Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет «Высшая школа экономики»

Факультет компьютерных наук

Магистерская программа «Науки о данных»

Департамент анализа данных и искусственного интеллекта

КУРСОВАЯ РАБОТА

На тему: Логические методы верификации программного обеспечения (Logical Methods for Software Verification)

Студент группы № м15НоД_ИССА
Мосин Василий Михайлович
(Ф.И.О.)

Руководитель КР
Профессор, д. ф.-м. н.,
Канович Макс Иосифович
(должность, звание, Ф.И.О.)

Консультант*
_____
(должность, звание, Ф.И.О.)

Москва, 2016

* указывается в случае назначения консультанта

Содержание

Глава 1

Введение

1.1 Общий обзор

В современной индустрии очень высокий темп разработки программного обеспечения, что делает время разработчиков очень ценным ресурсом. Автоматическая верификация существенно упрощает процесс создания программы, экономя время разработчиков. Сейчас широко применяется такой вид автоматической верификации, как статический анализ кода. Его популярность обуславливается тем, что этот метод не требует запуска программы и способен выдавать результат, основываясь лишь на анализе кода без его выполнения. Важную роль в статическом анализе играют логические методы. Однако для их применения в верификации программного обеспечения необходима теоретическая основа, позволяющая сделать переход от логических правил и выводов непосредственно к анализу кода. Такая теория уже существует, и в этой работе предварительно будут рассмотрены ее основы.

Вообще, область формальной верификации программ относительно недавно начала интенсивно развиваться. Но уже существует большое количество различных работ и статей на эту тему. К сожалению, данной тематикой в России пока мало занимаются, в основном исследования проводятся за рубежом. Активные разработки по применению логических методов в верификации программ ведутся в известной компании Facebook.

Большая часть статей, рассмотренных мной при подготовке к данной работе носит теоретический характер. Очень мало работ, в которых рассматриваются практические результаты применения логических методов в верификации программного обеспечения. Этот факт совместно с малым количеством русскоязычных работ по данной тематике стал для меня

основной причиной, по которой я решил рассмотреть и описать основы применения одного из логических методов, а также на простом примере показать реализацию программы, основанной на одном из существующих алгоритмов, и продемонстрировать результаты ее выполнения в реальных условиях.

1.2 Структура работы

Глава 2 (Предпосылки и обзор литературы) – рассматривает основные работы в данной области, затрагивающие как общие вопросы, так и частные. Эта глава показывает, как раскрывается понятие абдукции в существующей литературе, а также описывает теоретический аппарат для дальнейшего использования в работе.

Глава 3 (Идея разработки программы) – демонстрирует технические особенности и предположения устройства Java-программ, рассматривает случаи утечки памяти в них, а также описывает алгоритм, на котором основывается разработка.

Глава 4 (Реализация и описание программы) – аргументирует выбор языка разработки, объясняет структуру программы и последовательно описывает все функции в созданной реализации.

Глава 5 (Тестирование и анализ результатов) – приводит примеры кода тестируемых программ, иллюстрирует результаты и проводит их анализ.

Глава 6 (Заключение) – делает общие выводы, приводит примеры возможных улучшений созданной программы.

Глава 2

Предпосылки и обзор литературы

2.1 Абдукция

Понятие абдукции очень широкое. Оно подразумевает вывод гипотез из логических суждений. Абдукция активно применяется в философии. Здесь мы рассмотрим значения данного понятия в стандартной логике и в разделительной логике.

2.1.1 Абдукция в стандартной логике

В стандартной логике абдукция подразумевает в себе следующее.

Имея суждения A и B, необходимо найти такую гипотезу X, чтобы логический вывод

$$A \wedge X \vdash B$$

был верен. Решение X должно быть минимальным [1]. Это значит, что X должно содержать только необходимое для вывода B количество информации.

2.1.2 Абдукция в разделительной логике

Разделительная логика – это расширение так называемой логики Хора, которая используется для описания спецификации программ. Под спецификацией в данном случае понимаются тройки Хора вида

$$\{P\}\ C\ \{Q\},$$

где C – программа, P – предусловие, Q – постусловие. Здесь P и Q – это формулы разделительной логики.

Понятие абдукции в данном случае подразумевает в себе следующее.

Имея суждения A и B, необходимо найти такую гипотезу X, чтобы вывод

$$A * X \vdash B$$

был верен. Здесь $A * X$ – это разделяющая конъюнкция. Она верна для части памяти только тогда, когда существует такое ее разбиение на две непересекающиеся части памяти a и x, что A верно для a и X верно для x. В данном контексте под X можно подразумевать описание недостающей для вывода B части памяти.

2.2 Би-абдукция

Для анализа программ в основном применяется обобщение понятия абдукции – би-абдукция. Данное понятие состоит в комбинации абдукции и фреймового вывода.

2.2.1 Фреймовый вывод

Фреймовый вывод – двойственная к абдукции задача. Она состоит в решении следующей задачи.

