Санкт-Петербургский государственный университет

Алексеев Артур Игоревич

Выпускная квалификационная работа

Автоматическая генерация интеграционных тестов для веб приложений

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2021 «Программная инженерия»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Д. А. Мордвинов

Консультант:
инженер-исследователь, ООО «ИИнтелКод», М. П. Костицын

Рецензент:
старший академический консультант, ООО «Техкомпания Хуавэй», к. т. н.,
Д. С. Степанов

Санкт-Петербург
2025

Saint Petersburg State University

Artur Alekseev

Bachelor's Thesis

Automated integration test generation for web applications

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2021 «Software Engineering»*

Scientific supervisor:
C.Sc., docent D.A. Mordvinov

Consultant:
research engineer at at "LLC IIntelCode" M.P. Kosticyn

Reviewer:
senior academic consultant at "LLC Tech Company Huawei", C.Sc. D.S. Stepanov

Saint Petersburg
2025

Оглавление

Введение

В настоящее время количество веб-приложений стремительно растет, а их роль становится все более значимой. Многие из них выполняют критически важные функции, такие как обработка финансовых операций или хранение персональных данных пользователей. В связи с этим необходимость их тщательного тестирования приобретает особую актуальность.

На данный момент существует множество фреймворков для создания веб-приложений, и одним из лидеров в экосистеме Java является Spring [4]. Согласно опросу Developer Ecosystem [17], Spring является самым популярным веб-фреймворком среди Java-разработчиков благодаря своей скорости, простоте и безопасности.

Современные веб-приложения представляют собой сложные системы, состоящие из множества взаимосвязанных компонентов. Из-за этой сложности ручное тестирование может быть достаточно трудоемким.

Тесты для веб-приложений можно разделить на несколько видов по уровню детализации проверяемых компонентов. Например, модульные тесты обеспечивают самый высокий уровень изоляции и проверяют работу отдельных модулей, а интеграционные тесты анализируют работу всей системы целиком, от получения запроса до отправки ответа. Основное преимущество интеграционных тестов перед другими видами тестирования заключается в том, что они проверяют не только работу компонентов, но и взаимодействие между ними, выявляя проблемы, которые не обнаруживаются при других видах тестирования.

Однако, поскольку интеграционные тесты проверяют работу множества компонентов, их создание требует значительных усилий. Кроме того, при ручном тестировании некоторые уязвимости могут остаться незамеченными. Одним из решений этой проблемы является автоматическая генерация интеграционных тестов.

Среди различных подходов к автоматической генерации тестов особое место занимает символьное исполнение [18] — метод анализа белого ящика, позволяющий находить входные данные для тестов с макси-

мальным покрытием кода. Тем не менее, символьное исполнение имеет ряд ограничений при исследовании реальных приложений: экспоненциальный рост числа состояний, высокое потребление вычислительных ресурсов, сложности при взаимодействии с внешними библиотеками и файловой системой. Одним из возможных решений является комбинирование реального исполнения программы и классического символьного анализа.

Данная работа выполнена в рамках проекта USVM[1] — инструмента, предназначенного для символьного анализа программного кода на нескольких языках, включая Java.

[1]https://github.com/UnitTestBot/usvm

1. Постановка задачи

Целью данной работы является разработка подсистемы для автоматической генерации интеграционных тестов веб-приложений, построенных на фреймворке Spring. Для достижения данной цели были сформулированы следующие задачи.

1. Выполнить обзор особенностей фреймворка Spring, техники символьного исполнения, проекта USVM и существующих решений в области автоматической генерации интеграционных тестов.
2. Спроектировать архитектуру решения для автоматической генерации интеграционных тестов для веб-приложений.
3. Реализовать спроектированное решение в проекте USVM.
4. Провести сравнение результатов реализованой системы с аналогами.

2. Обзор

В данной главе описана техника символьного исполнения, рассмотрены особенности создания веб-приложений на фреймворке Spring, приведены основные особенности символьного движка USVM, а также описаны существующие инструменты для автоматической генерации интеграционных тестов.

2.1. Символьное исполнение

Символьное исполнение [11, 18, 24, 32] представляет собой метод анализа программ, позволяющий определять все возможные наборы входных данных, обеспечивающие выполнение каждого из потенциальных путей исполнения программы. В отличие от традиционного выполнения программы с реальными входными значениями, при котором анализируется лишь один путь исполнения, символьное исполнение способно исследовать множество путей выполнения программы одновременно.

Процесс символьного исполнения начинается с присваивания символьных значений всем входным данным программы. Анализ осуществляется символьным движком, который для каждого исследуемого пути выполнения обновляет символьное состояние. Данное состояние является фундаментальным понятием символьного исполнения и включает следующие компоненты.

1. Инструкция — текущая исполняемая инструкция программы.
2. Ограничения условия пути — набор ограничений на символьные значения, которые должны быть выполнены для того чтобы исполнение пошло по этому пути исполнения.
3. Символьная память — отображение имен переменных программы на их символьные значения или выражения.

Символьный интерпретатор выполняет инструкции программы, модифицируя ограничения пути и состояние памяти. При обработке ин-

струкции ветвления происходит обновление ограничений пути и создание нового символьного состояния, тогда как выполнение команды присваивания приводит к изменению содержимого символьной памяти.

По завершении символьного исполнения для определения конкретных значений входных данных применяются SMT-решатели [9, 10]. Его функция заключается в проверке существования удовлетворяющих значений для логической формулы условия пути и, в случае их существования, в генерации модели — отображения символьных значений на конкретные данные. На основе полученной модели может быть сформирован тестовый сценарий.

2.1.1. Динамическое символьное исполнение

Теоретически символьное исполнение способно анализировать все возможные пути выполнения программы, однако его практическое применение для исследования реальных приложений сопряжено со значительными трудностями [28, 34]. Классический подход сталкивается с рядом фундаментальных ограничений, включая: невозможность анализа внешних вызовов, экспоненциальный рост числа состояний, чрезмерное потребление вычислительных ресурсов. Эти ограничения преодолеваются с помощью динамического символьного исполнения, объединяющего классический символьный анализ с конкретным выполнением программы [26, 27].

Подход Concolic является одной из техник анализа, которая использует динамическое символьное исполнение [8, 7, 23]. Данный метод параллельно выполняет программу с конкретными значениями и символьный анализ по одному пути. Символьное исполнение сохраняет ограничения условия пути, которые затем преобразуются SMT-решателем в новые входные данные для исследования других путей. Итеративное повторение этого процесса обеспечивает полное покрытие путей выполнения.

Одним из инструментов, которые реализуют такой подход для платформы .NET, является Pex [29]. Для программ на языке Java существует инструмент JCute [22].

Другим подходом, реализующим принцип сочетания конкретного и символьного анализа, является выборочное символьное исполнение [21]. Суть подхода заключается в избирательном символьном исследовании лишь ключевых компонентов программы, тогда как остальные участки выполняются конкретно. Данная методика сохраняет корректность анализа [34], одновременно значительно снижая требования к вычислительным ресурсам за счет преимущественного использования конкретного исполнения.

```
int b(int x) {
    return x + 1;
}

int a(int i) {
    int result = b(3);
    return result + i;
}
```

Рис. 1: Пример программы для демонстрации выборочного символьного исполнения

Рассмотрим пример на листинге 1, иллюстрирующий принцип выборочного символьного исполнения. В данном примере представлены две функции: `a` и `b`. Предположим, что функция `b` исключена из области символьного анализа.

В таком случае, при исследовании функции `a`, во время исполнения инструкции `int result = b(3)` инструмент, для вычисления выражения `b(3)`, вместо символьного исследования, исполнит вызов функции `b` конкретно. Далее, полученный результат (`4`) будет записан в память. Применение данного подхода позволит уменьшить число исследуемых инструкций, то есть увеличит эффективность инструмента.

2.1.2. Аппроксимации

При символьном анализе программ существуют методы, для которых точная обработка всех возможных состояний с сохранением полной семантики представляет значительные трудности. Кроме того, некоторые методы требуют модификации для добавления специальной логики, необходимой для символьного анализа. Для решения этих проблем применяются различные виды аппроксимаций [3].

В контексте символьного исполнения аппроксимация означает замену сложных или неанализируемых фрагментов программы упрощенными символьными моделями. Такой компромисс между точностью и производительностью позволяет расширить покрытие анализа. В контексте символьного исполнения выделяют два вида аппроксимаций: символьные модели и подмена кода.

Символьные модели реализуются на уровне символьного движка и позволяют работать с символьным состоянием, имея доступ к большому количеству внутренних функций символьной машины. Данные виды аппроксимаций позволяют реализовать разные механизмы, которым не требуется сложная логика, но для которых необходима работа напрямую с символьным состоянием. Например, работа с рефлексией или создание API для техники подмены кода.

Подмена кода также является видом аппроксимаций и позволяет модифицировать функциональность методов и классов с использованием реального Java-кода. Этот тип аппроксимаций удобен для реализации упрощенных алгоритмов, не требующих интенсивной работы с символьным состоянием. Специальное API предоставляет доступ к ключевым функциям символьного движка, таким как создание новых символьных значений или наложение дополнительных ограничений пути. Примером применения этой техники являются упрощенные реализации символьных коллекций.

2.2. Фреймворк Spring

Фреймворк Spring предлагает удобную модель взаимодействия и конфигурации компонентов в Java-проектах. Он включает контейнер внедрения зависимостей с поддержкой различных слоев: инфраструктуры, событий, ресурсов, интернационализации, валидации и работы с базами данных. На этой основе строятся ключевые компоненты приложений, а также создаются другие фреймворки, упрощающие разработку веб-приложений, такие как Spring MVC и Spring Boot.

Spring MVC воплощает архитектурный шаблон Model-View-Controller, предлагая разработчикам четко разделенные компоненты: модели для хранения данных, контроллеры для обработки запросов и представления для визуализации информации. Контроллеры, помеченные специальными аннотациями, содержат методы-обработчики запросов. Эти методы принимают данные из запроса, работают с моделью и определяют подходящее представление для ответа. Благодаря встроенному контейнеру и механизмам внедрения зависимостей, Spring MVC автоматически управляет жизненным циклом компонентов и их взаимодействием, что упрощает создание масштабируемых веб-приложений.

В веб-приложениях бизнес-логика обычно реализуется в сервисах — наборе классов, помеченных аннотацией `@Service`. Работа с данными делегируется репозиториям — интерфейсам или классам, помеченным аннотацией `@Repository`, которые отвечают за взаимодействие с базой данных и доступ к данным.

