Санкт-Петербургский государственный университет

Осипова Александра Вадимовна

Выпускная квалификационная работа

Исследование инструментов фаззинга для генерации модульных тестов на Java

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2017 «Программная инженерия»*

Научный руководитель:
доц. кафедры СП, к.т.н. Ю.В. Литвинов

Рецензент:
Руководитель отдела ООО «Техкомпания Хуавей» Д.А. Иванов

Санкт-Петербург
2021

Saint Petersburg State University

Aleksandra Osipova

Bachelor's Thesis

Fuzzing tools research for unit test generation in Java

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2017 "Software Engineering"*

Scientific supervisor:
C.Sc., docent Y.V. Litvinov

Reviewer:
Department Manager at "Huawei" D.A. Ivanov

Saint Petersburg
2021

Оглавление

1. Введение

На сегодняшний день создано огромное количество программных продуктов, и их количество продолжает расти. На создание, тестирование, сопровождение требуются значительные ресурсы; если проект содержит миллионы строк программного кода, то ресурсов может потребоваться немыслимо много, и даже в таком случае выпустить продукт без дефектов и поддерживать его на должном уровне качества — задача трудновыполнимая [23]. При этом цена на обнаружение и исправление ошибки с течением времени значительно растёт [20], и не обнаруженный вовремя дефект может приводить к миллионным убыткам. Например, ошибка в программном обеспечении Маринера-1[1] стоила NASA сотни миллионов долларов.

Программисты, даже самые высококвалифицированные, совершают ошибки. Поэтому для повышения качества кода и минимизации количества дефектов конечного продукта необходимо проводить тестирование, начиная с самых ранних этапов разработки. Модульное тестирование — вид тестирования, ориентированный на обнаружение дефектов и программных регрессий в отдельных модулях системы [4]. Однако создание модульных тестов и поддержание их в актуальном состоянии — трудоёмкая деятельность, к тому же часто требующая написания большого количества шаблонного кода. Поэтому актуальна задача по автоматизации этого процесса.

Имеется множество инструментов, решающих данную задачу [9, 17, 21], но покрытие кода при их использовании не максимально и может быть улучшено [15]. В текущей работе для повышения покрытия рассматривается идея комбинирования уже существующих инструментов генерации модульных тестов с *фаззингом* — исполнением программы с передачей на вход неправильных, неожиданных или случайных данных[2]. Также представляет интерес то, каким образом различные виды

[1]Маринер-1 был уничтожен через 293 секунды после старта: антенна аппарата потеряла связь с наводящей системой на Земле, в результате управление взял на себя бортовой компьютер, программа которого содержала ошибку, https://en.wikipedia.org/wiki/Mariner_1, дата обращения: 7.12.2020.

[2]https://en.wikipedia.org/wiki/Fuzzing, дата обращения: 8.12.2020.

фаззеров работают с различными алгоритмами генерации тестов, что в дальнейшем может быть использовано для создания собственного инструмента для генерации тестов.

В рамках данной работы будут рассмотрены основные инструменты для генерации модульных тестов и фаззинга, а также создана система, совмещающая эти инструменты, которая должна повысить покрытие тестируемого кода и позволить исследовать совместную работу инструментов.

2. Постановка задачи

Цель данной работы — создание системы, совмещающей инструменты фаззинга и генерацию модульных тестов для Java, и исследование их совместной работы. Для достижения этой цели были поставлены следующие задачи:

- создать систему, совмещающую инструменты фаззинга и генерации модульных тестов, выбрать инструменты фаззинга и генерации тестов и интегрировать их в созданную систему;
- наладить работу системы в инфраструктуре Java Unit Testing Tool Competition;
- произвести сравнение совместной работы выбранных фаззеров и генераторов, сопоставить их работу с другими инструментами для генерации модульных тестов.

3. Обзор

3.1. Терминология

В этой секции изложены основные определения предметной области работы в контексте объектно-ориентируемого программирования.

Модульный тест — исполняемый фрагмент кода, проверяющий функциональность тестируемого класса. Тест состоит из инструкций (англ. *statements*), которые могут генерировать объекты через конструкторы, обращаться к полям и методам. Размер одного генерируемого теста обычно варьируется, как и количество тестов в тестовом наборе.

Тестовый набор (англ. *test suite*) представляет из себя набор сгруппированных тестов.

Регрессионный тест — тест, фиксирующий текущее состояние программы.

Генератор модульных тестов — инструмент, автоматически создающий модульные тесты.

Фаззинг (англ. *fuzzing*[3]) — исполнение тестируемой программы с входными данными, выбранными из пространства, выступающего за пределы пространства *ожидаемых* этой программой входных данных [2].

Фаззер — программа, проводящая тестирование с использованием фаззинга.

3.2. Фаззинг

В контексте работы фаззинг используется как процесс, генерирующий входные данные. Таким образом, модульный тест может быть написан или вручную, или создан автоматически другим инструментом, а затем запущен несколько раз с разными сгенерированными значениями аргументов.

[3]В переводе с английского «fuzzy» означает «нечёткий, неопределённый, пушистый».

3.2.1. История

Термин «fuzz» появился в 1988 году в Университете Висконсина. На семинаре Бартона Миллера была создана программа fuzzer для тестирования надежности приложений под Unix, эта программа генерировала случайные данные, которые передавались как параметры для тестируемых программ до тех пор, пока они не завершали выполнение с ошибкой[4].

