

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS**

**Верификация алгоритма консенсуса в базе данных ВК и тестирование кода на
императивном языке программирования на графе состояний TLA+**

Обучающийся / Student Чернацкий Евгений Геннадьевич

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования

Группа/Group М34391

Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика

Образовательная программа / Educational program Информатика и программирование
2019

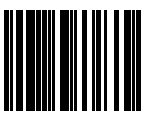
Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Бакалавр

Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки,
Университет ИТМО, институт прикладных компьютерных наук, доцент
(квалификационная категория "ординарный доцент")

Обучающийся/Student

Документ подписан	
Чернацкий Евгений Геннадьевич	
23.05.2023	

(эл. подпись/ signature)

Чернацкий
Евгений
Геннадьевич

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
17.05.2023	

(эл. подпись/ signature)

Аксенов
Виталий
Евгеньевич

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Чернацкий Евгений Геннадьевич
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group М34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Верификация алгоритма консенсуса в базе данных ВК и тестирование кода на императивном языке программирования на графе состояний TLA+
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Основные вопросы, подлежащие разработке / Key issues to be analyzed

Требуется разработать инструмент тестирования распределенных систем на основе модели, полученной из спецификации, написанной на языке TLA+. Разработанный инструмент должен быть использован для эффективного тестирования алгоритма репликации Viewstamped Replication, используемого в базе данных ВКонтакте GMS. Чтобы имелась возможность применить инструмент, требуется написать и верифицировать спецификацию TLA+ для алгоритма репликации Viewstamped Replication.

Дата выдачи задания / Assignment issued on: 01.03.2023

Срок представления готовой ВКР / Deadline for final edition of the thesis 15.05.2023

Характеристика темы ВКР / Description of thesis subject (topic)

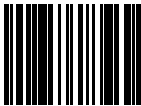
Тема в области фундаментальных исследований / Subject of fundamental research: нет / not

Тема в области прикладных исследований / Subject of applied research: да / yes

СОГЛАСОВАНО / AGREED:

Руководитель ВКР/
Thesis supervisor

Документ подписан	
----------------------	--

	
Аксенов Виталий Евгеньевич	
16.05.2023	

(эл. подпись)

Аксенов
Виталий
Евгеньевич

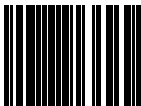
Задание принял к
исполнению/ Objectives
assumed BY

Документ подписан	
Чернацкий Евгений Геннадьевич	
16.05.2023	

(эл. подпись)

Чернацкий
Евгений
Геннадьевич

Руководитель ОП/ Head
of educational program

Документ подписан	
Станкевич Андрей Сергеевич	
22.05.2023	

(эл. подпись)

Станкевич
Андрей
Сергеевич

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS**

Обучающийся / Student Чернацкий Евгений Геннадьевич
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Верификация алгоритма консенсуса в базе данных ВК и тестирование кода на императивном языке программирования на графе состояний TLA+
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS**

Цель исследования / Research goal

Разработка инструмента тестирования распределенных систем на основе модели, полученной из спецификации, написанной на языке TLA+, для последующего применения при тестировании базы данных ВКонтакте GMS.

Задачи, решаемые в ВКР / Research tasks

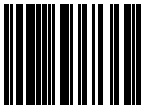
1. Написание и верификация спецификации TLA+ для алгоритма репликации, используемого в базе данных ВКонтакте GMS. 2. Реализация эффективного инструмента тестирования на основе модели, полученной из спецификации TLA+. 3. Применение реализованного инструмента для тестирования кода базы данных ВКонтакте GMS.

Краткая характеристика полученных результатов / Short summary of results/findings

Была изучена предметная область, написана и верифицирована спецификация TLA+ для алгоритма репликации Viewstamped Replication, используемого в базе данных ВКонтакте GMS, а также реализован инструмент для генерации тестов на основе модели TLA+, который впоследствии был применен для тестирования реализации базы данных ВКонтакте.

Обучающийся/Student

Документ подписан	
----------------------	--

	
Чернацкий Евгений Геннадьевич	
23.05.2023	

(эл. подпись/ signature)

Чернацкий
Евгений
Геннадьевич

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
17.05.2023	

(эл. подпись/ signature)

Аксенов
Виталий
Евгеньевич

(Фамилия И.О./ name
and surname)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
1. Обзор предметной области	8
1.1. Алгоритм репликации Viewstamped Replication	8
1.2. Язык спецификаций TLA^+	10
1.3. Тестирование на основе модели	12
1.4. Существующие решения	14
1.4.1. Modelator	14
1.4.2. Kayfabe	14
1.5. Постановка задачи	15
Выводы по главе 1	16
2. Теоретическое исследование	17
2.1. Спецификация алгоритма Viewstamped Replication	17
2.1.1. Описание состояния системы	17
2.1.2. Описание переходов системы	19
2.1.3. Описание свойств и инвариантов системы	22
2.2. Архитектура инструмента тестирования на основе модели	23
2.2.1. Сериализация состояния модели TLA^+	24
2.2.2. Извлечение графа состояний из TLC	26
2.2.3. Покрытие путями графа состояний	27
2.2.4. Абстрактный тест	28
2.3. Эффективное решение задачи покрытия путями графа	29
2.3.1. Наивный алгоритм	30
2.3.2. Оптимальный алгоритм	30
Выводы по главе 2	35
3. Практическое исследование	36
3.1. Результаты проверки модели TLC	36
3.2. Результаты решения задачи покрытия графа путями	36
3.3. Тестирование инструмента генерации тестов на основе модели ..	37
3.4. Особенности реализации абстрактного теста для алгоритма GMS	38
Выводы по главе 3	39
ЗАКЛЮЧЕНИЕ	40
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	41

ПРИЛОЖЕНИЕ А. Запуск разработанного инструмента на примере принципа Дирихле	42
---	----

ВВЕДЕНИЕ

Распределенные системы очень сложно разрабатывать. Даже если агентов всего несколько, число различных состояний системы может превышать тысячи, а то и миллионы. На таких масштабах модульное тестирование бессмысленно, а число крайних случаев, которые могут возникнуть в случае некорректной реализации системы, вполне способно выйти за практические рамки.

Одним из решением данной проблемы является применение *формальных методов*. В этой работе затрагивается язык формальной спецификации TLA^+ [8], используемый для разработки, моделирования и верификации программ, в частности — распределенных систем. Разработчику достаточно написать спецификацию своей системы, задать свойства и инварианты, которые система должна соблюдать, а затем воспользоваться *инструментом проверки модели* (англ. model checker) для проверки свойств и инвариантов на истинность.

Однако формальные методы доказывают выполнение свойств и инвариантов некоторой *модели* системы, а не ее конкретной реализации. В первую очередь язык спецификаций TLA^+ позиционируется как инструмент математического задания систем и ловли ошибок еще на этапе проектирования — конкретная реализация системы на каком-либо языке программирования все еще может являться ошибочной.

В большой степени побороться с этой проблемой позволяет *тестирование на основе модели* [6], проверяющее соответствие функциональности программы желаемому поведению, описанному с помощью некоторой модели. В случае этой работы поведение описывается с помощью спецификации TLA^+ , а модель задается графом состояний, который можно получить с помощью инструмента проверки модели TLC [10]. Каждое ребро в графе состояний соответствует одному из действий системы, описанных в спецификации TLA^+ . Таким образом, путям в графе состояний соответствуют цепочки действий — исполнения, что предоставляет простой и мощный способ генерации тестов, учитывающий как самые разные крайние случаи, так и обычные исполнения.

Команда ВКонтакте задалась вопросом корректности своей новой распределенной базы данных GMS, основывающейся на алгоритме репликации данных Viewstamped Replication [4]. В ходе этой работы была написана спе-

цификация TLA⁺ для алгоритма репликации Viewstamped Replication, а также разработан инструмент, тестирующий корректность алгоритма, использующегося в GMS и реализованного на языке программирования Go, относительно модели, порожденной спецификацией TLA⁺.

В первой главе будет проведен обзор предметной области, разобраны основные понятия и концепции, используемые в работе.

Во второй главе будет подробнее описана архитектура разрабатываемого решения, а также алгоритмы, используемые в работе.

В третьей главе будет рассматриваться результаты выполнения работы, включающие применение разработанного решения на практике.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Алгоритм репликации Viewstamped Replication

Объектом исследования данной дипломной работы является база данных GMS, разработанная и реализованная командой ВКонтакте. Приведенная база данных является распределенной по нескольким причинам:

- расположение данных географически близко к пользователям для минимизации задержки ответа;
- обеспечение работы системы даже при аварийном завершении работы нескольких узлов;
- масштабирование числа узлов в системе для распределения запросов на чтение.

Синхронизация состояния между узлами и выполнение требований выше достигается с помощью *алгоритмов репликации*, например Raft [5], Paxos [3] и Viewstamped Replication [4].

За основу распределенной базы данных GMS брался алгоритм Viewstamped Replication (VR). Авторы VR предлагают распределенный алгоритм с $2 \cdot f + 1$ узлами, синхронизирующий состояние между несколькими узлами, а также являющийся отказоустойчивым до тех пор, пока аварийно не прекратят работу более чем f узлов.

Алгоритм VR использует узел, являющийся *лидером*, для принятия клиентских запросов. При этом второстепенные узлы получают клиентские запросы от лидера, причем в одинаковом порядке. В случае аварийного прекращения работы текущего лидера роль лидера переходит к другому узлу. Передача роли лидера достигается с помощью увеличения специального числа `view-number` у каждого узла. Для каждого из возможных значений `view-number` один из узлов объявляется лидером: по умолчанию, это узел с номером `view-number % n`, где n — число узлов в системе. Таким образом, узлы постоянно проверяют текущего лидера на факт прекращения работы, и в случае аварийной ситуации — перевыбирают текущего лидера с помощью протокола `view change`.

