Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Метод використання неконтрольованого введення типу для мов програмування при
Назва теми
автоматичній генерації тестів

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.2301115.01.08.ПЗ

Виконав студент 2 курсу, група ІПЗм-23-1 ________ Підпис Д. О. Юртаєв Ім'я, ПРІЗВИЩЕ

Керівник канд. пед. наук, доцент Науковий ступінь, звання ________ Підпис Н. І. Праворська Ім'я, ПРІЗВИЩЕ

Нормоконтролер канд. техн. наук, доцент ________ Підпис Ю. В. Форкун Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри
інженерії програмного забезпечення ________ Підпис Л. П. Бедратюк Ім'я, ПРІЗВИЩЕ

Дата 02.12.2024

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
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
02.09.2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Юртаєв Денис Олександрович
Прізвище, ім'я, по батькові здобувача

1. Тема роботи Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів

Керівник роботи канд. пед. наук, доцент Праворська Н. І.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 26.08.2024 р. № 60
2. Строк подання студентом роботи на кафедру 02.12.2024 р.
3. Вихідні дані до роботи Матеріали науково-дослідної практики

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

1. Дослідження предметної області та постановка задачі
2. Підходи та методи вирішення поставлених задач
3. Проектування програмного забезпечення
4. Реалізація та тестування програмного забезпечення

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

Презентаційні матеріали (слайди)

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Ю. В. Форкун		
Нормоконтроль	Ю. В. Форкун		

7. Дата видачі завдання « 02 » вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Вивчення предметної області, формування мети та задач дослідження, визначення об'єкта та предмета дослідження	02.09-10.09.2024	
2 Робота над розділом 1 – вивчення джерел, аналіз сучасних підходів, методів і засобів за темою роботи, визначення задач, висновки	11.09-25.09.2024	
3 Робота над розділом 2 – розробка підходів дослідження поставленої задачі, висновки	26.09-10.10.2024	
4 Робота над науковими статтями	11.10-30.10.2024	
5 Робота над розділом 3 – аналіз вимог до програмного застосунку, розробка структури та вибір технологій, висновки	11.10-26.10.2024	
6 Робота над етапом 4 – реалізація та тестування програмного забезпечення	27.10-17.11.2024	
7 Попередній захист дипломної роботи	Листопад (згідно графіку)	
8 Узгодження постановки задач, отриманих результатів та висновків, написання вступу, загальних висновків, оформлення джерел та додатків, оформлення записки та графічних матеріалів відповідно до вимог	18.11-30.11.2024	
9 Перевірка роботи на наявність плагіату, нормоконтроль, брошурування пояснювальної записки, підготовка супровідних документів	01.12-04.12.2024	
10 Підготовка до захисту кваліфікаційної роботи	з 01.12.2024	

Студент ____________ Підпис ____________ Юртаєв. Д. О. Ім'я, ПРІЗВИЩЕ

Керівник роботи ____________ Підпис ____________ Праворська. Н. І. Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи: Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів.

Автор роботи: Юртаєв Денис Олександрович.

Керівник роботи: Праворська Наталія Іванівна.

Пояснювальна записка 110 с., 28 рис., 20 табл., 3 дод., 45 джерел.

Об'єктом даного дослідження є вихідні програмні коди, в яких визначаються типи даних необхідні для автоматичної генерації тестів.

Предметом дослідження даної кваліфікаційної роботи є метод неконтрольованого виведення типів для динамічних мов програмування під час автоматичної генерації тестів.

Вперше було запропоновано підхід, який комбінує статичний та динамічний аналіз для виведення типів вихідного коду з подальшою метою використання виведених типів замість випадкових під час автоматичної генерації тестів для мови програмування JavaScript.

Головною метою даної роботи є дослідження, аналіз та реалізація методу виведення типу під час автоматичної генерації тестів та їх ефективність.

Для досягнення поставленого завдання було проведено детальний аналіз предметної області та проаналізовано сучасні дослідження. На основі отриманої інформації було визначено технології та методи, розроблено програмне забезпечення з допомогою якого було проведено тестування.

Отримані результати дослідження демонструють покращення тестування в деяких випадках. Враховуючи результати, можна стверджувати, що використання методу виведення типу під час генерації тестів покращує можливості генерації тестів. Також стає зрозумілим, що в деяких ситуаціях це не приносить користі. Було виявлено деякі специфічні випадки, які не дають в повній мірі використати наш підхід.

Підпис

02.12.2024

Дата

ЗМІСТ

ВСТУП 6
1 ТЕОРИТИЧНІ ОСНОВИ ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ 8
1.1 Аналіз предметної області 8
1.2 Аналіз сучасних досліджень 16
1.3 Постановка задачі 23
1.4 Висновки 25
2 КОНЦЕПЦІЇ, МОДЕЛІ ТА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ 28
2.1 Концепція автоматичної генерації тестів 28
2.2 Динамічно типізовані мови програмування 30
2.3 Вивід типів 31
2.4 Моделі та методи удосконалення генерації тестів 35
2.5 Висновки 53
3 АЛГОРИТМИ ТА ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ 55
3.1 Алгоритми вирішення задачі 55
3.2 Проектування програмного засобу 57
3.3 Вибір технологій 67
3.4 Висновки 70
4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 71
4.1 Програмна реалізація 71
4.2 Розробка тестових сценаріїв 80
4.3 Аналіз результатів 87
4.4 Висновки 99
ВИСНОВКИ 101
ПЕРЕЛІК ДЖЕРЕЛ 103
ДОДАТОК А 110
ДОДАТОК Б 140
ДОДАТОК В 144

ВСТУП

Розробка програмного забезпечення є досить не простим процесом, який потребує немало ресурсів. Не менш ресурсовитратним процесом є процес написання тестів, який досить часто виконується вручну тими самими програмістами, які писали код. Покриття проекту тестами надає багато переваг, проте забирає велику кількість часу в розробників, але й без тестів обійтись також не просто, оскільки помилки та невірне виконання коду без них виявляти стає досить важко.

Досить важливою темою є автоматизація цього процесу. Тести згенеровані програмою не тільки економлять витрати людського ресурсу, але й виключають людський фактор з процесу написання тестів, що своєю чергою зменшує шанс появи помилок в тестах.

Можливості генерування тестів є досить різні, та підходи також відрізняються, оскільки мови програмування також досить різноманітні та мають свою специфіку. Деякі з підходів навіть використовують технологію обробку природної мови. Зазвичай дана технологія використовує машинне навчання, тому це не наш випадок. В нашому випадку дослідження націлене на пошуковий метод генерації, а саме його удосконалення.

На сьогодні, JavaScript є динамічно типізованою мовою програмування зі слабкою типізацією, в якій навіть відсутня можливість ручної анотації типів, що робить типи даних абсолютно неконтрольованими. Це є досить суттєвим фактором, оскільки багато алгоритмів пошукової генерації тестів, використовують випадкові типи даних. Дане дослідження спробує доповнити генерацію тестів механізмом, який зможе виконувати припущення що до типів даних, які необхідно використати під час генерації тестових випадків. Це буде зроблено за допомогою впровадження статичного та динамічного аналізу, припускаючи, що це має позитивно вплинути на обсяг покриття тестів.

Об’єктом даного дослідження є вихідні програмні коди в, яких визначаються типи даних необхідні для автоматичної генерації тестів.

Предметом дослідження кваліфікаційної роботи є метод неконтрольованого виведення типів для динамічних мов програмування під час автоматичної генерації тестів.

Головною метою даної роботи є детальне дослідження методу виведення типу під час автоматичної генерації тестів та їх ефективність.

Наукова новизна полягає в тому, що вперше було запропоновано вдосконалений підхід до аналізу вихідного програмного коду за допомогою статичного аналізу доповненого динамічним аналізом. Це дозволяє визначати більшість типів даних і, відповідно, генерувати більшу кількість тестових випадків.

Крім того, після етапу генерації тестів передбачається їх додатковий аналіз за допомогою динамічного аналізу, що дає змогу виправляти помилкові тестові випадки та покращувати їхню якість. Для досягнення поставленої мети було вирішено наступні завдання:

– проведено аналіз предметної області;

– здійснено аналіз сучасних досліджень;

– визначено методи, які допоможуть з виведенням типів під час генерації тестових випадків;

– сформовано структуру ПЗ, на основі якого було розроблено інструмент;

– визначили ряд технологій, які було використано для розробки інструменту та його тестування;

– проведено тести розробленого інструменту, на основі тестових проектів, здійснено аналіз результатів та зроблено висновки щодо ефективності запропонованого нами рішення.

1 ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Аналіз предметної області

Автоматичні тести – це тести, які генеруються автоматично до вже готового програмного коду та перевіряють на те як відповідний програмний код поводить себе з різними вхідними даними та чи відповідає результат нашим очікуванням. Зазвичай написанням тестів займаються ті самі програмісти, які писали код що підлягає тестуванню.

Традиційне тестування програмного забезпечення є досить трудомістким та дорогим процесом, який доводиться виконувати самостійно. В той самий час це є обов’язковою частиною розробки програмного забезпечення. Щоб полегшити процес ручного написання тестів, протягом останніх десятиліть, було проведено деякі дослідження та винайдено деякі техніки автоматичної генерації тестів.

На сьогодні, автоматична генерація тестів для мов програмування є однією з ключових тем у галузі розробки програмного забезпечення. Використання неконтрольованого введення типів стає все більш актуальним у контексті динамічно типізованих мов, таких як Python і JavaScript, де типи не завжди явно визначені.

Автоматична генерація тестів дозволяє зменшити витрату ресурсів розробників, спрямувавши час та кошти на інші більш важливі задачі. Для великих проектів написання тестів є дійсно досить трудомістким та дорогим процесом, проте допомагає виловити велику кількість помилок до релізу проект. Це також скорочує цикл розробки та мінімізує людський фактор у тестуванні.

Проте в даній темі не все так чудово як звучить, на перший погляд, оскільки тема є досить складною та може призвести до певних проблем, неточностей та в цілому складнощів у реалізації. Використання неконтрольованого введення типу для мов програмування під час автоматичної генерації тестів може призвести до серйозних проблем, таких як:

– уразливості в системі, неконтрольоване введення типу може створити лазівки для виконання несанкціонованого коду, наприклад, якщо користувач може

передати непередбачені типи даних, це може спричинити виконання небезпечних команд або викликати небажану поведінку програми;

– помилки в тестах, якщо типи даних не перевіряються належним чином, тестові випадки можуть бути створені некоректно, що призведе до неправдивих результатів тестування або непередбачених збоїв під час виконання;

– некоректні результати тестування, тести, створені на основі неконтрольованих типів введення, можуть не відповідати реальним сценаріям використання програми, що призведе до ненадійних або неправильних результатів;

– складність підтримки, неконтрольоване введення типу робить код важким для розуміння та підтримки, оскільки важко передбачити, які типи даних можуть бути введені, і як вони будуть оброблятися програмою.

Для уникнення цих проблем рекомендується:

– використовувати строгі перевірки типів на всіх етапах введення даних;

– перевіряти введення перед генерацією тестів;

– статичний аналіз коду для виявлення потенційних проблем з типами даних;

– автоматичне тестування з використанням типізованих мов або інструментів, що підтримують строгий контроль типів.

Таким чином, правильне використання неконтрольованого введення типів в автоматичній генерації тестів може суттєво полегшити процес тестування, підвищити його ефективність та знизити вартість розробки програмного забезпечення, але вимагає обережного підходу до мінімізації можливих ризиків.

Динамічні мови програмування є досить поширеними серед розробників програмного забезпечення, оскільки вони не вимагають точного декларування типів змінних, які вони використовують в коді. Це значно спрощує написання коду та пришвидшує процес розробки. Прикладами таких мов є JavaScript та Python [1]. Саме ці мови програмування на теперішній час є одними з найпопулярніших та найбільш використовуваними в різних сферах розробки програмних систем. Проте найбільше досліджень автоматизації тестів зроблено навколо типізованих мов програмування. Ці підходи покладаються на статичну інформацію про тип для генерації правильного введення вхідних даних для функцій і викликів конструктора

в тестів випадків. У динамічних мовах програмування розробникам не потрібно надавати цю інформацію, що дозволяє швидко розробляти прототипи функціональних можливостей без потреби у складній системі типів [2].

Таким чином, правильне використання неконтрольованого введення типів в автоматичній генерації тестів може суттєво полегшити процес тестування, підвищити його ефективність та знизити вартість розробки програмного забезпечення, але вимагає обережного підходу до мінімізації можливих ризиків.

Нехтування тестами під час розробки програмного забезпечення призводить до непередбачуваних помилок під час його робити, тому розробляти тести до програмних продуктів рекомендовано та є ключовим фактором для розробки якісного програмного продукту.

Дана робота сконцентрована не так детально на автоматичній генерації тестів, як такій, а на тому, як саме виведення типів даних в динамічно типізованих мовах програмування з неконтрольованими типами зможемо допомогти при генерації тестових випадків. А також допоможе дізнатись наскільки результат від цього зміниться та скільки ресурсів доведеться витратити на генерацію тестових випадків.

Вплив продуктивності можна виміряти, дивлячись на досягнуте структурне покриття та можливості виявлення несправностей згенерованих тестових випадків. Відбувається оцінювання того, чи може процес бути недорогим у часі, порівнюючи час, використаний процесом визначення типу, із загальним часом, використаним процесом створення тестового випадку. Загальна гіпотеза полягає в тому, що інформація про виконання згенерованих тестів дозволяє більш точно визначити тип. Крім того, вдосконалення виведених типів покращує можливості генерації тестів. Ця взаємність може створити цикл зворотного зв’язку, який, зрештою, ще більше покращує можливості створення тестів.

Доопрацюванням в даній темі є те, що генерацію тестів буде вдосконалено функціоналом, який буде робити припущення про типи даних виходячи з тих чи інших даних. Від правильності цих припущень буде залежати успішність генерування тестових випадків.

У цьому дослідженні запропоновано новий підхід до роботи з динамічно типізованими мовами програмування в контексті автоматизованої генерації тестів. Цей метод базується на неконтрольованому виведенні типів, що комбінує статичний аналіз із новаторською технікою динамічного аналізу. Динамічний аналіз передбачає використання даних про виконання, отриманих зі згенерованих тестів. У процесі пошуку тестових випадків виконуються згенеровані тести, що дозволяє оцінити покриття коду. Таким чином, інформація про виконання доступна без додаткових зусиль. Це робить техніку динамічного аналізу особливо ефективною для визначення типів у рамках генерації тестів. Завдяки цій інформації можна перевірити точність статично визначених типів та відповідно їх коригувати. Це також дозволяє адаптувати ймовірності варіантів типів для конкретних змінних, підвищуючи точність і ефективність підходу.

Технології статичного аналізу використовуються до запуску коду. Статичний аналіз – це метод дослідження програмного забезпечення, який здійснюється без виконання коду. Основна мета статичного аналізу – знайти помилки, вразливості або проблеми в коді на ранніх етапах розробки. Це дає змогу підвищити якість коду, зменшити кількість дефектів і уникнути потенційних загроз безпеці.

Для виконання статичного аналізу потрібно детально проаналізувати вихідний код. Для цього існує достатня кількість технологій. Щоб отримати необхідну інформацію, увесь код перетворюється на абстрактне синтаксичне дерево. Його необхідно обійти, щоб знайти зв’язки та їх елементи. Крім того, усі визначені користувачем об’єкти витягуються з дерева для створення описів типів об’єктів.

Зібрана інформація, яку можна використовувати для визначення типів складається з:

- елементів;
- відношень (або зв’язків);
- типів визначених користувачем.

Елементи представляють частини тестованого коду, для яких потрібно вивести типи. Елемент може бути ідентифікатором змінної або літералом. Ідентифікатори – це іменовані посилання на змінні, функції та властивості. Літерали – це необроблені

значення, які можна присвоїти змінним або константам. Оскільки літерали безпосередньо представляють тип, нам не потрібен висновок про тип даних. У динамічно типізованих мовах програмування ідентифікатори не мають явних типів. Щоб дізнатися більше про тип ідентифікатора, є можливість використати контекст ідентифікатора, тобто зв'язки або відношення.

Відношення – це вирази та операції, які включають один або кілька елементів і описують їхнє використання та зв'язок з іншими елементами. Вони дозволяють отримати більше інформації про типи елементів. Наприклад, якщо змінній A присвоєно значення B, і B є логічним літералом, можна зробити висновок, що A також має бути логічного типу в цій частині коду. Таким чином, аналізуючи відношення у коді, визначаються їхні властивості та виконуються відповідні висновки. Інформація про типи елементів отримується з відношень, у яких вони беруть участь. Важливим аспектом елементів є їхня область видимості: вона визначає, де саме ідентифікатор доступний у коді.

Після вилучення всіх елементів, зв'язків і можливих типів із коду, створюється модель імовірнісного типу для кожного елемента. Це досить просто для елементів, які представляють літерали, оскільки тип можна безпосередньо вивести з типу літерала. Однак не кожен елемент у коді є літералом. На основі отриманих співвідношень необхідно зробити припущення про тип даних.

Окрім типів елементів, відносини також мають тип результату. Розглянемо, наприклад, відношення: [L > R, a, b]. У цьому випадку тип результату є логічним, незалежно від типів a і b. Без додаткового контексту це не дає великої користі. Проте часто є ймовірність зустрітись з вкладеними відносинами в коді. Наприклад, [L = R, c, d*], де d* є відношенням [L > R, a, b]. Оскільки результат d* є логічним, можливо зробити висновок, що «c» також має бути логічним значенням. Проте замість того, щоб прямо призначати тип «c» як логічне значення, тип «c» стає залежним від типу d*. Це робиться тому, що в деяких відносинах результат залежить від залучених елементів. Таким чином, роблячи типи елементів взаємозалежними, фактично створюється мережа ймовірностей типів.

Імовірності застосовуються на етапі пошуку для вибору аргументів відповідного типу. Для вибору аргументу необхідно обрати один із типів, визначених у моделі типів. Для цього дослідження було розроблено два режими. Перший режим називається режимом рангової вибірки, у якому завжди вибирається тип з найвищою ймовірністю. Другий режим називається режимом пропорційної вибірки, де тип вибирається випадковим чином відповідно до його ймовірності, тобто якщо тип має 50% ймовірність бути правильним, він має 50% шансів бути обраним.

Об'єкти в JavaScript відіграють ключову роль як фундаментальні будівельні блоки мови. Вони слугують контейнерами для пар "ключ-значення", що дозволяє ефективно організовувати та зберігати дані. У JavaScript майже будь-який елемент, навіть примітивні типи даних, такі як числа або булеві значення, можна представити у вигляді об'єкта. Наприклад, масиви є особливим типом об'єктів, де індекси виступають у ролі ключів. Сучасний JavaScript також підтримує концепції об'єктно-орієнтованого програмування, дозволяючи розробникам створювати класи та інтерфейси для більш структурованого та зрозумілого коду.

Для визначення, які елементи тестованого проекту є об’єктами, проводиться аналіз їхньої взаємодії через доступ до властивостей (Property Accessor). Це передбачає порівняння доступних властивостей елемента з описами типів об’єктів. Якщо елемент бере участь у взаємодії з властивостями та є перетин між його властивостями та характеристиками описаного типу об’єкта, то цей тип розглядається як потенційний тип для даного елемента.

Динамічний аналіз виконується вже після запуску коду. У процесі такого аналізу перехоплюються та обробляються винятки, які виникають під час виконання програми. Для реалізації динамічного аналізу можуть використовуватися різні технології, зокрема інструментування коду. Наприклад, якщо у нас є певний тестовий сценарій, його можна модифікувати, попередньо проаналізувавши структуру коду (гілки, цикли, рекурсії тощо) та додавши виклики стороннього функціоналу. Найпростішим способом такої модифікації є впровадження логування, яке дозволяє програмісту отримувати інформацію про виконання програми на основі зібраних під час її роботи даних.

Окрім показників, що свідчать про належність елемента до певного типу, кожен елемент має також оцінки виконання. Ці оцінки використовуються для оновлення моделей типів на основі результатів виконання тестів. Якщо під час виконання тестового випадку виникає виняток TypeError, тоді відбувається аналіз повідомлення про помилку або трасування стека, щоб виявити імена змінних, використаних у згенерованому тесті. У разі знаходження відповідної змінної, тип, який використовувався для неї, отримує негативну оцінку виконання.

В даному дослідженні за основу буде взято пошуковий підхід до генерації тестів. Випадковий пошук (Random Search) – це простий, але ефективний підхід для генерації тестових випадків. Його суть полягає в генерації випадкових вхідних даних у межах заданих обмежень та подальшому аналізі результатів. Цей метод особливо корисний для тестування систем із великим простором можливих вхідних значень або в ситуаціях, коли немає чіткого розуміння про найкритичніші точки тестування.

Переваги випадкового пошуку:

- простота реалізації;
- може знайти помилки, які складно передбачити або змоделювати вручну;
- ефективний для пошуку нетипових сценаріїв;
- не потребує складних алгоритмів або великого обсягу знань.

Проте, створення тест-кейсів за допомогою пошукових алгоритмів передбачає використання таких методів, як генетичні алгоритми, імітація відпалу тощо. Це дозволяє автоматично генерувати або оптимізувати тестові випадки на основі визначених цілей.

Для цього необхідне розуміння предметної області. Потрібно визначити ціль тестових випадків (наприклад, максимізація покриття коду, знаходження граничних умов, виявлення дефектів). Визначити вхідні дані, тобто з'ясувати параметри або вхідні значення тестів (наприклад, текстові поля, числові значення, конфігурації системи).

Досить важливим методи, які застосовують в пошуковій генерації тестів є генетичні алгоритми. Генетичні алгоритми – це один із методів еволюційного обчислення, натхненний процесами природного відбору і генетики. Вони

використовуються для розв’язання оптимізаційних задач, коли класичні алгоритми можуть бути неефективними або складними для реалізації.

Основні ідеї та концепції:

– популяція: набір можливих розв'язків задачі (хромосом), які представлені у вигляді рядків (наприклад, бінарних, числових або символічних);

– ген: елементарна частина хромосоми, яка представляє певний параметр або змінну;

– фітнес-функція: функція, яка оцінює якість кожного розв'язку (хромосоми) щодо поставленої задачі;

– селекція: процес вибору найкращих хромосом для створення нового покоління, який імітує природний відбір;

– кросовер (схрещування): об'єднання двох хромосом для створення нових хромосом-нащадків;

– мутація: випадкові зміни в хромосомах для збереження різноманіття популяції.

Основні кроки алгоритму:

– ініціалізація: створення початкової популяції хромосом (випадковим чином або з використанням евристик);

– оцінка: обчислення значень фітнес-функції для кожної хромосоми;

– відбір: вибір хромосом для створення наступного покоління (методи: рулетка, турнірний відбір тощо);

– схрещування та мутація: створення нових хромосом із використанням кросовера та мутації;

– заміна: формування нового покоління шляхом заміни старих хромосом;

– зупинка: алгоритм завершується, коли досягнуто певної кількості поколінь, часу або заданого рівня оптимізації.

Переваги:

– придатність для нелінійних, складних, багатовимірних задач;

– паралельний пошук в просторі рішень;

– гнучкість: може працювати з будь-якою фітнес-функцією.

Недоліки:

– можливість застрягнути в локальних оптимумах;

– часом високі обчислювальні витрати;

– вимагає ретельного налаштування параметрів (розмір популяції, імовірність мутації, кількість поколінь).

В даному дослідженні не буде детального пояснення про реалізацію генетичних алгоритмів, проте згадати за них варто було, оскільки вони є важливою частиною для інструментів генерації тестів базованих на пошуку. Генетичні алгоритми широко використовуються для пошукової генерації тестів, особливо в автоматизованому тестуванні програмного забезпечення. Цей підхід дозволяє ефективно знаходити тестові випадки, що покривають важливі частини програмного коду та оптимізувати певні критерії, такі як покриття, продуктивність або виявлення помилок. Генетичні алгоритми використовують принципи еволюційного пошуку, де початкові тестові випадки (популяція) покращуються шляхом кросоверів, мутацій і селекції, щоб досягти оптимальних результатів.

1.2 Аналіз сучасних досліджень.

На сьогодні автоматична генерація тестів є актуальною темою в індустрії програмного забезпечення. Згідно з дослідженнями, більше ніж 70% помилок у програмному забезпеченні можна виявити на ранніх стадіях за допомогою тестування. Проте, проблема неконтрольованого введення типів, особливо в мовах з динамічною типізацією, залишається відкритою. Відсутність чітких типів у таких мовах може призвести до помилок при генерації тестів і неправильного їх виконання.

Динамічні мови програмування створюють новий виклик для автоматизованих генерацій тестів. Оскільки інформація про тип є недоступною, система має вгадати, які типи використовувались. Вгадування правильного типу додатково до генерації самих тестів, щоб збільшити покриття коду тестами, різко збільшує простір в якому

необхідно шукати дані для правильного результату. Дані дослідження сконцентровані на генерації тестів для JavaScript, що є досить непростим завданням. На цей час існує інструмент під назвою Pynguin, який генерує тести для Python. Даний інструмент для генерації тестів використовує необов’язкові анотації типів, оскільки Python має можливість писати анотації типів. Дана робота показує, що знання типів є дуже важливим компонентом. В стандартному JavaScript немає анотацій типів, а це означає, що генерація тестів має зробити припущення про те які типи використані в коді.

Існують різні шляхи для розв'язування проблеми неконтрольованих типів. Для прикладу, можна зробити припущення опираючись на контексті в якому використовується змінна. Неконтрольовані типи було досліджено раніше, щоб перетворити код без типізації в типізований [3, 4]. Фрагмент програмного коду 1.1. показує такий приклад перетворення коду. Функція розрахунку мінімального значення приймає два параметри та повертає одне значення. У функції eval_min не вказано типів вхідних та вихідних значень. В наступній функції eval_min_typed вже присутні анотації типів, які точно вказують на те що функція приймає два числа та повертає також число.

Фрагмент програмного коду 1.1:

```
function eval_min(a, b) {
    if (a < b) {
        return a;
    }

    return b;
}

function eval_min_typed(a: number, b: number): number {
    if (a < b) {
        return a;
    }

    return b;
}
```

Конвертація коду без типів у типізований код є досить важливою темою, оскільки великі та масштабні проекти краще підтримуються розробниками саме тоді коли використовується типізована мова програмування. Яскравим прикладом є мова програмування TypeScript, яка додала статичні типи, можливість створювати класи та інтерфейси (як у традиційних об’єктно-орієнтованих мовах програмування), а також підтримку підключення модулів. Більшість популярних JavaScript фреймворків підтримують TypeScript та значна частина модулів є написаними за допомогою цієї мови програмування.

Деякі з досліджень опираються на використання статичного аналізу та глибокого навчання [3, 5] для вгадування типу даних. Машинне навчання є досить дорогим та непростим процесом та потребує постійного перенавчання моделі. Це досить не бажаний процес з погляду генерації тестів базованих на пошуку, оскільки нам необхідно згенерувати багато тестів в короткий проміжок часу. Також варто зазначити те що код не виконується при статичному аналізі, а лише аналізується, тому варто також використовувати динамічний аналіз при якому програмний код виконується, з чого можна отримати додатково інформацію.

На сьогодні тема автоматичної генерації тестів є досить досліджуваною. Сучасні дослідження підкреслюють значення неконтрольованого введення типу як одного з ключових напрямків у галузі автоматичного тестування. Розробка нових інструментів та алгоритмів дозволяє значно підвищити надійність програмного забезпечення, особливо в контексті мов із суворими системами типів.

Серед основних досліджень в даній сфері можна виділити декілька напрямів. Перший напрямок – це автоматична генерація тестів на основі типів. Використання типів для генерації вхідних даних дає змогу створювати більш релевантні тестові випадки. Наприклад, мови програмування з системами типів (такі як Haskell, Scala) дозволяють використовувати інформацію про типи для визначення діапазону можливих вхідних даних. Дослідники працюють над розробкою технік, які дозволяють інструментам для автоматичного тестування (наприклад, QuickCheck) генерувати дані відповідно до типів вхідних змінних, що значно підвищує ефективність і точність тестування.

Другий напрямок – це типізація та семантична валідація. Неконтрольоване введення типу може перевіряти, наскільки добре система справляється з нестандартними або несподіваними типами даних. Наприклад, можна виявити помилки, що виникають через неправильне трактування типів або через надмірно суворі обмеження на типи даних.

Дослідники вивчають, як використовувати типові системи для валідації різних властивостей програм (наприклад, інваріантів) у процесі тестування.

Третій напрямок – Fuzz-тестування та пошук вразливостей. Неконтрольоване введення даних, зокрема випадкове fuzzing типів, активно використовується для пошуку помилок, пов'язаних із межами допустимих значень, неочікуваними типами даних або неявними припущеннями щодо типів.

Дослідження фокусуються на створенні інструментів, які використовують системи типів для формування критичних вхідних даних з метою виявлення вразливостей, що можуть бути використані зловмисниками.

Ще одним напрямком є покращення автоматизованих інструментів. Сучасні дослідження спрямовані на розширення можливостей наявних інструментів тестування. Наприклад, нові моделі тестування розширюють можливості інструментів, таких як QuickCheck, для складніших сценаріїв тестування. Використання AI та машинного навчання для вдосконалення алгоритмів автоматичного підбору типів і аналізу результатів тестування.

Сучасні дослідження, що стосуються неконтрольованого введення типу (fuzzing) для мов програмування під час автоматичної генерації тестів, сфокусовані на інтеграції штучного інтелекту для підвищення ефективності та точності тестування.

Одним із ключових напрямків є використання машинного навчання для генерації кращих вхідних даних. Наприклад, дослідження показали, що fuzz-тестування може бути вдосконалено за допомогою нейронних мереж, які генерують вхідні дані на основі минулих тестових випадків, таким чином значно підвищуючи шанси на виявлення прихованих помилок. Один із таких інструментів – FuzzAug, який використовує fuzzing як метод для доповнення даних і навчання нейромереж.

Інновації, такі як IntelliGen [6], дозволяють автоматизувати процес створення драйверів для fuzz-тестування. IntelliGen аналізує вихідний код і знаходить функції з найвищою пріоритетністю на основі вразливостей (наприклад, операцій з небезпечними вказівниками або пам'яттю). Це значно скорочує ручну роботу і забезпечує краще покриття тестування. IntelliGen використовує штучний інтелект для автоматизації вибору точок входу в програму для тестування, оптимізуючи процес шляхом пріоритезації вразливих функцій.

Детальніше зупинимось на досить перспективному підході генерації тестів, а саме пошуковому підходу.

Пошукова генерація тестових випадків (SBTCG, англ. search-based test case generation) – це метод тестування програмного забезпечення, який використовує алгоритми пошуку для автоматичної генерації тестових випадків. Цей підхід особливо корисний для створення складних тестів, які важко або довго створювати вручну.

Даний підхід має декілька основних концепцій та інші характеристики, які далі будуть розглянуті.

Алгоритми пошуку. Використовуються алгоритми пошуку, такі як генетичні алгоритми, імітація відпалу або рій частинок для дослідження великого простору потенційних тестових випадків. Метою алгоритму є знаходження випадків, які задовольняють певні критерії або оптимізують певні властивості, такі як покриття коду або виявлення помилок.

Функція пристосованості (фітнес-функція): У SBTCG визначається фітнес-функція, яка оцінює, наскільки кожен тестовий випадок відповідає цілям тестування (наприклад, максимізація покриття коду, досягнення граничних умов або виявлення помилок). Фітнес-функція допомагає алгоритму вирішити, які тестові випадки зберегти або відкинути.

Подання тестових випадків: Кожне потенційне рішення (тестовий випадок) подається у формі, яка дозволяє алгоритму маніпулювати ним. Наприклад, у генетичному алгоритмі тестові випадки можуть бути подані як хромосоми, де певні гени відповідають значенням вхідних даних.

Сценарії застосування: SBTCG часто використовують для тестування білого ящика для максимізації покриття коду, тестування на витривалість, аналізу граничних значень і тестування на основі шляхів. Його також використовують для тестування на основі моделей, коли модель можна подати у формі, яка доступна для роботи з алгоритмами пошуку.

