DOI: 10.15514/ISPRAS-2021-33(6)-6

Автоматическое исправление дефектов кода в системе Svace

*С.В. Сыромятников, ORCID: 0000-0001-8753-6315 <syrom@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25*

Аннотация. В данной статье рассматривается задача автоматического исправления дефектов кода на языках C/C++, найденных статическим анализатором на больших программных проектах. Описан опыт реализации соответствующего инструмента для статического анализатора Svace и принципы исправления дефектов разных типов. Особое внимание уделено исправлению разыменования нулевого указателя как наиболее важного и сложного из поддерживаемых в инструменте типов дефектов; приведена статистика работы инструмента по исправлению дефектов этого типа. Исследуются общие ограничения и специфика поставленной задачи, объясняется невозможность использования для её решения существующих систем автоматического исправления дефектов. В заключение кратко излагаются соображения по дальнейшему развитию реализованного продукта.

Ключевые слова: автоматическое исправление программ; статический анализ; дефект исходного кода; разыменование нулевого указателя.

Для цитирования: Сыромятников С.В. Автоматическое исправление дефектов кода в системе Svace. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр. 83-94. DOI: 10.15514/ISPRAS-2021-33(6)-6

Automatic Repair of Code Defects in the Svace System

*S.V. Syromiatnikov, ORCID: 0000-0001-8753-6315 <syrom@ispras.ru>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia*

Abstract. The main task described in this article is automatic fixing defects in C/C++ code found by a static analyzer on big software projects. We describe how we solved this task for Svace static analyzer and discuss main principles of automatic fixing defects of various types. We pay special attention to fixing null pointer dereference since it is the most important and sophisticated defect type among those we've supported. Statistics on fixes proposed for defects of this type is also provided. We discuss common limitations and other specificity of our task and explain why we cannot use existing automatic fixing tools for solving it. At the end we outline further steps of development of our tool.

Keywords: automatic program repair; automatic defect fixing; static analysis; code defects; null pointer dereference.

For citation: Syromiatnikov S.V. Automatic Repair of Code Defects in the Svace System. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 6, 2021, pp. 83-94 (in Russian). DOI: 10.15514/ISPRAS-2021-33(6)-6

1. Введение

Существующие системы статического анализа позволяют находить тысячи дефектов на проектах размером в миллионы строк кода. Разработчики этих систем постоянно работают над повышением точности анализа, стремясь минимизировать количество ложных

срабатываний детекторов дефектов. Однако исправление найденных дефектов кода по-прежнему требует значительных усилий и времени разработчика. Поэтому представляется актуальной следующая

Задача: предложить пользователю корректные варианты исправления дефектов, найденных статическим анализатором на больших проектах, и/или автоматически их исправить.

Такая постановка задачи накладывает важные ограничения.

- Большие проекты нередко не имеют исчерпывающего покрытия тестами.
- Инструмент автоматического исправления дефектов должен иметь высокую производительность.
- Поскольку мы разрабатываем инструмент для существующего активно используемого анализатора и для дефектов, которые он уже находит, мы крайне ограничены в модификации самого инструмента анализа. В то же время многие описанные в литературе системы основаны именно на глубокой интеграции детектора и исправителя дефектов ([1], [2]).

В силу ограничений а. и б. для решения поставленной задачи мы не можем использовать технологии, основанные на генерации и оценке в той или иной степени случайных вариантов, например, генетическое программирование ([3], [4]).

В своих исследованиях мы ограничимся языками C/C++ как одними из самых востребованных на рынке и в то же время сложных для анализа.

Здесь и далее будем называть вариант исправления корректным, если полученный после его применения код удовлетворяет следующим условиям:

- синтаксически корректен;
- не содержит исходного дефекта;
- сохраняет логику программы (за исключением исходного дефекта);
- не содержит новых привнесенных дефектов;
- приемлемо отформатирован.

