СОДЕРЖАНИЕ

Введение 9

1 Исследование существующих технологий и языков программирования в контексте областей разработки программного обеспечения 11

1.1 Современная индустрия разработки программного обеспечения . 11

1.1.1 Актуальность мультиязыкового статического анализа 12

1.2 Особенности мультиязыкового анализа 14

1.2.1 Сложности распознавания текста 15

1.2.2 Анализ и кодогенерация 16

1.2.3 Формирование выборки для анализа 19

1.2.4 Динамическая и неявная природа межъязыковых связей 20

1.2.5 Разнообразие парадигм программирования 20

1.2.6 Результаты исследования 21

2 Анализ состояния работ в области анализа межъязыковых исходных текстов программ 23

2.1 Теоретические и практические труды в отношении межъязыкового анализа 23

2.2 Pangea 23

2.3 Статический анализ JNI связей 25

2.4 MLSA 26

2.5 Граф вызовов, семантики и номинальной схожести 27

2.6 Polycall 28

2.7 Результаты анализа состояния работ 30

3 Постановка задачи, формирование требований к методу 31

3.1 Цели и задачи 31

3.2 Требования к методу анализа 31

4 Анализ существующих подходов к статическому анализу, проектирование архитектуры и исследование средств реализации 33
4.1 Виды структур представления информации в статическом анализе 33
4.2 Архитектура метода анализа 36
4.2.1 Парсинг мультиязыковых фрагментов кода 36
4.2.2 Унифицированное представление через ограничения 38
4.2.3 Вовлечение операционного окружения 44
4.2.4 Онтология 46
4.3 Исследование средств реализации 51
5 Описание основных сценариев использования анализатора в сочетании с инструментальными средствами программирования 53
5.1 Интеграция в различные инструментальные средства 53
5.2 Реализация LSP как одного из сценариев интеграции метода 55
6 Программная реализация анализатора и инфраструктуры 57
6.1 Общая архитектура 57
6.1.1 Составляющие анализатора 57
6.1.2 Протоколы взаимодействия 59
6.1.3 Реализация онтологии 60
6.2 Провайдер конфигурации и кода 61
6.3 Мультиязыковой транслятор 61
6.4 Решатель ограничений 63
6.5 Адаптер инструментального средства 64
7 Оценка характеристик метода на избранных тестовых проектах 67
7.1 Пример 1 – C# и Visual Basic 67
7.1.1 Общее описание 67
7.1.2 Окружение 68

7.1.3 Трансляция и решение 68
7.1.4 Онтология 69
7.1.5 Сценарии LSP 70
7.2 Пример 2 – C++, Python 3 и Shell 71
7.2.1 Общее описание 71
7.2.2 Окружение 71
7.2.3 Трансляция и решение 71
7.2.4 Онтология 72
7.2.5 Сценарии LSP 73
7.3 Пример 3 – Golang и JavaScript 74
7.3.1 Общее описание 74
7.3.2 Окружение 74
7.3.3 Трансляция и решение 74
7.3.4 Онтология 76
7.3.5 Сценарии LSP 76
Заключение 79
Список использованных источников 80
Приложение А Протокол мультиязыкового анализатора 84

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящем отчете о НИР применяют следующие термины с соответствующими определениями.

Семантика – смысловое содержание синтаксических конструкций программ.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

DSL – domain specific language — язык разработанный для определенной предметной области.

СУБД – система управления базами данных.

IDE – integrated development environment — интегрированная среда разработки.

GPL – general programming language — язык общего назначения.

ФС – файловая система.

ОО – object oriented — объектно ориентированный.

CDG – call-dependency graph — граф зависимостей вызовов.

ВЛСА – вероятностный латентно-семантический анализ.

AST – abstract syntax tree — абстрактное синтаксическое дерево.

CST – concrete syntax tree — частное синтаксическое дерево.

CFG – control flow graph — граф потока управления.

LSP – language server protocol — протокол языкового сервера.

ВВЕДЕНИЕ

Особенностью современной индустрии разработки программного обеспечения является применение нескольких языков программирования в одном программном проекте. Данная особенность порождает проблему проверки согласованности фрагментов кода, реализованных на разных языках и применяемых в составе одной программной системы. Как правило, такая несогласованность отслеживается только на этапе отладки или тестирования.

Одним из вариантов решения данной проблемы может стать проведение статического семантического анализа кода с выявлением несогласованности в таком коде автоматически. Для решения такого рода задач существуют различные инструментальные средства и часто такие средства встраиваются в среду разработки для поддержки разработчика в реальном времени.

Однако, на данный момент методы проведения статического анализа мультиязыкового кода развиты слабо. Большая часть методов ориентирована на специфические языки или проекты и плохо подходит для интеграции в инструментальные средства в общем случае. Вследствие этого мультиязыковые средства поддержки разработчика слабо развиты и обычно ограничены кругом узкоспециализированных проприетарных решений.

Метод мультиязыкового анализа, позволяющий гибко анализировать фрагменты кода на разных языках и независимый относительно анализируемых технологий является одним из решений данной проблемы.

Актуальность темы исследования обусловлена малым развитием инструментальных средств поддержки разработчика, ориентированных на мультиязыковой анализ. Такое положение вещей снижает продуктивность разработчиков и увеличивает время, отводимое на процессы кодирования и отладки.

Объектом исследования являются исходные тексты и семантические модели программ, написанных с использованием нескольких языков программирования.

Предметом исследования являются методы и алгоритмы анализа исходных текстов программ, ориентированных на мультиязыковой статический анализ.

Целью работы является упрощение процессов разработки ПО, путем создания метода выявления межъязыковых связей, позволяющего обеспечивать корректность и расширяемость соответствующих средств.

Для достижения данной цели необходимо решить следующие **задачи**:

1) изучить предметную область, а именно индустрию разработки ПО,

2) провести исследование текущих решений в области межъязыкового анализа,

3) разработать требования к методу и спроектировать его архитектуру,

4) описать прикладные сценарии использования метода и соответствующего анализатора,

5) создать первичную реализацию анализатора, поддерживающего определенный набор языков,

6) провести оценку показателей анализатора на избранных тестовых проектах.

Научная новизна работы заключается в создании метода анализа, позволяющего разрабатывать различные инструментальные средства, направленные на поддержку разработчика в процессе кодирования и отладки. При этом, такой метод позволит обеспечить корректность семантического анализа на межъязыковом уровне.

Практическая значимость работы выражается в создании прототипа мультиязыкового анализатора, обеспечивающего поддержку реализации сценариев LSP, что позволяет создавать «мультиязыковые» серверы LSP.

1 Исследование существующих технологий и языков программирования в контексте областей разработки программного обеспечения

1.1 Современная индустрия разработки программного обеспечения

Современные программные проекты, в отличие от многих программных проектов прошлого, гораздо чаще состоят из набора разных (порой разительно) технологических решений, предназначенных для решения определенного круга задач. Согласно [1], в одном программном проекте в среднем задействовано 5 языков программирования, один из которых является «основным», а остальные – специализированными языками предметной области (DSL). В заключении статьи авторы признают популярность мультиязыковых программных проектов и оценивают важность наличия соответствующих инструментальных средств для работы с такого рода проектами.

Также, нередко использование нескольких языков и в индустрии разработки. Так, авторы [2] провели исследование популярности мультиязыковых проектов в результате опроса 139 профессиональных разработчиков из разных сфер. Результаты опроса показали, что опрашиваемые имели дело с 7 различными языками, в среднем. При этом, в работу было вовлечено в среднем 3 пары связанных языков в контексте одного проекта. Более 90% опрашиваемых также сообщали о проблемах согласованности между языками, встречаемых при разработке в такой мультиязыковой среде.

Таким образом, в современной разработке программного обеспечения нередко использование нескольких языков вне зависимости от объемов проекта или вовлекаемой предметной области. Ситуация становится сложнее со временем, так как создание новых технологий разработки часто влечет за собой формирование определенной нотации или языка для управления или конфигурации. Например, это может касаться таких повсеместных технологий как СУБД, систем сборки, серверов приложений или скриптов развертывания.

Для наглядности, можно привести следующие языки, нередко фигурирующие в составе современных программных проектов:

– язык разметки HTML в составе проекта, использующего ASP фреймворк,

– язык скриптов командной строки в составе проекта, использующего язык C,

– язык запросов SQL в составе проекта, использующего Python и фреймворк Flask,

– язык препроцессора в составе файла исходного кода, реализованного на C++.

Заключительный пункт списка примеров приведен для того, чтобы показать характер связи различных технологий – разным языкам необязательно даже находится в раздельных файлах или модулях, нередки случаи полноценного переплетения различных синтаксисов и семантик.

1.1.1 Актуальность мультиязыкового статического анализа

Итак, мультиязыковые программные проекты нередки. Следовательно, имеет смысл использования различных техник работы с исходным кодом таких проектов, поддерживающих процесс разработки. Одной из таких техник является статический анализ исходного кода. Его основной сценарий использования это упрощение различных сценариев процесса разработки ПО – кодирования, верификации, рефакторинга и других.

Стоит уточнить что имеется под определением «статический анализ кода». Статический анализ это прежде всего набор различных техник по извлечению информации о программе без её явного запуска [3]. Таким образом, статический анализ может быть полезен в сценариях, которые не предполагают явного запуска программы – в сущности во всех сценариях процесса разработки ПО исключая этапы тестирования и запуска.

Возможные сценарии использования статического анализа кода включают (но не ограничиваются):

– оптимизацию программ во время компиляции,

– выявление потенциальных уязвимостей,

– доказательство сохранения определенных инвариантов,

– сбор определенной статистики,

– выявление «пахнущих» фрагментов кода,

– извлечение информации для помощи разработчику во время кодирования (линтинг),

– автоматический рефакторинг кода.

Так как сценарии использования статического анализа настолько разнообразны, в рамках данной работы решено было сосредоточится на сценариях разработки, которые помогают в процессе кодирования и поддержки проекта. Существуют различные инструментальные средства, способные упрощать такие сценарии. К таким средствам, к примеру, относятся:

– интегрированные средства разработки (IDE),

– линтеры (собирательное название инструментов, первым из которых был «Lint» [4]),

– инструменты автоматического рефакторинга,

– инструменты сбора статистики,

– различные кодогенераторы и фреймворки [5][6].

Необходимость в таких средствах присутствует и она достаточно высока. Так, согласно исследованию [7], использование средств поддержки разработчика (в данной работе это механизмы анализа и навигации по межъязыковым связям) позволяет улучшить как скорость разработки ПО, так и уменьшить количество совершаемых ошибок. Стоит заметить, что несмотря на количество времени, прошедшее с момента проведения исследования, принципы разработки ПО в данной предметной области (веб-разработка) не изменились и большинство программных проектов веб-приложений состоят как

минимум из двух языков. Обычно это разделение проводится по принципу фронтенд и бекенд.

Согласно обзорной статье об исследовании межъязыковых связей [8], современные средства межъязыкового и мультиязыкового анализа остаются всё еще плохо развитыми. Особенно это касается отношения корректности таких средств. Более того, большая их часть часто является частным решением, ориентированным на определенную предметную область или технологию, что снижает универсальность таких средств и усложняет их интеграцию.

Так как статья содержит большую выборку литературы по обнаружению и анализу межъязыковых связей, авторам удается установить широкую область применения межъязыкового анализа. В эту область входят такие дисциплины разработки ПО как:

- семантическое понимание программ,
- представление знаний при сопровождении систем,
- рефакторинг,
- моделирование ПО,
- валидация и верификация систем,
- визуализация систем,
- сбор статистик и метрик.

Как видно из списка, большинство этих дисциплин значительно пересекаются со сценариями использования статического анализа кода, что говорит о том что реализация такого анализа может быть очень актуальной для обеспечения упрощения и удешевления большого количества этапов разработки ПО.

1.2 Особенности мультиязыкового анализа

Мультиязыковой анализ вовлекает множество особенностей существующих инструментов, так как косвенно или непосредственно семантическая информация об этих инструментах должна быть вовлечена в процесс анализа.

Поэтому стоит рассмотреть особенности мультиязыкового анализа которые затрудняют разработку соответствующих анализаторов и процесс анализа в целом.

1.2.1 Сложности распознавания текста

Так как вовлекаемые фрагменты кода написаны на разных языках, проблема анализа таких фрагментов возникает уже на самом первом этапе – этапе распознавания текста. Обычно процесс анализа текста с определенной грамматикой не вызывает труда так как грамматика фиксирована и анализатор строится стандартными средствами – парсер генераторы, рекурсивный спуск и иные.

Однако, в случае парсинга текста на нескольких языках возникают сложности, так как создание универсальной грамматики (для всех анализируемых языков) обычно трудоемко и порой в целом невозможно. Таким образом остается несколько способов распознавания текста:

1) распознавание и анализ текстов по отдельности,

2) использование смешанных парсеров (например с использованием островных грамматик [9]),

3) использование единого представления для унификации различных парсеров.

Первый пункт является простым в реализации и работает в большинстве случаев, однако возникают сложности при анализе единого фрагмента (например, файла), который вовлекает несколько языков в анализ (к примеру CSS стили в заголовке страницы HTML). Также, такой подход не позволяет проводить совместный анализ и выявлять связи между модулями, реализованными на разных языках.

Второй пункт позволяет анализировать в том числе смешанные грамматики, но сборка таких парсеров довольно трудоемка и ограничена поддержкой лишь определенного количества языков.

Таким образом, наиболее оптимальным является вариант распознавания при котором мультиязыковые фрагменты изначально переводятся в единое представление (используя любой специфичный способ перевода, сохраняющий семантику), а уже затем анализируются. В некоторых случаях такой подход является наиболее удобным так как интересующая область анализа может вовлекать только специфическую часть информации, представленную в коде. В таком случае, универсальное представление может быть создано с учетом только такой информации и в результате получиться гораздо проще исходного языка. Это позволяет упростить анализатор и облегчить его дальнейшую поддержку.

1.2.2 Анализ и кодогенерация

Несмотря на то, что кодогенерация является сложностью которую приходится учитывать и при обычном статическом анализе, в мультиязыковых проектах эта сложность выходит на новый уровень. Это происходит в первую очередь потому, что большое количество технологических решений, вовлекающих разные языки используют разные подходы к кодогенерации и по разному её осуществляют.

Для понимания возможных проблем, рассмотрим следующий фрагмент кода на C++:

```
1   // function.h
2   #include <stdio.h>
3   template <typename T>
4   T f(T a) {
5   #ifdef SOMEBAR
6       return a * 2;
7   #else
8       return a + 2;
9   #endif
10  }
11
12  // main.cpp
```

```
13      #include "function.h"
14      int main()
15      {
16      #ifdef SOMEFOO
17          printf("%f", f<float>(10));
18      #else
19          printf("%d", f<int>(5));
20      #endif
21          return 0;
22      }
```

Представим что есть два файла, «function.h» и «main.cpp». Фрагменты, соответствующие файлам, помечены комментариями. При статическом анализе такого кода для определения того, какая версия функции в итоге будет сгенерирована при компиляции необходимо сделать следующее:

1) узнать какие значения принимают переменные среды при компиляции этих файлов,

2) для каждого из таких значений сгенерировать возможную конфигурацию кода,

3) проанализировать каждую конфигурацию и объединить результаты анализа.

В отношении данного примера, возможных конфигураций будет четыре. В общем случае количество конфигураций можно определить как мощность декартового произведения множеств значений вовлеченных переменных. Нетрудно представить, что такое количество может быть достаточно большим, например при количестве переменных равном 10 с учетом того что эти переменные принимают только 2 значения, количество конфигураций равно 1024.

Еще одной особенностью кодогенерации является частое вовлечение сторонних инструментов. В отличие от примера приведенного выше, многие фреймворки различных технологических стеков полагаются на кодогенерацию во время сборки проекта. Обычно эта кодогенерация вовлекает генера-

цию на уровне файлов. В данном случае статический анализ не сможет обеспечить анализ сгенерированных файлов без использования одного из двух подходов:

– запуск сборки и генерации во временной директории с последующей очисткой,

– абстрактная интерпретация команд генератора.

Первый подход может вызвать большие сложности если генерация имеет в зависимостях какое-либо недоступное на момент анализа окружение. Примером может быть генерация «файлов-определений» для проекта, который должен быть использован на конкретном встроенном устройстве. Такие файлы обычно являются способом объединения конфигурации программного кода и информации для устройства. К примеру, таким файлом является `c++config.h` который часто идет в поставке с избранным компилятором C++ и является встроенным заголовочным файлом. Нетрудно представить, что такой файл может различаться от компилятора к компилятору и даже от версии к версии, а также просто может не существовать, так как стандарт C++ не затрагивает встроенные определения. До момента размещения проекта в специализированном окружении, нет информации о том какого рода определения необходимо генерировать для анализа. Такое часто происходит при разработке встроенных систем по разным причинам: количество конфигураций устройств, упрощение процесса разработки, недоступность устройств и т.д.

