Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО”

Факультет Программной инженерии и компьютерной техники

Образовательная программа Системное и прикладное программное обеспечение

Направление подготовки (специальность) 09.04.04 Программная инженерия

ОТЧЕТ

о научно-исследовательской работе

Тема задания: «Выработка методов к анализу мультиязыковых текстов программ»

Обучающийся: Орловский М.Ю (**Фамилия И.О.**) Р4116 (**номер группы**)

Руководитель практики от университета: Маркина Т.А, доцент факультета ПИиКТ

САНКТ-ПЕТЕРБУРГ
2023 г.

СОДЕРЖАНИЕ

1 Эпоха

1.1 Исследовать возможность обобщения предлагаемого метода на сочетание различных языков программирования

При изначальной выработке метода был рассмотрен лишь один сценарий использования для метода – веб разработка. После рассмотрения иных сценариев использования, в первую очередь связанных с реальной коммерческой разработкой, возникла необходимость обобщения предлагаемого метода.

Был проанализирован характер информации, выявляемой методом и возможности применения такой информации в дальнейших сценариях. Таким образом, естественным образом была выработана единая семантика связи (семантика ребра в сети) – это семантика «Зависимость». Такая семантика была выбрана по нескольким причинам:

1) Многие семантические зависимости узлов являются бесполезными на этапе анализа – по ним сложно делать какие-либо выводы о структуре сети в связи с неупорядоченностью узлов при их связывании;

2) Семантические зависимости разного рода усложняют анализатор, делая его более специфичным и снижая гибкость;

3) Предполагается, что все семантические отношения между узлами можно будет вывести из готовой сети и соответствующей онтологии, при практическом применении этой сети в конкретном инструменте.

Таким образом, предполагается, что данная структура узла будет более гибкой в отношении способов хранения информации о семантике конкретных синтаксических конструкций.

Также, изменения коснулись информации, являющейся репрезентацией фрагмента кода в семантическом узле. Вместо позиционной информации (линия в файле, колонка в файле, путь к файлу) было принято решение использовать более структурированную информацию.

Выбор пал на использование конкретного AST подлежащего кода языка и на это есть несколько причин:

1) Появляется возможность получения доступа к AST из семантического узла, что полезно для инструментальных средств;

2) Предполагается, что использование структурированного представления может быть полезно для интеграции других структурированных представлений, если инструмент не имеет поддержки вывода AST.

Относительно поля «Атрибуты», отвечающего для связывания узлов, было принято решение ввести более широкую семантику «сравнения» узлов. В связи с этим, это поле решено представлять как гетерогенный список.

Гетерогенный список – список, состоящий из сущностей разных типов. Таким списком является, например, список или массив в языках с динамической типизацией (Python, JavaScript, Scheme, Clojure). Так как на типы элементов нет ограничений, список тоже может являться элементом такого списка, что задает рекурсивную вложенность и позволяет формировать дерево.

Предполагается реализация такого списка в формате s-выражений. Такой формат представления используется в языках семейства Lisp, что позволяет реализовывать мощные алгоритмы обработки кода как данных. В рамках данной работы данный формат избран по следующим причинам:

1) Простая рекурсивная реализация – три функции для манипуляции данными и одна для сравнения, представляющие собой простейшие алгоритмы;

2) Динамичность в отношении репрезентации данных – так как формат поддерживает древовидную структуру, многие структуры данных тривиально представляются в нем как дерево естественным образом;

3) Человеко-читаемость и простая машинная обработка.

Обновленная структура семантического узла представлена в таблице 1.

Таблица 1 – Обновленная структура семантического узла

Тип поля	Имя поля	Семантика
Узел	Идентификатор AST узла	Указывает позицию анализируемой синтаксической конструкции

Атрибуты	Гетерогенный список	Содержит список атрибутов узла
Вид	Перечисление	Является способом типизации конкретного семантического узла, исходя из ЯП, который он описывает

Алгоритм связывания узлов при этом не претерпевает явных изменений и действует по тому же принципу, что и ранее.

1.2 **Исследовать наиболее часто используемые парадигмы программирования (процедурное, ОО, декларативное) в контексте мультиязыкового анализа**

При изучении различных источников литературы о дизайне ЯП, было выяснено, что четкого разделения современных языков на определённые классы не существует. Точнее сказать, оно невозможно в общем случае, что не подходит для сценария межъязыкового анализа. Поэтому, было принято решение проводить классификацию самостоятельно.

Предполагается в качестве базы использовать общепринятое разделение языков по парадигмам [1]. Данное разделение представлено в таблице 2.

Таблица 2 – Базовая классификация ЯП по парадигмам

Парадигма	Основные особенности	Представители
Императивная	Переменные как способ манипуляции памятью; Мутабельность как основной способ проведения вычислений; Описание вычислений как конкретного набора инструкций	C, Pascal, Java, Golang, Bash, JavaScript и многие другие
Декларативная	Описание результата и его свойств, как способ его получения; Описание вычислений как высокоуровневых объявлений;	ML, Lisp, Haskell, SQL, Make, HTML и также многие другие

В качестве расширения предполагается использование парадигм, наиболее часто представимых в индустрии. Таким образом, многие промышленные языки можно будет описывать как единое представление в отношении некоторых семантических конструкций.

Исходя из данных соображений, вводится классификация, представленная в таблице 3.

Таблица 3 – Базовая классификация ЯП по парадигмам

Парадигма	Основные особенности	Представители
Процедурная и структурная	Инструкции сгруппированы в процедуры/подпрограммы; переменные как способ взаимодействия с памятью	C, Pascal, Golang, Bash, и многие другие
Объектно-ориентированная	Данные и код инкапсулированы в единое представление – объект; Объекты общаются посредством передачи сообщений друг-другу	Java (отчасти), C# (отчасти), JavaScript (отчасти), C++ (отчасти), Smalltalk, Erlang, Elixir, Self и другие
Функциональная	Функции, как способ выстроить процесс вычисления; Иммутабельность как фундамент формальных гарантий в коде	ML, Rust (отчасти), Haskell, Erlang, Elixir, Lisp
Макро	Процесс вычислений проводится путем лексических подстановок строк для достижения конечной строки; встречаются варианты, оперирующие на AST	Препроцессор C, Make, макросистема Rust, различные DSL в Web фреймворках

Данное разделение позволяет достигнуть следующих целей:

1) Исходя из рассматриваемых языков можно выбрать наиболее общие семантические конструкции, присущие языкам данной группы, и на их основе сформировать онтологию – такая онтология будет описывать все языки данной группы;

2) Данная классификация позволяет провести анализ сценариев использования в отношении к предметным областям, в которых они актуальны;

3) Рассмотрение кодогенерации в данной работе также является необходимостью, так как оно присуще большому количеству языков, активно используемых в индустрии.

Также, в контексте работы предполагается использование разделения другого рода. Предполагается разделение языков на две категории – GPL (General Purpose Language) и DSL (Domain Specific Language).

GPL представляют собой языки довольно сильно оторванные от конкретной предметной области и предназначенные для различных, порой сильно, областей. DSL являются языками которые предназначены в первую очередь для решения определенных задач в конкретной предметной области.

Такое разделение вводится чтобы отразить полноту зависимости конкретного языка от предметной области. Это полезно для неформального измерения сложности той или иной предметной области при анализе задействованных в ней языков.

1.3 **Рассмотреть различные подходы к представлению семантической информации**

Основным способом представления информации (и самым универсальным) является семантическая сеть. В данной работе был избран именно такой подход, однако, семантические сети трудно структурировать и формализовывать. И хотя в дальнейшем будет использоваться именно этот подход к представлению семантической информации, полезно рассмотреть иные подходы, активно используемые в компиляторах, оптимизаторах и анализаторах. Такие подходы отражены в таблице 4.

