Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів використання скінченних автоматів для оптимізації та підвищення ефективності кросплатформених мобільних додатків
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-4

Владислав МАРТИНОВ
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник проф. Ігор ШУБІН
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри ____________ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________
(підпис)
«___» _________ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Мартинову Владиславу Романовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів використання скінченних автоматів для оптимізації та підвищення ефективності кросплатформених мобільних додатків»

Затверджена наказом по університету від 15.04. 2025р. № 290 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 13.06.2025

3. Вихідні дані до роботи опис досліджуваних FSM, специфікації бібліотек Redux-Toolkit, XState, Mobx, Lottie, Rive, дані результатів тестування продуктивності скінченних автоматів при використанні в кросплатформених мобільних застосунках, фреймворк React-Native.

4. Перелік питань, що потрібно опрацювати в роботі

Аналіз та порівняння існуючих FSM, вибір бібліотек необхідних для дослідження, проєктування логічної моделі даних для проведення експериментальних досліджень, написання програмних рішень, проведення експериментів та аналіз отриманих результатів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	16.04 – 18.04.25	*виконано*
2	Виявлення проблематики теми	18.04 – 20.04.25	*виконано*
3	Аналіз та моделювання предметної області, розробка теоретичної частини	20.04 – 23.04.25	*виконано*
4	Планування експериментальної частини	23.04 – 25.04.25	*виконано*
5	Програмна реалізація проєкту	25.04 – 30.04.25	*виконано*
6	Експериментальні дослідження	01.05 – 05.05.25	*виконано*
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	06.05 – 09.05.25	*виконано*
8	Написання та оформлення статті та тез доповіді	13.05 – 15.05.25	*виконано*
9	Підготовка пояснювальної записки	10.05 – 28.05.25	*виконано*
10	Підготовка презентації та доповіді	29.05 – 10.06.25	*виконано*
11	Нормоконтроль	11.06 – 12.06.25	*виконано*
12	Рецензування	13.06 – 15.06.25	*виконано*
13	Занесення диплома в електронний архів	16.06 – 17.06.25	*виконано*
14	Попередній захист	18.06.2025	*виконано*
15	Допуск до захисту у зав. кафедри	19.06.2025	*виконано*

Дата видачі завдання 16 квітня 2025р.

Студент ________________ Владислав МАРТИНОВ
(підпис)

Керівник роботи ________________ проф. Ігор ШУБІН
(підпис) (посада, Власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 88 с., 15 рис., 5 табл., 17 джерел, 5 додатків.

КРОСПЛАТФОРМЕНА РОЗРОБКА, МОБІЛЬНИЙ ДОДАТОК, ОПТИМІЗАЦІЯ, ПРОДУКТИВНІСТЬ, СКІНЧЕННИЙ АВТОМАТ.

Об’єктом дослідження є скінченні автомати (Finite State Machines, FSM) для управління станами у кросплатформених мобільних додатках.

Метою роботи є розробка, тестування кросплатформених мобільних додатків, розроблених з використанням декларативного підходу та скінченних автоматів на новій та старій архітектурі у фреймворку React Native. Порівняння ефективності управління станами в мобільних застосунках та інтеграція кінцевих автоматів у розробку кросплатформених мобільних додатках на підставі оцінки їхньої ефективності.

Методами розробки та проєктування є використання фреймворку React Native (стара/нова архітектура), мов програмування Typescript, C++, Javascript, розробленого FSM для мобільних платформ Android/iOS з використанням бібліотеки Legend-State; оцінка процесів управління станами кросплатформних мобільних додатків.

У результаті роботи проаналізовані літературні та наукові джерела інформації, було створено FSM, проведене тестування кросплатформених додатків, обгрунтовано доцільність інтеграції FSM в розробку мобільних застосунків.

FINITE STATE MACHINE, CROSS-PLATFORM DEVELOPMENT, MOBILE APPLICATION, OPTIMIZATION, EFFICIENCY.

The object of the research is Finite State Machines (FSM) for state management in cross-platform mobile applications.

The purpose of the work is to develop and test cross-platform mobile applications

developed using a declarative approach and finite state machines on new and old architectures in the React Native framework. Comparison of the effectiveness of state management in mobile applications and integration of state machines in the development of cross-platform mobile applications based on an assessment of their effectiveness.

Development and design methods include the use of the React Native framework (old/new architecture), programming languages :Typescript, C++, Javascript, developed by FSM for Android/iOS mobile platforms using the Legend-State library; evaluation of state management processes of cross-platform mobile applications.

As a result of the work, literary and scientific sources of information were analyzed, an FSM was created, cross-platform applications were tested, and the feasibility of integrating FSM into the development of mobile applications was substantiated.

Завідувачу кафедри
ПІ___________________
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ_
(вчене звання, власне ім’я, прізвище)

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації
(та/або публікації анотації кваліфікаційної роботи) в електронному архіві
відкритого доступу ElAr KhNURE

Я, Мартинов Владислав Романович, студент гр. ІПЗм-23-4, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота на тему «Дослідження методів використання скінченних автоматів для оптимізації та підвищення ефективності кросплатформених мобільних додатків», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата Підпис

ЗМІСТ

Перелік скорочень 9
Вступ 10
1 Аналіз предметної галузі 12
1.1 Аналіз предметної галузі дослідження 12
1.2 Огляд існуючих методів управління станами кросплатформених мобільних додатків 13
1.3 Проблематика предметної галузі 16
2.Огляд й аналіз літературних, наукових джерел 19
2.1 Огляд літературних джерел 19
2.2 Аналіз літератури 28
2.3 Оцінка актуальності та новизни 30
2.4 Висновки з огляду літературних джерел 31
3 Постановка задачі 33
4 Теоретичне дослідження 34
4.1 Аналіз показників ефективності та вибір оптимальної архітектури скінченного автомату 34
5 Програмна реалізація 44
5.1 Вибір інструментальних засобів для розробки скінченого автомату FSM. 44
5.2 Базова архітектура react-native (old architecture) 47
5.3 Недоліки старої архітектури react native 50
5.4 Нова архітектура react-native (new architecture) 51
6 Експерементальне дослідження 55
6.1 Вихідні дані 55
6.2 Реалізація декларативного підходу 56
6.3 Реалізація управління станом за допомогою кінцевого автомата (FSM) та бібліотеки Legend-State 60
7 Аналіз результатів 62
7.1 Результати ефективності роботи мобільного застосунку з використанням декларативного підходу в React Native 62

7.2 Результати продуктивності мобільного застосунку з використанням FSM у React Native ... 64
Висновки ... 67
Перелік джерел посилання ... 69
Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії ... 71
Додаток А Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ ... 72
Додаток Б Слайди презентації ... 74
Додаток В Апробація у вигляді статті на 13-й Міжнародній науково-технічній конференції Інформаційні системи та технології ІСТ-2024 ... 81
Додаток Г Апробація у вигляді тез на XXVIII Міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті» ... 85
Додаток Д Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015 ... 88

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface

CPU – Central Prossecing Unit

DFA – Deterministic Finite Automata

FPGA – Field-Programmable Gate Array

FSM – Finite State Machines

FSMIM – Finite State Machines with Input Multiplexing

LUT– Look-Up-Table

NFA – Non-Deterministic Finite Automata

RAM – Random Access Memory

ROM – Read Only Memory

UI – User Interface

UX – User Experience

ВСТУП

У сучасному світі, зі зростанням використання мобільних пристроїв, важливість створення кросплатформених додатків, що забезпечують високу продуктивність та якість користувацького досвіду, значно зросла. Удосконалення створення таких додатків з використанням React та React Native відкриває широкі можливості для розробників завдяки можливості перевикористання коду, високій швидкості розробки та легкості впровадження інновацій і набуває особливої актуальності, оскільки вона стосується не тільки технологічних аспектів розробки, але й економічної ефективності, якості користувацького досвіду та швидкості впровадження інновацій на ринок програмного забезпечення. Відкритий код та активне спілкування у спільноті дозволяють швидко виявляти та усувати проблеми, розробляти нові функції та покращення, а також сприяють обміну знаннями та досвідом між розробниками. Це створює позитивне середовище для інновацій та росту, сприяє швидкому прогресу технологій та підвищує якість кінцевих продуктів [1].

Скінченні автомати (FSM) є одним із найперспективніших інструментів для управління станами у мобільних додатках. Вони забезпечують чітку структурованість, оптимізують складні сценарії роботи додатків, підвищують продуктивність та стабільність роботи, мінімізують ризики помилок. До того ж FSM спрощують розробку програмного продукту, завдяки використанню формального підходу до управління переходами між станами, що в свою чергу, робить програмний код більш зрозумілим і підтримуваним. Застосування FSM є особливо корисним для вирішення практичних задач, пов’язаних із організацією навігації, авторизацією користувачів, обробкою помилок, а також при управлінні взаємодією з мережею та внутрішніми API пристроями. Завдяки цьому скінченні автомати стають незамінним інструментом у кросплатформеній розробці, де додатки повинні відповідати високим стандартам ефективності та адаптивності до різних операційних систем.

Актуальність теми зумовлена динамічним розвитком технологій та необхідністю забезпечення якісного доступу до даних на різних пристроях, незважаючи на відмінність їхніх операційних систем, впровадженням сучасних методів управління станами мобільних додатків, які задовольняють потреби користувачів і забезпечують їх конкурентоспроможність на ринку.

Метою роботи є вивчення сучасних підходів до управління станами в мобільних додатках та інтеграція кінцевих автоматів для оптимізації управління станами кросплатформених мобільних додатків з метою підвищення їх продуктивності, стабільності та зручності підтримки.

Для досягнення мети було поставлено наступні задачі:

– провести аналіз предметної області та визначити ключові виклики у кросплатформеній розробці;
– дослідити існуючі методи управління станами, зокрема Redux Toolkit, MobX та Xstate.

Об’єктом дослідження є процеси управління станами у кросплатформених мобільних додатках.

Предметом дослідження є використання кінцевих автоматів для оптимізації управління станами у мобільних додатках.

Методи дослідження включають аналіз предметної області, аналіз існуючих методів управління станами кросплатформених мобільних додатків.

Одержані результати підтвердили ефективність впровадження кінцевих автоматів для оптимізації логіки управління станами мобільних кросплатформених додатків.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі дослідження

Мобільні додатки стали невід’ємною частиною повсякденного життя. Згідно з даними Eastern Peak, 87% користувачів смартфонів проводять значну частину часу у мобільних застосунках. У 2022 році кількість завантажень мобільних додатків перевищила 200 мільярдів, і ця цифра продовжує зростати. Частка мобільного трафіку також демонструє значний ріст: у 2022 році вона становила понад 60% порівняно з 10% у 2011 році. Очікується, що у 2024 році дохід від цього ринку сягне 420 мільярдів доларів, а до 2026 року прогнозується його збільшення до 542 мільярдів доларів [2].

У сучасній мобільній розробці значну увагу приділяють оптимізації логіки роботи додатків. Розробники стикаються з необхідністю забезпечення стабільності, адаптивності та високої продуктивності додатків, які повинні однаково ефективно працювати на різних операційних системах, таких як iOS та Android. З появою штучного інтелекту (ШІ) вимоги до мобільних додатків суттєво зросли, адже сучасні користувачі очікують не лише функціональності, але й розумного, контекстуального реагування програмного забезпечення на їх запити. Водночас технічні вимоги до програмних продуктів постійно ускладнюється, що створює додаткові виклики для розробників. Цей факт спонукає до вивчення та застосування в програмуванні нових підходів, які є основою ефективного функціонування сучасних мобільних додатків.

Кросплатформені мобільні додатки займають особливе місце в програмній інженерії, оскільки вони дозволяють зменшити фінансові витрати та час на розробку; спрощують підтримку та забезпечують гнучкість при додаванні нового функціоналу продукту. Використання таких інструментів, як React-native, дає можливість створювати додатки, які однаково добре працюють на кількох операційних системах, за рахунок використання єдиної кодової бази для різних платформ.

Але ефективне управління станами кросплатформених мобільних додатків залишається важливим викликом у процесі їх розробки.

Використання FSM допомагає чітко задати та визначати кінцевий набір станів додатку, координувати та узгоджувати процес переходу між станами, навіть за умови високої складності архітектури. скінченні автомати забезпечують модульність та масштабованість розробки, що є визначальним фактором для подальшого розвитку конкретного кросплатформеного мобільного додатку.

Розподіл функціоналу між окремими станами дозволяє легко вносити зміни в логіку, без ризику порушення роботи інших компонентів. Це забезпечує стабільність роботи навіть при складних сценаріях виконання.

1.2 Огляд існуючих методів управління станами кросплатформених мобільних додатків

Існує декілька бібліотек для управління станом кростплатформних мобільних додатків, кожна з яких має свої особливості, переваги та недоліки.

Redux Toolkit (RTK) - це офіційна бібліотека для роботи з Redux, яка була створена для подолання ключових недоліків класичного Redux. Вона вирішує проблеми надмірної кількості шаблонного коду, складності конфігурації глобального стану та низької продуктивності у додатках зі складною архітектурою. RTK - є універсальним інструментом, що спрощує створення редюсерів, дій, асинхронних запитів та налаштування проміжних шарів, об’єднуючи ці процеси в одному інтуїтивно зрозумілому API. Це робить бібліотеку ідеальним вибором як для новачків, так і для досвідчених розробників.

Метод createSlice інтегрує в собі визначення початкового стану, дій та редюсерів в одному місці, що значно спрощує підтримку та масштабування коду. Розробники більше не потребують створювати окремі файли для кожної дії чи редюсера, оскільки вся логіка управління станом описується у компактному і зрозумілому форматі. Це також полегшує роботу розробників, адже код стає більш структурованим і простим у читанні.

Redux Toolkit має метод для обробки асинхронних операцій - createAsyncThunk. Дана функція автоматично генерує стани очікування, успіху та помилки для кожного запиту, дозволяючи легко управляти асинхронною логікою

додатку. Це робить RTK особливо корисним для проєктів, які активно використовують API-запити чи мають складну архітектуру.

Попри свої переваги, Redux Toolkit може виявитися занадто складним для проєктів з простою архитектурою кінцевого продукту, де управління станом є простим і не потребує складної логіки. Проте для середніх і великих додатків RTK є ідеальним інструментом, який поєднує масштабованість із простотою використання. Його підтримка складних сценаріїв, таких як обробка асинхронних запитів та централізоване управління станом, робить його універсальним рішенням для багатьох проєктів.

Redux Toolkit є стандартом у світі Redux, адже його методи дозволяють розробникам зосередитися на бізнес-логіці, замість того щоб витрачати час на налаштування та вирішення дрібних технічних проблем. Завдяки активній підтримці спільноти та інтеграції з сучасними технологіями, RTK продовжує залишатися пріоритетним вибором для розробників [3].

MobX є простою та масштабованою бібліотекою для управління глобальним станом у JavaScript-додатках. Основою MobX є три концепції: спостережувані об’єкти (observables), які представляють стан додатка, дії (actions), що відповідають за зміну стану, та реакції (reactions), які автоматично синхронізують зміни стану з інтерфейсом користувача.