Второй подход является более сложным в реализации, так как различные кодогенераторы и фреймворки редко имеют исчерпывающую документацию по особенностям кодогенерации, поэтому кодирование такой информации часто будет неполным. Более того, особенности кодогенерации могут быть особенностью реализации определенного инструмента, что может поменяться со временем. Поэтому, практическая реализация таких анализаторов это сложный процесс, требующий большого количества времени и поддержки.

1.2.3 Формирование выборки для анализа

На первый взгляд формирование выборки кажется тривиальной проблемой. При моноязыковом анализе она решается использованием сторонних инструментов, позволяющих получить набор файлов для анализа. Такую информацию могут предоставлять системы сборки или развертывания, в крайнем случае достаточно поиска в файловой системе определенного множества файлов.

Однако, при вовлечении нескольких языков в проект, получение соответствующих файлов путем простого поиска может быть неадекватным решением по следующим причинам:

– компоновка проекта из отдельных, несвязанных проектов,

– сборка проекта зависящая от условий в текущем окружении,

– специфика компоновки файлов в единый модуль зависит от конкретного языка и может отличаться.

Соответственно, для успешной практической интеграции анализатора, способного проводить мультиязыковой анализ необходимо обеспечить также поддержку внешних инструментов, позволяющих правильно выбирать файлы для анализа основываясь на конфигурации операционного окружения проекта. Таким образом, анализ проекта первым этапом обязан обеспечивать анализ *операционного окружения* проекта. Такая особенность сильно усложняет анализатор и накладывает ограничения на технологии с которыми он может работать, так как операционное окружение в общем случае включает состояние всего компьютера: файловая система, переменные среды или реестр, подключение к сети и т.д.

В данной работе проблема выборки файлов для анализа не рассматривается, так как она несущественна для формирования метода межъязыкового анализа, который независим от какого-либо анализатора, его использующего.

1.2.4 Динамическая и неявная природа межъязыковых связей

Многие семантически важные связи в реальных мультиязыковых проектах являются неявными. В большей части это применимо к фреймворкам и системам сборки, так как порой семантическая информация распределена в разных местах проекта, вовлекая при этом и системные зависимости [2] [10].

С точки зрения практической реализации, неявные связи являются усложняющим обстоятельством, но не делающим анализ невозможным. К сожалению, то же нельзя сказать в отношении динамических связей. Под динамическими связями понимаются связи, которые устанавливаются на определенном этапе исполнения кода. Например, это может быть формирование строки в коде JavaScript, которая потом может быть использована чтобы получить определенные теги HTML страницы.

Несмотря на наличие динамических связей в определенных языках, значительное множество языков общего назначения (GPL), все же имеют статическую типизацию либо статическое время связывания идентификаторов, не вовлекающее процесс вычисления. Это позволяет проводить эффективный статический анализ таких языков. Однако, в случае гетерогенных проектов наличие динамической связи гораздо более распространено. Это связано с различными причинами, основная из которых, возможно, связана с разнообразием связываемых между собой технологий.

Например, очень частой динамической (но также вполне явной) связью является зависимость фрагмента от состояния ФС . Это практически всегда встречается в различных системах сборки – скрипты сборки напрямую зависят от наличия некоторых файлов в системе и при их отсутствии вынуждены прерывать исполнение сборки, порой после нескольких часов работы. Таким образом тратятся как физические ресурсы, так и снижается продуктивность программиста, что может приводить к ухудшению процесса разработки.

1.2.5 Разнообразие парадигм программирования

Как уже сказано в предыдущем подпункте, значительное множество GPL являются статически типизированными либо имеют статическое время

связывания идентификаторов. Однако, это касается по большей части императивных языков программирования. Примерами могут служить Java, C#, C++, Golang и другие. Многие DSL однако, являются динамическими языками в том смысле, что они имеют позднее связывание идентификаторов (обычно во время исполнения). Связано это в том числе с тем, что многие DSL имеют другую парадигму программирования, что также влияет на особенность их реализации. Примерами могут служить языки сборки, языки командной оболочки или языки выборки данных (например, языки запросов к СУБД).

Многие фреймворки «программируются» не напрямую, а через конфигурационные файлы. Примером может служить повсеместный язык разметки XML, который иногда эксплуатируется как язык описания каких-либо выражений на определенном DSL. Ярким примером может служить система сборки Apache Ant, в ней узлы XML дерева могут представлять обращения к переменным или даже целые выражения. Можно сказать, что любые конфигурационные данные проекта могут представлять семантически важную информацию, так как использование конфигурационных файлов в гетерогенных проектах повсеместно [10].

Это приводит к тому, что некоторые вовлекаемые в анализ языки не являются языками программирования, но всё же представляют семантическую ценность для анализа. Вследствие этого возникает проблема создания метода такого анализа, который в том числе будет вовлекать такого рода фрагменты.

1.2.6 Результаты исследования

Подводя итоги раздела 1, можно сказать что современные проекты действительно часто реализованы на различных языках, т.е. являются гетерогенными. Такой положение вещей затрудняет многие сценарии разработки ПО и создание метода анализа обеспечивающего выявление межъязыковых связей является хорошим способом решения многих возникающих проблем.

Также, на основе подпункта 1.2 можно выявить желаемые особенности метода мультиязыкового анализа:

1) анализ с использованием единого представления является более эффективным решением ввиду большей универсальности,

2) желательна поддержка сценариев кодогенерации, либо возможность их реализации,

3) необходимо вовлечение операционного окружения проекта для обеспечения корректности анализа,

4) эффективный анализ должен быть универсальным для многих языков и парадигм программирования.

2 Анализ состояния работ в области анализа межъязыковых исходных текстов программ

2.1 Теоретические и практические труды в отношении межъязыкового анализа

На данный момент область межъязыкового анализа (в т.ч. не только статического) является «terra incognita» – в том смысле, что многие исследования на данную тему несут частный характер и не имеют четко структурированной теоретической базы. Также, существуют концептуальные и онтологические сложности в отношении этой области знаний. Например, для определения термина «язык программирования» в данной области недостаточно рассматривать общепринятые варианты языков, так как чаще всего они являются GPL, чего недостаточно в контексте межъязыкового анализа. Термин «формальный язык» является более подходящим, но сильно расширяет предметную область, что затрудняет создание единой терминологической базы.

Несмотря на данные сложности, работы по межъязыковому анализу являются очень разнообразными и, рассмотрение определенного количества таких работ позволит выявить общие аспекты анализа, которые универсальны для всех методов. В данной работе была проведена выборка пяти различных работ по следующим показателям:

- парадигма анализируемых языков,
- способ извлечения знаний (вовлекаемая онтологическая модель),
- вовлекаемые языки,
- сценарии использования анализаторов,
- возможные недостатки подхода.

2.2 Pangea

Pangea [11] это фреймворк для осуществления статического анализа, позволяющий быстро разрабатывать различные анализаторы путем использования снимков системы (англ Object Model Snapshots). Такие снимки являют-

ся исполняемыми образами моделей Moose [12], которые в свою очередь получаются из анализа исходного кода с использованием *метамоделей* FAMIX [13].

Фреймворк предназначен в первую очередь для анализа систем на основе определенных метрик, которые в свою очередь собираются скриптом, реализованным на Smalltalk. В статье авторами показаны различные сценарии использования фреймворка (в основном для Java приложений) – сборка метрик и статистик, с последующим анализом. Авторами заявляется поддержка других языков (C/C++, C#, Smalltalk) путем использования других анализаторов, предоставляющих результаты анализа в формате FAMIX.

Несмотря на то, что фреймворк может быть использован для многих языков, его средствами нельзя достичь именно межъязыкового анализа, так как такой анализ подразумевает обработку межмодульных зависимостей, чего данный фреймворк делать не позволяет. Также, одним из серьезных ограничений (в том числе замеченном авторами) является ориентированность на ОО языки, что ограничивает круг поддерживаемых языков.

Стоит заметить, что применение метамоделей на основе ОО парадигмы может затруднять разработку соответствующих анализаторов. Основная причина этого – понятия «класс» и «объект» являются довольно размытыми семантически и их формальная реализация может разительно отличаться от языка к языку. Основным примером такой сложности может служить сравнение языков JavaScript и Java.

Несмотря на внешнюю схожесть, JavaScript куда больше напоминает сочетание Smalltalk и Scheme, что делает его более гибким и, в каком-то отношении, функциональным языком. Отражение этого заключается к примеру в том, что до стандарта ES6 [14], JavaScript считался чисто объектным языком и не содержал классов. Многие «классовые» конструкции были реализованы через механизм прототипного наследования и обычные функции. В случае Java класс является основополагающей конструкцией, основным блоком построения программ. В связи с этим, многие семантические особенности Java (например, виртуальные вызовы или определения процедур) сильно связаны

с семантикой классов. К примеру, невозможно определить процедуру, без создания соответствующего класса.

В конце концов попытка объединения семантических понятий «класс» и «объект» сделает онтологическую модель слишком сложной так как будет вовлекать абсолютно разные концепции из обоих языков. Решением данной проблемы может стать использование «меньшего» унифицированного представления, т.е. сопровождающего достаточное количество информации о сущности языка и при этом являющимся «наибольшим общим делителем» среди различного набора языков.

2.3 Статический анализ JNI связей

JNI [15] это набор API для взаимодействия виртуальной машины Java с нативным кодом (к примеру, реализованным на C/C++ или ассемблере). Основная цель такого интерфейса – доступ к низкоуровневым функциям, позволяющим ускорить исполнение кода на Java.

Как следует из описания, написание кода с использованием JNI влечет взаимодействие нескольких языков между собой, в случае рассматриваемой статьи это Java и C++. В рамках статьи рассматривается статический анализ на основе «S-MLDA». Это разработанный авторами анализатор, позволяющий строить граф зависимостей вызовов (CDG) между фрагментами кода реализованными на двух языках. Процесс построения такого графа заключается в парсинге и дальнейшем лексическом сравнении идентификаторов. В дальнейшем, такой граф позволяет выявить связи компонентов и интерпретировать их должным образом (например, сообщить о несогласованности).

Основным недостатком метода является процесс лексического сравнения для реализации модели анализа. Хотя и простой, такой метод является нестабильным и сложно поддерживаемым по следующим причинам:

– имена идентификаторов неявно вовлекаются в процесс анализа семантики,

– возможны ложноположительные или ложноотрицательные результаты анализа при различном именовании,

— от версии к версии имена могут меняться, что сделает анализ менее полным.

Таким образом, метод анализа семантики посредством лексического сравнения является довольно хрупким решением, поддерживающим малый набор языков. Также, такое решение является трудно поддерживаемым и расширяемым в дальнейшем.

Также стоит заметить что построение графа зависимостей вызовов является достаточно хорошим решением для многих языков, так как широкий спектр межъязыкового взаимодействия может быть выражен через вызовы функций из одного модуля в другом. С практической точки зрения такой подход является, пожалуй, наиболее простым в реализации, однако такая модель сохраняет очень мало информации о семантике исходного языка, что затрудняет дальнейшее расширение анализаторов на основе такой модели.

2.4 MLSA

MLSA (MultiLingual Software Analysis) [16] это метод, ориентированный на построение межъязыковых анализаторов путем организации определенного конвеера из анализаторов двух категорий – языкозависимых и языконезависимых. Используя исходное AST программы на определенном языке программирования, авторами предполагается создание транслятора из такого AST в обобщенную форму (в статье такой транслятор называется «interop фильтр»), а затем анализ такой обобщенной формы в контексте нескольких языков (такой анализатор называется «MLSA фильтр»). Трансляторы являются языкозависимыми, а анализаторы языконезависимыми.

Также, стоит отметить, что авторами активно используется островная грамматика [9] для распознавания только интересующих фрагментов кода. Таким образом, сочетание различных фильтров и островная грамматика в их основе создает гибкую и модульную систему при которой возможно создание различных анализаторов различной сложности.

Однако, основной проблемой данного подхода является рост числа необходимых анализаторов при увеличении количества взаимодействующих

языков. Допустим, при анализе 10 языков, взаимодействующих между собой, необходимо будет создать 10 языкозависимых трансляторов и 50 языконезависимых анализаторов, организованных в виде дерева. Конечно, такой случай является утрированным, однако как уже было сказано в разделе 1, в современном программном проекте в среднем содержится 5 языков.

Данный метод как и предыдущий полагается на использование графа зависимостей вызовов, что сужает его применимость в контексте языков, не имеющих функций (например форматы данных).

2.5 Граф вызовов, семантики и номинальной схожести

В статье [17] авторами решается проблема модуляризации межъязыкового ПО путем построения специализированного графа и анализа такого графа с использованием генетического алгоритма. В рамках данной работы интерес представляет граф, использованный для отражения зависимостей в межъязыковом коде.

В статье используется нестандартных композитный граф, состоящий из трех различных графов:

1) граф зависимостей вызовов,
2) граф семантических зависимостей,
3) граф номинальной схожести.

Все три графа имеют общую основу – узлы представляют собой «артефакты». Семантика ребер при этом зависит от типа графа. Стоит разобрать каждый граф по отдельности.

Граф зависимостей вызовов представляет собой стандартный граф, встречаемый в других работах на данную тему. Ребра данного графа представляют собой зависимость «вызов», что в общем плане является зависимостью «использует». В статье граф является взвешенным, что позволяет авторам кодировать количество «использований» между артефактами как вес соответствующего ребра. Построение графа зависимостей вызовов является языкоза-

висимой процедурой и подразумевает использование соответствующего анализатора для каждого вовлеченного языка.

Граф семантических зависимостей строится интересным образом. Авторами используется ВЛСА для получения значения схожести двух артефактов. Артефакты объединяются между собой и вес ребер в графе при этом соответствует схожести соединенных узлов. Таким образом, метод построения графа является чисто лексическим и вовлекает статистические методы.

Граф номинальной схожести строится по похожему принципу что и предыдущий, однако в данном графе вес ребра отвечает не за семантическую схожесть понятий, а за лексическую. Гипотеза авторов состоит в том, что осмысленно разработанные системы часто имеют схожие назавания связанных между собой компонентов.

В итоге три таких графа объединяются и анализируются в дальнейшем генетическим алгоритмом. Данный граф является интересной моделью представления информации так как позволяет сильно увеличить полноту анализа, пренебрегая корректностью. Учитывая, что корректность анализа не требуется во всех сценариях использования статических анализаторов (допустим как в случае работы авторов), такой подход выглядит действительно адекватным для определенного круга задач. Также, как граф семантических зависимостей так и граф номинальной схожести не требуют языкозависимого анализатора для построения, что сильно упрощает реализацию такого анализа на практике.

2.6 Polycall

Статья [18] интересна в первую очередь выбором онтологической модели. Вместо создания какой-либо модели на основе формальной спецификации, авторами предполагается использование существующего формального языка с собственной семантикой. В случае данной статьи это язык WASM [19]. WASM является низкоуровневым языком программирования, ориентированным на встраивание в различные решения, в первую очередь браузеры.

В рамках данной статьи WASM рассматривается как «наибольший общий делитель», который позволяет объединить семантику различных языков путем простой компиляции из избранного языка в WASM модуль. Такой подход позволяет сильно упростить процесс анализа, так как языкозависимым анализатором в данном случае выступает компилятор и основная работа может заключаться в анализе непосредственного самих WASM модулей.

Несмотря на многообещающие особенности такого подхода, он имеет ряд недостатков.

Во-первых, при компиляции в такое унифицированное представление теряется ряд типовой информации, так как обычно она не требуется для исполнения. Это влечет за собой вероятную потерю основной информации, способной обеспечивать полноту анализа.

Во-вторых, способ компоновки текстов исходного языка может отличаться от способа компоновки текстов языка унифицированного представления. Одним из ярких примеров является процесс трансформации имен (англ. Name mangling) при компиляции кода C++. Это деталь реализации компилятора, которая позволяет осуществлять перегрузку функций по типам принимаемых параметров. Впоследствии, при связывании фрагментов C++, компилятор использует именно такие имена. Этот механизм мешает возможности связывания с скомпилированным кодом другого языка, например в случае модулей реализованных на C. В связи с этим в C++ существует директива `extern "C"`.

В-третьих, многие языки в ходе компиляции подвергаются процессу ловеринга. Ловеринг это процесс переписывания изначальной программы в терминах более примитивных языковых конструкций с сохранением семантики. К примеру, в контексте JavaScript определение класса является чистым синтаксическим сахаром и может быть переписано с использованием более базовых концепций – функций и прототипов. Процесс ловеринга может сделать процесс дальнейшего анализа сильно сложнее, а в некоторых случаях информация и вовсе стирается, что делает невозможным корректный семантический анализ. Например, виртуальные методы C++ часто понижаются в отдельное определение структуры называемое *виртуальная таблица*. Таким обра-

зом, исчезает возможность получить возможные виртуальные методы класса, так как соответствующая ему виртуальная таблица больше никак семантически не связана с классом.

2.7 Результаты анализа состояния работ

Подводя итоги анализа, можно сформировать классификацию данных работ по критериям, описанным в 2.1. Данная классификация приведена в таблице 2.1.