Чтобы алгоритм VR оставался корректным после изменения лидера, состояние системы после выполнения протокола `view change` должно сохранять все выполненные клиентские операции и их порядок до текущего момента. Это достигается тем, что лидер перед выполнением клиентского запроса

ждет пока запрос не станет известен как минимум $f + 1$ узлам, а при исполнении протокола `view change` состояние нового лидера вычисляется из состояний $f + 1$ узлов. Таким образом, каждый выполненный клиентский запрос известен кворуму узлов, а протокол `view change` завершается тем, что кворум узлов обладает новым общим значением `view-number`.

Состояние одного из узлов в VR состоит из следующих переменных:

- текущий протокол `status`, исполняющийся узлом (обычный или `view change`);
- число `view-number`, обозначающее текущего лидера с точки зрения узла;
- массив `log`, описывающий журнал, содержащий полученные от клиентов запросы в определенном порядке;
- число `commit-number` уже исполненных запросов из массива `log`.

Алгоритм VR также предоставляет два способа восстановления состояния узла в случае его аварийного завершения работы:

- запись состояния узла на некоторый внешний носитель перед каждой отправкой сообщения;
- использование специального протокола восстановления, работающего независимо от наличия внешнего носителя.

Распределенная база данных GMS использует в своей реализации первый подход, вызывая системный вызов `fsync` перед каждой отправкой сообщения. Тогда после аварийного перезапуска узел сможет вернуться к своему прежнему состоянию, считав записанные на внешний носитель данные. В этом смысле аварийное завершение работы узла ничем не отличается от случая, в котором узел на некоторое время изолировался от других узлов и перестал отправлять и принимать сообщения.

Стоит отметить, что алгоритм репликации, реализованный в GMS, содержит несколько модификаций по сравнению с описанием алгоритма репликации VR, приведенного в соответствующей статье.

Во-первых, при исполнении узлами протокола `view change` не происходит передачи всего журнала, так как журнал узлов способен неограниченно расти в размерах. Вместо этого используется специальный *протокол скачивания*, позволяющий узлам передавать части своего журнала другим узлам. При этом, если при скачивании был обнаружен факт расхождения истории

журналов, то происходит удаление некорректного хвоста записей журнала и замещение его новыми записями. Дополнительно, если во время скачивания журнала возникает необходимость в выборе нового лидера с помощью протокола `view change`, то скачивание останавливается до выбора нового лидера, а затем начинается заново уже с нового узла.

Во-вторых, журнал узлов наряду с клиентскими запросами содержит специальные *view блоки*, содержащие значение `view-number` узла и добавляющиеся в журнал после успешного выбора нового лидера системы. Наличие `view` блоков упрощает реализацию протокола `view change` и, в частности, позволяет избавиться от передачи в сообщениях значения, равного `view-number` перед последним участием узла в протоколе `view change`.

1.2. Язык спецификаций TLA⁺

Формальные методы — различные математические техники, используемые для проектирования и верификации программ. При использовании формальных методов разработчик сначала математически описывает свое решение на некотором формальном языке — создает *спецификацию* решения, далее использует *инструмент проверки модели* для верификации корректности спецификации, а затем на ее основе реализует решение, например, на каком-либо языке программирования.

Существует множество методов и инструментов для формального описания и верификации программ. В данной дипломной работе был выбран TLA⁺ [8] — высокоуровневый язык спецификации для моделирования программ, в частности параллельных систем и распределенных систем. Спецификация на языке TLA⁺ представляет собой описание системы на языке математики и логики. В TLA⁺ базовая теория множеств позволяет накладывать ограничения на переменные системы, а темпоральная логика — задавать свойства и инварианты, зависящие от времени (например, что-то выполнено всегда или что-то рано или поздно произойдет). Состояние специфицируемой системы описывается с помощью переменных, записанных после выражения `VARIABLES`. Начальное состояние системы задается с помощью функции `Init`, содержащей для каждой переменной системы ее начальное значение. Переходы в системе задаются с помощью функции `Next`, определяющее следующее состояние системы в зависимости от предыдущего состояния системы (например, $a' = a + 1$ означает, что следующее значение a будет

на 1 больше чем предыдущее). При этом с помощью дизъюнкции и квантора существования можно определить несколько возможных переходов из одного состояния. Таким образом, совокупность `Init` и `Next` задает спецификацию некоторой системы на языке TLA^+ .

Листинг 1 – Пример спецификации TLA^+ . Задача о переливаниях.

```

----- MODULE DieHard -----
EXTENDS Integers , TLC

VARIABLES small , big

TypeOK == small \in 0..3 /\ big \in 0..5

FillSmall == small' = 3 /\ big' = big

FillBig == big' = 5 /\ small' = small

EmptySmall == small' = 0 /\ big' = big

EmptyBig == big' = 0 /\ small' = small

SmallToBig == IF big + small <= 5
                THEN big' = big + small /\ small' = 0
                ELSE big' = 5 /\ small' = small - (5 - big)

BigToSmall == IF big + small <= 3
                THEN big' = 0 /\ small' = big + small
                ELSE big' = small - (3 - big) /\ small' = 3

Next == FillSmall \/ FillBig \/ EmptySmall
        \/ EmptyBig \/ SmallToBig \/ BigToSmall

Init == big = 0 /\ small = 0

Spec == Init /\ [][Next]_<<small , big>>
=====

```

Инструмент проверки модели TLC [10] позволяет проверить спецификацию TLA^+ на корректность. При запуске инструмент строит модель системы, представляемую в виде ориентированного графа. Каждой вершине в графе соответствует состояние TLA^+ , содержащее конкретные значения переменных, а каждому ребру — одно из действий, описанных в спецификации. Если в спецификации TLA^+ присутствуют свойства и инварианты, то TLC также проверяет их истинность. Например, в случае алгоритма VR таким инвариантом

являлось бы утверждение, что все пары узлов в системе имеют одинаковый префикс в массиве `log` вплоть до минимального из двух `commit-number`.

Стоит отметить, что так как TLC явно строит граф состояний, а затем совершает его полный обход, то проверка на корректность возможна только для конечных моделей. Это критично для верификации алгоритма VR, так как длина журнала в узлах системы может неограниченно расти, а значит граф состояний модели имеет бесконечный размер. Чтобы избежать этой проблемы и получить возможность использовать инструмент TLC для верификации, требуется ограничить возможные значения переменных, в частности максимальную длину журнала и максимальное значение `view-number`.

1.3. Тестирование на основе модели

Хоть формальные методы и помогают разрабатывать программы без ошибок, они делают это на уровне спецификации. При этом корректность конкретной реализации системы на каком-либо языке программирования все еще остается без доказательства, ведь при написании кода все еще могла возникнуть ошибка, вызванная человеческим фактором. Таким образом, по корректности модели нельзя утверждать о корректности реализации системы.

Привычные методы тестирования никак не решают данную проблему:

- *модульное тестирование* покрывает незначительную часть возможных исполнений программы и в основном используется для тестирования крайних случаев, число которых в распределенных системах выходит за практические рамки;
- *фаззинг* генерирует случайные исполнения программы: многие крайние случаи при этом не покрываются.

Проблему можно решить с помощью *тестирования на основе модели* [6] — техники тестирования, в которой требуемое поведение системы описывается некоторой моделью. При этом модель может как отображать желаемое поведение системы, так и использоваться для создания тестов. Используемые модели, как и сгенерированные из них тесты, являются абстрактными и описывают необходимую часть поведения системы. Чтобы получить тест, который можно запускать, требуется выполнить реализацию абстрактного теста, сопоставив действия модели и действия реализации системы. Таким образом, тестирование на основе модели является видом функционального тестирова-

ния — проверяется функциональность системы, а не ее внутренняя реализация.

Часто модель, описывающая систему, задается в виде некоторого графа состояний, описывающего все возможные состояния системы и переходы между состояниями. Так как любому пути в этом графе соответствует некоторое исполнение системы, то в качестве генерируемых тестов могут служить различные пути в графе.

Инструмент проверки модели TLC в ходе проверки спецификации TLA^+ на корректность генерирует конечный граф состояний, который можно использовать для генерации тестов. Предположим, что каждому переходу из спецификации TLA^+ уже сопоставлены соответствующие действия в реализации системы, а также реализована функция $f(s)$, проецирующая состояние реализации системы в состояние модели. Тогда, используя некоторый путь в графе состояний модели, начинающемся в стартовой вершине, можно протестировать реализацию системы следующим образом: при переходе по очередному ребру в пути вызвать соответствующую функцию реализации системы, и после очередного перехода проверять значения переменных модели в текущей вершине пути и $f(s)$ на равенство.

У приведенного метода тестирования можно выделить следующие достоинства:

- детерминированный метод тестирования высококонкурентного кода;
- автоматическая генерация тестовых исполнений — отсутствие необходимости в написании тестовых исполнений вручную;
- исчерпывающее тестирование системы с точностью до заданной модели;

При этом данный метод тестирования также обладает следующими недостатками:

- низкая эффективность тестирования при большом размере модели;
- генерируемые исполнения не содержат некорректных действий — тестирование происходит только на «хороших» исполнениях;
- сложность написания теста в случае, если реализация системы плохо соответствует модели.

Именно этот метод тестирования исследуется и реализуется в этой дипломной работе.

1.4. Существующие решения

Существует множество инструментов автоматической генерации тестов на основе модели, однако если рассматривать генерацию тестов именно по модели TLA^+ , то существуют лишь два представителя: Modelator [9] и Kayfabe [7].

1.4.1. Modelator

Modelator — инструмент автоматической генерации тестов на основе модели. Modelator в качестве входа принимает спецификацию, написанную на TLA^+ , и затем с помощью Apalache — другого инструмента проверки модели — генерирует тесты, которые могут быть запущены на реализации этой спецификации. Несмотря на это, это решение имеет ряд ограничений:

- Apalache не производит построения полного графа состояний, а лишь генерирует случайные исполнения в рамках модели;
- Apalache не поддерживает приличную часть синтаксиса TLA^+ ;
- поддерживает тестирование кода только на языке программирования Python — ранее дополнительно поддерживались языки Rust и Go, но их поддержка прекратилась.

