

Санкт-Петербургский государственный университет

СТАРЦЕВ Матвей Сергеевич

Выпускная квалификационная работа

Символьный анализ приложений с реляционными базами данных

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

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

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

Консультант:
ведущий разработчик, ООО «ИИнтелКод», Костицин М. П.

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

Санкт-Петербург
2025

Saint Petersburg State University

Matvei Startsev

Bachelor's Thesis

Symbolic analysis of applications with relational databases

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2021 "Software Engineering"*

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

Consultant:
lead developer 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	4
1. Постановка задачи	6
2. Обзор	7
2.1. Метод символьного исполнения	7
2.2. Инструмент USVM	13
2.3. Методы анализа реляционных баз данных	16
2.4. Фреймворк SPRING JPA	18
3. Символьная модель для работы с базами данных	19
3.1. Структура модели баз данных	21
3.2. Парсинг запросов	26
3.3. Процесс символьного исполнения	29
4. Особенности реализации	33
4.1. Механизм создания кода	33
4.2. Создание глобальных полей таблиц	35
4.3. Валидация значений базы данных	36
4.4. Реализация классов-обёрток	37
4.5. Модификация data-классов	38
5. Тестирование и эксперименты	41
5.1. Способ тестирования модели базы данных	41
5.2. Сравнение результатов	42
Заключение	44
Список литературы	45

Введение

Тестирование — один из важнейших этапов разработки программного обеспечения. Большинство разработчиков стремится обеспечить покрытие кода тестами, чтобы в дальнейшем гарантировать корректность работы отдельных его компонентов. Однако создание тестов требует значительных затрат, сопоставимых с непосредственной разработкой кода системы [12]. Кроме того, программист может упустить из виду некоторые ветви исполнения программы или не учесть определённые сценарии её работы.

Для решения этих проблем существуют инструменты автоматической генерации тестов, использующие различные методы анализа исходного кода. Одним из таких методов является символьное исполнение [8, 9, 11, 21] — техника, позволяющая находить все достижимые пути исполнения программы и определять входные данные, необходимые для их прохождения. Однако у этого метода есть ограничения: он может работать медленно или не поддерживать анализ внешних систем, с которыми взаимодействует тестируемая программа [13, 24]. Одно из решений этих трудностей — использование аппроксимаций — замены сложных фрагментов кода более простыми [4, 23].

Данная работа реализована в рамках инструмента статического анализа USVM¹, разрабатываемого командой UTBot. В этом проекте ведётся работа по поддержке фреймворка SPRING для JAVA, который является самым популярным в JAVA-сообществе² и включает ORM-фреймворк SPRING JPA для работы с реляционными базами данных.

При анализе кода, использующего SPRING JPA, помимо общих проблем символьного исполнения, возникает дополнительная сложность: отсутствие реализаций методов для запросов. Кроме того, SPRING JPA активно взаимодействует с базой данных, записывая и читая данные из памяти, что традиционно представляет трудность для символьного исполнения [1, 10, 14]. Решение этих проблем возможно путём замены

¹<https://github.com/UnitTestBot/usvm> (дата обращения: 19 мая 2025 г.)

²<https://www.jetbrains.com/lp/devecosystem-2023/java> (дата обращения: 19 мая 2025 г.)

вызовов SPRING JPA на взаимодействие с символьной моделью реляционной базы данных.

Таким образом, актуальной становится задача разработки подсистемы моделирования реляционной базы данных для символьного исполнения.

1. Постановка задачи

Целью работы является разработка производительной подсистемы моделирования реляционных баз данных для символьного исполнения, ориентированной на JAVA-приложения, использующие ORM-фреймворк SPRING JPA. Для достижения этой цели были поставлены следующие задачи:

1. провести анализ существующих методов анализа кода, взаимодействующего с реляционными базами данных, изучить особенности SPRING JPA и возможности инструмента символьного исполнения USVM;
2. разработать символьную модель для работы с реляционными базами данных;
3. реализовать предложенную модель в рамках проекта USVM;
4. провести апробацию созданной и реализованной символьной модели.

2. Обзор

В данной главе представлен обзор техники символьного исполнения, методов статического анализа программ, работающих с реляционными базами данных, возможностей ORM-фреймворка SPRING JPA и инструмента для генерации тестов USVM.

2.1. Метод символьного исполнения

Метод символьного исполнения [5, 15] представляет собой технику анализа исходного кода, которая позволяет находить все возможные пути исполнения программы и соответствующие наборы входных данных.

В отличие от конкретного исполнения программы на реальных входных данных, символьное исполнение оперирует символьными значениями, описывающими переменные и память программы. Процесс начинается с создания символьных значений, заменяющих все входные данные программы. Затем символьная машина — анализатор программы, строит граф потока управления, содержащий все возможные пути выполнения. Для каждого пути анализатор выполняет вычисления над символьными значениями в соответствии с инструкциями программы.

Ключевым понятием символьного исполнения является символьное состояние, описывающее состояние программы в процессе анализа. Оно включает:

- указатель на текущую исследуемую инструкцию;
- ограничения условия пути π — логическую формулу, содержащую символьные значения и описывающую условия на входные данные, необходимые для достижения данной точки;
- отображение имён переменных программы в соответствующие символьные значения.

Символьная машина анализирует инструкции программы, создавая новые состояния на основе предыдущих. Для каждого состояния обнов-

ляется: текущая инструкция, условия пути и состояние памяти. При встрече условного оператора, вызывающего ветвление исполнения, из одного состояния создаётся несколько, каждое со своими ограничениями пути. Таким образом строится дерево состояний.

После завершения символьного исполнения по всем возможным путям, полученную информацию можно использовать для: выявления потенциальных ошибок, генерации тестов с высоким покрытием. Например, взяв π из листа дерева и проверив его выполнимость с помощью SMT-решателя, можно получить конкретные входные данные для теста, покрывающей соответствующую ветвь исполнения. Ожидаемый результат при этом выражается через символьные переменные, которые конкретизируются тем же решением.

2.1.1. Пример символьного анализа

```
bool target(int x, int y) {  
    x = 4;  
    if (x > y * 2) return true;  
    if (x < y) return false;  
    error();  
}
```

Листинг 1: Исходный код исследуемой символьно функции target.

Рассмотрим символьный анализ функции `target` (листинг 1). Изначально аргументам функции `x` и `y` присваиваются символьные значения α и β соответственно, а условие пути π инициализируется как `true`. Последовательно выполняя инструкции, построим дерево состояний (рис. 1):

- после первой инструкции `x = 4` создаётся новое состояние, где в символьной памяти переменной `x` присваивается конкретное значение 4;

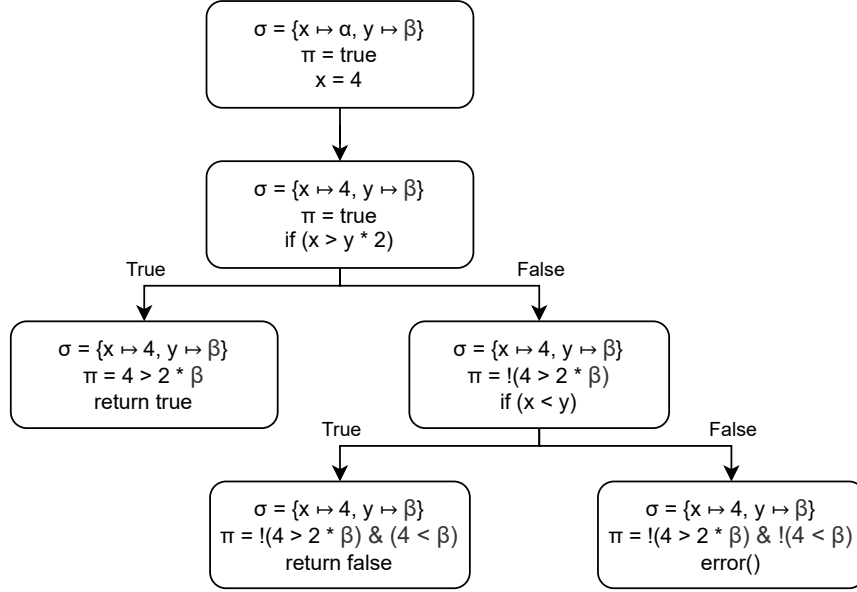


Рис. 1: Дерево состояний для функции **target**.

- При выполнении условия **if** ($x > y * 2$) происходит ветвление на два состояния:
 - ветка **then** с условием $\pi = (4 > \beta * 2)$. После выполнения инструкции **return true** анализ этой ветки завершается;
 - ветка **else** с условием $\pi = \neg(4 > \beta * 2)$.
- ветка **else** порождает ещё два состояния при анализе следующего условия **if** ($x < y$):
 - ветка **then** с комбинированным условием $\pi = \neg(4 > \beta * 2) \wedge (4 < \beta)$. Функция возвращает **false**;
 - ветка **else** с условием $\pi = \neg(4 > \beta * 2) \wedge \neg(4 < \beta)$, приводящим к выполнению **error()**.

Три конечных состояния полностью описывают возможное поведение функции. Для генерации тестовых случаев необходимо найти конкретные значения переменных, удовлетворяющие условиям каждого пути. Эта задача решается с помощью SMT-решателей.

Примеры тестовых случаев для функции **target**:

- возвращает `true` при $x = 4, y = 1$;
- возвращает `false` при $x = 4, y = 5$;
- вызывает ошибку при $x = 4, y = 3$.

