Санкт–Петербургский государственный университет

Садыков Рустам Ахметович

Выпускная квалификационная работа

Разработка виртуальной машины динамического символьного исполнения для языков программирования Java и Kotlin

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2019
«Современное программирование»

Научный руководитель:
доцент, факультет математики и компьютерных наук, к.ф.-м.н. Д. С. Шалымов

Рецензент:
старший инженер ООО «Техкомпания Хуавэй»
А. С. Меньшутин

Санкт-Петербург
2023 г.

Содержание

Введение

Символьное исполнение – это техника автоматизированного тестирования и анализа программного обеспечения, которая набирает популярность в последние годы. Она работает путем исследования различных путей выполнения программы для поиска входных данных, которые вызывают определенное поведение, ведущее к ошибкам, сбоям или нарушениям спецификаций программы.

Символьное исполнение необходимо, потому что традиционные методы тестирования, такие как ручное тестирование или простое случайное тестирование, часто неэффективны при обнаружении редких и сложных дефектов и уязвимостей в программных системах. В отличие от этого, оно может обеспечить более всестороннее покрытие пространства выполнения программы, что позволяет выявлять дефекты и уязвимости, которые традиционные методы тестирования упускают.

Однако данный метод может столкнуться с проблемами при анализе, поскольку требует значительных вычислительных ресурсов, особенно для сложных и больших программных систем. Более того, символьное исполнение анализирует только некоторую модель программы, что означает, что анализ может допускать ошибки, приводящие к ложным срабатываниям. Например, когда программа имеет большое количество путей выполнения или когда символьная виртуальная машина выполнения сталкивается с циклами, объем вычислений, необходимых для исследования всех возможных путей, может стать невозможным для дальнейшего анализа. Кроме того, иногда анализ ошибочно идентифицирует ошибки, когда они на самом деле их нет, что приводит к ложным тревогам. Это может привести к долгому времени анализа, его ненадежности или даже привести к сбою анализа, что затрудняет применение данной техники на практике.

Для решения этих проблем в данном дипломе представлена разработка виртуальной машины динамического символьного исполнения для языков программирования Java и Kotlin, расширяющую уже реализованную подсистему символьного исполнения подсистемой конкретного исполнения с целью уменьшения количества ложных срабатываний и увеличения покрытия

кода. Виртуальная машина (движок) реализует подход, который объединяет оба подхода, чтобы повысить эффективность и надежность анализа кода. Такая комбинация позволяет выполнять программу символьно и конкретно, уменьшая неточности чистого символьного исполнения и повышая качество анализа и покрытие анализируемой программы. Все это делает анализ более практичным и применимым для тестирования и анализа программного обеспечения.

Этот диплом описывает дизайн и реализацию виртуальной машины, включая архитектуру конкретного движка, структуры данных и алгоритмы. Также представлены экспериментальные результаты, которые позволяют дальнейший анализ эффективности виртуальной машины в обнаружении дефектов и уязвимостей в реальных программах.

Более того, подсистема конкретного исполнения, представленная в данном дипломе, интегрирована в инструмент UnitTestBot[1] – автоматизированную систему генерации модульных тестов для языков программирования Java и Kotlin. UnitTestBot реализует чистый символьный подход с расширенными возможностями, которые улучшают анализ кода и качество генерируемых тестов. Эта интеграция позволяет применять UnitTestBot более уверенно, обеспечивая эффективное тестирование больших и сложных программ и программных систем.

Инструмент UnitTestBot также предоставляет такую функциональность, как символьное исполнение, фаззинг, генерацию мок-объектов и минимизацию тестов, что дополнительно улучшает эффективность тестирования на основе динамического символьного исполнения. Подсистема конкретного исполнения разработана таким образом, чтобы легко интегрироваться в существующие рабочие процессы разработки данного продукта.

В целом, реализация виртуальной машины динамического символьного исполнения в UnitTestBot обеспечивает эффективное решение для автоматической генерации и анализа тестов. Этот диплом вносит вклад в развитие программной инженерии, предоставляя практический и эффективный подход для обеспечения качества и надежности разрабатываемых программ.

[1] UnitTestBot – инструмент автоматической генерации модульных тестов: https://www.utbot.org/

Постановка задачи

Целью данного диплома является разработка и реализация виртуальной машины динамического символьного исполнения для языков программирования Java и Kotlin с использованием уже реализованного символьного движка UnitTestBot, для более эффективной и робастной генерации тестов. Для достижения этой цели были поставлены следующие задачи:

1. выполнить обзор литературы о существующих инструментах на основе динамического символьного исполнения;
2. изучить структуру и кодовую базу инструмента UnitTestBot;
3. разработать архитектуру подсистемы конкретного исполнения;
4. реализовать разработанную архитектуру в рамках инструмента UnitTestBot.
5. интегрировать реализованную подсистему в UnitTestBot для получения виртуальной машины динамического символьного исполнения;
6. выяснить, может ли использование реализованного движка улучшить генерацию тестов путем увеличения покрытия кода и уменьшения количества неверных и ненадёжных тестов.

1. Обзор

В этой главе выполнен обзор основных концепций, используемых в динамическом символьном исполнении. А также обзор некоторых существующих инструментов, в частности инструмента UnitTestBot.

1.1. Символьное исполнение

Символьное исполнение [4, 7, 18] – широко используемая техника анализа программ, которая позволяет выполнять анализ в условиях неопределенных входных данных. *Неопределенные данные (символы)* – это абстракция конкретных объектов программы (обычно переменных), т.е. они могут принимать целый ряд значений, допустимых для типа данных объекта. Эти абстракции хранятся в специальной *символьной памяти*, управляемой символьным выполнением. Когда некоторая инструкция программы выполняется с использованием конкретных данных, результатом всегда является конкретный объект, и соответствующая инструкция, содержащая символьные аргументы, может генерировать другой символ или символьное выражение как её результат.

Рассмотрим символьное исполнение в рамках примера кода программы, предоставленной на рис. 1, которая вычисляет абсолютное значение целочисленной переменной. В данном случае, неопределенными данными является аргумент x, что означает, что он может принимать любое значение целочисленного типа. Поэтому, когда происходит ветвление по условию $x < 0$, символьная виртуальная машина зайдет в оба пути исполнения вместо одного

```
1 static int abs(int x) {
2     int y;
3     if (x < 0) {
4         y = -x;
5     } else {
6         y = x;
7     }
8     return y;
9 }
```

Рис. 1: Пример для демонстрации символьного исполнения.

конкретного, как в обычном исполнении программы. Чтобы этого достичь, движок создаст два разных *символьных состояния*, которые отражают соответствующие пути исполнения. Различие этих состояний состоит в добавлении условий $x < 0$ в одно и $x \geq 0$ в другое. Все условия на символьные переменные, хранящиеся в символьном состоянии, называются *условия пути*. Также символьное состояние содержит и ранее упомянутую символьную память. Во время исполнения инструкции $x = -y$, символьное состояние с условием пути $x < 0$ сохранит в ячейку символьной памяти y выражение $-x$. Аналогично, символьное состояние в другой ветке присвоит ячейке y в символьной памяти значение x. После исполнения программы получатся два состояния:

- $\Big\{$условие пути $\mapsto (x < 0), y \mapsto -x\Big\}$, с результатом $-x$;
- $\Big\{$условие пути $\mapsto (x \geq 0), y \mapsto x\Big\}$, с результатом x.

Для упрощения и ускорения работы символьного исполнения часто используются SMT-решатели, которые проверяют логические формулы на выполнимость. Они используются для проверки того, удовлетворяет ли хотя бы один набор конкретных входных данных для исполнения программы формуле условия пути. Если да, то движок исследует эту ветвь. В противном случае, анализ этого пути бессмысленен, т.к. недостижим. Кроме того, SMT-решатели могут находить конкретные значения, удовлетворяющие всем условиям, если такие решения вообще существуют. Более подробно о SMT-решателях описано в разделе 1.2.

1.2. SMT-решатели

В математической логике существует понятие выполнимости формулы, что означает возможность подстановки конкретных значений в переменные, чтобы всё выражение стало истинным. Эта концепция используется в *SAT-решателях* [10, 14], которые принимают формулу, состоящую из переменных и логических выражений, где каждая переменная может принимать значение истина или ложь. Такие формулы являются частью логики высказываний. Если формула выполнима, то решатель выдает SAT и значения переменных,

на которых формула истинна, что называется моделью. Если же формула не выполнима, то результатом является UNSAT. Если же не хватило ресурсов или времени для анализа, то UNKNOWN.

SMT-решатели [5, 16, 17] продолжают идею SAT-решателей, проверяя выполнимость формулы логики первого порядка, которая является более выразительной, чем логика высказываний. Они позволяют использовать в формуле функциональные, предикатные и константные символы, некоторые из которых могут быть частью теории. Теория состоит из заранее определенной сигнатуры и набора аксиом, которые определяют логику теории. Примеры теорий включают линейную целочисленную арифметику, линейную вещественную арифметику, неинтерпретированные функции и массивы. Из-за увеличения выразительности формулы задача проверки на выполнимость в общем случае становится неразрешимой. В связи с этим SMT-решатель выдает SAT вместе с моделью, если нашел ее, UNSAT, если доказал, что модели не существует, и UNKNOWN в противном случае.

1.3. Динамическое символьное исполнение

Динамическое символьное выполнение [8, 12, 13], является модификацией символьного исполнения, которая позволяет выполнить некоторые части кода конкретно, т.е. запустить исследуемую программу с конкретными данными. В дальнейшем под выполнением программы или ее части будем понимать *конкретное исполнение*.

Эта модификация символьного исполнения возникла как способ решения проблемы внешних методов, чей код недоступен или написан на другом языке программирования. Примеры таких методов включают функции, которые взаимодействуют с окружением. Благодаря возможности выполнения кода, динамическое символьное выполнение может вызывать внешние методы на определенных данных.

Существует еще одно преимущество динамического символьного исполнения. Символьное исполнение обычно работает, делая заранее некоторые предположения, которые делают возможным более эффективный анализ в определенных случаях. Иначе говоря, символьное исполнение строит мо-

дель программы и анализирует уже её. Однако, это может привести к ложноположительным и неправильным ответам. Эта проблема решается конкретным исполнением путем вызова метода, находящегося на тестирование.

