DOI: 10.15514/ISPRAS-2026-38(3)-24

Архитектурный подход к быстрой кросс-языковой навигации по связанным сущностям в программных проектах

Д.С. Дроздов, ORCID: 0000-0003-0381-1012 <ds-drozdov@yandex.ru>
С.С. Михалкович, ORCID: 0000-0003-0373-3886 <miks@sfedu.ru>
Южный федеральный университет,
Россия, 344006, г. Ростов-на-Дону, ул. Б. Садовая, д. 105/42.

Аннотация. В статье рассматривается новый архитектурный подход к быстрой навигации между связанными элементами программных проектов, написанных на разных языках. Приведен пример конкретной реализации архитектуры для языковой пары GraphQL и Go. Проблема навигации между фрагментами кода на разных языках решается за счет создания разметки проекта и автоматических переходов между соответствующими точками в коде. Подход ориентирован на повседневную деятельность разработчиков и сохраняет устойчивость по мере изменения кода. Метод основан на использовании легковесных грамматик и их синтаксических анализаторов для создания разметки проекта и установления связей между ее элементами. Введено определение нетривиального обработчика GraphQL-поля в Go как функции, содержащей вызовы или управляющие конструкции. Такие функции составляют основу разметки проекта на Go. Разработаны три специализированные легковесные грамматики, которые выделяют конструкции, однозначно характеризующие функцию на Go как нетривиальный обработчик поля. На этой базе построен алгоритм, который автоматически устанавливает соответствия между полями схемы языка запросов GraphQL и методами на Go. Алгоритм учитывает совокупность факторов: от соглашения об именовании методов до их синтаксических особенностей. Алгоритм при установке соответствия решает задачу ранжирования кандидатов на пару для данной точки. Рассчитаны метрики ранжирования MRR и hit@1. Показано, что новый подход обеспечивает медианное ускорение навигации по проекту примерно в 8 раз по сравнению с ручным поиском, а также полностью устраняет повторные переходы.

Ключевые слова: быстрая навигация; кросс-языковая навигация; многоязыковые проекты; разметка проектов; легковесные грамматики.

Для цитирования: Дроздов Д.С., Михалкович С.С. Архитектурный подход к быстрой кросс-языковой навигации по связанным сущностям в программных проектах. Труды ИСП РАН, том 38, вып. 3, часть 2, 2026 г., стр. 111–132. DOI: 10.15514/ISPRAS–2026–38(3)–24.

An Architectural Approach to Fast Cross-Language Navigation of Related Entities in Software Projects

D.S. Drozdov, ORCID: 0000-0003-0381-1012 <ds-drozdov@yandex.ru>
S.S. Mikhalkovich, ORCID: 0000-0003-0373-3886 <miks@sfedu.ru>

Southern Federal University,
105/42b, Bolshaya Sadovaya st., Rostov-on-Don, 344006, Russia.

Abstract. The paper describes an architectural approach to fast cross-language navigation in polyglot software projects. The core idea is to maintain an explicit, durable project markup, link language-level anchors and enable constant-time jumps between related locations. The approach has been instantiated for the GraphQL–Go pair. Lightweight parsers with three fine-grained grammars for Go (covering regular calls, anonymous calls, and control-flow constructs) have been used for automatic identification of non-trivial resolvers. These resolvers have been defined as a key part of code markup. The advantages of using lightweight grammars have been shown. We have implemented a matching algorithm ranking candidate Go methods for each GraphQL field by combining naming conventions with static code features. We have calculated hit@1 and MRR to assess the quality of the ranked candidate lists. Multiple open-source repositories have been used to validate the grammars and the matching algorithm and to conduct user-oriented evaluation. On the evaluated projects, the tool consistently delivers an approximate 8x speed-up. Beyond speed, we observe no repeated file visits and improved selection accuracy when compared to manual search.

Keywords: fast navigation; cross-language navigation; polyglot projects; project markup; lightweight grammars.

For citation: Drozdov D.S., Mikhalkovich S.S. An Architectural Approach to Fast Cross-Language Navigation of Related Entities in Software Projects. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 2, 2026, pp. 111-132 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-24.

1. Введение

Современное промышленное программирование требует работы с большими проектами, написанными на нескольких языках. В статье [1] отмечается, что профессиональный разработчик в среднем для каждого проекта использует семь языков и три связанные языковые пары. В многоязыковых проектах одна и та же функциональность может быть реализована с использованием кода на нескольких языках. Отсюда возникает задача навигации между связанными сущностями, написанными на разных языках. Основываясь на научных публикациях [2][8] и собственном опыте разработки, нетрудно выделить основные недостатки традиционной ручной навигации, сводящейся к поиску сущностей по именам и ключевым словам:

1. Высокие когнитивные затраты. Программист вынужден вручную сопоставлять сигнатуры методов, мысленно выстраивать связи между фрагментами кода и проводить «компиляцию в уме».
2. Низкая производительность. Процесс анализа кода занимает минуты, требует постоянного переключения контекста.
3. Ненадежность. В больших проектах с множеством одноименных сущностей велик риск установить ошибочную связь или пропустить ее.
4. Отсутствие накопления знаний. Связи, которые выстраивает разработчик при навигации, нигде не сохраняются и не передаются коллегам. Каждому последующему программисту придется повторять анализ снова.
5. Фрагментированное восприятие. Программист вынужден воспринимать функциональность как набор разрозненных файлов, связь между которыми он должен постоянно удерживать в голове.

В данном исследовании рассматривается новый архитектурный подход к быстрой кросс-языковой навигации по проектам, позволяющий решить описанные выше проблемы. В качестве платформы для экспериментов выбрана языковая связка GraphQL и Go. Основой архитектуры служит набор легковесных синтаксических анализаторов, общая теория которых описана в [9], а также разработанный авторами набор правил установки соответствия между синтаксическими деревьями разных языков. По сравнению с предыдущими работами [10]-[12] разметка создается не вручную, а автоматически с помощью предлагаемого авторами алгоритма, и учитывает сразу несколько языков.

В данной статье описываются преимущества использования легковесных синтаксических анализаторов по сравнению с полными в контексте межъязыковой навигации. Легковесные анализаторы для GraphQL и Go взяты из предыдущих работ авторов [11],[13]. Однако для точной разметки потребовалось ввести ряд новых определений и разработать дополнительные специализированные грамматики для Go. Использование нескольких небольших грамматик вместо единой упрощает разбор текста и является новым результатом по сравнению с публикациями [9][13][15].

В работе получены следующие результаты:

1. Дано определение быстрой кросс-языковой навигации по связанным сущностям в программных проектах. Описано и реализовано архитектурное решение для быстрой навигации в двухъязыковом проекте.
2. Дано формальное определение нетривиальных обработчиков GraphQL-полей, описана их роль в решении задачи разметки кода конкретной языковой пары GraphQL – Go. Созданы легковесные грамматики для распознавания управляющих операторов Go (if, for, switch, select), а также вызовов анонимных и обычных функций и методов. Использование данных грамматик значительно упрощает разметку проекта по сравнению с полным анализатором.
3. Реализован алгоритм автоматического создания разметки и связей для кросс-языковой навигации на примере пары GraphQL – Go.
4. Рассчитаны метрики ранжирования hit@1 и MRR для алгоритма поиска кандидатов на установку связи между точками на GraphQL и Go.
5. Рассчитан коэффициент ускорения, показывающий, во сколько раз навигация с помощью инструмента быстрее навигации, использующей поиск.

Практическим результатом работы является ускорение навигации по многоязыковому проекту в 7,89 раз, а также существенное снижение когнитивной нагрузки на программистов за счет повышения точности навигации до 100% и устранения повторных посещений одних и тех же сущностей.

Архитектурный подход, реализованный и подробно описанный для языковой пары GraphQL – Go, также применен для пары GraphQL – TypeScript. В статье приводится ссылка на соответствующую грамматику TypeScript и алгоритм разметки и связи точек в проекте.

2. Быстрая кросс-языковая навигация

2.1 Основные определения

Определение 2.1.1. Пусть P – многоязыковой программный проект, состоящий из множества F файлов, написанных на языках из множества L. Для каждого языка $l \in L$ задан легковесный синтаксический анализатор $Parse_l: text \to AST$, который за линейное время по количеству токенов в каждом файле $f \in F$, написанном на языке $l \in L$, строит синтаксическое дерево AST_f. При этом множество узлов дерева AST_f состоит из двух частей:

1. Подмножество $V_{Bind}(AST_f)$ узлов, которые отвечают за синтаксические конструкции, необходимые для установления связей. Такие узлы принято называть ***точками привязки*** [10]. Далее будем отождествлять точку привязки и позицию в коде, где начинается соответствующая ей синтаксическая конструкция.
2. Подмножество $V_{Any}(AST_f)$ специальных узлов Any, отвечающих за прочие синтаксические конструкции.

