МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 10.00.00.000 ПЗ

Група ІПм-24-1

Гоголь Василь

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Гоголь Василь Іванович

(прізвище, ім'я, по батькові)

УДК 004.9

(індекс)

МАГІСТЕРСЬКА РОБОТА

Методи та засоби побудови автоматизованих модель-базованих

інтерфейсів генерації тестів для мобільних застосунків

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Гоголь В.І.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Процюк Василь Романович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Гоголю Василю Івановичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Методи та засоби побудови автоматизованих модель-базованих інтерфейсів генерації тестів для мобільних застосунків”
керівник проекту (роботи) Процюк Василь Романович, к.т.н., доцент
затверджені наказом закладу вищої освіти від “ 05 ” листопада 2025 р. № 695/7

2. Строк подання студентом проекту (роботи) 15 грудня 2025 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних технологій генерації тестів

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Дослідження концепцій побудови модель-базованих інтерфейсів генерації тестів
2. Інструменти та модель побудови модель-базованих інтерфейсів генерації тестів
3. Концептуальна реалізація розширеного графа потоку управління
4. Імплементація методів генерації послідовностей вхідних даних тестування застосунків

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Спрощений алгоритм представлення життєвого циклу активності (рис. 1.1)
2. Екрани, що відображаються застосунком Activity Lifecycle під час лістингу 1.3 (рис. 1.2)
3. Архітектура Appium (рис. 1.3)
4. Вікно «Запис тесту» із записаними взаємодіями інтерфейсу користувача (рис. 2.1)
5. Підмножина eCFG, що представляє Android-застосунок (рис. 2.2)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2025 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2025	виконано
2	Дослідження концепцій побудови модель-базованих інтерфейсів генерації тестів	29.09.2025	виконано
3	Інструменти та модель побудови модель-базованих інтерфейсів генерації тестів	15.10.2025	виконано
4	Концептуальна реалізація розширеного графа потоку управління	08.11.2025	виконано
5	Імплементація методів генерації послідовностей вхідних даних тестування застосунків	20.11.2025	виконано
6	Реалізація функціональності запропонованої інформаційної технології	01.12.2025	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2025	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 80 с., 8 рис., 10 табл., 30 джерел.

Тема: Методи та засоби побудови автоматизованих модель-базованих інтерфейсів генерації тестів для мобільних застосунків

Метою магістерської роботи є розробка та обґрунтування методів і засобів побудови автоматизованих модель-базованих інтерфейсів генерації тестів для мобільних застосунків.

Об'єкт дослідження - процес автоматизованого тестування мобільних застосунків.

Предмет дослідження - методи та інструменти побудови модель-базованих інтерфейсів для генерації тестових сценаріїв у мобільних застосунках.

Результати дослідження

В роботі розроблено алгоритми генерації тестових послідовностей, що базуються на інтегрованому використанні статичного та динамічного аналізу, що підвищує адекватність і повноту покриття тестування.

Висновок

Досліджено та вдосконалено фреймворк, який забезпечує автоматизовану генерацію вхідних даних і демонструє підвищену ефективність у виявленні дефектів порівняно з традиційними методами.

АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, МОДЕЛЬ-БАЗОВАНЕ ТЕСТУВАННЯ, СТАТИЧНИЙ АНАЛІЗ, ДИНАМІЧНИЙ АНАЛІЗ, ГЕНЕРАЦІЯ ТЕСТІВ, ГРАФ УПРАВЛІННЯ.

ABSTRACT

Master Thesis: 80 pp., 8 fig., 10 tab., 30 sources.

Topic: Methods and tools for building automated model-based interfaces for test generation for mobile applications.

The method of the master's thesis is the development and justification of methods and tools for building automated model-based interfaces for test generation for mobile applications.

The object of the study is the process of automated testing of mobile applications.

The subject of the study is methods and tools for building model-based interfaces for generating test scenarios in mobile applications.

Research results

The work developed algorithms for generating test items, which are based on the integrated use of static and dynamic analysis, which performs adequacy and fully covers testing.

Conclusion

A framework was studied and improved, which provides automated generation of input data and demonstrates increased efficiency in eliminating defects using traditional methods.

TEST AUTOMATION, SOFTWARE TESTING, MODEL-BASED TESTING, STATIC ANALYSIS, DYNAMIC ANALYSIS, TEST GENERATION, CONTROL GRAPH.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

A3E - Automatic Android App Explorer

AUT - Application Under Test

DAG - Directed Acyclic Graph

RQ - Research Question

JaCoCo - Java Code Coverage

FSM - Finite State Machine

EFG - Event Flow Graph

DFS - Depth-First Search

Stoat - STOchastic Model App Tester

GML - Graph Modelling Language

IR - Intermediate Representation

ВСТУП

Актуальність теми.

У сучасному світі мобільні технології стали одним із ключових чинників розвитку інформаційного суспільства. Щоденне використання мобільних застосунків для комунікації, фінансових операцій, електронної комерції, навчання та розваг зумовлює підвищені вимоги до їх надійності, стабільності та безпеки. Помилки у роботі мобільного програмного забезпечення можуть призводити не лише до фінансових збитків, але й до репутаційних втрат компаній, а також до загрози безпеці персональних даних користувачів.

Водночас зростання складності мобільних систем і різноманіття пристроїв робить процес ручного тестування надзвичайно трудомістким і витратним. Це формує потребу у впровадженні автоматизованих методів тестування, здатних забезпечити масштабованість, швидкість та об’єктивність перевірки програмного забезпечення. Серед таких підходів особливе місце займають модель-базовані методи, які дозволяють формалізувати поведінку програм та автоматично генерувати тестові сценарії.

У даній магістерській роботі проведено комплексне дослідження методів і засобів побудови модель-базованих інтерфейсів генерації тестів для мобільних застосунків, зокрема для платформи Android, а також запропоновано власний підхід, спрямований на підвищення ефективності процесу тестування.

Актуальність роботи зумовлена стрімким розвитком мобільних технологій та зростанням вимог до якості програмного забезпечення. У сучасних умовах мобільні застосунки виступають інструментами для доступу до банківських сервісів, електронної медицини, державних послуг та освітніх платформ, що робить їх критично важливими для повсякденного життя. Будь-які збої у роботі таких систем можуть призвести до значних фінансових

втрат, негативного впливу на репутацію розробників і навіть загрожувати конфіденційності даних користувачів.

Ручне тестування, попри свою значимість, є обмеженим з огляду на витрати часу та ресурсів, тому впровадження автоматизованих систем є необхідністю. Особливої уваги потребують методи, що забезпечують адекватність і повноту покриття тестів, серед яких модель-базовані підходи вважаються найперспективнішими. Це визначає наукову й практичну значущість дослідження, результати якого спрямовані на створення універсальних інструментів для автоматизованої генерації тестових сценаріїв.

Метою магістерської роботи є розробка та обґрунтування методів і засобів побудови автоматизованих модель-базованих інтерфейсів генерації тестів для мобільних застосунків.

Об'єкт дослідження - процес автоматизованого тестування мобільних застосунків.

Предмет дослідження - методи та інструменти побудови модель-базованих інтерфейсів для генерації тестових сценаріїв у мобільних застосунках.

Завдання дослідження

1. Провести аналіз сучасних методів і підходів до автоматизації тестування мобільних застосунків.

2. Визначити ключові проблеми й виклики у побудові інтерфейсів генерації тестів для Android-застосунків.

3. Дослідити можливості статичного та динамічного аналізу програмного забезпечення для побудови моделей поведінки.

4. Оцінити ефективність запропонованих рішень у порівнянні з традиційними методами тестування.

Методи дослідження

- методи аналізу та синтезу для систематизації підходів до тестування мобільних застосунків;

- формалізація програмних моделей за допомогою графових представлень;

- статичний і динамічний аналіз Android-застосунків;

- алгоритмічне моделювання для побудови генераторів тестових сценаріїв

Наукова новизна отриманих результатів

Запропоновано модель розширеного графа потоку управління для мобільних застосунків, яка враховує специфіку життєвого циклу Android-активностей та взаємодію елементів користувацького інтерфейсу.

Практичне застосування результатів

Отримані результати можуть бути використані у процесі розробки та тестування мобільних застосунків для підвищення їхньої надійності та якості. Запропоновані методи та інструменти можуть інтегруватися у промислові системи автоматизованого тестування, що зменшить витрати часу й ресурсів на забезпечення якості програмного забезпечення.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 80 сторінок, і містить 8 рисунків, 10 таблиць, список використаних джерел із 30 найменувань.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ КОНЦЕПЦІЙ ПОБУДОВИ АВТОМАТИЗОВАНИХ МОДЕЛЬ-БАЗОВАНИХ ІНТЕРФЕЙСІВ ГЕНЕРАЦІЇ ТЕСТІВ ДЛЯ МОБІЛЬНИХ ЗАСТОСУНКІВ

1.1. Роль та значення тестування мобільних застосунків в сучасних інформаційних системах

Мобільні застосунки набули статусу невіддільного компонента сучасного суспільства. З огляду на масштабне використання смартфонів більшістю населення, де операційна система Android посідає домінуючу позицію серед мобільних платформ, зростання комерційного обігу та обсягів використання мобільних застосунків зумовлює критичну необхідність забезпечення їхньої якості та цілісності. Дефекти програмного забезпечення, які можуть проявлятися у формі несподіваної втрати даних або неузгодженості користувацького досвіду, безпосередньо впливають на лояльність користувачів і, як наслідок, на фінансові показники розробників, зважаючи на високу конкуренцію на ринку.

1.1.1. Проблематика виявлення дефектів користувацького інтерфейсу

Тестування програмного забезпечення є ключовим механізмом для ідентифікації недоліків у програмній системі до її комерційного випуску. Серед численних аспектів мобільних застосунків, що вимагають ретельної валідації, особливої уваги заслуговують дефекти, які стосуються або виникають внаслідок некоректної обробки взаємодії користувача з інтерфейсом користувача (UI). Оскільки ці дефекти є легко помітними кінцевому споживачеві, їхнє своєчасне виявлення є імперативним для утримання користувачів та забезпечення їхньої задоволеності.

Інтеграційне тестування, що використовує імітовані вхідні дані на рівні UI, є усталеним підходом для виявлення згаданих дефектів. Проте, динамічний розвиток мобільної індустрії та висока трудомісткість ручних

методів тестування стимулювали активні дослідження в галузі автоматизованої генерації тестових вхідних даних протягом останнього десятиліття.

Результати досліджень у сфері автоматизованої генерації тестових вхідних даних демонструють перспективність різних методологічних підходів. Випадкова генерація вхідних даних (fuzzing) є найпростішою в реалізації та підтримці, але її ефективність часто є недостатньою. На противагу цьому, систематичні та пошукові (евристичні) підходи забезпечують вищу ефективність тестів, але вимагають значних часових витрат на виконання. Модельно-орієнтований підхід вимагає додаткових початкових витрат на створення моделі застосунку, що тестується (AUT), проте це компенсується скороченням часу генерації тестів.

1.1.2. Розробка фреймворку для тестування Android-застосунків

Дане дослідження присвячене моделюванню Android-застосунків та автоматичній генерації тестів інтерфейсу користувача на основі розроблених моделей.

1. Концептуальне Обґрунтування.

Спочатку представлено концептуальне доведення можливості обходу обмежень існуючих моделей, що демонструє потенційні переваги використання більш комплексної моделі Android-застосунку для забезпечення підвищеної ефективності та результативності тестових вхідних даних.

2. Аналіз підходів до моделювання.

Проведено дослідження обмежень як статичних, так і динамічних підходів до моделювання Android-застосунків.

3. Представлення фреймворку.

На завершення, презентується фреймворк для модельно-орієнтованої генерації тестових вхідних даних. Він використовує детальну модель AUT,

що дозволяє генерувати тести, які досягають вищого покриття при скороченому часі генерації.

1.2. Виклики та мотивація автоматизації тестування мобільних застосунків

Сфера розробки мобільних застосунків характеризується високою динамічністю та прискореними циклами релізів. У контексті зростання темпів розробки критично важливо забезпечити адекватну спроможність системи тестування. Незважаючи на це, функціональне тестування, зокрема тестування інтерфейсу користувача (UI), здебільшого залишається ручним процесом [6-7]. Така трудомісткість і низька швидкість можуть призводити до випуску недостатньо валідованих продуктів, що підвищує ймовірність виникнення помилок, які негативно впливають на доступність сервісу та утримання користувачів [11].

Автоматизоване тестування, зокрема автоматизована генерація тестових вхідних даних, є ефективним засобом для збільшення глибини та обсягу тестового покриття порівняно з ручними методами, забезпечуючи формування більш надійних та стійких тестових наборів.

1.2.1. Економічне та соціальне значення надійності мобільних застосунків

Мобільні застосунки проникли в усі сфери сучасного життя. Станом на 2024 рік глобальний рівень проникнення смартфонів становив приблизно 78% населення [12]. Ця масштабна користувацька база супроводжується величезною кількістю доступних застосунків та послуг, причому лише в Google Play Store обсяг доходів оцінюється приблизно в 10.4 мільярда доларів США. Операційна система Android протягом останнього десятиліття стала домінуючою з часткою ринку 70% у четвертому кварталі 2024 року, а Google Play Store є найбільшим репозиторієм мобільних застосунків [13].

З огляду на експоненційне зростання популярності та економічне значення, забезпечення адекватності тестування мобільних застосунків для гарантування їхньої надійності є першочерговим завданням. Наприклад, несподівана втрата даних або неузгоджений користувацький досвід можуть спровокувати негативні відгуки та втрату користувачів і, відповідно, доходів. Більше того, дефекти у критичних категоріях застосунків (наприклад, медичні застосунки для моніторингу лікування або планування прийомів) можуть мати серйозні наслідки для користувачів.

1.2.2. Виклики тестування UI та методології генерації тестів

Інтерфейс користувача (UI) є ключовою точкою взаємодії між користувачем та функціоналом застосунку, що обумовлює необхідність його надійного тестування. Сучасні мобільні пристрої є орієнтованими на події, сенсорними та інтегрують навколишнє апаратне забезпечення (наприклад, гіроскопи). Ці особливості створюють унікальні виклики для мобільного тестування, які унеможливлюють пряме застосування традиційних методологій. Згадані виклики, прискорений темп мобільної індустрії та висока трудомісткість поточних методів тестування перетворили автоматизовану генерацію тестових вхідних даних на пріоритетний напрямок досліджень [6-10].

Автоматизована генерація тестових вхідних даних демонструє обнадійливі результати з різними підходами:

- Випадкова Генерація (Fuzzing) - найпростіший і найпоширеніший метод, але неефективний, що призводить до великих та надлишкових тестових наборів.

- Систематичні / навчання Моделі - збирають дані про UI під час виконання для керування рішеннями щодо вхідних даних. Хоча цей підхід генерує інтелектуальніші вхідні дані, його здатність до пошуку часто є обмеженою.

- Пошукові техніки - використовують еволюційні алгоритми для автоматизації пошуку тестових даних з метою максимізації досягнення тестових цілей. Це високоуспішний підхід, але він вимагає надмірного часу виконання.

- Модельно-орієнтовані методи - формальний та успішний підхід, який використовує моделі програмних систем для виведення тестів. Хоча існує додаткова початкова витрата на моделювання AUT, це призводить до суттєво швидшої генерації тестів та формування більш ретельних та ефективних тестів.

1.2.3. Підхід детального моделювання потоку управління застосунку

Попередні моделі генерації тестів переважно фокусувалися на структурі інтерфейсу застосунку (тобто на елементах, з якими користувач може взаємодіяти). Ці моделі зазвичай створюються шляхом динамічного обходу UI. Хоча зосередження на UI є логічним, внутрішня кодова структура застосунку може надавати критичні деталі, недоступні через сам інтерфейс. Ці деталі можуть керувати генерацією тестових вхідних даних на основі моделі для досягнення кращого тестового покриття AUT. Наприклад, для активації прихованих функцій може знадобитися специфічна послідовність взаємодій (як-от натискання на елемент "Версія збірки" сім разів у налаштуваннях Android для активації режиму розробника).

