Хмельницький національний університет
Факультет Інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА МАГІСТРА

Удосконалення методу виявлення помилок при тестуванні програмного забезпечення на основі трасування стеку

Рівень вищої освіти Другий (магістарський)
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітня програма Освітньо-професійна програма інженерія програмного забезпечення

ДРІПЗ 2001112.20.02.05 ПЗ

Виконав: студент 2 курсу, група ІПЗм-20-2 ____ підпис І.В. Гурман Ініціали прізвище

Керівник д-р фіз.-мат. наук, проф. ____ підпис Л.П. Бедратюк Ініціали прізвище

Нормоконтроль к.т.н., доцент ____ підпис Ю.В. Форкун Ініціали прізвище

До захисту допускаю:
Зав. кафедри ІПЗ д-р фіз.-мат. наук, проф. ____ Л.П. Бедратюк
6 грудня 2021 р.

Хмельницький 2021р.

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп’ютерних і телекомунікаційних систем
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
01 09 2021 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Гурману Івану Васильовичу
Прізвище, ім’я, по батькові студента

1. Тема проєкту (роботи) Удосконалення методу виявлення помилок при тестуванні програмного забезпечення на основі трасування стеку

Керівник проєкту (роботи) Бедратюк Леонід Петрович
Доктор фіз.-мат. наук, професор

Затверджена наказом ректора університету від 25.08.2021 р. № 102

2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2021 р.

3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики

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

1 Дослідження предметної області та постановка задачі
2 Моделі та методи для вирішення задачі
3 Алгоритми та технології вирішення задачі
4 Реалізація та тестування програмної системи
5 Оцінювання ефективності запропонованого підходу

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

Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Нечипорук А. О	4.12.21	4.12.21
Нормоконтроль	Форкун Ю. В	3.12.21	3.12.21

7. Дата видачі завдання <u>01 вересня 2021 р.</u>

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1 Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; визначення структури дипломної роботи	01.09 – 07.09.2021	
2 Робота над розділом 1 дипломної роботи – вивчення літературних джерел; аналіз відомих моделей, методів та засобів за темою роботи; висновки до розділу та постановка задачі	08.09 – 25.09.2021	
3 Робота над розділом 2 дипломної роботи – розробка моделей та методів вирішення поставленої задачі; висновки до розділу	26.09 – 10.10.2021	
4 Робота над науковими статтями	11.10 – 20.10.2021	
5 Робота над розділом 3 дипломної роботи – розробка алгоритмів та технологій, проектування ПЗ для вирішення поставленої задачі; висновки до розділу	11.10 – 26.10.2021	
6 Робота над розділом 4 дипломної роботи – програмна реалізація спроектованих рішень, результати експериментів, їх аналіз; висновки до розділу	27.10 – 15.11.2021	
7 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків; оформлення пояснювальної записки та графічних матеріалів згідно вимог стандартів	16.11 – 30.11.2021	
8 Попередній захист дипломної роботит	Листопад (згідно графіка)	
9. Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12 – 04.12.2021	
10 Підготовка до захисту дипломної роботи	05.12 – 08.12.2021	

Студент ______________ І. В. Гурман
Підпис Ініціали, прізвище

Керівник проєкту (роботи) ______________ Л. П. Бедратюк
Підпис Ініціали, прізвищ

РЕФЕРАТ

Тема дипломної роботи: «Удосконалення методу виявлення помилок при тестуванні програмного забезпечення на основі трасування стеку».

Автор роботи: Гурман Іван Васильович.

Керівник роботи: Бедратюк Леонід Петрович.

Пояснювальна записка: 99 с., 11 рис., 13 табл., 3 дод., 85 джерел.

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, СТЕК, ТРАСУВАННЯ СТЕКУ, АВТОМАТИЧНЕ ВІДТВОРЕННЯ ЗБОЇВ.

Об’єктом дослідження є процеси автоматизованого пошуку помилок у програмному забезпеченні.

Мета дослідження – удосконалення методу автоматичного виявлення помилок у програмному забезпеченні.

У роботі використані наступні методи дослідження та апаратура:

- спостереження, експеримент, абстрагування, аналіз та синтез, формалізація;
- сучасні інструментальні засоби проектування та програмування;
- персональний комп’ютер.

Налагодження збоїв програми – це дорогий і трудомісткий процес, який покладається на досвід розробника та вимагає знань про систему. Протягом багатьох років дослідницька спільнота розробила кілька автоматизованих підходів для полегшення налагодження. Серед цих підходів – відтворення збою на основі пошуку, яке намагається створити тестовий приклад, здатний відтворити даний збій, щоб зробити його видимим для розробників, виключно на основі трасування стека, включеного у звіт про аварійне завершення роботи. Ми вважаємо, що це робить відтворення збоїв ідеальним кандидатом для досягнення наскрізної локалізації помилок. У цій дипломній роботі ми досліджуємо та емпірично оцінюємо використання відтворення збоїв на основі пошуку в поєднанні з локалізацією несправностей на основі спектру для 50 аварій у реальному світі. Починаючи зі звіту про аварійне завершення роботи, ми створюємо тестові тести, що відтворюють аварійне завершення, і використовуємо їх у поєднанні з наявним або автоматично згенерованим набором модульних тестів як вхідні дані для локалізації несправностей на основі спектру. Наші результати показують, що, хоча рукописні тестові випадки залишаються найефективнішими в загальному сценарії, автоматично створені тестові випадки, що відтворюють аварійне завершення роботи, все ж зменшують кількість операторів, які підлягають дослідженню розробниками. Крім того, при розгляді найкращого сценарію, коли оцінюються лише тестові випадки, що відтворюють аварії, що охоплюють несправність, ми не спостерігаємо статистично значущої різниці між точністю локалізації несправності при використанні рукописних або автоматично згенерованих тестових випадків. Наші результати підтверджують доцільність наскрізної автоматичної локалізації збоїв. Результати також виявляють нові проблеми як для автоматичного генерування тестових прикладів, так і для локалізації помилок, а також при їх поєднанні.

3.12.21р

ABSTRACT

Master’s thesis: «Improving the Stack Trace-Based Method for Detection of Errors in Software Testing».

Author: Hurman Ivan Vasulovuch.

Head of work: Bedratyuk Leonid Petrovuch.

Master’s thesis consists of: 99 p., 11 pc., 13 tb., 3 add., 85 srs.

TESTING SOFTWARE, STACK, STACK TRACKING, AUTOMATIC FAULT REPRODUCTION.

The object of research is the processes of automated software debugging.

The purpose of the study is to improve the method of automatic detection of errors in software. The following research methods and equipment were used in the study:

The following research methods and equipment are used in the work:

- observation, experiment, abstraction, analysis and synthesis, formalization;
- modern tools for design and programming;
- Personal Computer. The study examined the procedure for attracting cryptocurrency investments; identified problems in the industry and ways to solve them; formed the basic requirements for the system and the functions it must perform.

Debugging a program is an expensive and time-consuming process that relies on the experience of the developer and requires knowledge of the system. Over the years, the research community has developed several automated approaches to facilitate debugging. These approaches include search-based crash reproduction, which attempts to create a test case that can reproduce the crash to make it visible to developers, based solely on the trace of the stack included in the crash report. We believe that this makes fault reproduction an ideal candidate for achieving end-to-end error localization. In this thesis, we investigate and empirically evaluate the use of search-based fault reproduction combined with spectrum-based fault localization for 50 real-world accidents. Starting with the crash report, we create crash tests that reproduce them and use them in combination with an existing or automatically generated set of modular

створюємо тестові тести, що відтворюють аварійне завершення, і використовуємо їх у поєднанні з наявним або автоматично згенерованим набором модульних тестів як вхідні дані для локалізації несправностей на основі спектру. Наші результати показують, що, хоча рукописні тестові випадки залишаються найефективнішими в загальному сценарії, автоматично створені тестові випадки, що відтворюють аварійне завершення роботи, все ж зменшують кількість операторів, які підлягають дослідженню розробниками. Крім того, при розгляді найкращого сценарію, коли оцінюються лише тестові випадки, що відтворюють аварії, що охоплюють несправність, ми не спостерігаємо статистично значущої різниці між точністю локалізації несправності при використанні рукописних або автоматично згенерованих тестових випадків. Наші результати підтверджують доцільність наскрізної автоматичної локалізації збоїв. Результати також виявляють нові проблеми як для автоматичного генерування тестових прикладів, так і для локалізації помилок, а також при їх поєднанні.

3.12.21р

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ........ 10
ВСТУП........ 11
1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ........ 16
  1.1 Аналіз предметної області, останніх досліджень та джерел........ 16
  1.2 Підходи засновані на спектру........ 17
  1.3 Підходи на основі зрізів........ 20
  1.4 Підходи на основі станів програми........ 23
  1.5 Підходи на основі машинного навчання........ 24
  1.6 Оцінювання........ 26
  1.7 Автоматична генерація блок-тесту........ 27
  1.8 Автоматичне відтворення збоїв........ 29
2 МОДЕЛІ ТА МЕТОДИ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ........ 34
  2.1 Моделі автоматичної локалізації збоїв у програмному забезпеченні........ 34
  2.2 Оцінка методу визначення несправності........ 39
  2.3 Набори даних........ 43
  2.4 Тестові набори........ 43
  2.5 Аналіз даних........ 48
3 АЛГОРИТМИ ТА ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ........ 50
  3.1 Розробка компонентів........ 50
  3.2 Екстрактор тестових кейсів DEFECTS4J........ 52
  3.3 Тестовий генератор BOTSING........ 55
  3.4 Генератор блоку тестування EVOSUITE........ 57
  3.5 Постобробка тестових наборів........ 59
  3.6 Локалізатор несправності GZOLTAR........ 59
  3.7 Постобробка даних знаходження несправності........ 63
4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ........ 65
  4.1 Реалізація сценарію........ 65
  4.2 Кращий сценарій........ 70
  4.3 Фактори впливу........ 74
5 ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО ПІДХОДУ........ 81
  5.1 Коефіцієнти подібності........ 81
  5.2 Програмні зміщення спектрів........ 82
  5.3 Генерація тестових наборів........ 84
  5.4 Перевірка достовірності........ 85
ВИСНОВКИ........ 87
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ........ 90
ДОДАТОК А Скрипти точок фходження процедур процесу тестування........ 100

ДОДАТОК Б Копія наукової публікації 113
ДОДАТОК В Презентаційні матеріали 119

ПЕРЕЛІК СКОРОЧЕНЬ

СЗВО – Спектр звернення до виконуваних операторів

ГПК – граф потоку керування

ГЗП – граф залежності програми

РБФ – радіальна базисна функція

СЗ – сфера залежності

ВСТУП

З моменту винаходу програмного забезпечення в 19 столітті його використання постійно зростало. У наш час програмне забезпечення є фундаментальною частиною нашого повсякденного життя, оскільки воно не лише використовується, але й є вкрай необхідним для багатьох систем та установ у всьому світі.

Незважаючи на значні зусилля, витрачені на тестування та перевірку програмного забезпечення, програмні системи все ще виходять з ладу. У звіті австрійської фірми з тестування програмного забезпечення ”Tricentis” підраховано, що помилки програмного забезпечення в 2016 році коштували світовій економіці 1,1 трильйона доларів і вплинули на 4,4 мільярда людей у всьому світі [3]. Якщо подумати про це, то помилки програмного забезпечення вплинули на більш ніж населення світу [3], а фінансові втрати перевищують мільярди доларів.

Коли виявлено збій, наприклад, через неправильну поведінку або повідомлення про помилку, розробник повинен налагодити код і виправити програму. Налагодження несправності може бути трудомістким і складним завданням, оскільки часто доступно мало інформації (наприклад, трасування стека, дамп ядра або опис помилки кінцевим користувачем).

Часто, коли повідомляється про збій, першим кроком для розробника є написання тестового прикладу для налагодження. Такі тестові випадки відтворюють збій, щоб зробити його видимим для розробника та перевірити, чи він виправлений після виправлення вихідного коду. Коли збій можна спостерігати, другим кроком є визначення розташування основної несправності, яка також називається локалізацією несправності [2]. Локалізація несправностей була складним, виснажливим і надто дорогим ручним завданням, а враховуючи розміри сучасних програмних систем, робота стала ще більш виснажливою та тривалою. Витрачений час, а отже, і ефективність локалізації несправностей залежать насамперед від досвіду розробника та його знайомства з програмою.

Протягом багатьох років дослідники розробили різні автоматизовані підходи, щоб допомогти розробникам у їхній практиці налагодження, включаючи, але не обмежуючись, генерацію тестових прикладів на основі пошуку [1] та автоматичну локалізацію помилок.

При написанні програмного забезпечення на Java розробник, як правило, пише приклади модульних тестів, щоб перевірити передбачувану поведінку методів і виявити помилки, пов’язані зі змінами в програмному забезпеченні. Замість того, щоб нудно писати одиничні тестові приклади вручну, дослідники розробили підходи генерування тестових прикладів на основі пошуку для їх автоматичного створення.

Ці підходи часто покладаються на мета-евристичний оптимізуючий алгоритм [4] через складність проблеми пошуку (тобто простір пошуку може містити нескінченну кількість тестових випадків, які можна створити). Наприклад, підходи до генерації тестових прикладів на основі пошуку на основі білих скриньок [5] використовують еволюційні алгоритми. Зазвичай ці підходи мають на меті максимізувати охоплення створеного набору тестів, наприклад, охоплення рядків або гілок [6], або слабку оцінку мутації. Крім того, дослідження показали, що згенеровані набори тестів за допомогою генерації тестових прикладів на основі пошуку можуть полегшити процес налагодження [7] і можуть використовуватися для виявлення реальних помилок.

В останні роки дослідники розробили інші підходи, зосереджені на налагодженні збоїв програмного забезпечення. Серед них підходи відтворення збоїв на основі пошуку [8] спрямовані на створення одного або кількох тестових випадків, які можуть відтворити збій. Ці тестові приклади служать основою для процесу налагодження, усуваючи етап створення тестового випадку для налагодження. Оскільки ці тестові приклади генеруються автоматично, вони зменшують зусилля розробника на налагодження.

На відміну від інших підходів генерування тестових прикладів на основі пошуку, відтворення збоїв не має на меті максимізувати охоплення або оцінку мутації. Натомість він спрямований на відтворення послідовності дій, які призвели до збою. Емпіричне оцінювання з розробниками продемонструвало корисність тестових випадків, що відтворюють аварії, для полегшення процесу налагодження [9], що призвело до більшої кількості виправлень за менший час. Ми стверджуємо, що процес налагодження можна ще спростити, поєднавши відтворення збоїв на основі пошуку з автоматичною локалізацією помилок.

Основна мета автоматизованої локалізації помилок полягає в тому, щоб зменшити процес налагодження шляхом виявлення (потенційно) несправних операторів (або дефектних операторів) у програмі [2]. Протягом багатьох років було розроблено багато підходів, з яких найпопулярнішою є методика локалізації несправностей на основі спектру [2]. Локалізація несправностей на основі спектру покладається на інформацію про покриття (програмний спектр), створену з набору тестових випадків, що пройшли і не пройшли. На основі спектру програми підхід визначає потенційно підозрілі заяви, які викликають основну помилку.

Для правильної поведінки локалізації помилок на основі спектру, спектр програми повинен містити принаймні один невдалий тестовий випадок, інакше виявити несправність не можна, тому застосування локалізації несправностей на основі спектру було б безглуздим. Коли повідомляється про нову помилку, можна припустити, що немає доступного тестового випадку для виявлення основної несправності. Тому розробник спочатку повинен написати тестовий приклад для налагодження і, таким чином, повинен зрозуміти, чому його не можна розкрити, зменшуючи переваги автоматичної локалізації помилок. У цій дипломній роботі ми використовуємо відтворення збоїв на основі пошуку, щоб автоматизувати визначення тестового випадку для налагодження, відновлюючи всі переваги автоматичної локалізації помилок.

Щоб визначити підозрілість оператора, локалізація помилок на основі спектру базується на коефіцієнті подібності, щоб виміряти, наскільки близький шаблон виконання оператора до шаблону виконання збою (наближеного за невдалим тестовим прикладом(-ами)). Більша схожість означає вищий рівень підозрілості оператора.

Розроблено та оцінено декілька коефіцієнтів подібності в області автоматизованої локалізації несправностей [10]. Перевага локалізації помилок на основі спектру полягає в тому, що вона покладається лише на інформацію про покриття наявних тестових випадків, що пройшли та не пройшли, що спрощує інтеграцію в існуючі інфраструктури тестування та налагодження. Недоліком є обмежена діагностична точність, оскільки жоден оптимальний коефіцієнт подібності не може перевершити всі інші [10].

Останнім часом автоматизована локалізація несправностей також використовується як попередній крок багатьох підходів до автоматизованого відновлення програм. Автоматизована локалізація несправності використовується для ідентифікації (потенційно) несправних заяв для усунення [11]. У таких умовах розробник, як правило, повинен написати тестовий приклад, який робить помилку помітною перед застосуванням підходу до відновлення програми. Ми припускаємо, що цей крок також можна автоматизувати за допомогою відтворення збоїв, і пропонуємо автоматично визначати несправність, яка спричинила збій, відкриваючи новий шлях до повної автоматизації наскрізного відновлення програм [12].

Ця теза знаменує собою перший крок до повної автоматизації наскрізного відновлення програм [12] шляхом емпіричної оцінки потенціалу поєднання відтворення збоїв на основі пошуку з локалізацією несправностей на основі спектру, і ми називаємо це автоматичною локалізацією несправностей.

Для нашої оцінки ми використовуємо BOTSING [13], систему відтворення збоїв на основі пошуку, EVOSUITE [14], систему генерації модульних тестів на основі пошуку та GZOLTAR [14], систему локалізації помилок на основі спектру. Дотримуючись найкращих практик [15], ми використовуємо 50 реальних несправностей з DEFECTS4J [16], які проявляються як збої та раніше використовувалися для відтворення збоїв [17]. Зокрема, ми порівнюємо ефективність локалізації помилок аварії за допомогою рукописних і автоматично згенерованих тестових випадків; досліджуємо, чи забезпечують певні комбінації письмових і автоматично створених тестових випадків вищу точність при використанні з різними коефіцієнтами для локалізації несправностей; і вручну аналізуємо та порівнюємо написані від руки та автоматично згенеровані тестові випадки, щоб визначити фактори, що впливають на точність локалізації несправностей.

Решта структури цієї дипломної роботи виглядає наступним чином. У Розділі 1 описується основна та пов’язана робота над темою. Спираючись на найсучасніші дослідження, визначається та обґрунтовується підхід до автоматичної локалізації несправностей при збої та описується реалізація цього підходу в розділі 2.

Далі, у розділі 3 описується методологія оцінювання для оцінки ефективності розробленого підходу, описаного в попередньому розділі. Для проведення оцінювання створюються компоненти, які допомагають нам отримати необхідні набори даних, у розділі 4

У розділі 5 представляються результати оцінювання, які були проведені для виявлення помилок при тестуванні, щоб визначити його доцільність. Розділ 6 присвячений обговоренню результатів роботи, і розгляду актуальності загроз сьогодні. Також надається підсумковий висновок і представлено, на мій погляд, цікаві майбутні дослідження з цього питання.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області, останніх досліджень та джерел

Локалізація несправності – це завдання визначення розташування несправностей (тобто локалізація дефектного оператора) у програмних продуктах. Традиційними та ручними методами, які використовуються для локалізації помилок, є:

- журналювання програми
- твердження
- точки зупинки
- профілювання.

Використовуючи журналювання програми, розробник вставляє оператори журналу в код для моніторингу значень змінних та іншої інформації про стан. Розробник може перевірити інформацію, щоб діагностувати основну помилку, визначивши, де стан програми досягає несподіваного значення.

Оператори - це предикати, які повинні бути істинними під час правильного виконання програми. Розробники додають ці предикати в програму як умовні оператори, які припиняють виконання, якщо критерії не виконуються. Оператори можна використовувати для виявлення некоректної поведінки програми під час виконання. [18]

Для подальшої перевірки програми можна додати точки зупинки, щоб призупинити програму в певній точці для перевірки поточного стану. Розробник може поступово додавати точки зупину, зупиняти програму рядок за рядком або переходити до методу для подальшої локалізації помилки. Ранні дослідження використовували цей підхід, щоб допомогти розробникам локалізувати помилку, поки програма виконується під контролем символічного налагоджувача [19].

Профілювання – це форма аналізу часу виконання для визначення продуктивності програми, використовуючи такі показники, як час виконання, використання пам’яті та підрахунок викликів методів. Зазвичай профілювання використовується для оптимізації продуктивності програми. Однак його також можна використовувати для локалізації помилок, наприклад, шляхом виявлення несподіваних частот різних методів, виявлення витоків пам’яті чи падіння продуктивності коду [20], а також вивчення побічних ефектів лінивого оцінювання.

Описані вище методи, як правило, ручні та покладаються на досвід розробника. Тому протягом багатьох років дослідники розробили різні підходи до автоматизації завдання локалізації несправностей. Основна мета автоматизованої локалізації помилок полягає в тому, щоб зменшити зусилля налагодження шляхом виявлення (потенційно) несправних операторів у програмі [2]. Таке зменшення досягається шляхом ранжирування заяв у програмі на основі підозри, що оператор причетний до помилки.

Вонг та ін. [2] класифікували ці підходи за категоріями, включаючи, але не обмежуючись ними:

- підходи на основі спектру
- на основі фрагментів
- на основі стану
- на основі машинного навчання

1.2 Підходи засновані на спектру

Локалізація несправностей на основі спектру, що використовується в підході, запропонованому в цій роботі, покладається на інформацію про покриття (програмний спектр) набору тестових випадків, що пройшли і не пройшли. Спектр програми детально описує інформацію про виконання програми з певної точки зору. Широко використовуваним спектром програм є Спектр звернення до виконуваних операторів (СЗВО), що описує, які оператори були виконані під час виконання. Зазвичай, в рамках локалізації помилок на основі спектру, спектр програми базується на інформації про виконання набору тестів програми.

Залежно від підходу, спектри програми базуються на проходженні тестових випадків, невдалих тестових випадках або обох.

Коллофелло та ін. [21] припустили, що програмні спектри можна використовувати для локалізації помилок програмного забезпечення. Спектр програми можна використовувати, порівнюючи спектр невдалих тестових випадків зі спектром успішних тестових випадків та аналізуючи відмінності [22].

Метою локалізації помилок на основі спектру є ідентифікація оператора з шаблоном виконання, який максимально наближений до шаблону відмови у всіх тестових випадках [2]. Інтуїтивно, оператор є більш підозрілим, коли шаблон виконання оператора більш схожий на шаблон збою всіх невдалих тестових випадків. Наприклад, оператор, який виконується лише за допомогою невдалого тестового випадку, ймовірно, є частиною основної помилки. І навпаки, якщо шаблон виконання оператора подібний до шаблону виконання всіх успішних тестових випадків, менша ймовірність, що оператор міститиме помилку.

Рівень близькості кількісно визначається коефіцієнтом подібності, а ступінь можна інтерпретувати як рівень підозрілості оператора. Протягом багатьох років було запропоновано багато коефіцієнтів подібності [23]. У таблиці 1.1 наведено чотири найбільш ефективні та добре вивчені коефіцієнти разом з їх алгебраїчною формою. Pearson et al. [24] показали, що жоден з цих чотирьох коефіцієнтів подібності не забезпечує статистично значущої відмінності від інших, якщо використовується для локалізації несправностей на основі спектру.

Таблиця 1.1 – Коефіцієнти подібності

Коефіцієнт	Алгебраїчна формула
DSTAR [25]	$S = \frac{N_{CF}^{*}}{N_{UF} + N_{CS}}$
OCHIAI [23]	$S = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$
BARINEL [26]	$S = 1 - \frac{N_{CF}}{N_{CS} + N_{CF}}$
TARANTULA [27]	$S = \frac{N_{CF} / N_F}{N_{CF} / N_F + N_{CS} / N_S}$

Для даного твердження σ N_F – це загальна кількість тестів, що пройшли невдало, N_F – загальна кількість пройдених тестів, N_{CF} – кількість тестів, що пройшли невдало, що охоплюють σ, N_{UF} – кількість тестів, які пройшли невдало, що не охоплюють σ, N_{CS} – кількість проходження тестів, що охоплюють σ, а N_{US} – кількість проходження тестів, які не охоплюють σ.

Щоб краще зрозуміти концепцію локалізації несправностей на основі спектру, ми подивимося на приклад, як показано в таблиці 1.2. Фрагмент коду в другому стовпці містить помилку в рядку 6, яка має бути d = a / b (замість d = b / a). Припустимо, що існує два пройдених тестових приклади t1 і t2 з входами a = 1, b = 1 і a = 2, b = 1, відповідно, і один невдалий тестовий приклад t3 з входом a = 1, b = 2. Стовпці з третього до п‘ятого містять охоплення операторів для кожного з чотирьох тестових випадків. Останній стовпець показує рівень підозрілості на заяву, розрахований за допомогою коефіцієнта TARANTULA.

Результатом локалізації несправностей на основі спектру є відсортований список тверджень на основі рівня підозрілості в останньому стовпці. У цьому випадку рядок 5 і рядок 6 позначені як найбільш підозрілі, що було б правильно, оскільки помилка викликана несправністю в рядку 6.

Таблиця 1.2 – Приклад локалізації несправностей на основі спектру

№	Команди	Test t_1 $a=1, b=1$	Test t_2 $a=2, b=1$	Test t_3 $a=1, b=2$	N_{CF}	N_{CS}	S
1	`if (a > b)`	+	+	+	1	2	0.50
2	`c = a + b`		+		0	1	0.00
3	`d = a * b`		+		0	1	0.00
4	`else`				0	0	0.00
5	`c = a - b`	+		+	1	1	0.67
6	`d = b / a`	+		+	1	1	0.67
7	`return c + d`	+	+	+	1	2	0.50
		Вдало	Вдало	Збій			

де S обчислюється за допомогою коефіцієнта подібності TARANTULA (+ оператор виконується тестовим прикладом)

Щоб застосувати до програми локалізацію помилок на основі спектру, набір тестів для цієї програми повинен мати властивість $N_F > 0$ і $N_S > 0$. Інтуїтивно, якщо $N_F = 0$, у програмі немає помилок. А якщо $N_S = 0$, усі оператори, виконані невдалим тестом, будуть позначені як підозрілі, що не додасть додаткової інформації для розробника.

Аналогічно, ми можемо припустити, що для дефектного оператора $N_{CF} > 0$, оскільки дефектний оператор повинен виконуватися принаймні одним невдалим тестом. OCHIAI, BARINEL і TARANTULA засновані на тому факті, що $N_{CS} + N_{CF} > 0$, що має місце як $N_{CF} > 0$. Аналогічно, DSTAR припускає, що для дефектного оператора $N_{CS} + N_{UF} > 0$. Однак це припущення не завжди виконується, оскільки всі невдалі тестові випадки можуть охоплювати оператор ($N_{UF} = 0$), і в той же час оператор не покривається жодним прохідним тестовим прикладом ($N_{CS} = 0$) (наприклад, через відсутність покриття тестом). Інтуїтивно, коли $N_{CF} = 0$ і $N_{CS} + N_{UF} = 0$, тоді оператор має бути найбільш підозрілим з усіх. Тим не менш, у цьому випадку DSTAR призводить до невизначеного рівня підозрілості через поділ на нуль. В оригінальній роботі [25] не було визначено жодної додаткової процедури для вирішення цієї проблеми. У цій дипломній роботі будемо вважати, що $N_{CS} + N_{UF} > 0$, а коли $N_{CS} + N_{UF} = 0$, коефіцієнт DSTAR дорівнює нулю.

1.3 Підходи на основі зрізів

Розшарування програми використовується для локалізації помилки шляхом зменшення розміру програми шляхом дедалі більшого видалення невідповідних частин програми, таким чином, що результуючий фрагмент програми має таку ж поведінку, як і вихідна програма щодо помилки. Програмний фрагмент створюється шляхом зведення програми до критерію зрізу, вказуючи оператор і набір змінних, які цікавлять. Критерієм зрізу є пара (s,V), де s – оператор у програмі, а V – підмножина змінних програми [28]. Зріз програми, враховуючи критерій зрізу (s,V), складається з набору операторів і предикатів, які прямо чи опосередковано впливають на змінні в V перед виконанням s.

Вайзер [28] запропонував першу реалізацію зрізу програми в 1978 році. Основна мета зрізу програми – зменшити розмір програми, щоб розробникам, не знайомим з програмою, не доводилося проходити всю програму для виправлення помилки. Концепція заснована на припущенні, що збій пов’язаний з неправильним призначенням змінної, і помилка повинна бути знайдена в зрізі, пов’язаному з операторами, що впливають на значення змінної. У сфері нарізки програм існують три типи підходів: статичний зріз, динамічний зріз та зріз виконання [2].

Статичний зріз, запропонований Вайзером [28], обчислюється без урахування вхідних даних програми [20]. Статичний зріз містить мінімальний набір виконуваних операторів, які впливають на змінні у V аж до виконання s.

Проблема статичного зрізу полягає в тому, що обчислення мінімальної підмножини висловлювань є невирішуваним [28]. У оригінальному підході фрагмент програми апроксимується за допомогою графу потоку керування (ГПК) програми. З тих пір дослідники намагаються покращити наближення, використовуючи різні методи. Оттенштейн та ін. [30] представив статичне зрізування з використанням графу залежності програми (ГЗП), використовуючи доступність графу. Останні підходи до статичного зрізу були застосовані для зменшення розміру двійкових виконуваних файлів [31] та для оптимізації перевірок типів [32].

Недоліком статичного зрізу є те, що фрагмент містить усі виконувані інструкції, які можуть вплинути на результат, в результаті чого фрагмент містить оператори, які можуть не мати відношення до помилки. Замість того, щоб міркувати, які оператори впливають на змінні, що цікавлять, динамічне зрізування використовує поведінку виконання програми (як правило, з використанням невдалого виконання). Таким чином, розробник не тільки знає, що могло статися, але й може бачити, що сталося. Динамічний зріз містить усі оператори, які впливають на результат програми щодо певного введення.

Корель та ін. [33] запропонував поняття динамічного зрізу, використовуючи динамічний аналіз для визначення операторів, які впливають на змінні, які цікавлять при виконанні конкретної програми. Оскільки враховується лише одне виконання, динамічний аналіз може значно зменшити розмір зрізу, полегшуючи локалізацію несправностей. Протягом багатьох років були запропоновані додаткові підходи, включаючи динамічне розрізання за кількома точками [34], орієнтоване на сценарії динамічне зрізування [35] та відповідне зрізування.

Оскільки збирання динамічних фрагментів може зайняти багато часу та місця, можна використовувати зріз виконання. Зріз виконання заснований на потоці даних тестових випадків, які можна використовувати для визначення місцезнаходження програмних помилок [36]. Виконавчий фрагмент для даного тестового випадку містить набір операторів, які виконуються цим тестовим прикладом. Іншими словами, він витягує всі окремі оператори в шляху виконання даного тестового прикладу.

Були розроблені різні інструменти для налагодження зрізів виконання, наприклад, eXVantage [37] та xSuds [10]. Агравал та ін. [36] використовував зріз виконання, порівнюючи фрагмент виконання одного невдалого та одного прохідного тестового випадку. Цей підхід було розширено за рахунок використання багатьох вдалих та невдалих тестових випадків [39].

Щоб порівняти різні підходи на основі зрізів, у таблиці 1.3 зображено приклад програми, яка містить помилку в рядку 7 (припускаючи, що рядок має бути $y = y * 2 * i$). Критерій зрізу визначається як (7, y). Статичний зріз містить усі оператори, які можуть вплинути на результат змінної y. Динамічний зріз містить оператори, що впливають на змінну y для даного тестового прикладу (у прикладі $a = 3$). Нарешті, зріз виконання містить оператори, які виконуються даним тестовим прикладом, тому в цьому випадку оператор виконується для $a = 3$.

Таблиця 1.3 – Різниця між статичним, динамічним і нарізанням виконання для програми з помилкою.

№	Програма	Статичний зріз для *y*	Динамічний зріз для *y* при *a=3*	Зріз виконання для *y* при *a=3*
1	`input(a)`	+	+	+
2	`int i = 2`	+	+	+
3	`int x = 0`			+
4	`int y = 1`	+		+
5	`if (i < a)`	+	+	+
6	`x = x + i`			+
7	`y = y * i //помилка`	+	+	+
8	`else`	+		
9	`x = x - i`			
10	`y = y / i`	+		
11	`return y`	+	+	
				+

1.4 Підходи на основі станів програми

Стан програми містить змінні та значення в певний момент під час виконання, які можна використовувати для локалізації помилок. Одним із способів є відносне налагодження [40], де поточний стан порівнюється з опорним станом, який, як відомо, є правильним. Інший спосіб - порівняти стан програми тестового прикладу, що пройшов, із станом програми невдалого тестового випадку.

Відносне налагодження починається з розробника, який формулює набір тверджень про критичні структури даних у стані посилання та стані виконання. Ці твердження визначають розташування, в яких стан обох програм має бути однаковим. Відносний налагоджувач несе відповідальність за керування виконанням двох програм, перевірку тверджень та звітування про будь-які відмінності [40]. Якщо повідомляється про різницю, розробник ізолює несправний код шляхом рекурсивного уточнення тверджень. Як тільки розмір ізольованого коду досить малий, можна використовувати методи ручного налагодження для визначення основної несправності. Дослідження показали, що відносне налагодження є корисною технікою, яка дозволяє розробникам знаходити помилки в програмах, які були модифіковані або перенесені на інші мови [41].

Зеллер та ін. [42] запропонував техніку, яка називається дельта-налагодженням, що порівнює стани проходження та невдачі тестів. Замість ранжування оператора дельта-налагодження визначає підозрілість змінних у програмі. Підозрілість змінної визначається шляхом заміни значення тестового прикладу, що пройшов, на значення невдалого тестового прикладу та перевірки, чи змінює це результат тесту. якщо не зустрінеться така сама помилка, змінна більше не вважається підозрілою.

1.5 Підходи на основі машинного навчання

У контексті локалізації помилок машинне навчання використовується для того, щоб дізнатися чи визначити місцезнаходження помилки на основі доступних даних (тобто вихідного коду, покриття коду та результатів виконання тестових випадків). Підходи до локалізації помилок на основі машинного навчання [43] мають однакові ключові етапи обробки з можливістю додаткової попередньої та постобробки. Тому підходи до локалізації помилок на основі машинного навчання можна узагальнити таким чином:

– Збір навчальних даних. Першим кроком є збір необхідних навчальних даних. Зазвичай навчальний набір даних складається зі спектру програми для кожного тестового випадку та відповідного результату (який схожий на локалізацію помилок на основі спектру). Залежно від підходу навчальний набір даних збагачується додатковою інформацією, як-от відношення між тестовими прикладами.

– Навчіть модель машинного навчання. Використовуючи навчальний набір даних, модель машинного навчання навчається та оптимізується, щоб модель могла передбачити результат тестового випадку. Наприклад, на основі операторів, виконаних тестовим прикладом.

– Оцінюйте підозрілі оператори. На основі навченої моделі оператори в програмі ранжуються з урахуванням підозрілості. Підозрілість розраховується на основі результатів навченої моделі.

Вонг і Ці [44] запропонували методику локалізації помилок із нейронної мережі зворотного поширення помилки. Нейронні мережі з зворотним поширенням мають просту структуру, їх легко реалізувати, і було доведено, що вони придатні для апроксимації складних нелінійних функцій [44]. Вхід нейронної мережі – це спектр звернення до виконуваного оператора, який використовується в локалізації помилок на основі спектру, а вихідним є те, чи повинен виконаний оператор викликати збій чи ні. Іншими словами, нейромережа намагається оцінити результат виконання (прохідний чи невдалий) на основі операторів, які виконуються. Після навчання моделі підозрілий рівень для кожного оператора обчислюється за допомогою віртуальних тестових випадків (тобто тестових випадків, які виконують лише один оператор).

Коли програма стає більшою, кількість операторів, які беруть участь у процесі локалізації несправностей, збільшується. Це призводить до великого збільшення нейромережі, такий підхід виявився менш ефективний, а також дорогий у обчисленні [44]. Щоб вирішити цю проблему, Вонг і Ці [44] додали зріз виконання, щоб зменшити розмір програми.

Спочатку запропонований метод був створений для мови програмування С. Однак цей підхід не залежить від парадигми розробки, тому зворотне поширення також працює на об’єктно-орієнтованих мовах програмування, таких як Java [43]. Крім того, Ascari et al. [43] запропонував підхід, який використовує опорні векторні машини з тією ж метою та методологією.

