Санкт-Петербургский государственный университет

Кафедра системного программирования

Смирнов Олег Евгеньевич

Автоматическое изменение кода на Python в соответствии в графовыми шаблонами

Отчёт по производственной практике

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

Консультант:
программист-исследователь
ООО "Интеллиджей Лабс" Лобанов А. В.

Санкт-Петербург
2021

Оглавление

Введение

Написание кода у современных программистов стало занимать гораздо меньше времени, чем раньше, и связано это не только с появлением большого числа готовых библиотек и модулей, но и с уже привычным, постоянным использованием интегрированных сред разработки (Integrated Development Environment, IDE) [34, 40]. Они предоставляют разработчику широкий выбор встроенных инструментов, таких как автодополнение кода [5], подсветка синтаксических ошибок [26], или даже рекомендация рефакторингов [37]. Многие из этих механизмов реализуются в IDE с помощью средств *статического анализа кода*, т.е. не используют информацию, полученную в процессе исполнения, а опираются лишь на внутреннюю структуру программы — абстрактное синтаксическое дерево (Abstract Syntax Tree, AST).

Инструменты статического анализа также используют и для *автоматического исправления ошибок* (automated program repair) [20]. Эта задача, в свою очередь, включает в себя как локализацию потенциальных ошибок или уязвимостей в коде, так и их автоматическое исправление. Исследованию возможных подходов к решению этой проблемы было посвящено немало научных работ в последние годы [32, 19, 33, 9, 13].

Чтобы понять, какие исправления в программе действительно было бы полезно уметь делать автоматически, исследователи часто анализируют историю изменений кода в открытых репозиториях. Так, в работе Nguyen et al. [21] о поиске шаблонов исправлений в коде на Java был предложен подход с использованием графовых представлений, которые сохраняли дополнительную информацию об изменении кода с помощью ребер с зависимостями между вершинами AST. Позже их алгоритм для сбора часто встречаемых изменений был адаптирован для языка Python студентом СПбГУ Вячеславом Бушевым в его дипломной работе [43].

В данной работе предлагается использовать новый подход к автоматическому исправлению кода: локализовать и изменять фрагменты программы в соответствии с собранными до этого шаблонами исправлений кода на Python используя их графовую структуру. Это позволяет

нам ожидать, что найденные уязвимости будут более осмысленными, приближенными к реальному опыту разработчиков, и потенциально более сложными, т.к. их графовые шаблоны будут также содержать дополнительную информацию о внутренних зависимостях между элементами кода.

Постановка задачи

Целью данной работы является разработка инструмента для автоматического исправления кода на Python в соответствии с графовыми шаблонами изменений в IDE PyCharm.

Для достижения данной цели в рамках работы были поставлены следующие задачи:

1. Проанализировать существующие подходы к поиску и автоматическому исправлению ошибок и уязвимостей в коде.
2. Проанализировать достоинства и недостатки использования графовых шаблонов изменений для исправления кода.
3. Реализовать алгоритм локализации графовых шаблонов в коде.
4. Реализовать механизм исправления локализованного фрагмента в соответствии с шаблоном.
5. Представить указанную функциональность в виде плагина к IDE PyCharm.
6. Провести апробацию плагина с участием реальных разработчиков.

1. Обзор

Любое изменение кода, которое можно совершать автоматически, может как изменять общую логику программы, исправляя некоторую уязвимость, так и не нарушать её, выполняя некоторое эквивалентное преобразование кода — например, рефакторинг или форматирование кода в соответствии с некоторыми правилами. Таким образом, всё зависит от определения «ошибки» в контексте задачи её поиска в целевом коде разработчика. В данной работе намеренно не проводится чёткая грань между указанными понятиями, а решается более широкая задача: поиск фрагментов кода, подходящих под некоторый шаблон изменения, и их последующее преобразование в соответствии с этим шаблоном. Далее будут описаны некоторые идеи и механизмы, используемые в статьях по автоматическому восстановлению программ от ошибок, а также предпосылки использования подхода с графовыми паттернами изменений.

1.1. Локализация «ошибок» в коде

Задачу поиска фрагментов кода, подходящих под определение «ошибки», исследователи решают по-разному. Это можно делать строго, например, требуя точного совпадения фрагмента целевого кода с подготовленным шаблоном «ошибки», или с помощью техник символьного вычисления проверять, удовлетворяет ли целевой фрагмент кода набору заданных ограничений. Однако также часто авторы работ предлагают использовать и статистические методы, например, вероятностные модели и машинное обучение [32].

1.1.1. Использование шаблонов «ошибок»

Авторы работы LASE [31] не ставят своей целью исправлять ошибки, но как раз используют идею поиска шаблонов как часть подхода к автоматическому изменению программ, сначала собирая и обобщая частые изменения в коде, а потом локализуя и применяя их. Для на-

бора изменений в разных фрагментах кода авторы ищут наибольшую общую подпоследовательность действий по редактированию AST, называя это систематическим изменением (systematic edit). Для каждого такого изменения конструируется *контекст* — AST кода *до* изменения с абстрактными именами переменных в вершинах. После этого на этапе локализации он сопоставляется с целевым AST с помощью алгоритма поиска наибольшего общего встроенного поддерева [28], который основан на применении методов динамического программирования. Однако этот способ локализации эффективен только тогда, когда «ошибку» можно гарантированно представить в виде *одного* поддерева AST.

Если же рассматривать код как последовательность токенов, так же представляя и искомый шаблон, поиск «ошибки» может сводиться к поиску подстроки (или подпоследовательности) из токенов. Такой подход использовали авторы инструмента DevReplay [11]. Они собирали набор паттернов возможных ошибок из истории изменений кода в репозитории за определенный промежуток времени, а затем искали их в коде разработчика с помощью регулярных выражений. Каждый паттерн представлялся в виде набора последовательных токенов из кода *до* изменения, по нему строилось регулярное выражение, в котором все имена переменных и идентификаторы заменялись на абстрактные, после чего происходил поиск подходящих под описание мест с возможной «ошибкой». Если выявленная «ошибка» получала подтверждение разработчика, фрагмент кода с ней заменялся на код из части *после* исходного шаблона изменения. Такой подход более универсален с точки зрения переиспользования для разных языков программирования, т.к. не зависит от представления кода в виде AST, которое специфично для языка, но совсем не очевидно, как с его помощью можно описывать сложные семантические шаблоны изменений, имеющие связи и зависимости между элементами сразу в нескольких разных местах в коде.