Отметим, что статические анализаторы в принципе не гарантируют истинность всех найденных дефектов, а ложное срабатывание детектора исправлять, разумеется, не надо. Однако проблемы оценки истинности дефекта мы в данной статье касаться не будем.

Кроме того, гарантировать корректность предлагаемых вариантов исправления не всегда возможно. Рассмотрим

Пример 1. Дефект “локальная переменная не инициализирована”.

Очевидный способ исправления данного дефекта - добавить инициализацию в определение переменной. Но каким значением ее инициализировать? Наиболее вероятное значение для численных переменных – ноль, для указателей – нулевой указатель. Однако в некоторых случаях логика программы может подразумевать другое значение, например, вместо инициализации указателя `char* нулём char *str = NULL`; на самом деле, должна быть его инициализация пустой строкой: `char *str = ""`; Выявить формальные признаки подобной ситуации непросто. Однако вариант с инициализацией нулём следует предложить пользователю, так как с большой вероятностью он его устроит.

Назовём тип дефекта исправимым, если для него хотя бы в некоторых формально определённых ситуациях возможна автоматическая генерация вариантов исправления, в большинстве случаев оказывающихся корректными.

Далее мы обсудим некоторые общие аспекты поставленной задачи, опишем опыт её решения для статического анализатора Svace и подробно рассмотрим схему автоматического исправления для некоторых исправимых типов дефектов.

2. Автоматическое исправление в Svace

2.1 Общая схема

В архитектуре Svace ([5], [6]) изначально не была заложена возможность автоматического исправления. Более того, статический анализатор Svace работает над внутренним представлением, которое весьма далеко от исходного кода и из которого можно извлечь довольно ограниченный объем информации, нужной для исправления этого кода. Поэтому автоматический исправитель разрабатывался нами как отдельный инструмент, а его работа в общих чертах может быть описана схемой, изображенной на рис. 1. Отметим, что выдаваемой анализатором информации о дефекте в ряде случаев оказалось недостаточно для его исправления, но добавление в выдачу дополнительной информации гораздо проще интеграции в анализатор автоматического исправления.

Рис. 1. Общая схема автоматического исправителя
Fig. 1. General scheme of the automatic repairer

Этап пользовательской верификации необходим по следующим причинам.

- На вход инструмента могут попасть ложные дефекты, которые исправлять не нужно.
- Инструмент может работать некорректно. Масштабное исправление кода без верификации может привести к катастрофическим последствиям.
- Как показано выше, даже в случае адекватной работы инструмента предлагаемое исправление может нарушать логику программы.
- Иногда бывает сложно выбрать из нескольких вариантов исправления оптимальный, и разумно предоставить этот выбор пользователю.

Порой бывает целесообразно предлагать автоматическое исправление не для всех дефектов данного типа, а для некоторого достаточно широкого их подмножества, осуществляя, таким образом, *предварительную фильтрацию* найденных дефектов. Важно, чтобы критерии этой фильтрации были формальными и понятными пользователю.

2.2 Входные данные

В системе Svace поддерживается автоматическое исправление для некоторых типов дефектов в коде на C/C++. Инструмент автоисправления написан на C++ на основе предоставляемой в clang технологии *LibTooling*. На вход он получает список дефектов, про каждый из которых известно следующее:

- тип;
- позиция в коде (строка и иногда столбец); для описания некоторых дефектов требуется несколько позиций, соответствующие примеры будут приведены ниже;
- указание, где искать исходный файл, который содержит ошибочный код (см. ниже);
- другая информация в зависимости от типа дефекта.

Исходные файлы, с которыми работает инструмент исправления дефектов – это частично препроцессированный пользовательский код: в нём есть все препроцессорные директивы, кроме директивы *#include*, вместо которой присутствует содержимое соответствующих заголовочных файлов; макросы также не раскрыты. Благодаря *#line*-директивам есть возможность идентифицировать фрагменты кода, заданные позициями в пользовательских файлах.

