DOI: 10.15514/ISPRAS-2026-38(6)-26



Автоматическая генерация тестов по состоянию исполнения с применением больших языковых моделей

[1] *М.А. Паршин, ORCID: 0009-0006-8486-3816 <mxprshn@yandex.ru>*
[2] *Д.С. Степанов, ORCID: 0000-0003-1719-0325 <stepanov@itmo.ru>*

[1] *Санкт-Петербургский государственный университет,*
Россия, 199034, г. Санкт-Петербург, Университетская наб., д. 7-9.

[2] *Университет ИТМО,*
Россия, 197101, г. Санкт-Петербург, Кронверкский пр., д. 49, лит. А.

Аннотация. В программных системах с длительным жизненным циклом, как правило, накапливается значительное количество высокоуровневых интеграционных тестов, которые исторически служат наиболее прагматичным средством проверки функциональности. Перспективным направлением представляется автоматическая генерация на их основе модульных тестов для отдельных компонентов («вырезка» модульных тестов, test carving), что позволяет повысить точность локализации дефектов и обеспечить быструю обратную связь при внесении изменений. В статье предложен метод автоматической генерации модульных тестов на основе состояния исполнения программы. В ходе выполнения существующего высокоуровневого теста фиксируются состояние целевого объекта, аргументы вызова, возвращаемый результат и взаимодействия с зависимостями. На основе собранных данных строится дескриптор исполнения, по которому затем синтезируется модульный тест: сначала детерминировано формируется предварительный код, после чего большая языковая модель выполняет его рефакторинг и стилистическую адаптацию. Итоговый код теста строится большой языковой моделью, что отличает предложенный метод от существующих инструментов вырезки модульных тестов. Описана реализация метода для JVM-программ на основе интерфейса отладчика JDI, ориентированная на работу с крупными проектами. Экспериментальная оценка проведена на 70 парах «тест/целевой метод» из семи открытых Java-проектов. Сравнивались тесты, построенные ИИ-агентом без данных исполнения, по дескриптору исполнения и путём рефакторинга предварительного кода, а также сам предварительный код, взятый в качестве итогового теста без рефакторинга. ИИ-агент с доступом к данным, собранным при помощи предложенного метода, построил модульные тесты, которые в среднем воспроизводят с теми же аргументами и результатом 86 % вызовов целевого метода из исходного теста, а при дополнительном учёте значений полей целевого объекта – 63%. Для тестов, сгенерированных агентом без доступа к данным исполнения, эти показатели составили 72% и 43% соответственно.

Ключевые слова: автоматическая генерация тестов; модульные тесты; большие языковые модели; отладочный интерфейс JDI; виртуальная машина JVM.

Для цитирования: Паршин М.А., Степанов Д.С. Автоматическая генерация тестов по состоянию исполнения с применением больших языковых моделей. Труды ИСП РАН, 2026, том 38, вып. 6, часть 2, стр. 67–92. DOI: 10.15514/ISPRAS-2026-38(6)-26.

Automatic Test Generation from Execution State Using Large Language Models

[1] *M.A. Parshin, ORCID: 0009-0006-8486-3816 <mxprshn@yandex.ru>*
[2] *D.S. Stepanov, ORCID: 0000-0003-1719-0325 <stepanov@itmo.ru>*

[1] *Saint Petersburg State University,*
7/9 Universitetskaya emb., St. Petersburg, 199034, Russia.

[2] *ITMO University,*
Kronverksky Pr. 49, bldg. A, St. Petersburg, 197101, Russia.

Abstract. In software systems with a long lifecycle, a substantial number of high-level integration and end-to-end tests tend to accumulate over time, historically serving as the most pragmatic means of verifying functionality. A promising direction is the automatic generation of unit tests for individual components based on these existing tests (unit test carving), which improves the precision of defect localization and provides rapid feedback when changes are introduced. In this paper, we propose a method for automatic unit test generation based on program execution state. During the execution of an existing high-level test, the target object state, call arguments, return value, and interactions with dependencies are captured. From the collected data, an execution descriptor is constructed and subsequently used to synthesize a unit test: first, preliminary code is deterministically generated, after which a large language model performs its refactoring and stylistic adaptation. Unlike existing test carving tools, the method relies on a large language model to produce the final test code, and its implementation targets large JVM projects. We describe an implementation of the method for JVM programs built on top of the JDI debugging interface. The method is evaluated on 70 pairs of a test and a target method from seven open-source Java projects. We compare tests built by an AI agent without execution data, from the execution descriptor and by refactoring the preliminary code, as well as the preliminary code itself taken as the final test without refactoring. An AI agent with access to the data collected by the proposed method built unit tests that, on average, reproduce 86% of the target method calls from the original test with the same arguments and result, and 63% when the values of the target object's fields are also taken into account. For tests generated by the agent without access to execution data, these figures are 72% and 43%, respectively.

Keywords: automatic test generation; unit tests; large language models; Java Debug Interface; JVM.

For citation: Parshin M.A., Stepanov D.S. Automatic Test Generation from Execution State Using Large Language Models. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 6, part 2, pp. 67-92 (in Russian). DOI: 10.15514/ISPRAS-2026-38(6)-26.

1. Введение

Тестирование является важным этапом разработки современных программных продуктов. Тесты обеспечивают качество программ, документируют их поведение и позволяют найти ошибки в коде. Различают тесты нескольких уровней: модульные (юнит-тесты) и интеграционные тесты [1]. Модульные тесты проверяют отдельный компонент в изоляции, интеграционные тесты – взаимодействие нескольких модулей, а E2E тесты – работу всей системы в целом. В то время как модульные тесты создаются разработчиками тестируемого модуля, за тесты более высоких уровней, как правило, отвечают инженеры по тестированию (Quality Assurance).

Преобладание интеграционных тестов над модульными является известной антипрактикой (antipattern): высокоуровневые тесты уступают модульным в точности локализации дефектов и потребляют больше ресурсов [2]. Однако в системах с длительным жизненным циклом такой дисбаланс распространён, поскольку в условиях унаследованной архитектуры высокоуровневые тесты остаются наиболее доступным средством верификации. Рациональной альтернативой служит набор легковесных модульных тестов, позволяющих выявлять дефекты до запуска ресурсоёмких тестов высокого уровня. В индустрии данная практика поддерживается мониторингом покрытия кода модульными тестами [3].

Создание модульных тестов требует от программиста значительного количества рутинной работы по определению тестовых сценариев и написанию однотипного кода. Существуют инструменты для автоматической генерации тестов, основанные на формальных методах, таких как символьное исполнение [4] и эволюционное тестирование [5], и на моделях искусственного интеллекта, в частности, на больших языковых моделях. Однако большинство инструментов генерирует модульные тесты в изоляции, используя в качестве контекста лишь тестируемый код или тестируемую сборку. Информация о реальных сценариях исполнения программы, которые происходят при запуске высокоуровневых тестов или «ручном» тестировании, не учитывается. Данные исполнения используются в методах «вырезки» модульных тестов (test carving): инструменты этого направления записывают состояние программы при выполнении системных тестов и строят по нему модульные тесты (см. разд. 7.2). Однако код теста в таких инструментах, как правило, строится детерминированно и имеет синтетический вид, тогда как большие языковые модели способны генерировать код, близкий к написанному разработчиками. Поэтому представляется целесообразным объединить вырезку модульных тестов и генерацию кода языковыми моделями, чтобы получать модульные тесты, работающие с реальными данными, которые не могут быть выведены одними лишь формальными анализаторами или языковыми моделями. При этом интеграционные тесты могут быть основой для множества модульных тестов, что способствует переходу к сбалансированной иерархии тестирования.

Результаты данной работы заключаются в следующем.

1. Предложен метод генерации модульных тестов по состоянию исполнения. Во время выполнения существующего высокоуровневого теста фиксируются состояние целевого объекта, аргументы вызова, результат выполнения и взаимодействия с зависимостями, а затем эти данные преобразуются в дескриптор исполнения. Код модульного теста синтезируется в два этапа: сначала по дескриптору детерминированно строится предварительный код теста, затем большая языковая модель выполняет его рефакторинг и стилистическую адаптацию. Рассматривается и вариант без предварительного кода, в котором языковая модель строит тест непосредственно по текстовому представлению дескриптора. Двухэтапная схема позволяет использовать фактические данные сценария исполнения и при этом получать код, близкий к принятому в проекте стилю.
2. Описана реализация метода для программ виртуальной машины Java (JVM) на основе интерфейса отладчика JDI. Для работы с промышленными проектами в реализации предусмотрен набор эвристик построения объектов, в том числе отдельные обработчики типовых библиотечных классов, а также правила выбора зависимостей, заменяемых «заглушками» (mocks).
3. Проведена экспериментальная оценка метода на 70 парах «тест/целевой метод» из семи открытых Java-проектов, включающих как серверные приложения, так и библиотеки. Сравнение четырёх режимов генерации позволяет отделить влияние данных исполнения, детерминированного построения предварительного кода и его рефакторинга языковой моделью. Код эксперимента, тексты запросов к модели и данные исполнения опубликованы [6].

Эксперимент показал, что с данными исполнения агент точнее воспроизводит значения, с которыми работает целевой метод. Тесты, построенные агентом по дескриптору исполнения, в среднем воспроизводят с теми же аргументами и результатом 86 % вызовов целевого метода из исходного теста, а при дополнительном учёте значений полей целевого объекта – 63 %. У тестов, сгенерированных агентом без доступа к данным исполнения, эти показатели

составили 72% и 43%. Способность тестов обнаруживать ошибки при этом сохраняется: тесты, построенные по дескриптору, обнаруживают 79% искусственно внесённых ошибок в исполняемом ими коде, а тесты без данных исполнения – 81%.

