НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

«На правах рукопису»
УДК 004.023

До захисту допущено:

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою

«Системне програмування і спеціалізовані комп'ютерні системи»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Семантичний спосіб поєднання версій програмного коду при одночасній розробці програм для підмножини мови Python»

Виконав:
студент ІІ курсу, групи КВ-02мп
Вовчок Олексій Володимирович _____

Науковий керівник:
Доцент кафедри СПіСКС, к.т.н., доцент
Марченко Олександр Іванович _____
_____

Рецензент:

_____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування і спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«___»_____2020 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Вовчку Олексію Володимировичу

1. Тема дисертації «Семантичний спосіб поєднання версій програмного коду при одночасній розробці програм для підмножини мови Python», науковий керівник дисертації Марченко Олександр Іванович, доцент кафедри СПіСКС, к.т.н., доцент, затверджені наказом по університету від «5» 11. 2021 р. №3682-С

2. Термін подання студентом дисертації 7.12.2021

3. Об'єкт дослідження: методика поєднання версій програмного коду при одночасній розробці програм

4. Вихідні дані:

- версії програмного коду, що були написані при одночасній розробці програм для підмножини мови Python, побудовані з використанням інструменту git.

5. Перелік завдань, які потрібно розробити
- аналіз існуючих рішень для задачі поєднання версій програмного коду при одночасній розробці програм;

- дослідження синтаксичної та семантичної специфіки поєднання програм, написаних на мові Python;
- створення семантичного способу поєднання версій програмного коду при одночасній розробці програм для підмножини мови Python на основі циклічного алгоритму доведення еквівалентності програм та алгоритму виведення семантичного ефекту модифікації програми;
- перевірка результатів створеного інструмента поєднання на реальних проєктах, написаних на мові Python;

6. Орієнтовний перелік графічного (ілюстративного) матеріалу -презентація

7. Орієнтовний перелік публікацій - 2 тез:
- публікація в VIII міжнародній науково-технічній Internet-конференції в НУХТ;
- публікація в XIV науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2021.

9. Дата видачі завдання 7.10.2020

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
	Вибір тематики роботи	10.09.2020	
	Вивчення літератури за обраною тематикою	10.01.2021	
	Постановка задачі для магістерської дисертації	01.02.2021	
	Аналіз існуючих рішень для поставленої задачі	01.05.2021	
	Підготовка першого розділу магістерської дисертації	10.09.2021	
	Підготовка другого розділу магістерської дисертації	01.10.2021	
	Підготовка третього розділу магістерської дисертації	10.10.2021	
	Підготовка четвертого розділу магістерської дисертації	01.11.2021	
	Підготовка графічної частини	10.11.2021	
	Оформлення документації магістерської дисертації	20.11.2021	
	Попередній розгляд магістерської дисертації на кафедрі	01.12.2021	

Студент Олексій ВОВЧОК

Науковий керівник Олександр МАРЧЕНКО

РЕФЕРАТ

Актуальність теми. В даний час програмні продукти створюються групами розробників. Зазвичай, кожному розробнику видається своє окреме завдання, результатом якого є видозмінений програмний продукт. Розробники працюють незалежно і одночасно, а тому можуть вносити суперечливі зміни. Ці зміни унеможливлюють автоматичну інтеграцію незалежно від створених модифікацій в функціональність програмного продукту. Попередні дослідження показали, що такі конфлікти трапляються часто і погіршують продуктивність розробки, оскільки їх розуміння та вирішення є складним і нудним завданням, причому неякісне виконання якого може внести дефекти в продукт. Тому розробка спеціальних способів поєднання версій програмного коду при одночасній розробці програм для підмножини мови Python та застосування їх на практиці є актуальною і важливою задачею, як з наукової, так і з практичної точки зору.

Об'єктом дослідження є методика поєднання версій програмного коду при одночасній розробці програм.

Предметом дослідження є семантичні способи поєднання версій програмного коду при одночасній розробці програм для підмножини мови Python.

Мета роботи: створення семантичний спосіб поєднання версій програмного коду при одночасній розробці програм для підмножини мови Python на основі циклічного алгоритму доведення еквівалентності програм та алгоритму виведення семантичного ефекту модифікації програми; перевірка результатів створеного інструмента поєднання на реальних проєктах, написаних на мові Python.

Наукова новизна полягає в наступному: вперше запропоновано спосіб поєднання версій програм, написаних на підмножині мови Python на основі циклічного алгоритму доведення еквівалентності програм та алгоритму виведення семантичного ефекту модифікації програми.

Практична цінність отриманих в роботі результатів полягає в тому, що запропоновані методи дають змогу автоматично поєднувати програми, написані на підмножині мови Python, звільняючи таким чином розробників від розуміння та вирішення складного і нудного поєднання версій програм, неякісне виконання якого може внести дефекти в продукт. Розроблений автоматичний інструмент поєднання програм, написаних на підмножині мови Python перевірено на реальних проєктах. Отриманий результат дозволяє знизити кількість затраченого розробниками часу на процес поєднання версій програм.

Апробація роботи.

1. Постановка задачі була представлена і обговорена VIII Міжнародній науково-технічній Internet-конференції присвяченій актуальним проблемам керування складними об'єктами та підготовки фахівців з автоматизації виробництва і використання інформаційних технологій

2. Основні положення і результати роботи були представлені та обговорювались на XIV науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2021 (Київ, 15-17 листопада 2021 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напряму досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи, наведено відомості про апробацію результатів і їхнє впровадження.

У першому розділі розглянуто існуючі методики поєднання версій програмного коду при одночасній розробці програм, а також проведений аналіз, який дає змогу визначити основні переваги та недоліки цих методик.

У другому розділі коротко описуються аспекти використання циклічного алгоритму доведення еквівалентності програм та алгоритму виведення семантичного ефекту модифікації програми.

У третьому розділі наведено опис реалізації спосіб поєднання версій програм, написаних на підмножині мови Python на основі циклічного алгоритму доведення еквівалентності програм та алгоритму виведення семантичного ефекту модифікації програми.

У четвертому розділі описуються теоретичні та експериментальні результати застосування способу на реальних проєктах.

У висновках представлені результати проведеної роботи.

Робота представлена на 81 аркушах, містить 14 джерел, 63 рисунків, 4 додатки.

Ключові слова: поєднання версій програм, Python, синтаксичний аналіз, семантичний аналіз, еквівалентність програм, модифікація програми.

ABSTRACT

Actuality of theme. Currently, software products are created by teams of developers. Usually, each developer is given a separate task, the result of which is a modified software product. Developers work independently and simultaneously, and therefore can make contradictory changes. These changes make it impossible to automatically integrate, regardless of the modifications created into the functionality of the software product. Previous research has shown that such conflicts occur frequently and impair development performance, as understanding and resolving them is a complex and tedious task, and poor performance can lead to product defects. Therefore, the development of special ways to combine versions of software code while developing programs for a subset of Python and their application in practice is an urgent and important task, both from a scientific and practical point of view.

The object of research is the method of combining versions of program code while developing programs.

The subject of the study is semantic ways of combining versions of program code while developing programs for a subset of the Python language.

Purpose: to create for the first time a semantic way to combine versions of program code while developing programs for a subset of Python based on a cyclic algorithm for proving the equivalence of programs and an algorithm for deriving the semantic effect of program modification; check the results of the created combination tool on real projects written in Python.

The scientific novelty is as follows: for the first time a method of combining versions of programs written in a subset of Python based on a cyclic algorithm for proving the equivalence of programs and an algorithm for deriving the semantic effect of program modification.

The practical value of the results is that the proposed methods allow you to automatically combine programs written in a subset of Python, thus freeing developers from

understanding and solving complex and tedious combination of software versions, poor performance of which can cause defects in the product. The developed automatic tool for combining programs written in a subset of the Python language has been tested on real projects. The result allows you to reduce the amount of time spent by developers on the process of combining versions of programs.

Approbation of work.

1. The statement of the problem was presented and discussed by the VIII International scientific and technical Internet-conference devoted to actual problems of management of difficult objects and preparation of experts in automation of production and use of information technologies

2. The main provisions and results of the work were presented and discussed at the XIV scientific conference of undergraduates and graduate students "Applied Mathematics and Computing" PMK-2021 (Kyiv, November 15-17, 2021).

Structure and scope of work. The master's dissertation consists of an introduction, four chapters and conclusions.

The introduction presents a general description of the work, assesses the current state of the problem, substantiates the relevance of research, formulates the purpose and objectives of research, shows the scientific novelty of the results and practical value of the work, provides information on approbation of results and their implementation.

The first section discusses the existing methods of combining versions of software code while developing programs, as well as an analysis that identifies the main advantages and disadvantages of these methods.

The second section briefly describes the aspects of using the cyclic algorithm for proving the equivalence of programs and the algorithm for deriving the semantic effect of program modification.

The third section describes the implementation of a method of combining versions of programs written in a subset of the Python language based on a cyclic algorithm for proving

the equivalence of programs and an algorithm for deriving the semantic effect of program modification.

The fourth section describes the theoretical and experimental results of the method on real projects.

The conclusions present the results of the work.

The work is presented on 81 sheets, contains 14 references, 4 appendices.

Keywords: combination of program versions, Python, parsing, semantic analysis, program equivalence, program modification.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

ГЗП – граф залежностей програми.

ІП – інструмент поєднання.

КП – конфлікт поєднання.

ЛА – лексичний аналіз.

МГЗП – міжпроцедурний ГЗП.

ППК – представлення програмного коду.

СА – синтаксичний аналіз.

ВСТУП

В даний час програмні продукти створюються групами розробників. Зазвичай, кожному розробнику видається своє окреме завдання, результатом якого є видозмінений програмний продукт. Розробники працюють незалежно і одночасно, а тому можуть вносити суперечливі зміни. Ці зміни унеможливлюють автоматичну інтеграцію незалежно від створених модифікацій в функціональність програмного продукту.

Попередні дослідження показали, що такі конфлікти трапляються часто і погіршують продуктивність розробки, оскільки їх розуміння та вирішення є складним і нудним завданням, причому неякісне виконання якого може внести дефекти в продукт [1], [2].

Зауважимо, що необхідність об'єднання програмного забезпечення залежить від обраного механізму контролю версій [6]. Скажімо існує підхід до використання системи контролю версій, де усі учасники працюють над одним набором програмних продуктів, а паралельне редагування одного і того ж продукту запобігається блокуванням. Такий протокол блокування передбачає сувору модель узгодженості в тому сенсі, що всі учасники завжди мають один екземпляр початкового коду. На практиці, як тільки кількість розробників програмного забезпечення, що працюють паралельно, перевищує певний, зазвичай дуже низький поріг, цей підхід стає суттєво незручним. З альтернативними підходом до використання системи контролю версій, кожен розробник може працювати над особистою копією програмного продукту. Перевага такого підходу в тому, що розробникам програмного забезпечення дозволено працювати повністю окремо на деякий час. Ціною цього є необхідність поєднання незалежних модифікацій. Час від часу особисті копії потрібно інтегрувати в нову спільну версію, а конфлікти між одночасними змінами потрібно вирішувати під час вже окремого процесу поєднання.

Поточні некомерційні інструменти переважно являють собою інструменти ручного текстового поєднання. Зокрема, такі ІП для системи контролю версій git, як meld або kdiff3 збирають текстові конфлікти. Цей процес являє собою збір рядків, що призводять до конфліктів поєднання. Далі користувачу надається можливість вибору між альтернативними версіями та навіть ручної редакції результуючого рядка.

Крім суто текстового поєднання також існують комерційні ІП, що полегшують роботу програміста в задачах поєднання. Такі ІП створюються для певної специфічної мови програмування, застосовуючи синтаксичну та семантичну інформацію про цільову мову. Таким чином, перетворюючи програмний код в певну структуру даних та задекларувавши правила поєднання цих структур даних, з'являється можливість автоматизувати задачу ручного поєднання програмного коду. За рахунок таких автоматизації можна досягти пришвидшення вирішення задачі поєднання ПК, звільнивши програміста від рутинної роботи. Водночас, строга формалізація правил поєднання забезпечать зниження ймовірності внесення дефектів в програмний продукт.

В той же час, для створення автоматизованого ІП необхідно побудувати систему, що проводить статичний аналіз коду, що враховує різноманітні семантичні властивості цільової мови програмування. Фактично має бути забезпечена часткова компіляція і симуляція запуску програми, щоб забезпечити потрібну поєднану модифікацію із двох конфліктуючих версій.

Щоб впоратися з цією проблемою, в літературі присутній ряд дослідницьких прототипів, які займаються синтаксичними та семантичними конфліктами поєднання. На жаль, виявити всі можливі семантичні конфлікти між двома версіями довільного програмного продукту неможливо [5], [13]. Щоб зменшити цю проблему, можна звузити алгоритм поєднання до чітко визначеної галузі, наприклад, конкретної мови програмування, певної фази життєвого циклу програмного забезпечення або певного типу програми (наприклад, розподілені програми).

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ЗАДАЧІ ПОЄДНАННЯ ПРОГРАМ

У цьому розділі наведено вичерпний огляд різних методів поєднання (що використовуються в дослідницьких прототипах, а також у комерційних інструментах), про які повідомлялося в літературі, і класифікує їх відповідно до ряду ортогональних розміри. Іншими словами, обговорюються такі альтернативи, як двостороннє або тристороннє поєднання, текстове, синтаксичне, семантичне чи поєднання на основі операцій; поєднання на основі стану або змін.

1.1 Постановка задачі

Нехай надано програмний проєкт, що розроблюється одночасно кількома незалежними розробниками. Кожен реалізує окрему функціональність, створюючи модифікації відносно певної початкової версії. По завершенню реалізації, всі версії повинні бути злиті в одну, поєднання виконується по дві версії, поки всі не будуть поєднані. Розглянемо детальніше, нехай A – початкова версія (master), B і C – незалежні версії (рисунок 1):

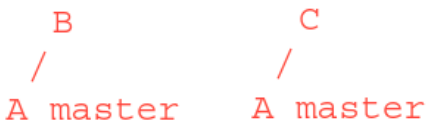

Рисунок 1 – Початковий стан версій

Спочатку поєднуємо B і master. В даному випадку виконується тривіальне перенесення модифікацій без видозмін із B в master (рисунок 2).

Рисунок 2 – Стан головної версії

Після того як B поєднано в master, можуть виникнути конфлікти поєднання при спробі додати C в master (рисунок 3).

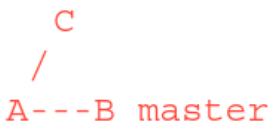

Рисунок 3 – Конфліктний стан

Конфлікт поєднання – проблема, що постає при поєднанні версій програмного продукту, що мають незалежні модифікації однієї й тієї ж частини коду. Задача полягає у необхідності автоматично вирішити якнайбільше таких конфліктів, тобто, використавши досліджений спосіб, побудувати поєднану модифікацію із двох, що конфліктують. А нерозв'язані конфлікти передати користувачу для ручного вирішення.

1.2 Двостороннє і тристороннє поєднання

Перше розрізнення можна зробити між двостороннім і тристороннім методами поєднання. Двостороннє поєднання намагається об'єднати дві версії програмного продукту, не покладаючись на початкову версію, від якої походять дочірні версії. При тристоронньому поєднанні інформація про початкову версію також використовується під час процесу поєднання. Це робить тристороннє поєднання потужнішим, ніж його двосторонній варіант, оскільки він здатний виявити більше конфліктів. Тому майже всі доступні на даний момент інструменти поєднання використовують тристороннє поєднання.

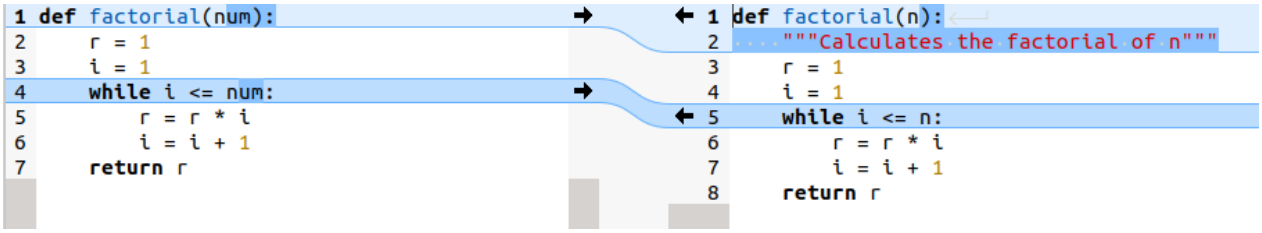

Рисунок 4 – Модифікація версії 1 (зліва початкова, справа кінцева)

Щоб показати різницю між двостороннім і тристороннім поєднаннями, більш чітко, розглянемо на прикладі, де представлені відповідні модифікації функції для обчислення факторіала (рисунок 4 і рисунок 5). Для ілюстрації використано утиліту meld [3].

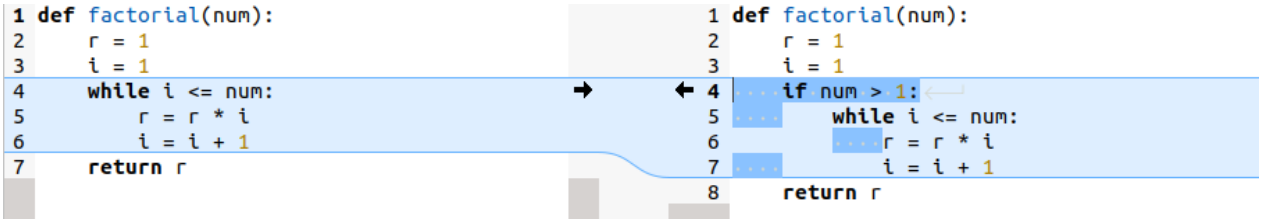

Рисунок 5 – Модифікація версії 2 (зліва початкова, справа кінцева)

Зміни, внесені для отримання кожної з цих розвинених версій, відображаються виділенням рядків, в яких відбулися зміни.

При двосторонньому поєднанні ми порівнюємо лише відмінності між версіями 1 та 2. Наприклад, у версії 1 перейменовано змінну n в num та додано документуючий рядок. З іншого боку, у версії 2 додано умовну конструкцію, якої немає у версії 1. Водночас, з інформації про окремі модифікації версій неможливо достовірно визначити чи відмінності, які спричинені додаванням рядка, видаленням або модифікацією, є лише в одній із розвинутих версій, чи одночасною модифікацією в обох версіях.

Покажемо, що тристороннє поєднання вирішує цю проблему. Рисунок 6 ілюструє тристороннє поєднання версій з попереднього прикладу.

