МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

Нгуен Карина Дамівна

Допускається до захисту:
завідувач кафедри
інформаційних технологій,
доктор технічних наук, доцент
_____Т. В. Нескородєва
«____» _____ 20__ р.

ТЕМА

«Розробка системи для статичного аналізу програмного коду»

Спеціальність 122 «Комп'ютерні науки»

Кваліфікаційна (бакалаврська) робота

Керівник:
Мартьянова Т. А.,
старший викладач кафедри
інформаційних технологій,
старший викладач

Оцінка:
___/___/_____
(бали за шкалою ЄКТС/за національною шкалою)
Голова ЕК:
_____
(підпис)

Вінниця – 2022

АНОТАЦІЯ

Нгуен К. Д. Розробка системи для статичного аналізу коду. Спеціальність 122 «Комп'ютерні науки», освітня програма «Сучасні інформаційні технології та програмування». Донецький національний університет імені Василя Стуса, Вінниця, 2022.

У кваліфікаційній роботі досліджено метод статичного аналізу, який можна використовувати як додаток до тестування для автоматичного виявлення певних проблем програмування.

Ключові слова: статичний аналіз, програмне забезпечення, програма, інструмент, код, COBOL, помилка, алгоритм.

59 с., 2 табл., 17 рис., 17 джерел.

Nhuen K. D. Development of a system for static analysis of program code. Specialty 122 "Computer Science", Educational Program "Modern Information Technology and Programming".Vasyl Stus Donetsk National University, Vinnytsia, 2021.

In the qualification work investigates the method of static analysis, which can be used as an adjunct to testing to automatically detect certain programming problems.

Keywords: static analysis, software, program, tool, code, COBOL, error, algorithm.

59 pp., 2 tables, 17 figures, 17 sources.

ЗМІСТ

ВСТУП

Історично, програми перевіряються за допомогою інструментів для відстеження виконання програми та спостереження за поведінкою виконання коду. Використання таких інструментів утомлює, оскільки вимагає побудови тестових даних для кожного з можливих випадків. Кожен із цих тестових випадків необхідно запустити вручну, щоб перевірити поведінку програми на цих даних.

Забезпечення правильної роботи програм у різних умовах використання завжди було важливою метою розробників програмного забезпечення та керівників проектів. Мови програмування з чітко визначеними типами даних і все більш складними компіляторами значно допомогли покращити якість програм, виявивши кілька семантичних помилок.

Різні методи статичного аналізу являють собою наступний крок з точки зору інструментів, які допомагають розробці програм. Ці інструменти дозволяють автоматично виявляти декілька класів програмних помилок без виконання програм.

У цьому документі представлено метод статичного аналізу, який можна використовувати як додаток до тестування для автоматичного виявлення певних проблем програмування.

У роботі представлено статичний аналіз, коротко описується сфера застосування цих методів, і читач знайомиться з деталями впровадження методу статичного аналізу.

Компанія, яка бажає використовувати статичний аналіз як інструмент допомоги розробці, потрібно прийняти кілька рішень.

Перше рішення полягає в тому, щоб чітко визначити проблему або проблеми, які потрібно виявити. Вибрана проблема або проблеми допоможуть вибрати представлення даних. Вибір адекватного представлення дає змогу спростити виявлення цільових аномалій.

Друге рішення зводиться до вибору методу перетворення вихідної програми. На цьому етапі дуже корисні методи компіляції, оскільки вони забезпечують кілька алгоритмів перетворення коду, які можна використовувати для статичного аналізу.

Останнє рішення, яке необхідно прийняти — це визначити формат подання діагнозу, виробленого інструментом аналізу. Коли програма була перетворена, на внутрішньому представленні запускається алгоритм статичного аналізу і виявляються помилки. Виявлення можливої помилки є метою інструментів аналізу. З іншого боку, щоб бути дійсно корисними, інструменти аналізу повинні виробляти діагностику, яка допомагає визначити джерело цих аномалій і спрямовувати користувача до їх вирішення.

Тип компанії, яка слугує моделлю і веде до написання цієї праці, бореться з кількома проблемами, які впливають на якість створеного коду. Основна комп'ютерна система була написана на початку 1990-х років командою близько двадцяти чоловік. Ця система, яка спочатку мала керувати однією лінійкою продуктів, з часом була модифікована, щоб дозволити маркетинг трьох нових ліній продуктів. Більшість людей, які допомагали створити оригінальну систему, сьогодні більше не працюють у компанії. Така ситуація зустрічається у великій кількості компаній, які експлуатують власні системи. Як правило, ці системи досить великі, оскільки містять кілька мільйонів рядків коду.

Численні зміни, внесені в ці системи, у поєднанні з часто неадекватною документацією та швидкістю плинності кадрів вимагають більш глибокої перевірки коду, щоб переконатися, що він працює належним чином. З іншого боку, програмісти часто змушені жертвувати тривалістю та якістю тестів, щоб укластися в терміни. З цієї причини користувачі системи на виробництві виявляють кілька несправностей. Наприклад, близько п'яти років тому кількість активних запитів на виправлення, зафіксованих користувачами системи, становила близько півсотні. Наразі користувачами системи зареєстровано понад 350 активних запитів на виправлення, і щодня надходить близько півсотні нових

запитів. Крім того, кілька транзакцій ненормально припиняються щодня під час виробництва, а відкладена обробка, яка виконується вночі, також стикається з подіями, які потребують віддаленого втручання на щоденній основі. Аналіз виниклих інцидентів виявляє декілька недоліків, які часто зустрічаються: використання змінних, які не завжди ініціалізовані, і часте переповнення таблиць у пам'яті.

Щоб впоратися з все більш актуальною проблемою якості, необхідно полегшити роботу програмістів за рахунок скорочення часу перевірки та полегшення виявлення найбільш часто зустрічаються випадків помилок. Основна мета есе — продемонструвати, як статичний аналіз коду можна використовувати для вирішення цієї проблеми.

Метою аналізу програми є автоматичне визначення властивостей програми. Інструменти розробки, такі як компілятори, засоби перевірки програм і засоби для розуміння програм, значною мірою покладаються на статичний аналіз. Есе продемонструє роботу статичного аналізу досить детально, щоб дозволити реалізувати представлені алгоритми. Також містить посилання на відповідні джерела для повного теоретичного введення, але не претендує на те, щоб чітко продемонструвати всі положення використаної теорії.

Доступні посилання зі статичного аналізу дуже докладні з теоретичної точки зору, але дуже мало торкаються аспекту реалізації. Написана робота має на меті заповнити цю прогалину, зосередивши увагу на трьох аспектах, які не розглядаються в літературі: як використовувати статичний аналіз для виявлення більш ніж одного типу проблеми одночасно, як створити інструмент статичного аналізу для виявлення цих інцидентів та проведення діагностики виявлених явищ.

Для висвітлення теоретичного аспекту та вибору алгоритму аналізу в есе використовується спеціалізована література. Потім цей алгоритм пояснюється детально. У цьому сенсі використані посилання добре відомі в області статичного аналізу. Есе починається з демонстрації важливості інструментів

статичного аналізу в середовищі, де витрати на розробку та виправлення помилок дуже високі.

Перші кроки статичного аналізу спрямовані на перетворення вихідного коду програми в проміжне уявлення, яке може використовуватися аналізатором. Це проміжне представлення складається з трьох частин: коду, графіка потоку керування та змінних програми. Це проміжне уявлення описано детально.

Також детально описано обраний для тесту алгоритм. Вибір даного алгоритму був зроблений тому, що він полегшує обробку програм, поділених на функції, шляхом виконання аналізу функцій. Результат аналізу функцій стає абстракцією для кожної з цих функцій.

Цю абстракцію потім можна використовувати для аналізу програм, які використовують ці функції.

Алгоритм складається з трьох кроків. Перший крок — згрупувати інструкції в основні блоки. Другий крок має на меті побудувати ієрархію регіонів із цих базових блоків. Нарешті, аналіз виконується прогресуючими базовими регіонами, поки програма не буде проаналізована.

Нарешті, тест показує, як аналіз може поставити діагноз при виявленні помилки. Постановка діагнозу вимагає включення якомога більше інформації, щоб допомогти знайти та виправити виявлені явища.

РОЗДІЛ 1. ВВЕДЕННЯ В СТАТИЧНИЙ АНАЛІЗ

Люди постійно роблять помилки: забута крапка з комою, зайві дужки, неправильно написане ім'я змінної. У більшості випадків ці помилки не мають жодного значення: компілятор виявляє їх, програміст виправляє код, і цикл розробки продовжується. Цей швидкий зворотний зв'язок з компілятором різко контрастує з помилками, які компілятор не виявляє. Деякі з цих помилок [18] можуть залишатися в коді (можливо, протягом кількох років), перш ніж проявитися. Чим більше часу потрібно для виявлення помилки, тим дорожче її виправити.

1.1 Опис статичного аналізу

Статичний аналіз включає аналіз тексту програми для вилучення інформації. Цей аналіз виконується без запуску програми. Таким чином, статичний аналіз відрізняється від динамічного аналізу, який полягає у виконанні програми з даними для перевірки її поведінки. Тип отриманої інформації, а також обрані методи залежать від використання, для якого призначений аналіз. Статичний аналіз використовується для виявлення помилок програмування або проектування, а також для визначення того, наскільки легко чи складно підтримувати код.

Не завжди можливо автоматично виявити всі помилки програмування. Дослідницька робота Алана Тьюринга продемонструвала в першій половині 20 століття, що для будь-якої задачі на результат механічного розрахунку не існує автоматизованого методу вирішення цієї проблеми, якщо він не є тривіальним. Алан Тьюринг [16] описав поняття алгоритму, визначивши машину Тьюринга. Потім він використав це визначення, щоб продемонструвати, що існують нерозв'язні випадки, тобто проблеми, які не мають алгоритму. Цей момент можна проілюструвати на прикладі, показаному на рисунку 1.1.

```
PROBLEME (ch) {
 Si TERMINE (ch) alors faire {…} tant que (vrai);
}
```

Рисунок 1.1 Програма PROBLEME ().

У цьому прикладі слід припустити, що функція COMPLETE() існує. Ця функція приймає як вхідний рядок символів, що представляє програму та її повертає в результаті логічне значення, яке має значення «true», якщо програма, отримана як вхід, завершується, і значення «false», якщо вона не завершується.

Протиріччя виникає, коли програма, передана як параметр до програми PROBLEME, є самою програмою PROBLEME. Дійсно, якщо функція TERMINATE повертає логічне значення "true", це означає, що функція дійшла висновку, що програма закінчується; програма ПРОБЛЕМА потім увійде в нескінченний цикл, що суперечить результату функції DONE. З іншого боку, якщо функція DONE повертає логічне значення «false» означає, що програма не завершується, але в цьому випадку ми не входимо в нескінченний цикл і програма завершується, що також суперечить результату функції COMPLETE. Тому неможливо мати функцію TERMINATE, яка може визначати для всіх програм і безпомилково, чи можна завершити цю програму чи ні.

Аналіз програм, [19] включаючи пошук можливих помилок під час виконання, тому не піддається розгляду: немає методу, який завжди міг би відповісти, не помиляючись, чи може програма створювати помилки під час виконання.

Хоч і неможливо виявити всі помилки, присутні в усіх програмах, це не привід повністю відмовлятися від методів статичного аналізу. У більшості випадків можна виявити велику кількість наявних проблем. Потрібно використовувати методи, які досить добре працюють у більшості реальних програм. Тому це приблизні методи. Ці наближені методи не є недоліком строгості. Використовувані методи повинні бути надійними, але не обов’язково оптимальними.

Оскільки неможливо виявити всі помилки, присутні в усіх програмах, інструменти статичного аналізу стикатимуться з ситуаціями, коли вони не можуть точно визначити, чи є помилка чи ні. Тому всі інструменти ризикують

повідомляти про інциденти, коли їх немає, або, навпаки, не можуть виявити помилки, які присутні в коді.

Однією з найпоширеніших скарг щодо інструментів статичного аналізу, за словами Браяна Чеса [3], є те, що ці інструменти дають занадто багато помилкових спрацьовувань, також відомих як помилкові тривоги. У цьому контексті хибнопозитивним є інцидент, виявлений, коли насправді проблеми не існує. Велика кількість помилкових результатів може стати перешкодою для використання інструментів статичного аналізу. Перегляд дуже довгого списку помилкових спрацьовувань може бути дуже неприємним для програміста, навіть перешкоджати використанню цих інструментів. Крім того, програмісти можуть упустити реальні проблеми, приховані в цьому списку помилкових спрацьовувань. Помилковий негатив – це існуюча несправність, яку скануючий інструмент не виявляє.

Усі інструменти сканування дають кілька хибнопозитивних або кілька помилково негативних результатів. Деякі виробляють обидва. Інструменти аналітики націлені на певний баланс між ними залежно від призначення інструмента. Витрати, понесені через невиявлену програмну помилку, відносно невеликі в порівнянні з витратами, які можуть бути понесені під час вторгнення. З цієї причини інструмент, спрямований на виявлення помилок програмування, намагається зменшити кількість помилкових спрацьовувань і готовий прийняти деякі помилкові негативи. З іншого боку, інструменти, [20] спрямовані на виявлення проблем із безпекою, радше погодяться створювати більше хибнопозитивних результатів, щоб зменшити ймовірність виникнення помилкових негативів.

Існує два основних сімейства формальних статичних аналізів програм: перевірка моделі та абстрактна інтерпретація.

1.1.1 Перевірка моделі

Перевірка моделі полягає в алгоритмічній перевірці, чи відповідає дана модель специфікації. У цьому контексті модель — це сама система або абстракція системи. Використовувана специфікація часто формулюється в термінах тимчасової логіки.

Це сімейство аналізу полягає в абстрагуванні моделі, яка відображає ту саму поведінку, що й вихідна система, але деякі аспекти якої були опущені для спрощення моделі. Цей тип аналізу часто використовується для перевірки конкуруючих систем.

Коли систему неможливо перевірити повністю, метод верифікації моделі вимагає побудови спрощеної моделі, яка зберігає основні характеристики системи [10]. Модель має бути побудована вручну, і цей крок є критичним, оскільки помилка під час побудови моделі може зробити результати аналізу недійсними. Після того, як модель побудована, її можна перевірити шляхом моделювання всіх можливих випадків, навіть якщо цей метод був неможливим із самої програми.