Таблица 4 – Представление информации об исходном коде

Название представления	Основные особенности	Плюсы (в рамках данной работы)	Минусы (в рамках данной работы)
AST	Представление иерархической структуры программы, посредством связывания её синтаксических конструкций в единое дерево	Легко анализируется, может быть использовано в анализах, где порядок анализа не важен; легко визуализируется; универсально для многих видов анализа	Сложно изменяется; не подходит для анализов, которые зависят от порядка; некоторые анализы неприменимы
CFG	Представление пути исполнения программы, моделируется связями (переходами) между узлами (операторами или выражениями)	Многие другие анализы легко реализуются через CFG; хорошо подходит для императивных языков	Плохо подходит для неимперативных языков, где путь исполнения программы задан неявно; сложнее реализовать чем AST
Семантическая сеть	Представление сущностей и их зависимостей в виде графа, узлы которого определяют сущности, а связи - отношения	Легко синтезируется; очень универсально; очень гибко и может быть использовано в различных анализах	Слабо структурировано; нет четкой схемы как связывать узлы между собой, по какому принципу

			выявлять сущности и отношения
Иерархия типов	Представление информации о связях сущностей в виде типов этих сущностей, формирующих определенную иерархию	Может быть простым вариантом представления большого объема информации в ОО и функциональных системах; имеет много формальных средств и методов для анализа	Сильно зависит от избранного языка; может не нести ценности, если язык имеет слабую типизацию; довольно сложна для анализа в общем случае
Онтология	Представление, основанное на формальных знаниях в конкретной предметной области	Однажды специфицированная онтология может быть использована в другом языке при наличии схожих семантик	Быстро становится сложной при расширении; обычно, предметные области (а в особенности ЯП) трудно формализуемы

Стоит заметить, что существует еще большое множество иных представлений, которые слабо подходят для описания семантики языков программирования: фреймовые, логические и вероятностные модели, а также нейронные сети. Хоть они и представляют ценность для реализации различных анализов, в рамках данной работы они либо слишком сложны для применения, либо не обеспечивают необходимой точности.

Исходя из перечисленных подходов к представлению семантической информации, можно заключить, что использование семантических сетей стоит совместить с иными способами представления информации. Поле «узел» семантического узла может послужить отличным источником дополнительной информации о внутреннем анализе фрагмента за счет того, что «узел» может представлять собой различные структуры – AST, CFG или другое графовое/древовидное представление.

2 Эпоха

2.1 Провести анализ текущих решений в области мультиязыкового анализа на прикладном уровне (IDE и инструментальные средства)

На данный момент, в индустрии разработки ПО мало качественных и достаточных инструментов для мультиязыковой разработки, чего очень не хватает пользователям языков программирования [2]. Однако, существует ряд решений, направленных на обеспечение инструментальной поддержки разработки. Многие решения из этой области разработаны для специфичных предметных областей, в первую очередь для веб-разработки. В рамках данной работы были рассмотрены несколько инструментов, поддерживающие (либо имеющие возможность) мультиязыковой разработки для анализа межъязыковых связей.

2.1.1 JetBrains Rider

JetBrains Rider это кроссплатформенная IDE имеющая поддержку различных языков, в первую очередь .Net семейства (C#, Visual Basic, F# и прочие), а также ряда DSL которые используются в соответствующих фреймворках (Blazor, XAML и др.) [3]. Она использует ReSharper как анализатор и инструмент для обеспечения продуктивности разработчика. Вкупе с IntelliJ IDEA, которая служит базисом для IDE, Rider обеспечивает большое количество удобной для разработчика функциональности. Краткий перечень такой функциональности включает:

— Обнаружение зависимостей между символами на мультиязыковом уровне;

— Переименование символов по их зависимостям;

— Контекстная информация о синтаксическом элементе;

— Обнаружение несоответствий при межъязыковом взаимодействии;

— Поиск определения или объявления символа в межъязыковом контексте;

— Поиск всех использований определенного символа в межъязыковом контексте;

— Автодополнение, доступное в межъязыковом контексте.

Несмотря на ряд продуктивных возможностей, можно выявить недостатки Rider, касающиеся межъязыкового анализа и рефакторингов. Одним из основных недостатков является узкий спектр поддерживаемых DSL – Rider хорошо взаимодействует с исходным кодом Blazor, но, к примеру файлы конфигурации или файлы проектов/решений им воспринимаются плохо. Ни одна (за исключением пункта 7) из возможностей перечисленных выше не применима к файлам JSON, YAML или SLN.

2.1.2 SonarQube

SonarQube это автоматический инструмент для проверки кода, предназначенный для увеличения качества кода, раннего устранения ошибок и повышения скорости внедрения. Основным инструментом, привлекающим внимание в рамках данного проекта, является SonarLint – универсальный статический анализатор, поддерживающий множество популярных языков программирования. SonarQube имеет возможность анализировать мультиязыковые проекты, хотя об анализе межъязыковых связей информации нет [4].

Возможности инструмента схожи с JetBrains ReSharper и также предоставляют результаты анализа в реальном времени, для возможности интеграции инструмента в IDE.

Одним из явных недостатков инструмента можно выявить высокую связность с инфраструктурой Sonar – его возможности ограничены, если не использовать полноценный SonarQube сервер и большое количество дополнительных инструментов, что может затруднить интеграцию SonarLint в небольшой проект.

2.1.3 Mulang

Mulang позиционируется как универсальный, мультиязыковой и мультипарадигменный статический анализатор, ориентированный на обнаружение ошибок и формирование предикатов (ожиданий) о коде [5].

Языки, поддерживаемые Mulang включают:

— C;

— Haskell;

— Java;

— JavaScript (ES6);

— Python (2 и 3);

— Prolog.

Также, имеется поддержка Ruby и PHP через специфические языковые инструменты.

Так как проект является открытым, есть возможность добавить поддержку определенных языков при необходимости.

Mulang является инструментом, позволяющим работать с кодом, как с базой знаний и, соответственно, не поддерживает полноценной интеграции с IDE. Однако, исходя из набора «ожиданий», сформированного разработчиками и из возможности определять свои «ожидания» можно заключить, что проект концептуально применим в IDE при разработке соответствующего плагина.

Небольшой набор «ожиданий» и ошибок, которые способен обнаруживать Mulang включает:

1) «Делает ли элемент вызов определенной функции?»;
2) «Присутствует ли переменная в выражении?»;
3) «Представлено ли вычисление как рекурсия?»;
4) «Используется ли оператор «если»?»;
5) Дупликация кода;
6) Ошибка в имени идентификатора;
7) Недостижимый код;

8) Длинный список параметров.

Таким образом, в открытом доступе нет инструментов, способных предоставить мультиязыковой анализ, необходимый в рамках данной работы.

2.2 **Провести анализ моделей представления семантической информации программ в контексте различных ЯП и технологических стеков**

Так как количество анализаторов с открытым исходным кодом обеспечивающих мультиязыковой анализ мало, были рассмотрены два инструмента и представления семантической информации, используемые в них.

2.2.1 Multilingual static analysis tool

В контексте MLSA используются различные модели представления семантической информации, а именно:

— Граф вызовов функций;

— CFG;

— Граф присваиваний;

— Граф зависимостей на уровне файлов и директорий.

Данные структуры представления информации замечательно работают в контексте одного языка, но плохо обобщаются на мультиязыковой анализ. Это связано со специфичностью конкретного анализа – к примеру граф присваиваний бесполезен в языке, присваиваний не поддерживающем.

Поэтому, вкупе с данными семантическими представлениями имеет смысл использовать более общую структуру, объединяющую данные представления на высоком уровне. Предполагается, что такая структура в том числе может быть семантической сетью, с единственной семантикой связи – «зависит».

2.2.2 Mulang

Mulang в свою очередь представляет особый интерес благодаря наличию унифицированного представления называемого авторами проекта «Abstract semantic tree».

Такое дерево состоит из узлов, попадающих в одну из 5 категорий:

— Выражения;

— Шаблоны;

— Типы;

— Равенства;

— Генераторы.

Равенства можно описать как отображение из списка шаблонов (параметров функции или параметров match выражения) в тело (выражение либо набор частичных выражений). Они семантически представляют собой математическое отображение из входных параметров/шаблонов в одно или несколько выражений и, следовательно, подходят в первую очередь для функциональных языков. Генераторы семантически представляют собой выражения, имеющие ленивый (по надобности) тип вычислений.