Таблица 2.1 — Состояние существующих анализаторов

Анализатор/ Метод	Парадигма	Способ извлечения знаний	Вовлекаемые языки	Сценарии использования	Недостатки подхода
Pangea (2.2)	OO	OMS и Moose модели	Java, C++, C#	Сборка метрик и анализ систем	Сложная онтологическая модель; Ограничение OO языками;
JNI связи (2.3)	OO/ Procedural	Лексическое сопоставление	Java/C++	Анализ FFI вызовов	Лексическое сравнение; Анализ исключительно функций;
MLSA (2.4)	OO/ Procedural	Interop/ Multilangual фильтры	C++/Python	Анализ FFI вызовов; Получение зависимостей	Большое количество межъязыковых анализаторов
Композитные графы (2.5)	Любая	Анализ вызовов, семантики и имен	Любые, C++/ JavaScript	IDE, инструменты визуализации	Отсутствие корректности; Частые ложноположительные результаты
Polycall (2.6)	Любая, поддерживающая WASM	Компиляция в унифицированное представление	C++, Golang, Rust	Получение зависимостей	Невыразимость представления с анализируемой семантикой

3 Постановка задачи, формирование требований к методу

3.1 Цели и задачи

Цель данной работы: «Упрощение процессов разработки ПО, путем создания метода выявления межъязыковых связей, позволяющего обеспечивать корректность и расширяемость соответствующих средств.»

Для достижения данной цели необходимо решить следующие задачи:

1) изучить предметную область, а именно индустрию разработки ПО,

2) провести исследование текущих решений в области межъязыкового анализа,

3) разработать требования к методу и спроектировать его архитектуру,

4) описать прикладные сценарии использования метода и соответствующего анализатора,

5) создать первичную реализацию анализатора, поддерживающего определенный набор языков,

6) провести оценку показателей анализатора на избранных тестовых проектах.

3.2 Требования к методу анализа

В требования к методу анализа входит:

1) возможность обнаружения межъязыковых зависимостей (в рамках фрагмента кода),

2) независимость в отношении анализируемых языков и технологий,

3) возможность гибкой настройки объемов анализа,

4) использование семантического представления с формально доказанными свойствами,

5) ориентированность на основные сценарии использования средств поддержки разработчика.

4 Анализ существующих подходов к статическому анализу, проектирование архитектуры и исследование средств реализации

4.1 Виды структур представления информации в статическом анализе

Из подраздела 2.7 становится ясно, что в отношении мультиязыкового анализа основополагающим механизмом является наличие общей структуры представления информации, позволяющей отражать различные сущности различных языков в единой форме. Выбор такой структуры может влиять на следующие характеристики метода анализа:

- разнообразие поддерживаемых языков,
- сложность разработки и поддержки анализаторов,
- корректность и полнота анализа,
- возможные сценарии использования.

В целом, выбор структуры влияет практически на все характеристики метода, поэтому стоит исследовать существующие структуры представления информации о программном коде.

AST представляет собой репрезентацию программного кода в древовидном формате. В качестве узлов дерева выступают различные синтаксические конструкции, а в качестве ребер связи между ними. AST является простой и универсальной структурой данных, позволяющей реализовывать множество анализов. Также, носителем AST может быть любой структурный формат (например S-выражения), что упрощает его построение, сериализацию и анализ. Однако, в контексте мультиязыкового анализа AST является неудобным решением, в первую очередь потому что оно часто вовлекает языкоспецифичные конструкции. К примеру, в некоторых языках отсутствует понятие «оператор» (англ statement), что различает AST таких языков от императивных языков на фундаментальном уровне.

eCST [20] является попыткой расширения концепции AST (а точнее CST) на большее количество языков. Авторы достигают этого путем введения

универсальных узлов, позволяющих отражать общие семантические концепции (итерацию, выбор или переход). К сожалению, такой формат представления информации всё еще неудобен так как языкоспецифичные узлы не могут быть конвертированы в такие универсальные узлы и остаются в дереве в первозданном виде.

CFG это граф, узлы которого представляют собой в общем случае операторы, а ребра поток управления. Такой граф полезен в первую очередь для компиляторных трансформаций и оптимизаций, так как позволяет моделировать исполнение программы и выстраивать на основе этого определенные инварианты. Граф является основой для фреймворков монотонного анализа [3] и активно используется в различных инструментах. Несмотря на достоинства, как и в случае с AST такой граф имеет зависимость от представляемого языка. Также, анализ потока управления сложен в контексте мультиязыкового статического анализа и проще осуществим в рамках динамического анализа.

CDG является формой представления информации о вызовах процедур друг другом в контексте определенного фрагмента кода. Соотвественно, узлы такого графа обычно составляют идентификаторы процедур, а ребра зависимость «вызывает». Стоит сказать, что зависимость «вызывает» обычно является направленной, поэтому граф имеет направление ребер. Такой граф является хорошим кандидатом на универсальное представление в рамках мультиязыкового анализа, что уже было исследовано [16]. Как было сказано в подразделе 2.4, основным недостатком графа зависимостей вызовов является его ориентированность на процедуры – исполняемые фрагменты кода. Это делает невозможным его применение в рамках других языков, в первую очередь форматов данных. Действительно, в современных приложениях семантически важная информация часто содержится в различных конфигурационных файлах, например XML или INI.

Система типов обычно не является ни структурой данных ни моделью представления семантической информации. Её основное назначение заключается именно в проверке программ на определенные классы ошибок, что достигается путем задания определенного набора суждений о сущностях в программе с их дальнейшей автоматической проверкой. Однако, современные ис-

следования в области системы типов показывают, что есть возможность кодирования и эффективной проверки ряда свойств программного кода, часть которых раньше могла быть проверена только интенсивным статическим анализом. Примером является механизм borrow-checking в языке Rust [21], который базируется на системе типов известной как система *владения*. Такой механизм позволяет проверять и устранять класс ошибок связанных с системными ресурсами: использование после освобождения, предотвращенние гонок данных и другие. В контексте мультиязыкового анализа система типов точно имеет первостепенное значение, так как создание корректных анализаторов возможно в том числе за счет использования корректной системы типов. Однако, одной системы типов для совершения анализа недостаточно, так как обычно ограничения на типы сущностей не берут в расчет механизмы областей видимости языков и семантику зависимостей сущностей между собой.

Таблица символов является повсеместной структурой данных (особенно в императивных языках) и используется в первую очередь для представления различной информации об идентификаторах. Так, таблица символов обычно содержит данные о местонахождении определения (или объявления) сущности, её тип, контекст в котором она определена и множество других атрибутов, специфичных для конкретного языка. Структурно таблица символов представляет собой набор пар (идентификатор, атрибуты). Первым и значимым расширением концепции таблицы символов является введение понятия *области видимости*. В языках с лексической областью видимости таблица символов является не таблицей, а стеком таблиц – это необходимо для отражения отношения вложенности между областями видимости. Таблица символов является очень хорошим решением в контексте мультиязыкового анализа, но имеет один серьезный недостаток. Он заключается в отсутствии возможности однозначного кодирования нелексических областей видимости, например в случае присутствия в языке механизма классов или модулей.

Граф свойств может быть определен в терминах теории графов как направленный мультиграф с метками вершин и ребер с собственными ребрами, где ребра имеют собственную идентичность [22]. Такие графы активно используются в графовых базах данных и хорошо подходят для отражения

разноструктурированной онтологической информации о сущностях в области знаний. Например, формат RDF [23] является основным способом представления информации в семантических сетях Интернета. Основанные на нем языки отражения онтологических понятий позволяют создавать машинно-читаемые семантические структуры. Основным недостатком графов свойств в контексте мультиязыкового анализа ПО является отсутствие сколько нибудь универсальной единой онтологической модели. Соотвественно, существующие модели (что рассмотрены в подразделе 2.2) являются специфичными решениями. Также, создание и поддержка таких моделей является довольно трудоемким процессом, так как требует достаточной компетенции специалиста.

Таким образом, на данный момент не существует идеальной структуры представления информации для осуществлений мультиязыкового анализа. Однако, существующие структуры и модели дают представление о том, какой она могла бы быть – идеальная модель являлась бы совмещением концепций системы типов и таблицы символов, с дополнительной возможностью связывания понятий по онтологическим признакам в определенные графы свойств. Таким образом, так или иначе такая модель должна реализовывать формат семантической сети в общем случае.

4.2 Архитектура метода анализа

4.2.1 Парсинг мультиязыковых фрагментов кода

Как было сказано в подразделе 1.2.1, парсинг текста является одной из первых задач, которые необходимо решить при мультязыковом анализе. При наличии единой модели представления информации, однако, такая задача существенно упрощается.

Дело в том, что используя единую модель состоящую из графа можно достичь разделения этапов извлечения информации о графе и анализа самого графа. Похожий подход используется, например, при реализации проверки типов методом Хиндли-Милнера [24]. Суть алгоритма заключается в использовании т.н. *ограничений* которые извлекаются из кода. Такие ограничения являются утверждениями о сущностях и в дальнейшем решаются вместе для

создания специальной подстановки, которая позволит удовлетворить все ограничения. Такой процесс решения называется *унификацией*.

По тому же принципу можно построить модель для мультиязыкового анализа, если в качестве ограничений воспринимать отношения узлов в графе, а в качесте процесса унификации процесс построения самого графа. Такой механизм распределенного анализа позволяет достичь трех целей:

– разделение метода на два этапа (генерация ограничений и их разрешение), что повышает модульность анализатора, его универсальность и открывает возможности для параллелизации,

– объединение анализаторов таким представлением как ограничения позволяет решить проблему парсинга мультиязыковых фрагментов путем реализации специализированных (заточенных под конкретный язык) анализаторов,

– открываются возможности для реализации инкрементального анализа и анализа неполного графа, что критично для таких средств как IDE и линтеры.

Таким образом, такие анализаторы являются *трансляторами* из избранного языка в унифицированное представление, заданное в виде ограничений.

Стоит заметить, что создание раздельных трансляторов под конкретный язык всё еще не решает проблему взаимодействия языков в общем случае, так как часто в коде на избранном языке встречаются фрагменты кода, реализованные на других языках. Обычно, такой код представляется строковым литералом. К примеру, фрагменты такого «встроенного» кода могут включать:

– путь к файлу или ресурсу (в формате URI),

– данные в определенном формате (например XML, JSON...),

– строка запроса к БД (например SQL),

– шаблон документа или веб страницы (в формате HTML),

Такие фрагменты в общем случае сложно обнаружить в исходном коде так как строковые литералы могут быть представлены в любом выражении.

Решением данной проблемы может служить инструмент распознавания определенного языка программирования по входной строке. Таким образом, трансляция определенного языка будет вовлекать распознавание фрагментов языков, описанных через строковые литералы. После распознавания такие фрагменты будут направлены соответствующим трансляторам.

Таким образом, общий алгоритм трансляции выглядит следующим образом:

1) на вход транслятору поступает фрагмент кода на определенном языке,

2) транслятор использует любые подходящие средства для анализа и трансляции данного кода (AST, CFG, абстрактная интерпретация и т.д.),

3) если транслятор обнаруживает строковый литерал он может отправить его на распознавание, которое позволит определить язык избранного фрагмента и в дальнейшем направить этот фрагмент другому транслятору,

4) все трансляторы всех языков выводят ограничения в общий буфер, для последующего разрешения.

Как видно из алгоритма, универсальное представление позволяет реализовывать гибкий анализ кода – сам процесс анализа зависит от конкретного транслятора и может вовлекать любые методы, вплоть до интерпретации или компиляции. Также такая трансляция происходит независимо от остальных трансляторов, что позволяет проводить анализ проектов инкрементально и по требованию.

4.2.2 Унифицированное представление через ограничения

В ходе работы над проектом были реализованы несколько представлений, способных быть достаточно универсальными и при этом распределенными. Были рассмотрены такие структуры как семантическая сеть и дерево модулей, однако такие представления ориентированы на другие сценарии использования, поэтому их корректная адаптация под статический анализ затруднительна. Однако, общей идеей в обоих структурах было создание графа

идентификаторов, каждый из которых обладал бы одной из двух семантик: либо *объявление*, либо *ссылка*.

Объявлением называется семантика создания имени ресурса. Если при этом ресурс привязывается к такому имени, то такое действие называется определением. Ссылкой же называется дальнейшее упоминание такого имени в тексте программы. В контексте мультиязыкового статического анализа данную семантику можно понимать следующим образом.

Объявлением является идентификатор, находящийся в определенном фрагменте кода определенного языка и факт того, является ли идентификатор объявлением определяется транслятором этого языка. Ссылкой является идентификатор находящийся *в другом фрагменте кода, реализованным на другом языке*. Факт того, что идентификатор является ссылкой тоже определяется транслятором избранного языка. Таким образом создается набор идентификаторов, определенные из которых можно сопоставить друг с другом, получив таким образом пары (объявление, ссылка). Эта базовая идея позволяет обнаруживать различные межъязыковые связи различной природы, так как на вид самого идентификатора не налагается никаких ограничений – это может быть URI файла, имя функции, имя модуля или ключ в БД.

Однако, современные проекты в подавляющем большинстве случаев структурированы с учетом изоляции имен. Такой механизм обычно называется механизмом *областей видимости*. Описанная выше схема пренебрегает таким механизмом, что ведет в сущности к лексическому сопоставлению идентификаторов, который не обладает ни должной корректностью, ни полнотой.

Учитывая, что понятие области видимости тесно связано с понятием «модуль» (так как последний часто определяется как именованная область видимости с возможностью импорта), логичным решением является использование универсальной системы модулей для организации идентификаторов в области видимости. Одной из таких систем может являться механизм *графов областей видимости* [25]. Графы областей видимости (далее просто графы областей) являются языконезависимой теорией описания таких механизмов как привязка имен и их разрешение. В оригинальной статье авторами создает-

ся соответствующий граф, узлами которого могут выступать идентификаторы и области видимости, а ребрами – различные отношения между ними. В граф входят такие бинарные отношения как:

- объявлен в,
- имеет ссылку в,
- должен быть разрешен в,
- родительская область видимости,
- достижим,
- видим,

Как видно из списка, данные отношения позволяют задавать богатую семантику разрешения имен, при этом не используя понятия определенного языка программирования. Авторам удается реализовать семантику импорта и включения модулей, а также структуру лексических областей видимости.

Данный фреймворк хорошо подходит для данной задачи, однако в дальнейшем авторами оригинальной статьи был создан другой, более подходящий для статического анализа фреймворк. В статье [26] описывается метод генерации графа областей через использование ограничений, ориентированный на реализацию статических анализаторов. Интерес в первую очередь представляют результаты работы авторов – им удалось существенно упростить фреймворк и включить более универсальные отношения в граф, за счет чего им удается увеличить количество информации, которую можно представить таким графом.

Основным изменением по сравнению с оригинальной работой 2015 года, однако, стало введение типов. В смежной работе [27] авторами рассмотрен механизм графов областей в для которого вводится еще один тип ограничений – ограничения на типы идентификаторов. Такой подход позволяет реализовать *смешанное* разрешение имен, когда процесс разрешения вовлекает как поиск идентификатора в графе, так и вывод типа. Основной причиной введения такого механизма стало использование таких языковых конструкций как записи. В отличие от модулей, записи могут иметь экземпляры, то есть в процесс раз-

решения поля записи вовлекается процесс разрешения типа экземпляра переменной.

Стоит заметить очень важную особенность фреймворка – авторами тщательно доказываются все заявленные свойства через задание формальной семантики. Таким образом обеспечивается корректность алгоритма разрешения имен с учетом изложенного исчисления. Это очень важное свойство, отличающее фреймворк авторов и, соответственно, данный метод, так как корректность позволяет создавать более надежные и точные средства автоматизированного анализа.

В качестве базы модели разрабатываемого метода анализа используется граф областей видимости в редакции статьи [26] , то есть задаваемый через ограничения. В отличие от графа описанного в [27], данный граф имеет более простую организацию и не вовлекает отношение подтипирования. Помимо ограничений, собираемых в ходе построения графа областей и анализа типов идентификаторов решено было ввести дополнительные виды ограничений, позволяющих обеспечивать проверку дальнейшую корректности связей в проекте. Ниже перечислены все задействованные в методе анализа виды ограничений.

1) Объявление/Ссылка,

   а) **Вид:** ограничение графа областей;

   б) **Вовлекаемые узлы:** область видимости в которой упомянут идентификатор, идентификатор;

   в) **Назначение:** связывание идентификатора и области видимости.

2) Прямое ребро,

   а) **Вид:** ограничение графа областей;

   б) **Вовлекаемые узлы:** две области видимости, наименование ребра;

   в) **Назначение:** обозначение прямой связи между областями, моделирование лексических областей видимости.

3) Ассоциация,

а) **Вид:** ограничение графа областей;

б) **Вовлекаемые узлы:** идентификатор или переменная, область видимости;

в) **Назначение:** «именование» определенной области видимости, либо указание того что данный идентификатор имеет привязанную область видимости.

