Научная статья
УДК 004.415.532.2
URL: https://trudymai.ru/published.php?ID=188284
EDN: https://www.elibrary.ru/FAOAHQ

ТРЁХЭТАПНАЯ МЕТОДОЛОГИЯ ГЕНЕРАЦИИ ВЕРИФИЦИРОВАННЫХ UI-ТЕСТОВ НА ОСНОВЕ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ

А.М. Титеев ✉

Московский авиационный институт (национальный исследовательский университет),

г. Москва, Россия

✉ loksader@yandex.ru

Цитирование: Титеев А.М. Трёхэтапная методология генерации верифицированных UI-тестов на основе больших языковых моделей // Труды МАИ. 2026. № 148. URL: https://trudymai.ru/published.php?ID=188284

Аннотация. Работа посвящена проблеме автоматизации тестирования пользовательских интерфейсов современных веб-приложений, где сохраняется разрыв между требованиями на естественном языке и автоматизированными проверками, а «ложноположительные» тесты создают иллюзию качества. Особенно остро проблема стоит в авиакосмической отрасли при тестировании интерфейсов для контроля телеметрии и предполётного планирования, где ошибки визуализации критичны, а специалисты по автоматизации дефицитны. Предложена трёхэтапная методология автоматической генерации верифицированных UI-тестов на основе спецификаций на естественном языке с применением большой языковой модели Llama 3.3 70B и фреймворка Playwright. На первом этапе формируется структурированная UI-спецификация, доступная нетехническим экспертам. На втором — генерируются классы Page Object, инкапсулирующие логику взаимодействия. На третьем — создаются тестовые сценарии с многоуровневой верификацией, включающей проверку синтаксиса, компилируемости, надёжности выполнения и прироста тестового покрытия. Ключевым элементом является мутационное тестирование посредством перехвата DOM-событий, позволяющее неинвазивно вносить дефекты в

интерфейс и проверять способность теста их обнаружить. В результате разработаны критерии многоуровневой фильтрации, отсеивающие нестабильные и ложноположительные тесты, и метод мутационного тестирования, не требующий модификации исходного кода приложения.

Ключевые слова: автоматизация тестирования; UI-тесты; большие языковые модели; генерация тестов; мутационное тестирование; Page Object Model; Playwright.

METHODOLOGY OF THREE STEP GENERATION OF VERIFIED UI-TESTS BASED ON LLM

A.M. Titeev

Moscow Aviation Institute (National Research University), Moscow, Russia

loksader@yandex.ru

Citation: Titeev A.M. Methodology of three step generation of verified UI-tests based on LLM // Trudy MAI. 2026. No. 148. (In Russ.). URL: https://trudymai.ru/published.php?ID=188284

Abstract. This work addresses the problem of automating the testing of user interfaces in modern web applications, where a gap persists between natural language requirements and automated checks, and "false-positive" tests create an illusion of quality. The issue is particularly acute in the aerospace industry when testing interfaces for telemetry monitoring and pre-flight planning, where visualization errors are critical and automation specialists are scarce. A three-stage methodology is proposed for the automatic generation of verified UI tests based on natural language specifications using the Llama 3.3 70B large language model and the Playwright framework. In the first stage, a structured UI specification accessible to non-technical experts is formed. In the second, Page Object classes encapsulating interaction logic are generated. In the third, test scenarios are created with multi-level verification, including checks for syntax, compilability, execution reliability, and coverage growth. A key element is mutation testing through DOM event interception, allowing defects to be non-invasively introduced into the interface and verifying the test's ability to detect them. As a result, multi-level filtering criteria have been developed to weed out unstable and false-

positive tests, along with a mutation testing method that does not require modification of the application's source code.

Keywords: test automation, UI tests; large language models; test generation; mutation testing; Page Object Model; Playwright.

Введение

Автоматизация тестирования пользовательских интерфейсов веб-приложений представляет собой задачу, характеризующуюся высокой трудоёмкостью на всех этапах жизненного цикла тестов. За последние два десятилетия разработан ряд инструментов и методологий, однако ключевые проблемы остаются нерешёнными: стоимость создания и сопровождения тестовых сценариев остаётся значительной, устойчивость тестов к изменениям структуры интерфейса ограничена, а потребность в специалистах, одновременно владеющих предметной областью и навыками программирования, сохраняется. Указанные ограничения приобретают особую значимость в системах с повышенными требованиями к надёжности, в частности в авиационных информационных системах. В данном классе приложений дефекты уровня пользовательского интерфейса — такие как некорректное отображение телеметрических данных или отказы в обработке управляющих команд — способны повлиять на действия оператора и, как следствие, на безопасность эксплуатации.

Существующие подходы к автоматизации тестирования пользовательских интерфейсов можно разделить на традиционные методы (ручное написание тестов с использованием паттерна Page Object, запись и воспроизведение действий, BDD-фреймворки) и подходы на основе больших языковых моделей. Традиционные методы либо требуют значительных затрат на программирование и поддержку тестов, либо создают разрыв между спецификацией и исполняемым кодом. Современные LLM-подходы успешно применяются для генерации модульных тестов, однако слабо иследованы в области решения задачи сквозного UI-тестирования и, как правило, не включают механизмов проверки способности тестов обнаруживать функциональные дефекты интерфейса. Цель

данной работы — предложить методологию сквозной генерации UI-тестов из спецификации на естественном языке, включающую этапы создания Page Object, генерации тестовых сценариев и их обязательной верификации с помощью мутационного тестирования, ориентированную на применение в высоконадёжных системах.

Традиционные подходы к автоматизации тестирования

Ручное написание автотестов остаётся наиболее распространённым подходом в индустрии. Тестировщик анализирует требования, изучает структуру приложения и создаёт тестовый код с использованием фреймворков автоматизации, таких как Selenium, Playwright или Cypress. Архитектурный подход Page Object Model [1] позволяет структурировать тестовый код, отделяя описание элементов интерфейса от логики тестовых сценариев, что помогает уменьшить время, затрачиваемое на поддержку тестов.

Инструменты записи действий пользователя представляют альтернативный подход, снижающий порог входа в автоматизацию. Playwright Codegen и Katalon Recorder позволяют записывать действия пользователя в браузере и воспроизводить их в виде автоматизированных тестов. Привлекательность данного подхода заключается в возможности создания тестов без написания кода. Тем не менее записанные тесты всё ещё требуют поддержки, т.к. при изменении интерфейса ломаются заданные локаторы (идентификаторы элемента на веб-странице).

Подход BDD с использованием фреймворков Cucumber, SpecFlow и Behave направлен на сокращение разрыва между требованиями и тестами. Спецификации описываются на языке Gherkin в формате Given-When-Then, понятном нетехническим специалистам. Однако BDD требует создания связующего кода, который транслирует шаги спецификации в исполняемые действия. Это порождает проблему двойной работы: необходимо поддерживать как спецификации, так и их программную реализацию. На практике связующий код часто становится узким местом, требующим навыков программирования, что не решает проблему высокого порога входа. Кроме того, спецификации со

временем расходятся с реальной реализацией тестов, что снижает ценность BDD как инструмента документирования.