Таким образом, генерация тестов инструментом Modelator очень напоминает фаззинг. Это кардинально отличается от идеи, рассматриваемой в этой дипломной работе: генерации всего графа состояний модели и последующей генерации исполнений, полностью покрывающих этот граф.

1.4.2. Kayfabe

Kayfabe — еще один инструмент для тестирования на основе модели. Kayfabe на вход получает спецификацию TLA^+ и граф состояний в формате *.dot, сгенерированный с помощью инструмента TLC. Далее инструментом генерируется такой обход графа состояний, что каждое состояние было посещено хотя бы один раз. При этом допускается из любого состояния произвести «сброс системы» — возвратиться в стартовое состояние и начать новое исполнение. Это решение также имеет ограничения:

- не поддерживаются действия, имеющие ненулевое число аргументов;
- сводится к задаче о коммивояжере, хотя существует более оптимальное решение, использующее поиск циркуляции минимальной стоимости;
- поддерживает тестирование кода только на языке программирования C#;

— отсутствует в свободном доступе — существует только в виде статьи.

Первое ограничение является довольно критичным в случае алгоритма репликации VR — когда в системе происходит какое-либо действие, необходимо знать узел, над которым это действие совершалось. Kayfabe пользуется возможностью TLC экспорта графа состояний в формат `*.dot`, который игнорирует аргументы действий модели при записи переходов — таким образом теряется информация о том, над каким узлом системы произошло действие.

Дополнительно, Kayfabe генерирует тестовые исполнения, решая задачу о коммивояжере — на основе графа состояний строится реберный граф, из которого строится полный граф, в котором между каждой парой вершин появляется ребро с весом, равным расстоянию между двумя вершинами в графе состояний. Таким образом, если на приведенном полном графе решить задачу коммивояжера, то получится обход, посещающий все переходы графа состояний как минимум один раз. К сожалению, задача о коммивояжере является NP-трудной задачей, а значит не имеет эффективного решения. В Kayfabe используется $3/2$ -аппроксимирующий алгоритм Кристофидеса [2] для приближенного решения задачи о коммивояжере, и, как утверждают авторы инструмента, алгоритм работает за разумное время на графах состояний с числом вершин порядка 10^3 . В разделе 2.3 рассматривается эффективный полиномиальный алгоритм, решающий задачу покрытия графа состояний минимальным по размеру множеством путей, начинающихся в стартовой вершине.

Таким образом, идея тестирования в Kayfabe во многом схожа с идеей этой дипломной работы. К сожалению, инструмент Kayfabe существует только в виде статьи, поэтому прямое сравнение результатов не представляется возможным.

1.5. Постановка задачи

Целью этой дипломной работы является разработка инструмента тестирования распределенных систем на основе модели, написанной на языке спецификаций TLA^+ , для последующего применения при тестировании базы данных ВКонтакте.

В работе решаются следующие задачи:

- написание и верификация спецификации TLA^+ для алгоритма репликации, используемого в базе данных Вконтакте;

- реализация эффективного инструмента для генерации тестов на основе модели TLA^+ ;
- применение реализованного инструмента для тестирования кода базы данных ВКонтакте.

Выводы по главе 1

В данной главе был проведен обзор предметной области дипломной работы. В частности, была описана идея алгоритма репликации Viewstamped Replication вместе с его модификациями, используемыми в распределенной базе данных ВКонтакте GMS, были введены язык спецификации TLA^+ и инструмент проверки модели TLC, была описана техника тестирования на основе модели и идея генерации тестов для программ, написанных на некотором языке программирования, по графу состояний TLC. Также проведено сравнение с существующими решениями генерации тестов на основе модели TLA^+ , описаны преимущества и недостатки рассматриваемой идеи, сформулированы цели и задачи работы.

ГЛАВА 2. ТЕОРЕТИЧЕСКОЕ ИССЛЕДОВАНИЕ

2.1. Спецификация алгоритма Viewstamped Replication

Первой задачей данной дипломной работы является написание спецификации для алгоритма репликации Viewstamped Replication.

2.1.1. Описание состояния системы

Листинг 2 – Описание состояния алгоритма VR в спецификации TLA⁺.

```

CONSTANTS ReplicasNumber , MaxLogQueryCount , MaxViewNumber

VARIABLES replicaState , queriesCount

QueryUID == 1 .. (MaxViewNumber + 1) * MaxLogQueryCount

ViewNumber == 0 .. MaxViewNumber

LogIndex == 0 .. MaxLogQueryCount

LogEntry == [ type: { "QueryBlock" }, query: QueryUID ]
             \cup [ type: { "ViewBlock" }, viewNumber: ViewNumber ]

Log == UNION { [ 1 .. n -> LogEntry ] : n \in LogIndex }

ReplicaIndex == 0 .. ReplicasNumber - 1

ReplicaState == [
    status: { "Normal", "ViewChange" },
    log: Log,
    viewNumber: ViewNumber,
    commitNumber: LogIndex,
    downloadingFrom: ReplicaIndex,
    receivedStartViewChangeQuorum: BOOLEAN
]

TypeOK ==
    /\ replicaState \in [ ReplicaIndex -> ReplicaState ]
    /\ queriesCount \in 0 .. (MaxViewNumber + 1) *
       MaxLogQueryCount

```

Состояние системы в спецификации TLA⁺ описывается с помощью переменных, имена которых записываются после ключевого слова VARIABLES. Спецификация алгоритма VR содержит следующие переменные:

- массив `replicaState`, содержащий записи, хранящие состояние каждого узла в системе;

- строка `status`, содержащая текущий протокол — `''Normal''` или `''ViewChange''`, — которому следует узел;
 - массив `log`, содержащий журнал узла, хранящий клиентские запросы (тип `''QueryBlock''`) и view блоки (тип `''ViewBlock''`);
 - число `viewNumber`, значение которого равно `view-number` узла из алгоритма VR;
 - число `commitNumber`, значение которого равно `commit-number` узла из алгоритма VR;
 - число `downloadingFrom`, значение которого равно номеру узла, с которого происходит скачивание журнала (может отсутствовать);
 - логическое значение `receivedStartViewChangeQuorum`, которое истинно в том случае, если текущий `status` узла равен `''ViewChange''` и узел принял сообщение `StartViewChange` от кворума узлов;
- число `queriesCount`, равное числу различных клиентских запросов, которые поступили в систему на данный момент.

Стоит отметить, что за переменными в TLA^+ не зафиксирован какой-либо тип — переменные могут принимать значение любого типа. Чтобы наложить ограничения на возможные значения переменных в TLA^+ часто используется специальный инвариант, именуемый `TypeOK`. Инвариант содержит теоретико-множественные утверждения про переменные спецификации, в частности, каким множествам должны принадлежать значения переменных.

Дополнительно, спецификация содержит три постоянные, записываемые в TLA^+ после ключевого слова `CONSTANTS`:

- число `ReplicasNumber`, равное числу узлов в системе;
- число `MaxLogQueryCount`, равное максимальному возможному числу элементов `log` типа `''QueryBlock''` у каждого узла системы;
- число `MaxViewNumber`, равное максимальному возможному значению `viewNumber` у каждого узла системы.

Наличие последних двух постоянных необходимо для конечности пространства состояний модели, так как иначе размер журнала `log` и число `viewNumber` узлов смогут расти неограниченно — конечность числа состояний важна для корректной работы инструмента проверки модели TLC. Зна-

чения постоянных задаются в стороннем файле конфигурации TLA^+ , необходимым для работы инструмента TLC. Описание постоянных, переменных и инварианта `TypeOK` в коде TLA^+ приведено на листинге 2.

Также стоит объяснить отсутствие в спецификации значения, соответствующего аварийному завершению работы узла. Так как команда ВКонтакте после каждого действия над узлом записывает текущее состояние узла на внешний носитель, то при аварийном завершении и последующем восстановлении узел будет иметь прежнее состояние. Таким образом, с точки зрения других узлов аварийное завершение узла неотличимо от ситуации, когда все сообщения от и до узла теряются в течение некоторого периода времени.

Значения переменных в начальном состоянии системы задаются с помощью оператора `Init` (листинг 3). В случае алгоритма VR, после его инициализации все узлы имеют `status` равный `Normal`, пустой журнал, и остальные числовые поля равные нулю, а переменная `queriesCount` также имеет нулевое значение.

2.1.2. Описание переходов системы

Листинг 3 – Спецификация алгоритма VR в TLA^+ .

```
Init ==
  /\ replicaState = [r \in ReplicaIndex |-> [
    status |-> "Normal",
    log |-> <<>>,
    viewNumber |-> 0,
    commitNumber |-> 0,
    downloadingFrom |-> r,
    receivedStartViewChangeQuorum |-> FALSE
  ]]
  /\ queriesCount = 0

Next ==
  \E r \in ReplicaIndex :
    \/ NormalOperation(r)
    \/ ViewChangeOperation(r)

Spec == Init /\ [][Next]_vars /\ WF_vars(Next)
```

Возможные переходы в спецификации TLA^+ задаются с помощью оператора `Next`, в котором, возможно с некоторым уровнем вложенности, определяются все возможные действия над системой (листинг 3).

В ходе работы алгоритма репликации VR узлы общаются между собой с помощью сообщений. При этом любое отправленное сообщение имеет возможность потеряться и не дойти до получателя. В спецификации алгоритма эта особенность реализуется с помощью существования отдельных переходов, соответствующих отправке и принятию сообщения. Таким образом, потеря сообщения в модели задается исполнением, в котором действие отправки произошло, а действие получения — нет.