Важную роль во фреймворке Spring MVC играет сервлет. Сервлет представляет собой Java-программу, работающая на сервере и обрабатывающая HTTP-запросы. В Spring MVC основной сервлет, обрабатывающий запросы, — это `DispatcherServlet`. Он отвечает за поиск подходящего обработчика (метода контроллера), управление моделью данных и формирование представления.

Spring Boot — это расширение экосистемы Spring, разработанное для упрощения создания автономных, готовых к эксплуатации приложений. Ключевыми преимуществами фреймворка являются автомати-

ческая конфигурация компонентов, встроенные серверы и минимизация ручной настройки.

2.2.1. Жизненный цикл запроса

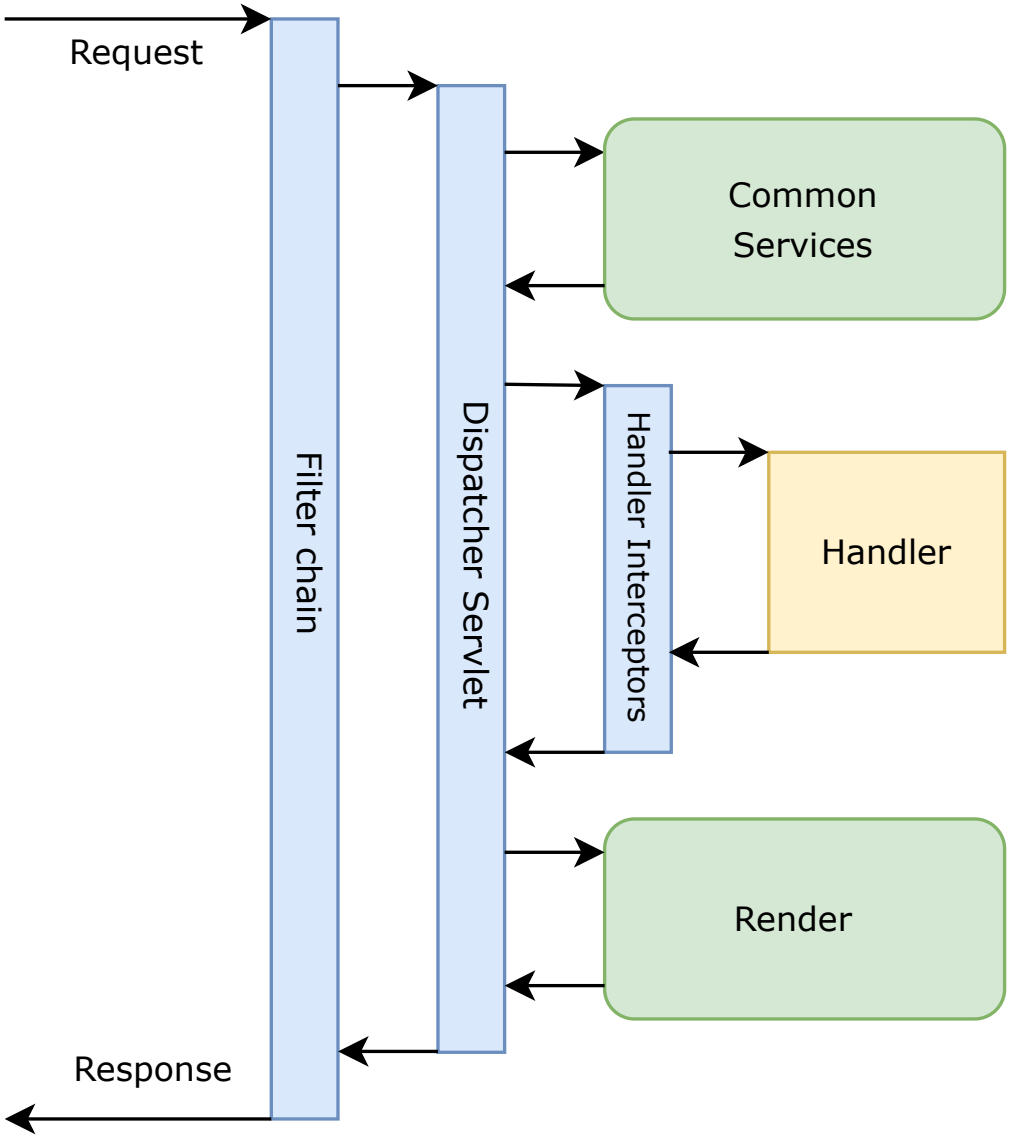

Рис. 2: Жизненный цикл запроса

Рассмотрим жизненный цикл запроса в Spring MVC, показанный на рис. 2. Процесс начинается с получения HTTP-запроса от клиента, который сервер преобразует в объект для последующей обработки компонентами системы. Затем запрос проходит через несколько этапов обработки.

Первым этапом является цепочка фильтров, которые являются частью сервера, а не фреймворка Spring [5]. Важно отметить, что на данном этапе еще не созданы объекты, характерные для обработки сервлетом, например, модель. Фильтры могут прервать обработку запроса до его передачи сервлету. Типичные задачи фильтров включают авторизацию, работу с файлами, логирование и ведение журналов аудита.

После фильтров запрос поступает в сервлет Spring MVC, где обработка осуществляется через цепочку объектов `HandlerInterceptor` и методы контроллеров. Сначала запрос проходит через интерсепторы, затем фреймворк определяет соответствующий метод контроллера для обработки.

`HandlerInterceptor` позволяют выполнять логику до вызова метода контроллера, работая в контексте сервлета. Они часто используются для тонкой настройки авторизации, специализированного логирования и манипуляций с моделью данных.

Основная обработка запроса происходит в методах контроллеров, помеченных аннотациями `@Controller` или `@RestController`. Фреймворк Spring MVC определяет нужный метод на основе URL и HTTP-метода запроса, вычисляет аргументы и его вызывает. Например, метод с аннотацией `@GetMapping(''/method/id'')` в классе, помеченном `@RequestMapping(''/controller'')`, будет обрабатывать запросы к `GET /controller/method/42`. В этих методах реализуется бизнес-логика: работа с базой данных, заполнение модели, определение представления или перенаправление.

После выполнения метода контроллера начинается этап формирования ответа. Для REST-контроллеров возвращается результат метода, для традиционных контроллеров — отрисовывается представление с данными из модели. При необходимости устанавливаются соответ-

ствующие HTTP-заголовки.

Завершающий этап включает прохождение ответа через интерсепторы и фильтры перед отправкой клиенту в указанном формате.

2.2.2. Виды тестов для веб-приложений

Фреймворк Spring предоставляет комплекс инструментов для автоматизированного тестирования веб-приложений, поддерживающий различные уровни изоляции компонентов. Основные виды тестирования перечислены ниже.

- Модульные тесты, обеспечивающие наивысший уровень изоляции компонентов. Для их реализации достаточно использовать специализированные фреймворки, такие как JUnit [35], без необходимости в дополнительных инструментах Spring.
- Компонентное тестирование, позволяющее проверять отдельные части системы с имитацией зависимостей. Типичный пример — тестирование контроллеров с мокированными сервисами и репозиториями.
- Интеграционные тесты, охватывающие полный цикл обработки запроса от получения до формирования ответа. Такой подход позволяет верифицировать не только отдельные компоненты, но и их взаимодействие в системе.

Фреймворк Spring включает инструмент `MockMvc`, который существенно облегчает написание интеграционных и компонентных тестов. Он позволяет тестировать веб-слой без развертывания реального сервера, исключая необходимость сетевых соединений. `MockMvc` предоставляет API для написания интеграционных тестов в стиле модульных. Для создания интеграционного теста необходимы следующие шаги.

1. Добавить тестовому классу аннотацию `@SpringBootTest`.
2. Объявить поле типа `MockMvc` с аннотацией `@Autowired`.

Фреймворк автоматически инициализирует контекст приложения и внедряет подготовленный экземпляр `MockMvc`. Для компонентного тестирования вместо `@SpringBootTest` используется `@WebMvcTest`, позволяющий изолировать тестируемые компоненты с помощью `MockBean` и `Mockito`.

Процесс тестирования включает:

- Вызов метода `perform` для имитации запроса
- Цепочку проверок `andExpect` для валидации результатов

2.3. Существующие решения

Для автоматической генерации тестов для веб-приложений, основанных на фреймворке Spring, применяются два основных подхода: модульное тестирование отдельных методов и интеграционное тестирование с использованием фаззинга.

Для интеграционного тестирования используется фаззинг [13] — метод автоматического тестирования, который использует принцип черного ящика [33]. Эта техника выявляет уязвимости и проблемы, генерируя случайные входные данные. Особенность подхода черного ящика заключается в отсутствии доступа к исходному коду тестируемой системы при генерации тестовых данных.

Фаззинг доказал свою эффективность в поиске уязвимостей — например, специалисты Google обнаружили с его помощью более 25 000 ошибок [6]. Однако по сравнению с символьным исполнением у фаззинга есть существенные ограничения.

- Техника требудет большое время поиска входных данных, вызывающих интересное поведение системы, относительно символьного исполнения.
- Даже используя методы, которые позволяют анализировать код тестируемой системы, находя при этом интересные значения, техника полагается на случайность, что не позволяет гарантировать максимальноое покрытие.

2.3.1. Инструмент UnitTestBot

Для автоматизации генерации тестов и анализа Java-приложений разработан инструмент UnitTestBot Java[2] [31, 19], представляющий собой часть семейства решений UnitTestBot [30]. Данный инструмент сочетает методы фаззинга и символьного выполнения, что позволяет эффективно создавать тестовые сценарии, в том числе интеграционные тесты для веб-приложений на основе фреймворка Spring.

UnitTestBot Java реализован в виде плагина для IntelliJ IDEA, предлагающего удобный интерфейс для генерации интеграционных тестов. Процесс создания тестовых сценариев для Spring-контроллеров осуществляется в несколько интуитивных шагов: пользователь вызывает меню с помощью горячих клавиш, выбирает конфигурацию Spring, предпочитаемый тестовый фреймворк и стратегию мокирования зависимостей, затем указывает целевые методы для тестирования и устанавливает временные ограничения. В результате плагин автоматически формирует тестовый класс с набором готовых тестовых сценариев.

В основе инструмента лежит фаззер, использующий методику серого ящика. Этот подход предполагает генерацию случайных входных значений для конкретного выполнения тестируемого кода с последующим анализом изменений в пути выполнения программы. Полученная информация о поведении системы позволяет фаззеру адаптивно корректировать входные данные для последующих итераций, постепенно увеличивая покрытие тестами и выявляя потенциальные уязвимости.