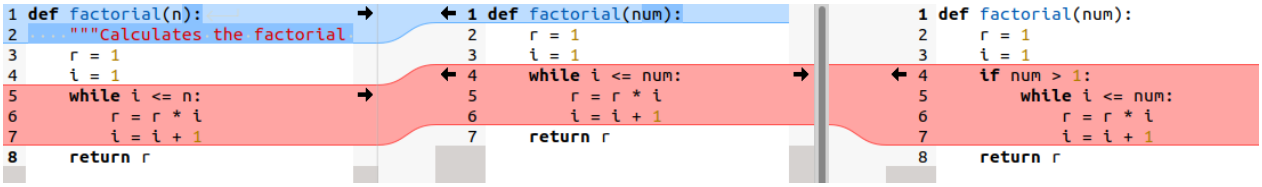

Рисунок 6 – Тристороннє поєднання версій 1 і 2 (зліва направо: модифікація 1, початкова версія, модифікація 2)

Наприклад, рядок “r = 1” версії 2 також присутній у початковій версії, що означає, що лише версія 1 внесла зміни до цього конкретного рядка. Ця додаткова інформація використовується алгоритмом поєднання, щоб визначити, які рядки версії 1 та версії 2 мають бути включені в об’єднану версію.

1.3 Типи поєднання за представленням програмного коду

Важливу відмінність між інструментами поєднання можна зробити на основі того, як представлені програмні продукти. Інструменти об’єднання на основі тексту розглядають програмне забезпечення як плоский текстовий файл. Кращою альтернативою є використання більш структурованої форми програмних продуктів (наприклад, дерево аналізу) або навіть розгляд семантичної інформації. Більш конкретно, можна розрізняти текстове, синтаксичне та семантичне поєднання.

1.3.1 Текстове поєднання

Інструменти поєднання на основі тексту розглядають програмні продукти лише як текстові файли (або двійкові файли). Найпоширенішим підходом є використання поєднання на основі рядків, де рядки тексту беруться як неподільні одиниці [7]. Прикладами цього є інструмент rcsmerge в Системі контролю версій (RCS) [6], інструмент Filemerge від Sun [1], система паралельних версій (CVS) [4] та інструменти поєднання, які можна знайти в комерційних інструментах керування конфігурацією.

Завдяки об’єднанню текстових файлів на основі рядків у паралельних модифікаціях можна виявити звичайні текстові рядки, а також рядки тексту, які були вставлені, видалені, змінені чи переміщені. На рис. 1 вже показаний приклад такого об’єднання на основі рядків. Однак через занадто грубу зернистість поєднання на основі ліній має той недолік, що він не може дуже добре обробляти дві паралельні модифікації одного рядка. Можна вибрати лише одну з двох модифікацій, але ці дві модифікації не можна об’єднати.

Незважаючи на свої недоліки, об’єднання на основі рядків залишається дуже корисним методом через його ефективність, масштабованість і точність. Згідно з деякими вимірюваннями, проведеними за допомогою тристороннього інструменту поєднання на основі рядків у промисловому прикладі [9], близько 90 відсотків змінених файлів можуть бути об’єдналися, не ставлячи жодних запитань. Більш складне завдання полягає в тому, щоб забезпечити автоматизовані способи вирішення тих 10 відсотків ситуацій, які не можна об’єднати автоматично. Причина, чому об’єднання на основі тексту не вдається в цих випадках пов’язано з тим, що текстові інструменти об’єднання не враховують ніякої синтаксичної чи семантичної інформації.

1.3.2 Синтаксичне поєднання

Синтаксичне поєднання [7] є потужнішим, ніж текстове, оскільки воно також враховує синтаксис програмних продуктів. Техніка поєднання на основі тексту часто знаходить неважливі конфлікти, такі як коментар до коду, який був паралельно змінений різними розробниками, або конфлікти, які виникають через введення деяких пробільних символів, які полегшують читання коду. Синтаксичне поєднання може ігнорувати всі ці конфлікти. Він видає конфлікт поєднання лише тоді, коли результат

об’єднання не є синтаксично правильним. Як конкретний приклад розглянемо умовну програму, "if (n mod 2)=0 then m:=n/2".

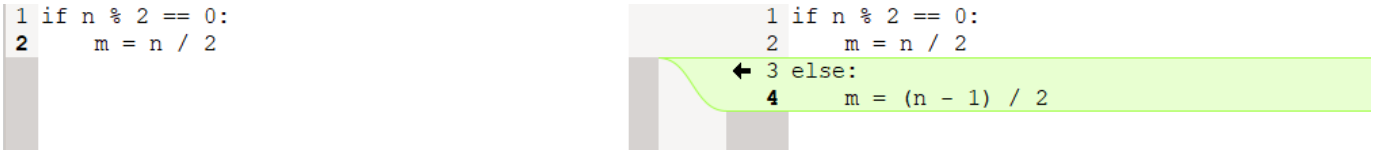

Рисунок 7 – Модифікація оператора новою гілкою виконання

Обидва розробники незалежно вирішують змінити цей фрагмент програми по-різному, але з тим же загальним ефектом. Перший розробник розширює фрагмент до "if (n mod 2)=0 then m := n/2 else m:=(n-1)/2" (рисунок 7), тоді як другий розробник модифікує його на "m:=n div 2" (де div це весь поділ).

Рисунок 8 - Заміна оператора іншим виразом

Хоча обидві зміни мають абсолютно однаковий ефект, чисто текстове поєднання, ймовірно, об’єднає ці дві модифікації у щось на зразок "m:=n div 2" else "m:=(n-1)/2", що синтаксично неправильно. Синтаксичне поєднання виявить цей конфлікт, щоб можна було вжити відповідних дій для вирішення проблеми. (У цьому випадку було б достатньо видалити частину else вручну).

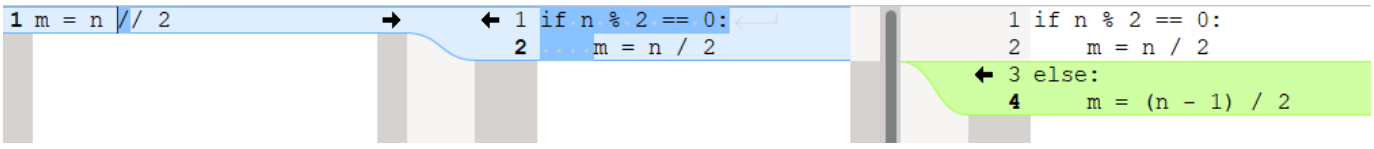

Рисунок 9 – Ситуація КП

Загальні прийоми синтаксичного поєднання можна розділити на категорії відповідно до їхньої базової структури даних: ті, які використовують (розбирають) дерева як базову структуру даних, і ті, які використовують графи як структуру даних.

Посилання [2], [3] та [7] є прикладами поєднання на основі дерева, які пропонують незалежний від мови тристоронній інструмент поєднання, який може виявляти синтаксичні конфлікти під час об'єднання дерев синтаксичного аналізу або абстрактних синтаксичних дерев. Cdiff [10], інший підхід поєднання на основі дерева, використовується для пошуку відмінностей між програмами на C++ шляхом порівняння їхніх дерев розбору. Цей підхід є обмеженим, оскільки він спирається на двостороннє поєднання і може використовуватися лише для об'єднання програм C++. Подібним інструментом, який дозволяє синтаксичним чином об'єднувати програми на C++, є TurboMixer, розроблений в ході проекту BeyondSniff [9].

Po і By [5] використовують атрибутивні графи як базове представлення програмних продуктів. Те ж саме стосується Менса [4], який додатково використовує методи переписування графів, щоб забезпечити формальну основу для об'єднання програмного забезпечення.

1.3.3 Семантичне поєднання

Інструменти синтаксичного поєднання не можуть виявити деякі часті конфлікти. Наприклад, під час поєднання версій 1a і 1b (рисунок 6) ми зіткнулися з проблемою в рядку 9 "if n>1:", де використовувалася змінна n, яка не оголошена в програмі. Це пояснюється тим, що кожна версія 1a і 1b використовує інше ім'я змінної (arg і n відповідно) для тієї ж мети. Синтаксичне поєднання не може виявити цей конфлікт, оскільки програма все ще синтаксично правильна. Тому ми назвемо це семантичним конфліктом. Точніше, це статичний семантичний конфлікт, оскільки

більшість компіляторів виявляють проблему і видають помилку «неоголошена змінна».

Підходи до поєднання на основі дерев, що базуються на деревах аналізу, не можуть виявити такого роду конфлікти, оскільки зв’язок між визначенням процедури (або функції) та викликом процедури (або функції) не є явним у дереві розбору чи абстрактному синтаксичному дереві. Підходи до поєднання на основі графів (такі як [4]) можуть виявити цей конфлікт, оскільки графічне представлення підтримує явний зв’язок між визначенням та його викликами, що дозволяє легко виявити несумісність між ними. Це також була ідея контекстно-залежного підходу Вестфетчель [6], де абстрактне синтаксичне дерево доповнюється контекстно-залежними відношеннями, які виражають прив’язки сутностей до їхніх декларацій.

Іноді навіть статичного семантичного поєднання недостатньо оскільки незалежні зміни паралельними розробниками все ще можуть призвести до несподіваної поведінки в об’єднаному результаті. Справді, немає жодних гарантій щодо того, як поведінка виконання об’єднаної програми пов’язана з поведінкою програм, що об’єднуються. Поведінкові конфлікти можна подолати, лише вдаючись до ще більш складних методів семантичного поєднання, які покладаються на семантику коду під час виконання. Більшість підходів до виявлення поведінкових конфліктів, як-от [7], [8], [14], [15], спираються на складні математичні формалізми, такі як денотаційна семантика, графи залежностей програми та зрізи графів програм. Існують і більш легкі підходи доступні, наприклад Semantic Diff [11], який використовує локальні графи залежності.

Покажемо приклад поєднання, яке призводить до поведінкового конфлікту через несподівану взаємодію об’єктно-орієнтованого успадкування з об’єднанням.

Клас Point (рисунок 10) містить дві змінні x і y, а також метод distance для обчислення евклідової відстані задана точка до початку координат.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self):
        return (self.x**2 + self.y**2)**0.5
```

Рисунок 10 – Початкова версія базового класу

Перший розробник програмного забезпечення вирішує додати новий підклас PolarPoint, який додатково реалізує метод radius, який реалізується шляхом виконання виклику метода distance (рисунок 11).

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self):
        return (self.x**2 + self.y**2)**0.5
```

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self):
        return (self.x**2 + self.y**2)**0.5


class PolarPoint(Point):
    def radius(self):
        return self.distance()
```

Рисунок 11 – Додавання успадкування

Другий розробник програмного забезпечення вирішує змінити реалізацію відстані.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self):
        return (self.x**2 + self.y**2)**0.5
```