Для двух частей памяти H и H' найти фрейм F, чтобы вывод

$$H \vdash H' * F$$

был верен. Под F можно понимать описание нерассмотренной части памяти, которая присутствует в H, но не присутствует в H'.

2.2.2 Определение би-абдукции

Таким образом, би-абдукция состоит в решении следующей задачи.

Для описаний H и H' конфигураций двух существующих частей памяти необходимо найти такие A и F, чтобы был верен вывод

$$H * A \vdash H' * F,$$

где F – это фрейм, а A называется анти-фрейм [2]. Здесь A соответствует X, обозначенному в предыдущем разделе.

2.3 Композиционный анализ формы

Для дальнейших рассуждений необходимо предварительно рассмотреть понятие композиционного анализа формы, которое применяется при автоматической верификации программ. Такое сложное название можно разделить два подпонятия – анализ формы и композиционность, которые будут рассмотрены в данной главе.

2.3.1 Анализ формы

Анализ формы – это метод статического анализа. Он заключается в исследовании свойств динамических структур данных и называется так, потому что эти свойства связаны с “формой” памяти [3]. Особенность анализа формы заключается в том, что он позволяет проводить автоматическую верификацию программы, содержащей указатели и связанные листы. Этот метод можно применять для поиска утечек памяти, висячих указателей, ошибок, связанных с переполнением массивов.

2.3.2 Композиционность

При применении би-абдукции к какой-либо функции происходит вычисление так называемого следа функции – части памяти, которую функция использует [4]. Благодаря этому становится возможным использовать локальный вывод спецификаций в глобальном контексте программы. Это значит, что можно составлять спецификации для каждой функции отдельно, вычисляя при этом след каждой такой функции независимо от других функций. Далее вычисленные таким образом спецификации функций используются для анализа формы всей программы путем составления троек Хора для каждой функции, начиная с конца программы. При этом каждая такая новая тройка составляется с учетом вычисленных на предыдущих этапах троек. В этом и заключается композиционность.

Методы автоматической верификации использующие композиционный анализ имеют способность к масштабированию, то есть они продолжают

хорошо работать, анализируя даже большие программы, так как такой анализ работает последовательно для маленьких частей кода.

Глава 3

Идея разработки программы

В данной главе выделяются основные моменты, связанные с памятью в Java-программах, описывается алгоритм для обнаружения утечек памяти и приводится наглядный пример его применения.

3.1 Java-программы

3.1.1 Сборщик мусора

Java использует специальный сборщик мусора, который очищает неиспользуемую память в программе. Он очень хорошо помогает разработчику, освобождая его от подобного рода задач. Однако, сборщик мусора не всегда работает полностью корректно. Суть его работы заключается в том, что он очищает только ту часть памяти, которая становится недоступна в ходе использования программы [5]. Но бывают и другие случаи, когда память нужно очищать.

3.1.2 Утечки памяти

В Java-программе обычно бывает так, что на часть памяти, которая уже была использована и готова к очистке, все еще ссылаются некоторые переменные из программы. Чаще всего это случается из-за того, что разработчик забыл присвоить *null* объекту, который больше не нужен. В этом случае сборщик мусора не сможет освободить данный участок памяти. Если размер этой части памяти очень большой, то это может привести к серьезным проблемам в работе программы.

3.1.3 Упрощенная Java-программа

В данной работе для наглядности результатов используется упрощенная модель Java-программы. Эта модель сознательно использует только основные

процедуры и допускает обобщение на случай большего количество процедур. Далее в работе используются следующие основные Java-процедуры:

- Создание объекта с помощью оператора *new*:
  Example x = new Example();
- Присвоение значения объекту через обращение с помощью точки к свойству *value* объекта:
  x.value = val;
- Явное присвоение *null* ненужному больше объекту:
  x = null;

3.2 Описание алгоритма

В работе реализован уже существующий Алгоритм 1 из статьи [5].

Алгоритм 1. Поиск утечек памяти

Plocs := *LabelPgm*(1; *Prg*);
Mspecs := *CompSpecs*();
LocPre := *ForwardAnalysis*(*Mspecs*);
LocFp := *BackwardAnalysis*(*Mspecs*);
forall *loc* ∈ *Plocs* **do**
    Pre := *LocPre*(*loc*);
    Fp := *LocFp*(*loc*);
    MLeak(*loc*) := $\{ R \mid H_1 \vdash H_2 * R \wedge H_1 \in Pre \wedge H_2 \in Fp \}$
end for

Алгоритм 1 определяет, какие объекты являются утечками памяти для каждой существенной точки в программе. Поэтому первой процедурой в этом алгоритме является *LabelPgm*(), которая вычисляет позиции в программе, для которых в дальнейшем будут определяться утечки памяти. Существенной точкой считаются те позиции, на которых находятся простые команды, не состоящие в *while* или *if* выражениях. В данной работе рассматривается

реализация Алгоритма 1, работающего только с линейными программами, то есть не содержащими ветвлений и циклов. Поэтому утечки памяти в данном случае буду вычисляться для каждой команды в программе.