У 2011 році Wong et al. [45] запропонував інший підхід, заснований на мережах радіальної базисної функції (РБФ), який використовує той самий вхід, що й вихід, що й нейронна мережа зворотного поширення. Перевага РБФ полягає в тому, що він має більш високу швидкість навчання і менш схильний до паралічів і локальних мінімумів.

1.6 Оцінювання

Як правило, підхід F до локалізації помилок приймає як вхідні дані програму P і набір тестів T з принаймні одним невдалим тестом. Він створює відсортований список операторів з $s_1 \geq \ldots \geq s_N$ де s_i означає підозрілий рівень оператора i, а N - кількість виконуваних операторів у програмі P. Ефективність підходу F заснована на відсоток програми P, який необхідно перевірити розробнику до виявлення дефектного оператора d [2].

Дослідники розробили кілька показників оцінки для визначення діагностичної точності, такими показниками є показник EXPENSE [27], оцінка EXAM [46], T-SCORE [47] та LIL [48].

Оцінка EXPENSE, показана в рівнянні 1.1, обчислює відсоток програми, який не потрібно перевіряти, коли оператори ранжуються відповідно до рівня підозрілості. У рівнянні 1.1 n — це ранг дефектного оператора d у звіті про локалізацію несправності, а N — загальна кількість заяв у програмі P.

$$Expense = \frac{N - n}{N} \tag{1.1}$$

Вонг та ін. [46] запропонував найбільш часто використовувану та прийняту метрику оцінювання, оцінку EXAM, яка також використовується в цій дипломній роботі. Оцінка EXAM, показана в рівнянні 1.2, подібна до оцінки EXPENSE, різниця полягає в тому, що вона обчислює відсоток коду, який необхідно перевірити розробнику. У рівнянні 1.2 n – це ранг дефектного оператора d у звіті про локалізацію несправності, а N – загальна кількість команд у програмі P.

$$Expense = \frac{n}{N} \tag{1.2}$$

T-SCORE, рівняння 1.3, призначений для нестатистичних методів локалізації помилок, які створюють невеликий набір підозрілих заяв у програмі. Він оцінює відсоток коду, який розробнику не потрібно досліджувати, перш ніж виявити дефектну заяву. Метрика використовує графік залежностей програми (ГЗП) для обчислення набору вершин у графі, які необхідно перевірити, щоб знайти дефектний оператор. Найменший набір вершин, що включає дефектний оператор, називається сферою залежності (СЗ).

$$T - Score = 1 - \frac{СЗ}{ГЗП} \qquad (1.3)$$

Втрата інформації про розташування (LIL) [48] є альтернативною метрикою оцінки, яка використовує міру розбіжності розподілу між розподілом рівня підозрілості та ідеальним очікуваним розподілом. Перевага метрики LIL полягає в тому, що вона не залежить від ранжованого списку команд і може застосовуватися до нестатистичних моделей, що робить її придатною для визначення ефективності методики локалізації помилок для автоматизованого відновлення програм [48].

1.7 Автоматична генерація блок-тесту

модульні тестові випадки – це тип тестових випадків, які використовуються при тестуванні програмного забезпечення. Основна мета модульних тестових випадків полягає в тому, щоб переконатися, що компонент (тобто розділ програми) відповідає його дизайну, поводиться відповідно до наміру та виявити помилки, викликані майбутніми змінами програми.

Крім того, згенеровані модульні тести можна використовувати для налагодження. Коли виявлено основну помилку збою, приклади модульного тестування дозволяють розробнику усунути компоненти, перевіривши, чи компонент функціонує належним чином, що стосується аварії. Чим вище охоплення коду, тим швидше розробник може видалити частини коду.

Дослідники запровадили кілька підходів до автоматизованого генерування тестів, що дозволяє автоматично генерувати одиничні тестові випадки за попередньо визначеними критеріями [49]. Наприклад, підходи до генерації тестових прикладів на основі пошуку в білому ящику [50] покладаються на еволюційні алгоритми для створення одиничних тестових випадків. EVOSUITE [50] – це підхід генерування тестових прикладів на основі пошуку для генерування одиничних тестів. Вхідними даними для EVOSUITE є один або кілька цільових класів у програмі, а вихідним є набір тестів, що максимізує заданий критерій тестування (наприклад, покриття рядків, покриття гілок або слабке покриття мутацій).

Попередні дослідження підтвердили, що EVOSUITE може генерувати тестові випадки з високим охопленням коду [51], потужністю виявлення реальних помилок [52], знижувати витрати на налагодження [53] та доповнювати написані від руки тестові випадки [54]. Крім того, EVOSUITE використовується в контексті автоматизованої локалізації несправностей [55]. Кампос та ін. [55] запропонував алгоритм на основі пошуку для зменшення ентропії діагностичного рейтингу. Вони емпірично встановили, що цей підхід зменшив кількість заяв, позначених як потенційно підозрілі (тому розробнику доводиться вивчати менше коду).

Перес та ін. [56] показали, що діагностична точність підходу автоматизованої локалізації несправностей залежить від якості тестового набору. Тому він представив метрику [57], яка називається DDU, для кількісної оцінки якості тестового набору на основі щільності, різноманітності та унікальності. Чим вище показник DDU, тим краща діагностична точність підходу автоматичної локалізації несправностей.

1.8 Автоматичне відтворення збоїв

Як згадувалося раніше, першим кроком для розробника зазвичай є написання тестового прикладу для налагодження, коли повідомляється про новий збій (або, принаймні, отримання вхідних даних, які відтворюють збій). Пошук цих вхідних даних може бути трудомістким і виснажливим завданням [58]. Дослідники запропонували різні автоматизовані методи для створення тестів, які відтворюють цільові збої. Загальна назва цих методів називається автоматизованим відтворенням аварій.

Для створення цих тестових випадків підходи до відтворення збоїв використовують дані часу виконання [59] або стеки [60,61], а також використовують запис – повтор або підхід після невдачі.

Для відтворення запису аварії використовується програмне або апаратне забезпечення та моніторинг для запису та спостереження даних про час виконання в момент збою. Розробник може відтворювати записані дані і, таким чином, використовувати для ідентифікації основної несправності [59]. Це робить ці підходи ефективними та простими у використанні, оскільки розробник може повернутися до стану програми до та після збою. Тому розробник може швидко спостерігати збій.

Протягом багатьох років було розроблено багато підходів до відтворення записів, включаючи RECRASH [62], BUGNET [63], JRAPTURE [64], SYMCRASH [59] та MOTIF [65]. Це підходить до всіх даних запису часу виконання, щоб дозволити розробнику відтворити стан програми. Порівняно з іншими підходами, SYMCRASH використовує підхід вибіркового запису, який відстежує лише важливі та складні методи, щоб зменшити накладні витрати приладів. Більше того, MOTIF використовує краудсорсингові дані для аналізу повторюваних моделей даних про збої, роблячи його більш ефективним [65], за умови, що збої з різних програм мають однакові характеристики.

Однак недоліками підходів із записом відтворення є зниження загальної продуктивності [66] через накладні витрати, спричинені приладами, та ризик конфіденційності через можливий збір конфіденційних даних [67].

Оскільки використання даних часу виконання викликає численні труднощі, підходи після збою більше підходять для загального використання. Підходи після збою копіюють аварію, використовуючи дані, доступні після збою. Найпоширенішими даними є трасування стека, оскільки вони легко збираються за допомогою систем відстеження помилок (наприклад, JIRA або trac) або можуть бути вилучені з файлів журналів. Однак більшість підходів після збою вимагають додаткових даних як вхідних даних, окрім трас стека (наприклад, дампи ядра, моделі програмного забезпечення, існуючий набір тестів або інваріанти класів) [67]. Прикладами підходів після відмови є STAR [67], JCHARMING [61] та MUCRASH [68]. STAR і JCHARMING використовують комбінацію трасування стека та перевірки моделі для створення тестового прикладу, що відтворює аварійне завершення, тоді як MUCRASH генерує тестовий приклад, змінюючи наявний тестовий приклад, доки не буде виявлено збій.

Підходи після збою на основі трасування стека — це підмножина підходів після збою, які покладаються виключно на стекові трасування як джерело інформації. Ця перевага робить ці підходи легко інтегрованими в існуючі програмні системи та не вимагає додаткових кроків для розробника (тобто додаткові дані не потрібні). Солтані та ін. [69] показали, що підходи до відтворення збоїв на основі пошуку, що спираються на єдиний цільовий керований генетичний алгоритм, перевершують усі інші підходи, засновані на стекові трасування, за коефіцієнтом відтворення збоїв (тобто можуть відтворювати більше збоїв). Крім того, Солтані та ін. [69] підтвердили, що згенеровані тестові випадки корисні для автоматичного налагодження та ручного налагодження.

У керованому генетичному алгоритмі популяція тестових випадків-кандидатів розвивається до тестового випадку, що відтворює аварійне завершення. Кожен тестовий приклад – це послідовність виконуваних операторів у програмі, що тестується. Початкова сукупність тестових випадків створюється шляхом генерації випадкових тестових випадків, які принаймні викликають один із методів у трасі стека. Еволюція – це ітераційний процес, популяція на кожній ітерації якого називається поколінням. У кожному поколінні придатність кожного тестового випадку оцінюється за допомогою функції придатності. Функція придатності кількісно визначає, наскільки даний кандидат близький до глобального оптимального. У разі відтворення аварії функція відповідності визначає подібність трасування стека, створеного тестовим прикладом, порівняно з вихідним стеком. Функцію придатності можна розширити за допомогою додаткової евристики для покращення якості тестових випадків. Кандидати, що підходять, стохастично відбираються та рекомбінуються для створення нових рішень-кандидатів (так звані вихідні) та формування нового покоління. Нове покоління потім використовується в наступній ітерації генетичного алгоритму. Зазвичай алгоритм припиняється, коли досягається задовільна пристосованість або після максимальної кількості ітерацій.

Більш детальна оцінка Солтані та ін. [70], продемонстрували здатність підходів відтворення аварій на основі пошуку відтворювати складні збої. Цей підхід визначає функцію придатності (звану Crash Distance), використовуючи три евристики для керування процесом пошуку:

– покриття рядка перевіряє, чи охоплює згенерований тестовий рядок коду, де поширюється виняток;

– покриття винятків вказує, чи згенерований тестовий приклад генерує той самий тип винятку, що й дана трасування стека;

– подібність трасування стека порівнює подібність вихідного стека з подібністю, отриманою в результаті тесту.

BOTSING [71] є добре перевіреною платформою з відкритим вихідним кодом для відтворення аварій на основі пошуку, яка реалізує функцію відповідності Crash Distance разом з іншими новими методами, що покращують її. Вхідні дані BOTSING – це трасування стека та один із кадрів у трасі стека (тобто один із рядків, що вказують клас і метод, де поширювався виняток) як цільовий кадр. Потім він генерує один або кілька одиничних тестів, які перевіряють клас у цільовому кадрі і (при виконанні) спричиняють збій, створюючи ідентичну трасу стека. Наприклад, надаючи BOTSING трасування стека з листингу 1.1 і цільовий кадр 3 (рядок 3), він генерує модульний тест для класу DocumentContentDisplayer, викликаючи NullPointerException, що поширюється через ті самі перші три кадри трасування стека.

Лістинг 1.1

```
0 java.lang.NullPointerException
1    at [...].XWikiDocument.getXDOM(...)
2    at [...].DocumentContentDisplayer.getContent (...)
3    at [...].DocumentContentDisplayer.display([...]:248)
4    at [...].DocumentContentDisplayer.display([...])
5    at [...].DefaultDocumentDlsplayer.display ([...])
```

Один із підходів відтворення аварії, реалізований в BOTSING, використовує дві допоміжні цілі. Поряд з відстанню аварії алгоритм використовує евристичні методи, різноманітність послідовності методів і довжину тесту, перетворюючи його в задачу пошуку кількох об’єктів. Евристика різноманіття послідовності методів дозволяє BOTSING відтворювати кілька тестових випадків, що відтворюють аварійне завершення, які різноманітні в послідовностях викликів методів. Крім того, довжина тесту гарантує, що BOTSING не генерує величезні тестові випадки.

Недавнє дослідження [71] показує, що алгоритми, що спираються на дві допоміжні цілі, збільшують здатність відтворювати збої, отже, збільшується коефіцієнт відтворення збоїв. Більше того, використовуючи цей багатоцільовий підхід, BOTSING може створити кілька тестових випадків, що відтворюють аварійне завершення, які відрізняються за послідовностями викликів методів, для одного даного збою.

Тестові випадки, згенеровані BOTSING, можна безпосередньо використовувати для автоматизованої локалізації помилок, оскільки згенерований тестовий приклад гарантує, що відбудеться збій, який включає цільовий кадр у свій стек. Однак вибір цільового кадру для відтворення аварії може вплинути на результат автоматичної локалізації несправностей.

Наприклад, припустимо, що дефектні оператори розташовані у кадрі 2 трасування стека в листингу 1.1 (тобто помилка в getDisplayName). При відтворенні цього збою з цільовим кадром, встановленим на 1, BOTSING створить модульний тест для класу XWikiDocument, який генерує NullPointerException зі стеком, що містить лише перший кадр (з рядка 0 до 1 у листингу 1.1). Хоча цей згенерований тестовий приклад зазнає невдачі (наприклад, якщо він не відповідає передумові методу getXDOM), він може не перекрити помилку. На практиці розробник не може заздалегідь знати, чи охоплює тестовий приклад, що відтворює збої, помилку чи ні, оскільки це означало б, що розробник уже знає, що це за несправність, і не потребує локалізації несправності в першу чергу. З цієї причини попередні дослідження рекомендували відтворювати вищі кадри для покриття основного дефекту [67, 69]

2 МОДЕЛІ ТА МЕТОДИ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ

2.1 Моделі автоматичної локалізації збоїв у програмному забезпеченні

Тестована програма, зображена в таблиці 3.1, показує фрагмент коду методу countMatches з охопленням різних тестових випадків $(t1,...,t4)$, написаних розробником. Метод підраховує кількість букви в реченні, ігноруючи регістр. Однак метод містить помилку: лічильник збільшується на 2, якщо символ у рядку є еквівалентом у верхньому регістрі символу пошуку (див. рядок 7 таблиці 2.1). Рядок 7 слід вважати += 1.

Таблиця 2.1 – Приклад програми з охопленням рукописних тестових випадків, тест-кейс, що відтворює аварійне завершення

№	оператори	t_1	t_2	t_3	t_4	t'
1	`countMatches(String sentence, char letter)`					
2	`int count = 0;`	+	+	+	+	+
3	`for (char idx : sentence.toArray ())`	+	+	+	+	+
4	`if (idx.isCapital ())`	+	+	+	+	
5	`idx = idx.toLowerCase();`			+	+	
6	`if (letter == idx)`			+	+	
7	`count += 2 //bug`				+	
8	`else`	+	+			
9	`if (letter == idx)`	+	+			
10	`count += 1;`	+				
1.	`return count;`	+	+	+	+	

З чотирьох написаних від руки тестових випадків $(t1,...,t4)$, *t4* є єдиними невдалими тестовими прикладами, оскільки він виконав дефектний оператор у рядку 7. Щоб локалізувати помилку, розробник міг застосувати на основі спектру локалізація несправності, оскільки є невдалий тестовий приклад. Застосовуючи локалізацію помилок на основі спектру, він дійсно ідентифікує дефектний оператор, оскільки рядок 7 є єдиним оператором, який виконується невдалим тестом. Використовуючи цю інформацію, розробник може швидко знайти дефектний оператор і виправити помилковий метод.

Коли програма виправлена, тестовий пакет зелений, настав час відправити код у виробництво. Після роботи користувач вводить комбінацію введення (`null, 'a'`), яка викликає `NullPointerException`, викликане в рядку 3. Записуючи винятки, розробник отримує уявлення про те, що в програмі є помилка через неочікуваний вхід. Неможливо застосувати локалізацію помилок на основі спектру для налагодження помилки, оскільки серед існуючих тестових випадків немає невдалого тестового випадку.

Щоб застосувати локалізацію помилок на основі спектру, розробник повинен написати тестовий приклад налагодження, який виявляє збій. Для цього розробник повинен зрозуміти основну причину збою і, отже, спочатку знайти помилку. Отже, локалізація несправностей на основі спектру втрачає свою основну мету, яка полягає в тому, щоб зменшити зусилля налагодження, коли застосовується до нещодавно повідомленого збою.

Щоб відновити основну мету, ми стверджуємо, що відтворення збоїв на основі пошуку (просто зване відтворення збоїв) може автоматизувати завдання написання тестового прикладу налагодження. Тестовий випадок налагодження буде невдалим через відтворення збою в поточній кодовій базі. у нашому прикладі таблиці 1.1 тестовий приклад t' для відтворення аварій було створено за допомогою відтворення аварій. При невдалому тестовому випадку з відтворенням аварії можна знову використовувати локалізацію несправностей на основі спектру. У цьому випадку застосування локалізації несправностей на основі спектру з додатковим тестом із відтворенням аварії визначить рядки 2 і 3 як потенційно несправні. Ми вважаємо це точним діагнозом, тому що патч буде перевіркою на нуль на початку методу.

На рисунку 2.1 зображено огляд запропонованого нами підходу автоматичної локалізації помилок при аварії. Цей підхід складається з трьох компонентів: відтворення аварійних ситуацій, генерування модульних тестів і локалізації помилок, додатковим компонентом яких є створення модульних тестів.

У лівій частині діаграми показано необхідне введення, тобто програма, тестовий пакет і звіт про аварійне завершення роботи. Основна причина цих введених даних полягає в тому, що їх легко отримати і не вимагають додаткових зусиль від розробника (тобто не потрібно визначати додаткову інформацію).

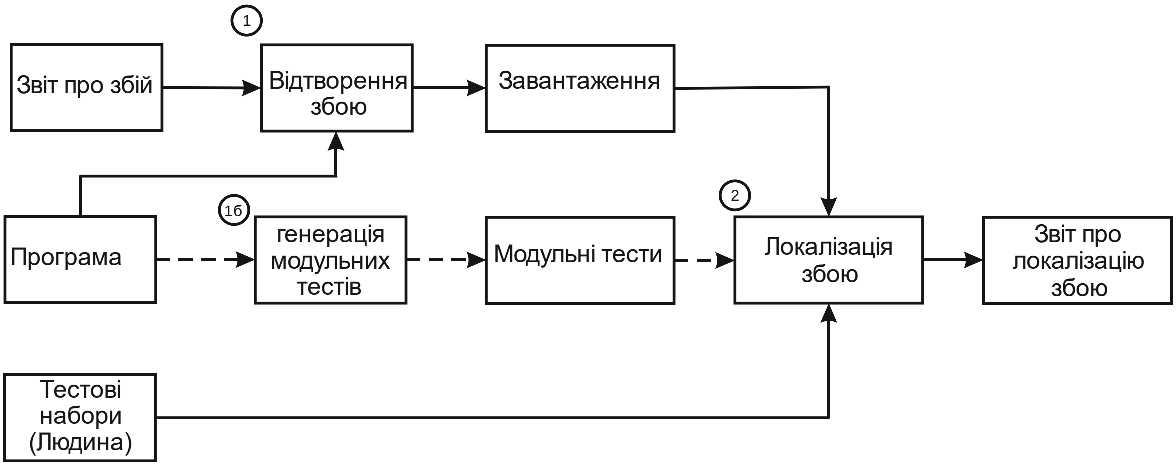


Рисунок 2.1 – Автоматична локалізація фатальних помилок

Програма є кодом програмного забезпечення, з якого виникло збій. Залежно від методів, що використовуються для відтворення збоїв, локалізації помилок та генерації модульних тестів, програма має бути вихідним кодом, скомпільованим кодом або обома. якщо для програми є набір тестів, то цей набір тестів є вхідним для автоматизованого процесу локалізації несправностей. Для нестатичних підходів набір тестів, швидше за все, слід надати у скомпільованій формі. Звіт про аварійне завершення роботи містить опис збою, залежно від підходу, формат звіту про аварійне завершення роботи може відрізнятися (наприклад, дамп ядра або трасування стека).

Результатом підходу є звіт про локалізацію несправності, зображений у правій частині діаграми. Як правило, звіт про локалізацію несправностей ранжує оператори в програмі, що тестується, на основі їх підозрілості щодо помилки збою.

На першому кроці ① звіт про аварійне завершення поєднується з програмою, що тестується, щоб створити один або кілька тестових випадків, що відтворюють аварійне завершення. Згенеровані тестові приклади повинні бути невдалими під час виконання, оскільки більшість підходів до локалізації помилок [2] залежать від невдалих тестових випадків. На другому кроці (1б) тестові випадки, що відтворюють збій, та існуючий набір тестів об’єднуються в процесі локалізації несправностей для написання звіту про локалізацію несправностей. Як альтернативу існуючому набору тестів можна використовувати згенеровані тестові випадки, додаючи крок ② до підходу. Це актуально, коли, наприклад, окремі частини програми недостатньо перевірені. У цьому випадку рекомендується створити набір тестів з достатнім рівнем покриття, як стверджує Перес та ін. [56] показали, що щільність тестового набору впливає на діагностичну точність локалізації несправності.

На першому кроці ① повинні бути згенеровані тестові випадки, що відтворюють збій. Дослідники розробили різні методики залежно від даних часу виконання [59,62, 72,] або трасування стека [60, 67, 68].

Щоб максимально зменшити зусилля з налагодження, ми вирішили використати підхід відтворення збоїв, який використовує виключно стекові трасування, оскільки сліди стека можна легко отримати і вони не спричиняють зниження продуктивності та не є ризиком для конфіденційності.

Багато підходів покладаються виключно на трасування стека. Солтані та ін. [69] показали, що підхід відтворення аварії на основі пошуку з використанням функції пристосованості Crash Distance перевершує всі інші методи на основі трасування стека з точки зору коефіцієнта відтворення аварій. Тестові випадки, створені за допомогою цього підходу, також корисні для налагодження вручну, дозволяючи розробнику ідентифікувати дефектний оператор із звітом про локалізацію помилок. Тому ми вирішили використовувати BOTSING [71], систему відтворення аварій на основі пошуку, яка реалізує функцію відповідності Crash Distance, для створення тестових випадків, що відтворюють аварійне завершення.

Для створення звіту про локалізацію несправностей на кроці ② необхідний підхід до автоматичної локалізації несправностей. Як показали Вонг та ін. [2] існує багато різних підходів, включаючи підходи на основі зрізів, спектру, стану та машинного навчання.

З цих чотирьох підходів ми вважаємо локалізацію несправностей на основі зрізів і стану непридатною. підходи, засновані на зрізах, оскільки більшість зрізів довгі та їх важко зрозуміти [73], що робить їх неефективними для локалізації помилок. Підходи, засновані на стані, оскільки він вимагає правильного опорного стану або вимагає повторного зворотного зв’язку від розробника.

Підходи на основі спектру та машинного навчання відповідають нашим вимогам до підходу автоматичної локалізації помилок. Однак ми вирішили використовувати локалізацію несправностей на основі спектру, оскільки в цій галузі було проведено більше досліджень, ніж у сфері локалізації несправностей на основі машинного навчання [2].

Підходи до локалізації несправностей на основі спектру покладаються на коефіцієнт подібності. Персон та ін. [24] не показали суттєвої різниці між найкраще вивченими коефіцієнтами. Крім того, ці коефіцієнти подібності не були перевірені за допомогою автоматизованих тестових випадків із відтворенням аварій. Таким чином, GZOLTAR [74] буде використовуватися як система локалізації несправностей на основі спектру, оскільки вона включає понад 20 коефіцієнтів подібності.

Генерація модульного тесту на кроці ②, необов'язково використовується, коли доступне мало тестування або його немає, виконує EVOSUITE [50], оскільки це найкращий доступний інструмент генерації модульних тестів із відкритим вихідним кодом [75].].

Крім того, попередні дослідження підтвердили, що EVOSUITE може генерувати тестові випадки з високим охопленням коду [51], виявленням реальних помилок [52] та зниженням витрат на налагодження [53]. Крім того, EVOSUITE використовується в контексті автоматизованої локалізації несправностей [55].

Підводячи підсумок, ми покладаємося на BOTSING [71], фреймворк відтворення аварій з відкритим кодом, для створення тестових випадків, що відтворюють аварійне завершення. Локалізація несправностей виконується за допомогою GZOLTAR [74], інструмента локалізації несправностей на основі спектру з відкритим вихідним кодом, який використовується в інших оцінках, що забезпечує понад 20 коефіцієнтів подібності [24]. Альтернативний крок для створення прикладів модульних тестів покладається на EVOSUITE [50], найкращий доступний інструмент для створення модульних тестів із відкритим вихідним кодом [75]

2.2 Оцінка методу визначення несправності

Для оцінки ефективності автоматичної локалізації збоїв у порівнянні з локалізацією помилок вручну використовуються для налагодження написані від руки тестові випадки. У цьому випадку ми припускаємо, що продуктивність ручної локалізації помилки аварії - це те, що потрібно покращити або принаймні відповідати. Тому формулюємо такі дослідницькі питання:

Питання 1: Як працює автоматична локалізація помилок аварії порівняно з локалізацією помилок вручну з точки зору точності, коли BOTSING генерує один або кілька тестових випадків з відтворенням аварій?

Ми прагнемо оцінити, наскільки автоматизована локалізація аварій можлива в загальному сценарії при використанні автоматичного відтворення аварії порівняно з рукописними тестовими прикладами. У загальному сценарії ми порівнюємо всі тестові випадки, що відтворюють аварії, створені BOTSING.

Однак, як згадувалося в Розділі 1, BOTSING вимагає цільового кадру, для якого створюється тестовий приклад. Тестовий випадок, що відтворює аварійне завершення роботи, буде моделювати поширення винятку, як у наведеній трасі стека до цього цільового рівня кадру. Немає гарантії, що це поширення не є очікуваною поведінкою (цільового) класу (наприклад, якщо (неявна) передумова не дотримується). З точки зору локалізації несправностей, це означає, що тестові приклади, що відтворюють аварії, (з невдачею) вносять шум у спектри програми. Ці тестові випадки не завершуються через основну помилку. В ідеалі в таких випадках BOTSING має бути спрямований на створення краш-відтворюючих тестів для більш високого кадру. На практиці, однак, заздалегідь невідомо, на які кадри слід орієнтуватися.

У нашому другому дослідницькому питанні ми вирішуємо це питання, розглядаючи найкращий сценарій, за яким BOTSING створює лише тестовий приклад, що відтворює аварійне завершення роботи, який запускає основну помилку.

Питання 2: Як автоматизована локалізація помилок виконується порівняно з локалізацією помилок вручну з точки зору точності, коли BOTSING генерує один або кілька тестових випадків з відтворенням аварії, що охоплюють помилку?

Нарешті, наше останнє дослідницьке питання має на меті розуміння можливостей і проблем автоматичної локалізації несправностей аварії та визначення сильних і слабких сторін автоматично створених тестових випадків, що відтворюють аварії.

Питання 3: Які фактори впливають на продуктивність та застосовність автоматичної локалізації несправностей при аварії?

2.3 Оціночні метрики

У літературі більшість автоматизованих методів локалізації несправностей оцінюються на основі штучних несправностей, створених шляхом мутації вихідного коду програми. Як правило, лише один оператор у коді змінюється, щоб створити помилку в коді. Таким чином, стандартна методика оцінки автоматизованого підходу *F* локалізації несправностей заснована на програмі *P* з одним дефектним оператором *d* [24]. Техніка локалізації помилок *T* виводить рейтинг операторів у *P*, упорядкований за рівнем підозрілості щодо збою. Потім діагностична точність обчислюється на основі рангу дефектного оператора d. Найпопулярнішим показником оцінювання є оцінка EXAM [46] (див. Рівняння 1.2).

Однак, щоб застосувати цю метрику до несправностей реального світу, ми повинні розширити стандартну методологію, запропоновану Пірсоном. [24]. Коли ми виконуємо нашу оцінку помилок у реальному світі, ми повинні враховувати зв’язки на рівні підозрілості, помилки з кількома операторами та помилки упущення.

Зв'язки в рівнях підозрілості. – стандартна методологія, передбачає, що кожен оператор в рейтингу має унікальний рівень підозрілості [24]. Однак у реальних проблемах існує ймовірність того, що два окремі кодові блоки або оператори мають однаковий рівень підозрілості. Ми припускаємо, що функція сортування, яка використовується для оператора порядку, довільно розриває ці зв’язки. Тому, коли кілька операторів мають однаковий рівень підозрілості, їм присвоюється однаковий ранг.

Помилка з кількома операторами – це помилка, для якої виправлення охоплює декілька операторів. Підраховано, що 76% [76] реальних помилок є помилками з кількома позиціями. У цьому випадку програма *P* містить кілька дефектних операторів *d*.

Ми припускаємо, що будь-який з дефектних операторів має бути локалізований, щоб зрозуміти основну причину збою. Таким чином, якщо програма *P* містить кілька дефектних заяв, то оцінка EXAM базується на дефектному твердження з найвищим рівнем підозрілості.

Помилка пропуску – це помилка, для якої виправлення полягає в додаванні нових операторів, а не в зміні чи видаленні існуючих. Отже, несправна програма P не містить дефектного оператора *d*, але деякі оператори відсутні. У цьому випадку непросто, яку заяву слід визначити, щоб локалізувати несправність.

Пірсон та ін. [24] визначив методологію та набір даних для вирішення цієї проблеми та визначення, які твердження слід звітувати. У разі пропуску ми не говоримо про дефектну заяву, а натомість називаємо їх заявами-кандидатами, оскільки ці оператори не обов’язково є дефектними.

Як загальне правило, оператори-кандидати для помилки пропуску є виконуваними оголошеннями до та після місця, куди мають бути вставлені нові оператори. Наприклад, у лістингу (таблиця 2.1) оператори-кандидати – це оператори, які можна виконати безпосередньо перед і безпосередньо після розташування виправлення.

Таблиця 2.1 – Оператори-кандидати у випадку if-else

0	**public void** exampleIfElse() {	
1	**if** (expression) {	
2	beforeStatement();	< candidate statement
3	} **else** {	
4	beforeStatement();	< candidate statement
5	}	
6	// patch	
7	afterStatement();	< candidate statement
9	}	

Однак у деяких випадках включаються додаткові оператори, наприклад, у лістингу (таблиця 2.2). У цьому випадку всі можливі твердження до та після локалізації виправлення включені в набір операторів-кандидатів, оскільки неможливо заздалегідь визначити, які з цих операторів вплинуть на результат програми.

Таблиця 2.2 – Оператори-кандидати у випадку блоку ініціалізації

0	**public void** exampleInitializerBlock() {	
1	map = **new** HashMap();	
2	map.put(k1, v1);	< candidate statement
3	// patch	< candidate statement
4	map.put(k3, v3);	< candidate statement
5	map.put(k4, v4);	< candidate statement
6	// patch	< candidate statement
7	**return** map;	< candidate statement
9	}	

2.3 Набори даних

Ми використовуємо набір даних DEFECTS4J [67] (версія 1.5.0), який складається з 435 реальних помилок із 6 проектів із відкритим кодом: JFREECHART, GOOGLE CLOSURE, APACHE COMMONS LANG, APACHE COMMONS MATH, MOCKITO та JODA-TIME . Для кожної помилки DEFECTS4J надає несправну та виправлену версію програми разом із мінімізованим виправленням, що представляє ізольоване виправлення помилки. Мінімізовані виправлення вказують, про які оператори в коді слід повідомляти за допомогою підходу до локалізації помилок. Окрім різних версій програми, доступний також рукописний набір тестів, включаючи тестовий приклад для ініціювання помилок (який не працює у несправній версії).

Для оцінки ми використовуємо 50 реальних несправностей з DEFECTS4J [76], які раніше використовувалися для відтворення аварій [70]. Крім того, ми використовуємо набір даних JCRASHPACK [70] і набір даних про локалізацію несправностей від Пірсона. [24].

Набір даних JCRASHPACK розширює набір даних DEFECTS4J, надаючи необхідні дані, необхідні для відтворення збоїв, включаючи трасування стека аварії та список відтворюваних кадрів. Набір даних локалізації помилок містить додаткову інформацію, необхідну для локалізації помилок, таку як список дефектних операторів і операторів-кандидатів, а також рядки коду для кожної з програм DEFECTS4J.

2.4 Тестові набори

Для кожного цільового кадру кожного аварії ми розглядаємо п'ять різних наборів тестів:

- рукописний набір тестів проходження (Man_{pass}+), наданий DEFECTS4J;

– рукописний набір тестів для невдач (Man_{fail}+), також наданий DEFECTS4J;

– набір тестів, що містить один тестовий приклад із відтворенням аварій (Botfaii1), згенерований BOTSING;

– набір тестів, що містить кілька тестових випадків, що відтворюють аварії (Bot_{fail}+), також створені BOTSING; і

– згенерований набір тестів (Evo_{pass}+) із блоковими тестами, згенерованими EVOSUITE. На рисунку 2.2 зображено огляд розглянутих наборів тестів для оцінки.

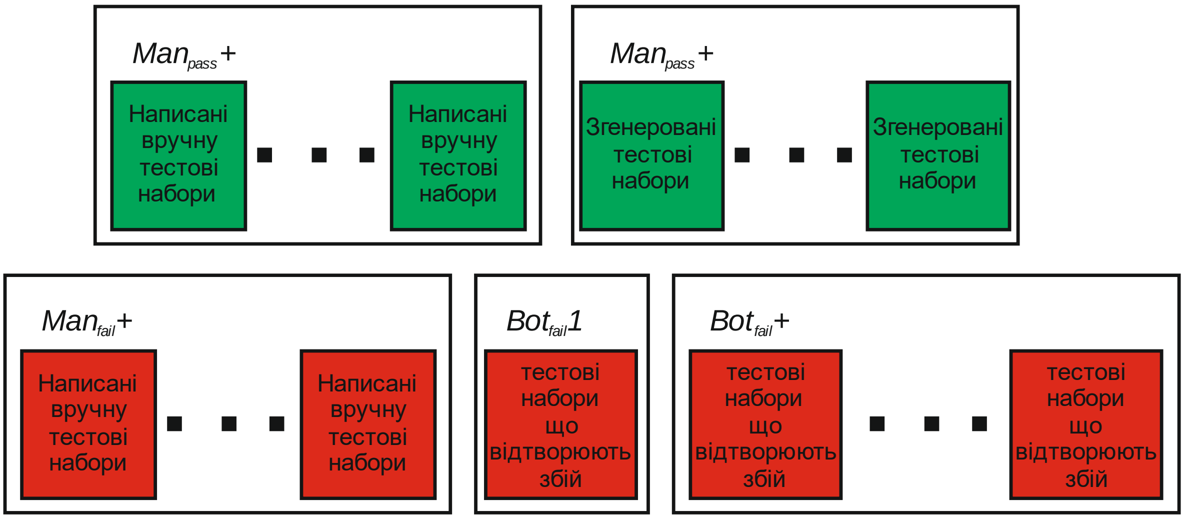


Рисунок 2.2 – Набори тестів, що використовуються для оцінювання.

Рукописні набори тестів витягуються з набору даних DEFECTS4J. Для оцінки ми розділили набір тестів для кожної програми на набір тестів, що містить відповідні тестові приклади проходження (Man_{pass}+), і набір тестів, що містить відповідні тестові приклади з невдачею (Man_{fail}+).

Щоб уникнути виконання всього набору тестів, ми фільтруємо наявний набір тестів і використовуємо лише відповідні тести. Тестовий випадок є релевантним для помилки, якщо він виконує принаймні один клас, перерахований у трасі стека збоїв.

DEFECTS4J також надає список відповідних тестових випадків, які є тестовими прикладами, які виконують принаймні один із модифікованих класів (тобто класів, які змінені в патчі). Однак ми стверджуємо, що наданий список не є вичерпним і не може використовуватися як є. По-перше, тому що він містить лише тестові випадки модифікованих класів і жодних одиничних тестів для всіх класів, перерахованих у трасі стека. А по-друге, тому що на практиці набір модифікованих класів заздалегідь невідомий.

Таким чином, ми визначаємо наступну методологію для отримання відповідних тестових випадків з усього набору тестів DEFECTS4J:

– Кожен тестовий приклад у наборі тестів виконується.

– Для кожного з тестових випадків JVM ClassLoader використовується, щоб визначити, які класи завантажуються і, отже, виконуються тестовим прикладом.

– Якщо один із завантажених класів знаходиться в трасі стека аварії, тестовий випадок буде позначено як відповідний тестовий випадок.

Після цього кожен тестовий випадок виконується на несправній версії програми DEFECTS4J, щоб отримати відповідні прохідні тести з відповідних тестових випадків.

Отримати відповідні невдалі тестові випадки набагато простіше, оскільки DEFECTS4J надає список тестових випадків, які викликають помилку. Ці тестові приклади мають такі властивості, що вони зазнають невдачі до виправлення та проходять після ремонту, а збій тестового випадку не є випадковим і не залежить від порядку виконання тесту.