Применение больших языковых моделей для генерации модульных тестов

Развитие больших языковых моделей стимулировало исследования в области автоматической генерации тестов. Обзорное исследование [2] по применению LLM(Large Language Models - больших языковых моделей) в тестировании идентифицирует генерацию тестов как одно из наиболее активных направлений исследований. Значительная часть работ сфокусирована на генерации модульных тестов для программного кода.

Авторы исследования [3] предложили фреймворк AGONETEST для автоматизированной генерации и оценки юнит-тестов на Java, создаваемых большими языковыми моделями. В состав фреймворка входит размеченный набор данных CLASSES2TEST (147473 тестовых классов), который сопоставляет классы с их тестовыми классами и может включать примеры для few-shot промптинга. AGONETEST позволяет унифицированно сравнивать различные LLM и стратегии промптов, однако текущие эксперименты ограничены двумя типами промптов, ручным отбором примеров и единственным запуском, что не позволяет оценить воспроизводимость результатов

В статье [4] предложили метод автоматической генерации тестовых сценариев на основе глубокого обучения, дополненный знаниями об assert-утверждениях и механизмом верификации. По результатам оценки на наборе Defects4J, A3Test показал 40,05% корректных тестов. Авторы также провели абляционное исследование для оценки вклада отдельных компонентов подхода в итоговую производительность.

В исследовании [5] провели систематическую оценку пяти open-source LLM и GPT-4 в генерации модульных тестов на 778 методах из 17 проектов Defects4J 2.0. Исследование выявило, что качество генерируемых тестов существенно зависит от дизайна промпта (запроса к языковой модели): стиль описания должен соответствовать обучающим данным модели, а выбор code features влияет на компилируемость и тестовое покрытие. Авторы идентифицировали

проблему галлюцинаций (генерация моделью правдоподобного, но фактически некорректного содержимого) как основную причину невалидных тестов: 30,68% ошибок компиляции вызваны обращениями к несуществующим идентификаторам, 17,25% — несоответствием параметров методов, 10,38% — попытками инстанцирования абстрактных классов.

В статье [6] представили TestPilot — систему адаптивной генерации тестов для JavaScript на основе модели Codex. TestPilot использует промпты, включающие сигнатуру функции, её реализацию, документацию и примеры использования, извлечённые из документации проекта. Ключевым элементом подхода является адаптивный компонент: если сгенерированный тест падает, система повторно делает запрос к модели с включением падающего теста и сообщения об ошибке. Эксперименты на 25 npm-пакетах показали медианное тестовое покрытие 68,2%, при этом 58,5% сгенерированных тестов содержат нетривиальные утверждения (assertions), зависящие от тестируемой функциональности.

Важным результатом исследования TestPilot является демонстрация значимости всех компонентов промпта. Эксперименты с исключением отдельных компонентов показали, что удаление любого из элементов (сигнатуры, документации, примеров или исходного кода) приводит либо к увеличению доли падающих тестов, либо к снижению тестового покрытия. Это подчёркивает необходимость предоставления модели максимально полного контекста для эффективной генерации.

В исследовании [7] разработали ChatUniTest — фреймворк для генерации модульных тестов на Java с акцентом на верификацию и исправление сгенерированных тестов. Система реализует трёхэтапную архитектуру: предобработка (парсинг проекта и анализ зависимостей), генерация и постобработка (валидация и исправление). Ключевым элементом является механизм адаптивного фокального контекста, который динамически формирует промпт с максимумом релевантной информации при соблюдении ограничения на количество токенов (минимальная единица текста, обрабатываемая языковой моделью).

Модуль верификации ChatUniTest включает последовательную проверку синтаксической корректности, успешной компиляции и выполнения без ошибок. Эксперименты на четырёх Java-проектах показали следующие результаты: 75% тест-классов имели хотя бы один корректно компилирующийся тест, 57% — хотя бы один надёжно проходящий тест, 25% — хотя бы один тест, увеличивающий тестовое покрытие. Общее тестовое покрытие составило 59,6%, что сопоставимо с инструментом EvoSuite (38,2%) и превосходит TestSpark (42,1%) на тестовом наборе.

ChatUniTest реализует двухуровневую стратегию исправления: rule-based для типичных ошибок (отсутствующие импорты, усечённые тесты) и LLM-based для сложных случаев. Система ограничивает количество попыток исправления 3-5 итерациями, после чего тест считается требующим ручного вмешательства. Важным аспектом является проверка качества утверждений: система использует обратный слайсинг программы для определения, зависит ли утверждение от тестируемого компонента.

В исследовании [8] представили TestGen-LLM — первую систему промышленного масштаба для автоматического улучшения существующих модульных тестов. В отличие от TestPilot и ChatUniTest, генерирующих тесты с нуля, TestGen-LLM фокусируется на расширении существующих человеко-написанных тестовых классов дополнительными тестовыми случаями, закрывающие пропущенные граничные условия.

Система использует ensemble-подход, комбинируя результаты двух внутренних LLM с различными стратегиями промптирования (extend_coverage, corner_cases, extend_test, statement_to_complete) и варьируемыми параметрами. Эксперименты на Kotlin-компоненах показали, что разные промпты вносят уникальный вклад: extend_coverage добавил 5 уникальных тестов, corner_cases — 1, extend_test — 2.

Ключевым элементом TestGen-LLM является система последовательной фильтрации, гарантирующая качество сгенерированных тестов. Кандидаты последовательно проверяются на:

1. синтаксическую корректность,

2. успешную компиляцию,
3. надёжное прохождение без flakiness (5 запусков),
4. увеличение тестового покрытия по сравнению со всеми существующими тестами.

Статистика фильтрации демонстрирует строгость процесса: из 32,531 попытки генерации только 1,321 (4%) успешно прошли все фильтры.

Промышленное развёртывание TestGen-LLM в период октябрь-декабрь 2023 года продемонстрировало практическую применимость подхода. Система успешно улучшила приблизительно 10% всех тест-классов, к которым была применена, причём 73% сгенерированных улучшений были приняты разработчиками и внедрены в production. В первом test-a-thon TestGen-LLM заняла 6-е место из 36 по количеству созданных тестовых случаев, продемонстрировав конкурентоспособность с человеческими тестировщиками.

Важным аспектом исследования является анализ причин падения тестов. Медианное распределение ошибок: корректность (24,3%, преимущественно type errors), timeout (23,1%, в основном из-за пропущенных вызовов done()), assertion errors (22,9%), file system errors (специфичны для доменов работы с файлами). Адаптивное исправление с использованием сообщений об ошибках позволило исправить 16,3% падающих тестов в среднем по всем категориям ошибок.

Отдельное направление исследований составляют самовосстанавливающиеся (self-healing) фреймворки автоматизации тестирования пользовательского интерфейса. В статье [9] предлагают комплексную архитектуру, использующую ML-модели для динамической идентификации локаторов, интеллектуальных механизмов ожидания и обнаружения аномалий, что позволяет снизить время поддержки тестов на 70%. Работа [10] развивают данный подход, используя мультимодальный подход с компьютерным зрением для того чтобы динамически чинить тестовые сценарии.

Особую сложность представляет генерация тестов для методов с высокой цикломатической сложностью: существующие LLM-генераторы демонстрируют падение тестового покрытия с 93% до 51% для таких методов. В статье [11] применяют декомпозицию программ (method slicing), разбивая сложные методы

на логические срезы и генерируя тесты для каждого среза отдельно с использованием Chain-of-Thought промптинга. Данный подход упрощает область анализа LLM и позволяет достичь прироста тестового покрытия с 32% до 55%.