Далее выполняется процедура *CompSpecs*(), которая вычисляет минимальные спецификации для каждого метода программы. Минимальные в этом случае значит необходимые для успешного выполнения метода без ошибок. Следом выполняются две основные процедуры *ForwardAnalysis*() и *BackwardAnalysis*(), которые называются прямой и обратный анализ и описаны отдельно в следующих двух разделах. В результате прямого анализа с помощью символьного выполнения программы вычисляются состояния памяти в каждой существенной точке. Но эти состояния зачастую получаются избыточными. А в результате обратного анализа вычисляются состояния памяти для каждой существенной точки в программе достаточные для безопасного выполнения всех процедур. На последнем этапе в цикле для каждой позиции в программе сравниваются предусловия, полученные в результате прямого и обратного анализа. Такое сравнение происходит с помощью фреймового вывода, и все избыточные состояния, которые были получены в результате прямого анализа, считаются утечками памяти.

3.2.1 Прямой анализ

Прямой анализ производится итеративно путем символьного выполнения каждой команды в программе, начиная с первой. На каждом шаге берется предусловие $\{P\}$, выполняется команда C и вычисляется постусловие $\{Q\}$, которое будет являться предусловием на следующей итерации. Таким образом, будут получены все избыточные состояния для каждой точки в программе.

3.2.2 Обратный анализ

Предположим, программа состоит из последовательности n команд $\{P_1\}C_1\{Q_1\}, \{P_2\}C_2\{Q_2\}, \dots, \{P_n\}C_n\{Q_n\}$, где $\{P_i\}$ и $\{Q_i\}$ – пред- и постусловия для команды C_i, вычисленные с помощью процедуры *CompSpecs*(). Обратный анализ начинается с конца программы. При это используется так называемое Би-абдуктивное Правило Последовательных Действий. Оно заключается в применении би-абдукции:

$$\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 * A\}C_1; C_2\{Q_2 * F\}}\ Q_1 * A \vdash P_2 * F,$$

где F и A – фрейм и анти-фрейм для композиции команд C_1 и C_2. Вычисленная таким образом спецификация для композиции последних двух команд применяется далее для вычисления спецификации для композиции последних трех команд и так далее. В результате вычисляются спецификации для всех команд.

3.2.3 Обобщение

Алгоритм 1 является алгоритмом статического анализа, который выявляет в каждой точке программы объекты, которые в дальнейшем не используются, но все еще доступны из программы. Сборщик мусора не очищает такие объекты, поэтому Алгоритм 1 очень полезен для предотвращения переполнения памяти.

Рассмотренный Алгоритм 1 основан на вычислении следа кусков кода, в результате чего становится возможным различать размещенные в памяти объекты, которые действительно нужны при выполнении программы, от тех, которые программой не используются [5]. Также, применение би-абдукции в данном алгоритме позволяет сделать его композиционным, за счет чего Алгоритм 1 приобретает масштабируемость к реальным программам с большим количеством кода и становится возможным его применение к неполным кускам кода.

3.3 Пример применения алгоритма

Для наглядности приведем пример работы Алгоритма 1 на простой Java-программе, показанной на Рисунке 1.

```
Testing a = new Testing();
Testing b = new Testing();
a.value = val;
b.value = val;
a = null;
```

Рисунок 1. Пример Java-программы

Данная программа содержит всего пять команд. Сначала создаются два объекта. Потом им присваиваются значения. В конце один из объектов очищается, второй остается в памяти и все еще доступен из программы. Применим Алгоритм 1 для поиска утечки памяти в рассмотренной выше программе.

Для удобства введем наименования для команд последовательно следующим образом: первая команда – $C1$, вторая – $C2$ и так далее. Так как программа линейная, то все точки в ней существенны, и спецификации будут вычисляться для каждой из них. Процедура *CompSpecs*() вычислит следующие спецификации для команд:

- $\{emp\}C1\{a \longmapsto \_\}$
- $\{emp\}C2\{b \longmapsto \_\}$
- $\{a \longmapsto \_\}C3\{a \longmapsto val\}$
- $\{b \longmapsto \_\}C4\{b \longmapsto val\}$
- $\{a \longmapsto val\}C5\{emp\}$

Здесь используются следующие обозначения спецификаций:

- $\{emp\}$ – в памяти ничего нет
- $\{a \longmapsto \_\}$ - в памяти размещен объект a без значения
- $\{a \longmapsto val\}$ – объекту a присвоено значение val