Для створення тестових випадків, що відтворюють аварійне завершення роботи, ми виконали BOTSING з бюджетом часу в 3 хвилини, трьома цілями пошуку та використавши багатоцільовий еволюційний алгоритм SPEA2 [71]. Алгоритм використовує головну ціль Crash Distance і дві другорядні цілі: Довжина тесту та Різноманітність послідовності методів. Усі інші параметри залишаються за замовчуванням.

Для кожного кадру кожної трасування стека ми запустили два раунди BOTSING, створивши два тестових набори ($\text{Bot}_{\text{fail}}1$) і ($\text{Bot}_{\text{fail}}+$). У першому випадку BOTSING зупиняється після того, як знайде перший тестовий приклад, що відтворює аварійне завершення (тобто, набір тестів складається з одного тестового випадку, що відтворює аварію). В останньому BOTSING продовжує процес пошуку після першого тестового випадку, що відтворює аварійне завершення роботи, і зупиняється після вичерпання бюджету часу (тобто набір тестів складається з кількох тестових випадків, що відтворюють аварії). BOTSING заснований на алгоритмі мета-евристичної оптимізації, тому немає гарантії, що тестовий приклад, що відтворює аварійне завершення, буде створено під час першого запуску. Тому ми надаємо BOTSING максимум 15 спроб створити тестовий приклад, що відтворює аварійне завершення роботи.

Для створення одиничних тестів ми використовуємо EVOSUITE з параметрами за замовчуванням, які також використовувалися в попередньому дослідженні Шамширі [77]. Шамширі використовував EVOSUITE з фітнес-функцією Whole Test Suite [34] і бюджетом часу 1 хвилину.

Подібно до (Man_{pass}+), ми створюємо модульні тестові випадки лише для відповідних класів. Таким чином, для кожного збою ми націлюємось на всі окремі класи, що з’являються в трасі стека, що призводить до набору тестів для кожного збою.

Набори тестів, згенеровані BOTSING і EVOSUITE, мають специфічні каркаси, які гарантують, що кожен тестовий приклад у наборі тестів завжди виконується в одному і тому ж стані, щоб уникнути недетермінованих тестових випадків (що може призвести до нестабільних тестових випадків). На жаль, через проблеми з сумісністю з GZOLTAR нам довелося видалити риштування зі згенерованих тестових випадків. Щоб компенсувати можливість того, що тестовий приклад може стати недетермінованим, ми додаємо крок для визначення та видалення недетермінованого тестового прикладу шляхом порівняння результатів тестового прикладу до та після видалення риштування. Якщо вихідні дані тестового випадку відрізняються, тестовий приклад буде видалено.

Крім того, EVOSUITE може генерувати прохідні тестові випадки, які створюють очікуваний неоголошений виняток. У цьому випадку EVOSUITE оточує тестовий випадок блоком try-catch, щоб переконатися, що тестовий приклад завершився успішно. Однак існує ймовірність, що згенерований тестовий приклад відтворює збій [69]. Це може спричинити перешкоди в підході автоматичної локалізації помилок, оскільки шлях виконання, що ініціює збій, буде позначено як прохідний. З цієї причини ми вирішили видалити всі тестові випадки, що охоплюють неоголошений виняток.

Наприклад, тестовий приклад, згенерований EVOSUITE для LANG-16b, показаний у лістингу. У цьому прикладі вхід "-0x", згенерований EVOSUITE, є можливим введенням, яке ініціює основну помилку, що спричиняє збій.

```
@Test(timeout = 4000)
public void test052 () throws Throwable {
  // Undeclared exception !
  try (
    Numberütils.ere atelnteger ("-Ox")
    fail("Expecting exception: NumberFormatExcept ion")
  } catch (NumberFormatException e) (
    verifyException("org.apache.commons.lang.math.Numberütils", e) ;
  }
}
```

У тому ж випадку BOTSING може створити тестовий приклад, що відтворює аварійне завершення роботи, який генерує виняток. Якщо викликаний виняток не є неоголошеним винятком, його оточує блок try-catch, що робить його прохідним тестовим випадком. Наприклад, тестовий приклад, згенерований для LANG-16b з цільовим кадром 1, показаним у лістингу. У цьому випадку ми припускаємо, що згенеровані тестові випадки справді мають невдачу. Тому ми видаляємо лише блок try-catch, а не весь тестовий приклад.

```
@Test(timeout = 4000)
public void testO () throws Throwable (
  try (
    NumberUt ils . ere ate Number ( " hnQf uK+F\" w>3% jyxr " )
    fail("Expecting exception: NumberFormatExcept ion")
  } catch (NumberFormatException e) (
    verifyExcept ion("org.apache.commons.lang.math.NumberUtil
  }
}
```

У нашому оцінюванні ми об’єднуємо різні набори тестів у п’ять різних комбінацій. Кожна конфігурація зберігається з одного набору невдалих тестових випадків, які викривають збій, і одного набору тестових випадків, що пройшли. У таблиці 2.3 наведено огляд різних конфігурацій.

Таблиця 2.3 – Оціночні конфігурації

Configuration Конфігурація	Description Опис
Man_{faii}^{+}-Man_{pass}^{+}	Contains only hand-written test cases, for both the crash exposing test cases as well as the relevant passing test cases. Містить лише написані вручну тести для виявлення випадків аварійного завершення роботи та відповідних вдалих тестів.
$Bot_{fail}1$-Man_{pass}^{+}	Contains one failing generated crash-reproducing test case and the hand-written relevant passing test cases. Містить один невдалий згенерований тест виявлення аварійного завершення роботи та написані вручну відповідні вдалі тести.
$Bot_{fail}+^{-}Man_{pass}+$	Contains multiple failing generated crash-reproducing test cases and the hand-written relevant passing test cases. Містить кілька невдалих згенерованих тестів виявлення аварійного завершення та написані вручну відповідні вдалі тести.
$Bot_{fail}1\ ^{-}Evo_{pass}+$	Contains one failing crash-reproducing test case and the automatically generated passing unit test cases. Містить один невдалий тест виявлення аварійного завершення роботи та автоматично згенеровані вдалі модульні тестові випадка.
$Bot_{fail}+^{-}\ Evo_{pass}+$	Contains multiple failing crash-reproducing test cases and the automatically generated passing unit test cases. Містить кілька невдалих текстів виявлення аварійного завершення роботи та автоматично згенеровані вдалі модульні тестові випадка.

2.5 Аналіз даних

Оскільки BOTSING може одночасно націлюватися лише на один кадр і може відтворювати декілька кадрів деяких трас стека, це означає, що BOTSING може створити тестовий приклад із відтворенням аварій для 98 кадрів для 50 збоїв, які тестуються. Крім того, ми хочемо порівняти чотири коефіцієнти подібності з таблиці 1.1. В результаті ми отримуємо 98 x 5 x 4 = 1960 точок даних. Далі ми зосередимося на статистичному аналізі.

Для цього ми порівнюємо різні комбінації на основі результатів іспитів. Для цього ми використовуємо статистику Варги-Делані [79], щоб вивчити розмір ефекту між двома комбінаціями. Якщо значення Al2 менше 0,5 для пари комбінацій (A, B), то комбінація A зменшує оцінку EXAM, і навпаки, якщо значення вище 0,5. Крім того, щоб визначити значущість розмірів ефекту, ми використовуємо непараметричний тест суми рангу Вілкоксона з a = 0,05 для помилки типу I.

Порівнюючи діагностичну точність комбінацій, наведених у таблиці 2.1, проводимо попарне порівняння між різними комбінаціями, дотримуючись процедури рейтингу турнірів, запропонованої Пірсоном. [24]. Рейтинг турніру розраховується шляхом порівняння парних результатів EXAM, присудження 1 балу переможцю, якщо він має статистично значно кращий (*p-value* < 0,05 і *A12* < 0,5) результатів EXAM, і ранжування конфігурацій за кількістю очок.

Повторимо це порівняння, але обмежимося наборами тестів, які містять принаймні один невдалий тестовий приклад, який покриває несправність.

Вручну дослідимо результати автоматизованого підходу локалізації аварії, щоб визначити потенційні фактори, що впливають на точність діагностики. Для аналізу ми розділили різні комбінації на три набори на основі їхньої ефективності в автоматизованому процесі локалізації аварії.

3 АЛГОРИТМИ ТА ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ

3.1 Розробка компонентів

Для отримання визначених оцінок ми розробили інструменти, щоб автоматизувати процес і зробити його легко відтворюваним.

Спочатку ми створюємо набір даних з різних наборів тестів, які використовуються для оцінки, що складається з рукописних тестових випадків (Man_{pass}+, Man_{fail}+), тестових випадків, що відтворюють аварійне завершення (Bot_{fail}1, Bot_{fail}+) і автоматично згенерованих одиничних тестів (Evo_{pass}+).

Для тестових наборів Man_{pass}+ і Man_{fail}+ ми витягуємо тестові випадки з набору даних DEFECTS4J. По-друге, тестові набори Bot_{fail}1 і Bot_{fail}+, ми генеруємо тестові випадки за допомогою BOTSING із визначеною конфігурацією. Нарешті, набір тестів Evopass+, ми створюємо тестові випадки за допомогою EVOSUITE з конфігураціями, визначеними в Розділі 2.3.

Для кожного з цих завдань було розроблено окремий компонент, щоб код був зрозумілим, оскільки кожен компонент несе єдину відповідальність, і щоб спростити повторення або переробити частини оцінювання. Завдяки модульній установці дослідження можна легко розширити за допомогою інших наборів тестів або іншого підходу до локалізації несправностей, оскільки не кожну частину потрібно переробляти.

На малюнку 5.1 зображено огляд компонентів, які використовуються для отримання набору даних тестових наборів. Для цього завдання використовуються три компоненти: витяжник тестового прикладу DEFECTS4J (1), генератор тестового прикладу BOTSING (2) і генератор тестового прикладу EVOSUITE (3)

Екстрактор тестових прикладів DEFECTS4J використовується для отримання відповідних тестових випадків, що пройшли та пройшли, з проекту DEFECTS4J. Набір тестів і звіт про аварію використовуються для визначення відповідних тестових випадків. Генератор тестових прикладів BOTSING використовується для створення одиничних і множинних тестових випадків, що відтворюють аварійне завершення, які відтворюють даний звіт про аварійне завершення роботи. Генератор модульного тесту EVOSUITE використовується для створення модульного тесту, що охоплює відповідні класи зі звіту про аварійне завершення роботи.

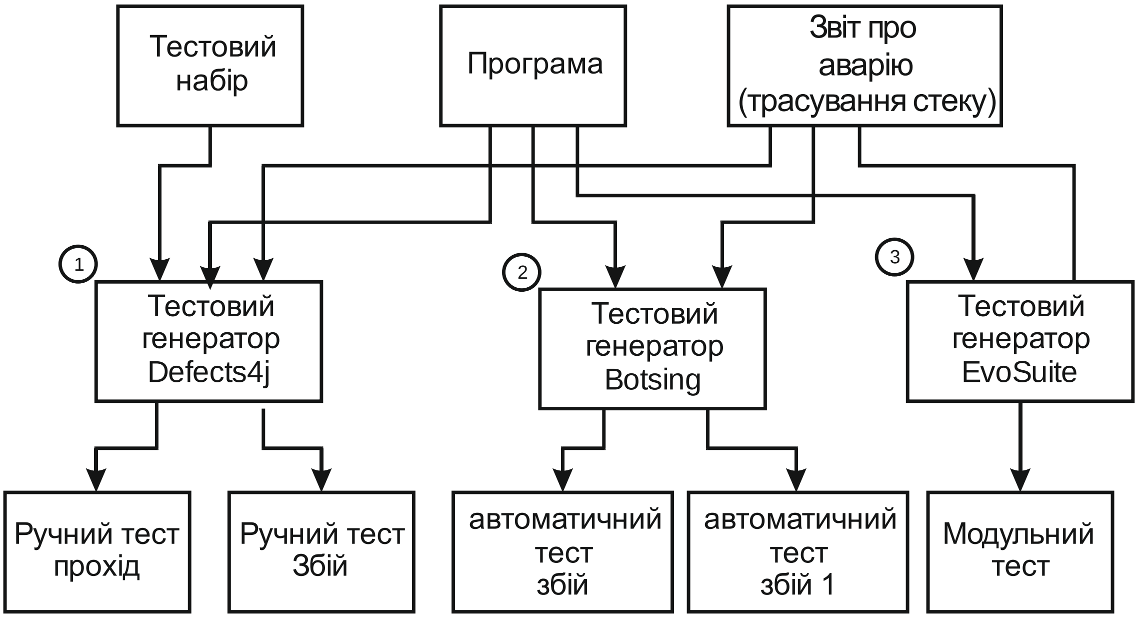


Рисунок 3.1 – Компоненти, що використовуються для отримання набору даних тестових наборів

Після завершення набору даних ми можемо застосувати локалізацію несправності до кожної з п’яти комбінацій (див. таблицю 2.1). Для виконання однієї з комбінацій ми використовуємо компоненти, як показано на малюнку 3.2. На першому кроці (1) ми запускаємо GZOLTAR, щоб застосувати локалізацію несправностей на основі спектру, використовуючи різні коефіцієнти подібності, щоб створити чотири звіти про локалізацію несправностей (тобто по одному для кожного коефіцієнта подібності). На другому кроці (2) ми запускаємо постобробку, щоб отримати всю необхідну інформацію для оцінювання зі звітів про локалізацію несправностей (наприклад, результати іспитів, покриття збоїв або спектри програми).

Виконання різних комбінацій організовано сценарієм, який автоматично виконує всі можливості для отримання 1960 точок даних, необхідних для аналізу даних.

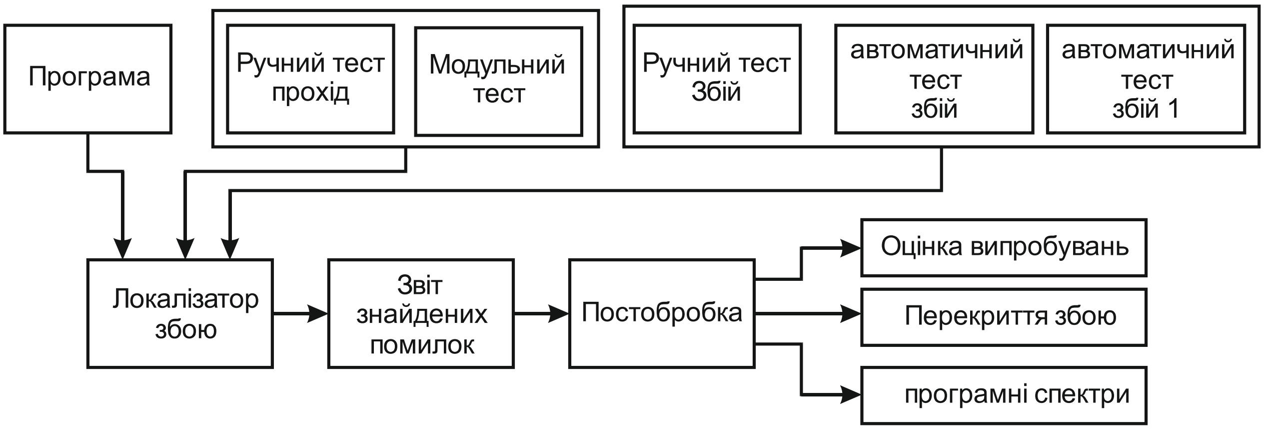


Рисунок 3.2 – Компоненти для отримання точок даних з набору даних

3.2 Екстрактор тестових кейсів DEFECTS4J

Першим кроком у зборі набору даних (див. рисунок 3.1) є вилучення відповідних рукописних тестів про проходження та невдачі з проекту DEFECTS4J, що тестується. Для цього ми створили контейнер DOCKER під назвою defects4j-extractor. Схема роботи defects4j-extractor показана на рисунку 5.3.

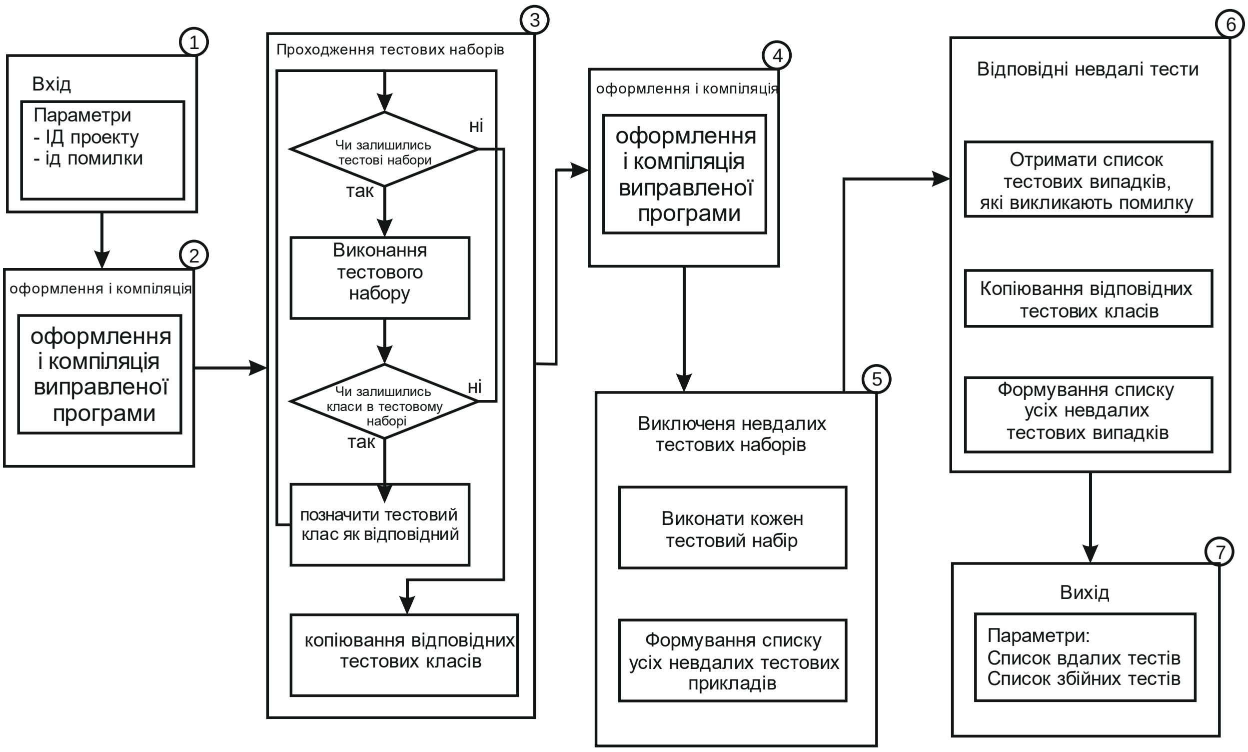


Рисунок 3.3 – Схема роботи defects4j-extractor

Вхідні дані 1 defects4j-extractor є project-id і bug_id проекту DEFECTS4J, а вихідні дані (7) є папкою, що містить усі відповідні тестові випадки, що пройшли та пройшли помилки.

На першому етапі перевірки та компіляції (2) фіксована версія проекту DEFECTS4J перевіряється та компілюється за допомогою інтерфейсу командного рядка DEFECTS4J (CLI). Фіксована версія проекту використовується, щоб гарантувати, що всі відповідні тестові класи можуть бути виконані без збоїв на наступному етапі. Крім того, він розкриває тестові випадки, які не вдалися у фіксованій версії, що, ймовірно, викликає іншу помилку або дефект у проекті.

На відповідному етапі проходження тестів @, відповідні тести проходження витягуються з існуючого набору тестів. Спочатку створюється список усіх тестових випадків за допомогою DEFECTS4J CLI. Щоб визначити відповідні тестові випадки, кожен тестовий приклад виконується за допомогою команди monitor.test, наданої DEFECTS4J CLI, яка повертає список класів, охоплених тестовим прикладом. Якщо один із охоплених класів знаходиться в трасі стека, що відповідає аварії, весь тестовий клас буде вказано як релевантний, оскільки зручніше скопіювати весь тестовий клас замість копіювання окремого тестового випадку.

Після того, як усі тестові приклади позначені, відповідні класи проходження тестів копіюються до каталогу test-pass/. Однак скопійовані тестові класи можуть містити невдалі тестові випадки. Тому на кроці (4) перевіряється та компілюється несправна версія проекту. На кроці (5) кожен тестовий клас виконується на несправній версії проекту, і всі невдалі тестові випадки перераховуються у файлі EXCLUDE_TEST_CASES. Файл EXCLUDE_TEST_CASES пізніше може бути використаний для виключення цих тестових випадків з автоматичної локалізації помилок, щоб гарантувати, що виконуються лише пройдені рукописні тести.

Нарешті, на кроці (6) витягуються відповідні невдалі тестові випадки. CLI DEFECTS4J може експортувати різні властивості, що залежать від версії, включаючи список тестових випадків, які викликають помилку. Тестові класи, що містять ці тестові приклади, копіюються в каталог test-fail/. Тестові випадки, які викликають помилку, перераховані в INCLUDE_TEST_CASES, тому лише ці тестові випадки можуть бути включені в автоматичну локалізацію помилок.

Для виконання defects4j-extractor є два необхідні вхідні дані: project-id для конкретного проекту та bug_id для конкретної помилки проекту DEFECTS4J. Приклад команди для запуску defects4j-extractor показаний у листингу, який витягує відповідні тести, що пройшли та не пройшли, з LANG-19b та зберігає їх у поточному каталозі.

```
docker run
—   e PROJECT_ID=Lang
—   e BUG_ID=9
-   v $(pwd)/tests-pass:/opt/runner/results/tests-pass
-   v $(pwd)/tests-fa il:/opt/runner/results/tests-fail defect4j-
extractor:latest
```

Після успішного виконання defects4j-extractor результати зберігаються в таких файлах і каталогах:

tests-pass/ містить тестові класи з принаймні одним відповідним тестовим прикладом.

tests-pass/EXCLUDE_TEST_CASES, що містить рядок для кожного невдалого тестового випадку у форматі className#testMethod. Тестові випадки в цьому файлі повинні бути виключені з локалізації помилок, щоб забезпечити виконання лише прохідних тестових випадків.

tests-faii/, що містить тестові класи з принаймні одним відповідним невдалим тестовим прикладом.

tests-faii/iNCLUDE_TEST_CASES, що містить рядок для кожної помилки, що запускає тестові випадки, у форматі className#testMethod. Тестові випадки в цьому файлі повинні бути включені в локалізацію помилок, щоб забезпечити виконання тестових випадків, що викликають помилку.

3.3 Тестовий генератор BOTSING

Метою цієї компоненти є автоматизація процесу відтворення збоїв для проектів у наборі даних DEFECTS4J. Компонент генерує один тестовий приклад із відтворенням аварійного завершення або кілька тестових випадків для даного проекту DEFECTS4J та цільового кадру. Для цього компонента ми створили контейнер DOCKER під назвою botsing-generator, блок-схема контейнера показана на малюнку 3.4.

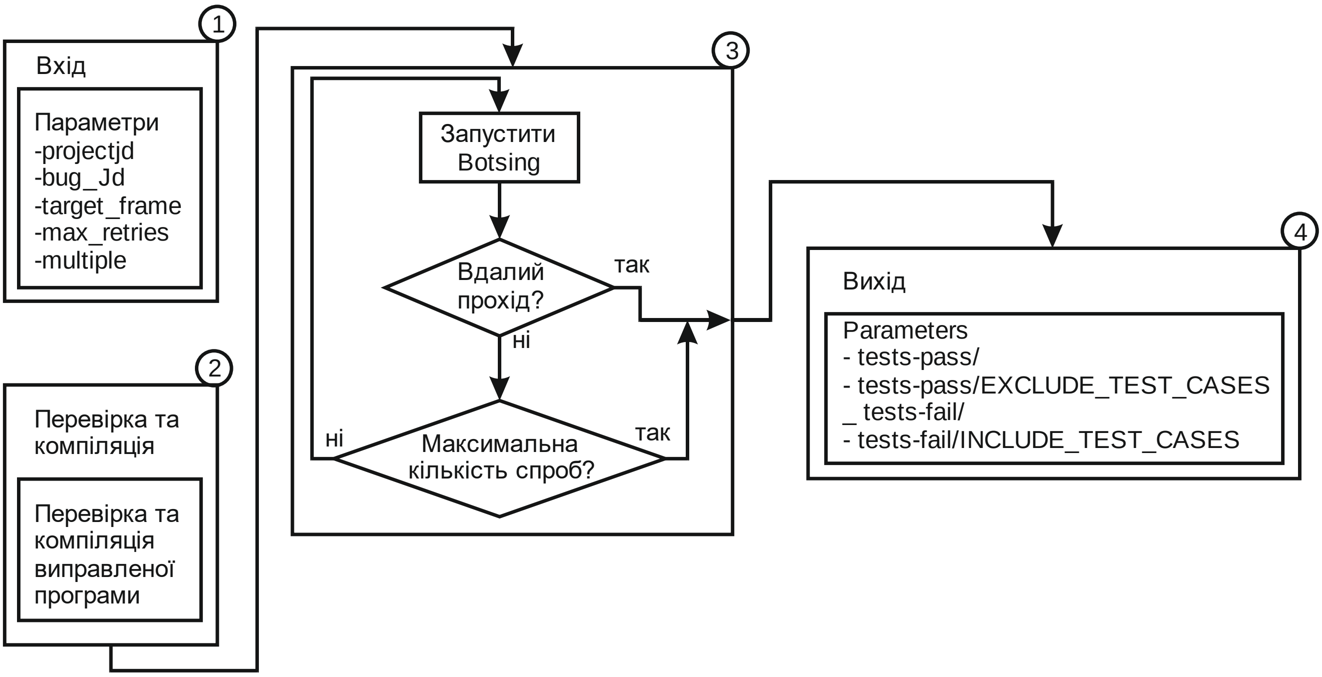


Рисунок 3.4 – Схема генератора тест-кейсів BOTSING.

Вхідні дані (1) *botsing-generator* є project_id та bug_id проекту DEFECTS4J. Разом із цільовим кадром, max_retries, що вказує максимальну кількість повторних спроб, якщо botsing не вдається згенерувати тестовий приклад з першого разу, і прапорець multiple, який вказує, чи слід генерувати кілька тестових випадків, що відтворюють аварійне завершення роботи. .

Вихід (4) – це папка, що містить набір тестів, що відтворюють аварійне завершення, та папку, що містить журнали останнього виконання BOTSING.

На етапі перевірки та компіляції (2) несправна версія проекту DEFECTS4J перевіряється та компілюється за допомогою DEFECTS4J CLI. Несправна версія використовується, щоб дозволити BOTSING генерувати тестові випадки, що відтворюють аварії на наступному кроці.

На етапі генерації тестового прикладу(3), що відтворює аварійне завершення, BOTSING генерує тестовий клас, що містить один або кілька тестових випадків залежно від знака множинного прапора.

Оскільки BOTSING базується на алгоритмі мета-евристичної оптимізації, що спирається на випадковість, цілком можливо, що за один запуск BOTSING не зможе створити рішення. Тим не менш, в іншому запуску BOTSING може знайти рішення. Тому, якщо рішення не знайдено (визначається значенням відповідності в журналах), BOTSING повторно запускається, доки не буде досягнуто максимальної кількості повторних спроб.

Крім того, BOTSING спирається на алгоритм мінімізації тестового випадку EVOSUITE. Іноді буває, що BOTSING генерує тестовий приклад, що відтворює аварійне завершення, але тестовий приклад зводиться до порожнього тестового випадку. У такому випадку BOTSING також повторюється, доки не буде досягнуто максимальної кількості повторних спроб.

Щоб запустити botsing-generator, вхідні дані, як описано вище, повинні бути надані через змінні середовища контейнера DOCKER. Приклад команди для запуску botsing-generator показаний у лістингу який генерує кілька тестових випадків для LANG-19b та цільового кадру 4, які відтворюють аварійне завершення. Результати зберігаються в поточному каталозі.

```
docker run
   —e PROJECT_ID=Lang
   —e BUG_ID=9
   —e TARGET_FRAME=4
   —e MAX_RETRIES=15
   —e MULTIPLE=true
-   v $(pwd)/tests-botsing:/opt/runner/results botsing-
generator:latest
```

Після успішного виконання botsing- generator результати зберігаються в таких каталогах:

– tests-botsing/, що містить тестовий клас, що відтворює аварійне завершення роботи з одним або кількома тестовими прикладами, залежно від режиму (один або кілька). Для кожного тестового класу є файл, що містить самі тестові класи (позначається className_ESTest), і файл, що містить каркаси EVOSUITE (позначається className_scaffolding_ESTest).

– tests-botsing/logs, що містять журнали останнього виконання BOTSING.

3.4 Генератор блоку тестування EVOSUITE

Метою цього компонента є автоматизація створення одиничних тестів для проектів у наборі даних DEFECTS4J. Компонент створює тестовий клас для кожного окремого класу в трасі стека, що відповідає аварії. Для цього компонента ми створили контейнер DOCKER під назвою evosuite-generator. Блок-схема контейнера показана на рисунку 3.5.

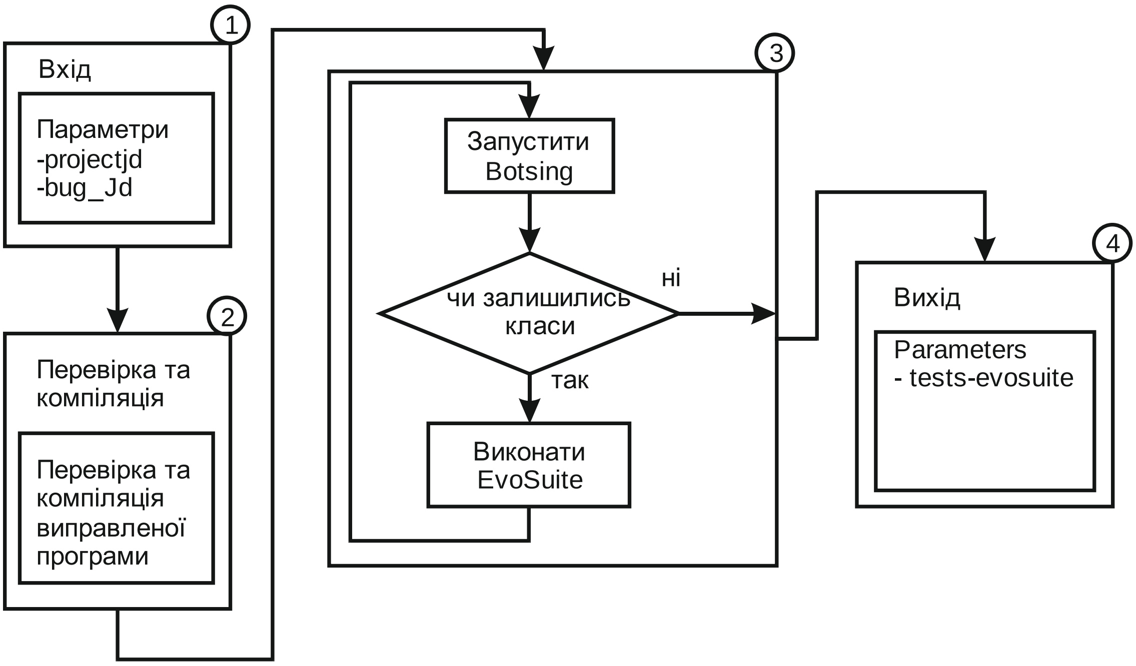


Рисунок 3.5 – Схема генератора блок-тестів EVOSUITE.

Вхідні дані (1) генератора evosuite – це ідентифікатор проекту та bug_id проекту DEFECTS4J, а також вихід (4) – це папка, що містить тестовий клас для кожного окремого класу у відповідній трасі стека.

На етапі перевірки та компіляції (2) несправна версія проекту DEFECTS4J перевіряється та компілюється за допомогою DEFECTS4J CLI. Використовується несправна версія, оскільки на практиці це буде єдина доступна версія.

На етапі генерування модульного тесту (3) кроку створюється список відповідних класів для помилки DEFECTS4J. Клас вважається релевантним, якщо клас знаходиться в одному з кадрів трасування стека і є частиною проекту DEFECTS4J. Після цього EVOSUITE виконується для створення класу модульного тестування для кожного з класів у списку. EVOSUITE виконується з використанням налаштувань за замовчуванням, як зазначено в документації.

Щоб запустити evosuite-generator, вхідні дані, як описано вище, повинні бути надані через змінні середовища контейнера DOCKER. Приклад команди для запуску evosuite-generator показаний у лістингу, який генерує класи модульного тестування для LANG-19b і зберігає результати в поточному каталозі.

```
docker run
  — e PROJECT_ID=Lang
  — e BUG_ID=9
  - v $ (pwd)/tests-evosuite:/opt/runner/results evosuite-
generator:latest
```

Після успішного виконання evosuite-generator результати зберігаються в таких каталогах:

tests-evosuite/, що містить класи модульних тестів, відповідні для даного проекту DEFECTS4J.

Для кожного тестового класу файл, що містить самі тестові класи (позначений як className_-ESTest), і файл, що містить риштування EVOSUITE (позначається className_- scaffolding_ESTest).

3.5 Постобробка тестових наборів

Як згадувалося в розділі 2.4, згенеровані тестові класи BOTSING і EVOSUITE несумісні з GZOLTAR, структурою локалізації помилок.

Тому ми повинні підготувати тестові класи до часткової оцінки локалізації несправностей. Це означає, що для тестових наборів BOTSING риштування та будь-які блоки спроби зловити будуть вилучені з тестових наборів. Крім того, назву тестових класів буде перейменовано таким чином, щоб вони містили слово BOTSING. Ця дія виконується тому, що назви тестових класів, згенерованих BOTSING і EVOSUITE, ідентичні для одного класу, що призводить до того, що один перекриває інший тестовий клас при вставці в GZOLTAR.

Для класів модульного тестування EVOSUITE стверджується, що каркаси та тестові випадки, які викликають неоголошений виняток, будуть видалені. Крім того, буде перейменовано назви тестових класів, щоб назва файлу включала слово EVOSUITE.

3.6 Локалізатор несправності GZOLTAR

Метою цієї компоненти є застосування локалізації несправностей до різних комбінацій тестових наборів, як описано в Розділі 2.4. Як згадувалося раніше, ми спираємося на структуру локалізації розломів під назвою GZOLTAR [74].

На Рисунку 3.6 зображено робочий процес фреймворку GZOLTAR. Робочий процес складається з чотирьох кроків, які необхідно виконувати один за одним.

Першим кроком (1) є перерахування всіх тестових випадків у проекті та визначення його типу (або JUnit, або TestNG). Після цього інструменти GZOLTAR додаються до проекту на кроці (2). Інструменти, що містять декілька API, дають змогу GZOLTAR відстежувати охоплення тестових випадків протягом усього проекту. Використовуючи список тестових випадків з кроку ф та інструменти з кроку (3), на кроці CD всі тестові випадки виконуються ізольовано, щоб спектр програми можна було побудувати та зберегти в серіалізованому об’єкті. На кроці (4) звіт про локалізацію несправності будується з використанням одного з реалізованих коефіцієнтів подібності. Оскільки спектр програми збережено, можна повторити крок ф, використовуючи різні коефіцієнти подібності, без повторного проходження кроків (1), (2) і (3).

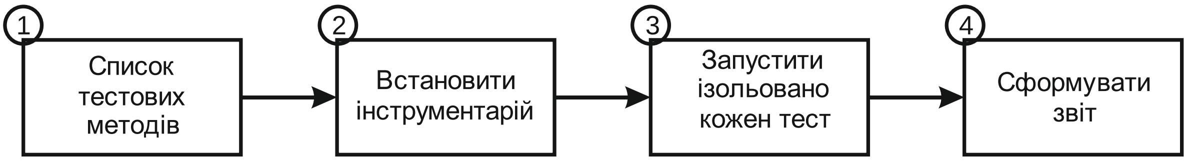


Рисунок 3.6 – Процес локалізації несправності за допомогою GZOLTAR

Після виконання GZOLTAR звіти про локалізацію несправностей зберігаються в таких файлах:

<formula>.ranking.csv, що містить рейтинг операторів у проекті, відсортований за рівнем підозрілості. <formula> – це назва формули, наприклад, при використанні коефіцієнта TARANTULA ім’я файлу – tarantula.ranking.csv.

matrix.txt, що містить матрицю спектру програми, де кожен рядок представляє тестовий приклад і його результат, а кожен стовпець представляє оператор (1 означає, що тестовий приклад охоплює рядок коду, 0 в іншому випадку).

tests.csv список усіх тестових випадків і їх результат, час виконання в наносекундах і трасування стека (у разі невдалого тестового випадку).

spectra.csv список усіх рядків коду, що виконується тестовим прикладом. Кожен рядок має такий формат:

className#methodName(methodParameters):lineNumber

Щоб автоматизувати процес оцінки, ми розширили початковий робочий процес і створили навколо нього контейнер DOCKER. Для цього компонента ми створили контейнер DOCKER під назвою gzoltar-runner, блок-схема контейнера показана на рисунку 5.7.

