Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Дружков Сергей Александрович

Выпускная квалификационная работа

Поиск клонов и автоматический рефакторинг Lua-программ в контексте задачи уменьшения размера кода для сетевых устройств

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2021
«Современное программирование»

Научный руководитель:
профессор кафедры системного программирования СПбГУ, д.т.н. Д. В. Кознов

Рецензент:
директор лаборатории ООО «Техкомпания Хуавэй» Д. А. Кудрявцев

Санкт-Петербург
2025

Saint Petersburg State University
Department of Mathematics and Computer Science

Sergei Druzhkov

Bachelor's Thesis

Clone detection and automatic refactoring of Lua programs for network devices

Education level: bachelor
Speciality 01.03.02 "Applied Mathematics and Computer Science"
Programme CB.5156.2021 "Modern Programming"

Scientific supervisor:
Professor, Sc.D. D. V. Koznov

Reviewer:
Laboratory director at "Huawei Technologies Co. Ltd."
D. A. Kudryavtsev

Saint Petersburg
2025

Содержание

Введение

Широко известен скриптовый язык Lua, разработанный в 1993 году Роберту Иерузалимски в Католическом университете Рио-де-Жанейро. Этот язык обладает облегчённым синтаксисом, позволяя создавать компактный и хорошо читаемый код, и часто используется в компьютерных играх, а также при разработке встроенных систем. Например, компании Cisco, Xiaomi и Huawei используют Lua в программном обеспечении для роутеров с целью проверки и модификации данных сетевых устройств. Интерпретатор языка написан на языке C и распространяется под MIT-лицензией[1]. Язык активно развивается и на сегодняшний день — 5.1 [39], 5.2 [40], 5.3 [41], 5.4 [42]. Эти версии имеют несколько отличающийся синтаксис, а также ряд отличий в стандартных библиотеках. Для задач, критичных по времени, для Lua 5.1 реализована поддержка JIT-компиляции (проект LuaJIT [45]).

Программное обеспечение роутеров крупной телекоммуникационной компании состоит из множества компонент, которые реализуют процесс маршрутизации сетевых пакетов, различные сетевые протоколы и интерфейсы, базу данных устройства, драйверы аппаратуры, задачи сетевого менеджмента и пр. Функциональность многих компонент похожа, что связано, в частности, с поддержкой не одного продукта, а целой линейки. По этой причине зачастую разработка ведётся путём копирования и доработки уже существующих компонент (copy-paste). Стоит также отметить, что роутеры и другие встроенные системы имеют существенные технические ограничения, и дублирование кода вынуждает производителей использовать дополнительную память, что снижает рентабельность итоговых изделий.

Проблема дублирования кода особенно остро стоит для приложений на языке Lua, т.к. этот язык используется на роутерах в режиме интерпретации (на это есть свои причины), что значительно увеличивает объём Lua-кода. Таким образом, задача уменьшения объёмов Lua-кода путём устранения дублирования является актуальной.

Существуют различные техники, позволяющие уменьшить итоговый размер программной системы. Например, можно использовать классические

[1] Данная лицензия позволяет модифицировать исходный код и продавать программное обеспечение, созданное на его основе

алгоритмы сжатия [25] или удаление из исполняемых файлов ненужных символов и отладочной информации [61]. Но для интерпретируемых языков обычно используются техники, основанные на преобразованиях исходного кода программ [64], [67]. Например, они используются для уменьшения размеров JavaScript-кода, чтобы экономить трафик и ускорить загрузку Web-страниц. Обычно перед распространением программы удаляются пробелы, комментарии, пустые строки и выполняется переименование локальных переменных. Также некоторые такие “минификаторы” поддерживают удаление неиспользуемого и недостижимого кода, а также другие оптимизации. Однако все эти преобразования не решают проблему дублирования кода.

Задача удаления дубликатов кода состоит из двух частей: (i) поиск клонов и (ii) выделение подходящих клонов в повторно используемые элементы, как правило, процедуры/методы (рефакторинг). Несмотря на многочисленные исследования в этой сфере [32], [48], [52], [71], а также имеющиеся успешные индустриальные решения (например, поиск клонов и выделение клонов в метод в многоязыковой среде разработки IntelliJ IDEA [22]), дальнейшие исследования продолжают оставаться актуальными. Это происходит в силу того, что индустриальные сценарии поиска клонов и рефакторинга, а также специфичный технологический контекст накладывают особые требования на обе задачи.

Постановка задачи

Целью данной выпускной квалификационной работы является разработка решения для поиска клонов и автоматического рефакторинга Lua-программ, предназначенных для сетевых устройств крупной телекоммуникационной компании, с целью уменьшения размеров кода с последующей интеграцией решения в конвейер сборки. Для достижения этой цели в рамках работы были сформулированы следующие задачи.

1. Провести обзор предметной области: методов поиска клонов и рефакторинга, а также особенностей языка программирования Lua.
2. Сформулировать требования к решению.
3. Выбрать алгоритмы и методы, подходящие для поиска клонов и автоматического рефакторинга Lua-программ, спроектировать архитектуру решения.
4. Реализовать решение.
5. Провести экспериментальное исследование различных аспектов разработанного решения (выбранных методов поиска и рефакторинга клонов).

1. Обзор

1.1. Поиск клонов исходного кода

Во время разработки программных систем часто происходит копирование фрагментов кода и последующее их использование с небольшими изменениями или без них. Такое явление называется дублированием кода. Идентичные или похожие фрагменты кода называются клонами. Дублирование кода является одной из значительных проблем, т.к. приводит к большим затратам на поддержку и развитие программы, влияет на объём занимаемой приложением памяти, а также может быть источником ошибок из-за несогласованности изменений между оригинальным и скопированными фрагментами кода.

1.1.1 Классификация клонов

Первые исследования по поиску клонов, а также программные инструменты начали появляться в 90-х годах прошлого века. Бурное развитие направления клонов кода привело к необходимости создания единой терминологии для упрощения коммуникации и сравнения различных подходов. В работе [56] выделены четыре основных типа клонов на основе текстовой и функциональной схожести, которые в будущем стали общепринятыми.

- **Тип 1 (T1).** Идентичные фрагменты кода, отличаются только форматирование кода и комментарии.

Оригинальный код	Клон типа T1

`int f(int x, int y) {`
`  int z = x + y;`
`  return z;`
`}`
	`int f(int x, int y) {`
	`  int z =`
	`  x + y; // sum`
	`  return z;`
	`}`

Таблица 1: Клон типа T1.

- **Тип 2 (T2).** В дополнение к отличиям типа 1 добавляются различия в именах переменных, типов и констант.

Оригинальный код	Клон типа T2

`int f(int x, int y) {`
`  int z = x + y;`
`  return z;`
`}`
	`char f(char a, char b) {`
	`  char c = a + b;`
	`  return c;`
	`}`

Таблица 2: Клон типа T2.

- **Тип 3 (T3).** Аналогично типу 2, но также возможно отсутствие/добавление/модификация отдельных инструкций.

Оригинальный код	Клон типа T3

`int f(int x, int y) {`
`  int z = x + y;`
`  return z;`
`}`
	`char f(char a, char b) {`
	`  char p = a * b;`
	`  return p;`
	`}`

Таблица 3: Клон типа T3.

- **Тип 4 (T4).** Фрагменты кода являются семантически эквивалентными, но могут быть записаны по-разному (т.е. отличаться синтаксически).

Оригинальный код	Клон типа T4

`int f(int n) {`
`  int s = 0;`
`  while (n > 0) {`
`  s += n--;`
`  }`
`  return s;`
`}`
	`int f(int n) {`
	`  if (n == 0) {`
	`  return 0;`
	`  } else {`
	`  return n + f(n - 1);`
	`  }`
	`}`

Таблица 4: Клон типа T4.

1.1.2 Методы поиска клонов

В работе [56] выделяется шесть основных групп методов поиска клонов; в последнее время в отдельную группу выделяются методы на основе машинного обучения. Итак, рассмотрим следующие группы методов поиска клонов:

- текстовые,
- лексические,
- синтаксические,
- семантические,
- метрические,
- комбинированные,
- основанные на машинном обучении.

Методы этих групп с разной эффективностью ищут описанные выше виды клонов. Они также отличаются по затрачиваемым ресурсам (времени работы, потребляемой памяти). Выбор метода зависит от многих факторов: времени работы и требований по объёму кодовой базы, необходимой точности, качеству поиска и т.д. Поэтому выбор правильного метода для каждой определённой задачи очень важен.

Текстовые методы рассматривают исходный код как обычный текст, и таким образом задача поиска клонов сводится к поиску совпадающих подстрок. К преимуществам данных методов можно отнести независимость от языка и быстроту. Успешно находятся клоны T1, частично клоны T2, поиск клонов T3 и T4 практически невозможен. Найденные клоны могут не учитывать границы выражений и даже функций, что вызывает проблемы при ручной и особенно автоматической обработке результатов.

Лексические методы похожи на предыдущую группу. Но здесь добавлен шаг превращения текста в набор токенов с последующей нормализацией и ведётся поиск совпадающих последовательностей токенов. Инструменты этой группы успешно находят клоны T1 и T2, также могут находить некоторые клоны T3. Методы данной группы имеют аналогичные преимущества и недостатки с текстовыми методами.

Синтаксические методы в начале работы строят абстрактное синтаксическое дерево (АСД) для программы, а затем ищут совпадающие поддеревья в нём. Использование АСД позволяет находить более сложные паттерны дублирования кода, в том числе клоны Т3, но далеко не все, т.к. добавление или удаление инструкций может значительно изменять структуру АСД. Построение АСД и работа с ним приводит к худшей скорости работы и большему потреблению памяти, чем предыдущие группы методов.

Семантические методы обычно используют граф зависимостей программы (ГЗП). Инструкции программы являются вершинами ГЗП, а рёбра отображают потоки управления и данных внутри программы. Благодаря этому ГЗП хранит в себе не только синтаксическую, но и семантическую информацию о программе внутри себя. Поиск клонов сводится к поиску совпадающих поддеревьев в ГЗП. Семантические методы позволяют находить клоны типа Т1, Т2 и Т3 с большой точностью, но обладают большей вычислительной сложностью, чем остальные методы.

Метрические методы вычисляют различные метрики фрагментов кода на основе АСД или ГЗП, а затем на основе них находят схожие фрагменты кода. Такой подход позволяет искать клоны всех типов, но с низкой точностью.

Комбинированные методы позволяют увеличить точность различных методов, но не решают их глубинных проблем. Стоит также отметить, что комбинированный метод работает не быстрее, чем самая медленная его часть, поэтому невозможно добиться увеличения производительности таким способом.

Методы, основанные на машинном обучении, слишком разнообразны, но можно выделить два основных направления. Первым является использование “классического” машинного обучения, когда задача поиска клонов представляется как задача кластеризации. В противовес этому набирает популярность использование больших языковых моделей для извлечения семантической информации. Именно с этим подходом связаны последние успехи в поиске клонов Т4. Использование машинного обучения как чёрного ящика приводит к не самой высокой точности методов из этой группы, хотя и даёт новые возможности.

1.1.3 Инструменты поиска клонов

Существует множество реализаций описанных выше методов, которые расширяют классические идеи для повышения точности и количества найденных клонов.

Название	Метод	Типы клонов	Год выпуска
MOSS [14]	Текстовый	T1	2002
Duploc [20]	Текстовый	T1, T2 (частично)	2000
NiCad [16]	Текстовый	T1, T2	2008 - 2020
CCFinderX [30]	Лексический	T1, T2	2009
CPD [18]	Лексический	T1, T2	2002
SourcererCC [59]	Лексический	T1, T2	2016
CCStokener [68]	Лексический	T1, T2, T3 (частично)	2023
CCAligner [69]	Лексический	T1, T2, T3 (частично)	2018
Deckard [29]	Синтаксический	T1, T2, T3	2007 - 2015
CloneDR [13]	Синтаксический	T1, T2	1998 - 2023
MSCCD [73]	Синтаксический	T1, T2, T3	2022
CCGraph [74]	Семантический	T1, T2, T3, T4 (частично)	2020
CCLearner [36]	ML	T1, T2, T3, T4 (частично)	2017
Oreo [57]	Комбинированный	T1, T2, T3, T4 (частично)	2018

Таблица 5: Инструменты поиска клонов.

Инструмент MOSS [14] предназначен для поиска заимствований в работах студентов и был разработан в Стэндфордском университете. В основе инструмента лежит идея использования отпечатков, исходный код программы разбивается на равные куски длины k (k-граммы), которые затем хэшируются. Затем, сравнивая хеши фрагментов как отпечатки, находятся клоны. MOSS также доступен в качестве онлайн-сервиса [51].

Алгоритм работы инструмента Duploc [20] состоит из двух частей. Вначале исходный текст разбивается на строки, удаляются комментарии и про-

белы. Затем строится булева матрица, где на пересечении i-ой строки и j-го столбца стоит 1 тогда и только тогда, когда i-ая строка в первом файле равна j-ой строке во втором файле, в противном случае стоит 0. Тогда клоны на этой матрице выглядят как диагонали из 1, алгоритм также допускает пропуски не больше заданного размера в таких диагоналях, что позволяет находить малый процент клонов Т3.

Инструмент NiCad [16] основан на языке TXL [17]. TXL используется для нормализации исходного кода программы, который потом разбивается на фрагменты. Каждый фрагмент является потенциальным клоном. Фрагменты сравниваются между собой построчно для поиска клонов. Использование TXL для нормализации позволяет NiCad успешно находить клоны Т2.

Для поиска клонов в CCFinderX [30] используется суффиксное дерево. По исходному коду программы, разбитому на токены, строится суффиксное дерево, с помощью которого находятся одинаковые подпоследовательности токенов, которые и считаются клонами. Существует распределённая реализация D-CCFinder [37], используемая для анализа сверхбольших проектов.