Приведенный в данном дипломе пример алгоритма имеет следующую структуру. Во-первых, когда символьное исполнение достигает терминального оператора, например, оператора return или throw, оно запускает конкретное исполнение для проверки и получения фактического результата. Это позволяет сократить количество неправильных результатов, не потеряв охвата кода. Еще одна ситуация возникает, когда истекает время символьного исполнения или SMT-решатель не может найти модель для дальнейшего выполнения. В таких случаях остаются незаконченные символьные состояния, то есть символьные состояния, которые не достигли терминального оператора. В чисто символьном исполнении нет способа получить результаты выполнения функции. Тем не менее, с использованием конкретного исполнения можно обработать такие неполные символьные состояния. Можно использовать последнюю модель этого состояния и просто выполнить код, используя конкретные данные из этой модели. Это позволяет сохранить и использовать проделанную работу, увеличивая покрытие кода.

1.4. Существующие решения

Эта глава кратко обозревает некоторые существующие инструменты, основанные на динамическом символьном исполнении, и описывает их преимущества и недостатки.

- **JDart.** JDart [15] – это динамический символьный инструмент для анализа программ, написанных на языке программирования Java. JDart работает на уровне байт-кода JVM. Поэтому его можно использовать для анализа кода на языке Kotlin, но, конечно же, с некоторыми ограничениями. У этого инструмента есть две отличительные особенности. Во-первых, он заменяет фактические инструкции байт-кода символьными, что позволяет собирать условия пути и конкретно исполнять код одновременно. Кроме того, JDart использует несколько SMT-решателей для вывода моделей. Однако JDart не поддерживает строки и массивы с

динамическим размером.

- **ACTEve.** ACTEve [2] – это инструмент конколик-тестирования приложений для Android, который символьно отслеживает события от момента их генерации в рамках фреймворка до момента их обработки в приложении. ACTEve является одним из редких инструментов генерации тестов, поддерживающих события, что делает его уникальным, но он слишком медленный для обычных ситуаций. Также его нельзя применять для анализа программ на языке Kotlin.

- **Triton.** Triton [19] – это библиотека динамического бинарного анализа, что означает, что ее можно применять к любой программе, написанной на любом языке. Она отлично подходит для низкоуровневого анализа, который может быть применен для поиска уязвимостей. Не смотря на то, что Triton может анализировать программы на языках Java и Kotlin, интерпретировать результаты анализа обратно на уровень Java или Kotlin может быть сложным. Кроме того, Triton может начать анализировать дополнительный код, связанный с JVM, но не относящийся к программе, которую нужно тестировать, что может повлиять на производительность.

В целом, каждый из существующих инструментов имеет свои преимущества и недостатки, а выбор подходящего инструмента зависит от конкретных требований и особенностей анализируемой программы. Тем не менее, можно выделить общие проблемы:

- Поддержка Kotlin – хотя эти инструменты могут быть как-то применены для анализа программ на Kotlin, они не поддерживают Kotlin в полной мере и имеют много ограничений;

- Слишком много ограничений – представленные инструменты нельзя применять на реальных крупных проектах и системах, несмотря на то, что они имеют уникальные возможности, улучшающие анализ в определенных случаях.

1.5. UnitTestBot

UnitTestBot - это инструмент для автоматической генерации модульных тестов и точного анализа кода. Он создает готовые к использованию тестовые случаи для программ на Java и Kotlin с понятными входными данными и комментариями.

UnitTestBot использует символьное исполнение для анализа программ, что позволяет обнаруживать сложные и хитрые ошибки и дефекты. Его символьная виртуальная машина реализует множество продвинутых функциональностей, которые позволяют применять его для анализа реальных крупных систем и программ. Вот несколько из них:

- Поддержка базовых структур данных, таких как массивы, хэш-таблицы и множества, с использованием оптимизированных версий этих коллекций для символьного анализа;
- Мокирование внешних классов и функций позволяет сократить символьное исполнение и анализ классов и функций, находящихся за пределами тестируемого класса.

Эти и подобные особенности делают UnitTestBot одним из лучших инструментов для автоматической генерации тестов.

Более того UnitTestBot уже реализовал прототип подсистемы конкретного исполнения, однако он нуждался в серьезных доработках, имея ряд существенных ограничений, в особенности, на больших проектах.

Данный диплом расширяет функциональность UnitTestBot путем создания подсистемы конкретного исполнения на базе прототипа и интеграции ее в символьный движок, что позволяет реализовать подход динамического символьного исполнения. На рисунке 2 показана общая архитектура UnitTestBot, на которой отображены разработанные и модифицированные компоненты. Вот описание наиболее значимых компонентов для данной работы:

- **TestCaseGenerator** – это компонент, ответственный за запуск и контролирование процесса генерации тестов. Он выбирает метод для тестирования, инициализирует и использует символьное исполнение и также обрабатывает результаты генерации.

- **SymbolicEngine** реализует технику символьного исполнения. Он извлекает информацию о тестируемом методе, строит граф потока управления для этого метода и возвращает результаты символьного исполнения.

- **Fuzzer** реализует технику случайного тестирования [1], иными словами *фаззинг*. Этот компонент активно использует конкретное исполнение для получения результатов выполнения функции и сбора покрытия кода, так как это необходимо для данного подхода.

- **Minimizer** – это компонент, который уменьшает количество сгенерированных тестов без потери покрытия кода. Ему также необходима информация о покрытии.

- **ConcreteExecutor** – это входная точка для техники конкретного исполнения. Также этот компонент ответственен за общение и InstrumentedProcess.

- **InstrumentedProcess** реализует конкретное исполнение в отдельном процессе. Он может исполнять методы, используя конкретные данные, и собирать статистику, такую как покрытие инструкций байткода.

В целом, UnitTestBot – это мощный инструмент для анализа программ и генерации тестов. Он имеет обширную кодовую базу и мощную функциональность, что делает его одним из лучших инструментов для генерации тестов. Цель данного диплома заключается в улучшении качества генерации тестов UnitTestBot путем реализации виртуальной машины динамического символьного выполнения.

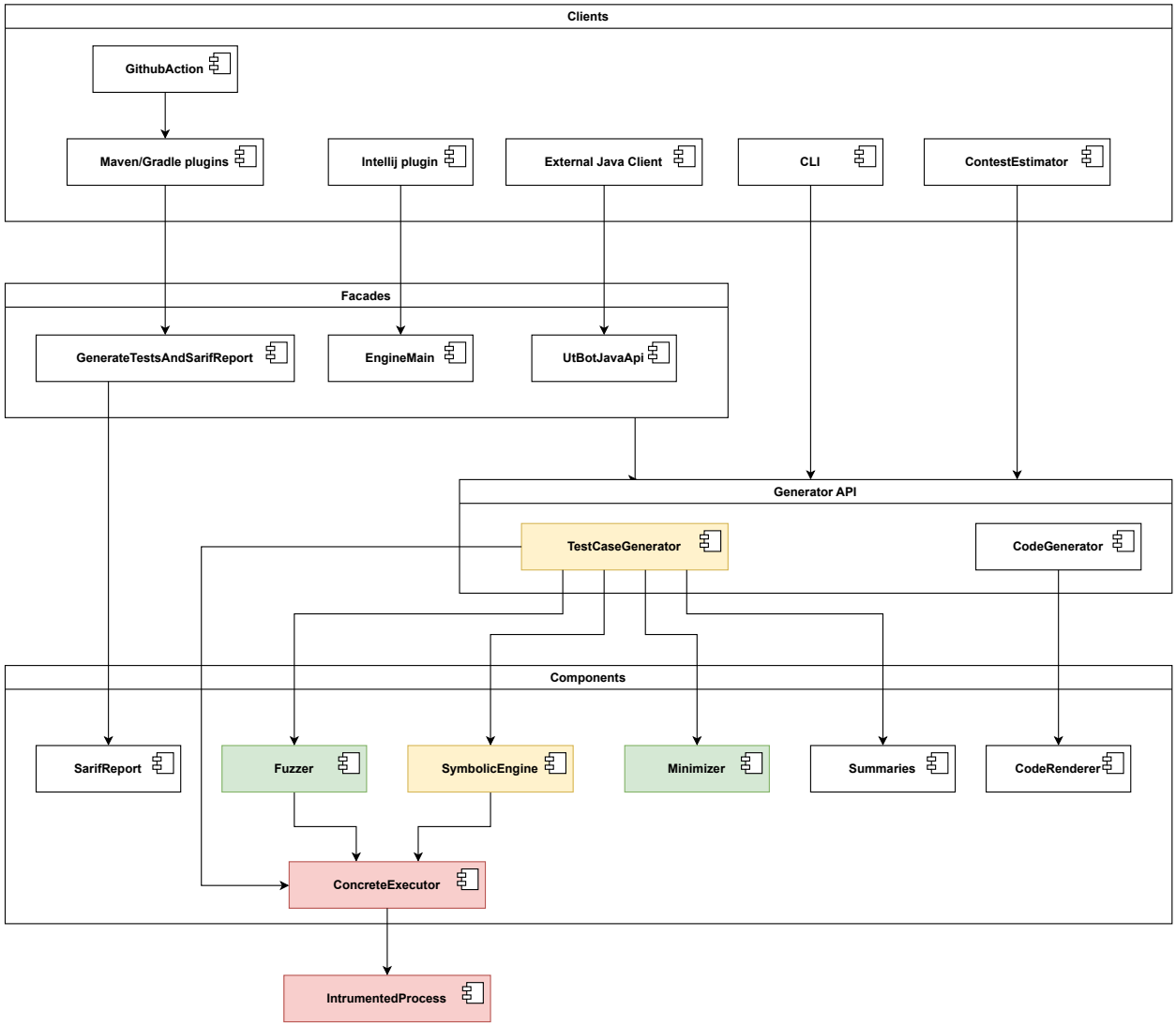


Рис. 2: Обзор общей архитектуры UnitTestBot.
Красные компоненты – компоненты подсистемы конкретного исполнения, разработанные в рамках прототипа и переделанные в контексте данного диплома.
Желтые компоненты – модифицированные, для интеграции в UnitTestBot.
Зеленые компоненты – разработаны благодаря подсистеме конкретного исполнения.