4) Разрешение,

а) **Вид:** ограничение графа областей;

б) **Вовлекаемые узлы:** идентификатор, идентификатор или переменная;

в) **Назначение:** указание того что данный идентификатор должен быть разрешен либо в указанный идентификатор, либо в подстановку переменной.

5) Уникальность,

а) **Вид:** ограничение графа областей;

б) **Вовлекаемые узлы:** коллекция имен;

в) **Назначение:** указание что коллекция имен не содержит дубликатов в рамках данной области видимости.

6) Подмножество,

а) **Вид:** ограничение графа областей;

б) **Вовлекаемые узлы:** две коллекции имен;

в) **Назначение:** указание того, что одна коллекция имен входит в другую коллекцию имен.

7) Аннотация типа,

а) **Вид:** типовое ограничение;

б) **Вовлекаемые узлы:** идентификатор, переменная;

в) **Назначение:** указание типа определенного идентификатора через переменную.

8) Равенство типов,

   а) **Вид:** типовое ограничение;

   б) **Вовлекаемые узлы:** две переменные;

   в) **Назначение:** обозначение равенства друх типовых переменный для их последующей унификации.

9) Обязан разрешиться,

   а) **Вид:** ограничение консистентности;

   б) **Вовлекаемые узлы:** идентификатор, область видимости;

   в) **Назначение:** наложение на ссылку ограничения, обязывающего дальнейшее связывание с объявлением.

10) Необходимый,

    а) **Вид:** ограничение консистентности;

    б) **Вовлекаемые узлы:** идентификатор, область видимости;

    в) **Назначение:** данное объявление должно иметь хотя бы одну ссылку.

11) Эксклюзивный,

    а) **Вид:** ограничение консистентности;

    б) **Вовлекаемые узлы:** идентификатор, область видимости;

    в) **Назначение:** данное объявление должно иметь не более одной ссылки.

12) Знаковый,

    а) **Вид:** ограничение консистентности;

    б) **Вовлекаемые узлы:** идентификатор;

в) **Назначение:** данное объявление может встречаться только в одном экземпляре во всем графе областей видимости.

Дополнительные виды ограничений консистентности («Обязан разрешиться», «Необходимый», «Эксклюзивный», «Знаковый») не затрагивают оригинальный механизм разрешения имен в графе областей и являются простым расширением, позволяющим добавить дополнительную информацию для инструментального средства. Такая информация будет интерпретироваться только им и не будет иметь влияния на процессы построения и решения графа.

4.2.3 Вовлечение операционного окружения

Текущее описание метода предполагает однонаправленное взаимодействие с инструментальным средством – метод, получая данные для анализа, проводит трансляцию и решение ограничений, предоставляя результаты в виде решенного графа и других ограничений.

Однако, для практической реализации метода неизбежно придется ответить на следующие вопросы:

1. Как будет происходить выборка кода для анализа?
2. Каким образом будет учитываться окружение проекта?

Стоит рассмотреть каждый вопрос в отдельности.

Выборка кода для анализа является сложной в общем случае задачей, так как подразумевает промежуточный анализ некоторых файлов проекта и его окружения для получения необходимых путей к файлам кода. Поэтому, извлечение кода в данном методе делегируется конкретному инструментальному средству. Это позволяет достичь следующих целей:

– упрощение метода анализа,

– облегчение поддержки специфических сценариев сборки и кодогенерации,

– повышение универсальности метода анализа.

Данное обособление метода является логичным, так как каждое конкретное инструментальное средство может быть заинтересовано в разных объемах кода и в разных его характеристиках.

Однако, для обеспечения полноценного анализа имеет смысл использовать операционное окружение проекта. Включение в граф областей такой информации может упростить сценарии анализа языков сборки и оболочки командной строки. Для анализа операционного окружения проекта предполагается отображение такого рода информации в виде явных данных в формате ограничений. Таким образом, появляется компонент *провайдер конфигурации*. Он ответственен за отображение данных об операционном окружении в формат ограничений перед этапом трансляции избранных языков. С его помощью могут быть выражены следующие данные об окружении:

– содержимое файловой системы (через граф областей),

– имена и тип различных системных переменных (через ограничения типизации),

Таким образом, возможна реализация анализа «над» разными языками программирования, так как вовлекается анализ внешнего мира, в котором они взаимодействуют. Это позволяет покрыть большую часть сценариев использования метода анализа в контексте различных инструментальных средств.

Стоит также заметить, что информация об источнике определенного ограничения может быть полезна для использования в инструментальных средствах при интерпретации этих решенных ограничений. Поэтому для каждого ограничения также вводится поле об источнике – пара из пути к файлу/фрагменту и названия языка на котором написан фрагмент. Такой подход также позволяет установить *границы* между модулями на различных языках, что может быть полезно для определенных проектов. Такие границы позволяют уменьшить количество ложноположительных результатов, если известно что определенная пара языков никак не должна взаимодействовать и между фрагментами таких языков не может быть междъязыковых связей.

Примером может служить пара из языка для написания документации(например Markdown) и другого языка программирования. Обнаружение

таких межъязыковых связей может быть нежелательно для разработчика инструментального средства и избежать этого можно путем последующей интрепретации решенных ограничений с учетом межъязыковых границ.

4.2.4 Онтология

Онтологией обычно называется набор определенных знаний о предметной области. Однако, в контексте данной работы такой термин будет использоваться для иных целей – им будет обозначаться набор механизмов, позволяющих *обеспечивать консистентность* между различными трансляторами различных языков. Такое название для группы данных механизмов было выбрано в первую очередь для сохранения изначального смысла определения – вне зависимости от того, каким образом онтология может быть реализована (например, как набор механизмов сохранения консистентности), она в любом случае несет в себе важную информацию о предметной области.

Таким образом, с практической точки зрения онтология реализуется через создание двух компонентов:

– типового контекста,

– верификатора структуры подграфа.

Типовый контекст содержит все данные для организации «настраиваемой» системы типов над графом областей. Например, в исходной работе [27] представлена более широкая система типов в сравнении с [26] так как помимо прочего включает механизм подтипирования. Таким образом, в типовый контекст входят обычные типы (англ. ground types), типовые конструкторы (например, функциональная стрелка или тип-произведение) и дополнительные конструкции избранной системы типов (например, отношения подтипирования).

Стоит заметить, что такой гибкий подход к системе типов позволяет создавать довольно мощные механизмы разрешения имен. Например, появляется возможность связывания объявлений и ссылок не только по именам, но и по *значениям*. Этого можно достичь, используя механизм типов пересечений и типов объединений [28]. Расширив систему типов таким образом, можно

будет присвоить идентификатору сильно ограничивающий тип. В качестве примера можно рассмотреть исходный код на языке TypeScript:

```
1    const a: 1 | "a" = 1
2    const b: 1 | "a" = "a"
3    const c: 1 | "a" = true
```

В данном случае определение типа `1 | "a"` указывает, что идентификатор с такой типовой аннотацией может быть привязан либо к значению 1, либо к строке `"a"`. Таким образом, строка 3 считается ошибочной и в контексте разрешения имен это значит, что такое имя никогда не будет связано с переменной иного типа. Такой подход позволяет сильно упростить некоторые сценарии наложения ограничений (например, когда необходимо показать что идентификатор всегда связывает определенное значение и оно известно наперед).

В современной индустрии используется большое количество ООП языков: Java, Kotlin, C#, PHP, TypeScript и многие другие. Хотя концепция *класса* и *объекта* примерно одинакова во всех языках, строгого соответствия между языками всё же нет. К тому же, в разных языках одинаковые синтаксически определения могут очень сильно отличаться по семантике. Например, объект класса в Java всегда будет сравниваться с другим объектом номинально, хотя в TypeScript такой же (синтаксически) объект может участвовать в структурном сравнении.

Этот факт, а еще факт того, что концепция модулей в языках программирования в целом тоже довольно сильно отличается от языка к языку, вынуждает разработчика транслятора определенного языка учитывать очень специфические языковые концепции. Такой подход неизбежно ведет к ограничению количества поддерживаемых в анализе языков и переусложнению модели анализа. Именно поэтому метод мультиязыкового анализа должен поддерживать максимально обобщенное представление – оно позволяет кодировать различную языковую семантику в единой и краткой форме. Но в таком случае появляется проблема согласованности – специфические конструкции одного языка могут быть представлены не так, как конструкции иного языка, но семантически они могут значить одно и то же. Эта проблема *неизбежна*, так как муль-

тиязыковой анализ вовлекает многие независимые трансляторы (на каждый язык), каждый из которых может быть разработан создан любым разработчиком.

Для решения данной проблемы предполагается использование механизма *структурной верификации*. Ограничения, помимо прочего, задают набор *известных фактов*. Эти факты являются начальными данными для алгоритма разрешения имен. Именно такие факты должны быть согласованны между трансляторами для того чтобы семантика классов и модулей различных языков была отражена единообразно, так как это позволяет производить верное разрешение имен в дальнейшем. Для обеспечения такой согласованности можно осуществлять верификацию структуры ограничений на этапе их генерации.

В качестве примера работы такого механизма можно рассмотреть следующий код на C# взятый из примера проекта на платформе dotnet [29]:

```
[ApiExplorerSettings(IgnoreApi = true)]
[Authorize]
[Route("controller/action")]
public class OrderController : Controller
{
    private readonly IMediator _mediator;

    public OrderController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet]
    public async Task<IActionResult> MyOrders()
    {
        Guard.Against.Null(User?.Identity?.Name,
        ↪  nameof(User.Identity.Name));
```

```
17              var viewModel = await _mediator.Send(new
                ↪  GetMyOrders(User.Identity.Name));
18
19              return View(viewModel);
20          }
21
22          [HttpGet("{orderId}")]
23          public async Task<IActionResult> Detail(int orderId)
24          {
25              Guard.Against.Null(User?.Identity?.Name,
                ↪  nameof(User.Identity.Name));
26              var viewModel = await _mediator.Send(new
                ↪  GetOrderDetails(User.Identity.Name, orderId));
27
28              if (viewModel == null)
29              {
30                  return BadRequest("No such order found for this
                    ↪  user.");
31              }
32
33              return View(viewModel);
34          }
35      }
```

Вышеприведенный класс имеет большое количество различных аннотаций (как в определении класса, так и в определении методов). Также, он наследует от класса `Controller`. Однако, на уровне межъязыковых связей, можно сказать что этот класс является одним из эндпоинтов веб-сервера, который выполняет различные действия по различным HTTP запросам (в данном примере это GET запросы). Разработчику моноязыкового транслятора хотелось бы каким-либо образом извлечь эту информацию в формальной форме и структура самого класса в контексте межъязыкового анализа для него неинтересна. Для этого хорошо подходит структура задаваемая ограничениями, но они являются слишком абстрактным представлением и не поддерживают

идею «абстрактного веб-сервера» напрямую. Тогда, может быть разработан набор правил верификации структуры веб-сервера (приведенных неформально):

1) имеет ли модуль название `WebServer`,

2) имеет ли он поле `GET` которое тоже является модулем,

3) имеет ли такой модуль поле `application/json`,

4) имеет ли такое поле схему json (иерархическую структуру, имеющую типы JSON),

5) имеет ли модуль `WebServer` поле `POST`,

6) и другие возможные проверки...

Таким образом, извлекаемые транслятором ограничения могут быть проверены на соответствие такому «архетипу» веб-сервера. Если ограничения не имеют такую структуру, то транслятор потерял согласованность с другими трансляторами и должен быть доработан.

В итоге, механизм верификации структуры позволяет достичь следующих целей:

1) упрощение поддержки мультиязыкового транслятора и согласованности моноязыковых трансляторов,

2) поддержка более сложных семантических конструкций на основе классов и модулей (веб-сервер, библиотека, графический ресурс и многие другие),

3) облегчение модели анализа с расширенной поддержкой языкоспецифичных конструкций (если они выражаемы через механизм классов или модулей).

Подводя итоги подраздела 4.2, можно составить общую диаграмму потока данных между вовлеченными модулями. Такая диаграмма представлена на рисунке 4.1.

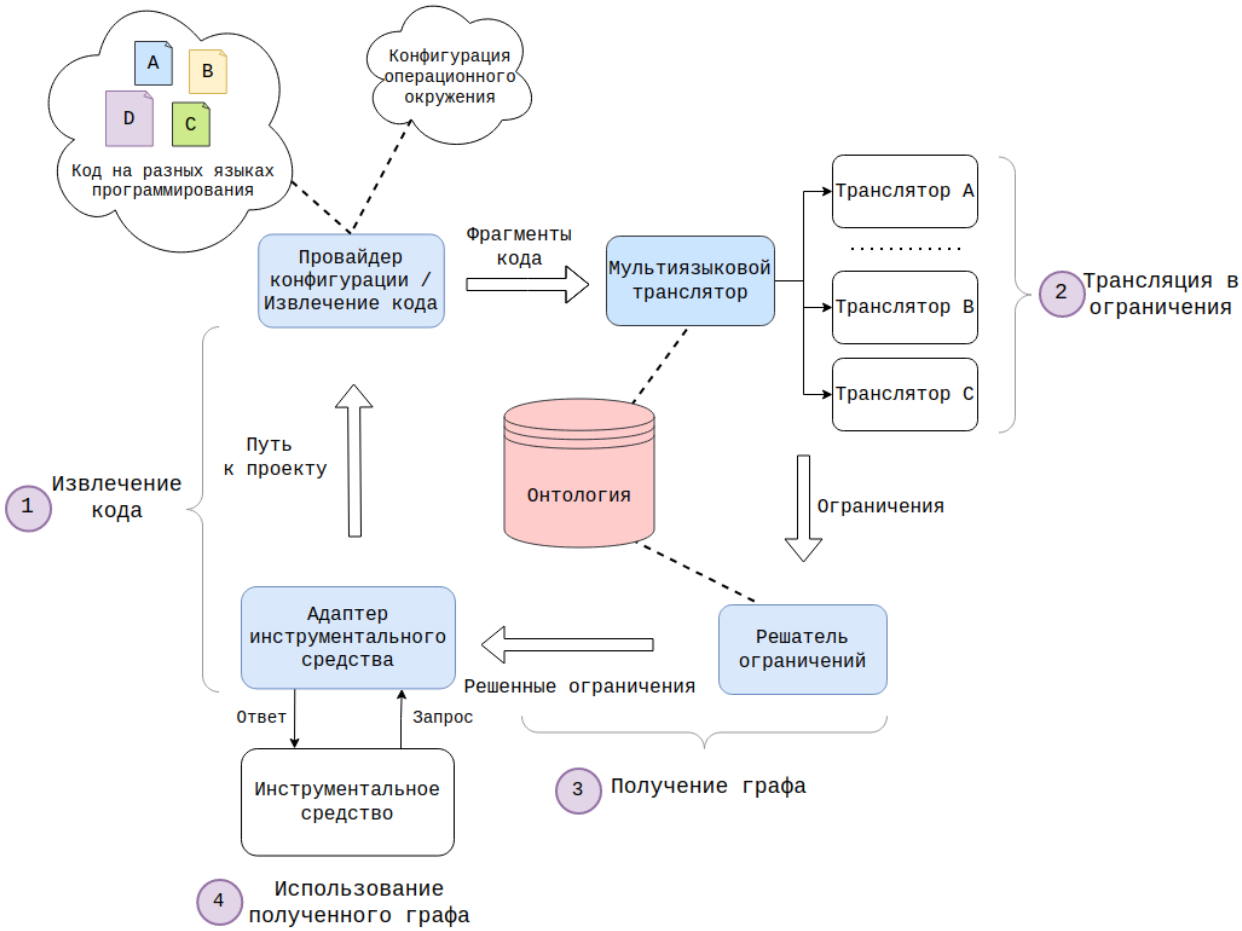

Рисунок 4.1 — Структура метода в виде диаграммы потока данных

4.3 Исследование средств реализации

Учитывая поставленные задачи логичным решением было выбрать популярный формат данных, который хорошо поддерживается современными языками. В качестве основного формата передачи (т.е. носителем) ограничений решено было выбрать JSON. Он является универсальным форматом структурированных данных, используемым в большом количестве различных технологий и инструментов. В данном случае, формат был выбран по следующим причинам:

– простота и универсальность,

– широкая поддержка в различных языках и средствах,

– относительная компактность,

– частая интеграция формата в существующих инструментальных средствах.

В целом язык не является основополагающим при разработке анализатора, так как не требуется никаких специфических технологий для его реализации. При разработке в рамках данной работы было решено выбрать язык Golang. Это прикладной императивный язык, поддерживающий компиляцию в машинный код и небольшой рантайм, при этом сопровождая большинство удобных функциональных возможностей для упрощения процесса разработки. Из плюсов языка в контексте данного проекта можно отметить следующие особенности:

– простота синтаксиса и семантики,

– наличие статической типизации,

– относительно легкая работа с динамическими (как JSON) данными,

– большое количество различных библиотек и очень высокое качество средств поддержки разработчика,

– быстрая компиляция и высокое быстродействие программ.

Минусами языка выступают: ограниченная система типов, слабая поддержка нетривиальных подходов к решению задач, сильный расчет на динамические проверки во время исполнения. Однако, для данного проекта эти минусы несущественны и не помешают реализации основных модулей анализатора.