Її головною особливістю є автоматичне відстеження та оновлення залежностей між видами станів і компонентами, що забезпечує синхронізацію інтерфейсу користувача з даними. Це робить MobX зручним інструментом для побудови реактивних додатків, де інтерфейс автоматично реагує на зміни стану. Завдяки цьому розробники можуть реалізовувати бізнес-логіку, а не витрачати час на налаштування складних механізмів оновлення UI.

MobX ефективно управляє складними сценаріями, гарантуючи, що лише необхідні компоненти будуть оновлюватися під час зміни стану. Це значно підвищує продуктивність додатків, особливо тих, що працюють із великими обсягами даних.

Однією з переваг MobX є простота у використанні. API бібліотеки інтуїтивно зрозумілий і легко освоюється навіть новачками. Крім того, MobX дозволяє уникнути превалювання шаблонного коду, що значно спрощує розробку; забезпечує гнучкість у виборі підходу до організації коду, підтримуючи як об’єктно-орієнтований, так і функціональний стилі програмування.

Однак MobX має й певні обмеження. Надмірне використання спостережуваних об’єктів, що призводить до ускладнення коду та зниження його читабельності. Крім того, автоматичні оновлення можуть ускладнювати процес узгодження даних в додатках із великою кількістю залежностей. Це стає проблемою у великих проєктах, де важливо чітко відстежувати взаємозв’язки між компонентами та станом. Ще одним недоліком є невелика кількість користувачів і недостатня підтримка бібліотеки, що може обмежувати доступність до розв’язання проблем.

MobX ідеально підходить для невеликих і середніх проєктів, де потрібна простота та ефективність. Її реактивне середовище дозволяє швидко створювати інтерактивні інтерфейси користувача з мінімальними зусиллями. У великих проєктах із комплексними вимогами до управління станами використання MobX може бути менш виправдана через потенційні труднощі з підтримкою та масштабуванням [4].

XState - це бібліотека для JavaScript та TypeScript, яка реалізує скінченні автомати та діаграми станів, спрощуючи управління складною логікою додатків. Основною перевагою XState є її здатність моделювати поведінку додатків у вигляді кінцевих автоматів, що забезпечує передбачувану, надійну та добре структуровану логіку роботи додатків. Бібліотека використовує концепцію подієво-орієнтованого програмування та акторну модель для роботи зі станами, що дозволяє організувати взаємодію між різними частинами додатка за допомогою повідомлень.

Одним із ключових аспектів XState є її здатність чітко розділяти стани та переходи між ними. Наприклад, XState дозволяє моделювати такі сценарії, як обробка асинхронних запитів, управління помилками чи складна послідовність

дій у користувацькому інтерфейсі. Завдяки вбудованим інструментам бібліотека підтримує як синхронні, так і асинхронні переходи між станами, що робить її універсальним інструментом для широкого спектра застосувань.

XState також надає можливість створювати візуальні діаграми станів, які є корисними для документування логіки додатка та її перевірки. Візуалізація діаграм спрощує розуміння навіть найскладніших сценаріїв, полегшуючи спільну роботу над проєктами та забезпечуючи передбачувану поведінку додатка. Крім того, XState підтримує інтеграцію з такими інструментами, як Stately Visualizer, який дозволяє тестувати та налагоджувати скінченні автомати в інтерактивному середовищі.

Однак, XState має певні обмеження. Для розробників, які незнайомі з концепцією кінцевих автоматів або акторної моделі, тобто мають низький поріг входження, час навчання, а відповідно і застосування бібліотеки, збільшується. Навіть базове використання XState вимагає розуміння основних принципів управління станами, таких як події, транзакції та ієрархія. Крім того, для невеликих або простих додатків використання XState є нераціональним, оскільки додає невиправдану складність проєкту.

XState часто використовується в розробці додатків, де необхідно забезпечити модульність, масштабованість та стабільність. Вона дозволяє організувати логіку додатка у вигляді чітко визначених модулів, які легко тестуються та масштабуються [5].

1.3 Проблематика предметної галузі

Кросплатформена розробка мобільних додатків є одним із найбільш перспективних напрямів у сучасній індустрії програмного забезпечення. Вона дає можливість створити додатки для кількох платформ. Проте, реалізація цієї концепції пов’язана з численними технічними та архітектурними викликами, серед яких ефективне управління станами додатків.

Однією з основних проблем є необхідність врахування специфіки кожної платформи під час розробки. Наприклад, у React Native компоненти інтерфейсу

можуть виглядати по-різному на Android і iOS через особливості системи рендерингу та стилізації. Крім того, інтеграція з нативними API, такими як камера, датчики геолокації або Bluetooth, часто потребує створення окремих рішень для кожної платформи. Це ускладнює розробку та збільшує час, необхідний для тестування та налаштування.

Ще однією складністю є різниця архітектури платформ. Наприклад, iOS використовує UIKit, тоді як Android базується на Views. Це означає, що деякі функціональні можливості можуть бути доступні лише на одній із платформ або реалізовані по-різному. Розробникам доводиться враховувати ці особливості, що додає складності при підтримці коду та впровадженні нових функцій.

Управління станами є ще одним критичним аспектом, який стає особливо складним у кросплатформених додатках. Взаємодія між різними екранами, обробка подій та підтримка асинхронних запитів вимагають чітко організованої структури станів. У традиційних підходах, таких як Redux або MobX, логіка станів може стати надмірно громіздкою, особливо у великих проєктах. Неправильно організовані стани можуть призводити до труднощів у підтримці, зростання кількості помилок та зниження продуктивності додатків.

Скінченні автомати (FSM) пропонують ефективний підхід до вирішення цих проблем. Вони дозволяють моделювати поведінку додатка, як набір станів із чітко визначеними переходами між ними, що значно спрощує обробку складних сценаріїв. Наприклад, використання FSM може спростити організацію таких процесів, як навігація між екранами, управління авторизацією або обробка асинхронних подій, зменшуючи кількість помилок і підвищуючи стабільність роботи додатка.

Тестування також є серйозною проблемою в ей розробці. Забезпечення використання набутого досвіду користувача на різних пристроях, з різними розмірами екранів і версіями операційних систем, потребує значних зусиль. Крім того, існує ризик появи помилок, пов’язаних із платформозалежною логікою або апаратними обмеженнями конкретних пристроїв.

Окрім того, розробка кросплатформених додатків потребує високого рівня компетенції від розробників. Вони повинні володіти не лише основними інструментами (наприклад, React Native), але й знаннями про архітектуру кожної платформи, особливості нативного коду та принципи оптимізації продуктивності. Впровадження FSM у цей процес може значно спростити структуру коду, дозволяючи розробникам зосередитися на бізнес-логіці, а не на вирішенні технічних проблем.

Таким чином, організація станів у таких додатках є одним із ключових технічних викликів, який потребує вдосконалених підходів. Використання кінцевих автоматів є перспективним рішенням, що дозволяє оптимізувати роботу додатків, підвищити їхню продуктивність і спростити процес підтримки.

2.ОГЛЯД Й АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Огляд літературних джерел

Традиційна архітектура для реалізації скінченних автоматів (FSM) у пам’яті ROM (Read Only Memory) представлена на рисунку 2.1(a), де p, m та n – це кількість бітів кодування станів, кількість входів та кількість виходів відповідно. Якщо припустити, що N_s – це кількість станів, то кількість бітів кодування розраховується за формулою $p = [\backslash log_{\{2\}x}]$.

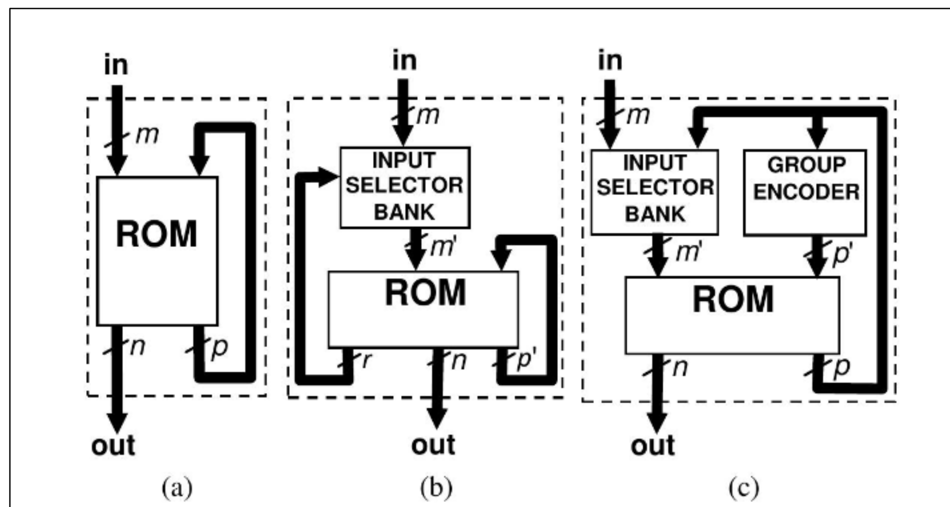


Рисунок 2.1 – Архітектури FSM на основі ROM: (a) традиційна, (b) FSMIM-T та (c) FSMIM-S (за даними [6])

Крім того, оскільки сучасні FPGA містять синхронні EMB, виходи FSM синхронізуються з годинником. Кожне слово ROM-блоку, реалізованого за допомогою EMB, містить наступний стан і виходи FSM для переходу. Таким чином, ширина ROM дорівнює n + p біт. Входи FSM і біти кодування поточного стану формують адреси ROM.

Таким чином, максимальна глибина ROM дорівнює 2^{m+p}слів, тобто максимальний розмір ROM експоненціально зростає з m і p. Якщо всі стани закодовані в інтервалі $[0, N_s)$, то глибина ROM дорівнює $2^{\{m\}N}_{\{s\}} \leq 2^{\{m+p\}}$слів.

Експоненціальне збільшення глибини ROM впливає не лише на площу, але й на швидкість. Час доступу до пам’яті погіршується, оскільки зростає кількість EMB, необхідних для реалізації ROM. Причини цього явища наступні: затримки

маршрутизації та затримки, спричинені LUT, що використовуються для з'єднання виходів EMB, коли глибина ROM перевищує максимальну глибину EMB.

Техніка FSMIM зменшує глибину пам'яті традиційних реалізацій FSM на основі ROM за рахунок зменшення як кількості входів FSM, що адресують ROM (m), так і кількості станів (N_s).Відповідне зменшення кількості необхідних EMB досягається за рахунок використання LUT. У архітектурах FSMIM (рисунок 1b, c) адресація ROM здійснюється за допомогою підмножини входів FSM, які впливають на її поведінку (вони називаються ефективними входами) для кожного стану. В обох архітектурах комбінаційна схема ISB вибирає ефективні входи поточного стану. ISB має m входів (входи FSM) та m' виходів (обрані входи); таким чином, глибина ROM може бути зменшена, якщо $m' < m$. Очевидно, значення m' визначається станом із найбільшою кількістю ефективних входів. Для стану $m < m'$ ефективними входами $m' - e$ входів, які не є ефективними, можуть вважатися сигналами "байдуже" (їх називають не вибраними входами) [6].

Послідовні схеми можна визначити як схеми, що складаються з секції комбінаторної логіки та іншої секції пам'яті, яка зазвичай представлена тригерами. Кожен етап, на якому послідовна схема переходить вперед, називається станом.

Кінцевий автомат (FSM) має скінченну кількість входів, які утворюють множину $N = \{N_1, N_2, \ldots, N_n\}$.

Таким чином, схема має скінченну кількість виходів, які визначаються множиною $M = \{M_1, M_2, \ldots, M_m\}$. Значення, що містяться у кожному елементі пам'яті, називаються змінними стану, і вони формують множину $K = \{K_1, K_2, \ldots, K_k\}$.

Значення, що містяться в елементах пам'яті K, визначають поточний стан машини. Внутрішні функції переходу генерують множину наступних станів $S = \{S_1, S_2, \ldots, S_s\}$, які залежать від входів N і поточних станів K машини і визначаються через комбінаторні схеми.

Значення S, які з'являються у функції переходу станів машини у момент часу t, визначають значення змінних стану у момент часу $t + 1$, а отже, і

наступний стан машини. Поведінку FSM можна описати за допомогою діаграми переходу станів або таблиці переходу станів. Діаграма переходу станів або таблиця переходу станів перераховує поточний стан, наступний стан, вхід і вихід.

Таблиця переходу станів має 2^N стовпців (по одному для кожного можливого випадку множини входів) і 2^K рядків (по одному для кожного можливого випадку множини станів). Діаграма переходів - це орієнтований граф, де кожен вузол представляє стан, а від кожного вузла виходять p орієнтованих дуг, що відповідають переходам станів. Кожна орієнтована дуга маркована входом, який визначає перехід, і виходом, який генерується. FSM визначає наступний стан $K(t+1)$, базуючись тільки на поточному стані $K(t)$ і поточному вході $N(t)$. FSM може бути представлений за допомогою рівняння 2.1:

$$K(t + 1) = f[K(t), N(t)] \tag{2.1}$$

де f – функція переходу станів.

Вихідне значення $M(t)$ визначається як рівняння 2.2, 2.3:

$$M(t + 1) = g[K(t)] \tag{2.2}$$

$$M(t + 1) = g[K(t), N(t)] \tag{2.3}$$

де g – вихідна функція.

Кінцевий автомат (FSM), властивості якого описані рівняннями (2.1) та (2.2), називається машиною Мура, а автомат, описаний рівняннями (2.1) та (2.3), називається машиною Мілі. Робота комп'ютерів базується на функціонуванні транзисторів, які, залежно від кількості накопиченого заряду, можуть інтерпретувати сигнал як високий (1), низький (0) або вимкнений (немає накопиченої енергії).

Оскільки комп'ютер працює з інтерпретацією двох електричних імпульсів, можна спостерігати, що це бінарна система, яка підпорядковується булевій алгебрі. Булева алгебра є алгебраїчною структурою, що визначає арифметику

логічних операторів. Ці оператори, що складаються із символів $S = \{0, 1\}$, утворюють бінарну систему. Концепції булевої алгебри також використовуються в електроніці, оскільки фізичні схеми зазвичай проєктуються в абстракціях, які називаються логічними схемами.

У Булевому просторі змінна є символом, який представляє координату в цьому просторі. Змінна або її заперечення називається літералом. Термін "добуток" визначається як булевий добуток одного або кількох літералів. Мінімальний терм або мінтерм – це добуток термів, який видає значення «1». Схема, яка включає всі змінні, у певних випадках може бути спрощена шляхом усунення надлишкових елементів, що зменшує її розмір. Булева функція, яка включає комбінацію мінітермів, називається імплікантом функції, а імплікант, який не може бути спрощений, тобто не містить іншої функції, називається основним імплікантом. Сума всіх імплікантів та основних імплікантів функції – це набір мінтермів, для яких результат функції дорівнює 1.

Для представлення FSM зазвичай використовуються слова або літери для позначення станів, оскільки кількість тригерів, необхідних для представлення FSM, обчислюється подібно до кількості рядків у таблиці істинності. При присвоєнні значення стану кожен літерал символізує значення, яке буде передано відповідному тригеру у певний момент часу. Об'єднання значень кожного тригера еквівалентне значенню, призначеному певному стану FSM.

