

Национальный исследовательский университет ИТМО
(Университет ИТМО)

На правах рукописи

Кореньков Юрий Дмитриевич

**Методы и средства анализа
исходных текстов программ и программных систем
на основе семантических моделей**

Специальность 05.13.11

«Математическое и программное обеспечение вычислительных машин, комплексов и
компьютерных сетей»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:

кандидат технических наук

Дергачев Андрей Михайлович

Санкт-Петербург 2020

Оглавление

РЕФЕРАТ.....	7
SYNOPSIS.....	35
ВВЕДЕНИЕ.....	60
ГЛАВА 1 Задачи и проблемы реализации анализа исходных текстов программ.....	68
1.1 Задачи семантического анализа и семантические модели программ	68
1.2 Исследование существующих подходов к семантическому анализу текстов программ	75
1.3 Исследование особенностей реализации анализаторов в составе интегрированных сред разработки	82
ГЛАВА 2 Метод предметно-ориентированного анализа исходных текстов программ.....	95
2.1 Основные термины, понятия и определения.....	95
2.2 Формализация метода анализа текстов программ	97
2.3 Область применения и требования к реализации	111
ГЛАВА 3 Язык описания семантических моделей программ	123
3.1 Описание синтаксических моделей программ	123
3.2 Описание семантических моделей программ.....	129
3.3 Описание семантических трансляций	133
ГЛАВА 4 Алгоритм итеративного преобразования семантических моделей программ	137
4.1 Разработка алгоритма и структур данных	137
4.2 Интеграция решения в среды разработки.....	143
4.3 Апробация разработанного решения	147
Заключение.....	157
Список сокращений и условных обозначений	159
Список литературы.....	160

Приложение А – расширенная грамматика спецификаций синтаксических моделей.....	169
Приложение Б – грамматика спецификаций семантических моделей ..	181
Приложение В – акты о внедрении.....	184
Приложение Г – тексты публикаций.....	187

РЕФЕРАТ**ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ**

Актуальность темы. В области разработки программ и программных систем большое значение имеет инструментальное программное обеспечение, предназначенное для поддержки процесса разработки. Основным инструментом является интегрированная среда разработки (Integrated Development Environment, IDE) – комплекс программ, предоставляющий разработчику средства редактирования исходного кода, контроля версий, отладки, трансляции, организации командной работы и многие другие возможности.

Одной из основных функций интегрированных сред разработки является редактирование исходного кода программ. Современные текстовые редакторы способны частично автоматизировать процесс разработки, что существенно повышает производительность труда программиста. Возможности текстовых редакторов позволяют выделять цветом синтаксические элементы (т.н. «подсветка синтаксиса» от англ. «syntax highlight»), уведомлять программиста об ошибках, автоматически дополнять синтаксические конструкции при наборе, выполнять навигацию по исходному коду, визуализировать структурную организацию программы.

Особенностью современной индустрии разработки программного обеспечения является применение нескольких языков программирования в одном программном проекте. Данная особенность порождает проблему проверки согласованности компонентов, реализованных на разных языках и применяемых в составе одной программной системы. Как правило, несогласованность компонентов отслеживается только на этапе отладки или тестирования. Например, если в программе на языке C# используется функция, реализованная в отдельном модуле на языке Си, и сигнатура этой функции изменилась, то отследить возникшую вследствие этого ошибку в

программной системе возможно только во время выполнения. Такое состояния дел существенно снижает эффективность процесса разработки.

Данная работа направлена на развитие подходов в области анализа исходного кода программ, написанных на нескольких языках программирования, с целью выявления подобных ошибок на этапе кодирования, что сократит количество циклов редактирования-отладки-тестирования и в целом положительно повлияет на процесс разработки программного продукта.

Степень разработанности проблемы. Большой вклад в теорию формальных языков и грамматик внесли Ноам Хомский, Стивен Клини, Масару Томита, Джей Эрли, Вон Пратт; в развитие реляционных моделей и отображений – Эдгар Кодд, Кристофер Дейт; в развитие семантических технологий и метамоделирование – Йорик Уилкс.

Практический вклад в область инструментального программного обеспечения вносят компании Microsoft, Oracle, IBM, JetBrains и другие. Передовыми решениями в данной области являются среды разработки предметно-ориентированных языков, такие как JetBrains MPS, Eclipse Xtext, которые не решают проблемы в полной мере и обладают рядом недостатков:

- 1) не позволяют выполнять анализ согласованности компонентов программ, написанных с использованием разных языков программирования;
- 2) не имеют возможности выполнения семантического анализа текстов программ во время редактирования;
- 3) применимы только в составе разработанной под них среды разработки (например, Racket, Kermeta, Eclipse Xtext);
- 4) не выполняют анализ исходного кода, написанного на алгоритмических языках программирования (например, JetBrains MPS).

Объект исследования – исходные тексты и семантические модели программ, написанных с использованием нескольких языков программирования.

Предмет исследования – методы и алгоритмы анализа исходных текстов программ в процессе их редактирования с использованием семантических моделей.

Цель диссертационной работы – выявление ошибок в программном коде, написанном с использованием нескольких языков программирования, на этапе редактирования исходных текстов программ.

В соответствии с целью в диссертационной работе ставятся и решаются следующие **задачи**.

1. Анализ способов внутреннего представления и обработки редактируемого текста программ в интегрированных средах разработки с целью выявления недостатков существующих средств поддержки мультиязыковых программных проектов.
2. Разработка метода предметно-ориентированного анализа исходных текстов программ в процессе редактирования на основе семантических моделей.
3. Разработка программной архитектуры и структур данных для организации хранения и обработки семантической информации в процессе редактирования и анализа исходных текстов программ.
4. Разработка и реализация программного средства анализа исходных текстов программ, апробация разработанного метода.

Методы исследования. При решении поставленных задач применялись методы объектно-ориентированного анализа, методы синтаксического и семантического анализа, метамоделирование, методы языково-ориентированного программирования.

Научная новизна работы. В диссертации получены следующие результаты, характеризующиеся научной новизной.

1. Метод предметно-ориентированного анализа исходных текстов программ, отличающийся от существующих возможностью выполнять анализ текстов программ, разрабатываемых с применением нескольких языков программирования.
2. Язык описания семантических моделей программ, концептуально отличающийся от существующих совместным применением реляционного представления и представления в виде графа, позволяющий адаптировать алгоритмы анализа семантических моделей программ к специфике конкретных программных проектов.
3. Алгоритм итеративных преобразований семантических моделей программ, использующий модифицированные структуры данных для внутреннего представления текста в ходе его редактирования и анализа, что позволяет уменьшить время отклика среды разработки или текстового редактора между вводом пользователя и отображением результатов анализа актуального состояния программного кода.

Практическая значимость работы.

1. Программная архитектура для представления динамических семантических моделей программ, позволяющая интегрировать реализацию разработанного метода предметно-ориентированного анализа в различные среды разработки или текстовые редакторы.
2. Анализатор исходных текстов программ, реализованный в виде программного компонента, допускающего его интеграцию с существующими средами разработки, прошедший апробацию в составе реализованного настраиваемого текстового редактора при решении практической задачи анализа текстов программ, разрабатываемых с использованием нескольких языков программирования.

Теоретическая значимость работы состоит в развитии теории формальных языков и грамматик в части согласованного анализа текстов

программ, написанных с использованием нескольких языков программирования.

Основные положения, выносимые на защиту.

1. Метод предметно-ориентированного анализа исходных текстов программ, разрабатываемых с применением нескольких языков программирования.
2. Язык описания семантических моделей программ, позволяющий настраивать алгоритмы анализа текста под конкретные программные проекты.
3. Алгоритм реактивного анализа и итеративного преобразования семантических моделей программ.

Апробация результатов исследования. Основные положения диссертационной работы и результаты исследований докладывались на одиннадцати всероссийских и международных конференциях. Среди них VIII, IX конгресс молодых ученых Университета ИТМО (КМУ) (2019, 2020 гг.), XLVI – XLIX научная и учебно-методическая конференция Университета ИТМО (2017-2020 гг.), IX Научно-практическая конференция молодых ученых «Вычислительные системы и сети (Майоровские чтения 2017)», 17th FRUCT Conference.

Обоснованность и достоверность научных положений и результатов обеспечены полнотой анализа теоретических и практических исследований, докладами и обсуждениями на конференциях и научных конгрессах, практической проверкой и внедрением полученных результатов.

Внедрение результатов исследования. Результаты исследования использованы в рамках выполнения НИР-ФУНД № 619296 «Разработка методов создания и внедрения киберфизических систем», внедрены в учебный процесс на факультете программной инженерии и компьютерной техники Университета ИТМО, прошли апробацию в ООО "Тюн-ит" при решении практической задачи анализа текстов программ, разрабатываемых с

использованием нескольких языков программирования, на наличие потенциальных уязвимостей, выявляемых на уровне исходного кода, что подтверждается актами о внедрении.

Публикации результатов исследования. По теме диссертации опубликовано семь основных работ, из них три статьи в журналах из перечня рецензируемых научных изданий, в которых должны быть опубликованы основные научные результаты диссертаций на соискание ученой степени кандидата и доктора наук, и две статьи в изданиях, индексируемых Scopus или Web of Science.

Личный вклад. Основные результаты, представленные в диссертации, получены лично автором. Из работ, выполненных в соавторстве, в диссертацию включены результаты, имеющие личный вклад автора по теме диссертации или смежным темам, таким как синтаксический анализ. Соавтором Дергачевым А.М. осуществлялось общее руководство исследованием, а также руководство работами по технической реализации предложенных средств и методов.

1. В работе «Исследование и разработка языкового инструментария на основе PEG-грамматики» автором полностью выполнена разработка программного расширения IDE Visual Studio, включающего модифицированный текстовый редактор AvalonEdit, выполняющий подсветку синтаксиса на основе разработанного синтаксического анализатора и динамически изменяемых грамматик. Логиновым И.П. осуществлялось тестирование предложенного языкового инструментария и разработка грамматик тестовых языков. Лаздиным А.В. осуществлялось сравнение разработанной функциональности с существующими аналогами.
2. В работе «Declarative target architecture definition for data-driven development toolchain» автором разработан синтаксический и семантический анализатор для языка описания систем команд,

предложенного Логиновым И.П. Соавтором Лаздиным А.В. осуществлялось исследование возможностей описания архитектур процессоров средствами компиляторов GCC и LLVM.

3. В работе «Application of Graph Databases for Static Code Analysis of Web-Applications» автором реализован анализ синтаксического дерева на предмет использования определенной функциональности, приводящей к эксплуатации уязвимостей веб-приложений. Соавтор Садырин Д.С. выполнял обзор средств, применяемых в статическом анализе кода. Логинов И.П. реализовал генерацию отчетов с результатами анализа. Ильина А.Г. выполняла исследование графовых поисковых запросов в терминах формальных языков.
4. В работе «Исследование и реализация алгоритма синтаксического анализа» автором предложено использование алгоритма «Марга» для построения адаптируемых синтаксических анализаторов и создан прототип его реализации для среды на базе Common Language Infrastructure. Павлова Е.В. осуществляла программную реализацию синтаксического анализатора на основе алгоритма «Марга». Логинов И.П. выполнял разработку тестовых предметно-ориентированных языков.
5. В работе «Итеративное преобразование семантических моделей программ» автором предложен метод итеративного преобразования семантических моделей программ, применимый для предметно-ориентированного анализа исходных текстов программ, разработан алгоритм реактивного анализа и итеративных преобразований семантических моделей программ, выполнена экспериментальная программная реализации.
6. В работе «Метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей» автором предложен язык описания спецификаций семантических моделей

программ и их отображений, а также описания синтаксических моделей исходных текстов программ для построения абстрактных семантических графов в ходе семантического анализа текстов программ.

7. В работе «Методы и средства обнаружения уязвимостей аллокаторов динамической памяти библиотеки glibc» автором выполнен анализ характера уязвимостей и исследование возможности их выявления на уровне написания исходного кода программ, разрабатываемых с применением языков C/C++ и ассемблера. Ильина А.Г. выполняла обзор методов верификации программ для проверки корректности во время их выполнения. Садырин Д.С. описал подход, позволяющий совместить символьное выполнение программы с реальным исполнением процессором ее машинного кода. Логинов И.П. выполнял практическое исследование особенностей средств анализа кода и механизмов динамического распределения памяти.

Соответствие диссертации паспорту научной специальности.

Диссертационная работа соответствует паспорту научной специальности 05.13.11 «Математическое обеспечение вычислительных машин, комплексов и компьютерных сетей», а проведенное исследование – формуле специальности. Исследование соответствует следующим пунктам паспорта специальности.

1. Модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования.
2. Языки программирования и системы программирования, семантика программ.

Объём и структура работы.

Диссертационная работа изложена на 127 страницах, состоит из введения, четырёх глав, содержащих 19 рисунков, четыре таблицы, и заключения. Список литературы включает 100 наименования.

СОДЕРЖАНИЕ РАБОТЫ

Во введении обоснована актуальность темы диссертационной работы, дана общая характеристика работы, определены объект и предмет исследования, сформулированы цели, задачи и методы исследования, представлена новизна, показана научная и практическая значимость, приведены основные результаты работы.

В первой главе исследуется предметная область – существующие подходы к анализу текстов программ, особенности реализации интегрированных сред разработки и анализаторов исходного кода программ.

Область разработки трансляторов программ в целом и анализаторов исходных текстов в частности является хорошо проработанной. Основная задача, которую решают анализаторы исходных текстов – проверка корректности программ с точки зрения синтаксиса и семантики, а также подготовка программы для дальнейшей трансляции в машинно-ориентированное представление (т.е. для синтеза машинного кода). Аспекты, связанные с синтезом кода, в данной работе не рассматриваются.

Анализ исходных текстов программ выполняется в несколько этапов. Прежде всего выполняется лексический анализ (сканирование) – процедура выделения элементов языка программирования, называемых лексемами, из текста программы. Результатом сканирования является множество токенов – лексем, которым сопоставлены категории языковых конструкций, такие как «ключевое слово», «символ-разделитель», «идентификатор», «литерал», «комментарий». Это множество служит входными данными для синтаксического анализатора. Синтаксический анализатор реализует следующий этап обработки текста программы, на котором выполняется проверка соответствия текста программы определенным правилам, совокупность которых называется грамматикой языка. Грамматика языка задает множество слов и символов, которые могут использоваться для записи программы, а также их возможные комбинации, позволяющие считать текст

правильно структурированным. Результатом синтаксического анализа является дерево разбора, отражающее синтаксическую структуру программы. Узлами дерева являются объявления типов, подпрограмм, отдельные операторы и т.д. Это дерево содержит данные обо всех элементах программы, записанных в исходном тексте, и их взаимодействии.

После проверки корректности программы в части синтаксиса, происходит ее дальнейшее преобразование с целью получения машинного кода. Однако в рамках данной работы интерес представляет анализ с целью реализации средств редактирования текста программ, а не последующей трансляции. Анализаторы, применяемые в интегрированных средах разработки, предназначены для облегчения восприятия текста программы, а также для выявления ошибок. Некоторые функциональные возможности могут быть реализованы на основе результатов синтаксического анализа, например, подсветка синтаксиса.

В ряде случаев требуется выполнять анализ текста с учетом контекста, в котором используется та или иная конструкция языка программирования. Например, в языке C# существует встроенный язык запросов к данным, называемый LINQ. Подсветка ключевых слов данного языка необходима только в составе выражений, описывающих запрос, а при использовании этих слов вне описаний запросов подсветка синтаксиса не должна выполняться. Для учета контекста, в котором используются различные языковые конструкции, а также для реализации проверки согласованности компонентов программ, реализованных на разных языках программирования, необходима реализация механизмов семантического анализа. Такие механизмы должны учитывать особенности типов данных, применяемых при написании текста программы в части допустимости выполнения записанных операций, в том числе, вызовы подпрограмм, а также области видимости типов и подпрограмм, доступность переменных. На уровне взаимодействия компонентов важно

устанавливать соответствие между импортируемой и экспортируемой функциональностью.

Проблема существующих сред разработки заключается в том, что для каждого языка программирования реализован отдельный анализатор, не обеспечивающий возможности взаимной интеграции или обмена данными с анализаторами для других языков программирования. Во многом это обусловлено различиями в синтаксисе и семантике языков программирования, из которых следует отсутствие возможности сопоставления результатов анализа текстов программ друг другу. На рисунке 1 приведена типовая схема реализации поддержки языков программирования в интегрированных средах разработки в отношении семантического анализа текстов программ.



Рисунок 1 – Типовая схема расширяемости интегрированных сред разработки в отношении семантического анализа текстов программ

Из рисунка видно, что при добавлении очередного языка реализация анализатора существенно усложняется за счет необходимости создания модулей сопряжения пар языков.

В ходе исследования данной проблемы был выполнен анализ способов внутреннего представления редактируемого текста программ, структуры программ, а также исследование недостатков существующих средств

поддержки мультиязыковой разработки. Способы представления редактируемого текста программы важны, так как оказывают влияние на время отклика пользовательского интерфейса в ответ на изменение текста. Например, подсветка ключевого слова после завершения его набора должна происходить мгновенно для программиста, чтобы обеспечить удобство использования. Описанный пример является типичной проблемой в использовании сред разработки.

На основе результатов анализа существующих решений и их недостатков сформулирован ряд задач, последовательно решаемых для достижения поставленной в диссертационной работе цели.

Первая задача заключается в разработке метода предметно-ориентированного анализа исходных текстов программ. Метод основан на применении предметно-ориентированного языка для описания элементов программ. Данный язык является метаязыком по отношению к языкам программирования и позволяет строить промежуточное представление программы с заданными классами атрибутов, общими для программ, разрабатываемых с использованием нескольких языках программирования. На основании данных атрибутов выполняется проверка соответствия семантических моделей программ. В данной работе под семантической моделью программы понимается промежуточное представление, описывающее свойства и связи ее элементов.

Следующая задача – реализация алгоритма анализа семантических моделей программ. Существующие способы представления семантических моделей программ и интеграции анализаторов, входящих в состав сред разработки, не позволяют использовать их для предметно-ориентированного анализа. Следовательно, необходима разработка альтернативной программной архитектуры и структур данных, позволяющих интегрировать реализацию разработанного метода предметно-ориентированного анализа в различные среды разработки или текстовые редакторы.

Заключительный этап работы – апробация разработанного метода и средств его адаптации в процессе разработки реальных программных продуктов. Создание программного средства на базе разработанных методов и программных архитектур необходимо, чтобы показать его работоспособность и практическую применимость. Такой анализатор также позволит оценить эффективность в сравнении с существующими средствами для сценариев, под которые все-таки существуют специализированные средства анализа текстов программ.

Таким образом, не существует универсального инструментального средства, обеспечивающего согласованную разработку программ с использованием нескольких языков программирования. Существующие специализированные средства имеют ряд ограничений по расширению функциональных возможностей в силу специфики программной реализации. Для решения проблемы необходимо разработать метод предметно-ориентированного анализа и средство адаптации его программной реализации под задачи отдельных проектов.

