УДК 004.4
DOI: 10.26102/2310-6018/2026.62.11.011

Классификация дефектов многопоточных программ как основа построения требований к тестированию

Д.К. Шульга

Санкт-Петербургский политехнический университет Петра Великого, Санкт-Петербург, Российская Федерация

Резюме. Рост вычислительной мощности многоядерных систем, распространение серверных платформ, облачных сервисов и высоконагруженных приложений привели к тому, что многопоточность стала одним из базовых механизмов современной разработки программного обеспечения. Вместе с тем расширение сферы её применения сопровождалось увеличением числа трудно обнаруживаемых нарушений, связанных не только с логикой алгоритма, но и с порядком исполнения конкурентных действий, организацией синхронизации, публикацией состояния и распределением доступа к общим ресурсам. В статье рассматривается проблема классификации дефектов многопоточных программ как необходимая теоретическая основа для построения эффективных методов тестирования. Цель работы состоит в систематизации основных типов конкурентных нарушений и формулировании требований к тестированию, вытекающих из их природы. Использованы системный и сравнительный анализ, структурно-логическая классификация дефектов, обобщение требований к инструментальным средствам и имитационное моделирование конкурентного исполнения. Выделены основные группы дефектов: состояния гонки, взаимные блокировки, livelock, голодание, нарушения видимости памяти, ошибки синхронизации, атомарностные и порядковые нарушения, дефекты жизненного цикла потоков и ресурсные конкурентные нарушения. Показано, что такие дефекты обладают отложенной наблюдаемостью, зависят от конкретных вариантов interleaving и нередко проявляются лишь в редких сценариях. На этой основе сформулированы ключевые требования к тестированию: учёт пространства расписаний, воспроизводимость, наблюдаемость межпоточных взаимодействий, контроль прогресса, чувствительность к модели памяти, сочетание статических и динамических методов, приемлемые накладные расходы. Впервые сформулирована проверяемая гипотеза о вероятностной природе обнаруживаемости конкурентных дефектов, введены количественные характеристики дефекта (окно конкуренции и коэффициент проявления) и предложен профиль наблюдаемости как критерий полноты тестирования; имитационный эксперимент подтверждает, что обнаруживаемость определяется режимом планирования, а не числом прогонов.

Ключевые слова: многопоточные программы, состояния гонки, взаимные блокировки, синхронизация, тестирование программного обеспечения, конкурентное исполнение, окно конкуренции, коэффициент проявления, имитационное моделирование.

Для цитирования: Шульга Д.К. Классификация дефектов многопоточных программ как основа построения требований к тестированию. *Моделирование, оптимизация и информационные технологии.* 2026;14(11). URL: https://moitvivt.ru/ru/journal/article?id=2505 DOI: 10.26102/2310-6018/2026.62.11.011

Classification of defects in multithreaded programs as a basis for formulating testing requirements

D.K. Shulga

Peter the Great St. Petersburg Polytechnic University, St. Petersburg, the Russian Federation

© Шульга Д.К., 2026

Abstract. The growth of multicore computing power and the widespread use of server platforms, cloud services, and high-load applications have made multithreading one of the basic mechanisms of modern software development. At the same time, the expansion of its use has been accompanied by an increase in hard-to-detect faults related not only to algorithmic logic but also to the order of concurrent actions, synchronization, state publication, and access to shared resources. The paper examines the classification of defects in multithreaded programs as a necessary theoretical basis for building effective testing methods. The purpose of the study is to systematize the main types of concurrency-related faults and formulate testing requirements arising from their nature. The methods used include systems and comparative analysis, structural classification of defects, generalization of requirements for verification tools, and simulation of concurrent execution. The main groups of defects are identified: race conditions, deadlocks, livelock, starvation, memory visibility violations, synchronization errors, atomicity and ordering violations, thread lifecycle defects, and concurrent resource defects. It is shown that such defects are characterized by delayed observability, depend on specific interleavings, and often manifest themselves only in rare execution scenarios. On this basis, the key testing requirements are formulated: accounting for the scheduling space, reproducibility, observability of inter-thread interactions, progress control, sensitivity to the memory model, integration of static and dynamic methods, and acceptable overhead. The paper formulates a testable hypothesis on the probabilistic nature of concurrency-defect detectability, introduces quantitative defect characteristics (the concurrency window and the manifestation coefficient), and proposes the observability profile as a testing-completeness criterion; simulation results confirm that detectability is governed by the scheduling regime rather than by the number of runs.

Keywords: multithreaded programs, race conditions, deadlocks, synchronization, software testing, concurrent execution, concurrency window, manifestation coefficient, simulation modeling.

For citation: Shulga D.K. Classification of defects in multithreaded programs as a basis for formulating testing requirements. *Modeling, Optimization and Information Technology.* 2026;14(11). (In Russ.). URL: https://moitvivt.ru/ru/journal/article?id=2505 DOI: 10.26102/2310-6018/2026.62.11.011

Введение

Современная программная инженерия все в большей степени опирается на параллельное и конкурентное выполнение вычислений. Многопоточность используется в серверных приложениях, системах обработки данных, распределенных платформах, встроенном программном обеспечении, телекоммуникационных решениях и компонентах операционных систем. Ее применение позволяет эффективнее использовать вычислительные ресурсы, сокращать время отклика и поддерживать одновременное обслуживание большого числа запросов. Однако вместе с ростом функциональных возможностей возрастает и сложность обеспечения корректности таких систем.

Если в последовательной программе результат в основном определяется входными данными и логикой алгоритма, то в многопоточной среде на поведение системы начинает влиять порядок взаимодействия потоков. Существенным становится взаимное расположение чтений и записей, моменты захвата и освобождения блокировок, порядок публикации состояния, распределение доступа к ресурсам и решения планировщика. Именно поэтому многопоточные дефекты нередко проявляются нестабильно: в одних условиях программа работает корректно, а в других демонстрирует скрытые нарушения, зависания или непредсказуемые результаты [1].

В научной литературе накоплен значительный материал по анализу конкурентного исполнения, обнаружению состояний гонки, взаимных блокировок, нарушений синхронизации и иных классов ошибок. Развиваются методы статического и динамического анализа, техники систематического исследования расписаний, инструменты детерминированного воспроизведения и средства стресс-тестирования [2]. Тем не менее на практике по-прежнему сохраняется разрыв между теоретической

полнотой анализа и реальной пригодностью методов для промышленного использования. Одна из причин заключается в том, что средства тестирования нередко ориентированы на отдельные симптомы нарушений, но не опираются на целостную типологию конкурентных дефектов.

В этой связи особую значимость приобретает задача классификации дефектов многопоточных программ. Такая классификация важна не только сама по себе. Она позволяет определить, какие свойства исполнения должны становиться объектом проверки, какие сценарии interleaving следует считать критическими, какие метрики и критерии тестирования являются содержательно значимыми и каким образом должна строиться диагностическая часть инструментального анализа. Следовательно, без систематизации дефектов невозможно перейти к построению обоснованных требований к тестированию многопоточных программ.

Целью статьи является систематизация основных дефектов многопоточных программ и формулирование требований к тестированию, вытекающих из природы конкурентных нарушений.