Однако, можно утверждать, что процесс фаззинга гораздо старше. Так, американский учёный Джеральд Вайнберг рассказывает о том, как еще в 50-х годах они в качестве стандартной практики при тестировании использовали либо взятые из мусорного ведра перфокарты, либо перфокарты случайных чисел, что помогало обнаруживать нежелательное поведение тестируемой программы[5].

3.2.2. Виды фаззинга

Выделяют три вида фаззинга: фаззинг чёрного ящика, фаззинг серого ящика и фаззинг белого ящика [2, 11], в зависимости от того, сколько информации требуется фаззеру от тестируемой программы во время исполнения. Но как и в случае с подходами для генерации тестов, разделение весьма условно, на практике, например, фаззеры белого ящика часто используют некоторые аппроксимации.

Термин «черный ящик», обычно использующийся при тестировании программного обеспечения, в случае с фаззингом обозначает аналогичное: фаззер во время своей работы не получает никакой информации от тестируемой программы, он может взаимодействовать с программой только через ввод/вывод программы. Большинство традиционных фаззеров относят к этой категории [2].

Фаззинг белого ящика — противоположность чёрного. Здесь программа анализирует внутреннее устройство тестируемой программы.

[4] https://en.wikipedia.org/wiki/Fuzzing, дата обращения: 15.12.2020.

[5] Блог преподавателя психологии и антропологии разработки ПО Джеральда Вайнберга, http://secretsofconsulting.blogspot.com/2017/02/fuzz-testing-and-fuzz-history.html, дата обращения: 16.12.2020.

Такой фаззер использует метод, который теоретически может исследовать все пути выполнения в тестируемой программе. В отличие от фаззинга черного ящика, фаззингу белого ящика требуется информация от тестируемой программы и она используется для руководства генерацией тестов. В частности, начиная исполнение с заданным конкретным входом, фаззер белого ящика сначала собирает символьные ограничения для всех условных операторов на пути исполнения. Следовательно, после одного исполнения такой фаззер объединяет все символьные ограничения с помощью конъюнкции для формирования ограничения пути. Затем фаззер белого ящика последовательно отрицает одно из ограничений и решает новое ограничение пути.

Накладные расходы на фаззинг белого ящика обычно значительно превосходят расходы на фаззинг черного ящика. Это связано с тем, что реализации динамического символьного исполнения часто используют динамические инструменты и SMT-решатели [2], что весьма трудоёмко.

Фаззинг серого ящика — что-то среднее между первыми двумя видами. Фаззер серого ящика может получать *некоторую* информацию о внутренней структуре тестируемой программы и/или её исполнении. Фаззеры серого ящика полагаются на приблизительную информацию, чтобы увеличить скорость и иметь возможность тестировать программу на большем количестве входных данных [2].

3.2.3. Инструменты

В целом существует большое количество фаззеров. Так в обзорной статье по фаззингу [2] упоминается свыше 60 инструментов. Но несмотря на то, что инструменты фаззинга имеют успех даже в реальных проектах[6], большинство фаззеров ориентированно на языки C, C++, то есть на языки с ручным управлением ресурсами, в то же время для Python, Java, C# фаззеры практически не представлены. Что интересно, имеющиеся для Java фаззеры чаще всего были портированы с других языков.

[6] http://lcamtuf.coredump.cx/afl/#bugs, дата обращения: 15.12.2020.

Одним из самых известных фаззеров является фаззер серого ящика AFL, и это весьма заслуженно, ведь он прост в использовании, но при этом эффективен, что не раз подтверждалось на практике [11]. Данный фаззер поддерживает фаззинг программ, написанных на C, C++ и Objective C, скомпилированных с помощью и GCC, и CLang, но его часто расширяют и портируют. Так в мире Java фаззеров появились Kelinci [14] и java-afl[7], предоставляющие интерфейс на Java для вызова AFL. Также существует модульная библиотека для фаззинга [18], в которой можно выбрать либо уже имеющийся фаззер, либо добавить свой. В данной библиотеке также есть возможность подключить AFL. Кроме AFL, имеется Zest [22], PerfFuzz [19] (расширение AFL) и некоторые другие инструменты, но они либо находятся в стадии прототипа, либо не имеют документации, поэтому не входят в текущий обзор. Отличительная особенность инструмента Zest в том, что он, используя junit-quickcheck[8] (создатели которого вдохновились библиотекой QuickCheck[9] для Haskell), позволяет генерировать структурированные входные данные, а не поток байтов, как в случае с AFL и его производными.

Фаззер белого ящика jFuzz [25] создан поверх Java Pathfinder, системы, верифицирующей байт-код Java. Но сами авторы фаззера говорят, что это только прототип, поэтому они не гарантируют стабильное поведение.

Также нашлось два инструмента с одинаковым названием JavaFuzz, и один[10] обновлялся довольно давно, в 2016 году, а на второй[11], фаззер серого ящика, можно обратить внимание при проведении экспериментов.

3.3. Генерация тестов

Основная задача при генерации модульных тестов для объектно-ориентированного кода состоит в том, чтобы создать небольшие наборы

[7] https://github.com/Barro/java-afl, дата обращения: 15.12.2020.
[8] https://github.com/pholser/junit-quickcheck, дата обращения: 17.12.2020.
[9] https://hackage.haskell.org/package/QuickCheck, дата обращения: 17.12.2020.
[10] https://github.com/cphr/javafuzz, дата обращения: 15.12.2020
[11] https://github.com/fuzzitdev/javafuzz, дата обращения: 15.12.2020

тестов, максимизирующие покрытие кода тестируемого класса. Общепринятыми методами решения этой задачи являются случайное тестирование [1, 13], тестирование на основе поиска (англ. *search-based testing*) [10, 13] и символьное исполнение [3, 12, 13].