1.1.2. Символьные вычисления

Еще один формальный подход к поиску уязвимостей в коде основан на идее символьного вычисления [38]: если представлять код как набор

преобразований над набором переменных, то можно оценивать результаты его исполнения статически, оперируя не конкретными константами и выражениями, а абстрактными «символами» из некоторого домена. Таким образом, появляется возможность осуществлять формальные проверки кода на выполнение некоторого набора ограничений, который должен быть задан в спецификации к рассматриваемой программе. И если в коде появляется некоторая уязвимость, нарушающая хотя бы одно из заданных ограничений, её можно локализовать и исправить, применив некоторый автоматически сгенерированный «патч» [29, 25]. Однако такой способ не подходит для задачи эквивалентного преобразования кода в соответствии с шаблоном, ведь ни одно из ограничений не будет нарушаться ни до, ни после изменения. В случае же шаблонов, меняющих поведение программы, трансформация паттерна изменения кода в набор формальных ограничений для его проверки представляет собой отдельную интересную задачу для будущих исследований. Однако стоит также отметить, что символьные вычисления зачастую занимают довольно много времени, а также подвержены проблеме комбинаторного взрыва количества путей исполнения (path explosion problem), что затрудняет их использование в качестве статического анализатора в on-line режиме (например, в редакторе кода в IDE).

1.1.3. Вероятностное предсказание уязвимостей

Во множестве работ по автоматическому восстановлению программ от ошибок локализация рассматривается лишь как одна из частей более комплексного подхода, в котором значимое место выделяется, например, для сбора шаблонов изменений кода и поиску среди них тех, которые бы исправляли ошибки или в принципе совершали некоторые осмысленные действия. Так, в работах исследовательской группы Люксембургского университета [17, 3, 39] для поиска мест с паттерном ошибки использовался сторонний фреймворк для динамического тестирования [22], а предсказание возможных ошибочных операторов в коде осуществлялось с помощью оценки метрики Отиаи [1]. В оригинале такой подход называется *spectrum-based fault localization*, и его

суть заключается в следующем: если разбить код на условные блоки и для каждого блока составить вектор значений, где i-тый элемент говорит, прошла ли программа i-тый тест, то вектор блока с ошибкой будет наиболее всего «похож» на предподсчитанный вектор для кода, в котором гарантированно есть данная ошибка. Под схожестью векторов в данном случае понимается оценка метрики сходства, например, при помощи коэффициента Жаккара или уже указанного коэффициента Отиаи. Однако данная методика жестко ограничена тем, что для каждого шаблона «ошибки» должен быть подготовлен релевантный набор тестов, при этом выявляющий именно эту (и только эту) «ошибку».

Последнее время в сфере локализации уязвимостей (как, в принципе, и в любой другой сфере), исследователи часто пытаются применять методы машинного обучения [18, 23, 2]. В большинстве случаев такие работы концентрируются на каком-либо конкретном типе уязвимостей, например, гонках данных (в инструменте DeepRace [10]), или ошибках, связанных с потенциально некорректным порядком аргументов функции или неверным использованием операторов (DeepBugs [36, 44]). Первый из указанных подходов опирается на использование свёрточных нейронных сетей [15], а во втором обучают классификатор сжатых представлений контекстов указанных типов ошибок на основе механизма word2vec [12, 14]. К сожалению, для обучения подобных типов моделей необходим огромный датасет, а также не всегда очевидно, как адаптировать такой подход для локализации строго заданных шаблонов «ошибок».

1.2. Автоматическое исправление фрагментов кода

После того, как потенциальный шаблон «ошибки» локализован, хотелось бы уметь исправлять код в соответствии с шаблоном изменения, т.е. изменять данный участок кода, при этом учитывая окружающий контекст и не нарушая общей логики программы в случае шаблона с эквивалентным преобразованием. В данном разделе будут кратко рассмотрены некоторые техники машинного обучения, которые приме-

няются для внесения изменений в код, а также классические способы применения набора изменений на основе скриптов редактирования.

1.2.1. Методы глубокого обучения

Даже интуитивно задача преобразования некорректного фрагмента кода в корректный похожа на задачу машинного перевода. В случае с естественными языками для её решения применяются техники *нейронного машинного перевода* (Neural Machine Translation, NMT) [4]. Они основаны на применении рекуррентных нейронных сетей и архитектуры Encoder-Decoder, но детальное описание устройства их работы лежит за рамками данного исследования. Подобные методики применяются и в случае работы с кодом как набором токенов, например, в работах Tufano et al. [41, 35]. Существуют и некоторые модификации подходов к автоматическому восстановлению программ от ошибок с помощью ансамблей моделей и уже упоминавшихся свёрточных нейронных сетей [13, 8]. Однако все рассмотренные методы требуют наличия достаточно большого датасета примеров того, как изменялся код в том или ином случае, потому что при маленьком наборе данных такие глубокие модели могут просто переобучиться. А если стоит задача исправлять любые заранее определенные типы ошибок или уязвимостей, собрать такой датасет *для каждого* типа представляется очень сложной задачей, требующей отдельного исследования.

1.2.2. Применение изменений к фрагментам кода

При наличии шаблона того, как нужно исправлять фрагмент кода с ошибкой (в частности, хотя бы фрагментов «до» и «после» исправления), можно использовать так называемые *сценарии редактирования*, и изменять код в соответствии с ними. Сценарий редактирования представляет собой последовательность действий нескольких видов над вершинами AST. Обычно это добавление и удаление вершин, но можно также рассматривать обновление и перемещение. Искомые сценарии получают при помощи готовых сторонних инструментов, вро-

де ChangeDistiller [6], как в работах Meng et al. по выявлению и применению систематических изменений к коду [31, 30], или более популярного GumTree [16], как, например, в работе [27]. Согласно исследованию [16], GumTree генерирует более короткие и точные скрипты редактирования по сравнению с другими подобными инструментами, поэтому для использования в данной работе был впоследствии выбран именно он.

1.3. Графовые шаблоны изменений кода

Для анализа того, как разработчики пишут код на том или ином языке, исследователи собирают частые изменения кода из открытых проектов на GitHub и проводят их эмпирическое исследование [17, 21]. Это может быть полезно также и для разработчиков IDE: если множество людей совершают какое-то типичное изменение в своем коде, выглядит разумным предоставлять разработчикам возможность сделать его автоматически в редакторе кода.

В случае, когда нужно сохранять больше семантики об исходном коде, исследователи используют его графовые представления — например, граф потока управления (Control Flow Graph, CFG) или граф зависимостей программы (Program Dependence Graph, PDG). Однако Nguyen et al. в своей работе CPatMiner [21] пошли ещё дальше: в виде графов они представляют уже *изменения кода* и с их помощью ищут те самые шаблоны изменений в программах на Java, которые встречаются чаще всего. В данном разделе пойдет речь об устройстве этих графов, развитии идеи их использования, а также мотивации данной работы.

