ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до магістерської роботи
на ступінь вищої освіти магістр

на тему: «**АВТОМАТИЧНЕ ВИЗНАЧЕНЯ ОБМЕЖЕНЬ ПАРАМЕТРІВ ДЛЯ СКЛАДНИХ ВЕБ-АРІ ДЛЯ ПЛАТІЖНИХ СИСТЕМ»**

Виконав: студент 6 курсу, групи ПДМ–61
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Пономарьов К.В.
(прізвище та ініціали)
Керівник Яскевич В. О.
(прізвище та ініціали)
Рецензент_____
(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти - «Магістр»
Спеціальність - 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
О.В. Негоденко
“___” _____20___ року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Пономарьову Костянтину Володимировичу
(прізвище, ім‘я, по батькові)

1.Тема роботи: «Автоматичне визначення обмежень параметрів для складних Веб-API для платіжних систем»
Керівник роботи к.т.н., старший викладач кафедри ІПЗ Владислав Олександрович Яскевич,
(прізвище, ім‘я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від “13” жовтня року № 230.
2. Строк подання студентом роботи 24.12.2020
3. Вихідні дані до роботи:
3.1 Вимоги до кваліфікаційної роботи магістра з актуальних завдань спеціальності;
3.2 Нормативні матеріали (стандарти, Гости);
3.3 Технічні вимоги;
3.4 Науково-технічна література з питань, пов‘язаних з темою роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
4.1 Порівняльний аналіз результатів, отриманих іншими авторами;
4.2 Методика дослідження;
4.3 Результати дослідження;
4.4 Висновки;
5. Перелік графічного матеріалу.

6. Дата видачі завдання 02.11.20

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	02.11.20	Виконано
2	Огляд існуючих рішень та літератури	03.11.20	Виконано
3	Вивчення документації API платіжної системи	12.11.20	Виконано
4	Збір кінцевих точок (параметрів) та їх обмежень з документації	16.11.20	Виконано
5	Розробка коду та вилучення обмежень параметрів зі статичного коду API	24.11.20	Виконано
6	Вступ, висновки, реферат	08.12.20	Виконано
7	Розробка обов‘язкових демонстраційних матеріалів	11.12.20	Виконано
8	Здача роботи	24.12.20	

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина магістерської роботи 62 с., 7 табл., 16 рис., 45 джерел.

API, ПАРАМЕТРИ, SDK, REST, HTTP, OAS, RAML, JAXB, ПЛАТІЖНІ СИСТЕМИ.

Об`єкт дослідження – платіжні системи та параметри їх веб-API.

Предмет дослідження – аналіз параметрів платіжних систем, розробка підходу автоматичного визначення обмежень параметрів, розробка відповідного програмного забезпечення.

Мета роботи – дослідження можливості автоматизації визначення обмежень параметрів для складних веб-API для платіжних систем.

Методи дослідження – аналіз, порівняння.

У роботі був проведений аналіз спільних особливостей електронних платіжних систем, визначено критично важливі аспекти їх функціонування.

Проаналізовано API платіжних систем.

Розглянуті два підходи автоматизації визначення обмежень параметрів для складних широкомасштабних API.

У роботі розглядається головним чином як кількість параметрів, які надає API, так і складність у самому вихідному коді. Детально розглянуто способи визначення обмежень параметрів, окреслено проблематику їх взаємодії.

Дослідницьке питання 1:

Наскільки ефективним є аналіз документації та статичного коду для виявлення обмежень параметрів у великому корпоративному API?

Аналіз коду та документації в сукупності отримує 66% обмежень між параметрами. Аналіз коду отримує 78% обмежень на один параметр.

Дослідницьке питання 2:

Які проблеми виникають під час аналізу документації або статичного коду

для виявлення обмежень між параметрами?

Щодо документації, безумовно, найпоширенішою причиною невиявлення обмежень є відсутність інформації, яка явно описує обмеження в Специфікаціях OpenAPI. Робота з цією відсутністю інформації є головною проблемою.

Для аналізу коду основними проблемами були нечутливість потоку даних та розробка підходу до розумного аналізу коду. Це включає рішення про те, як оцінювати циклічні структури управління, підтримуючи посилання на параметри у всьому коді API та вирішуючи, як поводитися з різноманітністю виразів мовою програмування.

Обраний найбільш ефективний підхід автоматичного визначення обмежень для складних веб-API.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API	прикладний програмний інтерфейс
SDK	набір із засобів розробки, утиліт і документації, який дозволяє програмістам створювати прикладні програми за визначеною технологією або для певної платформи (програмної або програмно-апаратної).
Framework	готовий до використання комплекс програмних рішень
REST	підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів
HTTP	клієнт-серверний протокол передачі даних
OAS	специфікація машиночитних файлів з інтерфейсами, для опису, створення, використання і візуалізації REST веб сервісів.
Guard	оператор раннього виходу. Подібно оператору if він перевіряє істинність заданої йому умови. Відмінність його роботи полягає у тому, що блок коду виконується тільки в тому випадку, якщо результат перевірки умови дорівнює значенню false.
RAML	мова на основі YAML, який використовується для опису інтерфейсів RESTful API.
YAML	зручний для читання людиною формат серіалізації даних, концептуально близький до мов розмітки, але орієнтований на зручність введення-виведення типових структур даних багатьох мов програмування.
JAXB	Java Архітектура для XML Binding дозволяє розробникам відображати(ставити у відповідність) класи Java у XML файли.

ВСТУП

Сьогодні бурхливо розвивається електронна комерція. Для розвинених країн купувати що-небудь в інтернеті стало вже звичною і буденною справою. У нашій країні також постійно зростає число бізнес-одиниць, які реалізовують свою продукцію за допомогою інтернет.

Щоб весь процес покупки відбувався в онлайн-режимі, був зручний і проходив в кілька кліків, існують електронні платіжні системи. Це відмінний інструмент, який допомагає зробити процес покупки дійсно комфортним для користувача, а торгівцю оперативно отримати кошти на свій розрахунковий рахунок.

Електронна платіжна система це підвид платіжної системи, яка забезпечує процес проходження транзакцій онлайн платежів через інтернет. Існує велика кількість систем прийому платежів, які доступні українському інтернет-торгівцю. Наприклад, сервіс Platon. Особливостями цієї системи можна назвати чіткий курс на підвищення конверсії інтернет-магазинів і онлайн-сервісів. Це забезпечується за рахунок гнучкого налаштування в системі протишахрайських фільтрів, високому показнику прохідності платежів, а також маркетингових фішках, які доповнюють платіжні рішення.

Платіжна система будь-якої країни є одним із головних елементів в діяльності як національної економіки в цілому, так і в банківській системі, оскільки виступає каналом зв’язку у загальній системі товарообігу країни. Платіжна система – це платіжна організація, члени платіжної системи та сукупність відносин, що виникають між ними при переведенні коштів. Проведення переказу коштів є обов’язковою функцією, що має виконувати платіжна система.

Платіжна система є набором процедур і правил, а також технічної інфраструктури. Все це в сукупності забезпечує можливість переказу грошових коштів безготівковим шляхом від одного суб’єкта іншому.

Сьогодні усі платіжні системи використовують інтерфейси програмування

веб-додатків (веб-API), що дозволяють програмам отримувати доступ до функціональних можливостей або даних служби за допомогою HTTP-запитів. При виконанні платежу API платіжної системи розраховує на банківський рахунок або картку як платіжну інформацію.

Веб-інтерфейси зазвичай надають посилання на API, яке описує, які операції доступні через яку кінцеву точку та які параметри є обов’язковими або необов’язковими для запитів до цих кінцевих точок. Однак ці параметри не завжди є просто обов’язковими або необов’язковими: чи потрібні вони, може залежати від наявності або значення іншого параметра. Отже, між параметрами існують залежності, а тому й обмеження.

Веб-API можуть мати обмеження на параметри, такі, що не всі параметри є або завжди обов'язковими, або завжди необов'язковими. Іноді наявність або значення одного параметра може спричинити необхідність використання іншого параметра. Крім того, параметри можуть мати обмеження щодо того, які типи значень є дійсними. Вони позначаються як багатопараметричні та однопараметричні обмеження відповідно.

Чіткий огляд обмежень у веб-API дуже важливий, оскільки це допомагає споживачам API інтегрувати API без необхідності підтримки компанії. Неповна або неправильна документація щодо цих обмежень може втратити багато часу та спричинити дорогі помилки інтеграції. В даний час ці обмеження задокументовані та підтримуються розробниками API вручну, що може бути трудомістким та складним завданням. Ця складність пов'язана з розміром та складністю кодової бази веб-служби, а також документацією, яку надають інші люди, ніж ті, хто пише код. Тому потрібні інструменти, які допомагають розробникам API виявляти та підтримувати обмеження в своїх API.

1 ДОСЛІДЖЕННЯ ПЛАТІЖНИХ СИСТЕМ

1.1 Розвиток платіжних систем

Перша масова карткова платіжна система під назвою Diners Club з'явилася в Америці в 1949 році. Спочатку компанія роздавала свої пластикові карти відвідувачам ресторанів, але згодом вони стали поширюватися через банки.

Технологія виявилася настільки вдалою, що вже через два роки карти Diners Club з'явилися навіть на іншому континенті - в Великобританії, а згодом - і по всьому світу.

Слідом за Diners Club виникли й інші відомі платіжні системи, такі як: Visa, MasterCard і American Express. Причому American Express, як і Diners Club спочатку мали лише кредитні картки. А ось MasterCard і Visa бувають як кредитними, так і дебетовими.

Платіжні системи є частиною грошової системи країни, їхня роль полягає в прискоренні грошового обороту та створенні більш зручних способів оплати. У Законі України «Про платіжні системи та переказ коштів в Україні» дається таке визначення терміну «платіжна система» - «це платіжна організація, учасники платіжної системи та сукупність відносин, що виникають між ними при проведенні переказу коштів. Проведення переказу коштів є обов'язковою функцією платіжної системи».

Розвиток платіжних систем в Україні характеризується нерозривним зв'язком із банківською системою, адже більшість платіжних систем, навіть не банківських, надають свої послуги кінцевим користувачам через банківські установи, що пов'язано з особливостями українського законодавства й розгалуженою мережею банківських відділень. Це дозволяє платіжним системам охоплювати значну частину території країни. Слід відзначити, що зазначена особливість характерна для більшості країн Європи, тоді як у країнах Азії значно більшу роль на ринку мають оператори мобільного зв'язку, а в США - спеціалізовані небанківські установи.

В Україні представлені всі можливі види платіжних систем:

- системи термінових грошових переказів;
- карткові платіжні системи;
- міжбанківські платіжні системи;
- платіжні системи на основі електронних грошей;
- термінальні платіжні системи;
- спеціалізовані платіжні системи.

1.2 Принципи функціонування платіжних систем

Платіжні системи повинні відповідати наступним вимогам:

- Стабільність у прийомі платежів.
- Вигідні тарифи.
- Гнучка система нарахування комісії за відсутності прихованих оплат.
- Тривала присутність на ринку (досвід), клієнтська підтримка, наявність сертифікатів.
- Простота інтеграції з сайтом.
- Логічний і комфортний для користувачів інтерфейс.
- Прийом усіх варіантів онлайн-платежів, що користуються попитом у споживчої аудиторії.

Незалежно від назви, географічної прив'язки та інших зовнішніх факторів, всі платіжні функціонують по одній схемі:

1. Споживач послуг вибирає дію, необхідну йому в даний момент - наприклад, зарахування грошей на карту, оплата покупки в інтернет-магазині і т. д.

2. Інформація про це надходить на термінал і далі - в банк, що обслуговує даного учасника.

3. Банк зв'язується з платіжною системою, а та, в свою чергу, - з банком-емітентом, в якому відкрито рахунок споживача.

4. Якщо з платоспроможністю клієнта немає проблем (тобто на рахунку є

кошти для здійснення платежу), банк видає дозвіл на списання грошей і відправляє відповідне розпорядження в процесинговий центр.

5. Інформація про транзакції повертається назад на термінал, де проводиться прийом платежу.

Складна, багатоступенева операція реально займає всього кілька секунд - всі відомості передаються у вигляді зашифрованого цифрового коду та не можуть бути зламані або перехоплені зовні.

В результаті швидкого обміну кожна зі сторін отримує свою вигоду: банк-емітент - комісію за надання послуг, торгова або сервісна точка - скорочення витрат за рахунок відсутності готівкових розрахунків, а покупці - знижки, швидку оплату і можливість здійснювати платежі в будь-якому місці.

1.3 Платіжні системи в Україні

Система електронних міжбанківських розрахунків Національного банку України експлуатується з 1993 р. і на сьогодні в цілому задовольняє вимоги та потреби банківської системи України.

Слід зауважити, що в Україні проблема функціонування платіжних систем залишається малодослідженою. Окремі питання, пов'язані зі сферою функціонування платіжних систем, досліджувалися в наукових працях таких вітчизняних і зарубіжних фахівців, як М. Вертузаев, В. Голубєв, О. Забродська, М. Зубок, Н. Єрьоміна, О. Котлярівське, В. Кравець, Л. . Миколаєва, І. Новак, Ю. Мельников, та інші.

Разом з тим слід зазначити, що розвиток сфери функціонування платіжних систем Україні характеризується постійним удосконаленням форм розрахунків, появою в зв'язку з цим нових суспільних відносин і прийняття відповідних нормативно - правових актів, які їх регулюють. Це є причиною того, що роботи зазначених учених не враховують повною мірою сучасний стан розвитку платіжних систем України, що обумовлює необхідність подальших досліджень у цьому напрямку.

Заслуговує уваги те, що в України в сучасних умовах функціонує дуже багато платіжних систем, які відрізняються між собою за роллю і характером розрахунків.

Система електронних міжбанківських розрахунків України — цілісна система, що базується на архітектурі «клієнт-сервер», призначена для виконання міжбанківських розрахунків у найзручнішій для конкретного користувача формі (on-line режим, кліринг тощо) та для моніторингу кореспондентських рахунків банківських установ України.

Єдиною системно-важливою платіжною системою в Україні, як і в попередні роки, залишається система електронних платежів Національного банку України.

За даними НБУ, статус соціально важливих платіжних систем підтвердили п’ять платіжних систем:

- «MasterCard», MasterCard International Incorporated, США;
- «Visa», Visa International Service Association, США;
- «Western Union», Western Union Financial Services Inc.США/Western Union Network, SAS, Франція;
- «FORPOST» (на сьогодні – NovaPay), ТОВ «Пост Фінанс»;
- «Поштовий переказ», ПАТ «Укрпошта».

За результатами моніторингу за 2020 рік значущими операторами послуг платіжної інфраструктури є ПрАТ «Український процесінговий центр» та ТОВ «ТАС ЛІНК».

2 АНАЛІЗ API В ПЛАТІЖНИХ СИСТЕМАХ

У цьому розділі надається уявлення про літературу, яка стосується обмежень параметрів API, та його якість. Розглядаються існуючі підходи, пов’язані з визначенням обмежень параметрів, і те, як вони виглядають на практиці.

2.1 Зручність використання та якість API

На сьогоднішній день дослідження API активно збільшуються.[25] Наявна література розширює як технічні, так і програмні аспекти атрибутів якості API. Технічні аспекти зосереджені на таких речах, як архітектурні стилі [18, 31], формати специфікацій (RAML, OAS) та вимоги до якості програмного забезпечення [12]. Програмні аспекти зосереджені на зручності використання API [20, 26, 28, 10]. Поділ на програмні та технічні аспекти не є однозначним. Незважаючи на те, що API в найширшому розумінні відносяться до загального типу інтерфейсу програмування, наша увага приділяється веб-API: веб-сервісам через HTTP.

Зручність використання, як аспект якості програмного забезпечення, часто зустрічається в літературі з дизайну API, оскільки в кінцевому підсумку API споживаються людьми для створення конкретних функціональних можливостей для власного використання. Зручність використання при розробці API пов'язана з технічними аспектами проектування API за допомогою декількох аспектів дизайну, таких як загальна архітектура та організація класів [34, 4], що, в свою чергу, впливає на сумісність та ремонтопридатність [33]. Будь-яке з таких рішень може мати вплив на загальну зручність використання API. Це і є взаємозв'язком між технічними та програмними аспектами якості API.

Що стосується технічних аспектів, які мають більш прямий зв'язок до зручності використання, архітектурні стилі API легко обговорюються. Для більшої наочності розглядаються RPC, REST та GraphQL як основні стилі [7]. REST - безумовно найпопулярніший стиль [27]. Вибір того чи іншого стилю залежить від

варіанту використання. RPC може бути корисним для служб, орієнтованих на дії. REST здебільшого використовується для сервісів, орієнтованих на ресурси, і GraphQL - для високореляційних даних [7, 32]. Зрештою, все це впливає на те, як архітектори програмного забезпечення думають про їх реалізацію та спосіб надання функціоналу користувачам API.

Що стосується програмних аспектів, використання API не є суворо визначеним у літературі. У рамках цієї теми основна увага приділяється ефективності та зрозумілості [25, 12]. Існують різні роботи з ідентифікації зручності використання факторів, а іноді надають для них метрики [25]. Здатність читати та запам'ятовувати є загальними показниками. Відсутність документації є основною перешкодою для засвоєння API. Robillard та DeLine [26] визначають п’ять факторів документації, що впливають на навчальний досвід розробників: документація про наміри, приклади коду, сценарії використання, формат та представлення.

Для розробників використання API є ключовим у процесі інтеграції. Навчальні перешкоди можуть призвести до вибору іншого сервісу або посилення інтеграційних зусиль для підтримки споживачів API. Дослідження показують, що значна частина проблем при інтеграції API може бути пов’язана з недійсними або відсутніми даними, введеними користувачем [3]. Ці помилки інтеграції стосуються обмежень параметрів, таких як відсутність необхідних параметрів або недійсні значення для наданих параметрів.

2.2 Обмеження параметрів веб-API на практиці

Дослідження частоти багатопараметричних обмежень для різних галузей [17], враховуючи лише REST API дає змогу побачити, що 85% REST API мають багатопараметричні обмеження, і в середньому 9,8% операцій містять обмеження. Деякі дослідження повідомляють порівнянні цифри [21, 35].

Більшість обмежень не є складними, лише 4% залежностей у REST API класифікуються як складні [17]. Складні обмеження включають декілька категорій

із наведених нижче категорій. Щодо нескладних обмежень, усі три категорії є приблизно однаково поширеними.

Існують такі категорії обмежень:

• **Ексклюзивні**: повинен бути присутній рівно один із набору параметрів.

• **Залежні**: наявність або значення одного параметра залежить від наявності або значення іншого параметра.

• **Групові**: група параметрів повинна бути цілком присутня або цілком відсутня.

Тоді як різні категорії, можливо, є підмножинами залежної категорії з точки зору логічної еквівалентності, корисно мати семантичну різницю між ними. В роботі розглядається такі категорії та підкатегорії:

• **Залежні**

P1 → *P2*: Наявність *P2* залежить від присутності *P1*.

P1 = V → *P2*: Наявність *P2* залежить від значення *P1*.

P1 = V → *P2 = V2*: Значення *P2* залежить від значення *P1*.

• **Ексклюзивні**

P1 or P2: потрібен *P1* або *P2*, за умови, що обидва є дійсними.

P1 xor P2: потрібні *P1* або *P2*, за умови, що обидва недійсні.

• **Групові**

P1 ↔ *P2*: потрібні обидва або жоден.

• **Арифметичні**

P1 > P2: P1 має бути більшим за *P2*.

P1 > P2 - X: *P1* має бути більше *P2* за мінусом деякої константи.

P1 + P2 + P... = Pn: сума параметрів повинна дорівнювати значенню іншого параметра.

Термін «(багато) параметричні обмеження», не використовується як такий послідовно в літературі [21], в інших роботах (в опрацьованій літературі) на це поняття посилаються по-різному. Деякі інші терміни, що використовуються для посилання на те саме поняття є «(багато) параметричні залежності параметрів» [17], «обмеження залежності» [35] або, загальніше, «специфікації методів» [22].

У цій роботі використовується термін «(багато) параметричні обмеження», з наступними визначеннями: обмеження між параметрами описує вимогу щодо наявності або значення параметра на основі присутності або значення іншого параметра, для даної операції веб-служби [35]. Обмеження з одним параметром описують вимогу щодо наявності або значення одного параметра. Вважаємо щось вимогою, якщо недотримання цього призводить до того, що запит не вдається, див. Розділ 3.1.2. Як результат, будь-які обмеження, що призводять до іншого результату, ніж передбачено запитом, не є суворими обмеженнями параметрів.

$$
\begin{aligned}
\mathsf{v} \in \text{Values} &::= \text{Number, String, Boolean or Parameter} \\
\mathsf{f} \in \text{Parameters} &::= \mathsf{s} \mid \mathsf{f.s} \mid \mathsf{f.[]} \\
\mathsf{t} \in \text{Types} &::= \mathsf{string} \mid \mathsf{number} \mid \mathsf{boolean} \mid \mathsf{object} \mid \mathsf{t[]} \mid \mathsf{null} \\
\mathsf{cd} \in \text{Constraint definitions} &::= \mathsf{s}(\mathsf{s}_1, ..., \mathsf{s}_n) = \mathsf{c} \\
\mathsf{c} \in \text{Constraint} &::= \mathsf{o} \mid \mathsf{lc} \mid \mathsf{s}(\mathsf{v}_1, ..., \mathsf{v}_n) \\
\mathsf{o} \in \text{Operations} &::= \mathsf{present(f)} \mid \mathsf{type(f)}{=}\mathsf{t} \mid \mathsf{length(f)} \oplus \mathsf{v} \mid \mathsf{value(f)} \oplus \mathsf{v} \\
\mathsf{lc} \in \text{Logical connectives} &::= \mathsf{and(c,\ c)} \mid \mathsf{or(c,\ c)} \mid \mathsf{not(c)} \mid \mathsf{implic(c,\ c)} \mid \mathsf{iff(c,\ c)} \\
\oplus \in \text{Math operators} &::= =, \mathsf{!=}, <, >, \mathsf{<=}, \mathsf{>=}
\end{aligned}
$$

Рисунок 2.1 – Машиночитний синтаксис мови обмежень від Oostvogels [21].