Стоит отметить, что сеть, по которой отправляются сообщения, никак не моделируется — из состояний всех узлов системы возможно однозначно понять, какие сообщения *могли* быть отправлены другими узлами, и выполнить соответствующее действия получения сообщения.

Спецификация алгоритма VR содержит переходы, соответствующие исполнению обычного протокола (переход `NormalOperation`) и исполнению протокола `view change` (переход `ViewChangeOperation`). Переход `NormalOperation` состоит из следующих действий:

- `ReceiveClientQuery` — получение лидером клиентского запроса, добавление его в журнал и отправка сообщения `Prepare` второстепенным узлам;
- `ReceivePrepare` — получение второстепенным узлом сообщения `Prepare`, добавление запроса в журнал с возможным увеличением `commitNumber` и отправка сообщения `PrepareOK` лидеру;
- `ReceivePrepareOKQuorum` — получение лидером сообщения `PrepareOK` от кворума узлов в системе и увеличение значения `commitNumber`.

Переход `ViewChangeOperation`, в свою очередь, состоит из следующих действий:

- `TimeoutStartViewChange` — обнаружение узлом необходимости смены лидера, изменение `status` на `''ViewChange''`, увеличение значения `viewNumber`, сброс значений `downloadingFrom` и `receivedStartViewChangeQuorum`, а также отправка сообщения `StartViewChange` остальным узлам;
- `ReceiveStartViewChangeQuorum` — получение узлом сообщения `StartViewChange` от кворума узлов в системе, изменение

- `receivedStartViewChangeQuorum` на значение `TRUE` и отправка сообщения `DoViewChange` новому лидеру;
- `ReceiveDoViewChangeQuorum` — получение новым лидером сообщения `DoViewChange` от кворума узлов, выбор лучшего журнала среди кворума узлов, изменение `status` на `''Normal''`, замена `downloadingFrom` номером узла с лучшим журналом и отправка этому узлу сообщения `StartDownload`;
 - `ReceiveStartDownload` — получение узлом сообщения `StartDownload` и отправка в ответ по одному сообщению `DownloadChunk` на каждый недостающий элемент журнала;
 - `ReceiveDownloadChunkLeader` — получение новым лидером сообщения `DownloadChunk`, добавление в журнал недостающего элемента и отправка сообщения `StartView` второстепенным узлам в случае получения последнего сообщения `DownloadChunk`;
 - `ReceiveStartView` — получение второстепенным узлом сообщения `StartView`, изменение `status` на `''Normal''`, замена `downloadingFrom` номером текущего лидера и отправка лидеру сообщения `StartDownload`;
 - `ReceiveDownloadChunkSecondary` — получение второстепенным узлом сообщения `DownloadChunk`, добавление в журнал недостающего элемента и, в случае получения последнего сообщения `DownloadChunk`, сброс значения `downloadingFrom`.

Листинг 4 – Пример описания одного из переходов алгоритма VR в спецификации TLA^+ .

```
TimeoutStartViewChange(replica) ==
  /\ GetViewNumber(replica) < MaxViewNumber
  /\ replicaState' = [replicaState EXCEPT
    ! [replica].status = "ViewChange",
    ! [replica].viewNumber = @ + 1,
    ! [replica].downloadingFrom = replica,
    ! [replica].receivedStartViewChangeQuorum = FALSE]
  /\ UNCHANGED <<queriesCount>>
```

Только описания `Init` и `Next` недостаточно для написания спецификации алгоритма VR и последующей верификации — TLA^+ по умолчанию допускает такие исполнения, в которых некоторые переходы являются пустыми,

то есть никак не изменяют состояние системы (англ. *stuttering*). Таким образом, спецификацией разрешаются исполнения, в которых начиная с некоторого момента состояние системы никогда не изменяется. Чтобы избежать этой проблемы, достаточно в спецификацию добавить условие *справедливости* [1] (англ. *fairness*). В частности, условие *слабой* справедливости позволяет исключить такие исполнения, в которых некоторое действие (в нашем случае *Next*) является постоянно допустимым для бесконечного числа подряд идущих состояний, но при этом ни разу не происходит в виде перехода. В TLA^+ условие слабой справедливости можно записать в виде $WF_vars(Next)$, и таким образом получить спецификацию (листинг 3), содержащую исключительно интересные исполнения.

2.1.3. Описание свойств и инвариантов системы

Чтобы успешно верифицировать спецификацию алгоритма, необходимо иметь что-то, что позволит проверить ее корректность. При наличии только описания состояния и переходов генерируются исполнения, но никакие факты про них не проверяются. В TLA^+ корректность проверяется с помощью свойств и инвариантов.

Листинг 5 – Описание инвариантов алгоритма VR в спецификации TLA^+ .

```
PrefixLogConsistency ==
  \A i, j \in ReplicaIndex :
    \/ i = j
    \/ IsPrefixOf(
      SubSeq(GetLog(i), 1, GetCommitNumber(i)),
      SubSeq(GetLog(j), 1, GetCommitNumber(j)))
    \/ IsPrefixOf(
      SubSeq(GetLog(j), 1, GetCommitNumber(j)),
      SubSeq(GetLog(i), 1, GetCommitNumber(i)))

LogCommitProgress ==
  []<> \A r \in ReplicaIndex :
    /\ GetStatus(r) = "Normal"
    /\ GetLogQueryCount(r) = MaxLogQueryCount
    /\ GetCommitNumber(r) = GetLogLength(r)
```

Алгоритм VR, как и любой алгоритм репликации, должен достигать синхронизации данных между узлами: если рассмотреть любую пару узлов, то исполненные запросы, записанные в их журналах, должны либо совпадать, либо одна последовательность запросов должна являться префиксом другого (один

из узлов отстал от другого). Это условие описано в спецификации TLA^+ с помощью инварианта `PrefixLogConsistency`.

Дополнительно, так как в спецификации длина журнала узлов ограничивается числом `MaxLogQueryCount`, то система рано или поздно должна приходить в одно и то же состояние, в котором все узлы имеют `status` равный `Normal`, число клиентских запросов в журнале `log` равно `MaxLogQueryCount`, и число `commitNumber` совпадающее с длиной журнала. Это условие описывается с помощью инварианта `LogCommitProgress`.

2.2. Архитектура инструмента тестирования на основе модели

Работу инструмента для генерации тестов по модели TLA^+ , описываемого в данной дипломной работе, можно охарактеризовать следующими этапами:

- а) Запуск TLC, которому на вход подаются спецификация и конфигурация TLA^+ .
- б) Конструирование инструментом TLC графа состояний и проверка корректности модели.
- в) Подача графа состояний на вход алгоритму покрытия графа путями, находящему минимальное по размеру множество путей, покрывающих все ребра графа, и последующий экспорт найденного покрытия графа в файл.
- г) Генерация абстрактного теста, запускающего тестируемую систему на найденном множестве путей (исполнений) и проверяющего согласованность состояний тестируемой системы, реализованной на языке программирования Go, и модели TLA^+ .

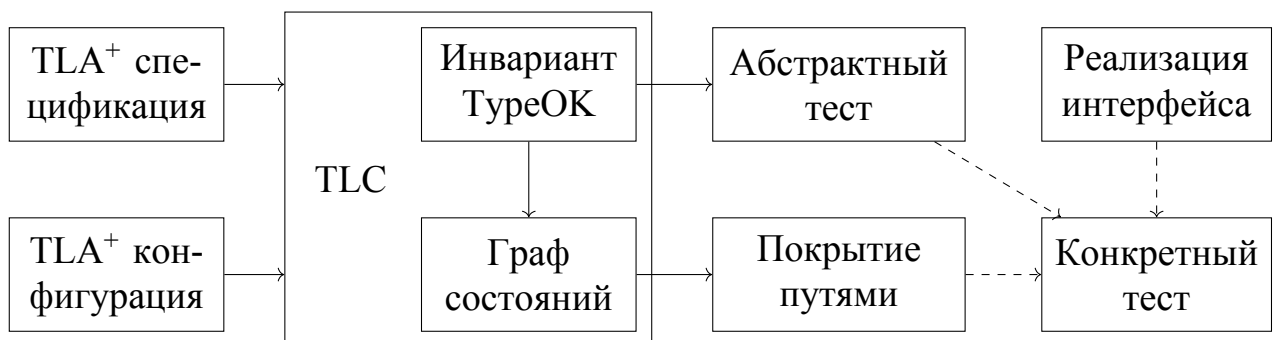


Рисунок 1 – Диаграмма, описывающая внутреннюю архитектуру инструмента генерации тестов по модели TLA^+ .

Для того, чтобы получить готовый тест, который можно запускать над реализацией системы, пользователю сначала необходимо запустить инструмент, подав ему на вход спецификацию и конфигурацию TLA^+ . После некоторого ожидания инструмент завершит свою работу и сгенерирует несколько файлов: файл, содержащий покрытие графа состояний путями, и файлы, содержащие исходный код на языке программирования Go, вместе образующие абстрактный тест. Чтобы иметь возможность запускать тест, требуется реализовать интерфейс абстрактного теста, содержащий метод `PerformAction(...)`, вызывающий соответствующие методы для каждого действия из спецификации TLA^+ , а также методы `Init()`, `Reset()` и `State()`, инициализирующий состояние системы, сбрасывающий состояние системы и отображающий состояние тестируемой системы в состояние модели TLA^+ соответственно. Последним шагом является непосредственная инициализация абстрактного теста с передачей ему реализации интерфейса — таким образом, получается конкретный тест, который можно запустить, вызвав метод `Test()`.

Стоит отметить, что полная регенерация теста требуется лишь при изменении спецификации TLA^+ : при изменении конфигурации TLA^+ изменяется лишь граф состояний и, соответственно, экспортируемый файл с множеством покрывающих путей, а файлы, содержащие исходный код абстрактного теста, никак не изменяются.