5 Описание основных сценариев использования анализатора в сочетании с инструментальными средствами программирования

5.1 Интеграция в различные инструментальные средства

Анализатор, поставляющий информацию, описанную в 4.2.2 способен поддерживать большое количество разнообразных инструментальных средств. Соответствующие возможности анализатора отражены в таблице 5.1.

Таблица 5.1 — Поддержка анализатором различных сценариев использования инструментальных средств

Сценарий использования	Поддержка	Необходимая информация
Обнаружение багов	Частично	Идентификаторы; типы идентификаторов; значения идентификаторов; языкоспецифичная информация
Обнаружение уязвимостей	Отсутствует	Полная информация о системном окружении; знание уязвимостей различных библиотек и фреймворков
Соблюдение стиля кода и стандартов	Частично	Полный доступ к тексту и языкоспецифичному AST
Оценка сложности кода	Частично	Взаимосвязи различных идентификаторов, знание языкоспецифичных конструкций
Анализ зависимостей	Полностью	Идентификаторы и связи объявление-ссылка

Стоит подробнее описать каждый из сценариев для выявления возможностей предлагаемого метода.

Обнаружение багов может производиться на разных уровнях. Багом в данном случае может считаться непреднамеренная функциональность (или её отсутствие), не отвечающая ожиданиям разработчика. Соотвественно, в категорию багов, выявляемых анализатором могут потенциально входить: неразрешенная ссылка, больше одной ссылки на определенный идентификатор, отсутствие ссылок на определенный идентификатор. Баги иного рода находятся

за рамками метода анализа, в первую очередь из-за отсутствия возможности отслеживать значения идентификаторов, т.е. отсутствие концепции *переменной* или *привязки*.

Обнаружение уязвимостей является довольно специфичным и сложным процессом, поэтому количество сопровождаемой методом информации чаще всего будет недостаточным. Это связано в первую очередь с отсутствием языкозависимой информации о типах или функциях, которые могут содержать уязвимость. Соответственно, для реализации средств обнаружения уязвимостей в ПО метод не подходит.

Соблюдение стиля кода и стандартов часто вовлекает идентификаторы. В контексте анализа стиля назаваний переменных в проекте метода может быть достаточно. Однако, метод не сопровождает никакой информации об исходном AST исходного языка, поэтому *применение* каких-либо рекомендаций по стилю может быть довольно сложным.

Оценка сложности кода вовлекает в себя использование различных метрик, например цикломатическая сложность функции, связность классов или высота дерева наследования. Метод позволяет оценивать такие метрики как: количество переменных в одной области видимости, связность на уровне модулей или файлов, количество использований какого-либо определения. Оценка иных метрик сложна в первую очередь по причине отсутствия языкоспецифичной информации.

Анализ зависимостей является важным процессом для обеспечения поддержки ПО. Рассматриваемый метод анализа является в сущности способом построить граф объявлений и ссылок (или def-use граф), поэтому для данного сценария использования он подходит очень хорошо. В сущности, метод позволяет обеспечивать *межъязыковой* анализ зависимостей, что может быть полезно в инструментах линтинга или сбора статистики.

Таким образом, хоть поддержка различных инструментальных средств в контексте мультязыкового анализа и ограничена, применимость метода всё еще довольно высока. Особенно это касается анализа зависимостей в проекте. Основной особенностью метода является стирание языкоспецифичной информации, что сильно уменьшает количество возможных сценариев исполь-

зования, но в контексте некоторых сценариев использования это не требуется. Одним из таких сценариев является поддержка разработчика в процессе кодирования.

5.2 Реализация LSP как одного из сценариев интеграции метода

В данной работе в качестве основного примера интеграции метода анализа было решено реализовать анализатор, совместимый с LSP [30]. LSP является протоколом, обобщающим языковые концепции для получения общих методов взаимодействия с программой в контексте инструментальных средств разработки (в первую очередь IDE).

В возможные сценарии LSP (обобщенно) входит:

1) найти объявление или определение идентификатора (типа или терма),
2) определить иерархию вызовов,
3) найти все ссылки на идентификатор,
4) построить иерархию вызовов/наследования,
5) подсветить идентификатор под курсором,
6) автодополнение идентификатора под курсором,
7) дать подсказку по проблеме под курсором (Inlay hint),
8) остальные функции связанные с редактированием (подсветка, форматирование).

В таблице 5.2 представленно разделение приведенных сценариев на основе необходимой для анализа информации.

Таблица 5.2 — Разделение сценариев по требуемой информации

Сценарий	Назначение	Необходимая информация
Поиск объявления или определения	Навигация по коду	Информация о областях видимости; информация о типах
Определение иерархии вызовов	Навигация по коду	Информация о областях видимости; информация о типах
Поиск ссылок на идентификатор	Навигация, редактирование	Информация о областях видимости; информация о типах
Подсветка символа	Редактирование	Информация о областях видимости
Автодополнение символа	Редактирование	Информация о областях видимости; информация о типах
Подсказка по проблеме	Редактирование, корректность	Специфическая для языка информация

Как видно из таблицы, даже при таком грубом разделении сценариев, описываемый метод способен поддержать подавляющее большинство сценариев LSP. В теории, возможно создание *универсального LSP анализатора*, позволяющего вовлекать не только языкоспецифичный анализ, но и межъязыковой. Такой анализатор будет иметь возможность сопровождать приведенные сценарии использования для любого количества языков и их комбинации, что может позволить сильно увеличить продуктивность разработчиков в отношении разработки и поддержки ПО.

6 Программная реализация анализатора и инфраструктуры

Данная реализация анализатора является первичной и её главная задача это иллюстрация работоспособности метода анализа. Поэтому данный анализатор включает дополнительную инфраструктуру, не рассматриваемую в методе. Такие компоненты анализатора будут отмечены соответствующим примечанием в тексте.

6.1 Общая архитектура

6.1.1 Составляющие анализатора

Исходя из раздела 4.2 анализатор, реализующий описываемый метод мультиязыкового анализа будет включать в себя следующие компоненты:

1) провайдер конфигурации и кода,

2) мультиязыковой транслятор,

3) решатель ограничений,

4) адаптер для конкретного инструментального средства (в данном случае LSP сервера).

Более подробный состав анализатора представлен на диаграмме компонентов на рисунке 6.1.

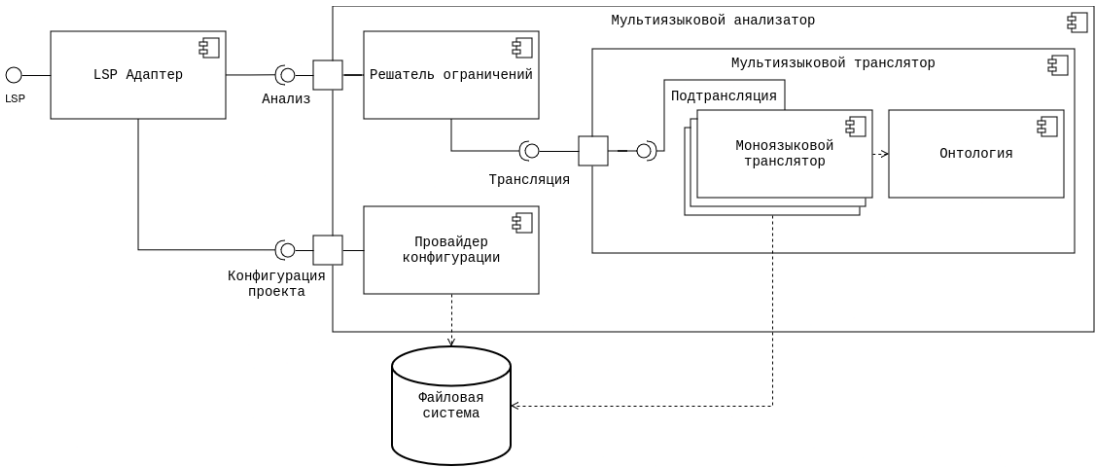

Рисунок 6.1 — Диаграмма компонентов анализатора

Также, порядок запросов к анализатору и вызовы соответствующих процедур приведены на диаграмме деятельности на рисунке 6.2.

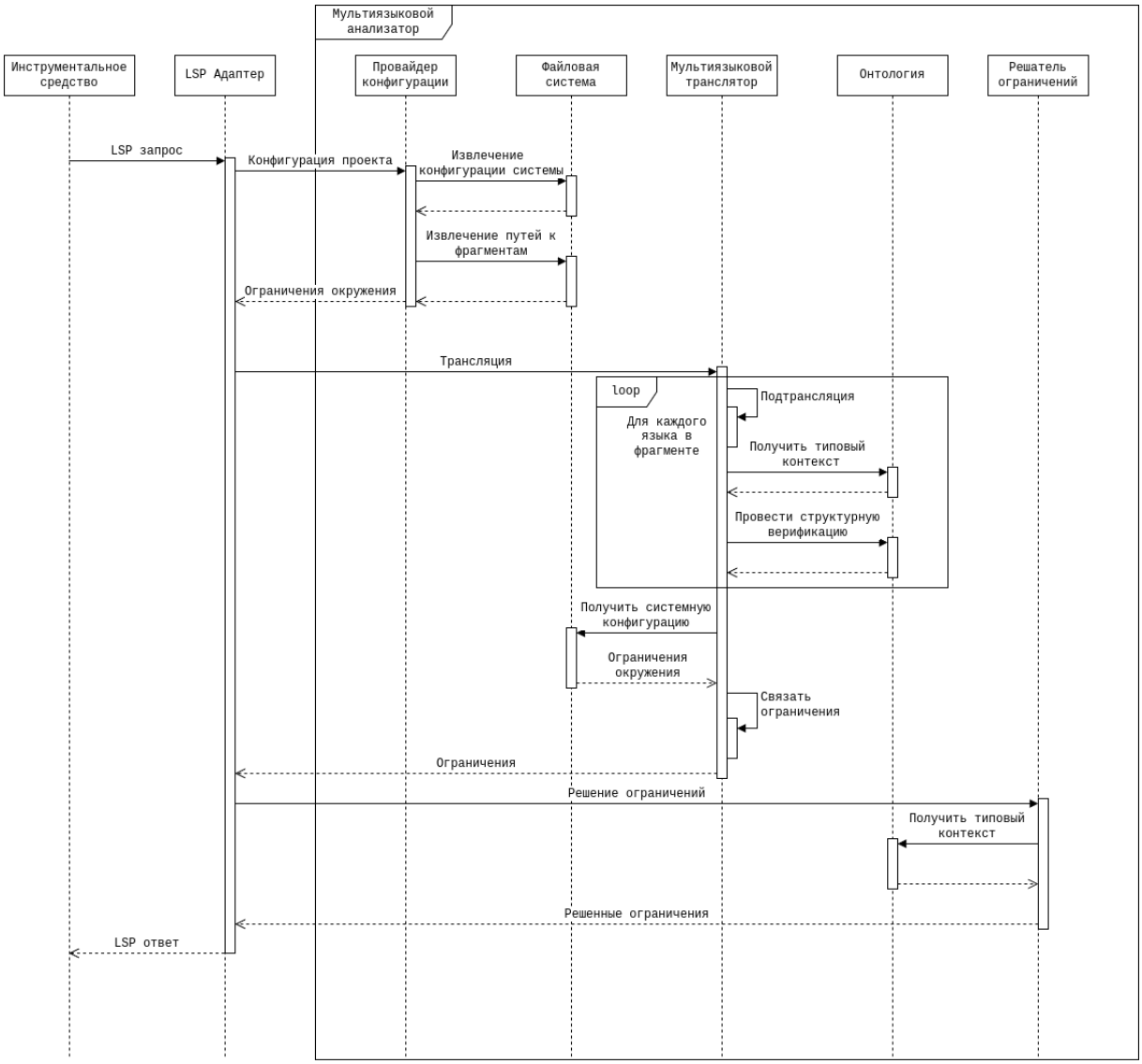

Рисунок 6.2 — Диаграмма деятельности анализатора

Стоит сразу уточнить несколько деталей относительно представленных диаграмм:

1) диаграммы представлены на русском языке, на довольно высоком уровне – это сделано с целью упрощения понимания компонентов анализатора,

2) на диаграммах не представленно кеширование результатов анализа,

3) на диаграмме деятельности представлен самый благоприятный сценарий, без возникновения ошибок – в случае ошибок используется стандартный механизм возврата ошибок JSON-RPC [31].

6.1.2 Протоколы взаимодействия

Приведенные на рисунке 6.1 интерфейсы компонентов являются протоколами взаимодействия. Каждый компонент в сущности представляет собой отдельное приложение, поэтому такие протоколы описывают в первую очередь межпроцессное взаимодействие. Такой подход к структуре анализатора был сделан по следующим причинам:

1) увеличение гибкости анализатора для расширения (в первую очередь в части количества моноязыковых трансляторов),

2) поддержка использования многозадачности системы для одновременного анализа множества фрагментов,

3) поддержка различных языков и технологий реализации компонентов.

Стоит обратить внимание на заключительный пункт – в отличие от некоторых мультиязыковых анализаторов, рассмотренных в разделе 2, подход с использованием обобщенной модели и общего протокола позволяет абстрагироваться от деталей реализации конкретного транслятора. Это может быть полезно к примеру в том случае, если для анализируемого языка уже существует инфраструктура для анализа и она на другом языке (чаще всего на нем самом).

К примеру, можно привести инфраструктуру LLVM и в частности Clang для анализа C++ – в проекте существует множество различных инструментов анализа C++, от построения AST и CFG до проведения различных оптимизаций. Разработчик моноязыкового транслятора может использовать эту инфраструктуру свободно, в его обязанность входит лишь сопровождение результатов анализа в опредленной протоколом форме. Этот факт, в свою очередь, в целом позволяет использовать различные техники статического анализа (меж-

процедурный анализ, построение DFG, абстрактная интерпретация) что повышает гибкость моноязыковых трансляторов без изменения всего анализатора.

Каждый из используемых протоколов базируется на механизме JSON-RPC. Это позволяет учесть стандартные сценарии возникновения ошибок при передаче данных или во время анализа, а также устанавливает совместимость с популярными инструментами (в первую очередь LSP). Описание всех протоколов в качестве общего протокола приведено в приложении А. В качестве языка описания используется TypeScript так как его типовые определения имеют достаточно высокую выразительность и его типы напрямую могут быть отражены в типы JSON.

6.1.3 Реализация онтологии

Как сказано в подразделе 4.2.4, онтология в данной работе является собирательным названием механизмов поддержания консистентности.

Программная реализация таких механизмов была сделана при помощи использования отдельной виртуальной машины в мультиязыковом трансляторе [32]. Такая машина является встраиваемым в приложение решением и направлена на исполнение кода JavaScript.

Подход по встраиванию виртуальной машины был избран по следующим причинам:

– скрипты JavaScript являются отдельной точкой расширения мультиязыкового анализатора и не затрагивают его компоненты, что повышает модульность,

– JavaScript является самым удобным средством для работы с JSON, так как является его суперсетом,

– использование полноценного языка программирования в отличие от языка описания данных позволяет без излишнего усложнения расширять функциональность онтологии.

6.2 Провайдер конфигурации и кода

В задачи провайдера конфигурации входит:

– создание «слепка» файловой системы проекта в виде списка дерева каталогов и файлов,

– создание базы данных окружения системы (переменные окружения, версии системных утилит и библиотек).

Ключевой особенностью провайдера является формат результирующих данных – после исполнения описанных задач, провайдер должен предоставить информацию в формате ограничений. Таким образом, информация об операционном окружении естественным образом включается в общую структуру графа областей. Это позволяет реализовывать разрешение имен не только на уровне языков, но и на уровне окружения, что может быть полезно в случае языков командной строки или языков сборки – в них идентификаторы команд часто используются из содержимого файловой системы.

Вопрос извлечения кода может быть рассмотрен в виде отдельного компонента анализатора (и ранее он так и рассматривался), но для упрощения архитектуры решено было совместить его с провайдером конфигурации. Задачей компонента извлечения является получение путей к файлам проекта, которые нужно проанализировать. В простейшем случае этого можно достичь путем рекурсивного обхода директорий с выборкой файлов с определенным расширением. Именно такой подход был избран для текущей реализации анализатора. Однако, выборка файлов может быть достаточно сложным процессом по ряду причин, описанных в подразделе 1.2.3.

6.3 Мультиязыковой транслятор

В общем случае, использование отдельного транслятора, реализованного для конкретного языка отдельно от остальных трансляторов является неразумным решением, так как происходит потеря информации о переплетении фрагментов кода на разных языках в рамках одного большого фрагмента. Поэтому, логичным выглядит реализация «мультиязыкового» транслятора, кото-

рый будет использовать различные монотрансляторы. Но чтобы такой транслятор работал корректно, нужен механизм распознавания определенного языка программирования по входной строке. Таким образом, трансляция определенного языка будет вовлекать распознавание фрагментов языков, описанных через различные фрагменты кода. После распознавания такие фрагменты будут направлены соответствующим трансляторам.