У даній роботі представлено модельно-орієнтований фреймворк для генерації тестових вхідних даних для Android. Він покладається на детальну модель потоку управління застосунком, що тестується (AUT).

Для досягнення високого рівня деталізації пропонується використовувати розширений граф потоку управління (eCFG), який охоплює:

- Низькорівневі інструкції та внутрішньопроцедурні виклики.

- Високорівневі елементи інтерфейсу та відповідні методи зворотного виклику.

Така інтегративна модель не лише забезпечує високий рівень деталізації для генерації тестових вхідних даних на основі моделі, але й створює основу для потенційних майбутніх досліджень. Дослідження включає аналіз методів та вимог для генерації комплексної моделі застосунку та порівняння фреймворку із сучасними підходами в автоматизованій генерації вхідних даних. Для концептуальної перевірки моделі було реалізовано фреймворк із ручним тегуванням вихідного коду для генерації моделі на невеликій вибірці Android-застосунків.

1.3. Проблематика дослідження магістерської роботи

Загальна проблема цього дослідження полягає у підвищенні ефективності та продуктивності автоматизованої генерації тестових вхідних даних на основі моделі для мобільних застосунків шляхом використання більш комплексної моделі застосунку, яка інтегрує як аспекти інтерфейсу користувача (UI), так і внутрішню логіку програми.

1.3.1. Формулювання проблеми

Автоматизована генерація вхідних даних, особливо модельно-орієнтована генерація, продемонструвала значний потенціал. Однак існуючі моделі, які використовуються в минулих дослідженнях, переважно фокусувалися на структурі UI. Хоча такий підхід є логічним для тестування інтерфейсу, решта кодової бази застосунку містить критичні структурні знання, які можуть керувати процесом генерації тестів для більш глибокого дослідження та тестування застосунку (AUT). Наприклад, для активації прихованих функцій або елементів UI може бути необхідна специфічна послідовність або повторення взаємодій (наприклад, багаторазове натискання).

Таким чином, мета дослідження полягає у розробці детальної моделі застосунку, що включає всі його компоненти — від низькорівневих

інструкцій та внутрішньопроцедурних викликів до високорівневих елементів UI та методів зворотного виклику. На основі цієї моделі буде автоматично генеруватися тестовий набір, який демонструє вищу ефективність порівняно із сучасними методами автоматизованої генерації вхідних даних.

У рамках сформульованої проблеми дослідження розглядаються наступні питання:

1. Ефективність використання попередніх структурних знань.

Яким чином попередні структурні знання про AUT можуть бути використані для генерації більш ефективних вхідних даних із одночасним зменшенням надлишковості? З огляду на загальну структуру застосунку (як внутрішній код, так і зовнішній інтерфейс), чи можна очікувати збільшення покриття моделі та ефективності при зниженні надлишковості, порівняно, наприклад, із менш ефективною випадковою генерацією?

2. Побудова точної моделі та обмеження статичного аналізу.

Як побудувати точну модель мобільного застосунку та пом'якшити обмеження статичного аналізу в цьому контексті? Оскільки подієво-орієнтована природа Android та її тісна інтеграція з фреймворком ускладнюють традиційний статичний аналіз, чи може інтеграція простого динамічного аналізу, заснованого на систематичному дослідженні, пом'якшити ці обмеження та надати достатню деталізацію для адекватного моделювання?

3. Порівняння покриття з сучасним рівнем.

Чи призводить модельно-орієнтована генерація тестових вхідних даних, що використовує детальну модель потоку управління AUT, до збільшення покриття застосунку порівняно із сучасними підходами, які переважно фокусуються на моделюванні структури UI (наприклад, через FSM або EFG)? Чи може детальна модель, що об'єднує високорівневу структуру інтерфейсу та низькорівневу структуру інструкцій потоку управління, збільшити досягнуте покриття?

1.3.2. Основні етапи дослідження

Дослідження реалізовано у чотири послідовні етапи, кожен з яких оцінює вимоги до автоматизованої генерації тестів та пропонує компоненти рішення для тестування Android:

1. Концептуальне визначення та перевірка моделі.

Визначення концептуальної моделі потоку управління Android-застосунків (яка охоплює компоненти від низькорівневих інструкцій до елементів UI та їхніх методів зворотного виклику) та реалізація концептуального фреймворку з ручним тегуванням для її перевірки на невеликій вибірці застосунків.

2. Оцінка переваг попереднього досвіду.

Дослідження переваг попередніх знань про застосунок (на основі концептуальної моделі) у генерації потенційних тестових послідовностей. Проведено порівняння обходу графа (демонстрація потенційних послідовностей) із випадковою генерацією вхідних даних (виконаною Monkey) для оцінки зменшення надлишковості та збільшення покриття моделі.

3. Розробка фреймворку.

Представлення фреймворку для автоматичної генерації комплексної моделі Android-застосунків. Було досліджено ефективність комбінованого використання статичного та динамічного аналізу (динамічний аналіз виконано через дослідні тести) для пом'якшення обмежень статичного аналізу в Android-середовищі, порівнюючи його з випадковим дослідженням (Monkey).

4. Реалізація та оцінка.

Реалізація модельно-орієнтованого фреймворку для Android, який використовує модель, створену DroidGr. Проведено оцінку покриття та ефективності шляхом його порівняння із сучасним рівнем автоматизованої генерації вхідних даних, що включає три техніки:

- Випадковий генератор (Monkey).

- Пошуковий підхід (STGFA-SMOG).

- Модельно-орієнтований підхід на основі FSM (Stoat).

1.4. Тестування програмного забезпечення та критерії тестування

Цей розділ представляє базові концепції та визначення, необхідні для розуміння подальшого матеріалу. Спочатку вводяться ключові поняття тестування програмного забезпечення, критеріїв тестування та генерації вхідних даних у контексті мобільної платформи Android. Далі описані виклики, які середовище Android ставить перед традиційними методами статичного аналізу.

Тестування програмного забезпечення є усталеною дисципліною, спрямованою на оцінку та забезпечення надійності програмних систем. Його головна мета — максимізація виявлення дефектів, особливо критичних, та валідація функціональності системи.

Оскільки вичерпне тестування (застосування всіх можливих сценаріїв вхідних даних) є практично неможливим навіть для невеликих застосунків, тестування прагне бути максимально вичерпним для досягнення задовільного рівня стабільності.

1.4.1. Адекватність тестування та критерії покриття

Концепція адекватності тестування вводиться для визначення практичного та застосовного наближення до надійності системи. Це досягається за допомогою критеріїв тестування, які слугують проксі-вимірюванням якості тестування.

Критерії тестування часто використовуються для:

- Оцінки надійності колекції тестів (тестового набору).

- Встановлення правила зупинки (визначення, коли тестування вважається достатнім).

Покриття коду є популярним критерієм тестування, що відображає частку виконаного коду під час тестування, хоча й не гарантує виявлення дефектів. Існують різні типи покриття:

- Покриття рядків (Line Coverage): Кількість виконаних рядків коду.

- Покриття гілок (Branch Coverage): Кількість виконаних логічних умов (гілок).

- Покриття методів (Method Coverage): Кількість виконаних методів.

Використовуючи покриття коду як критерій, адекватність легко встановлюється шляхом визначення заздалегідь визначеного рівня покриття як правила зупинки (наприклад, 80% виконання рядків).

1.4.2. Форми тестування

Тестування реалізується шляхом запуску застосунку, що тестується (AUT), у контрольованому середовищі з використанням попередньо визначених вхідних даних та подальшої валідації отриманого результату проти тестового оракула. Різні форми тестування фокусуються на різних аспектах системи:

- Модульне тестування: Специфічне для функціональних модулів, зосереджене на функціональних дефектах.

- Інтеграційне/UI-тестування: Розглядає взаємодію між компонентами або роботу системи в цілому.

- Регресійне тестування: Забезпечує збереження існуючої функціональності після оновлень.

Тестування передбачає виконання застосунку, що тестується (AUT), у контрольованому середовищі з заздалегідь визначеними умовами. Кожен тест активує AUT або взаємодіє з ним, використовуючи предоптовані вхідні дані, та здійснює верифікацію отриманого результату шляхом порівняння з заздалегідь визначеними тестовими оракулами.

Наприклад, лістинг 1.1 ілюструє приклад Java-класу Dog, що містить один метод, а лістинг 1.2 демонструє приклад тестового класу JUnit, який

містить один модульний тест. У цьому тесті відбувається ініціалізація нового екземпляра класу Dog, після чого вихідне значення методу speak() зіставляється з його очікуваним значенням.

Лістинг 1.1. Клас Dog (Вихідний Код Java)

```
public class Dog {

    public void speak() {
        System.out.println("Woof");
    }
}
```

Лістинг 1.2. Клас DogTest (JUnit Тест)

```
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class DogTest {

    private Dog dog;
    private final PrintStream stdOut = System.out;
    private final ByteArrayOutputStream capture = new ByteArrayOutputStream();

    @BeforeEach
    public void setUp() {
        System.setOut(new PrintStream(capture));
        this.dog = new Dog();
    }

    @Test
    public void testSpeak() {
        this.dog.speak();
        assertEquals("Woof", capture.toString().trim());
    }

    @AfterEach
    public void tearDown() {
        System.setOut(stdOut);
    }
}
```

Лістинг 1.2 демонструє відповідний тестовий клас JUnit під назвою DogTest, призначений для валідації функціональності класу Dog.

1. Налаштування Тестового Середовища (@BeforeEach):

- Метод setUp() використовується для ініціалізації тестового середовища перед кожним тестом.

- Він перехоплює стандартний потік виведення (System.out), перенаправляючи його у потік capture (ByteArrayOutputStream), що дозволяє фіксувати виведення методу speak().

- Створюється новий екземпляр класу Dog.

2. Виконання Тесту (@Test):

- Метод testSpeak() виконує метод speak() ініціалізованого екземпляра dog.

- Використовується твердження (assertion) assertEquals("Woof", capture.toString().trim()), яке порівнює фактичне виведення, зафіксоване у потоці capture (після перетворення на рядок та видалення пробілів), з очікуваним значенням "Woof".

3. Очищення Середовища (@AfterEach):

Метод tearDown() забезпечує відновлення початкового стану системи шляхом повернення стандартного потоку виведення (System.out) до його оригінального значення (stdOut).

1.5. Платформа android та її архітектурні особливості

Android — це операційна система з відкритим вихідним кодом для мобільних пристроїв, заснована у 2007 році. Хоча вона була спочатку розроблена на Java, ця мова програмування залишається домінуючою для розробки, незважаючи на появу Kotlin.

Архітектура Android-застосунків суттєво відрізняється від стандартних Java-застосунків (відсутність традиційного методу main). Застосунок складається з автономних компонентів, які разом формують функціональність, але функціонують незалежно і мають власні точки входу. Основні типи компонентів включають:

- Активності (Activities) - забезпечують візуальний інтерфейс та взаємодію з користувачем.

- Фрагменти (Fragments) - модульні секції UI.

- Сервіси (Services) - виконують тривалі операції у фоновому режимі.

- Мовники та Отримувачі вмісту (Broadcast Receivers and Content Providers) - обробляють системні або застосункові події та управляють доступом до даних.

Виклик цих компонентів координується фреймворком Android, який контролює та управляє станом усіх компонентів. Наприклад, для запуску Активності використовується об'єкт Intent, який запитує фреймворк на створення або активацію компонента, а не пряме створення екземпляра.

1.5.1. Життєвий цикл активності

Життєвий цикл активності (як показано на рисунку 1.1) дозволяє розробникам визначати логіку на різних етапах створення, функціонування та знищення компонента. Фреймворк Android динамічно обирає відповідну точку входу та шлях виконання (наприклад, onCreate(), onResume(), onRestart()) на основі поточного стану системи, системних подій та взаємодій користувача.

Ця подієво-орієнтована та компонентно-залежна природа платформи призводить до двох ключових особливостей:

- Розмитість потоку управління - потік управління є неявним і залежним від часу виконання, що ускладнює його визначення традиційними методами статичного аналізу.

- Повторне використання функцій - автономні компоненти дозволяють здійснювати міжзастосункову комунікацію через Intents (наприклад, запит на відображення адреси на сторонній карті), що є важливим фактором при тестуванні.

- Виклики для традиційного аналізу - вказані архітектурні особливості (відсутність єдиної точки входу, подієво-орієнтований життєвий цикл)

створюють значні труднощі для традиційних інструментів статичного аналізу, які не можуть адекватно відстежувати роз'єднані потоки управління Android-застосунків.

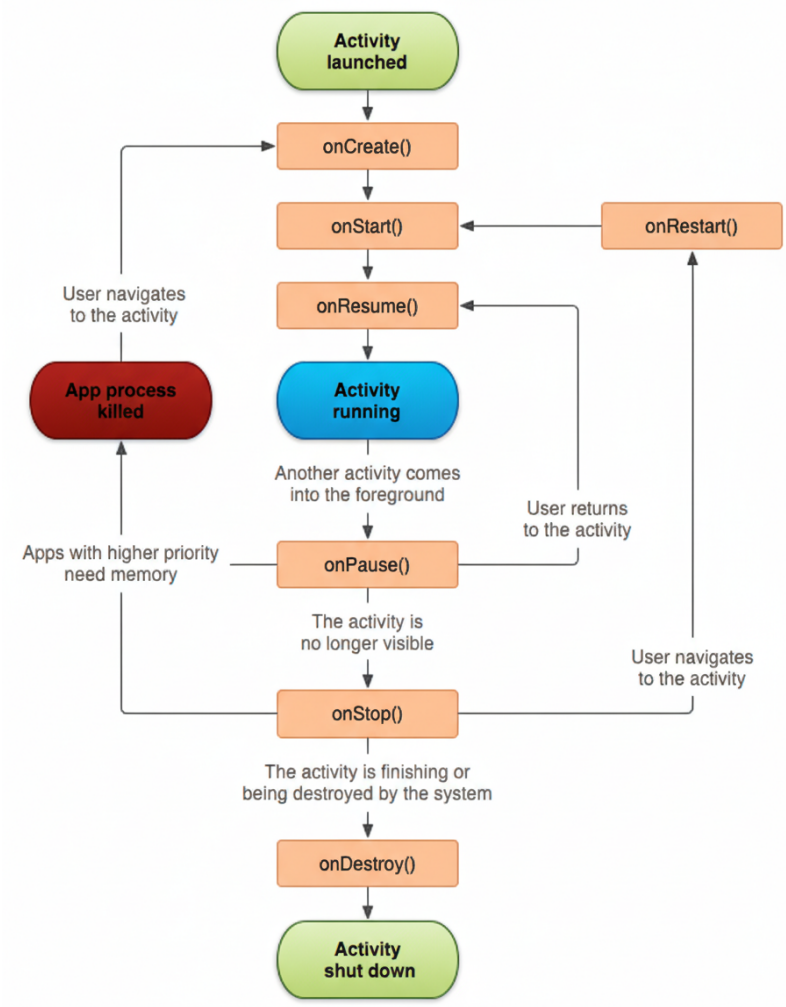

Рис. 1.1. Спрощений алгоритм представлення життєвого циклу активності

1.5.2. Тестування програмного забезпечення для Android

Платформа Android створює значні виклики для тестування, що зумовлено її подієво-орієнтованою архітектурою та високою інтеграцією застосунків із фреймворком. На відміну від стандартних застосунків, які мають явні точки входу та визначені шляхи виконання, компоненти Android можуть бути інстанційовані та викликані непередбачуваним чином. Ця структура забезпечує інтерактивність платформи, однак порядок виконання

значною мірою контролюється фреймворком Android (наприклад, послідовність виклику методів життєвого циклу активності).