2. Описание реализации

В данной главе описываются особенности реализации виртуальной машины динамического символьного исполнения в проекте UnitTestBot[2]. Реализация была написана на языках программирования Kotlin и Java и доступна здесь[3]. Далее будут проанализированы наиболее значимые части, а именно:

- Общая архитектура подсистемы;
- Процесс запуска Конкретного Движка;
- Инструментация байт-кода - добавление дополнительных инструкций байт-кода для контроля исполнения и собирания статистики;
- Межпроцессное общение между конкретным и символьным движками;
- Интеграция в UnitTestBot;
- Процесс исполнения – как пользовательский код запускается в UnitTestBot;
- Поддержка механизма мокирования – подмена поведения внешних функций;
- Обнаружение недетерминированного поведения;
- Поддержка Kotlin.

UnitTestBot уже предпринимал попытку реализовать подсистему конкретного исполнения, однако она требовала больших доработок для полноценного использования. В дальнейшем будет описано какие части были взяты за основу, какие были доработаны, а какие были абсолютно изменены или написаны с нуля.

[2]UnitTestBot – инструмент автоматической генерации модульных тестов: https://www.utbot.org/

[3]UnitTestBot имеет открытый исходный код: https://github.com/UnitTestBot/UTBotJava

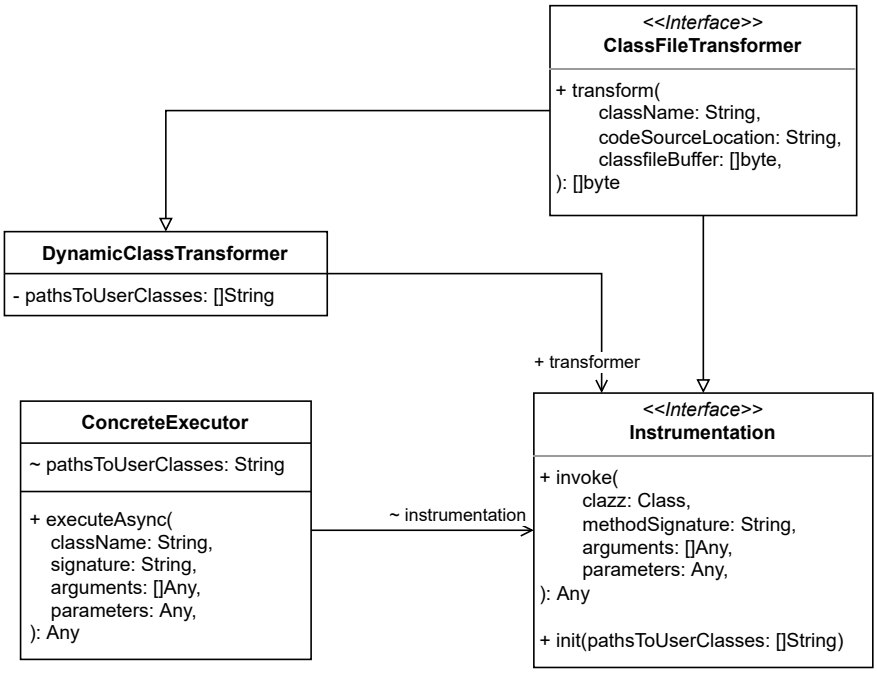


Рис. 3: Архитектура Конкретного Движка для UnitTestBot для Java и Kotlin.

2.1. Архитектура подсистемы

Главная идея заключается в использовании отдельного процесса для выполнения кода пользователя с целью обеспечения дополнительной безопасности. Такой процесс называется инструментированным процессом.

На рисунке 3 показана упрощенная версия диаграммы классов конкретного движка, части виртуальной машины динамического символьного исполнения, отвечающей за конкретное исполнения методов пользователя, и на рисунке 4 показаны диаграммы последовательностей, описывающие основные сценарии работы конкретного движка. Данная верхнеуровневая архитектура была уже разработана ранее и имеет множество плюсов, и поэтому взята за основу финального решения. Разработанная подсистема содержит четыре класса:

- ClassFileTransformer[4] – это интерфейс, предостовляемый стандартной библиотекой Java для инструментации байт-кода, подробнее будет описано в главе 2.3. Этот интерфейс предоставляет метод transform, который должен модифицировать исходный JVM байт-код.

[4]ClassFileTransformer документация: https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/ClassFileTransformer.html

- Instrumentation – это интерфейс, предоставляющий главную функциональность конкретного движка, и состоит из двух методов: init и invoke. Init инициализирует непосредственно объект инструментации. Invoke метод запускает определенный пользовательский метод с конкретными аргументами. Также, этот интерфейс наследует ClassFileTransformer, что означает, что наследники Instrumentation должны реализовывать логику, связанную с трансформацией байт-кода.

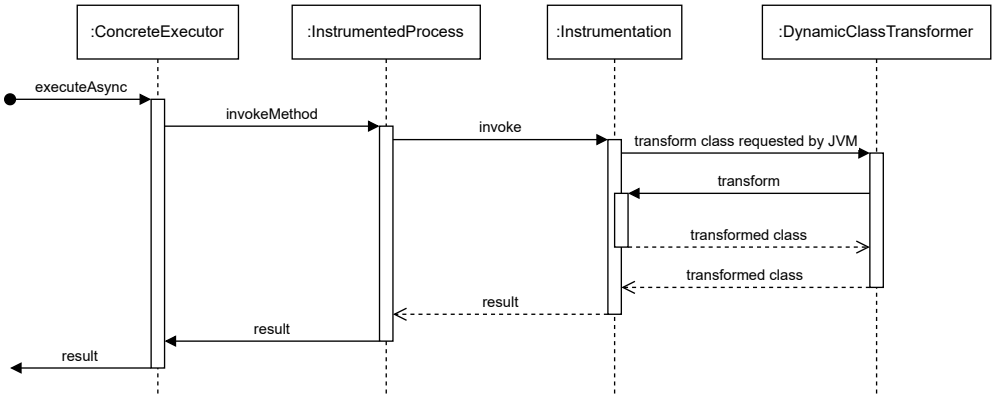
- DynamicClassTransformer – это класс, контролирующий какой загружаемый класс в JVM должен быть инструментирован, а какой нет. Если класс должен быть инструментирован, то используется заранее данный Instrumentation. Это решение основано он местоположение данного класса: класс должен быть трансформирован, если он находится в пользовательском пути до классов.

- ConcreteExecutor – это класс ответственный за создание и коммуникацию с процессом, где Instrumentation запускается, т.е. инструментированным процессом. Общение между процессами происходит за счет RD (Reactive Distributed communication framework[5]), который использует сокеты для передачи команд. Это будет описано в деталях в главе 2.4.

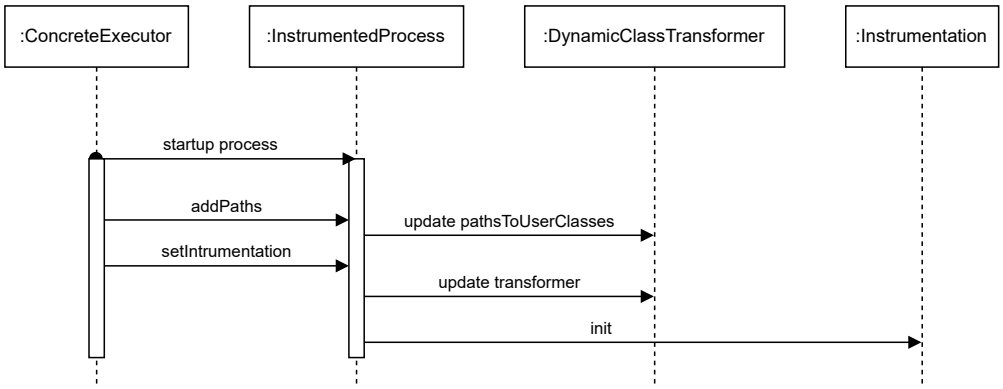
Цель данной архитектуры – разделить ответственность и логику на 3 части: запуск и необходимая инструментация для этого (Instrumentation), контроль процесса применения трансформации Instrumentation (DynamicClassTransformer), запуск и коммуникацию с инструментированным процессом и его контроль (ConcreteExecutor). Также ConcreteExecutor – это главная точка входа для инструментированного процесса, иными словами API для конкретного исполнения.

Сценарии использования (рис. 4) описывают как эти классы взаимодействуют друг с другом. Например (рис. 4a), когда вызывается метод executeAsync, ConcreteExecutor отправляет команду invokeMethod в инструментированный процесс, который затем вызывает метод invoke заранее установленного

[5]Reactive Distributed communication framework: https://github.com/JetBrains/rd



(a) Исполнение пользовательского кода



(b) Запуск

Рис. 4: Основные сценарии поведения Конкретного Движка.

Instrumentation класса. В ходе исполнения, когда JVM (Java Virtual Machine) встречает новый, еще не загруженный в память класс, он начинает процесс загрузки, запрашивая DynamicClassTransformer для трансформации класса, и, если это необходимо, Instrumentation класс используется для дальнейшей его трансформации.

2.2. Запуск конкретного движка

Запуск инструментированного процесса довольно сложен. Его следует запускать с помощью Java-агента.

Java-агент - это программа, которая выполняется в той же JVM, что и основное приложение, и может изменять поведение основного приложения или собирать информацию о его работе во время выполнения. Java-агент должен создать и добавить DynamicClassTransformer в конвейер загрузки новых

классов в JVM. Это необходимо для последующей инструментации байт-кода, которая будет подробно описана в главе 2.3.

После запуска JVM процесс должен установить соединение через RD и ожидать несколько команд, настраивающих пути к классам пользователя и используемую инструментацию. Это ясно описано в диаграмме последовательности 4b. Когда соединение и инструментация настроены, инструментированный процесс ожидает команд, в основном команды invokeMethod (рисунок 4a).