Пусть на основе AST_f задано множество точек привязки $B_f \subseteq \{v \mid v \in V_{Bind}(AST_f)\}$. Тогда ***разметкой*** кода проекта P будем называть объединение точек привязки по всем файлам проекта $\mathrm{Markup}(P) = \bigcup_{f \in F} B_f$ и кратко записывать $Markup$.

Определение 2.1.2. ***Быстрой кросс-языковой навигацией*** будем называть метод поддержки разработчика, основанный на построении и использовании семантического графа связей между сущностями проекта, написанными на разных языках. Данный метод позволяет формализовать и автоматизировать процесс установления и последующего использования семантических связей между сущностями проекта, обеспечивая мгновенный переход между ними непосредственно в среде разработки.

Пусть задан программный проект с разметкой $Markup$. Тогда введем его **формальную модель** в виде n-дольного графа семантических связей $G = (V, E)$, где

1. Множество вершин $V = \{v \mid v \in Markup\}$ разбито на n непересекающихся долей $Part_1, Part_2, \dots, Part_n$, каждая из которых соответствует множеству точек привязки одного из языков программирования или технологий проекта.
2. Ребра E проводятся только между вершинами из разных долей: $\forall (u, v) \in E : u \in Part_i, v \in Part_j, i \neq j$. Множество ребер $E = E_{fixed} \cup E_{ambiguous}$ состоит из: (1) однозначно установленных ребер E_{fixed}; (2) ребер-кандидатов $E_{ambiguous}$ – потенциальных связей, для которых система обнаружила несколько возможных соответствий.
3. Множество ребер E_{fixed} образует множество несвязных между собой n-дольных подграфов. Каждый такой подграф C_k представляет собой отдельную *функциональность* и содержит не более одной вершины из каждой доли $Part_1, Part_2, \dots, Part_n$, между которыми существуют всевозможные попарные связи. Вершины подграфа C_k будем называть ***связанными сущностями***.

Представим **реализацию** быстрой кросс-языковой навигации в виде трех фаз. Первая фаза выполняется один раз при запуске инструмента, вторая и третья – каждый раз при инициализации навигации.

Предварительная фаза (автоматическое установление связей). Система выполняет статический анализ проекта и формирует множество связей E_{fixed}. Для связей, которые не могут быть установлены однозначно, система сохраняет множество кандидатов $E_{ambiguous}$. Конкретная реализация данной фазы для языковой пары GraphQL – Go представлена в подразделе 2.7 данной работы.

Фаза выбора (ручное уточнение связей). При инициации навигации из точки на одном языке в точку на другом система проверяет наличие связи. Если на предыдущем шаге связь была установлена однозначно, то система показывает одну целевую точку. В противном случае предлагает пользователю ранжированный список кандидатов и затем фиксирует его выбор, переводя соответствующее ребро из $E_{ambiguous}$ в E_{fixed}.

Фаза перехода (устойчивая навигация). После выбора точки среди кандидатов (или при однозначном соответствии) осуществляется немедленный переход к целевой сущности. Установленная связь используется для всех последующих переходов.

Дополнительно отметим, что код, соответствующий точкам привязки из разметки, может изменяться. Устойчивость разметки (соответственно, и связей между ее точками) обеспечивается с помощью алгоритмической перепривязки, описанной в статьях [12],[14]. В данной работе акцент делается только на задачах создания разметки и связей между точками привязки.

2.2 Навигация между GraphQL и Go

Для программной реализации зафиксируем два языка программирования: GraphQL и Go. Согласно спецификации [16], схема GraphQL содержит перечень типов и их полей, каждому полю соответствует в основном языке метод, называемый *обработчиком поля* (в англоязычной литературе – *resolver*), который должен вернуть значение клиенту. В работе [10] подчеркивается, что данные сущности образуют ключевые функциональности проекта, поэтому их выделение является первоочередной задачей при создании разметки. В таком случае следует реализовать алгоритм, который установит связи между точками на GraphQL и Go. В рамках данного исследования будем рассматривать только случаи корректных проектов, где для каждого поля GraphQL существует обработчик (возможно, еще не полностью реализованный).

2.2.1 Поля GraphQL и их обработчики на Go

Опишем, какое множество точек на GraphQL и Go нас будет интересовать при решении задачи разметки. В данном исследовании будем рассматривать типы GraphQL, каждый тип имеет набор полей, а поле – набор аргументов. С точки зрения транспортного уровня GraphQL не выделяет самостоятельной сущности «функция» или «метод» как в других языках программирования [17]. Каждому полю GraphQL соответствует метод-обработчик, который содержит логику и отвечает за реализацию функциональности [18]. Связь между полем и его обработчиком и будет лежать в основе навигации. На рис. 1 представлено соответствие между полями GraphQL и их обработчиками.

Опишем, как реализован каждый из представленных выше обработчиков (рис. 1):

1. **Session**. Содержит вызов некоторой функции, которая находит сессию по id.
2. **AllSessions**. Содержит вызов некоторой функции, которая находит все сессии.
3. **CurrentAccessLevel**. Возвращает текущий уровень доступа пользователя, который хранится в объекте обработчика. Не имеет вызовов функций.
4. **CurrentVersion**. Возвращает текущую версию сервера, которая хранится в объекте обработчика. Не имеет вызовов функций.

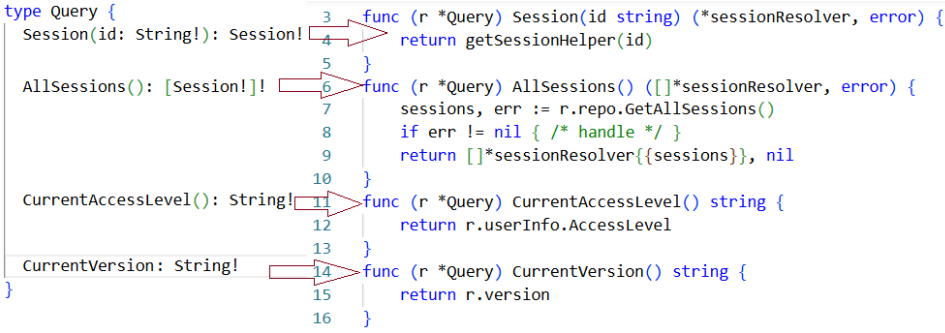

Рис. 1. Соответствие между полями GraphQL (слева) и их обработчиками (справа).
Fig. 1. Mapping between GraphQL fields (left) and their resolvers (right).

С точки зрения разметки кода нас интересуют только обработчики 1 и 2, поскольку именно в них совершаются значимые действия (вызовы других функций). Кроме этого, в разметку разумно включить методы-обработчики, которые содержат условные операторы, циклы или операторы выбора, поскольку данные конструкции несут смысловую нагрузку. Исходя из этого определим понятие нетривиального обработчика.

Определение 2.2.1. Метод-обработчик языка Go для поля схемы GraphQL будем называть ***нетривиальным***, если он содержит хотя бы один вызов функции или метода Go, либо один из следующих операторов: if, for, select, switch. Иначе такой метод-обработчик будем называть ***тривиальным***.

В таком случае методы 1 и 2 являются нетривиальными обработчиками, а 3 и 4 – тривиальными.

2.2.2 Проблема разметки

Мы определили множества интересующих точек на GraphQL и Go. При этом в разметку включаем только нетривиальные обработчики полей Go и сопоставленные им поля GraphQL.

Возникает вопрос, как определить набор связанных сущностей, который позволит переходить между точками. Очевидное решение сравнивать точки по имени является неполным по двум причинам. Во-первых, имя метода Go может не совпадать с именем поля GraphQL [19]. Во-вторых, методов с одинаковым названием может быть порядка десяти [20].

Помимо этого, необходимо исключить из разметки точки, относящиеся к тривиальным обработчикам. Как видно из примеров выше (рис. 1), по синтаксису поля GraphQL невозможно определить тип его обработчика. Действительно, нельзя опираться на количество аргументов, или на наличие круглых скобок, или на сигнатуру методов.

Для решения данной проблемы в настоящей работе разработаны легковесные синтаксические анализаторы, которые позволяют найти в теле обработчика те синтаксические конструкции, которые однозначно классифицируют его по определению 2.2.1. После этого в разметку кода включаются только нетривиальные обработчики.

2.3 Обзор имеющихся решений

2.3.1 Подходы к созданию разметки и семантических связей

Рассмотрим имеющиеся подходы к созданию разметки проекта и связей между ее точками. В работе [21] делается акцент на сигнатурах методов и на их наименовании. Анализ имен используется и в данной работе, однако он не работает в случае наличия одноименных функций, поэтому не может быть единственным инструментом. В статьях [22][23] предлагается опираться на специальные комментарии и аннотации, оставляемые разработчиками. Такой подход не используется в данном исследовании принципиально, потому что возлагает дополнительную работу на программистов и засоряет кодовую базу.