2.3.2. Инструмент RESTler

Фаззер RESTler — это инструмент, разработанный Microsoft Research для тестирования приложений, использующих REST API [1]. В отличие от традиционных подходов, данный инструмент использует специальный алгоритм генерации тестовых данных, основанный на анализе специфики API и извлечении полезной информации из истории выполненных запросов.

[2]https://github.com/UnitTestBot/UTBotJava

Основой для работы RESTler служит формализованная спецификация OpenAPI [12], предоставляющая структурированное описание тестируемого веб-сервиса. Данная спецификация позволяет описывать, создавать и визуализировать веб-сервисы, использующие REST.

Особенностью RESTler является способность формировать цепочки взаимосвязанных запросов, где выходные данные одного запроса становятся входными для следующего. Такой подход особенно эффективен для тестирования типичных REST-сценариев, включающих последовательное создание, модификацию и удаление ресурсов.

Инструмент RESTler имеет открытый исходный код [3], а также документацию, подробно описывающую его функциональность.

2.3.3. Фаззер EvoMaster

Инструмент EvoMaster[4] — это фаззер для автоматической генерации тестов для приложений, использующих REST API, GraphQL или RPC. Этот инструмент не просто находит входные данные, вызывающие ошибки, но и формирует готовые тестовые сценарии, в том числе для Java-приложений.

Ключевой особенностью EvoMaster является применение алгоритмов машинного обучения. Инструмент сочетает эволюционные алгоритмы с динамическим анализом программы для генерации рабочих сценариев.

EvoMaster поддерживает два режима тестирования: черного и белого ящика. В режиме черного ящика инструмент взаимодействует с API приложения независимо от языка реализации, однако эффективность такого подхода ниже по сравнению с тестированием белого ящика.

Для работы фаззера в режиме белого ящика требуется ручная инструментация кода, доступная для JVM-приложений. Процесс включает подключение специальной библиотеки, создание класса-наследника

[3]https://github.com/microsoft/restler-fuzzer

[4]https://github.com/WebFuzzing/EvoMaster

`EmbeddedSutController` или `ExternalSutController` и реализацию соответствующих методов.

По завершении работы EvoMaster генерирует файлы с интеграционными тестами на выбранном языке и платформе.

2.4. Инструмент USVM

USVM — универсальный инструмент для анализа кода, поддерживающий символьное исполнение программ на различных языках программирования, включая Java, Python и TypeScript. Инструмент содержит набор базовых классов, реализующих основные механизмы символьного исполнения, независимые от конкретного языка. Для добавления поддержки нового языка необходимо унаследоваться от этих классов и реализовать языко-специфичную логику, такую как представление памяти, загрузка исходного кода и особенности интерпретации.

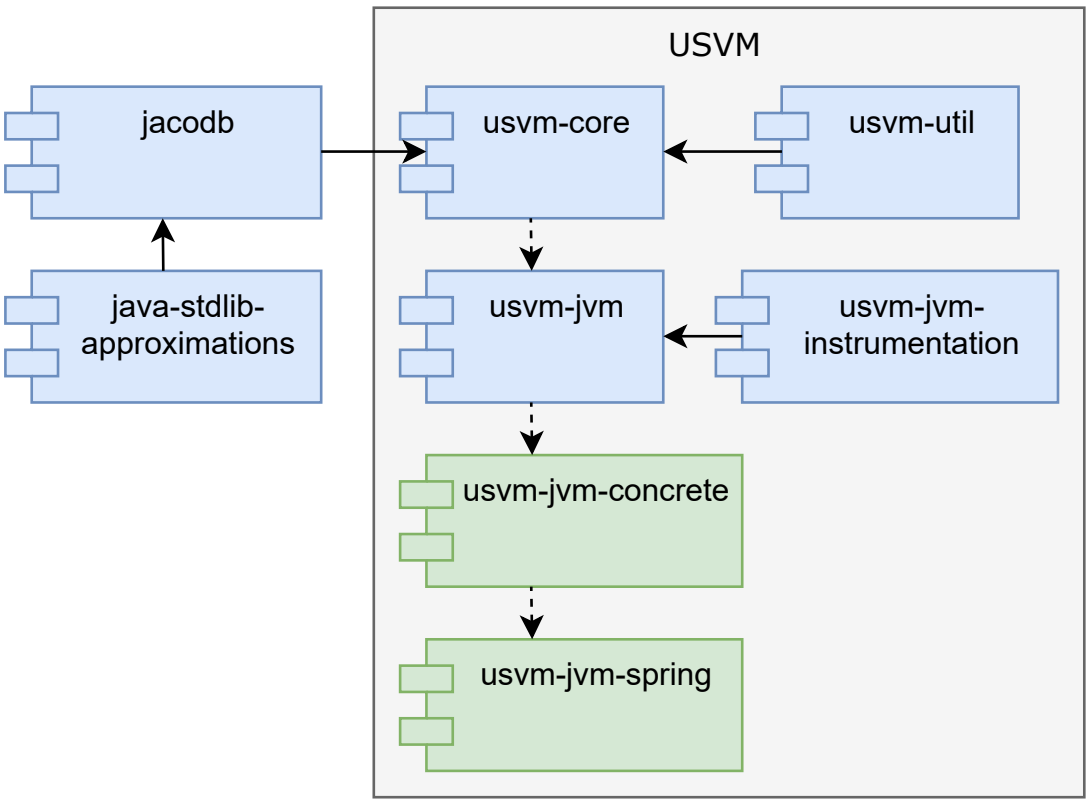

Рис. 3: Диаграмма модулей USVM

Архитектура проекта USVM, изображенная на Рис. 3, включает модули, перечисленные ниже.

- Библиотека JacoDB[5] отвечает за загрузку и хранение пользовательского кода. В USVM она используется для работы с байт-кодом анализируемых проектов, включая загрузку, представление и обработку типов.

- Проект `java-stdlib-approximation`[6] обеспечивает создание аппроксимаций. При загрузке кода через JacoDB методы и классы, помеченные специальными аннотациями, заменяются на альтернативные реализации из этого модуля.

[5]https://github.com/UnitTestBot/jacodb
[6]https://github.com/UnitTestBot/java-stdlib-approximations

- Ядро символьного исполнения реализовано в модуле `usvm-core`[7], который содержит базовые классы: интерпретатор, модель памяти, символьные выражения и компоненты для взаимодействия с SMT-решателями. Для добавления поддержки нового языка требуется наследование от этих классов с реализацией соответствующей языку логики.

- Специфика символьного исполнения Java сосредоточена в модуле `usvm-jvm`, содержащем языко-ориентированные реализации.

- Модуль `usvm-util` включает часто используемые типы данных и алгоритмы, применяемые в различных компонентах USVM.

- Модуль `jvm-instrumentation` предоставляет инструменты для работы с пользовательским кодом, включая исполнение методов.

В рамках данной работы в USVM были добавлены два новых модуля. Модуль `usvm-jvm-concrete` реализует механизмы выборочного символьного исполнения: хранение реальных Java-объектов в памяти, выполнение конкретных вызовов и специализированные символьные модели. Модуль `usvm-jvm-spring` обеспечивает поддержку веб-приложений на фреймворке Spring.

Инструмент USVM поддерживает создание тестов из символьного состояния и проверку их воспроизводимости. Для этого в инструменте реализованы: представление теста как последовательности инструкций и вызова метода, механизм воспроизведения и проверки результатов, а также преобразование тестов в Java-код.

[7] https://github.com/UnitTestBot/usvm

3. Архитектура

В данной главе описан алгоритм символьного исследования веб-приложений на фреймворке Spring, актуальные проблемы, возникающие при символьном исследовании веб-приложений, и их возможные решения. Рассмотрено конкретное исполнение и конкретная память, которые являются одной из основных идей алгоритма исследования. Также описана реализация подсистемы для автоматической генерации интеграционных тестов.

При исследовании веб-приложения важно четко разделять пользовательский код, который является объектом анализа, и код фреймворка, обеспечивающий его функционирование.

Фреймворк Spring включает не только конвейер обработки запросов, требующий символьного анализа, но и HTTP-сервер, отвечающий за прием запросов, их валидацию и преобразование в объекты для последующей обработки. Также фреймворк Spring отвечает за различные операции с заголовками и параметрами запроса, определение обработчика на основе пути и метода, а также множество вспомогательных процедур, инициализирующих контекст и формирующих конвейер обработки.

Пользовательский код, в свою очередь, содержит бизнес-логику приложения: методы контроллеров, взаимодействие с сервисами и репозиториями, а также другие алгоритмы обработки данных. Именно эта часть представляет интерес для символьного исследования.

Анализ методов фреймворка Spring, таких как запуск сервера или инициализация контекста, не только вычислительно сложен, но и не имеет практической ценности. Эти участки кода, как правило, не содержат сложных ветвлений, поскольку их выполнение не зависит от входных данных. Однако если метод фреймворка обрабатывает входные данные, его реализация исследуется с применением специальных аппроксимаций.

Опираясь на эти идеи, для символьного исследования веб-приложений был разработан специализированный подход. Его суть заключает-

ся в конкретном исполнении максимально большого количества методов фреймворка, при этом исследование кода пользователя происходит символьно.