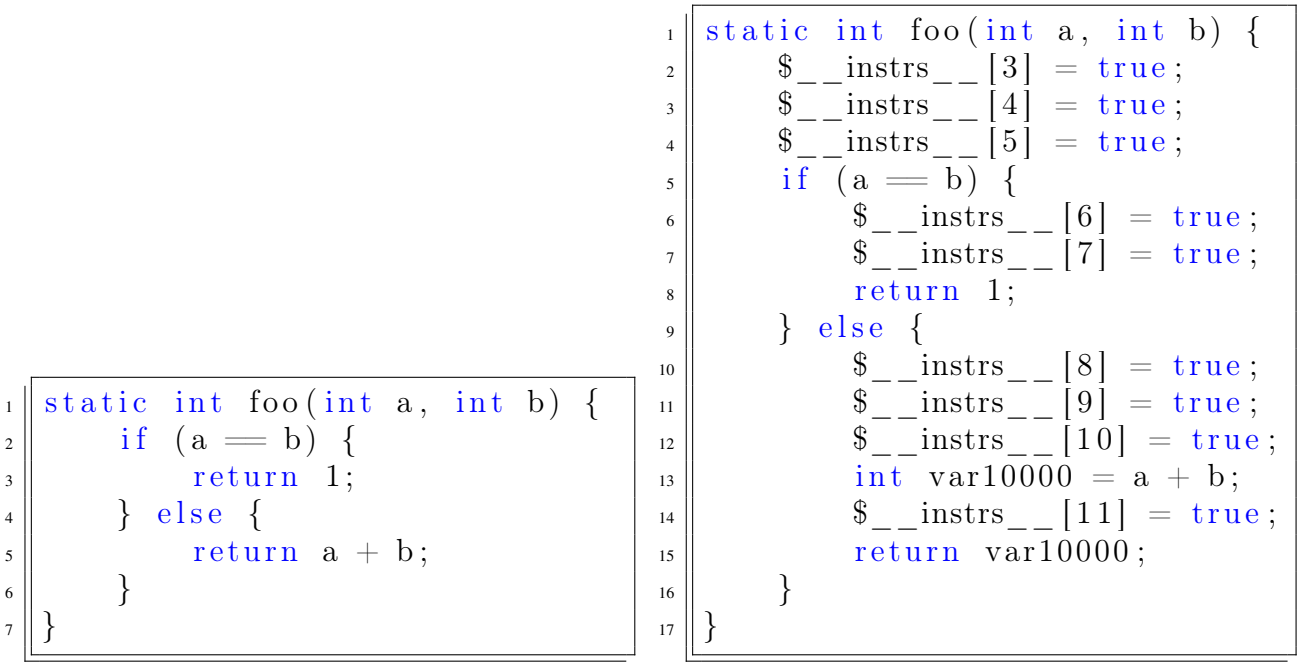
Что касается отладки, это важная функция, которая позволяет разработчикам находить сложные ошибки и исследовать непредвиденное поведение программ. Отладка отдельных процессов сложнее, но также необходима. Поэтому инструментированный процесс предоставляет способ отладки с помощью JDWP (Java Debug Wire Protocol[6]), используя дополнительные флаги при запуске. JDWP создает сервер, к которому могут подключаться клиенты для управления и наблюдения за состоянием сервера. Хотя это позволяет отлаживать инструментированный процесс, отладка инструментированного кода является сложной задачей и требует определенного опыта.

В ходе этой работы были обнаружены некоторые недочеты уже ранее реализованного запуска, связанные с модульной сборкой Java 9 версии и выше, исправлена обработка ситуаций с функцией assert, а также улучшен процесс откладки. Все это потребовало внесения дополнительных флагов при запуске процесса таких как "--add-opens", "--add-exports" и "-ea".

2.3. Инструментация байт-кода

Инструментация байт-кода - это процесс модификации байт-кода Java-класса во время выполнения, обычно для добавления или удаления инструкций байт-кода или сбора метрик во время выполнения программы. Инструментация байт-кода часто используется в Java-агентах и других инструментах, которые отслеживают и профилируют Java-приложения, в частности, UnitTestBot использует Java-агент, как описано выше. На рисунке 5 показан простой пример инструментации байт-кода, который реализует сбор информации о по-

[6]Java Debug Wire Protocol: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html



```
static int foo(int a, int b) {
    if (a == b) {
        return 1;
    } else {
        return a + b;
    }
}
```

(a) до инструментации

```
static int foo(int a, int b) {
    $__instrs__[3] = true;
    $__instrs__[4] = true;
    $__instrs__[5] = true;
    if (a == b) {
        $__instrs__[6] = true;
        $__instrs__[7] = true;
        return 1;
    } else {
        $__instrs__[8] = true;
        $__instrs__[9] = true;
        $__instrs__[10] = true;
        int var10000 = a + b;
        $__instrs__[11] = true;
        return var10000;
    }
}
```

(b) после инструментации

Рис. 5: Пример инструментации, которая модифицирует код так, чтобы собирать покрытые/исполненные инструкции в массив $__instrs__.

крытии инструкций байт-кода, перед каждой JVM инструкцией вставляя код, помечающий, что данная инструкция с уникальным номером была посещена.

Конкретный движок использует библиотеку ASM[7] для инструментирования байт-кода. ASM предоставляет низкоуровневый API для чтения, записи и трансформации байт-кода. Основой ASM является интерфейс ClassVisitor, который определяет методы для посещения различных элементов Java-класса, таких как поля, методы и инструкции. Разработчики могут реализовать собственный ClassVisitor для модификации байт-кода класса во время выполнения путем добавления или удаления инструкций, изменения существующих инструкций или добавления новых полей или методов.

В прототипной версии конкретного движка был разработан ClassVisitor под названием TraceListStrategy, который добавляет новые инструкции в код для сбора информации о посещенных инструкциях байт-кода во время выполнения. Когда TraceListStrategy встречает новую инструкцию байт-кода, он вычисляет уникальный номер, представляющий эту инструкцию, и вставляет некоторые дополнительные инструкции, сохраняющий данный номер в гло-

[7]Библиотека ASM: https://asm.ow2.io/

бальном хранилище. Однако вставленные инструкции не учитывали больших программ и долгих рекурсии и циклов, где количество выполненных инструкций было очень большое и вело к переполнению памяти хранилища. Поэтому вставляемые инструкции были переработаны на вызов метода visit класса RuntimeTraceStorage (рисунок 6), который сохраняет номер в глобальном хранилище, обрабатывая ситуации с переполнением памяти. После выполнения, это глобальное хранилище содержит все номера инструкций в порядке их посещения, что может позволить восстановить путь исполнения программы.

```
...
lconst_10
invokestatic RuntimeTraceStorage visit (J)V
aload_0
...
```

Рис. 6: Результат инструментации TraceListStrategy для инструкции aload_0 с уникальным номером 10.

Сам процесс инструментации происходит следующим образом. Когда JVM обнаруживает новый класс, который ранее не был загружен, JVM загружает его с использованием класса ClassLoader, а именно, находит местоположение файла этого класса, считывает его и вызывает уже добавленные экземпляры, наследующие интерфейс ClassFileTransformer, трансформируя считанный байт-код, в частности, используется DynamicClassTransformer для трансформации при помощи заранее определенного класса Instrumentation. А именно берет местоположение класса, и если оно оказывается в пользовательском пути до классов, то вызывается метод transform у Intrumentation.

В процессе использования данной подсистемы, было выяснено, что инструментация занимает значительное время, а также прерывание процесса инструментирования влечет за собой сбои в дальнейшей работе, связанной с классом, на котором инструментирование было прервано. Время исполнения пользовательского кода ограничено, т.к. достоверно неизвестно завершится ли выполнение данного кода или нет, поэтому по истечению выделенного времени выполнение прерывается. Но при этом не учитывалось время проведенное за инструментацией и загрузкой классов, и исполнение могло прерваться в момент инструментации. Таким образом периодический, особенно

на больших проектах, происходили сбои, приводящие к неправильной работе UnitTestBot в целом.

Было разработано и реализовано следующее улучшение. Создан специальный класс StopWatch для точного и многопоточного отслеживания времени исполнения программы за исключением времени потраченного на инструментирование классов. Многопоточность в реализации этого улучшения – важная часть, т.к. инструментация, загрузка классов, исполнение пользовательского кода и контроль времени этого исполнения это многопоточный процесс. Перед запуском пользовательского метода, StopWatch начинает отсчитывать время, а при вызове метода transform у класса DynamicClassTransformer, отсчет времени приостанавливался. При этом периодически проверялось время, пройденное с начала выполнения, если оно закончилось и при этом StopWatch не остановлен, то можно завершать исполнение кода, иначе ждать оставшееся время. Такое решение значительно сократило количество сбоев генерации тестов, что позволило применять данную подсистему на практике и для очень больших проектов. Однако, такое решение не гарантирует полного устранения данной проблемы, т.к. чтобы ее решить, необходимо более низкоуровневые модификации, а именно на уровне JVM.

Обобщая, прототипная реализация процесса инструментации не учитывала крайних ситуации, связанных с большим объемом программ, что приводило к неприменимости подсистемы конкретного исполнения на реальных проектах. Однако, после проделанных улучшений, учитывающих время инструментации и переполнение памяти хранилища покрытых инструкции, данная подсистема стала достаточно практичной и эффективной для ее применения на реальных и больших проектах.

2.4. Межпроцессное общение

Межпроцессное взаимодействие является одной из важных частей реализации. Оно позволяет обмениваться данными и координировать работу между процессами символьного и конкретного движков. В случае реализации конкретного движка механизм общения достигается с помощью Reactive

Distributed (RD) communication framework[8], который использует сокеты для связи между процессами.

RD является легковесной и реактивной библиотекой, основанной на передаче сообщений, который позволяет асинхронное и неблокирующее взаимодействие между процессами, что делает его подходящим для высокопроизводительных систем. RD предоставляет набор абстракций и API, которые упрощают разработку распределенных приложений. Основной абстракцией, предоставляемой RD, является канал, который представляет собой двунаправленный поток сообщений между двумя процессами. Каналы могут быть созданы и управляться с использованием API RD, а сообщения могут быть отправлены и приняты асинхронно с помощью методов отправки и получения, предоставляемых библиотекой.

Для обеспечения межпроцессного общения в реализации подсистемы конкретного исполнения класс ConcreteExecutor отвечает за создание и управление каналами RD между основным приложением и инструментированными процессами. При запуске инструментированного процесса ConcreteExecutor настраивает канал RD для обеспечения связи между двумя процессами. Кроме того, RD позволяет определить модель команд и сообщений заранее, что упрощает разработку протокола общения.

В реализации конкретного движка между основным приложением и инструментированным процессом может быть отправлено несколько команд. Эти команды позволяют выполнять различные типы взаимодействия, такие как запрос результатов выполнения пользовательской функции или отправка инструментированному процессу конфигурационной информации:

- AddPaths - устанавливает пути, по которым инструментированный процесс должен искать классы пользователя.
- Warmup - принудительно загружает классы из пути классов пользователя в JVM для инструментирования.
- SetInstrumentation - устанавливает фактический класс Instrumentation и инициализирует его.