Существуют также способы отыскания функциональностей с помощью динамического анализа [24]-[25]. Однако зачастую такой подход достаточно трудозатратный. Например, в случае установки связи между Go и GraphQL придется проводить трассировку для каждого способа интеграции GraphQL отдельно, поскольку каждая библиотека, используемая разработчиками Go, по-разному взаимодействует со схемой GraphQL.

В статье [26] предлагается выделять функциональности с помощью машинного обучения с опорой на абстрактные синтаксические деревья. Однако анализаторы, используемые в статье, более не поддерживаются, и неясно, как обобщить результаты на другие языки.

В работе [27] рассматривается подход к созданию межъязыковой разметки и связи кода на Java/JavaScript и документацией, а также трекером задач. Однако поскольку связь

осуществляется не между двумя языками, а между одним языком и сторонними ресурсами, то это можно рассматривать как разметку одного языка с дополнительной информацией.

В статьях [28][29] описываются способы создания разметки с помощью отслеживания движения глаз и позиции курсора у опытных разработчиков. Такой подход позволяет связывать точки на разных языках, однако для создания полноценной разметки требуется объемная работа по отслеживанию активности, а также дорогостоящее оборудование.

2.3.2 Большие языковые модели (LLM)

Задачу понимания и навигации по проекту можно решать и без создания явной разметки. Например, в статье [30] авторами разработан плагин для VS Code, который по выделенному фрагменту дает подсказки и объяснения пользователю, в числе которых может содержаться информация, необходимая для навигации. Тем не менее, плагин больше рассчитан на изучение кода, чем на перемещение по нему.

В работе [31] авторы применяют LLM для семантического поиска по коду. Модель, разработанная авторами статьи, работает с несколькими языками. Однако в работе отмечается невысокое качество для проектов на Go, что делает ее нежелательной для применения в настоящем исследовании. В статье [32] также применяются большие языковые модели для навигации по проекту. Плагин, разработанный авторами, предсказывает, куда дальше перешел бы разработчик. Это делается с помощью модели, обученной на датасете, хранящем информацию о движении глаз и курсора многих разработчиков. Основным недостатком является возможность перемещаться только внутри одного файла.

В последнее время разработчиками активно используется GitHub Copilot для работы с большими и незнакомыми проектами [33][34]. Для навигации по проекту можно задать запрос на установление связи между точками. Однако для эффективного поиска требуется задать хотя бы один опорный файл. Кроме того, известно, что Copilot может выдавать ссылки на несуществующие файлы или позиции в них или давать существенно разные результаты при незначительном изменении запроса или исходного кода [35][36].

2.3.3 Имеющиеся инструменты навигации для GraphQL и парного языка

Поскольку в данной работе кросс-языковая навигация рассматривается на примере связки GraphQL и парного языка, то следует рассмотреть инструменты для GraphQL. В открытом доступе опубликован плагин *GraphQL: Language Feature Support* [37] для VS Code. Исходя из описания, он позволяет перемещаться от определения в схеме GraphQL к реализации на языке TypeScript. Однако в замечаниях к расширению профессиональным сообществом отмечается, что функция навигации не работает.

Другой плагин *GraphQL* [38] обеспечивает перемещение между GraphQL и TypeScript за счет специальных аннотаций в коде TypeScript. В плагине отсутствует поддержка Go.

Еще одно расширение *Apollo GraphQL* [39] обеспечивает навигацию внутри файлов GraphQL и поддерживает федерацию – объединение множества файлов со схемами. Однако с его помощью нельзя переместиться в соответствующий код на Go.

2.3.4 Кодогенерация и языковые серверы

Для связки GraphQL и Go существует популярный инструмент *gqlgen* [40], который по схеме GraphQL генерирует код на Go. Поскольку кодогенерация стандартизирована, можно встроиться в ее процесс и создавать разметку одновременно с генерацией. Кроме того, можно воспользоваться информацией о том, в какие директории и файлы *gqlgen* помещает код. Однако данный подход неудобен тем, что для других языков программирования нужны иные инструменты, следовательно, такое построение разметки плохо масштабируется на другие системы. Помимо этого, несмотря на распространенность библиотеки *gqlgen*, некоторые проекты принципиально пишутся без кодогенерации или на других библиотеках.

Для навигации внутри кода, написанного на Go и GraphQL, традиционно используются языковые сервера: *gopls* [41] и *GraphQL LSP* [42] соответственно. Однако в настоящем исследовании мы разрабатываем архитектурный подход к *кросс-языковой* навигации. Данная проблема не может быть решена простым использованием двух языковых серверов, нужен некий контроллер над ними. Кроме того, применение полноценных языковых серверов является непростой задачей, усложняющейся с ростом количества языков.

Решение, представленное в данной работе, фактически реализует ту часть некоторого универсального многоязыкового сервера, которая отвечает за переход между связанными сущностями.

2.4 Архитектурное решение для кросс-языковой навигации

В данной работе для навигации по многоязыковому проекту создана архитектура, состоящая из следующих компонентов:

1. Набор легковесных синтаксических анализаторов для каждого языка.
2. Алгоритм автоматического создания разметки и установки связей между точками привязки на разных языках.
3. Панель для отображения созданной разметки и быстрого перехода между точками.
4. Алгоритмы автоматического обновления разметки при изменении кода.

В качестве набора легковесных синтаксических анализаторов для языковой пары GraphQL – Go используются анализаторы из работ [11][13], а также три дополнительных анализатора, разработанных в данной статье (подраздел 2.6) для классификации обработчиков полей. Алгоритм автоматического создания разметки и правил навигации является новым результатом, представленным в подразделе 2.7 настоящей работы. Данный алгоритм интегрирован в программный комплекс LanD Explorer, который обеспечивает отображение разметки на специальной панели, ее автоматическое обновление и техническую часть навигации по связным сущностям [12]. Устойчивость созданных связей и разметки обеспечивается благодаря алгоритмической перепривязке, реализованной в предыдущих работах [12][14].

2.5 Синтаксические анализаторы

Согласно определению 2.1.1 для разметки проекта требуется синтаксический анализатор, строящий синтаксическое дерево по коду программы на соответствующем языке. В данном исследовании принято решение использовать легковесные анализаторы на основе легковесных грамматик со специальным символом Any. В работах [9],[13][15] убедительно показаны следующие преимущества легковесных анализаторов над полными:

1. Единое внутреннее представление для двух и более языков. Нет необходимости разбираться во внутреннем представлении синтаксических деревьев на каждом языке и создавать адаптеры. Времязатратность такого исследования подчеркивается в работе [43]. В статье [44] дополнительно выделяются преимущества единого внутреннего представления синтаксических деревьев при межъязыковом анализе.
2. Нет необходимости обновлять анализаторы при выходе новой версии языка, если изменения не затрагивают рассматриваемые синтаксические сущности.
3. Возможность разбирать код с некоторыми синтаксическими ошибками.

Дополнительно отметим, что использование легковесных грамматик подразумевает написание только необходимых правил, что позволяет фокусировать внимание на нужных конструкциях и разрабатывать грамматики по внешнему виду описываемых конструкций.

Отдельно следует указать на то, что применение полного синтаксического анализатора зачастую не покрывает все случаи. Например, для GraphQL в открытом доступе можно найти

три известных анализатора, каждый из которых имеет собственные ограничения: *graphql-go-tools* [45] игнорирует некорневые типы GraphQL; *graphql-go* [46] не поддерживает федеративный способ хранения схем GraphQL; *gqlparser* [47] некорректно разбирает поля с пустым списком аргументов.

Чтобы реализовать разметку в общем виде, требуется анализатор, который будет разбирать любые виды схем GraphQL. С этой задачей успешно справляется легковесный анализатор, ранее разработанный авторами [11].

2.6 Легковесные грамматики и анализаторы

2.6.1 Общие принципы легковесных грамматик

Для создания легковесного синтаксического анализатора необходимо воспользоваться генератором LanD и написать для него легковесную грамматику соответствующего языка [12]. Такие грамматики являются разновидностью LR(1) грамматик и основываются на использовании специального символа Any, обозначающего часть кода, которую нужно пропустить при разборе [9].

Кроме того, легковесные грамматики являются подвидом островных грамматик [43]. Например, когда мы разбираем условные операторы, тело оператора является «водой» и обозначается символом Any, а ключевые слова и скобки – «островами». На рис. 2 приведен пример такой разметки. Сплошной заливкой выделены «острова», а штриховкой – «вода». Справа представлен соответствующий фрагмент легковесной грамматики.