Далее выполняется процедура *ForwardAnalysis*() прямого анализа, используя вычисленные на предыдущем шаге спецификации. Данная процедура вернет следующую цепочку спецификаций:

$$\{emp\}C1\{a \longmapsto \_\}C2\{a \longmapsto \_ * b \longmapsto \_\}C3\{a \longmapsto val * b \longmapsto \_\}C4$$

$$C4\{a \longmapsto val * b \longmapsto val\}C5\{b \longmapsto val\}$$

После этого выполняется процедура *BackwardAnalysis*() обратного анализа, начиная с вычисления тройки Хора для комбинации команд *C*4 и *C*5. При этом используется описанное ранее Би-абдуктивное Правило Последовательных Действий с учетом спецификаций, вычисленных с помощью процедуры *CompSpecs*():

$$\frac{\{b \longmapsto \_\}C4\{b \longmapsto val\} \quad \{a \longmapsto val\}C5\{emp\}}{\{b \longmapsto \_ * A\}C4; C5\{emp * F\}}\ b \longmapsto val * A \vdash a \longmapsto val * F.$$

Здесь $A = a \longmapsto val$, $F = b \longmapsto val$. Таким образом, тройка Хора для композиции команд *C*4 и *C*5 имеет вид:

$$\{b \longmapsto \_ *\ a \longmapsto val\}C4; C5\{\ b \longmapsto val\}.$$

Учитывая это, вычисляется тройка Хора для композиции команд *C*3, *C4* и *C*5:

$$\frac{\{a \longmapsto \_\}C3\{a \longmapsto val\} \quad \{b \longmapsto \_ *\ a \longmapsto val\}C4; C5\{\ b \longmapsto val\}}{\{a \longmapsto \_ * A\}C3; C4; C5\{b \longmapsto val * F\}}$$

$$a \longmapsto val * A \vdash b \longmapsto \_ * a \longmapsto val * F.$$

Здесь $A = b \longmapsto \_$, $F = emp$. То есть тройка Хора для композиции команд *C*3, *C4* и *C*5 имеет вид:

$$\{a \longmapsto \_ * b \longmapsto \_\}C3; C4; C5\{b \longmapsto val\}.$$

Аналогично, вычисляется спецификация для композиции команд *C*2, *C*3, *C4* и *C*5:

$$\frac{\{emp\}C2\{b \longmapsto \_\} \quad \{a \longmapsto \_ * b \longmapsto \_\}C3; C4; C5\{b \longmapsto val\}}{\{emp * A\}C2; C3; C4; C5\{b \longmapsto val * F\}}$$

$$b \longmapsto \_ * A \vdash a \longmapsto \_ * b \longmapsto \_ * F.$$

Здесь $A = a \mapsto \_$, $F = emp$. То есть спецификация для композиции команд $C2$, $C3$, $C4$ и $C5$ имеет вид:

$$\{a \mapsto \_\}C2; C3; C4; C5\{b \mapsto val\}.$$

Наконец, вычисляется спецификация для всей программы:

$$\frac{\{emp\}C1\{a \mapsto \_\} \quad \{a \mapsto \_\}C2; C3; C4; C5\{b \mapsto val\}}{\{emp * A\}C1; C2; C3; C4; C5\{b \mapsto val * F\}}$$

$$a \mapsto \_ * A \vdash a \mapsto \_ * F.$$

Здесь $A = emp$, $B = emp$. Спецификация для всей программы:

$$\{emp\}C1; C2; C3; C4; C5\{b \mapsto val\}.$$

Для наглядности представим предусловия, полученные с помощью прямого и обратного анализа, в виде Таблицы 1:

Таблица 1. Предусловия посредством прямого и обратного анализа

	$C1$	$C2$	$C3$	$C4$	$C5$
Pre	emp	$a \mapsto \_$	$a \mapsto \_ * b \mapsto \_$	$a \mapsto val * b \mapsto \_$	$a \mapsto val * b \mapsto val$
Fp	emp	$a \mapsto \_$	$a \mapsto \_ * b \mapsto \_$	$b \mapsto \_ * \ a \mapsto val$	$a \mapsto val$

Здесь *Pre* – предусловия, полученные в результате прямого анализа, *Fp* – предусловия, полученные в результате обратного анализа.

Все полученные предусловия равны, кроме предусловия для последней команды. В результате фреймового вывода:

$$a \mapsto val * b \mapsto val \vdash a \mapsto val * F$$

получается, что фрейм $F = b \mapsto val$. Это значит, что $b \mapsto val$ – это избыточное состояние памяти перед выполнением команды $C5$, а следовательно, $b \mapsto val$ является утечкой памяти.

В итоге, на простом примере была показана суть работы Алгоритма 1, который верно определил утечку памяти в демонстрационной программе.

Глава 4

Реализация и описание программы

Для реализации Алгоритма 1 был выбран язык Python 3. При реализации Алгоритма 1 очень много работы со строками и списками строк, начиная от считывания верифицируемого кода, который представлен в виде текста, и заканчивая операциями со спецификациями, которые я решил представлять в виде строковых констант. Язык Python 3 предоставляет удобный механизм работы со списками строк. Поэтому этот язык в данной работе используется обоснованно. В данной главе последовательно описаны все функции, используемые при реализации Алгоритма 1. Для наглядности описания функций дополнены их выводом после выполнения работы программы на демонстрационном примере из предыдущей главы, показанном на Рисунке 1, которые можно сопоставить с рассуждениями из раздела 3.3.