Етапи пошукової генерації тестових випадків:

– визначення цілей і обмежень тестування. Потрібно визначити, чого мають досягати тести (наприклад, високе покриття гілок або виявлення помилок) і встановити обмеження (наприклад, часові ліміти);

– вибір алгоритму пошуку. Підбирається відповідний алгоритм залежно від складності програмного забезпечення та цілей тестування. Генетичні алгоритми є популярними для SBTCG, але інші алгоритми можуть бути більш придатними залежно від випадку;

– розробка фітнес-функції. Потрібно визначити фітнес-функцію, яка оцінює ефективність кожного тестового випадку у досягненні цілей тестування. Ця функція буде основою для відбору в межах алгоритму пошуку;

– генерація початкових тестових випадків. Початок роботи з набором випадкових тестових випадків або вручну створеними початковими тестами;

– застосування алгоритму пошуку. Використання обраного алгоритму для розвитку тестових випадків протягом декількох ітерацій, здійснюючи відбір, мутацію та комбінування їх для підвищення ефективності;

– валідація та виконання тестових випадків. Після генерації тестових випадків їх виконують на програмному забезпеченні. Перевіряють, чи відповідають вони визначеним цілям тестування, таким як виявлення помилок або досягнення певного покриття;

– оцінка результатів. Аналіз отриманих тестових випадків і їх ефективності у досягненні цілей. Якщо потрібно, алгоритм або фітнес-функцію вдосконалюють.

Переваги SBTCG:

– ефективність. Автоматизація генерації тестових випадків, що заощаджує час і ресурси;

– високе покриття. Може забезпечити краще покриття коду, особливо у складних застосунках, досліджуючи комбінації, які не завжди інтуїтивно очевидні;

– адаптивність. Добре працює для різних типів цілей тестування (наприклад, функціональне, на витривалість).

Також використання пошукової генерації тестів ставить наступні виклики:

– визначення ефективної фітнес-функції. Створення фітнес-функції, яка точно відображає цілі тестування, може бути складним завданням;

– обчислювальні витрати. Деякі алгоритми пошуку, особливо ті, що працюють із великими просторами, можуть бути затратними для обчислень;

– локальні оптимуми. Алгоритми пошуку можуть застрягнути в локальних оптимумах, що призводить до менш ефективних тестових випадків.

Існує кілька автоматизованих інструментів для створення тестів, які працюють з динамічно типізованими мовами програмування:

– Artemis & SymJS, один з інструментів створення тестів для JavaScript. Він використовує алгоритм довільного тестування зі зворотним зв’язком для тестування веб-програм JavaScript;

– JSeft, інструмент генерації тестів JavaScript, зосереджений на створенні тестів на основі подій і модульних тестів функціонального рівня;

– Jalangi, фреймворк для динамічного аналізу JavaScript. Хоча це не інструмент для створення тестових випадків, він виконує concolic (конкретне та символічне) тестування;

– Pynquin, автоматизований фреймворк для створення модульних тестів для Python.

Виведення типів є темою дослідження вже досить давно. Коротко переглянемо найсучасніші методи виведення типу:

– JSNice, працює, розбираючи вхідну програму та перетворюючи її на мережу залежностей, що пов’язує невідомі з відомими властивостями. Одним із недоліків JSNice є те, що він може передбачати лише базові типи JavaScript. Це означає, що він не може стверджувати, що певна змінна належить до типу, визначеного користувачем;

– DeepTyper, модель глибокого навчання, створена для надання пропозицій типів для JavaScript [3]. Він складається з GRU (Gated Recurrent Units), типу нейронної мережі. DeepTyper навчається за допомогою великого набору проектів TypeScript від GitHub. Ці проекти використовуються для вивчення типів змінних у певних контекстах;

– NL2Type, подібно до моделі глибокого навчання DeepTyper [5]. Nl2Type складається з мережі LSTM (довго короткочасної пам’яті), типу нейронної мережі. На відміну від DeepTyper, NL2Type використовує коментарі в поєднанні з кодом як вхідні дані для прогнозування типів;

– LambdaNet, використовує комбінацію логічних обмежень і контекстних підказок [27]. Подібно до JSNice, він використовує підказки та логічні обмеження для побудови графіка залежностей. Однак, на відміну від JSNice, він використовує GNN (Graph Neural Network) для визначення типів;

– Type4Py, він зосереджений на Python і розв'язує проблему обмеженого словникового запасу, використовуючи глибоку стратегію вивчення подібності. Type4Py перевершує інші сучасні підходи, такі як Typilus, який також використовує глибоке навчання подібності [28].

Більшість пов’язаних робіт із виведення типу використовує машинне навчання для виконання обробки природної мови. Підходи не можуть вивести типи, визначені у вихідному коді розробником.

Сучасні дослідження підкреслюють значення неконтрольованого введення типу як одного з ключових напрямків у галузі автоматичного тестування. Розробка нових інструментів та алгоритмів дозволяє значно підвищити надійність програмного забезпечення, особливо в контексті мов із суворими системами типів.

1.3 Постановка задачі

Оскільки дослідження в даній темі є досить різноманітні, які використовують безліч підходів до розв'язання задач, необхідно сформувати ключові моменти над якими варто провести роботу, щоб досягти успіху у своєму дослідженні. Нагадуємо

що метою дослідження є автоматична генерація тестів з використанням неконтрольованих типів на прикладі мови програмування JavaScript.

Використання неконтрольованого введення типів дозволяє створювати крайні випадки, які можуть бути пропущені під час звичайного тестування. Це допомагає знаходити помилки, що виникають через рідкісні або несподівані комбінації вхідних даних.

Процес генерації тестових даних повинен бути повністю автоматизованим, щоб мінімізувати ручне втручання. Це важливо для великих програмних проектів, де кількість можливих комбінацій вхідних даних занадто велика для ручного тестування.

В даному дослідженні буде використаний підхід, який комбінує статичний аналіз та динамічний аналіз, а також базується на пошуковій генерації тестів.

Технологія динамічного аналізу складається з використання інформації, яку можливо отримати з виконаних згенерованих тестів. Під час процесу пошуку відбувається запуск згенерованих тестів, щоб розрахувати отримане покриття. Це є досить правильним підходом для виведення неконтрольованих типів. Інформація про виконаного коду дозволяє оцінити точність статичного аналізу та відповідно регулювати припущення про типи даних для конкретних змінних.

В цілому можна виділити декілька ключових задач даного дослідження:

– дослідити вплив виведення типу на ефективність створення тестових випадків для JavaScript;

– виміряти вплив на продуктивність використання інформації про виконання для підвищення точності виведення типу для JavaScript;

– зробити оцінку того, чи може використання виведення типу бути недорогим процесом.

Вплив продуктивності можна виміряти, дивлячись на досягнуте структурне покриття і можливості виявлення помилок створених тестів. Оцінюємо, чи може процес бути недорогим за часом, якщо порівняти час, який використовував процес виведення типу загальний час, який витрачається на процес генерації тесту. Загальна гіпотеза полягає в тому, що інформація про виконання згенерованих тестів дозволяє

більш точно визначити тип, крім того, вдосконалення виведених типів покращує можливості генерації тестів. Це може створити цикл зворотного зв’язку, який, зрештою, покращує можливість генерації тестів ще більше.

Метою дослідження є проектування та імплементація методу виведення типу під час автоматичної генерації тестів та визначення їх ефективності.

Також завдання ставить певні виклики серед яких:

– непередбачуваність результатів введення неконтрольованих типів;

– забезпечення правильного балансу між кількістю згенерованих тестів і їх ефективністю;

– оптимізація ресурсів для мінімізації часу виконання тестів і максимальної продуктивності.

Використання неконтрольованого введення типів для мов програмування при автоматичній генерації тестів є важливою, але складною темою. Хоча цей підхід має багато переваг, таких як економія часу та ресурсів, його реалізація вимагає обережності та ретельної валідації для уникнення проблем з продуктивністю та безпекою. Завдяки правильному підходу, поєднанню автоматизації з ручною перевіркою та впровадженню інструментів для безпеки типів, можна значно підвищити якість тестування й забезпечити стабільну роботу програмного.

1.4 Висновки

В даному розділі було частково розглянуто тему автоматичної генерації тестів та відповідний тематичний матеріал.

Серед розглянутого матеріалу варто визначити ключові аспекти:

– загальна інформація про автоматичну генерацію тестів;

– пошукові алгоритми генерації тестів;

– виведення типів даних;

– статичний аналіз;

– динамічний аналіз.

Даний розділ був оглядовим, було розглянуто напрямки автоматичного тестування, дізнались що ця тема не є повністю вивченою, а також підходити до розв'язання цієї задачі. Було проаналізованого певні інструменти, корисні для нашого дослідження, а саме:

– Artemis & SymJS;

– JSeft;

– Jalangi;

– Pynquin.

Наступними були проаналізовані інструменти для виведенні типів, а саме:

– DeepTyper, модель глибокого навчання, створена для надання пропозицій типів для JavaScript;

– NL2Type, подібний до моделі глибокого навчання DeepTyper [5];

– LambdaNet, використовує комбінацію логічних обмежень і контекстних підказок, а також використовує підказки та логічні обмеження для побудови графіка залежностей;

– Type4Py, він зосереджений на Python і розв'язує проблему обмеженого словникового запасу, використовуючи глибоку стратегію вивчення подібності.

Загалом сформовано декілька ключових завдань даного дослідження, а саме:

– визначення впливу виведення типу на ефективність створення тестових випадків для JavaScript;

– визначення впливу на продуктивність використання інформації про виконання для підвищення точності виведення типу для JavaScript;

– оцінка того, чи може використання виведення типу бути недорогим процесом.

Розглянута інформація є досить складною, але в підсумку, завершальним етапом цього розділу стало формулювання задачі, в рамках якої визначено напрямок подальших дій. Було обрано пошуковий підхід для генерації тестів, оскільки він не вимагає використання машинного навчання та інших ресурсовитратних технологій. Дане дослідження повинно вдосконалити його за допомогою статичного та динамічного аналізу та порівняти, наскільки значуща різниця між стандартною

генерацією тестів із випадковими типами даних і типами, що були виведені та проаналізовані за допомогою додаткових модулів. Окрім того, сподіваємось, що цей підхід дозволить швидко генерувати велику кількість тестів.

2 КОНЦЕПЦІЇ, МОДЕЛІ, ТА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ

2.1 Концепція автоматичної генерації тестів

Автоматична генерація тестів – це процес створення тестових сценаріїв або тестових даних за допомогою автоматизованих інструментів або алгоритмів, з метою перевірки працездатності програмного забезпечення. Ця концепція дозволяє зекономити час, знизити ризик людських помилок і підвищити покриття тестами.

Тестування програмного забезпечення є важливим етапом розробки програмного забезпечення [7]. Це дозволяє розробнику переконатись що ПЗ працює належним чином [8, 9]. Тестування є досить різноманітним. Базовий приклад тестування це модульне тестування (unit tests). Модульне тестування має за мету перевірити те як працюють модулі програми по окремості. Модулем програми зазвичай є найменша частина, яку можна протестувати, наприклад функція, процедура або метод. Тестування на рівні інтеграції по суті має на меті перевірити кілька функціональних можливостей у поєднанні, щоб оцінити, чи вони працюють разом правильно. Тестування на системному рівні спрямоване на перевірку програмного забезпечення в цілому. Щоб забезпечити, успішну роботу програмного забезпечення у майбутньому, розробники створюють набори тестів, які можуть бути запускати, коли це необхідно, щоб перевірити базу коду. Постійно запускаючи такі набори тестів після зміни коду називається регресійним тестуванням.

Покриття коду.

Визначити чи достатньо наборів тестів для виявлення несправностей/помилок, може бути складно. Як орієнтир, розробниками та тестувальниками часто використовуються метрики покриття коду. Покриття коду дозволяє бачити, які частини коду досягаються під час виконання тестів. У певних дослідженнях виявлено помірну або сильну кореляцію між ефективністю набору тестів та покриттям коду [10]. Ефективність набору тестів визначається його здатністю знаходити реальні помилки.

Існує декілька типів метрик покриття коду. Кожен з них фокусується на різних аспектах коду. Наприклад, покриття операторів визначає, які оператори

покриваються під час виконання певного програмного забезпечення. Однак код – це не просто фрагмент тексту, який виконується лінійно оператор за оператором. Замість цього код можна розглядати як графік, де кожна умова створює істинну та хибну «гілки». Покриття гілок дає уявлення про те, які гілки в графі програмного забезпечення охоплено. Між цими двома є відношення підрахунку, тобто 100% покриття гілок автоматично означає 100% покриття операторів.

З цієї причини доцільно використовувати метрику охоплення найвищу в ієрархії. Однак найвищий показник називається покриттям шляху. Покриття шляху вимірює, скільки можливих шляхів через «граф коду» охоплено. Кількість можливих шляхів коду зростає експоненціально зі збільшенням розміру програми [11]. Це робить надзвичайно дорогим охоплення всіх можливих шляхів коду та викликає сумніви, чи варто це робити. З цієї причини покриття гілок часто використовується розробниками програмного забезпечення.

Тестування в поточній індустрії.

Сучасна промислова практика покладається на створені вручну тести, що вимагає від розробників програмного забезпечення великої кількості часу та зусиль [4, 7, 12]. Таким чином, дослідники запропонували рішення для автоматизованої генерації тестів з 1970-х років [14]. Це призвело до появи різноманітних методів та інструментів, які дозволяють автоматично генерувати тести за допомогою програмного забезпечення. Ці методи включають символічне виконання [15, 16], конкретне і символічне виконання (Concolic) [17], випадкове тестування [13] і тестування програмного забезпечення на основі пошуку [4]. Ці методи будуть пояснені більш детально далі.

Випадкове тестування та Fuzzing.

Випадкове тестування є найпростішим методом. Інструменти, які використовують цю техніку, випадково генерують вхідні дані для функцій/програм. Ці виклики функцій потім виконуються та перевіряються на збої/виключення. Випадкове тестування часто не передбачає створення фактичного тестового випадку, натомість вхідні дані, які викликають збої/виключення, зберігаються, щоб пізніше показати їх кінцевому користувачеві. Оскільки випадкове тестування є суто

випадковим, воно не має жодних вказівок щодо досягнення високого охоплення чи високоякісних тестових випадків.

Символічне виконання та Concolic виконання.

Символічне виконання в контексті тестування програмного забезпечення має на меті виявити вимоги для досягнення певної точки в програмному забезпеченні. Точніше, яка комбінація вхідних даних призводить до виконання певних частин програмного забезпечення. Символьне виконання виконується за допомогою символьних значень для вхідних даних. Таким чином можливо створити обмеження для досягнення кожної умовної гілки в термінах цих символів. Concolic тестування дуже схоже на символічне виконання. Однак воно використовує конкретні значення замість символів для створення конкретних вхідних даних для максимального покриття коду.

Тестування програмного забезпечення на основі пошуку (SBST).

Методи тестування програмного забезпечення на основі пошуку часто містять випадковий компонент, подібний до методів випадкового тестування. Однак методи SBST також включають цілі. Часто ці цілі є галузевими. Цілі розгалужень – це, по суті, нерозкриті розгалуження в коді. Мета алгоритму полягає в тому, щоб охопити ці цілі галузі. Можливо розрахувати рівень наближення та відстань розгалуження, щоб керувати алгоритмом. Рівень підходу дорівнює кількості розгалужень між найближчою охопленою гілкою та метою гілки. Відстань гілки дорівнює значенням змінної, обчисленим за умовним виразом цільової гілки.

Методи SBST успішно використовувалися в літературі для автоматичної генерації тестів на різних рівнях тестування [4], таких як модуль [18], інтеграція [19] і тестування на системному рівні [20].

2.2 Динамічно типізовані мови програмування.

В статично типізованих мовах програмування, перевірка типів відбувається під час компіляції. В динамічно типізованих мовах програмування навпаки, перевірка

типу відбувається під час виконання. Це означає, що правильність типів не перевіряється заздалегідь. Натомість типи перевіряються лише під час виконання програми. Типовими прикладами динамічних мов програмування є JavaScript, Python, Ruby, PHP, Lua та Perl. Python і JavaScript вже кілька років є одними з найпопулярніших мов програмування [2, 3]. На рисунку 1.1 у розділі 1 показано різницю між динамічно типізованою мовою програмування і статично типізованою. Статична типізація вимагає від розробника явного визначення типу даних кожної змінної під час оголошення змінної. Сигнатури функцій також повинні явно вказувати тип повернення. У динамічній типізації це не потрібно. Популярність динамічно типізованих мов можна пояснити гнучкістю, яку вони пропонують розробникам [21]. Така гнучкість типів дозволяє розробникам швидко писати фрагменти коду без необхідності підтримувати складну систему типів. Одним із їх недоліків є те, що помилки типу виявляються лише під час виконання.

JavaScript.

Як згадувалося, прикладами динамічно типізованих мов є JavaScript і Python. Однак, на відміну від Python, JavaScript є мовою зі слабкою типізацією. У мові зі слабким типом змінні не прив’язані до певного типу. Іншими словами, мови зі слабкою типізацією неявно здійснюють перетворення між непов’язаними типами. У слабо типізованій мові дозволено, наприклад, додавати разом змінні різних типів. У строго типізованій мові це було б неприпустимо. Наприклад, наступний вираз «c=”abc”+1» не дозволяється в Python, оскільки змішується два різних типи в одному виразі. У JavaScript це цілком нормально, оскільки це слабо типізована мова програмування.

2.3 Вивід типів

Виведення типу – це здатність автоматично виводити тип виразу. Існує багато методів для визначення типу. Ці методи можна класифікувати як статичний вивід типу та динамічний вивід типу.

Статичний вивід типу використовує статичний аналіз для отримання інформації з вихідного коду. У літературі в основному використовуються три методи. Такими методами є обробка логічних обмежень, обробка контекстних підказок і обробка природної мови.

Логічні обмеження

По суті, логічні обмеження - це вирази у вихідному коді, які обмежують типи залучених змінних. Прикладом такого логічного обмеження є присвоєння. Якщо, наприклад, змінній присвоєно логічний літерал, зрозуміло, що змінна має логічний тип даних у цьому місці коду.

Обробка контекстних підказок.

Серед розробників заведено містити тип змінної в назві змінної, тобто, якщо змінна типу «tree» містить генеалогічне дерево, змінна може мати назву «familyTree». На відміну від обробки природної мови, асоціація між типами змінних і назвами змінних не вивчається. Натомість вони засновані на входженні назви типу в назву змінної. Це контекстна підказка. Ще одна контекстна підказка – це використання змінної. Змінні можуть брати участь у кількох операціях і виразах. Ці операції та вирази можуть дати розуміння того, яким може бути тип змінної.

Обробка природної мови.

Обробка природної мови (ОПМ) часто містить навчання пов’язувати весь контекст навколо змінної з типом змінної. Для ОПМ тип змінної не обов’язково вказувати в контексті змінної. Натомість тип визначається на основі того, що модель ОПМ навчилася асоціювати з контекстом змінної. Наприклад, слово «count» часто використовується в іменах змінних числового типу. Модель ОПМ навчиться пов’язувати «лічбу» з числовим типом. Іншим прикладом є обробка коментарів, залишених розробниками, щоб дізнатися про тип змінної.

У цьому дослідженні метод ОПМ не буде розглядався через те що даний метод певною мірою потребує машинного навчання, а це є дуже складним та дорогим процесом, який не бажано використовувати для автоматичної генерації тестів, оскільки це потребує не менше ресурсів ніж написання тестів вручну.

Окрім статичного виведення типу, існує також динамічне виведення типу. Динамічне виведення типу використовує динамічний аналіз виконання програми для отримання інформації про типи змінних. Динамічний аналіз можна використовувати, наприклад, для визначення статичних типів, використовуючи інформацію, зібрану з динамічних прогонів [22]. В іншому дослідженні динамічний аналіз використовується для спеціалізації типу [23].

Вивід типів є темою дослідження протягом досить тривалого часу [24, 25]. Далі в розділі коротко висвітлено історію галузі та обговорено найсучасніші методи виведення типу.

JSNice.

Одне з найбільш ранніх досліджень визначення типів для JavaScript було опубліковано у 2015 році [26]. Дослідження представляє масштабований механізм прогнозування під назвою JSNice. Механізм спрямований на передбачення імен ідентифікаторів і типів анотацій змінних. JSNice працює, аналізуючи вхідну програму та перетворюючи її на мережу залежностей, що пов’язує невідомі з відомими властивостями. На основі навчальних даних ця мережа залежностей потім використовується для визначення типів і імен невідомих властивостей. Дослідження зосереджено на загальній проблемі виведення властивостей програм і, таким чином, служить основою для подальших досліджень виведення типу. Одним із недоліків JSNice є те, що він може передбачати лише базові типи JavaScript. Це означає, що він не може стверджувати, що певна змінна належить до типу, визначеного користувачем. Інша проблема з JSNice полягає в тому, що він не може розглядати широкий контекст у програмі чи функції, тобто він не може диференціювати змінні в різних контекстах з однаковими іменами. JSNice перевіряє згенерований код, запускаючи Google Closure Compiler, інструмент, який перевіряє типи JavaScript із додатковими анотаціями типу. Однак це не надійно. Компілятор Closure може лише перевірити, чи можливий заданий тип, а не чи це тип, який задумав розробник.

DeepTyper.

DeepTyper – це модель глибокого навчання, створена для надання пропозицій типу для JavaScript [3]. Він складається з GRU (Gated Recurrent Units), типу

нейронної мережі. DeepTyper навчається за допомогою великого набору проектів TypeScript від GitHub. Ці проекти використовуються для вивчення типів змінних у певних контекстах. DeepTyper працює подібно до JSNice. Також зазначається, що поєднання двох засобів забезпечує значно кращі результати. У цьому гібридному режимі JSNice спочатку звертається до певного типу. Якщо JSNice не має відповіді, використовується DeepTyper. Автор приписує результати гібридного режиму двом підходам, які надзвичайно доповнюють один одного. JSNice майже завжди правильний щодо передбаченого типу, але часто невизначений і не надає тип. З іншого боку, DeepTyper робить більше прогнозів, але прогнози частіше бувають неправильними. Це поняття показує, що коли методи, які використовує JSNice, дають невизначений результат, деякі додаткові механізми передбачення можуть допомогти у визначенні типу. DeepTyper фокусується на 11000 найбільш часто використовуваних типах. Це обмежує інструмент у виведенні типів, визначених користувачем, тобто типів, визначених у вихідному коді, для яких необхідно вивести типи. Щоб увімкнути DeepTyper для вивчення визначених користувачем типів, DeepTyper слід перенавчати за допомогою вихідного коду, де визначено ці типи.

NL2Type.

NL2Type подібний до моделі глибокого навчання DeepTyper [5]. Nl2Type складається з мережі LSTM (Long Short Term Memory), тип нейронної мережі. На відміну від DeepTyper, NL2Type використовує коментарі в поєднанні з кодом як вхідні дані для прогнозування типів. Автори показали, що він значно перевершив як JSNice, так і DeepTyper. Подібно до негібридного підходу DeepTyper, NL2Type є підходом обробки природної мови. Як і DeepTyper, NL2Type здатний передбачати лише визначений набір типів. NL2Type оптимально працює для 1000 типів.

LambdaNet.

LambdaNet використовує комбінацію логічних обмежень і контекстних підказок [27]. Подібно до JSNice, він використовує підказки та логічні обмеження для побудови графіка залежностей. Однак, на відміну від JSNice, він використовує GNN (Graph Neural Network) для визначення типів. Автори LambdaNet відзначають,

що їх підхід значно перевершує DeepTyper. LambdaNet може передбачити 100 різних типів.

Type4Py.

Type4Py є одним із найсучасніших у виведенні типів. Він зосереджений на Python і розв'язує проблему обмеженого словника типів, використовуючи глибоку стратегію навчання подібності. Type4Py перевершує інші сучасні підходи, такі як Typilus, який також використовує глибоке навчання подібності [28]. Type4Py розглядає контекстні та природні підказки типів, надаючи ідентифікатори, контекст коду та видимі підказки типів як функції, з яких він вивчає асоціації типів. Хоча Type4Py може обробляти необмежену кількість типів, тобто нескінченно великий словник типів, він не може робити прогнози для типів, які лежать за межами попередньо визначених кластерів типів.

Більшість пов’язаних робіт із виведення типу використовує машинне навчання для виконання оброки природної мови. Підходи не можуть вивести типи, визначені у вихідному коді розробником. Іншими словами, словниковий запас їх типів обмежений наданими навчальними даними. Крім того, описані методи зосереджені лише на статичному аналізі.

2.4 Моделі та методи удосконалення генерації тестів.

Існує кілька автоматизованих інструментів для створення тестів, які працюють з динамічно типізованими мовами. У цьому розділі висвітлено деякі з цих інструментів, щоб обговорити їхню мету, результати та недоліки. Для JavaScript підходи до створення тестових випадків можна класифікувати за простором введення, на якому вони працюють. Таких просторів два: простір подій і простір значень [29]. Простір подій стосується порядку подій у JavaScript. Простір подій здебільшого обертався навколо інтерфейсів користувача, наприклад, порядку натискання певних кнопок. На з іншого боку, простір значень стосується значень, які використовуються для виконання певних функцій. Обидві ці класифікації

зосереджені на клієнтському JavaScript. На додаток, більшість літератури про генерацію тестів JavaScript зосереджено на клієнтських програмах. Однак модульне тестування для серверного JavaScript схоже на підходи, орієнтовані на простір значень.

Artemis & SymJS.

Artemis є одним із перших інструментів генерації тестів для JavaScript [30]. Він використовує алгоритм довільного тестування зі зворотним зв'язком для перевірки веб-програм JavaScript. Artemis оперує як простором подій, так і простором значень. Він не використовує жодної форми виведення типу. SymJS – це система автоматичного символічного тестування для клієнтських веб-додатків JavaScript [31]. Він заснований на Artemis [30] і має на меті покращити дослідження простору цінностей за допомогою Concolic виконання. SymJS враховує лише два типи: числа та рядки. Дослідниками було запропонував покращення шляхом створення символьних вводів на основі ручних анотацій типу. Іншими словами, вони використовують анотації типу, які розробники можуть визначити в документації функції JavaScript. Хоча використання документації для вилучення типів є надійним розв'язанням проблеми, воно вимагає, щоб документація була написана в певному форматі, який містить тип необхідних змінних. Це також вимагає, щоб документація типу була дуже конкретною. Наприклад, якщо для функції потрібен певний тип об'єкта, не дуже корисно вказувати тип аргументу як «Об'єкт». Натомість він має вказувати точні властивості об'єкта.

JSeft.

JSeft – це інструмент генерації тестів JavaScript, зосереджений на створенні тестів на основі подій і модульних тестів функціонального рівня [32]. Інструмент зосереджено на тестуванні веб-додатків. JSeft починає зі створення графіка потоку стану (state-flow graph, SFG). Цей SFG потім використовується для генерації тестів на основі подій. На основі елементів веб-додатку витягуються стани функції JavaScript для створення модульних тестів функціонального рівня. Оскільки JSeft працює на рівні Document Object Model (DOM), з якого безпосередньо витягує виклики функцій з аргументами, йому не потрібно турбуватися про визначення

типів. Він може просто витягувати типи аргументів зі знайдених викликів функцій. Хоча це робить JSeft менш актуальним для цього дослідження, він надає цікаву ідею, яку можна використати під час визначення типу. Можливо отримати підказки щодо типів аргументів, які слід використовувати, дивлячись на виклики певної функції.

Jalangi.

Jalangi вперше був представлений як основу для динамічного аналізу JavaScript [33]. Хоча це не інструмент для створення тестових випадків, він виконує Concolic тестування. Спочатку типи вхідних значень вибиралися на основі їх безпосереднього використання в умовах гілки, які використовуються при тестуванні. Іншими словами, ширший контекст, у якому використовуються змінні, не розглядався. У цьому контексті усвідомлення типу означало, що алгоритм тестування Concolic обробляв обмеження типу окремо від обмежень розгалуження. Додавання усвідомлення типу значно зменшило кількість надлишкових вводів, тобто вводів, які не досягають нового покриття коду під час використання інших типів аргументів. І тестування на основі пошуку, і тестування Concolic страждають від проблем масштабування, коли кількість можливих комбінацій вхідних даних велика. Результати вказують на те, що знання обмежень типу зменшує кількість можливих комбінацій вводу. Доступ до інформації про тип аргументів також може бути корисним для методів тестування на основі пошуку.

Pynguin.

Pynguin – це автоматизований фреймворк для створення модульних тестів для Python [34]. Це має на меті перевірити ефективність створення тестів для динамічно типізованих мов програмування. Крім того, мета полягає в тому, щоб емпірично оцінити вплив інформації про тип на можливості генерації тестів Pynguin. Pynguin використовує техніку, засновану на пошуку, і застосовує підхід до генерації тестів у цілому. Як і SymJS, він використовує анотації типів, додані розробниками. Результати показують, що генерація тестів на основі пошуку ефективна для більшості тестів при роботі з DTL. Автори також дійшли висновку, що включення інформації про тип дозволяє Pynguin охоплювати більші частини коду. Однак

кажуть, що SUT (Software Under Test), які використовують складніші типи, вигода більша, ніж SUT, які використовують лише прості типи.

Підсумовуючи, можна стверджувати, що більшість автоматизованих інструментів тестування для JavaScript зосереджені на клієнтських програмах замість серверних програм Node.js [30, 31, 32]. Підходи, орієнтовані на простір подій, не застосовуються до програм на стороні сервера. Однак методи, орієнтовані на простір значень, можуть бути застосовані до інструментів автоматизованого тестування для програм на стороні сервера. Інструменти, які зосереджуються на серверних програмах JavaScript, часто лише створюють тестові випадки системного рівня [35, 36, 37]. Обговорювані інструменти та підходи або використовують лише кілька підказок типів, або повністю ігнорують типи аргументів. Використані типізовані підказки здебільшого складаються з анотацій типів, наданих розробником.

Враховуючи літературу, виявлено, що створення тестів для JavaScript зосереджено в основному на клієнтських програмах. Можна зробити висновок, що поточні автоматизовані підходи до створення тестових випадків для динамічно типізованих мов не використовують повноцінні методи виведення типу. Що стосується методів виведення типу, більшість страждає від словника фіксованого розміру. Без перенавчання моделей обробки природної мови методики не можуть передбачити типи, визначені користувачем. Перенавчання є дорогим і тому небажаним під час автоматизованої генерації тестів. У цьому дослідженні пропонується новий неконтрольований підхід до роботи з динамічно типізованими мовами програмування під час автоматизованої генерації тестів. Цей підхід використовуватиме декілька розглянутих методів виводу типу статичного аналізу у поєднанні з новою технікою динамічного аналізу, унікальною для створення тестових випадків. Новий підхід працює без нагляду, не потребує навчання та має нескінченний словниковий запас.

Це дослідження оцінює вплив на продуктивність використання статичного виводу типу (Static Type Inference) для генерації тестів для JavaScript. Крім того, вимірюється вплив використання статичного виводу типу та динамічного виводу

типу. Загальна ідея цього полягає в тому, що якщо можливо підвищити точність виведених типів шляхом включення інформації про виконання з процесу пошуку, тобто виконання динамічного аналізу, якість згенерованих тестів також покращиться. Тобто підхід досягне вищого структурного охоплення, оскільки він містить більше інформації про типи аргументів. Щоб оцінити цю гіпотезу, пропонуємо новий підхід, який інтегрує неконтрольований імовірнісний висновок типу в процес генерації тестів на основі пошуку, щоб отримати необхідну інформацію про тип.

Цей підхід базується на попередніх дослідженнях виведення типу. Він покладається на методи статичного аналізу для виконання початкового виводу типу. Використовувані методи є логічним висновком у поєднанні з контекстним висновком. В ідеалі підходи до створення тестів присвячують більшу частину свого бюджету часу пошуку тестів із високим рівнем покриття. Таким чином, використання методів машинного навчання не є сприятливим, оскільки навчання таких моделей є дорогим. Оскільки техніки обробки природної мови зазвичай потребують певної форми машинного навчання, розглядатись такі техніки не будуть. На відміну від більшості подібних робіт, новий підхід також використовує методи висновку типу динамічного аналізу. Він розглядає інформацію про правильність виведених типів шляхом аналізу інформації про виконання згенерованих тестів. Потім використовуємо цю інформацію для вдосконалення моделей імовірнісного типу, створених за допомогою методів висновку типу статичного аналізу. Неконтрольований вивід імовірнісного типу включає чотири фази, як показано на рисунку 2.1.