Во второй главе представлен разработанный метод предметно-ориентированного анализа текстов программ. Основная идея метода заключается в использовании настраиваемых процедур динамической трансляции и итеративных преобразований семантических моделей программ по результатам анализа. То есть, в зависимости от набора языков программирования, применяемых при разработке программных систем, создаются спецификации семантической трансляции отдельных аспектов исходного текста в метамодель, общую для этих языков. Предлагаемый метод отличается от существующих тем, что для представления семантической модели программы нет необходимости заранее создавать специализированные анализаторы. Используется представление модели программы на базе семантической сети, являющейся метамоделью по отношению к грамматикам языков, задействованных в разработке конкретного программного проекта.

Семантические модели формируются на основе спецификаций, которые формально можно представить записью вида:

$$S = (T_s, P_s, V_s, C_s, D_s), \quad (1)$$

где $t_s \in T_s$ – идентификаторы типов, $p_s \in P_s$ – идентификаторы свойств, V_s – схемы значений, записываемые в виде:

$$V_s = \{v_s = (t, q): t \in T_s \cup T_0, \quad (2)$$

$$q = \begin{cases} \text{false для одиночных значений} \\ \text{true для наборов значений} \end{cases},$$

при $v_s = (t, q)$ – пара, задающая способ представления значения атрибута в семантической модели (t – тип значения, q – его кратность); $C_s = \{c_s = t_s \rightarrow P_{t_s}: P_{t_s} \subseteq P_s\}$ – принадлежность свойств типам, записываемая в виде отображений идентификаторов типов t_s на набор свойств P_{t_s} ; $D_s = \{d_s = (t_s, p_s) \rightarrow v_s\}$ – схемы типов, описываемые в виде отображений пар вида тип-свойство (t_s, p_s) на схему значения v_s ; T_0 – встроенные литеральные типы значений.

Формальное представление семантической модели программы можно записать в виде:

$$M = (S_m, N_m, A_m, V_m, E_m, F_m), \quad (3)$$

где: S_m – спецификация семантической модели, $n_m \in N_m$ – идентификаторы элементов, $a_m \in A_m$ – идентификаторы атрибутов, V_m – значения атрибутов, описываемые в виде пар "схема"- "набор литеральных значений или идентификаторов элементов":

$$V_m = \{v_m = (v_s, X_{v_m}): v_s \in V_s, v_s = (t, q), \quad (4)$$

$$|X_{v_m}| \in \begin{cases} \{0,1\}, & \text{если } q = \text{false,} \\ \{\mathbb{N}^0\}, & \text{если } q = \text{true,} \end{cases}$$

$$X_{v_m} = \left\{ x_{v_m} : \begin{cases} x_{v_m} = \{\text{literal value}\}, \text{если } t \in T_0 \\ x_{v_m} \in N_m, \text{если } t \in T_s \end{cases} \right\}$$

};

$E_m = \{e_m = n_m \rightarrow (t_{n_m}, P_{n_m})\}$ – элементы, описываемые в виде отображений идентификаторов элементов на пары вида "тип"- "набор отношений", заданных парами свойство-атрибут (элементу сопоставляется тип и набор атрибутов, соответствующих свойствам этого типа); $F_m = \{f_m = (n_m, p_s) \rightarrow (a_{n_m}, v_{n_m})\}$ – поля элементов, описываемые в виде отображений пар узел-свойство на пары атрибут-значение (свойству элемента сопоставляется атрибут и ассоциированное с ним значение):

Спецификация синтаксической модели – это частный случай спецификации семантической модели (1), расширенной контекстно-свободной грамматикой:

$$S = (T_s, P_s, V_s, C_s, D_s, G_s), \quad (5)$$

где компоненты грамматики $G_s = (\Sigma_{G_s}, N_{G_s}, P_{G_s}, S_{G_s})$ следующие: Σ_{G_s} – набор терминальных символов, $n_{G_s} \in N_{G_s}$ – набор нетерминальных символов, P_{G_s} – набор правил вида $n_{G_s} \rightarrow w$, где n_{G_s} – нетерминал, w – любая последовательность терминалов и нетерминалов, S_{G_s} – стартовый нетерминал.

Тогда синтаксическая модель – частный случай семантической модели со спецификацией синтаксической модели в качестве S_m (3).

Формальное представление спецификации семантической трансляции:

$$R_{1 \rightarrow 2} = (S_1, S_2, Q_{1 \rightarrow 2}, P_{1 \rightarrow 2}, I_{1 \rightarrow 2}), \quad (6)$$

где S_1 – спецификация исходной семантической модели, S_2 – спецификация целевой семантической модели, $Q_{1 \rightarrow 2}$ – набор контекстных отображений, $P_{1 \rightarrow 2}$ – набор свободных отображений, I – набор начальных отображений для исходной модели m_{s1} и целевой m_{s2} :

$$\begin{aligned}
 Q_{1 \rightarrow 2} &= \{q_{t_{s1} \rightarrow t_{s2}} = (n_{s1}, m_{s1}, q') \rightarrow m'_{s2}\} \\
 P_{1 \rightarrow 2} &= \{p_{t_{s1} \rightarrow t_{s2}} = (n_{s1}, m_{s1}) \rightarrow m'_{s2}\} \\
 I_{1 \rightarrow 2} &= \{\iota_{t_{s1} \rightarrow t_{s2}} = m_{s1} \rightarrow m'_{s2}\}
 \end{aligned} \tag{7}$$

где каждая из частных трансляций q , p , и ι , описанных функциями $f_{t_{s1} \rightarrow t_{s2}}$, задает отображение элемента типа t_{s1} исходной модели m_{s1} в элемент типа t_{s2} целевой модели m_{s2} ,

Начальные отображения $\iota_{t_{s1} \rightarrow t_{s2}}$ задают корневые трансляции, с которых начинается формирование зависимостей между элементами моделей. Они могут включать обращения к свободным отображениям $p_{t_{s1} \rightarrow t_{s2}}$ и контекстным отображениям $q_{t_{s1} \rightarrow t_{s2}}$, задающим прямые зависимости между одним исходным элементом и одним целевым без дополнительных аргументов и с учетом дополнительных аргументов q' соответственно. Каждое отображение связывает между собой структурные элементы моделей m_{s1} и m_{s2} посредством реляционных отображений: начальные отображения, выступающие в качестве источников информации для инициации алгоритма, выполняющего семантическую трансляцию; свободные отображения, применяющиеся к конкретным элементам исходной модели; контекстные отображения, использующиеся для выполнения последующих шагов алгоритма семантической трансляции, инициируемых косвенно при выполнении предшествующих шагов. В качестве узла аргумента и контекста при этом будут использоваться данные, полученные на предшествующих шагах, например, при применении первичных отображений.

В формализованном виде разработанный метод можно записать следующим образом.

- Пусть множество Ψ – тексты τ программной системы, написанные на языках с синтаксическими спецификациями S_τ :

$$(\tau_n, S_{\tau_n}) \in \Psi, 1 \leq n \leq |\Psi|, \tag{8}$$

где S_{τ_n} – синтаксическая спецификация текста программы τ_n , n – номер текста τ программной системы. Тогда результатом синтаксического анализа этих текстов будет множество синтаксических моделей Υ :

$$\Upsilon = \{v_{S_\tau} \forall (\tau, S_\tau) \in \Psi\}, |\Psi| = |\Upsilon|, \quad (9)$$

где множество Ψ – тексты τ программной системы, v_{S_τ} – синтаксическая модель по спецификации S_τ для текста программы τ .

2. Пусть Θ – предметные области, в рамках которых выполняется анализ текстов программной системы, заданные набором спецификаций семантических трансляций:

$$\Theta = \{R_n = (S_{1n}, S_{2n}, \dots)\}, 1 \leq n \leq |\Theta|, \quad (10)$$

где R_n – спецификация семантической трансляции, S_{1n} – спецификация исходной семантической модели, S_{2n} – спецификация целевой семантической модели.

3. Предметно-ориентированный анализ на основе семантических моделей для текста τ даёт набор семантических моделей Ω_τ , являющийся набором всех семантических моделей, полученных в результате трансляций $f(m, r)$, начиная с синтаксической модели v_{S_τ} , прямо или косвенно для данной семантической модели $m = (S, \dots, \dots, \dots)$ при наличии подходящей спецификации семантической трансляции $r = (S, \dots, \dots, \dots) \in \Theta$ для спецификации семантической модели S :

$$\Omega_\tau = \bigcup_{\substack{m_0 = v_{S_\tau} \\ m_n = (S, \dots, \dots, \dots) \\ \exists r_{n \rightarrow n+1} = (S, \dots, \dots) \in \Theta}} \{m_n\}, m_{n+1} = f(m_n, r_{n \rightarrow n+1}) \quad (11)$$

где m_n – семантическая модель (3), $r_{n \rightarrow n+1}$ – спецификация семантической трансляции (6), $f(m_n, r_{n \rightarrow n+1})$ – функция, осуществляющая трансляцию:

$$\begin{aligned}
f(m_{S_a}, r_{a \rightarrow b}) &= \varsigma(S_b, \{m' = \iota(m_{S_a}) \forall \iota \in I, \\
m_{S_a} &= (S_a, \dots, \dots) \\
r_{a \rightarrow b} &= (S_a, S_b, Q, P, I)\}),
\end{aligned} \tag{12}$$

где $r_{a \rightarrow b}$ – спецификация семантической трансляции (6) из семантической модели m_{S_a} по спецификации S_a в семантическую модель по спецификации S_b , m' – частичная целевая модель, получающаяся в результате применения трансляции $\iota \in I$ из спецификации $r_{a \rightarrow b}$ к модели m_{S_a} , ς – функция, объединяющая набор семантических моделей m'_Σ с общей спецификацией S в одну общую модель:

$$\begin{aligned}
\varsigma &= (S, m'_\Sigma) \rightarrow (S, \\
N_b &= \varsigma'(m'_\Sigma, (S, N_{m'_\Sigma}, \dots) \rightarrow N_{m'_\Sigma}), \\
A_b &= \varsigma'(m'_\Sigma, (S, A_{m'_\Sigma}, \dots) \rightarrow A_{m'_\Sigma}), \\
V_b &= \varsigma'(m'_\Sigma, (S, V_{m'_\Sigma}, \dots) \rightarrow V_{m'_\Sigma}), \\
E_b &= \varsigma'(m'_\Sigma, (S, E_{m'_\Sigma}, \dots) \rightarrow E_{m'_\Sigma}), \\
F_b &= \varsigma'(m'_\Sigma, (S, F_{m'_\Sigma}, \dots) \rightarrow F_{m'_\Sigma}) \\
),
\end{aligned} \tag{13}$$

где $(S, N_b, A_b, V_b, E_b, F_b)$ – семантическая модель (3): ς' – функция, объединяющая результаты применения функции, заданной аргументом z , к каждому из членов множества, заданного аргументом Y :

$$\varsigma' = (Y, z) \rightarrow \bigcup z(y) \forall y \in Y \tag{14}$$

4. M_Γ – все семантические модели (9) для всех текстов Y (8) (первичные синтаксические и полученные из них трансляциями в совокупности):

$$M_\Gamma = \bigcup \Omega_\tau \forall v_{S_\tau} \in Y, \tag{15}$$

а Γ – итоговый набор объединённых семантических моделей:

$$\Gamma = \{\gamma_S = \varsigma(s, X) : \forall x = (s, \dots, \dots) \in X \subseteq M_\Gamma\}, \tag{16}$$

состоящий из объединённых функцией ς (13) семантических моделей x (3) по каждой из доступных спецификаций s (1).

Спецификации семантических моделей могут быть дополнены предикатами, заданными в любой из интерпретаций семантических моделей. Построение набора Γ не требует выполнения всех ограничений и предикатов семантических и синтаксических моделей, так как они описывают формальные зависимости между элементами данных и их членами. По удовлетворённости этих ограничений и предикатов разработчик программного обеспечения может судить о соответствии текстов программной системы данным спецификациям.

Применение предложенного метода основано на использовании предметно-ориентированного языка. Данный язык предназначен для описания спецификаций моделей программ и представляет собой систему понятий, сочетающую элементы реляционных отображений, представления знаний на основе семантических сетей и статической типизации. Это позволяет:

- 1) снизить порог вхождения разработчиков, составляющих спецификации для анализа текстов программ, благодаря широко известным концепциям объектно-ориентированного анализа и реляционных преобразований в основе;
- 2) упростить выявление ошибок в спецификациях текстов и семантических моделях программ за счет статической типизации.

На рисунке 2 приведена обобщенная схема взаимодействия текстового редактора и анализатора. Преобразование представлений исходного текста программы уровня текстового редактора осуществляется программным компонентом, отвечающим за синтаксический анализ программ. В ходе редактирования текста внутренние модели программы меняются на всех уровнях ее представления, а результаты анализа визуализируются в текстовом редакторе в виде подсветки изменений и возможных сообщений об ошибках.

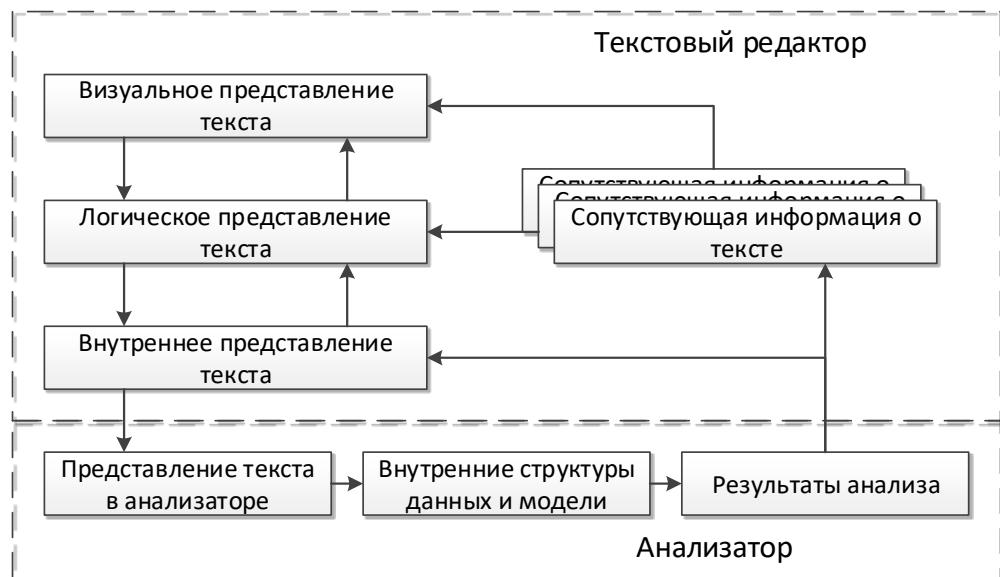


Рисунок 2 – Схема взаимодействия текстового редактора и анализатора

Для обеспечения высокого быстродействия программной реализации разработанного метода предложен алгоритм итеративных преобразований, выполнение которого происходит по событию ввода. В обобщенном виде алгоритм представлен на рисунке 3. Реализация алгоритма предполагает использование анализатором и текстовым редактором общих структур данных, что позволяет сократить количество внутренних перестроений структур данных при изменении текста программы.

Таким образом, предложенный метод предметно-ориентированного анализа исходных текстов программ позволит упростить процедуру выявления ошибок в программных проектах, разрабатываемых с использованием нескольких языков программирования. Это становится возможным за счет интеграции настраиваемого анализатора исходных текстов программ с редакторами программного кода и выполнения анализа в процессе редактирования исходного текста программы. Такое решение позволяет уменьшить время отклика между внесением изменений в текст программы и реакцией среды разработки на результаты анализа внесенных изменений в виде подсветки синтаксиса, уведомления об ошибках, выделения несогласованных частей текста программы.

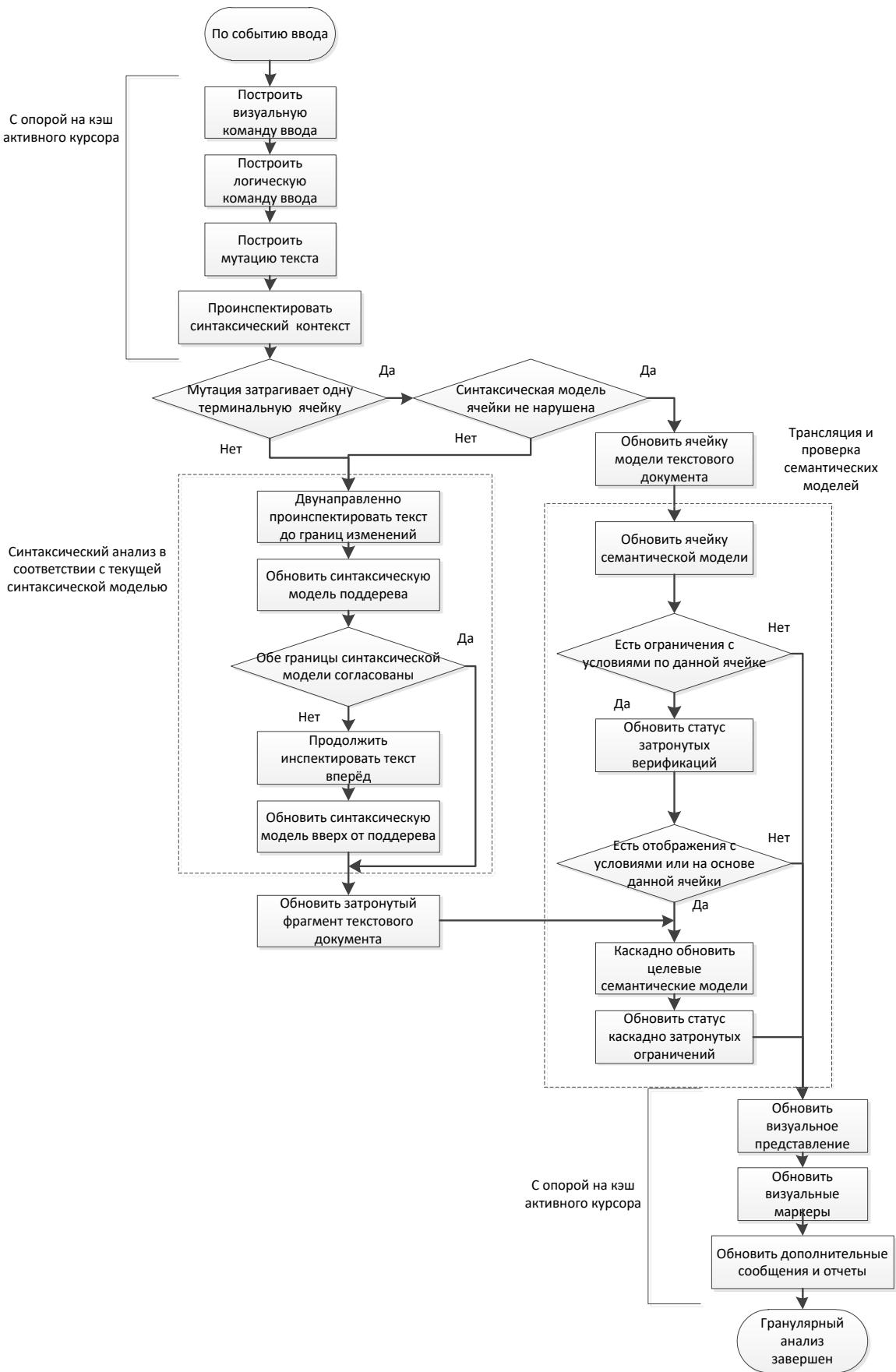


Рисунок 3 – Обобщенный вид алгоритма семантического анализа текста программ в процессе их редактирования

В третьей главе представлен разработанный предметно-ориентированный язык описания семантических моделей программ и алгоритм итеративных преобразований семантических моделей программ.