4.1 Чтение программы

Реализация функции чтения программы *program_reading*(*file*) представлена в Листинге 1.

Листинг 1. Функция чтения программы

```
def program_reading(file):
    f = open(file)
    program = []
    for line in f:
        if line.find('new') != -1:
            program.append([line.split(' ')[1], 'new'])
            continue
        if line.find('.') != -1:
            program.append([line.split('.')[0], '.'])
            continue
        if line.find('null') != -1:
            program.append([line.split(' ')[0], 'null'])
            continue
        print("ERROR")
        f.close()
        return
    f.close()
    return program
```

На вход подается название файла с Java-программой. Чтение программы происходит построчно. Каждая строка проверяется на наличие служебных слов и знаков (‘*new*’, ‘.’, ’*null*’). Команда запоминается в результирующем списке в виде пары [‘*объект*’, ‘*служебное слово/знак*’]. Если в строке не было найдено ни одного служебного слова или знака, то выводится сообщение об ошибке на экран. Пример результата выполнения работы данной функции приведен на Рисунке 2.

```
[['a', 'new'], ['b', 'new'], ['a', '.'], ['b', '.'], ['a', 'null']]
```

Рисунок 2. Пример результата выполнения функции *program_reading*(*file*)

4.2 Вычисление спецификаций

Реализация функции вычисления спецификаций *specification_computing*(*program*) представлена в Листинге 2.

Листинг 2. Функция вычисления спецификаций

```
def specification_computing(program):
    spec_post = []
    spec_pre = []
    for method in program:
        if method[1] == 'new':
            spec_pre.append([['emp']])
            spec_post.append([[method[0], '_']])
            continue
        if method[1] == '.':
            spec_pre.append([[method[0], '_']])
            spec_post.append([[method[0], 'val']])
            continue
        if method[1] == 'null':
            spec_pre.append([[method[0], 'val']])
            spec_post.append([['emp']])
            continue
    return spec_pre, spec_post
```

На вход подается считанная в виде списка команд с помощью функции из предыдущего раздела программа. Для каждого элемента в списке (для каждой команды) в зависимости от служебного слова/знака, стоящего на второй позиции сохраняются пред- и постусловия каждой команды в

специальных списках. Результат выполнения данной функции показан на Рисунке 3.

```
[[['emp']], [['emp']], [['a', '_']], [['b', '_']], [['a', 'val']]]
[[['a', '_']], [['b', '_']], [['a', 'val']], [['b', 'val']], [['emp']]]
```

Рисунок 3. Пример результата выполнения функции *specification_computing*(*program*)

Как видно, функция *specification_computing*(*program*) возвращает два списка: первый – список предусловий, второй – список постусловий для каждой команды. Здесь и далее [‘emp’] означает, что память пуста, [‘a’, ‘_’] означает, что объект *a* размещен в памяти, [‘a’, ‘val’] означает, что объекту *a* присвоено значение.

4.3 Прямой анализ

Функция *forward_analysis*(*program*), выполняющая прямой анализ, представлена в Листинге 3.

Листинг 3 (начало). Функция прямого анализа

```
def forward_analysis(program):
    forward = []
    forward.append([['emp']])
    for method in program:
        if method[1] == 'new':
            if forward[-1] == [['emp']]:
                forward.append([[method[0], '_']])
                continue
            if forward[-1].count([method[0], '_']) != 0:
                temp = forward[-1].copy()
                forward.append(temp)
                continue
            if forward[-1].count([method[0], 'val']) != 0:
                temp = forward[-1].copy()
                ind = temp.index([method[0], 'val'])
                temp.remove([method[0], 'val'])
                temp.insert(ind, [method[0], '_'])
                forward.append(temp)
                continue
            else:
                temp = forward[-1].copy()
                temp.append([method[0], '_'])
                forward.append(temp)
                continue
```

Листинг 3 (продолжение). Функция прямого анализа

```
    if method[1] == '.':
        if forward[-1] == [['emp']]:
            forward.append([[method[0], 'val']])
            continue
        if forward[-1].count([method[0], 'val']) != 0:
            temp = forward[-1].copy()
            forward.append(temp)
            continue
        if forward[-1].count([method[0], '_']) != 0:
            temp = forward[-1].copy()
            ind = temp.index([method[0], '_'])
            temp.remove([method[0], '_'])
            temp.insert(ind, [method[0], 'val'])
            forward.append(temp)
            continue
        else:
            temp = forward[-1].copy()
            temp.append([method[0], 'val'])
            forward.append(temp)
            continue
    if method[1] == 'null':
        if forward[-1] == [['emp']]:
            forward.append([['emp']])
            continue
        if forward[-1].count([method[0], 'val']) != 0:
            temp = forward[-1].copy()
            temp.remove([method[0], 'val'])
            if temp:
                forward.append(temp)
                continue
            else:
                forward.append([['emp']])
                continue
        if forward[-1].count([method[0], '_']) != 0:
            temp = forward[-1].copy()
            temp.remove([method[0], '_'])
            if temp:
                forward.append(temp)
                continue
            else:
                forward.append([['emp']])
                continue
        else:
            temp = forward[-1].copy()
            forward.append(temp)
            continue
forward.pop()
return forward
```

