УДК 004.4`2 DOI https://doi.org/10.35546/kntu2078-4481.2025.4.3.13

М. Р. КАЛАШНІКОВ
студент магістратури кафедри комп'ютерних систем,
мереж і кібербезпеки
Національний аерокосмічний університет
«Харківський авіаційний інститут»
ORCID: 0009-0003-7774-9239

В. С. ХАРЧЕНКО
доктор технічних наук, професор,
завідувач кафедри комп'ютерних систем, мереж і кібербезпеки
Національний аерокосмічний університет
«Харківський авіаційний інститут»
ORCID: 0000-0001-5352-077X

ІНТЕЛЕКТУАЛЬНИЙ ПІДХІД ДО ПІДВИЩЕННЯ СТАБІЛЬНОСТІ СИСТЕМ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ НА БАЗІ НАВЧАННЯ З ПІДКРІПЛЕННЯМ

У дослідженні увага зосереджувалася на розробці інструменту ефективізації автоматизованої системи тестування із використанням інтелектуальної архітектури на базі парадигми навчання з підкріпленням, спрямованої на покращення стабільності тестування, причому Selenium застосовувалася в якості практичного прикладу. У роботі були визначені основні причини появи нестабільності в межах тестування графічних користувацьких інтерфейсів, а саме мінливість DOM-структур, асинхронна візуалізація, мережева затримка та ригідність синхронізації, з огляду на що була сформульована багаторівнена адаптивна модель, задача якої полягає в уникненні цих факторів за допомогою механізмів навчання з підкріпленням. Запропонований Інтелектуальний рівень з підкріпленням передбачає реструктуризацію класичної ієрархії Selenium, тобто IDE, Client–Server, та Grid в об'єднану структуру із самонавчанням, здатну проводити автономне підлаштування до мінливих параметрів системи. В межах цього рівня, Агент підкріплення IDE виконує локальну адаптацію шляхом регуляції інтервалів відтворення та вибору локаторів у відповідь на структурні зміни у DOM. Адаптивний канал Client–Server, що включає Агенти стабілізації виконання та ієрархічної адаптації локаторів, забезпечує стабільність виконання тестових сценаріїв за рахунок підлаштування інтервалів синхронізації, політики очікування та стратегій відновлення структури локаторів, виходячи з телеметричних даних у реальному часі. На рівні розподіленого контролю знаходиться Середовище навчання Grid, в якому застосовується кооперативне багатоагентне навчання, спрямоване на врівноваження тестових навантажень на всіх вузлах, в ході чого відбувається оптимізація загальносистемної стабільності та кількості тестів, які система може виконати за одиницю часу. Порівняльний аналіз показав, що в той час, як традиційні інструменти, такі як Cypress та Playwright, реалізовують ригідні процедури стабілізації, запропонована архітектура уособлює досягнення адаптивізації та стабілізації шляхом постійного навчання та аналізу телеметричного зворотного зв'язку, що сприяє спрощеному проведенню сталого самооптимізаційного тестування.

Ключові слова: *штучний інтелект, стабільність тестів, автоматизація, QA, адаптивні алгоритми.*

M. R. KALASHNIKOV
Master's Student at the Department of Computer Systems,
Networks and Cybersecurity
National Aerospace University "Kharkiv Aviation Institute"
ORCID: 0009-0003-7774-9239

V. S. KHARCHENKO
Doctor of Technical Sciences, Professor,
Head of the Department of Computer Systems, Networks and Cybersecurity
National Aerospace University "Kharkiv Aviation Institute"
ORCID: 0000-0001-5352-077X

INTELLIGENT APPROACH TO ENHANCING THE STABILITY OF AUTOMATED TESTING SYSTEMS BASED ON REINFORCEMENT LEARNING

This research develops and substantiates an intelligent reinforcement-based architecture aimed at enhancing the stability of automated testing systems, using Selenium as a practical foundation. The study identifies the root causes of instability in UI testing – DOM volatility, asynchronous rendering, network latency, and rigid synchronization – and formulates a multilayered adaptive model that mitigates these factors through reinforcement learning mechanisms. The proposed Intelligent Reinforcement Layer restructures Selenium's classical hierarchy, namely IDE, Client–Server, and Grid, into a unified self-learning framework capable of autonomous adjustment to environmental drift. Within the IRL, the IDE Reinforcement Agent performs local adaptation by regulating playback timing and locator selection in response to structural changes in the DOM. The Client–Server Adaptive Layer, comprising the Execution Stability Agent and Hierarchical Locator Adaptation Agent, ensures execution stability by adjusting synchronization intervals, wait policies, and locator recovery strategies based on real-time telemetry. At the distributed level, the Grid Learning Environment applies cooperative multi-agent learning to balance test workloads across nodes, optimizing system-wide stability and throughput. The research formalizes unified reinforcement-learning logic, defines state–action structures for each subsystem, and introduces a collective experience buffer for cross-layer training. Comparative analysis demonstrates that while traditional frameworks such as Cypress and Playwright implement fixed stabilization procedures, the proposed IRL architecture achieves adaptive resilience through continuous learning and feedback. The study concludes that integrating reinforcement learning into Selenium's architecture yields a self-optimizing testing environment capable of maintaining long-term stability, reducing flakiness, and improving the robustness of automated QA processes under evolving application conditions.