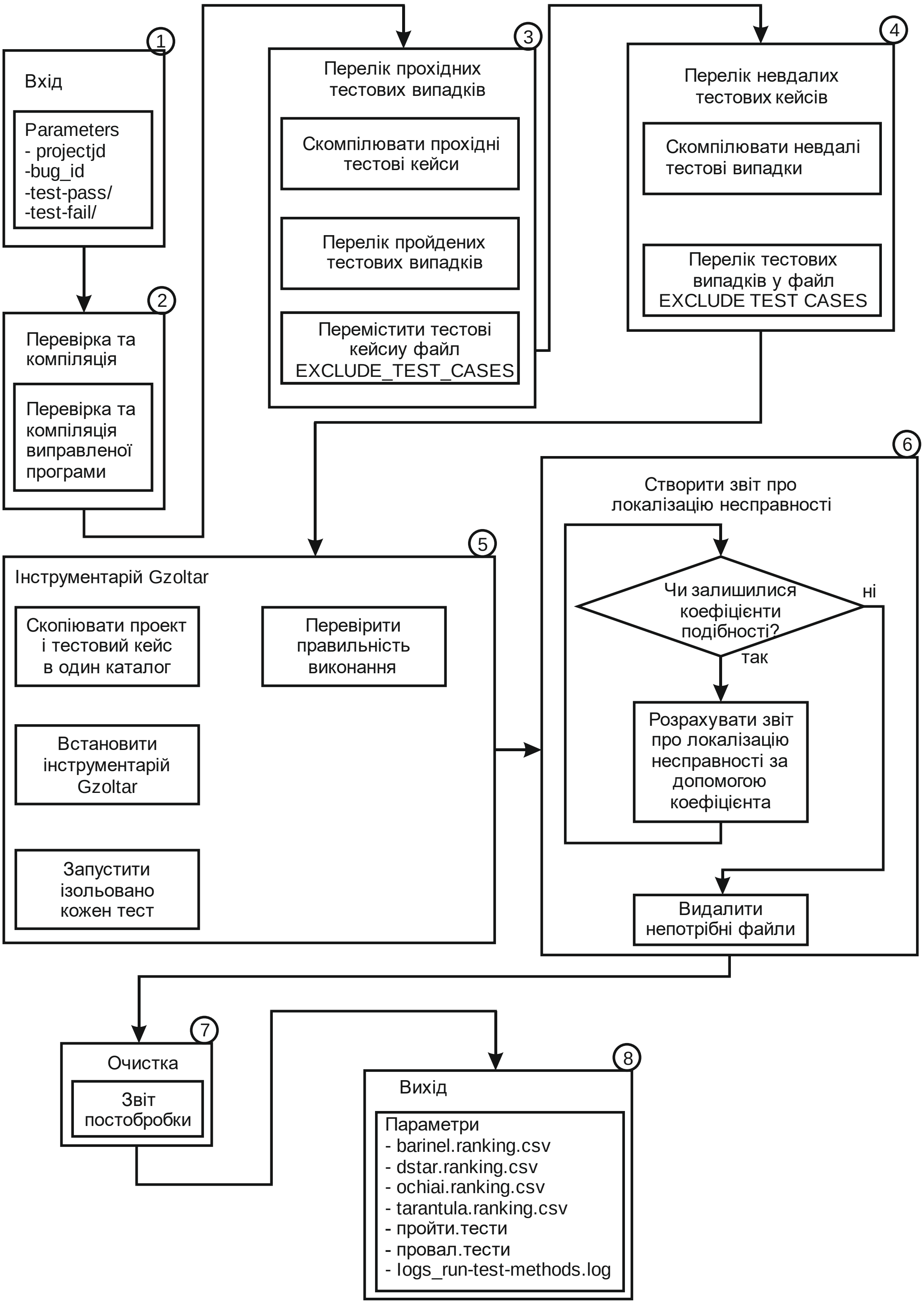


Рисунок 5.7 – Блок-схема контейнера GZoltar

Вхідні дані (1) є project-id та bug_id проекту DEFECTS4J. Разом з каталогом, що містить вдалі тестові приклади (test-pass/) і каталогом, що містить невдалі тестові випадки (test-fail/).
]

На етапі перевірки та компіляції (2) несправна версія проектів DEFECTS4J перевіряється та компілюється за допомогою DEFECTS4J CLI. Використовується несправна версія, оскільки на практиці це буде єдина доступна версія.

Використовуючи несправну версію та каталог, що містить тестові випадки, що пройшли, на кроці (3) створюється список пройдених тестових випадків. Подібно до оригінального робочого процесу GZOLTAR, список створюється шляхом компіляції тестових випадків і використанням команди listTestMethods, наданої в GZOLTAR CLI. Однак індексуються лише тестові випадки в каталозі test-pass/, а тестові випадки в EXCLUDE_TEST_CASES видаляються з цього списку.

Після цього на кроці (4) компілюються невдалі тестові випадки, а тестові приклади, перелічені у файлі INCLUDE_TEST_CASES, додаються до списку тестових випадків, що пройшли. Отже, є список усіх тестових випадків, які використовуються в процесі локалізації несправностей.

Потім крок (5), на якому спочатку скомпільований проект і скомпільовані тестові приклади збираються в одному каталозі. По-друге, інструментарій GZOLTAR додається до всіх класів у каталозі, після чого кожен тестовий приклад виконується ізольовано (як у вихідному робочому процесі). На додаток до вихідного робочого процесу, після виконання тестових випадків, журнали перевіряються, щоб переконатися, що все пройшло, як очікувалося (тобто кожен прохідний тестовий приклад пройшов, а кожен невдалий тестовий приклад не пройшов).

Використовуючи серійний об’єкт з кроку (5), звіти про локалізацію несправностей генеруються для чотирьох коефіцієнтів подібності в таблиці 1.1 на кроці (6). Після створення звітів про локалізацію помилок деякі непотрібні файли видаляються, в тому числі серійний об’єкт (який може стати досить великим).

На передостанньому кроці (7) виконується подальша обробка звітів про локалізацію несправностей. Оскільки проект і тестові приклади об’єднані в одному каталозі, звіти також містять тестові оператори (тобто рядки коду тестового сценарію). Тестові твердження видаляються зі звітів, оскільки вони не мають жодного відношення до збою.

Результатом (8) є звіти про локалізацію несправності (по одному для кожного коефіцієнта подібності) і деяка статистична інформація про процес локалізації несправності.

Щоб запустити *gzoltar-runner*, вхідні дані, як описано вище, повинні бути надані через змінні середовища контейнера DOCKER. Приклад команди для запуску *gzoltar-runner* наведено в лістингу, який складає звіти про локалізацію помилок для LANG-19b, використовуючи тестові приклади проходження з каталогу <*path-to-passing-tests*> і невдалі тести з каталог <*path-to-failing-tests*>, і зберігає результати в поточному каталозі.

```
docker run
   -e PROJECT_ID=Lang
   -e BUG_ID=9
   -v <path-to-passing-tests>:/opt/runner/tests-pass
   -v <path-to-failing-tests>:/opt/runner/tests-fail
   -v $ (pwd)/results:/opt/runner/results/sfl/txt
   gzoltar-runner:latest
```

Після успішного виконання gzoltar-runner результати зберігаються у файлах з вихідного робочого процесу та в таких файлах:

`barinel.ranking.csv`, `dstar.ranking.csv`, `ochiai.ranking.csv`,`tarantula.ranking.csv`, що містить рейтинг висловлювань у проекті, відсортований за рівнем підозрілості, розрахований за допомогою BARINEL, DSTAR, OCHIAI та Коефіцієнт TARANTULA, відповідно.

`gzoitar.tests`, що містить список усіх тестових випадків, які використовуються в процесі локалізації несправностей.

`log_run-test-methods.log`, що містить результати всіх тестів. Якщо тестовий приклад не вдається, журнали містять також створену трасування стека.

3.7 Постобробка даних знаходження несправності

Після того, як усі результати були зібрані, ми застосували до набору даних постобробку. Щоб отримати бали EXAM та двічі перевірити, чи кожна комбінація була виконана правильно.

Для оцінок EXAM ми спочатку склали рейтинг тверджень на основі методології розширеного оцінювання, описаної в Розділі 2.2. Ми також очистили рейтинг, видаливши невиконані оператори та заяви з нульовим рівнем підозрілості. Використовуючи рейтинг і дефектні твердження з набору даних про локалізацію несправностей [24], ми обчислили бали EXAM для кожної з комбінацій тестів.

Щоб визначити, чи перекриває тестовий приклад із відтворенням аварії дефектний оператор (і, отже, покриває помилку), ми переглянули оцінку EXAM. Коли не вдалося підрахувати бали EXAM, це означало, що дефектна відомість не була виконана. Ми можемо сказати це з впевненістю, тому що рейтинг містить лише ті оператори, які були виконані принаймні одним невдалим тестом (інакше рівень підозрілості був би вищим за нуль).

Використовуючи журнали, надані кожним компонентом, ми можемо переконатися, що кожна комбінація виконана правильно. Ми також вручну перевірили, чи кожен компонент було виконано правильно, оскільки в деяких випадках не вдалося виконати тестовий приклад (перевірено в gzoltar-runner), але це пов’язано з іншою причиною, ніж очікувана (наприклад, відсутня бібліотека або неправильний системний часовий пояс).

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

4.1 Реалізація сценарію

Розглянувши загальний сценарій, найкращий сценарій та фактори впливу, ми повинні розглянути тестові випадки, що відтворюють аварійне завершення, створені BOTSING. Як описано в розділі 3.5, ми використали загалом 98 кадрів із 50 збоїв DEFECTS4J.

У зв'язку з тим, що алгоритм BOTSING покладається на випадковість, BOTSING не зміг створити непорожні набори тестів, що відтворюють аварії, для всіх 98 кадрів у наборі даних. У таблиці 1.1 показано, скільки тестових наборів можна було б створити з 98 цільових кадрів із 50 тестованих проектів DEFECTS4J. Наприклад, single-BOTSlNG ($Bot_{fail}1$) створив 83 непорожніх набори тестів (тобто, що містить один тестовий приклад, що відтворює аварійне завершення роботи) з 98 цільових кадрів, а декілька BOTSlNG ($Bot_{fail}+$) створив 85 непорожніх тестових наборів (тобто , що містить принаймні один тестовий приклад із відтворенням аварії). Після видалення неповноцінних тестових випадків (через видалення риштувань EVOSUITE) ми отримали 80 непорожніх тестових наборів $Bot_{fail}1$ і 84 непорожніх тестових наборів $Bot_{fail}+$.

Таблиця 4.1 – Кількість збоїв, для яких BOTSING міг створити набір тестів для відтворення аварійного завершення, який містить один ($Bot_{fail}1$) або кілька ($Bot_{fail}+$) тести для відтворення збоїв

	$Bot_{fail}1$	$Bot_{fail}+$
Збої	50	50
Загальна кількість цільових кадрів	98	98
Непорожні набори тестів, створені BOTSING	83	85
з яких містять нестабільні тести, що відтворюють краш	3	1
з яких виконати дефектну заяву	51	65

З 80 непорожніх наборів тестів $Bot_{fail}1$ 51 містить тестовий приклад, що відтворює аварійне завершення роботи, який виконує дефектні оператори. Для 84 непорожніх наборів тестів Bot_{fail}+ стверджується, що 65 містять тестовий приклад, що відтворює аварійне завершення роботи, який виконує дефектні оператори.

Цікаво, що кількість непорожніх наборів тестів для Bot_{fail}1 і Bot_{fail}+ не однакова, що не зрозуміло. Можна було б очікувати, що якщо BOTSING зможе генерувати непорожній набір тестів з кількома тестовими прикладами, що відтворюють аварійне завершення, то можна було б також створити непорожній набір тестів з одним тестовим прикладом, що відтворює аварійне завершення.

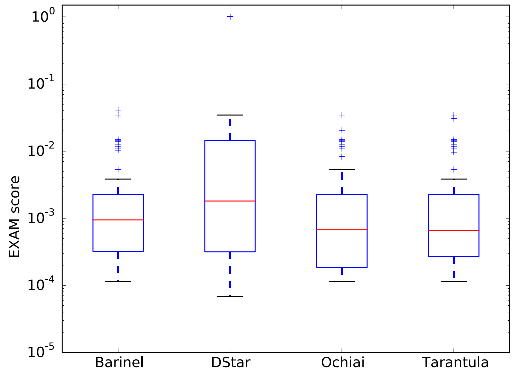
Однак, швидше за все, це пов’язано з тим, що множинний BOTSING виконує вдвічі більше генерації та оцінки в межах того самого бюджету пошуку в 180 секунд. Ми не змогли пояснити причину цього належним чином, оскільки один і кілька BOTSING виконуються з використанням однакової кількості ресурсів (1 чотирьох-ядерний ЦП і 12 Гб пам’яті, що забезпечується параметрами часу виконання DOCKER).

Після видалення нестабільних тестових випадків ми маємо 80 непорожніх наборів тестів, загальних для Bot_{fail}1 і Bot_{fail}+, які можна використовувати для справедливого порівняння результатів для всіх конфігурацій і, отже, для відповіді

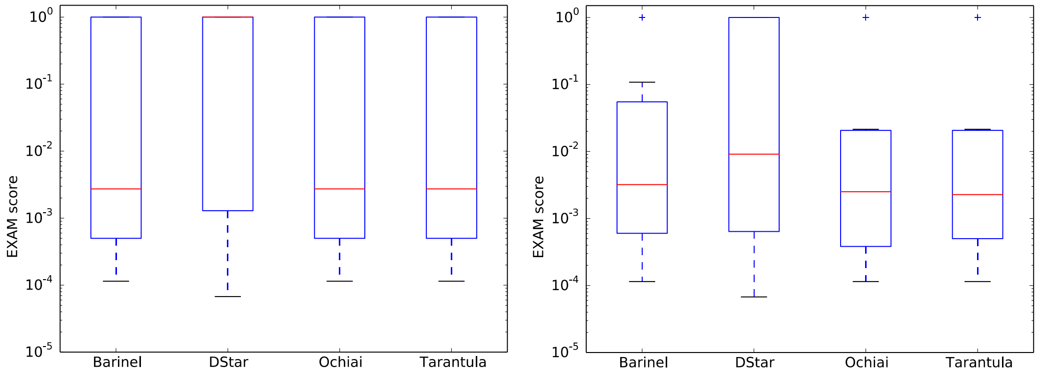
Як описано в Розділі 3.2, метрикою оцінки, яка використовується для визначення точності діагностики підходів до локалізації несправності, є оцінка EXAM. Згадайте, що нижчий бал EXAM означає кращу продуктивність, оскільки він обчислює відсоток висловлювань перед перевіркою помилкового твердження.

На рисунку 4.1 зображено діаграми для результатів ЕКСПІТУ 80 наборів тестів для різних конфігурацій і коефіцієнтів. Середні бали ЕКЗАМ для кожної з комбінацій конфігурацій і коефіцієнтів наведені в таблиці 4.2.

Рукописні тестові приклади (Man_{fail}+-Man_{pass}+) досягають найкращого результату EXAM для кожного з коефіцієнтів подібності в порівнянні з результатами EXAM для автоматизованих конфігурацій локалізації несправностей (див. рисунок 6.1а та таблицю 4.2). Загалом, рукописні тестові приклади (Man_{fai}l+-Man_{pass}+) з коефіцієнтом OCHIAI досягають найкращого результату ЕКЗАМ (див. таблицю 6.2).

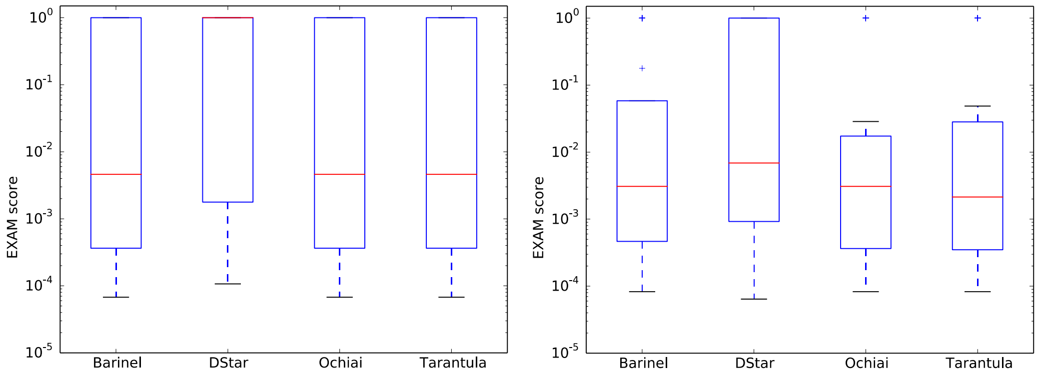


(a) Man$_{fail}$$^{+}$-Man$_{pass}$$^{+}$



(b) Bot$_{fail}$^{1}-Man$_{pass}$$^{+}$

(c) Bot$_{fail}$$^{+}$-Man$_{pass}$$^{+}$



(d) Bot$_{fail}$^{1}-Evo$_{pass}$$^{+}$

(e) Bot$_{fail}$$^{+}$-Evo$_{pass}$$^{+}$

Рисунок 4.1 – Представлення результатів показника EXAM для 80 наборів тестів, загальних для Bot$_{fail}$1 і Bot$_{fail}$+, з тестовим прикладом, що відтворює аварійне завершення роботи (загальний сценарій).

Таблиця 4.2 – Середній показник EXAM із 80 різних комбінацій тестів, спільних для Bot$_{fail}$1 та Bot$_{fail}$+.

Конфігурація	Barinel	DStar	Ochiai	Tarantula
Man$_{fail}$+ -Man$_{pass}$+	0.0031	0.2149	0.0027	0.0029
Bot$_{fail}$1 -Man$_{pass}$+	0.3647	0.5395	0.3647	0.3647
Bot$_{fail}$l+ -Man$_{pass}$ +	0.2418	0.3777	0.2402	0.2399
Bot$_{fail}$1 -Evo$_{pass}$+	0.3647	0.5766	0.3647	0.3647
Bot$_{fail}$+ -Evo$_{pass}$+	0.2422	0.4145	0.2399	0.2403

Цікаво, що конфігурація Man_{fail}+-Man_{pass}+ з коефіцієнтом DSTAR має досить низьку продуктивність порівняно з іншими коефіцієнтами подібності. Таке зниження продуктивності викликано тим, що з 80 страт 17 отримали EXAM 1,0.

Після ручного аналізу ми виявили, що це пов'язано з проблемою в тому, як коефіцієнт DSTAR використовує програмні спектри, як пояснюється в Розділі 2.1. Це означає, що сума невдалих тестових випадків, які не охоплюють помилку, і кількість пройдених тестових випадків, які покривають помилку, дорівнює нулю, що робить коефіцієнт DSTAR непридатним для використовуваних тестових випадків. Ця проблема виникла не лише в рукописному наборі тестів, а й у всіх тестових наборах у нашому наборі даних. Отже, коефіцієнт DSTAR працює найгірше з усіх коефіцієнтів подібності для кожної з конфігурацій. Ми вирішили залишити бали EXAM 1,0 як покарання за такий випадок.

З Рисунку 4.1 і таблиці 4.2 можна помітити, що для конфігурації Bot_{fail}1 -Man_{pass}+ і Bot_{fail}1 -Evo_{pass}+ найкращий результат EXAM отримується за допомогою коефіцієнтів BARINEL, TARANTULA або OCHIAI (рисунок 4.1b і 4.1d). . Найкраща оцінка EXAM для конфігурації Bot_{fail}+-Man_{pass}+ досягається за допомогою коефіцієнта TARANTULA (Рисунок 4.1c), але відмінності з коефіцієнтами OCHIAI та BARINEL невеликі (таблиця 4.2). Bot_{fail}1 -Evo_{pass}+ має майже таку саму продуктивність порівняно з Bot_{fail}1 -Man_{pass}+, досягнуту лише за допомогою коефіцієнта OCHIAI (Рисунок 4.1e), але різниця з коефіцієнтом TARANTULA невелика.

Рисунок 4.1 та таблиця 4.2 показують, що автоматизовані конфігурації з використанням кількох тестових випадків, що відтворюють аварійне завершення роботи (Bot_{fail}+), працюють краще, ніж конфігурації з використанням одного тестового випадку з відтворенням збоїв (Bot_{fail}1). Крім того, для всіх автоматизованих конфігурацій локалізації аварії вважається, що коефіцієнти подібності OCHIAI і TARANTULA дають найкращий результат, а BARINEL займає друге місце.

У таблиці 4.3 наведено результати парного турнірного рейтингу різних комбінацій конфігурацій і коефіцієнтів, як описано в розділі 2.5, для загального сценарію. Ранг створюється шляхом порівняння парних результатів EXAM, присуджуючи 1 бал переможцю, якщо він працює статистично значно краще (p-value < 0,05 і A12 < 0,5).

Таблиця 4.3 – Рейтинг усіх комбінацій тестів конфігурації та коефіцієнтів.

	Конфігурація	Коефіцієнт	Кращий показник	avg. A12
1	Man_{fail}+ -Man_{pass}+	OCHIAI	16	0.30
2	Man_{fail}+ -Man_{pass}+	TARANTULA	16	0.30
3	Man_{fail}+ -Man_{pass}+	BARINEL	16	0.31
4	Man_{fail}+ -Man_{pass}+	DSTAR	8	0.37
5	Bot_{fail}+ -Evo_{pass}+	OCHIAI	4	0.37
6	Bot_{fail}+ -Evo_{pass}+	TARANTULA	4	0.37
7	Bot_{fail}+ -Man_{pass}+	OCHIAI	3	0.36
8	Bot_{fail}+ -Man_{pass}+	TARANTULA	3	0.36
9	Bot_{fail}+ -Evo_{pass}+	BARINEL	2	0.36
10	Bot_{fail}+ -Man_{pass}+	BARINEL	2	0.37
11	Bot_{fail}1 -Evo_{pass}+	BARINEL	2	0.39
12	Bot_{fail}1 -Evo_{pass}+	OCHIAI	2	0.39
13	Bot_{fail}1 -Evo_{pass}+	TARANTULA	2	0.39
14	Bot_{fail}+ -Man_{pass}+	BARINEL	2	0.40
15	Bot_{fail}+ -Man_{pass}+	OCHIAI	2	0.40
16	Bot_{fail}+ -Man_{pass}+	TARANTULA	2	0.40
17	Bot_{fail}+ -Man_{pass}+	DSTAR	1	0.41

З таблиці 4.3 ми можемо уявити, що рукописні комбінації тестів (Man_{fail}+ -Man_{pass}+) значною мірою виграють. Однак ручна комбінація з DSTAR краща лише в половині випадків порівняно з іншими комбінаціями вручну.

Крім того, для автоматизованого підходу до локалізації помилок збоїв ми можемо помітити, що використання кількох тестових випадків, що відтворюють аварії (Bot_{fail}+), працює краще, ніж використання одного тестового випадку з відтворенням збоїв (Bot_{fail}1). Оскільки конфігурації Bot_{fail}+ перемагають у 25 спробах, а Bot_{fail}1 перемагають лише в 7 випадках. Це підтверджує наше спостереження за діаграмами на рисунку 4.1 та результатами з таблиці 4.2.

Крім того, зауваження щодо коефіцієнтів подібності справедливо, оскільки OCHIAI та TARANTULA мають найкращий коефіцієнт для автоматизованого підходу до локалізації аварії. Далі йде BARINEL, який виграє принаймні один раз у кожній з комбінацій, і, нарешті, DSTAR має лише одну перемогу за комбінацією Bot_{fail}+ -Man_{pass}+.

Наші результати підтверджують, що немає суттєвої різниці між чотирма коефіцієнтами подібності при застосуванні до несправностей у реальному світі за допомогою рукописних тестових випадків, як показав Персон [24]. Жоден з коефіцієнтів подібності з використанням комбінації рукописного набору тестів (Man_{fail}+-Man_{pass}+) не перемагає проти інших комбінацій рукописних тестів (тобто p- value > 0,05).

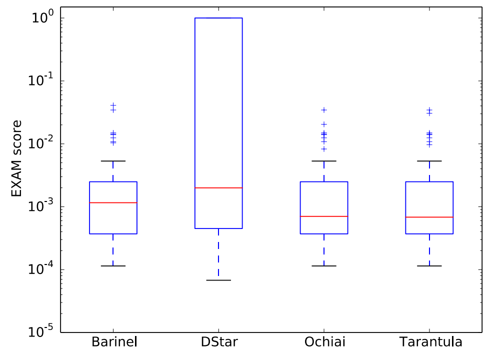
Загалом, рукописні тестові приклади (Man_{fail}+-Man_{pass}+) працюють краще, ніж автоматизовані конфігурації локалізації помилок збоїв. Однією з причин є те, що згенерований тестовий приклад із відтворенням аварії не дає гарантії, що тестовий випадок справді викликає основну помилку. Наприклад, коли помилка розташована в кадрах, вищих за відтворений кадр.

Для автоматизованих конфігурацій локалізації аварійних збоїв найкращі результати дають коефіцієнти подібності OCHIAI або TARANTULA в поєднанні з набором тестів, що містить кілька тестових випадків, що відтворюють аварії (Bot_{fail}+).

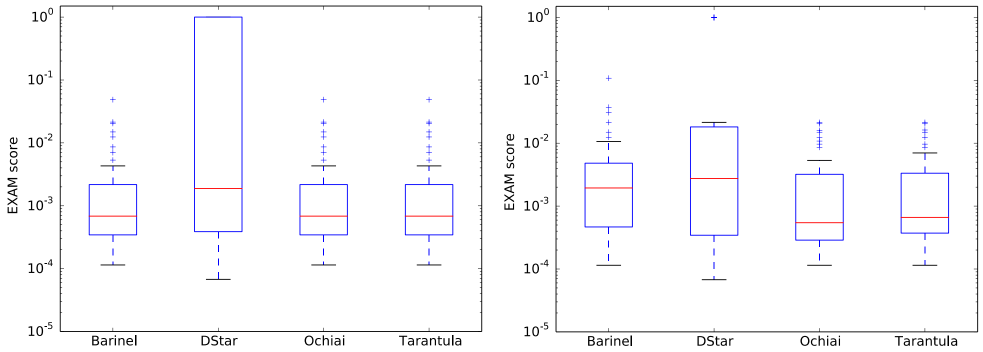
4.2 Кращий сценарій

Як записано в таблиці 4.1, BOTSING міг створити 51 (з 80) наборів тестів для Bot_{fail}1 і 65 (з 84) тестових наборів для Bot_{fail}+ для різних цільових кадрів. Для цих наборів тестів принаймні один тестовий випадок, що відтворює аварійне завершення, покриває один із дефектних операторів. Ми зосередимося на цих 51 наборах тестів, загальних для Bot_{fail}1 та Bot_{fail}+.

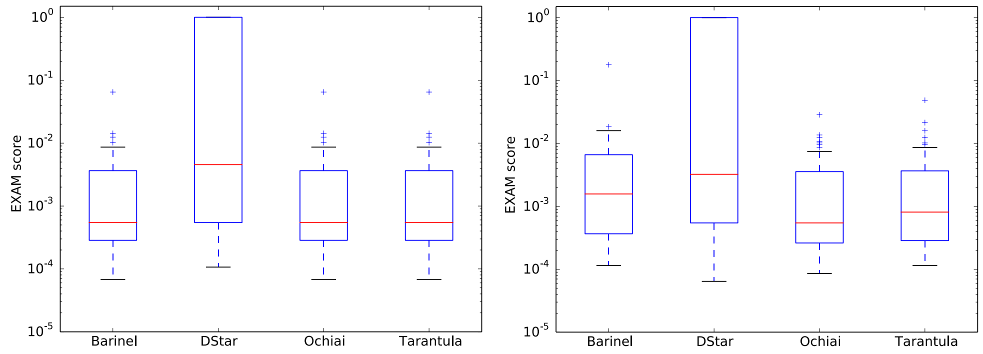
На рисунку 4.2 зображено оцінку EXAM за 51 набором тестів для різних конфігурацій і коефіцієнтів. Одне, що спадає на думку, – це продуктивність автоматизованих конфігурацій локалізації збоїв, які покращилися порівняно з продуктивністю в загальному сценарії див. рисунок 4.1. Як підозрюється, частково низьку продуктивність автоматизованих конфігурацій локалізації аварійних збоїв можна пояснити тим фактом, що немає гарантії, що тестові випадки, що відтворюють аварію, покривають несправність.



(a) Man_{fail}^{+}-Man_{pass}^{+}



(b) Bot_{fail}^{1}-Man_{pass}^{+} (c) Bot_{fail}^{+}-Man_{pass}^{+}



(d) Bot_{fail}^{1}-Evo_{pass}^{+} (e) Bot_{fail}^{+}-Evo_{pass}^{+}

Рисунок 4.2 – Оцінка EXAM за 51 тестовий набір із невдалим тестовим прикладом, що відтворює збій, що викликає помилку (найкращий сценарій)

У середньому, OCHIAI є найкращим коефіцієнтом подібності для конфігурацій Man_{fail}+-Man_{pass}+, як показано в таблиці 4.4. Для конфігурацій Bot_{fail}+-Man_{pass}+, використовуючи коефіцієнт TARANTULA або OCHIAI, досягайте кращого середнього бала EXAM. Найкраща продуктивність досягається за допомогою конфігурації Bot$_{\text{fail}}$+-Evo$_{\text{pass}}$+ у поєднанні з коефіцієнтом подібності OCHIAI. Подібно до загального сценарію, для конфігурацій Bot$_{\text{fail}}$1 -Man$_{\text{pass}}$+ і Bo$_{\text{fail}}$1 -Evo$_{\text{pass}}$+ стверджується, що найкращий результат EXAM досягається за допомогою коефіцієнта BARINEL, OCHIAI або TARANTULA.

Таблиця 4.1 – Усереднена оцінка EXAM із 51 різної непорожньої комбінації тестів.

Конфігурація	BARINEL	DSTAR	OCHIAI	TARANTULA
Man$_{\text{fail}}$+ -Man$_{\text{pass}}$+	0.0039	0.2776	0.0034	0.0036
Bot$_{\text{fail}}$1 -Man$_{\text{pass}}$+	0.0035	0.2776	0.0035	0.0035
Bot$_{\text{fail}}$+ -Man$_{\text{pass}}$+	0.0064	0.2188	0.0032	0.0031
Bot$_{\text{fail}}$1 -Evo$_{\text{pass}}$+	0.0035	0.3359	0.0035	0.0035
Bot$_{\text{fail}}$+ -Evo$_{\text{pass}}$+	0.0071	0.2768	0.0030	0.0038

Останнє можна пояснити, розглянувши формули трьох коефіцієнтів подібності. Для обох одиничних конфігурацій, що відтворюють аварійне завершення (Bot$_{\text{fail}}$1), вважається, що $N_F = 1$, оскільки існує лише тестовий приклад для відтворення аварій. Використовуючи припущення, визначене в розділі 2.1 (для дефектного оператора $N_{CF} > 0$), ми можемо зробити висновок, що для дефектного оператора в конфігураціях, що відтворюють один збій, має бути, що $N_{CF} = 1$ = 1 і $N_{UF} = 1$. Коли ми застосовуємо значення трьох коефіцієнтів подібності, отримуємо, що:

$$\text{OCHIAI} = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}} = \frac{1}{\sqrt{1 \times (1 + N_{CS})}} = \frac{1}{\sqrt{1 + N_{CS}}}$$

$$\text{BARINEL} = 1 - \frac{N_{CS}}{N_{CS} + N_{CF}} = 1 - \frac{N_{CS}}{N_{CS} + 1} = \frac{1}{1 + N_{CS}}$$

$$\text{TARANTULA} = \frac{N_{CF}/N_F}{N_{CF}/N_F + N_{CS}/N_S} = \frac{1/1}{1/1 + N_{CS}/N_S} = \frac{1}{1 - N_{CS}/N_S}$$

З цього ми можемо зробити висновок, що ці три коефіцієнти подібності математично подібні і демонструють однакову поведінку (тобто всі формули залежать лише від N_{CS}, враховуючи, що N_S фіксований для всіх дефектних операторів) при застосуванні до програми, що містить одну помилку. тестовий приклад. Це призводить до того, що ранжування тверджень за всіма цими коефіцієнтами однакове, а отже, і оцінка EXAM.

Порівнюючи різні коефіцієнти подібності з рисунків 4.2b і 4.2c, ми бачимо, що використання Man$_{pass}$+ у поєднанні з Bot$_{fail}$1 або Bot$_{fail}$+ працює статистично краще (p-value < 0,05) при використанні OCHIAI (A12 = 0,331) або TARANTULA (A342 = 0). більше, ніж коефіцієнт DSTAR.

Коефіцієнт BARINEL у поєднанні з одиничними конфігураціями, що відтворюють аварії (Bot$_{fail}$1) (A 12 = 0,331), також досягає статистично значущої різниці щодо тих самих конфігурацій за допомогою коефіцієнта DSTAR.

Якщо розглядати лише непорожні набори тестів, які гарантовано покривають помилку, ми бачимо, що жодна комбінація коефіцієнтів подібності не перевершує інші, за винятком коефіцієнта DSTAR.

Крім того, ми не помічаємо жодної статистичної різниці (тобто p-значення > 0,05)), коли даний коефіцієнт подібності використовується в поєднанні з рукописною конфігурацією тестових випадків (Man$_{fail}$+-Man$_{pass}$+) або будь-якими іншими автоматизованими конфігураціями, що відтворюють збої.

Таблиця 4.5 – Рейтинг із 51 різних комбінацій тестового набору конфігурації та коефіцієнта.

№	Конфігурація	Коефіцієнт	Кращий	avg. Â12
1	Bot$_{fail}$1 -Evo$_{pass}$+	BARINEL	5	0.32
2	Bot$_{fail}$1 -Evo$_{pass}$+	OCHIAI	5	0.32
3	Bot$_{fail}$1 -Evo$_{pass}$+	TARANTULA	5	0.32
4	Bot$_{fail}$+ -Evo$_{pass}$+	OCHIAI	5	0.33
5	Bot$_{fail}$+ -Man$_{pass}$+	OCHIAI	5	0.34
6	Bot$_{fail}$1 -Man$_{pass}$+	BARINEL	5	0.35
7	Bot$_{fail}$1 -Man$_{pass}$+	OCHIAI	5	0.35
8	Bot$_{fail}$1 -Man$_{pass}$+	TARANTULA	5	0.35
9	Bot$_{fail}$+ -Evo$_{pass}$+	TARANTULA	5	0.35
10	Man$_{fail}$+ -Man$_{pass}$+	OCHIAI	5	0.35
11	Bot$_{fail}$+ -Man$_{pass}$+	TARANTULA	5	0.36
12	Man$_{fail}$+ -Man$_{pass}$+	TARANTULA	5	0.36
13	Man$_{fail}$+ -Man$_{pass}$+	BARINEL	5	0.37
14	Bot$_{fail}$+ -Evo$_{pass}$+	BARINEL	1	0.37
15	Bot$_{fail}$+ -Man$_{pass}$+	BARINEL	1	0.38

У таблиці 4.5 наведено результати парного турнірного рейтингу різних комбінацій конфігурацій і коефіцієнтів, як описано в розділі 1.5, для найкращого сценарію.

З таблиці 4.5 ми можемо помітити, що коефіцієнт DSTAR працює найгірше з усіх коефіцієнтів, оскільки він працює не краще, ніж будь-яка з інших комбінацій.

Крім того, ми бачимо, що жодна комбінація значно не перевершує інші (крім DSTAR), оскільки всі комбінації мають однакову кількість виграшів. Лише продуктивність Bot_{fail}+-Man_{pass}+ і Bot_{fail}+-Evo_{pass}+ з BARINEL трохи нижча (1 перемога проти 5 перемог), але ця різниця статистично не значуща.

У кращому випадку ми не спостерігаємо статистично значущої різниці між автоматичними конфігураціями локалізації помилок аварії (Bot_{fail}1 -Man_{pass}+, Bot_{fail}+-Man_{pass}+, Bot_{fail}1-Evo_{pass}+, Bot_{fail}+-Evo_{pass}+) та рукописними тестовими прикладами (Man_{fail}+-Man_{pass}+), які були там у загальному сценарії. Жодна комбінація конфігурації та коефіцієнта подібності не перевершує інші комбінації.

