Санкт-Петербургский государственный университет

Программная инженерия
Кафедра Системного Программирования

Гарифуллина Гузель Раифовна

Автоматическое исправление ошибок в программном коде

Бакалаврская работа

Научный руководитель:
к.т.н. доц. Брыксин Т. А.

Рецензент:
ст. преп. Луцив Д. В.

Санкт-Петербург
2018

SAINT PETERSBURG STATE UNIVERSITY

Software engineering
Department of Software Engineering

Guzel Garifullina

Automatic code repair

Bachelor's Thesis

Scientific supervisor:
assoc. prof. Bryksin T. A.

Reviewer:
lecturer Luciv D. V.

Saint Petersburg
2018

Оглавление

Введение

На текущий момент исследования, изучающие исправление ошибок в программном коде, бурно развиваются. Всё это связано с тем, что процесс поиска и исправления ошибок является не только утомительным и трудоемким, но и очень часто повторяющимся процессом. Чтобы исправить ошибку, разработчику необходимо: проанализировать отчёт об ошибке, понять проблему, локализовать дефект, выполнить исправление и проверить новый код на регрессионных тестах – что является нетривиальной задачей.

Можно выделить основные направления в области автоматического исправления ошибок:

- исправляющие определенный класс ошибок;
- использующие символьное исполнение;
- системы генерации и валидации исправлений (generate-and-validate systems).

Многие эти системы имеют проблему масштабируемости: инструменты, хорошо работающие на маленьких примерах, плохо работают на реальных больших проектах (около 100 тыс. строк кода). Одним из немногих инструментов, решающих данную проблему, является Prophet[10].

Prophet – инструмент, исправляющий ошибки в программном коде посредством генерации, ранжирования и проверки упорядоченных исправлений на тестовом наборе данных. Prophet умеет исправлять шесть классов ошибок, которые исправляются путем изменения одного места программы. Например, классы ошибок, исправляемые добавлением условия перед утверждением или заменой утверждения. Данное множество не является полным, поэтому данная работа направлена на расширение набора ошибок, исправляемых Prophet.

Постановка задачи

Целью данной выпускной квалификационной работы является добавление поддержки исправления новых классов ошибок в инструмент Prophet[10]. Для достижения данной цели были поставлены следующие задачи.

– Реализовать генерацию исправлений для новых классов ошибок.

– Расширить модель корректного кода.

– Провести экспериментальные исследования работы алгоритма.

1. Обзор предметной области

1.1. Основные подходы автоматического исправления ошибок в программном коде

На данный момент область занимающаяся исправлением ошибок бурно развивается.

Основные методы исправления ошибок можно поделить ***по классу исправляемых ошибок***. Есть инструменты, которые направлены на исправление конкретных классов ошибок, например на утечки памяти[13], пропуск вызова библиотечной функции[23] и другие[3, 6]. В противоположность им есть системы, исправляющие широкий набор часто встречающихся ошибок. Эти системы можно поделить ***по входным спецификациям***.

Есть два основных подхода специфицирования поведения системы: декларативными спецификациями[1, 14] или тестовым набором. Недостатком первого подхода является плохая масштабируемость: тяжело написать спецификации для реального большого проекта. Второй подход, хотя накладывает более слабые ограничения на проект, требует для корректности работы полноты тестового набора. Данное требование необходимо для отсутствия генерации правдоподобных, но неправильных исправлений.

Методы, использующие тестовый набор, можно поделить ***по методам поиска и генерации исправлений***:

– методы использующие символьное исполнение[4, 11, 12, 15],

– системы генерации и валидации исправлений[2, 5, 9, 10, 19, 20].

Оба метода могут работать на больших реальных проектах. Например, в программе-анализаторе трафика сети Wireshark около 2814 тыс. строк кода. Основной задачей первого подхода является поиск противоречий в исходном коде с помощью символьного исполнения на тестовом наборе и дальнейшее его решение с помощью smt-решателей[7].

Второй подход направлен на улучшение генерации исправлений и дальнейшего обхода. Например, среди систем генерации и валидации есть инструменты, которые используют для генерации другие приложения-доноры[2, 5], генетическое программирование[20], алгоритм рандомного поиска[19] или детерминированные алгоритмы[9, 16].

Одна из важных проблем систем генерации и валидации исправлений – это генерация правдоподобных исправлений, которые проходят все тесты, но являются неправильными[12, 17]. Например, к таким исправлениям относятся исправления, удаляющие функциональность. В результате могут отключаться необходимые проверки или не выполняться необходимые действия в коде.

Главная причина генерации и успешного прохождения верификации неправильных исправлений – это неполнота тестового набора. Поэтому важной задачей систем генерации и валидации исправлений является приоритезация правильных исправлений среди правдоподобных. Инструмент Prophet[10] хорошо справляется с данной задачей в основном из-за алгоритма ранжирования исправлений, который использует машинное обучение.

1.2. Алгоритм работы инструмента Prophet

На вход инструменту Prophet подается исходный код программы на C и набор положительных и отрицательных тестов, выявляющих ошибку. На основе входных данных происходит локализация дефекта и в дальнейшем валидация сгенерированных исправлений. Полный процесс исправления ошибки схематически показан на рис. 1.

На этапе локализации происходит анализ потоков выполнения программы, запущенной на предоставленных тестах. В результате выдается упорядоченный список локаций с вероятностью содержания ошибки, где наибольший приоритет имеют утверждения, которые 1) часто выполняются на отрицательных тестах, 2) редко выполняются на положительных тестах и 3) выполняются позже на отрицательных тестах.