Предлагаемая методология

Предлагаемая методология основана на последовательном преобразовании спецификации на естественном языке в верифицированные исполняемые тесты. Процесс разделён на три основных этапа генерации с последующей верификацией. Подобная многоэтапная архитектура находит подтверждение в современных исследованиях: ChatUniTest [7] выделяет этапы предобработки, генерации и постобработки, а TestGen-LLM [8] использует последовательную фильтрацию кандидатов для обеспечения гарантий качества.

На первом этапе создаётся UI-спецификация — структурированное описание элементов интерфейса и ожидаемого поведения приложения на естественном языке. Данная спецификация может быть создана или отредактирована специалистами без навыков программирования: техническими писателями, бизнес-аналитиками или владельцами продукта. Важность контекстной информации для эффективной генерации подтверждается исследованием TestPilot [6], где показано, что исключение любого компонента промпта (сигнатуры функции, документации, примеров или исходного кода) приводит к снижению качества генерируемых тестов. ChatUniTest [7] развивает эту идею, предлагая механизм адаптивного фокального контекста, который динамически формирует контекст с максимумом полезной информации при соблюдении ограничения на количество токенов. В предлагаемом подходе UI-спецификация выполняет роль такого контекста, обеспечивая модель информацией об иерархии элементов, ожидаемом поведении и типичных сценариях взаимодействия.

На втором этапе языковая модель анализирует UI-спецификацию и генерирует классы Page Object, инкапсулирующие взаимодействие с элементами интерфейса. Сгенерированные классы наследуются от библиотеки базовых компонентов, что обеспечивает единообразие структуры и интеграцию с

системой отчётности. Использование паттерна Page Object Model [1] согласуется с промышленными практиками автоматизации тестирования и упрощает поддерживаемость сгенерированного кода.

На третьем этапе на основе Page Object и описанных в спецификации сценариев генерируются тестовые сценарии. Система создаёт как позитивные тесты, проверяющие корректную работу функциональности, так и негативные тесты, проверяющие обработку ошибочных ситуаций. Подход множественной генерации кандидатов, применённый в TestGen-LLM [8], демонстрирует эффективность ensemble-стратегии: использование различных промптов и параметров модели позволяет генерировать уникальные тестовые сценарии, увеличивая общее тестовое покрытие функциональности.

Далее происходит верификация тестов, которая обеспечивает гарантию работоспособности сгенерированных тестов. Фреймворк автоматически запускает тесты, исправляет обнаруженные ошибки и проверяет способность тестов обнаруживать дефекты посредством мутационного тестирования - внесение небольших изменений (мутаций) в исходный код программы и проверку способности тестов обнаружить эти изменения. Мутация, не обнаруженная тестами, указывает на недостаточность тестового покрытия. Необходимость систематической верификации сгенерированных тестов подчёркивается в исследованиях применения LLM для генерации тестов [2] .

Ключевым преимуществом разделения на этапы является возможность локализации ошибок и вмешательства на любом промежуточном шаге. Если сгенерированный Page Object содержит некорректные локаторы, ошибка обнаруживается и исправляется до генерации тестов. Если UI-спецификация неполна, технический писатель может дополнить её без необходимости понимания тестового кода.

Ключевым отличием предлагаемой методологии от существующих подходов является систематическая проверка способности сгенерированных тестов обнаруживать дефекты. Тест, который успешно проходит, но не способен выявить введённый дефект не обеспечивает реальной защиты от регрессий. Для авиационных информационных систем цена такого ложного срабатывания

особенно высока: некорректное отображение телеметрических данных или невыполненная команда из-за скрытого дефекта интерфейса могут привести к ошибочным решениям оператора. Поэтому проверка способности теста обнаруживать именно нефункциональные, но критические дефекты уровня пользовательского интерфейса становится обязательной.

Этап генерации Page Object и тестов

Эффективность генерации тестов критически зависит от структуры промптов, предоставляемых языковой модели. Исследование TestPilot [6] на 25 npm-пакетах продемонстрировало медианное тестовое покрытие 68,2% при использовании промптов, включающих полный контекст функции. При этом эмпирическая оценка показала, что все компоненты промпта (сигнатура, документация, примеры использования, исходный код) вносят существенный вклад в качество результата. Для предлагаемой методологии UI-тестирования это означает необходимость включения в промпт не только структуры интерфейса, но и описания ожидаемого поведения, типичных сценариев взаимодействия и граничных случаев.

ChatUniTest [7] предлагает механизм адаптивного фокального контекста, который решает проблему ограниченного размера контекстного окна модели (максимальный объём текста (в токенах), который модель может обработать за один запрос). Механизм динамически определяет, какая информация наиболее релевантна для конкретной задачи генерации, и включает её в промпт с приоритетом. В контексте UI-тестов такой подход позволяет приоритизировать критичные элементы интерфейса (формы ввода, кнопки действий) и бизнес-логику взаимодействий над второстепенными элементами визуального оформления.

Эффективность различных стратегий промптирования была исследована в рамках развёртывания TestGen-LLM [8]. Эксперименты на 86 Kotlin-компонентах показали, что различные типы промптов вносят уникальный вклад в общий результат. Промпт типа "extend_coverage" (расширение покрытия) добавил 5 уникальных тестов, "corner_cases" (граничные случаи) — 1 уникальный тест,

"extend_test" (расширение существующих тестов) — 2 уникальных теста. Это подтверждает целесообразность ensemble-подхода с использованием нескольких стратегий промптирования для достижения максимального тестового покрытия.

В предлагаемой методологии применяются специализированные шаблоны промптов для каждого этапа генерации. Для создания Page Object промпт фокусируется на структурных аспектах интерфейса и типах элементов. Для генерации тестовых сценариев промпт включает описание ожидаемого поведения и граничных условий. Для этапа верификации промпт дополняется информацией об ошибках из предыдущих попыток выполнения, что обеспечивает адаптивное исправление.

Этап верификации с последовательной фильтрацией

Этап верификации реализует систему последовательных фильтров, каждый из которых проверяет определённое свойство сгенерированного теста. Подход вдохновлён архитектурой TestGen-LLM [8], успешно применённой в промышленных условиях для улучшения существующих тестов. В процессе развёртывания было обработано 32,531 попытка генерации, из которых только 4% (1,321 тест) успешно прошли все фильтры и были приняты разработчиками. Эта статистика демонстрирует строгость верификационного процесса и необходимость множественных проверок для гарантии качества.

Первый фильтр проверяет синтаксическую корректность сгенерированного кода. ChatUniTest [7] использует специализированные парсеры для верификации синтаксиса и сообщает о 75% успешности на этом этапе при тестировании на проектах Apache Commons-CLI и Commons-CSV. Синтаксические ошибки, обнаруживаемые на данном этапе, часто связаны с незакрытыми скобками, неверными отступами или превышением лимита токенов, приводящим к усечению сгенерированного кода.