Научная гипотеза и научная новизна. Работа опирается на проверяемую научную гипотезу, состоящую из двух взаимосвязанных положений. Первое положение: обнаруживаемость дефекта многопоточной программы подчиняется вероятностной закономерности. Если дефект проявляется в одном прогоне с вероятностью q, то при независимом порождении расписаний вероятность его обнаружения не менее чем за n прогонов описывается зависимостью $P(n) = 1 – (1 – q)^n$, а число прогонов, необходимое для обнаружения с достоверностью α, удовлетворяет неравенству $n \geq \ln(1 – \alpha) / \ln(1 – q)$. Второе положение: величина q не является неизменным внутренним свойством программы. Она определяется совместным действием структуры дефекта – ширины окна конкуренции w и числа конфликтующих потоков k – и режима планирования, прежде всего, интенсивности вытеснений π, то есть вероятности того, что планировщик прервет текущий поток внутри критического интервала.

Из гипотезы следует практически значимое следствие. При низкой интенсивности вытеснений, и в особенности при невытесняющем планировании, когда поток исполняется до завершения, критический интервал не прерывается, величина q стремится к нулю, и обнаруживаемость дефекта не может быть обеспечена никаким увеличением числа прогонов. Иными словами, объем стресс-тестирования не является самостоятельным критерием качества: требование должно формулироваться в терминах управляемого порождения расписаний, а не в терминах количества запусков. Проверке этого следствия посвящен имитационный эксперимент, описанный ниже.

Научная новизна исследования состоит в следующем.

Во-первых, предлагаемая классификация строится не на внешнем симптоме нарушения и не на наборе инструментальных шаблонов, как в известных обзорных работах, а на едином инварианте – структуре конкурентного исполнения, а именно на паре «множество конфликтующих операций – требуемое отношение порядка между ними». Именно структурой этого отношения различаются все выделенные классы дефектов, что позволяет описывать их в единой системе понятий и сопоставлять каждому классу наблюдаемые события исполнения, а не только внешние проявления.

Во-вторых, введены количественные характеристики дефекта, отсутствующие в обзорных классификациях: окно конкуренции w и коэффициент проявления q. Получено соотношение, связывающее эти характеристики с целевой достоверностью обнаружения и требуемым объемом тестирования, что позволяет сопоставлять классы дефектов по затратам на их выявление, а не только по описанию механизма нарушения.

В-третьих, предложено понятие профиля наблюдаемости – минимального набора фиксируемых событий и отношений между ними, необходимого и достаточного для

достоверной идентификации дефекта данного класса. Профиль наблюдаемости переводит требование наблюдаемости межпоточных взаимодействий из качественной рекомендации в проверяемый критерий полноты тестирования.

В-четвертых, получены оригинальные эмпирические результаты имитационного эксперимента, подтверждающие вероятностную модель обнаружения и показывающие, что решающим фактором обнаруживаемости является режим планирования, а не число прогонов. Тем самым обзорно-обобщающий материал дополнен в работе собственными формальными и эмпирическими результатами.

Материалы и методы

Методологической основой исследования послужили положения теории параллельных вычислений, современные представления о конкурентном исполнении программ, подходы к статическому и динамическому анализу кода, а также работы, посвященные вопросам надежности программного обеспечения в условиях многопоточности [3]. В работе использованы методы системного анализа, сравнительного анализа научных источников, структурно-логической классификации и теоретического обобщения.

Исследование строилось в несколько этапов. На первом этапе был проведен анализ особенностей многопоточного исполнения как среды возникновения специфических дефектов. На втором этапе были выделены основные типы нарушений, проявляющихся в условиях конкурентного доступа к данным, синхронизации и распределения ресурсов. На третьем этапе на основе полученной классификации были обобщены требования к тестированию, отражающие необходимость работы не только с входными данными, но и с пространством вариантов исполнения.

При анализе исходили из того, что дефект многопоточной программы не может рассматриваться исключительно как локальная ошибка кода. Его необходимо понимать как нарушение организации конкурентного исполнения, которое при одном или нескольких допустимых сценариях взаимодействия потоков приводит к некорректному состоянию, ошибочному результату, потере прогресса либо иной форме отклонения от ожидаемого поведения системы.

Для проверки сформулированной гипотезы использован имитационный эксперимент на абстрактной модели конкурентного исполнения. Исполнение рассматривается как последовательность квантов планирования, в каждом из которых выполняется одна операция одного из потоков; планировщик с вероятностью π выполняет вытеснение, переключаясь на случайно выбранный готовый поток, а с вероятностью $1 - \pi$ продолжает исполнение текущего потока. Такая параметризация позволяет единообразно описать как вытесняющее планирование ($\pi \to 1$), так и невытесняющее планирование с исполнением потока до завершения ($\pi = 0$). Исследовались два класса дефектов: нарушение целостности составной операции (потеря обновления при неатомарном инкременте общего счетчика; число потоков $k = 4$, ширина окна конкуренции $w = 0…16$) и взаимная блокировка, возникающая при инверсии порядка захвата двух ресурсов ($k = 2…8$, половина потоков захватывает ресурсы в обратном порядке). Для каждой конфигурации выполнялось не менее 12 000 независимых имитационных прогонов, по которым оценивался коэффициент проявления q; эмпирические зависимости $P(n)$ сопоставлялись с теоретическим законом, а по найденным значениям q рассчитывалось требуемое число прогонов n для достоверности $\alpha = 0{,}95$ и $\alpha = 0{,}99$. Эксперимент носит модельный характер: он не воспроизводит конкретную аппаратную платформу, язык программирования или

диспетчер операционной системы, а служит проверке структурных положений гипотезы о связи обнаруживаемости дефекта с режимом планирования.

Результаты

Специфика дефектов многопоточных программ. В последовательных программах структура ошибок обычно описывается через некорректные вычисления, ошибки алгоритма, нарушения логики ветвления, неправильную работу с памятью или ресурсами. В многопоточной среде к этим источникам добавляется особый класс нарушений, происхождение которого связано не с логикой одного вычислительного потока, а с отношениями между несколькими линиями исполнения. Именно этим объясняется высокая скрытность конкурентных дефектов, их нестабильность и зависимость от конкретного interleaving [4].

Для многопоточной программы важно не только то, какие действия выполняются, но и в каком порядке они становятся наблюдаемыми. Ошибка может быть заложена в структуре исполнения задолго до того, как проявится видимый сбой. Например, нарушение публикации данных возникает в одном фрагменте программы, а обнаруживается уже тогда, когда другой поток использует частично обновленное состояние. Ошибка координации может быть допущена на этапе обработки сигнала, а внешне проявиться как зависание на поздней стадии исполнения. Это означает, что многопоточный дефект обладает отложенной наблюдаемостью и нередко требует реконструкции причинной цепочки событий.

Классификация дефектов многопоточных программ. Результатом проведенного анализа стала классификация основных типов дефектов многопоточных программ, представленная в Таблице 1.