1.1.2 Абстрактна інтерпретація

Відповідно до Кузо [4], абстрактну інтерпретацію можна визначити як часткове виконання програми для отримання інформації про її семантику, наприклад, про її структуру керування або про потік даних, без необхідності її обробки. Програма позначає серію процесів у даному всесвіті значень або об’єктів. Абстрактна інтерпретація полягає у використанні цього позначення для опису обробки в іншому всесвіті абстрактних об’єктів, щоб результати абстрактної інтерпретації могли надати інформацію про конкретні обчислення програми.

Абстрактна інтерпретація дає можливість суворо міркувати про програми, роблячи можливими наближення. З огляду на мову програмування або специфікації, абстрактна інтерпретація полягає у визначенні семантики, пов’язаної відношеннями абстракції.

Найточнішою семантикою, природно, є та, яка дуже точно описує фактичне виконання програми. Ця семантика називається конкретною семантикою. Наприклад, конкретна семантика імперативної мови програмування може асоціювати з кожною програмою повний набір слідів, які вона створює. Трасування виконання – це послідовність можливих послідовних станів виконання програми. Стан складається із значення використовуваних змінних. Більш абстрактну семантику можна вивести з конкретної семантики.

Щоб зробити можливим проведення статичного аналізу програми, з неї виводиться певна обчислювана семантика. Наприклад, можна вибрати представлення змінних програми за їх станом. Цей стан можна [21] вибрати з наступних станів: ініціалізований або невизначений. Те саме стосується інструкцій, представлених дією, що виконується щодо змінних: встановити, прочитати, посилатися, змінити або звільнити.

Цей рівень абстракції не втрачає точності для аналізу, який бажає знайти інструкції, які використовують неініціалізовані змінні. Для інших операцій абстракція втрачає точність. Неможливо визначити, чи доступ до запису в таблиці знаходиться в межах, які визначають цю таблицю. Таких втрат точності, загалом, неможливо уникнути, якщо зробити семантику доступною для вирішення. Існує компроміс між точністю аналізу та його здійсненністю з точки зору обчислюваності або складності.

1.2 Визначення проблем

Статичний аналіз використовується частіше, ніж більшість людей думає. Головним чином тому, що існує кілька типів статичного аналізу. В розділі 2 описані основні види використання статичного аналізу.

Мета цієї праці — детально продемонструвати, як статичний аналіз може бути використаний для виявлення помилок при розробці програми. Тому тест обмежується виявленням частих помилок програмування за допомогою підходу абстрактної інтерпретації. Вибрані типи помилок програмування дуже прості, а

частини коду, які використовуються як приклади, дуже короткі, щоб забезпечити більш детальний аналіз, який може скеровувати до можливої реалізації представлених методів.

Помилки програмування, які розглядаються в тесті, бувають двох типів, а саме: використання неініціалізованих змінних і перевірка доступу до елементів масиву. Обидва ці типи помилок є досить поширеними в середовищах процедурного програмування. Будь-який код, [22] навіть створений досвідченим програмістом, може містити аномалії. Деякі з цих проблем виявляються та виправляються під час модульного тестування. Інші інциденти виявляються та виправляються під час офіційного тестування якості. Процес визначення належного рівня перевірки є суб'єктивним. Навіть нескінченна кількість тестів не може забезпечити відсутність помилок у програмі. Однак, чим більше людина перевіряє програму на наявність проблем, тим більша ймовірність її знайти. Кількість ситуацій для перевірки складної програми може бути практично нескінченною.

Основна причина перевірки програм із проблемами полягає в тому, що виявити та виправити всі проблеми інколи не вигідно, навіть якщо б це було можливо. За Бернштейном [2], найпоширенішим способом перевірки якості системи є перевірка її в кінці циклу розробки. Виконання тестів вимагає ресурсів і хоча покращує якість коду, оскільки кількість можливих ситуацій наближається до нескінченності, час і ресурси для виконання цих перевірок також стають нескінченними. Перевірка якості за допомогою тестових випадків дуже неефективна. Оскільки в більшості випадків кількість випадків, які підлягають перевірці, занадто велика, проведені тести охоплюють лише частину ситуацій, з якими можна зіткнутися, що потенційно залишає помилки, які виникають після доставки продукту.

Попередження дефектів не тільки можливо, але й необхідно при розробці програмного забезпечення. Однак, щоб запобігання дефектам було ефективним, потрібен формалізований процес інтеграції цієї стратегії в цикл розробки. Цей

формалізований підхід має включати застосування найкращих галузевих практик, щоб уникнути поширених проблем. Більше того, цей підхід має підтримуватися адаптованою інфраструктурою, яка автоматизує кілька повторюваних завдань, щоб поширювати практику запобігання дефектам від одного проекту до іншого. Не слід відмінювати випробування. Процес [23] перевірки наприкінці циклу розробки не є ефективним методом покращення якості продукції. З іншого боку, тестування може і повинно використовуватися для перевірки якості продукції.

Використання інструментів аналізу, спрямованих на виявлення якомога більшої кількості аномалій програмування, автоматично звільняє людські ресурси, які можуть бути спрямовані на виявлення інших типів помилок і таким чином покращуючи якість програм, що поставляються. Таким чином, ці інструменти є основою процесу розробки і дозволяють зменшити кількість дефектів на ранніх етапах процесу. Ці інструменти покращують якість продукту за дуже низьку вартість. Реалізація інструменту аналізу не дуже дорога: інструмент для виявлення аномалій, описаних у есе, був написаний за 2 місяці. Після того, як використання такого продукту інтегровано в процес розробки, періодичні витрати дуже низькі.

Статичний аналіз можна застосувати до різних типів мови: процедурної, об'єктно-орієнтованої, функціональної чи логічної. Дослідження [28] охоплює лише процедурні мови, в основному орієнтуючись на COBOL у прикладах, хоча продемонстровані методи не обмежуються лише цією мовою.

Деякі сучасні компілятори дозволяють виявляти певні помилки програмування, які інші компілятори не виявляють. Деякі компілятори можуть генерувати код для виявлення переповнень таблиці під час виконання, але це параметр, який часто пропускають, щоб підвищити швидкість виконання. На ринку все ще є багато коду, розробленого на COBOL. Компілятори COBOL, хоча і надійні, не мають усіх можливостей, які пропонуються компіляторами для останніх мов.

«У всьому світі використовується 180-200 мільярдів рядків коду COBOL. Більшість транзакцій у цьому світі засновані на Cobol. Щороку пишуться мільярди рядків важливого для бізнесу коду Cobol. »2 ([12], с. 99)

Тому важливо не нехтувати основними методами статичного аналізу, які згодом можна поширити на виявлення інших помилок програмування. Вибір мови COBOL ґрунтується на особистому досвіді з цією мовою, об'ємі існуючих програм COBOL і тому факті, що мова [29] COBOL розроблена спеціально для фінансових додатків, і здається, що жодна інша мова не підходить так добре для потреби цих додатків.

Роберт Л. Гласс [9] згадує чотири характеристики, які вважаються важливими для фінансових додатків: керування неоднорідними структурами даних, виконання точної десяткової арифметики, легке створення звітів, а також легка обробка дуже великої кількості даних. Більшість цих можливостей відсутні в сучасних мовах програмування, за винятком мови COBOL, яка є адекватною для всіх чотирьох функцій.

РОЗДІЛ 2. АБСТРАКТНА ІНТЕРПРЕТАЦІЯ

Абстрактна інтерпретація ґрунтується на визначенні абстракції семантики програми, написаної даною мовою. Ця абстракція програми використовує менш складний вигляд коду, ніж вихідна програма, що спрощує її аналіз. Пропоноване використання абстракції жертвує інформацією під час процесу абстракції. Таким чином, рішення проблеми, засноване на абстрактній інтерпретації, може не бути рішенням вихідної проблеми. Отже, мистецтво аналізу символьної програми полягає в розробці відповідної абстракції для даної програми, яка дає змогу розв'язати цей тип аномалії.

Метою статичного аналізу є автоматичне визначення певних властивостей системи. Вихідний код — це опис специфікації ситуації, яку програма призначила для вирішення. Інструкції програми утворюють конкретну модель, що представляє цю специфікацію. Аналізуючи вихідний код як він є, можна проаналізувати всі спостережувані властивості в ньому. Ці властивості можуть бути настільки різноманітними, як і визначення часу, витраченого на обробку ітерацій, забезпечуючи доступ до змінної лише тоді, коли [24] ця змінна ініціалізована, підтверджуючи, що межі масиву дотримуються при зверненні до його елемента.

Існує багато спостережуваних властивостей залежно від потреб аналізу. Зазвичай властивості, що представляють інтерес для даної програми, складаються з підмножини всіх спостережуваних властивостей. Тому в цьому випадку корисно спростити відображення поведінки програми, щоб полегшити обробку аналізу. Наприклад, якщо мета аналізу полягає в тому, щоб спостерігати за поведінкою програми щодо числових змінних, щоб переконатися, що не може відбутися поділ на нуль, немає сенсу розглядати інструкції, які не впливають на числові змінні. Тому в рамках цього аналізу можна ігнорувати інструкції, що виконуються над рядками символів.

Залежно від властивостей, за якими які слід спостерігати, перший крок абстрактної інтерпретації полягає у виборі абстракції, яка буде

використовуватися для спрощення програми, що аналізується. Як вхідні дані програма має інструкції, які мають на меті виконати обробку, для якої вона була розроблена. Потім ця програма перетворюється, а аналіз здійснюється на спрощеній версії програми. Ця спрощена версія являє собою абстракцію оригінальної програми. Вибір абстракції повинен бути таким, щоб властивості, які спостерігаються в цій спрощеній версії, були ідентичними тим, які були б отримані, якби детальний аналіз проводився в оригінальній програмі.

У цій главі абстрактне тлумачення вперше вводиться неформально за допомогою прикладу. Згодом формулюється більш суворий опис абстрактної інтерпретації. Нарешті, модель абстракції вибирається для вирішення двох типів інцидентів, на які спрямований тест: забезпечення ініціалізації [25] змінної перед посиланням на неї та перевірка меж таблиць при доступі до одного з елементів масиву.

2.1 Неформальне визначення

Абстрактне тлумачення вводиться на дуже простому та інтуїтивно зрозумілому прикладі. Цей приклад походить зі статті Кузо [6], яка сама заснована на статті Мішеля Сінцоффа [15]. У статті Сінцоффа автор використовує знакову арифметику як модель абстракції. У наведеному тут прикладі властивість для спостереження є ознакою результату.

Текст «-1515 * 17» може позначати обчислення в абстрактному всесвіті $\{(+), (-), (\pm)\}$, де семантика арифметичних операторів визначається правилом знаків.

Абстрактний цикл -1515*17→ -(+)*(+)→ (-)*(+)→ (-)доводить, що « -1515 * 17» — від’ємне число. Абстрактну інтерпретацію цікавить конкретна властивість універсуму обчислень (знак результату в наведеному прикладі). Абстрактна інтерпретація дає короткий опис одного аспекту виконання програми. Отже, -1515+17→ -(+) +(+)→ (-)+(+)→ (±), що означає, що тільки за знаком операндів неможливо визначити знак результату.

Попередній приклад продемонстрував, що застосування абстракції дає змогу спростити аналіз програми. У наступному розділі представлено більш формальне визначення інтерпретації абстрактної програми.

2.2 Формальне визначення

Статичний аналіз програм складається з абстрактної оцінки цих програм. Під час абстрактного оцінювання програми абстрактні значення асоціюються зі змінними замість конкретних значень, які були б отримані під час виконання програми. Основні мовні операції інтерпретуються відповідно до обраної абстракції. Абстрактна інтерпретація тоді полягає в спостереженні за операціями програми над цими абстрактними значеннями. Можна використовувати будь-який набір абстрактних значень за умови, що ці значення дотримуються наступної властивості: набір абстрактних значень повинен дозволити аналізу досягти стабільного стану після кінцевої кількості ітерацій.

Інтерпретація основних інструкцій мови, а також вибір абстракції значень залежать від динамічних властивостей, які витягуються з програми. Оскільки вибір абстракції відповідає певним властивостям, метод аналізу, представлений у главі 4, дає можливість гарантувати, що аналіз закінчиться кінцевою кількістю ітерацій і що результат аналізу є правильним, хоча він може містити деякий ступінь невизначеності.

Набір абстрактних значень повинен утворювати впорядкований набір значень, який називається решіткою. У Додатку 3 представлені деякі властивості теорії множин, корисні для обговорення абстракції.

Абстракція програми представлена в рамках перевірки прав доступу до елементів масиву. Змінна, яка використовується для індексації масиву, може приймати одне або кілька цілих значень. Набір значень, які може приймати ця змінна, буде абстрагуватися в інтервал значень. Нижня межа інтервалу представляє найменше можливе значення, а верхня межа інтервалу представляє найбільше значення.

Теорія, що використовується для абстрактної інтерпретації, багато в чому базується на роботах Патріка Кузо [5].

2.3 Перевірка доступу до таблиці

У цьому розділі використання інтервалу значень для перевірки доступу до елементів масиву взято зі статті Патріка Кузо [5], яка представлена в додатку 4.

Вибір відповідної абстракції здійснюється з урахуванням мети аналізу. Щоб виявити доступ до масиву поза його межами, достатньо визначити діапазон можливих значень змінних, які використовуються для індексації записів цього масиву. Для цього типу ситуації можна використовувати діапазони значень як область абстрактних значень для числових змінних.

Таким чином, набір V_C конкретних значень є набором цілих чисел Z (між межами $-\infty$ і $+\infty$) і V_A абстрактних значень - це набір інтервалів цілих чисел позначається [a, b], де $a \in Z$, $b \in Z$ і $a \leq b$.

Функція конкретизації g визначається так:

$$g : V_A \rightarrow V_C,\ g([a, b]) = \{x \mid (x \in Z) / (a \leq x \leq b)\}$$

Функція a використовується для отримання інтервалу значень з набору цілих чисел. Функція g повертає набір цілих чисел, які є частиною інтервалу. З абстрактним значенням [3, 34] функція g повертає множину елементів, що входять до інтервалу, тобто множину {3, 4, 5, …, 33, 34}.