2. Обзор

2.1 Уровни тестирования

Кодовая база современных программных продуктов включает в себя множество компонентов, таких как классы и сервисы. Требуется тестировать как взаимодействие компонентов между собой, так и логику работы отдельных модулей в изоляции.

Выделяют интеграционные и модульные тесты.

Интеграционные тесты подразумевают запуск всей системы в реальных (или приближённых к реальным) условиях и взаимодействие с ней по внешним интерфейсам. Тесты данного типа гарантируют корректность работы программы с точки зрения пользователя, но являются ресурсозатратными. Они требуют значительных вычислительных мощностей на развёртывание окружения и исполняются продолжительное время. Функциональные тесты проводятся инженерами по тестированию «вручную» или автоматизируются при помощи скриптов для эмуляции взаимодействия с интерфейсом [7].

Интеграционные тесты также проверяют взаимодействие модулей, но, в отличие от функциональных тестов, изолируют определённое подмножество компонентов и симулируют остальное окружение с помощью «заглушек» (mocks) [8]. При интеграционном тестировании в большей степени верифицируются внутренняя логика системы и внутренние интерфейсы. Создание интеграционных тестов является ответственностью как инженеров по тестированию, так и разработчиков.

Модульные тесты изолируют тестируемый компонент целиком, полностью симулируя все зависимости, и не требуют длительного времени на исполнение. Модульные тесты создаются самими разработчиками модуля и могут многократно запускаться, чтобы проверить корректность вносимых в него изменений. Поскольку модульные тесты фиксируют и документируют поведение компонентов, вводятся метрики, измеряющие полноту и качество тестового набора [3]. Такими метриками являются, например, тестовое покрытие и доля обнаруженных мутантов [9].

Целесообразно поддерживать различное количество тестов каждого типа исходя из их назначения и сложности запуска. Так, наличие обширного набора модульных тестов для каждого модуля системы позволяет эффективно обнаруживать регрессии. При неуспешном прохождении модульного теста ошибка может быть быстро локализована и воспроизведена разработчиком. Множество более ресурсоёмких интеграционных и функциональных тестов, напротив, уместно поддерживать ограниченным. Данный подход к количественному распределению тестов называется пирамидой тестирования [1]. Противоположная ситуация, при которой большинство тестов является высокоуровневым, часто классифицируется как антипрактика (antipattern) под названием «рожок мороженого» (testing ice-cream cone).

2.2 Факторы, затрудняющие модульное тестирование

Проблемы тестовых данных и тестового оракула. Процесс разработки модульных тестов включает два ключевых этапа: определение тестовых данных и формулирование тестового оракула, то есть условия, по которому тест признаётся успешно пройденным. И то и другое требует от разработчика значительных усилий.

Подготовка тестовых данных предполагает не только выбор типичных входов, но и систематическое покрытие краевых случаев. Определение таких случаев сложно формализуется и зависит от опыта разработчика, понимания им внутренней логики

компонента и потенциальных неявных ограничений. Кроме того, создание объектов, необходимых для тестового окружения, зачастую связано с написанием большого количества шаблонного (boilerplate) кода, что особенно характерно для объектно-ориентированных систем.

Составление тестового оракула представляет собой самостоятельную трудную задачу. Согласно исследованиям тестовых оракулов, основная сложность заключается в том, что корректное ожидаемое поведение компонента не всегда может быть выражено в виде простого сравнения значений; разработчику требуется выделить стабильные инварианты и свойства, которые должны сохраняться при различных входах. Это особенно проблематично в условиях эволюционирующего или плохо документированного кода, где фактическое поведение системы известно лишь частично. В таких случаях возникает так называемая проблема оракула: невозможно однозначно определить, что должно считаться корректным результатом. Тестировщик вынужден либо полагаться на неформальные знания, либо ориентироваться на текущее (возможно, уже деградировавшее) поведение системы [10].

Отдельного внимания заслуживает ситуация с «унаследованным» кодом (legacy). В больших зрелых системах отсутствуют чёткие спецификации, исходное проектное намерение может быть утрачено, а поведение модуля определяется совокупностью исторических решений. В связи с этим разработчикам трудно как подобрать показательные данные для тестов, так и сформировать корректный оракул. В результате тесты либо становятся чрезмерно хрупкими, фиксируя текущее состояние системы вплоть до деталей реализации, либо, напротив, слишком слабыми и неспособными обнаруживать регрессии.

Эти трудности подтверждены многочисленными исследованиями, которые выделяют подбор данных и формулировку оракулов как один из наиболее трудоёмких этапов тестирования, на котором часто допускаются ошибки [10–12]. В контексте данной работы именно эти проблемы мотивируют разработку методов, использующих информацию о фактических запусках программы: реальное состояние исполнения позволяет автоматически извлекать репрезентативные тестовые данные и выводить корректные тестовые оракулы, отражающие подтверждённое поведение системы.

Экономические и организационные ограничения. В промышленных условиях процесс тестирования определяется не только техническими соображениями, но и бизнес-требованиями: ограниченными бюджетами, жёсткими сроками поставки и давлением на скорость вывода функциональности на рынок. Эмпирические исследования показывают, что давление сроков (time pressure), то есть ощущение недостатка времени для выполнения задачи, повсеместно встречается в программной инженерии и напрямую связано со сжатыми или чрезмерно оптимистичными графиками проектов [13]. В таких условиях разработчики и тестировщики вынуждены искать баланс между глубиной проверки и соблюдением сроков, что часто приводит к сокращению объёма автоматизированного тестирования, упрощению сценариев или частичному отказу от низкоуровневых тестов в пользу быстрых поверхностных проверок.

Одновременно крупные технологические компании декларируют высокий приоритет тестирования и вводят формальные требования к метрикам качества, таким как покрытие кода и доля обнаруженных мутантов. В ряде индустриальных отчётов и обзоров отмечается, что на тестирование может приходиться порядка четверти общего бюджета разработки, а компании инвестируют значительные ресурсы в автоматизацию и инфраструктуру тестов [14].

В результате на практике возникает противоречие: с одной стороны, индустрия осознаёт важность систематического тестирования и вводит формальные требования, с другой – бизнес-ограничения и хроническое давление сроков мешают выстраивать устойчивые процессы разработки и сопровождения тестов. Это делает особенно актуальными методы,

которые снижают трудозатратность создания тестов и позволяют улучшать качество тестового набора без существенного увеличения издержек на стороне команд разработки.

3. Метод

В данном разделе описывается метод автоматического получения модульного теста по фактическому (наблюдаемому) сценарию исполнения программы. В частности, источником такого сценария исполнения может быть тест более высокого уровня, например интеграционный тест.

3.1 Высокоуровневое описание метода

Предполагается объектно-ориентированная среда, в которой существуют экземпляры классов (объекты), обладающие состоянием (полями) и поведением (методами).

Задачей описываемого метода является генерация модульного теста, который воспроизводит поведение целевой функции, наблюдаемое при её реальном исполнении. Такой тест должен восстанавливать исходное состояние целевого объекта, значения аргументов, возвращаемое значение и, при необходимости, взаимодействия с внешними зависимостями.

Для построения теста требуется собрать *дескриптор исполнения*, представляющий собой структурированное описание вызова целевого метода. Дескриптор фиксирует данные, достаточные для последующего синтеза теста: исходное состояние целевого объекта, аргументы вызова, результат выполнения и информацию о вызовах зависимостей, которые должны быть воспроизведены или замещены в тестовом окружении.

Метод представляет собой последовательность трёх этапов (рис. 1). На *этапе наблюдения* перехватываются события сценария исполнения программы, относящиеся к целевому вызову и связанным с ним обращениям к зависимостям. Затем на *этапе формализации* из этих событий извлекаются значения объектов и инкрементально строится дескриптор исполнения. После завершения выполнения целевого метода начинается *этап синтеза*, на котором построенный дескриптор преобразуется в исходный код модульного теста. Этап синтеза включает два шага: сначала детерминировано строится предварительный код теста, затем большая языковая модель выполняет его рефакторинг и стилистическую адаптацию.

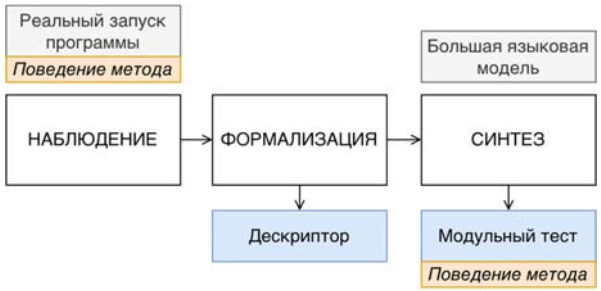


Рис. 1. Высокоуровневая схема метода.
Fig. 1. High-level overview of the method.

3.2 Этап наблюдения

Этап наблюдения является первой стадией метода и предназначен для фиксации тех событий сценария исполнения, на основе которых далее будет построен дескриптор исполнения. На этом этапе одновременно с работой программы регистрируются входы в методы и выходы из них; построение дескриптора при этом происходит шаг за шагом по мере поступления соответствующих событий.

Набор методов, события для которых подлежат перехвату, может изменяться динамически по ходу наблюдения. Например, в начале достаточно перехватывать вызов целевого метода и конструктора объекта, на котором он выполняется. Затем, когда становятся известны аргументы целевого вызова и связанные с ними зависимости, в этот набор могут быть добавлены дополнительные методы, чьё поведение требуется зафиксировать для последующего воспроизведения в тесте.

Таким образом, для реализации этапа наблюдения требуется инструментальная надстройка над программой, предоставляющая следующий интерфейс:

- `enableIntercept(methodId)` – включить перехват вызовов метода с идентификатором `methodId`;
- `onInterceptEntry(callId, methodId, this, arguments)` – обработчик события входа в метод;
- `onInterceptExit(callId, methodId, returnValue)` – обработчик события выхода из метода.