Key words: *artificial intelligence, test stability, QA automation, adaptive algorithms.*

Постановка проблеми

Автоматизація тестування набула критично важливого значення в контексті сучасної парадигми розробки програмного забезпечення, створивши підгрунтя для підтримки якості програмних продуктів на належному рівні та прискорення циклів розробки в межах парадигми безперервної інтеграції та розгортання. Тим не менш, стабільність систем автоматизованого тестування, особливо в контексті тестування користувацьких інтерфейсів (user interface – UI), залишається одним з актуальних проблемних аспектів процесу забезпечення якості програмних продуктів. Нестабільність тестів проявляється у нерегулярності їхніх результатів (flaky tests) за умов повної незмінності вихідного коду.

Головна причина нестабільності автоматизації на базі графічних користувацьких інтерфейсів тісно пов'язана з їхньою динамічністю та непередбачуваною природою. Навіть за умов незначної зміни в інтерфейсній розмітці, асинхронної візуалізації, мережевої затримки, або затримок у оновленнях структур веб-сторінки (тобто у документній об'єктній моделі, Document Object Model – DOM) можуть виникнути хибні негативні результати. Традиційні автоматизаційні фреймворки, такі як Selenium та Cypress, передбачають застосування попередньо визначених елементних локаторів та статичної часової регуляції, що як правило не оновлюються по мірі оновлення користувацького інтерфейсу.

Відтак, стале підвищення складності веб-застосунків та потреба у безперервному розгортанні вимагають поєднання адаптивних та інтелектуальних механізмів в автоматизовані системи тестування. Технології машинного навчання характеризуються наявністю необхідних інструментів для вирішення окресленої проблеми нестабільності шляхом забезпечення можливості самовиправлення тестів (self-healing tests), виявлення аномалій та синхронізації виходячи з даного контексту. За допомогою інтелектуального моделювання, автоматизована система здатна аналізувати дані виконання тестових сценаріїв, ідентифікувати закономірності нестабільностей та автономно підлаштовуватися до зовнішніх змін у UI або середовищі розробки. Відтак, у цьому дослідженні розглядається інтелектуальний підхід удосконалення стабільності автоматизованих систем на прикладі інструменту для автоматизації операцій у веб-браузері Selenium, який також використовується для тестування веб-застосунків.

Аналіз останніх досліджень і публікацій

У роботі [1] оцінюється можливість використання методів машинного навчання для передбачення нестабільних тестів у Python-проектах. Досліджується, чи можуть моделі навчання під наглядом виявляти нестабільність ще до додавання тесту в набір. Було реалізовано та порівняно три класифікатори, а саме Naive Bayes (Наївний баєсів класифікатор), SVM (Support vector machine, метод опорних векторів) і Random Forest (Випадковий ліс), на предмет "запахів коду" ("test smells"), отриманих із коду після фільтрації й нормалізації токенів. Для оцінки використовувалися різні співвідношення даних для тренування/тестування (50/50–90/10) і метрики точності, повноти, F1 та ROC (receiver operating characteristic, робоча характеристика приймача). Результати показали, що Random Forest має найвищу точність (>90%), але низьку повноту (<10%), тоді як Naive Bayes і SVM характеризуються більш збалансованими показниками. Незважаючи на перевагу RF у точності, SVM краще виявляє ширші закономірності виникнення нестабільностей. Подальший аналіз показав, що асинхронні очікування, випадковість,

обчислення з плаваючою точкою та I/O-операції є сильними індикаторами, тоді як багатопоточність і часові залежності – менш репрезентативними. Підсумовано, що RF є ефективним для високоточної детекції, але для поліпшення повноти майбутні дослідження мають враховувати чинники на рівні інфраструктури тестування та контексту.

У дослідженні [2] було проаналізовано 345 проектів, які використовують Pyro, PyMC3, TensorFlow Probability та PyTorch, і виявлено 75 звітів про помилки та зміни у коді у 20 проектах, що стосувалися виправлення нестабільних тестів. Після аналізу вручну було визначено основні причини: алгоритмічний недетермінізм, похибки з плаваючою точкою та непослідовне використання випадкових початкових значень (сідів, seed). Зазвичай розробники усувають ці проблеми через коригування порогових значень перевірок, фіксацію сідів або видалення нестабільних тестів. На основі цих висновків запропоновано метод FLASH – статистичний підхід, що передбачає виявлення нестабільних тестів, спричинених випадковими процесами виконання. FLASH багаторазово запускає тести з різними сідами і застосовує діагностику збіжності (тест Гевеке) для визначення моменту, коли отримано достатньо вибірок для статистичного висновку. Далі оцінюється ймовірність збоїв перевірок за рахунок моделювання розподілів спостережуваних значень, і виявлення тестових сценаріїв, результати яких змінюються між виконаннями. Метод було перевірено на 20 відкритих проектах, де за допомогою нього було знайдено 11 нових нестабільних тестів (10 підтверджено розробниками), а також успішно відтворено раніше відомі випадки. Результати демонструють, що залучення статистичних підходів у тестування підвищує надійність програм, у яких результати мають випадковий характер.