На следующем этапе Prophet генерирует множество всевозможных

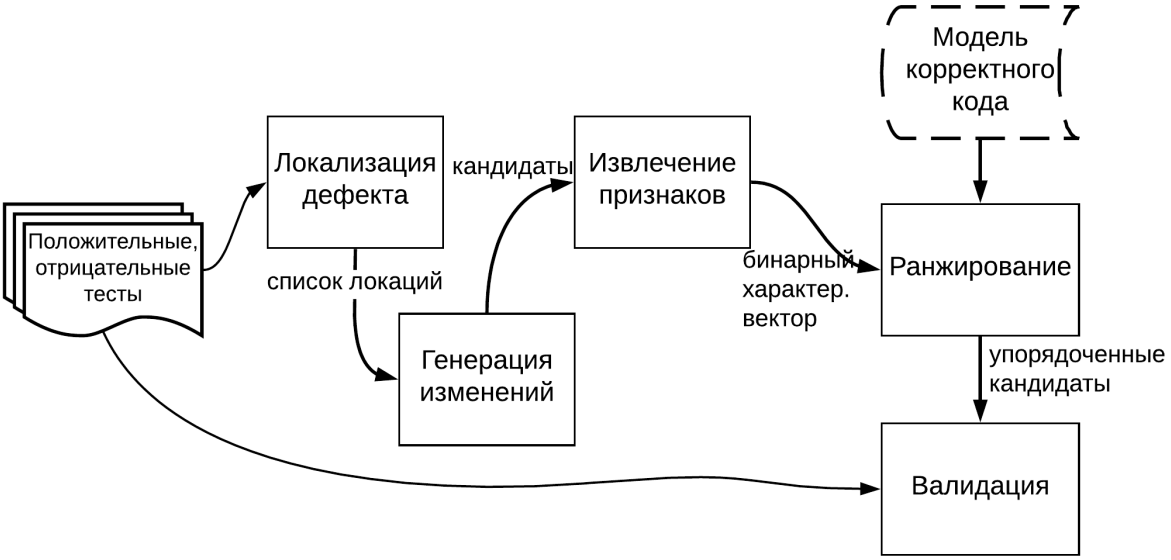

Рис. 1: Алгоритм работы Prophet

исправлений, каждое из которых изменяет одно утверждение в найденной локации. Prophet использует правила трансформации, идентичные SPR[9], а именно:

1. изменение условия (TightenCondition, LoosenCondition),
2. добавление условия (Guard, SpecialGuard),
3. условное добавление управляющего утверждения (IfExit),
4. добавление инициализации переменной (AddInit),
5. изменение утверждения и значения переменной (Replace, ReplaceString),
6. добавление измененного утверждения (AddAndReplaceKind).

Данный этап определяет классы ошибок, которые инструмент может исправить, и, следовательно, задает пространство поиска кандидатов.

В дальнейшем происходит извлечение универсальных признаков у сгенерированных изменений:

– признаки состояния – свойства, отвечающие за взаимодействие переменных, входящих в область изменения;

– признаки модификаций – свойства, фиксирующие вид изменений и различия между исходным и сгенерированным утверждением и его окружающими утверждениями.

Например, для исходного и измененного утверждения будут вычисляться признаки входящих в них переменных, зависимости между этими переменными, вид утверждения, признаки трех предшествующих и трех последующих утверждений, вид исправления.

На этапе ранжирования подсчитывается вероятностная оценка кандидатов с использованием заранее обученной модели. На финальную оценку кандидата, влияют не только универсальные признаки исправления, но и вероятностная оценка места, полученная на этапе локализации. Данный этап определяет порядок обхода кандидатов.

В результате упорядоченные кандидаты проходят поочередную валидацию: вначале каждый кандидат пытается пройти негативные тесты и в случае успеха – положительные. Из-за пересборки проекта и запуска тестов данный этап является самым долгим и ресурсозатратным, поэтому правильный порядок обхода кандидатов очень важен.

На рис. 2 показан пример исправления ошибки неучтенной единицы[1]. Слева разработчик неправильно определил границы цикла: индекс должен был быть строго меньше *PATH_SIZE*, чтобы массив не вышел за свои границы. Справа показано сгенерированное исправление Prophet. Если индекс равен 60, то цикл заканчивает свое выполнение, что соответствует корректному поведению, так как размер массива равен 60 (*PATH_SIZE*).

Для обучения модели корректного кода используется 778 примеров. На этапе обучения Prophet выполняет:

– для каждого изменения в тренировочном наборе данных Prophet анализирует структурные отличия AST в изменении разработчика и проверяет, что изменение программиста находится в пространстве поиска Prophet;

[1]Ошибка неучтенной единицы (ошибка off-by-one) — логическая ошибка в алгоритме, заключающая в неправильном определении границ величины ровно на единицу.