2.3 Машиночитне представлення

Oostvogels[21] надає машиночитну мову для представлення обмежень

параметрів. Ця мова специфікації, орієнтована на обмеження, підтримує представлення обмежень між параметрами, а також обмежень з одним параметром. Цей синтаксис можна побачити на рис. 2.1.

Ця мова використовується, щоб представити обмеження, з незначними відмінностями у представленні. Наприклад, за умови обмеження, що «якщо наведено A, значить потрібен B», ми будемо представляти це як $A \rightarrow B$, на відміну від імплікативного (присутній (A); даний (B)).

2.4 Автоматичне визначення обмежень параметрів

Існує декілька статей, що окреслюють підходи до автоматичного визначення обмежень параметрів складних веб-API. Ці підходи покладаються на документацію, відповіді API або аналіз коду, щоб вивести такі обмеження.

Gao та ін. [11] використовують підхід на основі дерева рішень для виведення обмежень між параметрами. Інформація для заповнення дерева рішень виводиться із спостереження відповідей API для даного обмеження кандидата. Ці кандидати обираються з використанням набору евристики та з урахуванням відгуків API. Останнє включає аналіз повідомлень про помилки, наданих API як зворотний зв’язок. Тоді як підхід зміг вивести 145 із 154 обмежень, визначених вручну, було оцінено невелику кількість API. Досліджувані API містять приблизно 5 параметрів на кінцеву точку в середньому.

Робота Atlidakis та співавторів [2] використовує підхід нечіткого типу для пошуку залежностей між параметрами для різних кінцевих точок. Тобто він спрямований на виявлення залежностей між параметром в кінцевій точці A та іншою кінцевою точкою B. Для керування процесом нечіткого використання використовуються специфікації OpenAPI та зворотний зв'язок з відповідями API. Нечіткий підхід запускає більшу кількість запитів API, приблизно 5000.

Ці підходи розроблені для виведення обмежень з документації, але не розглядають код як вхідні дані. Wu та ін. [5] передбачає автоматичну ідентифікацію обмежень між параметрами шляхом виведення кандидатів на обмеження за допомогою онлайн-посилання на API та доступних наборів для розробки програмного забезпечення (SDK). Потім цих кандидатів перевіряють, зателефонувавши до державної веб-служби із органами запитів, які задовольнять або порушать обмеження щодо кандидатів. Підхід Wu та співавторів використовує комбінацію NLP та аналізу потоків даних для документації та аналізу SDK відповідно. Далі розглянемо подробиці їхнього підходу, оскільки використовуємо його як ідею для підходу в цій роботі.

Для аналізу документації вони використовують підхід двофазної фільтрації,

який називається «вільною» та «жорсткою» стратегією. Жорстка стратегія використовує заздалегідь визначені шаблони для узгодження з текстом. Наприклад, шаблон «A або B має бути вказано» відповідає конкретному реченню з використанням відстані Жаккара. Вільна стратегія «генерує кандидата на обмеження, коли кількість різних параметрів, що з'являються в описуваному реченні, збігається з кількістю параметрів у шаблоні» [5]. Якщо два параметри зустрічаються в одному і тому ж реченні, то, як вважається, вони якимось чином пов'язані. Вільний вибір використовується як найкраща стратегія у статті.

Для аналізу SDK API вони виконують аналіз потоку даних на графіках управління потоками, сформованих методами SDK. У цьому процесі генеруються всі можливі комбінації наборів параметрів, які потім використовуються для виведення обмежень. Процес генерації цих комбінацій залежить від гілок на графі управління потоком окремого методу та комбінацій параметрів, доступних у цих гілках. Літерал або змінна в коді включається в потік даних, коли він відповідає імені параметра, доступному в онлайн-документації. Метод виведення обмежень з комбінацій не пояснюється.

Підхід досягає точності та відкликання близько 95% для чотирьох простих API. У той час як підхід спирається як на код, так і на документацію за проектом, результати вказують на те, що більшість обмежень між параметрами виведені з документації. Документація надала загалом 351 кандидата, а код - 36 кандидатів. Кандидати, засновані на документації, мали меншу точність, ніж кандидати, засновані на SDK, - 20,8% та 100,0% відповідно, але протилежне вірно для відкликання - 82,9% та 40,9% відповідно.

2.5 Технології, які використовуютья у роботі

2.5.1 GraphQL

GraphQL – це стандарт декларування структури і способів отримання даних або синтаксис, який описує, як можна зчитувати дані з серверу [5].

GraphQL має три основні характеристики:

- дозволяє клієнту точно вказати, які дані йому потрібні;
- полегшує агрегацію даних з декількох джерел;
- використовує систему типів для опису даних.

При такому підході, крім гнучкості, зменшується кількість запитів та обсяг даних на транспортному рівні. GraphQL API базується на трьох основних будівельних блоках: схемі (schema), запитах (queries) і розпізнавачах (resolvers). У GraphQL передбачені такі види операцій: запит (зчитування даних), мутація (запис даних) або підписка (безперервне зчитування даних) [8]. Будь-яка з таких операцій – просто рядок, який необхідно зібрати відповідно до специфікації мови запитів GraphQL. Як тільки така операція прийде на сервер з клієнтської програми, її можна буде інтерпретувати за допомогою усієї схеми GraphQL і віддати дані, яких потребує клієнтський застосунок. GraphQL може працювати з будьяким високорівневим мережевим протоколом (найчастіше використовується HTTP) та з будь-яким форматом даних (зазвичай використовується JSON).

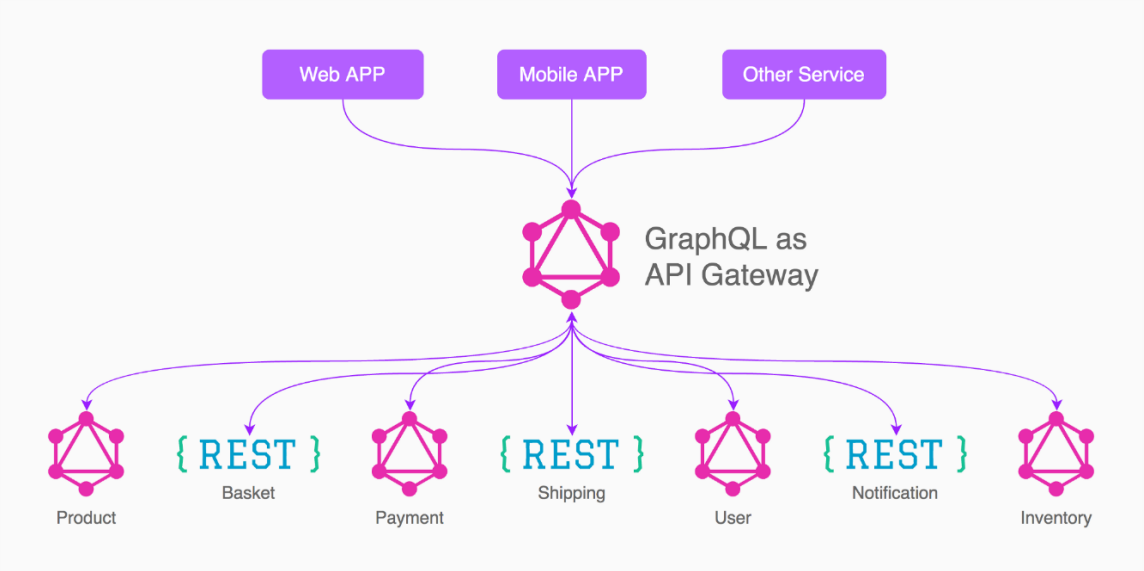


Рисунок 2.2 – Типове використання GraphQL

Перевагою використання GraphQL є декларативність. Також, до переваг відносять сильну та чітку типізацію, відсутність проблем з версіонуванням. Ще один важливий аспект GraphQL – його ієрархічний характер. GraphQL

побудований на взаємозв'язку між об'єктами, що спрощує формування запитів, де службі RESTful може знадобитися багаторазова система запитів «request/response» або складна операція об'єднання в SQL [15]. Головним недоліком прийнято вважати складність реалізації на серверній частині. Зазвичай, саме з цієї причини GraphQL використовують як додатковий шар між клієнтом і веб-сервісами. При цьому веб-сервіси не використовують GraphQL, а надають доступ до даних за допомогою REST API.

Завдання автоматизації та оптимізації процесу створення документації є неактуальним, адже вищезгадана GraphQL-схема і є документацією для користувачів мережевих програмних прикладних інтерфейсів. Більше того, вже створено GraphiQL – кросплатформений інструмент для розробки застосунків з використанням GraphQL [23], графічний інтерфейс якого зображений на рис. 1.2.

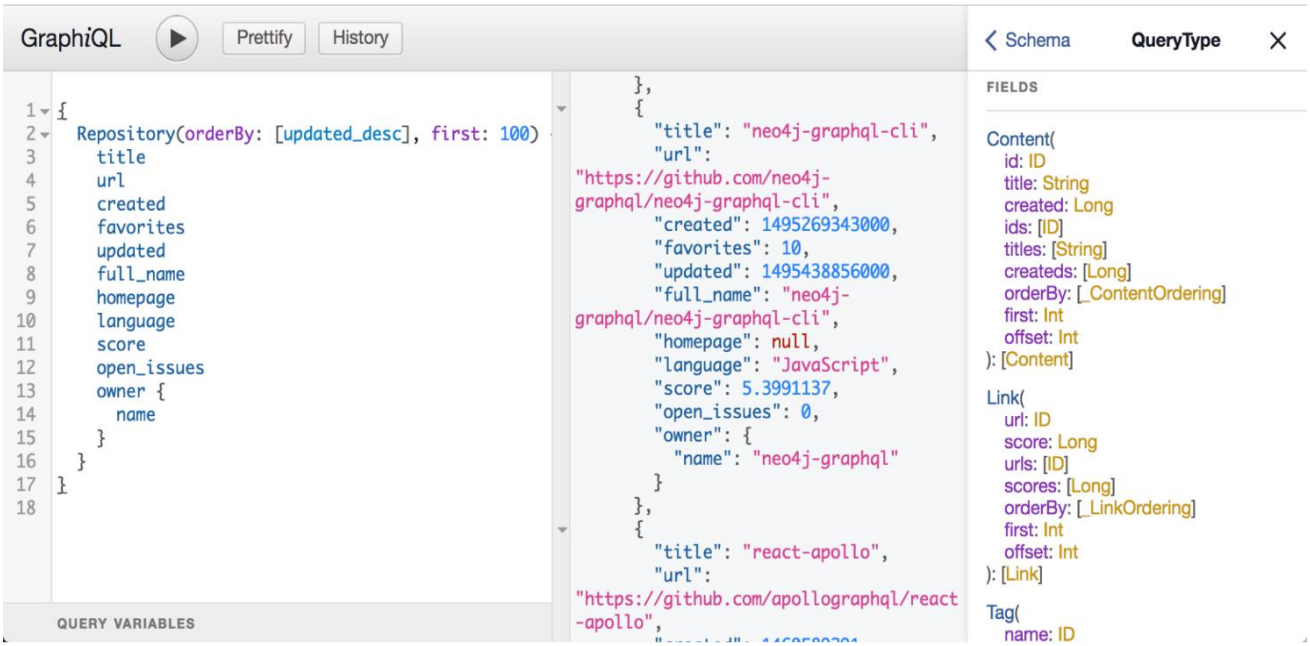


Рисунок 2.3 – Графічний інтерфейс інструменту GraphiQL

За допомогою цього інструменту програміст має можливість переглянути документацію в читабельному вигляді, яка буде автоматично згенеровано на основі схеми. Крім того, переглядаючи документацію, можна відправляти запити або мутації та відразу переглядати результат в зручній формі.

2.5.2. JSON-RPC

JSON-RPC – це протокол віддаленого виклику процедур, який використовує JSON в якості формату даних. Даний протокол багато в чому схожий на XML-RPC, його специфікація визначає кілька типів даних і правила їх обробки [24]. Він розроблений, як простий, гнучкий та зрозумілий для всіх стандарт. JSON-RPC базується на відсиланні запитів до сервера, який реалізує віддалений протокол.

Всі дані, що передаються – запити, серіалізовані в JSON. Запит – виклик певного методу, наданого віддаленою системою. Він повинен містити три обов'язкових компоненти:

- «method» – рядок з назвою методу;
- «params» – дані, які передаються методу як параметри;
- «id» – значення будь-якого типу, яке використовується для встановлення відповідності між запитом і відповіддю.

Сервер повинен повернути відповідь на кожен отриманий запит. Відповідь повинна містити такі властивості:

- «result» – дані, які повертає метод. Якщо сталася помилка під час виконання методу, ця властивіст повинна бути встановлена в null;
- «error» – код помилки у випадку помилки під час виконання методу, інакше null;
- «id» – те ж саме значення, що і в запиті, до якого відноситься дана відповідь.

Для ситуацій, коли відповідь не потрібна, були введені нотифікації. Нотифікації відрізняються від запиту відсутністю «id».

Як основну перевагу JSON-RPC можна відзначити його простоту та інтуїтивність. Часто при розробленні API програмісти, які нічого не знають про стандарти, самі проектують інтерфейси зі схожою структурою запитів та відповідей. JSON-RPC добре підходить для веб-сервісів з невеликим обсягом функціональності та типів даних. Проте, нестача механізмів кешування та версіонування, відсутність чіткої специфікації роблять даний стандарт непридатним для об’ємних веб-сервісів.

Стандарт JSON-RPC є дуже простим, тому простою є і задача генерування документації для API, які використовують цей стандарт. Зокрема, все, що повинна містити документація – це список методів, параметрів, відповідей та кодів помилок. З цієї задачею добре справляються реалізації цього стандарту для різних платформ. Зокрема, це – JSON-RPC.NET для платформи .NET, go/net/rpc для GoLang, php-json-rpc для PHP [36].

2.5.3. SOAP

SOAP – протокол обміну структурованими повідомленнями в розподілених обчислювальних системах [37]. Для протоколу SOAP не існує різниці між викликом процедури і відповіддю на виклик, він просто визначає формат послання (message) у вигляді XML-документу. Послання може містити виклик процедури, відповідь на нього, запит на виконання якихось інших дій. Специфікація SOAP не використовує аналіз вмісту послання, вона задає тільки його стандарт для його оформлення.

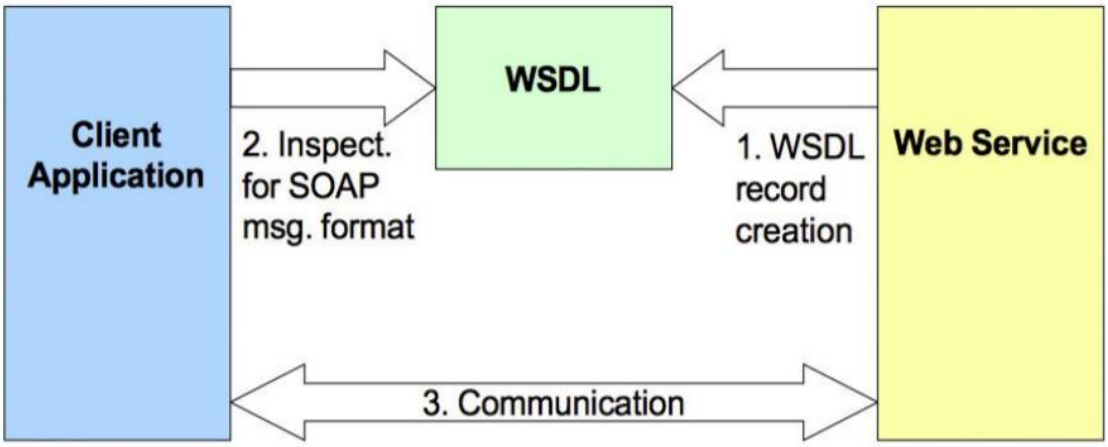


Рис. 2.4. Обмін повідомленнями між клієнтом і сервером з використанням SOAP

SOAP заснований на мові XML і розширює один з протоколів прикладного рівня – HTTP, FTP, SMTP і т.д. Як правило, найчастіше використовується HTTP. SOAP-повідомлення є XML-документом, який складається з трьох основних елементів: конверт (SOAP Envelope), заголовок (SOAP Header) і тіло (SOAP Body).

До переваг використання стандарту SOAP можна віднести його незалежність від транспортного протоколу та дуже чітку специфікацію. Водночас, недоліками є його складність та негнучкість, а також обмеження форматом XML, який є доволі об’ємним з точки зору транспортування.

Для створення документація для SOAP-API використовують WSDLмову

визначення інтерфейсу сервісу, яка заснована на форматі XML, що описує функціональність сервісу і спосіб доступу до нього [38]. Цю мову використовують як для створення документації, так і для генерації коду, тому задача створення документації для SOAP-API є неактуальною.

2.5.4. REST API

REST – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів [39]. Був описаний і популяризований Роєм Філдінгом, одним із творців протоколу HTTP. Філдінг розробив REST паралельно з HTTP 1.1 базуючись на попередній версії 1.0 [40].

REST, як і кожен архітектурний стиль, має декілька архітектурних обмежень. Одне з обмежень – це клієнт-серверна архітектура. Архітектура такого типу вимагає розділення відповідальності між компонентами, які займаються зберіганням та оновленням даних (сервером), і тими компонентами, які займаються відображенням даних на інтерфейсі користувача та реагування на дії з цим інтерфейсом (клієнтом) [30]. Таке розділення дозволяє компонентам працювати незалежно. Наступним обмеженням є те, що взаємодії між сервером та клієнтом не мають стану, тобто кожен запит містить всю необхідну інформацію для його обробки, і не покладається на те, що сервер знає щось з попереднього запиту. Додатковим обмеженням стилю REST є те, що системи, написані в цьому стилі, повинні підтримувати кешування, тобто дані, які передаються сервером, повинні містити інформацію про те, чи можна їх кешувати, і якщо можна, то як довго. Це дозволяє збільшувати продуктивність, уникаючи зайвих запитів, але також зменшує надійність системи через те, що дані в кеші можуть бути застарілими.

На практиці, REST API – це набір URI, HTTP викликів до цих URI та велика кількість представлень ресурсів у форматі JSON або XML, багато з яких будуть містити перехресні посилання. За основу адресації береться покриття ресурсів унікальними ідентифікаторами. Обмеження одноманітності інтерфейсу частково реалізовано за допомогою комбінацій URI і HTTP дієслів і їх використанням відповідно до стандартів і конвенцій. Ресурси повинні бути іменниками, а дія над ресурсом – це дієслово. URI завжди повинен посилатися на ресурс, а не на дію.

URI	HTTP Verb	Outcome
.../api/employees	GET	Gets list of employees
.../api/employees/1	GET	Gets employee with Id = 1
.../api/employees	POST	Creates a new employee

Рисунок 2.5. Типовий ресурс URI та набір для доступу до нього

Документація до REST API повинна містити набір всіх ресурсів, їх ідентифікаторів та представлень, а також набір URI та HTTP-дієслів для доступу до ресурсів, інформацію про авторизацію, можливі коди помилок та відповідей. Сама специфікація REST API не передбачає ніяких автоматичних механізмів для створення документації, як це відбувається при використанні GraphQL за допомогою схеми, або в SOAP за допомогою WSDL. Водночас документація до

REST API є важливим артефактом для клієнтської частини, а процес її створення та підтримки можна оптимізувати, детально проаналізувавши вимоги та методи.

3 ІСНУЮЧІ ПІДХОДИ

Дані підходи мають два різних типи аналізу: аналіз документації та аналіз коду. Метою обох підходів є автоматичне отримання обмежень веб-API в машиночитному поданні. Аналіз коду витягує обмеження з одним параметром (наприклад, $X > 5$) та багатопараметричні обмеження (наприклад, X або Y), аналіз документації лише визначає обмеження між параметрами. Огляд процесу на високому рівні наведено на рис. 3.1.