2.1.2. SMT-решатели

SMT-решатели [2, 7, 22] являются инструментами для автоматизированной проверки выполнимости логических формул в различных теориях. На вход они принимают логическую формулу, состоящую из предикатов, констант и функциональных символов сигнатуры соответствующей теории. Если эта формула выполнима, то решатель возвращает SAT³ и модель, подстановка которой даст истину. Если формула невыполнима и решатель сумел это доказать, то он возвращает UNSAT. При этом решатель может вернуть UNKNOWN или просто зависнуть, так как некоторые теории и их комбинации неразрешимы.

Ниже перечислены некоторые теории, поддерживаемые современными решателями:

- теория линейной целочисленной арифметики. Сигнатура данной теории включает в себя целые числа, операции сложения и вычитания, а также предикаты «равенство» и «меньше»;
- теория битовых векторов. В сигнатуру данной теории входят числа, каждое из которых представляется битовым вектором фиксированной длины (аналогично представлению чисел в машинной арифметике), предикаты «равенство» и «меньше», операции сложения и произведения, а также ряд битовых операций: «и», «или», «исключающее или», «не», «сдвиг влево», «сдвиг вправо», «конкатенация», «взятие подвектора»;
- теория массивов. Сигнатура описываемой теории дополняет сигнатуру линейной арифметики двумя операциями: чтением элемента массива по указанному индексу (`select`) и записью значения в

³Выполнима (англ. satisfiable)

элемент массива по указанному индексу (store). В общем случае теория неразрешима [3];

- теория неинтерпретированных функций. В сигнатуру данной теории входят произвольные функциональные символы с известной арьностью, а также предикат «равенство»;
- теория алгебраических типов данных. Сигнатура данной теории состоит из рекурсивных типов, операции «сумма» и «произведение» над ними. В комбинации с теорией неинтерпретированных функций её можно использовать для верификации кода [19];
- теория конечных множеств. В сигнатуру данной теории входят константы, предикаты «принадлежность», «подмножество» и «равенство множеств», а также функции объединения, пересечения, разности и синглтона для множеств.

2.1.3. Проблемы символьного исполнения

Применение метода символьного исполнения сопряжено с рядом практических сложностей [13, 17, 18]:

- экспоненциальный рост числа состояний. Каждый раз, когда символьное исполнение проходит через инструкцию ветвления, количество символьных состояний удваивается. Это приводит к их экспоненциальному росту и значительному увеличению времени анализа. В частности, эта проблема актуальна при исследовании коллекций с символьными длинами, так как для их обхода необходимо разветвиться количество раз, равное числу элементов. В итоге получается 2 в степени числа элементов символьных состояний;
- взаимодействие с окружением. Современные символьные анализаторы обычно работают с ограниченным набором языков программирования (например, JAVA и KOTLIN). Однако на практике

приложения часто взаимодействуют с внешними компонентами (такими как системные вызовы ОС или микросервисы);

- проблемы трансляции в SMT-формулы. Языковые конструкции по-разному отображаются на теории SMT-решателей, причем некоторые семантические особенности могут быть выражены только в специфических теориях. Однако поддержка расширенных теорий снижает производительность решателей, создавая дилемму между выразительностью и эффективностью.

2.1.4. Аппроксимации

Для анализа приложений, взаимодействующих с внешними системами, активно применяется техника аппроксимации. Под этим термином понимают замену фрагментов кода, которые либо слишком сложны для символьного анализа, либо принципиально не поддерживаются используемым инструментом [4, 23].

Характерный пример — замена вызовов системных функций (таких как получение текущей даты) на символьные переменные. Такой подход позволяет избежать нежелательных побочных эффектов, связанных с реальными системными вызовами. Другой распространённый случай — использование упрощённых реализаций коллекций вместо их сложных аналогов с хеш-функциями, при условии сохранения ключевых семантических свойств.

Основная сложность при работе с аппроксимациями — необходимость точного сохранения семантики исходного кода. Если это условие нарушается, могут возникать ошибочные состояния и невалидные тесты. На практике добиться идеального соответствия крайне сложно. Кроме того, важно понимать, что аппроксимации имеют естественные ограничения — они могут покрывать только ограниченный набор функций. При исследовании новых систем требуется дополнительная работа по созданию соответствующих аппроксимаций.

2.2. Инструмент USVM

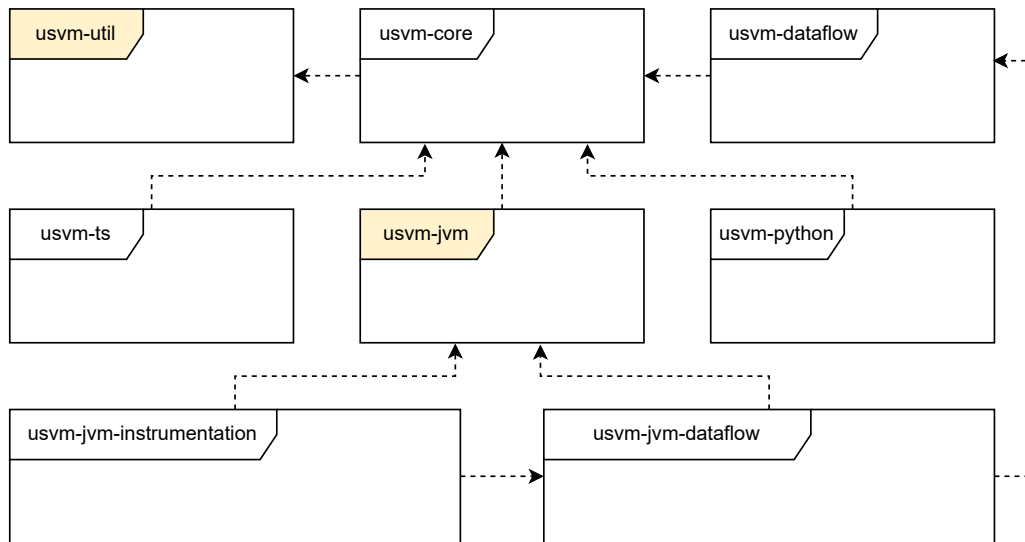


Рис. 2: Архитектурная схема USVM.

USVM — проект универсальной символьной машины, который в своём ядре абстрагируется от конкретных языков программирования. Он содержит модули, отвечающие за общие процессы символьного исследования, и отдельные модули, специализирующее исследование для разных языков. Для представления условий пути используются термины теорий битовых векторов и массивов.

На рис. 2 представлена компонентная архитектура USVM. Желтым цветом выделены компоненты, модифицированные в рамках данной работы.

Основные компоненты системы:

- **usvm-utils** — служебные функции для работы с инженерными коллекциями и реализации повторяющейся логики;
- **usvm-core** — ядро системы, реализующее базовую логику символьного исполнения и предоставляющее API для управления символьными состояниями и условиями пути;
- **usvm-dataflow** — общая инфраструктура для IFDS-анализа и анализа уязвимостей (TAINTED ANALYSIS);

- **usvm-jvm** — адаптер для JAVA, обеспечивающий трансляцию байт-кода в универсальное промежуточное представление;
- **usvm-ts** — аналогичный адаптер для TYPESCRIPT;
- **usvm-python** — модуль поддержки языка PYTHON;
- **usvm-jvm-instrumentation** — инструментарий для модификации JAVA-кода;
- **usvm-jvm-dataflow** — специализированная реализация DATAFLOW-анализа для JAVA.

2.2.1. Библиотека JACODB

Для работы с байт-кодом в рамках проекта используется библиотека JACODB⁴. Данная библиотека предоставляет возможность анализа байт-кода вне процесса JVM с сохранением полученной информации в базе данных. По функциональности JACODB аналогична JAVA REFLECTION, но работает с байт-кодом, хранящимся в файловой системе, представляя его в виде структурированных высокоуровневых элементов: классов, методов и полей с соответствующими интерфейсами для манипуляции.

Важной особенностью JACODB является механизм фич⁵ — специальных классов, модифицирующих соответствующий код согласно заданным правилам. При запросе байт-кода класса по имени или типу JACODB предварительно применяет все установленные фичи перед передачей результата пользователю.

Фичи в JACODB классифицируются по уровню воздействия на три типа: **JcInstExtFeature** для модификации тел методов (предоставляет метод для замены набора инструкций), **JcClassExtFeature** для изменения структуры классов (позволяет добавлять и удалять методы и поля), и **JcClasspathFeature** — наиболее глобальная фича, способная влиять

⁴<https://github.com/UnitTestBot/jacodb/> (дата обращения: 19 мая 2025 г.)

⁵от англ. features

на сам процесс применения фичей (например, реализуя механизмы кеширования).

2.2.2. Аппроксимации

Для реализации аппроксимаций в USVM разработан отдельный проект JAVA-STDLIB-APPROXIMATIONS, содержащий упрощённые версии стандартных JAVA-классов и методов, а также классов популярных библиотек и фреймворков. Эти реализации снабжены специальными подсказками для символьной машины, оптимизирующими процесс их исполнения. При анализе кода, когда встречается зааппроксимированный класс, USVM использует соответствующий модифицированный байт-код.

```
public E _setElement(int index, E element) {
    SymbolicList<E> storage = this.storage;
    Engine.assume(storage != null);
    E oldValue = storage.get(index);
    storage.set(index, element);
    return oldValue;
}
```

Листинг 2: Аппроксимация метода set.