У рамках роботи [3] представлено FlakyCat, що є методом класифікації нестабільних тестів за їхніми причинами з використанням Few-Shot навчання (за невеликої кількості екземплярів). FlakyCat передбачає використання попередньо навченої моделі CodeBERT для представлення коду і сіамську нейромережу для навчання класифікації з обмеженою кількістю розмічених даних. Модель оптимізується за допомогою функції втрат triplet-loss, щоб розмістити тести у просторі подібності, де приклади одного типу нестабільності групуються. Було створено набір із 451 нестабільного тесту з 13 категорій, який після аугментації містив 964 зразки. Підхід оцінювався відносно класичних методів (Random Forest, SVM, KNN (k-nearest neighbors – k-найближчі сусіди), Decision Tree (Дерево прийняття рішень)) і традиційних ознак (лексичних, на основі запахів коду) за метриками F1, MCC (Matthews Correlation Coefficient (коефіцієнт кореляції Метьюза), AUC (area under ROC curve, площа під ROC-кривою). FlakyCat досяг найвищих результатів (F1 = 0.73, MCC = 0.65, AUC = 0.83), перевершивши всі базові методи. Аналіз за категоріями показав, що тести з асинхронними очікуваннями, невпорядкованими вибірками та часовими залежностями легше ідентифікувати, ніж тести, пов’язані з багатопоточністю. Додатково запропоновано підхід на основі CodeBERT, який виділяє ключові оператори коду, що впливають на рішення моделі. Робота доводить, що Few-Shot навчання з семантичними вкладеннями коду може ефективно класифікувати типи нестабільності та допомагати у їх усуненні.

У межах дослідження [4] був представлений IDFLAKIES – метод виявлення нестабільних тестів і класифікації їх як NOD (Non-Order-Dependent, тест, збій якого не залежить від порядку виконання) або victim (тест, “постраждалий” від порядку виконання). Підхід складається з трьох етапів: встановлення, виконання та класифікації. На першому етапі тести повторно запускаються у звичному порядку для відфільтрування стабільно помилкових випадків. На другому – виконуються у зміненому порядку. Під час класифікації кожен тест, що зазнав збою, повторно запускається як у звичному, так і у модифікованому порядку до точки збою. Якщо тест знову завершується збоєм під час виконання у зміненому порядку, але проходить у звичному, його класифікують як victim; у протилежному випадку– як NOD. У разі кількох збоїв процедура може повторюватися для підвищення достовірності. Метод враховує такі параметри як кількість повторів і спосіб перемішування тестів, що може значно збільшити час виконання.

Варто також звернути уваги на роботу [5], де був представлений FLAKEFLAGGER – інструмент для виявлення нестабільних тестів на основі моделі машинного навчання. Для кодування тестів використано набір із 8 числових і 8 булевих ознак, що позначали наявність запахів коду, однак пізніше булеві ознаки вже не враховувалися через низьку інформативність. У дослідженні 24 Java-проектів модель досягла значення MCC = 0.65. Був також представлений розширений набір FLAKE16, який включав додаткові метрики, як наприклад кількість I/O-операцій файлової системи та пікове споживання пам’яті. Оцінювання на 26 Python-проектах показало, що FLAKE16 стабільно перевищує FLAKEFLAGGER за ефективністю виявлення як NOD, так і victim тестів.

Головним внеском роботи [6] необхідно виділити представлення та емпіричну оцінку CANNIER – гібридної системи виявлення нестабільних тестів, яка поєднує підходи повторного запуску та машинного навчання. CANNIER використовує ймовірності нестабільності, передбачені моделлю машинного навчання, як припущення для зменшення простору повторних виконань. Оцінювання на 89 668 тестах у 30 Python-проектах (з використанням стратифікованої 10-кратної крос-валідації та MCC) показало, що CANNIER зменшив витрати часу до 88%, зберігаючи продуктивність (MCC ≈ 0.53–0.65). При застосуванні до класифікаційного етапу, алгоритмом

IDFLAKIES досягнуто скорочення часу на 84% при мінімальній втраті точності. Аналіз також показав, що комбінація середніх значень динамічних ознак тестів покращує роботу моделі, а такі характеристики, як багатопоточність, I/O-навантаження та використання пам'яті, тісно корелюють із ймовірністю нестабільності тестових сцеаріїв.

У цій роботі [7] запропоновано FlaKat – систему машинного навчання для класифікації нестабільних тестів відповідно до їхніх першопричин. З використанням міжнародного набору даних International Dataset of Flaky Tests (IDoFT), проводиться аналіз різних стратегій представлення та навчання. Розглянуто три методи векторизації – TF-IDF (TF – term frequency, IDF – inverse document frequency), doc2vec і code2vec – а також методи зменшення розмірності (LDA (Latent Dirichlet Allocation), PCA (Principal Component Analysis), UMAP (Uniform Manifold Approximation and Projection)) для оцінки збереження структури у просторі ознак. Для розв'язання проблеми дисбалансу даних застосовано вибіркові методи SMOTE і Tomek Link, а для класифікації протестовано моделі KNN, SVM і Random Forest. Експериментальні результати показують, що поєднання TF-IDF і Random Forest забезпечує найвищу точність класифікації (F1 = 0.67). Додатково введено нову метрику здатності моделі виявляти нестабільні тести (Flakiness Detection Capacity – FDC), яка оцінює узгодженість класифікатора з використанням інформаційно-теоретичного аналізу. Вона виявилася більш стабільною, ніж F1, що підкреслює її ефективність для оцінювання систем виявлення нестабільних тестів.