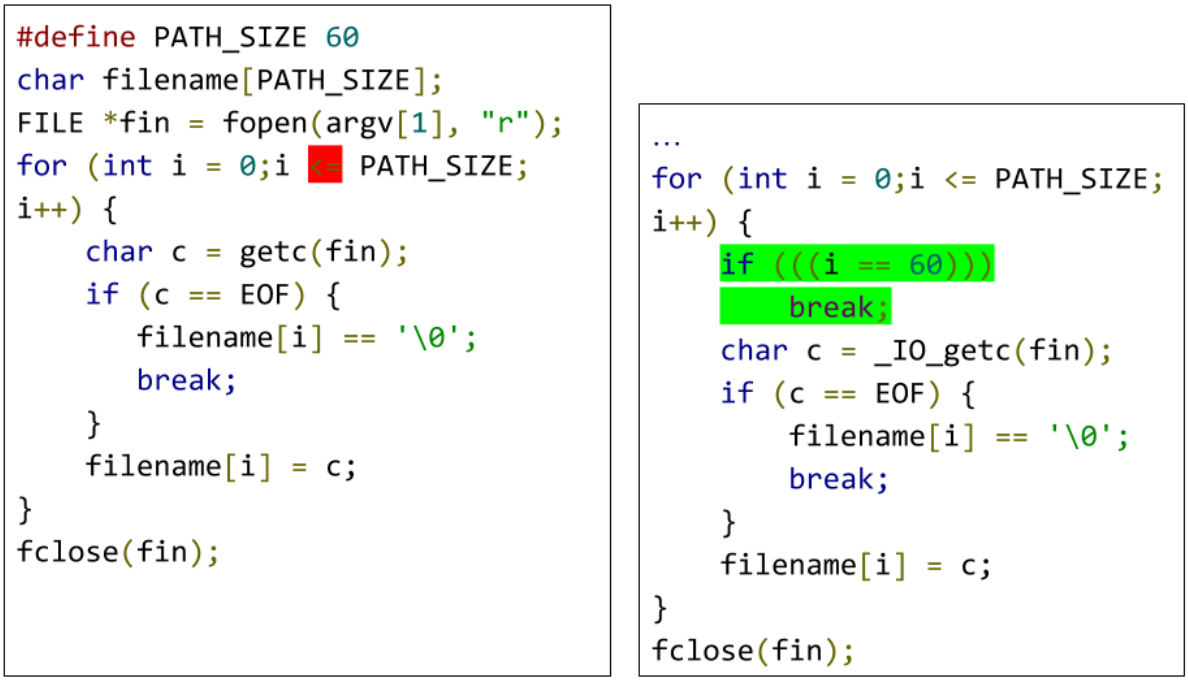

Рис. 2: Исправление ошибки неучтенной единицы инструментом Prophet

– происходит имитация работы локализатора: в исходном коде с помощью аппроксимации локализатора находятся потенциальные места с дефектом;

– в найденных местах и в известной локации с ошибкой происходит генерация изменений;

– из каждого кандидата извлекаются признаки;

– модель обучается так, чтобы уменьшить среднюю долю кандидатов, которые имеют ранг выше, чем правильное исправление разработчика.

2. Поддержка исправления новых классов ошибок

2.1. Анализ тестового набора данных

Чтобы расширить пространство генерируемых исправлений, был проанализирован тестовый набор данных[21], на основе которого оценивалась работа Prophet и происходило сравнение с другими системами.

Данный датасет содержит 69 выявленных ошибок из реальных больших проектов[2]. 19 примеров находятся в пространстве поиска Prophet и 18 из них исправляются. 50 ошибок находятся вне пространстве поиска Prophet, следовательно не могут быть исправлены инструментом.

Многие неподдерживаемые ошибки требуют нетривиальных исправлений и изменяют от двух мест в программе. Чтобы не накладывать дополнительные временные расходы на работу Prophet, при выборе нового класса ошибок для добавления в инструмент учитывались сложность необходимого изменения и требуемое увеличение размера поискового пространства для исправления. В результате выбранный пример показан на рис 3.

Исходное исправление ошибки соответствует коммиту ecb9d80[8] в PHP интерпретаторе[18]. В функции *php_json_encode* в первой строчке происходит зануление ошибки кодирования, что в общем случае некорректно из-за вложенности формата json файлов. Например, если json файл содержит массив объектов, то при кодировании программа не выявит ошибку, если последний элемент успешно пройдет процесс кодирования. Поэтому процесс зануления ошибки необходимо вынести до места в программе, где начинается обход json файла.

В общем случае данный класс ошибок можно определить, как ошибки, исправляемые вынесением первого утверждения функции перед кодом, который вызывает эту функцию. Данное исправление изменяет два места в программе.

[2]Всего в исходном наборе данных содержится 105 примеров, но 36 из них не используются в оценке работы алгоритма, так как данные примеры отвечают за изменение функциональности и не содержат ошибок.

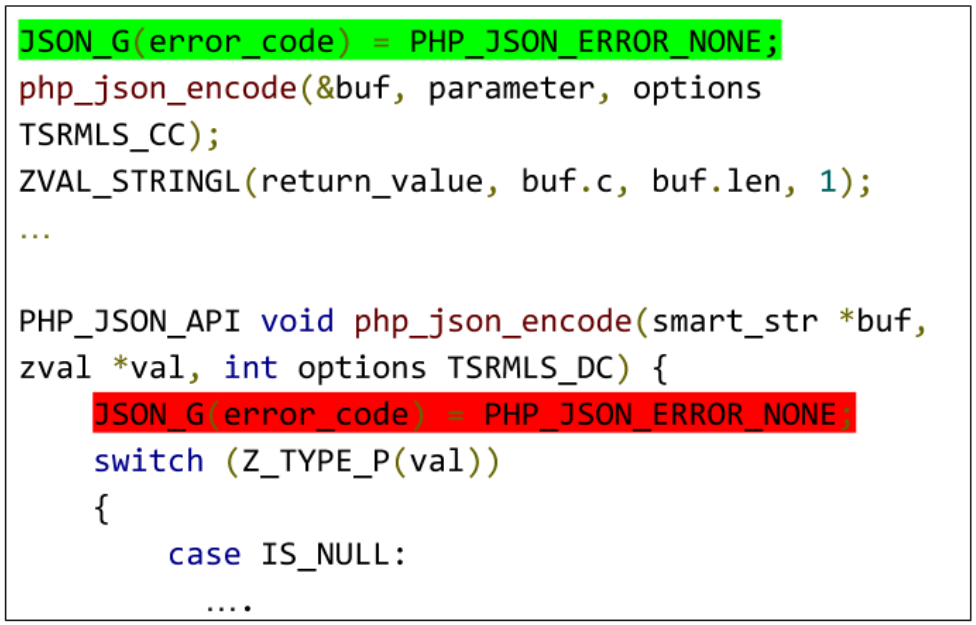

Рис. 3: Исправление ошибки php-308525-308529. Красный цвет показывает, какие утверждения необходимо удалить, чтобы код стал корректным, а зеленый – какие надо добавить

2.2. Реализация исправления новых классов ошибок

Для генерации исправлений Prophet последовательно обходит подозрительные локации и для каждого запускает генерацию кандидатов. Упрощенная диаграмма классов показана на рис. 4.

RepairCandidateGenerator – класс, генерирующий всевозможные изменения для подозрительной локации.

GlobalAnalyzer – класс, который хранит общую информацию для всех локаций из одного файла, например глобальные переменные.

LocalAnalyzer – класс, который для определенной локации (файла и номера строки) хранит всю необходимую для генерации информацию (в том числе *GlobalAnalyzer* файла), например название функции, в котором находится утверждение, утверждения-кандидаты на вставку перед данной локацией.