В состав мультиязыкового статического анализатора PMD [54] входит инструмент поиска клонов CPD [18]. Для получения последовательности токенов используется ANTLR [10] грамматика. Поиск клонов в последовательности лексем происходит с помощью алгоритма Рабина-Карпа. CPD имеет возможность игнорировать имена переменных, литералы и аннотации, что позволяет ему эффективно искать клоны Т2.

Инструмент SourcererCC [59] ориентирован на анализ больших проектов. Первым шагом является построение последовательности токенов для каждого файла. После построения последовательности токенов SourcererCC для каждого блока выбирает подмножество токенов, на основе которых он будет проиндексирован. При поиске клонов индекс позволяет легко находить потенциальные клоны, схожесть клонов же оценивается на основе числа совпадений токенов, соответствующих блокам кода.

Современный инструмент для поиска клонов CCStokener [68] расширяет лексический метод добавлением дополнительной семантической информации к токенам. Примерами такой дополнительной информации являются информация о структуре окружающего кода и информация о взаимосвязях

с соседними токенами. Новый подход позволяет значительно улучшить возможности поиска клонов Т3.

В инструменте CCAligner [69] используется идея скользящего окна для поиска клонов. Исходный код преобразуется в последовательность токенов, затем используется скользящее окно фиксированной длины l и дополнительный числовой параметр e, описывающий число разрешённых несовпадений между фрагментами кода. Для каждого окна эффективно строятся все $(l-e)$-граммы, которые сохраняются в индексе. Этот подход позволяет искать клоны Т3 с большим числом отличий, контролируемым параметром e.

Инструмент Deckard [29] использует АСД для поиска клонов. Для каждого поддерева строится собственный характеристический вектор. Близкие вектора в терминах евклидова расстояния считаются клонами. Такой подход позволяет находить достаточно много клонов Т3, но использование неформального подхода не даёт никаких гарантий корректности на найденные клоны.

Коммерческий детектор клонов CloneDR [13] поставляется в составе набора инструментов DMSToolkit [19], разработанных компанией Semantic Designs. Для поиска клонов Т1 и Т2 инструмент ищет изоморфные поддеревья в АСД. Найденные деревья объединяются для получения клонов Т3. Стоит отметить поддержку множества языков: C/C++, C#, Java, Python, Ada, Cobol и другие.

Языконезависимый детектор клонов MSCCD [73] использует ANTLR [10] грамматики для генерации АСД, что позволяет ему поддерживать более 150 языков программирования. Для поиска клонов используется подход на основе “мешка” токенов. Для каждого блока кода генерируется мешок токенов, содержащий значимые токены. Мерой схожести между блоками кода служит процент общих токенов.

Инструмент CCGraph [74] использует граф зависимостей программы для поиска клонов. Вначале нормализуется структура ГЗП и происходит фильтрация на основе характеристических векторов кода. Потом используется алгоритм Вайсфейлера-Лемана [26] для поиска изоморфных подграфов. Использование ГЗП позволяет находить часть клонов Т4.

В инструменте CCLearner [36] используются глубокие нейронные сети

для ответа на вопрос, являются ли два метода клонами. Токены, полученные после лексического анализа, разбиваются на восемь категорий, для каждой категории считается частота встречаемости в методе. Затем на основе частот рассчитывается вектор схожести для каждого метода и функции, который затем поступает на вход нейронной сети, которая решает задачу классификации, являются ли два метода клонами или нет.

Метод, используемый в Oreo [57], состоит из двух этапов. На первом этапе для каждого метода строится вектор метрик. Затем для поиска клонов T1 и T2 строится индекс на основе хэширования характеристических векторов. Для поиска клонов T3 и T4 используются нейронные сети для кластеризации векторов, полученных на предыдущем шаге.

1.2. Рефакторинг клонов

Рефакторинг — преобразование исходного кода программы без изменения наблюдаемого поведения [23]. Существуют различные методы рефакторинга — так, Мартин Фаулер в своей знаменитой книге “Refactoring: Improving the Design of Existing Code” описал более ста методов рефакторинга.

1.2.1 Методы рефакторинга

Удаление клонов (частный случай рефакторинга) — это слияние двух или большего числа дублированных фрагментов кода. Только часть из стандартных методов рефакторинга применима для работы с клонами. В качестве примера методов рефакторинга клонов приведём некоторые, описанные в работе [50].

- **Извлечение метода.** Данный вид рефакторинга подразумевает, что часть кода некоторого метода извлекается в новый метод.
- **“Подъём” метода.** Данный вид рефакторинга заключается в перемещении общего для нескольких подклассов метода в суперкласс.
- **Перемещение метода.** Этот вид рефакторинга применяется к методу класса, если он чаще обращается к полям другого класса, чем к своим.

- **Параметризация метода.** В рамках данного вида рефакторинга несколько методов, обычно отличающихся литералами, объединяются в один. В полученный метод данные, различающиеся литералами, передаются через новые параметры.

- **Создание шаблонного метода.** Для создания шаблонного метода необходимо определить абстрактный класс с определением общей структуры алгоритма, а в наследниках переопределить шаги.

Наиболее универсальными из них являются извлечение метода (функции) и параметризация метода (функции), т.к. они применимы для большинства языков программирования, в то время как остальные — только для языков с поддержкой объектной ориентированности.

1.2.2 Инструменты рефакторинга клонов

Существует много детекторов клонов, но анализ их вывода и поиск важных для рефакторинга клонов — непростая задача для программиста. Автоматизация рефакторинга клонов может помочь программистам уменьшить усилия на поддержание и развитие кодовой базы, избежать ошибок, сэкономить время. Можно разделить все инструменты на три группы в зависимости от участия человека: ручной рефакторинг, полуавтоматический рефакторинг и автоматический рефакторинг.

Инструменты из группы ручного рефакторинга представляют собой тонкую прослойку над детектором клонов, ранжируя его результаты и подсказывая наиболее подходящий метод рефакторинга в зависимости от ситуации. Однако окончательное принятие решения и написание кода остаётся за программистом, что может приводить к неочевидным ошибкам.

Инструменты полуавтоматического рефакторинга обычно интегрированы в IDE. Основное отличие от предыдущей группы состоит в автоматической проверке возможности рефакторинга, а также генерации необходимого для этого кода, но всё равно требуют участия человека для генерации новых имён программных сущностей и проверки корректности итогового кода. Решения

этой группы часто ограничены пользовательским интерфейсом и возможностями рефакторинга только двух клонов за один раз.

Самым сложным сценарием является автоматический рефакторинг, т.к. вся ответственность за корректность преобразований кода ложится на программу. Для больших и старых систем иногда это оказывается единственным сценарием, т.к. нельзя отвлекать разработчиков от текущих задач, а сложность анализа кода и количество клонов слишком велики для обработки человеком. Также эта группа инструментов позволяет удалять сразу целые группы клонов, что затруднительно для инструментов из предыдущих групп.

Название	Тип рефакторинга	Типы клонов	Год выпуска
SUPREMO [34]	Ручной	T1	2002
IDEA [28]	Полуавтоматический	T1, T3	2002
CeDAR [63]	Полуавтоматический	T1, T2	2012
JDeodarant [48]	Полуавтоматический	T1, T2, T3	2015
CLoRT [12]	Автоматический	T1, T2	1999
JTestParametrizer [72]	Автоматический	T1, T2	2018
PyTeRor [32]	Автоматический	T1, T2	2024

Таблица 6: Инструменты рефакторинга клонов.

Интерактивная система SUPREMO [34] работает на основе детектора клонов Duploc [20], что делает её независимой от языка программирования. Она может предлагать наиболее подходящий рефакторинг в зависимости от контекста пары клонов: они находятся в одном методе, в одном классе, в соседних классах в иерархии и другие. Система в основном нацелена на объектно-ориентированные языки, такие как Java, C++, SmallTalk.

Интегрированная среда разработки от Jetbrains IntelliJ IDEA Ultimate [28] предоставляет функциональность поиска [9] и удаления дубликатов кода [22] для языков Java, Kotlin, JavaScript, TypeScript и Groovy, а также языков разметки HTML, XML и CSS. Извлечение метода — единственный поддерживаемый вариант рефакторинга. Имеется поддержка анонимизации имён локальных переменных, имён методов и классов, а также констант, что позволяет находить и удалять клоны T1 и T2.

Плагин для Eclipse CeDAR [63] позволяет находить и удалять клоны T1 и T2 с любой гранулярностью. CeDAR использует Deckard для поиска клонов, а затем с помощью движка рефакторинга Eclipse проверяет возможность рефакторинга. Поддерживаемые рефакторинги: извлечение метода, поднятие метода и добавление статического метода. Работает только с языком программирования Java.

Ещё один плагин для Eclipse JDeodarant [48] позволяет находить и рефакторить клоны T1, T2 и T3 в программах, написанных на Java. Для поиска клонов используются: CCFinder, Deckard, CloneDR, NiCad и ConQAT. JDeodarant самостоятельно проверяет клоны на возможность рефакторинга, для этого используется список из восьми правил [66]. Поддерживаемые рефакторинги: извлечение метода, поднятие метода, параметризация метода, создание шаблонного метода и добавление статического метода.

Инструмент автоматического рефакторинга клонов CLoRT [12] для Java поддерживает работу с клонами T1 и T2 на уровне методов. Поиск осуществляется самописным алгоритмом на основе АСД. Для рефакторинга клонов применяется паттерн стратегия. Проверка корректности осуществлялась на JDK. После удаления клонов система успешно работала и прошла все тесты.

Следующим инструментом является JTestParametrizer [72], ориентированный на удаление клонов тестов в программах на Java с помощью параметризации похожих тестов средствами известного фреймворка JUnit. Поддерживается работа только с клонами T2 на уровне функций. Работает поверх JDeodarant, используя его для поиска клонов и параметризации разницы между ними. Из-за этого инструмент ограничен в работе только с парами клонов. После применения инструмента на пяти открытых проектах в среднем 94% тестов прошли успешно.

Последним инструментом из группы автоматического рефакторинга на момент написания является PyTeRor [32], нацеленный на удаления клонов тестов в Python. Инструмент использует NiCad [16] для поиска клонов. PyTeRor работает только с клонами T2 на уровне функций. Рефакторинг происходит с помощью встроенной поддержки параметризации тестов в pytest. Результат применения инструмента на девяти открытых проектах показал 100% успешное выполнение тестов после применения рефакторинга.

1.3. Отечественные исследования в области поиска клонов

Наиболее активно в России исследования в области поиска клонов ведутся в Институте системного программирования РАН (ИСП РАН). Там было разработано несколько инструментов поиска клонов программ, а также методики их применения для улучшения качества программного обеспечения.

В диссертациях [5], [6] представлен детектор клонов CCD, работающий на основе графа зависимостей программы (ГЗП). Там же предложен метод для генерации ГЗП на основе промежуточного представления LLVM [38], что позволяет получить ГЗП во время компиляции кода без дополнительных накладных расходов. Также автор предложил метод поиска семантических ошибок, вызванных некорректным копированием фрагментов кода (в частности, поиск несогласованных имен переменных). Ещё одним применением разработанного инструмента является поиск копий известных уязвимостей в проектах и использованных библиотеках. К потенциальным недостатком детектора CCD можно отнести значительное время работы, достигающее почти четырёх часов при анализе больших проектов (например, ядра Linux).

Инструмент BINCCD [2] того же коллектива нацелен на поиск клонов кода в бинарных файлах. Вначале из бинарного файла восстанавливается структура программы (граф вызовов, граф потока управления и т.д.). После этого применяется уже разработанный алгоритм на основе ГЗП. Инструмент позволяет находить нарушения лицензий, копии вредоносного кода и пр.

Ещё одна диссертация, подготовленная в ИСП РАН, также посвящена поиску клонов [1]. В ней представлен метод поиска клонов произвольных фрагментов в исходном и исполняемом коде, основанный на ГЗП. ГЗП может быть построен на основе промежуточного представления LLVM [38], грамматик ANTLR [10], байт-кода Java и ассемблера. Данный метод позволяет добиться полноты 100% для клонов T1 и T2 и более 90% для клонов T3 на BigCloneBench [62], что значительно превосходит результаты существующих открытых инструментов.

В работе [7] представлен метод инкрементального поиска клонов для использования в контексте IDE. Метод основан на использовании суффиксного дерева для поиска схожих подпоследовательностей токенов. Метод поз-

воляет быстро находить клоны T1 и T2. На основе данного метода был разработан прототип плагина для IntelliJ IDEA [28].

В диссертации [3] разработана методика вынесения рекомендаций по повышению скорости исполнения программ на основе схожести фрагмента кода с базой эталонных реализаций алгоритмов. Поиск осуществляется на основе векторных представлений программы, полученных с помощью цепей Маркова. Для сравнения векторных представлений фрагментов кода используется расстояние Йенсена-Шеннона [49].

Задача поиска дубликатов может возникать не только в контексте программного кода. Программное обеспечение может отправлять трассировку стека после аварийного завершения (crash) для последующего анализа. Часто трассировки очень схожи, т.к. вызваны одними и теми же ошибками. Автоматическая группировка может значительно упростить и ускорить обработку сообщений об ошибках. Для решения данной проблемы в компании JetBrains был разработан алгоритм TraceSim [55], основанный на комбинации TF-IDF, алгоритмов выравнивания строк и машинном обучении.

Ещё одной областью исследований является поиск дубликатов в документации. В работах [35], [46], [47] представлены различные методы поиска дублирующихся комментариев. Предложенные методы позволяют находить как точные, так и близкие клоны. На основе тестирования различных алгоритмов, начиная от различных метрик схожести строк до методов машинного обучения, установлено, что наиболее оптимальным с точки зрения качества (точность и полнота) и производительности является алгоритм, основанный на поиске наибольшей общей подстроки. Данный алгоритм достигает 94% точности и 74% полноты.