3.1. Детальное описание алгоритма

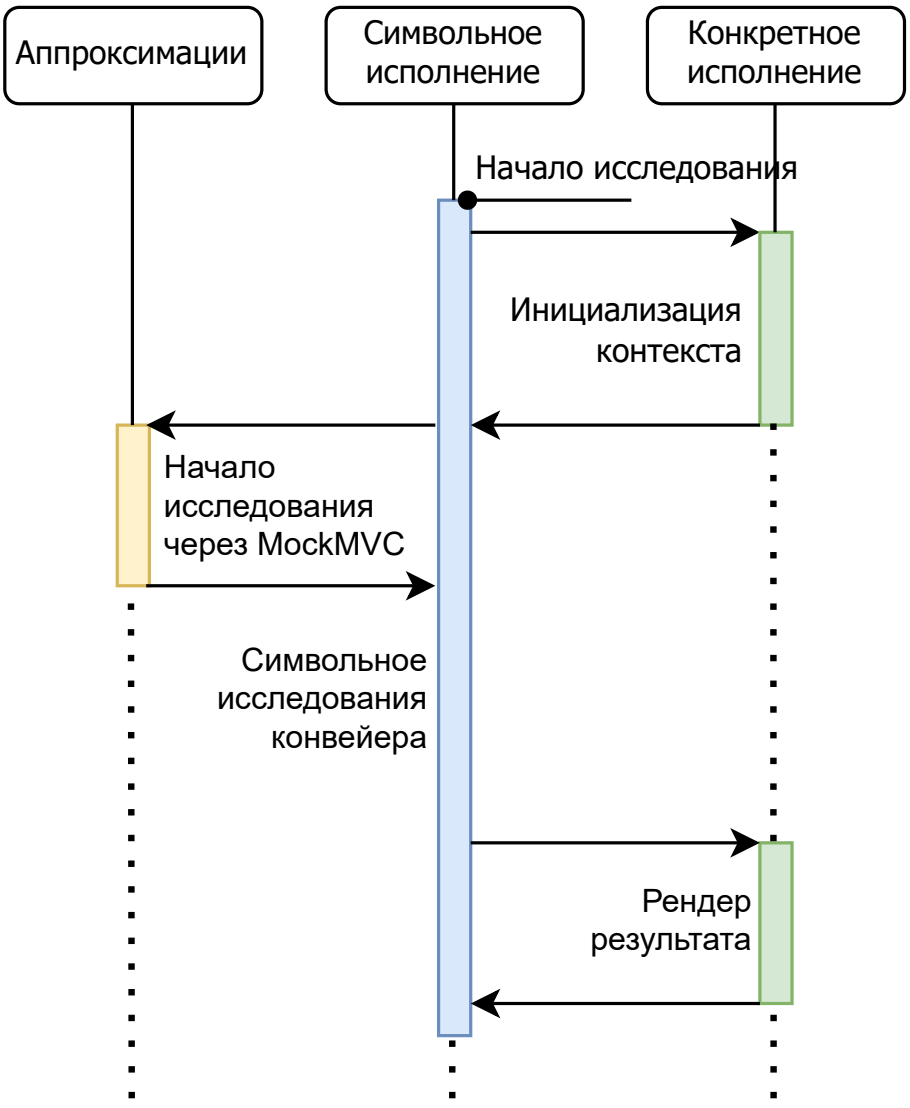

Рис. 4: Диаграмма исполнения веб-приложения

Далее приводится детальное описание алгоритма исследования, схематично представленного на Рис. 4.

Исследование веб-приложения начинается с создания движком пустого интеграционного теста с аннотацией `@WebMvcTest`. Создание данного класса и дальнейшая работа с ним позволяют избежать запуска HTTP-сервера. На основе данного теста средствами фреймворка создается тестовое окружение, в рамках которого происходит инициализация контекста приложения и формирование конвейера обработки запросов для символьного исследования.

Следующий этап алгоритма анализа веб-приложений начинается в методе `afterRefresh`, который вызывается сразу после построения конвейера обработки запроса. Данный метод, с помощью подмены кода, выполняет логику по запуску обработки запроса. Символьный движок анализирует проект, идентифицируя все контроллеры и их методы, выступающие в роли обработчиков запросов. Для каждого обнаруженного обработчика создается отдельное символьное состояние. Исполнение, которое начнется с этого состояния, приведет к выполнению кода данного обработчика.

Затем, используя функциональность Spring для интеграционного тестирования, конвейер обработки запросов запускается с искусственно сформированным запросом. Этот запрос содержит необходимые данные, такие как URL и HTTP-метод, обеспечивающие вызов целевого обработчика.

На этапе символьного исполнения многие методы Spring вызываются конкретно или упрощаются с помощью аппроксимаций. Этот процесс продолжается до момента обработки результата выполнения пользовательского кода.

После завершения анализа пользовательской логики, то есть когда дальнейший код относится исключительно к фреймворку, применяется этап конкретизации. Он заключается в замене символьных данных в состоянии на конкретные значения, полученные из модели с помощью SMT-решателя и позволяет эффективно выполнять обширные участки кода, ответственные за постобработку результатов, включая генерацию

HTML-страниц на основе данных модели и выбранного представления.

3.2. Архитектура подсистемы

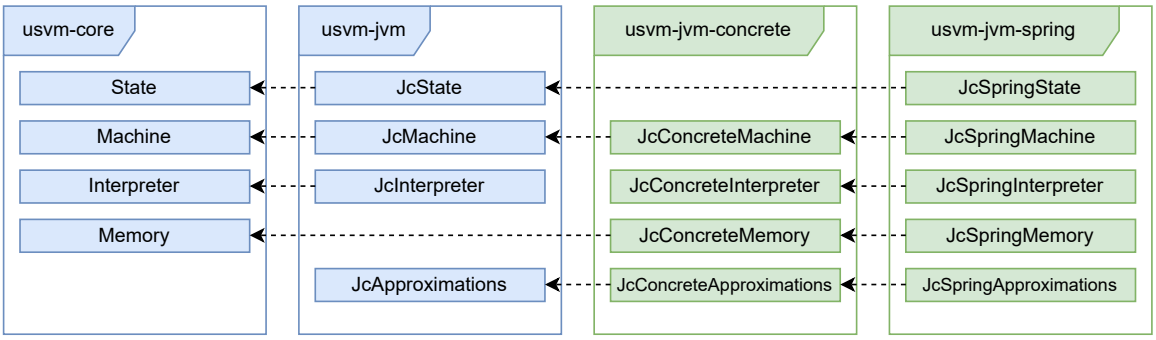

Рис. 5: Архитектура подсистемы

Для реализации данного алгоритма была создана подсистема, архитектура которой показана на Рис. 5. Подсистема состоит из двух ключевых модулей. Первый модуль, `usvm-jvm-concrete`, содержит набор классов, которые наследуют соответствующие классы, реализованные в модуле `usvm-jvm` и отвечают за работу с конкретным исполнением. Модуль включает следующие основные классы.

- `JcConcreteMachine` — модифицированная версия символьной машины с дополнительными конфигурациями и переопределенными методами создания компонентов движка.
- `JcConcreteInterpreter` — расширенный символьный интерпретатор, поддерживающий работу с конкретными вызовами и рефлексией.
- `JcConcreteMemory` — память, которая отвечает за хранение как реальных, так и символьных объектов. Класс `JcConcreteMemory` поддерживает различные операции, которые относятся к реальному исполнению, например, конкретизацию.
- `JcConcreteApproximations` — набор символьных моделей для работы с рефлексией, а также вспомогательными функциями движка.

Второй модуль, `usvm-jvm-spring`, содержит набор классов, реализующих алгоритмы и подходы, специфичные для исследования веб-приложений. Многие из них наследуются от классов проекта `usvm-jvm-conc-rete`. Основные классы из данного модуля также перечислены ниже.

- `JcSpringPinnedValues` — класс, отвечающий за работу с закрепленными значениями, специальным механизмом, который помогает записать символьные значения отдельно от запроса, чтобы сохранить его конкретность, оставляя в запросе только реальные данные. На этом механизме основаны классы для получения входных данных из запроса и алгоритмы, отвечающие за создание теста из состояния.

- `JcSpringState` — расширенное символьное состояние с поддержкой данных, специфичных для исследования веб-приложений, таких как закрепленные значения или результаты мокированных методов.

- `JcSpringMachine` — адаптированная символьная машина с переопределенной логикой подсчета покрытия.

- `JcSpringInterpreter` — интерпретатор с дополнительной поддержкой мокирования методов.

- `JcSpringMemory` — расширенное представление памяти с ограничениями на методы, которые можно вызывать конкретно.

- `JcSpringApproximations` — класс, который содержит набор символьных моделей, специфичных для исследования фреймворка Spring. В том числе, работа с закрепленными значениями и предоставление API для аппроксимаций из модуля `java-stdlib-approximations`.

3.3. Работа с входными данными

Ключевой принцип нашего алгоритма заключается в максимальном использовании конкретных данных при анализе. Однако для его реализации требуется выполнение множества методов фреймворка Spring, большинство из которых работает с HTTP-запросами. Это создает необходимость обеспечить конкретность самого запроса как объекта, сохраняя при этом символьную природу содержащихся в нем данных, таких как заголовки, параметры, URL и тело запроса.

Для решения этой проблемы был проведен анализ методов, ответственных за извлечение входных данных из HTTP-запроса. Примеры классов, отвечающих за работу с входными данными, приведены ниже.

- `ArgumentResolver` — класс, вычисляющий аргументы методов контроллера.
- `ModelBinder` — класс, отвечающий за привязку данных из запроса к объекту из модели.
- `MockHttpServletRequest` — класс, который содержит методы, использующиеся для получения входных данных напрямую из запроса.

Для таких классов были созданы аппроксимации, которые вместо работы с объектом запроса создавали новое символьное значение.

За хранение и использование таких символьных значений отвечает новый механизм — `PinnedValues`, или закрепленные значения, реализованный на уровне символьного состояния. В закрепленных значениях находятся все символьные значения, которые являются частью запроса, тем самым позволяя самому объекту запроса быть полностью конкретным.

3.4. Конкретное исполнение

При исследовании программ некоторые методы могут быть очень трудно исполнять символьно, так как внутри метода может содержать-

ся большое количество инструкций и ветвлений, сложные рекурсивные или циклические алгоритмы. Также в методе могут содержаться внешние вызовы или операции, которые не поддерживаются движком, например, чтение переменных среды, работа с файлами и директориями, или вызов методов внешних библиотек, чей язык реализации не поддерживается символьным движком. Символьный анализ такого метода может занять большое количество времени или породить огромное количество состояний.

Наш подход является частным случаем подхода выборочного символьного исполнения. Множеством методов, которые вызываются конкретно, являются методы фреймворка Spring, в которых нет вероятности ветвления и отсутствует работа с входными данными, а также исключены методы фреймворка, которые необходимо исполнять для того, чтобы достичь кода пользователя, который мы хотим анализировать символьно.

Такой подход сильно ускоряет и упрощает исследование, так как методы с нетривиальной логикой, например, парсинг строк или инициализация контекста, будут исполнены конкретно. Но для того, чтобы вызвать метод конкретно, нужно иметь реальные значения всех аргументов, статических классов, глобальных переменных и самого объекта, у которого вызывается метод.

3.4.1. Конкретная память

Для того чтобы можно было конкретно исполнить метод, нужно иметь в памяти все объекты, которые будут использованы в вызове в качестве аргументов, или затронуты при исполнении, в том числе сами аргументы, их поля, статические классы, глобальные переменные. Для хранения всех объектов в реальном виде существует конкретная память. В конкретной памяти все переменные, которые не являются символьными, хранятся в виде реальных Java-объектов.

В USVM хранение данных происходит в механизме регионов. Регионы поддерживают функции чтения и записи и позволяют хранить символьные данные в контексте различных операций, например, чтение

или запись в поле объекта. Память состоит из набора регионов, каждый из которых отвечает за хранение данных, принадлежащих своей категории значений, и при исполнении инструкций для чтения или записи интерпретатор обращается к памяти, в которой находится соответствующий регион, и происходит чтение или запись.