Значення, що знаходяться в елементах пам'яті, у комбінації представляють поточний стан. Тригери підключені до комбінаторних схем, які змінюють значення, що містяться в тригері, на кожному імпульсі годинника, змушуючи тригери представляти значення, призначене наступному стану автомата, переходячи від поточного стану до наступного. Комбінаторна схема, відповідальна за цю зміну станів, є результатом спрощення виразів, отриманих із входів певного тригера та стимулів, які будуть подані. Схема отримує значення виходу тригера та значення стимулу стану машини. Набір вихідних значень представляє наступний стан, який прийме автомат.

Присвоєння станів є фундаментальним, коли є намір оптимізувати, оскільки воно безпосередньо пов'язане з розміром виразу, який здійснюватиме зміну між поточним станом і наступним станом. Зміна розподілу значень значно впливає на розмір виразу, що, у свою чергу, збільшує розмір схеми [7].

Основні ідеї та варіації пояснюють фундаментальні концепції кінцевих автоматів (FSM), такі як стани, переходи, входи та виходи. Обговорюються багато варіантів FSM, включаючи детерміновані DFA та недетерміновані NFA автомати, а також машини Мура та Мілі. Початковий кінцевий автомат (FSM) є моделлю будь-якої системи, яка регулює її поведінку або роботу. Одним із застосувань є повністю автоматизована система керування пральною машиною. Користувач повністю автоматичної пральної машини повинен лише завантажити брудний одяг і миючий засіб у машину, натиснути кнопку старту, а машина зробить усе інше.

Комп'ютери на основі кінцевих автоматів можуть бути детермінованими або недетермінованими. Недетерміновані скінченні автомати та передбачувані скінченні автомати є двома основними видами кінцевих автоматів [8].

Мова вважається регулярною, якщо вона приймається детермінованим кінцевим автоматом, що є її основним призначенням. Скінченні автомати зазвичай вивчаються за допомогою заздалегідь визначених мов бінарних рядків. Бінарні рядки можуть бути використані для побудови як традиційних, так і нетрадиційних мов. Будь-який рядок, що починається з цифри 0, класифікується як мова бінарних рядків. До цієї мови належать такі рядки, як 001, 010, 0 і 01111; однак 111, 10000, 1 і 11001100 до неї не належать.

FSM класифікуються як детерміновані (також відомі як детерміновані скінченні автомати) або недетерміновані (також відомі як недетерміновані скінченні автомати Детерміновані скінченні автомати (DFA) мають лише один унікальний перехід із кожного стану для кожного символу входу, що забезпечує повністю передбачувану поведінку. Недетерміновані скінченні автомати (NFA) дозволяють кілька переходів із одного стану для одного і того ж символу входу, що призводить до менш передбачуваної поведінки.

Скінченні автомати (FSM) пропонують низку переваг, що робить їх корисними інструментами в різних галузях та застосуваннях. Ось основні переваги використання FSM:

- простота та інтуїтивність, скінченні автомати мають обмежену кількість станів і переходів, тому вони прості для вивчення, аналізу та реалізації;
- чіткість, FSM забезпечують наочне уявлення поведінки системи, що сприяє документації та комунікації складних процесів і логіки управління;
- низьке навантаження на обчислювальні ресурси, FSM ефективно використовують ресурси, вимагаючи менше пам’яті та потужності обробки, що робить їх ідеальними для систем із обмеженими ресурсами;
- детермінізм, для заданої послідовності входів і поточного стану детерміновані скінченні автомати (DFA) завжди видають однаковий результат, ця передбачуваність корисна для застосувань, де потрібні надійність і детермінізм;
- легкість налаштування, у випадку виникнення проблем у системах, заснованих на FSM, їх часто легко виявити, аналізуючи переходи між станами та вхідні дані;
- системи реального часу, завдяки швидкій реакції на входи та передбачуваним переходам між станами, FSM ідеально підходять для систем реального часу та застосувань управління;
- формальний аналіз, FSM можна формально перевіряти за допомогою математичних підходів, таких як аналіз досяжності станів і перевірка моделей, щоб підтвердити властивості системи та її правильність;
- модульність, FSM дозволяють розділяти складні системи на модульні компоненти, кожен з яких представлений власним FSM. Це спрощує проєктування та обслуговування системи;
- застосування в теорії обчислень, FSM є базовими у теорії обчислень і використовуються для розпізнавання регулярних мов, що робить їх

корисними у задачах парсингу та лексичного аналізу в мовах програмування та компіляторах [9].

При створенні інтерактивних мобільних додатків, часто використовують анімації, якими керує користувач. Наприклад, після замовлення товару з’являється анімація, яка підтверджує успішне оформлення покупки. Більш складний приклад - кроки онбордингу, де користувач повинен натискати навігаційні кнопки для вибору дії / переходу вперед, і в більшості випадків кожен крок супроводжується анімацією. Для реалізації задач по інтеграції анімації розробники використовують різні підходи.

Перший, застосування GIF-файлів. Кількість GIF-файлів, які використовуються, відповідає кількості кроків онбордингу. Зазвичай це рішення достатньо ефективне з точки зору продуктивності та користувацького досвіду. Але у випадку коли необхідно досягнути високої якості відображення GIF-файлів, одержуємо значне збільшити розміру додатку на кілька мегабайт, що є суттєвим недоліком даного підходу.

Другий, використання платформи LottieFiles. Яка надає вільний доступ до файлів анімації для мобільних платформ Android та iOS, який був створений розробниками з Airbnb для використання LottieFiles - JSON-формату анімації, експортованого за допомогою плагіна BodyMoving із Adobe AfterEffects

Альтернативою LottieFiles є платформа Rive, яка має веб-редактор, що дозволяє налаштовувати анімації в реальному часі. Цей редактор дає змогу користувачам створювати інтерактивні анімації, які можуть реагувати на введені дані. Наявність анімацій на основі тригерів є значною перевагою, особливо для мобільних платформ.

Отже, для реалізації анімації Rive, можна застосувати кінцевий автомат, що дасть можливість використовувати один файл анімації для всіх кроків онбордингу. Слід зазначити, що розмір файлу анімації, який використовується в даному випадку, значно менший у порівнянні з JSON-анімаціями Lottie. Крім того, не потрібно зберігати окремі файли для кожного кроку онбордингу, що також економить кілька кілобайт у розмірі додатку.

Розглянемо питання продуктивності в контексті FPS, використання CPU та споживання пам’яті. Проведемо порівняння анімації, створеної для Lottie, із аналогічною анімацією, реалізованою для Rive.

Вихідними даними є однакові файли анімації, розміщені на LottieFiles та Rive. Для порівняння використані ті ж самі файли анімації реалізовані в мобільному додатку на React Native.

Метою аналізу є оцінка продуктивності обох рішень у мобільному середовищі, щоб визначити, яке з них забезпечує більшу ефективність при збереженні якості та інтерактивності анімацій. Основними метриками для порівняння є стабільність кадрів (FPS), навантаження на центральний процесор та обсяг оперативної пам’яті, необхідний для виконання анімацій.

Отримані результати дали можливість зробити висновки щодо доцільності використання Rive або Lottie залежно від конкретних вимог до продуктивності та обмежень мобільних додатків (див. рис. 2.2 та 2.3).

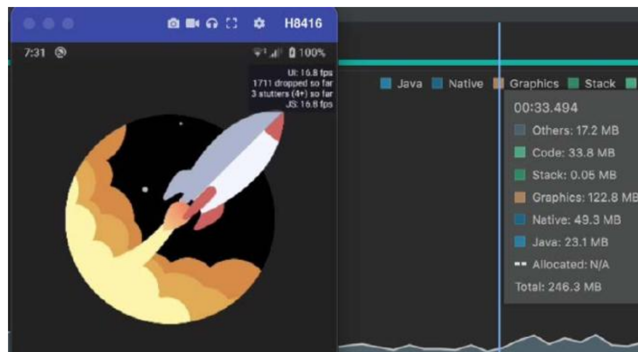


Рисунок 2.2 – Результат використання Lottie (за даними [10])

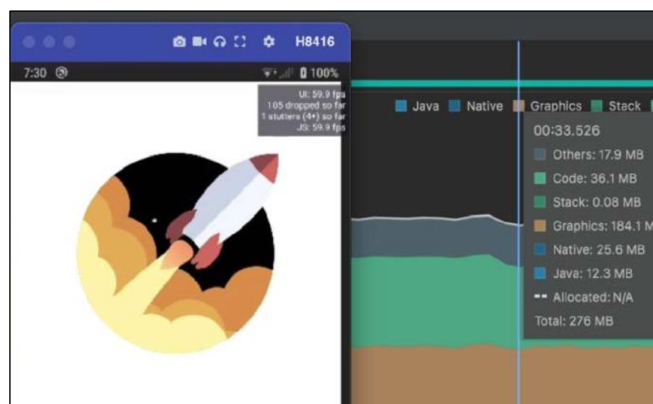


Рисунок 2.3 – Результат використання Rive (за даними [10])

Розглянемо дані, щодо використання оперативної пам’яті. Основні компоненти пам’яті представлені у трьох частинах: Java, Native та Graphics.

Java: ця частина відображає обсяг пам’яті, виділений для коду, написаного на Java або Kotlin. Lottie використовує приблизно 23 МБ оперативної пам’яті, тоді як Rive - близько 12 МБ.

Native: ця частина показує обсяг пам’яті, виділений для коду на C або C++. Lottie використовує 49 МБ оперативної пам’яті, а Rive - 25 МБ.

Graphics: ця частина представляє пам’ять, яка використовується для графічних буферних черг, необхідних для відображення пікселів на екрані. У цьому випадку Lottie споживає 123 МБ, тоді як Rive використовує 184 МБ оперативної пам’яті.

Загальне споживання пам’яті становить для Lottie: - 246 МБ, тоді як для Rive - 276 МБ.

Таким чином, хоча Rive перевершує Lottie в Java і Native, його графічне навантаження є більшим, що в підсумку призводить до трохи більшого загального споживання пам’яті. Це слід враховувати під час вибору інструмента для створення анімацій, особливо у додатках із суворими вимогами до обсягу доступної пам’яті.

Результати демонструють (див. рис. 2.4), що Rive перевершує Lottie у всіх аспектах, за винятком графіки.

	JAVA	NATIVE	GRAPHICS
MEMORY CONSUMPTION	JAVA or Kotlin Code	C or C++ code	pixels on the screen
LOTTIE	23 MB	49 MB	123 MB
RIVE	12 MB	25 MB	184 MB

Рисунок 2.4 – Результати експерименту (за даними [10])

Для кінцевого користувача важливо, щоб додаток працював із частотою 60 FPS, забезпечуючи плавний і комфортний фреймрейт екрану. При виборі між

споживанням пам’яті та частотою кадрів, більшість розробників можуть віддати перевагу FPS, оскільки більшість сучасних пристроїв мають достатній обсяг пам’яті для задоволення потреб додатку.

Таким чином, хоча Rive споживає більше пам’яті для графіки, його здатність підтримувати високу частоту кадрів робить його більш привабливим вибором у додатках, де якість і плавність анімацій є критичними для користувача.

При створенні додатку без використання кінцевих автоматів, розробникам доведеться реалізовувати логіку вручну у своєму коді. При кожній зміні інтерактивності анімації необхідно буде оновлювати попередній код.

Кінцевий автомат Rive дає можливість розробникам написати код , який при різноманітті вхідних даних, буде розширятися та використовуючи анімацію реалізувати інтерактивність.

У випадку необхідності зміни анімації, але з тими самими вхідними даними, розробнику достатньо лише замінити джерело анімації,що значно спрощує процес підтримки та адаптації інтерактивних анімацій, забезпечуючи ефективніший та гнучкіший робочий процес [10].

2.2 Аналіз літератури

Літературний аналіз демонструє значну увагу науковців і практиків до використання кінцевих автоматів (FSM), як інструменту для моделювання поведінки систем та управління їхніми станами. Основною перевагою FSM є формалізація складних логічних процесів, що забезпечує передбачуваність та стабільність роботи програмних рішень. Це робить FSM надзвичайно корисними для реалізації завдань із високими вимогами до надійності та ефективності.

Основні принципи функціонування FSM, включаючи поняття станів, переходів, входів і виходів. Визначено, що FSM є універсальною моделлю, яка може бути адаптована до різних типів завдань, таких як лексичний аналіз у мовах програмування, моделювання систем реального часу та керування робототехнічними пристроями. Також розглянуто особливості детермінованих (DFA) та недетермінованих (NFA) кінцевих автоматів, а також машин Мура та

Мілі, які пропонують різні підходи до організації виходів. Ці концепції є фундаментальними для розробки адаптивних FSM, які можуть враховувати специфіку різних платформ [11].

Традиційна архітектура FSM на основі пам'яті ROM, яка є ефективною у невеликих системах, демонструє обмеження у масштабованості. Зокрема, обсяг ROM експоненційно зростає зі збільшенням кількості входів і станів, що суттєво впливає на продуктивність і обчислювальні витрати. Запропонована техніка FSMIM дозволяє оптимізувати використання пам'яті за допомогою зменшення кількості ефективних входів, що адресують ROM. Використання LUT у цих архітектурах дає змогу зменшити затримки маршрутизації та підвищити загальну швидкість роботи FSM. Цей підхід є перспективним для реалізації FSM у системах із великим обсягом даних та обмеженими ресурсами, таких як мобільні платформи [12].

Підкреслюється важливість FSM для реалізації складних сценаріїв у реальних системах, таких як управління робототехнікою, автоматизоване керування транспортними засобами та інтерактивні користувацькі інтерфейси. Виявлено, що FSM дозволяють зменшити обчислювальні витрати за рахунок чіткого розмежування станів і переходів, що особливо важливо для систем із обмеженими ресурсами, наприклад, мобільних пристроїв. Крім того, формальна верифікація FSM забезпечує високу точність і передбачуваність роботи системи [9].

Розглянуто проблему оптимального призначення станів у FSM. Визначено, що коректне розташування станів дозволяє мінімізувати складність логічних схем, які забезпечують переходи між станами. Це досягається шляхом зменшення розміру комбінаторних схем і оптимізації використання тригерів, що має прямий вплив на продуктивність системи. Також підкреслено, що ефективне призначення станів є важливим фактором для масштабованості та стабільності програмних рішень, зокрема у кросплатформених додатках [13].

Закцентовано увагу на практичному використанні FSM у мобільних додатках, зокрема для управління інтерактивними анімаціями. FSM забезпечують

можливість створення компактних і високопродуктивних рішень, які дозволяють розробникам зосередитися на бізнес-логіці замість технічних деталей. Порівняння двох популярних бібліотек анімацій - Lottie та Rive - демонструє переваги FSM у підвищенні інтерактивності та зменшенні обсягу пам'яті, необхідної для зберігання анімацій. Використання FSM у цьому контексті також спрощує процес оновлення додатків, забезпечуючи більшу гнучкість для дизайнерів і розробників [10].

