hacerlo sobre Maestro, dado que permite escribir test cases tanto para Android como iOS. Luego, definimos el modelo para el algoritmo de manera tal que cada nodo represente las acciones disponibles para realizar en una aplicación, y las aristas cómo estas se relacionan entre sí. Para poder construir el modelo de esa manera, fue necesario utilizar conceptos como la desambiguación de vistas [18] e identificación de vistas [4].

Finalmente realizamos un análisis cuantitativo y cualitativo. Respecto al cuantitativo, estudiamos la efectividad del algoritmo de View Ranking comparado contra Random (Ver 4) en cuanto su Model y Code Coverage logrado, la eficiencia que tiene para utilizar acciones sin uso, y la velocidad en la que aumenta su Model Coverage a lo largo de su fase exploratoria. En el mismo pudimos afirmar que, mediante un test estadístico, View Ranking supera en estas métricas a Random, exceptuando por el Code Coverage. Por el lado del análisis cualitativo, buscamos examinar las diferencias de los modelos generados de View Ranking para Android y iOS. Para que esta comparación mantuviera coherencia, se seleccionaron dos aplicaciones multiplataforma (es decir, ejecutables tanto en iOS como Android). Aquí pudimos observar que los modelos generados por iOS presentan problemas debido a la incorrecta detección de elementos clickeables.

En síntesis, el algoritmo de View Ranking es una heurística efectiva y eficiente para incrementar el Model Coverage, y crear modelos de aplicaciones móviles. A su vez, la herramienta logra generar test suites re-ejecutables para iOS y Android. Por otro lado, Android presenta una facilidad más amplia a la hora de representar sus vistas en comparación con iOS.

5.1. Trabajo Futuro

Algunos de los puntos de mejora que hemos notado y creemos que podrían mejorar la implementación actual de View Ranking, son los siguientes:

- Mejora en el mecanismo de hasheo: como hemos mencionado en 4.7, el mencanismo de identificación de acciones para aquellas que no tienen un selector asociado actualmente tiene un criterio bastante granular, lo que provoca un crecimiento de estados.
- Inclusión de más comandos: los comandos elegidos probablemente estén restringiendo el mundo de posibilidades para interactuar con una aplicación. Agregar más implicaría tener más chances de ejecutar código interno de la aplicación, de este modo incrementando el Code Coverage.
- Mejora en el análisis de las vistas de iOS: las vistas en iOS requieren de un tratamiento más especial que en Android. Mejorar su análisis podría revertir lo visto en el análisis cualitativo (Ver 4.6.4).
- Utilizar el modelo producido para derivar casos de test: como hemos mencionado en (1), el objetivo del *Model Based Testing* es poder tener un modelo del cual derivar

casos de test. Es interesante ver qué tan útiles podrían ser los test generados directamente desde del modelo producido por la herramienta presentada en esta tesis.

- Agregado de assertions: Por último, los tests generados en esta tesis no cuentan con aserciones. En [18] se comentan procedimientos para agregarlos, y así poder generar test suites más interesantes.

6. ANEXO

Plataforma	PackageName	Version
iOS	com.apple.DocumentsApp	17.4.0
iOS	com.apple.MobileAddressBook	17.4.0
iOS	com.apple.MobileSMS	17.4.0
iOS	com.apple.mobilecal	17.4.0
iOS	com.apple.news	17.4.0
iOS	com.apple.reminders	17.4.0
Android	bored.codebyk.mintcalc	1.1.1
Android	com.NightDreamGames.Grade.ly	2.6.3
Android	com.benarmstead.simplecooking	1.0.9
Android	com.better.alarm	3.15.04
Android	com.harr1424.listmaker	2.3.0
Android	com.minimalisticapps.priceconverter	2.8.0
Android	com.money.manager.ex	2024.01.17
Android	com.sesu8642.infusion_timer	1.4.0
Android	com.trianguloy.urlchecker	2.13.1
Android	com.wmstein.tourcount	3.4.5
Android	de.salomax.currencies	1.15.0
Android	dev.bartuzen.qbitcontroller	0.8.4
Android	dev.linwood.butterfly.nightly	2.0.1
Android	dev.randombits.intervaltimer	1.0.7
Android	info.metadude.android.congress.schedule	1.63.2
Android	livio.rssreader	1.0.3
Android	me.johnmh.boogdroid	0.0.3
Android	nl.viter.glider	2.8.0
Android	org.fdroid.fdroid	1.19.0
Android	org.koitharu.kotatsu	6.6.1
Android	org.sirekanyan.outline	0.1.24
Android	org.transdroid.lite	2.5.24
Android	org.wikipedia	2.7.5
Android	br.com.colman.petals	3.20.2
Android	com.ferrarid.converterpro	4.2.0
Android+iOS	dev.johnoreilly.confetti	0.8.14
Android+iOS	dev.tutorial.kmpizza	1.0.0

