ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Гаев Александр Анатольевич

АДАПТАЦИЯ LINCHECK ДЛЯ ТЕСТИРОВАНИЯ МНОГОПОТОЧНЫХ АЛГОРИТМОВ НА C/C++

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
Е.А. Моисеенко

Руководитель
канд.физ.-мат.наук, доц.
Д.Н. Москвин

Консультант
Н.Д. Коваль

Санкт-Петербург 2021

Оглавление

Процесс разработки многопоточных программ гораздо сложнее, чем последовательных. Есть множество различных причин для этого, будь то переупорядочивание операций компиляторами, либо ослабленные модели памяти. Всё это приводит к необходимости тестировать многопоточные программы ещё тщательнее. Для этих задач разрабатываются различные инструменты, но, к сожалению, таких инструментов не так много для C++, что подтверждается тем фактом, что существующие многопоточные библиотеки на C++ разрабатывают собственную инфраструктуру для тестирования. Для помощи таким библиотекам в тестировании в данной работе инструмент Lincheck был адаптирован для тестирования программ на C++. Был предоставлен интерфейс для использования Linchceck из C++, а также сгенерирована динамическая библиотека для подключения к существующим проектам. Работоспособность инструмента была протестирована на существующих популярных многопоточных библиотеках на C++.

Ключевые слова: многопоточность, тестирование, линеаризуемость.

The process of developing multithreaded programs is much more complex than developing sequential programs. There are many different reasons for this: reordering operations by compilers or weakened memory models. All this leads to the need to test multithreaded programs even more thoroughly. Various tools were developed for these tasks, but unfortunately, there are few such tools for C++, which is confirmed by the fact that existing multithreaded libraries in C++ develop their infrastructure for testing. The Lincheck tool has been adapted for testing programs in C++ to help such libraries in testing. An interface was provided to use Linchceck from C++, and a dynamic library was generated to connect to existing projects. The tool was applied to existing popular multithreaded libraries in C++.

Keywords: multithreading, testing, linearizability.

Введение

Многопоточные программы обычно полагаются на оптимизированные реализации многопоточных структур данных, таких как стеки, очереди и множества [21], чьи операции выполняются параллельно между ядрами процессора для максимальной эффективности [24]. Однако программировать эти «параллельные структуры данных» непросто. Синхронизация между операциями должна быть минимизирована, чтобы сократить время отклика и увеличить пропускную способность [32]. Тем не менее, этот минимальный объем синхронизации также должен быть достаточным для обеспечения того, чтобы операции вели себя так, как если бы они выполнялись атомарно, одна за другой, так, чтобы те, кто используют структуру данных, могли полагаться на свою последовательную спецификацию. Эти противоположные требования вместе с общей трудностью рассуждений о чередовании потоков, делают параллельные структуры данных большим источником ошибок [22]. Всё это ведёт к необходимости верифицировать многопоточные программы.

1. Формальное доказательство. Общепринятым способом доказательства корректности является предоставление формального доказательства, которое формируется с помощью строгих правил и проверяется другими людьми на ошибки. Такой способ, к сожалению, может содержать ошибки в доказательстве, и будет проверять лишь идею доказательства, никак не проверяя её реализацию на ошибки. Также существуют инструменты, такие как Coq [7], которые в комбинации с фреймворками как Iris [15] способны проверять формальное описание алгоритма, но не могут подтвердить корректность реализации.

2. Model checking. Суть model checking состоит в проверке всех возможных исполнений. Более формально она состоит в выделении всех возможных состояний программы, всех возможных переходов между состояниями, и проверке корректности всех возможных путей из начальных состояний на корректность. Для этого программу запускают в контролируемой среде исполнения, осуществляя многопоточную работу всеми возможными способами(исследуются все возможные переключения контекста потоков).

3. Тестирование. Тестирование программ является самым популярным и общепринятым способом проверить корректность написанного кода. Оно позволяет проверять именно тот код, который будет использоваться на практике. Оно позволяет найти баланс между количеством усилий, затраченных на проверку кода, и степенью уверенности в том, что код на самом деле корректный.

В рамках этой работы большее внимание будет уделено именно тестированию, поскольку оно гораздо чаще других способов используется на практике.

Актуальность работы

Язык C++ позволяет оперировать напрямую с потоками операционной системы, обращаться к специфичным системным вызовам, и использовать особенные инструкции процессора. Поэтому критичные по производительности вещи зачастую пишутся именно на нём. Многопоточность позволяет увеличить производительность критически важных частей, но повышает вероятность допустить ошибку в ней, поэтому вопрос тестирования кода и поиска в нём ошибок становится ещё актуальнее.

Тем не менее, необходимо каким-то образом определять критерий корректности кода. Тесты позволяют проверять определенные сценарии и инварианты, но в научном сообществе выделяют более удобные и научные критерии. Наиболее используемым является линеаризуемость [12]. Существуют также и другие критерии, такие как сериализуемость [4], последовательная согласованность [18], quasi-linearizability [1], quiescent consistency [28]. Линеаризуемость является локальным и неблокируемым свойством. Это позволяет проверять линеаризуемость определенной компоненты, и далее работать с ней как с черным ящиком, который гарантированно потокобезопасен и атомарен.

За последние десятилетия был проделан значительный объем работы над инструментами для проверки корректности многопоточных программ. Из-за ограниченного места я сосредоточусь только на тех инструментах, которые связаны с моей работой.

Для тестирования многопоточного кода было разработано множество различных инструментов для тестирования, в частности JCStress [16], Line-Up [20], и CONCURRIT [5]. JCStress работает только с языком java, не ориентируется на удобство, требует задавать полный сценарий исполнения и выписывать все возможные результаты, и изначально разрабатывался для тестирования низкоуровневых свойств реализаций модели памяти java. Line-Up является закрытой разработкой Microsoft Research, который перестал разрабатываться в 2011 году. Он работает только с языком C и недоступен для использования и анализа. CONCURRIT является немного более удобным языком для написания многопоточных тестов, и не умеет автоматически тестировать структуру данных на линеаризуемость, поэтому является неудобным.

Lincheck [30] — это фреймворк для тестирования многопоточных структур данных под JVM, предоставляющий возможность декларативного написания тестов. Поддерживает тестирование линеаризуемости и других критериев. Поскольку Lincheck написан на языке Kotlin и его идея не зависит от платформы, то с помощью технологии Kotlin Multiplatform можно получить кроссплатформенный инструмент для тестирования программ. Это позволит разрабатывать функционал один раз для всех платформ, поэтому именно Lincheck был выбран для адаптации для тестирования кода на C++.

Постановка задачи

Целью данной работы является адаптация Lincheck для тестирования многопоточных алгоритмов на C++.

