МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра комп'ютерних наук

КРУК Михайло Сергійович

Математичне та програмне забезпечення зниження надлишковості програмних кодів Java-додатків / Mathematical Tools and Software for Code Redundancy Reduction in Java Applications

спеціальність: 121 - Інженерія програмного забезпечення
освітньо-професійна програма - Інженерія програмного забезпечення

Кваліфікаційна робота

Виконав студент групи ІПЗм-21
М. С. Крук

_____

Науковий керівник:
к.е.н., доцент, Л. І. Гончар

_____

Кваліфікаційну роботу
допущено до захисту:

"___" _____ 20___ р.

Завідувач кафедри
_____ **А. В. Пукас**

ТЕРНОПІЛЬ - 2022

ЗМІСТ

ВСТУП

Актуальність теми. Якість програмного забезпечення є критично важливим аспектом розробки програмного забезпечення [1,2]. Використання високорівневих мов програмування, таких як Java, C#, дозволяє підвищити ефективність роботи програміста, скоротити час і вартість розробки та зрештою підвищити якість програмного забезпечення. Автоматичне управління пам'яттю дозволяє позбавитися цілого класу помилок, пов'язаних з управлінням пам'яттю. Сувора типізація, автоматичні перевірки на етапі виконання на рівні мови забезпечують безпеку коду, що виконується. Більш високорівневі абстракції, багаті стандартні бібліотеки дозволяють скоротити кількість помилок за рахунок перевикористання коду. Зручний інструментарій інтегровані середовища розробки підвищують ефективність роботи програміста, що спрощують рефакторинг коду.

У той же час основними мовами програмування для вбудовуваних пристроїв були і залишаються низькорівневі мови, наприклад C або C++. Обчислювальна потужність, обсяг оперативної та енергонезалежної пам'яті вбудованих пристроїв на кілька порядків менше, ніж у персональних комп'ютерів та серверів. Низькорівневі мови програмування дозволяють ефективно використовувати обмежені можливості цільових пристроїв. Недоліком таких мов є висока складність розробки.

Завдання, які вирішуються при розробці додатків для вбудованих пристроїв, мають багато спільного, при цьому застосовувані технології часто не дозволяють перевикористовувати існуючі рішення. Усі, хто стикається з розробкою додатків для вбудованих пристроїв, вимушені реалізовувати ту саму функціональність, перш ніж почати реалізовувати бізнес-логіки програми. Прикладом такої функціональності є взаємодія з периферією за

допомогою стандартних інтерфейсів, мережева взаємодія, шифрування тощо [3-5]. Якийсь час тому ці проблеми були актуальними і при розробці для персональних комп'ютерів та серверів. Вирішенням даних проблем стало застосування вищого рівня та безпечних мов програмування та платформ, таких як Java та .NET.

Застосування мови Java для розробки програмного забезпечення для вбудовуваних пристроїв дозволяє істотно скоротити час і вартість розробки та підвищити якість цільового продукту.

Можливість застосування мови Java для вбудованих пристроїв розвивалася від початку його розробки. Однак ця мова пред'являє більш високі вимоги до ресурсів, порівняно з мовами C чи C++. Це плата, яку доводиться платити за високорівневі абстракції, безпеку та велику стандартну бібліотеку. Реалізація Java-платформи для обмежених ресурсних пристроїв повинна бути оптимізована за розміром, а не за швидкістю виконання.

Стандартна модель поширення Java-додатків передбачає, що на пристрій може бути завантажено та виконано довільну програму. При цьому реалізація Java-платформи повинна підтримувати все, що описується у її специфікації. Така модель називається відкритою. Для вбудованих пристроїв набір виконуваних програм зазвичай визначено заздалегідь, і немає необхідності виконувати довільні програми. Модель, в якій весь виконуваний код відомий заздалегідь, називається закритою. У закритій моделі реалізація Java-платформи може бути спеціалізована для заданого додатку. Так, наприклад, у закритій моделі можна проаналізувати, які можливості платформи використовуються програмою, і видалити невикористовувані.

Таку спеціалізацію називатимемо зниженням надмірності. Крім того, в закритій моделі для заданого додатку можна розробити спеціалізований набір інструкцій, який зменшить сумарний розмір виконуваного коду та інтерпретатора, необхідного для його виконання.

Зв'язок роботи з науковими програмами, планами, темами

Напрямок виконаних досліджень безпосередньо пов'язаний з науково-дослідним напрямком кафедри “комп’ютерних наук” Західноукраїнського національного університету.

Мета і задачі дослідження

Метою роботи є розробка методів та засобів для зниження надлишковості програмних кодів Java-додатків.

Відповідно до поставленої мети у роботі потрібно вирішити такі основні завдання дослідження:

1. Дослідити відомі алгоритми зниження надмірності програм з точки зору їх застосовності при використанні роздільної ініціалізації.

2. Дослідити відомі алгоритми стиснення бінарного коду. Вивчити їх застосування для стиснення Java байт-коду в закритій моделі.

3. Розробити алгоритми зниження надмірності Java-программ, застосовні під час окремої ініціалізації.

4. Розробити алгоритм стиснення Java байт-коду в закритій моделі, застосовний для вбудованих систем з обмеженими ресурсами.

5. Реалізувати запропоновані алгоритми та експериментальним шляхом досілдити їх ефективність.

Об’єкт дослідження – процеси зниження надлишковості програмних кодів Java-додатків.

Предмет дослідження – методи та програмні засоби для зниження надлишковості програмних кодів Java-додатків.

Методи дослідження

В роботі використовувалися методи теорії компіляторів, теорії множин і теорії графів, теорії алгоритмів та математичної логіки, а також методи розробки програмного забезпечення.

Наукова новизна одержаних результатів

Запропоновано метод еквівалентного перетворення Java-додатків та інтерпретатора, необхідного для його виконання, що скорочує їх сумарний розмір за допомогою спеціалізації набору інструкцій інтерпретатора.

Практичне значення одержаних результатів

В рамках виконання магістерського дослідження розроблено програмне забезпечення для зниження надлишковості програмних кодів Java-додатків.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ ПІДХОДІВ СКОРОЧЕННЯ РОЗМІРУ ПРОГРАМНИХ ДОДАТКІВ

1.1. Аналіз методів скорочення розміру програмних додатків.

Розглядається завдання зниження надмірності шляхом видалення методів, полів і класів, що не використовуються. Наводиться огляд стандартних алгоритмів аналізу досяжності методів, полів та класів. Описується механізм роздільної ініціалізації (ромізації). Відзначаються обмеження існуючих алгоритмів зниження надмірності, які дозволяють використовувати їх при ромізації.

Інший розглянутий механізм скорочення розміру програми полягає у використанні інтерпретованого подання замість машинного коду. Для заданої програми в закритій моделі можна досягти більш ефективного кодування шляхом спеціалізації набору інструкцій, що інтерпретується. У розділі наведено огляд існуючих механізмів спеціалізації набору інструкцій. В огляді зазначаються недоліки існуючих алгоритмів спеціалізації Java байт-коду.

Якщо метод f може бути викликаний у результаті виконання методу g, метод f вважається досяжним із методу g. Для знаходження множини методів, які можуть бути виконані при виконанні програми, будується транзитивне замикання методів, що досягаються з точок входу програми. Аналіз досяжності способів – ключовий аналіз при зниженні надмірності, оскільки всі наступні аналізи аналізують код способів, виділених цьому етапі.

Для мов з динамічною диспетчеризацією обчислення множини методів, досяжних з цього, є нетривіальним завданням. У таких мовах викликаний метод який завжди визначається статично. Найбільш поширеним прикладом динамічної диспетчеризації є віртуальні виклики.

Механізм віртуальних викликів використовується для реалізації поліморфізму у багатьох об'єктно-орієнтованих мовах програмування. При наявності динамічної диспетчеризації множина досяжних методів доводиться обчислювати консервативно. Обчислена множина досяжних методів є надмножиною множини методів, які будуть виконані при виконанні програми. Кількість «зайвих» методів залежатиме від точності аналізу.

При зниженні надмірності від точності аналізу досяжності методів залежить точність подальших аналізів віддаленості та підсумкова ефективність алгоритму.

У мові Java є два механізми динамічної диспетчеризації – це віртуальні та інтерфейсні виклики. В обох випадках метод визначається на етапі виконання в залежності від типу аргументу одержувача.

Аналогічно прямим викликам, віртуальні та інтерфейсні виклики містять статичний опис викликаного методу, проте при виконанні такі виклики зв'язуються з реалізацією зазначеного методу класу об'єкта-отримувача.

```
class Base {
  void foo() {
    System.out.println("Base.foo");
  }
}
class Derived {
  void foo() {
    System.out.println("Derived.foo");
  }
}
Base object = factory();
object.foo();
```

Рис. 1.1. Приклад типізації об’єктів

У даному прикладі в залежності від типу об’єкта object може бути визваний або метод Base.foo, або метод Derived.foo. При статичному аналізі інформація про об’єкти часу виконання. Можна вважати, що виклик може бути пов'язаний із будь-яким методом, сумісним на ім'я та сигнатуру. Однак

такий підхід є вкрай неефективним. Зазвичай для аналізу досяжності використовується консервативне наближення множини типів об'єктів-одержувачів. Розглянемо існуючі підходи до обчислення цієї множини.

Алгоритми Class Hierarchy Analysis та Rapid Type Analysis. У найпростішому випадку вважатимуться, що багато типів об'єкта одержувача обмежено статичним типом одержувача. Тобто для Java кількість можливих типів - це множина всіх підкласів статичного класу об'єкта-отримувача.

Алгоритм CHA є неефективним при аналізі універсальних бібліотек. Такі бібліотеки часто містять велику кількість різних реалізацій базових інтерфейсів чи класів. Зазвичай додаток використовує невелику частину цих реалізацій, але для більшої гнучкості при роботі з об'єктами використовуються інтерфейси базових типів.

Прикладом може бути бібліотека стандартних колекцій Java. У пакет java.util містить 10 класів, що реалізують інтерфейс List. Для наступного коду досяжними вважатимуться реалізації методу add всіх 10 класів:

```
List<String> list = new ArrayList<String>();
list.add("one");
```

Точність аналізу можна підвищити, якщо врахувати, що нестатичний метод класу може бути викликаний тільки тоді, коли існують або можуть бути створено об'єкти цього класу або його підкласів. Уточнений алгоритм називається Rapid Type Analysis (RTA) [5]. У процесі роботи алгоритму вирахується множина непрямих викликів і множина класів, екземпляри яких можуть бути використані для непрямих дзвінків. Позначимо ці множини *IndirectInvocations і InstantiableClasses* відповідно. Ітеративний алгоритм починається з додавання до замикання точок входу програми. Алгоритм завершується досягненням нерухомої точки. На кожній ітерації виконуються такі дії.

1. Для кожного раніше не обробленого методу із замикання переглядається його код:

а) методи, що викликаються інструкціями прямого виклику, додаються у замикання;

б) непрямі виклики додаються до множини *IndirectInvocations*;

в) класи, які використовуються для створення об'єктів за допомогою інструкції, додаються до множини *InstantiableClasses.*

2. Для кожного виклику з множини *IndirectInvocations* обчислюється переріз множини підкласів статичного класу виклику з множиною класів. Отриманий перетин – це класи, екземпляри яких можуть бути використані як об'єкти-одержувачі для виклику. Реалізації викликаного методу в даних класах добавляться в замикання.

Класи, що належать множині *InstantiableClasses*, далі будемо називатися інстанційними.

Продемонструємо різницю між алгоритмами CHA і RTA на прикладі із рисунку 1.2. Результат аналізу досяжності для даної програми приведений на рисунку 1.3.

```
class A {
  void bar() {};
}
class B extends A {
  void bar() {};
}
class C extends A {
  void bar() {};
}
class Main {
  public static void main(String[] args) {
    new B();
    foo(new A(), new A(), null);
  }

  static void foo(A a, A b, A c) {
    c.bar();
  }
}
```

Рис. 1.2. Приклад програми для аналізу досяжності методів

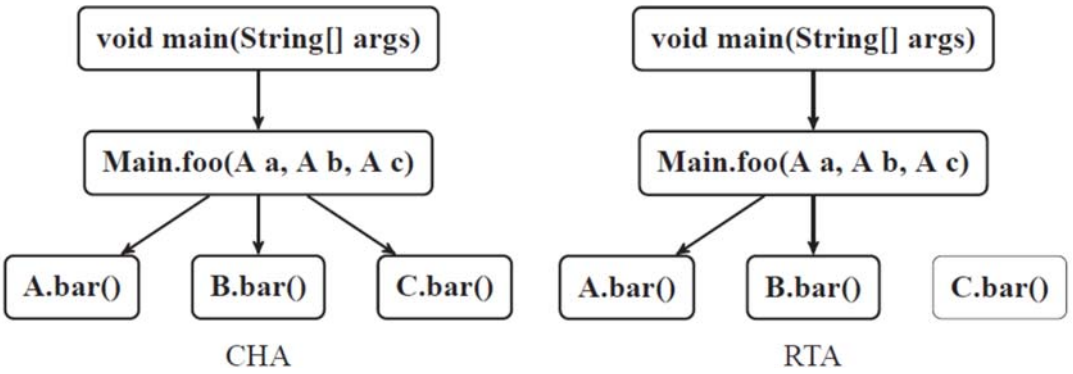


Рис. 1.3. Результат аналізів CHA та RTA

Жирним виділено досяжні методи. Ребра позначають відношення досяжності.

Алгоритм X Type Analysis. Для оцінки можливих типів об'єктів у будь-якій точці програми алгоритм RTA використовує одну множину InstantiableClasses. Точніші алгоритми будують граф потоку даних програми і обчислюють множину типів, що досягають, для кожного вузла в графі. Алгоритми відрізняються точністю моделювання потоку даних. Наприклад, алгоритм XTA, описаний у [12], моделює потік даних між методами програми У мові Java між методами об'єкти передаються через аргументи, значення, що повертаються, і посилання в купі (посилання можуть утримуватися в полях об'єктів та елементах масивів). Ще одна ситуація, коли об'єкти передаються між методами, – це скидання винятків.

Оскільки скинутий виняток може бути оброблено будь-яким методом, що знаходиться вище за стеком викликів, такі аналізувати складніше. Однак так можуть бути передані лише об'єкти-виключення – екземпляри підкласів класу java.lang.Throwable. Враховуючи, що множина підкласів класу java.lang.Throwable зазвичай невелика, запропонований алгоритм не моделює скидання винятків точно, а використовує одну множину для опису можливих підкласів класу java.lang.Throwable у будь-якій точці програми. Потік даних можна моделювати з різною точністю.

1. CTA – для кожного класу використовується одна множина, що описує, що досягають типи для всіх полів та методів цього класу.

2. MTA – для кожного класу використовується одна множина, що описує типи для полів цього класу. Кожному методу відповідає одна множина типів, що досягають.

3. FTA – для кожного класу використовується одна множина, що описує типи, що досягають, для методів цього класу. Кожному полю відповідає одна множина типів, що досягають.

4. XTA – для кожного методу та поля використовуються окремі множини.

На рисунку 1.4 відображено, запропоновані алгоритми, порівняно з алгоритмами аналізу досяжності методів, описаними раніше. Від менш точних ліворуч до точніших праворуч.

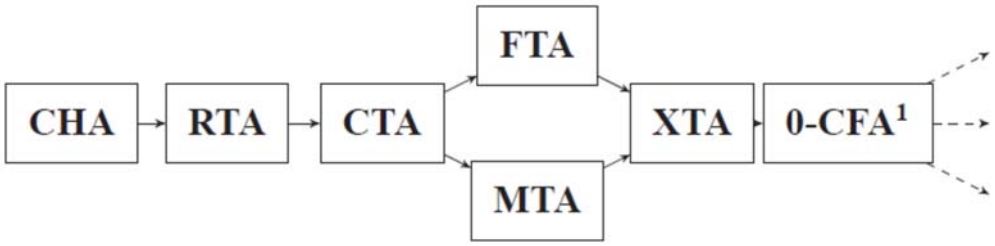


Рис. 1.4. Алгоритми аналізу досяжності методів

При моделюванні потоку даних алгоритм не враховує порядок виконання програми та стан локальних змінних. Результат аналізу XTA для програми наведено рисунку 1.5.

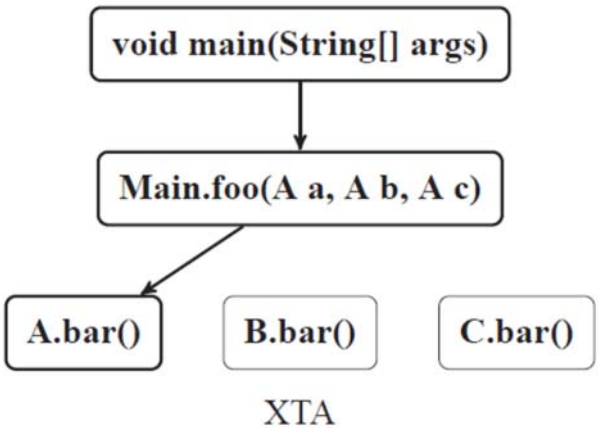


Рис. 1.5. Результат аналізу XTA

Використання алгоритму XTA дозволяє скоротити кількість досяжних методів у середньому на 1,6% порівняно з алгоритм RTA. Жирним виділено досяжні методи. Ребра позначають відношення досяжності.

1.2. Алгоритми Declared Type Analysis, Variable Type Analysis

Іншим прикладом більш точного аналізу є алгоритм Variable-Type Analysis (VTA), який моделює потік даних між змінними. Алгоритм VTA будує тип propagation граф, вузли якого моделюють змінні посилальних типів, а саме:

- локальні змінні,
- поля,
- елементи масивів,
- аргументи,
- значення, що повертаються.