На этапе формализации события, зафиксированные в ходе наблюдения, преобразуются в структурированное представление, пригодное для последующего синтеза теста. Общая схема этого преобразования приведена на рис. 2. Для его описания требуется ввести модель данных, задающую способ представления объектов, вызовов методов и итогового дескриптора исполнения.

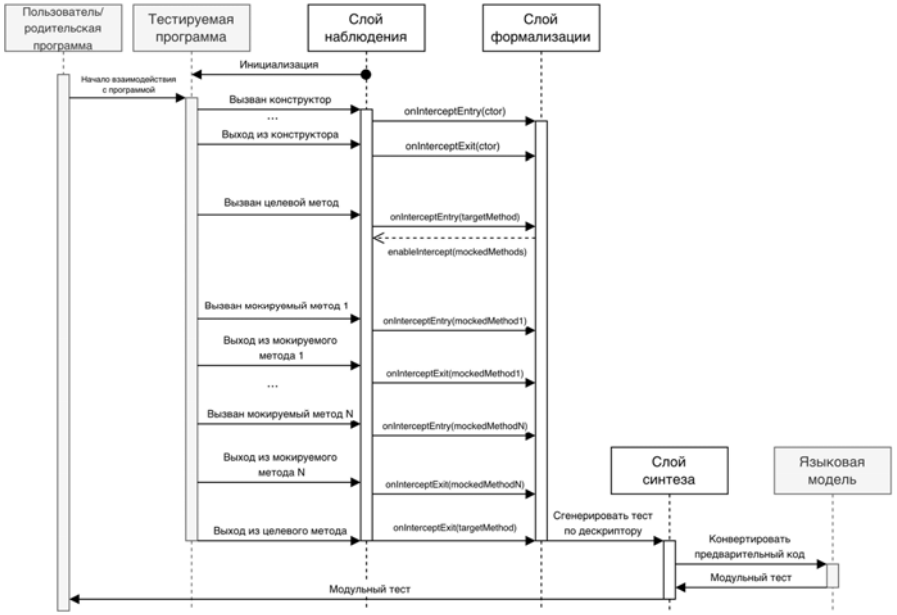


Рис. 2. Схема взаимодействия этапов наблюдения и формализации.
Fig. 2. Interaction between the observation and formalization stages.

3.3 Модель формализации

PrimitiveValue – скалярное примитивное значение:
PrimitiveValue ::= int | bool | string | ...

Данный тип соответствует примитивным типам языка программирования и не имеет внутренней объектной структуры.

ObjectDescriptor – дескриптор объекта (унифицированное представление как примитивов, так и составных объектов):

- `id: string` – идентификатор объекта (например, адрес или хеш), позволяющий различать объекты и отслеживать повторные ссылки;
- `fields: Map<string, ObjectDescriptor>` – отображение имён полей в их дескрипторы; используется для составных объектов;
- `primitiveValue: PrimitiveValue?` – значение примитива; задано тогда и только тогда, когда объект является примитивом;
- `mockedCalls: CallDescriptor[]` – журнал (список) дескрипторов вызовов данного объекта; используется, если объект заменяется в тесте «заглушкой» и требуется воспроизвести его наблюдаемое поведение.

CallDescriptor – дескриптор вызова метода:

- `methodId: string` – идентификатор метода (например, полное имя: класс/сигнатура);
- `arguments: ObjectDescriptor[]` – дескрипторы фактических аргументов;
- `returnValue: ObjectDescriptor?` – дескриптор возвращаемого значения; равен `null` для процедур/методов без возвращаемого значения.

ResultDescriptor – итоговый дескриптор, являющийся результатом работы алгоритма и входом для генератора тестов:

- `this: ObjectDescriptor` – дескриптор целевого объекта, на котором выполняется целевой метод;
- `constructorCall: CallDescriptor` – сведения о конструкторе (или фабричном методе), с помощью которого создан целевой объект в реконструируемом сценарии;
- `targetMethodCallDescriptor: CallDescriptor` – дескриптор вызова целевого метода (аргументы и результат).

3.4 Алгоритм формализации

В листинге 1 представлен алгоритм формализации. Он принимает на вход события, инициированные на этапе наблюдения, и по мере их поступления итеративно формирует дескриптор исполнения `ResultDescriptor`.

Перед началом работы активируется перехват для целевого метода и конструкторов его класса. Идентификаторы `TARGET_CALL_ID` и `TARGET_CTOR_CALL_ID` обозначают соответственно вызов целевого метода и вызов конструктора, создавшего целевой объект, а `CTOR_METHOD_ID` и `TARGET_METHOD_ID` – идентификаторы самих этих методов. Далее события `onInterceptEntry` и `onInterceptExit`, возникающие на этапе наблюдения, сопоставляются по идентификатору вызова `callId` и используются для поэтапного заполнения соответствующих `CallDescriptor`. При обработке входа в метод фиксируются аргументы вызова и определяется, относится ли событие к целевому методу, конструктору целевого объекта или к взаимодействию с ранее выделенной зависимостью; при обработке выхода фиксируется возвращаемое значение.

Если в ходе обработки целевого вызова эвристика `shouldMock`/`determineMockedCalls` выявляет зависимость, нежелательную для реального сценария исполнения модульного

теста, для неё создаётся «заглушка» и дополнительно активируется перехват её методов. Благодаря этому в итоговом дескрипторе сохраняются не только аргументы и результат целевого метода, но и наблюдаемое поведение зависимостей, которое затем может быть воспроизведено в тестовом окружении.

```
Глобальные структуры:
resultDescriptor ← ResultDescriptor()
resultDescriptor.constructorCall ← CallDescriptor(CTOR_METHOD_ID)
resultDescriptor.targetMethodCallDescriptor ← CallDescriptor(TARGET_METHOD_ID)
mockObjects ← ∅  ▷ Map<objectId, ObjectDescriptor>
activeCalls ← ∅  ▷ Map<callId, CallDescriptor>

onInterceptEntry(callId, methodId, this, arguments)
    isTargetCall ← (callId = TARGET_CALL_ID)
    isCtorCall ← (callId = TARGET_CTOR_CALL_ID)
    if isTargetCall:
        resultDescriptor.this ← this.copy()
        for each (fieldName, fieldValue) ∈ resultDescriptor.this.fields:
            if shouldMock(fieldValue):
                mockedField ← fieldValue.copy()
                mockObjects[fieldValue.id] ← mockedField
                mockedMethods ← determineMockedCalls(fieldValue)
                for each m ∈ mockedMethods:
                    enableIntercept(m)
                resultDescriptor.this.fields[fieldName] ← mockedField
        callDescriptor ← resultDescriptor.targetMethodCallDescriptor
    else if isCtorCall:
        callDescriptor ← resultDescriptor.constructorCall
    else:
        mockDescriptor ← mockObjects[this.id]
        callDescriptor ← CallDescriptor(methodId)
        mockDescriptor.mockedCalls += callDescriptor
    activeCalls[callId] ← callDescriptor
    if isTargetCall or isCtorCall:

        for each arg ∈ arguments:
            if shouldMock(arg):
                mockedArg ← arg.copy()
                mockObjects[arg.id] ← mockedArg
                mockedMethods ← determineMockedCalls(arg)
                for each m ∈ mockedMethods:
                    enableIntercept(m)
                callDescriptor.arguments += mockedArg
            else:
                callDescriptor.arguments += arg

onInterceptExit(callId, methodId, returnValue)
    callDescriptor ← activeCalls[callId]
    callDescriptor.returnValue ← returnValue
    delete activeCalls[callId]
```

Листинг 1. Алгоритм формализации.
Listing 1. Formalization algorithm.

Вспомогательные структуры `activeCalls` и `mockObjects` используются соответственно для сопоставления незавершённых вызовов и накопления сведений о «заглушках». Помимо возвращаемого значения, дескриптор вызова может содержать исключение, которым завершился вызов, а итоговый дескриптор – результаты статических вызовов, выявленных статическим анализом (см. разд. 4.1). Аргументы вызовов зависимостей в реализации не сохраняются: «заглушка» воспроизводит последовательность возвращаемых значений.

3.5 Синтез кода

Входными данными этапа синтеза является дескриптор исполнения, построенный на предыдущем этапе. Он содержит сведения, необходимые для воспроизведения зафиксированного сценария в виде модульного теста: состояние целевого объекта, аргументы вызова, результат выполнения, а также, при необходимости, информацию о взаимодействиях с зависимостями.

На первом шаге по дескриптору исполнения детерминированно строится предварительный код теста, в котором восстанавливаются создание объектов, подготовка аргументов, вызов целевого метода, конфигурирование «заглушек» и проверки результата. На втором шаге предварительный код передаётся большой языковой модели для рефакторинга и стилистической адаптации. Такое разделение позволяет, с одной стороны, жёстко связать семантику теста с зафиксированным сценарием исполнения, а с другой – привести итоговый код к форме, удобной для чтения и сопровождения.

3.6 Ограничения и допущения

Для упрощения изложения алгоритм в листинге 1 описан для одного вызова целевого метода. Если в ходе сценария целевой метод вызывается несколько раз, для каждого вызова строится отдельный `ResultDescriptor`; в реализации фиксируется не более 100 таких вызовов. Вызов, начатый во время выполнения другого вызова целевого метода, не фиксируется.

Кроме того, вызовы конструкторов отслеживаются только для целевого объекта. Остальные объекты восстанавливаются по значениям их полей (см. разд. 4.2), поскольку отслеживание всех конструкторов требует слишком больших вычислительных затрат.

В модели формализации (см. разд. 3.3) объект представлен графом именованных полей. Массивы и стандартные коллекции JDK (списки, множества, очереди, словари) в реализации представляются последовательностями элементов и восстанавливаются в тесте через их публичный интерфейс.

4. Реализация для JVM

В данном разделе описывается реализация предложенного метода для языков, исполняемых на виртуальной машине Java (JVM), таких как Java и Kotlin.