Разработанный язык позволяет выполнять адаптацию предложенного метода и алгоритма предметно-ориентированного анализа к требованиям каждого конкретного программного проекта. В таблице 1 приведены фрагменты определения синтаксической и семантической модели программы, а также пример отображения пространства имен одной модели на другую.

Описав необходимые преобразования для выполнения семантического анализа исходных текстов программ, написанных, например, на языках программирования Java и JavaScript, разработчики могут в дальнейшем осуществлять анализ согласованности частей программных проектов, написанных с использованием этих языков.

Таким образом, разработанный язык описания семантических и синтаксических моделей позволяет реализовать метод предметно-ориентированного анализа исходных текстов, обеспечивая при этом повторное использование спецификаций при анализе исходных текстов различных программных проектов. Использование предлагаемого языка позволит вывести разработку настраиваемых анализаторов на уровень, доступный разработчикам программного обеспечения, не специализирующимся в области формальных языков и грамматик, компиляторов.

Четвертая глава посвящена разработке программного средства, реализующего описанный метод. Представляются детали реализации необходимых структур данных и алгоритмов, варианты интеграции разработанного решения в среды разработки для дальнейшего применения и оценки.

Таблица 1 – Пример определения предметно-ориентированной модели

Модель	Пример описания
Синтаксическая	<pre>ruleSet: attrs complexName '{' imports body '}'; body: item*; item: rule ruleSet; }; ruleSetImport: attrs alias complexName ';'; rule: attrs complexName args ':' body ';';</pre>
Семантическая	<pre>syntax { scope: { name: string; rules: rule[]; namespace { namespaces: namespace[]; } rule { expr; } }; }</pre>
Отображение	<pre>scope(s: scope) { parent = s; namespace(rs: pds1.ruleSet) from (null, rs) in pds1.definition.body.item.ruleSet { name = rs.complexName; namespaces = from crs in rs.body.item.ruleSet select namespace(this, crs); rules = from r in rs.body.item.rule select rule(this, r); } rule(r: pds1.rule) from (null, r) in pds1.definition.body.item.rule { name = r.complexName; rules = from cr in r.body.simple.rule select rule(this, cr); expr = syntax.expr(this, r.body.simple.expr); } };</pre>

Рассматриваются два способа интеграции:

- 1) в качестве встраиваемого модуля;
- 2) в качестве внешнего сервиса.

При интеграции в качестве встраиваемого модуля реализация анализатора загружается в среду разработки или текстовый редактор, что позволяет выполнять анализ кода программы непосредственно в процессе его редактирования. Такой подход более эффективен с точки зрения производительности, так как не требует преобразования данных для передачи между процессами, выполняющими редактирование и анализ семантических моделей по отдельности.

При интеграции в качестве внешнего сервиса среда разработки или текстовый редактор взаимодействует с программным средством, реализующим анализ кода программы, посредством механизмов межпроцессного взаимодействия, например, через разделяемую память, сетевое соединение или канал. Такой подход позволяет использовать один экземпляр сервиса совместно с несколькими экземплярами среды разработки. Это снижает затраты памяти, так как отсутствует необходимость дублирования представлений моделей программ и реализации анализатора для всех задействованных в работе над программным проектом экземпляров сред разработки или текстовых редакторов. Кроме этого, данный подход позволяет исключить повторный анализ текста неизменённой программы в том случае, когда среда разработки была аварийно закрыта.

В заключении четвёртой главы приводится описание реализации анализатора и результаты его экспериментального применения в составе разработанного настраиваемого редактора текста, рассматриваются детали процесса анализа производительности. Представляются способы маскировки недостатков производительности семантических анализаторов. Такие меры затрудняют сравнение предлагаемого метода с существующими решениями, а

также имеют свои недостатки, так как вводят разработчика в заблуждение относительно корректности текста программы.

На рисунке 4 представлены результаты сравнения среднего времени отклика анализатора. Эксперименты показали положительные результаты применения разработанного анализатора в процессе разработки программных проектов. Реализованный метод позволяет выполнять семантический анализ исходных текстов программ в процессе их редактирования, и предоставлять разработчику программного проекта улучшенные, по сравнению с существующими, средства поддержки редактирования текста программы, такие как автодополнение, подсветка синтаксиса, и другие, в особенности для случаев использования в программном проекте нескольких языков программирования одновременно. Это позволяет успешно выявлять ошибки несогласованности текста программ в многоязыковых программных проектах.

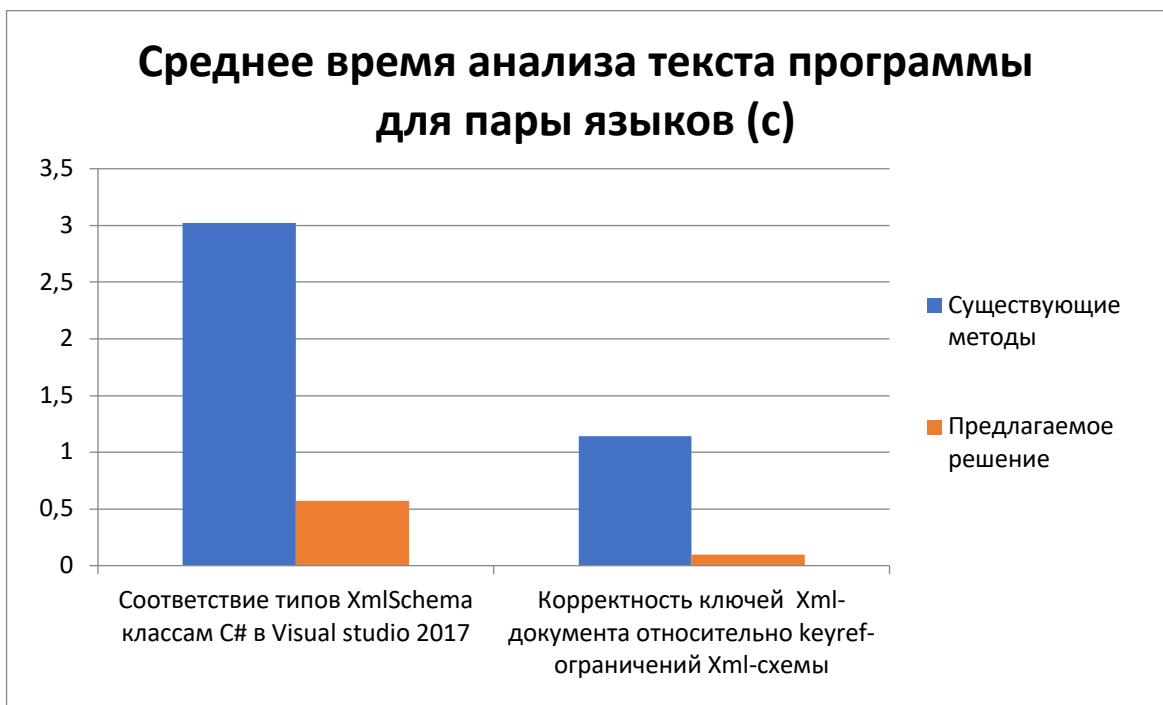


Рисунок 4 – Сравнение среднего времени отклика анализатора на компьютере, оснащенном Xeon E5-1620 3.6GHz 16GB RAM

Таким образом, программная реализация описанного метода и языка описания предметно-ориентированных семантических моделей позволяет, во-

первых, обеспечить создание настраиваемого анализатора исходных текстов программ, допускающего его интеграцию с текстовыми редакторами сред разработки, во-вторых, обеспечить раннее выявление ошибок в процессе редактирования исходного текста программы.

В заключении сформулированы основные научные и практические результаты данной диссертационной работы, показаны перспективы их применения, даны рекомендации по продолжению исследований в области анализа исходных текстов программ и программных систем.

В диссертационной работе получены следующие результаты.

1. Разработан метод предметно-ориентированного анализа на основе итеративных преобразований семантических моделей программ, позволяющий осуществлять семантический анализ текстов программ, разрабатываемых с применением нескольких языков программирования.
2. Разработан язык описания семантических моделей программ, позволяющий адаптировать алгоритм реактивного анализа семантических моделей программ к специфике конкретных программных проектов.
3. Разработан алгоритм реактивного анализа семантических моделей программ, позволяющий уменьшить время отклика среды разработки или текстового редактора между вводом пользователя и отображением результатов анализа актуального состояния программного кода.
4. Разработана программная архитектура и структуры данных для представления текста и динамически изменяющихся семантических моделей программ, что позволяет интегрировать реализацию разработанного метода предметно-ориентированного анализа в различные среды разработки.
5. Разработан анализатор исходных текстов программ, допускающий его интеграцию с существующими средами разработки и прошедший

апробацию в составе реализованного конфигурируемого текстового редактора при решении практической задачи анализа текстов программ, разрабатываемых с использованием нескольких языков программирования.

В части **рекомендаций и перспектив дальнейшей разработки темы**

следует отметить задачу создания репозитория синтаксических и семантических моделей для наиболее распространенных в индустрии разработки программного обеспечения сочетаний языков программирования. Это позволит обеспечить совершенствование предложенного метода и расширить возможности по практическому применению. Дальнейшее исследование алгоритмов синтаксического анализа в части возможности их настройки позволит повысить производительность программной реализации метода, что важно для пользователей, поскольку обеспечивает отзывчивость графического интерфейса среды разработки.

Список публикаций по теме диссертации

В международных изданиях, индексируемых в базе данных Scopus

1. Sadyrin D., Dergachev A., Loginov I., Korenkov I., Ilina A. Application of Graph Databases for Static Code Analysis of Web-Applications // CEUR Workshop Proceedings - 2020, Vol. 2590, pp. 1-9
2. Korenkov I., Loginov I., Dergachev A., Lazdin A. Declarative target architecture definition for data-driven development toolchain // 18th International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM-2018 - 2018, Vol. 18, No. 2.1, pp. 271-278

В изданиях из списка ВАК Минобрнауки РФ

3. Кореньков Ю.Д. Метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей // Научно-технический вестник Поволжья - 2020 - № 7 - С. 32-37

4. Дергачев А.М., Садырин Д.С., Ильина А.Г., Логинов И.П., Кореньков Ю.Д. Методы и средства обнаружения уязвимостей аллокаторов динамической памяти библиотеки glibc // Научно-технический вестник Поволжья - 2020. - № 1. - С. 79-83
5. Кореньков Ю.Д., Логинов И.П. Исследование и разработка языкового инструментария на основе PEG-грамматики // Известия высших учебных заведений. Приборостроение - 2015. - Т. 58. - № 11. - С. 934-938

Публикации в прочих изданиях

6. Логинов И.П., Павлова Е.В., Кореньков Ю.Д. Исследование и реализация алгоритма синтаксического анализа // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. – СПб: Университет ИТМО, [2020]. Режим доступа: <https://kmu.itmo.ru/digests/article/3956> - 2020
7. Кореньков Ю.Д. Итеративное преобразование семантических моделей программ // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. – СПб: Университет ИТМО, 2020. - Режим доступа: <https://kmu.itmo.ru/digests/article/4180>, своб. - 2020

SYNOPSIS

The relevance. In the software development area, software tools are of great importance to support the development process. The main tool is the Integrated Development Environment (IDE) - a set of programs that provides the developer with tools for editing source code, version control, debugging, translation, teamwork, and many other features.

One of the main functions of integrated development environments is editing the source code of programs. Modern text editors are able to partially automate the development process, which significantly increases the productivity of the programmer. Features of text editors allow you to highlight syntax elements by a different color, notify the programmer about errors, automatically add syntactic constructions when typing, navigate through the source code, and visualize the structural organization of the program.

A feature of the modern software development industry is the use of several programming languages in one software project. This feature gives rise to the problem of checking the consistency of components implemented in different languages and used as part of one software system. As a rule, component inconsistency is tracked only during the debugging or testing phase. For example, if a C# program uses a function implemented in a separate C module, and the signature of this function has changed, then the resulting error in the program system can be traced only at runtime. This state of affairs significantly reduces the efficiency of the development process.

The goal of this work is developing approaches in the field of analysis of the source code of programs written in several programming languages in order to identify such errors at the coding stage, which will reduce the number of editing-debugging-testing cycles and, in general, will have a positive effect on the software development process.

State of the art. Noam Chomsky, Stephen Kleene, Masaru Tomita, Jay Earley, Vaughan Pratt made a great contribution to the theory of formal languages

and grammars; in the development of relational models and mappings – Edgar Codd, Christopher Date; in the development of semantic technologies and metamodeling – Yorick Wilkes.

Microsoft, Oracle, IBM, JetBrains and others are making hands-on contributions in the field of software tools. The leading solutions in the field of this topic are the development environments for domain-specific languages, such as JetBrains MPS, Eclipse Xtext, which do not fully solve the problem and have a number of disadvantages:

- 1) do not have the ability to perform joint analysis of program models written using several languages simultaneously;
- 2) do not have the ability to perform semantic analysis while editing text;
- 3) are limited to use by their own development environment (for example, Racket, Kermeta, Eclipse Xtext);
- 4) do not analyze the source code written in algorithmic programming languages (for example, JetBrains MPS).

The object of the research is the source code and semantic models of programs written in several programming languages.

The subject of research is methods and algorithms for the analysis of source codes of programs in the process of editing them using semantic models.

The goal detection of errors in the source code, written using several programming languages, at the stage of editing the source code of programs.

In compliance with the goal in the dissertation work, the following **tasks** are set and solved.

1. Analysis of methods of internal representation and processing of edited text of programs in integrated development environments in order to identify the shortcomings of existing tools for supporting multilingual software projects.
2. Development of the method of domain-specific analysis of source code of programs in the process of editing based on semantic models.

3. Development of data structures and software architecture for the internal representation of the edited text and semantic models of programs.
4. Development and implementation of a software tool for analysis of source code of programs, testing the developed method.

Research methods. When solving the set tasks, the methods of object-oriented analysis, methods of syntactic and semantic analysis, metamodeling, methods of language-oriented programming were used.

The scientific novelty of the work. The dissertation obtained the following results, which are characterized by scientific novelty.

1. The method of domain-specific analysis of source codes of programs, which differs from the existing ones by the ability to analyze the texts of programs developed using several programming languages.
2. The language for describing semantic models of programs, conceptually different from the existing ones by the joint use of relational representation and representation in the form of a graph, allowing to adapt the algorithms for analyzing semantic models of programs to the specifics of specific software projects.
3. An algorithm for iterative transformations of semantic models of programs, using modified data structures for the internal representation of text during its editing and analysis, which allows reducing the response time of the development environment or text editor between user input and displaying the analysis results of the current state of the program code.

The practical significance of the work.

1. Software architecture for the presentation of dynamic semantic models of programs, which allows integrating the implementation of the developed method of domain-specific analysis into various development environments or text editors.
2. The source code analyzer is implemented as a software component that allows its integration with existing development environments, which has been tested

as part of an implemented configurable text editor in solving the practical problem of analyzing program texts developed using several programming languages.

The theoretical contribution of this work consists of the future development of the theory of the formal languages in terms of the analysis of program source code written using several programming languages.

The main contributions.

1. The method of domain-specific analysis of source code of programs developed using several programming languages.
2. A language for describing semantic models of programs, which allows customize text analysis algorithms for specific software projects.
3. Algorithm for reactive analysis and iterative transformation of semantic program models.

Approbation of the research results. The main points of the work were presented at the following scientific conferences: VIII, IX Congress of Young Scientists of ITMO University (CMU) (2019, 2020), XLVI - XLIX Scientific and Educational Conference of ITMO University (2017-2020), IX Scientific and Practical Conference of Young Scientists "Computing Systems and Networks (Majorov Readings 2017)", 17th FRUCT Conference, XI Majorov International Conference on Software Engineering and Computer Systems (2019).

The reliability of results is validated by completeness of analysis of theoretical and practical research, discussions at conferences and scientific congresses, practical verification and implementation of the results obtained.

Implementation of work results. The results of the work were obtained within the research № 619296 «Development of methods for the creation and implementation of cyber-physical systems» and introduced into the development and production processes of LLC "Tune-It" company in solving problem of analyzing the texts of programs developed using several programming languages for the presence of potential vulnerabilities identified at the source code level, which is

confirmed by acts of implementation. Also results used in the educational process of ITMO University.

Publication of research results. The main results on the topic of the dissertation are presented in seven main works have been published, of which three articles in journals recommended by the Higher Attestation Commission, in which the main scientific results of dissertations for the PhD degree should be published, and two articles in journals indexed by Scopus.

The personal contribution of the author. The main results presented in the dissertation were obtained personally by the author. Dissertation includes results that have the author's personal contribution on the topic of the dissertation or related topics such as parsing. Co-author Dergachev A.M. carried out general methodological supervising, as well as the organization of work on the technical implementation of the proposed tools and methods.

- In the work «Research and development of language workbench based on PEG-grammar» the author developed the IDE Visual Studio plug-in, which includes a modified text editor AvalonEdit, modified for syntax highlighting based on the developed parser and dynamically changed grammars. Loginov I.P. tested the proposed language tools, the development of grammars of test languages, and Lazdin A.V. – made comparison of the developed functionality with existing analogues.
- In the work «Declarative target architecture definition for data-driven development toolchain» the author developed the syntactic and semantic analyzers for a language for describing target platform architectures. This language was developed by Loginov I.P. A.V. Lazdin the study of the possibilities of describing processor architectures by means of the compilers GCC and LLVM was carried out.
- In the work «Application of Graph Databases for Static Code Analysis of Web-Applications» the author has implemented the analysis of the syntax tree for the use of certain functionality, leading to the exploitation of

vulnerabilities in web applications. Co-author Sadyrin D.S. reviewed the tools used in static code analysis. Loginov I.P. implemented the generation of reports with the analysis results. Ilyina A.G. carried out research on graph search queries in terms of formal languages.

- In the work «Research and implementation of the parsing algorithm» the author proposed the use of the MARPA algorithm to build adaptable parsers and created a prototype of its implementation for an environment based on the Common Language Infrastructure. Pavlova E.V. implemented software implementation of the parser based on the MARPA algorithm, I.P. Loginov carried out the development of test domain-specific languages.
- In the work «Iterative transformation of semantic models of programs» the author proposes a method for iterative transformation of semantic models of programs, applicable for the domain-specific analysis of source codes of programs and created a prototype of its implementation.
- In the work «Method of domain-specific analysis of source texts of programs based on semantic models» the author proposed a language for describing the specifications of semantic models of programs and their mappings, describing syntax models of source codes of programs for constructing abstract semantic graphs in the course of semantic analysis of program texts.
- In the work «Methods and means for detecting vulnerabilities in dynamic memory allocators of the glibc library» the author analyzed the nature of vulnerabilities and investigated the possibility of detecting them at the level of writing the source code of programs developed using C/C++ and assembler. Ilyina A.G. reviewed methods of program verification to check correctness during execution process. Sadyrin D.S. described an approach that allows you to combine symbolic program execution with real. Loginov

I.P. carried out a practical study of the features of code analysis tools, memory use.