Tab. 6.1: Listado de aplicaciones utilizadas con sus versiones

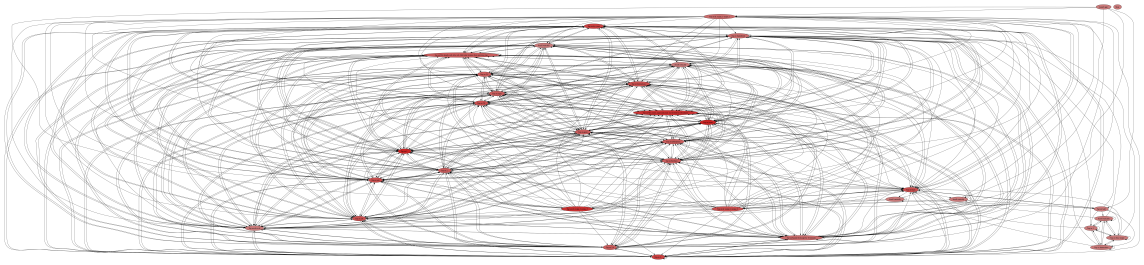

Fig. 6.1: Modelo de KMPizza para iOS.

Bibliografía

[1] Nadia Alshahwan. Talk by Nadia Alshahwan. https://www.youtube.com/live/BM89PFDwZuU?si=QE5OGFFfLQ3W95j9, 2020. ISSTA.

[2] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[3] Michael Auer. Wallmauer: Robust code coverage instrumentation for android apps. *AST 2024*, 2024.

[4] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 238–249, New York, NY, USA, 2016. Association for Computing Machinery.

[5] Edsger Dijkstra. Notes on structured programming. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF, 1970.

[6] Marcelo Medeiros Eler, Jose Miguel Rojas, Yan Ge, and Gordon Fraser. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126, 2018.

[7] Facebook. Sapienz at Facebook's App. https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/, 2018.

[8] World Economic Forum. 4 charts that explain the decline of the pc. https://web.archive.org/web/20231204074823/https://www.weforum.org/agenda/2016/04/4-charts-that-explain-the-decline-of-the-pc/, 2016.

[9] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[10] W.E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[11] Google Inc. Android Open Source Project. https://source.android.com, 2008.

[12] Google Inc. Espresso. https://developer.android.com/training/testing/espresso, 2016.

[13] JetBrains. Kmm samples. https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-samples.html, 2024.

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

[15] Lisandro Di Meo. Maestro extended with automatic test generation. https://github.com/LisandroDiMeo/maestro, 2024.

[16] mobile.dev. Maestro. https://maestro.mobile.dev, 2022.

[17] mobile.dev. Maestro at github. https://github.com/mobile-dev-inc/maestro, 2022.

[18] Iván Arcuschin Moreno, Lisandro Di Meo, Michael Auer, Juan Pablo Galeotti, and Gordon Fraser. Brewing up reliability: Espresso test generation for android apps. In *ICST*. IEEE, 2024.

[19] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of ui-based flaky tests, 2021.

[20] Richard L. Sauder. A general test data generator for cobol. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), page 317–323, New York, NY, USA, 1962. Association for Computing Machinery.

[21] Statista. Mobile Market Share. https://web.archive.org/web/20240306150509/https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, 2042.

[22] Wikipedia contributors. F-droid — Wikipedia, the free encyclopedia, 2024.