**Министерство образования и науки Российской Федерации**

**Федеральное агентство по образованию**

**ФГАОУ ВО «Санкт-Петербургский политехнический университет Петра Великого»**

**Институт компьютерных наук и технологий**

На правах рукописи

Научно-квалификационная работа (диссертация)

# МЕТОДЫ ОБНАРУЖЕНИЯ И ИСПРАВЛЕНИЯ ОШИБОК ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ОСНОВЕ АНАЛИЗА ПРОГРАММНЫХ РЕПОЗИТОРИЕВ С ПОМОЩЬЮ МАШИННОГО ОБУЧЕНИЯ

Направление подготовки 09.06.01 «Информатика и вычислительная техника»

*Код и наименование*

Направленность 09.06.01_06 «Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

*Код и наименование*

Автор работы,
аспирант группы
3560901/60601:
Бельский А.

Научный
руководитель:
доцент, к.т.н.,
Ицыксон В. М.

Санкт-Петербург 2020

# ОГЛАВЛЕНИЕ

# ВВЕДЕНИЕ

Объемы разрабатываемого программного обеспечения в последнее время постоянно растут, цикл разработки сокращается, что обычно приводит к тотальному снижению качества программных продуктов. Это недопустимо в таких областях, как встраиваемые системы в медицине, энергетике, машиностроении, финансовом секторе и других, так как может привести к существенным материальным потерям или опасности для жизни и здоровья людей.

Для преодоления указанных проблем разработчики используют различные методы повышения качества программного обеспечения. К таким методам относятся, например, тестирование, верификация или статический анализ программного обеспечения. Однако, все общепринятые методы повышения качества имеют определенные ограничения, они не могут в полной мере гарантировать качество программ. Например, тестирование может обнаруживать ошибки в программном обеспечении, но не может гарантировать отсутствие ошибок в протестированном программном обеспечении. Кроме того, имеются целые классы программ, например, параллельные системы, поведение которых может быть недетерминированно и тестирование которых неэффективно. Формальные методы, такие как дедуктивная верификация, статический анализ пока ограничены размером анализируемых программ и могут применяться к только к узкой области программных проектов.

В последнее время в программной инженерии активно проводятся исследования в области анализа и применения накопленного гигантского опыта миллионов программистов при написании сотен тысяч программных проектов. Этот опыт зафиксирован в программных репозиториях (Version control systems, VCS) в виде истории изменения проектов и комментариев к коммитам, а также в системах управления задачами и ошибками (issue tracking и bag tracking) в виде истории изменения задач и ошибок. Существует большое число методов, которые анализируют накопленную информацию и извлекают из нее знания, используемые при решении разных задач программной инженерии. Такие методы хорошо показали себя в разных областях программной инженерии. В последнее время такие подходы

стали применяться и в области обнаружения и исправления программных ошибок, при этом используются не только артефакты анализируемых программных проектов, но и не задействованный ранее потенциал информации, хранящейся в сотнях тысяч программных репозиториев и позволяющих переиспользовать и обобщать опыт миллионов разработчиков.

**Объектом исследования** являются методы генерации программного кода [1]. Суть данных методов заключается в автоматическом создании программного кода специальными приложением (генератор кода) на основании заданных исходных параметрах и заранее определенных шаблонов. Подобные специальные приложения применяется в целях сокращения времени разработки и повышения качества разрабатываемого программного обеспечения за счет исключения человеческого фактора и унификации процессов разработки.

**Предметом исследования** являются методы автоматизированного исправления ошибок программного кода [2], которые входят в класс методов генерации программного кода. Данные методы представляют из себя специальные алгоритмы генерации программного кода с целью получения исправлений ошибок программного кода (патчей) на основании исходного кода программы, содержащей ошибку, тестовых сценариев, опыта успешных патчей из программных репозиториев, и других исходных данных.

**Целью работы** является создание метода автоматизированного исправления ошибок программного кода, основанного на извлечении опыта успешных патчей из программных репозиториев.

Достижение поставленной цели осуществляется за счет выполнения следующих **задач**:

- провести аналитический обзор существующих методов автоматизированного исправления ошибок программного кода;
- провести сравнительный анализ существующих методов автоматизированного исправления ошибок программного кода;
- разработать новый, ранее не существовавший метод автоматизированного исправления ошибок программного кода путем применения методов

машинного обучения к извлеченному опыту успешных патчей из программных репозиториев;

- разработать программный инструментарий, реализующий разработанный метод.

В процессе реализации метода автоматизированного исправления ошибок программного кода, представленного в рамках данной работы, применялись **следующие методы**: рекурсивный спуск [3], логистическая регрессия [4], градиентный спуск [5].

**Научная новизна** представленной работы заключается в применении классов методов Data Mining и машинного обучения к программным репозиториям, системам управления задачами и дефектами и другим базам данных с артефактами программных проектов с целью извлечения знания об успешно решенных задачах исправления программных ошибок другими разработчиками.

**Теоретическая значимость** работы заключается в разработке нового, ранее не существовавшего метода автоматизированного исправления ошибок программного кода для языка программирования ABAP путем применения классов методов Data Mining и машинного обучения, широко используемых в аналогичных работах других авторов, таких как Data-Guided Repair of Selection Statements [23] и Automatic Patch Generation by Learning Correct Code [25], чьи подходы к решению подобного рода задач были взяты за основу при разработке нового метода.

**Практическая значимость** работы заключается в получении программного инструментария для языка программирования ABAP, реализующего разработанный метод автоматизированного исправления ошибок программного кода и результаты испытаний данного программного инструментария на реальных проектах SAP.

Выпускная квалификационная работа организована следующим образом. Первая глава посвящена выбору и обоснованию критериев для проведения сравнительного анализа существующих методов и обзору предметной области. Вторая глава содержит анализ и описание поставленной задачи исследования и пути ее решения. Третья глава иллюстрирует схему и вербальное описание сути разработанного авторами метода, а также посвящена детальному описанию

разработанного метода и включает в себя алгоритмы, математическую модель, используемые технологии и методы. В четвертой главе приведено описание реализации разработанного метода как программного инструментария. В пятой главе приведены результаты тестирования, разработанного метода и анализ полученных результатов. В заключении подводятся итоги, проводится оценка полученных результатов и формулируются планы дальнейших исследовании.

# 1. ОБЗОР И АНАЛИЗ СУЩЕСТВУЮЩИХ МЕТОДОВ ИСПРАВЛЕНИЯ ОШИБОК ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В этой главе сформулированы критерии для проведения сравнительного анализа существующих методов обнаружения и исправления ошибок программного обеспечения, проводится обзор предметной области и результаты сравнительного анализа обозреваемых методов.

## 1.1 Критерии для проведения сравнительного анализа существующих методов

На текущий момент, методы обнаружения и исправления ошибок программного обеспечения можно разделить на синтаксический подход (syntactic approach) и семантический подход (semantic approach).

В свою очередь алгоритмы обнаружения и исправления ошибок программного обеспечения, основанные на синтаксическом подходе, можно классифицировать следующим образом :

- эволюционные алгоритмы (генетическое программирование);
- машинное обучение.

### *1.1.1 Эволюционные алгоритмы (генетическое программирование)*

Генетическое программирование (Genetic programming) - метод стохастического поиска решения проблем, основанный на идеях эволюционной генетики, который включают в себя генотип (генетический материал индивида), хранящийся в памяти, дифференциальное воспроизводство этих генотипов и вариаций, которые создаются процессами, аналогичными биологическим процессам мутации и кроссовера [6]. В рамках данной работы мы будем рассматривать представленный метод применительно к поиску подходящих исправлений ошибок программного обеспечения, алгоритм которого отражен на рисунке 1.1.

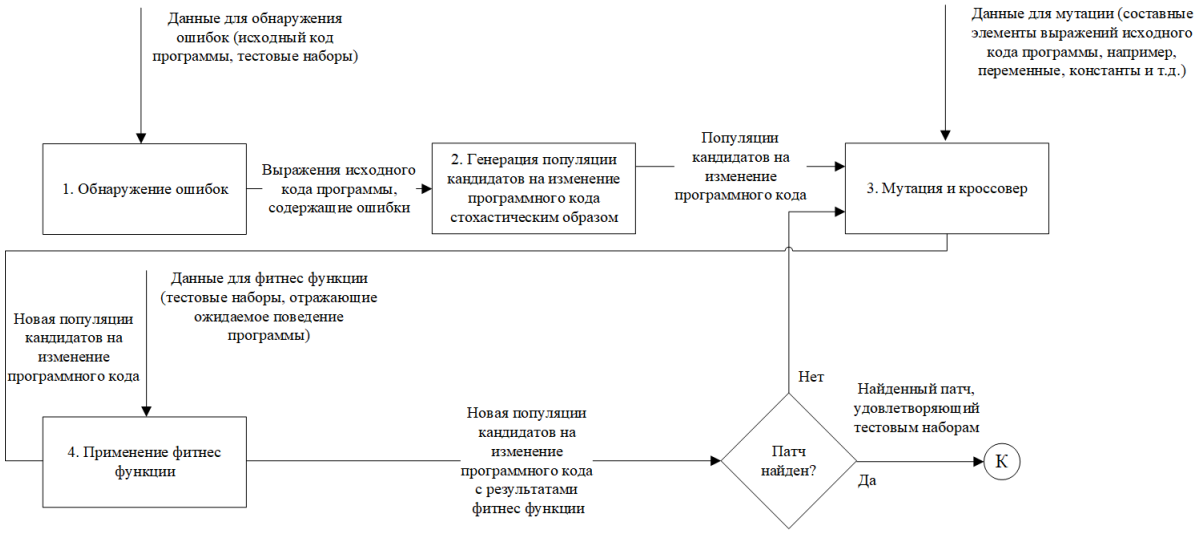


Рисунок 1.1 Алгоритм обнаружения и исправления ошибок программного обеспечения, основанный на генетическом программировании

1. Обнаружение выражений исходного кода программы, содержащие ошибки. Набор данных формируется на основании исходного кода программы и тестовых наборов путем применения статического и динамического анализа.
2. Для выполнения генетического алгоритма необходимо сформировать исходную популяцию кандидатов на изменение программного кода. Данная популяция формируются стохастическим образом для обнаруженных выражений исходного кода программы, содержащих ошибки.
3. С помощью операторов мутации (изменение составной части элемента популяции, например, изменение символа в выражении с “+” на “-”) и кроссовера (обмен составными частями между элементами популяции, например, обмен переменными между двумя выражениями) формируется новая популяция кандидатов на изменение программного кода.
4. К полученной новой популяции применяется фитнес функция (функция, сформированная на основании тестовых наборов, которые описывают ожидаемое поведение программы) для валидации полученных результатов. В случае отсутствия подходящих кандидатов на изменение программного кода алгоритм переходит на шаг 2, пока не будет найдены удовлетворяющие фитнес функции кандидаты.

### *1.1.2 Машинное обучение*

В последнее время, в связи со значительным ростом объема информации, хранящейся в программных репозиториях широкое распространение получило применение методов машинного обучения для решения проблем обнаружения и исправления ошибок программного обеспечения. Реализация данного класса методов искусственного интеллекта представлена на рисунке 1.2:

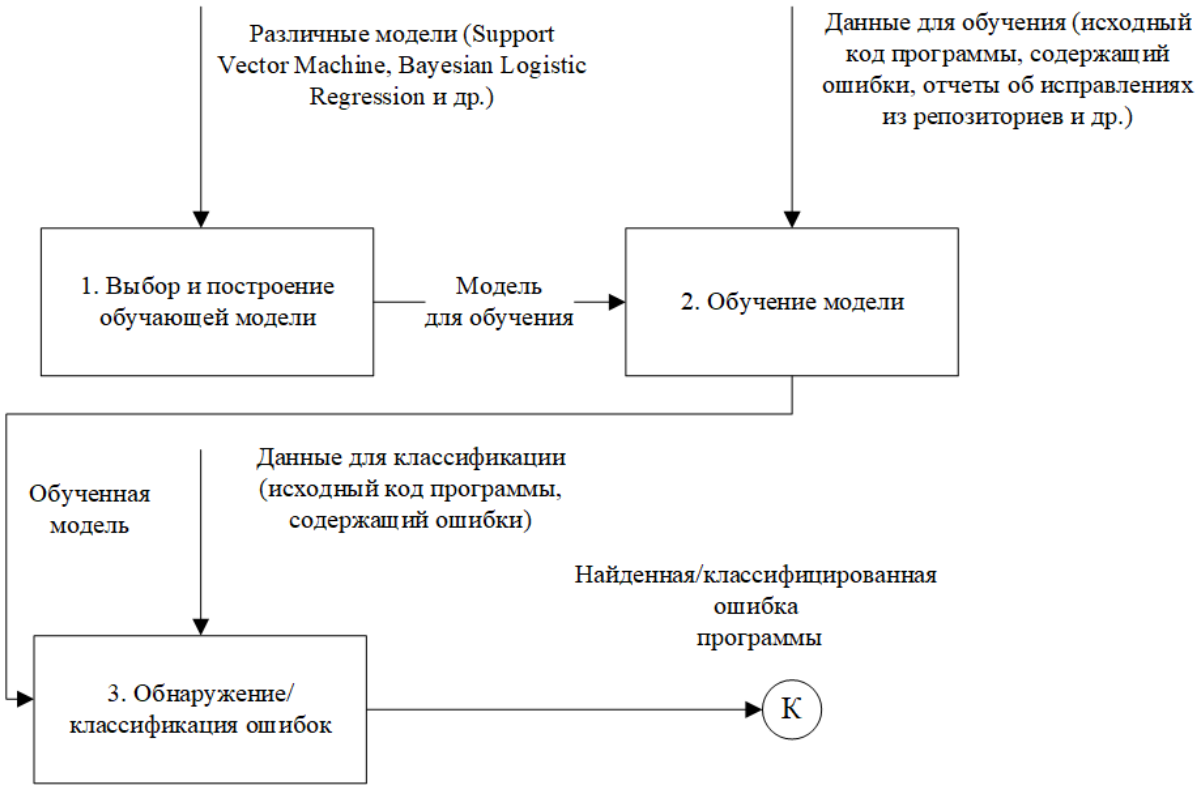


Рисунок 1.2 Алгоритм обнаружения и исправления ошибок программного обеспечения, основанный на машинном обучении

1. Выбор и построение обучающей модели. На текущий момент существуют различные алгоритмы построения моделей машинного обучения, например, метод опорных векторов (Support Vector Machine, Bayesian Logistic Regression и др.) [7].
2. Производится обучение модели с учителем, в данном случае выбирается исходный код программы и отчеты об ошибках из репозиториев с открытым кодом.
3. Получившуюся на шаге 2 обученную модель применяют для задач классификации, например, для решения задач обнаружения ошибок в исходном

коде программы или определения подходящих патчей, классифицируемых на общих с ошибкой параметрах.

*1.1.*3 *Семантический подход*

Появление семантического подхода для решения задач исправления ошибок программного обеспечения обусловлено наличием таких ограничений в синтаксическом подходе, как масштабируемость (scalability), отсутствие возможности мультистроковой модификации исходного кода программ, низкая эффективность, низкое быстродействие и т.д.

Большинство рассматриваемых в данной статье методов обнаружения и исправления ошибок программного обеспечения используют именно семантический подход, алгоритм которого представлен на рисунке 1.3.

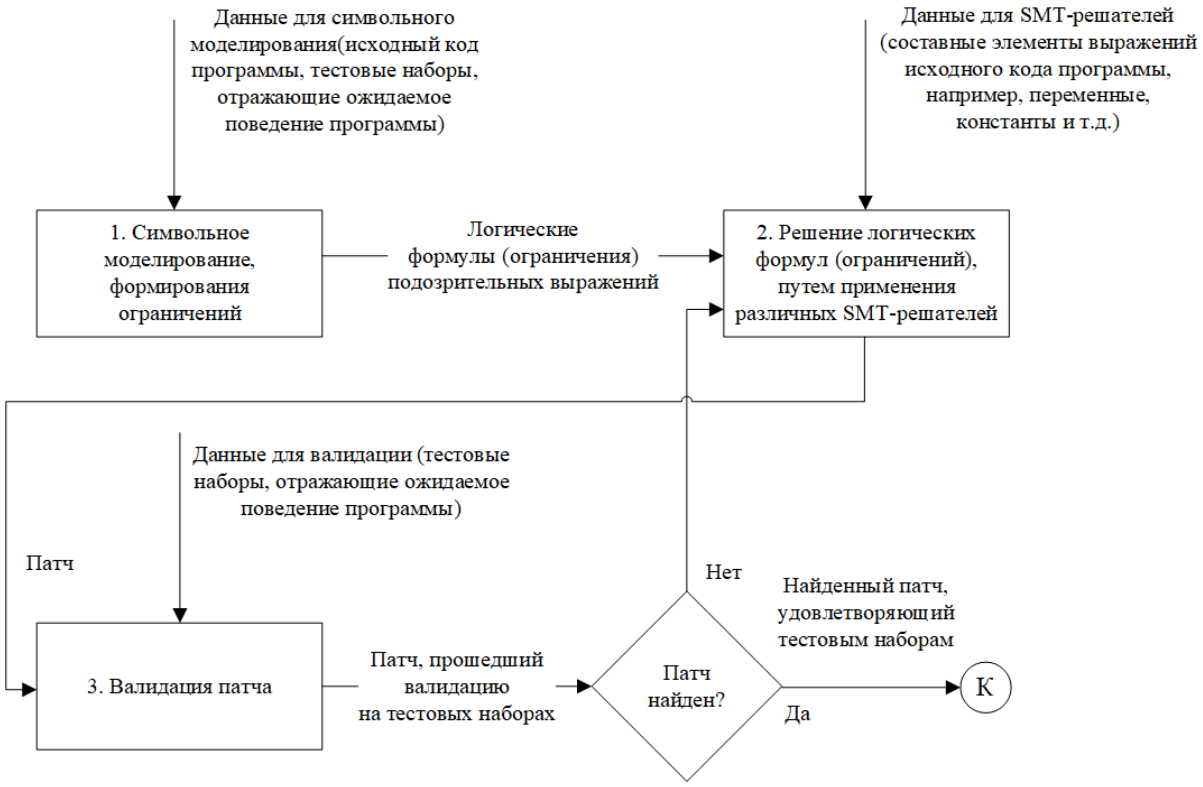


Рисунок 1.3. Алгоритм обнаружения и исправления ошибок программного обеспечения, основанный на семантическом подходе

1. Формирование набора ограничений для выражения, содержащего ошибку. Определения данных ограничений производится путем применения методов символического выполнения программ [8] к исходному коду программы и тестовым наборам. Символическое выполнение программ сводится к замене

подозрительных выражений или частей выражения на переменные. Далее запускается моделирование исходного кода программы и зная входные и выходные данные программы производится вычисление ожидаемого значения переменных. Таким образом формулируются логические формулы (ограничения) подозрительных выражений, выполнение которых приводит к ожидаемому состояние программы в зависимости от входных и выходных данных в соответствии с тестовыми сценариями.

2. Для получения патча производится решение логических формул (ограничений), полученных на шаге 1, путем применения различных SMT-решателей (SMT solvers) [9], использующих составные элементы выражений исходного кода программы (например, переменные, константы и т.д.).
3. Производится валидация полученного патча. В случае наличия ошибок при выполнении программы с патчем алгоритм переходит на шаг 2, пока не будет найден удовлетворяющий тестовым наборам патч.

По результатам приведенных экспериментальных результатов в работах, описывающих существующие методы обнаружения и исправления ошибок программного обеспечения, можно сформулировать следующие критерии для проведения их сравнительного анализа:

1 **Язык программирования.** Оценивается количество языков программирования, с которыми может работать метод обнаружения и исправления ошибок программного обеспечения. Чем больше языков программирования доступно для работы с методом обнаружения и исправления ошибок программного обеспечения, тем лучше.

**2 Типы ошибок.** Оценивается количество и качество ошибок, с которыми может работать метод обнаружения и исправления ошибок программного обеспечения. Чем больше распространенных ошибок доступно для исправления методом обнаружения и исправления ошибок программного обеспечения, тем лучше.

**3 Количество строк исходного кода программы.** Оценивается количество строк исходного кода программ, на которых проводились эксперименты методов обнаружения и исправления ошибок программного обеспечения. По данному

параметру можно судить о применимости предложенного авторами метода к реальным программам, а также достаточности исходных данных эксперимента для подтверждения гипотезы. Чем больше количество строк исходного кода программ было обработано методом обнаружения и исправления ошибок программного обеспечения, тем лучше.

4 **Количество исправляемых ошибок.** Оценивается количество ошибок программы, для которых должно быть найдено исправление в рамках предложенного метода обнаружения и исправления ошибок программного обеспечения. По данному параметру можно судить о применимости предложенного авторами метода к реальным программам, а также достаточности исходных данных эксперимента для подтверждения гипотезы. Чем больше количество исправляемых ошибок было обработано методом обнаружения и исправления ошибок программного обеспечения, тем лучше. Данный параметр введен в дополнение к параметру Количество строк исходного кода программы, так как в экспериментальных результатах представленных авторами методов обнаружения и исправления ошибок программного обеспечения не всегда отражается последний.