Різні екземпляри однієї змінної моделюються за допомогою одного вузла в графі. Ребра в графі позначають можливі присвоєння між модельованими об'єктами під час виконання програми. Передача об'єктів через аргументи і значення, що повертаються, моделюється аналогічно присвоєнню.

На відміну від алгоритму RTA, алгоритм VTA – песимістичний. Для моделювання передачі об'єктів між функціями необхідно знати, з якими методами може бути пов'язаний кожний виклик. Для непрямих викликів це заздалегідь невідомо. Для вирішення цієї проблеми алгоритм VTA використовує заздалегідь обчислений консервативний граф викликів. Такий граф може бути збудований за допомогою алгоритмів CHA чи RTA. При моделюванні передачі об'єктів через аргументи і значення непрямих викликів, що повертаються, передбачається, що непрямий виклик може бути пов'язаний з будь-яким з методів консервативного графіка викликів. Таким

чином, ефективність алгоритму залежить від вихідного консервативного графіку викликів.

За наявності методів їх операції також моделюються в type propagation графі, проте їх код не аналізується автоматично, а описується вручну.

З кожним вузлом у type propagation графі асоціюється множина типів ReachingTypes. Це багато класів, на екземпляри яких може вказувати відповідна змінна. Для обчислення множин досягаючих типів у коді методів аналізуються присвоювання виду $lhs = new\ A()$. Для кожного вираження такого виду в множину $ReachingTypes$ вузла, що описує змінну lhs, додається клас A. Потім множина $ReachingTypes$ для кожного вузла обчислюється як об'єднання множини відтворення типів всіх вузлів доступних з даного.

Запропоновано варіацію алгоритму під назвою Declared-Type Analysis (DTA), яка моделює різні змінні одного типу за допомогою одного вузла у графі.

Описані алгоритми потоково-нечутливі, тобто не враховують порядок виконання програми. Результат аналізів VTA та DTA для програми наведено на рисунок 1.6. Жирним виділено досяжні методи. Ребра позначають відношення досяжності.

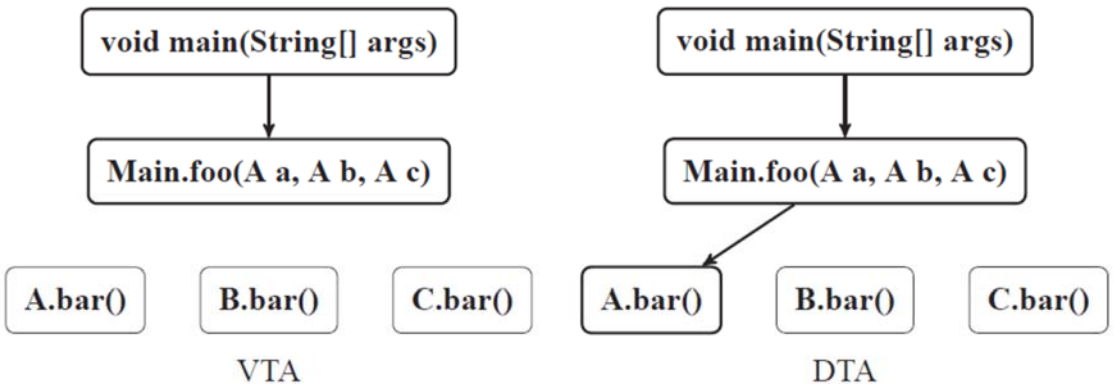


Рис. 1.6. Результат аналізів VTA та DTA

Згідно з даними, наведеними у [6-10] авторами, використання алгоритму VTA скорочує кількість досяжних методів порівняно з алгоритмом RTA загалом на 11,4 %. Для алгоритму DTA ця цифра значно менша – 3%. Ці дані та цифри, наведені для алгоритму XTA, отримані різними авторами для різного набору додатків та не можуть бути використані для безпосереднього порівняння.

1.3. Зниження надлишковості Java-додатків

Вищеописані алгоритми часто застосовують зниження по надмірності Java програм. Так, у статтях [11-16] автори описують інструмент Jax Application Extractor (рисунок 1.7), який здійснює виділення мінімального підмножини бібліотек для заданої програми.

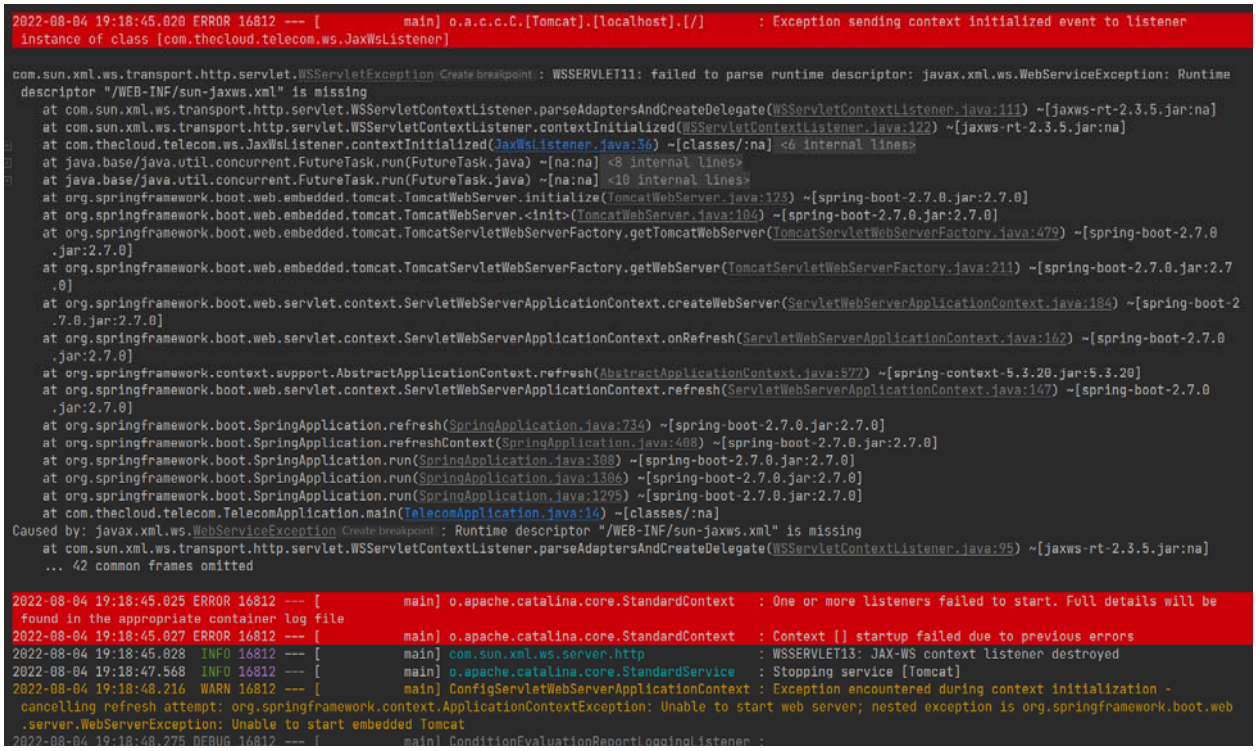


Рис. 1.7. Інструмент Jax Application Extractor

Основні можливості інструменту.

- Видалення недосяжних методів. Для аналізу досяжності використовують алгоритми CHA, RTA, XTA.

– Видалення полів, що не використовуються, для яких в досяжному коді немає операцій читання.

– Видалення класів, що не використовуються.

- Об'єднання суміжних класів в ієрархії наслідування за умови, що таке перетворення не збільшує розмір екземплярів класу.

- Перейменування класів, методів та полів.

Автори зазначають, що залежність рефлексивного коду в загальному випадку не можуть бути проаналізовані статично. Цей інструмент дозволяє описати такі залежності вручну. Крім того, зазначається, що аналізований Java-код може містити native-методи, реалізація яких написана іншою мовою, найчастіше C/C++. Native-методи можуть створювати екземпляри класів, викликати інші методи Java, звертатися до полів. Такі залежності також мають бути описані вручну.

Віртуальна машина JamaicaVM може бути спеціалізована для заданої програми за допомогою інструмента Jamaica Builder (рисунок 1.8). Результатом роботи інструменту є виконуваний файл, що включає віртуальну машину, додаток і бібліотеки, необхідні для його виконання.

Під час спеціалізації можуть бути видалені методи, поля та класи, які не використовуються додатком. Документація віртуальної машини не описує алгоритми, але деякі властивості використовуваних алгоритмів були визначені експериментальним шляхом.

– Аналіз досяжності методів використовує алгоритм, аналогічний алгоритму RTA.

– Аналіз видалення полів дозволяє видаляти поля примітивних типів, для яких у коді досяжних методів є операції запису, але немає операцій читання. Для полів об'єктних типів така оптимізація не виконується.

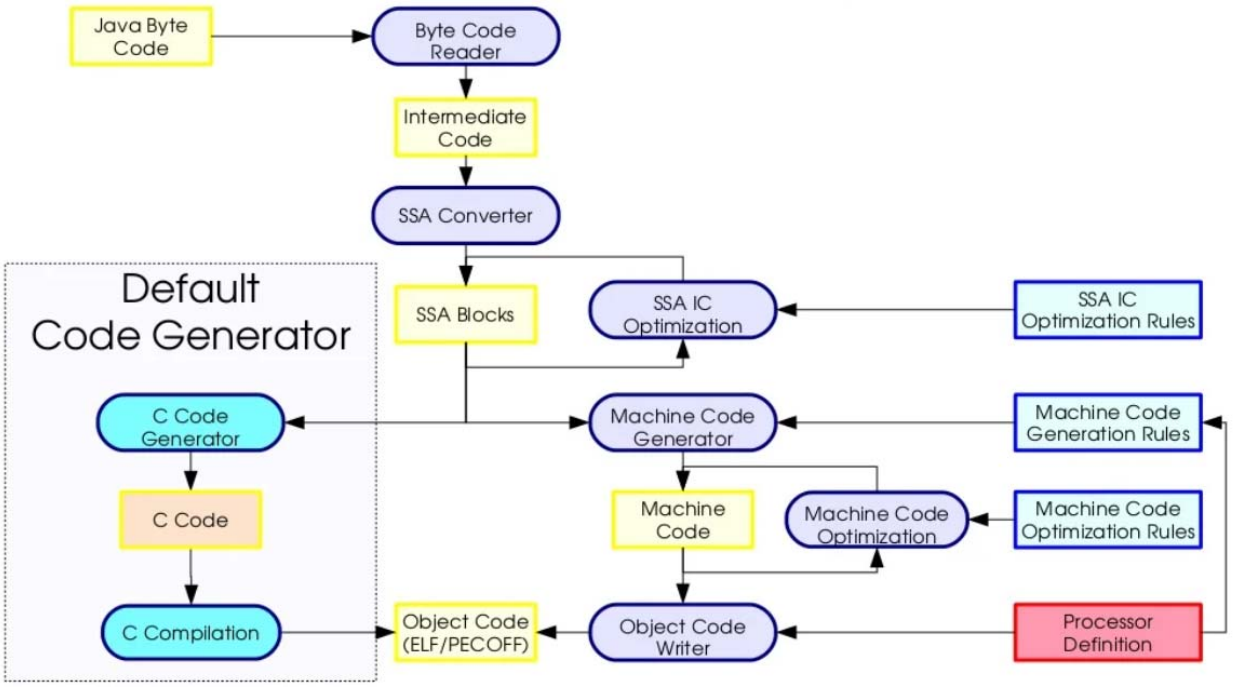


Рис. 1.8. Інструмент Jamaica Builder

Jamaica Builder надає можливість вказати класи, які мають бути безумовно включені до спеціалізованої віртуальної машини. У підсумковий файл, що виконується, будуть включені всі методи і поля даних класів незалежно від результатів аналізу досяжності. Дана можливість повинна бути використана для опису залежностей native-методів та коду, що використовує рефлексію.

У деяких віртуальних машинах алгоритми зниження надмірності використовуються оптимізації завантажувального образу при роздільній ініціалізації. Механізм роздільної ініціалізації, описаний в [20], був реалізований системі Java In The Small (рисунок 1.9). Під час підготовки образу на хостівому пристрої здійснюється ініціалізація потоків програми, після чого видаляються методи, поля, об'єкти та класи, що не використовуються додатком. Для обчислення множини досяжних методів використовується алгоритм CHA. У цьому автори не описують алгоритми аналізу віддаленості полів і класів.

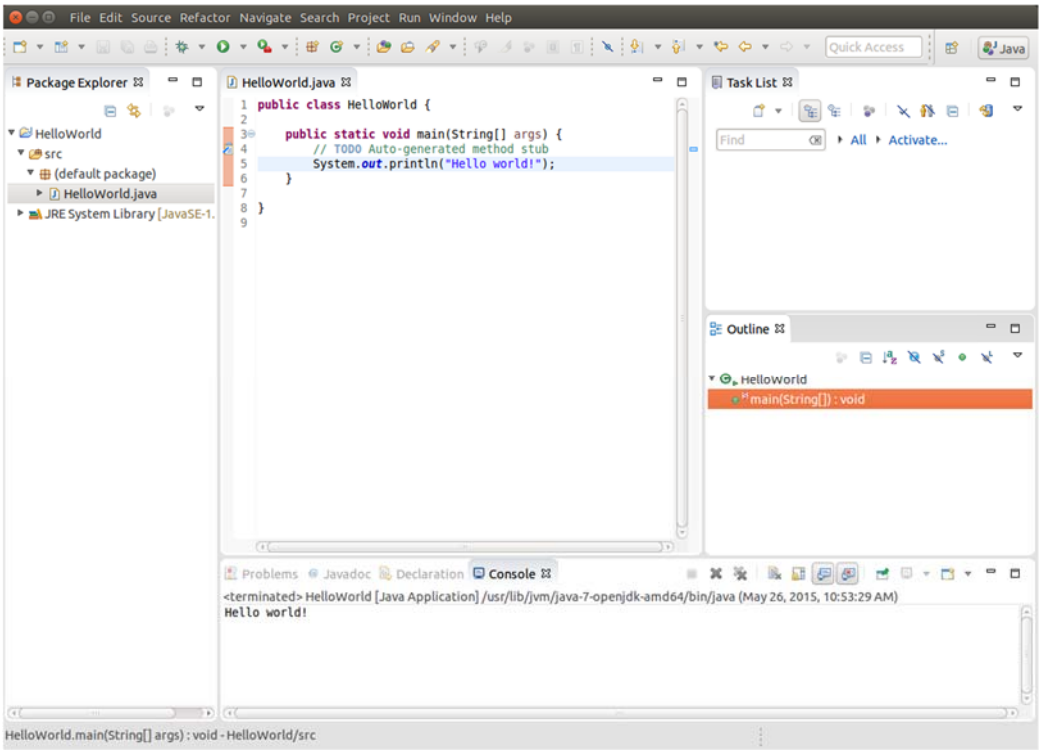


Рис. 1.9. Інструмент Java In The Small

Інший приклад використання алгоритмів зниження надмірності для оптимізації завантажувального образу описано у статті [21]. Авторами описується система ExoVM (рисунок 1.10), що автоматично спеціалізує віртуальну машину для заданої програми. Для прискорення запуску та скорочення споживання пам'яті ExoVM використовує окрему ініціалізацію.

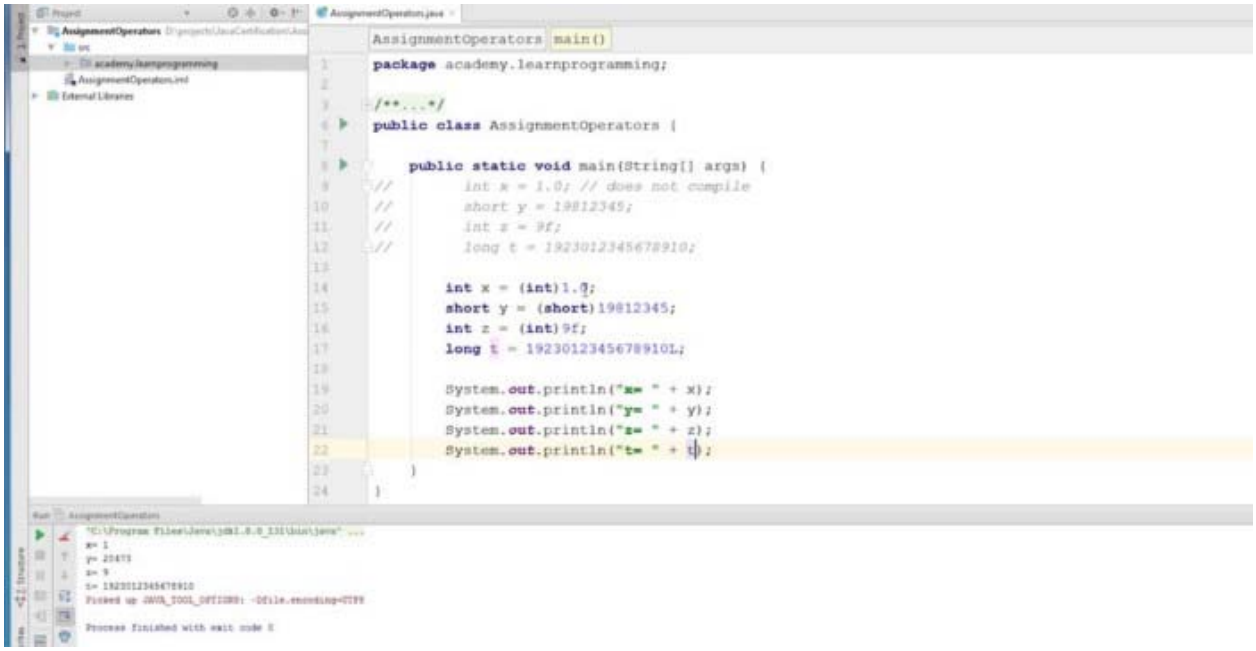


Рис. 1.10. Інструмент ExoVM

Програма завантажується та ініціалізується повнофункціональною версією віртуальної машини. У процесі ініціалізації дозволяються всі символічні посилання та ініціалізуються всі завантажені класи. При цьому у статті зазначається, що специфікація мови вимагає лінивого завантаження та ініціалізації, і що рання ініціалізація всіх класів програми може порушити її поведінку.