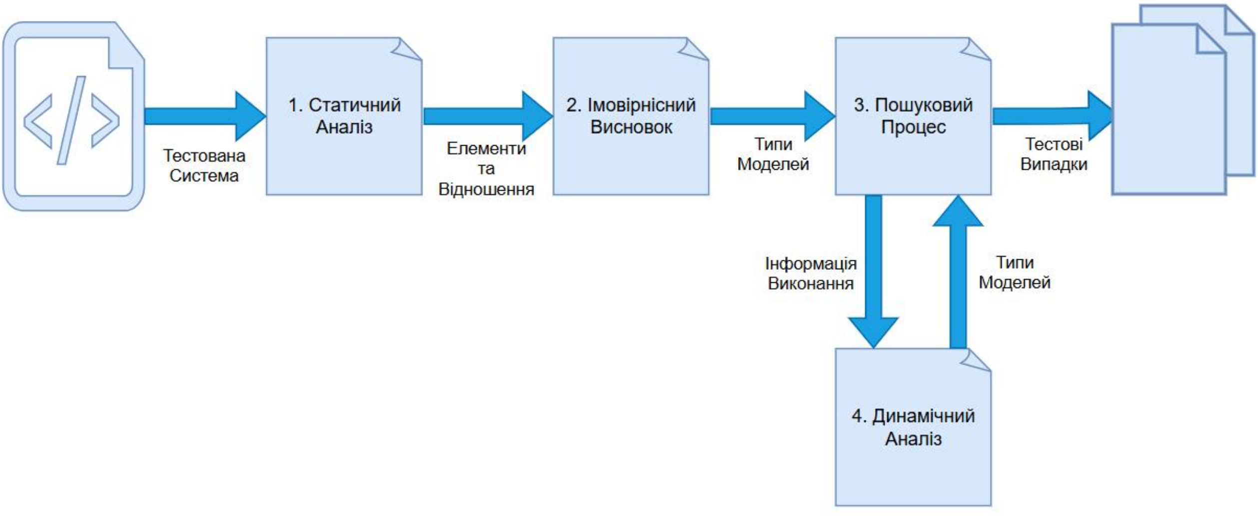


Рисунок 2.1 – Схема генерації тестових випадків

Перший етап складається зі статичного аналізу щодо виведення типу тестованого програмного забезпечення, результатом якого є набір витягнутих елементів і зв'язків. Другий етап використовує витягнуті елементи та зв'язки для створення моделей імовірнісного типу для кожного елемента. Третій етап складається з процесу пошуку. Під час процесу пошуку моделі типів використовуються для вибірки вхідних даних, що призводить до тестових випадків, які охоплюють цілі галузі. Згенеровані тести виконуються під час процесу пошуку. З виконання витягується інформація про правильність типів, використаних у тестовому випадку. На четвертому етапі інформація про виконання використовується для коригування моделей імовірнісного типу, щоб вони стали більш точними, ніж раніше. Третя та четверта фази продовжують взаємодіяти одна з одною до тих пір, поки дозволяє бюджет або покриваються всі цілі галузі. Оскільки мета полягає в тому, щоб знайти набір тестів, який охоплює більшість гілок і виявляє помилки, кінцевий набір тестів є результатом процесу. Варто розглянути кожну з описаних фаз більш детально.

Перший етап складається з перевірки SUT і його залежностей для збору інформації, яка може бути використана для визначення типів. Зібрана інформація складається з:

– елементи: ідентифікатори та літерали;

– відносини: вирази та операції, що включають один або більше елементів;

– визначені користувачем типи: опис типів на основі класів, інтерфейсів, прототипованих функцій або ініціалізаторів об’єктів.

Щоб отримати необхідну інформацію, увесь код перетворюється на абстрактне синтаксичне дерево (Abstract Syntax Tree, AST). Далі проходимо елементи дерева, щоб знайти зв’язки та їх залучені елементи. Крім того, усі визначені користувачем об’єкти витягуються з AST для створення описів типів об’єктів.

Елементи.

Елементи представляють частини тестованого програмного забезпечення, для яких необхідно вивести типи. Елемент може бути ідентифікатором змінної або літералом. Ідентифікатори – це іменовані посилання на змінні, функції та властивості. Літерали – це необроблені значення, які можна присвоїти змінним або константам. Оскільки літерали безпосередньо представляють тип, нам не потрібен висновок. Однак, як раніше було описано, ідентифікатори не мають явних типів. Щоб дізнатися більше про тип ідентифікатора, можна використати контекст ідентифікатора, тобто зв’язки.

Відношення.

Відношення – це вирази та операції, що включають один або більше елементів. Вони описують, як використовуються елементи та як вони пов’язані з іншими елементами. Відношення можуть розповісти нам більше про типи елементів. Як приклад співвідношення, припустимо, що змінній L присвоєно R (L = R). Якщо R є логічним літералом, з чого виникає висновок, що L також має бути логічним у цій точці коду. Відношення витягуються з AST і перетворюються у формат, який дозволяє легко ідентифікувати. На рисунку 2.2 показано менший зв’язок між змінною, «a» та літералом 2 у рядку 2. Відношення перетворюється та записується як [L < R, a, 1], як показано на рисунку 2.3.

```
1 function example(a) {
2     if (a < 1) {
3         return 0;
4     }
5     return a
6 }
7
8 example(2);
```

Рисунок 2.2 – example функція.

```
1 [L R, example, a]
2 [L < R, a, 1]
3 [L -> R, example, 0]
4 [L -> R, example, a]
5 [L(R), example, 2]
```

Рисунок 2.3 – Відношення на основі example функції.

Список деяких можливих зв'язків наведено в таблиці 2.1. У таблиці вказано категорію оператора, назву оператора та формат зв’язку для всіх операторів, доступних у JavaScript. В даній таблиці список не є повним. Більшість цих операцій взято з веб-документації MDN від Mozilla [38].

Таблиця 2.1 – Приклад деяких відношень в JavaScript.

Категорія	**Оператор**	**Відношення**
Рівність	Рівність	L==R
	Нерівність	L!=R
	Строга рівність	L===R
	Строга нерівність	L!==R
Бінарні бітові операції	Бітовий AND	L&R
	Бітовий OR	L|R
	Бітовий XOR	L^R
Тернарні	Умовний	C?L:R
Присвоєння	Присвоєння	L=R

Окрім очевидних зв'язків, таких як двійкова операція або вираз присвоювання, повний список відношень також містить зв'язки, такі як аргументи функції, повернення функції та виклики функції. Запис таких додаткових відносин дає додаткове уявлення про можливі типи елементів. Наприклад, функція «example», що має параметр «a». Припустимо, що необхідно зробити висновок про тип «a». Далі у фрагменті коду, рядок 8, показує, що приклад функції викликається з використанням «2» як аргументу, який є числовим літералом. Це означає, що параметр «a» може бути числовим. Однак, оскільки JavaScript є динамічно типізованою мовою, не можна сказати це з упевненістю, тобто функція прикладу може дозволити бути як числовим, так і рядковим, або будь-яким іншим типом.

Часто код не такий простий, як минулий приклад. Код регулярно містить вкладені відношення. Вкладені відношення – це відношення, де залучені елементи самі є відношеннями. На рисунку 2.4 продемонстровано приклад таких вкладених відносин.

```
1  const x = (a == b ? 6 : 10);
```

Рисунок 2.4 – Дещо складніший приклад коду.

Константі «x» присвоюється значення, тому форматоване відношення дорівнює [L = R, x, y*]. Відформатований зв'язок не містить усієї правої частини призначення. Замість цього визначається y*. y* – це штучний елемент, який вказує на відношення, яке є у правій частині призначення. Права частина присвоєння виявляється тернарним оператором. y* вказує на [C?L : R, z* , 6, 10]. Знову створюється штучний елемент z*, який вказує на відношення рівності в умовній частині потрійного оператора. Отже, z* вказує на кінцеве співвідношення [L == R, a, b]. Хоча код здається досить простим, тут є три зв'язки, які містяться одне в одному. Ці співвідношення показані на рисунку 2.5.

```
1  [L = R, x, y*]
2  y* = [C?L : R, z*, 6, 10]
3  z* = [L == R, a, b]
```

Рисунок 2.5 – Розписані відношення коду показаного на рисунку 2.4.

Ці вкладені відносини дають додаткове уявлення про типи елементів, як показано на рисунку 2.6 та рисунку 2.7. Присутні два відношення. Перше – це відношення присвоєння між «х» та y*, тобто [L = R, x, y*]. Тут y* вказує на порівняння, [L < R, a, b]. Відношення порівняння вкладено у відношення присвоєння. Оскільки відношення L < R завжди призводить до булевого значення, можна зробити висновок, що х має бути логічним у цій точці коду.

```
const x = a < b;
```

Рисунок 2.6 – вкладені відношення.

```
1  [L = R, x, y*]
2  y* = [L < R, a, b]
```

Рисунок 2.7 – відношення.

Область дії.

Як показано в попередніх підрозділах, інформацію про типи елементів можна витягти з відношень, у яких ці елементи беруть участь. Однак одним з важливих аспектів елементів, які ще не обговорювались, є охоплення. Область дії ідентифікатора визначає його доступність. На рисунку 2.8 показано приклад того, що робить область видимості. По-перше, константі «х» присвоюється значення «5». Константа «х» визначається в так званій глобальній області видимості. Далі визначається функція, яка створює нову область. Ця область має доступ до посилань глобальної області, але також може мати власні посилання, які доступні лише для неї самої та її підобластей. У даній функції спостерігаємо, що визначена інша константа «х». Зверніть увагу, що з цього моменту кожне посилання на «х» в області видимості функції посилається на щойно визначену константу, а не на константу з глобальної області видимості.

```
1  const x = 5
2
3  function example(a) {
4      const x = "Hello "
5      return x + a
6  }
```

Рисунок 2.8 – Приклад змінної з прихованою областю видимості функції.

Це явище називається змінним затіненням. Коротко кажучи, коли ідентифікатор повторно оголошується у вужчій області, оголошення у вужчій області затіняє оголошення в ширшій області. У контексті фази вилучення цей принцип затінення є фундаментальним, оскільки змінна «x» у глобальній області видимості не є такою ж змінною, як змінна в області видимості функції. Вони можуть мати різні види. У цьому випадку «x» із глобальної області видимості є числовим, тоді як «x» з області видимості функції є рядком. Щоб відстежувати область видимості елементів, зберігаємо елементи та зв'язки з відповідним ідентифікатором області видимості. Ідентифікатор області дозволяє методам висновку поєднувати різні типи з елементами з однаковими іменами.

Складні типи.

У JavaScript об'єкти є важливою частиною мови. Об'єкти в JavaScript – це сховища пар ключ-значення. Крім примітивних типів, таких як булеві значення або числа, майже все можна представити як об'єкт. Масив можна, наприклад, розглядати як спеціальний об'єкт, де ключами є числа. У сучасних версіях JavaScript розробники можуть визначати класи та інтерфейси, які створюють більш об'єктно-орієнтований підхід до JavaScript. Оскільки об'єкти відіграють таку важливу роль у JavaScript, дуже важливо, щоб типи об'єктів також могли бути визначені. Щоб визначити тип об'єкта, спочатку необхідно витягти всі описи типів об'єктів, доступні в тестованому програмному забезпечені. До них належать визначення класів та інтерфейсів, а також стандартні об'єкти, такі як масив або функція.

Ймовірнісний висновок.

Після вилучення всіх елементів, зв’язків і можливих типів в тестованому програмному забезпеченні, починається друга фаза. На цьому етапі створюємо модель імовірнісного типу для кожного елемента. Це просто для елементів, які представляють літерали, оскільки тип можна безпосередньо вивести з типу літерала. Однак не кожен елемент у коді є літералом. На основі отриманих співвідношень робимо припущення. Наприклад, коли обробляється відношення [L = R, x, 5], можна зробити висновок, що в цій конкретній точці коду «x» має бути числового типу, оскільки йому присвоєно літеральне значення «5». Однак немає впевненості, що «x» є числом до чи після цього конкретного відношення. Змінна «x» може бути повторно призначена до іншого типу. Таким чином, система підрахунку балів використовується замість того, щоб фіксувати числове значення «x». Моделі типів складаються з карти, що пов’язує типи з балами. Якщо елемент, можливо, належить до певного типу, тоді цей тип має ненульову оцінку в моделі типу елемента. Отримуємо ці оцінки з отриманої інформації про відношення. Використовуючи попередній приклад співвідношення [L = R, x, 5], «x» призначається точка для числового типу. Якщо «x» пізніше буде присвоєно значення рядка, він також отримає бал за тип рядка. Приклад простий, оскільки права частина присвоєння є літеральним значенням. Однак, як згадувалося раніше, JavaScript погано типізується. Наступне співвідношення демонструє це: [L + R, x,5]. У цьому відношенні «5» додається до «x». Хоча права частина знову є буквальним значенням, нічого не можна сказати про «x». Можливо лише зробити припущення, що «x», ймовірно, числове, але з такою ж імовірністю це рядок. Для цього конкретного відношення доцільно давати однакову оцінку як для рядкового, так і для числового типу.

Розділення складних типів.

Щоб визначити, які елементи тестованого програмного забезпечення є об'єктами, перевіряємо елементи для відношення Property Accessor. Порівнюються доступні властивості з доступними описами типів об’єктів, якщо елемент бере участь в одному або кількох зв’язках доступу до властивостей. Якщо існує перекриття між властивостями елемента та властивостями опису типу об’єкта,

призначається опис типу як можливий тип елемента. На рисунку 2.9 зображена діаграму Венна прикладу одного з тестів.

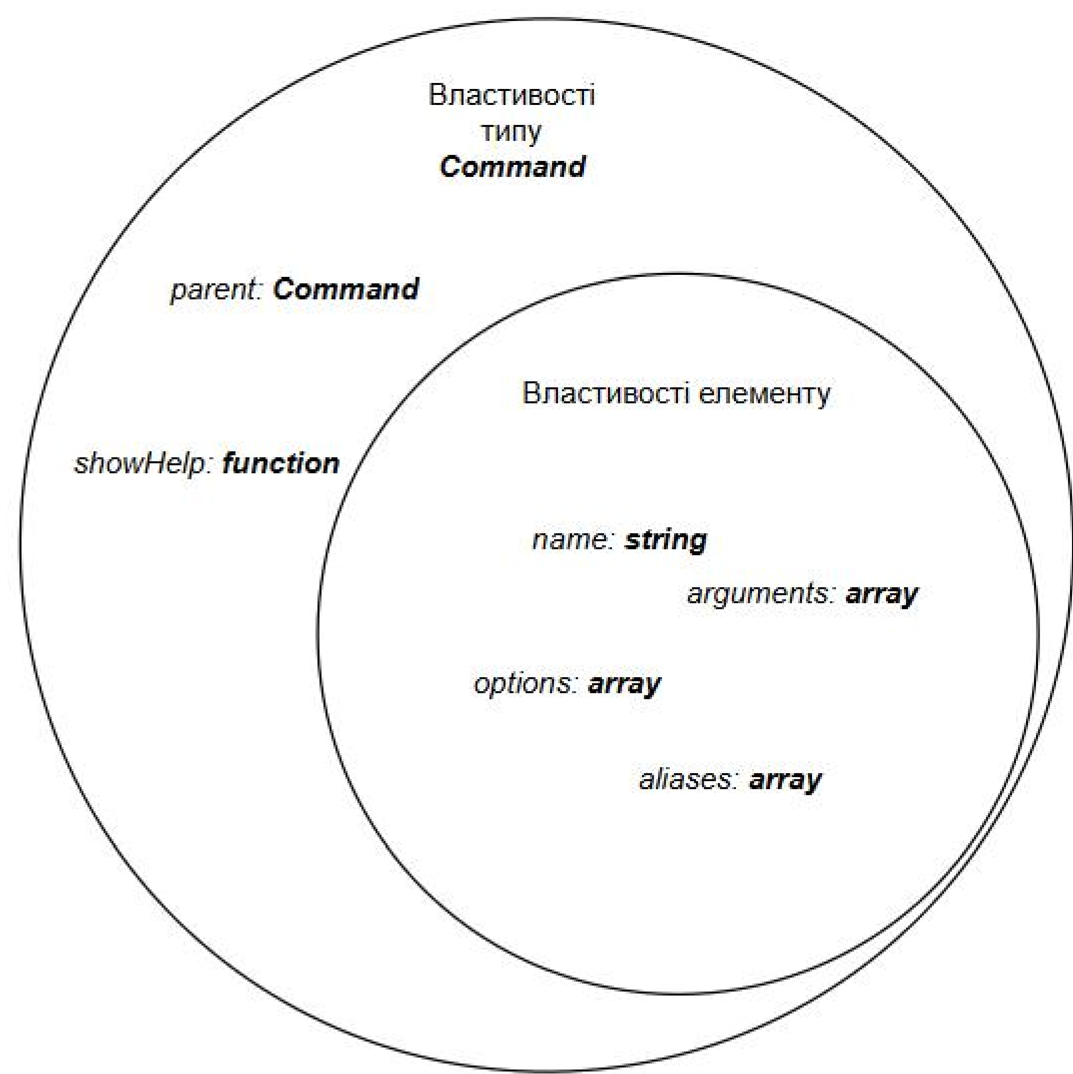


Рисунок 2.9 – Приклад діаграми Венна опису складного типу та доступних властивостей елементів.

У бенчмарку знаходиться складний тип під назвою Command. Цей об’єкт Command має кілька властивостей, таких як ім’я, набір аргументів, батьківська команда тощо. На діаграмі Венна, об’єкт Command показано як найбільше коло. Тепер припустімо, що потрібно визначити тип елемента «x». У (гіпотетичному) контексті коду «x» спостерігаємо, що використовуються властивості імені, аргументів, параметрів і псевдонімів «x». Показуємо це як менше коло на діаграмі

Венна. Очевидно, що властивості «х» і властивості типу Command збігаються. Через це перекриття призначаємо оцінку типу Command для елемента «х».

На додаток до відповідних описів об’єктів створюється анонімний тип об’єкта, який призначається як можливий тип. Цей тип анонімного об’єкта точно відповідає властивостям елемента. Анонімний тип об’єкта корисний, коли у вихідному коді немає відповідного опису типу. Це гарантує можливість використовувати відповідний тип під час процесу пошуку.

Імовірності.

Після присвоєння балів можливим типам, використовуємо бали для обчислення ймовірностей типів. Чим вище оцінка певного типу, тим вища ймовірність того, що елемент належить до цього типу. Імовірність «pi» типу «i» обчислюється шляхом ділення оцінки на загальну оцінку ${}_i s_i$, як показано в формулі 2.1, сума ймовірностей дорівнює 1.

$$p_i = \frac{s_i}{\sum_i s_i} \tag{2.1}$$

Імовірності використовуються на етапі пошуку для вибірки аргументів правильного типу. Для вибірки аргументу необхідно вибрати один із типів у моделі типу. Для цього дослідження було створено два режими. Перший режим – рангової вибірки. У цьому режимі завжди вибирається тип із найвищою ймовірністю. Другий режим – пропорційної вибірки. У цьому режимі тип вибирається випадковим чином на основі його ймовірності, тобто якщо тип має 50% вірогідності бути правильним типом, він має 50% шанс бути обраним.

Залежності типів.

Крім типів елементів, самі відносини також мають тип результату. Розглянемо, наприклад, співвідношення: [L > R, a, b]. У цьому відношенні тип виведення є логічним незалежно від типів «a» і «b». Без додаткового контексту це не дуже корисно. Часто доводиться стикатись з вкладеними відносинами в коді. Наприклад, [L = R, c, d*], де d* дорівнює [L > R, a, b]. Оскільки результат d* є логічним значенням, можна зробити висновок, що «c» також має бути логічним значенням. Однак замість того, щоб призначати бал для логічного значення «c», робимо тип «c»

залежним від типу d*. Робимо це, тому що є відносини, де результат відносин залежить від залучених елементів. Роблячи типи елементів залежними один від одного, створюється мережа ймовірностей типу.

На рисунку 2.10 показано вихідний код. Розглянемо функцію add з двома аргументами «a» і «b». У другому рядку змінна «c» визначена такою, що дорівнює «a + b». Результат типу відношення «a + b» залежить від типів «a» і «b». Змінна «c» безпосередньо залежить від результату співвідношення.

```
1 function add(a, b) {
2     const c = a + b
3     return c
4 }
5
6 function main () {
7     const e = add("Hello ", "World")
8     print(e)
9 }
```

Рисунок 2.10 – Код.

У другій функції в рядку 7, функція add викликається з двома аргументами, а саме «Hello» і «World». Тепер можна сказати, що тип першого аргументу add залежить від типу літерала «Hello». Цей літерал є рядком, який вказує, що «a», ймовірно, також належить до цього типу. Це поширюється всією мережею. Мережа ймовірності типу гарантує, що щоразу, коли стає доступною нова інформація про тип певного елемента, вона поширюється на всі моделі пов'язаного типу. Через залежності типів між елементами вкрай важливо, щоб для кожного елемента була створена модель типу, а не лише для цікавих елементів.

Також показано відносини залежності у вигляді орієнтованого графа на рисунку 2.11. На графіку стрілки вказують від залежного до залежного. У рядку 3 знаходимо оператор return, який повертає «c». На графіку це показано як тип повернення функції add, що залежить від типу «c».

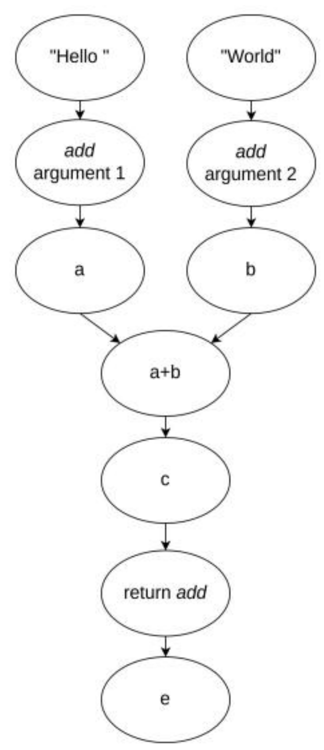


Рисунок 2.11 – Відносини залежності.

Процес пошуку.

Третя фаза підходу – процес пошуку. На цьому етапі використовується евристика для створення тестових випадків з метою виявлення помилок у тестованому програмному забезпеченні. Використовується динамічний багатоцільовий алгоритм сортування [39]. Цей алгоритм перевершує інші алгоритми, коли справа доходить до автоматизованої генерації тестів.

Еволюційний підхід випадковим чином генерує набір тестів. Потім ці тестові приклади оцінюються, дивлячись на те, наскільки вони близькі до досягнення цілей певної галузі. Найперспективніші тестові приклади відбираються для наступного покоління, інші «вбиваються». Популяція тестів, що залишилася, потім використовується для створення нової популяції, потомства. Ця популяція нащадків створюється за допомогою варіаційних операторів, таких як мутація та crossover для членів батьківської популяції. Потім популяція нащадків додається до загальної популяції, і процес повторюється. І під час випадкової генерації, і під час зміни тестових випадків моделі типів використовуються для вибірки аргументів для викликів функцій або конструкторів.

Оцінка тестів вимагає, щоб тести запускались для тестованого ПЗ. Під час виконання тестів у тестованому ПЗ можуть виникати винятки. Ці винятки іноді можуть надати інформацію про правильність типів використаних аргументів. Ця інформація зберігається та використовується на четвертій фазі.

Динамічний аналіз.

На третьому етапі моделі ймовірнісного типу використовуються для створення тестових випадків із можливо правильними типами. Однак, оскільки моделі ймовірнісні, немає гарантії, що використовується правильний тип. Поки ці тестові випадки виконуються, деякі тестові випадки можуть викликати винятки в тестованому ПЗ. Деякі з них можуть бути фактичними помилками або навмисними винятками в програмному забезпеченні. Інші – це помилки типу, які вказують на невідповідність типу десь у вихідному коді. Причиною цієї невідповідності може бути неправильно визначений тип аргументу. Під час фази динамічного аналізу викинуті винятки перехоплюються та обробляються.

Окрім показників типу, які вказують на те, що елемент належить до певного типу, кожен елемент також має оцінки виконання. Ці оцінки виконання використовуються, щоб оновлювати моделі типів, використовуючи результати виконання тестів. Якщо під час виконання тестового прикладу виникає виняток TypeError, йде перехід до сканування повідомлення про помилку або трасування стека на наявність імен змінних, які використовуються у згенерованому тестовому випадку. Якщо відповідну змінну знайдено, тоді призначаємо негативну оцінку виконання типу, який використовувався для цієї змінної. На рисунку 2.12 наведено приклад трасування стека. З першого рядка трасування стека можна зробити висновок, що властивість опису об’єкта cmd мала бути функцією. У цьому прикладі властивість description була відібрана як рядок, тому моделі типу властивості description буде присвоєно негативний бал для рядка типу. Оцінки виконання моделі типу використовуються для розрахунку ймовірності типу. Ідея полягає в тому, що для кожної помилки TypeError, яка виникає, налаштовуються моделі типів, щоб вони ставали точнішими.

```
TypeError: cmd.description is not a function
    at Help.subcommandDescription (.syntest/instrumented/benchmark/
      top10npm/commanderjs/lib/help.js:3945:16)
    at _callee$ (.syntest/tests/tempTest.spec.js:5:22)
    at tryCatch (node_modules/regenerator-runtime/runtime.js:63:40)
    at Generator.invoke [as _invoke] (node_modules/regenerator-
      runtime/runtime.js:294:22)
    at Generator.next (node_modules/regenerator-runtime/runtime.js
      :119:21)
    at asyncGeneratorStep (.syntest/tests/tempTest.spec.js:7:103)
    at _next (.syntest/tests/tempTest.spec.js:9:194)
    at .syntest/tests/tempTest.spec.js:9:364
    at new Promise (<anonymous>)\n at Context.<anonymous> (.syntest/
      tests/tempTest.spec.js:9:97)
```

Рисунок 2.12 – Приклад трасування стека з корисною помилкою.

Щоб обчислити остаточну ймовірність «pi» типу «i», використовується формула 2.2. Перший дріб у рівнянні дорівнює попередньо наведеній формулі. Другий дріб представляє ймовірність результату виконання. Оскільки оцінка виконання, es_i дорівнює нулю або є від'ємним (якщо використаний тип дає TypeErrors), додається абсолютне значення найбільш негативної оцінки $min_i(es_i)$ до оцінок виконання всіх типів. Це гарантує, що початковий найбільш негативний бал тепер має нульовий бал, а всі інші бали є позитивними. Ці модифіковані оцінки ідентифікуються es'_i Після цього робимо те ж саме, що і зі звичайними оцінками, тобто ділимо бал es'_i«i» за сумою всіх балів es'_i.

$$p_i = \frac{s_i}{\sum_i s_i} \cdot \frac{es_i + |min_i(es_i)|}{\sum_i (es_i + |min_i(es_i)|)} = \frac{s_i}{\sum_i s_i} \cdot \frac{es'_i}{\sum_i es'_i} \quad (2.2)$$

Тепер слід зазначити, що після цього множення сума ймовірностей більше не дорівнює 1. Щоб виправити це, ділимо всі ймовірності на суму ймовірностей у кінці процесу.

Варіанти підходу.

Враховуючи етапи, описані в цьому розділі, було створено п'ять різних варіантів підходу, щоб відповісти на запитання дослідження. Ці варіанти наведено в таблиці 2.2. Перший варіант є базовим, з яким будуть порівнюватись чотири інші варіанти. Базовий рівень не використовує статичний або динамічний висновок типу.

Типи, які використовуються для базової лінії, є абсолютно випадковими, тобто всі типи мають однакову ймовірність використання. Оскільки статичний аналіз не проводиться, базова лінія здатна відбирати лише примітивні типи (число, логічне значення тощо) і стандартні складні типи JavaScript (Object, Array, String). Другий і третій варіанти використовують лише статичний висновок типу для своїх моделей типів. Іншими словами, вони не використовують четверту фазу. Другий і третій варіанти використовують вибірку на основі рангів і пропорційну вибірку відповідно. Четвертий і п’ятий варіанти використовують статичний і динамічний висновок типу. Один використовує режим рангової вибірки, а інший використовує режим пропорційної вибірки.

Таблиця 2.2 – Варіанти підходу.

Варіант	**Режим вибірки**	**Режим аналізу**
Базовий	-	-
СА (статичний аналіз) на основі рангу	На основі рангу	Статичний аналіз
СА (статичний аналіз) пропорційний	Пропорційний	Статичний аналіз
СА+ДА (статичний + динамічний аналіз) на основі рангу	На основі рангу	Статичний аналіз та динамічний аналіз
СА+ДА (статичний + динамічний аналіз) пропорційний	Пропорційний	Статичний аналіз та динамічний аналіз

2.5 Висновки.

В даному розділі було описано концепцію автоматичної генерації тестів. Серед таких концепцій розглянуто, що таке автоматична генерація тестів, його підвиди та характеристики, такі як покриття коду.

Наступним кроком, було розглянуто динамічні мови програмування та їх специфіка, із-за чого є різні підходи до автоматичної генерації. Важливий аспектом також був розглянутий вивід типів. Огляд наявних методологій виведення типу вказав на те, що існує багато способів вирішення цього завдання.

Важливим етапом даного розділу став розгляд подальшої стратегії для реалізації виводу типів для подальшої автоматичної генерації тестів шляхом додання статичного та динамічного аналізу в пошуковий метод генерації тестів. Було розроблено загальну схему, яка описує стратегію по якій працюватиме виведення типу та генерація тестів. Далі також було висвітлено елементи, важливі для нашого дослідження, серед яких:

– елементи;
– відношення;
– складні типи;
– імовірності.

Завершальним етапом даного розділу є представлення таблиці варіантів підходу. Дані варіанти описують способи тестування результатів нашого дослідження, які будуть порівнюватись між собою по завершенню роботи.

3 АЛГОРИТМИ ТА ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ

3.1 Алгоритми вирішення задачі

Визначившись з методами розв’язку задачі та розглянувши достатню кількість матеріалу у попередньому розділі, можемо перейти до більш детального опису алгоритму.

Розглянемо детальніше процедуру всього процесу генерації тестів. Даний процес складається з наступних етапів:

– статичний аналіз коду;

– визначення типів даних;

– процес генерації тестів на основі пошуку;

– динамічний аналіз;

– повторний процес генерації тестів на основі пошуку з використанням результатів динамічного аналізу;

– генерація тестових випадків.

Спершу використовуємо статичний аналіз для виявлення помилок та збору інформації для тестованого програмного забезпечення. Далі, базуючись на результатах статичного аналізу, намагаємось визначити можливі типи даних.

Наступним етапом пошуковий алгоритм генерує тестові випадки, та додатково виконує деякий код який не зміг бути аналізований під час статичного аналізу. Це виконання коду та збір даних і є динамічним аналізом, який також допомагає в генерації тестових випадків. Фінальним етапом є готові тестові випадки, які мають в більшій мірі бути задовільними, оскільки для них було зібрано інформацію про типи даних ПЗ.

В цілому можна розділити програмний засіб на три основні групи компонентів відповідно до раніше описаного сценарію роботи.

Перша група компонентів складатиметься з елементів необхідних для функціонування статичного аналізу. Серед таких компонентів варто виділити абстрактне синтаксичне дерево та граф потоку керування.

Абстрактне синтаксичне дерево (AST, Abstract Syntax Tree) – це деревоподібна структура даних, яка використовується для представлення синтаксичної структури вихідного коду в мовах програмування. Кожен вузол дерева відповідає певній конструкції коду, такій як змінна, функція, оператор або умовний блок. AST широко застосовується в компіляторах, інтерпретаторах і інструментах аналізу коду, оскільки воно абстрагує деталі синтаксису та зосереджується на логічній структурі.

Граф управління потоком (Control Flow Graph, CFG) – це графічне представлення потоку управління в програмі або конкретній функції/модулі. Він допомагає зрозуміти, як виконуються різні частини програми, що корисно для аналізу, оптимізації, налагодження та тестування коду.

Друга група компонентів відповідатиме за виведення типів даних. Це механізм у мовах програмування, який дозволяє компілятору або інтерпретатору автоматично визначати тип змінної або виразу на основі контексту без явного оголошення типу. У нашому випадку нам необхідний даний механізм для аналізу типів, оскільки JavaScript динамічно-типізована мова програмування з слабкою типізацією. Необхідно вивести типи даних перед тим як використовувати їх в генерації тестів. Серед основних складових можна виділити Type Resolver та Execution Information Integrator.

Type Resolver – це механізм або компонент, який використовується для визначення або обробки типів у програмуванні. Він допомагає системі інтерпретувати типи змінних, параметрів функцій або даних у структурі. Це особливо корисно в системах із динамічним або змішаним типізуванням, або в середовищах, які підтримують метапрограмування.

Execution Information Integrator (EII) – це концепт або компонент у програмуванні, що забезпечує збір, обробку та об'єднання інформації про виконання системи або програми. Такий інтегратор корисний для аналізу продуктивності, відстеження помилок, логування, моніторингу та інших завдань, пов'язаних із виконанням програмного коду.

Остання група компонентів відповідатиме за пошуковий алгоритм та буде використовуватись в останню чергу, оскільки спершу необхідно проводити статичний аналіз тестованого програмного забезпечення.

3.2 Проектування програмного засобу

В основі програмного засобу лежать три основні групи компонентів. Перша група компонентів відповідає за виконання статичного аналізу. Друга за виведення типу, а третя за безпосередньо виконання пошукового алгоритму генерації тестів. Повна схема розроблених компонентів зображена на рисунку 3.1.