Виявляється, що набір конкретних значень містить більше значень, ніж вихідний набір. У цей момент з’являється неточність, оскільки набір конкретних значень, які походять від g(α(e)) відрізняється від e. Важливо переконатися, що ця неточність не вплине на валідність аналізу. Граничні значення інтервалу завжди виходять із набору конкретних значень за визначенням, наведеним вище. Більше того, єдиними значеннями, які важливо зберегти, щоб мати можливість перевіряти доступ до елементів масиву, є межі інтервалу. Отже, аналіз дійсний, незважаючи на неточність,

внесену функцією g.

На малюнку 2.1 показана частина ферми. Ця решітка представляє абстракцію на основі інтервалів значень. Ця решітка містить інтервали в порядку зростання від нижньої частини малюнка до верху. Чим вищий інтервал на малюнку, тим він вищий за порядком c. Операція (ілюструється шляхом знаходження перетину прямих, що виходять із двох інтервалів, наприклад [2,3] ([4,6] - це інтервал [2,6], який знаходиться на перетині лінії, що з'єднує інтервал [2,3] та інтервал [5,6] Це перший інтервал, який включає обидва інтервали.

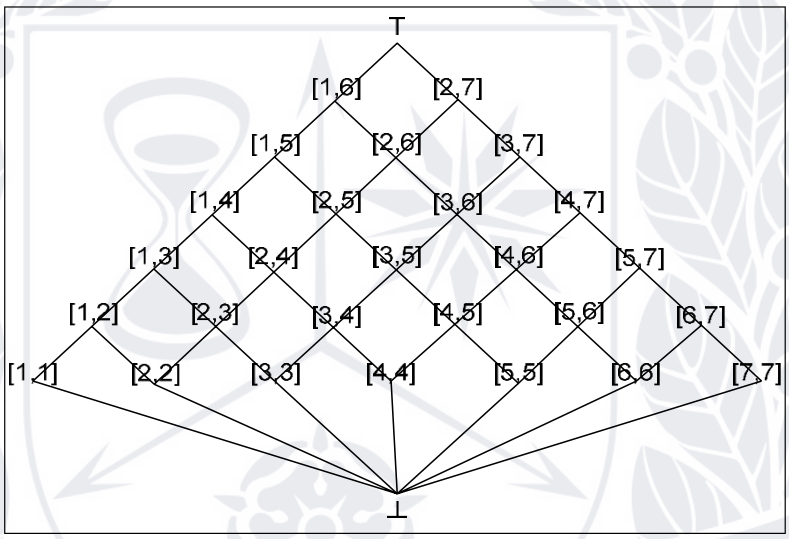

Рисунок 2.1 Часткова решітка для інтервалів цілих значень

Нижній рядок на малюнку 2.1 містить елемент «⊥», що вказує на змінну без значення. Другий рядок представляє конкретні значення, які є цілими числами. Ці значення представлені у вигляді інтервалу, що містить лише одне значення. Кожен рядок представляє все більші і більші інтервали до верхнього рядка, який представляє нескінченний інтервал [-∞, +∞].

2.4 Доступ до змінної

У попередньому розділі була обрана абстракція для представлення числових змінних, які використовуються для доступу до елементів масиву. Метою тесту є також виявлення випадків використання змінної до її ініціалізації.

Для цієї другої проблеми необхідно визначити, чи має змінна значення чи ні. Для числових змінних ми вже маємо значення «⊥», яке вказує на те, що змінна не має значення та діапазон значень для представлення інших ситуацій. Для інших змінних програми домен абстрактних значень може складатися з двох значень:

«⊥» вказує, що змінна не має значення, а «d» вказує, що змінна визначена.

Враховуючи, що мова COBOL в основному використовується для обробки інформації, що надходить з файлів, можна додати значення в область абстрактних значень {⊥, d}, щоб мати можливість провести більш повний аналіз. Для цих цілей можна використовувати деякі значення, запропоновані Као і Ченом [11]. Таким чином, домен, який використовується для змінних, може стати {⊥, d, f, o, c, T}. Значення «d» вказує, що змінна визначена, «f» вказує, що значення було отримано з файлу, «o» вказує, що значення змінної походить з файлу і що цей файл відкритий, але немає записів було прочитано, тому змінна ще не визначена, і, нарешті, "c" вказує, що файл закритий, а відповідна змінна більше не визначена. Область може бути розширена, щоб відобразити всі ситуації, які потрібно охопити, залежно від властивостей, які необхідно спостерігати.

У тесті достатньо визначити, чи ініціалізована змінна під час її використання. Область абстракції {⊥, d, T} використовується для підтвердження того, що змінна ініціалізована перед її використанням. Значення «T» вказує на те, що в даній контрольній точці неможливо дізнатися, ініціалізована змінна чи ні.

РОЗДІЛ 3. ВНУТРІШНЄ ПРЕДСТАВЛЕННЯ ПРОГРАМИ

Інструменти сканування зазвичай працюють з вихідного коду програм. Ці програми трансформуються, щоб витягти їх синтаксичні структури та полегшити обробку. У цій главі описується представлення, що використовується в цьому тесті, але не вказуються деталі реалізації.

Програма складається з інструкцій та даних, на які діють ці інструкції. Інструкції бувають двох типів: інструкції перетворення даних та інструкції, що керують послідовністю виконання програми. У першій частині розділу йдеться про інструкції з перетворення даних. Інструкції, які використовуються [26] для визначення послідовності виконання обробки, представлені нижче. Представлення даних розглядається в кінці розділу.

3.1 Трансформація програми

Статичний аналіз виконується за певною програмою, наданою в певному форматі. Більшість інструментів аналізу коду обробляють програми так само, як компілятори, тобто з вихідного коду. Малюнок 3.1 ілюструє етапи компілятора, а малюнок 3.2 ілюструє етапи інструмента статичного аналізу. Ці два малюнки демонструють, що інструмент статичного аналізу виконує майже таку ж роботу з обробки вихідного коду, як і компілятор.

Перші чотири етапи обробки спрямовані на перетворення програми з вихідного коду на внутрішнє представлення. Подання програми введення у форматі, яким легко маніпулювати людиною. Метою перетворення вихідної програми є створення представлення, яким можна було б легше маніпулювати програмою.

Перетворення вихідного коду в проміжний код за допомогою інструменту статичного аналізу ідентично тому, що робить компілятор. Коли отримано проміжний код, обробка інструменту аналізу більше не має нічого спільного з компілятором. Компілятор повинен створювати інструкції визначеною

машинною мовою. Ці інструкції повинні давати точне представлення конкретної семантики програми.

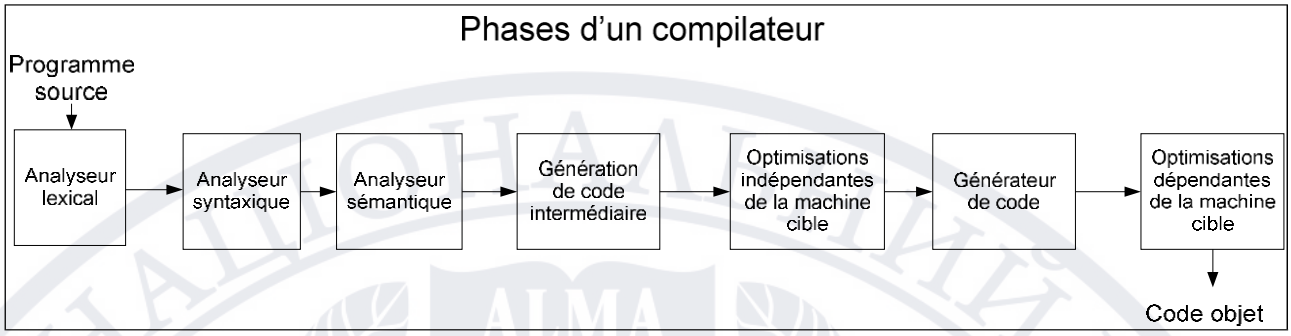

Рисунок 3.1 Фази складання.

Інструмент статичного аналізу не зберігає повну семантику програми, він зберігає достатньо інформації для постановки діагнозу потенційних виявлених інцидентів. Проміжне уявлення повинно мати можливість маніпулювати абстрактними значеннями. Компілятор створює код, який маніпулює конкретними значеннями.

У тесті перші чотири фази на малюнках 3.1 і 3.2 не розглядаються. Ці фази дуже добре висвітлені в літературі про компіляторів, включаючи книгу Ахо, Лама, Сеті та Уллмана [1]. Четверта фаза, хоча вона має ту саму назву для компілятора, що й для інструмента статичного аналізу, може давати різний результат залежно від інструменту. Результатом є внутрішнє представлення програми, і це представлення є темою цього розділу.

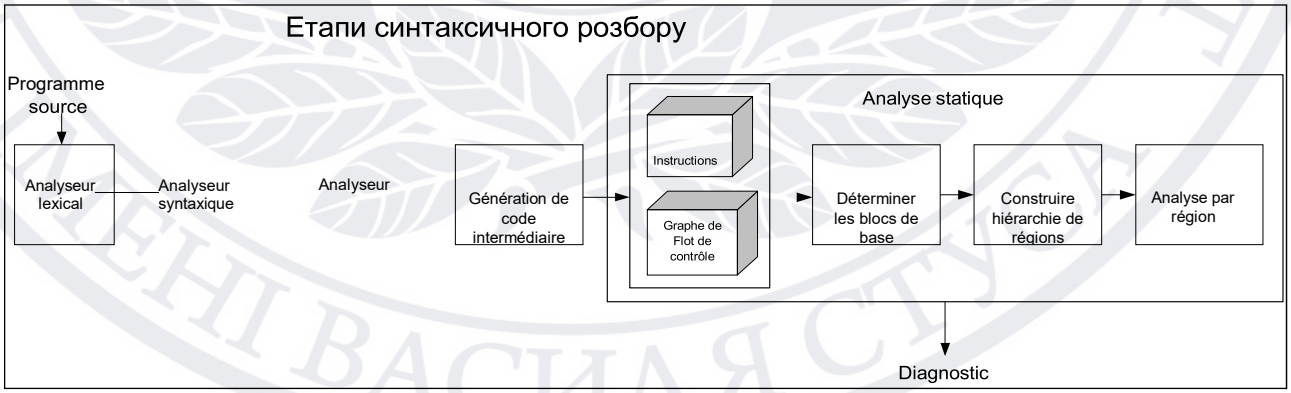

Рисунок 3.2 Фази статичного аналізу

Проміжний крок генерації коду створює представлення програми. Це подання легше використовувати в статичному аналізі, ніж об'єктний код. Якщо проміжне представлення програми не залежить від мови оригіналу і є достатньо

загальним, то це проміжне представлення дозволяє повторно використовувати модуль статичного аналізу для інших мов. Оскільки процедурні мови та об’єктно-орієнтовані мови після компіляції можуть виконуватися на одному процесорі, можна отримати проміжне уявлення, яке враховує цю особливість. Процесор не знає поняття об'єктів або структури даних. Процесор має інструкції маніпулювання даними, інструкції для керування потоком виконання програми та ряд елементарних даних, які йому відомі. Однак для досягнення цього необхідно враховувати відмінності в представленні об’єктів і викликах методів, наприклад віртуальних методів, які ускладнюють визначення методів, які фактично використовуються під час виклику.

Оскільки компілятори та інструменти статичного аналізу мають багато спільного, логічно покладатися на теорію компілятора, щоб прийняти рішення про внутрішнє представлення програм, яким легко маніпулювати. Проміжна мова, запропонована в цьому есе, включає кілька ідей, взятих із книги Ахо, Лама, Сеті та Уллмана [1], зокрема використання внутрішніх інструкцій фіксованої довжини, групування інструкцій у блоки, а також представлення управління потоком за допомогою графіка.

3.2 Інструкції з маніпулювання даними

Запущена програма отримує доступ до даних, вона зберігає ці дані у вигляді значень, призначених змінним, далі перетворює вміст цих змінних і, нарешті, отримує результат цих перетворень. Метою статичного аналізу є спостереження за певними властивостями під час маніпуляції з даними.

Оригінальна програма для аналізу написана з метою маніпулювання конкретними значеннями. Абстракція, обрана для аналізу, має на меті полегшити аналіз шуканих властивостей. Тому оригінальні інструкції повинні бути замінені інструкціями, які можуть діяти на вибрану абстракцію, а не на конкретні значення вихідної програми.

Хоча інструкції змінюються під час перетворення вихідного коду, отримані інструкції продовжують використовувати значення для створення нових значень. З іншого боку, ці значення є абстрактними цінностями, а не конкретними значеннями.

3.2.1 Теги

Програма розпізнає ряд змінних, і ці змінні мають тип, який залежить від мови або процесора. Список цих змінних, включаючи їх тип і атрибути цих змінних, створюється під час аналізу програми та використовується під час аналізу.

3.2.2 Програмні стани

Стан пам’яті, також званий станом, — це зв’язок між змінними та значеннями. Наприклад, даний стан може пов’язати значення 1 зі змінною «I» після інструкції ініціалізації. Стан змінюються інструкціями програми. Для статичного аналізу стан повинен представляти всі змінні програми з їх абстрактним значенням. Наприклад, програма з двома змінними «x» і «y» може мати стан {(x, [1, 2]), (y, [-10, 20])}. Цим позначенням позначають множину, що складається з двох пар. Кожна з пар (x, [1, 2]) і (y, [-10, 20]) представляє зв’язок між змінною та її абстрактним значенням. Для процедурної мови тип змінної не є обов’язковим у стані, оскільки цей тип не змінюється під час виконання програми, і тоді було б зайвим повторювати цей тип у всіх станах, де змінна знайдена. Тип змінної доступний зі списку змінних, створених під час перетворення вихідної програми. Для об’єктно-орієнтованої мови тип об’єкта має бути частиною стану, оскільки тип об’єкта може змінюватися під час виконання.

Вираз з оператора - це в основному функція для отримання значення зі значень змінної даного стану. Наприклад, ми можемо розв’язати вираз «x + y», використовуючи значення змінних «x» і «y» зі стану, що передує оператору, який містить цей вираз. Результатом виразу є нове значення. Для спрощення тесту

передбачається, що оцінка виразу не має побічного ефекту на змінні, які складають цей вираз. Це стосується мови COBOL, але не мови C, де вираз на кшталт «x = i++; змінює не тільки змінну "x", а й змінну "i".