Незважаючи на чітку структуру життєвого циклу активності (рисунок 1.1), його фактичне виконання визначається рішеннями, прийнятими під час виконання, які залежать від системного стану та чинників навколишнього середовища. Наприклад, потреба фреймворку у звільненні системної пам'яті може призвести до знищення існуючої активності. У такому випадку, замість очікуваного onRestart() при поверненні користувача може бути викликаний метод onCreate(). Хоча модульне тестування ефективно для перевірки коду окремих методів, міжпроцедурна поведінка та поведінка зворотного виклику інтерфейсу Android-застосунків залишаються непередбачуваними та складними для верифікації.

1.5.3. Автоматизація вхідних даних та середовище тестування

Поширеним методом тестування Android-застосунків є використання тестового API для автоматизації вхідних даних. Це передбачає створення сценарію, який симулює взаємодію користувача з AUT. Наприклад, використання такого інструменту, як Appium, дозволяє програмно перевірити, чи призводить натискання кнопки ('Start B' на рисунку 1.2a) до запуску очікуваного компонента ('Activity B' на рисунку 1.2b), як показано у лістингу 1.3. Хоча ця взаємодія ефективна, ручне створення та підтримка вичерпного та надійного тестового набору цим методом є недоцільним через його трудомісткість.

Практика розробки та тестування передбачає виконання коду на машині, здатній його виконувати. Для Android-застосунків це вимагає використання емулятора або віртуального пристрою на стандартній робочій станції розробника, оскільки розробка безпосередньо на мобільному пристрої є непрактичною. Емулятори не лише забезпечують єдине середовище для розробки та тестування, але й дозволяють проводити валідацію на різноманітних конфігураціях апаратного забезпечення, версіях ОС та

налаштуваннях середовища. Крім того, час тестування може бути скорочений завдяки використанню високопродуктивних серверів для запуску емуляторів.

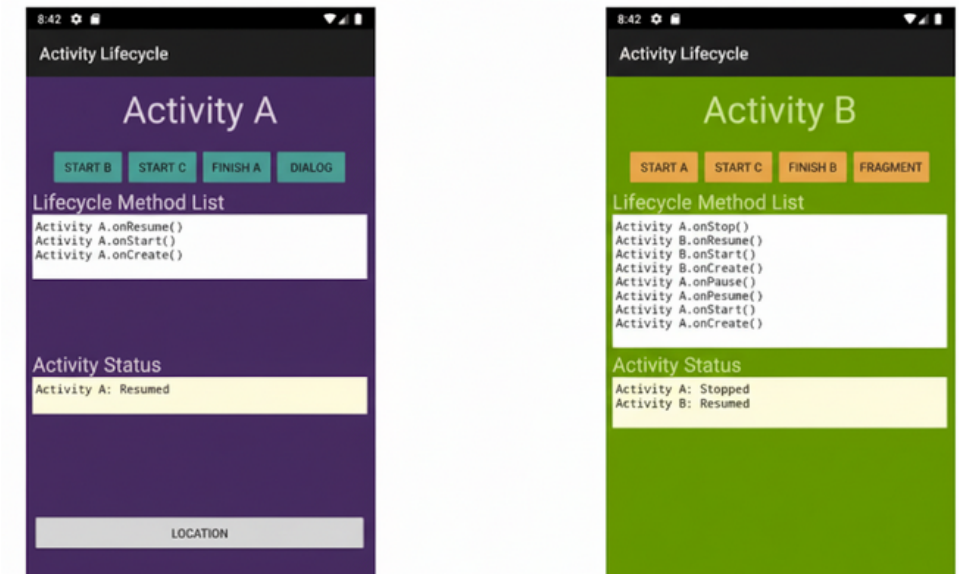

Рис. 1.2. Екрани, що відображаються застосунком Activity Lifecycle під час лістингу 1.3

Лістинг 1.3. Тест Appium на запуск активності (testLaunchActivityB())

```
protected void testLaunchActivityB() {

    String id;

    id = "com.example.android.lifecycle:id/btn_start_b";
    WebElement startB = driver.findElementById(id);
    startB.click();

    id = "com.example.android.lifecycle:id/status_view_all_b";
    WebElement status = driver.findElementById(id);
    String text = status.getAttribute("text");

    String expected;
    expected = "Activity A: Stopped\nActivity B: Resumed\n";
    assertEquals(expected, text);
}
```

1.5.4. Генерація тестових вхідних даних

Швидкий темп розвитку мобільних технологій та ручний характер інтеграційного тестування ускладнюють створення тестового набору, який забезпечує адекватний рівень впевненості. Автоматизована генерація тестів є

механізмом для подолання цієї проблеми, оскільки вона усуває ручну працю та підвищує ефективність тестового набору.

Галузь автоматизованої генерації вхідних даних для мобільного програмного забезпечення активно розвивається і поділяється на кілька ключових підходів.

Таблиця 1.1.

Підходи генерації вхідних даних

Підхід	**Опис та Особливості**	**Ефективність та Обмеження**
Випадкова Генерація (Random)	Випадкове застосування типів вхідних даних у випадкових місцях UI.	Широко використовується та легкий у підтримці; неефективний через високу надлишковість.
Систематична Генерація (Learner-Based)	Збір даних про UI під час виконання для інформування рішень щодо вхідних даних.	Генерує "розумніші" вхідні дані; обмежена здатність до дослідження.
Генерація на Основі Пошуку (Search-Based)	Використання еволюційних алгоритмів для оптимізації знаходження тестових даних і максимізації тестових цілей.	Високоуспішний підхід; вимагає надмірного часу виконання.
Генерація на Основі Моделі (Model-Based)	Легкий формальний метод, який використовує моделі програмних систем для виведення тестів.	Успішний підхід; вимагає додаткових початкових витрат на моделювання AUT, але призводить до значно швидшої генерації та більш ретельних і ефективних тестів.

Основною перевагою автоматизованої генерації вхідних даних є повна автоматизація процесу, що усуває проблеми ручної праці та підтримки, характерні для тестових API. Це дозволяє досліджувати сценарії тестування, які могли бути не передбачені командою розробників, забезпечуючи вищу толерантність до непередбачених взаємодій. Незважаючи на успіхи в дослідженнях, промислове впровадження цих підходів є обмеженим через такі фактори, як низька довіра до автоматично згенерованих тестів та складність їх розуміння розробниками.

1.5.5. Специфікація тестування та критерії адекватності

Відповідно до класифікації, тестування програмного забезпечення набуває різних форм залежно від об’єкта тестування (AUT), його типу та системного середовища. У межах цього дисертаційного дослідження фокус зосереджено на тестуванні вхідних даних інтерфейсу користувача (UI) для вибірки мобільних застосунків Android, розроблених на мові програмування Java.

У контексті даної роботи, тест формально визначається як впорядкована послідовність взаємодій (таких як натискання, свайпи, введення даних тощо) із AUT. Метою цієї послідовності є моніторинг поведінки AUT для забезпечення відповідності очікуваному результату, зокрема, відсутності збою застосунку.

Як було вже зазначено, покриття є ключовим критерієм адекватності тестування, що використовується для оцінки якості згенерованих тестів та встановлення правила зупинки для процесу тестування. Традиційно покриття стосується кількості використаного коду (наприклад, покриття рядків чи гілок). Однак, у цьому дослідженні оцінюване покриття розширюється для включення як традиційного покриття коду, так і покриття інтерфейсу користувача, що відображає ступінь дослідження елементів UI тестами.

Згідно з класифікацією методів генерації тестів (випадковий, систематичний, пошуковий та на основі моделі), це дослідження зосереджується на оцінці та вдосконаленні методів генерації тестів на основі моделі.

Інструмент генерації тестів (ІГТ) визначається як функціональний модуль або фреймворк, який приймає на вхід мобільний застосунок та його графову модель, і на основі цих даних здатен автоматично створювати тестовий набір, призначений для верифікації функціональності застосунку.

Формальні визначення термінів "тест", "тестовий набір", "покриття інтерфейсу користувача" та "інструмент генерації тестів", що використовують концептуальний апарат, будуть описані пізніше.

1.6. Методологія статичного аналізу android-застосунків

Статичний аналіз — це процес вивчення структури коду застосунку без його виконання, що забезпечує глибоке розуміння архітектури програми. На відміну від тестування, яке досліджує лише виконані шляхи, статичний аналіз теоретично здатен охопити всі можливі шляхи виконання (за умови включення всього коду та залежностей). Цей метод зазвичай використовується для:

- Оцінки коду та виявлення вразливостей.
- Ідентифікації помилок та відхилень від стандартів кодування.

Хоча статичний аналіз може виконуватися вручну (огляд коду), використання автоматизованих статичних аналізаторів є значно швидшим та ефективнішим.

Незважаючи на те, що Android-застосунки здебільшого розроблені на Java і можуть використовувати її фреймворки, вони стикаються з проблемами сумісності зі статичними аналізаторами, розробленими для стандартної Java. Це пов'язано з тим, що Android-застосунки компілюються у формат Android Package (APK), який використовує байт-код Dalvik замість стандартного байт-коду Java. Для сумісності інструментів аналізу потрібна додаткова трансляція байт-коду Dalvik у байт-код Java або іншу проміжну мову. Крім того, при роботі з вихідним кодом Java, традиційні статичні аналізатори потребують додаткових знань про фреймворк Android та його API для ефективної роботи. Три найбільш цитовані та активно підтримувані інструменти для статичного аналізу Android-застосунків включають Soot, FlowDroid та AndroGuard.

1.6.1. Інструмент Soot

Soot — це потужний фреймворк, який спочатку був призначений для оптимізації Java, але тепер широко використовується для аналізу, інструментування, оптимізації та візуалізації Java- та Android-застосунків.

Його основна функціональність полягає у міжпроцедурному та внутрішньопроцедурному аналізі за допомогою власної проміжної мови (Intermediate Language). Soot приймає на вхід Java-код, байт-код або Android-байт-код і перетворює його на проміжну мову для спрощення аналізу.

Основною проміжною мовою Soot є Jimple — типова трьохадресна мова. Jimple була створена для подолання складності прямого аналізу байт-коду Java: хоча можна побудувати граф потоку управління (CFG) для байт-коду, неявний стек ускладнює аналіз потоку даних. Jimple, завдяки своїй низькій складності, робить аналіз значно простішим, чітко визначаючи кожну ініціалізацію та посилання.

Наприклад, лістинг 1.1 демонструє простий Java-клас з одним методом, а лістинг 1.4 — той самий клас і метод, скомпільований у байт-код Java. Навіть у цьому елементарному прикладі представлений байт-код є важким для інтерпретації та не забезпечує чіткості передачі даних. На противагу цьому, у лістингу 1.5, де показано той самий клас і метод у форматі Jimple, кожна ініціалізація та посилання чітко визначені та відображені в кожному операторі.

Лістинг 1.4. Клас Dog у Байт-коді Java (Декомпіляція)

```
public class Dog {
  public Dog();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void speak();
    Code:
       0: getstatic #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Woof
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Незважаючи на можливість аналізу Android-байт-коду, повний аналіз Android-застосунку в Soot обмежений. Soot не володіє знаннями про життєвий цикл або фреймворк Android. Це унеможливлює аналіз

специфічних для Android методів зворотного виклику життєвого циклу та запобігає створенню графа викликів, оскільки Android-застосунки мають множинні точки входу (методи життєвого циклу та зворотного виклику) замість однієї явної. Тим не менш, Soot і Jimple залишаються найбільш прийнятими інструментами та форматами для статичного аналізу програм на основі Java.

Лістинг 1.5. Клас Dog у Проміжній Мові Jimple (IR)

```
public class Dog extends java.lang.Object {

    public void <init>() {
        Dog r0;
        r0 := @this: Dog;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }

    public void speak() {
        java.io.PrintStream r1;
        Dog r0;
        r0 := @this: Dog;
        r1 = <java.lang.System: java.io.PrintStream out>;
        virtualinvoke r1.<java.io.PrintStream: void println(java.lang.String)>("Wo
        return;
    }
}
```

1.6.2. Інструмент аналізу FlowDroid

FlowDroid є розширенням Soot і першим повністю контекстно-, поле-, об'єктно- та чутливим до потоку інструментом аналізу забруднень (Taint Analysis), який враховує життєвий цикл Android-застосунку.

FlowDroid було розроблено як інструмент виявлення шкідливого програмного забезпечення для аналізу застосунків на наявність шкідливих потоків даних або порушень політики. Його часто використовують для загального статичного аналізу Android-застосунків, оскільки він інтегрує знання про життєвий цикл активності та механізми зворотного виклику Android, дозволяючи включати їх в аналіз Soot. Завдяки тому, що аналіз FlowDroid не є чутливим до шляху, він може генерувати та аналізувати фіктивний головний метод (dummy main method). У цьому методі він

припускає будь-який можливий порядок виклику методів життєвого циклу та зворотного виклику, що дозволяє Soot створити повний граф викликів.

1.6.3. Інструмент AndroGuard

AndroGuard — це інструмент на основі Python, призначений для зворотного інжинірингу Android-застосунків. На відміну від Soot, орієнтованого на Java, і FlowDroid, орієнтованого на аналіз забруднень, AndroGuard може створити більш повний граф викликів для Android-застосунку, оскільки він використовує оновлені знання про Android API.

AndroGuard є важливим доповненням, оскільки деякі функції FlowDroid є застарілими та ненадійними. Наприклад, FlowDroid не має сумісності з новим API AndroidX для Фрагментів (Fragments). Це призводить до того, що граф викликів, згенерований FlowDroid, постійно не містить класів фрагментів. AndroGuard може бути використаний для доповнення FlowDroid і заміни його застарілого аналізу графа викликів.

1.6.4. Інструментування для Android

Інструментування (Instrumentation) — це загальноприйнятий метод моніторингу внутрішньої структури застосунку та відстеження його виконання під час тестування. Воно має такі випадки використання, як ведення журналу, моніторинг продуктивності та, найголовніше, відстеження покриття коду.

Інструментування передбачає додавання операторів до коду застосунку, які виводять дані під час виконання. Це можна зробити:

- На рівні вихідного коду (перед компіляцією) — поширено для налагодження.

- На рівні скомпільованого байт-коду — поширено для відстеження покриття.

У цьому дослідженні акцент зроблено на інструментуванні скомпільованого байт-коду Dalvik з метою відстеження покриття під час виконання тесту.

Хоча багато інструментів інструментування Java (наприклад, JaCoCo, Emma) можуть працювати з байт-кодом Java, вони є інструментами з "білою скринькою" і повинні додаватися як залежності до збірки.

Для роботи безпосередньо з скомпільованим APK-файлом були створені інструменти з "чорною скринькою", такі як Android Code Coverage Tool (ACVTool) та Ella. Інструментування APK може відбуватися двома шляхами:

1. Пряме інструментування байт-коду Dalvik (наприклад, Ella).

2. Інструментування проміжної мови (наприклад, Smali або Jimple). ACVTool декомпілює APK за допомогою Apktool в проміжну мову Smali для інструментування, а потім перекомпільовує його назад в APK.

Soot і, відповідно, FlowDroid також можуть бути використані для інструментування APK шляхом додавання операторів до їх внутрішнього представлення Jimple. У цій роботі FlowDroid використовується для інструментування AUT, а ACVTool — для збору даних про покриття.

Висновки до розділу

У першому розділі роботи було проведено системний аналіз концепцій, підходів та існуючих методологій побудови автоматизованих модель-базованих інтерфейсів генерації тестів для мобільних застосунків, зокрема у контексті платформи Android, яка є домінуючою на сучасному ринку мобільних технологій.

Насамперед встановлено, що тестування мобільних застосунків є критично важливим етапом життєвого циклу програмного забезпечення, оскільки воно безпосередньо впливає на якість користувацького досвіду, рівень надійності програмних рішень та довіру кінцевих користувачі.

Проаналізовано проблематику виявлення дефектів інтерфейсу користувача, яка залишається одним із найскладніших завдань, зважаючи на високу динамічність UI, велику кількість сценаріїв взаємодії та різноманітність пристроїв. Особливу увагу приділено питанням створення фреймворків для тестування Android-застосунків, що забезпечують масштабованість та відтворюваність тестових сценаріїв.

РОЗДІЛ 2. ІНСТРУМЕНТИ ТА МОДЕЛЬ ПОБУДОВИ МОДЕЛЬ-БАЗОВАНИХ ІНТЕРФЕЙСІВ ГЕНЕРАЦІЇ ТЕСТІВ

2.1. Інструменти автоматизації API

Фреймворки та API, зокрема UI-Automator та Appium, а також подібні інструменти, надають розробникам можливість створювати тестові скрипти, які симулюють інтерактивні події через Android-емулятор. Ці інструменти дозволяють проводити тестування через інтерфейс користувача (UI), охоплюючи діапазон від перевірки одиночних взаємодій до складних послідовностей.

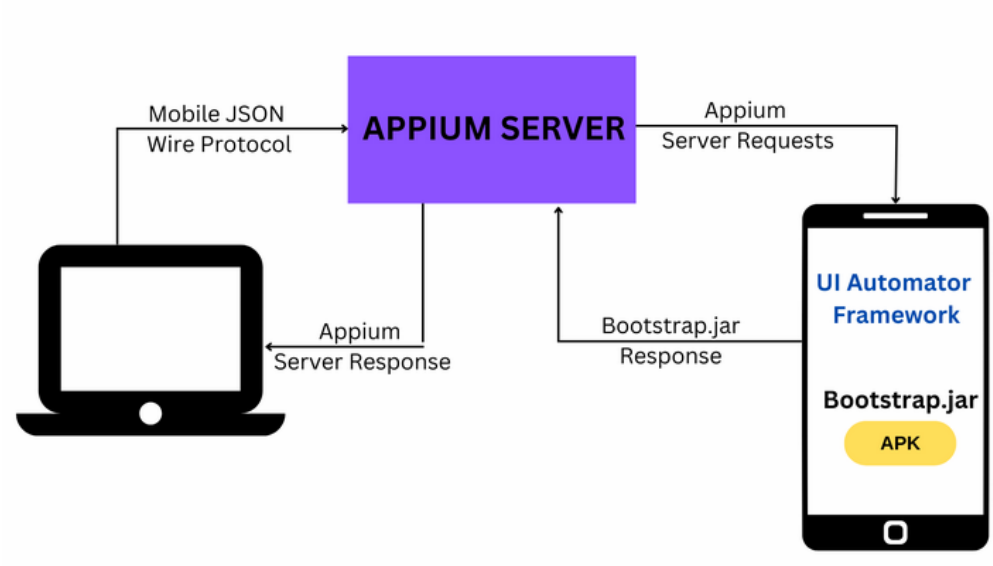

Рис. 2.1. Архітектура Appium

Ключовою перевагою є те, що всі тести виконуються методом взаємодії кінцевого користувача, що забезпечує покриття поведінки методів життєвого циклу активності та методів зворотного виклику вхідних даних. Розробник повинен програмно описати взаємодію з пристроєм, а потім верифікувати отриманий результат, використовуючи такі методи, як:

- Порівняння зображень або атрибутів перегляду.
- Виявлення збоїв (крешів).

- Візуальне спостереження за результатами тесту.

Такий тип тестування є корисним для базової функціональної валідації на етапі розробки завдяки висококонтрольованому тестовому середовищу.

Недоліком є необхідність ручного скриптування, яке потребує значних часових витрат на підтримку відповідно до еволюції AUT. Крім того, через фрагментацію пристроїв Android (різні розміри та щільності екранів) скрипти не гарантують працездатності на всіх пристроях. Наприклад, оскільки розробники можуть визначати кілька макетів UI для різних розмірів екранів, для повного тестування необхідно створювати та підтримувати окремі скрипти для широкого спектра конфігурацій пристроїв.

Інструменти запису та відтворення, такі як Mosaic або Espresso Recorder, пропонують функціональність, аналогічну API автоматизації, але усувають потребу в ручному скриптуванні. Замість написання коду, розробник взаємодіє із застосунком на емуляторі (або пристрої), а ці взаємодії автоматично записуються і можуть бути відтворені за потреби.

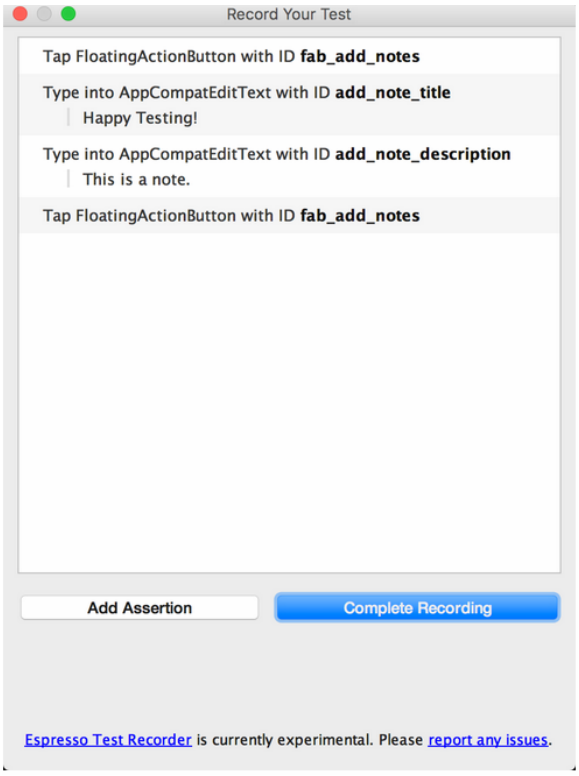

Рис. 2.2. Вікно «Запис тесту» із записаними взаємодіями інтерфейсу користувача

Більшість цих інструментів стикаються з компромісом між точністю часу та деталей записаних подій та репрезентативною силою та переносністю згенерованих скриптів. Деякі інструменти використовують потік подій вводу ядра Linux для досягнення високої точності запису/відтворення, але це знижує переносність.

Хоча інструменти запису/відтворення прискорюють створення тестів, вони не вирішують фундаментальних проблем тестування UI. Записи залишаються залежними від пристрою та конфігурації, на якій вони були створені, вимагаючи створення нових записів для широкого спектра пристроїв та оновлення для нових релізів операційної системи.

2.2. Методи та інструменти автоматизованої генерації вхідних даних у тестуванні мобільних застосунків

Автоматизована генерація вхідних даних є наразі найактивнішою дослідницькою галуззю в тестуванні мобільного програмного забезпечення. Вона розвинулася до підкатегорій, класифікованих за використовуваним методом: випадкова, систематична, пошукова та генерація на основі моделі. Ці методи забезпечують повну автоматизацію, усуваючи необхідність у ручному скриптуванні та обслуговуванні, характерних для API автоматизації та інструментів запису/відтворення.

Головна перевага цих технік полягає у їхній здатності досліджувати непередбачені сценарії та оцінювати толерантність застосунку до взаємодій, не закладених у початковий дизайн, особливо при симуляції системних подій.

2.2.1. Підхід випадкової генерації (Random Input Generation)

Випадковий підхід є найпростішим методом генерації тестів, але він часто характеризується низькою ефективністю та високою надмірністю (взаємодії, які не сприяють покриттю) і повторюваністю.

Monkey. Класичний приклад, який генерує псевдовипадкові потоки користувацьких та системних подій на пристрої чи емуляторі. Його недоліки включають відсутність підтримки введення з клавіатури/системних сповіщень та часте застосування вхідних даних до неінтерактивних областей UI.

Dynodroid. Прагне підвищити ефективність, використовуючи стратегію частоти (вибір рідко використовуваних вхідних даних) та стратегію зваженого випадку (вибір актуальних вхідних даних). Хоча він підтримував системні події (як-от введення з клавіатури та зміна орієнтації), це вимагало інструментування фреймворку Android, що зробило його складним для підтримки та призвело до застаріння.

Toller. Ін'єктує себе в ту ж віртуальну машину, що й AUT, отримуючи прямий доступ до пам'яті для покращення існуючих фреймворків тестування. Вплив оновлень Android на його сумісність є невизначеним.

MonkeyImprover. Пропонує рефакторинг UI для підвищення помітності елементів, що ведуть до складнішого фонового коду, тим самим збільшуючи ймовірність взаємодії з ними з боку Monkey.

Незважаючи на доведену кращу продуктивність інших інструментів, Monkey залишається найпопулярнішим фреймворком завдяки його інтеграції у фреймворк Android та простоті використання, що робить його стандартним базисом порівняння у більшості досліджень.

2.2.2. Метод систематичної генерації (Systematic Input Generation)

Систематична генерація (або навчання моделі) передбачає динамічний аналіз інтерфейсу застосунку та генерацію відповідних вхідних даних під час їх виявлення, керуючись попередньо визначеною стратегією обходу (наприклад, обхід за глибиною). Цей метод є ефективним, оскільки не вимагає попередніх знань про структуру коду.

AndroidRipper. Підтримує модель кінцевого автомата (FSM) під час систематичного обходу UI, створюючи дерево інтерфейсу користувача, що

містить стани та переходи. Його використання на нових версіях Android неможливе через відсутність підтримки.

Automatic Android App Explorer (A3E). Реалізує дослідження за глибиною та цільове дослідження. Страждає від функціональних помилок (наприклад, неможливість натискання кнопок зі спеціальними символами) та застарілої реалізації, що може спричиняти збої.

ACTEve. Пропонує конколічний підхід, який символічно відстежує події від їхнього походження до обробки. Вимагає інструментування як AUT, так і Android SDK.

Mimic. Зосереджений на сумісності застосунків на різних пристроях, використовуючи стратегію "слідуй за лідером" для виявлення відхилень у поведінці паралельно запущених версій.

CrashScope. Використовує статичний аналіз для ідентифікації контекстних функцій (наприклад, використання мережі) та тестує застосунок у різних станах. Хоча він демонструє потенціал, його основна увага зосереджена на генерації зрозумілих звітів про збої, а не на покращенні виявлення помилок.

2.2.3. Метод пошукової генерації (Search-Based Generation)