Compliance with the passport of the scientific specialty.

The dissertation work corresponds to the passport of the scientific specialty 05.13.11 «Mathematical support of computers, complexes and computer networks», and the research carried out corresponds to the specialty formula. The study corresponds to the following points of the specialty passport.

1. Models, methods and algorithms for the design and analysis of programs and software systems, their equivalent transformations, verification and testing.
2. Programming languages and programming systems, program semantics.

Structure of work.

The thesis consists of an introduction, four chapters, a conclusion, a list of references (100 sources). Contains 127 pages of text, including four tables and 19 figures.

CONTENTS

In the introduction, the relevance of the topic of the dissertation work is substantiated, a general description of the work is given, the object and subject of research are determined, the goals, objectives and methods of research are formulated, the novelty is presented, the scientific and practical significance is shown, the main results of the work are given.

The first chapter examines the subject area – the existing approaches to the analysis of program texts, the implementation features of integrated development environments and analyzers of program source code.

The area of developing program translators in general and source code analyzers, in particular, is well-developed. The main task that the source code analyzers solve is to check the correctness of programs in terms of syntax and semantics, as well as to prepare the program for further translation into a machine-

oriented representation (i.e., for synthesizing machine code). Aspects related to code synthesis are not considered in this work.

The analysis of the source code of programs is carried out in several stages. First of all, lexical analysis (scanning) is performed - the procedure for extracting programming language elements, called lexemes, from the program text. The result of scanning is a set of tokens - tokens, which are associated with categories of language constructions, such as «keyword», «separator characters», «identifier», «literal», «comment». This set serves as input to the parser. The parser implements the next stage of processing the text of the program, at which it checks the correspondence of the text of the program to certain rules, the totality of which is called the grammar of the language. The grammar of a language specifies a set of words and symbols that can be used to write a program, as well as their possible combinations that allow the text to be considered correctly structured. The result of parsing is a parse tree that reflects the syntactic structure of the program. The nodes of the tree are declarations of types, subroutines, individual statements, etc. This tree contains data about all program elements recorded in the source code and their interaction.

After checking the correctness of the program in terms of syntax, it is further transformed in order to obtain machine code. However, within the framework of this work, it is of interest to analyze with the aim of implementing tools for editing the text of programs, rather than subsequent translation. Analyzers used in integrated development environments are designed to facilitate the perception of the text of the program, as well as to detect errors. Some functionality can be implemented based on the results of parsing, such as syntax highlighting.

In some cases, it is required to analyze the text taking into account the context in which one or another programming language construct is used. For example, C# has a built-in data query language called LINQ. Keyword highlighting of a given language is required only as part of expressions describing a query, and syntax highlighting should not be performed when these words are used outside of query

descriptions. To take into account the context in which various language constructs are used, as well as to implement the consistency check of program components implemented in different programming languages, it is necessary to implement semantic analysis mechanisms. Such mechanisms should take into account the peculiarities of the data types used when writing the program text in terms of the admissibility of performing the recorded operations, including subroutine calls, as well as the scope of types and subroutines, and the availability of variables. At the level of interaction between components, it is important to establish a correspondence between the imported and exported functionality.

The problem with existing IDEs is that a separate analyzer is implemented for each programming language, which does not provide the possibility of mutual integration or data exchange with analyzers for other programming languages. This is widely due to differences in the syntax and semantics of programming languages, from which it follows that it is impossible to compare the results of analyzing program texts to each other. Figure 1 shows a typical scheme for implementing support for programming languages in integrated development environments in relation to semantic analysis of program texts.

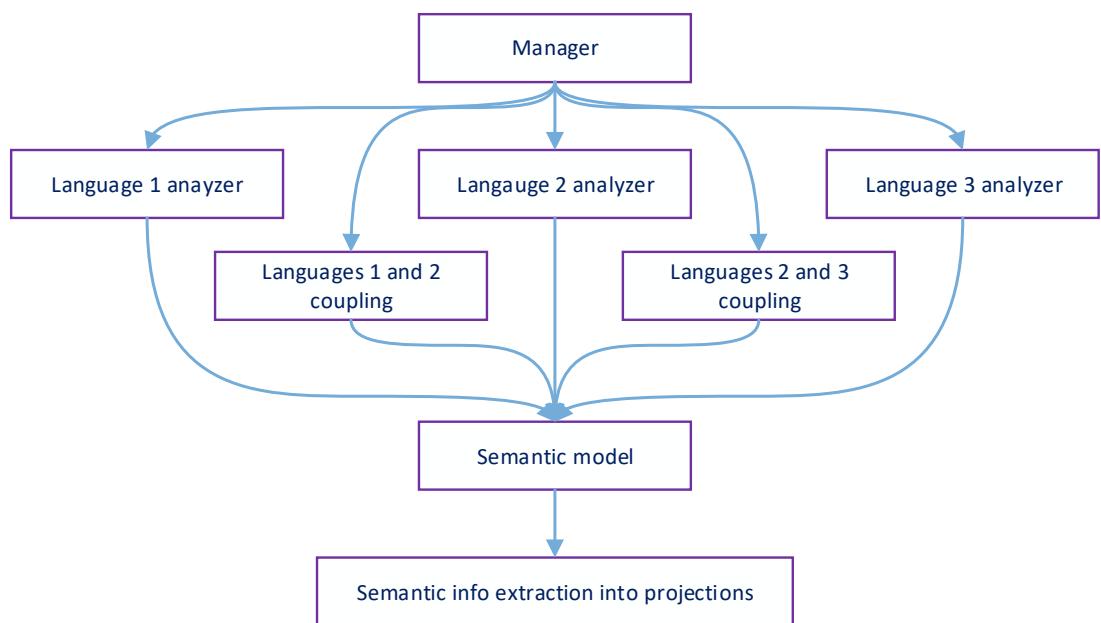


Figure 1 – A typical scheme for the extensibility of integrated development environments in relation to semantic analysis of program texts

It can be seen from the figure that when adding another language, the implementation of the analyzer becomes much more complicated due to the need to create interface modules for language pairs.

In the course of the study of this problem, an analysis was made of the methods of an internal representation of the edited text of programs, the structure of programs, as well as a study of the shortcomings of existing tools for supporting multilingual development. The way in which the edited program text is presented is important because it affects the response time of the user interface in response to text changes. For example, highlighting a keyword after completing its typing should be instantaneous for the programmer to ensure usability. The described example is a typical problem in using development environments.

Based on the results of the analysis of existing solutions and their shortcomings, a number of tasks are formulated that are consistently solved to achieve the goal set in the thesis.

The first task is to develop a method for domain-specific analysis of program source codes. The method is based on the use of a domain-specific language to describe program elements. This language is a metalanguage in relation to programming languages and allows you to build an intermediate representation of a program with specified classes of attributes common to programs developed using several programming languages. Based on these attributes, the conformity of the semantic models of the programs is checked. In this work, the semantic model of a program is understood as an intermediate representation that describes the properties and relationships of its elements.

The next task is to implement an algorithm for analyzing semantic models of programs. Existing methods of representing semantic models of programs and integrating analyzers that are part of development environments do not allow using them for domain-specific analysis. Therefore, it is necessary to develop alternative software architecture and data structures that allow integrating the implementation

of the developed method of domain-specific analysis into various development environments or text editors.

The final stage of work is the approbation of the developed method and means of its adaptation in the process of developing real software products. Creation of a software tool based on the developed methods and software architectures is necessary to show its efficiency and practical applicability. Such an analyzer will also allow you to evaluate the effectiveness in comparison with existing tools for scripts, for which there are still specialized tools for analyzing program texts.

Thus, there is no one-size-fits-all tool for consistent software development across multiple programming languages. The existing specialized tools have a number of restrictions on expanding functionality due to the specifics of the software implementation. To solve the problem, it is necessary to develop a method of subject-oriented analysis and a means of adapting its software implementation to the tasks of individual projects.

The second chapter presents the developed method of domain-specific analysis of program texts. The main idea of the method is to use custom procedures for dynamic translation and iterative transformations of semantic program models based on the analysis results. That is, depending on the set of programming languages used in the development of software systems, a description of the procedure for translating individual aspects of the source text into a metamodel common for these languages is created. The proposed method differs from the existing ones in that there is no need to create specialized analyzers in advance to represent the semantic model of a program. A representation of the program model based on a semantic network is used, which is a metamodel in relation to the grammars of the languages involved in the development of a specific software project.

Semantic models are formed on the basis of specifications, which can formally be represented by a record of the form:

$$S = (T_s, P_s, V_s, C_s, D_s), \quad (1)$$

where $t_s \in T_s$ – type identifiers, $p_s \in P_s$ – property identifiers, V_s – value schemas, written in the form:

$$V_s = \{v_s = (t, q): t \in T_s \cup T_0\}, \quad (2)$$

$$q = \begin{cases} \text{false} - \text{for single values} \\ \text{true} - \text{for sets of values} \end{cases}$$

},

for $v_s = (t, q)$ – a pair that specifies the way the attribute value is represented in the semantic model (t – type of value, q – multiplicity of value); $C_s = \{c_s = t_s \rightarrow P_{t_s}: P_{t_s} \subseteq P_s\}$ – properties belonging to types, written in the form of mappings of identifiers of types t_s to a set of properties P_{t_s} ; $D_s = \{d_s = (t_s, p_s) \rightarrow v_s\}$ – type schemas described as mappings of type-property pairs (t_s, p_s) for schema values v_s ; T_0 – built-in literal value types.

The formal representation of the semantic model of the program can be written as:

$$M = (S_m, N_m, A_m, V_m, E_m, F_m), \quad (3)$$

where: S_m – semantic model specification, $n_m \in N_m$ – element identifiers, $a_m \in A_m$ – attribute identifiers, V_m – value of attributes, described as pairs "schema"- "set of literal values of element identifiers":

$$V_m = \{v_m = (v_s, X_{v_m}): v_s \in V_s, v_s = (t, q)\}, \quad (4)$$

$$|X_{v_m}| \in \begin{cases} \{0,1\}, \text{ если } q = \text{false}, \\ \{\mathbb{N}^0\}, \text{ если } q = \text{true}, \end{cases}$$

$$X_{v_m} = \left\{ x_{v_m}: \begin{cases} x_{v_m} = \{\text{literal value}\}, & \text{if } t \in T_0 \\ x_{v_m} \in N_m, & \text{if } t \in T_s \end{cases} \right\}$$

};

$E_m = \{e_m = n_m \rightarrow (t_{n_m}, P_{n_m})\}$ – elements described as mappings of element identifiers to pairs of the type-set of relations type, specified by pairs of property-attribute (an element is matched with a type and a set of attributes corresponding to properties of this type); $F_m = \{f_m = (n_m, p_s) \rightarrow (a_{n_m}, v_{n_m})\}$ – element fields,

described in the form of mappings of node-property pairs into attribute-value pairs (an attribute and an associated value are mapped to an element property):

A syntactic model specification is a special case of a semantic model specification (1) extended by a context-free grammar:

$$S = (T_S, P_S, V_S, C_S, D_S, G_S), \quad (5)$$

where grammar components $G_S = (\Sigma_{G_S}, N_{G_S}, P_{G_S}, S_{G_S})$ following: Σ_{G_S} – terminal character set, $n_{G_S} \in N_{G_S}$ – a set of nonterminal symbols, P_{G_S} – set of rules of the form $n_{G_S} \rightarrow w$, where n_{G_S} – nonterminal, w – any sequence of terminals and non-terminals, S_{G_S} – starting nonterminal.

Then the syntactic model is a special case of the semantic model with the specification of the syntactic model as S_m (3).

Formal representation of a semantic translation specification:

$$R_{1 \rightarrow 2} = (S_1, S_2, Q_{1 \rightarrow 2}, P_{1 \rightarrow 2}, I_{1 \rightarrow 2}), \quad (6)$$

where S_1 – source semantic model specification, S_2 – target semantic model specification, $Q_{1 \rightarrow 2}$ – set of context mappings, $P_{1 \rightarrow 2}$ – set of context-free mappings, I – set of initial mappings for the original model m_{s1} and target m_{s2} :

$$\begin{aligned} Q_{1 \rightarrow 2} &= \{q_{t_{s1} \rightarrow t_{s2}} = (n_{s1}, m_{s1}, q') \rightarrow m'_{s2}\} \\ P_{1 \rightarrow 2} &= \{p_{t_{s1} \rightarrow t_{s2}} = (n_{s1}, m_{s1}) \rightarrow m'_{s2}\} \\ I_{1 \rightarrow 2} &= \{\iota_{t_{s1} \rightarrow t_{s2}} = m_{s1} \rightarrow m'_{s2}\} \end{aligned} \quad (7)$$

where each of the particular translations $f_{t_{s1} \rightarrow t_{s2}}$ (q , p , и ι), described by functions, sets the mapping of an element of type t_{s1} source model m_{s1} to element of type t_{s2} of target model m_{s2} ,

Initial mappings $\iota_{t_{s1} \rightarrow t_{s2}}$ set root translations, from which the formation of dependencies between model elements begins. These can include calls to free mappings $p_{t_{s1} \rightarrow t_{s2}}$ and context mappings $q_{t_{s1} \rightarrow t_{s2}}$, specifying direct dependencies between one source element and one target element without additional arguments and taking into account additional arguments q' . Each display connects the structural elements of the models m_{s1} and m_{s2} through relational mappings. Initial mappings

proceed as sources of information to initiate the semantic translation algorithm. Context-free mappings applied to specific elements of the original model, and context mappings, used to perform subsequent steps of the semantic translation algorithm, initiated indirectly during the preceding steps. In this case, the data obtained in the previous steps, for example, when using primary mappings, will be used as an argument and context node.

In a formalized form, the developed method can be written as follows.

1. Let the set Ψ – texts τ of software system, written in languages with syntactic specifications S_τ :

$$(\tau_n, S_{\tau_n}) \in \Psi, 1 \leq n \leq |\Psi|, \quad (8)$$

where S_{τ_n} – syntactic specification of program text τ_n , n – number of texts τ of software system. Then the result of the parsing of these texts will be set of syntactic models Υ :

$$\Upsilon = \{v_{S_\tau} \forall (\tau, S_\tau) \in \Psi\}, |\Psi| = |\Upsilon|, \quad (9)$$

where set Ψ – texts τ of software system, v_{S_τ} – syntax model by specification S_τ for source text of program τ .

2. Let Θ – domain areas within which the analysis of the texts of the software system is carried out, given by a set of specifications for semantic translations:

$$\Theta = \{R_n = (S_{1n}, S_{2n}, \dots)\}, 1 \leq n \leq |\Theta|, \quad (10)$$

where R_n – semantic translation specification, S_{1n} – original semantic model specification, S_{2n} – target semantic model specification.

3. Domain-specific analysis based on semantic models for text τ gives the set of semantic models Ω_τ , which is the union of all semantic models obtained as a result of translations $f(m, r)$, starting with the syntax model v_{S_τ} directly or indirectly for a given semantic model $m = (S, \dots, \dots)$ with a suitable semantic translation specification $r = (S, \dots, \dots) \in \Theta$ for the semantic model specification S :

$$\Omega_\tau = \bigcup_{\substack{m_0=v_{S_\tau} \\ m_n=(S,\dots,\dots,\dots) \\ \exists r_{n \rightarrow n+1}=(S,\dots,\dots,\dots) \in \theta}} \{m_n\}, m_{n+1} = f(m_n, r_{n \rightarrow n+1}) \quad (11)$$

where m_n – semantic model (3), semantic translation specification $r_{n \rightarrow n+1}$ (6), $f(m_n, r_{n \rightarrow n+1})$ – translating function:

$$f(m_{S_a}, r_{a \rightarrow b}) = \varsigma(S_b, \{m' = \iota(m_{S_a}) \forall \iota \in I, \quad (12)$$

$$m_{S_a} = (S_a, \dots, \dots, \dots)$$

$$r_{a \rightarrow b} = (S_a, S_b, Q, P, I)\}),$$

where $r_{a \rightarrow b}$ – semantic translation specification (6) from semantic model m_{S_a} by specification S_a to semantic model by the specification S_b , m' – partial target model resulting from the translation application $\iota \in I$ from specification $r_{a \rightarrow b}$ for model m_{S_a} , ς – a function that combines a set of semantic models m'_Σ with general specification S to one general model $(S, N_b, A_b, V_b, E_b, F_b)$ (3):

$$\varsigma = (S, m'_\Sigma) \rightarrow (S, \quad (13)$$

$$N_b = \varsigma'(m'_\Sigma, (S, N_{m'_\Sigma}, \dots, \dots) \rightarrow N_{m'_\Sigma}),$$

$$A_b = \varsigma'(m'_\Sigma, (S, A_{m'_\Sigma}, \dots, \dots) \rightarrow A_{m'_\Sigma}),$$

$$V_b = \varsigma'(m'_\Sigma, (S, V_{m'_\Sigma}, \dots, \dots) \rightarrow V_{m'_\Sigma}),$$

$$E_b = \varsigma'(m'_\Sigma, (S, E_{m'_\Sigma}, \dots, \dots) \rightarrow E_{m'_\Sigma}),$$

$$F_b = \varsigma'(m'_\Sigma, (S, F_{m'_\Sigma}, \dots, \dots) \rightarrow F_{m'_\Sigma})$$

),

where ς' – a function that combines the results of applying the argument z to each of the members of the set specified by the argument Y :

$$\varsigma' = (Y, z) \rightarrow \bigcup z(y) \forall y \in Y \quad (14)$$

4. M_Γ – all semantic models (9) for all texts Γ (8) (primary syntactic and translations derived from them in aggregate):

$$M_\Gamma = \bigcup \Omega_\tau \forall v_{S_\tau} \in \Gamma, \quad (15)$$

and Γ – common semantic model space combining models with a common specification:

$$\Gamma = \{\gamma_s = \varsigma(s, X) : \forall x = (s, \dots, \dots) \in X \subseteq M_\Gamma\}, \quad (16)$$

consisting of combined by function ς (13) semantic models x (3) for each of the available specifications s (1).

Semantic model specifications can be supplemented by predicates given in any of the semantic model interpretations. The construction of the space Γ does not require the fulfillment of all the constraints and predicates of semantic and syntactic models, since they describe formal dependencies between data elements and their members. According to the satisfaction of these constraints and predicates, the software developer can judge whether the texts of the software system comply with these specifications.

The application of the proposed method is based on the use of a domain-specific language. This language is intended for describing the specifications of program models. It is a system of concepts that combines elements of relational mappings, knowledge representation based on semantic networks and static typing. This allows:

- 1) to lower the threshold of entry for developers who draw up specifications for the analysis of program texts, as it is based on the well-known concepts of object-oriented analysis and relational transformations;
- 2) to simplify the identification of errors in the specifications of texts and semantic models of programs due to static typing.

Figure 2 shows a generalized diagram of the interaction between a text editor and an analyzer. The transformation of the representations of the source text of a text editor-level program is carried out by a program component responsible for parsing programs. During text editing, the syntax model of the program changes at all levels of its presentation, and the analysis results are visualized in a text editor in the form of highlighting changes and possible error messages.