Таким образом, общий алгоритм трансляции выглядит следующим образом:

1) на вход мультиязыковому транслятору поступает фрагмент кода,

2) транслятор использует инструмент распознавания для подбора соответствующего монотранслятора и отправляет ему этот фрагмент,

3) на вход транслятору поступает фрагмент кода на определенном языке,

4) транслятор использует любые подходящие средства для анализа и трансляции данного кода (AST, CFG, абстрактная интерпретация и т.д.),

5) если транслятор обнаруживает строковый литерал он может отправить его на распознавание мультиязыковому транслятору, который позволит определить язык избранного фрагмента и в дальнейшем направит этот фрагмент другому транслятору,

6) все трансляторы всех языков выводят ограничения в общий буфер мультиязыкового транслятора, для последующего разрешения.

В качестве инструмента распознавания можно использовать инструменты машинного обучения для классификации. К примеру, на данный момент существуют довольно мощные инструменты по распознаванию кода основанные на нейронных сетях [33]. Однако, в данном анализаторе был использован более простой подход: путем отправки фрагмента всем имеющимся трансляторам можно сразу получить все необходимые данные – в таком случае сообщение об ошибке придет от всех трансляторов, кроме одного. Результаты именно этого транслятора являются верными.

Результаты трансляции различных монотрансляторов объединяются в контексте системного окружения. Объединение не вовлекает решения ограни-

чений, происходит простое связывание узлов графа областей и переменных типовых ограничений. Следует заметить, что в текущей версии анализатора не реализован инкрементальный подход к извлечению конфигурации и трансляции, однако концептуальных препятствий к этому нет и такие процессы могут быть организованы инкрементально для повышения быстродействия.

6.4 Решатель ограничений

Алгоритм решения ограничений подробно описан в [26] и в данной работе не изменяется. Дополнительные ограничения, вводимые в данной работе служат лишь дополнительным источником информации для инструментального средства и не задействованы в алгоритме разрешения имен.

Оригинальная статья описывает сценарий, при котором одной ссылке может сопоставиться несколько объявлений. В таком случае описывается процесс деления вычисления на ветви, по ветви на каждое возможное определение. Так как такая недетерминированность сильно замедляет вычисление, в данном анализаторе было принято решение принебречь полнотой анализа и брать первое совпадающее определение.

В результате работы алгоритма решения ограничений формируется типовый контекст ψ и известный граф областей G. С точки зрения решателя это означает, что можно сформировать *решенные ограничения* – ограничения, в которых не присутствуют неизвестные переменные. Такие ограничения можно называть фактами о проекте.

Стоит заметить, что в случае неполного решения ограничений (алгоритм заходит в тупик при наличии необработанных ограничений), результат решения всё еще актуален и его можно использовать для дальнейшей отправки в инструментальное средство. К сожалению, авторами статьи не было формально доказано соответствие нерешенных ограничений действительно присутствующим ошибкам в коде, поэтому часть нерешенных ограничений не может быть использована в качестве источника информации об ошибках.

Как и в оригинальной статье, инкрементальная реализация алгоритма не была рассмотрена по причине высокой сложности и некоторых техниче-

ских вопросов в оригинальном описании алгоритма. Однако, как и в случае с трансляцией, концептуальных препятствий к созданию инкрементального алгоритма решения ограничений нет.

6.5 Адаптер инструментального средства

Адаптер представляет собой сервер LSP, умеющий исполнять определенные сценарии LSP [34]. Сервер написан на TypeScript в первую очередь по причине присутствия нативного API для работы с методами LSP и соответствующими структурами данных.

Конфигурация мультиязыкового анализатора создана с учетом stateless реализации, т.е. сам анализатор не хранит состояние вычислений после определенных этапов. Вместо этого, было решено вынести этот слой управления состоянием в инструментальное средство, в данном случае сервер LSP. Такой подход позволяет упростить анализатор и сделать его более гибким.

Еще одной важной особенностью является принцип работы сервера – в отличие от многих существующих серверов LSP для определенных языков, такой сервер не взаимодействует с файлами проекта напрямую. Единственная информация о проекте которая ему известна это решенные ограничения, т.е. факты об областях, идентификаторах и типах. С каждым ограничением связывается определенное местоположение – откуда такое ограничение было получено. Такая конструкция подзволяет облеспечить сервер всей необходимой информацией по межъязыковым зависимостям.

В связи с этим появляется несколько особенностей, которые делают применение мультиязыкового анализатора в сервере LSP специфическим:

– инкрементальный анализ является нелинейным,

– сообщения об ошибках не могут связаны с процессом решения ограничений.

В оригинальной статье [26] при разработке алгоритма разрешения имен, авторами учитывается очень важное свойство графов областей – нелинейность при разрешении имен. Используя пример авторов: допустим, существу-

ет граф G в котором ссылка r_i в области S разрешается в объявление d_i в родительской области S'. Тогда, в большем графе G', в котором объявление d_j находится уже в самой области S, ссылка r_i неизбежно будет разрешена именно в это объявление, а старое объявление из S' будет затенено. Таким образом, при увеличении графа, предыдущие результаты решения графа могут оказаться неверными, что может обязывать в необходимости запуска решателя еще один раз.

Как было сказано в 6.4, нерешенные ограничения не могут быть использованы напрямую как источник ошибок, по крайней мере это не было доказано. Именно поэтому список ограничений был дополнен *ограничениями консистентности*, которые позволяют инструментальному средству определить ошибочную ситуацию и составить соответствующий отчет.

Стоит заметить, что гибкость мультиязыкового анализатора позволяет учесть обе вышеперечисленные особенности, если имеется возможность вовлечения эвристик о предметной области анализа – в первую очередь о языках и их возможных связях. Например, запуск решателя на всех ограничениях после изменения или дополнения некоторых файлов не всегда необходим, так как такие файлы могут не изменять межъязыковые связи вовсе. Добавление таких эвристик часто присутствует в таких инструментах как LSP сервер и мультиязыковой анализ в данном случае не становится исключением.

Для интеграции полученного сервера в инструментальное средство решено было использовать Visual Studio Code и соответствующий API. Для данной задачи было разработано небольшое расширение и предоставлены команды для работы с LSP сервером через UI. Такие команды описаны в таблице 6.1. Стоит заметить, что локальная конфигурация команд и кеширование не описываются, так как это аспект реализации конкретного расширения для IDE.

Таблица 6.1 — Команды для работы с мультиязыковым сервером LSP из Visual Studio Code

Команда	Используемая информация	Описание команды
`crossy/configure`	Путь к проекту	Сбор информации об операционном окружении проекта
`crossy/translate`	Путь к онтологии	Трансляция кода проекта с сохранением полученных ограничений
`crossy/solve`	Путь к онтологии	Решение ограничений и сохранение полученной подстановки
`crossy/analyze`	Путь к онтологии	Выполнение трансляции и решения ограничений с применением полученной подстановки
`crossy/reset`	–	Удаление сохраненного кеша

7 Оценка характеристик метода на избранных тестовых проектах

Для подтверждения характеристик метода анализа было создано три тестовых проекта. Проекты были созданы исходя из следующих критериев:

– использование нескольких популярных языков в одном проекте,

– каждый проект решает задачу в определенной уникальной предметной области,

– используемые языки должны представлять различные парадигмы программирования,

– каждый из проектов должен вовлекать различные сценарии использования LSP.

В перечисленных примерах используются рисунки для визуализации графа областей, а именно рисунки 7.1, 7.4 и 7.7. Для данных рисунков принято следующее соглашение: розовыми линиями отмечается отношение «определяет», синими – «ссылается», красными – «имеет общую область видимости», зелеными – «проистекает» или «следует» (в случае анализа DFG). Некоторые идентификаторы имеют соответствующие плашки с присвоенными типами.

Все полученные ограничения находятся в открытом репозитории проекта в виде файлов кода на языке TypeScript. [1]

7.1 Пример 1 – C# и Visual Basic

7.1.1 Общее описание

Приложение, реализованное на C# использует слой взаимодействия на уровне байткода IL и имеет в зависимостях библиотеку, реализованную на Visual Basic. Весь проект объединен через Solution файл. Такой сценарий часто встречается в legacy приложениях, при переходе кодовой базы с Visual Basic на C#.

[1] https://github.com/UberDever/crosslingual-analysis/tree/master/projects/crossy/lsp-adapter/examples

Основной проблемой в проекте является отсутствие информации о том, где конкретно объявлен класс на другом языке и какие поля он содержит.

Вовлекаемые языки: C#, Visual Basic, XML (`.sln`, `.vbproj` и `.csproj` файлы). Предположительные предметные области: Энтерпрайз приложения, частные коммерческие решения. Рассмотренные парадигмы языков: ООП, структурированные данные.

7.1.2 Окружение

При извлечении кода могут быть использованы файлы проектов `.vbproj` и `.csproj`, в дальнейшей достаточно построение «слепка» файловой структуры проекта.

7.1.3 Трансляция и решение

Трансляция может вовлекать в себя стандартный reaching definition анализ [3] и прямое отображение статической структуры классов и объектов на ограничения графов областей. При отображении не были использованы структурные предикаты, так как структура модулей и классов C# не имеет определенной семантики в отношении типов и возможной вложенности.

Полученный граф областей вкупе с типовыми ограничениями представлен на рисунке 7.1.

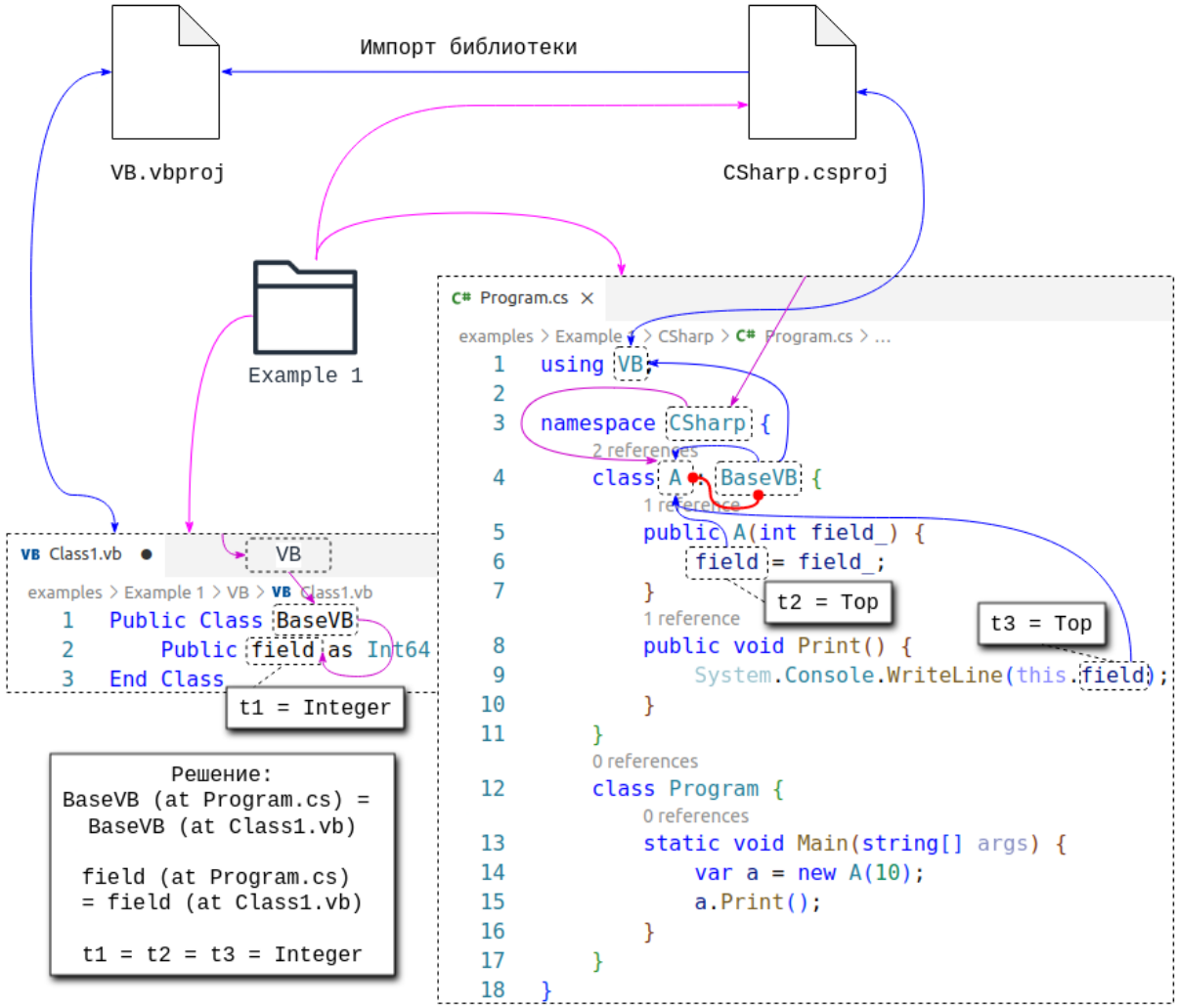

Рисунок 7.1 — Визуализация графа областей и типовых ограничений для примера 1

7.1.4 Онтология

Типы, добавленные в онтологию:

— Integer – целочисленное число неспецифицированной точности,

— Top – супертип всех возможных типов, также является маркером отсутствия информации об идентификаторе.

7.1.5 Сценарии LSP

Для данного примера было решено реализовать два сценария использования, характерных для ООП языков: «Type hierarchy» (получение иерархии типов) и «Go to definition/declaration» (перейти к объявлению/использованию). Предполагаемые примеры внешнего вида реализации таких сценариев показаны на рисунке 7.2 и рисунке 7.3 соответственно.

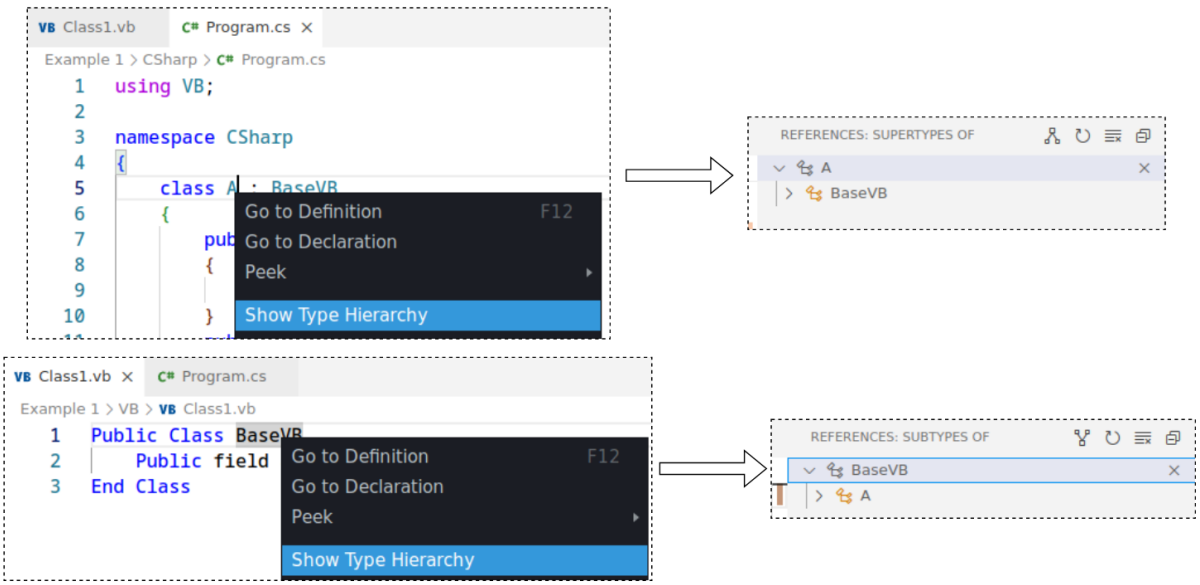

Рисунок 7.2 — Сценарий LSP «Type hierarchy»

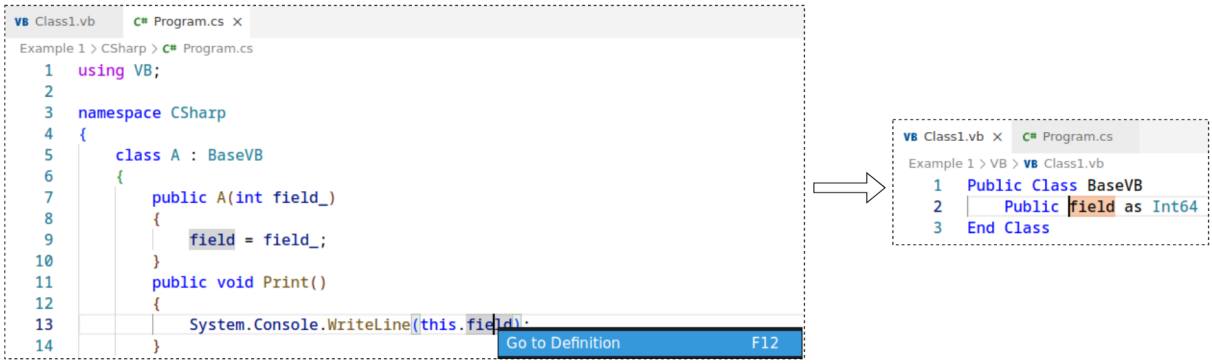

Рисунок 7.3 — Сценарий LSP «Go to definition/declaration»

7.2 Пример 2 – C++, Python 3 и Shell

7.2.1 Общее описание

Проектом служит библиотека, реализованная на C++ и предназначенная для задач научных вычислений. Для этой библиотеки разработана небольшая инфраструктура в виде скриптов командной оболочки для сборки и запуска тестирования библиотеки. Также, для библиотеки реализован тестовый фреймворк в виде отдельного скрипта на Python 3.