Проте, беручи до уваги вище зазначену наукову документацію, питання, пов'язане з розробка інтелектуального підходу удосконалення стабільності автоматизованих систем тестування, все ще залишається недостатньо дослідженим та потребує подальшого опрацювання.

Формулювання мети дослідження

Метою роботи є розробка інтелектуального підходу удосконалення стабільності автоматизованих систем тестування на прикладі інструменту для автоматизації операцій у веб-браузері Selenium із використанням навчання з підкріплення.

Викладення основного матеріалу дослідження

Selenium представляється в якості структурованого багаторівневого автоматизаційного фреймворку, що робить взаємодію між розроблюваними тестовими сценаріями та тестованими веб-застосунками більш прямою. Його модульна архітектура складається з 4 головних компонентів, а саме Selenium IDE (Integrated Development Environment – інтегроване середовище розробки), Selenium Client (WebDriver), Selenium Server, and Selenium Grid, що разом формують ієрархічний цикл керування, тобто IDE → Client → Server → Grid [8].

Відтак, на базовому рівні знаходиться Selenium IDE, що функціонує в якості середовища реєстрації та відтворення тестів. Ця ланка дозволяє враховувати взаємодії на рівні користувачів, генеруючи тестові сценарії які надалі можуть бути відтвореними для регресійного тестування. Незважаючи на це, цей рівень залежить від статично записуваних локаторів та послідовностей інструкцій.

Над IDE розташовується Selenium Client (WebDriver), в якому регулюються логічні взаємозв'язки між тестовими сценаріями та веб-застосунком. Відтак, відбувається переклад інструкцій користувачів в операції на рівні браузера за допомогою WebDriver API та інтерфейсу RemoteWebDriver. Варто зауважити, що цей взаємозв'язок є синхронізованим та послідовним, тобто кожна інструкція може бути виконана тільки після отримання відгуку.

Selenium Server виконує роль проміжного рівня, що зв'язує Client з драйверами браузера. За посередництва нього, вхідні команди встановлюються у чергу, які потім надсилаються до відповідних браузерних інстансів (instance – екзмепляр). Втім, він також функціонує детерміністичним чином, не проводячи аналітичне спостереження за метриками продуктивності, або виникненням аномалій у відгуках. Внаслідок цього, навіть за умови виявлення неналежних станів на рівнях Client та IDE, цей рівень залишатиметься позбавленим необхідної адаптивності.

Selenium Client та Selenium Server функціонують у поєднанні, створюючи Клієнтсько-серверний канал, що функціонує в якості базового комунікаційного інструмента Selenium. Client надсилає інструкції через WebDriver API, в той час, як Server виконує ці інструкції за допомогою Application Driver API.

Найвища ланка в окресленій ієрархічній структурі, тобто Selenium Grid, необхідна для проведення розподіленого та паралельного виконання тестових сценаріїв на декількох вузлах. Ця ланка працює на базі центрально-вузлової архітектури (hub-and-node), що забезпечує розширюваність тестового середовища, однак позбавлена аналітичної здатності під час виконання тестів. Розподіл задач залежить суто від попередньо визначених налаштувань вузлів та статичних налаштувань планування.

Загалом, ці рівні формують організований стек детермінованих процесів, тобто будь-яке очікування, блокування та виконання відбувається без огляду на контекстну динаміку роботи системи. Отже, кожний рівень має наділятися виділеним адаптивним підмодулем, здатним враховувати локальні порушення стабільності та відповідати на них, тобто агент з підкріпленням для IDE, призначений для адаптації часових параметрів відтворення команд, агент-стабілізатор виконання в межах каналу Client–Server для регуляції синхронізації, та координаційний агент для Grid з метою врівноваження розподілених обчислювальних задач між вузлами [9].

Відтак, Інтелектуальний рівень з підкріпленням (Intelligent Reinforcement Layer – IRL) слугує в якості мета-архітектури, що охоплює всі традиційні складові Selenium в єдину адаптивну систему. На базовому рівні, в межах IRL встановлюється пул колективного досвіду, який збирає телеметричні дані з усіх ланок ієрархічної структури Selenium. До цих телеметричних даних насамперед входять час затримки взаємодії з браузером, частота змін DOM, показники ідентифікаційної точності локаторів, рівні використання ресурсів вузлів та коефіцієнти успішності виконання.

Структура IRL передбачає наявність 3 підрівнів адаптації:

– локальна адаптація на рівні IDE, де проводиться тонке налаштування часових параметрів інструкцій та відновлення старих прив'язок до локаторів з огляду на мінливість DOM;

– середньорівнева адаптація в межах каналу Client–Server, де відбувається динамічне налаштування інтервалів очікування, стратегій синхронізації та зміна локаторів відповідно до вихідної інформації після виконання тесту;

– глобальна адаптація на рівні Grid, в контексті якої виконується збалансування робочого навантаження вузлів задля того, щоб стабілізаційні заходи, виявлені локально, поширювалися по всій системі.