На остальных трех категориях стоит остановится поподробнее.

2.2.2.1 Выражения

Выражение является основной описания любого типа вычисления. Mulang AST моделирует через выражения иные конструкции, описывающие вычисления – в первую очередь операторы и объявления. Такой подход является гибким, хотя и размывает границу между чистыми вычислениями и вычислениями с сайд-эффектами.

В Mulang представлен весь стандартный, несколько упрощенный набор семантических конструкций, описывающих выражения, встречающийся в многих популярных императивных и функциональных языках. Из-за высокого уровня абстракции можно выявить некоторые недоработки онтологии.

Она выражает общие языковые конструкции, но делает это неоднородно, к примеру цикл языка C описан как отдельная конструкция ForLoop, хотя семантически она может являться общим случаем итерации, так как через неё выражаема конструкция For (являющаяся генератором). Также, онтология не покрывает семантику некоторых языков, хотя причин для этого не имеет. К примеру, конструкция Class имеет лишь одно поле для обозначения базового класса, однако к C++ это в общем случае не подходит.

2.2.2.2 Шаблоны

Шаблоны в терминах Mulang AST обозначают параметры или, собственно, шаблоны, предназначенные для обозначения хода вычислений. Они не несут значений сами по себе и являются абстракцией над многими управляющими и структурирующими конструкциями языков. К примеру, они могут быть использованы для обозначения параметров функций, параметров в выражениях match и switch, а также для обозначения литеральных конструкций (числа, строки, списки, кортежи и т. д.).

При этом многие шаблонные конструкции удаляют изначальную семантическую информацию о моделируемых семантических концепциях и служат, скорее, как ad-hoc решение, направленное на выражение минимальной информации о конструкции.

2.2.2.3 Типы

Типы в Mulang AST представляют собой довольно слабый, но тем не менее, достаточный набор семантик для описания операций над типами. Типы обозначают операции, которые могу проводиться над заданными типами как над множествами или функциями над множествами. К примеру, есть возможность задать параметризированный тип или задать ограничения на тип. Также есть возможность явного обозначения типа у определенного идентификатора.

Стоит заметить, что все конструкции оперируют над строками – таким образом, большая часть специфичной семантической информации остается необработанной и, следовательно, неиспользованной.

Если подытожить, то основные характеристики данного семантического представления включают:

1) Представление достаточно полное для многих языков (особенно высокоуровневых);

2) Некоторые конструкции выражений могут быть доработаны или расширены, в зависимости от задачи;

3) Конструкции шаблонов и равенств позволяют поверхностно описать необходимую семантику управляющих и структурных конструкций языка;

4) Базовое описание типов является малопригодным в общем случае, т. к. не имеет подкрепления в виде формальной системы и представляет собой набор для описания конструкций типов из различных языков;

5) В нотации представлений из-за специфики решаемой инструментом задачи не задает явным образом контекст времени связывания сущностей и идентификаторов, что очень важно для поддержки кодогенерации.

В итоге семантическое представление рассмотренных инструментов хоть и не подходит полностью для решения задачи, поставленной в данной работе, но может послужить источником для реализации некоторых аспектов собственного представления. В том числе имеет смысл использования некоторых конструкций представления типовой информации, задействованной в инструменте Mulang. В отношении MLSA можно сказать, что комбинация различных представлений внутриязыковой семантики на уровне специфичных представлений плохо расширяемо. Для решения такой проблемы предполагается использование более обобщенного представления.

2.3 **Исследование влияния прикладных областей языков на возможность обобщения метода мультиязыкового анализа**

Согласно исследованию [6], проведенному для выяснения важных аспектов мультиязыковой разработки, было выяснено что большая часть

разработчиков, использующих несколько языков в проекте, занимается веб-разработкой и клиент-серверными решениями. Также, было выяснено, что наиболее частым сочетанием языков была пара GPL/DSL, что объясняется большим количеством DSL существующих в индустрии веб-разработки.

Согласно этому исследованию, наиболее популярные возможности, предоставляемые инструментами разработчика, включали в себя (от частых к редким по инструментальной поддержке):

1) Подсветка синтаксиса;
2) Переименование схожих семантически идентификаторов;
3) Навигация по коду;
4) Анализ ошибок и нарушений консистентности.

Исходя из исследования [7], основные прикладные области, имеющие явную гетерогенную структуру в отношении языков программирования, включают:

1) Энтерпрайз разработку (J2EE, .NET Platform);
2) Веб-разработку (современные JavaScript фреймворки, Electron);
3) Встроенные системы.

Также, стоит учесть, что и в иных областях нередки ситуации использования различных языков (в первую очередь это касается файлов конфигурации или сборки).

Основные сложности, которые существуют при анализе мультиязыковых текстов программ, применимо к разным предметным областям, представлены в таблице 5.

Таблица 5 – Сложности мультиязыкового анализа

Предметная область	Проблема	Пояснение
Энтерпрайз, Веб-разработка	Большое количество различных языков, созданных для разных целей	Использование разных языков для фронтенд и бекенд разработки, а также для операций с БД является очень частым явлением и вкупе с другими языками для конфигурации и сборки может очень усложнить проект
Энтерпрайз	Широко распределенная система зависимостей	Зависимости между языками могут существовать не только на привычном уровне исходного кода, но также и на

		уровне файлов или даже целых отдельных API (например HTTP)
Встроенные системы	Большое разнообразие аппаратных средств	Из-за высокого разнообразия аппаратных средств, такие универсальные языки как C или ассемблер не имеют достаточной выразительной силы, в связи с чем возникает необходимость в дополнительных средствах (например, в кодогенерации)
Встроенные системы	Специфическая семантика в зависимости от аппаратных средств	Многие компиляторы для встроенных систем (C, C++, Python) не всегда следуют стандартам в угоду машинно-специфичным возможностям, из-за чего семантика исходного языка может быть нарушена, что затрудняет статический анализ

Таким образом, обобщение мультиязыкового анализатора выглядит как достаточно сложная задача. В отношении первых двух проблем, перечисленных в таблице, решением может быть достаточно полная формальная онтология, поддерживающая широкий набор используемых языков. В отношении оставшихся двух проблем предполагается соответствующая конфигурация метода при работе в такой предметной области.

3 Эпоха

3.1 Рассмотреть природу входной метаинформации и способы её обработки и хранения

Метод, описываемый в данной работе, является агрегатором различных специфичных анализаторов, каждый из который проводит свой собственный анализ для конкретного языка программирования.

Данный анализ называется внутриязыковым и обычно проводится над AST, полученных каким-либо образом для избранных программ, написанных на избранных языках программирования. Способ получения AST, специфический анализ и язык программирования, на котором реализован анализ не имеют для метода анализа, рассматриваемого в данной работе, особого значения.

Анализатор вправе использовать любые методы обхода AST и извлечения информации из него. Также, допустимо использование других структур данных, например CFG или DFG до тех пор, пока они имеют единое представление.

В дальнейшем, предполагается использование таких представлений для получения готовой внутриязыковой семантической информации.

3.2 Сформулировать формально метод языкового анализа

В ходе изучения предметной области, была прочитана статья [8] описывающая формализм для структурирования программ, используя такую сущность как модуль. Модуль по природе своей является инструментом разделения функциональности программы на части, способные взаимодействовать между собой.

В статье описывается, предположительно, первая формально доказанная, в отношении корректности, система модулей для языка программирования. Формально также вводятся такие понятия как окружение, сигнатура, привязка, связывание. Также вводится такое понятие как набор

связей (англ. linkset) представляющее собой отображение модуля после процесса связывания.

Статья представляет интерес в первую очередь по причине описания формально обоснованного фреймворка модульной системы, которая может быть напрямую применена в данной работе. Действительно, в отношении мультиязыкового анализа на этапе межъязыкового анализа можно утверждать, что взаимодействующие фрагменты кода на разных языках есть не что иное как программные модули, и их связывание позволяет получить описывающую их взаимодействие семантическую сеть.