Задачи:

1. Перевести Lincheck на систему сборки gradle

   Изначально Lincheck использует систему сборки maven, которая не подходит для адаптации, поскольку не поддерживает технологию Kotlin Multiplatform

2. Переписать Lincheck на Kotlin Multiplatform

   Для использования Lincheck из C++ необходимо, чтобы Lincheck был доступен в виде библиотеки с интерфейсом на C/C++. Для этого его необходимо перевести на Kotlin/Native. В то же время необходимо сохранить JVM версию Lincheck, для этого общую часть необходимо переписать на Kotlin/Common

3. Разработать удобный интерфес на C++

   Для того, чтобы можно было удобно использовать Lincheck прямо из C++

4. Протестировать популярные библиотеки на C++

   Для подтверждения того, что всё работает

Достигнутые результаты

В данной работе Lincheck был адаптирован для тестирования кода на C++. Для этого он был переведен на Kotlin Multiplatform, собран в динамическую библиотеку с интерфейсом на C, и была написана удобная обертка на C++ для использования этой библиотеки. Поддержан стресс режим тестирования. Были протестированы существующие популярные библиотеки для многопоточного программирования на C++, а также написаны собственные тесты, которые проверяют, что Lincheck работает на структурах с внедренными ошибками.

Структура работы

В главе 1 представлен обзор существующих инструментов для тестирования многопоточных структур данных.

В главе 2 представлено описание работы Lincheck, технологии Kotlin Multiplatform, и способ применения этой технологии для адаптации Lincheck.

В главе 3 подробно описан процесс переписывания Lincheck на Kotlin Multiplatform.

В главе 4 описывается интерфейс для использования Lincheck из C++.

В главе 5 представлены результаты тестирования популярных многопоточных библиотек на C++.

В заключении представлены результаты проделанной работы и их анализ, а также рассматриваются возможности дальнейшей деятельности.

1. Обзор литературы

1.1. Линеаризуемость

Для удобства разработки и формализации корректности разрабатываются различные критерии корректности многопоточных программ. Самый популярный и используемый из них — это линеаризуемость

Линеаризуемость в многопоточном программировании — это свойство программы, при котором результат любого параллельного выполнения процедур (операций) эквивалентен некоторому последовательному выполнению, сохраняющему порядок операций. Каждая операция имеет время начала и окончания, и на операциях можно задать частичный порядок, при котором операция исполняется раньше другой, если время окончания операции было до времени начала другой операции. Последовательное исполнение должно сохранять этот порядок, и соответствовать исполнению в одном потоке. Эквивалентность исполнений означает эквивалентность набора операций, их аргументов и врзвращаемых значений.

Таким образом, чтобы доказать линеаризуемость исполнения, необходимо найти последовательный вариант этого же исполнения, при котором сохраняются все результаты.

1.2. Инструменты для тестирования многопоточных алгоритмов

Существует множество инструментов для тестирования многопоточных алгоритмов. Ниже представлена таблица, в которой представлены самые популярные инструменты, которые анализируются на 3 критерия: практичность, возможности, и доступность.

Практичность означает то, ориентировался ли инструмент на практичность при разработке, и является ли он достаточно удобным для использования в промышленных проектах.

Возможности означает то, способен ли инструмент тестировать структуры на различные сложные критерии корректности, такие как линеаризуемость.

Доступность означает то, доступен ли инструмент для открытого использования. Можно ли применять его и анализировать его код.

Название	Практичность	Возможности	Доступность	Последнее обновление
JCStress [16]	-	-	+	2021
Line-Up [20]	+	+	-	2010
CONCURRIT [5]	-	-	+	2013
Lincheck [30]	+	+	+	2021

JCStress. JCStress [16] — это инструмент для тестирования программ на языке Java. Он требует полного указания сценария исполнения, и всех возможных исходов. Это не является удобным для разработчика. Изначально JCStress разрабатывался для тестирования низкоуровневых свойств реализаций модели памяти Java, для исследования возможных побочных эффектов от модели памяти, и не ориентировался на промышленное использование.

Line-Up. Line-Up [20] — это инструмент для декларативного тестирования программ на языке C# на линеаризуемость. Он был разработан в 2010 году группой разработки Microsoft Research и закрыт для использования и анализа извне. Тем не менее, есть научная работа, описывающая его особенности. К сожалению инструмент перестал разрабатываться в 2010 году и никакой информации о его разработке нет.

CONCURRIT. CONCURRIT [5] — это чуть более удобный язык, позволяющий описывать многопоточные тесты для кода на языке C++. Он был разработан в 2013 году, и помогает лишь писать собственные тесты, но никак не помогает тестировать код на линеаризуемость, и, соответственно, не является практичным.

Lincheck. Lincheck [30] — это декларативный фреймворк для тестирования многопоточных алгоритмов и структур данных. Идея его работы достаточно простая: он декларативно получает высокоуровневое описание тестируемой структуры данных и настройки тестирования, далее он генерирует множество случайных сценариев исполнения. Сценарий исполнения включает в себя операции, исполняемые до параллельной части, множество параллельных операций, которые будут исполняться в параллельных потоках, и операции, исполняющиеся после параллельной части. После генерации случайных сценариев он исполняет каждый сценарий много раз, анализируя каждое исполнение на линеаризуемость. Линеаризуемость проверяется с помощью поиска последовательного варианта исполнения тех же операций с сохранением их частичного порядка, с получением эквивалентных результатов.

Другие инструменты. Существуют также научные работы, посвященные проверке линеаризуемости [27, 9]. Есть также работы, которые демонстрируют подход к генерации большого количества юнит тестов [3, 27], что схоже с генерацией случайных сценариев в Lincheck. Другие же работы ориентированы на управление исполнением многопоточных тестов, более удобное написание и лучшее качество [14]. Некоторые работы позволяют тестировать различные критерии корректности [10].

Были предложены и Model Checking инструменты, такие как CHESS [6], и Java Path Finder [11], многие из них значительно превосходят lincheck по возможностям, а некоторые даже поддерживают модели с ослабленной памятью [2, 8]. Однако такие

инструменты проверки моделей трудно использовать в промышленной разработке, так как они требуют переписывания алгоритма на определенном языке.

Существуют инструменты, которые вместо поиска фактических ошибок ищут гонки по данным, которые могут вызвать ошибки [26, 25, 29, 13]. Тем не менее, эти инструменты не эффективны для тестирования lock-free алгоритмов, которые, как правило, не имеют гонок.

В общем существуют и другие инструменты, которые превосходят Lincheck в конкретных приложениях.

Стоит отметить, что на данный момент популярные библиотеки на C++ используют собственную инфраструктуру для тестирования, вместо использования существующих инструментов.

1.3. Выводы

На данный момент существующие многопоточные библиотеки вынуждены писать собственную инфраструктуру для тестирования и тестировать код на ограниченном наборе сценариев, не проверяя линеаризуемость.

Существует ровно два практичных инструмента для тестирования, которые умеют тестировать линеаризуемость по декларативному высокоуровневому описанию структуры данных: Line-Up и Lincheck. Line-Up перестал разрабатываться в 2010 году, доступен только для языка C#, и не доступен для использования и анализа. Lincheck же является современным инструментом, но умеет тестировать только языки, работающие на виртуальной машине Java(JVM), такие как Java, Kotlin, Scala.