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self):
        return abs(self.x) + abs(self.y)
```

Рисунок 12 – Модифікація метода базового класу

Через це, однак, реалізація радіуса стає недійсною при поєднанні цих двох модифікацій. Справді, радіус завжди слід обчислювати за допомогою евклідової відстані, що більше не так.

1.3.4 Структурне поєднання

Реструктуризація та рефакторинг вважаються основними методами підвищення працездатності програмних систем та усунення наслідків застаріння програмного забезпечення та ерозії програмного забезпечення [8], [12]. У цій мірі вводяться перетворення рефакторингу та реструктуризації, які мають особливу властивість зберігати поведінку, тобто не змінюють семантику програмного продукту, хоча його структура може істотно змінюватися. Конфлікти структурного поєднання виникають, коли одна із змін у програмному продукті є реструктуризацією, і алгоритм поєднання не може вирішити, яким чином має бути структурований результат об’єднання. Як приклад такого конфлікту розглянемо наступну ситуацію. Програмні продукти, використані в цьому прикладі, представляють об’єктно-орієнтовану ієрархію класів, що показано в нотації UMLTM (рисунок 13). Починаючи з абстрактного класу Shape з трьома підкласами Circle, Rectangle і Square, один модифікатор вирішує створити новий підклас Parallelogram (за допомогою операції CreateSubclass). Інший модифікатор самостійно вирішує реорганізувати підкласи Rectangle і Square, вводячи новий проміжний суперклас Quadrangle, який фіксує спільну поведінку обох (за допомогою операції InsertClass).

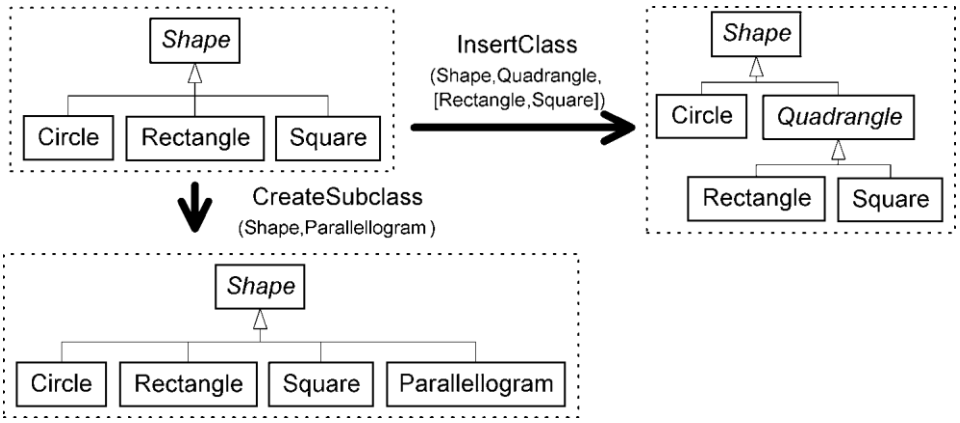

Рисунок 13 – Структурний конфлікт

При об’єднанні виникає питання, чи слід паралелограм зробити підкласом чотирикутника чи залишити його прямим підкласом форми? У випадку паралелограма ми повинні чітко вибрати першу альтернативу. Однак у випадку трикутника слід вибрати другий варіант. Отже, вхід від користувача необхідний для вирішення конфлікту.

На жаль, більшість існуючих інструментів і методів поєднання не забезпечують жодної підтримки для виявлення конфліктів структурного поєднання. Основна причина полягає в тому, що інформація, необхідна для виявлення цих конфліктів, зазвичай є неявною і не може бути виведена лише з вихідного коду.

1.3.5 Поєднання на основі стану, зміни або операції

Ще можна розрізняти об’єднання на основі стану та поєднання на основі змін. При об’єднанні на основі стану під час об’єднання враховується лише інформація в оригінальній версії та/або її редакціях. На відміну від цього, об’єднання на основі змін додатково використовує інформацію про точні зміни, які були виконані під час модифікації програмного забезпечення. Очевидно, що методи двостороннього поєднання завжди засновані на стані, оскільки вони можуть порівнювати лише дві редакції без урахування того, як ці редакції були отримані.

Поєднання на основі операцій – це особливий вид поєднання на основі змін, що моделює зміни як явні операції (або перетворення) [3]. Ці операції модифікації можуть бути як завгодно складними і зазвичай відповідають командам, що видаються в середовищі розробки програмного забезпечення. З цієї причини послідовності операцій зміни іноді називають історіями команд [4].

Підходи до поєднання на основі операцій можуть покращити виявлення конфліктів і забезпечити кращу підтримку при вирішенні цих конфліктів [5], [6], [9]. Приклад такої посиленої виразності показаний на прикладі рис. 1. Ми вже згадували про наявність семантичного конфлікту через те, що версія 1a вводить виклик процедури factorial десь в основній частині програми, тоді як версія 1b змінила factorial процедури на функцію. За допомогою інструмента керування версіями на основі операцій обидві зміни можуть бути виражені в термінах модифікаційних операцій:

1. rename_formal(factorial, num, n)
2. replace_statement(factorial, 3, if_statement(“n > 0”, get_statement(factorial, 3)))

Щоб виявити конфлікт, нам просто потрібно порівняти обидві операції, щоб побачити, що вони впливають на один і той самий об’єкт сутності несумісним чином.

Іншими словами, щоб виявити конфлікти поєднання, ми це робимо не потрібно повністю порівнювати паралельні редакції, оскільки достатньо порівняти лише операції модифікації, які були застосовані для отримання кожної з версій. Для певних комбінацій операцій повідомлятимуться про конфлікти. Хоча використовувані операції, повідомлення про конфлікти та запропоновані стратегії вирішення можуть відрізнятися від одного підходу до іншого, основна ідея, здається, загалом узгоджена [4].

Поєднання на основі операцій є загальним у тому сенсі, що його можна використовувати для виявлення синтаксичних, структурних і навіть деяких видів семантичних конфліктів. Наприклад, Едвардс [14] використовує підхід, заснований на

операціях, для виявлення та вирішення семантичних конфліктів, які виникають виключно через семантику, надану програмою. Хоча лише самі програми «знають», як їх поведінка може викликати конфлікти, ці знання фіксуються шляхом визначення конкретних операцій, які можуть виконуватися в кожній програмі, і як ці операції можуть викликати конфлікти.

Підхід, заснований на операції, полегшує реалізацію механізму багаторазового скасування/повторення. Для скасування достатньо виконати останні застосовані операції в протилежному напрямку, а для повторного виконання ми просто повторно застосовуємо операції. У фреймворку спільної програми GINA [3] об’єднання історій команд досягається шляхом вибіркового використання механізму повторення для застосування змін одного розробника до версії іншого. Цей підхід не дуже ефективний у випадку довгої історії команд або коли деталізація історії занадто обширна.

1.4 Виявлення конфліктів

У розділі 2 виділено три види невідповідностей, які можуть виникнути під час поєднання: синтаксичні, структурні та семантичні конфлікти. Більше того, семантичні конфлікти можна додатково поділяти залежно від того, чи можна їх виявити під час компіляції чи під час виконання. У цьому розділі розглядаються кілька питання, пов’язані з цими конфліктами, більш детально. Спочатку ми розглянемо різні підходи, які можна використовувати для виявлення конфліктів поєднання. Далі ми обговорюємо деякі методи вирішення конфліктів поєднання. Нарешті, ми звертаємось до деяких евристик, щоб спробувати зменшити (іноді дуже велику) кількість виявлених конфліктів.

У цьому розділі ми обговорюємо деякі цікаві методи виявлення конфліктів і дізнаємося, як і які типи конфліктів вони можуть виявляти.

1.4.1 Поєднання матриць

Багато інструментів поєднання використовують спеціальний підхід для виявлення конфліктів поєднання. За допомогою підходу до поєднання, заснованого на операціях, конфлікти можна виявляти більш дисципліновано, порівнюючи операції модифікації, які паралельно застосовувалися різними розробниками. Усі пари операцій, які призводять до невідповідності, підсумовуються в так звану таблицю конфліктів або матрицю поєднання. Це дає змогу виявляти конфлікти поєднання шляхом простого пошуку таблиці.

Дещо більш загальний підхід використовує Feather [7]. Замість того, щоб зберігати операції безпосередньо в матриці поєднання, вони аналізуються, щоб визначити, які зміни властивостей специфікації вони викликають, і замість цього зберігається ця інформація. Потім можна виявити конфлікти, порівнявши всі можливі зміни властивостей специфікації. Це полегшує впровадження нових операцій модифікації. Потрібно лише визначити, як нововведені операції можна розкласти з точки зору змін властивостей специфікації.

Мансон і Деван [14] пропонують структуру поєднання, яка повністю базується на ідеї матриць поєднання. У цій структурі матриці не тільки вказують конфлікти, які можуть виникнути, а й дії, які слід виконати для вирішення конфліктів. Матриці поєднання послідовно використовуються на кожному рівні деталізації (тобто, вони мають справу з операціями як над атомарними, так і над складеними програмними продуктами) і для кожного типу програмного продукту, який розглядається. Можна використовувати багато різних матриць поєднання, залежно від таких факторів, як тип користувача, який виконує поєднання, час, коли були виконані операції, або будь-який інший фактор, який вважається релевантним для керування процесом поєднання.

Інший спосіб узагальнення підходу матриці поєднання досліджується у незалежному від домену формалізмі Mens [4], де програмні продукти представлені у

вигляді графів, модифікація представлена переписуванням графів, а графи типів використовуються для визначення обмежень, специфічних для домену. яким мають задовольняти всі графи в певній області. Спираючись на формальні властивості переписування графів (точніше, поняття паралельної залежності), можна формально визначити, які пари операцій модифікації (формально представлені як створення графів) призводять до конфлікту синтаксичного поєднання. Це призводить до формальної і незалежної від області характеристики інтуїтивного поняття матриці поєднання.

Основна проблема з матрицями поєднання виникає, коли ми хочемо об'єднати більше двох версій. Наприклад, якщо є M видів операцій і існує N версій, які потрібно об'єднати, ми отримаємо N-вимірну матрицю поєднання, яка вимагає поліноміальної складності простору *O(MN)*.

1.4.2 Набори конфліктів

Як альтернатива матрицям поєднання, Едвардс [14] використовує набори конфліктів у контексті спільних програм для групування потенційно конфліктних комбінацій операцій на основі семантики, наданої програмою. Залежно від типу програми (наприклад, багатокористувацьке середовище розробки програмного забезпечення, спільний редактор креслень, програма розподіленого компонування офісних меблів), види операцій і пов'язані з ними конфлікти поєднання можуть суттєво відрізнятися. Набори конфліктів відповідають типам конфліктів, які можуть існувати в програмі. Як такі, вони статично визначені в тому сенсі, що вони залишаються фіксованими, доки семантика програми не змінюється. Наприклад, у програмі розподіленого макета можна розрізняти тимчасові та просторові конфлікти. Операції, які належать до одного конфліктного набору, потенційно можуть викликати конфлікти при об'єднанні. Будь-яка дана операція може одночасно брати участь у

кількох наборах конфліктів. Конфлікт встановлює масштабованість адрес, оскільки вони обмежують кількість операцій, які ми повинні враховувати під час пошуку конфліктів.

1.4.3 Методи виявлення семантичних конфліктів

Горвітц [14] був першими, хто розробив потужний алгоритм для поєднання (або «інтеграції») версій програм без семантичних конфліктів, заснований на семантиці дуже простої мови програмування на основі присвоєння. Алгоритм поєднання використовує базове представлення графів залежностей програми [6] і використовує поняття зрізу програми [5] для пошуку потенційних конфліктів поєднання. Незважаючи на свою потужність, алгоритм має ряд обмежень. Наприклад, якщо одна версія програми змінює спосіб виконання обчислень без зміни зовнішньої поведінки (тобто, введення/виводу), тоді як інший варіант додає код, який використовує лише результати обчислення, виникне конфлікт. Янг та ін. [65] долають це обмеження, вводячи семантико-зберігають трансформації. Binkley та ін. [8] пропонують інше розширення для обробки програм, які містять процедури (які можуть бути взаємно рекурсивними). Хоча всі ці підходи дозволяють нам виявляти поведінкові конфлікти, вони мають важливий недолік, що вони залишаються обмеженими певною мовою реалізації.

Берзіньш [7] використовує більш загальний підхід, надаючи незалежне від мови визначення семантичного поєднання. Таким чином, узагальнення традиційної денотаційної семантики [5] використовується для забезпечення додаткової структури, необхідної для формального визначення конфліктів семантичного поєднання. Більш конкретно, використовуються алгебри Браувера, які є узагальненням булевих алгебр [8]. Реалістичним прикладом семантичної області, яку необхідно моделювати за

допомогою алгебр Браувера, є PSDL, мова з жорсткими обмеженнями реального часу. Dampier та ін. [13] описують метод об’єднання змін у цій конкретній мові.

Основним недоліком підходу Берзіньша є те, що він безпосередньо працює на основну семантичну інтерпретацію моделі, тому її не можна використовувати для діагностики та визначення місця розташування конфлікти між змінами в конкретному синтаксичному представленні програми. Це робить цей підхід непрактичним, оскільки його не можна використовувати для визначення фактичного джерела семантичного конфлікту в програмному забезпеченні. Берзіньш [6] певною мірою вирішує цю проблему, обмежуючи загальний формалізм лише деякими окремими випадками.

Інший метод, який видається більш практичним у тому сенсі, що його можна застосувати до реальних мов програмування, таких як C, використовується Semantic Diff [7]. Цей інструмент двостороннього поєднання визначає відмінності між двома версіями процедури за допомогою локальних графів залежності для порівняння їх поведінки введення-виводу, що спостерігається. Цей підхід є повністю автоматичним, швидким і простим у використанні, але обмежений у тому сенсі, що він може знайти лише локальні семантичні відмінності, оскільки він лише порівнює процедури одна з одною. Отже, він не може виявити більш глобальні міжпроцедурні семантичні конфлікти. Хоча цей підхід був проілюстрований для програм на C, він, здається, можна узагальнити і на інші мови програмування. Однак наявність пізнього зв’язування та поліморфізму в об’єктно-орієнтованих програмах робить застосування цього підходу в об’єктно-орієнтованих програмах далеко не тривіальним.

Mens [3] представив альтернативний легкий підхід для виявлення конфліктів семантичного поєднання (рис. 1). Конфлікт непослідовних методів виник через те, що неявне припущення, що addAll завжди викликає add, було порушено. Цей конфлікт можна виявити, переглянувши лише результат об’єднання, якщо ми задокументуємо, які зміни були внесені якими розробниками. У цьому конкретному випадку розробник

1 додав виклик методу add до розміру атрибута, тоді як розробник 2 видалив виклик методу addAll до методу add. Якщо ми використовуємо графи для представлення програмних продуктів, цю конфліктну ситуацію можна виявити як екземпляр більш загального шаблону графа (де ми ігноруємо назви сутностей, які видаляються). Якщо екземпляр цього шаблону графа можна знайти в результаті поєднання, повідомляється про неузгоджений конфлікт методів. Подібним чином інші семантичні конфлікти, такі як непередбачувана рекурсія, можуть бути виявлені шляхом пошуку екземплярів інших попередньо визначених шаблонів графів. Одним з недоліків цього підходу є те, що складні поведінкові конфлікти не завжди можуть бути виявлені. Іншим недоліком є те, що пошук підграфа в графі може бути дуже дорогою операцією, тому підхід не можна масштабувати до великих програмних систем.

Висновки до розділу 1

Важливу відмінність між інструментами поєднання можна зробити на основі того, як представлені програмні продукти. Інструменти об’єднання на основі тексту розглядають програмне забезпечення як плоский текстовий файл. Кращою альтернативою є використання більш структурованої форми програмних продуктів (наприклад, дерево аналізу) або навіть розгляд семантичної інформації. Більш конкретно, можна розрізняти текстове, синтаксичне та семантичне поєднання.

2 ГРАФ ЗАЛЕЖНОСТЕЙ ПРОГРАМИ ЯК ПРЕДСТАВЛЕННЯ СЕМАНТИКИ ПРОГРАМИ

У цьому розділі наведені основний принцип роботи використаних алгоритмів для вирішення поставленої задачі семантичного поєднання програм на підмножині мови Python.

2.1 Граф залежностей програми

Граф залежностей програми (англ. “program dependency graph”) для програми P, що позначається G(P), є орієнтованим графом, вершини якого з’єднані кількома видами ребер [11]. Вершини GP представляють оператори присвоєння та конструкції керування, які зустрічаються в програмі P. Крім того, GP включає три інші категорії вершин:

1. Існує виділена вершина, яка називається вершиною входу.
2. Для кожної змінної x, для якої є шлях у стандартному графу потоку керування для P, на якому знаходиться x використовується до визначення (див. [1]), існує вершина, яка називається початковим визначенням x. Ця вершина представляє присвоєння x початкового значення. Вершина позначена як «x := initial_state(x)».
3. Для кожної змінної x, названої в кінцевому операторі P, є вершина, яка називається кінцевим використанням x. Він представляє доступ до остаточного значення x, обчисленого P, і має мітку «final_use(x)».

Ребра GP представляють залежності між компонентами програми. Ребро представляє або керуючу залежність, або залежність від даних.

2.1.1 Залежність контролю

Ребра керуючої залежності позначаються як true, так і false, а початком ребра залежності контролю завжди є вершина входу або вершина-предикат. Ребро керуючої залежності від вершини v1 до вершини v2 , позначене як v1 →cv2 , означає, що під час виконання, коли оцінюється предикат v1, і його значення збігається з міткою на ребрі до v2 , тоді компонент програми, представлений v2, зрештою буде виконано, якщо програма завершиться. Спосіб визначення ребер залежності керування для довільних програм наведено в [7]; однак, оскільки ми припускаємо, що програми включають лише оператори присвоєння, умовні та while, ребра залежності керування GP можна визначити набагато простіше. Для мови Python, залежності керування відображають структуру вкладеності програми. Граф GP містить ребро керуючої залежності від вершини v1 до вершини v2 GP, якщо виконується одна з наступних умов:

- v1 є вершиною входу, а v2 представляє компонент P, який не вкладений у жодний цикл або умовне; ці ребра позначені як true.
- v1 представляє керуючий предикат, а v2 являє собою компонент P, безпосередньо вкладений у цикл або умовну структуру, предикат якого представлений v1. Якщо v1 є предикатом циклу while, то →c v2 позначено як істинно; якщо v1 є предикатом умовного оператора, ребро v1 →c v2 позначається як true чи false відповідно до того, чи зустрічається v2 у гілці then чи else, відповідно.

2.1.2 Залежність даних

Ребро залежності даних від вершини v1 до вершини v2 означає, що обчислення програми можуть бути змінені, якщо відносний порядок компонентів, представлених v1 і v2, був протилежним. Графи залежностей програми містять два типи ребер

залежності від даних, що представляють залежності потоку та залежності від порядку. Ребра залежності від даних ГЗП обчислюються за допомогою аналізу потоку даних. Для обмеженої мови, що розглядається в цьому розділі, необхідні обчислення можуть бути визначені синтаксично.

Граф залежностей програми містить ребро залежності потоку від вершини v1 до вершини v2, якщо виконується все з наступного:

1) v1 є вершиною, яка визначає змінну x.
2) v2 є вершиною, яка використовує x.
3) Керування може досягати v2 після v1 через шлях виконання, вздовж якого немає проміжного визначення x. Тобто в стандартному графі потоку керування для програми є шлях, за допомогою якого визначення x у v1 досягає використання x у v2. (Вважається, що початкові визначення змінних відбуваються на початку графа потоку керування; вважається, що кінцеве використання змінних відбувається наприкінці графа потоку керування).

2.1.3 Залежність потоку

Залежність потоку, яка існує від вершини v1 до вершини v2, буде позначатися $v1 \rightarrow_f v2$.

Залежності від потоку можна далі класифікувати як циклові (англ. “loop-carried”) та цикло-незалежні (англ. “loop-independent”). Залежність від потоку $v1 \rightarrow_f v2$ є цикловою з циклом L (позначається $v1 \rightarrow_{lc(L)} v2$), якщо на додаток до критеріїв залежності потоку також виконується наступне:

- Існує шлях виконання, який задовольняє умовам (3) вище, і включає ребра повернення до предиката циклу L.
- v1 і v2 укладені в цикл L.

Залежність потоку v1 →f v2 є цикло-незалежною тоді (позначається v1 → li v2), якщо на додаток до (1), (2) і (3) вище, існує шлях виконання, який задовольняє (3) вище і не містить ребра повернення до предиката циклу L, який охоплює v1 і v2. Можна мати як v1 → lc (L) v2, так і v1 → li v2.

2.1.4 Залежність порядку визначення

Граф залежностей програми містить ребро залежності визначення порядку від вершини v1 до вершини v2, якщо все з наступного виконується:

- v1 і v2 визначають ту саму змінну.
- v1 і v2 знаходяться в одній гілці будь-якого умовного оператора, який охоплює їх обидва.
- Існує програмний компонент v3 такий, що v1 →f v3 і v2 →f v3 .
- v1 зустрічається зліва від v2 в абстрактному синтаксичному дереві програми.

Залежність порядку від v1 до v2 із «свідком» v3 позначається v1 → do (v3) v2 .

Зауважимо, що граф залежностей програми є мультиграфом (тобто він може мати більше одного ребра певного виду між двома вершинами). Якщо між двома вершинами є більш ніж одне ребро циклової залежності потоку, то кожна позначається різним циклом. Якщо між двома вершинами є більше одного ребра з визначенням порядку, кожне з них позначається вершиною, яка залежить як від визначення, яке відбувається в джерелі ребра, так і від визначення, яке відбувається в цілі ребра.

```
def main():
    sum = 0
    i = 1
    while i < 11:
        sum = sum + i
        i = i + 1
    return (sum, i)
```

Рисунок 14 – Приклад програми

Далі показано приклад програми (рисунок 14) та граф залежностей цієї програми (рисунок 15).

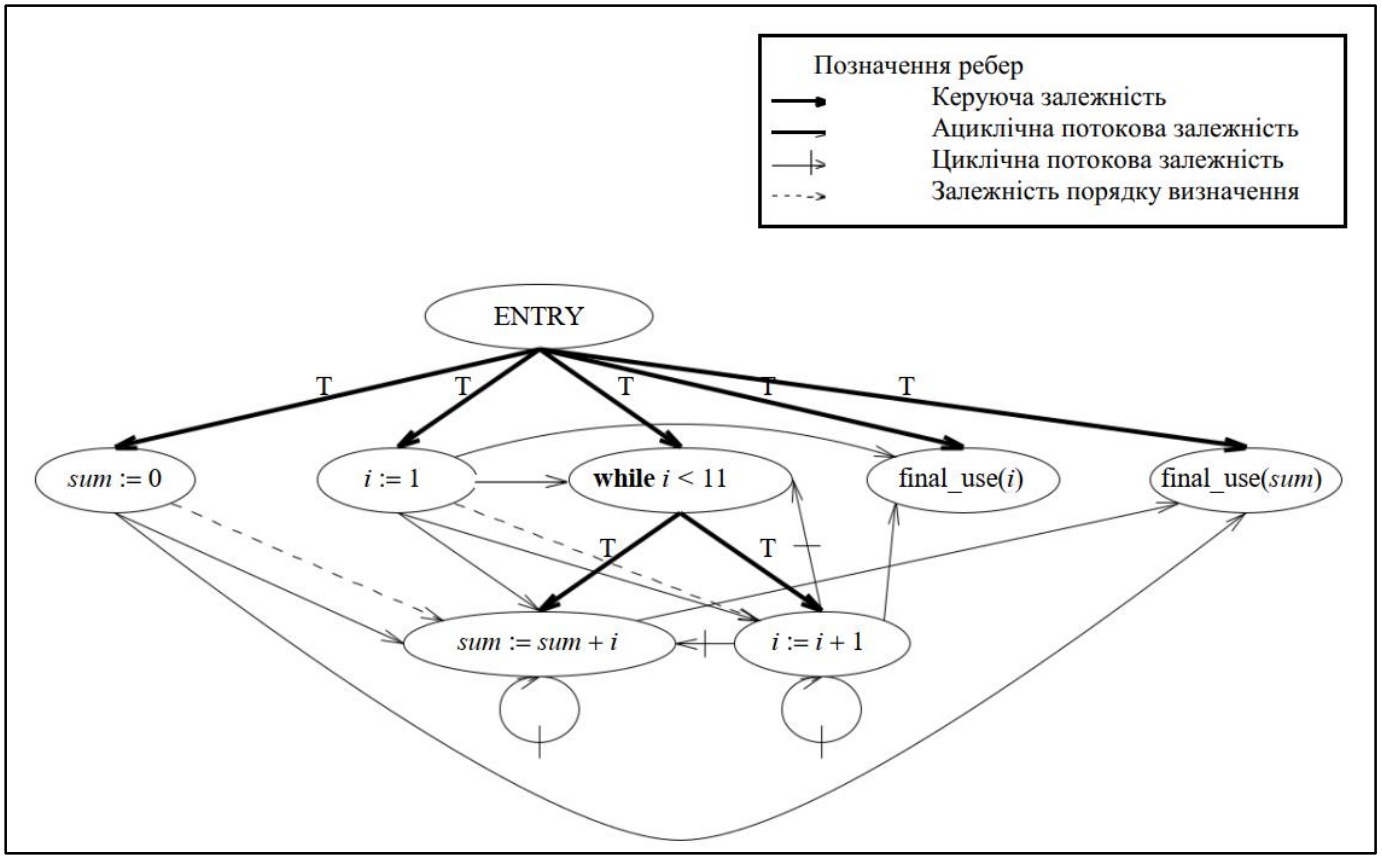

Рисунок 15 – Граф залежностей програми

Потовщені стрілки представляють ребро залежності контролю; суцільні стрілки представляють ребра залежності потоку, що не залежать від петлі; суцільні стрілки з хеш-позначкою представляють ребра залежності потоку, що переноситься по петлі; пунктирні стрілки представляють ребра залежності від порядку.

2.2 Внутрішньопроцедурний зріз

Для вершин s графа програмної залежності G, зріз G щодо s, позначений G / s, є графом, що містить усі вершини, від яких s має транзитивний потік або керуючу залежність (тобто всі вершини, які можуть досягати s через потік та/або контрольні ребра). Псевдокод алгоритму списку обчислення вершин зрізу ГЗП.

Нехай

- *G*: ГЗП
- *S*: множина вершин *G*
- *WorkList* множина вершин *G*
- *v*, *w*: вершина в *G*

Початок.

1. WorkList := S
2. Поки WorkList ≠ ∅

2.1.Вибрати і видалити v із WorkList

2.2.Помітити v

2.3.Для кожної вершини w, що містить ребро $w \rightarrow_f v$ або $w \rightarrow_c v$ з E(G)

2.3.1. Додати w в WorkList

Кінець.

Зв’язок між ГЗП та його фрагментом розглянуто в статті [20]. Ми говоримо, що G є можливим ГЗП, якщо G є графом програмної залежності деякої програми P.