2.3 Оцінка актуальності та новизни

Аналіз літературних джерел підтверджує, що скінченні автомати (FSM) є ефективним інструментом для моделювання складних логічних процесів та управління станами кросплатформених мобільних додатків. Вони активно використовуються для задач автоматизації, управління реального часу, розпізнавання мов та аналізу даних. Проте, незважаючи на широке застосування FSM, більшість досліджень зосереджуються на їх використанні у вузьких контекстах або на оптимізації архітектур для спеціалізованих систем, таких як FPGA або робототехніка. Лише невелика частка наукових робіт звертається до інтеграції FSM у процеси мобільної розробки, зокрема у кросплатформені додатки.

Одним із важливих висновків літературного огляду є те, що сучасні дослідження обмежуються теоретичними або технічними аспектами FSM, такими як оптимізація пам'яті, розробка ефективних алгоритмів переходів між станами або використання FSM у вузькоспеціалізованих системах.

Щодо кросплатформеної мобільної розробки, література надає обмежені рішення. Хоча мобільні додатки залишаються одним із найбільш динамічних напрямів розробки програмного забезпечення, управління станами у цих додатках зазвичай реалізується з використанням традиційних підходів, таких як Redux або MobX. Проте ці бібліотеки мають обмежену підтримку складних сценаріїв з великою кількістю станів і переходів, що може ускладнювати масштабованість і продуктивність коду. Застосування FSM для розробки кросплатформених

мобільних додатків не отримало достатньої уваги у наукових дослідженнях, незважаючи на їхній потенціал.

Новизна дослідження полягає у поєднанні методології FSM із сучасними інструментами кросплатформеної розробки, такими як React Native. У той час, як більшість наукових робіт акцентує увагу на однобічних аспектах використання FSM, ця робота спрямована на інтеграцію кінцевих автоматів у повний цикл розробки мобільних додатків, зосереджуючись на оптимізації логіки, підвищенні продуктивності та забезпеченні гнучкості коду. Крім того, це дослідження має на меті адаптувати FSM для специфічних вимог кросплатформених середовищ, включаючи інтерактивність інтерфейсів, асинхронну роботу та підтримку різних операційних систем.

Актуальність цієї роботи зумовлена зростаючою популярністю кросплатформеної розробки. Вона дозволяє значно скоротити витрати часу та ресурсів за рахунок використання єдиної кодової бази, але водночас створює низку викликів, серед яких управління станами. Застосування FSM у цьому контексті дозволяє вирішити ключові проблеми, зокрема уніфікацію логіки, забезпечення модульності та зменшення кількості помилок. Результати цього дослідження можуть стати основою для розробки нових методів і підходів, які підвищать ефективність створення мобільних додатків.

Таким чином, дане дослідження має як теоретичну, так і практичну цінність. Теоретичний аспект полягає у розробці нових підходів до інтеграції FSM у мобільну розробку, тоді як практичний акцент спрямований на створення конкретних рекомендацій для розробників щодо використання FSM у реальних проєктах.

2.4 Висновки з огляду літературних джерел

На основі аналізу літературних джерел було встановлено, що скінченні автомати (FSM) є ефективним інструментом для моделювання поведінки складних систем. Вони забезпечують формалізований підхід до управління станами, що дозволяє підвищити передбачуваність, стабільність та ефективність

програмних систем. Однак, більшість наукових робіт і досліджень зосереджуються на використанні FSM у вузькоспеціалізованих галузях, таких як робототехніка, апаратне забезпечення та теорія обчислень, і недостатньо уваги приділяють інтеграції FSM у кросплатформену розробку мобільних додатків.

Проведений огляд та аналіз літературних джерел демонструють, що сучасні дослідження здебільшого спрямовані на оптимізацію архітектур FSM, включаючи зменшення апаратних витрат, підвищення продуктивності пам’яті та вдосконалення алгоритмів переходів між станами. Водночас, питання інтеграції FSM у процеси розробки кросплатформенних мобільних додатків залишаються майже недослідженими.

Зокрема, обмежено розглядається використання FSM для вирішення завдань, пов’язаних із складною логікою навігації, обробкою асинхронних подій або підтримкою інтерактивних інтерфейсів користувача.

Аналіз показав, що традиційні підходи до управління станами, такі як Redux або MobX, хоча і є ефективними, мають обмеження у контексті складних мобільних додатків. Вони часто не забезпечують достатньої модульності, масштабованості та передбачуваності, необхідних для ефективної роботи кросплатформених рішень. Використання FSM у цьому контексті може стати перспективним напрямом, що дозволить оптимізувати процеси управління станами.

Загалом, результати огляду свідчать про необхідність дослідження інтеграції FSM у кросплатформену мобільну розробку. Це відкриває можливості для створення нових методів оптимізації логіки управління станами, що матимуть як теоретичну, так і практичну цінність. Подальші дослідження в цій галузі сприятимуть покращенню якості мобільних додатків, підвищенню їхньої стабільності та забезпеченню зручності для кінцевих користувачів.

3 ПОСТАНОВКА ЗАДАЧІ

Основною метою квалфікаційної роботико є вивчення сучасних підходів до управління станами в мобільних додатках та інтеграція кінцевих автоматів у розробку кросплатформених мобільних додатках на підставі оцінки їхньої ефективності.

Для досягнення поставленої мети необхідно виконати наступні завдання:

– вивчити предметну галузь з обраної тематики та виявити основні виклики, пов’язані з управлінням станами у кросплатформених мобільних додатках;
– проаналізувати існуючі бібліотеки для управління станами, такі як Redux Toolkit, MobX, XState, та визначити їх переваги та обмеження;
– провести теоретичне дослідження показників ефективності використання скінченних автоматів при управлінні станами кросплатформених мобільних додатків;
– спроєктувати та реалізувати FSM для мобільного кросплатформеного додатку;
– провести експериментальне дослідження ефективності роботи FSM в тестовому мобільному додатку на різних моделях архітектур;
– проаналізувати результати експерименту та сформулювати рекомендації щодо ефективності інтеграції FSM в розробку кросплатформених мобільних додатків.

Одержані результати можуть бути використані в проєктуванні та розробці кросплатформених мобільних додатків.

4 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

4.1 Аналіз показників ефективності та вибір оптимальної архітектури скінченного автомату

Під час дослідження порівняємо різні методи використання скінченних автоматів для оптимізації та підвищення ефективності кросплатформених мобільних додатків. Це включатиме аналіз і порівняння підходів до побудови логіки переходів між станами, управління ресурсами та взаємодії з користувачем.

Для аналізу скористаємося лінійною адаптивною згорткою з ваговими коефіцієнтами. Вибір буде проводитись серед наступних методів, опис яких вказано нижче:

Використання простих скінченних автоматів з відкритою бібліотекою (Метод А).

Під час реалізації Методу А основний акцент робиться на використанні перевірених бібліотек, які вже мають великий набір функцій та добре документовані. Це дозволяє значно скоротити час на розробку, оскільки більшість задач, пов’язаних із побудовою логіки переходів між станами, вже вирішені. Наприклад, XState забезпечує підтримку вкладених автоматів, історичних станів і масштабованої логіки, що може бути корисним у проєктах з обмеженими ресурсами.

Однак, недоліком такого підходу є залежність від сторонніх рішень. У випадку, якщо функціонал бібліотеки обмежений або не відповідає специфічним вимогам проєкту, можливості налаштування можуть бути обмежені. Також, оновлення бібліотек або припинення їх підтримки розробниками може стати додатковим ризиком для довгострокового проєкту.

Метод А ідеально підходить для невеликих або середніх проєктів, де важливими є швидкий старт і мінімізація ризиків на етапі розробки. Наприклад, у мобільних додатках із простою логікою (таких як чати або калькулятори) цей метод може забезпечити високу продуктивність при низькій складності.

Інтеграція складних скінченних автоматів із власною оптимізацією (Метод B):

Метод B передбачає створення індивідуальних рішень, які дозволяють максимально адаптувати скінченні автомати до потреб конкретного проєкту. Цей підхід вимагає значних витрат часу і ресурсів, але забезпечує високий рівень контролю над усіма аспектами роботи автоматів. Наприклад, для додатків, що мають різну поведінку залежно від платформи (Android чи iOS), власні автомати можуть враховувати специфіку кожної платформи, забезпечуючи оптимальну продуктивність та стабільність.

Додаткова перевага такого підходу – можливість інтегрувати адаптивні алгоритми управління ресурсами. Наприклад, автомат може самостійно вирішувати, коли очищати кеш, зменшувати навантаження на процесор або переводити додаток у режим зниженої активності, залежно від доступних ресурсів пристрою. Це особливо актуально для додатків, які працюють на пристроях із обмеженими апаратними характеристиками.

Проте недоліком цього методу є його складність і необхідність у висококваліфікованих розробниках. Створення та налаштування таких автоматів потребує глибокого розуміння теорії скінченних автоматів, алгоритмів оптимізації та особливостей цільових платформ. Окрім того, розробка власних рішень може потребувати значного часу на тестування та усунення помилок.

Метод B найкраще підходить для великих і довготривалих проєктів, де висока продуктивність та унікальні вимоги є критично важливими. Наприклад, у додатках із високим навантаженням (таких як відеострімінгові платформи чи мобільні ігри), де важливо динамічно керувати ресурсами, цей метод може забезпечити конкурентну перевагу.

Використання скінченних автоматів із гібридними підходами (Метод C):

Метод C пропонує баланс між швидким впровадженням та гнучкістю. Використовуючи готові бібліотеки як основу, можна швидко реалізувати базову логіку переходів між станами. Одночасно додаючи власні оптимізації, можна адаптувати роботу автоматів до конкретних потреб проєкту. Наприклад, стандартні функції з XState можуть бути розширені додатковими обробниками станів, які враховують навантаження на пристрій або специфіку обробки даних.

Однією з ключових особливостей цього підходу є можливість використання lightweight - режимів. У ситуаціях із високим навантаженням, такими як обробка великого обсягу даних у реальному часі (наприклад, для фінансових або аналітичних додатків), гібридний підхід дозволяє перемикатися на спрощені сценарії роботи автоматів, зменшуючи навантаження на систему. Також цей метод підтримує стратегії відкладеного виконання задач (lazy execution), коли ресурсомісткі дії виконуються лише за необхідності.

Головною перевагою методу є його універсальність: він дозволяє швидко запустити проєкт завдяки використанню бібліотек, а згодом оптимізувати найважливіші аспекти для підвищення ефективності. Однак цей підхід також може мати недоліки, такі як необхідність ретельної інтеграції сторонніх бібліотек із власними рішеннями, що іноді може призводити до конфліктів або перевантаження коду.

Метод C найкраще підходить для середніх та великих проєктів, де є вимога швидкого запуску з можливістю подальшого вдосконалення. Наприклад, у проєктах із поступово зростаючими вимогами (такі як маркетплейси або навчальні платформи), цей метод дозволяє спочатку забезпечити базовий функціонал, а потім зосередитись на оптимізації критичних компонентів.

Використання скінченних автоматів для динамічного управління UI/UX (Метод D):

Метод D орієнтований на підвищення якості взаємодії користувача з додатком, використовуючи скінченні автомати для гнучкого управління інтерфейсом. Застосування цього підходу дозволяє створювати інтуітивно зрозумілі сценарії взаємодії, що динамічно адаптуються залежно від стану програми, платформи чи навіть поведінки користувача.

Наприклад, у мобільних додатках із багатим UI/UX (такі як стрімінгові або соціальні мережі), скінченні автомати можуть забезпечувати плавні переходи між екранами, зміну теми інтерфейсу залежно від стану (наприклад, темний/світлий режим), та обробку складних послідовностей дій, як-от затримка у відображенні елементів до завершення попередньої анімації.

Ключова перевага цього підходу полягає у можливості автоматизації обробки подій та черг дій. Наприклад, якщо користувач активно взаємодіє з додатком, автомат може визначити пріоритетність певних задач, таких як відображення сповіщень, завантаження даних або активація анімацій. У випадках, коли активність користувача знижується, автомат може переводити інтерфейс у режим оптимізації ресурсів.

Додатковою перевагою є адаптація анімацій. Скінченні автомати дозволяють налаштувати анімації під специфічні дії користувача, забезпечуючи плавність і послідовність переходів, що позитивно впливає на сприйняття програми.

Однак цей підхід вимагає детального проєктування станів і переходів, що може стати викликом у складних додатках із великою кількістю сценаріїв. Також необхідність інтеграції автоматів із системами рендерингу та анімації може ускладнити реалізацію.

Метод D ідеально підходить для додатків, орієнтованих на користувацький досвід, таких як мобільні ігри, додатки для онлайн-шопінгу, або інтерактивні освітні платформи. Завдяки цьому підходу можна досягти високого рівня задоволення користувачів, забезпечуючи адаптивний, гнучкий і привабливий інтерфейс.

Використання скінченних автоматів для управління ресурсами (Resource Management) (Метод E):

Метод E спрямований на ефективне використання апаратних ресурсів мобільного пристрою шляхом застосування скінченних автоматів для управління споживанням пам’яті, процесорного часу та інших обчислювальних ресурсів. Цей підхід особливо актуальний для додатків, які працюють у середовищах із обмеженими ресурсами (старі пристрої, багатозадачність, тривала робота в тлі).

Однією з ключових переваг цього методу є можливість адаптації додатка залежно від його стану. Наприклад, у пасивному режимі (коли додаток працює у фоновому режимі або очікує дії користувача) автомат може зменшити частоту оновлення даних, відкласти виконання важких обчислень або звільнити кешовані

ресурси. У активному режимі, навпаки, автомат може підвищити пріоритетність виконання завдань, щоб забезпечити максимально плавну взаємодію з користувачем.

Додаткова перевага методу полягає в оптимізації важких обчислень. Наприклад, якщо додаток виконує ресурсоємні операції (таких як рендеринг великих обсягів даних або обробка мультимедійного контенту), автомат може розподілити їх у часі або виконувати обчислення лише за необхідності. Це дозволяє зменшити навантаження на процесор і зберегти автономність пристрою.

Метод також може бути інтегрований із системами моніторингу ресурсів. Наприклад, автомат може реагувати на сигнали перевантаження пам’яті або збільшення енергоспоживання, змінюючи стратегію роботи додатка. Це підвищує стабільність і надійність, особливо у складних додатках із великим числом активних процесів.

Основними недоліками цього методу є його складність у реалізації та необхідність глибокого розуміння архітектури пристрою й операційної системи. Наприклад, розробникам потрібно враховувати особливості управління ресурсами для кожної платформи (Android, iOS), що може збільшити час і вартість розробки.

Метод E ідеально підходить для додатків, які виконують критичні для користувача завдання або тривалий час працюють у тлі. Приклади включають стрімінгові сервіси, навігаційні додатки та інструменти для аналізу великих обсягів даних. Завдяки цьому підходу можна досягти суттєвого покращення продуктивності, зменшення енергоспоживання та підвищення загальної ефективності.