Таким образом, решено было использовать терминологию, описанную в данной статье и адаптировать предлагаемую систему модулей под задачу мультиязыкового анализа. Стоит сразу заметить, что предполагается частичная адаптация терминологии и концепций, но для неё необходимы частичные изменения, что приводит к потере формально доказанной корректности такой системы.

3.2.1 Формирование модульной системы

Для избежания путаницы с модульными системами, представленными в различных языках программирования, термин «модуль» в данной работе применяться не будет. Взамен него предлагается использовать термин «Фрагмент кода».

Фрагмент кода состоит из набора утверждений. Окружением называется утверждение, используемое для объявления зависимости фрагмента от другого фрагмента. Сигнатурой называется утверждение, используемое для обозначения возможности создания зависимости от данного фрагмента. Каждое окружение должно быть связано только с одной сигнатурой, однако сигнатура может быть использована во многих окружениях. Также, у каждого фрагмента есть внутренние зависимости, представляемые избранным внутриязыковым анализатором способом.

Фрагменты кода выявляются на этапе внутриязыкового анализа и создаются независимо относительно других фрагментов. Они представляют

собой логические единицы, поэтому наличие файла с программным кодом необязательно, достаточно возможности организации какой-либо ссылки на порцию кода, описываемую фрагментом.

3.2.2 Система типов для организации связей

Каждое утверждение имеет тип – являющийся, в своем роде, уникальным идентификатором утверждения и в то же время задающий правила связывания фрагментов через связывание сигнатур с соответствующими окружениями. Так как проблема связывания фрагментов стоит на межъязыковом уровне, вводится простая и очень обобщенная система типов, базирующаяся на типизированном лямбда-исчислении первого порядка.

Описание нотации такой системы в BNF нотации:

```
A, B ::= N | A -> B | A x B | A + B | Any | None
a, b ::= a | \(a : A)b | b(a)
```

В данной системе задается следующий частичный порядок:

1) Любой тип принадлежит типу Any;
2) Ни один тип не принадлежит типу None;
3) Тип N является номинальным типом и населен только одним значением.

Такая простая система типов позволяет задавать различные семантические отношения, достаточные для нужд анализа.

Описанные термы могут быть использованы как дополнение к типам. Их основное использование – задание дополнительной семантики в случае, если через тип она плохо выразима или невыразима вовсе.

В избранных сценариях использования также были использованы конструкторы типов (например List, обозначающий список элементов). Стоит заметить, что последнее выражаемо через сумму произведений избранных типов, поэтому такой конструктор и подобные ему служат лишь для упрощения записи.

3.2.3 Связывание фрагментов кода

Процесс связывания можно описать следующим псевдокодом:

```
let link_fragments(m1, m2) =
     for i in m1.environment:
          for e in m2.signature:
               if i.type == e.type and
                  i.value == e.value and
                  was_not_linked(i, e):
                    link(i, e)

for (m1, m2) in fragments.unique_pairs():
     if ontology.languages_compatible(m1.language, m2.language):
          link_fragments(m1, m2)
          link_fragments(m2, m1)
```

Вследствие широкой семантической природы связей между фрагментами может возникнуть ситуация, при которой необходимо задать порядок связывания фрагменты между собой. Предполагается, что такой порядок необходимо выстраивать только в рамках внутриязыкового анализа (т. е. только фрагменты, относящиеся к одному языку, будут иметь определённый порядок).

Таким образом вводится порядок (как целочисленный номер), назначаемый каждому фрагменты на этапе внутриязыкового анализа и обеспечивающий правильное связывание.

В алгоритме, представленном выше, порядок вводится через упорядочивание уникальных пар модулей по их целочисленному номеру.

3.3 **Выбрать оптимальные структуры данных для представления метаинформации и хранения результатов анализа**

Для представления метаинформации в конкретном фрагменте, как и в случае семантического узла предполагается использование гетерогенного списка. Это в первую очередь касается динамических структур данных, таких как типы и термы для описания окружений и сигнатур.

Для хранения результатов анализа, а именно пар связей узлов, возможно использование бинарного формата данных либо же уже упомянутых s-выражений. Одним из преимуществ s-выражений в данном

случае будет выступать их простота и наглядность при преобразовании в текстовое представление.

Такой формат очень легко обрабатывается инструментом и, так как структура данных заранее известна и фиксирована, не требует сложных манипуляций. Это упрощает интеграцию анализатора.

4 Эпоха

4.1 Провести разработку прототипа для тестирования метода в различных сценариях

Для разработки анализатора был избран язык Go, так как он является очень простым инструментом для прототипирования и, в отличие от языков с отсутствием статической типизации, является куда более надежным в отношении процесса разработки и тестирования.

Стоит упомянуть, что избранный стек не влияет на применимость анализатора, поэтому мог быть избран любой другой стек технологий.

4.2 Разработать избранную модель представления семантической информации для использования в прототипе

В ходе работы были реализованы описанные выше структуры для описания семантической информации, а именно:

— Фрагмент кода;

— Утверждение, окружение и сигнатура;

— Гетерогенный список;

— Онтология.

Следует заметить, что ввиду статической природы языка Go, реализация гетерогенного списка является довольно неоптимальной и немного переусложненной. Данные проблемы можно решить выбором языка с более мощной системой типов, позволяющего использовать больший контроль над представлением данный (например, C++ или Rust).

4.3 Провести тестирование и собрать соответствующую статистику

4.3.1 Описание сценариев использования

При разработке анализатора были рассмотрены сценарии использования, отраженные в таблице 6.

Таблица 6 – Рассматриваемые сценарии использования

Название сценария	Описание сценария	Рассматриваемые особенности
Клиент-серверное взаимодействие по HTTP	Используются два языка: C# и JavaScript. На первом реализован сервер с использованием ASP.NET Core, на втором клиентское веб-приложение, делающее запросу к этому серверу. Очень популярный сценарий использования в веб-разработке	Обобщенный вызов – моделирование зависимостей как вызовов функций; Рассмотрение взаимодействия нетипизированного и типизированного языков;
Вызов C функции из Python	Используются три языка: C, Python и Shell. На первом реализована библиотека для быстрых вычислений, на втором инфраструктура для манипуляции числовыми данными, на третьем – скрипт для компиляции библиотеки и запуска вычислений. Очень популярный сценарий использования в научных вычислениях	Связь посредством неявной зависимости – файловой системы; Зависимость модулей на различных уровнях одновременно (файловая система и код); Тернарная зависимость языков;
Условная компиляция C кода в зависимости от команд компилятора	Используются два языка: C и Shell. На первом реализована программа, а на втором скрипт для её сборки. В зависимости от поставляемых компилятору флагов результирующая программа отличается. Очень популярный сценарий в кроссплатформенной разработке и встраиваемых системах	Зависимость GPL от DSL; Частный вариант зависимости программного кода от кода системы сборки или конфигурации;

Стоит заметить, что в данных сценариях рассмотрены не все категории языков по классификации из таблицы 3. А именно это касается функциональной парадигмы. Это связано с тем, что функциональные языки относительно непопулярны в индустрии и малоинформативны для демонстрации примеров использования. Последнее связано с сильной системой типов и типы в таких языках обычно не представлены в тексте, а выводятся компилятором. Это усложняет анализ, не показывая существенных отличий от такого же случая с императивными языками.

Программный код сценария 1, 2 и 3 отражен в приложении А, Б и В соответственно. Ниже представлен и проанализирован результат анализа данных сценариев и интерпретация полученных результатов.

4.3.2 Сценарий 1

На рисунке 1 отображена схема взаимодействия фрагментов кода.