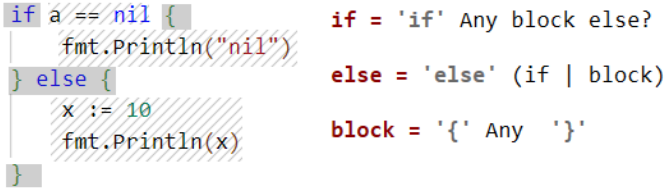

Рис. 2. «Вода» и «острова» в разметке метода. Фрагмент грамматики с Any.
Fig. 2. "Water" and "islands" in code markup. Part of grammar with "Any" symbol.

Последовательность символов, пропущенная при разборе, сохраняется в виде текста в соответствующем узле Any, что позволяет в дальнейшем разобрать его, например более детальным анализатором.

2.6.2 Новые легковесные грамматики в архитектуре кросс-языковой навигации

Как отмечено в подразделе 2.2.1, для языковой пары GraphQL – Go в разметку принято решение включать только нетривиальные обработчики полей. Согласно определению 2.2.1 такие методы-обработчики должны содержать как минимум один вызов метода или функции, либо управляющий оператор if, for, switch, select. Тогда для создания разметки необходимо не просто найти методы на языке Go, но и отфильтровать их в зависимости от содержимого. Для этого рассмотрим три новые легковесные грамматики Go: (1) обычных вызовов, (2) анонимных вызовов и (3) управляющих операторов. Синтаксические анализаторы, сгенерированные по этим грамматикам, будем применять последовательно для того, чтобы установить факт наличия конструкций, характеризующих обработчик поля как нетривиальный (определение 2.2.1).

В данной работе впервые по сравнению с предыдущими публикациями [9],[12]-[15] применяется подход создания нескольких небольших грамматик вместо единой грамматики. Такой подход имеет ряд преимуществ. Во-первых, каждая грамматика фокусируется на

одной семантической конструкции. Во-вторых, можно легко включать и исключать новые грамматики, поскольку они не зависят друг от друга. Наконец, в каждой грамматике снижается сложность правил за счет уменьшения числа рекурсии в них, что упрощает создание легковесных грамматик для малознакомых языков.

Далее опишем ключевые аспекты создания трех новых легковесных грамматик языка Go. Полный текст грамматик доступен в репозитории [48].

2.6.3 Грамматика обычных вызовов

Рассмотрим различные варианты вызовов в языке Go:

1. Обычный вызов: `Sin(x)`.
2. Вызов функции из пакета: `math.Sin(x)`.
3. Вызов метода структуры: `usr.Login(password)`.
4. Вызов элемента массива или слайса: `users[0].Login(password)`.
5. Вызов функции после приведения типа: `u.(func(string) int)("123")`.
6. Вызов функции, которую вернула предыдущая функция: `u.Login("123")()`.

При этом возможны и комбинации таких вариантов, например `data.u[0].(*User).Login("123")()`. Ключевая проблема состоит в том, что символ «`.`» может использоваться в двух случаях: обращение к функциям пакетов и к полям и методам структуры; приведение типа.

Во втором случае тип, к которому выполняется приведение, заключается в круглые скобки, что похоже на вызов функции. Учитывая эти особенности, сформулируем правила, описывающие вызовы.

```
call = fun ('(' entity* ')')+     fun = ID ('[' entity* ']')* ('.' fun2)?
fun2 = fun | '(' Any ')' ('[' entity* ']')* ('.' fun2)?
```

В данной грамматике терминал `ID` означает любой допустимый идентификатор в Go.

В языке Go есть 18 стандартных функций, таких как `new`, `len`, `close` и так далее, которые требуется исключить из разбора, потому что они не отвечают за функциональность. Для этого в грамматике достаточно определить терминалы `new`, `len`, `close` и прочие. В таком случае вызовы стандартных функций не будут подходить под описанное выше правило `fun` и будут пропущены. Кроме того, в грамматике определены терминалы для всех 27 стандартных типов Go и их синонимов, чтобы инициализаторы вида `int(0)` также игнорировались.

2.6.4 Грамматика анонимных вызовов

В языке Go среди всех вызовов выделяются вызовы анонимных функций, поскольку они имеют особую форму записи. Наиболее очевидным способом записать правило для анонимного вызова в Go является следующий:

```
anon_f_call = 'func' '(' Any ')' Any '{' Any '}' '(' Any ')'
```

Однако в качестве возвращаемого значения на позиции `Any` может оказаться структура или интерфейс Go, чья запись содержит фигурные скобки, которые будут конфликтовать со скобками, отвечающими за тело функции. Для этого заменим `Any` на более детальный фрагмент `Any|struct|interface` и для удобства разобьем правила на подправила. Получим следующую грамматику:

```
content = entity*                 entity = Any | anon_f_call
anon_f_call = anon_f '(' Any ')'  anon_f = anon_f_title '{' Any '}'
anon_f_title = 'func' '(' Any ')' (Any|struct|interface)
struct = 'struct' '{' Any '}'     interface = 'interface' '{' Any '}'
```

2.6.5 Грамматика управляющих операторов Go

Рассмотрим процесс создания грамматики для следующих операторов Go: if, for, switch, select. Каждый из них имеет обязательную часть – тело, которое всегда обрамлено фигурными скобками. Определим для этого правило `block = '{' Any '}'`. Тогда правила для условного оператора будут следующими:

```
content = entity*                    if = 'if' Any block else?
entity = if | block | Any            else = 'else' (if | block)
block = '{' entity* '}'
```

При детальном тестировании можно обнаружить, что данная грамматика некорректно описывает те единичные случаи, когда условие оператора содержит конструкции с фигурными скобками. Проблема возникает из-за того, что по спецификации Go в большинстве случаев условие можно не заключать в круглые скобки, поэтому трудно отследить, где начинается тело оператора [49]. Например, для кода `if a == struct{int}{3} { print(a) }` в качестве условного оператора будет определена подчеркнутая часть, а в качестве условия – только `a == struct{int}`.

Согласно грамматике, фигурные скобки в условии встречаются в следующих случаях и их комбинациях:

1. Инициализация анонимной структуры: `struct{ Id int }{1}`.
2. Инициализация слайса или массива: `[3]int{1, 2, 3}`.
3. Инициализация любого другого объекта: `User{Id: 1}`.
4. Заголовок анонимной функции, в котором есть структура или интерфейс: `func(int) struct{ Id int }`.
5. Тело анонимной функции: `func(User) int { return 0 }`.

Примечательно, что оригинальный синтаксический анализатор Go для случая 3 требует, чтобы такая конструкция обрамлялась круглыми скобками [50], поэтому для нее не требуется отдельного правила. Для сценариев 1 и 2 анализатор учитывает, что фигурные скобки относятся к инициализатору, а не являются границами тела оператора. Поступим аналогичным образом и добавим в грамматику правила, которые описывают инициализаторы 1 и 2.

```
init = anon '{' Any '}' | ('[' Any ']')+ Any ('{' Any '}')?
anon = ('struct'|'interface') '{' Any '}'
```

Далее правила для тела и вызова анонимной функции (случаи 4–6) заимствуются из подраздела 2.6.4. Теперь мы можем описать правило для условия и скорректировать для if:

```
cond = Any | init | anon_f | anon_f_call | anon_f_title
if = 'if' cond* block else?
```

Полученные результаты нетрудно распространить на другие операторы: for, switch и select.

2.6.6 Валидация легковесных грамматик

Для проверки корректности грамматик используется доработанная версия алгоритма валидации из работы [51]. Сначала подбирается набор тестовых репозиториев из числа проектов, опубликованных на GitHub. Затем полный синтаксический анализатор Go проходится по файлам из набора, находит функции и методы и разбирает их тела; результатом разбора является синтаксическое дерево операторов и вызовов. Дополнительно сохраняется текст функции или метода, который затем разбирает легковесный синтаксический анализатор. Наконец, сравниваются синтаксические деревья, построенные полным и легковесным анализаторами.

В качестве репозиториев для проверки выбрано пять проектов, их состав и результаты распознавания представлены в табл. 1. В первом столбце приведена ссылка на GitHub из списка литературы в конце статьи.

Табл. 1. Результаты работы легковесных анализаторов.
Table 1. Results of lightweight parsing.

Репозиторий	Всего вызовов	Распознано вызовов	Всего анонимных вызовов	Распознано анонимных вызовов	Всего операторов	Распознано операторов
[52]	311857	311857 (100%)	28	28 (100%)	148429	148429 (100%)
[53]	655748	655748 (100%)	3470	3470 (100%)	259428	259428 (100%)
[54]	228761	228761 (100%)	1331	1331 (100%)	139167	139167 (100%)
[55]	203397	203397 (100%)	1127	1127 (100%)	59310	59310 (100%)
[56]	164904	164904 (100%)	488	488 (100%)	39837	39837 (100%)

По результатам эксперимента установлено, что все три грамматики верно разбирают вызовы и операторы Go, ложноположительные и ложноотрицательные срабатывания отсутствуют.

