Санкт-Петербургский государственный университет

Программная инженерия

Бушев Вячеслав Валериевич

Инструмент для поиска шаблонов изменений в коде на языке Python

Выпускная квалификационная работа бакалавра

Научный руководитель:
доцент кафедры СП СПбГУ, к.т.н. Т. А. Брыксин

Консультант:
программист-исследователь
ООО "Интеллиджей Лабс"
Е.О. Богомолов

Рецензент:
доцент кафедры КТС СПбГУ, к.ф-м.н. С. В. Погожев

Санкт-Петербург
2020

SAINT PETERSBURG STATE UNIVERSITY

Software Engineering

Viacheslav Bushev

A tool for mining change patterns in Python code

Bachelor's Thesis

Scientific supervisor:
Associate professor, Ph.D. Timofey Bryksin

Scientific advisor:
Research programmer
"IntelliJ Labs" Co. Ltd
Egor Bogomolov

Reviewer:
Associate professor, Ph.D. Sergey Pogozhev

Saint Petersburg
2020

Оглавление

Введение **4**

1. Обзор предметной области **7**

1.1. Анализ изменений в коде . . . 7

1.1.1. ChangeDistiller . . . 7

1.1.2. PyCT . . . 8

1.1.3. Lase . . . 8

1.1.4. GumTree . . . 9

1.1.5. CPatMiner . . . 9

1.2. Архитектура проекта CPatMiner . . . 13

1.3. Инструменты для построения графов . . . 14

1.4. Сбор изменений в коде . . . 15

1.4.1. Источники изменений . . . 15

1.4.2. Методы сбора изменений с GitHub . . . 16

2. Архитектура решения и его реализация **18**

2.1. Архитектура решения . . . 18

2.2. Сбор изменений . . . 19

2.3. Построение графов из исходного кода . . . 21

2.3.1. Граф потока управления и потока данных . . . 22

2.3.2. Транзитивное замыкание . . . 23

2.4. Граф изменений . . . 26

2.5. Поиск шаблонов . . . 28

2.6. Вывод результатов . . . 30

3. Апробация **34**

3.1. Инструменты и данные для проведения опроса . . . 34

3.2. Опрос и его результаты . . . 34

Заключение **36**

Список литературы **37**

Введение

На сегодняшний день существует множество компаний, разрабатывающих программное обеспечение. Программисты, которые там работают, производят колоссальные объемы кода. Программное обеспечение постоянно меняется, поскольку изменяется его окружение и требования к нему, а люди придумывают новые способы использования программных продуктов. Например, проект Visual Studio Code уже содержит более 65 000 коммитов[1], а IntelliJ IDEA Community — более 280 000 коммитов[2]. Код и его изменения являются источником данных для различных исследований, связанных, в том числе, с анализом эволюции программного обеспечения [23, 21].

Недавние исследования показывают, что изменения в коде периодически повторяются [21, 7, 17]. Это наблюдение является отправной точкой для данной работы. Из него мы можем сделать вывод: разработчики не только испытывают одинаковые проблемы, но и решают их похожими способами. Повторяющиеся изменения кода образуют *шаблоны изменений*, анализ которых поможет улучшить средства автодополнения и проверки программного кода в *IDE* (Integrated development environment, интегрированная среда разработки) и, возможно, другие инструменты, используемые программистами [11, 3]. Кроме того, найденные шаблоны могут быть источником данных для проведения эмпирических исследований на различные темы, например, о том, как пишут код программисты из разных сфер деятельности.

Для того чтобы найти осмысленные семантические[3] шаблоны изменений в коде, необходимо решить ряд нетривиальных задач. Требуется собрать изменения, выделить среди них те, которые представляют ценность, сгруппировать однотипные образцы и вывести результат в простом для понимания виде.

[1] Visual Studio Code на GitHub: `https://github.com/microsoft/vscode` (Дата обращения: 01.06.2020).

[2] IntelliJ Idea Community на GitHub: `https://github.com/JetBrains/intellij-community` (Дата обращения: 01.06.2020).

[3] Изменения кода считаются семантическими, если они влияют на поведение программы. Например, переименование переменной — синтаксическое, а не семантическое изменение.

На текущий момент существует множество инструментов по поиску шаблонов изменений в коде на языке Java [10, 16]. Аналогичных инструментов для языка Python практически нет, однако и его анализ является актуальной задачей, в том числе, для улучшения IDE[4].

Таким образом, данная работа направлена на создание инструмента для поиска шаблонов изменений в коде на языке Python. Изменения кода собираются из коммитов в репозиториях на GitHub[5], поскольку эта платформа является крупнейшим хостингом кода, содержащим более 100 миллионов репозиториев [22]. В рамках одного коммита может быть множество не связанных между собой изменений, кроме того, нужно учитывать, что их синтаксическая схожесть не всегда влечет семантическую эквивалентность, поэтому, для того чтобы выделить осмысленные шаблоны, код представляется в виде специального графа *fgPDG* (fine-grained program dependence graph, детализированный граф зависимостей программы). Поиск шаблонов учитывает особенности связей между его узлами и включает в себя проверку графов изменений на изоморфизм, которая реализована с помощью эвристического алгоритма. Описанный выше подход был впервые представлен в работе Nguyen et al. [10] и реализован в инструменте CPatMiner[6] для языка Java. Для апробации разработанного в данной работе инструмента были собраны шаблоны изменений кода за апрель 2020 года из 5 642 репозиториев на GitHub. Чтобы оценить качество и осмысленность найденных в коде шаблонов изменений, был составлен и проведен опрос среди авторов этих изменений.

[4]Python — один из самых популярных языков программирования. Например, в мае 2020 он занял первую строчку в рейтинге PYPL, URL: `https://pypl.github.io/PYPL.html` (Дата обращения: 01.06.2020).

[5]Главная страница GitHub: `http://github.com/` (Дата обращения: 01.06.2020).

[6]Проект с открытым исходным кодом, доступен на GitHub: `https://github.com/nguyenhoan/CPatMiner` (Дата обращения: 01.06.2020).

Постановка цели и задач

Цель работы — разработать инструмент для поиска шаблонов изменений в коде на языке Python.

Для достижения цели были поставлены следующие задачи:

- провести обзор предметной области, изучить существующие инструменты и методы сбора и анализа изменений в коде;
- разработать инструмент для поиска шаблонов изменений в коде на языке Python;
- провести апробацию разработанного инструмента.

1. Обзор предметной области

В данной главе рассмотрены инструменты и методы сбора и анализа изменений в коде, их преимущества и недостатки, а также обоснован выбор некоторых из них для достижения поставленной цели.

1.1. Анализ изменений в коде

Вначале был изучен ряд существующих инструментов и подходов для анализа изменений в коде. Большинство из них нацелено на язык Java и использует методы, основанные на операциях над абстрактным синтаксическим деревом программы (abstract syntax tree, *AST*). Цели исследований изменений кода сильно варьируются. Одни направлены на идентификацию и классификацию типов изменений [5, 23], другие — на поиск шаблонов исправления ошибок [19], третьи работают с повторами кода и ранжируют их, выделяя при этом те, которые наиболее важно изменять и отслеживать [18]. Есть и такие работы, которые, как текущая, направлены на поиск ранее неизвестных шаблонов изменений в коде [16, 17, 10]. Данный раздел представляет из себя краткий обзор подобных работ.

1.1.1. ChangeDistiller

Одним из известных инструментов для извлечения и классификации изменений является ChangeDistiller [5]. Данный инструмент использует алгоритм на основе сравнения AST исходного кода, который называется Change Distilling. Алгоритм производит сопоставление узлов деревьев и находит минимальный *сценарий редактирования* (edit script, последовательность операций вставки, удаления, перемещения, изменения меток узлов AST, которая производит преобразование одного AST в другое), с помощью которого извлекается информация о том, какого типа изменения были сделаны. Инструмент обнаруживает синтаксические изменения, среди которых, например, “добавление параметра в метод”, “изменение выражения”, “изменение возвращаемо-

го значения” и другие[7]. В основе алгоритма лежат результаты статьи Chawathe et al. [4], причем Change Distilling в 45% случаев находит сценарий редактирования, более приближенный к минимальному [5]. ChangeDistiller написан на Java, однако имеет интерфейсы, которые позволяют модифицировать его для обработки других языков.

1.1.2. PyCT

Результаты предыдущей работы используются в ряде других исследований. Например, в статье Lin et al. [23] были проанализированы типы изменений кода, которые делают программисты на языке Python. С этой целью разработан инструмент PyCT, использующий адаптированный алгоритм Change Distilling. Авторы реализовали поддержку анализа кода на Python, что позволило определить частоту различных изменений на целевом языке. Оказалось, что изменения функций и выражений — самые популярные, а самые редко встречающиеся — изменения структуры циклов. В статье также сравнивался код популярных библиотек и фреймворков из различных областей программирования (веб, нейролингвистическое программирование, большие данные и другие). Полученные результаты показывают, что среди рассмотренных проектов наблюдается схожее распределение частот различных типов изменений.