Более того, Lincheck практически полностью написан на языке Kotlin, который является мультиплатформенным, что может быть использовано для использования Lincheck для тестирования кода на C++.

2. Lincheck, Kotlin Multiplatform

Lincheck начал разрабатываться в 2016 году группой devexperts и в 2019 году перешёл в экосистему Kotlin. Он использует систему сборки maven, написан с помощью языков Kotlin и Java, и предназначен для декларативного тестирования многопоточных структур данных, написанных на языках, работающих на виртуальной машине Java(JVM), на линеаризуемость. Идею его работы можно вкратце описать диаграммой, представленной на рисунке 1.

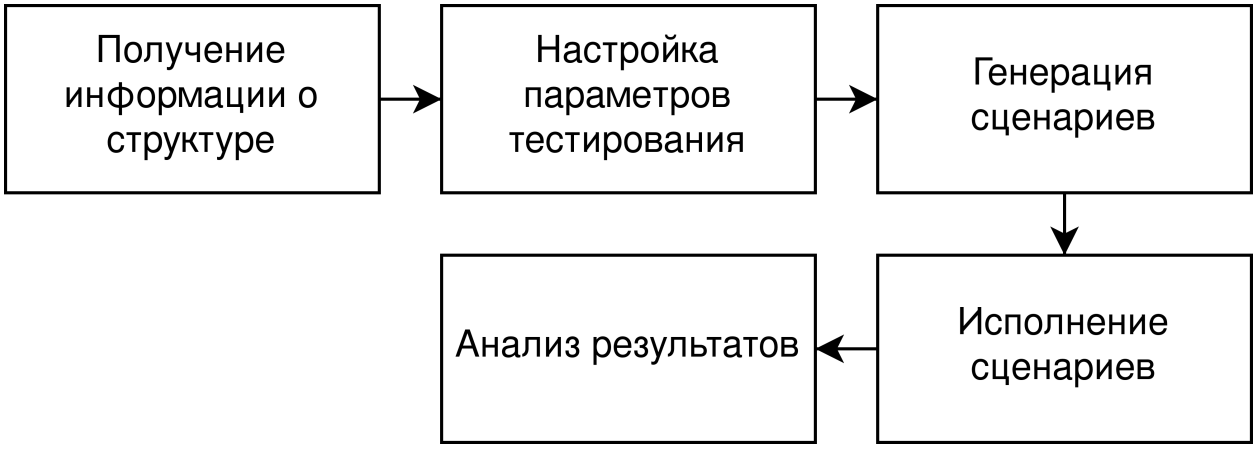

Рис. 1: Идея работы Lincheck.

2.1. Работа Lincheck

Lincheck декларативно получает высокоуровневую информацию о структуре данных, затем получает информацию о параметрах тестирования, затем генерирует много случайных сценариев, затем исполняет каждый сценарий много раз и анализирует результаты на корректность. Пример описания теста приведен в листинге 1

```
class StackTest {
    val stack = Stack<Int>()

    @Operation fun pop(): Int = stack.pop()
    @Operation fun push(value: Int) = stack.push(value)

    @Test fun test() {
    Options opts = new StressOptions()
        .iterations(10)
        .threads(3)
        .logLevel(LoggingLevel.INFO);
    LinChecker.check(StackTest.class, opts);
  }
}
```

Листинг 1: Описание Lincheck теста на Kotlin/JVM

Тест описывается классом, который должен иметь конструктор без аргументов, и множество методов, помеченных аннотацией `Operation`, которые означают тестируемые многопоточные операции. Также необходимо создать объект типа `Options` с помощью методов которого можно настроить параметры тестирования, такие как количество итераций(количество различных случайных сценариев для тестирования), количество параллельных потоков для тестирования, и другие.

Далее класс, который правильным образом размечен для тестирования, и объект типа `Options` необходимо передать в метод `Linchecker.check`, что запустит сам процесс тестирования. Lincheck с помощью Java Reflection проанализирует все аннотации и проведёт тестирование, вернув результат обратно.

2.2. Kotlin Multiplatform

Язык Kotlin является мультиплатформенным, что позволяет один раз писать код, который будет исполняться на различных платформах. Существуют платформы Kotlin/JVM, Kotlin/Native, Kotlin/JS. Kotlin/Common является общей платформой, код на которой способен исполняться на любой платформе.

Kotlin/JVM позволяет программе исполняться на виртуальной машине Java(JVM), а также имеет тесную интеграцию с языком Java, и позволяет использовать все библиотеки из Java, в частности, Java Reflection.

Kotlin/Native позволяет программе компилироваться в машинный код и исполняться напрямую на процессоре, а также имеет тесную интеграцию с языком C. Важной особенностью является то, что программу на Kotlin/Native можно собрать в динамическую библиотеку с общедоступным интерфейсом на языке C, и использовать её напрямую из C/C++, с помощью этого интерфейса.

Kotlin/JS позволяет программе исполняться в любом месте, где может исполняться язык Javascript, в частности в браузерах. Нас он не будет интересовать, так как в нём, по сути, нет настоящей многопоточности.

Lincheck написан на Kotlin/JVM и Java, и изначально не подразумевалось, что он будет использовать Kotlin Multiplatform. Поэтому в нём используются Java-специфичные вещи, наиболее значимой из которых является Java Reflection.

Если получится перевести Lincheck на Kotlin/Native, это позволит получить тесную интеграцию с языком C, и тем самым позволит тестировать программы, написаные на C и C++

2.3. Описание этапов и план

На рисунке 2 показан каждый этап работы Lincheck. Зеленым выделены те компоненты, которые могут быть успешно переведены в Kotlin/Common часть. Синим выделены те, которые имеют разную реализацию на Kotlin/JVM и Kotlin/Native.