5 **Показатель успеха (процент исправленных ошибок).** Оценивается процент ошибок программы, для которой было найдено исправление в рамках предложенного авторами метода обнаружения и исправления ошибок программного обеспечения. По данному параметру можно судить об успешности применения, предложенного авторами метода обнаружения и исправления ошибок программного обеспечения. Чем больше процент исправленных ошибок, тем лучше.

6 **Скорость выполнения алгоритма (секунд на обработку 1 строки исходного кода программы).** Оценивается время на выполнение алгоритма, предложенного авторами метода обнаружения и исправления ошибок программного обеспечения. Чем меньше время на выполнение алгоритма, тем лучше. Очевидно, что данный показатель зависит не только от оптимальности предложенного авторами

алгоритма, но и от оборудования, на котором проводился эксперимент, но в рамках данного исследования мы не учитываем этот фактор.

## 1.2 Обзор существующих методов обнаружения и исправления ошибок программного обеспечения

В данном разделе рассматриваются существующие методы обнаружения и исправления ошибок программного обеспечения в части примененных принципов, подходов, технологий и показателей эффективности.

### 1.2.1 Методы, использующие генетическое программирование

*GenProg: A Generic Method for Automatic Software Repair [10]*

Метод GenProg (Genetic Program Repair) производит обнаружение ошибок в программном коде путем проведения динамического анализа кода и исправление ошибок в программном коде путем применения алгоритмов генетического программирования к исходному коду программы и тестовым наборам для автоматического создания патчей. Метод GenProg является первой работой, которая показывает применения алгоритмов генетического программирования к исправлению ошибок в реальных программах с задокументированными ошибками, представленные в открытом доступе. GenProg выполняет преобразование исходного кода программы в абстрактное синтаксическое дерево (abstract syntax tree) путем применения утилиты CIL toolkit [11], далее авторы применяет существующие алгоритмы выборки такие как стохастическая универсальная выборка (stochastic universal sampling) [12] и соревновательная выборка (tournament selection) [13] для определения выражений, подходящих для исправления. После к полученным выражениям производится применение операторов мутации и кроссовера для формирования новых экземпляров патчей и их оценка применимости с помощью фитнес функция (Fitness Function). Для более эффективного определения искомого патча авторы используют дельта-отладку (delta debugging) [14], которая определяет наихудший тестовый набор для проверки.

Метод GenProg работает со следующими типами ошибок: бесконечный цикл,

ошибки сегментации, переполнение буфера (5 видов), уязвимость формата строки.

Предложенный авторами метод производит обнаружение и исправление ошибок в программах C. Тест был проведен на 120000 строк кода. Процент успешных обнаружений и исправлений 77 процентов (нет данных по количеству ошибок). Время выполнения - 5704 секунд.

*Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices [15]*

Метод Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем использования эволюционного алгоритма (Evolutionary Algorithm) в программах ASM и ELF. Разработанный авторами метод выполняет стохастическая локализацию неисправностей (Stochastic Fault Localization) путем определения количества выполнений инструкции случайным образом с применением Гауссовская свертки (Gaussian convolution) для подавления шумов, инструкции с наибольшей частотой повторений в отрицательных тестовых наборах являются подозрительными. Далее применяются алгоритмы генетического программирования, такие как операторы кроссовера и мутации. С помощью фитнес функции, получившиеся версия программы проверяются на возможность прохождения всех тестовых наборов. В целях снижения времени выполнения и увеличения размера анализируемой программы авторы применяют собственный разработанный метод распределенного генетического алгоритма (Distributed Genetic Algorithm), который заключается в выполнении методов исправления ошибок в программном коде на нескольких связанных устройствах и обмене между ними информации о результатах вычислений.

Метод Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices работает со следующими типами ошибок: использование буфера локального стека, сегментация, бесконечный цикл, неправильная сортировка дублирующих входных данных, переполнение буфера.

Предложенный авторами метод производит обнаружение ошибок в программах ASM и ELF. Тест был проведен на 117682 строк кода. Процент успешных обнаружений - 70,75 процентов. Время выполнения - 26104,15 секунд.

*relifix: Automated Repair of Software Regressions [16]*

Метод relifix производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем применения алгоритмов программной мутации, основой которых послужила работа R. DeMillo, R. J. Lipton, and F. Sayward. Program mutation: A new approach to program testing [93]. Предложенный авторами метод сконцентрировался на новой области ошибок программного кода - регрессионный ошибки. Авторы вручную обработали 73 реальных регрессионных ошибки из CoREBench benchmarks [17]. Метод relifix производит поиск ошибок путем вычисления рейтинга подозрительности для каждого выражения исходного кода программы используя Ochiai formula [18]. Для выполнения генерации кандидатов патчей путем применения операторов генетического программирования (мутация) авторы использовали clang-mutate tool[1]. В целях сокращения времени на выполнение фитнес функции, предложенный авторами метод, выполняет приоритезацию тестового набора путем разделения на прогрессивный и регрессивный тест.

Метод relifix работает со следующими типами ошибок: ошибки регрессии.

Relifix производит исправление ошибок в программах C. Тест был проведен на 864000 строк кода. Процент успешных исправлений - 65,7 процентов (24/35). Время выполнения не указано.

*ASTOR: A Program Repair Library for Java [19]*

Метод Astor (Automatic Software Transformations for program Repair), является улучшением существующих методов автоматического исправления ошибок программного кода GenProg [10], Kali[20] и MutRepair [21], основанные на

технологии генетического программирования. Методы GenProg, Kali и MutRepair авторы адаптировали под использование приложений Java. В разработанном методе авторы заменили оригинальный модуль обнаружения ошибок GenProg на Ochiai [22] и в некоторых случаях авторы исключили из оригинального метода применение оператора кроссовера для генерации патчей.

Данный метод производит обнаружение и исправление ошибок в программах Java. Тест был проведен на - нет данных тыс. строк кода. Процент успешных обнаружений и исправлений - 14,7 процентов (33 из 224 ошибок). Время выполнения - 48582 секунд.

*History Driven Program Repair [23]*

Метод history-based program repair производит исправление ошибок в программном коде путем извлечения шаблонов исправлений из исторических данных исправлений различных проектов и применение данных шаблонов для ранжирования, полученных с помощью генетических алгоритмов, кандидатов на исправление. К особенностям разработанного авторами метода можно отнести способность работать с ошибками, исправление которых требует изменение в нескольких строках исходного кода, а также использование сотен ревизионных систем для формирования корректных патчей. Предложенный авторами метод производит извлечение данных об исправлениях из сервиса Github. В рамках разработки метода авторы руководствовались принципами Ray et al. [24] для извлечения данных из крупных репозиториев с открытым кодом Java проектах. Для извлечения данных авторы использовали Github Archive[2] как базу данных, содержащую информацию об активностях разработчиков. Далее извлеченные данные с Github преобразовывались в форму абстрактных синтаксических деревьев (Abstract Syntax Tree (AST путем применения утилиты GumTree [3], которая производит сравнение двух AST, описывающих исходный код программы, содержащий ошибку и исправленную версию, с целью формирования данных об исправлениях. Для вычисления степени распространенности шаблонов исправления ошибок авторы применяли утилиту

gSpan[4]. Для формирования кандидатов патчей авторы используют операторы мутации из существующих методов исправления ошибок в программном кода, использующих алгоритмы генетического программирования, такие как GenProg[10] и PAR [25] и ранжируют полученные результаты по степени распространенности шаблонов исправления ошибок. На окончательном шаге авторы выполняют валидацию полученных патчей путем запуска тестовых наборов.

Метод history-based program repair работает со следующими типами ошибок: нет данных.

History-based program repair производит исправление ошибок в программах Java (по аналогии с GenProg, PAR). Тест был проведен на 5139 тыс. строк кода. Процент успешных исправлений - 25,5 процентов (23/90). Время выполнения 27600 секунд.

[1] https://github.com/eschulte/clang-mutate

[2] https://githubarchive.org

[3] https://github.com/GumTreeDiff/gumtree

[4] https://www.cs.ucsb.edu/ xyan/software/gSpan.htm

### 1.2.2 Методы, использующие машинное обучение

*R2Fix: Automatically Generating Bug Fixes from Bug Reports [26]*

Метод R2Fix производит исправление ошибок в программном коде путем анализа отчетов об ошибках из репозиториев с открытым кодом и применения шаблонов исправлений. Для решения задачи классификации отчетов об ошибках авторы применяют методы дерево решений (Decision Tree), опорных векторов (Support Vector Machine (SVM)), и байесовской логистической регрессии (Bayesian Logistic Regression (BLR)). Для генерации патчей предложенный авторами метод выполняет поиск исправлений в полученных классифицированных отчетов об ошибках из репозиториев по определенным параметрам (имя файла, имя функции, имя указателя и т.д.), а также применяет утилиту Coccinelle [27].

Метод R2Fix работает со следующими типами ошибок: переполнение буфера, пустой указатель, утечки памяти.

Предложенный авторами метод производит обнаружение и исправление ошибок в программах C/C++. Тест был проведен на 17300000 строк кода. Процент успешных обнаружений и исправлений 71,3 процентов (48 из 90 ошибок). Время выполнения - нет данных.

*Data-Guided Repair of Selection Statements [28]*

Метод Data-Guided Repair of Selection Statements производит исправление ошибок в программном коде за счет определения корректного поведения системы, основанного на применении методов машинного обучения и комбинаторного поиска. Предложенный авторами метод выполняет прогноз ожидаемых выходных данных программы на основании анализа входных данных, как декартового произведения всех возможных вариаций и применения принципа геометрической близости подозрительных входных значений (которые приводят к отклонению полученных результатов тестовых сценариев относительно выходных значений) к положительным входным значениям (которые приводят к получению положительных результатов тестовых сценариев относительно выходных значений). Для выполнения оценки вероятности авторы использует метод опорных векторов (support vector machines - SVM), а для его верификации задачи выполнимости булевых формул (SAT solver). Далее производится генерация нового (корректного) оператора условия выбора на основании входных и выходных значений путем применения алгоритма машинного обучения ID3 decision-tree learning algorithm [29].

Метод Data-Guided Repair of Selection Statements работает со следующими типами ошибок: недостающие проверки переменных, неправильное значение данных в условиях, неправильные / отсутствующие условия соединения, неправильные / отсутствующие проверки диапазона.

Предложенный авторами метод производит исправление ошибок в программах

SAP (язык ABAP). Количество строк кода не указано. Процент успешных исправлений - 85 процентов (нет данных). Время выполнения 4036,2 секунд.

*Automatic Patch Generation by Learning Correct Code [30]*

Метод Prophet производит обнаружение ошибок программного кода путем применения существующих методов, основанных на динамическом анализе кода, и исправление ошибок в программном коде путем построения вероятностных моделей как результат анализа патчей из открытых репозиториев. Предложенный авторами метод производит поиск ошибок в исходном коде программы путем использования существующих методов [31, 32, 33]. Метод Prophet формирует список кандидатов на исправление (патчей) путем использования существующих подходов таких как схемы трансформации (transformation schemas) и поэтапное исправление ошибок с синтезом условий (staged program repair with condition synthesis), описанные в методе SPR [34]. Следующим шагом выполняется определения свойств кандидатов патчей, например, определение зависимости переменных исходного кода с ошибкой относительно операторов, содержащихся в патче. Далее производится ранжирование кандидатов патчей путем применения обученной вероятностной модели к их свойствам. Обучение вероятностной модели производится путем применения метода градиентного спуска к тренировочным данным, полученным на основании данных успешных патчей, представленных в открытых репозиториях. Заключительным шагом выполняется валидация кандидатов патчей путем проверки на тестовых наборах в порядке вероятности успешности патча.

Метод Prophet работает со следующими типами ошибок: нет данных.

Prophet производит исправление ошибок в программах C (по аналогии с GenProg, SPR, Kali). Тест был проведен на 5139 тыс. строк кода. Процент успешных исправлений - 26 процентов (18/69). Время выполнения - нет данных.

*ELIXIR: effective object oriented program repair [34]*

Метод ELIXIR производит обнаружение ошибок программного кода путем

проведения динамического анализа кода и исправление ошибок в программном коде путем формирования патчей в соответствии с шаблонами и применения к ним инструментов ранжирования в соответствии с контекстом исходного кода программы и данных из отчетов об ошибках. Для поиска ошибок предложенная авторами метод обрабатывает исходный код программы и тестовые сценарии путем использования существующего алгоритма Ochiai [18]. ELIXIR генерирует кандидаты на исправления для всех подозрительных выражений в соответствии со схемами преобразования программ (шаблоны модификации, например, расширение типа с float до double). Далее производится ранжирование полученных кандидатов на исправление путем определения свойств каждого кандидата на исправление (например, контекстное сходство между исходным кодом с ошибкой и исправлением, или отчетом об ошибках из программных репозиториев) и применения обученной модели логистической регрессии. Завершающим этапом производится валидация полученных кандидатов на исправление с наивысшей оценкой релевантности на тестовых наборах.

Метод ELIXIR работает со следующими типами ошибок: неверный тип переменной, неверное возвращаемое значение функции, пустой указатель, выход за пределы массива, некорректный инфикс оператор, некорректные значения выражений в условных операторах, ошибки в вызове методов.

Предложенный авторами метод производит обнаружение и исправление ошибок в программах Java. Тест был проведен на 2087000 строк кода. Процент успешных обнаружений и исправлений 60 процентов (48/80). Время выполнения - нет данных.

### 1.2.3 Методы, использующие семантический подход

*Automated Error Localization and Correction for Imperative Programs [36]*

Метод Automated Error Localization and Correction for Imperative Programs производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем использования алгоритма на основе задач выполнимости формул в теориях

(Satisfiability Modulo Theory (SMT)). Автор провели оценку метода на 8 программах из the Siemens suite [37]. Предложенный авторами метод на первом шаге производит замену правой стороны выражений (right-hand side) операторов присваивания специальной функцией для проведения анализа программы. Далее выполняется символьного моделирования (symbolic execution) программы, результатом которого являются данные, которые приводят исходную программу к отклонениям от спецификации. Далее на основе моделей (Model-Based Diagnosis), представленного в работе R. Reiter [38] производится определение выражений, содержащих ошибку (и). На заключительном этапе на основании диагностических данных с помощью задач выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)) производится подбор подходящих исправлений.

Метод Automated Error Localization and Correction for Imperative Programs работает со следующими типами ошибок: оператор присваивания.

Представленный метод производит обнаружение и исправление ошибок программного кода в программах C. Количество строк кода программ не указано. Процент успешных исправлений не указан. Время выполнения - 1325 секунд.

*Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving [39]*

Методы PHPQuickFix и PHPRepair производит обнаружение ошибок программного кода путем проведения статического и динамического анализа кода и исправление ошибок в программном коде путем решения задач выполнимости булевых формул (SAT). Метод PHPQuickFix, выполняющий точечные изменения в операторах вывода на экран в программах PHP, основанный на статическом анализе кода. Данный метод для каждой строки исходного кода формирует стек каждого открывающего HTML-тега и далее анализирует на пропущенные или лишние HTML-теги. При обнаружении ошибки алгоритм добавляет или убирает необходимый HTML-тег. Метод HPRepair, выполняющий комплексные изменения в операторах вывода на экран в программах PHP, основанный на динамическом анализе кода и

решении задач выполнимости булевых формул. Данный метод выполняет построение трассировки печати, и далее производится построение уравнения ограничения ошибок путем составления уравнения для каждого тестового сценария, где с левой стороны формируется конкатенация маркировок, а с правой стороны ожидаемый результат теста. Каждое уравнение ограничений ошибок решается с помощью инструмента Kodkod [40], основанного на решение SAT-based constraint уравнений, результатом которых являются патчи для исправления ошибок.

Методы PHPQuickFix и PHPRepair работают со следующими типами ошибок: ошибки операторов вывода на экран.

Разработанные авторами методы производят обнаружение ошибок в программах PHP. Тест был проведен на 22799 строк кода. Процент успешных обнаружений и исправлений 86 процентов (525/610). Время выполнения - не указано.

*SemFix: Program Repair via Semantic Analysis [41]*

Метод SemFix производит обнаружение ошибок в программном коде путем проведения статического анализа кода и исправление ошибок в программном коде путем применения задач выполнимости формул в теориях (Satisfiability Modulo Theory (SMT) к решению ограничений, которые формируются на основе исходного кода программы и символьного моделирования (symbolic execution). На основе исходного кода программы и тестовых примеров производится статистический поиск ошибок путем применения технологии Tarantula [42]. Далее для каждой строки исходного кода программы с ошибкой формируется функция зависимостей, при которых тестовый набор проходит без ошибок. Для построения подобных функций используется подходы символьного моделирования путем применения технологии KLEE [8]. На заключительном этапе SemFix выполняет программный синтез патчей за счет применения задач выполнимости формул в теориях (SMT) для решения функций зависимостей восстановления путем применения технологии Z3 SMT solver [9].

Метод SemFix работает со следующими типами ошибок: некорректные константы, некорректные операторы сравнения, некорректные логические операторы, некорректные арифметические операторы, отсутствие кода, избыток кода.

SemFix производит обнаружение и исправление ошибок в программах C. Тест был проведен на 10585 строк кода. Процент успешных обнаружений и исправлений 53,33 процентов (48 из 90 ошибок). Время выполнения - 500 секунд на 90 ошибок.

*Automatic Patch Generation Learned from Human-Written Patches [25]*

Метод Pattern-based Automatic program Repair (PAR) производит обнаружение ошибок в программном коде путем проведения динамического анализа кода и исправление ошибок в программном коде путем использования алгоритма, основанного на применении шаблонов исправления, полученных путем анализа существующих патчей программных ошибок. На первом этапе PAR выполняет поиск ошибок в исходном коде и составляет рейтинг подозрительности путем применения существующего алгоритма поиска ошибок [43]. Далее разработанный авторами метод генерирует кандидатов на исправление ошибки путем применения шаблонов исправлений, полученных на основе анализа существующих патчей программных ошибок (был проведен анализ 65 тыс 626 патчей). Оценка применимости патча производится PAR с помощью фитнес-функции путем выполнения тестового набора для каждого сгенерированного кандидата патча. Для выполнения данного анализа автора метода адаптировали существующие методы [43, 44].

Метод PAR работает со следующими типами ошибок: нет данных.

PAR производит исправление ошибок в программах Java. Тест был проведен на 483004 строк кода. Процент успешных исправлений - 22.6 процентов (27/119). Время выполнения не указано.

*MintHint: Automated Synthesis of Repair Hints [45]*

Метод MintHint производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем применения алгоритма на основе статистического корреляционного анализа (statistical correlation analysis) для определения выражений, которые могут появится в исправленной версии программы, и шаблонов исправлений для формирования подсказок. Метод MintHint, который объединяет в себе символьный, статистический и семантические подходы для формирования патчей. На основании исходного кода программы и тестовых сценариев MintHint производит поиск ошибок и составляет рейтинг подозрительных выражений путем применения алгоритма Ochiai, реализованного в утилите Zoltar tool [46]. Далее для каждого подозрительного выражения MintHint формирует функцию, называемую трансформатором состояний, описывающую соответствие входных и выходных значений путем выполнения динамического символического выполнение и решения ограничений KLEE [8]. Далее MintHint производит поиск выражений в исходном коде программы, с вычислением коэффициента корреляции ожидаемым выражениям с учетом ограничений входных и выходных данных трансформаторов состояний путем применения метода the Spearman coefficient and the Spearman partial correlation coefficient [47]. На заключительном этапе предложенный авторами метод выполняет синтез подсказок на исправление ошибок, ранжируя их в соответствии с пороговым значением коэффициента корреляции выходных значений трансформаторов состояний путем применения метода pattern matches [48].

Метод MintHint работает со следующими типами ошибок: отсутствие выражения, некорректные константы, некорректные переменные, некорректные константы, ложные выражения, фильтрация корректных операторов, один или несколько вхождений.

MintHint производит подсказки для исправления ошибок в программах C. Тест был проведен на 1307 строк кода. Процент успешных исправлений - 73 процентов (19/26). Время выполнения - нет данных.

*Vejovis: suggesting fixes for JavaScript faults [49]*

Метод Vejovis производит обнаружение ошибок в программном коде путем использования комбинации статистического и динамического анализа кода и исправление ошибок в программном коде путем применения алгоритма, разработанного на основе решений строковых ограничений (string constraint solver) для DOM JavaScript ошибок (DOM-based JavaScript faults). На первом этапе Vejovis производит поиск в тексте исходного кода программы участков кода, в которых возможны симптомов ошибок путем применения приложения JavaScript RHINO и CRAWLJAX [50]. Далее предложенный авторами метод выполняет анализ симптомов, путем применения метода решения строковых ограничений HAMPI [51], и формирует текстовые сообщения программисту с предложением о предлагаемом методе исправления на основании заранее определенных шаблонов.