На вход подается считанная на первом этапе программа. Программа обрабатывается построчно. На каждом шаге в зависимости от вида команды и от предыдущего состояния памяти вычисляется состояние памяти программы после выполнения команды. Функция возвращает список предусловий для каждой команды. Пример результата выполнения функции приведен на Рисунке 4.

```
[[['emp']], [['a', '_']], [['a', '_'], ['b', '_']],
 [['a', 'val'], ['b', '_']], [['a', 'val'], ['b', 'val']]]
```

Рисунок 4. Пример результата выполнения функции *forward_analysis*(*program*)

4.4 Обратный анализ

Код функции *backward_analysis*(*pre*, *post*), выполняющей обратный анализ приведен в Листинге 4.

Листинг 4. Функция обратного анализа *backward_analysis*(*pre*, *post*)

```
def backward_analysis(pre, post):
    backward = []
    backward.append(pre[-1])
    loc = len(pre)
    i = 1
    while i < loc:
        abduct = bi_abduction(post[-i-1], backward[0])[0]
        backward.insert(0, separating_conjunction(pre[-i-1], abduct))
        i += 1
    return backward
```

На вход подаются списки пред- и постусловий, вычисленные с помощью функции *specification_computing*(*program*). Функция работает итеративно. Функция обратного анализа основана на использовании Би-абдуктивного Правила Последовательных Действий на каждом шаге, поэтому внутри функции используются отдельные процедуры би-абдукции и разделяющей конъюнкции, которые будут описаны ниже. Здесь и далее в коде переменная с названием *abduct* означает анти-фрейм, вычисленный в результате применения функции би-абдукции.

Функция обратного анализа, аналогично функции прямого анализа, возвращает список предусловий. Пример результата выполнения функции обратного анализа приведен на Рисунке 5.

```
[[['emp']], [['a', '_']], [['a', '_'], ['b', '_']],
 [['b', '_'], ['a', 'val']], [['a', 'val']]]
```

Рисунок 5. Пример результата выполнения функции обратного анализа *backward_analysis*(*pre*, *post*)

4.4.1 Би-абдукция

Код функции *bi_abduction*(*post*, *pre*) приведен в Листинге 5.

Листинг 5. Функция би-абдукции *bi_abduction*(*post*, *pre*)

```
def bi_abduction(post, pre):
    post1_copy = post.copy()
    pre2_copy = pre.copy()
    if post1_copy == [['emp']]:
        abduct = pre2_copy
        frame = [['emp']]
        return abduct, frame
    if pre2_copy == [['emp']]:
        abduct = [['emp']]
        frame = post1_copy
        return abduct, frame
    for element in post1_copy:
        if pre2_copy.count(element) != 0:
            post1_copy.remove(element)
            pre2_copy.remove(element)
    for element_post in post1_copy:
        for element_pre in pre2_copy:
            if element_post[0] == element_pre[0]:
                if element_post[1] == 'val' and element_pre[1] == '_':
                    post1_copy.remove(element_post)
                    pre2_copy.remove(element_pre)
                if element_post[1] == '_' and element_pre[1] == 'val':
                    post1_copy.remove(element_post)
    if not pre2_copy:
        abduct = [['emp']]
    else:
        abduct = pre2_copy
    if not post1_copy:
        frame = [['emp']]
    else:
        frame = post1_copy
    return abduct, frame
```

Эта функция принимает в качестве аргументов два состояния памяти – *post* и *pre* и находит для них и возвращает фрейм *F* и анти-фрейм *A*, так чтобы следующий вывод был верен:

$$post * A \vdash pre * F.$$

4.4.2 Разделяющая конъюнкция

Функция *separating_conjunction*(*list1*, *list2*) предназначена для вычисления предусловия после того, как с помощью функции *bi_abduction*(*post*, *pre*) был найден анти-фрейм. Данная функция принимает на вход два состояния памяти (одно из них – вычисленный анти-фрейм) в виде списков *list1* и *list2* и возвращает состояние памяти в результате применения разделяющей конъюнкции вида:

$$list1 * list2.$$

Код, реализующий функцию *separating_conjunction*(*list1*, *list2*), представлен в Листинге 6.