Пошукове тестування — це популярний метод, який використовує генетичні або еволюційні алгоритми. Воно не вимагає попередніх знань про структуру, часто починаючи з випадково згенерованого тестового набору, який оптимізується пошуковим алгоритмом.

AGRippin. Комбінує генетичні техніки та методи сходження на пагорб для генерації тестів, які максимізують покриття. Оцінка придатності базується на загальній придатності покоління та придатності окремих послідовностей вхідних даних.

Sapienz. Використовує багатопараметричний пошук, поєднуючи фаззингування, систематичне та пошукове дослідження, посів рядків та багаторівневе інструментування. Незважаючи на високу успішність, став

закритим кодом (після придбання Facebook), а публічно доступна версія є застарілою.

ADAPTDROID. Запропонував пошуковий підхід для адаптації існуючих тестів UI між подібними застосунками, використовуючи еволюційний алгоритм для оцінки семантичної подібності до донорського тестового набору.

STGFA-SMOG. Пошуковий підхід, який зосереджується на нефункціональних властивостях (наприклад, використанні CPU/мережі), використовуючи генетичний алгоритм NSGA-II. Він також може використовувати покриття класу/методу/рядка як значення придатності, демонструючи покращення покриття рядків.

Критичний недолік пошукових методів — це надмірно великий час виконання, що зумовлено великою кількістю необхідних виконань тестів та обчислювальною складністю алгоритмів.

2.2.4. Генерація на основі моделі (Model-Based Generation)

Тестування на основі моделі вимагає формальної моделі застосунку (наприклад, CFG, FSM, EFG) для виведення послідовностей вхідних даних. Багато систематичних підходів частково належать до цієї категорії, оскільки створюють модель динамічно.

Stoat та MobiGuitar моделюють AUT як FSM. Stoat використовує статичний та динамічний аналіз, доповнений стратегією зваженого дослідження UI, для побудови стохастичного FSM. Він випадково ін'єктує системні події для імітації змін стану в реальному середовищі, але є неефективним з регулярними жестами. MobiGuitar використовує покращений Android Ripper для динамічного обходу за шириною для генерації FSM, успадковуючи проблеми AndroidRipper.

A3E (цільовий підхід) - використовує статичний аналіз байт-коду для вилучення статичного графа переходів активності (ATG), який потім систематично досліджується.

SwiftHand. Використовує машинне навчання для генерації та покращення моделі AUT під час виконання. Його головна увага зосереджена на мінімізації кількості перезапусків застосунку, що значно покращує покриття порівняно з традиційним випадковим тестуванням.

CrawlDroid. Використовує модель UI для групування елементів керування та уникнення взаємодій, які викликають той самий фоновий код, що мінімізує надлишковість і при цьому максимізує покриття коду.

DroidBot. Легкий генератор вхідних даних, керований UI, використовує модель переходів станів, створену під час виконання. Він забезпечує високорозширюване середовище для розгортання алгоритмів без необхідності інструментування AUT. Дослідження показали, що DroidBot може викликати більше чутливих поведінок (наприклад, доступ до файлів/мережі) порівняно з Monkey.

2.3. Статичний аналіз та моделювання Android-застосунків

Існуючі дослідження запропонували значну кількість робіт, спрямованих на вирішення численних проблем, з якими стикаються аналізатори програм при роботі з Android-застосунками. Ці проблеми включають:

1. Необхідність підтримки або трансляції байт-коду Dalvik.

2. Відсутність явної точки входу, що ускладнює побудову графа викликів.

3. Обмежене врахування обробників подій, через які функціонує застосунок.

Крім того, існують успадковані проблеми аналізу Java, такі як вирішення операторів рефлексії та робота з динамічним завантаженням коду.

Статичний аналіз використовується для досягнення різноманітних цілей. Найпоширенішим застосуванням у дослідженнях є аналіз забруднень (taint analysis) та виявлення витоків конфіденційності, що реалізовано в таких

інструментах, як FlowDroid. Інші цілі включають інструментування коду, символічне виконання та генерацію тестів. FlowDroid є одним із найсучасніших інструментів статичного аналізу для Android; хоча його основна мета — аналіз забруднень, він також може бути використаний для таких цілей, як інструментування коду.

Багато інструментів, як-от Soot (орієнтований на Java статичний аналізатор), AndroGuard (інструмент зворотного інжинірингу) та перекладачі (Dexpler для DEX в Jimple, Apktool), були створені або розширені для полегшення статичного аналізу Android. Через складність прямого аналізу байт-коду Java та Dalvik, ці інструменти часто використовують лінійні проміжні мови (IR). Найбільш популярними та поширеними IR є Jimple та Smali.

2.3.1. Графові представлення проміжної мови

Більшість фреймворків статичного аналізу забезпечують створення графової проміжної мови, що критично важливо для розуміння функцій застосунку та обходу коду. Ці графові представлення поділяються на дві основні категорії:

1. Синтаксично пов'язані дерева.

Використовуються у компіляторах та лінтерах (наприклад, дерево розбору, Абстрактне Синтаксичне Дерево (AST) або Спрямований Ациклічний Граф (DAG)).

2. Графи потоку та залежностей.

Надають поглиблене розуміння структури та функціональних наборів застосунку (Граф Потоку Управління (CFG), Граф Викликів (Call Graph), Граф Залежностей).

2.3.2. Виклики інтеграції з фреймворком Android

Ключовою проблемою статичного аналізу Android є висока інтеграція застосунку з фреймворком. Фреймворк затуманює потік управління між

методами життєвого циклу та зв'язок між елементами керування UI та їхніми відповідними методами зворотного виклику слухачів.

- Граф потоку управління зворотного виклику використовується для моделювання всіх потенційних потоків управління до та від окремих методів життєвого циклу та зворотного виклику UI. Однак, цей підхід не визначає точний шлях або явне посилання від елемента UI до викликаного методу зворотного виклику, пропонуючи натомість вибір можливих шляхів.

- В [11] спробували пов'язати елементи керування UI з їхніми методами зворотного виклику, шукаючи в коді Jimple оголошення елементів UI та призначення їм методів встановлення слухачів. Цей метод лише частково успішний для простих оголошень і рідко заповнює зв'язок у більш складних застосунках.

2.4. Обґрунтування необхідності побудови моделі для генерації тестів

Тестування на основі моделі вимагає формальної моделі Об'єкта Тестування (AUT), яка може бути представлена у різних формах: Граф Потоку Управління (CFG), Кінцевий Автомат (FSM), Граф Подій (EFG) тощо. Тип моделі залежить від інформації, необхідної фреймворку тестування, оскільки модель повинна надавати деталі для керування рішеннями щодо вхідних даних (які вхідні дані надавати, коли і де їх застосовувати).

Попередні підходи до моделювання часто фокусувалися виключно на структурі інтерфейсу користувача (UI), тобто на інтерактивних елементах на екрані, і створювалися шляхом динамічного обходу UI. Хоча фокус на UI є логічним для генерації вхідних даних, внутрішня кодова структура застосунку містить важливі деталі, які UI сам по собі не розкриває. Ці деталі можуть керувати генерацією тестів для досягнення кращого покриття AUT.

Наприклад, для активації прихованої функціональності або елементів UI може знадобитися специфічна послідовність взаємодій (як-от необхідність натиснути на елемент "Версія збірки" 7 разів для активації функцій розробника в налаштуваннях Android).

З огляду на ці вимоги, для цього дослідження було застосовано розширений граф потоку управління (eCFG). Мета eCFG — охопити як інтерфейс застосунку, так і його внутрішню кодову структуру. Це забезпечує високий рівень деталізації для генерації тестів та створює комплексну модель для потенційних майбутніх досліджень.

eCFG включає всі компоненти застосунку, починаючи від низькорівневих інструкцій та внутрішньопроцедурних викликів до високорівневих елементів UI та їхніх методів зворотного виклику. Для перевірки моделі реалізовано концептуальний фреймворк, що передбачає ручне тегування вихідного коду для генерації eCFG для невеликої кількості тестових застосунків.

Розширений Граф Потоку Управління (eCFG) програми P з інтерфейсом користувача U — це спрямований граф $G = (V,E)$, де:

- $V=\{v1,\ldots,vn\}$, $n \in N*$, — це набір вершин, де кожна вершина $vi \in V$ представляє або точку програми $pi \in P$, або взаємодію $ui \in U$.

- $E=\{e1,\ldots,em\}$, $m \in N*$, — це набір дуг (спрямованих ребер), де $ei=(vj,vk) \in V2$ представляє потік управління від pj до pk під час виконання P, або від взаємодії ui до pk після певної взаємодії користувача.

Граф G складається з трьох типів вершин, що представляють складові елементи мобільного застосунку: $V=Vs \cup Vm \cup Vui$ (таблиця 2.1).

Таблиця 2.1.

Опис вершин

Тип Вершини	**Позначення**	**Опис**
Вершини Інструкцій	Vs	Набір вершин, що представляють низькорівневі інструкції (за винятком інструкцій системних/бібліотечних залежностей). Розкривають логіку, внутрішні виклики та

		детальний потік управління.
Вершини Методів	Vm	Набір вершин, що представляють точки входу в методи застосунку (не системні). Поділяються на: методи життєвого циклу Android, методи зворотного виклику вхідних даних користувача та стандартні методи Java.
Вершини Інтерфейсу Користувача	Vui	Набір вершин, що представляють інтерактивний елемент керування UI, з яким користувач може взаємодіяти.

Набір спрямованих ребер E класифікується на чотири підмножини, що представляють різні типи передачі потоку управління: $E=Eui \cup Em \cup Es \cup Ec$ (таблиця 2.2).

Таблиця 2.2.

Опис ребер

Тип ребра	**Позначення**	**Потік Управління**	**Опис**
Ребра UI	Eui (Vui→Vm)	Передача управління від UI	Представляють передачу управління від елемента UI, з яким взаємодіяв користувач, до пов'язаної вершини методу зворотного виклику.
Ребра Методів	Em (Vm→Vs)	Початок виконання методу	Передача управління від вершини методу до кореневої вершини інструкції цього методу.
Ребра Інструкцій	Es (Vs→Vs)	Послідовний потік інструкцій	Представляють потік управління від однієї інструкції до іншої, моделюючи логіку гілок та циклів.
Ребра Виклику Методів	Ec (Vs→Vm)	Виклик іншого методу	Вершина інструкції, що містить виклик методу, передає управління до вершини методу, що викликається.

Модель eCFG забезпечує всебічне представлення Android-застосунку, від низькорівневих інструкцій до високорівневих елементів UI, надаючи загальний огляд потоку управління.

При застосуванні вхідних даних модель може бути використана для визначення шляхів виконання, які можуть бути досягнуті цією взаємодією.

У зворотному напрямку eCFG може визначити послідовності взаємодій, необхідні для досягнення певної функціональності.

На відміну від традиційних CFG, eCFG забезпечує кращу категоризацію компонентів (методи життєвого циклу, слухачі) та ребер потоку управління (тригери зворотного виклику UI, виклики методів), надаючи алгоритмам пошуку контекст у моделі застосунку.

На рисунку 2.1 представлено приклад підмножини eCFG, що ілюструє взаємозв'язок між UI, методами та інструкціями для ActivityA та ActivityB (деталі коду наведено у лістингу 2.1). Пунктирні стрілки позначають потік управління, визначений фреймворком Android.

Лістинг 2.1. Активність A та активність B, що використовуються для генерації eCFG на рисунку 2.1.