3.2.3 Контрольна точка

Контрольна точка позначає місце в коді програми. Контрольні точки розташовані між кожним оператором, а також на початку та в кінці програми.

Інструкцію можна розглядати як функцію, що змінює стан. Змінений стан — це стан контрольної точки, що передує інструкції. Інструкція створює новий стан на контрольній точці після цієї інструкції. Тому контрольна точка є стабільним місцем, де можна спостерігати властивості змінних. Інструкцію можна розглядати як функцію, яка отримує стан як вхід для створення нового стану.

3.2.4 Довідкова інформація

У системі керування базою даних кортеж займає рядок, який перетинає стовпці, кожен із яких представляє атрибут. У цьому дослідженні термін кортеж використовується для представлення набору атрибутів, що описують об’єкт. Контекст — це кортеж, що складається з контрольної точки та стану. Контекст не розкриває нічого про минулі обчислення або про те, як потік управління прибув до контрольної точки. З іншого боку, контекст визначає саме майбутнє лікування, тобто решту виконання. Якщо виконання програми починається з цієї контрольної точки і зі станом пам’яті, те, що відбуватиметься в програмі з цієї контрольної точки, повністю визначається.

Мета аналізу — отримати всі контексти програми, тобто визначити для кожної контрольної точки набір значень, які можуть приймати змінні програми.

3.2.5 Маніпулювання абстрактними значеннями

Під час статичного аналізу не завжди можна точно визначити значення змінної, і тоді необхідно використовувати апроксимацію цього значення.

Наприклад, у програмі на малюнку 3.3 неможливо визначити точне значення змінної «I» після виконання інструкції «IF», оскільки неможливо дізнатися значення отриманої змінної «B» під час читання файлу.

```
READ FILE-A INTO B.
IF B > 10

    MOVE 1 TO I

ELSE

    MOVE 3 TO I
```

Рисунок 3.3 Приклад невизначеного значення

Хоча неможливо точно визначити значення змінної «I» після оператора IF, можна встановити, що це значення дорівнює 1 або 3. Також можна вважати, що значення знаходиться між 1 і 3 включно. Таким чином, значення змінної «I» може бути представлено інтервалом значення. Після виконання оператора IF цей інтервал дорівнює [1, 3]. Цей інтервал вводить значення 2, яке не є одним із можливих станів змінної «I» у цій контрольній точці. Однак ця неточність не важлива для оцінки доступу до елементів масиву. Дійсно, якщо доступ до елемента «I» таблиці «A» дійсний для граничних значень інтервалу, які є значеннями 1 і 3; тоді доступ дійсний для значення 2. Таким чином, значення 2 можна ігнорувати.

Беручи до уваги згадані абстракції, інструкції мови оригіналу можуть бути перетворені в дію на стан обраної абстракції. Таким чином, мета перетворення програми полягає в тому, щоб змінити інструкції мови оригіналу в інструкції щодо абстрактних значень. Наприклад, інструкція COBOL «MOVE 1 TO I», яка має ефект ініціалізації змінної «I» до значення 1, можна перевести шляхом ініціалізації інтервалу, що представляє змінну «I» до значення [1, 1]. Аналогічно, арифметичний вираз, який збільшує змінну на 1, матиме ефект зміни діапазону значень відповідної змінної.

Вибір для представлення числових змінних складається з інтервалу значень. Конкретне значення змінної можна змінити за допомогою інструкцій

програми. Діапазон значень, який використовується для представлення стану змінної, потрібно змінити, щоб відобразити оцінений вираз. Наприклад, враховуючи стан (I, [2, 7]), твердження «COMPUTE I = I * 2 + 1» має вивести стан (I, [5, 15]) у контексті контрольної точки за інструкцією .

Неініціалізована числова змінна представлена значенням «⊥». Змінна, значення якої отримано із зовнішнього для програми джерела, ініціалізується відповідно до максимального інтервалу, який може приймати ця змінна. Наприклад, якщо змінна "I" визначається як "PICTURE S99", тому найбільший діапазон, який може приймати ця змінна, становить [-99, 99]. Якщо змінна визначена без знака, наприклад, «PICTURE 99», то максимальний діапазон дорівнює [0, 99].

Проміжні інструкції, що використовуються у внутрішньому представленні програми, призначені для відображення впливу вихідних інструкцій програми на абстрактні значення змінних. Тому проміжна мова має інструкції, що дозволяють змінювати інтервали значень. Існують також інструкції для підтвердження того, що в даній точці програми змінна ініціалізована і що її значення є частиною заданого інтервалу. Для зручності обробки всі внутрішні інструкції мають однаковий розмір. Ці інструкції складаються з трьох полів: коду операції, який визначає операцію, яку потрібно виконати, першого операнда, який вказує на змінну, яка є об’єктом операції, і другого операнда, значення якого залежить від операції. Інструкції з проміжної мови наведені в таблиці 3.1.

Інструкція INTERV використовується для виявлення ситуацій, на які спрямований статичний аналіз. Коли зустрічається ця інструкція, якщо контекст містить інтервал значень, які не входять до меж таблиці, зазначеної інструкцією, потім створюється діагноз інциденту.

Інструкція ISDEF використовується для визначення, коли ця інструкція зустрічається, чи ініціалізована відповідна змінна чи ні.

Інструкції TEST є результатом перетворення тестів оригінальної програми. Ці інструкції дозволяють орієнтувати потік керування відповідно до значення змінної. Інші інструкції мають на меті змінити абстрактні значення, що відповідають змінним, відповідно до семантики вихідних інструкцій програми. Слід зазначити, що інструкції, що змінюють потік управління, інструкції тестування та інструкції керування для циклів не представлені проміжними інструкціями. Ці оператори спрямовують потік керування під час виконання і є предметом наступного розділу.

Таблиця 3.1 Перелік проміжних інструкцій

Операція	1-й операнд	2-й операнд	Опис
INTERV	таблиця	змінна	Підтверджує, що діапазон значень змінної повністю міститься в клемах таблиці.
INIT	Числова змінна	Значення або змінна	Замінює інтервал новим інтервалом, створеним із значення або поточного інтервалу зазначена змінна.
*	Числова змінна	Значення або змінна	Змінює поточний інтервал змінної.
+	Числова змінна	Значення або змінна	Змінює поточний інтервал змінної.
-	Числова змінна	Значення або змінна	Змінює поточний інтервал змінної.
/	Числова змінна	Значення або змінна	Змінює поточний інтервал змінної.
ISDEF	Змінна	Нічого.	Підтверджує, що змінна має значення на цьому етапі.

Продовження таблиці 3.1

TEST= TEST> TEST<	Змінна	Значення або змінна	Дозволяє виконати базовий тест.

3.3 Потік керування

Інструкції маніпулювання даними є важливими в аналізі програми. Порядок, у якому виконуються ці інструкції, також має вирішальне значення для можливості спостерігати за поведінкою програми. Таким чином, перетворення вихідної програми повинно витягувати та зберігати порядок виконання інструкцій вихідної програми.

Одна з основних труднощів статичного аналізу полягає в аналізі повторюваних інструкцій. Моделювання ітераційних інструкцій включає аналіз змінних і областей пам'яті, які змінюються в цих циклах. Ці змінні називаються змінними повторюваності. Наприклад, у простій ітерації виду «ВИКОНАТИ ЗМІНЮВАТИ I ВІД 1 НА 1 ДО I > H», складність пов'язана з повторюваною змінною «I», значення якої змінюється, щоб послідовно приймати значення інтервалу [1 , H].

Цю зміну можна представити у вигляді рецидивів. У випадку попереднього прикладу рекурентне відношення задається у вигляді $I_{k+1} = I_{k+1}$, де I_{k+1}, $k \geq 0$ – значення змінної "I" в кінці ітерації k+1 . Перед введенням першої ітерації змінна «I» ініціалізується значенням 1, що дає можливість ініціалізувати повторність $I_0 = 1$. З рекурентного відношення можна визначити значення змінної «I» на будь-якій ітерації циклу.

Існує кілька методів визначення рекурентних відносин. Один із підходів заснований на генеруванні функцій. Породжуючі функції докладно висвітлені в книзі Вілфа [17].

Для цілей тесту не потрібно отримувати відношення повторюваності циклів. Досить визначити найменше і найбільше значення, які можуть приймати повторювані змінні. Малюнок 3.4 ілюструє типову форму оператора ітерації в програмі COBOL.

```
PERFORM VARYING I
        FROM    1
        UNTIL   I > MAX
        OR      . . .
. . .
END-PERFORM.
```

Рисунок 3.4 Формат циклу в COBOL

Коли цей формат інструкції PERFORM виконується, можна визначити діапазон значень, які змінна «I» може приймати в цьому циклі. Цей інтервал дорівнює [1, MAX]. Якщо частина твердження «VARYING … FROM … BY …» пропущена, неможливо визначити значення інтервалу до початку статичного аналізу. Аналогічно, якщо тестова частина оператора PERFORM не містить предикатів для змінної "I", неможливо визначити повний діапазон значень, які змінна "I" може прийняти перед виконанням статичного аналізу. У цьому випадку статичний аналіз повинен припустити, що змінна «I» не має верхніх меж, крім тих, які накладаються внутрішнім представленням змінної.

Тому під час перетворення вихідної програми необхідно витягти інформацію про ітераційні інструкції. Необхідна інформація буває двох типів: змінні, змінені під час різних ітерацій, і максимальне значення, яке можуть приймати ці змінні.

На малюнку 3.4 змінна «I» є змінною, зміненою під час ітерацій, можливо, що в тілі циклу є інші змінні. Максимальне значення, яке може прийняти змінна "I", - це "MAX", тому що коли змінна має значення більше ніж «MAX», цикл закінчується. Якщо в предикаті ітераційного оператора є інші умови, можливо, є й інші максимальні значення для вилучення з циклу.

Якщо пропозиції, що утворюють предикат циклу, не пов’язані логічними операторами «або», важко визначити максимальні значення для змінних. У

цьому випадку легше ігнорувати ці предикати і вважати, що змінні не мають максимального значення. Ця ситуація може призвести до того, що статичний аналіз виявить аномалію в цей момент і дасть хибнопозитивний результат. Бажано, щоб обмежити кількість помилкових спрацьовувань, надати змінним конкретне значення в цій ситуації. Це значення має відрізнятися від значень, які може приймати змінна. Це дає змогу відрізнити це значення від значення, отриманого під час обробки. Вихід діагностики може ігнорувати ці проблеми, коли зустрічається змінна з цим конкретним значенням. Більшість інструментів сканування дозволяють користувачам зробити вибір, щоб обмежити кількість хибнопозитивних або помилково негативних результатів. Томас Фарінгер і Бернхард Шольц [8] пропонують метод символічного аналізу, що дозволяє обмежити кількість помилкових спрацьовувань у таких ситуаціях. Однак цей метод є складнішим, ніж той, що представлено в цій роботі.

3.3.1 Базовий блок

Коли кожна інструкція вихідної програми була переведена на проміжні інструкції, проміжні інструкції розбиваються на елементарні блоки. Новий блок створюється з першою інструкцією в програмі, а наступні інструкції додаються до нього, поки не зустрінеться інструкція розгалуження або мітка. За відсутності переходу та мітки елемент керування послідовно переходить від однієї інструкції до наступної, і ця інструкція потім стає частиною основного блоку. Таким чином, базовий блок — це блок, що складається з послідовних інструкцій, де керування може надходити лише через головну інструкцію, яка є першою інструкцією блоку.

Ахо, Лам, Сеті та Ульман [1] представляють алгоритм, який полягає у визначенні головних інструкцій, а потім побудові елементарних блоків. Правила визначення головних інструкцій наступні: перша інструкція в програмі є головною інструкцією, інструкція, яка є метою гілки, є головною інструкцією, і, нарешті, будь-яка інструкція, що йде безпосередньо за гілкою, є головною

інструкцією. Головні інструкції – це інструкції, які, ймовірно, отримають керування з місця, відмінного від попередньої інструкції.

Після того, як головні інструкції визначені, елементарні блоки складаються з головної інструкції та всіх інструкцій, які слідують за нею до наступної головної інструкції або до кінця програми.

3.3.2 Графік потоку керування

Проміжні інструкції, показані в Таблиці 3.1, не містять жодних інструкцій для керування потоком керування. Замість цього потік управління представляється за допомогою графіка. У графі потоку керування вершинами є проміжні інструкції програми у вигляді базових блоків. Між двома блоками виникає дуга, якщо можливо, що другий блок безпосередньо слідує за першим у можливому виконанні програми.

Графік, який використовується для представлення потоку керування, описаний у вигляді блок-схеми та адаптований зі статті Кузо [5]. Перевага такого представлення полягає в обмеженні кількості різних елементів, що спрощує статичний аналіз, як показано в розділі 4. Вибране подання використовує символи, представлені на малюнку 3.5. Це графічне зображення використовується в ілюстративних цілях у цьому есе. У внутрішньому представленні, що використовується інструментом аналізу, використовується структура даних, показана на малюнку 4.4. Однак між графічним і внутрішнім представленням існує відповідність один до одного.

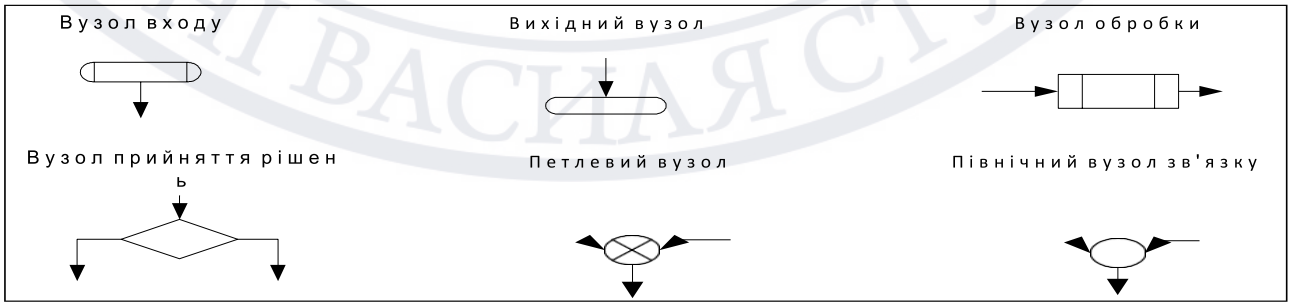

Рисунок 3.5 Основні елементи потоку управління

Графік потоку управління складається з таких елементів:

• вузол входу використовується для відображення єдиної точки входу програми, цей вузол без дуги входу відображає той факт, що він є початком програми;

• для кожного базового блоку використовується вузол обробки або прийняття рішення, останній складається з інструкцій, що генеруються під час трансляції програми;

• коли кілька дуг з'єднуються в заданій команді, вузол з'єднання вставляється перед інструкцією для отримання цих дуг туди, вузол з'єднання отримує тільки дві вхідні дуги;

• вузол виходу використовується в кінці програми, цей вузол є єдиним вузлом, який не має дуги виходу.