4.1 Реализация этапа наблюдения

Для реализации этапа наблюдения использовался интерфейс *Java Debug Interface* (JDI) [15]. Он позволяет программно устанавливать точки останова, получать события выполнения и извлекать значения из текущего кадра стека, в том числе `this`, аргументы вызова и возвращаемое значение. В качестве альтернатив рассматривались интерфейс `XDebugger` [16] (доступен в API разработки плагинов для IntelliJ IDEA) и `JVMTI` [17] (предназначен для низкоуровневого инструментирования JVM из машинного кода). В реализации был выбран JDI, поскольку он позволяет построить событийную модель `onInterceptEntry`/`onInterceptExit` непосредственно, без разработки отдельного агента на языке C++.

В терминах JDI операция `enableIntercept(methodId)` соответствует программной установке точек останова в начале метода и во всех точках его завершения, включая завершение по исключению; обработчики `onInterceptEntry` и `onInterceptExit` реализуются на основе соответствующих событий срабатывания этих точек. Дополнительно в реализации учитывается тождественность экземпляров, что позволяет корректно связывать вызов конструктора с целевым объектом.

Ключевой инженерной проблемой является число срабатываний точек останова. Для снижения накладных расходов при перехвате вызовов «заглушённых» методов используется дополнительная фильтрация по получателю (`this`): точки настроены так, чтобы срабатывать только при вызове метода на требуемом экземпляре. Это позволяет исключить нерелевантные вызовы тех же методов на других объектах и уменьшить объём событий, передаваемых на этап формализации.

Дополнительно поддерживаются *статические вызовы*, не имеющие получателя `this`. Для их воспроизведения модель дескрипторов была расширена списком `staticCalls: CallDescriptor[]`. Релевантные статические вызовы выявляются статическим анализом кода целевого метода, при этом глубина обхода графа вызовов эвристически ограничена.

4.2 Дополнительные стратегии построения объектов

В реализации применяется набор эвристик, упрощающих работу с объектами сложной структуры, характерными для реальных JVM-приложений. К таким эвристикам относятся специализированные обработчики для типовых библиотечных классов, а также сохранение (serialization) и восстановление (deserialization) объектов со стабильным внешним представлением. Зависимости, нежелательные для реального сценария исполнения в тесте, заменяются «заглушками». В общем случае объект восстанавливается через подходящий конструктор или, при необходимости, через рефлексию. Используемые способы создания объектов различных типов сведены в табл. 1.

Табл. 1. Классификация объектов JVM и способы их создания в тесте.
Table 1. Classification of JVM objects and their instantiation in tests.

Тип объекта	Условие	Способ создания	Пример
Библиотечный объект с отдельным обработчиком	Тип входит в множество отдельно обрабатываемых типов	Конструктор или фабричный метод (задан заранее)	`LocalDateTime`, `BigDecimal`, `UUID`
Сохраняемый объект	Объект может быть сохранён/восстановлен одной из доступных библиотек, но слишком сложен или не может быть создан конструктором	Восстановление из сохранённого представления	Содержание HTTP-запросов
Сложный объект, заменяемый «заглушкой»	Эвристика (выделение сложных сервисов и репозиториев)	«Заглушка», созданная с помощью библиотеки «заглушек»	Сложные пользовательские сервисы, репозитории
Остальные объекты	–	Конструктор (выбирается эвристикой) или рефлексия	POJO

Отдельные обработчики для типовых библиотечных классов. Для часто встречающихся типов используются специализированные обработчики, обеспечивающие компактное представление в дескрипторе и воспроизводимое создание объекта в тесте. К ним относятся стандартные библиотечные классы JVM, например `LocalDateTime`, `BigDecimal`, `UUID` и `Optional`, а также классы Spring Framework [18].

Сохраняемые объекты. Если объект может быть сохранён в текстовом виде одной из доступных библиотек, то в тесте он восстанавливается из сохранённого представления. Эта

эвристика особенно полезна для Web-приложений, где аргументы и результаты часто уже имеют стандартное внешнее представление, например на языке JSON.

Объекты, подлежащие замене на «заглушки»: их участие в реальном сценарии исполнения теста нежелательно или трудно воспроизводимо, и поэтому они моделируются при помощи «заглушек». К этой категории, в частности, относятся сложные сервисы, обращения к внешним базам данных и пр.

Общий случай: конструктор или рефлексия. Если перечисленные выше эвристики не применимы, предпринимается попытка выбрать подходящий конструктор объекта. Приоритет отдаётся конструкторам с ограниченным числом параметров, однозначно инициализирующим поля объекта. Если такой конструктор не найден, объект создаётся с использованием прикладного интерфейса *Reflection API* [19] с последующей инициализацией его полей.

4.3 Реализация этапа синтеза

Этап синтеза в реализованном прототипе разделён на два последовательно выполняемых шага. На первом шаге по дескриптору исполнения детерминированно строится предварительный код теста, воспроизводящий наблюдённый сценарий на уровне программных конструкций. На этом шаге восстанавливаются создание или получение целевого объекта, подготовка аргументов, вызов целевого метода, конфигурирование «заглушек» и проверки результата. Для различных типов сценариев используются разные шаблоны тестового окружения, в частности отдельно обрабатываются обычные модульные тесты и сценарии, ориентированные на веб-обёртку тестируемого приложения.

Предварительный код строится не напрямую как текст, а через промежуточное представление теста. Оно фиксирует создаваемые объекты, вызовы конструкторов и методов, присваивания полям, статические вызовы, «заглушки» и утверждения. Перед генерацией итогового текста к этому представлению применяется набор преобразований, устраняющих мёртвые фрагменты, нормализующих вызовы и упрощающих вспомогательные конструкции, возникающие при восстановлении состояния объектов.

В листинге 2 приведён предварительный код, построенный для метода `updateInstanceConfig` класса `InstanceService` проекта Apollo [20].

```
@Test
void updateInstanceConfigReproducedTest() {
    InstanceConfig v = new InstanceConfig();
    v.setId(8L);
    v.setInstanceId(1L);
    v.setConfigClusterName("someConfigClusterName");
    v.setReleaseKey("anotherReleaseKey");
    Date v1 = new Date(1791727201772L);
    v.setDataChangeCreatedTime(v1);
    v.setConfigAppId("someConfigAppId");
    v.setConfigNamespaceName("someConfigNamespaceName");
    v.setDataChangeLastModifiedTime(v1);
    InstanceRepository v2 = Mockito.mock(InstanceRepository.class);
    InstanceConfigRepository mocked =
Mockito.mock(InstanceConfigRepository.class);
    Mockito.when(mocked.findById(Mockito.<Object>any()))
        .thenReturn((Optional<Object>) (Object) Optional.of(v));
    Mockito.when(mocked.save(Mockito.<Object>any())).thenReturn(v);
    InstanceService v3 = new InstanceService(v2, mocked);
    ReflectionTestUtils.setField(v3, "instanceRepository", v2);
    ReflectionTestUtils.setField(v3, "instanceConfigRepository", mocked);
    InstanceConfig v4 = v3.updateInstanceConfig(v);
    Assertions.assertNotNull(v4);
    Assertions.assertEquals(1L, v4.getInstanceId());
    Assertions.assertEquals("someConfigClusterName", v4.getConfigClusterName());
```

```
    Assertions.assertEquals("anotherReleaseKey", v4.getReleaseKey());
    Assertions.assertNotNull(v4.getDataChangeCreatedTime());
    Assertions.assertEquals("someConfigAppId", v4.getConfigAppId());
    Assertions.assertEquals("someConfigNamespaceName",
v4.getConfigNamespaceName());
}
```

Листинг 2. Пример предварительного кода теста.
Listing 2. Example of preliminary test code.

Целевой объект и зафиксированный аргумент восстанавливаются вызовами конструкторов и методов установки полей, репозитории заменяются «заглушками» с наблюдаемым поведением, а проверки сравнивают результат с зафиксированными значениями.

На втором шаге большая языковая модель выполняет рефакторинг предварительного кода. Языковая модель не строит тест с нуля, а преобразует уже сгенерированный код: улучшает именование, упрощает структуру и приводит текст к стилю, принятому в проекте. Запрос к модели требует сохранить семантику теста: все вызовы целевого метода, зафиксированные значения, поведение «заглушек» и проверки. Рефлексия заменяется вызовами публичного интерфейса класса там, где они задают то же состояние объекта, а новые проверки допускаются только для уже зафиксированных значений.

Наряду с двухэтапной схемой возможна генерация теста языковой моделью непосредственно по дескриптору исполнения. В этом случае дескриптор каждого зафиксированного исполнения целевого метода передаётся модели в текстовом виде, в формате JSON. Он содержит состояние целевого объекта, аргументы, результаты вызовов зависимостей, а также возвращённое значение или выброшенное исключение.