Таблица 1 – Основные типы дефектов многопоточных программ
Table 1 – Main types of multithreaded program defects

Тип дефекта	Содержание нарушения	Типичное проявление
Состояния гонки	Отсутствие требуемого порядка между конфликтующими обращениями к общему состоянию	Нестабильные результаты, потеря обновлений, чтение промежуточных значений
Взаимные блокировки	Циклическое ожидание ресурсов или событий	Полная остановка группы потоков
Livelock	Потоки остаются активными, но не обеспечивают полезный прогресс	Бесконечное повторение действий без достижения результата
Голодание	Отдельный поток систематически лишается доступа к ресурсу	Неопределенно долгое ожидание или фактическое невыполнение работы
Нарушения видимости памяти	Изменения не становятся доступными другим потокам в ожидаемом порядке	Использование устаревших данных, рассогласование состояния
Ошибки синхронизации	Некорректное использование примитивов координации	Потерянные сигналы, ложные пробуждения, пропуск событий
Нарушения атомарности	Логически единая операция выполняется как набор независимых шагов	Разрушение инвариантов составного состояния
Порядковые нарушения	События происходят в допустимой технически, но недопустимой логически последовательности	Преждевременное чтение, обращение к неинициализированному состоянию

Таблица 1 (продолжение)
Table 1 (continued)

Дефекты жизненного цикла	Некорректное создание, завершение, отмена или повторное использование потоков и задач	Утечки потоков, зависшие задачи, потеря исключений
Конкурентные ресурсные дефекты	Некорректное совместное использование буферов, очередей, файлов, соединений	Потеря данных, двойное освобождение, исчерпание ресурсов

Наиболее типичным конкурентным дефектом является состояние гонки. Его сущность состоит в том, что два или более потока обращаются к одному и тому же разделяемому состоянию без достаточного согласования порядка, причем хотя бы одно обращение изменяет это состояние. В результате итог работы программы становится зависимым от варианта interleaving. Гонки проявляются как потеря обновления, чтение промежуточного состояния, разрушение связанного инварианта структуры данных [5].

Не менее существенны взаимные блокировки. В этом случае несколько потоков переходят в состояние циклического ожидания и утрачивают возможность дальнейшего продвижения. Формально данные при этом могут оставаться корректными, однако система теряет способность к выполнению полезной работы. Близким по последствиям, но иным по механизму является livelock: потоки продолжают выполнять действия, однако не продвигают программу к завершению. Еще одна разновидность нарушения прогресса – голодание, при котором отдельный поток систематически оттесняется от доступа к ресурсу [6].

Особую группу образуют дефекты, связанные с моделью памяти. Даже если в программе отсутствует явная гонка за одну переменную, ошибка может возникнуть из-за неверной публикации нескольких взаимосвязанных изменений. Поток способен увидеть флаг готовности до того, как станут доступны все данные, которым этот флаг должен соответствовать. Подобные нарушения особенно опасны, поскольку зависят от архитектуры процессора, компилятора, уровня оптимизации и конкретных условий исполнения [7].

Значимы и ошибки синхронизации. Они включают потерянные сигналы, ложные пробуждения, неправильную логику использования условных переменных, ошибки в инициализации барьеров, несогласованную работу с флагами состояния. Такие нарушения показывают, что проверке подлежит не только защита разделяемых данных, но и сама логика координации между потоками [8].

Атомарностные и порядковые дефекты занимают промежуточное положение между ошибками доступа и ошибками координации. В первом случае логически единая операция разбивается на независимые шаги, между которыми вмешивается другой поток. Во втором случае события происходят в формально допустимой для среды, но недопустимой для логики приложения последовательности. Оба класса нарушений особенно характерны для высоконагруженных и асинхронных систем, где причинно-следственные зависимости между событиями выражены слабо.

Наконец, нельзя игнорировать дефекты жизненного цикла потоков и задач, а также конкурентные ресурсные нарушения. В современных системах проблемы возникают не только на уровне чтения и записи переменных, но и на уровне управления пулами потоков, фоновой активностью, задачами, очередями, дескрипторами и буферами. Эти нарушения часто долго остаются скрытыми и проявляются только под нагрузкой или в длительном режиме эксплуатации [9].

Формальная модель дефекта и окно конкуренции. Определение 1. Дефект многопоточной программы описывается тройкой $D = \langle C, \prec, E \rangle$, где $C = \{o^1, \ldots, o^m\}$ – множество конфликтующих операций над общим состоянием, $\prec$ – требуемое отношение порядка между ними, соблюдение которого необходимо для корректности, а E – наблюдаемый эффект нарушения этого порядка: некорректное состояние, ошибочный результат либо потеря прогресса. Класс дефекта определяется структурой отношения $\prec$: его отсутствие дает состояния гонки; наличие цикла ожиданий – взаимные блокировки; сохранение активности при отсутствии продвижения – livelock и голодание; неполнота или несогласованность порядка наблюдаемости – нарушения видимости памяти и публикации состояния; нарушение целостности составной операции – атомарностные дефекты; нарушение порядка операций жизненного цикла – дефекты создания, завершения и отмены потоков и задач. Именно структура отношения порядка, а не внешний симптом, служит основанием деления в Таблице 1, благодаря чему все выделенные классы описываются в единой системе понятий.

Определение 2. Окном конкуренции дефекта назовем число квантов планирования w, которые могут быть предоставлены другим потокам внутри интервала между конфликтующей операцией и операцией, восстанавливающей требуемое отношение порядка. Для гонки с потерей обновления это интервал между чтением и записью разделяемого значения; для взаимной блокировки – интервал между захватом удерживаемого ресурса и попыткой захвата следующего; для нарушения публикации состояния – интервал между записью данных и записью признака готовности. Величина w характеризует ширину критического интервала и непосредственно определяет вероятность того, что вытеснение потока попадет внутрь него.

Определение 3. Профилем наблюдаемости дефекта данного класса назовем минимальное множество видов фиксируемых событий и отношений между ними, регистрация которых необходима и достаточна для достоверного отнесения наблюдаемого нарушения к этому классу. Профиль наблюдаемости задает требования к трассировке конкурентного исполнения и, как показано ниже, позволяет сформулировать измеримые критерии полноты тестирования.

Утверждение 1. Если расписания порождаются независимо и по одному и тому же вероятностному закону, а дефект проявляется в одном прогоне с вероятностью q, то вероятность его обнаружения не менее чем за n прогонов равна $P(n) = 1 – (1 – q)^n$, а число прогонов, необходимое для обнаружения с достоверностью α, удовлетворяет неравенству $n \geq \ln(1 – \alpha) / \ln(1 – q)$. Доказательство. События проявления дефекта в разных прогонах независимы, поэтому вероятность того, что дефект не проявится ни в одном из n прогонов, равна $(1 – q)^n$; требуемое неравенство получается логарифмированием.

Следствие 1. При $q \to 0$ требуемое число прогонов неограниченно возрастает: дефект, не проявляющийся при данном режиме планирования, не может быть обнаружен никаким конечным увеличением числа прогонов. В частности, при невытесняющем планировании ($\pi = 0$) критический интервал не прерывается, поэтому $q = 0$ независимо от ширины окна конкуренции, и дефект остается полностью ненаблюдаемым. Отсюда следует, что критерий качества тестирования многопоточных программ не может выражаться только количеством прогонов: он должен характеризовать множество порождаемых расписаний. Это положение проверено экспериментально.