1.1.3. Lase

Инструмент Lase осуществляет поиск шаблонов изменений в коде на Java в виде контекстно-независимого сценария редактирования AST, а затем применяет операции из сценария для автоматического изменения повторяющегося кода [16]. Для построения сценария редактирования из файлов Lase использует алгоритм Change Distilling. Общие операции изменения AST находятся с помощью алгоритма поис-

[7]Полный список поддерживаемых изменений для Java можно найти по ссылке: `https://bitbucket.org/sealuzh/tools-changedistiller/src/feee5be3724a3eabfb7c415554cb26f2258a65f4/src/main/java/ch/uzh/ifi/seal/changedistiller/model/classifiers/ChangeType.java#lines-51` (Дата обращения: 01.06.2020).

ка наибольшей общей подпоследовательности LCEOS [12], причем полная эквивалентность операций не требуется, их равенство определяется приближенно с использованием алгоритма сравнения биграмм [2]. Контекстная независимость обеспечивается заменой общих для операций изменения кода типов, методов и имен переменных.

1.1.4. GumTree

Другой широко используемый алгоритм поиска минимального сценария редактирования AST — GumTree [8]. Одноименный инструмент работает с универсальным форматом, в который может быть преобразован код на любом языке. Данный инструмент легко расширяем за счет продуманной модульной архитектуры — практически каждый модуль можно заменить. Кроме того, алгоритм GumTree более точно и кратко представляет изменения, связанные с перемещением кода, по сравнению с Change Distilling [8]. GumTree используется, например, инструментом Coming [15], выполняющим поиск предварительно описанных по определенным правилам шаблонов, а также в работе по поиску шаблонов исправления ошибок [13] (в обеих статьях были проанализированы только репозитории на языке Java).

1.1.5. CPatMiner

Наиболее близкая к данной работе статья была опубликована в конце 2019 года [10]. В ней описывается поиск семантических, ранее неизвестных шаблонов изменений в коде на языке Java. Авторы статьи показывают, что алгоритмы поиска шаблонов, основанные на представлениях кода, учитывающих только синтаксические преобразования, приводят к неправильной классификации и объединению разных по смыслу фрагментов изменений. Например, если изменения представляют из себя одинаковые операции над AST, это не гарантирует их семантическую эквивалентность, поэтому такие изменения нельзя считать равносильными.

Для исправления недостатков существующих методов было введено

новое представление кода в виде графа FGPDG (fine-grained program dependence graph, детализированный граф зависимостей программы), отражающее зависимости языковых конструкций друг от друга. FGPDG — направленный ациклический граф, который имеет три типа узлов, представляющих:

- *данные* (например, переменные, литералы);
- *операции над ними* (присваивание, операции сложения, вычитания, логические операции и другие);
- *управляющие конструкции* (*if*, *for* и остальные).

Узлы соединяются ребрами, которые могут быть одного из двух типов. *Управляющие ребра* показывают, что выполнение потомка контролируется предком. *Ребра потока данных* отображают поток данных во время исполнения программы. Последний тип имеет различные метки, чтобы конкретизировать зависимости:

- *def* используется для того, чтобы показать определение одного узла данными из другого;
- *recv* связывает объект, вызывающий метод, с самим методом;
- *ref* используется для представления ссылок на определенные ранее переменные;
- *para* для параметров;
- *cond* для условных конструкций;
- *qual* для квалифицированных имен[8].

Пример графа изменений для кода на Листинге 1 изображен на Рисунке 1. Здесь и далее на рисунках графов узлы данных соответствуют эллипсам, управляющие конструкции — ромбам, а операции — прямоугольникам.

[8]Квалифицированные имена для классов и интерфейсов в JAVA представляют из себя имя пакета и следующее за ним имя класса/интерфейса, разделенные точкой (например, math.FractionCalculator).

```java
int a = 10, b = 4;
print(a + b);
```

```java
int a = 10, b = 4;
if (a + b > 20) {
    return;
}
print(a + b);
```

Листинг 1: Изменения Java-кода, соответствующие графу на Рисунке 1.

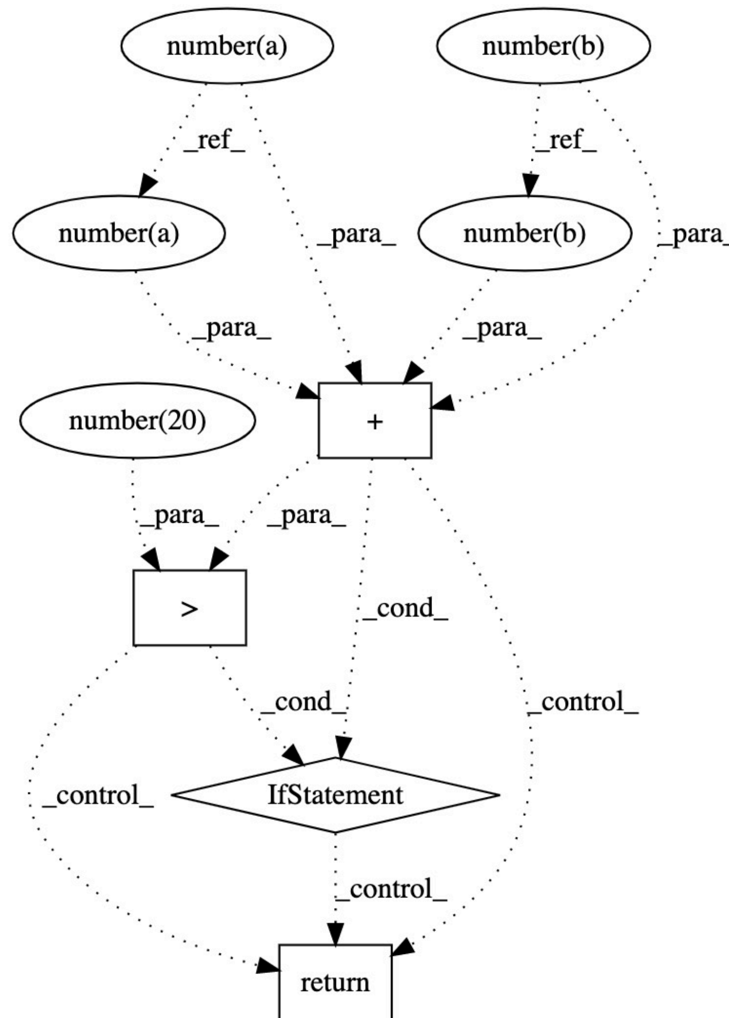

Рис. 1: Граф изменений для кода на Листинге 1.

Изменения извлекаются из модифицированных функций с помощью специального алгоритма поиска сценария редактирования [6], который основывается на структурной похожести поддеревьев AST. Результат представляется в виде двух fgPDG, построенных для исходного кода

до и после. Те узлы, которые не изменились, но оказывают влияние на поведение программы, тоже содержатся в итоговом графе. Например, узлы операций считаются измененными, если изменились входящие в них данные, поскольку в таком случае меняется поток данных в программе. Ребро *map* при этом сопоставляет соответствующие друг другу конструкции до и после изменений.

Кроме нового представления изменений кода авторы данной статьи предложили свой алгоритм поиска семантических шаблонов среди них. Изменения называются *семантическим шаблоном*, если соответствующий им граф fgPDG повторяется по крайней мере x раз (порог задается пользователем). Все повторения при этом считаются образцами шаблона. Алгоритм сначала выбирает шаблоны минимального размера, а затем они рекурсивно расширяются с помощью смежных узлов с учетом их особенностей и связей. Для текущего шаблона **p** размером **k** расширение размером **k+1** генерируется следующим образом: управляющие узлы и узлы данных добавляются, если из них есть исходящее в **p** ребро, а узлы операций подходят в том случае, если есть ребра, соединяющие их с **p** в обоих направлениях. В целях оптимизации на следующем этапе алгоритма рассматриваются не все возможные расширения, а только наиболее часто встречающиеся. Для этого расширения группируются: в одну группу попадают те из них, для которых графы изменений изоморфны. Если расширения встречаются недостаточно часто, они отбрасываются. Для проверки графов на изоморфизм используется эвристический алгоритм, основанный на векторизации графов и хэшировании [1].

На основе описанного подхода был разработан новый инструмент — CPatMiner, — который извлекает изменения кода из заданных репозиториев, строит для них соответствующие графы, а затем производит поиск ранее неизвестных шаблонов изменений с помощью нового алгоритма. CPatMiner находит более осмысленные шаблоны изменений по сравнению со старыми методами, основанными на синтаксических представлениях кода [16]. Кроме того, если нет информации о порядке изменений, CPatMiner показывает лучшие результаты по сравнению

с алгоритмом, представленным в работе [17] по поиску шаблонов с помощью отслеживания изменений кода непосредственно из IDE [10]

Таким образом, в текущей работе решено использовать методы, лежащие в основе CPatMiner, для представления кода и поиска шаблонов изменений в коде на языке Python. Для поиска различий между измененными файлами используется инструмент GumTree в силу его модульности, универсальности, простоты использования и эффективности извлечения сценария редактирования по сравнению с аналогичными инструментами [8].

1.2. Архитектура проекта CPatMiner

Поскольку данная работа основывается на результатах исследования с открытым исходным кодом, одна из доступных возможностей — переиспользование его фрагментов. В данном разделе рассматривается архитектура CPatMiner и приводятся аргументы, почему его переиспользование невозможно.