В листинге 3 приведён тест, полученный из листинга 2 в ходе эксперимента (см. разд. 5). Языковая модель заменила синтетические имена переменных содержательными, удалила лишние приведения типов и обращения к рефлексии.

```
@Test
void updatesExistingInstanceConfig() {
  InstanceConfig instanceConfig = new InstanceConfig();
  instanceConfig.setId(8L);
  instanceConfig.setInstanceId(1L);
  instanceConfig.setConfigClusterName("someConfigClusterName");
  instanceConfig.setReleaseKey("anotherReleaseKey");
  Date changeTime = new Date(1791727201772L);
  instanceConfig.setDataChangeCreatedTime(changeTime);
  instanceConfig.setConfigAppId("someConfigAppId");
  instanceConfig.setConfigNamespaceName("someConfigNamespaceName");
  instanceConfig.setDataChangeLastModifiedTime(changeTime);

  InstanceRepository instanceRepository = Mockito.mock(InstanceRepository.class);
  InstanceConfigRepository instanceConfigRepository =
      Mockito.mock(InstanceConfigRepository.class);
  Mockito.when(instanceConfigRepository.findById(Mockito.anyLong()))
      .thenReturn(Optional.of(instanceConfig));
  Mockito.when(instanceConfigRepository.save(Mockito.any(InstanceConfig.class)))
      .thenReturn(instanceConfig);

  InstanceService instanceService =
      new InstanceService(instanceRepository, instanceConfigRepository);

  InstanceConfig updatedConfig =
instanceService.updateInstanceConfig(instanceConfig);

  assertNotNull(updatedConfig);
  assertEquals(1L, updatedConfig.getInstanceId());
  assertEquals("someConfigClusterName", updatedConfig.getConfigClusterName());
  assertEquals("anotherReleaseKey", updatedConfig.getReleaseKey());
  assertNotNull(updatedConfig.getDataChangeCreatedTime());
  assertEquals("someConfigAppId", updatedConfig.getConfigAppId());
```



```
  assertEquals("someConfigNamespaceName", updatedConfig.getConfigNamespaceName());
}
```

Листинг 3. Тест из листинга 2 после рефакторинга языковой моделью.
Listing 3. The test from Listing 2 after refactoring by the language model.

5. Эксперименты

Экспериментальная оценка направлена на проверку того, насколько данные исполнения, полученные предложенным методом, позволяют строить модульные тесты, воспроизводящие поведение программы при выполнении высокоуровневых тестов. Кроме того, оценивалось влияние отдельных частей этапа синтеза: детерминированного построения предварительного кода и его рефакторинга языковой моделью. Оценка проводилась по четырём исследовательским вопросам:

- **RQ1.** Насколько точно сгенерированные модульные тесты воспроизводят наблюдаемое исполнение целевого метода?
- **RQ2.** Обнаруживают ли сгенерированные тесты ошибки в исполняемом ими коде?
- **RQ3.** Каков вклад детерминированного построения предварительного кода и его рефакторинга языковой моделью в качество итогового теста?
- **RQ4.** Каково качество кода сгенерированных тестов?

5.1 Экспериментальная установка

Набор данных. Эксперименты проводились на семи открытых Java-проектах (см. табл. 2). Проекты выбирались по следующим критериям: наличие собственного набора тестов, включающего тесты выше модульного уровня, а также воспроизводимость сборки на зафиксированной версии без внешней инфраструктуры. Кроме того, выбирались проекты из разных предметных областей и с разной архитектурой – последнее особенно важно, поскольку от архитектуры зависят набор зависимостей, заменяемых «заглушками», а также сложность восстанавливаемых объектов.

Табл. 2. Проекты экспериментального набора.
Table 2. Subject projects.

Проект	Назначение	Версия (коммит)	Кол-во пар (прямой вызов)	Кол-во пар (косвенный вызов)	Общее кол-во пар
Apollo [20]	Web-приложение для управления конфигурацией	998b88f	6	9	15
Klaw [21]	Web-приложение для управления Apache Kafka	99f8953	6	5	11
MyBatis 3 [22]	Библиотека объектно-реляционного отображения	673798b	4	8	12
jsoup [23]	Библиотека разбора и обработки HTML	7860d08	4	7	11
Checkstyle [24]	Статический анализатор Java-кода	47b4b77	4	5	9
JGit [25]	Реализация системы контроля версий Git	226017b	5	4	9
Liquibase [26]	Миграции схемы базы данных	f6ddd56	0	3	3
Всего			29	41	70

Элементом набора данных служит пара из теста и целевого метода, исполняемого при выполнении этого теста. Целевым методом может служить метод, не являющийся конструктором или методом доступа к полю и имеющий зависимости от других классов проекта. Пары разделены на две группы в зависимости от того, как тест обращается к целевому методу. В парах с прямым вызовом тест вызывает целевой метод непосредственно, а в парах с косвенным вызовом целевой метод вызывается через цепочку вызовов, и его фактические аргументы в тексте теста отсутствуют. Всего набор содержит 70 пар: 29 – с прямым вызовом и 41 – с косвенным.

Объём набора был ограничен длительностью и стоимостью эксперимента.

Режимы генерации. Для каждой пары модульные тесты строились в четырёх режимах, различающихся данными исполнения, которые получает агент:

- базовый режим: агент получает только код проекта и исходный тест, данные исполнения не используются;
- генерация по дескриптору: агент дополнительно получает дескриптор каждого зафиксированного исполнения целевого метода в формате JSON;
- рефакторинг предварительного кода: агент получает предварительный код теста, построенный по дескриптору, и выполняет его рефакторинг без изменения семантики (см. разд. 4.3);
- предварительный код без рефакторинга: предварительный код используется в качестве итогового теста без изменений.

Агент и условия генерации. Тесты строились ИИ-агентом Pi версии 0.84.1 [27] с языковой моделью GPT-6.1 Sol компании OpenAI [28] при среднем уровне рассуждений (medium). Параметр `temperature` не задавался, поскольку для рассуждающих моделей он не поддерживается. Агент Pi был выбран, поскольку он является открытым и минималистичным: его конфигурация и системные запросы легко переопределяются. Это позволило создать для эксперимента ограниченную среду с заданным набором инструментов и команд.

Во всех режимах агент получал одинаковую постановку задачи: построить изолированный модульный тест, воспроизводящий исполнения целевого метода в ходе исходного теста. Запросы различались только блоком, описывающим данные исполнения (листинг 4).

```
# Общая часть
Repository: {{repository}} at commit {{commit}}
Language: {{language}}, JDK: {{jdk}}, working directory: {{working_directory}}

Target method:
  class: {{target_class}}
  method: {{target_method}}
  signature: {{target_signature}}

Driving scenario: {{scenario_description}}
Scenario test source: {{scenario_source_paths}}

Task: write one isolated unit-test class at {{output_test_file}} that reproduces
the executions of the target method that occur when the driving scenario runs. For
each execution reproduce the state of the receiver object, the arguments, the
interactions of the target method with its collaborators, and the outcome (the
returned value or the thrown exception).

Write one test method per distinct execution. Executions with the same receiver
state, arguments, collaborator interactions and outcome count as one.

Requirements for the test:
- it calls the target method directly and does not run the driving scenario;
- it does not start an application context, a database, a network server or other
infrastructure;   collaborators that depend on such infrastructure are replaced
with mocks that behave as in the scenario;
- it checks the outcome of every execution and, where relevant, the state after
```



```
the call.

Available commands (argv arrays), run them with the bash tool exactly as written:
{{commands}}

Rules:
- do not modify production code, existing tests, test resources, build files or
dependencies;

- do not add dependencies, logging or instrumentation;
- existing mocking libraries may be used;
- helper and inner classes inside the generated class are allowed;
- compile and run the generated class and repair only that class until it passes.

# Базовый режим
Condition: source code only.

No execution data is provided.

# Генерация по дескриптору
Condition: execution dump.

A dump of the target method executions recorded while the driving scenario ran is
provided as one JSON file per recorded execution in {{execution_input}}, in
recording order.

For each recorded execution it contains the receiver state, the arguments, the
calls to collaborators with their arguments and results, and the outcome. Use the
recorded values as test data and expected results; do not replace them with
invented values.

# Рефакторинг предварительного кода
Condition: preliminary test.

A preliminary test class generated deterministically from a recording of the
driving scenario is provided at:
{{execution_input}}

It already reproduces the recorded executions of the target method: object
construction, arguments, mock configuration, the target call and checks. Turn it
into the final test class at {{output_test_file}} without changing what the tests
do:
- give tests and variables meaningful names;
- remove redundant code such as unnecessary casts and fully qualified names;
- replace reflection with the public API where the public API sets the same state;
- extract shared setup and follow the style of the existing tests of the project.

Do not change the semantics of the preliminary test: keep every target call, every
recorded value, the behaviour of every mock and every check; do not remove checks
and do not add or remove test methods. You may add checks only on values already
recorded in the preliminary test (for example the returned value, the state after
the call, or the interactions with mocks configured there). If the preliminary
test does not compile or fails, make only the minimal change needed to make it
compile and pass, keeping the recorded values.
```

Листинг 4. Запросы к модели: общая часть и блоки режимов (приводятся в исходном виде).
Listing 4. Prompts: common part and mode blocks.

Системный запрос, задающий ограничения рабочей среды агента, приведён в репозитории эксперимента [6].

Каждый запуск агента выполнялся в новой копии проекта на зафиксированной версии, из которой удалены существующие модульные тесты класса целевого метода, кроме исходного теста. Агент мог читать файлы проекта, изменять только файл создаваемого теста и выполнять три команды: сборку проекта, запуск исходного теста и запуск созданного теста. Время одного запуска агента ограничено 10 минутами.

Данные исполнения собирались один раз до генерации и передавались агенту в виде файла, поэтому входные данные агента в обоих прогонах совпадают. Режимы с участием языковой модели выполнялись в двух запусках, и результаты усреднялись по ним. Предварительный

код строится детерминированно, поэтому без рефакторинга он оценивался однократно, без повторных запусков.

Применимость. Для 7 из 70 пар реализации метода не удалось получить дескриптор исполнения, а для 20 пар – построить предварительный код. Отказы связаны с выходом за лимит времени и внутренними ошибками реализации. Далее результаты приводятся по 63 парам для генерации по дескриптору и по 50 парам для режимов с предварительным кодом. Каждый режим сравнивается с базовым на тех же парах (см. табл. 3–6).

Доступность кода и данных. Код реализации метода, выполняющей сбор данных исполнения и построение предварительного кода, закрыт. Код эксперимента опубликован в открытом репозитории [6]. Репозиторий содержит код запуска агента и вычисления метрик, тексты запросов к модели, а также записанные данные исполнения и предварительный код для всех пар.

5.2 RQ1. Воспроизведение наблюдаемого поведения

Для ответа на первый исследовательский вопрос сравнивались трассы исполнения целевого метода в исходном тесте и в сгенерированном модульном тесте.