Метод Vejovis работает со следующими типами ошибок: некорректное переменная, выражение, некорректный индекс массива, пустая ссылка.

Vejovis производит обнаружение и исправление ошибок программного кода в программах JavaScript. Количество строк кода программ не указано. Процент успешных исправлений - 91 процент (20/22). Время выполнения - 528,1 секунда.

*Automated Fixing of Programs with Contracts [52]*

Метод AutoFix производит обнаружение ошибок программного кода путем проведения статического и динамического анализа кода и исправление ошибок в программном коде путем использования логики предикатов, а также с использование Эйфелевой программы (Eiffel program). Авторы AutoFix для получения тестовых наборов использую существующую систему с открытым кодом The Eiffel verification environment. Далее предложенный авторами метод на основании тестовых примеров с использованием Daikon [53] и собственным алгоритмом формируют массив снимков (snapshots), описывающие с помощью логики предикатов поведение анализируемой программы, а также успешность ее завершения для каждого участка

исходного кода. На основании полученных данных производится поиск ошибок путем выполнения статического и динамического анализа в соответствии с контрактами (Contracts) [54], которые описывают спецификацию каждого класса Eiffel программ и тестовыми наборами. На заключительном этапе метод AutoFix формирует патчи путем подбора значений предиката определенных типов: булево, число, ссылка (подбор ссылки производится на основании построенной модели поведения класса, полученной из абстракции предикатов) или определения правильного значения выражения путем подбора выражений определенных типов: булево (добавление отрицания), число (добавление значения +1/-1 к выражению).

Метод AutoFix работает со следующими типами ошибок: нет данных.

Данный метод производит обнаружение ошибок в программах Eiffel. Тест был проведен на 72920 строк кода. Процент успешных обнаружений- 42 процентов (86/204). Время выполнения - 49884 секунд.

*Safe Memory-Leak Fixing for C Programs [55]*

Метод Leak Fix осуществляет обнаружение ошибок программного кода, связанных с утечками памяти, путем проведения статического анализа и исправление ошибок программного кода путем проведения анализа указателей через развитие существующего подхода single static assignment form (SSA). На первом шаге Leak Fix формирует граф для исходного кода программы путем применения подхода single static assignment form (SSA). Далее выполняется идентификация процедур по следующей классификации: процедуры выделения, использования и освобождения памяти. На заключительном этапе предложенный авторами метод выполняет обнаружение и исправление утечек памяти путем проведения внутрипроцедурного анализа, который включает в себя несколько подходов. Прямой анализ потока данных, т.е. определение необходимости освобождения памяти до или в каждом его узле графа. Обратный анализ потока данных, в рамках которого производится обратный анализ графа на предмет необходимости освобождения памяти после или в каждом его узле. Обход ребер графа, в рамках которого производится анализ ребер

графа на предмет необходимости освобождения памяти для каждой переменной. Прямой, жадный алгоритм, который производит поиск участков в графе для наиболее оптимального места для освобождения памяти. Далее производится сопоставление полученных данных в виде участков графа и переменных для освобождения памяти с исходным кодом программы для последующей генерации исправлений.

Разработанный авторами метод работает со следующими типами ошибок: утечки памяти.

Leak Fix производит исправление ошибок в программах C. Тест был проведен на 522 тыс. строк кода. Процент успешных исправлений - 28 процентов (25/89). Время выполнения 299.62 секунд.

*Staged Program Repair with Condition Synthesis [34]*

Метод SPR (staged program repair) производит обнаружение ошибок программного кода путем применения существующих методов, основанных на динамическом анализе кода и исправление ошибок в программном коде путем применения алгоритма поэтапного определения исправления (staged repair algorithm), который позволяет эффективно производить поиск данных в исходном коде программы для последующего синтеза условий (condition synthesis), согласно которым производится формирование патчей. Предложенный авторами алгоритм производит поиск ошибок в исходном коде программы путем использования существующих алгоритмов [31, 32, 33]. Далее SPR производит трансформацию исходного кода программы в соответствии с заранее определенными схемами, такими как схема добавления или удаления абстрактного условия оператор if, добавление абстрактного условия, добавление нового оператора потока управления (return, break, goto), добавление оператора инициализации памяти, замены значений в операторах, функциях и т.д. На заключительном этапе предложенный авторами метод генерирует исправления согласно схемам трансформации, а для каждого абстрактного условия формирует последовательность значений, при которых достигается успешное прохождение всех тестовых сценариев.

Разработанный авторами метод работает со следующими типами ошибок: нет данных.

SPR производит исправление ошибок в программах C.

Тест был проведен на 5139000 строк кода. Процент успешных исправлений - 28 процентов (19/69). Время выполнения 323820 секунд.

*DirectFix: Looking for Simple Program Repairs [56]*

Метод DirectFix производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем использования задач выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)). На первоначальном этапе предложенный авторами алгоритм производит трансформацию исходного кода программы в формулы трассировки. Формула трассировки - логическая формула, которая описывает алгоритм исходного кода программы, со всеми переменными, связями между ними, выражениями и ожидаемым возвращаемыми результатами. Для трансформации исходного кода программы в формулу трассировки авторы используют существующие утилиты VCC [57] и Boogie [58]. Далее DirectFix генерирует условия исправлений, содержащие семантику программы и ожидаемые возвращаемые значения программы, а также описывают структуры выражений (условные операторы и операторы присваивания) исходного кода программы используя локальные переменные. На следующем шаге метода формируется модель программы в виде измененной формулы трассировки, которая удовлетворяет всем полученным условиям исправлений путем применения pMaxSMT (Partial Maximum Satisfiability) решатель, который реализован на основании Z3 [9] и ориентированный на алгоритм Fu and Malik [59]. На заключительном этапе производится формирование патча путем применения функции Lval2Prog [60] к полученной модели после применения решателя pMaxSMT.

Разработанный авторами метод работает со следующими типами ошибок: условные операторы и операторы присваивания.

DirectFix производит исправление ошибок в программах C. Тест был проведен на 4128 строк кода. Процент успешных исправлений - 59 процентов (58/98). Время выполнения 1000 секунд.

*CLOTHO: saving programs from malformed strings and incorrect string-handling [61]*

Метод CLOTHO производит обнаружение ошибок в программном коде путем проведения статического и динамического анализа кода и исправление ошибок в программном коде путем определения статических и динамических коллекций ограничений для ошибок в строковых объектах (String object) или аргументах функций API. На первичном этапе предложенный авторами метод производит анализ исходного кода программы с целью определения минимального набора исправлений, которые не приведут к изменению логики программы. Далее CLOTHO проводит различного рода анализ минимального набора исправлений исходного кода. Анализ испорченности, который заключается в исключении выражений из последующего анализа, которые ссылаются на методы по работе с входящими/исходящими данными, с такими как системные файлы, консоль, сеть и графический пользовательский интерфейс (file system, console, network, and GUI). Для выполнения данного анализа авторы используют фреймворк INFOFLOW[1]. Анализ графа вызова, который производит анализ исходного кода на предмет наличия блоков исключений (try-catch block), которые уже добавили разработчики с целью устранения ошибок. Данные участки кода также исключаются из последующего анализа. Для данного анализа авторы используют фреймворк SOOT [2]. Анализ связанных определений, в рамках которого из дальнейшего анализа исключаются выражения, уже участвующие в исправлении программного кода. Далее предложенный авторами метод выполняет сбор всевозможных ограничений, которые должны выполняться в исправленной версии программы, такие как: минимальная длина строковых объектов, максимальная длина строковых объектов, префикс

строковых объектов и т.д., путем проведения статистического и динамического анализа. На заключительном этапе в рамках предложенного авторам метода производится формирование исправление программного кода с учетом определенных ранее ограничений.

Разработанный авторами метод работает со следующими типами ошибок: бесконечный цикл, нулевая строка, ошибка сегментации, переполнение буфера.

CLOTHO производит обнаружение и исправление ошибок программного кода в программах Java. Тест был проведен на 67200 строк кода. Процент успешных исправлений - 73 процента (нет данных). Время выполнения - не указано.

*Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT [62]*

Метод Nopol производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем трансформирования выполнения тестовых наборов в задачи выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)) и в случае нахождения решения обратная трансформация в исходный код программы. Предложенный авторами алгоритм производит анализ исходного кода программы с целью поиска предполагаемого включения патча и ожидаемого возвращаемого значения переменной в данном месте, при котором программа успешно проходит все тестовые наборы. Для осуществления данного алгоритма авторы используют метод замены значения (value replacement) [63], при котором во время выполнения программы одно значение заменяется на другое, при этом на данном этапе полученные данные ранжируются по степени подозрительности с помощью алгоритма Ochiai [22]. Далее Nopol выполняет сбор данных для построения патчей, таких как: значения локальных переменных, значения параметров методов, значения полей и констант, данных ожидаемых результатов тестов, параметры методов (например, size(), isEmpty() и т.д.). На заключительном этапе предложенный авторами метод производит перевод собранных данных для исправлений в SMT формулы. Для выполнения данной операции авторы расширили уже существующие методы, используемые SMT [41, 60].

Получение патча из SMT модели производится путем решения формулы SMT и обратного перевода полученных значений в исходный код программы для формирования патча.

Разработанный авторами метод работает со следующими типами ошибок: условные операторы.

Nopol производит исправление ошибок в программах Java. Тест был проведен на - нет данных строк кода. Процент успешных исправлений - нет данных процентов (нет данных). Время выполнения - нет данных.

*Qlose: Program Repair with Quantitative Objectives [64]*

Метод Qlose (Quantitatively close) производит исправление ошибок в условных операторах и в операторах присваивания  в программном коде путем генерации патчей с помощью Sketch [65] на основе ограничений MAX-SMT, получаемых по принципу минимизации изменений в синтаксической и семантической составляющей исходного кода программы. Предложенный авторами метод производит замену в исходном коде программы всех выражения с условными операторами и операторами присваивания на функцию вычисления нового значения. Функция вычисления нового значения в условном операторе или операторе присваивания генерирует возможные варианты значений используя линейную комбинацию констант и переменных исходного кода программы. Для ограничения получаемых значений предложенный авторами алгоритм использует SMT ограничения. Для реализации данного шага авторы используют систему генерации кода Sketch [65], которая поддерживает алгоритмы решения SMT ограничений. Далее Qlose производит определение синтаксических и семантических различий между исходным кодом программы и получаемым кандидатом на исправление. На заключительном этапе предложенный авторами метод выбирает исправления, которое удовлетворяет всем тестовым наборам и содержат минимальное значения расстояния, т.е. минимальное изменения в синтаксической и семантической составляющей исходного кода.

Разработанный авторами метод работает со следующими типами ошибок: условные операторы и операторы присваивания.

Qlose производит исправление ошибок в программах C#. Тест был проведен на 81 строк кода. Процент успешных исправлений - 100 процентов (нет данных). Время выполнения 82,3 секунд.

*Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis [66]*

Метод Angelix производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем использования задач выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)). Предложенный авторами алгоритм выполняет преобразование исходного кода программы путем добавления условного оператора (if(1)) к выражениям, которые представлены в исходном коде программы без условных операторов. Далее Angelix выполняет статистический поиск ошибок в выражениях исходного кода программы путем применения Jaccard formula [67]. На следующем этапе данного метода производится формирование ограничений исправлений (repair constraint) на основании исходного кода программы, тестовых наборов и данных о выражениях, содержащих ошибку. Для формирования ограничений исправлений предложенный авторами алгоритм выполняет управляемое символьное выполнение программы (controlled custom symbolic execution), в рамка которого произодится неполное выполнение программы на основании тестовых сценариев с заменой подозрительных выражений на символьные переменные. За счет данного подхода авторы заявляют, что их алгоритм производит символьное выполнение программы независимо от размеров исходного кода программы. Для выполнения символьного выполнения программы авторы реализовали свой алгоритм на основании KLEE [8]. На завершающем этапе Angelix производит синтез патчей путем применения component-based repair synthesis algorithm (CBRS) [56]. Далее CBRS формирует патч путем применения решателя MaxSMT к полученным ограничениям с целью получения минимальных изменений исходного кода программы.

Разработанный авторами метод работает со следующими типами ошибок: нет данных.

Angelic производит исправление ошибок в программах C. Тест был проведен на 4573 тыс. строк кода. Процент успешных исправлений - 87,5 процентов (28/32). Время выполнения 7020 секунд.

*JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder [68]*

Метод JFIX производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем использования задач выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)). Метод JFIX был протестирован на 47 ошибках из the IntroClass benchmark [69]. На первичном этапе предложенный авторами метод производит поиск ошибок в исходном коде программы путем применения существующего алгоритма Ochiai [18]. В найденных ошибках исходного кода программы JFIX производит установку символьных переменных взамен найденных ошибок. Далее при помощи существующего алгоритма Symbolic PathFinder [70] производится выполнение программы через тестовые сценарии с целью формирования условий пути, в которых достигается отсутствие ошибок во всех тестовых сценариях. Из полученных условий пути формируются спецификации для последующего формирования патчей. Далее предложенный авторами метод формирует патчи для исправления исходного кода программы путем обработки полученных ранее спецификаций с использованием алгоритма SyGuS solvers [71]. Также JFIX способен переводить данные из полученных на предыдущем шаге спецификаций для использования в других сторонних алгоритмах генерации патчей, таких как Angelix [66], Enumerative и CVC4 [72]. На завершающем этапе производится валидация полученных патчей путем прохождения тестовых наборов.

JFIX работает со следующими типами ошибок: нет данных.

Данный метод производит обнаружение ошибок в программах Java. Тест был проведен на 175000 строк кода. Процент успешных обнаружений и исправлений 100 процентов (47/47). Время выполнения - 240 секунд.

*CRSearcher: Searching Code Database for Repairing Bugs [73]*

Метод CRSearcher производит обнаружение ошибок программного кода путем проведения статического анализа кода и исправление ошибок в программном коде путем анализа репозиториев с открытым кодом с использованием комбинированного подхода: статический анализ кода, анализ через Абстрактное синтаксическое дерево (Abstract Syntax Tree (AST) и алгоритмы поиска плагиата Running Karp-Rabin Greedy-String-Tiling Algorithm (RKR-GST). Авторы используют существующий статический анализатор JAVA кода FindBugs [3] для поиска мест в исходном коде программы, который содержит ошибки. Для найденных ошибок CRSearcher производит выбор диапазона соседних строк кода, которые связаны с выражением, содержащим ошибку. Выбор диапазона необходим для последующего поиска похожего исходного кода программы в репозиториях с открытым кодом для последующего поиска патча. Для определения диапазона авторы прибегают к синтаксическому анализу и анализу через Абстрактное синтаксическое дерево (Java Abstract Syntax Tree (AST)) используя синтаксический анализатор Eclipse JDT8[4]. Далее предложенный авторами метод производит поиск похожего исходного кода в репозиториях на исходный код выбранного диапазона с ошибкой используя алгоритм на основе маркеров Running Karp-Rabin Greedy-String-Tiling Algorithm [74]. Следующим этапом авторы используют существующий статический анализатор JAVA кода FindBugs [5] для определения релевантности найденных фрагментов исходных кодов программ в репозиториях к строкам исходного кода программы, содержащей ошибку. На завершающем этапе CRSearcher предлагает на выбор список подходящих фрагментов исходных кодов программ из программных репозиториев к строкам исходного кода

программы, содержащей ошибку для проведение последующего анализа программистом.

Разработанный авторами метод работает со следующими типами ошибок: переполнение целого числа, ссылка на нулевой указатель, ошибка в условном операторе, неверный параметр в методе, неверная последовательность операторов

Разработанный авторами метод производит обнаружение и исправление ошибок в программах Java. Тест был проведен на - нет данных тыс. строк кода. Процент успешных обнаружений и исправлений - нет данных процентов. Время выполнения - нет данных.

*Semantic program repair using a reference implementation [75]*

Метод SemGraft производит обнаружение ошибок программного кода путем проведения статистического анализа кода и исправление ошибок в программном коде путем путем использования задач выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)). Для поиска ошибок предложенная авторами метод обрабатывает исходный код программы и тестовые сценарии путем использования статистического поиска ошибок существующего алгоритма [76]. В найденных ошибках исходного кода программы SemGraft производит установку символьных переменных взамен найденных ошибок. Далее производится символьное выполнение программы с найденными ошибками и эталонной программы. Результатом данного шага становится конечные условия пути и символьные выходные состояния для программы с найденными ошибками и эталонной программы. Из полученных данных формируются спецификации для последующего формирования патчей. На следующем шаге SemGraft формируются условия проверки, которые включают в себя пары конечных условий и символьных выходных состояний программы с найденными ошибками и эталонной программы. Данные пары условий пути и символьных выходных состояний определяются по следующему принципу: условие пути программы с найденными ошибками и эталонной программы должны коррелировать, но символьные выходные состояния должны различаться. Далее в

соответствии с условиями проверки с использованием задач выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)) формируется контрпример входных данных. На основании контрпримера входных данных и условия пути эталонной программы определяется корректное символьное выходное состояние, по которому определяется соответствующее условие пути программы с найденными ошибками. Далее для данного условия пути программы с найденными ошибками и контрпримера входных данных с помощью существующего метода Angelix [66] извлекается Angelic forest как контрпример входных данных и выходное значение для строки кода с ошибкой. На завершающем этапе с помощью существующего алгоритма Angelix [66] использующего задачи выполнимости формул в теориях (Satisfiability Modulo Theory (SMT)) формируется экземпляр исправления ошибки программы.

Разработанный авторами метод работает со следующими типами ошибок: нет данных.

Данный метод производит обнаружение и исправление ошибок в программах C. Тест был проведен на 4573000 строк кода. Процент успешных обнаружений и исправлений - 100 процентов (нет данных). Время выполнения - 16200 секунд.

*Static automated program repair for heap properties [77]*

Метод FootPatch производит исправление ошибок в программном коде путем использования логики разделение (Separation Logic). На первичном этапе FootPatch переводит исходный код программы в язык логики предикатов SmallfootIntermediate Language [78] описывающий семантику анализируемой программы. Далее используя формальную систему Логики разделения (Separation logic), которая является расширением Логики Хоара (Hoare logic) предложенный авторами метод производит поиск спецификаций на исправление, которые описывают необходимое изменение в логике поведения программы. На следующем этапе предложенный авторами метод трансформирует полученные спецификаций на исправление из языка логики предикатов, описывающий семантику программы, на язык исходного кода программы. Далее использую уже полученную семантическую модель исходной

программы предложенный авторами метод применяет найденное ошибочное состояние программы для использования в статическом анализе для поиска места ошибки. На завершающем этапе FootPatch производит проверку совместимости патчей двумя способами. Первый способ заключается в проверке совместимости типов переменных, используемых в патче и исходном коде программы. Второй способ заключается в повторной компиляции исходного кода программы для исключения синтаксических ошибок в примененном патче.

Разработанный авторами метод работает со следующими типами ошибок: утечка ресурсов, утечка памяти, несуществующая ссылка.

Разработанный авторами метод производит обнаружение и исправление ошибок в программах C, C++, ObjectiveC, Java. Тест был проведен на 721700 строк кода. Процент успешных обнаружений и исправлений - 37,24 процентов (54/145). Время выполнения - 2338 секунд.

*Automated program repair with canonical constraints [79]*

Метод Canonical Search And Repair (CSAR) производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем применения канонизированных ограничений (canonicalized constraints). Авторы используют существующие алгоритмы поиска ошибок, например, на базе семантического подхода Tarantula [80]. Далее CSAR для найденных ошибок производит символическое выполнение программы для получения условий пути с описанием корректной семантики программы. Авторы данной работы для получения условий пути использовали Java Symbolic Pathfinder [70]. На следующем шаге предложенный авторами метод конвертирует условия пути в нормальную форму путем применения метода Green [81]. Предложенный авторами метод применяет строковые метрики к нормализованным условиям пути и далее классифицируют полученные результаты с целью сокращения количества потенциальных патчей применяя подходы, описанные в работе [82]. На завершающем этапе CSAR выполняет переименование переменных в полученных патчах для

группировки возможных патчей и валидирует полученные варианты патчей через юнит-тестирование.

Разработанный авторами метод работает со следующими типами ошибок: нет данных.

Данный метод производит обнаружение и исправление ошибок в программах Java. Тест был проведен на - нет данных тыс. строк кода. Процент успешных обнаружений и исправлений - 66,66 процентов (214/354). Время выполнения - нет данных.