Повинен бути принаймні один шлях від вузла одного входу до кожного з інших вузлів на графіку. Це означає, що будь-який цикл на графі містить принаймні один вузол з’єднання.

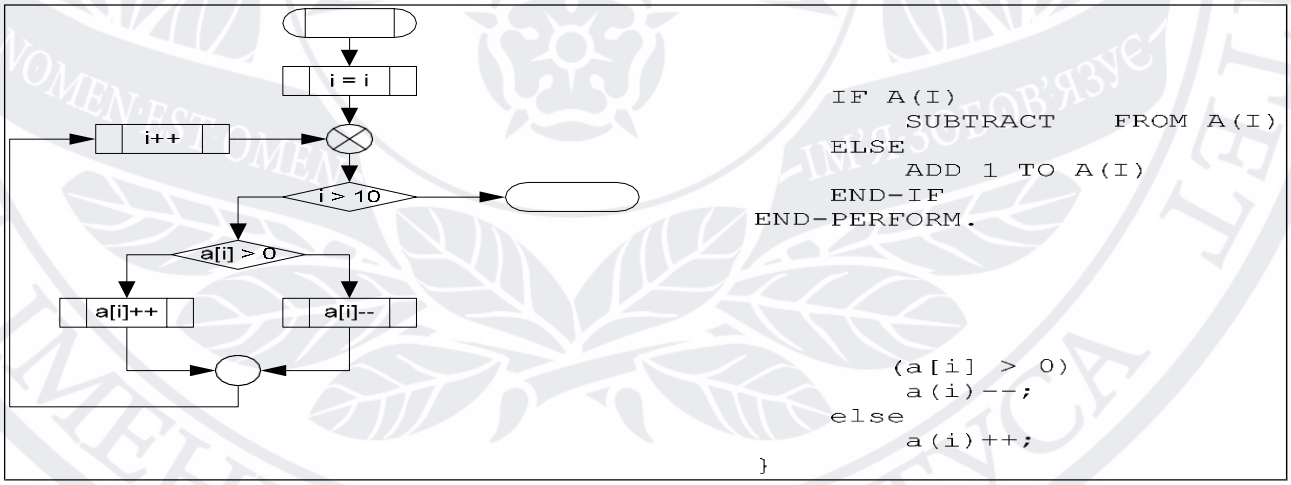

Рисунок 3.6 Приклад блок-схеми управління

На малюнку 3.6 представлено приклад потоку керування для частини програми. Відповідна програма COBOL і C знаходиться праворуч від малюнка.

Для будь-якого конкретного застосування абстрактного алгоритму оцінки необхідно визначити оцінку основних блоків: вузла обробки, вузла прийняття рішення, а також вузлів, що приєднуються. Теорія представлена в додатку 5.

3.4 Теги

Оскільки метод абстрактного аналізу використовує абстракцію даних програми, само собою зрозуміло, що внутрішнє представлення змінних залежить від обраної абстракції.

Цілі змінні представлені у вигляді інтервалів значень. Для цих змінних достатньо знати, чи змінна не визначена, або знати діапазон значень, які ця змінна може представляти в даному контексті. Для масивів потрібно знати кількість записів із визначення масиву. Для інших змінних достатньо знати, чи є змінна невизначеною (⊥), ініціалізованою (d) або неможливо визначити, ініціалізована змінна чи ні (T). Остання ситуація можлива в ситуації, показаній на малюнку 3.7.

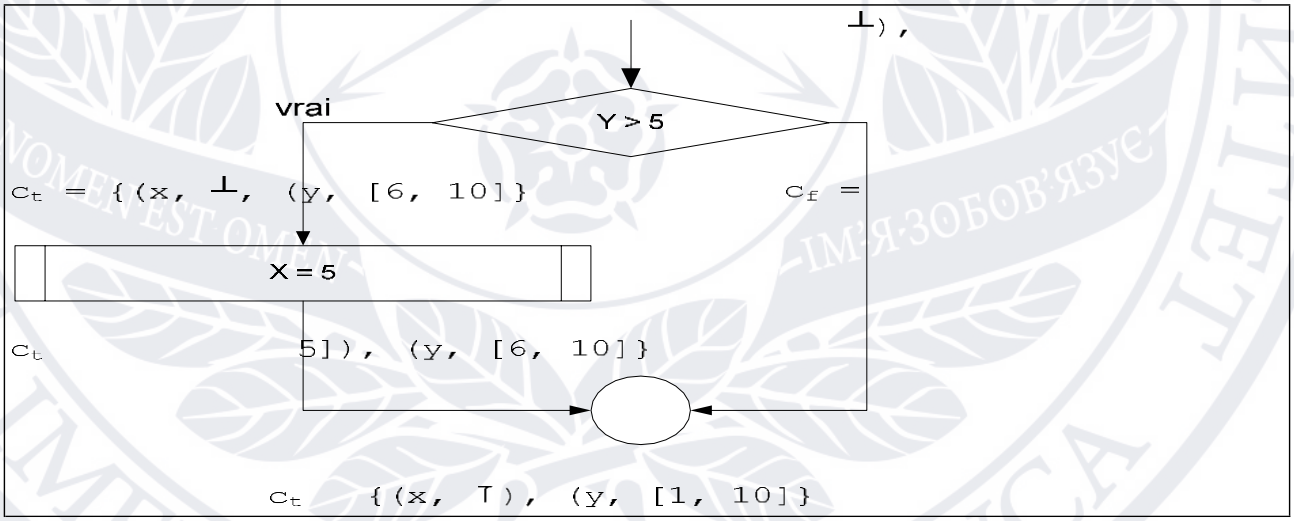

Рисунок 3.7. Рішення, що призводить до невизначеного значення

РОЗДІЛ 4. АНАЛІЗ ПРОГРАМИ

У цьому розділі представлено алгоритм аналізу, вибраний для розв'язання завдань, на які спрямований тест, і для демонстрації використання алгоритму на прикладі.

Алгоритм, представлений у цій главі, походить від Ахо, Лама, Сеті та Ульмана [1]. Цей алгоритм називається «аналіз регіону».

При аналізі за регіонами програма розглядається як ієрархія регіонів. Регіон — це частина графіка потоку даних, яка має лише одну точку входу. Ця концепція, яка дає змогу розглядати код як ієрархію, має бути інтуїтивно зрозумілою, оскільки процедура, структурована в блоки, природно організована в ієрархії регіонів.

Кожна інструкція в структурованій програмі є областю, оскільки потік керування починається на початку інструкції. Кожен рівень вкладення операторів відповідає рівню в ієрархії регіонів. Більш формально, область керуючого потокового графа являє собою сукупність вузлів N і дуг A таких, що:

• У N є голова t, яка домінує над усіма вузлами N.

• Якщо вузол m може досягти вузла n в N, не проходячи через t, то m також знаходиться в N.

• A складає множину всіх дуг у потоці керування між вузлами n_1 і n_2 в N, за винятком, можливо, деяких, які входять до t.

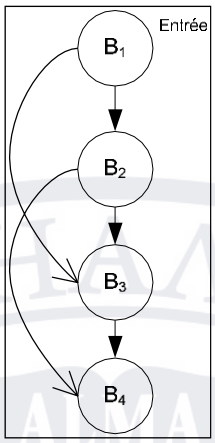

Рисунок 4.1 Приклад регіонів

Природний цикл – це область, але область не обов'язково має дугу повернення і не може мати циклів. На малюнку 4.1 вузли B_1 і B_2, включаючи дугу $B_1 \rightarrow B_2$, утворюють область, а також вузли B_1, B_2 і B_3 з дугами $B_1 \rightarrow B_2$, $B_2 \rightarrow B_3$ і $B_1 \rightarrow B_3$. Однак граф, утворений вузлами B_2 і B_3 з дугою $B_2 \rightarrow B_3$, не утворює область, оскільки управління може ввійти в цей граф через вузли B2 і B3. Іншими словами, жоден з вузлів B_2 або B_3 не домінує над іншим, тому умова (1) правил регіону не виконується. Навіть якщо ми вибрали вузол B_2, наприклад, як голову, умова (2) не виконується, оскільки до вузла B_3 можна дістатися з B_1, не проходячи через B_2, а B_1 не знаходиться в «області».

Для потреб алгоритму аналізу за регіонами розглядаються три типи областей: елементарні області, області тіла та області циклу.

4.1 Ієрархії регіонів

Щоб побудувати ієрархію регіонів, потрібно знайти природні ітерації. У мовах, які не використовують прямі інструкції розгалуження, такі як "go to", два цикли в програмі або не перетинаються, або включені один в одного. Процес розкладання графа потоку керування на його ієрархію ітераційних областей починається з розгляду кожного основного блоку як області окремо, це елементарні регіони. Після цього необхідно впорядкувати природні цикли

зсередини назовні, тобто починаючи з найбільш вкладених ітерацій. Обробка циклу складається із заміни його вузлом у два кроки:

• Спочатку тіло циклу B (яке включає всі вузли та дуги, крім дуги повернення до голови) замінюється вузлом, що представляє область R. Дуги до початку циклу B тепер входять у вузол області R. дуга з будь-якого виходу циклу B замінюється дугою, яка йде від області R до того самого місця призначення. Однак, якщо розглянута дуга є дугою повернення до головки петлі, то вона стає петлею на R. Тоді область R називається областю тіла.

• Нарешті, ми повинні побудувати область R', яка представляє повний природний цикл B. R' називається петлею. Єдина відмінність між областю R і областю R' полягає в тому, що остання включає дугу повернення до головки петлі B.

Ці два кроки повторюються, зводячи все більш широкі петлі на одиничні вузли, спочатку зі зворотним бантом, а потім без цього банта. Згодом усі природні цикли зводяться до одного вузла або може бути кілька вузлів без циклів, утворюючи ациклічний граф з більш ніж одним вузлом. У першому випадку процес завершено, але в другому випадку для повного потокового гра фа $f_{R,SORTIE[B]}$ необхідно побудувати інший вузол області. Алгоритм побудови регіону детально наведено в додатку 6.

Приклад застосування алгоритму представлено в Додатку 8.

4.2 Огляд аналізу за регіонами

Для кожного регіону R і кожної субрегіону R' R ми повинні створити передатну функцію $f_{R,ENTRÉE[R']}$, який підсумовує наслідки виконання всіх можливих шляхів, які йдуть від запису R до запису R', залишаючись у R.

Блок B в R є вихідним блоком області R, якщо він має ребро з блоку B на блок поза R. Нам також потрібно створити передатну функцію для кожного

вихідного блоку B з R, яка називається, який підсумовує наслідки виконання всіх можливих шляхів у R, що ведуть від входу області R до виходу блоку B.

Алгоритм робить це, піднімаючись вгору по ієрархії регіонів і створюючи функції передачі для все більших і більших регіонів. Ви повинні почати з основних блоків B, де $f_{B,ENTRÉE[B]}$ — функція ідентифікації, а $f_{B,SORTIE[B]}$ — функція передачі для самого блоку B. Коли обробка просувається вгору по ієрархії,

• Якщо R є областю тіла, дуги, що належать до R, утворюють ациклічний графік областей, які входять до R. Необхідно приступити до створення передатних функцій у топологічному порядку включених областей.

• Якщо R є петлею, ми просто повинні розглянути вплив дуги повернення до головки R.

Нарешті, досягається вершина ієрархії і створюється передатна функція для області R_n, тобто функція для всього графіка.

Наступним кроком є обчислення контекстів на вході та виході з кожного блоку. Регіони обробляються в зворотному порядку, починаючи з області R_n і рухаючись вниз по ієрархії. Для кожного регіону необхідно обчислити вхідні контексти. Для області R_n застосуйте $f_{Rn,ENTRÉE[R]}$(ENTRÉE[CONTEXTE]), щоб отримати значення потоку даних на вході регіонів, включених в R_n. Цей процес слід повторювати до тих пір, поки не будуть оброблені основні блоки ієрархії регіонів.

РОЗДІЛ 5 ДІАГНОСТИКА ПОМИЛОК

У попередньому розділі було продемонстровано, а потім проілюстровано на прикладі, використання алгоритму аналізу регіонів для виявлення двох простих помилок програмування: використання неініціалізованої змінної та доступу до таблиці за межами цієї.

При виявленні потенційної помилки необхідно провести діагностику, щоб програміст міг визначити ситуації, коли ці помилки можуть виникнути, і виправити ці помилки.

Інформація, необхідна для діагностики помилки, була виключена з розгляду прикладу 4.1, щоб не ускладнювати його. У цій главі описується інформація, яку необхідно отримати з різних етапів аналізу програми, щоб мати можливість діагностувати виявлені помилки.

У першому розділі вказується інформація, яка повинна бути включена в діагностику, щоб вона була корисною. У наступному розділі показано, як умови, що виникли під час послідовності виконання, можуть допомогти поставити точнішу діагностику можливих помилок. Згодом решта глави присвячена вказівці того, як етап аналізу програми дозволяє отримати інформацію, необхідну для постановки діагнозу.

5.1 Діагностика

Щоб діагностика була корисною, недостатньо визначити інструкцію, де може виникнути помилка. Цілком можливо, що лише деякі послідовності виконання стикаються з помилкою в цій інструкції. Ця ситуація проілюстрована на прикладі.

У більшості прикладів у цій главі код представлений COBOL, а не проміжною мовою для полегшення розуміння. Праворуч від кожного твердження представлений результуючий контекст твердження для змінних, що цікавлять.