Після того, як усі класи програми ініціалізовані, здійснюється аналіз досяжності, званий feature analyis. На відміну від Jax Application Extractor, система ExoVM аналізує досяжність не тільки для сутностей мови, а й для окремих компонентів віртуальної машини. У статті визначаються відношення досяжності між різними сутностями мови (методами, полями, об'єктами та класами) та компонентами віртуальної машини. У завантажувальний образ спеціалізованої віртуальної машини включаються лише ті сутності та компоненти, які можна досягти з точок входу програми.

Оскільки аналіз досяжності здійснюється після ініціалізації класів, на даний момент аналізу у пам'яті можуть існувати об'єкти, створені у процесі ініціалізації. Ці об'єкти необхідно враховувати під час аналізу досяжності методів. Для аналізу досяжності методів ExoVM використовує підхід, що ґрунтується на алгоритмах RTA та RMA. А саме можливими об'єктами-одержувачами непрямих викликів вважаються:

– об'єкти, що інстанціюються досяжним кодом;

– інстанційовані об'єкти, які можна досягти через посилання в полях, що читаються досяжним кодом.

Для видалення полів, що не використовуються, ExoVM використовує критерій видаляємості. Видаляємими вважаються поля, для яких у досяжному коді немає операцій читання. Цей критерій дозволяє видаляти поля, що не використовуються, що ініціалізуються в конструкторах або ініціалізаторах класів. Однак, як було зазначено раніше, така оптимізація порушує семантику фіналізації об'єктів і може змінити видиму поведінку програми.

При аналізі досяжності залежності native-методів не аналізуються автоматично. Натомість реалізація аналізу містить опис залежностей для відомих native-методів стандартної бібліотеки класів. Якщо в аналізованому додатку виявляється досяжним native-метод, для якого невідомі залежності, аналіз зупиняється з помилкою.

У [23] розглядається ще один приклад системи виділення мінімальної підмножини бібліотек для заданої Java-програми. Виділення підмножини здійснюється за допомогою графа залежностей. Вузли у графі описують сутності програми та бібліотек, такі як класи, методи, поля, інтерфейсні та віртуальні виклики. Ребра у графі відображають залежності між ними, наприклад, відносини спадкування між класами, залежності між методами та визначальними їх класами, залежності методів, що визначаються їх байт-кодом. Залежності непрямих викликів обчислюються за допомогою аналізу CHA. Згодом ці залежності уточнюються за допомогою аналізу типів, що інстанціюються (RTA). Мінімальна підмножина бібліотек обчислюється як транзитивне замикання графа залежностей для заданих точок входу програми. Залежності native-методів та коду, що використовує рефлексію, пропонується описувати як додаткові точки входу програми.

Варто зазначити, що залежності, описані таким чином, будуть включені в замикання незалежно від того, включений чи залежить від них код у замикання чи ні.

Усі вищеописані реалізації зниження надмірності реалізують консервативний аналіз використовуваності компонентів. У підсумковий додаток та бібліотеки включаються компоненти, котрим не вдалося довести невикористовуваність. Більш складні та точні алгоритми аналізу дозволяють довести невикористання більшої кількості компонентів, тобто дозволяють скоротити підсумковий розмір програми та бібліотек. У статтях [22-29] пропонується альтернативний підхід, при якому цільовий пристрій може завантажувати відсутні компоненти мережі. Це дозволяє видаляти компоненти, що не використовуються, більш агресивно. Навіть якщо в

результаті аналізу не вдалося довести невикористання компонента, цей компонент може бути спекулятивно видалений.

Якщо компонент буде згодом необхідний для виконання програми, він буде завантажений через мережу. Така конфігурація дозволяє скоротити споживання пам'яті на цільовому пристрої, порівняно з реалізаціями, що використовують консервативний аналіз.

1.4. Постановка задачі дослідження

Завдання скорочення розміру Java байт-коду часто сприймається як частина більш загального завдання компресії Java клас-файлів. Стандартний формат клас-файлів не дуже ефективний з погляду розміру. Більшу частину розміру клас-файлів складає символічна інформація з пулу констант, що описує імена методів, полів та класів. На практиці реалізації Java для вбудованих пристроїв часто використовують альтернативні формати клас-файлів.

Метою роботи є розробка методів та засобів для зниження надлишковості програмних кодів Java-додатків.

Відповідно до поставленої мети у роботі потрібно вирішити такі основні завдання дослідження:

1. Дослідити відомі алгоритми зниження надмірності програм з точки зору їх застосовності при використанні роздільної ініціалізації.

2. Дослідити відомі алгоритми стиснення бінарного коду. Вивчити їх застосування для стиснення Java байт-коду в закритій моделі.

3. Розробити алгоритми зниження надмірності Java-программ, застосовні під час окремої ініціалізації.

4. Розробити алгоритм стиснення Java байт-коду в закритій моделі, застосовний для вбудованих систем з обмеженими ресурсами.

5. Реалізувати запропоновані алгоритми та експериментальним шляхом досілдити їх ефективність.

Висновки до першого розділу

1. Зниження надмірності шляхом видалення полів методів і класів, що не використовуються, широко застосовується для скорочення розміру програми у закритій моделі. Завдання зниження надмірності добре вивчена. Багато відомих алгоритмів були запропоновані для мови C++ і потім були адаптовані для Java.

2. Використання інтерпретованого коду дозволяє скоротити розмір програми у порівнянні з машинним кодом. Спеціалізація набору інструкцій для заданої програми дозволяє досягти ще більш компактне кодування. Таке кодування не вимагає попередньої декомпресії і тому широко застосовується для вбудованих систем із обмеженими ресурсами.

3. Спеціалізовані інструкції забезпечують більш ефективне ко дування за рахунок наступних оптимізацій:

– згортка аргументу – значення часто використовуваного аргументу зафіксовано інструкцією і не кодується в потоці інструкцій;

– укорочування аргументу – аргумент інструкції кодується у потоці інструкцій більш компактно, ніж в оригінальній інструкції;

- згортання послідовності інструкцій - одна інструкція кодує послідовність декількох кодів.

РОЗДІЛ 2
МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ ЗНИЖЕННЯ НАДЛИШКОВОСТІ ПРОГРАМНИХ КОДІВ JAVA-ДОДАТКІВ

2.1. Ініціалізація класів Java-додатків

Відповідно до специфікації мови Java ініціалізація класу відбувається за першим зверненням до класу. Звертанням вважається:

- виконання однієї з інструкцій *new, invokestatic, getstatic,putstatic*;

– рефлексивний доступ до класу, наприклад, за допомогою методу *Class.forName*.

Перед ініціалізацією класу ініціалізуються суперкласи. У процесі ініціалізації виконується статичний ініціалізатор класу – метод *<clinit>.*

При підготовці завантажувального образу деякі класи можуть бути ініціалізовані заздалегідь. Така оптимізація скорочує час ініціалізації програми на цільовому пристрої. Крім того, рання ініціалізація прискорює виконання програми. У загальному випадку інструкції *new, invokestatic, getstatic, putstatic* повинні перевіряти ініціалізований чи клас, до якого відбувається звернення. Після того, як клас ініціалізований, інструкції, що посилаються на даний клас, можуть бути замінені швидкими версіями, яких така перевірка відсутня. Якщо код методів розміщується в незміненій пам'яті, дана оптимізація неможлива на етапі виконання.

Ініціалізація класу скорочує кількість досяжних методів, оскільки метод *<clinit>* не може бути викликаний стандартними інструкціями і після ініціалізації стає недосяжним. У той же час, при ініціалізації можуть створювати об'єкти, які збільшують розмір образу. У деяких випадках - їх рання ініціалізація скорочує розмір образу. Це відбувається тоді, коли сумарний розмір методів, що стали недосяжними, виявляється більшим, ніж сумарний розмір створених під час ініціалізації об'єктів.

Загалом рання ініціалізація класів порушує специфікацію мови та може змінити видиму поведінку програми. Тому не всі класи можна ініціалізувати заздалегідь. Результат ініціалізації деяких класів може залежати від послідовності ініціалізації, яка у свою чергу визначається послідовністю виконання програми. Ініціалізація класу може мати побічні ефекти, наприклад, ініціалізатор може здійснювати операції введення-виведення. Результат роботи ініціалізатора може залежати від того, викликаний він на хості або на цільовому пристрої. Наприклад, ініціалізатор може використовувати метод *java.lang.System.getProperty()*, результат роботи якого відрізняється на різних пристроях. Виконання ініціалізатора може призводити до скидання виключення. Якщо ініціалізація відбулася в процесі виконання програми при першому зверненні до класу, скинутий виняток може бути оброблено кодом, що викликав ініціалізацію. За ранньої ініціалізації такі винятки не можна обробити.

Не будемо автоматично ініціалізувати клас, якщо в його ініціалізації присутня хоча б одна з наступних операцій.

- Цикл. Ініціалізатор із циклами може ніколи не завершитися.

- Виклик методів. Зокрема, заборонено виклик конструкторів об'єктів.

Якщо в коді ініціалізатора є виклик методу, то для аналізу безпеки ініціалізації потрібен аналіз коду методів.

– Звернення до статичного поля класу, якщо цей клас не виявляється класом, що ініціалізується, або одним з його суперкласів. Дана операція може призвести до небезпечної ініціалізації класу. Якщо поле належить класу, що ініціалізується, або його суперкласу, то на момент виконання операції відповідний клас вже буде ініціалізований.

Ця евристика може виявитися занадто консервативною. Розробнику програми надано можливість скасувати заборону на ініціалізацію при допомозі анотації *@InitAtBuild.*

З одного боку, ініцілізація класів впливає на досяжність методів. Методи, що використовуються виключно для ініціалізації класів, стають

недосяжними після ініціалізації та можуть бути видалені. З іншого боку, до аналізу досяжності невідомо, які класи можна використовувати досяжним кодом. Таким чином, ініціалізація класів до аналізу досяжності може призвести до ініціалізації класів, які не будуть використані у процесі виконання.

Ініціалізація таких класів може порушити поведінку програми за рахунок виконання мертвого коду. Крім того, ініціалізація зайвих класів може призвести до створення об'єктів, що не використовуються, але досяжних. Рисунок 2.1 ілюструє циклічні залежності між ініціалізацією класів та аналізом досяжності методів. Для того, щоб дозволити ці залежності, будемо ініціалізувати класи у процесі побудови замикання.

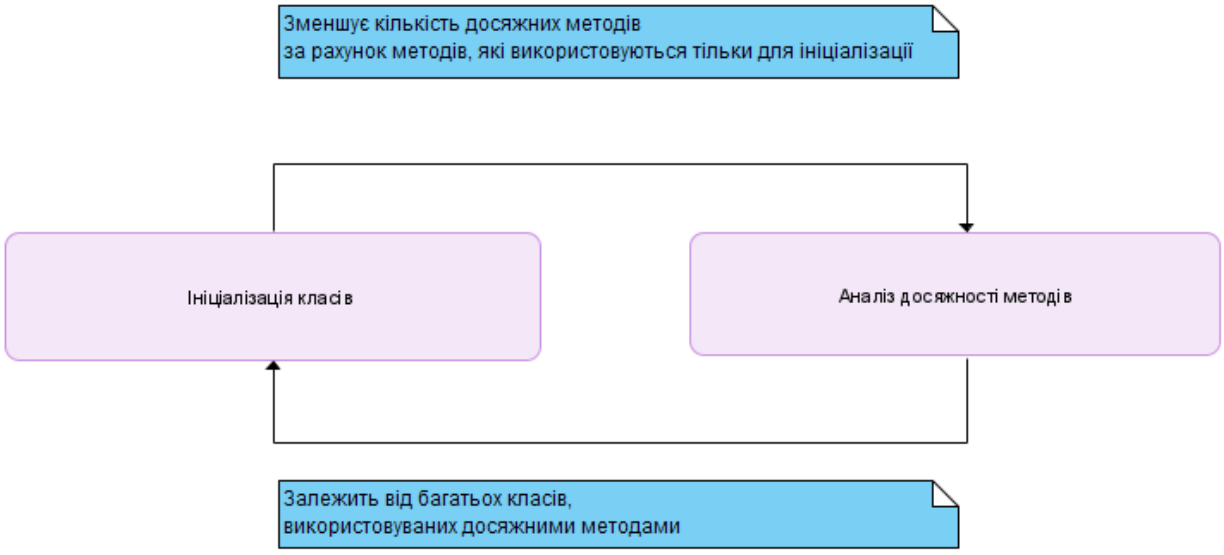


Рис. 2.1. Залежності між ініціалізацією класів та аналізом досяжності методів

Для кожного потенційно ініціалізованого класу існує вибір: ініціалізувати клас у процесі побудови замикання або відкласти ініціалізацію до етапу виконання. У першому випадку метод *<clinit>* має бути виконаний, у другому – повинен бути доданий у замикання, його залежності повинні бути проаналізовані.

При додаванні методу в замикання ми аналізуємо його код та консервативно оцінюємо ефекти його виконання. Наприклад, аналізуючи

інструкції *new* в тілі методу, алгоритм RTA обчислює множину класів, які можуть бути інстанційовані методом. Для методів, які були виконані в процесі аналізу, замість консервативної оцінки ми можемо точно проаналізувати ефекти їхнього виконання. Так, наприклад, для аналізу досяжності непрямих викликів можна точно обчислити множину об'єктів, створених у процесі викоконання методу.

Якщо під час виконання ініціалізатора відбувається скидання виключення, створення образу завершується помилкою. Таким чином, для деяких коректних з точки зору специфікації програм не можна створити завантажувальний образ.

Java – мова з автоматичним керуванням пам’яттю. Складальник сміття (GC) відстежує досяжність об'єктів у купі і видаляє недосяжні об'єкти. Досяжними вважаються об'єкти, які можуть бути використані в будь-якому з можливих продовжень виконання програми. Суворіше досяжність об'єктів визначається як досяжність за посиланнями в об'єктних полях.

Програма має кілька способів відстежити видалення об'єкта. Перший із них – визначити метод *finalize* у класі об'єкта. Метод *finalize* визначений у класі *java.lang.Object* і може бути перевизначений у підкласи. Фіналізатором класу називатимемо реалізацію методу *finalize* в даному класі. Фіналізатором об'єкта називається фіналізатор класу об'єкта.

Фіналізатор об'єкта викликається при знищенні об'єкта збирачем сміття. Клас *Object* містить пусту реалізацію методу *finalize*. Об'єкт з непустим фіналізатором називатимемо фіналізованим. Інший спосіб складається у встановленні на об'єкт спеціального посилання. Спеціальні посилання в Java передставлені класами *WeakReference*, *SoftReference*, *PhantomReference*.

Спеціальні посилання не впливають на досяжність об'єктів і обнулюються, якщо об'єкт, на який вказувало посилання, було знищено збирачем сміття. Об'єкти, видалення яких додаток може відстежити, будемо

називати невдалими, оскільки видалення таких об'єктів з образу може порушити поведінку програми.

Специфікація мови дозволяє скорочувати час життя об'єкта в локальних змінних, проте явно забороняє таку оптимізацію для посилань у купі.

Таке обмеження дозволяє реалізувати ідіому *finalizer guaridan*. Дана ідіома вирішує проблему фіналізації суперкласу у разі, коли метод finalize перевизначено у підкласі. Розглянемо таку ситуацію. Клас Base визначає метод *finalize* (рисунок 2.2).

finalize.

```
class Base {
  protected void finalize() {
    /* Finalize Base class instance */
  }
}
```

Рис. 2.2. Визначення методу *finalize*

Клас *Derived* успадковується від класу *Base* та перевизначає метод *finalize* (рисунок 2.3).

```
class Derived extends Base {
  protected void finalize() {
    /* Finalize Derived class instance */
  }
}
```

Рис. 2.3. Визначення класу *Derive*

При знищенні екземпляра класу *Derived* буде викликано фіналізатора, оголошений у класі *Derived*. Якщо реалізація фіналізатора *Derived* не визиває фіналізатора класу *Base* явним чином, фіналізатор класу *Base* не буде викликано.

Ідіома *finalizer guaridan* вирішує цю проблему в такий спосіб, який представлено на рисунку 2.4.

```
class Base {
    private final Object finalizerGuardian = new Object() {
        protected void finalize() {
            /* Finalize Base class instance */
        }
    }
}
```

Рис. 2.4. Визначення класу *Base*

При знищенні об’єкта, що реалізує клас *Base* або його підклас, об’єкт *finalizerGuardian* стає недосяжним і також видаляється збирачем сміття. При його знищенні викликається фіналізатор, який здійснює фіналізацію класу *Base*. Фіналізатор об’єкту *finalizerGuardian* викликається, навіть якщо підклас класу *Base* перевизначає метод *finalize* і не викликає фіналізатор суперкласу явно.

Деякі з об'єктів, створених під час ініціалізації класів, залишаться досяжними для додатка і можуть бути використані для непрямих викликів. Для аналізу досяжності методів необхідно проаналізувати ефекти ініціалізації класів та визначити, які об’єкти залишаться досяжними для програми. У простому випадку можна вважати, що це всі об’єкти, переживші складання сміття після ініціалізації класів. Однак подальше видалення невикористовуваних полів може зробити деякі з цих об’єктів недосяжними.