Стоит заметить, что различия между этими методами часто нечеткие. Так, при символьном исполнении для выбора следующего пути могут использоваться различные методы поиска.

3.3.1. Случайное тестирование

Случайное тестирование [1], пожалуй, самый незамысловатый способ генерации тестов. В базовом алгоритме для объектно-ориентированного класса генерация состоит из создания последовательности вызовов выбранных случайным образом методов класса, и для каждого аргумента метода случайным образом выбирается ранее созданный объект нужного типа или создаётся новый экземпляр объекта требуемого типа путём вызова одного из его конструкторов или фабричных методов (также выбирается случайным образом).

Существует усовершенствованный подход — управляемое случайное тестирование. В нём генерация начинается со случайных входных данных в качестве аргументов, а затем используются некоторые дополнительные знания для получения новых входных данных. Одним из основных представителей этой категории методов является случайное тестирование с обратной связью [8], которое улучшает генерацию тестов за счёт полученной с предыдущих итераций обратной связи, которая направляет процесс создания входных данных на последующих итерациях. Это позволяет снизить количество повторяющихся и/или недопустимых входных данных [13], используемых для тестирования.

Инструменты для генерации модульных тестов, использующие методы случайного тестирования, обладают преимуществами масштабируемости и простоты реализации, но используют много ресурсов на генерацию недопустимых, бесполезных (не улучшающих покрытие) входных данных ввиду случайной природы методов. Добавление обратной связи позволяет нивелировать данный недостаток [13].

Как в исследованиях, так и в реальных проектах в качестве эталонной реализации метода случайного тестирования используется инструмент Randoop[12] [6, 8, 13, 15, 17]. Данный инструмент является одним из самых стабильных, поддерживаемых по сей день инструментов для генерации тестов для Java. Также он достаточно прост в использовании, запускается с помощью несложных инструкций из консоли и позволяет создать для класса модульные тесты в JUnit формате. Randoop реализует подход случайной генерации с обратной связь и может использоваться как для регрессионного тестирования, так и для обнаружения дефектов.

3.3.2. Тестирование на основе поиска

При тестировании на основе поиска задача генерации тестов решается с помощью алгоритмов поиска, например, генетических алгоритмов или алгоритма имитации отжига. В таком случае генерация входных данных формулируется как проблема поиска (англ. *search problem*), множество возможных входных значений формирует пространство решений, а метрика, которую хочется максимизировать, например, покрытие кода, кодируется как функция приспособленности. Таким образом, поиск будет осуществляться с помощью выбранной функции приспособленности, которая представляет из себя эвристику, оценивающую, насколько близко текущее решение к оптимальному. Руководствуясь функцией приспособленности, алгоритм поиска итеративно выдаёт лучшие решение до тех пор, пока либо не будет найдено оптимальное решение, либо не будет выполнено условие остановки (например, истечение выделенного на генерацию времени).

На данный момент важнейшее место среди инструментов генерации модульных тестов на Java (всех, а не только среди использующих поход тестирования на основе поиска) занимает инструмент EvoSuite [6, 9, 10, 13]. Помимо удобства использования (например, можно запускать как из консоли, так и из интегрированных сред разработки Eclipse,

[12] https://github.com/randoop/randoop, дата обращения: 13.12.2020.

IntelliJ IDEA), полностью автоматизированного запуска (не нужно как-либо подготавливать тестируемые классы), стабильности, EvoSuite является победителям в соревнованиях Java Unit Testing Tool Competition последних нескольких лет [7, 15]. EvoSuite использует генетический алгоритм для генерации тестов формата JUnit, ориентируясь на метрики покрытия кода.

Наборы тестов являются особями популяции. Генетический алгоритм итеративно отбирает особей на основе их приспособленности. Затем к выбранным особям применяется процессы кроссинговера и мутации. От поколения к поколению приспособленность особей постепенно улучшается, пока либо не будет найдено решение, либо поиск не будет прекращен другим способом (например, когда он достигнет фиксированного предела количества поколений). Кроссинговер представляет собой обмен тестами между двумя родительскими тестовыми наборами, в то время как мутация может произвольно обновлять набор тестов, добавляя новые, отбрасывая или изменяя существующие. Изменение существующего теста в таком случае включает удаление, изменение или вставку инструкций (например, вызовов методов тестируемого класса).

3.3.3. Символьное исполнение

Символьное исполнение — это аналитический подход, основанный на правилах вывода. Он использует символьные переменные как входные данные программы и представляет значения программных переменных как символьные выражения, а путь исполнения — выражением над символьными переменными (ограничением пути) [16]. Символьное исполнение применяется для поиска всех возможных путей исполнения программы. Данный подход способен находить пути исполнения программы, которые вызывают ошибки, даже без компиляции. Для решения ограничений пути, как правило, применяется SMT-решатель, который вычисляет уже конкретные значения переменных, необходимых для того, чтобы поток исполнения пошёл по данному пути. Несмотря на очевидные преимущества и силу, этот подход обычно реализуется в мелком масштабе, так как в крупных проектах наблюдается значительный

рост всех возможных путей исполнения, что требует серьёзных вычислительных ресурсов [12]. Именно по этой причине этот подход только недавно стал применяться для генерации тестов, хотя идея символьного исполнения появилась в 70-х годах.