Второй фильтр проверяет компилируемость или выполнимость теста в целевой среде. Этот этап выявляет семантические проблемы: отсутствие необходимых импортов, обращение к несуществующим методам или переменным, несоответствие типов. Типичная ошибка "cannot find symbol" часто

решается добавлением соответствующих импортов. Для UI-тестов компиляционная проверка дополнительно верифицирует корректность селекторов элементов и доступность используемых методов фреймворка автоматизации.

Третий фильтр выполняет тест и проверяет его прохождение. Критически важным аспектом является обнаружение нестабильных (flaky) тестов — тестов, которые проходят в некоторых случаях и падают в других при идентичных условиях выполнения. TestGen-LLM использует методологию множественного запуска: тест выполняется минимум 5 раз, и только тесты, прошедшие все запуски, допускаются к дальнейшему использованию. Эксперименты показали, что 75% сгенерированных тестов успешно компилируются, 57% проходят надёжно (без flakiness), и только 25% увеличивают тестовое покрытие функциональности. Эти цифры демонстрируют существенное сокращение кандидатов на каждом этапе фильтрации.

Проблема нестабильных тестов особенно актуальна для UI-тестирования в силу асинхронности операций интерфейса, вариативности времени загрузки элементов и зависимости от состояния браузера или устройства. В статье TestGen-LLM отмечается, что flaky-тесты представляют одну из наиболее значительных проблем для промышленного тестирования, и их автоматическая генерация недопустима.

Четвёртый фильтр измеряет прирост тестового покрытия кода или функциональности. В TestGen-LLM применяется строгий критерий: любой тест, не увеличивающий тестовое покрытие по сравнению со всеми существующими тестами, отбрасывается как избыточный. Этот подход гарантирует, что каждый принятый тест вносит уникальный вклад в тестовое покрытие. В контексте промышленного развёртывания за период с октября по декабрь 2023 года TestGen-LLM успешно улучшил приблизительно 10% всех тест-классов, к которым был применён, причём 73% рекомендаций были приняты разработчиками.

Пятый фильтр проверяет наличие и качество утверждений (assertions) в тесте. Тест без утверждений или с тривиальными утверждениями может

достигать высокого тестового покрытия, но не предоставляет полезных проверок корректности поведения. TestPilot [6] определяет нетривиальное утверждение как утверждение, зависящее по меньшей мере от одной функции тестируемого компонента. Исследование показало, что 58,5% сгенерированных тестов содержат нетривиальные утверждения, причём именно эти тесты обеспечивают основную долю полезного тестового покрытия: медианное тестовое покрытие от нетривиальных тестов составляет 62,7% против 68,2% от всех тестов, что указывает на незначительный вклад тривиальных проверок.

ChatUniTest [7] использует метод обратного слайсинга программы (backward program slicing) для автоматической идентификации нетривиальных утверждений. Система вычисляет обратный слайс от каждого утверждения и проверяет, содержит ли он импорт тестируемого компонента. Этот подход позволяет автоматически отфильтровать тесты с утверждениями, не связанными с целевой функциональностью.

Шестой фильтр использует мутационное тестирование. Мутационное тестирование является распространённым методом оценки качества тестовых наборов [12]. Исследование [13] на основе анализа 357 реальных дефектов из проектов Defects4J показало сильную корреляцию между mutation score (доля мутаций, обнаруженных тестовым набором, от общего числа введённых мутаций) и способностью обнаруживать реальные дефекты. Недавнее исследование [14] подтверждает практическую востребованность мутационного тестирования, выявив более 3500 активных git репозиториев, использующих данную технику.

Однако применение традиционного мутационного тестирования к UI-тестам сталкивается с проблемой создания мутаций на уровне пользовательского интерфейса. Модификация исходного кода компонента может не отражать типичные дефекты пользовательского интерфейса, связанные с обработкой событий, валидацией ввода или навигацией между состояниями. Кроме того, классический подход требует пересборки приложения для каждой мутации, что плохо совместимо с многоступенчатыми процессами сборки современных веб-приложений, включающими транспиляцию и бандлинг. В связи с этим в настоящей работе предлагается метод, выполняющий инъекцию

дефектов непосредственно в среду исполнения браузера, не затрагивая исходные файлы приложения.

Предлагаемая методология реализует мутационное тестирование посредством инъекции перехватчиков событий в DOM (Document Object Model) страницы. В отличие от традиционных подходов, такой механизм не требует модификации исходного кода и перекомпиляции: перехватчик регистрируется в фазе захвата события непосредственно во время выполнения теста. Он позволяет заблокировать выполнение критического действия без изменения визуального состояния элемента: пользователь (или автоматизированный тест) может взаимодействовать с элементом, элемент визуально реагирует на взаимодействие, но целевое действие не выполняется. Тест, корректно проверяющий результат действия, обнаружит отсутствие ожидаемого эффекта и завершится с ошибкой.

Процесс верификации включает следующие шаги: определение критического действия в тесте, инъекция перехватчика для блокировки этого действия, повторное выполнение теста и анализ результата. Тест считается качественным, если он падает при блокировке критического действия. Тест, продолжающий проходить при заблокированном действии, содержит недостаточные проверки и не гарантирует обнаружение реального дефекта.

Определение критического действия выполняется языковой моделью на основе анализа кода теста и контекста из UI-спецификации. Модель идентифицирует элемент, действие над которым является ключевым для проверяемой функциональности, и возвращает соответствующий селектор. Точность определения зависит от полноты описания функциональности в спецификации.

Данный подход к мутационному тестированию пользовательского интерфейса специфичен для действий, инициируемых пользовательским взаимодействием. Функциональность, активируемая таймерами, WebSocket-сообщениями или другими источниками событий, требует альтернативных методов верификации. Текущая реализация сфокусирована на наиболее

распространённых сценариях пользовательского взаимодействия: клики, ввод данных, отправка форм, навигация.

Кандидаты тестовых случаев, прошедшие через все фильтры, гарантированно улучшают существующий набор тестов и предоставляют надёжный сигнал для регрессионного тестирования. Такая строгость фильтрации устраняет необходимость ручной валидации каждого сгенерированного теста, что критически важно для масштабного промышленного применения.

Этап автоматического исправления (self-healing)

Если тест не проходит верификацию, запускается процесс автоматического исправления. Концепция self-healing в автоматизации тестирования предполагает автономное обнаружение, диагностику и исправление сбоев без ручного вмешательства [9]. Предлагаемая методология использует двухуровневую стратегию, аналогичную реализованной в ChatUniTest: исправление на основе правил для известных типов ошибок и исправление с использованием языковой модели для сложных случаев.

Первый уровень применяет правила исправления для типичных категорий ошибок. Для ошибок типа "cannot find symbol" система копирует все операторы импорта из целевого класса в сгенерированный тест; для усечённых тестов, возникающих при превышении лимита токенов, система добавляет недостающие закрывающие скобки. Эти простые правила эффективно устраняют наиболее частые проблемы генерации без необходимости повторного обращения к языковой модели.

Для UI-тестов набор правил расширяется специфическими проблемами веб-автоматизации: таймауты ожидания элементов, устаревшие селекторы, отсутствие явных ожиданий завершения асинхронных операций. Правила первого уровня способны автоматически корректировать таймауты, добавлять ожидания видимости элементов и исправлять простые ошибки в селекторах.