1.3.1. CPatMiner

Более формально, в своей работе [21] авторы вводят понятие «детализированного графа зависимостей программы» (в оригинале fine-grained PDG, здесь и далее: fgPDG). Его вершины могут быть трёх типов: вершины *данных*, соответствующие переменным и литералам в

исходном коде, вершины *операций*, соответствующие различным вызовам функций и операциям в целом, и вершины *контроля*, соответствующие управляющим конструкциям, вроде `if` или `for`. Все они могут быть связаны ориентированными ребрами *зависимостей по данным*, определяющими специфику потока данных в коде, а также *управляющими* ориентированными ребрами, которые показывают, что конечная вершина ребра контролируется исходной. Каждая вершина графа соответствует вершине исходного AST кода, а ребра могут идти только «вниз» по дереву, от предков к потомкам. Таким образом, fgPDG — ациклический ориентированный граф, которые можно построить на этапе статического анализа кода во время обхода AST.

После этого авторы определяют термин *граф изменений* — это граф, построенный из двух fgPDG кода *до* и *после* изменения, в котором «соответствующие» друг друг вершины соединены ребрами *отображения*. Под «соответствующими» здесь понимаются вершины, не изменённые совсем, или в которых изменилось только название метки, вроде имени переменной. Для того, чтобы собрать *семантические шаблоны* изменений, авторы анализируют историю коммитов в реальных Java-репозиториях, извлекая из них необходимые графы изменений и расширяя их рекурсивным алгоритмом сбора шаблонов. Описание процесса работы этого алгоритма выходит за рамки данной работы, детали его реализации можно найти в оригинальном репозитории проекта[1].

1.3.2. PythonChangeMiner

В силу того, что реализация алгоритма в CPatMiner является AST-специфичной, т.е. процесс построения графа напрямую зависит от анализируемого языка программирования и поддерживаемых конструкций в процессе разбора, его невозможно было переиспользовать для других языков. В частности, особый интерес представлял Python, инструментов по поиску шаблонов изменений в котором на тот момент практически не было, и поэтому в дипломной работе Вячеслава Бушева [43]

[1] `https://github.com/nguyenhoan/CPatMiner`

До:

```
for i in range(len(static)):
    dim = static[i]
```

После:

```
for i, dim in enumerate(static):
```

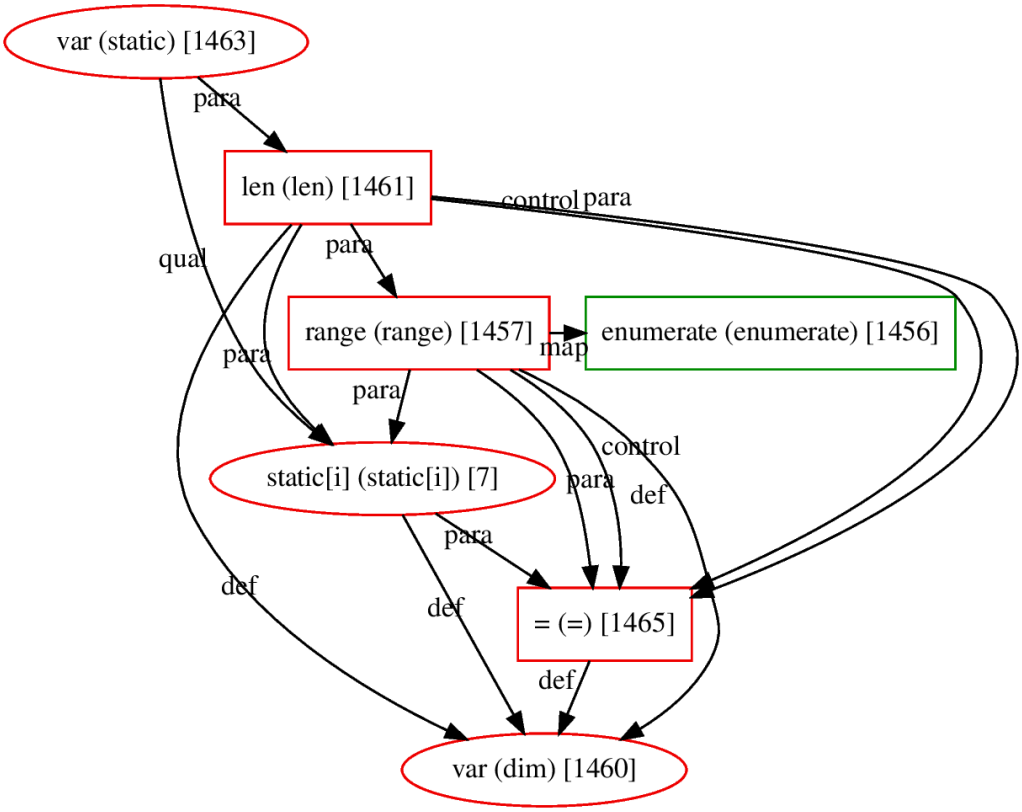

Рис 1: Пример семантического графового шаблона изменения кода на Python, который был обнаружен с помощью реализованного инструмента сразу в нескольких репозиториях (имена переменных опускались). Шаблон представляет собой замену цикла `for`, в котором осуществляется итерация по списку, на `for` с использование встроенной функции `enumerate`, которая генерирует для каждого элемента коллекции не только номер, но и значение сразу в виде кортежа.

указанный алгоритм поиска был адаптарован для него.

Позже с помощью реализованного инструмента[2] в рамках лаборатории JetBrains Research[3] провели масштабное исследование шаблонов изменений кода на Python [7], которые были собраны из 120 популярных GitHub-репозиториев. Репозитории выбирались среди четырёх

[2] `https://github.com/JetBrains-Research/python-change-miner`

[3] `https://research.jetbrains.org/ru-ru/groups/ml_methods/`

глобальных «доменов» (областей знания): анализ данных, машинное обучение, медиа и web-инструменты. Всего было собрано 7481 шаблонов изменений, из которых 803 проанализировали вручную, т.к. только они встречались хотя бы в двух различных репозиториях.