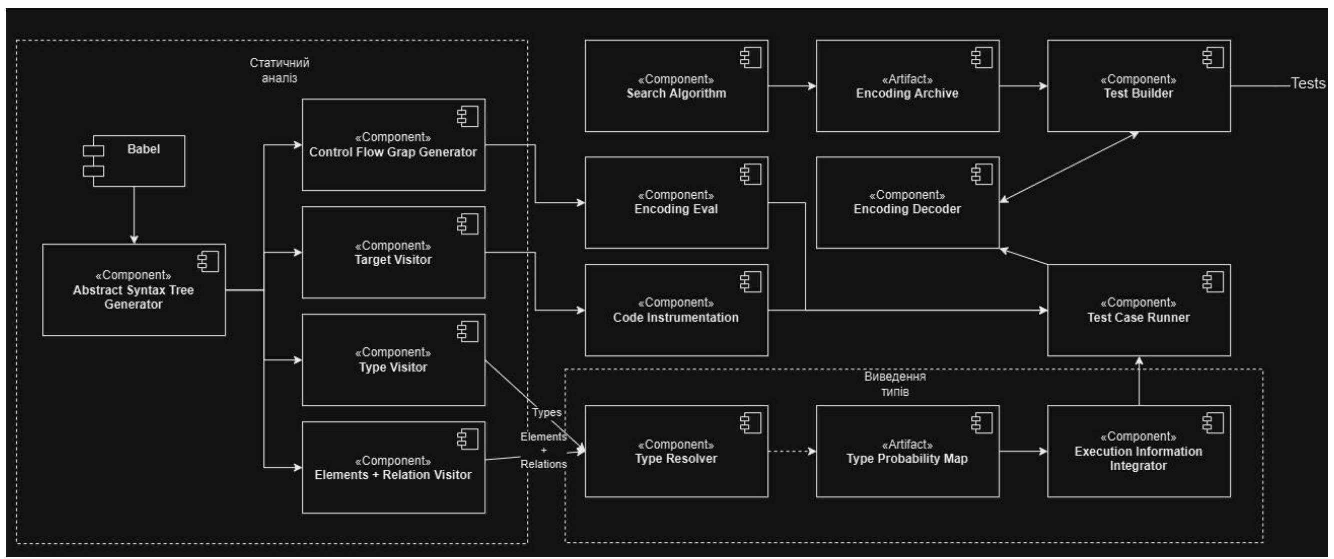


Рисунок 3.1 – Схема розроблених компонентів програмного засобу.

Далі детальніше опишемо все що показано на схемі. Розпочнемо з групи статичного аналізу. В схемі показано що статичний аналіз складається з генератора абстрактного синтаксичного дерева, генератора графа потоку керування, цільового відвідувача, відвідувача типу та відвідувача елементів і зв’язків.

Статичний аналіз починається з перетворення вихідного коду в абстрактне синтаксичне дерево. Для цього використаємо бібліотеку Babel. Інші компоненти статичного аналізу також використовуватимуть цю бібліотеку для проходження AST за допомогою шаблону відвідувача.

Наступним компонентом є генератор графів потоку керування. Як вказує назва, цей компонент використовує AST для створення графіка потоку керування (CFG). CFG – це графічне представлення всіх шляхів коду, які можуть бути пройдені під час виконання програми. Вузли на цьому графіку представляють фрагменти коду без будь-яких стрибків. Спрямовані ребра між вузлами представляють стрибки в потоці керування. На рисунку 3.2 наведено приклад такої CFG.

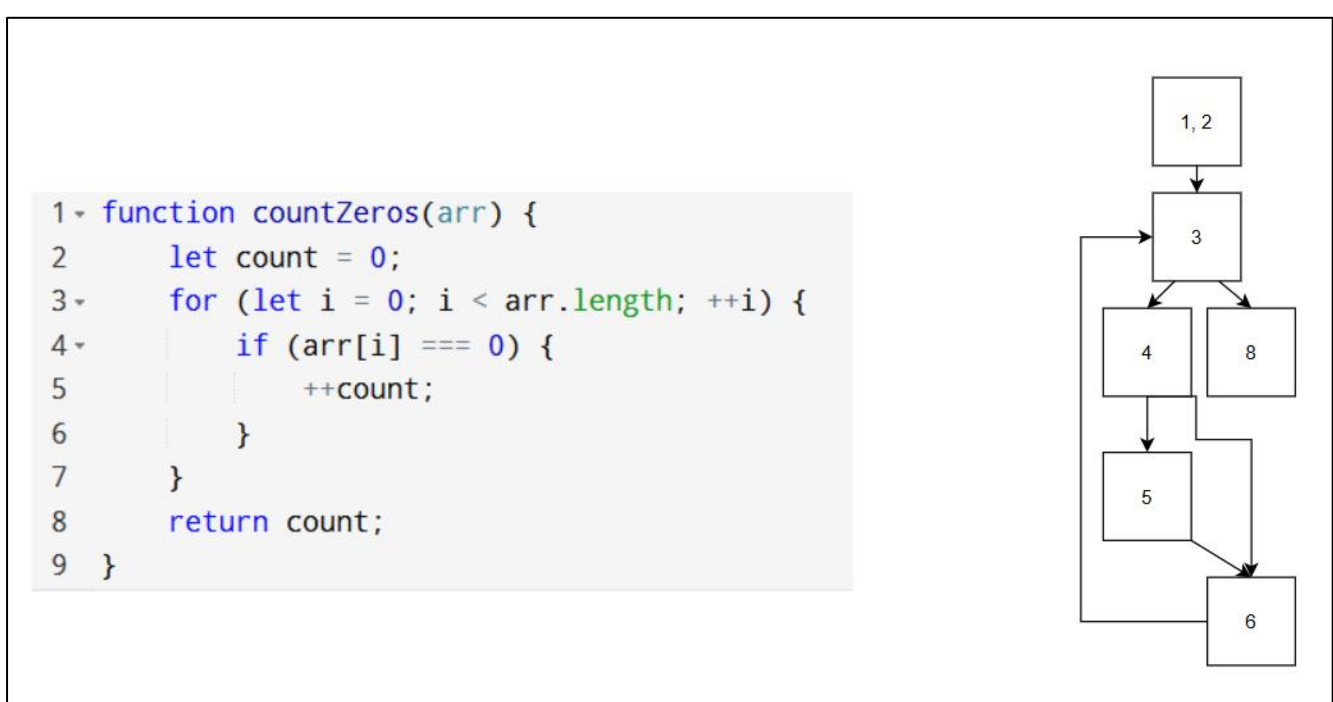


Рисунок 3.2 – Приклад коду для керування трансляцією графа потоку.

Показано вихідний код із пронумерованими рядками та показано результуючу CFG із відповідними номерами рядків для кожного вузла. У рядку 3 зустрічаємо першу точку розгалуження, цикл for. У CFG дві гілки виходять з вузла 3. Крайня ліва гілка – це випадок, коли i < arr.length, і таким чином входимо в цикл for та потрапляємо в рядок 4. Крайня права гілка, по суті, є «хибною» гілкою, де не входимо в цикл for і таким чином безпосередньо переходимо до оператора return у рядку 8.

Згенерований CFG використовується в процесі пошуку, щоб визначити, наскільки далекий певний тестовий приклад від покриття певної гілки. Робимо це, дивлячись на те, які гілки охоплює тест. Наприклад, скажімо, ми виконуємо тест X для функції countZeros. Результати покриття показують, що рядки 1, 2, 3 і 8

охоплено, тобто цикл for не було введено. Тепер хочемо дізнатись, як далеко ми до покриття лінії 5. Робимо це, дивлячись на CFG. Спочатку обчислюємо найкоротші шляхи від кожного з охоплених вузлів/ліній до вузла 5. Важливо відзначити, що ці шляхи спрямовані, тобто в прикладі неможливо пройти від 8 до 3. Далі, з усіх найкоротших шляхів, йдемо найкоротшими. У цьому випадку найкоротший шлях від охопленого вузла до 5 – це шлях 3, 4, 5, який має довжину 2. Ця метрика називається рівнем підходу.

Щоб краще зрозуміти що таке граф управління потоком, розглянемо псевдокод на рисунку 3.3.

```
1  function example(x):
2      if x > 0:
3          print("Positive")
4      else:
5          print("Non-positive")
6      print("Done")
```

Рисунок 3.3 – Псевдокод.

За допомогою розглянутого псевдокоду можемо вивести наступний граф:

- вузол 1: початок функції;
- вузол 2: перевірка умови x > 0;
- вузол 3: друк Positive (гілка if);
- вузол 4: друк Non-Positive (гілка else);
- вузол 5: друк Done.

Граф включає ребра:

- від вузла 1 до вузла 2;
- від вузла 2 до вузла 3 (якщо x > 0);
- від вузла 2 до вузла 4 (якщо x ≦ 0);
- від вузла 3 та вузла 4 до вузла 5.

Такий граф дозволяє досить точно бачити те як керування переходить між різними частинами коду.

Далі розглянемо цільовий відвідувач (Target Visitor). Витягаючи всі класи, методи та функції, цільовий відвідувач збирає можливі цілі з AST. Створюємо тести лише для цілей, експортованих у вихідний код. Отримані цілі містять інформацію про область класу/методу/функції, необхідні параметри та параметр повернення.

Відвідувач типів (Type Visitor) відповідає за пошук типів, визначених користувачем. Ці визначені користувачем типи використовуються для визначення того, які елементи повинні мати такі типи. Type Visitor витягує класи, визначені користувачем об'єкти, прототипні функції та інтерфейси. Результатом вилучення є набір «складних об'єктів». Важливо зазначити, що Type Visitor не дивиться виключно на вихідний код блоку, що тестується. Замість цього він аналізує всю кодову базу, в якій знаходиться код блоку що тестується.

Відвідувач елементів і відношень. Окрім відвідувача типу, у нас також є відвідувач елемента та зв'язку. Цей відвідувач збирає всі елементи та зв'язки, до яких він залучений, з абстрактного синтаксичного дерева. Таким чином, наприкінці цього процесу відвідування, маємо повний набір усіх взаємодій між змінними та константами.

Наступною групою компонентів є група виведення типу (Type Inference). Хоча висновок типу тісно пов'язаний зі статичним аналізом, він вимагає власної групи компонентів. Ця група складається з Type Resolver та Execution Information Integrator, як раніше показано. Найважливішим артефактом у цій групі компонентів є карта ймовірності типу, яка створюється інструментом розпізнавання типів і пізніше змінюється інтегратором інформації про виконання. По суті, розпізнавач типів виконує початковий статичний висновок типу, тоді як інтегратор інформації про виконання виконує динамічний висновок типу.

Розпізнавач типів (Type Resolver) використовує результати статичного аналізу, пов'язаного з типом, відвідувач типу та відвідувач елемента та відношення. Використовуються ці результати для визначення типів елементів. Першим кроком засобу визначення типів є визначення типів кожного з примітивних елементів. Цей крок відносно простий, оскільки очевидно, що, наприклад, рядковий примітив має тип string. Однак це вирішальний крок, щоб зробити мережу висновків завершеною.

Далі визначаємо відносини, що робиться в два кроки. Спочатку намагаємося зробити висновок про залучені елементи відношення. Це робиться шляхом припущень щодо певних відносин. Наприклад, відношення порівняння, ймовірно, включає числові елементи. Однак це не завжди очевидне, оскільки JavaScript дуже гнучкий щодо того, які типи елементів можна використовувати в будь-якому відношенні. Іншим прикладом може бути відношення рівності. У цьому випадку очікується, що два елементи будуть одного типу. Таким чином, ми робимо ймовірності типу двох елементів слабо залежними один від одного. Другим кроком розв'язання відносин є вирішення самих фактичних відносин. Іноді можна зробити висновок про результуючий тип відношення, не знаючи типів залучених елементів. Наприклад, відношення typeof завжди повертає рядок. З іншого боку, тип відношення повернення дорівнює типу повернутого елемента.

Далі вирішуємо «складні» елементи. Вони складаються з усіх елементів, які є об'єктом у відношенні Property Accessor. Тепер, порівнюючи властивості, доступ до яких здійснюється в межах елемента, з властивостями «складних об'єктів», знайдених за допомогою Type Visitor, можемо зробити висновок, який тип «складного об'єкта» належить до якого «складного» елемента. Крім того, завжди створюємо один анонімний складний тип об'єкта, який відповідає властивостям, до яких здійснюється доступ. Це робиться таким чином, що в карті ймовірності типу завжди присутній можливий тип, який можна використовувати для тестових випадків.

Як пояснювалося раніше, весь імовірнісний висновок використовує систему підрахунку балів, де за кожним натяком на те, що елемент належить до певного типу, призначаються бали цього типу на карті ймовірностей типу елемента. Після завершення статичного аналізу ми використовуємо бали для розрахунку ймовірності кожного типу. Кінцевим результатом висновку є карта ймовірності типу для кожного елемента. Ця карта ймовірностей складається з ймовірностей для кожного можливого типу. Імовірніші типи мають вищу ймовірність.

Інтегратор інформації про виконання є другим компонентом, який використовується для визначення типу. Він відповідає за коригування карти

ймовірності типу кожного елемента, коли стає доступною нова інформація про виконання. Ця інформація про виконання надається шляхом виконання тестів під час процесу пошуку. Більшість типів у карті ймовірності типу елемента насправді не застосовуються. Це пояснюється тим, що елементи часто мають лише один дійсний тип, тоді як карти ймовірностей можуть містити декілька. Наприклад, елемент X може мати карту ймовірності з ймовірністю 0,3 для рядка та 0,7 для числа. У JavaScript можна мати код, який дозволяє X бути рядком або числом. Однак найчастіше творець цього коду передбачав, що X буде лише одним із двох. Хоча JavaScript є дуже гнучким, використання складніших типів може призвести до неочікуваної поведінки, якщо використовується неправильний тип. Код може, наприклад, викликати помилку TypeError. Під час процесу пошуку ці TypeErrors виловлюються та використовуються для підвищення точності ймовірностей у карті ймовірностей типу. Поточна версія інструменту робить це шляхом сканування повідомлень про помилки на предмет імен параметрів, які використовує тестоване програмне забезпечення. Якщо є збіг, припускаємо, що тип, використаний для цього параметра, ймовірно, неправильний. Таким чином, надаємо цьому типу негативний бал для оцінки виконання.

Як згадувалося в другому розділі, пошуковий процес генерації тестових випадків з використанням методів, заснованих на пошуку, покладається на покриття гілок як керівництво для створення достатнього набору тестів. Нам потрібно знати, які гілки оцінюються під час певного виконання, щоб відстежувати покриття гілок. На рисунку 3.4 представлений приклад вихідного коду.

```
1 function max(a, b) {
2     if (a < b) {
3         return a;
4     }
5     else {
6         return b;
7     }
8 }
```

Рисунок 3.4 – Вихідний код.

Ми можемо зробити це, змінивши оригінальний вихідний код, покриття якого хочемо відстежувати. Вихідний код, який складається з простої функції, яка повертає максимальне значення двох заданих аргументів a і b. У цьому прикладі оператор if створює рівно одну гілку. На рисунку 3.5 ви можете побачити результуючий інструментальний код.

```
1  function max(a, b) {
2      track('function', 0);
3      record(0, 'a < b', [a, b]);
4      if (a < b) {
5          track('branch', 0, true);
6          return a;
7      }
8      else {
9          track('branch', 0, false);
10         return b;
11     }
12 }
```

Рисунок 3.5 – Інструментальний код.

У рядку 2 здійснюється перший виклик нашої функції треку. Він відстежує виклик функції «0». Так само в рядку 5 відстежується гілка «0». Крім того, ми відстежуємо, що «справжня» частина цієї гілки була виконана. У рядку 8 знову бачимо відстеження гілки «0», але цього разу виконується «хибна» частина гілки. Окрім покриття розгалужень, також записуємо умови розгалуження та значення залучених елементів у міру їх оцінки. У рядку 3 показуємо приклад цього. Тут записуємо умову та значення елементів в умові гілки «0». Ця додаткова інформація дозволяє розрахувати відстань розгалуження на додаток до рівня наближення. Для створення інструментального коду використаємо старішу версію istanbul.js [40]. Однак, оскільки istanbul.js не призначений для автоматизованих інструментів генерації тестів, довелося внести деякі зміни, щоб створити версію, яка відповідає вимогам нашого дослідження. Наприклад, оригінал не розглядає петлі як

розгалуження. Це створює проблему, коли рівень підходу є не зовсім правильним, що дає алгоритму пошуку менше вказівок. Іншою модифікацією є додавання вищезгаданого запису умов розгалуження та значень елементів.

Метою дослідження є генерація тестів, що означає, спробу синтезувати код за допомогою генетичного алгоритму. Це називається генетичним програмуванням [41]. Однак генетичне програмування – це більше, ніж просто мутація рядка, поки він не сформує тестовий приклад. Замість цього використовуємо кодування, яке описує, як має бути відформатований тестовий приклад. Оскільки код можна представити у вигляді дерева (наприклад, AST), має сенс також представити кодування тестів у вигляді дерева. Це кодування є деревом операторів. На рисунку 3.6 подано приклад такого дерева.

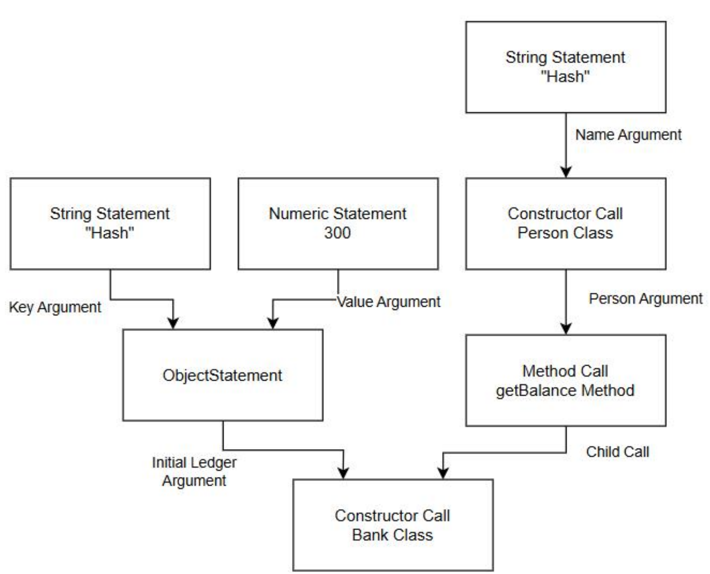


Рисунок 3.6 – Приклад дерева кодування.

Кодування поділяється на чотири класи. Перший клас – це примітивний клас, який складається з операторів Bool, Null, Numeric, String і Undefined. Цей клас містить основні будівельні блоки кодування. Другий клас є складним класом, що складається з операторів Array, Object і Arrow Function. Ці оператори є більш складними, оскільки вони можуть мати дочірні оператори. Оператор масиву, наприклад, може бути порожнім, але також може бути заповнений, наприклад, числовими операторами. Оператор об'єкта завжди має дочірні оператори в парах. Один є ключем, а інший є значенням. Оператор ключа завжди має бути рядком.

Оператор функції стрілка має лише оператор повернення в поточній реалізації. Зверніть увагу, що ці складні оператори можуть, поряд з простими операторами, також мати складні нащадки. Третій клас є кореневим класом. Він складається з оператора виклику функції та оператора виклику конструктора. Називаємо цей клас кореневим класом, оскільки корінь дерева кодування завжди є конструктором або викликом функції. Обидва ці твердження можуть містити аргументи. Як дочірні оператори складних операторів, оператори аргументів використовуються для надання аргументів, необхідних для виклику конструктора або функції. Виклик конструктора відрізняється від виклику функції тим, що він може мати дочірні елементи поряд з необхідними аргументами. Ці додаткові нащадки складаються з викликів методів із четвертого й останнього класу. Цей останній клас є класом дій і складається лише з оператора виклику методу. Ці виклики методів є нащадками оператора виклику конструктора. По суті, це виклики методів створеного класу.

У нижній частині дерева знаходимо виклик конструктора як корінь. Тут клас Bank створюється за допомогою одного аргументу. Цей аргумент є оператором об'єкта, що містить одну пару ключ-значення. Виклик конструктора також містить один оператор виклику методу під назвою getBalance. Для цього виклику методу потрібен один аргумент типу Person, який також є класом і, таким чином, може бути створений іншим викликом конструкції. Для класу Person потрібен єдиний аргумент, ім'я Person, яке є оператором рядка. Декодований тестовий випадок прикладу дерева кодування показаний на рисунку 3.7.

```
1  it("getBalance method", () => {
2      const initialBalance = 300
3      const ledger = {
4          "Hash": 300
5      }
6      const bank = new Bank(ledger)
7
8      const name = "Hash"
9      const person = new Person(name)
10
11     const balance = bank.getBalance(person)
12 })
```

Рисунок 3.7 – Декодований тестовий випадок.

Вибір цих дерев кодування відбувається за допомогою компонента Encoding Sampler. Алгоритм пошуку постійно використовує цей компонент. Спочатку шляхом вибірки початкової генеральної сукупності, а пізніше, коли потрібно буде вибрати нові кодування, щоб зберегти високу різноманітність. Деревоподібна структура кодування дозволяє дискретизатору повторно дискретизувати все дерево або піддерева кодування. Програма кодування використовує цілі, надані цільовим відвідувачем, у поєднанні з картами ймовірності типу для вибірки дерев кодування з відповідними типами. Хоча метою є створення тестів із типами, які відповідають вихідному коду, цікаво час від часу використовувати неправильні типи. Наприклад, використання неправильного типу аргументу для певного виклику функції може призвести до несподіваного виконання, яке мало бути можливим, що призведе до виявлення помилки. Знаходження таких помилок може бути дуже цінним. З цієї причини вибірка відбирає випадковий тип із малою ймовірністю, визначеною параметром ймовірності випадкового типу.

Для того, щоб розвинути популяцію кодувань до набору корисних тестових випадків, тобто набору тестових випадків із високим покриттям гілок, кодування потрібно оцінити так, щоб найбільш перспективні кодування могли перейти до наступного покоління. Щоб виконати цю оцінку, надсилаємо кодування до програми Test Case Runner, яка використовує декодер кодування для перетворення дерева кодування на текстове представництво. Після цього програма для виконання тестів переходить до виконання тесту з інструментальним вихідним кодом. Після виконання тесту інформація про виконання повертається до компонента Encoding Evaluation. Як згадувалося раніше, ця інформація потім поєднується з CFG для розрахунку рівня наближення та відстані розгалуження кодування.

Коли бюджет алгоритму пошуку вичерпується, він створює архів кодування. Цей архів складається з усіх кодувань, які є ключовими охоплювати певні гілки або спричиняти унікальні збої. Потім передаємо архів до Test Suite Builder, який використовує архів і Encoding Decoder для створення остаточного набору тестів. На додаток до просто поєднуючи всі тести разом, Test Suite Builder також створює всі твердження, які повинен мати правильний тестовий приклад. Ці твердження

складаються з перевірок значень, що повертаються під час виконання викликів функцій і методів. Нарешті, якщо під час виконання тесту виникають винятки, оператор невдачі стверджується, щоб створити виняток.

3.3 Вибір технологій

Спочатку варто визначитись з тим як саме має виглядати розроблюваний інструмент. У нашому випадку потрібне щось просте та орієнтоване на розробників програмного забезпечення, користувачів, яким потрібні прості та гнучкі інструменти з можливістю визначати різні налаштування досить просто. Для даного випадку чудово підійде інтерфейс командного рядка (CLI).

Розробники зможуть використовувати розроблюваний інструмент через простий у користуванні інтерфейс командного рядка (CLI). Інтерфейс командного рядка має широкі можливості конфігурації та пропонує ряд опцій. Основними характеристиками CLI є:

– пряме управління: користувач вводить текстові команди, програма відображає результати у вигляді тексту;

– простота виконання складних задач: легко автоматизувати завдання за допомогою скриптів, більша гнучкість у порівнянні з графічними інтерфейсами користувача;

– мала ресурсозатратність: не потребує графічних елементів, що дозволяє працювати навіть на слабкому обладнанні;

– швидкість: досвідченні користувачі можуть виконувати завдання значно швидше).

Серед існуючих прикладів CLI, які активно використовуються як розробниками програмного забезпечення так й іншими людьми в сфері IT є:

– Unix/Linux Shell: Bash, Zsh;

– Windows Command Prompt або Power Shell;

– Інструменти розробників: Git, Node.js, Docker, Python CLI.

Запуск інструменту надасть купу інформації щодо процесу. Спочатку надається деяка загальна інформація, наприклад налаштування параметрів і включені файли. Далі відображається панель завантаження, яка вказує на те, наскільки просунулися з покриттям. Поруч із цією смужкою вказано бюджет часу, що залишився. Нарешті, після завершення процесу пошуку надається невеликий огляд розглянутих гілок, операторів і функцій.

В даному дослідженні будемо використовувати мову програмування TypeScript, оскільки програмувати саме в цій мові дещо надійніше чим в чистому JavaScript. TypeScript – це по суті JavaScript із додатковими можливостями для створення більш стабільного та легкого у підтримці коду.

Серед переваг TypeScript можна визначити наступне:

– статична типізація: дозволяє визначати типи змінних на етапі компіляції, зменшуючи кількість помилок у виконанні;

– краща підтримка в IDE: сучасні середовища розробки, як-от Visual Studio Code, забезпечують розширену автодопомогу та перевірку помилок;

– інтерфейси та типи: дає змогу створювати складні структури даних та контракти, що зручно у великих проектах;

– масштабованість: легше підтримувати та розширювати великий код;

– сучасні функції: впроваджує нові можливості ECMAScript раніше, ніж вони підтримуються в JavaScript;

– екосистема підтримує інтеграцію з такими інструментами, як Webpack, Babel тощо.

Також наведено основні відмінності JavaScript та TypeScript у таблиці 3.1.

Таблиця 3.1 – Відмінності JavaScript та TypeScript.

Функція	**TypeScript**	**JavaScript**
Типізація	Статична (обов’язкова або необов’язкова)	Динамічна

Продовження таблиці 3.1.

Функція	**TypeScript**	**JavaScript**
Компіляція	Потребує компіляції у JavaScript	Не потребує компіляції
Виявлення помилок	На етапі компіляції, менше помилок у виконанні	Помилки виявляються лише під час виконання
Поріг входу	Вищий, особливо для новачків	Низький, чудовий для швидкого старту
Призначення	Великі проекти, командна робота, масштабованість	Невеликі проекти, швидка розробка

Програмний засіб буде розроблений за допомогою Node.js. Це потужне середовище виконання, яке часто використовується JavaScript розробниками для виконання різних задач.

Серед ключових особливостей Node.js варто зазначити:

– неблокуючий ввід/вивід: події керуються асинхронною архітектурою для обробки багатьох запитів одночасно;

– масштабованість: ідеально підходить для додатків, які потребують обробки великої кількості з'єднань (наприклад, чати або ігри в реальному часі);

– багата екосистема: npm (Node Package Manager) пропонує тисячі бібліотек та модулів для розширення функціональності;

– кросплатформеність: працює на основних операційних системах, таких як Windows, macOS та Linux;

– єдина мова: можливість використовувати JavaScript і на клієнтській, і на серверній стороні.

В програмному засобі буде включено підтримку CommonJS та ECMAScript модулів Node.js. В цілому надаємо перевагу ECMAScript, тому будемо використовувати саме його у своїх прикладах.

Для розробки був обраний об’єктно-орієнтованих підхід та було використано різні шаблони проектування. Це допоможе в майбутньому змінити та допрацювати різні компоненти максимально зручно з мінімальними затратами на переписування коду.

3.4 Висновки

В даному розділі було розглянуто алгоритми, архітектуру та в цілому структуру основних компонентів розроблюваного програмного забезпечення. Опис елементів розпочато із загальних складових продукту та опису алгоритмів.

Спочатку описано загальні концепції та складові елементи, які потребують проектування. Серед таких елементів був огляд на абстрактне синтаксичне дерево, граф потоку управлінням та розпізнавач типів.

Далі детально спроектовано всі складові, а саме компоненти програмного продукту, та деякі з них було виділено в окремі групи. Певні компоненти відповідають тільки за статичний аналіз, в той час, як інші за виведення типів та пошукові процеси для генерації тестів. В даному етапі, також було описано загальну схему компонентів, розглянуто приклад керування трансляції графу потоку, інструментальний код, дерево кодування та декодування тестового випадку.

Вирішальним етапом було визначення того, які інструменти будуть використані для імплементації програмного засобу. На даному етапі вже можна визначити, яким має бути вигляд програмного засобу в фіналі розробки. В нашому випадку обраними інструментами є мова програмування TypeScript та середовище Node.js. Програмне забезпечення буде представлено у вигляді інтерфейсу командного рядка (CLI), оскільки воно відповідає нашим потребам в гнучкому та простому налаштуванні, а також є знайомим інтерфейсом для багатьох розробників.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Програмна реалізація

Фінальним етапом даної роботи стала імплементація програмного забезпечення. На основі обраних технологій було розроблено інтерфейс командного рядка (CLI), виконання якого можна налаштувати за допомогою різних параметрів.

Запуск інструменту надасть купу інформації щодо процесу. Спочатку надається деяка загальна інформація, наприклад налаштування параметрів і включені файли. Далі відображається панель завантаження, яка вказує на результати покриття. Поруч із цією смужкою вказано бюджет часу, що залишився. Нарешті, після завершення процесу пошуку надається невеликий огляд розглянутих гілок, операторів і функцій.

Після того, як інструмент буде готовий, результати будуть розташовані в каталозі test. Цей каталог містить журнали процесу, статистику виконання та згенеровані тестові приклади. Приклад такого тесту показано в рисунку 4.1. Ці тести відповідають формату. Імена змінних мають *формат_<назва аргументу>_<тип>_<випадковий рядок>*. Як показано, результати викликів функцій перевіряються за допомогою тверджень. Коли очікується помилка, ми загортаємо виклик функції в блок try-catch.

```
1  it('test for detectUndirectedCycle ', async () => {
2      const _isDirected_boolean_WdyT = true;
3      const _graph_Graph_lk5v = new Graph( _isDirected_boolean_WdyT )
4      const _vertex_undefined_5ax = undefined;
5
6      expect(JSON.parse(JSON.stringify( _graph_Graph_lk5v ))).to.deep.equal
           ({"vertices":{} ,"edges":{} ,"isDirected":true})
7
8      try {
9      const _returnValue_any_wkP1 = await _graph_Graph_lk5v.
10     getNeighbors(_vertex_undefined_5ax)
11     } catch (e) {
12         expect(e).to.be.an('error')
13     }
14 });
```

Рисунок 4.1 – Приклад згенерованого тесту.

Статистика в каталозі test створюється у двох різних форматах. Перший містить статистику, доступну в кінці експерименту. Сюди входять різні типи структурного покриття, час ініціалізації та пошуку, а також значення певних параметрів, наприклад використане випадкове початкове число. Другий тип містить кілька типів структурного охоплення для кожного покоління разом із міткою часу. Другий тип статистики дає змогу досліджувати результативність у часі.

Інтерфейс командного рядка має широкі можливості конфігурації та пропонує ряд опцій. Ці параметри детально описані в таблиці 4.1. Параметри можна надати безпосередньо як аргументи CLI або шляхом створення файлу конфігурації у кореневому каталозі свого сховища. У цьому файлі можна налаштувати кожен параметр.

Таблиця 4.1 – Параметри CLI.

Параметр	**Аргумент**	**Опис**
target_root_diretory	string	Шлях до кореневого каталогу.
include	array of strings	Множина шляхів, які додатково будуть включені.
exclude	array of strings	Множина шляхів, які необхідно виключити.
search_time	number	Кількість часу в секундах на одну тестовану одиницю.
type_inference_mode	string	Режим виведення типу
incorporate_execution_information	boolean	Включення інформації про виконання
random_type_probability	number	Імовірність випадковості

Перший параметр – це шлях до кореневого каталогу тестованого програмного забезпечення. Усе в кореневому каталозі буде використано під час статичного аналізу, наприклад, для пошуку типів. Наступним варіантом є аргумент include. Цей параметр має отримати масив шляхів до рядків або шаблонів. Ці шляхи/шаблони використовуватимуться для пошуку можливих тестованих одиниць. Опція виключення робить прямо протилежне. Він виключає будь-які тестовані одиниці, знайдені за допомогою заданих шляхів/шаблонів. Важливо зауважити, що параметр виключення переважає над параметром включення. Наступним є параметр часу пошуку. Він визначає бюджет за секунди на тестовану одиницю. Параметр type_inference_mode – це режим виведення типу. Цей параметр може мати одне з трьох можливих значень, а саме: none, proportional (пропорційний) та ranked (ранговий). Ці режими безпосередньо відповідають варіантам вибірки раніше, які ми раніше описали в другому розділі у таблиці 2.1. В таблиці було описано ідею пропорційної вибірки та рангової вибірки. Далі маємо опцію включення інформації про виконання (incorporate_execution_information). Ця опція вирішує, чи використовувати інформацію про виконання, отриману від виконання тестового прикладу процесу пошуку. Нарешті, параметр імовірності випадкового типу (random_type_probability) має бути десятковим числом, яке визначає ймовірність того, що випадковий тип використовується замість виведеного типу.