Данные файлы генерируются в процессе работы основного анализатора Svace. Их использование позволяет генерировать исправления без доступа к пользовательскому коду, например, на удалённом сервере.

2.3 Принцип работы

Стандартными средствами clang’a наш инструмент исправления строит AST для частично препроцессированного файла и находит в нём синтаксические конструкции, соответствующие нужному дефекту. Они ищутся с использованием технологии *ASTMatcher* и с учетом их известной позиции.

Пример 2. Есть дефект “Условие if-выражения всегда ложно” в позиции *foo.c:2:4*.

Чтобы найти локализацию этого дефекта в коде, нужно найти условное выражение (*IfStmt*), перейти к его условию и проверить, что это условие имеет искомую позицию.

Предлагаемое исправление дефекта – это информация о том, как надо модифицировать текст пользовательского файла, чтобы дефект пропал. Реализованное нами автоисправление основано именно на модификации текста, а не на трансформации AST’a.

Для каждого типа дефектов используется свой набор *ASTMatcher*-запросов, свои правила и эвристики. Во многих случаях фильтрация исправляемых дефектов происходит именно за счёт ограниченности этого набора. Иногда осуществить нужную фильтрацию узлов AST’a сложно, и рациональнее отсеять неисправимые дефекты непосредственно в анализаторе, модифицировав соответствующий детектор.

Пример 3. Дефект “память, выделенная при помощи *new[]*, освобождена при помощи *delete*”.

```

1: class Complex {
2: public:
3:   ...
4:   static void* operator new (size_t size) {
5:     return new char[size];
6:   }
7: };
8:
9: void foo() {
10:   Complex *complex = new Complex();
11:   ...
12:   delete complex;
13: }
  
```

Анализатор в состоянии отследить, что изначально память была выделена с помощью *new[]*. Стандартный способ исправления данного дефекта – замена *delete* на *delete[]* – как универсальное решение, очевидно, не подходит. Отсечь данную ситуацию (*new[]* внутри *new*) при анализе AST’a сложно, так для её обнаружения нужен анализ графа вызовов. Однако в анализаторе Svace её несложно обнаружить в процессе работы детектора.

2.4 Выходные данные

На выходе инструмент генерирует файлы в формате *unified diff* или *yaml*, описывающие необходимую для исправления дефектов модификацию кода. Применение diff-файлов к пользовательскому коду осуществляется стандартными средствами Linux (утилита *patch*).

Для применения yaml-файлов используется инструмент *clang-apply-replacements*, также реализованный на базе clang.

3. Подходы к исправлению дефектов

Вообще говоря, не обязательно вносить изменения в код именно в том месте, где статический анализатор обнаружил ошибку. Внесение изменений в других местах порой целесообразнее с точки зрения программиста, но может требовать более глубокой интеграции анализатора и исправителя. В зависимости от сложности определения позиции для модификации подходы к автоматическому исправлению можно классифицировать следующим образом.

- **Исправление в месте обнаружения дефекта:** на вход инструменту автоматического исправления поступает позиция, в которой дефект был обнаружен, тот находит требуемое место в коде (или AST'е) и на основе доступного ему анализа определяет, что и как требуется изменить. Поддержка автоматического исправления для таких дефектов требует минимальных изменений в системе их поиска.

Пример 4. Дефект “присвоенное переменной значение нигде не используется”.

Очевидное исправление такого дефекта – удалить такое присваивание полностью или частично в зависимости от наличия побочных эффектов. Изменения в каких-то других местах кода не требуются.

Большинство реализованных в нашем инструменте автоматических исправлений относится к этому типу. Кроме упомянутого в *Примере 4*, это, например, “неинициализированный в конструкторе член класса”, “использование небезопасной функции” и “сравнение с всегда истинным/всегда ложным результатом”.