На першому кроці збирається інформація про параметри кінцевих точок веб-API із специфікації OpenAPI (OAS). Найважливішою службовою інформацією є: тип даних кожного параметра, необхідний

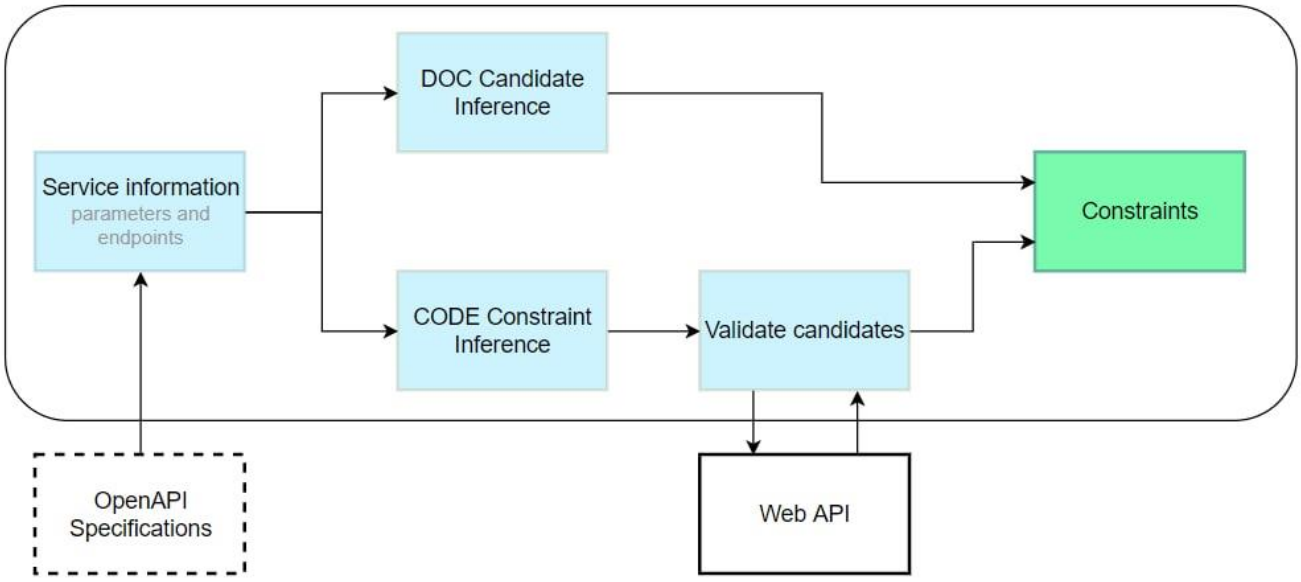


Рисунок 3.1 – Огляд процесу, що показує початковий етап збору інформації про послуги, а потім підхід до аналізу документації та коду.

параметр, будь-які значення перерахування, їх опис, а також материнські та дочірні параметри. Ця інформація допомагає виконувати ряд завдань, таких як генерація значень за замовчуванням для надсилання запитів до API та виявлення посилань на параметри. Ця інформація використовується в обох підходах.

Для підходу, заснованого на документації, проводиться аналіз текстової

документації, щоб зробити висновки про обмеження. Цей процес має два етапи. Спочатку витягуються набори параметрів кандидата з описів OAS. Як визначаються ці кандидати описано у Розділі 3.1. Кандидати збираються для зменшення загальної кількості запитів, необхідних для висновку про обмеження. Оскільки обмеження можуть бути між будь-якими двома або більше параметрами, перевіряються всі комбінації на наявність обмежень. Спроба всіх комбінацій вимагає експоненціальної кількості запитів щодо кількості параметрів, що призводить до занадто великої кількості запитів.

По-друге, перевіряються кандидати, зібрані на попередньому кроці. Кандидати лише показують нам, які параметри можуть мати обмеження між ними. Під час перевірки ставиться задача зробити точне обмеження, яке застосовується.

Для кодового підходу розглядається структура управління методами у вихідному коді для вилучення обмежень. Можна зробити висновок про використання параметрів у цій структурі управління та про передумови, що застосовуються до їх використання. Наприклад, *if (X != null){Y}* дозволить зробити висновок про необхідність Y з передумовою надання X у запиті.

3.1 Аналіз документації

Аналізуючи текстову документацію веб-API, можна зробити висновок про наявність обмежень між параметрами. Аналіз документації складається з двох різних кроків: пошуку наборів параметрів, які можуть мати обмеження між ними (кандидатами), а потім визначення точних обмежень між параметрами за допомогою надсилання запитів API до предметного API (валідація).

3.1.1 Формування кандидатів

Використовується опис параметрів OAS для пошуку кандидатів. API Explorer Platon візуалізує ці параметри, а також їх описи для всіх загальнодоступних кінцевих точок. Імовірно, що опис одного параметра може

посилатися на інший параметр, який натякає на можливе обмеження між двома параметрами. Наприклад з урахуванням параметра *bankAccount* з описом «Реквізити банківського рахунку. Потрібен параметр *bankAccount*, або *card*.», припускаємо, що ці два можуть мати обмеження між собою.

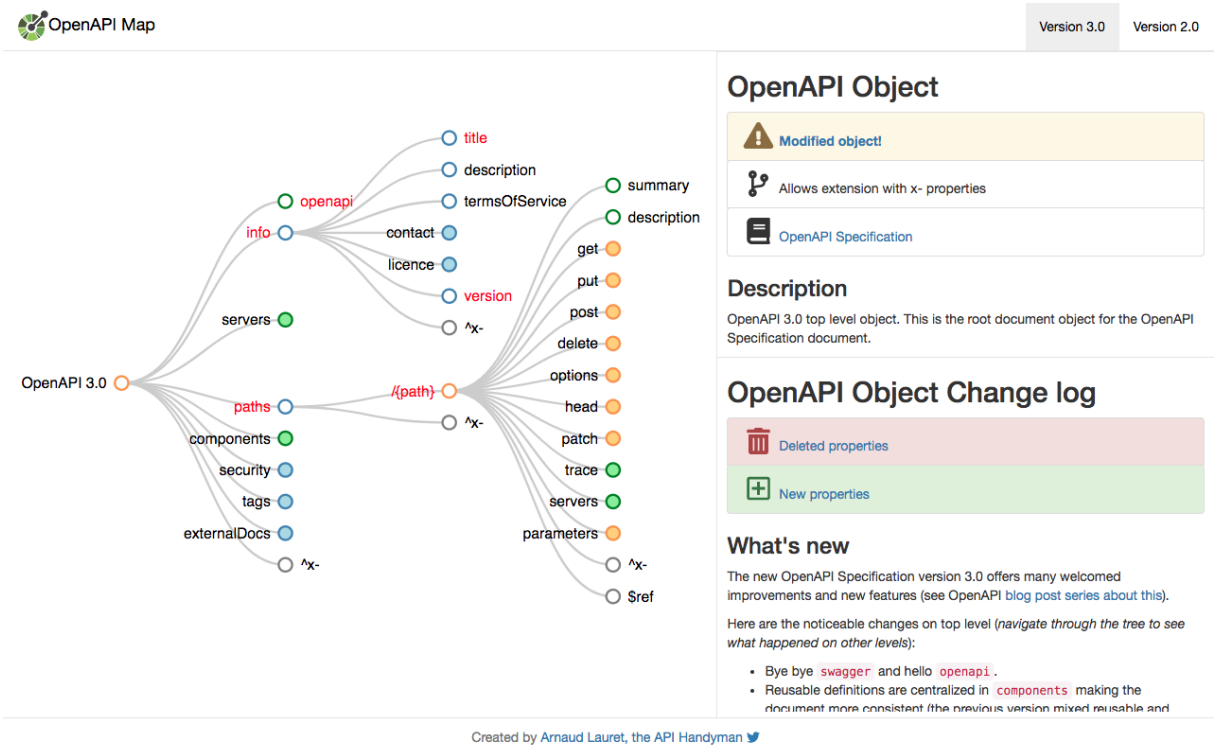


Рисунок 3.2 – Карта Open Api (OAS)

Для формування кандидатів ми використовуємо матрицю співіснування [6]. Ця матриця співіснування містить рядок і стовпець для кожного параметра в даній кінцевій точці. Для заповнення цієї матриці співіснувань ми автоматично аналізуємо опис кожного параметра; якщо опис параметра містить ім'я іншого параметра, то їх відповідний запис у матриці оновлюється. Наприклад для попереднього прикладу клітинка, що відповідає банківському акаунту і картці, буде оновлена на одиницю.

Чи потрібен параметр, може залежати від значення іншого параметра. Такі обмеження, що залежать від вартості, вимагають отримання додаткової інформації на етапі перевірки. Більш конкретно, ми повинні знати, які значення є важливими для яких параметрів. Робиться це, перевіряючи, чи в описі згадується якесь із

значень переліку, яке надає OAS. Якщо значення переліку параметра згадується в описі, тоді це значення позначається та використовується на наступному етапі перевірки.

Табл. 3.1 – Приклад матриці співіснування для кінцевої точки з чотирма параметрами. Будь-який запис X траплявся занадто часто і ігнорується.

	card	**bankAccount**	**type**	**reference**
card	-	2	0	X
bankAccount	-	-	1	X
type	-	-	-	X
reference	-	-	-	-

Деякі параметри можуть траплятися надзвичайно часто в описах. Це часто пов’язано з тим, що імена параметрів є загальноприйнятими як слово в природному тексті. Такі слова, як «reference» та «value», як правило, використовуються, не будучи посиланнями на параметр. Це б дало багато нерелевантних кандидатів. Отже, ми ігноруємо параметри, які спільно зустрічаються із занадто багатьма іншими параметрами. У табл. 3.1 це позначено знаком X.

Генеруємо набори кандидатів на основі матриці суміжності. Ми генеруємо кандидатів з кожного рядка матриці. Для кожного рядка включаємо параметр, що відповідає цьому рядку в кандидаті, та будь-які параметри, які відбулися принаймні один раз із цим параметром. Кандидати, що мають лише один параметр, видаляються. Дотримуючись табл. 3.1, створюємо набори [*card, bankAccount*] та [*bankAccount, type*]. Після цього ці кандидати використовуються як вхідні дані до етапу перевірки.

3.1.1.1 Розбір семантики

Документація також містить більш конкретну інформацію про саме точне обмеження. Можна знайти такі описи, як «x або y». Ця семантика не розглядається в обмеженнях через труднощі, пов’язані з точним синтаксичним аналізом такої семантики, що призводить до втрати точності. Не всі описи

параметрів обов’язково точні або семантично чіткі. У випадку «A або B» незрозуміло, чи використовується воно виключно чи включно. Невиконання будь-якого з таких припущень означає, що всі комбінації мають бути випробувані. Що можливо, враховуючи малий розмір кандидатів.

Досліджено використання вбудовування слів [19] як засіб пошуку кандидатів. Використовуючи вбудовані слова, можна отримати семантичну подібність між словами та реченнями. Передумова полягає в тому, що параметри із семантично подібними описами частіше пов'язані між собою в обмеженні. На практиці такий вид міри подібності не відфільтрував кількість можливих комбінацій параметрів настільки, щоб бути життєздатним самостійно.

3.1.2 Перевірка кандидатів

В результаті аналізу документації отримуємо набори параметрів, які можуть мати обмеження між ними (кандидатами), і для кожного параметра, значення яких були знайдені в документації. Метою є з’ясувати точне обмеження між параметрами, яке застосовується до цих параметрів, якщо такі є. Робиться це, генеруючи запити та спостерігаючи відповідь API на збій. Якщо запит не вдається, це говорить про те, що певне обмеження не було виконано.

Табл. 3.2 – Таблиця перевірки для двох параметрів, де 0 та 1 вказують на відсутність та наявність параметрів. У стовпці результату вказується, чи був запит успішним (T) чи ні (F).

card	bank	Result	Constraint
0	0	F	-
0	1	T	and(!card, bank)
1	0	T	and(card, !bank)
1	1	T	and(card, bank)

Формується таблиця істинності для кожного кандидата. У цій таблиці істинності кожен рядок вказує поточні або відсутні параметри та чи успішним був

результат відповідного запиту. У такій таблиці істинності представляються всі можливі комбінації параметрів. Прикладом такої групи є [*card*; *bank*], для якої відповідну таблицю істинності можна побачити в табл. 3.2. Для кожного рядка в цій таблиці генерується базовий запит, включаючи параметри, зазначені як наявні, і параметри, позначені як відсутні, видаляються. Потім цей запит надсилається до API, і відповідь перевіряється на помилку. Якщо запит не вдається, то результат оновлюється відповідно.

Відповідно до табл. 3.2, будь-яке з обмежень, витягнутих із успішних рядків, має бути істинним, щоб задовольнити обмеження. Диз'юнктивна нормальна форма між вдалими рядками утворює обмеження, яке застосовується між банком та картою. Якщо всі запити успішні, тоді жодне обмеження не застосовується.

Табл. 3.3 – Таблиця перевірки для двох параметрів, стан може бути «gas» або «solid». Рядки, в яких обидва стовпці «state» мають значення true, виключаються.

state = gas	state = solid	temperature	Result
0	0	0	F
0	0	1	T
0	1	0	F
0	1	1	T
1	0	0	T
1	0	1	T

Для кандидатів, які містять параметри з позначеними значеннями, цей процес працює дещо інакше. Для параметра з позначеними значеннями всі значення, визначені на попередньому кроці, додаються як стовпець до таблиці істинності. Окрім позначених значень, включено стовпець, що містить неідентифіковане значення. Це має бути більшою впевненістю у тому, що обмеження насправді залежить від значення параметра, а не просто від наявності параметра. Приклад цього процесу можна побачити в табл. 3.3. У цьому випадку *state* дорівнював *gas* як ідентифіковане значення переліку.

3.1.2.1 Створення запиту

Створення дійсних запитів є основною частиною перевірки раніше створених кандидатів. Для цього запити будуються шляхом модифікації базового запиту відповідно до модифікацій, накладених таблицею істинності. Після такої таблиці істинності параметри, зазначені як присутні, включаються, а параметри, зазначені як відсутні, вилучаються з базового запиту. Наприклад, враховуючи, що базовий запит включає параметр *accountName*, то на основі таблиці 3.1 ми генеруємо чотири тіла запиту, показані на рис. 3.3.

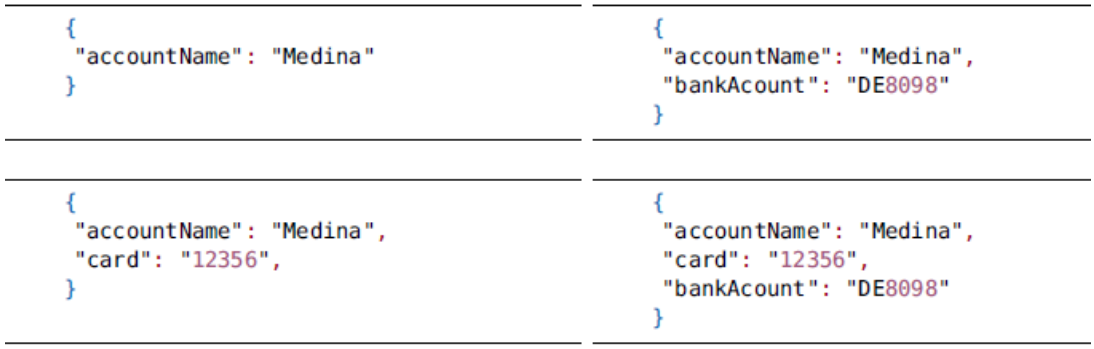


Рисунок 3.3 – Приклади тіл запитів, створених на основі таблиці 3.1.

3.1.2.2 Створення базового запиту

Базовий запит – запит за замовчуванням, вказаний для кожної кінцевої точки, який завжди повинен мати успішний результат. Ці запити за замовчуванням можуть бути вказані вручну, або вони можуть бути сформовані з OAS. OAS визначає необхідні параметри, включаючи всі, що має призвести до дійсного базового запиту. Коли цього не було, потрібно було вручну додати відсутні параметри до базового запиту.

3.1.2.3 Генерування значень параметрів

Параметри, надані в запиті, повинні мати дійсні значення. Те, що вважається дійсним, залежить від того, які значення вагомі для даного параметра. Наприклад, якщо параметр представляє дату, надаючи будь-яке значення, яке не є датою, мало сенсу. Ми використовуємо значення, визначене вручну, або значення за замовчуванням. Значення за замовчуванням залежить від типу параметра.

Тип параметра можна визначити з OAS, які в даний час визначаються як *string, number, integer, boolean, array* та *object.* Для кожного з цих типів можна налаштувати стандартне значення. String може за замовчуванням повертати «str», а int може повертати «0». Для деяких параметрів такого значення за замовчуванням може бути недостатньо, і в цьому випадку потрібно вручну визначити стандартне значення. Це стосується таких параметрів, як згаданий раніше приклад дати, номери карток та імена рахунків.

3.1.2.4 Виявлення помилки запиту

Чи був запит успішним чи ні, визначається головним чином кодом стану HTTP, який повертається як відповідь на запит. Як правило, 2xx вважається успіхом, а 4xx і 5xx - невдачею. Конкретні коди та відповіді залежать від предметного API. API може відповісти «200: OK», надаючи порожнє тіло відповіді, залежно від того, чи є це нормальною відповіддю, такі відповіді також можуть бути додані до визначення помилки. Тобто визначення помилки, можливо, доведеться скоригувати для конкретного домену. Запити можуть також провалитися через проблеми із підключенням.

3.1.2.5 Обмеження запитів

Веб-служби можуть обмежувати кількість запитів, що надходять протягом певного періоду часу. Обмеження запитів не розглядається у нашому процесі перевірки, оскільки ми маємо доступ до локальної збірки API, яка може функціонувати так само, як розгорнутий загальнодоступний веб-API. Можна робити скільки завгодно запитів, проте лише обмежені часом, який займає локальна збірка для обробки запиту. Якщо локальна побудова неможлива, тоді альтернативою є доступ до облікового запису веб-служби без такого обмеження швидкості.

3.1.2.6 Обчислювальні витрати

Загальний час, необхідний для перевірки обмежень, залежить від ряду факторів, найголовніше від кількості кандидатів, кількості параметрів для кандидата та кількості запитів, які можна зробити кожну секунду. Для кожного кандидата потрібно перевірити всі комбінації параметрів. Кількість комбінацій, припускаючи, що кожен параметр присутній або відсутній, дорівнює 2^p, де p - кількість параметрів. Можна робити приблизно 5 запитів в секунду. Час виконання, необхідний у секундах, дорівнює $\Sigma_{i=1}^{C} 2^{|P_{C_i}|}/5$, де C позначає набір кандидатів, а $|P_{C_i}|$ - кількість параметрів у кандидата i.

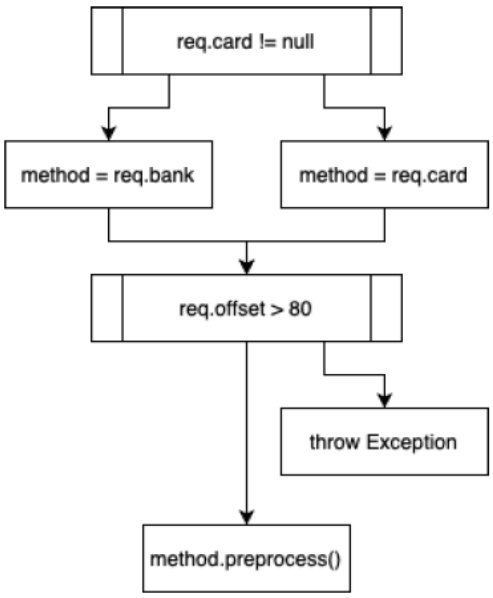


Рисунок 3.4 – Графік потоку управління, що відповідає методу, показаному в додаткку А.1.

Даний параметр може мати кілька значень, це обчислення змінюється. Кількість комбінацій визначається як $\Pi_{i=1}^{p} |values(p_i)| + 1$. Дотримуючись таблиці 3.3, знаючи, що gas може мати 2 значення, а temperature - лише 1, робимо 6 запитів. Заміна цього терміна замість $|P_{C_i}|^2$ дасть нам очікуваний час роботи кандидатів, які враховують конкретні значення.

Надаючи цю інформацію, можна побачити, що кількість параметрів у кандидатів буде домінувати в середовищі виконання досить швидко. Однак на

практиці кандидати, як правило, мають невелику кількість параметрів, тому контроль за кількістю кандидатів, як правило, є пріоритетом.

3.2 Аналіз коду

За допомогою аналізу коду ставиться задача витягти обмеження з контрольної структури вихідного коду. Для цього аналізуються методи, що мають відношення до обробки запитів HTTP, поданих до API. Метод, який називається «метод контролера», як правило, відповідає за обробку запитів, зроблених до однієї кінцевої точки API. Починаючи з цього методу контролера, виявляється доступ до параметрів та аналізуються будь-які структури управління та використовувані параметри викликів методів. У рамках прикладу веб-API в основному використовують мову програмування Java. Як такі, керуючі структури в основному включають оператори *if-else*, оператори *switch* і цикли *for*.

```
def handle(Request req)
    if req.getCard() != null then
        method = req.getCard()
        validateCard(method)
    else
        method =
          req.getBankAccount()
    end
    if req.getOffset() > 80 then
        throw Exception()
    end
    ⋮
    method.preprocess()
```

Рисунок 3.5 – Приклад методу обробки запиту API

Слідуючи фрагменту коду на рис. 3.5, ми можемо побачити, як ми могли зробити висновок про залежність «*card*» від «*bankAccount*» та обмеження на значення «*offset*». Тобто, якщо на картці не вказані платіжні реквізити, тоді нам буде потрібен рахунок у запиті. Що стосується зміщення, ми знаємо, що воно повинно бути меншим або рівним 80. На практиці існує велика кількість викликів,

пов’язаних із виведенням таких залежностей, але на цьому прикладі ми встановлюємо базову інтуїцію.