В листинге 2 представлен пример аппроксимации метода `set` класса `ArrayList`. Код начинается с получения ссылки на символьное хранилище данных (`SymbolicList`). Далее с помощью функции `Engine.assume` добавляется условие ненулевого значения хранилища в текущее символьное состояние. Затем выполняются операции чтения и записи, непосредственно взаимодействующие с символьной памятью. В заключение метод возвращает предыдущее значение элемента, что соответствует оригинальной семантике `ArrayList.set`.

2.3. Методы анализа реляционных баз данных

В данном разделе рассматриваются существующие подходы к статическому анализу реляционных баз данных.

2.3.1. Верификация на CoQ

В работе [19] предложен подход верификации реляционных баз данных с использованием языка CoQ. Реляции представлены как множества кортежей, смоделированные через алгебраические типы данных, где порядок значений соответствует порядку атрибутов таблицы, а каждый кортеж представляет отдельную строку.

Запросы декомпозируются на базовые операции: выборку, проекцию, перестановку, объединение, разность и декартово произведение. Каждая операция реализована как преобразование алгебраических типов. На этой основе с помощью механизмов CoQ формулируются утверждения о состояниях памяти до и после выполнения запросов.

2.3.2. Итерационные доказательства

Инструмент COSETTE [6] предоставляет механизм доказательства эквивалентности SQL-запросов. Его архитектура включает два взаимодействующих компонента: первый доказывает семантическое равенство запросов, второй ищет контрпримеры их неравенства.

Компонент поиска контрпримеров работает итеративно, на каждом шаге фиксируя максимальный размер таблиц и используя символьное представление базы данных. Процесс начинается с создания двух идентичных символьных состояний, над которыми выполняются преобразования согласно анализируемым запросам. Полученные ограничения передаются SMT-решателю для проверки возможности неравенства результатов. При отрицательном ответе размер таблиц увеличивается, и процесс повторяется.

Компонент доказательства равенства трансформирует оба запроса в термины языка CoQ и строит формальное доказательство их эквивалентности.

2.3.3. Теория конечных множеств

Работа [16] предлагает расширение теории конечных множеств для реляционных операций. Авторы вводят дополнительные функциональные символы и правила вывода, представляя таблицы как множества кортежей. Расширенная теория включает операции: конструктор кортежей, реляционное произведение, реляционное соединение, переупорядочивание и транзитивное замыкание. Данная реализация доступна в SMT-решателе CVC⁶.

2.3.4. Символьное исполнение

Инструмент QEX [20] применяет метод символьного исполнения для тестирования приложений, работающих с реляционными базами данных. Его ключевая особенность — трансляция запросов в специализированную теорию для SMT-решателя, комбинирующую различные формальные системы, включая теорию алгебраических типов данных.

Таблицы представлены как списки кортежей с сохранением порядка атрибутов. Запросы преобразуются в наборы аксиом — правил перезаписи представлений таблиц, которые применяются последовательно до достижения фиксированной точки. Результаты преобразований передаются SMT-решателю для дальнейшего анализа.

2.3.5. Выводы

Анализ рассмотренных подходов показывает, что решения, основанные на теориях алгебраических типов данных или конечных множеств, не применимы в USVM из-за ограничений используемого SMT-решателя. Итерационный подход, несмотря на свою универсальность, неприемлем из-за существенного замедления общего процесса анализа при многократном выполнении для таблиц разного размера.

Однако ключевые идеи организации таблиц как множеств кортежей и декомпозиции запросов на элементарные операции были адаптированы в рамках данной работы.

⁶<https://github.com/cvc5/cvc5> (дата обращения: 19 мая 2025 г.)

2.4. Фреймворк SPRING JPA

SPRING JPA представляет собой фреймворк, который значительно упрощает работу с JAVA PERSISTENCE API (JPA) за счет автоматизации генерации стандартного кода доступа к данным. Построенный поверх JPA, он предлагает удобный интерфейс для выполнения базовых CRUD-операций и формирования запросов на основе соглашений об именовании методов или их параметров.

Конфигурация сопоставления объектов с реляционной схемой базы данных осуществляется через аннотации, накладываемые на поля `data`-классов. Эти аннотации позволяют определить стратегии загрузки данных (например, `eager` или `lazy loading`), а также настроить различные типы связей между сущностями, такие как `one-to-many` или `many-to-one`. Например, для связи типа `one-to-many` можно явно указать, должно ли поле-коллекция всегда загружаться вместе с родительской сущностью или только по требованию.

Основным механизмом взаимодействия с таблицами базы данных в SPRING JPA являются репозитории — специальные интерфейсы, которые наследуются от базовых интерфейсов фреймворка, таких как `JpaRepository`. Эти интерфейсы параметризуются двумя типами: типом первичного ключа сущности и типом самой сущности. Важно отметить, что методы репозитория не содержат реализации — их семантика определяется автоматически на этапе компиляции.

SPRING JPA поддерживает несколько способов определения пользовательских запросов. Простейший подход основан на соглашениях об именовании методов — например, метод `findByName` автоматически преобразуется в запрос поиска по полю `name`. Для более сложных случаев используется аннотация `@Query`, позволяющая задать запрос непосредственно на HQL (HIBERNATE QUERY LANGUAGE) — объектно-ориентированном диалекте SQL. HQL предоставляет дополнительные возможности, такие как использование имен параметров метода в теле запроса, работа с объектами вместо таблиц и точный контроль стратегий загрузки связанных сущностей.

3. Символьная модель для работы с базами данных

В данном разделе описывается архитектура подсистемы моделирования реляционных баз данных, разработанной для интеграции в проект USVM. Представленное решение является частью символьной виртуальной машины и взаимодействует с её основными компонентами.

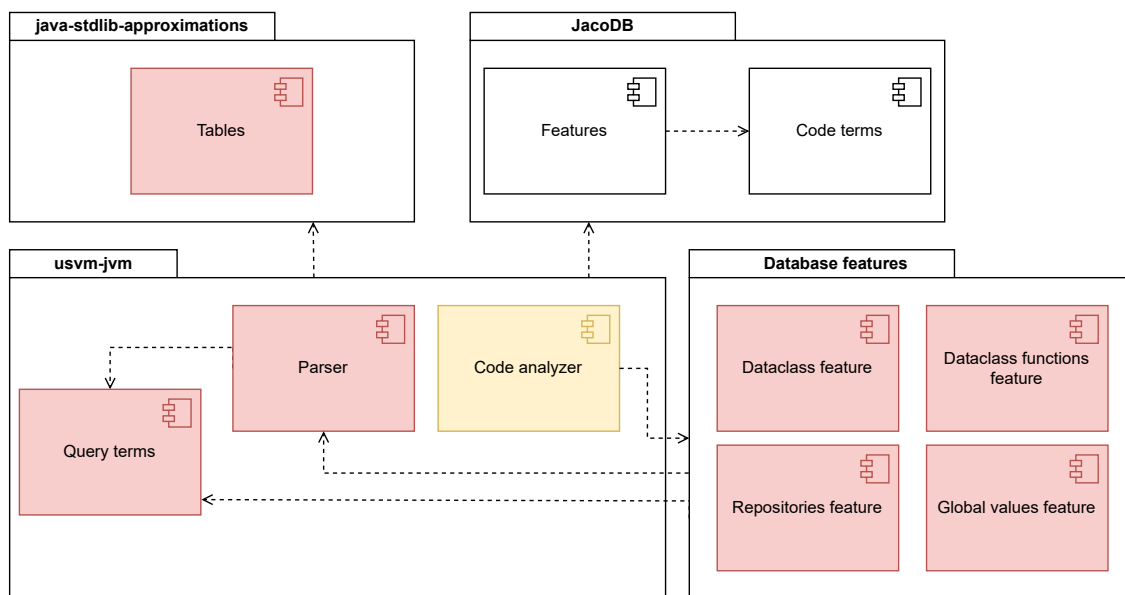


Рис. 3: Архитектурная схема решения.

На рис. 3 показана интеграция системы работы с базами данных в архитектуру USVM. Красным цветом обозначены новые компоненты, желтым — модифицированные, белым — неизменные части системы. Далее подробно рассмотрим ключевые элементы архитектуры.

Code analyzer — основной модуль, выполняющий символьное исполнение кода программы. Для работы с байт-кодом использует механизм фичей библиотеки JacoDB.

Features — базовый набор фичей JacoDB, обеспечивающий чтение, хранение и модификацию пользовательского кода в процессе анализа.

Code terms — классы, описывающие представление кода в терминах библиотеки JACODB. Используются как для анализа существующего кода, так и для генерации нового.

Dataclass features — группа специализированных фичей, реализующих интеграцию с базами данных:

- *dataclass feature* — расширяет data-классы дополнительными полями и методами, необходимыми для генерации запросов;
- *dataclass functions feature* — реализует тела методов, добавленных *dataclass feature* (разделение соответствует архитектуре JACODB);
- *repositories feature* — анализирует семантику методов-запросов в репозиториях и генерирует соответствующий код для символического исследования;
- *global values feature* — добавляет статические переменные, описывающие состояние базы данных.

Parser — модуль обработки запросов, используемый в *RepositoriesFeature*. Он включает парсер HQL на основе оригинального парсера SPRING JPA и специализированный парсер для методов с семантикой, определяемой по имени. Результатом работы всегда является *QueryTerm*.