- **Исправление, позицию которого можно однозначно определить по трассе дефекта.**

Пример 5. Дефект “используется неинициализированная переменная”.

В данном случае позиция ошибки – это не то место, которое нужно исправлять. Исправлять нужно декларацию данной переменной. Позиция этой декларации есть в трассе дефекта, и детектор мог бы передать её инструменту исправления. В этом случае нужны изменения в детекторе, но они минимальны и реализовать их несложно.

- **Случаи, когда исправление требуется внести в несколько не связанных между собой местах кода, но каждое такое место однозначно определяется по трассе дефекта.**

Пример 6. Рассмотрим следующий случай дефекта “указатель был разыменован, а затем проверен на равенство NULL”:

```
1: char *buf = getbuf();
2: memset(buf, '\0', 5);
...
13: if (! buf) {
14:     goto err;
15: }
```

Если код в строках 3-12 удовлетворяет определенным условиям, в частности, не содержит присваиваний переменной buf, то адекватным исправлением данной проблемы будет переместить проверку buf на равенство NULL перед вызовом memset. Для этого нужно знать две позиции - разыменования и проверки.

- **Более сложные алгоритмы вычисления позиции исправления.** Примерами могут служить системы MemFix ([1]), выявляющая и исправляющая утечки ресурсов, и VFix ([2]), предназначенная для поиска и исправления разыменований нулевого указателя. В обеих системах в детекторе дефектов происходит не только обнаружение проблемы в коде, но и на основе достаточно сложного потокового анализа определяется оптимальное место её исправления. В нашем случае использование подобных алгоритмов означало бы реализацию новых детекторов дефектов или принципиальную переделку имеющихся,

что выходит за рамки поставленной задачи.

4. Шаблоны исправления

Шаблоном исправления мы будем называть совокупность

- синтаксических конструкций, имеющих требуемые позиции в коде и удовлетворяющих определенным ограничениям, и
- правил модификации этих конструкций.

Например, в *Примере 2* такой синтаксической конструкцией будет if-выражение, условие которого имеет заданную позицию и не имеет побочных эффектов, а соответствующая модификация состоит в удалении этого выражения из кода, причем способ удаления зависит от наличия else-ветки. Как видно из *Примера 6*, иногда дефекту соответствует не одна синтаксическая конструкция, а несколько, не являющихся поддеревьями друг друга.

Иногда к одному дефекту применимы несколько шаблонов исправления. В таких случаях теоретически возможны разные подходы:

- выдать пользователю все сгенерированные исправления, оставив за ним выбор наилучшего варианта;
- автоматически выбрать наиболее подходящий шаблон и применить только его;
- сгенерировать все возможные исправления и выбрать наилучшее (например, на основе тестов); в нынешней системе Svace этот вариант едва ли применим;
- некоторая комбинация перечисленных выше вариантов.

В настоящее время в нашей системе предпочтительным является второй вариант.

В нашем инструменте подлежащие исправлению конструкции ищутся при помощи шаблонов ASTMatcher, а далее к найденным узлам AST'a применяются реализованные на C++ дополнительные проверки. Правила модификации кода тоже запрограммированы на C++.

Существуют инструменты, по сути предлагающие некоторую формализацию нашего понятия шаблона исправления. Это, например, системы, основанные на patch-подобных языках, таких как SmPL ([7], [8]) или PATL ([9]). Есть также инструменты, позволяющие формально описать синтаксическую структуру фрагмента кода до исправления и после него на языке, близком к целевому языку программирования, например, Nobrainer ([10]) или Refaster ([11]). В ряде случаев такие инструменты могут использоваться для автоматического исправления. Однако выразительной мощности основанного на SmPL Coccinelle, по-видимому, не хватит для решения нашей задачи: например, там отсутствует возможность проверки позиции искомой конструкции или некоторых её семантических свойств. Практика использования Nobrainer показала, что при поддержке новых шаблонов также регулярно требовалось расширение его возможностей. Кроме того, ни один из этих языков, по-видимому, не позволяет реализовать шаблон, соответствующий *Примеру 6*, т. е. исправление в не связанных между собой местах кода.