С момента проведения экспериментальных исследований инструментов генерации модульных тестов для Java в статьях [6,13] в области инструментов, использующих метод символьного исполнения, не многое поменялось. Последние изменения в инструменте CATG [24] произошли в начале 2018 года. Инструмент SPF [5] планомерно развивается, но у обоих инструментов не появилось исчерпывающей документации и стабильности относительно таких гигантов как Randoop и EvoSuite.

3.3.4. Сравнение

Большинство современных решений используют случайные и эволюционные алгоритмы для генерации модульных тестов. Проведённое в 2017 году сравнение подходов случайного тестирования и тестирования на основе поиска на проекте с закрытым исходным кодом LifeCalc показало, что тестирование на основе поиска с использованием EvoSuite обнаружило не более 56% ошибок, а случайное тестирование с использованием Randoop — не более 38%. Так как зрелого инструмента, использующего символьный подход для генерации тестов для Java проектов, на тот момент не было, исследователи не включили его в сравнение [13]. Однако символьное тестирование является достаточно многообещающим подходом, так KLEE, инструмент для генерации тестов для C, при символьном тестировании ядра HiStar показал неплохой результат покрытия строк кода в 76,4%, в то время как результат случайного тестирования — 48,0% [3].

Несмотря на терминологию, и случайные, и эволюционные алгоритмы для генерации модульных тестов можно отнести к случайному подходу генерации, так как эти методы генерируют случайные последовательности вызовов, преобразующих объекты классов в произвольные состояния. Как следствие, тестовый набор, созданный с помощью направляемых случайных или эволюционных методов, может лишь ча-

стично фиксировать поведение программы, в отличие от символьного подхода. В то же время и символьный подход имеет свои недостатки, в частности требовательность к ресурсам.

3.3.5. Вывод

В целом, существующие решения показывают неплохие результаты по покрытию кода, выявлению ошибок и скорости работы, но результаты не максимальны, а значит, могут остаться необнаруженные дефекты, приводящие к сбоям, которых хотелось бы избежать. Поэтому важно продолжать работу в области генерации модульных тестов, чему и посвящена данная ВКР.

4. Особенности реализации системы

4.1. Стек технологий

В качестве основного языка разработки системы, совмещающей инструменты генерации тестов и фаззинга, был выбран Kotlin. Так как данная работа фокусируется на генерации модульных тестов для проектов, написанных на языке Java, то разумно для поставленной задачи рассматривать языки JVM. Выбор пал на Kotlin ввиду того, что он обеспечивает более лаконичный код, чем, например, сам язык Java, без ущерба для производительности или безопасности.

Для парсинга и построения AST выбрана библиотека JavaParser по причине наличия хорошей документации и её специализации на Java, в отличие, например, от ANTLR[13].

В качестве системы сборки используется Gradle[14], он превосходит Maven по производительности[15], а также позволяет писать инструкции к сборке на Kotlin.

4.2. Процесс создания теста

Так как в текущей работе рассматривается идея комбинирования уже существующих инструментов генерации модульных тестов с фаззингом, то процесс создания модульного теста может быть разбит на несколько этапов. Диаграмма на Рис. 1 отражает данные этапы.

Сначала происходит генерация модульного теста одним из предназначенных для этого инструментов. На выходе имеется набор Java-файлов с классами, содержащими тестовые методы. Далее осуществляется парсинг имеющихся файлов, строятся AST. Следующим этапом в разобранных тестах происходит поиск мест, значения которых будут предаваться фаззеру: строки, различные примитивные типы. Это могут быть как аргументы конструкторов сложных объектов, так и вызывае-

[13] https://www.antlr.org/about.html, дата обращения: 17.12.2020.
[14] https://docs.gradle.org/current/userguide/what_is_gradle.html, дата обращения: 17.12.2020.
[15] https://gradle.org/maven-vs-gradle/, дата обращения: 17.12.2020.

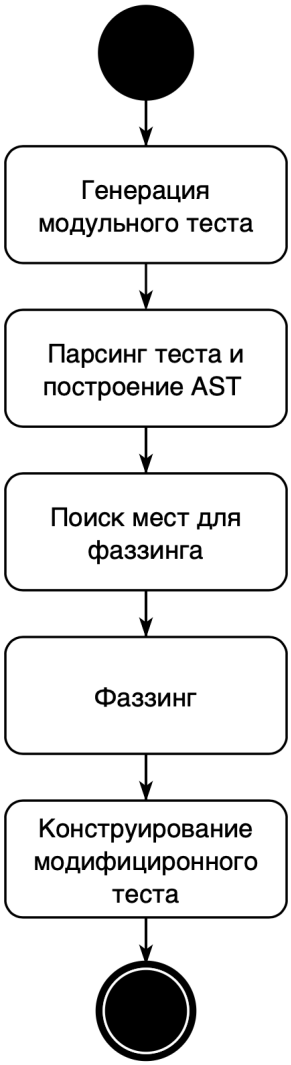

Рис. 1: Диаграмма активностей, отображающая процесс создания теста с модифицированными входными значениями

мых в процессе тестового сценария методов. После завершения процесса фаззинга для метода имеется набор новых значений, при подстановке которых появляются новые сценарии, способные улучшить метрики покрытия. Остаётся сконструировать модифицированный тест: поместить сгенерированные фаззером значения в скелет оригинального теста, построить новые утверждения (англ. *asserts*), сформировать тестовый класс и записать в файл.