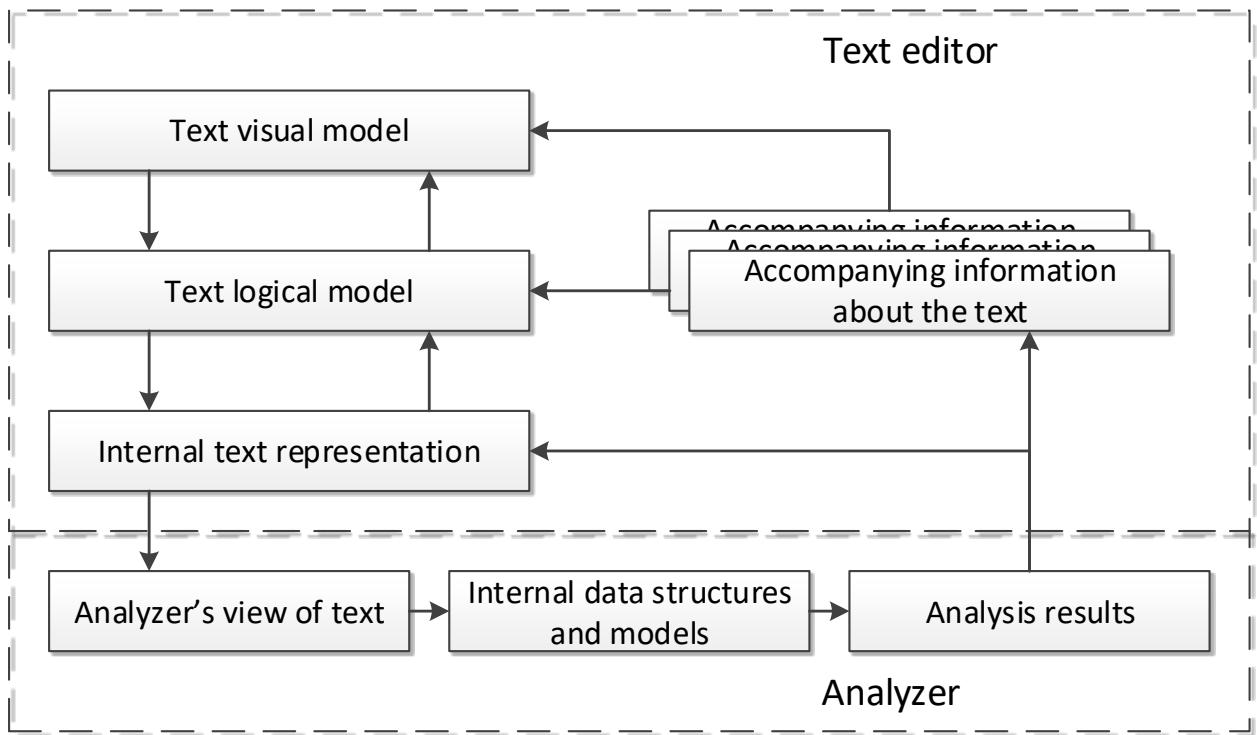


Figure 2 – Structure of interaction between text editor and analyzer

To ensure high performance of the software implementation of the developed method, an iterative transformation algorithm is proposed, the execution of which occurs on an input event. In a generalized form, the algorithm is shown in Figure 3. The implementation of the algorithm assumes the use of the analyzer and the text editor of common data structures, which reduces the number of the internal rebuilding of data structures when changing the program text.

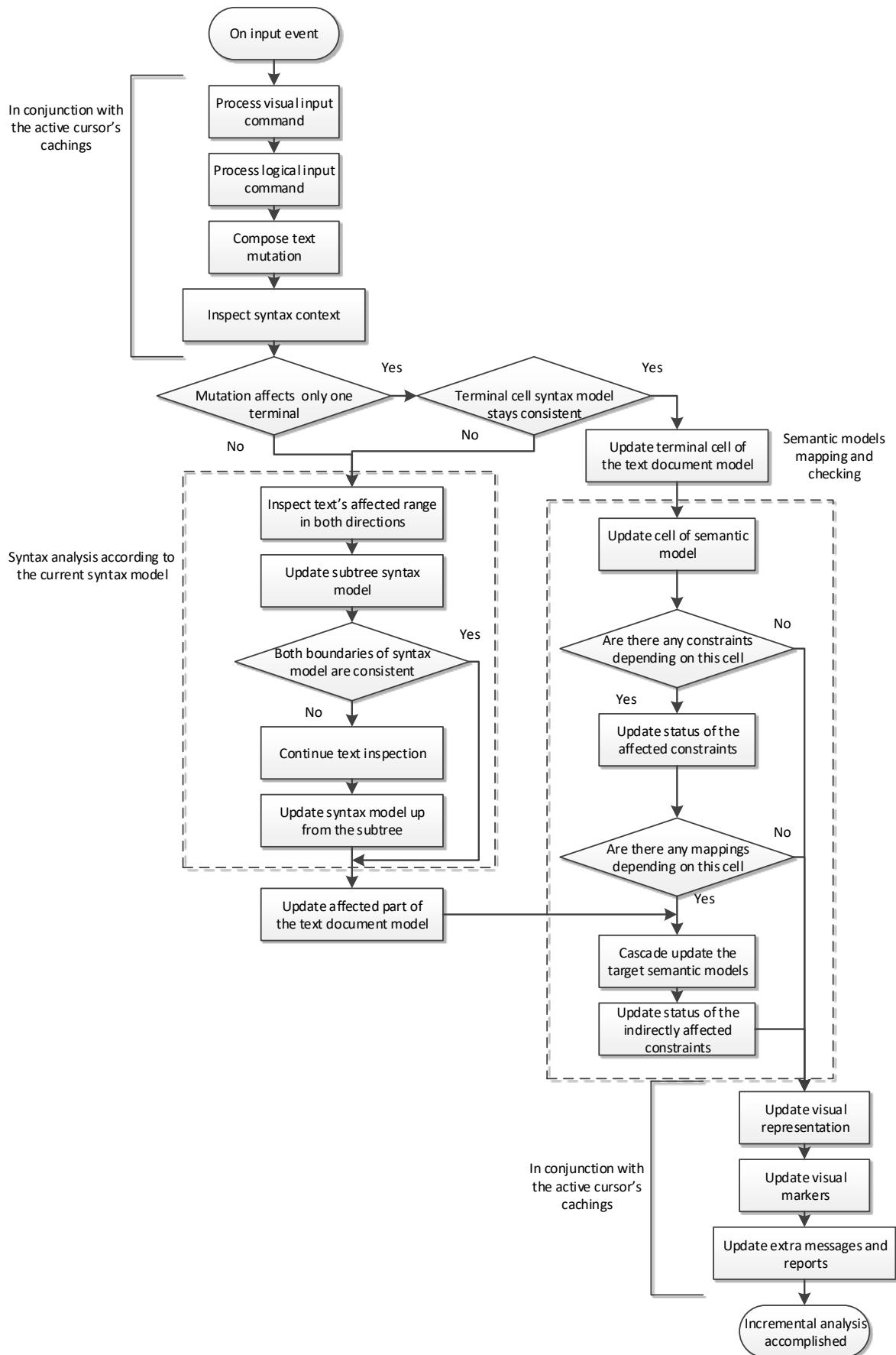


Figure 3 – Generalized view of the algorithm for semantic analysis of the text of programs in the process of editing

Thus, the proposed method of domain-specific analysis of source codes of programs will simplify the procedure for detecting errors in software projects developed using several programming languages. This is made possible by integrating a custom source code analyzer with program code editors and performing analysis while editing the source code of the program. This solution allows reducing the response time between changes in the program text and the reaction of the development environment to the results of the analysis of the changes made in the form of syntax highlighting, error notification, highlighting inconsistent parts of the program text.

The third chapter presents the developed domain-specific language for describing semantic program models and an algorithm for iterative transformations of semantic program models.

The developed language makes it possible to adapt the proposed method and algorithm of domain-specific analysis to the requirements of each specific software project. Table 1 shows fragments of the definition of the syntactic and semantic model of the program, as well as an example of mapping the namespace of one model to another.

So, for example, having once described the necessary transformations to perform semantic analysis of source codes of programs written in the programming languages Java and JavaScript, developers can further carry out domain-specific analysis in relation to the pairing of parts of projects written using these languages.

Thus, the developed language for describing semantic and syntactic models makes it possible to implement the method of domain-specific analysis of source texts, while ensuring the reuse of specifications when analyzing the source texts of various software projects. The use of the proposed language will make it possible to bring the development of custom analyzers to a level accessible to software developers who do not specialize in the field of formal languages and grammars, and compilers.

Table 1— Example of defining a domain-specific model

Model	Example of description
Syntactical	<pre> ruleSet: attrs complexName '{}' imports body '}'; body: item*; item: rule ruleSet; '; ruleSetImport: attrs alias complexName ';'; rule: attrs complexName args ':' body ';'; </pre>
Semantical	<pre> syntax { scope: { name: string; rules: rule[]; namespace { namespaces: namespace[]; } rule { expr; } }; } </pre>
Mapping	<pre> scope(s: scope) { parent = s; namespace(rs: pds1.ruleSet) from (null, rs) in pds1.definition.body.item.ruleSet { name = rs.complexName; namespaces = from crs in rs.body.item.ruleSet select namespace(this, crs); rules = from r in rs.body.item.rule select rule(this, r); } rule(r: pds1.rule) from (null, r) in pds1.definition.body.item.rule { name = r.complexName; rules = from cr in r.body.simple.rule select rule(this, cr); expr = syntax.expr(this, r.body.simple.expr); } }; </pre>

The fourth chapter is devoted to the development of a software tool that implements the proposed method. The details of the implementation of the necessary data structures and algorithms, options for integrating the developed solution into the development environment for further application and evaluation are described.

Two ways of integration are considered:

- 1) as a plug-in module;
- 2) as an external service.

When integrated as an external service, a development environment or text editor interacts with a software tool that implements the analysis of the program code through interprocess communication mechanisms, for example, through shared memory, a network connection, or a pipe. This approach allows a single service instance to be shared with multiple development environment instances. This is more efficient in terms of consumed memory since there is no need to duplicate program model representations and implement an analyzer for all instances of the development environment or text editor involved in the work on a software project. In addition, this approach allows you to exclude re-analysis of the text of an unchanged program in the case when the development environment process was abnormally terminated.

When integrated as an external service, a development environment or text editor interacts with a software tool that implements the analysis of the program code through interprocess communication mechanisms, for example, through shared memory, a network connection, or a pipe. This approach allows a single service instance to be shared with multiple development environment instances. This is more efficient in terms of consumed memory since there is no need to duplicate program model representations and implement an analyzer for all instances of the development environment or text editor involved in the work on a software project. In addition, this approach allows you to exclude re-analysis of the text of an unchanged program in the case when the development environment process was abnormally terminated.

At the end of the fourth chapter, a description of the analyzer implementation and the results of its experimental use as part of the developed custom text editor is given, details of the performance analysis process are considered. The ways of masking the performance shortcomings of semantic analyzers that function while editing the analyzed source code of the program are discussed. Such measures make it difficult to compare the proposed method with existing solutions, and also have their own drawbacks, since they mislead the developer regarding the correctness of the program text.

Figure 4 shows the results of a comparison of the average response time of the analyzer. The experiments have shown positive results of using the developed analyzer in the process of developing software projects. The implemented method allows you to perform semantic analysis of source codes of programs in the process of editing them, and provide the developer of a software project with improved support for editing program text, such as autocompletion, syntax highlighting, and others, in particular for cases of using several languages in a software project at the same time. This allows you to successfully detect errors of the inconsistency of the text of programs in multilingual software projects.

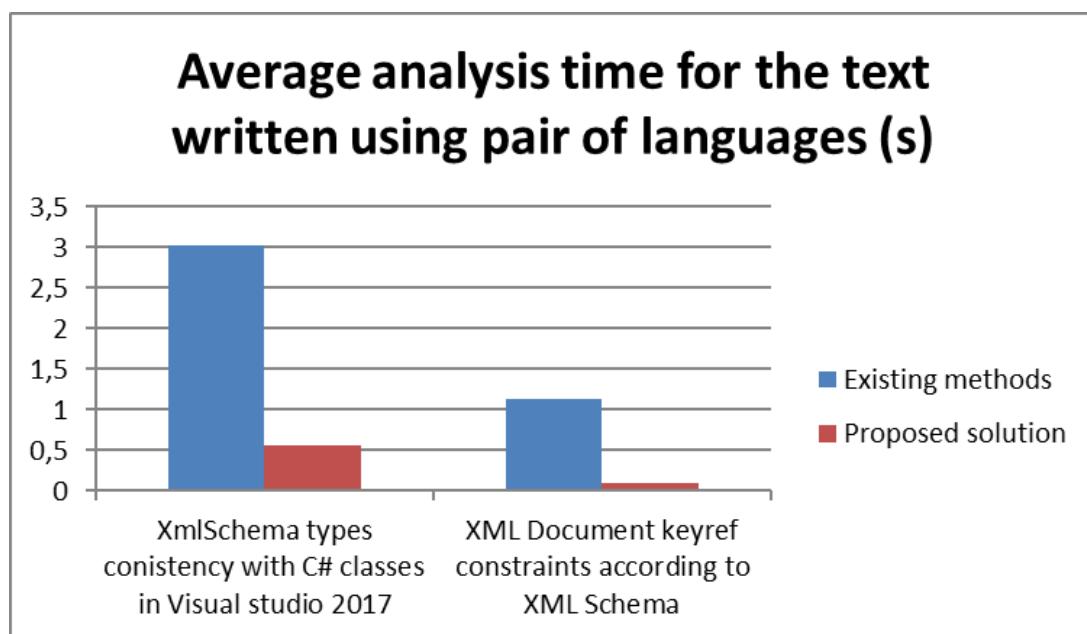


Figure 4 – Comparison of the average response time of the analyzer on a computer equipped with a Xeon E5-1620 3.6GHz 16GB RAM

Thus, the software implementation of the described method and the language for describing domain-specific semantic models allows, firstly, to ensure the creation of a custom analyzer of source codes of programs that allows its integration with text editors of development environments, and secondly, to ensure early detection of errors in the process of editing the source code. the text of the program.

In the conclusion, the main scientific and practical results of this dissertation work are formulated, the prospects for their application are shown, recommendations are given for the continuation of research in the field of analysis of source codes of programs and software systems.

In the dissertation work, the following results were obtained.

1. A method of domain-specific analysis based on iterative transformations of semantic models of programs has been developed, which allows for the semantic analysis of texts of programs developed using several programming languages.
2. The language of description of semantic models of programs has been developed, which allows adapting the algorithm of reactive analysis of semantic models of programs to the specifics of specific software projects.
3. An algorithm for reactive analysis of semantic models of programs has been developed, which makes it possible to reduce the response time of the development environment or a text editor between user input and displaying the results of the analysis of the current state of the program code.
4. A software architecture and data structures have been developed to represent text and dynamically changing semantic models of programs, which allows integrating the implementation of the developed method of domain-specific analysis into various development environments.
5. A source code analyzer has been developed that allows its integration with existing development environments and has been tested as part of an implemented configurable text editor when solving a practical problem of analyzing program texts developed using several programming languages.

In terms of **recommendations and prospects for further development** of the topic, it should be noted the work on the development of a repository of syntactic and semantic models for the most common combinations of programming languages in the software development industry. This will ensure the improvement of the proposed method and expand the possibilities for practical application. Further research of parsing algorithms in terms of the possibility of their customization will improve the performance of the software implementation of the method, which is important for users since it provides responsiveness of the graphical interface of the development environment.

Publications of the author

Publications in international editions, indexed in Scopus

1. Sadyrin D., Dergachev A., Loginov I., Korenkov I., Ilina A. Application of Graph Databases for Static Code Analysis of Web-Applications // CEUR Workshop Proceedings - 2020, Vol. 2590, pp. 1-9
2. Korenkov I., Loginov I., Dergachev A., Lazdin A. Declarative target architecture definition for data-driven development toolchain // 18th International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM-2018 - 2018, Vol. 18, No. 2.1, pp. 271-278

Articles in the journals included into the List of Higher Attestation

Commission

3. Korenkov I.D. Method of domain-specific analysis of source texts of programs based on semantic models // Scientific and Technical Volga region Bulletin - 2020 - № 7 - pp. 32-37
4. Dergachev A.M., Sadyrin D.S., Ilina A.G., Loginov I.P., Korenkov I.D. Methods and means for detecting vulnerabilities in dynamic memory allocators of the glibc library // Scientific and Technical Volga region Bulletin - 2020. - № 1. - pp. 79-83

5. Korenkov I.D., Loginov I.P. Research and development of language workbench based on PEG-grammar // Journal of Instrument Engineering - 2015. - Vol. 58. - № 11. - pp. 934-938

Other publications

6. Loginov I.P., Pavlova E.V., Korenkov I.D. Research and implementation of the parsing algorithm // Collection of abstracts of the congress of young scientists. Electronic edition. - SPb: ITMO University, 2020. Access mode: <https://kmu.itmo.ru/digests/article/3956> - 2020
7. Korenkov I.D. Iterative transformation of semantic models of programs // Collection of abstracts of the congress of young scientists. Electronic edition. - SPb: ITMO University, 2020. – Access mode: <https://kmu.itmo.ru/digests/article/4180> - 2020

ВВЕДЕНИЕ

Актуальность темы. В области разработки программ и программных систем большое значение имеет инструментальное программное обеспечение, предназначенное для поддержки процесса разработки. Основным инструментом является интегрированная среда разработки (Integrated Development Environment, IDE) [22, 64, 97] – комплекс программ, предоставляющий разработчику средства редактирования исходного кода, контроля версий, отладки, трансляции, организации командной работы и многие другие возможности.

Одной из основных функций интегрированных сред разработки является редактирование исходного кода программ. Современные текстовые редакторы способны частично автоматизировать процесс разработки, что существенно повышает производительность труда программиста. Возможности текстовых редакторов позволяют выделять цветом синтаксические элементы (т.н. «подсветка синтаксиса» от англ. «syntax highlight») [13], уведомлять программиста об ошибках, автоматически дополнять синтаксические конструкции при наборе [52], выполнять навигацию по исходному коду, визуализировать структурную организацию программы.

Особенностью современной индустрии разработки программного обеспечения является применение нескольких языков программирования в одном программном проекте [70]. Данная особенность порождает проблему проверки согласованности компонентов, реализованных на разных языках и применяемых в составе одной программной системы [19]. Как правило, несогласованность компонентов отслеживается только на этапе отладки или тестирования. Например, если в программе на языке C# [40] используется функция, реализованная в отдельном модуле на языке Си, и сигнатура этой функции изменилась, то отследить возникшую вследствие этого ошибку в программной системе возможно только во время выполнения. Такое состояния дел существенно снижает эффективность процесса разработки.

Данная работа направлена на развитие подходов в области анализа исходного кода программ, написанных на нескольких языках программирования, с целью выявления подобных ошибок на этапе кодирования, что сократит количество циклов редактирования-отладки-тестирования и в целом положительно повлияет на процесс разработки программного продукта.