5. Автоматическое исправление разыменований нулевого указателя

Дефект “разыменование нулевого указателя” для пользователей очень важен, поскольку может представлять собой серьезную уязвимость ([12]). Поэтому неудивительно, что пользователи проявляют к его автоматическому исправлению повышенный интерес. Как можно его исправить? Очевидное решение – добавить перед разыменованием проверку указателя на NULL – вызывает много вопросов. Какая часть кода не должна выполняться, когда указатель равен NULL? Что делать, если он равен NULL? В частности, если разыменование происходит внутри return-выражения, какое значение нужно вернуть? Иногда дефект данного типа суть проявление проблем в логике программы, и его исправление на самом деле может быть гораздо сложнее. Рассмотрим

Пример 7.

```
1: char *default_name = "";
```

```

2: char *default_string_value = NULL;
3:
4: typedef struct {
5:     char *name;
6:     char *description;
7: } described_name;
8:
9: void set_default_name(described_name *name) {
10:    // должно быть default_name вместо default_string_value
11:    name->name = default_string_value;
12: }
13:
14: void someproc() {
15:     described_name name;
16:     ...
17:     set_default_name(&name);
18:     ...
19:     if (name.name[0] == '\0') // ALARM!!!
20:         return;
21:     ...
22: }

```

Однако после изучения найденных на реальных проектах дефектов мы пришли к выводу, что существуют синтаксически различимые случаи, когда вполне вероятный вариант исправления предложить можно.

В Svace существует большое количество детекторов, выявляющих те или иные случаи разыменования нулевого указателя, и качество работы некоторых из них довольно высоко. На основании экспертной оценки из этих детекторов было отобрано несколько, выдающих заметный процент автоматически исправимых дефектов, и одновременно предложено несколько шаблонов для исправления этих дефектов. Основой для экспертных оценок, как правило, были результаты анализа на проекте Tizen-5.5.

Отобранные детекторы можно разделить на три типа:

- обнаруживающие потенциальное разыменование нуля путем анализа трассы от присваивания указателю нуля (*source*) до разыменования этого указателя (*sink*);
- обнаруживающие разыменование указателя, когда перед этим в трассе была проверка этого указателя на равенство нулю (DEREF_AFTER_NULL);
- обнаруживающие проверку на равенство нулю указателя, который перед этим был разыменован (NULL_AFTER_DEREF).

Были выделены следующие шаблоны исправления.

- 1) **FixCondition.** Если разыменование происходит в условии if/for/while, то добавить проверку в начало условия:

```

if (deref(ptr)) ... ⇒
if (ptr && deref(ptr)) ...
if (!ptr || deref(ptr)) ...

```
- 2) **InitWithAlternative.** Разыменование в правой части присваивания или декларации следует заменить на тернарное выражение (если разыменуемый указатель равен 0, присваиваем значение по умолчанию). Как показала практика, данный шаблон следует ограничить только случаями, когда в правой части находится указатель.

```

Data *data = obj->getData(); ⇒
Data *data = obj ? obj->getData() : 0;

```
- 3) **ReturnWithAlternative.** Аналогичен предыдущему шаблону: разыменование в return-выражении следует завернуть в if-else блок (если указатель равен 0, нужно вернуть значение по умолчанию).

```

return a->data; ⇒
if (a)
return a->data;
else
return 0;

```

- 4) **SimpleCheck.** Добавить проверку для инструкции, в которой происходит разыменование:

```

deref(underefable); ⇒
if (underefable)
deref(underefable);

```
- 5) **CheckWithRelated.** Добавить проверку для инструкции, в которой происходит разыменование, и нескольких следующих инструкций, семантически связанных с ней:

```

x = y->f;
z = x + 1;
f(z);
⇒
if (y) {
x = y->f;
z = x + 1;
f(z);
}