[8]Реактивный распределенный коммуникационный фреймворк: https://github.com/JetBrains/rd

- InvokeMethodCommand - запрашивает выполнение метода с помощью предоставленной инструментации.

Данная часть реализации подсистемы конкретного исполнения была спроектирована и реализована командно, где я играл важную роль в проектировании, контроле и проверке реализации межпроцессного взаимодействия.

2.5. Интеграция в UnitTestBot

Эта глава описывает, как UnitTestBot использует подсистему конкретного исполнения. Во-первых, генерация UnitTestBot состоит из двух частей:

- Фаззинг - техника, которая генерирует случайные входные данные без глубокого анализа кода.

- Динамическое символьное исполнение - техника, которая была подробно описана выше.

Обе эти части активно используют конкретное исполнение в UnitTestBot.

Чистый фаззинг не может определить результат выполнения без запуска метода. Следовательно, код пользователя должен быть выполнен, т.к. результат выполнения необходим для корректной генерации тестов. Кроме того, фаззинг использует метрику, такую как покрытие кода, для улучшения своей генерации. Поэтому подсистема конкретного исполнения является неотъемлемой частью фаззинга.

Существует множество подходов к сочетанию конкретного и символического выполнения. Символьный движок UnitTestBot использует конкретный движок в двух случаях. Первый случай - это проверка правильности вычисленного результата символьного исполнения, то есть, когда символьный движок достигает конечного оператора (например, оператора return), он использует SMT-решатель для получения конкретных входных данных и результата. Однако результат может потребоваться скорректировать, поскольку символьное исполнение делает некоторые предположения заранее, которые могут привести к неправильной работе, т.е. анализирует лишь какую-то модель программы, а не ее саму. Таким образом, вызов метода с использованием подсистемы

конкретного исполнения решает эту проблему, предоставляя действительный результат выполнения программы на конкретных данных. Другой случай связан с неудачным символьным исполнением из-за ограничений по времени и вычислительным ресурсам. В такой ситуации исходный символьный движок не мог получить результаты и отбрасывал такие состояния. Тем не менее, это является пустой тратой проделанной работы. Конкретный движок может выполнить метод пользователя с использованием таких незавершенных состояний, взяв последнюю модель входных данных, которую получилось посчитать, и запустив её конкретным исполнением, тем самым получая ожидаемый результат выполнения метода и генерируя тест. Такая стратегия была разработана в первой версии динамического символьного исполнения UnitTestBot и взята без изменений.

Также в ходе данной работы были попробованы другие подходы комбинирования различных стратегий, однако они не принесли значительных результатов и зачастую были хуже, чем существующее решение. Одной из таких стратегий была попытка совместить фаззинг и символьное исполнение: изначально брались значения сгенерированные фаззингом и пытались служить приоритетом для символьного исполнения, т.е. символьные состояния, соответствующие сгенерированным значениям, анализировались в первую очередь. Такой подход должен был улучшить качество тестов, однако такого не произошло на практике.

2.6. Процесс исполнения

UtExecutionInstrumentation - это класс, реализующий интерфейс Instrumentation
UnitTestBot использует этот класс в качестве основного наследника Instrumentation
для конкретного исполнения. Целью UtExecutionInstrumentation является вызов кода пользователя со собором покрытых инструкций и с некоторыми дополнительными функциями, которые будут описаны в последующих главах.

UtExecutionInstrumentation должен быть адаптирован для моделей из символьного движка (то есть символьных моделей). Он принимает входные данные в виде некоторых классов, используемых в символьном исполнении,

например, структур, описывающих, как создать экземпляр с помощью последовательности методов, или просто литералов. UtExecutionInstrumentation должен создавать из заданных символьных моделей фактические входные данные, объекты JVM, которые могут быть использованы для вызова пользовательских методов в качестве аргументов. Кроме того, результат вызова должен быть снова переведен в символьную модель.

Относительно выполнения кода пользователя, UtExecutionInstrumentation создает конвейер, состоящий из фаз, а именно:

1. **Фаза создания значений** (ValueConstructionPhase) отвечает за создание фактических входных значений из символьных моделей. Она создает экземпляры для параметров и статических полей. UtExecutionInstrumentation использует механизмы reflection для создания необходимых экземпляров.

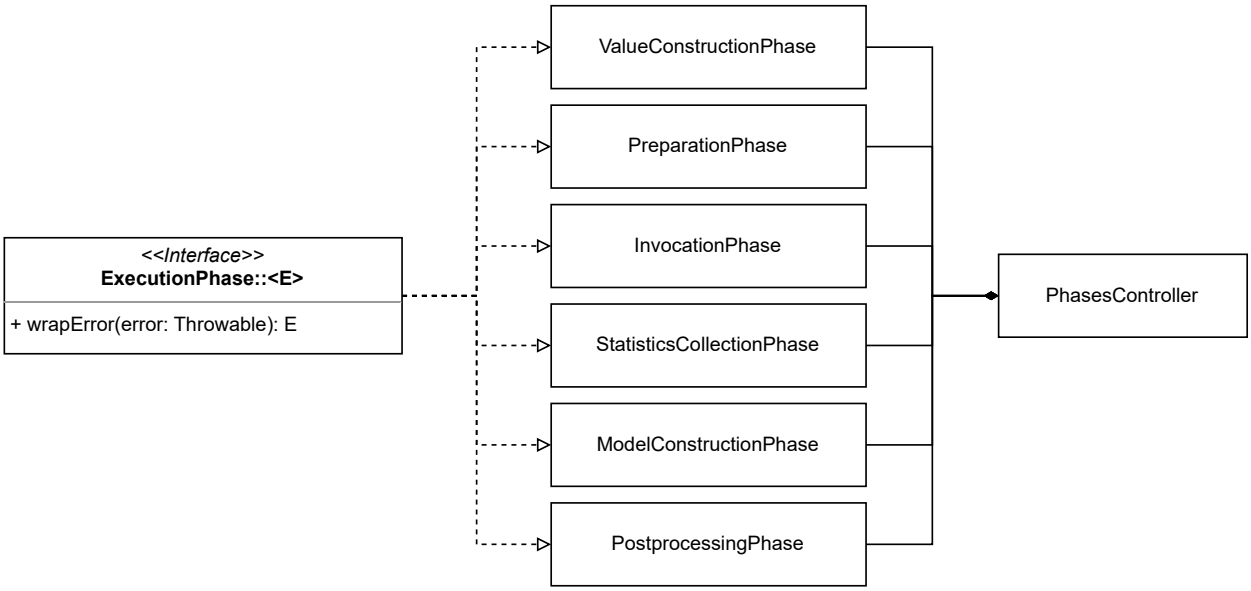
2. **Фаза подготовки** (PreparationPhase) устанавливает статические поля уже созданными значениями и очищает все хранилища, оставленные после предыдущих вызовов и фаз, например, глобальное хранилище для покрытых инструкций.

3. **Фаза вызова** (InvocationPhase) выполняет указанный метод пользователя.

4. **Фаза сбора статистики** (StatisticsCollectionPhase) собирает все необходимые метрики после выполнения, такие как покрытые инструкций.

5. **Фаза создания моделей** (ModelConstructionPhase) создает символьные модели из результатов вызова и статических полей.

6. **Фаза постобработки** (PostprocessingPhase) сбрасывает статические поля и некоторые кэши.



(a) Диаграмма классов.

```
PhasesController().computeConcreteExecutionResult {
    try {
        val params = executePhaseInTimeout(valueConstructionPhase) {
            constructParameters(stateBefore)
        }

        preparationPhase.start { resetTrace() }

        val concreteResult = executePhaseInTimeout(invocationPhase) {
            invoke(clazz, methodSignature, params)
        }

        val coverage = statisticsCollectionPhase.start { getCoverage(clazz) }

        val executionResult = executePhaseInTimeout(modelConstructionPhase) {
            convertToExecutionResult(concreteResult, returnClassId)
        }

        UtConcreteExecutionResult(executionResult, coverage)
    } finally {
        postprocessingPhase.start { resetStaticFields() }
    }
}
```

(b) Упрощенный пример использования фаз исполнения. Функция start запускает фазу с обработкой исключений. Функция executePhaseInTimeout запускает исполнение фазы с ограничением времени.

Рис. 7: Диаграмма классов и пример использования фаз исполнения.

Диаграмма классов (рис. 7a) показывает архитектуру разделения на фазы. ExecutionPhase – это интерфейс с методом wrapError, который занимается тем, что оборачивает исключение во время исполнения фазы в исключение специального типа, где для каждой фазы свой тип. PhasesController содержит все фазы и занимается их инициализацией и передачей информации. На рисунке 7b показан упрощенный пример использования реализованной архитектуры. Для удобной работы были реализованы функции-расширения внутри класса PhasesController: start и executePhaseInTimeout. Функция start занимается выполнением кода с обработкой исключений методом wrapError. Функция executePhaseInTimeout делает то же самое, что и start, только с ограничением по времени, здесь активно используется StopWatch, который был описан в главе 2.3.

Такое архитектурное и логическое разделение на фазы, разработанное в ходе данного диплома, позволяет более точно и гибко контролировать процесс вызова. Например, UtExecutionInstrumentation устанавливает временные ограничения для фаз создания значений, вызова и создания моделей. Кроме того, если происходит ошибка в одной из этих фаз, можно определить, в какой именно фазе она произошла. Все это позволяет быстрее находить ошибки или недочеты реализации и уменьшать количество кода, а также это позволяет делать обработку непредвиденных ситуации более гибкой и адаптивной.

2.7. Поддержка мокирования

Мокирование - это техника, используемая в разработке программного обеспечения для имитации поведения реальных объектов или систем. Обычно это делается для тестирования поведения кода в изоляции, без зависимости от фактической реализации этих объектов или систем. Поддержка механизмов мокирования в UnitTestBot означает, что анализ внешних зависимостей может быть уменьшен, и, в конечном итоге, это может повысить качество создаваемых тестов. Вот почему поддержка мокирования в подсистеме конкретного исполнения является неотъемлемой и важной частью реализации.

```
SomeClass mock = mock(SomeClass.class);
when(mock.someMethod()).thenReturn(42);
methodUnderTest(mock);
```

Рис. 8: Пример мокирования объекта типа SomeClass с использованием библиотеки Mockito в сгенерированных тестах.