Профили наблюдаемости как критерий полноты тестирования. Из определений 1–3 следует, что каждому классу дефектов соответствует собственный профиль наблюдаемости, а полнота тестирования может оцениваться как покрытие не только входных данных, но и критических интервалов конкурентного исполнения. В Таблице 2

классы дефектов сопоставлены с критическими интервалами, минимальными наборами фиксируемых событий и количественными критериями полноты.

Таблица 2 – Профили наблюдаемости основных классов дефектов многопоточных программ
Table 2 – Observability profiles of the main classes of multithreaded program defects

Класс дефекта	Критический интервал (окно конкуренции)	Минимальный профиль наблюдаемости	Количественный критерий полноты
Состояния гонки	Интервал между конфликтующими обращениями к общему состоянию	Обращения к разделяемой памяти с указанием потока и версии значения; порядок обращений	Доля покрытых пар конфликтующих обращений внутри критических интервалов
Взаимные блокировки	Интервал между захватом удерживаемого ресурса и запросом следующего	Захваты и освобождения ресурсов, блокировки, граф ожиданий	Доля воспроизведённых пересекающихся частичных порядков захвата
Livelock	Цикл повторяющихся состояний без продвижения к цели	Повторяющиеся переходы, отсутствие достижения целевых состояний	Обнаружение в трассе цикла без прогресса
Голодание	Интервал между последовательными предоставлениями ресурса потоку	Длительности ожидания по потокам, распределение захватов	Превышение порога ожидания при прогрессивном исполнении системы
Нарушения видимости памяти	Интервал между записью и её глобальной наблюдаемостью	Времена наблюдаемости записей, отношения «чтение — запись»	Доля проверенных требований публикации (release/acquire)
Ошибки синхронизации	Интервал между изменением условия и его проверкой или пробуждением	Операции над примитивами синхронизации, состояния ожидания, потерянные сигналы	Покрытие предикатов ожидания и ветвей проверки условия
Нарушения атомарности	Интервал внутри составной операции	Границы составной операции, вмешавшиеся посторонние обращения	Отсутствие посторонних операций над объектом в каждом интервале атомарности
Порядковые нарушения	Интервал между зависимыми событиями разных потоков	Причинно-следственные связи (happens-before), точки инициализации и использования	Доля проверенных пар, связанных требуемым отношением порядка
Дефекты жизненного цикла	Интервалы создания, завершения и отмены потоков и задач	Операции жизненного цикла, владение задачами, потеря исключений	Покрытие переходов жизненного цикла и режимов отказа
Конкурентные ресурсные дефекты	Интервал владения буфером, соединением, дескриптором	Выделение и освобождение ресурсов, исчерпание пулов	Покрытие пар «выделение – освобождение» и сценариев исчерпания

Таблица 2 показывает, что требования к тестированию, сформулированные ниже как качественные принципы, допускают операционализацию: каждому требованию соответствует измеримый критерий – доля покрытых критических интервалов, доля воспроизведенных частичных порядков захвата ресурсов, отсутствие в трассе циклов без прогресса, доля проверенных публикаций состояния, полнота покрытия переходов жизненного цикла. Тем самым классификация дефектов становится не только средством описания, но и основанием для построения метрик тестового покрытия конкурентного исполнения.

Имитационная проверка гипотезы об обнаруживаемости дефектов. Проверка гипотезы выполнялась на описанной в разделе «Материалы и методы» абстрактной модели планирования. Для дефекта класса «потеря обновления» каждый поток однократно выполняет критическую секцию вида «чтение разделяемого значения – вычисление – запись результата», причем между чтением и записью размещается w квантов, которые могут быть предоставлены другим потокам; дефект фиксируется, если итоговое значение счетчика оказывается меньше числа потоков. Для дефекта класса «взаимная блокировка» каждый поток последовательно захватывает два ресурса, причем половина потоков использует обратный порядок захвата; дефект фиксируется, если к моменту остановки исполнения хотя бы один поток остается заблокированным. Рассчитанные значения коэффициента проявления q приведены в Таблице 3.

Таблица 3 – Коэффициент проявления q при различных параметрах планирования (гонка с потерей обновления, k = 4)
Table 3 – Manifestation coefficient q under different scheduling parameters (lost update, k = 4)

w \ π	**0,00**	**0,01**	**0,05**	**0,10**	**0,25**	**0,50**	**1,00**
w = 0	0,000	0,022	0,094	0,182	0,409	0,691	0,958
w = 1	0,000	0,036	0,176	0,324	0,645	0,900	0,999
w = 2	0,000	0,057	0,253	0,444	0,796	0,971	1,000
w = 4	0,000	0,088	0,388	0,636	0,928	0,997	1,000
w = 8	0,000	0,160	0,587	0,835	0,992	1,000	1,000
w = 16	0,000	0,273	0,810	0,969	1,000	1,000	1,000

Данные Таблицы 3 подтверждают второе положение гипотезы и обнаруживают существенную неоднородность режимов планирования. При невытесняющем планировании ($\pi = 0$) коэффициент проявления равен нулю при всех рассмотренных значениях w: критический интервал не прерывается, и дефект не проявился ни в одном из 12 000 прогонов. Уже при малой интенсивности вытеснений вероятность проявления становится отличной от нуля и возрастает как с ростом π, так и с расширением окна конкуренции w: чем шире критический интервал, тем выше вероятность того, что вытеснение попадет внутрь него. Так, при $w = 2$ значение q растет от 0,057 при $\pi = 0,01$ до 1,000 при $\pi = 1$, тогда как при $w = 16$ и $\pi = 0,05$ оно составляет уже 0,810. При интенсивном вытесняющем планировании, моделирующем идеализированное случайное планирование, значение q приближается к единице, то есть дефект обнаруживается практически при первом прогоне.

Таблица 4 переводит полученные значения q в требуемый объем тестирования. Различие между режимами планирования оказывается принципиальным: там, где при интенсивном вытеснении достаточно одного-двух прогонов, при пониженной интенсивности требуются десятки и сотни прогонов, а при невытесняющем планировании задача обнаружения дефекта вообще не имеет конечного решения (в Таблице 4 такие конфигурации отмечены прочерком). Зависимость вероятности обнаружения от числа прогонов, представленная на Рисунке 1, показывает, что

эмпирические данные согласуются с теоретическим законом $P(n) = 1 - (1 - q)^n$ при соответствующем значении q: расхождения не выходят за пределы статистической погрешности имитационного эксперимента.