```
- 6) **MoveBreakingCheck** (только для дефектов типа NULL_AFTER_DEREF). Если простая проверка указателя на равенство 0 содержит выход из программы, функции, цикла, блока, то переместить эту проверку на позицию перед разыменованием:

```

char *buf = (char *)malloc(5);
memset(buf, '\0', 5);
...
if (!buf) {
goto err;
}
⇒
char *buf = (char *)malloc(5);
if (!buf) {
goto err;
}
memset(buf, '\0', 5);
...

```
- 7) **MovePositiveCheck** (только для дефектов типа NULL_AFTER_DEREF). Если указатель просто проверяется на неравенство 0, то переместить проверку на позицию перед разыменованием:

```

char *buf = (char *)malloc(5);
memset(buf, '\0', 5);
...
if (buf) {
print(buf);
}
⇒
char *buf = (char *)malloc(5);
if (buf) {
memset(buf, '\0', 5);
...
print(buf);
}

```

Все перечисленные шаблоны (кроме CheckWithRelated) уже реализованы в нашем инструменте.

Очевидно, нередко для одного и того же дефекта будут применимы сразу несколько из этих шаблонов. В системе Svace было принято решение выдавать пользователю не все возможные исправления, а только то, которое соответствует *наиболее приоритетному* шаблону. Когда автоматическое исправление применяется к большим проектам, такое решение снижает нагрузку на пользователя, которому нужно верифицировать сотни вариантов исправления. Шаблоны были упорядочены по приоритету следующим образом:

- 1) MovePositiveCheck;
- 2) MoveBreakingCheck;
- 3) InitWithRelated;
- 4) FixCondition;
- 5) ReturnWithRelated;
- 6) SimpleCheck.

Как было произведено упорядочение? Во-первых, некоторые из рассматриваемых шаблонов не пересекаются по условиям (например, MovePositiveCheck/MoveBreakingCheck или FixCondition/ReturnWithRelated), и их относительный порядок не важен. Также более приоритетным считался шаблон с более строгими условиями (например, SimpleCheck применим в большинстве случаев, поэтому его приоритет самый низкий). Во всех неоднозначных ситуациях относительный порядок основывался на экспертной оценке.

Результаты автоматического исправления разыменований нуля, найденных в проекте Firefox-53.0.3, представлены в Табл. 1.

Табл. 1. Автоматическое исправление разыменований нуля

Table 1. Automatic repair of NULL dereferences

Checker Name	Defects	Fixes	Reviewed	Good	Not Required	Bad	Needs Join	Good Required
NULL_AFTER_DEREF	108	49	49	76%	4%	12%	8%	79%
DEREF_AFTER_NULL	249	109	109	94%	2%	3%	1%	95%
DEREF_AFTER_NULL. EX	471	426	127	11%	81%	5%	3%	58%
DEREF OF NULL.EX	78	36	36	44%	39%	14%	3%	73%

Значения столбцов:

- **Defects** – общее число найденных дефектов данного типа;
- **Fixes** – число предложенных исправлений;
- **Reviewed** – число проанализированных исправлений;
- **Good** – доля приемлемых исправлений;
- **Not Required** – доля случаев, когда исправление не нужно (ложное срабатывание детектора);
- **Bad** – доля неверных исправлений;
- **Needs Join** – доля семантически верных исправлений, которые, тем не менее, нужно объединить с соседними (т. е., например, если несколько соседних выражений обрабатываются в одно и то же условие, то следует объединить их в блок, который и поместить под это условие);
- **Good Required** – доля приемлемых исправлений для истинных дефектов.