4.3. Архитектура системы

Диаграмма на Рис. 2 отражает общую архитектуру системы.

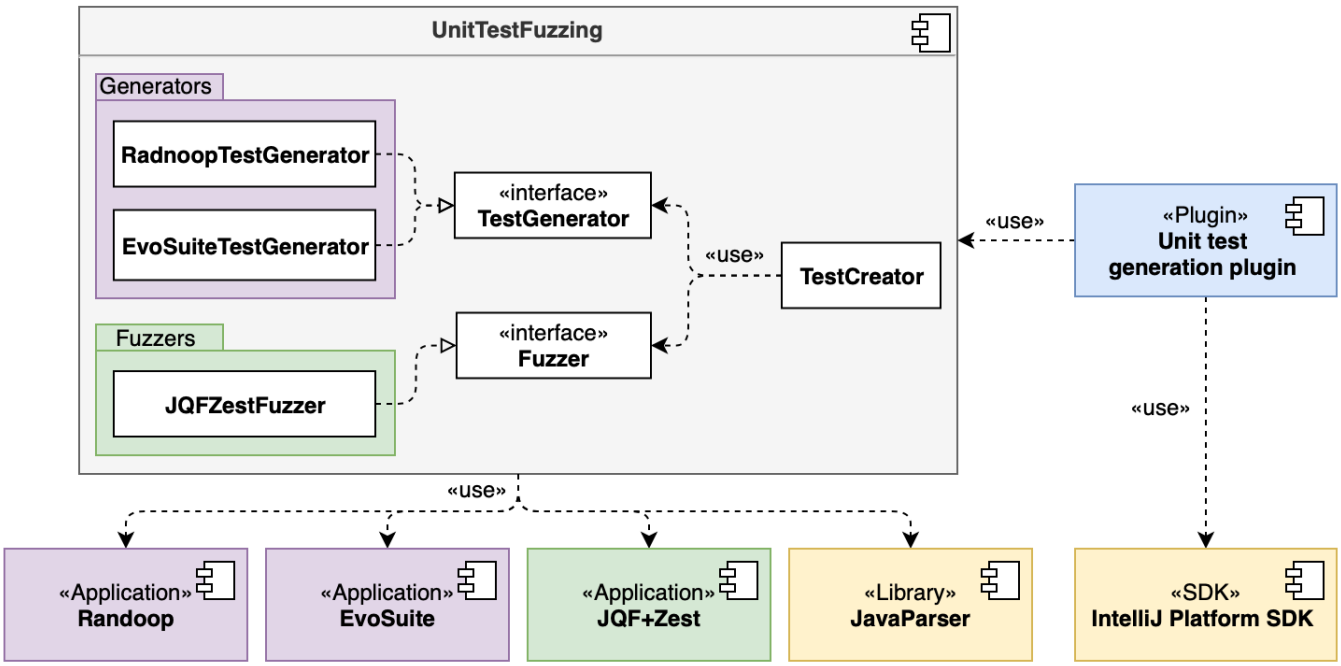

Рис. 2: Диаграмма компонентов, иллюстрирующая архитектуру созданной системы

Основные элементы данной системы — интерфейсы генератора модульных тестов и фаззера, взаимодействие между которыми и является основной задачей системы. Интерфейсы унифицируют работу с различными существующими инструментами, таким образом, могут быть интегрированы и другие генераторы и фаззеры.

С помощью системы создавать модульные тесты для класса можно как из командной строки, так и используя плагин для IntelliJ IDEA.

4.4. Детали реализации

4.4.1. Генераторы

В рамках данной работы интерфейс генератора был реализован для двух инструментов модульных тестов: Randoop и EvoSuite, упомянутые в обзоре эталонные представители случайного и эволюционного подходов соответственно. Оба инструмента генерируют тесты в формате JUnit 4. При запуске системы с помощью флага имеется возможность

выбрать, какой из генераторов использовать. Запуск инструментов реализован через класс CommandExecutor, который исполняет переданную команду в необходимой директории и логирует процесс исполнения, перенаправляя поток вывода, что позволяет более безболезненно добавлять и другие инструменты.

4.4.2. Фаззинг

В качестве инструмента для фаззинга был интегрирован фаззер серого ящика JQF+Zest. Так как в отличие, например, от знаменитого AFL, он генерирует не просто InputStream, а сразу значения необходимых типов, что идеально подходит для решения поставленной задачи.

JQF+Zest способен анализировать то, насколько успешно протекает процесс фаззинга, направляя его, за счёт чего на выходе имеются не просто значения нужных типов, а уникальные значения, которые так или иначе улучшают покрытие исходного тестируемого кода. Чтобы этого добиться, требуется написать тестовый драйвер, который содержит в себе метод с аннотацией @Fuzz. Аргументы этого метода JQF+Zest и будет фаззить, а исходя из информации об исполнении тела метода будет направлять процесс фаззинга.

