УДК 004.441

РАЗРАБОТКА МЕТОДА ДЛЯ МЕЖЪЯЗЫКОВОГО СЕМАНТИЧЕСКОГО АНАЛИЗА ТЕКСТОВ ПРОГРАММ

Орловский М.Ю.[1] (магистрант)
Научный руководитель – доцент, кандидат технических наук, Логинов И.П.[1]
1 - Университет ИТМО
email: uberdever@niuitmo.ru

Аннотация
В современной индустрии разработки программного обеспечения нередким является применение нескольких языков программирования в одном программном проекте. Такой подход порождает проблему учета согласованности модулей проекта, реализованных на разных языках. Предложенный метод статического анализа предполагает семантический анализ связей таких модулей для дальнейшего использования такого анализа в инструментальных средствах. Особенностями анализа являются гибкость объема анализа, многоязычность и учет операционного окружения проекта.

Ключевые слова
Статический анализ, мультиязыковой анализ, инструментальные средства, семантический анализ, семантические сети, среды разработки.

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. В рамках данной статьи интересно то множество технологических решений, использующее собственные языки предметной области (DSL). В качестве примеров 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. Так как используемые в проекте модули программного кода, за редким исключением, практически всегда семантически связаны между собой, то возникает необходимость поддержания согласованности таких модулей. Особенности семантических связей различных языков обычно затрудняют процесс разработки и при кодировании или рефакторинге нередки случаи нарушения согласованности. Как правило, такую несогласованность нельзя выявить до проведения процесса отладки или тестирования. Таким образом, в рамках тезиса рассматривается проблема поддержания согласованности модулей, реализованных на различных языках программирования.

Чаще всего существует возможность статического анализа различных гетерогенных модулей проекта для выявления их связей, по аналогии с тем, как это делает программист. Несмотря на это, текущие средства анализа являются недостаточными для программных проектов большого объема [1]. Также, стоит отметить проприетарную природу многих средств анализа и их неразрывную связь с другими инструментальными средствами, что затрудняет интеграцию таких анализаторов и адаптацию под конкретный проект. И несмотря на наличие открытых средств для статического мультиязыкового анализа, их применимость всё равно ограничена реализуемым стеком технологий и извлекаемой семантической информацией [2] [3].

Основная проблема мультиязыкового анализа, конечно, заключается в большом разнообразии используемых технологий, но также стоит учесть различные сценарии использования такого рода анализа. Это также влияет на сложность анализаторов и их применимость в различных программных проектах.

В рамках исследования решено было сосредоточится на наиболее полезных сценариях использования, поддерживающих разработчика в процессе кодирования. В качестве эталона

для этого был выбран Language Server Protocol [4]. Он представляет собой протокол, рассчитанный на сопряжение различных анализаторов программного кода с различными средами разработки для предоставления стандартизированного набора функций по работе с кодом. Наиболее важными сценариями протокола являются: автодополнение; поиск ссылок на идентификатор; определение идентификатора; выявление различных иерархий (например вызова функций или наследования) [5]. Так как в рамках тезиса рассмотрены только межъязыковые семантические связи, остальные функции LSP представляют малый интерес.

Исходя из сценариев использования метода можно сформировать следующие требования:

- отзывчивость (задержка анализа до трех секунд при использовании кешированных данных);
- гибкость (возможность проведения разных объемов анализа);
- универсальность (независимость от применяемых в проекте технологий);
- корректность (анализ должен производить минимальное количество ложноположительных результатов).

Учитывая предполагаемые сценарии использования, можно выделить виды необходимой информации для проведения соответствующего анализа. Вовлекаемые сценарии оперируют *идентификаторами* – уникальными (в рамках области видимости) строками, обладающими определенной *связью* с сущностями программного проекта. Например, идентификатор может быть связан с файлом (тогда он будет именем файла), с переменной (тогда он будет именем переменной) или классом (тогда он будет именем класса). Такая семантика очень хорошо кодируется через типизацию, путем присваивания идентификатору определенного *типа*. Также, очень важной семантической ролью идентификатора является его *вид использования* – это может быть либо определение идентификатора, либо ссылка на идентификатор.

Согласно данным наблюдениям, предполагается следующий вид анализа: уникальные (в рамках области видимости) идентификаторы выявляются для дальнейшего их связывания между модулями, реализованными на разных языках, по принципу определение - ссылка. Для обеспечения такого анализа необходимо решить четыре общие задачи: обеспечение анализа операционного окружения; извлечение программного кода для анализа; обеспечение уникальности идентификаторов и их областей видимости; корректная и наиболее полная типизация идентификаторов исходя из семантики языка, в котором идентификатор фигурирует.