Загальна процедура навчання всіх агентів в рамках IRL проводиться згідно з формулюванням навчання з підкріпленням (reinforcement-learning – RL). Функціонування кожного агента A_i формулюється, враховуючи пару стан-операція (s_i , a_i), де в якості стану розглядається робочі умови, контекстно пов'язані з рівнем агента, а операція відповідає конкретному коригувальному заходу в межах системи. Загальний принцип оновлення на базі Q-навчання формулюється таким чином:

$$Q_{t+1}(s_i, a_i) = Q_t(s_i, a_i) + \alpha\left[R_t + \beta \max_{a'} Q_t(s_i', a') - Q_t(s_i, a_i)\right] \quad (1)$$

де α відповідає коефіцієнту навчання, що визначає ступінь впливу нових вивчених закономірностей на оновлення моделі, а β позначає коефіцієнт дисконтування, що регулює співвідношення між поточними та майбутніми винагородами, s_i' відображає наступний спостережуваний стан після застосування операції a_i , а R_t є скалярною функцією винагороди, що відображає загальне покращення стабільності.

Об'єднана функція винагороди виражається так:

$$R_t = \mu_1 - \mu_2 - \mu_3 \quad (2)$$

де коефіцієнти μ_1 , μ_2 , μ_3 відображають динамічні вагові коефіцієнти, що відповідають ступеням точності, логічної узгодженості та продуктивності, відповідно. Ці вагові коефіцієнти змінюються під час проведення тестування за посередництва мета-навчання. Кожна підсистема спрямована на налаштування цієї базової логіки відповідно до свого функціонального контексту та області видимості.

Перший підмодуль, Агент підкріплення IDE (IDE Reinforcement Agent – IDEA) визначає принцип виконання зареєстрованих тестових операцій, фактично зосереджуючись на виявленні аномалій під час виконання тестових сценаріїв та подальшій автономній модифікації часових параметрів виконання операцій, вибору локаторів та стратегій повторного відтворення.

На часовому кроці t , IDEA розглядає тестове середовище в якості свого локального вектора станів, визначеного так:

$$s_t = \{t_{\text{навантаж}}, \delta_{\text{мінл}}, r_{\text{лок}}\} \quad (3)$$

де $t_{\text{навантаж}}$ відповідає зафіксованому часу навантаження сторінки або часу візуалізації елементу, $\delta_{\text{мінл}}$ позначає ступінь мінливості DOM, що відображає структурні зміни, або зміни в атрибутах з моменту останнього виконання, а $r_{\text{лок}}$ є балом точності елементного локатора, зафіксованого в даний момент часу, що обчислюється на основі ступеня збігу, або частоти успішних виконань за весь час.

Регуляційна політика IDEA формується на основі модифікованої функції винагород:

$$R_t = \gamma_1(1 - \delta_{\text{мінл}}) + \gamma_2 r_{\text{лок}} - \gamma_3 t_{\text{навантаж}} \quad (4)$$

де вагові коефіцієнти γ_1 , γ_2 , γ_3 визначають відносну важливість структурної стабільності, точності на рівні локаторів та ефективності.

Враховуючи окреслене формулювання, IDEA набуває здатності знижувати час зайвого очікування, при цьому уникаючи передчасних або помилкових взаємодій. З часом ця підсистема досягає рівноваги між швидкістю виконання та стабільністю. У процесі навчання агент поступово формує таблицю Q-значень, де кожне Q-значення відображає очікувану корисність вибору певної дії у конкретному стані середовища. В остаточному підсумку, Q-значення оновлюються згідно зі згаданим вище правилом оновлення.

На адаптивному рівні каналу Client–Server відбувається регуляція стабільності виконання Selenium в реальному часі, функціонуючи між WebDriver API та Application Driver API. Отже, регуляція стабільності проводиться шляхом керування таких нечітких параметрів та сутностей, як ступінь мережевої затримки, асинхронні відгуки

на інструкції, або затримка синхронізації драйвера. З огляду на це, в межах структури цього рівня передбачена наявність двох кооперативних агентів. а саме Агента стабільності виконання (Execution Stability Agent – QESA) та Агента ієрархічної адаптації локаторів (Hierarchical Locator Adaptation Agent – HLAA).

QESA перехоплює телеметричні дані, в даному випадку ступені затримки запитів, часи відгуку DOM та тимчасові затримки підтвердження зі сторони сервера, та надсилає їх до колективного пулу досвіду. У той же час, HLAA функціонує у поєднанні з QESA, проводячи спостереження за локаторною точністю ідентифікації потрібного елемента в інтерфейсі за різних умов під час виконання тестового сценарію. У випадку помилкової ідентифікації локатором, або перевищення часу очікування, за рахунок HLAA використовуються дані з колективного пулу для динамічного перевибирання, або реконструкції елементних локаторів виходячи з вивчених стратегій відновлення структури, а саме, наприклад, ступеня контекстної спорідненості DOM, або метрик подібності атрибутів.