```
def main
    i = 1;
    while i < 11:
        i = i + 1
    return i
```

Рисунок 16 – Програма, що складається з операторів зрізу

Для будь-якого S ⊆ V(G), якщо G є можливим ГЗП, зріз G / S також є можливою ГЗП; він відповідає програмі P′ (рисунок 16), отриманій шляхом обмеження синтаксичного дерева P лише операторами та предикатами у V(G / S) [20]. Рисунок 17 ілюструє граф, який є результатом взяття зрізу графа залежності програми (рисунок 14) щодо вершини кінцевого використання для і разом з однією програмою, якій він відповідає.

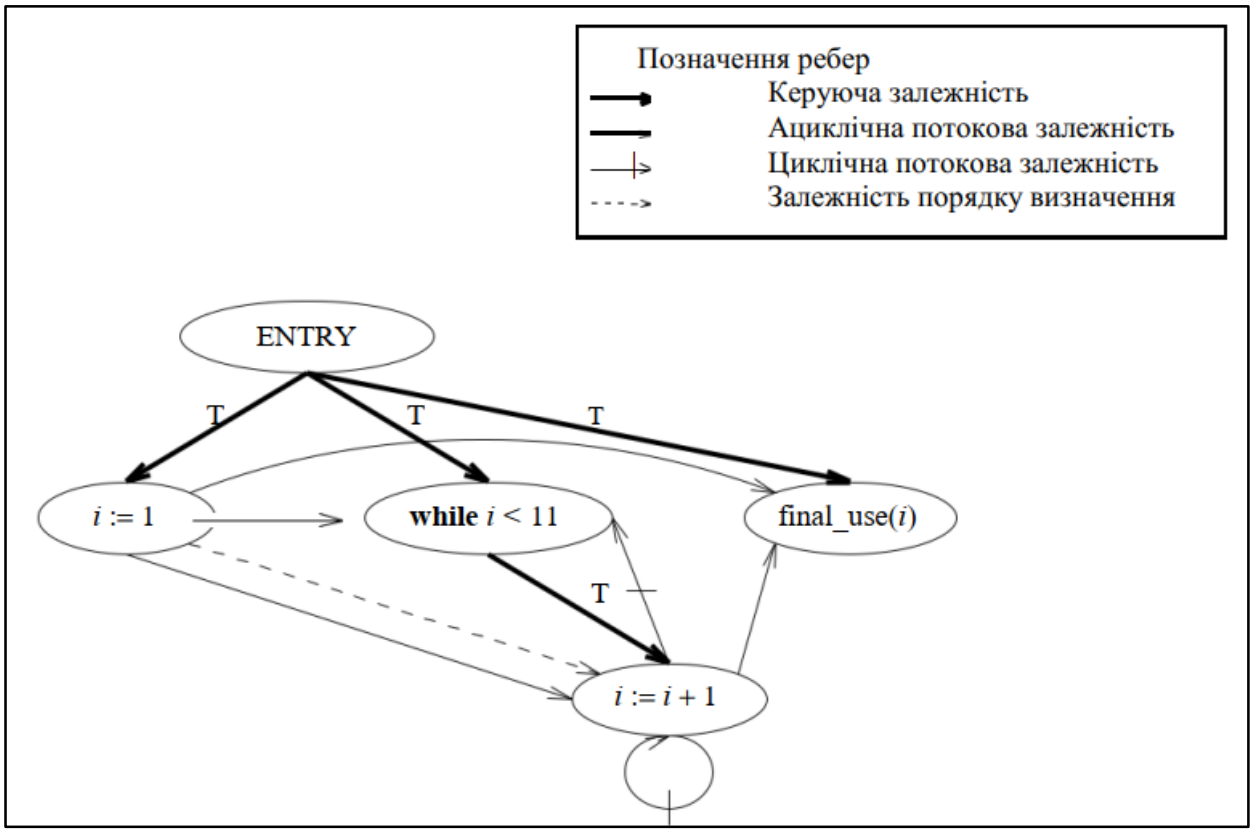

Рисунок 17 – Приклад зрізу ГЗП

Значення внутрішньопроцедурного зрізу полягає в тому, що він фіксує частину поведінки програми в тому сенсі, що для будь-якого початкового стану, на якому програма зупиняється, програма і зріз обчислюють однакову послідовність значень для кожного елемента зрізу [20]. У нашому випадку точка програми може бути:

- оператором присвоєння;
- оператором керування;
- кінцевим використанням змінної в кінцевому операторі.

Оскільки предикат оператора чи управління може бути досягнуто багаторазово в програмі, під «обчислюванням однієї і тієї ж послідовності значень для кожного елемента зрізу» ми маємо на увазі:

- для будь-якого оператора присвоєння однакова послідовність значень призначається до цільова змінна;
- для предиката така ж послідовність логічних значень вироблений;
- для кожного кінцевого використання створюється однакове значення змінної.

2.3 Циклічний метод визначення еквівалентності програм

У даній роботі використовується система для доведення еквівалентності програм на детермінованих мовах, які забезпечені операційною семантикою, визначеною переписуванням. Так звані символічні програми, в яких певні вирази чи інструкції залишаються неінтерпретованими, також можна розглядати за допомогою цього підходу. Запропонована дедуктивна система є круговою; ми показуємо, що він правильний і слабо повний. Ці два результати означають, що наша система правильно вирішує проблему еквівалентності програм, як ми її поставили. Ми показуємо, що ця система працює як для програм, які завершуються, так і для програм, які не завершуються. Нарешті, ми описуємо прототип реалізації нашої дедуктивної системи в структурі K.

Програми можуть належати до будь-якої детермінованої мови, семантика якої визначається набором правил перезапису. Еквівалентність дозволяє узгоджувати декілька інструкцій однієї програми з кількома інструкціями іншої. Дедуктивна система є круговою: її висновки можна повторно використовувати як гіпотези контрольованим способом. Він не гарантовано завершиться, але коли він завершується, він правильно вирішує проблему еквівалентності програми завдяки своїй надійності та слабким властивостям повноти. Ці властивості неформально представлені нижче та формалізовані в роботі.

Цей метод також підходить для доведення еквівалентності символічних програм. Це програми, в яких деякі вирази та/або оператори є символьними змінними, які позначають набори конкретних програм, отриманих шляхом заміни символічних змінних конкретними виразами та/або операторами.

Приклад 1. Наступні програми (іменуються for та while) є символічними (рисунок 18):

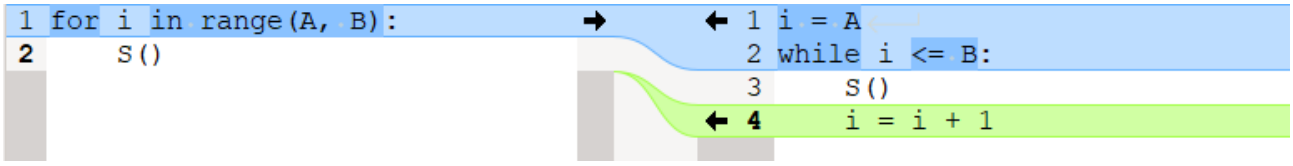

Рисунок 18 – Приклад для визначення еквівалентності програм

Їх символічні змінні i, A, B, S може відповідати, відповідно, будь-який ідентифікатор (i), арифметичний вираз (A, B) і виклик функції (S()), виконання якої завершується.

Типове використання системи еквівалентності програм полягає в наступному:

1) визначення операційної семантики мови програмування L. Зауважимо, що цей підхід працює лише для детермінованих мов;
2) розширення семантики L на символічні висловлювання та вирази;

3) запуск дедуктивної системи над розширеною семантикою, щоб перевірити еквівалентність програм у L.

Запуск дедуктивної системи по суті означає символічне виконання семантики мови. Це призводить до одного з трьох можливих результатів:

- успішне завершення, і в цьому випадку програми, надані як вхід до дедуктивної системи, є еквівалентними через надійність дедуктивної системи;
- завершення із збоєм, у цьому випадку програми, що надаються як вхід до дедуктивної системи, не є еквівалентними через слабку повноту системи;
- нескінченне виконання – у цьому випадку не можна зробити висновку про еквівалентність.

Можливість неприпинення притаманне будь-якій автоматичній системі доведення еквівалентності програми, оскільки проблема еквівалентності нерозв’язна. Проте, ця система завершується, коли програми, які їй надаються як вхідні дані, завершуються, а також коли вони не завершуються, але поводять себе певним допустимим чином.

Виконання семантики полягає в перетворенні конфігурацій, які є парами, побудованими з:

- операцій, які залишилося виконати, і
- відображення змінних у значення, що позначають поточний стан.

Наприклад, починаючи з конфігурації (рисунок 19).

```
1 for i in range(A, B):
2     S()
```

Рисунок 19 – Приклад до визначення семантичного ефекту

після кількох семантичних кроків досягає конфігурації hP_1; $state_1(i)$, де або

- P_1 – порожня програма;
- $state_1$ – це ефект i = A на стан, і $state_1(i)$ на $state_1(B)$

або

- P_1 це цикл з лічильником і від (A + 1) до B, що викликає S();
- $state_1$ – це ефект (i =A; S()) на стан, і $state_1(i)$ state1(B).

Аналогічно, починаючи з hi = A ; тоді як i <= B do { S ; i = i + 1 }; statei, після кількох семантичних кроків, досягається конфігурація hP_2; $state_2(i)$, де або

- P_2 – це порожня програма, і
- $state_2$ – це дія i = A на state, а $state_2$ (i) > $state_2$ (B)

або

- P_2 дорівнює i = A + 1 ; тоді як i <= B do { S ; i = i + 1 }
- state 2 – це дія послідовності i =A; S на state, і state2(i) state2(B).

Тобто після деяких перетворень вихідних конфігурацій можна отримати або:

- рівні конфігурації, з порожніми програмами (зліва);
- конфігурації з непустими програмами (у правій частині), які в певному сенсі визначені в статті, екземпляри вихідних конфігурацій.

На цьому етапі наша дедуктивна система завершує роботу і повідомляє, що дві початкові конфігурації еквівалентні. Це пояснюється тим, що система визначає, що програми можуть поводити себе лише подібним чином у майбутньому, назавжди відтворюючи поведінку, проявлену після перших кількох семантичних кроків. Таким чином, навіть незважаючи на те, що символічні виконання не завершуються – оскільки цикли for та while не мають скінченних меж – система доведення завершується тут успішно.

2.4 Алгоритм визначення семантичного ефекту програми

Наведемо особливості практичного інструменту семантичного розрізнення. По-перше, набір термінів його звіту повинен бути семантичним, а не синтаксичним. Семантичний ефект функції – це відношення між її входами та виходами; Таким чином, різниця між двома версіями має бути виражена в термінах поведінки введення-виведення. Це не означає заперечення цінності синтаксичних звітів. Фрагмент, що вказує на зачеплені рядки коду, підходить для деяких програм, але не повідомляє супроводжувачу, як це вплине на клієнтів процедури; це лише зменшує розмір програми, з якої ця інформація має бути виведена.

По-друге, інструмент повинен бути повністю автоматичним, швидким і простим у використанні. Зрозуміло, що це виключає будь-які види доведення теорем (які вимагають надання лем користувачем і пошук, який може не завершитися). Але це також бореться з глобальними аналізами, такими як міжпроцедурний потік даних. Програма, яку ми аналізували, містить процедури, які в середньому мають кілька сотень рядків; Щоб знайти відмінності між двома версіями такої процедури, нашому інструменту (дослідницький прототип, закодований на ML) потрібно близько десяти секунд. Проте вся програма містить понад мільйон рядків коду. Компіляція програм такого розміру може зайняти кілька днів.

По-третє, підсумок має бути точним. Визначення точних семантичних відмінностей між двома версіями програми є невирішеною проблемою, тому жоден автоматичний інструмент не може бути абсолютно точним, звичайно. Отже, інструмент семантичного відмінності, на відміну від текстового, може робити помилки: пропускати справжні відмінності або знаходити помилкові відмінності там, де їх насправді немає.

Аналіз програм, розроблений для компіляторів, не може терпіти помилок, які ставлять під загрозу правильність коду, тому вони завжди точні. Існуючі методи, які порівнюють версії програми, виросли з роботи з оптимізації компілятора, і приймають

це припущення, завжди помиляючись на стороні помилкових відмінностей, але ніколи не упускаючи реальної відмінності. Наприклад, методи зниження вартості регресійного тестування шляхом усунення непотрібних тестів або не повторного тестування коду, що не змінено, можуть безпечно знайти відмінності там, де їх немає, оскільки це призводить лише до непотрібних виконання тестів, але ніколи не повинні пропускати відмінності, що може призвести до неадекватного тестування.

Коли вихід інструменту призначений для використання людиною, може знадобитися інший тип точності. Завжди помилятися в одному напрямку, переоцінюючи відмінності, може бути катастрофічним, якщо користувач переповнений неправдивими повідомленнями. І випадкове пропуск різниці є менш критичним; Людина-користувач менш точна, ніж компілятор, і може використовувати результати, які не є повністю надійними.

З цієї причини зроблено радикальний крок, намагаючись максимізувати точність результату нашого інструменту, порушивши його надійність. Інструмент, заснований на стандартному підході до порівняння, розглядає дві версії як ідентичні лише тоді, коли їхні графи залежності є ізоморфними [2]. Це вимагає, щоб вузли графів збігалися синтаксично; єдина семантична абстракція полягає у відносному впорядкуванні сентенцій. У результаті майже будь-яка зміна коду буде позначена як відмінність, включаючи всі звичайні трансформації, що зберігають сенс, які часто застосовуються програмістами під час обслуговування: перейменування локальних змінних, додавання або видалення тимчасових змінних для підвиразів, пакування змінних. у структуру запису тощо.

Висновки до розділу 2

Циклічний метод визначення еквівалентності програм використовується як система для доведення еквівалентності програм на детермінованих мовах, які забезпечені операційною семантикою. Зауважимо, що для системи повної за Тюрингом, – це алгоритмічно нерозв’язна задача. Але якщо внести певні обмеження на функціональну повноту, то можна застосувати циклічний метод для визначення еквівалентності програм.

Алгоритм визначення семантичного ефекту програми використовується для визначення семантичних ефектів процедур. Семантичний ефект функції – це відношення між її входами та виходами; Таким чином, різниця між двома версіями має бути виражена в термінах поведінки введення-виведення.

Враховуючи наведений в даному розділі аналіз циклічного методу визначення еквівалентності програм та алгоритму визначення семантичного ефекту програми, можемо описати повноцінний семантичний спосіб поєднання програм, написаних на підмножині мови Python.

3 СПОСІБ СЕМАНТИЧНОГО ПОЄДНАННЯ ПРОГРАМ

У даній роботі пропонується семантичний спосіб поєднання версій програм на підмножині мови Python. В даному випадку вказано найвищий рівень представлення програмного коду – семантичний. В той же час, семантичне представлення програмного коду є надбудовою над представленнями нижчого рівня – синтаксичного. В даному розділі описуються створені складові способу поєднання за рівнями представлення програмного коду.

3.1 Лексичний рівень

В даному підрозділі наводиться опис способу поєднання відносно лексичного ППК.

3.1.1 Обробка коментарів

Опишемо спосіб поєднання коментарів у випадку виникнення КП. Під час ЛА, текст коментаря перетворюється в рядок символів. Цей коментар додається до термінала, відносно початку або кінця.

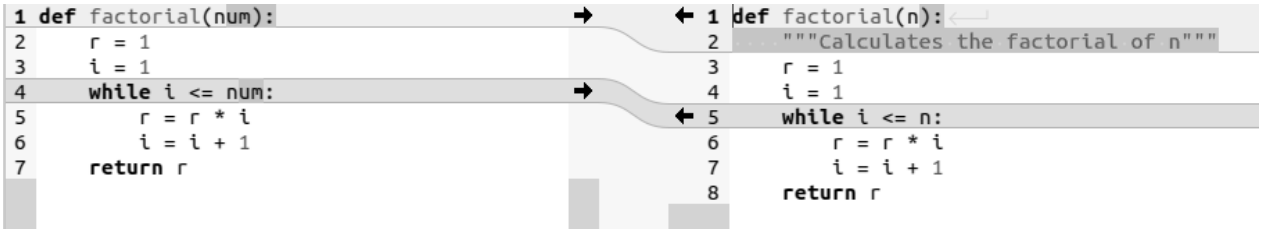

Рисунок 20 – Приклад обробки коментаря

Таким чином, коментарі зберігаються під час лексичного аналізу програмного коду.

3.1.2 Обробка пробільних символів

Пробільні символи зберігаються разом із сусідніми лексемами. Якщо вони беруть участь в конфлікті поєднання, перевагу має така конфігурація пробільних символів, яка відповідає конвенції із проставлення пробільних символів, описаної в стандартизації мови Python.

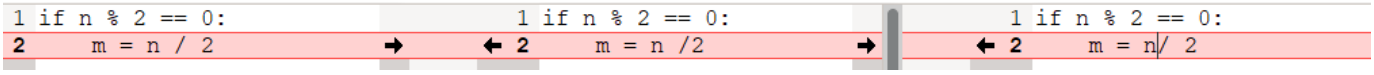

Рисунок 21 – Приклад обробки пробільних символів

Таким чином, навіть неважливі для синтаксису мови Python, пробільні символи не ігноруються і зберігаються на стадії лексичного аналізу.

3.2 Синтаксичний рівень

Щоб проілюструвати проблему вирішення синтаксичних конфліктів, ми використовуємо запущений приклад простої реалізації стека (рисунок 22). Відтепер ми називаємо цю програму базовою програмою або просто Stack. Вона містить клас Stack, який містить елементи поля та два методи push і pop.

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if len(self.items) > 0:
            return self.items.pop(0)
        return None
```

Рисунок 22 – Клас Stack

Тепер припустимо, що програміст хоче додати нову функцію Top, але хоче розробити функцію у власній гілці, незалежно від основної гілки (тобто базової програми). З цією метою програміст створює гілку з новою версією Top (рисунок 23).

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if len(self.items) > 0:
            return self.items.pop(0)
        return None
```

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def top(self):
        return self.items[0]

    def pop(self):
        if len(self.items) > 0:
            return self.items.pop(0)
        return None
```

Рисунок 23 – Версія з методом top

Крім того, припустимо, що інший програміст згодом додає функцію Size безпосередньо до основної гілки, створюючи відповідну версію програми (рисунок 24).