Таким образом, в системе необходимо было динамически генерировать нужные драйвера. Что и было сделано с помощью конструирования требуемых классов во время исполнения с использованием библиотеки JavaParser, их компиляции и передачи фаззеру. Для агрегации результатов работы фаззера также генерируется нужный класс, который с помощью модифицированного ObjectOutputStream, сохраняет все полученные наборы значений для каждого конкретного метода.

4.4.3. Построение утверждений

Для того, чтобы тесты являлись тестами необходимы утверждения, которые и будут проверяться во время тестирования. В созданной системе процесс конструирования регрессионных утверждений для новых сгенерированных тестовых методов состоит из удаления устаревших утвер-

ждений, запуска тестов без них, получения и фиксации значений методов и объектов в том состоянии, в котором тестируемая программа находится в данный момент, и, наконец, создания новых утверждений.

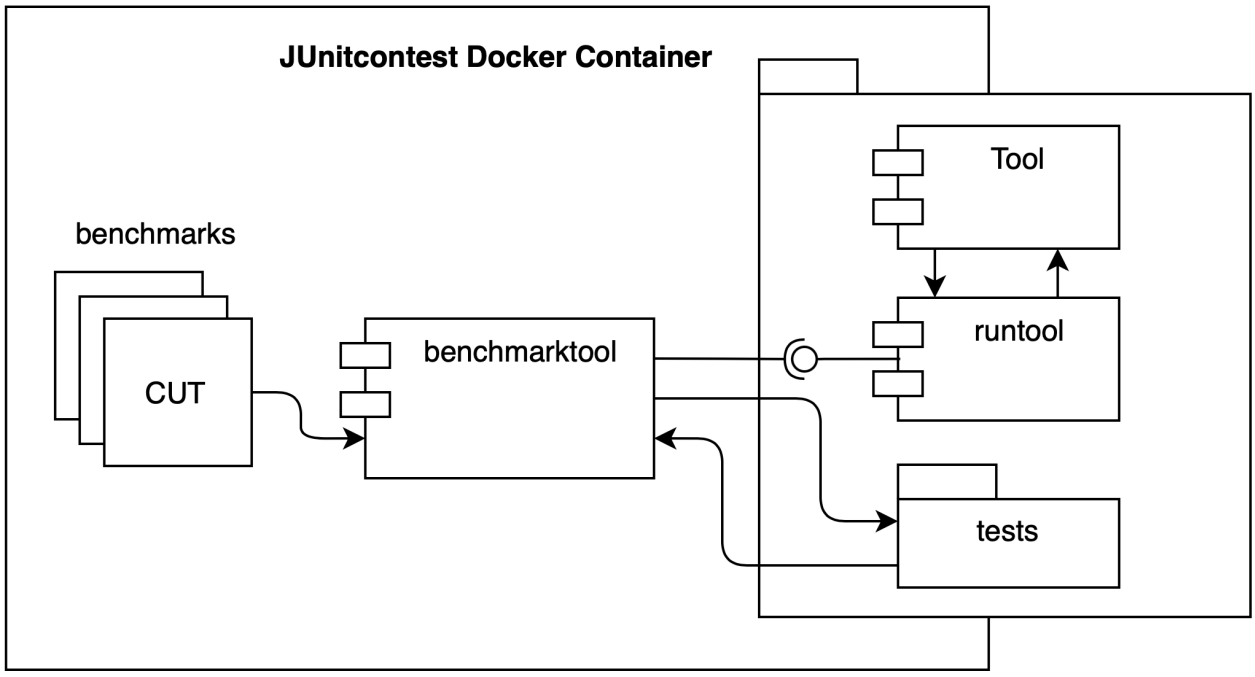

Рис. 3: Диаграмма архитектуры Java Unit Testing Tool Competition

5. JUnit Testing Tool Competition

Java Unit Testing Tool Competition — соревнования, проводимые ежегодно для стимуляции интереса к проблеме автоматической генерации тестов для программ, написанных на Java. Организаторы создали специальную инфраструктуру, с помощью которой они проводят эксперименты в рамках соревнований. Каждый инструмент-участник запускается на наборе классов Java, взятых из проектов с открытым исходным кодом. У инструментов есть фиксированный бюджет времени для создания тестов для каждого класса. Оценка рассчитывается специальным образом на основе эффективности и результативности. Помимо этого сама инфраструктура находится в открытом доступе, поэтому каждый желающий может использовать её для тестирования своих наработок.

Ранее была возможность запуска только в операционной системе Linux, но несколько лет назад организаторы полностью перенесли инфраструктуру в Docker-контейнер.

На Рис. 3 представлена архитектура инфраструктуры соревнова-

ний. При запуске инфраструктуры вся работа происходит в Docker-контейре, при этом ввиду монтирования необходимых локальных папок сохраняется удобство работы. Например, при изменении версии инструмента достаточно обновить его локально, нет необходимости перезапуска самого контейнера. Также есть возможность добавлять свои классы, для которых будут генерироваться тесты.

Для запуска инструмента в инфраструктуре необходимо реализовать интерфейс runtool, который имеет несложный протокол взаимодействия с benchmarktool, программой, ответственной за проведение экспериментов, а также удовлетворить некоторым требованиям, например, таким как версия сгенерированных JUnit-тестов и их расположение.

В рамках данной работы была обновлена (переведена на более свежую версию инструмента) реализация интерфейса runtool для Randoop (для него организаторы сами реализовали интерфейс в качестве примера участникам), налажена работа генератора EvoSuite и реализованы все необходимые требования для созданной в рамках данной ВКР системы, что в теории позволяет участвовать в последующих соревнованиях, а также проводить эксперименты.