RL логіка клієнтсько-серверного каналу передбачає розширення функції винагород за рахунок додаткових системних та комунікаційних метрик. Вектор станів у часі t визначається так:

$$s_t = \left\{ \tau_{\text{інстр}}, \lambda_{\text{мереж}}, \sigma_{\text{синх}}, \rho_{\text{локг}} \right\} \tag{5}$$

де $\tau_{\text{інстр}}$ є середнім часом відгуку на інструкції, $\lambda_{\text{мереж}}$ – мережевою або пакетною затримкою, $\sigma_{\text{синх}}$ – зміщеннями синхронізації між відгуками клієнта та сервера, а $\rho_{\text{локг}}$ – індексом точності поточно активних локаторів, відповідно до даних з HLAA.

Колективна функція винагороди агента формулюється так:

$$R_t = \mu_1 - \mu_2 - \mu_3 - \mu_4 \tag{6}$$

де μ_4 реалізує штраф до нестабільної синхронізації та затриманих підтверджень.

Обидва агенти оновлюють свої внутрішні Q-таблиці на основі досвіду з колективного пулу даних та даних IDEA, відповідно до усталеного формулювання. Варто зазначити, що Q-значення з IDEA виконують роль локальних апріорних даних, що спрямовують початкове налаштування політики обох агентів каналу. Тобто ознаки роботи системи, що забезпечує її стабільність, навчені під час відтворення тестових сценаріїв на першій ланці, безпосередньо впливають на прийняття рішень під час виконання.

З часом, QESA та HLAA спільно оптимізують виконання 3 адаптивних механізмів, а саме:

– корекція інтервалу очікування, в ході якого відбувається динамічна модифікація явних і неявних очікувань відповідно до зафіксованих параметрів затримок виконання інструкцій та індикаторів стану DOM;

– калібрування політики повторного відтворення, де проводиться вивчення оптимальної частоти повторних відтворень при тимчасових збоях, запобігаючи як зайвим очікуванням, так і передчасним завершенням тестових сценаріїв або окремих операцій взаємодії з інтерфейсом;

– автоматична зміна локаторів відповідно до різних стратегій пошуку елементів (XPath, CSS-селектори, ID, текстові атрибути).

З концептуальної точки зору, клієнтсько-серверний рівень проводить зв'язок між адаптацією на мікрорівні (IDEA) з координацією на макрорівні (до Grid).

Багатоагентний координаційний рівень Grid є ланкою найвищої абстракції IRL, що розширює адаптивність Selenium для роботи в багатовузлових інфраструктурах тестування, що у цьому дослідженні має назву Середовища навчання Grid (Grid Learning Environment – GLE). В межах цієї архітектури, кожний вузол Selenium Grid виконує функцію автономного агента RL, здатного враховувати свій локальний робочий стан та надсилати необхідну інформацію до контролера взаємодії між агентами π_H.

Вектор станів кожного вузла формулюється так:

$$s_i = \left\{ L_i, T_i, F_i \right\} \tag{7}$$

де L_i є середньою затримкою мережі, або виконання інструкції в межах вузла i, T_i є кількістю успішно виконаних тестових сценаріїв за одиницю часу, а F_i є локальним коефіцієнтом нестабільності, тобто часткою нестабільних, або провалених тестових сценаріїв за останній час.

На цьому рівні визначається глобальна функція винагород, яка збирає локальну інформацію з усіх вузлів N:

$$R_{\text{глоб}} = \frac{1}{N} \sum_{i=1}^{N} (\mu_1 S_i - \mu_2 L_i - \mu_3 F_i) \tag{8}$$

де S_i відображає кількість успішних виконань тестових сценаріїв в межах вузла i.

В той час, як оновлення Q-значень відбувається за стандартною процедурою, на найвищому рівні структури GLE знаходиться, як було згадано, контролер взаємодії між агентами π_H, що визначає принцип розподілу тестових операцій по вузлах. Для цієї задачі застосовуються політики вибору на базі функції softmax, виходячи з навчених Q-значень:

$$\pi_H(A_i \mid S) = \frac{e^{\kappa Q_i(S_i)}}{\sum_{j=1}^{N} e^{\kappa Q_j(S_j)}} \tag{9}$$

де к відповідає температурному параметру, що регулює співвідношення між виявленням нових закономірностей та використанням старих.