Реализация конкретной памяти находится в классе `JcConcreteMemory`, который наследован от класса `Memory`. Для хранения объектов внутри конкретной памяти используется класс `JcConcreteMemoryBindings`. В нем находятся отображения символьных адресов физическим объектам и физических адресов символьным адресам. Использование `JcConcreteMemoryBindings` происходит в конкретных регионах, наборе классов, унаследованных от соответствующих регионов и реализующих особенности работы с конкретными значениями. При чтении или записи происходит проверка на конкретность частей операции, после чего объект пишется в `JcConcreteMemoryBindings` или вызывается метод родительского региона.

При вызове метода `read` проверяется конкретность всех частей операции (например, при записи по индексу в массив, тот факт, что ссылка на массив конкретная и индекс тоже конкретный). Если все конкретно, то происходит чтение из `JcConcreteMemoryBindings`, конвертация Java-объекта в выражение, понятное интерпретатору, и возвращение результата. В противном случае вызывается метод родительского региона. При записи объекта тоже проверяется условие на конкретность всех компонентов операции, и происходит запись в `JcConcreteMemoryBindings`. Если интерпретатор записывает в конкретный объект символьное значение, то ссылка на объект помечается как символьная, и теперь операции для работы с ним идут через стандартные регионы.

Одним из главных преимуществ конкретной памяти является возможность осуществить конкретный вызов. Для того чтобы сделать конкретный вызов метода, нужно либо убедиться, что все переменные конкретны, либо конкретизироваться, то есть заменить символьные значения на реальные объекты, используя модель.

3.5. Используемые аппроксимации

Многие методы фреймворка Spring представляют собой сложные алгоритмы, которые обладают достаточно простой семантикой. Например, когда выполняется контроллер, его аргументы берутся из запроса и конвертируются в необходимые типы. Исследовать такой метод символьно не выйдет, так как он породит огромное количество состояний и займет много времени для исследования, а если применить технику конкретного исполнения, то результат будет некорректным: мы будем ожидать символьное значение, отвечающее за пользовательский ввод, а получим конкретное значение. Для того чтобы символьно исследовать код с подобными методами, используются аппроксимации.

В аппроксимациях описаны более эффективные или корректные реализации разных методов с точки зрения символьного исполнения. Существует два вида аппроксимаций: подмена байт-кода конкретного метода или реализация данного метода внутри самого движка. Использование первого метода является более удобным для написания упрощенных реализаций методов с сохранением семантики. Второй метод позволяет взаимодействовать с методами движка и символьным состоянием, но описывать сложные алгоритмы проблематично, так как написать алгоритм в явном виде невозможно.

При исследовании веб-приложений на фреймворке Spring было аппроксимировано множество классов. Ключевая функциональность, которая была аппроксимирована, описана ниже.

- Начало исследования, при котором предотвращается запуск HTTP-сервера и начинается символьное исследование. Приложение анализируется и находятся все контроллеры, по заданным аннотациям контроллеров определяются пути и методы запроса, а после этого создается состояния, которые должны дойти до соответствующих контроллеров, чтобы символьно их проанализировать.

- Обработка входных данных напрямую из запроса, используя вызовы таких методов, как `getHeader`.

- Мокирование методов сервисов и репозиториев.
- Вычисление аргументов контроллера, результат работы этих методов тесно связан с запросом, так как практически всегда данные получаются с использованием информации из запроса.
- Классы, содержащие алгоритмы, которые сложно исследуются символьным движком, например `Model` или `Page`.

3.6. Создание теста из состояния

Результатом символьного анализа является набор состояний, из которых, с помощью SMT-решателя, можно получить модели, используя которые можно получить значения входных данных и результата, и на их основе создать тест. Тест представляет собой класс с аннотацией `@WebMvcTest`. Внутри данного тестового класса объявлено поле типа `MockMvc` с аннотацией `@Autowired`, значение которого используется для обработки запроса, и поля с аннотацией `@MockBean` для мокирования сервисов и репозиториев. Метод теста с аннотацией `@Test` содержит логику, описанную ниже.

1. Создаются и инициализируются необходимые объекты для теста. Например: заполняются массивы, создаются и заполняются объекты, которые будут сериализованы в формат JSON.
2. Вызываются методы фреймворка для мокирования репозиториев и сервисов.
3. С помощью инструментов Spring Boot Test создается запрос, через соответствующие методы проставляются значения заголовков и параметров. Если запрос содержит тело в формате JSON, то добавляется вызов метода `writeValueAsString` у класса `ObjectMapper`, после которого сериализованный объект запишется в тело запроса.
4. Вызывается метод `perform` класса `MockMVC`, который выполняет запрос и получает ответ от сервера.

5. У объекта, который получился в качестве результата, вызываются методы `andExpect`, которые проверяют некоторые свойства ответа, в том числе его содержание, название представления, заголовки.

6. Если в процессе произошло исключение, которое было обработано, то с помощью метода `getResolvedException` получается объект исключения, и используя его вызываются методы `assert`, которые проверяют корректность типа и сообщения исключения.

7. Если результатом исполнения программы стало исключение, которое не было обработано, то вызов метода `perform` оборачивается в лямбда-функцию, и происходит вызов `assertThrows`, который проверяет, что было выброшено корректное исключение.

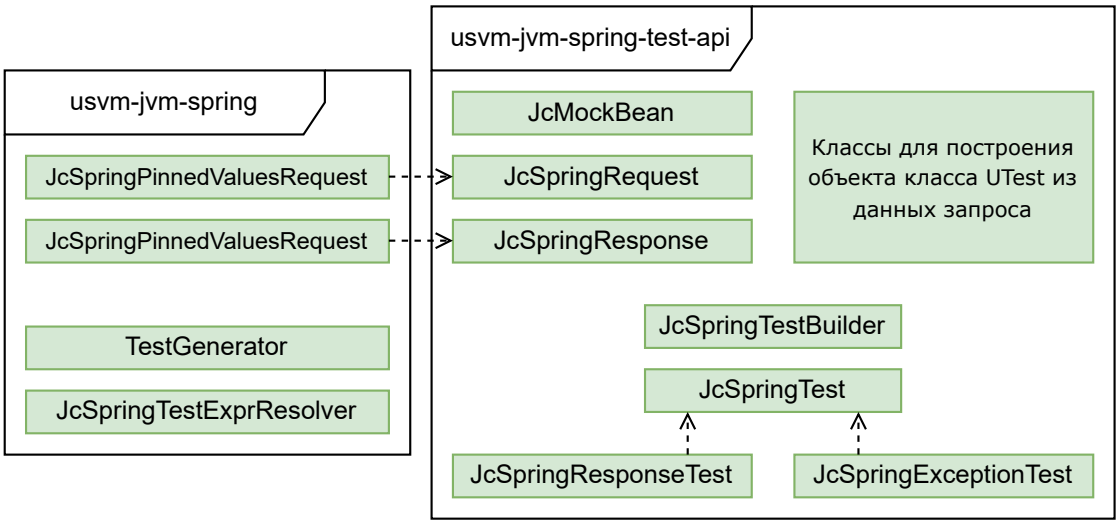

Рис. 6: Архитектура подсистемы для создания тестов

Для создания таких тестов в проекте USVM была создана новая подсистема, архитектура которой показана на Рис. 6. Данная подсистема была реализована в модуле `usvm-jvm-spring-test-api`. Так как в проекте USVM уже имеется формат для созданных тестов — `UTest`, задачей данной подсистемы является формирование объекта класса `Utest` из различных входных данных. Для реализации такой подсистемы был

создан набор классов, представленных на Рис. 6, основные из которых описаны ниже.

- `JcSpringTest` — основной класс, который отвечает за хранение данных о запросе, ответе и результатах мокирования сервисов и репозиториев. Также он отвечает за генерацию объекта класса `UTest`, используя информацию из запроса и ответа и набор классов, в которых реализована вспомогательная логика по построению объекта типа `UTest`.

- `JcSpringResponseTest` и `JcSpringExceptionTest` — наследники класса `JcSpringTest`, которые кроме инициализации контекста, формирования запроса и выполнение обработки запроса позволяют добавить дополнительные проверки на результат. `JcSpringResponseTest` позволяет добавить вызовы методов `andExpect`, которые выполняют проверку корректности ответа. `JcSpringExceptionTest` добавляет различные проверки на выброшенные исключения.

- `JcSpringTestBuilder` — класс, который предоставляет набор инструментов для построения объекта класса `JcSpringTest`.

- `JcMockBean` — класс, который хранит данные об объекте, методы которого были замокированы. Этот класс содержит упорядоченный набор результатов значений вызовов методов, используя который, с помощью инструмента `Mockito`, можно сделать аналогичный мок в коде объекта класса `UTest`.

- `JcSpringRequest` и `JcSpringResponse` — интерфейсы, которые отвечают за получение данных об HTTP-запросе и ответе. Они имеют методы, например `getPath`, `getContent` или `getCookies`, которые используются для получения данных о запросе или ответе. Используя интерфейс `JcSpringRequest`, в `JcSpringTest` создаются вызовы методов для инициализации запроса, а с помощью `JcSpringResponse` создается набор проверок на корректность ре-

зультата. Данные интерфейсы реализованы в проекте `usvm-jvm--spring`.

- Набор классов для построения объекта класса `UTest` — классы которые содержат инструменты, позволяющие удобно добавлять сложные цепочки вызовов методов.

Также, в проекте `usvm-jvm-spring` был создан набор классов, который отвечает за создание теста в формате `UTest` из состояния.

- `JcSpringTestExprResolver` служит для вычисления символьных значений и представления их в виде тестовых выражений для теста в формате `UTest`.

- Реализации интерфейсов `JcSpringRequest` и `JcSpringResponse`, которые берут значения из закрепленных значений в состоянии, а далее, используя объект класса `JcSpringTestExprResolver` транслируют их в тестовые выражения.

- `TestGenerator` содержит набор методов, который использует все классы из модулей `usvm-jvm-spring` и вышеперечисленные классы и позволяет создавать тест напрямую из символьного состояния.

4. Детали реализации

В данной главе описаны детали реализации подсистемы для автоматической генерации интеграционных тестов для веб-приложений. Подробно рассматривается логика, отвечающая за настройку контекста и запуск исследования, обработку входных данных, работу с исключениями и фреймворком Spring Security.