Листинг 6. Функция разделяющей конъюнкции *separating_conjunction*(*list1*, *list2*)

```
def separating_conjunction(list1, list2):
    temp1 = list1.copy()
    temp2 = list2.copy()
    if temp1 == [['emp']]:
        return temp2
    if temp2 == [['emp']]:
        return temp1
    for element2 in temp2:
        for element1 in temp1:
            if element1[0] == element2[0]:
                if element1[1] != element2[1]:
                    element = [element1[0], 'val']
                    ind1 = temp1.index(element1)
                    ind2 = temp2.index(element2)
                    temp1.pop(ind1)
                    temp2.pop(ind2)
                    temp1.insert(ind1, element)
                    temp2.insert(ind2, element)
    for element2 in temp2:
        if temp1.count(element2) == 0:
            temp1.append(element2)
    return temp1
```

4.5 Фреймовый вывод

Функция фреймового вывода *frame_inference*(*list1*, *list2*) принимает на вход список *list1* предусловий, полученных с помощью прямого анализа, и список *list2* предусловий, полученных с помощью обратного анализа. Код, реализующий данную функцию, представлен в Листинге 7.

Листинг 7. Функция фреймового вывода *frame_inference*(*list1*, *list2*)

```
def frame_inference(list1, list2):
    leaks = []
    loc = len(list1)
    i = 0
    while i < loc:
        temp1 = list1[i].copy()
        temp2 = list2[i].copy()
        if temp1 == [['emp']]:
            leaks.append([['emp']])
            i += 1
            continue
        if temp2 == [['emp']]:
            leaks.append(temp1)
            i += 1
            continue
        temp = []
        for element1 in temp1:
            if element1 not in temp2:
                temp.append(element1)
        temp1 = temp
        for element1 in temp1:
            for element2 in temp2:
                if element1[0] == element2[0]:
                    temp1.remove(element1)
                    temp1.append([element2[0], '_'])
        if not temp1:
            leaks.append([['emp']])
        else:
            leaks.append(temp1)
        i += 1
    return leaks
```

В качестве результата функция фреймового вывода возвращает для каждой точки в программе все те избыточные состояния памяти, которые есть в *list1*, но нет в *list2*. Данный результат является окончательным и содержит утечки памяти для каждой программной точки. Пример результата функции фреймового вывода показан на Рисунке 6.

```
[[['emp']], [['emp']], [['emp']], [['emp']], [['b', 'val']]]
```

Рисунок 6. Пример результата выполнения функции фреймового вывода *frame_inference*(*list1*, *list2*)

4.6 Вывод результата

При поиске утечек памяти в больших программах вывод результат в виде цепочки утечек памяти, как на Рисунке 6, не очень удобен. Поэтому программа была дополнена простой вспомогательной функцией *printing*(*list1*), которая производит вывод результата в другом виде. Код данной функции представлен в Листинге 8.

Листинг 8. Функция вывода результата *printing*(*list1*)

```
def printing(list1):
    for element in list1:
        print(element)
```

Данная функция принимает на вход список вычисленных ранее утечек памяти и выводит их на экран построчно, для каждой команды в программе, начиная с первой. Пример такого вывода представлен на Рисунке 7.

```
[['emp']]
[['emp']]
[['emp']]
[['emp']]
[['b', 'val']]
```

Рисунок 7. Пример выполнения функции *printing*(*list1*)

Результат точно такой же что и на Рисунке 6, только в более удобном виде. Например, здесь в момент выполнения первых четырех команд утечек памяти нет, а в момент выполнения последней команды появилась одна утечка памяти.

4.7 Общая последовательность выполнения функций

Все функции, используемые в программе, были описаны выше. Таким образом, для поиска утечек памяти в нужной программе необходимо выполнить все процедуры в порядке, представленном в Листинге 9.

Листинг 9. Общая последовательность выполнения функций

```
file = 'program.java'
program = program_reading(file)
pre, post = specification_computing(program)
forward = forward_analysis(program)
backward = backward_analysis(pre, post)
leaks = frame_inference(forward, backward)
printing(leaks)
```

Здесь переменной *file* присваивается названия файла, в котором лежит Java-программа для верификации. В результате последовательного выполнения функций на экран выводятся обнаруженные утечки памяти для каждой программной точки.

Глава 5

Тестирование и анализ результатов

В данной главе проверяется работы программы на различных входных данных и описываются полученные результаты. Для этой цели были вручную сгенерированы три Java-кода, содержащие команды, описанные в разделе 3.1.3, в линейном порядке.