1.4. Особенности языка Lua

Lua является популярным скриптовым динамически типизированным языком программирования. Его отличительными особенностями являются простота и гибкость. Для простоты дальнейшего изложения выбрана версия Lua 5.3 [41], но в конце данного раздела будут приведены основные различия между разными версиями.

```
chunk ::= block
block ::= {stat} [retstat]
stat ::= ‘;’ | do block end | label | break | goto Name |
     varlist ‘=’ explist | functioncall |
     while exp do block end | repeat block until exp |
     if exp then block {elseif exp then block} [else block] end |
     for Name ‘=’ exp ‘,’ exp [‘,’ exp] do block end |
     for namelist in explist do block end |
     function funcname funcbody | local function Name funcbody |
     local attnamelist [‘=’ explist]
attnamelist ::= Name attrib {‘,’ Name attrib}
attrib ::= [‘<’ Name ‘>’]
retstat ::= return [explist] [‘;’]
label ::= ‘::’ Name ‘::’
funcname ::= Name {‘.’ Name} [‘:’ Name]
varlist ::= var {‘,’ var}
var ::= Name | prefixexp ‘[’ exp ‘]’ | prefixexp ‘.’ Name
namelist ::= Name {‘,’ Name}
explist ::= exp {‘,’ exp}
exp ::= nil | false | true | Numeral | LiteralString | ‘...’ |
     functiondef | prefixexp | tableconstructor |
     exp binop exp | unop exp
prefixexp ::= var | functioncall | ‘(’ exp ‘)
functioncall ::= prefixexp args | prefixexp ‘:’ Name args
args ::= ‘(’ [explist] ‘)’ | tableconstructor | LiteralString
functiondef ::= function funcbody
funcbody ::= ‘(’ [parlist] ‘)’ block end
parlist ::= namelist [‘,’ ‘...’] | ‘...’
tableconstructor ::= ‘’ [fieldlist] ‘’
fieldlist ::= field fieldsep field [fieldsep]
field ::= ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp
fieldsep ::= ‘,’ | ‘;’
binop ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘//’ | ‘^’ | ‘%’ |
     ‘&’ | ‘~’ | ‘|’ | ‘>>’ | ‘<<’ | ‘..’ | and |
     ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’ | or
unop ::= ‘-’ | not | ‘#’ | ‘~’
```

Рис. 1: Грамматика Lua 5.3 в расширенной форме Бэкуса-Наура [41]

Lua использует лексическую область видимости, поэтому существуют три типа переменных: локальные, глобальные и внешние локальные переменные (upvalues). Интересной особенностью языка является работа с глобальными переменными. Любое обращение к глобальной переменной преобразуется в `_ENV["NAME"]`, где `_ENV` это специальная таблица, называемая окружением, ассоциированная с каждой функцией. Существует общее глобальное окружение `_G`, которое используется по умолчанию. `_ENV` является обычной переменной, которую можно переопределить или присвоить новое значение. На основе этого механизма работает система модулей внутри языка.

В Lua поддерживаются следующие основные типы данных: nil, bool, number, string, table, function, thread, userdata. Стоит отметить, что в списке типов, представленном выше, первые четыре типа являются типами значений, а оставшиеся — ссылочными типами. Поскольку в Lua нет возможности определять классы и структуры, то все пользовательские данные хранятся в таблицах. Даже массивы в Lua эмулируются с помощью таблиц.

Использование таблиц для хранения пользовательских данных приводит к необходимости большого числа синтаксического “сахара”, то есть семантически избыточных конструкций, которые созданы лишь для удобства пользователей. Например, обращение к полю `var.field` представляется как `var["field"]`, определение метода `M:method(params)` записывается как `M.method(self, params)`, множественные присваивания и пр.

Ещё одной необычной особенностью языка Lua является концепция метатаблиц. Каждому объекту соответствует какая-то метатаблица. Например, при попытке обратиться по ключу или индексу с помощью квадратных скобок к какому-то значению будет вызван метод, лежащий по ключу `"__index"` в метатаблице объекта. На самом деле метатаблицы — это просто служебные таблицы, к которым пользователь имеет доступ и может сам их создавать и настраивать. Они позволяют переопределить поведение операторов (`#`, `[]`, `()`, `+`, `-` и т.д.) у объекта, что позволяет эмулировать некоторые аспекты объектно-ориентированного программирования.

Множество синтаксического “сахара”, возможность динамически изменять поведение объекта (метатаблицы), отсутствие статической типизации являются большим вызовом для средств статического анализа.

Стандартная библиотека состоит из небольшого числа модулей. Кроме типичных для других языков модулей (работа с операционной системой, математические функции, работа со строками и таблицами, UTF-8, операции ввода-вывода), существуют более специфичные модули. Lua имеет встроенную библиотеку для работы с корутинами (`coroutine`). Интерпретатор Lua работает только в пределах одного потока, но использование корутин позволяет добиться эффективной асинхронной обработки данных. Другим примером является отладочная библиотека (`debug`), которая предоставляет доступ к части внутреннего состояния интерпретатора. Это позволяет разрабатывать свои инструменты для Lua на её основе, например, отладчик или профилировщик. Однако использование этой библиотеки может легко нарушить часть инвариантов языка, поэтому её использование не рекомендуется. Также имеется возможность вызывать C/C++ функции из Lua и наоборот, благодаря C API, что даёт лёгкую возможность для взаимодействия между кодом, написанным на разных языках.

Обсудим различия в версиях языка Lua. В версии Lua 5.1 по сравнению с Lua 5.2 отсутствуют метки и возможности неявной работы с окружением, т.к. не существует специальной переменной `_ENV` (для этого используются функции `setfenv` и `getfenv`). В версии Lua 5.3 добавлен целозначный тип. В последней на текущий момент версии Lua 5.4 появились два новых типа атрибутов переменных — `<const>` и `<close>`. LuaJIT почти полностью совпадает с Lua 5.1, за исключением добавления специальных суффиксов к определению чисел для лучшей работы с FFI. Конечно, были большие изменения стандартной библиотеки, внутренней реализации интерпретатора, но с точки зрения языка все изменения шли согласованно и приводили только к расширению возможностей, поэтому имеется возможность в едином стиле обрабатывать код, написанный на разных версиях языка.

1.5. Выводы

На данный момент не существует специализированных детекторов клонов для Lua. Языконезависимые детекторы клонов обычно используют специальный формат для описания грамматики и правил языка. Такой подход имеет

ограничения, связанные с одинаковой обработкой синтаксических конструкций различных языков. Среди языконезависимых детекторов стоит выделить детекторы, позволяющие искать клоны кода на Lua: Duploc [20] (текстовый метод), CPD [18] (лексический метод) и MSCCD [73] (синтаксический метод). Однако они не поддерживают синтаксический сахар, специфичный для Lua. Классические детекторы клонов обычно ориентированы на поиск наибольшего числа клонов. Однако такой подход снижает точность, а найденные клоны не всегда могут быть удалены.

Большинство инструментов рефакторинга клонов используют внутри себя отдельные детекторы клонов. Такой подход приводит к необходимости анализа и фильтрации результатов их работы, что влечёт за собой дополнительные накладные расходы. Наиболее популярным методом рефакторинга клонов является извлечение метода (функции). Для анализа возможности рефакторинга могут использоваться средства IDE или самописный анализ. Однако даже современные средства разработки для популярных языков, такие как clangd, не позволяют произвести безопасный рефакторинг, например, возникают проблемы с некорректным выделением функции при использовании в коде замыканий и глобальных переменных. Для Lua же не существует полноценной IDE, а существующие плагины не имеют встроенной функциональности рефакторинга, поэтому такой путь невозможен. Условия рефакторинга, описанные в работе [66], являются необходимыми, но недостаточными. Они сильно зависимы от языка Java и требуют доработки под другие языки программирования. Можно избежать проблем, описанных выше, если сразу искать рефакто-ориентированные клоны. Например, среди клонов T2 можно выделить подгруппу, где переименование переменных было согласованно, остальные же клоны T2 невозможно отрефакторить, поэтому нет смысла их искать.

Все инструменты автоматического рефакторинга работают на уровне функций, т.к. в таком случае клоны не пересекаются и можно легко их выделить в новые функции. В работе [71] изучается процент дублирования кода программ, написанных на языках C и COBOL. Для поиска клонов использовался инструмент CCFinderX [30]. Было предложено несколько стратегий удалений клонов. Базовый метод, состоящий только в удалении клонов функ-

ций. Полный метод, состоящий в жадном удалении всех клонов без проверок на корректность рефакторинга. Эвристический метод, схож с предыдущим, но проверяет клоны на пересечение (сводится к задаче комбинаторной оптимизации). Процент клонов колеблется от 1 до 10 в зависимости от проекта. Аналогично колеблется процент клонов, которые можно удалить разными методами. Для базового метода число удалённых клонов колеблется от 5% до 15% от общего числа найденных клонов. Для эвристического метода количество удалённых клонов составляет от 35% до 45% от общего числа клонов. В статье [52], являющейся ответом на предыдущую, было показано, что реальное число клонов, которые можно безопасно удалить, в среднем в 3-4 раза меньше. Основная проблема возникает при удалении вложенных и пересекающихся клонов, т.к. необходимо производить их удаление согласованно. Алгоритм, описанный в статьях, заключается в итеративном удалении клонов и последующем повторном запуске поиска клонов, что влечёт большое время работы из-за повторяющегося этапа поиска клонов.

2. Требования к решению

Данная глава описывает функциональные и нефункциональные требования, которым должно удовлетворять разработанное решение, а также процесс разработки требований, представляющий интерес в виду итеративности и нелинейности.

2.1. Функциональные требования

Были выделены следующие функциональные требования.

- Решение должно предоставлять возможность поиска клонов на языке программирования Lua, включая учёт таких особенностей языка, как работа с глобальными переменными, большое количество синтаксического сахара и пр.
- Решение должно позволять конфигурировать поиск клонов: минимальный размер клона в токенах, гранулярность (функция или блок), область поиска (один файл или весь проект).
- Решение должно предоставлять возможность автоматического рефакторинга групп одинаковых клонов одновременно, а не только пар клонов.
- Решение должно предоставлять возможность поиска и рефакторинга вложенных клонов.
- Решение должно позволять конфигурировать рефакторинг клонов: область рефакторинга (один файл или весь проект), а также название и расположение модуля с новыми функциями.
- Решение должно быть встроено в существующий конвейер сборки прошивки для сетевых устройств компании заказчика.

2.2. Нефункциональные требования

Решение должно удовлетворять следующим нефункциональным требованиям.

- Поддерживать работу с большой кодовой базой (≈ 400Мб, ≥ 10MLOC).
- Иметь минимально возможное число зависимостей от внешних библиотек и приложений для работы в контексте сборки прошивки сетевых устройств.
- Обеспечивать детерминированность результатов работы для воспроизводимости сборок.
- Сохранять при рефакторинге структуру кода для удобства отладки.
- Обеспечивать безопасность всех преобразований кода для сохранения его работоспособности.

2.3. Процесс разработки требований

Успешная разработка требований при решении задач на стыке исследований и бизнеса является основополагающим фактором успеха. Сколько бы эффективным ни казался результат, он будет применяться для конкретных бизнес-сценариев, и может оказаться, что, несмотря на свой передовой "характер", он не оптимален для данного контекста или даже вовсе неприменим. С другой стороны, бизнес-требования изменчивы и часто могут быть даже теоретически невыполнимыми.

Выходом является итеративное улучшение требований к таким решениям, а для этого необходимо находиться в постоянном контакте с пользователями.

Изначальная идея предложенного решения заключалась в полуавтоматическом поиске клонов и рефакторинге кода. Проверка человеком позволяла бы выявить возможные ошибки и помогала бы в поддержании "чистоты" кодовой базы. Для проверки концепции был разработан алгоритм поиска клонов на основе SimHash [60] — специальной хэш-функции, которая для близких строк возвращает близкие значения, что позволяет по её значениям оценить схожесть изначальных строк. Для парсинга исходного кода использовалась библиотека Tree-sitter [65], которая позволяет по грамматике языка сгенерировать код парсера на C. Также применялись дополнительные преобразования к полученному АСД для дополнительной унификации. В результате работы

на тестовом проекте было найдено более 10000 потенциальных клонов, обработать которые “вручную” оказалось неприемлемым. Параллельно разрабатывались другие средства уменьшения размера Lua-кода, и они работали в автоматическом режиме. В итоге было решено объединить эти решения и осуществлять поиск клонов и рефакторинг полностью в автоматическом режиме.

Следующие итерации разработки требований к решению были нацелены на улучшение и расширение его возможностей. Было выдвинуто требование по осуществлению поиска и рефакторинга клонов функций в пределах одного файла. После успешной реализации этого требования появилось следующее требование — повторить всё то же на уровне блоков кода в пределах всего проекта. Между разными уровнями гранулярности поиска клонов и областями поиска существует компромисс и по времени работы решения, и по уровню влияния на производительность итогового Lua-кода, поэтому важно уметь удобно их конфигурировать. Более подробное описание принятых решений будет приведено в главе, посвящённой реализации.

Код прошивки сетевого устройства состоит из кода на C, конфигурационных XML-файлов и Lua-скриптов. Конфигурационные файлы служат для описания взаимосвязей между разными компонентами сетевого устройства, а также связывают Lua и C код через общие идентификаторы. Процесс сборки прошивки состоит из множества этапов, включающих компиляцию, линковку, сборку различных пакетов и их объединение. При сборке пакетов с Lua-кодом происходит кодогенерация на основе конфигурационных файлов и оптимизация получившегося кода с точки зрения размера. Поскольку предполагалось, что решение должно участвовать в процессе сборки прошивки, то важна воспроизводимость результатов работы решения и безопасность выполняемых им преобразований исходного кода. Контекст использования решения накладывает ограничения на возможные зависимости, т.к. они должны быть одобрены юридическим отделом компании. Также сервера, используемые для сборки, имеют ограничения на выход в Интернет и используемое программное обеспечение, поэтому важно избегать зависимостей от стороннего программного обеспечения.