Перейдемо до того що розглянемо деякі з аспектів реалізації. Спершу почнемо з частини статичного аналізу, а саме абстрактного синтаксичного дерева. Для простої та надійної імплементації цього механізму було використано бібліотеку Babel.

Babel – це популярний компілятор JavaScript, який дозволяє використовувати сучасні можливості JavaScript (ES6+ синтаксис) вже сьогодні, навіть якщо ваші середовища (наприклад, браузери або Node.js) не підтримують їх. Babel трансформує ваш код у сумісну версію, що працює у старіших середовищах.

Основні можливості включають:

– транспіляція: перетворює код із сучасного JavaScript (ES6+) у старіші версії (ES5);

– підтримка API: Додає функціональність (за допомогою core-js та інших бібліотек);

– плагіни та пресети: Дозволяють гнучко налаштовувати функціонал;

– підтримка експериментальних функцій: Ви можете використовувати функції JavaScript, які ще не стандартизовані.

Ця бібліотека цікава тим, що вона ефективно може перетворити JavaScript код у абстрактне синтаксичне дерево. Для цього нам спершу потрібно створити конфігураційний файл, назвемо його defaultBabelConfig.ts. Вміст даного файлу показаний на рисунку 4.2.

```
1  import { TransformOptions } from "@babel/core";
2
3  export const defaultBabelOptions: TransformOptions = {
4    configFile: false,
5    babelrc: false,
6    ast: true,
7    sourceMaps: true,
8    compact: false,
9    comments: true,
10   parserOpts: {
11     allowReturnOutsideFunction: true,
12     sourceType: "module",
13     plugins: [
14       "asyncGenerators",
15       "classProperties",
16       "dynamicImport",
17       "objectRestSpread",
18     ],
19   },
20   plugins: [],
21 };
```

Рисунок 4.2 – Файл конфігурації для Babel.

Далі використавши цю конфігурацію, створимо клас фабрики абстрактного синтаксичного дерева. Повний вміст класу зображений на рисунку 4.3. За

допомогою такої нескладної маніпуляції, маємо можливість працювати з абстрактним синтаксичним дерево. В майбутньому залишається лише пройтись по всіх його гілкам.

```
1  import { transformSync } from "@babel/core";
2  import * as t from "@babel/types";
3  import { AbstractSyntaxTreeFactory as FrameworkAbstractSyntaxTreeFactory } from
       "@syntest/analysis";
4  import { Result, success } from "@syntest/diagnostics";
5
6  import { defaultBabelOptions } from "./defaultBabelConfig";
7
8  export class AbstractSyntaxTreeFactory
9    implements FrameworkAbstractSyntaxTreeFactory<t.Node>
10 {
11   convert(filepath: string, source: string): Result<t.Node> {
12     const options: unknown = JSON.parse(JSON.stringify(defaultBabelOptions));
13
14     const codeMap = transformSync(source, options);
15
16     return success(codeMap.ast);
17   }
18 }
```

Рисунок 4.3 – Фабрика абстрактного синтаксичного дерева.

Далі дещо оглянемо фабрику ControlFlowGraph. Повний код зображений на рисунку 4.4. Даний клас має один основний метод для використання, а саме convert, який приймає в якості аргументів шлях до файлу та абстрактне синтаксичне дерево. Основний алгоритм описаний в класі ControlFlowVisitor (відвідувач), він виконує обхід вузлів, за допомогою раніше переданого абстрактного синтаксичного дерева, в результаті чого вихідним результатом буде можливість контролювати потік управління в програмі. В результаті маємо такий собі граф з вузлами та ребрами, де вузлами можуть бути перевірки умов, виклик функції і ще багато всього іншого, а ребрами є перехід від виконання однієї функції до іншої.

```
1  import { traverse } from "@babel/core";
2  import * as t from "@babel/types";
3  import { ControlFlowGraphFactory as FrameworkControlFlowGraphFactory } from
       "@syntest/analysis";
4  import { contractControlFlowProgram, ControlFlowProgram } from "@syntest/cfg";
5  import { Result } from "@syntest/diagnostics";
6
7  import { Factory } from "../Factory";
8
9  import { ControlFlowGraphVisitor } from "./ControlFlowGraphVisitor";
10
11 export class ControlFlowGraphFactory
12   extends Factory
13   implements FrameworkControlFlowGraphFactory<t.Node>
14 {
15   convert(filePath: string, AST: t.Node): Result<ControlFlowProgram> {
16     const visitor = new ControlFlowGraphVisitor(filePath, this.syntaxForgiving);
17     traverse(AST, visitor);
18
19     return contractControlFlowProgram(visitor.cfg);
20   }
21 }
```

Рисунок 4.4 – Фабрика графу управління потоком.

Для інструментального коду нам знадобляться деякі класи, щоб описати карту локацій, гілок, виразів тощо.

Спершу розглянемо клас Location та StatementMap зображені на рисунку 4.5.

```
1  export type Location = {
2    id: string;
3    start: {
4      line: number;
5      column: number;
6      index: number;
7    };
8    end: {
9      line: number;
10     column: number;
11     index: number;
12   };
13 };
14
15 export type StatementMap = {
16   [id: string]: Location;
17 };
```

Рисунок 4.5 – Клас локації та виразу.

Клас Location містить в собі початок та кінець, які в свою чергу є номером рядка, номером стовпчика та індексом. Клас StatementMap описує якийсь вираз, частинку кода і містить в собі відповідно множину типу локації (Location), де кожна локація має відповідний унікальний текстовий ідентифікатор. Результатом є те що StatementMap працює по принципу хеш-таблиці для Location. За відповідною логікою оголошуємо класи BranchMap та FunctionMap, тобто карта для гілок та функцій відповідно до такої самої мети як в класі StatementMap. Клас BranchMap та FunctionMap зображені на рисунку 4.6.

```
1  export type BranchMap = {
2    [id: string]: {
3      line: number;
4      type: string;
5      loc: Location;
6      locations: [Location, Location];
7    };
8  };
9
10 export type FunctionMap = {
11   [id: string]: {
12     name: string;
13     line: number;
14     decl: Location;
15     loc: Location;
16   };
17 };
```

Рисунок 4.6 – Клас BranchMap та FunctionMap.

Далі описуємо класи даних інструментації (InstrumentationData) та карту інструментації (InstrumentationDataMap), які зображені на рисунку 4.7. Екземпляр інструментальних даних агрегує в собі хеш, карту виразу, карту гілок та карту функцій. Карта інструментальних даних відповідно є хеш-таблицею інструментальних даних, де в якості ідентифікатора буде тестовий рядок, який вказує на шлях.

```
1  export type InstrumentationData = {
2    hash: string;
3    statementMap: StatementMap;
4    branchMap: BranchMap;
5    fnMap: FunctionMap;
6    s: {
7      [id: string]: number;
8    };
9    f: {
10     [id: string]: number;
11   };
12   b: {
13     [id: string]: number[];
14   };
15 };
16
17 export type InstrumentationDataMap = {
18   [path: string]: InstrumentationData;
19 };
```

Рисунок 4.7 – Класи InstrumentationData та InstrumentationDataMap.

Все раніше описане пізніше знадобиться нам в під час динамічного аналізу, коли доведеться відслідковувати стан виконання коду. Для цього нам знадобиться інструментувати JavaScript код за допомогою наступної функції зображеної на рисунку 4.8. Самими поширеними та простими прикладами інструментації коду є логування та трасування, що широко використовується розробниками під час створення програмного забезпечення. Візьмемо всі шляхи до вхідних файлів та розпочнемо їх обходити, інструментуючи, тобто додаючи свої сторонні функції. В результаті будуть створені модифіковані копії вхідного коду, запустивши який, почнемо збирати інформацію про те як і що виконується.

```
async instrumentAll(
    storageManager: StorageManager,
    rootContext: RootContex,
    targets: Target[],
    instrumentedDirectory: string
): Promise<void> {
  const absoluteRootPath = path.resolve(rootContext.rootPath);
  const destinationPath = path.join(instrumentedDirectory, path.basename(absoluteRootPath)
  );
  storageManager.copyToTemporaryDirectory(
    [absoluteRootPath], [...destinationPath.split(path.sep)]
  );
  const targetPaths = [...targets.values()].map((target) => target.path);
  for (const targetPath of targetPaths) {
    const result = rootContext.getSource(targetPath);
    if (isFailure(result)) continue;
    const instrumentedSource = await this.instrument(
      unwrap(result),
      targetPath
    );
    const _path = path
      .normalize(targetPath)
      .replace(absoluteRootPath, destinationPath);
    const directory = path.dirname(_path);
    const file = path.basename(_path);
    storageManager.store(
      [...directory.split(path.sep)],
      file,
      instrumentedSource,
      true
    );
  }
}
```

Рисунок 4.8 – Функція інструментації вхідного JavaScript коду.

Далі опишемо функцію створення тестів (Рисунок 4.9). В самому процесі приймають участь досить багато складових, але все ж спробуємо описати це більш простим прикладом. На вхід функції подається шлях до каталогу з вхідним кодом, та шлях до каталогу де мають бути згенеровані тести. Обходимо всі вхідні файли, та для кожного з них намагаємось створити тест. Відповідно маючи файл example.js, отримаємо на виході файл test-example.spec.js. В звичайному була б спроба передати випадкові типи даних в тестовий випадок, але оскільки ми використовуватимемо

статичний та динамічний аналіз, точність типів даних, які будуть використані в тестування дещо більшою.

```
createSuite(
    archive: Map<Target, JavaScriptTestCase[]>,
    sourceDirectory: string,
    testDirectory: string,
    gatherAssertionData: boolean,
    compact: boolean,
    final = false
  ) {
    const paths: { [key: string]: number } = {};
    for (const target of archive.keys()) {
      if (compact) {
        const decodedTest = this.decoder.decode(
          archive.get(target), gatherAssertionData, sourceDirectory);
        const testPath = this.storageManager.store(
          [testDirectory], `test-${target.name}.spec.js`, decodedTest, !final);
        if (paths[testPath] !== undefined) {
          throw new ImplementationError("Should only be one of each path!");
        }
        paths[testPath] = archive.get(target).length;
      } else {
        for (const testCase of archive.get(target)) {
          const decodedTest = this.decoder.decode(testCase, gatherAssertionData, sourceDirectory);
          const testPath = this.storageManager.store([testDirectory], `test-${target.name}-${testCase
              .id}.spec.js`, decodedTest, !final);
          if (paths[testPath] !== undefined) {
            throw new ImplementationError("Should only be one of each path!");
          }
          paths[testPath] = 1;
        }
      }
    }
    return paths;
  }
```

Рисунок 4.9 – Функція створення тестів.

Всі розглянуті раніше ідеї з попереднього розділу належним чином імплементовані, а це означає що можемо вирушати до наступного етапу.

4.2 Розробка тестових сценаріїв

Тестування програмного забезпечення буде проводитись на основі контрольних показників Було створено бенчмарк для оцінки ефективності задуманого підходу. Цей тест складається з п’яти проектів, а саме:

– express, веб-фреймворк для Node.js;

– commander, структура командного рядка для Node.js;

– moment, бібліотека JavaScript для аналізу, перевірки, обробки та форматування дат;

– algorithms, бібліотека, що містить популярні алгоритми та структури даних;

– lodash, бібліотека JavaScript, яка надає службові функції для звичайних завдань програмування.

Ці проекти були обрані на основі кількості зірок на GitHub або щотижневих завантажень із Node Package Manager у спільноті JavaScript. Крім того, обрані проекти представляють різноманітний набір синтаксису та стилів коду, що позитивно вплине на результати тестування.

Далі вручну відібрано підмножину файлів із еталонних проектів, які будуть використані для цієї оцінки. Файли було обрано на основі кількох факторів. По-перше, файл повинен містити щось, що можна перевірити, тобто експортовану функцію або клас. По-друге, файл повинен мати цикломатичну складність (Cyclomatic Complexity, CC) принаймні 2. Оскільки нижча складність означає менше лінійно незалежних шляхів через вихідний код, нам, швидше за все, знадобиться менше тестів для досягнення високого структурного покриття [42]. Хоча для цього дослідження можна зробити висновок, що досягнення високого рівня охоплення у файлах з нижчою складністю все ще є нетривіальним, оскільки іноді це потребує правильно визначених типів. З цієї причини для великих проектів із великою кількістю файлів було обрана підмножина файлів із діапазоном значень складності.

Дані про тестові бенчмарки занесені в таблиці. В даних таблицях описаний бенчмарк, для якого буде проводитись тестування та його файли. Також описана кількість тестованих одиниць, складність, кількість рядків коду та кількість гілок. Статистика порівняльного аналізу для Commander.js описано в таблиці 4.2.

Таблиця 4.2 – Статистика порівняльного аналізу Commander.js.

Бенчмарк	Файл	Кількість тестованих одиниць	Складність	Кількість рядків коду	Кількість гілок
Commander.js	Help.js	1	50	406	66
	Options.js	2	20	324	18
	suggestSimilar.js	1	21	100	32

Далі в таблиці 4.3 описана точно така сама статистика, але для Express.

Таблиця 4.3 – Статистика порівняльного аналізу для Express.

Бенчмарк	Файл	Кількість тестованих одиниць	Складність	Кількість рядків коду	Кількість гілок
Express	query.js	1	5	47	6
	layer.js	1	17	181	22
	route.js	1	23	225	30
	application.js	1	42	661	52
	request.js	1	35	525	44
	response.js	1	133	1169	174
	utils.js	7	28	304	34
	view.js	1	14	182	16

Наступним кроком було зроблено статистику порівняльного аналізу для бібліотеки moment.js. Всі подробиці вказані у таблиці 4.4

Таблиця 4.4 – Статистика порівняльного аналізу для moment.js.

Бенчмарк	Файл	Кількість тестованих одиниць	Складність	Кількість рядків коду	Кількість гілок
Moment.js	query.js	1	5	47	6
	layer.js	1	17	181	22
	route.js	1	23	225	30

Продовження таблиці 4.4.

Бенчмарк	Файл	Кількість тестованих одиниць	Складність	Кількість рядків коду	Кількість гілок
Moment.js	application.js	1	42	661	52
	request.js	1	35	525	44
	response.js	1	133	1169	174
	utils.js	7	28	304	34
	view.js	1	14	182	16

В наступній таблиці описано вибрані файли з бібліотеки. Статистика зображена у таблиці 4.5.

Таблиця 4.5 – Статистика порівняльного аналізу для JavaScript алгоритмів.

Бенчмарк	Файл	Кількість тестованих одиниць	Складність	Кількість рядків коду	Кількість гілок
JavaScript Algorithm	breathFirstSearch.js	1	7	75	8
	graphBridges.js	1	5	95	8
	detectDirectedCycle.js	1	5	93	8
	detectUndirectedCycle.js	1	5	59	8
	dijkstra.js	1	6	80	10
	eulerianPath.js	1	9	101	14
	floydWarshall.js	1	4	72	6
	hamiltonianCycle.js	1	6	134	10
	kruskal.js	1	6	62	10
	prim/prim.js	1	8	73	12
	stronglyConnectedComponents.js	1	5	133	8
	bfTravellingSalesman.js	1	7	104	14
	Knapsack.js	1	24	195	40
	CountingSort.js	1	9	78	14
	Matrix.js	12	26	309	38
	RedBlackTree.js	1	22	323	34

У фінальній таблиці 4.6 наведено файли та статистику бібліотеки lodash. Дана бібліотека містить все необхідне для вирішення повсякденних завдань програмування. Було обрано найбільш релевантні файли з достатньою кількістю рядків коду та розгалужень

Таблиця 4.6 – Статистика порівняльного аналізу для бібліотеки lodash.

Бенчмарк	Файл	Кількість тестованих одиниць	Складність	Кількість рядків коду	Кількість гілок
Lodash	equalArrays.js	1	19	84	24
	hasPath.js	1	11	53	8
	random.js	1	11	73	14
	result.js	1	6	53	10
	slice.js	1	11	47	20
	split.js	1	9	42	8
	toNumber.js	1	12	65	20
	transform.js	1	10	59	12
	truncate.js	1	19	113	34
	unzip.js	1	5	43	6

Для розрахунку цикломатичної складності було використано бібліотеку Plato, яка є інструментом для статичного аналізу вихідного коду JavaScript. На додаток до складності, надаємо кількість гілок та кількість вихідних рядків коду, також відомих як фізичні рядки коду.

Для деяких проектів, файли потрібно було виключити або змінити. Наприклад, у проекті Commander.js два файли містять оператори, які завершують роботу всієї програми. Це запланована поведінка для проекту. Однак весь інструмент завершує роботу під час створення тестового прикладу, коли досягаються ці оператори. З цієї причини такі файли слід виключити. Однак часто буває так, що файли в межах проекту залежать один від одного. Таким чином, коли ці залежності неправильно імітуються, і інструмент перевіряє пов’язаний файл, він фактично також перевіряє частини виключених файлів. У деяких випадках вихідні твердження знову

досягаються. З цієї причини було змінено проблемні файли так, щоб вони не виходили з програми. Звичайно, це не оптимальне рішення. Для майбутньої роботи довгостроковим та хорошим рішенням було б належне імітування залежності тестованого програмного забезпечення.

Як згадувалося раніше, лише файли з експортованими функціями або класами розглядалися для тестування. Ці функції та класи є тестованими одиницями, які тестує інструмент. Може виникнути питання, чому б не протестувати окремі методи класу. Функції зазвичай не мають і не змінюють станів. З іншого боку, методи класу часто змінюють стан батьківського класу. Це не стосується всього коду, але в спільноті інформатики погоджено, що методи класу повинні взаємодіяти зі станом класу. В іншому випадку вони можуть бути статичними функціями. Тому з цих причин цей підхід перевіряє або функцію лише одним викликом цієї функції, або клас, викликаючи один чи більше його методів. Зауважте, що у файлах може бути більше не експортованих функцій і класів, що збільшує цикломатичну складність, кількість гілок і кількість рядків вихідного коду.

Комплексні підходи та інструменти, подібні до розробленого під час цього дослідження, часто мають численні варіанти конфігурації у формі параметрів. Для емпіричної оцінки підходу необхідно було зробити вибір щодо значень параметрів. Розглядаємо значення параметрів, які найбільше відповідають меті дослідження. Деякі зі значень цих параметрів отримані з попередньої роботи або вибрані з певних причин і тому не змінюються. Різноманітні параметри – це ті, які допомагають відповісти на запитання дослідження та мають найбільший вплив на ефективність підходу.

Починаючи з постійних параметрів, перше, що спадає на думку, це вибір алгоритму пошуку. Ґрунтуючись на результатах відповідної роботи, можна дійти висновку, що DynaMOSA [39] є найбільш прийнятним вибором. DynaMOSA – це багатоцільовий генетичний алгоритм, який спеціалізується на створенні тестів. Це дозволяє DynaMOSA вирішувати проблеми масштабованості, які виникають під час роботи з сотнями цілей гілок. Оскільки DynaMOSA адаптована для вирішення проблем масштабованості, які виникають у інших генетичних алгоритмів із багатьма

об'єктами, їй не потрібен величезний розмір популяції для підтримки генетичного різноманіття. Загальне значення за умовчанням для розміру популяції становить 50 [39]. Тепер, оскільки кількість цілей розгалужень у файлі порівняльного тесту не надто велика, стандартний розмір сукупності 50 було визнано достатнім. Максимально допустимий час пошуку на тестовану одиницю склав 120 секунд, що є загальноприйнятим у літературі [2, 43]. З результатів ми дійшли висновку, що 120 секунд – достатній час, щоб продемонструвати можливості підходу. Еталонний тест складається з 97 тестованих одиниць, що означає, що проведення одного експерименту зайняло 3 години 14 хвилин. Крім того, через стохастичну природу підходу кожен експеримент потрібно проводити принаймні 50 разів, щоб виміряти середню продуктивність.

Щоб відповісти на запитання дослідження, деякі параметри повинні бути змінені протягом експериментів. Це дозволяє нам досліджувати вплив певних параметрів на ефективність підходу. Перший цікавий параметр – це «режим виведення типу». Як вказує назва, цей параметр визначає, який режим виведення типу використовується для визначення типів елементів. На момент написання статті існує три режими.

Перший – None, який є базовим режимом, оскільки він повністю опускає висновок типу. У базовому режимі моделі типів ігноруються, тому кожен тип має однакову ймовірність бути вибірковим.

Другий режим ранжирований, режим рангової вибірки. У режимі рангової вибірки завжди вибираємо тип із найвищою ймовірністю в моделі типу. Тобто для будь-якого елемента завжди вибираємо найімовірніший тип.

Третій і останній режим – пропорційний, режим пропорційної вибірки. У цьому режимі алгоритм вибирає випадковий тип із моделі типу на основі ймовірностей. Іншими словами якщо певний тип має високу ймовірність, він має високу ймовірність бути відібраним. На відміну від режиму рангової вибірки, у режимі пропорційної вибірки малоймовірних типів немає виключено з вибірки. Другим змінним параметром є параметр «включити інформацію про виконання». Цей параметр вимагає логічного значення та визначає, чи включає підхід

інформацію про виконання для підвищення точності виведених типів, тобто динамічного висновку типу.

Якщо розглядати схему описану в другому розділі даної роботи, то цей параметр визначає, чи активна фаза 4 чи ні. Якщо для цього параметра встановлено значення false, моделі типів базуються виключно на статичному висновку типу. Якщо цей параметр хибний, називаємо варіант підходу лише СА (статичний аналіз), якщо він істинний, називаємо варіант СА+ДА (статичний та динамічний аналіз).

4.3 Аналіз результатів

Настав аналіз результатів, отриманих в результаті тестування. Як згадувалося раніше, було проведено кожен експеримент 50 разів для розрахунку середньої продуктивності. Це необхідно через природу підходу. Під час кожного випробування, кожну із 97 тестових одиниць, розглядається окремо протягом 120 секунд. Водночас протягом цих 120 секунд кожне покоління записує кількість охоплених цілей із поточною міткою часу. Після закінчення 120 секунд остаточне покриття гілок записується окремо разом з іншими статистичними даними про пробіг.

Кожен експеримент проводився на одній системі, щоб переконатися, що апаратне забезпечення не вплинуло на результати експериментів. Відповідні специфікації налаштування системи можна знайти в таблиці 4.7.

Таблиця 4.7 – Специфікації апаратного забезпечення.

Параметр	**Назва**	**Інформація**
CPU	Intel Core i5-8300H CPU 2.30GHz (Coffee Lake)	1 CPU, 8 logical and 4 physical cores
RAM	-	16 GB
Drive	-	10 Gbs networked SSD network
OS	Ubuntu	20.04 LTS

Щоб порівняти різницю продуктивності між різними параметрами експерименту, медіана та міжквартильний діапазон (IQR) для кожного експерименту обчислюються на основі остаточного покриття гілки. Для цих експериментів медіану було обрано замість середнього, щоб запобігти спотворенню результатів через аномалії. Також повідомляємо IQR, щоб дати уявлення про поширення результатів.

На додаток до медіани та IQR важливо визначити, чи результати різних експериментів значно відрізняються один від одного. Для цього було використано непарний тест Вілкоксона зі знаком рангу [44] з порогом 0,05. Цей непараметричний статистичний тест визначає, чи суттєво відрізняються два розподіли даних. Іншими словами, якщо розподіли даних суттєво не відрізняються, їх можна взяти з одного розподілу. Критерій знакового рангу Вілкоксона поєднується зі статистикою Варґи-Делані (Vargha-Delaney) $\widehat{A}_{12}$ [45] для опису розміру ефекту результату, який визначає величину різниці між двома розподілами даних.

$\widehat{A}_{12}$ Варґи-Делані — це статистичний показник, який використовується для оцінки розміру ефекту в порівняльних дослідженнях, особливо для порядкових або непараметричних даних. Він показує ймовірність того, що випадково обране значення з однієї групи буде більшим за випадково обране значення з іншої групи. Даний статичний показник описується формулою 4.1.

$$A_{12} = \frac{W}{n_1 \cdot n_2} \tag{4.1}$$

Параметр W означає кількість випадків, коли значення з першої групи більше за значення з другої групи, плюс половина випадків, коли значення рівні.

Параметри n_1 та n_2 означають розміри вибірок для першої та другої груп відповідно.

Перейдемо до огляду результатів. Всього було представлено п’ять проектів над якими буде проводитись тестування, всі вони є бібліотеками, які виконують різні прикладні задачі за допомогою JavaScript, від звичайних побутових до складних маніпуляцій з датами та використанням різних алгоритмів, як наприклад сортування підрахунком.

Всього буде по два набори результатів на кожен проект, перший набір є медіанним охопленням гілок разом з міжквартальним діапазоном для порівняльного файлу для кожного варіанту та базової лінії. Другий набір результаті описує попарне порівняння базової лінії, рангової вибірки та варіанта пропорційної вибірки.

Під базовою лінією мається на увазі сама генерація тестів з випадковими типами без участі статичного або статичного та динамічного аналізу. В результаті зможемо порівняти те як саме виведення типу впливає на продуктивність генерації тестів, що і буде основним результатом нашого дослідження.

На даному етапі можемо визначити, який вплив на ефективність використання виведених типів порівняно з випадковими типами на тестовому охопленні, згенерованому автоматизованими інструментами створення тестових випадків. Для цього опишемо таблиці результатів.

Спершу розглянемо Commander.js. У таблиці 4.8 та таблиці 4.9 бачимо що, варіант рангової вибірки значно перевершує базову лінію лише у файлі help.js. Для інших двох файлів базова лінія перевершує варіант рангової вибірки. Варіант пропорційної вибірки перевершує як базову лінію, так і рангову вибірку для всіх трьох файлів.

У даних таблицях присутні відповідні позначення:

– M (медіана);
– IQR (міжквартильний діапазон);
– СА (статичний аналіз);
– СА+ДА (статичний + динамічний аналіз).

Таблиця 4.8 – Медіана та IQR покриття гілок для Commander.js.

Файл	**Базова лінія**		**Рангова вибірка**				**Пропорційна вибірка**			
			СА		*СА+ДА*		*СА*		*СА+ДА*	
	M	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*
help.js	0.20	0.015	0.40	0.045	0.44	0.076	0.53	0.030	0.52	0.045
option.js	0.44	0.111	0.33	0.056	0.50	0.000	0.50	0.111	0.50	0.000
suggestSimilar.js	0.66	0.219	0.55	0.156	0.55	0.188	0.75	0.031	0.75	0.062

Також в таблиці 4.9 поряд значень $\widehat{A}_{12}$ присутні позначення в дужках, які означають діапазон розміру ефекту відповідно до статичного показника $\widehat{A}_{12}$ Варґи-Делані. Маємо декілька наступних позначень латинськими літерами, а саме:

- L (значний ефект);
- M (середній ефект);
- S (маленький ефект);
- N (майже без ефекту).

Таблиця 4.9 – Попарне порівняння базової лінії, рангової вибірки та пропорційної вибірки для Commander.js.

Файл	**Базова лінія / Рангова вибірка (CA)**		**Базова лінія / Пропорційна вибірка (CA)**		**Рангова вибірка (CA) / Пропорційна вибірка (CA)**	
	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$
help.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.995 (L)
option.js	0.00	0.206 (L)	0.00	0.646 (S)	0.00	0.922 (L)
suggestSimilar.js	0.00	0.519 (N)	0.00	0.980 (L)	0.00	0.916 (L)

Для порівняльного тесту Express було виявлено, що варіант вибірки на основі рангів значно перевершує базовий рівень у файлі utils.js. В даній вибірці розглянуто шість файлів.

Базовий рівень перевершує варіант рангової вибірки на двох файлах. На інших трьох файлах різниця несуттєва. Варіант пропорційної вибірки значно перевершує базовий рівень для двох файлів. На інших чотирьох файлах різниця несуттєва. Ті самі результати можна спостерігати, порівнюючи варіант пропорційної вибірки з варіантом рангової вибірки. Відповідні результати покриття гілок та порівняння підходів занесені в таблиці. Медіани, IQR покриття гілок та порівняння підходів для проекту Express вказані у таблиці 4.10 та таблиці 4.11 відповідно.

Таблиця 4.10 – Медіана та IQR покриття гілок для Express.

Файл	Базова лінія		Рангова вибірка				Пропорційна вибірка			
			СА		*СА+ДА*		*СА*		*СА+ДА*	
	M	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*
application.js	0.63	0.019	0.62	0.019	0.63	0.019	0.65	0.019	0.65	0.019
query.js	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000
request.js	0.30	0.000	0.30	0.000	0.30	0.000	0.30	0.000	0.30	0.000
response.js	0.15	0.011	0.14	0.016	0.14	0.017	0.15	0.011	0.16	0.010
utils.js	0.56	0.029	0.62	0.000	0.62	0.000	0.62	0.029	0.60	0.029
view.js	0.06	0.000	0.06	0.000	0.06	0.000	0.06	0.000	0.06	0.000

Таблиця 4.11 – Попарне порівняння базової лінії, рангової вибірки та пропорційної вибірки для Express.

Файл	Базова лінія / Рангова вибірка (СА)		Базова лінія / Пропорційна вибірка (СА)		Рангова вибірка (СА) / Пропорційна вибірка (СА)	
	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$
application.js	0.00	0.232 (L)	0.00	0.588 (S)	0.00	0.830 (L)
query.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
request.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
response.js	0.00	0.203 (L)	0.29	0.510 (N)	0.00	0.801 (L)
utils.js	0.00	0.847 (L)	0.00	0.820 (L)	0.00	0.350 (S)
view.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)

Далі розглянемо тести розповсюджених алгоритмів на JavaScript, серед яких присутні такі як сортування підрахунком, червоно-чорне дерево та інші. Більша частина тестів алгоритмів націлена на методи, які працюють з графами. Хоча тест алгоритмів JavaScript складається з кількох підтестів, обговорюватимемо їх як один.

Результати показують, що варіант рангової вибірки значно перевершує базовий рівень для 9 файлів. Це значно перевищено в два рази. Він працює аналогічно з іншими 8 файлами. Варіант пропорційної вибірки перевершує базовий рівень на 13

файлах і перевершує його в нуль разів. Пропорційна вибірка значно перевершує варіант рангової вибірки у 8 разів. Відповідні результати медіан, IQR покриття гілок вказані в таблиці 4.12.

Таблиця 4.12 – Медіана та IQR покриття гілок для алгоритмів JavaScript.

Файл	**Базова лінія**		**Рангова вибірка**				**Пропорційна вибірка**			
			<u>*CA*</u>		<u>*CA+ДА*</u>		<u>*CA*</u>		<u>*CA+ДА*</u>	
	M	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*
articulationPoints.js	0.00	0.000	0.00	0.000	0.08	0.000	0.08	0.000	0.08	0.000
bellmanFord.js	0.00	0.000	0.17	0.000	0.33	0.125	0.33	0.167	0.33	0.000
bfTravellingSalesman.js	0.00	0.000	0.08	0.000	0.08	0.167	0.08	0.000	0.12	0.167
breadthFirstSearch.js	0.25	0.000	0.38	0.000	0.38	0.000	0.38	0.000	0.38	0.000
depthFirstSearch.js	0.17	0.167	0.17	0.167	0.17	0.000	0.17	0.000	0.17	0.167
detectDirectedCycle.js	0.00	0.000	0.12	0.000	0.38	0.000	0.38	0.000	0.38	0.000
detectUndirectedCycle.js	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000
dijkstra.js	0.00	0.000	0.10	0.000	0.20	0.000	0.10	0.100	0.20	0.000
eulerianPath.js	0.00	0.000	0.00	0.000	0.21	0.000	0.21	0.000	0.21	0.000
floydWarshall.js	0.00	0.000	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000
graphBridges.js	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000
hamitonianCycle.js	0.00	0.000	0.00	0.000	0.00	0.200	0.00	0.000	0.20	0.200
kruskal.js	0.10	0.000	0.30	0.100	0.30	0.000	0.40	0.100	0.40	0.100
prim.js	0.08	0.000	0.17	0.000	0.17	0.083	0.17	0.000	0.17	0.000
stronglyConnectedComponent.js	0.00	0.000	0.00	0.000	0.38	0.500	0.25	0.000	0.38	0.500
Knapsack.js	0.57	0.000	0.50	0.150	0.50	0.150	0.57	0.050	0.57	0.069
Matrix.js	0.74	0.046	0.71	0.026	0.74	0.053	0.79	0.079	0.76	0.046
CountingSort.js	0.92	0.083	0.92	0.000	0.92	0.000	0.92	0.000	0.92	0.000
RedBlackTree.js	0.21	0.000	0.26	0.000	0.26	0.000	0.26	0.029	0.26	0.029

В таблиці 4.13 продемонстровані результати порівняння базової лінії, рангової вибірки та пропорційної вибірки алгоритмів. Загалом варіант пропорційної вибірки значно перевершує базовий рівень.