4.1. Конфигурация контекста

Во фреймворке Spring для написания интеграционных тестов веб-приложений используется `MockMvc`. Данный инструмент предоставляет набор методов для тестирования логики обработки запросов без необходимости запуска реального сервера. При проведении символьного анализа приложений `MockMvc` применяется для инициации обработки запроса после завершения конфигурации приложения. Для этого сначала создается и настраивается объект `MockMvc`, после чего для начала обработки запроса вызывается метод `perform`.

Для символьного анализа веб-приложений используются средства тестирования, предоставляемые фреймворком Spring. Такой подход позволяет избежать запуска сервера и проводить анализ приложения напрямую, с возможностью мокирования компонентов, от которых требуется абстрагироваться.

Для запуска веб-приложения в тестовом окружении движок генерирует классы `NewTestSpring` и `StartSpringTestClass`. Класс `StartSpringTestClass` помечен аннотацией `@WebMvcTest`, обеспечивающей создание веб-приложения в тестовом окружении для интеграционного тестирования.

Класс `NewTestSpring` служит точкой входа для исследования. В его методе `startSpring` создается экземпляр `StartSpringTestClass`, после чего с помощью методов `getTestContext` и `getApplicationContext` инициализируется контекст: формируется конвейер обработки запросов и выполняется логика инициализации приложения.

Создание и настройка приложения выполняются в классе `Spring-`

ApplicationImpl. После завершения конфигурации контекста, включающей регистрацию всех компонентов и построение конвейера обработки, в классе SpringApplicationImpl вызывается метод afterRefresh. Для продолжения символьного анализа проекта с использованием аппроксимаций и модификации байт-кода во время вызова метода afterRefresh выполняются следующие действия.

1. В символьном состоянии из стека вызовов удаляются все предыдущие методы. Это позволяет избежать возврата по стеку вызовов в конце исследования, так как вся необходимая информация для формирования теста будет уже получена.

2. Создается и настраивается объект типа MockMvc, который в дальнейшем будет использоваться для выполнения запросов.

3. Определяются все возможные URL-адреса и HTTP-методы, при которых выполняется код контроллера.

4. Для каждого адреса и метода с помощью объекта MockMvc через метод perform инициируется обработка символьного запроса: создается новое состояние и начинается его обработка.

Для того, чтобы найти все интересующие нас при исследовании URL-адреса и HTTP-методы для исполнения кода метода контроллера, веб-приложение пользователя анализируется, и находятся все классы, которые являются наследниками Controller или RestController. Далее, алгоритм находит все методы контроллеров, извлекает из их аннотаций данные о URL-адресах и HTTP-методах и сохраняет эти данные. Например, если в контроллере аннотация @RequestMapping(''api''), и в его методе аннотация @GetMapping(''/hello''), то в множество путей и методов будет добавлена пара ''api/hello'', ''GET''. При необходимости на найденные пути применяется фильтр, а переменные пути, обозначенные в фигурных скобках, заменяются на стандартное значение для данного типа. Это нужно для того, чтобы строка была конкретной при поиске контроллера, а позже, при его анализе, методы, которые могут

вернуть переменную из пути, будут аппроксимированы и вернут символьную переменную.

Для определения всех URL-адресов и HTTP-методов, необходимых для выполнения кода методов контроллера, проводится анализ пользовательского веб-приложения, который имеет следующие этапы.

1. Выявляются все классы, являющиеся наследниками `Controller` или `RestController`
2. Для каждого контроллера находятся все его методы
3. Из аннотаций методов извлекаются данные о URL-адресах и HTTP-методах
4. Полученные данные сохраняются

При необходимости к найденным путям применяется фильтрация, а переменные пути (обозначенные фигурными скобками) заменяются стандартными значениями соответствующих типов. Это позволяет использовать конкретные строки при поиске обработчика запроса.

4.2. Работа с входными данными

В системе существует множество классов, ответственных за получение входных данных. Для реализации ключевой концепции нашего алгоритма с использованием модификации байт-кода методы этих классов, отвечающие за получение входных параметров, аппроксимируются. Далее рассмотрены методы и классы, отвечающие за получение входных данных из запроса, и приведены особенности их аппроксимаций.

4.2.1. Аргументы методов контроллера

Обработчики запросов в Spring могут содержать параметры с различными аннотациями. При вызове такого метода фреймворк Spring автоматически создает соответствующие аргументы на основе данных

запроса. За эту функциональность отвечают классы, реализующие интерфейс `HandlerMethodArgumentResolver`.

Среди тех классов, на которые были написаны аппроксимации, были классы, которые работают с параметрами, заголовками, переменными пути. При вызове метода, который отвечает за получение данных из запроса и конвертацию в нужный формат, нам известно название аргумента, а также тип, в который результат нужно конвертировать. Используя эти данные и API символьного движка, создается и возвращается символьное значение, которое также записывается в закрепленные значения для дальнейшего использования при создании теста.

4.2.2. Поддержка JSON

Помимо работы с примитивными типами данных, фреймворк Spring предоставляет удобные механизмы для работы со сложными объектами. В частности, поддерживается передача объектов через тело запроса в формате JSON.

По умолчанию Spring использует библиотеку `fasterxml.jackson` для обработки JSON. Разработчик может применять аннотации для наложения ограничений на поля объектов или задания значений по умолчанию. Также с помощью специальных аннотаций можно определить конструктор или фабричный метод, который будет использоваться для создания объекта из JSON-данных, переданных в теле запроса.

Для корректной обработки возможностей библиотеки работы с JSON было принято решение реализовать аппроксимацию методов обработки аргументов контроллера особым образом, отличающимся от подхода, применённого к другим классам обработки параметров запроса.

Алгоритм создания символьных объектов для JSON-данных реализован через модификацию процесса десериализации. Вместо создания символьного объекта при вызове метода обработки запроса, для создания символьного значения используются специально аппроксимированные классы десериализатора.

Данная аппроксимация позволяет создавать символьные объекты, которые соответствуют всем ограничениям, заданным аннотациями, а

также позволяет исполнять код конструктора или фабричного метода. Такой результат достигается за счет использования реальных механизмов десериализации.

Особенностью работы алгоритма является дополнительная устойчивость к мутациям, примененных к результату работы. Для этого создаются и заполняются два объекта одновременно, а при записи примитивных свойств в оба записывается одно и то же символьное значение. В конце исполнения одна копия записывается в закрепленные значения, а другая возвращается как результат.

4.2.3. Привязка модели

Для передачи сложных объектов также может использоваться механизм привязки модели. Привязка модели представляет собой процесс автоматического заполнения объекта данными из HTTP-запроса. Например, если метод контроллера содержит параметр с аннотацией `@ModelAttribute` и для этого параметра разрешена привязка модели, то объект будет создан, а его свойства заполнены через параметры запроса, данные формы или переменные пути.

Согласно конвенции [2], для привязки модели используется следующий формат.

- `name` — привязка значения к свойству `name`
- `account.name` — привязка к вложенному свойству `name` объекта `account`
- `accounts[2]` — привязка к третьему элементу коллекции `accounts`
- `accounts[KEY]` — привязка к элементу ассоциативного массива `accounts` с ключом `KEY`

Нашей задачей являлось написать символьную аппроксимацию на метод привязки модели так, чтобы в свойства объекта были записаны символьные значения, а также чтобы эти значения были записаны в закрепленные значения. Также необходимо учесть код пользователя,

запрещающий или разрешающий привязывать некоторые значения модели.

Для реализации данной функциональности была разработана аппроксимация метода `bind` класса `ServletRequestDataBinder`, отвечающего за основную логику привязки данных. Применённый алгоритм аналогичен подходу, использованному при реализации десериализации, с максимальным использованием доступных объектов из базового класса. Алгоритм рекурсивно обрабатывает объекты и коллекции, записывая символьные значения для полей примитивных типов. В отличие от механизма JSON-десериализации, данный подход не только создаёт символьные значения, но и сохраняет их в закреплённых значениях с ключами, соответствующими полным путям до свойств в соответствии с конвенциями Spring.

4.3. Обработка исключений

При обработке запроса могут возникать исключения двух основных типов, значимых для нашего исследования: обработанные и необработанные.

Отличие обработанных исключений заключается в том, что фреймворк Spring выполняет специальную логику их обработки. В таких случаях вызов метода `perform` класса `MockMvc` не завершается с ошибкой, а возвращает результат, содержащий исключение в специальном поле. Например, если класс исключения помечен аннотацией `@ResponseStatus`, фреймворк обрабатывает это исключение — метод `perform` возвращает HTTP-ответ с указанным статусом, а выброшенное исключение сохраняется в соответствующем поле результата.

В отличие от обработанных исключений, необработанные исключения не перехватываются фреймворком Spring, что приводит к завершению работы метода `perform` класса `MockMvc` с выбросом исключения.

Наш алгоритм поддерживает оба типа исключений. Для выявления необработанных исключений вызов метода `perform` в аппроксимации начального исследования оборачивается в блок `try-catch`, где перехва-

ченное исключение сохраняется в закрепленных значениях. Для работы с обработанными исключениями используется метод `getResolvedException`.

При генерации тестов для необработанных исключений создается вызов `assertThrows`, принимающий лямбда-функцию с вызовом `perform`. Это позволяет убедиться в факте возникновения исключения и проверить соответствие типа исключения ожидаемому. Для обработанных исключений проверяются тип и текст сообщения, если оно было задано.

4.4. Поддержка Spring Security

Фреймворк Spring Security фактически стал стандартом для обеспечения безопасности веб-приложений, построенных на Spring[25]. Он предоставляет комплекс средств защиты приложения, а также удобный механизм управления пользователями, их ролями и персональными данными.

Поскольку Spring Security широко применяется в веб-приложениях, его поддержка необходима и для символьного анализа. Было реализовано два варианта работы с Spring Security: отключение конфигураций и фильтров или полноценное символьное исследование.

При реализации подхода с отключением конфигураций Spring Security в классе `StartSpringTestClass` модифицируются параметры аннотации `@WebMvcTest`. В аннотацию добавляются следующие параметры.

- Параметр `excludeFilters` — отключает все фильтры Spring Security через механизм `@ComponentScan`, исключая классы-наследники `WebSecurityConfigurer`
- Параметр `excludeAutoConfiguration` — исключает классы автоматической конфигурации.

Данный подход обеспечивает возможность исследования веб-приложения без фильтров безопасности и является более простым, по срав-

нению с полной поддержкой Spring Security.

Альтернативный подход предполагает символьное выполнение методов Spring Security. В этом случае вместо отключения фильтров и конфигураций безопасности выполняются следующие действия.

- В проекте идентифицируются конфигурационные классы с аннотацией `@EnableWebSecurity`.
- Найденные конфигурации подключаются с использованием аннотации `@Include`.