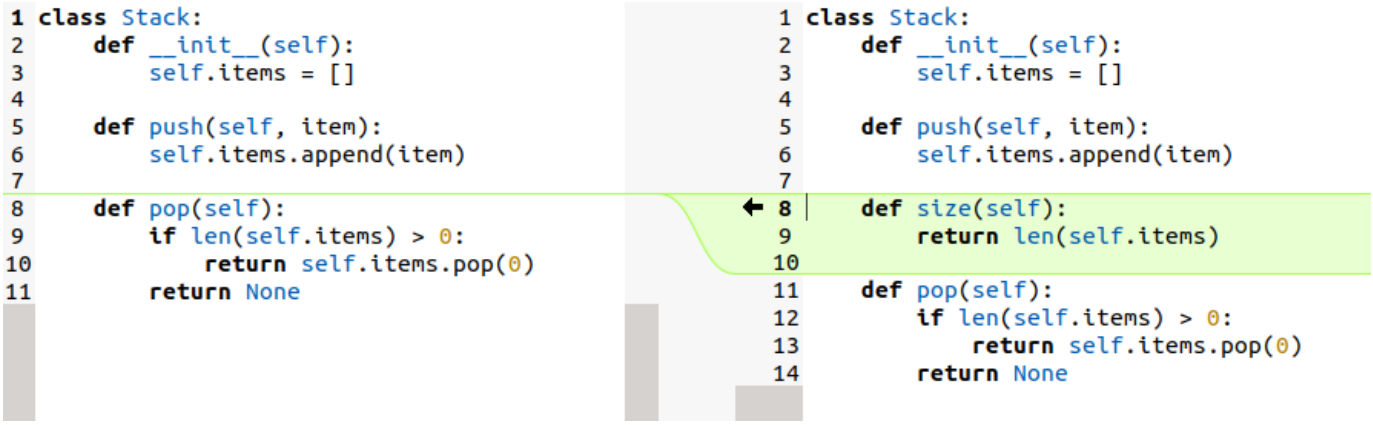

Рисунок 24 – Версія з методом size

Нарешті, припустимо, що в певний момент часу дві гілки знову об’єднаються, щоб об’єднати обидві редакції, включаючи нові функції.

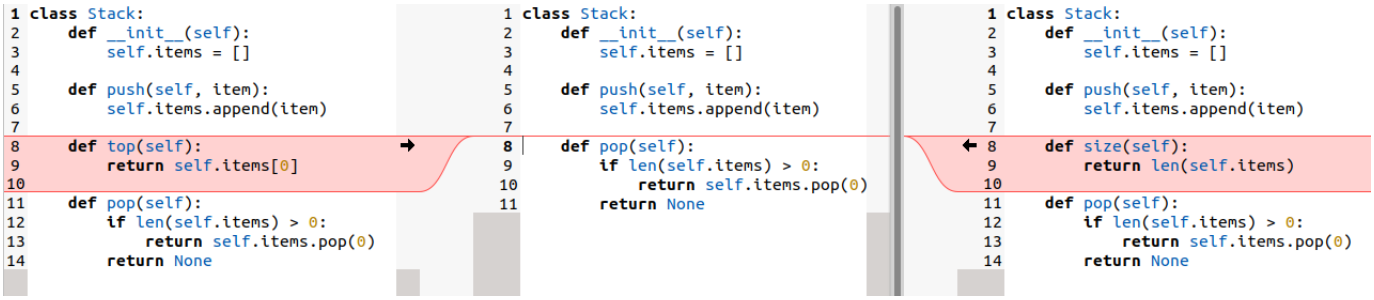

Рисунок 25 – Конфлікт

Об’єднання двох гілок передбачає об’єднання двох версій TOP і SIZE на основі спільного предка, базової програми STACK. Цей процес також називають тристороннім поєднанням, оскільки він включає три програми або документи [2]. У нашому прикладі процес поєднання повідомляє про конфлікт, який не можна вирішити автоматично за допомогою неструктурованого поєднання. Рисунок 25 ілюструє вихід інструменту поєднання Meld для цього прикладу. На рисунку показано, що процес поєднання не може об’єднати два нові методи top і size таким чином, щоб обидва могли бути присутніми в об’єднаній програмі.

Цей приклад дуже простий, але він вже ілюструє проблеми неструктурованого поєднання. Інструмент неструктурованого поєднання працює виключно на основі текстових рядків або маркерів. Він ідентифікує нові фрагменти тексту з урахуванням загального предка (базова програма) і зберігає загальні фрагменти до та після нових фрагментів. Якщо дві редакції змінюють або розширюють текст в одному регіоні, система повідомляє про конфлікт, тобто вона не може вирішити, як об'єднати зміни чи розширення. У нашому прикладі інструмент поєднання знає, що два незалежні фрагменти тексту (які фактично реалізують два методи top і size) додаються до одного розташування базової програми (яке огороджено двома фрагментами, які реалізують методи push і pop). Проблема полягає в тому, що інструмент неструктурованого поєднання не знає, що ці фрагменти є методами, і що поєднання двох насправді є простим, як ми проілюструємо далі.

```
1  class Stack:
2      def __init__(self):
3          self.items = []
4
5      def push(self, item):
6          self.items.append(item)
7
8      def size(self):
9          return len(self.items)
10
11     def top(self):
12         return self.items[0]
13
14     def pop(self):
15         if len(self.items) > 0:
16             return self.items.pop(0)
17         return None
18
```

Рисунок 26 – Очікуваний результат поєднання

Чому неструктурована система контролю редагування не в змозі вирішити конфлікт, який виникає при об'єднанні версій TOP і SIZE? Як зазначалося раніше,

інструмент неструктурованого поєднання не знає, що два фрагменти реалізують методи Python, порядок яких не має значення в межах оголошення класу. Якби інструмент знав, що базова програма та дві версії насправді є програмами Python, він міг би автоматично вирішити конфлікт. Насправді існує два способи вирішення конфлікту: спочатку включити метод top, а потім метод size (рисунок 26), або навпаки.

Проілюструємо можливості структурованого поєднання на наступному прикладі.

```
1 class Stack:                                  1 from collections.collection import Collection
2     def __init__(self):                      2
3         self.items = []                      3
4                                              4 class Stack(Collection):
5     def push(self, item):                    5     def __init__(self):
6         self.items.append(item)              6         self.items = []
7                                              7
```

Рисунок 27 – Додавання базового класу Collection

Припустимо, що ми маємо реалізацію базового стека, і ми створюємо дві незалежні версії, одну для успадкування від базового класу Collection (рисунок 27), а іншу для розробки для розробки функціональності серіалізації (рисунок 28).

```
1 class Stack:                                  1 from json import JSONEncoder
2     def __init__(self):                      2
3         self.items = []                      3
4                                              4 class Stack(JSONEncoder):
5     def push(self, item):                    5     def __init__(self):
6         self.items.append(item)              6         self.items = []
7                                              7
```

Рисунок 28 – Додавання базового класу для JSON серіалізації

Об’єднання двох версій з базовою програмою за допомогою неструктурованого поєднання викликає два конфлікти. По-перше, система не може об’єднати два нових оператори імпорту, а по-друге, вона не може об’єднати два реалізація двох версій. Запропонований спосіб здатний автоматично розв’язувати конфлікти та виробляти

бажаний результат, тобто імпорти розміщуються один за одним, а версії об’єднуються (рисунок 29).

```
1 from collections.collection import Collection
2 from json import JSONEncoder
3
4
5 class Stack(Collection, JSONEncoder):
6     def __init__(self):
7         self.items = []
8
```

Рисунок 29 – Результат поєднання використання базових класів

Варто зазначити, що даний випадок спрацює повністю на синтаксичному рівні. У випадку, якщо ще будуть додані зміни в __init__ метод з викликами super, то такий КП вирішиться вже на семантичному рівні ППК.

Основна ідея полягає в представленні програмних продуктів у вигляді дерев і наданні інформації про те, як вузли певного типу (наприклад, методи чи класи) та їх піддерева об’єднуються.

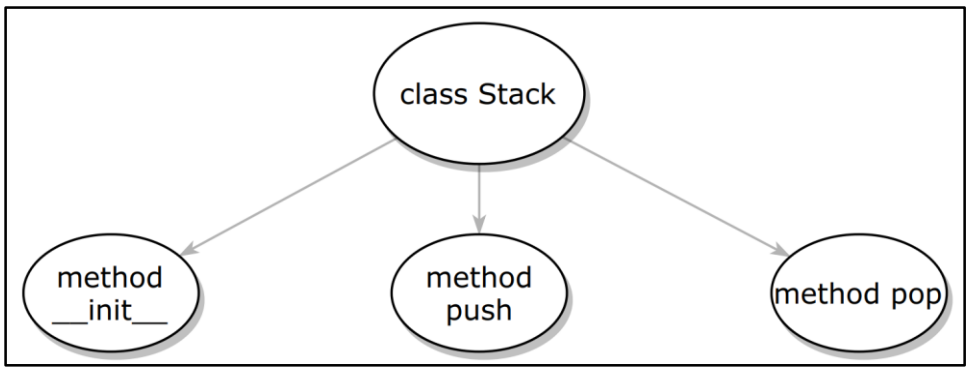

Рисунок 30 – Початкове дерево розбору

Таке дерево (рисунок 30) по суті є деревом аналізу, деревом структури програми (також відоме як дерево синтаксичних функцій) [12].

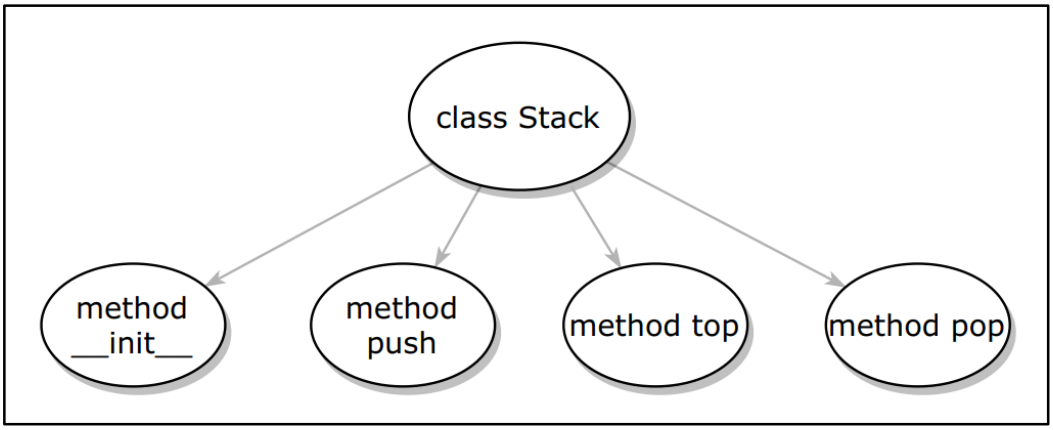

Рисунок 31 – Дерево розбору першої модифікації

Ми показуємо спрощене дерево розбору програми базової програми STACK, далі показано спрощене дерево структури програми TOP (рисунок 31) і SIZE (рисунок 32).

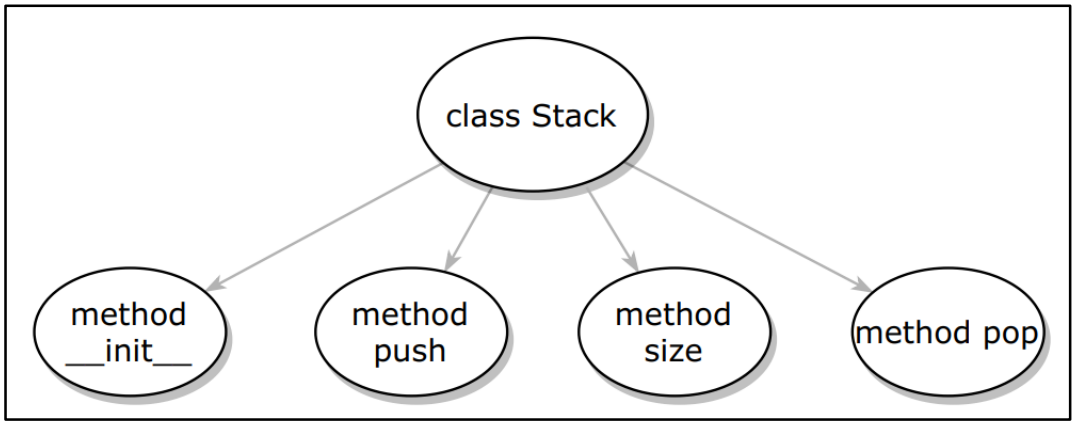

Рисунок 32 – Дерево розбору другої модифікації

Важливо відзначити, що не вся синтаксична інформація представлена в дереві. Наприклад, немає вузлів, які представляють оператори чи вирази. Але синтаксична інформація не втрачена; він міститься у вигляді простого тексту на листках дерева (асоційовану з вузлом метода). Отже, дерево структури програми не обов’язково є повним деревом розбору, але абстрагується від деяких деталей і представляє їх у вигляді простого тексту.

Вибір того, який тип синтаксичного елемента буде представлений окремим вузлом, залежить від виразності, якої ми хочемо досягти за допомогою поєднання. Пояснимо цей вибір на прикладі стека. Беручи три дерева структури програми як вхідні дані, інструмент поєднання може отримати бажаний результат, просто накладаючи дерева (рисунок 33).

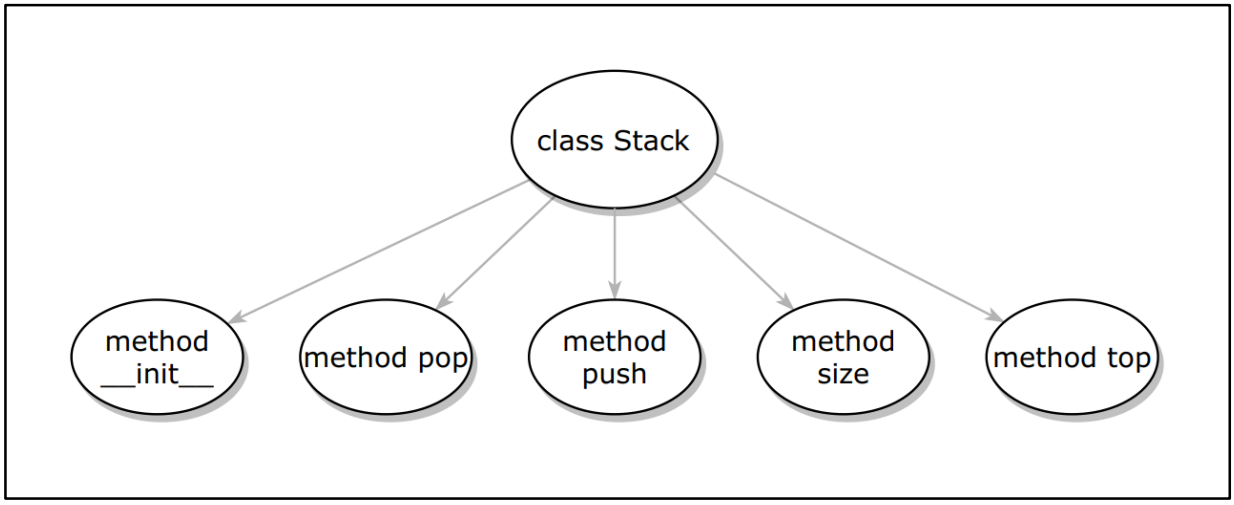

Рисунок 33 – Об’єднане дерево розбору

Цей алгоритм працює, оскільки порядок оголошення методів не має значення. Якби дві редакції додали методи з ідентичними назвами, інструменту довелося б об’єднати оператори їх тіла (що буде виконано на більш високому рівні ППК). Це було б складніше, оскільки їхній порядок має значення (оскільки оператори не обов’язково мають унікальні імена).

Навіть маючи всю синтаксичну інформацію про мову Python, можуть бути випадки, коли ми не можемо сказати, як об’єднати послідовності операторів. Це причина, чому вирішено представляти методи у вигляді листів, а їх наповнення опрацьовувати вже враховуючи більш високорівневе ППК. У прикладі на малюнку 8 ми бачимо, що неструктуроване поєднання 10 не може об’єднати різні версії двох версій класу. За допомогою структурованого поєднання ми можемо досягти цього,

оскільки знаємо, що списки типів можна об’єднати. У випадку ж якщо інформації, як об’єднати піддерева розбору, маючи суто синтаксичні та синтаксичні правила? В такому випадку, елементи представляються у більш високорівневому типі поєднання, вже з використанням, зокрема семантичних правил поєднання. Тобто, якщо конфлікт виникає всередині тіла методу, ми не можемо вирішити його автоматично на рівні синтаксичних правил.

3.2.1 Семантичний рівень

У даній роботі пропонується спосіб семантичного поєднання для мови Python, подібний за загальною ідеєю способу, що використовується у Semantic Diff для інших мов програмування.

У загальному випадку результат вирішення конфлікту поєднання модифікацій одного й того ж коду двома програмістами – поєднаний модифікований код – може бути, зокрема, таких видів:

- додавання нової функціональності;
- стильове покращення – рефакторинг (англ. "refactoring").

Додавання нової функціональності обов’язково призводить до зміни результату виконання програми, тоді як рефакторинг змінює тільки текст програми без зміни результату її виконання.

З великою ймовірністю конфліктні фрагменти можуть одночасно містити будь-яку кількість модифікацій коду обох видів. Нехай маємо конфлікт поєднання, що складається з модифікацій в обох версіях деякого коду. Пропонується розбиття обох версій на часткові модифікації коду, що однозначно підпадають під один з видів, та поєднувати окремо кожну з них. Таким чином маємо окремі конфлікти поєднання, що обмежуються наступними випадками комбінацій видів модифікації коду:

- рефакторинг поєднується з рефакторингом. Такий конфлікт можливо розв’язати за наявності критерію якості рефакторингу. Для однозначного визначення яку із версій модифікації обрати, необхідні наперед задані пари фрагментів конструкцій мови програмування із однозначно вказаним фрагментом, якому надається перевага;
- нова функціональність поєднується з рефакторингом. Для розв’язання такого конфлікту, необхідно застосувати рефакторинг до модифікації, що змінює функціональність;
- нова функціональність поєднується з новою функціональністю. В такому випадку необхідно надавати можливість користувачу самостійно збирати обидві версії модифікації коду в одну за мінімальну кількість вручну виконуваних дій. Для цього застосуємо спосіб запропонований в [3] для виведення семантичних ефектів модифікації коду.

Для визначення рефакторингу виникає необхідність визначення еквівалентності коду. Зауважимо, що для системи повної за Тюрингом, – це алгоритмічно нерозв’язна задача [4]. Але якщо внести певні обмеження на функціональну повноту, то можна застосувати циклічний метод для визначення еквівалентності програм [5]. Для цього програми приводяться до проміжного представлення, збираючи сентенції, що не підпадають під вимоги циклічного методу.

3.2.2 Міжпроцедурний ГЗП

Тепер перейдемо до визначення графа залежності системи. Міжпроцедурний ГЗП являє собою розширення графів залежностей, визначених у розділі 2.2, представляє програми мовою, яка включає функції та виклики процедур.

Наше визначення графа системної залежності моделює мову з такими властивостями:

1. Повна система складається з однієї (основної) програми та набору допоміжних процедур.
2. Параметри передаються за значенням-результатом.

Повинно стати зрозумілим, що наш підхід не прив’язаний до перерахованих вище особливостей мови. Моделювання різних функцій вимагатиме певної адаптації; однак основний підхід застосовний до мов, які дозволяють вкладені області видимості, і мов, які використовують різні механізми передачі параметрів.

Граф залежностей системи включає в себе граф залежності програми, який представляє основну програму системи, графи залежності процедур, які представляють допоміжні функції системи, і деякі додаткові ребра. Ці додаткові ребра бувають двох видів:

- ребра, які представляють прямі залежності між місцем виклику та викликаною процедурою;
- ребра, які представляють транзитивні залежності через виклики.