Второй уровень использует языковую модель для исправления ошибок, не обнаруженных правилами. Методология основана на адаптивном компоненте

TestPilot: модель повторно промптируется с включением падающего теста и сообщения об ошибке. Эксперименты показали, что адаптивный подход способен исправить в среднем 16.3% падающих тестов. Более детальная статистика по типам ошибок демонстрирует различную эффективность для разных категорий проблем: timeout-ошибки исправляются в 30.3% случаев (часто добавлением недостающего вызова завершения теста), assertion-ошибки — в 16,7% случаев (корректировка ожидаемых значений на основе фактических результатов выполнения).

ChatUniTest формулирует промпт исправления, интегрирующий тип ошибки, сообщение об ошибке и контекст некорректного теста. Система ограничивает количество итераций исправления (обычно 3-5 попыток), чтобы избежать бесконечных циклов и неоправданного расхода вычислительных ресурсов. Тесты, не исправленные за установленное количество попыток, как правило требуют изменения логики или доступа к контексту, недоступному языковой модели.

Качество исправленных тестов верифицируется повторным прохождением через все фильтры. Это гарантирует, что автоматически исправленный тест удовлетворяет тем же критериям качества, что и успешно сгенерированный с первой попытки.

Реализация

Реализация предлагаемой методологии основана на выборе технологий, обеспечивающих воспроизводимость результатов, возможность локального развёртывания и совместимость с современными практиками разработки веб-приложений.

В качестве языковой модели выбрана Llama 3.3 70B Instruct. Выбор обусловлен несколькими факторами. Во-первых, модель распространяется под открытой лицензией, что обеспечивает воспроизводимость экспериментов и возможность верификации результатов независимыми исследователями. Во-вторых, открытая модель допускает локальное развёртывание, что существенно для организаций с требованиями к конфиденциальности данных: исходный код приложения и спецификации не передаются внешним сервисам.

Взаимодействие с моделью реализовано через стандартный API, совместимый с форматом OpenAI. Это обеспечивает возможность замены модели без изменения кодовой базы системы. Для каждого этапа генерации разработаны специализированные промпты, учитывающие контекст задачи и формат ожидаемого результата.

Архитектура системы и библиотека компонентов

Фреймворк автоматизации тестирования Playwright выбран в качестве целевой платформы для генерируемых тестов. Выбор обусловлен несколькими ключевыми преимуществами, критичными для контекста автоматической генерации UI-тестов.

Во-первых, Playwright обеспечивает кроссбраузерность через единый API для Chromium, Firefox и WebKit (Safari). В отличие от Selenium, требующего различных драйверов для каждого браузера, Playwright использует протокол DevTools для прямого управления браузерами, что обеспечивает более стабильное выполнение и снижает вероятность flaky-тестов. Это критично в контексте автоматической генерации, где отсутствует ручная отладка нестабильных тестов.

Во-вторых, Playwright предоставляет встроенные механизмы автоматического ожидания (auto-waiting), которые существенно снижают количество ошибок, связанных с синхронизацией. Перед каждым действием фреймворк автоматически ожидает выполнения условий готовности элемента: присутствие в DOM, видимость, доступность для взаимодействия, стабильность (отсутствие перемещения или анимации). Эта функциональность устраняет необходимость явного указания ожиданий в большинстве случаев, что снижает сложность сгенерированного кода и повышает его надёжность.

В-третьих, Playwright предоставляет мощные возможности для работы с современными Single Page Application: перехват и модификацию сетевых запросов, эмуляцию геолокации и устройств, работу с Web Workers и Service Workers, выполнение произвольного JavaScript в контексте страницы. Последняя

возможность критична для реализации механизма мутационного тестирования через DOM-инъекции.

В-четвёртых, Playwright имеет обширную и хорошо структурированную документацию с примерами кода на TypeScript, что упрощает задачу языковой модели при генерации корректного кода. Модель может использовать паттерны из документации как референсные примеры, повышая вероятность генерации синтаксически корректного и идиоматического кода.

В-пятых, фреймворк активно развивается с регулярными релизами и поддержкой современных веб-стандартов. Playwright поддерживает актуальные версии браузеров без необходимости обновления драйверов, что критично для долгосрочной поддерживаемости сгенерированных тестов.

Тесты генерируются на языке TypeScript, обеспечивающем статическую типизацию и улучшенную поддержку в средах разработки. Типизация способствует раннему обнаружению ошибок и упрощает рефакторинг сгенерированного кода при необходимости ручной доработки. Статическая типизация также упрощает задачу языковой модели: явное указание типов параметров и возвращаемых значений снижает вероятность генерации семантически некорректного кода.

Архитектура системы построена по модульному принципу и включает шесть основных компонентов. Модуль парсинга UI-спецификации отвечает за извлечение структурированной информации из документов на естественном языке, идентифицируя описания элементов интерфейса, их свойства и ожидаемое поведение. Модуль взаимодействия с языковой моделью инкапсулирует логику формирования промптов, отправки запросов к API модели и парсинга ответов. Генератор Page Object создаёт классы, представляющие страницы и компоненты интерфейса, на основе обработанной спецификации. Генератор тестов формирует тестовые сценарии, используя сгенерированные Page Object и описания поведения из спецификации. Движок верификации выполняет последовательную проверку сгенерированных тестов по критериям синтаксической корректности, компилируемости, надёжности выполнения, увеличения тестового покрытия и качества утверждений. Модуль интеграции с

системой отчётности обеспечивает формирование детализированных отчётов о результатах генерации и верификации.

Компоненты взаимодействуют через чётко определённые интерфейсы, что обеспечивает возможность независимой модификации и тестирования каждого модуля. Такая архитектура упрощает интеграцию альтернативных реализаций отдельных компонентов, например, замену языковой модели или фреймворка автоматизации без изменения остальных частей системы.

Библиотека базовых компонентов составляет фундамент архитектуры генерируемых Page Object. Библиотека предоставляет набор переиспользуемых классов, абстрагирующих типовые операции с элементами интерфейса. Базовый абстрактный класс определяет общий интерфейс для всех компонентов, включающий методы ожидания появления элемента в DOM, проверки его видимости, выполнения взаимодействия и получения атрибутов. Специализированные классы для различных типов элементов (кнопки, поля ввода, выпадающие списки, чекбоксы, радиокнопки) наследуются от базового класса и реализуют специфичные для типа операции.

Использование паттерна Page Object Model для структурирования UI-тестов является установленной практикой, повышающей поддерживаемость тестового кода [1]. Инкапсуляция логики взаимодействия в компоненты также упрощает задачу языковой модели при генерации Page Object. Вместо детального описания последовательности низкоуровневых операций (локализация элемента, ожидание видимости, выполнение действия, проверка результата) модель оперирует высокоуровневыми концепциями. Это снижает вероятность ошибок генерации, повышает читаемость сгенерированного кода и упрощает его понимание при необходимости ручной модификации.

Механизм верификации через DOM-инъекции

Верификация способности тестов обнаруживать дефекты реализована посредством динамической инъекции перехватчиков событий в DOM страницы. Механизм основан на использовании возможности выполнения произвольного JavaScript-кода в контексте тестируемой веб-страницы. Система регистрирует

обработчик события в фазе захвата (capture phase) — ранней фазе распространения события в DOM, предшествующей обработчикам, установленным веб-приложением.