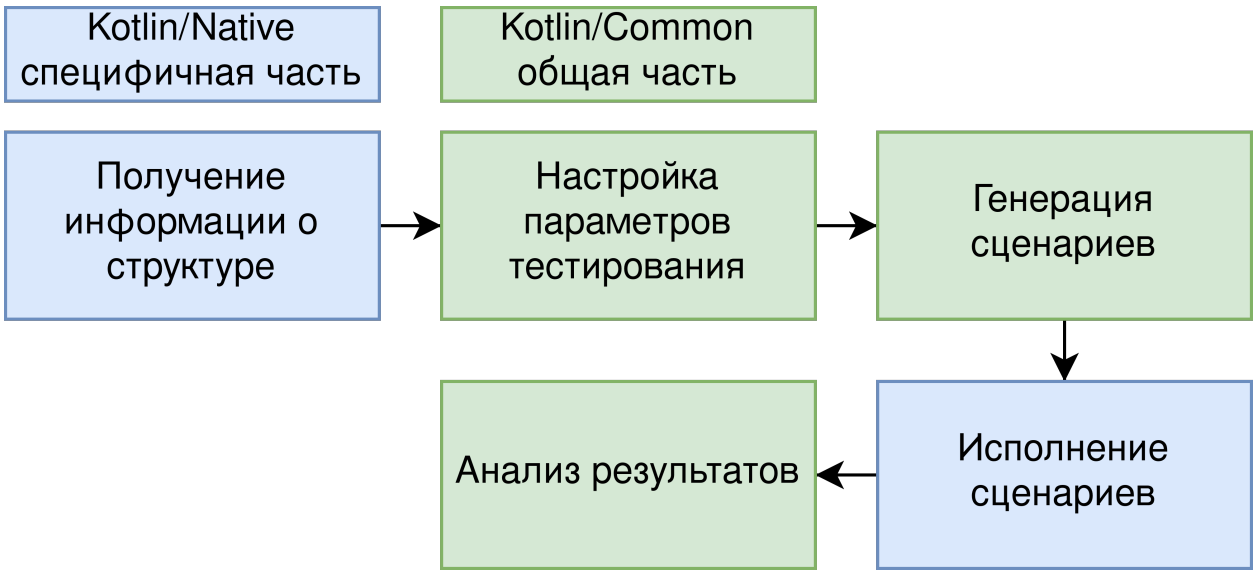

Рис. 2: Разделение компонент Lincheck на Kotlin/Common и Kotlin/Native.

Далее я расскажу про то, что происходит на каждом этапе, и почему некоторые компоненты были написаны заново в Kotlin/Native части

Получение информации о структуре данных Для работы Lincheck должен создавать начальное состояние, и применять к нему операции. Для этого ему необходимо каким-то образом передать генератор начального состояния, и операции.

В Kotlin/JVM части достаточно передать класс, и для генерации начального состояния будет использован конструктор без аргументов, а для операций — методы, помеченные аннотацией `Operation`.

В Kotlin/Native нет возможности использовать аннотации, поэтому необходимо придумать новый способ получения генератора начального состояния и операций. Было решено для передачи начального состояния использовать просто функцию-генератор, а операции выражать с помощью функций, которые принимают первым аргументом состояние. Подробнее про способ передачи будет в следующей главе.

Настройка параметров тестирования Lincheck имеет большое количество настроек, в листинге 2 представлена часть из них. Все они необходимы для работы Lincheck, и позволяют регулировать скорость, тщательность и сложность тестирования. Все они настраиваются с помощью вышеупомянутого класса `StressOptions`.

Было решено перевести класс StressOptions в Kotlin/Common часть, тем самым унифицировав опции тестирования.

Генерация сценариев Генерация сценариев происходит с помощью случайного выбора набора операций, которые будут исполняться. Генерируется определенное, настраиваемое количество операций до параллельной части, в каждом параллельном

```
invocationsPerIteration(count: Int)
iterations(count: Int)
threads(count: Int)
actorsPerThread(count: Int)
actorsBefore(count: Int)
actorsAfter(count: Int)
requireStateEquivalenceImplCheck(require: Boolean)
minimizeFailedScenario(minimize: Boolean)
logLevel(level: LogLevel)
```

Листинг 2: Список параметров тестирования

потоке, и после параллельной части.

Эту часть было решено перевести в Kotlin/Common.

Исполнение сценариев Исполнение сценариев происходит в три этапа. Сначала главный поток исполняет операции до параллельной части, затем он отправляет параллельные операции на исполнение в специальную систему, которая умеет исполнять параллельную часть в фиксированном числе потоков, затем он исполняет последнюю последовательную часть. После собираются все результаты и отдаются на этап анализа.

Эта часть привязана к платформе, на которой исполняется, поскольку она требует работы с потоками. В Kotlin/JVM используются потоки Java, а в Kotlin/Native особенная модель Worker'ов. Было решено написать всю подсистему исполнения сценариев с нуля(запуск операций, сбор результатов) на Kotlin/Native.

Анализ результатов Анализ результатов строит граф состояний, который описывает состояние структуры данных и возможные переходы с помощью операций. Это необходимо для того, чтобы оптимально производить поиск последовательной истории, эквивалентной исполнению. Запускается перебор всех историй, который постепенно пополняет граф состояний, и постоянно сравнивает результаты с фактическими от параллельного исполнения.

Этот этап не привязан к платформе, поэтому было решено перевести его в Kotlin/Common часть.

2.4. Выводы

Была проанализирована технология Kotlin Multiplatform и разработан план по переводу каждого этапа работы Lincheck в Kotlin/Common или Kotlin/Native.

3. Переписывание Lincheck на Kotlin/Native

Как было показано в предыдущей главе, переписывание Lincheck на Kotlin/Common и Kotlin/Native позволяет получить тесную интеграцию с C, тем самым открывает возможность тестировать код на C/C++ с помощью Lincheck. В этой главе описывается процесс перевода Lincheck на Kotlin/Common и Kotlin/Native, и возникшие сложности в этом процессе.

3.1. Система сборки Lincheck

Изначально Lincheck использовал систему сборки maven, которая отлично подходит для написания программ на языке Java и Kotlin/JVM, но не умеет работать с технологией Kotlin Multiplatform. Для её использования Lincheck был переведен на систему сборки gradle, и настроен для технологии Kotlin Multiplatform.

3.2. Различия Kotlin/JVM и Kotlin/Native

Чтобы понять, что же нужно сделать, чтобы код на Kotlin/JVM стал работать на Kotlin/Native, необходимо разобраться, в чём их отличие.

1. Kotlin/JVM позволяет использовать все возможности языка Java. Kotlin/Native не имеет доступа к Java. В частности, нет возможности использовать аннотации для описания теста

2. Kotlin/JVM имеет модель памяти Java. Kotlin/Native имеет свою, особенную модель памяти, которая запрещает доступ к изменяемым объектам с разных потоков, хранит для каждого объекта флаг изменяемости, и имеет модель Workers для многопоточной работы.

3.3. Lincheck на Kotlin/Common и Kotlin/Native

В идеальном варианте хотелось бы переписать Lincheck полностью на Kotlin/Common, но, к сожалению, на данный момент это невозможно, поскольку Lincheck использует JVM-специфичные трансформации кода, аннотации, и модель памяти.

Получение информации о структуре данных Получение информации о структуре данных в Kotlin/JVM, как показано в листинге 1, производится с помощью аннотаций. К сожалению в Kotlin/Native нет возможности использовать аннотации, так как нет Java Reflection. Поэтому эта часть была переписана на Kotlin/Native, и был разработан интерфейс, позволяющий указывать все необходимые данные для тестирования наиболее удобным образом. Kotlin/Native интерфейс для тестирования пред-

ставлен в листинге 4. С помощью паттерна ”Строитель” можно указывать всю необходимую информацию о структуре, такую как начальное состояние и параллельные операции, которые будут производиться над этим состоянием.

```
LincheckStressConfiguration<HashMap>().apply {
    iterations(10)
    threads(3)

    initialState { HashMap() }

    operation(IntGen(), IntGen(), HashMap::put)
    operation(IntGen(), HashMap::get)
}.runTest()
```

Листинг 3: Описание Lincheck теста на Kotlin/Native

Настройка параметров тестирования Настройка параметров ранее осуществлялась с помощью аннотаций, либо класса `StressOptions`, с помощью методов которого можно было указывать все настройки. Класс Options был переведен в Kotlin/Common, а `LincheckStressConfiguration` был отнаследован от `StressOptions`, позволяя указывать информацию о структуре и настраивать параметры тестирования в одном месте.

```
class LincheckStressConfiguration<Instance>() : StressOptions()
```

Листинг 4: Наследование от StressOptions из Kotlin/Common

Генерация сценариев Генерация сценариев была представлена с помощью класса `RandomExecutionGenerator`, который также был переведен в Kotlin/Common. Сам алгоритм генерации остался без изменений, изменилось лишь представление объектов, с которыми он работает.