Щоб представити структуру управління методом, ми використовуємо граф управління потоком (ГУП). Ми генеруємо такі ГУП для кожного методу, який ми аналізуємо. ГУП показує, які гілки можна взяти, і як такий він може бути використаний, щоб знати, які параметри використовуються в цих гілках і які передумови застосовуються до цих гілок. Оскільки ГУП повідомляє нам, які гілки ведуть до недійсних станів, наприклад, створюючи винятки, ми можемо зробити висновок про те, які передумови можуть спричинити недійсність запиту. Для прикладу метода на рис. 3.4 створюється ГУП, показаний на рис. 3.5.

3.2.1 Граф потоку управління

Ми збираємо обмеження, переглядаючи заяви ГУП. У цьому процесі ми збираємо дерево передумов та наслідків для представлення обмежень. Як приклад такого дерева, слідуючи фрагменту 1, ми отримаємо обмеження, показані на рис. 3.5. Точні передумови, що застосовуються, можна зробити з ГУП. Наприклад, з ГУП на рис. 3.5, передумова *req.card != null* має *true* та *false* гілки. З цього можна зробити висновок, що заперечення цієї передумови призвело б нас твердження *method = req.card*. У дереві обмежень це представлено як *card = null → {present(bankAccount)}*.

```
card != null →{present(card) },
card = null →{present(bankAccount) },
offset > 80→{InvalidState }
```

Рисунок 3.6 – Приклад дерева обмежень, на основі фрагменту 1

Циклічні вирази, такі як цикл for, можуть бути важкими для статичного аналізу. Це пояснюється тим, що умова розриву циклу може бути складною. Однак ми помітили, що точний аналіз циклічних виразів не був важливим для висновку про обмеження. Циклічні оператори іноді використовувались для параметрів, що

мають значення масиву. Наприклад "*people*": [{"*name*": "*Frank*", …}; …]. Для таких значень масиву будь-які умови в тілі циклу застосовуватимуться до всіх значень, які будуть повторені. Отже, аналізу тіла циклу for один раз було б достатньо.

3.2.2 Чутливість

Ми проводимо аналіз, який є чутливим до потоку, частково чутливим до шляху та чутливим до контексту. При аналізі розглядаються гілки графа контролю потока (ГУП) без явного врахування попередньо оціненого шляху. Це робить аналіз лише частково чутливим до шляху. Для прикладу цього розглянемо вузол C, до якого можна отримати A або B. Під час оцінки C програма визначає, чи пройшов би шлях виконання A або B. Якщо змінна змінена у двох ексклюзивних гілках, то остання модифікація обрано. Подібним чином, ми також не відстежуємо умови даних, які можуть виникнути в результаті того чи іншого шляху. Наприклад, якщо для гілки встановлено *offset > 80* як guard, тоді ми не припускаємо нічого про значення зміщення поза тілом гілки.

3.2.3 Міжпроцедурний аналіз

Метод контролера викликає інші методи, включаючи ці виклики в аналізі. Це важливо, щоб знайти всі обмеження. Будь-який метод, на який покладається метод контролера, може також здійснювати виклики інших методів. Щоб представити цю мережу викликів функцій, ми генеруємо статичний графік викликів із методом контролера як коренем. Починаючи з методу контролера, ми рекурсивно будуємо цей графік викликів до заданої глибини. Для нас було достатньо глибини 15.

Коли ми здійснюємо виклик функції в потоці керування, тіло викликаного методу обчислюється з урахуванням його поточного контексту. Після фрагмента 1 *validateCard(method)* є прикладом такого виклику, в якому *method* представляє його контекст. Після аналізу тіла методу дерево обмежень інтегрується з деревом обмежень методу контролера. Припустимо, що для виклику *validateCard(method)* ми отримаємо дерево *card.cvc = null → InvalidState*, тоді це буде додано до вихідного дерева, як показано на рис. 3.4, і отримаємо нове дерево, показано на рис.

3.7. Зверніть увагу, що отримане дерево додається не на випадковому рівні, а в тілі оператора, що передував виклику методу. Тобто в тілі *card != null*.

```
card != null →{
  present(card),
  cvc = null →{InvalidState }},
card = null →{present(bank) },
offset > 80→{InvalidState }
```

Рисунок 3.7 – Приклад дерева обмежень на основі фрагмента 1 із обмеженнями, отриманими в результаті аналізу інтегрованого методу *validateCard(method)*.

3.2.4 Аналіз guard

До цього часу були розглянуті лише прості умови як guard, однак на практиці вони можуть ускладнюватися. Правильна оцінка та синтаксичний аналіз їх має важливе значення для правильного виведення обмежень, оскільки вони утворюють передумови будь-яких обмежень, які ми знаходимо. Наприклад, для обмеження *card.cvc != null* передумову *card.country = "NL"* можна вилучити із твердження, такого як *if(country.equals("NL"))*.

Мови програмування, як правило, мають велику різноманітність у висловлюваннях, які можуть бути використані в guard або інших частинах коду. У наступних розділах буде видно, як мати справу з розв'язанням ряду цих виразів і як вони використовуються при синтаксичному аналізі guards до машиночитного формату, орієнтованого на обмеження.

3.2.4.1 Стек змінних

Знання того, які змінні відповідають якому параметру, є важливим для вилучення обмежень параметрів із коду. Коли іде звернення до змінної, потрібно знати, чи пов’язана вона з параметром, і як така, що стосується обмежень, які ми будемо вилучати. Для підтримки таких посилань ми підтримуємо змінний стек. Наш стек змінних відстежує відомі конкретні значення змінних та параметри, яким відповідає яка змінна. Наприклад, *int maxLength = Constant.maxLength* буде

перетворено на запис *maxLength*, показаний на рис. 3.6, а *String card = request.getCard()* - на відповідний запис. Як ми знаємо *request.getCard()* відповідає параметру картки. Для змінної *isVaildCard* умовою є результат виклику функції (див. Розділ 3.2.4).

```
maxLength: {
  value: 100,
  condition: null
},
card: {
  value: null,
  condition: param(card)
},
isValidCard: {
   value: null,
   condition: param(card.cvc) != null || param(card.iban) != null
}
```

Рисунок 3.8 – Приклад стеку змінних.

Для примітивів Java, включаючи *string*, ми оцінюємо основні операції, такі як додавання та віднімання. Наприклад ”*En*” + ”*_US*” визначено як “*en_US*”. Що стосується булевих значень, ми розв’язуємо бінарні операції лише в тому випадку, якщо можна сказати, що вони, безумовно, хибні чи правдиві. Наприклад для *A||B* де *A = true* ми знаємо, що вираз є істиною. Якщо такі вирази присвоюються змінній, ми відповідно оновлюємо стек змінних. Будь-який вираз, який ми не можемо вирішити, призводить до того, що значення дорівнює *null*.

Для колекцій, таких як масиви, ми відстежуємо вміст колекції, якщо вміст є примітивним чи значенням переліку. Враховуючи, що API часто використовують прості типи, ці базові колекції є найбільш значущими для виведення обмежень. Наприклад, якщо стек відстежує список країн у змінній *countries*, а змінна country відповідає параметру *country*, то пізніше ми могли б проаналізувати твердження *if*(*countries.contains*(*country*)) на значущу передумову обмеження.

3.2.4.2 Виклики функцій

Guards можуть залежати від результату виклику логічної функції, наприклад від функції, показаної у фрагменті 2. Наприклад, *if*(*isValidCard*(*card*)){…}. Для того, щоб зробити висновок, які обмеження застосовуються до результату «true» або «false», ми використовуємо адаптацію до підходу за замовчуванням для аналізу викликів функцій. У цьому адаптованому підході збираються всі умови, які призводять до того, що функція повертає значення "true". Для простоти ми вважаємо, що функції не повертають значення null. Дотримуючись методу в фрагменті 2, «true» містило б *or*(*card.cvc != null, card.iban != null*) у своєму наборі передумов. Тоді, якщо guard хибна, ми знаємо, що застосовується заперечення умов в true. На рис. 3.7 показано, як результати оцінки такої функції можуть зберігатися у стеку змінних. Це стосується таких випадків, як *Boolean validCard = isValidCard*(*card*).

```
Function isValidCard(Card card)
    if card.getCvc() != null then
        return true
    end
    if card.getIban() != null then
        return true
    end
    return false
end
```

Рисунок 3.9 – метод, який перевіряє, чи є картка дійсною чи ні.

Для збору умов, в результаті яких функція повертає true, ми підтримуємо адаптовану версію дерева обмежень за замовчуванням, яке ми будуємо. Приклад цього можна побачити на рис. 3.8. У цій адаптованій версії оператори повернення додаються до дерева. Таким чином ми знаємо всі умови, які відповідають істині. Якщо оператор return повертає вираз, ми відповідаємо цьому виразу істиною. Наприклад якби метод у фрагменті 2 містив просто *return card.getCvc() != null || card.getIban() != null*, тоді ці умови відповідали б істині.

3.2.4.3 Поширені вирази

Основна мова Java включає загальнодоступні методи, логіку яких важко зробити з використанням статичного аналізу, але все одно їм можна надати значення окремо через їх загальну природу. У цьому випадку ми не використовуємо процес синтаксичного аналізу за замовчуванням, а прив'язуємо вираз до заданого вручну машиночитаного виводу. Прикладами цього є метод *.length()* для рядків та метод *.contains(arg)* для колекцій. Ми маємо справу з *.length()*, щоб мати змогу виводити обмеження на довжину параметрів типу рядка, а операція *.equals(arg)* може бути проаналізована як просте обмеження рівності. Ми застосували ту саму концепцію для декількох типових методів, що використовуються в Platon.

3.2.4.4 Аналіз guard

Коли ми зустрічаємо в коді умовний вираз, наприклад оператор if, ми хочемо проаналізувати його guard таким чином, щоб він був виражений у машиночитному форматі. Ми аналізуємо guard як колекцію логічних І та АБО. Ці сполучні елементи відповідають машиночитному поданню, представленому в розділі 2.3. У процесі синтаксичного аналізу цих тверджень обчислюються вирази, що трапляються безпосередньо в guard, тобто будь-які згадані змінні отримуються зі стеку змінних, і будь-який вираз вирішується так, як описано раніше.

```
card.cvc != null → {return true },
card.iban != null → {return true }
```

Рисунок 3.10 – Приклад дерева обмежень, адаптованого для булевих функцій, на основі фрагмента 2.

Наприклад, guard !*isValidCard(card) & card.getIssuer() != merchBank*. Припустимо, що *isValidCard(card)* - це той самий метод, що описаний у фрагменті 2. Ми аналізуємо цей метод як виклик логічної функції, який забезпечує нам умови *and(card.cvc = null, card.iban = null)*. Тоді, оскільки *card.getIssuer()* повертає

емітента, ми знаємо, що вираз відповідає параметру емітента (див. Розділ 3.2.5). Із стеку змінних ми бачимо, що для merchBank встановлено значення "Платон". Склавши всю цю інформацію, цей guard буде проаналізований для *and(and(card.cvc = null, card.iban = null), issuer != “Platon”)*.

3.2.4.5 Нерозбірливі вирази

Частини умовних операторів, які неможливо проаналізувати як обмеження параметра, аналізуються як непроаналізовані, але все одно відображаються у поданні. Оскільки код (часто) пишеться для читання людьми, це дозволяє нам зберігати деяку інформацію, яка може бути в умові. Слідуємо за попереднім прикладом !*isValidCard(card) & card.getIssuer() != null*. Припустимо, що нам не вдалося вирішити посилання на *isValidCard(card)*. Guard був би проаналізований до *and(!Unparsed(isValidCard(card)), issuer != null)*.

3.2.4.6 Оцінка guard

Іноді ми можемо оцінити, чи гарантовано guard буде помилковою чи правдивою. Коли guard гарантовано хибний, тоді тіло, що відповідає цьому шляху, не оцінюється. Це виключає будь-які обмеження, які могли б бути в тілі. Подібним чином, якщо guard істинний, тоді будь-які підключені оператори *else* / *else-if* не обробляються.

Чи зможемо ми оцінити, чи гарантовано захист істинним чи хибним, залежить від самого умовного твердження. Якщо брати до уваги *if(A||B)* де *A = true*, то ми знаємо, що цей guard завжди відповідає істині. Для *if(A&B)*, ми знаємо, що цей guard завжди є хибністю, якщо вказати *A = false*. За допомогою цих двох ідіом ми оцінюємо будь-який більший складений вираз, що складається з логічних І та АБО.

Булеве значення виразу можна отримати різними способами. У більшості релевантних випадків ми знаємо значення, оскільки воно встановлене у конфігурації фреймворку. Це особливо актуально, оскільки метод контролера може бути використаний повторно за допомогою різних конфігурацій. Ця конфігурація

може виключати запуск певних частин коду і як такої виключати обмеження, наявні в цій частині коду.

3.2.5 Проблематика в реальних платіжних системах

У цьому розділі підкреслимо низку проблем, які необхідно розглянути, щоб змусити аналіз коду працювати на реальних системах. Для наступних завдань будуть запропоновані рішення, які можуть працювати не для всіх систем, але використовувались як робочі рішення для результатів цієї роботи.

3.2.5.1 Повторювані імена параметрів

В API з інкапсуляцією об’єктів одне і те ж ім’я параметра може

використовуватися кілька разів для різних параметрів. Прикладом цього є параметр "*reference*" у ряді кінцевих точок Platon. В результаті цього будь-яке посилання на "*reference*" може посилатися на кілька параметрів "*reference*".

Для аналізу документації будь-яке посилання на повторюване ім'я параметра вважається посиланням на найближчий параметр щодо параметра, в описі якого було знайдено ім'я. Міра відстані - це кількість кордонів між двома параметрами. Де всі параметри верхнього рівню пов’язані, а всі підпараметри пов’язані з батьками, утворюючи дерево. При однаковій відстані враховуються обидва параметри.

Для аналізу коду правильний параметр виводиться з контексту останніх змінних, до яких здійснювався доступ. Наприклад, якщо ми просто отримали доступ до параметра "*card*", то ми можемо зробити висновок, що параметр "*reference*", ймовірно, відповідає "*card.reference*", а не (наприклад) "*bank.reference*".

3.2.5.2 Запит на перетворення об’єкта

Зазвичай запит, переданий API, десеріалізується з вихідного формату (JSON, XML) до об'єктної моделі. Завдяки цьому перетворенню може бути втрачено прямий зв’язок між параметрами запиту та змінними / полями, доступ до яких здійснюється в коді. Пов’язання змінних у коді з параметрами запиту є важливим.

Рішення може відрізнятися залежно від системи. Хоча відстеження полів у процесі десеріалізації може бути неможливим, ми можемо зробити припущення щодо відповідності полів класу з іменами параметрів. Для фреймворку Spring MVC імена полів повинні відповідати ключам параметрів. В межах API Platon об’єктні моделі, що використовуються під час десеріалізації, мають поля, імена яких безпосередньо відповідають іменам параметрів у вихідному запиті. Це дозволяє зробити висновок, що *address.getCountry()* відповідає параметру *country*, оскільки цей метод повертає поле *country*.

У деяких випадках імена полів внутрішнього класу можуть відрізнятися від імен параметрів на запит споживача API. Це може бути пов’язано з різними причинами. Ми виявили, що використання анотацій JAXB та відображення декількох параметрів в одному і тому ж полі класу під час десеріалізації є актуальним. У анотаціях JAXB ім’я зовнішнього параметра може відрізнятися від імені внутрішнього поля класу. Оскільки ці анотації розміщені над полями класу, інструменти відображення можуть легко виявити, коли це робиться. Для відображення параметрів у різні поля, визначення такого відображення вручну пом'якшує проблему. Коли два параметри були зіставлені з одним і тим самим полем, ми пов'язали це поле з обома зовнішніми параметрами.

3.2.5.3 Доступ до параметрів

Для того, щоб виявити обмеження, ми визначаємо, чи має частина коду доступ до параметра, чи змінна представляє параметр. Доступ до параметра не завжди означає, що він потрібен, якщо він залежить, від використовуваного фреймворку. Наприклад, у Flask параметри доступні з картоподібної структури. У цьому випадку для *param = req[‘param’]* параметр, до якого здійснюється доступ, означає, що він повинен бути наданий. Коли фреймворк перетворює запит на внутрішні об'єктні моделі, це не так. У десеріалізації надані параметри використовуються для заповнення відповідних екземплярів моделі. Пізніше ці екземпляри використовуються в потоці методу контролера.

Коли запити десеріалізовані як такі, доступ атрибута об’єкта, що відповідає параметру, не означає, що параметр необхідний. Ми визначаємо, чи потрібен параметр, на основі тверджень. Наприклад, дивлячись на фрагмент 3, зрозуміло, що для досягнення *process(address) card* не може бути null, і тому *card* потрібен у цій гілці.

3.2.6 Перевірка дерева обмежень

Статичний аналіз коду іноді може призвести до помилкових спрацьовувань, тому наявність етапу перевірки для підвищення точності має сенс. Однак використання нерозділених операторів робить автоматичну перевірку обмежень неефективною.

```
...
card = request.getCard()
address = request.getAddress()
if card != null then
  |  process(address)
end
...
```

Рисунок 3.11 – Приклад фрагмента коду, що показує, як доступ до атрибута об’єктів, що відповідає параметру, прямо не означає, що цей параметр необхідний.

Якщо якесь обмеження містить нерозділені елементи, стає неможливим безпечно генерувати запити перевірки; якщо запит на перевірку не вдається, ми не знаємо, чи це було пов’язано з обмеженнями, описаними нерозбірливим твердженням. Ми все ще могли б застосувати процес перевірки на обмеження, які не мають нерозбитих елементів. Однак більшість помилкових спрацьовувань містили нерозділені твердження. Крім того, помилкові спрацьовування мали, як правило, велику кількість важких для інтерпретації нерозділених елементів. Як результат, розмежування між істинними та помилково позитивними результатами, як правило, є відносно простим для людей.

4 МЕТОДОЛОГІЯ ДОСЛІДЖЕННЯ

Основним дослідницьким запитанням у цій роботі є «Чи можна (багато) параметричні обмеження у складних веб-API ідентифікувати за допомогою автоматизованих методів?». Щоб це визначити, ми розділили це запитання на такі дослідницькі питання (ДП):

ДП1: Наскільки ефективними є аналіз документації та статичного коду в виявленні обмежень параметрів у великомасштабному корпоративному API? Відповідь можна знайти, порівнюючи автоматично ідентифіковані обмеження з вручну зібраною основною істиною як для одно-, так і для багатопараметричних обмежень. Нас особливо цікавить, скільки обмежень можна ідентифікувати та кількість помилкових спрацьовувань. Помилкові спрацьовування особливо важливі, оскільки вони можуть перешкоджати процесу прийняття інструментів статичного аналізу [13].

ДП2: Які проблеми виникають при використанні аналізу документації або статичного коду для виявлення обмежень між параметрами? Окрім вищезазначених показників, ми прагнемо зрозуміти проблеми, що виникають при спробі автоматичного виявлення обмежень для широкомасштабних галузевих API. Ми робимо це, зокрема, для глибшого розуміння життєздатності та майбутніх напрямків кожного підходу; чи може підхід спрацювати, і що потрібно, щоб підхід спрацював? Унікальним для нашої роботи є аналіз складних API, які стикаються з унікальними проблемами з точки зору кількості параметрів для аналізу та структури супровідного коду.

4.1 Вибрані кінцеві точки

В процесі виконання роботи було розглянуто репрезентативний набір API Platon та кінцеві точки, які були загальнодоступними. Ці API та відповідні кінцеві точки можна дослідити за допомогою провідника API1. На момент написання роботи існує три різні загальнодоступні API: Checkout (*checkout*), Payments (*pal*),

and Platon for Platforms (*cal*). У Провіднику API Platon ці різні API можна розрізнити за URL-адресою сервера.

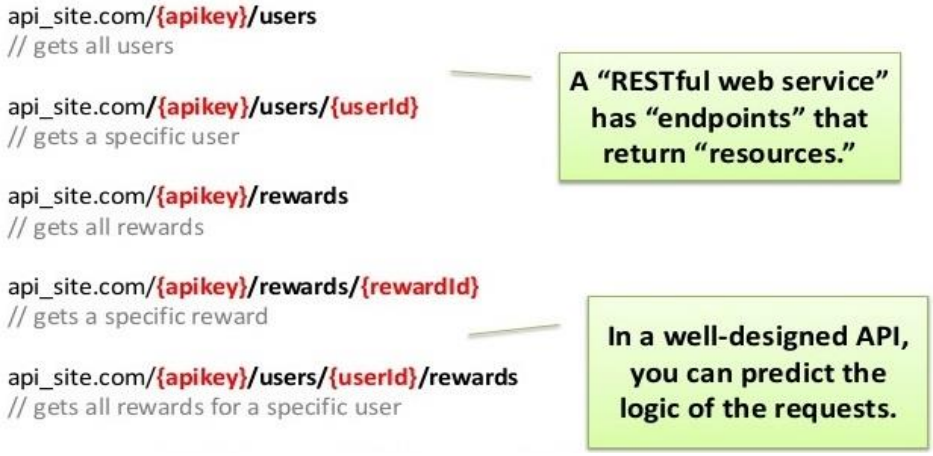


Рисунок 4.1 – Приклад кінцевих точок в API.