Также был проведен опрос среди вносивших собранные изменения разработчиков с целью выяснить, какие из этих изменений могут быть автоматизированы, т.е. локализоваться и применяться встроенными средствами самой IDE, а не усилиями разработчиков. Результаты опроса показали, что среди собранных изменений такие есть, а т.к. улучшение инструмента для их сбора тоже представляет собой простор для исследований, можно полагать, что искомых семантических шаблонов изменений вполне реально найти и больше, просто применяя различные эвристики к исходному алгоритму. Именно поэтому в данной работе и была поставлена цель реализовать инструмент для локализации графовых шаблонов в коде, который был бы способен находить и исправлять действительно сложные паттерны просто за счет использования графов вида fgPDG, а также был бы расширяемым, т.е. в потенциале способным принимать на вход абсолютно любые собранные графы изменений. Таким образом, описываемый в этой работе инструмент не только улучшает (с помощью графов зависимостей) классический подход к восстановлению программ от ошибок, но и находит практическое применение полученных шаблонов.

1.4. IntelliJ Platform

С технической точки зрения наиболее удобным для оценки качества и эффективности работы такого инструмента представляется реализовать его в виде плагина к популярной для языка Python IDE. Такой средой разработки является, например, PyCharm от компании JetBrains[4], реализованный на базе IntelliJ Platform[5]. Данная платформа (в частности, её SDK[6]) предоставляет широкий набор возможностей для под-

[4] `https://www.jetbrains.com/ru-ru/pycharm/`

[5] `https://jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html`

[6] SDK — Software Developmant Kit, набор средств для разработки ПО

держки различных языковых инструментов, средства для создания статических анализаторов кода и удобный API[7] для создания интерфейсов. Особенностью IntelliJ Platform является *Program Structure Interface (PSI)* — в широком смысле, интерфейс для универсального взаимодействия с моделями кода. В данной же работе используется понятие *PSI-дерева* кода, которое можно понимать как конкретное синтаксическое дерево, содержащее подробную информацию о всех токенах (в частности, их позиции в коде) в своих вершинах.

Для того, чтобы анализировать структуру кода в редакторе «на ходу», в IntelliJ Platform существуют механизмы, которые называются *инспекциями кода* (code inspections). Это внутренние процессы IDE, запускающиеся в фоне и помогающие разработчику поддерживать качество кода за счет различных уведомлений и подсветки потенциально ошибочных мест. Таким образом, механизм инспекций идеально подходит для реализации собственного инструмента для статического анализа кода и локализации шаблонов в коде. Но после того, как такой шаблон найден, необходимо предложить разработчику возможность автоматически исправить данный фрагмент. Всплывающее меню с подсказкой и вариантами действий (и возможностью их осуществления) в IntelliJ Platform называется *интенцией* (code intention).

[7]API — Application Program Interface, описание способов взаимодействия с программой

2. Описание реализации

В данной главе будет детально описан подход работы с графовыми шаблонами изменений кода после их сбора и анализа, архитектура инструмента для поиска искомых паттернов в коде разработчика, алгоритмы локализации и исправления в соответствии с шаблоном изменения, а также некоторые принятые технические решения и их мотивация.

2.1. Обзор предлагаемого подхода

Описываемый в данной работе подход к решению задачи автоматического исправления кода на основе графовых шаблонов представлен на рис. 2. Он состоит из нескольких этапов:

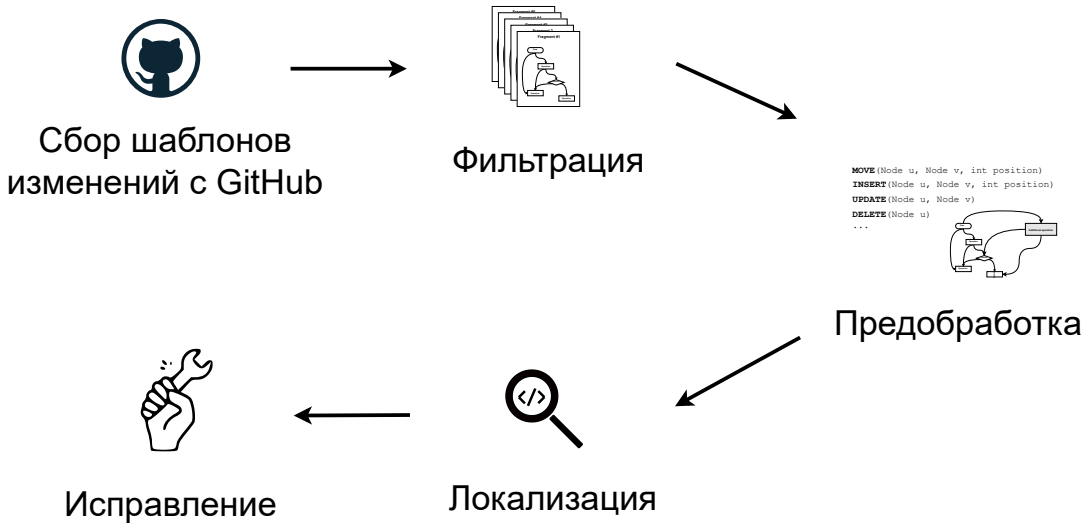

Рис 2: Предлагаемый подход к автоматическому исправлению кода на основе графовых шаблонов частых изменений.

1. Собрать с помощью инструмента PythonChangeMiner графовые шаблоны изменений кода на Python из репозиториев на GitHub.

2. Некоторым образом отфильтровать их, т.е. отобрать шаблоны изменений для последующей автоматизации.

3. Подготовить избранные шаблоны к загрузке в инструмент с помощью автоматического скрипта предварительной обработки.

4. В качестве инспекции для IDE в режиме on-line строить fgPDG для кода разработчика и локализовать в нем изоморфные подграфы шаблонов из части *до* изменения.

5. При подтверждении разработчиком, исправлять локализованный фрагмент графа кода с помощью шаблона изменения.

Стоит отметить, что инструмент, реализация которого являлась целью данной работы, включает в себя лишь задачи по автоматизации последних трёх этапов описываемого подхода. Предварительный сбор шаблонов изменений и их фильтрация — задача экспертов и пользователей инструмента, однако для полноты описания решения данные этапы все равно нуждаются в упоминании. Код описываемого далее плагина для IDE PyCharm можно найти в открытом репозитории на GitHub[8].

2.2. Фильтрация графовых шаблонов

Как уже было упомянуто ранее, эксперты лаборатории JetBrains Research вручную проанализировали релевантность 803-х шаблонов изменений кода из 7481 — рассматривались лишь те изменения, которые встречались в коммитах хотя бы двух разных репозиториях на GitHub [7]. На данный момент из них выбрано десять в качестве экспериментальных для последующей автоматизации (в том числе, наиболее полезных по результатам опроса авторов этих изменений, разработчиков на GitHub и команды IDE PyCharm [7]). Все избранные шаблоны предлагают эквивалентные преобразования над кодом и вносят в основном стилистические изменения, основываясь на замене устаревших API-вызовов или следовании нормам написания кода на языке Python[9].