Для порівняння версій було обрано наступні параметри:

– час виконання - оцінює швидкість виконання основних операцій у додатку. Вимірювання проводяться у мілісекундах (мс). Чим коротший час виконання, тим кращий результат;
– стабільність - відображає надійність роботи додатка і базується на кількості критичних помилок і збоїв на 1000 запусків. Для його узгодження з іншими параметрами, було змінено полярність значення:

чим менше збоїв, тим вищий показник стабільності. Для розрахунків використовується нормована шкала, де більші значення відповідають кращій стабільності;

- час розробки, оцінює тривалість реалізації, включаючи розробку, тестування та інтеграцію методу. Для зручності аналізу використовується порядкова шкала від 1 до 10, де більше значення відповідає коротшому часу розробки. Таким чином, параметр відображає перевагу швидшої реалізації, що є важливим для проєктів з обмеженим часом;
- кросплатформеність, визначає рівень спільності коду між платформами Android та iOS, а також простоту інтеграції скінченних автоматів у кросплатформені фреймворк React Native. Вимірюється як відсоток спільного коду між платформами: чим більший відсоток, тим кращий показник кросплатформенності;
- економія ресурсів, оцінює ефективність використання оперативної пам’яті (RAM) та процесорного часу (CPU) додатком. Для аналізу використовується природна шкала, яка враховує обсяг ресурсів, що споживаються. Чим більше ресурсів економиться, тим вищий показник. У розрахунках використовується шкала від 1 до 10, де 1 відповідає низькому рівню економії (високе споживання), а 10 - максимальному рівню економії ресурсів.

Виконання замірів. Значення показників ефективності для кожного методу представлені у таблиці 4.1

Таблиця 4.1 Результати вимірів показників ефективності

Методи	Час виконання (мс)	Стабільність (збої /1000 запусків)	Час розробки (шкала 1-10)	Крос платформеність (%)	Споживання ресурсів
A	120	2	10	90	3
B	80	1.5	1	70	8
C	100	2	8	85	5
D	110	1.8	3	80	1
E	90	1.2	6	75	10

У процесі аналізу виявлено, що окремі метрики мають різну спрямованість, що ускладнює подальшу порівняльну оцінку методів. Для забезпечення уніфікації виконано перетворення значень показників ефективності таким чином, щоб чим більше значення відповідало кращому результату.

Перетворення здійснювалося відповідно до таких правил:

Час виконання. Враховуючи, що менші значення часу виконання є кращими, для цього критерію застосовано формулу 4.1:

$$X' = max(X) - X \quad (4.1)$$

де X – вихідне значення;

X' – перетворене значення.

Стабільність. Оскільки стабільність спрямований показник, “чим більше, тим краще”, її значення залишаються без змін $X' = X$.

Час розробки. Даний параметр представлений у порядковій шкалі зі зростаючою спрямованістю, тому додаткового перетворення не потребує.

Кросплатформеність. Значення даного показника також не потребують перетворення, оскільки більші значення відповідають кращим показникам: $X' = X$.

Економія ресурсів. Показник відповідає спрямованості, тому перетворення $X' = X$ не виконувалося.

Результатом такого перетворення є узгодження спрямованості всіх показників ефективності, що дозволяє коректно застосовувати методи багатокритеріальної оптимізації.

Розрахунок значень для нормування показників. Для кожного показника було обчислено нормовані значення за формулою 4.2:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (4.2)$$

де X – значення показника для певного методу;

X_{max} і X_{min} – найгірше та найкраще значення відповідного показника.

Отримані значення представлені в таблиці 4.2.

Таблиця 4.2 Результати розрахунку значень для нормування показників

Методи	Різниця часу виконання	Різниця стабільності	Час розробки	Крос платформеність	Економія ресурсів
A	0	0	1	1	0.222
B	1	0.625	0	0	0.777
C	0.5	0	0.777	0.75	0.444
D	0.25	0.25	0.222	0.5	0
E	0.75	1	0.555	0.25	1

Визначення вагових коефіцієнтів за методом пропорційного розподілу.

Для врахування важливості кожного показника було використано пропорційний метод розподілу вагових коефіцієнтів. Кожен показник оцінювався за шкалою від 1 до 10 залежно від його впливу на кінцевий вибір методу. Отримані оцінки було нормалізовано, щоб отримати вагові коефіцієнти.

Оцінка важливості показників:

– час виконання (w_1): 0,3 - час виконання є важливим, оскільки продуктивність методів безпосередньо впливає на ефективність додатка;
– стабільність (w_2): 0.25 - забезпечує надійності роботи додатка є пріоритетним завданням, особливо для користувацьких сценаріїв;
– час розробки (w_3): 0.15 - час розробки має нижчий пріоритет порівняно з іншими показниками;
– кросплатформеність (w_4): 0.2 - простота адаптації між платформами є значущим аспектом для багатоплатформних проєктів;
– економія ресурсів (w_5): 0.1 - хоча економія ресурсів важлива, вона є менш критичною порівняно з іншими показниками.

Загальна формула 4.3 для розрахунку ефективності методу:

$$U = w_1 \cdot X_1 + w_2 \cdot X_2 + w_3 \cdot X_3 + w_4 \cdot X_4 + w_5 \cdot X_5 \quad (4.3)$$

де w_1, w_2, w_3, w_4, w_5 – відповідні вагові коефіцієнти ефективності показників,

X_1, X_2, X_3, X_4, X_5 — відповідні значення показників.

Розрахунки ефективності методів.

Метод A:

$$U_A = 0.3 \cdot 0 + 0.25 \cdot 0 + 0.15 \cdot 1 + 0.2 \cdot 1 + 0.1 \cdot 0.222 = 0.372$$

Метод B:

$$U_B = 0.3 \cdot 1 + 0.25 \cdot 0.625 + 0.15 \cdot 0 + 0.2 \cdot 0 + 0.1 \cdot 0.777 = 0.534$$

Метод C:

$$U_C = 0.3 \cdot 0.5 + 0.25 \cdot 0 + 0.15 \cdot 0.777 + 0.2 \cdot 0.75 + 0.1 \cdot 0.444 = 0.461$$

Метод D:

$$U_D = 0.3 \cdot 0.25 + 0.25 \cdot 0.25 + 0.15 \cdot 0.222 + 0.2 \cdot 0.5 + 0.1 \cdot 0 = 0.271$$

Метод E:

$$U_E = 0.3 \cdot 0.75 + 0.25 \cdot 1 + 0.15 \cdot 0.555 + 0.2 \cdot 0.25 + 0.1 \cdot 1 = 0.708$$

На основі проведеного аналізу показників ефективності методів та використання вагових коефіцієнтів, найбільш ефективним підходом виявився метод скінченних автоматів для управління ресурсами. Метод дозволяє оптимізувати обчислювальні процеси, мінімізувати використання оперативної пам’яті та процесорного часу, а також забезпечувати адаптивність у різних робочих умовах додатка. Такий підхід є оптимальним вибором для проєктів, що потребують раціонального використання апаратних ресурсів.

На другій позиції метод інтеграції складних скінченних автоматів із власною оптимізацією. Підхід вимагає значного часу на реалізацію, але забезпечує високий рівень контролю над процесами, що є важливою умовою для завдань із підвищеними вимогами до стабільності та продуктивності.

Найменш результативним виявився підхід використання скінченних автоматів для динамічного управління інтерфейсом користувача. Метод є корисним для покращення взаємодії з користувачем і адаптації дизайну, але поступається за такими показниками, як ефективність використання ресурсів і стабільність.

Результати оцінки ефективності методів.

Метод E , що отримав найвищий показник корисності $U_E = 0.708$, визнано найкращим.

Другим за ефективністю став метод B , із значенням $U_B = 0.534$, що демонструє високу стабільність і продуктивність.

Метод D, який має найнижчий показник $U_D = 0.271$, виявився найменш ефективним.

5 ПРОГРАМНА РЕАЛІЗАЦІЯ

5.1 Вибір інструментальних засобів для розробки скінченого автомату FSM

Для реалізації скінченого автомата (FSM) у мобільному застосунку було використано наступні open-source бібліотеки: React Native та Legend-State.

React Native – сучасний кросплатформенний фреймворк, що дозволяє створювати мобільні застосунки для iOS та Android з використанням JavaScript або TypeScript.

Основними перевагами React Native є:

- розширена мультиплатформеність. React Native надає можливість, розгортання застосунків не лише на мобільних пристроях iOS та Android, але й на настільних платформах (Windows, macOS), а також у середовищах Vision OS – нової операційної системи від Apple, що відкриває можливості адаптації застосунків для пристроїв із доповненою реальністю. Крім того, React Native підтримує tvOS для створення застосунків для телевізійних платформ, таких як Apple TV та Google TV, а також має інтеграцію з вебтехнологіями, дозволяючи використовувати React Native for Web для розгортання застосунків у браузері;
- гнучка архітектура забезпечує можливість використання єдиного кодового базису для різних платформ, що суттєво зменшує витрати на розробку, спрощує підтримку та забезпечує однорідний користувацький досвід на всіх пристроях. Такий підхід дозволяє ефективно керувати ресурсами команди, одночасно підтримуючи декілька платформ без необхідності дублювати логіку застосунку;
- інтеграція нативного коду. React Native надає можливість взаємодії з нативними компонентами операційної системи через мости (bridges) або нову архітектуру Fabric, що забезпечує ефективне використання апаратних можливостей пристрою. Підтримка мов Swift, Objective-C для iOS, Kotlin, Java для Android, а також можливість написання продуктивного коду на C++ дозволяє реалізовувати платформоспецифічні

функції, такі як робота з Bluetooth, використання ARKit, доступ до датчиків пристрою та оптимізовані нативні анімації;
- розвинена екосистема і підтримка спільноти. Завдяки відкритому вихідному коду та активній підтримці від Meta (Facebook), React Native має широку базу розробників, які щодня працюють над удосконаленням фреймворку, виправленням помилок та інтеграцією нових технологій. Велика кількість готових open-source бібліотек спрощує розробку, дозволяючи використовувати рішення для навігації, управління станом, анімацій та роботи з мережею без необхідності писати низькорівневий код. Висока залученість спільноти забезпечує оперативне вирішення проблем та постійний розвиток екосистеми, що робить React Native одним із найперспективніших рішень для створення кросплатформенних застосунків.
- Legend State – це сучасна бібліотека для управління станом у React та React Native, яка використовує реактивний підхід для оптимізації рендерингу та підвищення продуктивності. На відміну від традиційних підходів, таких як Redux або MobX, Legend State забезпечує мінімізацію зайвих оновлень, знижуючи навантаження на головний потік (UI Thread) та покращуючи продуктивність застосунку.

Основні переваги Legend State:
- реактивне управління станом. Бібліотека використовує спостережувані (observable) змінні, які автоматично відстежують зміни у стані та оновлюють тільки ті компоненти, які залежать від змінених даних. Це усуває проблему зайвих ререндерів, що властива Redux, та забезпечує високу продуктивність навіть у застосунках із великою кількістю компонентів;
- Local-first підхід. Legend State підтримує локальне зберігання стану (Local-first architecture), що дозволяє: зберігати стан на пристрої, що підвищує продуктивність та забезпечує миттєвий доступ до даних; працювати в офлайн-режимі, дозволяючи користувачам отримувати

доступ до інформації навіть без підключення до мережі; зменшувати навантаження на сервер, оскільки дані оновлюються локально та синхронізуються з сервером лише за потреби. Такий підхід особливо корисний для React Native застосунків, де швидкість взаємодії користувача з інтерфейсом є критичною;

- проста та ефективна архітектура. На відміну від Redux, який вимагає конфігурації глобального стору, редукторів та middleware, Legend State використовує просту декларативну API, що не потребує складної ініціалізації. Вона дозволяє зменшити обсяг коду та підвищити гнучкість у розробці;
- гнучкість та модульність. Бібліотека дозволяє створювати локальні та глобальні стани, які можуть бути незалежними або взаємопов'язаними. Це спрощує управління даними у великих проєктах та зменшує складність логіки оновлення стану;
- інтеграція зі скінченними автоматами (FSM). Legend State чудово підходить для реалізації скінченних автоматів, оскільки дозволяє легко керувати змінами стану та відстежувати переходи між ними. Завдяки реактивній природі бібліотеки, можна ефективно моделювати логіку зміни станів, що особливо важливо для автентифікації, керування навігацією, обробки запитів до серверу та управління складними UI-компонентами;
- висока продуктивність. Використання реактивних обчислень та автоматичної оптимізації оновлень дозволяє значно знизити кількість викликів ререндеру компонентів.

Це особливо важливо для мобільних застосунків, де обмежені обчислювальні ресурси, а кожен зайвий виклик UI-оновлення може вплинути на плавність роботи інтерфейсу.

5.2 Базова архітектура React-Native (old architecture)

Стара архітектура React Native базується на трьох основних потоках виконання та механізмі комунікації між ними через JavaScript Bridge. Ця модель дозволяє поєднувати JavaScript-логіку з нативними компонентами платформи, забезпечуючи інтеграцію між React та операційними системами iOS та Android.

Основні компоненти архітектури:

– JavaScript Thread – потік, який відповідає за виконання бізнес-логіки застосунку, рендеринг віртуального дерева компонентів (Virtual DOM) та обробку подій;
– Native (Main) Thread – потік, що керує відображенням нативного UI та виконує виклики до нативних API пристрою;
– Background Thread (Shadow Thread) – потік, який виконує обчислення розміщення компонентів, використовуючи Yoga C++ Layout Engine;
– Bridge (міст між JavaScript та нативною частиною) – механізм асинхронної комунікації, що передає JSON-серіалізовані дані між JavaScript Thread та Native Thread.

Загальний вигляд старої архітектури RN відображений на рисунку 5.1.

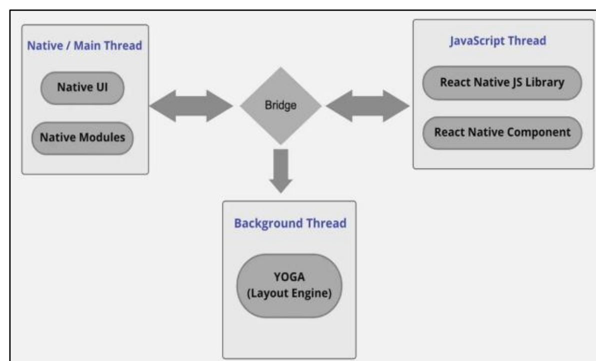


Рисунок 5.1 RN – стара архітектура (за даними [9])

У старій архітектурі React Native JavaScript Thread (JS Thread) є центральним потоком, відповідальним за виконання основної бізнес-логіки застосунку. До його основних завдань належать обчислення стану компонентів, виконання мережевих запитів та ініціювання оновлень інтерфейсу користувача. JS

Thread також відповідає за виконання JavaScript-коду сторонніх бібліотек, наприклад, таких як React Navigation, Redux чи AsyncStorage, які забезпечують додаткову логіку та управління станом застосунку. Взаємодія з нативним інтерфейсом здійснюється шляхом передачі відповідних команд у нативний потік через механізм JavaScript Bridge.

На відміну від вебплатформи, React Native не використовує концепцію Virtual DOM. Замість цього JS Thread підтримує внутрішню модель компонентів у пам'яті та визначає, які з них потребують оновлення, додавання або видалення. Ця модель оновлень потім передається до Native Thread, де безпосередньо застосовується до нативних компонентів (UIView на iOS або View на Android).