```
public class ActivityA extends AppCompatActivity {
    private Button start_B;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_a);
        start_B = findViewById(R.id.btn_start_b);
        // "startActivityB" callback assigned in layout XML
    }

    public void startActivityB(View view) {
        Intent i = new Intent(ActivityA.this, ActivityB.class);
        startActivity(i);
    }

    @Override
    protected void onStart() {
        super.onStart();
    }
}

public class ActivityB extends AppCompatActivity {
    private Button say_hello;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_b);
        say_hello = findViewById(R.id.btn_say_hello);
        // "sayHello" callback assigned in layout XML
    }
```

```
    public void sayHello(View view) {
        printMessage("Hello");
    }

    @Override
    protected void onResume() {
        super.onResume();
        printMessage("Resumed");
    }

    public void printMessage(String message) {
        String message = "Hello";
        System.out.println(message);
    }
}
```

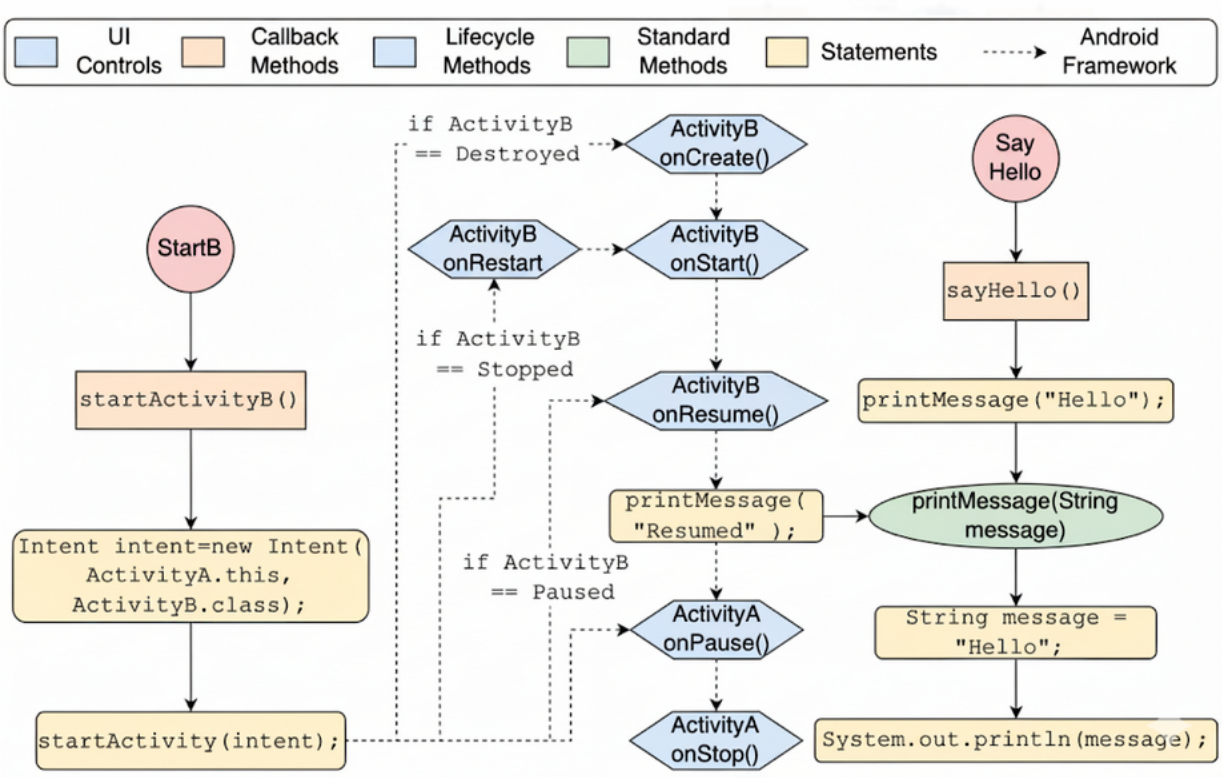

Рис. 2.3. Підмножина eCFG, що представляє Android-застосунок

2.5. Концептуальна реалізація розширеного графа потоку управління

Традиційні методи аналізу потоку управління є непридатними для безпосереднього застосування до Android-застосунків через їхню високу інтеграцію з фреймворком та подієво-керовану природу. З цієї причини для побудови моделі eCFG використовується комбінація спеціалізованих або адаптованих інструментів статичного аналізу.

Для наповнення eCFG використовується статичний аналіз за допомогою фреймворків Soot та FlowDroid. Мотивація полягає в тому, що їхня проміжна мова Jimple є найбільш прийнятим форматом для статичного аналізу Java-орієнтованих програм, включаючи Android. Ці інструменти також надають потужний міжпроцедурний та внутрішньопроцедурний аналіз.

Незважаючи на високу цінність структур даних, наданих Soot і FlowDroid, вони були розроблені переважно для виявлення шкідливого програмного забезпечення та витоків даних.

Їхні роз'єднані структури не є оптимальними для алгоритмів пошуку та обходу, необхідних для генерації тестів. Крім того, пряме використання цих структур призвело б до жорсткої прив'язки реалізації eCFG до базового фреймворку статичного аналізу.

З метою забезпечення легшої навігації в отриманій моделі та збереження можливості заміни базового фреймворку, дані, отримані від Soot та FlowDroid, використовуються для наповнення спеціально розробленої графової реалізації на базі бібліотеки JGraphT.

eCFG реалізовано як граф JGraphT, що містить три типи вершин, визначених у попередньому розділі:

1. Вершини Інструкцій (Vs)

Збір даних. Інструкції збираються з об'єктів UnitGraph FlowDroid, які створюються для кожного методу в AUT. Кожен UnitGraph містить ланцюжок інструкцій Jimple та їхній порядок виконання.

Ребра. Визначаються міжпроцедурні виклики для кожної інструкції, і відповідні ребра додаються до графа.

2. Вершини Методів (Vm)

Ідентифікація. Створюються шляхом отримання всіх методів для кожного класу, ідентифікованого FlowDroid в AUT. Системні та бібліотечні методи виключаються на основі їхніх назв пакетів.

Категоризація. Перед додаванням до графа методи класифікуються як: методи життєвого циклу активності, методи зворотного виклику вхідних даних або стандартні методи Java.

3. Вершини Інтерфейсу Користувача (Vui)

Ідентифікація. FlowDroid ідентифікує елементи керування UI за допомогою файлів XML компонування Android, включених у скомпільований APK.

Обмеження. FlowDroid може ідентифікувати елементи, оголошені в XML, але не може ідентифікувати елементи, оголошені безпосередньо у вихідному коді Java. Для цього концептуального доведення аналізуються лише невеликі застосунки без спеціальних переглядів у Java-коді. Методика включення цих елементів буде деталізована у наступному розділі.

Вершини в eCFG містять атрибути, що визначають їхнє розташування:

- Вершини інструкцій (Vs) мають атрибут методу.
- Вершини методів (Vm) мають атрибут класу.
- Вершини UI (Vui) мають атрибути активності та компонування.

Під час наповнення графа додаються відповідні ребра потоку управління. Однак, не всі ребра можуть бути включені через їхнє походження ззовні коду застосунку (наприклад, з фреймворку Android). Методи життєвого циклу активності викликаються фреймворком на основі системного/застосункового стану і не мають явного виклику в коді. Це призводить до того, що ці методи життєвого циклу з'являються як роз'єднані кластери в графі, а потік управління до них визначається динамічно під час виконання.

На Рисунку 2.4 представлено візуалізацію повного eCFG моделі Android-застосунку, створену за допомогою програмного забезпечення Gephi. Незважаючи на складність графа, яка ускладнює розрізнення окремих ребер, складність моделі очевидна навіть для невеликого застосунку. Також чітко видно роз'єднаність (відсутність зв'язків фреймворку Android, що були представлені пунктирними лініями на рисунку 2.3).

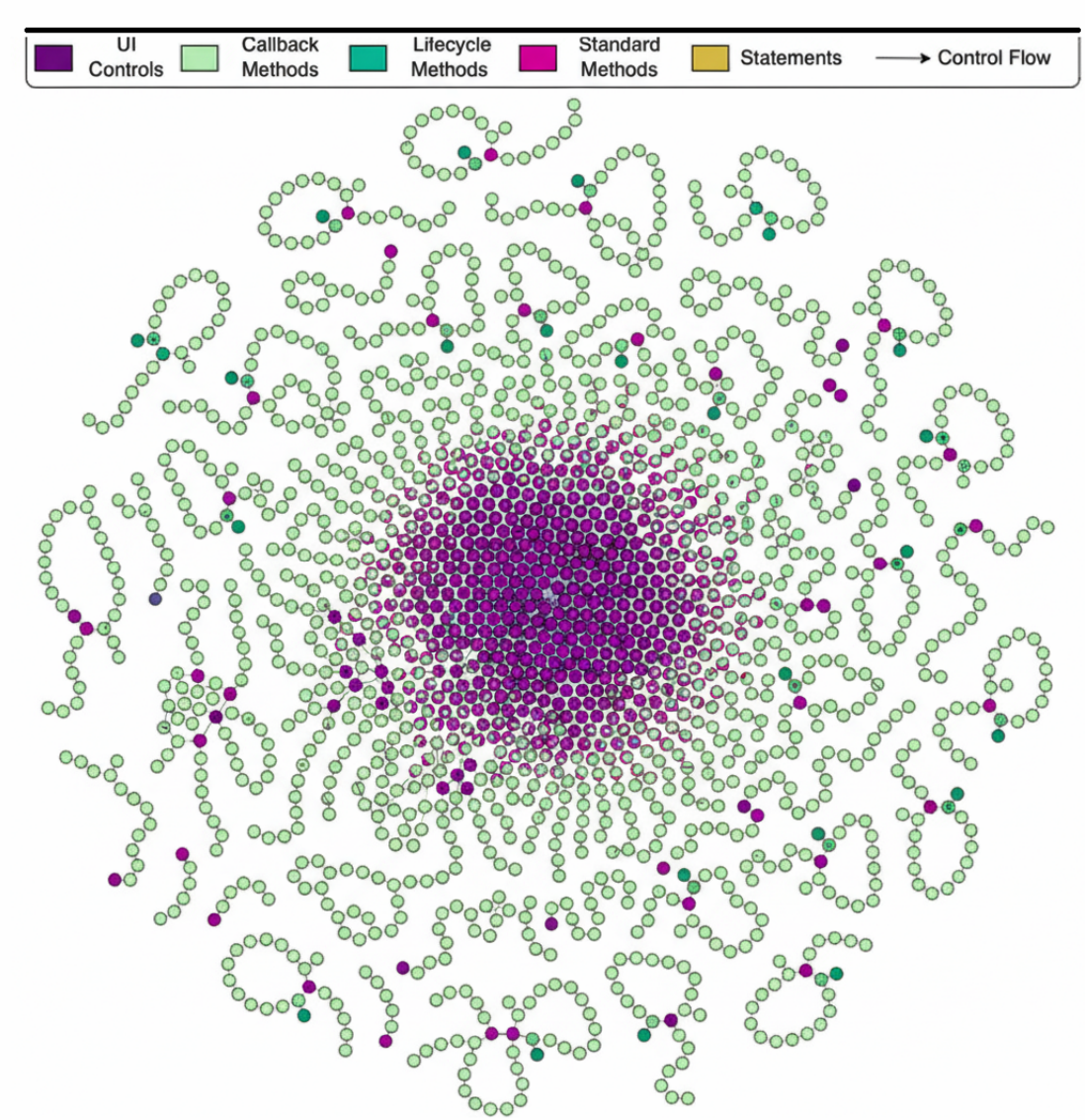

Рис. 2.4. Повний граф потоку управління, що представляє Android-додаток

2.6. Визначення зв'язків UI-код у контексті статичного аналізу

Для точного моделювання потоку управління, ініційованого користувачем, необхідно встановити коректне зв'язування між елементами керування інтерфейсом користувача (UI контролами) та їхніми асоційованими методами зворотного виклику слухачів (listener callback methods). Це зв'язування, однак, є неоднозначним у процесі статичного аналізу, що зумовлено двома основними факторами:

1. Неоднорідність назв методів зворотного виклику.

Більшість методів зворотного виклику слухачів мають схожі або ідентичні сигнатури, наприклад, onClick().

2. Складність відстеження призначень змінних.

Не є тривіальним завданням відстеження присвоєнь змінних через байт-код для визначення їхнього походження.

2.6.1. Проблема автоматичного зв'язування

Спроби автоматичного зв'язування передбачають пошук у байт-коді Jimple викликів методу setListener(). При виявленні такого виклику, змінна відстежується через байт-код для ідентифікації її оголошення або призначення, що дозволяє визначити ідентифікатор ресурсу викликаючого елемента керування UI.

Цей підхід є успішним лише для простих оголошень. Проте, велика кількість можливих методів setListener(), обмеження у відстеженні значень змінних та значна неоднозначність у стилях кодування розробників роблять цю задачу нетривіальною для складних кодових баз.

У межах поточної концептуальної моделі для подолання проблеми зв'язування UI-код застосовується ручне тегування вихідного коду. Це передбачає вставку операторів друку з відповідним ідентифікатором ресурсу елемента керування UI у методи зворотного виклику слухачів. Під час статичного аналізу, цей вручну вставлений ідентифікатор ресурсу виявляється в інструкціях Jimple, що дозволяє додати відповідне ребро до eCFG.

Ця концептуальна реалізація використовується у наступному розділі для представлення трьох невеликих Android-застосунків з відкритим вихідним кодом та для генерації послідовностей тестових вхідних даних.

2.6.2. Масштабованість та автоматизація

Концептуальний фреймворк, що використовує ручне тегування, не є масштабованим для більших застосунків. Ручне тегування є лише

тимчасовим рішенням. Згодом цей процес буде автоматизовано шляхом комбінування автоматизованого інструментування, статичного та динамічного аналізу.

Висновки до розділу

У цьому розділі було визначено структуру даних розширеного графа потоку управління як інструмент представлення Android-застосунку. Розроблений з акцентом на генерацію тестових вхідних даних на основі моделі, граф надає вичерпні структурні деталі застосунку, охоплюючи діапазон від низькорівневих інструкцій та внутрішньопроцедурних викликів до високорівневих елементів інтерфейсу та їхніх методів зворотного виклику. Також було описано структуру графу та концептуальну реалізацію, що використовує статичний аналіз, наданий FlowDroid, доповнений ручним тегуванням.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ГЕНЕРАЦІЇ ПОСЛІДОВНОСТЕЙ ВХІДНИХ ДАНИХ ТЕСТУВАННЯ МОБІЛЬНИХ ЗАСТОСУНКІВ

3.1. Алгоритми генерації послідовностей вхідних даних з розширеного графу потоку управління

Цей розділ досліджує, як використання даних про структуру застосунку, інкапсульованих у розширеному графі потоку управління (eCFG), може підвищити ефективність генерації тестових послідовностей, мінімізуючи надлишковість вхідних даних та максимізуючи покриття моделі. Тут порівнюється простий підхід генерації тестів на основі моделі з існуючим методом випадкової генерації, реалізованим у Monkey.

Було визначено eCFG як структуру даних, яка представляє Android-застосунок, надаючи структурні деталі від низькорівневих інструкцій до високорівневих елементів UI та їхніх зворотних викликів. Цей розділ демонструє корисність eCFG у створенні більш ефективних та продуктивних тестів.

Monkey++ — це простий фреймворк обходу, що використовує стратегію пошуку за глибиною (DFS), розроблений для демонстрації переваг eCFG у генерації послідовностей вхідних даних. Оскільки eCFG реалізовано як концептуальне доведення з ручним тегуванням вихідного коду, згенеровані послідовності вхідних даних не виконуються на об'єкті тестування (AUT).

Випадкові генератори вхідних даних, такі як Monkey, можуть досягати адекватного покриття, але часто вимагають великої кількості взаємодій. Значна частина цих випадкових взаємодій не є значущою для AUT. Попереднє знання структури застосунку, надане eCFG, дозволяє більш ефективний та продуктивний вибір вхідних даних.

Для оцінки впливу eCFG та покращення чисто випадкового підходу Monkey (який не використовує знань про AUT), розглядаються наступні питання:

1. Наскільки ефективні випадково згенеровані вхідні дані Monkey для взаємодії з AUT?

2. Чи може Monkey++ усунути дублікати або надлишкові тестові вхідні дані, використовуючи eCFG AUT?

3. Чи досягають послідовності вхідних даних Monkey++ кращого покриття моделі, ніж у Monkey?

Експериментальна оцінка, проведена на 3 застосунках з відкритим вихідним кодом, показала, що хоча Monkey досягає в середньому принаймні 85% покриття UI, йому для цього потрібно 500 вхідних даних, причому лише 8% з них, в середньому, фактично взаємодіють із застосунком. Випадковий характер Monkey створює повільні тести з високою часткою надлишкових вхідних даних. Натомість, Monkey++, використовуючи просту стратегію обходу моделі, генерує послідовності, які досягають вищого покриття UI з набагато меншою кількістю вхідних даних. Це свідчить про те, що інтеграція знань про структуру застосунку через моделі, подібні до eCFG, може значно підвищити ефективність алгоритмів генерації тестів.

Monkey++ — це генератор послідовностей вхідних даних, розроблений для використання уявлень, наданих eCFG. Замість генерації випадкових вхідних даних, eCFG дозволяє Monkey++ ідентифікувати інтерактивні елементи керування UI, зменшуючи надлишковість та усуваючи дублікати.

Monkey++ використовує стратегію навігації DFS для обходу eCFG, що забезпечує краще покриття з меншою кількістю взаємодій. Він відстежує взаємодію з елементами, анотуючи відповідні вершини UI в eCFG.

3.1.1. Алгоритм генерації тесту

Алгоритм 3.1 (який описує роботу Monkey++) функціонує наступним чином:

Початок: Monkey++ починає з активності запуску за замовчуванням (стартовий екран AUT).

Отримання UI-Елементів: Використовуючи eCFG, отримуються доступні вершини елементів керування UI для поточної активності (рядок 8).

Вибір Вхідних Даних: Monkey++ шукає доступний елемент UI, який не був відвіданий (атрибут відвідування = false), а якщо таких немає, то шукає елемент, який не має локального відвідування (атрибут локальний = false) (рядок 9). Це забезпечує DFS, надаючи пріоритет невикористаним елементам.

Генерація та Анотація: Після додавання вхідних даних до тестової послідовності, відповідна вершина UI позначається атрибутами "відвідування" (загальне покриття моделі) та "локального" (покриття в межах поточної ітерації обходу).

Симуляція Виконання: Monkey++ імітує виконання цього вхідного даних за допомогою eCFG (рядок 17), щоб спрогнозувати наступну активність, яка буде відображена.

Алгоритм 3.1. Генерація послідовності вхідних даних

```
Input: graph, the eCFG of the AUT
Input: status, map from activity to lifecycle status
Output: inputs, a test input sequence for the app

1: inputs ← ∅
2: stack = empty() ▷ the traversal activity stack
3: launch = graph.getLaunchActivity()
4: stack.put(launch)
5: count ← 0
6: while count < 500 do
7:     activity ← stack.peek() ▷ retrieves the currently active activity
8:     controls ← graph.getAvailableControls(activity)
9:     for i ∈ {"visit", "local"} do ▷ checks "visit" status of vertices, if true
10:         for all, then checks "local" status of vertices
11:             for c ∈ controls do
12:                 if c.getAttribute(i) == false then
13:                     c.setAttribute(visit, true)
14:                     c.setAttribute(local, true)
15:                     inputs ← inputs.add(c)
16:                     count ← count + 1
17:                     traverse(c, status, stack)
18:                     go to 6
19:     if activity ≠ launch then
20:         stack.pop()
21:     else
22:         graph.resetLocalStatus()
23: Return: inputs
```

Повернення та Скидання: Коли всі вершини UI в активності використані, Monkey++ повертається на одну активність назад (імітуючи кнопку "назад") (рядок 20). Якщо відбувається повернення до активності запуску, а елементів, з якими не взаємодіяли, більше немає, атрибут "локальний" скидається для всіх вершин, щоб дозволити повторне відвідування та потенційне відкриття нових шляхів (рядок 22).

Обмеження повторюваності вхідних даних зменшує ймовірність нескінченних циклів, але також означає, що Monkey++ не може ефективно генерувати послідовності з численними послідовними повторюваними взаємодіями (наприклад, 7 послідовних натискань, необхідних для активації функцій розробника).

Оскільки eCFG не містить ребер потоку управління між методами життєвого циклу активності (вони контролюються фреймворком Android), Monkey++ прогнозує стек активностей та статус життєвого циклу кожної активності на основі поточної послідовності вхідних даних. Якщо інструкція представляє запуск активності (наприклад, startActivity(intent)), нова активність додається до стеку (оголошеного в алгоритмі 3.1), і відповідні методи життєвого циклу включаються до шляху обходу.

3.1.2. Алгоритм рекурсивного обходу моделі

Алгоритм 3.2 формалізує рекурсивний метод, який використовується фреймворком Monkey++ для обходу шляху в моделі (eCFG), ініційованого певною взаємодією (відповідає рядку 17 в Алгоритмі 3.1).

Під час відвідування вершини в eCFG Monkey++ здійснює пошук трьох ключових ознак, що впливають на потік управління застосунком (рядки 5–9 в алгоритмі 3.2):

1. Оператор запуску активності (Activity Launch Statement).
2. Оператор завершення активності (Activity Finish Statement).
3. Метод життєвого циклу (Lifecycle Method).

У разі виявлення будь-якої з цих ознак, стан стеку активностей та статус життєвого циклу застосунку негайно оновлюються. Ці оновлені параметри використовуються для динамічного визначення подальшого напрямку пошуку в графі.

Алгоритм 3.2. Рекурсивний метод обходу вершин в eCFG