[8] `https://github.com/JetBrains-Research/revizor`

[9] `https://www.python.org/dev/peps/pep-0020/`

Предобработанные графы изменений этих шаблонов поставляются вместе с описываемым инструментом внутри JAR-файла плагина для IDE PyCharm в качестве ресурсов.

2.3. Локализация графовых шаблонов в коде

Для каждого выбранного шаблона (т.е. графа изменения кода) можно выделить *порождённые подграфы вида fgPDG* для кода «до» и «после» в силу конструкции его построения, которая уже была описана ранее. Порождённым подграфом вида fgPDG здесь называется граф, образованный из подмножества вершин графа изменения вместе со всеми рёбрами, соединяющими пары вершин из этого подмножества. Именно такой граф для кода «до» и предлагается искать в процессе локализации.

Для того, чтобы сопоставлять графы шаблонов с графом кода разработчика, последний нужно тоже предварительно построить. Так как процесс конструирования fgPDG в PythonChangeMiner был напрямую связан с использованием Python-библиотеки ast[10], алгоритм пришлось повторно реализовать на языке Kotlin в рамках плагина с использованием PSI-дерева. Таким образом, предлагаемая инспекция в режиме on-line строит fgPDG кода, открытого в редакторе IDE у разработчика.

Ключевая идея локализации же состоит в том, что задача поиска fgPDG-шаблона в целевом fgPDG кода разработчика эквивалентна задаче поиска изоморфного подграфа в графе (которая в общем случае является NP-полной). В данной работе, в частности, было решено воспользоваться готовой реализацией эвристического алгоритма поиска изоморфных подграфов VF2 [42] из библиотеки JGraphT [24]. Таким образом, каждый раз при обновлении PSI-дерева кода в редакторе плагин вызывает инспекцию, которая заново строит fgPDG нового кода, и проверяет его на возможные изоморфизмы с подграфами шаблонов с помощью библиотечного класса `VF2SubgraphIsomorphismInspector`.

[10] `https://docs.python.org/3/library/ast.html`

2.3.1. Проблема сопоставления имён

Для использования вышеуказанного класса необходимо задать способ сравнения рёбер и вершин двух графов между собой на предмет равенства в изоморфизме. И если с ребрами всё очевидно (просто сравнивать типы рёбер достаточно), то способов сравнить вершины fgPDG шаблона и fgPDG целевого кода есть несколько, особенно в случае, когда сравниваются вершины *данных*, содержащие имена переменных и атрибутов. Так, например, вершины с метками `list_of_items`, `lst` или `my_list` не должны сопоставляться с шаблоном строго, т.к. переменная списка у разработчика может называться как угодно, в то время как вершина, содержащая имя `collections.Callable`, должна совпасть с вершиной только с точно таким же именем. Метод сопоставления имён в вершинах данных предлагается определять автоматически на этапе предварительной обработки, однако можно сделать это и вручную в консоли разработчика, т.к. все средства для разметки в описываемом плагине имеют один и тот же интерфейс.

2.4. Предварительная обработка шаблонов

Важно отметить, что каждый шаблон, полученный в результате работы инструмента PythonChangeMiner, по сути не задает какого-то одного графа изменения кода. Там присутствует сразу несколько *графов-фрагментов* — это графы, которые нашлись в процессе поиска шаблонов исходным инструментом в результате процесса «рекурсивного расширения», и они все изоморфны между собой. Кроме того, для каждого графа-фрагмента сохранялись версии кода изменяемой функции «до» и «после», а также некоторая мета-информация. Директории шаблонов с указанными наборами файлов и являются входными данными для скрипта предобработки.

Изначально в скрипте происходит загрузка указанных шаблонов в оперативную память, построение и кэширование PSI необходимых фрагментов кода и соответствующих fgPDG. Для каждого шаблона у пользователя также запрашивается текст для всплывающего окна

с подсказкой в IDE. В качестве репрезентативного графа-изменения шаблона выбирается тот фрагмент, в котором осуществляется наименьшее количество действий по изменению AST кода (по сути, репрезентативным может являться любой фрагмент в силу изоморфизма, но с указанным удобнее работать). Далее, для вершин данных в графах-фрагментах эвристически определяется один из трёх способов сопоставления с вершинами в коде разработчика:

1. Сопоставление суффикса имени данной вершины с наибольшим общим суффиксом всех имён вершин в той же позиции из графов-фрагментов. Например, это полезно в случае атрибута `.keys`, чтобы `vocab.keys` или `d.keys` могли в результате изоморфизма отобразиться на вершину с именем `obj.dict.keys`. Этот способ выбирается, когда искомый общий суффикс имен вершин графов-фрагментов содержит в себе хотя бы один атрибут имени целиком.

2. Сопоставление имени данной вершины *только* с конкретными именами, встреченными в *вершинах данных* в той же позиции в графах-фрагментах. Например, как в случае с `collections.Callable`. Такой способ выбирается, если не подошел предыдущий и все имена во всех соответствующих вершинах фрагментов одинаковы, либо содержат атрибуты.

3. Произвольное сопоставление имен данной вершины — полезно в уже рассмотренном случае с пользовательскими произвольными именами переменных. Выбирается, если не подходят предыдущие два варианта.

После того, как определены режимы сопоставления вершин, для каждого шаблона изменения вычисляется соответствующий ему *сценарий редактирования*. В данной работе все последующие изменения кода в соответствии с шаблоном осуществляются именно с помощью действий над вершинами PSI-дерева кода. Для выделение набора таких действий из двух фрагментов кода «до» и «после» используется

уже упомянутый инструмент GumTree, запускающийся на их PSI-представлении. Однако даже для репрезентативного графа-изменения такой сценарий редактирования PSI-дерева может содержать лишние действия, никак не связанные с шаблоном изменения, но содержащиеся в коммите. Чтобы избавляться от них, в данной работе используется подход, описанный авторами инструментов LASE [31] и SYDIT [30] — поиск наибольшей общей подпоследовательности таких действий среди все графов-фрагментов. Авторы указанных работ также описывали и построение некоторого контекста для корректного сравнения действий над вершинами деревьев, но в случае, когда есть изоморфные графы зависимостей, это перестает быть проблемой.