SourceContextManager – класс, обеспечивающий взаимодействие с исходным кодом, хранит информацию об имеющихся глобальных и локальных анализаторах и позволяет по локации получить *LocalAnalyzer*.

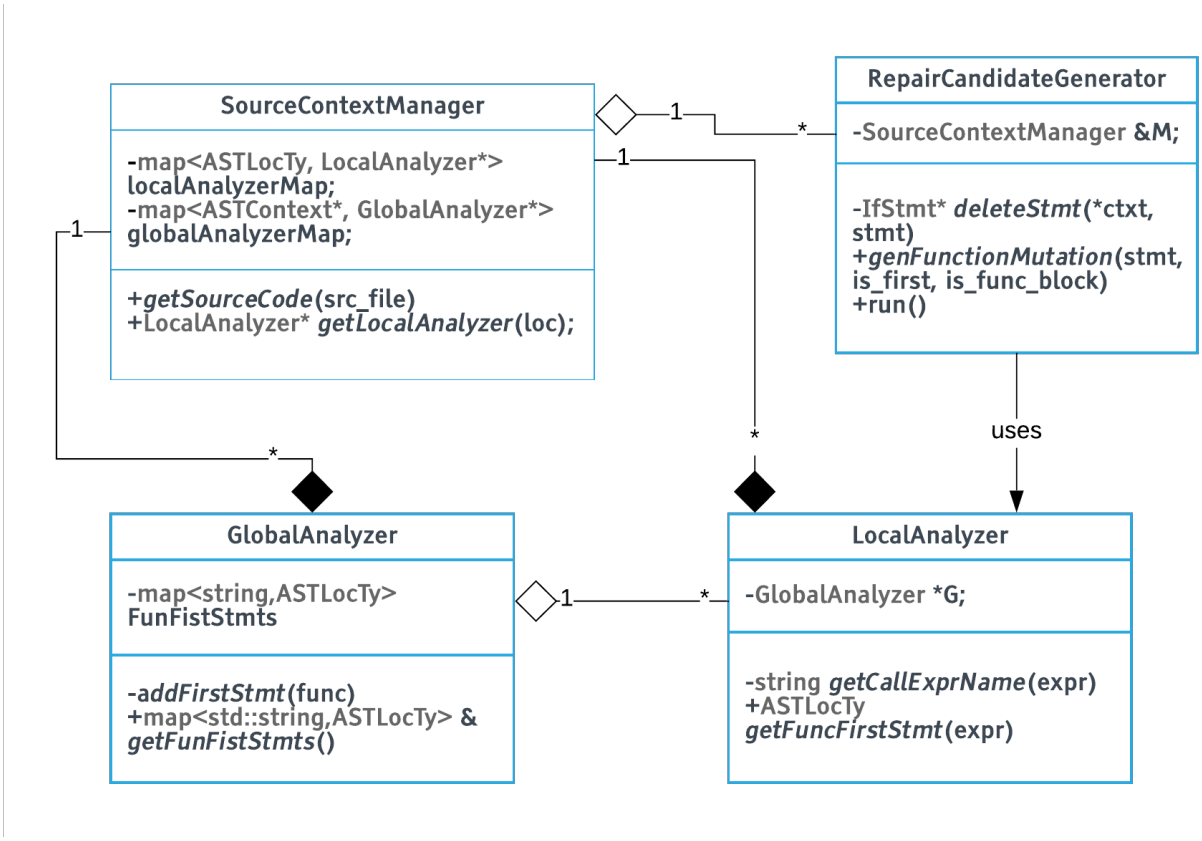

Рис. 4: Упрощенная диаграмма классов Prophet для генерации исправлений в одной локации

Для поддержки исправления нового класса ошибок в *RepairCandidateGenerator* была добавлена функция, генерирующая новые исправления вида *FunctionMutation*. Для этого происходит проверка, является ли подозрительное утверждение вызовом функции, иначе ничего не генерируется. Впоследствии перед подозрительным утверждением добавляется первое утверждение вызываемой функции и в вызываемой функции удаляется соответствующее утверждение. При первом обходе файла (при инициализации *GlobalAnalyzer*) находятся все функции, которые определены в этом же файле и содержат простое первое утверждение. В рамках данной работы утверждение считается простым, если оно содержит только операции присваивания, вызовы функций, бинарные операции, но не объявление новых переменных. В результате обхода составляется хэш-таблица, сопоставляющая имя функции и локацию первого утверждения функции.

В *LocalAnalyzer* было добавлено извлечение названия функции из

утверждения и возврат локации первого утверждения, если оно существует и соответствует вышеизложенным требованиям. В результате данная локация используется для нахождения и удаления первого утверждения функции.

Из-за ограничений на простоту первого утверждения в результате генерации в среднем добавляется не более 10 конкретных кандидатов, что очень важно, ведь чем больше пространство поиска, тем сложнее организовать обход.

3. Расширенная модель корректного кода

3.1. Новые признаки и их извлечение

Новое исправление изменяет два места в программе: добавляет утверждение перед вызывающим кодом и удаляет утверждение внутри функции, поэтому необходимо при обходе кандидата учитывать и извлекать признаки с двух мест в программе.

Внутри кандидата хранится набор базовых действий, которые необходимы для реализации исправления. При обходе кандидата Prophet учитывал только первое действие, так как второе возможное действие было изменение выражения внутри утверждения, признаки которого извлекались отдельно. В итоге при обходе была добавлена поддержка нескольких различных действий, в результате общий набор признаков можно вычислить по формуле:

$$F = R + n * M \tag{1}$$

F – общий набор признаков, соответствующий исправлению;

R – вид исправления;

M – все исходные признаки состояния и признаки модификаций без вида исправления;

n – максимальное число действий, поддерживаемое изменением (при n = 1 получится исходная формула).

В результате признаков стало 7206 (было 3515) из-за добавления нового вида исправления (*MoveStmtRepair*) и задания максимального числа действий (n), равное двум.