В UnitTestBot поддерживаются три аспекта механизма мокирования, взятые из популярной и широко используемой библиотеки Mockito[9]:

- мокирование объектов,
- мокирование статических функций,
- мокирование создания объектов.

Ниже описано, как они поддерживаются на уровне реализации конкретного движка.

2.7.1 Мокирование объектов

Мокирование объектов создает конкретные экземпляры определенного типа, такого как класс, интерфейс или абстрактный класс. Поведение методов этих объектов может быть изменено. Таким образом, можно предположить, что описание того, как мокировать экземпляр, сводится к тому, что должны возвращать его методы. Поэтому, когда символьное исполнение хочет сделать мок экземпляра, он создает специальную символьную модель, содержащую описание ожидаемых результатов его методов. На рисунке 8 показан пример такого типа мокирования. Естественно, такая модель мокирования не полная, хотя и очень распространенная на практике. Полное же описание всех возможных сценариев поведения невозможно, т.к. их бесконечное количество, а именно столько, сколько и функций можно создать.

Для реализации такого типа мокирования, была взята прототипная версия реализации конкретного исполнения, где конкретный движок использует библиотеку Mockito, чтобы избежать дополнительной работы, связанной с

[9]Библиотека Mockito: https://site.mockito.org/

```
MockedStatic mock = Mockito.mockStatic(SomeClass.class);
when(someInstance::someStaticMethod).thenReturn(42);
methodUnderTest();
```

Рис. 9: Пример мокирования статической функции класса SomeClass с использованием Mockito в сгенерированных тестах.

генерацией новых классов во время выполнения для мокирования экземпляров с типом интерфейса или абстрактного класса. В ходе дипломной работы было доработано поведение, когда мокирование было частичным, а именно, когда символьный движок ошибся или недоанализировал путь исполнения, он может создать незавершенную модель, где количество вызовов методов такого экземпляра будет больше, чем описанных значений в модели. В таких случаях доработка заключалось в вызове реального метода. Создание мок-экземпляров происходит в фазе создания значений.

2.7.2 Мокирование статических функций

Мокирование статических функций более сложное, чем мокирование экземпляров, поскольку они не привязаны к конкретному объекту. Пример мокирования статической функции показан на рисунке 9. Mockito может мокировать статические функции, но правильная его настройка может быть сложной, а также является не эффективным и неконтролируемым для подсистемы конкретного исполнения. Более того, потенциально инструментации Mockito могут конфликтовать с другими инструментациями. UtExecutionInstrumentation выполняет дополнительные преобразования байт-кода для поддержки мокирования статических функций в UnitTestBot.

Символьное исполнение генерирует дополнительные инструкции (т.е. специальные структуры данных), чтобы отразить мокирование статических функций, которые затем передаются в конкретный движок. Эти инструкции содержат ожидаемые результаты выполнения статических функций в виде символьных моделей. Основная идея заключается в добавлении новых инструкций байт-кода в начало статического метода, которые проверяют, установлена ли определенная переменная, представляющая необходимость в мокировании, в значение true. Если она установлена в true, то метод должен воз-

```
Mockito.mockConstruction(SomeClass.class, (mock, context) -> {
    when(mock.someMethod()).thenReturn(42);
});
methodUnderTest();
```

Рис. 10: Пример мокирования создания объекта типа SomeClass с использованием Mockito в сгенерированных тестах.

вращать предварительно созданное значение; в противном случае, он должен продолжить выполнение функции.

Во время фазы создания значений UtExecutionInstrumentation обнаруживает очередную инструкцию, представляющую мокирование статической функции, устанавливает соответствующую переменную в значение true, создает значения из переданных символических моделей и добавляет их в глобальное хранилище моков. Эти действия гарантируют, что при вызове статической функции во время выполнения теста она возвращает ожидаемый результат.

Такое решение было разработано в прототипе конкретного исполнения UnitTestBot и взято практически без изменений.

2.7.3 Мокирование создания объектов

Мокирование создания объектов является наиболее сложным аспектом поддержки мокирования и было разработано с нуля, т.к. в прототипе подсистемы не поддерживалось. Оно заменяет объекты, создаваемые во время исполнения тестируемого метода, на мок-объекты. На рисунке 10 приведен пример мокирования создания объекта. Однако сложность этой функции заключается в том, что создание объекта состоит из нескольких инструкций байт-кода, включая выделение памяти и инициализацию объекта, которые могут находиться в разных местах в коде.

Для решения этой проблемы можно предположить, что анализируемый код написан на языках Java и Kotlin. Инструкции байт-кода для создания объектов обычно следуют предсказуемому шаблону (см. рисунок 11).

Первая инструкция - это new, которая выделяет память для объекта, за которой следуют инструкции для параметров конструктора, которые могут

```
new
...
invokespecial SomeClass <init> (LSomeClass;)V
```

Рис. 11: Ожидаемый шаблон создания объекта.

включать создание других экземпляров (рисунок 12). После всех этих инструкций следует вызов конструктора. Состояние стека JVM[10] должно быть таким же, как до этих инструкций, за исключением результирующего объекта на вершине стека.

Для обнаружения этого шаблона и правильной обработки вложенных вызовов конструктора был реализован стек для инструкций new. Когда инструментация встречает инструкцию new, она добавляет эту инструкцию в стек, а затем, когда встречает вызов конструктора, извлекает инструкцию new из верхней части стека, что говорит о том, что обнаружен нужный шаблон и его можно преобразовать по необходимости.

UtExecutionInstrumentation преобразует код, обнаруживая описанные шаблоны, и затем окружает эти фрагменты кода проверкой переменной, представляющей необходимость в мокировании создания того или иного объекта. При необходимости он помещает уже созданный мок-объект в стек или выполняет инструкции создания в противном случае. В результате, при необходимости, все инструкции между new и вызовом конструктора заменяются извлечением мок-объекта.

Конечно, в реальном коде могут встречаться различные шаблоны, которые нарушают сделанные предположения. Например, некоторые оптимизации компилятора могут преобразовывать байт-код, чтобы сделать его более эффективным, нарушая инвариант со стеком. Поэтому реализация должна уточняться для правильной обработки различных шаблонов.

Установка переменных, необходимых для мокирования создания экземпляров, и создание мок-объектов для замены происходят в фазе построения значений, аналогично мокированию статических функций.

[10]Документация по стеку JVM: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5.2

```
new SomeClass(new SomeClass());
```

(a) Java код

```
new
new
invokespecial SomeClass <init> ()V
invokespecial SomeClass <init> (LSomeClass;)V
```

(b) Сгенерированный байт-код

Рис. 12: Пример сгенерированного байт-кода после компиляции Java кода, который вызывает конструкторы один в другом.

2.8. Обработка недетерминированного поведения

Во время символьного исполнения в UnitTestBot может быть обнаружено недетерминированное поведение, например, при использовании методов класса "java.util.Random" для генерации случайных целых чисел. Хотя обычно символьный движок генерирует инструкцию для мокирования этого метода для детерминирования его поведения, это место может быть пропущено и не достигнуто из-за ограничений времени и вычислительных ресурсов или же из-за неточности анализа. Более того, чистый фаззинг также не может обнаружить такие ситуации. В результате UnitTestBot может создать мигающий тест, то есть тест, который дает разные и несогласованные результаты без изменений в тестируемом коде. Другими словами, это тест, который иногда проходит и иногда не проходит без видимой причины. Мигающие тесты являются распространенной проблемой в разработке и могут привести к замедлению и неэффективности тестирования.

Подсистема конкретного исполнения решает эту проблему[11], и ниже описано как. Во-первых, можно предположить, что конкретный движок заранее знает методы и объекты с недетерминированным поведением, то есть у него есть список полных имен таких методов и объектов. Поэтому была разработана инструментация, которая добавляет инструкции байт-кода, сохраняющие результаты выполнения таких методов в NonDeterministicResultsStorage. На рисунке 13 показан пример результата этой инструментации.

[11]Реализация обнаружения недетерминированного поведения доступна здесь: https://github.com/SBOne-Kenobi/UTBotJava/tree/sbone/non-deterministic-detection

```
...
dup
invokestatic NonDeterministicResultsStorage saveInstance (LObject;)V
invokevirtual java.util.Random nextInt ()I
dup
ldc "java.util.Random nextInt()I"
invokestatic NonDeterministicResultsStorage storeCall (ILString;)V
...
```

Рис. 13: Пример результата инструментации для метода "Random.nextInt". Сначала, необходимо сохранить объект, который будет использован для вызова недетерминированного метода, а затем записать результат исполнения этой функции в хранилище.

Перед вызовом недетерминированного метода, верхняя часть стека JVM состоит из экземпляра класса Random и параметров для вызова метода nextInt. В данном случае параметров нет. Для дальнейшего мокирования требуется сохранить экземпляр и параметры перед вызовом метода. После сохранения всей информации в NonDeterministicResultsStorage, вызывается метод, и после его вызова результат вызова также сохраняется в хранилище. Во время всех манипуляций необходимо внимательно следить за согласованностью стека JVM, поскольку любая ошибка может привести к сбою всей программы.

После выполнения тестируемого метода в NonDeterministicResultsStorage содержатся результаты выполнения всех недетерминированных методов, которые были выполнены. Из этой информации во время фазы построения моделей конкретный движок может расширить инструкции для мокирования и передать их обратно, позволяя генерировать правильные тесты с предсказуемым поведением.

Таким образом обнаружение недетерминированного поведения, разработанное в ходе данного диплома, улучшило надежность генерируемых тестов, уменьшая количество мигающих тестов. На рисунке 14 приведен пример результатов реализованной функциональности. Подробнее о результатах будет описано дальше.

2.9. Поддержка Kotlin

Kotlin – это язык программирования, основанный на JVM, что означает, что код, написанный на этом языке, преобразуется в инструкции байт-кода. Поэтому поддержка Kotlin относительно проста, но требует некоторой дополнительной работы, связанной с деталями реализации инструментаций.