5.1 Тест 1

Для первого теста был сгенерирован простой Java-код, показанный на Рисунке 8.

```
Testing a = new Testing();
a.value = val;
Testing b = new Testing();
a = null;
b.value = val;
Testing c = new Testing();
c.value = val;
c = null;
```

Рисунок 8. Программа 1

Здесь фигурируют три объекта, каждому из которых было присвоено значение. Однако один объект не обнуляется. В этом случае программа правильно определила утечку памяти, берущую начало после пятой команды. Результат вывода программы представлен на Рисунке 9.

```
[['emp']]
[['emp']]
[['emp']]
[['emp']]
[['emp']]
[['b', 'val']]
[['b', 'val']]
[['b', 'val']]
```

Рисунок 9. Результат 1

5.2 Тест 2

Для второго теста тоже используется довольно простой код. Однако здесь намеренно допущена логическая ошибка – значение одному из объектов присваивается до его создания, хотя формально программа работает без утечек памяти. Код второй тестовой программы представлен на Рисунке 10.

```
Testing a = new Testing();
Testing b = new Testing();
c.value = val;
Testing c = new Testing();
a.value = val;
b.value = val;
a = null;
b = null;
c = null;
```

Рисунок 10. Программа 2

В этом случае программа выдает совсем непонятный результат, представленный на Рисунке 11.

```
[['emp']]
[['emp']]
[['emp']]
[['emp']]
[['c', '_']]
[['c', '_']]
[['c', '_']]
[['c', '_']]
[['c', '_']]
```

Рисунок 11. Результат 2

Если исправить верифицируемую программу, переставив присвоение значения объекту *c* после его создания, то утечек памяти не обнаружится, как и должно быть. Отсюда можно сделать вывод, что разработанная программа имеет недостаток – она теряет работоспособность на кодах с ошибками.

5.3 Тест 3

В качестве третьей верифицируемой программы используется код с бо́льшим количеством объектов, представленный на Рисунке 12.

```
Testing a = new Testing();
Testing b = new Testing();
b.value = val;
Testing x = new Testing();
Testing c = new Testing();
c.value = val;
a.value = val;
c = null;
Testing y = new Testing();
y.value = val;
a = null;
Testing z = new Testing();
y = null;
Testing d = new Testing();
Testing f = new Testing();
d.value = val;
f.value = val;
d = null;
Testing g = new Testing();
g.value = val;
g = null;
```

Рисунок 12. Программа 3

Результат выполнения программы по поиску утечек памяти показан на Рисунке 13. Как видно, программа правильно определила все утечки памяти в коде. Причем, обнаруженные утечки накапливаются кумулятивно в процессе выполнения программы. Результат на Рисунке 13 это правильно отражает: появившись впервые, утечка продолжает отображаться на каждом шаге до конца, в то время как новые утечки присоединяются к старым.

```
[['emp']]
[['emp']]
[['emp']]
[['b', 'val']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_']]
[['b', 'val'], ['x', '_'], ['z', '_']]
[['b', 'val'], ['x', '_'], ['z', '_']]
[['b', 'val'], ['x', '_'], ['z', '_']]
[['b', 'val'], ['x', '_'], ['z', '_']]
[['b', 'val'], ['x', '_'], ['z', '_']]
[['b', 'val'], ['x', '_'], ['z', '_'], ['f', 'val']]
[['b', 'val'], ['x', '_'], ['z', '_'], ['f', 'val']]
[['b', 'val'], ['x', '_'], ['z', '_'], ['f', 'val']]
[['b', 'val'], ['x', '_'], ['z', '_'], ['f', 'val']]
```

Рисунок 13. Результат 3

Глава 6

Заключение

В работе был рассмотрен один из видов верификации программного обеспечения – статический анализ кода. При этом основной темой было применение такого логического метода, как абдукция. В работе было дано описание данного понятия, а также смежных с ним понятий – фреймового вывода и би-абдукции. Также были обозначены преимущества применения данных логических методов в верификации программного обеспечения. В частности, было отмечено, что би-абдукция позволяет проводить композиционный анализ формы, смысл понятия которого также раскрывается в данной работе.

Практическая часть была посвящена рассмотрению использования би-абдукции при поиске утечек памяти в Java-программах. В результате выполнения работы был реализован один из существующих алгоритмов. При разработке программы использовался язык Python 3. Целью создания программы, реализующей поиск утечек памяти, было желание наглядно продемонстрировать на простом и понятном примере практическое применение логических методов в верификации программного обеспечения.

В данной работе было дополнительно проведено и описано тестирование созданной программы. При этом использовались вручную сгенерированные упрощенные линейные Java-коды. В целом тестирование показало удовлетворительные результаты, однако было обнаружено, что разработанная программа неверно идентифицирует утечки памяти, если в верифицируемом коде есть структурные или логические ошибки.

В качестве дальнейших исследований предполагается модифицировать алгоритм, использованный в данной работе, так, чтобы перед поиском утечек памяти проводилась проверка на наличие других ошибок в коде.

Список источников

[1] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, Hongseok Yang *Compositional Shape Analysis by means of Bi-Abduction* 2011.

[2] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez *Moving Fast with Software Verification* 2015.

[3] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher *A Survey of Automated Techniques for Formal Software Verification* 2008.

[4] Cristiano Calcagno and Dino Distefano *Infer: An Automatic Program Verifier for Memory Safety of C Programs* 2011.

[5] Dino Distefano and Ivana Filipovic *Memory Leaks Detection in Java by Bi-Abductive Inference* 2010.