Исполнение сценариев Исполнение сценариев было написано с нуля, и представлено классами `ParallelThreadsRunner` и `FixedActiveThreadsExecutor`. Класс `ParallelThreadsRunner` умеет исполнять последовательные части сценария и собирать результаты, отдавая их на этап анализа. `FixedActiveThreadsExecutor` умеет запускать параллельную часть в фиксированном числе потоков.

Kotlin/Native вводит модель владения изменяемыми объектами, когда каждый изменяемый объект принадлежит ровно одному потоку, и доступен только в нём. А у нас все операции производятся над единственным изменяемым состоянием. Поэтому был использован небезопасный экспериментальный режим Kotlin/Native, который выключает проверки корректности модели памяти, и не гарантирует стабильную работу. На

самом деле из-за этого каждый поток думает, что владеет изменяемым объектом, и работает с ним как с обычным. Проблемы возникают, когда сборщик мусора пытается удалить этот объект. Из-за многопоточной небезопасности сборщика мусора, при параллельном доступе к изменяемому объекту, будь то удаление, или просто передача владения, происходит некорректный подсчёт ссылок на объект, что приводит к некорректной работе сборщика мусора, и непределенному поведению, которое может выражаться в двойном удалении памяти или нарушении инвариантов сборщика мусора.

Было замечено, что в этом небезопасном режиме всё стабильно работает, если не делать количество запусков одного сценария слишком большим. Экспериментальным путём было установлено число в 500 запусков каждого сценария, при чём количество различных сценариев можно увеличивать без последствий.

В настоящий момент сборщик мусора Kotlin/Native переписывается на потокобезопасный [17], что позволит избавиться от этого ограничения и получить стабильную работу. Lincheck был проверен с помощью Kotlin/Native компилятора с выключенным сборщиком мусора, и не было замечено никаких проблем в работе, при чём количество запусков одного сценария без проблем увеличивалось до сотен тысяч.

Анализ результатов Верификация результатов была представлена множеством объектов типа `Verifier`, в частности, `LinearizabilityVerifier`, который проверяет линеаризуемость. Большинство объектов типа `Verifier` полагаются на объект типа `LTS`(Labeled Transition System), который представлен как граф состояний, с переходами в виде операций, и который помогает сильно ускорить поиск последовательной истории исполнения. Все эти классы были переведены в Kotlin/Common часть.

3.4. Java Reflection

Lincheck для передачи информации, необходимой для тестирования, активно использует Java Reflection. Это сильно помогает, поскольку Lincheck использует трансформацию кода для поддержки `suspend` функций и для вставки необходимых вызовов для управляемого исполнения. К сожалению, в Kotlin/Common перевести напрямую это невозможно, поэтому все такие случаи были заменены на expect-actual классы, которые позволяют иметь один интерфейс в Kotlin/Common, и разную реализацию в Kotlin/JVM и Kotlin/Native. Пример использования expect-actual механизма представлен в листинге 5.

Так, в листинге 3 представлена часть внутреннего класса, который хранит информацию о конфигурации тестирования, до перевода, и после. Видно, что ранее для передачи информации о тестовом классе, о генераторе случайных сценариев, о верификаторе, который анализирует корректность исполнения, и о последовательной

```kotlin
// Kotlin/Common
expect class TestClass {
    fun createInstance(): Any
}

// Kotlin/JVM
actual class TestClass(val clazz: Class<*>) {
    val name = clazz.name

    actual fun createInstance(): Any =
        clazz.getDeclaredConstructor().newInstance()
}

// Kotlin/Native
actual class TestClass(val create: () -> Any?) {
    actual fun createInstance(): Any = create()
}
```

Листинг 5: Expect-actual механизм

спецификации, использовался Java Reflection. В Kotlin/Common это было перенесено с помощью абстрактных классов, как `TestClass` и `SequentialSpecification`, так и с помощью функциональных типов, там где Java Reflection использовался исключительно для вызова конструктора.

```kotlin
// Before, Kotlin/JVM
class StressCTestConfiguration(
testClass: Class<*>,
gen: Class<out ExecutionGenerator>,
verifier: Class<out Verifier>,
seqSpec: Class<*>?
)
```

: StressCTestConfiguration раньше

```kotlin
// After, Kotlin/Common
class StressCTestConfiguration(
testClass: TestClass,
gen: (_, _) -> ExecutionGenerator,
verifier: (spec) -> Verifier,
seqSpec: SequentialSpecification<*>
)
```

: StressCTestConfiguration сейчас

Рис. 3: Внутреннее представление конфигурации теста в Lincheck, до и после.

Абстрактные классы необходимы там, где реализация каких-то вещей должна быть различной на Kotlin/JVM и Kotlin/Native. Это может быть Java Reflection и трансформация кода. В листинге 5 представлен класс `TestClass`, который умеет создавать начальное состояние с помощью `TestClass.createInstance`. В Kotlin/JVM части он реализован с помощью Java Reflection и вызова конструктора, а в Kotlin/Native он реализован с помощью функции-генератора. В Kotlin/JVM необходимо сохранить доступ к Java Reflection, поскольку он используется для трансформации кода, и для соответствия названий методов у тестируемой структуры данных, и у её последовательной спецификации.

3.5. Тестирование

Было написано 20 тестов, которые тестируют Lincheck из Kotlin/Native. Большинство из них было написано по аналогии с уже существующими Kotlin/JVM тестами. Было протестировано, что Lincheck успешно находит ошибки в некорректных реализациях структур данных, и успешно тестирует корректные.

В листинге 6 представлен пример тестирования счетчика на Kotlin/Native с помощью Lincheck, а также пример вывода сообщения о некорректной структуре данных.

```
class Counter : VerifierState() {
    var state: Int = 0

    fun increment(): Int {
        return ++state
    }

    fun decrement(): Int {
        return --state
    }

    override fun extractState(): Any {
        return state
    }

    @Test
    fun test() {
        LincheckStressConfiguration<Counter>().apply {
            iterations(10)
            invocationsPerIteration(500)
            actorsBefore(2)
            threads(3)
            minimizeFailedScenario(true)

            initialState { Counter() }

            operation(Counter::increment, "increment")
            operation(Counter::decrement, "decrement")
        }.runTest()
    }
}

// Test result
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Init part:
[decrement(): -1]
Parallel part:
| increment(): 0 | increment(): 0 |
Post part:
[decrement(): -1]
```

Листинг 6: Тестирование счетчика на Kotlin/Native с помощью Lincheck

3.6. Выводы

Lincheck был переписан на Kotlin/Common и Kotlin/Native, и протестирован на работоспособность в качестве инструмента для тестирования Kotlin/Native структур данных.