Во-первых, поддержка мокирования создания экземпляров, описанная выше, требует некоторых предположений, связанных с шаблонами байт-кода. Необходимо убедиться, что Kotlin в основном генерирует соответствующий байт-код. Во-вторых, нужно было исключить любые преобразования байт-кода всех внутренних классов Kotlin и синтетически генерируемых методов в классах пользователя, которые Kotlin часто использует. Это слишком сложно, потому что нет открытой документации по этим случаям, что означает, что это должно быть исследовано путем изучения фактических примеров байт-кода, создаваемого Kotlin, в реальных проектах.

3. Тестирование и результаты

Эта глава описывает подход к тестированию реализованной подсистемы динамического символьного исполнения для языков программирования Java и Kotlin. Затем она представляет сравнение двух подходов: символьного исполнения и динамического символьного выполнения, в рамках UnitTestBot, что позволит оценить эффективность реализованной подсистемы. Более того, она содержит и другие результаты и возможные улучшения разработанной подсистемы и генерации тестов UnitTestBot в целом.

3.1. Тестирование корректности подсистемы

Реализованы два типа тестирования корректности. Первый тип тестирования заключается в проверке инструментированного процесса как системы инструментирования и сбора статистики. Этот тип тестирования был реализован в два этапа:

1. Реализация специального загрузчика классов (ClassLoader), который загружает и инструментирует только указанные классы.

2. Реализация инструментации, которая помогает убедиться, что класс правильно инструментирован. Эта инструментация аналогична инструментации для мокирования.

Все тесты первого типа выполняют следующие действия:

1. Выбор классов для инструментирования с помощью реализованного загрузчика классов.

2. Передача ожидаемых данных в качестве входных параметров для инструментации.

3. Запуск тестируемых методов и проверка фактических результатов.

Такой тип тестов помогает убедиться, что классы были правильно инструментированы.

Второй тип тестирования корректности – это тестирование системы динамического символьного исполнения. Оно основано на уже реализованных компонентах для тестирования символьного движка с использованием различных тестовых случаев, начиная от функций с одним оператором if и до сложных алгоритмов с использованием структур данных. Сначала запускаются и настраиваются символьный и конкретный движки. Затем система запускает динамическое символьное исполнение, получает результаты в качестве входных параметров, моков, результатов конкретного выполнения и покрытия инструкций, и проверяет, что полученная информация соответствует ожидаемой. Например, тест может проверить, что динамическое символьное выполнение покрывает не менее 90% инструкций метода.

3.2. Оценка эффективности подсистемы

Здесь описаны подход и результаты сравнения UnitTestBot с конкретным и без конкретного исполнения. Что позволяет говорить об эффективности реализованной подсистемы.

3.2.1 Измерения

С использованием UnitTestBot на реальных проектах было проведено сравнение символьного и динамического символьного исполнений. Конфигурация генерации была следующей: время анализа каждого класса - 60 секунд, тесты на ошибки должны проходить, без мокирования, без фаззинга и с предварительным прогревом (команда Warmup) перед исполнением в случае динамического символьного исполнения.

Использовался плагин UnitTestBot для IntelliJ IDEA. Сначала проект открывался в IntelliJ IDEA, а затем запускалась генерация тестов с помощью плагина. По окончании генерации полученные тесты выполняются с использованием инструмента Jacoco для сбора покрытия. Количество ошибок – это количество тестов, которые работают неправильно или являются мигающими.

Ниже приведено описание проектов, выбранных для сравнения, которые охватывают большинство конструкций языка Java:

- **Guava**[12] – набор основных библиотек на языке Java от Google, включающий новые типы коллекций (например, multimap и multiset), библиотеку для работы с графами и множество других функциональностей. Он широко используется в большинстве проектов на языке Java внутри Google и многими другими компаниями.

  - **com.google.common.math**. Этот пакет содержит различные математические утилиты и инструменты, такие как расчет статистик и линейные трансформации.
  - **com.google.common.graph**. Это библиотека для моделирования данных, структурированных в виде графа, то есть сущностей и связей между ними.

- **Fastjson**[13] – это библиотека на языке Java, которая может преобразовывать объекты Java в их представление в формате JSON. Она также может преобразовывать строку JSON в эквивалентный объект JVM.

  - **com.alibaba.fastjson**. Это основной пакет Fastjson, содержащий основные классы-представления JSON структуры.

Тестирование проводилось на ноутбуке DELL G15 5520 с операционной системой Windows 11 Home, процессором 12-го поколения Intel(R) Core(TM) i7-12700H и 16 ГБ оперативной памяти.

3.2.2 Результаты

Результаты измерений приведены в таблице 1. Получившиеся измерения показывают, что количество неправильных тестов уменьшилось. Это особенно заметно в математическом пакете guava. Для других пакетов снижение числа неправильных тестов менее значительно. Однако это можно объяснить ограничениями в UnitTestBot на сложность и вложенность структур. И т.к. остальные пакеты используют много вложенных структур и классов, UnitTestBot обрезает структуры, теряя точность и корректность.

[12]Guava – набор основных библиотек на языке Java от Google. https://github.com/google/guava
[13]Fastjson – это библиотека на языке Java для работы с JSON. https://github.com/alibaba/fastjson

Название пакета	% покрытия строк	% покрытия ветвей	% неправильных тестов
Символьное исполнение			
com.google.common.math	54.65	52.72	5.80
com.google.common.graph	29.36	21.35	6.05
com.alibaba.fastjson	46.59	33.40	15.87
Динамическое символьное исполнение			
com.google.common.math	56.52	53.90	0.59
com.google.common.graph	31.94	24.36	5.26
com.alibaba.fastjson	45.60	31.85	12.23

Таблица 1: Результаты запуска UnitTestBot на реальных проектах.
60 секунд на класс. Без использования мокирования.

Другой, но очень значимый результат – это покрытие. Генерация тестов сохраняет примерно ту же степень покрытия кода. Более того, в некоторых случаях оно увеличивается, например, в пакетах guava. Таким образом, UnitTestBot сохраняет свое качество и эффективность.

В целом, результаты соответствуют ожиданиям, согласно выбранной стратегии комбинирования. А значит реализованное динамическое символьное исполнение позволяет снизить количество ложных срабатываний, при этом сохраняя покрытие кода. Поэтому его можно применять на практике для повышения качества генерации тестов, не теряя эффективности.

3.3. Расширения UnitTestBot

Разработанная подсистема конкретного исполнения позволила разработать и реализовать новые подсистемы в UnitTestBot. Эти компоненты показаны на рис. 2. И в этой главе они кратко описываются.

3.3.1 Минимизация

Минимизация тестов [3, 20] – это подход к сокращению количества тестов без ухудшения качества, например, покрытия кода. Это позволяет сократить время выполнения тестов, что является важным для процесса непрерывной интеграции и доставки (CI/CD).

Поскольку конкретное исполнение может предоставлять информацию о покрытии кода, UnitTestBot может использовать ее для применения любого алгоритма минимизации тестов. Это расширяет возможности продукта и делает его более полезным для разработчиков.

Минимизация в UnitTestBot разделяет сгенерированные тесты на разные группы по типам, например, такие как успешные тесты, тесты на исключения, тесты с истечением времени. Затем тесты в каждой группе минимизируются независимо жадным алгоритмом: тесты приоритезируются и дальше пытаются удаляться в этом порядке, если при удалении теста покрытие меняется, это означает, что его нельзя удалять, и тогда он остается.

Хотя минимизацию тестов можно реализовать, используя только символьное исполнение, взяв пути, пройденные им, все еще существует проблема ложных срабатываний, которые могут привести к потере тестов и покрытия кода. Поэтому виртуальная машина динамического символьного исполнения является важной и существенной для минимизации.

3.3.2 Фаззинг

Фаззинг [1] (тестирование методом случайных данных или тестирование методом "обезьяны") - это техника тестирования программного обеспечения, которая заключается в подаче случайных или некорректных входных данных программе с целью выявления уязвимостей или ошибок, которые могут вызвать сбой программы, непредвиденное поведение или компрометацию безопасности системы.

Фаззинг направлен на проверку того, насколько хорошо программа обрабатывает непредвиденный или некорректный ввод. Путем отправки различных типов входных данных, таких как случайные строки, числа или файлы, инструменты для фаззинга могут выявить потенциальные переполнения буфера, утечки памяти и другие уязвимости безопасности.

Поскольку UnitTestBot нацелен на генерацию модульных тестов, для каждого теста требуются ожидаемые результаты. Поэтому для фаззинга требуется вызов кода пользователя на указанных входных данных. Кроме того, продвинутый фаззинг, реализованный в UnitTestBot, использует информацию,

такую как покрытие инструкций, для адаптации своей генерации с использованием генетического алгоритма. Поэтому в этом случае необходима подсистема конкретного исполнения.

Фаззинг с движком динамического символьного исполнения открывает возможности для новых алгоритмов [6, 9, 11], которые объединяют преимущества этих подходов и могут улучшить генерацию тестов. Кроме того, UnitTestBot уже независимо комбинирует и чередует эти подходы, что значительно увеличивает покрытие кода и качество тестов.

В заключение, расширение возможностей UnitTestBot путем создания виртуальной машины динамического символьного исполнения оказалось полезным для продукта. Особо следует отметить подсистемы минимизации тестов и фаззинга. Минимизация тестов эффективно сокращает количество тестов при сохранении качества, а фаззинг помогает выявлять уязвимости программы с помощью случайных или непредвиденных входных данных. Подсистема конкретного исполнения имеет важное значение для обеих функций. Кроме того, подсистема динамического символьного исполнения позволяет реализовывать новые алгоритмы и может улучшить генерацию тестов. В целом, подсистема конкретного исполнения значительно повысила покрытие кода и качество тестов в UnitTestBot, учитывая предоставленные возможности по реализации новых алгоритмов.

3.4. Результаты обнаружения недетерминированного поведения

Поскольку UnitTestBot использует фаззинг и конкретное исполнение неполных символьных состояний, возникают ситуации, когда UnitTestBot создает мигающие тесты. Например, в коде, использующем класс Random, показанном на рисунке 14a, функция создает экземпляр класса Random и затем возвращает сумму входной переменной и случайного целого числа от одного до четырех.