В следующих подразделах подробнее описываются отдельные компоненты инструмента, приведенные на рисунке 1.

2.2.1. Сериализация состояния модели TLA^+

Необходимость в сериализации состояния модели TLA^+ возникает в ходе генерации графа состояний. Дополнительно, абстрактный тест, генерируемый инструментом, должен уметь десериализовывать состояние модели для дальнейшего сравнения с возвращаемым значением метода `State()`, который характеризует текущее состояние тестируемой системы в терминах модели.

В качестве формата сериализации для решения данной задачи был выбран JSON. При этом стоит обратить внимание, что в простой реализации инструмента пользователю при реализации метода `State()` потребуется ручное построение JSON объекта, описывающего состояние модели. Этого

можно избежать с помощью кодогенерации структуры на языке Go, описывающей переменные состояния модели TLA^+ и их типы, а также с помощью использования встроенного в язык Go десериализатора в виде функции `json.Unmarshal`.

Как уже упоминалось в разделе 2.1.1, в языке спецификаций TLA^+ отсутствует понятие типа — вместо этого в спецификации задается особый инвариант `TypeOK`, который с помощью множеств и операции принадлежности `\in` задает ограничения на переменные модели. В общем случае, задача определения по множеству в языке TLA^+ соответствующего типа в императивном языке программирования является сложной. Несмотря на это, разработчики спецификаций часто записывают утверждения принадлежности, которые относительно просто транслируются в привычные языки программирования, например `value \in Nat` соответствует переменной `value uint32` в Go, а `intArray \in [1..n -> Int]` — переменной `intArray []int`.

Таблица 1 – Таблица соответствия множеств из языка TLA^+ и типов Go

TLA^+	Описание	Go
<code>Nat</code>	Множество натуральных чисел \mathbb{N}	<code>uint32</code>
<code>Int</code>	Множество целых чисел \mathbb{Z}	<code>int32</code>
<code>[A -> B]</code>	Множество функций из A в B	<code>map[A]B</code>
<code>[0..n -> A],</code> <code>[1..n -> A]</code>	Множество функций из (отрезка) натуральных чисел \mathbb{N} в A	<code>[]A</code>
<code>a..b</code>	Отрезок целых чисел \mathbb{Z}	<code>int32</code>
<code>[x: A, y: B,</code> <code>...]</code>	Множество записей (англ. record)	<code>type M</code> <code>struct { x</code> <code>A; y B; ...</code> <code>}</code>
<code>{"a", "b",</code> <code>...}</code>	Простое множество, заданное перечислением объектов	<code>type E</code> <code>int32; const</code> <code>(a E =</code> <code>iota; b; ...</code> <code>)</code>
<code>{x \in A:</code> <code>P(x)}</code>	Операция фильтрации множества $\{x \in A : P(x)\}$ (игнорируется)	<code>A</code>
<code>—</code>	Произвольное выражение (тип по умолчанию)	<code>interface{}</code>

Итого, требуется написать генератор структур на языке Go по описанию инварианта `TypeOK` в файле TLA^+ . При этом писать свой синтаксиче-

ский анализатор TLA^+ бессмысленно, так как TLC уже содержит внутри себя готовое решение — анализатор, созданный с помощью генератора синтаксических анализаторов JavaCC. Таким образом, достаточно разобрать абстрактное синтаксическое дерево, сгенерированное анализатором, и преобразовать его в структуру на языке Go. Соответствие конструкций языка TLA^+ и типов Go представлено в таблице 1.

2.2.2. Извлечение графа состояний из TLC

Инструмент проверки модели TLC проверяет описанные в спецификации TLA^+ свойства и инварианты, обходя все возможные состояния и исполнения системы. Само множество состояний и исполнений можно представить в виде графа состояний, где стартовое состояние порождается с помощью начального действия `Init`, а на переходах между состояниями написаны действия в модели, причем началу перехода соответствует состояние модели до действия, а концу перехода — состояние после.

Стоит отметить, что все возможные переходы системы задаются в TLA^+ с помощью одного действия `Next`, но при этом в автомате произвольное состояние может иметь несколько исходящих переходов с одинаковым действием, из-за чего возникает недетерминированность. Этой недетерминированности можно избежать, если `Next` — это дизъюнкция некоторых других действий, так как в этом случае можно однозначно понять, какое из действий в дизъюнкции было выполнено. То же самое можно сказать про квантор существования, если его переменная принимает конечное число различных значений — в этом случае квантор существования ничем не отличается от дизъюнкции.

Инструмент TLC уже представляет возможность экспорта графа состояний в особом формате `*.dot` — формате, используемом пакетом утилит для визуализации графов Graphviz. Однако эта возможность имеет несколько ограничений, которые не дают напрямую ей воспользоваться:

- действия, записанные на переходах графа состояний, экспортируются без своих аргументов;
- описание одного состояния графа состояний включает переменные модели и их значения, но оно состоит из единственной строки, имеющей синтаксис TLA^+ .

Таким образом, экспорт графа состояний в виде `*.dot` файла полезен исключительно для визуализации графа, но не для использования и обработки сторонним программным обеспечением. Требуется написать собственный экспортер, который будет лишен приведенных ограничений.

Первая проблема является недосмотром разработчиков TLC — в графе состояний, генерируемом TLC в ходе проверки модели на корректность, действия уже хранятся вместе со значениями своих аргументов, но при экспорте автомата в `*.dot` формат они игнорируются. Для решения второй проблемы достаточно сериализации состояния TLA^+ в формате JSON.

Обе проблемы невозможно решить, не корректируя исходный код TLC, написанный на языке программирования Java. Таким образом, в работе требуется написать модуль к TLC, расширяющий его функционал сериализацией состояния в JSON формате, генерацией структуры на языке Go, описывающей состояние модели, и экспортом графа состояний.

2.2.3. Покрытие путями графа состояний

В ходе тестирования требуется посетить все переходы в графе состояний и убедиться в согласованности системы и модели в каждом состоянии. При этом стоит учесть, что реальные распределенные системы, в частности GMS, содержат множество ограничений, не позволяющих применить для решения задачи простые алгоритмы:

- отсутствие поддержки отката последней выполненной операции — базы данных имеют необратимые запросы (например, удаление), что не позволяет применить простой обход в глубину;
- отсутствие поддержки клонирования состояния всей системы — возможна борьба за ресурсы, что не позволяет применить обход в глубину со стеком состояний;
- отсутствие поддержки инициализации системы не в стартовом состоянии модели — модель намного более абстрактная, чем реализация системы, и переменных модели недостаточно, чтобы полностью воспроизвести состояние реализации системы.

Дополнительно, граф состояний алгоритма репликации Viewstamped Replication является ациклическим, так как после каждой операции в системе один из параметров системы строго увеличивается: размер журнала, `commit-number` или `view-number` одного из узлов. Таким образом, при желании

вернуться и посетить другую ветвь графа состояний, необходимо начать новое исполнение из стартового состояния. Другими словами, требуется найти такое множество путей, соответствующих исполнениям системы, такое, что каждое ребро графа исполнений покрывается как минимум одним путем. При этом также имеет смысл минимизировать число путей в найденном множестве для уменьшения времени, занимаемым тестированием. В разделе 2.3 подробно описан алгоритм, решающий данную задачу, а также приведено доказательство его асимптотической сложности, равной $\mathcal{O}(D \cdot |E| + D^2 \cdot |V|)$, где D — диаметр графа состояний.

Полученное множество путей записывается в выходной JSON файл в массив `''executions''`. При этом каждое исполнение (путь) записывается в виде массива, представимого в виде $v_1 e_1 v_2 \dots v_n$, где v_i — вершина пути, а e_i — ребро пути. Так как суммарная длина всех путей, которыми покрывается граф состояний, может иметь порядок $D \cdot |E|$, то многие вершины графа будут встречаться в различных исполнениях значительно больше одного раза. Чтобы уменьшить размер генерируемого JSON файла, последний дополнительно содержит массив `''states''`, содержащий значения переменных для каждого возможного состояния. Эта оптимизация позволяет в исполнениях вместо непосредственного хранения сериализованных JSON объектов использовать индексы массива `''states''`.

2.2.4. Абстрактный тест

Основная задача генерируемого теста — проверить, что реализация системы на императивном языке программирования *согласуется* с моделью. Каждому действию в модели должно соответствовать некоторое действие в реализации системы, а также должен существовать способ получения значения переменных модели из реализации системы. Тогда, если для каждого перехода в модели соответствующее действие в реализации системы приводит в то же самое состояние, что и переход в модели, то можно говорить, что реализация системы согласуется с моделью.

При запуске абстрактного теста в самом начале вызывается метод `Init()`, инициализирующий стартовое состояние тестируемой системы. Далее для каждого шага (ребра) в текущем исполнении (пути) происходит следующее:

- Вызов метода `State()`, отображающего текущее состояние тестируемой системы в состояние модели, и последующее сравнение с ожидаемым состоянием модели.
- Вызов метода `PerformAction(...)`, который исполняет действие, записанное на следующем ребре пути, вместе с соответствующими аргументами, при условии существования следующего ребра.
- Переход к следующему шагу и повторение перечисленных действий.

Если в ходе исполнения приведенных выше действий был достигнут конец исполнения (пути), то у абстрактного теста вызывается метод `Reset()`, сбрасывающий состояние тестируемой системы в стартовое, и действия повторяются до тех пор, пока исполнения не закончатся.

2.3. Эффективное решение задачи покрытия путями графа

Сформулируем решаемую задачу на более формальном языке. Задан ориентированный ациклический мультиграф без петель $G = (V, E)$, обладающий следующим свойством: в нем присутствует единственная вершина, называемая стартовой вершиной, в которую не входит ни одно ребро, а также из этой вершины достижимы все вершины графа. Требуется найти минимальное по размеру множество простых путей P такое, что каждый путь начинается в стартовой вершине, и для каждого ребра в мультиграфе существует путь из P , который его покрывает.