CPatMiner написан на языке Java и разделен на два проекта. Цель первого — пройти по заданному списку репозиториев, извлечь оттуда изменения, построить для них соответствующие графы fgPDG и сохранить результат в файл в виде Java-объекта. Код построения графа не может быть переиспользован или дополнен, поскольку он рассчитан на язык Java, а в рамках данной работы необходимо построить fgPDG для Python, имеющего абсолютно другую формальную грамматику и, в том числе, новые языковые конструкции. Кроме того, чтобы построить граф для исходного кода на Python, необходимо совершить обход его AST, что значительно проще сделать с помощью стандартного Python-модуля pyast, чем пытаться работать с исходным кодом другого языка на Java.

Второй проект использует выходные данные первого, которые представляют из себя Java-объекты, чтобы найти в них шаблоны. Это означает, что для совместимости нужно следовать такому же формату, что невозможно в силу различия построения графов для разных языков.

Как следствие, переиспользование проекта CPatMiner для языка Python невозможно. Поэтому было решено написать свой инструмент на Python, реализующий построение графов для изменений в коде и поиск шаблонов среди них.

1.3. Инструменты для построения графов

Перед созданием собственного инструмента построения графов fgPDG, отображающих потоки данных и потоки управления в программе, были рассмотрены несколько существующих библиотек по построению аналогичных графов.

Одна из них — библиотека с открытым исходным кодом от IBM под названием pyflowgraph[9]. Граф, который она строит, основан на динамическом анализе программы, поэтому, чтобы получить результат, исходный код нужно будет выполнить. Как следствие, проект не может быть использован, поскольку подобный подход потенциально отсекает значительную часть изменений в коде, блокирующих основной поток исполнения программы, так как они не могут быть обработаны в автоматическом режиме.

Для построения графа также рассматривался проект staticfg[10]. Он строит граф потока управления программы, но основывается на статическом анализе её исходного кода. Результат работы staticfg сильно отличается от fgPDG как своими узлами, так и связями между ними. Тем не менее, идея рекурсивного алгоритма обхода AST, реализованная в staticfg подходит для использования в данной работе.

Таким образом, переиспользование существующих библиотек и инструментов для построения графов fgPDG в полной мере невозможно, однако подход рекурсивного обхода AST для построения графа был позаимствован из библиотеки staticfg.

[9] pyflowgraph на GitHub: `https://github.com/IBM/pyflowgraph` (Дата обращения: 01.06.2020).
[10] staticfg на GitHub: `https://github.com/coetaur0/staticfg` (Дата обращения: 01.06.2020).

1.4. Сбор изменений в коде

Прежде чем строить графы и искать шаблоны изменений в коде, изменения необходимо собрать. CPatMiner для этого использует Java-библиотеку JGIT, которая позволяет обойти локально сохраненные репозитории, проанализировать их коммиты и найти модифицированные файлы для последующего извлечения кода. В данном разделе рассмотрены альтернативные методы, в частности, применимые для языка Python.

1.4.1. Источники изменений

Платформа GitHub является крупнейшим хостингом кода [14], основанным на системе контроля версий, содержащим более 100 миллионов репозиториев [22], среди которых множество публичных[11]. Таким образом, она может быть использована в качестве источника для сбора изменений в коде на языке Python. Существуют также и другие популярные платформы, например, BitBucket[12].

Альтернативный подход к сбору изменений представлен в статье Negara et al [17]. Её авторы собирают изменения прямо из IDE. Кодовая база GitHub намного больше, и использовать её значительно удобнее, поскольку это не требует привлечения сторонних разработчиков, готовых установить инструменты для отслеживания изменений кода в режиме реального времени в свою среду разработки. Кроме того, формат изменений, собранных непосредственно из IDE, отличается от формата кода с GitHub, что усложняет дальнейшую разработку из-за возникающего расхождения с подходом CPatMiner.

Таким образом, основным источником данных в текущей работе является платформа GitHub.

[11] Поиск по публичным репозиториям на GitHub: `https://github.com/search?q=is:public` (Дата обращения: 01.06.2020).

[12] Главная страница проекта BitBucket: `https://bitbucket.org/product/` (Дата обращения: 01.06.2020).

1.4.2. Методы сбора изменений с GitHub

Для того чтобы собрать изменения с GitHub, можно составить список публичных репозиториев, для каждого из них — список коммитов, затем для каждого коммита сформировать набор измененных файлов. Изменения кода будут получены в результате сравнения старого файла с новым файлом в рамках одного коммита. Помимо изменений полезно зафиксировать информацию о том, кто является их автором, в каком файле и когда они произошли. В данном разделе рассмотрены различные методы сбора перечисленных данных.

Простой метод получения необходимой информации — использование GitHub API[13]. Это решение неэффективно, поскольку оно предполагает отправку множества запросов, что ненадежно и требует большого количества времени. Кроме того, количество запросов для одного пользователя имеет ограничение — 5000/час[14].

Помимо официального API существуют также сторонние ресурсы, которые предоставляют информацию с GitHub. Например, проект GHTorrent [9] содержит архивы с данными из репозиториев с 2013 года[15], а GH Archive — с декабря 2011 года[16]. Данные размещаются на соответствующих сайтах проектов и в хранилище Google BigQuery, которое позволяет осуществлять удобную выборку с помощью SQL-подобного языка, а затем просматривать и скачивать результат. Хранилище имеет определенные ограничения, например, любой SQL-запрос должен выполняться не более 6 часов[17]. Несмотря на преимущества по сравнению с предыдущим методом, использование архивов не подходит для данной работы, поскольку они сохраняют только общую информацию о репозиториях, а не содержимое файлов с исходным кодом.

---

[13] Документация API GitHub доступна по ссылке: `https://developer.github.com/v3` (Дата обращения: 01.06.2020).

[14] Сведения об ограничениях опубликованы по ссылке: `https://developer.github.com/v3/#rate-limiting` (Дата обращения: 01.06.2020).

[15] Информация из раздела "Downloads" на сайте проекта GHTorrent, URL: `https://ghtorrent.org/downloads.html` (Дата обращения: 01.06.2020).

[16] Информация с главной страницы GH Archive, URL: `https://www.gharchive.org/` (Дата обращения: 01.06.2020).

[17] Полный список ограничений представлен на сайте, URL: `https://cloud.google.com/bigquery/quotas` (Дата обращения: 01.06.2020).

Для того чтобы не делать множество сетевых операций, можно заранее скачать репозитории и использовать инструменты для их обхода. Самая известная Python-библиотека для обхода git-репозиториев — GitPython — предоставляет абстракции всех git-объектов. Библиотека позволяет получить всю необходимую информацию, поэтому может быть рассмотрена в качестве инструмента для сбора изменений.

GitPython имеет низкоуровневые абстракции, из-за которых, например, усложняется реализация извлечения исходного кода файлов до коммита и после него. Чтобы устранить этот недостаток, в работе используется фреймворк PyDriller, который является оберткой над GitPython [20]. Его основное преимущество — более высокоуровневые абстракции, с которыми проще и быстрее работать.

Таким образом, для извлечения изменений и информации о них решено использовать фреймворк PyDriller, поскольку он значительно ускоряет разработку по сравнению с GitPython взамен на незначительные потери производительности [20].

2. Архитектура решения и его реализация

В данной главе рассматривается разработанное решение для поиска шаблонов изменений в коде на языке Python и описывается его архитектура и реализация.

2.1. Архитектура решения

Разработанное решение представляет из себя настраиваемую Python-библиотеку[18], основными компонентами которой являются следующие модули:

- *vcs* — осуществляет обход репозиториев и извлечение исходного кода из коммитов;
- *pyflowgraph* — выполняет построение графов для исходного кода;
- *changegraph* — строит граф изменений для исходного кода до и после них;
- *patterns* — осуществляет поиск шаблонов среди построенных с помощью CHANGEGRAPH графов изменений.

Библиотека имеет интерфейс командной строки (*CLI*, Command-line interface), предоставляющий возможность запуска инструмента в одном из следующих режимов:

- *pfg* — построение графа потока управления и потока данных для конкретного файла с исходным кодом на языке Python;
- *cg* — построение графа изменений для двух заданных файлов;
- *collect-cgs* — сбор изменений из локально сохраненных репозиториев, построение и сохранение графов изменений;
- *patterns* — поиск шаблонов среди построенных графов изменений.

[18]Исходный код опубликован на GitHub, URL: `https://github.com/JetBrains-Research/code-change-miner` (Дата обращения: 01.06.2020).

Все настройки библиотеки и используемых в данной работе скриптов задаются в json-файле *conf/settings.json*. Среди них находятся и те, которые управляют процессом поиска шаблонов изменений в коде. Например, с помощью *patterns_min_frequency* можно указать, сколько раз должен повторяться граф изменений, чтобы он считался шаблоном, а с помощью *traverse_file_max_line_count* — задать ограничение на количество строк в рассматриваемых файлах (если количество строк превосходит указанное в настройке число, то такой файл не рассматривается).