После успешной адаптации Lincheck для тестирования программ на Kotlin/Native открывается возможность тестировать программы на C/C++, поскольку, как описано в предыдущей главе, Kotlin/Native имеет тесную интеграцию с C и позволяет генерировать динамическую библиотеку с интерфейсом на C.

4. Интерфейс на C++ для Lincheck

После перевода Lincheck на Kotlin/Common и Kotlin/Native необходимо предоставить удобный способ использовать его возможности из C++. Это позволит программистам на C++, не изучая язык Kotlin, использовать возможности Lincheck для тестирования.

4.1. Интерфейс на C

Kotlin/Native позволяет собирать программу в динамическую библиотеку с общедоступным интерфейсом на C. Тем не менее, для работы необходимо преобразовывать структуры данных на C в структуры данных на Kotlin/Native, и уже потом отправлять их на тестирование. Для этого в Kotlin/Native был разработан класс `NativeAPIStressConfiguration`, который принимает необходимые параметры из C, оборачивает их в Kotlin/Native объекты, и отправляет их на тестирование.

```
1  class NativeAPIStressConfiguration : LincheckStressConfiguration<Any>() {
2      fun setupOperation2(
3          arg1_gen_initial_state: CCreator,
4          arg1_gen_generate: CPointer<CFunction<(CPointer<*>) -> CPointer<*>>>,
5          arg1_toString: ToStringCFunction,
6          arg1_destructor: CDestructor,
7          op: CPointer<CFunction<(CPointer<*>, CPointer<*>) -> CPointer<*>>>,
8          seq_spec: CPointer<CFunction<(CPointer<*>, CPointer<*>) ->
               CPointer<*>>>,
9          result_destructor: CDestructor,
10         result_equals: EqualsCFunction,
11         result_hashCode: HashCodeCFunction,
12         result_toString: ToStringCFunction,
13         operationName: String,
14         nonParallelGroupName: String? = null,
15         useOnce: Boolean = false,
16     ) = apply {
17         val arg1_paramgen = object :
               ParameterGenerator<ParameterGeneratorArgument> {
               arg1_gen_initial_state, arg1_gen_generate, arg1_toString,
               arg1_destructor }
18         val actorGenerator = ActorGenerator(
19             function = { instance, arguments -> op, seq_spec,
                   result_destructor, result_equals, result_equals,
                   result_hashCode, result_toString },
20             parameterGenerators = listOf(arg1_paramgen)
21         )
22         actorGenerators.add(actorGenerator)
23         addToOperationGroups(nonParallelGroupName, actorGenerator)
24     }
25 }
```

Листинг 7: Обработка информации об операции с двумя аргументами из C в Kotlin/Native

Этот интерфейс не предназначен для прямого использования, поэтому там передаются указатели на функции, которые на Kotlin/Native стороне оборачиваются в Kotlin/Native функции. В листинге 7 представлена часть этого интерфейса на Kotlin/Native.

В листинге 7 представлена функция, позволяющая указать операцию с одним аргументом. Как видно, для неё необходима информация о генераторе аргументов, которая представляется четыремя функциями: `arg1_gen_initial_state` генерирует начальное состояние генератора, `arg1_gen_generate` по состоянию генератора генерирует аргумент, изменяя состояние генератора, `arg1_toString` преобразует аргумент в строку, которая будет выводиться, если будет обнаружена некорректная реализация структуры данных, `arg1_destructor` необходим для контроля памяти, выделенной в C/C++.

Для описания операции и последовательной спецификации используются указатели на функции, которые принимают общее состояние первым аргументом, а затем принимают аргументы операции.

Также операция имеет возвращаемое значение, которое, по аналогии с аргументом, умеет превращаться в строку, умеет очищать за собой память, а также сравниваться с такими же возвращаемыми значениями. Это необходимо для анализа линеаризуемости, поскольку при анализе сравниваются результаты.

В языке C нет объектов. Kotlin/Native объекты там представляются в виде анонимных структур с анонимными функциями, поэтому для полного их описания необходимо передавать много различной информации, в частности методы `equals`, `hashCode`, `toString`.

4.2. Интерфейс на C++

Для удобного использования Lincheck был разработан интерфейс на C++, который в удобном виде принимает информацию о структуре данных, и отправляет её в интерфейс на C для тестирования. В листинге 8 представлен интерфейс на C++, с помощью которого можно использовать Lincheck, а в листинге 9 представлен пример использования.

Суть интерфейса Суть интерфейса заключается в получении информации о структуре данных, такой как конструкторы, деструкторы, хеш-функция, функция сравнения, функция преобразования объекта в строку, а также об объектах, генерирующих случайные аргументы для операций, и передаче этой информации в правильном формате в интерфейс на C. В листинге 7 описан полный список. Для их получения используются стандартные практики из C++, которые пытаются найти правильную специализацию необходимых функций. Интерфейс требует определения шаблонных клас-

```
template<typename TestClass, typename SequentialSpecification>
class LincheckConfiguration {
    void iterations(int count);
    void invocationsPerIteration(int count);
    void minimizeFailedScenario(bool minimizeFailedScenario);
    void threads(int count);
    void actorsPerThread(int count);
    void actorsBefore(int count);
    void actorsAfter(int count);
    template<void (*init)()> void initThreadFunction();
    template<void (*finish)()> void finishThreadFunction();
    template<void (TestClass::*validate)()> void validationFunction();

    // operation without arguments
    template<
        typename Ret,
        Ret (TestClass::*op)(),
        Ret (SequentialSpecification::*seq_spec)()
    >
    void operation(const char *operationName);

    // operation with 2 arguments
    template<
        typename Ret,
        typename Arg1,
        typename Arg2,
        Ret (TestClass::*op)(Arg1, Arg2),
        Ret (SequentialSpecification::*seq_spec)(Arg1, Arg2)
    >
    void operation(const char *operationName);
}
```

Листинг 8: Интерфейс на C++

сов, таких как `Lincheck::hash<T>`, `operator==(T, T)`, `Lincheck::to_string<T>` для возвращаемых значений(результатов операций) и последовательной спецификации, `Lincheck::to_string<T>` и `Lincheck::ParameterGenerator<T>` для аргументов. Для самых используемых, стандартных типов, эти структуры определены в `lincheck.h`

Устройство интерфейса Он описан в заголовочном файле `lincheck.h`, и всё взаимодействие происходит через класс `LincheckConfiguration`, который принимает тип тестируемой структуры данных и тип последовательной спецификации, как типовые аргументы.