Query terms — иерархия классов, описывающих структуру запросов. Каждый класс соответствует определённой конструкции запроса и предоставляет метод трансляции в исполняемый код.

Tables — реализация ленивой модели таблиц с символическим размером. Каждый класс таблицы инкапсулирует отложенное действие, что позволяет избежать полного обхода и связанного с ним экспоненциального роста числа путей.

3.1. Структура модели баз данных

На рис. 4 представлена структура модели баз данных, реализованной в проекте аппроксимаций инструмента USVM.

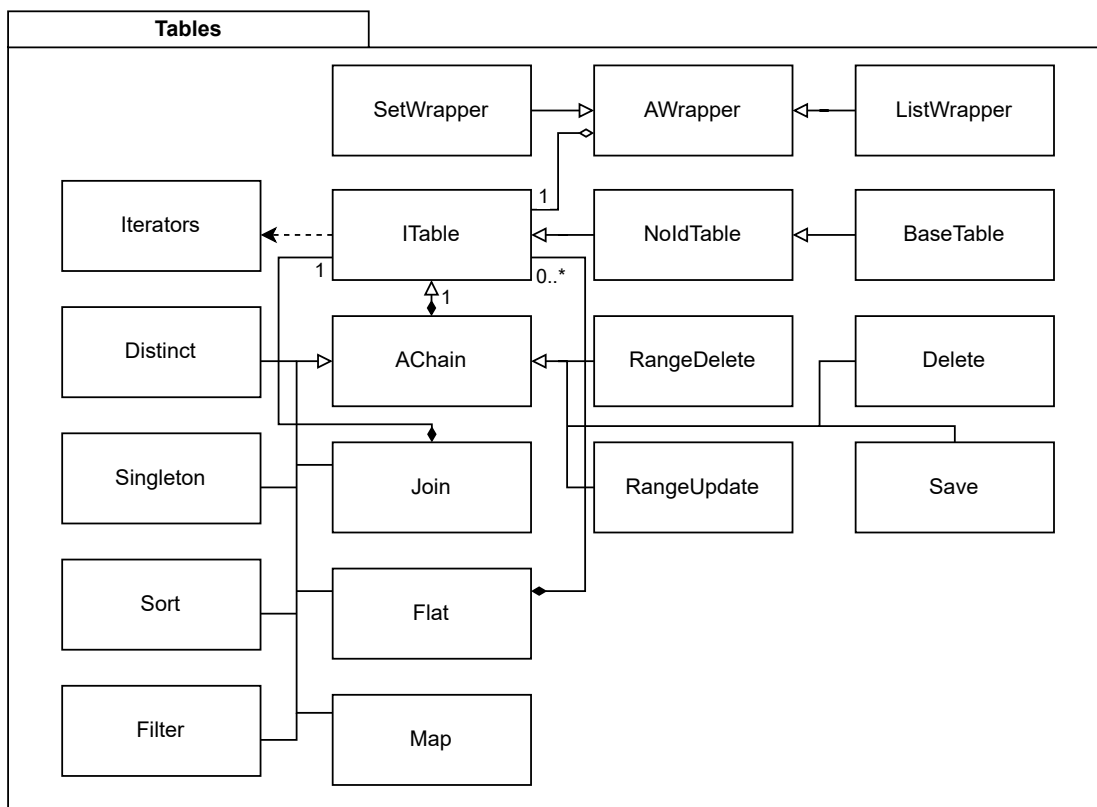


Рис. 4: Структура модели баз данных.

Как упоминалось ранее, для минимизации обходов символьных баз данных с символьными размерами (чтобы избежать проблемы экспоненциального взрыва путей) все операции над данными в символьном исполнении выполняются лениво. Это реализовано в разработанной модели баз данных. Любое действие над данными представляется отдельным классом-таблицей, который инкапсулирует все предыдущие классы-действия. При обращении к ним вышележащая таблица сначала перенаправляет запрос вниз по цепочке, а затем применяет свое действие к полученному результату, возвращая модифицированные данные. Таким образом, несколько последовательно вызванных фильтров в запросе будут представлены цепочкой классов, каждый со своим пре-

дикатом. Опишем все сущности модели баз данных.

ITable — базовый интерфейс, который реализуют все таблицы-классы. Он гарантирует, что каждая таблица может предоставить информацию о типе возвращаемых данных и создать специфичный для себя итератор.

Iterators. Каждой таблице соответствует свой итератор, реализующий действие, предполагаемое семантикой таблицы, в методах `hasNext` и `next`.

NoIdTable — таблица, описывающая таблицу базы данных. Для представления данных используется двумерный массив объектов, где первое измерение соответствует столбцам таблицы, а второе — строкам. При инициализации задает свой размер неотрицательным символьным значением и заполняет массив данных символьными значениями на основе переданных типов данных столбцов. Итератор этой таблицы возвращает массивы объектов, представляющие строки таблицы.

BaseTable — альтернативное представление таблицы базы данных, используемое только для хранения сериализованных в массивы объектов `data`-классов. Фреймворк `SPRING JPA` гарантирует, что у таких классов должно быть идентифицирующее поле, все значения которого уникальны. Поэтому итератор таблицы с помощью вызовов метода `Engine.assume` исключает ветви исполнения, где это ограничение не выполняется.

AChain — абстрактный класс, наследники которого могут выступать как элементы цепочки действий, но не как ее начало. Содержит поле с нижележащей таблицей. В исследуемом коде основанием цепочки являются таблицы `NoIdTable` и `BaseTable`.

Filter — таблица, фильтрующая данные, получаемые от своей под-

таблицы. Решение о выдаче элементов принимается на основе функции-предиката, обязательной для инициализации. Проверка выполняется в функции `next` соответствующего итератора. Если текущий элемент не удовлетворяет условию, он пропускается и не передается выше по цепочке.

Map — таблица, выполняющая преобразования над данными. Это действие описывается передаваемой в конструктор функцией. Может использоваться как для проекции строк, так и для десериализации их в data-классы при использовании конструктора, созданного классом `DataclassFeature`.

Sort — таблица, сортирующая получаемые данные на основе переданной функции. Поскольку операция сортировки требует полного вычисления данных, отложить ее на уровне методов итератора невозможно. Единственной ленивой частью в этой таблице является поздняя инициализация.

Flat. Данные большинства таблиц могут быть совершенно произвольными. В цепочке методов может присутствовать таблица **Map**, способная изменять тип передаваемых значений. Одним из возможных типов может оказаться `ITable`, для которого может потребоваться операция объединения элементов. Для этого итератор **Flat**-таблицы сначала возвращает элементы из первой таблицы, используя ее итератор, а затем переходит к следующей, повторяя эти действия для всех таблиц набора.

Join — таблица, реализующая операцию JOIN из SQL. Результатом соединения двух таблиц всегда является таблица, тип данных которой представляет собой массив объектов (строки таблицы). Такая реализация выбрана для максимального соответствия семантике баз данных. Для инициализации **Join** требует пару таблиц и пару сериализаторов для них. Соединение производится только для строк, удовлетворяющих

заданному предикату. Итератор таблицы `Join`, останавливаясь на одном элементе первой таблицы, лениво итерируется по второй, соединяя строки и возвращая результат.

Singleton — используется для создания таблицы из одного элемента. При передаче сериализатора во время инициализации можно получить таблицу из `data`-класса. Эта возможность применяется, когда нужно выполнить `Join` между объектом и его полем-коллекцией других `data`-классов. В этом случае `Singleton` `data`-класса выступает первой таблицей для `Join`, а поле — второй.

Distinct — таблица, гарантирующая уникальность всех возвращаемых элементов. Итератор таблицы запоминает все уже выданные элементы и сравнивает с ними новых кандидатов.

RangeDelete — таблица, удаляющая все элементы, удовлетворяющие переданному предикату. Подобно `Filter`-таблице, откладывает свое действие на уровень итератора. Поскольку `RangeDelete` имитирует изменение данных в одной из таблиц, она должна обеспечивать уникальность идентификаторов для `BaseTable`. Поэтому итератор таблицы сочетает функциональность итераторов `Filter` и `BaseTable`.

RangeUpdate — таблица для условного обновления элементов. Применяет заданную функцию (аналогично таблице `Map`) к строкам, удовлетворяющим переданному предикату. Изменение данных может нарушить инварианты базовых таблиц, например, уникальность идентификаторов. Таблица проверяет эти условия и возвращает соответствующую ошибку при их нарушении.

Save. Таблица для сохранения одного элемента. Если элемент уже присутствует в таблице (определяется по значению столбца идентификатора для `BaseTable`), то вместо добавления нового элемента обновляется соответствующая строка. Итератор таблицы запоминает значе-

ние, которое должно быть сохранено, и если очередной выдаваемый элемент требует обновления, возвращает сохраненное значение. Если обновление не произошло, последним возвращенным элементом будет это значение.

Delete — таблица для удаления конкретного элемента. Ее семантика аналогична таблице **Save**, но вместо сохранения или обновления элемент исключается из таблицы. Удаление может не произойти, если ни одна строка не соответствует условию.

AWrapper. В фреймворке SPRING JPA результатом вызова метода-запроса могут быть коллекции разных типов. Реализация всех их интерфейсов в классах таблиц была бы слишком сложной, поэтому для них созданы классы-обёртки. Эти классы содержат цепочку таблиц — поле типа **ITable**, определенное в абстрактном классе **AWrapper**. В классе реализован метод **unwrap**, возвращающий хранящуюся цепочку таблиц, который может использоваться при составлении запросов.