Степень разработанности проблемы. Большой вклад в теорию формальных языков и грамматик внесли Ноам Хомский [6, 20, 21], Стивен Клини [60, 61], Масару Томита [67, 89, 90], Джей Эрли [37], Вон Пратт [74]; в развитие реляционных моделей и отображений – Эдгар Кодд [23, 24, 25, 26], Кристофер Дейт [32, 33]; в развитие семантических технологий и метамоделирование – Йорик Уилкс [89, 99].

Практический вклад в область инструментального программного обеспечения вносят компании Microsoft, Oracle, IBM, JetBrains [73] и другие. Передовыми решениями в области данной темы являются среди разработки предметно-ориентированных языков, такие как JetBrains MPS, Eclipse Xtext [42], которые не решают проблемы в полной мере и обладают рядом недостатков:

- 1) не имеют возможности выполнения совместного анализа моделей программ, написанных с использованием одновременно нескольких языков;
- 2) не имеют возможности выполнения семантического анализа во время редактирования текста;
- 3) ограничены в применении собственной средой разработки (например, Racket [43], Kermeta [54], Eclipse Xtext);
- 4) не выполняют анализ исходного кода, написанного на алгоритмических языках программирования (например, JetBrains MPS).

Объект исследования – исходные тексты и семантические модели программ, написанных с использованием нескольких языков программирования.

Предмет исследования – методы и алгоритмы анализа исходных текстов программ в процессе их редактирования с использованием семантических моделей.

Цель диссертационной работы – выявление ошибок в программном коде, написанном с использованием нескольких языков программирования, на этапе редактирования исходных текстов программ.

В соответствии с целью в диссертационной работе ставятся и решаются следующие **задачи**.

1. Анализ способов внутреннего представления и обработки редактируемого текста программ в интегрированных средах разработки с целью выявления недостатков существующих средств поддержки мультиязыковых программных проектов.
2. Разработка метода предметно-ориентированного анализа исходных текстов программ в процессе редактирования на основе семантических моделей.
3. Разработка структур данных и программной архитектуры для внутреннего представления редактируемого текста и семантических моделей программ.
4. Разработка и реализация программного средства анализа исходных текстов программ, апробация разработанного метода.

Методы исследования. При решении поставленных задач применялись методы объектно-ориентированного анализа, методы синтаксического и семантического анализа, метамоделирование, методы языково-ориентированного программирования.

Научная новизна работы. В диссертации получены следующие результаты, характеризующиеся научной новизной.

1. Метод предметно-ориентированного анализа исходных текстов программ, отличающийся от существующих возможностью выполнять анализ текстов программ, разрабатываемых с применением нескольких языков программирования.
2. Язык описания семантических моделей программ, концептуально отличающийся от существующих совместным применением реляционного представления и представления в виде графа, позволяющий адаптировать алгоритмы анализа семантических моделей программ к специфике конкретных программных проектов.
3. Алгоритм итеративных преобразований семантических моделей программ, использующий модифицированные структуры данных для внутреннего представления текста в ходе его редактирования и анализа, что позволяет уменьшить время отклика среды разработки или текстового редактора между вводом пользователя и отображением результатов анализа актуального состояния программного кода.

Практическая значимость работы.

1. Программная архитектура для представления динамических семантических моделей программ, позволяющая интегрировать реализацию разработанного метода предметно-ориентированного анализа в различные среды разработки или текстовые редакторы.
2. Анализатор исходных текстов программ, реализованный в виде программного компонента, допускающего его интеграцию с существующими средами разработки, прошедший апробацию в составе реализованного конфигурируемого текстового редактора при решении практической задачи анализа текстов программ, разрабатываемых с использованием нескольких языков программирования.

Теоретическая значимость работы состоит в развитии теории формальных языков и грамматик в части согласованного анализа текстов

программ, написанных с использованием нескольких языков программирования.

Основные положения, выносимые на защиту.

1. Метод предметно-ориентированного анализа исходных текстов программ, разрабатываемых с применением нескольких языков программирования.
2. Язык описания семантических моделей программ, позволяющий настраивать алгоритмы анализа текста под конкретные программные проекты.
3. Алгоритм реактивного анализа и итеративного преобразования семантических моделей программ.

Апробация результатов исследования. Основные положения диссертационной работы и результаты исследований докладывались на одиннадцати всероссийских и международных конференциях. Среди них VIII, IX конгресс молодых ученых Университета ИТМО (КМУ) (2019, 2020 гг.), XLVI – XLIX научная и учебно-методическая конференция Университета ИТМО (2017-2020 гг.), IX Научно-практическая конференция молодых ученых «Вычислительные системы и сети (Майоровские чтения 2017)», 17th FRUCT Conference.

Обоснованность и достоверность научных положений и результатов обеспечены полнотой анализа теоретических и практических исследований, докладами и обсуждениями на конференциях и научных конгрессах, практической проверкой и внедрением полученных результатов.

Внедрение результатов исследования. Результаты исследования использованы в рамках выполнения НИР-ФУНД № 619296 «Разработка методов создания и внедрения киберфизических систем», внедрены в учебный процесс на факультете программной инженерии и компьютерной техники Университета ИТМО, прошли апробацию в ООО "Тюн-ит" при решении практической задачи анализа текстов программ, разрабатываемых с

использованием нескольких языков программирования, на наличие потенциальных уязвимостей, выявляемых на уровне исходного кода, что подтверждается актами о внедрении.

Публикации результатов исследования. По теме диссертации опубликовано семь основных работ, из них три статьи в журналах из перечня рецензируемых научных изданий, в которых должны быть опубликованы основные научные результаты диссертаций на соискание ученой степени кандидата и доктора наук, и две статьи в изданиях, индексируемых Scopus или Web of Science.

Личный вклад. Основные результаты, представленные в диссертации, получены лично автором. Из работ, выполненных в соавторстве, в диссертацию включены результаты, имеющие личный вклад автора по теме диссертации или смежным темам, таким как синтаксический анализ. Соавтором Дергачевым А.М. осуществлялось общее руководство исследованием, а также руководство работами по технической реализации предложенных средств и методов.

1. В работе «Исследование и разработка языкового инструментария на основе PEG-грамматики» [4] автором полностью выполнена разработка программного расширения IDE Visual Studio, включающего модифицированный текстовый редактор AvalonEdit, выполняющий подсветку синтаксиса на основе разработанного синтаксического анализатора и динамически изменяемых грамматик. Логиновым И.П. осуществлялось тестирование предложенного языкового инструментария и разработка грамматик тестовых языков. Лаздиным А.В. осуществлялось сравнение разработанной функциональности с существующими аналогами.
2. В работе «Declarative target architecture definition for data-driven development toolchain» [63] автором разработан синтаксический и семантический анализатор для языка описания систем команд,

предложенного Логиновым И.П. Соавтором Лаздиным А.В. осуществлялось исследование возможностей описания архитектур процессоров средствами компиляторов GCC и LLVM.

3. В работе «Application of Graph Databases for Static Code Analysis of Web-Applications» [81] автором реализован анализ синтаксического дерева на предмет использования определенной функциональности, приводящей к эксплуатации уязвимостей веб-приложений. Соавтор Садырин Д.С. выполнял обзор средств, применяемых в статическом анализе кода. Логинов И.П. реализовал генерацию отчетов с результатами анализа. Ильина А.Г. выполняла исследование графовых поисковых запросов в терминах формальных языков.
4. В работе «Исследование и реализация алгоритма синтаксического анализа» [5] автором предложено использование алгоритма «Магра» для построения адаптируемых синтаксических анализаторов и создан прототип его реализации для среды на базе Common Language Infrastructure. Павлова Е.В. осуществляла программную реализацию синтаксического анализатора на основе алгоритма «Магра». Логинов И.П. выполнял разработку тестовых предметно-ориентированных языков.
5. В работе «Итеративное преобразование семантических моделей программ» [2] автором предложен метод итеративного преобразования семантических моделей программ, применимый для предметно-ориентированного анализа исходных текстов программ, разработан алгоритм реактивного анализа и итеративных преобразований семантических моделей программ, выполнена экспериментальная программная реализации.
6. В работе «Метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей» [3] автором предложен язык описания спецификаций семантических моделей

программ и их отображений, а также описания синтаксических моделей исходных текстов программ для построения абстрактных семантических графов в ходе семантического анализа текстов программ.

7. В работе «Методы и средства обнаружения уязвимостей аллокаторов динамической памяти библиотеки glibc» [1] автором выполнен анализ характера уязвимостей и исследование возможности их выявления на уровне написания исходного кода программ, разрабатываемых с применением языков C/C++ и ассемблера. Ильина А.Г. выполняла обзор методов верификации программ для проверки корректности во время их выполнения. Садырин Д.С. описал подход, позволяющий совместить символьное выполнение программы с реальным исполнением процессором ее машинного кода. Логинов И.П. выполнял практическое исследование особенностей средств анализа кода и механизмов динамического распределения памяти.

ГЛАВА 1 Задачи и проблемы реализации анализа исходных текстов программ

1.1 Задачи семантического анализа и семантические модели программ

В данной работе понятие семантика рассматривается применительно к языкам программирования, трансляторам и анализаторам исходных текстов программ, как способ представления конструкций языков программирования посредством формальных математических моделей, для построения которых может использоваться аппарат математической логики, λ -исчисление, теория графов, теория множеств, реляционная алгебра. Если лексический и синтаксический анализ имеют дело со структурными особенностями программы – внешними текстовыми конструкциями языка, то семантика ориентирована на содержательную интерпретацию «смысла» объектов, описанных в программе, таких как типы данных, переменные, функции, и во внутреннем представлении выглядит как система взаимосвязанных объектов. Например, в языках программирования Си/C++ переменная «ссылается» на описание типа данных, а производный тип данных «ссылается» на базовый тип. Для каждого множества объектов одного типа создаются так называемые семантические таблицы. При этом элементы различных таблиц связаны между собой (объекты разного типа «ссылаются» друг на друга) и логически образуют структуру, подобную базе данных, схема которой описывается в терминах предметной области языка программирования. Такое внутреннее представление семантики программы в виде множества именованных объектов, с которыми работает программа, с описанием их свойств, характеристик и связей называется семантической моделью программы.

Таким образом, для реализации трансляторов или анализаторов любого языка программирования помимо синтаксиса имеет значение семантика, а именно семантическая модель программы, формируемая исходным кодом на данном языке программирования. Семантический анализ определяет, имеет ли

созданная в тексте программы синтаксическая структура допустимое значение, и выявляет такие ошибки, как несоответствие типов и параметров, использование зарезервированных слов, многократное объявление переменных в области видимости, попытка доступа к переменным вне области видимости и другие. Для программных проектов, в которых задействовано несколько языков программирования, имеет значение согласованность между частями программного кода, написанными на различных языках. Например, для веб-приложения, реализованного в соответствии с архитектурой SPA (Single-Page Application) [72], имеет значение согласованность программного интерфейса (Application Programming Interface, API) веб-сервиса, написанного на языке программирования Java, с клиентским веб-приложением, написанным на языке JavaScript, которое взаимодействует с данным веб-сервисом. Для анализа такой согласованности необходимо иметь правила отображения объектов семантической модели в терминах одного языка программирования на объекты семантической модели в терминах другого языка программирования или объектно-ориентированного языка.

Можно выделить два основных сценария выполнения такого анализа. Первый сценарий подразумевает выполнение анализа на этапе сборки проекта или в процессе статического анализа кода программы отдельным автономным инструментом – анализатором [68]. Второй – выполнение анализа в процессе редактирования текста программы текстовым редактором или интегрированной средой разработки. В первом случае результатом работы анализатора является отчёт о структуре семантической модели со статусом проверяемых в ходе анализа ограничений и списком несоответствия между моделью программы, построенной на основе её текста, и заданными параметрами анализа. Для применения результатов анализа необходима новая итерация – повторное редактирование текста программы, в процессе которого могут быть внесены некорректные изменения, с последующим повторным анализом и получением нового отчета. Во втором случае результатом работы

анализатора является визуальная обратная связь с разработчиком, редактирующим код программы, которая заключается в подсветке синтаксиса, всплывающих подсказках и диалоговых окнах среды разработки. Второй сценарий является предпочтительным, так как результаты анализа, выполняемого в ответ на каждое внесённое в текст программы изменение, визуализируются в процессе редактирования текста программы, указывая на место в программном коде, требующее внимания разработчика.

Таким образом, можно выделить два основных подхода к организации анализа семантических моделей программ:

- 1) последовательное построение всех необходимых семантических моделей на основе текста программы с последующим анализом согласованности;
- 2) гранулярное построение семантических моделей с одновременным анализом их совместимости и корректности по мере построения в процессе редактирования текста программы.

Рассмотрим некоторые способы представления семантической информации о программах. Перед этим перечислим ряд задач, решение которых требует использования семантической информации.

1. Создание компиляторов, неотъемлемой частью которых является анализатор текстов программ [14], порождающий артефакты, содержащие информацию о программе, её алгоритмах и структурах данных в форме, отличающейся от исходного текста.
2. Создание интерпретаторов и динамических сред выполнения для языков высокого уровня [75, 79].
3. Создание инструментов статического анализа и верификации программ, анализирующих тексты программ на соответствие спецификациям [41] и ограничениям. Такие инструменты не входят в состав текстовых редакторов и интегрированных сред разработки. Данные задачи связаны с переформулированием информации об анализируемой

программе или алгоритме из предметной области исходного языка в предметную область анализа. Последующий анализ выполняется в терминах и понятиях специальной модели, не связанной с исходным языком или программным проектом.

4. Создание интегрированных сред разработки, предусматривающих возможности автоматизации действий разработчика, таких как подсветка синтаксиса, автодополнение, элементы статического анализа – операции над текстом программы и проверка его корректности относительно некоторых ограничений в процессе редактирования.

Решение вышеперечисленных задач связано с необходимостью формализованного представления семантической информации, извлекаемой из текста программы, и требует от разработчика понимания онтологии предметной области – концептуального описания множества объектов и связей между ними в терминах конкретного языка программирования. Это необходимо для построения структур данных, вмещающих семантическую информацию.

Для представления семантической информации о тексте программы могут использоваться различные способы представления данных. Это зависит от конкретных требований программного проекта, выбранного варианта программной архитектуры и используемого инструментария.

1. Специализированные структуры данных [100, 12] и объектные модели, элементы которых соответствуют элементам предметной области текста программы. Элементы семантической информации в данном случае представлены в памяти экземплярами конкретных сущностей, например, структур или классов.
2. Обобщённые структуры данных и способы их представления, изначально разработанные для представления различных моделей данных, таких как реляционные или в виде графа [53]. Элементы

соответствующего представления данных (таблицы или графы) могут размещаться как внутри программы, так и во внешних базах данных.

Во втором случае возможно использование как представлений, опирающихся в явном виде на схему данных [11], так и материализованных представлений, которые отличаются материализацией данных в форме структур, внутренняя организация которых строится на основе онтологии предметной области и соответствующей семантической информации [9].

При решении конкретной задачи независимо от выбранного представления (объектная модель, обобщённая модель в виде графа или реляционная модель [82]), объём семантической информации будет один и тот же. На уровне смысловых связей он будет идентичен в силу требований самой задачи. Это не зависит от использования явных ссылок между объектами в структуре данных, между элементами реляционных отношений или отношений сущностей семантической сети. При этом для разработчика, заинтересованного в семантической информации, важен не столько способ её представления, сколько способ описания допустимых над семантической информацией действий. Способ представления семантической информации влияет на алгоритмы семантического анализа, а способы описания спецификаций, шагов анализа и ограничений влияют на возможности применения разработчиком анализатора для анализа семантической информации, извлекаемой из текста программы.

Теперь рассмотрим этапы жизненного цикла программного обеспечения (ПО), на которых может понадобиться анализ исходных текстов программ. Центральное место в процессе разработки ПО занимает написание самого текста программы [51], а также сборка программного проекта из исходных текстов. На рисунке 1 показаны важные этапы жизненного цикла ПО [76], в которых затрагивается текст программы. В частности, к ним относятся тестирование и исправление обнаруженных ошибок [7].

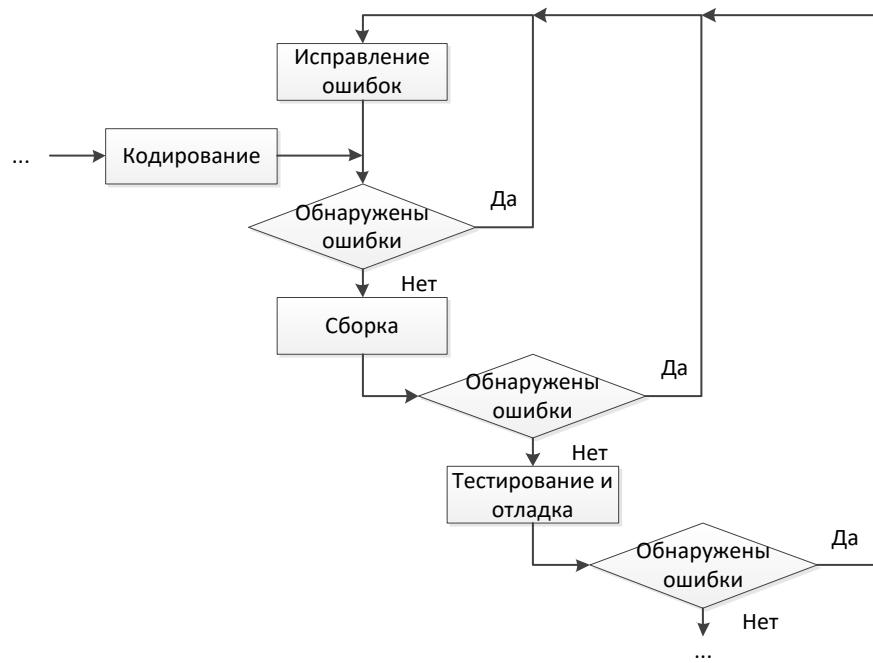


Рисунок 1 – Фрагмент жизненного цикла программного проекта

Каждый из этих этапов неразрывно связан с выявлением и исправлением ошибок в тексте программы. На этапе написания текста программы для этого применяются средства, интегрированные в среды разработки и анализирующие текст программы в процессе редактирования. На этапе сборки проекта для этого применяются трансляторы, в частности компиляторы или генераторы кода [22], анализирующие текст программы в процессе сборки. Далее следует этап тестирования, в ходе которого выявляются ошибки, связанные с влиянием структуры программы на её функциональность, являющуюся предметом тестирования. Независимо от способа организации процесса разработки ПО процесс тестирования требует предварительной сборки программного проекта, чему в свое время предшествует написание исходного кода, а значит, цикл выявления и исправления ошибок связан с несколькими различными этапами жизненного цикла ПО, что требует различного времени исправления ошибок в зависимости от этапа их выявления. Чем раньше ошибка будет выявлена и исправлена, тем меньше времени займёт разработка проекта в целом.

В программных проектах появление ошибок, связанных с написанием программного кода на нескольких языках программирования, зависит от согласованности элементов текста программы, отвечающих за передачу данных или управления между частями программы, написанными на разных языках. Например, при вызове из программы, написанной на языке C#, функции, написанной на языке Си, сигнатура этой функции должна быть отражена по строго определённым правилам, гарантирующим корректную обработку такого вызова посредством механизмов `p-invoke` [8]. Это справедливо и при наличии дополнительных требований к организации исходного текста программы. Например, использование автоматизированных средств сериализации [18] может требовать дополнительной разметки описаний типов данных специальными атрибутами, наличия конструкторов по умолчанию и т.п.