Наші результати показують, що використання коефіцієнтів OCHIAI або TARANTULA дає найкращий результат EXAM. Однак при використанні одного тестового випадку з відтворенням збоїв (Bot_{fail}1) не має значення, який із коефіцієнтів використовується (крім DSTAR), оскільки в цьому випадку коефіцієнти математично подібні і залежать лише від N_{CS}.

Нарешті, результати свідчать про те, що коефіцієнт DSTAR може бути не придатним для використання в автоматизованій локалізації несправностей.

4.3 Фактори впливу

Нарешті, ми визначили фактори, які впливають на діагностичну точність підходу автоматичної локалізації помилок аварії, ми класифікували виконання в Bot_{fail} > Man_{fail}+, Bot_{fail} = Man_{fail}+ і Bot_{fail} < Man_{fail}+, як описано в Розділі 2.5.

Для кожної категорії ми вручну дослідили потенційні фактори, що впливають на точність локалізації несправностей на основі спектру, визначивши сильні та слабкі сторони автоматично створених тестових випадків, що відтворюють аварії. Далі надається ідентифікований фактор для кожної категорії з репрезентативним прикладом.

Згідно з існуючою літературою [80, 81, 57], кількість висловлювань підтвердження на тестовий приклад впливає на діагностичну точність підходів до локалізації несправностей на основі спектру. Зокрема, коли тестовий приклад містить кілька операторів assert, у яких один і той же метод виконується кілька разів

Лістинг 4.1

```
public void test00() {
  assertTrue(isValidPhoneNumber("06 -12345678"));
  assertTrue(isValidPhoneNumber("0031612345678"));
  assertTrue(isValidPhoneNumber("+31612345678"));
}
```

Якщо тестовий приклад містить кілька тверджень, і одне з них зазнає невдачі, то локалізація помилок на основі спектру не може відрізнити невдалий шлях виконання від безвідмовних (наприклад, коли другий оператор assert дає збій). Це явище, весь шлях виконання позначено як невдалий, що призводить до зниження точності діагностики. У своїй роботі Хуан і Монперус [81] показали, що очищені тестові випадки, тобто тестові випадки, що містять один оператор assert, який виконує один метод, можуть позитивно впливати на продуктивність локалізації помилок на основі спектру, оскільки помилка на основі спектру Підхід до локалізації може краще розрізняти шляхи виконання (збій або проходження). Крім того, при використанні очищених тестових випадків усі твердження будуть виконуватися замість тверджень, доки одне не вийде, що надає більше інформації.

Лістинг 4.2

```
public void test00() {
  assertTrue(isValidPhoneNumber("06 -12345678"));
}

public void test01() {
```

```
  assertTrue(isValidPhoneNumber("0031612345678"));
}
public void test02() {
  assertTrue(isValidPhoneNumber("+31612345678"));
}
```

Ми помічаємо, що для кількох автоматизованих конфігурацій локалізації помилок аварії ($\mathrm{Bot}_{\mathrm{fail}}$) мають кращу продуктивність, ніж рукописна конфігурація ($\mathrm{Man}_{\mathrm{fail}}$+-$\mathrm{Man}_{\mathrm{pass}}$+), коли рукописні тестові випадки не є очищеними тестами. У цих конфігураціях рукописні невдалі тестові випадки ($\mathrm{Man}_{\mathrm{fail}}$+) містять кілька операторів, які виконують один і той же метод кілька разів (тобто мають формат, показаний у листингу 4.1).

Наприклад, один із рукописних тестових випадків для LANG-19b наведено в лістингу 6.1. Цей тестовий приклад не вдається, оскільки вхід, використаний у рядку 4, викликає основну помилку. У цьому випадку шляхи виконання першого, другого і третього виконання методу translate будуть позначені як помилкові в спектрі програми. Однак вхідні дані, які використовуються в першому і другому виконанні, є дійсними, що може внести шум у процес локалізації збою. Через неочищені тестові випадки виникає ймовірність того, що недефектний оператор безпідставно підозрюється у несправності через помилку, що ініціює виконання в тому ж тестовому випадку.

Лістинг 4.3

```
public void testOutOfBounds() {
  NumericEntityUnescaper neu = new NumericEntityUnescaper();
  assertEquals("Test &", neu.translate("Test &"));
  assertEquals("Test &#", neu.translate("Test &#"));
  assertEquals("Test &#x", neu.translate("Test &#x"));
  assertEquals("Test &#X", neu.translate("Test &#X"));
}
```

Цей ефект (тобто неочищені тестові випадки) не з’являється в тестових випадках, що відтворюють аварійне завершення, оскільки BOTSING генерує лише одне невдале виконання на тестовий приклад, тож певним чином існує лише одне твердження. Таким чином, для кожного тестового випадку, що відтворює аварійне завершення роботи, лише один шлях виконання позначається як невдалий. Це призводить до підвищення точності діагностики для тестових випадків, що відтворюють аварії (Bot$_{fail}$) порівняно з конфігурацією Man$_{fail}$+-Man$_{pass}$+.

Для 69% збоїв, які створюють невелике трасування стека (< 2 кадри), точність діагностики при використанні наборів тестів Botfan дорівнює або краща, ніж при використанні тестових наборів Man$_{fail}$+. Основною причиною, здається, є низька кількість параметрів несправних методів, тому що невеликі збої зазвичай включають менші методи (наприклад, допоміжні методи). У цьому випадку BOTSING може досягти високого охоплення гілок (завдяки додатковому цілі рознесення послідовності методів), що призводить до більш ефективних тестових випадків для локалізації несправностей на основі спектру.

Однак після перевірки пакетів тестів Bot$_{fail}$ і Man$_{fail}$+ помічено різницю у вхідних значеннях, які використовуються для ініціювання збою. Менші збої в основному пов’язані з несподіваними вхідними значеннями, які, як припускають розробники методу, ніколи не відбуваються (і все ж це сталося через крайній випадок у системі).

Наприклад, LANG-1b містить метод createNumber(), який неправильно обробляє великі шістнадцяткові числа. Метод використовує довжину шістнадцяткового рядка, щоб визначити, чи має шістнадцятковий рядок створити Integer, Long або BigInteger. Однак метод використання довжини рядка може визначити, в який об’єкт має поміститися число, є неправильним. Оскільки межа між Integer і Long знаходиться між 8-значними шістнадцятковими числами "0x7FFFFFFF" і "0x80000000". Тому в несправній версії програми шістнадцятковий рядок «0x80000000» парситься на об’єкт Integer, що викликає збій.

Дивлячись на ефективність локалізації несправностей на основі спектру комбінацій Botfail+-Manpass+ і Botfail+-Evopass+, ми бачимо, що вони порівнянні з продуктивністю Manfail+-Manpass+ з результатом EXAM 0,00035. Однак, перевіряючи тестові випадки, що відтворюють аварійне завершення роботи, ми помітили, що вхідні дані, згенеровані BOTSING, не схожі на ті, що використовуються в рукописних тестових кейсах (тобто всі ці вхідні дані починаються з "0x" або "-0x"). . Більшість вхідних даних, згенерованих BOTSING, є випадковими рядками, такими як "S/VZ9k'&" і "4j8cvkguH". Ці випадкові рядки викликають трасування стека, подібне до трасування стека, викликаного використанням входу "0x80000000", обидва вхідні дані викликають виняток аналізу (незалежно від того, з абсолютно різних причин чи ні).

Ці результати дають цікаве уявлення про тестові випадки, що відтворюють аварії, створені за допомогою BOTSING. Хоча подібні тестові приклади з випадковими вхідними рядками, здавалося б, мало допоможуть розробникам, їх все одно можна використовувати для точного визначення несправного оператора за допомогою локалізації помилок на основі спектру.

На додаток до перших двох факторів, ми помічаємо, що цільовий кадр стека аварії, для якого генеруються тестові випадки, впливає на точність діагностики локалізації помилок на основі спектру.

Наприклад, для аварії TIME-5b з трасуванням стека, що складається з 3 кадрів, і помилкою, розташованою в методі третього кадру (тобто дефектний кадр є останнім кадром у трасі стека). З 3-х кадрів BOTSING може генерувати тестові випадки, що відтворюють аварії, для другого (кадр 2) і третього (кадр 3). Загалом, BOTSING може безпосередньо викликати цільовий метод у тестовому випадку (наприклад, якщо цільовий метод є відкритим) і опосередковано (наприклад, коли цільовий метод не видимий або викликається іншим методом, викликаним у тестовому випадку).

При застосуванні BOTSING на TIME-5b із цільовим кадром встановлено значення 2, лише 80% тестових випадків, що відтворюють аварії, викликають цільовий метод кадру 2. Інші 20% охоплюють кадр 2, викликаючи той самий метод, що і вказаний у кадрі 3, тому, по суті, це тестовий приклад для цільового кадру 3. Ці тести, що відтворюють аварійне завершення, у поєднанні з одиничними тестами, створеними EVOSUITE (Bot_{fail}+-Evo_{pass}+), дають результат EXAM 0,03329.

Однак при безпосередньому націлюванні на кадр 3 ми спостерігаємо значне покращення оцінки EXAM (0,00022) для тієї ж конфігурації (тобто Bot_{fail}+-

Evo$_{pass}$+). Основним фактором цього покращення є зміна співвідношення тестових випадків, що безпосередньо залучають неправильний метод кадру 3 (100%). Іншими словами, під час націлювання на кадр 2 тестові випадки, які безпосередньо викликають цільовий метод кадру 2, додають шум до спектру програми. в принципі, ці тести, що відтворюють аварійне завершення роботи, призводять до того, що шляхи виконання, які, швидше за все, є правильними (тобто BOTSING порушує передумову), позначаються як несправні.

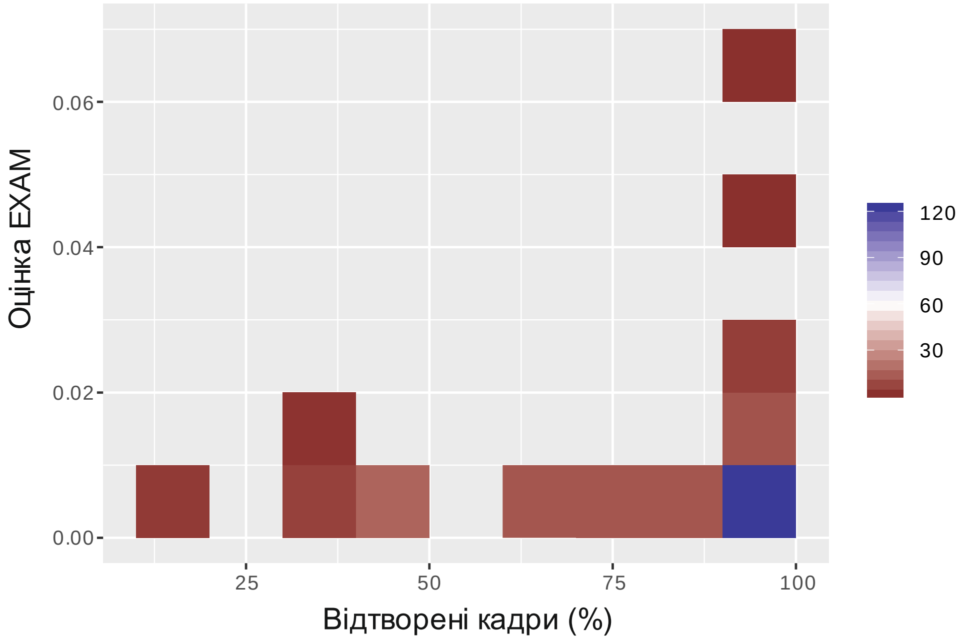


Рисунок 4.3 – Розподіл балів EXAM при використанні коефіцієнта OCHIAI з різними наборами тестів, створених BOTSING. Відтворені кадри вказують на відсоток кадрів, відтворених створеним набором тестів.

Ця тенденція підтверджується при перегляді результатів ЕКЗАМ комбінації за допомогою тестових наборів Bot$_{fail}$. На РИСУНКУ 4.3 зображено розподіл балів EXAM при використанні коефіцієнта OCHIAI у поєднанні з наборами тестів Bot$_{fail}$. Найкращий результат EXAM досягається при використанні наборів тестів, створених для найвищих кадрів у трасі стека. Ці результати свідчать про те, що при використанні BOTSING слід орієнтуватися на найвищий можливий фрейм, щоб охопити якомога більше методів. Це підтверджує попередні спостереження, зроблені пов'язаними дослідженнями щодо ручної ідентифікації несправностей за допомогою автоматичного відтворення аварій [67,69].

На ефективність автоматичної локалізації несправностей впливають різні фактори, пов'язані з тестовими прикладами, що відтворюють аварійне завершення роботи. Очищений характер тестів, що відтворюють аварійне завершення роботи, створених BOTSING, є корисним, якщо цільовий кадр не знаходиться нижче дефектного кадру. Оскільки це неможливо визначити заздалегідь, при використанні BOTSING завжди слід націлюватися на найвищий кадр. Результати показують, що форма вхідних даних не має відношення до локалізації несправностей на основі спектру, якщо вона викликає збій.

5 ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО ПІДХОДУ

5.1 Коефіцієнти подібності

Під час оцінки ми порівняли чотири найкраще вивчені коефіцієнти подібності. На рисунку 4.2 представлено розподіл балів EXAM відповідних наборів тестів для різних конфігурацій. Серед різних коефіцієнтів коефіцієнт DSTAR працює найгірше. Це суперечить результатам, опублікованим Персоном. [24], що показує, що немає суттєвої різниці між DSTAR та OCHIAI та TARANTULA при оцінці помилках у реальних програмних продуктах.

За допомогою ручного аналізу ми виявили, що погану продуктивність можна простежити до (неявного) припущення, зробленого дослідником за коефіцієнтом DSTAR. Якщо оператор виконується всіма невдалими тестовими випадками $(N_{UF} = 0)$, і якщо немає прохідного тесту для виконання оператора $(N_{CS} = 0)$, то формула DSTAR стає недійсною (тобто DSTAR = $(N_{CF} = 0)$). Це сталося в 11 (з 50) збоїв під час використання рукописних тестових випадків (Man_{fail}+-Man_{pass}+).

Допустиме припущення, що різні існуючі коефіцієнти подібності роблять те саме (неявне) припущення, що (потенційно) помилковий оператор покривається принаймні одним прохідним тестовим прикладом. Ми базуємо цю гіпотезу на основі набору даних, використаного для створення та оцінки цих коефіцієнтів подібності. Чотири коефіцієнти раніше оцінювалися переважно з використанням контрольного набору Сіменса [23, 25, 27, 82]. Цей набір тестів містить набори тестів, які задовольняють властивість, що кожна гілка програми покривається щонайменше 30 тестами [82]. Ця властивість підтверджує нашу гіпотезу, оскільки кожен оператор програми принаймні один раз покривається успішним тестом.

В даний час значна частина існуючих коефіцієнтів подібності базується на рукописних тестах [2]. Оскільки підхід автоматичної локалізації несправностей аварії покладається на створені тестові випадки, які мають інші характеристики порівняно з написаними від руки тестові випадки (наприклад, тестові випадки, що відтворюють аварії, очищаються), ми вважаємо, що він впливає на точність автоматизованої локалізації помилок збою за допомогою існуючі коефіцієнти подібності.

5.2 Програмні зміщення спектрів

Як зазначено в Розділі 4.3, вибір цільового кадру для генерації набору тестів, що відтворює краш, може вплинути на результат процесу локалізації несправності. У прикладі результат значно змінився під час націлювання на кадр 3 через зміну співвідношення тестових випадків, які безпосередньо викликали помилковий метод. Однак точність діагностики може бути знижена не тільки шляхом націлювання на інший кадр. Це також можливо при націлюванні на той самий кадр, оскільки генератор тестових прикладів може надавати перевагу певним шляхам виконання.

Наприклад, для аварії TIME-5b, коли націлювались на кадр 3, BOTSING створив 60 тестових випадків, що відтворюють аварії, які мають лише два різних шляхи виконання через помилковий метод. На перший погляд, це не проблема. Однак більш детальне дослідження показує, що розподіл 60-ти тестових проходів між цими двома шляхами виконання не є збалансованим (тобто 59 для першого шляху і лише 1 для другого).

Зрештою, відсутність балансу не вплинула на точність діагностики для TIME-5b, тому що дефектний оператор знаходиться на шляху виконання, охопленому 59 тестами, що відтворюють крах. Однак це може бути не завжди так.

У своїй поточній реалізації BOTSING генерує набір тестів, що містить необмежену кількість тестових випадків, що відтворюють аварію, використовуючи три цілі: мінімізацію відстані аварії, головну мету, яку необхідно досягти, щоб відтворити аварію, і дві допоміжні цілі. Максимізація різноманітності послідовності методів і мінімізація довжини тесту для полегшення процесу пошуку. Ці три цілі теоретично забезпечують те, що BOTSING створив різноманітні тестові випадки, що відтворюють крах. Однак різноманіття методів, викликаних у тестових випадках, що відтворюють аварійне завершення, не гарантує, що це розмаїття буде відображено в шляхах виконання до основної помилки (як це має місце під час націлювання на третій кадр аварії TIME-5b).

Приклад двох тестових випадків, згенерованих BOTSING, показаний у лістингу 5.1 (припускаємо, що помилка в методі createNumber()). Тестові випадки test01 і test02 мають різні шляхи виконання, оскільки test02 має додатковий виклик методу в рядку 7. Однак, якщо розглядати його з точки зору основної помилки, шляхи виконання через несправний метод однакові, тому що додатковий оператор не змінює шлях виконання через дефектний метод.

Лістинг 5.1 – Приклад коду двох тестових випадків, які мають різноманіття методів. Однак це різноманіття не відображається на шляхах виконання до основної помилки (припускаючи, що помилка знаходиться в методі createNumber()).

```
0 @Test(timeout = 4000)
1 public void test01() throws Throwable {
2   NumberUtils.createNumber("hnQf");
3 }
4
5 @Test(timeout = 4000)
6 public void test02() {
7   NumberUtils.isParsable(null);
8   NumberUtils.createNumber("hnQf");
9 }
```

Одним із способів збалансувати розподіл виконання між різними шляхами виконання за допомогою несправного методу є використання метрики щільності-різноманіття-унікальності (DDU) [56]. Метрика DDU кількісно визначає діагностику набору тестів, а максимізація метрики підвищує ефективність локалізації несправностей на основі спектру [56]. У своїй роботі Перес та ін. [57] використовують DDU як (основну) мету пошуку для створення модульних тестів за допомогою EVOSUITE. Однак це може бути неможливим безпосередньо для відтворення збоїв, оскільки максимізація метрики DDU може суперечити цілі мінімізації відстані до аварії (тобто DDU прагне охопити унікальні та різноманітні шляхи виконання, тоді як відтворення збоїв цікавить лише ті) . У нашій майбутній роботі ми плануємо проекспериментувати з відтворенням збоїв на основі пошуку з метрикою DDU як основною і другорядною метою, щоб підвищити ефективність створених тестів для локалізації несправностей.

5.3 Генерація тестових наборів

У розділі 4.2 зазначено, що немає статистичної різниці в точності діагностики локалізації несправностей під час використання рукописних (Man$_{fail}$+) або автоматично згенерованих тестових випадків із відтворенням аварії (Botf$_{ail}$1 або Bot$_{fail}$+), якщо ці тестові випадки охоплюють дефектна заява.

Крім того, як показано на рисунку 4.3, найкращий результат EXAM досягається за допомогою тестових випадків з відтворенням аварії, що охоплюють більшість трас стека (тобто тестові випадки з відтворенням аварій, створені для вищих кадрів). Це говорить про те, що тестовий приклад із відтворенням аварій буде корисним для автоматизованої локалізації помилок збою, якщо він охоплює достатньо кадрів, і чим вище, тим краще. Це посилає потужний сигнал дослідницькому співтовариству з відтворення аварій, щоб вони прагнули відтворити якомога найвищі кадри.

Однак націлюватися на більш високі кадри для відтворення збоїв легше сказати, ніж зробити [70]. Відтворення більш високих кадрів вимагає більш просунутих підходів відтворення аварій на основі пошуку [71] і, потенційно, більшої обчислювальної потужності, яка на практиці не завжди може бути доступна в середовищі розробки. Майбутні дослідження повинні досліджувати підходи для досягнення балансу між вартістю обчислень і цільовим рівнем кадру, щоб створити корисні тестові випадки, що відтворюють аварії, наприклад, використовуючи статичний аналіз [83] або прогнозні моделі [84] для виявлення потенційних можливостей. несправні рамки.

5.4 Перевірка достовірності

Було вибрано 50 збоїв із DEFECTS4J, які були попередньо проаналізовані в JCRASH-PACK [70] та наборі даних про локалізацію несправностей Пірсон та ін. [24]. Ми також використовували BOTSING, EVOSUITE та GZOLTAR із конфігурацією, описаною в розділі 3. Ми не можемо гарантувати, що ці інструменти не мають дефектів, але EVOSUITE та GZOLTAR є довгостроково встановленими найсучаснішими, вивченими, які використовуються багатьма користувачами, BOTSING є свіжою і добре перевіреною реалізацією підходу EVOCRASH [69].

Було розроблено автоматизований конвеєр локалізації помилок, що призводять до аварійного завершення роботи, в цілому (наскрізний). З огляду на випадковість, пов’язану з генерацією тестових випадків, майбутня робота повинна включати розширену оцінку конвеєра шляхом зосередження та повторення генерації тестових випадків, що відтворюють аварії. Вважаю, що це дасть більше тестів, що відтворюють крах, і підвищить статистичну значущість висновків.

Зовнішня валідність

Не можна гарантувати, що отримані результати можна узагальнити для всіх збоїв. Однак, дотримуючись рекомендацій Пірсона та ін. [24], використовувалося «збої через збої» в реальному програмному забезпеченні. Звичайно, врахування більшої кількості збоїв підвищить довіру до наших результатів, але, враховуючи дослідницький характер цієї роботи, ми вважаємо, що використання меншого набору збоїв, попередньо вивченого як для відтворення збоїв, так і для локалізації несправностей, дасть нам більш глибоке уявлення про результати.

Повторення результатів. Пакет реплікації нашої емпіричної оцінки доступний на Github. Репозиторій містить усі інструменти, створені, як описано в

Розділі 4, у форматі контейнера DOCKER, сценарії для запуску автоматичного конвеєра локалізації збоїв, дані, описані в цій дипломній роботі, включаючи набір даних тестового прикладу, і сценарії для відтворення. статистичний аналіз.

Для детального опису компонентів, які використовуються при оцінюванні, ми звертаємося до документації, наданої в дипломній роботі (Розділ 4). Щоб виконати контейнери DOCKER і сценарії, які використовуються для виконання всього автоматичного конвеєра локалізації помилок збою, ми звертаємося до README.md у сховищі.

ВИСНОВКИ

У цій дипломній роботі представлено, наскільки мені відомо, перший автоматизований конвеєр локалізації несправностей, у якому ми поєднали відтворення аварії на основі пошуку, локалізацію несправностей на основі спектру та аналізу вмісту стеку. Конвеєр створює тестові приклади, що відтворюють аварійне завершення роботи, із трасування стека, включеного в звіт про аварійне завершення роботи. Ці тестові приклади в поєднанні з існуючим або автоматично згенерованим набором тестів використовуються як вхідні дані для підходу до локалізації несправностей. Використовуючи підхід до локалізації помилок на основі спектру, ідентифікуються потенційно несправні рядки коду з основною метою – скорочення зусиль для налагодження для розробників.

Результати нашого емпіричного оцінювання 50 помилок у реальному світі показують, що автоматично створені тестові випадки, що відтворюють аварійне завершення роботи, зменшують кількість заяв, які підлягають дослідженню розробниками. Однак у загальному сценарії написані від руки тестові випадки залишаються найбільш ефективними, оскільки не всі збої ще можна відтворити для достатньо високого кадру (тобто, принаймні, містить дефектний кадр). Зрештою, при розгляді найкращого сценарію, коли тестові випадки, що відтворюють аварії, дійсно покривають несправність, ми не помічаємо статистично значущої різниці між точністю локалізації несправності між рукописними та автоматично згенерованими тестами.

Крім того, наше дослідження чотирьох коефіцієнтів подібності показує, що немає явного переможця (немає статистично значущої різниці). Однак ми можемо з впевненістю стверджувати, що DSTAR не підходить для автоматичної локалізації помилок збоїв через припущення, що заява має бути принаймні покрита успішним тестовим прикладом. Крім того, не має значення, який коефіцієнт подібності (крім DSTAR) застосовується при використанні одного тестового випадку з відтворенням аварій, оскільки в цьому випадку коефіцієнти подібності мають подібну математичну поведінку.

Результати підтверджують доцільність конвеєра автоматичної локалізації збоїв і відкривають нові шляхи до наскрізної автоматизованої локалізації несправностей і автоматизованого відновлення програм.

Показано, реалізований автоматизований підхід до локалізації помилок можливий. Тим не менш, є ще вдосконалення, які можна зробити, натхненні ручним аналізом, проведеним для оцінки, і проблемами, з якими ми зіткнулися під час цього дослідження.

Потенційні вдосконалення конвеєра автоматичної локалізації несправностей можна отримати, дослідивши такі зміни:

– Найважливішим аспектом автоматизованого підходу до локалізації помилок при аварії є те, що створюється тестовий приклад із відтворенням аварії, який охоплює достатньо високий рівень кадру (тобто містить принаймні дефектний кадр). Однак, чим вищий рівень кадру, тим інтенсивнішим буде процес пошуку. Таким чином, буде корисно дослідити підходи до пошуку інформації, щоб визначити прийнятний рівень кадру для відтворення аварії на основі пошуку, збалансування вартості обчислень і точності локалізації несправностей.

– Щоб покращити якість тестових випадків, що відтворюють аварійне завершення роботи, BOTSING можна розширити, включивши нові цілі на основі метрики DDU, запропонованої Пересом та ін. [56]. За допомогою цієї метрики ми сподіваємося підвищити точність локалізації несправностей у тестовому прикладі, що відтворює аварію.

– у реалізованому автоматизованому підході до локалізації несправностей при аварії ми використовували локалізацію несправностей на основі спектру для частини локалізації несправностей трубопроводу. Однак, як стверджують Вонг та ін. [2] існує багато різних підходів до автоматизованої локалізації несправностей. Було б корисно знати, який із цих підходів найкраще підходить для автоматичної локалізації помилок (наприклад, підходи на основі машинного навчання).

– Наразі звіт про локалізацію помилок експортується як файл CSV, що містить відсортований список дефектних операторів. Таким чином, було б корисно розширити підхід автоматичної локалізації несправностей при аварії, щоб у підсумку він візуалізував звіт, наприклад, за допомогою техніки візуалізації, запропонованої Джонс та ін. [85].

– Визначення найкращого методу для створення одиничних тестів. У нашій оцінці згенеровані тестові випадки також включають тестові випадки, які можуть бути невідповідними для основної помилки (наприклад, інші методи в класі, які знаходяться за межами трасування стека). Цей час обчислень можна краще витратити на створення тестових випадків, які стосуються помилки.

Для подальшого покращення оцінки конвеєра автоматичної локалізації аварії пропоную наступні моменти, які можуть бути цікавими для майбутньої роботи:

– Оцінка включала наскрізну оцінку підходу автоматичної локалізації несправностей при аварії. Отже, не оцінювались ефекти, викликані випадковістю BOTSING. Тому ми повинні досліджувати вплив випадковості, повторюючи крок відтворення збою та аналізуючи результат процесу локалізації несправностей.

– Розширення поточної оцінки за допомогою нових збоїв, наприклад, із збоями XWiki або Elasticsearch з набору даних JCRASHPACK [70]. На додаток до цього, DEFECTS4J [76] нещодавно було оновлено з додаванням 11 нових проектів. Це стосується не тільки каналів в цілому, але й окремо B OTSING, оскільки ще не зрозуміло, чи можна відтворити ці аварії.

– З метою впровадження автоматизованої локалізації збоїв у існуючих інфраструктурах тестування та налагодження було б дуже корисно знати статистику часу виконання конвеєра, коли він виконується в цілому.

– EVOCRASH [69], предок BOTSING, був оцінений за допомогою контрольованого експерименту для визначення корисності для налагодження. У нашій оцінці ми розглядаємо лише підхід до автоматичної локалізації несправностей аварії з діагностичною точністю, визначеною результатами EXAM. Проте також було б корисно оцінити ефективність автоматичної локалізації несправностей при аварії за допомогою контрольованого експерименту. Особливо у випадку, коли BOTSING генерує випадковий вхід, який корисний для локалізації помилок, але коли ми не впевнені, чи корисні вони для розробника.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Mariano Ceccato. An empirical study about the effectiveness of debugging when random test cases are used. / Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. // 2012 34th International Conference on Software Engineering (ICSE), pages 452-462. IEEE, 2012.

2. W Eric Wong. A survey on software fault localization. / Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa.// IEEE Transactions on Software Engineering 42(8):707- 740, 2016.

3.Software fail watch: The politics of software defects [Електронний ресурс] – 2019. – Режим доступу до ресурсу: https://www.tricentis.com/blog/software-fail-watch-q2-2018/.

4. Phil McMinn. Search-based software testing: Past, present and future. // IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pages 153-163. IEEE, 2011.

5. Gordon Fraser. Evosuite: automatic test suite generation for object- oriented software. / Andrea Arcuri. // 13th European conference on Foundations of software engineering, pages 416-419, 2011.

6. Gordon Fraser. Whole test suite generation./ Andrea Arcuri.// IEEE Transactions on Software Engineering, 39(2):276-291, 2012.

7. Sebastiano Panichella. The impact of test case summaries on bug fixing performance: An empirical investigation. / Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. // 38th International Conference on Software Engineering, pages 547-558, 2016.

8. Pouria Derakhshanfar. Search-based crash reproduction using behavioural model seeding. Software Testing, Verification and Reliability. / Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. // 30(3):e1733, 2020.

9. Mozhan Soltani. Search-Based Crash Reproduction and Its Impact on Debugging. / Annibale Panichella, and Arie Van Deursen. // IEEE Transactions on Software Engineering, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2018.2877664.

10. Rui Abreu. A practical evaluation of spectrum-based fault localization. / Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. // Journal of Systems and Software, 82 (11):1780-1792, 2009.

11. Mark Harman. Achievements, open problems and challenges for search based software testing. / Yue Jia, and Yuanyuan Zhang. // 8th International Conference on Software Testing, Verification and Validation (ICST), pages 1-12. IEEE, 2015.

12. Nadia Alshahwan. Deploying search based software engineering with sapienz at facebook. / Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. // In International Symposium on Search Based Software Engineering, pages 3-45. Springer, 2018.

13. Pouria Derakhshanfar. Search-based crash reproduction using behavioural model seeding. / Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. // Software Testing, Verification and Reliability, 30(3):e1733, 2020.

14. Jose Campos. Gzoltar: an eclipse plug-in for testing and debugging. / Andre Riboira, Alexandre Perez, and Rui Abreu. // In Proceedings of the 27 th IEEE/ACM International Conference on Automated Software Engineering, pages 378-381, 2012.

15. Spencer Pearson. Evaluating and improving fault localization. / Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. // In 2017IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 609-620. IEEE, 2017.

16. Rene Just. Defects: A database of existing faults to enable controlled testing studies for java programs. / Darioush Jalali, and Michael D Ernst. // International Symposium on Software Testing and Analysis, pages 437-440, 2014.

17. Mozhan Soltani. A benchmark-based evaluation of search-based crash reproduction. / Pouria Derakhshanfar, Xavier Devroey, and Arie Van Deursen. // Empirical Software Engineering, 25(1):96-138, 2020.

18. David S. Rosenblum. A practical approach to programming with assertions. // IEEE Transactions on software engineering, 21(1):19-31, 1995.

19. Deborah S Coutant. A practical approach to source-level debugging of globally optimized code. / Sue Meloy, and Michelle Ruscetta. // ACM SIGPLAN Notices, 23(7): 125-134, 1988.

20. Matthias Hauswirth Low-overhead memory leak detection using adaptive statistical profiling. / Trishul M Chilimbi.// ACM SIGPLAN notices, volume 39, pages 156¬164. ACM, 2004.

21. James S Collofello. Towards automatic software fault location through decision-to-decision path analysis. / Larry Cousins. // afips, page 539. IEEE, 1999

22. Rui Abreu. An evaluation of similarity coefficients for software fault localization. / Peter Zoeteweij, and Arjan JC Van Gemund.// 12th Pacific Rim International Symposium on Dependable Computing (PRDC’06), pages 39-46. IEEE, 2006.

23. Rui Abreu. A practical evaluation of spectrum-based fault localization. / Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. // Journal of Systems and Software, 82 (11):1780-1792, 2009.

24. Spencer Pearson. Evaluating and improving fault localization. / Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. // IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 609-620. IEEE, 2017.

25. W Eric Wong. The dstar method for effective software fault localization. / Vidroha Debroy, Ruizhi Gao, and Yihao Li. // IEEE Transactions on Reliability, 63(1):290- 308,2013.

26. Rui Abreu. Spectrum-based multiple fault localization. / Peter Zoeteweij, and Arjan JC Van Gemund. // IEEE/ACM International Conference on Automated Software Engineering, pages 88-99. IEEE, 2009.

27. James A Jones. Empirical evaluation of the tarantula automatic fault-localization technique. / Mary Jean Harrold. // 20th IEEE/ACM international Conference on Automated software engineering, pages 273-282, 2005.

28. Mark Weiser. Program slicing. // 5th international conference on Software engineering, pages 439-449. IEEE Press, 1981.

29. David W Binkley. A survey of empirical results on program slicing. / Mark Harman. // Advances in Computers, 62(105178):105-178, 2004.

30. Karl J Ottenstein. The program dependence graph in a software development environment. / Linda M Ottenstein. // ACM SIGSOFT Software Engineering Notes, 9(3): 177-184, 1984.

31. Akos Kiss. Using dynamic information in the interprocedural static slicing of binary executables. / Judit Jasz, Tibor Gyimothy. // Software Quality Journal, 13(3): 227-245, 2005.

32. Frank Tip. A slicing-based approach for locating type errors. / TB Dinesh. // ACM Transactions on Software Engineering and Methodology (TOSEM), 10(1):5-55,2001.

33. Bogdan Korel Stad-a system for testing and debugging: User perspective. / Janusz Laski. // Second Workshop on Software Testing, Verification, and Analysis, pages 13-20. IEEE, 1988.

34. Xiangyu Zhang. Locating faulty code by multiple points slicing. / Neelam Gupta, and Rajiv Gupta. // Software: Practice and Experience, 37(9):935-961, 2007.

35. Ju Qian. Baowen Xu. Scenario oriented program slicing. / Baowen Xu. // ACM symposium on Applied computing, pages 748-752, 2008.

36. Hiralal Agrawal. Fault localization using execution slices and dataflow tests. / Joseph R Horgan, Saul London, W Eric Wong. // Sixth International Symposium on Software Reliability Engineering. ISSRE’95, pages 143-151. IEEE, 1995.

37. W Eric Wong. An integrated solution for testing and analyzing java applications in an industrial setting. / Jenny Li. // 12th Asia-Pacific Software Engineering Conference (APSEC’05), pages 8-pp. IEEE, 2005.

38. Hira Agrawal. Mining system tests to aid software maintenance. / James L Alberi, Joseph R Horgan, J Jenny Li, Saul London, W Eric Wong, Sudipto Ghosh, and Norman Wilde. // Computer, 31(7):64-73, 1998.

39. W Eric Wong. Smart debugging software architectural design in sdl. / Tatiana Sugeta, Yu Qi, and Jose C Maldonado. // Journal of Systems and Software, 76(1):15-28, 2005.

40. David Abramson. Relative debugging and its application to the development of large numerical models. / Ian Foster, John Michalakes, and Rok Sosic. // ACM/IEEE Conference on Supercomputing, pages 51-51. IEEE, 1995.

41. David Abramson. Relative debugging in an integrated development environment. / Clement Chu, Donny Kurniawan, and Aaron Searle. // Software: Practice and Experience, 39(14):1157-1183, 2009.

42. Andreas Zeller. Ralf Hildebrandt. Simplifying and isolating failure-inducing input. / Ralf Hildebrandt. // IEEE Transactions on Software Engineering, 28(2):183-200, 2002.

43. Luciano C Ascari. Exploring machine learning techniques for fault localization. / Lucilia Y Araki, Aurora RT Pozo, and Silvia R Vergilio. // 10th Latin American Test Workshop, pages 1-6. IEEE, 2009.

44. W Eric Wong. Bp neural network-based effective fault localization. / Yu Qi. // International Journal of Software Engineering and Knowledge Engineering, 19(04):573-597, 2009.

45. W Eric Wong. Effective software fault localization using an rbf neural network. / Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. // IEEE Transactions on Reliability, 61(1):149-169, 2011.