3.2. Обучение модели

Для процесса обучения была добавлена поддержка исправлений вида *FunctionMutation* в скрипт, сравнивающий исправления разработчика с сгенерированным исправлением. Впоследствии с использованием данного скрипта был проведен анализ тренировочной выборки по ви-

дам исправлений, необходимых для исправления ошибки. Результаты показаны в таблице 1.

Программа / Вид исправ.	php	python	subversion	другие	всего
TightenCondition	72	38	103	103	316
Guard	22	18	41	57	138
ReplaceString	26	18	41	18	103
IfExit	27	15	8	21	71
AddAndReplace	24	15	10	19	68
Replace	10	7	34	12	63
SpecialGuard	4	3	3	6	16
AddInit	2	0	0	0	2
FunctionMutation	1	0	0	0	1
все	188	114	240	236	778

Таблица 1: Число ошибок каждого вида в тренировочном наборе, где по горизонтали – название проектов из тренировочного датасета, по вертикали – вид исправления

В силу того, что примеров для обучения мало и данные в тренировочном и тестирующем наборе пересекаются, Prophet для каждого проекта обучает свою модель на общих данных посредством исключения этого проекта из обучающего датасета.

Интересно, что ошибка, требующая инициализации переменной (php-309516-309535), исправлятся в тестовом наборе данных, несмотря на отсутствие примеров в обучающем датасете. Это можно объяснить отсутствием в пространстве поиска других исправлений, проходящих все тесты, и использованием ранга локации при подсчете конечной оценки.

В ходе анализа было выявлено, что исправлений вида *FunctionMutation* в тренировочной выборке нет (единственный пример соответствует ошибке в тестовом наборе). В итоге пришлось добавить искусственные примеры для обучения модели. Зависимость числа примеров и ранга схемы трансформации[3] для правильного исправления для php-308525-308529

[3]Схемы трансформации включают в себя конкретные схемы изменений и схемы с абстрактным условием. Ранг схемы трансформации – порядковый номер схемы трансформации, упорядоченный по вероятностной оценке, где низкий приоритет имеют ранги исправлений наиболее соответствующие успешным исправлениям с высокой оценкой.

показана в таблице 2.

Число примеров	Ранг схемы	Оценка
0	10354	-8.68
1	11547	-8.72
2	4935	-7.27
3	1512	-6.11
4	544	-5.53

Таблица 2: Зависимость числа дополнительных примеров и ранжирования правильного исправления для php-308525-308529

Среди добавленных примеров первый пример соответствует занулению индекса массива, который передается функции как указатель. Второй пример соответствует присваиванию глобальной переменной значения глобальной константы. В третьем примере происходит обнуление массива внутри объекта, переданного по ссылке в качестве входного параметра функции. В четвертом примере происходит присваивание значения по умолчанию глобальной переменной.

В результате четырех примеров было достаточно для обучения модели. Следует отметить, что все три исправления вида *FunctionMutation* имеют достаточно высокий ранг схемы трансформации, поэтому не было необходимости в добавлении новых примеров. Данное наблюдение можно объяснить генерацией малого число исправлений такого вида и следовательно слабым влиянием на среднюю долю кандидатов, которые имеют ранг выше, чем правильное исправление разработчика в тренировочной выборке.

4. Апробация

4.1. Схема эксперимента

В исходной статье[10] для экспериментов использовались Amazon EC2 Intel Xeon 2.6GHz машины, запущенные на 64-битном Ubuntu сервере (для fbc использовалась 32-битная виртуальная машина с Intel Core 2.7Ghz). В данной работе все эксперименты были проделаны с использованием настроенного docker контейнера на 64-битной Ubuntu машине с процессором Intel Core i5-3317U 1.70GHz × 4 с 8GB оперативной памятью и 32GB SSD-кешем.

В исходной статье Prophet запускался на первых 200 локациях с временными ограничениями в 12 часов. В связи с тем, что локализатор недостаточно хорошо находит место ошибки для php-308525-308529 (правильная локация имеет 513 ранг), для проверки исправления нового класса ошибки в ходе данной работы вручную добавляется необходимая локация в конец первых 200 локаций перед этапом, отвечающим за генерацию исправлений.

4.2. Анализ результатов

В итоге через 134 минуты пользовательского времени (665 минут реального и 37 минут системного времени) было сгенерировано исправление для php-308525-308529 см. рис. 5.

Пространство поиска содержало 42590 всевозможных кандидатов. Среди них правильное исправление имело 2637 ранг исправления[4] и было первым исправлением, который Prophet принял. Нетрудно заметить, что данное исправление идентично исправлению разработчика.

Чтобы оценить влияние измененной модели корректного кода на существующие исправляемые ошибки, было проведено сравнение работы исходной системы и системы с новой моделью, результаты показаны в таблице 3. Каждая запись второго столбца (“Пространство поиска”)

[4]Ранг исправления в отличие от ранга схемы трансформации считается по всему пространству поиска, то есть среди конкретных схем и инстанцированных схем с абстрактным условием.

```
//prophet generated patch
(json_globals.error_code) = PHP_JSON_ERROR_NONE;
php_json_encode(&buf, parameter, options TSRMLS_CC);
....

 PHP_JSON_API void php_json_encode(smart_str *buf, zval
*val, int options TSRMLS_DC) {
       //prophet generated patch
        if (0)
          (json_globals.error_code) = PHP_JSON_ERROR_NONE;
        switch (Z_TYPE_P(val))
        {
                case IS_NULL:
                ....
```

Рис. 5: Сгенерированное правильное исправление для php-308525-308529

имеет вид X(Y/Z), где X – размер исходного пространства поиска до добавления новых признаков в модель, Y – число правильных, Z – число правдоподобный исправлений в пространстве поиска. Столбцы “Ранг до” и “Ранг после” соответствует рангу корректного исправления до и после добавления новых признаков в модель соответственно. Столбцы “Function Mutation” и “Ранг схемы после” соответствует количеству исправлений вида *FunctionMutation* и рангу правильной схемы трансформации после добавления новых признаков в модель.

В результате использования новой модели корректного кода правильные исправления были найдены первыми среди правдоподобных, ранги правильных исправлений в основном стали лучше и пространство поиска увеличилось не более чем на 10 конкретных схем трансформаций.