На малюнку 5.1 змінна «I» ініціалізується під час виконання команди 5 лише тоді, коли була виконана інструкція 3. Якщо змінна «X» має значення нуль, то оператор 3 не виконується, а змінна «I» не має значення, коли виконується оператор 5. Коли зустрічається умовний оператор, вхідний контекст інструкції перетворюється на два контексти, один для «істинної» гілки [27] умови, а іншу для «хибної» гілки. Кінцеві контексти гілки "true" і "false" потім об'єднуються після тестового оператора. Під час цього злиття, як показано на малюнку 5.1, змінна може бути визначена в одному з двох контекстів, але не в іншому. У цьому випадку значення «T» використовується для вказівки на те, що змінна може бути встановлена або ні в цей момент.

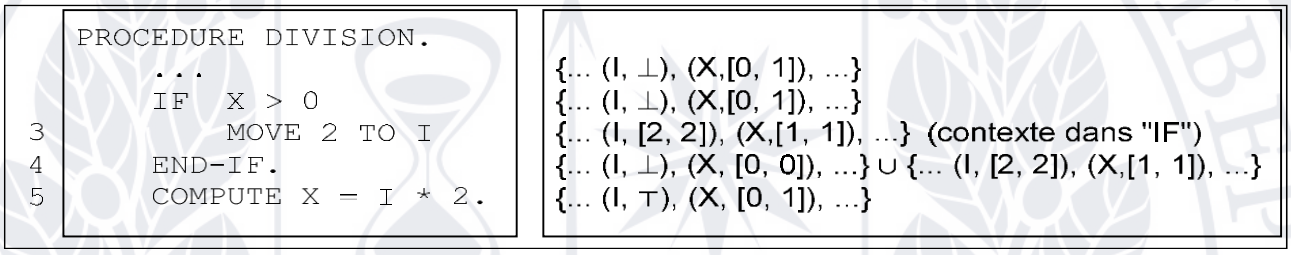

Рисунок 5.1 Програма, де помилка залежить від шляху виконання.

На малюнку 5.1 помилка виявлена в команді 5, оскільки контекст на вході цієї інструкції має значення «T» для змінної «I», яка використовується цією інструкцією. На цьому етапі можна провести таку діагностику: "змінна I може бути невизначена в інструкції 5". З таким діагнозом у програміста немає підказок, які б могли допомогти йому визначити, коли змінна може бути невизначеною.

Було б корисніше мати трасування виконання, яке призвело до помилки: "змінна I не визначена в операторі 5. Трасування: {1, 2, 5}". У цій діагностиці трасування виконання — це послідовність інструкцій, що виконуються до тих пір, поки помилка не буде виявлена. Дотримуючись цієї послідовності операторів, легше визначити, що програма не виконує оператор 3, який вказує, що умовний оператор 2 дає умову, де встановлена змінна «I», отже, оператор також вказує умови, за яких змінна не визначена.

Цю інформацію можна додати до діагностики: "змінна I не визначена в інструкції 5. Трасування: {1, 2 (X = 0), 5}". У цій діагностиці трасування виконання поєднується з контекстною інформацією для змінних, що використовуються в умовному операторі, для отримання більш точної діагностики. Контекст перед оператором вказує, що можливі значення для змінної «X» знаходяться в діапазоні [0, 1] і умова виконується для значень, більших за нуль. Тому єдине значення, для якого не виконується інструкція, яка ініціалізує змінну "I", - це нульове значення.

5.2 Контекстна інформація в діагностиці

Приклад у попередньому розділі продемонстрував, що контекстну інформацію можна використовувати для постановки діагностики, який допомагає визначити обставини, за яких виникає помилка. Цей розділ ілюструє застосування цієї техніки, коли умовний оператор складається з предикатів, пов’язаних разом логічними операторами «і» та «або», щоб утворити більш складну умову.

На малюнку 5.2 показано, як складна умова розбивається на серію проміжних операцій TEST. Кожна з цих інструкцій виконує перевірку однієї змінної. Граф потоку керування зв’язує ці операції, щоб зберегти семантику вихідної складної умови.

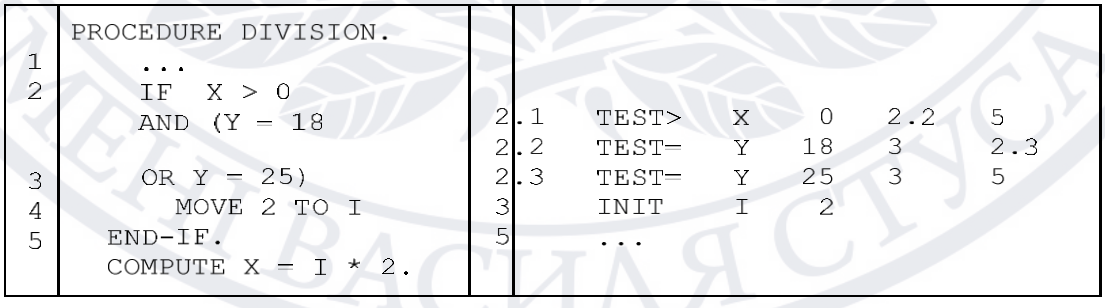

Рисунок 5.2 Приклад комплексного тесту.

Графік потоку управління показаний на рисунку 5.3.

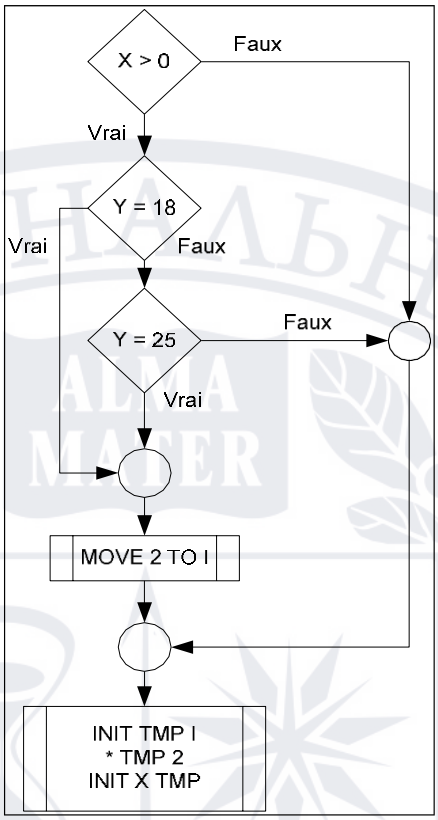

Рисунок 5.3 Потік контролю для комплексного тесту.

На малюнку 5.4 показані контексти, пов’язані з проміжним кодом умовного оператора. Для умовних операторів створюються два контексти. Перший контекст, ліворуч, — це контекст, що передається до першої інструкції «істинної» гілки оператора IF, другий контекст передається до першої інструкції «неправильної» гілки оператора IF. Коли зустрічається вузол об’єднання, два контексти, які об’єднуються на цьому вузлі об’єднання, об’єднуються, щоб створити контекст, який потім передається до наступної інструкції.

```
2.1   TEST>   X    0   2.2   5
2.2   TEST=   Y   18   3     2.3
2.3   TEST=   Y   25   3     5
(jonction vrai 2.2 et vrai 2.3)
3     INIT    I    2
5     ...
```

```
{(I,⊥), (X,[0,1]), (Y,[15,18])}
{(I,⊥), (X,[1,1]), (Y,[15,18])}  {(I,⊥), (X,[0,0]), (Y,[15,18])}
{(I,⊥), (X,[1,1]), (Y,[18,18])}  {(I,⊥), (X,[1,1]), (Y,[15,17])}
∅                                {(I,⊥), (X,[1,1]), (Y,[15,17])}
{(I,⊥), (X,[1,1]), (Y,[18,18])}
```

Рисунок 5.4. Контексти в результаті складного тесту.

На малюнку 5.4 слід зазначити, що тестова інструкція 2.3 не створює контекст для гілки "true", оскільки в якості входу для цього тесту змінна "Y" не

може мати значення 25. Тому неможливо, щоб "true" " виконується гілка тесту 2.3. При виконанні оператора 3 змінна «I» не визначена, і в цьому випадку точно отримується вміст змінних «X» і «Y», які викликають помилку.

5.3 Трасування виконання

У цьому розділі показано, як визначити трасування виконання, яке відтворює цю помилку.

Метод аналізу за регіонами не розглядає програмну інструкцію за інструкцією. Таким чином, трасування виконання має створюватися, починаючи з точки, де виявлена помилка, і працювати назад через інструкції до тих пір, поки не буде досягнута інструкція, яка дала змінній погане значення, або поки не буде виявлено початок програми.

Запропонований у тесті метод полягає в анотуванні контекстів додатковою інформацією, що дозволяє визначити інструкції, які надали значення змінним. Побудова графа керуючого потоку полегшує цю обробку, накладаючи певні правила під час розробки графа. Побудова графіка потоку керування описана в розділі 3.3.2. Важливим правилом є обмеження кількості керуючих дуг, які можуть входити або виходити з вузла. У вузол з’єднання можуть входити лише дві керуючі дуги. Для всіх інших типів вузлів може входити лише одна дуга. Крім того, єдиний вузол, де може вийти більше однієї дуги, - це вузол рішення.

Кожна інструкція перетворює свій вхідний контекст, щоб створити вихідний контекст. Контекст складається із стану. Стан — це список змінних з їх значенням у цій контрольній точці. Для кожної змінної [30] необхідно додати ще один фрагмент інформації, який складається з пари, що вказує одну або дві інструкції, які можуть бути джерелом значення змінної. Це показано за допомогою програми на малюнку 5.5.

```
    PROCEDURE
    DIVISION.
1   READ FICH1 INTO X.
2       IF   X > 0
3           MOVE 2 TO I
        ELSE
4           MOVE 1 TO I
5           MOVE X TO J
6       END-IF.
7   COMPUTE I = I * J.
```

Рисунок 5.5 Приклад контексту з розширеним станом.

Перша інструкція ініціалізує змінну «X» із прочитаного запису. Тому змінна "X" може мати будь-яке значення, дозволене її визначенням. Таким чином, стан змінної «X» дорівнює (X, [0,999], (1,0)). Перша частина цього кортежу - це ім'я змінної, друга частина - діапазон значень, які це значення може приймати в стані, а третя частина складається з двох номерів інструкцій, які, можливо, сприяли створенню цього значення. Після твердження 1 значення змінної «X» походить із твердження 1.

Другий номер команди завжди дорівнює нулю, за винятком вихідного контексту вузла приєднання. У цьому випадку два номери оператора вказують на останню частину «істинної» частини та «хибну» частину умовного оператора, який змінив змінну.

Оператор 2 не змінює жодних змінних, але створює два вихідні контексти для гілок «true» та «false» умовного оператора. Контекст філії

«true» — {(X, [1, 999], (1,0))}, а контекст для гілки «false» — {(X, [0, 0], (1,0))}.

Твердження 3 дає значення змінній “I”: (I, [2, 2], (3,0)). Твердження 4 також дає значення змінній “I”: (I, [1, 1], (4,0)).

Інструкція 5 ініціалізує змінну «J», копіюючи в неї значення змінної «X». Стан змінної «J» стає (J, [0, 0], (5, 0)), оскільки змінна «X» має нульове значення у випадку, коли умова інструкції 2 є хибною, як вказує контекст, який походить із оператора 2 і входить у "неправдиву" частину тесту.

Інструкція 6 представляє з'єднання двох гілок, що виходять з тесту. Два стани змінної "X" об'єднуються, щоб дати: (X, [0, 999], (1, 0)), а два стани змінної "I" дають (I, [1, 2], (3, 4)). Стан змінної "I" вказує на те, що значення змінної "I" у цій точці може бути 1 або 2 і що це значення походить з інструкції 3 або 4. Стан змінної "J" вказує, що змінна може бути невизначеною в цей момент або може мати значення: (J, T, (6, 0)). Як зазначено в цьому контексті, не інструкція 5 змінила значення змінної «J», а скоріше злиття двох контекстів, що надходять із тесту, отже, інструкція 6.

Інструкція 7 намагається отримати доступ до змінної «J» і виявляє аномалію, оскільки змінна

"J" може бути невизначеним у цей момент. Для побудови трасування виконання необхідно використовувати номери інструкцій, які були додані до звітів. Контекст введення інструкції 7 для змінної “J” такий: (J, T, (6, 0)). Цей контекст повідомляє нам, що останнім оператором, який змінився на змінну "J", є оператор 6. Оператор 6 є об'єднанням двох контекстів для значення змінної "J". Для змінної «J» один із цих контекстів має значення, яке не відображає помилку, це інструкція 5 і, отже, не є джерелом помилки. Інший контекст вказує, що змінна "J" не визначена, і це контекст, який потрібно включити в трасування виконання. Таким чином, можна було повернутися від інструкції 7, до інструкції 6, а потім до інструкції 3, щоб побудувати часткове трасування виконання. Ці сліди можна вивести з модифікацій змінної «J».

5.4 Побудова траси виконання

Щоб отримати повне відстеження виконання, необхідно знайти всі виконані інструкції, навіть якщо вони не змінюють змінну. Контекст повинен

бути анотований номерами рядків одного або обох операторів, які досягли оператора.

Отже, контекст тепер формує список змінних з їх значенням, а контекст також має пару, що вказують на одну або дві інструкції, що надають контроль над інструкцією, з якої походить контекст. Як і у випадку зі змінними, кортеж, доданий до контексту, матиме номер рядка, а номер другого рядка [28] дорівнює нулю, за винятком випадків, коли контекст отримано з вузла об'єднання.

У випадку на малюнку 5.5 контекст твердження 1 виглядає так: {(X, [0, 999], (1, 0))} / (1,0). Прийнято позначення полягає в тому, щоб додати кортеж, що містить інструкції, які ведуть до контексту, передуючи цьому кортежу символ «/», щоб відокремити кортеж від списку станів змінних.

Контекст твердження 6 такий: {(X, [0, 999], (1, 0)), (I, [1, 2], (3, 4)), (J, T, (6, 0))} / (3, 5). Цей контекст вказує, що елемент керування може виходити з оператора 3 або оператора 5 залежно від значення умови оператора 2.

Таблиця 5.1 ілюструє контексти прикладу 5.1. Ці контексти містять оператор, де ця змінна, можливо, отримала своє значення. Крім того, контекст містить кількість одного або двох операторів, з яких виник контекст, який створив вихідний контекст.