Всего было проведено четыре большие встречи для выявления требова-

ний к решению в течение шести месяцев. Параллельно в это же время велась реализация очередной порции требований. Процесс разработки решения был основан на инкрементальном улучшении с учётом новой итерации требований. Идеи и предложения, появившиеся во время итерации, служили основой для требований к следующей итерации. Раннее предоставление прототипов позволяло выявлять крайние случаи и исправлять ошибки до релизов.

3. Архитектура

В данной главе описывается архитектура предложенного решения — компонентный состав решения, взаимодействие компонент и сценарий работы решения.

Поиск клонов реализован на основе метода, предложенного в Vuddy [31]. Выбор этого метода обусловлен приемлемым быстродействием и ориентированностью на большие проекты. Использование более сложных методов не является целесообразным, т.к. рефакторинг клонов T3 и тем более T4 является открытой и сложной задачей. Выбранным методом рефакторинга является извлечение функции с последующей её параметризацией [8]. Использование данного метода рефакторинга позволяет максимально сохранить структуру кода для удобства будущей отладки, что является одним из требований к решению.

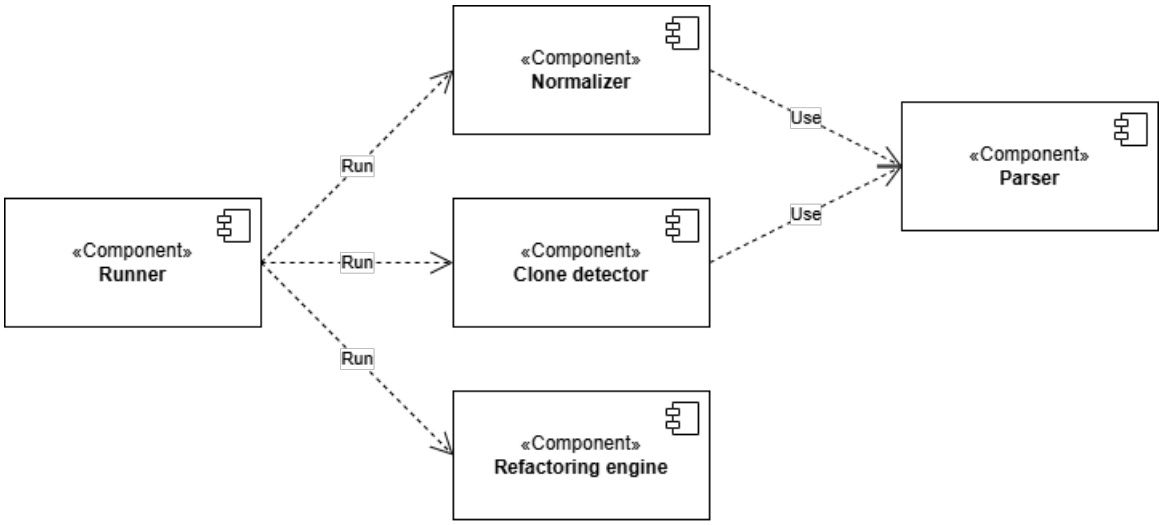

Рис. 2: Диаграмма компонент решения.

На рисунке 2 представлена структура предложенного решения — в виде диаграммы компонент UML [4], [24]. Кратко опишем эти компоненты.

- `Runner` — управляет всеми компонентами и потоком данных решения, отвечает за параллельную обработку Lua-файлов.
- `Clone detector` — первая из двух основных компонент, отвечает за поиск рефакто-ориентированных клонов T1 и T2 для языка программи-

рования Lua.

- `Refactoring engine` — вторая из основных компонент, реализует алгоритм однопроходного рефакторинга клонов, в том числе и вложенных. Также отвечает за генерацию вспомогательного модуля с новыми функциями.
- `Parser` — библиотека для разбора Lua-программ, предоставляет возможность получить лексическую и синтаксическую информацию.
- `Normalizer` — вспомогательная компонента, отвечающая за предварительную обработку исходного текста Lua-файлов для увеличения числа найденных клонов.

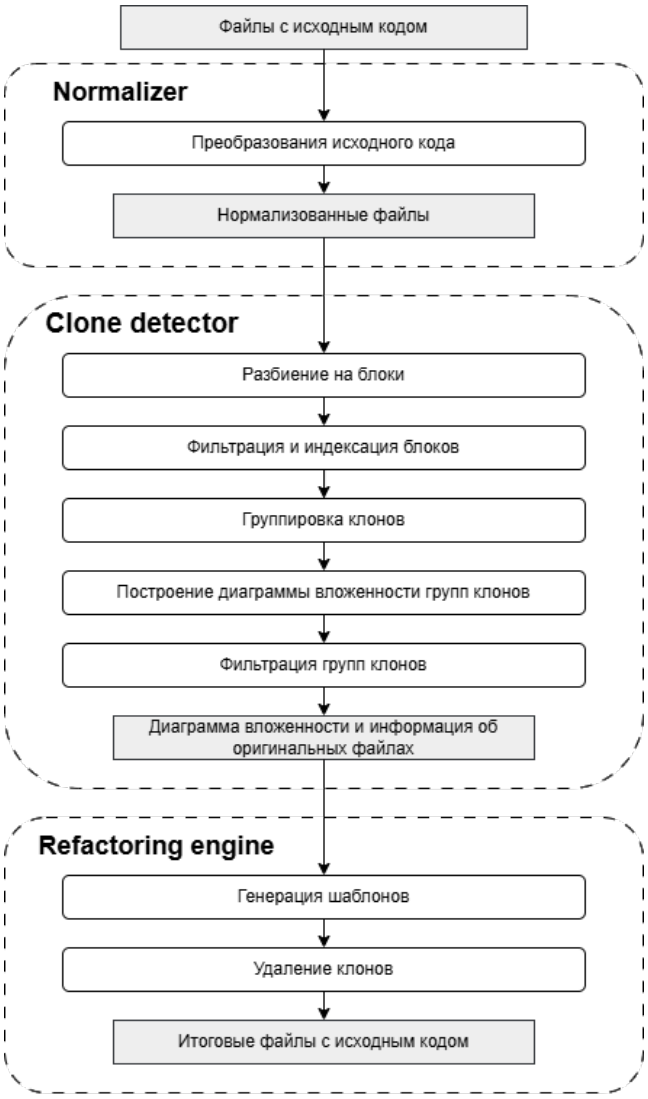

Рис. 3: Диаграмма рабочего процесса.

На рисунке 3 изображён сценарий работы решения — от получения на вход списка Lua-файлов до получения итоговых файлов после удаления клонов кода.

Компонента `Runner` управляет работой остальных компонент, за исключением парсера. Она отвечает за продвижение Lua-файлов по конвейеру обработки: нормализация, поиск клонов, рефакторинг. Если решение запущено в режиме работы с каждым файлом по отдельности, то можно добиться полной параллельности при обработке файлов. Если выбран анализ всего проекта, то поиск клонов и нормализация могут выполняться параллельно, в то время как рефакторинг может выполняться только в одном потоке, после получения всех предыдущих результатов.

Компонента нормализации исходного кода реализует различные трансформации, сохраняющие семантику, чтобы унифицировать код и, следовательно, увеличить число найденных клонов. Также она позволяет превратить часть клонов T3 в клоны T2, которые проще найти и для которых возможно выполнить рефакторинг.

Детектор клонов использует лексический подход, предложенный в работе [31], с добавлением синтаксической информации для поиска рефакто-ориентированных клонов T1 и T2. В начале работы для каждого файла строится дерево блоков кода. Для проверки возможности рефакторинга блока кода проверяются несколько условий, подробно описанных в следующей главе. После чего строится диаграмма групп блоков кода по отношению вложенности. В самом конце полученные группы фильтруются с точки зрения выгодности рефакторинга. Также дополнительно сохраняется вектор токенов каждого входного файла для последующего рефакторинга.

Компонента рефакторинга отвечает за удаление найденных групп клонов кода. В качестве метода рефакторинга используется извлечение функции вместе с её параметризацией, аналогичный метод используется в инструменте PyTeRor [32]. На основе информации о группе клонов генерируется шаблон общей функции, которая их заменит, с учётом информации о вложенных клонах. В конце все новые функции добавляются в соответствующие исходные файлы или новый модуль, а сами клоны заменяются на вызов соответствующей новой функции.

Библиотека для парсинга Lua используется внутри компоненты нормализации и детектора клонов. Она ориентирована на работу со всеми основными версиями Lua, её главной целью при разработке являлась корректность, т.к. она является базовым элементом решения. Для достижения данной цели было принято решение извлечь код лексера и парсера из кода интерпретатора Lua 5.3 в силу его понятности и небольшого размера. Такой подход даёт гарантии идентичности результатов с эталонной реализацией Lua. Единственными изменениями послужили добавление расширений из LuaJIT и Lua 5.4 на уровне лексера. Использование же существующих сторонних библиотек для парсинга Lua затруднительно, поскольку они написаны на нём самом, что накладывает ряд ограничений на производительность и взаимодействие с остальными компонентами.

4. Особенности реализации

Данная глава содержит информацию о реализации компонент, кратко описанных в предыдущей главе. Подробно описываются ключевые механизмы решения, а также соответствующая мотивация.

4.1. Нормализация

Современные исследования в области поиска клонов, в основном, сосредоточены на клонах T3 и T4, которые требуют больших усилий для рефакторинга и удаления, если это вообще возможно. Как показано в обзоре, наиболее быстрые детекторы клонов ограничены клонами T2, но дополнительный шаг предобработки может помочь преобразовать часть клонов T3 в клоны T2. В работе [53] описывается такой шаг для программ на Java, который назван *нормализацией*. Применить все преобразования нормализации, описанные в этой работе, невозможно из-за различий между Lua и Java, но существуют другие специфичные для Lua преобразования, которые позволяют добиться лучших результатов при поиске клонов.

1. **Удаление пустых `return` в конце функций.** По стандарту Lua отсутствие `return` в конце функции равносильно тому, что она не возвращает значений, поэтому такое преобразование абсолютно безопасно.

Оригинальный код	Нормализованный код

`function f(x)`
`  print(x)`
`  return`
`end`
	`function f(x)`
	`  print(x)`
	`end`

Таблица 7: Удаление пустого `return`.

2. **Удаление инициализации локальной переменной с помощью `nil`.** По стандарту каждая локальная переменная имеет начальное значение `nil`.

Оригинальный код	Нормализованный код

`function f()`
`  local ret = nil`
`  local err = nil`
`  ret, err = g()`
`  return ret`
`end`
	`function f(x)`
	`  local ret`
	`  local err`
	`  ret, err = g()`
	`  return ret`
	`end`

Таблица 8: Удаление ненужной инициализации.

3. **Удаление лишних точек с запятой.** Изначально точка с запятой была добавлена в синтаксис языка только для того, чтобы избежать двойственности во время цепочек вызовов. Однако на Lua часто пишут C/C++ программисты, они привыкли ставить точку с запятой в конце строки, перенося такой стиль написания кода на Lua-программы.

Оригинальный код	Нормализованный код

`local str = "Hello";`
`print(str);`
`(foo or bar)(str);`
	`local str = "Hello"`
	`print(str);`
	`(foo or bar)(str)`

Таблица 9: Удаление ненужных точек с запятой.

4. **Переименование меток.** Метки и, соответственно, оператор `goto` используются в Lua-коде достаточно часто и не являются антипаттерном. Из-за отсутствия в языке `continue` этот оператор часто эмулируется с помощью меток, которым присваиваются различные имена —`continue`, `CONTINUE`, `CON`. Аналогично языку C, метки могут использоваться для закрытия системных ресурсов при выходе из функции, т.к. аналогов деструкторов в языке нет (для решения этой проблемы в Lua 5.4 был добавлен атрибут `<close>`). Метки являются локальными для каждой функции, поэтому возможно их простое согласованное переименование.

Оригинальный код	Нормализованный код

`for i = 1,10 do`
`  if i == 5 then`
`  goto continue`
`  end`
`  print(i)`
`  ::continue::`
`end`
	`for i = 1,10 do`
	`  if i == 5 then`
	`  goto a`
	`  end`
	`  print(i)`
	`  ::a::`
	`end`

Таблица 10: Переименование меток.

5. **Удаление неиспользуемого кода.** Таким кодом являются неиспользуемые локальные переменные и их инициализация, неиспользуемые параметры функции, неиспользуемые локальные функции и метки, на которые нет перехода. Часто неиспользуемый код возникает при копировании, когда функциональность немного изменилась и часть логики программы стала излишней.

Оригинальный код	Нормализованный код

`function f(x, y, ...)`
`  local z`
`  local ret, tmp`
`  ret = x + x`
`  return ret`
`end`
	`function f(x)`
	`  local ret`
	`  ret = x + x`
	`  return ret`
	`end`

Таблица 11: Удаление неиспользуемого кода.

6. **Объединение объявлений локальных переменных.** Язык Lua позволяет множественные присваивания и объявления переменных. Объявление каждой новой переменной дётся с новой строки и затрудняет сопоставление, если порядок объявлений не совпадает, а использование каждый раз ключевого слова `local` расточительно с точки зрения размера кода.

Оригинальный код	Нормализованный код

`function f()`
`  local ret`
`  local err`
`  ret, err = g()`
`  if err ~= nil then`
`  return err`
`  end`
`  print(ret)`
`  return nil`
`end`
	`function f()`
	`  local ret, err`
	`  ret, err = g()`
	`  if err ~= nil then`
	`  return err`
	`  end`
	`  print(ret)`
	`  return nil`
	`end`

Таблица 12: Объединение объявлений локальных переменных.