Розглянемо наступний приклад, який пердставлено на рисунку 2.5. На етапі ініціалізації класів ініціалізується клас *Main*. Ініціалізатор класу *Main* створює екземпляр класу A. Після складання сміття клас A буде додано до множини *InstantiableClasses*, метод *A.foo()* буде включений у замикання. Однак поле *Main.a*, через яке екземпляр класу A був досяжним, що не використовується досяжним кодом. Після видалення цього поля екземпляр класу A, на який більше немає посилань, може бути знищений збирачем сміття. Оскільки досяжний код не створює екземплярів цього класу, об'єкти цього типу не можуть бути аргументами непрямих викликів. Метод *A.foo()* не може бути викликаним.

```
@InitAtBuild
class Main {
  static A a = new A();
  static B b = new B();
  static Main main = new Main();

  public static void main(String ... args) {
    main.foo();
  }

  void foo() {
    System.out.println("Main.foo");
  }
}

class A extends Main {
  void foo() {
    System.out.println("A.foo");
  }
}
```

Рис. 2.5. Визначення класу *Main*

Проблему подібних залежностей можна вирішити, повторюючи пошук недосяжних методів після видалення полів, що не використовуються, поки на черговій ітерації не буде видалено жодного методу. У гіршому випадку на кожній ітерації з множини *InstantiableClasses* буде видалятися один клас. Загальна кількість ітерацій такого алгоритму обмежується кількістю аналізованих класів.

Однак ітеративний підхід не вирішує проблеми циклічних залежностей. Така залежність показана на прикладі класу B та поля *Main.b*. Поле *Main.b* не буде видалено, тому що до нього існує звернення з методу *B.foo()*. Віртуальний метод *B.foo()*, у свою чергу, включений у замикання тільки тому, що об’єкт класу A можна досягти через поле Main.b.

Використання збирача сміття для обчислення множини об'єктів дотичних для додатку неможливо, тому що збирач сміття проходить по всіх полях, не враховуючи, чи використовуються вони досяжним кодом. Для

точного аналізу необхідно побудувати множину полів, що читаються і множину об’єктів, досяжних додатку.

Зчитуваними будемо називати поля, для яких у коді досяжних методів існують інструкції читання (*getstatic, getfield*). Щоб обчислити множину об'єктів, досяжних для додатка, обійдемо граф об'єктів починаючи з кореневих посилань. При обході спеціальні посилання обробляються нарівні із звичайними посиланнями. Не відвідуватимемо об'єкти за посиланнями в нечитаних полях. Всі відвідувані об'єкти вважатимемо досяжними для програми.

Крім того, всі об'єкти, що фіналізуються, досяжні для програми через це посилання при виконанні методу *finalize*. Рисунок 2.6 ілюструє аналіз досяжності методів, що використовує описаний аналіз досяжності об'єктів, на наведеному вище прикладі.

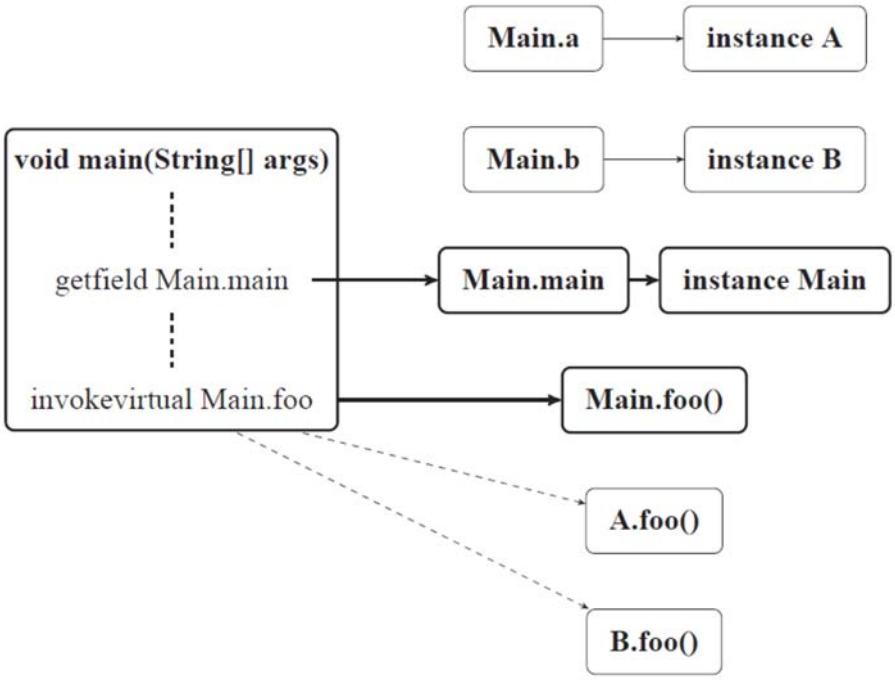


Рис. 2.6. Аналіз досяжності методів

Жирним виділено методи, поля та об'єкти, досяжні для програми. Ребра, виділені жирним, позначають відношення досяжності між методами, полями та об’єктами. Пунктирні ребра позначають відношення досяжності між методами.

2.2. Метод опису додаткових залежностей

Для зниження надмірності необхідно аналізувати залежності між методами, полями, класами та об'єктами. У більшості випадків необхідні залежності можуть бути вилучені з байт-коду. Проте аналіз деяких залежностей із байт-коду або складний, або зовсім неможливий. Так наприклад, рефлексія дозволяє звертатися до методів, полів і класів, використовуючи динамічний пошук на ім'я в процесі виконання. Залежності, що виникають при використанні рефлексії складно аналізувати статично. Інший приклад - Native-методи. Такі методи не мають Java байт-коду, їх реалізація написана іншою мовою, найчастіше C/C++. Код цих методів може звертатися до Java-методаів, полів та класів, використовуючи спеціальний інтерфейс, що надається віртуальною машиною. Прикладом такого інтерфейсу може бути JNI.

Для того щоб коректно обробляти такі ситуації, необхідно передбачити можливість вручну описувати додаткові залежності. Для коректної роботи алгоритму можна надати можливість безумовно заборонити видалення вибраних методів, полів та класів. Однак найчастіше метод, поле або клас не можна видаляти тому, що існують використовувані методи. Якщо ці методи виявляться недосяжними і будуть видалені, їх залежність також може бути видалена. Для запропонованих алгоритмів реалізовано механізм опису додаткових залежностей окремих методів. Для кожного методу можуть бути зазначені використовувані поля, методи та класи. Такі залежності аналізуються поряд із залежностями з байт-коду.

Додаткові залежності описуються за допомогою конфігураційних файлів. Такий запис перераховує додаткові залежності методу <ім'я методу>. Для методу можуть бути зазначені наступні залежності:

- *MethodCall* <ім'я методу> - методи, що викликаються з даного;
- *FieldRead* <ім'я поля> - поля, що читаються цим методом;
– *FieldWrite* <ім'я поля> – поля, які записуються даним методом;

– *ClassNew* <ім'я класу> – класи, екземпляри яких створюються в методі;

- *ClassAccess* <ім'я класу> - класи, на які даний метод посилається.

Для ручного опису залежностей надано можливість використовувати Java-анотації, які згодом конвертуються в конфігураційний файл. Приклад опису додаткових залежностей за допомогою анотацій наведено у лістингу на рисунку 2.7. Лістинг, представлений на рисунку 2.8 демонструє конфігураційний файл, згенерований з даних анотацій.

```
@Local(ClassAccess = {"some.class.A"})
Class getClassA() {
  return Class.forName("some.class.A");
}
@Local(ClassAccess = {"some.class.B"})
Class getClassA() {
  return Class.forName("some.class.B");
}
Class foo() {
  return getClassA();
}
```

Рис. 2.7. Приклад опису додаткових залежностей

```
Method getClassA
  ClassAccess some.class.A
EndMethod

Method getClassB
  ClassAccess some.class.B
EndMethod
```

Рис. 2.8. Залежності після конвертації в синтаксис конфігураційних файлів

При аналізі залежностей зазначеного методу, поля, методи та класи, перераховані у списку залежностей, будуть враховані поряд із залежностями, вилученими з байт-коду. Приклад аналізу методу *foo()* з приведений на рисунку 2.9. У цьому прикладі додаткові залежності методу *getClassB()* не використовуються, оскільки сам метод виявляється недосяжним.

Запропоновані алгоритми зниження надмірності не аналізують залежності, що виникають під час використання рефлексії, автоматично. Такі залежності повинні бути описані вручну за допомогою конфігураційних файлів або Java-анотацій. Жирним виділено досяжні методи та класи, ребра між методами та класами позначають залежність.

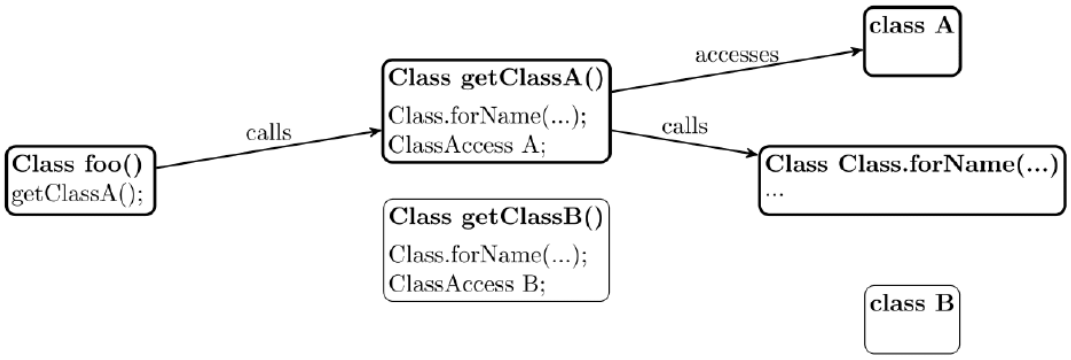


Рис. 2.9. Приклад аналізу додаткових залежностей

2.3. Алгоритм аналізу досяжності методів

Опишемо алгоритм аналізу досяжності методів, що здійснює вибіркову ініціалізацію використовуваних класів. Для цього доповнимо алгоритм RTA вибірковою ініціалізацією класів. Вхідними даними алгоритму будуть:

- ієрархія завантажених класів, що включає в себе байт-код методів даних класів;

- набір точок входу програми;

- конфігураційні файли.

На виході алгоритм породжує преініціалізований стан пам'яті віртуальної машини та множини:

– *ReachableMethods* – множина досяжних методів та його робоче підмножина;

– *IndirectInvocations* – множина методів, що використовуються для непрямих викликів;

– *InstantiableClasses* – множина класів, екземпляри яких доступні для додатку;

- *InitializableClasses* - множина класів, потенційно ініціалізованих досяжним кодом;

– *ReadFields* – множина полів, що читаються.

У процесі роботи алгоритм підтримує робочі підмножини відповідних множин: *NewReachableMethods*, *NewIndirectInvocations*, *NewInstantiable Classes*, *NewInitializableClasses*. При додаванні нового елемента до множини, яка має робочу підмножину, будемо додавати елемент у множину і відповідну робочу підмножину.

На початку роботи алгоритму точки входу програми додаються в множині *ReachableMethods*. Поки що хоча б одна з множин *NewReachableMethods* і нові класи не порожні, виконуються такі дії.

1. Для кожного методу з множини *NewReachableMethods* аналізуємо його залежності. Спочатку переглядаємо код методу:

а) методи, що викликаються інструкціями прямого виклику, додаємо у множину *ReachableMethods*;

б) методи, що використовуються інструкціями, спрямовані на віртуальний і invokeinterface, запам'ятовуємо в множині IndirectInvocations;

в) класи, які використовуються для створення об'єктів за допомогою інструкції нової, запам'ятовується в множині *InstantiableClasses*;

г) класи, на які посилаються інструкції new, invokestatic, getstatic, putstatic, додаємо в множину *InitializableClasses*;

д) поля, що використовуються інструкціями getstatic, getfield, додаємо в множину *ReadFields*.

Потім обробляємо залежності методу, описані в конфігураційних файлах:

а) методи, зазначені в залежностях *MethodCall*, додаємо в множину *ReachableMethods*;

б) класи, зазначені у залежностях *ClassNew*, запам'ятовуємо у множині *InstantiableClasses;*

в) класи, зазначені у залежностях *ClassNew* та *ClassAccess*, додаємо в множину *InitializableClasses*;

г) поля, зазначені в залежностях *FieldRead*, додаємо в множину *ReadFields*. Проаналізовані методи видаляємо з множини *NewReachable*.

2. Обробляємо класи з множини нових інітіалізованих класів. Оброблені класи видаляємо з множини *NewInitializableClasses*.

3. Виконуємо складання сміття, фіналізуємо віддалені об'єкти.

4. Обходимо об'єкти, досяжні програми. Для цього:

а) обходимо всі об'єкти, що фіналізуються;

б) обходимо граф об'єктів, досяжних з кореневих посилань за звичайним та спеціальним посиланням. За посиланням, що міститься у поле, переходимо, тільки якщо поле належить множині *ReadFields*. Класи відвідуваних об'єктів додаємо до множини *InstantiableClasses*.

5. У множину *ReachableMethods* додаємо методи, доступні через виклики:

а) додаємо методи класів з множини *InstantiableClasses*, досяжні через непрямі виклики методів з множини *NewIndirectInvocations*;

б) додаємо методи класів з множини *NewInstantiableClasses*, досяжні через непрямі виклики методів з множини.

6. До множини *ReachableMethods* додаємо фіналізатори.

7. Обнулюємо множину *NewInstantiableClasses*, *NewIndirect Invocations*.

Усі множини, що обчислюються алгоритмом, ростуть монотонно. Кількість елементів у множинах *ReachableMethods* і *InitializableClasses* обмежена загальною кількістю аналізованих методів та класів відповідно. Таким чином гарантується завершення ітеративного алгоритму. Після завершення алгоритму методи, що не належать до об’єднання множин

IndirectInvocations і *ReachableMethods*, можна видалити, не порушивши поведінки програми. Методи, що належать різниці множин у прямих обговореннях і *Reachablemethods* є ефективно абстрактними. Тіла таких методів можна видалити.

2.4. Алгоритм можливостей видалення полів

Після видалення недосяжних методів видаляються поля, що не використовуються. Поля, які можна видалити, не порушивши поведінки програми, назвемо видаленими. Визначимо такі критерії невидаленості поля.

– Досяжний код має операції читання поля. Зауважимо, що операції запису є достатнім критерієм невдалості. Нерідко поля, ініціалізовані в конструкторі або статичному ініціалізаторі, не використовуються після ініціалізації.

– Існує доступ до поля, який може призвести до ініціалізації класу.

– Поле не видаляється, якщо екземпляр поля може містити посилання, яке перешкоджає знищенню невдалих об’єктів. Видалення поля може порушити досяжність об'єктів, які можна досягти через посилання в полі.

Опишемо алгоритм обчислення множини невидаляємих полів згідно з даними критеріями. Вхідними даними алгоритму є:

– ієрархія завантажених класів, що включає опис полів і байт-код методів даних класів;

- множина *ReadFields*;

- конфігураційні файли;

- преініціалізований стан пам'яті віртуальної машини.

На виході алгоритм породжує множину UnremovableFields, яке містить всі поля, видалення яких може порушити видиму поведінку програми.

1. Ініціалізуємо множину *UnremovableFields* полями з множини ReadFields, оскільки всі поля, що читаються невдалими.

2. Додамо поля, доступ до яких може призвести до ініціалізації класу. У Java байт-коді існують дві інструкції доступу до полів, які мають побічний ефект ініціалізації класу: *getstatic* і *putstatic*. Усі поля, що використовуються інструкціями getstatic, були включені в множину *UnremovableFields* як зчитувані поля. Залишається проаналізувати інструкції *putstatic* у коді досяжних методів.

У деяких випадках можна гарантувати, що клас, що містить поле, буде ініціалізовано на момент виконання інструкції. Наприклад:

- якщо клас, що містить поле, ініціалізований на момент аналізу (тобто був ініціалізований у процесі побудови замикання);

– якщо клас, що містить аналізований метод, є підкласом класу, що містить поле.

В іншому випадку включимо поле, яке використовується інструкцією *putstatic*, в множину *UnremovableFields*.

3. Додамо поля, які можуть містити посилання, що перешкоджають знищеню неудаляємих об'єктів. Вважатимемо, що об'єктне поле може містити ненульове посилання в процесі виконання, якщо поле було ініціалізовано ненульовим посиланням у процесі аналізу, або у досяжних методах існують операції запису поля. Обчислимо множини *WrittenObjectFields* – множини об'єктних полів, для яких існують операції запису в досяжних методах.

– Проаналізуємо байт-код доступних методів. Додамо об'єктні поля, що використовуються інструкціями *putstatic*, *putfield*.

– Проаналізуємо залежності доступних методів. Додамо об'єктні поля, зазначені в якості *FieldWrite* залежностей.

У простому випадку можна консервативно припустити, що будь-яке поле, яке може містити ненульове посилання, перешкоджає зищенню невидаляємих об’єктів. Точніший алгоритм, що обчислює множину полів, які можуть містити посилання, що перешкоджають знищенню невидаляємих об’єктів.

Вважатимемо, що поле не можна видаляти, якщо через посилання в поле може досяжний невидаляємий об’єкт. Раніше ми виділили два типи невидаляємих об’єктів: об'єкти, що фіналізуються, та об'єкти, доступні за допомогою спеціальних посилань. Для перевірки досяжності фінальних об’єктів застосуємо аналіз типів.

– Статичне поле може посилатися на екземпляр класу A, якщо виконано хоча б одна з двох умов:

– у процесі ініціалізації полю було присвоєно посилання на екземпляр класу A;

– у коді досяжних методів є операції запису на полі, і клас A належить множині підкласів оголошеного типу поля.

– Нестатичне поле може посилатися на екземпляр класу A, якщо виконано хоча б одна з двох умов:

– у процесі ініціалізації екземпляру поля було присвоєно посилання на екземпляр класу A;

– у коді доступних методів є операції запису в полі, клас A належить множині підкласів оголошеного типу поля, і клас, що містить поле, належить множині.

– Примірник класу A може посилатися на екземпляр класу B, якщо клас B належить множині *InstantiableClasses*, і у класу A існує нестатичне поле, яке може посилатися на екземпляр класу B.