У наведених нижче розділах ми використовуємо «процедуру» як загальний термін, що стосується як основної програми, так і допоміжних процедур, коли різниця між ними не має значення.

3.2.3 Виклики процедур і передача параметрів

Розширення визначення графів залежностей для обробки викликів процедур вимагає представлення передачі значень між процедурами. При розробці представлення передачі параметрів ми маємо три цілі:

1. Повинна бути можливість побудувати ГЗП окремої функції (включаючи обчислення залежностей даних) з мінімальними знаннями про інші компоненти системи.

2. Граф системної залежності повинен складатися з прямого зв’язку графа залежності програми та графів залежності процедури.
3. Повинна бути можливість ефективно витягти точний міжпроцедурний зріз шляхом обходу внутрішньопроцедурного ГЗП.

Ціль (3) є предметом подальшого опису, де представлено алгоритм розрізання графа системної залежності.

Щоб досягти цілей, викладених вище, наші графи моделюють наступний двоетапний механізм передачі параметрів під час виконання: Коли процедура P викликає процедуру Q, значення передаються з P в Q за допомогою проміжних тимчасових змінних, один для кожного параметра. Інший набір тимчасових змінних використовується, коли Q повертається для передачі значень назад до P. Перед викликом P копіює значення фактичних параметрів у тимчасові елементи виклику; Q потім ініціалізує локальні змінні з цих тимчасових елементів. Перед поверненням Q копіює повернуті значення в тимчасові елементи, з яких P отримує їх.

Ця модель передачі параметрів представлена в графах залежності процедур за допомогою використання п’яти нових типів вершин. Місце виклику представлене за допомогою вершини місця виклику; передача інформації представлена за допомогою чотирьох типів вершин параметрів. На стороні виклику передача інформації представлена набором вершин, які називаються вершинами фактичних вхідних і фактичних вихідних параметрів. Ці вершини, які залежать від керування від вершини місця виклику, представляють оператори присвоєння, які копіюють значення фактичних параметрів у тимчасові виклики та з тимчасових повернення відповідно. Аналогічно, передача інформації в викликаній функції представлена набором вершин, які називаються формальними і формальними вершинами. Ці вершини, які залежать від керування від вершини входу процедури, представляють оператори присвоєння, які копіюють значення формальних параметрів із тимчасових викликів та у тимчасові результати, відповідно.

Використовуючи цю модель, залежності даних між процедурами обмежуються залежностями від фактичних вхідних вершин до формальних вхідних вершин і від формальних вихідних до фактичних вихідних вершин. З'єднання ГЗП процедур для формування графа залежності системи є простим, включає додавання трьох нових типів ребер:

1) ребро виклику додається з кожної вершини місця виклику до відповідної вершини входу процедури;
2) ребро вхідного параметра додається з кожної фактично-вхідної вершини на місці виклику до відповідної формально-вхідної вершини у викликаній процедурі;
3) вихідне ребро параметра додається з кожної формально вихідної вершини у викликаній функції до відповідної фактично вихідної вершини на місці виклику. (Ребра виклику — це новий тип краю залежності керування; межі вхідних і вихідних параметрів — це нові види краю залежності даних).

Ще одна перевага цієї моделі полягає в тому, що залежності потоку можна обчислити звичайним способом, використовуючи аналіз потоку даних на графіку потоку керування процедурою. Граф потоку керування для функції включає вузли, аналогічні вершинам фактичного входу, фактичного вихідного, формального входу та формального виходу графа залежності процедури. Графік потоку керування процедурою починається з послідовності присвоєнь, які копіюють значення з тимчасових викликів до формальних параметрів, і закінчується послідовністю присвоєнь, які копіюють значення з формальних параметрів у повертають тимчасові параметри. Кожен оператор виклику в рамках функції представлений на графіку потоку керування функції послідовністю присвоєнь, які копіюють значення з фактичних параметрів у тимчасові виклики, за якими слідує послідовність присвоєнь, які копіюють значення з повернення тимчасових до фактичних параметрів.

Важливим питанням є те, які значення передаються з місця виклику до викликаної функції і назад. Цей момент обговорюється далі в Розділі 4.2, де представлена стратегія, в якій результати міжпроцедурного аналізу потоку даних використовуються для опускання деяких вершин параметрів з графіків залежності процедури. Наразі ми будемо вважати, що всі фактичні параметри копіюються в тимчасові файли виклику і витягуються з тимчасових елементів, що повертаються. Таким чином, вершини параметра, пов'язані з викликом функції P до функції Q, визначаються наступним чином (GP позначає графік залежності функції для P):

- у GP, підпорядкованій вершині місця виклику, яка представляє виклик Q, існує фактична вершина для кожного фактичного параметра *e* виклику Q. Вершини фактичного входу позначаються *r_in := e*, де *r* є ім'я формального параметра;
- для кожного фактичного параметра a, який є змінною (а не виразом), існує фактично вихідна вершина. Вони позначаються як *a := r_out* для фактичного параметра *a* і відповідного формального параметра *r*.

Вершини параметрів, пов'язані з входом до функції Q та поверненням із функції Q, визначаються таким чином (GQ позначає графік залежності функції для Q): для кожного формального параметра r Q, GQ містить формально-вхідну вершину і формально вихідну вершину. Ці вершини позначені *r = r_in* і *r_out = r* відповідно.

```
 1 def main():
 2       sum = []
 3       i = 1;
 4       while i < 11:
 5           foo(sum, [i])
 6       return
 7
 8 def foo(x, y):
 9       add(x, y)
10       inc(y)
11       return
12
13 def add(a, b):
14       a.append(b)
15       return
16
17 def inc(z):
18       add(z, 1)
19       return
20
```

Рисунок 34 – Приклад програми для побудови міжпроцедурного ГЗП

Наприклад, рисунок 39 показує відповідні графіки залежності програми (рисунок 34) та процедури, пов’язані з ребрами входу параметра, ребрами вихідних параметрів і ребрами виклику. (ребра порядку визначення не показані. Ребра, що представляють керуючі залежності, показані без міток; усі такі ребра в цьому прикладі будуть позначені як true).

В загальному випадку параметри виклику функції або функції може виконується як за значенням, так і за посиланням.

1. При передачі змінної за значенням, в змінну формального параметру присвоюється копія значення фактичного параметру.
2. При передачі змінної за посиланням, об’єкт, що створився у викликаючій процедурі, присвоюється змінній, що відповідає формальному параметру.

В мові Python, параметри передаються за посиланням. В той же час, введено поняття змінюваних (англ. “mutable”) і незмінюваних (англ. “immutable”) типів даних,

значення яких при передачі в функцію може бути відповідно зміненим або не зміненим при модифікації значення у викликаній функції [13]. Наприклад, для програми (рисунок 35) введено змінну *a*, яка при передачі в функцію *foo*.

```
1  def foo(a):
2      a += 1
3      print(a)
4
5
6  if __name__ == '__main__':
7      a = 1
8      foo(a)
9      print(a)
10
```

Рисунок 35 – Приклад програми з незмінюваним параметром

Оскільки цілочисельний тип належить до незмінюваних типів, “a += 1”, вводить нову змінну, що перевизначає параметр a як нову локальну змінну.

```
Shell
2
1
>
```

Рисунок 36 – Вивід програми з незмінюваним параметром

З іншого боку, якщо використати змінюваний тип та операцію, що змінює значення переданої змінної, отримаємо повноцінну модифікацію об’єкта ініціалізованого в контексті виклику. Наприклад, візьмемо вбудований тип List та його деструктивний метод append (рисунок 37).

```
1  def foo(a):
2      a.append(1)
3      print(a)
4
5
6  if __name__ == '__main__':
7      a = []
8      foo(a)
9      print(a)
10
```

Рисунок 37 – Приклад програми із змінюваним параметром

Враховуючи те, що a залишається модифікованою після виклику foo, маємо результат (див. рисунок 38).

```
Shell
[1]
[1]
>
```

Рисунок 38 – Вивід програми із змінюваним параметром

Таким чином, необхідно виділити операції, що можуть модифікувати переданий параметр. Враховуючи, що в мові Python одна з базових операцій модифікацій можна описати як модифікацію поля класу. Таким чином, метод, що викликає модифікацію поля класу, також перетворюється на деструктивну операцію.

Складним аспектом міжпроцедурного зрізу є відстеження контексту виклику, коли фрагмент «опускається» у викликану процедуру.

Ключовим елементом нашого підходу є використання характеристичних ребер графа зв’язків у графі системної залежності. Ці ребра представляють перехідні залежності даних від фактичних вхідних вершин до фактичних вихідних вершин через

виклики процедури. Наявність таких ребер дозволяє нам обійти проблему контексту виклику; операція зрізування може обходити виклик без необхідності спускатися в нього.

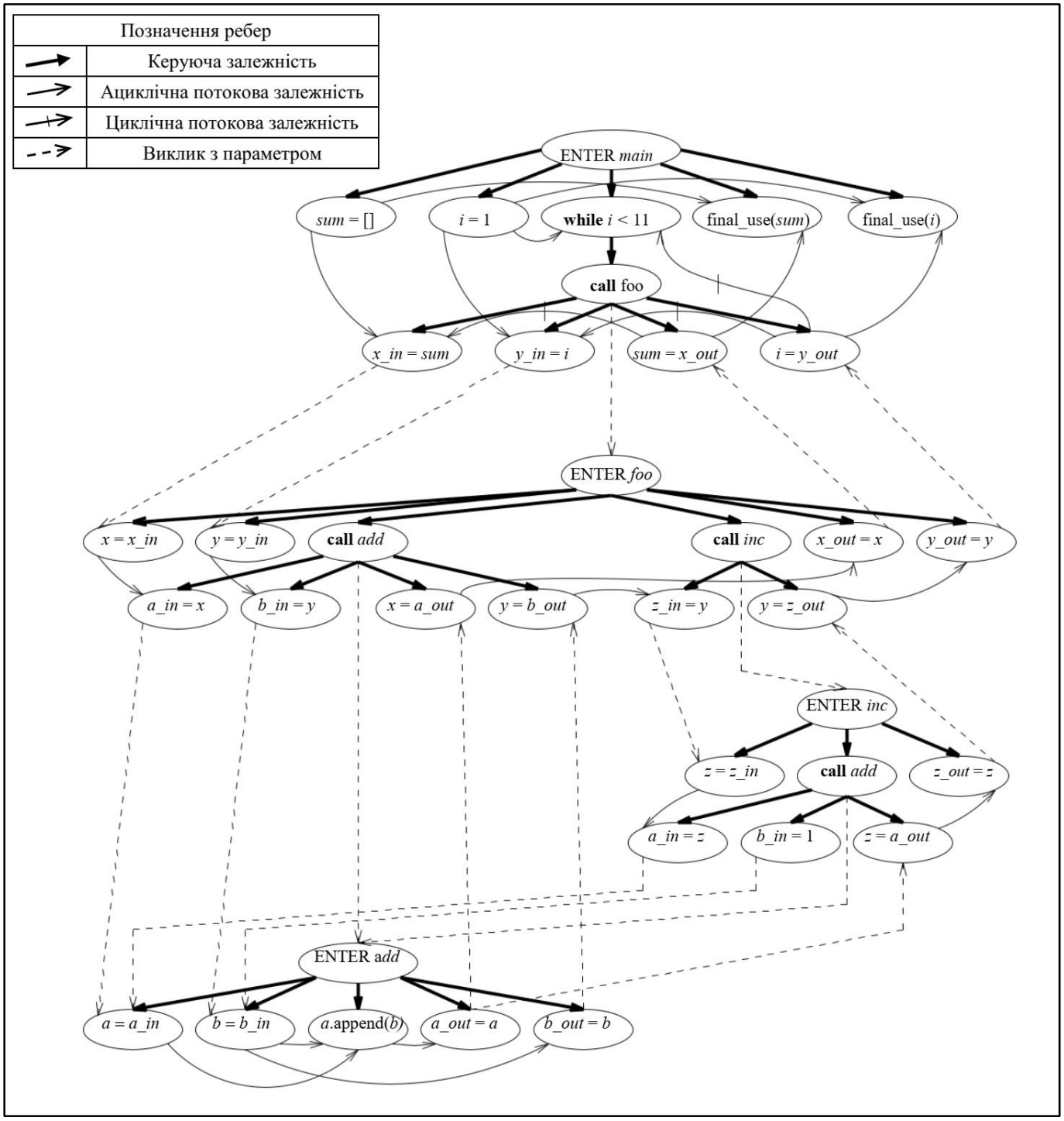

Рисунок 39 – Приклад міжпроцедурного ГЗП

Наведемо алгоритм міжпроцедурного зрізу.

Обчислення зрізу графіка залежності системи G щодо набору вершин S виконується в два етапи. Обидва етапи 1 і 2 працюють на МГЗП, використовуючи, по суті, метод, представлений у підпункті 2.2, для формування внутрішньопроцедурного зрізу – граф обходиться, щоб знайти набір вершин, які можуть досягати заданого набору вершин уздовж певних типів ребер.

Мета полягає в тому, щоб отримати зріз МГЗП G відносно деякої вершини s в функції P;

Етапи 1 і 2 можна охарактеризувати наступним чином:

- етап 1 визначає вершини, які можуть досягати s і знаходяться або в самому P, або в процедурі, яка викликає P (прямо або транзитивно). Оскільки ребра вихідних параметрів не дотримуються, обхід на Фазі 1 не «спускається» в процедури, викликані P. Однак наслідки таких процедур не ігноруються; Наявність ребер транзитивної залежності потоку від вершин фактично вхідних до фактичних (ребер підпорядкованих характеристик) дозволяє виявляти вершини, які можуть досягти s лише через виклик процедури, хоча обхід графа фактично не опускається до викликаної процедури;
- етап 2 визначає вершини, які можуть досягати s з процедур (транзитивно), викликаних P, або з процедур, викликаних процедурами, які (транзитивно) викликають P. Оскільки ребра виклику та ребра вхідних параметрів не дотримуються, обхід в етапі 2 не виконується. «підйом» у функції виклику; ребра залежності транзитивного потоку від вершин фактичного входу до фактичного вихідного роблять такі «підйоми» непотрібними.

Обхід в етапі 1 слідує за ребрами потоку, керуючими ребрами, ребрами виклику та ребрами входу параметра, але не слідує за ребрами порядку визначення або вихідними ребрами параметрів. Обхід в етапі 2 слідує за ребрами потоку, керуючими

ребрами та ребрами вихідних параметрів, але не слідує за ребрами порядку визначення, ребрами виклику чи ребрами входу параметра.

3.2.4 Глобальні змінні

Глобальні змінні не підтримуються, оскільки для локального ГЗП процедури неможливо достовірно визначити всі шляхи попередньої ініціалізації значення глобальної змінної…….

3.2.5 Перетворення конструкцій yield в проміжну мову

Відповідно до семантики Python, можна провести трансформацію коду, що використовує оператор yield в код, що використовує звичайні цикли for.

Оператор yield фактично заміняє оператор return для функції, всередині якої цей оператор використовується. Тоді така функція перетворюється на генератор.

```
1 def foo():
2     for i in range(2):
3         for j in range(3):
4             yield i, j
5
6
7 for i, j in foo():
8     print("i=%d, j=%d\n" % (i, j))
9
10
```

Рисунок 40 – Приклад з оператором yield

Виклик такої функції повертає iterable об’єкт (рисунок 40), тобто такий об’єкт, до якого можна застосувати цикл for-in або обхідні функції з фунціональної

бібліотеки. Основна відмінність від return в тому, що в місці виклику цього оператора підставляється код, що обробляє ітераційний елемент послідовності.

Іншими словами, наведений код з оператором yield можна замінити на відповідний еквівалентний з використанням тільки for-in (рисунок 41).

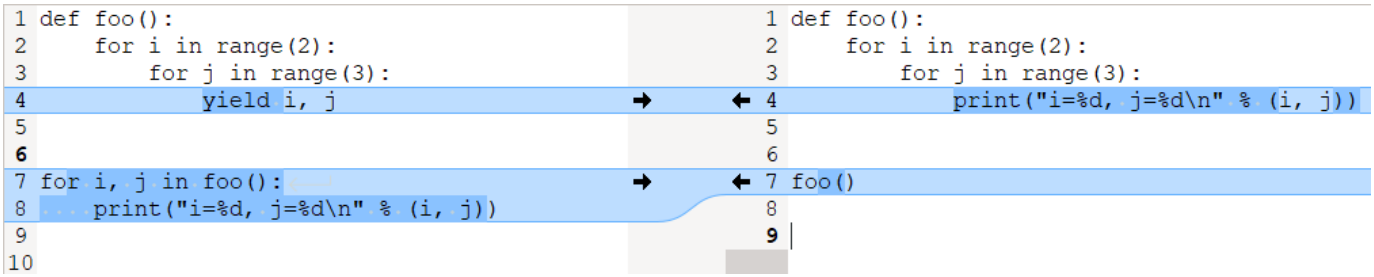

Рисунок 41 – Приклад видалення оператора yield

Таким чином є можливість без введення додаткових залежностей в ГЗП вирішувати задачу поєднання шляхом приведення непідтримуваного коду з оператором yield у відповідний підтримуваний запропонованим способом.

3.2.6 Групування модифікацій в один семантичний ефект

Розглянемо приклад виявлення ефекту модифікації при перейменуванні змінної (рисунок 42).

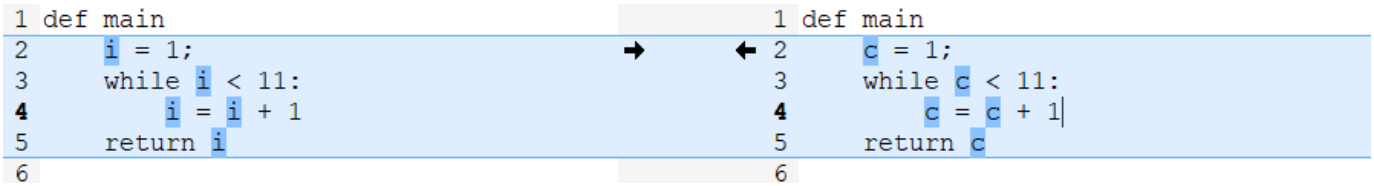

Рисунок 42 – Приклад модифікації перейменування

За даними ГЗП, серед списку змінних в графі виводиться перейменування змінної в вершинах графа. Для виявлення перейменувань всередині тіла функції

вводиться таблиця ідентифікаторів, яка аналізуються окремо від графа, виявляючи перейменування спочатку за виявленням еквівалентних вершин ГЗП. Далі в еквівалентних вершин проставляється однакові ідентифікатори, які далі знаходяться в таблиці ідентифікаторів і реєструється перейменування.