Таблица 4 – Требуемое число прогонов для обнаружения дефекта с достоверностью α = 0,99
Table 4 – Required number of runs to detect the defect with confidence α = 0.99

Конфигурация имитационной модели	Коэффициент проявления q	n при α = 0,95	n при α = 0,99
Гонка с потерей обновления, w = 2, π = 0,01	0,057	51	78
Гонка с потерей обновления, w = 2, π = 0,10	0,444	6	8
Гонка с потерей обновления, w = 2, π = 1,00	1,000	1	1
Гонка с потерей обновления, w = 16, π = 0,05	0,810	2	3
Взаимная блокировка, k = 4, π = 0,05	0,035	84	129
Взаимная блокировка, k = 4, π = 1,00	0,672	3	5
Невытесняющее планирование, π = 0	0,000	–	–

Для дефекта класса «взаимная блокировка» зависимость обнаруживаемости от режима планирования выражена иначе: при невытесняющем планировании цикл ожиданий не образуется (q = 0,000), тогда как при π = 0,05 значение q составляет 0,035, а при π = 1,00 – 0,672. Требуемое для его обнаружения число прогонов в несколько раз больше соответствующим значениям для гонки с потерей обновления при той же интенсивности вытеснений, что показывает зависимость требований к тестированию не только от режима планирования, но и от класса дефекта.

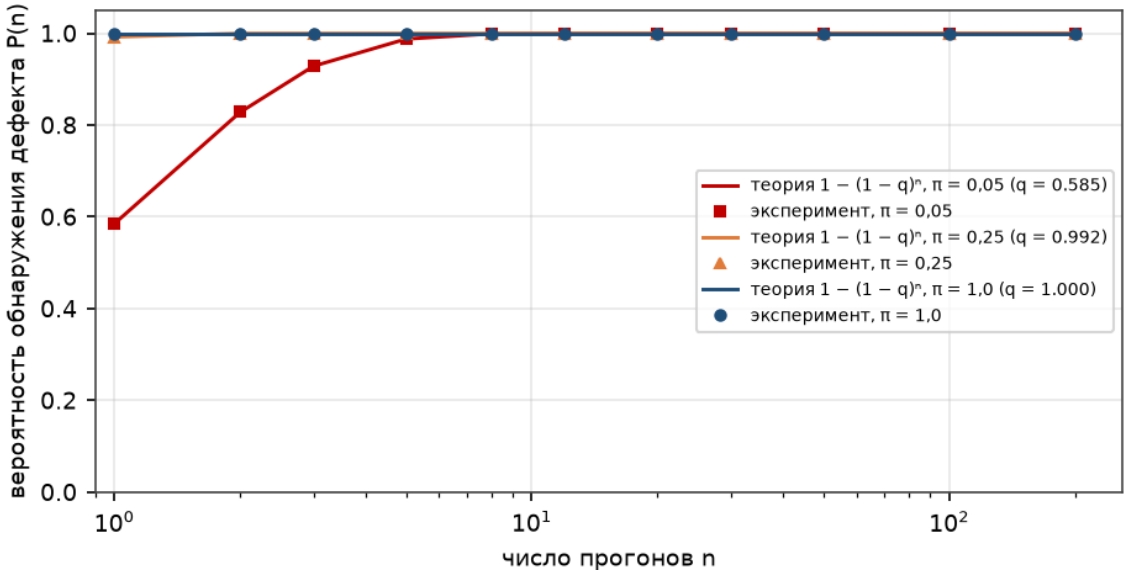


Рисунок 1 – Зависимость вероятности обнаружения дефекта от числа прогонов
Figure 1 – Detection probability versus the number of runs

Рисунок 2 иллюстрирует два принципиальных обстоятельства: при низкой интенсивности вытеснений обнаруживаемость дефекта определяется шириной окна конкуренции (Рисунок 2*а*), а требуемое число прогонов изменяется на несколько

порядков при переходе от интенсивного вытесняющего планирования к слабо вытесняющему (Рисунок 2*б*).

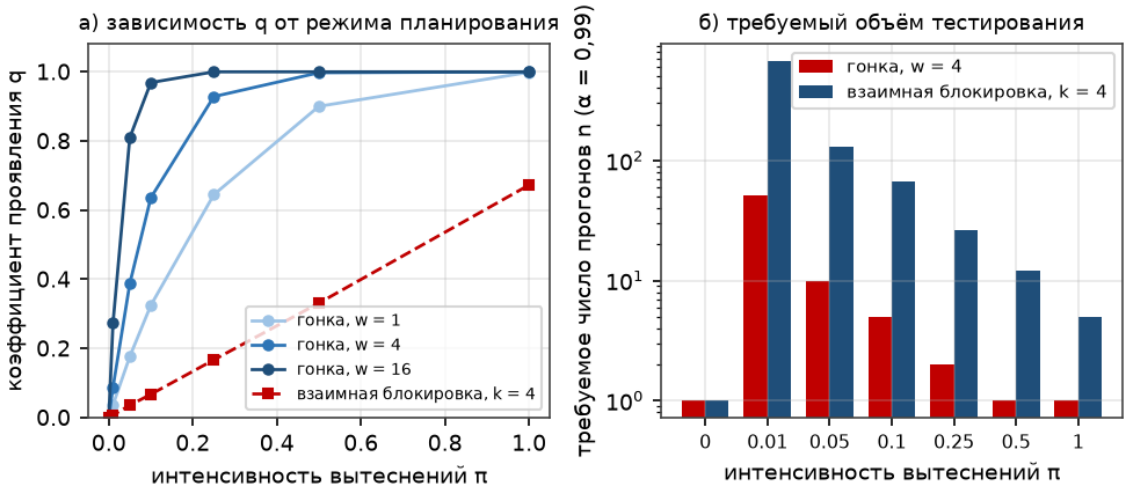


Рисунок 2 – Влияние режима планирования на обнаруживаемость дефектов
Figure 2 – Effect of the scheduling regime on defect detectability

Таким образом, эксперимент подтверждает оба положения гипотезы: обнаруживаемость дефекта описывается вероятностной зависимостью, а ее определяющим параметром выступает режим планирования, а не число прогонов. Практический вывод состоит в том, что критерием качества тестирования многопоточной программы должна служить не длительность стресс-тестирования, а свойства порождаемого множества расписаний: охват критических интервалов и обеспечение их прерывания в контролируемых точках. Увеличение числа прогонов не компенсирует недостаточную интенсивность вытеснений, поскольку при $q = 0$ величина требуемого n неограниченно возрастает.

Требования к тестированию многопоточных программ. Систематизация дефектов позволяет сформулировать требования к тестированию многопоточных программ. Первое из них состоит в том, что тестирование не может ограничиваться только входными данными. Один и тот же набор входов в многопоточной системе порождает множество различных траекторий исполнения. Следовательно, объектом исследования должно становиться пространство расписаний, а не единичный запуск программы.

Второе требование связано с необходимостью анализа конфликтов доступа и межпоточных зависимостей. Простое многократное выполнение тестов не гарантирует попадания в критический interleaving. Поэтому тестирование должно быть ориентировано на выделение потенциально опасных участков исполнения: конфликтующих обращений к общему состоянию, точек синхронизации, критических фаз публикации данных и сценариев захвата ресурсов [9].

Третье требование – воспроизводимость. Для практики разработки мало обнаружить факт сбоя; необходимо обеспечить возможность его повторного вызова и последующей локализации причины. В многопоточной среде это особенно трудно, поскольку дефект часто связан с редким сочетанием событий. Следовательно, эффективное тестирование должно либо сохранять существенные параметры исполнения, либо включать управляемое вмешательство в планирование, либо фиксировать трассы событий и причинно-следственные связи.