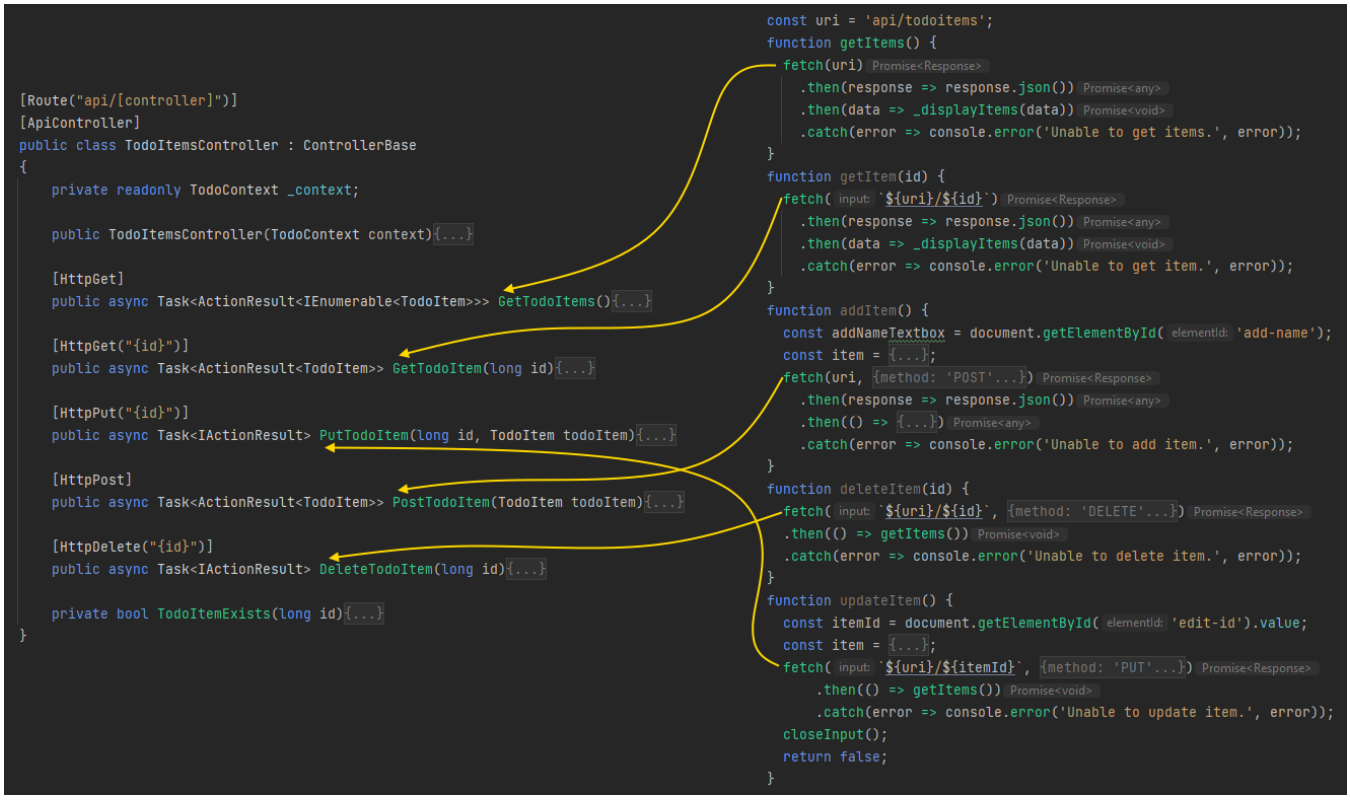

Рисунок 1 – Схема взаимодействий сценария использования 1

Здесь и далее для обозначения межфрагментных (межъязыковых) связей используется направленная стрелка желтого цвета.

Стрелка отражает семантику «зависит», и более подробная семантика зависимости задается в онтологии, используемой при проведении анализа. Таким образом, такая онтология может быть использована в том числе для интерпретации результатов анализа.

На рисунке 2 приведено текстовое представление результатов анализа.

```
fragment "/ ... /usecase1/server_controller.cs" : signature
    (GET api/TodoItems) : (→ Unit(List TodoItem))
    (GET api/TodoItems) : (→ Int Unit)
    (DELETE api/TodoItems) : (→ Int Unit)
    (PUT api/TodoItems) : (→ Int(→ TodoItem Unit))
    (POST api/TodoItems) : (→ TodoItem TodoItem)
end, environment
end =
end

fragment "/ ... /usecase1/client_fetch.js" : signature
end, environment
    (GET api/TodoItems) : (→ Int Any)
    (GET api/TodoItems) : (→ Unit Any)
    (DELETE api/TodoItems) : (→ Int Any)
    (POST api/TodoItems) : (→ Any Any)
    (PUT api/TodoItems) : (→ Int Any)
end =
end

do Http request
in "/ ... /usecase1/client_fetch.js" which need (GET api/TodoItems): (→ Int Any)
provided by "/ ... /usecase1/server_controller.cs"; with (GET api/TodoItems): (→ Int Unit)

do Http request
in "/ ... /usecase1/client_fetch.js" which need (GET api/TodoItems): (→ Unit Any)
provided by "/ ... /usecase1/server_controller.cs"; with (GET api/TodoItems): (→ Unit(List TodoItem))

do Http request
in "/ ... /usecase1/client_fetch.js" which need (DELETE api/TodoItems): (→ Int Any)
provided by "/ ... /usecase1/server_controller.cs"; with (DELETE api/TodoItems): (→ Int Unit)

do Http request
in "/ ... /usecase1/client_fetch.js" which need (POST api/TodoItems): (→ Any Any)
provided by "/ ... /usecase1/server_controller.cs"; with (POST api/TodoItems): (→ TodoItem TodoItem)

do Http request
in "/ ... /usecase1/client_fetch.js" which need (PUT api/TodoItems): (→ Int Any)
provided by "/ ... /usecase1/server_controller.cs"; with (PUT api/TodoItems): (→ Int(→ TodoItem Unit))
```

Рисунок 2 – Текстовое представление сценария использования 1

Первые два абзаца отвечают за определение фрагментов в следующем формате:

— Имя фрагмента;

— Сигнатура, состоящая из набора утверждений;

— Окружение, состоящее из набора утверждений;

— Тело модуля, идущее после знака равно.

В данном сценарии использования, тела фрагментов пусты, также, как и окружение одного и сигнатура другого. Таким образом можно утверждать, что между данными фрагментами выявлена только однонаправленные связи.

Описание таких связей представлено на рисунке 2 ниже объявления. Трактовка такого описания звучит следующим образом: «задействована

определенная семантика в окружении определенного фрагмента, которая сопровождается из сигнатуры другого фрагмента». В данном случае все связи имеют семантику «HTTP запрос».

4.3.3 Сценарий 2

На рисунке 3 отображена схема взаимодействия фрагментов кода.

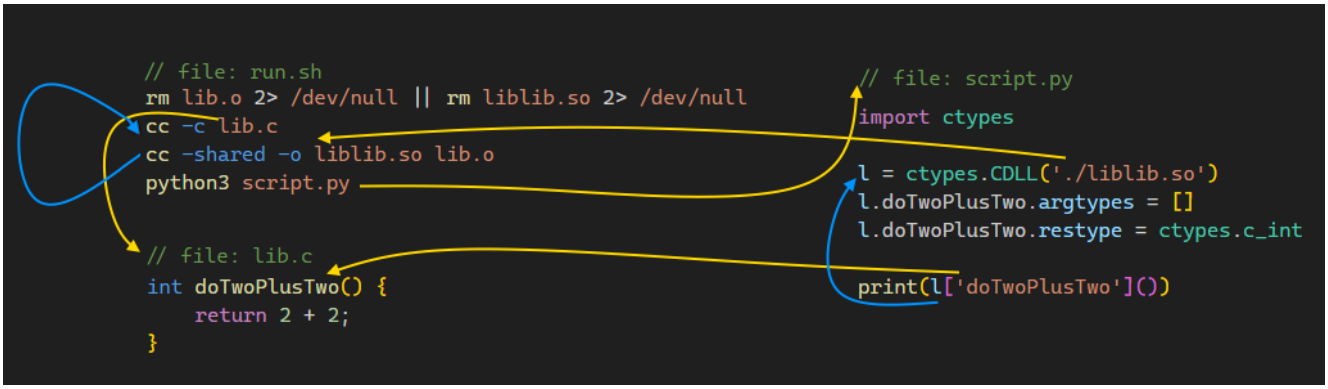

Рисунок 3 – Схема взаимодействий сценария использования 2

В данном случае желтые стрелки всё также обозначают межмодульные связи, а вот синие – внутримодульные связи. Решение отразить внутримодульные связи в данном сценарии продиктовано обозначением полноты межязыкового анализа – как видно из рисунка, такой анализ не позволяет составить полноценную сеть зависимостей, что достигается только с введением внутримодульных связей.