Для независимого сравнения трасс использовался инструмент BTrace [29]. Для каждого вызова целевого метода записывались графы полей целевого объекта и аргументов до глубины 5, сигнатуры вызовов, выполняемых целевым методом, а также возвращённое значение или класс выброшенного исключения. Сгенерированный тест также выполнялся под управлением BTrace, после чего полученная трасса сопоставлялась с эталонной.

Сценариями будем называть отличающиеся друг от друга вызовы целевого метода в эталонной трассе. Повторяющиеся вызовы учитываются один раз. Сравнение сценариев проводилось на трёх вложенных уровнях:

- структура (S): совпадают способ завершения вызова (возврат значения или исключение определённого класса) и последовательность вызовов, которые целевой метод выполняет в ходе работы. При этом сравниваются только сигнатуры вызываемых методов, без их аргументов. Таким образом, сравнение показывает, что в тесте целевой метод обращается к тем же зависимостям в том же порядке и завершается так же, как в исходном тесте;
- вход и выход (A): дополнительно совпадают аргументы и возвращаемое значение; этот уровень служит основным показателем воспроизведения данных;
- состояние (B): дополнительно совпадает состояние целевого объекта.

Объекты сравниваются по графу полей, без использования методов `equals` и `toString`. Для значений, которые в модульном тесте заменяются «заглушками» (зависимостей, прокси, объектов инфраструктурных типов), сравнивается только то, задано ли значение: в обеих трассах оно должно быть либо равно `null`, либо отлично от `null`. Перед сравнением одинаково для всех режимов нормализуются значения, зависящие от запуска: время, UUID (Universally Unique Identifier, универсальный 128-биный уникальный идентификатор), хеш-коды объектов и пути к временным файлам.

Для каждой пары вычислялась доля сценариев эталона, воспроизведённых сгенерированным тестом на каждом уровне. Тест, который не компилируется или не проходит, не воспроизводит ни одного сценария. Доля усредняется сначала по запускам, а затем по парам.

Предварительный код без рефакторинга проходит лишь в 9 из 50 пар, но в этих парах воспроизводит все сценарии, поэтому доля воспроизведённых сценариев на всех уровнях составляет 18 %. Разности с базовым режимом на уровнях A и B для пар с прямым и косвенным вызовом целевого метода показаны на рис. 3.

Сравнение по уровню S почти не различает тесты базового режима и тесты, построенные по дескриптору: на этом уровне они воспроизводят в среднем 98% и 94% сценариев. Это

объясняется тем, что в 48 из 70 пар все вызовы целевого метода в исходном тесте одинаково завершаются, обращаются к одним и тем же зависимостям и различаются только данными.

Различие режимов проявляется на уровнях A и B (рис. 3). На 63 парах тесты, построенные по дескриптору, воспроизводят с теми же аргументами и результатом в среднем 86% сценариев против 72 % у тестов базового режима, а с учётом состояния целевого объекта – 63% против 43%. На 50 парах тесты, полученные рефакторингом предварительного кода, с учётом состояния воспроизводят 63% сценариев против 37% у базового режима, а на уровне A значимого различия нет (74 % против 69 %).

Табл. 3. RQ1: доля воспроизведённых сценариев по режимам, %.
Table 3. RQ1: share of reproduced scenarios by mode, %.

Кол-во пар	Режим	S, структура	A, вход и выход	B, состояние
50	Базовый режим	97	69	37
	Предварительный код без рефакторинга	18	18	18
	Рефакторинг предварительного кода	85	74	63
	Генерация по дескриптору	93	83	61
63	Базовый режим	98	72	43
	Генерация по дескриптору	94	86	63

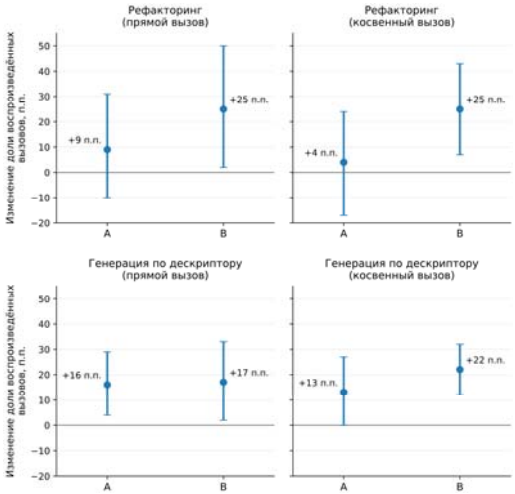


Рис. 3. RQ1: разность доли воспроизведённых сценариев по сравнению с базовым режимом на уровнях A и B (среднее и 95-процентный доверительный интервал, п. п.).
Fig. 3. RQ1: difference in the share of reproduced scenarios compared to the baseline at levels A and B (mean and 95% confidence interval, percentage points).

5.3 RQ2. Обнаружение ошибок

Для оценки способности тестов обнаруживать ошибки использовалось мутационное тестирование с помощью инструмента PIT [30]. В код целевого класса вносились небольшие искусственные изменения (мутанты), имитирующие дефекты, после чего на изменённой

программе выполнялся только сгенерированный тест. Мутант считается обнаруженным, если тест на нём не проходит.

Для каждого теста вычислялась доля обнаруженных мутантов среди мутантов, затронутых тестом, то есть находящихся в исполняемом им коде. Доля усредняется по прошедшим тестам пары, а затем по парам, в которых прошли тесты обоих сравниваемых режимов.

Табл. 4. RQ2: обнаружение мутантов, затронутых тестом, по сравнению с базовым режимом на тех же парах.
Table 4. RQ2: detection of mutants covered by the test compared to the baseline on the same pairs.

Режим	Количество пар	Доля обнаруженных мутантов у тестов базового режима, %	Доля обнаруженных мутантов у тестов режима, %
Генерация по дескриптору	61	81	79
Рефакторинг предварительного кода	48	81	57
Предварительный код без рефакторинга	9	85	50

Тесты, построенные по дескриптору, обнаруживают в среднем 79 % затронутых мутантов, а тесты базового режима на тех же парах – 81 %, и значимого различия между ними нет. Таким образом, использование данных исполнения не снижает способность тестов обнаруживать ошибки. Тесты, полученные рефакторингом предварительного кода, обнаруживают лишь 57 % затронутых мутантов. Они содержат в среднем вдвое меньше проверок, чем тесты базового режима, поскольку сам предварительный код проверяет только зафиксированный результат, а при рефакторинге модель добавляет мало новых проверок. Предварительный код без рефакторинга проходит лишь в 9 парах, поэтому его значение следует рассматривать как ориентировочное.

5.4 RQ3. Вклад построения предварительного кода и его рефакторинга

В RQ3 оценивался вклад двух шагов этапа синтеза: детерминированного построения предварительного кода и его рефакторинга языковой моделью. Детерминированное построение фиксирует создание объектов, подготовку аргументов, конфигурирование «заглушек» и вызов целевого метода до обращения к языковой модели, но при этом может ограничивать модель при построении дополнительных проверок. Для каждого режима рассматривались доля пар, для которых тест компилируется и проходит без «ручных» исправлений, а также затраты на генерацию: медианы времени работы агента и числа вызовов инструментов.

Табл. 5. RQ3: применимость, прохождение тестов и затраты на генерацию.
Table 5. RQ3: applicability, passing tests and generation effort.

Показатель	Базовый режим	Предварительный код без рефакторинга	Рефакторинг предварительного кода	Генерация по дескриптору
Количество пар с входными данными	70	50	50	63
Тест компилируется и проходит, %	100	18	94	96
Медиана времени работы агента, сек	200	–	171	195
Медиана числа вызовов инструментов	15	–	13	16

Рефакторинг языковой моделью необходим для использования предварительного кода: без него тест компилируется лишь в 12 из 50 пар и проходит в 9. Основные причины – несогласованные типы при настройке «заглушек» (21 пара, в основном в проектах Apollo и MyBatis) и обращения к недоступным символам (15 пар). После рефакторинга тест проходит в 94 % пар и воспроизводит с теми же аргументами и результатом в среднем 74 % сценариев вместо 18 %.

На одних и тех же 50 парах тесты, полученные рефакторингом предварительного кода, и тесты, построенные по дескриптору, с учётом состояния целевого объекта воспроизводят почти одинаковую долю сценариев: 63% и 61%. Таким образом, детерминированное построение предварительного кода не дало выигрыша в воспроизведении по сравнению с генерацией по дескриптору. Затраты на генерацию в обоих режимах сопоставимы: медианы времени работы агента составили 171 и 195 секунд.

5.5 RQ4. Качество кода сгенерированных тестов

Для ответа на четвёртый исследовательский вопрос для каждого режима оценивались свойства сгенерированных тестов как сопровождаемого кода. Рассматривались наличие и количество «заглушек». Кроме того, учитывались признаки, затрудняющие чтение кода: обращения к рефлексии и синтетические имена переменных вида `v0`. Эти показатели вычисляются механически и не позволяют судить о готовности тестов к включению в промышленный репозиторий, поэтому они используются только для сравнения режимов между собой.

Табл. 6. RQ4: показатели качества кода сгенерированных тестовых классов (в скобках – значения базового режима на тех же парах).
Table 6. RQ4: code quality indicators of generated test classes (baseline values on the same pairs in parentheses).