Таблиця 5.1 Контекст прикладу 5.1

Інструкція	Контекст до інструкції

Продовження таблиці 5. 1

1	{(X, [0, 999], (1, 0))} / (0, 0)
2	{(X, [1, 999], (1, 0))} / (1, 0) et {(X, [0, 0], (1, 0))} / (1, 0)
3	{(X, [1, 999], (1, 0)), (I, [2, 2], (3, 0))} / (2, 0)
4	{(X, [0, 0], (1, 0)), (I, [1, 1], (4, 0))} / (2, 0)
5	{(X, [0, 0], (1, 0)), (I, [1, 1], (4, 0)), (J, [0, 0], (5, 0))} / (4, 0)
6	{(X, [0, 999], (6, 0)), (I, [1, 2], (3, 4)), (J, T, (5, 0))} / (3, 5)
7	{(X, [0, 999], (6, 0)), (I, T, (7, 0)), (J, T, (5, 0))} / (6, 0)

Контекст оператора 6 показує, що змінна «J» вже має значення, яке вказує на те, що ця змінна може бути невизначеною. На цьому етапі не вказується жодної помилки, тому що ця ситуація може бути нормальною: якщо на змінну більше не посилається в решті програми або якщо значення змінної знову змінено, перш ніж ця змінна не використовуватиметься.

Інструкції 7 потрібен вміст змінної «J», і в цей момент значення «T» вказує, що вона може бути невизначеною. Тому на цьому етапі виявляється помилка. Слід побудувати трасування виконання. Оскільки трасування будується на основі помилкової інструкції і йде вгору по потоку виконання, ця трасування створюється в зворотному порядку.

Інструкція 7 створює часткове трасування: {7}. Заява 7 вказує, що вхідний контекст оператора походить із оператора 6.

І Інструкція 6 додасть інформацію до трасування: {6, 7}. З іншого боку, оскільки інструкція 6 є вузлом з'єднання, а не виконуваною інструкцією в оригінальній програмі, трасування залишається незмінним: {7}. Твердження 6 вказує, що для створення контексту були використані два контексти, твердження 3 і 5. Контекст оператора 7 вказує, що значення оператора 7 може виходити з контексту 5. У контексті, створеному оператором 5, змінна «J» визначено. Тому до трасування виконання потрібно додавати не контекст. Контекст оператора 3

не містить змінної «J», що означає, що змінна «J» не має значення на даний момент. Потім інструкція 3 додається до трасування виконання: {3, 7}. Твердження 3 вказує на те, що контекст, використаний для цього твердження, походить із твердження 2.

Твердження 2 є умовним твердженням. Умова складається з перевірки значення змінної «X». Оскільки твердження 3 є частиною «істинної» гілки тесту, а умовою є «X > 0», аналіз визначає значення змінної «X», які передаються в «істинному» контексті, і ці копіюються в трасі виконання:

{2 (X = [1, 999]), 3, 7}. Контекст твердження 2 походить із твердження 1.

Інструкція 1 додається до трасування виконання: {1, 2 (X = [1, 999]), 3, 7}. Оскільки контекст команди 1 не дає номер команди, початок програми досягнуто. Таким чином, слід виконання: {1, 2 (X = [1, 999]), 3, 7}.

Цей розділ дає уявлення про інформацію, необхідну для визначення траси виконання, яка може керувати ідентифікацією аномалій, наявних у програмі. Можна надати додаткову інформацію, оскільки контекст кожної з інструкцій відомий.

5.5 Приклад діагностики

У попередньому розділі показано, як зібрати інформацію, необхідну для постановки діагнозу. Щоб програміст міг використовувати цю інформацію, діагностика повинна бути легкою для розуміння та інтерпретації. У цьому розділі розглядається трасування виконання, створене в попередньому розділі, і наводиться приклад діагностики продукту.

Представлення діагнозу, природно, залежить від середовища, що використовується для його подання. Якщо інструмент аналізу інтегрований в середовище розробки, цілком можливо, що діагностика відображатиметься у вікні й дозволяє за допомогою посилань здійснювати навігацію між діагностикою та вихідною програмою. У разі використання конкретного інструменту, незалежного від середовища розробки, діагностика може мати

форму простого звіту. В останньому випадку важливо дати можливість легко знайти інструкції оригінальної програми з діагностики.

Діагностика, представлена в цьому розділі, передбачає, що діагностика проводиться незалежно від середовища розробки. Він представляє інструкції з інформацією про трасування виконання та значення змінних. На малюнку 5.6 наведено приклад діагностики для програми на малюнку 5.5.

На малюнку 5.6 кожна інструкція в трасі виконання друкується зі своїм номером рядка. У рядках, що передують інструкції, змінні, які використовує інструкція, відображаються зі своїм значенням. У рядках після інструкції змінні, змінені інструкцією, відображаються з новим значенням. Зауважте, що діагностика включає не всі змінні програми. Приклад програми дуже простий, він містить сім інструкцій і три змінні. Справжня програма має набагато більше інструкцій і змінних. Щоб уникнути надто об’ємного діагностичного звіту, важливо включити відповідну інформацію, яка дозволить програмісту належним чином проаналізувати виявлену аномалію.

Між кожною інструкцією вставляється рядок-роздільник, щоб чітко розмежувати кроки трасування виконання. Трасування представлено в порядку виконання інструкцій, а не в порядку, в якому будується трасування, що дасть трасування, починаючи з інструкції з помилкою і повертаючись назад під час виконання програми. Цей останній метод робить консультацію з діагнозом менш інтуїтивною для користувача.

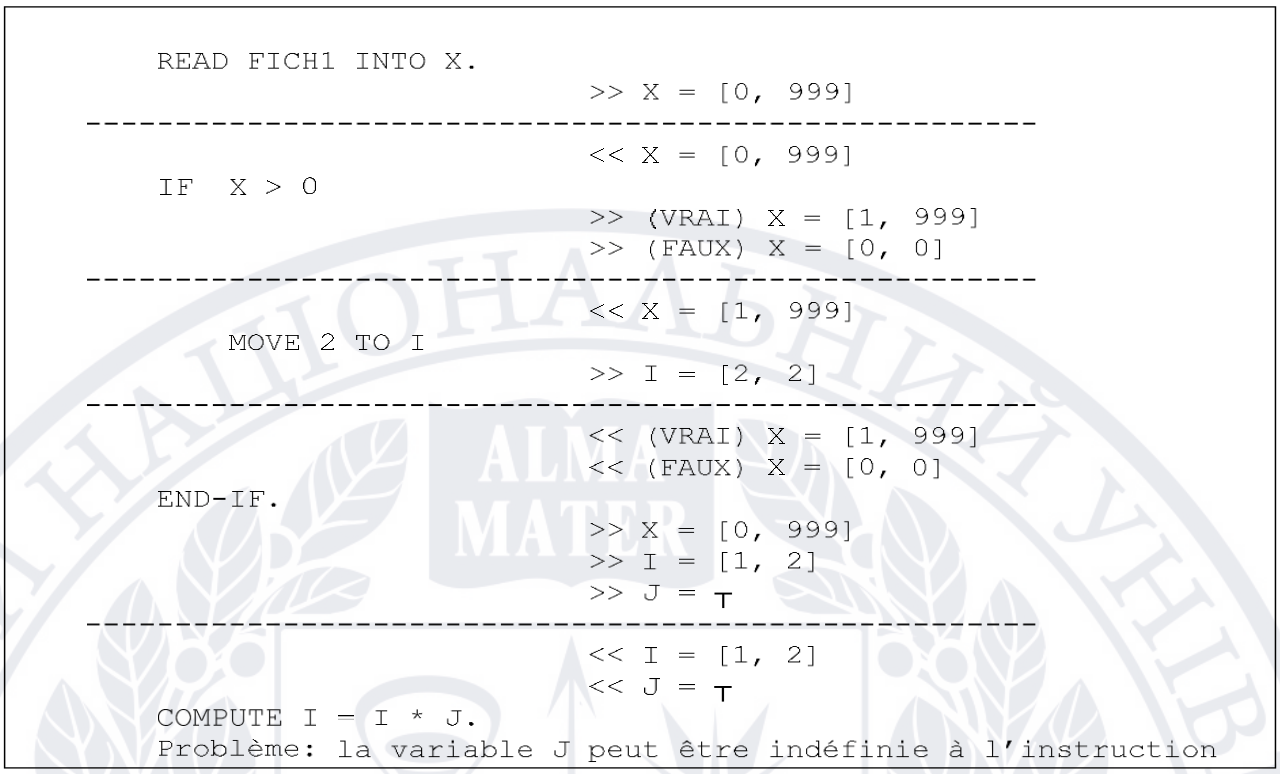

Рисунок 5.6 Приклад діагностики зі змінними значеннями.

Діагноз на малюнку 5.6 можна інтерпретувати наступним чином. За вказівкою 7 виявляється потенційна несправність і там проводиться діагностика помилки. Змінна, яка викликає аномалію, є частиною повідомлення про помилку. Це змінна «J». Значення змінної «J» як вхід до інструкції вказує на значення «T», що означає, що на даному етапі можливо, що змінна не визначена. Отже, принаймні один шлях виконання йде до цієї інструкції, тоді як змінна «J» не визначена.

Попередня інструкція в трасі виконання — це інструкція 6. Після виконання цієї інструкції змінна «J» фактично має значення «T». Однак значення змінної "J" більше не відображається в трасі виконання. На даний момент не зрозуміло, чому виникає помилка. Було б корисніше включати в діагностику на кожному кроці трасування виконання змінну з помилкою, навіть якщо вона не використовується інструкцією. Ця діагностика показана на рисунку 5.7.

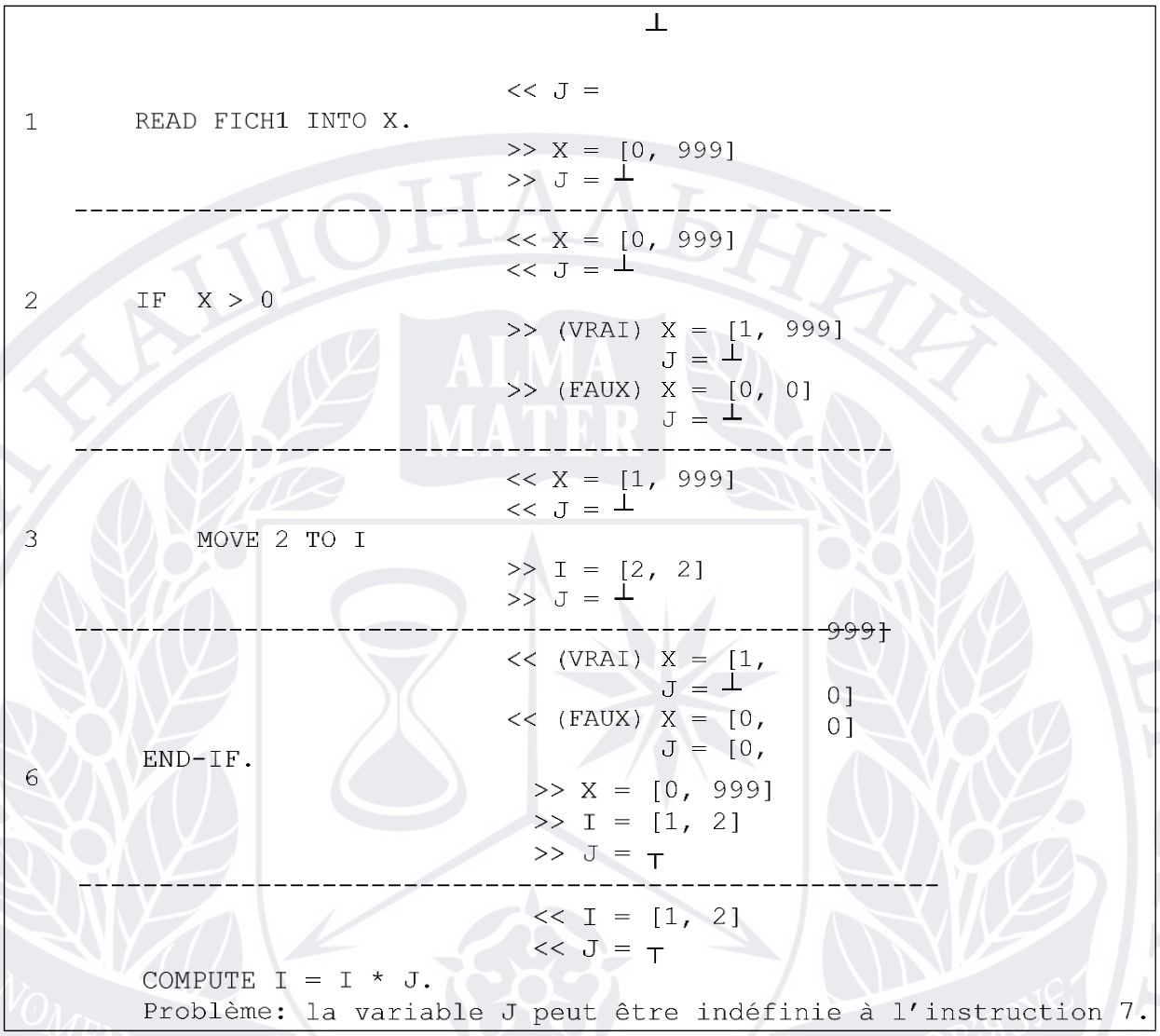

Рисунок 5.7 Приклад діагностики зі значеннями змінної помилки

Як показано на рисунку 5.7, значення змінної «J» з'являється до та після кожної інструкції в трасі виконання. Діагностика вказує на вході інструкції 6, що під час злиття контекстів «істинної» частини умовної інструкції з контекстом «хибної» частини, яку змінна «J» не визначає, не визначається в контексті «справжня» частина. Це вказує на те, що змінна «J» не ініціалізується, коли умова «X > 0» істинна.

Значення "T" означає, що два контексти були об'єднані, і що один з цих контекстів містить значення для змінної, а інший контекст вказує, що змінна не визначена. Таким чином, під час постановки діагнозу можна визначити стан, який призводить до наявності двох різних контекстів на цьому етапі.