Перед вызовом метода `perform` движком создается пользователь с символьными ролями и учетными данными. Используя методы `with` и `user` он добавляется в запрос.

Данный способ позволяет работать с пользователем и его ролями в обработчике запроса, а также создавать тесты, которые могут проверять корректность доступа. Однако такой подход сложнее реализовать, поэтому при исследовании присутствует вероятность ошибки.

4.5. Воспроизведение и отрисовка тестов

Результатом символьного анализа веб-приложения является интеграционный тест. В данной подглаве описан процесс генерации теста в формате `UTest`, а также некоторые особенности преобразования теста в Java-код.

4.5.1. Формат UTest

Формат хранения тестов в движке USVM реализован через специальный класс `UTest`. С точки зрения данного формата, тест представляет собой набор инструкций инициализации (`initStatements`) и вызов тестируемого метода (`callMethodExpression`).

Для представления различных инструкций и выражений существуют специальные классы: Класс `UTestStatement` и `UTestExpression`. `UTestStatement` используется для представления инструкции. Возможные инструкции и их классы перечислены ниже.

- `UTestSetFieldStatement` и `UTestSetStaticFieldStatement` представляют инструкции по записи поля или статического поля.
- `UTestArraySetStatement` представляет инструкцию, которая записывает объект по индексу в массив.
- `UTestBinaryConditionStatement` представляет ветвление с условием, являющимся бинарным оператором. Данная инструкция позволяет проверить условие и исполнить указанный набор инструкций в зависимости от выполнения условия.

Абстрактный `UTestExpression` используется для представления выражения. Его реализации для представления различных выражений приведены ниже.

- `UTestMockObject` — мокирование объекта. Позволяет переопределить результаты вызовов методов, а также значения полей.
- `UTestGlobalMock` — мокирование всех объектов заданного типа.
- `UTestMethodCall` — вызов метода.
- `UTestStaticMethodCall` — вызов статического метода.
- `UTestConstructorCall` — вызов конструктора.
- `UTestAllocateMemoryCall` — аллокация пустого объекта заданного типа.
- `UTestBinaryConditionExpression` — аналог условного оператора, который позволяет выполнить условие, и в зависимости от результата вернуть одно из двух выражений.
- `UTestArithmeticExpression` — выполнение арифметической операции.
- `UTestGetStaticFieldExpression` — чтение статического поля.

- `UTestConstExpression` — представление константы, в том числе значения `null`.
- `UTestGetFieldExpression` — чтение поля объекта.
- `UTestArrayLengthExpression` — чтение длины массива.
- `UTestArrayGetExpression` — чтение элемента массива по индексу.
- `UTestCreateArrayExpression` — создание массива.
- `UTestCastExpression` — приведение объекта к заданному типу.
- `UTestClassExpression` — константа, которая имеет тип `Class<?>`.

В рамках данной работы в множество выражений были добавлены `UTestAssertThrowsCall` и `UTestAssertEqualsCall`. `UTestAssertThrowsCall` имеет поле с выражением, которое является вызовом метода, и проверяет, что при вычислении данного метода произойдет исключение.

4.5.2. Воспроизведение тестов

При написании интеграционного теста пользователь не реализует логику запуска, настройки контекста и инициализации полей тестового класса — эти задачи выполняет тестовый фреймворк. Однако при работе с тестами в формате UTest необходимо явно вызывать соответствующие функции фреймворка.

Для инициализации обработки запроса в тест включается специальный набор инструкций, перечисленный ниже.

1. Создание объекта класса `TestContextManager` — основной точки входа тестового фреймворка, отвечающей за инициализацию контекста. В конструктор передается сгенерированный движком класс `StartSpringTestClass`, дополненный аннотациями для корректной инициализации контекста и полей тестового класса.
2. Создание объекта класса `StartSpringTestClass`.

3. Вызов метода `prepareTestInstance` у объекта `TestContextManager` с передачей созданного тестового класса. Этот метод выполняет инициализацию контекста и внедрение зависимостей, включая инициализацию поля `mockMvc` с аннотацией `@Autowired` тестового класса.

4. Получение поля `mockMvc` из объекта `StartSpringTestClass`.

После выполнения указанных инструкций тест будет содержать инициализированный объект класса `MockMvc` и настроенный контекст приложения. Для обработки запроса достаточно вызвать метод `perform` у созданного объекта `MockMvc`. Затем с помощью тестовых выражений описывается оставшаяся логика теста.

4.5.3. Отрисовка тестов

Преобразование тестов из формата `UTest` уже реализовано в USVM, однако рендеринг интеграционных тестов для веб-приложений имеет специфические особенности. Прямое преобразование всех инструкций и выражений из сгенерированного `UTest` приведет к неработоспособному тесту, поскольку в тестовом проекте пользователя отсутствуют классы, сгенерированные символьным движком.

Основной идеей для решения данной задачи является преобразование теста так, чтобы метод, содержащий логику теста, находился в специальном классе с необходимыми аннотациями и полями, аналогичными тем, которые использовались при исследовании. Обращения к полям созданного `StartSpringTestClass` должны быть преобразованы в обращения к полям нового класса, а инструкции по инициализации контекста должны быть удалены.

Для решения данной задачи был создан набор классов, реализующий логику отрисовки с учетом особенностей подхода, а также создан механизм трансформеров, который отвечает за преобразование изначального теста в более оптимальный и корректный формат. Действия, которые происходят, чтобы преобразовать тест в формате `UTest` в более оптимальное представление, перечислены ниже.

1. Обращение к полям объекта типа `StartSpringTestClass` заменяется на обращения к полям в классе, в котором создается метод. При этом, если необходимого поля нет, то оно создается, используя информацию из поля прошлого тестового класса. Данная логика реализована в классе `JcSpringMvcTestBlockRenderer`.

2. В новый класс добавляются все аннотации из прошлого тестового класса. Данная логика реализована в классе `JcSpringMvcTestClassRenderer`.

3. Вызов метода `prepareTestInstance` удаляется из нового теста. Эта логика реализована в классе `JcSpringMvcTestTransformer`.

5. Тестирование

В данной главе описан процесс тестирования подсистемы, выбор и описание проектов для тестирования и сравнение с аналогами. Для тестирования получившейся подсистемы были выбраны веб-приложения на фреймворке Spring, для которых инструменту необходимо было сгенерировать интеграционные тесты.

Проект `spring-petclinic`[8] — это эталонное веб-приложение, разработанное командой Spring Framework в качестве демонстрационного проекта. Он реализует систему учета пациентов ветеринарной клиники, включая работу с владельцами, питомцами и визитами, и использует классическую архитектуру MVC с Hibernate [15] для взаимодействия с реляционной базой данных, что делает его репрезентативным тестовым примером.

Синтетические бенчмарки[9] — это специализированное веб-приложение, созданное для валидации функциональности разработанного решения. Оно охватывает следующие ключевые аспекты.

- Примитивные типы данных с различными аннотациями в параметрах методов контроллеров, служащие для проверки корректности аппроксимаций, отвечающих за получение входных данных из переменных пути и параметров и заголовков запроса.
- Сложные типы данных для проверки механизмов привязки модели и десериализации JSON.
- Взаимодействие с моделью данных.
- Обработка различных методов HTTP-запроса, таких как получение таблицы всех заголовков.
- Обработка исключений.
- Работа с `HttpEntity`.

[8] https://spring-petclinic.github.io/

[9] https://github.com/arthur100500/usvm-spring-benchmarks

- Мокирование сервисов и репозиториев.

Для тестирования проектов различными инструментами в них были внесены изменения, улучшающие процесс исследования. В оба проекта была добавлена библиотека для генерации спецификации OpenAPI, необходимой для работы фаззеров, использующих метод чёрного ящика.

В синтетическом бенчмарке некоторые обработчики, хотя и поддерживались Spring, не соответствовали спецификации RFC 7231 [16]. Например, они допускали передачу данных в теле GET-запроса. Для формирования корректной спецификации OpenAPI эти методы переработали в соответствии со стандартом.

Поскольку многие инструменты фаззинга работают с REST API, для автоматической генерации интеграционных тестов в `spring-pet-clinic` заменили стандартные контроллеры на REST-контроллеры. Вместо возврата отрисованного представления они стали возвращать только его название в виде строки. Это изменение не влияет на результаты тестирования, так как проверка корректности отрисовки представлений не входит в задачи тестирования.

5.1. Инструменты, выбранные для сравнения

Для сравнения реализованной подсистемы с аналогами были выбраны два инструмента, реализующие различные алгоритмы фаззинга.

В качестве инструмента для тестирования методом чёрного ящика с использованием OpenAPI-спецификации был выбран EvoMaster. Этот выбор обусловлен результатами независимых исследований [20], подтверждающими его эффективность, а также возможностью генерации JUnit-тестов с использованием библиотеки RestAssured, что упрощает работу с тестами и измерение покрытия.

Для генерации тестов инструмент EvoMaster требует OpenAPI-спецификацию. Она была создана с помощью библиотеки `springdoc-openapi-starter-webmvc-ui`, автоматически предоставляющей спецификацию по адресу `/v3/api-docs` [14].

Процесс запуска фаззинга через интерфейс командной строки включает следующие параметры.

- `--blackBox true` — выбор режима черного ящика
- `--outputFormat JAVA_JUNIT_5` — формат генерации тестов
- `--problemType REST` — тестирование REST API
- `--bbSwaggerUrl` — установка URL OpenAPI-спецификации
- `--maxTime` — создание временного ограничения

Результатом работы становится директория `generated_tests`, в которой инструмент создает Java-файлы, содержащие созданные интеграционные тесты.

В качестве инструмента, который реализует технику фаззинга белого ящика, был выбран инструмент UnitTestBot Java. Данный продукт позволяет автоматически создавать интеграционные тесты на веб-приложения на основе фреймворка Spring с помощью фаззинга без дополнительной ручной инструментации.

Для того, чтобы использовать данный инструмент, необходимо установить среду разработки IntelliJ IDEA. Далее, используя встроенный менеджер плагинов или официальный GitHub-репозиторий, требуется установить UnitTestBot Java.

Для создания набора интеграционных тестов пользователю нужно выполнить следующие шаги.

1. Выделенить тестируемые контроллеры в среде разработки
2. Вызвать контекстного меню через горячие клавиши или графический интерфейс
3. Указать тестовый проект, который будет дополнен созданными тестами
4. Выбрать Spring-конфигурацию

5. Установить временной лимит для тестирования

Результат работы — сгенерированные файлы, содержащие интеграционные тесты в указанном тестовом проекте.

5.2. Результаты тестирования

Тестирование проводилось на системе, которая имеет следующие характеристики.