4.3. Анализ признаков новой модели корректного кода

Обученная модель корректного кода соответствует параметрам, которые задают веса признакам изменения. В данной модели параметры

Дефект	Пространство поиска	Ранг до	Function Mutation	Ранг после	Ранг схемы после
php-309579-309580	51306(1/2)	767	0	314	3
php-309892-309910	36940(1/21)	462	0	632	7
php-310991-310999	87574(1/1)	907	1	305	8
php-308525-308529	42587(0)	-	3	2637	544

Таблица 3: Результаты исправления ошибок

признаков находятся в диапазоне (-3.0, 3.0), где положительные значения соответствуют признакам, которые характеризуют правильные исправления. Далее выбираются самые значимые параметры модели корректного кода и рассматривается их изменение по мере добавления новых примеров.

В ходе добавления новых примеров важные признаки исходной модели не сильно изменились и сохранили относительное местоположение см. рис. 6, 7.

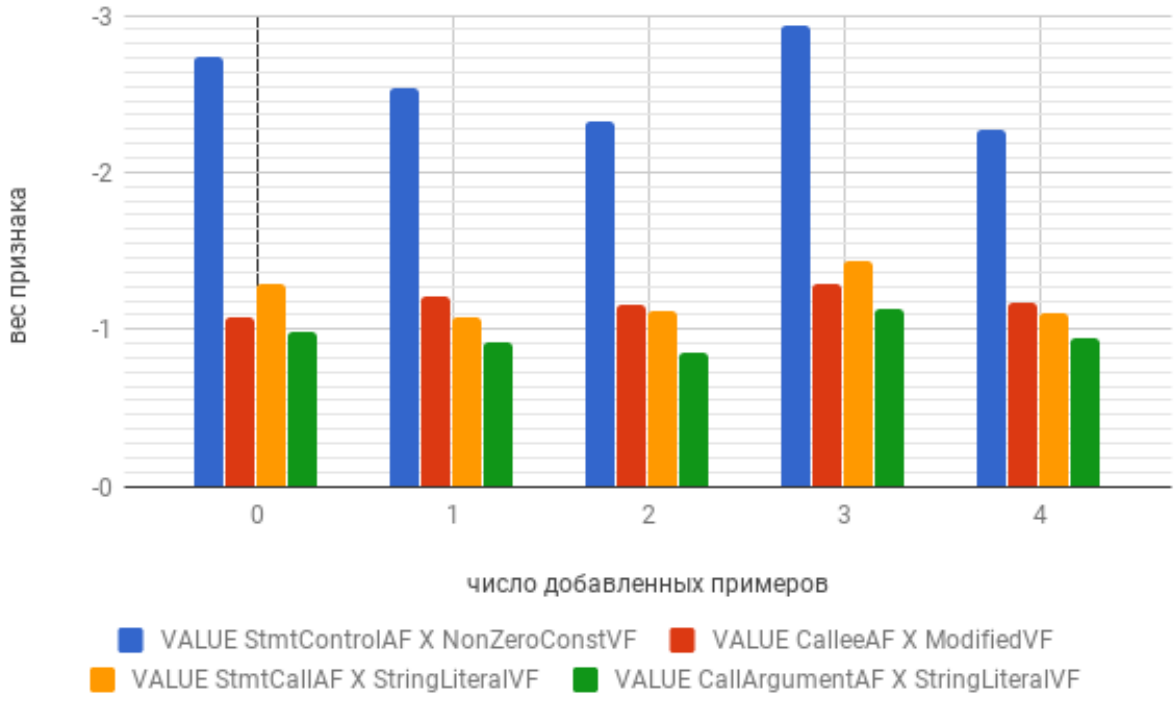

Рис. 6: Изменение веса самых важных отрицательных признаков исправления по мере добавления новых примеров

Новый признак вида *MoveStmtRepair* изначально почти никак не

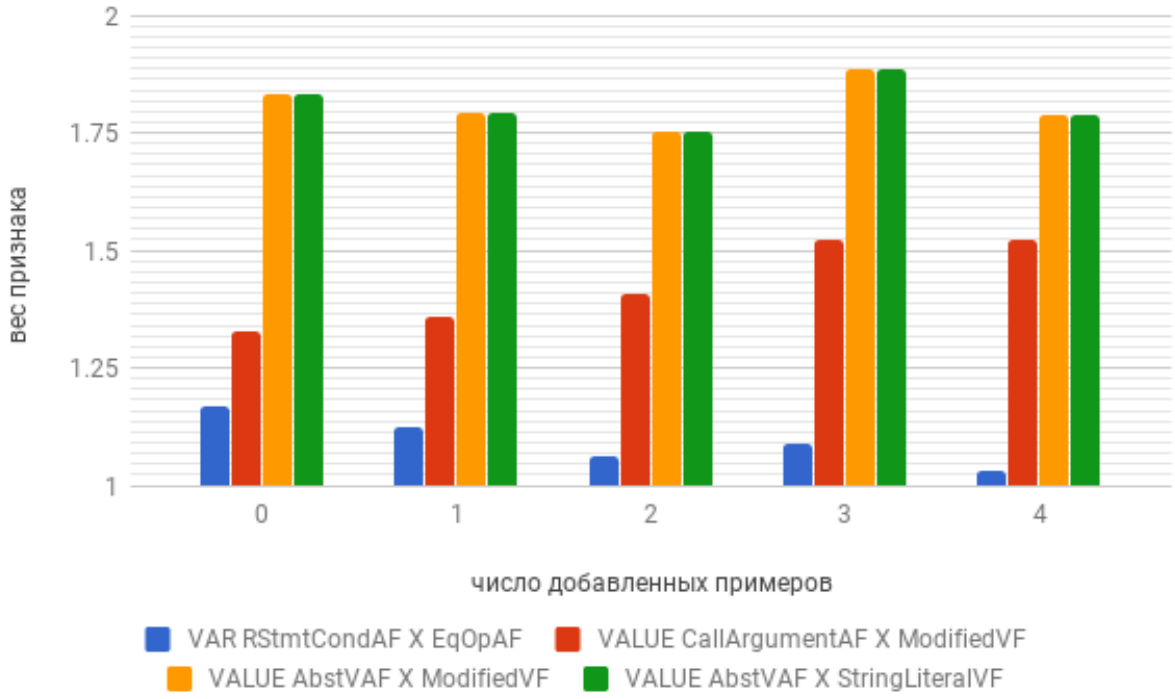

Рис. 7: Изменение веса самых важных положительных признаков исправления по мере добавления новых примеров