С помощью полученного сценария редактирования, общего для всех фрагментов шаблона, происходит «дополнение» репрезентативного графа шаблона дополнительными вершинами, задействованными в этих действиях. Мотивация этого шага достаточно очевидна: если локализовать в коде разработчика только искомый граф фрагмента, то для осуществления его изменения в соответствии со сценарием редактирования необходимо будет использовать вершины, которые участвуют в этих самых действиях по изменению PSI-дерева кода. Однако для того, чтобы гарантировать, что они будут существовать, проще всего просто добавить их в граф шаблона в качестве дополнительных вершин и провести рёбра зависимостей от них до остальных вершин графа шаблона.

Таким образом, после окончания скрипта предварительной обработки шаблонов, имеется директория с паттернами, каждый из которых представлен четырьмя файлами:

1. `graph.dot` — основной fgPDG, графовое представление шаблона изменения кода.

2. `actions.json` — сериализованный сценарий редактирования PSI-дерева фрагмента кода.

3. `labels_groups.json` — режимы сопоставления имен переменных в вершинах данных графа.

4. `description.txt` — описание шаблона для всплывающего окна-«подсказки» в IDE.

Эта директория помещается в раздел ресурсов плагина и может поставляться вместе с его распространяемой копией.

2.5. Автоматическое исправление кода

После установки плагина в IDE PyCharm, при каждом изменении файла с расширением `.py` запускается инспекция, которая, как уже было упомянуто, строит fgPDG для всех функций файла в текущем редакторе кода. Если в каких-либо из этих графов обнаруживается изоморфный некоторому шаблону порождённый подграф, соответствующие токены кода «подсвечиваются», и разработчику предлагается действие по исправлению данного фрагмента кода с текстом подсказки, указанным на этапе предобработки шаблонов. В случае, когда в одной строке кода присутствует сразу несколько токенов, соответствующих вершинам обнаруженного подграфа, подсвечивается их наименьший общий предок (Least Common Ancestor, LCA) в PSI-дереве кода.

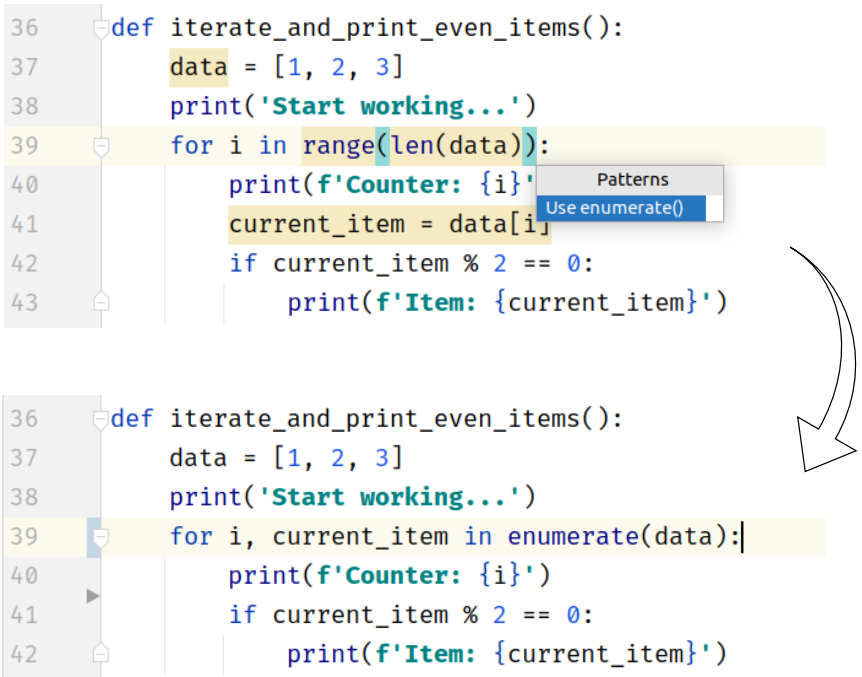

Рис 3: Пример локализации и авто-замены в плагине.

Если разработчик подтверждает предложенное исправление, для локализованного фрагмента кода в соответствии с шаблоном изменения запускается сохраненный заранее сценарий редактирования. Он содержит действия над вершинами PSI-дерева четырёх типов: добавление, удаление, обновление метки и перемещение по дереву. Применение последовательности таких действий для разработчика эквивалентно автозамене кода в редакторе, которую он также может впоследствии отменить, т.к. изменения PSI-дерева происходят внутри одного атомарного взаимодействия с IDE (`WriteAction`). Пользовательский интерфейс такой авто-замены, предоставляемой плагином, представлен на рис. 3.

2.6. Архитектура плагина

Диаграмма компонентов инструмента, включающая в себя модули ядра, скриптов предобработки и поиска графовых шаблонов в коде представлена на рис. 4. Основным компонентом является `Kernel`, содержащий в себе логику построения fgPDG для PSI кода на Python, а также классы-«фасады» для взаимодействия с библиотеками GumTree и JGraphT. Все они используются компонентом `Plugin`, реализующем инспекцию кода для локализации шаблонов (`PatternBasedInspection`) и предоставляющем UI для применения автоматических исправлений на основе интенций (`PatternBasedAutoFix`). Работа с графовыми шаблонами изменений в плагине осуществляется с помощью компонента `PatternsRepository`, который использует хранилище искомых шаблонов в `Resources`. Скрипт предобработки описан специальным компонентом `Preprocessing`: там содержатся используемые модели данных (`Pattern`, `EditActions`, `CodeChangeSample`) и компоненты, реализующие преобразование графовых шаблонов, полученных инструментом PythonChangeMiner в формат, используемый данным плагином. `CLIRunner` запускается из консоли, инстанцирует образ IntelliJ IDEA в фоновом режиме без графического интерфейса (headless mode) и обходит указанные директории с графовыми шаблонами, загружая PSI фрагментов кода и необходимые наборы сценариев редактирования с

помощью кэширующих загрузчиков (`CachingLoaders`), а также автоматически определяет режимы сопоставления имён переменных в вершинах данных графа (`MatchingModeLabelers`). Для реализации плагина и всех скриптов предобработки был использован язык Kotlin.