*Context-aware patch generation for better automated program repair [83]*

Метод CapGen производит обнаружение ошибок программного кода путем проведения динамического анализа кода и исправление ошибок в программном коде путем построения абстрактных синтаксических деревьев (Abstract Syntax Tree (AST). Предложенный авторами метод производит обработку исходного кода путем проведения анализа через абстрактные синтаксические деревья (Abstract Syntax Tree (AST)). Авторы используют существующий анализатор кода GZoltar для поиска узлов в дереве AST программы, которая содержит ошибки. Для получения пространства ошибок и подозрительных значений авторы улучшили анализатор кода GZoltar путем применения алгоритма Ochiai [84]. Для найденных узлов дерева AST, содержащих ошибки, CapGen производит определение возможных операций мутаций (замена, вставка или удаление) для различных типов узлов с индексом вероятности. Индекс вероятности для операции мутации определяется путем поиска для каждого типа узла и его места в структуре дерева AST похожих ошибок в базе данных патчей [23]. Далее предложенный авторами метод производит поиск значений для операций мутаций в дереве AST исходной программы с составлением индекса схожести. Индекс схожести определяется путем поиска структурно похожих фрагментов исходного кода, которые могут быть использованы как значения для операций мутаций, путем поиска значений, содержащих одинаковое имя и тип переменной в похожих фрагментах исходного кода, которые могут быть использованы как значения для операций

мутаций, а также путем поиска значений, содержащих одинаковое влияние или зависимость от узлов дерева AST. На завершающем этапе для каждого подозрительного узла дерева AST генерируется патч на основе данных возможных операций мутации и возможных значений. Далее составляется рейтинг патчей согласно индексу подозрительности узлов дерева AST, индекса вероятности применения операции мутации и индекса схожести значений.

Разработанный авторами метод работает со следующими типами ошибок: нет данных.

Данный метод производит обнаружение и исправление ошибок в программах Java. Тест был проведен на 231000 строк кода. Процент успешных обнаружений и исправлений - 84 процентов (22/25). Время выполнения - 13047,6 секунд.

[1] https://github.com/secure-software-engineering/soot-infoflow

[2] http://www.sable.mcgill.ca/soot/

[3]http ://findbugs.sourceforge.net/

[4]http://www.eclipse.org/jdt/

[5]http ://findbugs.sourceforge.net/

### 1.2.4 Сравнительный анализ методов обнаружения и исправления ошибок программного обеспечения

В целях выяснения современного состояния развития методов обнаружения и исправления ошибок программного обеспечения был произведен сравнительный анализ, результаты которого представлены в таблице 1.1.

Таблица 1.1 Сравнительная таблица методов обнаружения и исправления ошибок программного обеспечения

| Метод | Класс метод | Оценка по параметрам | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | a | | | | | | |
| GenProg: A Generic Method for Automatic Software Repair | Genetic programming | 1 (C) | 4 (бесконечный цикл, ошибки сегментации, переполнение буфера, уязвимость формата строки) | 120000 | 0 (нет данных) | 77 | 0,047533 |
| Automated repair of binary and assembly programs for cooperating embedded devices | Genetic programming | 1 (ASM) | 4 (использование буфера локального стека, сегментация, бесконечный цикл, неправильная сортировка дублирующих входных данных, переполнение буфера) | 31862 | 0 (нет данных) | 70,75 | 0,819288 |
| Relifix: automated repair of software regressions | Genetic programming | 1 (C) | 0 (нет данных) | 864000 | 35 | 65,7 | 0 (нет данных) |
| ASTOR: A Program Repair Library for Java | Genetic programming | 1 (Java) | 0 (нет данных) | 0 (нет данных) | 224 | 14,7 | 0 (нет данных) |
| History Driven Program Repair | Genetic progra | 1 (Java) | 0 (нет данных) | 0 (нет данны | 90 | 25,5 | 0 (нет данны |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | mming | | | x) | | | x) |
| R2Fix: Automatically Generating Bug Fixes from Bug Reports | Machine learning | 2 (C/C++) | 3 (переполнение буфера, пустой указатель, утечки памяти) | 17300000 | 80 | 71,3 | 0 (нет данных) |
| Data-guided repair of selection statements | Machine learning | 1 (ABAP) | 4 (недостающие проверки переменных, неправильное значение данных в условиях, неправильные / отсутствующие условия соединения, неправильные / отсутствующие проверки диапазона) | 0 (нет данных) | 0 (нет данных) | 85 | 0 (нет данных) |
| Automatic Patch Generation by Learning Correct Code | Machine learning | 1 (C) | 0 (нет данных) | 5139000 | 69 | 26 | 0 (нет данных) |
| ELIXIR: effective object oriented program repair | Machine learning | 1 (Java) | 8 (неверный тип переменной, неверное возвращаемое значение функции, | 2087000 | 80 | 60 | 0 (нет данных) |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | пустой указатель, выход за пределы массива, некорректный инфикс оператор, некорректные значения выражений в условных операторах, ошибки в вызове методов) | | | | |
| Automated error localization and correction for imperative programs | Semantics approach | 1 (C) | 1 (ошибка оператора присваивания) | 0 (нет данных) | 0 (нет данных) | 0 (нет данных) | 0 (нет данных) |
| Automated repair of HTML generation errors in PHP applications using string constraint solving | Semantics approach | 1 (PHP) | 1 (ошибка оператора вывода на экран) | 22799 | 610 | 86 | 0 (нет данных) |
| SemFix: Program Repair via Semantic Analysis | Semantics approach | 1 (C) | 6 (некорректные константы, некорректные операторы сравнения, некорректные логические операторы, | 10585 | 90 | 53,33 | 0,047237 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | некорректные арифметические операторы, отсутствие кода, избыток кода) | | | | |
| Automatic patch generation learned from human-written patches | Semantics approach | 1 (Java) | 0 (нет данных) | 483004 | 119 | 22,6 | 0 (нет данных) |
| MintHint: automated synthesis of repair hints | Semantics approach | 1 (C) | 7 (отсутствие выражения, некорректные константы, некорректные переменные, некорректные константы, ложные выражения, фильтрация корректных операторов, один или несколько вхождений) | 1307 | 26 | 73 | 0 (нет данных) |
| Vejovis: suggesting fixes for JavaScript faults | Semantics approach | 1 (JavaScript) | 4 (некорректная переменная, выражение, некорректный индекс массива, пустая ссылка) | 0 (нет данных) | 22 | 91 | 0 (нет данных) |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Automated Fixing of Programs with Contracts | Semantics approach | 1 (Eiffel) | 0 (нет данных) | 72920 | 204 | 42 | 0,684092 |
| Safe Memory-Leak Fixing for C Programs | Semantics approach | 1 (C) | 1 (утечка памяти) | 522000 | 89 | 28 | 0,000574 |
| Staged Program Repair with Condition Synthesis | Semantics approach | 1 (C) | 0 (нет данных) | 5139000 | 69 | 28 | 0,063012 |
| DirectFix: Looking for Simple Program Repairs | Semantics approach | 1 (C) | 2 (условные операторы, операторы присваивания) | 4128 | 98 | 59 | 0,242248 |
| CLOTHO: saving programs from malformed strings and incorrect string-handling | Semantics approach | 1 (Java) | 4 (бесконечный цикл, нулевая строка, ошибка сегментации, переполнение буфера) | 67200 | 0 (нет данных) | 73 | 0 (нет данных) |
| Automatic Repair of Buggy If Conditions and Missing Preconditions with | Semantics approach | 1 (Java) | 1 (ошибка в условном операторе) | 0 (нет данных) | 0 (нет данных) | 0 (нет данных) | 0 (нет данных) |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| SMT (Nopol) | | | | | | | |
| Qlose: Program Repair with Quantitative Objectives | Semantics approach | 1 (C#) | 2 (условные операторы, операторы присваивания) | 81 | 0 (нет данных) | 100 | 1,016049 |
| Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis | Semantics approach | 1 (C) | 2 (ошибка в вызове функции, нежелательные выражения) | 4573 | 32 | 87,5 | 1,535097 |
| JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder | Semantics approach | 1 (Java) | 0 (нет данных) | 175000 | 47 | 100 | 0,001371 |
| CRSearcher: Searching Code Database for Repairing Bugs | Semantics approach | 1 (Java) | 5 (переполнение целого числа, ссылка на нулевой указатель, ошибка в условном операторе, неверный параметр в методе, неверная последовательность операторов) | 0 (нет данных) | 0 (нет данных) | 0 (нет данных) | 0 (нет данных) |
| Contract-based program repair without the | Semantics approac | 1 (Java) | 0 (нет данных) | 320000 | 31 | 80 | 2,063963 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| contracts | h | | | | | | |
| Semantic program repair using a reference implementation | Semantics approach | 1 (C) | 0 (нет данных) | 4573000 | 0 (нет данных) | 100 | 0,003543 |
| Static automated program repair for heap properties | Semantics approach | 4 (C, C++, ObjectiveC, Java) | 3 (утечка ресурсов, утечка памяти, несуществующая ссылка) | 721700 | 145 | 37,24 | 0,00324 |
| Automated program repair with canonical constraints | Semantics approach | 1 (Java) | 0 (нет данных) | 0 (нет данных) | 354 | 66,66 | 0 (нет данных) |
| Context-aware patch generation for better automated program repair | Semantics approach | 1 (Java) | 0 (нет данных) | 231000 | 25 | 84 | 0,056483 |

Оценку методов обнаружения и исправления ошибок программного обеспечения будем производить по сумме оценок по всем параметрам. Оценку по параметрам примем равной 1, если параметр удовлетворен полностью, 0 – не удовлетворен, от 0 до 1 – удовлетворен частично.

Предложенные параметры для оценки методов обнаружения и исправления ошибок программного обеспечения имеют не одинаковый приоритет. Степень значимости параметров отражают их весовые коэффициенты, приведенные в таблице 1.2.

Таблица 1.2 Весовые значения параметров

| Номер параметра | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Вес параметра | 0,15 | 0,15 | 0,1 | 0,1 | 0,3 | 0,2 |

Суммарный вес параметров равен 1.

На основании весовых значений параметров можно вычислить суммарную оценку каждого из методов обнаружения и исправления ошибок программного обеспечения и произвести их сравнение. Интегральная оценка R вычисляется по следующей формуле:

$$R = \sum_{i=1}^{5} \frac{ki \times pi}{pibest} + \frac{k6 \times p6best}{p6}$$, где

i – номер параметра;

k – вес i-го параметра;

p – оценка метода по i-му параметру;

pbest – лучшая оценка метода по i-му параметру;

Результаты вычислений интегральной оценки методов обнаружения и исправления ошибок программного обеспечения R представлен в таблице 1.3.

Таблица 1.3 Интегральная оценка методов обнаружения и исправления ошибок программного обеспечения

| № | Метод | Класс | Интегральная оценка R |
|---|---|---|---|
| 1 | R2Fix: Automatically Generating Bug Fixes from Bug Reports | Machine learning | 0,458264754 |
| 2 | JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder | Semantics approach | 0,429341479 |
| 3 | Automated repair of HTML generation errors in PHP | Semantics | 0,41438178 |

| | | | |
|---|---|---|---|
| | applications using string constraint solving | approach | 6 |
| 4 | Semantic program repair using a reference implementation | Semantics approach | 0,396113896 |
| 5 | ELIXIR: effective object oriented program repair | Machine learning | 0,392678338 |
| 6 | MintHint: automated synthesis of repair hints | Semantics approach | 0,39201985 |
| 7 | Vejovis: suggesting fixes for JavaScript faults | Semantics approach | 0,389106557 |
| 8 | Static automated program repair for heap properties | Semantics approach | 0,381101988 |
| 9 | Qlose: Program Repair with Quantitative Objectives | Semantics approach | 0,375112667 |
| 10 | Data-guided repair of selection statements | Machine learning | 0,3675 |
| 11 | Safe Memory-Leak Fixing for C Programs | Semantics approach | 0,35646908 |
| 12 | GenProg: A Generic Method for Automatic Software Repair | Genetic programming | 0,346591959 |
| 13 | Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis | Semantics approach | 0,342846598 |
| 14 | CLOTHO: saving programs from malformed strings and incorrect string-handling | Semantics approach | 0,331888439 |
| 15 | SemFix: Program Repair via Semantic Analysis | Semantics approach | 0,327218663 |
| 16 | Automated repair of binary and assembly programs for | Genetic | 0,32507331 |

| | | | |
|---|---|---|---|
| | cooperating embedded devices | programming | 9 |
| 17 | Context-aware patch generation for better automated program repair | Semantics approach | 0,296951923 |
| 18 | Automated program repair with canonical constraints | Semantics approach | 0,295512787 |
| 19 | Contract-based program repair without the contracts | Semantics approach | 0,284486912 |
| 20 | DirectFix: Looking for Simple Program Repairs | Semantics approach | 0,268560027 |
| 21 | Relifix: automated repair of software regressions | Genetic programming | 0,245331925 |
| 22 | Automated Fixing of Programs with Contracts | Semantics approach | 0,19753077 |
| 23 | Staged Program Repair with Condition Synthesis | Semantics approach | 0,164325849 |
| 24 | Automatic Patch Generation by Learning Correct Code | Machine learning | 0,156516678 |
| 25 | CRSearcher: Searching Code Database for Repairing Bugs | Semantics approach | 0,13125 |
| 26 | History Driven Program Repair | Genetic programming | 0,128754098 |
| 27 | Automatic patch generation learned from human-written patches | Semantics approach | 0,127600127 |
| 28 | ASTOR: A Program Repair Library for Java | Genetic programming | 0,118321311 |
| 29 | Automated error localization and correction for | Semantics | 0,05625 |

| | | | |
|---|---|---|---|
| | imperative programs | approach | |
| 30 | Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT (Nopol) | Semantics approach | 0,05625 |

По результатам вычислений наивысший балл получил метод R2Fix: Automatically Generating Bug Fixes from Bug Reports [26] - 0,458264754, далее идут JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder [68] - 0,429341479 и Automated repair of HTML generation errors in PHP applications using string constraint solving [34] - 0,414381786.

Наибольшее число языков программирования, с которыми может работать метод обнаружения и исправления ошибок программного обеспечения показал метод Static automated program repair for heap properties [77], основанный на семантическим подходе (Semantics approach).

Наибольшее количество типов ошибок, с которыми может работать метод обнаружения и исправления ошибок программного обеспечения показал метод ELIXIR: effective object oriented program repair [34], основанный на машинном обучении (Machine learning).

Наибольшее количество строк исходного кода программы, с которыми может работать метод обнаружения и исправления ошибок программного обеспечения показал метод R2Fix: Automatically Generating Bug Fixes from Bug Reports [26], основанный на машинном обучении (Machine learning).

Наибольшее количество ошибок, с которыми может работать метод обнаружения и исправления ошибок программного обеспечения показал метод Automated repair of HTML generation errors in PHP applications using string constraint solving [39], основанный на семантическом подходе (Semantics approach).

Наибольший процент исправленных ошибок показал метод JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder [68], основанный на семантическом подходе (Semantics approach).

Наибольшую скорость выполнения алгоритма обнаружения и исправления ошибок программного обеспечения показал метод Safe Memory-Leak Fixing for C Programs [55], основанный на семантическом подходе (Semantics approach).

Таким образом, применение методов, относящихся к классам, использующим машинное обучение и семантический подход, показывают наилучшие результаты в области обнаружения и исправления ошибок программного обеспечения, развитие и комбинация которых в будущем является наиболее перспективным направлением исследования.

### 1.3 Выводы

В рамках данной главы были сформулированы критерии для проведения сравнительного анализа существующих методов. Далее согласно критериям было отобрано 30 существующих методов автоматического исправления ошибок программного кода и проведен их аналитический обзор. По результатам аналитического обзора существующих методов был проведен их сравнительный анализ и выбраны лучшие мировые практики в данной области.

# 2. ПОСТАНОВКА ЗАДАЧИ И ВЫБОР ПУТИ РЕШЕНИЯ

В этой главе рассматриваются требования к разрабатываемому методу и программному инструментарию в рамках постановки задачи исследования и приводится обоснование выбранного подхода в рамках анализа поставленной задачи и выбора путей ее решения.

## 2.1 Постановка задачи исследования

В рамках проводимого исследования требуется разработать эффективный метод автоматического формированию исправлений ошибок программного кода, использующий современные методы программной инженерии, такие как Data mining, машинное обучение и обладающий следующими требованиями:

- разрабатываемый метод должен производить исправление более одного типа ошибок программного обеспечения за приемлемое время;
- разрабатываемый метод должен производить исправление ошибок программного обеспечения на основе опыта успешных исправлений (патчей) множества других проектов для языка программирования ABAP;
- разрабатываемый метод должен производить исправление ошибок программного обеспечения на основе анализа контекста исходного кода с ошибкой;
- разрабатываемый метод должен производить исправление ошибок программного обеспечения без использования спецификаций и других средств автоматизированной генерации кода;
- разрабатываемый метод должен иметь возможность извлекать данные из успешных патчей любых проектов с открытым кодом и определять их свойства;
- разрабатываемый метод должен иметь возможность обучать модель машинного обучения извлеченными свойствами из успешных патчей;
- разрабатываемый метод должен производить генерацию патчей по заранее определенным шаблонам на основании метаданных исходного кода с ошибкой;

- разрабатываемый метод должен иметь возможность ранжировать сгенерированные патчи путем применения обученной модели машинного обучения на данных свойств успешных патчей.

На основании разработанного метода автоматического формирования исправлений ошибок программного кода требуется разработать программный инструментарий, реализующий разработанный метод и обладающий следующими требованиями:

- программный инструментарий должен быть разработан на современной платформе, использующей принципы объектно-ориентированного программирования;
- программный инструментарий должен иметь понятный и удобный графический интерфейс;
- программный инструментарий должен являться приложением win32 и запускаться без запуска дополнительных программ и выполнять все функции разрабатываемого метода в полном объеме в одном окне графического интерфейса;
- программный инструментарий должен быть разработан без применения платных программных библиотек и других программных продуктов;
- программный инструментарий должен быть с открытым исходным кодом.

### 2.2 Анализ задачи и выбор пути решения

Согласно обзору и анализу существующих методов в области автоматического исправления ошибок программного обеспечения (см. главу 1) можно выделить следующие методы.

Методы, основанные на технологии генетического программирования. Данный класс методов является методом стохастического поиска решения проблем, основанный на идеях эволюционной генетики, который включают в себя генотип (генетический материал индивида), хранящийся в памяти, дифференциальное воспроизводство этих генотипов и вариаций, которые создаются процессами, аналогичными биологическим процессам мутации и кроссовера [6].

Методы, основанные на семантическом подходе. Суть этих методов сводится к определению набора ограничений для выражения, содержащего ошибку путем применения методов символического выполнения программ [8] и решения данных ограничений путем применения различных SMT-решателей [9].

Методы, основанные на классе методов машинного обучения [86]. Суть этих методов сводится к построению моделей машинного обучения [7] на базе исходных кодов программ, содержащих ошибки и их исправления, а также комментарии и другие данные из репозиториев исходных кодов, таких как GitHub и др. Далее обученную модель применяют для задач классификации, например, для решения задач обнаружения ошибок в исходном коде программы или определения подходящих патчей, классифицируемых на общих с ошибкой параметрах.

Основным недостатком методов, основанных на технологии генетического программирования, является случайный подбор всевозможных вариантов патчей без анализа как контекста исходного кода с ошибкой, так и аналогичных патчей. В свою очередь в методах, основанных на семантическом подходе, уже широко производится анализ контекста исходного кода с ошибкой, но никак не используются опыт аналогичных патчей для усиления алгоритма автоматического формирования патчей. В то же время, методы, основанные на классе методов машинного обучения, наиболее близки по реализации к поставленной в работе задаче, так как производят анализ как контекста исходного кода с ошибкой, так и аналогичных патчей.

Таким образом, в рамках проводимого исследования необходимо разработать метод автоматического формирования исправлений ошибок программного кода на основе использования накопленного ранее опыта создания патчей. Необходимо разработать алгоритм, основанный на методах машинного обучения, позволяющий автоматически формировать патчи для различного рода ошибок в программном коде без использования спецификаций и других средств автоматизированной генерации кода.

### 2.3 Выводы

В рамках данной главы была сформулирована задача исследования, и пути ее решения на базе проведенного обзора и анализа существующих методов в области

автоматического исправления ошибок программного обеспечения, которая заключается в разработке метода и программного инструмента, основанного на методах машинного обучения, позволяющего автоматически формировать исправления ошибок программного кода, описание которого приведено в следующих главах.

# 3. ПРОЕКТИРОВАНИЕ МЕТОДА АВТОМАТИЧЕСКОЕ ФОРМИРОВАНИЕ ИСПРАВЛЕНИЙ ОШИБОК ПРОГРАММНОГО КОДА НА ОСНОВЕ АНАЛИЗА ПРОГРАММНЫХ РЕПОЗИТОРИЕВ

В этой главе будет представлена схема и вербальное описание сути разрабатываемого метода согласно поставленной задачи исследования, а также в данной главе будет приведено детальное описание разрабатываемого метода, которое будет включать в себя алгоритмы, математическую модель, используемые технологии и методы.

## 3.1 Схема разрабатываемого метода

Суть предлагаемого метода заключается в автоматическом формировании патчей для ошибок в ABAP-программах путем генерации кандидатов патчей по заранее определенным шаблонам и ранжирования полученных результатов по вероятности успешного применения, определяемой на основании вероятностной модели, полученной с помощью методов машинного обучения. В свою очередь, вероятностная модель формируется за счет обучения на данных успешных и неуспешных патчей ABAP-программ. Основная идея предлагаемого метода представлена на рисунке 3.1.