- Операционная система — Microsoft Windows 10.
- Процессор — AMD Ryzen 5 5600X, 3.7 ГГц.
- Объем оперативной памяти — 32 Гб.

Временное ограничение было установлено исходя из практического требования к скорости генерации тестового набора, предусматривающего создание тестов для одного контроллера в течение одной минуты. Данный временной лимит был выбран на основе компромисса между необходимостью обеспечения достаточного времени для полноценного анализа контроллера и требованием оперативности процесса тестирования. Для проекта `spring-petclinic` ограничение составило 6 минут, для синтетических бенчмарков — 7 минут.

Для измерения покрытия кода были использованы встроенные возможности среды разработки IntelliJ IDEA. Процедура измерения включала запуск набора тестовых сценариев, сгенерированного инструментом, а далее анализ полученного покрытия с выделением исключительно методов контроллеров.

В процессе тестирования собиралась статистика по воспроизводимости тестов. Тест считается воспроизводимым, если его результат совпадает с ожидаемым.

При выполнении тестов, сгенерированных различными инструментами, часть из них завершалась неудачно. В частности, тесты от UnitTestBot Java и EvoMaster намеренно включали сценарии с необработанными исключениями, где успешное выполнение теста подразу-

мевало генерацию исключения. UnitTestBot Java помечает такие тесты специальными комментариями, указывающими ожидаемое исключение. EvoMaster группирует аналогичные тесты в отдельном файле `EvoMaster_faults_Test.java`. Тесты, приводящие к ошибкам выполнения и не имеющие соответствующей маркировки, считаются невоспроизводимыми.

Разработанная подсистема обрабатывает тесты с исключениями с помощью метода `assertThrows`, поэтому заранее ошибочных сценариев не создается. К невоспроизводимым относятся тесты, которые не прошли валидацию при создании или привели к неожиданному ошибочному поведению во время выполнения.

Результаты тестирования приведены в таблицах 1 и 2.

Инструмент	Покрытие, %	Кол-во тестов	Воспроизводимость, %
EvoMaster	46	50	90
UnitTestBot Java	40	55	92
USVM	70	267	87

Таблица 1: Результаты тестирования синтетических бенчмарков

Инструмент	Покрытие, %	Кол-во тестов	Воспроизводимость, %
EvoMaster	23	26	100
UnitTestBot Java	46	63	98
USVM	61	64	78

Таблица 2: Результаты тестирования проекта `spring-petclinic`

5.3. Анализ результатов тестирования

На основе результатов тестирования можно утверждать, что реализованная в данной работе система демонстрирует результаты, сопоставимые и даже превосходящие рассмотренные решения по автоматической генерации интеграционных тестов, использующие фаззинг методом черного и серого ящиков.

Такой результат ожидаем. В частности, в проекте с синтетическими бенчмарками практически во всех методах присутствует ветвление, при котором входные данные сравниваются с определенным значением. Для фаззинга, использующего метод черного или серого ящика, без детального анализа кода сложно подобрать входные данные, позволяющие покрыть такие ветки. Также, фаззерам для достижения значимых результатов требуется гораздо больше времени, чем выделенное в тестах.

Тем не менее, несмотря на полученные результаты, разработанная система имеет ряд ограничений.

- Поддерживаются только некоторые версии библиотек и фреймворков. В текущей реализации работает только Spring версии выше 3.0.1.

- Требуется добавлять новые аппроксимации и улучшать движок для проектов, которые еще не исследовались. Spring содержит множество классов и методов, которые необходимо упрощать без изменения семантики. Для тестируемых проектов большинство аппроксимаций уже написано и проверено, однако анализ новых приложений может быть затруднителен.

- Проблемы символьного исполнения, такие как экспоненциальный рост числа состояний или работа с внешними вызовами, решены лишь частично и только для методов Spring. Например, если в тестируемом коде присутствует бесконечный цикл, то может сильно увеличиться количество состояний, а следственно — время исследования.

- Из-за неточностей в реализации аппроксимаций и недоработок инструмента доля неудачно воспроизводящихся тестов оказывается выше, чем при использовании фаззеров.

6. Заключение

В ходе работы были получены следующие результаты.

1. Был выполнен обзор используемых подходов и символьного движка USVM, а также существующих решений в области генерации тестов, а именно — UnitTestBot Java, RESTler и EvoMaster.

2. Был разработан алгоритм для символьного анализа веб-приложений, созданных на основе фреймворка Spring, и спроектирована архитектура подсистемы, реализующей данный алгоритм в рамках проекта USVM.

3. Подсистема была реализована в USVM с использованием языков программирования Kotlin и Java.

4. Было проведено тестирование реализованного решения и выполнено сравнение с существующими решениями.

Код реализованной подсистемы находится в GitHub-репозиториях USVM[10] и `java-stdlib-approximations`[11].

[10]https://github.com/arthur100500/usvm

[11]https://github.com/arthur100500/java-stdlib-approximations

Список литературы

[1] Atlidakis Vaggelis, Godefroid Patrice, Polishchuk Marina. Restler: Stateful rest api fuzzing // 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) / IEEE. — 2019. — P. 748–758.

[2] Broadcom. — Beans Conventions. — URL: https://docs.spring.io/spring-framework/reference/core/validation/beans-beans.html#beans-beans-conventions (дата обращения: 2025-19-05).

[3] Breuer Peter T, Pickin Simon. Symbolic approximation: an approach to verification in the large // Innovations in Systems and Software Engineering. — 2006. — Vol. 2, no. 3. — P. 147–163.

[4] Broadcom. Why Spring. — URL: https://spring.io/why-spring (дата обращения: 2024-12-22).

[5] Broadcom. Документация Spring. Фильтры. — URL: https://docs.spring.io/spring-framework/reference/web/webmvc/filters.html (дата обращения: 2024-12-29).

[6] Google. — ClusterFuzz. — URL: https://google.github.io/clusterfuzz/ (дата обращения: 2024-12-29).

[7] Coppa Emilio, Izzillo Alessio. Testing concolic execution through consistency checks // Journal of Systems and Software. — 2024. — Vol. 211. — P. 112001.

[8] Coppa Emilio, Yin Heng, Demetrescu Camil. Symfusion: hybrid instrumentation for concolic execution // Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. — 2022. — P. 1–12.

[9] De Moura Leonardo, Bjørner Nikolaj. Z3: An efficient SMT solver // International conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 337–340.

[10] De Moura Leonardo, Bjørner Nikolaj. Z3: An efficient SMT solver // International conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 337–340.

[11] Enhancing symbolic execution by machine learning based solver selection / Sheng-Han Wen, Wei-Loon Mow, Wei-Ning Chen et al. // Proceedings of the NDSS Workshop on Binary Analysis Research. — 2019.

[12] Foundation The Linux. Спецификация OpenAPI. — URL: https://www.openapis.org/ (дата обращения: 2025-05-25).

[13] Fuzzing: State of the art / Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia et al. // IEEE Transactions on Reliability. — 2018. — Vol. 67, no. 3. — P. 1199–1218.

[14] Generating Accurate OpenAPI Descriptions from Java Source Code / Alexander Lercher, Christian Macho, Clemens Bauer, Martin Pinzger // arXiv preprint arXiv:2410.23873. — 2024.

[15] Hat Red. Hibernate. — URL: https://hibernate.org/ (дата обращения: 2025-05-25).

[16] IETF. Стандарт RFC 7231. — URL: https://datatracker.ietf.org/doc/html/rfc7231 (дата обращения: 2025-05-25).

[17] JetBrains. Опрос Developer Ecosystem. — URL: https://www.jetbrains.com/lp/devecosystem-2023/java/ (дата обращения: 2024-12-22).

[18] King James C. Symbolic execution and program testing // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385–394.

[19] SBST tool competition 2022 / Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, Fiorella Zampetti // Proceedings of the 15th Workshop on Search-Based Software Testing. — 2022. — P. 25–32.

[20] Sartaj Hassan, Ali Shaukat, Gjøby Julie Marie. REST API Testing in DevOps: A Study on an Evolving Healthcare IoT Application // arXiv preprint arXiv:2410.12547. — 2024.

[21] Selective symbolic execution / Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea // Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep). — 2009.

[22] Sen Koushik, Agha Gul. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools: (Tool Paper) // Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18 / Springer. — 2006. — P. 419–423.

[23] Sen Koushik, Marinov Darko, Agha Gul. CUTE: A concolic unit testing engine for C // ACM SIGSOFT software engineering notes. — 2005. — Vol. 30, no. 5. — P. 263–272.

[24] Sethu Ramesh. Systems and methods to reverse engineer code to models using program analysis and symbolic execution. — 2018. — 22. — US Patent App. 15/268,011.

[25] Broadcom. — Spring Security. — URL: https://spring.io/projects/spring-security (дата обращения: 2025-19-05).

[26] State of the art: Dynamic symbolic execution for automated test generation / Ting Chen, Xiao-song Zhang, Shi-ze Guo et al. // Future Generation Computer Systems. — 2013. — Vol. 29, no. 7. — P. 1758–1773.

[27] Sydr: Cutting edge dynamic symbolic execution / Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts et al. // 2020 Ivannikov ISPRAS Open Conference (ISPRAS) / IEEE. — 2020. — P. 46–54.

[28] Symbolic execution and recent applications to worst-case execution, load testing and security analysis / Corina S Pasareanu, Rody Ker-

sten, L Kasper, Quoc-Sang Phan // Advances in Computers. — 2018. — Vol. 22.

[29] Tillmann Nikolai, De Halleux Jonathan. Pex – white box test generation for .NET // International conference on tests and proofs / Springer. — 2008. — P. 134–153.

[30] UnitTestBot. — UnitTestBot. — URL: https://www.utbot.org/ (дата обращения: 2025-22-04).

[31] Utbot java at the sbst2022 tool competition / Dmitry Ivanov, Alexey Menshutin, Denis Fokin et al. // Proceedings of the 15th Workshop on Search-Based Software Testing. — 2022. — P. 39–40.

[32] Veritesting challenges in symbolic execution of Java / Vaibhav Sharma, Michael W Whalen, Stephen McCamant, Willem Visser // ACM SIGSOFT Software Engineering Notes. — 2018. — Vol. 42, no. 4. — P. 1–5.

[33] Verma Akanksha, Khatana Amita, Chaudhary Sarika. A comparative study of black box testing and white box testing // International Journal of Computer Sciences and Engineering. — 2017. — Vol. 5, no. 12. — P. 301–304.

[34] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.

[35] The JUnit Team. — Фреймворк JUnit. — URL: https://junit.org/junit5/ (дата обращения: 2025-19-05).