влиял на модель (см. рис. 8), но впоследствии стал иметь большой положительный вес. Скорее всего положительная оценка связана с малым количеством кандидатов данного типа. Остальные признаки уменьшили свои веса, в результате чего, диапазон значений параметров вида сократился. Также значимый положительный вес (0.31) приобрел признак первой локации, отвечающий за то, чтобы искомое изменяемое утверждение при *MoveStmtRepair* было вызовом функции.

Среди новых добавленных признаков, отвечающих за второе место в программе, к выраженным отрицательным признакам можно отнести разыменование указателя, изменение значения указателя и вызов функции, использующей локальную переменную. К положительным признакам – наличие операции присваивания, изменения глобальной переменной и вызов функции от указателя. По мере добавления примеров эти положительные признаки постепенно приобретают вес. Данный процесс показан на рис. 9. При добавлении первого примера веса этих признаков близки к нулю. Данное наблюдение можно объяснить от-

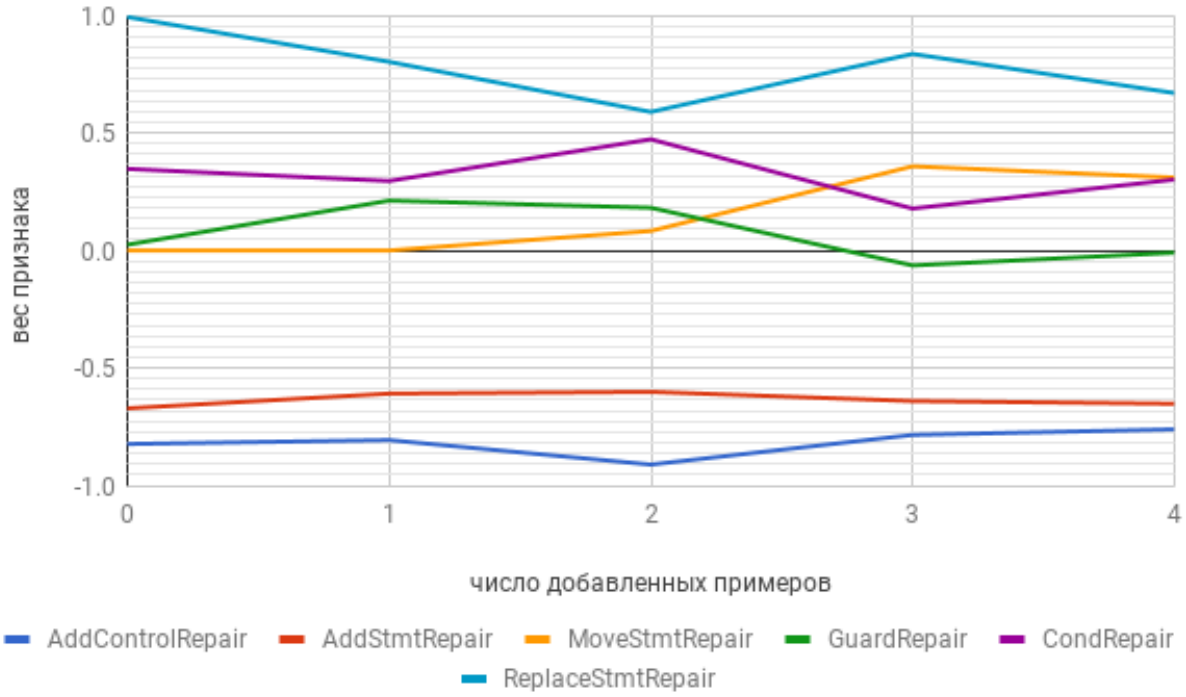

Рис. 8: Изменение веса признаков вида исправления по мере добавления новых примеров

сутствием в начальной модели признаков, отвечающих за второе место изменения, и примерам соответствующим им.