Основной проблемой в проекте является отсутствие информации о сигнатурах функций, экспортируемых такой библиотекой.

Вовлекаемые языки: C++, Python 3, Shell. Предположительные предметные области: Научные вычисления, DevOps и тестирование, машинное обучение. Рассмотренные парадигмы языков: Динамические языки, процедурные языки, языки командной оболочки.

7.2.2 Окружение

В данном проекте используются стандартные BusyBox утилиты (`rm`, `mkdir`, `mv` и т.д.), а также компилятор `c++`. Информация об этих инструментах может быть использована при анализе зависимостей с соответствующей выдачей отчета об ошибках, однако в данном примере такая информация не анализируется. Анализатором Shell используется знание о принципе работы флагов компилятора `c++`, что также частично является информацией из окружения (так как флаги могут зависить от, например, версии компилятора).

7.2.3 Трансляция и решение

В данном примере использован интенсивный внутренний анализ графа потока данных, для выявления точек (а точнее, идентификаторов), которые потенциально взаимодействуют на межъязыковом уровне. Так, например, для кода Python 3 была выявлена связь идентификатора `lib_dir/lib.so` и всех функций, которые брались из соответствующей библиотеки. Также, при ана-

лизе Shell из информации о принципе работы компилятора c++ удается вывести, что файлы `lib.cpp` и `lib.so` имеют общую область видимости в отношении идентификаторов, которые ими экспортируются.

Полученный граф областей вкупе с типовыми ограничениями представлен на рисунке 7.4.

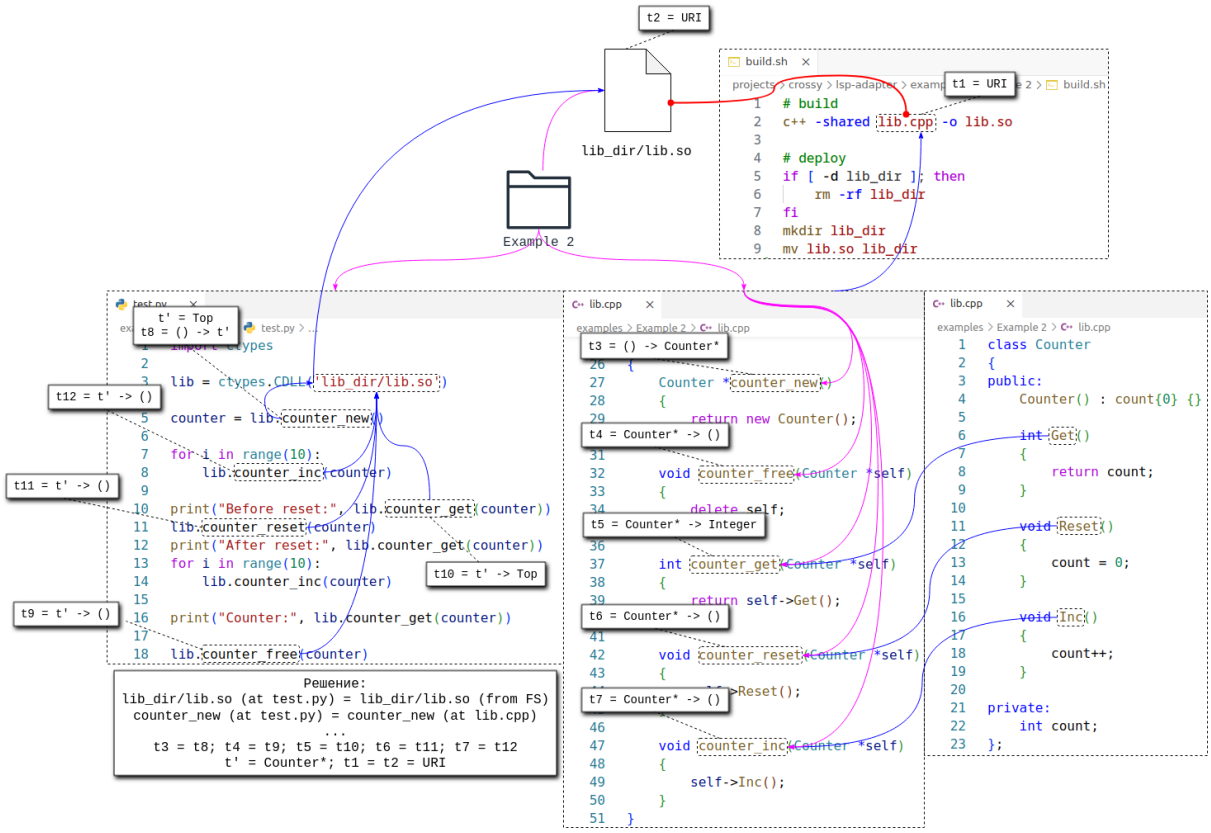

Рисунок 7.4 — Визуализация графа областей и типовых ограничений для примера 2

7.2.4 Онтология

В онтологию был добавлен структурный предикат «библиотека»: в сущности, библиотекой может считаться идентификатор, имеющий тип `URI` и имеющий связанную область видимости. В данной области видимости могут быть объявлены идентификаторы имеющие только функциональный тип и не имеющие вложенных областей видимости. Такой предикат позволяет моделировать простейшие библиотеки на языке C, с отсутствием внешних глобальных переменных, что может быть характерно для определенных проектов.

7.2.5 Сценарии LSP

Так как для данного проекта важна информация о сигнатурах функций библиотеки, решено было реализовать два поддерживающих разработчика сценария: «Completion» (автодополнение) и «Signature help» (подсказка сигнатуры). Предполагаемые примеры внешнего вида реализации таких сценариев показаны на рисунке 7.5 и рисунке 7.6 соответственно.

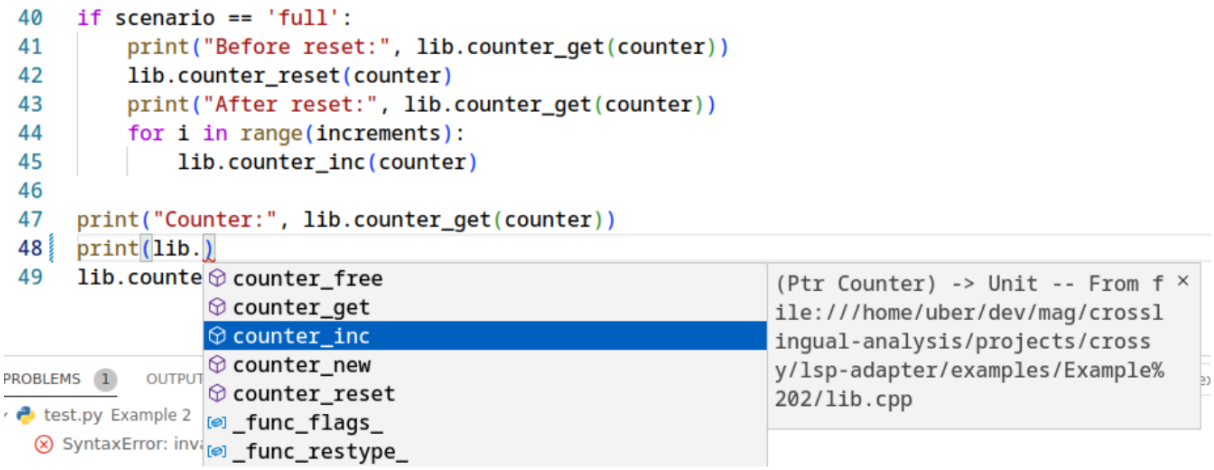

Рисунок 7.5 — Сценарий LSP «Completion»

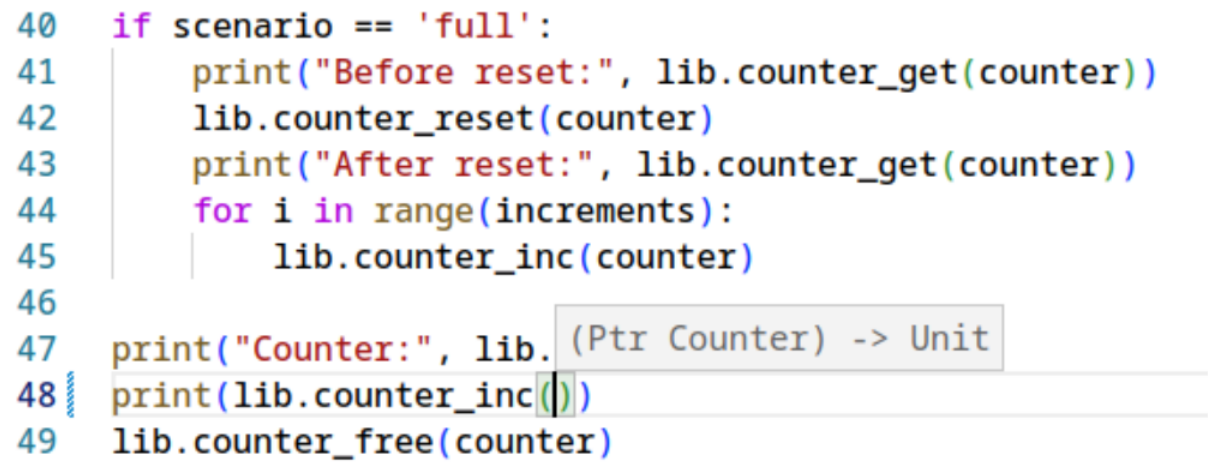

Рисунок 7.6 — Сценарий LSP «Signature help»

7.3 Пример 3 – Golang и JavaScript

7.3.1 Общее описание

Проект представляет собой простейшее фуллстак веб-приложение, с двунаправленным взаимодействием между клиентом и сервером. Клиентом выступает браузер, который подгружает HTML страницу и соответствующий JavaScript код для обеспечения интерактивности такой страницы. Сервер реализован на Golang и является стандартным HTTP сервером, обеспечивающи обмен данными посредством JSON. Таким образом, проект представляет стандартное приложение, использующее Rest API.

Сервер имеет внутреннее состояние, которое изменяется при соответствующих запросах с веб-страницы. Клиент, в свою очередь, обновляет состояние веб-страницы при ответе сервера.

Основной проблемой проекта является неявная связь между сетевыми путями сервера и клиента, а также отсутствие выраженной схемы данных, которыми они обмениваются.

7.3.2 Окружение

В данном проекте не используется явное окружение, однако серверу требуется сетевой порт для возможности запуска и отправки сообщений. Такая информация не используется в анализе, так как она имеет небольшую применимость.

7.3.3 Трансляция и решение

Анализ данного проекта является самым сложным из всех перечисленных примеров. Это связано с тем, что проекты, использующие взаимодействие по сети, имеют специализированные библиотеки и фреймворки, а уровень таких библиотек достаточно высоким и вовлекающим большое количество различных компонентов. В данном случае, при анализе сервера Golang потребуется вывести собственную структуру подграфа, обозначающую веб-

сервер. Используя интенсивный анализ потока данных, получается использовать эту структуру для инкапсуляции ключевых особенностей веб-сервера – наличие определенных путей, которые могут содержать информацию при применении различных запросов к ним (HTTP.GET или HTTP.POST).

Такая структура (или «идея») веб-сервера также ожидается и на стороне клиента, так как доступ к определенным путям возможен только при соблюдении определенного протокола взаимодействия (в данном случае HTTP и Rest API).

Помимо этих особенностей, веб-сервер имеет определение типа данных, передаваемых по сети, с автоматически генерируемой структурой (через использование тегов структур [35]). Такая информация является очень специфической и требует серьезного знания того, как используются теги структур от анализатора. Тем не менее, извлечение такой информации возможно и её отображение в граф областей допустимо.

Полученный граф областей вкупе с типовыми ограничениями представлен на рисунке 7.7.

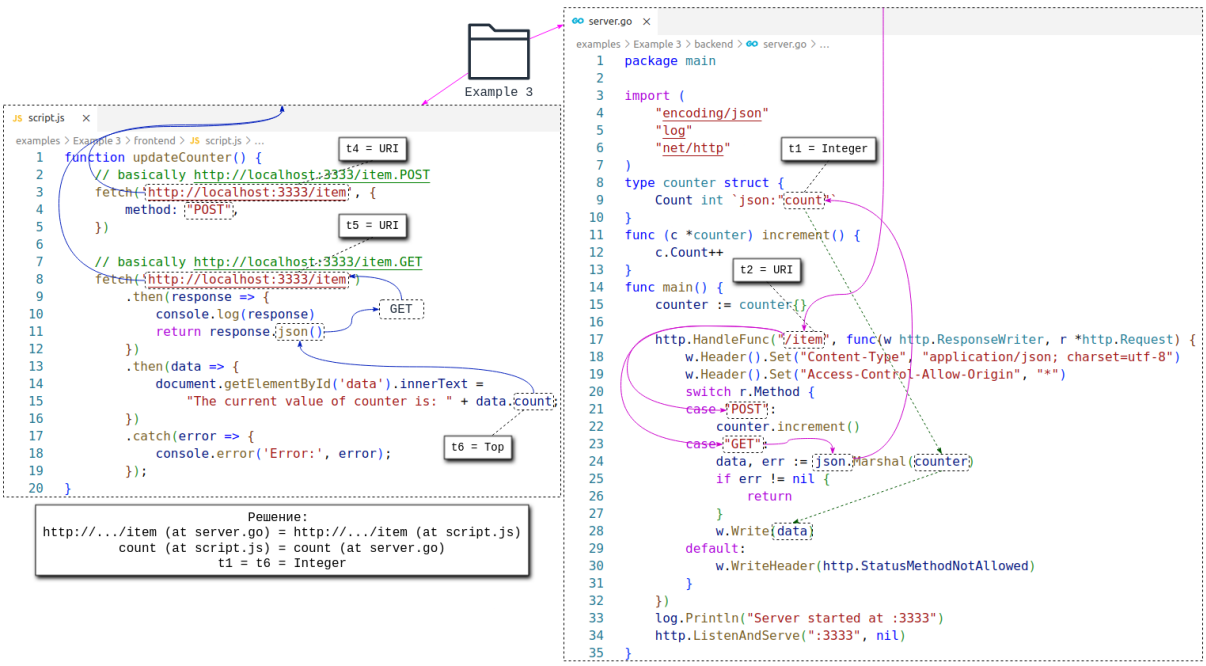

Рисунок 7.7 — Визуализация графа областей и типовых ограничений для примера 3

7.3.4 Онтология

В дополнение к рассмотренным типам, в онтологию вошли новые типы:

– Numeric – числовой тип неопределенного размера и точности (рациональное число),

– Bool – булевый тип,

– String – строка в случайной кодировке, исходящей из принятых в проекте,

– Unit – тип, населенный единственным значением.

Был добавлен структурный предикат «веб-сервер» для протокола HTTP. Для такого предиката выполняется ряд общих для веб-серверов условий: такой сервер должен быть представлен URI, он обязан иметь поля `GET, POST` и иные (в зависимости от проекта), а также такие поля должны содержать данные в определенном формате (в данном случае JSON). Также был добавлен предикат «JSON-объект», допускающий любой уровень вложенности областей видимости, но ограничивающий типы, присваиваемые идентификаторам (только Numeric, Bool, Unit, String).

7.3.5 Сценарии LSP

Для упрощения поддержки согласованности проектов между собой решено было реализовать два сценария: «Rename symbol» (переименование символа под курсором) и «Document diagnostics» (диагностика документа). Предполагаемые примеры внешнего вида реализации таких сценариев показаны на рисунке 7.8 и рисунке 7.9 соответственно.

