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», о 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, система сборки, сервер приложений или скрипты 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, обеспечивающей разные сценарии использования, является статический анализ исходного кода.

Стоит уточнить что имеется под определением «статический анализ кода». Статический анализ это прежде всего набор различных техник по извлечению информации о программе без явного её 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,
- выявление потенциальных уязвимостей,

– доказательство сохранения определенных инвариантов,
– сбор определенной статистики,
– выявление «пахнущих» фрагментов кода,
– помощь разработчику во время кодирования,
– автоматический рефакторинг кода.

Так как сценарии использования статического анализа настолько разнообразны, в рамках данной работы решено было сосредоточится на аспектах разработки, которые помогают в процессе кодирования и поддержки проекта. В качестве прикладной реализации таких аспектов выступают различные инструментальные средства. 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:

- 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 возникает проблема так как создание универсальной грамматики (для всех анализируемых языков) обычно либо сложно, либо невозможно. Таким образом остается несколько способов распознавания текста:

1. распознавание текстов по отдельности,
2. использование смешанных парсеров (например с использованием островных грамматик [9]),
3. использование единого представления.

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). Второй пункт позволяет анализировать смешанные грамматики в том числе, но сборка таких анализаторов 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 распознаются и анализируются. В некоторых случаях такой подход является наиболее удобным так как интересующая область анализа может вовлекать только ту информацию, которая получается после преобразования в унифицированное представление. В таком случае изначальное преобразование является семантически корректным этапом и позволяет упросить анализатор.

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-прослоек для проекта, который должен быть использован на конкретном встроенном устройстве. До момента размещения проекта в специализированном окружении, нет информации о том какие прослойки необходимо генерировать. 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 в угоду упрощения анализатора и повышения его универсальности.

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 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. 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 (и часто 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, запросов к СУБД).

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

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

Подводя итоги подраздела 1.3, можно сформировать следующие желаемые особенности метода мультиязыкового анализа:

1. анализ должен проводится с использованием единого представления для анализа распознанной информации,
2. желательна поддержка сценариев кодогенерации, либо возможность их реализации,

3. необходимо вовлечение операционного окружения проекта для обеспечения корректности анализа,
4. анализ должен быть универсальным для многих языков и парадигм программирования.

2 АНАЛИЗ СОСТОЯНИЯ РАБОТ В ОБЛАСТИ АНАЛИЗА МУЛЬТИЯЗЫКОВЫХ ИСХОДНЫХ ТЕКСТОВ ПРОГРАММ

2.1 Теоретические исследования в области мультиязыкового анализа

2.1.1 Анализ мультиязыковых фрагментов на основе JEE приложений

2.1.2 Универсальное решение на основе CG

2.1.3 Обзорное исследование работ по мультиязыковому анализу

2.2 Практические решения в области мультиязыкового анализа

2.2.1 MLSA

2.2.2 Mulang

2.2.3 Pangea

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ПО — программное обеспечение

СУБД — система управления базами данных

ФС — файловая система

LSP — language server protocol

IDE — integrated development invironment

AST — abstract syntax tree

DSL — domain specific language

GPL — general purpose programming language

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.