2.7 Алгоритм навигации для языковой пары GraphQL – Go

Рассмотрим новый алгоритм создания разметки для языковой связи GraphQL – Go, а также связей между точками привязки.

Алгоритм состоит из нескольких частей:

1. Найти все методы Go и поля GraphQL с помощью легковесных синтаксических анализаторов.
2. Отфильтровать методы Go.
3. Составить пары вида <поле GraphQL; список кандидатов на обработчик поля>.
4. Рассчитать оценку соответствия `score` для каждого кандидата из пункта 3 и выбрать кандидата с наибольшим значением. В случае, если между максимальным и последующим кандидатами разница меньше 1, пользователю предлагаются кандидаты для ручного выбора.

Рассмотрим части 1–3. Вначале мы получаем список всех методов Go и полей GraphQL с помощью легковесных синтаксических анализаторов из предыдущих работ [10][12]. Далее отфильтровываются лишние методы Go, которые удовлетворяют хотя бы одному из следующих условий:

1. Имя метода Go не найдено в списке имен полей GraphQL. Сравниваются нормализованные имена.
2. Имя ресивера[1] содержит подстроку unimplemented без учета регистра. Такие методы генерируются автоматически и не содержат значащего кода.
3. Метод не содержит ни одной из синтаксических конструкций, перечисленных в определении 2.2.1 нетривиального обработчика. Для проверки этого условия последовательно применяются новые легковесные анализаторы, построенные по грамматикам из подраздела 2.6.3–2.6.5.

Затем для каждого метода Go мы просматриваем все поля GraphQL, совпадающие по имени, и добавляем в словарь новое значение, если количество аргументов метода Go соответствует одному из способов их перечисления в обработчиках полей. В качестве ключа словаря выступает поле GraphQL, а значения – список методов-кандидатов вместе с величиной `score`, отражающей, насколько вероятно данный метод является искомым обработчиком.

[1] Ресивер — структура, к которой принадлежит метод.

Величина `score` складывается из следующих частей (соответствующие пункты подписаны в Алгоритме 2.7.1 с указанием конкретных значений):

1. Соглашения о наименовании. Учитываются различные способы наименования, характерные для разных библиотек работы с GraphQL. Чем больше соглашений о наименовании выполнено, тем выше вероятность того, что перед нами искомый обработчик.
2. Доля вызовов, характерных для методов-заглушек, среди всех вызовов. Такими вызовами являются конструкции вида `id.Get(N)`. Определение таких вызовов выполняется с помощью синтаксического анализатора, построенного по грамматике из подраздела 2.6.3. Чем выше такая доля, тем выше вероятность, что метод является заглушкой для тестов.
3. Коэффициент вариации количества строк в методах в структуре, к которой относится метод, и в файле, в котором расположен метод. Исходя из анализа открытых репозиториев, установлено, что обычно методы-обработчики имеют примерно одинаковое количество строк в каждом из них, а также что такие методы группируются в файлы, которые больше не содержат других методов. Поэтому чем выше коэффициенты вариации, тем ниже вероятность того, что рассматриваемый кандидат является требуемым обработчиком. Веса, с которыми учтены коэффициенты вариации, подобраны экспериментально: их значения определялись перебором с шагом 0.05 при условии, что сумма двух весов равна 1.

Ключевые моменты программной реализации приведенного алгоритма представлены в репозитории [48].

```
procedure MakeNavEdges(D):
   e_fixed ← ∅;  e_ambiguous ← ∅
   foreach gqlField, candidates in D:
      if len(candidates) = 1:
         e_fixed ← append(e_fixed, {gqlField, candidates[0]})
         continue
      foreach cand in candidates:
         score ← 0;  method ← cand.method
         if ReceiverName(method) contains "resolver":                |
            score ← score + 1                                        |
         if ReceiverName(method) = ParentName(gqlField):             | (1)
            score ← score + 1                                        |
         else if ReceiverName(method) contains ParentName(gqlField): |
            score ← score + 0.75
         score ← score + (1 - MockCallsRatio(cand))                  | (2)
         score ← score + (1 – CVForStruct(method)) * 0.25 +          | (3)
            + (1 – CVForFile(method)) * 0.75
         cand.score ← score
      bestCandidate, ambiguousCands ← SortCandidates(candidates)
      if bestCandidate != null:
         e_fixed ← append(e_fixed, {gqlField, bestCandidate})
      else foreach ambCand in ambiguousCands:
         e_ambiguous ← append(e_ambiguous, {gqlField, ambCand})
   return e_fixed, e_ambiguous
```

Алгоритм 2.7.1. Установка соответствия между точками привязки на GraphQL и Go.
Algorithm 2.7.1. Establishing correspondence between GraphQL and Go binding points.

В результате работы общего алгоритма мы получаем следующее:

1. Разметка проекта. В качестве множества точек привязки выступают найденные точки в схеме GraphQL и отфильтрованные точки в коде на Go.
2. Набор связанных сущностей и кандидатов на них. Для пар, ребра которых попали в набор `e_fixed`, переход возможен сразу. Для `e_ambiguous` требуется однократное ручное действие для выбора подходящего кандидата из упорядоченного списка. Сортировка списка происходит по оценке соответствия `score`, сохраненной вместе с вершиной-кандидатом.

Таким образом, на примере языковой связи GraphQL – Go показан алгоритм построения разметки и установки соответствия между фрагментами кода на разных языках.

2.8 Демонстрация быстрой навигации для языковой пары GraphQL – Go

Рассмотрим схему быстрой навигации и ручного поиска на примере пары GraphQL – Go. На рис. 3 изображено несколько сценариев навигации.

Сценарий I описывает *первичный* переход от точки на GraphQL к точке на Go. В случае быстрой навигации пользователю предлагается несколько точек-кандидатов в виде ранжированного списка (зачастую список и вовсе состоит из одного искомого элемента); при ручном поиске множество кандидатов неупорядоченно и содержит больше элементов.

Сценарий II демонстрирует *повторный* переход между точками. Инструмент быстрой навигации уже хранит эту связь, поэтому переход выполняется мгновенно, в отличие от ручного поиска, который требуется выполнить снова.

Сценарий III предполагает переход от найденной точки к первоначальной. Благодаря наличию сохраненной связи, инструмент быстрой навигации выполняет это мгновенно.

Таким образом, инструмент быстрой кросс-языковой навигации существенно упрощает работу с проектом и переход между языками.

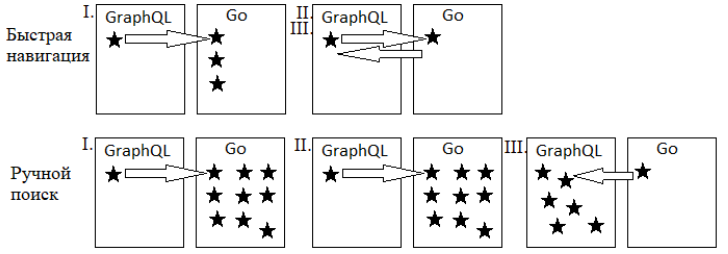

Рис. 3. Быстрая кросс-языковая навигация GraphQL – Go в сравнении с ручным поиском.
Fig. 3. Fast cross-language GraphQL–Go navigation vs. manual search.

3. Оценка качества кросс-языковой навигации

Для оценки качества работы алгоритма навигации в пределах языковой пары GraphQL – Go обратимся к репозиториям с открытым исходным кодом, разметим проект и установим связи вручную и с помощью алгоритма, далее сверим результаты. Результаты анализа приведены в табл. 2, ссылки на репозитории даны в конце статьи. Во втором столбце указано количество эталонных пар, установленных вручную. В третьем столбце приведено количество пар, однозначно установленных алгоритмом. В четвертом столбце указано количество потенциальных связей, для которых требуется ручная верификация пользователем. Формат записи следующий: «*количество точек в GraphQL* x *количество кандидатов в ранжированном списке*». Из полученных данных следует, что ручные действия пользователя требуются только в 2,5% случаев, в остальных ситуациях мы определяем подходящего кандидата самостоятельно.

Поскольку при построении связей между точками мы находим одного или нескольких кандидатов и упорядочиваем их по величине, описанной в подразделе 2.7, то имеет место задача ранжирования. Для оценки качества решения такой задачи рассчитаем общепринятые метрики, приведенные в столбцах 5–6. Величина hit@1 показывает, как часто искомый элемент попадает в топ-1 кандидатов. Метрика MRR (Mean reciprocal rank) показывает, насколько высоко в среднем нужный кандидат стоит в ранжированном списке. По результатам экспериментов установлено, что алгоритм всегда ставит на первое место искомого кандидата, поэтому значения метрик следующие: hit@1 = 100%, MRR = 1.