7. **Преобразование обращения к полю таблицы в обращение по ключу.** Обращение к полю является синтаксическим сахаром для обращения по ключу, причём накладывается ограничение, что ключ должен быть корректным именем в Lua. Поэтому можно избавиться от этого синтаксического сахара для унификации.

Оригинальный код	Нормализованный код

`local tbl = {`
`  key = 1,`
`  x = 2`
`}`
`print(tbl.key)`
	`local tbl = {`
	`  ["key"] = 1,`
	`  ["x"] = 2`
	`}`
	`print(tbl["key"])`

Таблица 13: Преобразование полей в ключи.

Первые шесть преобразований являются опциональными, тогда как последнее применяется всегда, поскольку оно оказывает значительный эффект на число найденных клонов. Можно также отметить, что все преобразования за исключением последнего сами по себе уменьшают размер кода, что и является целью работы всего решения. Как уже было сказано, последнее

преобразование увеличивает размер кода, поэтому требуется в конце работы решения превратить обращение по ключу в обращение к полю, где это возможно, чтобы нивелировать данный эффект.

Входом компоненты нормализации служит файл с исходным кодом на языке программирования Lua. Парсер строит АСД на основе него, а затем в зависимости от проставленных пользователем опций применяются все или только часть преобразований. Выходом компоненты является нормализованный исходный текст файла. Можно заметить естественный параллелизм этапа нормализации по файлам, который позволяет ускорить обработку файлов.

4.2. Поиск клонов

В работе используется алгоритм лексического поиска клонов из инструмента Vuddy [31]. Однако были выполнены следующие значительные улучшения алгоритма: уменьшение гранулярности поиска клонов до уровня блоков, консистентное переименование переменных, дополнительная фильтрация найденных клонов (возможность и выгодность рефакторинга, размер, использование “опасной“ функциональности).

4.2.1 Разбиение на блоки

В начале работы алгоритма для анализируемой программы строится АСД. При этом извлекаемое из АСД дерево базовых блоков кода не является оптимальным с точки зрения поиска клонов кода. К блокам кода можно дополнительно добавить все виды циклов, условные выражения и выражения `do ... end`. При этом можно заметить, что добавленные выражения разбивают базовые блоки на части, которые тоже выделяются в отдельные блоки кода.

Такое преобразование сохраняет древовидную структуру блоков, а также следующее свойство: если блоки пересекаются, то один блок вложен в другой. Выделение дополнительных блоков позволяет находить большее число клонов. При построении дерева каждому блоку присваивается идентификатор на основе идентификатора файла и номера блока в файле, что гарантирует его уникальность. Каждый блок определяется номером первого и последнего токена в нём, а также хранит информацию о вложенных в него блоках.

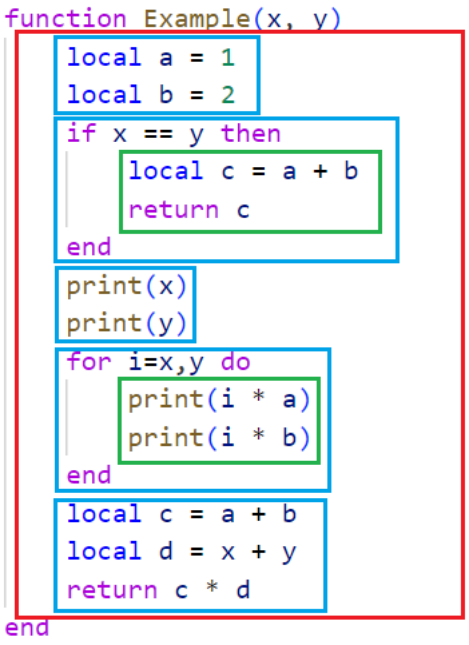

Рис. 4: Пример разбиения на блоки.

4.2.2 Фильтрация и индексация

По разным причинам не все блоки могут быть отрефакторены. Мы можем фильтровать их уже в момент поиска клонов, чтобы облегчить работу рефакторинга и уменьшить число данных, передающихся между компонентами. Для проверки возможности извлечения функции достаточно проверить несколько условий.

Фрагмент кода не должен содержать переходов вне этого блока. Такими переходами могут быть `return`, `break` или `goto`. Однако стоит отметить, что `break` может относиться к вложенному циклу и не влиять на поток управления. Аналогично, если `goto` ведёт в метку внутри блока и все переходы на метку идут изнутри блока, то это допустимо.

Следующим важным моментом является работа с локальными переменными. Внутри блока могут быть обращения к переменным, объявленным вне него, а переменные, объявленные в блоке, могут использоваться за его пределами. Причём в первом случае можно выделить два варианта: переменная объявлена вне блока, но внутри функции с блоком, или объявлена вне текущей функции. Информация о переменных последнего типа будет важна при

индексации, а информацию об остальных переменных достаточно сохранить для последующего рефакторинга.

С переменными имеются следующие особенности. В Lua есть возможность определять вложенные функции, которые могут менять локальные переменные в той функции, в которой они определены. Такие изменения невозможно отследить статически, поскольку такие функции могут быть присвоены глобальным переменным. Это, в свою очередь, может привести к ситуации, когда мы выделили новую функцию на месте блока кода и передали туда копию локальной переменной в расчёте на то, что она не будет меняться, а она изменилась из-за вызова глобальной функции где-то в глубине стека вызовов. Аналогичную проблему можно воспроизвести и на других языках — C++ или Kotlin. Поэтому было принято решение отбрасывать блоки, если функция, в которой они определены, содержит вложенные функции.

Аналогично, для безопасности будущего рефакторинга отбрасываются такие функции, в которых имеется явное обращение к окружению (`_ENV`, `_G`, `getfenv` или `setfenv`) или используется библиотека `debug`, которая предоставляет доступ к внутреннему состоянию интерпретатора. Также отбрасывается блок, если его размер меньше заданной пользователем нижней границы. Это оказывается целесообразным, чтобы не находить слишком маленькие клоны, рефакторить которые малоосмысленно.

После этапа фильтрации нужно проиндексировать оставшиеся блоки. Для этого все литералы блока (строки, числа, `true`, `false`, `nil`) заменяются на `LITERALi`, где `i` — порядковый номер. Аналогично все обращения к глобальным переменным заменяются на `GLOBALi`. Для локальных переменных процедура отличается тем, что все использования одной локальной переменной заменяются на одно и тоже новое имя, чтобы переименование локальных переменных между клонами было согласованно. Исключением из этого правила являются только локальные переменные, объявленные вне функции (upvalues) — их имена сохраняются. Таким образом, полученное переименование сохраняет граф потока управления внутри блока, а также структуру локальных переменных (в том числе и захваченных), но позволяет вариации в литералах и глобальных переменных. Данный факт нам потребуется при последующем рефакторинге.

Оригинальный код	Переименованный код

`local x, y = 1`
`print(x)`
`print(y)`
`if y == nil then`
`  y = 0`
`end`
`local z = {}`
`z["key"] = x`
	`local LOCAL1, LOCAL2 = LITERAL1`
	`GLOBAL1(LOCAL1)`
	`GLOBAL2(LOCAL2)`
	`if LOCAL2 == LITERAL2 then`
	`  LOCAL2 = LITERAL3`
	`end`
	`local LOCAL3 = {}`
	`LOCAL3[LITERAL4] = LOCAL1`

Таблица 14: Пример переименования.

Вместе с переименованием выполняется сохранение информации об изначальных именах и значениях для последующего рефакторинга. Дополнительно собирается информация об использовании специальных переменных: `self` и эллипсис (`...`). В конце генерируется уникальный отпечаток на основе обработанных токенов без учёта комментариев, пробелов и переводов строк. Если были upvalues, то информация об их положении в файле и хеш-сумма файла учитываются при генерации уникального отпечатка, чтобы разделять upvalues с одинаковыми именами, но из разных файлов, или из одного файла, но из разных мест.

Для того чтобы избежать дублирования информации между блоками об исходных именах и значениях заменённых переменных и литералов, структура, представленная на рисунке 5, хранит номер токена в изначальном файле, которому они соответствовали (`Locals`, `Globals`, `Literals`). Дополнительно хранится информация о внешних переменных (`Params`), использующихся внутри блока, информация о переменных (`InsideLocals`), объявленных внутри блока и использующихся вне этого блока, информацию о внешних локальных переменных (`OutsideLocals`), в которые было присваивание внутри блока. Эта информация представлена в виде индексов соответствующих переменных в векторе `Locals`. В поле `Children` хранятся идентификаторы детей с учётом переподвешивания после фильтрации части блоков. По ним можно восстановить всё дерево блоков.

```
1  struct Block {
2    size_t ID;
3    std::string Fingerprint;
4    int Start;
5    int End;
6    std::vector<size_t> Children;
7    std::vector<std::vector<int>> Locals;
8    std::vector<int> Globals;
9    std::vector<int> Literals;
10   std::vector<int> Params;
11   std::vector<int> InsideLocals;
12   std::vector<int> OutsideLocals;
13   bool Self;
14   bool Ellipsis;
15   bool Return;
16 };
```

Рис. 5: Информация о блоке.

4.2.3 Построение диаграммы вложенности клонов

Если у двух блоков имеются одинаковые отпечатки, то они объединяются в одну группу. По построению, если два блока попали в одну группу, то их дети тоже попадут в одну группу. Можно заметить, что полученные таким образом группы клонов образуют частично-упорядоченное множество (ЧУМ) по отношению вложенности. Естественным образом изобразить ЧУМ является диаграмма Хассе. Мы будем поддерживать схожую структуру данных, которая более подробно описана ниже, для удобства будущего рефакторинга. Однако, в отличие от диаграммы Хассе, ребро будет проведено между двумя группами тогда и только тогда, когда существует блок из первой группы, являющийся родителем блока из второй группы.

Каждая группа хранит указатели на группы родителей блоков (`Parents`), входящих в неё. Причём ребра, задаваемые этими указателями, могут быть

кратными. Это нужно для обработки случаев, когда у блока есть несколько одинаковых подблоков, чтобы учитывать число таких повторов. Также сохраняется информация об идентификаторах блоков, которые образуют группу; она сохраняется в двух множествах — `TopLevelIDs`, `LowLevelIDs`. В первом хранятся идентификаторы блоков без родителей, а во втором все остальные. Наконец, также сохраняются отображения из идентификатора блока в группу и из отпечатка в группу.

```
1 struct Group {
2   std::unordered_set<size_t> TopLevelIDs;
3   std::unordered_set<size_t> LowLevelIDs;
4   std::unordered_multiset<const Group*> Parents;
5 };
```

Рис. 6: Информация о группе.

Алгоритм построения диаграммы вложенности заключается в следующем. Новые блоки добавляются по файлам. Блоки в пределах одного файла добавляются в порядке обратного обхода дерева блоков. Если по отпечатку блока уже существует группа, то он добавляется в неё, а если нет, то создаётся новая группа и он добавляется в неё, причём всегда в `TopLevelIDs`. Затем просматриваются группы, соответствующие его детям, и идентификаторы его подблоков перемещаются в `LowLevelIDs` из `TopLevelIDs`, т.к. обход дерева обратный, то его дети уже были добавлены, что обеспечивает корректность.

После конца обработки всех файлов происходит сжатие. Существует большое число групп, в которые попал только один блок (то есть у него нет дубликатов). Мы бы хотели избавиться от них и освободить память. По построению, если в группу попало больше одного блока, то и во вложенные в неё группы попало больше одного блока. Поскольку если блок попал в группу, то его подблоки попадали во вложенные подгруппы. Тогда поиск таких групп можно начать с тех групп, у которых нет родителей. Для всех детей единственного блока перемещаем их идентификаторы из `LowLevelIDs` в `TopLevelIDs` соответствующих групп, а также удаляем группу из родителей. Затем рекурсивно запускаемся от групп детей, если в них только один блок. Тогда в

результате сжатия останутся только группы, соответствующие клонам. Потому что если в группе один блок, то у него максимум одна родительская группа, в которой тоже один блок. Такой цепочкой можно дойти до группы, у которой изначально не было родителей, с которой мы начали процесс сжатия.

4.2.4 Фильтрация групп клонов

Найденные ранее клоны уже проверены на возможность рефакторинга. Однако не все найденные клоны выгодно рефакторить с точки зрения уменьшения размера кода. Некоторые же клоны нельзя отрефакторить безопасно из-за ограничений интерпретатора. Давайте научимся определять, является ли группа подходящим кандидатом на рефакторинг.

После сжатия, произведённого на предыдущем шаге, в каждой группе есть хотя бы два блока. Однако не все группы нужно рефакторить. Единственным исключением являются группы с одним родителем и без блоков без родителей. Такие группы состоят из блоков, чьи родительские блоки лежат в одной группе, то есть все блоки из неё являются частью одного большего клона и не имеет смысла их рефакторить отдельно.

Поскольку выбранный способ рефакторинга — это извлечение кода в функции, то у новой функции может быть большое число параметров. Это, в свою очередь, влияет на читаемость полученного кода. Кроме того, из-за ограничений Lua-интерпретатора не может быть больше 250 локальных переменных, в том числе параметров у функций. Поэтому было принято решение ограничить максимальное число новых параметров до 25.

Также для фильтрации может применяться эвристика, позволяющая оценить число удалённых и добавленных токенов при рефакторинге. В число удалённых токенов входят токены блоков текущей группы с учётом рефакторинга вложенных блоков. В число добавленных токенов входят токены новой функции и токены на вызов этой функции вместо удалённых блоков.