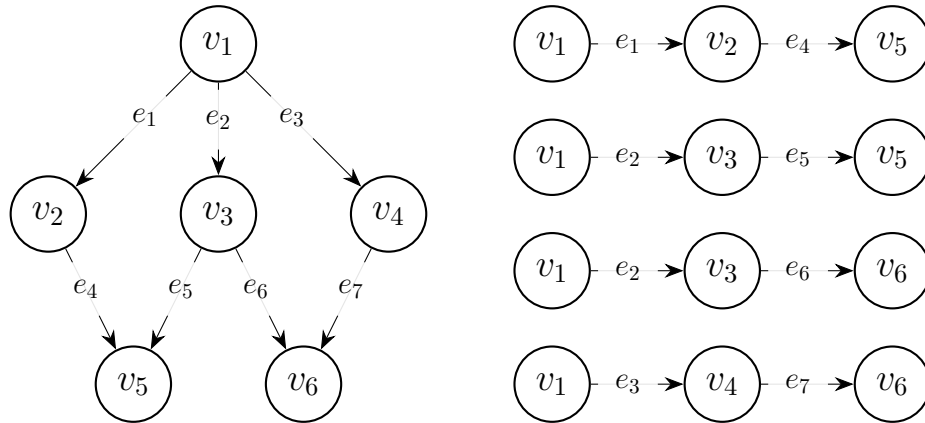


Рисунок 2 – Пример решения задачи покрытия путями: слева — граф G , справа — множество путей P .

Определение 1. Диаметр графа называется число D , равное расстоянию между наиболее удаленными друг от друга вершинами графа (при условии существования между ними пути). В контексте этой задачи диаметр графа

также равен максимальному расстоянию от стартовой вершины до остальных вершин в графе.

В практической части работы рассматриваются графы, имеющие малое значение диаметра $D \leq 30$ и большое число вершин $|V| \approx 10^6$. Учитывая эти факты, требуется разработать алгоритм, решающий поставленную задачу за разумное время для ограничения значений D и $|V|$.

2.3.1. Наивный алгоритм

Существует наивное решение приведенной задачи, использующее обход в глубину. Во время обхода графа хранится стек ребер, представляющий из себя простой путь из стартовой вершины до текущей. При обработке ранее не встречавшегося ребра в ответ добавляется весь путь из стека вместе с текущим ребром. Так как каждому ребру из графа соответствует ровно один путь из ответа, то число путей в ответе в точности равно $|E|$. Асимптотическая сложность приведенного алгоритма составляет $\mathcal{O}(D \cdot |E|)$, так как на каждое ребро в графе в ответ добавляется путь длины не более чем D .

Это решение можно незначительно улучшить, если заметить, что в ответе встречается очень много бесполезных путей, ребра которых уже покрываются другими путями. Это можно предотвратить, если добавлять в ответ путь только в том случае, когда обход в глубину заходит в тупик: очередная вершина уже была посещена обходом, либо у вершины отсутствуют исходящие ребра. В худшем случае асимптотическая сложность алгоритма все еще составляет $\mathcal{O}(D \cdot |E|)$.

К сожалению, оба решения не дают оптимального ответа на поставленную задачу: во многих графах минимальное число путей в разы меньше, чем получаемый ответ в приведенных выше алгоритмах.

2.3.2. Оптимальный алгоритм

Рассмотрим новый граф \check{G} , полученный из графа G добавлением нового ребра из каждой вершины графа, не являющейся стартовой, в стартовую. При таком преобразовании в графе появляются циклы, но можно заметить, что все простые циклы в получившемся графе пересекаются в стартовой вершине.

Рассмотрим множество простых путей, которое является ответом на задачу. Если к каждому пути из ответа добавить ребро, идущее из последней вершины пути в стартовую, то получатся простые циклы. Таким образом, если

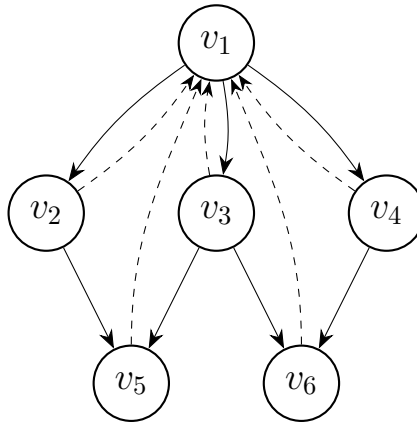


Рисунок 3 – Пример графа \check{G} . Добавленные ребра обозначены пунктирными стрелками.

ребра $E(G)$ в графе \check{G} удастся покрыть минимальным числом простых циклов, то ответ для графа G можно получить удалением из циклов ребра, идущего в стартовую вершину.

Пусть граф \check{G} является Эйлеровым. В этом случае ответом на задачу является Эйлеров цикл, разбитый на простые циклы. Так как все простые циклы в графе \check{G} пересекаются в стартовой вершине, число простых циклов, на который разбивается любой Эйлеров цикл, равно полустепени исхода стартовой вершины. Это число является минимальным, так как при любом меньшем значении найдется ребро, исходящее из стартовой вершины, не покрытое никаким циклом.

В случае графа \check{G} , не являющегося Эйлеровым, задачу покрытия простыми циклами можно решить сведением к задаче о поиске циркуляции минимальной стоимости в этом же графе со следующими ограничениями:

- Ребра $e \in E(G)$ имеют нижнюю границу на поток $l(e) = 1$, верхнюю границу на поток $u(e) = +\infty$ и стоимость единицы потока $c(e) = 0$.
- Оставшиеся (обратные) ребра $e \in E(\check{G}) \setminus E(G)$ имеют $l(e) = 0$, $u(e) = +\infty$ и $c(e) = 1$.

Таким образом значение потока $f(e)$ на каждом ребре будет равно числу простых циклов, проходящих через данное ребро. Наличие ограничения $l(e) = 1$ для ребер $e \in E(G)$ гарантирует покрытие каждого ребра как минимум одним из циклов (путей). Так как единственными ребрами, которые имеют ненулевую стоимость $c(e) = 1$, являются обратные ребра $e \in E(\check{G}) \setminus E(G)$, то поиск циркуляции минимальной стоимости будет соответствовать минимизации числа циклов (путей) в ответе на исходную задачу.

Стоит отметить, что из сведения к циркуляции минимальной стоимости напрямую возможно получить только число циклов в ответе, но не сами циклы. Чтобы получить простые циклы, составляющие ответ, необходимо каждое ребро $e \in E(\check{G})$ заменить на $f(e)$ кратных ребер, через каждое из которых возможно провести единичный поток. После данного преобразования каждое ребро покрывается ровно один раз, а значит полученный граф является Эйлеровым.

Задача о поиске циркуляции минимальной стоимости сводится к задаче о поиске максимального потока минимальной стоимости. Пусть $\delta_+^G(v)$ и $\delta_-^G(v)$ — полустепень исхода и полустепень захода вершины $v \in V(G)$ соответственно. При данном сведении, к графу \check{G} добавляются сток s и исток t , а также следующие ребра:

- Ребро $e = (s, v)$ с пропускной способностью $c(e) = \delta_-^G(v) - \delta_+^G(v)$, если $\delta_+^G(v) < \delta_-^G(v)$.
- Ребро $e = (v, t)$ с пропускной способностью $c(e) = \delta_+^G(v) - \delta_-^G(v)$, если $\delta_+^G(v) > \delta_-^G(v)$.

Назовем построенную сеть G^* . После нахождения максимального потока минимальной стоимости, потоку по ребру $e \in E(\check{G})$ до сведения будет соответствовать значение $f(e) + l(e)$, где $f(e)$ — поток по ребру после сведения.

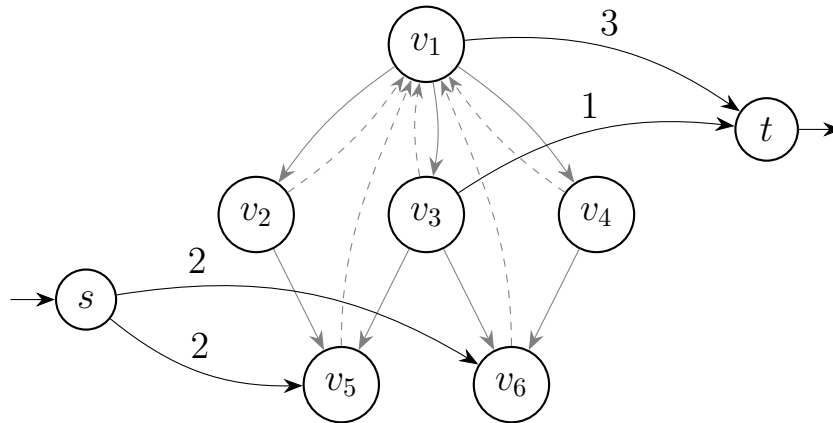


Рисунок 4 – Пример сети G^* . У ребер в сети показаны только конечные пропускные способности.

Одно из решений задачи максимального потока минимальной стоимости заключается в поиске максимального потока в сети, а затем в удалении циклов отрицательной стоимости в остаточной сети. Далее приведены несколько теорем, утверждающих об существовании эффективного решения для задач

поиска максимального потока и удаления циклов минимальной стоимости в сети G^* .

Теорема 2. Существует алгоритм поиска максимального потока в сети G^* , асимптотическая сложность которого равна $\mathcal{O}(D \cdot |E| + D^2 \cdot |V|)$.