ListWrapper — один из классов-обёрток, реализующий интерфейс стандартной коллекции **List**. Вызовы методов **ListWrapper** инициируют вычисления в цепочке таблиц, так как по семантике их недопустимо откладывать — любой вызов функций коллекции представляет собой вызов из пользовательского кода, который запрещено игнорировать. Для минимизации вычислений все результаты кэшируются, поэтому повторные вызовы одних и тех же функций не приводят к повторным вычислениям.

SetWrapper — класс-обёртка для интерфейса коллекции **Set**. Использует те же принципы, что и **ListWrapper**, но дополнительно гарантирует уникальность своих значений.

3.2. Парсинг запросов

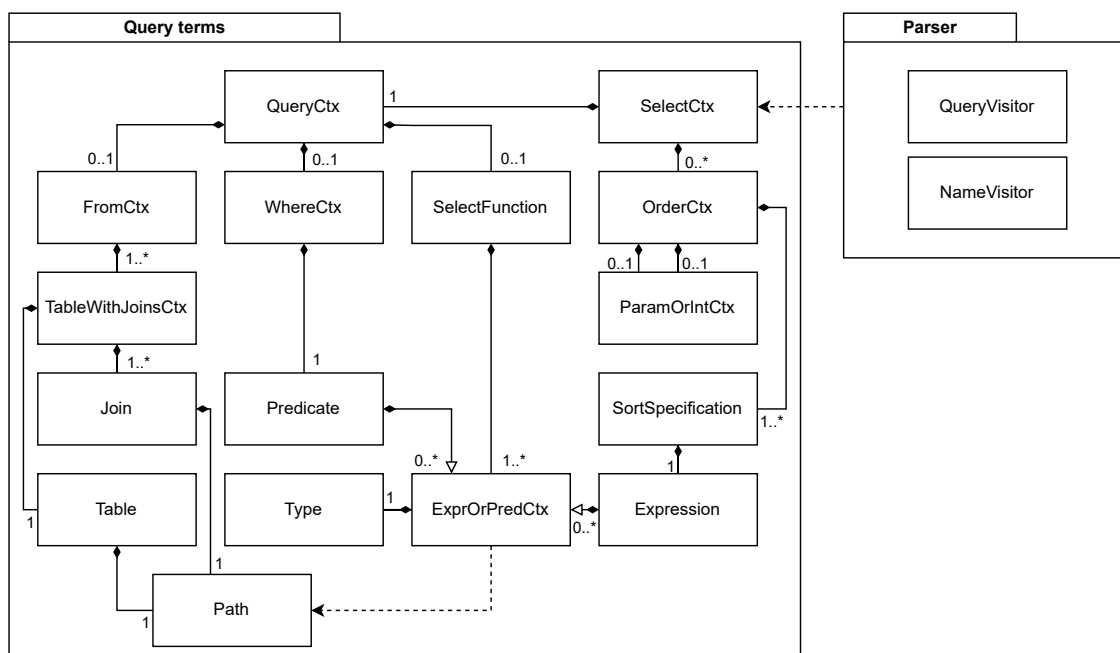


Рис. 5: Система генерации запросов.

На рис. 5 представлена система генерации запросов. На основе пользовательских методов-запросов конструируются термы, каждый из которых транслируется в код для анализа. Рассмотрим эту систему подробнее.

Как уже было сказано выше, для определения семантики методов-запросов используются два класса-парсера, отвечающие каждый за отдельное представление:

- для запросов на языке HQL используется парсер на основе парсера из фреймворка SPRING JPA. Чтобы породить собственные термы на основе запросов, но переиспользовать грамматику языка, в классе `QueryVisitor` был переопределён визитор;
- для методов-запросов, чья семантика заключена в имени, был реализован класс `NameVisitor`. Он использует регулярные выражения, разбирая весь ограниченный набор языковых конструкций имён.

Результатом парсинга является терм. Его использует класс `RepositoriesFeature` для заполнения пустых тел методов-запросов, вызывая соответствующий метод терма. Большинство термов вложены друг в друга и сопоставляются с какой-то одной конструкцией языков запросов:

- ***selectCtx*** — терм-результат, содержит в себе подтермы, описывающие весь запрос. При трансляции в код может оборачивать значения своих подтермов в **Sort**-таблицы и классы-обёртки в соответствии с возвращаемым типом функции-запроса;
- ***orderCtx*** — терм, описывающий порядок элементов в коллекции, возвращаемой запросом. Транслируется в лямбда-функцию, которая используется при инициализации **Sort**-таблиц в терме `SelectCtx`;
- ***paramOrIntCtx*** — терм, который описывает либо целочисленное значение, либо значение одного из аргументов оригинального метода-запроса. Используется для задания параметров сортировки в терме `OrderCtx`, например, числа элементов, которые должны быть пропущены. Транслируется либо в литерал, либо в обращение к аргументу;
- ***sortSpecification*** — терм, специфицирующий значения, по которым будет производиться сортировка. Транслируется в лямбда-функцию, которую использует терм `OrderCtx`. Эта функция берёт элемент запроса, выделяет в нём значение и модифицирует его в соответствии с семантикой запроса. Полученные значения будут сравниваться между собой в **Sort**-таблице;
- ***queryCtx*** — терм, описывающий, из какой таблицы и какие данные должны быть возвращены. Комбинирует транслированные значения своих подтермов в цепочку из **Map** и **Filter** таблиц;
- ***fromCtx*** — терм, который содержит информацию о том, к какой из таблиц адресован запрос и с какими таблицами она должна

быть соединена. При трансляции начинает формировать цепочку таблиц с одной из базовых таблиц, присоединяя к ней с помощью Join-таблиц другие;

- ***tableWithJoinsCtx***. Язык HQL позволяет делать запрос одновременно к нескольким таблицам, поэтому в терме **FromCtx** находится несколько термов **TableWithJoinsCtx**, которые описывают каждый свою собственную базовую таблицу и свой набор Join-операций;
- ***table*** — терм с именем базовой таблицы. Транслируется в обращение к одному из глобальных полей с таблицами, которые были созданы классом **GlobalValuesFeature**;
- ***join*** — терм, описывающий сущность, с которой должна быть соединена основная в запросе таблица. Хранит в себе имя, но в отличие от **Table**-терма это имя может быть сложным. Например, **Owner.pets**, где **pets** — поле класса **Owner** со списком значений **data**-класса **Pet**. Такой Join-терм должен быть транслирован в комбинацию нескольких **Map**, **Flat** и **Join** таблиц;
- ***path*** — терм, который хранит пути, состоящие из имён, до запрашиваемых сущностей. Не транслируется во что-то конкретное, но на основе его значений принимаются решения в других термах;
- ***whereCtx*** — терм, отвечающий за фильтрацию элементов в результате запроса. Содержит в себе терм предиката, который участвует при трансляции в **Filter**-таблицу;
- ***predicate*** — набор термов-предикатов, которые могут быть использованы в терме **WhereCtx**. Транслируются в лямбда-функции на основе хранящегося терма-выражения. В теле этих функций могут встречаться как простые литералы, так и сравнения элементов строки с аргументами основного метода-запроса;
- ***selectFunction*** — терм, описывающий, какие и в каком виде должны быть возвращены значения на основе термов-выражений.

Это могут быть как data-классы, так и значения одного из полей, к которым будут применены какие-то действия. Эти воздействия выражаются в лямбда-функции, в которую транслируется терм. Лямбда-функция используется в терме `QueryCtx` для инициализации Map-таблиц;

- ***exprOrPredCtx*** — терм, который наследуют классы терма-предиката и терма-выражения. Содержит в себе тип выражения;
- ***type*** — терм-тип. Так как для инициализации некоторых таблиц, например, Map-таблицы, необходимо передавать значение типа, то терм `Type` транслируется в значения JAVA-типа `Class<T>`;
- ***expression*** — набор термов-выражений, описывающих функции языка HQL над значениями и сами эти значения. В зависимости от конкретного терма, могут быть транслированы в простые литералы, в обращения к аргументам оригинального метода-запроса, в простые операции над другими выражениями и их сравнения.

3.3. Процесс символьного исполнения

Рассмотрим базовые сценарии обращения символьного исследования к реализованной подсистеме моделирования баз данных, которые представлены на рис. 6.

В блоке `dataclass`, который на рис. 6 указан первым, представлен сценарий, где символьное исполнение встречает data-класс, чьи тела методов и поля должны участвовать в исследовании. Чтобы получить код для анализа, `Code analyzer` обращается к `Features` — набору классов, преобразующих код, который содержит классы `Dataclass` и `DataclassFunctions`. `Features` перенаправляет запрос к ним, так как только они могут на него ответить. Эти классы собирают информацию о полях в пользовательском data-классе, на её основе создают новые поля и методы, необходимые для работы модели баз данных, и передают их в символьное исполнение.