4.3. Автоматический рефакторинг клонов

Данная задача успешно решена на уровне функций для программ на языках Python [32] и Java [72]. Однако при переходе на уровень блоков воз-

никают новые проблемы. В этом случае требуется проверять возможность выделения блока в новую функцию. Также клоны могут пересекаться, и надо это учитывать при рефакторинге. Простое итеративное решение в виде рефакторинга одной группы клонов с последующим повторным запуском процесса поиска клонов, описанное в [52], является неоптимальным с точки зрения времени работы. Знание о том, что клоны, найденные на предыдущем шаге, не могут пересекаться, кроме вложенных, позволяет построить однопроходный алгоритм рефакторинга, лишённый описанных выше проблем.

4.3.1 Генерация шаблонов

Целесообразно сначала сгенерировать новые функции для выбранных групп клонов и только после этого выполнять рефакторинг. Такие функции будем называть *шаблонами*. Генерация шаблонов происходит рекурсивно, начиная с тех групп, у которых нет родителей. При первом заходе в такую группу выбирается мастер-блок, на основе которого и будет генерироваться шаблон.

```
1  def visit(block):
2    group = getGroupByBlock(block)
3    if visited[group]:
4      return [block]
5    refactored_blocks = []
6    for child in block.children:
7      refactored_blocks.append(visit(child))
8    if filtered[group]:
9      return refactored_blocks
10   generate_template(block, refactored_blocks)
11   visited[group] = True
12   return [block]
```

Рис. 7: Псевдокод алгоритма обхода диаграммы вложенности клонов.

При рефакторинге учитывается то, как отрефакторены вложенные клоны. Функция `visit` возвращает список самых верхних отрефакторенных бло-

ков в поддереве, задаваемом `block`. Если группа, в которой лежит блок, была отфильтрована, то в качестве результата возвращается объединение результатов для детей этого блока, т.к. они или их дети могли быть отрефакторены. Если группа может быть отрефакторена, то при первом заходе в неё генерируется шаблон для группы с учётом отрефакторенных детей и возвращается идентификатор текущего блока, поскольку он был отрефакторен. При повторном заходе сразу возвращается идентификатор блока.

Опишем генерацию шаблона более подробно. При переименовании мы заменяли литералы, локальные и глобальные переменные на новые переменные. Только в этих местах блоки могут отличаться в пределах группы. Заметим, что переименование локальных переменных было согласованно, и их можно не трогать, т.к. их исходные имена нам не важны. Однако опционально можно восстановить имена, если они совпадали между всеми блоками в группе, для удобства отладки. Осталось разобраться с глобальными переменными и литералами.

Если для определённой позиции совпадают исходные значения литералов между всеми блоками в группе, то можно просто сохранить исходное значение из мастер-блока в шаблоне. Если же значения различаются, то надо параметризовать разницу между различными блоками добавлением нового параметра в шаблон. Можно заметить, что такой алгоритм приводит к избыточному числу параметров шаблона. Для оптимизации их числа можно объединить несколько параметров в один. Это можно сделать, если для каждого блока в группе исходные значения, соответствующие этим параметрам, были равны.

Работа с глобальными переменными немного отличается. Как уже было описано в обзоре языка Lua, обращение к глобальной переменной является лишь синтаксическим сахаром для выражения `_ENV["name"]`. То есть достаточно передать в новую функцию окружение исходной функции и имя глобальной переменной, а затем обратиться к ней через данное окружение по имени. Также необходимо передавать в шаблон `self` и эллипсис, если они использовались в блоке. Аналогично литералам, в данном случае можно применить оптимизацию, позволяющую сократить число новых параметров шаблона.

Оригинальный код	Отрефакторенный код

`do`
`  local x = 1`
`  print(x)`
`end`
`do`
`  local y = 2`
`  log(y)`
`end`
	`function new(_ENV, GLOBAL1, LITERAL1)`
	`  local LOCAL1 = LITERAL1`
	`  _ENV[GLOBAL1](LOCAL1)`
	`end`
	`do`
	`  new(_ENV, "print", 1)`
	`end`
	`do`
	`  new(_ENV, "log", 2)`
	`end`

Таблица 15: Пример рефакторинга с глобальными переменными и литералами.

Вернёмся к локальным переменным. Как уже было описано, с точки зрения рефакторинга их можно разделить на две группы: локальные переменные, объявленные вне блока, но использующиеся в нём; локальные переменные, объявленные в блоке, но использующиеся после него.

Переменные первой группы должны стать новыми параметрами шаблона, т.к. их значения используются внутри блока. Однако, если в такую переменную было выполнено присваивание, то из функции требуется вернуть новое значение и восстановить его в месте вызова. Поскольку Lua позволяет множественный `return`, то это не является проблемой. Стоит отметить также и то, что переменные из первой группы совпадают для всех блоков и легко определяются из синтаксиса программы, поэтому вся необходимая информация уже имеется после индексации блока при поиске клонов.

Для переменных из второй группы ситуация отличается. Поскольку они определены внутри блока, то мы не должны создавать для них новых параметров, но мы все ещё должны возвращать их значения из функции. Однако то, какие переменные используются после блока, зависит от кода вокруг него. Поэтому для определения минимального необходимого множества таких переменных необходимо выполнить объединение по всем блокам в группе. Соответственно, информацию о таких переменных можно хранить

только вместе с информацией о сгенерированном шаблоне. Перед вызовом шаблона надо определить такие переменные, а после вызова новой функции присвоить значения в них.

Оригинальный код	Отрефакторенный код

`local x, y = 1, 2`
`if cond() then`
`  x = x * 2`
`  y = y + 1`
`end`
`print(x + y)`
	`local function new(LOCAL1, LOCAL2)`
	`  LOCAL1 = LOCAL1 * 2`
	`  LOCAL2 = LOCAL2 + 1`
	`  return LOCAL1, LOCAL2`
	`end`
	`local x, y = 1, 2`
	`if cond() then`
	`  x, y = new(x, y)`
	`end`
	`print(x + y)`

Таблица 16: Пример рефакторинга с возвращением значений из функции.

В вышеприведённых примерах при генерации вызова шаблона использовались оригинальные значения переменных и литералов, но это невозможно при рефакторинге вложенных клонов. При генерации шаблона для группы часть вложенных блоков могла уже быть отрефакторена и для них сгенерированы шаблоны, поэтому при генерации вызова их шаблона нужно вместо подстановки изначальных значений использовать новые переменные, полученные при параметризации ранее.

Ещё одной проблемой является именование новых функций. При рефакторинге вложенных клонов мы уже должны знать имена функций, соответствующих им, чтобы сгенерировать код для их вызова. Причём если в группе все блоки из одного файла, то шаблон можно сделать локальной функцией в этом файле. А если есть в группе есть блоки из разных файлов, то шаблон должен находиться в общем внешнем модуле. Для того чтобы решить данную проблему, в пределах каждого файла шаблоны последовательно нумеруются (также отдельно нумеруются шаблоны для межфайловых клонов), что позволяет легко сгенерировать соответствующее имя при вызове.

```
1 struct Template {
2   int ID;
3   bool Extern;
4   std::vector<int> GlobalParams;
5   std::vector<int> LiteralParams;
6   std::vector<int> Declarations;
7 };
```

Рис. 8: Информация о шаблоне.

Поле `Extern` описывает ожидаемое расположение шаблона в исходном коде: внутри файла или в новом модуле. Следующие два поля хранят индексы глобальных переменных и литералов, которые стали новыми параметрами шаблона. В поле `Declarations` хранятся индексы локальных переменных блока, которые могут использоваться после него.

Стоит отметить, что при генерации шаблонов мы неявно удалили клоны блоков из `LowLevelIDs`, поскольку они были частью больших клонов. Однако остались блоки из `TopLevelIDs`, для которых алгоритм действий отличается.

4.3.2 Удаление клонов

До этого шага никаких реальных преобразований исходного кода не происходило. По результатам предыдущего шага у нас есть шаблон для каждой группы, и теперь осталось обработать правильно блоки, чьи идентификаторы лежат в `TopLevelIDs`. Мы имеем позиции начала и конца этих блоков в исходном тексте, поэтому не составляет труда заменить эти фрагменты на вызов соответствующих функций, как уже было описано ранее. Осталось только найти, где расположить новые функции.

Как уже было сказано, если все блоки в группе находятся в одном файле, то генерируется локальная функция для этого файла. Поскольку Lua обеспечивает лексическую область видимости, то достаточно поместить новую функцию в самом начале файла. Однако особенность заключается в том, что функция должна быть объявлена после объявлений всех захваченных изначальной функцией внешних локальных переменных (`upvalues`), чтобы про-

извести их корректный захват. Поэтому целесообразно разместить её перед самым определением первой функции, содержащей отрефакторенный блок. Такой подход имеет единственную проблему: он не может корректно обработать рекурсивные локальные функции, поскольку они захватывают сами себя. Чтобы решить эту проблему, достаточно поместить объявление локальной переменной, соответствующей такой функции, перед новыми функциями, а из определения убрать ключевое слово `local`. Осталось заметить, что нужно при вставке шаблонов сохранять порядок от меньшего к большему, т.к. в процессе рефакторинга вложенных клонов, при генерации шаблона, выполняется вызов соответствующей новой функции.

Новый модуль	Отрефакторенный код

`local GENERATED`
`GENERATED = {`
`  function (_ENV, GLOBAL1)`
`  local LOCAL1 =`
`  _ENV[GLOBAL1]()`
`  GENERATED[2](LOCAL1)`
`  end,`
`  function (_ENV, LOCAL1)`
`  if LOCAL1 then`
`  print(LOCAL1)`
`  end`
`  end`
`}`
`return GENERATED`
	`local GENERATED =`
	`  require("generated")`
	
	`function f()`
	`  GENERATED[1]("foo")`
	`end`
	
	`function g(x)`
	`  GENERATED[2](x)`
	`  local y = not x`
	`  return y`
	`end`

Таблица 17: Пример сгенерированного кода.

При работе с межфайловыми клонами ситуация упрощается, поскольку единственная возможность — это поместить их в новый модуль. Однако наивная реализация приводит к проблемам с множеством новых глобальных имён, которые должны быть уникальными и желательно короткими. Чтобы избежать этих проблем, модуль с новыми функциями просто экспортирует

таблицу с этими функциями, из которой можно получить нужную функцию по её номеру. Модуль загружается в самом начале каждого файла, где был найден межфайловый клон, а ссылка на таблицу с сгенерированными функциями из нового модуля сохраняется в локальную переменную, чтобы ускорить обращения к нужным функциям.

Заметим, что внутри межфайлового клона могут быть только другие межфайловые клоны. Поэтому обращение к ним будет также происходить через общую таблицу, что обеспечивает корректность. Поскольку идентификаторы клонов являются последовательными числами, начинающимися с единицы, то имеется возможность использовать краткую форму записи таблиц в Lua, которая задаёт массив, что позволяет дополнительно сэкономить место.

В конце хочется отметить компромиссы при рефакторинге клонов. Несмотря на то, что алгоритм рефакторинга не зависит от гранулярности клонов и области их поиска, полученный в результате код может отличаться по эффективности. Если рефакторинг клонов происходит в пределах одного файла и клоны ищутся на уровне функций, то полученный код будет практически не отличим по производительности от оригинального. Это обеспечивается тем, что новые функции будут локальными и их вызовы происходят достаточно быстро. Также Lua поддерживает оптимизацию хвостового вызова, что помогает избежать выделение лишнего кадра стека при вызове новой функции вместо тела оригинальной. При переходе к рефакторингу клонов на уровне блоков мы лишаемся этой оптимизации, что может привести к значительному увеличению числа кадров стека и связанных с этим накладных расходов. При переходе к рефакторингу межфайловых клонов возрастают накладные расходы на вызов новых функций, т.к. ведётся их поиск в общей таблице по индексу.

Похожая проблема была отмечена в работе [58], где описывается алгоритм объединения функций для уменьшения размера исполняемого файла. Среднее замедление работы составляет менее 5%, однако для некоторых проектов оно может достигать 20%. Такое падение производительности вызвано проблемами при работе с кэшем процессора и частыми ошибками предсказателя переходов на сгенерированном коде. Поэтому в общем случае оптимиза-

ции с точки зрения размера должны подвергаться только функции, лежащие на, так называемом, “холодном” пути исполнения. Обычно Lua не используется в высоконагруженных приложениях, поэтому описанные выше проблемы не должны проявляться.

5. Экспериментальное исследование

В данной главе описывается инфраструктура экспериментального исследования созданного решения, поставленные исследовательские вопросы и метрики для них, а также полученные результаты и их интерпретация.

5.1. Инфраструктура экспериментов

Для выполнения экспериментов было выбрано несколько Lua-проектов. Первая группа состоит из двух анонимизированных проектов крупной телекоммуникационной компании: CE6866 [15] и VRP [27]. Вторую группу составляют Lua-проекты с открытым исходным кодом Xmake [70], Lua Language Server [44], Kong [33], APISIX [11].

Проект	Размер, Мб	Число файлов	Число строк	Версия Lua	Число тестов
CE6866	216.41	22736	5593690	Lua 5.3, LuaJIT	—
VRP	220.23	23437	5385891	Lua 5.3, LuaJIT	—
Xmake	5.7	1477	101920	Lua 5.4, LuaJIT	164
Lua Language Server	1.56	242	47823	Lua 5.3	2257
Kong	3.03	549	74092	LuaJIT	2989
APISIX	1.65	264	39976	LuaJIT	5746

Таблица 18: Проекты, выбранные для тестирования.

Проект CE6866 является частью прошивки промышленных сетевых коммутаторов. Проект VRP является частью операционной системы для сетевых устройств. Следующий проект, Xmake, является кроссплатформенным средством сборки проектов на C/C++ и имеет более 10 тысяч звёзд на GitHub. Языковой сервер (Language Server) для Lua написан на нём самом и является активным проектом на GitHub, имеющим на текущий момент 3.5 тысячи звёзд. Последние два проекта относятся к области API-шлюзов и оба активно разрабатываются, имея 40 и 14.8 тысяч звёзд соответственно. Также в таблице предоставлены данные о количестве тестов в исследуемых проектах

(за исключением закрытых проектов в силу их особенностей), которые будут нужны для ответа на один из исследовательских вопросов.