На рисунке 4 приведено текстовое представление результатов анализа.

```
fragment "/ ... /usecase2/script.py" : signature
    (/ ... /usecases/usecase2/script.py) : (URI)
end, environment
    (doTwoPlusTwo) : (→ Unit Any)
    (/ ... /usecases/usecase2/liblib.so) : (URI)
end =
    (/ ... /usecases/usecase2/liblib.so) ⇒ (doTwoPlusTwo)
end

fragment "/ ... /usecase2/run.sh" : signature
    (/ ... /usecases/usecase2/liblib.so) : (URI)
    (/ ... /usecases/usecase2/run.sh) : (URI)
end, environment
    (/ ... /usecases/usecase2/lib.c) : (URI)
    (/ ... /usecases/usecase2/script.py) : (URI)
end =
    (/ ... /usecases/usecase2/lib.c) ⇒ (/ ... /usecases/usecase2/liblib.so)
end

fragment "/ ... /usecase2/lib.c" : signature
    (doTwoPlusTwo) : (→ Unit Int)
    (/ ... /usecases/usecase2/lib.c) : (URI)
end, environment
end =
end

use file produced by shell command
in "/ ... /usecase2/script.py" which need (/ ... /usecases/usecase2/liblib.so): (URI)
provided by "/ ... /usecase2/run.sh"; with (/ ... /usecases/usecase2/liblib.so): (URI)

lookup file in directory
in "/ ... /usecase2/run.sh" which need (/ ... /usecases/usecase2/script.py): (URI)
provided by "/ ... /usecase2/script.py"; with (/ ... /usecases/usecase2/script.py): (URI)

call C function
in "/ ... /usecase2/script.py" which need (doTwoPlusTwo): (→ Unit Any)
provided by "/ ... /usecase2/lib.c"; with (doTwoPlusTwo): (→ Unit Int)

lookup file in directory
in "/ ... /usecase2/run.sh" which need (/ ... /usecases/usecase2/lib.c): (URI)
provided by "/ ... /usecase2/lib.c"; with (/ ... /usecases/usecase2/lib.c): (URI)
```

Рисунок 4 – Текстовое представление анализа сценария использования 2

В данном сценарии представлено три фрагмента. Первые два при этом имеют внутренние связи.

Явным отличием от сценария 1 является введение новой семантики – семантики файлов в файловой системе. Такая семантика настолько распространена, что её описание предоставляет очень полезную информацию при межъязыковом анализе. Также, в данном примере используется язык оболочки Shell, основным предметом обработки которой являются файлы.

Возможность связи файлов между собой обеспечивается в том числе фактом того, что при анализе очередного файла можно по умолчанию добавить экспорт этого файла в формируемый анализом фрагмент. Это возможно, так как файл своим существованием обеспечивает наличие такой информации и дальнейшее её использование при формировании связей.

Для обеспечения дальнейшей расширяемости в онтологию введено более общее понятие – понятие URI (Universal Resource Identifier). Оно обеспечивает описание не только файловой системы, а в целом любой сущности, идентификатор которой имеет иерархическую структуру.

За счет внутренних и внешних связей в данном сценарии использования удается выявить все семантические связи и построить достаточно полный граф для полноценного использования в дальнейшем.

4.3.4 Сценарий 3

На рисунке 5 отображена схема взаимодействия фрагментов кода.

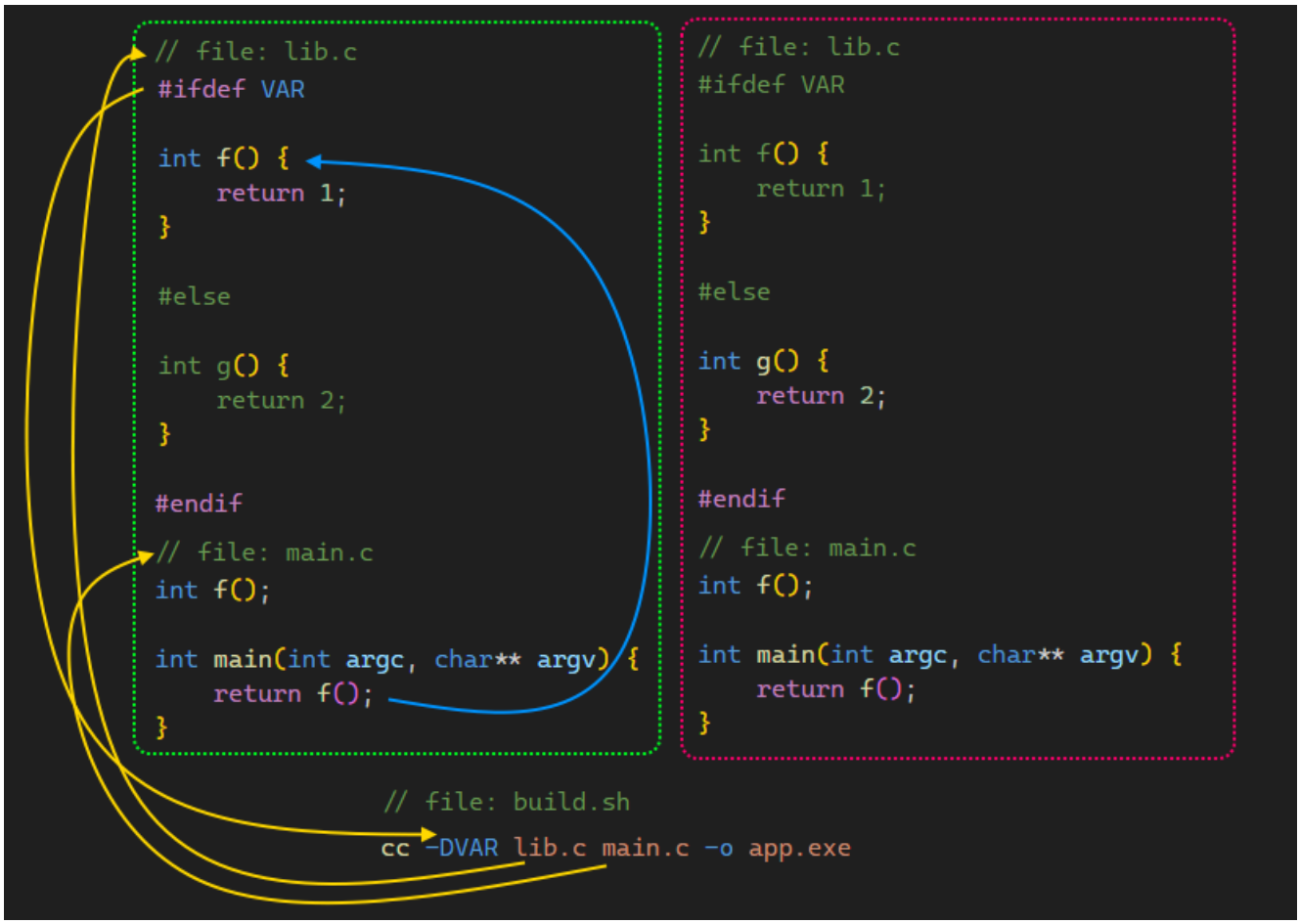

Рисунок 5 – Схема взаимодействий сценария использования 3

Предыдущие обозначения сохраняют силу и к ним добавлено новое обозначение – пунктирная рамка обозначает границы фрагмента. Это необходимо для обработки случая кодогенерации, который происходит в данном сценарии. В зависимости от значения переменной препроцессора `VAR` в кодовой базе будет представлен либо левый вариант программного кода, либо правый.