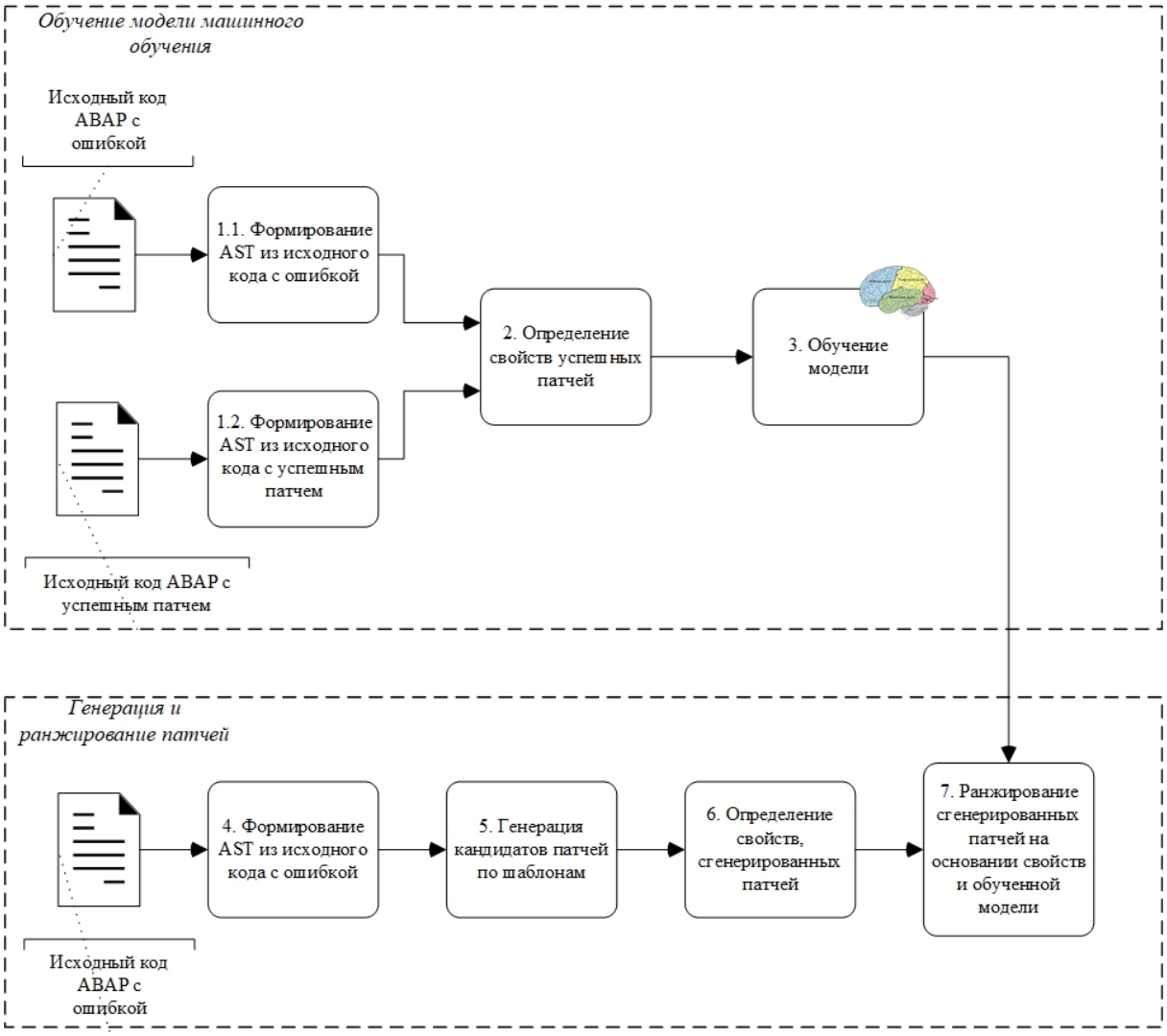


Рисунок 3.1 Схема предлагаемого метода

Предполагаемый метод содержит два основных контура – «Обучение модели машинного обучения» и «Генерация и ранжирование патчей», в рамках которых можно выделить семь функциональных блоков.

Блок 1 «Формирование абстрактного синтаксического дерева». Абстрактное синтаксическое дерево (Abstract syntax tree – AST) [87] строится на основе исходного кода с ошибкой и патча. На основании текста исходного кода ABAP-программы, содержащей ошибку, и исправления данной ошибки (патч), формируется два независимых AST, путем применения метода рекурсивного спуска [88]. Дальнейшая работа по анализу исходного кода ABAP-программы производится над AST, что дает более точное представление о типах элементов ABAP-программы (переменные, константы, операторы и т.д.) и их взаимосвязях.

Блок 2 «Определение свойств успешных патчей». Для обучения вероятностной модели были сформулированы свойства успешных патчей, которые определяются путем анализа AST исходного кода с ошибкой и AST исходного кода патча, полученные в блоке 1. Например, если исправление программы было сформировано путем добавления проверки на пустое значение переменной перед выполнением оператора деления, то эту особенность можно заложить в виде свойства успешного патча и использовать для обучения вероятностной модели.

Блок 3 «Обучение модели». В этом блоке производится обучение модели машинного обучения на данных свойств успешных патчей, полученных в блоке 2. В результате обученную модель можно использовать для прогнозирования вероятности успешности применения любого патча ABAP-программы.

Блок 4 «Формирование AST из исходного кода с ошибкой». По исходному коду ABAP-программы, для которой необходимо автоматически сгенерировать патч, формируется ее AST, аналогично блоку 1.

Блок 5 «Генерация кандидатов патчей по шаблонам». На основании AST из программы с ошибкой выделяются данные по всем переменным и константам. Из полученного массива переменных/констант производится генерация массива возможных условий. На основании полученного массива переменных/констант и массива возможных условий производится генерация патчей по шаблонам.

Блок 6 «Определение свойств, сгенерированных патчей». Для сгенерированных патчей в блоке 5 производится определение их свойств, аналогично тому, как это производилось для успешных патчей в блоке 2.

Блок 7 «Ранжирование сгенерированных патчей на основании свойств и обученной модели». На основании обученной модели, полученной в блоке 3, и свойств, сгенерированных патчей, полученных в блоке 6, производится определение вероятности успешности применения, для каждого сгенерированного патча. Полученный список сгенерированных патчей сортируется по убыванию вероятности успешности применения. Сгенерированные патчи с максимальной вероятностью успешности применения считаются целевыми.

### 3.2 Описание разработанного метода

Рассмотрим более детально этапы разработанного метода и особенности реализации методов и моделей, заложенных в блоках, представленных на Рисунок 3.1.

#### 3.2.1 Формирование AST из исходного кода

Для проведения анализа ABAP программу необходимо перевести в формализованное представление, пригодное для дальнейшей обработки. В данной работе мы используем абстрактное синтаксическое дерево. Так как для используемого в работе языка ABAP отсутствует официальная грамматика для генераторов парсеров (например, для ANTLR), то авторами был разработан легковесный парсер, основанный на методе рекурсивного спуска. Собственное формирование AST из исходного кода выполняется в блоках 1 и 4, представленных на схеме на рисунке 3.1. Преследуемая цель - более точное определение типов объектов и их взаимосвязей для дальнейшего анализа ABAP-программ. Укрупненный алгоритм разбора ABAP-программ представлен в виде псевдокода в листинге 1.

```
1 Input: S
2 for str in S do {
3     for element in str do {
4             L(l) = element
5     }
6 }
7 for l in L do {
8     t(l) = l
9     t(t_t) = defClass(l)
10    t(b_t) = defBug(str)
11 }
12 for <l, t_t, b_t> in T do {
13    P(p_p) = defParent(P)
14    P(l) = l
15    P(t_n) = t_t
16    P(b) = b_t
17 }
```

Листинг 1. Алгоритм формирования AST

В строке 1 алгоритма входными данными является массив строк $str \in S$ из которых состоит исходный код ABAP-программы. В строках 2-8 для каждой строки $str$ исходного кода формируется массив лексем $L$. В строках 9-13 формируется массив токенов $t \in T$ путем определения для каждой лексемы $l$ из массива лексем $L$ следующей информации:

- типа токена $t_t$ (заголовок, оператор, скобки, число, переменная, тип), который определяется путем отнесения каждой лексемы к классу объектов языка программирования;
- признака ошибки токена $b_t$, которая определяется путем выполнения условия: если строка $str$ исходного кода содержала ошибку, то все токены $t$, которые относились к лексемам $L$, будут иметь значение истина.

В строках 14-20 формируется массив $P$ узлов $p$ дерева AST из массива токенов $T$. Каждый узел $p$ выглядят следующим образом:

$$\langle p_p | l | t_n | b \rangle \in P, \text{ где}$$

- $p_p$ – ссылка на родительский узел $p$ дерева $P$;
- $l$ – лексема;
- $t_n$ – тип узла, определяется из типа токена $t_t$;
- $b$ – признак ошибки узла, определяется из признака ошибки токена $b_t$;

Массив $P$ узлов $p$ дерева AST формируется при помощи метода рекурсивного спуска, который заключается в рекурсивном прохождении по всему массиву токенов $t \in T$ и выстраиванию их взаимосвязей через ссылки $p_p$ по грамматическим правилам языка программирования ABAP, представленным на рисунке 3.2.

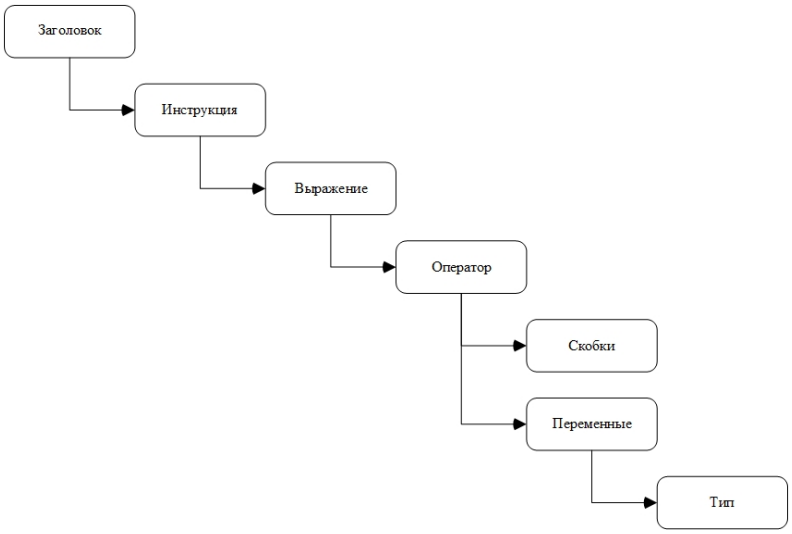


Рисунок 3.2 Грамматические правила языка программирования ABAP

### 3.2.2 Определение свойств патчей

Определение свойств патчей выполняется в блоках 2 и 6, представленных на схеме на рисунке 4. По результатам многолетнего опыта работы с языком программирования ABAP на реальных проектах и создания тысяч исправлений ошибок, а также по результатам анализа патчей к ABAP-программам из репозиториев с открытым кодом, авторами метода были сформулированы 15 свойств патчей, которые представленные в таблице 3.1. Эти свойства извлекаются из исходного кода ABAP-программ с ошибкой и патчем. Для определения свойств предварительно

производится определение разницы между $P_{bug}$ AST исходного кода с ошибкой и $P_{patch}$ AST исходного кода с патчем в виде индексов узла начала разницы $idx_{start}(P_{patch})$ и окончания разницы $idx_{end}(P_{patch})$. Также предварительно определяется список всех переменных патча $v \in V(P_{patch})$ в пределах $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$.

Таблица 3.1 Свойства патчей

| **Имя свойства** | **Алгоритм определения** |
|---|---|
| Тип ошибки $F_1$ | Определяется вручную, возможные варианты:<br>1 – деление на 0.<br>2 – обращение по пустому указателю.<br>3 – ошибка в условном операторе.<br>4 – ошибка в условии цикла. |
| Тип модификации патча $F_2$ | 1. Добавление проверки. Если узлы дерева $P_{patch}$ в диапазоне между $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$, в которых $\exists l$ = if, то $F_2$ = 1.<br>2. Изменение условия проверки. Если узлы дерева $P_{patch}$ в диапазоне между $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$ , в которых $\forall l \neq$ if, но связанные с ними узлы дерева $p_p \in P_{patch}$, в которых $\exists l$ = if, то $F_2$ = 2.<br>3. Изменение условия цикла. Если узлы дерева $P_{patch}$ в диапазоне между $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$, в которых $\forall l \neq$ loop, но связанные с ними узлы дерева $p_p \in P_{patch}$, в которых $\exists l$ = loop, то $F_2$ = 3.<br>4. Иначе если другие изменения, то $F_2$ = 4 |
| Расположение модификации патча $F_3$ | Определяется узлы дерева $P_{bug}$, содержащие ошибку. Для этого определяются индексы $idx_{start}(P_{bug})$ и $idx_{end}(P_{bug})$ узлов дерева $P_{bug,}$ у которых $\exists b$ = true.<br>Далее на основании данных расположения $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$ патча в дереве $P_{patch}$ и расположения узлов |

| | |
|---|---|
| | $idx_{start}(P_{bug})$ и $idx_{end}(P_{bug})$ узлов дерева $P_{bug}$, содержащих ошибку, определяется место возникновения модификации патча по следующему правилу:<br>**1.** Если $idx_{start}(P_{bug}) >= idx_{start}(P_{patch})$ и $idx_{end}(P_{bug}) >= idx_{end}(P_{patch})$, то $F_3 = 0$.<br>**2.** *Если* $idx_{start}(P_{bug}) < idx_{start}(P_{patch})$ и $idx_{end}(P_{bug}) >= idx_{end}(P_{patch})$, то $F_3 = 1$.<br>**3.** Иначе $F_3 = 2$. |
| Наличие оператора if в месте ошибки $F_4$ | Если в узлах дерева $P_{bug}$, у которых $\exists b$ = true и $\exists l$ = if, то $F_4 = 1$ иначе 0. |
| Наличие оператора loop в месте ошибки $F_5$ | Если в узлах дерева $P_{bug}$, у которых $\exists b$ = true и $\exists l$ = loop, то $F_5 = 1$ иначе 0. |
| Наличие операторов /,*,+,- в месте ошибки $F_6$ | Если в узлах дерева $P_{bug}$, у которых $\exists b$ = true и $\exists l$ = /,*,+,-, то $F_6 = 1$ иначе 0. |
| Наличие оператора вызова в месте ошибки $F_7$ | Если в узлах дерева $P_{bug}$, у которых $\exists b$ = true и $\exists l$ = =>, то $F_7 = 1$ иначе 0. |
| Наличие переменной в операторе if в патче $F_8$ | Определяются узлы дерева $P_{patch}$ в диапазоне между $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$, в которых $\exists l$ = if. Далее в полученных узлах если $\exists l = v$, то $F_8 = 1$ иначе 0. |
| Наличие переменной в операторе loop в патче $F_9$ | Определяются узлы дерева $P_{patch}$ в диапазоне между $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$, в которых $\exists l$ = loop. Далее в полученных узлах если $\exists l = v$, то $F_9 = 1$ иначе 0. |

| | |
|---|---|
| Наличие переменной в операторах /,*,+,- в патче $F_{10}$ | Определяются узлы дерева $P_{patch}$ в диапазоне между $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$, в которых $\exists l$ = /,*,+,-. Далее в полученных узлах если $\exists l = v$, то $F_{10}$ = 1 иначе 0. |
| Наличие переменной в операторе вызова в патче $F_{11}$ | Определяются узлы дерева $P_{patch}$ в диапазоне между $idx_{start}(P_{patch})$ и $idx_{end}(P_{patch})$, в которых $\exists l$ = =>. Далее в полученных узлах если $\exists l = v$, то $F_{11}$ = 1 иначе 0. |
| Наличие переменной в операторе if в месте ошибки $F_{12}$ | Определяются узлы дерева $P_{bug}$, у которых $\exists b$ = true, и $\exists l$ = if. Далее в полученных узлах если $\exists l = v$, то $F_{12}$ = 1 иначе 0. |
| Наличие переменной в операторе loop в месте ошибки $F_{13}$ | Определяются узлы дерева $P_{bug}$, у которых $\exists b$ = true, и $\exists l$ = loop. Далее в полученных узлах если $\exists l = v$, то $F_{13}$ = 1 иначе 0. |
| Наличие переменной в операторах /,*,+,- в месте ошибки $F_{14}$ | Определяются узлы дерева $P_{bug}$, у которых $\exists b$ = true, и $\exists l$ = /,*,+,-. Далее в полученных узлах если $\exists l = v$, то $F_{14}$ = 1 иначе 0. |
| Наличие переменной в операторе вызова в месте ошибки $F_{15}$ | Определяются узлы дерева $P_{bug}$, у которых $\exists b$ = true, и $\exists l$ = =>. Далее в полученных узлах если $\exists l = v$, то $F_{15}$ = 1 иначе 0. |

### 3.2.3 Обучение модели

Обучение модели выполняется в блоке 3 из схемы на рисунке 3.1. Для решения задач классификации с учителем в машинном обучении существует ряд моделей со своими достоинствами и недостатками. Авторами метода была выбрана модель логистической регрессии [89], так как при небольшом количестве свойств данная модель показывает лучшее быстродействие при аналогичной точности, чем другие методы машинного обучения, такие как нейронные сети или метод опорных векторов,

при этом модель логистической регрессии более удобна в реализации и адаптации [90], а также широко применяется в аналогичных работах других авторов.

Для обучения модели используется следующая матрица размера $m \times 15$ входных данных:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_{11}$ | $F_{12}$ | $F_{13}$ | $F_{14}$ | $F_{15}$ | $F_{16}$ | $F_{17}$ | $F_{18}$ | $F_{19}$ | $F_{110}$ | $F_{111}$ | $F_{112}$ | $F_{113}$ | $F_{114}$ | $F_{115}$ |
| $F_{21}$ | $F_{22}$ | $F_{23}$ | $F_{24}$ | $F_{25}$ | $F_{26}$ | $F_{27}$ | $F_{28}$ | $F_{29}$ | $F_{210}$ | $F_{211}$ | $F_{212}$ | $F_{213}$ | $F_{214}$ | $F_{215}$ |
| … | … | … | … | … | … | … | … | … | … | … | … | … | … | … |
| $F_{m1}$ | $F_{m2}$ | $F_{m3}$ | $F_{m4}$ | $F_{m5}$ | $F_{m6}$ | $F_{m7}$ | $F_{m8}$ | $F_{m9}$ | $F_{m10}$ | $F_{m11}$ | $F_{m12}$ | $F_{m13}$ | $F_{m14}$ | $F_{m15}$ |

В матрице $m$ – количество примеров в виде свойств $P_{bug}$ и $P_{patch}$ (см. раздел Формирование AST из исходного кода).

Само обучение производится для модели логистической регрессии:

$$prediction = \frac{1}{1 + e^{-\theta \times F}}$$

Суть обучения модели логистической регрессии – это определение коэффициентов $\theta$ для свойств $F$ успешных патчей (см. раздел Определение свойств патчей), которые в дальнейшем можно использовать для построения прогноза $prediction$ для любых сгенерированных патчей ABAP-программ на основании их свойств. Определение коэффициентов $\theta$ выполняется с помощью метода градиентного спуска [91], согласно которому одновременно выполняются следующие вычисления:

$$\theta_0 = \theta_0 - \alpha \times \frac{1}{m}(prediction - y)$$

$$\theta_1 = \theta_1 - \alpha \times \frac{1}{m}(prediction - y) \times F_1 + \frac{\lambda}{m} \times \theta_1$$

$$\dots$$

$$\theta_{15} = \theta_{15} - \alpha \times \frac{1}{m}(prediction - y) \times F_{15} + \frac{\lambda}{m} \times \theta_{15}, \text{ где:}$$

- $y$ – результат успешности применения $P_{patch}$ к $P_{bug}$ (задается вручную, 0 – неуспешный патч, 1 – успешный патч);

- $\alpha$ – коэффициент скорости обучения (задается вручную и используется для регулирования точности и скорости протекания процесса определения $\theta$);
- $\lambda$ – коэффициент регуляризации (используется для уменьшения вероятности возникновения переобучения модели).