Инъецированный обработчик события выполняет две ключевые операции. Во-первых, он останавливает дальнейшее распространение события, предотвращая вызов обработчиков, установленных веб-приложением. Во-вторых, он отменяет действие по умолчанию для элемента, блокируя встроенное поведение браузера. Визуально элемент продолжает реагировать на взаимодействие пользователя: кнопка может менять цвет при наведении, отображать эффект нажатия при клике. Однако целевое действие — отправка формы на сервер, переход по ссылке, вызов API-метода, изменение состояния приложения — не выполняется.

Результат инъекции создаёт мутацию поведения приложения, аналогичную введению реального дефекта в обработчик события. С точки зрения пользователя (или автоматизированного теста) элемент выглядит работающим, но фактически не выполняет своей функции. Тест, корректно проверяющий результат действия (например, проверяющий отправку данных на сервер или появление сообщения об успехе), обнаружит отсутствие ожидаемого эффекта и завершится с ошибкой. Напротив, тест, проверяющий только визуальное состояние элемента или не проверяющий результат действия вовсе, продолжит выполнение, создавая ложное впечатление успешного прохождения и демонстрируя свою неспособность обнаружить реальный дефект.

Преимуществом данного подхода к мутационному тестированию является его неинвазивность относительно исходного кода приложения. Традиционное мутационное тестирование требует модификации исходного кода путём внесения мутаций (замены операторов, изменения констант, удаления условий), перекомпиляции и повторного запуска тестов для каждой мутации [12]. Для сложных веб-приложений с многоступенчатым процессом сборки, включающим транспиляцию, минификацию, бандлинг и code splitting, такой подход может быть затруднителен или невозможен без значительных модификаций

инфраструктуры сборки. DOM-инъекция выполняется в runtime, в момент выполнения теста, без изменения исходных файлов приложения.

Ограничением подхода является его применимость преимущественно к функциональности, активируемой пользовательскими событиями DOM: клики, двойные клики, ввод данных, отправка форм, drag-and-drop операции. Функциональность, инициируемая таймерами (setTimeout, setInterval), WebSocket-сообщениями, загрузкой данных через fetch или XMLHttpRequest, изменениями в IndexedDB или другими асинхронными источниками, не связанными с DOM-событиями, требует альтернативных методов мутационного тестирования. Текущая реализация сфокусирована на наиболее распространённых сценариях пользовательского взаимодействия, которые составляют основную часть UI-тестов веб-приложений. Для более сложных сценариев возможно расширение подхода путём инъекции mock-объектов для API браузера или перехвата асинхронных операций через Service Workers.

Процесс верификации включает четыре последовательных шага. На первом шаге языковая модель анализирует код теста и контекст из UI-спецификации для определения критического действия — действия, от успешности которого зависит корректность проверяемой функциональности. Модель получает исходный код тестовой функции и релевантный фрагмент UI-спецификации, описывающий назначение элементов и ожидаемое поведение. На выходе модель возвращает селектор элемента (CSS-селектор, XPath или текстовое содержимое) и тип события для блокировки (click, submit, change и т.д.). Точность определения критического действия зависит от полноты описания функциональности в спецификации: чем детальнее описан ожидаемый результат действия, тем точнее модель идентифицирует ключевое взаимодействие.

На втором шаге система выполняет инъекцию перехватчика для идентифицированного элемента и события. Инъекция происходит непосредственно перед началом выполнения теста, обеспечивая блокировку целевого действия на всём протяжении тестового сценария. На третьем шаге тест выполняется повторно в условиях заблокированного действия. На четвёртом шаге система анализирует результат повторного выполнения. Тест считается

качественным и способным обнаруживать дефекты, если он завершается с ошибкой при заблокированном критическом действии. Тест, продолжающий проходить успешно, маркируется как недостаточно проверяющий функциональность и требующий доработки.

Интеграция с системой отчётности

Интеграция с системой отчётности Allure обеспечивает наглядное представление результатов генерации и верификации тестов. Allure формирует HTML-отчёты с детализированной информацией о каждом этапе процесса, включая древовидную структуру тестовых наборов, статистику прохождения, историю выполнения и приложенные артефакты.

Структура шагов теста формируется автоматически посредством встроенной в методы базовых компонентов функциональности логирования. Каждое действие пользователя, выполненное через методы компонентов (клик по кнопке, ввод текста в поле, выбор элемента из списка), регистрируется как отдельный шаг в отчёте с описанием на естественном языке. Например, действие "нажать кнопку входа" отображается в отчёте как шаг "Click: Login button (#login-btn)", где указывается тип действия, семантическое название элемента и его селектор. Такая детализация упрощает понимание последовательности операций теста и ускоряет локализацию проблемы при падении.

При падении теста система автоматически прикрепляет к отчёту несколько типов диагностической информации. Во-первых, скриншот страницы в момент возникновения ошибки позволяет визуально оценить состояние интерфейса и выявить очевидные проблемы (отсутствие ожидаемого элемента, некорректное отображение). Во-вторых, снимок HTML-структуры DOM на момент ошибки предоставляет детальную информацию для анализа проблем с локаторами. В-третьих, лог консоли браузера, включающий JavaScript-ошибки и предупреждения, помогает идентифицировать проблемы на стороне приложения. Эта информация упрощает диагностику без необходимости повторного воспроизведения падения теста.

Для тестов, прошедших цикл автоматического исправления, отчёт содержит специальный раздел с историей итераций. Для каждой итерации фиксируется тип обнаруженной ошибки (синтаксическая, компиляционная, runtime), текст сообщения об ошибке, применённая стратегия исправления (rule-based или LLM-based) и внесённые изменения в код теста. Эта информация обеспечивает прозрачность процесса автоматического исправления и позволяет оценить эффективность различных стратегий восстановления для разных типов ошибок.

Результаты верификации отображаются в отдельном разделе отчёта со структурированной статистикой по каждой фазе проверки. Статистика включает: количество тестов, успешно сгенерированных и прошедших с первой попытки без исправлений; количество тестов, исправленных автоматически с указанием количества итераций и применённых стратегий; количество тестов, не прошедших автоматическое исправление и требующих ручного вмешательства с детализацией причин; результаты мутационного тестирования с указанием тестов, успешно обнаруживших введённые мутации, и тестов, продолжающих проходить при заблокированном критическом действии.

Экспериментальная проверка методологии на платформе Open MCT

Для эмпирической валидации предложенной трёхэтапной методологии был проведён пилотный эксперимент на открытой платформе Open MCT (Open Mission Control Technologies) — веб-приложении, разработанном NASA Ames Research Center для визуализации телеметрии, мониторинга состояния космических аппаратов и оперативного управления полётными заданиями. Выбор объекта обусловлен его прямым соответствием авиационно-космической тематике, наличием нестандартных элементов интерфейса (интерактивные временные шкалы, составные индикаторы, динамически конфигурируемые панели), а также открытой кодовой базой, допускающей локальное развёртывание и неинвазивное мутационное тестирование.

Тестовое окружение и конфигурация объекта

Эксперимент проводился на стабильной версии Open MCT 4.2.1, развёрнутой локально на Node.js 22. Приложение функционирует как одностраничное (SPA) на базе HTML/CSS/JavaScript с модульной архитектурой. Для тестирования были выбраны три типовые функциональные области, характерные для рабочих мест операторов космических миссий:

- Управление объектами в древовидном навигаторе (создание, переименование, удаление элементов через контекстное меню);
- Компоновка отображаемых панелей (добавление виджетов «AlphaNumeric», «Plot», «Gauge» на холст, изменение их расположения);
- Работа с временной шкалой (навигация, масштабирование, фиксация временных маркеров).