Без реализации обнаружения недетерминированности в конкретном движке, фаззинг приводит к мигающему тесту (см. рисунок 14b). Однако эту проблему устраняет расширение конкретного исполнения, реализованное в контексте данного диплома (см. рисунок 14c). Сгенерированный тест

```java
public class ClassWithRandom {
    public int randomSum(int x) {
        Random r = new Random();
        return r.nextInt(1, 4) + x;
    }
}
```

(a) Тестируемый класс с использованием класса Random.

```java
public final class ClassWithRandomTest {
    @Test
    public void testRandomSumReturns8() {
        ClassWithRandom classWithRandom = new ClassWithRandom();

        int actual = classWithRandom.randomSum(6);

        assertEquals(8, actual);
    }
}
```

(b) Результат работы фаззинга без обнаружения недетерминированности.

```java
public final class ClassWithRandomTest {
    @Test
    public void testRandomSumReturns8() {
        MockedConstruction mockedConstruction = null;
        try {
            mockedConstruction = mockConstruction(Random.class,
                (Random randomMock, Context context) ->
                    (when(randomMock.nextInt(anyInt(), anyInt())))
                        .thenReturn(2)
            );
            ClassWithRandom classWithRandom = new ClassWithRandom();

            int actual = classWithRandom.randomSum(6);

            assertEquals(8, actual);
        } finally {
            mockedConstruction.close();
        }
    }
}
```

(c) Результат работы фаззинга с обнаружением недетерминированности.

Рис. 14: Пример для демонстрации обнаружения недетерминированного поведения.

мокирует создание экземпляра класса Random и гарантирует, что созданный экземпляр возвращает число два в качестве результата вызова метода nextInt.

В заключение, проблема использования фаззинга и выполнения неполных символьных состояний в UnitTestBot, приводящая к мигающим тестам, решена реализацией обнаружения недетерминированности в конкретном движке. Такое расширение позволяет создавать мок-экземпляр класса Random, который гарантирует предсказуемое поведение и обеспечивает надежные и предсказуемые тесты, как показано на рисунке 14.

3.5. SBFT 2023

SBFT 2023[14] – это соревнование для статических анализаторов, проведенное в 2023 году, где одна из категорий является генерация тестов для Java. UnitTestBot принял участие в этом соревновании по данной категории, используя реализованную и описанную в этом дипломе подсистему конкретного исполнения, а также модуль фаззинга.

Также в соревновании участвовали такие инструменты генерации тестов как

- Kex[15] – платформа для генерации тестов на основе динамического символьного исполнения;
- Randoop[16] – инструмент, использующий случайное тестирование;
- Evosuite[17] – платформа, генерация тестов которой основана на SBST подходе, который также является методом случайного тестирования.

Сравнение инструментов проводилось на разных больших проектах, используя различные метрики, в том числе покрытие строк и покрытие веток. Для этого использовался ранее упомянутый инструмент Jacoco. Проекты, используемые для измерения: Apache Commons Collections, JSoup, ta4j, Spatial4j, threeten-extra.

[14]SBFT 2023: https://sbft23.github.io/
[15]Kex: https://github.com/vorpal-research/kex
[16]Randoop: https://github.com/randoop/randoop
[17]Evosuite: https://www.evosuite.org/

По проведенным экспериментам, получились следующие данные. Медиана покрытия по строкам и по веткам по каждому классу всех проектов для UnitTestBot – 62.1% и 50% для генерации с 60 секундами на класс соответственно, и 68.0% и 55.5% с 120 секундами на класс соответственно. В результате, UnitTestBot занял второе место, проиграв Evosuite по покрытию, у которого показатели были в основном выше 90%.

3.6. Будущие улучшения

Несмотря на то, что UnitTestBot является одним из лучших инструментов для автоматической генерации тестов, а разработанная подсистема улучшает качество его работы, у продукта все еще есть несколько недостатков.

Во-первых, у UnitTestBot нет возможности разделять между собой состояния фаззинга и динамического символьного исполнения. Это приводит к повторению проделываемой работы. Существуют различные способы комбинирования этих методов, чтобы сделать анализ более эффективным и решить эту проблему. Например, информация о покрытии кода после фаззинга может быть использована для сокращения путей выполнения для последующего символьного анализа. Основным препятствием при реализации этой функциональности может быть в сложности перевода покрытия инструкций байт-кода в путь графа исполнения, используемого в символьном движке.

Следующая точка улучшения связана со стратегией комбинирования символьного и конкретного исполнений. Как уже упоминалось выше, UnitTestBot комбинирует их достаточно поверхностно: запускается конкретное исполнение, когда символьное достигает терминальных состояний или из-за ограничений SMT-решателя. Хотя такой подход и улучшает качество тестов, расширенные стратегии могут улучшить и оптимизировать генерацию тестов еще лучше.

Например, анализ функций, независимых от символьных переменных, может быть ускорен, поскольку результат символьного исполнения таких функций не повлияет на символьное состояние. Однако результат этих функций по-прежнему важен для дальнейшего анализа. Поэтому его можно вы-

зывать конкретным исполнением без использования символьного. Основной проблемой при реализации этой функции может быть понимание того, что метод не зависит от символьных переменных, так как это требует дополнительного анализа.

Обобщая вышесказанное, несмотря на то, что UnitTestBot является мощным инструментом для автоматической генерации тестов, всё ещё существует ряд ограничений и проблем. Например, невозможность совместного использования состояний между фаззингом и динамическим символьным исполнением, а также необходимость улучшения стратегии комбинирования символьного и конкретного исполнений. Несмотря на эти ограничения, UnitTestBot остается ценным инструментом для автоматизированного тестирования программного обеспечения.

Заключение

Данный диплом достиг своей цели. Разработана и реализована виртуальная машина для динамического символьного исполнения на языках программирования Java и Kotlin на основе уже имеющегося символьного движка и прототипа подсистемы конкретного исполнения в UnitTestBot. Разработанная подсистема также интегрирована в инструмент UnitTestBot, что сделало генерацию тестов более надежной без снижения покрытия кода. В процессе реализации были получены следующие результаты:

1. Проведен обзор существующих инструментов, основанных на динамическом символьном исполнении: JDart, ACTEve и Triton;

2. Изучена структура и кодовая база инструмента UnitTestBot;

3. Разработана и реализована архитектура подсистемы конкретного исполнения, взяв за основу, доработав и улучшив уже готовый прототип данной подсистемы;

4. Реализованная подсистема интегрирована в UnitTestBot, тем самым получена виртуальная машина для динамического символьного исполнения, поддерживающая языки программирования Java и Kotlin;

5. Установлено, что реализованный движок улучшает генерацию тестов путем существенного сокращения числа некорректных и ненадежных тестов без потери покрытия кода;

6. Получены новые возможности для разработки и реализации новых подсистем и алгоритмов для улучшения качества UnitTestBot, в частности, уже реализованные модули фаззинга и минимизации.

7. UnitTestBot с реализованной подсистемой конкретного исполнения принял участие в соревновании по генерации тестов для Java, SBFT 2023, где занял второе место, показав хорошие результаты работы на реальных проектах.

Список литературы

[1] Shaukat Ali и др. “A systematic review of the application and empirical investigation of search-based test case generation”. В: *IEEE Transactions on Software Engineering* 36.6 (2009), с. 742—762.

[2] Saswat Anand и др. “Automated concolic testing of smartphone apps”. В: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, с. 1—11.

[3] Ricardo Anido и др. “Test suite minimization for testing in context”. В: *Software Testing, Verification and Reliability* 13.3 (2003), с. 141—155.

[4] Roberto Baldoni и др. *A Survey of Symbolic Execution Techniques*. 2018. arXiv: 1610.00502 [cs.SE].

[5] Clark Barrett и др. “CVC4”. В: *Computer Aided Verification*. Под ред. Ganesh Gopalakrishnan и Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, с. 171—177. ISBN: 978-3-642-22110-1.

[6] Pietro Braione и др. “Combining symbolic execution and search-based testing for programs with complex heap inputs”. В: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017, с. 90—101.

[7] Cristian Cadar и Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. В: *Commun. ACM* 56.2 (февр. 2013), с. 82—90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. URL: https://doi.org/10.1145/2408776.2408795.

[8] Ting Chen и др. “State of the art: Dynamic symbolic execution for automated test generation”. В: *Future Generation Computer Systems* 29.7 (2013), с. 1758—1773.

[9] Kivanc Doganay и др. *Search-based testing for embedded telecommunication software with complex input structures: An industrial case study*. 2014.

[10] Niklas Eén и Niklas Sörensson. “An Extensible SAT-solver”. В: *International Conference on Theory and Applications of Satisfiability Testing*. 2003.

[11] Juan Pablo Galeotti, Gordon Fraser и Andrea Arcuri. “Improving search-based test suite generation with dynamic symbolic execution”. В: *2013 ieee 24th international symposium on software reliability engineering (issre)*. IEEE. 2013, с. 360—369.

[12] Patrice Godefroid, Nils Klarlund и Koushik Sen. “DART: Directed automated random testing”. В: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, с. 213—223.

[13] Patrice Godefroid, Michael Y Levin, David A Molnar и др. “Automated whitebox fuzz testing.” В: *NDSS*. Т. 8. 2008, с. 151—166.

[14] Eugene Goldberg и Yakov Novikov. “BerkMin: A fast and robust Sat-solver”. В: *Discrete Applied Mathematics* 155.12 (2007). SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing, с. 1549—1561. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2006.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0166218X06004616.

[15] Kasper Luckow и др. “JD art: a dynamic symbolic analysis framework”. В: *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22*. Springer. 2016, с. 442—459.

[16] Leonardo de Moura и Nikolaj Bjørner. “Z3: an efficient SMT solver”. В: т. 4963. Апр. 2008, с. 337—340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24.

[17] Aina Niemetz и Mathias Preiner. *Bitwuzla at the SMT-COMP 2020*. 2020. arXiv: 2006.01621 [cs.LO].

[18] Corina Păsăreanu и Willem Visser. “A survey of new trends in symbolic execution for software testing and analysis”. В: *International Journal on Software Tools for Technology Transfer* 11 (окт. 2009), с. 339—353. DOI: 10.1007/s10009-009-0118-1.

[19] Florent Saudel и Jonathan Salwan. “Triton: Concolic execution framework”. В: *Symposium sur la sécurité des technologies de l’information et des communications (SSTIC)*. 2015.

[20] Shin Yoo и Mark Harman. “Regression testing minimization, selection and prioritization: a survey”. В: *Software testing, verification and reliability* 22.2 (2012), с. 67—120.