Для анализа операционного окружения предполагается применение способа «понижения» информации об окружении с внешнего, неявного уровня на уровень проекта. Одним из самых простых способов является извлечение такой информации с последующим её кодированием в пригодную для анализа форму. Для конкретного операционного окружения возможна реализация специального модуля, анализирующего его и поставляющего такую информацию. Носитель информации технически может быть любым, однако разумным вариантом выглядит структурированный формат, например JSON. Такой модуль является *провайдером конфигурации.*

Задача по извлечению кода представляется достаточно прямолинейной – исходя из операционного окружения можно составить набор фрагментов (либо файлов) кода, которые в дальнейшем нужно будет анализировать. Задача по извлечению структуры проекта из операционного окружения ложится на провайдера конфигурации, а само извлечение будет производиться модулем *извлечения кода*.

Обеспечение уникальности идентификаторов предполагает использование специфической языконезависимой системы, позволяющей структурировать идентификаторы определенным образом с учетом их областей видимости. За прошедшие года предпринимались попытки создания такого формализма и одним из многообещающих примеров является фреймворк *графов областей видимости*, используемый, например, в Spoofax [6]. Основная идея фреймворка – создание системы, позволяющей конфигурировать языкоспецифичные правила разрешения имен и областей видимости с приведением их к

общей форме через введение *ограничений видимости*. Используя заранее заданные правила трансляции AST в такие ограничения, можно разработать транслятор любого структурированного языка. Таким образом, процесс построения графов областей вовлекает извлечение интересующего кода с последующей его трансляцией в ограничения видимости. Такой модуль является модулем *трансляции*. Стоит заметить, что для корректного связывания идентификаторов процесс трансляции должен извлекать не только информацию об ограничениях видимости, но и информацию позволяющую типизировать идентификаторы. Такой информацией может быть набор *ограничений типизации*, вводимый по аналогии с методом вывода типов, построенным на базе унификации.

Таким образом, имея информацию об идентификаторах в виде набора ограничений (видимости и типизации) возможно использовать универсальный *решатель*, принимающий ограничения на вход и делающий логический вывод исходя из них. Результатом работы решателя будет информация о типах идентификаторов, а также об их связях.

Все вышеприведенные модули (провайдер конфигурации, модуль извлечения кода, транслятор и решатель) связаны определенным протоколом взаимодействия. Такой протокол является «сердцем» метода и содержит в себе следующие компоненты:

1) компонент взаимодействия с инструментальными средствами (формат входных данных анализатора и найденных межъязыковых зависимостей);
2) компонент взаимодействия с операционным окружением (схема и носитель информации);
3) система описания межъязыковых зависимостей (возможные языковые связи, возможные виды ограничений видимости и типизации).

Будем называть такой протокол *онтологией*.

Совмещая все модули воедино, можно получить метод, который используя онтологию и данные проекта извлекает информацию о межъязыковых зависимостях в определенной форме. Следующим шагом может быть интеграция такого метода анализа в какое-либо инструментальное средство, например IDE. Для этих целей довольно хорошо подходит уже упомянутый LSP – используя его формат можно добиться создания *мультиязыкового языкового сервера*, который может быть использован в любой IDE поддерживающей LSP.

В качестве иллюстрации метода рассмотрим пример, представленный на рисунке 1.