Табл. 2. Результаты оценки качества алгоритма кросс-языковой навигации в паре GraphQL – Go.
Table 2. Test results for the cross-language navigation algorithm in the GraphQL – Go pair.

(1)	(2)	(3)	(4)	(5)	(6)
Репозиторий	**Ручная разметка**	**Автоматическая разметка**		**Метрики ранжирования для автоматической разметки**	
	$\|E\|$	$\|E_{fixed}\|$	$\|E_{ambiguous}\|$	**hit@1**	**MRR**
[57]	24	24	0	100%	1
[58]	70	69	1 точка х 3 канд.	100%	1
[59]	91	91	0	100%	1
[60]	6	6	0	100%	1
[61]	17	16	1 точка х 2 канд.	100%	1
[62]	8	8	0	100%	1
[63]	134	127	7 точек х 2 канд.	100%	1
[64]	12	12	0	100%	1

4. Оценка эффективности быстрой навигации

При навигации между связанными сущностями, реализованными на разных языках, разработчик должен найти на втором языке точку, соответствующую выбранному элементу на первом языке. При ручном поиске это достигается за счет использования глобального поиска по имени и анализа найденных кандидатов. В литературе подобная деятельность описывается линейной моделью: общее время растет пропорционально количеству вариантов, которые необходимо просмотреть [65]-[66].

В предлагаемом инструменте быстрой навигации ситуация иная. Алгоритм заранее вычисляет связи между элементами и формирует ранжированный список кандидатов, состоящий, как правило, из одного-трех элементов. Правильная точка всегда присутствует среди кандидатов и по данным эксперимента (раздел 3) располагается на первой позиции. Таким образом, время автоматической навигации должно быть существенно меньше, чем время ручной навигации. Подтвердим предположение с помощью эксперимента.

4.1 Эксперимент

Определение 4.1.1. Пусть Δ_1 – суммарное время, затраченное разработчиком на навигацию по проекту с помощью ручного поиска, а Δ_2 – с помощью нового инструмента. Тогда ***коэффициентом ускорения навигации*** будем называть величину $\mu = \Delta_1/\Delta_2$.

Выберем в каждом репозитории по 10 точек на GraphQL и предложим трем разработчикам найти для них соответствующую пару на Go с помощью ручного поиска в IDE и с помощью инструмента для автоматической навигации. Вычислим коэффициент ускорения навигации и точность, с которой выполнено сопоставление. Дополнительно будем отслеживать процент

повторных посещений одних и тех же точек в процессе ручного поиска. Как отмечается в статье [67], чем выше такой процент, тем выше стресс у программиста.

Поскольку объём выборки в проведенном эксперименте ограничен, а отдельные измерения времени обладают естественной вариативностью, в качестве обобщающих показателей используются медиана и диапазон наблюдаемых значений, которые считаются устойчивыми метриками при малых выборках [66][68].

В табл. 3 приведены результаты наблюдений для первого разработчика. На данных репозиториях коэффициент ускорения принимал значения в диапазоне 3,75–14 с медианой 7,31. Кроме того, точность сопоставления при ручном поиске составила 80–100% против 100% при автоматической навигации. Доля повторных посещений варьировалась от 9% до 38% для ручного поиска.

В табл. 3 и далее не приводятся точность сопоставления и процент повторных посещений при использовании инструмента, поскольку они составляют 100% и 0% соответственно. Это объясняется тем, что в инструменте предлагается либо одна точка для навигации, либо несколько в виде ранжированного списка с метрикой hit@1 = 100%. В последнем случае испытуемые переходили в первую в списке точку и уверенно выбирали ее как искомую.

Проведем аналогичные наблюдения с двумя другими испытуемыми. Результаты представлены в табл. 4-5.

По результатам серии экспериментов значение коэффициента μ варьировалось от 2,02 до 14 с медианой наблюдаемых значений 7,89. Кроме того, процент повторных посещений снизился с медианных 23% до 0%, а точность сопоставления возросла с 90% до 100%.

Табл. 3. Результаты эксперимента по оценке быстрой навигации. Испытуемый – Разработчик 1.
Table 3. Results of the experiment on evaluating fast navigation. Subject: Developer 1.

	[57]	[58]	[59]	[61]	[63]
Время (инструмент), Δ_2, сек.	17	32	16	16	20
Время (ручной поиск), Δ_1, сек.	109	120	129	117	280
Точность сопоставления (ручной поиск)	80%	80%	100%	100%	100%
Процент повторных посещений (ручной поиск)	9%	17%	38%	9%	33%

Табл. 4. Результаты эксперимента по оценке быстрой навигации. Испытуемый – Разработчик 2.
Table 4. Results of the experiment on evaluating fast navigation. Subject: Developer 2.

	[57]	[58]	[59]	[61]	[63]
Время (инструмент), Δ_2, сек.	11	37	19	14	23
Время (ручной поиск), Δ_1, сек.	123	131	150	103	294
Точность сопоставления (ручной поиск)	90%	90%	100%	90%	100%
Процент повторных посещений (ручной поиск)	9%	9%	44%	29%	23%

Табл. 5. Результаты эксперимента по оценке быстрой навигации. Испытуемый – Разработчик 3.
Table 5. Results of the experiment on evaluating fast navigation. Subject: Developer 3.

	[57]	[58]	[59]	[61]	[63]
Время (инструмент), Δ_2, сек.	12	61	22	11	31
Время (ручной поиск), Δ_1, сек.	114	123	124	98	245
Точность сопоставления (ручной поиск)	70%	80%	90%	100%	90%
Процент повторных посещений (ручной поиск)	17%	23%	38%	29%	44%

Отметим, что эксперимент проводился на профессиональных разработчиках. Для начинающих программистов провести ручной поиск сложнее, зачастую им требуется консультация старшего специалиста, чтобы выбрать нужную сущность. Использование инструмента для автоматической навигации позволяет самостоятельно разобраться с проектом разработчику любого уровня.

5. Универсальность архитектурного подхода

В текущем исследовании рассмотрена реализация архитектурного подхода для GraphQL и Go. Данная языковая пара является нетривиальной из-за описанных в статье особенностей грамматик Go (см. подразделы 2.6.3–2.6.5), а также из-за традиционного наличия большого количества одноименных методов.

Действуя аналогичным образом, нетрудно определить грамматику методов и стрелочных функций языка TypeScript, использующихся в качестве обработчиков полей, а также алгоритм связи между точками на TypeScript и GraphQL. Данные результаты представлены в репозитории.

Наконец, в работе [9] представлена легковесная грамматика C#, которая в числе прочего описывает методы языка (полный текст грамматики также приведен в нашем репозитории). Пользуясь правилами из этой грамматики и имеющейся грамматикой GraphQL, можно построить разметку проекта и реализовать алгоритм связи для языковой пары GraphQL – C#.

6. Ограничения архитектурного подхода

Использование легковесных синтаксических анализаторов как основы предлагаемого архитектурного подхода предъявляет требования к разработке и валидации соответствующих легковесных грамматик. Для языков со специфической грамматикой может потребоваться детальный разбор отдельных конструкций, как например анализ условного оператора в Go (см. подраздел 2.6.5).

Реализация подхода интегрирована в инструмент LanD Explorer, обеспечивающий графический интерфейс и сопряжение с IDE Microsoft Visual Studio. Интеграция с Visual Studio Code в текущей версии не выполнена и требует дополнительных доработок.

Текущий подход не поддерживает кросс-языковую навигацию в тех случаях, когда текст на одном языке встроен в текст на другом языке. В ряде языков программирования схемы GraphQL иногда записываются в виде строковых литералов непосредственно в исходном коде, а не выделяются в отдельные файлы. Для обработки таких проектов необходим механизм автоматизированного обнаружения и извлечения схем из программного текста.

7. Заключение

В данной работе предложен архитектурный подход к быстрой кросс-языковой навигации по связанным сущностям программного проекта и показана его применимость на языковой связке GraphQL – Go. Ключевая идея состоит в создании явной разметки проекта и связей между ее частями. Для этого разработан и экспериментально валидирован набор легковесных анализаторов и грамматик, ориентированных на обнаружение тех синтаксических конструкций, которые критичны для навигации и классификации методов-обработчиков Go. Созданы и валидированы три специализированные грамматики Go: для обычных вызовов, для анонимных вызовов и для управляющих операторов. Показано, что их совместное применение корректно выделяет нетривиальные обработчики полей согласно предложенному определению и образует разметку проекта.

Кроме того, разработан алгоритм автоматического установления соответствий между полями схемы GraphQL и методами-обработчиками на Go, который использует соглашения об именовании, структуру кода и статические признаки для ранжирования кандидатов. По

результатам экспериментов на ряде открытых репозиториев вычислены метрики ранжирования кандидатов hit@1 и MRR, которые равны 100% и 1 соответственно. Помимо этого, установлено, что число случаев, требующих ручного анализа, не превышает 2,5%.