На малюнку 5.7 у операторі 6 два контексти об’єднані. У контексті «false», змінна «J» визначена, а в іншому контексті її немає. «Істинна» гілка умовного оператора виконується, коли змінна «X» більше нуля. Таким чином, діагностика може вказувати на те, що змінна «J» не визначена в інструкції 7, оскільки змінна «X» має значення більше нуля в інструкції 2. Цей тип діагностики дає змогу надати користувачеві максимум інформації для полегшення. робота з виправлення несправностей.

Раніше зазначалося, що для великої програми діагностика може зайняти досить багато часу. Цікавою особливістю, яка полегшує використання діагностики, є надання програмісту контролю над рівнем інформації, що надається в діагностиці. Включення змінної помилки до всіх інструкцій у трасування виконання може допомогти вирішити проблему. Інструмент повинен дозволити програмісту вказати список змінних, які також повинні бути включені або виключені з журналів виконання, щоб дозволити отримати більше або менше інформації та полегшити локалізацію помилок.

5.6 Помилкові результати

Однією з труднощів, яка заважає прийняттю інструментів статичного аналізу, є кількість помилкових спрацьовувань, які виявляються цими інструментами. Існують випадки, коли необхідно знайти компроміс між складністю скануючого інструменту та точністю сканування. Простий інструмент сканування може позначати помилкові спрацьовування, які може усунути більш складний інструмент.

Якщо кількість помилкових спрацьовувань дуже висока порівняно з кількістю виявлених реальних аномалій, можливо, програміст втратить довіру до інструменту, якщо йому доведеться постійно перевіряти довгий список потенційних інцидентів, які не є реальними проблемами. Крім того, якщо розгляд списку потенційних проблем займає більше часу, ніж виконання модульних тестів, процес розробки стає громіздким. Це зменшує одну з переваг інструментів аналізу

З цих причин існують стратегії для обмеження кількості помилкових спрацьовувань. Одна з цих стратегій полягає в тому, щоб дозволити програмісту вказати рівень точності, який необхідно досягти під час аналізу. Наприклад, якщо предикат, який керує виконанням циклу, є занадто складним, щоб дозволити аналізу визначити точні значення повторюваних змінних, інструмент може сигналізувати про аномалію, не будучи впевненим, що може статися інцидент. Програміст повинен мати можливість вказати інструменту сканування ігнорувати такі проблеми чи ні. Це дає змогу обмежити помилкові спрацьовування, які обумовлені рівнем складності аналізу.

Інша стратегія полягає в тому, щоб дозволити програмісту вказати для кожної помилки, виявленої статичним аналізом, чи є вона хибнопозитивною чи ні. Потім інструмент сканування підтримує базу даних для кожної відсканованої програми, і коли програма буде просканована знову в майбутньому, ці помилкові результати більше не повідомляються.

ВИСНОВКИ

Завдання цієї наукової роботи — представити статичний аналіз як інструмент, який може допомогти команді розробників легше створювати якісний код. Точніше, випробування продемонстрували, як такий інструмент може автоматично виявляти поширені проблеми, що виникають у галузі.

Для того, щоб надати програмістам інструмент, який дозволить їм надавати код кращої якості, важливо визначити основні проблеми якості, які необхідно вирішити. У цьому тесті було обрано два типи аномалій, які часто зустрічаються на практиці, а саме використання змінних, які не ініціалізуються перед посиланням, і переповнення таблиць.

Коли зроблено вибір ситуацій, що підлягають виявленню, необхідно вибрати абстракцію, яка дає змогу локалізувати цільові аномалії. У тесті числові змінні, які, ймовірно, будуть використані для доступу до таблиці, представлені у вигляді інтервалу значень. Для інших змінних достатньо зберегти стан, що вказує, чи ініціалізована змінна чи ні.

Програми, що підлягають аналізу, повинні бути трансформовані, щоб мати можливість виявити наявність цільових проблем. Внутрішнє представлення інструкцій вибирається відповідно до обраної абстракції. Внутрішнє представлення програми складається з трьох основних частин: списку даних, які використовує програма, інструкцій для маніпулювання даними та інструкцій, які керують послідовністю виконання програми. У науковій роботі описано внутрішнє представлення, що дозволяє аналізувати програму.

У літературі описано кілька методів статичного аналізу. У цій науковій роботі було обрано один із цих методів: аналіз за регіонами. Цей вибір керувався розглянутою мовою COBOL, оскільки цей метод добре підходить до структури програми COBOL, яка поділена на ієрархії параграфів. Було представлено цей метод аналізу та використано приклад, щоб продемонструвати, як він працює.

Коли статичний аналіз виявляє аномалію, важливо провести точну та детальну діагностику ситуації, щоб дозволити програмісту знайти аномалію та визначити умови, за яких вона може виникнути. Тест показав, як можна забезпечити повне відстеження виконання. У цьому відстеженні інформація про вміст змінних, що призводить до інциденту, допомагає програмісту визначити причини.

Поширеною ситуацією, з якою стикаються при використанні інструменту статичного аналізу, є помилкові спрацьовування. Багато інструментів ризикують виявити аномалії, які насправді не є проблемами. Занадто багато помилкових спрацьовувань може призвести до того, що користувач не буде переглядати всі виявлені аномалії. Можливо, програміст не захоче використовувати цей інструмент, якщо у нього створюється враження, що він витрачає більше часу на перевірку аномалій, які не просто вирішують реальні проблеми. У дослідженні були згадані дві стратегії для управління рівнем діагностичної деталізації та певного контролю над кількістю отриманих помилкових результатів.

Тест показав, як працює інструмент статичного аналізу та як розробити такий інструмент з урахуванням ситуацій, які потрібно виявити. Однак важливо подумати, як використовувати такий інструмент аналізу в процесі розвитку бізнесу. Щоб бути ефективним, програміст повинен використовувати інструмент аналізу, як він використовує налагоджувач. Як тільки програма компілюється без помилок і навіть перед початком її тестування, програміст повинен використовувати статичний аналіз програми, щоб підтвердити, що в програмі немає частих помилок, які може виявити інструмент. Якщо інструмент виявить такі помилки, програміст може виправити їх без необхідності тестування програми. Після того, як інструмент сканування більше не знайде помилок, програміст повинен перейти до модульного тестування.

Важливо зазначити, що інструмент сканування не призначений для заміни тестування, оскільки ці інструменти не можуть виявити всі аномалії. Статичний аналіз дозволяє автоматично виявляти певні категорії помилок. Виявлення та

виправлення цих помилок відбувається швидше за допомогою статичного аналізу, ніж за допомогою тестових випадків.

Тест показав, що внутрішнє представлення даних, тобто абстракція, обрана для змінних, є важливою для полегшення аналізу. Цей вибір репрезентації має бути зроблений з урахуванням ситуацій, які підлягають виявленню.

Як уже згадувалося у науковій роботі, жоден інструмент не може виявити всі аномалії, тому існує прямий зв’язок між складністю інструменту аналізу та точністю результату, який необхідно отримати.

Цікавим напрямком дослідження є можливість розробити структуру глобального аналізу, незалежну від внутрішнього представлення, і яка дасть можливість повторно використовувати алгоритми, представлені в аналізі, з мінімальними змінами для вирішення інших проблемних ситуацій.

У науковій роботі не розглядалися методи аналізу між процедурами, які дозволяють аналізувати процедуру або функцію та зберігати результат аналізу для використання в тих точках, де ці функції використовуються в коді. Ці методи можна було б перевірити.

Список посилань

[1] Ахо, А. В., Лам, М. С., Сеті, Р. Р. Ульман, Дж. Д., упорядники: принципи, методи та інструменти, 2-е видання, Pearson Education Inc., 2007, 1009 с.

[2] Burnstein, I., Практичне тестування програмного забезпечення: Процесно-орієнтований підхід, Спрінгер, 2002, 400 с.

[3] Шахи, Б. і Захід, J., Безпечне програмування зі статичним аналізом, Pearson Education Inc., червень 2007, 587 с.

[4] Кусто, П. і Кусто, Р., Абстрактна інтерпретація: єдина ґратчаста модель статичного аналізу програм шляхом побудови або наближення точок фіксації, запис конференції четвертого щорічного Симпозіуму ACM SIGPLAN-SIGACT з принципів мов програмування, січень 1977 р., с. 1977 р., с. 1). 238–252.

[5] Кусто, П. і Кусто, Р., Статична перевірка властивостей динамічного типу змінних, Дослідницький звіт R.R. 25, Laboratoire IMAG, Університет Гренобля, листопад 1975, 18 с. 18 с.

[6] Кусот, П., і Кузо, Р., На шляху до універсальної моделі статичного аналізу програм, Laboratoire IMAG, Університет Гренобля, січень 1977, 90 с.

[7] Дейві, Б. А. і Прістлі, Х. А., Вступ до решітки і порядку, 2-е видання, Кембриджський університет преси, 2002, 298 с.

[8] Фахрінджер, Т. і Шольц, Б., Розширений символічний аналіз для компіляторів, Спрінгер, 2003, 132 с.

[9] Гласс, Р. Л., Коболєв – протиріччя і загадка, комунікації АКМ, т. 40, No 9, вересень 1997 р., с. 11-13.

[10] Гольцман Г. J., Спін-модель чекер, Аддісон Уеслі, 2004, 597 с.

[11] Као, Г. і Чень, Т. Ю., Аналіз потоку даних для COBOL, Повідомлення ACM SIGPLAN, том 19, No 7, липень 1984 р., с. 18-21.

[12] Lämmel, R. and De Schutter, K., Що означає для Коболя аспектно-орієнтоване програмування? Матеріали 4-ї міжнародної конференції з розробки програмного забезпечення, орієнтованого на аспекти, березень 2005 р., с. 99-110.

[13] Пападімітріу, К. Х., Обчислювальна складність, Аддісон Уеслі Лонгман, серпень 1995, 523 с.

[14] RTI, Економічні наслідки неадекватної інфраструктури для тестування програмного забезпечення, травень 2002, 309 с.

[15] Sintzoff, M., Розрахунок властивостей програм за оцінками за конкретними моделями, повідомлення ACM SIGPLAN, vol. 7, No 1, січень 1972 р., с. 203-207.

[16] Тейшер, К., Алан Тьюринг: Життя і спадщина великого мислителя, Спрінгер 2004, 542 с.

[17] Вільф, Г. С., Генеруюча функція, 3-е видання, А, К. Пітерс, ТОВ, 2006, 242 с.

[18] Gomes, I., Morgado, P., Gomes, T., & Moreira, R. (2009). An overview on the static code analysis approach in software development. Faculdade de Engenharia da Universidade do Porto, Portugal.

[19] Emanuelsson, P., & Nilsson, U. (2008). A comparative study of industrial static analysis tools. Electronic notes in theoretical computer science, 217, 5-21.

[20] Novikov, A. S., Ivutin, A. N., Troshina, A. G., & Vasiliev, S. N. (2017, June). The approach to finding errors in program code based on static analysis methodology. In 2017 6th Mediterranean Conference on Embedded Computing (MECO) (pp. 1-4). IEEE.

[21] Louridas, P. (2006). Static code analysis. Ieee Software, 23(4), 58-61.

[22] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013, May). Why don't software developers use static analysis tools to find bugs?. In 2013 35th International Conference on Software Engineering (ICSE) (pp. 672-681). IEEE.

[23] Panichella, S., Arnaoudova, V., Di Penta, M., & Antoniol, G. (2015, March). Would static analysis tools help developers with code reviews?. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER) (pp. 161-170). IEEE.

[24] Gosain, A., & Sharma, G. (2015). Static analysis: A survey of techniques and tools. In Intelligent Computing and Applications (pp. 581-591). Springer, New Delhi.

[25] Gopan, D., & Reps, T. (2007, August). Guided static analysis. In International Static Analysis Symposium (pp. 349-365). Springer, Berlin, Heidelberg.

[26] Wichmann, B. A., Canning, A. A., Clutterbuck, D. L., Winsborrow, L. A., Ward, N. J., & Marsh, D. W. R. (1995). Industrial perspective on static analysis. Software Engineering Journal, 10(2), 69-75.

[27] Khare, S., Saraswat, S., & Kumar, S. (2011, February). Static program analysis of large embedded code base: an experience. In Proceedings of the 4th India Software Engineering Conference (pp. 99-102).

[29] Кукішев, О. О. (2019). Виявлення проблемних ділянок коду за допомогою інструментарію статичного аналізу.

[30] Harrison, W. (1987, February). An extensible static analysis tool for COBOL programs. In Proceedings of the 15th annual conference on Computer Science (pp. 285-291).

[31] Hennell, M. A., McNicol, W. M., & Hawkins, J. (1980). The static analysis of Cobol programs. ACM SIGSOFT Software Engineering Notes, 5(4), 17-25.

[32] Tischler, R., Schaufler, R., & Payne, C. (1983). Static analysis of programs as an aid to debugging. ACM SIGPLAN Notices, 18(8), 155-158.

[33] Torsun, I. S., & Al-Jarrah, M. M. (1981). Dynamic analysis of COBOL programs. Software: Practice and Experience, 11(9), 949-961.

[34] Giacobazzi, R., Lovato, A., & Mastroeni, I. A Static Analyzer for COBOL-85 Programs.

Нгуен Карина Дамівна
Прізвище, ім’я, по батькові

Факультет інформаційних і прикладних технологій
Факультет

122 «Комп’ютерні науки»
Шифр і назва спеціальності

Сучасні інформаційні технології та програмування
Освітня програма

ДЕКЛАРАЦІЯ АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ

Усвідомлюючи свою відповідальність за надання неправдивої інформа- ції, стверджую, що подана кваліфікаційна (магістерська) робота на тему:
«РОЗРОБКА СИСТЕМИ ДЛЯ СТАТИЧНОГО АНАЛІЗУ ПРОГРАМНОГО КОДУ / DEVELOPMENT OF A SYSTEM FOR STATIC ANALYSIS OF PROGRAM CODE» є написаною мною особисто.

Одночасно заявляю, що ця робота:

- не передавалась іншим особам і подається до захисту вперше;
- не порушує авторських та суміжних прав, закріплених статтями 21-25 Закону України «Про авторське право та суміжні права»;
- не отримувались іншими особами, а також дані та інформація не отримувались у недозволений спосіб.

Я усвідомлюю, що у разі порушення цього порядку моя кваліфікаційна (магістерська) робота буде відхилена без права її захисту, або під час захисту за неї буде поставлена оцінка «незадовільно».

дата підпис