СОДЕРЖАНИЕ

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.

1.2 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.

1.3 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.

1.3.1 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.

1.3.2 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 генерацию 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.

1.3.3 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.

1.3.4 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 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.

1.3.5 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 выраже-

ний 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.

1.4 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 1.3 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 МУЛЬТИЯЗЫКОВЫХ ИСХОДНЫХ ТЕКСТОВ ПРОГРАММ

2.1 Теоретические и практические труды в отношении мультиязыкового анализа

На данный момент область мультиязыкового 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 мультиязыкового анализа. Термин «формальный язык» является более подходящим, но сильно расширяет предметную область, что затрудняет создание единой терминологической базы.

Несмотря на данные сложности, работы по мультиязыковому 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 являются 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 по-

строения 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 (англ Dependecy Call Graph) 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 мультиязыковых 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 мультиязыкового 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 мультиязыковом 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 языкозависимой 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 инфор-

мация и вовсе стирается, что делает невозможным корректный семантический анализ. Например, виртуальные методы C++ часто понижаются в отдельное определение структуры называемое *виртуальная таблица*. Таким образом, 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 мультиязыковых 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

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ПО — программное обеспечение

СУБД — система управления базами данных

ФС — файловая система

ОО — объектно-ориентированный

ВЛСА – вероятностный латентно-семантический анализ

LSP — language server protocol

IDE — integrated development invironment

AST — abstract syntax tree

DSL — domain specific language

GPL — general purpose programming language

API — application programming interface

JNI — java native interface

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. P. Mayer and A. Bauer, «An empirical analysis of the utilization of multiple programming languages in open source projects», in Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, Nanjing, China, 2015.

2. 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

3. Static Program Analysis Anders Møller and Michael I. Schwartzbach Department of Computer Science, Aarhus University. May 2023. https://cs.au.dk/~amoeller/spa/

4. Johnson, Stephen C. (25 October 1978). ”Lint, a C Program Checker”. Comp. Sci. Tech. Rep. Bell Labs: 78–1273. CiteSeerX 10.1.1.56.1841.

5. https://doc.qt.io/qt-6/metaobjects.html

6. https://react.dev/

7. 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. pp 168-184 https://doi.org/10.1007/978-3-642-33666-9_12

8. S. Latif, Z. Mushtaq, G. Rasool, F. Rustam, N. Aslam, and I. Ashraf, ‘Pragmatic evidence of cross-language link detection: A systematic literature review’, Journal of Systems and Software, vol. 206, p. 111, 2023.

9. Moonen, Leon. (2001). Generating Robust Parsers using Island Grammars.

10. J. M. Fernandes, G. H. Travassos, V. Lenarduzzi, and X. Li, Quality of Information and Communications Technology: 16th International Conference,

QUATIC 2023, Aveiro, Portugal, September 11–13, 2023, Proceedings. Springer Nature Switzerland, 2023.

11. A. Caracciolo, A. Chis, B. Spasojevic and M. Lungu, ”Pangea: A Workbench for Statically Analyzing Multi-language Software Corpora,” 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, Canada, 2014, pp. 71-76, doi: 10.1109/SCAM.2014.39.

12. O. Nierstrasz, S. Ducasse, and T. Gîrba, “The story of Moose: an agile reengineering environment,” in Proceedings of the European Software Engineering Conference (ESEC/FSE’05). New York, NY, USA: ACM Press, Sep. 2005, pp. 1–10, invited paper.

13. S. Demeyer, S. Tichelaar, and S. Ducasse, “FAMIX 2.1 — The FAMOOS Information Exchange Model,” University of Bern, Tech. Rep., 2001

14. https://tc39.es/ecma262/

15. https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html#java-native-interface-overview

16. Anne Marie Bogar, Damian M. Lyons, David Baird, ”Lightweight Call-Graph Construction for Multilingual Software Analysis,” 2018.

17. Kargar, M., Isazadeh, A., Izadkhah, H. Improving the modularization quality of heterogeneous multi-programming software systems by unifying structural and semantic concepts. J Supercomput 76, 87–121 (2020). https://doi.org/10.1007/s11227-019-02995-3

18. Zheng, W., & Hua, B. Effective Call Graph Construction for Multilingual Programs. 2023.

19. https://webassembly.org/