Доказательство. Обратимся к доказательству асимптотической сложности алгоритма Диница, которая в произвольных сетях составляет $\mathcal{O}(|V|^2 \cdot |E|)$. Единственные ребра, которые могут насытиться в сети G^* — это исходящие из s или входящие в t ребра. В графе \check{G} расстояние между любыми двумя вершинами не больше, чем $D + 1$, так как из вершины возможно добраться до стартовой вершины по обратному ребру, а от стартовой вершины до всех остальных расстояние не больше чем D . Таким образом, если существует путь из s в t , то его длина не больше чем $D + 3$ (ребра из \check{G} никогда не насыщаются), из чего следует то, что алгоритм Диница производит $\mathcal{O}(D)$ итераций поиска блокирующего потока. Из этого также следует, что один обход в глубину в поиске блокирующего потока выполняется за $\mathcal{O}(k + D)$, где k — число передвижений указателя на первое полезное исходящее ребро. Каждый обход в глубину насыщает как минимум одно ребро в сети, а так как число ребер, которые возможно насытить, составляет $\mathcal{O}(|V|)$, то суммарно поиск блокирующего потока имеет асимптотическую сложность $\mathcal{O}(|E| + D \cdot |V|)$, а алгоритма Диница в целом — $\mathcal{O}(D \cdot |E| + D^2 \cdot |V|)$. \square

Теорема 3. Существует алгоритм поиска максимального потока в сети G^* , асимптотическая сложность которого равна $\mathcal{O}(D \cdot |E| \cdot \log \max_{v \in V(G)} |\delta_+^G(v) - \delta_-^G(v)|)$.

Доказательство. Обратимся к доказательству асимптотической сложности алгоритма Диница с масштабированием потока, которая в произвольных целочисленных сетях составляет $\mathcal{O}(|V| \cdot |E| \cdot \log c_{max})$. На очередной итерации масштабирования потока рассматриваются только ребра, имеющие пропускную способность не менее чем A . При этом в остаточной сети отсутствуют пути, по которым возможно протолкнуть больше чем $2A$ потока, поэтому в остаточной сети максимальный поток не может превышать $2A \cdot |V|$, потому что в графе G^* не более $|V|$ ребер, которые возможно насытить. Так как по каждому пути проталкивается значение потока не менее A , то алгоритм Диница суммарно по всем поискам блокирующего потока проталкивает поток не более

чем $2 \cdot |V|$ путям, а значит, применяя рассуждения аналогичные предыдущей теореме, имеет асимптотическую сложность $\mathcal{O}(D \cdot |E| + D \cdot |V|) = \mathcal{O}(D \cdot |E|)$.

В сети G^* присутствуют ребра с бесконечной пропускной способностью, поэтому задавать начальное значение A равное c_{max} не имеет смысла. Заметим, что если задать начальное значение $A > \max_{v \in V} |\delta_+^G(v) - \delta_-^G(v)|$, то в сети G^* не будет существовать пути из s в t . Таким образом, начальное значение A ограничено значением $\max_{v \in V} |\delta_+^G(v) - \delta_-^G(v)|$, а асимптотическая сложность алгоритма в целом — $\mathcal{O}(D \cdot |E| \cdot \log \max_{v \in V(G)} |\delta_+^G(v) - \delta_-^G(v)|)$. \square

Теорема 4. Существует алгоритм удаления циклов отрицательной стоимости в сети G^* , асимптотическая сложность одной итерации которого равна $\mathcal{O}(|E|)$.

Доказательство. Так как единственные ребра, которые имеют ненулевую стоимость в сети G^* — это обратные ребра $e \in E(\check{G}) \setminus E(G)$, то отрицательный цикл в остаточной сети G_f^* должен содержать обратное к e ребро стоимости -1 . Из этого также следует, что любой отрицательный простой цикл должен содержать ровно одно ребро стоимости -1 , так как иначе цикл будет содержать стартовую вершину более, чем два раза. При этом отрицательный простой цикл не может содержать обратного ребра, потому что цикл перестанет быть отрицательным. Таким образом, чтобы найти отрицательный цикл в сети G_f^* , достаточно запустить обход в ширину из стартовой вершины, первым шагом переходящий по ребрам отрицательной стоимости, а далее игнорирующий обратные ребра. Если приведенный алгоритм нашел путь до стартовой вершины, то этот путь в точности является искомым отрицательным циклом. Асимптотическая сложность обхода в ширину составляет $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$. \square

В худшем случае удаление циклов отрицательной стоимости может занять $\mathcal{O}(|E|)$ итераций, так как при удалении одного из циклов насыщается как минимум одно из ребер, обратных к ребрам $E(\check{G})$. В практической части будет показано, что в рассматриваемых графах после исполнения алгоритма Диница отрицательных циклов остается очень мало.

Стоит отметить, что при доказательстве асимптотической сложности алгоритма Диница в задаче покрытия графа состояний путями никак не использовался факт ацикличности графа — асимптотическая сложность останется прежней даже в случае, когда граф состояний содержит циклы. Ацикличность

графа использовалась при доказательстве корректности и асимптотики алгоритма удаления циклов отрицательной стоимости для гарантии того, чтобы в графе \check{G} все простые циклы пересекались в стартовой вершине. Если требуется удаление отрицательных циклов при рассмотрении графа, не являющегося ациклическим, то требуется использование более сложного алгоритма поиска отрицательных циклов — в частности, алгоритма Форда-Беллмана), имеющего асимптотическую сложность $\mathcal{O}(|V| \cdot |E|)$.

Выводы по главе 2

В данной главе было приведено подробное описание спецификации TLA^+ алгоритма репликации Viewstamped Replication, используемого в распределенной базе данных ВКонтакте GMS, была приведена архитектура инструмента тестирования на основе модели TLA^+ вместе с подробным описанием его различных компонент и этапов исполнения, а также был описан алгоритм, эффективно решающий задачу покрытия графа состояний путями, вместе с доказательствами его корректности и асимптотической сложности.

ГЛАВА 3. ПРАКТИЧЕСКОЕ ИССЛЕДОВАНИЕ

3.1. Результаты проверки модели TLC

В таблице 2 представлены результаты запусков проверки модели TLC на спецификации алгоритма Viewstamped Replication для некоторых значений постоянных. Запуски производились на восьми потоках на процессоре Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz.

Таблица 2 – Таблица с результатами успешных запусков проверки модели TLC алгоритма Viewstamped Replication.

Значения постоянных	Диаметр D	Число состояний $ V $	Число переходов $ E $	Время проверки
ReplicasNumber $\leftarrow 3$ MaxLogQueryCount $\leftarrow 3$ MaxViewNumber $\leftarrow 2$	26	207 933	1 340 753	00:00:40
ReplicasNumber $\leftarrow 3$ MaxLogQueryCount $\leftarrow 4$ MaxViewNumber $\leftarrow 2$	29	2 030 058	12 921 365	00:06:21
ReplicasNumber $\leftarrow 3$ MaxLogQueryCount $\leftarrow 5$ MaxViewNumber $\leftarrow 1$	26	581 749	3 035 901	00:01:57
ReplicasNumber $\leftarrow 5$ MaxLogQueryCount $\leftarrow 1$ MaxViewNumber $\leftarrow 2$	24	166 260	2 266 623	00:01:04
ReplicasNumber $\leftarrow 5$ MaxLogQueryCount $\leftarrow 2$ MaxViewNumber $\leftarrow 2$	29	5 837 512	75 127 216	00:47:25

3.2. Результаты решения задачи покрытия графа путями

Реализованный инструмент для генерации тестов на основе модели для заданной спецификации и конфигурации TLA^+ генерирует файл, содержащий покрытие графа состояний путями (исполнениями) В таблице 3 приведены результаты запусков тестов с конфигурациями, аналогичными конфигурациям в таблице 2. Запуски производились в одном потоке на процессоре AMD Ryzen 5 4600H 3.00 GHz.

Стоит отметить, что число отрицательных циклов значительно меньше числа ребер $|E|$. Таким образом, удалять отрицательные циклы после наход-

Таблица 3 – Таблица с результатами решения задачи покрытия графа состояний TLC путями.

Значения постоянных	Число путей $ P $	Число отриц. циклов	Время поиска путей	Размер JSON файла
ReplicasNumber $\leftarrow 3$ MaxLogQueryCount $\leftarrow 3$ MaxViewNumber $\leftarrow 2$	248 626	0	6.17 с	111 Мбайт
ReplicasNumber $\leftarrow 3$ MaxLogQueryCount $\leftarrow 4$ MaxViewNumber $\leftarrow 2$	2 453 461	0	55.23 с	1 178 Мбайт
ReplicasNumber $\leftarrow 3$ MaxLogQueryCount $\leftarrow 5$ MaxViewNumber $\leftarrow 1$	534 711	2	13.72 с	286 Мбайт
ReplicasNumber $\leftarrow 5$ MaxLogQueryCount $\leftarrow 1$ MaxViewNumber $\leftarrow 2$	437 158	35	9.86 с	167 Мбайт
ReplicasNumber $\leftarrow 5$ MaxLogQueryCount $\leftarrow 2$ MaxViewNumber $\leftarrow 2$	15 181 992	438	265.58 с	> 6 441 Мбайт

дения максимального потока вовсе не обязательно, так как это не дает значительного выигрыша на стадии непосредственного тестирования.

3.3. Тестирование инструмента генерации тестов на основе модели

Инструмент для генерации тестов, описанный в данной дипломной работе, тестировался на примерах моделей, в большинстве своем приведенных в официальном репозитории TLC. Входные и выходные данные одного из тестовых запусков, а также возможная реализация теста описаны и представлены на примере принципа Дирихле в приложении А.

При этом учитывался тот факт, что многие примеры моделей из репозитория TLC имеют граф состояний, который не является ациклическим. На подобных моделях при решении задачи покрытия графа состояний путями этапа удаления циклов отрицательной стоимости не происходило, так как для этого пришлось бы использовать менее эффективный алгоритм Форда-Беллмана.

Дополнительно был протестирован добавленный в TLC класс StateGraphPathExtractor, решающий задачу о покрытии путями

графа состояний, на различных тестах, свойственных задаче поиска максимального потока в транспортной сети.