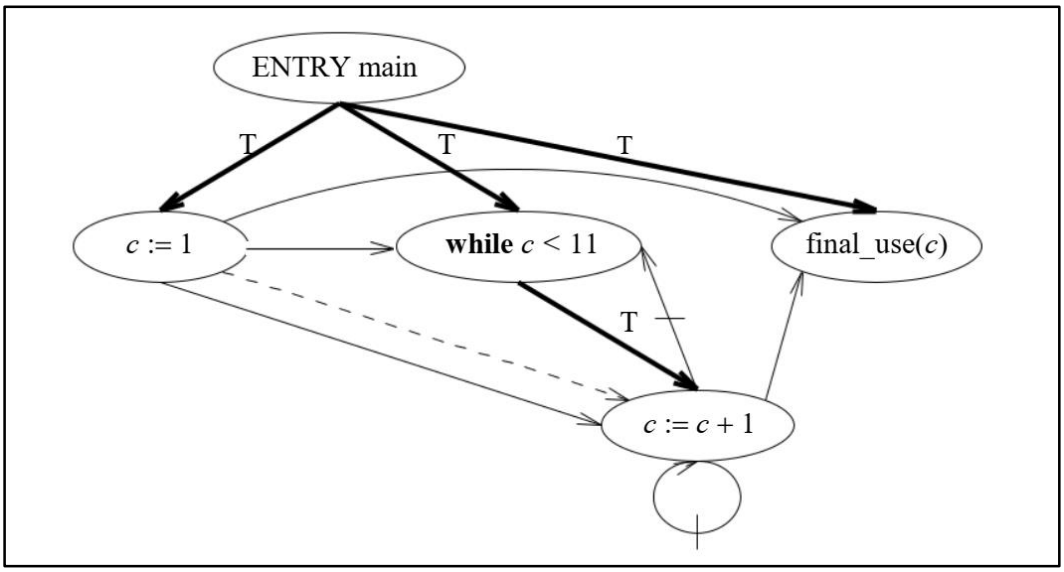

Рисунок 43 – ГЗП після модифікації перейменування

Також наведемо приклад модифікації, що впливає на ефект програми (рисунок 44).

```
1 def main                                  1 def main
2     i = 1;                                2     i = 1;
3     while i < 11:                         3     while i < 11:
4         i = i + 1                     ➜ ← 4         i = 2 + i
5     return i                              5     return i
6                                           6
```

Рисунок 44 – Приклад модифікації виразу

Таким чином, маємо вершину, що відповідає вершині присвоєння на ГЗП (рисунок 45).

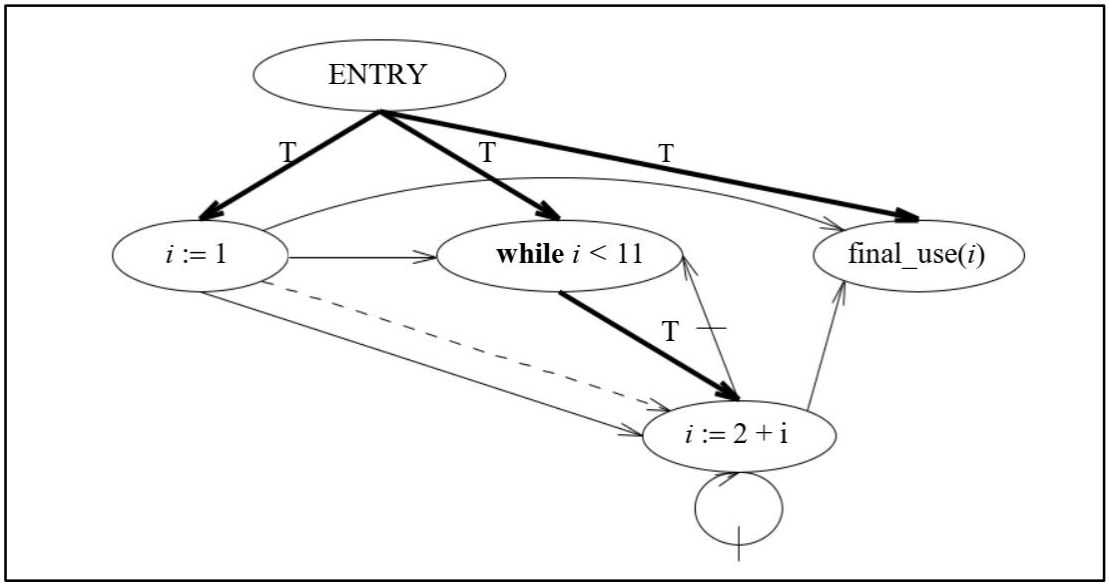

Рисунок 45 – Приклад ГЗП модифікації виразу

3.2.7 Поєднання МГЗП

Розглянемо процес поєднання окремих ГЗП, що були представлені раніше (рисунок 43 і рисунок 45). Перша модифікація являє собою модифікацію перейменування змінної, а друга – заміна в операторі присвоєння.

Пропонується виконати наступну послідовність дій:

1. Застосовувати модифікацію перейменування відносно зміни в згаданій таблиці ідентифікаторів.
2. Застосувати відповідну модифікацію заміни виразу.

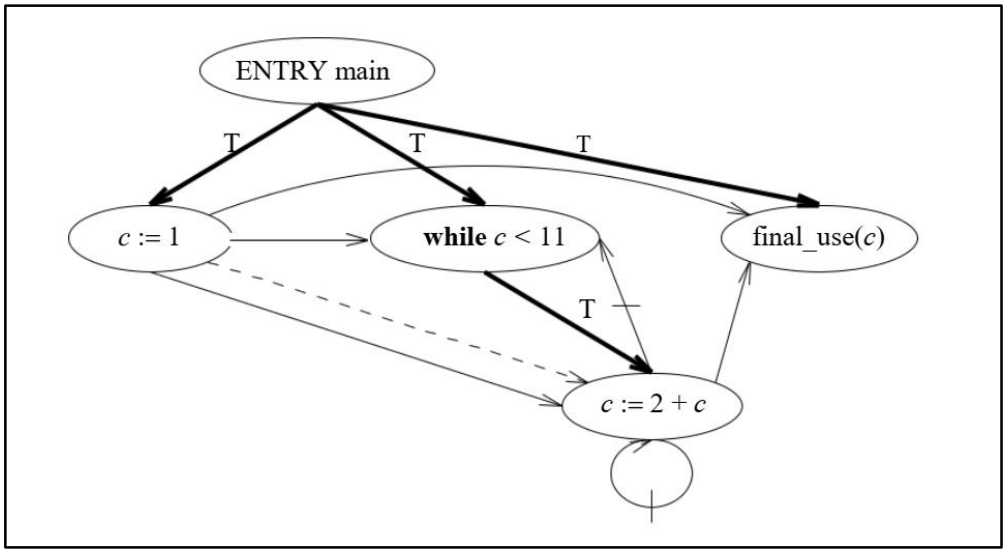

Рисунок 46 – ГЗП поєднаних модифікацій

На основі ГЗП можна побудувати поєднаний програмний код (рисунок 47).

```
def main
    c = 1;
    while c < 11:
        c = 2 + c
    return c
```

Рисунок 47 – Поєднаний програмний код

Висновки до розділу 3

Запропонований спосіб складається із кількох рівнів абстракції за ППК:

1. Лексичний – для обробки конфліктів, що виникають всередині лексем і можуть бути тривіально розв’язані;
2. Синтаксичний – для обробки конфліктів, що виникають при поєднанні піддерев розбору та мають тривіальні правила для їх поєднання (для випадків створення, перейменування, перемішення або видалення певної сутності (терміналу);
3. Семантичний – для обробки конфліктів, що потребують семантичних даних для вирішення КП. В якості семантичних даних використано мультиграф залежностей програми з наступними модифікаціями:
    - додатковою інформацією про виклики функцій та переданих параметрів;
    - виявлення модифікації об’єкта, що був переданий за посиланням та збереженням інформації про модифікацію в ГЗП.

Кожен з рівнів є надбудовою над попереднім і являє собою повний додатковий обхід над ППК більш високого рівня.

4 ЕКСПЕРИМЕНТАЛЬНІ ДАНІ ТЕСТУВАННЯ ЗАПРОПОНОВАНОГО СПОСОБУ

Спочатку, на основі історії контролю версій певних програмних проєктів з відкритим кодом, взятих із системи контролю версій git, були відтворені конфлікти поєднання, що мали місце при побудові функціональності цих проєктів. Потім ці конфлікти були передані на автоматичну обробку та визначена кількість конфліктів, що були правильно розв’язані. Після чого, була підрахована кількість розв’язаних автоматично конфліктів та конфліктів із згрупованим семантичним ефектом, що забезпечує прискорення ручного процесу поєднання.

4.1 Частка автоматичного вирішення КП

В даному підрозділі показано порівняльну статистику появи автоматичних та автоматизованих вирішення КП.

Проілюструємо кількість розв'язаних КП в проєкті CPython (рисунок 48).

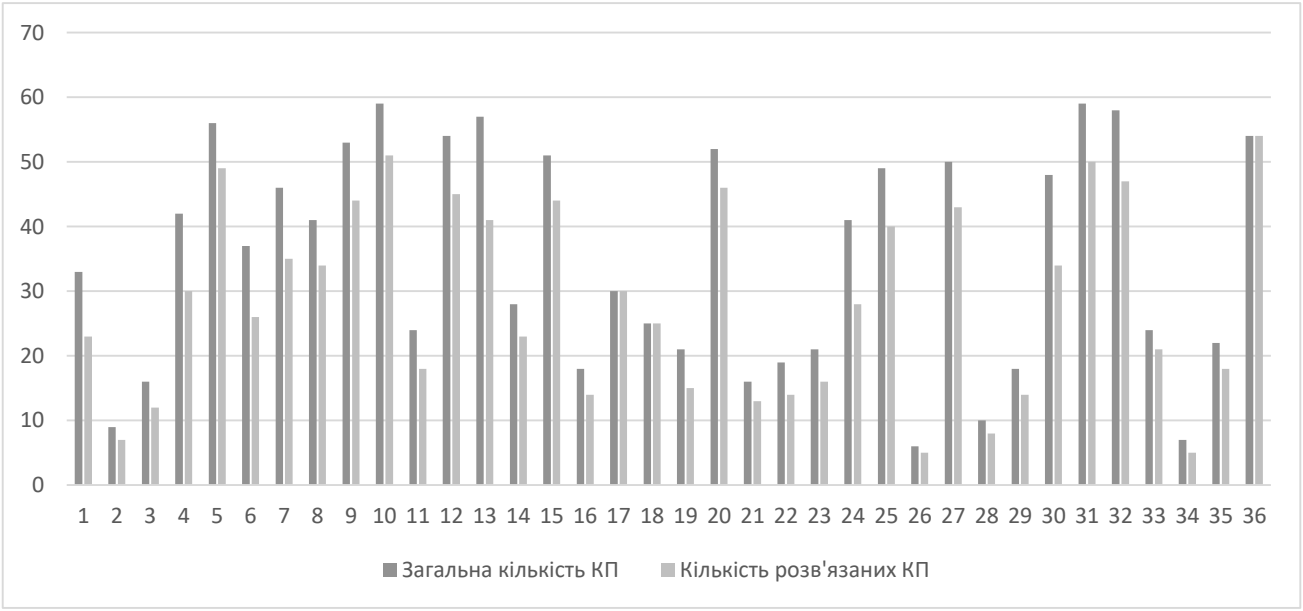

Рисунок 48 – Діаграма розв'язаних конфліктів в проєкті CPython

Проілюструємо кількість розв'язаних КП в проєкті Django (рисунок 49).

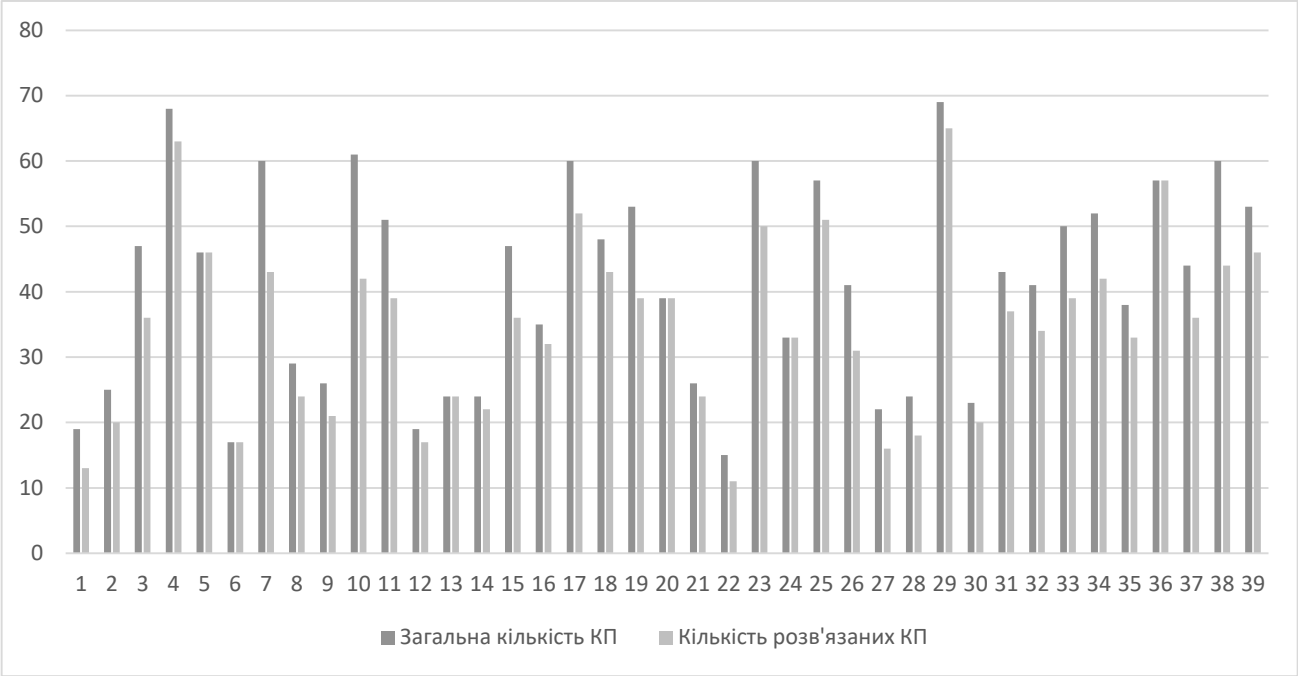

Рисунок 49 – Діаграма розв'язаних конфліктів в проєкті Django

Проілюструємо кількість розв'язаних КП в проєкті Home Assistant (рисунок 50).

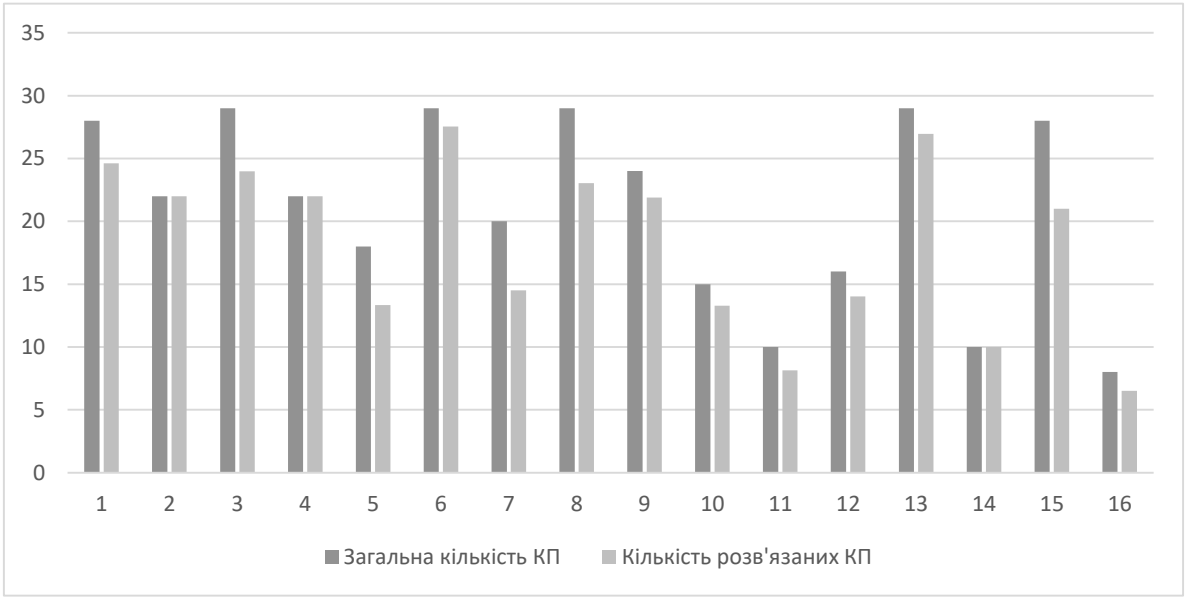

Рисунок 50 – Діаграма розв'язаних конфліктів в проєкті Home Assistant

Отже, як видно з діаграм, частина автоматизованих розв’язувань конфліктів переважає значно над тими, що виконані вручну. Як видно з отриманих даних, найгірше відношення – 74.38%, а найкраще – 100%.

4.2 Частка автоматизованого поєднання

Покажемо окрему кількість автоматичного, автоматизованого та ручного вирішення КП. Ці дані покажуть інформацію про теоретичні передбачення щодо часових параметрів окремих випадків поєднання. Відповідно до середніх вимірів, найшвидшим розв’язанням КП є повністю автоматичне, далі пришвидшене, але не на порядок – автоматизоване за збором семантичного ефекту та подальшої участі користувача. Відповідно, найповільнішим є ручне тестове поєднання, оскільки користувач має справу з текстовою інформацією напряму.

Проілюструємо кількість розв'язаних КП за типами вирішень в проєкті CPython (рисунок 51).

Рисунок 51 – Діаграма типів вирішень КП в проєкті CPython

Проілюструємо кількість розв'язаних КП за типами вирішень в проєкті Django (рисунок 52).

Рисунок 52 – Діаграма типів вирішень КП в проєкті Django

Проілюструємо кількість розв'язаних КП за типами вирішень в проєкті Home Assistant (рисунок 53).