Четвертым требованием является наблюдаемость межпоточных взаимодействий. Если анализ ограничивается только конечным результатом, значительная часть

нарушений остается незамеченной. Требуется фиксировать обращения к разделяемым данным, операции синхронизации, длительность ожиданий, переходы потоков между состояниями, конфликты за ресурсы и точки публикации состояния. Для многопоточной среды наблюдаемость выступает необходимым условием содержательного анализа [9].

Пятое требование связано с учетом модели памяти. Методы тестирования, которые ориентированы только на блокировки и критические секции, нечувствительны к нарушениям видимости и публикации данных. Между тем именно этот класс ошибок во многих случаях оказывается наиболее скрытым и опасным. Поэтому тестирование должно быть способно работать не только с явными data race, но и с нарушениями порядка наблюдаемости операций.

Шестое требование – контроль прогресса. Корректная многопоточная программа должна не только сохранять допустимое состояние, но и обеспечивать продвижение потоков к завершению полезной работы. Следовательно, тестирование должно включать механизмы обнаружения взаимных блокировок, livelock, голодания, неограниченного роста очередей ожидания и других форм потери прогресса.

Седьмое требование состоит в необходимости сочетать статические и динамические методы анализа. Статический анализ позволяет выделять потенциально опасные участки, не дожидаясь проявления сбоя. Динамическое тестирование, в свою очередь, показывает фактическое поведение программы в реальном исполнении. Только совместное использование этих подходов дает возможность одновременно обнаруживать потенциальные дефекты и проверять, как они реализуются в конкретных сценариях.

Восьмое требование относится к масштабируемости и накладным расходам. Простые методы, пригодные для малых примеров, часто теряют практический смысл на программах промышленного масштаба. Поэтому качественное тестирование многопоточных программ должно обеспечивать приемлемое соотношение между полнотой анализа, точностью результатов и стоимостью применения.

Девятое требование – управляемость порождения расписаний. Как показано в разделе об имитационной проверке гипотезы, обнаруживаемость дефекта определяется не числом прогонов, а тем, прерывается ли критический интервал исполнения и с какой интенсивностью. Поэтому тестирование должно включать механизм управляемого вмешательства в планирование: принудительные вытеснения в точках конфликтующих обращений, воспроизведение заданных частичных порядков, варьирование интенсивности переключений. Только при этом условии число прогонов становится содержательной величиной, поскольку каждый прогон исследует новое расписание, а не воспроизводит одно и то же.

Обсуждение

Полученные результаты позволяют рассматривать классификацию дефектов многопоточных программ не просто как удобный способ их описания, а как необходимое теоретическое основание для построения более точных и практически применимых методов тестирования. В большинстве прикладных работ, посвященных многопоточности, акцент делается либо на отдельных классах нарушений, прежде всего на состояниях гонки и взаимных блокировках, либо на конкретных инструментах анализа, ориентированных на фиксированный набор шаблонов ошибок. Такой подход дал значимые результаты в развитии средств контроля конкурентного исполнения, однако он не устранил фундаментальную проблему: многопоточный дефект по своей природе редко сводится к одной технической причине. На практике ошибка почти всегда возникает на пересечении нескольких факторов – структуры доступа к общему

состоянию, логики координации, правил публикации данных, особенностей модели памяти и конкретного варианта interleaving. Именно поэтому классификация, построенная только по внешнему симптому, оказывается недостаточной.

Принципиально важным представляется то обстоятельство, что дефекты многопоточных программ обладают иной эпистемологической природой по сравнению с ошибками последовательного кода. В последовательной программе можно достаточно уверенно соотнести ошибку с конкретным участком алгоритма, условием ветвления, некорректной формулой либо неверной обработкой исключения. В многопоточной системе это соотношение становится опосредованным. Нарушение возникает в структуре взаимодействия, а наблюдаемый отказ нередко фиксируется на значительном удалении от той точки исполнения, где была заложена его причина. Отсюда следует, что тестирование конкурентных систем не может ограничиваться регистрацией финального сбоя. Оно должно восстанавливать динамику исполнения, то есть отвечать не только на вопрос о том, произошла ли ошибка, но и на вопрос, каким образом система пришла к ошибочному состоянию. В этом отношении предложенная классификация ценна именно тем, что она связывает симптом с механизмом возникновения нарушения.

Анализ показал, что наиболее устойчивой ошибкой практики остается тенденция редуцировать проблему многопоточности к состояниям гонки. Действительно, гонки по данным являются одним из самых частых и заметных классов дефектов. Они хорошо поддаются формальному определению, удобно иллюстрируются и имеют значительную прикладную важность. Однако сосредоточение только на них формирует суженное представление о конкурентной корректности. Между тем взаимные блокировки, livelock, голодание, ошибки публикации состояния, порядковые нарушения и дефекты жизненного цикла потоков не менее значимы, а в реальных промышленных системах зачастую оказываются даже более затратными по последствиям. Они хуже воспроизводятся, труднее локализуются и нередко не фиксируются простыми средствами наблюдения. Следовательно, оценка качества многопоточной программы должна строиться на более широком понимании конкурентного дефекта.

С научной точки зрения особого внимания заслуживает группа нарушений, связанных не с прямым конфликтом за ресурс, а с несоответствием между предполагаемой и фактической семантикой исполнения. Именно в этой зоне находятся дефекты видимости памяти, нарушения публикации данных и значительная часть порядковых ошибок. Для разработчика подобные нарушения особенно коварны, поскольку они маскируются под корректную последовательную логику. Код может быть написан аккуратно, критические участки могут быть выделены, а результат все равно будет зависеть от архитектуры платформы, поведения компилятора, реального порядка наблюдаемости изменений и конкретного сценария исполнения. Это позволяет сделать важный вывод: многопоточное тестирование должно работать не только с разделяемыми данными как таковыми, но и с семантической стороной конкурентного исполнения, то есть с тем, как состояния становятся доступными другим потокам и какие предположения о порядке действий реально поддерживаются системой[1].

Отсюда вытекает и более широкий методологический вывод. При анализе конкурентных программ объектом исследования должна выступать не только программа как текст, но и программа как исполняемая событийная система. Такой переход от статического представления к динамическому особенно важен для тестирования. В традиционном функциональном подходе тестовые данные выбираются так, чтобы

[1] OpenMP Architecture Review Board. *OpenMP Application Programming Interface. Version 6.0 November 2024.* OpenMP. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf (дата обращения: 15.05.2026).

пройти определенные ветви алгоритма и сравнить полученный результат с ожидаемым. Для многопоточной среды этого недостаточно. Здесь необходимо исследовать не только пространство входов, но и пространство расписаний. Иначе говоря, один и тот же тест должен рассматриваться как множество возможных сценариев, зависящих от порядка событий, точек синхронизации, длительности ожиданий и конфликтов доступа. В этом контексте предложенная классификация дефектов представляет ценность именно потому, что она указывает, какие группы событий и зависимостей следует считать потенциально опасными.