Дополнительно для анализа был выбран проект Lua Benchmarks [43], который является набором Lua-программ для тестирования производительности различных версий интерпретатора Lua. Поскольку данный проект используется для ответа только на один исследовательский вопрос, то он не представлен в таблице 18 наравне с остальными.

Эксперименты производились на рабочей станции со следующими характеристиками: Windows 11 (WSL2), Intel Core i7-8565U, RAM 16 Гб.

5.2. Исследовательские вопросы

В рамках экспериментального исследования были сформулированы следующие исследовательские вопросы.

RQ1: Насколько предложенное решение позволяет уменьшить размер кода? Этот вопрос является ключевым, поскольку именно уменьшение размера исходного кода является основной целью решения.

RQ2: Как влияют различные виды нормализации на количество найденных клонов? Дополнительный шаг нормализации кода позволяет перевести часть клонов T3 в клоны T2 для последующего рефакторинга. В рамках данного вопроса хочется оценить результаты применения разных видов нормализации с точки зрения поиска клонов.

RQ3: Является ли рефакторинг клонов эквивалентным преобразованием? Данный вопрос особенно важен для планируемого промышленного применения решения, поскольку недопустимо внесение ошибок в полученный код.

RQ4: Как область анализа (файл, проект) и гранулярность поиска клонов (функция, блок) влияют на результаты работы решения? Этот вопрос нацелен на изучение целесообразности и значимости данных настроек решения.

RQ5: Как предложенное решение влияет на производительность целевых приложений? Целью данного вопроса является исследование замедления производительности целевого приложения в связи с созданием но-

вых функций в ходе рефакторинга: априори, запроцедуривание фрагментов кода ухудшает производительность приложения.

5.3. Метрики

Чтобы ответить на первый вопрос, необходимо измерить процент уменьшения размера кода после рефакторинга. Наиболее важны результаты для проектов CE6866 и VRP, поскольку они являются целевыми для решения.

Для ответа на второй вопрос требуется измерить, насколько процентов изменится число найденных клонов в зависимости от опций нормализации. Для измерения был выбран проект CE6866, поскольку он является целевым и полученные на нём результаты будут более репрезентативными с точки зрения заинтересованных сторон.

Для оценки корректности решения (третий вопрос) необходимо определить количество непройденных тестов к целевому приложению после выполнения рефакторинга. Поскольку есть проблемы с полным тестовым покрытием целевой кодовой базы, то использовались открытые Lua-проекты (Xmake, Lua Language Server, Kong, Apisix), имеющие хороший набор тестов. Для целевой кодовой базы (проекты VRP и CE6866) проверка корректности результатов была выполнена вручную для отдельных фрагментов.

В свете четвёртого вопроса наиболее подходящими метриками являются число найденных клонов и процент уменьшения размера кода, а также время работы и потребление памяти. Первые две метрики отвечают за поиск и рефакторинг клонов соответственно, тогда как две последние важны для оценки применимости решения для целевых проектов, а также помогают оценить качество программной реализации алгоритмов.

Для ответа на последний вопрос требуется более чётко обозначить контекст использования Lua. Часто Lua используется как встраиваемый язык для C/C++ приложений, поэтому достаточно сложно в унифицированной манере замерить время работы Lua-части и общее замедление приложения, поскольку происходит постоянное взаимодействие между языками в обе стороны. Однако есть много приложений, которые написаны только на Lua. Чтобы оценить их замедление, можно использовать стандартные бенчмарки [43]. Необходи-

мо измерить время их прохождения при условии применения предложенных методов рефакторинга для всех сущностей (функций и блоков) вне зависимости от того, являются ли они клонами. Это позволит дать строгую верхнюю оценку на потенциальное замедление, вызванное рефакторингом клонов.

5.4. Результаты

Перейдём теперь к описанию результатов экспериментального исследования в соответствии с поставленными вопросами. Однако вначале введём некоторые общие обозначения: Func — поиск клонов выполняется на уровне функций, Block — на уровне блоков кода. Single — область поиска составляет один Lua-файл, Cross — все файлы проекта.

RQ1: Насколько предложенное решение позволяет уменьшить размер кода?

Проект	Базовый размер, Кб	Режим	Новый размер, Кб	Δ, %	Время работы, с	Потребление памяти, Мб
CE6866	129005	Single + Func	127820	-0.919	35	735
		Cross + Func	125102	-3.025	40	2917
		Single + Block	125940	-2.375	39	743
		Cross + Block	122019	-5.415	62	3704
VRP	139473	Single + Func	138008	-1.050	40	665
		Cross + Func	134339	-3.681	45	3101
		Single + Block	134642	-3.463	43	670
		Cross + Block	129357	-7.253	70	3935
Xmake	2255	Single + Func	2238	-0.741	1.2	58
		Cross + Func	2184	-3.149	1.4	86
		Single + Block	2213	-1.847	1.3	59
		Cross + Block	2137	-5.215	1.5	110
Lua Language Server	748	Single + Func	747	-0.185	0.5	48
		Cross + Func	746	-0.287	0.6	65
		Single + Block	743	-0.713	0.5	57
		Cross + Block	742	-0.783	0.6	68
Kong	1461	Single + Func	1460	-0.078	0.7	51
		Cross + Func	1459	-0.098	0.9	63
		Single + Block	1451	-0.685	0.8	56
		Cross + Block	1449	-0.828	1.0	64

APISIX	730	Single + Func	729	-0.136	0.3	42
		Cross + Func	729	-0.135	0.3	43
		Single + Block	725	-0.672	0.4	42
		Cross + Block	725	-0.647	0.4	43

Таблица 19: Результаты уменьшения размера кода.

На основе данных, представленных в таблице 19, можно ответить на данный вопрос. Удалось добиться уменьшения размера кода на 5-7% для первых трёх проектов, включая целевые. Однако для других трёх проектов получилось добиться уменьшения только на 0.7% в среднем. Это отчасти ожидаемое явление, т.к. процент уменьшения кода пропорционален числу клонов в проекте. Как показано в таблице 21, последние три проекта имеют наименьшее число клонов (<0.5 клона на Кб исходного кода) относительно размера по сравнению с остальными проектами (>1.5 клона на Кб исходного кода).

RQ2: Как влияют различные виды нормализации на количество найденных клонов?

Режим	Изначальное число клонов	Опции нормализации	Новое число клонов	Δ, %
Single + Func	14488	returns	14494	+0.04
		nils	14501	+0.09
		semicolons	14668	+1.24
		labels	14488	+0.00
		unused	14509	+0.14
		join	14496	+0.06
		all	14717	+1.58
Cross + Func	38881	returns	38902	+0.05
		nils	38987	+0.27
		semicolons	39427	+1.40
		labels	38881	+0.00
		unused	38993	+0.29
		join	38921	+0.10
		all	39715	+2.15

Single + Func	104812	returns	104829	+0.02
		nils	104754	-0.05
		semicolons	104953	+0.13
		labels	104812	+0.00
		unused	104711	-0.09
		join	104856	+0.04
		all	104894	+0.08
Cross + Block	225855	returns	225973	+0.05
		nils	225731	-0.05
		semicolons	227883	+0.90
		labels	225864	+0.003
		unused	225705	-0.06
		join	226487	+0.28
		all	228556	+1.20

Таблица 20: Результаты нормализации для проекта CE6866 [15].

Решение предоставляет шесть различных трансформаций для нормализации: удаление лишних `return` (return), необязательной инициализации (nils), лишних “;” (semicolons), неиспользуемого кода (unused); преобразование обращения по полю в обращение по ключу; переименование меток (labels); объединение объявлений переменных (join). Также их все можно применять одновременно (all). Использование нормализации позволяет увеличить число найденных клонов на 1-2% в зависимости от режима работы.

Хочется отметить, что некоторые опции нормализации иногда приводят к незначительному снижению числа найденных клонов. Это обусловлено тем, что при использовании соответствующих трансформаций кода удаляется большое число токенов, из-за чего полученные клоны становятся меньше, чем установленный нижний лимит, и отбрасываются. Поэтому важен правильный подбор минимального размера клона для каждого проекта.

RQ3: Является ли рефакторинг клонов эквивалентным преобразованием для рассматриваемых проектов?

Работа решения была проверена на тестах открытых проектов с различными опциями конфигурации: режим работы, опции нормализации, минимальный размер клона и т.д. Поскольку время работы решения на открытых проектах относительно мало, то возможно перебрать большое число воз-

можных конфигураций за разумное время. По результатам работы все тесты исследуемых проектов были успешно пройдены для всех протестированных конфигураций решения. Также было вручную проверено более 100 групп клонов из закрытых проектов. На основе данных результатов можно сделать вывод о корректности поиска и рефакторинга клонов.

RQ4: Как область анализа (файл, проект) и гранулярность поиска клонов (функция, блок) влияют на результаты работы решения?

Проект	Режим	Число групп клонов	Общее число клонов
CE6866	Single + Func	5432	14488
	Cross + Func	12505	38881
	Single + Block	37407	104812
	Cross + Block	53910	225855
VRP	Single + Func	5329	14050
	Cross + Func	12855	40626
	Single + Block	41140	118933
	Cross + Block	54896	247409
Xmake	Single + Func	129	340
	Cross + Func	319	941
	Single + Block	640	1673
	Cross + Block	1055	3596
Lua Language Server	Single + Func	22	51
	Cross + Func	27	66
	Single + Block	124	298
	Cross + Block	133	335
Kong	Single + Func	13	31
	Cross + Func	15	41
	Single + Block	154	390
	Cross + Block	172	455
APISIX	Single + Func	9	19
	Cross + Func	10	21
	Single + Block	111	257
	Cross + Block	122	285

Таблица 21: Результаты поиска клонов.

Таблица 21 показывает результаты поиска клонов для тестовых проектов. Уменьшение гранулярности поиска с функций до блоков увеличивает число найденных клонов в среднем более чем в 5 раз. Расширение области

анализа на весь проект повышает их количество в среднем в 2 раза. Рефакторинг также демонстрирует улучшение результатов в 2-3 раза при увеличении области анализа и уменьшении гранулярности.

Можно заметить, что большинство клонов кода находится на уровне блоков, поскольку дублирование целых функций происходит не так часто. Также достаточно много клонов находится внутри одного файла, тогда как число межфайловых клонов сильно колеблется в зависимости от проекта. Поскольку в Lua часто один файл — это один модуль, то копирование кода в пределах одного модуля является относительно частой практикой (например, какие-то проверки или преобразования данных), что и является причиной. С другой стороны, копирование кода между модулями уже зависит от специфики проекта: много видов сетевых протоколов и их версий (CE6866, VRP) или поддержка большого числа платформ и языков (Xmake).

Обсудим производительность и потребление памяти нашего решения. Эти параметры зависят от области анализа, поскольку при межфайловом анализе нужно хранить информацию обо всём проекте в оперативной памяти. Уменьшение гранулярности увеличивает количество найденных клонов, а значит, время работы и потребление памяти. Однако даже для самых крупных проектов время работы составляет менее полутора минут при потреблении памяти менее 4 ГБ.

RQ5: Как влияет предложенное решение на производительность целевых приложений?

Режим	Lua 5.1	Lua 5.2	Lua 5.3	Lua 5.4	LuaJIT 2.1 interpreter	LuaJIT 2.1
Vanilla	43.22 с.	41.99 с.	35.71 с.	24.86 с.	15.12 с.	5.39 с.
Single + Func	45.17 с. (+4.51%)	43.11 с. (+2.67%)	36.59 с. (+2.46%)	25.88 с. (+4.10%)	15.78 с. (+4.36%)	5.39 с. (+0.00%)
Cross + Func	45.36 с. (+4.95%)	43.54 с. (+3.69%)	36.74 с. (+2.88%)	25.84 с. (+3.94%)	15.76 с. (+4.23%)	5.41 с. (+0.37%)
Single + Block	57.05 с. (+31.99%)	55.14 с. (+31.32%)	43.79 с. (+22.63%)	32.89 с. (+32.30%)	18.38 с. (+21.56%)	5.61 с. (+4.08%)
Cross + Block	45.17 с. (+36.26%)	43.11 с. (+34.69%)	44.96 с. (+25.90%)	34.03 с. (+34.88%)	19.04 с. (+25.92%)	5.62 с. (+4.27%)

Таблица 22: Результаты бенчмарка.

Можно заметить большие различия в зависимости от версии Lua и режима работы решения. Замедление от рефакторинга на уровне функций составляет $< 5\%$, что является приемлемым результатом. Однако рефакторинг на уровне блоков приводит к уже более существенному замедлению ($\approx 25\%$). Полученная верхняя оценка для рефакторинга всех сущностей (функций или блоков) может быть перенесена на рефакторинг клонов. Можно предположить, что процент замедления прямо пропорционален проценту отрефакторенного кода. Основываясь на собственном эксперименте (**RQ1**) и работе [52], можно сделать вывод о том, что среднее число клонов, которые можно удалить, составляет менее десяти процентов от всей кодовой базы. Значит, для усреднённого проекта реальное замедление составит, скорее всего, несколько процентов в зависимости от версии интерпретатора.

Хочется заметить, что использование LuaJIT позволяет почти полностью избежать проблем с ухудшением производительности по сравнению с другими версиями. Поэтому можно сделать предположение, что JIT умеет определять и хорошо оптимизировать такой вид рефакторинга. Это наблюдение позволяет предложить использование такого рода оптимизаций по размеру и для других языков программирования с поддержкой JIT-компиляции, в частности JavaScript и Python.

Заключение

В ходе выполнения выпускной квалификационной работы были достигнуты следующие результаты.