На Рисунке 2 представлена диаграмма последовательности, демонстрирующая процесс поиска шаблонов изменений в коде с помощью разработанного инструмента (созданные в рамках данной работы модули имеют зеленый фон, а внешние библиотеки и фреймворки — серый). После предварительного задания настроек необходимо запустить библиотеку в режиме COLLECT-CGS. Первым при этом запускается модуль VCS, который извлекает измененные файлы из репозиториев с помощью фреймворка PyDriller. Затем для каждой пары файлов до и после изменений CHANGEGRAPH строит граф изменений. Для этого версии исходного кода рассматриваются отдельно, и вызывается PYFLOWGRAPH, который генерирует графы потока управления и потока данных. С использованием GumTree из них извлекаются изменившиеся узлы, на основе которых и строится требуемый граф изменений. Построенные графы сохраняются на жестком диске. После завершения предыдущего этапа необходимо повторно запустить библиотеку, но уже в режиме PATTERNS. Тогда модуль PATTERNS выполнит загрузку графов, поиск шаблонов и сохранит результат в виде html-файлов.

2.2. Сбор изменений

Для обхода репозиториев по коммитам модуль VCS использует класс RepositoryMining из фреймворка PyDriller. RepositoryMining извлекает исходный код измененных файлов и некоторую информацию о коммитах, в том числе дату и хэш коммита, имя автора, его электрон-

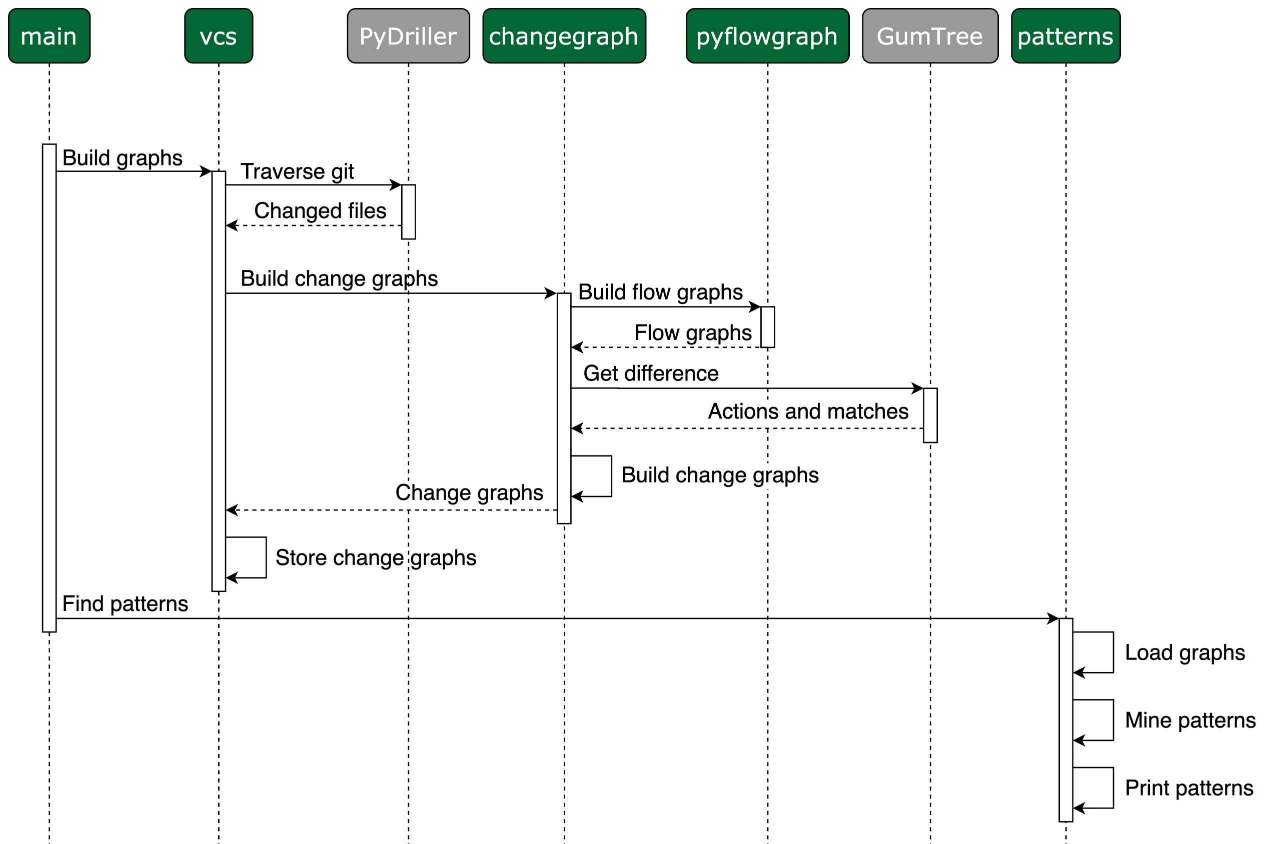

Рис. 2: Диаграмма работы инструмента.

ный адрес и другие данные.

Для каждого файла до и после изменений VCS выполняет сопоставление изменившихся методов по их квалифицированному имени. *Квалифицированное имя* в данном контексте включает в себя имена метода и всех классов, в которых он содержится[19]. Благодаря этому различаются и не сопоставляются методы, находящиеся внутри модуля, с методами внутри класса, методы из разных классов и другие. Кроме того, учитывается порядок: два метода с одинаковыми названиями из одной и той же области видимости будут сопоставлены с двумя другими методами с такими же названиями в порядке очередности их определения. Такой подход будет неправильно реагировать, например, на перемещения методов, однако он является удовлетворительным для данной работы, поскольку подавляющее большинство изменений имеет другой характер.

[19]Для метода *run* внутри класса *Dummy*, вложенного в класс *Compiler*, полным именем является *Compiler.Dummy.run*.

После сопоставления методов VCS вызывает функцию из модуля CHANGEGRAPH для построения соответствующих им графов изменений. Результаты сохраняются в файлы с заданной в настройках периодичностью. Для этого используется стандартная библиотека PICKLE[20].

Чтобы ускорить сбор изменений и построение графов, для каждого репозитория коммиты рассматриваются параллельно. Это обеспечивается благодаря стандартной библиотеке MULTIPROCESSING[21], которая позволяет использовать все доступные на устройстве процессорные ядра для проведения необходимых вычислений. MULTIPROCESSING не используется на уровне обхода всех репозиториев, поскольку полная загрузка информации даже из одного репозитория требует большого количества оперативной памяти.

В процессе распараллеливания сбора изменений была обнаружена утечка памяти, которая происходила в библиотеке PyDriller при передаче экземпляров класса Commit в другие процессы. Для обмена данными MULTIPROCESSING использует PICKLE, что стало причиной утечки, поэтому было принято решение предварительно преобразовывать объекты коммитов в словари. О найденной ошибке сообщено разработчикам фреймворка, и они уже работают над её исправлением[22].

2.3. Построение графов из исходного кода

На этапе построения графов изменений модуль CHANGEGRAPH сначала рассматривает каждую версию исходного кода по отдельности. Для них строится граф потока управления и потока данных (далее — *граф потоков*) с помощью другого модуля — PYFLOWGRAPH. В данном разделе описывается его функциональность.

[20]Библиотека PICKLE, URL: https://docs.python.org/3/library/pickle.html (Дата обращения: 01.06.2020).

[21]Библиотека MULTIPROCESSING, URL: https://docs.python.org/3/library/multiprocessing.html (Дата обращения: 01.06.2020).

[22]Отчёт об ошибке в PyDriller, URL: https://github.com/ishepard/pydriller/issues/102 (Дата обращения: 01.06.2020).

2.3.1. Граф потока управления и потока данных

Построение графа потоков осуществляется с использованием стандартного модуля AST[23], который преобразовывает текстовое представление кода в его абстрактное синтаксическое дерево. На следующем шаге *ASTVisitor* выполняет рекурсивный обход полученного дерева и строит граф, исходя из того, какие объекты AST при этом встречаются. Например, при посещении экземпляра класса NUM (из модуля AST) ASTVISITOR создает соответствующий ему узел данных — *DataNode* с типом *literal*. Аналогично создаются другие узлы, в том числе для операций (*OperationNode*) и управляющих конструкций (*ControlNode*). Зависимости между выражениями отображаются с помощью ребер. В частности, если рассматривается операция, то сначала строятся графы для каждого аргумента, а затем создается граф, объединяющий их: добавляется узел самой операции и входящие в неё ребра *para*, исходящие из последнего созданного узла в каждом графе.

В процессе построения графа строится контекст видимых переменных и обновляется узел, соответствующий последней управляющей конструкции, напрямую влияющей на выполнение рассматриваемой операции. Изначально им является временный узел *START* (который в дальнейшем, при построении графа изменений, удаляется). Затем, к примеру, при посещении оператора IF таким узлом становится соответствующий ему CONTROLNODE. Контекст переменных заменяется, например, в случае рассмотрения lambda-функции, чтобы избежать неверного добавления новых переменных в старую область видимости.

PYTHON имеет множество высокоуровневых конструкций, облегчающих разработку, но усложняющих его синтаксический анализ. Например, операция присваивания может задавать сразу нескольких переменных или, с помощью оператора “звездочка” (*), присваивать n объектов $n + c$ переменным. Реализация всех возможных конструкций PYTHON очень затратна, поэтому была обеспечена поддержка только самых основных из них. В Таблице 1 указан статус поддержки элементов AST

[23] Абстрактная грамматика и описание возможностей модуля AST для PYTHON 3.8, URL: `https://docs.python.org/3.8/library/ast.html` (Дата обращения: 01.06.2020).