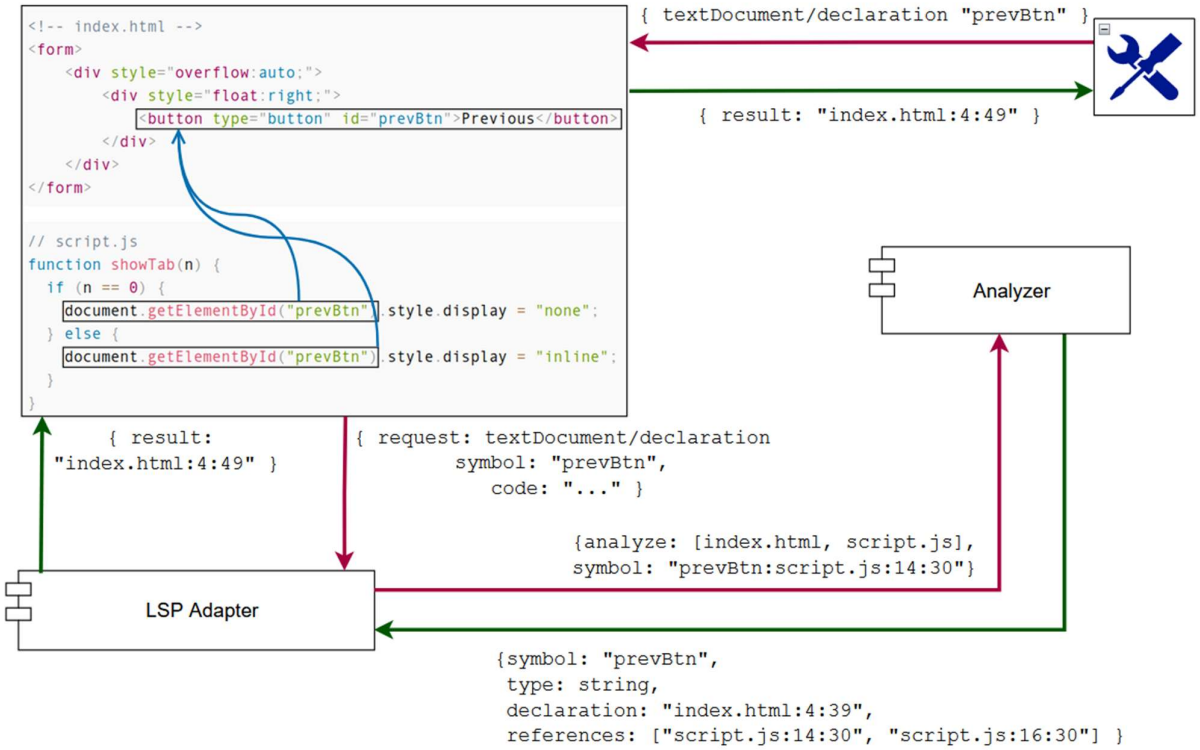

Рис. 1. Пример применения метода вместе с инструментальным средством

Здесь, операционным окружением и источником кода является проект, реализованный на двух языках – HTML и JavaScript. Соответствующие файлы проекта именованы согласно комментариям в коде. Красными стрелками на рисунке отражены запросы, зелеными соответствующие ответы. Инструмент запрашивает определение какого-либо идентификатора, имеющего связь со строковым значением “prevBtn”. Адаптер LSP конвертирует такой запрос в понятную для анализатора форму и отправляет запрос анализатору. Анализатор, согласно заранее заданной онтологии, находит определение идентификатора (в данном случае в одном экземпляре) и отправляет эту информацию для конвертации адаптеру. Адаптер преобразует эту информацию в формат LSP и сообщает инструменту, что может быть отражено в виде всплывающего окна в IDE. Выявленные связи также отражены в листинге кода на рисунке 1 синими стрелками.

Таким образом, метод соответствует заявленным требованиям и является самодостаточным решением, позволяющим реализовывать различные анализаторы межъязыковых зависимостей. В качестве дальнейшего развития данного метода предполагаются следующие направления исследования:

- использование методов машинного обучения для определения языка по входной строке, такое дополнение позволит повысить полноту анализа;
- использование формы кода, позволяющей вовлекать в анализ не только идентификаторы, но и выражения (например SSA форма [7]).

1. T. van der Storm and J. J. Vinju, “*Towards multilingual programming environments*” Sci. Comput. Program. – 2015 – С. 143–149.
2. Главный репозиторий Multilingual Static Analysis tool – [Электронный ресурс]. – Режим доступа: https://github.com/MultilingualStaticAnalysis/MLSA (дата обращения: 21.01.2024).
3. Официальный ресурс проекта Mulang – [Электронный ресурс]. – Режим доступа: https://mumuki.github.io/mulang (дата обращения: 21.01.2024).
4. Официальная страница Language Server Protocol – [Электронный ресурс]. – Режим доступа: https://microsoft.github.io/language-server-protocol/ (дата обращения: 02.02.2024).
5. N. Gunasinghe and N. Marcus, *Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools*. Apress, 2021.
6. Описание графов областей видимости в Spoofax – [Электронный ресурс]. – Режим доступа: https://spoofax.dev/references/statix/scope-graphs/ (дата обращения: 02.02.2024).
7. R. A. Kelsey, ‘A correspondence between continuation passing style and static single assignment form’, in *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, San Francisco, California, USA, 1995, pp. 13–22.