Через асинхронну природу цього процесу може виникати помітна затримка між оновленням стану в JavaScript та його фактичним рендерингом в нативному інтерфейсі.

Важливим завданням JS Thread також є обробка подій користувача, таких як жести, натискання кнопок або зміни введення тексту. Події відстежуються спочатку в нативному середовищі й потім асинхронно передаються у JS Thread через JavaScript Bridge. Після цього JavaScript-код здійснює аналіз отриманих подій, визначаючи, чи потрібно оновлювати UI або змінювати стан застосунку.

Проте навіть незначні затримки в обробці JavaScript-коду можуть спричинити погіршення взаємодії користувача з інтерфейсом. Native (Main) Thread у старій архітектурі React Native є основним потоком виконання операційної системи, який відповідає за відображення нативних компонентів інтерфейсу користувача, обробку анімацій, а також взаємодію із системними викликами та апаратними ресурсами пристрою.

Саме на цьому потоці безпосередньо відбувається оновлення інтерфейсу за допомогою нативних компонентів, специфічних для кожної платформи, таких як, наприклад, UIView в iOS або View в Android. Ці оновлення здійснюються на основі команд, що надходять від JavaScript Thread через JavaScript Bridge у формі серіалізованих JSON-даних.

Однією з ключових задач Main Thread є забезпечення стабільної та плавної роботи інтерфейсу, особливо під час анімацій та реакції на взаємодії користувача. Проте надмірне навантаження на цей потік може призвести до зниження швидкодії застосунку, оскільки він відповідає не лише за відображення графічних елементів, а й за обробку усіх системних подій. У результаті перевантаження потоку анімації можуть втрачати плавність, а інтерфейс реагуватиме на жести користувача із затримками, що погіршує загальний користувацький досвід.

Передача команд для оновлення інтерфейсу з JavaScript Thread до Native Thread відбувається за допомогою JavaScript Bridge, який використовує асинхронний спосіб передачі даних у форматі JSON. Цей механізм спричиняє певні недоліки через затримки, що виникають у процесі серіалізації, передачі та десереалізації даних перед їхнім застосуванням у головному потоці. У випадках, коли зміни інтерфейсу відбуваються часто та швидко, такі затримки можуть суттєво знижувати швидкість оновлення компонентів.

Окрім відображення інтерфейсу, Main Thread також відповідає за виконання викликів до нативних API, що потребують прямого доступу до ресурсів пристрою. Це можуть бути операції з камерою, датчиками руху, геолокацією чи файловою системою.

Background Thread, також відомий як Shadow Thread, у старій архітектурі React Native виконує допоміжну роль, займаючись обчисленнями розміщення та позиціонування компонентів інтерфейсу користувача. Його головна задача полягає у попередньому опрацюванні та визначенні точних координат, розмірів та стилів елементів перед тим, як вони будуть відображені у нативному середовищі. Завдяки винесенню цих розрахунків у окремий потік, основний потік (Main Thread) може уникнути значного навантаження та затримок під час рендерингу, особливо в застосунках із великою кількістю компонентів або складним макетом.

У цьому потоці активно використовується спеціалізований рушій для обчислення макетів – Yoga Layout Engine, написаний мовою програмування C++. Yoga реалізує гнучку систему компонування на основі стандарту Flexbox, що дозволяє чітко і точно визначити положення та розміри кожного елемента

інтерфейсу. Він працює повністю у фоновому режимі, тому ці обчислення не створюють зайвого навантаження на основний потік відображення інтерфейсу (Main Thread), а натомість результати розрахунків передаються у нативний потік для фінального застосування.

Взаємодія між Shadow Thread та Main Thread здійснюється через внутрішні механізми React Native: після того як Yoga C++ Layout Engine завершує всі необхідні обчислення, фінальні координати компонентів передаються у Native (Main) Thread для оновлення візуального представлення компонентів на екрані. Завдяки цьому користувач бачить результат роботи макету лише після завершення всіх попередніх обчислень, що дозволяє уникнути некоректного проміжного розміщення компонентів та знижує навантаження на графічну підсистему пристрою.

5.3 Недоліки старої архітектури React Native

Стара архітектура React Native базується на використанні асинхронної моделі взаємодії через JavaScript Bridge. Незважаючи на те, що така модель забезпечує гнучкість та простоту інтеграції JavaScript-коду з нативним середовищем, вона має низку суттєвих обмежень, що негативно впливають на загальну продуктивність та якість роботи застосунку.

Основними проблемами старої архітектури є:

- асинхронна комунікація між потоками. Асинхронний характер JavaScript Bridge призводить до затримок у передачі даних між JavaScript Thread та Native Thread. Особливо це проявляється під час складних обчислень, великої кількості запитів до API чи інтенсивної взаємодії з користувачем, коли застосунок не може миттєво застосувати зміни у нативному середовищі;
- високі накладні витрати на серіалізацію даних. Кожен обмін між JavaScript та Native Thread здійснюється через JSON-сереалізовані повідомлення. Сереалізація та десереалізація великих обсягів інформації потребує додаткового часу та ресурсів, що призводить до додаткових

затримок у процесі оновлення інтерфейсу, особливо в умовах інтенсивних анімацій;

- відсутність синхронної взаємодії з нативним кодом. Асинхронна природа JavaScript Bridge не дозволяє реалізовувати синхронні запити до нативних модулів. Це стає перешкодою для роботи застосунків, яким необхідна швидка відповідь від операційної системи, наприклад, при роботі з сенсорами, геолокацією чи обробкою даних у реальному часі;
- обмежена швидкодія при складних анімаціях. Через те, що всі оновлення інтерфейсу повинні проходити через JavaScript Bridge, анімації, які вимагають швидкого оновлення кадрів, можуть сповільнюватися або втрачати плавність. Це особливо помітно на менш продуктивних пристроях, де додаткові витрати на серіалізацію команд можуть спричинити суттєві «підвисання» UI;
- складність відлагодження. Асинхронна комунікація та використання JSON-формату для передачі даних ускладнюють процес відлагодження застосунку. Аналіз помилок та проблем продуктивності ускладнюється необхідністю відслідковувати численні асинхронні повідомлення, що циркулюють між JavaScript та нативною частиною.

5.4 Нова архітектура React-Native (new architecture)

Нова архітектура React Native базується на вдосконаленій моделі потоків виконання та покращеному механізмі комунікації між ними з використанням JavaScript Interface (JSI). Цей підхід замінює попередню асинхронну модель JavaScript Bridge, забезпечуючи пряму, ефективну та синхронну взаємодію між JavaScript-кодом і нативним середовищем операційних систем iOS та Android.

Основна перевага нової архітектури – значне підвищення швидкодії, стабільності та інтерактивності мобільних застосунків.

Компоненти нової архітектури React Native:

– JavaScript Thread – основний потік, відповідальний за виконання бізнес-логіки застосунку, керування станом React-компонентів, обробку подій користувача та взаємодію з новим механізмом JSI (JavaScript Interface).
– Native (Main) Thread – потік, що безпосередньо відповідає за відображення нативного інтерфейсу користувача, виконання викликів до API операційної системи та інтеграцію з нативними TurboModules, які замінюють застарілий механізм Native Modules.
– Background Thread (Shadow Thread) – потік, який використовує рушій Yoga Layout Engine для виконання попередніх обчислень макета, визначаючи координати, розміри та позиціонування компонентів перед їх відображенням.

Загальний вигляд нової архітектури RN відображений на рисунку 5.2.

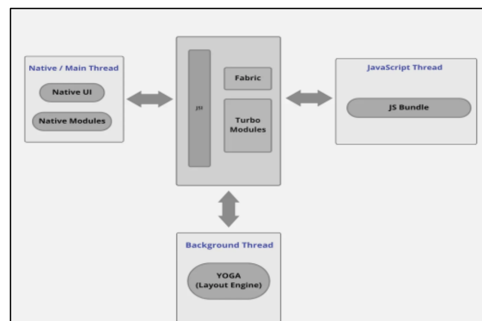


Рисунок 5.2 RN – нова архітектура (за даними [9])

JavaScript Interface (JSI) є одним з ключових удосконалень нової архітектури React Native, яке покликане замінити собою старий асинхронний механізм JavaScript Bridge.

JSI (JavaScript Interface) – це низькорівневий інтерфейс, що дозволяє JavaScript напряму викликати методи нативного коду, написані на таких мовах, як Swift, Objective-C, Kotlin чи C++, і навпаки. Завдяки цьому React Native може виконувати синхронні виклики та отримувати миттєву відповідь, що значно покращує швидкість взаємодії та зменшує затримки в оновленні інтерфейсу.

На платформі Android для взаємодії між JavaScript та нативною частиною використовується JNI (Java Native Interface), який забезпечує зв’язок між кодом на Java чи Kotlin та JavaScript-движком. Використання JNI у поєднанні з JSI дозволяє

значно скоротити час, необхідний для передачі даних між JavaScript і нативною частиною, усуваючи зайві етапи серіалізації та десеріалізації JSON-повідомлень, характерних для старого підходу з JavaScript Bridge. Це покращує швидкодію мобільних застосунків, особливо у ситуаціях, що вимагають частих і швидких оновлень UI або взаємодії з сенсорами пристроїв.

Важливим нововведенням є також поява Turbo Modules – спеціалізованих модулів, що реалізують ефективнішу комунікацію між JavaScript та нативним середовищем. Turbo Modules дозволяють створювати нативні модулі, до яких JavaScript-код звертається безпосередньо, без проміжної JSON-серіалізації. На відміну від старих нативних модулів, TurboModules можуть виконувати виклики синхронно, що суттєво покращує продуктивність застосунків. Це особливо важливо для реалізації модулів, які вимагають мінімальної затримки, наприклад, модулів для роботи з файловою системою або модуля роботи з високочастотними подіями.

Ще одним важливим компонентом нової архітектури є Fabric – новий рушій для управління нативним інтерфейсом, який замінює стару систему на базі UIManager.

Fabric дозволяє використовувати синхронне оновлення компонентів та забезпечує більшу ефективність взаємодії з інтерфейсом за рахунок усунення проміжних етапів серіалізації та передачі даних. Це забезпечує швидше оновлення елементів інтерфейсу, покращує плавність анімацій та усуває проблеми, пов’язані з затримками, які були притаманні старій архітектурі.

У новій архітектурі також було оптимізовано використання Yoga Layout Engine. Тепер він інтегрований напряму в Fabric, що дозволяє швидше та точніше розраховувати розташування компонентів у Background Thread без необхідності передачі великого обсягу серіалізованих даних між потоками. Це підвищує ефективність розрахунку макетів, зменшує затримки та дозволяє застосунку стабільно працювати навіть зі складними та часто змінюваними макетами екранів.

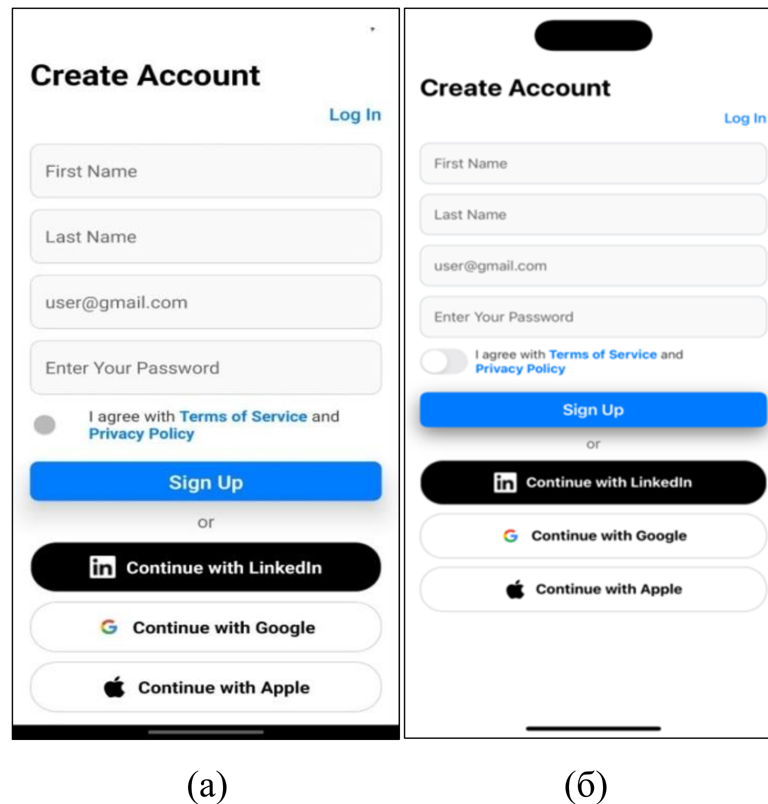
Таким чином, нова архітектура React Native, що використовує JSI, TurboModules та Fabric, усуває ключові проблеми попередньої архітектури,

пов'язані з асинхронною передачею даних через міст. Це дозволяє суттєво підвищити продуктивність, скоротити затримки взаємодії з користувачем та значно покращити загальний досвід використання застосунків, особливо у випадках, коли необхідна висока швидкодія, інтерактивні анімації та інтенсивна взаємодія з апаратними ресурсами мобільних пристроїв.

6 ЕКСПЕРЕМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

6.1 Вихідні дані

У межах практичної частини роботи було проведено тестування кінцевого автомата (FSM) з використанням спроєктованого мобільного додатку, створеного на базі кросплатформенного фреймворку React Native (стара / нова архітектура) для платформ Android та iOS. Вибір наведеного дизайну екранів користувача, рисунок 5.3, обумовлений їх високою частотою взаємодії та великою варіативністю станів, які можуть виникати під час реєстрації користувачів.



(а) (б)

Рисунок 6.1 – User Interface при реєстрації в мобільному додатку для мобільних платформ: (а) Android, (б) iOS (рисунок виконано самостійно)

На представлених екранах користувачі мають доступ до таких користувацьких дій, як: заповнення текстових полів (ім’я, прізвище, адреса електронної пошти, пароль); взаємодія з елементами керування (перемикачами, кнопками реєстрації та авторизації); можливість вибору альтернативних способів входу через сервіси сторонніх провайдерів Google, Apple, LinkedIn.

При проведенні тестування порівнювалися дві різні реалізації управління станом екрану реєстрації: перша – на основі декларативного підходу, що використовується за замовчуванням у фреймворку React Native та друга – з використанням кінцевого автомата (FSM).

Приорітетним показником оцінки користувацької зручності розробленого застосунку були обрані: плавність роботи інтерфейсу (FPS); швидкість реакції застосунку на введення даних користувачем; коректність зміни станів компонентів інтерфейсу залежно від логіки взаємодії; загальний вплив обраного підходу на стабільність і продуктивність мобільного застосунку.

Тестування проводилося на пристроях iPhone 15 Pro та Samsung Galaxy FE у режимах з частотою оновлення екранів 60 і 120 Гц. Також було проведено порівняльний аналіз продуктивності у двох різних архітектурах React Native: старій архітектурі з використанням JavaScript Bridge і новій, що використовує JSI, Turbo Modules та Fabric.