Складніше довести недосяжність об'єктів, доступних за допомогою спеціальних посилань. У Java-коді спеціальні посилання представлені екземплярамами класів *java.lang.WeakReference*, *java.lang.SoftReference*, *java.lang.PhantomReference*. У вихідному коді тип спеціального посилання параметризований класом об'єкта, який вказує посилання. При компіляції Java байт-код ця інформація втрачається через стирання типів. У простому випадку можна вважати, що якщо додаток створює екземпляри спеціальних класів посилань, то вони можуть посилатися на об'єкти будь-яких класів, що ініціалізуються.

Опишемо алгоритм, що обчислює *RefToFinalizableFields* - множина полів, які можуть містити посилання, що перешкоджають знищенню невидаляємих об’єктів. Усі поля з цієї множини є невидаляємs і повинні бути включені в множину *UnremovableFields* після завершення алгоритму. Вхідними даними алгоритму є:

– ієрархія завантажених класів, що включає опис полів і байт-код методів даних класів;

- Множини *InstantiableClasses*, *ReadFields*, *WrittenObjectFields*;

- Переініціалізований стан пам'яті віртуальної машини.

Вважатимемо, що додаток використовує спеціальні посилання, якщо множина InstantiableClasses містить якийсь із класів спеціальних посилань. Якщо програма використовує спеціальні посилання, то множина *RefToFinalizableFields* має бути доповнена об’єктними полями, які можуть містити ненульові посилання у процесі виконання. Для цього включимо у множину *RefToFinalizableFields* поля, що містять ненульові посилання, і поля з множини *WrittenObjectFields*.

Якщо програма не використовує спеціальні посилання, побудуємо множину полів, через посилання в яких можуть бути досяжні об'єкти, що фіналізуються.

Для кожного класу попередньо обчислимо дві множини.

- *MayReferTo(class)* - множина полів, оголошений тип яких сумісний із класом class. Тобто множина полів, які можуть містити посилання на екземпляри класу class. Для обчислення переберемо всі поля, кожне поле додамо в множину *MayReferTo(class)* підкласів нового типу поля.

– *ReferTo(class)* – множина полів, які містять посилання на екземпляри класу class. Для обчислення обійдемо граф об’єктів, які можна досягти із кореневих посилань. При відвідуванні посилання до поля додамо це поле до множини *ReferTo(class)*, де class – клас об'єкта, який вказує посилання.

У процесі роботи алгоритму будемо обчислювати такі множини:

- *RefToFinalizableFields, NewRefToFinalizableFields* - множина полів, які можуть прямо або опосередковано посилатися на фіналізовані класи, та його робоча підмножина;

- *RefToFinalizableClasses, NewRefToFinalizableClasses* - множина класів, екземпляри яких можуть прямо чи опосередковано посилатися на фіналізовані класи, та її робоча підмножина.

При додаванні нового елемента до множини *RefToFinalizableFields* або *RefToFinalizableClasses* будемо додавати елемент у множину і відповідну робочу підмножину.

Спочатку ініціалізуємо множину *RefToFinalizableClasses*, у яких визначено метод *finalize*. Поки множина *NewRefToFinalizableClasses* не порожня, виконуємо такі дії.

1. Для кожного класу з множини *NewRefToFinalizableClasses* поля, які можуть посилатися на екземпляр класу, додаються в *RefToFinalizableFields*. Це поля, що належать множині

$$MayReferTo(class) \cap (ReferTo(class) \cup WrittenObjectFields.$$

Оброблені класи видаляються з множини NewRefToFinalizableClasses.

2. Для кожного нестатичного поля з множини *NewRefToFinalizableFields* класи, екземпляри яких можуть містити поле, додаються до множини *RefTofinalableclasses*. Це підкласи класу, в якому визначено поле, що належать множині *InstantiableClasses.*

Опрацьовані поля видаляються з множини *NewRefToFinalizableFields.* Рисунок 2.10 показує приклад аналізу типів.

Усі множини, що обчислюються алгоритмом, ростуть монотонно. Кількість елементів у множині *RefToFinalizableClasses* обмежено загальною кількістю аналізованих класів. Таким чином гарантується завершення ітеративного алгоритму.

Ребро між полем та класом означає, що поле може посилатися наекземпляр класу. Класи, виділені жирним, належать множині

RefToFinalizableClasses. Поля, виділені жирним, належать множині *RefToFinalizableFields*, і не можуть бути видалені.

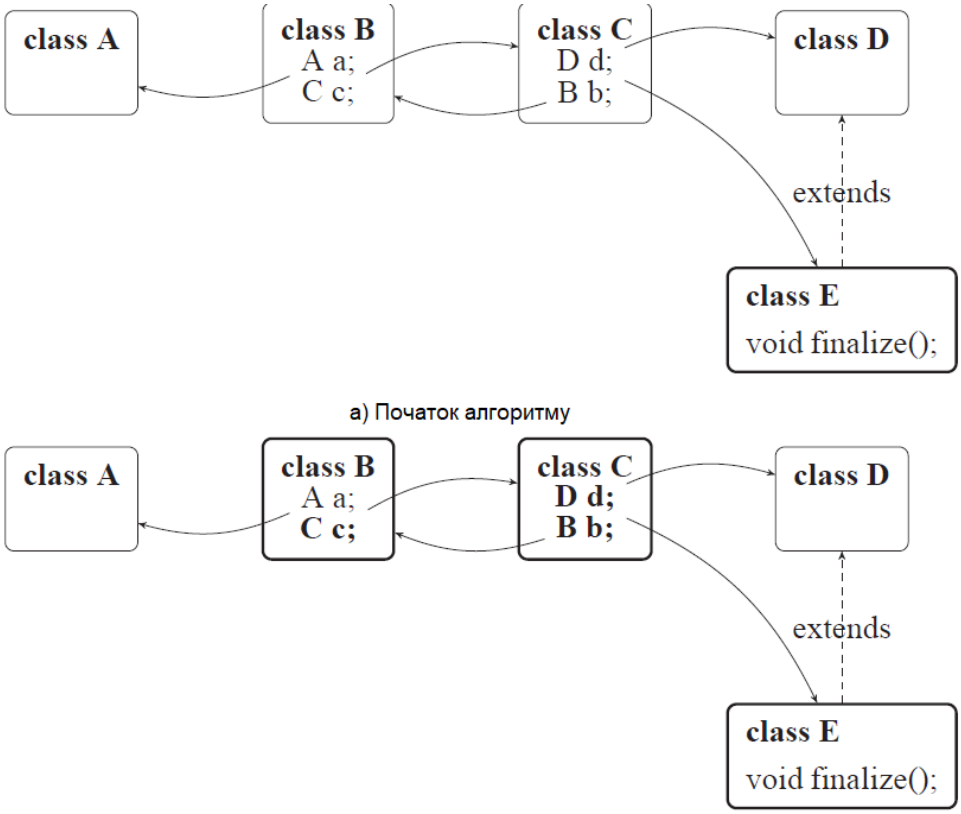


Рис. 2.10. Приклад аналізу досяжності невидаляємих об’єктів

2.5. Метод побудови словника послідовностей

Для створення нових інструкцій будемо використовувати шаблони, покриваючі послідовності інструкцій і задовольняють наступним обмеженням:

1. Довжина послідовності не перевищує $max\_pattern\_length$.

2. Послідовність не перетинає межі розширених базових блоків з однією точкою входу та множинними виходами. Іншими словами, в коді немає переходів всередину послідовності.

Для початку побудуємо словник всіх послідовностей вихідної програми, які відповідають цим обмеженням. Для кожної послідовності

зберігатимемо число її входжень у коді програми. Реалізуємо відповідний асоціативний масив за допомогою префіксного дерева. Ребра даного дерева позначаються інструкціями. Кожна вершина такого дерева описує послідовність інструкцій, яку можна відновити, виписавши інструкції всіх ребер на шляху від кореня до цієї вершини. У кожній вершині зберігається кількість входжень відповідної послідовності. Корінь дерева відповідно дорівнює порожній послідовності. Таку послідовність будемо позначати ε Приклад префіксного дерева словника послідовностей показаний на рисунку 2.11.

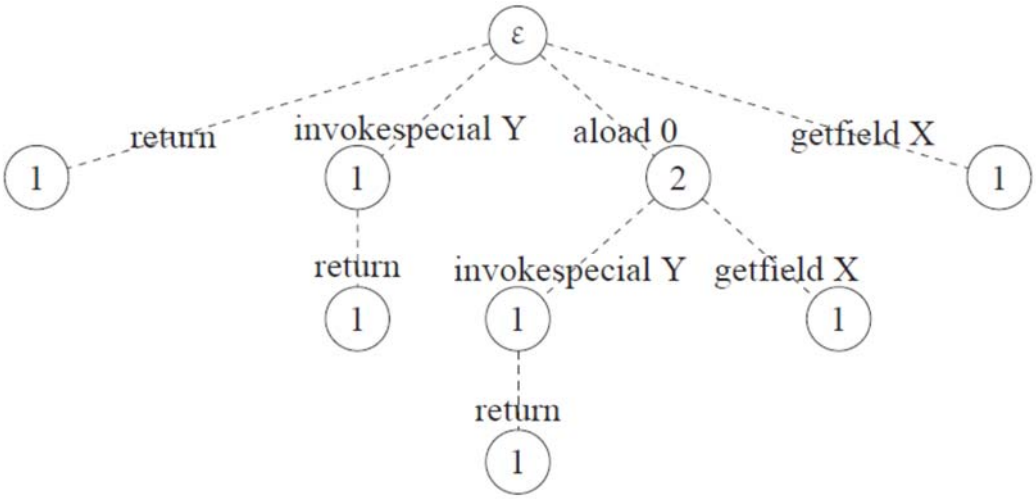


Рис. 2.11. Приклад префіксного дерева словника послідовностей для послідовностей $aload\ 0, getfield\ X, aload\ 0\ invokespecialY\ \ return$

Опишемо алгоритм побудови словника послідовностей. Розіб'ємо тіло програми на розширені базові блоки та аналізуватимемо код кожного із блоків окремо. Позначимо інструкцію у позиції блоку $instr(i)$. Для кожної позиції всередині блоку обчислимо множину $Sequences\ (i)$ – множина послідовностей, що закінчуються в і-ій позиції. Для цього використовуємо динамічне програмування. На початку базового блоку ця множина складається з одного елемента порожньої послідовності.

$$Sequences\ (0) = \{\varepsilon\} \tag{2.1}$$

Для кожної наступної позиції множина обчислюється на підставі множини попередньої позиції.

– У цій позиції може початися нова послідовність, тому в множину додається порожня послідовність - корінь дерева.

– Кожна послідовність, не довша за $max\_pattern\_length$, з множини попередньої позиції може бути продовжена поточною інструкцією, тому в множину додається розширена послідовність. Для кожної розширеної послідовності входження збільшується на одиницю.

$$Sequences\ (i+1) = \{\varepsilon\} \cup \bigcup_{\substack{s \in Sequences(i) \\ length(s) < max\_pattern\_length}} \{extend(s, instr(i+1))\} \qquad (2.2)$$

Операція $xtend(s, i)$ продовжує послідовність, задану вершиною префіксного дерева s, інструкцією i і повертає розширену послідовність. Якщо розширена послідовність раніше не зустрічалася, вона додається до дерева шляхом додавання відповідного ребра з вершини s. Верхня оцінка кількості послідовностей для програми завдовжки

$$length = \sum_{k=1}^{max\_pattern\_length} (length - k + 1) \qquad (2.3)$$

Множину шаблонів можна представити у вигляді префіксного дерева, аналогічно з того, як представлено множину послідовностей. Ми не будемо зберігати префіксне дерево шаблонів у явному вигляді. Натомість використовуємо префіксне дерево послідовностей для перебора множини шаблонів у тому порядку, якби ми обходили префіксне дерево шаблонів у глибину. Такий підхід дозволить суттєво скоротити обсяг пам'яті, необхідний для зберігання ня словника.

Перебір шаблонів будемо здійснювати рекурсивно, починаючи з порожнього шаблона. На кожному кроці обчислюватимемо множину усіх

можливих продовжень для поточного шаблону. Далі опишемо процес обчислення всіх можливих продовжень для заданого шаблону.

Для кожного шаблону обчислюватимемо множину $MatchingSequences(p)$. - множина всіх послідовностей, що покриваються шаблоном. Послідовності в цій множині представлені вершинами префіксного дерева. Множина $MatchingSequences$ для порожнього шаблону містить один елемент – порожню послідовність.

Множина вихідних ребер для даної вершини префіксного дерева визначимо *Edges(s)*. Ця множина визначає множину інструкцій, якими може бути продовжено відповідну послідовність. Наступна множина визначає множину інструкцій програми, якими може бути продовжено заданий шаблон p.

$$SIContinuations(p) = \{Edges(s) | s \in MatchingSequences(p) \quad (2.4)$$

За множиною $\boldsymbol{SIContinuations(p)}$ визначимо множину інструкцій шаблона, якими може бути продовжено шаблон.

$$\boldsymbol{PIContinuations(p) = \{PI(si) | si \in SIContinuations(p)\}}, \quad (2.5)$$

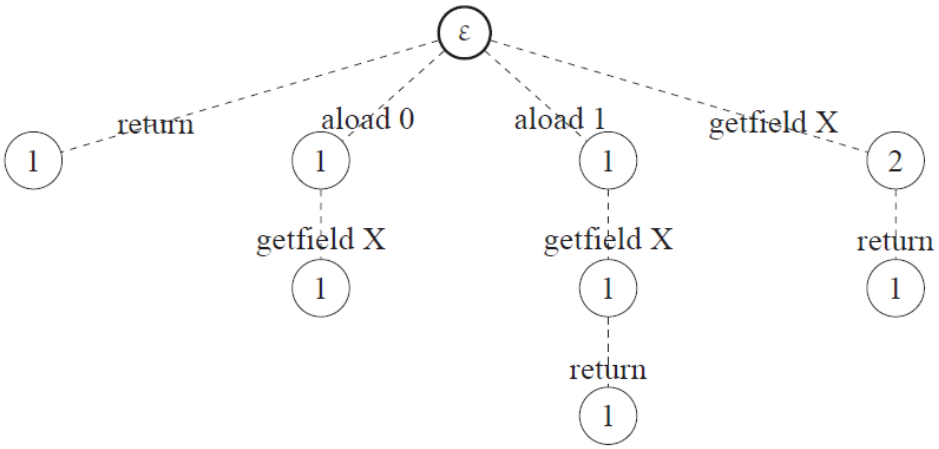
де $\boldsymbol{PI(si)}$– множина інструкцій шаблону, якими можна покрити інструкцію послідовності $\boldsymbol{si}$.

Множина $\boldsymbol{SIContinuations(p)}$ визначає всі можливі продовження шаблон $\boldsymbol{p}$. Тепер для кожного продовження необхідно обчислити множину послідовностей, що покриваються продовженням. Це множина вершин, у якій можна перейти з вершин поточної множини послідовностей по ребрах, що відповідають новій інструкції шаблону.

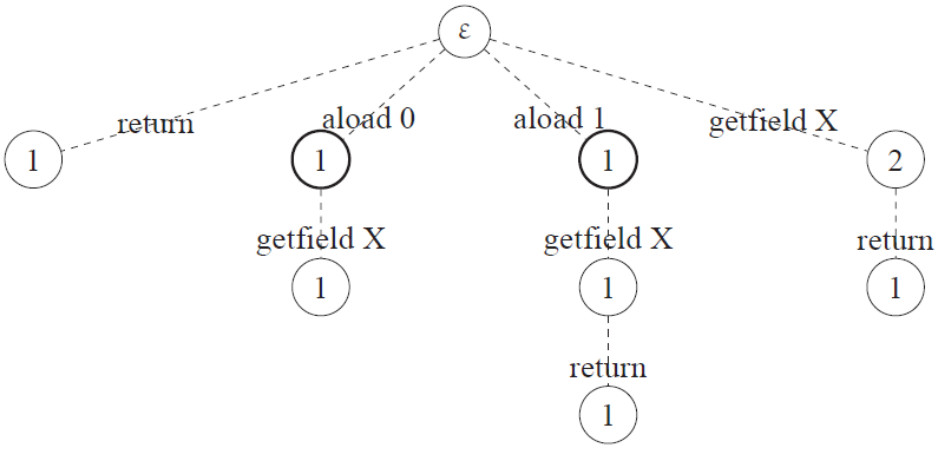
$$MatchingSequences(p, pi) = \bigcup_{s \in MatchingSequences(p)} \bigcup_{si \in SIContinuations(p)} \{extend(s; si) jmatches(si; pi)\}$$

де $matches\ (si, pi)$ - функція, яка визначає, чи відповідає інструкція послідовності si інструкції шаблону pi.

На рисунку 2.12 показано процес побудови шаблону $aload * getfield\ X$ для послідовностей $aload\ 1\ getfield\ X\ return$.



а) порожній шаблон



б) шаблон aload * getfield X, кількість входжень дорівнює 2

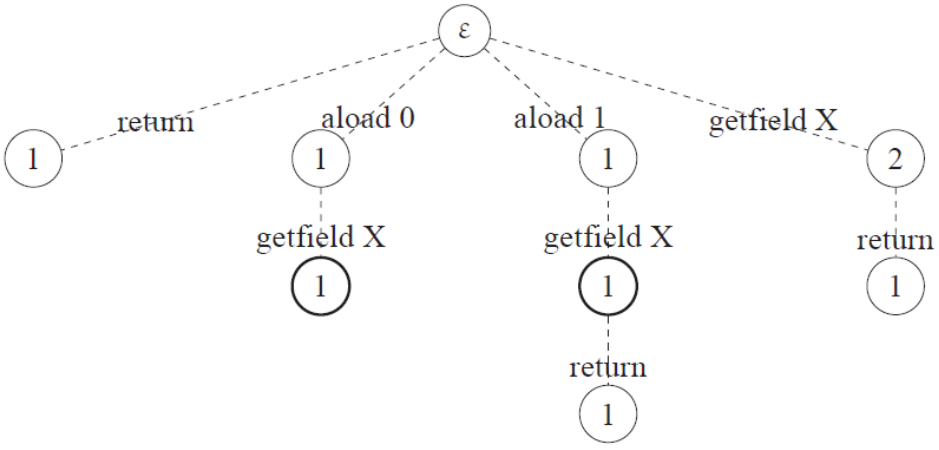


Рис. 2.12. Приклад префіксного дерева словника послідовностей

Не будемо розширювати шаблони інструкціями переходів із фіксованим аргументом, тобто не згортатимемо аргументи таких інструкцій. Це пов’язано з тим, що значення аргументу інструкції переходу є зміщенням у тіло методу і може змінюватися при введенні спеціалізованих інструкцій.