Экспериментально рассчитан коэффициент ускорения навигации. Инструмент быстрой навигации дает ускорение в 7,89 раз по медиане по сравнению со стандартным поиском в среде разработки, дополнительно увеличивая точность навигации до 100% и устраняя повторные переходы по сущностям проекта.

Архитектурный подход, подробно описанный для пары GraphQL – Go, распространен на пару GraphQL – TypeScript: соответствующая легковесная грамматика TypeScript, а также алгоритм разметки и установления связей между точками опубликованы в репозитории [48].

Список литературы / References

[1]. Mayer P., Kirsch M., Le M. A. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. Journal of Software Engineering Research and Development, vol. 5, 2017, Article 1, pp. 1–33. DOI: 10.1186/s40411-017-0035-z.

[2]. Yang H., Lian W., Wang S., Cai H. Demystifying Issues, Challenges, and Solutions for Multilingual Software Development. In Proc. of the IEEE/ACM International Conference on Software Engineering (ICSE), 2023, pp. 1840–1852. DOI: 10.1109/ICSE48619.2023.00157.

[3]. Latif S., Mushtaq Z. Pragmatic evidence of cross-language link detection: A systematic literature review. Journal of Systems and Software, vol. 206, 2023, Article 111825. DOI: 10.1016/j.jss.2023.111825.

[4]. Minelli R., Mocci A., Lanza M. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In Proc. of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC), 2015, pp. 25–35. DOI: 10.1109/ICPC.2015.12.

[5]. Robillard M. P., Coelho W., Murphy G. C. How effective developers investigate source code: An exploratory study. IEEE Transactions on Software Engineering, vol. 30, no. 12, 2004, pp. 889–903. DOI: 10.1109/TSE.2004.101.

[6]. Krein J. L., MacLean A. C., Knutson C. D., Delorey D. P., Eggett D. L. Impact of Programming Language Fragmentation on Developer Productivity: A Sourceforge Empirical Study. International Journal of Open Source Software and Processes, vol. 2, no. 2, 2010, pp. 41–61. DOI: 10.4018/JOSSP.2010040104.

[7]. Hao R. L., Glassman E. L. Approaching Polyglot Programming: What Can We Learn from Bilingualism Studies? In Proc. of the 10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019), OASIcs, vol. 76, 2020, pp. 1:1–1:7. DOI: 10.4230/OASIcs.PLATEAU.2019.1.

[8]. Damevski K., Shepherd D. C. A field study of how developers locate features in source code. Empirical Software Engineering, vol. 21, no. 2, 2016, pp. 724–747. DOI: 10.1007/s10664-015-9373-9.

[9]. Goloveshkin A. V., Mikhalkovich S. S. Tolerant parsing with a special kind of “Any” symbol: the algorithm and practical application. Trudy ISP RAN/Proc. ISP RAS, 2018, vol. 30, issue 4, pp. 7–28. DOI: 10.15514/ISPRAS-2018-30(4)-1.

[10]. Головешкин А. В., Михалкович С. С. Разметка сквозных функциональностей в коде программы. Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции, 2019, стр. 245–256. DOI: 10.20948/abrau-2019-83. / Goloveshkin A. V., Mikhalkovich S. S. Marking up crosscutting concerns in a program code. Scientific Service in the Internet: Proceedings of the XXI Scientific Conference, 2019, pp. 245–256 (in Russian). DOI: 10.20948/abrau-2019-83.

[11]. Дроздов Д. С., Михалкович С. С. Создание и постобработка легковесных грамматик Go и GraphQL для разметки функциональностей кода. Труды XXXI Всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития», 2024, стр. 163–165. / Drozdov D. S., Mikhalkovich S. S. Creation and post-processing of lightweight Go and GraphQL grammars for code functionality markup. In Proc. of the XXXI Scientific Conference “Modern information technologies: trends and prospects of development”, 2024, pp. 163–165 (in Russian).

[12]. Головешкин А. В., Михалкович С. С. LanD: инструментальный комплекс поддержки послойной разработки программ. Труды XXV Всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития», 2018, стр. 53–56. / Goloveshkin A. V., Mikhalkovich S. S. LanD: a framework for layer-by-layer program development. In Proc. of the XXV Scientific Conference “Modern information technologies: trends and prospects of development”, 2018, pp. 53–56 (in Russian).

[13]. Дроздов Д. С., Михалкович С. С. Разработка легковесных парсеров с разной детализацией языка Go. Электронные библиотеки, 2024, т. 27, № 6, стр. 857–877. DOI: 10.26907/1562-5419-2024-27-6-857-877. / Drozdov D. S., Mikhalkovich S. S. Development of lightweight parsers with different Go language granularity. Russian Digital Libraries Journal, 2024, vol. 27, issue 6, pp. 857–877 (in Russian). DOI: 10.26907/1562-5419-2024-27-6-857-877.

[14]. Goloveshkin A. V., Mikhalkovich S. S. Using improved context-based code description for robust algorithmic binding to changing code. Procedia Computer Science, 2021, vol. 193, pp. 239–249. DOI: 10.1016/j.procs.2021.10.024.

[15]. Дроздов Д. С., Михалкович С. С. Эффективность по памяти и времени легковесных парсеров с разной детализацией языка Go. Научный сервис в сети Интернет: труды XXVI Всероссийской научной конференции, 2024, стр. 85–98. DOI: 10.20948/abrau-2024-10. / Drozdov D. S., Mikhalkovich S. S. Memory and time efficiency of lightweight parsers with different Go language granularity. Scientific Service in the Internet: Proceedings of the XXVI Scientific Conference, 2024, pp. 85–98 (in Russian). DOI: 10.20948/abrau-2024-10.

[16]. GraphQL. October 2021 Edition. Available at: https://spec.graphql.org/October2021/, accessed 10.04.2026.

[17]. Amareen S., Soto Dector O., Dado A., Bosu A. GraphQL Adoption and Challenges: Community-Driven Insights from StackOverflow Discussions. arXiv preprint, 2024. DOI: 10.48550/arXiv.2408.08363.

[18]. Buna S. GraphQL in Action. Manning Publications, Shelter Island, NY, 2021. 384 p.

[19]. Raiturkar J. Hands-On Software Architecture with Golang: Design and architect highly scalable and robust applications using Go. Packt Publishing Ltd, Birmingham–Mumbai, 2018. 500 p.

[20]. Yellavula N. Hands-On RESTful Web Services with Go: Develop elegant RESTful APIs with Golang for microservices and the cloud. Packt Publishing Ltd, Birmingham–Mumbai, 2020. 393 p.

[21]. Griswold W. G. Coping with Crosscutting Software Changes Using Information Transparency. In Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION 2001), Lecture Notes in Computer Science, vol. 2192. Berlin–Heidelberg: Springer, 2001, pp. 250–265. DOI: 10.1007/3-540-45429-2_17.

[22]. Ying A. T. T., Wright J. L., Abrams S. Source Code That Talks: An Exploration of Eclipse Task Comments and Their Implication to Repository Mining. SIGSOFT Software Engineering Notes, vol. 30, no. 4, 2005, pp. 1–5. DOI: 10.1145/1082983.1083152.

[23]. Sulír M., Nosáľ M. Sharing developers' mental models through source code annotations. In Proc. of the 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), Annals of Computer Science and Information Systems, vol. 5, 2015, pp. 997–1006. DOI: 10.15439/2015F301.

[24]. Tonella P., Ceccato M. Aspect mining through the formal concept analysis of execution traces. In Proc. of the 11th WCRE 2004, 2004, pp. 112–121. DOI: 10.1109/WCRE.2004.13.

[25]. Ceccato M., Marin M., Moonen L. Applying and combining three different aspect mining techniques. Software Quality Journal, vol. 14, no. 3, 2006, pp. 209–231. DOI: 10.1007/s11219-006-9217-3.

[26]. Tomassetti F., Rizzo G., Torchiano M. Spotting automatically cross-language relations. In Proc. of the 2014 Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 338–342. DOI: 10.1109/CSMR-WCRE.2014.6747189.

[27]. Inozemtseva L., Subramanian S., Holmes R. Integrating software project resources using source code identifiers. In Proc. of the 36th International Conference on Software Engineering, New Ideas and Emerging Results (ICSE-NIER), 2014, pp. 400–403. DOI: 10.1145/2591062.2591108.

[28]. Konopka M. Combining Eye Tracking with Navigation Paths for Identification of Cross-Language Code Dependencies. In Proc. of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015), Bergamo, Italy. ACM, 2015, pp. 1057–1059. DOI: 10.1145/2786805.2807561.