46. Eric Wong. A crosstab-based statistical method for effective fault localization. / Tingting Wei, Yu Qi, and Lei Zhao. // 1st international conference on software testing, verification, and validation, pages 42-51. IEEE, 2008.

47. Chao Liu. Gplag: detection of software plagiarism by program dependence graph analysis. / Chen Chen, Jiawei Han, and Philip S Yu. // 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 872-881,2006.

48. Seokhyeon Moon. Ask the mutants: Mutating faulty programs for fault localization. / Yunho Kim, Moonzoo Kim, and Shin Yoo. // IEEE Seventh International Conference on Software Testing, Verification and Validation, pages 153-162. IEEE, 2014.

49. Phil McMinn. Search-based software testing: Past, present and future. // Fourth International Conference on Software Testing, Verification and Validation Workshops, pages 153-163. IEEE, 2011.

50. Gordon Fraser. Evosuite: automatic test suite generation for object-oriented software. / Andrea Arcuri. // 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 416-419, 2011.

51. Gordon Fraser. A large-scale evaluation of automated unit test generation using evosuite. / Andrea Arcuri. // ACM Transactions on Software Engineering and Methodology (TOSEM), 24(2):1-42, 2014.

52. M Moein Almasi. An industrial evaluation of unit test generation: Finding real faults in a financial application. / Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. // IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pages 263-272. IEEE, 2017.

53. Sebastiano Panichella. The impact of test case summaries on bug fixing performance: An empirical investigation. / Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. // 38th International Conference on Software Engineering, pages 547-558, 2016.

54. Mariano Ceccato. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. / Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. // ACM Transactions on Software Engineering and Methodology (TOSEM), 25(1):1-38, 2015.

55. Jose Campos. Entropy-based test generation for improved fault localization. / Rui Abreu, Gordon Fraser, and Marcelo d’Amorim. // 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 257-267. IEEE, 2013.

56. Alexandre Perez. A test-suite diagnosability metric for spectrum-based fault localization approaches. / Rui Abreu, and Arie van Deursen. // 39th International Conference on Software Engineering (ICSE), pages 654-664. IEEE, 2017.

57. Alexandre Perez. A theoretical and empirical analysis of program spectra diagnosability. / Rui Abreu, and Arie Van Deursen. // IEEE Transactions on Software Engineering, 2019. doi: 10.1109/tse.2019.2895640.

58. Wolfgang Mayer. Can ai help to improve debugging substantially? debugging experiences with value-based models. / Markus Stumptner, Dominik Wieland, and Franz Wotawa. // 15th European Conference on Artificial Intelligence, pages 417-421. IOS Press, 2002.

59. Yu Cao. Symcrash: selective recording for reproducing crashes. / Hongyu Zhang, Sun Ding. // 29th ACM/IEEE international conference on Automated software engineering, pages 791-802, 2014.

60. Francesco A Bianchi. Reproducing concurrency failures from crash stacks. / Mauro Pezze, Valerio Terragni. // In Proceedings of the 201711th Joint Meeting on Foundations of Software Engineering, pages 705-716, 2017.

61. Mathieu Nayrolles. A bug reproduction approach based on directed model checking and crash traces. / Abdelwahab Hamou-Lhadj, Sofiene Tahar, Alf Larsson. // Journal of Software: Evolution and Process, 29(3):e1789, 2017.

62. Shay Artzi. Recrash: Making software failures reproducible by preserving object states. / Sunghun Kim, Michael D Ernst. // European conference on object-oriented programming, pages 542-565. Springer, 2008.

63. Satish Narayanasamy. Bugnet: Continuously recording program execution for deterministic replay debugging. / Gilles Pokam, Brad Calder. // 32nd International Symposium on Computer Architecture (ISCA’05), pages 284-295. IEEE, 2005.

64. John Steven. Jrapture: A capture/re- play tool for observation-based testing. / Pravir Chandra, Bob Fleck, Andy Podgurski. // ACM SIGSOFT international symposium on Software testing and analysis, pages 158-167, 2000.

65. Maria Gomez. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. / Romain Rouvoy, Bram Adams, and Lionel Seinturier. // IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pages 88-99. IEEE, 2016.

66. Jeremias Robler. Reconstructing core dumps. / Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. // Sixth International Conference on Software Testing, Verification and Validation, pages 114-123. IEEE, 2013.

67. Ning Chen. Star: Stack trace based automatic crash reproduction via symbolic execution. / Sunghun Kim. // IEEE transactions on software engineering, 41(2):198-220, 2014.

68. Jifeng Xuan. Crash reproduction via test case mutation: let existing test cases help. / Xiaoyuan Xie, Martin Monperrus. // 10th Joint Meeting on Foundations of Software Engineering, pages 910-913, 2015.

69. Mozhan Soltani. Search-Based Crash Reproduction and Its Impact on Debugging. / Annibale Panichella, Arie Van Deursen. // IEEE Transactions on Software Engineering, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2018.2877664.

70. Mozhan Soltani. A benchmark-based evaluation of search-based crash reproduction. / Pouria Derakhshanfar, Xavier Devroey, Arie Van Deursen. // Empirical Software Engineering, 25(1):96-138, 2020.

71. Pouria Derakhshanfar. Crash Reproduction Using Helper Objectives. / Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. // Genetic and Evolutionary Computation Conference Companion (GECCO ’20 Companion), Cancun, Mexico, 2020. ACM. doi: 10.1145/3377929.3390077.

72. Jonathan Bell, Chronicler: Lightweight recording to reproduce field failures. / Nikhil Sarda, Gail Kaiser. // 35th International Conference on Software Engineering (ICSE), pages 362-371. IEEE, 2013.

73. W Eric Wong. Effective program debugging based on execution slices and inter-block data dependency. / Yu Qi. // Journal of Systems and Software, 79(7):891-903, 2006.

74. Gzoltar: an eclipse plug-in for testing and debugging. / Jose Campos, Andre Riboira, Alexandre Perez, Rui Abreu. // 27th IEEE/ACM International Conference on Automated Software Engineering, pages 378-381, 2012.

75. Fitsum Kifetew. Java unit testing tool competitionseventh round. / Xavier Devroey, Urko Rueda. // 12th International Workshop on Search-Based Software Testing (SBST), pages 15-20. IEEE, 2019.

76. Rene Just. Defects4j: A database of existing faults to enable controlled testing studies for java programs. / Darioush Jalali, Michael D Ernst. // International Symposium on Software Testing and Analysis, pages 437-440, 2014.

77. Sina Shamshiri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. / Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, Andrea Arcuri. // 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 201-211. IEEE, 2015.

78. Gordon Fraser Whole test suite generation. / Andrea Arcuri. // IEEE Transactions on Software Engineering, 39(2):276-291, 2012.

79. Andras Vargha. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. / Harold D Delaney. // Journal of Educational and Behavioral Statistics, 25(2):101-132, 2000.

80 Rui Abreu. On the accuracy of spectrumbased fault localization. / Peter Zoeteweij, Arjan JC Van Gemund. // In Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), pages 89-98. IEEE, 2007.

81. Jifeng Xuan. Test case purification for improving fault localization. / Martin Monperrus. // 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 52-63, 2014.

82. Rui Abreu. Spectrum-based multiple fault localization. / Peter Zoeteweij, Arjan JC Van Gemund. // IEEE/ACM International Conference on Automated Software Engineering, pages 88-99. IEEE, 2009.

83. Rongxin Wu. Crashlocator: locating crashing faults based on crash stacks. / Hongyu Zhang, Shing-Chi Cheung, Sunghun Kim. // International Symposium on Software Testing and Analysis, pages 204-214, 2014.

84. Yongfeng Gu. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. / Jifeng Xuan, Hongyu Zhang,

Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, Tieyun Qian. // Journal of Systems and Software, 148: 88-104, 2019.

85. James A Jones. Visualization of test information to assist fault localization. / Mary Jean Harrold, John Stasko. // 24th International Conference on Software Engineering. ICSE 2002, pages 467-477. IEEE, 2002.

86. Гурман І.В. Оціночні функції і метрики для виявлення помилок при тестуванні програмного забезпечення. / Джулій А.В., Чешун В.М., Чорненький В.І. // Вимірювальна та обчислювальна техніка в технологічних процесах. – 2021. – №2 (299). – С. 57-63.

ДОДАТОК А

Скрипти точок фходження процедур процесу тестування

Підключення бази даних реальних несправностей і експериментальної інфраструктура для забезпечення контрольованих експериментів у дослідженні.

```
DIR_SCRIPT=$(cd $(dirname ${BASH_SOURCE[0]}) && pwd)
DIR_APPLICATION=${DIR_SCRIPT}/application
DIR_RESULTS=${DIR_SCRIPT}/results

# ----------------------------------------------------------------------------
--
# Helper function
# ----------------------------------------------------------------------------
--
function header() {
echo ''
echo
"#############################################################################
#####"
echo -e "# #\r# $1"
echo
"#############################################################################
#####"
}

function message() {
echo -e
"...........................................................................
$1\r$2" >&2
}

function die() {
message "ERROR" "$@" >&2
exit 1
}

# ----------------------------------------------------------------------------
--
# Check usage
# ----------------------------------------------------------------------------
--
USAGE='Environment variables PROJECT_ID, BUG_ID are required!'
```

```
[[ "${PROJECT_ID}" != "" ]] || die "${USAGE}"
[[ "${BUG_ID}" != "" ]] || die "${USAGE}"

# ----------------------------------------------------------------------------
--
# Checkout & Compile the Defects4J project
# ----------------------------------------------------------------------------
--
function checkout_and_compile() {
header 'APPLICATION CHECKOUT'

message "INFO" "Checkout ${PROJECT_ID}-${BUG_ID}f"
defects4j checkout -p ${PROJECT_ID} -v ${BUG_ID}f -w ${DIR_APPLICATION}

message "INFO" "Compile project and test suite"
defects4j compile -w ${DIR_APPLICATION}
}

# ----------------------------------------------------------------------------
--
# List all the relevant classes.
# ----------------------------------------------------------------------------
--
function list_relevant_classes() {
header 'LIST RELEVANT CLASSES'

RELEVANT_CLASSES=()
for LINE in $(cat ${DIR_SCRIPT}/crashes/${PROJECT_ID}-${BUG_ID}b.log | tail -n
+2); do
if [[ "${LINE}" == "at" ]]; then
continue
fi

ELEMENT=$(echo ${LINE} | sed -e 's/.[^.]*(.*)$//')
message "ADDED" "${ELEMENT}"
RELEVANT_CLASSES+=("${ELEMENT}")
done

RELEVANT_CLASSES=$(echo ${RELEVANT_CLASSES[@]} | tr " " "\n" | sort | uniq)
}

# ----------------------------------------------------------------------------
--
# List all the relevant test classes
# ----------------------------------------------------------------------------
--
function list_relevant_test_classes() {
```

```
header 'LIST RELEVANT PASSING TEST CLASSES'

RELEVANT_TEST_CLASSES=()
defects4j export -p tests.all -w ${DIR_APPLICATION} >/tmp/all-test-cases.txt
2>/dev/null

echo -n 'Total number of test cases: '
cat /tmp/all-test-cases.txt | wc -l
echo ''

for LINE in $(cat /tmp/all-test-cases.txt); do
defects4j monitor.test -t ${LINE} -w ${DIR_APPLICATION} >/tmp/loaded-classes
2>/dev/null || continue
MESSAGE="SKIP"

for RELEVANT_CLASS in ${RELEVANT_CLASSES[@]}; do
if grep -q ${RELEVANT_CLASS} "/tmp/loaded-classes"; then
RELEVANT_TEST_CLASSES+=("${LINE}")
MESSAGE="ADDED"
break
fi
done

message ${MESSAGE} ${LINE}
done </tmp/all-test-cases.txt
}

# -----------------------------------------------------------------------------
--
# Check if not missing any test classes based on the Defects4J list
# -----------------------------------------------------------------------------
--
function check_missing_test_cases() {
header 'RESULTS'

message "INFO" "List relevant test classes"

printf "%s\n" "${RELEVANT_TEST_CLASSES[@]}" >/tmp/relevant-test-cases
defects4j export -p tests.relevant -w ${DIR_APPLICATION} -o /tmp/defects4j-
relevant-test-cases 2>/dev/null

if [[ $(grep -Fxv -f /tmp/relevant-test-cases /tmp/defects4j-relevant-test-
cases | wc -l) -gt 0 ]]; then
message 'ERROR' "Did not find all relevant test cases"

grep -Fxv -f /tmp/relevant-test-cases /tmp/defects4j-relevant-test-cases
>>/tmp/relevant-test-cases
message 'INFO' 'Adding missing relevant test cases'
```

```
fi

message "INFO" "Found $(grep -Fxv -f /tmp/defects4j-relevant-test-cases
/tmp/relevant-test-cases | wc -l) additional test cases"
}

# ----------------------------------------------------------------------------
--
# Copy the relevant test cases
# ----------------------------------------------------------------------------
--
function copy_relevant_test_cases() {
message "INFO" "Copy relevant test cases"
BUILD_DIR_TEST=$(defects4j export -p dir.src.tests -w ${DIR_APPLICATION}
2>/dev/null)

DIR_TEST_PASS=${DIR_RESULTS}/tests-pass
rm -rf ${DIR_TEST_PASS}/*

for CLASS in $(cat /tmp/relevant-test-cases); do
SOURCE_FILE_NAME=${DIR_APPLICATION}/${BUILD_DIR_TEST}/${CLASS//./\/}.java
TARGET_FILE_NAME=${DIR_TEST_PASS}/${CLASS//./\/}.java

mkdir -p $(dirname ${TARGET_FILE_NAME})
cp ${SOURCE_FILE_NAME} ${TARGET_FILE_NAME}
done
}

# ----------------------------------------------------------------------------
--
# Exclude the failing test cases
# ----------------------------------------------------------------------------
--
function exclude_test_cases() {
message "INFO" "Checkout ${PROJECT_ID}-${BUG_ID}b"
defects4j checkout -p ${PROJECT_ID} -v ${BUG_ID}b -w ${DIR_APPLICATION}
2>/dev/null

message "INFO" "Compile project and test suite"
defects4j compile -w ${DIR_APPLICATION} 2>/dev/null

message "INFO" "Compress test suite"
TAR_FILE_NAME=${PROJECT_ID}-${BUG_ID}b-relevant.1.tar.bz2
TAR_FILE=${DIR_SCRIPT}/${TAR_FILE_NAME}

cd ${DIR_TEST_PASS}
tar -cjf ${TAR_FILE} org
cd ${DIR_SCRIPT}
```

```
message "INFO" "List failing test cases"
printf "%s\n" $(defects4j test -s ${TAR_FILE} -w ${DIR_APPLICATION}
2>/dev/null) >/tmp/failing-tests

cat > ${DIR_TEST_PASS}/EXCLUDE_TEST_CASES
for LINE in $(tail -n +4 /tmp/failing-tests); do
if [[ "${LINE}" == "-" ]]; then
continue
fi

echo ${LINE} | sed -e 's/::/#/g' >>${DIR_TEST_PASS}/EXCLUDE_TEST_CASES
done

rm ${TAR_FILE}
}

# ----------------------------------------------------------------------------
--
# List the failing test cases
# ----------------------------------------------------------------------------
--
function list_failing_test_cases() {
header "LIST FAILING TEST CASE"

DIR_TEST_FAIL=${DIR_RESULTS}/tests-fail
mkdir -p ${DIR_TEST_FAIL}
rm -rf ${DIR_TEST_FAIL}/*

touch ${DIR_TEST_FAIL}/INCLUDE_TEST_CASES
echo $(defects4j export -p tests.trigger -w ${DIR_APPLICATION} 2>/dev/null) |
sed -e 's/::/#/g' >>${DIR_TEST_FAIL}/INCLUDE_TEST_CASES

for LINE in $(cat ${DIR_TEST_FAIL}/INCLUDE_TEST_CASES); do
CLASS_NAME=$(echo ${LINE} | sed -e 's/#.*//' | sed -e 's/\./\//g').java
SOURCE_FILE_NAME=${DIR_APPLICATION}/${BUILD_DIR_TEST}/${CLASS_NAME}
TARGET_FILE_NAME=${DIR_TEST_FAIL}/${CLASS_NAME}

mkdir -p $(dirname ${TARGET_FILE_NAME})
cp ${SOURCE_FILE_NAME} ${TARGET_FILE_NAME}

message "ADDED" ${LINE}
done
}

# ----------------------------------------------------------------------------
--
# Startup
```

```
# ---------------------------------------------------------------------------
--
function main() {
checkout_and_compile

list_relevant_classes
list_relevant_test_classes
check_missing_test_cases
copy_relevant_test_cases
exclude_test_cases

list_failing_test_cases
}

main
exit $?
```

Автоматична генерація тестових наборів JUnit для класів Java

```
DIR_SCRIPT=$(cd $(dirname ${BASH_SOURCE[0]}) && pwd)
DIR_APPLICATION=${DIR_SCRIPT}/application
DIR_RESULTS=${DIR_SCRIPT}/results

# -----------------------------------------------------------------------------
--
# Helper function
# -----------------------------------------------------------------------------
--
function header() {
echo ''
echo
"##############################################################################
#####"
echo -e "# #\r# $1"
echo
"##############################################################################
#####"
}

function message() {
echo -e
"............................................................................
$1\r$2" >&2
}

function die() {
message "ERROR" "$@" >&2
exit 1
}

# -----------------------------------------------------------------------------
--
# Libraries
# -----------------------------------------------------------------------------
--
EVO_SUITE_JAR=${DIR_SCRIPT}/libs/evosuite-1.0.6.jar

[[ -s "$EVO_SUITE_JAR" ]] || die "$EVO_SUITE_JAR does not exist or it is
empty!"

# -----------------------------------------------------------------------------
--
```

```
# Check usage
# ----------------------------------------------------------------------------
--
USAGE='Environment variables PROJECT_ID, BUG_ID are required!'

[[ "${PROJECT_ID}" != "" ]] || die "${USAGE}"
[[ "${BUG_ID}" != "" ]] || die "${USAGE}"

# ----------------------------------------------------------------------------
--
# Checkout & Compile the Defects4J project
# ----------------------------------------------------------------------------
--
function checkout_and_compile() {
header 'APPLICATION CHECKOUT'

message "INFO" "Checkout ${PROJECT_ID}-${BUG_ID}b"
defects4j checkout -p ${PROJECT_ID} -v ${BUG_ID}b -w ${DIR_APPLICATION}

message "INFO" "Compile project and test suite"
defects4j compile -w ${DIR_APPLICATION}
}

# ----------------------------------------------------------------------------
--
# List all the relevant classes based on the stack trace of the crash
# ----------------------------------------------------------------------------
--
function list_relevant_classes() {
header 'RELEVANT CLASSES'

RELEVANT_CLASSES=()
for LINE in $(cat ${DIR_SCRIPT}/crashes/${PROJECT_ID}-${BUG_ID}b.log | tail -n
+2); do
if [[ "${LINE}" == "at" ]]; then
continue
fi

ELEMENT=$(echo ${LINE} | sed -e 's/.[^.]*(.*)$//')
message "ADDED" "${ELEMENT}"
RELEVANT_CLASSES+=("${ELEMENT}")
done

UNIQUE_RELEVANT_CLASSES=$(echo ${RELEVANT_CLASSES[@]} | tr " " "\n" | sort |
uniq)
}
```

```
# ---------------------------------------------------------------------------
--
# Run EvoSuite
# ---------------------------------------------------------------------------
--
function run_evosuite() {
header 'EVOSUITE'

BUILD_DIR_TEST=${DIR_APPLICATION}/$(defects4j export -p dir.bin.classes -w
${DIR_APPLICATION} 2>/dev/null)
EVO_SUITE="java -jar ${EVO_SUITE_JAR}"

for RELEVANT_CLASS in ${UNIQUE_RELEVANT_CLASSES}; do
${EVO_SUITE} -class ${RELEVANT_CLASS} -projectCP ${BUILD_DIR_TEST} -
Dtest_dir=${DIR_RESULTS}
done
}

# ---------------------------------------------------------------------------
--
# Startup
# ---------------------------------------------------------------------------
--
function main() {
checkout_and_compile
list_relevant_classes
run_evosuite
}

main
exit $?
```

Використання фреймворку Botsing для відтворення збою

```
DIR_SCRIPT=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)
DIR_APPLICATION=${DIR_SCRIPT}/application
DIR_RESULTS=${DIR_SCRIPT}/results

# ------------------------------------------------------------------------------
--
# Helper function
# ------------------------------------------------------------------------------
--
function header() {
echo ''
echo
"###############################################################################
#####"
echo -e "# #\r# $1"
echo
"###############################################################################
#####"
}

function message() {
echo -e
".............................................................................
$1\r$2" >&2
}

function die() {
message "ERROR" "$@" >&2
exit 1
}

# ------------------------------------------------------------------------------
--
# Libraries
# ------------------------------------------------------------------------------
--
BOTSING_REPRODUCTION_JAR=${DIR_SCRIPT}/botsing/botsing-reproduction-1.0.8-
SNAPSHOT.jar
EVOSUITE_JAR=${DIR_SCRIPT}/botsing/evosuite.jar
HAMCREST_JAR=${DIR_SCRIPT}/botsing/hamcrest-core-1.1.jar
JUNIT_JAR=${DIR_SCRIPT}/botsing/junit.jar

[[ -s ${BOTSING_REPRODUCTION_JAR} ]] || die "$BOTSING_REPRODUCTION_JAR does
not exist or it is empty!"
[[ -s ${EVOSUITE_JAR} ]] || die "$EVOSUITE_JAR does not exist or it is empty!"
```

```
[[ -s ${HAMCREST_JAR} ]] || die "$HAMCREST_JAR does not exist or it is empty!"
[[ -s ${JUNIT_JAR} ]] || die "$JUNIT_JAR does not exist or it is empty!"

# ---------------------------------------------------------------------------
--
# Libraries
# ---------------------------------------------------------------------------
--
USAGE='Environment variables PROJECT_ID, BUG_ID, TARGET_FRAME, MAX_RETRIES and
MULTIPLE are required!'

[[ "${PROJECT_ID}" != "" ]] || die "${USAGE}"
[[ "${BUG_ID}" != "" ]] || die "${USAGE}"
[[ "${TARGET_FRAME}" != "" ]] || die "${USAGE}"
[[ "${MAX_RETRIES}" != "" ]] || die "${USAGE}"
[[ "${MULTIPLE}" != "" ]] || die "${USAGE}"
[[ "${MULTIPLE}" == "true" ]] && MULTIPLE=1 || MULTIPLE=0

# ---------------------------------------------------------------------------
--
# Checkout & Compile the Defects4J project
# ---------------------------------------------------------------------------
--
function checkout_and_compile() {
header 'APPLICATION CHECKOUT'

message "INFO" "Checkout ${PROJECT_ID}-${BUG_ID}b"
defects4j checkout -p ${PROJECT_ID} -v ${BUG_ID}b -w ${DIR_APPLICATION}

message "INFO" "Compile project and test suite"
defects4j compile -w ${DIR_APPLICATION} 2>/dev/null
}

# ---------------------------------------------------------------------------
--
# Run botsing
# ---------------------------------------------------------------------------
--
function generate_crash_reproducing_test_case() {
header 'RUN BOTSING'

CRASH_LOG=${DIR_SCRIPT}/crashes/${PROJECT_ID}-${BUG_ID}b.log
DIR_BUILD_SOURCE=${DIR_APPLICATION}/$(defects4j export -p dir.bin.classes -w
${DIR_APPLICATION} 2>/dev/null)
DIR_BUILD_TEST=${DIR_APPLICATION}/$(defects4j export -p dir.bin.tests -w
${DIR_APPLICATION} 2>/dev/null)

mkdir -p ${DIR_RESULTS}
```

```
rm -rf ${DIR_RESULTS:?}/*

REPRODUCTION_LOGS=${DIR_RESULTS}/logs/botsing.reproduction.log
mkdir -p ${DIR_RESULTS}/logs

run_until_success_or_max_reached
message 'DONE' 'Done'
}
# ---------------------------------------------------------------------------
--
# Run Botsing once.
# ---------------------------------------------------------------------------
--
function execute_botsing() {
BUILD_JARS=$(defects4j export -p cp.test -w ${DIR_APPLICATION} 2>/dev/null)

java -jar ${BOTSING_REPRODUCTION_JAR} \
-project_cp ${DIR_BUILD_SOURCE}:${DIR_BUILD_TEST}:${BUILD_JARS} \
-search_algorithm SPEA2 \
-fitness WeightedSum:TestLen:CallDiversity \
$([[ ${MULTIPLE} == '1' ]] && echo '-continue_after_reproduction' || echo '')
\
-crash_log ${CRASH_LOG} \
-Dtest_dir=${DIR_RESULTS} \
-Dno_runtime_dependency=false \
-target_frame ${TARGET_FRAME} \
-Dsearch_budget=180 \
-Dpopulation=100 \
-Dcatch_undeclared_exceptions=false >${REPRODUCTION_LOGS}
}
# ---------------------------------------------------------------------------
--
# Check if a run is successful
# ---------------------------------------------------------------------------
--
function check_if_run_successfull() {
for ((TRY = 0; TRY < ${MAX_RETRIES}; ++TRY)); do
execute_botsing

TEST_CASE_NAME=$(find ${DIR_RESULTS} -name '*_ESTest.java' | head -n 1)
if grep -Fq 'notGeneratedAnyTest' ${TEST_CASE_NAME}; then
message 'ERROR' 'Generated an empty test case. Re-try'
else
if [[ ${MULTIPLE} == 1 ]]; then
if grep -Fq 'Detect a new crash reproducing test case' ${REPRODUCTION_LOGS};
then
message 'SUCCESS' 'Found the optimal solution'
SUCCESSFUL=true
```

```
break
else
message 'ERROR' 'Failed to find a solution. Re-try'
fi
else
if grep -Fq 'target crash is covered' ${REPRODUCTION_LOGS}; then
message 'SUCCESS' 'Found the optimal solution'
SUCCESSFUL=true
break
else
message 'ERROR' 'Failed to find a solution. Re-try'
fi
fi
fi
done
}
# ----------------------------------------------------------------------------
--
# Run Botsing until it succesfully generated a test case or when
# it reached the maximum number of retries
# ----------------------------------------------------------------------------
--
function run_until_success_or_max_reached() {
message 'INFO' 'Running Botsing on the problem'
SUCCESSFUL=false
check_if_run_successfull

if [[ "${SUCCESSFUL}" == false ]]; then
cd ${DIR_RESULTS}
find * ! -name 'logs' -type d -exec rm -rf {} +
cd ${DIR_SCRIPT}
die "Failed to generate a test case"
fi

tail -n 10 ${REPRODUCTION_LOGS}
}
# ----------------------------------------------------------------------------
--
# Startup
# ----------------------------------------------------------------------------
--
function main() {
checkout_and_compile
generate_crash_reproducing_test_case
}

main
exit $?
```

ДОДАТОК Б

Копія наукової публікації

УДК 004.415.53
DOI

ГУРМАН І. В., ЧЕШУН В. М.
Хмельницький національний університет
ДЖУЛІЙ А. В., ЧОРНЕНЬКИЙ В. І.
Університет економіки і підприємництва, м.Хмельницький

ОЦІНОЧНІ ФУНКЦІЇ І МЕТРИКИ ДЛЯ ВИЯВЛЕННЯ ПОМИЛОК ПРИ ТЕСТУВАННІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Значні напрацювання діагності і науковців та створення великої кількості методів тестування програмного забезпечення на сьогодні не вирішили проблему повної локалізації дефектів програмного коду і не зменшили актуальність цієї задачі.

Дослідження присвячене аналізу можливості підвищення ефективності методів автоматизованого тестування програмного забезпечення із застосуванням різних варіантів оціночних функцій і метрик, які широко використовуються для оптимізації тестів і оцінки якості результатів тестування. В роботі розглянуто базові класи методів тестування програмного забезпечення, напрямки і технології автоматизації тестування, проведено аналіз зв'язку функцій придатності, коефіцієнтів подібності та метрик із результатами тестування. Розглянуті коефіцієнти подібності Кульчинського, D^2*, Tarantula, Rogers&Tanimodo, Ochiai, Barinel, а також метрики Expense, Exam-Score, F3 (Jin і Orso), Laghari, T-Score, Mean Average Precision, Ulysis, G тощо. Від якості обраних функцій і їх відповідності методу тестування значною мірою залежить і результат локалізації дефектів програмного коду, що є передумовою зменшення ризику збоїв в роботі програмного забезпечення, фінансових та моральних збитків розробників та користувачів програмних продуктів.*

Хоча більшість функцій та метрик орієнтовано на статистичні методи автоматизованого тестування програмного забезпечення на основі спектру, вони можуть бути використані або адаптовані до застосування і в інших методах.

Ключові слова: тестування програмного забезпечення, функція придатності, коефіцієнт *подібності, метрика.*

GURMAN I., CHESHUN V.
Khmelnytsky National University, Khmelnytsky, Ukraine
DZHULIY A., CHORNENKYI V.
University of Economics and Entrepreneurship, Khmelnytsky, Ukraine

EVALUATION FUNCTIONS AND METRICS FOR ERROR DETECTION WHEN TESTING SOFTWARE

Significant achievements of diagnostics and scientists and the creation of a large number of software testing methods to date have not solved the problem of complete localization of software code defects and have not reduced the relevance of this task.

The study is devoted to the analysis of the possibility of improving the efficiency of automated software testing methods using different options of evaluation functions and metrics, which are widely used to optimize tests and assess the quality of test results. The paper considers the basic classes of software testing methods, directions and technologies of test automation, analyzes the relationship of suitability functions, similarity coefficients and metrics with test results. The similarity coefficients of Kulchinsky, D2, Tarantula, Rogers & Tanimodo, Ochiai, Barinel, as well as metrics Expense, Exam-Score, F3 (Jin and Orso), Laghari, T-Score, Mean Average Precision, Ulysis, G, etc. are considered. Features of the selected functions and their compliance with the testing method largely determines the result of localization of software code defects, which is a prerequisite for reducing the risk of software failures, financial and moral losses of software developers and users.

Although most functions and metrics focus on statistical methods of automated spectrum-based software testing, they can be used or adapted for use in other methods.

Key words: software testing, suitability function, similarity coefficient, metrics.

Вступ. Постановка проблеми. Стрімка інформатизація суспільства відбувається у безпосередньому зв’язку з розвитком мов і технологій створення програмних продуктів. IT-галузь сьогодні виходить на лідерські позиції в економіках багатьох країн і потребує все більше фахівців, а жорстка конкуренція призводить до інтенсифікації їх роботи. Незважаючи на постійне вдосконалення інструментів і технологій створення програмних продуктів, відмовитись від участі людини в його розробці на сьогодні неможливо, що зумовлює вплив людського фактору на отримуваний результат і неминуче виникнення помилок в програмних продуктах.

Дослідження, проведене ще в 2002 році, показало, що лише для економіки США дефекти програмного забезпечення становили щорічні витрати в 60 мільярдів доларів, і з тих пір спостерігається тенденція до зростання ціни помилок програмування [1]. Подекуди дефекти програмного забезпечення мають катастрофічні наслідки, чому прикладом може слугувати вибух в червні 1982 року силою в три кілотонни на газопроводі в Сибіру, втрата Європейським космічним агентством в червні 1996 року ракети-носія Ariane 5, яка була розроблена «Чорний понеділок» фондової біржі Wall Street 19 жовтня 1987 року тощо [2].

Проблеми наявності помилок програмного коду, поряд із високими вимогами щодо функціональності і надійності програмних продуктів, загострюють питання забезпечення їх якості. Одним із основних інструментів для цього стає тестування, що актуалізує завдання розробки нових ефективних методів тестування програмних продуктів та вдосконалення існуючих.

Аналіз останніх досліджень і публікацій. Тестування програмного забезпечення проходить протягом усього циклу розробки програмного забезпечення і, традиційно, потребує понад 50% зусиль, що витрачаються на розробку та обслуговування програмного продукту [4, 5]. Локалізація дефектів у програмному забезпеченні спрямована на виявлення помилок та несправних функцій у програмному продукті та зменшення ризиків збоїв у його роботі і є одним із найбільш складних, трудомістких, виснажливих та витратних заходів у налагодженні програм [4, 6]. Це зумовило появу великої кількості методів локалізації, також ідентифікованих як методи тестуванні програмного забезпечення.

Базовими вважаються такі методи локалізації дефектів, як журналювання програми або введення операторів друку, застосування тверджень, введення точок зупину та профілювання [1, 3], до яких в [5] додається метод покриття коду. Ці методи, зазвичай, класифікуються «ручними» і спираються на досвід розробника або експерта. Ефективність «ручного» керування тестуванням програмного забезпечення в сучасних умовах визнана малоефективною. Це пов’язано з тим, що виявлення, локалізації та усунення дефектів програмного коду є процесом нетривіальним і схильним до помилок прийняття рішень, в якому навіть досвідчені розробники помиляються майже в 90% випадків у своїх початкових припущеннях, намагаючись визначити причину неочікуваного спрацьовування програми, яке відхиляється від планованого [1].

Автоматизація тестування програмного забезпечення відбувається за різними напрямками. В [7] пропонується перелік основних напрямків автоматизації: автоматизація управління тестуванням; автоматизоване тестування; автоматизація кросбраузерного тестування; автоматизація навантажувального тестування; автоматизація відслідковування помилок; автоматизація тестування API. Серед основних технологій автоматизованого тестування в [8] вказуються приватні рішення; тестування під керуванням даними; тестування під керуванням ключовими словами; використання фре ймворків; запис та відтворення; поведінкове тестування. Скласти повний перелік методів автоматизованого тестування програмного забезпечення, а, тим більш, засобів реалізації цих методів неможливо. В [9] наведено базову класифікацію методів Вонга, яка ділить їх на методи локалізації дефектів на основі спектру, на основі фрагментування програм, на основі стану та на основі машинного навчання.

Хоча наявні різні напрямки автоматизації тестування програмного забезпечення та існує значне різноманіття самих методів тестування, завдання ефективної локалізації помилок не є вирішеним. Свідченням цього є помилки в кодах і збої в роботі програмного забезпечення провідних розробників програмного забезпечення, які при розробці програмного забезпечення велику кількість ресурсів витрачають на налагодження. Всім знайомі повідомлення Microsoft про збої в роботі програмного забезпечення з пропозицією відправити звіт розробнику. За статистикою syzbot за період з листопаду 2017 року по квітень 2021 року було виявлено 979 помилок ядра Linux. Суттєва кількість помилок і збоїв в роботі програмного забезпечення фіксується незважаючи на те, що діяльність з тестування та налагодження може легко коливатися від 50 до 75 відсотків загальної вартості розробки [1].

Здебільшого тестувальники-практики використовують типові функції і метрики, передбачені для реалізації певного методу пошуку і локалізації дефектів. В наукових робот спостерігається тенденція до систематизації і аналізу властивостей оціночних функцій та метрик, а також до дослідження ефективності методів тестування при застосуванні різних критеріїв та способів оцінювання ефективності елементів та результатів діагностичних експериментів [5, 9, 12]. Цьому питанню приділено значно менше уваги і наукових робіт, ніж створенню самих методів тестування.

Метою роботи є визначення та аналізі типових оціночних функцій і метрик, за рахунок яких при тестуванні програмного забезпечення виконуються оцінка якості і оптимізація тестів, а також оцінка якості результатів тестування.

Виклад основного матеріалу. Застосування ефективних методів оцінювання параметрів і результатів діагностичних експериментів є основою для ефективного управління тестовими випробуваннями та їх оптимізації, а також є передумовою для максимального покриття наявних дефектів і їх усунення.

На сьогодні найбільшого поширення набули методи автоматизованого тестування на основі спектру, спрямовані на ефективну локалізацію несправних компонентів за допомогою дослідження поведінки програми [6].

Це робиться шляхом збирання шаблонів виконання різних комбінацій компонентів і відповідних результатів у спектр. Методи локалізації на основі спектру виявилися надзвичайно корисними для локалізації несправностей у великих кодових базах [11]. Спектральні методи спрямовані на виявлення дефектів (хибних командних рядків, функцій або фрагментів програми) на основі статистичного аналізу тестових спектрів. Спектри містять інформацію патерну активності кожного тесту та дають змогу виділити провальні тести як результат. При локалізації помилок коду на основі спектру відбувається ідентифікація оператора (твердження) із шаблоном виконання, максимально наближеним до шаблону збоїв усіх тестових випадків [9].