Поддержка	Типы объектов AST
Полная	BinOp, UnaryOp, Lambda, Return, Continue, Break, Pass, Dict, Set, Await, Compare, JoinedStr, Starred, Attribute, Subscript, Name, List, Tuple
Частичная	AugAssign, AnnAssign, Assign[1], Raise[2], Expr, Try[3], While[4], For[4], Call[5], FormattedValue[6]
Отсутствует	Delete, Global, NonLocal, Assert, With, AsyncWith, Import, ImportFrom, IfExp, ListComp, SetComp, DictComp, GeneratorExp, Yield, YieldFrom

[1] Некоторые типы объектов не могут быть в левой части оператора присваивания, среди них Subscript, Attribute и другие.
[2] Аргументы не учитываются.
[3] Не учитываются ветки else и finally.
[4] Не учитывается ветка else.
[5] Поддерживается, если функция — Attribute или Name.
[6] Поддерживается без параметра conversion.

Таблица 1: Информация о поддерживаемых объектах AST.

(из модуля AST).

2.3.2. Транзитивное замыкание

Изменения, относящиеся к одному и тому же шаблону, могут находится на различных, не обязательно идущих подряд строках кода. Чтобы учесть такие ситуации для зависимых друг от друга узлов, связанных через промежуточный узел, строится дополнительное ребро. Полученный после построения таких ребер граф будем называть *транзитивным замыканием* исходного графа. На Рисунке 3 приведен пример транзитивного замыкания графа для кода на Листинге 1. Узел операции *assignment* имеет исходящее ребро *para* в узел вызова функции *print*, поскольку переменная *b*, которая является непосредственным аргументом вызова функции, определяется соответствующей операцией присваивания. Здесь и далее на графах Python-кода узлы имеют метки, соответствующие языковым выражениям. При этом в треугольных скобках уточняется тип узла, а в квадратных — его номер в порядке обхода AST.

```python
    b = 8
    print(b)
```

Листинг 2: Код на Python для графа на Рисунке 3.

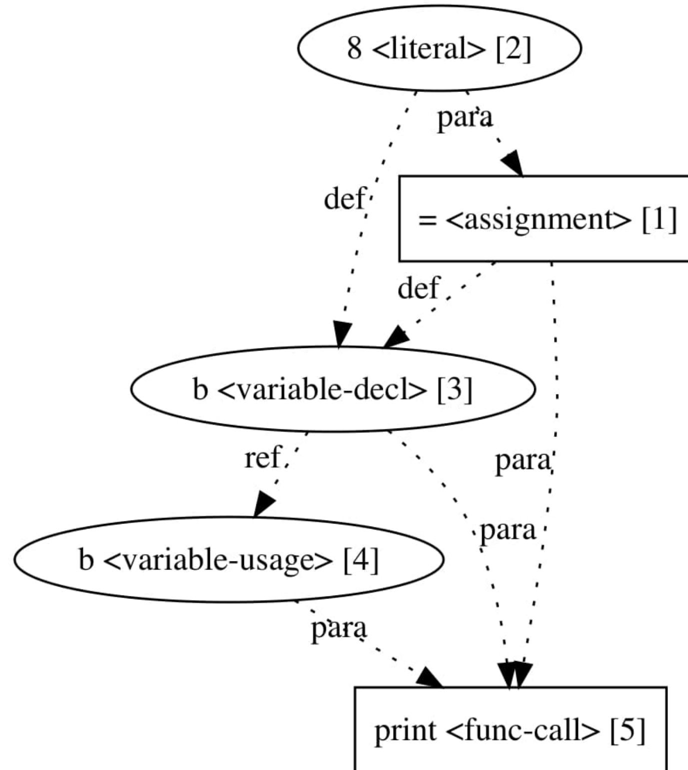

Рис. 3: Транзитивное замыкание графа для кода на Листинге 1.

В процессе изучения замыкания в CPatMiner было обнаружено, что иногда инструмент неправильно строит его для управляющих конструкций. Например, в случае, когда оператор *if* содержит только одну условную ветвь, в которой встречается команда *return*, выполнение *if* влияет на код после него, и CPatMiner корректно показывает это с помощью дополнительных *control* ребер. В случае же, когда *if* имеет две условные ветви, первая из которых не содержит оператора *return*, а вторая содержит, CPatMiner этого не учитывает и не достраивает дополнительных ребер, тем самым упуская зависимость от выполнения *if* для следующих за условным выражением конструкций (см. Листинг 2 и Рисунок 4). Кроме того, при построении графа fgPDG в ряде случа-

ев инструмент неверно обозначает тип ветви для управляющих ребер, выбирая между истинной и ложной.

```java
int a = 10, b = 4;
if (a > 0) {
    b = 8;
} else {
    return;
}

call(b);
```

Листинг 3: Код на Java для графа на Рисунке 4.

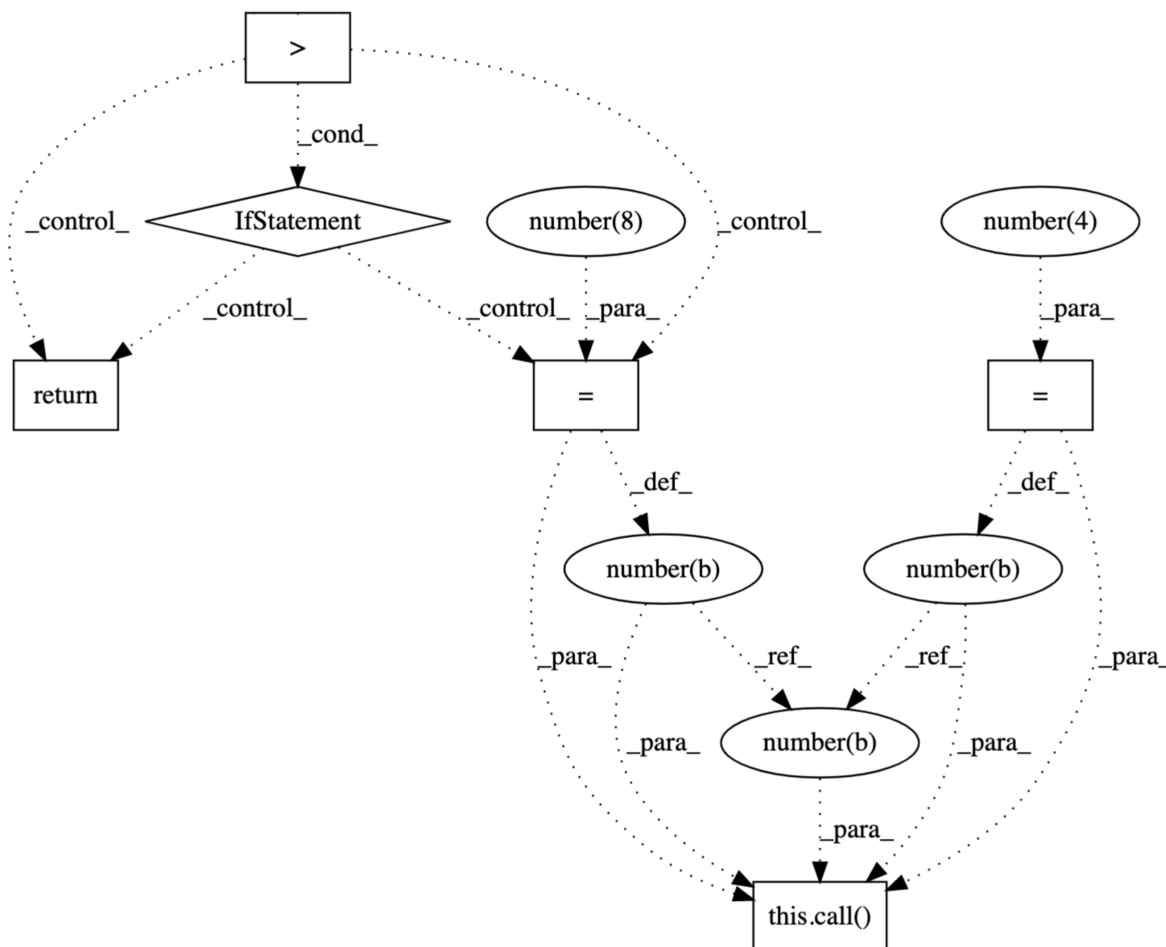

Рис. 4: Фрагмент fgPDG для кода на Листинге 2.