То же самое относится и к случаям использования предметно-ориентированных языков (Domain Specific Language – DSL) как встроенных [49], так и внешних [55]. Встроенные предметно-ориентированные языки могут накладывать существенные ограничения на корректность текста программы с точки зрения его структуры. Такие ограничения не проверяются средствами разработки, ориентированными на семантический анализ исходного кода, написанного на языке программирования общего назначения. Разработка и анализ внешних DSL на сегодняшний день возможны посредством создания специализированных анализаторов или посредством использования средств проектирования языков и языковых анализаторов [10, 50, 65]. Они также обладают рядом ограничений по части поддержки интеграции в существующие процессы разработки.

Таким образом, процедура анализа исходных текстов программ специфична для каждого программного проекта, а именно зависит от состава используемых в проекте языков программирования общего назначения и предметно-ориентированных языков. Также на процедуру анализа влияет

способ представления элементов прикладной предметной области в исходном коде. Для адаптации процедуры анализа под особенности программных проектов необходим настраиваемый инструментарий, отвечающий за семантический анализ. В рамках выполнения данной работы вопрос анализа рассматривается шире и представлен как предметно-ориентированный анализ текстов программ или семантических моделей программ с учетом требований и ограничений, накладываемых семантикой программных проектов.

1.2 Исследование существующих подходов к семантическому анализу текстов программ

При исследовании существующих подходов к семантическому анализу текстов программ был рассмотрен ряд научных работ и инженерных решений.

В статье [93] сравниваются три подхода к реализации внешних и встроенных предметно-ориентированных языков, ни один из которых не предполагает возможности проверки корректности семантической информации, извлекаемой из анализируемых текстов в процессе их редактирования.

В работе [87] рассматривается проверка корректности предметно-ориентированных моделей посредством их трансляции в модели Event-B с одновременным анализом специализированными инструментами. Такой подход исключает возможность проверки корректности анализируемого текста в процессе его редактирования. Он также требует, чтобы все анализируемые аспекты текста программы или программной системы могли быть сформулированы в терминах Event-B моделей. Все остальные используемые в программном проекте языки программирования (в том числе общего назначения) должны быть отображены в те же самые Event-B модели, что в общем случае невозможно.

В работе [57] изложен метамодельный подход для представления информации о предметных областях. Предлагается использование

специального языка КМЗ для описания метамоделей анализируемых языков, включая их семантику, и дальнейшее отображение собранной из исходных текстов информации в тексты на других языках. Такая трансляция выполняется с помощью внешних по отношению к редактору инструментов и подразумевает дальний анализ целевых моделей независимым от исходных языков способом, что делает невозможным связывание информации об ошибках с первичным исходным текстом.

В статье [66] также рассматривается использование языка КМЗ и предлагается использование взаимных отображений между интересующими предметными областями и установление логических связей между их моделями, смысловая нагрузка которых зависит от потребностей анализа. При этом рассматриваемый инструментарий ограничен применением в составе Eclipse Modeling Framework и предполагает проверку согласованности моделей текстов на различных языках друг относительно друга как отдельный шаг в процессе построения проекта, а не в процессе их редактирования, так как требует поочередного выполнения всех модельных трансформаций, включающих полный анализ и генерацию промежуточных текстов.

В работе [79] рассматриваются некоторые способы описания семантики языков программирования с точки зрения задачи интерпретации текстов на этих языках. В случаях, когда рассматриваемый способ (например, с использованием XSemantics) подразумевает возможность создания семантических анализаторов для проверки текстов, генеративный подход исключает возможность совместной проверки текстов на различных языках, для комбинации которых не был создан специализированный анализатор.

В работе [16] показан подход на основе семантической метамодели с последовательным построением синтаксической модели на основе текста, но не на основе дерева разбора. Генерация модели выполняется в рамках заданной метамоделью комбинации языков программирования, что исключает

возможность адаптации созданного анализатора к новым комбинациям анализируемых языков.

В работе [80] описывается подход к анализу операционных и трансляционных семантик языков программирования, предназначенных для описания алгоритмов. Представленный метод действует в рамках Eclipse Modeling Framework, что дополнительно ограничивает способы его использования также, как и для КМ3 [66].

В статье [84] рассматривается анализ текста программ в процессе редактирования для редактора, не являющегося текстовым в рамках специализированной среды разработки. Редактор ориентирован на синтаксис конкретного языка, что полностью исключает возможность применения данного подхода к анализу текстов на языках, не созданных изначально именно для такой среды разработки.

В статье [43] показывается применение динамической семантической трансляции с целью извлечения информации из реляционных источников данных. В качестве источника данных рассматривается полноценная реляционная база данных, что не целесообразно для текстового редактора.

В статье [87] обсуждается использование способа представления информации о структуре предметно-ориентированных языков в виде графа и применение семантических отображений для трансформации семантических моделей. Анализ экземпляров этих семантических моделей не рассматривается.

В работе [36] показан подход, сочетающий инкрементальное свободное редактирование текста в пределах области, соответствующей одному языку. Композиция языков при этом обрабатывается за счет ручного управления моделью. Это полностью исключает возможность применения данного подхода к анализу текстов, сочетающих несколько языков. Рассматривается использование метода разделения структур данных между представлениями

текста различных уровней, но построение семантических моделей и их анализ не рассматривается.

В работе [59] рассматриваются вопросы проверки корректности текстов на предметно-ориентированных языках посредством генерирования на их основе специализированных моделей и применения к этим моделям методов формального анализа. Рассматривается исключительно генеративный подход. При этом вопрос анализа текстов, написанных с использованием нескольких языков программирования, не рассматривается. Также не рассматривается вопрос динамической трансформации моделей и поддержка процесса работы с текстом в целом.

В работе [27] также обсуждается использование трансляционных и генеративных подходов для построения интерпретируемых моделей процессов или алгоритмов на языках, обладающих операционной семантикой. Поддержка процесса работы с текстом не рассматривается.

В работе [92] описывается применение императивно заданных проверок к предметно-ориентированным моделям. Вся реализация функциональности, связанной с анализом текстов, рассматривается на базе императивных функций обратного вызова. То же относится и к межмодельным преобразованиям. Они рассматриваются не декларативно, а императивно, что ограничивает способы их выполнения. Помимо этого, данное решение по природе своей организации тесно связано с IDE Eclipse, что сильно ограничивает его применение.

Статья [38], представляющая собой сравнение ряда различных инструментариев для разработки и поддержки предметно-ориентированных языков, являясь примером хорошей аналитической работы, игнорирует существенные аспекты. Так, например, утверждается, что среда JetBrains MPS поддерживает модель свободного редактирования текста, что не соответствует действительности. Не рассматриваются также и способы реализации сравниваемой функциональности, что может оказывать значительное влияние

на пользовательский опыт и эффективность использования инструмента. Игнорируются возможности использования созданных языков и поддержки работы с ними за пределами собственных сред разработки.

В статье [87] обсуждается использование шаблонов спецификации для трансляции семантики выполнения предметно-ориентированных языков с целью оценки поведения. Вопросы сочетания нескольких языков и извлечения полезной в процессе редактирования текстов информации для поддержки процесса разработки не рассматриваются.

Статья [96] рассматривает способы построения редакторов, управляемых спецификациями моделей, которым должен подчиняться редактируемый текст. С точки зрения работы пользователя с редактором, выделяется два подхода. Первый – подход на основе фиксированного набора визуальных шаблонов, в форме иерархии которых пользователю предоставляется возможность манипуляций в терминах структуры объектной модели. Второй – подход на основе текстовых распознавателей (синтаксических анализаторов), где пользователю для работы предлагается традиционное текстовое представление, не ограниченное визуально и структурно. Внутренняя модель при этом строится в ходе анализа редактируемого текста отдельно от процесса редактирования. Рассматриваются также пути совмещения этих способов, но не рассматривается пути поддержания актуальности семантических моделей редактируемого текста, необходимых для обеспечения функциональности сред разработки.

Статья [48] рассматривает инкрементальный семантический анализ в разрезе развития подходов на основе атрибутных грамматик. Приведён достаточно подробный анализ возможных ситуаций при редактировании текста программы, влияющих на структуру семантической модели, выводимой из анализируемого текста. Рассматриваемый здесь подход предполагает большой набор взаимосвязанных алгоритмов и структур

данных, что осложняет разработку и поддержку такого анализатора. Кроме того, конфигурирование такой системы становится нетривиальной задачей, если в роли разработчика предметно-ориентированного языка, анализ которого будет выполняться, выступает не специалист в данной области.

В статье [77] показывается в общем случае идея инкрементального семантического анализа для текстовых редакторов, управляемых атрибутными грамматиками с деревом в качестве семантической модели. При этом рассматривается семантический анализ в контексте редактирования текста, ограниченного моделью синтаксиса, но не свободного редактирования текста программы. На практике семантические модели программ представляются в виде произвольных объектных моделей, структура которых ограничена схемой семантических отношений между элементами языка программирования.

В статье [39] рассматривается отображение данных из табличной формы в RDF представление (Resource Description Framework), являющееся семантической моделью. Таким образом, сущность из состава модели данных, описывающих программу, может быть отображена в семантическую сеть. В качестве таких сущностей могут выступать элементы таблиц символов, структуры узлов синтаксических деревьев, и так далее.

Статья [9] расширяет вышеописанное представление до возможности отображения сложных реляционных схем на семантические сети. Это позволяет говорить о возможности описания ограничений над графиками, представляющими семантические сети, в терминах реляционных моделей. А так как, по существу, схема реляционной сущности является аналогом схемы данных, ассоциируемых с узлом семантической сети, это также дает возможность говорить об отображениях между семантическими сетями, формулируя их в терминах реляционных моделей, а значит, и применять к ним те же алгоритмические подходы.

Работа [28] в значительной степени покрывает историю развития техник инкрементального семантического анализа. Несмотря на то, что рассматривается редактирование анализируемого текста в свободной форме, отношения в семантической модели рассматриваются в пределах дерева атрибутированных узлов, что нецелесообразно для представления семантических моделей программ в терминах предметных областей, отличных от исходного языка.

Работа [17] показывает, как выражения условий над графами семантических сетей могут быть интерпретированы в терминах реляционных множеств. Из этого также следует возможность автоматической декомпозиции запросов над графиком семантической сети для создания эффективной реализации триггеров, позволяющих обнаруживать факт изменения результата такого запроса вследствие модификации узла семантической модели.

Работы [56, 78, 95] как и многие другие, рассматривают инкрементальный анализ текстов на основе атрибутированных деревьев, что также не отвечает потребностям анализа предметно-ориентированных семантических моделей, получаемых на основе текстов программ, так как структура этих деревьев диктуется исходным языком.

Следует уточнить, что предметом исследования являются именно способы выполнения анализа семантической информации, полученной на основе текстов программ, а не только синтаксических структур в исходном тексте, что исключает из рассмотрения средства генерации парсеров, требующие самостоятельной реализации логики семантического анализа.

Проведенное исследование показало, что представленные в рассмотренных работах средства анализа могут быть разбиты на три группы:

- 1) не предназначенные для использования в реальном времени в процессе редактирования текста;

- 2) предназначенные для использования в процессе редактирования текста программы внутри специализированной среды разработки или специализированного режима редактирования;
- 3) предназначенные для использования в процессе редактирования моделей программ в терминах, близких к модели абстрактного синтаксического дерева, не рассматривающие анализ текстов программ.

Таким образом, ни один из существующих инструментов, предназначенных для выполнения семантического анализа, не может быть встроен в интегрированные среды разработки для выполнения анализа текстов программ в процессе их редактирования, если изначально он не создавался для этой интегрированной среды разработки. Кроме того, написание спецификаций, задающих правила семантического анализа, во всех случаях предполагает изучения специализированных предметных областей, таких как языки и среды метамоделирования, синтаксический и семантический анализ, онтологическое моделирование. Вывод – решение задачи разработки анализатора, необходимого для конкретного программного проекта, требует от разработчика специальной подготовки и отвлекает от работы по основному проекту.

1.3 Исследование особенностей реализации анализаторов в составе интегрированных сред разработки

Для исследования особенностей программной реализации существующих подходов к анализу текстов программ, применяющихся в современных инструментальных средствах разработки программного обеспечения, были выбраны наиболее популярные по данным Google Trends интегрированные среды разработки, основные из которых представлены в таблице 1.

Таблица 1 – Популярные IDE

	Название IDE	Сайт проекта
1	Visual Studio	https://visualstudio.microsoft.com/ru/
2	Eclipse	https://www.eclipse.org/
3	Visual Studio Code	https://code.visualstudio.com/
4	IDEA	https://www.jetbrains.com/idea/
5	NetBeans	https://netbeans.org/
6	Xcode	https://developer.apple.com/xcode/

Все эти среды разработки обладают техническими возможностями интеграции семантических анализаторов и могут использовать результаты работы анализаторов, предоставляя разработчику функциональные возможности, частично автоматизирующие процесс разработки программного обеспечения. Реализация таких функциональных возможностей в различных средах разработки отличается особенностями программной архитектуры и способами внутреннего представления текстов программ. К таким возможностям относятся:

- 1) подсветка синтаксиса – отображение текста с использованием различных стилей в зависимости от классификации лексем, синтаксических конструкций и окружающего контекста;
- 2) навигация по тексту программ с помощью специальных команд, таких как «переход к определению» или «переход к использованию идентификатора»;
- 3) форматирование – расстановка пробелов и распределение текста по строкам в зависимости от рода синтаксических конструкций и их назначения;
- 4) автодополнение – подсказки по возможностям корректного ввода в тех или иных позициях текста, опирающиеся на информацию о структуре контекста и трактовку его элементов;

- 5) контекстные подсказки, содержащие различные сообщения, например, информацию об ошибках, ассоциируемую с тем или иным элементом структуры текста;
- 6) проверки и подсказки по стилю кодирования в зависимости от классификации лексем, синтаксических конструкций и окружающего контекста.

Во всех случаях единственным способом включения в процесс редактирования специфических для отдельного программного проекта проверок исходного кода является реализация расширения среды разработки. Единственным способом, автоматически решающим вопрос совместимости двух частей программного проекта, является генерация кода этих частей на основе общей спецификации. Это является эквивалентом создания специализированного расширения интегрированной среды разработки, но без возможности анализа текстов программ в процессе редактирования. Реализация такой возможности осуществима при условии интеграции создаваемого расширения с текстовым редактором интегрированной средой разработки для совместного использования семантической информации, что в существующих средах разработки либо не предусмотрено, либо затруднено.

Таким образом, по результатам исследования возможностей семантических анализаторов наиболее популярных по данным Google Trends интегрированных сред разработки были выделены пять групп функциональности.

1. Языковая поддержка – поддержка разработки на различных языках программирования.
2. Управление процессом анализа – возможность настройки встроенных анализаторов.
3. Организация доступа к моделям программ – возможность использования информации о структуре программ анализаторами.

4. Работа с мультиязыковыми программами – возможность встроенных анализаторов выполнять анализ текстов программ, написанных с использованием нескольких языков программирования.
5. Расширяемость системы – возможность создания расширений, функциональность которых требует модификации внутренней модели представления данных.

В результате анализа документации и исследования архитектуры программных интерфейсов, были получены сравнительные данные, представленные с оценками по пятибалльной шкале в таблице 2. Данная оценка показывает сравнительную функциональность той или иной группы.

Таблица 2 – Оценка функциональности, связанной с семантическим анализом текста программ в ходе их редактирования

Категории программных средств	Инструментальное средство	Группа функциональности					Суммарная оценка
		1	2	3	4	5	
Интегрированные среды разработки	Visual Studio 15.1	5	4	2	2	4	17
	IntelliJ IDEA 2017.2	5	4	0	3	4	16
Дополнения к средам и редакторам	ReSharper 2018.1	2	4	0	2	0	8
	YouCompleteMe	4	4	3	0	3	14
Гибридные решения	OmniSharp 1.31 (Roslyn)	1	4	4	2	4	15
	Eclipse JDT Language Server 4.10	1	4	3	2	5	15

Основным недостатком существующих инструментальных средств семантического анализа является их неприменимость для анализа исходных текстов, написанных с использованием нескольких языков программирования. Существенное влияние на решение данной задачи оказывают особенности реализации анализаторов, применяемых в составе текстовых редакторов. Данные особенности связаны с тем, каким образом происходит редактирования текста программы.

Редактирование текстов программ обычно выполняется с использованием средств текстового редактора, позволяющего вносить сложные изменения за несколько простых шагов, в результате выполнения которых текст программы остается корректным. Примером является средство автоматического рефакторинга [45]. Так, для переименования какой-либо сущности в тексте программы, например, подпрограммы, достаточно изменить имя в месте ее объявления. Новое имя в выражениях вызова подпрограммы будет заменено автоматически. Это позволяет уменьшать вероятность внесения ошибок за счет ограничения размера фрагмента текста, который редактируется вручную. При редактировании вручную необходимо проверять корректность измененного текста как можно чаще, чтобы исключить возможность внесения большого количества синтаксических ошибок, из-за которых станет сложно заметить и быстро исправить более существенные логические ошибки в тексте программы.

Можно выделить два сценария анализа исходных текстов программ:

- 1) глобальный, т.е. анализ полного объема текста программы, представленной набором файлов с исходным кодом;
- 2) локальный, т.е. анализ последовательности операций по изменению отдельных частей текста (например, добавление или удаление отдельных символов).

Глобальный анализ связан с полным повторным синтаксическим разбором текста программы для каждой итерации анализа. Исторически именно такой подход рассматривается как основа для построения трансляторов. При этом, как правило, не рассматривается повторное использование результатов предыдущего вызова анализатора. Использование такого подхода в процессе редактирования текста связано с необходимостью полного сопоставления синтаксических моделей текстов программ для выявления измененных фрагментов. Это повышает сложность анализа, распространяя его на объем всего текста программы или программной

системы в целом как для этапа синтаксического анализа, так и для этапа сопоставления синтаксических представлений при выявлении изменённых фрагментов.

При выполнении локального анализа выявление изменённых фрагментов производить не нужно, так как данные об изменениях текста программы содержат информацию о том, в какие части текста вносились изменения, а также содержание этих изменений.

В процессе анализа программной архитектуры интегрированных сред разработки были исследованы структуры данных, используемые для представления текста при реализации текстовых редакторов. Это такие структуры данных, как «Gap buffer», «Piece table», «Rope» и их неизменяемые модификации, отличающиеся от классических тем, что узлы, составляющие их структуру, не могут изменяться. Для представления сопутствующей информации о тексте применяются ассоциативные структуры данных, использующие в качестве ключей позиции в тексте, такие как «Interval Tree», буфера атрибутов, и тому подобные. Под сопутствующей информацией подразумеваются данные об областях подсветки синтаксиса, всплывающих подсказках, о стилях, используемых при отображении.