Практический смысл такого подхода заключается в изменении самого принципа построения тестов. Если дефекты многопоточных программ возникают в структуре взаимодействия, то и тестирование должно строиться как управляемое исследование этого взаимодействия. Бессистемное многократное выполнение программы под нагрузкой может выявить часть грубых нарушений, но не обеспечивает ни полноты, ни объяснимости результата. *Наоборот, оно часто создает иллюзию надежности: если ошибка не проявилась при серии запусков, программа воспринимается как корректная, хотя критическое interleaving просто не было реализовано. Следовательно, более перспективным является такой подход, при котором тестирование ориентируется на конфликты доступа, сценарии захвата ресурсов, точки публикации состояния, причинно-следственные зависимости и критические фазы жизненного цикла потоков. Именно эта логика и связывает классификацию дефектов с требованиями к тестированию.*

Важным результатом проведенного анализа является подтверждение того, что воспроизводимость должна рассматриваться не как вторичная характеристика метода, а как один из центральных критериев его качества. Для многопоточной программы обнаружение дефекта без возможности повторного восстановления сценария его возникновения имеет ограниченную ценность. Разработчик получает симптом, но не получает надежного основания для исправления и последующей верификации результата. Особенно это касается нарушений, связанных с редкими interleaving, ошибками видимости и сложными зависимостями между событиями. Поэтому требования к тестированию, сформулированные на основе предложенной классификации, закономерно включают необходимость накопления трасс, фиксации структуры исполнения и, по возможности, управляемого вмешательства в расписание. Без этого даже точное обнаружение дефекта остается неполным.

Отдельного обсуждения заслуживает вопрос о соотношении статических и динамических методов. Результаты анализа показывают, что противопоставление этих подходов в контексте многопоточных программ является методологически непродуктивным. Статический анализ эффективен как средство раннего выявления потенциально опасных участков: конфликтных обращений, пересекающихся путей захвата ресурсов, подозрительных шаблонов синхронизации, неполных отношений порядка. Однако он неизбежно сталкивается с проблемой избыточности предупреждений и ограниченной способности различать реальные и мнимые угрозы. Динамическое тестирование, напротив, фиксирует фактическое поведение программы, но зависит от того, удалось ли воспроизвести опасный сценарий исполнения. В этом отношении проведенная классификация косвенно подтверждает необходимость их интеграции: разные типы дефектов требуют разной комбинации статических и динамических средств. Гонки и ресурсные нарушения требуют одного баланса методов, дефекты прогресса – другого, нарушения видимости и публикации состояния – третьего.

Не менее существенным является и вывод о том, что многопоточные дефекты следует различать по степени их эксплуатационной критичности. Не каждое

конкурентное нарушение немедленно приводит к аварийному завершению программы. Некоторые ошибки долго остаются латентными, накапливаются и лишь впоследствии проявляются как резкая деградация производительности, исчерпание ресурсов, нарушение логической целостности или потеря управляемости системой. В промышленных условиях такие дефекты часто оказываются даже опаснее грубых отказов, поскольку они дольше скрываются и сложнее диагностируются. По этой причине в обсуждаемой классификации оправдано выделение не только собственно дефектов корректности, но и производительных аномалий конкурентного происхождения[2]. Такой подход расширяет представление о целях тестирования: оно должно обеспечивать не только логическую правильность работы, но и устойчивость поведения системы под реальной нагрузкой.

Полученные результаты следует соотнести с известными работами, задающими общий контекст исследования. В классических работах по конкурентному программированию сложность многопоточности рассматривается как принципиальное свойство модели исполнения: отмечается, что потоки как средство организации вычислений плохо соответствуют интуитивным представлениям разработчика о порядке действий, а распространение многоядерных систем превращает конкурентные дефекты в массовое явление [10, 11]. Эти работы убедительно обосновывают необходимость специальных средств контроля конкурентного исполнения, однако остаются на качественном уровне описания и не вводят характеристик, позволяющих сопоставить классу дефектов ожидаемый объем тестирования. Настоящая работа дополняет эту линию исследования количественной стороной, показывая, что обнаруживаемость дефекта является измеримой величиной, зависящей от структуры дефекта и режима планирования.

Существенно и то, что предложенная модель объясняет известный практический парадокс: многократный запуск программы под нагрузкой нередко не выявляет дефект, который затем проявляется в эксплуатации. Согласно полученным результатам, причина заключается не в редкости дефекта как такового, а в низкой интенсивности вытеснений внутри критического интервала: расписания, порождаемые при повторных запусках, оказываются статистически близкими, и последовательность прогонов не дает накопления информации о пространстве расписаний. Тем самым подтверждается сформулированное требование об управляемом вмешательстве в планирование и о фиксации трасс: без разнообразия расписаний число прогонов утрачивает диагностический смысл.

Следует отметить и ограничения проведенного исследования. Во-первых, предложенная классификация носит теоретико-методологический характер и не претендует на исчерпывающее описание всех частных форм нарушений, встречающихся в конкретных языках, библиотеках или исполнительных средах. На уровне практической реализации каждый выделенный класс распадается на множество вариантов, зависящих от модели памяти, особенностей runtime-среды, способа организации асинхронности и архитектуры приложения. Во-вторых, в статье не ставилась задача построить количественную модель сравнительной опасности различных типов дефектов; акцент был сделан на их систематизации и на выводе требований к тестированию. В-третьих, ряд современных высокоуровневых моделей конкурентности, включая реактивные конвейеры, actor-based архитектуры и гибридные событийно-потоковые среды, здесь рассматриваются только в той мере, в какой они воспроизводят общие закономерности межпоточного взаимодействия.[12]

[2] При подготовке выделенных курсивом фрагментов текста использовался генеративный инструмент ChatGPT. Модель GPT-5.5. URL: https://chatgpt.com/

Отдельно следует оговорить ограничения выполненной имитационной проверки. Использованная модель является абстрактной: она воспроизводит структуру планирования и конфликтов доступа, но не моделирует конкретную систему команд, особенности модели памяти аппаратной платформы, поведение оптимизирующего компилятора и политики реальных диспетчеров операционной системы. Поэтому полученные числовые значения q и n имеют иллюстративный характер и не должны переноситься на конкретные программные системы без дополнительной калибровки. Содержательное значение имеет не абсолютная величина этих чисел, а установленная зависимость: обнаруживаемость дефекта определяется режимом планирования и структурой дефекта, а при отсутствии вытеснений внутри критического интервала обнаружение принципиально недостижимо за конечное число прогонов.

Вместе с тем именно эти ограничения намечают перспективные направления дальнейшей работы. Представляется целесообразным перейти от общей классификации к построению формализованной карты соответствия между типами дефектов, свойствами исполнения и средствами их тестирования. Практически это означает необходимость ответить на вопрос, какие именно наблюдаемые события, трассы, метрики и процедуры управления расписанием являются наиболее информативными для каждого класса нарушений. Кроме того, перспективным направлением является разработка критериев приоритизации дефектов с учетом их эксплуатационной критичности, воспроизводимости и стоимости исправления. Для промышленных систем такая приоритизация особенно важна, поскольку ресурсы тестирования всегда ограничены, а полное исследование пространства interleaving для сложных программ недостижимо[3].