Левый фрагмент (зеленая рамка) отражает логический случай, когда переменная `VAR` объявлена, а значит функция f определена и система может быть скомпонована. В обратном случае (красная рамка) функция f не определена и вместо неё определяется функция g. В этом случае система скомпонована быть не может и выявляется ошибка.

Также, стоит обратить внимание, что в отличие от предыдущих сценариев, в данном сценарии фрагмент кода не имеет полного соответствия файлу. Фрагмент здесь представляется как способ объединения элементов кодовой базы в единую сущность, и поэтому он может задействовать куски нескольких файлов.

На рисунке 6 приведено текстовое представление результатов анализа.

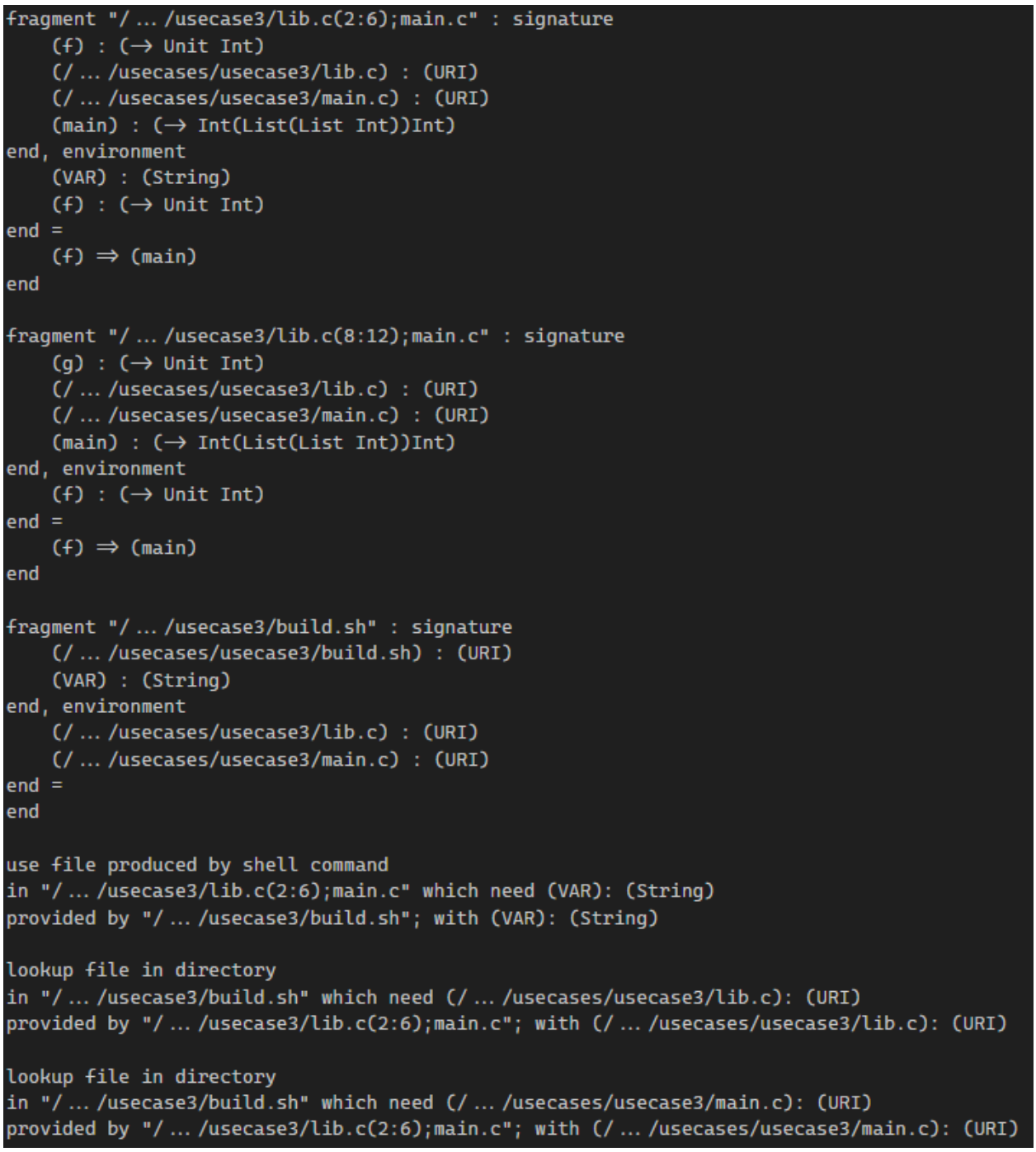

Рисунок 6 – Текстовое представление анализа сценария использования 3

Из рисунка видно, что случае первых двух фрагментов идентификатор фрагмента является составным. В обоих фрагментах задействованы два файла, причем оба фрагмента включают разные его части. Таким образом и достигается возможность анализа кодогенерации.

Также, для анализа такой кодогенерации необходимо ввести порядок связывания фрагментов. В данном сценарии первый фрагмент является более «полным» и поэтому будет обработан первее чем второй. В таком случае

обеспечивается корректная логика кодогенерации. В случае, если первый фрагмент уже связан с другим фрагментом, второй фрагмент связан не будет (так как одному окружению соответствует одна и только одна сигнатура). Такое поведение видно по описанию связей, находящееся после описания фрагментов. Ни в одной из таких связей второй модуль не задействован.

4.4 Исследовать ограничения предлагаемого метода

Данный метод, хоть и является довольно универсальным и, отчасти, формализованным, он всё же имеет ряд недостатков.

Основным его недостатком является предположение, что при внутреннем анализе семантика многих конструкций языка будет рассмотрена с точки зрения т. н. «внешнего мира». Т. е. анализатор кода на C# должен также предоставлять информацию о том, что публичные методы контроллера, обозначенные соответствующими аннотациями, представляют собой семантическую информацию об экспортируемых функциях. Такой недостаток может быть исправлен разработкой расширения для анализатора избранного языка программирования, что позволит формировать такую информацию.

Вторым ограничением метода является зависимость от результатов внутреннего анализа для сопровождения достаточной информации. Например, в случае больших фрагментов кода, имеющих несколько межфрагментных зависимостей, число внутренних зависимостей будет на порядки больше и, соответственно, основная семантическая информация будет недоступна. Это ограничение также может быть исправлено при разработке расширения для избранного языка программирования с учетом адаптации семантической информации о внутреннем анализе для дальнейшего её использования.

Третьим ограничением метода является система типов, введенная для формирования связей между фрагментами. В данный момент она слишком примитивна, несмотря на наличие тип сумм и типов произведений. Также, все номинальные являются несовместимы между собой, хотя семантически

это не всегда так. Примером является тип int в Си и тип Int в Java, семантика которых одинакова (за исключением области возможных значений). Также, для отражения сайд-эффектов требуется гораздо более сложная система типов (такая, которая позволяет описывать монадические вычисления), но её реализация требует серьезной теоретической и практической проработки для обеспечения практической полезности.

Заключительным, четвертым, недостатком метода является обеспечение подхода к условной компиляции. Формирование явного порядка связывания усложняет анализ и не покрывает все возможные случаи. Решение проблемы условной компиляции может заключаться в выполнении одного из предложенных условий:

— Анализируемый код не будет претерпевать никаких синтаксических и семантических изменений в ходе сборки системы;

— Семантическая информация о таких изменениях будет отражена в структуре фрагмента, возможно, понадобится введение параметризованных фрагментов [9].

4.5 **Рассмотреть иные варианты получения информации для работы метода**

Как видно из рассмотренных сценариев использования, большая часть семантической информации о межъязыковых связях находится в соответствующих DSL – в первую очередь это касается языков сборки и развертывания.

Таким образом, имеет смысл рассмотрения определенной области для анализа как среды программирования, имеющей свои уровни и особенности. Наиболее ярко это отражается во введении в онтологию типа `URI` для обеспечения правильного анализа сценариев 2 и 3. При мультиязыковом анализе следует рассматривать всю среду программирования – от возможностей, предоставляемых операционной системой до специфических семантических особенностей различных сопровождающих процесс

фреймворков: систем сборки, развертывания, конфигурации и кодогенерации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Четыре основные парадигмы программирования – URL: https://homes.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html.