Укорочувати аргументи інструкцій переходів у своїй можна, оскільки значення зміщень що неспроможні збільшуватися.

Висновки до другого розділу

1. У цьому розділі було розглянуто завдання зниження надмірності Java-додатків під час використання ранньої ініціалізації класів.

2. Запропоновано алгоритм аналізу видаляємості полів, що дозволяє видаляти ініціалізовані, але не використовувані поля. Запропонований алгоритм зберігає семантику фіналізації та не видаляє поля, якщо це може порушити досяжність об’єктів, фіналізація яких необхідна додатку.

3. Розглянуто завдання скорочення розміру інтерпретації коду шляхом спеціалізації набору інструкцій. Цей підхід є розвитком широко використовуваної ідеї словникового стискування. Був запропонований алгоритм спеціалізації набору інструкцій Java байт-коду, що скорочує розмір коду заданого додатку та інтерпретатора, необхідного для його виконання.

РОЗДІЛ 3
ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗНИЖЕННЯ НАДЛИШКОВОСТІ ПРОГРАМНИХ КОДІВ JAVA-ДОДАТКІВ

3.1. Загальна архітектура системи

В рамках виконання магістерського дослідження було реалізовано інструмент для автоматичної спеціалізації Java-платформи Oracle Java ME Embedded для заданої програми у закритій моделі. Платформа Oracle Java ME Embedded доступна на апаратних платформах різних виробників. При реалізації цього інструменту були використані алгоритми, запропоновані у розділі 2. Процес спеціалізації платформи зображено, у вигляді спеціальної архітектурної схеми програмного забезпечення на рисунку 3.1.

Спеціалізація Java-коду платформи для заданої програми здійснюється в інструменті ромізації віртуальної машини CLDC Hotspot Implementation (Romizer). Romizer є спеціальною версією віртуальної машини, яка виконується на хостівому пристрої в процесі побудови Java-плафторми.

Вхідними даними інструмента є системні класи та класи програми. На виході інструмент генерує завантажувальний образ ініціалізованого стану віртуальної машини. У процесі підготовки образу завантажуються системні класи та класи програми, дозволяються символічні посилання, здійснюються оптимізації у пам'яті віртуальної машини, покликані скоротити розмір образу та прискорити виконання на цільовому пристрої. Стан пам'яті ініціалізованої віртуальної машини зберігається як вихідний файл мовою C++ ROMImage.cpp. Цей файл визначає статичні масиви, вміст яких визначає стан пам'яті віртуальної машини.

Для спеціалізації Java-коду платформи при підготовці образу в інструменті romizer були реалізовані алгоритми зниження надлишковості, представлені в розділі 2. Дані алгоритми були реалізовані як частина

оптимізації, здійснюваних після завантаження класів та дозволу символічних посилань.

Результатом зниження надлишковості за допомогою реалізованих алгоритмів являється:

– ініціалізація деяких класів, що використовуються додатком.

– видалення з пам'яті віртуальної машини методів, полів та класів, які не використовуються у завантаженому коді.

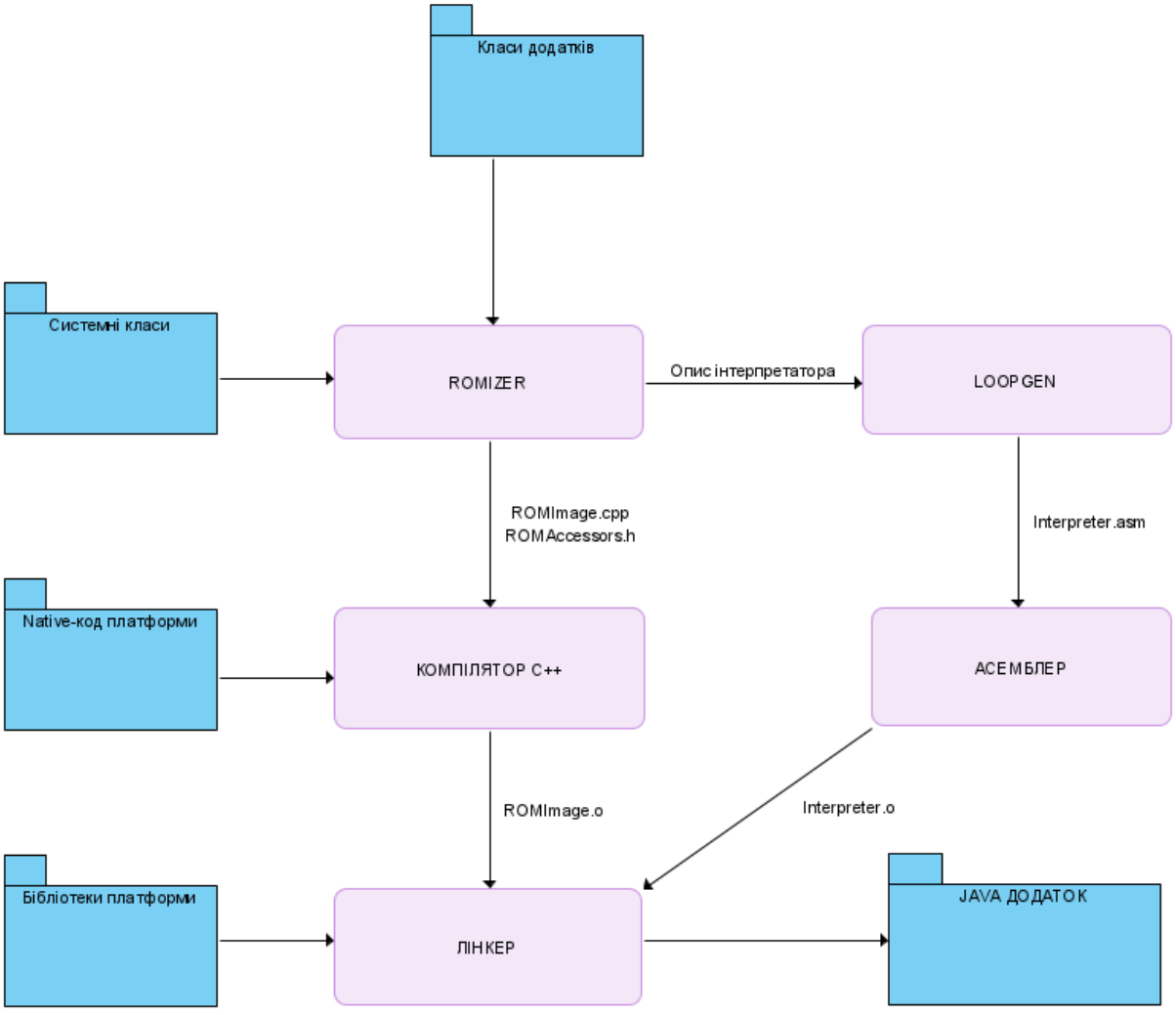


Рис. 3.1. Загальна архітектура системи зниження надлишковості програмних кодів Java-додатків

Після зниження надмірності romizer спеціалізує інтерпретатор для компактного кодування байт-коду методів, що залишилися. В результаті спеціалізації байт-код методів у пам'яті віртуальної машини переписується в

новий набір інструкцій. На виході romizer генерує файл з описом спеціалізованого інтерпретатора, який потім використовується для генерації асемблерного коду інтерпретатора за допомогою інструменту loopgen. Після цього код інтерпретатора компілюється асемблером цільової платформи. У поточній реалізації romizer використовує підмножину оптимізацій. Повністю запропонований алгоритм спеціалізації набору інструкцій було реалізовано у вигляді окремого інструменту. Для заданого набору клас-файлів інструмент породжує код у новому наборі інструкцій та опис інтерпретатора, необхідного для його виконання. Для вимірювання ефективності алгоритму спеціалізації набору інструкцій було використано цей інструмент.

Крім завантажувального образу romizer генерує файл ROMAccessors.h, що містить статичний інтерфейс для доступу до полів Java-об'єктів. Даний інтерфейс використовується для взаємодії з Java-об'єктами в native-коді віртуальної машини та системних класів.

Файли ROMImage.cpp, ROMAccessors.h компілюються разом із native-кодом платформи, а потім лінкуються разом з інтерпретатором і бібліотеками в один файл, що виконується. У процесі лінкування здійснюється спеціалізація native-коду платформи за рахунок видалення функцій, що не використовуються. Підсумковий файл, що виконується, містить додаток і реалізацію платформи, спеціалізовану для заданої програми.

Системні класи віртуальної машини, на яку реалізовані запропоновані алгоритми, містять велику кількість native-методів. Для коректної роботи алгоритмів зниження надмірності необхідно враховувати залежність таких методів. Непоодинокі ситуації, коли деякі поля використовуються тільки native-методами. Наприклад, поля, що зберігають дескриптори нативних ресурсів, які можуть не використовуватися в Java-коді. Видалення таких полів некоректно, якщо досягнемо хоча б один native-метод, який використовує поле.

У роботі запропоновано механізм для ручного опису таких залежностей. При великій кількості native-методів ручний опис залежностей

неефективний і загрожує помилками. Опишемо метод автоматичного аналізу залежностей native-методів.

Нативний інтерфейс віртуальної машини, для якої реалізовано описані алгоритми, що не дозволяє викликати Java-методи з native-методів, тому подальший опис стосується лише аналізу використання полів.

Аналогічний підхід можна використовувати для аналізу досяжності методів. Для кожного native методу необхідно знати, які Java поля використовуються реалізацією цього методу. Для доступу до Java-сутностей інтерфейс JNI використовує динамічний пошук на ім'я, що робить статичний аналіз його використання важким. Однак для заданого набору класів можна згенерувати статичний нативний інтерфейс, який моделюватиме Java-сутності з допомогою сутностей C++. Аналізувати використання такого інтерфейсу значно простіше. Так, ієрархію Java-класів можна продублювати ієрархією C++ класів, згенерувавши для кожного Java-класу відповідний йому клас C++. Для кожного Java-поля генерується accessor-метод у відповідному класі C++. Accessor-метод повертає посилання на поле і використовується як для запису, так і та для читання Java-поля. За наявності можливості виклику Java-методів з нативного коду, Java-методи можуть бути змодельовані аналогічним чином за допомогою C++ методів.

У лістингу на рисунку 3.2 наведено фрагмент інтерфейсу, згенерованого для класу java.lang.String, та приклад його використання.

При використанні такого статичного інтерфейсу аналіз Java-залежностей native-методів зводиться до аналізу досяжності C++ методів.

Для кожного C++ методу, що є реалізацією native-методу, необхідно обчислити набір досяжних методів-аксесорів. Залежностями даного методу є Java-поля, що відповідають знайденим аксесорам.

Аналізатор нативного коду реалізований у вигляді самостійної програми, яка запускається в процесі складання середовища виконання та системних класів. Аналізатор приймає на вхід набір вихідних файлів C++ разом з флажками компіляції. На виході аналізатор генерує файл

конфігурації з описами залежностей. Для розбору C++ коду використовується бібліотека libclang, яка надає доступ до AST вихідного коду. Для кожного методу, що є реалізацією native-методу, за допомогою обходу в глибину обчислюється множина досяжних із нього методів. Якщо множина методів-аксесорів не порожня, то конфігураційний файл додається запис, оголошує Java-поля, відповідні досяжним аксесорам, залежностями методу, що аналізується.

```
// Java
package java.lang;
class String {
  char value[];
  int offset;
  int count;
  ...
}

// C++
struct jchar_array {
  jint length();
  jchar* elements();
};

struct Java_java_lang_String :
  public Java_java_lang_Object {
  jchar_array*& value();
  jint& offset();
  jint& count();
};

Java_java_lang_String* java_string;
char* char_base =
  java_string->value()->elements();
char_base += java_string->offset();
int length = java_string->count();
std::string str(char_base, length);
```

Рис. 3.2. Фрагмент інтерфейсу, згенерованого для класу java.lang.String

Для аналізу віртуальних викликів використовується алгоритм CHA, тобто для даного віртуального виклику досяжними є всі можливі реалізації віртуального методу. Аналіз непрямих викликів через вказівники на функції не підтримується. Якщо при обчисленні множини методів, що досягаються,

зустрівся виклик через покажчик на функцію, генерується попередження про те, що залежності аналізованого native-методу повинні бути описані вручну.

3.2. Програмна реалізація та опис інструкції користувача

В ході виконання магістерської роботи реалізовано додаток для зниження надлишковості програмних кодів Java-додатків. На рисунку 3.3 представлено головне вікно розробленого додатку.

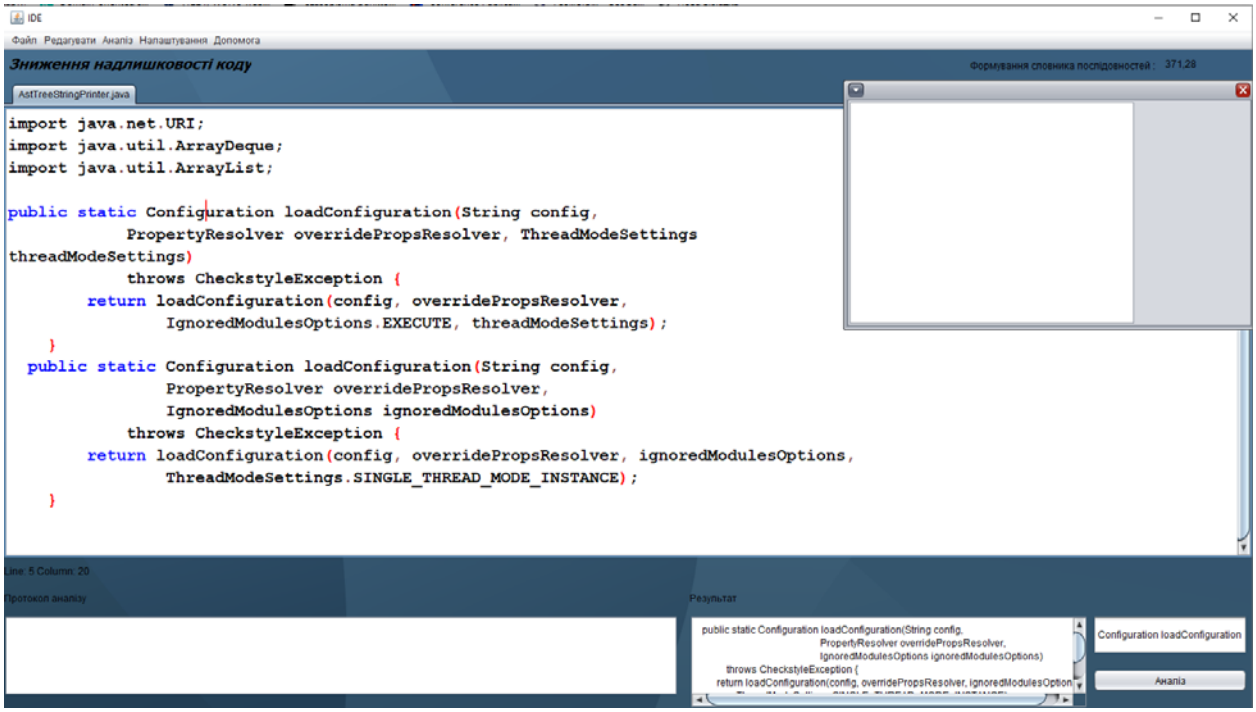


Рис. 3.3. Головне вікно додатку для зниження надлишковості програмних кодів Java-додатків

Як видно з рисунку 3.3, у користувача є можливість завантажити Java-проект для аналізу, налаштувати параметри аналізу, переглянути протокол та результат проведеного аналізу.

Для вибору алгоритму, що буде використовуватися для аналізу, можна скористатися функціоналом, який представлено на рисунку 3.4.