Интерфейс содержит 23 уникальных типа интерактивных компонентов, включая нестандартные: SVG-индикаторы, динамически обновляемые таблицы, ползунки временной шкалы, контекстные меню, вызываемые правой кнопкой мыши. UI-спецификация составлялась аналитиком, не имеющим опыта программирования, и содержала 76 строк структурированного текста на естественном языке, описывающих 9 сквозных пользовательских сценариев (5 позитивных, 4 негативных, включая сценарии с отсутствием прав на удаление объектов).

Генерация выполнена с использованием модели Llama 3.3 70B Instruct, развёрнутой на локальном сервере. Фреймворк исполнения тестов — Playwright 1.48, целевой браузер — Chromium 128 (headless). Стенд: Ubuntu 24.04 LTS, CPU AMD Ryzen 9 7950X (16 ядер), GPU NVIDIA RTX 4090 (24 ГБ), 128 ГБ ОЗУ.

Метрики и процедура оценивания

Для количественной оценки результативности использовалась многоуровневая система метрик, согласующаяся с подходами TestGen-LLM [8] и ChatUniTest [7]:

- SR_syn — доля тестов, прошедших синтаксический анализ парсером TypeScript без ошибок.

- SR_comp — доля тестов, успешно скомпилировавшихся (с учётом импортов Page Object).
- SR_stable — доля тестов, продемонстрировавших стабильное прохождение (5 идентичных запусков, 0 отказов).
- SR_cov — доля тестов, увеличивших тестовое покрытие сценариев из UI-спецификации (каждый тест должен закрывать хотя бы один новый шаг сценария, не закрытый ранее принятыми тестами).
- SR_mut — доля тестов, обнаруживших хотя бы одну DOM-мутацию из множества критических (mutation score на уровне теста > 0%).
- Mutation Score (MS) — доля обнаруженных мутаций от общего числа внесённых (финальный набор тестов).

Процедура полностью автоматизирована. Для каждого сценария генерировалось до 12 тестов-кандидатов. Тест, не прошедший очередной фильтр, направлялся на автоматическое исправление (максимум 4 итерации; первые две — rule-based: добавление явных ожиданий, подстановка импортов; последующие две — LLM-based с включением сообщения об ошибке в промпт). По исчерпании лимита попыток тест исключался.

Количественные результаты генерации и фильтрации

Суммарно системой было инициировано 108 попыток генерации (12 попыток на каждый из 9 сценариев). Динамика прохождения по этапам фильтрации представлена в таблице 1.

Таблица 1

Результаты последовательной фильтрации тестовых кандидатов для Open MCT

Этап фильтрации	Кол-во тестов	% от предыдущего этапа	% от исходных попыток
Сгенерировано кандидатов	108	—	100.0%
Синтаксически корректны (SR_syn)	101	93.5%	93.5%
Успешно компилируются (SR_comp)	88	87.1%	81.5%
Стабильны при 5 запусках (SR_stable)	65	73.9%	60.2%
Увеличивают тестовое покрытие сценариев (SR_cov)	51	78.5%	47.2%
Прошли мутационное тестирование (SR_mut)	39	76.5%	36.1%

Интегральная эффективность методологии составила 36.1%: 39 из 108 сгенерированных тестов прошли все стадии верификации без ручного вмешательства и продемонстрировали способность обнаруживать искусственно внесённые дефекты. Результирующий набор тестирует 8 из 9 заявленных сценариев (88.9%). Единственный непротестированый сценарий («удаление объекта при отсутствии прав с проверкой всплывающего уведомления») требовал обработки события, инициируемого через WebSocket-уведомление, что выходит за рамки текущего механизма DOM-инъекций.

Средняя эффективность автоматического исправления составила: rule-based стратегия — восстановление 63% тестов, отсеянных на этапах компиляции и стабильности (преимущественно добавление waitForSelector и исправление локаторов); LLM-based стратегия — восстановление 28% тестов после трёх попыток. Наиболее частой причиной неустранимых отказов являлись логические ошибки утверждений (некорректные ожидаемые значения), которые модель не могла скорректировать без дополнительного контекста предметной области.

Мутационное тестирование через DOM-инъекции

Для финального набора из 39 тестов было сформировано 72 критические DOM-мутации, охватывающие типовые дефекты UI уровня:

- Блокировка обработчика click на кнопках создания/удаления объектов;
- Блокировка обработчика contextmenu (правая кнопка) для элементов навигатора;
- Блокировка события change на ползунке масштабирования временной шкалы;
- Блокировка submit/dragend для манипуляций с виджетами.

Тесты обнаружили 67 мутаций, что соответствует **Mutation Score = 93.1%**. Пять необнаруженных мутаций относились к сценариям, где тест проверял лишь видимость элемента после действия, но не верифицировал изменение состояния приложения (например, виджет добавлялся на холст, но тест не проверял его фактическое присутствие в DOM-дереве новой панели). Это подтвердило

релевантность мутационного фильтра как средства отсева тестов с недостаточными утверждениями.

Качественный анализ: пример сценария «Создание объекта навигации»

Фрагмент UI-спецификации, поданной на вход модели:

«В левой части экрана расположен навигатор объектов (древовидный список). При нажатии правой кнопкой мыши на корневом элементе “My Items” появляется контекстное меню с пунктом “Create”. После выбора “Create” в дереве появляется новый объект с именем “Unnamed Object”, а его свойства открываются в центральной панели.»

По этому описанию моделью генерируется Page Object класс (рисунок 1), а затем тест (рисунок 2).

```
export class NavigatorPane extends BaseComponent {
  private readonly treeItem = (label: string) =>
    this.page.locator(`.c-tree__item__label:text-is("${label}")`);
  private readonly contextMenu = this.page.locator('.c-menu');
  private readonly createOption = this.page.locator('text=Create');

  async createChildObject(parentLabel: string) {
    await this.treeItem(parentLabel).click({ button: 'right' });
    await this.contextMenu.waitFor({ state: 'visible' });
    await this.createOption.click();
  }
}
```

Рисунок 1 – Сгенерированный Page Object

```
test('Создание объекта через контекстное меню навигатора', async ({ page }) => {
  const navigator = new NavigatorPane(page);
  await page.goto('/#/browse/mine');
  await navigator.createChildObject('My Items');
  const newItem = page.locator('.c-tree__item__label:text-is("Unnamed Object")');
  await expect(newItem).toBeVisible();
  await expect(page.locator('.c-object-view')).toContainText('Unnamed Object');
});
```

Рисунок 2 – Сгенерированный тест

При инъекции мутации (перехват и отмена события click на пункте «Create») тест ожидаемо падает на проверке toBeVisible для нового элемента с интервалом таймаута, корректно сигнализируя о нефункциональности обработчика

контекстного меню. Это демонстрирует работоспособность механизма верификации на реальном интерфейсе.