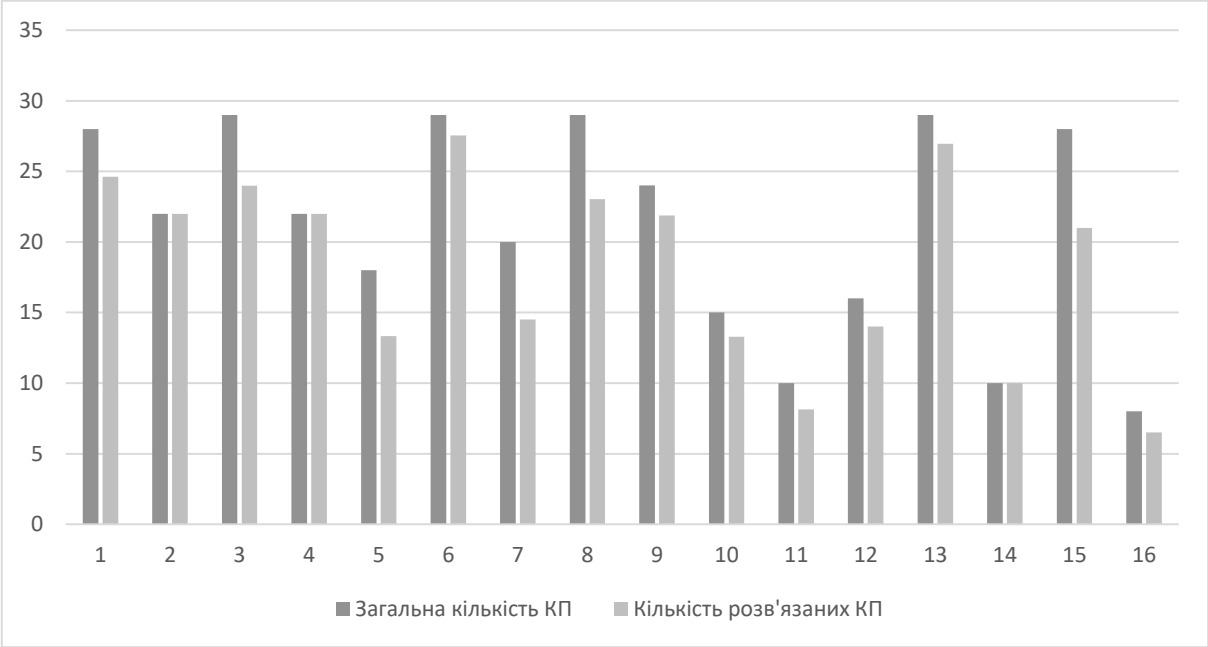

Рисунок 53 – Діаграма типів вирішень КП в проєкті Home Assistant

4.3 Приріст швидкодії виконання задачі поєднання за рахунок автоматизації

Для визначення приросту швидкодії виконано заміри часу виконання ручних та автоматизованих поєднувань.

Для CPython (рисунок 54).

Рисунок 54 – Діаграма часових замірів розв’язання КП в проєкті CPython

Відносний приріст швидкодії (рисунок 55).

Рисунок 55 – Відносний приріст швидкодії розв’язання КП в проєкті CPython

Для Django (рисунок 56).

Рисунок 56 – Діаграма часових замірів розв’язання КП в проєкті Django

Відносний приріст швидкодії (рисунок 57).

Рисунок 57 – Відносний приріст швидкодії розв’язання КП в проєкті Django

Для Home Assistant (рисунок 58).

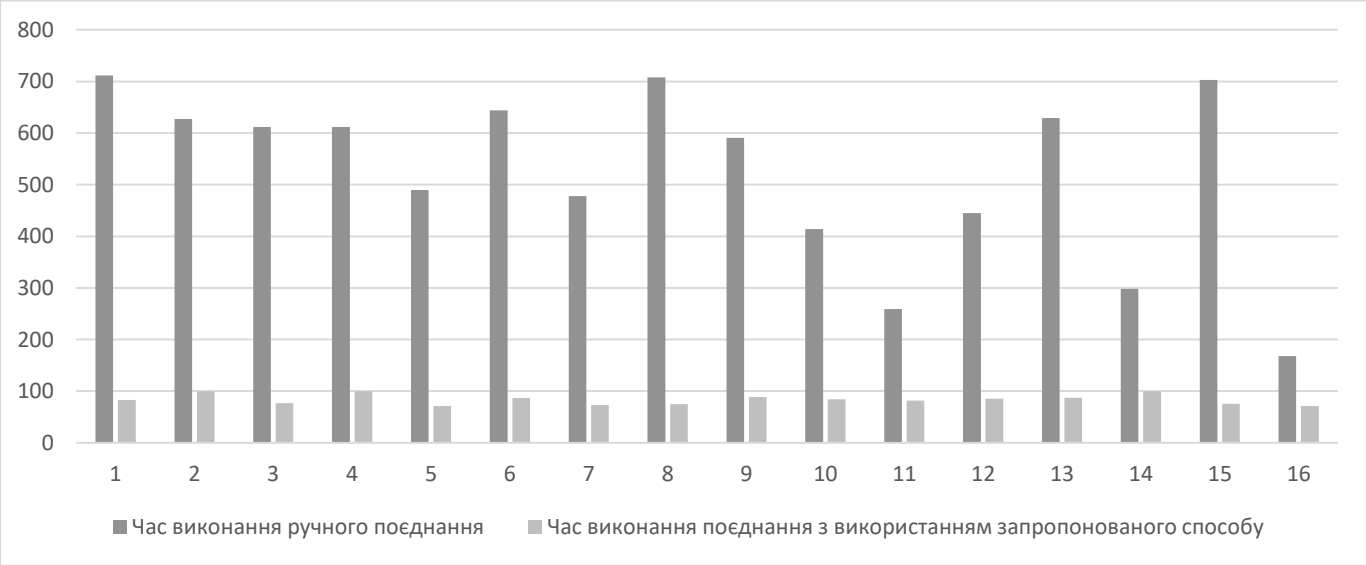

Рисунок 58 – Діаграма часових замірів розв’язання КП в проєкті Home Assistant

Відносний приріст швидкодії (рисунок 59).

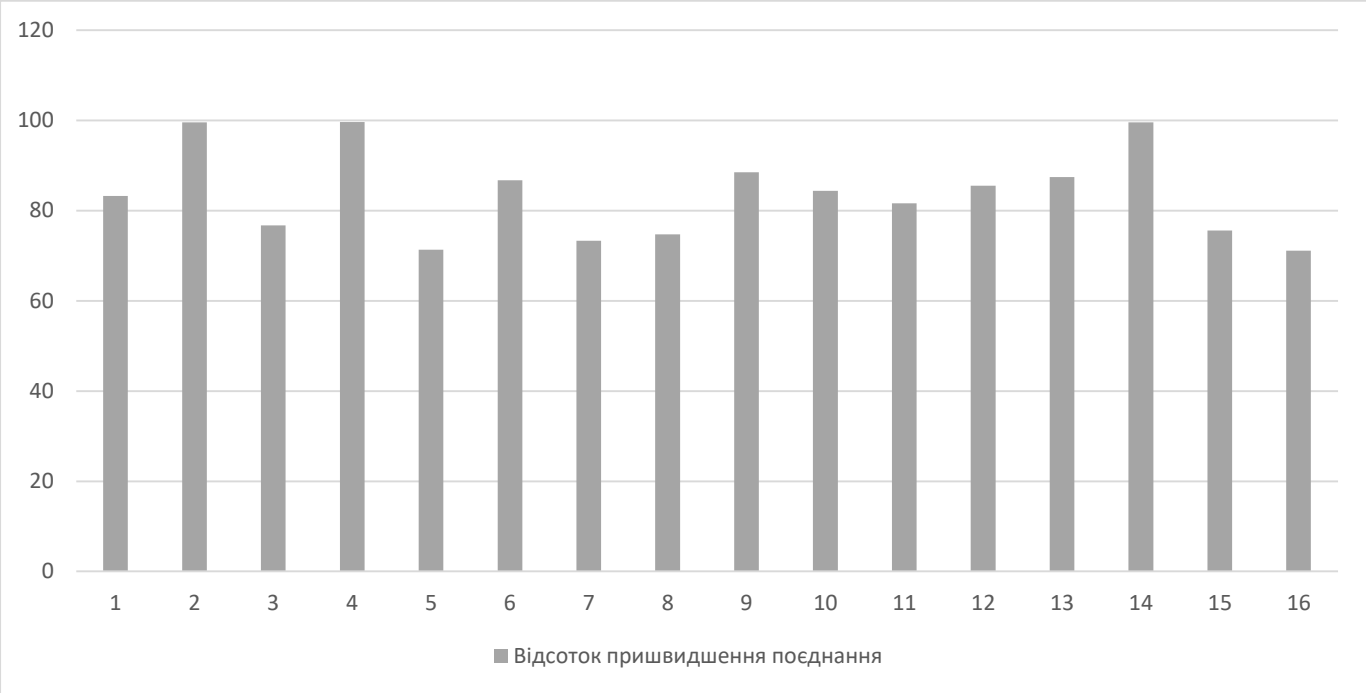

Рисунок 59– Відносний приріст швидкодії розв’язання КП в проєкті Home Assistant

Для автоматичних і ручних проведено 5 повторних вимірів та вибрано середнє значення.

На швидкість виконання поєднування впливають наступні фактори:

- кількість автоматично розв’язаних конфліктів;
- кількість автоматично не розв’язаних конфліктів, що потім були передані користувачеві на ручне поєднання;
- кількість конфліктів, семантичний ефект модифікацій яких було згруповано, пришвидшивши при цьому їх поєднання.

4.4 Класифікація конфліктів за рівнями вирішення

Запропонований спосіб поєднання працює з програмним кодом, представленим на різних рівнях – від лексем до графу залежностей програм. Покажемо розподіл за рівнями ІП автоматично вирішених конфліктів поєднання без участі користувача.

Проілюструємо кількість розв'язаних КП за типами вирішень в проєкті CPython (рисунок 60).

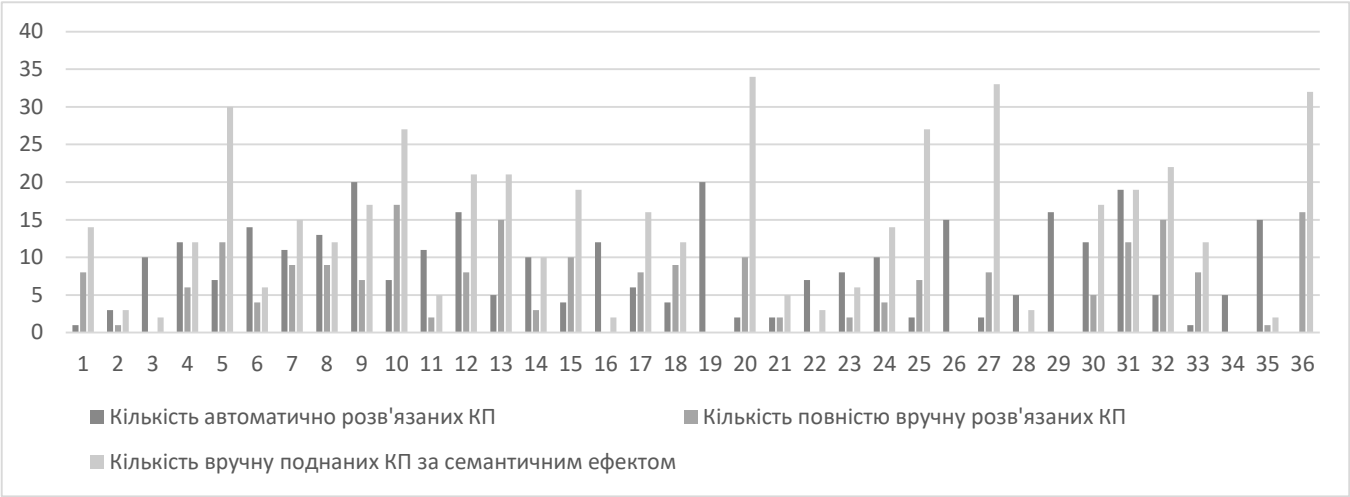

Рисунок 60 – Розподіл кількості КП за рівнями обробки в проєкті CPython

Проілюструємо кількість розв'язаних КП за типами вирішень в проєкті Django (Рисунок 61).

Рисунок 61 – Розподіл кількості КП за рівнями обробки в проєкті Django

Проілюструємо кількість розв'язаних КП за типами вирішень в проєкті Home Assistant (Рисунок 62).

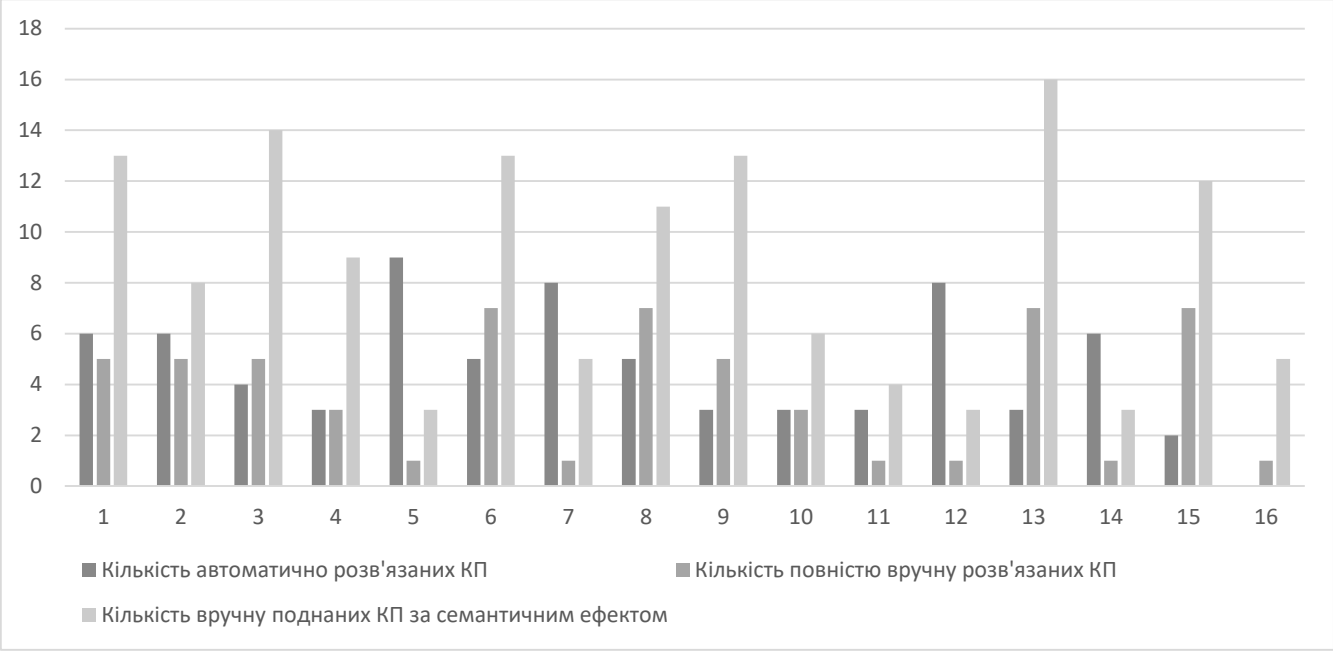

Рисунок 62– Розподіл кількості КП за рівнями обробки в проєкті Home Assistant

Висновки до розділу 4

Для автоматичних і ручних проведено 5 повторних вимірів та вибрано середнє значення.

На швидкість виконання поєднування впливають наступні фактори:

- кількість автоматично розв’язаних конфліктів;
- кількість автоматично не розв’язаних конфліктів, що потім були передані користувачеві на ручне поєднання;
- кількість конфліктів, семантичний ефект модифікацій яких було згруповано, пришвидшивши при цьому їх поєднання.

На основі експериментальних даних показано ефективність інструменту поєднання, що працює за запропонованим способом.

ВИСНОВКИ

Поєднання версій програмного коду, які виникають при його одночасній модифікації декількома розробниками є невід'ємною частиною розробки програмного забезпечення. Тому ефективне рішення задачі автоматичного вирішення конфліктів при таких поєднаннях може значно допомогти розробникам в швидкодії та якості процесу розробки програмного забезпечення. У даній роботі описано основну класифікацію способів вирішення цієї задачі. Серед дво- і тристоронніх способів поєднання визначено, що тристоронній спосіб є більш ефективним для ручного вирішення конфліктів. Серед текстових, синтаксичних та семантичних способів поєднання найбільш гнучкими у використанні є текстові способи, тоді як семантичні способи дозволяють реалізувати найкращу автоматизацію розв'язання конфліктів при поєднанні версій програмного продукту.

Запропонований спосіб складається із кількох рівнів абстракції за ППК:

4. Лексичний – для обробки конфліктів, що виникають всередині лексем і можуть бути тривіально розв'язані;
5. Синтаксичний – для обробки конфліктів, що виникають при поєднанні піддерев розбору та мають тривіальні правила для їх поєднання (для випадків створення, перейменування, перемішення або видалення певної сутності (терміналу);
6. Семантичний – для обробки конфліктів, що потребують семантичних даних для вирішення КП. В якості семантичних даних використано мультиграф залежностей програми з наступними модифікаціями:
    - додатковою інформацією про виклики функцій та переданих параметрів;
    - виявлення модифікації об'єкта, що був переданий за посиланням та збереженням інформації про модифікацію в ГЗП.

Кожен з рівнів є надбудовою над попереднім і являє собою повний додатковий обхід над ППК більш високого рівня.

На основі експериментальних даних показано ефективність інструменту поєднання, що працює за запропонованим способом.

ЛІТЕРАТУРА

1. Brun Y, Holmes R, Ernst MD, Notkin D (2013) Early detection of collaboration conflicts and risks. IEEE Trans Softw Eng 39:1358–1375

2. Kasi BK, Sarma A (2013) Cassandra: proactive conflict minimization through optimized task scheduling. In: Proceedings of the 2013 international conference on software engineering. ICSE ‘13. IEEE Press

3. Meld Help Page https://meldmerge.org/help/

4. J.W. Hunt and M.D. McIllroy, “An Algorithm for Differential File Comparison,” Technical Report 41, AT&T Bell Laboratories Inc., 1976.

5. B. Berliner, “CVS II: Parallelizing Software Development,” Proc. The Advanced Computing Systems Professional and Technical Association (SPNIK) Conf., pp. 22-26, 1990.

6. Git Merge Documentation https://git-scm.com/docs/git-merge

7. J. Buffenbarger, “Syntactic Software Merging,” Software Configuration Management: Selected Papers SCM-4 and SCM-5, J. Estublier, ed., pp. 153-172, 1995.

8. W. Yang, “How to Merge Program Texts,” J. Systems and Software, vol. 27, no. 2, pp. 129-135, 1994.

9. D. Jackson and D.A. Ladd, “Semantic Diff: A Tool for Summarizing the Effects of Modifications,” Int’l Conf. Software Maintenance, 1994.

10. Ferrante, J., Ottenstein, K., and Warren, J., “The program dependence graph and its use in optimization,” ACM Transactions on Programming Languages and Systems 9(3) pp. 319-349 (July 1987).

11. Badger, Lee and Weiser, Mark, “Minimizing Communication for Synchronizing Parallel Dataflow Programs,” Proceedings of the 1988 International Conference on Parallel Processing II, Software pp. 122-126 Penn State, (August 1988).

12. S. Apel and C. Lengauer, “Superimposition: A Language-Independent Approach to Software Composition,” in Proceedings of the International Symposium on

Software Composition (SC), ser. Lecture Notes in Com-puter Science, vol. 4954. Springer-Verlag, 2008, pp. 20–35.

13. Python Language Reference https://docs.python.org/3.10/reference/index.html

14. W.K. Edwards, “Flexible Conflict Detection and Management in Collaborative Applications,” Proc. Symp. ser Interface Software and Technology, 1997.