Перечисленные недостатки были исправлены в данной работе. Для этого построение замыкания разделяется на три этапа. Сначала строится замыкание потока данных. Алгоритм его построения основан на обходе графа в глубину. На следующем этапе отдельно строится замыкание, которое включает в себя только управляющие ребра. Алго-

ритм аналогичен предыдущему. На последнем этапе строится замыкание управляющими ребрами с учетом промежуточных ребер данных. Алгоритм также основан на обходе графа в глубину, однако теперь при посещении каждого узла k выполняется следующая логика:

- для k рассматривается каждый *входящий узел* n (узел n будем называть входящим в узел k, если n имеет хотя бы одно исходящее в k ребро), и если он ещё не был посещен, то сначала алгоритм вызывает построение замыкания для него;

- для n рассматривается входящий в него узел m, а затем выполняется поиск ближайшего к m узла c, соответствующего управляющей конструкции;

- если m соответствует управляющей конструкции, то $m = c$;

- в ином случае для определения ближайшего узла алгоритм перебирает все исходящие из m управляющие узлы и выбирает тот, которому соответствует минимальный номер по порядку обхода объектов AST с учетом существования пути из выбранного узла до k;

- после того, как ближайший узел найден, добавляется новое *control* ребро из c в k, соответствующее по типу и метке ребру из c в n.

В результате получается транзитивное замыкание со всеми необходимыми управляющими ребрами (см. Листинг 3 и Рисунок 5).

2.4. Граф изменений

После построения графов потоков с помощью модуля PYFLOWGRAPH, CHANGEGRAPH выполняет построение графа изменений. Для того чтобы выявить изменившиеся узлы, используется инструмент GumTree. В данном разделе подробно описывается работа модуля CHANGEGRAPH.

Как было отмечено ранее, модуль CHANGEGRAPH сначала рассматривает исходный код до и после изменений по отдельности и строит

```python
a = 10
b = 4
if a > 0:
    b = 8
else:
    return

print(b)
```

Листинг 4: Код на Python для графа на Рисунке 5.

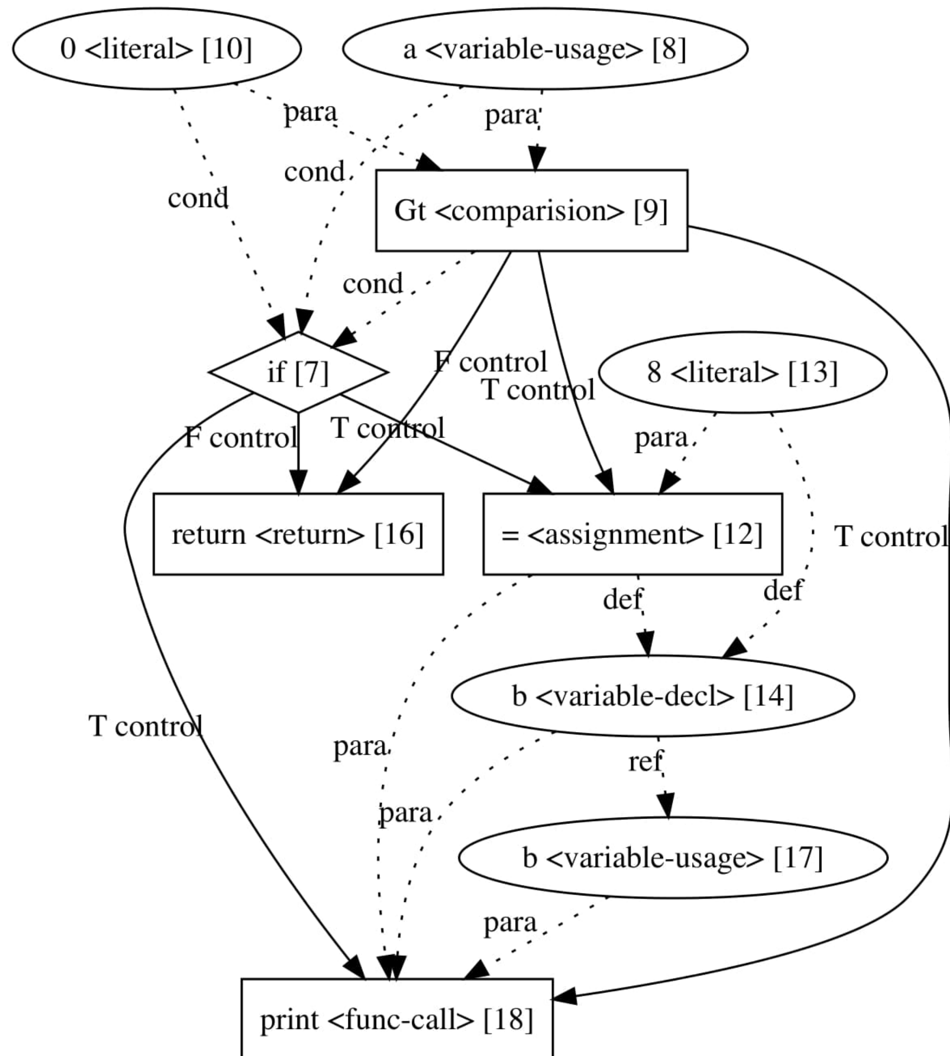

Рис. 5: Фрагмент графа потоков для кода на Листинге 3.

для него графы потоков. Затем применяется инструмент GumTree. Во внешнем процессе вызывается команда *gumtree parse*, которая преобразует каждую версию кода в соответствующие деревья (формат уни-

фицирован для всех языков программирования). Команда выводит результат в формате JSON. Инструмент, реализованный в данной работе, считывает результат и создает объекты деревьев. После этого вызывается *gumtree jsondiff*. Команда возвращает в качестве результата идентификаторы сопоставленных до и после изменений узлов, а также минимальный сценарий редактирования. С помощью обхода в глубину эта информация записывается в соответствующие узлы созданных деревьев. На следующем этапе они сопоставляются с узлами построенных в самом начале графов потоков. В итоговом графе изменений содержатся те узлы, которые были отмечены измененными, а также все ребра, которые их связывают. Кроме того, поскольку GumTree находит только синтаксические изменения, некоторые дополнительные узлы также добавляются в граф, исходя из зависимостей между выражениями в коде. В частности, если изменяется узел "использование переменной", то в граф добавляется узел "объявление переменной", а если изменяются аргументы вызова функции, то добавляется и сам вызов функции.

2.5. Поиск шаблонов

Поиск шаблонов выполняет модуль PATTERNS. Алгоритм поиска перенесен из инструмента CPatMiner и адаптирован для языка Python. В данном разделе представлено краткое описание реализованной адаптации и подробно рассмотрены некоторые задачи, которые были решены в процессе реализации.

После того, как инструмент собрал графы изменений, его необходимо запустить в режиме поиска шаблонов (*patterns*). При этом сначала выполняется загрузка построенных графов, а затем осуществляется построение основы будущих изменений — пар узлов до и после них. Пары, представляющие изоморфные подграфы, группируются. Для таких групп в дальнейшем строятся и рекурсивно расширяются шаблоны. На Рисунке 6 изображены основные классы модуля PATTERNS. Они выполняют следующие функции:

- Miner осуществляет поиск шаблонов и вывод результатов;

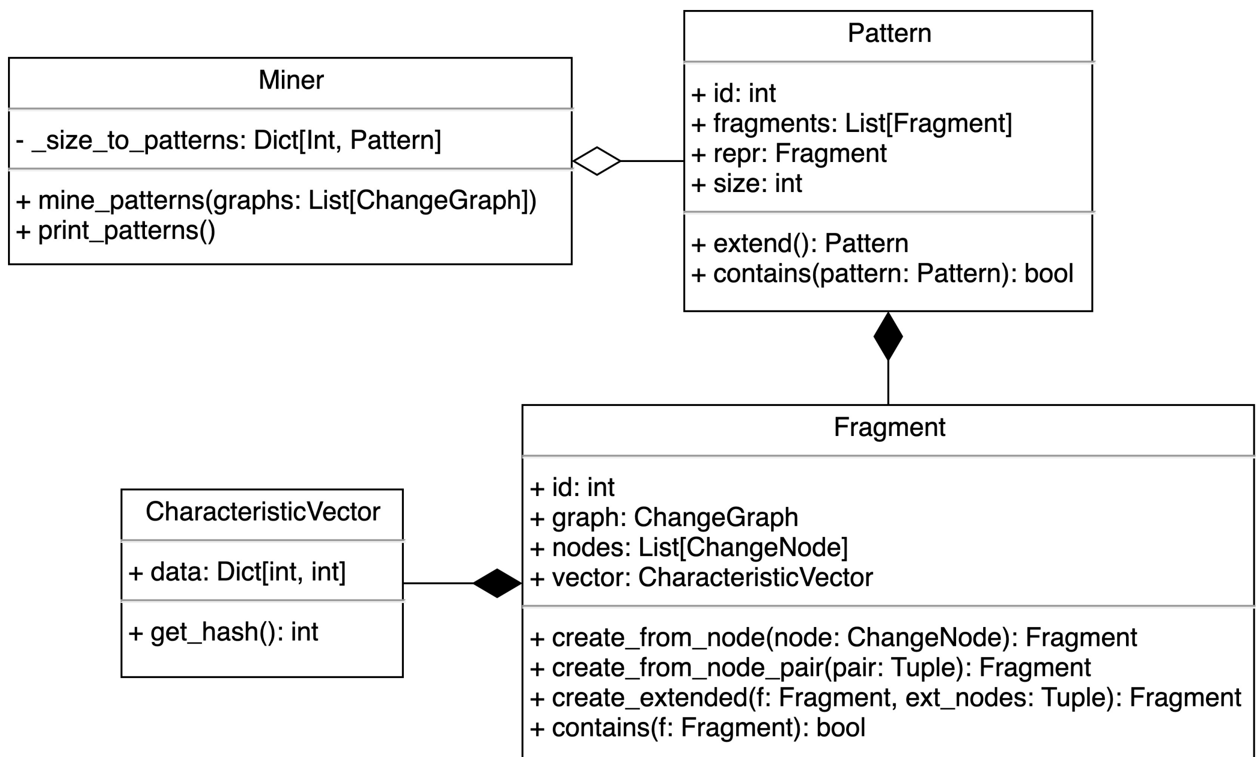

Рис. 6: Диаграмма основных классов модуля PATTERNS.