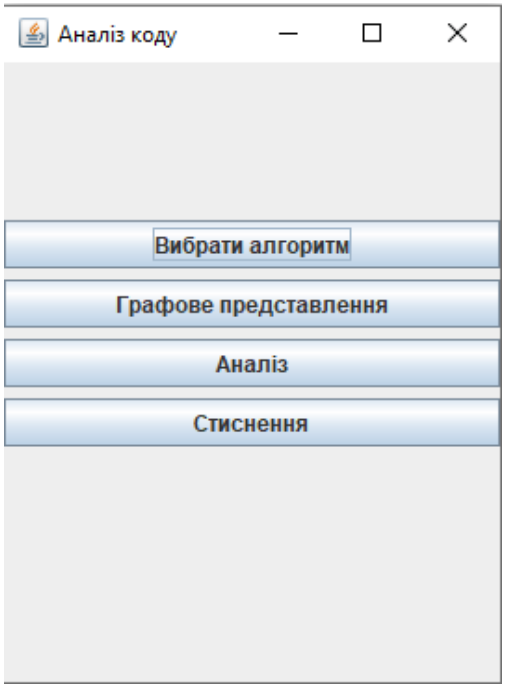


Рис. 3.4. Вікно додатку для вибору алгоритму аналізу надлишковості програмного коду

Після вибору алгоритму в користувача зявляється можливість проведення аналізу програмного коду (рисунок 3.5) та здійснення процедури відповідної оптимізації (рисунок 3.6)

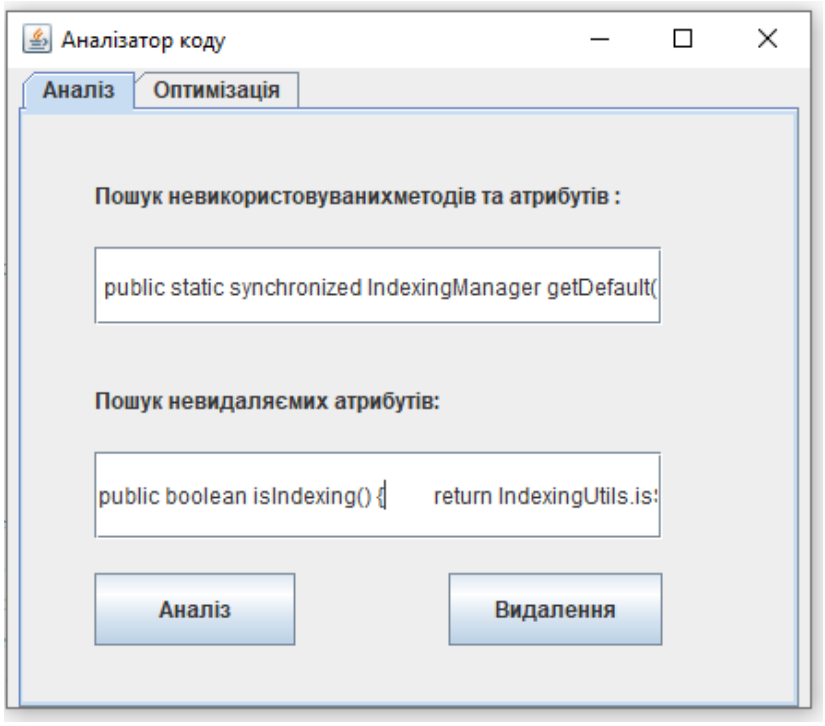


Рис. 3.5. Вікно додатку для аналізу програмного коду

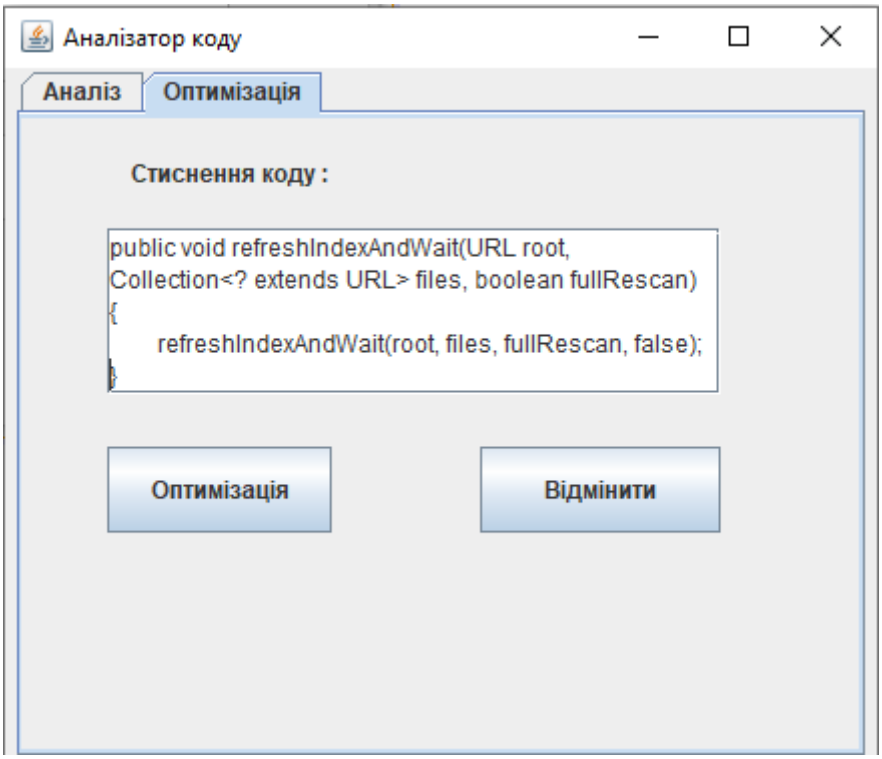


Рис. 3.6. Вікно додатку для оптимізаці програмного коду

На рисунку 3.7 представлено форму із основними налаштуваннями, які необхідно вибрати в процесі аналізу надлишковості програмних кодів Java-додатків.

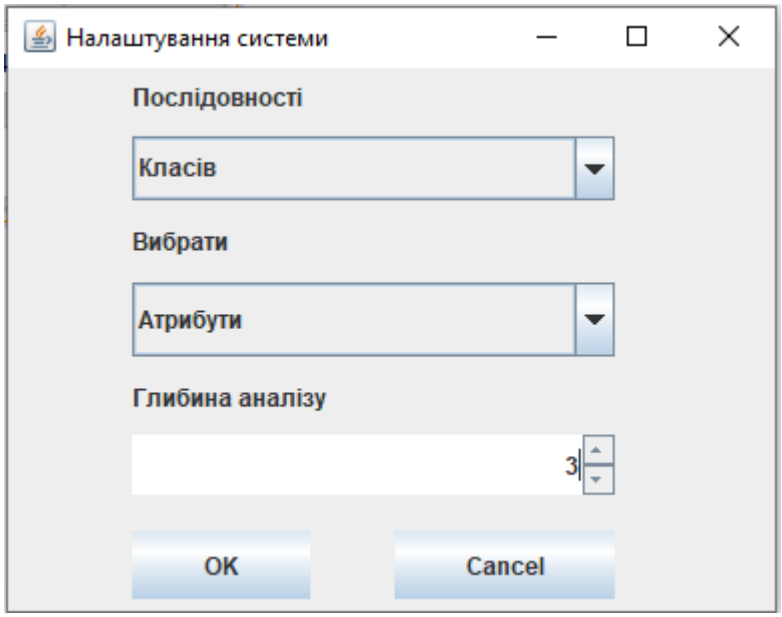


Рис. 3.7. Вікно з налаштуваннями для аналізу програмного коду

Після проведеного аналізу в користувача є можливість зберегти отримані результати та відправити їх для повтрної компіляції у відповідне середовище розробки.

3.3. Експериментальні дослідження алгоритмів зниження надлишковості програмних кодів Java-додатків

Для оцінки ефективності реалізованих алгоритмів зниження надмірності та порівняння їх з існуючими рішеннями було проведено ряд чисельних експериментів. Сформулюємо завдання експериментів.

1. Визначити скорочення розміру, що забезпечується реалізованими алгоритмами.

2. Визначити, як використання більш точного аналізу впливає на розмір образу. Реалізований алгоритм аналізу досяжності методів є адаптацією алгоритму RTA аналізу ініціалізованого стану віртуальної машини. Альтернативний підхід реалізовано у віртуальній машині JITS, де використовується більш консервативний аналіз досяжності методів CHA. Цей аналіз залежить від ініціалізованого стану віртуальної машини.

3. Продемонструвати збереження поведінки під час використання вибіркової ініціалізації. Визначити, наскільки вибіркова ініціалізація класів скорочує розмір образу порівняно з безумовною ініціалізацією. Реалізований алгоритм аналізу досяжність методів вибірково ініціалізує класи, які можуть бути ініціалізовані в процесі роботи програми. Альтернативний підхід реалізований у віртуальній машині ExoVM, де всі завантажені класи ініціалізуються, безумовно, до аналізу досяжності. Безумовна ініціалізація класів може порушити поведінку програми та збільшити розмір образу.

4. Продемонструвати збереження семантики фіналізації під час використання реалізованого алгоритму. Визначити як видалення полів, що ініціалізуються, але не використовуються, впливає на розмір образу. Реалізований алгоритм аналізу віддаленості полів дозволяє видаляти поля,

що ініціалізуються, але не використовуються, зберігаючи при цьому семантику фіналізації об'єктів Більшість розглянутих систем зниження надмірності порушують семантику фіналізації, видаляючи об'єктні поля, що ініціалізуються, але не використовуються. У віртуальній машині JamaicaVM об'єктні поля, для яких є операції запису не видаляються.

5. Визначити, яку кількість native-методів проаналізовано автоматично, скільки залежностей згенеровано автоматично. Система зниження надмірності, реалізована в рамках цієї роботи автоматично аналізує залежність нативних методів. Жодна з розглянутих систем зниження надмірності не аналізує міжмовні залежності автоматично.

Зазначимо, що у всіх розглянутих системах зниження надмірності видалення методів, полів і класів, що не використовуються, використовується спільно з іншими оптимізаціями. Ці оптимізації можуть впливати на підсумковий розмір програми. Крім того, у різних системах використовуються різні конфігурації та реалізації стандартних бібліотек. Таким чином, навіть для однієї і тієї ж вихідної програми, набір аналізованих класів може істотно відрізнятися. Дані фактори не дозволяють безпосередньо порівнювати розміри програм, згенерованих різними системами. Поширеною практикою є реалізація порівнюваних алгоритмів в одній віртуальній машині та порівняння розмірів додатків, згенерованих при застосуванні різних алгоритмів. Для вирішення завдань експериментів у віртуальній машині CLDC HI були реалізовані модифікації запропонованого алгоритму.

Для вимірювань було використано 8 додатків для наступних конфігурацій:

CLDC – реалізація стандарту Connected Limited Device Configuration.

– MEEP – реалізація стандарту Java ME Embedded Profile.

Інформація про використані програми наведена в таблиці 3.1. Додаток 1 є прикладом мінімальної програми для платформи CLDC, а додатки 2-7 використовуються у наборі тестів. Цей набір тестів часто використовується для оцінки продуктивності реалізації Java-платформи для пристроїв, що

вбудовуються. Зокрема, дані додатки були використані для аналізу ефективності алгоритмів зниження надмірності віртуальної машини ExoVM. Додаток 8 є прикладом програми для конфігурації з великою кількістю класів.

Таблиця 3.1

Аналізовані програмні додатки

№	Додаток	Конфігурація	Опис
1	Hello World	CLDC	додаток, що друкує ”Hello World”
2	Chess	CLDC	гра шахи
3	Crypto	CLDC	криптографічний пакет
4	Parallel	CLDC	тест на багатопоточне виконання
5	kXML	CLDC	парсер
6	PNG	CLDC	PNG зображень
7	RegExp	CLDC	пакет обчислення регулярних виразів
8	AMS MEEP	MEEP	Application Management System

Експеримент 1. Визначити скорочення розміру, що забезпечується реалізованими алгоритмами. Для цього експерименту були виміряні розміри образів:

– none – без зниження надмірності;

– base – зі зниженням надмірності з допомогою реалізованих алгоритмів.

Інформація про розмір образів та відносне скорочення розміру наведено у таблиці 3.2 та рисунку 3.8. Застосування реалізованих алгоритмів зниження надлишковості в закритій моделі скорочує розмір образу на 57,62-95,18% (середнє значення 74,57%).

Таблиця 3.2

Експеримент 1. Розмір образу в байтах та відносне скорочення розміру

№	Додаток	none	base (vs none)
1	Hello World	255968	12340 (-95,18 %)
2	Chess	288516	76404 (-73,52 %)
3	Crypto	291476	86684 (-70,26 %)
4	Parallel	308284	84944 (-72,45 %)
5	kXML	311224	80472 (-74,14 %)
6	PNG	289196	63652 (-77,99 %)
7	RegExp	301652	74288 (-75,37 %)
8	AMS MEEP	1613388	683832 (-57,62 %)

Експеримент 2. Як використання більш точного аналізу впливає на розмір образу.

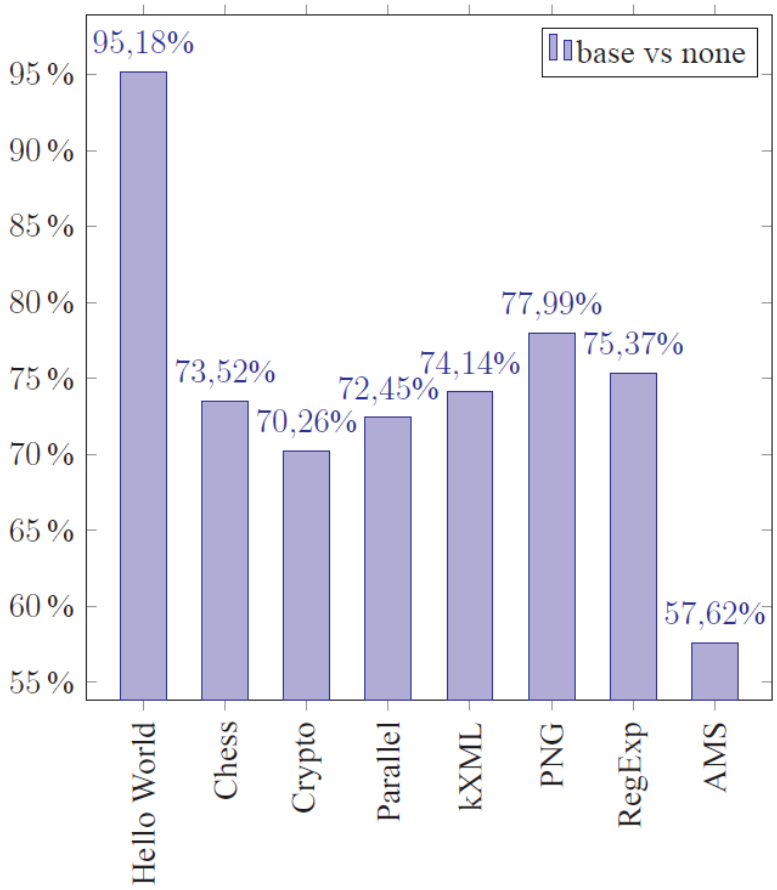


Рис. 3.8. Відносне скорочення розміру образу

Для цього експерименту була реалізована модифікація запропонованого алгоритм, який використовує аналіз CHA для дозволу непрямих викликів (cha). Виміряно розмір образів, отриманих при використанні вихідного та модифікованого алгоритмів. Інформація про розмір образів та відносне скорочення розміру наведена в таблиці 3.3 та рисунку 3.9. Застосування запропонованого алгоритму аналізу досяжності методів скорочує розмір образу на 29,57–50,86 % (середнє значення 40,88 %) проти алгоритмом CHA.

Таблиця 3.3

Експеримент 2. Розмір образу в байтах та відносне скорочення розміру

№	Додаток	cha	base (vs cha)
1	Hello World	19040	12340 (-35,19 %)
2	Chess	125052	76404 (-38,90 %)
3	Crypto	131556	86684 (-34,11 %)
4	Parallel	149896	84944 (-43,33 %)
5	kXML	152268	80472 (-47,15 %)
6	PNG	129532	63652 (-50,86 %)
7	RegExp	142732	74288 (-47,95 %)
8	AMS MEEP	970948	683832 (-29,57 %)

Експеримент 2. Як використання більш точного аналізу впливає на розмір образу.

Експеримент 3. Продемонструвати збереження поведінки під час використання вибіркової ініціалізації. Визначити, наскільки вибіркова ініціалізація класів скорочує розмір образу порівняно з безумовною ініціалізацією. Для цього експерименту була реалізована модифікація запропонованого алгоритму, який безумовно ініціалізує завантажені класи до аналізу досяжності (init-all).

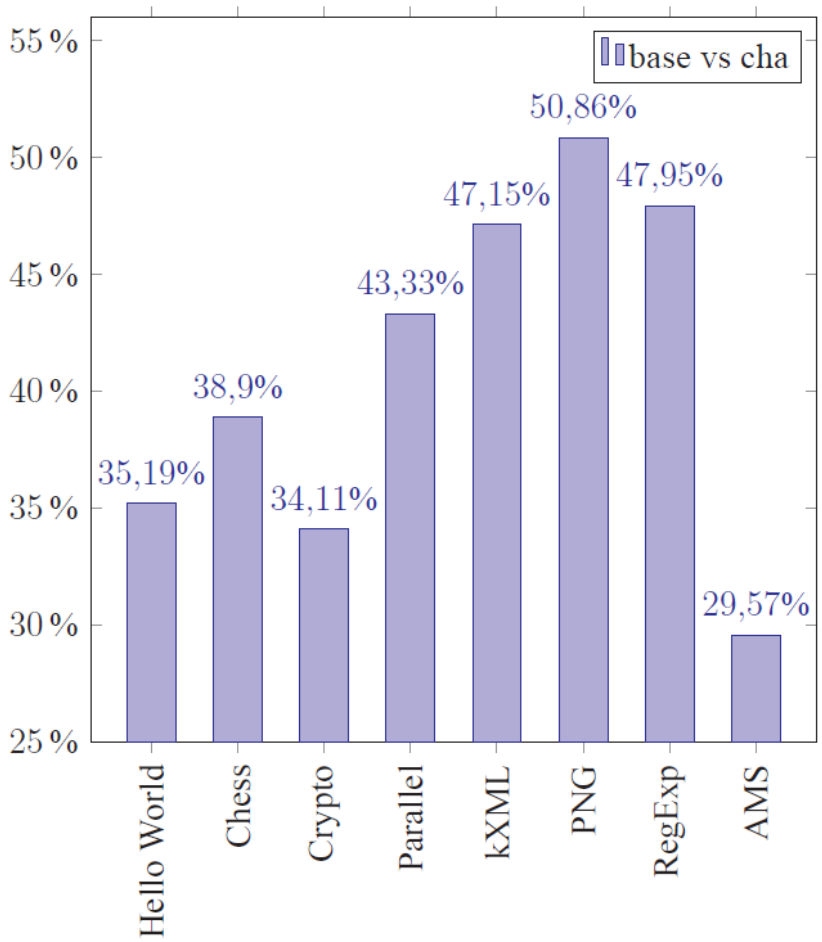


Рис. 3.9. Експеримент 2. Відносне скорочення розміру образу

Наступний тест демонструє порушення поведінки при безумовній ініціалізації класів.

```
class A {
  static {
    Test.a_is_initialized = true;
  }
}
class Test {
  public static boolean a_is_initialized;
  public static void main(String args[]) throws Exception {
    if (a_is_initialized)
      throw new Exception("A could not have been initialized");
  }
}
```

Ромізація з використанням модифікованого алгоритму призводить до ініціалізації класу A та кидання винятку під час виконання програми.