Далее, с помощью методов можно выставлять все необходимые настройки тестирования, такие как количество итераций и потоков. Каждая операция задаётся методом operation, который является шаблонным, и принимает первым типовым аргументом тип возвращаемого значения, далее принимает типы аргументов, которые будут переданы операции. Последние два типовых аргумента отвечают за операцию и последовательный вариант. В качестве обычных аргументов передаётся название операции

```
#include "lincheck.h"

int main() {
    LincheckConfiguration<Counter, Counter> conf;
    conf.iterations(10);
    conf.threads(3);
    // return type, argument type, operation, sequential specification
    conf.operation<int, int, &Counter::add, &Counter::add>("add");
    conf.runTest();
}
```

Листинг 9: Пример использования Lincheck из C++

для красивого вывода об ошибке.

Для запуска тестирования необходимо вызвать метод `runTest`, который произведёт тестирование, и вернёт сообщение об ошибке, если она была найдена.

В своей реализации C++ интерфейс генерирует функции, которые принимают и возвращают `void*`, с помощью добавления правильного типа к `void*`, вызова необходимой функции, и возврата правильного значения через кучу. Далее он передаёт сгенерированные функции в интерфейс на C, тем самым настраивая тестирование.

4.3. Выводы

Был реализован интерфейс на C++, с помощью которого можно использовать Lincheck для тестирования структур данных, написанных в C++.

5. Тестирование

С помощью интерфейса на C++, представленного в прошлой главе, были протестированые популярные многопоточные библиотеки на C++. А именно: libcuckoo[1], folly[2], boost::lockfree[3], libcds[4]. Информация о протестированных библиотеках и структурах данных представлена в таблице 1

Таблица 1: Краткое описание тестируемых структур данных.

Библиотека	Структура данных	Описание
Libcuckoo	HashMap	Хеш таблица с операциями присваивания значения по ключу, удаления ключа, и получения значения по ключу
Folly	HashMap	Хеш таблица с операциями присваивания значения по ключу, удаления ключа, и получения значения по ключу
	DynamicBoundedQueue	Очередь, динамически меняющая объем выделенной памяти, с верхним ограничением на количество элементов, с операциями добавления элемента в очередь и извлечения элемента из очереди
	UnboundedQueue	Очередь, динамически меняющая объем выделенной памяти, без верхнего ограничения на количество элементов, с операциями добавления элемента в очередь и извлечения элемента из очереди
	[SM]P[SM]C(Un)BoundedQueue	Вариации DynamicBoundedQueue и UnboundedQueue, в которых ограничивается число потоков-писателей и потоков-читателей (single/multiple producer, single/multiple consumer)
Boost::Lockfree	Queue	Очередь с операциями добавления и удаления элемента
	SPSC_Queue	Очередь с операциями добавления и удаления элемента, где в каждый момент времени есть не более одного потока, который извлекает элемент, и не более одного, который добавляет элемент
	Stack	Стек с операциями добавления и удаления элемента
Libcds	MSQueue	Очередь Майкла и Скотта [23] с операциями добавления и удаления элемента
	TreiberStack	Стек Трайбера [31] с операциями добавления и удаления элемента

Конфигурация оборудования. Эксперименты производились на Lenovo Yoga Slim 7 14ARE05 с процессором AMD Ryzen 7 4800U и 16гб оперативной памяти. Использовалась операционная система Linux.

Конфигурация тестов. Для настроек тестирования было использовано 100 итераций по 500 запусков с 4 операциями в каждом параллельном потоке. Было использовано 3 параллельных потока.

Структура тестов. Все тесты были написаны в одном cmake проекте. В файле `CMakeLists.txt` описан способ подключения Lincheck к существующему проекту, а

[1]https://github.com/efficient/libcuckoo
[2]https://github.com/facebook/folly
[3]https://github.com/boostorg/lockfree
[4]https://github.com/khizmax/libcds

также подключена технология `gtest`, с помощью который были написаны тесты. Библиотеки на языке C++ заргужаются, собираются, и подключаются автоматически с github.

Производительность Было замерено время работы каждого теста. Данные представлены в таблице 2. Тесты с приставкой `Bad` означают, что тест должен был упасть, и проверялось наличие ошибки. Время работы для таких тестов отражает скорость обнаружения ошибки, и оно не превышает двух секунд. Остальные тесты тестируют соответствующие структуры данных на 100 ирациях и 500 запусках на каждой итерации, с 3 потоками. В среднем тестирование занимает 2 минуты, с небольшим разбросом.

Таблица 2: Список тестов и время работы.

Тест	Время работы
LibcdsTest.BadSequentialQueueTest	90ms
LibcdsTest.BadSequentialStackTest	1sec
LibcdsTest.ConcurrentQueueHPTest	2min 13ms
LibcdsTest.ConcurrentQueueDHPTest	2min 13ms
LibcdsTest.ConcurrentTreiberStackHPTest	1min 46ms
LibcdsTest.ConcurrentTreiberStackDHPTest	1min 52ms
LibcuckooTest.BadSequentialMapTest	1sec
LibcuckooTest.HashMapTest	5min 40ms
FollyTest.BadSequentialMapTest	1sec
FollyTest.BadSequentialQueueTest	80ms
FollyTest.HashMapTest	2min 2sec
FollyTest.MPMCDynamicBoundedQueueTest	2min 30sec
FollyTest.MPSCDynamicBoundedQueueTest	5min 3sec
FollyTest.BadMPSCDynamicBoundedQueueTest	50ms
FollyTest.SPMCDynamicBoundedQueueTest	1min 50sec
FollyTest.SPSCDynamicBoundedQueueTest	1min 27sec
FollyTest.BadSPSCDynamicBoundedQueueTest	130ms
FollyTest.MPMCUnboundedQueueTest	2min 7sec
FollyTest.MPSCUnboundedQueueTest	4min 45sec
FollyTest.SPMCUnboundedQueueTest	1min 49sec
FollyTest.SPSCUnboundedQueueTest	1min 25sec
BoostLockfreeTest.BadSequentialQueueTest	120ms
BoostLockfreeTest.BadSequentialStackTest	30ms
BoostLockfreeTest.QueueTest	2min 0sec
BoostLockfreeTest.StackTest	1min 39sec
BoostLockfreeTest.BadSPSCQueueTest	970ms
BoostLockfreeTest.SPSCQueueTest	1min 18sec

Результаты Все структуры данных успешно прошли тестирование, и Lincheck не выявил ни одной ошибки. С каждым тестом в паре был тест последовательной спецификации, который проверял, что Lincheck выявляет ошибки в последовательной версии структуры.

Также были протестированы модифицированные `MSQueue` и `TreiberStack` из libcds, с добавленными искуственными ошибками. Lincheck успешно их обнаружил за несколько секунд и предоставил развернутый сценарий для воспроизведения.

5.1. Выводы

С помощью инструмента были протестированы популярные многопоточные библиотеки на C++. В результате полученный инструмент, с одной стороны, проводит проверку корректности для потокобезопасных структур данных, а, с другой стороны, находит ошибки в некорректных реализациях. Время работы инструмента на корректных структурах в среднем составляет две минуты для тестирования на 3 потоках, используя 100 различных сценариев с 500 запусками каждого. Некорректные структуры данных обнаруживаются значительно быстрее, в среднем за одну секунду.