Ми вибрали кінцеві точки на основі наступних критеріїв; кінцева точка повинна містити обмеження між параметрами, а внутрішня логіка повинна бути досить несхожою на будь-яку раніше вибрану кінцеву точку. Цей критерій несхожості випливає із спостереження, що кінцеві точки часто мають однакові (багато) параметричні обмеження в результаті сильного повторного використання коду. Включення таких подібних кінцевих точок призвело б до незбалансованого набору кінцевих точок, в якому ми б ефективно аналізували один і той же код кілька разів.

Були вибрані наступні API та кінцеві точки; Checkout: */payments*, *Payments: /authorise*, */capture*, */storeDetailAndSubmitThirdParty*, */getCostEstimate*, Platon for Platforms: */createAccountHolder*, */getAccountHolder*, */updateAccountHolder*, */createAccount*, */uploadDocument*. Усі ці кінцеві точки можна знайти за допомогою провідника API.

4.2 Репрезентативний набір обмежень

Репрезентативний набір обмежень було зібрано вручну для кожної з обраних кінцевих точок. Ці обмеження включають як багато-, так і однопараметричні обмеження. Особливо важливі три етапи цієї роботи: збір, вибірка та представлення.

4.2.1 Збір

З огляду на те, що заздалегідь була відома лише низка обмежень, довелося ретельно перевірити код усіх вибраних кінцевих точок на наявність обмежень. У цьому процесі ми починаємо з методу контролера і слідуємо коду до кінця, беручи до уваги будь-які обмеження, які ми знаходимо на своєму шляху. Будь-які обмеження були перевірені шляхом створення запитів API, що відповідають обмеженню, щоб забезпечити їх коректність. Крім того, розробників відповідних API було попрошено вказівки щодо вказівки на відомі їм обмеження та загальну логіку обробки запитів, пов’язаних із цим API.

4.2.2 Вибірка

Деякі обмеження між параметрами є тривіальними, тому не кожне

обмеження, яке є технічним обмеженням, включається. Наприклад, з огляду на те, що ми маємо об’єкт *address* з посеред іншого, полем *country*, яке, як відомо, є обов’язковим. У цьому випадку *address* → *country* технічно є обмеженням між параметрами.

Однак будь-яке з таких обмежень інкапсуляції виключається на основі їх частого виникнення та тривіальності через те, що вони заздалегідь відомі в документації та напряму присутні у коді.

Не всі способи оплати включені, але замість цього використовується один репрезентативний спосіб оплати для кожної категорії методів оплати, наприклад відкритий інвойс або банківський переказ. Це робиться головним чином через

величезну кількість способів оплати, які всередині можуть покладатися на подібну логіку.

4.2.3 Представлення

Те, як представлені обмеження, може сильно вплинути на статистику результатів. Наприклад, якщо ми вирішимо представляти A → B & C як A → B і A → C, то в підсумку ми отримаємо подвійні обмеження. Те саме стосується A || B → C. Як правило, ми групуємо логічні АБО та логічні І разом, для того, щоб відповідати тому, як обмеження повинні бути присутніми в операторах IF; численні умови в guard призвели б до ряду наслідків. IF-оператори є особливо поширеною структурою управління для кодування обмежень.

4.3 Групування проблем

Щоб отримати уявлення про проблеми, з якими стикаються два підходи, ми поміщаємо невизначені обмеження між параметрами в одну або кілька категорій викликів відповідно до причин, які призводять до того, що обмеження не виявляються. Ці категорії викликів можна побачити в додатку А, для аналізу документації та коду відповідно.

4.4 Аналіз кінцевих точок

Ми аналізуємо всі вибрані кінцеві точки для обмежень окремо для документації та підходу на основі коду. Перш ніж запускати аналіз документації, ми розгортаємо локальну збірку API та налаштовуємо правильні дані аутентифікації для того, щоб робити запити до цього API. Після цього ми вказуємо кінцеву точку, яку хочемо проаналізувати. Це гарантує, що ми завантажуємо правильну службову інформацію з файлів OAS.

4.4.1 Порівняння результатів, отриманих підходами

Ми вручну порівняли результати, отримані різними підходами. Якщо виявлене обмеження та обмеження істинної логіки логічно еквівалентні, то ми вважаємо їх однаковим обмеженням. Враховуючи, що обидва підходи представляють вихід обмежень з використанням логічних формулювань, це порівняння можна здійснити безпосередньо.

Іноді ідентифікували лише частину обмеження. Наприклад, якщо взяти A → B & C, він буде ідентифікувати лише A → C. У цих випадках ми відхиляємось від стандарту представлення, встановленого в Розділі 4.2.3, і представляємо обмеження A → B як невстановлене та A → C як визначене.

4.4.2 Хибні результати

Обидва підходи можуть призвести до помилкових результатів. Для аналізу документації ймовірність отримати хибний результат зменшується через додану стадію перевірки. Для аналізу документації будь-яка таблиця істинності, яка пропонувала обмеження між двома параметрами, яких там не було, буде розглядатися як хибний результат.

```
card != null → {
   card.cvc = null → {InvalidState },
   card.iban = null → {InvalidState }
}
```

Рисунок 4.1 – Приклад дерева обмежень, що містить два помилкових спрацьовування.

Для аналізу коду будь-який унікальний шлях у обмеженні три, який неправильно припускає, що його передумови ведуть до недійсного стану, вважається помилково позитивним. Наприклад, дерево обмежень на рис. 4.1 містить два помилкових спрацьовування:

card != null → card.cvc != null та *card != null → card.iban != null.*

5 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

5.1 Дослідницьке питання 1

Наскільки ефективним є аналіз документації та статичного коду для виявлення обмежень параметрів у великомасштабному корпоративному API?

5.1.1 Багатопараметричні обмеження

Щоб відповісти на ДП1 щодо обмежень між параметрами, ми покажемо кількість обмежень між параметрами, визначених кожним підходом, у табл. 5.1.

Табл. 5.1 – Для кожної кінцевої точки розглядається: загальна кількість ідентифікованих вручну обмежень між параметрами, кількість обмежень, виявлених кодом та аналізом документації, з відповідними помилковими спрацьовуваннями (FP) та кількість обмежень, визначених обома підходами.

	Total	Code	Code FP	Doc	Doc FP	Both
/payments	17	11	2	0	0	0
/authorise	15	11	4	3	0	2
/capture	5	2	0	1	0	0
/storeDetailAndSubmit...	5	2	0	1	0	1
/createAccountHolder	4	0	0	3	0	0
/getAccountHolder	1	0	0	0	0	0
/updateAccountHolder	1	1	0	0	0	0
/createAccount	1	0	1	1	0	0
/uploadDocument	3	1	1	1	0	1
/getCostEstimate	1	0	0	1	0	0
Total:	53	28	8	11	0	4

Спостереження 1: Аналіз коду та документації разом виявляє 66% обмежень між параметрами. Аналіз коду та документації виявив 35 ((28 + 11) -4) = 53 обмежень загалом. Як правило, використовуючи обидва підходи, аналіз може знайти принаймні деякі обмеження для кожної кінцевої точки.

Спостереження 2: Аналіз коду та документації виявляє різні обмеження. Між 28 та 11 виявленими обмеженнями, як за допомогою аналізу коду та документації, було знайдено лише 4. Для цього 3 різні кінцеві точки сприяли загальному перекриванню 4.

Спостереження 3: Аналіз коду виявляє більше обмежень, але з більшою кількістю помилкових спрацьовувань. Аналіз коду виявляє приблизно в 2,5 рази більше обмежень, ніж аналіз документації, однак він дає ряд помилкових спрацьовувань. Аналіз документації виявляє менше обмежень, але не дає помилкових спрацьовувань.

5.1.2 Однопараметричні обмеження

Для однопараметричних обмежень (параметрів) ми надаємо інформацію для всіх кінцевих точок, які мали обмеження параметрів.

Табл. 5.2 – Загальна кількість та кількість ідентифікованих обмежень одного параметра для кожної кінцевої точки з використанням аналізу коду.

	Total	Identified
/payments	9	8
/authorise	14	10
/capture	5	5
/storeDetailAndSubmit...	4	4
/createAccountHolder	4	1
/createAccount	1	1
Total:	37	29

Сюди входить лише аналіз коду, оскільки аналіз документації не налаштований на пошук обмежень з одним параметром. Хоча це можливо, але ми не отримали помилкових спрацьовувань.

Спостереження 4: Для більшості обмежень з одним параметром використовуються зручні для виведення структури коду для перевірки значень параметрів. Наприклад, *fraudOffset*, який повинен бути меншим за 999,

буде здійснено за допомогою перевірки, подібної до *if (request.getFraudOffset() < 999).* Деякі помітні складні випадки включають регулярні вирази, які не аналізуються на щось значуще, і синтаксичний аналіз дат.

Спостереження 5: Аналіз коду виявляє 78% обмежень з одним параметром. Аналіз коду виявляє 29 з 37 обмежень на один параметр. Для деяких кінцевих точок йому вдається знайти всі обмеження параметрів. Відсоток знайдених обмежень для одного параметра перевищує загальну кількість обмежень між параметрами.

Таким чином, аналіз коду та документації разом отримує 66% обмежень між параметрами. Аналіз коду отримує 78% однопараметричних обмежень.

5.2 Дослідницьке питання 2

Які є проблеми у використанні аналізу документації або статичного коду для виявлення обмежень між параметрами?

5.2.1 Аналіз документації

Для кожної кінцевої точки ми позначили обмеження з причинами невизначення. Ці чотири причини описані в таблиці A.1, а їх короткі описи містяться в заголовку табл. 5.3. Щоб відповісти на ДП2, ми перевіряємо та обговорюємо ці причини більш детально.

5.2.2 Брак інформації

Безумовно, найпоширенішою причиною невизначення обмеження є відсутність інформації про це обмеження. Без описів, що описують обмеження безпосередньо, важче ідентифікувати їх лише за допомогою описів. Перевірка описів параметрів на пряме посилання на параметри не буде працювати. Використання контекстної пов’язаності описів параметрів потенційно може бути використано як інформація. Однак первинні результати, як обговорювалось у розділі 3.1.1, дійсно свідчать про те, що це навряд чи можливо.

У деяких випадках ми навіть не знаємо, що параметр присутній у документації. Існування таких параметрів можна виявити, переглядаючи об'єктні моделі коду, як обговорюється далі в розділі 5.2.2. Однак для аналізу документації це не варіант, а це означає, що висновок про будь-яке з таких обмежень неможливий, якщо не надана додаткова інформація вручну. Сюди входить надання опису параметра, якого немає в OAS. Надання описів вручну з єдиною метою зробити висновок про обмеження є заплутаним. У цьому випадку безпосереднє зазначення обмежень, які можуть застосовуватися до цього параметра, може бути більш ефективним.

Табл. – 5.3: Для кожної кінцевої точки таблиця показує причини, за якими аналіз документації не зміг визначити обмеження між параметрами. Одне обмеження, можливо, не було виявлено з багатьох причин.

	A1 No Info	A2 Implicit Info	A3 No Value	A4 Validation
/payments	15	2	0	0
/authorise	7	2	3	2
/capture	4	0	0	0
/storeDetailAndSubmit...	3	1	0	0
/createAccountHolder	0	1	0	0
/getAccountHolder	0	0	0	0
/updateAcountHolder...	0	1	0	0
/createAccount	0	0	0	0
/uploadDocument	2	0	0	0
/getCostEstimate	0	0	0	0
Total:	30	7	3	2

5.2.3 Неявні посилання

Деякі обмеження не були виявлені через використання неявної інформації. Був ряд випадків, коли OAS включав документацію про обмеження, але в описі не було прямо вказано назву параметра.

Для того, щоб встановити зв'язок між неявною інформацією та параметром, на який посилається, потенційно можуть використовуватися вбудовування слів.

Тому замість перевірки прямого збігу для вимірювання схожості інформації можна використовувати заходи схожості. Однак це може породити занадто багато кандидатів для валідації, подібно до вкладених слів, описаних у Розділі 3.1.1.

5.2.4 Значення не виявлено

Знайти обмеження значень, від яких залежать параметри, може бути важко. Бувають випадки, коли, як правило, серед інших причин, обмеження, залежне від значення, не виявляється, оскільки значення не виявляється або зовсім невідоме. Наприклад, для обмеження *recurring.contract* = "*ONECLICK*" → *card.cvc* значення *ONECLICK* не виявлено. А для обмеження *country* = "*Ukraine*" → *stateOrProvince* значення “Ukraine” не виявлено. Надання цих значень як значень перерахування OAS дозволить виявити ці значення.

Це два різні випадки невизначених значень. Повторювані типи контрактів обмежені за кількістю та специфічні для компанії, коди країн є загальним стандартом (ISO-3166-1 alpha-2), для якого існує більша кількість значень. Тому їх включення до OAS мало б меншого сенсу і не забезпечило б вирішення проблеми. Враховуючи, що значення не завжди присутні в описах за допомогою спеціального форматування, аналіз описів для вилучення значень також загалом неможливий.

Залежні від вагомості обмеження залишаються відкритим викликом для будь-якої моделі чорних ящиків, включаючи аналіз документації. Документація може надати нам деякі значення, такі як значення переліку в OAS. Однак це не завжди так. Хоча в деяких випадках зазначення можливих значень може бути можливим, це є менш життєздатним у випадку *value+additionalValue<100000*, оскільки ми, ймовірно, не знаємо значення або маємо будь-яку попередню інформацію про нього.

5.2.5 Неспостережувані обмеження

Запити API, що використовуються для перевірки, можуть не вдатися через неспостережувані обмеження. Може бути виявлена лише частина обмеження, що призводить до того, що ця виявлена частина не ідентифікується як обмеження.

Наприклад, припустимо, що ми виявляємо зв'язок між *contract* ="*ONECLICK*" та *card.cvc*. Справжнім обмеженням є *contract* = "*ONECLICK*" → *card.cvc* & *shopperInteraction*. У цьому випадку ми не можемо автоматично зробити висновок про обмеження між *contract* = "*ONECLICK*" та *card.cvc*. Це пов’язано з тим, що невиконана вимога *shopperInteraction* призведе до помилки деяких запитів перевірки. Будь-який запит, включаючи контракт, буде невдалим, оскільки він також повинен мати *shopperInteraction*.

Хоча ми не можемо виявити неспостережуване обмеження, ми можемо отримати деяку інформацію з таких випадків. У разі виникнення цього сценарію запит завжди буде невдалим, якщо присутній параметр із неспостереженим обмеженням. Неспостережене обмеження може бути обмеженням між параметрами, як було описано раніше, або обмеженням на значення параметра. Результат однаковий для обох випадків; ми не знаємо причини несправності. У цьому випадку ми повинні підняти позначку щодо цього, щоб причина могла бути вирішена вручну.

5.2.6 Запит залежностей

Запити API можуть залежати від послідовності попередніх запитів. Для Платона класичним прикладом є кроки авторизації та захоплення, на яких крок захоплення фіксує платіж, санкціонований на етапі авторизації. Як результат, авторизація повинна бути використана перед захопленням, і запит захоплення залежить від поля, що повертається запитом авторизації, а саме посилання на PSP. Без надання дійсного посилання на PSP запит захоплення завжди буде невдалим, що ускладнює висновок обмежень.

Під час генерації запитів на перевірку такі обмеження між різними кінцевими точками можуть бути проблемою. Особливо це стосується випадків, коли значення, повернене попереднім запитом, може використовуватися лише в залежному другому запиті один раз. Наприклад, це може статися для створення облікових записів. Один запит міг створити обліковий запис, а другий - закрити його. Повторна спроба закрити вже закритий рахунок призведе до помилки. У

цьому випадку попередній запит повинен бути надісланий для кожного запиту перевірки, щоб отримати дійсне значення.

Табл. 5.4 – Для кожної кінцевої точки таблиця показує причини, за якими аналіз коду не зміг визначити обмеження між параметрами. Одне обмеження, можливо, не було виявлено з багатьох причин.

	B1: Detection	B2: Dereferenced	B3: Variable Stack	B4: Preconditions	B5: Control Structure	B6: Data Flow	B7: Arithmetic	B8: Framework
/payments	1	1	0	1	1	1	0	2
/authorise	0	3	2	1	3	0	1	0
/capture	0	3	2	0	3	0	1	0
/storeDetailAndSubmit...	0	0	0	0	0	3	0	3
/createAccountHolder	0	2	2	0	4	2	0	0
/getAccountHolder	0	0	0	0	0	1	0	0
/updateAccountHolder...	0	0	0	0	0	0	0	0
/createAccount	0	1	0	0	0	0	0	0
/uploadDocument	0	2	0	0	0	0	0	0
/getCostEstimate	0	1	1	0	0	0	0	0
Total:	1	13	7	2	11	7	2	5

Щодо документації, безумовно, найпоширенішою причиною не виявлення обмежень є відсутність інформації, яка явно описує обмеження в Специфікаціях OpenAPI. Робота з цією відсутністю інформації є головною проблемою.

5.3 Статичний аналіз коду

Для аналізу коду ми визначили ряд проблем, з якими стикається наш статичний аналіз коду при витяганні обмежень між параметрами. Деякі з цих викликів були вирішені таким чином, що вони більше не впливають на результати, візьмемо, наприклад, виклик «Шаблон дизайну». Ці виклики все ще включаються,

оскільки вони можуть бути актуальними для інших API. Для кращого розуміння проблем, що існують, пропонується фіктивний потік API разом із фрагментами коду, що містять ряд проблем, пояснених у цьому розділі.

Можна спостерігати, що, на відміну від аналізу документації, для аналізу коду відсутність інформації не була головною проблемою. Часто обмеження кодувались в межах потоку управління методів контролера відносно прямо. Це робить підхід статичного аналізу коду, який фокусується на аналізі тверджень контролю, ефективним. Однак є випадки, коли такий базовий підхід повинен бути скоригований або розширений. Ми докладніше обговоримо ці випадки у наступних розділах.

5.3.1 Параметр не виявлено

Якщо вираз не пов'язаний з параметром, з цієї пов'язаної частини коду не буде виведено жодних обмежень. В рамках нашого підходу це сталося, оскільки на ці параметри не посилаються в специфікаціях OpenAPI. З точки зору бізнесу на ці параметри можна не посилатися, оскільки функція застаріла або тому, що певна функціональність призначена лише для використання вибраною групою споживачів API. З точки зору проектування API ці параметри можуть бути відсутніми, оскільки документацію все ще потрібно додати.

Як вирішення цієї проблеми можна вказати будь-який доступ до полів моделі даних як доступ до параметра, незалежно від того, чи є поля, до яких здійснюється доступ, в документації чи ні. Для цього нам довелося б виявити, чи є клас частиною моделей даних. Це можна зробити, вказавши вручну моделі даних. З огляду на відносно невелику кількість класів моделі даних, це життєздатно. Вручну вказати імена параметрів, яких немає в OAS, є альтернативним рішенням. Однак для цього потрібно знати, яких параметрів немає в OAS. У нашому випадку це було не те, що ми завжди знали б заздалегідь.

5.3.2 Параметр без посилання

Велика кількість посилань на параметри в якийсь момент розмежовується. Коли виникає цей виклик, ми спочатку можемо пов’язати вираз із параметром, але після деяких кроків це посилання втрачається. Зазвичай посилання втрачається через те, що розроблений аналізатор не може проаналізувати всі вирази Java. У оцінюваних API це зазвичай траплялося під час створення об’єкта. Або створення об’єкта під час десеріалізації запиту до внутрішньої моделі даних, або при створенні нового об’єкта в потоці методу контролера. Останнє було більш поширеним, ніж перше.

У рамках десеріалізації два або більше параметри можуть використовуватися для заповнення одного і того ж поля моделі даних. Наприклад, для об'єкта картки код країни можна або встановити безпосередньо з параметра *countryCode*, або отримати з наданого IBAN. Як результат, ми не знаємо, що частина наданого IBAN обмежена кодом країни. Зберігання посилань на десериалізацію Platon становить значну проблему, не з простих рішень. Це пов’язано здебільшого з тим, що етапи десериалізації тісно переплітаються із використовуваною структурою (див. Розділ 5.2.2).

Часто ряд параметрів використовувався при створенні нового об'єкта моделі даних. Потім цей об’єкт перевірявся на наявність обмежень. Якщо поля в новій моделі не відповідали відомим іменам параметрів, обмеження виявити не вдалося. Таким чином, статичне обслуговування об'єктів є важливим для статичного аналізу коду. Для створення об’єктів у потоці методу контролера можна зберегти деякі посилання, якщо ми будемо відстежувати об’єкти у нашому стеку статичних змінних. При синтаксичному аналізі конструкції об’єкта ми теоретично можемо відстежувати, які поля цього об’єкта відповідають яким параметрам. Через обмеження за часом ми не досліджували це більш детально.