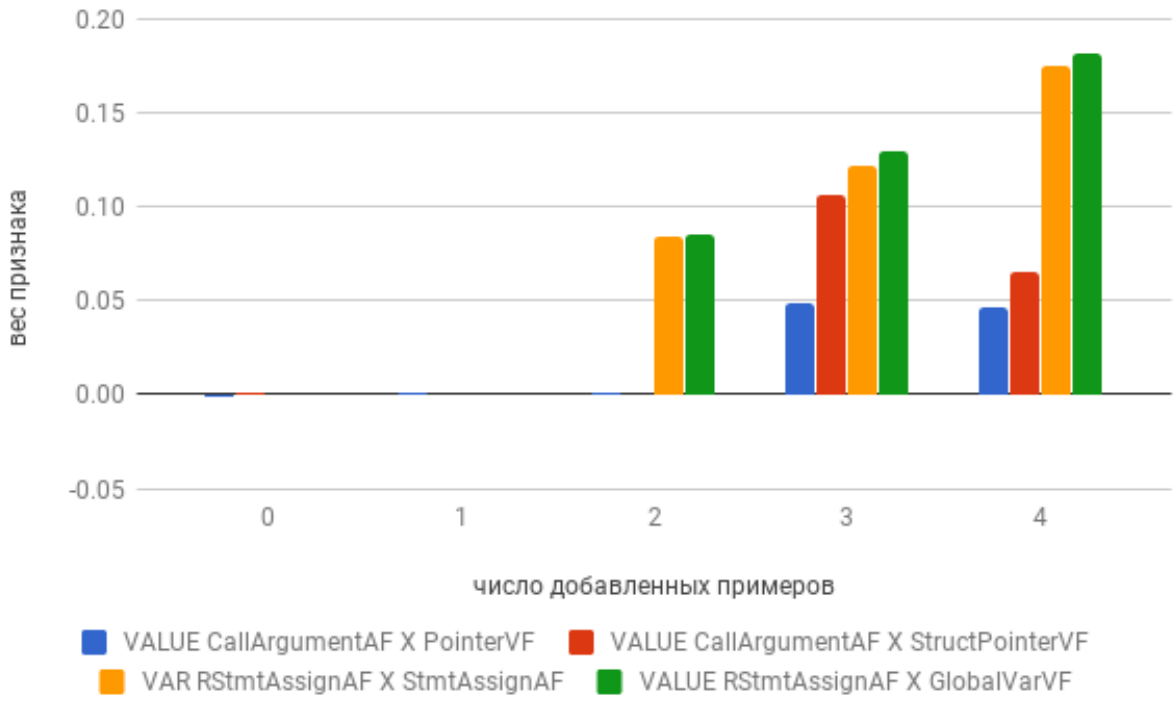

Рис. 9: Изменение весов признаков второй локации по мере добавления новых примеров

Заключение

В ходе выполнения данной работы были получены следующие результаты.

- Реализована генерация исправлений для новых классов ошибок, исправляемых вынесением первого утверждения функции перед вызывающим кодом.
- Расширена и обучена новая модель корректного кода посредством добавления новых признаков для поддержки исправлений, изменяющих код в 2 местах программы, и добавления новых примеров в тренировочный набор данных.
- Выявлены самые значимые признаки успешных исправлений и проведен анализ влияния новых тренировочных примеров на параметры модели корректного кода.

В результате было расширено пространство генерируемых исправлений в инструменте Prophet[10] и исправлена ошибка из тестового набора данных. Исходный код данной работы представлен в репозитории проекта[22].

Список литературы

[1] Automated fixing of programs with contracts / Yi Wei, Yu Pei, Carlo A Furia et al. // Proceedings of the 19th international symposium on Software testing and analysis / ACM. — 2010. — P. 61–72.

[2] Automatic error elimination by horizontal code transfer across multiple applications / Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, Martin Rinard // ACM SIGPLAN Notices / ACM. — Vol. 50. — 2015. — P. 43–54.

[3] Automatic fix for C integer errors by precision improvement / Xi Cheng, Min Zhou, Xiaoyu Song et al. // Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual / IEEE. — Vol. 1. — 2016. — P. 2–11.

[4] Automatic repair of buggy if conditions and missing preconditions with SMT / Favio DeMarco, Jifeng Xuan, Daniel Le Berre, Martin Monperrus // Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis / ACM. — 2014. — P. 30–39.

[5] Automatically patching errors in deployed software / Jeff H Perkins, Sunghun Kim, Sam Larsen et al. // Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles / ACM. — 2009. — P. 87–102.

[6] Cheng Xi. RABIEF: range analysis based integer error fixing // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering / ACM. — 2016. — P. 1094–1096.

[7] De Moura Leonardo, Bjørner Nikolaj. Satisfiability modulo theories: introduction and applications // Communications of the ACM. — 2011. — Vol. 54, no. 9. — P. 69–77.

[8] Fix Bug 54058, invalid utf-8 doesn't set json_encode() in all cases. — URL: https://github.com/php/php-src/search?q=ecb9d80&type=Commits.

[9] Long Fan, Rinard Martin. Staged program repair with condition synthesis // Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering / ACM. — 2015. — P. 166–178.

[10] Long Fan, Rinard Martin C. Automatic patch generation by learning correct code // POPL. — 2016.

[11] Mechtaev Sergey, Yi Jooyong, Roychoudhury Abhik. Directfix: Looking for simple program repairs // Proceedings of the 37th International Conference on Software Engineering-Volume 1 / IEEE Press. — 2015. — P. 448–458.

[12] Mechtaev Sergey, Yi Jooyong, Roychoudhury Abhik. Angelix: Scalable multiline program patch synthesis via symbolic analysis // Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on / IEEE. — 2016. — P. 691–701.

[13] Safe memory-leak fixing for c programs / Qing Gao, Yingfei Xiong, Yaqing Mi et al. // Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on / IEEE. — Vol. 1. — 2015. — P. 459–470.

[14] Samanta Roopsha, Olivo Oswaldo, Emerson E Allen. Cost-aware automatic program repair // International Static Analysis Symposium / Springer. — 2014. — P. 268–284.

[15] Semfix: Program repair via semantic analysis / Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, Satish Chandra // Proceedings of the 2013 International Conference on Software Engineering / IEEE Press. — 2013. — P. 772–781.

[16] Weimer Westley, Fry Zachary P, Forrest Stephanie. Leveraging program equivalence for adaptive program repair: Models and first

results // Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on / IEEE. — 2013. — P. 356–366.

[17] An analysis of patch plausibility and correctness for generate-and-validate patch generation systems / Zichao Qi, Fan Long, Sara Achour, Martin Rinard // Proceedings of the 2015 International Symposium on Software Testing and Analysis / ACM. — 2015. — P. 24–36.

[18] The source code of PHP interpreter. — URL: `https://github.com/php/php-src`.

[19] The strength of random search on automated program repair / Yuhua Qi, Xiaoguang Mao, Yan Lei et al. // Proceedings of the 36th International Conference on Software Engineering / ACM. — 2014. — P. 254–265.

[20] A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each / Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, Westley Weimer // Software Engineering (ICSE), 2012 34th International Conference on / IEEE. — 2012. — P. 3–13.

[21] A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each / Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, Westley Weimer // Software Engineering (ICSE), 2012 34th International Conference on / IEEE. — 2012. — P. 3–13.

[22] Исходный код инструмента Prophet с расширенным пространством исправляемых классов ошибок. — URL: `https://github.com/Guzele/prophet-src`.

[23] Якимов Иван Александрович, Кузнецов Александр Сергеевич. Поиск недостающих вызовов библиотечных функций с использованием машинного обучения // Труды института системного программирования РАН. — 2017. — Т. 29, № 6. — С. 117–134.