Показатель	Базовый режим	Предварительный код без рефакторинга	Рефакторинг предварительного кода	Генерация по дескриптору
Тестовые классы с «заглушками», %	53	92 (64)	80 (64)	69 (57)
«Заглушки» в тестовом классе, медиана, шт.	1	8 (2)	4 (2)	3 (1)
Тестовые классы с обращениями к рефлексии, %	13	90 (16)	38 (16)	28 (14)
Тестовые классы с синтетическими именами переменных, %	12	98 (12)	21 (12)	13 (13)

Предварительный код без рефакторинга имеет явные признаки машинно построенного кода: рефлексия используется в 90 % тестовых классов, а синтетические имена переменных – в 98%. После рефакторинга языковой моделью эти доли снижаются до 38 и 21 %, но остаются выше, чем у тестов базового режима на тех же парах (16 и 12 %). Тесты, построенные по дескриптору, чаще содержат «заглушки», чем тесты базового режима на тех же парах (69% и 57% тестовых классов), поскольку воспроизводят зафиксированные взаимодействия с зависимостями.

6. Ограничения

Ограничения метода. Тестовый оракул, построенный по данным исполнения, является регрессионным: он фиксирует наблюдавшееся поведение, в том числе возможно ошибочное. Реализация строит дескриптор исполнения и предварительный код не для всех пар набора (см. разд. 5.1), а допущения модели формализации перечислены в разд. 3.6. Сгенерированные

тесты привязаны к конкретным вызовам зависимостей и могут обращаться к закрытым полям через рефлексию, поэтому изменение реализации целевого метода может приводить к их падению. Устойчивость тестов к изменениям кода в работе не оценивалась. Кроме того, реализация метода использует интерфейс отладчика JDI и поэтому применима только к программам, исполняемым на JVM.

Ограничения эксперимента. Трассировщик не записывает аргументы и результаты вызовов, выполняемых целевым методом, поэтому для этих вызовов сравниваются только сигнатуры. Графы объектов сравниваются до глубины 5, а для коллекций записываются первые 200 элементов, поэтому усечённые множества и словари сравниваются только по типу и размеру. Список недетерминированных значений составлен «вручную», и его полнота проверялась лишь повторным снятием эталона.

От агента скрывались только тесты класса целевого метода, поэтому в проекте могут оставаться другие тесты, вызывающие этот метод, и они доступны агенту во всех режимах. Рабочая копия проекта не изолирована средствами операционной системы, а сборка и запуск тестов выполняют код проекта. В связи с этим каждый запуск автоматически проверялся на обращения агента и сгенерированного теста за пределы рабочей копии. Такие обращения обнаружены в 5 запусках: в 4 случаях сгенерированный тест читал файлы данных исполнения и при проверке на чистой копии проекта не проходил, а одно срабатывание оказалось ложным.

В эксперименте использовались одна языковая модель и один агент, а для каждой пары осуществлялось только два запуска, поэтому для других моделей и агентов результаты могут отличаться. Набор включает только открытые Java-проекты и не является случайной выборкой, поэтому результаты не обобщаются на произвольные JVM-системы. Наконец, показатели качества кода вычисляются механически, а «ручная» оценка читаемости тестов не проводилась.

7. Обзор смежных работ

7.1 Автоматическая генерация модульных тестов

До появления практически применимых больших языковых моделей инструменты автоматической генерации тестов основывались преимущественно на формальных методах. К данным методам относятся символьное исполнение [32], эволюционное тестирование [5] и тестирование на основе формальных спецификаций [7, 31].

Несмотря на эффективность формальных методов в генерации тестов с высоким покрытием, их практическая применимость в цикле разработки программных продуктов ограничена. Символьное исполнение кода с внешними зависимостями, такими как сторонние фреймворки и интерфейсы баз данных, в общем случае затруднено из-за сложности ограничений пути. Фаззинг и эволюционное тестирование по своей сути порождают тесты со случайными данными, которые решают задачу поиска ошибок, но не документируют поведение компонента. Также общей проблемой формальных методов является сложность генерации объектно-ориентированного кода. Модели, которыми оперируют формальные методы, описывают внутреннюю структуру объектов, но не содержат информации о том, как создать соответствующий объект в коде. Несмотря на то, что вышеперечисленные проблемы эвристически решаются в конкретных инструментах [33, 34], формальные методы продолжают оставаться средством дополнительной верификации, отдельным от основного цикла тестирования.

С возникновением больших языковых моделей стала возможна автоматическая генерация кода, и, в частности, тестов, основанная не на формальном описании программы, а на анализе её текстового представления. Языковые модели строят ответ как последовательность токенов, вероятную в заданном контексте, что позволяет использовать их как для синтеза программного кода, так и для рефакторинга.

7.2 «Вырезка» модульных тестов

Получение модульных тестов из исполнения системных тестов восходит к работе Elbaum и соавторов [35], в которой введены дифференциальные модульные тесты. При выполнении системного теста инструментированная программа сохраняет состояние перед вызовом целевого метода, после чего вызов воспроизводится изолированно из этого состояния. Оракулом служит совпадение с записанным состоянием после вызова, поэтому такой тест обнаруживает различия между версиями программы. Kampmann и Zeller [36] в инструменте BASILISK перенесли подход на программы на языке C: вырезанные тесты параметризуются и применяются для поиска ошибок, а не для защиты от регрессий. Тот же принцип использует инструмент TestGen, применяемый в промышленности [37]. Плагины компилятора добавляют в целевые методы запись объекта, аргументов и результата при выполнении E2E-тестов, а тест проверяет, что при тех же данных метод возвращает записанное значение.

Запись на границе метода применяется и к данным, собираемым при работе приложения в эксплуатации. Инструмент Pankti [38] сохраняет целевой объект, аргументы и результат слабо протестированных методов и строит по ним дифференциальные тесты. RICK [39] дополнительно записывает вызовы зависимостей, найденные статическим анализом, и заменяет зависимости «заглушками» с записанными ответами. Помимо результата, его тесты проверяют аргументы, число и порядок вызовов к зависимостям. В этих инструментах, как и в TestGen, объекты загружаются в тесте из сохранённых файлов. Поскольку из такого теста трудно понять, как получено исходное состояние, в ряде работ повышение понятности является отдельной целью. MicroTestCarver [40] выделяет модульные тесты из «ручных» и скриптовых интеграционных тестов, восстанавливает объекты вызовами конструкторов и заменяет «заглушками» классы за пределами анализируемого пакета. ProDJ [41] представляет наблюдаемые объекты в виде исходного кода на Java, а Modest [42] строит тесты для Java и Pharo по трассам исполнения через независимое от языка представление теста.

Характеристики рассмотренных инструментов приведены в табл. 7.

Табл. 7. Инструменты построения модульных тестов по данным исполнения.
Table 7. Tools for unit test generation from execution data.

Инструмент	Источник исполнений	Восстановление состояния в тесте	«Заглушки»	Языковая модель
Elbaum и др. [35]	Системные тесты	Восстановление из XML	Нет	Нет
BASILISK [36]	Системные тесты (язык C)	Код, воссоздающий данные в куче	Нет	Нет
TestGen [37]	E2E-тесты, «ручной» запуск приложения	Восстановление из JSON	Нет	Нет
Pankti [38]	Работа приложения в эксплуатации	Восстановление из XML	Нет	Нет
RICK [39]	Работа приложения в эксплуатации	Восстановление из XML	Да, по записанным вызовам	Нет
MicroTestCarver [40]	«Ручные» и скриптовые E2E-тесты	Код: конструкторы, при необходимости рефлексия	Да, для классов вне анализируемого пакета	Нет
ProDJ [41]	Исполнение приложения	Код: конструкторы и методы установки полей	Не рассматриваются	Нет
Modest [42]	Трассы исполне-ния (Java, Pharo)	Код, построенный по модели теста	Нет	Нет
Предложенный метод	Существующие тесты проекта	Код: конструкторы и методы установки полей, при необходимости рефлексия или восстановление	Да, с наблюдаемым поведением	Да

Во всех этих работах набор наблюдаемых методов задаётся до запуска, а код теста строится детерминированно, без языковой модели. В предложенном методе набор перехватываемых методов расширяется по ходу исполнения, объекты по возможности создаются кодом теста, а зависимости заменяются «заглушками» с наблюдаемым поведением. Данные исполнения передаются языковой модели либо в виде дескриптора, либо в виде построенного по нему предварительного кода теста.

7.3 Улучшение эффективности формальных методов за счёт реальных данных

Методы сбора данных о реальном выполнении программы могут использоваться для повышения эффективности формальных методов анализа. В частности, в ряде исследований предлагается применять реальные данные в качестве начальных значений для фаззинга и мутационного тестирования [43-44]. Предполагается, что мутации значений, полученных из реальных запусков, а не случайно сгенерированных, с большей вероятностью приводят к обнаружению ошибочного поведения. Аналогичным образом могут быть реализованы более эффективные стратегии поиска при динамическом символьном исполнении [45-46].

В отличие от указанных подходов, предложенный в данной работе метод ориентирован на непосредственную генерацию тестов с использованием большой языковой модели. Тем не менее, строящиеся дескрипторы являются универсальными и могут использоваться также для уточнения формальных методов.

8. Заключение

В данной работе предложен метод автоматической генерации модульных тестов по состоянию исполнения программы. В ходе выполнения существующего высокоуровневого теста метод фиксирует состояние целевого объекта, аргументы вызова, результат выполнения и взаимодействия с зависимостями. На основе этих данных строится дескриптор исполнения, по которому затем синтезируется модульный тест, воспроизводящий наблюдавшийся сценарий. Синтез выполняется в два этапа: сначала по дескриптору детерминированно строится предварительный код теста, затем большая языковая модель выполняет его рефакторинг с сохранением зафиксированных значений и проверок.

Экспериментальная оценка проведена на 70 парах «тест/целевой метод» из семи открытых Java-проектов. Сравнивались тесты, построенные ИИ-агентом без данных исполнения, по дескриптору исполнения и путём рефакторинга предварительного кода, а также сам предварительный код, взятый в качестве итогового теста без рефакторинга.