5.3.3 Статичний змінний стек

Зберігання базових значень у стеку змінних достатньо для виведення більшості обмежень між параметрами. Ці основні значення включають примітиви Java, рядки та посилання на параметри, як описано в розділі 3.2.4. Цього, як правило, достатньо, оскільки поля в запитах API зазвичай також обробляють основні значення. Наприклад, "*name*" буде рядком, а "*amount*" буде цілим числом. Отже, параметри, які є об'єктами, існують із комбінацій цих базових значень. Наприклад "*person*" *: {*"*name*" *:* "*Hanson*"*; ... }*. Тільки підтримка базових значень не завжди достатня. Це особливо актуально для об'єктів, як обговорюється в розділі 5.2.2 B.2. Розширення діапазону виразів, які можна проаналізувати для оновлення стеку змінних, коротко обговорюється в розділі 5.2.

5.3.4 Передумови

Деякі обмеження мають складні передумови, внаслідок яких потрібен інший параметр чи ні. Наприклад, розглянемо гіпотетичну функцію *isValidIban (iban)*, в якій дійсність самого IBAN залежить від великої кількості умов, які підхід має на меті проаналізувати. Зазвичай такі передумови мають вирази, які важко аналізувати статичному аналізу коду. Як результат, наш підхід створить передумови, що містять часто велику кількість нерозбірливих елементів. Вони важко читаються людьми, навіть якщо передумова може бути повністю розібрана. Ось чому у цього є своя категорія викликів, хоча суть проблеми стосується як B.2, так і B.3.

По суті, це завдання стосується виду виразів, який може аналізувати наш інструмент статичного аналізу. Враховуючи, що ми не можемо взяти до уваги всі вирази будь-якої мови [14], важливо визначити, які вирази важливі при статистичному аналізі коду для обмежень параметрів. Наприклад, метод Java за замовчуванням *Boolean.parseBoolean (String s)* може бути функцією, яка універсально використовується через різні API.

5.3.5 Структура управління

Базова оцінка циклів for має значення для вилучення обмежень. Цикли For використовуються в ряді випадків для перевірки обмежень параметрів у самому вихідному коді. Коли використовували цикли for, це зазвичай робилося для параметрів зі значеннями масиву. Наприклад, "*people*":*[{*"*name*": "*Frank*", *...}, ...]*. Як обговорювалось у розділі 3.2.1, оцінки тіла циклу for одного разу було б достатньо для висновку про обмеження. Цикли For також використовуються для арифметичних обмежень. Зокрема для типу P1 + P2 + ... = Pn. Для цього можна вказати, що всі елементи параметра типу масиву вносять свій внесок у суму.

Оператори комутаторів були загальними для обмежень, що залежать від значення. Природно, що оператори case добре піддаються виконанню логіки, специфічної для певного значення параметра.

5.3.6 Потік даних

Обмеження потоку даних включали відсутність чутливості до шляху через умови на певних гілках. Вплив на результати помітний, тому при використанні статичного аналізу коду для виведення обмежень між параметрами кращим є підхід, який чутливий до потоку даних.

```
Function getAccount(request)
    if request.auth == True then
        if request.ID != null then
            return ...
        else if request.user != null then
            return ...

        Throw Exception()
    end
end
```

Рисунок 5.1 – Метод оцінки деяких властивостей об’єкта.

Щодо умов на провальних шляхах, рис. 5.1 показує, як невідстеження цих умов може спричинити неправильне вилучення обмежень. У цьому випадку для того, щоб потрапити на виняток, застосовуватимуться невід'ємні умови *ID == null*

та *user == null*. Ми не відстежуємо ці умови. Таким чином, аналіз коду витягне неправильне обмеження *auth = True → InvalidState*.

5.3.7 Синтаксис арифметичного обмеження

Арифметичні обмеження включають параметри, які пов’язані між собою за допомогою арифметики. Наприклад, *A + B > 10*. Ці обмеження є загальноприйнятими [17], 107 з 633 виявлених обмежень є арифметичними обмеженнями. З цих 107 обмежень *A >= B* був найпоширенішим.

У рамках нашого тематичного дослідження такі обмеження часто безпосередньо були присутні в коді. Наприклад *A >= B* мав би відповідне твердження *if (A >= B)*. Це полегшило їх отримання. Інший тип арифметичних обмежень передбачає додавання параметрів, які повинні задовольняти деяким критеріям. Це сталося для *amount + additionalAmount < number* та для *sum(split.value) = total*. Подібно до вищезазначеного арифметичного обмеження, обмеження *amount+additionalAmount < number* часто було безпосередньо присутнє в коді як таке, і таким чином його легко витягти. Це не стосувалося обмежень, таких як *sum(split.value) = total*, кодування яких здійснювалося за допомогою циклічного відключення.

5.3.8 Шаблони дизайну

Шаблони проектування використовуються для вирішення типових проблем в галузі програмного забезпечення. Ці шаблони можуть бути будь-якими з класичних шаблонів або індивідуальними шаблонами дизайну. «Фабричний метод» був єдиним шаблоном дизайну, для якого був необхідний аналіз, щоб зробити висновок про обмеження параметрів.

Фабрична схема була особливо актуальною для способів оплати. Способи оплати мають свої вимоги; екземпляр способу оплати перевірить, чи задовольняє запит його вимоги. Визначити, який спосіб оплати відповідає якому екземпляру, важко зробити повністю автоматично за допомогою статичного аналізу коду. Цю проблему було вирішено, вказавши вручну, який спосіб оплати відповідає якому

класу Java, а який метод у цьому класі застосував обмеження. Це повністю обходить необхідність статичного аналізу коду для вирішення відповідного екземпляру способу оплати.

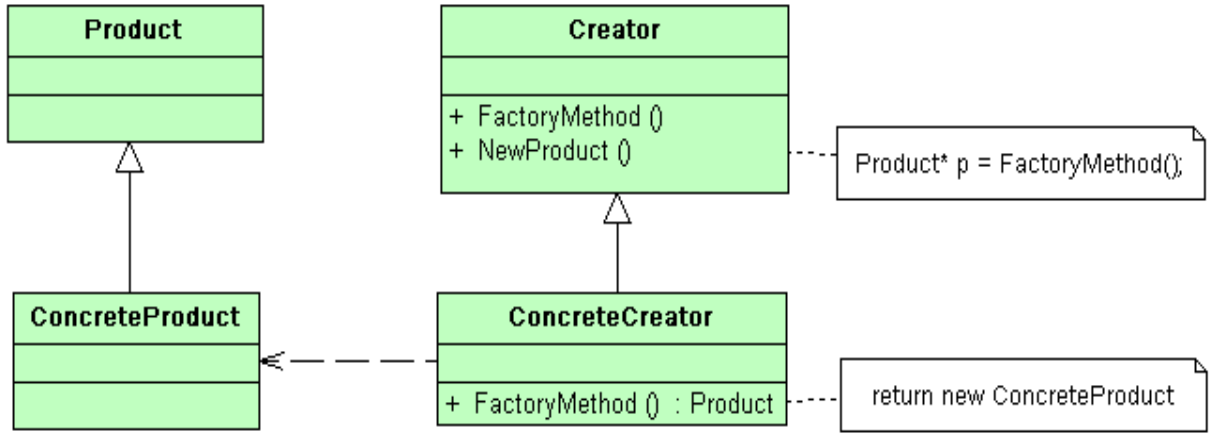


Рисунок 5.2 – Приклад шаблону проектування «Фабричний метод».

5.3.9 Некодові обмеження

Вихідний код - не єдине місце, де можуть бути обмеження. Як приклад цього, обмеження можуть застосовуватися на рівні бази даних. У цьому випадку максимальна довжина поля може бути обмежена специфікаціями таблиці, або запит SQL може залежати від наявності певного поля.

В ідеалі будь-яке обмеження обробляється безпосередньо в самому коді. Це забезпечує зворотній зв'язок у відповідях API, роблячи API більш прозорим.

Таким чином, для аналізу коду основними проблемами були нечутливість

потоку даних та розробка підходу до розумного аналізу коду. У тому числі рішення про те, як оцінювати циклічні структури управління, підтримуючи посилання на параметри у всьому коді API та вирішуючи, як поводитися з різноманітністю виразів мовою програмування.

6 ОБГОВОРЕННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

У цьому розділі ми порівнюємо свою роботу з існуючими, порівнянними підходами, обговорюємо використання формальних обмежень, можливі проблеми безпеки, те, як ми очікуємо узагальнення аналізу коду, та загрози дійсності.

6.1 Порівняння з попередніми роботами

Основні відмінності нашого підходу щодо підходів, що існують у літературі, полягають у наступному: наш підхід був зосереджений на API з великою кількістю параметрів, і на аналізі коду для виведення обмежень.

Існуючі роботи, які спеціально зосереджувались на обмеженнях між параметрами для веб-API, оцінювали лише API з невеликою кількістю параметрів. Це може зробити методи, засновані на пошуку, більш ефективними, що відображається в результатах. В своїх роботах Gao та ін. [11] та Wu та ін. [35] вдається досягти точності 90%, тоді як ми отримуємо точність 20%, використовуючи підхід до аналізу документації, багато в чому схожий на той, що застосовувався у роботі Wu та ін.

Щодо аналізу коду, Wu та ін. – це єдина робота, яку ми знайшли, яка описує аналіз коду для виведення обмежень між параметрами (див. розділ 2.4). Існує кілька відмінностей. Перш за все, їх підхід виконує аналіз потоку даних, тоді як ми безпосередньо витягуємо структуру обмежень. Наслідком цього є те, що наш підхід не має суворої необхідності у валідації кандидатів, подібно до того, що робить аналіз документації. Генерування комбінацій параметрів, до яких здійснюється доступ, так само, як в згаданих авторів, також призвело до вибуху можливих комбінацій. По-друге, їх підхід не стосується багатьох функцій, які має будь-яка програмна система. Ключовими прикладами є аналіз методів окремо, на відміну від даного контексту, та не підтримка обмежень, незалежних від значень.

Робота Pandita та ін. [22] не надає жодної конкретної інформації про обмеження між параметрами, а також не аналізує лише веб-API. З інформацією,

яку вони надали, пряме порівняння неможливе.

6.2 Аналіз коду для складних API

Різниця між статичним кодом для складних та простих API може бути не такою сильною, як можна подумати. Прості та складні API покладаються на один і той же набір виразів (Java) і часто розгортають декілька фреймворків в процесі. Таким чином, велика кількість проблем, описаних у розділі 5.2.2, стосуватиметься і простого API.

Складні API матимуть більше коду, що робить швидкий адекватний аналіз першочерговим завданням. Широкий аналіз кожного методу може бути занадто інтенсивно обчислювальним. Підхід, описаний у цій роботі, здатний аналізувати методи досить швидко для великих програмних систем. Однак, продовжуючи розвивати цей підхід, зокрема, щоб зробити його потоком даних чутливим, цей аспект потрібно враховувати.

6.3 Використання формальних обмежень

За умови достатньо повного набору формальних обмежень існує два основних варіанти використання: документація та перевірка на відповідність. Тут також ми не можемо автоматично визначити всі обмеження. Це звужує практичне використання обмежень, які можна виявити.

6.3.1 Документація

Зовнішня документація повинна допомогти кожному, хто використовує API, інтегруватися краще. З цією метою офіційні обмеження слід застосовувати таким чином, щоб допомогти споживачеві API. Через велику кількість обмежень, як правило, не вигідно постійно показувати всі обмеження. Тому наявність більш динамічної документації, яка використовує поступове розкриття інформації, щоб вказати на відповідні обмеження для користувача, є цікавим варіантом

використання. Наприклад вимоги до конкретного способу оплати відображатимуться лише тоді, коли споживач API вкаже, що бажає використовувати цей спосіб оплати.

6.3.2 Перевірка відповідності

Враховуючи офіційний набір обмежень у машиночитному форматі, ми можемо перевіряти відповідність запитів та API. Martin-Lopez [16] описує інструмент, який може зробити обидві ці речі. Можливість відправляти невідповідні запити до того, як вони потрапляють до контролера API, може заощадити час і ресурси. Засіб, описаний Martin-Lopez, також генерує велику кількість тестових кейсів на основі цих обмежень. Таким чином, API можна також контролювати на предмет змін в обмеженнях та можливих невідповідностей бізнес-вимогам, що відображаються через надані обмеження. Таке тестування веб-API, орієнтоване на обмеження, не є основним предметом у поточній літературі [1, 2, 9, 29].

6.4 Проблеми безпеки

Обмеження, витягнуті з вихідного коду, відображають код певною мірою. Таким чином, обмеження можуть розкрити небажану інформацію. Це актуально для можливих атак на API та розкриття конфіденційної інформації.

Обмеження надають інформацію про граничні умови, що використовуються в коді. Зловмисна третя сторона може потенційно використовувати цю інформацію для більш легкого використання API. Таким чином, автоматичне оприлюднення всіх виявлених обмежень не є кращим. Що стосується конфіденційної інформації, обмеження можуть виявити особливості, які ще перебувають у стадії розробки перед тим, як бути оголошеними громадськості.

Обмеження можуть також виявити параметри, які призначені лише для використання вибраною групою споживачів API.

6.5 Узагальнення

Даний підхід розроблений для конкретної системи, яка використовує певні правила кодування. Він використовує основу віддаленого виклику процедури (RPC), написану на Java. Протягом усієї розробки підходу ми мали на увазі будь-які припущення, які можуть обмежити підхід в інших сценаріях. У цьому розділі обговоримо, як можна узагальнити поточний підхід.

6.5.1 Мова програмування

Підхід до аналізу коду може працювати незалежно від ключової мови програмування API. Основними особливостями підходу є спостереження доступу до параметрів у коді та аналіз структур управління, в яких ці параметри використовуються. Більшість популярних об'єктно-орієнтованих мов містять подібні структури управління. Спостереження за доступом до параметрів буде відрізнятись від мови до мови та від фреймворку до фреймворку (див. Розділ 3.2.5), але якщо зв’язок між параметрами запиту та виразами в коді може бути встановлений, цей підхід буде працювати.

6.5.2 Фреймворки API

Існує велика кількість фреймворків API, у цьому розділі ми обговорюємо Jersey, Spring та Flask через їх популярність. Взагалі, фреймворки мають справу з відправленням запиту, надісланого в API, до відповідного методу (контролера). Фреймворки не містять жодної логіки щодо того, що робити із вмістом запиту, що надійшов до його методу контролера.

Фреймворки Jersey, Spring та Flask забезпечують подібну функціональність. Вони створюють класи контролерів за допомогою методів контролерів, позначених як такі з анотаціями. Наприклад *@GetMapping("=end point")* для методів у Spring та *@Path("end point")* для класів у Jersey. Вони також надають функціональність для вказівки, з якими методами HTTP вони повинні працювати.

Такі явні анотації роблять тривіальним відповідність правильного методу контролера кінцевій точці.

Для доступу до параметрів у відправлених запитах усі три фреймворки забезпечують узгодження. У тілах запитів Flask можна отримати доступ як карту, де параметри в запиті безпосередньо відповідають ключу, необхідному для доступу до них. Наприклад, *language = req_data["language"]* можна використовувати для доступу до параметра мови. Spring і Jersey також мають прямий зв'язок між параметрами, надісланими в запиті, і змінними в коді. В обох випадках це робиться шляхом анотування параметрів методу контролера. Ці анотації різні для кожного фреймворку. Для Jersey параметр може бути анотований таким чином *methodName(@FormParam("deliveryAddress") String deliveryAddress).*

На закінчення, ключові особливості, необхідні для статичного аналізу коду для багатопараметричних обмежень, є для трьох популярних фреймворків.

6.5.3 Стиль запиту

Стиль запиту - це спосіб передачі параметрів до API у запиті. Ми розглядаємо три основні напрямки: RPC, REST та GraphQL.

Фреймворки, що базуються на віддаленому виклику процедур (RPC), розглядають запити API як виклики функцій, де аргументи функції поміщаються або в рядок запиту, або в тіло. Це суперечить REST, який часто включає такі параметри шляху, як */store/orders/orderID.* Параметри шляху навряд чи будуть задіяні в обмеженнях. Дослідження показали, що близько 0,8% обмежень стосуються параметрів шляху [17], які всі були розташовані в одному API.
Параметри тіла та запиту використовуються часто. Єдиною відмінністю підходу є метод кодування параметрів для запиту.

GraphQL відхиляється щодо стилю запиту. Для (HTTP) операцій GET запити можна порівняти з SQL. Будь-які мутації, такі як DELETE та PATCH, використовують стиль, подібний до RPC. Для запитів GET запити фільтрують дані, а не виконують над ними операції. Таким чином, важко продумати будь-які

обмеження, які призводять до помилок, і обмеження між параметрами можуть не бути присутніми в таких запитах.

6.6 Реальні загрози

Внутрішня дійсність. Основна інформація, яку ми використовували, складається з репрезентативного набору обмежень, які ми збирали вручну для кожної з обраних кінцевих точок. У цьому процесі ми ретельно перевірили відповідний код і перевірили їх, подавши запити до API. Таким чином, можна бути впевненим, що маємо правильний набір обмежень. Для повноти потрібно було перевірити всі наявні кінцеві точки в Platon на наявність обмежень. З відповідних кінцевих точок ми вибрали такі обмеження, щоб вони представляли різноманітний вибір коду. Отже, результати відображають різні API в Platon.

Зовнішня дійсність. Враховуючи, що це дослідження є тематичним дослідженням, проведеним в одній системі, для подальшого узагальнення результатів необхідні дослідження інших складних API. Однак, враховуючи розмір та масштаби програмного забезпечення Platon, можна сказати, що результати, виявлені в даному дослідженні, є репрезентативними для інших систем. З розділу 6.5 витікає очікування, що підхід буде узагальненим для різних мов програмування, фреймворків та стилів запитів.

ВИСНОВКИ

У даній магістерській роботі виконані наступні завдання:

1. Було розроблено два підходи для виявлення обмежень параметрів для складних веб-API. Один підхід аналізує онлайн-документацію з метою висновку про обмеження між параметрами, інший залежить від аналізу статичного коду, для того щоб віднайти багато- та однопараметричні обмеження з потоку управління вихідним кодом API. Підходи на основі документації та вихідного коду здатні ідентифікувати 21% та 53% відсотків обмежень відповідно.

2. Було з'ясовано, що ці два підходи стикаються в основному з окремими проблемами. Підхід, заснований на документації, значною мірою страждає від браку явної інформації, що описує обмеження. Статичний аналіз коду, як правило, може витягувати обмеження з вихідного коду, підтримуючи стек базової змінної, оцінюючи виклики методів та аналізуючи умови в операторах *for, switch* та *if-else*.

3. Основні проблеми, з якими ми зіткнулись у цьому тематичному дослідженні, були нечутливість потоку даних та розробка підходу до аналізу статичного коду.

4. Визначена ефективність аналізу документації та статичного коду для виявлення обмежень параметрів у великому корпоративному API. Аналіз коду та документації в сукупності отримує 66% обмежень між параметрами. Аналіз коду отримує 78% обмежень з одним параметром.

5. Були виявлені наступні проблеми у використанні аналізу документації або статичного коду для виявлення обмежень між параметрами:

- Щодо документації, найпоширенішою причиною невиявлення обмежень є відсутність інформації, яка явно описує обмеження в Специфікаціях OpenAPI. Робота з цією відсутністю інформації є головною проблемою.

- Для аналізу коду основними проблемами були нечутливість потоку даних та розробка підходу до розумного аналізу коду. Це включає рішення про те, як оцінювати циклічні структури управління, підтримуючи посилання на

- параметри у всьому коді API та вирішуючи, як поводитися з різноманітністю виразів мовою програмування.

6. За умови великої кількості запитів, які можна зробити для локальної збірки API, підходи на основі пошуку наборів параметрів показали свою ефективність.