Відтак, описана архітектура оглядово наведена на рисунку 1

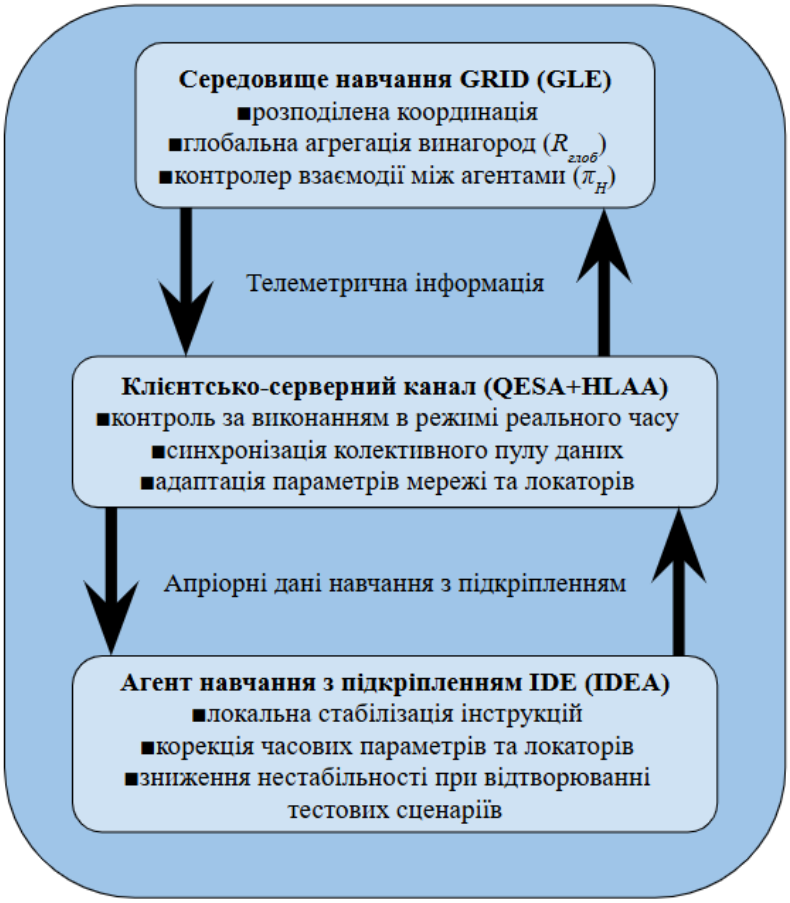

Рис. 1. Загальна архітектура теоретичної моделі

Варто зазначити, що така структура, де кожний вузол незалежно покращує свою ефективність за посередництва навчання з підкріпленням, однак де вони всі мають колективну задачу досягнення глобальної оптимізації за рахунок контролера, є аналогічною до кооперативної навчальної поведінки систем ройового інтелекту (swarm intelligence).

Впровадження мета-архітектури IRL у структуру Selenium має низку концептуальних переваг над сучасними фреймворками автоматизації. По-перше, на відміну від класичного Selenium WebDriver, який не забезпечує вбудованих механізмів розумного очікування або автоматичних повторних виконань операцій, система з IRL самостійно здатна регулювати синхронізацію та відновлення етапів тесту [10]. По-друге, такі інструменти, як Cypress та Playwright, вже впроваджують деякі засоби підвищення стабільності (автоматичне очікування елементів, регуляція повторів тестових сценаріїв). Проте в Cypress і Playwright ці можливості реалізовані як фіксовані алгоритми, тоді як IRL застосовує адаптивне навчання, тобто агенти підлаштовують стратегії під поведінку конкретного застосунку та продовжують оптимізувати їх з накопиченням досвіду. Таким чином, IRL потенційно перевершує зазначені фреймворки у здатності реагувати на нові типи змін середовища, які не були явно передбачені розробниками інструмента. По-третє, IRL-архітектура зберігає широку сумісність Selenium (підтримку різних мов програмування, браузерів і платформ), одночасно додаючи інтелектуальні можливості. Для порівняння, Cypress є обмеженим JavaScript, а його архітектура накладає певні обмеження (наприклад, відсутність повноцінної підтримки багатовіконності), тоді як IRL може бути інтегрований у вже наявні різномовні фреймворки на базі Selenium.

Окремо слід відзначити інструмент Percy, що спеціалізується на візуальному тестуванні. Percy використовує рушій комп’ютерного зору для виявлення візуальних відхилень в інтерфейсі користувача шляхом аналізу структур

DOM [11]. Це є корисним для контролю регресій зовнішнього вигляду веб-застосунку, проте Percy не вирішує проблем функціональної стабільності тестів, як наприклад, непередбачуваних збоїв через динамічні зміни на сторінках. Запропонований підхід, навпаки, зосереджений на функціональній надійності тестування: його метою є забезпечення того, щоб автоматизовані тести стабільно проходили при відсутності помилок у самому додатку. Отже, IRL та Percy можна розглядати як взаємодоповнювальні засоби забезпечення якості: IRL підтримує коректність і стабільність виконання тестових сценаріїв, тоді як Percy додатково перевіряє відсутність небажаних візуальних змін.

В цілому, результати теоретичного дослідження свідчать, що запропонована багаторівнева IRL-архітектура здатна підвищити стабільність систем автоматизованого тестування. Вона робить тестові сценарії більш адаптивними до змін у додатку та загальній інфраструктурі тестування, зменшує кількість хибних тестових сценаріїв і знижує потребу в ручному налаштуванні тестів. Це вигідно відрізняє підхід IRL від наявних рішень і підкреслює його потенціал як інтелектуального засобу підвищення надійності автоматизованого тестування.

Висновки

Порівняльний аналіз показав, що в той час, як традиційні інструменти, такі як Cypress та Playwright, реалізовують ригідні процедури стабілізації, запропонована архітектура уособлює досягнення адаптивізації та стабілізації шляхом постійного навчання та аналізу телеметричного зворотного зв'язку, що сприяє спрощеному проведенню сталого самооптимізаційного тестування. У майбутніх дослідженнях доцільно зосередитись на практичній реалізації запропонованої архітектури IRL та її оцінюванні у реальних умовах розробки. Подальший розвиток може включати впровадження методів глибокого навчання з підкріпленням, таких як Deep Q-Network або Actor–Critic, для роботи з більш складними станами в інфраструктурах тестування. Перспективним напрямом є також інтеграція IRL у пайплайни CI/CD для забезпечення безперервного навчання на основі даних тестових прогонів у реальному часі.