Заключение

Теоретически показано, что сочетание предложенной архитектуры (Page Object Model на базе библиотеки компонентов), открытой LLM и процедуры верификации позволяет получать исполняемые проверки напрямую из эксплуатационной документации на естественном языке. Это снижает зависимость от дефицитных специалистов по автоматизации, одновременно повышая стабильность проверок нестандартных элементов, характерных для авиационных и космических информационных систем.

Конфликт интересов

Автор заявляет об отсутствии конфликта интересов.

Conflict of interest

The author declares no conflict of interest.

Благодарности

Автор выражает глубокую признательность доценту кафедры компьютерные науки и прикладная математика Московского авиационного института Владимиру Николаевичу Лукину за помощь в редактировании статьи.

Acknowledgments

The author expresses deep gratitude to Vladimir Nikolaevich Lukin, Associate Professor of the Department of Computer Science and Applied Mathematics at the Moscow Aviation Institute, for his assistance in editing the article.

Список источников

1. Maurizio Leotta, Diego Clerissi, Filippo Ricca, Cristiano Spadaro Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study// 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, 2013, pp. 108-113, DOI:10.1109/ICSTW.2013.19

2. Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, Qing Wang Software Testing with Large Language Models: Survey, Landscape, and Vision, 2023, DOI: 10.48550/arXiv.2307.07221

3. Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, Claudio Bartolini Lopes LLMs for Automated Unit Test Generation and Assessment in Java: The AgoneTest Framework, 2025, DOI: 10.48550/arXiv.2511.20403

4. Saranya Alagarsamy, Chakkrit Tantithamthavorn, Aldeida Aleti A3Test: Assertion-Augmented Automated Test Case Generation, 2023, DOI: 10.48550/arXiv.2302.10352

5. Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, Junjie Chen On the Evaluation of Large Language Models in Unit Test Generation, 2024, DOI: 10.48550/arXiv.2406.18181

6. Max Schäfer, Sarah Nadi, Aryaz Eghbali, Frank Tip Adaptive Test Generation Using a Large Language Model, 2023, DOI: 10.48550/arXiv.2302.06527

7. Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, Jianwei Yin ChatUniTest: A Framework for LLM-Based Test Generation, 2024, DOI: 10.48550/arXiv.2305.04764

8. Eleanor Foley, Ashley Jacob, Ronit Kapoor Generating Test Cases Through Large Language Models, 2024, URL: https://digital.wpi.edu/downloads/pv63g461d

9. Sutharsan Saarathy, Suresh Bathrachalam, Rajendran Bharath Self-Healing Test Automation Framework using AI and ML // International Journal of Strategic Management, 2024, vol 3, pp. 45-77, DOI: 10.47604/ijsm.2843

10. Zemin Su, Cuiqin Bai, Shaowen Wei, Kangyong Liu, Yang Liu, Zhenxuan Huan Self-Healing UI Test Automation via Multi-Modal Fusion, 2025, URL: https://ieeexplore.ieee.org/abstract/document/11047603

11. Zejun Wang, Kaibo Liu, Ge Li, Zhi Jin HITS: High-coverage LLM-based Unit Test Generation via Method Slicing, 2024, DOI: 10.48550/arXiv.2408.11324

12. Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, Mark Harman Mutation Testing Advances: An Analysis and Survey // Advances in Computers, vol 112, pp. 275-378, DOI: 10.1016/bs.adcom.2018.03.015

13. René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, Gordon Fraser Are mutants a valid substitute for real faults in software testing? // FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 654-665, URL: https://www.researchgate.net/publication/277664637_Are_Mutants_a_Valid_Substitute_for_Real_Faults_in_Software_Testing

14. Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada, Medina-Bulo, Sergio Segura1 Mutation testing in the wild: findings from GitHub, 2022, URL: https://www.researchgate.net/publication/362078955_Mutation_testing_in_the_wild_findings_from_GitHub

References

1. Maurizio Leotta, Diego Clerissi, Filippo Ricca, Cristiano Spadaro Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study// 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, 2013, pp. 108-113, DOI:10.1109/ICSTW.2013.19

2. Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, Qing Wang Software Testing with Large Language Models: Survey, Landscape, and Vision, 2023, DOI: 10.48550/arXiv.2307.07221

3. Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, Claudio Bartolini Lopes LLMs for Automated Unit Test Generation and Assessment in Java: The AgoneTest Framework, 2025, DOI: 10.48550/arXiv.2511.20403

4. Saranya Alagarsamy, Chakkrit Tantithamthavorn, Aldeida Aleti A3Test: Assertion-Augmented Automated Test Case Generation, 2023, DOI: 10.48550/arXiv.2302.10352

5. Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, Junjie Chen On the Evaluation of Large Language Models in Unit Test Generation, 2024, DOI: 10.48550/arXiv.2406.18181

6. Max Schäfer, Sarah Nadi, Aryaz Eghbali, Frank Tip Adaptive Test Generation Using a Large Language Model, 2023, DOI: 10.48550/arXiv.2302.06527

7. Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, Jianwei Yin ChatUniTest: A Framework for LLM-Based Test Generation, 2024, DOI: 10.48550/arXiv.2305.04764

8. Eleanor Foley, Ashley Jacob, Ronit Kapoor Generating Test Cases Through Large Language Models, 2024, URL: https://digital.wpi.edu/downloads/pv63g461d

9. Sutharsan Saarathy, Suresh Bathrachalam, Rajendran Bharath Self-Healing Test Automation Framework using AI and ML // International Journal of Strategic Management, 2024, vol 3, pp. 45-77, DOI: 10.47604/ijsm.2843

10. Zemin Su, Cuiqin Bai, Shaowen Wei, Kangyong Liu, Yang Liu, Zhenxuan Huan Self-Healing UI Test Automation via Multi-Modal Fusion, 2025, URL: https://ieeexplore.ieee.org/abstract/document/11047603

11. Zejun Wang, Kaibo Liu, Ge Li, Zhi Jin HITS: High-coverage LLM-based Unit Test Generation via Method Slicing, 2024, DOI: 10.48550/arXiv.2408.11324

12. Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, Mark Harman Mutation Testing Advances: An Analysis and Survey // Advances in Computers, vol 112, pp. 275-378, DOI: 10.1016/bs.adcom.2018.03.015

13. René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, Gordon Fraser Are mutants a valid substitute for real faults in software testing? // FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 654-665, URL: https://www.researchgate.net/publication/277664637_Are_Mutants_a_Valid_Substitute_for_Real_Faults_in_Software_Testing

14. Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada, Medina-Bulo, Sergio Segura1 Mutation testing in the wild: findings from GitHub, 2022, URL: https://www.researchgate.net/publication/362078955_Mutation_testing_in_the_wild_findings_from_GitHub

Информация об авторах

Александр Максимович Титеев, аспирант, Московский авиационный институт (национальный исследовательский университет); г. Москва, Россия; ORCID: https://orcid.org/0009-0003-7754-1550; e-mail: loksader@yandex.ru

Information about the authors

Alexandr M. Titeev, post graduate student, Moscow Aviation Institute (National Research University); Moscow, Russia; ORCID: https://orcid.org/0009-0003-7754-1550; e-mail: loksader@yandex.ru

Получено 18 декабря 2025 ● Принято к публикации 13 мая 2026 ● Опубликовано 25 июня 2026
Received 18 December 2025 ● Accepted 13 May 2026 ● Published 25 June 2026