ПЕРЕЛІК ПОСИЛАНЬ

1. Andrea Arcuri. Restful api automated test case generation. In 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), с. 9–20. IEEE, 2019.
2. Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Rest-ler: automatic intelligent rest api fuzzing. arXiv preprint arXiv:1806.09739, 2019.
3. Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. An exploratory study on faults inweb api integration in a large-scale payment company. In 2018 IEEE/ACM 40th International Confer-ence on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), с. 13–22. IEEE, 2018.
4. Len Bass and Bonnie E John. Linking usability to software architecture patterns through general scenarios. Journal of Systems and Software, с.187–197, 2003.
5. Learn GraphQL [Електронний ресурс] – Режим доступу: https://graphql.org/learn/.
6. Stefan Bordag. A comparison of co-occurrence and similarity measures as simulations of context. In International Conference on Intelligent Text Processing and Computational Linguistics, с. 52–63. Springer, 2008.
7. Mark Boyd. How to choose architectural styles and specification formats for your apis, Sep 2017. URL https://www.programmableweb.com/news/how-to-choose-architectural-styles-and-specification-formats-your-apis/analysis/2017/09/27.
8. GraphQL Core Concepts Tutorial [Електронний ресурс] – Режим доступу: https://www.howtographql.com/basics/2-core-concepts/.
9. Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Automatic generation of test cases for rest apis: a specificationbased approach. In 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), с. 181–190. IEEE, 2018
10. Fabian Fagerholm and Jürgen Münch. Developer experience: Concept and definition. In 2012 international conference on software and system process (ICSSP), с. 73–77. IEEE, 2012.
11. Chushu Gao, Jun Wei, Hua Zhong, and Tao Huang. Inferring data contract for web-

based api. In 2014 IEEE International Conference on Web Services, с. 65–72. IEEE, 2014.
12. ISO/IEC. Iso/iec 25010: 2011 systems and software engineering-systems and software quality requirements and evaluation (square)-system and software quality models, 2011.
13. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In 2013 35th International Conference on Software Engineering (ICSE), с. 672–681. IEEE, 2013.
14. Panagiotis Louridas. Static code analysis. с. 58–61, 2006.
15. Переваги GraphQL | Blog Meline [Електронний ресурс] – Режим доступу: https://meline.lviv.ua/development/other/graphql/.
16. Alberto Martin-Lopez. Automated analysis of inter-parameter dependencies in web apis. In International Conference on Software Engineering, ACM Student Research Competition, 2020.
17. Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. A catalogue of inter-parameter dependencies in restful web apis. In International Conference on Service-Oriented Computing, с. 399–414. Springer, 2019.
18. Mark Masse. REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. " O'Reilly Media, Inc.", 2011.
19. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems, с. 3111–3119, 2013.
20. Brad A Myers and Jeffrey Stylos. Improving api usability. Communications of the ACM, с. 62–69, 2016.
21. Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Inter-parameter constraints in contemporary web apis. In International Conference on Web Engineering, с. 323–335. Springer, 2017.
22. Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In 2012 34th International Conference on Software Engineering (ICSE), с. 815–825, 2012.

23. Wieruch, R. The Road to Graphql: Your Journey to Master Pragmatic Graphql in JavaScript with React.Js and Node.Js. Wieruch Robbin – 1st edition. – New York: Independently published, 2018. – 314 p.
24. JSON-RPC 2.0 Specification [Електронний ресурс] – Режим доступу: https://www.jsonrpc.org/specification.
25. Irum Rauf, Elena Troubitsyna, and Ivan Porres. A systematic mapping study of api usability evaluation methods. Computer Science Review, 33:49–68, 2019.
26. Martin P Robillard and Robert Deline. A field study of api learning obstacles. Empirical Software Engineering, с. 703–732, 2011. Bibliography
27. Wendel Santos. Which api types and architectural styles are most used?, Feb 2018. URL https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26
28. Ahmed Seffah and Eduard Metzker. The obstacles and myths of usability and software engineering. Communications of the ACM, с. 71–76, 2004.
29. Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. Transactions on Software Engineering, с. 1083–1099, 2017.
30. REST Architectural Constraints [Електронний ресурс] – Режим доступу: https://restfulapi.net/rest-architectural-constraints/.
31. Raj Srinivasan. Rpc: Remote procedure call protocol specification version 2, 2005.
32. Phil Sturgeon. Understanding rpc vs rest for http apis, September 2016. URL https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/.
33. Jeffrey Stylos, Benjamin Graf, Daniela K Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. A case study of api redesign for improved usability. In 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, с. 189–192. IEEE, 2008.
34. Jaroslav Tulach. Practical API design: Confessions of a Java framework architect. Apress, 2008.
35. Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Inferring dependency constraints on parameters for web services. In Proceedings of the 22nd international conference on World Wide Web, с. 1421–1432, 2013.

36. Implementations – JSON-RPC [Електронний ресурс] – Режим доступу: https://www.jsonrpc.org/archive_json-rpc.org/implementations.html.
37. SOAP Version 1.2 Part 0: Primer (Second Edition) [Електронний ресурс] – Режим доступу: https://www.w3.org/TR/2007/REC-soap12-part0- 20070427/.
38. WSDL Web Services Description Language [Електронний ресурс] – Режим доступу: https://www.guru99.com/wsdl-web-services-descriptionlanguage.html/.
39. Architectural Styles and the Design of Network-based Software Architectures [Електронний ресурс] – Режим доступу: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.
40. History of REST APIs - Mobapi [Електронний ресурс] – Режим доступу: https://www.mobapi.com/history-of-rest-apis/.

Додаток А. Аналіз коду - нещасливий потік

Для кращого розуміння того, як можуть виглядати виклики на практиці, ми пропонуємо модель API, яка опрацьовує безліч обмежень у своєму потоці. Цей потік містить багато конструкцій, з якими важко мати справу при статичному аналізі коду, тому такий Додаток називається нещасливим потоком. Як показано на рис. А.1, HTTP-запит надсилається до API із заданою конкретною кінцевою точкою. Диспетчер обробляє цей запит. По-перше, запит десеріалізується за допомогою десеріалізатора (див. Лістинг А.1) у внутрішню модель даних (див. Лістинг А.2). Ця модель передається як запит до / платіжного контролера, який має два послідовних завдання, як показано в лістингу А.3 та лістингу А.4. Друге завдання також породжує новий потік завдань, здійснюючи виклик контролеру (див. Лістинг А.6).

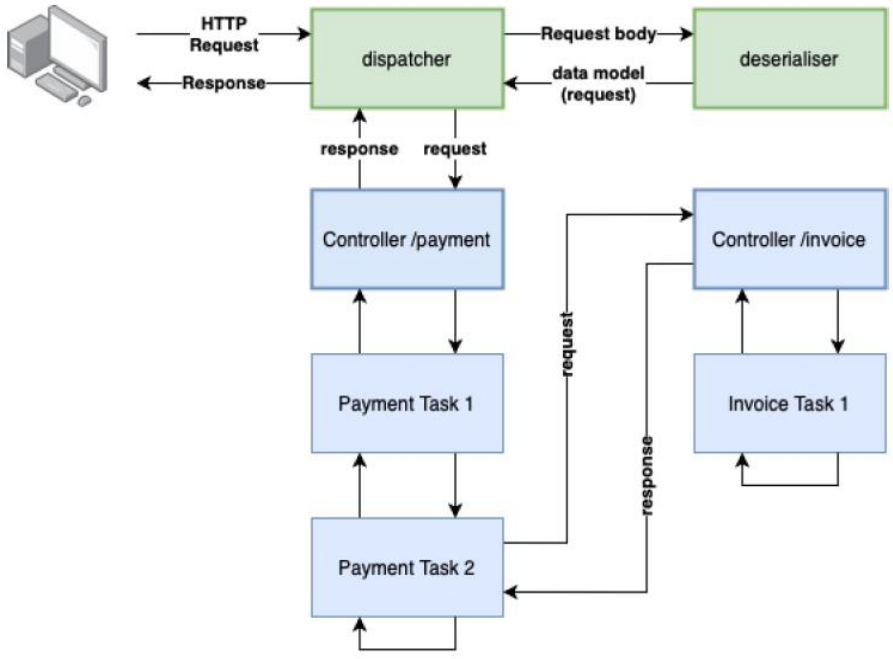


Рисунок А.1: Потік API, в якому запит робиться та обробляється диспетчером. Спочатку запит десеріалізується у свою внутрішню модель даних (запит), потім цей запит передається контролеру для кінцевої точки /payment. Ця кінцева точка включає два завдання та здійснює виклик контролеру /invoice.

Лістинг А.1 – Десеріалізатор запиту

```
class Deserialiser
{
    public static PaymentRequest deserialize(HttpRequest request) {
        // Create a new payment request model and populate it with the body.
        PaymentRequest pr = new PaymentRequest();
        JSONObject body = request.getBody();

        pr.setCard(body.get("card"));
        pr.setEncryptedCard(body.get("encryptedCard"));
        pr.setPaymentMethod(body.get("paymentMethod"));

        pr.setAmount(body.get("amount"));

        JSONObject invoiceDetails = body.get("invoiceDetails");
        pr.setInvoiceDetails(invoiceDetails);

        // Shopper email might have been provided through a different field.
        // B.2. the invoiceDetails.shopperEmail is dereferenced from the
            request's shopperEmail field.
        String shopperEmail = body.get("shopperEmail") != null ?
                                  body.get("shopperEmail") :
                                      invoiceDetails.get("shopperEmail");
        pr.setShopperEmail(shopperEmail);
        pr.setShopperCountry(body.get("shopperCountry"));

        return pr;
    }
}
```

Лістинг А.2. Модель платіжного запиту

```
class PaymentRequest extends AbstractRequest {
    Card card;
    EncryptedCard encryptedCard;
    String paymentMethod;

    Amount amount;
    InvoiceDetails invoiceDetails;

    String shopperEmail;
    String shopperCountry;

    // -- getters / setters --
}
```

Лістинг А.3 – Перше завдання, виконане в потоці /Payment

```
class PaymentTask1 {
    public boolean in(AbstractRequest request) {
        // Decrypt the card and add it to the request.
        decryptCard(request);
        // ...
    }

    public boolean out(AbstractRequest request) {
        buildResponse(request);
    }

    public void decryptCard(AbstractRequest request) {
        // Try to decrypt if there is another card.
        if (request.getEncryptedCard() != null) {
            try {
                Card card = Util.decrypt(request.getEncryptedCard());
                request.setCard(card);
            } catch (Exception decryptException) {
                Logger.info("Provided encrypted details invalid.");
            }
        }
    }
}
```

Лістинг А.4 – Друге завдання, виконане в потоці /Payment

```
class PaymentTask2 {
    public boolean in(AbstractRequest request) {
        // ...
        Card card = request.getCard();
        validateCard(card);

        // Design pattern: An initiation is retrieved from the PMInitiation
            factory, the specific handle method defines the
        // constraints.
        PMInitation initation = PMInitation.get(request.getBrand());
        initation.handle(request);

        if (request.getInvoiceDetails() != null) {
            // Spawn a new task chain for invoice requests.
            // B.8: Tracing through the logic of InvoiceController.handle()
            // is not feasible due to the complexity of the framework,
            // however invoice task 1 does enforce constraints.
            InvoiceController.handle(request);
        }
    }

    public void validateCard(Card card) throws Exception {
        // The card has to be provided.
        if (card != null) {
            // Check the details of the card...
            if (Util.isEmptyOrNull(card.getHolder())) {
                throw Exception("No card holder provided.");
            }
        } else {
            // B.2: Either encryptedCard or card was needed. We can not
            // infer this, since we do not know that the card field can be
            // populated by both the encryptedCard information and normal
            // card information. Payment task 1 relates to this.
            throw Exception("No card provided");
        }
    }
}
```

Лістинг А.5 – Фабричний шаблон, що використовується для методів оплати

```
    ...

    Handler handler;

    PMInitation(Handler handler) {
        this.handler = handler;
    }

    public static PMInitation get(String method) {
        // Mapping of alternatives.
        switch (method) {
            case "iDeal":
                method = "ideal";
                break;
            default:
                break;
        }

        // Find the correct initiation.
        for(PMInitation pmInitation : PMInitation.values()) {
            if (pmInitation.name().equals(method)) {
                return pmInitation;
            }
        }

        // Fallback option, nothing found.
        return fallback;
    }

    public void handle(Request request) {
        handler.handle(request);
    }
}

class IDeal() {
        public static void handle(Request request) throws Exception {
            if (request.getShopperEmail() == null) {
                throw Exception("Shopper email is missing");
            }
            // ...
    }
}
```

Лістинг А.6 – Перше і єдине завдання, виконане в потоці /invoice

```
class InvoiceTask1 {
    public boolean in(PaymentRequest request) {
        InvoiceDetails details = request.getInvoiceDetails();
        OpenInvoice openInvoice = details.getOpenInvoice();

        if (openInvoice != null) {
            int whole = openInvoice.getTotalAmount();
            int sum = 0;

            // B.5: We only iterate the body of the for-loop once.
            for (Entry entry : openInvoice.getEntries()) {
                // B.3.: A new object is created in which the reference to
                    the invoiceDetails.openInvoice.amount is lost.
                OpenInvoiceParameters params = new OpenInvoiceParameters(
                    openInvoice.getAmount(), openInvoice.getDueDate());
                OpenInvoiceValidator.validate(params);

                // Add the amount of a single entry to the sum.
                sum += openInvoice.getAmount();
            }

            // B.7: We do not keep track of all the amounts that
                contributed to the sum.
            // Therefore we can not infer this arithmetic constraint.
            if (sum != whole) {
                Logger.info("...");
                throw Exception("The whole can not be greater than the sum
                    of its parts.");
            }
        }
    }
}

class OpenInvoiceParameters {
    Amount openInvoiceAmount;
    Date dueDate;

    public OpenInvoiceParameters(Amount amount, Date dueDate) {
        this.openInvoiceAmount = amount;
        this.dueDate = dueDate;
    }

    // -- getters / setters
}
```

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

АВТОМАТИЧНЕ ВИЗНАЧЕНЯ ОБМЕЖЕНЬ ПАРАМЕТРІВ ДЛЯ СКЛАДНИХ ВЕБ-API ДЛЯ ПЛАТІЖНИХ СИСТЕМ

Студент: Пономарьов Костянтин Володимирович, ПДМ-61

Науковий керівник: к.т.н., старший викладач, Яскевич Владислав Олександрович

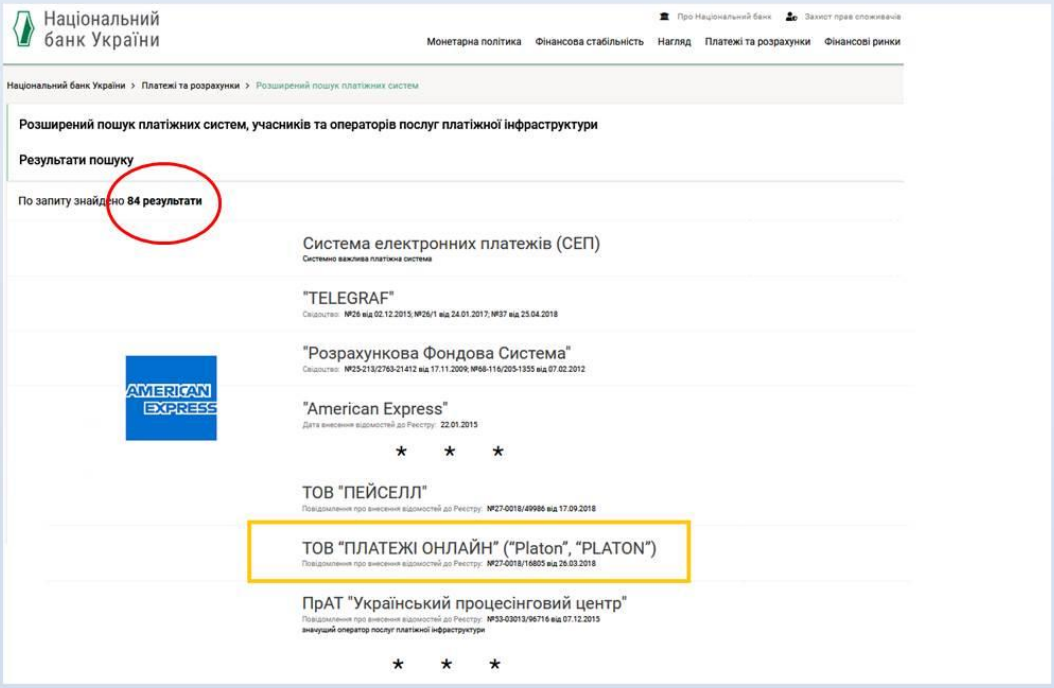
ІСНУЮЧА ПРОБЛЕМА

- Електронна комерція – основний тренд сьогодення
- Платіжні системи – підґрунтя існування електронної комерції
- Оптимізація роботи платіжних систем – актуальна необхідність
- Параметри веб- API платіжних систем взаємно пов`язані та мають обмеження, визначення яких важливо для розробників доданків
- Користувачі API платіжних систем, як правило, не мають чіткої інформації по обмеженням параметрів

ІСТОРИЧНИЙ ОГЛЯД

- 1960-ті рр. – перші системи електронної комерції (США)
- 1980-ті рр. – ISO розробила новий стандарт Electronic Data Interchange
- 1993 р. – початок експлуатації Системи електронних міжбанківських розрахунків НБУ
- 2020р. – в Україні працює 84 (!) платіжні системи.

СТАН ПЛАТІЖНИХ СИСТЕМ В УКРАЇНІ

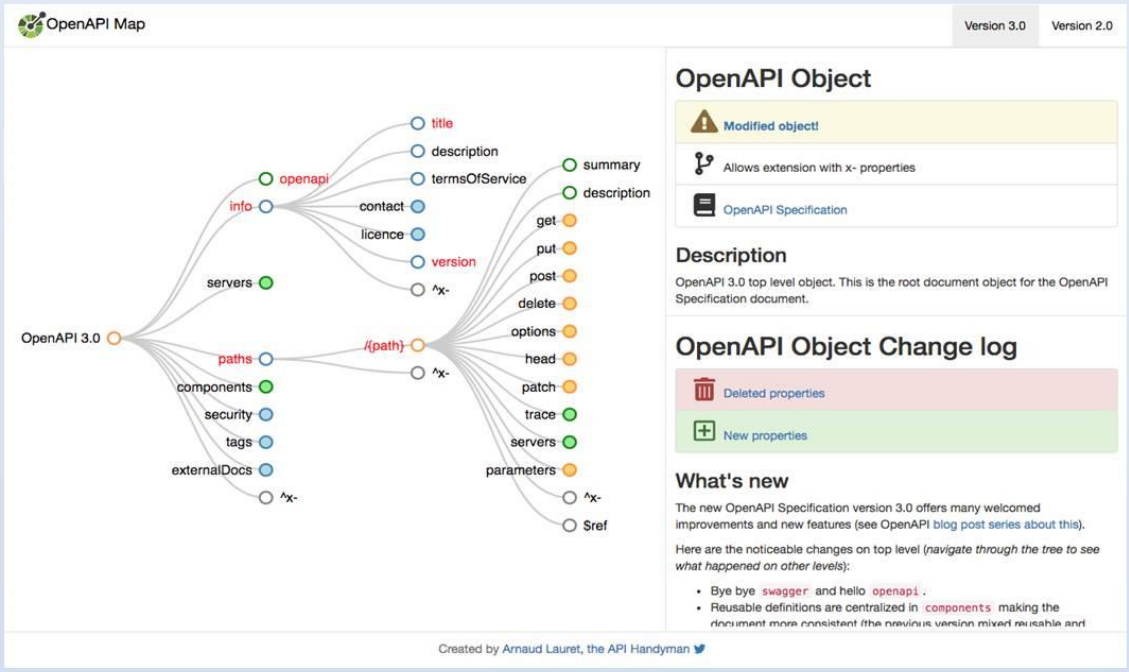


СУТЬ РОБОТИ

- *Об`єкт дослідження* – процес розробки прикладного ПЗ платіжних систем.
- *Предмет дослідження* – параметри веб-API платіжних систем.
- *Мета роботи* –розробка автоматичного визначення обмежень параметрів для складних веб-API платіжних систем.
- *Методи дослідження* – аналіз параметрів, порівняння підходів.

ТИПОВА СХЕМА РЕАЛІЗАЦІЇ ЕЛЕКТРОННОГО ПЛАТЕЖУ

КАРТА OpenApi Specifications (OAS)



ОБМЕЖЕННЯ ПАРАМЕТРІВ ПЛАТІЖНОЇ СИСТЕМИ

Чіткий огляд обмежень у веб-API дуже важливий, оскільки це допомагає споживачам API інтегрувати API без необхідності підтримки компанії. Неповна або неправильна документація щодо цих обмежень може втратити багато часу та спричинити дорогі помилки інтеграції. В даний час ці обмеження задокументовані та підтримуються розробниками API **вручну**, що може бути трудомістким та складним завданням. Ця складність пов'язана з розміром та складністю кодової бази веб-служби, а також документацією, яку надають не ті люди, які пишуть код. **Тому потрібні інструменти, які допомагають розробникам API виявляти та підтримувати обмеження в своїх API**.

РОЗРОБКА МЕТОДИКИ ДОСЛІДЖЕННЯ

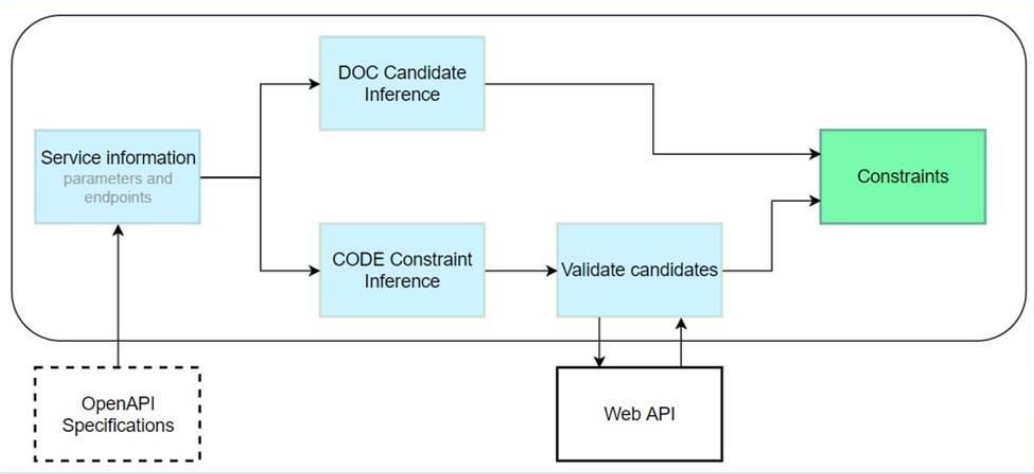
У даній магістерській роботі використані **два** підходи для виявлення обмежень параметрів для складних веб-API.