Список використаної літератури

1. Ahmad A., Leifler O., Sandahl K. An Evaluation of Machine Learning Methods for Predicting Flaky Tests. Linköping University, Sweden. DiVA Portal, 2020. URL: https://www.diva-portal.org/smash/get/diva2:1537340/FULLTEXT01.pdf

2. Dutta S., Shi A., Choudhary R., Zhang Z., Jain A., Misailovic S. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. Cornell University, 2020. 14 p. URL: https://saikatdutta.web.illinois.edu/papers/flash-issta20.pdf

3. Akli A., Haben G., Habchi S., Papadakis M., Le Traon Y. FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning. University of Luxembourg, 2023. 12 p. URL: https://orbilu.uni.lu/bitstream/10993/55848/1/FlakyCat.pdf

4. Lam W., Oei R., Shi A., Marinov D., Xie T. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019. P. 312–322. DOI: 10.1109/ICST.2019.00038

5. Alshammari A., Morris C., Hilton M., Bell J. FlakeFlagger: Predicting Flakiness without Rerunning Tests. Proceedings of the International Conference on Software Engineering (ICSE), 2021.

6. Parry O., Kapfhammer G. M., Hilton M., McMinn P. Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models. Empirical Software Engineering. 2023. Vol. 28. No. 72. DOI: 10.1007/s10664-023-10307-w

7. Lin S., Liu R., Tahvildari L. FlaKat: A Machine Learning-Based Categorization Framework for Flaky Tests, 2024. arXiv:2403.01003 [cs.SE]. URL: https://arxiv.org/pdf/2403.01003v1

8. Nyamathulla S., Ratnababu P., Shaik N. S. A Review on Selenium Web Driver with Python. Annals of the Romanian Society for Cell Biology, 2021. P. 16760–16768.

9. Alferidah S. K., Ahmed S. Automated Software Testing Tools. 2020. Proceedings of the International Conference on Computing and Information Technology (ICCIT-1441). IEEE.

10. Ulili S. Playwright vs Puppeteer vs Cypress vs Selenium (E2E Testing). 2025. Better Stack Community Comparisons. URL: https://betterstack.com/community/comparisons/playwright-cypress-puppeteer-selenium-comparison/

11. Revolutionizing Visual Testing on Web using Automation and AI: Halodoc's Journey with Percy. 2024. Percy Visual Testing Blog, Halodoc. URL: https://blogs.halodoc.io/percy-web/

References

1. Ahmad, A., Leifler, O., & Sandahl, K. (2020). *An evaluation of machine learning methods for predicting flaky tests*. Linköping University. https://www.diva-portal.org/smash/get/diva2:1537340/FULLTEXT01.pdf

2. Dutta, S., Shi, A., Choudhary, R., Zhang, Z., Jain, A., & Misailovic, S. (2020). *Detecting flaky tests in probabilistic and machine learning applications* (14 pp.). Cornell University. https://saikatdutta.web.illinois.edu/papers/flash-issta20.pdf

3. Akli, A., Haben, G., Habchi, S., Papadakis, M., & Le Traon, Y. (2023). *FlakyCat: Predicting flaky tests categories using few-shot learning* (12 pp.). University of Luxembourg. https://orbilu.uni.lu/bitstream/10993/55848/1/FlakyCat.pdf

4. Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019). iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 IEEE 12th International Conference on Software Testing, Verification and Validation (ICST)* (pp. 312–322). https://doi.org/10.1109/ICST.2019.00038

5. Alshammari, A., Morris, C., Hilton, M., & Bell, J. (2021). *FlakeFlagger: Predicting flakiness without rerunning tests*. Proceedings of the International Conference on Software Engineering (ICSE).

6. Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2023). Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering, 28*(72). https://doi.org/10.1007/s10664-023-10307-w

7. Lin, S., Liu, R., & Tahvildari, L. (2024). *FlaKat: A machine learning-based categorization framework for flaky tests*. arXiv:2403.01003. https://arxiv.org/pdf/2403.01003v1

8. Nyamathulla, S., Ratnababu, P., & Shaik, N. S. (2021). A review on Selenium Web Driver with Python. *Annals of the Romanian Society for Cell Biology*, 16760–16768.

9. Alferidah, S. K., & Ahmed, S. (2020). Automated software testing tools. In *Proceedings of the International Conference on Computing and Information Technology (ICCIT-1441)*. IEEE.

10. Ulili, S. (2025). *Playwright vs Puppeteer vs Cypress vs Selenium (E2E Testing)*. Better Stack Community. https://betterstack.com/community/comparisons/playwright-cypress-puppeteer-selenium-comparison/

11. Halodoc. (2024). *Revolutionizing visual testing on web using automation and AI: Halodoc's journey with Percy*. Percy Visual Testing Blog. https://blogs.halodoc.io/percy-web/

Дата першого надходження рукопису до видання: 23.11.2025
Дата прийнятого до друку рукопису після рецензування: 19.12.2025
Дата публікації: 31.12.2025