Заключение

Результаты

В результате данной работы Lincheck был адаптирован для тестирования кода на C++, исходный код доступен на Github [19]

Полученный инструмент был проверен с помощью тестирования популярных библиотек на C++ и самописных структур данных. Корректные реализации алгоритмов успешно проходят проверку, а в некорректных реализациях инструмент находит ошибку.

Дальнейшая работа

В дальнейшем может быть полезно оптимизировать скорость работы, так как Kotlin/Native находится в бета версии, и ещё не оптимизирован достаточно сильно. Также можно добавить поддержку model checking режима работы, что позволит генерировать более информативное сообщение об ошибке. Также можно добавить поддержку блокирующихся и неточных структур данных, так как они иногда используются на практике.

Список литературы

[1] Afek Yehuda, Korland Guy, Yanovsky Eitan. Quasi-linearizability: Relaxed consistency for improved concurrency // International Conference on Principles of Distributed Systems / Springer. — 2010. — P. 395–410.

[2] Alglave Jade, Kroening Daniel, Tautschnig Michael. Partial Orders for Efficient Bounded Model Checking of Concurrent Software // Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044. — 2013. — 07. — P. 141–157.

[3] Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code / Adrian Nistor, Qingzhou Luo, Michael Pradel et al. // 2012 34th International Conference on Software Engineering (ICSE) / IEEE. — 2012. — P. 727–737.

[4] Bernstein Philip A, Hadzilacos Vassos, Goodman Nathan. Concurrency control and recovery in database systems. — Addison-wesley Reading, 1987. — Vol. 370.

[5] CONCURRIT: A domain specific language for reproducing concurrency bugs / Tayfun Elmas, Jacob Burnim, George Necula, Koushik Sen // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2013. — P. 153–164.

[6] Chess: A systematic testing tool for concurrent software : Rep. / Technical Report MSR-TR-2007-149, Microsoft Research ; Executor: Madanlal Musuvathi, Shaz Qadeer, Thomas Ball et al. : 2007.

[7] Barras Bruno, Boutin Samuel, Cornes Cristina et al. The Coq proof assistant reference manual: Version 6.1 : Ph. D. thesis / Bruno Barras, Samuel Boutin, Cristina Cornes et al. ; Inria. — 1997.

[8] Effective Stateless Model Checking for C/C++ Concurrency / Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, Viktor Vafeiadis // Proc. ACM Program. Lang. — 2017. — Dec. — Vol. 2, no. POPL. — Access mode: `https://doi.org/10.1145/3158105`.

[9] Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing / Guangpu Li, Shan Lu, Madanlal Musuvathi et al. // Proceedings of the 27th ACM Symposium on Operating Systems Principles. — 2019. — P. 162–180.

[10] Emmi Michael, Enea Constantin. Sound, complete, and tractable linearizability monitoring for concurrent collections // Proceedings of the ACM on Programming Languages. — 2017. — Vol. 2, no. POPL. — P. 1–27.

[11] Havelund Klaus, Pressburger Thomas. Model checking java programs using java pathfinder // International Journal on Software Tools for Technology Transfer. — 2000. — Vol. 2, no. 4. — P. 366–381.

[12] Herlihy Maurice P., Wing Jeannette M. Linearizability: A correctness condition for concurrent objects. // ACM Transactions on Programming Languages and Systems. — 1990. — Vol. 12, no. 3. — Access mode: https://doi.org/10.1145/78969.78972.

[13] Huang Jeff, Meredith Patrick O'Neil, Rosu Grigore. Maximal Sound Predictive Race Detection with Control Flow Abstraction // SIGPLAN Not. — 2014. — Jun. — Vol. 49, no. 6. — P. 337–348. — Access mode: https://doi.org/10.1145/2666356.2594315.

[14] Improved multithreaded unit testing / Vilas Jagannath, Milos Gligoric, Dongyun Jin et al. // Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. — 2011. — P. 223–233.

[15] Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning / Ralf Jung, David Swasey, Filip Sieczkowski et al. // ACM SIGPLAN Notices. — 2015. — Vol. 50, no. 1. — P. 637–650.

[16] Java Concurrency Stress (jcstress). — https://github.com/openjdk/jcstress. — Accessed: 2021-05-30.

[17] Kotlin/Native Memory Management Roadmap. — https://blog.jetbrains.com/kotlin/2020/07/kotlin-native-memory-management-roadmap/, note = Accessed: 2021-05-30.

[18] Lamport Leslie. How to make a multiprocessor computer that correctly executes multiprocess programm // IEEE Computer Architecture Letters. — 1979. — Vol. 28, no. 09. — P. 690–691.

[19] Lincheck Pull Request: Kotlin/Native and C/C++ support. — https://github.com/Kotlin/kotlinx-lincheck/pull/68. — Accessed: 2021-05-30.

[20] Line-up: a complete and automatic linearizability checker / Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, Roy Tan // ACM SIGPLAN Notices. — 2010. — Access mode: https://doi.org/10.1145/1809028.1806634.

[21] Liskov Barbara, Zilles Stephen. Programming with abstract data types // ACM Sigplan Notices. — 1974. — Vol. 9, no. 4. — P. 50–59.

[22] Michael Maged M. ABA prevention using single-word instructions // IBM Research Division, RC23089 (W0401-136), Tech. Rep. — 2004.

[23] Michael Maged M, Scott Michael L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms // Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. — 1996. — P. 267–275.

[24] Moir Mark, Shavit Nir. Concurrent data structures // Handbook of data structures and applications. — Chapman and Hall/CRC, 2018. — P. 741–762.

[25] Naik Mayur, Aiken Alex, Whaley John. Effective Static Race Detection for Java // Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '06. — New York, NY, USA : Association for Computing Machinery, 2006. — P. 308–319. — Access mode: https://doi.org/10.1145/1133981.1134018.

[26] O'Callahan Robert, Choi Jong-Deok. Hybrid Dynamic Data Race Detection // Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — PPoPP '03. — New York, NY, USA : Association for Computing Machinery, 2003. — P. 167–178. — Access mode: https://doi.org/10.1145/781498.781528.

[27] Pradel Michael, Gross Thomas R. Fully automatic and precise detection of thread safety violations // Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. — 2012. — P. 521–530.

[28] Quiescent consistency: Defining and verifying relaxed linearizability / John Derrick, Brijesh Dongol, Gerhard Schellhorn et al. // International Symposium on Formal Methods / Springer. — 2014. — P. 200–214.

[29] Sen Koushik. Race directed random testing of concurrent programs // PLDI '08. — 2008.

[30] Testing concurrency on the JVM with lincheck. / Nikita Koval, Maria Sokolova, Alexander Fedorov et al. // PPoPP '20. — 2020. — Access mode: https://doi.org/10.1145/3332466.3374503.

[31] Treiber R Kent. Systems programming: Coping with parallelism. — International Business Machines Incorporated, Thomas J. Watson Research ..., 1986.

[32] The art of multiprocessor programming / Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear. — Newnes, 2020.