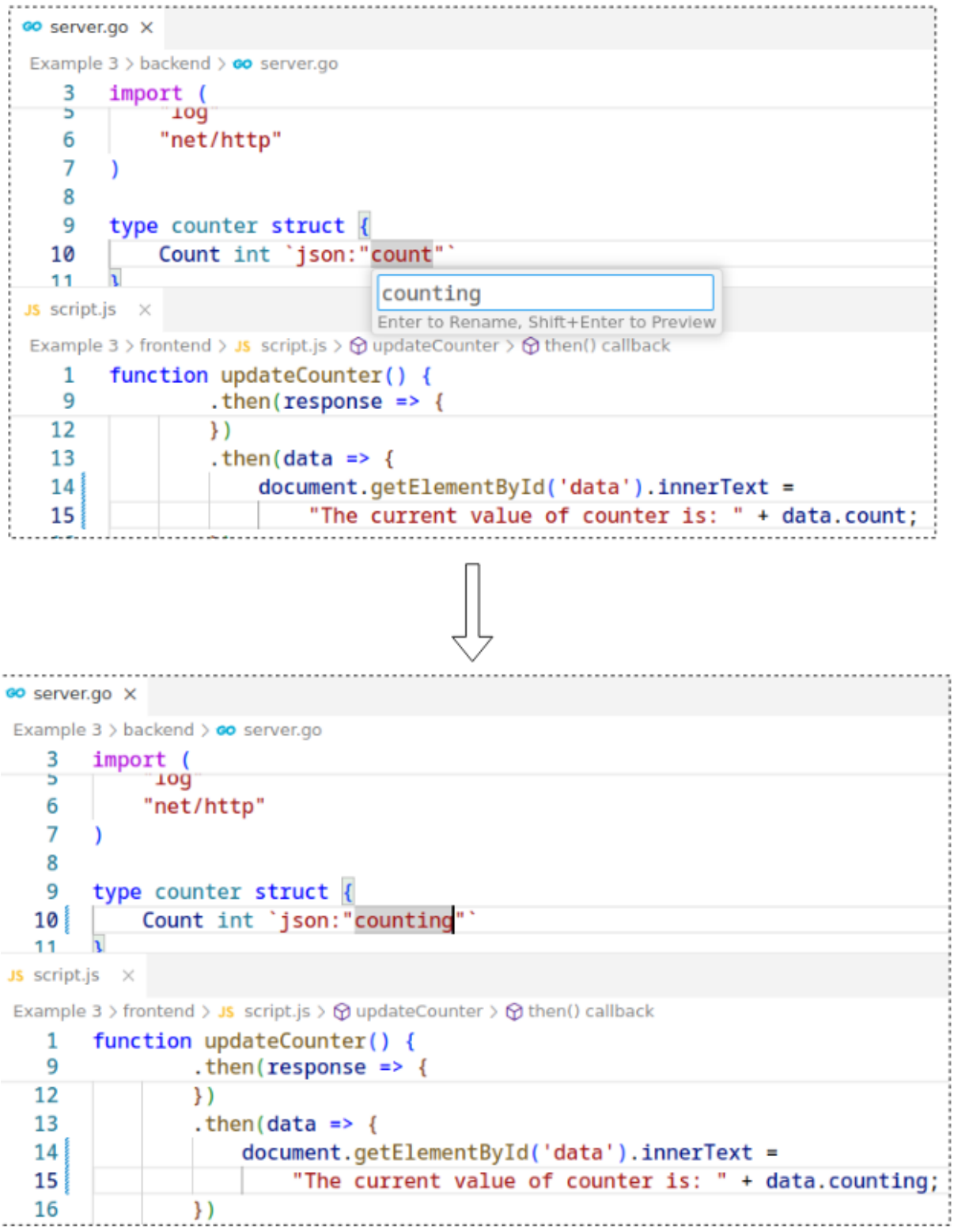

Рисунок 7.8 – Сценарий LSP «Rename symbol»

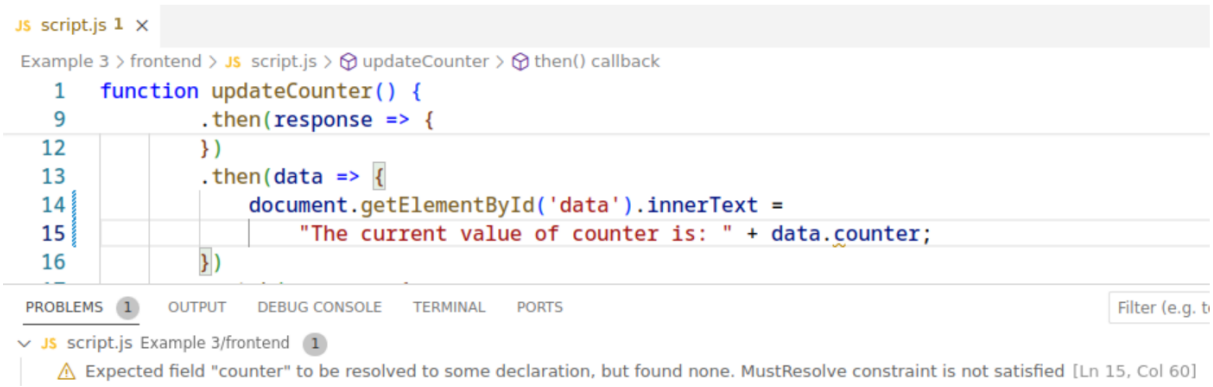

Рисунок 7.9 — Сценарий LSP «Document diagnostics»

ЗАКЛЮЧЕНИЕ

В ходе данной работы было проведено исследование существующих методов мультязыкового анализа, выявлены их достоинства и недостатки. Проанализированы существующие инструменты поддержки разработчика, поддерживающие анализ множества языков и их связей.

Был разработан метод мультиязыкового статического анализа, позволяющий обеспечить инструментальные средства поддержки разработчика. Данный метод отвечает требованиям к гибкости и языконезависимости, а также к корректности семантических свойств используемого представления.

Также, для подтверждения работоспособности данного метода был разработан мультиязыковой анализатор и адаптер избранного инструментального средства (сервера LSP). В ходе разработки были созданы протокол и онтология, вместе позволяющие обеспечивать согласованность мультиязыковых фрагментов кода. Была произведена тестовая интеграция сервера LSP в IDE Visual Studio Code с демонстрацией определенных сценариев использования LSP.

Задачи выпускной квалификационной работы выполнены в полном объеме, поставленная цель достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Mayer Philip, Bauer Alexander. An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects. — 2015. — 04.

2. Mayer Philip, Kirsch Michael, Le Minh-Anh. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. — 2017. — 12. — Т. 5.

3. Møller Anders, Schwartzbach Michael I. Static Program Analysis. — 2018. — October. — Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

4. Johnson S. C., Hill Murray. Lint, a C Program Checker. — 1978. — Режим доступа: https://api.semanticscholar.org/CorpusID:59749883.

5. Foundation Qt. Метаобъектный компилятор Qt 6. — 2024. — Режим доступа: https://doc.qt.io/qt-6/metaobjects.html (дата обращения: 25.03.2024).

6. Community React. Фреймворк React. — 2024. — Режим доступа: https://react.dev/ (дата обращения: 25.03.2024).

7. Pfeiffer Helge, Wasowski Andrzej. Cross-Language Support Mechanisms Significantly Aid Software Development. — 2015. — 07.

8. Pragmatic evidence of cross-language link detection: A systematic literature review / Latif Saira, Mushtaq Zaigham, Rasool Ghulam, Rustam Furqan, Aslam Naila и Ashraf Imran // Journal of Systems and Software. — 2023. — Т. 206. — С. 111825. — Режим доступа: https://www.sciencedirect.com/science/article/pii/S0164121223002200.

9. Moonen L. Generating robust parsers using island grammars // Proceedings Eighth Working Conference on Reverse Engineering. — 2001. — С. 13–22.

10. Hosry Aless, Anquetil Nicolas. External Dependencies in Software Development // Quality of Information and Communications Technology / под ред. Fernandes José Maria, Travassos Guilherme H., Lenarduzzi Valentina,

Li Xiaozhou. — Cham : Springer Nature Switzerland. — 2023. — С. 215–232.

11. Pangea: A Workbench for Statically Analyzing Multi-language Software Corpora / Caracciolo Andrea, Chis Andrei, Spasojevic Boris и Lungu Mircea // 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. — 2014. — С. 71–76.

12. Nierstrasz Oscar, Ducasse Stéphane, Gîrba Tudor. The story of Moose: an agile reengineering environment. — 2005. — 01. — Т. 30. — С. 1–10.

13. Demeyer Serge, Tichelaar S., Ducasse Stéphane. FAMIX 2. 1-the FAMOOS information exchange model. — 2001. — 01.

14. EcmaScript 262. — 2024. — Режим доступа: https://tc39.es/ecma262/ (дата обращения: 23.03.2024).

15. Java Native Interface. — 2024. — Режим доступа: https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html#java-native-interface-overview (дата обращения: 02.04.2024).

16. Bogar Anne, Lyons Damian, Baird David. Lightweight Call-Graph Construction for Multilingual Software Analysis. — 2018. — 01. — С. 362–371.

17. Kargar Masoud, Isazadeh Ayaz, Izadkhah Habib. Improving the modularization quality of heterogeneous multi-programming software systems by unifying structural and semantic concepts // The Journal of Supercomputing. — 2019. — Т. 76. — С. 87 – 121. — Режим доступа: https://api.semanticscholar.org/CorpusID:202733767.

18. B. Zheng W. & Hua. Effective Call Graph Construction for Multilingual Programs. — 2023. — Режим доступа: https://csslab-ustc.github.io/publications/2023/call-graph.pdf.

19. Webassembly. — 2024. — Режим доступа: https://webassembly.org/ (дата обращения: 05.04.2024).

20. Rakić Gordana, Budimac Zoran. Introducing enriched concrete syntax trees // Proc. of the 14th International Multiconference on Information Society

(IS), Collaboration, Software And Services In Information Society (CSS). — 2011. — 01. — Т. A. — С. 211–214.

21. Rust programming language. — 2024. — Режим доступа: https://www.rust-lang.org/ (дата обращения: 15.04.2024).

22. Граф свойств. — 2024. — Режим доступа: https://github.com/opencypher/openCypher/blob/master/docs/property-graph-model.adoc (дата обращения: 15.04.2024).

23. Формат RDF. — 2024. — Режим доступа: https://www.w3.org/TR/PR-rdf-syntax/Overview.html (дата обращения: 15.04.2024).

24. Milner Robin. A theory of type polymorphism in programming // Journal of Computer and System Sciences. — 1978. — Т. 17, № 3. — С. 348–375. — Режим доступа: https://www.sciencedirect.com/science/article/pii/0022000078900144.

25. A Theory of Name Resolution / Néron Pierre, Tolmach Andrew P., Visser Eelco и Wachsmuth Guido // Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings / под ред. Vitek Jan. — Springer. — 2015. — Т. 9032 из Lecture Notes in Computer Science. — С. 205–231. — Режим доступа: http://dx.doi.org/10.1007/978-3-662-46669-8_9.

26. A constraint language for static semantic analysis based on scope graphs / Antwerpen Hendrik, Neron Pierre, Tolmach Andrew, Visser Eelco и Wachsmuth Guido. — 2016. — 01. — С. 49–60.

27. Language-Independent Type-Dependent Name Resolution : Отчет : TUD-SERG-2015-006 / Delft University of Technology, Software Engineering Research Group ; исполн.: van Antwerpen Hendrik, Néron Pierre, Tolmach Andrew P. и др. — Delft, The Netherlands : 2015. — July.

28. Dezani-Ciancaglini Mariangiola, Hindley J.Roger. Intersection types for combinatory logic // Theoretical Computer Science. — 1992. — Т. 100,

№ 2. — С. 303–324. — Режим доступа: https://www.sciencedirect.com/science/article/pii/030439759290306Z.

29. Пример приложения интернет-магазина на C#. — 2024. — Режим доступа: https://github.com/dotnet-architecture/eShopOnWeb (дата обращения: 15.04.2024).

30. LSP specification. — 2024. — Режим доступа: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification (дата обращения: 15.04.2024).

31. JSON RPC протокол. — 2024. — Режим доступа: https://www.jsonrpc.org/specification (дата обращения: 28.04.2024).

32. Нативная виртуальная машина JavaScript для Golang. — 2024. — Режим доступа: https://github.com/dop251/goja (дата обращения: 28.04.2024).

33. Инструмент для распознавания языка программирования по входной строке. — 2024. — Режим доступа: https://github.com/yoeo/guesslang (дата обращения: 28.04.2024).

34. Сценарии использования LSP. — 2024. — Режим доступа: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#languageFeatures (дата обращения: 15.04.2024).

35. Теги структур в языке Golang. — 2024. — Режим доступа: https://go.dev/wiki/Well-known-struct-tags (дата обращения: 28.04.2024).

ПРИЛОЖЕНИЕ А

ПРОТОКОЛ МУЛЬТИЯЗЫКОВОГО АНАЛИЗАТОРА

```
// Честно взято из протокола LSP 3.17
export type URI = string;
export type integer = number;
export type uinteger = number;

export interface Position {
    line: uinteger
    character: uinteger
}

export interface Range {
    start: Position
    end: Position
}

interface From {
    language: string | undefined
}

export interface SourceFile extends From {
    uri: URI
    range?: Range
}

export interface SourceCode extends From {
    code: string
}

// TODO: здесь должны учитываться многие возможные локации (см
↪ LocationLink[])
```

```
export type Source = SourceFile | SourceCode | undefined

export interface Identifier {
    name: string
    source: Source
}

// По порядку: неизвестные объявления, неизвестные области,
 ↪  неизвестные типы
export type VariableType = "delta" | "sigma" | "tau"

export interface Variable {
    index: uinteger
    type: VariableType
}

export interface Scope {
    index: uinteger
    source: Source
}

export type NameCollectionType = "declared" | "referenced" |
 ↪  "visible"

export interface NameCollection {
    scope: Variable | Scope
    names: NameCollectionType
}

export interface UsageConstraint {
    identifier: Identifier
    scope: Variable | Scope
    usage: "declaration" | "reference"
```

```
}

export interface ResolutionConstraint {
    reference: Identifier
    declaration: Variable
}

export interface UniquenessConstraint {
    names: NameCollection
}

export interface TypeDeclarationConstraint {
    declaration: Identifier | Variable
    type: Variable
}

// Непрозрачный, структура зависит от онтологии и системы типов
export type Type = Object

export interface TypeEqualConstraint {
    lhs: Variable
    rhs: Variable | Type
}

export interface DirectEdgeConstraint {
    from: Variable | Scope
    to: Variable | Scope
    label: string
}

export interface AssociationConstraint {
    declaration: Identifier | Variable
    scope: Variable | Scope
```

```
}

export interface NominalEdgeConstraint {
    scope: Variable | Scope
    reference: Identifier
    label: string
}

export interface SubsetConstraint {
    lhs: NameCollection
    rhs: NameCollection
}

export interface MustResolveConstraint {
    reference: Identifier
    scope: Variable | Scope
}

export interface EssentialConstraint {
    declaration: Identifier
    scope: Variable | Scope
}

export interface ExclusiveConstraint {
    declaration: Identifier
    scope: Variable | Scope
}

export interface IconicConstraint {
    declaration: Identifier
}

export type Constraint =
```

```
    UsageConstraint |
    ResolutionConstraint |
    UniquenessConstraint |
    TypeDeclarationConstraint |
    TypeEqualConstraint |
    DirectEdgeConstraint |
    AssociationConstraint |
    NominalEdgeConstraint |
    SubsetConstraint |
    MustResolveConstraint |
    EssentialConstraint |
    ExclusiveConstraint |
    IconicConstraint

// Для каждого *Response -- сценарий ошибки отдается согласно
  ↪ JSON RPC


export interface SubtranslationRequest {
    code: Source
    ontology: URI
}

export interface SubtranslationResponse {
    constraints: Constraint[]
    unrecognized: Source
}

// метод crossy/translate
export type TranslationRequest = SubtranslationRequest
export type TranslationResponse = SubtranslationResponse

// метод crossy/solve
```

```
export interface SolveRequest {
    constraints: Constraint[]
    ontology: URI
}

export type Substitution = {
    tag: "delta"
    data: Identifier
} | {
    tag: "sigma"
    data: Scope
} | {
    tag: "tau"
    data: Type
}

export interface SolveResponse {
    substitution: Map<Variable, Substitution>
}

// метод crossy/analyze
// объединение crossy/translate и crossy/solve с применением полученной подстановки
export interface AnalyzeRequest {
    code: Source
    ontology: URI
}

export interface AnalyzeResponse {
    constraints: Constraint[] // эти ограничения не должны содержать переменных
}

```

```
// метод crossy/configure
export interface ConfigureRequest {
    filesystem?: { workspace: URI }
    systemVariables?: boolean
    utilities?: boolean
}

export type FilesystemConfiguration = UsageConstraint |
    ↪ AssociationConstraint | DirectEdgeConstraint // рекурсивное
    ↪ дерево каталогов и файлов относительно проекта
export type SystemVariablesConfiguration =
    ↪ TypeDeclarationConstraint | TypeEqualConstraint //
    ↪ присваивание каждой системной переменной типа/статического
    ↪ значения
export type UtilitiesConfiguration = TypeDeclarationConstraint |
    ↪ TypeEqualConstraint // присваивание каждой утилите
    ↪ (библиотеке, приложению) типа/статического значения

export interface ConfigureResponse {
    configuration: (FilesystemConfiguration |
     ↪  SystemVariablesConfiguration | UtilitiesConfiguration)[]
    code: { language: string, source: Source }[]
}
```