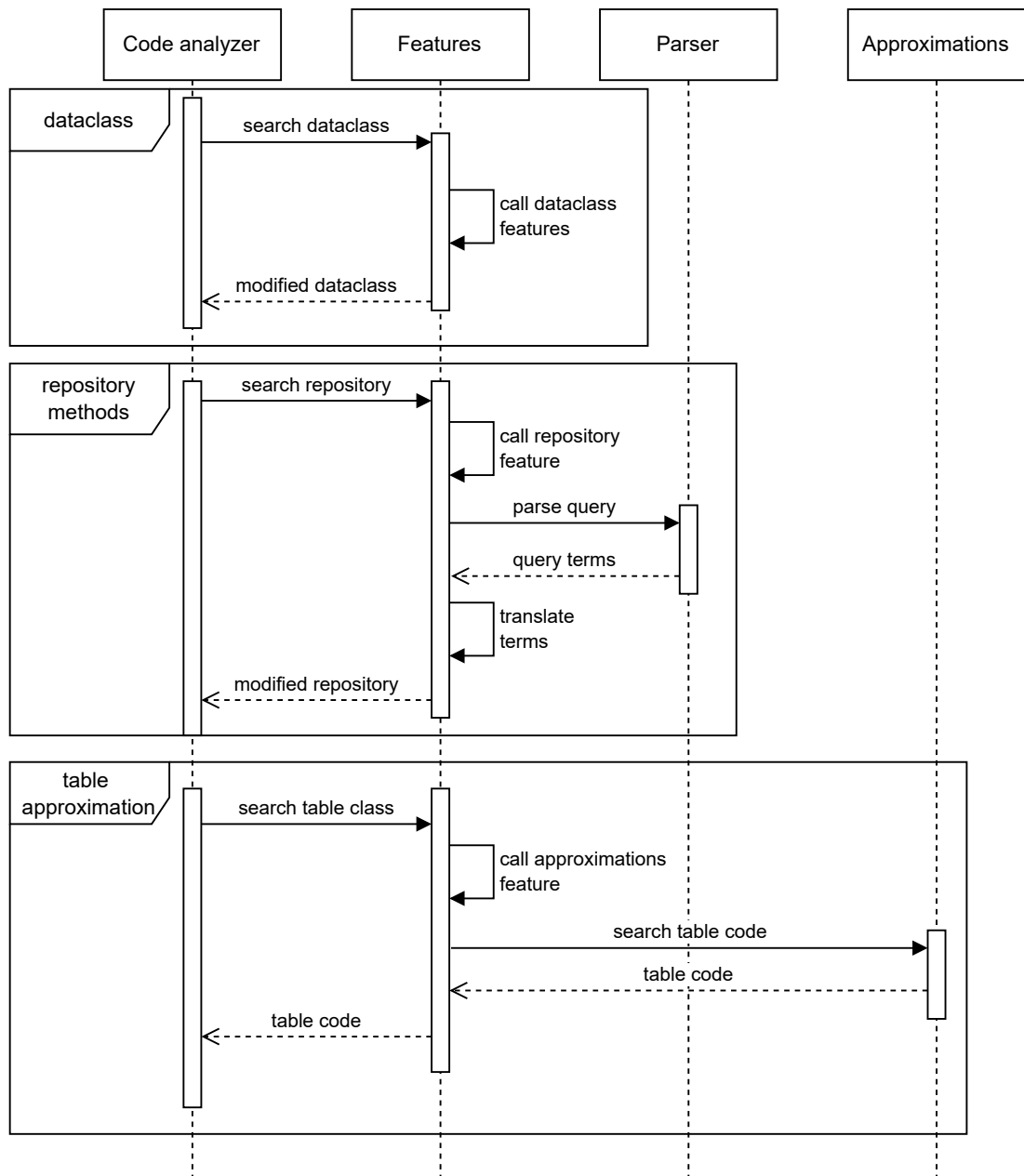


Рис. 6: Процесс символьного исследования.

Во втором блоке описан сценарий исследования методов класса репозитория. Как и в первом случае, **Code analyzer** обращается к **Features**, чтобы получить код методов репозитория. В этом сценарии во **Features** на запрос отвечает класс **Repository**. Он анализирует метод-запрос из пользовательского кода и определяет, каким образом выражена его семантика: через запрос на языке HQL или по имени метода. Затем вызывается соответствующий парсер, который возвращает терм запроса. Класс **Repository** вызывает метод терма для трансляции в код и возвращает результат для символьного анализа.

В последнем сценарии представлена ситуация, в которой анализ встречается класс из модели баз данных — например, класс-таблицу или класс-обёртку. **Code analyzer**, используя стандартный механизм, запрашивает код этого класса. Поскольку классы модели реализованы в проекте аппроксимаций, на запрос во **Features** отвечает стандартный класс, ответственный за замену кода на аппроксимации. В данном случае он использует для аппроксимации код классов-таблиц, описанных выше. Они передаются как ответ на запрос символьному исполнению.

3.3.1. Пример кода для анализа метода-запроса

Рассмотрим код тела функции репозитория **findAll** (листинг 3), сгенерированный подсистемой моделирования баз данных на основе запроса, указанного в аннотации **@Query**. Этот код будет выполнен символьно для анализа взаимодействия с базой данных.

Согласно семантике функции **findAll**, описанной на языке HQL в аннотации **@Query**, она должна возвращать список всех объектов типа **Book**, сохранённых в базе. В первой строке сгенерированного кода происходит обращение к одной из глобальных переменных базы данных, содержащей **BaseTable** для data-класса **Book**. Таблица **BaseTable** хранит данные в сериализованном виде — в виде массивов объектов. Для десериализации данных **BaseTable** оборачивается таблицей типа **MappedTable**, которая инициализируется с помощью лямбда-функции (в ней используется созданный статический десериализатор класса **Book**). Чтобы возвращаемая коллекция соответствовала интерфейсу

List, она оборачивается классом-обёрткой ListWrapper, реализующим этот интерфейс. Полученное значение возвращается из функции.

```
@Query("SELECT book FROM Book book")
List<Book> findAll() {
    BaseTable _base = GlobalDatabases.book;
    MappedTable _map = new MappedTable(
        _base,
        (Object[] row) -> Book._staticNew(row)
    );
    List<Book> _wrapper = new ListWrapper(_map);
    return _wrapper;
}
```

Листинг 3: Сгенерированный код функции findAll.

4. Особенности реализации

Данная глава описывает особенности реализации подсистемы моделирования реляционных баз данных для символьного исполнения. Реализация выполнена на языках KOTLIN (исследование кода) и JAVA (аппроксимация кода). Исходный код доступен по ссылкам^{7,8}. Далее подробно рассмотрим ключевые аспекты реализации:

- механизм создания кода;
- создание глобальных полей таблиц;
- валидацию значений баз данных;
- реализацию классов-обёрток;
- модификацию пользовательских data-классов.

4.1. Механизм создания кода

Как упоминалось ранее, исследуемый JAVA-код описывается значениями классов библиотеки JACODB, а для его модификации используется механизм фичей — цепочки классов-генераторов, изменяющих код по заданным правилам. Для символьной модели базы данных исследуемый код в большинстве случаев создаётся с нуля — например, генерируются тела методов репозиториев.

Для модификации кода в рамках одного метода в USVM существует класс `BlockGenerationContext`. Он содержит контекст генерации кода:

- список всех инструкций метода;
- позицию последней инструкции;
- следующий свободный индекс для переменной.

Класс предоставляет базовые методы для работы с инструкциями:

⁷<https://github.com/Tozarin/usvm> (дата обращения: 19 мая 2025 г.)

⁸<https://github.com/Tozarin/java-stdlib-approximations> (дата обращения: 19 мая 2025 г.)

- `addInstruction` — добавляет инструкцию в конец списка;
- `nextLocalVar` — создаёт объект переменной со свободным индексом;
- `replaceInstructionAtLocation` — заменяет инструкцию по указанной позиции.

Возможности KOTLIN позволяют доопределять методы класса вне его файла. Поэтому в рамках данной работы был реализован ряд методов-расширений для `BlockGenerationContext`:

- `generateIsEqual` — генерирует инструкции сравнения двух значений и возвращает переменную с результатом;
- `compare` — создаёт инструкции сравнения значений по заданному условию;
- `toBoolean` — преобразует значение типа `bool` в `Boolean` (необходимо для предикатов, так как результаты сравнений могут иметь тип `bool`, а JAVA-лямбды могут возвращать только ссылочные типы);
- `generatedMethodArgumentVar` — извлекает аргумент генерируемого метода в отдельную переменную;
- `generateObjectArray` — создаёт массив объектов заданного размера;
- `putArgumentsToArray` — объединяет все аргументы функции в один массив объектов (используется для передачи контекста между методами);
- `generateNewWithInit` — создаёт и инициализирует объект указанного типа с помощью подходящего конструктора;
- `generateStaticCall` — генерирует вызов статического метода;

- `generateVirtualCall` — создаёт вызов метода класса для заданного объекта;
- `generateGlobalTableAccess` — помещает значение глобальной переменной таблицы в отдельную переменную (используется во всех запросах);
- `generateLambda` — оборачивает метод в лямбда-функцию для инициализации таблиц.

Методы, генерирующие код для запросов и data-классов, также реализованы как расширения `BlockGenerationContext` и представляют собой комбинации вызовов описанных вспомогательных методов.

4.2. Создание глобальных полей таблиц

Глобальные переменные, хранящие классы-таблицы, создаются классом `DatabaseGenerator` на начальном этапе исследования SPRING-специфичного кода. К этим полям обращается сгенерированный код методов репозиториев и data-классов.