Таблиця 4.13 – Попарне порівняння базової лінії, рангової вибірки та пропорційної вибірки для алгоритмів JavaScript.

Файл	**Базова лінія / Рангова вибірка (CA)**		**Базова лінія / Пропорційна вибірка (CA)**		**Рангова вибірка (CA) / Пропорційна вибірка (CA)**	
	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$
articulationPoints.js	1.00	0.500 (N)	0.00	1.000 (L)	0.00	1.000 (L)
bellmanFord.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.840 (L)
bfTravellingSalesman.js	0.00	1.000 (L)	0.00	1.000 (L)	0.08	0.530 (N)
breadthFirstSeach.js	0.00	0.992 (L)	0.00	0.992 (L)	1.00	0.500 (N)
depthFirstSearch.js	0.32	0.510 (N)	0.05	0.540 (N)	0.08	0.530 (N)
detectDirectedCycle.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.970 (L)
detectUndirectedCycle.js	1.00	0.500 (N)	0.03	0.550 (N)	0.03	0.550 (N)
dijkstra.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.690 (M)
eulerianPath.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.890 (L)
floydWarshall.js	0.00	1.000 (L)	0.00	1.000 (L)	1.00	0.500 (N)
graphBridges.js	1.00	0.500 (N)	0.32	0.510 (N)	0.32	0.510 (N)
hamitonianCycle.js	1.00	0.500 (N)	0.01	0.570 (N)	0.01	0.570 (N)
kruskal.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.630 (S)
prim.js	0.00	1.000 (L)	0.00	1.000 (L)	0.16	0.520 (N)
stronglyConnectedComponents.js	1.00	0.500 (N)	0.00	1.000 (L)	0.00	1.000 (L)
Knapsack.js	0.00	0.030 (L)	0.00	0.360 (S)	0.00	0.913 (L)
Matrix.js	0.00	0.230 (L)	0.00	0.668 (M)	0.00	0.863 (L)
CountingSort.js	0.03	0.542 (N)	0.01	0.572 (N)	0.08	0.530 (N)
RedBlackTree.js	0.00	0.954 (L)	0.00	0.962 (L)	0.04	0.630 (S)

У тесті Lodash спостерігаємо, що варіант рангової вибірки значно перевершує базовий рівень для трьох файлів. Базовий рівень значно перевершує рангову вибірку на 1 файлі. Для інших шести файлів суттєвої різниці немає. Варіант пропорційної вибірки працював подібно до варіанту вибірки на основі рангів, однак базовий рівень ніколи не перевершував варіант вибірки на основі рангів. Варіант рангової

вибірки значно перевершує варіант пропорційної вибірки для двох файлів. Результати тесту Lodash описані в таблицях 4.14 та 4.15 відповідно.

Таблиця 4.14 – Медіана та IQR покриття гілок для Lodash.

Файл	**Базова лінія**		**Рангова вибірка**				**Пропорційна вибірка**			
			СА		*СА+ДА*		*СА*		*СА+ДА*	
	M	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*
equalArrays.js	0.08	0.000	0.71	0.042	0.67	0.083	0.075	0.042	0.75	0.042
hasPath.js	0.75	0.000	0.75	0.000	0.75	0.000	0.75	0.000	0.75	0.000
random.js	1.00	0.000	1.00	0.000	1.00	0.054	1.00	0.000	1.00	0.000
result.js	0.90	0.000	0.80	0.000	0.80	0.000	0.90	0.000	0.90	0.000
slice.js	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000
split.js	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000
toNumber.js	0.60	0.000	0.65	0.000	0.65	0.000	0.65	0.050	0.65	0.050
transform.js	0.83	0.000	0.83	0.167	0.83	0.250	0.83	0.000	0.83	0.062
truncate.js	0.38	0.000	0.59	0.029	0.59	0.000	0.59	0.000	0.59	0.000
unzip.js	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000

Таблиця 4.15 – Попарне порівняння базової лінії, рангової вибірки та пропорційної вибірки для Lodash.

Файл	**Базова лінія / Рангова вибірка (СА)**		**Базова лінія / Пропорційна вибірка (СА)**		**Рангова вибірка (СА) / Пропорційна вибірка (СА)**	
	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$
equalArrays.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.740 (L)
hasPath.js	0.44	0.504 (N)	0.01	0.585 (S)	0.00	0.597 (S)
random.js	0.00	0.410 (S)	1.00	0.500 (N)	0.00	0.590 (S)
result.js	0.00	0.045 (L)	0.03	0.559 (N)	0.00	0.953 (L)
slice.js	0.01	0.570 (N)	1.00	0.500 (N)	0.01	0.430 (N)
split.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
toNumber.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.640 (S)
transform.js	0.00	0.446 (N)	0.00	0.688 (M)	0.00	0.694 (M)

Продовження таблиці 4.15.

Файл	Базова лінія / Рангова вибірка (CA)		Базова лінія / Пропорційна вибірка (CA)		Рангова вибірка (CA) / Пропорційна вибірка (CA)	
	p	$\hat{A}_{12}$	p	$\hat{A}_{12}$	p	$\hat{A}_{12}$
truncate.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	0.610 (S)
unzip.js	0.03	0.550 (N)	1.00	0.500 (N)	0.03	0.450 (N)

У тесті Moment виявлено, що варіант вибірки на основі рангів значно перевершує базовий рівень для 10 файлів. Для інших 9 файлів суттєвої різниці немає. Ті самі результати справедливі для варіанту пропорційної вибірки. Варіант пропорційної вибірки значно перевершує варіант вибірки на основі рангів для одного файлу та значно перевершує варіант для іншого файлу. Детальні результати описані в таблицях 4.16 та 4.17.

Таблиця 4.16 – Медіана та IQR покриття гілок для Moment.

Файл	Базова лінія		Рангова вибірка				Пропорційна вибірка			
			CA		CA+ДА		CA		CA+ДА	
	M	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*
add-substract.js	0.00	0.000	0.71	0.071	0.71	0.054	0.71	0.071	0.71	0.071
calendar.js	0.05	0.000	0.45	0.091	0.45	0.000	0.45	0.091	0.45	0.091
check-overflow.js	0.05	0.000	0.60	0.000	0.60	0.050	0.60	0.000	0.60	0.000
compare.js	0.14	0.000	0.14	0.000	0.14	0.000	0.14	0.000	0.14	0.000
constructor.js	0.38	0.000	0.56	0.000	0.56	0.000	0.56	0.031	0.56	0.062
date-from-array.js	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000
diff.js	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000
format.js	0.08	0.000	0.08	0.000	0.08	0.000	0.08	0.000	0.08	0.000
from-anything.js	0.74	0.029	0.76	0.029	0.76	0.029	0.76	0.029	0.76	0.029
from-array.js	0.02	0.000	0.04	0.000	0.04	0.000	0.04	0.000	0.04	0.000
from-object.js	0.50	0.000	0.50	0.000	0.50	0.000	0.50	0.000	0.50	0.000
from-string-and-array.js	0.00	0.000	0.31	0.000	0.31	0.000	0.31	0.000	0.31	0.000

Продовження таблиці 4.16 – Медіана та IQR покриття гілок для Moment.

Файл	**Базова лінія**		**Рангова вибірка**				**Пропорційна вибірка**			
			СА		*СА+ДА*		*СА*		*СА+ДА*	
	M	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*
from-string-and-format.js	0.06	0.000	0.59	0.031	0.50	0.180	0.53	0.180	0.52	0.219
from-string.js	0.06	0.000	0.16	0.000	0.16	0.000	0.16	0.000	0.16	0.000
get-set.js	0.14	0.000	0.23	0.034	0.41	0.045	0.45	0.045	0.45	0.045
locale.js	0.33	0.167	0.33	0.000	0.33	0.000	0.33	0.000	0.33	0.000
min-max.js	0.12	0.000	0.12	0.000	0.12	0.000	0.12	0.000	0.12	0.000
start-end-of.js	0.10	0.000	0.10	0.000	0.10	0.000	0.10	0.000	0.10	0.000
valid.js	0.38	0.000	0.38	0.000	0.38	0.000	0.38	0.000	0.38	0.000

Таблиця 4.17 – Попарне порівняння базової лінії, рангової вибірки та пропорційної вибірки для Moment.

Файл	**Базова лінія / Рангова вибірка (СА)**		**Базова лінія / Пропорційна вибірка (СА)**		**Рангова вибірка (СА) / Пропорційна вибірка (СА)**	
	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$
add-substract.js	0.00	1.000 (L)	0.00	1.000 (L)	0.16	0.480 (N)
calendar.js	0.00	1.000 (L)	0.00	1.000 (L)	0.32	0.497 (N)
check-overflow.js	0.00	1.000 (L)	0.00	1.000 (L)	0.26	0.586 (S)
compare.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
constructor.js	0.00	1.000 (L)	0.00	1.000 (L)	0.10	0.556 (N)
date-from-array.js	0.32	0.510 (N)	0.32	0.510 (N)	1.00	0.500 (N)
diff.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
format.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
from-anything.js	0.00	0.724 (M)	0.00	0.814 (L)	0.00	0.634 (S)
from-array.js	0.00	1.000	0.00	1.000	1.00	0.500 (N)
from-object.js	1.00	0.500 (N)	0.00	0.580 (S)	0.00	0.580 (S)
from-string-and-array.js	0.00	1.000 (L)	0.00	1.000 (L)	1.00	0.500 (N)
from-string-and-format.js	0.00	1.000 (L)	0.00	0.950 (L)	0.00	0.194 (L)
from-string.js	0.00	1.000 (L)	0.00	1.000 (L)	1.00	0.500 (N)
get-set.js	0.00	1.000 (L)	0.00	1.000 (L)	0.00	1.000 (L)

Продовження таблиці 4.17 – Попарне порівняння базової лінії, рангової вибірки та пропорційної вибірки для Moment.

Файл	Базова лінія / Рангова вибірка (СА)		Базова лінія / Пропорційна вибірка (СА)		Рангова вибірка (СА) / Пропорційна вибірка (СА)	
	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$	p	$\widehat{A}_{12}$
locale.js	0.00	0.340 (S)	0.00	0.360 (S)	0.16	0.520 (N)
min-max.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
start-end-of.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
valid.js	1.00	0.500 (N)	0.08	0.530 (N)	0.08	0.530 (N)

Проведемо підсумки результатів, отриманих з раніше оглянутих таблиць порівнянь. Загалом було визначено:

– варіант рангової вибірки СА (статичного аналізу) перевершує базовий рівень у 24 рази, з яких 24 є значущими;

– варіант пропорційної вибірки СА перевершує базову лінію в 31 раз, з яких 31 раз є значущим;

– варіант пропорційної вибірки СА перевершує варіант рангової вибірки СА 16 разів, з яких 16 були значущими.

Загалом, враховуючи результати, можна сказати, що використання виведення типу під час генерації тестів покращує можливості генерації тестів. Також зрозуміло, що в деяких ситуаціях це не приносить користі. Можемо сказати, що пропорційна вибірка явно перевершує рангову вибірку, коли для визначення типів використовується лише статичний аналіз. Результати вказують на те, що використання варіанта пропорційної вибірки рідко дає перевагу як базовому, так і ранговому варіанту вибірки, враховуючи поточний контрольний показник.

Враховуючи п’ять проектів, цікаво, що деякі показники отримують більше переваг від виведених типів, ніж інші. Щоб з’ясувати, чому існує ця різниця, потрібен був більш глибокий аналіз контрольних показників.

Тест Commander.js складається з класів із методами, які часто мають один із визначених користувачем об’єктів як аргумент. Це пояснює, чому використання виведення типу дає такий приріст продуктивності для цього тесту, оскільки використання випадкових аргументів типу в таких методах часто призводило б до збоїв.

Хоча тест Express складається з функцій як з простими типами, так і з аргументами складного типу, функції часто лише передають аргументи іншим функціям, тобто існує не так багато взаємодій з аргументами складного типу. Це ускладнює для поточних налаштувань визначення типів, що може пояснити відносно низьку вигоду від використання визначення типу в цьому тесті

Тест JavaScript Algorithms містить численні структури даних і алгоритми. Кожна структура даних має свій тип. Алгоритми, у свою чергу, використовують ці структури даних. Таким чином, використання виведення типу повинно бути вигідним для досягнення високого структурного покриття під час генерації тестових випадків. Однак, як було показано раніше, визначення типу є корисним лише для 13 із 19 файлів. Після дослідження згенерованих тестових прикладів виявилося, що варіанти виведення типу в більшості випадків виявляли правильний тип. Однак через складність наявних функцій варіанти підходу все одно не можуть охоплювати більше, ніж базова лінія. Це особливо вірно для п’яти файлів, де середнє охоплення дорівнює нулю. Точніше кажучи, файли, про які йдеться, використовують функції зворотного виклику, які передаються зовнішній функції. Ці функції зворотного виклику містять більшість гілок. У своєму поточному стані не можемо отримати правильний графік потоку керування такими функціями. Як тільки інструмент буде вдосконалено для роботи з такою складністю, зможемо виявити, що висновок про тип дійсно надає перевагу для відповідних файлів тестів.

Виявилося, що тест Lodash складається з функцій із переважно нативними аргументами типу JavaScript. Це пояснює, чому визначення типу не вплинуло на продуктивність більшості файлів. Файли, на які найбільше вплинуло визначення типу, наприклад, equalArrays.js і truncate.js, є двома файлами з найскладнішими

типами аргументів. Наприклад, truncate.js була єдиною функцією, яка потребувала об'єкт параметрів із певними атрибутами.

Останній тест Moment. Для цього тесту було 9 файлів, для яких визначення типу не дало переваги. Оскільки тест використовує складні типи, очікується, що визначення типу буде корисним. Після аналізу контрольного тесту здається, що Moment використовує синтаксис, де експортовані функції не є самостійними, оскільки вони використовують ключове слово «this». Замість визначення цих функцій як методів у класі, Moment імпортує всі функції та призначає їх об'єкту moment у головному файлі. Це налаштування робить неможливим запуск деяких функцій ізольовано. Через це щоразу, коли ключове слово «this» зустрічається під час тестування, створюється виняток, досягти будь-якого подальшого покриття.

4.4 Висновки

В першому етапі даного розділу проведено детальний аналіз певних аспектів реалізації програмного забезпечення. Було розглянуто деякі деталі, програмного забезпечення, а саме:

- абстрактне синтаксичне дерево;
- граф управління потоком;
- інструментальний код.

Наступним етапом стала розробка тестових сценаріїв. Обрано п'ять популярних проектів, які містять JavaScript код для різних сценаріїв. Серед цих проектів є код, який виконує як алгоритмічні задачі, так і звичайні базові задачі програмування. Вибрана саме така вибірка проектів, оскільки кодова база у цих проектів достатньо відрізняється, як і області їх застосування.

Параметри кожного JavaScript файлу були описані за наступними параметрами:

- кількість тестованих одиниць;
- цикломатична складність;

– кількість рядків коду;

– кількcть гілок.

Далі кожен з цих файлів пройшов тестування. Для кожного з них генерувались тести. Був присутній базовий підхід до генерації тестів, підхід з ранговою вибірку та застосуванням статичного аналізу та динамічного аналізу, а також з пропорційна вибірка з застосуванням статичного аналізу та динамічного аналізу.

По завершенню тестування було складено таблиці результатів, в яких порівнюється покриття коду різних методів, використовуючи в якості показників медіани та міжквартильні розмахи. Додатково було проведено порівняння базової вибірки, рангової вибірки та пропорційної вибірки за допомогою статичного показника $\widehat{A}_{12}$ Варґи-Делані.

Вирішальним моментом стало визначення ефективності запропонованого способу модифікації для автоматичної генерації тестів, а саме виведення типу. Виведення типу хоч і має певні успішні результати, проте в деяких місцях його зовсім немає. Деякі з протестованих проектів майже не мали ніякої вигоди з застосування виведення типу та залучення всіх наших нововведень, в той час, як деякі отримали певну вигоду.

Було виявлено деякі специфічні випадки, які не дають в повній мірі використати наш підхід. Наприклад у алгоритмах JavaScript є досить складні структури та розгалуження доповненні рекурсіями, які на даному етапі поки що не можуть належним чином бути оброблені. В той час Moment імпортує всі функції та призначає їх об’єкту moment у головному файлі. Це налаштування робить неможливим запуск деяких функцій ізольовано.

Теперішні отримані результати можна вважати цілком успішними. В майбутньому все ще є простір для досліджень даної теми.

ВИСНОВКИ

На основі викладеної інформації та даних, отриманих в ході проведеного дослідження можна проаналізувати отримані результати відносно кожного з розділів.

В першому розділі описана предметна область, що таке автоматичне тестування та розглянули відповідні питання, які з цим всім пов’язані. Фінальним етапом даного розділу була постановка задачі, яка полягає в тому щоб розробити або вдосконалити підхід до генерації тестів для JavaScript вирішивши питання неконтрольованого введення типів. Основне питання полягало не просто в генерації тестів, а як саме аналізувати типи в мовах, які є динамічними та слабо типізованими. Отримані результати демонструють, що тема виведення типу під час автоматичної генерації тестів не є достатньо дослідженою.

В другому розділі відбулось детальне ознайомлення з предметною областю та темою дослідження. Було переглянуто підходи до вирішення проблеми автоматичного тестування. Також розглянуто безпосередньо JavaScript та проблеми виведення типу, які пов’язані з його неконтрольованою та слабкою динамічною типізацією. Далі описаний статичний та динамічний аналіз, як вони можуть допомогти нам у проблеми з виведенням типу даних під час автоматичної генерації тестів, щоб уникнути використання випадкових типів. Відповідно до питань даного, було проаналізовано відповідно літературу, яка допомогла нам під час дослідження. Основними здобутками даного розділу було створення схеми процесу тестування, а також огляд деяких елементів цього процесу. Вирішальним етапом стала розробка таблиці, в якій описано підходи, які потім були використані для проведення тестування.

Третій розділ даної роботи містить детальне проектування основних компонентів нашої системи. Було розроблено схему компонентів. В даній схемі описані компоненти, а також згрупували деякі з них по відповідним групам. Завершальним етапом третього розділу був вибір технологій для реалізації нашого рішення. На основі відповідних вимог, було обрано мову програмування TypeScript,

середовище Node.js, а також визначено що розроблюваний інструмент буде реалізований у вигляді інтерфейсу командного рядка (CLI).

Фінальним етапом всієї роботи став четвертий розділ, в якому розглянуто розроблений інструмент. Спочатку коротко переглянуто деякі компоненти, а далі описано тестування та результати. Загалом було обрано п’ять проектів, які виконують різні задачі у програмуванні. Серед таких задач є як базові, так і більш складні, наприклад алгоритми націлені на роботу з графами тощо. Загальний процес тестування полягав у тому щоб згенерувати тести для наступних проектів:

– Commander.js;
– Express;
– JavaScript алгоритми;
– Lodash;
– Moment.

Далі виходячи з покриття коду для перерахованих проектів, можемо зробити певні висновки про ефективність методу виведення даних для автоматичної генерації тестів. Для цього створено таблиці результатів, в яких порівняно покриття коду, використовуючи в якості показників медіани та міжквартильні розмахи.

Відповідно до поставлених задач, отримано наступні результати:

– проаналізовано специфіку автоматичної генерації тестів та виведення типів;
– проведено аналіз існуючих методів генерації тестів та виведення даних;
– вдосконалено метод генерації тестів, шляхом виведення типів даних за допомогою комбінації статичного та динамічного аналізу;
– виявлено покращення тестування в деяких випадках.

Виходячи з покриття коду для протестованих проектів, зроблено висновки про ефективність методу неконтрольованого виведення типів даних для динамічних для автоматичної генерації тестів.

Враховуючи результати, можна стверджувати, що використання виведення типів даних для автоматичної генерації тестів покращує можливості генерації тестів та збільшує покриття коду тестами. Також зрозуміло, що в деяких ситуаціях даний метод не приносить користі.

ПЕРЕЛІК ДЖЕРЕЛ

1. Stack overflow developer survey 2022. URL https://survey.stackoverflow.co/2022/#most-popular-technologies-language.

2. Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 758–769, 2017. doi: 10.1109/ICSE.2017.75.

3. Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi:10.1145/3236024. 3236051. URL https://doi.org/10.1145/3236024.3236051.

4. Phil McMinn. Search-based software test data generation: a survey: Research articles. Software Testing, Verification & Reliability, 14:105–156, 2004.

5. Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 304–315, 2019. doi:10.1109/ICSE.2019.00045.

6. Mingrui Z. Jianzhong L. Fuchen M. Huafeng Z. Yu J. IntelliGen: Automatic Driver Synthesis for Fuzz Testing. URL: https://arxiv.org/pdf/2103.00862.

7. Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 86(8): 1978 – 2001, 2013. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2013.02.061. URL: https://www.sciencedirect.com/science/article/pii S0164121213000563.

8. Ahmed Mateen, Marriam Nazir, and Salman Afsar Awan. Optimization of test case generation using genetic algorithm (ga). ArXiv, abs/1612.08813, 2016.

9. Esmaeel Nikravan, Farid Feyzi, and Saeed Parsa. Enhancing path-oriented test data generation using adaptive random testing techniques. In 2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI), pages 510–513, 2015. doi: 10.1109/KBEI.2015.7436097.

10. Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 560–564, 2015. doi: 10.1109/SANER.2015.7081877.

11. Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. Pathafl: Pathcoverage assisted fuzzing. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, ASIA CCS '20, page 598–609, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367509. doi: 10.1145/3320269.3384736. URL https://doi.org/10.1145/3320269.3384736.

12. George Candea and Patrice Godefroid. Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances, pages 505–531. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91908-9. doi: 10.1007/978-3-319-91908-9 24. URL https://doi.org/10.1007/978-3-319-91908-9_24.

13. T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In Michael J. Maher, editor, Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making, pages 320–329, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30502-6.

14. L.A. Clarke. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, SE-2(3):215–222, 1976. doi: 10.1109/TSE.1976.233817.

15. Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. ACM Comput. Surv., 51(3), may 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL https://doi.org/10.1145/3182657.

16. James C. King. Symbolic execution and program testing. Commun. ACM, 19(7): 385–394, jul 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252.

17. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065036. URL https://doi.org/10.1145/1065010.1065036.

18. Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE ’11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025179. URL: https://doi.org/10.1145/2025113.2025179.

19. Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. Generating class-level integration tests using call site information, 2020. URL: https://arxiv.org/abs/2001.04221.

20. Andrea Arcuri. Restful api automated test case generation with evomaster. ACM Trans. Softw. Eng. Methodol., 28(1), jan 2019. ISSN 1049-331X. doi: 10.1145/3293455. URL: https://doi.org/10.1145/3293455.

21. Laurence Tratt. Chapter 5 dynamically typed languages. volume 77 of Advances in Computers, pages 149–184. Elsevier, 2009. doi: https://doi.org/10.1016/S0065-2458(09)01205-4. URL: https://www.sciencedirect.com/science/article/pii/S0065245809012054.

22. Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In Proceedings of the 38th Annual ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’11, page 459–472, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926437. URL https://doi.org/10.1145/1926385.1926437.

23. Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542528. URL https://doi.org/10.1145/1542476.1542528.

24. Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In Andrew P. Black, editor, ECOOP 2005 - Object Oriented Programming, pages 428–452, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31725-8.

25. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, Static Analysis, pages 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03237-0.

26. Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". SIGPLAN Not., 50(1):111–124, jan 2015. ISSN 0362-1340. doi: 10.1145/2775051.2677009. URL https://doi.org/10.1145/2775051.2677009.

27. Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks, 2020. URL https://arxiv.org/abs/2005.02161.

28. Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, page 91–105, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385997. URL https://doi.org/10.1145/3385412.3385997.

29. Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In Proceedings of the Network and Distributed System Security Symposium, NDSS

2010, San Diego, California, USA, 28th February - 3rd March 2010. The Internet Society, 2010. URL http://www.isoc.org/isoc/conferences/ndss/10/pdf/06.pdf.

30. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11, page 571–580, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985871. URL https://doi.org/10.1145/1985793.1985871.

31. Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: Automatic symbolic testing of javascript web applications. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, page 449–459, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635913. URL https://doi.org/10.1145/2635868.2635913.

32. Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Jseft: Automated javascript unit test generation. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pages 1–10, 2015. doi: 10.1109/IC ST.2015.7102595.

33. Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, page 488–498, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2491447. URL: https://doi.org/10.1145/2491411.2491447.

34. Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In Aldeida Aleti and Annibale Panichella, editors, Search-Based Software Engineering, pages 9–24, Cham, 2020. Springer International Publishing. ISBN 978-3-030-59762-7.

35. Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Empirical comparison of black-box test case generation tools for restful apis.

In 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 226–236, 2021. doi: 10.1109/SCAM52516.2021.00035.

36. Renata Hodov´ an, D´ aniel Vince, and´ Akos Kiss. Fuzzing javascript environment apis´ with interdependent function calls. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, Integrated Formal Methods, pages 212–226, Cham, 2019. Springer International Publishing. ISBN 978-3-030-34968-4.

37. Daniela Meneses Vargas, Alison Fernandez Blanco, Andreina Cota Vidaurre, Juan Pablo Sandoval Alcocer, Milton Mamani Torres, Alexandre Bergel, and Stephane Ducasse. Deviation testing: A test case generation technique for graphql apis. In 11th International Workshop on Smalltalk Technologies (IWST), pages 1–9, 2018.

38. Expressions and operators - javascript: Mdn. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators.

39. Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering, 44(2):122–158, 2018. doi: 10.1109/TSE.2017.2663435.

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

41. John R. Koza and Riccardo Poli. Genetic Programming, pages 127–164. Springer US, Boston, MA, 2005. ISBN 978-0-387-28356-2. doi: 10.1007/0-387-28356-0 5. URL: https://doi.org/10.1007/0-387-28356-0_5.

42. Dinuka R Wijendra and KP Hewagamage. Analysis of cognitive complexity with cyclomatic complexity metric of software. International Journal of Computer Applications, 975:8887.

43. M.J.G. Olsthoorn, A. van Deursen, and A. Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In Proceedings - 2020 35th IEEE/ACM International Conference on Automated

Software Engineering, ASE 2020, Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, pages 1224–1228. ACM/IEEE, 2020. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3418930. Virtual/online event due to COVID-19; 35th IEEE/ACM¡br/¿International Conference on Automated Software Engineering (ASE ’20),, ASE ’20 ; Conference date: 21-09-2020 Through 25-09-2020.

44. William Jay Conover. Practical nonparametric statistics, volume 350. John Wiley & Sons, 1998.

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

46. Юртаєв Д. О. Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів. Збірник наукових праць «Актуальні проблеми комп'ютерних наук АПКН-2024», с. 562-564, 2024.

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД

А.1 – Код відвідувача абстрактного синтаксичного дерева.

```
import { NodePath } from "@babel/core";
import { Scope as BabelScope, TraverseOptions } from "@babel/traverse";
import * as t from "@babel/types";
import { ImplementationError } from "@syntest/diagnostics";
import { getLogger, Logger } from "@syntest/logging";

import { globalVariables } from "./globalVariables";
import { reservedKeywords } from "./reservedKeywords";

export const MemberSeparator = " <-> ";

export class AbstractSyntaxTreeVisitor implements TraverseOptions {
  [k: `${string}|${string}`]: (
    this: any,
    path: NodePath<any>,
    state: any
  ) => void;

  protected static LOGGER: Logger;

  protected _filePath: string;

  protected _syntaxForgiving: boolean;

  protected _scopeIdOffset: number;

  protected _thisScopes: Set<string> = new Set([
    "ClassDeclaration",
    "FunctionDeclaration",
  ]);
  protected _thisScopeStack: number[] = [];
  protected _thisScopeStackNames: string[] = [];

  get filePath() {
    return this._filePath;
  }

  get syntaxForgiving() {
    return this._syntaxForgiving;
  }

  get scopeIdOffset() {
    return this._scopeIdOffset;
  }

  constructor(filePath: string, syntaxForgiving: boolean) {
    AbstractSyntaxTreeVisitor.LOGGER = getLogger("AbstractSyntaxTreeVisitor");
    this._filePath = filePath;
    this._syntaxForgiving = syntaxForgiving;
  }

  protected _getUidFromScope(scope: BabelScope): number {
    return (<{ uid: number }>(<unknown>scope))["uid"];
  }

  public _getNodeId(path: NodePath<t.Node> | t.Node): string {
    const loc = "node" in path ? path.node.loc : path.loc;
    if (loc === undefined) {
      throw new ImplementationError(
        `Node ${path.type} in file '${this._filePath}' does not have a location`
      );
    }
```

```
    const startLine = (<{ line: number }>(<unknown>loc.start)).line;
    const startColumn = (<{ column: number }>(<unknown>loc.start)).column;
    const startIndex = (<{ index: number }>(<unknown>loc.start)).index;
    const endLine = (<{ line: number }>(<unknown>loc.end)).line;
    const endColumn = (<{ column: number }>(<unknown>loc.end)).column;
    const endIndex = (<{ index: number }>(<unknown>loc.end)).index;

    return
`${this._filePath}:${startLine}:${startColumn}:::${endLine}:${endColumn}:::${startInd
ex}:${endIndex}`;
  }

  public _getBindingId(path: NodePath<t.Node>): string {
    if (
      path.parentPath.isLabeledStatement() &&
      path.parentPath.get("label") === path
    ) {
      if (this.syntaxForgiving) {
        AbstractSyntaxTreeVisitor.LOGGER.warn(
          `Unsupported labeled statement found at ${this._getNodeId(path)}`
        );
        return this._getNodeId(path);
      } else {
        throw new ImplementationError(
          "Cannot get binding for labeled statement"
        );
      }
    }

    if (
      path.parentPath.isMemberExpression() &&
      path.parentPath.get("property") === path
    ) {
      if (
        !path.isIdentifier() &&
        !path.isStringLiteral() &&
        !path.isNumericLiteral()
      ) {
        return this._getNodeId(path);
      }
      return (
        this._getBindingId(path.parentPath.get("object")) +
        MemberSeparator +
        (path.isIdentifier() ? path.node.name : `${path.node.value}`)
      );
    }

    if (
      path.parentPath.isClassMethod() &&
      path.parentPath.get("key") === path
    ) {
      return this._getNodeId(path);
    }

    if (
      (path.parentPath.isObjectProperty() ||
        path.parentPath.isObjectMethod()) &&
      path.parentPath.get("key") === path
    ) {
      return this._getNodeId(path);
    }

    if (
```

```
    path.parentPath.isImportSpecifier() &&
    path.parentPath.node.local.name !==
      ("name" in path.parentPath.node.imported
        ? path.parentPath.node.imported.name
        : path.parentPath.node.imported.value)
  ) {
    return this._getNodeId(path);
  }

  if (
    path.parentPath.isExportSpecifier() &&
    path.parentPath.get("exported") === path
  ) {
    return this._getNodeId(path);
  }

  if (
    path.parentPath.isExportSpecifier() &&
    path.parentPath.parentPath.has("source")
  ) {
    return this._getNodeId(path);
  }

  if (!path.isIdentifier()) {
    return this._getNodeId(path);
  }

  const binding = path.scope.getBinding(path.node.name);

  if (
    binding === undefined &&
    (globalVariables.has(path.node.name) ||
      reservedKeywords.has(path.node.name))
  ) {
    return `global::${path.node.name}`;
  } else if (binding === undefined) {
    if (this.syntaxForgiving) {
      AbstractSyntaxTreeVisitor.LOGGER.warn(
        `Cannot find binding for ${path.node.name} at ${this._getNodeId(
          path
        )}`
      );
      return this._getNodeId(path);
    } else {
      throw new ImplementationError(
        `Cannot find binding for ${path.node.name} at ${this._getNodeId(
          path
        )}`
      );
    }
  } else {
    return this._getNodeId(binding.path);
  }
}

public _getThisParent(
  path: NodePath<t.Node>
): NodePath<
  | t.FunctionDeclaration
  | t.FunctionExpression
  | t.ObjectExpression
  | t.Class
  | t.Program
> {
```

```
let parent = path.getFunctionParent();

if (parent === undefined || parent === null) {
  if (this.syntaxForgiving) {
    AbstractSyntaxTreeVisitor.LOGGER.warn(
      `ThisExpression without parent function found at ${this._getNodeId(
        path
      )}`
    );
    return undefined;
  } else {
    throw new ImplementationError(
      `ThisExpression without parent function found at ${this._getNodeId(
        path
      )}`
    );
  }
}

while (parent.isArrowFunctionExpression()) {
  parent = parent.getFunctionParent();

  if (parent === undefined || parent === null) {
    if (this.syntaxForgiving) {
      AbstractSyntaxTreeVisitor.LOGGER.warn(
        `ThisExpression without parent function found at ${this._getNodeId(
          path
        )}`
      );
      return undefined;
    } else {
      throw new ImplementationError(
        `ThisExpression without parent function found at ${this._getNodeId(
          path
        )}`
      );
    }
  }
}

if (parent.isClassMethod() || parent.isClassPrivateMethod()) {
  const classParent = path.findParent((p) => p.isClass());
  if (classParent === undefined || classParent === null) {
    throw new ImplementationError(
      `ThisExpression without parent class found at ${this._getNodeId(
        path
      )}`
    );
  }
  return <NodePath<t.Class>>classParent;
}

if (parent.isObjectMethod()) {
  const objectParent = path.findParent((p) => p.isObjectExpression());
  if (objectParent === undefined || objectParent === null)
    throw new ImplementationError(
      `ThisExpression without parent object found at ${this._getNodeId(
        path
      )}`
    );
  }
  return <NodePath<t.ObjectExpression>>objectParent;
}
```

```
    if (parent.isFunctionDeclaration() || parent.isFunctionExpression()) {
      return parent;
    }

    throw new ImplementationError(
      `ThisExpression without parent function found at ${this._getNodeId(path)}`
    );
  }

  enter = (path: NodePath<t.Node>) => {
    AbstractSyntaxTreeVisitor.LOGGER.silly(
      `Visiting node ${path.type} in file '${this._filePath}': location:
${path.node.loc?.start.line}:${path.node.loc?.start.column} -
${path.node.loc?.end.line}:${path.node.loc?.end.column} - type: ${path.node.type}`
    );
  };

  exit = (path: NodePath<t.Node>) => {
    AbstractSyntaxTreeVisitor.LOGGER.silly(
      `Exiting node ${path.type} in file '${this._filePath}': location:
${path.node.loc?.start.line}:${path.node.loc?.start.column} -
${path.node.loc?.end.line}:${path.node.loc?.end.column} - type: ${path.node.type}`
    );
  };

  public Program: (path: NodePath<t.Program>) => void = (path) => {
    if (this._scopeIdOffset === undefined) {
      this._scopeIdOffset = this._getUidFromScope(path.scope);
      this._thisScopeStack.push(this._getUidFromScope(path.scope));
      this._thisScopeStackNames.push("global");
    }
  };

  Scopable = {
    enter: (path: NodePath<t.Scopable>) => {
      if (!this._thisScopes.has(path.node.type)) {
        return;
      }

      if (!("id" in path.node)) {
        return;
      }

      let id = "anonymous";

      if (path.node.id !== null) {
        id = path.node.id.name;
      }

      const uid = this._getUidFromScope(path.scope);
      this._thisScopeStack.push(uid);
      this._thisScopeStackNames.push(id);
    },
    exit: (path: NodePath<t.Scopable>) => {
      if (!this._thisScopes.has(path.node.type)) {
        return;
      }

      this._thisScopeStack.pop();
      this._thisScopeStackNames.pop();
    },
  };

  protected _getCurrentThisScopeId() {
```

```
    if (this._thisScopeStack.length === 0) {
      throw new ImplementationError("Invalid scope stack!");
    }

    return this._thisScopeStack[this._thisScopeStack.length - 1];
  }
}
```