6. Эксперименты

6.1. Постановка экспериментов

Для экспериментов использовался компьютер с процессором 6-Core Intel Core i7, тактовой частотой 2,2 GHz, RAM DDR4 с объемом памяти 16Gb, под управлением ОС macOS Big Sur 11.2.3. Эксперименты проводились в Docker-контейнере инфраструктуры Java Unit Testing Tool Competition, подсчёт метрик производился с использованием библиотеки JaCoCo.

6.1.1. Набор данных

При выборе классов для тестирования генераторов модульных тестов важно было учитывать насколько факторов: соответствие реальному ПО (то есть это не должны быть искусственно подобранные примеры кода), охват различных областей разработки, нетривиальность (в противном случае эксперименты бесполезны). Согласно этим, и некоторым другим критериям в Java Unit Testing Tool Competition 2020 года [7] был отобран набор классов из следующих проекта: Fescar/Seata[16], Guava[17], Spoon[18]. Для проведения экспериментов в рамках работы было взято случайное подмножество классов с Java Unit Testing Tool Competition 2020 года в количестве 10 штук.

6.1.2. Временной бюджет

Для каждого инструмента было проведено по 3 запуска на каждом из классов с временным бюджетом в 300 секунд. При этом у совмещенных инструментов из отведённых 300 секунд 30 секунд отводилось на работу генератора тестов. Также, чтобы проанализировать влияние фаззинга на результат, отдельно были подсчитаны метрики для тестов, которые были сгенерированы в течение отведённых раннее 30 секунд, как часть

[16] https://github.com/seata/seata, дата обращения: 03.05.2021
[17] https://github.com/google/guava, дата обращения: 03.05.2021
[18] https://github.com/INRIA/spoon/, дата обращения: 03.05.2021

процесса работы совмещенного инструмента.

6.1.3. Метрики

Так как цели генерации тестов — поиск ошибок и фиксация текущего состояния программы, в качестве основных метрик рассматриваются метрики покрытия строк кода и покрытия путей.

6.2. Результаты

На Рис. 4 можно наблюдать результаты проведённых экспериментов. Как можно заметить, результаты инструментов и этих же инструментов, но с применённым к ним фаззингом едва ли отличаются. Также стоит подчеркнуть, что за то же время сам инструмент справляется лучше. Можно сделать заключение, что в теории фаззинг может улучшить результаты работы генератора, но это вероятно только тогда, когда сам инструмент достигает пика своих возможностей, и притом фазеру даётся достаточно много времени. В рамках рабочей рутины или в контексте соревнований такие сценарии слабопредставимы.

Таким образом, идея применять фаззинг для улучшения эффективности существующих генераторов при небольшом временном бюджете не даёт желаемого прироста.

Инструмент	Временной бюджет	Среднее покрытие кода	Среднее покрытие условий	Медианное покрытие условий
utfRandoop	300	40.62	28.93	25.57
randoop	30	40.62	28.70	25.57
randoop	300	58.28	50.19	59.37
utfEvosuite	300	60.38	53.07	58.68
evosuite	30	60.38	53.07	58.68
evosuite	300	68.19	58.63	65.35

Рис. 4: Таблица с результатами экспериментов (временной бюджет в секундах, покрытие в процентах)

7. Заключение

В рамках данной работы были получены следующие результаты:

- создана система[19], совмещающая инструменты генерации тестов и фаззинга
  - добавлены генераторы Randoop, EvoSuite;
  - добавлен фаззер JQF+Zest;
- налажена работа в инфраструктуре Java Unit Testing Tool;
- поставлены эксперименты.

[19] https://github.com/Software-Analysis-Team/Unit-Test-Fuzzer, дата обращения: 6.05.2021.

Список литературы

[1] Arcuri A., Iqbal M. Z., Briand L. Random Testing: Theoretical Results and Practical Implications // IEEE Transactions on Software Engineering. — 2012. — Vol. 38, no. 2. — P. 258–277.

[2] The Art, Science, and Engineering of Fuzzing: A Survey / V. J. M. Manès, H. Han, C. Han et al. // IEEE Transactions on Software Engineering. — 2019. — P. 1–1.

[3] Cadar Cristian, Dunbar Daniel, Engler Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — OSDI'08. — USA : USENIX Association, 2008. — P. 209–224.

[4] Cohn Mike. Succeeding with Agile: Software Development Using Scrum. — 1st edition. — Addison-Wesley Professional, 2009. — ISBN: 0321579364.

[5] Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing Nasa Software / Corina S. Pundefinedsundefinedreanu, Peter C. Mehlitz, David H. Bushnell et al. // Proceedings of the 2008 International Symposium on Software Testing and Analysis. — ISSTA '08. — New York, NY, USA : Association for Computing Machinery, 2008. — P. 15–26. — Access mode: https://doi.org/10.1145/1390630.1390635 (online; accessed: 17.12.2020).

[6] Cseppentő Lajos, Micskei Zoltán. Evaluating code-based test input generator tools // Software Testing, Verification and Reliability. — 2017. — Vol. 27, no. 6. — P. e1627. — e1627 stvr.1627.