3.4. Особенности реализации абстрактного теста для алгоритма GMS

В алгоритме GMS сообщения между узлами системы передаются по специальным каналам, которые существуют для каждой пары узлов. При этом в спецификации TLA^+ каналы никак не моделируются — это вызвано тем, что реализация алгоритма совершает какие-либо полезные действия только при получении некоторым узлом сообщения. Поэтому каждому действию в модели TLA^+ , кроме `TimeoutStartViewChange`, соответствует принятие узлом некоторого сообщения по сети, а отсутствие действия означает либо потерю сообщения, либо отсутствие факта отправки сообщения. Таким образом, все исполнения, порожденные моделью TLA^+ , содержат в переходах полную информацию о принятых сообщениях узлами, а значит в реализации абстрактного теста отдельно хранить все отправленные сообщения не требуется — их содержимое можно получить из аргументов соответствующих действий.

Дополнительно, в GMS уже присутствуют написанные фаззинг тесты. Так как реальная система имеет открытые ресурсы в виде файловой системы и каналов между узлами, то фаззинг тесты содержат симуляцию этих ресурсов: запись в файловую систему заменяется записью в оперативную память, а каналы между узлами — глобальным массивом, содержащим все отправленные, но еще не дошедшие до адресата, сообщения. При этом на каждом шагу тест делает одно случайное действие из следующих:

- переинициализация системы и сброс состояния каждого из узлов;
- получение текущим лидером нового клиентского запроса и отправка соответствующих сообщений;
- срабатывание таймера у одного из узлов и инициация выборов нового лидера посредством протокола `view change`;
- скачивание одним из узлов одного блока журнала посредством протокола скачивания, если этот узел в данный момент уже у кого-то скачивает журнал;
- получение одним из узлов какого-либо сообщения в сети, и выполнения соответствующих действий по принятию, в том числе отправка новых сообщений;
- удаление из сети одного из сообщений, что соответствует его потере.

Листинг 6 – Пример реализации отображения одного из переходов алгоритма VR на модель TLA^+ в тесте на языке программирования Go.

```
func (m *VRUModelMapping) TimeoutStartViewChange(r int) {
    receiver := m.sch.Cluster[r]
    receiver.StartViewChange(receiver.ViewNumber + 1)
}
```

Таким образом, для большинства действий из модели TLA^+ уже существуют соответствующие действия реализации системы — достаточно взять соответствующие функции, использующиеся в фаззинг тестах, что и было сделано при совместной работе с разработчиками распределенной базы данных GMS.

Выводы по главе 3

В данной главе были приведены результаты проверки модели TLC для спецификации алгоритма репликации Viewstamped Replication на некоторых значениях постоянных и результаты запуска описанного в разделе 2.3 алгоритма для решения задачи покрытия графа состояний путями. Было описано тестирование разработанного инструмента генерации тестов на основе модели, в частности, алгоритма покрытия графа путями. Дополнительно были описаны особенности и трудности, которые возникли при реализации абстрактного теста, сгенерированного разработанным инструментом.

ЗАКЛЮЧЕНИЕ

В ходе данной дипломной работы были выполнены все поставленные задачи: была изучена предметная область, написана и верифицирована спецификация TLA^+ для алгоритма репликации, используемого в базе данных ВКонтакте, а также реализован инструмент для генерации тестов на основе модели TLA^+ , который впоследствии был применен для тестирования реализации базы данных ВКонтакте.

В частности, было проведено сравнение решения с аналогами, был разработан эффективный алгоритм, решающий задачу покрытия всех ребер графа состояний путями, а также было приведено доказательство корректности и асимптотической сложности алгоритма.

В будущем планируется дальнейшее усовершенствование инструмента тестирования на основе модели, представленного в данной дипломной работе, в частности:

- поддержка большего подмножества синтаксиса TLA^+ ;
- поддержка большего числа языков программирования, в которых генерируются тесты;
- уменьшение занимаемого места на диске JSON файлом, хранящим исполнения, покрывающие все переходы графа состояний модели (возможно, отказ от формата JSON);
- поддержка использования рассматриваемого метода тестирования в процессе непрерывной интеграции.

Дополнительно планируется использование результатов работ коллег — Никиты Синяченко и Идриса Яндарова — для оптимизации размера графа состояний модели TLC и поддержки тестирования протокола реконфигурации алгоритма Viewstamped Replication соответственно.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 *Косяков М. С.* Введение в распределенные вычисления. — 2014.
- 2 *Сердюков А. И.* О некоторых экстремальных обходах в графах // Управляемые системы. — 1978. — № 17. — С. 76–79.
- 3 *Lamport L.* Time, clocks, and the ordering of events in a distributed system // Concurrency: the Works of Leslie Lamport. — 2019. — С. 179–196.
- 4 *Liskov B., Cowling J.* Viewstamped replication revisited. — 2012.
- 5 *Ongaro D., Ousterhout J.* In search of an understandable consensus algorithm // 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). — 2014. — С. 305–319.
- 6 *Utting M., Legeard B.* Practical Model-based Testing: A Tools Approach. — Morgan Kaufmann Publishers, 2007. — (Morgan Kaufmann). — ISBN 9780123725011.
- 7 *Dorminey S.* Kayfabe: Model-Based Program Testing with TLA+/TLC [Электронный ресурс]. — 2020. — URL: https://conf.tlapl.us/2020/11-Star_Dorminey-Kayfabe_Model_based_program_testing_with_TLC.pdf.
- 8 *Lamport L.* The TLA+ Home Page [Электронный ресурс]. — URL: <https://lamport.azurewebsites.net/tla/tla.html>.
- 9 Modelator: Model-based testing tool [Электронный ресурс]. — 2021. — URL: <https://github.com/informalsystems/modelator>.
- 10 TLC Model Checker [Электронный ресурс]. — URL: <https://github.com/tlaplus/tlaplus>.

ПРИЛОЖЕНИЕ А. ЗАПУСК РАЗРАБОТАННОГО ИНСТРУМЕНТА НА ПРИМЕРЕ ПРИНЦИПА ДИРИХЛЕ

Листинг А.1 – Спецификация TLA⁺ для принципа Дирихле.

```

---- MODULE Dirichlet ----

EXTENDS Integers

CONSTANTS N, M

ASSUME ValidN ==
  N \in Nat \ {0}

ASSUME ValidM ==
  /\ N \in Nat
  /\ M > N

VARIABLES step , counters

vars == <<step , counters>>

Indices == 0 .. N - 1

TypeOK ==
  /\ step \in Int
  /\ counters \in [Indices -> Nat]

Init ==
  /\ step = 0
  /\ counters = [i \in Indices |-> 0]

IncrementCounter(i) ==
  /\ step < M
  /\ step' = step + 1
  /\ counters' = [counters EXCEPT ![i] = @ + 1]

Next == \E i \in Indices: IncrementCounter(i)

Spec == Init /\ [][Next]_vars /\ WF_vars(Next)

Terminated == step = M

```

```
ExistsExtra == \E i \in Indices: counters[i] > 1
```

```
DirichletProp ==
```

```
<>Terminated /\ [](Terminated => ExistsExtra)
```

```
====
```

Листинг А.2 – Конфигурация TLA^+ для принципа Дирихле.

```
CONSTANT N = 5
CONSTANT M = 6
INVARIANT TypeOK
SPECIFICATION Spec
PROPERTY DirichletProp
```

Листинг А.3 – Выходной JSON файл инструмента, содержащий сериализованные состояния TLA^+ и покрытие графа состояний путями. Сокращенный код обозначен многоточием (. . .). Были добавлены дополнительные пробельные символы для лучшей удобочитаемости.

```
{
  "states": [
    {
      "counters": [0, 0, 0, 0, 0],
      "step": 0
    },
    {
      "counters": [1, 0, 0, 0, 0],
      "step": 1
    },
    ...
  ],
  "executions": [
    [0, [0, 0], 1, [0, 0], 6, [0, 0], 21, [0, 0], 56, [0, 0],
      126, [0, 0], 252],
    [0, [0, 0], 1, [0, 0], 6, [0, 0], 21, [0, 0], 56, [0, 0],
      126, [0, 1], 253],
    ...
  ]
}
```

Листинг А.4 – Основная часть сгенерированного исходного кода заглушки теста на языке программирования Go для принципа Дирихле.

```

type DirichletModelState struct {
    Step      int    `json:"step"`
    Counters []int `json:"counters"`
}

type DirichletModelMapping struct {
}

func (m *DirichletModelMapping) PerformAction(id int, args []any)
{
    switch id {
    case 0:
        m.IncrementCounter(int(args[0].(float64)))
        break
    }
}

func (m *DirichletModelMapping) Init() {
}

func (m *DirichletModelMapping) Reset() {
}

func (m DirichletModelMapping) State() DirichletModelState {
    return DirichletModelState{}
}

func (m *DirichletModelMapping) IncrementCounter(i int) {
}

```

Листинг А.5 – Пример реализации заглушки теста на языке программирования Go для принципа Дирихле.

```

type DirichletModelState struct {
    Step      int    'json:"step"'
    Counters []int 'json:"counters"'
}

type DirichletModelMapping struct {
    Step int
    Arr  [5]int
}

func (m *DirichletModelMapping) PerformAction(id int, args []any)
{
    switch id {
    case 0:
        m.IncrementCounter(int(args[0].(float64)))
        break
    }
}

func (m *DirichletModelMapping) Init() {
    m.Step = 0
    m.Arr = [5]int{}
}

func (m *DirichletModelMapping) Reset() {
    m.Step = 0
    m.Arr = [5]int{}
}

func (m DirichletModelMapping) State() DirichletModelState {
    return DirichletModelState{
        Step:      m.Step,
        Counters: m.Arr[:],
    }
}

func (m *DirichletModelMapping) IncrementCounter(i int) {
    m.Step++
    m.Arr[i]++
}

```