Заключение

Проведенное исследование позволяет сделать следующие выводы.

Во-первых, дефекты многопоточных программ образуют самостоятельный класс нарушений, который не может быть описан только в терминах обычных алгоритмических или ресурсных ошибок. Их природа определяется структурой конкурентного исполнения, порядком взаимодействия потоков, механизмами синхронизации и правилами публикации состояния.

Во-вторых, к основным типам дефектов многопоточных программ относятся состояния гонки, взаимные блокировки, livelock, голодание, нарушения видимости памяти, ошибки синхронизации, атомарностные и порядковые нарушения, дефекты жизненного цикла потоков и конкурентные ресурсные нарушения. Каждая из этих групп имеет собственный механизм возникновения и требует специального подхода к диагностике.

В-третьих, систематизация дефектов позволяет сформулировать требования к тестированию многопоточных программ. К их числу относятся учет пространства расписаний, воспроизводимость нарушений, наблюдаемость межпоточных взаимодействий, контроль прогресса, учет модели памяти, сочетание статических и динамических методов анализа, а также приемлемые накладные расходы.

В-четвертых, сформулирована и проверена научная гипотеза о вероятностной природе обнаруживаемости конкурентных дефектов. Показано, что вероятность обнаружения дефекта за n прогонов описывается зависимостью $P(n) = 1 – (1 – q)^n$, а коэффициент проявления q определяется шириной окна конкуренции, числом потоков и интенсивностью вытеснений. Имитационный эксперимент подтвердил, что при

[3] Gosling J., Joy B., Steele G., et al. Chapter 17: Threads and Locks. In: *The Java® Language Specification. Java SE 17 Edition*. Oracle America; 2021. P. 737–764.

невытесняющем планировании дефект не обнаруживается ни при каком числе прогонов, а при интенсивном вытесняющем планировании он выявляется практически с первого прогона.

В-пятых, научная новизна работы состоит в переходе от обзорно-обобщающего описания дефектов к их формальному представлению: классификация построена на едином инварианте – структуре отношения порядка между конфликтующими операциями; введены количественные характеристики дефекта (окно конкуренции и коэффициент проявления); предложен профиль наблюдаемости как критерий полноты тестирования; получены эмпирические зависимости обнаруживаемости дефектов от режима планирования. Перспективным направлением дальнейшей работы является распространение предложенной модели на дефекты, связанные с моделью памяти, и построение на ее основе процедур управляемого порождения расписаний для программных систем промышленного масштаба.

СПИСОК ИСТОЧНИКОВ / REFERENCES

1. Herlihy M., Shavit N., Luchangco V., Spear M. *The Art of Multiprocessor Programming.* Cambridge: Morgan Kaufmann; 2020. 576 p.
2. Pacheco P., Malensek M. *An Introduction to Parallel Programming.* Cambridge: Morgan Kaufmann; 2021. 496 p.
3. Кулямин В.В. Обзор методов динамического анализа программного обеспечения. *Труды Института системного программирования РАН.* 2023;35(4):7–44. https://doi.org/10.15514/ISPRAS-2023-35(4)-1
   Kuliamin V.V. Survey of Software Dynamic Analysis Methods. *Proceedings of the Institute for System Programming of the RAS.* 2023;35(4):7–44. (In Russ.). https://doi.org/10.15514/ISPRAS-2023-35(4)-1
4. Рагозина П.И., Игнатьев В.Н. Обнаружение ошибок взаимоисключающей блокировки в программах на языке C# при помощи методов статического анализа. *Труды Института системного программирования РАН.* 2022;34(4):63–78. https://doi.org/10.15514/ISPRAS-2022-34(4)-5
   Ragozina P.I., Ignatyev V.N. Detection of erroneous usage of synchronization monitor in C# via static analysis. *Proceedings of the Institute for System Programming of the RAS.* 2022;34(4):63–78. (In Russ.). https://doi.org/10.15514/ISPRAS-2022-34(4)-5
5. Доронин О.В., Дергун К.И., Дергачев А.М., Ильина А.Г., Горлач С.П. Тестирование многопоточных приложений с блокировками на неатомарных переменных. *Научно-технический вестник информационных технологий, механики и оптики.* 2020;20(2):243–248. https://doi.org/10.17586/2226-1494-2020-20-2-243-248
   Doronin O.V., Dergun K.I., Dergachev A.M., Ilina A.G., Gorlatch S.P. Testing of multithreaded applications with locks on non-atomic variables. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics.* 2020;20(2):243–248. (In Russ.). https://doi.org/10.17586/2226-1494-2020-20-2-243-248
6. Герлиц Е.А., Мутилин В.С. Фреймворк автоматизации тестирования на гонки по данным. *Труды Института системного программирования РАН.* 2025;37(1):107–120.
   Gerlits E.A., Mutilin V.S. Towards a Test Automation Framework for Data Race Testing. *Proceedings of the Institute for System Programming of the RAS.* 2025;37(1):107–120. (In Russ.).

7. Руденчик В.П., Андрианов П.С., Мутилин В.С. Ограничение количества переключений потоков при динамическом анализе многопоточных программ. *Труды Института системного программирования РАН.* 2025;37(6):133–148. Rudenchik V.P., Andrianov P.S., Mutilin V.S. Bounding Thread Switches in Dynamic Analysis of Multithreaded Programs. *Proceedings of the Institute for System Programming of the RAS.* 2025;37(6):133–148. (In Russ.).
8. Lu Sh., Park S., Seo E., Zhou Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII), 01–05 March 2008, Seattle, WA, USA*. New York: ACM; 2008. P. 329–339. https://doi.org/10.1145/1346281.1346323
9. Musuvathi M., Qadeer Sh., Ball Th., Basler G., Nainar P.A., Neamtiu I. Finding and Reproducing Heisenbugs in Concurrent Programs. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), 08–10 December 2008, San Diego, CA, USA*. USENIX Association; 2008. P. 267–280.
10. Lee E.A. The problem with threads. *Computer.* 2006;39(5):33–42. https://doi.org/10.1109/MC.2006.180
11. Sutter H., Larus J. Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry. *Queue.* 2005;3(7):54–62. https://doi.org/10.1145/1095408.1095421
12. Kleppmann M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* Sebastopol: O'Reilly Media; 2017. 616 p.

ИНФОРМАЦИЯ ОБ АВТОРЕ / INFORMATION ABOUT THE AUTHOR

Шульга Дмитрий Константинович, аспирант, Санкт-Петербургский политехнический университет Петра Великого, Санкт-Петербург, Российская Федерация.

Dmitrii K. Shulga, Postgraduate, Peter the Great St. Petersburg Polytechnic University, St. Petersburg, the Russian Federation.

Статья поступила в редакцию 08.07.2026; одобрена после рецензирования 02.11.2026; принята к публикации 18.11.2026.

The article was submitted 08.07.2026; approved after reviewing 02.11.2026; accepted for publication 18.11.2026.