При вычислении коэффициентов $\theta$ также вычисляется функция затрат $J$, которая при каждой итерации вычислений должна стремиться к нулю, и отражает прогресс и корректность выполнения метода градиентного спуск:

$$J = \frac{1}{m} \times \sum_{i=1}^{m} (-y_i \times log\ (prediction) - (1 - y_i) \times log\ (1 - prediction)) + \frac{\lambda}{2 \times m} \times \sum_{j=2}^{15} {\theta_j}^2$$

### 3.2.4 Генерация кандидатов патчей по шаблонам

Генерация кандидатов патчей по шаблонам выполняется в блоке 5 на схеме метода на рисунок 4. Здесь выполняется формирование кандидатов патчей по заранее определенным шаблонам из объектов исходного кода ABAP-программы с ошибкой. Этот алгоритм представлен в листинге 2.

```
1 Input: P_fix
2 v_fix = defVarible(P_fix)
3 for v_fix1 in V_fix do {
4     for v_fix2 in V_fix do {
5             CND_fix(cnd_fix) = v_fix1>v_fix2
6             CND_fix(cnd_fix) = v_fix1<v_fix2
7             CND_fix(cnd_fix) = v_fix1=v_fix2
8             CND_fix(cnd_fix) = v_fix1≠v_fix2
9     }
10    CND_fix(cnd_fix) = v_fix1 is initial
11    CND_fix(cnd_fix) = v_fix1 is not initial
12 }
13 for cnd_fix in CND_fix do {
14    GeneratePatchAddIf (cnd_fix)
15    GeneratePatchEditIf (cnd_fix)
16    GeneratePatchEditCycle (cnd_fix)
17 }
```

Листинг 2. Алгоритм генерации кандидатов патчей по шаблонам

В строке 2 определяется массив переменных $V_{fix}$ дерева $P_{fix}$, которое было получено путем формирования AST (см. раздел Формирование AST из исходного кода) из текста программы, для которой необходимо автоматически сгенерировать патч. Массив переменных $V_{fix}$ определяется из $l$ узлов $p_{fix} \in P_{fix}$, у которых $\exists b$ = true и $\exists t_n$ = Varible. В строках 3-14 генерируется массив условий $CND_{fix}$ путем выполнения декартового произведения массива переменных $V_{fix}$ и массива степеней сравнения (>, <, =, ≠, не заполнено, заполнено). В строках 15-17 генерируется кандидаты патчей $P_{fixpatch}$ путем добавления проверки (оператор if) с условием $cnd_{fix}$ из массива условий $CND_{fix}$ перед местом ошибки. В строках 15-18 генерируется кандидаты патчей $P_{fixpatch}$ путем замены условия в операторе проверки (if) на $cnd_{fix}$ из массива условий $CND_{fix}$ в месте ошибки. В строках 15-19 генерируется кандидаты патчей $P_{fixpatch}$ путем

изменения условия в операторе цикла (loop) на $cnd_{fix}$ из массива условий $CND_{fix}$ в месте ошибки.

Далее для сгенерированных кандидатов патчей $P_{fixpatch}$ определяются свойства (см. раздел Определение свойств патчей) и производится определение степени успешности применения (см. раздел Ранжирование сгенерированных патчей на основании свойств и обученной модели логистической регрессии).

### 3.2.5 Ранжирование сгенерированных патчей на основании свойств и обученной модели логистической регрессии

Ранжирование сгенерированных патчей на основании свойств и обученной модели логистической регрессии выполняется в блоке 7 схемы метода, представленного на рисунке 3.1. Здесь выполняется определение степени успешности $prediction_{fixpatch}$ для каждого сгенерированного кандидата патчей $P_{fixpatch}$ (см. раздел Генерация кандидатов патчей по шаблонам) путем применения обученной модели логистической регрессии:

$$prediction_{fixpatch} = \frac{1}{1+e^{-\theta \times F_{fixpatch}}}, \text{ где}$$

- $\theta$ получено в процессе обучения модели логистической регрессии (см. раздел Обучение модели логистической регрессии);
- $F_{fixpatch}$ получено в процессе определения свойств патчей для кандидатов патчей $P_{fixpatch}$ (см. раздел Определение свойств патчей).

Далее выбираются $P_{fixpatch}$ с максимальным значением $prediction_{fixpatch}$, что означает выбор кандидатов патчей с максимальной вероятностью успешности применения на основании анализа существующих патчей.

## 3.3 Выводы

В данной главе была приведена схема и вербальное описание разрабатываемого метода, представленная на рисунке 4, а также детальное описание, что включало в себя алгоритмы, представленные на листинге 1 и 2, математическую модель и используемые технологии, и методы, представленные в виде матрицы входных параметров и математических формул. На основании результатов проектирования

разрабатываемого метода была выполнена реализация программного инструментария, описание которого представлено в следующей главе.

# 4. РАЗРАБОТКА ПРОГРАММНОГО ИНСТРУМЕНТАРИЯ

В этой главе будут представлены технические подробности реализации программного инструментария к разработанному методу, что включает в себя информацию о выбранной платформе, используемым библиотекам, графическому интерфейсу и описательной части алгоритма.

## 4.1 Платформа и используемые библиотеки

В рамках данной работы было разработано оконное приложение Win32 на языке C++ на платформе .Net Framework 4.7.2 с использованием Visual Studio 2019.

В процессе реализации оконного приложения были использованы следующие стандартные библиотеки C++ [92]:

- algorithm – заголовочный файл библиотеки, который определяет функции шаблона контейнера стандартной библиотеки C++, которые выполняют алгоритмы;
- atlstr.h – заголовочный файл для доступа к наборам классов Active Template Library (ATL) по работе с COM-объектами;
- cmath – заголовочный файл для доступа к стандартным библиотекам C для работы с математическими операциями;
- cstring – заголовочный файл для доступа к стандартной библиотеке C по работе с C-строками и функциями по работе с памятью;
- fstream – заголовочный файл для доступа к стандартной библиотеке C++ по чтению/записи данных из/в файлы;
- iostream – заголовочный файл для доступа к стандартной библиотеке C++ по управлению чтением из стандартных потоков и записью в них;
- stdio.h – заголовочный файл для доступа к стандартной библиотеке C по работе со стандартными операциями вводы/выводы;
- string – заголовочный файл для доступа к стандартной библиотеке C++ по работе со строками;
- vector – заголовочный файл для доступа к стандартной библиотеке C++ по работе с контейнерами;

- Windows.h – заголовочный файл для доступа к стандартной библиотеке C++ по работе с оконными приложениями Windows, представленными через Windows API.

### 4.2 Схема структур, классов и описание их методов

Реализованное в рамках данной работы оконного приложение состоит из множества структур и классов, основные представлены на рисунке 4.1. Представленные на рисунке 4.1 структуры можно описать следующим образом:

- Token – структура используется для хранения данных токена, таких как тип токена typeToken, значения токена valueToken и признака ошибки Bug (принадлежность токена к строке программного кода, содержащей ошибку);
- NodeAST – структура используется для хранения данных узла AST и включает в себя ссылку на родительский узел parrentNodeAST, тип узла typeNodeAST, значение узла valueNodeAST и признака ошибки Bug (принадлежность узла к строке программного кода, содержащей ошибку);
- ASTplusSize – структура используется для хранения AST и данных о его размере и включает в себя ссылку на первый узел AST и его размер sizeAST.
- IndexPatchAST – структура используется для хранения данных о начальной startIndexChg и конечном endIndexChg индексе изменения узла AST, которые произвел патч.
- featuresFromCode – структура используется для хранения данных о извлекаемых свойствах feature1…15 из программного кода.
- SmartToken – класс используется для хранения текущего (обрабатываемого) токена curToken и метода getNextToken() для получения следующего токена из памяти (арифметика указателей).
- SmartAST – класс используется для хранения текущего (обрабатываемого) узла AST curNodeAST и метода getNextNodeAST() для получения следующего узла AST из памяти (арифметика указателей).

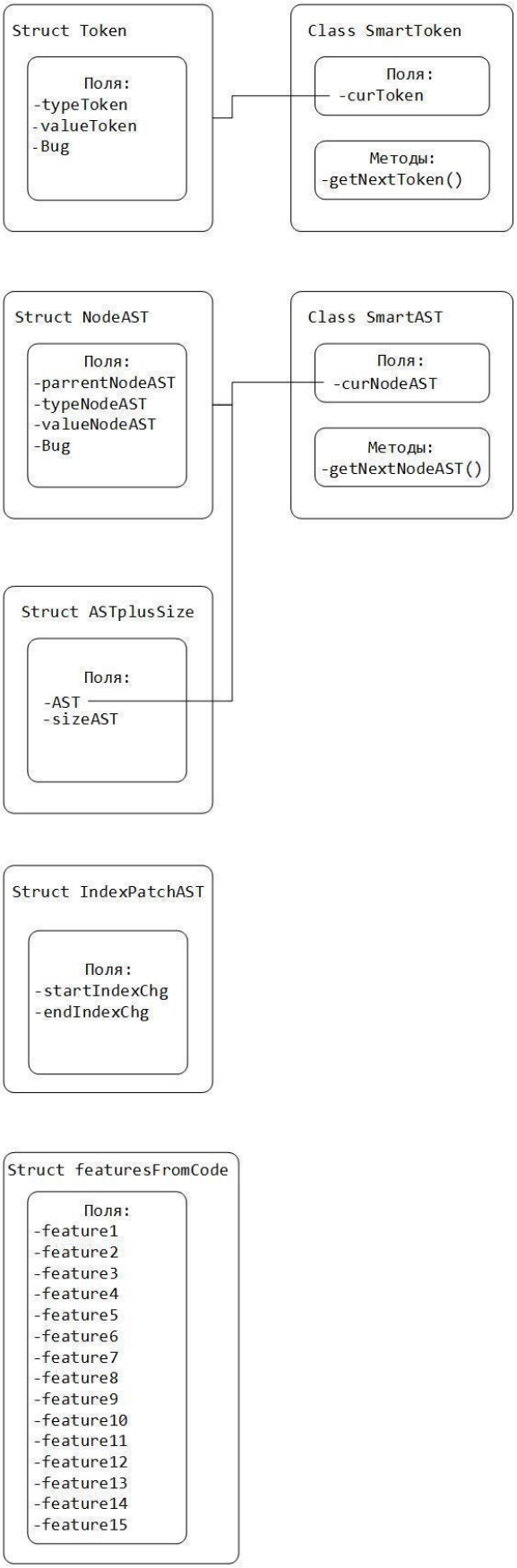


Рисунок 4.1 Схема структур и классов оконного приложения

Основная логика работы оконного приложения реализована в файлах *.cpp и *.h и описана в следующем разделе.

### 4.3 Графический интерфейс и логика работы программы

Реализованное в рамках данной работы оконного приложение состоит из двух функциональных блоков: обучение модели (Learning model), описание которого представлено в разделе 4.3.1, и генератор патчей (Patch generator), описание которого представлено в разделе 4.3.2.

#### 4.3.1 Обучение модели (Learning model)

В рамках функционального блока обучение модели (Learning model) оконного приложение графический интерфейс представлен на рисунке 4.2.

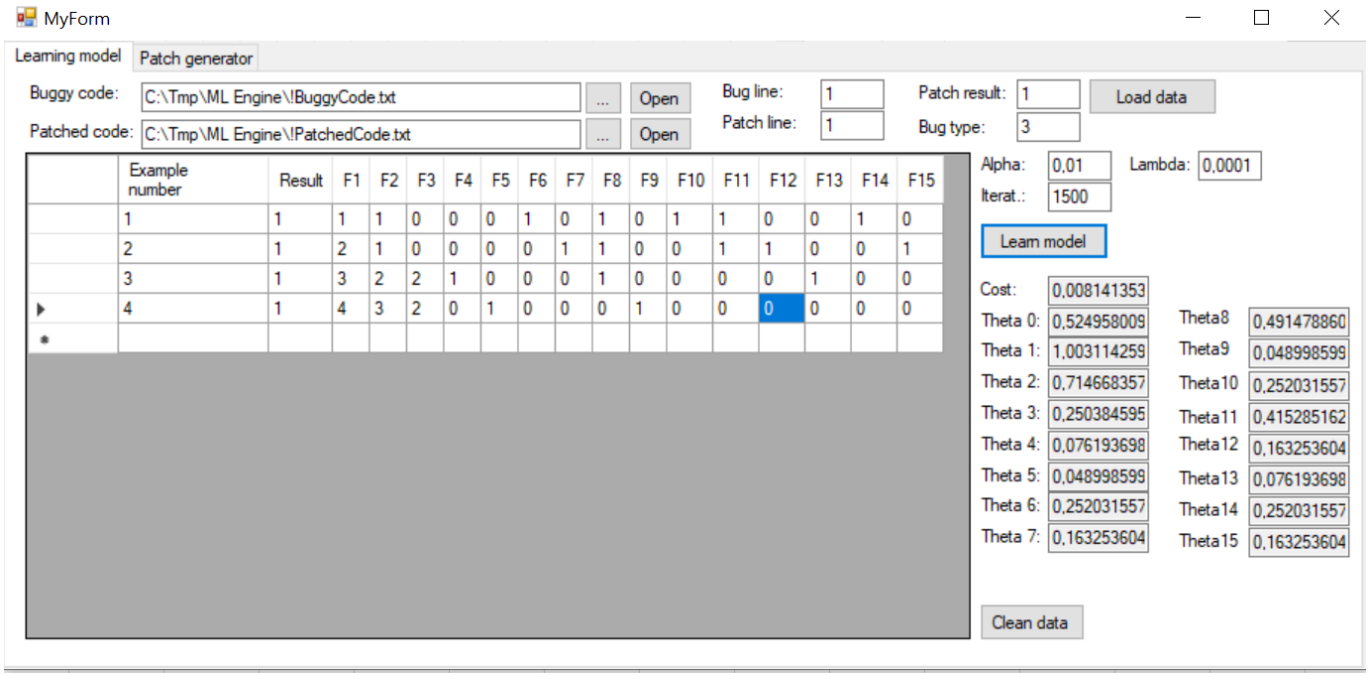


Рисунок 4.2 Графический интерфейс функционального блока обучение модели (Learning model) оконного приложения

Алгоритм работы функционального блока обучение модели (Learning model) оконного приложения представлен на рисунке 4.3.

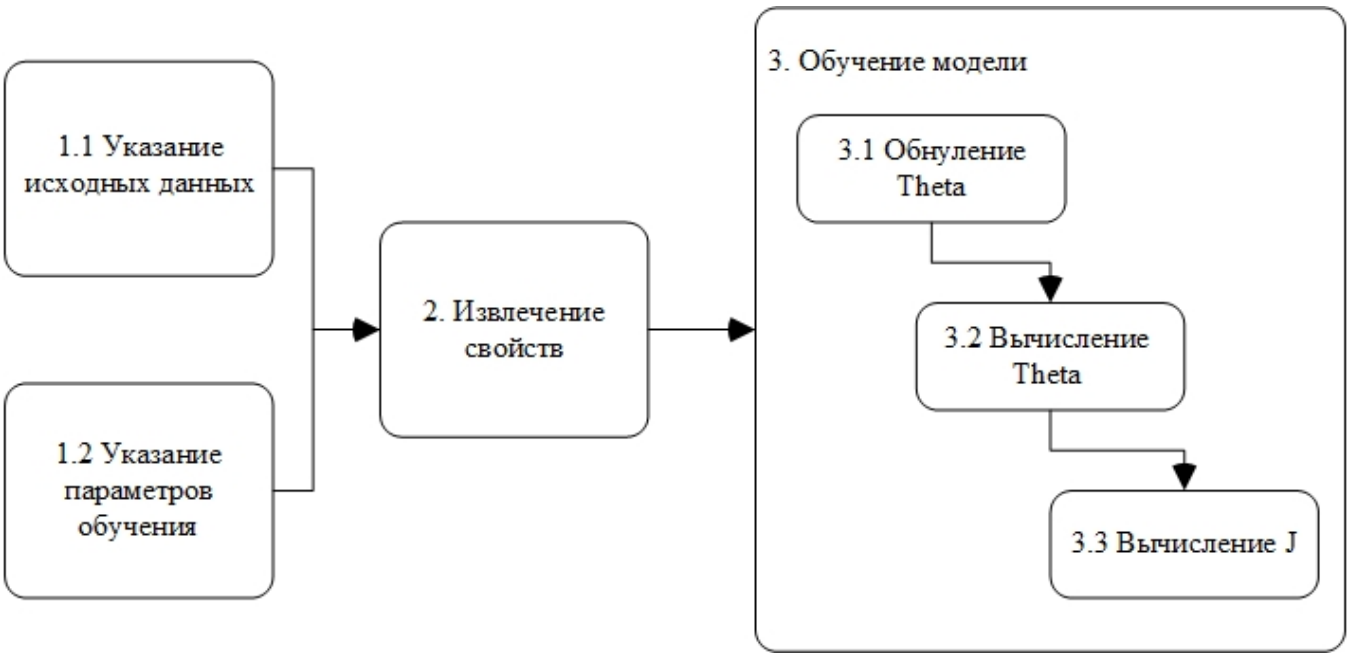


Рисунок 4.3 Алгоритм работы функционального блока обучение модели (Learning model) оконного приложения

Блок 1.1 «Указание исходных данных». В начальном блоке алгоритма пользователем указываются следующие данные:

- Buggy code – путь к файлу, содержащему программный код с ошибкой;
- Patched code – путь к файлу, содержащему программный код с патчем;
- Bug line – номер строки программного кода, содержащей ошибку;
- Patch line – номер строки программного кода, содержащей патч;
- Patch result – признак успешности патча (0 – не успешный патч, 1 – успешный патч);
- Bug type – тип ошибки (1 – деление на 0; 2 – обращение по пустому указателю; 3 – ошибка в условном операторе; 4 – ошибка в условии цикла.);

Блок 1.2 «Указание параметров обучения». Пользователем указываются параметры для обучения модели логистической регрессии (более подробно см. раздел 3.2.3 Обучение модели):

- Alpha – параметр, отвечающий за скорость обучения;
- Iterat – параметр, отвечающий за количество итераций обучения;
- Lambda – параметр, отвечающий за регуляризацию.

Блок 2 «Извлечение свойств». По кнопке Load data производится чтения файлов из Buggy code и Patched code и выполняется алгоритм извлечения свойств успешного патча (см. раздел 4.3.3). Результат свойств успешного патча из структуры featuresFromCode записывается в таблицу в колонки F1-F15.

Блок 3 «Обучение модели». По кнопке Learn model производится обучение модели логистической регрессии данными из структуры featuresFromCode с помощью метода градиентного спуска (более подробно см. раздел 3.2.3 Обучение модели) следующим образом.

Блок 3.1 «Обнуление Theta». Алгоритм задает начальные значения Theta0-Theta15 равные 0;

Блок 3.2 «Вычисление Theta». Запускается цикл по количество итераций, равный значению поля Iterat, в котором вычисляются Theta0-Theta15 по формуле определения $\theta$ (см. раздел 3.2.3 Обучение модели) для всех примеров из featuresFromCode.

Блок 3.3 «Вычисление J». Вычисляется функция затрат J по формуле определения $J$ (см. раздел 3.2.3 Обучение модели), которая отражает прогресс и корректность выполнения метода градиентного спуск.

### 4.3.2 Генератор патчей (Patch generator)

В рамках функционального блока генератор патчей (Patch generator) оконного приложение графический интерфейс представлен на рисунке 4.4.

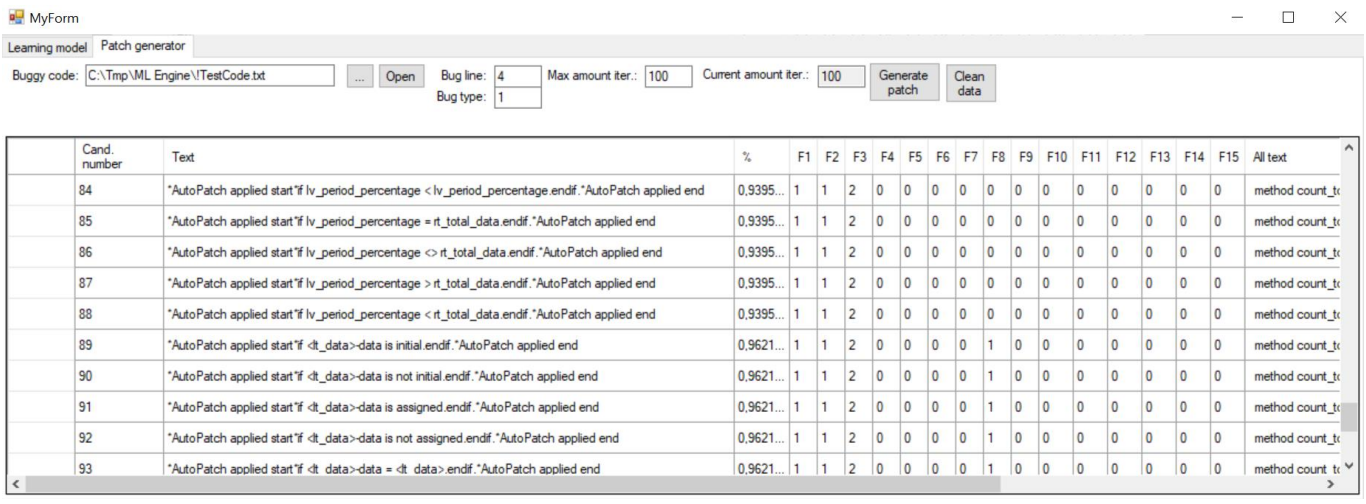

Рисунок 4.4 Графический интерфейс функционального блока обучение модели (Learning model) оконного приложения

Алгоритм работы функционального блока генератор патчей (Patch generator) оконного приложения представлен на рисунке 4.5.