А.2 – Код відвідувача графа управління потоком.

```
import { NodePath } from "@babel/core";
import * as t from "@babel/types";
import { AbstractSyntaxTreeVisitor } from "@syntest/ast-visitor-javascript";
import {
  ControlFlowFunction,
  ControlFlowGraph,
  ControlFlowProgram,
  Edge,
  EdgeType,
  Location,
  Node,
  NodeType,
} from "@syntest/cfg";
import { ImplementationError } from "@syntest/diagnostics";
import { getLogger, Logger } from "@syntest/logging";

export class ControlFlowGraphVisitor extends AbstractSyntaxTreeVisitor {
  protected static override LOGGER: Logger;

  private _nodesList: Node[];
  private _nodes: Map<string, Node>;
  private _edges: Edge[];

  private _labeledBreakNodes: Map<string, Set<string>>;
  private _labeledContinueNodes: Map<string, Set<string>>;

  private _regularBreakNodesStack: Set<string>[];
  private _regularContinueNodesStack: Set<string>[];
  private _returnNodes: Set<string>;
  private _throwNodes: Set<string>;

  private _functions: ControlFlowFunction[];

  private _currentParents: string[];
  private _edgeType: EdgeType;

  get cfg(): ControlFlowProgram {
    if (!this._nodes.has("ENTRY")) {
      throw new ImplementationError("No entry node found");
    }
    if (!this._nodes.has("SUCCESS_EXIT")) {
      throw new ImplementationError("No success exit node found");
    }
    if (!this._nodes.has("ERROR_EXIT")) {
      throw new ImplementationError("No error exit node found");
    }

    if (this._nodesList.length !== this._nodes.size) {
      throw new ImplementationError("Number of nodes dont match");
    }

    const entryNode = this._nodes.get("ENTRY");
    const successExitNode = this._nodes.get("SUCCESS_EXIT");
```

```
    const errorExitNode = this._nodes.get("ERROR_EXIT");

    this._connectToParents(successExitNode);

    for (const returnNode of this._returnNodes) {
      this._edges.push(
        this._createEdge(
          this._nodes.get(returnNode),
          successExitNode,
          EdgeType.NORMAL
        )
      );
    }

    for (const throwNode of this._throwNodes) {
      this._edges.push(
        this._createEdge(
          this._nodes.get(throwNode),
          errorExitNode,
          EdgeType.EXCEPTION
        )
      );
    }

    if (this._regularBreakNodesStack.length > 0) {
      ControlFlowGraphVisitor.LOGGER.warn(
        `Found ${this._regularBreakNodesStack.length} break node stacks that are not
connected to a loop`
      );
    }

    if (this._regularContinueNodesStack.length > 0) {
      ControlFlowGraphVisitor.LOGGER.warn(
        `Found ${this._regularContinueNodesStack.length} continue node stacks that
are not connected to a loop`
      );
    }

    if (this._labeledBreakNodes.size > 0) {
      ControlFlowGraphVisitor.LOGGER.warn(
        `Found ${this._labeledBreakNodes.size} break node labels that are not
connected to a label exit`
      );
    }

    if (this._labeledContinueNodes.size > 0) {
      ControlFlowGraphVisitor.LOGGER.warn(
        `Found ${this._labeledContinueNodes.size} continue node labels that are not
connected to a label exit`
      );
    }

    return {
      graph: new ControlFlowGraph(
        entryNode,
        successExitNode,
        errorExitNode,
        this._nodes,
        this._edges
      ),
      functions: this._functions.map((function_, index) => {
        if (
          this._functions.filter((f) => f.name === function_.name).length > 1
        ) {
```

```
        function_.name = `${function_.name} (${index})`;
      }
      return function_;
    }),
  };
}

constructor(filePath: string, syntaxForgiving: boolean) {
  super(filePath, syntaxForgiving);
  ControlFlowGraphVisitor.LOGGER = getLogger("ControlFlowGraphVisitor");

  this._nodesList = [];
  this._nodes = new Map<string, Node>();
  this._edges = [];

  this._labeledBreakNodes = new Map();
  this._labeledContinueNodes = new Map();
  this._regularBreakNodesStack = [];
  this._regularContinueNodesStack = [];
  this._returnNodes = new Set<string>();
  this._throwNodes = new Set<string>();

  this._functions = [];

  this._currentParents = [];

  this._edgeType = EdgeType.NORMAL;

  const entry = new Node("ENTRY", NodeType.ENTRY, "ENTRY", [], {});
  const successExit = new Node("SUCCESS_EXIT", NodeType.EXIT, "EXIT", [], {});
  const errorExit = new Node("ERROR_EXIT", NodeType.EXIT, "EXIT", [], {});

  this._nodes.set(entry.id, entry);
  this._nodes.set(successExit.id, successExit);
  this._nodes.set(errorExit.id, errorExit);
  this._nodesList.push(entry, successExit, errorExit);

  this._currentParents = [entry.id];
}

private _getBreakNodes(): Set<string> {
  if (this._regularBreakNodesStack.length === 0) {
    throw new ImplementationError("No break nodes found");
  }
  return this._regularBreakNodesStack[
    this._regularBreakNodesStack.length - 1
  ];
}

private _getContinueNodes(): Set<string> {
  if (this._regularContinueNodesStack.length === 0) {
    throw new ImplementationError("No continue nodes found");
  }
  return this._regularContinueNodesStack[
    this._regularContinueNodesStack.length - 1
  ];
}

private _getLocation(path: NodePath): Location {
  return {
    start: {
      line: path.node.loc.start.line,
      column: path.node.loc.start.column,
      index: (<{ index: number }>(<unknown>path.node.loc.start)).index,
```

```
        },
        end: {
          line: path.node.loc.end.line,
          column: path.node.loc.end.column,
          index: (<{ index: number }>(<unknown>path.node.loc.end)).index,
        },
      };
    }

    private _createNode(path: NodePath): Node {
      const id = `${this._getNodeId(path)}`;
      const node = new Node(
        id,
        NodeType.NORMAL,
        path.node.type,
        [
          {
            id: id,
            location: this._getLocation(path),
            statementAsText: path.toString(),
          },
        ],
        {},
        path.node.type
      );

      if (this._nodes.has(id)) {
        throw new ImplementationError("Node already registered", {
          context: { nodeId: id },
        });
      }
      this._nodes.set(id, node);
      this._nodesList.push(node);

      return node;
    }

    public _getPlaceholderNodeId(path: NodePath<t.Node>): string {
      if (path.node.loc === undefined) {
        throw new ImplementationError(
          `Node ${path.type} in file '${this._filePath}' does not have a location`
        );
      }

      const startLine = (<{ line: number }>(<unknown>path.node.loc.start)).line;
      const startColumn = (<{ column: number }>(<unknown>path.node.loc.start))
        .column;
      const startIndex = (<{ index: number }>(<unknown>path.node.loc.start))
        .index;
      const endLine = (<{ line: number }>(<unknown>path.node.loc.end)).line;
      const endColumn = (<{ column: number }>(<unknown>path.node.loc.end)).column;
      const endIndex = (<{ index: number }>(<unknown>path.node.loc.end)).index;

      return
`${this._filePath}:${startLine}:${startColumn}:::${endLine}:${endColumn}:::${startInd
ex}:${endIndex}`;
    }

    /**
     * Create a placeholder node for a node that is not in the AST, but is used in the
CFG.
     * Uses the end location of the parent node as the start and end location of the
placeholder node.
     * @param path
```

```
 * @returns
 */
private _createPlaceholderNode(path: NodePath, double = false): Node {
  let id = `placeholder:::${this._getPlaceholderNodeId(path)}`;
  if (double) {
    id = "placeholder:::" + id;
  }
  const location = this._getLocation(path);
  const node = new Node(
    id,
    NodeType.NORMAL,
    path.node.type,
    [
      {
        id: id,
        location: {
          start: {
            line: location.start.line,
            column: location.start.column,
            index: location.start.index,
          },
          end: {
            line: location.end.line,
            column: location.end.column,
            index: location.end.index,
          },
        },
        statementAsText: path.toString(),
      },
    ],
    {},
    path.node.type
  );

  if (this._nodes.has(id)) {
    throw new ImplementationError("Node already registered", {
      context: { nodeId: id },
    });
  }
  this._nodes.set(id, node);
  this._nodesList.push(node);

  return node;
}

private _createEdge(
  source: Node,
  target: Node,
  edgeType: EdgeType,
  label = ""
): Edge {
  return new Edge(
    `${source.id}->${target.id}`,
    edgeType,
    label,
    source.id,
    target.id,
    "description"
  );
}

private _connectToParents(node: Node) {
  for (const parent of this._currentParents) {
    this._edges.push(
```

```
      this._createEdge(this._nodes.get(parent), node, this._edgeType)
    );
  }

  this._edgeType = EdgeType.NORMAL;
}

public Block: (path: NodePath<t.Block>) => void = (path) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering block at ${this._getNodeId(path)}`
  );


  if (path.isProgram() && this._scopeIdOffset === undefined) {
    this._scopeIdOffset = this._getUidFromScope(path.scope);
    this._thisScopeStack.push(this._getUidFromScope(path.scope));
    this._thisScopeStackNames.push("global");
  }

  for (const statement of path.get("body")) {
    statement.visit();
  }

  path.skip();
};

public Function: (path: NodePath<t.Function>) => void = (path) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering function at ${this._getNodeId(path)}`
  );

  if (!this._nodes.has(this._getNodeId(path))) {
    const node = this._createNode(path);

    this._connectToParents(node);
    this._currentParents = [node.id];
  }

  const subVisitor = new ControlFlowGraphVisitor(
    this.filePath,
    this.syntaxForgiving
  );
  path.traverse(subVisitor);

  if (!subVisitor._nodes.has("ENTRY")) {
    throw new ImplementationError("No entry node found");
  }

  const name = path.has("id")
    ? (<NodePath<t.Identifier>>path.get("id")).node.name
    : path.has("key")
    ? (<NodePath<t.Identifier>>path.get("key")).node.name
    : "anonymous";

  const cfp = subVisitor.cfg;

  this._functions.push(
    {
      id: this._getNodeId(path),
      name: name,
      graph: cfp.graph,
    },
    ...cfp.functions
  );
```

```
    path.skip();
  };

  public Statement: (path: NodePath<t.Statement>) => void = (path) => {
    ControlFlowGraphVisitor.LOGGER.debug(
      `Entering statement: ${path.type}\tline: ${path.node.loc.start.line}\tcolumn:
${path.node.loc.start.column}`
    );

    if (this._nodes.has(this._getNodeId(path))) {
      throw new ImplementationError("Id already used", {
        context: { nodeId: this._getNodeId(path) },
      });
    } else {
      const node = this._createNode(path);

      this._connectToParents(node);
      this._currentParents = [node.id];
    }
  };

  public Expression: (path: NodePath<t.Expression>) => void = (path) => {
    ControlFlowGraphVisitor.LOGGER.debug(
      `Entering Expression at ${this._getNodeId(path)}`
    );

    if (this._nodes.has(this._getNodeId(path))) {}
     else if (!path.isLiteral() && !path.isIdentifier()) {
      const node = this._createNode(path);

      this._connectToParents(node);
      this._currentParents = [node.id];
    }
  };

  public AssignmentPattern: (path: NodePath<t.AssignmentPattern>) => void = (
    path
  ) => {
    ControlFlowGraphVisitor.LOGGER.debug(
      `Entering AssignmentPattern at ${this._getNodeId(path)}`
    );

    const branchNode = this._createNode(path);
    this._connectToParents(branchNode);
    this._currentParents = [branchNode.id];

    const testNode = this._createNode(path.get("left"));
    this._connectToParents(testNode);
    this._currentParents = [testNode.id];

    this._edgeType = EdgeType.CONDITIONAL_TRUE;
    const consequent = this._createPlaceholderNode(path);
    this._connectToParents(consequent);
    this._currentParents = [consequent.id];

    const consequentNodes = this._currentParents;

    this._currentParents = [testNode.id];
    this._edgeType = EdgeType.CONDITIONAL_FALSE;

    const sizeBefore = this._nodesList.length;
    path.get("right").visit();
```

```
  if (sizeBefore === this._nodesList.length) {
    if (path.has("right")) {
      const alternate = this._createNode(path.get("right"));
      this._connectToParents(alternate);
      this._currentParents = [alternate.id];
    } else {
      throw new ImplementationError(
        "AssignmentPattern does not have a default value"
      );
    }
  }

  const alternateNodes = this._currentParents;

  this._currentParents = [...alternateNodes, ...consequentNodes];

  path.skip();
};

public Conditional: (path: NodePath<t.Conditional>) => void = (path) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering IfStatement at ${this._getNodeId(path)}`
  );

  const branchNode = this._createNode(path);
  this._connectToParents(branchNode);
  this._currentParents = [branchNode.id];

  const testNode = this._createNode(path.get("test"));
  this._connectToParents(testNode);
  this._currentParents = [testNode.id];

  this._edgeType = EdgeType.CONDITIONAL_TRUE;
  let sizeBefore = this._nodesList.length;
  path.get("consequent").visit();

  if (sizeBefore === this._nodesList.length) {
    const consequent = this._createNode(path.get("consequent"));
    this._connectToParents(consequent);
    this._currentParents = [consequent.id];
  }
  const consequentNodes = this._currentParents;

  this._currentParents = [testNode.id];
  this._edgeType = EdgeType.CONDITIONAL_FALSE;

  sizeBefore = this._nodesList.length;
  if (path.has("alternate")) {
    path.get("alternate").visit();
  }

  if (sizeBefore === this._nodesList.length) {
    if (path.has("alternate")) {
      const alternate = this._createNode(path.get("alternate"));
      this._connectToParents(alternate);
      this._currentParents = [alternate.id];
    } else {
      const alternate = this._createPlaceholderNode(path);
      this._connectToParents(alternate);
      this._currentParents = [alternate.id];
    }
  }

  const alternateNodes = this._currentParents;
```

```
  this._currentParents = [...alternateNodes, ...consequentNodes];

  path.skip();
};

public LabeledStatement: (path: NodePath<t.LabeledStatement>) => void = (
  path
) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering LabeledStatement at ${this._getNodeId(path)}`
  );

  const label = path.get("label").node.name;

  this._labeledBreakNodes.set(label, new Set());
  this._labeledContinueNodes.set(label, new Set());

  const labelNode = this._createNode(path);
  this._connectToParents(labelNode);

  this._currentParents = [labelNode.id];
  const beforeSize = this._nodes.size;
  path.get("body").visit();

  if (beforeSize === this._nodes.size) {
    const placeholderNode = this._createPlaceholderNode(path.get("body"));
    this._connectToParents(placeholderNode);
    this._currentParents = [placeholderNode.id];
  }

  const labelExit = this._createPlaceholderNode(path, true);

  this._currentParents.push(...this._labeledBreakNodes.get(label));
  this._connectToParents(labelExit);
  this._currentParents = [labelExit.id];
  for (const continueNode of this._labeledContinueNodes.get(label)) {
    this._edges.push(
      this._createEdge(
        this._nodes.get(continueNode),
        labelNode,
        EdgeType.BACK_EDGE
      )
    );
  }

  this._labeledBreakNodes.delete(label);
  this._labeledContinueNodes.delete(label);

  path.skip();
};

public DoWhileStatement: (path: NodePath<t.DoWhileStatement>) => void = (
  path
) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering DoWhileStatement at ${this._getNodeId(path)}`
  );

  const doWhileNode = this._createNode(path);
  this._connectToParents(doWhileNode);
  this._currentParents = [doWhileNode.id];

  this._regularBreakNodesStack.push(new Set());
```

```
  this._regularContinueNodesStack.push(new Set());

  const size = this._nodesList.length;
  path.get("body").visit();

  let firstBodyNode = this._nodesList[size];
  if (firstBodyNode === undefined) {
    const placeholderNode = this._createPlaceholderNode(path.get("body"));
    this._connectToParents(placeholderNode);
    this._currentParents = [placeholderNode.id];
    firstBodyNode = placeholderNode;
  }

  const loopNode = this._createNode(path.get("test"));
  this._connectToParents(loopNode);

  this._currentParents = [loopNode.id];
  this._edgeType = EdgeType.CONDITIONAL_TRUE;
  const consequent = this._createPlaceholderNode(path.get("test"));
  this._connectToParents(consequent);

  this._edges.push(
    this._createEdge(consequent, firstBodyNode, EdgeType.BACK_EDGE)
  );

  this._currentParents = [loopNode.id];
  this._edgeType = EdgeType.CONDITIONAL_FALSE;
  const alternate = this._createPlaceholderNode(path);
  this._connectToParents(alternate);

  this._currentParents = [alternate.id];
  const loopExit = this._createPlaceholderNode(path, true);

  this._currentParents.push(...this._regularBreakNodesStack.pop());
  this._connectToParents(loopExit);
  this._currentParents = [loopExit.id];

  for (const continueNode of this._regularContinueNodesStack.pop()) {
    this._edges.push(
      this._createEdge(
        this._nodes.get(continueNode),
        loopNode,
        EdgeType.BACK_EDGE
      )
    );
  }

  path.skip();
};

public WhileStatement: (path: NodePath<t.WhileStatement>) => void = (
  path
) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering WhileStatement at ${this._getNodeId(path)}`
  );

  const whileNode = this._createNode(path);
  this._connectToParents(whileNode);
  this._currentParents = [whileNode.id];

  this._regularBreakNodesStack.push(new Set());
  this._regularContinueNodesStack.push(new Set());
```

```
  const loopNode = this._createNode(path.get("test"));

  this._connectToParents(loopNode);

  this._currentParents = [loopNode.id];
  this._edgeType = EdgeType.CONDITIONAL_TRUE;
  const beforeSize = this._nodes.size;
  path.get("body").visit();

  if (beforeSize === this._nodes.size) {
    const placeholderNode = this._createPlaceholderNode(path.get("body"));
    this._connectToParents(placeholderNode);
    this._currentParents = [placeholderNode.id];
  }

  this._edgeType = EdgeType.BACK_EDGE;
  this._connectToParents(loopNode);

  this._currentParents = [loopNode.id];
  this._edgeType = EdgeType.CONDITIONAL_FALSE;
  const alternate = this._createPlaceholderNode(path);
  this._connectToParents(alternate);
   this._currentParents = [alternate.id];
  const loopExit = this._createPlaceholderNode(path, true);

  this._currentParents.push(...this._regularBreakNodesStack.pop());
  this._connectToParents(loopExit);
  this._currentParents = [loopExit.id];

  for (const continueNode of this._regularContinueNodesStack.pop()) {
    this._edges.push(
      this._createEdge(
        this._nodes.get(continueNode),
        loopNode,
        EdgeType.BACK_EDGE
      )
    );
  }

  path.skip();
};

public ForStatement: (path: NodePath<t.ForStatement>) => void = (path) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering ForStatement at ${this._getNodeId(path)}`
  );

  const forNode = this._createNode(path);
  this._connectToParents(forNode);
  this._currentParents = [forNode.id];

  this._regularBreakNodesStack.push(new Set());
  this._regularContinueNodesStack.push(new Set());

  if (path.has("init")) {
    const init = path.get("init");
    if (init.isVariableDeclaration()) {
      const node = this._createNode(init.get("declarations")[0].get("init"));

      this._connectToParents(node);
      this._currentParents = [node.id];
    } else {
      init.visit();
    }
```

```
    }
     let testNode;
    if (path.has("test")) {
      testNode = this._createNode(path.get("test"));
      this._connectToParents(testNode);
      this._currentParents = [testNode.id];
      this._edgeType = EdgeType.CONDITIONAL_TRUE;
    }

    let beforeSize = this._nodes.size;
    path.get("body").visit();

    if (beforeSize === this._nodes.size) {
      const placeholderNode = this._createPlaceholderNode(path.get("body"));
      this._connectToParents(placeholderNode);
      this._currentParents = [placeholderNode.id];
    }

    if (path.has("update")) {
      beforeSize = this._nodesList.length;
      path.get("update").visit();

      if (beforeSize === this._nodesList.length) {
        throw new ImplementationError(
          `No node was added for the update part of the for loop`
        );
      }
    }

    if (path.has("test")) {
      this._edgeType = EdgeType.BACK_EDGE;
      this._connectToParents(testNode);
      this._currentParents = [testNode.id];
      this._edgeType = EdgeType.CONDITIONAL_FALSE;

      const alternate = this._createPlaceholderNode(path);
      this._connectToParents(alternate);
      this._currentParents = [alternate.id];
    } else {
      this._currentParents = [];
    }

    const loopExit = this._createPlaceholderNode(path, true);

    this._currentParents.push(...this._regularBreakNodesStack.pop());
    this._connectToParents(loopExit);
    this._currentParents = [loopExit.id];

    for (const continueNode of this._regularContinueNodesStack.pop()) {
      this._edges.push(
        this._createEdge(
          this._nodes.get(continueNode),
          testNode,
          EdgeType.BACK_EDGE
        )
      );
    }

    path.skip();
  };

  public ForInStatement: (path: NodePath<t.ForInStatement>) => void = (
    path
  ) => {
```

```
ControlFlowGraphVisitor.LOGGER.debug(
  `Entering ForInStatement at ${this._getNodeId(path)}`
);

const forInNode = this._createNode(path);
this._connectToParents(forInNode);
this._currentParents = [forInNode.id];

this._regularBreakNodesStack.push(new Set());
this._regularContinueNodesStack.push(new Set());

if (!path.has("left")) {
  throw new ImplementationError("ForInStatement left not implemented", {
    context: { nodeId: this._getNodeId(path) },
  });
}
if (!path.has("right")) {
  throw new ImplementationError("ForInStatement right not implemented", {
    context: { nodeId: this._getNodeId(path) },
  });
}

path.get("left").visit();

const testNode = this._createPlaceholderNode(path.get("left"));
this._connectToParents(testNode);
this._currentParents = [testNode.id];

this._edgeType = EdgeType.CONDITIONAL_TRUE;
const beforeSize = this._nodes.size;
path.get("body").visit();

if (beforeSize === this._nodes.size) {
  const placeholderNode = this._createPlaceholderNode(path.get("body"));
  this._connectToParents(placeholderNode);
  this._currentParents = [placeholderNode.id];
}

this._edgeType = EdgeType.BACK_EDGE;
this._connectToParents(testNode);

this._currentParents = [testNode.id];
this._edgeType = EdgeType.CONDITIONAL_FALSE;
const alternate = this._createPlaceholderNode(path);
this._connectToParents(alternate);

this._currentParents = [alternate.id];
const loopExit = this._createPlaceholderNode(path, true);

this._currentParents.push(...this._regularBreakNodesStack.pop());
this._connectToParents(loopExit);
this._currentParents = [loopExit.id];

for (const continueNode of this._regularContinueNodesStack.pop()) {
  this._edges.push(
    this._createEdge(
      this._nodes.get(continueNode),
      testNode,
      EdgeType.BACK_EDGE
    )
  );
}

path.skip();
```

```
  };

  public ForOfStatement: (path: NodePath<t.ForOfStatement>) => void = (
    path
  ) => {
    ControlFlowGraphVisitor.LOGGER.debug(
      `Entering ForOfStatement at ${this._getNodeId(path)}`
    );

    const forOfNode = this._createNode(path);
    this._connectToParents(forOfNode);
    this._currentParents = [forOfNode.id];

    this._regularBreakNodesStack.push(new Set());
    this._regularContinueNodesStack.push(new Set());

    if (!path.has("left")) {
      throw new ImplementationError("ForInStatement left not implemented", {
        context: { nodeId: this._getNodeId(path) },
      });
    }
    if (!path.has("right")) {
      throw new ImplementationError("ForInStatement right not implemented", {
        context: { nodeId: this._getNodeId(path) },
      });
    }

    path.get("left").visit();

    const testNode = this._createPlaceholderNode(path.get("left"));
    this._connectToParents(testNode);
    this._currentParents = [testNode.id];

    this._edgeType = EdgeType.CONDITIONAL_TRUE;
    const beforeSize = this._nodes.size;
    path.get("body").visit();

    if (beforeSize === this._nodes.size) {
      const placeholderNode = this._createPlaceholderNode(path.get("body"));
      this._connectToParents(placeholderNode);
      this._currentParents = [placeholderNode.id];
    }

    this._edgeType = EdgeType.BACK_EDGE;
    this._connectToParents(testNode);

    this._currentParents = [testNode.id];
    this._edgeType = EdgeType.CONDITIONAL_FALSE;
    const alternate = this._createPlaceholderNode(path);
    this._connectToParents(alternate);

    this._currentParents = [alternate.id];
    const loopExit = this._createPlaceholderNode(path, true);

    this._currentParents.push(...this._regularBreakNodesStack.pop());
    this._connectToParents(loopExit);
    this._currentParents = [loopExit.id];

    for (const continueNode of this._regularContinueNodesStack.pop()) {
      this._edges.push(
        this._createEdge(
          this._nodes.get(continueNode),
          testNode,
          EdgeType.BACK_EDGE
```

```
      )
    );
  }

  path.skip();
};

public SwitchStatement: (path: NodePath<t.SwitchStatement>) => void = (
  path
) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering SwitchStatement at ${this._getNodeId(path)}`
  );

  this._regularBreakNodesStack.push(new Set());

  const switchNode = this._createNode(path);
  this._connectToParents(switchNode);
  this._currentParents = [switchNode.id];

  const testNode = this._createNode(path.get("discriminant"));
  this._connectToParents(testNode);
  this._currentParents = [testNode.id];
  let fallThrough: string[] = [];

  for (const caseNode of path.get("cases")) {
    if (caseNode.has("test")) {
      const caseTestNode = this._createNode(caseNode.get("test"));
      this._connectToParents(caseTestNode);
      this._currentParents = [caseTestNode.id];

      this._edgeType = EdgeType.CONDITIONAL_TRUE;
      const consequentNode = this._createNode(caseNode);
      this._connectToParents(consequentNode);
      this._currentParents = [consequentNode.id, ...fallThrough];

      if (caseNode.get("consequent").length > 0) {
        for (const consequentNode of caseNode.get("consequent")) {
          consequentNode.visit();
        }
      }

      const trueParents = this._currentParents;
      fallThrough = [...trueParents];

      this._currentParents = [caseTestNode.id];
      this._edgeType = EdgeType.CONDITIONAL_FALSE;
      const alternateNode = this._createPlaceholderNode(caseNode);
      this._connectToParents(alternateNode);
      this._currentParents = [alternateNode.id];
    } else {
      if (caseNode.get("consequent").length === 0) {
        const placeholderNode = this._createPlaceholderNode(caseNode);
        this._connectToParents(placeholderNode);
        this._currentParents = [placeholderNode.id];
      } else {
        for (const consequentNode of caseNode.get("consequent")) {
          consequentNode.visit();
        }
      }
    }
  }
  const switchExit = this._createPlaceholderNode(path, true);
```

```
  this._currentParents.push(
    ...fallThrough,
    ...this._regularBreakNodesStack.pop()
  );
  this._connectToParents(switchExit);
  this._currentParents = [switchExit.id];

  path.skip();
};

public BreakStatement: (path: NodePath<t.BreakStatement>) => void = (
  path
) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering BreakStatement at ${this._getNodeId(path)}`
  );

  const node = this._createNode(path);
  this._connectToParents(node);
  if (path.has("label")) {
    const label = path.get("label").node.name;
    if (!this._labeledBreakNodes.has(label)) {
      throw new ImplementationError("Label does not exist for break node", {
        context: { label: label, id: this._getNodeId(path) },
      });
    }
    this._labeledBreakNodes.get(label).add(node.id);
  } else {
    this._getBreakNodes().add(node.id);
  }
  this._currentParents = [];
  path.skip();
};

public ContinueStatement: (path: NodePath<t.ContinueStatement>) => void = (
  path
) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering ContinueStatement at ${this._getNodeId(path)}`
  );

  const node = this._createNode(path);
  this._connectToParents(node);

  if (path.has("label")) {
    const label = path.get("label").node.name;
    if (!this._labeledContinueNodes.has(label)) {
      throw new ImplementationError("Label does not exist for break node", {
        context: { label: label, id: this._getNodeId(path) },
      });
    }
    this._labeledContinueNodes.get(label).add(node.id);
  } else {
    this._getContinueNodes().add(node.id);
  }
  this._currentParents = [];
  path.skip();
};

public ReturnStatement: (path: NodePath<t.ReturnStatement>) => void = (
  path
) => {
  ControlFlowGraphVisitor.LOGGER.debug(
    `Entering ReturnStatement at ${this._getNodeId(path)}`
```

```
    );

    const node = this._createNode(path);
    this._connectToParents(node);
    this._currentParents = [node.id];

    if (path.has("argument")) {
      path.get("argument").visit();
    }

    for (const nodeId of this._currentParents) {
      this._returnNodes.add(nodeId);
    }

    this._currentParents = [];
    path.skip();
  };

  public ThrowStatement: (path: NodePath<t.ThrowStatement>) => void = (
    path
  ) => {
    ControlFlowGraphVisitor.LOGGER.debug(
      `Entering ThrowStatement at ${this._getNodeId(path)}`
    );

    const node = this._createNode(path);
    this._connectToParents(node);

    this._throwNodes.add(node.id);
    this._currentParents = [];
    path.skip();
  };
}
```