Ефективна локалізація дефектів програмного забезпечення в таких методах сильно залежить від якості спектрів [6, 11, 16], яка, в свою чергу залежить від математичного підґрунтя процедур синтезу тестів. Набори тестів можуть створюватися вручну або за допомогою методів автоматичного генерування тестів [Campos et al., 2014]. Генератор набору тестів може створити набір тестів *T*, оптимізуючи функцію придатності *f*, яка приймається як міра якості набору тестів [6]. Оператор визнається більш підозрілим, якщо шаблон його виконання подібний до шаблонів збою всіх невдалих тестових випадків. Оператор визнається менш підозрілим, якщо шаблон його виконання оператора подібний до шаблону виконання успішних тестових випадків. Рівень підозрілості оператора характеризується коефіцієнтом подібності, який обчислюється за наявними статистичними даними результатів тестування. На сьогодні розроблена велика кількість оціночних коефіцієнтів подібності, жоден з яких не претендує на повноту покриття і універсальність. Зокрема, в [5, 12] приводиться математичний опис більше 30 таких коефіцієнтів, що далеко не є повним переліком.

Розглянемо деякі типові приклади оціночних коефіцієнтів подібності.

Для представлення розрахункових формул обчислення коефіцієнтів подібності скористаємося описами типових розрахункових значень, запропонованими в [12] : N – загальна кількість успішно пройдених тестів; N_S – загальна кількість успішно пройдених тестів; N_F – загальна кількість невдалих тестів; N_C – загальна кількість тестів, що охоплюють підозріле твердження; N_U – загальна кількість тестів, що не охоплюють підозріле твердження; N_{CS} – кількість успішно пройдених тестів, що охоплюють підозріле твердження; N_{US} – кількість успішно пройдених тестів, що не охоплюють підозріле твердження; N_{CF} – кількість невдалих тестів, що охоплюють підозріле твердження; N_{UF} – кількість невдалих тестів, що не охоплюють підозріле твердження.

Стосовно наведених значень можна вказати їх взаємозв’язок:

$$N_S = N_{CS} + N_{US}; \tag{1}$$

$$N_F = N_{CF} + N_{UF}; \tag{2}$$

$$N_U = N_{US} + N_{UF}; \tag{3}$$

$$N_C = N_{CS} + N_{CF}; \tag{4}$$

$$N = N_S + N_F = N_C + N_U. \tag{5}$$

З урахуванням формул 1-5 значення загальних кількостей N, N_S, N_F, N_C і N_U можуть бути замінені в розрахункових формулах коефіцієнтів на суми їх складових.

Коефіцієнт Кульчинського [5, 12]:

$$K_K = \frac{N_{CF}}{N_{CS} + N_{UF}}. \tag{7}$$

Коефіцієнт D^2 [5]:

$$K_{D^2} = \frac{N_{CF}^2}{N_{CS} + N_{UF}}. \tag{8}$$

Коефіцієнт Tarantula [11]:

$$K_T = \frac{\dfrac{N_{CF}}{N_{CF} + N_{UF}}}{\dfrac{N_{CF}}{N_{CF} + N_{UF}} + \dfrac{N_{CS}}{N_{CS} + N_{US}}} = \frac{\dfrac{N_{CF}}{N_F}}{\dfrac{N_{CF}}{N_F} + \dfrac{N_{CS}}{N_S}}. \tag{6}$$

Коефіцієнт Rogers&Tanimodo [12]:

$$K_{R\&T} = \frac{N_{CF} + N_{US}}{N_{CF} + N_{US} + N_{CS} + N_{UF}} = \frac{N_{CF} + N_{US}}{N}. \tag{9}$$

Коефіцієнт Ochiai [11]:

$$K_O = \frac{N_{CF}}{\sqrt{\left(N_{CF}+N_{UF}\right)\times\left(N_{CF}+N_{CS}\right)}} = \frac{N_{CF}}{\sqrt{N_F \times \left(N_{CF}+N_{CS}\right)}}. \tag{10}$$

Коефіцієнт Barinel [11]:

$$K_B = 1-\frac{N_{CS}}{N_{CS}+N_{CF}} = 1-\frac{N_{CS}}{N_C}. \tag{11}$$

Незважаючи на різноманіття функцій коефіцієнтів, з результатів досліджень найбільш розповсюджених коефіцієнтів [11] слідує, що жоден із коефіцієнтів подібності не забезпечує статистично значущої різниці порівняно з іншими при використанні для локалізації несправностей на основі спектру, тому їх вибір є прерогативою тестувальника або розробника засобів реалізації методу тестування тощо.

Оцінки підозрілості тверджень в програмному коді широко застосовуються для оцінки ефективності тестування. Існують різні метрики оцінки якісних характеристик тестового процесу.

Метрика Expense [13] реалізується за ранжуванням тверджень відповідно до їх рівня підозрілості використовується для оцінки відсотку програми, що не потребує перевірки для виявлення помилки:

$$M_{\text{Expense}} = \frac{K-k}{K}, \tag{12}$$

де k - це ранг твердження з помилкою у звіті про локалізацію несправності, а K - загальна кількість тверджень у програмі.

Метрика Exam-Score [5, 11, 13] є однією із найпопулярніших і обчислює відсоток коду, який необхідно перевірити. Метрика Exam-Score базується на використанні тих самих параметрів ранжування тверджень, що і метрика Expense:

$$M_{\text{Exam-Score}} = \frac{k}{K}. \tag{13}$$

Окремим класом метрик оцінки якості тестових випробувань є метрики затрати зусиль на тестування, прикладами яких є описані в [14] метрика F3 (Jin і Orso) та її модифікація від Laghari.

У метриці F3 витрачені зусилля оцінюють, вказуючи кількість вірних тверджень, які в середньому мають бути перевірені, перш ніж буде виявлено помилкове твердження:

$$M_{\text{F3}} = m+l+1, \tag{14}$$

де m – кількість тверджень без помилки, яким присвоєно вищий бал підозрілості, ніж твердженню з помилкою; l – – кількість тверджень без помилки, яким присвоєно такий самий бал підозрілості, як і твердженню з помилкою. У модифікованій метриці F3 від Laghari при оцінці ранжується вага помилок на підставі урахування їх «грубості»:

$$M_{\text{Laghari}} = m+\frac{l+1}{2}. \tag{15}$$

Метрика T-Score [5, 11] орієнтована на використання нестатистичних методів локалізації помилок, які передбачають невеликий набір підозрілих тверджень у програмі. Метрика оцінює відсоток коду, який розробник не повинен перевіряти, перш ніж виявляти помилкове твердження. Метрика використовує граф-модель (PDG) залежності програми для обчислення набору вершин у графі, які необхідно перевірити, щоб досягти помилкового твердження. Найменший набір вершин, що містить помилкове твердження, називається сферою залежності (DS):

$$M_{\text{T-Score}} = \frac{|DS|}{|PDG|}, \tag{16}$$

де $|PDG|$ - загальна кількість вершин в граф-моделі діагностичного процесу; $|DS|$ - кількість вершин в граф-моделі, що входять в сферою залежності.

Метрика усередненого значення точності (Mean Average Precision – MAP) [14] запозичена для пошуку помилок в програмному коді із методів пошуку інформації. MAP визначається шляхом обчислення середнього значення точності:

$$M_{\mathrm{MAP}} = \frac{1}{M}\sum_{i=1}^{K} P(i)\,rel(i), \tag{17}$$

де *M* - кількість тверджень з помилкою у звіті про локалізацію несправності; *K* - загальна кількість тверджень у програмі; *P*(*i*) – оцінка помилки *i*-го твердження (точність *i*-го твердження); *rel*(*i*) – бінарний індикатор, що вказує, містить *i*-те твердження помилку чи ні.

Подібна метрика обґрунтовується в [15] при аналізі підходу до тестування на критеріях щільності-різноманіття-унікальності (DDU), за якими створюються «хороші» набори тестів шляхом покращення певних структурних властивостей спектрів. Метрика описується під назвою Ulysis:

$$M_{\mathrm{Ulysis}} = \sum_{i=1}^{K} P(i)\,W(i), \tag{18}$$

де *W*(*i*) – бінарний індикатор, який оцінює підозрілість *i*-го твердження на помилку (*W*(*i*)=0 – якщо найвищий ранг підозрілості, характерний для *i*-го твердження; *W*(*i*)=1 – якщо найвищий ранг підозрілості, нехарактерний для *i*-го твердження).

В [10] для реалізації тестування за методом DDU пропонується використати альтернативну метрику, що базується на використанні індексу Джіні-Сімпсона для вимірювання різноманітності (G). Метрика G обчислює ймовірність того, що два випадково вибраних елементи тесту будуть різних видів:

$$M_{\mathrm{G}} = 1 - \frac{\sum_{i=1}^{K} k \times (k-1)}{K \times (K-1)}, \tag{19}$$

де *k* – кількість тестів, які мають однакову активність (здатність виявляти дефекти).

Висновки

Розглянуті оціночні функції і метрики є типовими прикладами математичного апарату, застосовуваного для оптимізації процесу тестування програмного забезпечення, підвищення його ефективності та оцінювання якості отриманих результатів. Від якості обраних функцій і їх відповідності методу тестування значною мірою залежить і результат локалізації дефектів програмного коду, що є передумовою зменшення ризику збоїв в роботі програмного забезпечення, фінансових та моральних збитків розробників та користувачів програмних продуктів. Хоча більшість розглянутих функцій та метрик орієнтовано на статистичні методи автоматизованого тестування програмного забезпечення на основі спектру, але можуть бути використані або адаптовані до застосування і в інших методах.

Література

1. Alexandre Perez, Rui Abreu, Eric Wong A survey on fault localization techniques. Publication date: 2014. [Електронний ресурс]. – Режим доступу : https://inlnk.ru/megw7

2. Ціна помилки: найдорожчі комп'ютерні баги в історії IT [Електронний ресурс]. – Режим доступу : https://www.vectornews.net/news/society/66325-cna-pomilki-naydorozhch-kompyutern-bagi-v-storyi-t.html

3. W. Eric Wong, Vidroha Debroy. A Survey of Software Fault Localization: Technical Report UTDCS-45-09. Department of Computer Science The University of Texas at Dallas, November 2009. 19 p.

4. Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, Sunghun Kim. CrashLocator: locating crashing faults based on crash stacks. ISSTA 2014: Proceedings of the 2014 International Symposium on Software Testing and AnalysisJuly 2014. P. 204–214. DOI: https://doi.org/10.1145/2610384.2610386

5. Xiao Hong Su, Dan Dan Gong, Tian Tian Wang, Pei Jun Ma. A Survey of Automated Software Fault Localization Approach. Applied Mechanics and Materials, May 2014, Volumes 556-562. P. 6102-6105.

6. Prantik Chatterjee, Abhijit Chatterjee, Jose Campos, Rui Abreu. Diagnosing Software Faults Using Multiverse Analysis. Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20), 2020. P. 1629-1635.

7. Єгорова О.В. Програмні засоби для тестування програмного забезпечення / О.В. Єгорова, В.П. Бичок // Молодий вчений, Технічні науки. – 2019. – № 11 (75). – С. 680-684.

8. Аналіз інструментів для автоматизованого тестування програмного забезпечення / Ольга Мелкозерова, Алексей Нарежний, Сергей Малахов. // Комп'ютерні науки та кібербезпека. – 2019. – № 1. – С.75-84. DOI: https://doi.org/10.26565/2519-2310-2019-1-07

9. W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. IEEE Transactions on Software Engineering, 42(8), 2016. P.707–740.

10. Alexandre Perez, Rui Abreu, Arie van Deursen A test-suite diagnosability metric for spectrum-based fault localization approaches. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). P. 654-664.

11. Spencer Pearson, Jose´ Campos, Rene´ Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In 2017 IEEE/ACM 39th International Conference on

Software Engineering (ICSE), IEEE, 2017. P. 609–620. DOI: https://doi.org/10.1109/ICSE.2017.62

12. W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. IEEE Transactions on Reliability, 63(1):, IEEE, 2013. P. 290–308.
13. James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM international Con- ference on Automated software engineering, 2005. P. 273–282.
14. Aaron Ang, Alexandre Perez, Arie van Deursen, Rui Abreu. Revisiting the practical use of automated software fault localization techniques. In 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2017. P. 175–182.
15. Diego Calvanese, Julien Corman, Davide Lanti, and Simon Razniewski Counting Query Answers over a DL-Lite Knowledge Base International Joint Conference on Artificial Intelligence, 2020. P. 1629–1635.

References

1. Alexandre Perez, Rui Abreu, Eric Wong A survey on fault localization techniques. Publication date: 2014. [Elektronnyi resurs]. – Rezhym dostupu: https://inlnk.ru/megw7
2. Tsina pomylky: naidorozhchi kompiuterni bahy v istorii IT [Elektronnyi resurs]. – Rezhym dostupu: https://www.vectornews.net/news/society/66325-cna-pomilki-naydorozhch-kompyutern-bagi-v-storyi-t.html
3. W. Eric Wong, Vidroha Debroy. A Survey of Software Fault Localization: Technical Report UTDCS-45-09. Department of Computer Science The University of Texas at Dallas, November 2009. 19 p.
4. Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, Sunghun Kim. CrashLocator: locating crashing faults based on crash stacks. ISSTA 2014: Proceedings of the 2014 International Symposium on Software Testing and AnalysisJuly 2014. P. 204–214. DOI: https://doi.org/10.1145/2610384.2610386
5. Xiao Hong Su, Dan Dan Gong, Tian Tian Wang, Pei Jun Ma. A Survey of Automated Software Fault Localization Approach. Applied Mechanics and Materials, May 2014, Volumes 556-562. P. 6102-6105.
6. Prantik Chatterjee, Abhijit Chatterjee, Jose Campos, Rui Abreu. Diagnosing Software Faults Using Multiverse Analysis. Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20), 2020. P. 1629-1635.
7. Yehorova O.V. Prohramni zasoby dlia testuvannia prohramnoho zabezpechennia / O.V. Yehorova, V.P. Bychok // Molodyi vchenyi, Tekhnichni nauky. – 2019. – № 11 (75). – S. 680-684.
8. Analiz instrumentiv dlia avtomatyzovanoho testuvannia prohramnoho zabezpechennia / Olha Melkozerova, Aleksei Narezhnyi, Serhei Malakhov. // Kompiuterni nauky ta kiberbezpeka. – 2019. – № 1. – S.75-84. DOI: https://doi.org/10.26565/2519-2310-2019-1-07
9. W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. IEEE Transactions on Software Engineering, 42(8), 2016. P.707–740.
10. Alexandre Perez, Rui Abreu, Arie van Deursen A test-suite diagnosability metric for spectrum-based fault localization approaches. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). P. 654-664.
11. Spencer Pearson, Jose´ Campos, Rene´ Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017. P. 609–620. DOI: https://doi.org/10.1109/ICSE.2017.62
12. W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. IEEE Transactions on Reliability, 63(1):, IEEE, 2013. P. 290–308.
13. James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM international Con- ference on Automated software engineering, 2005. P. 273–282
14. Aaron Ang, Alexandre Perez, Arie van Deursen, Rui Abreu. Revisiting the practical use of automated software fault localization techniques. In 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2017. P. 175–182.
15. Diego Calvanese, Julien Corman, Davide Lanti, and Simon Razniewski Counting Query Answers over a DL-Lite Knowledge Base International Joint Conference on Artificial Intelligence, 2020. P. 1629–1635.

ДОДАТОК В

Презентаційні матеріали

Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

Гурман Іван Васильович

Удосконалення методу виявлення помилок при тестуванні програмного забезпечення на основі трасування стеку

Improving the Stack Trace-Based Method for Detection of Errors in Software Testing

Спеціальність 121 – інженерія програмного забезпечення
Освітньо-професійна програма – інженерія програмного забезпечення

Кваліфікаційна робота

Науковий керівник: Доктор фізико-математичних наук, професор Бедратюк Л.П.

ЗАГАЛЬНА ХАРАКТЕРИСТИКА КВАЛІФІКАЦІЙНОЇ РОБОТИ

Мета магістерської роботи полягає у покращенні ефективності роботи методу автоматичного виявлення помилок у програмному забезпеченні.

Об’єкт дослідження: процес автоматичного пошуку (виявлення) помилок у програмному забезпеченні.

Предмет дослідження: алгоритми, методи і засоби тестування програмного забезпечення

Завдання дослідження:

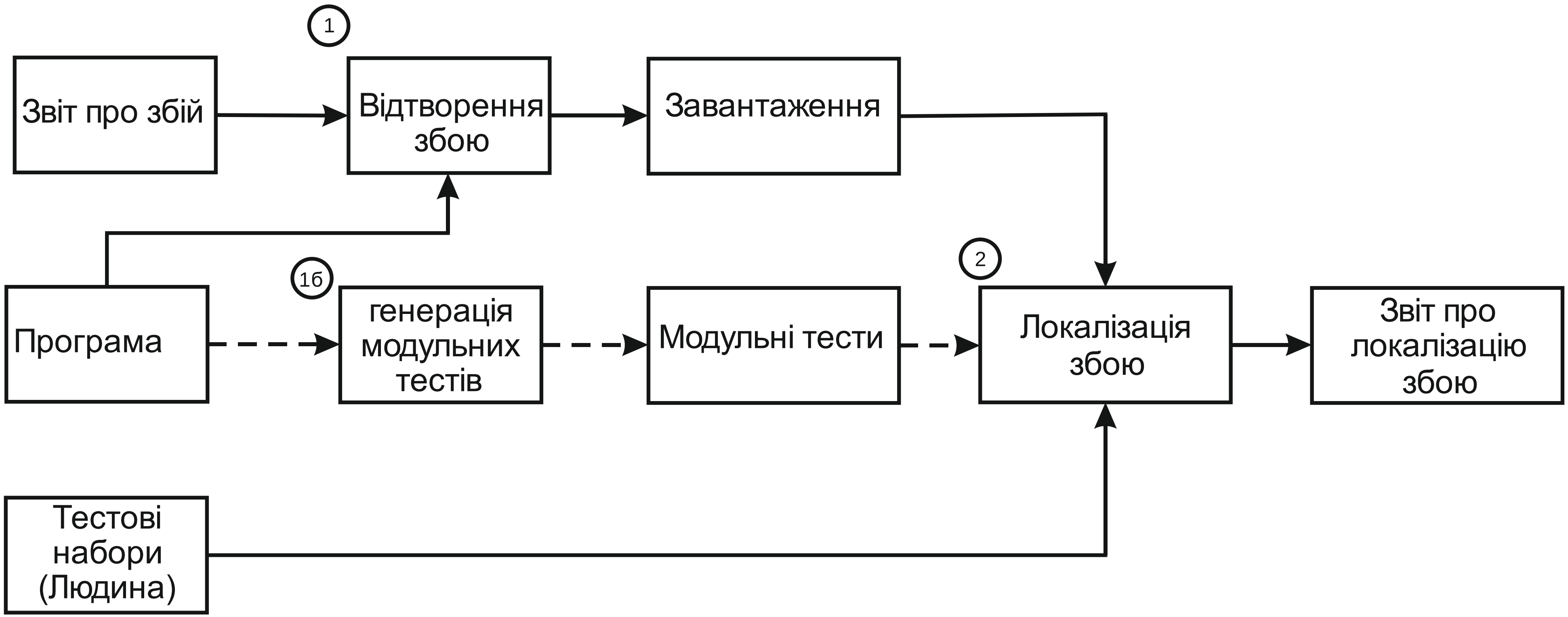
- Провести аналіз сучасних методів, алгоритмів і засобів тестування програмного забезпечення
- Здійснити порівняльний аналіз існуючих підходів до виявлення помилок у програмному забезпеченні
- Визначити як працює автоматична локалізація помилок у порівнянні з ручною відносно швидкості та точності визначення.
- Розробити алгоритм, який би дозволив автоматично виявляти помилки у програмному коді, базуючись на аналізі вмісту програмного стеку.

Наукова новизна роботи полягає в удосконаленні методу автоматичного визначення помилок у програмному забезпеченні за рахунок аналізу вмісту програмного стеку.

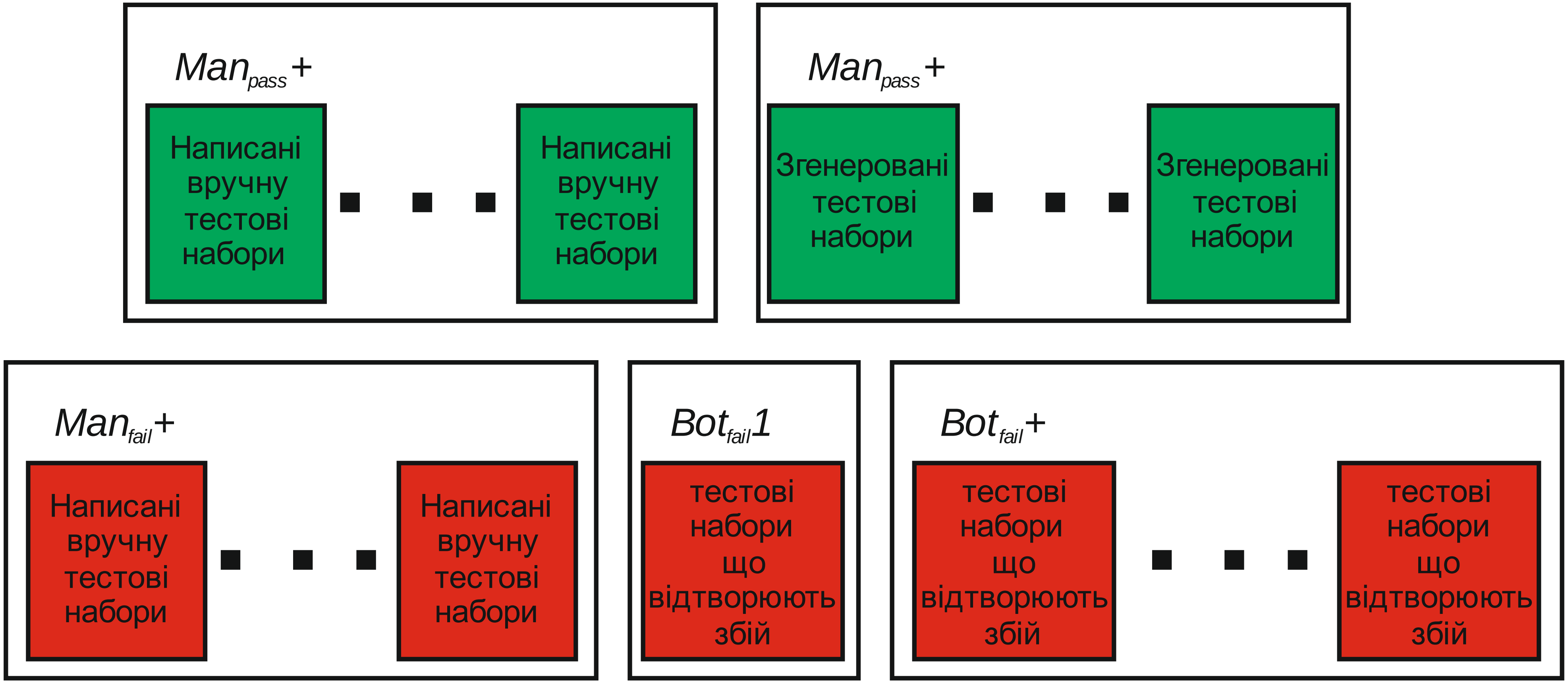
Методи дослідження аналітичні і експериментальні дослідження, методи системного аналізу, математичної статистики, теорії ймовірності.

Практична цінність полягає у підвищенні точності визначення місця в програмному коді, що призвело до збою у виконанні програми, що в свою чергу призведе до зменшення кількості людино годин при тестуванні та відлагодженні програмних продуктів.

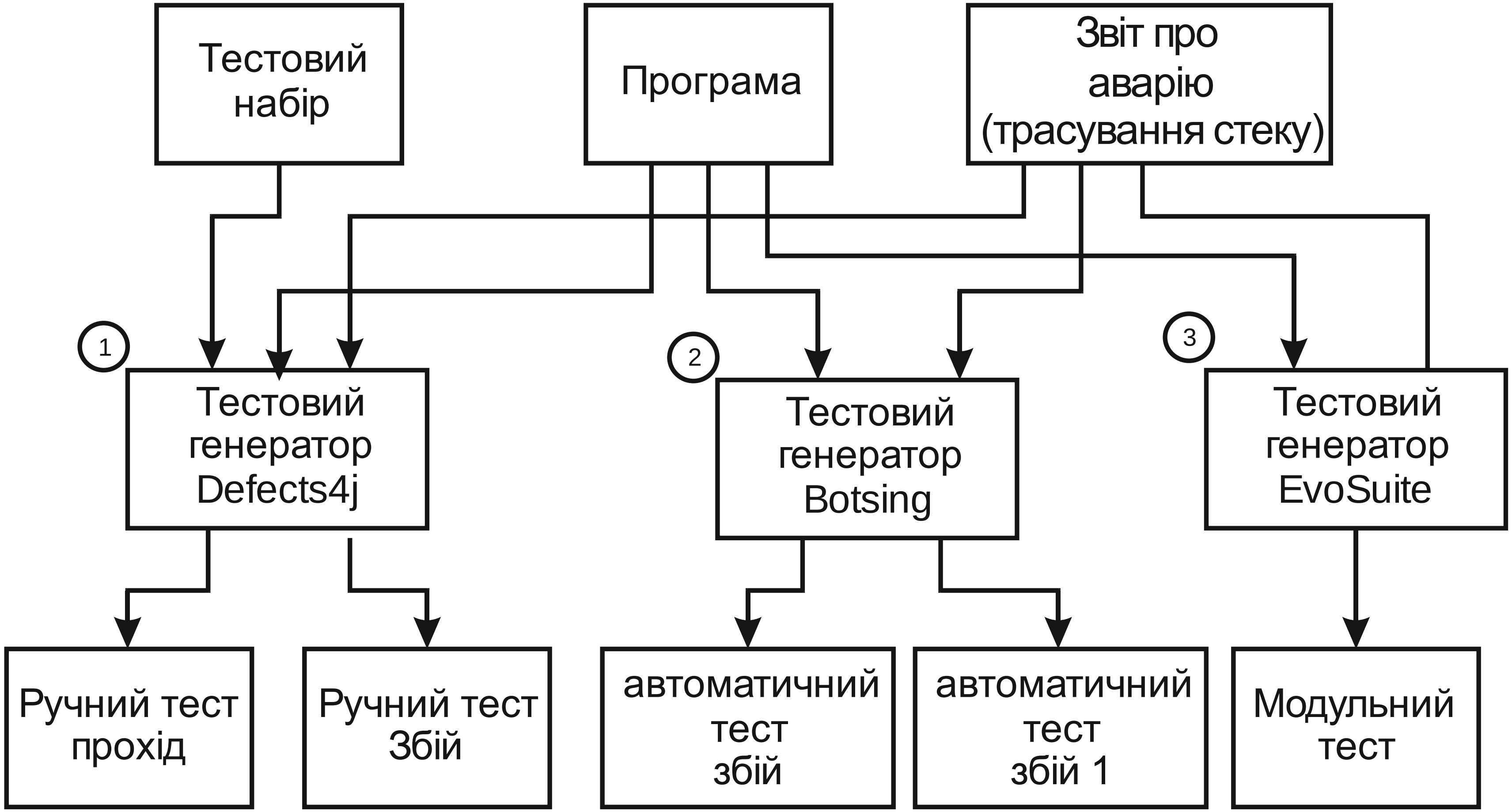
Публікації. По темі магістерської роботи опубліковано статтю в міжнародному науково-технічному журналі Вимірювальна та обчислювальна техніка в технологічних процесах №2 2021р.



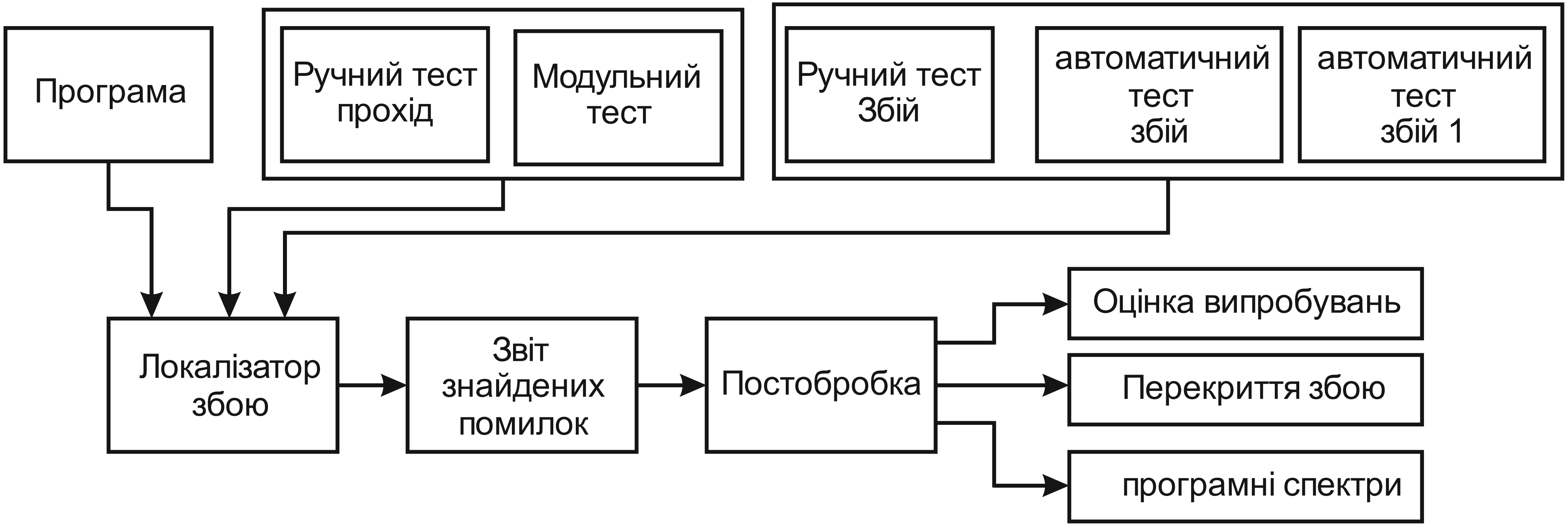
Автоматична локалізація помилок при аварії



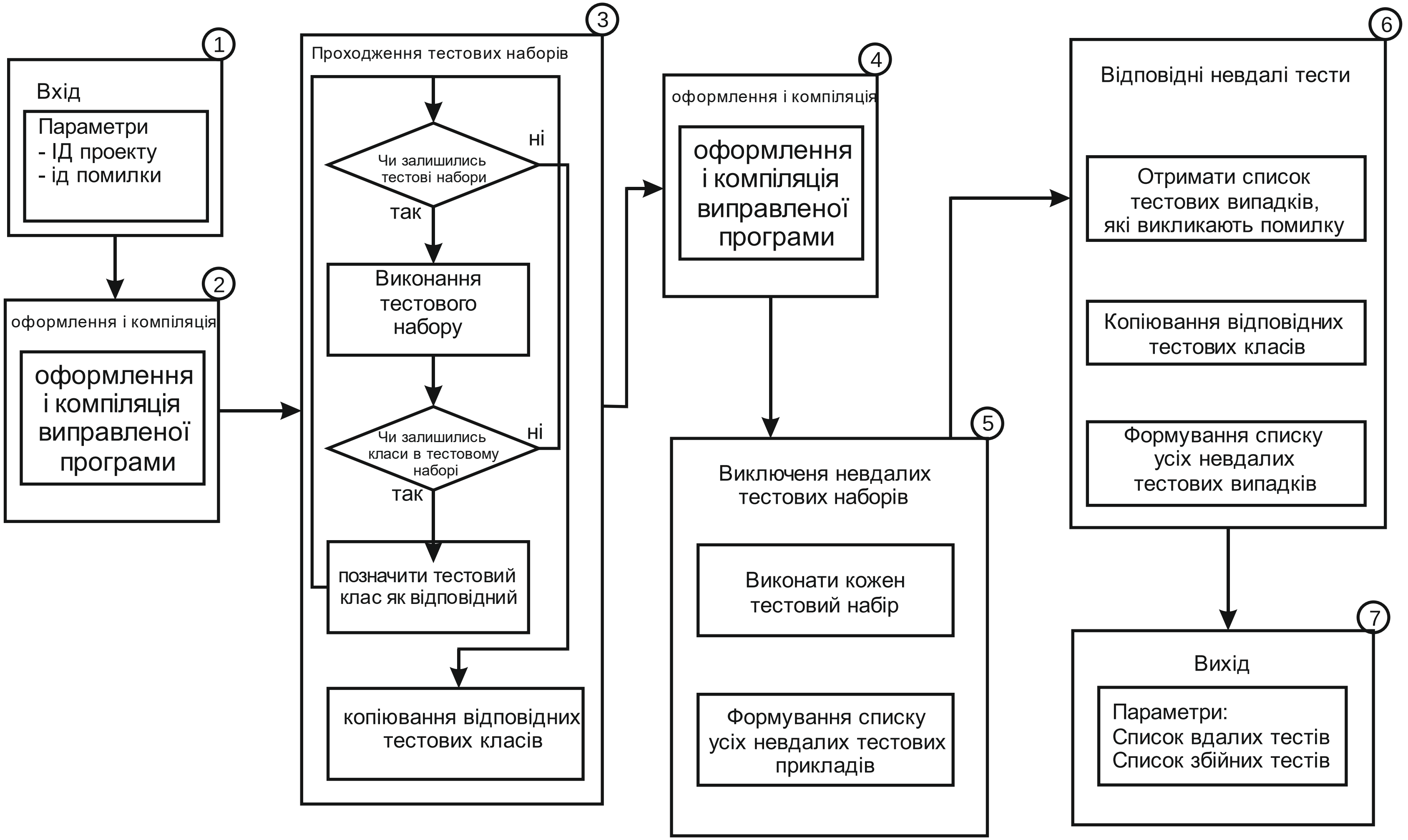
Набори тестів, що використовуються для оцінювання



Компоненти, що використовуються для отримання даних тестових наборів



Компоненти для отримання точок даних з набору даних.



Блок-схема процесу відсіювання тестових випадків defects4j-extractor

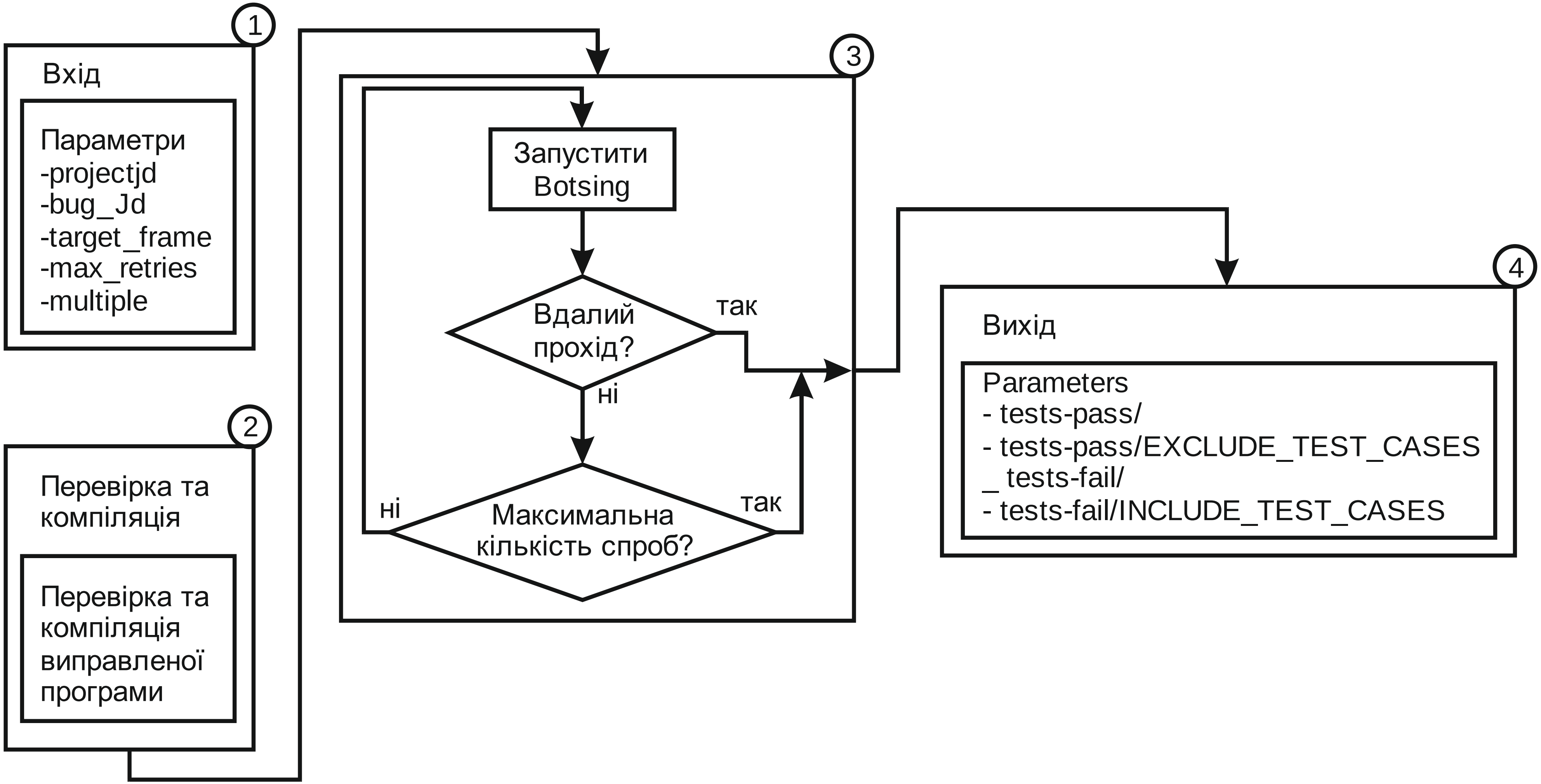


Схема генератора тест-кейсів BOTSING.