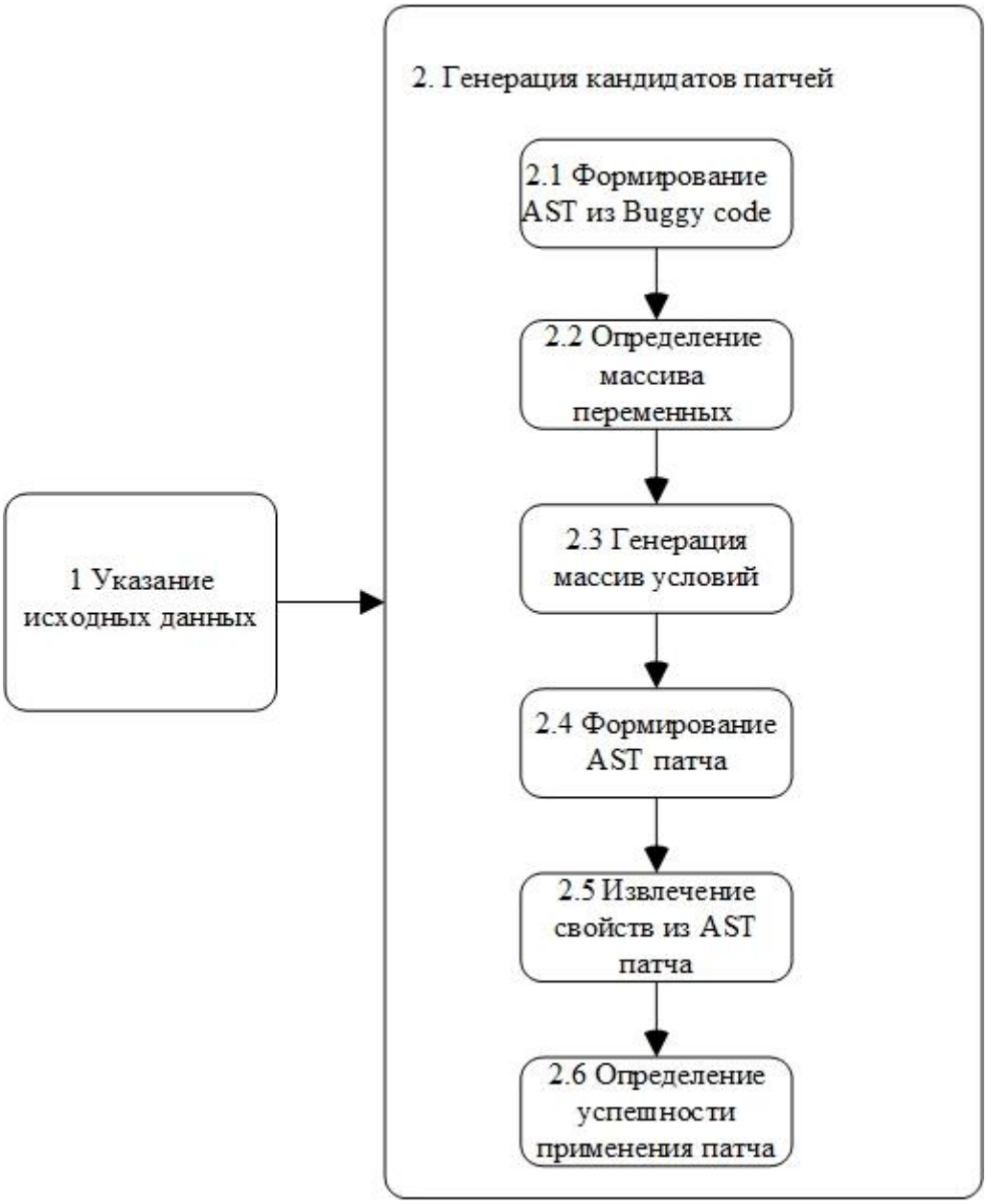


Рисунок 4.5 Алгоритм работы функционального блока генератор патчей (Patch generator) оконного приложения

Блок 1 «Указание исходных данных». В начальном блоке алгоритма пользователем указываются следующие данные:

- Buggy code – путь к файлу, содержащему программный код с ошибкой;
- Bug line – номер строки программного кода, содержащей ошибку;

- Bug type – тип ошибки (1 – деление на 0; 2 – обращение по пустому указателю; 3 – ошибка в условном операторе; 4 – ошибка в условии цикла);
- Max amount iter – максимальное количество возможных для генерации кандидатов патчей;

Блок 2 «Генерация кандидатов патчей». По кнопке Generate patch запускается алгоритм генерации кандидатов патчей, который состоит из следующих шагов.

Блок 2.1 «Формирование AST из Buggy code». Формируется AST в виде структуры ASTplusSize из программного кода, указанного в Buggy code (алгоритм описан в шагах 1,2 раздела 4.4.3 Алгоритм извлечения свойств успешного патча);

Блок 2.2 «Определение массива переменных». Из данных AST в ASTplusSize определяется массив переменных PatchVaribles (все узлы AST, у которых typeNodeAST = Varible);

Блок 2.3 «Генерация массив условий». Генерируется массив условий и далее на их основании массив кандидатов патчей по шаблонам (см. раздел 3.2.5 Ранжирование сгенерированных патчей на основании свойств и обученной модели логистической регрессии). Программный код кандидата патча отображается в поле Text, а полный экземпляр исправленной программы в поле All text;

Блок 2.4 «Формирование AST патча». Для каждой пары сгенерированного кандидата патча формируется AST в виде структуры ASTplusSize (алгоритм описан в шагах 1,2 раздела 4.4.3 Алгоритм извлечения свойств успешного патча);

Блок 2.5 «Извлечение свойств из AST патча». Для каждой пары ASTplusSize из сгенерированного кандидата патча и ошибки определяются свойства featuresFromCode (см. раздел 4.4.3 Алгоритм извлечения свойств успешного патча);

Блок 2.6 «Определение успешности применения патча». Для каждого сгенерированного кандидата патча на основании его свойств featuresFromCode определяется его успешности применения в поле % по формуле, описанной в разделе 3.2.5 Ранжирование сгенерированных патчей на основании свойств и обученной модели логистической регрессии.

### 4.3.3 Алгоритм извлечения свойств успешного патча

Алгоритм извлечения свойств успешного патча выполняется в функциональном блоке оконного приложения обучение модели (Learning model) и генератор патчей (Patch generator) представлен на рисунк 4.6.

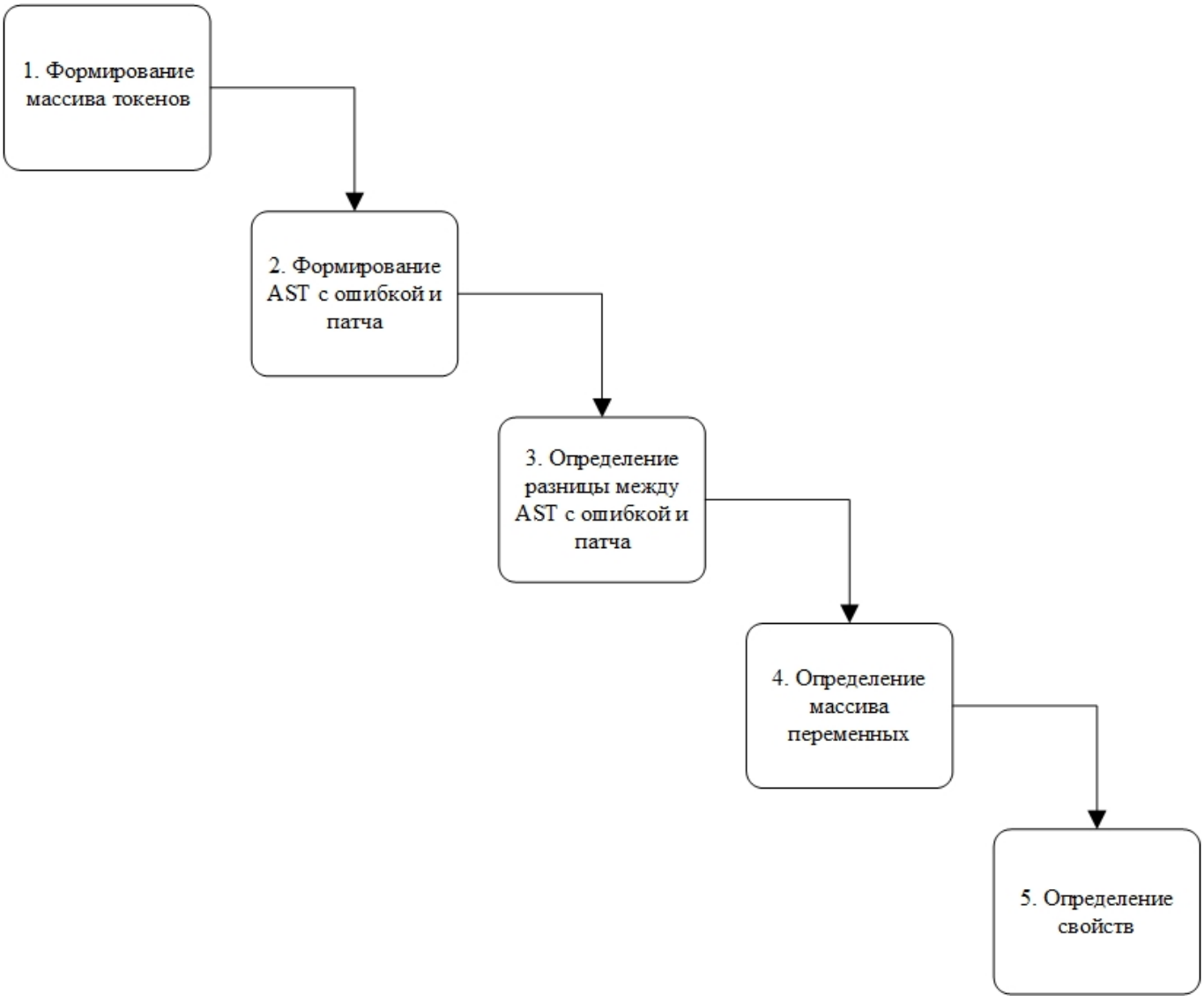


Рисунок 4.6 Алгоритм извлечения свойств успешного патча оконного приложения

Блок 1 «Формирование массива токенов». Для каждой строки программного кода из файлов Buggy code и Patched code (или Buggy code и сгенерированного патча) формируются массивы токенов, описанные структурой Token. При чтении строк программного кода исключается табуляции и строки, содержащие комментарии. Для этого каждого слово в строке программного кода помещается в valueToken. Если строка программного кода имеет индекс Bug line или Patch line, то в Token атрибут

Bug = 1, иначе 0. Также в соответствии с синтаксисом языка программирования ABAP определится тип токена typeToken (например, если встречаем слово loop, то значит typeToken = Operator);

Блок 2 «Формирование AST с ошибкой и патча». На основании массивов токенов, описанные структурой Token генерируется два AST в виде структуры ASTplusSize для программного кода, содержащего ошибку и патч. Формирование AST выполняется методов рекурсивного спуска (см. раздел 3.2.1 Формирование AST из исходного кода) суть которого заключается в рекурсивном прохождении всего массива токенов Token, где каждый текущий обрабатываемый элемент Token записывается в поле CurToken класса SmartToken, а переход к следующему элементу Token осуществляется с помощью метода getNextToken класса SmartToken. В процессе обработки CurToken класса SmartToken создаются записи узлов AST с помощью метода getNextNodeAST путем обновления поля curNode класса SmartAST в описанной структуре NodeAST, где тип узла AST typeNodeAST определяется из типа токена typeToken, значение узла valueNodeAST из значения токена valueToken, признак принадлежности узла к ошибочной строке Bug из поля токена Bug, и поле узла parrentNodeAST заполняется ссылка на родительский узел AST из поля curNode класса SmartAST. Далее определяется дальнейшая рекурсия в соответствии с грамматикой языка программирования ABAP и в зависимости от типа токена typeToken и значения токена valueToken (см. раздел 3.2.1 Формирование AST из исходного кода). Например, если тип токена оператор, то следующие токены это выражение, у которых родительский узел будет узел оператора.

Блок 3 «Определение разницы между AST с ошибкой и патча». На основании полученных AST в ASTplusSize, описывающих программный код, содержащий ошибку и патч определяется структура IndexPatchAST, которая содержит индексы начала и окончания разницы между данными двумя AST.

Блок 4 «Определение массива переменных». Из данных AST в ASTplusSize определяется массив переменных PatchVaribles (все узлы AST, у которых typeNodeAST = Varible).

Блок 5 «Определение свойств». Формируется данные свойств патчей, описанные структурой featuresFromCode, извлекаемые из ASTplusSize программного кода с ошибкой и патча на основании правил, описанных в таблице 3.1.

### 4.4 Выводы

В данной главе были приведены особенности реализации программного инструментария для описанного ранее метода автоматического формирования исправлений ошибок программного кода на основе анализа программных репозиториев. Кроме того, была представлена выбранная платформа, используемые библиотеки, схема классов и их описание, а также был представлен графический интерфейс и логика работы реализованного программного инструментария.

# 5. ТЕСТИРОВАНИЕ И АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

В этой главе представлена программа и методика тестирования, отражены результаты тестирования разработанного метода, а также проводится сравнительный анализ полученных результатов разработанного метода с аналогами.

## 5.1 Программа и методика испытаний

Для проведения качественного и релевантного тестирования разработанного метода процесс тестирования выполнялся в соответствии с шагами, представленными в таблице 5.1.

Таблица 5.1 Шаги тестирования

| **Шаг тестирования** | **Описание** |
|---|---|
| 1. Указание исходных данных для обучения модели | Вручную указываются исходные данные для обучения модели, такие как:<br>● файл, содеражащий исходных код ABAP программы, содержащий ошибку;<br>● файл, содеражащий исходных код ABAP программы, содержащий успешный патч, исправляющий ошибку из пункта 1;<br>● номер строки программного кода, содержащей ошибку;<br>● номер строки программного кода, содержащей патч;<br>● признак успешности патча (0 – не успешный патч, 1 – успешный патч);<br>● тип ошибки (1 – деление на 0; 2 – обращение по пустому указателю; 3 – ошибка в условном операторе; 4 – ошибка в условии цикла); |

| | |
|---|---|
| | ● параметр, отвечающий за скорость обучения;<br>● параметр, отвечающий за количество итераций обучения;<br>● параметр, отвечающий за регуляризацию. |
| 2. Запуск обучения модели | Запускается алгоритм обучения модели. Ожидается получение сообщения об успешном завершении обучения. В ходе обучения J должна стремиться к 0. |
| 3. Указание исходных данных для генерации патча | После окончания обучения вручную указываются данные для генерации патчей:<br>● путь к файлу, содержащему программный код с ошибкой;<br>● номер строки программного кода, содержащей ошибку;<br>● тип ошибки (1 – деление на 0; 2 – обращение по пустому указателю; 3 – ошибка в условном операторе; 4 – ошибка в условии цикла);<br>● максимальное количество возможных для генерации кандидатов патчей; |
| 4. Первичный запуск алгоритма генерации кандидатов патчей | Запускается алгоритм генерации кандидатов патчей. Начало запуска генерации патчей фиксируется секундомером. Если целевой патч найден, то тестирование переходит на шаг 5, иначе тестирование завершается и фиксируется затраченное время и количество итераций на не успешный поиск патча. |
| 5. Окончательный запуск алгоритма генерации | Анализируется количество итераций, которое было необходимо для поиска целевого патча. Далее исходные данные о максимальном количестве возможных для генерации кандидатов патчей корректируются и алгоритм генерации |

| | |
|---|---|
| кандидатов патчей | кандидатов патчей запускается повторно, время фиксируется секундомером. Полученное количество сгенерированных кандидатов патчей и время, зафиксированное секундомером считается искомым. |

Объект тестирования разработанного метода должен обладать следующими свойствами:

- исходный открытый код, разработанный на языке ABAP, из программных репозиториев типа GitHub и др;
- исходный открытый код, разработанный на языке ABAP, релевантный для типов, исправляемых разработанным методом ошибок;
- исходный открытый код, разработанный на языке ABAP, должен содержать ошибки типа: деление на 0, вызов функции по пустому указателю, ошибка в условном операторе, ошибка в условии цикла;
- исходный открытый код, разработанный на языке ABAP, должен содержать полную реализацию метода класса или программы.

### 5.2 Результаты тестирования метода

Разработанный метод был протестирован на 10 проектах на языке ABAP, содержащих ошибку. Часть примеров подготовлены авторами в соответствии с требуемыми типами ошибок для оценки работы метода, другая часть - реальные проекты. Результаты тестирования представлены в таблице 5.2.

Таблица 5.2. Результаты тестирования подхода

| **Название примера исходного кода** | **Тип ошибки** | **Число строк исходного кода** | **Количество сгенерированных кандидатов патчей** | **Время выполнения, сек** | **Патч успешно сгенерирован** |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| ABAPException.abap[1] | Деление на 0 | 34 | 300 | 66 | Нет |
| mycalculator.abap[2] | Деление на 0 | 25 | 100 | 14 | Да |
| SubRoutines.abap[3] | Деление на 0 | 59 | 1200 | 836 | Да |
| AbapRep_usingclassHana.abap[4] | Вызов функции по пустому указателю | 27 | 800 | 371 | Нет |
| zma_dp_strategy.prog.abap[5] | Вызов функции по пустому указателю | 33 | 700 | 285 | Да |
| zcl_pi_static.clas.abap[6] | Вызов функции по пустому указателю | 46 | 100 | 12 | Да |
| TestCodeWithIfBug.abap[7] | Ошибка в условном операторе | 17 | 200 | 37 | Нет |

[1]https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/efc47953337bb8fbaeee506ee9a3c701bfa4f498/ABAPException.abap

[2]https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/master/mycalculator.abap

[3]https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/master/SubRoutines.abap

[4]https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/master/AbapRep_usingclassHana.abap

[5]https://github.com/Huargh/OO-Design-Patterns-in-ABAP/blob/master/src/zma_dp_strategy.prog.abap

[6]https://github.com/ivangurin/abapPI/blob/5f30db0cc7a408a759ad833fe14f6e803b1b46bf/src/zcl_pi_static.clas.abap

[7] https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithIfBug.abap

| | | | | | |
|---|---|---|---|---|---|
| TestCodeWithIfBug2.abap[8] | Ошибка в условном операторе | 11 | 50 | 9 | Да |
| TestCodeWithCycleBug.abap[9] | Ошибка в условии цикла | 13 | 20 | 8 | 1 |
| TestCodeWithCycleBug2.abap[10] | Ошибка в условии цикла | 13 | 50 | 12 | 0 |
| | | **276** | **3520** | **1650** | **6/10** |

В первом столбце представлено название проекта с ошибкой и ссылка на репозиторий исходных кодов GitHub. Во втором столбце представлен тип ошибки, для которого были сгенерированы патчи. В третьем столбце представлено количество строк исходного кода с ошибкой. Четвертый столбец содержит количество сгенерированных кандидатов патчей для исправления ошибки для каждого проекта. В рамках испытаний формировалось такое количество кандидатов патчей, которое было достаточно для получения ожидаемого результата. В пятом столбце представлено время, которое потребовалось для генерации кандидатов патчей для каждого проекта с ошибкой. В последнем столбце отражен признак успешности генерации патчей для каждого проекта с ошибкой. Здесь подразумевается, что патч считается успешно сгенерированным, если среди всех сгенерированных кандидатов патчей с максимальной вероятностью успешности применения $prediction_{fixpatch}$ находится искомый патч.

[8]https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithIfBug2.abap
[9]https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithCycleBug.abap
[10]https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithCycleBug2.abap

Испытания разработанного авторами метода проводились на стенде со следующими характеристиками: Intel Core i3-7100U 2.40 Ghz, 4.00 Gb RAM, Windows 10.

В целях оценки эффективности разработанного метода был произведен сравнительный анализ основных параметров с существующими методами исправления ошибок программного обеспечения (см. раздел 1.2.4 Сравнительный анализ методов обнаружения и исправления ошибок программного обеспечения), результаты которого представлены в таблице 5.3.