[7] Devroey Xavier, Panichella Sebastiano, Gambi Alessio. Java Unit Testing Tool Competition: Eighth Round // Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. — ICSEW'20. — New York, NY, USA : Association for

Computing Machinery, 2020. — P. 545–548. — Access mode: https://doi.org/10.1145/3387940.3392265 (online; accessed: 17.12.2020).

[8] Feedback-Directed Random Test Generation / Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, Thomas Ball // Proceedings of the 29th International Conference on Software Engineering. — ICSE '07. — USA : IEEE Computer Society, 2007. — P. 75–84. — Access mode: https://doi.org/10.1109/ICSE.2007.37 (online; accessed: 17.12.2020).

[9] Fraser Gordon, Arcuri Andrea. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software // Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. — ESEC/FSE '11. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 416–419. — Access mode: https://doi.org/10.1145/2025113.2025179 (online; accessed: 17.12.2020).

[10] Fraser G., Arcuri A. Whole Test Suite Generation // IEEE Transactions on Software Engineering. — 2013. — Vol. 39, no. 2. — P. 276–291.

[11] Fuzzing: State of the Art / H. Liang, X. Pei, X. Jia et al. // IEEE Transactions on Reliability. — 2018. — Vol. 67, no. 3. — P. 1199–1218.

[12] Godefroid Patrice. Test Generation Using Symbolic Execution // IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012) / Ed. by Deepak D'Souza, Telikepalli Kavitha, Jaikumar Radhakrishnan. — Vol. 18 of Leibniz International Proceedings in Informatics (LIPIcs). — Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012. — P. 24–33. — Access mode: https://drops.dagstuhl.de/opus/volltexte/2012/3845 (online; accessed: 17.12.2020).

[13] An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application / M. M. Almasi, H. Hemmati, G. Fraser et al. // 2017 IEEE/ACM 39th International Conference on Software

Engineering: Software Engineering in Practice Track (ICSE-SEIP). — 2017. — P. 263–272.

[14] Kersten Rody, Luckow Kasper, Păsăreanu Corina S. POSTER: AFL-Based Fuzzing for Java with Kelinci // Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. — CCS '17. — New York, NY, USA : Association for Computing Machinery, 2017. — P. 2511–2513. — Access mode: https://doi.org/10.1145/3133956.3138820 (online; accessed: 17.12.2020).

[15] Kifetew Fitsum, Devroey Xavier, Rueda Urko. Java Unit Testing Tool Competition: Seventh Round // Proceedings of the 12th International Workshop on Search-Based Software Testing. — SBST '19. — Montreal, Quebec, Canada : IEEE Press, 2019. — P. 15–20. — Access mode: https://doi.org/10.1109/SBST.2019.00014 (online; accessed: 17.12.2020).

[16] King James C. Symbolic Execution and Program Testing // Commun. ACM. — 1976. — Jul. — Vol. 19, no. 7. — P. 385–394. — Access mode: https://doi.org/10.1145/360248.360252 (online; accessed: 17.12.2020).

[17] Pacheco Carlos, Ernst Michael D. Randoop: Feedback-Directed Random Testing for Java // Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion. — OOPSLA '07. — New York, NY, USA : Association for Computing Machinery, 2007. — P. 815–816. — Access mode: https://doi.org/10.1145/1297846.1297902 (online; accessed: 17.12.2020).

[18] Padhye Rohan, Lemieux Caroline, Sen Koushik. JQF: Coverage-Guided Property-Based Testing in Java // Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. — ISSTA 2019. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 398–401. — Access mode: https://doi.org/10.1145/3293882.3339002 (online; accessed: 17.12.2020).

[19] PerfFuzz: Automatically Generating Pathological Inputs / Caroline Lemieux, Rohan Padhye, Koushik Sen, Dawn Song // Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. — ISSTA 2018. — New York, NY, USA : Association for Computing Machinery, 2018. — P. 254–265. — Access mode: https://doi.org/10.1145/3213846.3213874 (online; accessed: 17.12.2020).

[20] S.M.K Quadri, Farooq Sheikh Umar. Software Testing – Goals, Principles, and Limitations // International Journal of Computer Applications. — 2010. — 09. — Vol. 6.

[21] SUSHI: A Test Generator for Programs with Complex Structured Inputs / Pietro Braione, Giovanni Denaro, Andrea Mattavelli, Mauro Pezzè // Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. — ICSE '18. — New York, NY, USA : Association for Computing Machinery, 2018. — P. 21–24. — Access mode: https://doi.org/10.1145/3183440.3183472 (online; accessed: 17.12.2020).

[22] Semantic Fuzzing with Zest / Rohan Padhye, Caroline Lemieux, Koushik Sen et al. // Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. — ISSTA 2019. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 329–340. — Access mode: https://doi.org/10.1145/3293882.3330576 (online; accessed: 17.12.2020).

[23] Simmonds Devon M. Complexity and the Engineering of Bug-Free Software // Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS) / The Steering Committee of The World Congress in Computer Science, Computer .... — 2018. — P. 94–100.

[24] TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications / Haruto Tanno, Xiaojing Zhang,

Takashi Hoshino, Koushik Sen // Proceedings of the 37th International Conference on Software Engineering - Volume 2. — ICSE '15. — Florence, Italy : IEEE Press, 2015. — P. 717–720.

[25] jFuzz: A concolic whitebox fuzzer for Java / Karthick Jayaraman, D Harvison, Vijay Ganesh, Adam Kiezun // Proceedings of the First NASA Formal Methods Symposium. — 2009. — 01. — P. 121–125.