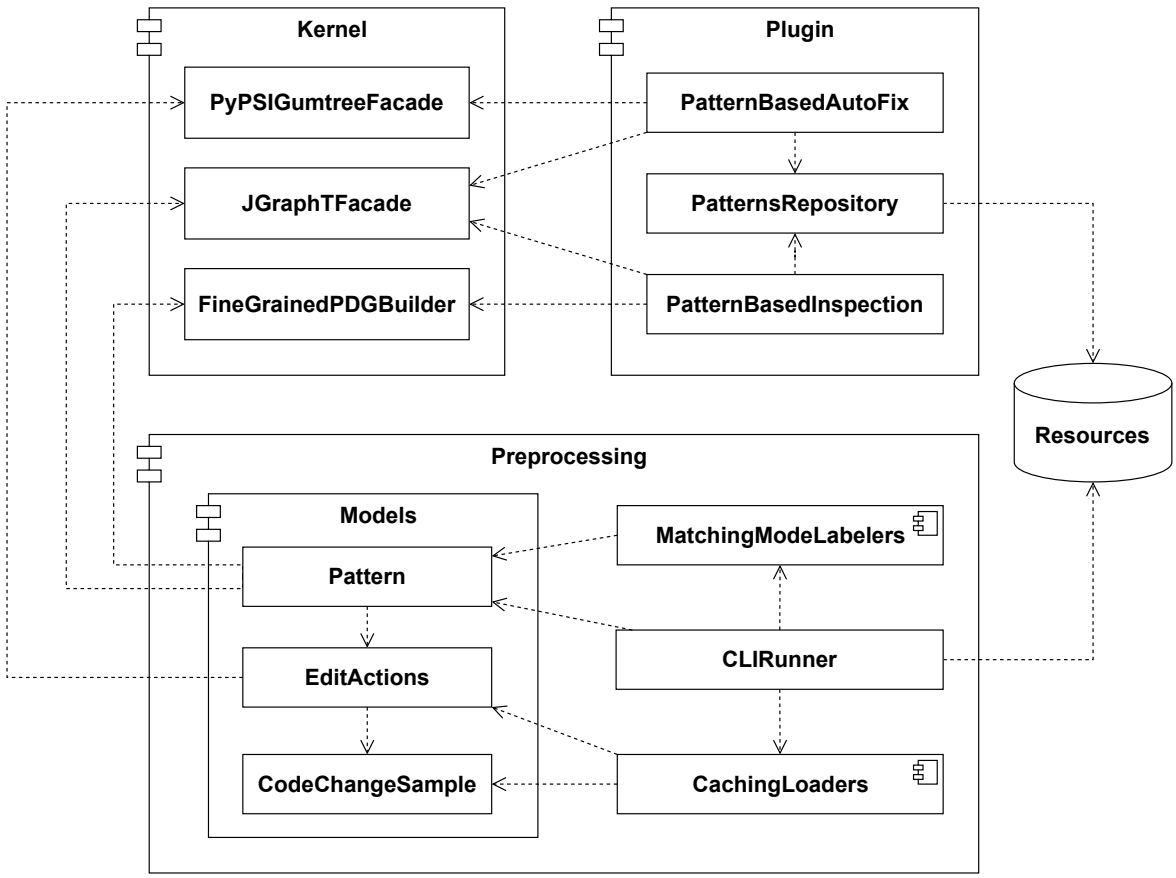

Рис 4: Диаграмма компонентов плагина для IDE PyCharm.

3. Апробация

В качестве предварительного исследования 9 разработчиков протестировали описываемый плагин на тестовом проекте, содержавшем в себе все поддерживаемые на данные момент графовые шаблоны. Все участники апробации имели от двух до пяти лет профессионального опыта в индустрии и в опросе подтверждали, что они часто используют инспекции и интенции в IDE для улучшения качества кода. Также все они отметили, что идея использования частых изменений других разработчиков open-source проектов на GitHub в качестве подсказок в редакторе кода IDE выглядит перспективной и потенциально полезной, даже в образовательных целях.

После тестирования все участвующие в апробации разработчики прошли опрос, в котором нужно было оценить различные аспекты работы плагина по шкале от 1 (крайне неудовлетворительно) до 4 (очень удовлетворительно). В среднем по каждому из аспектов были получены следующие оценки:

- Корректность производимых плагином эквивалентных преобразований кода: 3.66/4.
- Удобство использования плагина для автоматического исправления кода: 3.77/4.
- Производительность плагина с точки зрения не влияния на производительность IDE: 3.88/4.
- Визуализация подсказок в редакторе кода IDE: 3.33/4.

Как видно из результатов опроса, первые три аспекта работы плагина получили сравнительно высокие оценки, однако незначительно ниже других разработчики оценили часть с визуализацией подсказок. Это связано с тем, что подсвечивать избранные для апробации шаблоны как предупреждения об ошибках не совсем корректно в силу того, что они вносят скорее стилистические правки. Также иногда возникали

неудобства с подсветкой токенов одного шаблона в различных местах кода (в силу того, что шаблоны графовые, зависимости по рёбрам могут идти сколь угодно далеко по коду). Некоторые из предложенных замечаний были учтены в текущей версии плагина[11].

[11] `https://github.com/JetBrains-Research/revizor`

4. Заключение

В ходе производственной практики были достигнуты следующие результаты:

1. Проведен обзор предметной области, изучены методы автоматического восстановления программ от ошибок, выявлены существующие недостатки и ограничения.

2. Предложен расширяемый подход к изменению кода на основе графовых шаблонов.

3. Реализован процесс поиска соответствий шаблону в целевом графе кода на Python на этапе локализации с помощью изоморфизмов подграфов.

4. Реализованы алгоритмы предобработки шаблонов, выделения общих сценариев редактирования и их применения к коду разработчика.

5. Разработан плагин[12] для IDE PyCharm.

6. Проведена апробация плагина с участием реальных разработчиков.

В будущем планируется доработка части с визуализацией подсказок в существующей версии плагина, поддержка большего числа паттернов и публикация статьи на эту тему.

[12] `https://github.com/JetBrains-Research/revizor`

Список литературы

[1] Abreu Rui, Zoeteweij Peter, Van Gemund Arjan JC. On the accuracy of spectrum-based fault localization // Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007) / IEEE. — 2007. — P. 89–98.

[2] Automatic Feature Exploration and an Application in Defect Prediction / Yu Qiu, Yun Liu, Ao Liu et al. // IEEE Access. — 2019. — Vol. 7. — P. 112097–112112.

[3] Avatar: Fixing semantic bugs with fix patterns of static analysis violations / Kui Liu, Anil Koyuncu, Dongsun Kim, Tegawendé F Bissyandé // 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) / IEEE. — 2019. — P. 1–12.

[4] Bahdanau Dzmitry, Cho Kyunghyun, Bengio Yoshua. Neural machine translation by jointly learning to align and translate // arXiv preprint arXiv:1409.0473. — 2014.

[5] 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.

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

[7] Golubev Yaroslav, Li Jiawei, Bryksin Timofey et al. Changes from the Trenches: Should We Automate Them? — 2021. — 2105.10157.