2. Pfeiffer, RH., Wąsowski, A. (2012). Cross-Language Support Mechanisms Significantly Aid Software Development. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds) Model Driven Engineering Languages and Systems. MODELS 2012. Lecture Notes in Computer Science, vol 7590. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33666-9_12.

3. JetBrains Rider, официальная страница – URL: https://www.jetbrains.com/rider/.

4. Solarlint, официальная страница – URL: https://www.sonarsource.com/products/sonarlint/.

5. Официальная страница инструмента Mulang – URL: https://mumuki.github.io/mulang/.

6. Mayer, P., Kirsch, M. & Le, M.A. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *J Softw Eng Res Dev* **5**, 1 (2017). https://doi.org/10.1186/s40411-017-0035-z.

7. Z. Mushtaq and G. Rasool, "Multilingual source code analysis: State of the art and challenges," *2015 International Conference on Open Source Systems & Technologies (ICOSST)*, Lahore, Pakistan, 2015, pp. 170-175, doi: 10.1109/ICOSST.2015.7396422.

8. Luca Cardelli. 1997. Program fragments, linking, and modularization. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '97). Association for Computing Machinery, New York, NY, USA, 266–277. https://doi.org/10.1145/263699.263735.

9. Andrew W. Appel and David B. MacQueen. 1994. Separate compilation for Standard ML. SIGPLAN Not. 29, 6 (June 1994), 13–23. https://doi.org/10.1145/773473.178245.

ПРИЛОЖЕНИЕ А
Программный код сценария использования 1

1) Файл server_controller.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoItemsController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoItemsController(TodoContext context)
        {
            _context = context;
        }

        [HttpGet]
        public   async   Task<ActionResult<IEnumerable<TodoItem>>>
GetTodoItems()
        {
            return await _context.TodoItems.ToListAsync();
        }

        [HttpGet("{id}")]
        public         async         Task<ActionResult<TodoItem>>
GetTodoItem(long id)
        {
            var            todoItem            =            await
_context.TodoItems.FindAsync(id);

            if (todoItem == null)
            {
                return NotFound();
            }

            return todoItem;
        }

        [HttpPut("{id}")]
        public  async  Task<IActionResult>  PutTodoItem(long  id,
TodoItem todoItem)
        {
            if (id != todoItem.Id)
            {
                return BadRequest();
```

```
                }

                _context.Entry(todoItem).State                          =
EntityState.Modified;

                try
                {
                    await _context.SaveChangesAsync();
                }
                catch (DbUpdateConcurrencyException)
                {
                    if (!TodoItemExists(id))
                    {
                        return NotFound();
                    }
                    else
                    {
                        throw;
                    }
                }

                return NoContent();
            }

            [HttpPost]
            public          async          Task<ActionResult<TodoItem>>
PostTodoItem(TodoItem todoItem)
            {
                _context.TodoItems.Add(todoItem);
                await _context.SaveChangesAsync();

                //     return CreatedAtAction("GetTodoItem", new { id
= todoItem.Id }, todoItem);
                return CreatedAtAction(nameof(GetTodoItem), new { id
= todoItem.Id }, todoItem);
            }

            [HttpDelete("{id}")]
            public async Task<IActionResult> DeleteTodoItem(long id)
            {
                var            todoItem            =            await
_context.TodoItems.FindAsync(id);
                if (todoItem == null)
                {
                    return NotFound();
                }

                _context.TodoItems.Remove(todoItem);
                await _context.SaveChangesAsync();

                return NoContent();
```

```
            }

            private bool TodoItemExists(long id)
            {
                return _context.TodoItems.Any(e => e.Id == id);
            }
        }
```

2) Файл client_fetch.js

```
let todos = [];
const uri = 'api/todoitems';
function getItems() {
  fetch(uri)
    .then(response => response.json())
    .then(data => _displayItems(data))
    .catch(error   =>   console.error('Unable   to   get   items.',
error));
}
function getItem(id) {
  fetch(`${uri}/${id}`)
    .then(response => response.json())
    .then(data => _displayItems(data))
    .catch(error => console.error('Unable to get item.', error));
}
function addItem() {
  const addNameTextbox = document.getElementById('add-name');
  const item = {
    isComplete: false,
    name: addNameTextbox.value.trim()
  };
  fetch(uri, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
    .then(response => response.json())
    .then(() => {
      getItems();
      addNameTextbox.value = '';
    })
    .catch(error => console.error('Unable to add item.', error));
}
function deleteItem(id) {
  fetch(`${uri}/${id}`, {
    method: 'DELETE'
  })
  .then(() => getItems())
```

```
        .catch(error => console.error('Unable to delete item.',
error));
      }
      function updateItem() {
        const itemId = document.getElementById('edit-id').value;
        const item = {
          id: parseInt(itemId, 10),
          isComplete:                    document.getElementById('edit-
isComplete').checked,
          name: document.getElementById('edit-name').value.trim()
        };
        fetch(`${uri}/${itemId}`, {
          method: 'PUT',
          headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
          },
          body: JSON.stringify(item)
        })
            .then(() => getItems())
            .catch(error => console.error('Unable to update item.',
error));
        closeInput();
        return false;
      }

      function displayEditForm(id) {
        const item = todos.find(item => item.id === id);

        document.getElementById('edit-name').value = item.name;
        document.getElementById('edit-id').value = item.id;
        document.getElementById('edit-isComplete').checked            =
item.isComplete;
        document.getElementById('editForm').style.display = 'block';
      }


      function closeInput() {
        document.getElementById('editForm').style.display = 'none';
      }

      function _displayCount(itemCount) {
        const name = (itemCount === 1) ? 'to-do' : 'to-dos';

        document.getElementById('counter').innerText = `${itemCount}
${name}`;
      }

      function _displayItems(data) {
        const tBody = document.getElementById('todos');
        tBody.innerHTML = '';
```

```
        _displayCount(data.length);

        const button = document.createElement('button');

        data.forEach(item => {
          let isCompleteCheckbox = document.createElement('input');
          isCompleteCheckbox.type = 'checkbox';
          isCompleteCheckbox.disabled = true;
          isCompleteCheckbox.checked = item.isComplete;

          let editButton = button.cloneNode(false);
          editButton.innerText = 'Edit';
          editButton.setAttribute('onclick',
`displayEditForm(${item.id})`);

          let deleteButton = button.cloneNode(false);
          deleteButton.innerText = 'Delete';
          deleteButton.setAttribute('onclick',
`deleteItem(${item.id})`);

          let tr = tBody.insertRow();

          let td1 = tr.insertCell(0);
          td1.appendChild(isCompleteCheckbox);

          let td2 = tr.insertCell(1);
          let textNode = document.createTextNode(item.name);
          td2.appendChild(textNode);

          let td3 = tr.insertCell(2);
          td3.appendChild(editButton);

          let td4 = tr.insertCell(3);
          td4.appendChild(deleteButton);
        });

        todos = data;
      }
```

ПРИЛОЖЕНИЕ Б
Программный код сценария использования 2

1) Файл lib.c

```
int doTwoPlusTwo()
{
    return 2 + 2;
}
```

2) Файл run.sh

```
#!/bin/sh

rm lib.o 2> /dev/null || rm liblib.so 2> /dev/null
cc -c lib.c
cc -shared -o liblib.so lib.o
python3 script.py
```

3) Файл script.py

```
import ctypes
l = ctypes.CDLL('./liblib.so')
l.doTwoPlusTwo.argtypes = []
l.doTwoPlusTwo.restype = ctypes.c_int

print(l['doTwoPlusTwo']())
```

ПРИЛОЖЕНИЕ В
Программный код сценария использования 3

1) Файл build.sh

```
#!/bin/sh
cc -DVAR lib.c main.c -o app.exe#ifdef VAR
```

2) Файл lib.c

```
int f() {
    return 1;
}

#else

int g() {
    return 2;
}

#endif
```

3) Файл main.c

```
int f();

int main(int argc, char **argv)
{
    return f();
}
```