Таблица 5.3. Сравнительный анализ разработанного метода с аналогами.

| **Метод** | **Класс метода** | **Оценка по параметрам** | | | | |
|---|---|---|---|---|---|---|
| | | **Язык программирования** | **Тип ошибки** | **Число строк исходного кода** | **Успешность** | **Время** |
| Разработанный в рамках текущей работы метод | Machine learning | 1 | 3 | 276 | 60 | 5,7 |
| GenProg: A Generic Method for Automatic Software Repair | Genetic programming | 1 | 4 | 120000 | 77 | 0,047533 |

| | | | | | | |
|---|---|---|---|---|---|---|
| Automated repair of binary and assembly programs for cooperating embedded devices | Genetic programming | 1 | 4 | 31862 | 70,75 | 0,819288 |
| Relifix: automated repair of software regressions | Genetic programming | 1 | 0 | 864000 | 65,7 | 0 |
| ASTOR: A Program Repair Library for Java | Genetic programming | 1 | 0 | 0 | 14,7 | 0 |
| History Driven Program Repair | Genetic programming | 1 | 0 | 0 | 25,5 | 0 |
| R2Fix: Automatically Generating Bug Fixes from Bug Reports | Machine learning | 2 | 3 | 17300000 | 71,3 | 0 |
| Data-guided repair of selection statements | Machine learning | 1 | 4 | 0 | 85 | 0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| Automatic Patch Generation by Learning Correct Code | Machine learning | 1 (C) | 0 | 5139000 | 26 | 0 |
| ELIXIR: effective object oriented program repair | Machine learning | 1 | 8 | 2087000 | 60 | 0 |
| Automated error localization and correction for imperative programs | Semantics approach | 1 | 1 | 0 | 0 | 0 |
| Automated repair of HTML generation errors in PHP applications using string constraint solving | Semantics approach | 1 | 1 | 22799 | 86 | 0 |
| SemFix: Program Repair via Semantic Analysis | Semantics approach | 1 | 6 | 10585 | 53,33 | 0,047237 |
| Automatic patch generation learned | Semantics approach | 1 | 0 | 483004 | 22,6 | 0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| from human-written patches | | | | | | |
| MintHint: automated synthesis of repair hints | Semantics approach | 1 | 7 | 1307 | 73 | 0 |
| Vejovis: suggesting fixes for JavaScript faults | Semantics approach | 1 | 4 | 0 | 91 | 0 |
| Automated Fixing of Programs with Contracts | Semantics approach | 1 | 0 | 72920 | 42 | 0,684092 |
| Safe Memory-Leak Fixing for C Programs | Semantics approach | 1 | 1 | 522000 | 28 | 0,000574 |
| Staged Program Repair with Condition Synthesis | Semantics approach | 1 | 0 | 5139000 | 28 | 0,063012 |
| DirectFix: Looking for Simple Program Repairs | Semantics approach | 1 | 2 | 4128 | 59 | 0,242248 |

| | | | | | | |
|---|---|---|---|---|---|---|
| CLOTHO: saving programs from malformed strings and incorrect string-handling | Semantics approach | 1 | 4 | 67200 | 73 | 0 |
| Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT (Nopol) | Semantics approach | 1 | 1 | 0 | 0 | 0 |
| Qlose: Program Repair with Quantitative Objectives | Semantics approach | 1 | 2 | 81 | 100 | 1,016049 |
| Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis | Semantics approach | 1 | 2 | 4573 | 87,5 | 1,535097 |
| JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder | Semantics approach | 1 | 0 | 175000 | 100 | 0,001371 |

| | | | | | | |
|---|---|---|---|---|---|---|
| CRSearcher: Searching Code Database for Repairing Bugs | Semantics approach | 1 | 5 | 0 | 0 | 0 |
| Contract-based program repair without the contracts | Semantics approach | 1 | 0 | 320000 | 80 | 2,063963 |
| Semantic program repair using a reference implementation | Semantics approach | 1 | 0 | 4573000 | 100 | 0,003543 |
| Static automated program repair for heap properties | Semantics approach | 4 | 3 | 721700 | 37,24 | 0,00324 |
| Automated program repair with canonical constraints | Semantics approach | 1 | 0 | 0 | 66,66 | 0 |
| Context-aware patch generation for better automated program repair | Semantics approach | 1 | 0 | 231000 | 84 | 0,056483 |

В первом столбце представлено название метода исправления ошибок программного обеспечения. Во втором столбце представлен класс метода

исправления ошибок программного обеспечения. В третьем столбце отражено количество исправляемых типов ошибок. В четвертом столбце представлено количество строк исходного кода с ошибкой. Пятый столбец содержит показатель успеха (процент исправленных ошибок). В шестом столбце представлена скорость выполнения алгоритма (секунд на обработку 1 строки исходного кода программы).

В результате проведенных экспериментов успешно было найдено 6 патчей для 10 программ с ошибкой за 1650 секунд, что соответствует или превосходит показатели эффективности аналогичных методов, сравнительный анализ которых был приведен в главе 1. Кроме того, результаты проведенных экспериментов свидетельствуют о реальности применения методов машинного обучения для автоматического формирования патчей, но в то же время полученные точность и скорость работы свидетельствует о необходимости проведения дополнительных испытаний, более качественного обучения модели логистической регрессии, увеличении мощности испытательного стенда, а также других улучшений разработанного метода. Эти улучшения предполагается разработать и внедрить в последующих работах.

### 5.3 Выводы

В рамках данной главы было представлено описание исходных данных и результаты испытаний разработанного метода, и сравнение его с аналогами. Были выделены основные 5 метрик для проведения оценки работоспособности разработанного метода, а также приведено описание тестового стенда.

# ЗАКЛЮЧЕНИЕ

В данной работе мы провели обзор, классификацию и определили лучшие международные практики в области методов обнаружения и исправления ошибок программного обеспечения. Для проведения данного исследования нами были выделены 30 существующих синтаксических и семантических методов обнаружения и исправления ошибок программного обеспечения за период с 2012 по 2018 годы. Количество и качество обозреваемых исследований в области методов обнаружения и исправления ошибок программного обеспечения позволяют с уверенностью сказать, что данное направление является актуальным, но в тоже время, с появлением новых подходов и технологий, еще не до конца исследованным.

В ходе проведенного исследования был разработан метод автоматического формирования исправлений ошибок программного кода ABAP-программ на основе анализа существующих патчей, который производит генерацию кандидатов патчей и ранжирует полученные результаты с помощью методов машинного обучения. Полученные предварительные результаты испытаний позволяют утверждать, что применение методов машинного обучения для решения задач автоматического исправления ошибок в программах является перспективным направлением программной инженерии.

Разработанный в рамках данной работы программный инструмент в дальнейшем может использоваться для проведения исследований применения классов методов искусственного интеллекта для решения задач генерации программного кода, например, в целях тестирования различных методов машинного обучения, такие как SVM, нейронные сети, Deep learning, так и для тестирования различных гипотез по извлечению полезных данных (Data mining) из программных репозиториев. Направления дальнейшего развития данной работы:

- проведение более глубокого тестирования разработанного метода на более широком множестве реальных проектов;
- расширение разработанного метода в части поддержки новых языков программирования;

- расширение множества извлекаемых свойств и списка исправляемых типов ошибок;
- применение более сложных моделей машинного обучения в целях улучшения показателей эффективности метода.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Herrington J. Code generation in action. – Manning Publications Co., 2003.
2. Goues C. L., Pradel M., Roychoudhury A. Automated program repair //Communications of the ACM. – 2019. – Т. 62. – №. 12. – С. 56-65.
3. An object oriented approach to constructing recursive descent parsers.
4. Kleinbaum D. G. et al. Logistic regression. – New York: Springer-Verlag – 2002.
5. Ruder S. An overview of gradient descent optimization algorithms //arXiv preprint arXiv:1609.04747. – 2016.
6. S. Forrest, “Genetic Algorithms: Principles of Natural Selection Applied to Computation,” Science, vol. 261, pp. 872-878, Aug. 1993.
7. Witten I. H. et al. Data Mining: Practical machine learning tools and techniques. – Morgan Kaufmann, 2016.
8. Cadar C. et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs //OSDI. – 2008. – Т. 8. – С. 209-224.
9. De Moura L., Bjørner N. Z3: An efficient SMT solver //International conference on Tools and Algorithms for the Construction and Analysis of Systems. – Springer, Berlin, Heidelberg, 2008. – С. 337-340.
10. Le Goues C. et al. Genprog: A generic method for automatic software repair //Ieee transactions on software engineering. – 2012. – Т. 38. – №. 1. – С. 54.
11. Necula G. C. et al. Cil: An infrastructure for C program analysis and transformation //International Conference on Compiler Construction. – 2002. – С. 213-228.
12. Eiben A. E. et al. Introduction to evolutionary computing. – Berlin : springer, 2003. – Т. 53.
13. Miller B. L., Goldberg D. E. Genetic algorithms, selection schemes, and the varying effects of noise //Evolutionary computation. – 1996. – Т. 4. – №. 2. – С. 113-131.
14. Zeller A. Yesterday, my program worked. Today, it does not. Why? //ACM SIGSOFT Software engineering notes. – Springer-Verlag, 1999. – Т. 24. – №. 6. – С. 253-267.
15. Schulte E. et al. Automated repair of binary and assembly programs for cooperating embedded devices //ACM SIGPLAN Notices. – ACM, 2013. – Т. 48. – №. 4. – С. 317-328.

16. Tan S. H., Roychoudhury A. relifix: Automated repair of software regressions //Proceedings of the 37th International Conference on Software Engineering-Volume 1. – IEEE Press, 2015. – С. 471-482.
17. Böhme M., Roychoudhury A. Corebench: Studying complexity of regression errors //Proceedings of the 2014 International Symposium on Software Testing and Analysis. – ACM, 2014. – С. 105-115.
18. Abreu R., Zoeteweij P., Van Gemund A. J. C. On the accuracy of spectrum-based fault localization //Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). – IEEE, 2007. – С. 89-98.
19. Martinez M., Monperrus M. Astor: A program repair library for java //Proceedings of the 25th International Symposium on Software Testing and Analysis. – ACM, 2016. – С. 441-444.
20. Qi Z. et al. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems //Proceedings of the 2015 International Symposium on Software Testing and Analysis. – ACM, 2015. – С. 24-36.
21. Debroy V., Wong W. E. Using mutation to automatically suggest fixes for faulty programs //Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. – IEEE, 2010. – С. 65-74.
22. Abreu R., Zoeteweij P., Van Gemund A. J. C. An evaluation of similarity coefficients for software fault localization //Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on. – IEEE, 2006. – С. 39-46.
23. Le X. B. D., Lo D., Le Goues C. History driven automated program repair. – 2016.
24. Ray B. et al. A large scale study of programming languages and code quality in github //Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. – ACM, 2014. – С. 155-165.
25. Kim D. et al. Automatic patch generation learned from human-written patches //Proceedings of the 2013 International Conference on Software Engineering. – IEEE Press, 2013. – С. 802-811.

26. Liu C. et al. R2Fix: Automatically generating bug fixes from bug reports //2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. – IEEE, 2013. – С. 282-291.
27. Padioleau Y., Lawall J. L., Muller G. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers //Electronic Notes in Theoretical Computer Science. – 2007. – Т. 166. – С. 47-62.
28. Gopinath D. et al. Data-guided repair of selection statements //Proceedings of the 36th International Conference on Software Engineering. – ACM, 2014. – С. 243-253.
29. Mitchell T. M. et al. Machine learning. WCB. – 1997.
30. Long F., Rinard M. Automatic patch generation by learning correct code //ACM SIGPLAN Notices. – 2016. – Т. 51. – №. 1. – С. 298-312.
31. Zeller A., Hildebrandt R. Simplifying and isolating failure-inducing input //IEEE Transactions on Software Engineering. – 2002. – Т. 28. – №. 2. – С. 183-200.
32. Jose M., Majumdar R. Cause clue clauses: error localization using maximum satisfiability //ACM SIGPLAN Notices. – 2011. – Т. 46. – №. 6. – С. 437-446.
33. Chandra S. et al. Angelic debugging //Proceedings of the 33rd International Conference on Software Engineering. – ACM, 2011. – С. 121-130.
34. Long F., Rinard M. Staged program repair with condition synthesis //Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. – ACM, 2015. – С. 166-178.
35. Le Cessie S., Van Houwelingen J. C. Ridge estimators in logistic regression //Applied statistics. – 1992. – С. 191-201.
36. Könighofer R., Bloem R. Automated error localization and correction for imperative programs //Proceedings of the International Conference on Formal Methods in Computer-Aided Design. – FMCAD Inc, 2011. – С. 91-100.
37. Do H., Elbaum S., Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact //Empirical Software Engineering. – 2005. – Т. 10. – №. 4. – С. 405-435.
38. Reiter R. A theory of diagnosis from first principles //Artificial intelligence. – 1987. – Т. 32. – №. 1. – С. 57-95.

39. Samimi H. et al. Automated repair of HTML generation errors in PHP applications using string constraint solving //Software Engineering (ICSE), 2012 34th International Conference on. – IEEE, 2012. – С. 277-287.
40. Torlak E. A constraint solver for software engineering: finding models and cores of large relational specifications : дис. – Massachusetts Institute of Technology, 2009.
41. Nguyen H. D. T. et al. Semfix: Program repair via semantic analysis //Software Engineering (ICSE), 2013 35th International Conference on. – IEEE, 2013. – С. 772-781.
42. Jones J. A., Harrold M. J., Stasko J. Visualization of test information to assist fault localization //Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on. – IEEE, 2002. – С. 467-477.
43. Le Goues C. et al. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each //Software Engineering (ICSE), 2012 34th International Conference on. – IEEE, 2012. – С. 3-13.
44. Weimer W. et al. Automatically finding patches using genetic programming //Proceedings of the 31st International Conference on Software Engineering. – IEEE Computer Society, 2009. – С. 364-374.
45. Kaleeswaran S. et al. Minthint: Automated synthesis of repair hints //Proceedings of the 36th International Conference on Software Engineering. – ACM, 2014. – С. 266-276.
46. Janssen T., Abreu R., van Gemund A. J. C. Zoltar: A toolset for automatic fault localization //Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. – IEEE Computer Society, 2009. – С. 662-664.
47. Cramér H. Mathematical Methods of Statistics. Princeton Landmarks in Mathematics and Physics Series. – 1945.
48. Zhang K., Statman R., Shasha D. On the editing distance between unordered labeled trees //Information processing letters. – 1992. – Т. 42. – №. 3. – С. 133-139.
49. Ocariza Jr F. S., Pattabiraman K., Mesbah A. Vejovis: suggesting fixes for JavaScript faults //Proceedings of the 36th International Conference on Software Engineering. – ACM, 2014. – С. 837-847.

50. Mesbah A., Van Deursen A., Lenselink S. Crawling Ajax-based web applications through dynamic analysis of user interface state changes //ACM Transactions on the Web (TWEB). – 2012. – Т. 6. – №. 1. – С. 3.
51. Meawad F. et al. Eval begone!: semi-automated removal of eval from javascript programs //ACM SIGPLAN Notices. – 2012. – Т. 47. – №. 10. – С. 607-620.
52. Pei Y. et al. Automated fixing of programs with contracts //arXiv preprint arXiv:1403.1117. – 2014.
53. Ernst M. D. et al. Dynamically discovering likely program invariants to support program evolution //IEEE transactions on software engineering. – 2001. – Т. 27. – №. 2. – С. 99-123.
54. Meyer B. Object-oriented software construction. – New York : Prentice hall, 1988. – Т. 2. – С. 331-410.
55. Gao Q. et al. Safe memory-leak fixing for c programs //Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on. – IEEE, 2015. – Т. 1. – С. 459-470.
56. Mechtaev S., Yi J., Roychoudhury A. Directfix: Looking for simple program repairs //Proceedings of the 37th International Conference on Software Engineering-Volume 1. – IEEE Press, 2015. – С. 448-458.
57. Cohen E. et al. VCC: A practical system for verifying concurrent C //International Conference on Theorem Proving in Higher Order Logics. – Springer, Berlin, Heidelberg, 2009. – С. 23-42.
58. Barnett M. et al. Boogie: A modular reusable verifier for object-oriented programs //International Symposium on Formal Methods for Components and Objects. – Springer, Berlin, Heidelberg, 2005. – С. 364-387.
59. Fu Z., Malik S. On solving the partial MAX-SAT problem //International Conference on Theory and Applications of Satisfiability Testing. – Springer, Berlin, Heidelberg, 2006. – С. 252-265.
60. Jha S. et al. Oracle-guided component-based program synthesis //Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. – ACM, 2010. – С. 215-224.

61. Dhar A. et al. CLOTHO: Saving programs from malformed strings and incorrect string-handling //Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. – ACM, 2015. – С. 555-566.
62. DeMarco F. et al. Automatic repair of buggy if conditions and missing preconditions with SMT //Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis. – ACM, 2014. – С. 30-39.
63. Jeffrey D., Gupta N., Gupta R. Fault localization using value replacement //Proceedings of the 2008 international symposium on Software testing and analysis. – ACM, 2008. – С. 167-178.
64. D'Antoni L., Samanta R., Singh R. Qlose: Program repair with quantitative objectives //International Conference on Computer Aided Verification. – Springer, Cham, 2016. – С. 383-401.
65. Solar-Lezama A. Program sketching //International Journal on Software Tools for Technology Transfer. – 2013. – Т. 15. – №. 5-6. – С. 475-495.
66. Mechtaev S., Yi J., Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis //Proceedings of the 38th international conference on software engineering. – ACM, 2016. – С. 691-701.
67. Chen M. Y. et al. Pinpoint: Problem determination in large, dynamic internet services //null. – IEEE, 2002. – С. 595.
68. Le X. B. D. et al. JFIX: semantics-based repair of Java programs via symbolic PathFinder //Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. – ACM, 2017. – С. 376-379.
69. Le Goues C. et al. The ManyBugs and IntroClass benchmarks for automated repair of C programs //IEEE Transactions on Software Engineering. – 2015. – Т. 41. – №. 12. – С. 1236-1256.
70. Păsăreanu C. S. et al. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis //Automated Software Engineering. – 2013. – Т. 20. – №. 3. – С. 391-425.
71. BACH L. D. X., Lo D., GOUES C. L. Empirical study on synthesis engines for semantics-based program repair. – 2016.

72. Alur R. et al. Syntax-guided synthesis //Formal Methods in Computer-Aided Design (FMCAD), 2013. – IEEE, 2013. – C. 1-8.
73. Wang Y. et al. CRSearcher: Searching Code Database for Repairing Bugs //Proceedings of the 9th Asia-Pacific Symposium on Internetware. – ACM, 2017. – C. 16.
74. Wise M. J. String similarity via greedy string tiling and running Karp-Rabin matching //Online Preprint, Dec. – 1993. – T. 119.
75. Mechtaev S. et al. Semantic Program Repair Using a Reference Implementation //Proceedings of ICSE. – 2018.
76. Jones J. A., Harrold M. J., Stasko J. Visualization of test information to assist fault localization //Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on. – IEEE, 2002. – C. 467-477.
77. van Tonder R., Le Goues C. Static Automated Program Repair for Heap Properties. – 2018.
78. Berdine J., Calcagno C., O'hearn P. W. Smallfoot: Modular automatic assertion checking with separation logic //International Symposium on Formal Methods for Components and Objects. – Springer, Berlin, Heidelberg, 2005. – C. 115-137.
79. Hill A., Păsăreanu C. S., Stolee K. T. Automated program repair with canonical constraints //Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. – ACM, 2018. – C. 339-341.
80. Jones J. A., Harrold M. J. Empirical evaluation of the tarantula automatic fault-localization technique //Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. – ACM, 2005. – C. 273-282.
81. Visser W., Geldenhuys J., Dwyer M. B. Green: reducing, reusing and recycling constraints in program analysis //Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. – ACM, 2012. – C. 58.
82. Elbaum S., Rothermel G., Penix J. Techniques for improving regression testing in continuous integration development environments //Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. – ACM, 2014. – C. 235-245.

83. Wen M. et al. Context-Aware Patch Generation for Better Automated Program Repair. – ICSE, 2018.
84. Qi Y. et al. Using automated program repair for evaluating the effectiveness of fault localization techniques //Proceedings of the 2013 International Symposium on Software Testing and Analysis. – ACM, 2013. – С. 191-201.
85. SAP SE. ABAP—Keyword Documentation. https://help.sap.com/doc/abapdocu_latest_index_htm/latest/en-US/index.htm –2019.
86. Thomas G. Dietterich. Machine learning. Encyclopedia of Computer Science. John Wiley and Sons Ltd., GBR, 1056–1059 – 2003.
87. Cui B. et al. Code comparison system based on abstract syntax tree //2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT). – IEEE, – С. 668-673 – 2010
88. Matthew S. Davis. An object oriented approach to constructing recursive descent parsers. SIGPLAN Not. 35, 2 (Feb.2000), 29–35. DOI:https://doi.org/10.1145/345105.345113 – 2000
89. Kleinbaum D. G. et al. Logistic regression. – New York: Springer-Verlag – 2002.
90. Kalantar B. et al. Assessment of the effects of training data selection on the landslide susceptibility mapping: a comparison between support vector machine (SVM), logistic regression (LR) and artificial neural networks (ANN) //Geomatics, Natural Hazards and Risk. – 2018. – Т. 9. – №. 1. – С. 49-69.
91. Ruder S. An overview of gradient descent optimization algorithms //arXiv preprint arXiv:1609.04747. – 2016.
92. Microsoft. C++ language documentation. https://docs.microsoft.com/ru-ru/cpp/cpp/?view=vs-2019
93. DeMillo R. A., Lipton R. J., Sayward F. G. Program mutation: A new approach to program testing //Infotech State of the Art Report, Software Testing. – 1979. – Т. 2. – №. 1979. – С. 107-126.