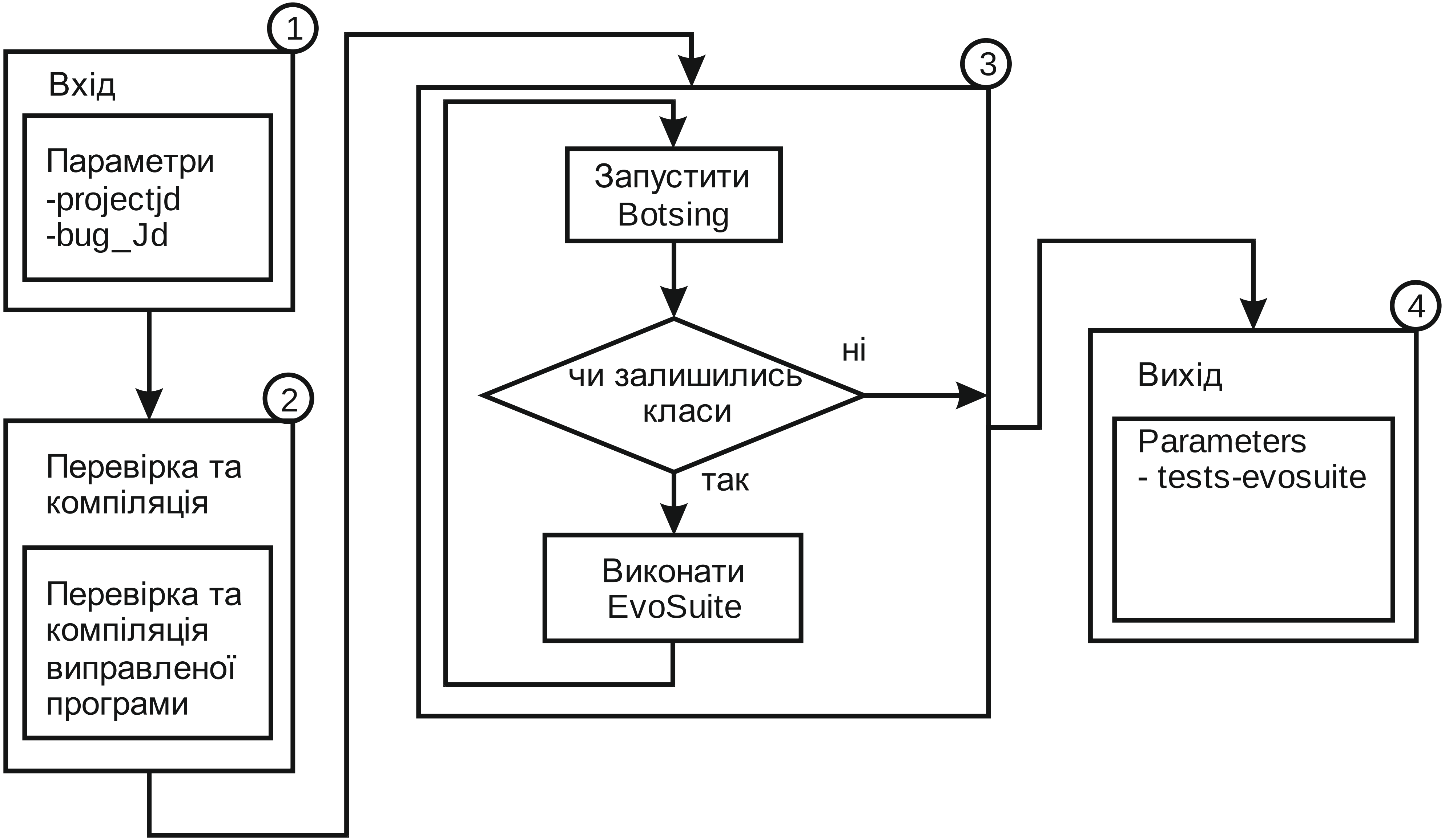
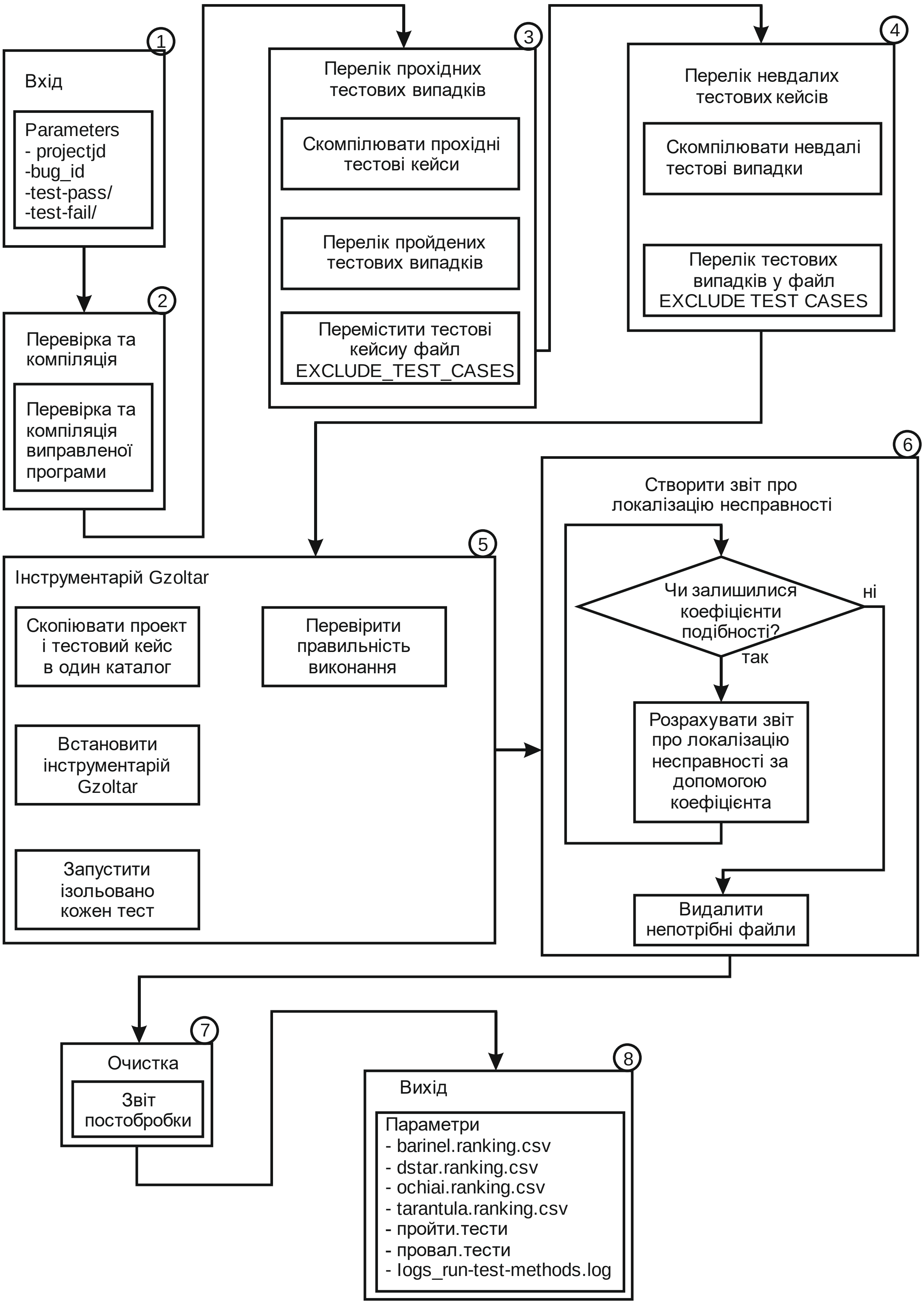
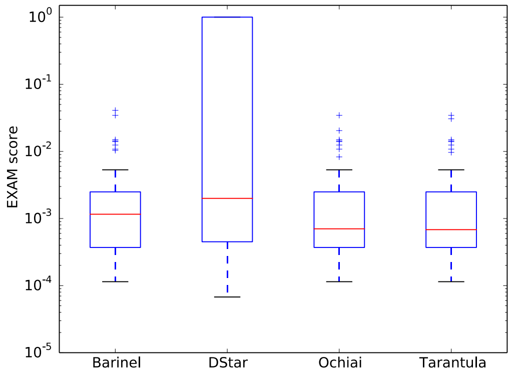


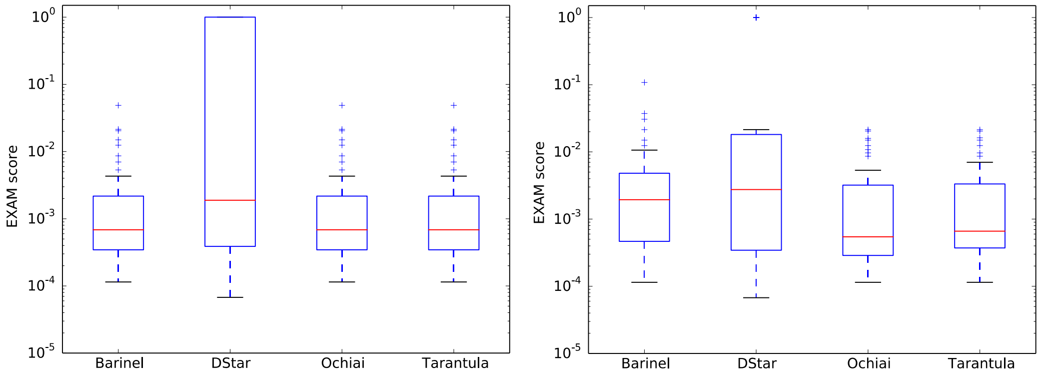
Схема генератора блок-тестів EVOSUITE



контейнера GZoltar

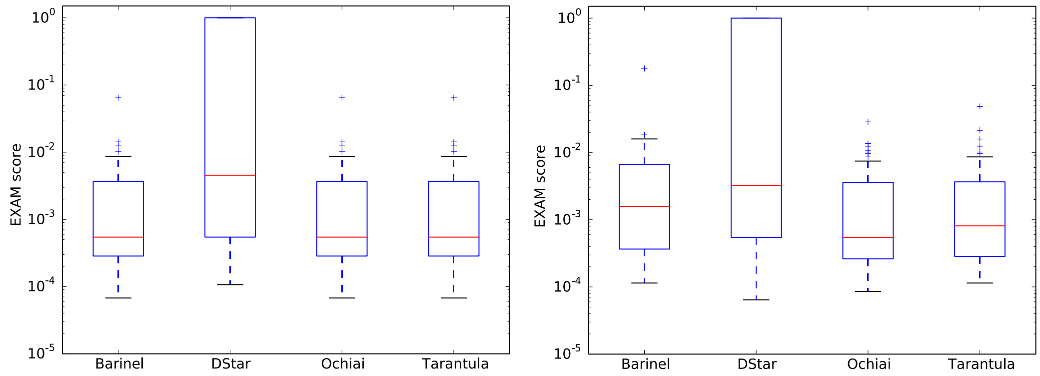


(a) Man$_{fail}^{+}$-Man$_{pass}^{+}$



(b) Bot$_{fail}^{1}$-Man$_{pass}^{+}$

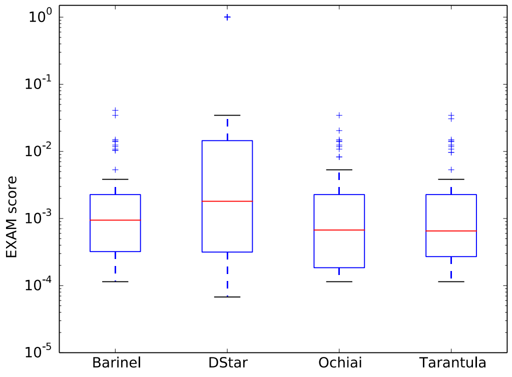
(c) Bot$_{fail}^{+}$-Man$_{pass}^{+}$



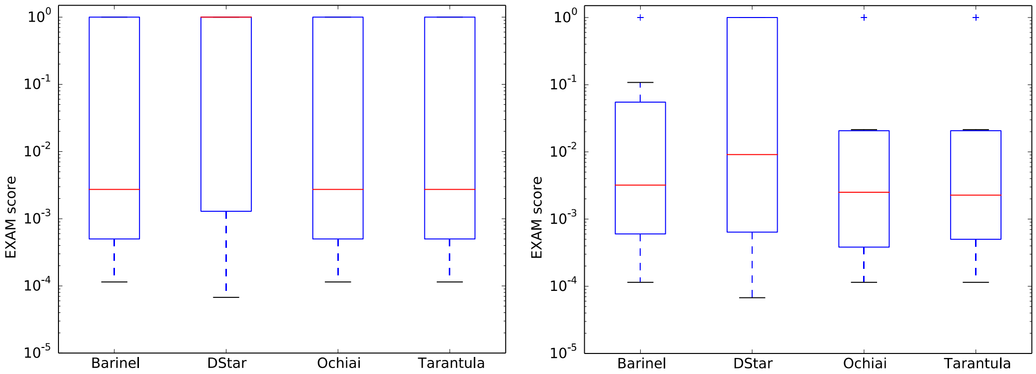
(d) Bot$_{fail}^{1}$-Evo$_{pass}^{+}$

(e) Bot$_{fail}^{+}$-Evo$_{pass}^{+}$

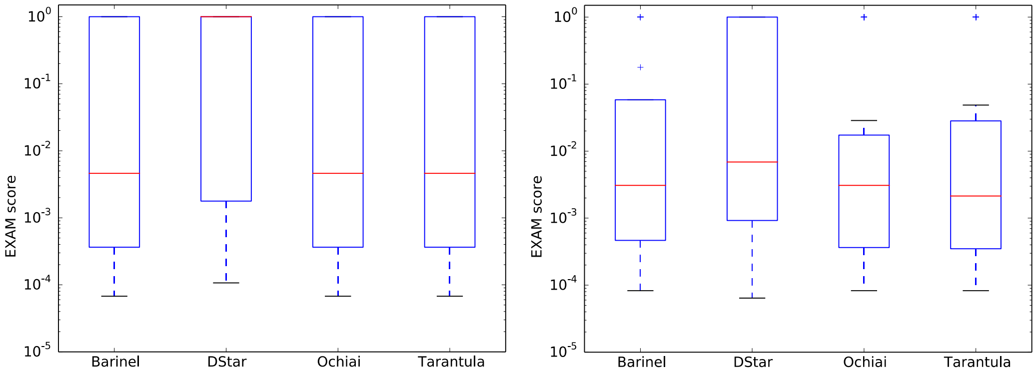
Результати показника EXAM для 80 наборів тестів загальних для Bot$_{fail}$1 і Bot$_{fail}$+, з тестовим прикладом, що відтворює аварійне завершення роботи



(a) $\text{Man}_{fail}{}^{+}$-$\text{Man}_{pass}{}^{+}$



(b) $\text{Bot}_{fail}{}^{1}$-$\text{Man}_{pass}{}^{+}$ (c) $\text{Bot}_{fail}{}^{+}$-$\text{Man}_{pass}{}^{+}$

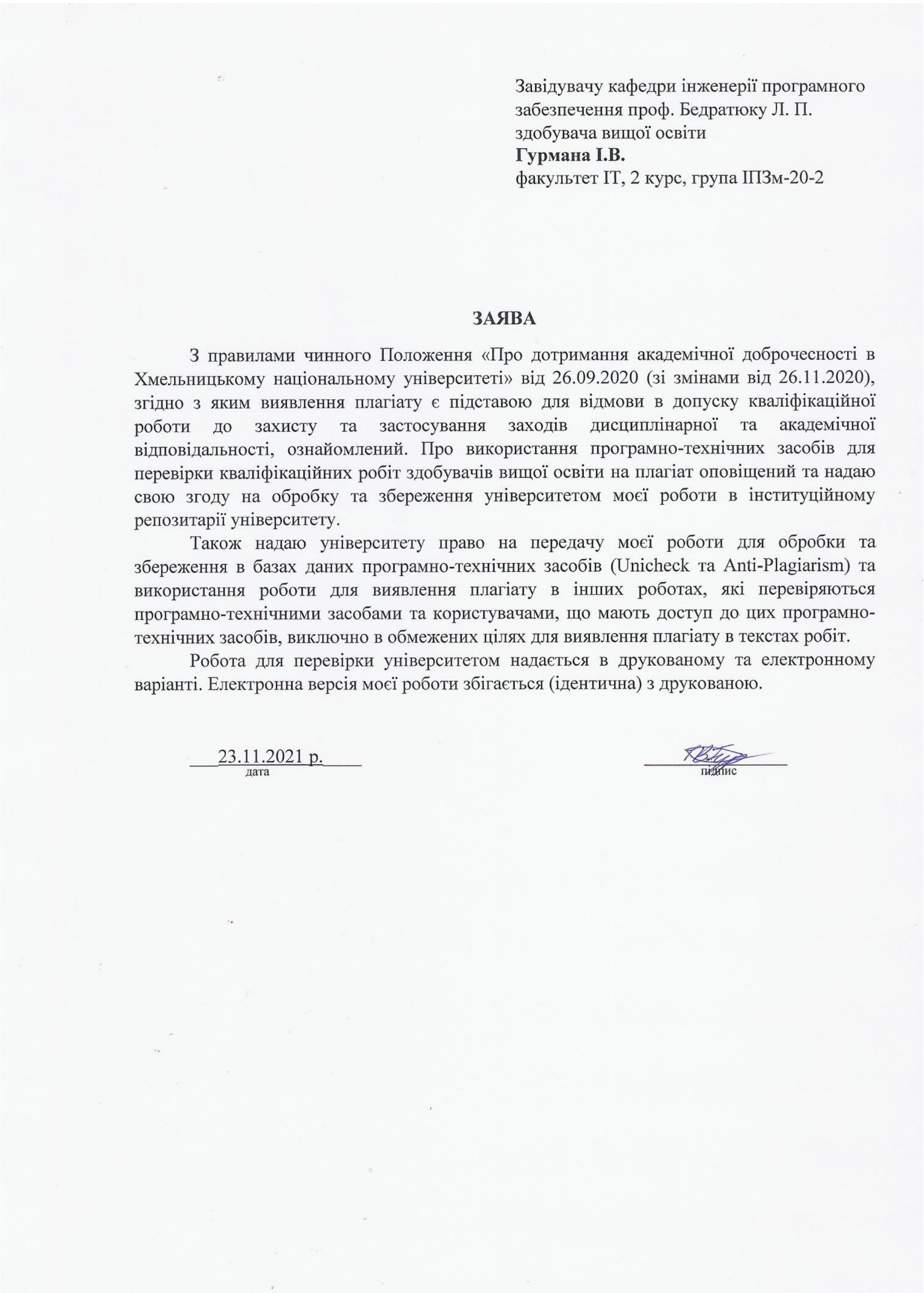


(d) $\text{Bot}_{fail}{}^{1}$-$\text{Evo}_{pass}{}^{+}$ (e) $\text{Bot}_{fail}{}^{+}$-$\text{Evo}_{pass}{}^{+}$

Оцінка за показникомEXAM 51-го тестового набору із невдалим тестовим прикладом, що відтворює збій, що викликає помилку (найкращий сценарій)

ВИСНОВКИ

У цій дипломній роботі представлено, автоматизований конвеєр локалізації несправностей, у якому ми поєднали відтворення аварії на основі пошуку та локалізацію несправностей на основі спектру. Конвеєр створює тестові приклади, що відтворюють аварійне завершення роботи, із трасування стека, включеного в звіт про аварійне завершення роботи. Ці тестові приклади в поєднанні з існуючим або автоматично згенерованим набором тестів використовуються як вхідні дані для процесу локалізації несправностей. Використовуючи такий підхід до локалізації помилок, ідентифікуються потенційно помилкові рядки коду з основною метою зменшення зусиль з відлагодження для розробників.



Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Гурмана І.В.
факультет ІТ, 2 курс, група ІПЗм-20-2

ЗАЯВА

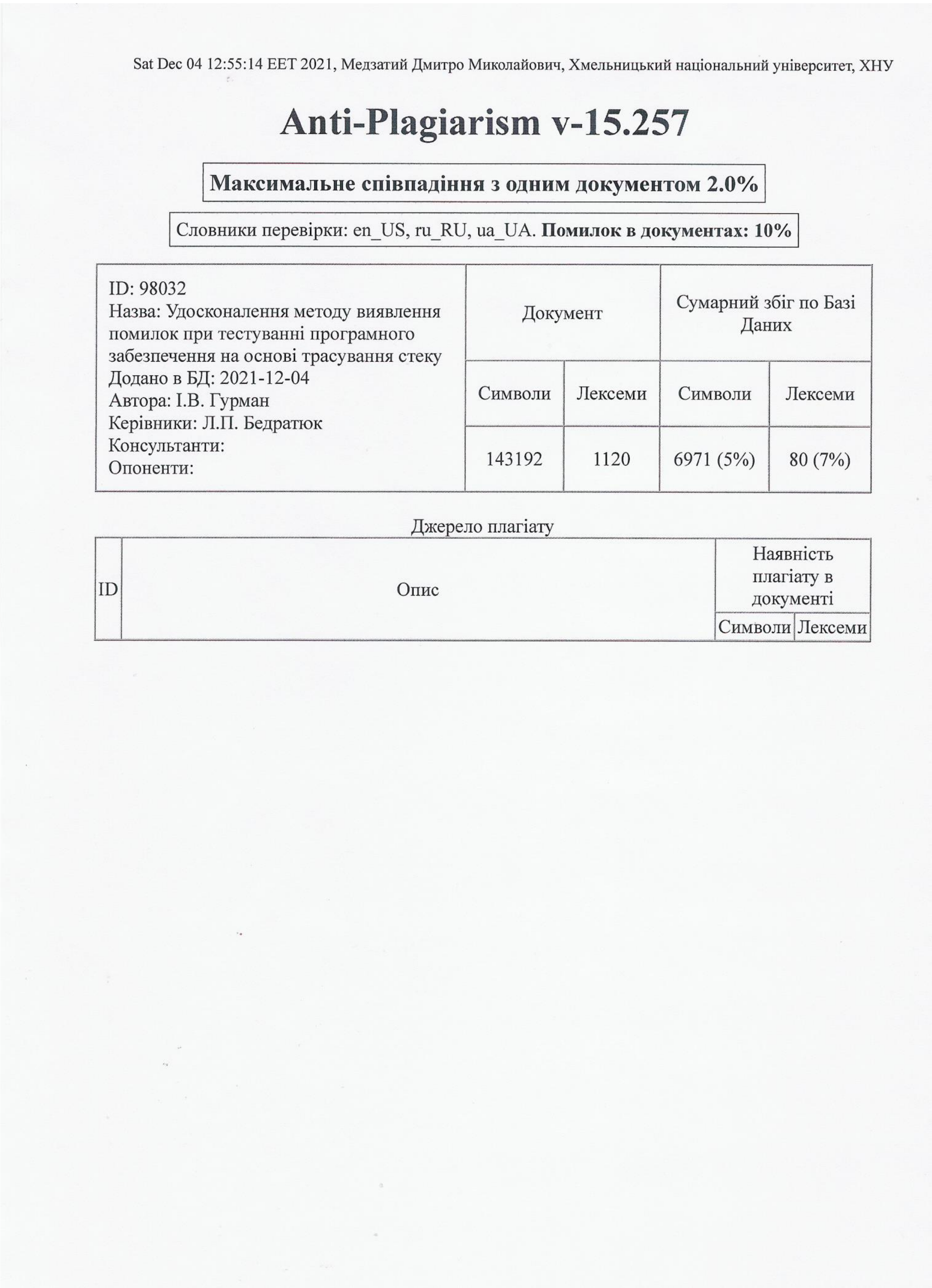
З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

___23.11.2021 р.____
дата

____
підпис



Sat Dec 04 12:55:14 EET 2021, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

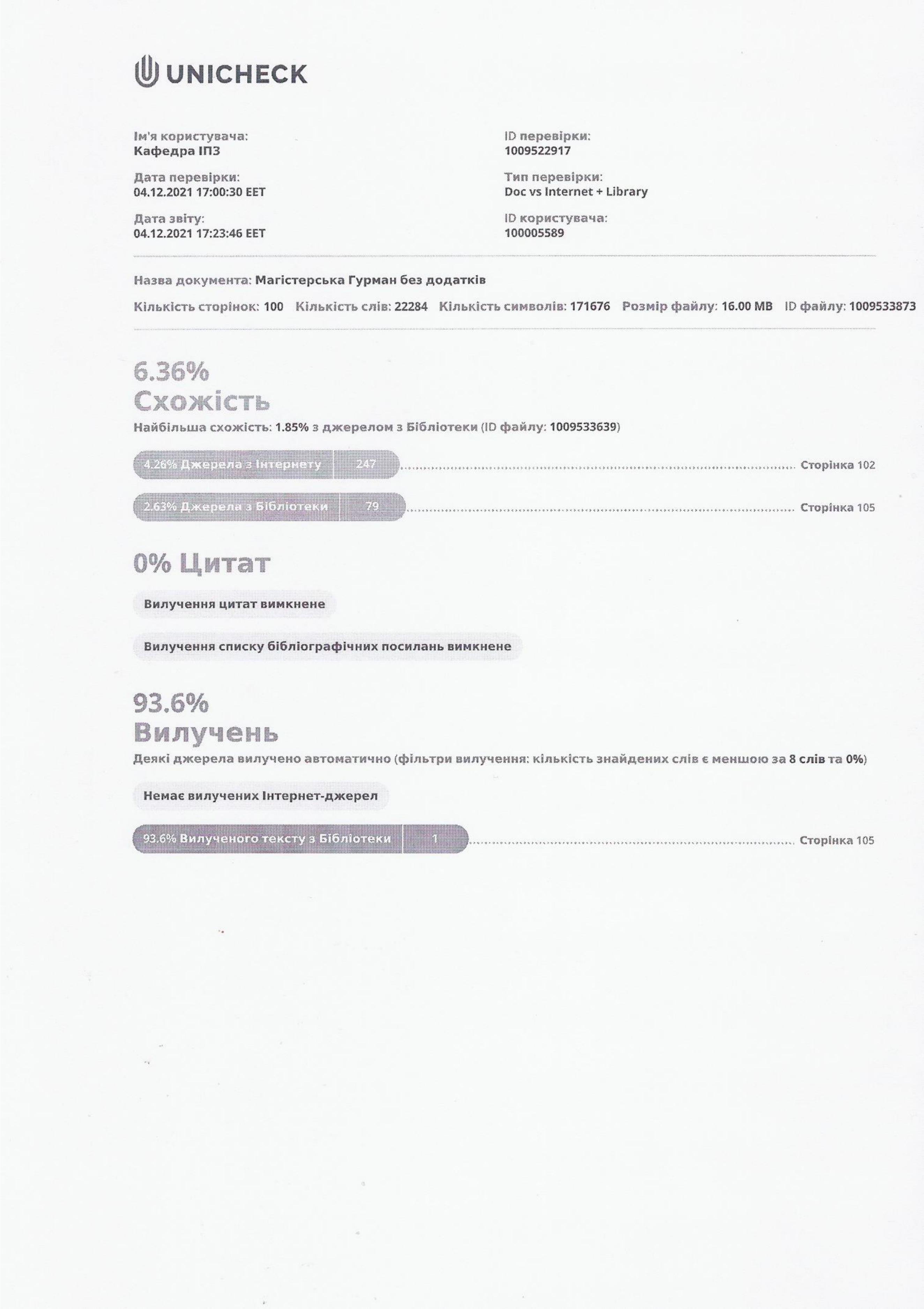
Максимальне співпадіння з одним документом 2.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 10%**

ID: 98032 Назва: Удосконалення методу виявлення помилок при тестуванні програмного забезпечення на основі трасування стеку Додано в БД: 2021-12-04 Автора: І.В. Гурман Керівники: Л.П. Бедратюк Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	143192	1120	6971 (5%)	80 (7%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми



UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1009522917

Дата перевірки:
04.12.2021 17:00:30 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
04.12.2021 17:23:46 EET

ID користувача:
100005589

Назва документа: Магістерська Гурман без додатків

Кількість сторінок: 100 **Кількість слів:** 22284 **Кількість символів:** 171676 **Розмір файлу:** 16.00 MB **ID файлу:** 1009533873

6.36%
Схожість

Найбільша схожість: 1.85% з джерелом з Бібліотеки (ID файлу: 1009533639)

4.26% Джерела з Інтернету 247 .... Сторінка 102

2.63% Джерела з Бібліотеки 79 .... Сторінка 105

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

93.6%
Вилучень

Деякі джерела вилучено автоматично (фільтри вилучення: кількість знайдених слів є меншою за 8 слів та 0%)

Немає вилучених Інтернет-джерел

93.6% Вилученого тексту з Бібліотеки 1 .... Сторінка 105

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Дипломник Гурман Іван Васильович

Тема Удосконалення методу виявлення помилок при тестуванні програмного забезпечення на основі трасування стеку

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломної роботи:

Кількість листів креслень 0; кількість сторінок записки 99

1.Короткий зміст ДР та прийнятих рішень В рамках магістерської роботи проведено аналіз методів а алгоритмів автоматичного пошуку ромилок в програмному забезпеченні після виникнення збоїв у його роботі. Удосконалено метод локазізації помилок програмного коду за рахунок використання інформації з програмного стеку для більш точної генерації тестових кейсів перевірки програмного забезпечення

2. Висновок про відповідність ДР поставленому завданню Дипломна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як в теоретичній, так і в практичній частині дипломної роботи.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі обґрунтовується актуальність теми роботи, формулюються цілі та завдання дослідження, описується практична значимість отриманих результатів. У першому розділі розглянуті наукові праці різних дослідників в області автоматичної локалізації помилок програмного коду. В другому розділі описуються методи вирішення поставленої задачі. У третьому розділі описуються алгоритми розв’язання задачі. У четвертому розділі описана реалізація системи що ґрунтується на прийнятих проектних рішеннях. У п’ятому розділі проведено аналіз ефективності запропонованого підходу

4. Позитивні сторони роботи Дипломна робота має інноваційне рішення, а саме автоматизований конвеєр локалізації і якому поєднано відтворення збою програми на основі даних з програмного стеку, та пошук несправності на основі спектру

5. Негативні сторони роботи Для відтворення збою програми потрібно мати звіт про аварійне завершення роботи з вмістом програмного стеку, що не завжди можливо коли користувач віддалений від розробників.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми дипломної роботи з дотриманням стандартів. В загальному графічне оформлення виконане на достатньому рівні. Пояснювальна записка відповідає нормам для її оформлення.

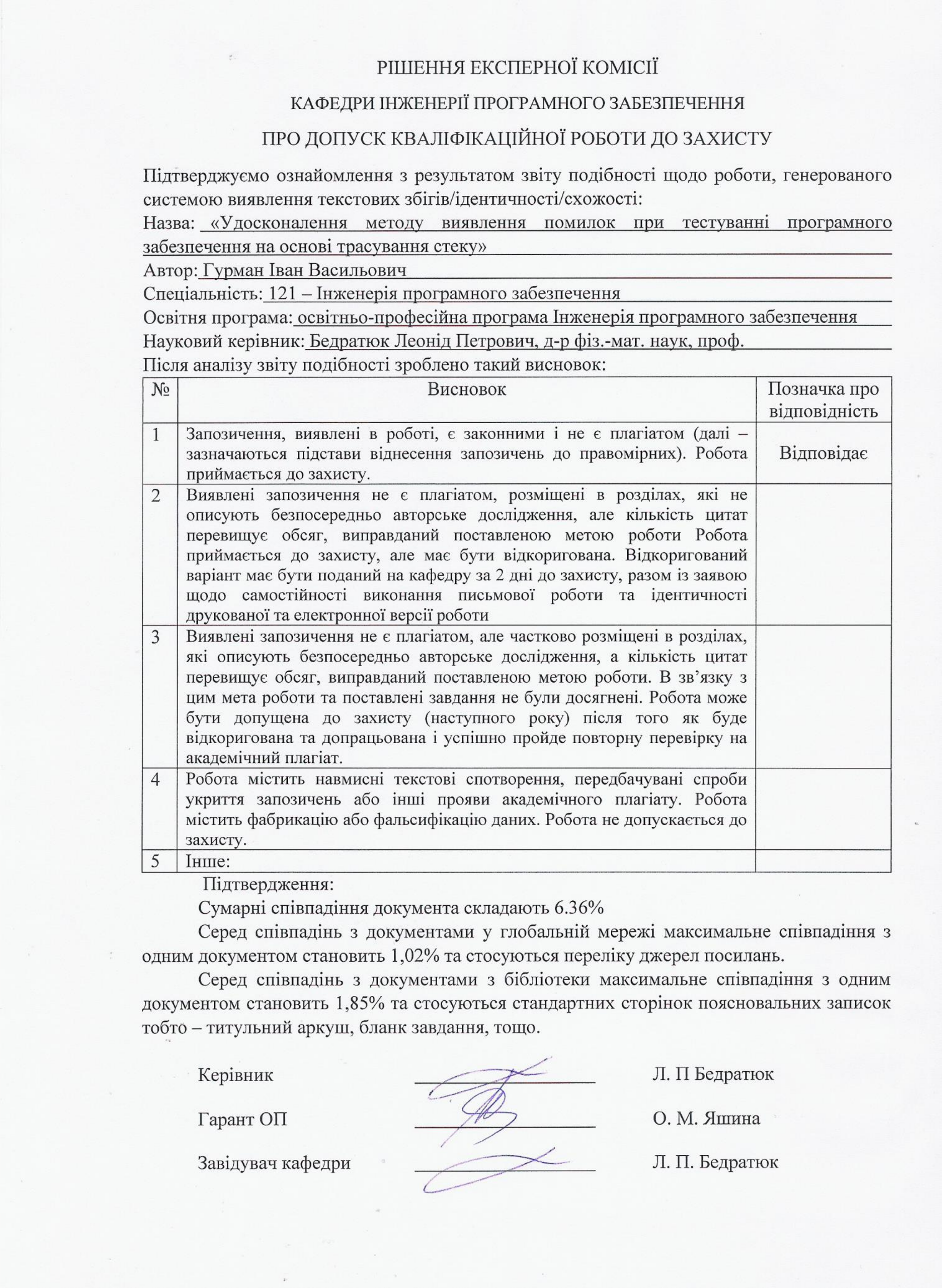
7. Відгук про роботу в цілому В загальному дипломна робота заслуговує позитивної оцінки. Весь матеріал дипломної роботи структурований, чіткий та послідовний. Усі розділи роботи послідовні та логічні, що дозволяє чітко розуміти викладений матеріал в рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для досягнення поставленої задачі.

8. Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінку «добре».

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) Нечипорук Андрій Олександрович, к.т.н, доцент кафедри комп'ютерної інженерії та інформаційних систем.

“ 3 ” 12 2021 р. ____________
(підпис)



РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Удосконалення методу виявлення помилок при тестуванні програмного забезпечення на основі трасування стеку»

Автор: Гурман Іван Васильович

Спеціальність: 121 – Інженерія програмного забезпечення

Освітня програма: освітньо-професійна програма Інженерія програмного забезпечення

Науковий керівник: Бедратюк Леонід Петрович, д-р фіз.-мат. наук, проф.

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних). Робота приймається до захисту.	Відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Сумарні співпадіння документа складають 6.36%

Серед співпадінь з документами у глобальній мережі максимальне співпадіння з одним документом становить 1,02% та стосуються переліку джерел посилань.

Серед співпадінь з документами з бібліотеки максимальне співпадіння з одним документом становить 1,85% та стосуються стандартних сторінок пояснювальних записок тобто – титульний аркуш, бланк завдання, тощо.

Керівник ______________ Л. П Бедратюк

Гарант ОП ______________ О. М. Яшина

Завідувач кафедри ______________ Л. П. Бедратюк