Инициализация полей выполняется объектами класса `TableManager`. Эти объекты хранят цепочки базовых таблиц — `BaseTable`, `NoIdTable`, `SaveTable` и другие. Интерфейс `TableManager` предоставляет возможность добавления новых таблиц в эти цепочки. Данный механизм позволяет отражать изменения данных в базе, вносимые пользователем, сохраняя при этом начальное состояние таблицы в отдельном поле. Это сохраненное состояние может быть использовано для инициализации базы данных при воспроизведении теста.

Для создания экземпляров `BaseTable` и `NoIdTable` используется информация о типах полей data-классов и характере связей между ними. Эти данные собираются и обрабатываются классом `JcTableCollector`, формирующим набор объектов типа `TableInfo`. Каждый такой объект содержит список колонок для конкретного data-класса, связанные сущности и типы связей между ними.

Для отражения связей one-to-one, one-to-many и many-to-many система добавляет новые колонки по аналогии с реляционными базами данных. В некоторых случаях требуется создание отдельных таблиц, не описывающих непосредственно пользовательские data-классы.

Рассмотрим пример: класс `Book` содержит поле `id` — целочисленный идентификатор, и поле `pages` — список объектов класса `Page`. Класс `Page` имеет только поле `key`, выполняющее роль идентификатора (аналогично `id` в `Book`). В этом случае система создаст дополнительную таблицу типа `NoIdTable` с двумя колонками: `book_id` и `pages_key`.

4.3. Валидация значений базы данных

Фреймворк `SPRING JPA` позволяет валидировать data-классы при сохранении, предотвращая модификацию базы данных некорректными значениями. Пользователь может настроить эту валидацию, добавляя аннотации к полям data-классов из стандартного набора фреймворка. Для символьной модели это означает, что начальные состояния данных должны удовлетворять предикатам-валидаторам — в противном случае такие значения не могли бы попасть в базу данных.

Поскольку аннотации пользовательских data-классов известны статически до начала символьного исполнения, валидацию модели можно выполнить на этапе её инициализации. Для этого в `TableManager` в цепочку таблиц добавляется специальная таблица, применяющая метод `Engine.assume`. Этот метод модифицирует условия пути для текущего символьного состояния, в результате чего символьное исполнение не станет исследовать ветви, содержащие только невалидные данные.

Аргументом метода `Engine.assume` выступает значение типа `bool`, выражающее ограничение на символьные значения. Для валидации модели это значение конструируется с использованием стандартных реализаций валидаторов `SPRING JPA`. Сначала создаётся объект валидатора, затем он инициализируется на основе аннотации валидируемого поля с использованием внутренних механизмов `SPRING JPA`. Для каждого символьного значения, соответствующего этому полю, вызывает-

ся функция `isValid` валидатора, которая выполняется символично, порождая новые состояния. Среди них остаются только те, для которых `isValid` возвращает `true`, благодаря действию `Engine.assume`. Эти состояния используются для дальнейшего исполнения.

4.4. Реализация классов-обёрток

Как упоминалось ранее, фреймворк SPRING JPA для описания полей-коллекций и возвращаемых типов методов-запросов использует коллекции из ограниченного набора, такие как `List`, `Set`, `Map` и другие. Поскольку прямая реализация этих интерфейсов в классах-таблицах оказалась бы чрезмерно сложной, для их поддержки были разработаны специальные классы-обёртки.

Помимо реализации интерфейсов коллекций, классы-обёртки выполняют ещё одну важную функцию — минимизацию символьных вычислений за счёт кеширования ранее использованных объектов. В частности, класс `ListWrapper`, реализующий интерфейс `List`, использует следующий механизм кеширования (рис. 7).

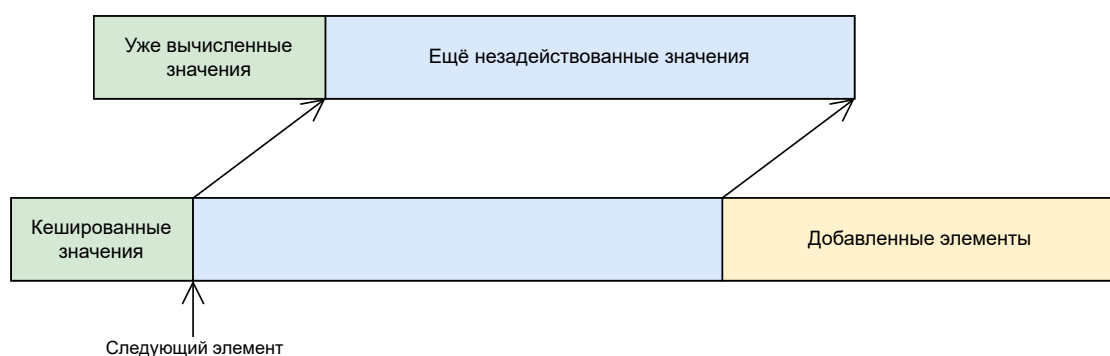


Рис. 7: Механизм кеширования в `ListWrapper`.

Внутренняя структура класса разделяет данные на две компоненты: основную таблицу с исходными данными и список кешированных элементов. Синхронизация между ними осуществляется через набор управляющих переменных, отслеживающих границы между вычисленными и необработанными элементами. Эти границы изменяются при

двух условиях: при довычислении элементов (сдвиг вперёд) и при удалении элементов (сдвиг назад).

Алгоритм вычисления элементов работает по следующему принципу: если запрашиваемый индекс выходит за пределы текущей вычисленной области, система кеширует все элементы от текущей границы до требуемого индекса. Такой подход исключает появление «дыр» в кеше.

Операции модификации реализованы следующим образом. При добавлении элементов в конец списка они помещаются в конец кеша без изменения основной таблицы. Вставка по индексу требует предварительного вычисления элементов до указанной позиции. Операции удаления и доступа работают по аналогичной схеме с предварительным вычислением при необходимости.

4.5. Модификация data-классов

Для внедрения символьной модели базы данных в символьное исполнение модифицируется код пользовательских data-классов. Добавляется ряд методов и полей, позволяющих сериализовывать и десериализовывать объекты data-классов из строкового представления баз данных, а также упрощающих конструирование запросов к модели.

Если класс связан отношением one-to-one с другим классом, ему добавляется поле для хранения идентификатора связанного объекта. Это значение используется для корректной десериализации объекта из базы данных при заполнении оригинального поля. Когда для связи между сущностями требуется промежуточная таблица, в обоих классах создаётся поле типа **Set**, содержащее идентификаторы связанных объектов. Оно инициализируется как выборка из промежуточной таблицы с фильтрацией по значению идентификатора класса.

Новые поля участвуют в десериализации следующим образом. В data-класс добавляется конструктор, принимающий строковое представление класса — массив объектов. Поля простых типов заполняются прямым обращением по индексу массива. Для полей, представляющих

другие data-классы или коллекции объектов, используется запрос к символьной модели. Формируется цепочка таблиц, начинающаяся с базовой таблицы связанного класса и включающая **Filter**-таблицу и **Map**-таблицу. Аргументами при создании таблиц выступают конструкторы других классов и лямбда-функции, которые добавляются в data-класс и используют значения промежуточных полей.

Поскольку в SPRING JPA тип поля-коллекции может быть только одним из базовых интерфейсов коллекций, цепочка таблиц оборачивается в класс-обёртку. Если поле представляет другой data-класс, у класса-обёртки вызывается метод **first**, возвращающий первый элемент. Благодаря ленивости символьной модели, запросы для инициализации сложных полей, не используемых в исходном коде программы, не будут выполняться символично.

Помимо описанных методов, в класс добавляются статические методы, возвращающие значение поля или сериализующие объект. Они используются как лямбда-функции для инициализации таблиц при составлении запросов. Для любого data-класса существует метод с фиксированным названием, возвращающий значение идентификатора объекта.

4.5.1. Реализация методов **Save** и **Delete**

Для внесения изменений в символьную модель базы данных в data-классы добавляются два ключевых метода:

- **save** — обновляет существующую запись или добавляет новую запись об объекте;
- **delete** — удаляет запись об объекте из модели.

Основная сложность реализации этих методов связана с их потенциальным рекурсивным воздействием на связанные объекты. Например, удаление объекта **Book**, содержащего коллекцию объектов **Page**, может потребовать каскадного удаления всех связанных страниц. Такое поведение определяется пользователем через аннотации data-классов, где

для сложных полей можно указать тип зависимости — например, запретить рекурсивное сохранение, но разрешить обновление и удаление связанных объектов.

Информация о типах зависимостей класса собирается компонентом `JcTableCollector` и сохраняется в объекте типа `TableInfo`. Эти данные используются при генерации методов `Save` и `Delete` внутри `data`-классов. Если тип каскадной связи разрешает рекурсивные операции, методы автоматически вызываются для связанных сущностей.

Оба метода получают контекст выполнения — набор ссылок на объекты, для которых методы уже были выполнены. Это предотвращает циклическое выполнение операций над одними и теми же объектами. При каждом вызове метода выполняется проверка наличия объекта в контексте, и если объект уже обработан, выполнение метода прерывается.

Методы также обновляют данные в промежуточных таблицах связей. Для упрощения генерации соответствующего кода в аппроксимациях был реализован класс `SaveUpdDelManyManager`, который вызывается из методов `Save` и `Delete` и формирует запросы для модификации таблиц по стандартным шаблонам. Например, удаляет все строки, где значение первого столбца соответствует идентификатору объекта, что означает удаление всех связей данного объекта с объектами его связанного поля.