6.2 Реалізація декларативного підходу

Для реалізації екрана реєстрації користувачів був використаний стандартний декларативний підхід React Native з використанням локального стану компонента (useState). В декларативному підході кожен стан екрану визначається окремо, а зміни стану безпосередньо впливають на відображення інтерфейсу.

Синтаксично в коді на початку компонента AuthDefault було створено декілька основних змінних стану, які використовуються для відображення та збереження інформації, введеної користувачем, а також для індикації помилок та стану завантаження.

Змінними стану мобільного додатку були обрані:

- email та password: для зберігання адреси електронної пошти та пароля користувача;
- firstName та lastName: для зберігання імені та прізвища користувача;
- agreeTerms: логічна змінна, яка фіксує згоду користувача з умовами використання додатку та політикою конфіденційності;

– errors: об’єкт, який містить інформацію про помилки валідації введених користувачем даних;
– isLoading: логічна змінна, що відображає стан завантаження, який є індикатором при виконанні асинхронних операцій.

В складі компонента AuthDefault реалізовано функції: validate, handleSubmit. Функція validate дозволяє: перевірити коректність введення користувачем інформації у реєстраційну форму; виконує послідовну перевірку полів форми на відповідність заданим критеріям: наявність тексту в полях «ім’я» та «прізвище»; правильний формат електронної пошти та мінімальна довжина пароля.

У разі невідповідності функція формує об’єкт помилок та встановлює його значення у стані компонента. Результатом виконання функції є логічне значення, що підтверджує валідацію форми.

Функція handleSubmit відповідає за кінцеву обробку форми реєстрації. При її виклику запускається процес валідації. При позитивному результаті валідації, форма передає введені користувачем дані на наступний етап обробки (у поточній реалізації – логування інформації в консоль). У випадку помилок форма не відправляється, а користувач отримує відповідні повідомлення з маркуванням та описом помилок.

Окрім стандартної реєстрації, було реалізовано додаткові асинхронні функції для авторизації користувача через зовнішні сервіси: signInWithGoogle, signInWithApple та signInWithLinkedIn. Асинхронні функції використовують стан isLoading для демонстрації процесу завантаження та здійснюють імітацію виконання авторизації шляхом створення асинхронної затримки.

Після завершення асинхронної операції відповідна функція виводить інформацію в консоль і повертає застосунок у початковий стан.

Програмний код працює коректно у більшості типових випадків. Однак на пристроях з обмеженими обчислювальними ресурсами або в ситуаціях, коли користувач вводить текст дуже швидко, виникають проблеми з оновленням інтерфейсу, обумовлені асинхронною природою React Native.

Аналіз порядоку операцій, які виконуються під час введення символів у компонент <TextInput /> дає можливість довести низьку ефективність використання декларативного підходу при розробці компонента AuthDefault (див.рис. 6.2).

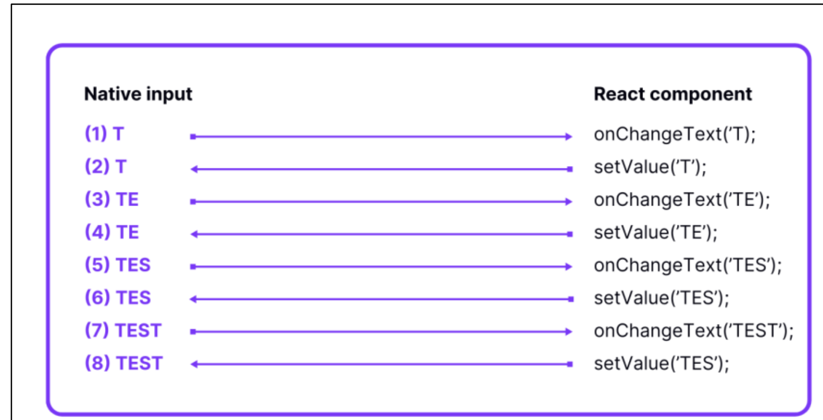


Рисунок 6.2 – Схема послідовності передачі введених даних з native сторони в середовище RN (за даними [5])

При введенні користувачем нового символу, React Native отримує оновлення через властивість onChangeText (операція 1 рисунок 5.2.). React Native обробляє ці дані та оновлює свій внутрішній стан, використовуючи функцію setState. Значення JavaScript-компонента синхронізується зі значенням нативного компонента введеного тексту (операція 2 рисунок 6.2).

Описаний підхід має значні переваги. React Native виступає «єдиним джерелом істини», який детермінує значення всіх полів вводу у застосунку. Що дозволяє змінювати або перевіряти дані безпосередньо під час введення. Наприклад, здійснювати валідацію, застосовувати маску або повністю змінювати значення даних, враховуючи прописані критерії.

Незважаючи на чіткість і коректність роботи React Native, має суттєві недоліки. Вони проявляються у випадках, коли ресурси пристрою є обмеженими або користувач вводить текст з дуже високою швидкістю.

За таких умов виникає проблема мерехтіння інтерфейсу або навіть втрата введених символів (див. рис. 6.3).

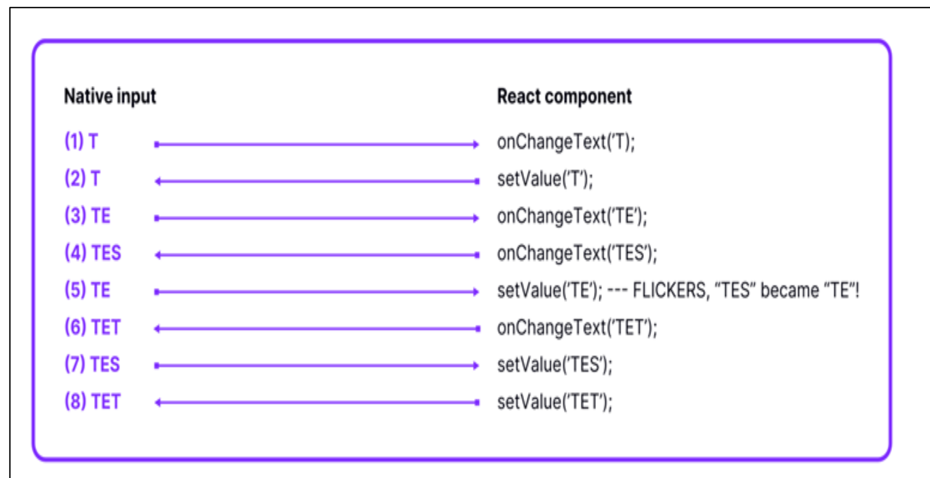


Рисунок 6.3 – Схема послідовності передачі введених даних з native сторони в середовище RN з використанням технічно обмежених пристроїв (за даними [5])

Причиною є затримка синхронізації стану між JavaScript-компонентом і нативним компонентом введення, що негативно впливає на користувацький досвід.

Ще однією проблемою декларативного підходу React Native є повторний рендеринг всього компоненту при зміні навіть одного конкретного поля форми. Це призводить до зайвих витрат ресурсів, сповільнює швидкість оновлення інтерфейсу та знижує загальну продуктивність застосунку.

Також при швидкій взаємодії з інтерфейсом, користувач потенційно може викликати кілька дій одночасно. Наприклад, натиснути декілька кнопок авторизації або швидко змінювати фокус між текстовими полями введення. Це спричиняє ситуації, коли компонент починає обробляти декілька станів паралельно, що призводить до некоректної роботи застосунку, конфліктів у логіці й додаткового навантаження на пристрій.

Вищезазначені проблеми обґрунтовують необхідність та актуальність пошуку альтернативного підходу до управління станом мобільних додатків, зокрема за допомогою кінцевих автоматів, які дозволяють уникнути та забезпечити стабільне та швидке оновлення компонентів інтерфейсу користувача.

6.3 Реалізація управління станом за допомогою кінцевого автомата (FSM) та бібліотеки Legend-State

Для розв’язання проблем, властивих декларативному підходу при керуванні станами мобільного застосунку у React Native, було реалізовано альтернативну архітектуру на основі кінцевого автомата (FSM) у поєднанні з реактивною бібліотекою Legend-State. Код імплементації додаток Б.

Підхід дозволяє ефективно реалізувати всі можливі стани інтерфейсу, переходи між ними та реакції на користувацькі дії. Структура створена з використанням FSM забезпечує більшу контрольованість логіки, знижує ймовірність конфліктів між діями та підвищує продуктивність мобільного застосунку.

В основі реалізації лежить реактивний об’єкт stateMachine$, який створено за допомогою функції observable.

В якому зберігаються:

- values: реактивні поля, що містять дані, введені користувачем (ім’я, прізвище, email, пароль, згоду з умовами використання);
- errors: реактивний об’єкт з повідомленнями та індикацією помилок, які виникають під час валідації;
- state: поточний стан кінцевого автомата, який може набувати одного з п’яти значень: idle, validating, submitting, success, error;
- isLoading: індикатор виконання асинхронної операції;
- send(event): основний механізм зміни стану автомата та ініціації відповідних дій;
- три методи: signInWithGoogle, signInWithLinkedIn і signInWithApple, які імітують асинхронну автентифікацію.

Основна логіка керування станами зосереджена в методі send, який приймає події типу VALIDATE, SUBMIT, SUCCESS, ERROR, RESET. Залежно від поточного стану та події, виконується перехід до нового стану, відповідно до словника transitions.

Наприклад, після події VALIDATE виконується валідація даних, і FSM переходить або в error, або в success, залежно від результатів перевірки. У разі SUBMIT викликається попередня перевірка, і якщо вона успішна – симулюється надсилання форми, після чого виконується перехід у стан success.

Інтерфейс форми побудовано з використанням реактивних компонентів TextInput, Switch та Show з бібліотеки Legend-State / React-Native, які оновлюються тільки при зміні відповідного значення. Це дозволяє уникнути непотрібних ререндерів усього компонента, зменшуючи навантаження на головний потік, підвищуючи стабільність анімацій при високій частоті оновлення екрана.

Ключова перевага такого підходу – чітке розмежування станів і виключення можливості одночасного перебування в кількох конфліктних станах. Це особливо важливо в контексті мобільних інтерфейсів, де користувач може швидко взаємодіяти з декількома елементами, такими як кнопки авторизації або поля вводу. FSM гарантує, що в момент натискання лише одна логіка буде виконуватися, відповідно до дозволеного переходу.

Суттєвою перевагою FSM є легкість масштабування функціоналу мобільного застосунку. Завдяки чіткій ієрархічній структурі станів і переходів, створеної на базі FSM, є можливість додавати нову логіку, сценарії взаємодії застосунку з користувачем, розширити механізми валідації без суттєвих змін існуючого коду.

Підхід є ефективним для програмної реалізації складних багатокрокових форм, де одночасно виникає необхідність одночасного запуску процесів валідації, відправлення запитів та зміни структури UI.

Таким чином, поєднання Legend-State та FSM дозволяє досягти високої продуктивності, передбачуваної поведінки інтерфейсу та гнучкої архітектури для розширення функціоналу в майбутньому, що є основним критерієм технічної та користувацької якості сучасних мобільних додатків, і відповідає ринковим запитам замовників.

7 АНАЛІЗ РЕЗУЛЬТАТІВ

Для оцінки продуктивності роботи мобільного додатку, спроєктованого з використанням розробленого скінченного автомату, було використано середовище розробки скросплатформений framework React Native на базі старої та нової архітектури застосунку та метрики.

Average FPS (середня частота кадрів за секунду) – показник плавності відображення інтерфейсу застосунку. Більш високе значення FPS свідчить про кращу плавність та якість користувацького досвіду.

Average CPU Usage (середнє завантаження процесора) – характеризує, який відсоток потужності центрального процесора пристрою використовується для роботи застосунку. Нижчий рівень використання процесора вказує на оптимальніше використання ресурсів пристрою.

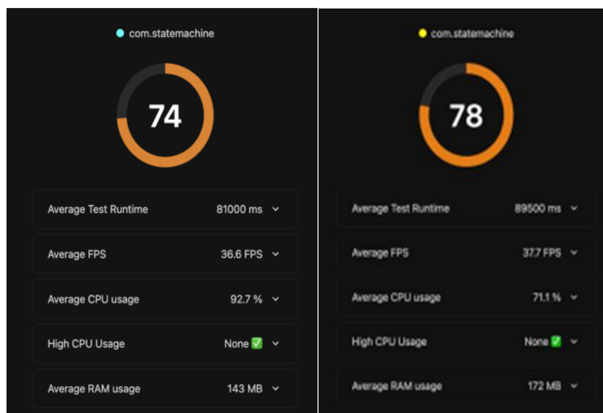
High CPU Usage (пікове навантаження процесора) – індикатор, який вказує на наявність пікових значень завантаження CPU. Мінімальні або відсутні пікові значення свідчать про стабільнішу роботу застосунку.

Average RAM Usage (середнє використання оперативної пам’яті) – визначає, скільки оперативної пам’яті використовується застосунком у середньому під час його роботи. Менші значення свідчать про ефективніше управління ресурсами пам’яті.

CPU Usage per Thread (навантаження CPU за потоками) – метрика, яка дає детальнішу інформацію про те, які конкретні потоки виконання (JavaScript Thread, Bridge Thread, UI Thread) спричиняють найбільше навантаження на процесор.