[29]. Sharafi Z., Bertram I., Flanagan M., Weimer W. Eyes on code: A study on developers' code navigation strategies. IEEE Transactions on Software Engineering, vol. 48, no. 5, 2022, pp. 1692–1704. DOI: 10.1109/TSE.2020.3032064.

[30]. Nam D., Macvean A., Hellendoorn V. J., Vasilescu B., Myers B. Using an LLM to Help With Code Understanding. In Proc. of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), 2024, Article 97, pp. 1–13. DOI: 10.1145/3597503.3639187.

[31]. Patil A., Pachpute S. Unified Deep Semantic Search on Code. International Journal of Engineering and Advanced Technology (IJEAT), vol. 9, no. 5, 2020, pp. 872–876. DOI: 10.35940/ijeat.E9861.069520.

[32]. Paltenghi M., Pandita R., Henley A. Z., Ziegler A. Follow-up Attention: An Empirical Study of Developer and Neural Model Code Exploration. IEEE Transactions on Software Engineering, vol. 50, no. 10, 2024, pp. 2568–2582. DOI: 10.1109/TSE.2024.3445338.
[33]. Pandey R., Singh P. Transforming Software Development: Evaluating the Efficiency and Challenges of GitHub Copilot in Real-World Projects. arXiv preprint, 2024. DOI: 10.48550/arXiv.2406.17910.
[34]. Wermelinger M. Using GitHub Copilot to Solve Simple Programming Problems. In Proc. of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada. ACM, New York, NY, USA, 7 p. DOI: 10.1145/3545945.3569830.
[35]. Rahman M. M., Kundu A. Code Hallucination. arXiv preprint, 2024. DOI: 10.48550/arXiv.2407.04831.
[36]. Tian Y., Yan W., Yang Q., Zhao X., Chen Q., Wang W., Luo Z., Ma L., Song D. CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification. Proceedings of the AAAI Conference on Artificial Intelligence, 2025, vol. 39, no. 24, pp. 25300–25308. DOI: 10.1609/aaai.v39i24.34717.
[37]. GraphQL: Language Feature Support. Extension for Visual Studio Code. Available at: https://marketplace.visualstudio.com/items?itemName=GraphQL.vscode-graphql, accessed 10.04.2026.
[38]. GraphQL. Extension for Visual Studio Code (Orsen Kucher). Available at: https://marketplace.visualstudio.com/items?itemName=orsenkucher.vscode-graphql, accessed 10.04.2026.
[39]. Apollo GraphQL. Extension for Visual Studio Code. Available at: https://marketplace.visualstudio.com/items?itemName=apollographql.vscode-apollo, accessed 10.04.2026.
[40]. Gqlgen. Library for building GraphQL servers in Go. Available at: https://github.com/99designs/gqlgen, accessed 10.04.2026.
[41]. Gopls. Go language server. Available at: https://pkg.go.dev/golang.org/x/tools/gopls, accessed 10.04.2026.
[42]. GraphiQL. GraphQL LSP ecosystem. Available at: https://github.com/graphql/graphiql/tree/main, accessed 10.04.2026.
[43]. Moonen L. Generating robust parsers using island grammars. In Proc. of the 8th Working Conference on Reverse Engineering (WCRE 2001), Stuttgart, Germany, 2–5 Oct. 2001. IEEE Computer Society, 2001, pp. 13–22. DOI: 10.1109/WCRE.2001.957806.
[44]. Gorchakov A. V., Demidova L. A. Methods and Algorithms for Cross-Language Search of Source Code Fragments. In Proc. of the 2024 International Conference on Information Technologies (InfoTech 2024), Sofia, Bulgaria. IEEE, 2024, pp. 1–4. DOI: 10.1109/InfoTech63258.2024.10701403.
[45]. Graphql-go-tools. GraphQL router and API gateway framework in Go. Available at: https://github.com/wundergraph/graphql-go-tools/, accessed 10.04.2026.
[46]. Graphql-go. GraphQL server for Go with a focus on ease of use. Available at: https://github.com/graph-gophers/graphql-go, accessed 10.04.2026.
[47]. Gqlparser. GraphQL parser for Go. Available at: https://github.com/vektah/gqlparser, accessed 10.04.2026.
[48]. Quick Navigation. Available at: https://github.com/dmitry-drozdov/quick_nav, accessed 10.04.2026.
[49]. Bodner J. Learning Go: An Idiomatic Approach to Real-World Go Programming. O'Reilly Media, Sebastopol, CA, 2024. 468 p.
[50]. Freeman A. Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang. Apress, Berkeley, CA, 2022. 1076 p.
[51]. Дроздов Д. С., Михалкович С. С. Валидация легковесных грамматик языка Go. В сборнике: Современные информационные технологии: тенденции и перспективы развития: материалы XXXII научной конференции. 2025, стр. 156–160. / Drozdov D. S., Mikhalkovich S. S. Validation of lightweight Go language grammars. In: Modern information technologies: trends and prospects of development: Proceedings of the XXXII Scientific Conference. 2025, pp. 156–160 (in Russian).
[52]. Azure Service Operator for Kubernetes. Available at: https://github.com/Azure/azure-service-operator, accessed 10.04.2026.
[53]. Kubernetes. Production-grade container orchestration system. Available at: https://github.com/kubernetes/kubernetes, accessed 10.04.2026.
[54]. Docker CE. Archived repository. Available at: https://github.com/docker-archive/docker-ce, accessed 10.04.2026.
[55]. Sourcegraph Public Snapshot. Code AI platform with code search and Cody. Available at: https://github.com/sourcegraph/sourcegraph-public-snapshot, accessed 10.04.2026.

[56]. Chainlink. Decentralized oracle network node. Available at: https://github.com/smartcontractkit/chainlink, accessed 10.04.2026.
[57]. Boost. Tool for Filecoin storage providers. Available at: https://github.com/filecoin-project/boost, accessed 10.04.2026.
[58]. Core-geth. Configurable Go implementation of the Ethereum protocol. Available at: https://github.com/etclabscore/core-geth, accessed 10.04.2026.
[59]. Exo. Process manager and log viewer for development. Available at: https://github.com/deref/exo, accessed 10.04.2026.
[60]. HydroAPI. Public API for Hydro. Available at: https://github.com/HydroOSS/HydroAPI, accessed 10.04.2026.
[61]. Mattermost Playbooks plugin. Available at: https://github.com/mattermost/mattermost-plugin-playbooks, accessed 10.04.2026.
[62]. SlabAPI. API backbone for the Slab project. Available at: https://github.com/TylerConlee/SlabAPI, accessed 10.04.2026.
[63]. Sturdy. Open-source real-time version control platform. Available at: https://github.com/sturdy-dev/sturdy, accessed 10.04.2026.
[64]. Toolkit for Cardano. Smart contract development tools for Cardano. Available at: https://github.com/SundaeSwap-finance/toolkit-for-cardano, accessed 10.04.2026.
[65]. Card S. K., Moran T. P., Newell A. The Psychology of Human-Computer Interaction. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983. 469 p.
[66]. Nielsen J. Usability Engineering. Morgan Kaufmann Publishers, San Francisco, CA, 1994. 362 p.
[67]. Singh A., Henley A. Z., Fleming S. D., Luong M. V. An Empirical Evaluation of Models of Programmer Navigation. In Proc. of the ICSME 2016, USA, 2016, pp. 9–19. DOI: 10.1109/ICSME.2016.84.
[68]. Lazar J., Feng J. H., Hochheiser H. Research Methods in Human-Computer Interaction. 2nd ed. Cambridge, MA, Morgan Kaufmann Publishers (an imprint of Elsevier), 2017. 534 p.

Информация об авторах / Information about authors

Дмитрий Сергеевич ДРОЗДОВ – аспирант Южного федерального университета по научной специальности «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей». В 2023 году получил степень магистра по направлению подготовки «Фундаментальная информатика и информационные технологии». Область интересов: языки программирования, компиляторы, грамматики.

Dmitry Sergeevich DROZDOV – postgraduate student of the Southern Federal University, scientific specialty "Mathematical and software support for computing systems and computer networks". In 2023, he received his master's degree in the specialty "Fundamental informatics and information technology". Area of interest: programming languages, compilers, grammars.

Станислав Станиславович МИХАЛКОВИЧ – кандидат физ.-мат. наук, доцент, заведующий кафедрой информатики и вычислительного эксперимента Южного федерального университета, руководитель проекта PascalABC.NET. Основные научные интересы: разработка компиляторов, управление прорезающей функциональностью в программах, теория типов.

Stanislav Stanislavovich MIKHALKOVICH – Cand. Sci. (Phys.-Math.), Assoc. Prof., Head of the Department of computer science and computational experiments at the Southern Federal University, Head of the PascalABC.NET project. Main research interests: compiler development, management of crosscutting concerns in programs, type theory.