5. Тестирование и эксперименты

В данной главе описан способ тестирования реализованной в рамках инструмента USVM подсистемы моделирования реляционных баз данных для символьного исполнения.

5.1. Способ тестирования модели базы данных

Тестирование проводилось с помощью существующей в рамках инструмента USVM системы для автоматической генерации тестов веб-приложений, использующих фреймворк SPRING.

Вначале система собирает в исходном коде все классы, которые наследуют от класса `Controller` или имеют аннотацию `@RestController`. Такие классы называются контроллерами; их методы обрабатывают запросы, направленные приложению пользователями.

Далее система запускает символьное исследование методов контроллеров. После завершения анализа или по истечении отведённого времени генерируются тесты для каждого метода на основе полученных состояний. Важной характеристикой созданных тестов является покрытие — метрика, которая измеряется в процентах и представляет собой отношение покрытых тестами инструкций к общему числу инструкций в контроллерах.

Тестирование было выполнено на демонстрационном проекте SPRING-PETCLINIC⁹, использующем SPRING JPA для работы с базой данных, а также на наборах контроллеров тестового проекта SPRING-TEST-DEMO¹⁰.

- *simple* — контроллеры, которые вызывают простые методы запросов `findAll` и `findById`, а также удаляют и изменяют данные в базе с помощью методов `save` и `delete`;
- *search* — контроллеры, использующие более сложные конструкции языка HQL, например, выражение `LIKE`;

⁹<https://github.com/spring-projects/spring-petclinic> (дата обращения: 21 мая 2025 г.)

¹⁰<https://github.com/Tozarin/spring-test-demo> (дата обращения: 22 мая 2025 г.)

- *aggregator* — контроллеры, которые применяют агрегирующие функции, такие как COUNT, AVG, SIZE и другие. В запросах также используются конструкции для работы с таблицами: HAVING и GROUP BY;
- *advance* — контроллеры, запросы которых включают функции языка HQL, например IS EMPTY, а также сложные логические условия в предикатах.

Тестирование проводилось на ноутбуке APPLE MACBOOK PRO с процессором M1 (архитектура ARM) и 16 ГБ оперативной памяти. В качестве операционной системы использовалась MACOS. Символьное исследование было ограничено 250000 инструкциями, что соответствует временному ограничению в 3 минуты. Ограничение на количество инструкций было равномерно распределено между всеми контроллерами.

5.2. Сравнение результатов

Группа контроллеров	Исслед. конструкций	Число конструкций	Покрытие без модели (%)	Покрытие с моделью (%)
spring-petclinic	33	33	11	39
spring-test-demo/simple	15	15	0	54
spring-test-demo/search	50	50	0	62
spring-test-demo/aggregator	52	79	0	0
spring-test-demo/advance	59	85	0	11

Таблица 1: Сравнение покрытия контроллеров

В таблице 1 представлено сравнение покрытия контроллеров для инструмента USVM с подсистемой моделирования реляционных баз данных и без неё. Первый столбец содержит наборы контроллеров, анализировавшихся инструментом USVM. Во втором столбце указано количество распознанных конструкций языка HQL в методах репозитория. Третий столбец отражает общее количество HQL-конструкций в использованных методах-запросах. В предпоследнем столбце показан процент

тестового покрытия, полученного инструментом USVM без использования символьной модели базы данных. Последний столбец демонстрирует уровень покрытия, достигнутый при использовании символьной модели базы данных.

Предварительные результаты тестирования свидетельствуют, что для групп контроллеров, содержащих вызовы методов-запросов, USVM с подсистемой моделирования реляционных баз данных обеспечивает более высокое покрытие по сравнению с версией без этой подсистемы. Ячейки таблицы, выделенные зелёным цветом, отмечают случаи увеличения уровня покрытия.

Особого внимания заслуживают наборы контроллеров, для которых не удалось исследовать все HQL-конструкции (отмечены красным цветом). Для этих контроллеров инструмент USVM с моделью базы данных показывает более низкое покрытие по сравнению с другими группами. Это происходит, потому что эти контроллеры содержат конструкции, которые на данный момент не поддерживаются, а именно: агрегирующие функции (COUNT, AVG, GROUP BY) и функций HQL (такие как LENGTH, SIZE).

Заключение

В ходе работы были получены следующие результаты:

- выполнен обзор методов и технологий:
 - метода символьного исполнения, включая актуальные проблемы области и некоторые применяемые инженерные решения;
 - методов анализа реляционных баз данных: верификация на CoQ, итерационные доказательства, расширение теории конечных множеств, комбинированные теории;
 - возможностей механизмов доступа к данным в SPRING JPA;
 - структуры и модулей инструмента USVM, проекта с аппроксимациями для языка JAVA, а также возможностей библиотеки JACODB.
- разработана символьная модель для работы с реляционными базами данных на основе аппроксимирующих ленивых коллекций данных и SMT-теории массивов;
- разработанная модель реализована в рамках USVM с использованием языков KOTLIN (исследование кода) и JAVA (аппроксимация кода);
- выполнена апробация символьной модели через генерацию тестового покрытия, подтвердившая, что USVM с поддержкой моделирования баз данных эффективнее обрабатывает вызовы методов-запросов по сравнению с версией без моделирования.

Список литературы

- [1] Automatic verification of heap-manipulating programs / Yurii Kostyukov, Konstantin Batoev, Dmitry Mordvinov et al. // arXiv preprint arXiv:1906.10204. — 2019.
- [2] Bjørner Nikolaj, Phan Anh-Dung, Fleckenstein Lars. [νz-an optimizing SMT solver](#) // Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2015. — P. 194–199.
- [3] Bradley Aaron R, Manna Zohar, Sipma Henny B. [What’s decidable about arrays?](#) // Verification, Model Checking, and Abstract Interpretation / Springer. — 2006. — P. 427–442.
- [4] 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.
- [5] Cadar Cristian, Sen Koushik. Symbolic execution for software testing: three decades later // [Communications of the ACM](#). — 2013. — Vol. 56, no. 2. — P. 82–90.
- [6] Cosette: An Automated Prover for SQL / Shumo Chu, Chenglong Wang, Konstantin Weitz, Alvin Cheung // CIDR. — 2017. — P. 1–7.
- [7] 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.
- [8] Dependence guided symbolic execution / Haijun Wang, Ting Liu, Xiaohong Guan et al. // [IEEE Transactions on Software Engineering](#). — 2016. — Vol. 43, no. 3. — P. 252–271.
- [9] [Differential symbolic execution](#) / Suzette Person, Matthew B Dwyer, Sebastian Elbaum, Corina S Păsăreanu // Proceedings of the 16th

ACM SIGSOFT International Symposium on Foundations of software engineering. — 2008. — P. 226–237.

- [10] Kapus Timotej, Cadar Cristian. [A segmented memory model for symbolic execution](#) // Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2019. — P. 774–784.
- [11] King James C. Symbolic execution and program testing // [Communications of the ACM](#). — 1976. — Vol. 19, no. 7. — P. 385–394.
- [12] Labuschagne Adriaan, Inozemtseva Laura, Holmes Reid. [Measuring the cost of regression testing in practice: A study of Java projects using continuous integration](#) // Proceedings of the 2017 11th joint meeting on foundations of software engineering. — 2017. — P. 821–830.
- [13] [Loop-extended symbolic execution on binary programs](#) / Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song // Proceedings of the eighteenth international symposium on Software testing and analysis. — 2009. — P. 225–236.
- [14] Memory models in symbolic execution: key ideas and new thoughts / Luca Borzacchiello, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu // [Software Testing, Verification and Reliability](#). — 2019. — Vol. 29, no. 8. — P. e1722.
- [15] Păsăreanu Corina S, Visser Willem. A survey of new trends in symbolic execution for software testing and analysis // [International journal on software tools for technology transfer](#). — 2009. — Vol. 11, no. 4. — P. 339–353.
- [16] [Relational constraint solving in SMT](#) / Baoluo Meng, Andrew Reynolds, Cesare Tinelli, Clark Barrett // Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings / Springer. — 2017. — P. 148–165.

- [17] Selective symbolic execution / Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea // Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep). — 2009.
- [18] [Symbolic execution of complex program driven by machine learning based constraint solving](#) / Xin Li, Yongjuan Liang, Hong Qian et al. // Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. — 2016. — P. 554–559.
- [19] [Toward a verified relational database management system](#) / Gregory Malecha, Greg Morrisett, Avraham Shinnar, Ryan Wisnesky // Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 2010. — P. 237–248.
- [20] Veanes Margus, Tillmann Nikolai, De Halleux Jonathan. [Qex: Symbolic SQL query explorer](#) // Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16 / Springer. — 2010. — P. 425–446.
- [21] Veritesting Challenges in Symbolic Execution of Java / Vaibhav Sharma, Michael W. Whalen, Stephen McCamant, Willem Visser // [SIGSOFT Softw. Eng. Notes](#). — 2018. — Vol. 42, no. 4. — P. 1–5.
- [22] [cvc4](#) / Clark Barrett, Christopher L Conway, Morgan Deters et al. // Computer Aided Verification: 23rd International Conference / Springer. — 2011. — P. 171–177.
- [23] The developmental onset of symbolic approximation: beyond nonsymbolic representations, the language of numbers matters / Iro Xenidou-Dervou, Camilla Gilmore, Menno van der Schoot, Ernest CDM van Lieshout // [Frontiers in psychology](#). — 2015. — Vol. 6. — P. 487.

- [24] 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.