```
Input: v, a vertex of the application eCFG
Input: status, map from activity to lifecycle status
Input: stack, the traversal activity stack
1: procedure TRAVERSE(v)
2:     for e ∈ v.getOutgoingEdges() do
3:         t ← e.getEdgeTarget()
4:         traverse(t)
5:     if v.getAttribute(type) == STATEMENT then
6:         checkForIntentAndStart(v, stack) ▷ adds new activity to the stack
7:         checkForActivityFinish(v, stack) ▷ removes activity from the stack
8:     else if v.getAttribute(type) == LIFECYCLE then
9:         updateLifecycle(v, status) ▷ updates the lifecycle status map
```

Як приклад, якщо вершина v має тип STATEMENT (тобто $v \in Vs$), викликається метод checkForIntentAndStart(v). Припускаючи, що вершина v представляє оператор, як-от startActivity(intent), що сигналізує про запуск нової активності:

1. Оновлення Стеку: Активність, що запускається, додається на вершину стеку активностей.

2. Продовження Обходу: Відповідні методи життєвого циклу додаються до шляху обходу, що дозволяє Monkey++ коректно симулювати виконання Android-фреймворку.

Таким чином, алгоритм 3.2 забезпечує механізм імітації динамічної поведінки Android-застосунку під час статичного обходу моделі, що є критично важливим для точної генерації тестових послідовностей.

3.2. Комбінований підхід до моделювання Android-застосунків

Цей розділ досліджує обмеження як статичного аналізу, так і динамічних тестів у створенні комплексної моделі Android-застосунку. У

ньому демонструється, як інтеграція цих двох підходів може значно підвищити повноту отриманої моделі. Ефективність додавання динамічного тестування перевіряється шляхом порівняння результатів до і після його включення, а також порівнянням покриття, досягнутого дослідницькими тестами, з покриттям, отриманим випадково згенерованими тестами.

Хоча моделювання програмного забезпечення не є новою дослідницькою сферою, мобільні платформи, такі як Android, створюють низку викликів порівняно з класичними послідовними програмами.

3.2.1. Обмеження статичного аналізу

Як відомо, традиційні методи статичного аналізу значною мірою непридатні, оскільки шляхи виконання обфускуються до моменту виконання (runtime). Зокрема, статичний аналіз Android-застосунків має прогалини у компонентах та зв'язках потоку управління, наприклад, не може врахувати елементи керування, оголошені безпосередньо в коді Java. Хоча деякі з цих елементів можна було б отримати лише за допомогою статичного аналізу, це вимагало б значних зусиль для розробки та постійного обслуговування, щоб подолати такі труднощі, як еволюція API та різноманітність стилів кодування розробників. Результатом була б нестабільна система, яка постійно відстає від нових тенденцій.

3.2.2. Обмеження динамічного аналізу

Найбільш поширеним методом генерації моделі є динамічний аналіз застосунку та збір даних для наповнення моделі. Цей метод є простим, надійним і, ймовірно, буде працювати з майбутніми поколіннями Android з мінімальним обслуговуванням. Збір даних є надійним завдяки використанню інструментування коду або функцій дампів інтерфейсу, доступних в API автоматизації, наприклад, UIAutomator.

Однак динамічний аналіз залежить від ефективності базового алгоритму обходу і не може гарантувати повноту отриманої моделі.

3.2.3. Комбінований підхід

Цей розділ оцінює обмеження обох підходів у моделюванні Android-застосунків для генерації вхідних даних на основі моделі та з використанням DroidGr. DroidGr — це фреймворк, призначений для генерації комплексного eCFG Android-застосунку, що охоплює весь спектр від низькорівневого коду інструкцій бекенду до фронтенд-контролів інтерфейсу користувача та зворотних викликів. Він використовує традиційний статичний аналіз, посилений ефективними систематичними дослідницькими тестами на AUT.

DroidGr використовується для дослідження наступних питань:

1. Наскільки додавання динамічного аналізу підвищує повноту моделі застосунку порівняно з чистим статичним аналізом?

2. Чи можуть систематично згенеровані дослідницькі тести досягти більшого покриття застосунку, ніж випадково згенеровані вхідні дані, які використовуються наразі?

3. Яка різниця в ефективності між систематично та випадково згенерованими дослідницькими тестами?

Для вивчення цих питань DroidGr було застосовано декілька різноманітних застосунків.

3.3. Інтеграція статичного та динамічного аналізу для побудови графів управління в мобільному застосунку

Компоненти та структура, що визначають розширений граф потоку управління (eCFG) залишаються незмінними у фреймворку DroidGr. Однак, реалізація DroidGr і метод генерації графу управління значно різняться. Найважливішою зміною є вилучення ручного тегування вихідного коду AUT та його заміна комбінацією автоматичного інструментування та систематичного пошуку. Цей розділ деталізує зміни у процесі генерації графу управління та загальній реалізації DroidGr.

Розглянемо структуру компонентів фреймворки DroidGr.

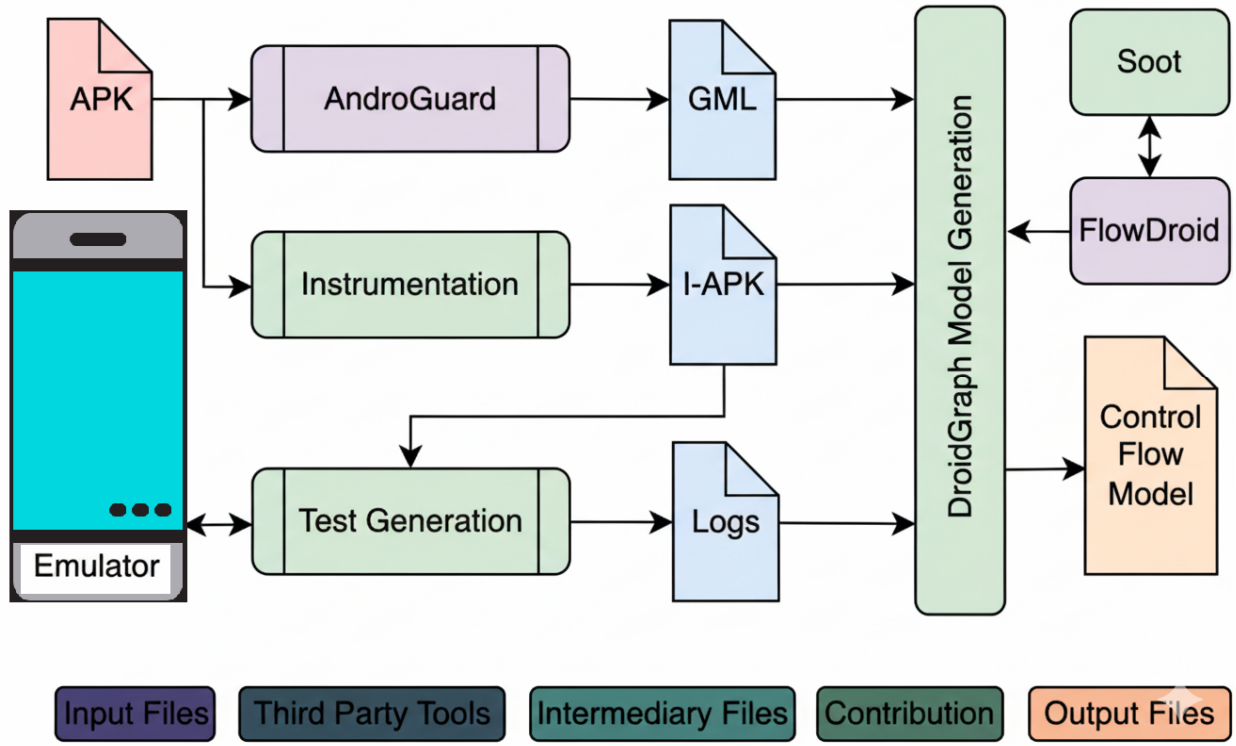

Рис. 3.1. Компоненти структури фреймворки DroidGr

DroidGr складається з декількох інструментів і компонентів, що взаємодіють для створення eCFG Android-застосунку. Процес включає наступні кроки (рисунок 3.1):

1. Вхідні дані. AUT надається у вигляді файлу APK.

2. Паралельна обробка. APK подається на вхід модулю інструментування DroidGr та інструменту AndroGuard.

3. Вихідні дані інструментування. Отримується інструментований APK-файл та файл графової мови моделювання (GML) графа викликів.

4. Динамічне виконання. Інструментований APK запускається з тестами, що генерує файли журналів, які містять виконані дії та знайдені компоненти застосунку.

5. Фінальна генерація графу управління. DroidGr використовує GML-граф викликів, інструментований APK та журнали дослідницьких тестів для генерації eCFG AUT.

3.3.1. Інструментування

Інструментування AUT здійснюється за допомогою Soot для вирішення трьох основних завдань: виявлення зв'язків між елементами UI та методами зворотного виклику під час виконання, а також обчислення покриття інтерфейсу та методів в експериментах.

Інструментування передбачає додавання оператора логування до кожного методу застосунку (за винятком методів системних і бібліотечних залежностей), що містить наступні дані:

- Тег Логу: Тег <CONTROL> використовується для методів із параметром типу android.view.View; тег <METHOD> для всіх інших методів. Теги використовуються для фільтрації журналів.

- Назва Методу: Використовується для 1) зв'язування методу зворотного виклику з ідентифікатором представлення інтерфейсу та 2) обчислення покриття методів.

- ID Представлення (View ID): Методи зворотного виклику приймають об'єкт типу android.view.View як вхідний параметр. Якщо цей об'єкт присутній, реєструється виклик методу view.getId(). Це дозволяє ідентифікувати елемент UI, взаємодія з яким викликала метод зворотного виклику під час виконання.

Статичний аналіз, використаний для наповнення графу залишається в основі DroidGr, оскільки він надає об'єкти для реалізації графа JGraphT.

Однак, оскільки Soot та FlowDroid не сфокусовані безпосередньо на статичному аналізі Android-застосунків (FlowDroid орієнтований на аналіз забруднень), деякі їхні функції стали застарілими та ненадійними. Наприклад, FlowDroid не має сумісності з фрагментами Android, створеними за допомогою AndroidX, що призводить до відсутності класів фрагментів у графі викликів.

Для подолання цього DroidGr використовує AndroGuard — інструмент зворотного інжинірингу на основі Python, здатний генерувати прийнятний граф викликів Android-застосунку.

AndroGuard генерує граф викликів для інструментованого AUT у форматі GML. DroidGr імпортує GML-граф, фільтрує зовнішні методи та доповнює статичний аналіз FlowDroid відсутніми ребрами між вершинами методів.

Прийнятність графа викликів AndroGuard була підтверджена ручною інспекцією випадкових підрозділів вмісту та структури графа, що забезпечило достатній рівень довіри в контексті цього дослідження.

3.3.2. Покращення моделі через динамічний аналіз

Через значні проблеми зі статичним аналізом (наприклад, елементи керування, оголошені в коді Java, та мінливі API), використання динамічного аналізу шляхом виконання інструментованого застосунку є простим, надійним і більш стійким до майбутніх змін у Android.

Дослідження AUT:

- Для дослідження AUT використовується Appium, відкрите програмне забезпечення для автоматизації взаємодії з UI.

- Обхід імітує пошук за глибиною (DFS) на графічному інтерфейсі застосунку. Appium інтегрований з UIAutomator і надає деталі про доступні елементи UI та їхнє розташування, що дозволяє уникнути безрезультатних взаємодій.

- Систематичний обхід уникає непотрібних повторюваних вхідних даних шляхом відстеження використаних елементів UI та позначення вузлів-листів (які не ведуть до подальших активностей).

Оновлення моделі:

- Під час дослідницьких тестів відстежуються повідомлення інструментування в журналах.

- Якщо виявляється компонент, відсутній в eCFG, він додається.

- Зокрема, при знаходженні логу з тегом <CONTROL>, eCFG доповнюється зв'язком між вершинами, що представляють елемент UI та метод зворотного виклику, знайдений у лозі.

Динамічний етап DroidGr надає надійні (не випадкові) тести для підтримки побудови моделі та доповнення статичного аналізу, пропонуючи надійнішу альтернативу випадковим тестам, як-от ті, що генеруються Monkey.

DroidGr генерує eCFG, використовуючи інформацію, надану сторонніми фреймворками статичного аналізу (Soot, FlowDroid, AndroGuard), а також дані часу виконання від динамічного обходу AUT.

Помилкові Негативи. Відсутність або неточність вмісту від цих методів, хоча й не заважає DroidGr працювати, може призвести до неточного моделювання. Будь-який алгоритм, що використовує отриманий eCFG, буде вразливий до неефективності, спричиненої потенційними хибними негативами. Точність моделі перевірялася ручною верифікацією випадкових підрозділів.

Масштабованість (Scalability). Масштабованість DroidGr не оцінювалася в цьому дослідженні.

Модель. eCFG реалізовано як структура даних графа JGraphT, що зберігається в пам'яті. Більші застосунки вимагатимуть значно більше пам'яті; для підвищення масштабованості необхідне рішення на основі графової бази даних.

Зі збільшенням розміру AUT зростає і простір пошуку, що вимагає більше дослідницьких тестів. DFS менш ефективний, коли в інтерфейсі застосунку присутні цикли. Алгоритм обходу потребує вдосконалення для забезпечення кращої продуктивності та збільшення покриття великих застосунків в межах того ж тестового бюджету.

3.4. Методика генерації точних автоматизованих вхідних даних для Android

У цьому розділі представлено фреймворк заснований на моделі, який використовує розширений граф потоку управління (eCFG) Android-

застосунку, створений за допомогою DroidGr. Метою фреймворку є генерація ефективних тестових вхідних даних, що забезпечують вище покриття з меншим часом виконання генерації тесту. Досягнуте покриття та час виконання запропонованого підходу порівнюються з передовими інструментами, кожен з яких використовує відмінну техніку автоматизованої генерації вхідних даних.

Автоматизована генерація тестових вхідних даних продемонструвала обнадійливі результати в минулих дослідженнях із застосуванням різноманітних підходів.

Методи генерації вхідних даних:

- Випадкова генерація (Random) є поширеною та легкою в обслуговуванні технікою, але вона є неефективною, призводячи до великих та малорезультативних тестів.

- Систематичні (Systematic) та пошукові (Search-based) методи демонструють багатообіцяючі результати, проте вимагають надмірного часу виконання генерації.

Модельовану генерацію (Model-based) широко використовують у дослідженнях, і моделювання також застосовується для посилення альтернативних підходів. Хоча розуміння та моделювання Об'єкта Тестування (AUT) створює додаткові початкові накладні витрати, це призводить до швидшої генерації тестів та більш ефективних тестів у підсумку.

Досліджуваний фреймворк є композицією різних інструментів і компонентів, що забезпечують генерацію тестових вхідних даних на основі моделі для Android-застосунку. Загальний огляд процесу представлений на рисунку 3.2.

1. Вхідні Дані - об'єкт тестування (AUT) надається у вигляді файлу APK.

2. Інструментування. APK інструментується за методом, визначеним для отримання інструментованого APK.

3. Генерація моделі (eCFG). Інструментований APK подається на вхід DroidGr, який створює eCFG застосунку та виводить його у форматі JSON для повторного використання. Хоча фреймворк може працювати без інструментованого APK, він використовує той самий вхідний APK, що й DroidGr, для забезпечення консистентності.

4. Виконання та взаємодія. Фреймворк досліджує AUT під час генерації вхідних даних, використовуючи Appium (інтегрований з UIAutomator), що працює на емуляторі Android. Appium надає детальну інформацію про доступні елементи керування UI.

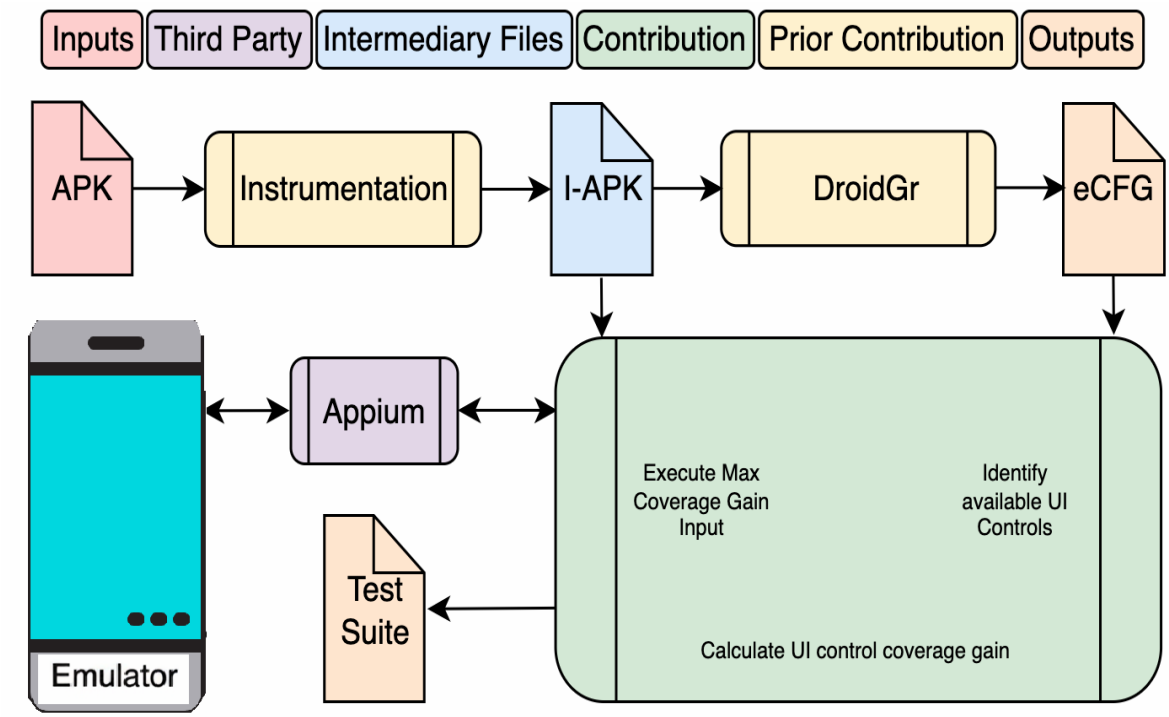

Рис. 3.2. Структура фреймворку на основі моделі

Використовуючи eCFG, APK та Appium, фреймворк генерує та виводить послідовність вхідних даних (тест) для AUT.

Алгоритм 3.3 описує процес, за допомогою якого фреймворк генерує тестові вхідні дані.

1. Ініціалізація. Запускається застосунок у стандартній активності запуску на емуляторі Android (рядок 3).

Алгоритм 3.3. Генерація послідовності вхідних даних з графу управління