- PATTERN используется для представления шаблонов;
- FRAGMENT представляет образцы изменений кода;
- CHARACTERISTICVECTOR является характеристическим вектором фрагментов. Его экземпляры содержат информацию о частоте повторения путей в конкретном графе. С помощью вектора генерируется хэш, используемый для разбиения фрагментов на группы согласно статье [1].

Итак, первая итерация построения шаблонов тривиальна. Каждый шаблон состоит из фрагментов, созданных для рассмотренных ранее групп из пар узлов. На следующих итерациях выполняется рекурсивное расширение уже построенных шаблонов. Для них осуществляется поиск всех подходящих узлов и строятся соответствующие им фрагменты, которые группируются по хэшу характеристического вектора. Из групп выбирается та, которая встречается наибольшее количество раз. Если её размер превышает минимальный порог (заданный пользователем), то элементы группы становятся образцами нового шаблона,

и процесс расширения продолжается.

Для хранения хэша характеристического вектора фрагментов на Java инструмент CPatMiner использует целочисленный тип данных *int*. Однако в Python при работе с числами не происходит переполнения, поэтому для увеличения производительности при подсчете изоморфных групп оно было реализовано искусственно.

Каждая группа фрагментов состоит из уникальных объектов за счет проверки фрагментов на эквивалентность. В CPatMiner при сравнении фрагментов была допущена ошибка из-за использования типа *ArrayList*: фрагменты с разным порядком добавления одних и тех же узлов считались различными. Кроме того, сами узлы сравнивались по ссылке. В данной работе при сравнении используется тип данных *set*, а сами узлы сравниваются по идентификатору, который присваивается им еще в процессе построения графов. Таким образом удается исправить ошибки CPatMiner и исключить одинаковые образцы из результатов. При сравнении также используется следующая оптимизация: если суммы идентификаторов, входящих в соответствующие фрагменты, не совпадают, то дальнейших действий не производится, поскольку фрагменты точно не равны[24].

Крупные шаблоны изменений могут содержать в себе шаблоны меньших размеров. Для того чтобы такие результаты не отображались, перед их выводом выполняется фильтрация. Фрагменты-представители шаблонов сравниваются, и если больший содержит все узлы меньшего, то меньший шаблон удаляется. Узлы фрагментов все так же сравниваются по идентификаторам, а не по ссылкам как в CPatMiner.

2.6. Вывод результатов

Модуль patterns также осуществляет вывод результатов. Для этого после поиска шаблонов вызывается метод *print_patterns*. В этом разделе рассматривается данный метод, процесс и формат вывода шаблонов, а также реализованная для этого функциональность.

[24]Сумма идентификаторов узлов предварительно подсчитывается на этапе построения фрагментов,

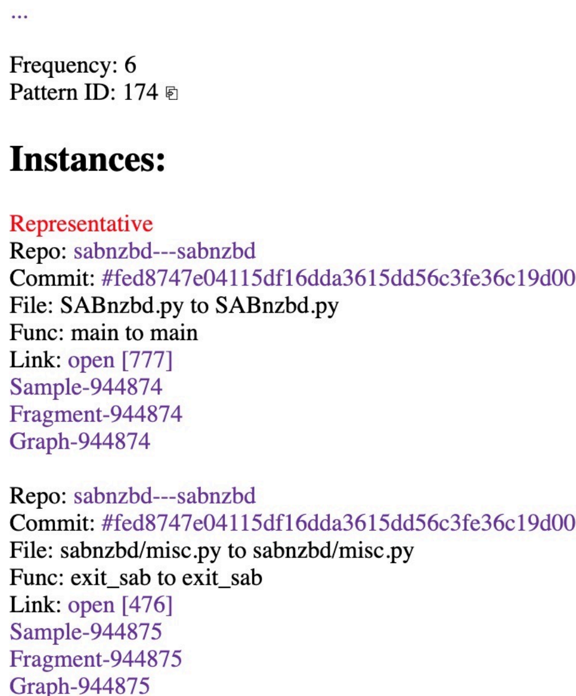

Рис. 7: Пример вывода информации о шаблоне (details.html).

Метод *print_patterns* создает соответствующие шаблонам директории по заданному в настройках пути. Они создаются сначала для каждой группы шаблонов одного размера, а затем для самих шаблонов. В соответствующей шаблону директории находятся несколько основных файлов — *details.html* и *sample-*.html*[25]. Первый содержит информацию о том, сколько раз повторялся шаблон, в каких репозиториях, коммитах, файлах и функциях он встречался. На Рисунке 7 изображена часть такого файла для шаблона изменений, найденного в репозитории SABNZBD[26]. Ссылка, начинающаяся со слова *Sample*, ведет в *sample-*.html*. Пример такого файла представлен на Рисунке 8. В нем содержатся фрагменты кода до изменений и после них, а также некоторая информация о том, где они были сделаны. Изменения, которые относятся к рассматриваемому шаблону, выделяются непрозрачным текстом и жирным шрифтом. При этом на каждую изменившуюся строчку кода можно нажать, чтобы открыть данное изменение на GitHub. При наведении такие строчки также меняют фон на желтый (как на строч-

а не при сравнении.

[25]Символ “*” здесь и далее обозначает комбинацию из латинских букв и цифр.

[26]Ссылка на репозиторий: `https://github.com/sabnzbd/sabnzbd` (Дата обращения: 01.06.2020).

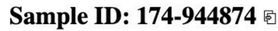

Рис. 8: Пример вывода изменений в шаблоне (sample.html).

ке 1 504 на Рисунке 8). Идентификатор рассматриваемого фрагмента можно скопировать, нажав на кнопку рядом с ним, а код — развернуть и свернуть при необходимости. Описанная функциональность реализована с помощью JavaScript и CSS. Для подсветки синтаксиса языка Python используется библиотека highlight.js[27].

Для того чтобы сопоставить узлы графов с выделенными фрагментами кода, при построении графов использовалась библиотека asttokens, которая добавляет объектам AST больше информации об их положении в исходном коде. В частности, asttokens добавляет каждому объекту AST поля с его первым и последним строковым токеном. Для каждого токена можно узнать позицию символов, соответствующих его началу и концу. В узлы графа записываются *синтакси-*

[27]Библиотека доступна по ссылке: `https://highlightjs.org/` (Дата обращения: 01.06.2020).

ческие интервалы от начала первого токена до конца последнего. При этом учитывается, что некоторые объекты AST соответствуют бóльшим синтаксическим конструкциям, чем нужно. Например, операция сложения — это не только знак “+”, но и её операнды. Использование таких данных привело бы к неверному выделению изменившегося кода, поэтому сначала подынтервалы операндов исключаются из общего интервала, и только потом он записывается в узел операции. На этапе вывода результатов синтаксические интервалы узлов объединяются (если возможно), а затем символы, соответствующие каждому интервалу, заключаются в специальные тэги.

Кроме html-файлов экспортируются pdf-изображения фрагментов и графов изменений. Вывод данных — настраиваемый. В частности, если включить опцию *patterns_output_details*, то для каждого фрагмента будет создан дополнительный json-файл с информацией, касающейся конкретного изменения (контакты и имя автора, имена файлов, методов и др.).

3. Апробация

Для оценки качества разработанного инструмента были собраны шаблоны изменений за апрель 2020 года в ряде проектов с открытым исходным кодом на GitHub, и был проведен опрос среди авторов этих изменений. В данной главе рассматривается содержательная часть опроса и используемые для его проведения средства и инструменты.

3.1. Инструменты и данные для проведения опроса

Изменения кода собраны из репозиториев разного масштаба и популярности. Для скачивания репозиториев с GitHub был написан Python-скрипт[28]. Сначала были загружены все репозитории с количеством звезд от 10 000. Затем с разным шагом (по количеству звезд) загружались менее популярные проекты вплоть до репозиториев с 50 звездами. В результате удалось скачать 5 642 репозитория, среди которых осуществлялся поиск шаблонов изменений. Инструмент обнаружил более 1 800 шаблонов, авторами которых являются более 358 разработчиков.

Опрос был составлен с помощью сервиса Google Forms. Найденные шаблоны были опубликованы онлайн с помощью GitHub Pages[29]. С целью опроса 358 разработчиков были отправлены электронные письма на те почтовые адреса, которые привязаны к коммитам, соответствующим сделанным изменениям. Каждый автор опрашивался только один раз, даже если его электронный адрес встречался в нескольких коммитах или шаблонах сразу. Для рассылки сообщений был написан Python-скрипт. Письма были отправлены только тем разработчикам, которые не скрыли свой адрес электронной почты.

3.2. Опрос и его результаты

Опрос был составлен на английском языке, чтобы охватить бо́льшую аудиторию, и содержал следующие вопросы.