Ромізація з використанням запропонованого алгоритму не ініціалізує клас A виконання програми завершується коректно.

Для оцінки впливу вибіркової ініціалізації розмір образа був виміряно розмір образів, отриманих при використанні вихідного та модифікованих алгоритмів. Інформація про розмір образів та відносnt скорочення розміру наведено у таблиці 3.4 та рисунку 3.10. Застосування вибірковоної ініціалізації під час аналізу досяжності методів скорочує розмір образу на 0,3–11,71 % (середнє значення 4,27 %) проти безумовної ініціалізацією всіх класів.

Слід зазначити, що ініціалізатори класів нерідко використовують метод System.getPropery та аналогічні платформо-залежні методи, результат виконання яких при підготовці образу та на цільовому пристрої може відрізнятись. Рання ініціалізація таких класів також може призвести порушення поведінки програми. У наборі класів конфігурації CLDC дано ним властивістю має 1 клас, у конфігурації MEEP - 47 класів.

Експеримент 4. Продемонструвати збереження семантики фіналізації під час використання реалізованого алгоритму. Визначити, як видалення ініціалізованих полів, що не використовуються, впливає на розмір образу.

Для даного експерименту були реалізовані дві модифікації запропонованого алгоритму:

- dont-remove - найбільш консервативна модифікація, не видаляє ініціалізовані, але об'єктні поля, що не використовуються;

– dont-preserve – модифікація не зберігає семантику фіналізації та удаляє всі об'єктні поля, що ініціалізуються, але не використовуються.

Наступний тест демонструє порушення семантики фіналізації при видалення полів, що ініціалізуються, але не використовуються.

```
class A {}
class B {}
public class Test {
  static A removable = new A();
  static B unremovable = new B();
  static WeakReference<B> weak = new WeakReference<>(unremovable);
  public static void main(String args[]) throws Exception {
    if (weak.get() == null)
      throw new Exception("Object should be reachable!");
  }
}
```

При ромізації тесту з використанням модифікації dont-preserve видаляютьються поля removable і unremovable. Видалення поля unremovable призводить до порушення досяжності екземпляра класу B, на який посилалося поле.

Таблиця 3.4

Експеримент 3. Розмір образу в байтах та відносне скорочення розміру

№	Додаток	init-all	base (vs init-all)
1	Hello World	13976	12340 (-11,71 %)
2	Chess	80332	76404 (-4,89 %)
3	Crypto	88560	86684 (-2,12 %)
4	Parallel	86972	84944 (-2,33 %)
5	kXML	83944	80472 (-4,14 %)
6	PNG	67108	63652 (-5,15 %)
7	RegExp	76980	74288 (-3,50 %)
8	AMS MEEP	685884	683832 (-0,30 %)

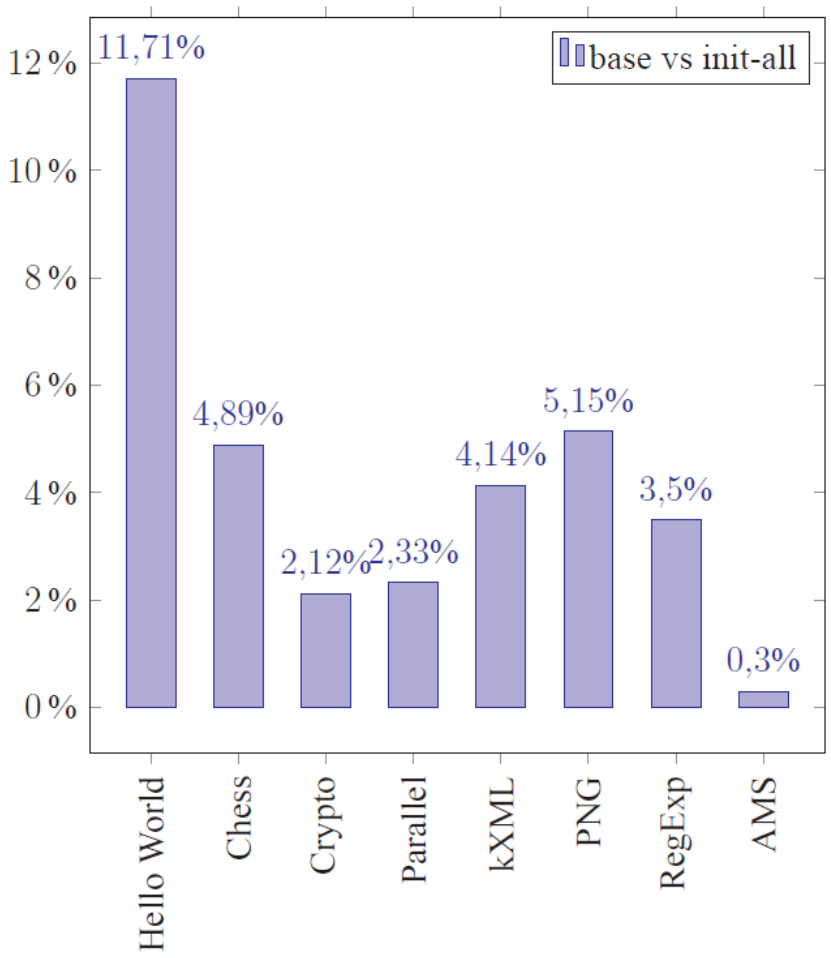


Рис. 3.10. Експеримент 3. Відносне скорочення розміру образу

Об'єкт видаляється збирачем сміття та слабке посилання на цей об'єкт обнулюється. У процесі виконання тесту скидається виняток, оскільки слабке посилання не містить посилання на об'єкт.

Ромізація з використанням запропонованого алгоритму видаляє поле removable, але в той же час не видаляє поле unremovable. Таким чином, зберігається досяжність об'єкта, доступного за допомогою спеціальної посилання та виконання програми завершується коректно.

Програма, згенерована за допомогою інструмента Jamaica Builder, завершує виконання коректно. Jamaica Builder не видаляє поля removable та unremovable, тому що для них є операції запису. Аналогічна поведінка демонструє модифікацію алгоритму dont-remove.

Для оцінки впливу різних критеріїв удалимості полів на розмір образу було виміряно розмір образів, отриманих при використанні вихідного та модифікованих алгоритмів. Інформація про розмір образів та відносне скорочення розміру наведена на рисунку 3.11. Крім того, для кожного образу було виміряно кількість включених до нього полів. Ця інформація наведена у таблиці 3.6 та рисунку 3.12.

Видалення полів, що ініціалізуються, але не використовуються, за допомогою запропонованого критерію скорочує розмір образу на 0,64–4,37 % (середнє значення 1,32%) та скорочує кількість полів в образі на 5,55–13,08% (середнє значення 10,09%). Застосування запропонованого алгоритму аналізу віддаленості полів не призводить до суттєвого скорочення розміру образу, але скорочує кількість полів у образі, що може скоротити динамічне споживання пам'яті.

Видалення всіх полів без урахування семантики фіналізації, що ініціалізуються, але не використовуються, призводить до незначного скорочення розміру образу (середнє значення 0,02%) та кількості полів (середнє значення 1,55%) порівняно із запропонованим алгоритмом.

Таблиця 3.5

Експеримент 4. Розмір образу в байтах та відносне скорочення розміру

№	Додаток	dont-remove	base (vs dont-remove)	dont-preserve (vs base)
1	Hello World	12340	12340 (-4,37 %)	12340 (-0,00 %)
2	Chess	77064	76404 (-0,86 %)	76384 (-0,03 %)
3	Crypto	87240	86684 (-0,64 %)	86664 (-0,02 %)
4	Parallel	85580	84944 (-0,74 %)	84924 (-0,02 %)
5	kXML	81100	80472 (-0,77 %)	80436 (-0,04 %)
6	PNG	64352	63652 (-1,09 %)	63632 (-0,03 %)
7	RegExp	75156	74288 (-1,15 %)	74268 (-0,03 %)
8	AMS MEEP	690152	683832 (-0,92 %)	683812 (-0,00 %)

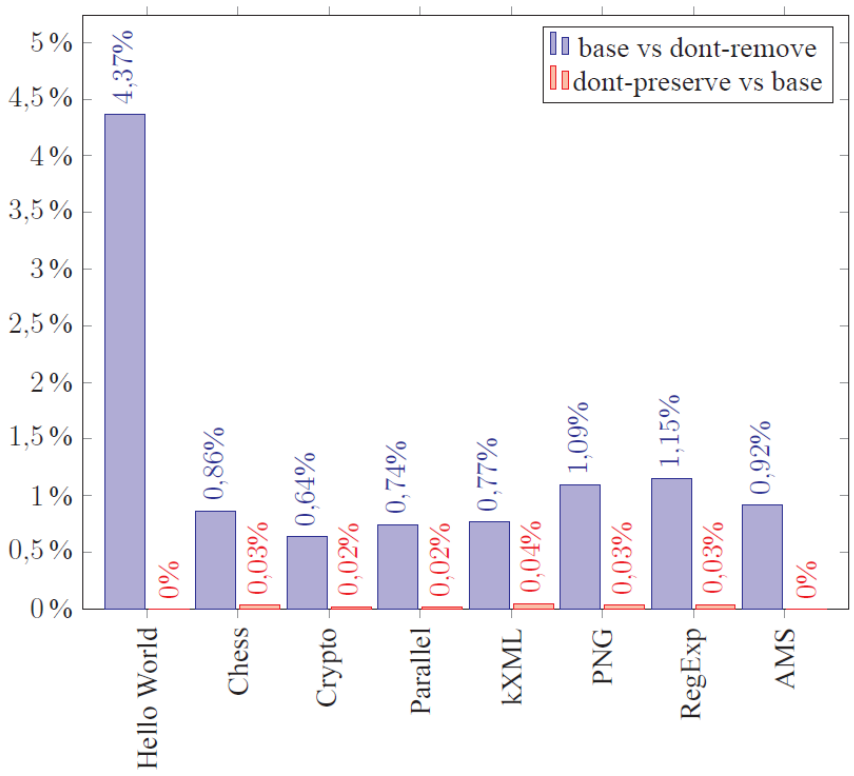


Рис. 3.11. Експеримент 4. Відносне скорочення розміру образу

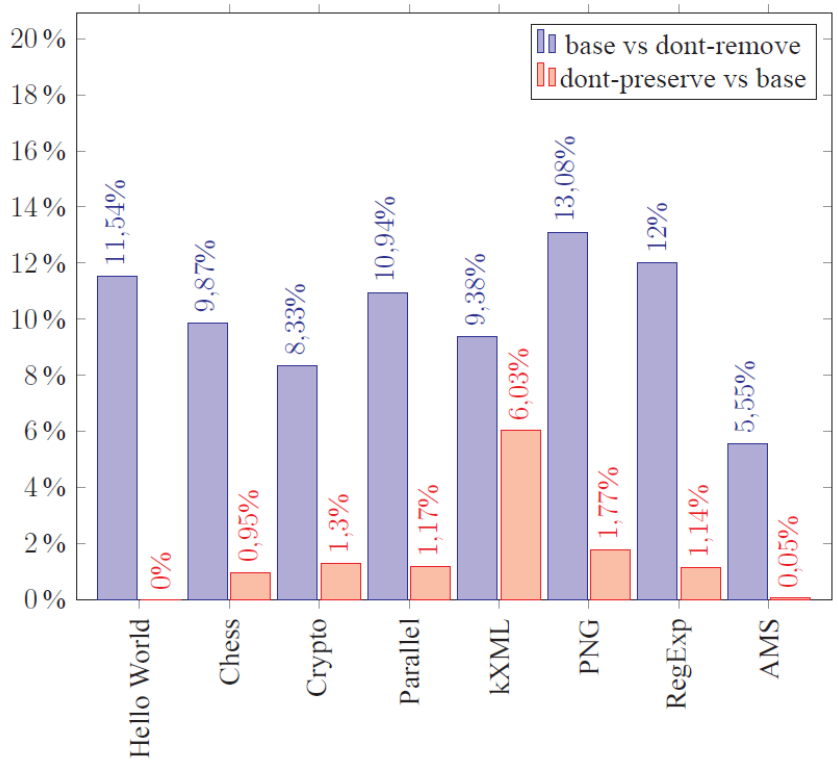


Рис. 3.12. Експеримент 4. Відносне скорочення розміру образу

За результатами проведених експериментів можна зробити наступні висновки:

1. Застосування запропонованих алгоритмів зниження надлишковості в закритій моделі скорочує розмір образу на 57,62-95,18% (середнє значення 74,57%).

2. Застосування запропонованого алгоритму аналізу досяжності методів скорочує розмір образу на 29,57-50,86% (середнє значення 40,88%) проти алгоритму CHA.

3. Застосування вибіркової ініціалізації в процесі аналізу досягнення стискання методів скорочує розмір образу на 0,3–11,71 % (середнє значення 4,27%) порівняно з безумовною ініціалізацією всіх класів сов, що застосовується в системі ExoVM.

4. Видалення полів, що ініціалізуються, але не використовуються, за допомогою запропонованого алгоритму скорочує кількість полів у образі на 5,55–13,08 % (середнє значення 10,09 %) і скорочує розмір образу 0,64-4,37% (середнє значення 1,32%). Видалення всіх полів, що не використовуються, але без урахування семантики фіналізації призводить до незначного скорочення кількості полів (середнє значення 1,55%) і розміру образу (середнє значення 0,02%) по порівнянню з запропонованим алгоритмом.

5. При великій кількості native-методів застосування автоматичного аналізу міжмовних залежностей суттєво спрощує опис додаткових залежностей.

Висновки до третього розділу

1. Наведено опис програмної реалізації алгоритмів, запропонованих у розділі 2. Проведено порівняння запропонованих алгоритмів із існуючими

рішеннями. На підставі цього порівняння сформульовано завдання для експериментального дослідження програмної реалізації.

2. На підставі експериментів можна зробити висновок, що застосування запропонованих алгоритмів при спеціалізації Java-платформи для заданого додатку у закритій моделі дозволяє скоротити апаратні вимоги платформи, не порушуючи поведінку цієї програми.

ВИСНОВКИ

В результаті виконання магістерської роботи отримано наступні теоретичні та практичні результати:

1. Зниження надмірності шляхом видалення полів методів і класів, що не використовуються, широко застосовується для скорочення розміру програми у закритій моделі. Завдання зниження надмірності добре вивчена. Багато відомих алгоритмів були запропоновані для мови C++ і потім були адаптовані для Java.

2. Використання інтерпретованого коду дозволяє скоротити розмір програми у порівнянні з машинним кодом. Спеціалізація набору інструкцій для заданої програми дозволяє досягти ще більш компактне кодування. Таке кодування не вимагає попередньої декомпресії і тому широко застосовується для вбудованих систем із обмеженими ресурсами.

3. Спеціалізовані інструкції забезпечують більш ефективне ко дування за рахунок наступних оптимізацій:

– згортка аргументу – значення часто використовуваного аргументу зафіксовано інструкцією і не кодується в потоці інструкцій;

– укорочування аргументу – аргумент інструкції кодується у потоці інструкцій більш компактно, ніж в оригінальній інструкції;

- згортання послідовності інструкцій - одна інструкція кодує послідовність декількох кодів.

4. Запропоновано алгоритм аналізу видаляємості полів, що дозволяє видаляти ініціалізовані, але не використовувані поля. Запропонований алгоритм зберігає семантику фіналізації та не видаляє поля, якщо це може порушити досяжність об’єктів, фіналізація яких необхідна додатку.

5. Розглянуто завдання скорочення розміру інтерпретації коду шляхом спеціалізації набору інструкцій. Цей підхід є розвитком широко використовуваної ідеї словникового стискування. Був запропонований

алгоритм спеціалізації набору інструкцій Java байт-коду, що скорочує розмір коду заданого додатку та інтерпретатора, необхідного для його виконання.

6. Наведено опис програмної реалізації алгоритмів, запропонованих у роботі. Проведено порівняння запропонованих алгоритмів із існуючими рішеннями. На підставі цього порівняння сформульовано завдання для експериментального дослідження програмної реалізації.

7. На підставі експериментів можна зробити висновок, що застосування запропонованих алгоритмів при спеціалізації Java-платформи для заданого додатку у закритій моделі дозволяє скоротити апаратні вимоги платформи, не порушуючи поведінку цієї програми.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мартін Фаулер, Джеймс Льюїс Мікросервіси [Електронний ресурс]. – Режим доступу : https://martinfowler.com/articles/microservices.html
2. Стівен Уоттс, Лора Шифф Огляд архітектури моноліту проти мікросервісів [Електронний ресурс]. – Режим доступу : https://www.bmc.com/blogs/microservices-architecture/
3. Douce, C.; Livingstone, D.; Orwell, J. Automatic Test-Based Assessment of Programming: A Review. J. Educ. Resour. Comput. 2005, 5, 215–221. [Google Scholar] [CrossRef]
4. Ala-Mutka, K.M. A Survey of Automated Assessment Approaches for Programming Assignments. Comput. Sci. Educ. 2005, 15, 83–102. [Google Scholar] [CrossRef]
5. Lajis, A.; Baharudin, S.A.; Kadir, D.A.; Ralim, N.M.; Nasir, H.M.; Aziz, N.A. A Review of Techniques in Automatic Programming Assessment for Practical Skill Test. J. Telecommun. Electron. Comput. Eng. 2018, 10, 109–113. [Google Scholar]