Таким образом, реализованный алгоритм автоматического исправления дефектов заданных четырех типов показал приемлемое качество для истинных дефектов (см. последний столбец). Однако в ряде случаев процент адекватных исправлений среди всех предложенных вариантов оказался низким из-за некачественной работы самого детектора, т.е. большого числа ложных срабатываний, и отсеять эти ложные срабатывания за счёт предусловий шаблонов исправления также не удалось.

6. Заключение и направление дальнейших исследований

Нами был реализован инструмент, предлагающий варианты исправления для дефектов разных типов, найденных статическим анализатором Svace, как в относительно простых случаях (неинициализированная переменная), так и в довольно сложных (разыменование нулевого указателя). Инструмент показал приемлемое качество работы и производительность.

В дальнейшем предполагается реализовать автоматическое исправление большего числа дефектов. Однако пользователю интересны в первую очередь сложные дефекты (переполнение буфера, утечка ресурсов и т. п.), которые по трудности исправления сопоставимы с разыменованием нуля. Разрабатывать для каждого из них свой набор шаблонов, привлекая экспертов, очень накладно. Поэтому есть надежда генерировать шаблоны для того или иного типа дефектов автоматически. Для этого требуется большой проект, имеющий длинную историю изменений и давно анализируемый анализатором Svace. Идея состоит в том, чтобы сопоставить найденные в коде дефекты с коммитами в репозитории и получить возможные исправления дефектов того или иного типа. Если полученные исправления можно будет объединить в кластеры, это будут возможные шаблоны автоматического исправления.

Список литературы / References

- [1] Lee J., Hong S., Oh H. MemFix: Static analysis-based repair of memory deallocation errors for C. In Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 95-106.
- [2] Xu X., Sui Yu. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In Proc. of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 512-523.
- [3] Petke J., Haraldsson S.O. et al. Genetic Improvement of Software: A Comprehensive Survey. IEEE Transactions on Evolutionary Computation, vol. 22, no. 3, 2018, pp. 415-432.
- [4] Durieux T., Cornu B. Dynamic patch generation for null pointer exceptions using metaprogramming. In Proc. of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 349-358.
- [5] Belevantsev A., Borodin A. et al. Design and development of svace static analyzers. In Proc. of the Ivannikov Memorial Workshop, 2018, pp. 3-9.
- [6] Иванников В.П., Белеванцев А.А. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7 / Ivannikov V.P., Belevantsev A.A. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computing Software, vol. 40, issue 5, 2014, pp. 265-275.
- [7] Kang H.J., Thung F. et al. Semantic Patches for Java Program Transformation. In Proc. of 33rd European Conference on Object-Oriented Programming (ECOOP), 2019, pp. 1-27.
- [8] Padiou Y., Hansen R.R. et al. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In Proc. of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems, 2006, 6 p.
- [9] Wang C., Jiang J. et al. Transforming Programs between APIs with Many-to-Many Mappings. In Proc. of the 30th European Conference on Object-Oriented Programming (ECOOP), 2016, 26 p.
- [10] Савченко В. В., Сорокин К. С. и др. Nobrainer: инструмент преобразования C/C++ кода на основе примеров. Программирование, том 46, no. 5, 2020 г., стр. 33-46 / Savchenko V.V., Sorokin K.S. et al. Nobrainer: A Tool for Example-Based Transformation of C/C++ Code. Programming and Computer Software, vol. 46, no. 5, pp. 362-372.
- [11] Wasserman L. Scalable, example-based refactorings with Refaster. In Proc. of the ACM Workshop on Refactoring Tools, 2013, pp. 25-28.
- [12] CWE-476: NULL Pointer Dereference. URL: <https://cwe.mitre.org/data/definitions/476.html>.

Информация об авторах / Information about authors

Сергей Владимирович СЫРОМЯТНИКОВ – научный сотрудник. Сфера научных интересов: статический анализ исходного кода программ, автоматическое исправление дефектов кода.

Sergey Vladimirovich SYROMIATNIKOV – Researcher. Research interests: static analysis of source code, automatic repair of code defects.