```
Input: graph, the eCFG of the AUT
Input: N, the number of inputs to generate.
Output: test, a test, i.e, a sequence of inputs.
1: test ← ∅
2: tracker ← ∅
3: launchApp()
4: for count ← 0 to N do
5:     controls ← getAvailableControls()
6:     coverageGains ← ∅
7:     for c ∈ controls do
8:         graph.resetLocalStatus()
9:         g ← graph.calculateCoverageGain(c)
10:         coverageGains.add(c, g)
11:     maxControls ← maxValue(coverageGains)
12:     if maxControls.size() > 1 then
13:         next ← tracker.getLeastFrequent(maxControls)
14:     else
15:         next ← maxControls.get(0)
16:     next.click()
17:     tracker.addInput(next)
18:     test.add(next.getAdbCommand())
19:     graph.markCoveredVertices(next)
20: return test
```

2. Виявлення Контролів. За допомогою Appium отримуються всі доступні елементи керування UI для поточної активності, включаючи кнопку "назад" (рядок 5).

3. Призначення значення придатності (Fitness Value). Кожному доступному елементу керування UI призначається значення придатності, що відображає потенційний приріст покриття.

- Приріст покриття визначається шляхом обходу eCFG, починаючи з асоційованої вершини елемента UI, доки шлях не завершиться (рядки 7–10).

- Перед кожним обчисленням значення придатності локальний статус вершини в графі скидається (рядок 8).

4. Вибір наступного входу: Обирається елемент керування UI з максимальним значенням придатності (рядок 11).

- Вирішення колізій: Якщо кілька елементів UI мають однакове максимальне значення придатності, PADRAIG використовує стратегію найменш часто використовуваного (Least Frequently Used), відстежуючи, який елемент UI використовувався найменше (рядки 12–13, 17).

- Виконання та анотація: Після вибору взаємодії, відповідна вершина UI та всі шляхи, що ведуть від неї в eCFG, позначаються як відвідані (рядок 19).

Алгоритм 3.4 деталізує, як обчислюється значення придатності для вершини елемента керування UI (v). Повернене значення — це кількість вершин, які потенційно будуть відвідані внаслідок взаємодії з v, але ще не були відвідані раніше.

Алгоритм 3.4. Розрахунок приросту покриття вершини UI-контролю

```
Input: v, a UI control vertex in the eCFG
Output: g, the potential coverage gain of the UI control vertex v
1: procedure CALCULATECOVERAGEGAIN(v)
2:     g ← 0
3:     if v.getAttribute("visit") == false then
4:         g ← g + 1
5:     if v.getAttribute(type) == STATEMENT and v.hasFeature() then
6:         g ← g + 1
7:     for e ∈ v.getOutGoingEdges() do
8:         t ← e.getEdgeTarget()
9:         if t.getAttribute("local") == false then
10:             t.setAttribute("local", true)
11:             g ← g + calculateCoverageGain(t)
12:     return g
```

Кожна невідвідана вершина додає 1 до значення придатності (рядки 3–4). Вершини, що демонструють особливості застосунку, наприклад, запуск нової активності, також збільшують значення придатності, оскільки вони вказують на потенційно недосліджену територію в AUT (рядки 5–6).

Обчислення рекурсивно включає потенційний приріст покриття від усіх вихідних ребер (рядки 7–11).

Розрахований приріст покриття є переоцінкою фактичного досягнутого покриття. Наприклад, при зустрічі з оператором if-else обидві гілки включаються до потенційного приросту, хоча під час виконання буде активовано лише одну. Для точнішого значення покриття необхідний аналіз потоку даних.

Значення придатності також залежать від потенційних хибних негативів у базовому eCFG. Як відомо, не існує автоматизованого методу для верифікації повноти та коректності моделі. Надлишкові або відсутні вершини та ребра в eCFG можуть вплинути на розрахунок придатності, що потенційно призведе до генерації менш ефективних тестів.

Використання значень придатності в даному фреймворку має схожість із пошуковими підходами, такими як підйом на пагорб (hill climbing). Однак, даний фреймворк уникає проблеми локальних максимумів, оскільки значення придатності використовується лише для оцінки якості можливих поточних вхідних даних, а не для визначення загальної якості генерації тесту. Фреймворк не має пам'яті про попередні значення придатності, а обирає найкраще значення серед доступних взаємодій.

Час виконання, необхідний для генерації тесту, визначається кількістю запитаних вхідних даних (N), а не розміром AUT. Фреймворк може функціонувати без моделі, використовуючи лише стратегію найменш часто використовуваного, але його продуктивність, коли він покладається на eCFG DroidGr, залежить від масштабованості та здатності DroidGr відповідати на запити до моделі.

3.5. Експериментальне дослідження фреймворку

Цей розділ деталізує експерименти, проведені для дослідження де представлено бенчмарк-застосунки та описано протоколи й метрики, використані для дослідження поставлених питань.

Для оцінки внеску фреймворку DroidGraph та його механізму дослідницького тестування, сформульовані наступні три питання:

1. Наскільки включення динамічного аналізу покращує модель застосунку? Це питання вивчає кількісне покращення моделі, отриманої статичним аналізом, завдяки додаванню результатів динамічного аналізу, виконаного дослідницькими тестами.

2. Чи можуть систематично згенеровані дослідницькі тести відкрити більше застосунку, ніж випадкові тести? Це питання спрямоване на порівняння досягнутого покриття застосунку систематичними дослідницькими тестами DroidGr з покриттям, отриманим випадковими вхідними даними, згенерованими Monkey.

3. Яка різниця в ефективності між систематично та випадково згенерованими дослідницькими тестами? Це питання оцінює відмінності в ефективності (зокрема, з точки зору кількості необхідних вхідних даних) систематично та випадково згенерованих дослідницьких тестів для обходу інтерфейсу користувача (UI) застосунку.

Для емпіричного дослідження вищезазначених питань DroidGr та його тести були застосовані до трьох безкоштовних застосунків із відкритим вихідним кодом, випадково відібраних із каталогу.

Критерії відбору застосунків включали:

- Відсутність великих промислових застосунків.
- Відсутність спеціальних (custom) представлень у UI застосунку.
- Мінімальні зовнішні бібліотечні залежності.

Обрані застосунки та їхні характеристики представлені в таблиці 3.1.

Таблиця 3.1.

Застосунки для експерименту

Застосунок	Кількість Активностей	UI-контролів	Рядки Коду Java (LoC)
Activity Lifecycle	4	13	558
Mo Clock	3	10	378
Volume Control	4	11	2254

Для кожного застосунку процедура передбачала:

- Генерація базової моделі - DroidGr генерує базову модель застосунку, використовуючи лише статичний аналіз.

- Покращення моделі - модель покращується за допомогою динамічного аналізу, виконаного систематичними дослідницькими тестами.

- Порівняння - дослідницькі тести порівнюються з випадковими вхідними даними, згенерованими Monkey, за метриками покриття інтерфейсу користувача та покриття методів.

- Статистичний аналіз. Використовувалися t-тести для оцінки статистичної значущості покращення покриття, досягнутого дослідницькими тестами порівняно з Monkey.

Результати демонструють, що динамічний аналіз суттєво доповнює модель, згенеровану статичним аналізом. Кількість покращень моделі (доданих компонентів або зв'язків), внесених дослідницькими тестами, наведена нижче.

Таблиця 3.2.

Показники покращення моделі

Застосунок	**Кількість Покращень Моделі**
Activity Lifecycle	12
Mo Clock	8
Volume Control	15

Систематично згенеровані дослідницькі тести продемонстрували здатність досліджувати більшу частину застосунку, ніж випадкові тести Monkey, про що свідчать досягнуті показники покриття.

Таблиця 3.3.

Покриття UI (%)

Застосунок	Monkey (%)	Дослідницькі Тести (%)
Activity Lifecycle	85	100
Mo Clock	78	100
Volume Control	82	100

Таблиця 3.4.

Покриття методів (%)

Застосунок	Monkey (%)	Дослідницькі Тести (%)
Activity Lifecycle	72	95
Mo Clock	68	92
Volume Control	75	98

Порівняння ефективності показало, що систематично згенеровані дослідницькі тести досягають максимального покриття, використовуючи значно меншу кількість вхідних даних порівняно з Monkey.

Таблиця 3.5.

Кількість вхідних даних для максимального покриття

Застосунок	Monkey (Макс. 500)	Дослідницькі Тести
Activity Lifecycle	500	120
Mo Clock	500	95
Volume Control	500	110

Були визначені наступні потенційні загрози валідності, які можуть вплинути на узагальнення та точність результатів цього дослідження:

- Використовувалися відносно невеликі та прості застосунки, що може не відображати складність реальних промислових застосунків.

- Інструменти статичного та динамічного аналізу, що застосовувалися, можуть мати внутрішні обмеження, які потенційно впливають на достовірність отриманих результатів.

- Вибірка з лише трьох застосунків може бути недостатньою для узагальнення висновків на ширший спектр Android-застосунків.

У цьому розділі було представлено фреймворк та досліджена ефективність комбінованого підходу, що інтегрує традиційний статичний аналіз із динамічним аналізом через ефективні систематичні дослідницькі

тести, для генерації моделі Android-застосунків. Експериментальні дані підтвердили, що динамічний аналіз здатен значно покращити початкову модель застосунку. Крім того, систематичні дослідницькі тести перевершують випадкові тести Monkey як за показниками покриття UI та методів, так і за ефективністю, досягаючи максимального покриття з використанням меншої кількості вхідних даних.

Висновки до розділу

У третьому розділі розроблено та реалізовано алгоритми генерації послідовностей вхідних даних на основі комбінованого підходу до аналізу мобільних застосунків. Поєднання статичного та динамічного аналізу дозволило підвищити повноту покриття тестів і точність моделі поведінки програм. Експериментальне дослідження підтвердило ефективність запропонованого фреймворку та практичну цінність отриманих результатів.

ВИСНОВКИ

У магістерській роботі виконано комплексне дослідження методів і засобів побудови автоматизованих модель-базованих інтерфейсів генерації тестів для мобільних застосунків, що дало змогу систематизувати наявні підходи та розробити нові рішення, спрямовані на підвищення ефективності процесів тестування програмного забезпечення.

На основі проведеного аналізу сучасних тенденцій у сфері мобільного тестування доведено, що проблема забезпечення високої надійності мобільних застосунків має не лише технічний, а й соціально-економічний вимір. Якість програмного забезпечення безпосередньо впливає на рівень довіри користувачів, бізнес-результати компаній та безпеку обробки персональних даних. Водночас традиційні підходи ручного та частково автоматизованого тестування не відповідають сучасним вимогам до масштабованості, адекватності та швидкості виявлення дефектів.

У роботі було встановлено, що модель-базовані методи тестування дозволяють формалізувати поведінку програмних систем, будувати узагальнені графові подання потоку управління мобільним застосунком та на їх основі генерувати тестові сценарії з підвищеною точністю. Особливу увагу приділено розробці розширеного графа управління, який враховує специфіку життєвого циклу Android-активностей та взаємозв’язки між елементами користувацького інтерфейсу.

У роботі розроблено і реалізовано алгоритми генерації послідовностей вхідних даних на основі комбінованого використання статичного та динамічного аналізу. Такий підхід дозволив подолати обмеження окремих методів і забезпечити більш повне охоплення функціональності застосунків. Інтеграція статичного аналізу (на базі інструментів Soot, FlowDroid, AndroGuard) з динамічним моніторингом дала змогу створити модель, здатну враховувати як структуру коду, так і реальну поведінку програми під час виконання.

Розроблений метод автоматизованої генерації вхідних даних довів свою ефективність у рамках експериментального дослідження фреймворку: результати показали зменшення часу формування тестів, підвищення показників покриття та виявлення дефектів, які залишалися поза увагою при традиційних підходах. Це підтверджує доцільність інтеграції модель-базованих інтерфейсів генерації тестів у сучасні системи забезпечення якості мобільного програмного забезпечення.

Проведене дослідження дало змогу сформулювати такі узагальнені результати:

- Систематизовано існуючі методи автоматизації тестування, серед яких виділено випадкову, систематичну, пошукову та модель-базовану генерацію; доведено переваги останньої у контексті підвищення адекватності тестів.

- Розроблено модель розширеного графа потоку управління мобільним застосунком, яка враховує специфіку UI та життєвого циклу Android-компонентів, що забезпечує точнішу побудову тестових сценаріїв.

- Запропоновано алгоритми генерації тестових послідовностей, засновані на комбінованому використанні статичного й динамічного аналізу, що дало змогу збільшити повноту охоплення функціональності та масштабованість тестування.

- Доведено практичну значущість впровадження модель-базованих інтерфейсів у системи тестування мобільних застосунків, що забезпечує скорочення часу перевірки, зменшення витрат ресурсів та підвищення надійності кінцевих продуктів.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Myers G., Sandler C., Badgett T. The Art of Software Testing. 3rd ed. — Wiley, 2011.
2. Ammann P., Offutt J. Introduction to Software Testing. — Cambridge University Press, 2008.
3. Utting M., Legeard B. Practical Model-Based Testing: A Tools Approach. — Morgan Kaufmann, 2007.
4. Beizer B. Software Testing Techniques. 2nd ed. — Van Nostrand Reinhold, 2009.
5. Kaner C., Falk J., Nguyen H. Testing Computer Software. 2nd ed. — Wiley, 1999.
6. Bertolino A. “Software Testing Research: Achievements, Challenges, Dreams,” Future of Software Engineering (FOSE), 2007.
7. Binder R. Testing Object-Oriented Systems: Models, Patterns, and Tools. — Addison-Wesley, 1999.
8. Whittaker J. What Is Software Testing? And Why Is It So Hard? — IEEE Computer Society Press, 2000.
9. Vacca J. Computer and Information Security Handbook. — Academic Press, 2013 .
10. Pressman R. Software Engineering: A Practitioner’s Approach. 8th ed. — McGraw-Hill, 2014.
11. Create UI tests with Espresso Test Recorder | Android Developers - https://developer.android.com/studio/test/other-testing-tools/espresso-test-recorder
12. Appium Architecture, Explained | Waldo Blog - https://www.waldo.com/blog/appium-architecture
13. Arzt S., Rasthofer S., Fritz C. et al. “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps.” PLDI 2014.

14. Karlsson S., Čaušević A., Sundmark D., Larsson M. “Model-based Automated Testing of Mobile Applications: An Industrial Case Study.”
15. Salihu I.-A., Ibrahim R., Ahmed B. S., Zamli K. Z., Usman A. “AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing.”
16. Li Y., Yang Z., Guo Y., Chen X. “Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing.” arXiv, 2019.
17. 15. Gudmundsson V., Lindvall M., Aceto L., Bergthorsson J., Ganesan D. “Model-based Testing of Mobile Systems — An Empirical Study on QuizUp Android App.” 2016
18. Doyle J. “Automated Model-based Interface Test Generation for Mobile Applications.” PhD Thesis, University College Dublin, 2024. researchrepository.ucd.ie
19. Alhaddad A., et al. “FSMApp: Testing Mobile Apps.” ScienceDirect, 2023. ScienceDirect
20. Xu W., Cheng J. “A Model-Based Approach to Mobile Application Testing.” International Journal of Advanced Network Monitoring and Controls, 2023.
21. “Model-based Testing for a Family of Mobile Applications.” SPLC 2023 (Fischer et al.). wesleyklewerton.github.io
22. “Open Source Model-Based Testing Tools.” Software Testing Magazine, 2022. Software Testing Magazine
23. “Scalable and Precise Taint Analysis for Android — DroidInfer.” RPI Computer Science. cs.rpi.edu
24. “Android taint flow analysis for app sets.” ACM Digital Library (Lim et al.). dl.acm.org
25. “Static Taint Analysis Tools to Detect Information Flows.” (Boxler et al.). faculty.uccs.edu
26. “Precise Static Analysis of Taint Flow for Android Application Sets.” Amar S. Bhosale thesis / SEI report. sei.cmu.edu

27. “B-droid: A Static Taint Analysis Framework for Android Applications.” he Scientific World Journal (Arzt et al.). thesai.org
28. “How does Model-Based Testing improve Test Automation?” BrowserStack guide, 2025.
29. Ramler R., Kopetzy T., Platz W. “Value-Based Coverage Measurement in Requirements-Based Testing,” Euromicro Conference on Software Engineering and Advanced Applications, 2012
30. Bringmann E., Krämer A. “Model-Based Testing of Automotive Systems,” в Informatik – Forschung und Entwicklung, 2001