Аналіз онлайн-документації

Аналіз статичного коду

ІСНУЮЧІ ПІДХОДИ ДЛЯ ВИЯВЛЕННЯ ОБМЕЖЕНЬ ПАРАМЕТРІВ

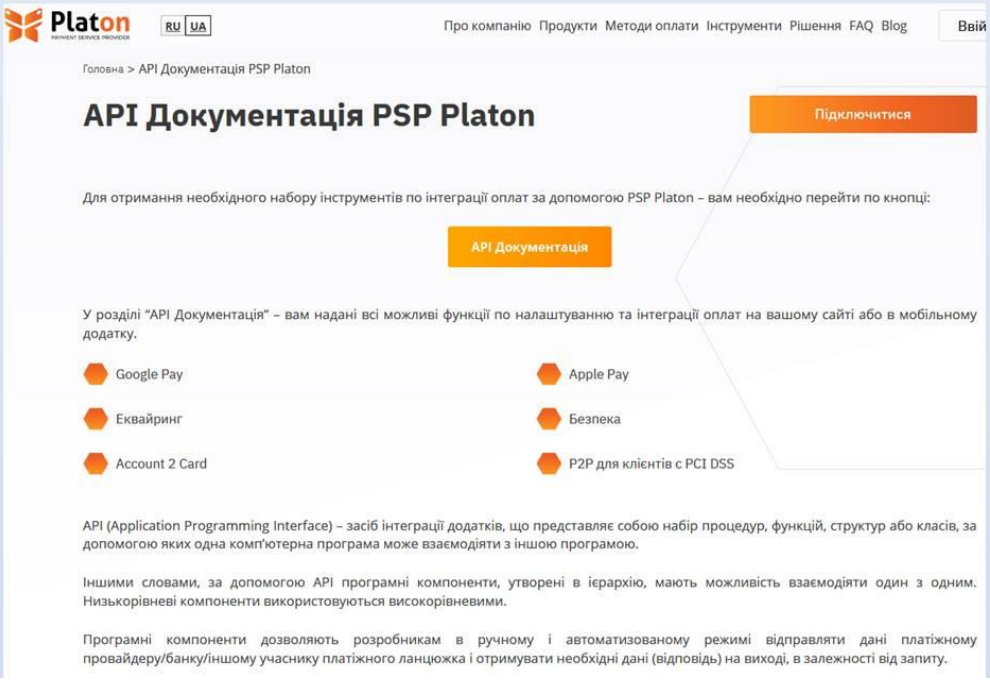
На першому кроці збирається інформація про параметри кінцевих точок веб-API із специфікації OpenAPI (OAS)



Огляд процесу, що показує початковий етап збору інформації про послуги, а потім підхід до аналізу документації та коду.

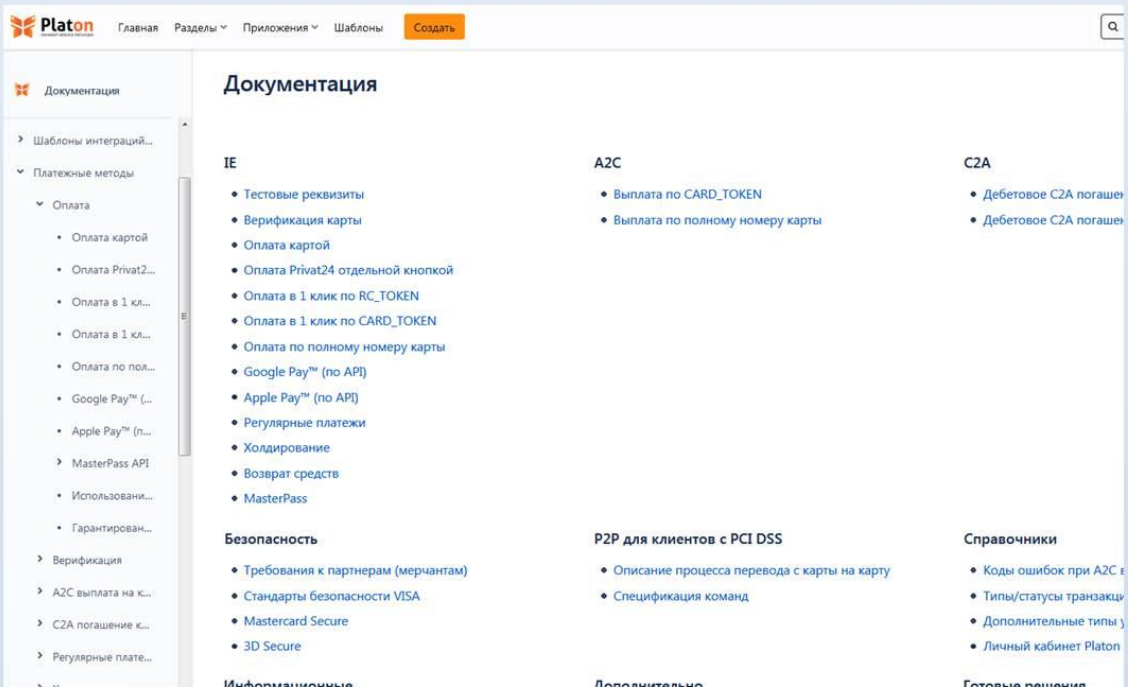
АНАЛІЗ ДОКУМЕНТАЦІЇ

ДОКУМЕНТАЦІЯ "PLATON" У ВІДКРИТОМУ ДОСТУПІ

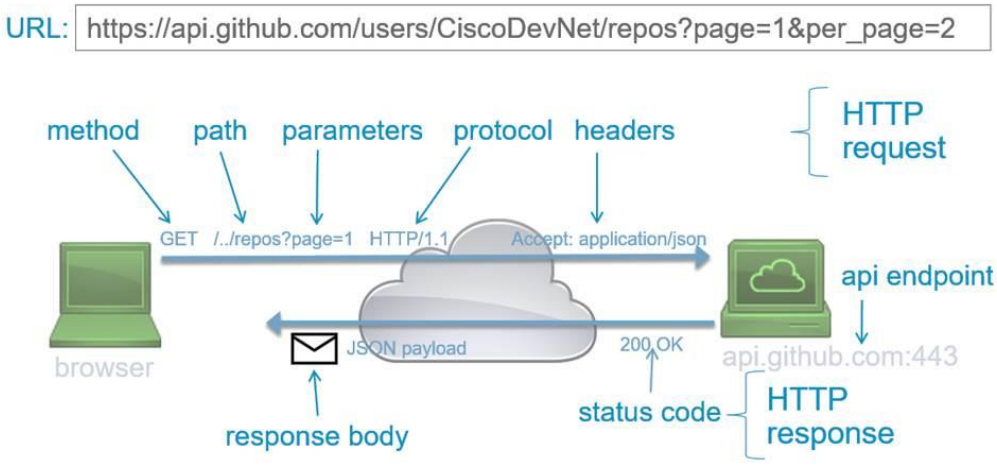


АНАЛІЗ ДОКУМЕНТАЦІЇ

ДОКУМЕНТАЦІЯ "PLATON" У ВІДКРИТОМУ ДОСТУПІ

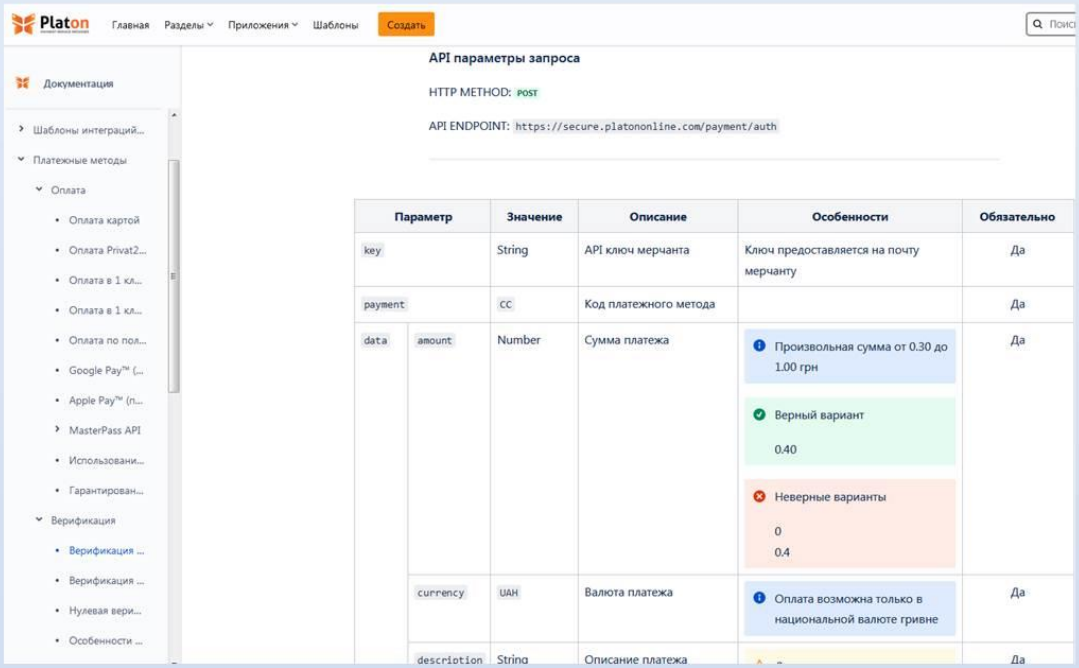


АНАТОМІЯ ЗАПИТІВ API



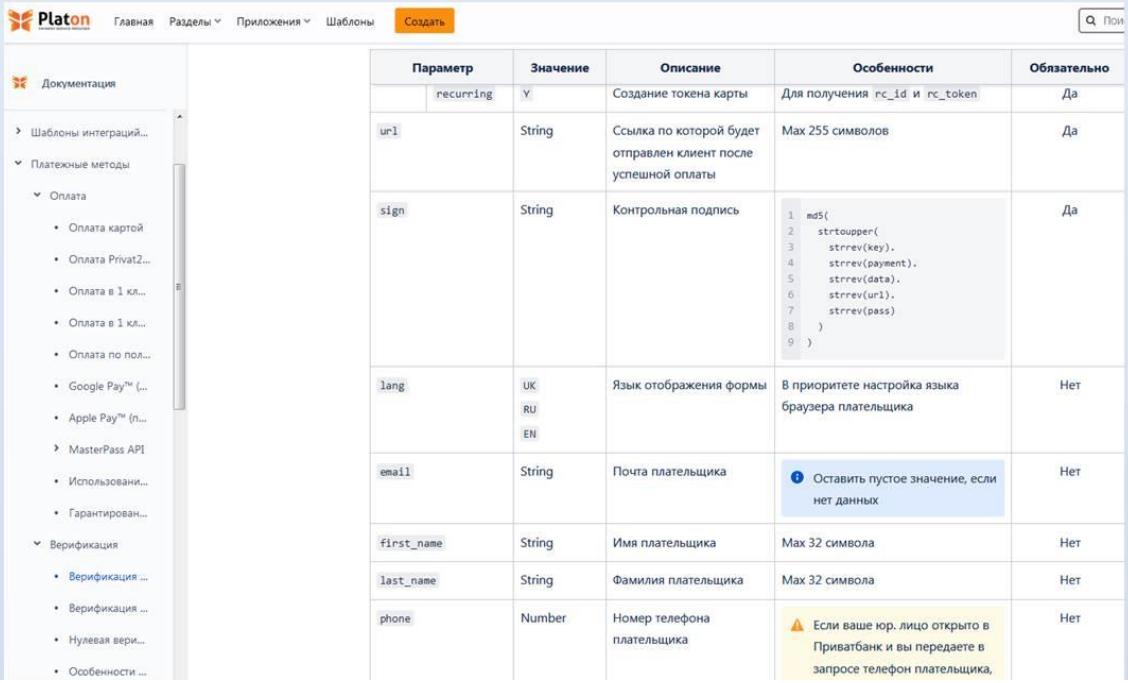
АНАЛІЗ ДОКУМЕНТАЦІЇ

ПАРАМЕТРИ ПЛАТІЖНОЇ СИСТЕМИ



АНАЛІЗ ДОКУМЕНТАЦІЇ

ПАРАМЕТРИ ПЛАТІЖНОЇ СИСТЕМИ



Параметр	Значение	Описание	Особенности	Обязательно
recurring	Y	Создание токена карты	Для получения rc_id и rc_token	Да
url	String	Ссылка по которой будет отправлен клиент после успешной оплаты	Max 255 символов	Да
sign	String	Контрольная подпись	md5(strtoupper(strrev(key). strrev(payment). strrev(data). strrev(url). strrev(pass)))	Да
lang	UK RU EN	Язык отображения формы	В приоритете настройка языка браузера плательщика	Нет
email	String	Почта плательщика	Оставить пустое значение, если нет данных	Нет
first_name	String	Имя плательщика	Max 32 символа	Нет
last_name	String	Фамилия плательщика	Max 32 символа	Нет
phone	Number	Номер телефона плательщика	Если ваше юр. лицо открыто в Приватбанк и вы передаете в запросе телефон плательщика,	Нет

АНАЛІЗ ДОКУМЕНТАЦІЇ

Матриця співіснування для формування кандидатів

	card	bankAccount	type	reference
card	-	2	0	X
bankAccount	-	-	1	X
type	-	-	-	X
reference	-	-	-	-

Табл. 3.1. Приклад матриці суміжності для кінцевої точки з чотирма параметрами.

\

АНАЛІЗ ДОКУМЕНТАЦІЇ

Матриця суміжності для формування кандидатів

	card	bankAccount	type	reference
card	-	2	0	X
bankAccount	-	-	1	X
type	-	-	-	X
reference	-	-	-	-

Табл. 3.1. Приклад матриці суміжності для кінцевої точки з чотирма параметрами.

Перевірка кандидатів

card	bank	Result	Constraint
0	0	F	-
0	1	T	and(!card, bank)
1	0	T	and(card, !bank)
1	1	T	and(card, bank)

Табл. 3.2 – Таблиця перевірки для двох параметрів, де 0 та 1 вказують на відсутність та наявність параметрів. У стовпці результату вказується, чи був запит успішним (T) чи ні (F).

АНАЛІЗ ДОКУМЕНТАЦІЇ

СТВОРЕННЯ ЗАПИТУ

```
{
 "accountName": "Medina"
}
```

```
{
 "accountName": "Medina",
 "bankAcount": "DE8098"
}
```

```
{
 "accountName": "Medina",
 "card": "12356",
}
```

```
{
 "accountName": "Medina",
 "card": "12356",
 "bankAcount": "DE8098"
}
```

Рисунок 3.3 – Приклади тіл запитів, створених на основі таблиці 3.1 (матриця співіснування)

АНАЛІЗ ДОКУМЕНТАЦІЇ

ОБЧИСЛЮВАЛЬНІ ВИТРАТИ

Загальний час, для перевірки обмежень, залежить від ряду факторів, найголовніше від кількості кандидатів, кількості параметрів для кандидата та кількості запитів, які можна зробити кожну секунду.

Необхідний час виконання, у секундах, дорівнює , $\sum_{i=1}^{C} 2^{|P_{C_i}|}/5$ де *C* позначає набір кандидатів, а $|P_{C_i}|$ - кількість параметрів у кандидата i.

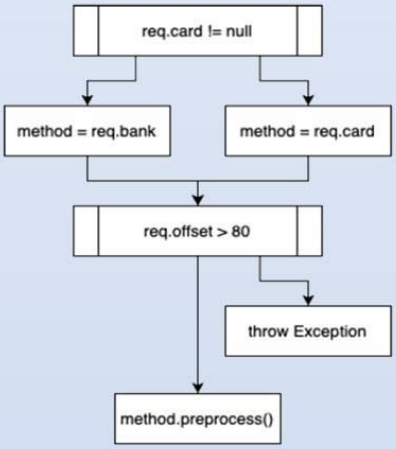
Даний параметр може мати кілька значень, це обчислення змінюється. Кількість комбінацій визначається як $\prod_{i=1}^{p} |values(p_i)| + 1$
Дотримуючись таблиці 3.3, знаючи, що gas може мати 2 значення, а temperature - лише 1, робимо 6 запитів. Заміна цього терміну замість $|P_{C_i}|$
дасть нам очікуваний час роботи кандидатів які враховують конкретні значення.

АНАЛІЗ СТАТИЧНОГО КОДУ

За допомогою аналізу коду ставиться задача витягти обмеження з контрольної структури вихідного коду. Для цього аналізуються методи, що мають відношення до обробки запитів HTTP, поданих до API.

```
def handle(Request req)
    if req.getCard() != null then
        method = req.getCard()
        validateCard(method)
    else
        method =
         req.getBankAccount()
    end
    if req.getOffset() > 80 then
        throw Exception()
    end
    ⋮
    method.preprocess()
```

Приклад методу обробки запиту API



Відповідний Граф потоку керування (складається для кожного методу)

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Дослідницьке питання 1

Наскільки ефективним є аналіз документації та статичного коду для виявлення обмежень параметрів у великомасштабному корпоративному API?

	Total	Code	Code FP	Doc	Doc FP	Both
/payments	17	11	2	0	0	0
/authorise	15	11	4	3	0	2
/capture	5	2	0	1	0	0
/storeDetailAndSubmit...	5	2	0	1	0	1
/createAccountHolder	4	0	0	3	0	0
/getAccountHolder	1	0	0	0	0	0
/updateAccountHolder	1	1	0	0	0	0
/createAccount	1	0	1	1	0	0
/uploadDocument	3	1	1	1	0	1
/getCostEstimate	1	0	0	1	0	0
Total:	53	28	8	11	0	4

Табл. 5.1 – Для кожної кінцевої точки розглядається: загальна кількість ідентифікованих вручну обмежень між параметрами, кількість обмежень, виявлених кодом та аналізом документації, з відповідними помилковими спрацьовуваннями (FP) та кількість обмежень, визначених обома підходами

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Дослідницьке питання 1

Наскільки ефективним є аналіз документації та статичного коду для виявлення обмежень параметрів у великомасштабному корпоративному API?

	Total	Identified
/payments	9	8
/authorise	14	10
/capture	5	5
/storeDetailAndSubmit...	4	4
/createAccountHolder	4	1
/createAccount	1	1
Total:	37	29

Табл. 5.2 – Загальна кількість та кількість ідентифікованих обмежень одного параметра для кожної кінцевої точки з використанням аналізу коду

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Дослідницьке питання 1

Наскільки ефективним є аналіз документації та статичного коду для виявлення обмежень параметрів у великомасштабному корпоративному API?

1. Аналіз коду та документації разом виявляє 66% обмежень між параметрами.
2. Аналіз коду та документації виявляє різні обмеження.
3. Аналіз коду виявляє більше обмежень, але з більшою кількістю помилкових спрацьовувань.
4. Для більшості обмежень з одним параметром використовуються зручні для виведення структури коду для перевірки значень параметрів.
5. Аналіз коду виявляє 78% обмежень з одним параметром.

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Дослідницьке питання 2

Які є проблеми у використанні аналізу документації або статичного коду для виявлення обмежень між параметрами?

АНАЛІЗ ДОКУМЕНТАЦІЇ :

1. Брак інформації.
2. Неявні посилання.
3. Значення не виявлено.
4. Неспостережувані обмеження.
5. Запит залежностей.

АНАЛІЗ КОДУ :

1. Параметр не виявлено.
2. Параметр без посилання.
3. Статичний змінний стек.
4. Передумови.
5. Структура управління.
6. Потік даних.
7. Синтаксис арифметичного обмеження.
8. Шаблони дизайну.
9. Некодові обмеження.

ВИСНОВКИ

1. Було розроблено два підходи для виявлення обмежень параметрів для складних веб-API. Підходи на основі документації та вихідного коду здатні ідентифікувати 21% та 53% відсотків обмежень відповідно.
2. Підхід, заснований на документації, значною мірою страждає від браку явної інформації, що описує обмеження. Статичний аналіз коду, як правило, може витягувати обмеження з вихідного коду.
3. Основні проблеми даного дослідження, були нечутливість потоку даних та розробка підходу до аналізу статичного коду.
4. Визначена ефективність аналізу документації та статичного коду для виявлення обмежень параметрів у великому корпоративному API. Аналіз коду та документації в сукупності отримує 66% обмежень між параметрами. Аналіз коду отримує 78% обмежень з одним параметром.
5. Були виявлені проблеми у використанні аналізу документації або статичного коду для виявлення обмежень між параметрами.
6. За умови великої кількості запитів, які можна зробити для локальної збірки API, підходи на основі пошуку наборів параметрів показали свою ефективність.

ДЯКУЮ ЗА УВАГУ!