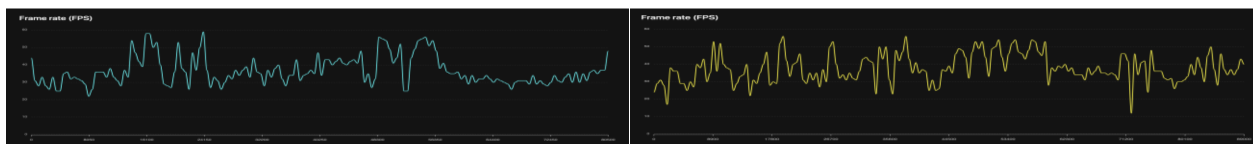
7.1 Результати ефективності роботи мобільного застосунку з використанням декларативного підходу в React Native

Результати тестування мобільного застосунку, розробленого із застосуванням декларативного підходу в середовищі React Native, з використанням старої та нової архітектур мобільного додатку представлені на рисунках 7.1 – 7.4 та таблиці 7.1



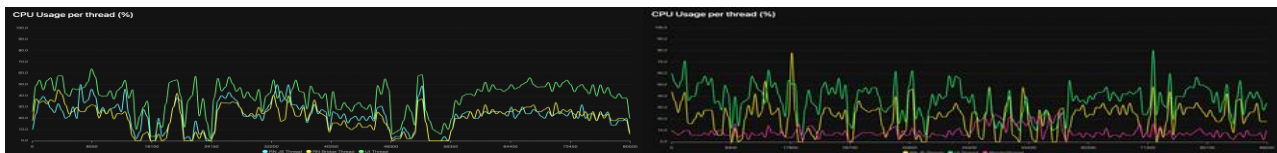
(а) (б)

Рисунок 7.1 – Загальні показники ефективності роботи мобільного додатку (декларативний підхід): (а) стара архітектура; (б) нова архітектура



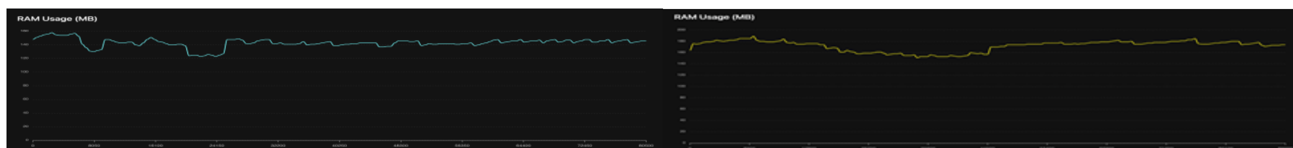
(а) (б)

Рисунок 7.2 – Графіки показників Average FPS екрану реєстрації (декларативний підхід): (а) стара архітектура; (б) нова архітектура



(а) (б)

Рисунок 7.3 – Графіки показників Average CPU Usage екрану реєстрації (декларативний підхід): (а) стара архітектура; (б) нова архітектура



(а) (б)

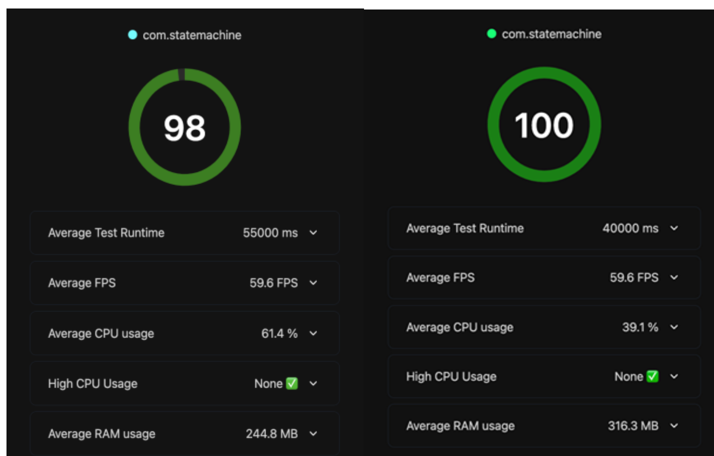
Рисунок 7.4 – Графіки показників Average RAM Usage екрану реєстрації (декларативний підхід): (а) стара архітектура; (б) нова архітектура

Таблиця 7.1 – Результати вимірювання метрик при декларативному підході

Метрика	Результати вимірювань		
	Стара архітектура	Нова архітектура	Різниця, %
Average FPS, FPS	36,6	37,7	+ 3,01
Average CPU Usage, %	92,70	71,10	-23,32
High CPU Usage	відсутні пікові значення	відсутні пікові значення	-
Average RAM Usage, MB	143	172	+ 20,28

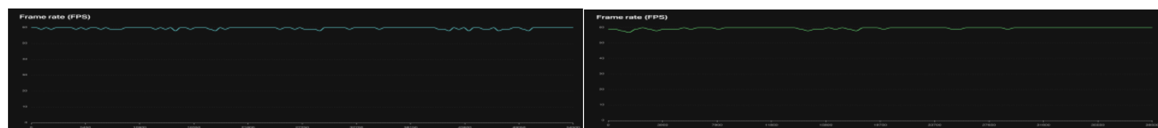
7.2 Результати продуктивності мобільного застосунку з використанням FSM у React Native

Результати тестування мобільного застосунку, розробленого із застосуванням FSM в середовищі React Native, з використанням старої та нової архітектур представлені на рисунках 7.5 – 7.8 та таблиці 7.2.



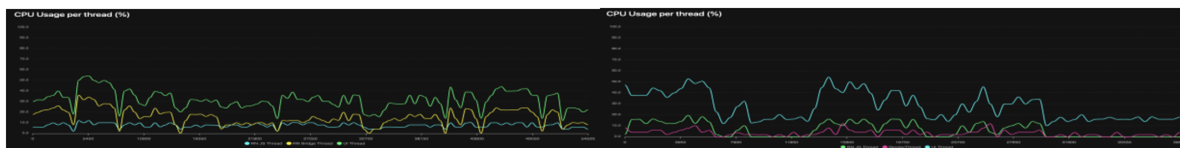
(а) (б)

Рисунок 7.5 – Загальні показники ефективності роботи мобільного додатку (FSM): (а) стара архітектура; (б) нова архітектура



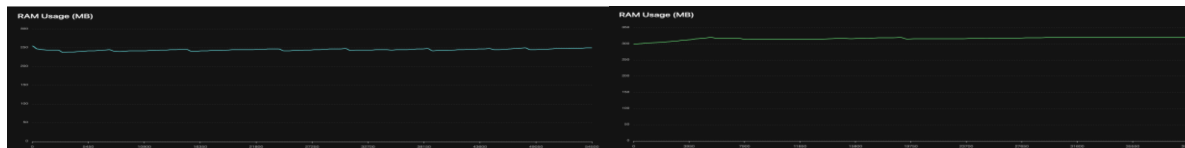
(а) (б)

Рисунок 7.6 – Графіки показників Average FPS екрану реєстрації (FSM): (а) стара архітектура; (б) нова архітектура



(а) (б)

Рисунок 7.7 – Графіки показників Average CPU Usage екрану реєстрації (FSM): (а) стара архітектура; (б) нова архітектура



(а) (б)

Рисунок 7.8 – Графіки показників Average RAM Usage екрану реєстрації (FSM): (а) стара архітектура; (б) нова архітектура

Таблиця 7.2 – Результати вимірювання метрик з FSM

Метрика	Результати вимірювань		
	Стара архітектура	Нова архітектура	Різниця, %
Average FPS, FPS	59,6	59,6	-
Average CPU Usage, %	61,40	39,1	-36,30
High CPU Usage	відсутні пікові значення	відсутні пікові значення	-
Average RAM Usage, MB	244,8	316,3	+29,23

Порівняльння продуктивності мобільного додатку наведено в таблицях 7.3 та 7.4.

Таблиця 7.3 – Порівняння продуктивності мобільного додатку з використанням різних підходів управління станами (стара архітектура)

Метрика	Декларативний підхід	Legend State (FSM)	Різниця, %
Average FPS, FPS	36,6	59,6	+62,84
Average CPU Usage, %	92,70	61,40	-33,76
High CPU Usage	відсутні пікові значення	відсутні пікові значення	-
Average RAM Usage, MB	143	244,8	+71,33

Таблиця 7.4 – Порівняння продуктивності мобільного додатку з використанням різних підходів управління станами (нова архітектура)

Метрика	Декларативний підхід	Legend State (FSM)	Різниця, %
Average FPS, FPS	37,7	59,6	+58,09
Average CPU Usage, %	71,10	39,10	-45,01
High CPU Usage	відсутні пікові значення	відсутні пікові значення	-
Average RAM Usage, MB	172	316,3	+83,84

ВИСНОВКИ

Результатом проведеного дослідження ефективності та доцільності інтеграції скінченних автоматів у розробку кросплатформених мобільних застосунків є порівняльний аналіз використання декларативного та FSM методів. Дослідження містить теоретичне обгрунтування та експериментальну частину.

Проведений аналіз літературних та наукових джерел інформації дозволив порівняти існуючи методи управління станами кросплатформених мобільних додатків, визначити основну проблематику даної галузі, обгрунтувати та спроєктувати архітектуру FSM, обрати середовище розробки.

Проведене експериментальне дослідження дозволило порівняти два підходів до управління інтерфейсною логікою - декларативного та на основі скінченного автомата (FSM).

Порівняльний аналіз результатів продуктивності кросплатформеного мобільного додатку з використанням декларативної моделі та скінченного автомату (FSM) у двох архітектурних конфігураціях (старій та новій) підтвердив переваги FSM-моделі в більшості використаних метрик.

Результати із застосуванням старої архітектури:

- FPS зріс із 36,6 до 59,6, що свідчить про збільшення стабільності зображення на + 62,84%;
- середнє навантаження на CPU зменшилося з 92,7% до 61,4%, що підтверджує зниження навантаження на центральний процесор мобільного телефону на - 33,76%;
- використання оперативної пам’яті збільшилося з 143 MB до 244,8 MB, що складає + 71,33%.

Результати із застосуванням нової архітектури:

- FPS покращилось з 37,7 до 59,6 , що складає + 58,09%;
- показник навантаження на центральний процесор CPU зменшився з 71,1% до 39,1%, що становить - 45,01%;

– зросло використання оперативної пам’яті з 172 MB до 316,3 MB, приріст склав + 83,84%.

Декларативний підхід на основі базового React Native у рамках старої архітектури показав найгірші результати з усіх конфігурацій. Спостерігалося надмірне навантаження на JS-потік (92,7 %), значне падіння FPS до 36,6.

Перехід до нової архітектури React Native із збереженням декларативної логіки дозволив частково зменшити навантаження: середнє використання CPU знизилося до 71,1 %, а FPS дещо зріс до 37,7. Використання Legend-State у поєднанні зі State Machine на старій архітектурі дало суттєвий приріст продуктивності. Середній FPS досяг стабільного значення 59,6, а середнє використання CPU знизилося до 61,4 %.

Найкращі показники було зафіксовано при використанні Legend-State + State Machine разом із новою архітектурою React Native. Ця комбінація дозволила досягти майже максимальної продуктивності інтерфейсу: FPS становив 59,6, CPU-навантаження знизилося до 39,1 %.

Незначним недоліком стало збільшення споживання оперативної пам’яті до 316,3 MB, що є прийнятним в умовах сучасних мобільних пристроїв.

Завдяки синхронній взаємодії JavaScript із нативним середовищем через JSI, а також ефективному оновленню UI через Fabric, ця конфігурація забезпечила найвищу плавність роботи застосунку.

Отримані результати підтверджують доцільність впровадження State Machine-підходів у проєктах із високими вимогами до продуктивності та стабільності інтерфейсу.

Поєднання нового архітектурного стеку React Native з реактивними бібліотеками та автомато-орієнтованим управлінням станом демонструє найкраще співвідношення між ефективністю, адаптивністю та масштабованістю для реальних мобільних сценаріїв.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Безверхий О. І., & Куценко О. І. (2024). Шляхи оптимізації кросс-платформного застосування за допомогою бібліотеки React та React Native framework. Системи та технології , 67 (1), 30-35. https://doi.org/10.32782/2521-6643-2024-1-67.5 (дата звернення: 24.12.2024).
2. Розробка мобільних додатків: Тенденції, які варто знати у 2023 URL: https://careers.easternpeak.com/blog/mobile-app-development-trends/ (дата звернення: 24.12.2024).
3. Redux Toolkit Documentation: https://redux-toolkit.js.org/introduction/getting-started (дата звернення: 24.12.2024).
4. Mobx documentation URL: https://mobx.js.org/README.html (дата звернення: 17.10.2024).
5. Xstate documentation URL: https://stately.ai/docs/xstate (дата звернення: 24.12.2024).
6. A New Approach for Implementing Finite State Machines with Input Multiplexing. MDPI. URL: https://www.mdpi.com/2079-9292/12/18/3763 (дата звернення: 24.12.2024).
7. Da Silva Ribeiro R., Lima de Carvalho R., Da Silva Almeida T. Optimization of state assignment in a finite state machine. Academic Journal on Computing, Engineering and Applied Mathematics. 2021. Vol. 3, no. 1. P. 9–16. URL: https://doi.org/10.20873/uft.2675-3588.2022.v3n1.p9-16 (дата звернення: 24.12.2024).
8. J Lojda, R Panek and Z Kotasek (2021).Automatically-designed fault-tolerantsystems: failed partitions recovery. 2021IEEE East-West Design & Test Symposium (EWDTS). IEEE, Available at : https://ieeexplore.ieee.org/document/9580996 (дата звернення: 24.12.2024)
9. Verma K. Research on Finite State Machine and Its Real Life Time Applications. Journal of Informational Technologies and Science. 2023. P. 10–20. URL:https://www.researchgate.net/publication/376857599_Research_on_Finite_State_Machine_and_Its_Real_Life_Time_Applications (дата звернення: 24.12.2024)

10. React Native Optimization Guide 2024 .P. 76-82 https://www.callstack.com/ebook/the-ultimate-guide-to-react-native optimization#ebook-hero (дата звернення: 24.12.2024)

11. Finite State Machines: An Introduction to FSMs and their Role in ComputerScience. SoftwareDominos.URL: https://softwaredominos.com/home/software-engineering-and-computer-science/finite-state-machines-an-introduction-to-fsms-and-their-role-in-computer-science/ (дата звернення: 24.12.2024).

12. Garcia-Vargas I., Senhadji-Navarro R. A New Approach for Implementing Finite State Machines with Input Multiplexing. Electronics. 2023. Vol. 12, no. 18. P. 3763. URL: https://doi.org/10.3390/electronics12183763 (дата звернення: 24.12.2024).

13. Da Silva Ribeiro R., Lima de Carvalho R., Da Silva Almeida T. Optimization of state assignment in a finite state machine. Academic Journal on Computing, Engineering and Applied Mathematics. 2021. Vol. 3, no. 1. P. 9–16. URL: https://doi.org/10.20873/uft.2675-3588.2022.v3n1.p9-16 (дата звернення: 24.12.2024).

14. Vladyslav Martynov Github URL: https://github.com/VladyslavMartynov10/Archiving (дата звернення: 05.06.2025).

15. Usachov, V., & Shubin, I. (2025). A structural-functional model of learning in computerized learning systems. INNOVATIVE TECHNOLOGIES AND SCIENTIFIC SOLUTIONS FOR INDUSTRIES, (1(31), 127–142. https://doi.org/10.30837/2522-9818.2025.1.127 .

16. Dolhyi, A., & Shubin, I. (2025). Research of modern methods and tools of ontological engineering in the context of creating intellectual systems. INNOVATIVE TECHNOLOGIES AND SCIENTIFIC SOLUTIONS FOR INDUSTRIES, (1(31), 32–48.https://doi.org/10.30837/2522-9818.2025.1.032.

17. Шубін, І., & Сотник, І. (2025). Метод алгебро-логічного моделювання предметних галузей для систем видобування знань. Herald of Khmelnytskyi National University. Technical Sciences, 349(2), 407-416. https://doi.org/10.31891/2307-5732-2025-349-59.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

15. Usachov, V., & Shubin, I. (2025). A structural-functional model of learning in computerized learning systems. *INNOVATIVE TECHNOLOGIES AND SCIENTIFIC SOLUTIONS FOR INDUSTRIES*, (1(31), 127–142. https://doi.org/10.30837/2522-9818.2025.1.127 .

16. Dolhyi, A., & Shubin, I. (2025). Research of modern methods and tools of ontological engineering in the context of creating intellectual systems. *INNOVATIVE TECHNOLOGIES AND SCIENTIFIC SOLUTIONS FOR INDUSTRIES*, (1(31), 32–48.https://doi.org/10.30837/2522-9818.2025.1.032.

17. Шубін, І., & Сотник, І. (2025). Метод алгебро-логічного моделювання предметних галузей для систем видобування знань. Herald of Khmelnytskyi National University. Technical Sciences, 349(2), 407-416. https://doi.org/10.31891/2307-5732-2025-349-59.