- Произведён обзор предметной области:
  - изучены методы поиска клонов и их реализации (NiCad, CCStokener, Oreo и другие), а также проведён их сравнительный анализ;
  - изучены методы рефакторинга и применимость их для удаления клонов кода, а также проведено сравнение инструментов для удаления дубликатов кода;
  - изучены синтаксис и семантика языка Lua, проведено сравнение разных версий языка;
  - для данной работы выбран лексический метод поиска клонов, а в качестве метода рефакторинга — извлечение функции с параметризацией.
- Выявлены функциональные и нефункциональные требования к решению на основе взаимодействия с заказчиком.
- Спроектирован конвейер работы решения, состоящий из трёх этапов: нормализация, поиск клонов и рефакторинг.
- Реализовано решение по поиску клонов и автоматическому рефакторингу Lua-программ:
  - переписан парсер и лексер из оригинальной реализации Lua 5.3 для упрощения интеграции с решением;
  - реализовано 7 методов дополнительной нормализации исходного кода Lua-программ;
  - разработана компонента поиска клонов кода в Lua-программах, поддерживающая поиск клонов T1 и T2, для которых возможен рефакторинг;
  - реализован метод однопроходного рефакторинга клонов, в том числе и вложенных, как в пределах одного файла, так и всего проекта.

- Проведено экспериментальное исследование характеристик разработанного решения, выбранных методов поиска и рефакторинга клонов.
- Начато внедрение решения в конвейер сборки крупной технологической компании.
- Результаты работы были представлены на конференции «Современные технологии в теории и практике программирования» и опубликованы в сборнике тезисов докладов конференции.

Глоссарий

- **АСД** — абстрактное синтаксическое дерево. Конечное помеченное ориентированное дерево, в котором внутренние вершины соответствуют операторам языка программирования, а листья соответствуют их операндам.
- **ГЗП** — граф зависимостей программы. Вершинами ГЗП являются инструкции программы. Рёбра соответствуют потокам управления и данных внутри программы.
- **ЧУМ** — частично-упорядоченное множество. Множество с заданным на нём отношением частичного порядка (рефлексивность, транзитивность, антисимметричность).
- **IDE** — интегрированная среда разработки.
- **T1** — клон типа 1. Идентичные фрагменты кода, отличающиеся только в форматировании кода и комментариях.
- **T2** — клон типа 2. Фрагменты кода, отличающиеся только в форматировании, комментариях, именах переменных, именах типов и значениях литералов.
- **T3** — клон типа 3. В дополнение к отличиям типа 2 возможно отсутствие/добавление/модификация отдельных инструкций.
- **T4** — клон типа 4. Семантически эквивалентные фрагменты кода, но отличающиеся в синтаксисе.

Список литературы

[1] Арутюнян М. С. Статический анализ исходного и исполняемого кода на основе поиска клонов кода : дис. – Ин-т систем. программирования, 2025.

[2] Асланян А. К. и др. Платформенно-независимый и масштабируемый инструмент поиска клонов кода в бинарных файлах // Труды Института системного программирования РАН. – 2016. – Т. 28. – №. 5. – С. 215-226.

[3] Горчаков А. В. Математическое и алгоритмическое обеспечение интеллектуального статического анализа программных систем для специализированных гетерогенных вычислительных платформ : дис. – РТУ МИРЭА, 2024.

[4] Кознов Д. В. Языки визуального моделирования: проектирование и визуализация программного обеспечения. – 2004.

[5] Саргсян С. С. Методы оптимизации алгоритмов статического и динамического анализа : дис. – Ин-т систем. программирования, 2024.

[6] Саргсян С. С. Методы поиска клонов кода и семантических ошибок на основе семантического анализа программы : дис. – Ин-т систем. программирования, 2016.

[7] Сухинин А. А. Инкрементальное обнаружение клонов в контексте IDE: магистерская диссертация: 09.04.01. – 2016.

[8] AlOmar E. A., Mkaouer M. W., Ouni A. Behind the intent of extract method refactoring: A systematic literature review // IEEE Transactions on Software Engineering. – 2024. – Т. 50. – №. 4. – С. 668-694.

[9] Analyze duplicates. — URL: `https://www.jetbrains.com/help/idea/analyzing-duplicates.html` (дата обращения: 02.02.2025).

[10] ANTLR. — URL: `https://www.antlr.org/` (дата обращения: 02.02.2025).

[11] APISIX. — URL: https://github.com/apache/apisix (дата обращения: 11.02.2025).

[12] Balazinska M. et al. Partial redesign of Java software systems based on clone analysis // Sixth Working Conference on Reverse Engineering (Cat. No. PR00303). – IEEE, 1999. – С. 326-336.

[13] Baxter I. D. et al. Clone detection using abstract syntax trees // Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). – IEEE, 1998. – С. 368-377.

[14] Bowyer K. W., Hall L. O. Experience using "MOSS" to detect cheating on programming assignments // FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011). – IEEE, 1999. – Т. 3. – С. 13B3/18-13B3/22 vol. 3.

[15] CE6800. — URL: https://e.huawei.com/en/products/switches/data-center-switches/ce6800 (дата обращения: 11.02.2025).

[16] Cordy J. R., Roy C. K. The NiCad clone detector // 2011 IEEE 19th international conference on program comprehension. – IEEE, 2011. – С. 219-220.

[17] Cordy J. R. TXL-a language for programming language tools and applications // Electronic notes in theoretical computer science. – 2004. – Т. 110. – С. 3-31.

[18] CPD. — URL: https://github.com/pmd/pmd/blob/main/docs/pages/pmd/userdocs/cpd/cpd.md (дата обращения: 02.02.2025).

[19] DMS. — URL: http://www.semdesigns.com/Products/DMS/DMSToolkit.html (дата обращения: 02.02.2025).

[20] Ducasse S., Rieger M., Demeyer S. A language independent approach for detecting duplicated code // Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360). – IEEE, 1999. – С. 109-118.

[21] Eclipse IDE. — URL: https://eclipseide.org/ (дата обращения: 02.02.2025).

[22] Find and replace code duplicates. — URL: https://www.jetbrains.com/help/idea/find-and-replace-code-duplicates.html (дата обращения: 02.02.2025).

[23] Fowler M. Refactoring: improving the design of existing code. – Addison-Wesley Professional, 2018.

[24] Fowler M. UML distilled: a brief guide to the standard object modeling language. – Addison-Wesley Professional, 2018.

[25] gzip. — URL: https://www.gnu.org/software/gzip/ (дата обращения: 02.02.2025).

[26] Huang N. T., Villar S. A short tutorial on the weisfeiler-lehman test and its variants // ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). – IEEE, 2021. – С. 8533-8537.

[27] Huawei Technologies Co., Ltd. VRP Fundamentals // Data Communications and Network Technologies. – Singapore : Springer Nature Singapore, 2022. – С. 73-111.

[28] IntelliJ IDEA Ultimate. — URL: https://www.jetbrains.com/idea/ (дата обращения: 02.02.2025).

[29] Jiang L. et al. Deckard: Scalable and accurate tree-based detection of code clones // 29th International Conference on Software Engineering (ICSE'07). – IEEE, 2007. – С. 96-105.

[30] Kamiya T. CCFinderX: An interactive code clone analysis environment // Code Clone Analysis: Research, Tools, and Practices. – 2021. – С. 31-44.

[31] Kim S. et al. Vuddy: A scalable approach for vulnerable code clone discovery // 2017 IEEE symposium on security and privacy (SP). – IEEE, 2017. – С. 595-614.

[32] Kingston S., I Pun V. K., Stolz V. Automated clone elimination in Python tests // International Symposium on Leveraging Applications of Formal Methods. – Cham : Springer Nature Switzerland, 2024. – С. 97-114.

[33] Kong. — URL: https://github.com/Kong/kong (дата обращения: 11.02.2025).

[34] Koni-N'Sapu G. G. A scenario based approach for refactoring duplicated code in object oriented systems : дис. – Diploma thesis, University of Bern, 2001.

[35] Koznov D. V. et al. Calculating Similarity of Javadoc Comments // Programming and Computer Software. – 2024. – Т. 50. – №. 1. – С. 85-89.

[36] Li L. et al. Cclearner: A deep learning-based clone detection approach // 2017 IEEE international conference on software maintenance and evolution (ICSME). – IEEE, 2017. – С. 249-260.

[37] Livieri S. et al. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder // 29th International Conference on Software Engineering (ICSE'07). – IEEE, 2007. – С. 106-115.

[38] LLVM. — URL: https://llvm.org/ (дата обращения: 02.02.2025).

[39] Lua 5.1. — URL: https://www.lua.org/manual/5.1/ (дата обращения: 02.02.2025).

[40] Lua 5.2. — URL: https://www.lua.org/manual/5.2/ (дата обращения: 02.02.2025).

[41] Lua 5.3. — URL: https://www.lua.org/manual/5.3/ (дата обращения: 02.02.2025).

[42] Lua 5.4. — URL: https://www.lua.org/manual/5.4/ (дата обращения: 02.02.2025).

[43] Lua Benchmarks. — URL: https://github.com/gligneul/Lua-Benchmarks (дата обращения: 10.05.2025)

[44] Lua Language Server. — URL: https://github.com/LuaLS/lua-language-server (дата обращения: 11.02.2025).

[45] LuaJIT. — URL: https://luajit.org/ (дата обращения: 02.02.2025).

[46] Luciv D. V. et al. Interactive near duplicate search in software documentation // Programming and Computer Software. – 2019. – Т. 45. – №. 6. – С. 346-355.

[47] Luciv D. V. et al. Detecting near duplicates in software documentation // Programming and Computer Software. – 2018. – Т. 44. – С. 335-343.

[48] Mazinanian D. et al. JDeodorant: clone refactoring // Proceedings of the 38th international conference on software engineering companion. – 2016. – С. 613-616.

[49] Menéndez M. L. et al. The jensen-shannon divergence // Journal of the Franklin Institute. – 1997. – Т. 334. – №. 2. – С. 307-318.

[50] Mondal M., Roy C. K., Schneider K. A. A survey on clone refactoring and tracking // Journal of Systems and Software. – 2020. – Т. 159. – С. 110429.

[51] MOSS. — URL: https://theory.stanford.edu/~aiken/moss/ (дата обращения: 02.02.2025).

[52] Nakagawa T. et al. How compact will my system be? A fully-automated way to calculate Loc reduced by clone refactoring // 2019 26th Asia-Pacific Software Engineering Conference (APSEC). – IEEE, 2019. – С. 284-291.

[53] Pizzolotto D., Inoue K. Blanker: a refactor-oriented cloned source code normalizer // 2020 IEEE 14th International Workshop on Software Clones (IWSC). – IEEE, 2020. – С. 22-25.

[54] PMD. — URL: https://github.com/pmd/pmd (дата обращения: 02.02.2025).

[55] Rodrigues I. M. et al. TraceSim: An alignment method for computing stack trace similarity // Empirical Software Engineering. – 2022. – Т. 27. – №. 2. – С. 53.

[56] Roy C. K., Cordy J. R. A survey on software clone detection research // Queen's School of computing TR. – 2007. – Т. 541. – №. 115. – С. 64-68.

[57] Saini V. et al. Oreo: Detection of clones in the twilight zone // Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. – 2018. – С. 354-365.

[58] Saito Y. et al. Multiple Function Merging for Code Size Reduction // ACM Transactions on Architecture and Code Optimization. – 2024.

[59] Sajnani H. et al. SourcererCC: Scaling code clone detection to big-code // Proceedings of the 38th international conference on software engineering. – 2016. – С. 1157-1168.

[60] Sood S., Loguinov D. Probabilistic near-duplicate detection using simhash // Proceedings of the 20th ACM international conference on Information and knowledge management. – 2011. – С. 1117-1126.

[61] strip. – URL: https://www.man7.org/linux/man-pages/man1/strip.1.html (дата обращения: 02.02.2025).

[62] Svajlenko J., Roy C. K. Evaluating clone detection tools with BigCloneBench // 2015 IEEE international conference on software maintenance and evolution (ICSME). – IEEE, 2015. – С. 131-140.

[63] Tairas R., Gray J. Increasing clone maintenance support by unifying clone detection and refactoring activities // Information and Software Technology. – 2012. – Т. 54. – №. 12. – С. 1297-1307.

[64] Terser. — URL: https://github.com/terser/terser (дата обращения: 02.02.2025).

[65] Tree-sitter. — URL: https://tree-sitter.github.io/tree-sitter/ (дата обращения: 02.02.2025).

[66] Tsantalis N., Mazinanian D., Krishnan G. P. Assessing the refactorability of software clones // IEEE Transactions on Software Engineering. – 2015. – Т. 41. – №. 11. – С. 1055-1090.

[67] UglifyJS. — URL: https://github.com/mishoo/UglifyJS (дата обращения: 02.02.2025).

[68] Wang W. et al. CCStokener: Fast yet accurate code clone detection with semantic token // Journal of Systems and Software. – 2023. – Т. 199. – С. 111618.

[69] Wang P. et al. CCAligner: a token based large-gap clone detector // Proceedings of the 40th International Conference on Software Engineering. – 2018. – С. 1066-1077.

[70] Xmake. — URL: https://github.com/xmake-io/xmake/ (дата обращения: 11.02.2025).

[71] Yoshida N. et al. How slim will my system be? estimating refactored code size by merging clones // Proceedings of the 26th Conference on Program Comprehension. – 2018. – С. 352-360.

[72] Zhao J. Automatic refactoring for renamed clones in test code : дис. – University of Waterloo, 2018.

[73] Zhu W. et al. MSCCD: grammar pluggable clone detection based on ANTLR parser generation // Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. – 2022. – С. 460-470.

[74] Zou Y. et al. CCGraph: a PDG-based code clone detector with approximate graph matching // Proceedings of the 35th IEEE/ACM international conference on automated software engineering. – 2020. – С. 931-942.