[28] Все скрипты, которые относятся к тем или иным исследованиям в работе, могут быть найдены в директории *research/tools* в репозитории проекта на GitHub.

[29] Главная страница Github Pages: `https://pages.github.com/` (Дата обращения: 01.06.2020).

- Если возможно, кратко опишите выделенные изменения, и почему вы их сделали[30].

- Можете ли вы дать им название[31]?

- Хотели бы вы, чтобы эти изменения были автоматизированы[32]?

Из 358 разработчиков 46 прошли опрос. Все респонденты смогли ответить на первый вопрос и описать выделенные изменения. Кроме того, как показано на Рисунке 9, более половины авторов смогли дать этим изменениям названия, а 43.5% (20 человек из 46) хотели бы, чтобы соответствующие изменения были автоматизированы.

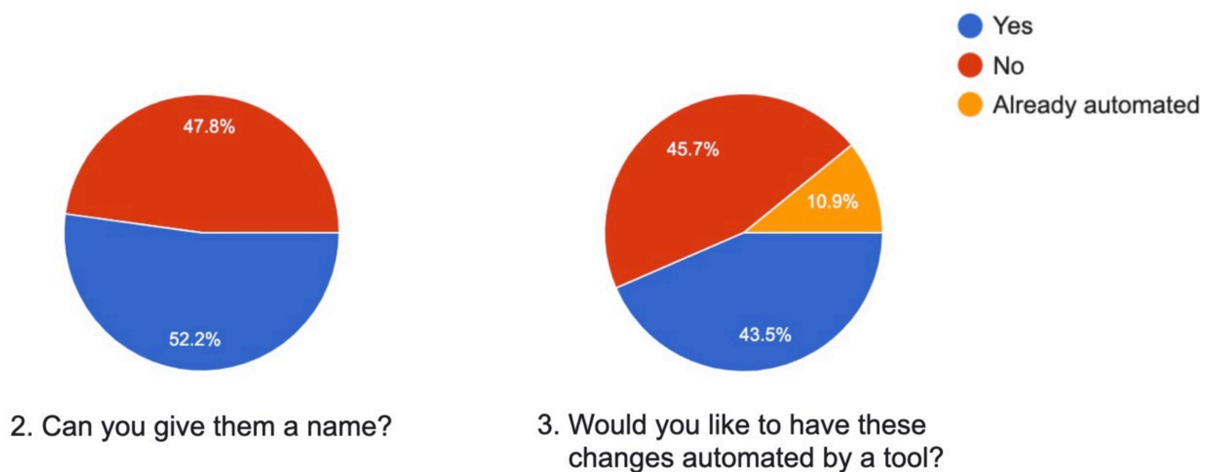

Рис. 9: Результаты проведенного опроса.

Таким образом, разработанный инструмент находит осмысленные шаблоны изменений, авторы которых хотели бы их автоматизировать.

[30]Briefly describe highlighted changes and why you made them, if possible. For example, “comparing a variable with None to avoid further AttributeError on accessing its fields”.

[31]Can you give them a name? For example, “None Check”, “Adding a condition branch”, “Closing resources”.

[32]Would you like to have these changes automated by a tool?

Заключение

В ходе данной работы получены следующие результаты.

- Проведен обзор инструментов и методов сбора и анализа изменений в коде.
- На основе нового подхода по поиску шаблонов изменений с помощью представления кода в виде графов fgPDG разработан инструмент[33] для поиска шаблонов в коде на языке Python, который осуществляет:
  - обход репозиториев и сбор изменений (с помощью PyDriller);
  - построение графов потока управления и потока данных для исходного кода;
  - построение графа изменений (с использованием GumTree);
  - поиск шаблонов на основе эвристического алгоритма группировки изоморфных графов.
- Найдено более 1 800 шаблонов изменений, собранных из 5 642 публичных репозиториев среди коммитов за апрель 2020 года. Проведен опрос 358 авторов изменений, который подтверждает осмысленность и качество собранных инструментом шаблонов.

Из полученных результатов можно сделать вывод о том, что разработанный инструмент может быть успешно применен для поиска семантических шаблонов изменений в коде на языке Python с целью дальнейшего анализа и автоматизации часто повторяющихся изменений. Инструмент может быть улучшен главным образом с помощью добавления поддержки построения графов потоков для неподдерживаемых на данный момент объектов AST. Это позволит анализировать и находить больше шаблонов изменений.

[33] Исходный код инструмента опубликован на платформе GitHub и доступен по ссылке: `https://github.com/JetBrains-Research/code-change-miner` (Дата обращения: 01.06.2020).

Список литературы

[1] Accurate and efficient structural characteristic feature extraction for clone detection / Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham et al. // International Conference on Fundamental Approaches to Software Engineering / Springer. — 2009. — P. 440–455.

[2] Adamson George W, Boreham Jillian. The use of an association measure based on character structure to identify semantically related pairs of words and document titles // Information storage and retrieval. — 1974. — Vol. 10, no. 7-8. — P. 253–260.

[3] Bruch Marcel, Monperrus Martin, Mezini Mira. Learning from examples to improve code completion systems // Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. — 2009. — P. 213–222.

[4] Change detection in hierarchically structured information / Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, Jennifer Widom // Acm Sigmod Record. — 1996. — Vol. 25, no. 2. — P. 493–504.

[5] Change distilling: Tree differencing for fine-grained source code change extraction / Beat Fluri, Michael Wuersch, Martin PInzger, Harald Gall // IEEE Transactions on software engineering. — 2007. — Vol. 33, no. 11. — P. 725–743.

[6] Clone management for evolving software / Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham et al. // IEEE transactions on software engineering. — 2011. — Vol. 38, no. 5. — P. 1008–1026.

[7] FSE 2014: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. — New York, NY, USA : ACM, 2014.

[8] Fine-grained and accurate source code differencing / Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc et al. // Proceedings of the 29th ACM/IEEE international conference on Automated software engineering / ACM. — 2014. — P. 313–324.

[9] Gousios Georgios. The GHTorrent dataset and tool suite // Proceedings of the 10th Working Conference on Mining Software Repositories. — MSR '13. — Piscataway, NJ, USA : IEEE Press, 2013. — P. 233–236. — Access mode: `http://dl.acm.org/citation.cfm?id=2487085.2487132`.

[10] Graph-based Mining of In-the-wild, Fine-grained, Semantic Code Change Patterns / Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig et al. // Proceedings of the 41st International Conference on Software Engineering. — ICSE '19. — Piscataway, NJ, USA : IEEE Press, 2019. — P. 819–830.

[11] Graph-based pattern-oriented, context-sensitive source code completion / Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen et al. // 2012 34th International Conference on Software Engineering (ICSE) / IEEE. — 2012. — P. 69–79.

[12] Hunt James W, Szymanski Thomas G. A fast algorithm for computing longest common subsequences // Communications of the ACM. — 1977. — Vol. 20, no. 5. — P. 350–353.

[13] Islam Md Rakibul, Zibran Minhaz F. How bugs are fixed: exposing bug-fix patterns with edits and nesting levels // Proceedings of the 35th Annual ACM Symposium on Applied Computing. — 2020. — P. 1523–1531.

[14] Lean GHTorrent: GitHub data on demand / Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, Andy Zaidman // Proceedings of the 11th working conference on mining software repositories. — 2014. — P. 384–387.

[15] Martinez Matias, Monperrus Martin. Coming: a tool for mining change pattern instances from git commits // Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings / IEEE Press. — 2019. — P. 79–82.

[16] Meng Na, Kim Miryung, McKinley Kathryn S. LASE: locating and applying systematic edits by learning from examples // 2013 35th International Conference on Software Engineering (ICSE) / IEEE. — 2013. — P. 502–511.

[17] Mining fine-grained code changes to detect unknown change patterns / Stas Negara, Mihai Codoban, Danny Dig, Ralph E Johnson // Proceedings of the 36th International Conference on Software Engineering. — 2014. — P. 803–813.

[18] Mondal Manishankar, Roy Chanchal K, Schneider Kevin A. SPCP-Miner: A tool for mining code clones that are important for refactoring or tracking // 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER) / IEEE. — 2015. — P. 484–488.

[19] Osman Haidar, Lungu Mircea, Nierstrasz Oscar. Mining frequent bug-fix code changes // 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE) / IEEE. — 2014. — P. 343–347.

[20] Spadini Davide, Aniche Maurício, Bacchelli Alberto. PyDriller: Python framework for mining software repositories // Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018. — New York, New York, USA : ACM Press, 2018. — P. 908–911. — Access mode: http://dl.acm.org/citation.cfm?doid=3236024.3264598.

[21] A Study of Repetitiveness of Code Changes in Software Evolution / Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen et al. //

Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. — ASE'13. — Piscataway, NJ, USA : IEEE Press, 2013. — P. 180–190.

[22] Warner Jason. Thank you for 100 million repositories. — `https://github.blog/2018-11-08-100m-repos/`. — 2018. — Accessed: 01.06.2020.

[23] An empirical study on the characteristics of Python fine-grained source code change types / Wei Lin, Zhifei Chen, Wanwangying Ma et al. // 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME) / IEEE. — 2016. — P. 188–199.