[8] CoCoNuT: combining context-aware neural translation models using ensemble for program repair / Thibaud Lutellier, Hung Viet Pham,

Lawrence Pang et al. // Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. — 2020. — P. 101–114.

[9] DeepDelta: learning to repair compilation errors / Ali Mesbah, Andrew Rice, Emily Johnston et al. // Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2019. — P. 925–936.

[10] DeepRace: Finding Data Race Bugs via Deep Learning / Ali Tehrani, Mohammed Khaleel, Reza Akbari, Ali Jannesari // arXiv preprint arXiv:1907.07110. — 2019.

[11] DevReplay: Automatic Repair with Editable Fix Pattern / Yuki Ueda, Takashi Ishio, Akinori Ihara, Kenichi Matsumoto // arXiv preprint arXiv:2005.11040. — 2020.

[12] Distributed representations of words and phrases and their compositionality / Tomas Mikolov, Ilya Sutskever, Kai Chen et al. // Advances in neural information processing systems. — 2013. — P. 3111–3119.

[13] ENCORE: Ensemble learning using convolution neural machine translation for automatic program repair / Thibaud Lutellier, Lawrence Pang, Viet Hung Pham et al. // arXiv preprint arXiv:1906.08691. — 2019.

[14] Efficient estimation of word representations in vector space / Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean // arXiv preprint arXiv:1301.3781. — 2013.

[15] Face recognition: A convolutional neural-network approach / Steve Lawrence, C Lee Giles, Ah Chung Tsoi, Andrew D Back // IEEE transactions on neural networks. — 1997. — Vol. 8, no. 1. — P. 98–113.

[16] 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. — 2014. — P. 313–324.

[17] Fixminer: Mining relevant fix patterns for automated program repair / Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé et al. // Empirical Software Engineering. — 2020. — P. 1–45.

[18] Fu Wei, Menzies Tim. Revisiting unsupervised learning for defect prediction // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. — 2017. — P. 72–83.

[19] Getafix: Learning to fix bugs automatically / Johannes Bader, Andrew Scott, Michael Pradel, Satish Chandra // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, no. OOPSLA. — P. 1–27.

[20] Goues Claire Le, Pradel Michael, Roychoudhury Abhik. Automated program repair // Communications of the ACM. — 2019. — Vol. 62, no. 12. — P. 56–65.

[21] Graph-based mining of in-the-wild, fine-grained, semantic code change patterns / Hoan Anh Nguyen, Tien N Nguyen, Danny Dig et al. // 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) / IEEE. — 2019. — P. 819–830.

[22] Gzoltar: an eclipse plug-in for testing and debugging / José Campos, André Riboira, Alexandre Perez, Rui Abreu // Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. — 2012. — P. 378–381.

[23] Improving bug detection via context-based code representation learning and attention-based neural networks / Yi Li, Shaohua Wang, Tien N Nguyen, Son Van Nguyen // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, no. OOPSLA. — P. 1–30.

[24] JGraphT–A Java library for graph data structures and algorithms / Dimitrios Michail, Joris Kinable, Barak Naveh, John V Sichi // arXiv preprint arXiv:1904.08355. — 2019.

[25] Könighofer Robert, Bloem Roderick. Automated error localization and correction for imperative programs // 2011 Formal Methods in Computer-Aided Design (FMCAD) / IEEE. — 2011. — P. 91–100.

[26] Kummerfeld Sarah K, Kay Judy. The neglected battle fields of syntax errors // Proceedings of the fifth Australasian conference on Computing education-Volume 20 / Citeseer. — 2003. — P. 105–111.

[27] Le Xuan Bach D, Lo David, Le Goues Claire. History driven program repair // 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) / IEEE. — Vol. 1. — 2016. — P. 213–224.

[28] Lozano Antoni, Valiente Gabriel. On the maximum common embedded subtree problem for ordered trees // String Algorithmics. — 2004. — P. 155–170.

[29] Mechtaev Sergey, Yi Jooyong, Roychoudhury Abhik. Angelix: Scalable multiline program patch synthesis via symbolic analysis // Proceedings of the 38th international conference on software engineering. — 2016. — P. 691–701.

[30] Meng Na, Kim Miryung, McKinley Kathryn S. Sydit: creating and applying a program transformation from an example // Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. — 2011. — P. 440–443.

[31] 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.

[32] Monperrus Martin. The living review on automated program repair. — 2020.

[33] Neural program repair by jointly learning to localize and repair / Marko Vasic, Aditya Kanade, Petros Maniatis et al. // arXiv preprint arXiv:1904.01720. — 2019.

[34] Norman Ronald J, Nunamaker Jay F. Integrated development environments: technological and behavioral productivity perceptions // Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track / IEEE Computer Society. — Vol. 2. — 1989. — P. 996–997.

[35] On learning meaningful code changes via neural machine translation / Michele Tufano, Jevgenija Pantiuchina, Cody Watson et al. // 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) / IEEE. — 2019. — P. 25–36.

[36] Pradel Michael, Sen Koushik. DeepBugs: A learning approach to name-based bug detection // Proceedings of the ACM on Programming Languages. — 2018. — Vol. 2, no. OOPSLA. — P. 1–25.

[37] Silva Danilo, Tsantalis Nikolaos, Valente Marco Tulio. Why we refactor? confessions of github contributors // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. — 2016. — P. 858–870.

[38] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia et al. // ACM Comput. Surv. — 2018. — Vol. 51, no. 3.

[39] You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems / Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé et al. // 2019 12th IEEE conference on software testing, validation and verification (ICST) / IEEE. — 2019. — P. 102–113.

[40] Zayour Iyad, Hajjdiab Hassan. How much integrated development environments (ides) improve productivity? // JSW. — 2013. — Vol. 8, no. 10. — P. 2425–2431.

[41] An empirical study on learning bug-fixing patches in the wild via neural machine translation / Michele Tufano, Cody Watson, Gabriele Bavota et al. // ACM Transactions on Software Engineering and Methodology (TOSEM). — 2019. — Vol. 28, no. 4. — P. 1–29.

[42] A (sub) graph isomorphism algorithm for matching large graphs / Luigi P Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento // IEEE transactions on pattern analysis and machine intelligence. — 2004. — Vol. 26, no. 10. — P. 1367–1372.

[43] Бушев Вячеслав Валериевич. Инструмент для поиска шаблонов изменений в коде на языке Python. — 2020.

[44] Тучина Анастасия Игоревна. Поиск ошибок в коде при помощи глубокого обучения в IntelliJ IDEA. — 2019.