Эксперимент показал, что с данными исполнения агент точнее воспроизводит значения, с которыми работает целевой метод. Тесты, построенные агентом по дескриптору исполнения, в среднем воспроизводят с теми же аргументами и результатом 86% вызовов целевого метода из исходного теста, а при дополнительном учёте значений полей целевого объекта – 63%. У тестов, сгенерированных агентом без доступа к данным исполнения, эти показатели составили 72 и 43 %. Способность тестов обнаруживать ошибки при этом сохраняется: тесты, построенные по дескриптору, обнаруживают 79% искусственно внесённых ошибок в исполняемом ими коде, а тесты без данных исполнения – 81%.

Дальнейшее развитие метода связано с поддержкой более широкого набора типов данных, построением более содержательных проверок в тестах и оценкой того, насколько сгенерированные тесты устойчивы к изменениям кода. Также интерес представляет интеграция предложенного подхода с модельно-ориентированными средствами разработки [47-48].

Список литературы / References

[1]. Cohn M. Succeeding with Agile: Software Development Using Scrum. Upper Saddle River, NJ, Addison-Wesley Professional, 2009, 475 p.

[2]. Hamill W. Automated Testing and the Evils of Ice Cream. 2013. Available at: https://willhamill.com/2013/08/12/automated-testing-and-the-evils-of-ice-cream, accessed 12.10.2026.

[3]. Ivankovic M., Petrovic, G. Code Coverage at Google. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 955-963.

[4]. Baldoni R., Coppa E., D'Elia D. C. et al. A Survey of Symbolic Execution Techniques. arXiv preprint, 2018. DOI: 10.48550/arXiv.1610.00502.

[5]. Fraser, G. Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11), 2011, pp. 416-419.

[6]. Паршин М.А., Степанов Д.С., Кознов Д.В. Test generation from execution state: код и данные эксперимента. Available at: https://github.com/mxprshn/test-generation-from-execution-state, accessed 12.10.2026.

[7]. Сортов А.А., Хорошилов А.В. Функциональное тестирование Web-приложений на основе технологии UniTesK. Труды Института системного программирования РАН, 2004, том 8, вып. 1, стр. 77-98.

[8]. Fowler M. Integration Test. martinfowler.com, 2018. Available at: https://martinfowler.com/bliki/IntegrationTest.html, accessed 12.10.2026.

[9]. Jia Y., Harman M. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering, vol. 37, no. 5, 2011, pp. 649-678.

[10]. Barr E. T., Harman M., McMinn P. et al. The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering, 2015, vol. 41, no. 5, pp. 507-525.

[11]. Weyuker E. J. On Testing Non-Testable Programs. The Computer Journal, 1982, vol. 25, no. 4, pp. 465-470.

[12]. Daka E., Fraser G. A Survey on Unit Testing Practices and Problems. 2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE), 2014, pp. 201-211.

[13]. Kuutila M., Mäntylä M., Farooq U., Claes M. Time pressure in software engineering: A systematic review. Information and Software Technology, 2020, vol. 121, art. 106257.

[14]. Lee J., Kang S., Lee D. Survey on software testing practices. IET Software, 2012, vol. 6, pp. 275-282.

[15]. Oracle. Java Debug Interface (JDI) Overview. 2014. Available at: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/, accessed 12.10.2026.

[16]. JetBrains. IntelliJ Platform XDebugger API: XDebugProcess (source). Available at: https://github.com/JetBrains/intellij-community/blob/master/platform/xdebugger-api/src/com/intellij/xdebugger/XDebugProcess.java, accessed 12.10.2026.

[17]. Oracle. JVM Tool Interface (JVMTI). Available at: https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html, accessed 12.10.2026.

[18]. Spring Framework Documentation. 2026. Available at: https://docs.spring.io/spring-framework/reference/index.html, accessed 12.10.2026.

[19]. Oracle. Package java.lang.reflect (Java SE 21, JDK 21). 2023. Available at: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/reflect/package-summary.html, accessed 12.10.2026.

[20]. apolloconfig/apollo. Apollo is a reliable configuration management system. Available at: https://github.com/apolloconfig/apollo, accessed 12.10.2026.

[21]. Aiven-Open/klaw. Klaw is a self-service Apache Kafka Topic Management/Governance tool. Available at: https://github.com/Aiven-Open/klaw, accessed 12.10.2026.

[22]. mybatis/mybatis-3. MyBatis SQL mapper framework for Java. Available at: https://github.com/mybatis/mybatis-3, accessed 12.10.2026.

[23]. jhy/jsoup. jsoup: the Java HTML parser. Available at: https://github.com/jhy/jsoup, accessed 12.10.2026.

[24]. checkstyle/checkstyle. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. Available at: https://github.com/checkstyle/checkstyle, accessed 12.10.2026.

[25]. eclipse-jgit/jgit. JGit, a pure Java implementation of the Git version control system. Available at: https://github.com/eclipse-jgit/jgit, accessed 12.10.2026.

[26]. liquibase/liquibase. Liquibase: database schema change management. Available at: https://github.com/liquibase/liquibase, accessed 12.10.2026.
[27]. Pi coding agent. Available at: https://www.npmjs.com/package/@earendil-works/pi-coding-agent, accessed 12.10.2026.
[28]. OpenAI. GPT-6.1 Sol Model. Available at: https://developers.openai.com/api/docs/models/gpt-6.1-sol, accessed 12.10.2026.
[29]. BTrace. A safe, dynamic tracing tool for the Java platform. Available at: https://github.com/btraceio/btrace, accessed 12.10.2026.
[30]. PIT Mutation Testing. Available at: https://pitest.org, accessed 12.10.2026.
[31]. Kuliamin V.V., Petrenko A.K., Kossatchev A.S., Burdonov I.B. The UniTesK Approach to Designing Test Suites. Programming and Computer Software, 2003, vol. 29, no. 6, pp. 310-322.
[32]. Vishnyakov A., Fedotov A., Kuts D. et al. Sydr: Cutting Edge Dynamic Symbolic Execution. 2020 Ivannikov ISPRAS Open Conference (ISPRAS), 2020, pp. 46-54.
[33]. Thummalapenta S., Xie T., Tillmann N. et al. Synthesizing Method Sequences for High-Coverage Testing. Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2011, pp. 189-206.
[34]. Abdullin A., Itsykson V. Reanimator: from Test Data to Code and Back. Programmnaya Ingeneria, 2022, vol. 13, no. 12, pp. 483-496.
[35]. Elbaum S., Chin H.N., Dwyer M.B., Jorde M. Carving and Replaying Differential Unit Test Cases from System Test Cases. IEEE Transactions on Software Engineering, 2009, vol. 35, no. 1, pp. 29-45.
[36]. Kampmann A., Zeller A. Carving Parameterized Unit Tests. arXiv preprint, 2018. DOI: 10.48550/arXiv.1812.07932.
[37]. Alshahwan N., Harman M., Marginean A. et al. Observation-Based Unit Test Generation at Meta[1]. Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, 2024, pp. 173-184.
[38]. Tiwari D., Zhang L., Monperrus M., Baudry B. Production Monitoring to Improve Test Suites. IEEE Transactions on Reliability, 2022, vol. 71, no. 3, pp. 1381-1397.
[39]. Tiwari D., Monperrus M., Baudry B. Mimicking Production Behavior with Generated Mocks. Transactions on Software Engineering, 2024, vol. 50, no. 11, pp. 2921-2946.
[40]. Deljouyi A., Zaidman A. Generating Understandable Unit Tests through End-to-End Test Scenario Carving. 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), 2023, pp. 107-118.
[41]. Wachter J., Tiwari D., Monperrus M., Baudry B. Serializing Java Objects in Plain Code. Journal of Systems and Software, 2026, vol. 234, art. 112721.
[42]. Darbord G., Anquetil N., Verhaeghe B., Etien, A. A Multi-Language Tool for Generating Unit Tests from Execution Traces. 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025, pp. 241-251.
[43]. Jaygarl H., Kim S., Xie T., Chang C.K. OCAT: object capture-based automated testing. Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10), 2010, pp. 159-170.
[44]. Galland O., Böhme M. In-vivo Fuzzing by Amplifying Actual Executions. Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 2025, pp. 1566-1578.
[45]. Thummalapenta S., de Halleux J., Tillmann N., Wadsworth S. DyGen: Automatic Generation of High-Coverage Tests via Mining Gigabytes of Dynamic Traces. Tests and Proofs, Berlin, Heidelberg, Springer, 2010, pp. 77-93.
[46]. Wang Q., Orso A. Improving Testing by Mimicking User Behavior. Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 488-498.
[47]. Терехов А.Н., Кознов Д.В., Романовский К.Ю., Долгов П., Иванов А.Н. Объектно-ориентированная методология разработки информационных систем и программного обеспечения систем реального времени. Объектно-ориентированное визуальное моделирование. Санкт-Петербург, 1999, с. 4-20.
[48]. Иванов А., Кознов Д., Мурашева Т. Поведенческая модель RTST++. Записки семинара Кафедры системного программирования «Case-средства RTST++», 1998, № 1, с. 37-52.

[1] Meta Platforms Inc. признана экстремистской организацией, её деятельность запрещена на территории Российской Федерации.

Информация об авторах / Information about authors

Максим Алексеевич ПАРШИН – аспирант математико-механического факультета Санкт-Петербургского государственного университета. Сфера научных интересов: анализ программ, системы искусственного интеллекта.

Maksim Alekseevich PARSHIN – postgraduate student of the Faculty of Mathematics and Mechanics, St. Petersburg State University (SPbSU). Research interests: program analysis, artificial intelligence systems.

Даниил Сергеевич СТЕПАНОВ – кандидат технических наук, доцент института прикладных компьютерных наук НИУ ИТМО. Сфера научных интересов: программная инженерия, статический и динамический анализ программ, прикладное использование больших языковых моделей.

Daniil Sergeevich STEPANOV – Candidate of Technical Sciences, Associate Professor at the Institute of Applied Computer Science, ITMO University. His research interests include software engineering, static and dynamic program analysis, and applied uses of large language models.