A.3 – Перетинач дерева.

```
import { ImplementationError } from "@syntest/diagnostics";
import { prng } from "@syntest/prng";
import { Crossover } from "@syntest/search";

import { JavaScriptTestCase } from "../../testcase/JavaScriptTestCase";
import { ActionStatement } from "../../testcase/statements/action/ActionStatement";
import { ConstantObject } from "../../testcase/statements/action/ConstantObject";
import { ConstructorCall } from "../../testcase/statements/action/ConstructorCall";
import { Statement } from "../../testcase/statements/Statement";

type SwapStatement = {
  parent: Statement;
  childIndex: number;
  child: Statement;
};

type MatchingPair = {
  parentA: SwapStatement;
  parentB: SwapStatement;
};

export class TreeCrossover extends Crossover<JavaScriptTestCase> {
  public crossOver(parents: JavaScriptTestCase[]): JavaScriptTestCase[] {
    if (parents.length !== 2) {
      throw new ImplementationError(
        `Expected exactly 2 parents, got: ${parents.length}`
      );
```

```
    }

  const rootA: ActionStatement[] = (<JavaScriptTestCase>parents[0].copy())
      .roots;
    const rootB: ActionStatement[] = (<JavaScriptTestCase>parents[1].copy())
      .roots;

    const swapStatementsA = this.convertToSwapStatements(rootA);
    const swapStatementsB = this.convertToSwapStatements(rootB);

    const crossoverOptions: MatchingPair[] = [];

    for (const swapA of swapStatementsA) {
      for (const swapB of swapStatementsB) {
        if (swapA.child.variableIdentifier === swapB.child.variableIdentifier) {
          if (
            swapA.child instanceof ConstructorCall &&
            !(swapB.child instanceof ConstructorCall)
          ) {
            continue;
          }

          if (
            swapB.child instanceof ConstructorCall &&
            !(swapA.child instanceof ConstructorCall)
          ) {
            continue;
          }

          if (
            swapA.child instanceof ConstructorCall &&
            swapB.child instanceof ConstructorCall &&
            swapA.child.export.id !== swapB.child.export.id
          ) {
            continue;
          }

          if (
            swapA.child instanceof ConstantObject &&
            !(swapB.child instanceof ConstantObject)
          ) {
            continue;
          }

          if (
            swapB.child instanceof ConstantObject &&
            !(swapA.child instanceof ConstantObject)
          ) {
            continue;
          }

          if (
            swapA.child instanceof ConstantObject &&
            swapB.child instanceof ConstantObject &&
            swapA.child.export.id !== swapB.child.export.id
          ) {
            continue;
          }

          crossoverOptions.push({
            parentA: swapA,
            parentB: swapB,
          });
        }
```

```
      }
    }

    if (crossoverOptions.length > 0) {
      const matchingPair = prng.pickOne(crossoverOptions);
      const parentA = matchingPair.parentA;
      const parentB = matchingPair.parentB;

      if (parentA.parent !== undefined && parentB.parent !== undefined) {
        parentA.parent.setChild(parentA.childIndex, parentB.child.copy());
        parentB.parent.setChild(parentB.childIndex, parentA.child.copy());
      } else if (parentB.parent !== undefined) {
        if (!(parentB.child instanceof ActionStatement)) {
          throw new TypeError(
            "expected parentB child to be an actionstatement"
          );
        }
        rootA[parentA.childIndex] = parentB.child.copy();
        parentB.parent.setChild(parentB.childIndex, parentA.child.copy());
      } else if (parentA.parent === undefined) {
        if (!(parentA.child instanceof ActionStatement)) {
          throw new TypeError(
            "expected parentA child to be an actionstatement"
          );
        }
        if (!(parentB.child instanceof ActionStatement)) {
          throw new TypeError(
            "expected parentB child to be an actionstatement"
          );
        }
        rootA[parentA.childIndex] = parentB.child.copy();
        rootB[parentB.childIndex] = parentA.child.copy();
      } else {
        if (!(parentA.child instanceof ActionStatement)) {
          throw new TypeError(
            "expected parentA child to be an actionstatement"
          );
        }
        parentA.parent.setChild(parentA.childIndex, parentB.child.copy());
        rootB[parentB.childIndex] = parentA.child.copy();
      }
    }

    return [new JavaScriptTestCase(rootA), new JavaScriptTestCase(rootB)];
  }

  protected convertToSwapStatements(roots: ActionStatement[]): SwapStatement[] {
    const swapStatements: SwapStatement[] = [];

    for (const [index, root] of roots.entries()) {
      swapStatements.push({
        parent: undefined,
        childIndex: index,
        child: root,
      });
    }

    const queue: Statement[] = [...roots];

    while (queue.length > 0) {
      const statement = queue.shift();

      if (statement.hasChildren()) {
        for (let index = 0; index < statement.getChildren().length; index++) {
```

```
          const child = statement.getChildren()[index];
          swapStatements.push({
            parent: statement,
            childIndex: index,
            child: child,
          });
          queue.push(child);
        }
      }
    }

    return swapStatements;
  }
}
```

A.4 – Будівник контексту тестових випадків.

```
import * as path from "node:path";

import { Export } from "@syntest/analysis-javascript";
import {
  globalVariables,
  reservedKeywords,
} from "@syntest/ast-visitor-javascript";
import { ImplementationError } from "@syntest/diagnostics";
import { getLogger, Logger } from "@syntest/logging";

import { ClassActionStatement } from
"../testcase/statements/action/ClassActionStatement";
import { FunctionCall } from "../testcase/statements/action/FunctionCall";
import { ObjectFunctionCall } from
"../testcase/statements/action/ObjectFunctionCall";
import { Statement } from "../testcase/statements/Statement";

type Import = RegularImport | RenamedImport;

type RegularImport = {
  name: string;
  renamed: false;
  module: boolean;
  default: boolean;
};

type RenamedImport = {
  name: string;
  renamed: true;
  renamedTo: string;
  module: boolean;
  default: boolean;
};

type Require = {
  left: string;
  right: string;
};

export class ContextBuilder {
  protected static LOGGER: Logger;
  private targetRootDirectory: string;
  private sourceDirectory: string;

  private globalNameCount: Map<string, number>;
  private testNameCount: Map<string, number>;

  private imports: Map<string, Import[]>;
```

```
private statementVariableNameMap: Map<Statement, string>;

constructor(targetRootDirectory: string, sourceDirectory: string) {
  ContextBuilder.LOGGER = getLogger("ContextBuilder");
  this.targetRootDirectory = targetRootDirectory;
  this.sourceDirectory = sourceDirectory;

  this.globalNameCount = new Map();
  this.testNameCount = new Map();

  this.imports = new Map();
  this.statementVariableNameMap = new Map();
}

nextTestCase() {
  this.statementVariableNameMap = new Map();
  this.testNameCount = new Map();
}

getOrCreateVariableName(statement: Statement): string {
  if (this.statementVariableNameMap.has(statement)) {
    return this.statementVariableNameMap.get(statement);
  }

  let variableName = statement.name;

  variableName = variableName.replaceAll(/[^A-Za-z]/g, "");

  variableName = variableName[0].toLowerCase() + variableName.slice(1);

  variableName =
    reservedKeywords.has(variableName) || globalVariables.has(variableName)
      ? "local" + variableName[0].toUpperCase() + variableName.slice(1)
      : variableName;

  if (
    statement instanceof ClassActionStatement ||
    statement instanceof FunctionCall ||
    statement instanceof ObjectFunctionCall
  ) {
    variableName += "ReturnValue";
  }

  let count = -1;
  if (
    this.globalNameCount.has(variableName) &&
    this.testNameCount.has(variableName)
  ) {
    count = Math.max(
      this.globalNameCount.get(variableName),
      this.testNameCount.get(variableName)
    );
  } else if (this.globalNameCount.has(variableName)) {
    count = this.globalNameCount.get(variableName);
  } else if (this.testNameCount.has(variableName)) {
    count = this.testNameCount.get(variableName);
  }

  if (count === -1) {
    this.testNameCount.set(variableName, 1);
  } else {
    this.testNameCount.set(variableName, count + 1);
    variableName += count;
```

```
    }

    this.statementVariableNameMap.set(statement, variableName);
    return variableName;
  }

  getOrCreateImportName(export_: Export): string {
    const import_ = this._addImport(export_);

    return import_.renamed ? import_.renamedTo : import_.name;
  }

  private _addImport(export_: Export): Import {
    const path_ = export_.filePath.replace(
      path.resolve(this.targetRootDirectory),
      path.join(this.sourceDirectory, path.basename(this.targetRootDirectory))
    );

    const exportedName = export_.renamedTo;
    let import_: Import = {
      name: exportedName === "default" ? "defaultExport" : exportedName,
      renamed: false,
      default: export_.default,
      module: export_.module,
    };
    let newName: string = exportedName;

    if (this.imports.has(path_)) {
      const foundImport = this.imports.get(path_).find((value) => {
        return (
          value.name === import_.name &&
          value.default === import_.default &&
          value.module === import_.module
        );
      });
      if (foundImport !== undefined) {
        return foundImport;
      }
    }

    let count = -1;
    if (
      this.globalNameCount.has(exportedName) &&
      this.testNameCount.has(exportedName)
    ) {
      count = Math.max(
        this.globalNameCount.get(exportedName),
        this.testNameCount.get(exportedName)
      );
    } else if (this.globalNameCount.has(exportedName)) {
      count = this.globalNameCount.get(exportedName);
    } else if (this.testNameCount.has(exportedName)) {
      count = this.testNameCount.get(exportedName);
    }

    if (count === -1) {
      this.globalNameCount.set(exportedName, 1);
    } else {
      this.globalNameCount.set(exportedName, count + 1);
      this.testNameCount.set(exportedName, count + 1);
      newName = exportedName + count.toString();

      import_ = {
        name: exportedName,
```

```
      renamed: true,
      renamedTo: newName,
      default: export_.default,
      module: export_.module,
    };
  }

  if (!this.imports.has(path_)) {
    this.imports.set(path_, []);
  }

  this.imports.get(path_).push(import_);
  return import_;
}

private _getImportString(_path: string, import_: Import): string {
  if (import_.module) {
    throw new ImplementationError(
      "Only non module imports can use import statements"
    );
  }
  if (import_.renamed) {
    return import_.default
      ? `import ${import_.renamedTo} from "${_path}";`
      : `import {${import_.name} as ${import_.renamedTo}} from "${_path}";`;
  } else {
    return import_.default
      ? `import ${import_.name} from "${_path}";`
      : `import {${import_.name}} from "${_path}";`;
  }
}

private _getRequireString(_path: string, import_: Import): Require {
  if (!import_.module) {
    throw new ImplementationError(
      "Only module imports can use require statements"
    );
  }

  const require: Require = {
    left: "",
    right: `require("${_path}")`,
  };

  if (import_.renamed) {
    require.left = import_.default
      ? import_.renamedTo
      : `{${import_.name}: ${import_.renamedTo}}`;
  } else {
    require.left = import_.default ? import_.name : `{${import_.name}}`;
  }

  return require;
}

getImports(assertionsPresent: boolean) {
  let requires: Require[] = [];
  let imports: string[] = [];

  for (const [path_, imports_] of this.imports.entries()) {
    for (const import_ of imports_) {
      if (import_.module) {
        requires.push(this._getRequireString(path_, import_));
      } else {
```

```
          imports.push(this._getImportString(path_, import_));
        }
      }
    }

    requires = requires
      .filter((value, index, self) => self.indexOf(value) === index)
      .sort();

    imports = imports
      .filter((value, index, self) => self.indexOf(value) === index)
      .sort();

    if (assertionsPresent) {
      imports.push(
        `import chai from 'chai'`,
        `import chaiAsPromised from 'chai-as-promised'`,
        `const expect = chai.expect;`,
        `chai.use(chaiAsPromised);`
      );
    }

    return {
      imports,
      requires,
    };
  }
}
```

ДОДАТОК Б

(обов'язковий)

КОПІЯ НАУКОВИХ ТЕЗ

Міністерство освіти і науки України
Хмельницький національний університет

ЗБІРНИК НАУКОВИХ ПРАЦЬ

за матеріалами XVI Всеукраїнської науково-практичної конференції
«Актуальні проблеми комп'ютерних наук АПКН-2024»

15-16 листопада 2024

Хмельницький 2024

УДК 004.4

Юртаєв Д.О.

Хмельницький національний університет

МЕТОД ВИКОРИСТАННЯ НЕКОНТРОЛЬОВАНОГО ВВЕДЕННЯ ТИПУ ДЛЯ МОВ ПРОГРАМУВАННЯ ПРИ АВТОМАТИЧНІЙ ГЕНЕРАЦІЇ ТЕСТІВ

Розглянуто прикладні аспекти використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів. Автоматична генерація тестів забезпечує швидшу розробку програмного забезпечення та допомагає уникати людського фактору під час тестування, що позитивно впливає на подальшу розробку програмних засобів.

Applied aspects of using uncontrolled type input for programming languages in automatic test generation are considered. Automatic test generation ensures faster software development and helps to avoid the human factor during testing, which has a positive effect on further software development.

З розвитком нових технологій та постійним ускладненням нового програмного забезпечення, написання тестів для програмного забезпечення стає досить складним та деколи проблемним процесом на який витрачається багато ресурсів.

На сьогодні, автоматична генерація тестів для мов програмування є однією з ключових тем у галузі розробки програмного забезпечення. Використання неконтрольованого введення типів стає все більш актуальним у контексті динамічно типізованих мов, таких як Python і JavaScript, де типи не завжди явно визначені.

Метою роботи є вивчення підходів до неконтрольованого введення типу, аналіз проблем, які з цим пов'язані, та визначення методів, які можуть допомогти мінімізувати ризики й підвищити ефективність автоматизованого тестування. Глибше розуміння цієї теми має потенціал значно полегшити процес розробки, зробити його безпечнішим і продуктивнішим, при умові що автоматична генерація тестів буде покривати достатню кількість коду.

Неконтрольоване введення типу передбачає генерацію випадкових або спеціально сформованих даних для автоматичного тестування програм з метою виявлення прихованих помилок, перевірки поведінки системи в різних сценаріях або оцінки коректності типів.

Проте в даній темі не все так посто як звучить, на перший погляд, оскільки тема є досить складною та може призвести до певних проблем, неточностей та в цілому складнощів у реалізації. Використання неконтрольованого введення типу

Тестове покриття є важливим критерієм для оцінки якості тестування програмного забезпечення. Воно визначає, наскільки повно тестування охоплює код програми. Ось кілька основних видів тестового покриття:

- покриття операторів (statement coverage);
- покриття гілок (branch coverage);
- покриття умов (condition coverage);
- покриття шляхів (path coverage);
- покриття функцій (function coverage).

Кожен з цих видів покриттів визначає відсоток покриття. Щоб порахувати кожен з цих критеріїв можна скористатися наступною формулою:

$$R = \frac{C}{N} 100\%, \quad (1)$$

де C – кількість виконаних функцій, N – загальна кількість функцій.

Отже, наступні задачі полягають у дослідженні та створенні системи автоматичного тестування, можливо навіть на основі fuzzing, яка буде здатна генерувати тести з неконтрольованим введенням типів для виявлення уразливостей у програмних системах із високою точністю та ефективністю. Система повинна бути здатна адаптуватися до мови програмування JavaScript та забезпечувати широкий спектр тестових випадків із мінімальною участю розробників.

Перелік посилань

1. Mingrui Z. Jianzhong L. Fuchen M. Huafeng Z. Yu J. IntelliGen: Automatic Driver Synthesis for Fuzz Testing. URL: https://arxiv.org/pdf/2103.00862
2. Renata Hodovan, Daniel Vince, and Akos Kiss. Fuzzing javascript environment apis with interdependent function calls. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, Integrated Formal Methods, pages 212–226, Cham, 2019. Springer International Publishing. ISBN 978-3-030-34968-4.
3. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, Static Analysis, pages 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03237-0.

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Кафедра інженерії програмного забезпечення

Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів

Виконав:
Студент II курсу, група ІПЗм-23-1
Юртаєв Денис Олександрович

Керівник:
доцент, к. пед. наук
Праворська Наталя Іванівна

1

Рисунок В.1 – Слайд 1.

Актуальність теми

Актуальність теми роботи полягає у необхідності вдосконалені методу автоматичної генерації тестів, який дозволить швидко та без витрати великої кількості ресурсів покращувати швидкість розробки програмного забезпечення, а також підвищити якість тестів.

Необхідність такої розробки ґрунтується на наявності невирішених питань в галузі автоматичної генерації тестів.

2

Рисунок В.2 – Слайд 2.

Об'єкт, предмет та мета дослідження

Об'єкт дослідження – вихідні програмні коди, в яких визначаються типи даних необхідні для автоматичної генерації тестів.

Предмет дослідження – метод неконтрольованого виведення типів для динамічних мов програмування під час автоматичної генерації тестів.

Мета дослідження – детальне дослідження методу виведення типу під час автоматичної генерації тестів та їх ефективність.

3

Рисунок В.3 – Слайд 3.

Наукова новизна

Наукова новизна полягає в тому, що вперше було запропоновано вдосконалений підхід до аналізу вихідного програмного коду за допомогою статичного аналізу доповненого динамічним аналізом. Це дозволяє визначати більшість типів даних і, відповідно, генерувати більшу кількість тестових випадків.

4

Рисунок В.4 – Слайд 4.

Завдання дослідження

Завданням роботи є:
- проаналізувати специфіку автоматичної генерації тестів та виведення типів;
- провести аналіз існуючих методів;
- розробити та вдосконалити методи та алгоритми;
- виконати проектування програмної системи на основі методів та алгоритмів;
- проаналізувати отримані результати дослідження та сформувати рекомендації щодо доцільності впровадження результатів дослідження.

5

Рисунок В.5 – Слайд 5.

Аналіз стану проблеми та інших рішень

Основні проблеми у галузі:

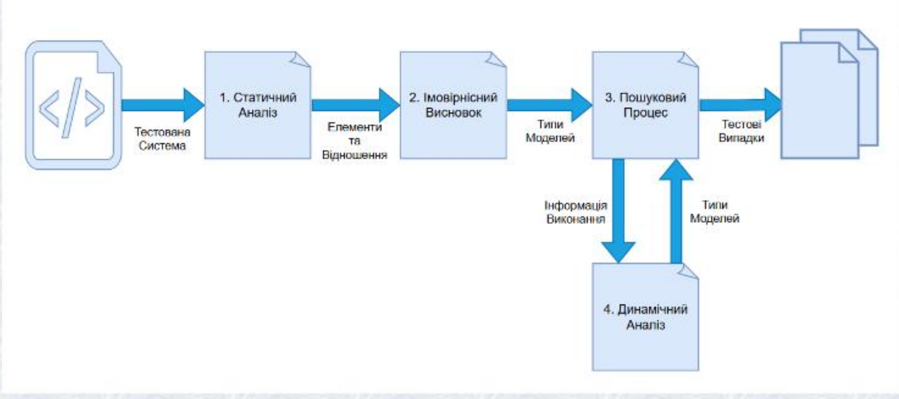
- складність виведення типів даних створених розробником ПЗ.
- обмежений словниковий запас у виведенні типів;
- дороговизна методів обробки природної мови;
- недостатня кількість методів, призначених для динамічно-типізованих мов.

6

Рисунок В.6 – Слайд 6.

Розроблені методи

Для покращення методу тестування було розроблено та задіяно методи статичного та динамічного аналізу.



7

Рисунок В.7 – Слайд 7.

Перший розділ

В першому розділі було оглянуто основні методики, які використовуються для автоматичного тестування.

З'ясувалось що, дослідження в даній темі є досить різноманітні, які в свою чергу використовують безліч підходів до вирішення задач.

Також було визначено, що в даному дослідженні буде використаний підхід, який комбінує статичний аналіз та динамічний аналіз.

8

Рисунок В.8 – Слайд 8.

Перший розділ

Постановка задач:

1. Дослідити вплив виведення типу на ефективність створення тестових випадків для JavaScript;
2. Виміряти вплив на продуктивність використання інформації про виконання для підвищення точності виведення типу для JavaScript.
3. Зробити оцінку того, чи може використання виведення типу бути недорогим процесом

9

Рисунок В.9 – Слайд 9.

Другий розділ

Інструменти створення тестових випадків та виведення типу:

– Jseft (інструмент генерації тестів JavaScript, зосереджений на створенні тестів на основі подій і модульних тестів функціонального рівня);
– Pynguin (автоматизований фреймворк створення модульних тестів Python);
– NL2Type (модель глибокого навчання, використовує коментарі в поєднанні з кодом як вхідні дані для прогнозування типів);
– Type4Py (зосереджений на Python і розв'язує проблему обмеженого словникового запасу, використовуючи глибоку стратегію вивчення подібності).

10

Рисунок В.10 – Слайд 10.

Другий розділ

Результати дослідження:

— більшість пов'язаних робіт із виведення типу використовує машинне навчання для виконання оброки природної мови;

— підходи не можуть вивести типи, визначені у вихідному коді розробником;

— створення тестів для JavaScript зосереджено в основному на клієнтських програмах;

— створення тестових випадків для динамічно типізованих мов не використовують повноцінні методи виведення типу.

11

Рисунок В.11 – Слайд 11.

Другий розділ

Варіанти підходу тестування.

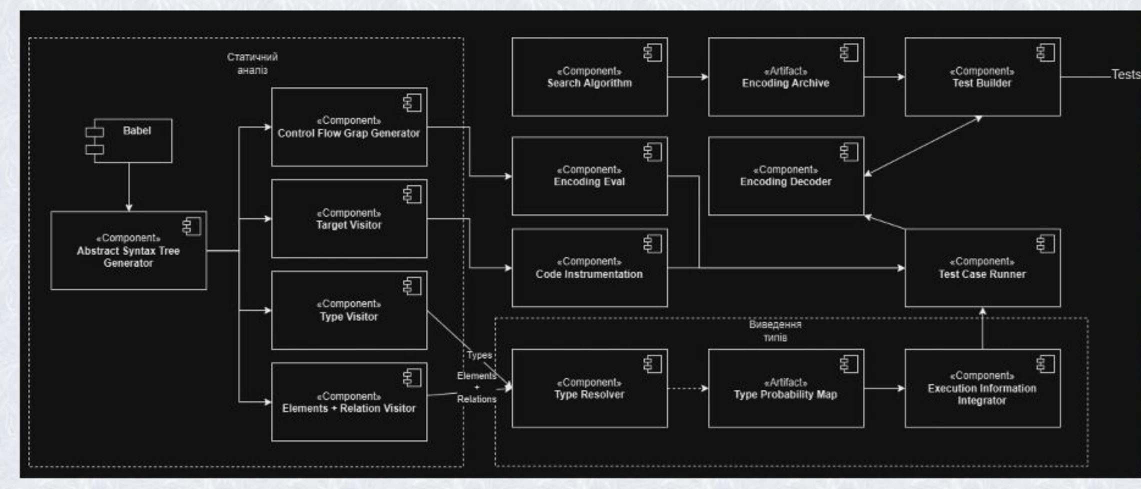
Варіант	**Режим вибірки**	**Режим аналізу**
Базовий	-	-
СА (статичний аналіз) на основі рангу	На основі рангу	Статичний аналіз
СА (статичний аналіз) пропорційний	Пропорційний	Статичний аналіз
СА+ДА (статичний + динамічний аналіз) на основі рангу	На основі рангу	Статичний аналіз та динамічний аналіз
СА+ДА (статичний + динамічний аналіз) пропорційний	Пропорційний	Статичний аналіз та динамічний аналіз

12

Рисунок В.12 – Слайд 12.

Третій розділ

Загальна схема компонентів.



13

Рисунок В.13 – Слайд 13.

Четвертий розділ

Імплементація інструменту дає змогу проаналізувати вірність гіпотези про виведення типів.

5 проектів JavaScript, обраних для тестування:
— Express (веб-фреймворк для node.js);
— Commander (структура командрого рядка node.js);
— Moment (бібліотека для аналізу та обробки дат);
— Algorithms (бібліотека, що містить популярні алгоритми);
— Lodash (службові функції для звичайних завдань програмування).

14

Рисунок В.14 – Слайд 14.

Четвертий розділ

Статистика тестованих проектів, покриття коду різних методів та їх порівняння.

Бенчмарк	**Файл**	**Кількість тестованих одиниць**	**Складність**	**Кількість рядків коду**	**Кількість гілок**
Express	query.js	1	5	47	6
	layer.js	1	17	181	22
	route.js	1	23	225	30
	application.js	1	42	661	52
	request.js	1	35	525	44
	response.js	1	133	1169	174
	utils.js	7	28	304	34
	view.js	1	14	182	16

15

Рисунок В.15 – Слайд 15.

Четвертий розділ

Результати покриття коду та їх порівняння.

Файл	**Базова лінія**		**Рангова вибірка**				**Пропорційна вибірка**			
			СА		*СА+ДА*		*СА*		*СА+ДА*	
	M	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*	*M*	*IQR*
application.js	0.63	0.019	0.62	0.019	0.63	0.019	0.65	0.019	0.65	0.019
query.js	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000
request.js	0.30	0.000	0.30	0.000	0.30	0.000	0.30	0.000	0.30	0.000
response.js	0.15	0.011	0.14	0.016	0.14	0.017	0.15	0.011	0.16	0.010
utils.js	0.56	0.029	0.62	0.000	0.62	0.000	0.62	0.029	0.60	0.029
view.js	0.06	0.000	0.06	0.000	0.06	0.000	0.06	0.000	0.06	0.000

Файл	**Базова лінія / Рангова вибірка (СА)**		**Базова лінія / Пропорційна вибірка (СА)**		**Рангова вибірка (СА) / Пропорційна вибірка (СА)**	
	p	$\hat{A}_{12}$	p	$\hat{A}_{12}$	p	$\hat{A}_{12}$
application.js	0.00	0.232 (L)	0.00	0.588 (S)	0.00	0.830 (L)
query.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
request.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)
response.js	0.00	0.203 (L)	0.29	0.510 (N)	0.00	0.801 (L)
utils.js	0.00	0.847 (L)	0.00	0.820 (L)	0.00	0.350 (S)
view.js	1.00	0.500 (N)	1.00	0.500 (N)	1.00	0.500 (N)

16

Рисунок В.16 – Слайд 16.

Наукові публікації

Для даної роботи було опубліковано наукові тези:

— Юртаєв Д. О. Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів. Збірник наукових праць «Актуальні проблеми комп'ютерних наук АПКН-2024», 2024, с. 562-564

17

Рисунок В.17 – Слайд 17.

Отримані результати

В ході проведено дослідження, отримані наступні результати:

– проаналізовано специфіку автоматичної генерації тестів та виведення типів;

– проведено аналіз існуючих методів генерації тестів та виведення даних;

– вдосконалено метод генерації тестів, шляхом виведення типів даних за допомогою комбінації статичного та динамічного аналізу;

– виявлено покращення тестування в деяких випадках.

Враховуючи результати проведеного дослідження, можна стверджувати, що використання виведення типу під час генерації тестів покращує їх можливості, проте також стає зрозумілим, що в деяких ситуаціях це не приносить користі.

Було виявлено деякі специфічні випадки, які не дають в повній мірі використати наш підхід.

Тому можна вважати, що мета дослідження досягнута, всі поставлені задачі реалізовані.

18

Рисунок В.18 – Слайд 18.

Висновок

Виходячи з покриття коду для протестованих проектів, зроблено висновки про ефективність методу неконтрольованого виведення типів даних для динамічних для автоматичної генерації тестів.

Враховуючи результати, можна стверджувати, що використання виведення типів даних для автоматичної генерації тестів покращує можливості генерації тестів та збільшує покриття коду тестами.

Також зрозуміло, що в деяких ситуаціях даний метод не приносить користі.

19

Рисунок В.19 – Слайд 19.

Дякую за увагу!

20

Рисунок В.20 – Слайд 20.

Завідувачу кафедри інженерії програмного забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Юртаєва Д.О.
факультет ІТ, 2 курс, група ІПЗм-23-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і 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 й 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 (StrikePlagiarism 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.

02.09.2024
дата

підпис

Завідувачу кафедри
інженерії програмного забезпечення
проф. Леоніду БЕДРАТЮКУ
студента групи ІПЗм-23-1
Юртаєв Д. О.
Ім'я, ПРІЗВИЩЕ

ЗАЯВА

Прошу закріпити за мною тему кваліфікаційної роботи освітнього ступеня «магістр» за спеціальністю 121 «Інженерія програмного забезпечення»:

Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів

(керівник кваліфікаційної роботи – Праворська Н.І.)
Ім'я, ПРІЗВИЩЕ

02.09.2024
Дата

Підпис здобувача

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів подібності щодо роботи, продукованими програмно-технічним засобом(ами) перевірки текстів на плагіат.
Назва: «Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів»
Автор: Юртаєв Денис Олександрович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Праворська Наталя Іванівна, канд. пед. наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені у 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 два дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої й електронної версії роботи.	
3	Виявлені запозичення не є плагіатом, але частково розміщені у 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 доопрацьована 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:

Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:

1) у тексті кваліфікаційної роботи системою перевірки на плагіат StrikePlagiarism виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, у назвах розділів/підрозділів, у назвах публікацій переліку джерел посилання тощо;

2) в якості запозичень системою StrikePlagiarism було зафіксовано деякі послідовності вихідного коду і посиланання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) запозичення, виявлені в тексті роботи, є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 6%. За системою StrikePlagiarism коефіцієнт подібності 1 складає 6% , коефіцієнт подібності 2 складає 1,8%.

Дата 02.12.2024

Завідувач кафедри ІПЗ — Леонід БЕДРАТЮК

Гарант освітньої програми — Оксана ЯШИНА

Керівник кваліфікаційної роботи — Наталя ПРАВОРСЬКА

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Д. О. Юртаєв

Співавтор:

Назва: Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів

Науковий керівник: канд. пед. наук, доцент Н. І. Праворська

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:6%

Коефіцієнт подібності 2:1.8%

Мікропробіли: 41

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2024-12-04 22:33:01.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2024-12-04 (Борщун Ю.В.)

Дата експерт

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач Юртаєв Денис Олександрович

Тема Метод використання неконтрольованого введення типу для мов програмування при автоматичній генерації тестів

Спеціальність 121 «Інженерія програмного забезпечення»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 0 ; кількість сторінок записки 112

1.Короткий зміст роботи та прийнятих рішень.

В роботі описується метод застосування введення типу під час генерації тестів. Пропонується використати метод генерації тестів з залученням технологій статичного та динамічного аналізу, в результаті чого ми зможемо покрити тестами більший обсяг коду.

2. Висновок про відповідність роботи поставленому завданню:

Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи:

Перший розділ роботи містить загальні думки та аналіз, який стосується предметної області автоматизованого тестування та наших нововведень, над якими працює дослідник. Також присутня інформація про деякі сучасні дослідження та існуючі інструменти. Другий розділ більш детально розкриває тему дослідження та згадані раніше наукові матеріали. Детально розглядаються необхідні концепції, такі як виведення типу, динамічні мови програмування та автоматичне тестування. В третьому розділі дослідник займається проектуванням всіх основних складових програмного інструменту, базуючись на інформації розглянутій в минулих розділах. Четвертий розділ стає завершальним. Дослідник продемонстрував результати, які він отримав з використання розроблюваного інструменту. Було проведено відповідні тестування та порівняв їх між собою. Ступінь використання досягнень науки і техніки цілком відповідають передовим методам.

4. Позитивні сторони роботи

Робота висвітлює важливу та досить цікаву тему, яка насамперед не просто про автоматизацію тестування, але й про роботу з неконтрольованими типами даних для динамічно типізованих мов програмування, що насправді є досить важливим. Серед позитивних сторін можна виділити основна ідея дослідження в повній мірі розкрита та наочно продемонстрована за допомогою відповідних тестувань та результатів.

5. Негативні сторони роботи

Серед негативних сторін роботи може бути лише надлишок деякої інформації, оскільки було розглянуто багато різних аспектів, які стосуються даного дослідження.

6. Оцінка графічного оформлення та пояснювальної записки роботи:

Таблиці та рисунки присутні в достатній мірі та в повному обсязі відповідають описаним в роботі елементам.

7. Відгук про роботу в цілому

Робота є досить цікавою та розглядає достатньо важливу тему.

8. Інші зауваження: відсутні.

9. Оцінка кваліфікаційної роботи

Робота оцінюється на задовільний рівень.

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) [illegible] Сергій Миколайович, д.т.н., професор, професор кафедри комп'ютерної інженерії та інформаційних систем, заступник декана з цифрових інформаційних технологій з наукової та міжнародної роботи.

«5» грудня 202 4 р. ______________ (підпис)