Одной из основных проблем при реализации текстовых редакторов для языков программирования является задача интеграции анализаторов, результаты работы которых редактор должен отображать. Интеграция заключается в передаче информации об исходных текстах между внутренними представлениями данных текстового редактора и анализатора. На рисунке 2 приведена общая структура взаимодействия текстового редактора и анализатора, определяющая пространства имен и уровни визуального, логического, синтаксического и семантического представления моделей исходного текста программы. Преобразование представлений данных текстового редактора осуществляется программным компонентом, отвечающим за синтаксический анализ моделей программ и анализ текста

программного кода. В ходе редактирования текста модель программы меняется на всех уровнях ее представления, а результаты анализа визуализируются в текстовом редакторе в виде подсвеченных изменений и сообщений в соответствии с набором правил и ограничений, которым должен соответствовать используемый в данном программном проекте язык программирования. Передача информации об изменении текста из редактора в анализатор и результатов анализа обратно в структуры данных текстового редактора сопряжена с накладными расходами на отображение данных между этими структурами данных.

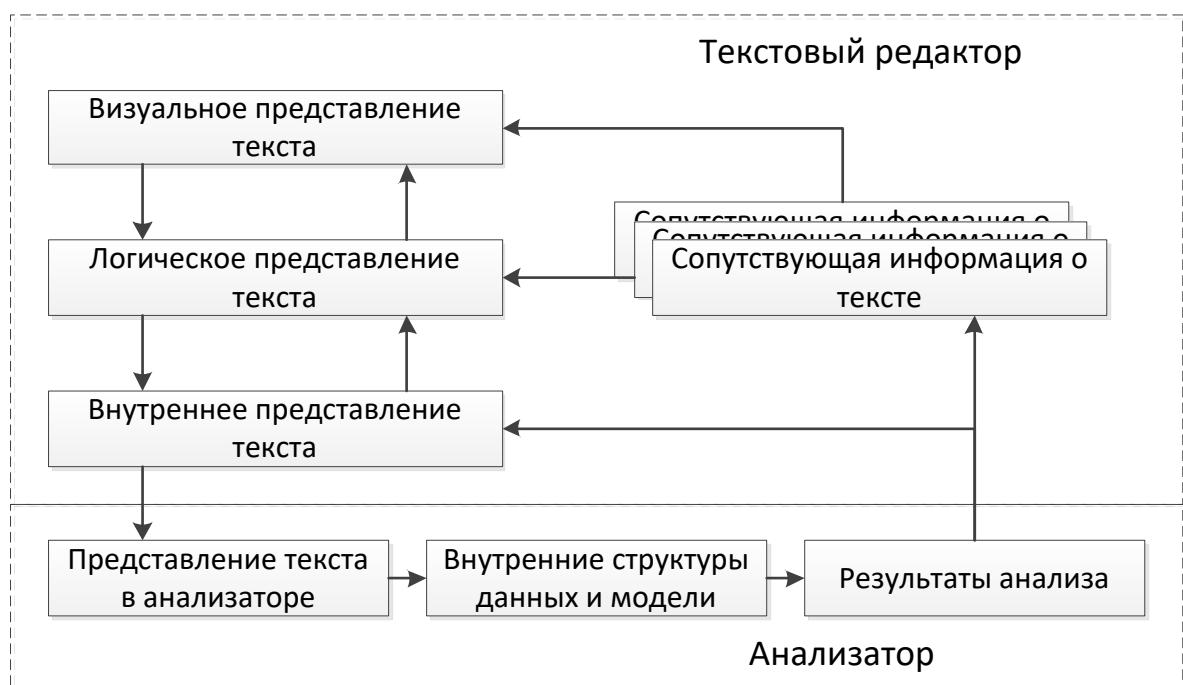


Рисунок 2 – Схема процесса передачи информации в текстовых редакторах распространённых IDE

При передаче информации в анализатор накладные расходы заключаются в следующих операциях:

- 1) формирование «снимка» состояния текста на момент начала процедуры анализа;
- 2) дублирование текстовой информации изменённого фрагмента при передаче в анализатор;

3) отображение позиции в тексте между снимками состояний различных версий при доступе анализатора к незатронутым фрагментам текста.

При обратной передаче информации из анализатора в текстовый редактор, накладные расходы заключаются в изменении структур данных, обеспечивающих визуальное представление текста. Это связано с тем, что необходимо сопоставить результаты анализа текста его местоположению в редакторе.

Можно обобщить и отметить следующие недостатки существующих решений.

1. Заранее определенный набор языков, для которых возможен семантический анализ. Добавление поддержки новых языков возможно только посредством специализированных расширений.
2. Особенности программной архитектуры анализаторов и текстовых редакторов не гарантируют согласованность внутреннего представления и визуального отображения результатов анализа текстов программ с актуальным состоянием анализируемого текста. Пример приведен на рисунке 3. Подсветка потенциальной ошибки неправильно указывает ее местоположение, а текст всплывающей подсказки не соответствует актуальному состоянию текста программы.
3. Мультиязыковой анализ ограничен встроенными сочетаниями языков и определенными типами программных проектов. Например, применение семантического анализатора языка шаблонов Razor ограничено проектами ASP.NET.
4. При мультиязыковом анализе отсутствуют настройки, доступные при анализе проектов с одним языком.
5. Использование имеющихся способов представления семантической информации для предметно-ориентированного анализа ограничено или невозможно.

```

        SmtpUseSsl = false,
        SmtpUseDefaultCredentials = true,
    };
```

`bool LearningServiceConfiguration.SmtpUseDefaultCredentials { get; set; }`

The field 'ServiceContext._disposables' is never used

```

private static readonly DisposableList _disposables = new DisposableList();

[BeforeTestRun]
public static void Setup()
{
    _disposables.Add(new MicroLearningService(TestServiceConfiguration));
}
```

Рисунок 3 – Пример некорректного анализа текста программы

В результате исследования программной архитектуры интегрированных сред разработки было выявлено, что их возможности анализа текстов программ сводятся к реализации индексирующих модулей [69], способных строить семантические модели и обрабатывать правила только одного языка программирования.

В тех случаях, когда требуется выполнять анализ текстов программ, разрабатываемых с применением нескольких языков программирования или предметно-ориентированных языков, необходимо использование нескольких семантических моделей. Примером может служить программный проект, разрабатываемый с использованием двух языков программирования – Си и С#. Изменение сигнатуры функции на языке Си сделает некорректной часть кода на языке С#, зависящую от этой же функции. Такая ошибка в нарушенной части программы может быть выявлена только в ходе тестирования разрабатываемой программной системы. Однако части программы на Си и С# могут разрабатываться разными программистами, решающими не связанные между собой задачи. Для устранения такой ошибки может понадобится дополнительное согласование между группами Си и С# разработчиков.

На сегодняшний день интегрированные среды разработки не выполняют такого анализа. Необходимо создание специализированных модулей сопряжения семантических моделей, расширяющих возможности существующих сред разработки. При этом возможности по настройке таких

расширений зачастую сильно ограничены, а совместное использование нескольких семантических моделей часто невозможно из-за особенностей программной архитектуры среды разработки, в частности в виду изолированности и ограниченности программных интерфейсов среды разработки. На рисунке 4 приведена типовая схема расширяемости интегрированных сред разработки в отношении семантического анализа текстов программ.



Рисунок 4 – Типовая схема расширяемости интегрированных сред разработки в отношении семантического анализа текстов программ

Из рисунка видно, что при добавлении очередного языка реализация анализатора существенно усложняется и является уникальной для каждой группы совмещаемых в программном проекте языков. Задача создания таких расширений является для программиста вторичной по отношению к основному программному проекту, что делает их реализацию специфичной и часто ограничивает применение таких узконаправленных расширений одним проектом, ввиду невозможности использовать для решения той же проблемы в другом программном проекте.

Таким образом, задача разработки методов и средств, позволяющих выполнять предметно-ориентированный анализ над семантическими моделями программ, является актуальной. Также актуальность данного

вопроса обусловлена отсутствием на сегодняшний день средств выполнения такого анализа непосредственно в процессе редактирования программного кода, что является одним из основных способов выявления ошибок в тексте программ.

Выводы по главе 1

В ходе исследования существующих подходов к семантическому анализу и особенностей их реализации был выполнен анализ способов внутреннего представления текстов программ в интегрированных средах разработки, анализаторах и генераторах кода, выявлены недостатки существующих средств поддержки мультиязыковой разработки, среди которых следует отметить:

- особенности программной архитектуры, приводящие к некорректной индикации результатов анализа, в части соответствия актуальному состоянию текста программы;
- способы представления текста в текстовых редакторах, препятствующие обеспечению корректности и своевременности выполняемого анализа текстов программ во время их редактирования.

Таким образом, не существует универсального инструментального средства, позволяющего выполнять семантический анализ исходных текстов программ, разрабатываемых с использованием нескольких языков программирования, а существующие специализированные инструментальные средства имеют ряд ограничений по расширению своих функциональных возможностей в силу специфики программной реализации. Это осложняет выявление ошибок в процессе редактирования исходных текстов программ. Решению этой проблемы и посвящена данная диссертационная работа, для чего на основе анализа недостатков существующих подходов был сформулирован ряд задач.

Первая задача – разработка метода предметно-ориентированного анализа исходных текстов программ на основе семантических моделей, предполагающего выполнение анализа текста программы непосредственно в процессе его редактирования. Метод должен учитывать особенности способов представления данных в текстовых редакторах и требования к гранулярному обновлению внутреннего представления семантических моделей программ. Далее необходимо обобщить данные о предметных областях применения данного метода, разработать алгоритм и формализованные средства его адаптации к требованиям различных программных проектов в соответствии с используемыми в них языками программирования или предметно-ориентированными языками. Одним из таких средств адаптации может стать язык описания семантических моделей программ, учитывающий особенности реляционного способа представления данных и представление данных в виде графа.

Следующая задача – реализация алгоритма анализа семантических моделей программ. Для выполнения данной задачи необходима программная архитектура и структуры данных, обеспечивающие внутреннее представление информации о тексте программ и их семантических моделях в процессе редактирования исходных текстов. Так как существующие способы представления семантических моделей и интеграции семантических анализаторов в средах разработки не позволяют использовать их для предметно-ориентированного анализа, необходима разработка и применение для этой цели альтернативной программной архитектуры и структур данных, позволяющих интегрировать реализацию разработанного метода предметно-ориентированного анализа в различные среды разработки или текстовые редакторы.

Заключительный этап работы – апробация разработанного метода в процессе разработки реальных программных продуктов. Создание программного средства на базе предлагаемого подхода необходимо, чтобы

показать его работоспособность и применимость. Такой анализатор также позволит оценить эффективность разработанного метода в сравнении с существующими специализированными средствами анализа исходного кода программ.

ГЛАВА 2 Метод предметно-ориентированного анализа исходных текстов программ

2.1 Основные термины, понятия и определения

Ключевыми понятиями в работе являются семантическая модель программы и предметно-ориентированный анализ.

Модель – это форма отображения определённого фрагмента исходной системы, содержащая её существенные свойства и являющаяся объектом исследования или анализа.

Предметная область – это система взаимосвязанных терминов или понятий. Например, система понятий языка программирования, система понятий прикладной области или специальная система понятий, вводимая для решения определённой задачи.

Семантическая модель программы – это модель, описывающая элементы программы, их свойства, характеристики и связи в терминах языка программирования или другой предметной области. На уровне семантических моделей выполняется смысловое сопряжение между частями текста программной системы, представленными в терминах разных предметных областей.

В контексте данной работы исходной системой является текст программы, а предметом анализа – элементы семантической модели, являющиеся смысловыми единицами, формирующими модель программы. Анализируемая информация представлена на уровне исходного текста, на уровне дерева синтаксического разбора (синтаксической модели программы) и на уровне семантической модели программы.

Для представления информации об элементах модели и взаимосвязях между ними могут использоваться семантические сети. Формальным представлением семантической сети является граф, в котором каждая вершина соответствует элементу множества сущностей или понятий, а дуги

соответствуют отношениями между этими сущностями. Формальные языки, языки программирования и моделирования данных, разрабатываются по строго определенным правилам с опорой на строго организованную систему понятий и определений. Это позволяет строить семантические модели исходных текстов программ в терминах, определённых семантическими спецификациями, разрабатываемыми для каждого из используемых в программном проекте языков программирования. Такой набор терминов будет являться метамоделью для экземпляра семантической сети.

Предметно-ориентированный анализ – анализ свойств семантической модели, специфичных для способа её применения. Предметно-ориентированный анализ текста программы – анализ свойств текста программы, специфичных для способа применения синтаксических конструкций языка, на котором написана программа. Таким языком может быть язык общего назначения, предметно-ориентированный язык (Domain-Specific Language, DSL) или язык запросов, например язык структурированных запросов (Structured Query Language, SQL), и другие.

В процессе выполнения предметно-ориентированного анализа модели подвергаются преобразованиям по заранее определенным правилам – трансляциям. Для формального описания трансляций и самих моделей программ используются спецификации трансляции, синтаксические и семантические спецификации. Спецификации синтаксических моделей, в терминах которых необходимо прочитать исходные тексты на первом шаге анализа, представляются в форме грамматик, описывающих структуру текста. Правила этих грамматик трактуются также как схемы узлов абстрактного синтаксического графа, материализуемого на базе той же семантической сети, что и семантические модели. Спецификации семантических моделей описывают схему элементов предметных областей, в терминах которых необходимо выполнить анализ. Спецификации семантических трансляций задают отображения между семантическими моделями.

Анализируемая информация может быть представлена как в виде объектных моделей, так и в реляционной форме, где каждому типу объекта реляционной модели будет соответствовать таблица со схемой, описывающей поля, соответствующие свойствам объекта. То же действительно и для представления данных в виде графа, снабжённых аналогичной схемой с классификацией узлов графа, соответствующей набору типов элементов объектной модели. С этой точки зрения семантическая трансляция данных из одной предметной области в другую будет соответствовать набору реляционных запросов (отображений) из одной схемы данных в другую. Использование при этом гибридного или графового представления данных для материализации семантических моделей позволяет использовать графовые примитивы для описания нереляционных отношений, таких как пути в абстрактном синтаксическом графе. При этом для описания трансляций графовое представление может быть использовано совместно с объектно-реляционным.

2.2 Формализация метода анализа текстов программ

Спецификация семантической модели необходима для описания семантики текста программы с точки зрения некоторой предметной области. Она фактически является метаданными для семантической модели, формируемой из исходного кода. В общем случае формально специфициацию семантической модели программы можно представить записью вида:

$$S = (T_s, P_s, V_s, C_s, D_s), \quad (1)$$

где:

$t_s \in T_s$ – идентификаторы типов;

$p_s \in P_s$ – идентификаторы свойств;

V_s – схемы значений, записываемые в виде:

$$V_s = \{v_s = (t, q): \quad (2)$$

$$t \in T_s \cup T_0 ,$$

$$q = \begin{cases} \text{false для одиночных значений} \\ \text{true для наборов значений} \end{cases};$$

при каждой паре $v_s = (t, q)$, задающей способ представления значения атрибута в семантической модели (t – тип значения, q – его кратность);

C_s – принадлежность свойств типам, записываемая в виде отображений идентификаторов типов t_s на набор свойств P_{t_s} :

$$C_s = \{c_s = t_s \rightarrow P_{t_s} : P_{t_s} \subseteq P_s\}; \quad (3)$$

D_s – схемы типов, описываемые в виде отображений пар вида тип-свойство (t_s, p_s) на схему значения v_s :

$$D_s = \{d_s = (t_s, p_s) \rightarrow v_s\}; \quad (4)$$

T_0 – встроенные литеральные типы значений.

Реляционная интерпретация спецификации семантической модели как схемы базы данных:

- $t_s \in T_s$ – имена таблиц;
- $p_s \in P_s$ – имена колонок;
- $V_s = \{v_s = (t, q)\}$ – типы данных, ассоциированных с сущностью (2):
 - способ ассоциации $\begin{cases} \text{поле таблицы, если } q = \text{false} \\ \text{таблица соединений, если } q = \text{true} \end{cases}$
 - тип значения в поле $\begin{cases} \text{встроенный литеральный, если } t \in T_0 \\ \text{внешний ключ, если } t \in T_s \end{cases}$,
- $C_s = \{c_s = t_s \rightarrow P_{t_s}\}$ – имена колонок таблицы (3);
- $D_s = \{d_s = (t_s, p_s) \rightarrow v_s\}$ – схемы полей (4).

Формальное представление семантической модели программы можно записать в виде:

$$M = (S_m, N_m, A_m, V_m, E_m, F_m), \quad (5)$$

где:

S_m – спецификация семантической модели (1);

$N_m \in N_m$ – идентификаторы элементов;

$a_m \in A_m$ – идентификаторы атрибутов;

V_m – значения атрибутов, описываемые в виде пар "схема"- "набор значений" (схема (2) литеральных значений или идентификаторов элементов):

$$V_m = \{v_m = (v_s, X_{v_m}): \quad (6)$$

$$v_s \in V_s, v_s = (t, q),$$

$$|X_{v_m}| \in \begin{cases} \{0,1\}, & \text{если } q = \text{false}, \\ \{\mathbb{N}^0\}, & \text{если } q = \text{true}, \end{cases}$$

$$X_{v_m} = \left\{ x_{v_m}: \begin{cases} x_{v_m} = \{\text{literal value}\}, & \text{если } t \in T_0 \\ x_{v_m} \in N_m, & \text{если } t \in T_s \end{cases} \right\}$$

};

E_m – элементы, описываемые в виде отображений идентификаторов элементов на пары вида "тип"- "набор отношений", заданных парами свойство-атрибут (элементу сопоставляется тип и набор атрибутов, соответствующих свойствам этого типа):

$$E_m = \{e_m = n_m \rightarrow (t_{n_m}, P_{n_m}): \quad (7)$$

$$t_{n_m} \in T_s,$$

$$P_{n_m} = \{(p, a_m): p \in C_s(t_{n_m})\}$$

};

F_m – поля элементов, описываемые в виде отображений пар узел-свойство на пары атрибут-значение (свойству элемента сопоставляется атрибут и ассоциированное с ним значение):

$$F_m = \{f_m = (n_m, p_s) \rightarrow (a_{n_m}, v_{n_m}): \quad (8)$$

$$E_m(n_m) = (t_{n_m}, \{(p_s, a_{n_m})\}),$$

$$v_{n_m} = (v_{s_{n_m}}, x), v_{s_{n_m}} = D_s(t_{n_m}, p_s),$$

$$x \in X, (v_{s_{n_m}}, X) \in V_m$$

}

Реляционная интерпретация семантической модели как экземпляра базы данных:

- S_m – схема семантической модели (1);
- $n_m \in N_m$ – глобально уникальные идентификаторы записей (строк таблиц);
- $a_m \in A_m$ – глобально уникальные идентификаторы полей записей;
- $V_m = \{v_m = (v_s, X_{v_m})\}$ – типизированные значения полей (6);
- $E_m = \{e_m = n_m \rightarrow (t_s, \{(p_s, a_m)\})\}$ – поля записи в соответствии со схемой (7);
- $F_m = \{f_m = (n_m, p_s) \rightarrow (a_{n_m}, (v_s, x))\}$ – типизированные значения полей записи (8).

Интерпретация семантической модели в виде графа будет записана следующим образом:

Пусть $G = (N_g, L_g)$ – граф с вершинами $n_g \in N_g$ и ребрами $l_g = (n_{1g}, n_{2g}) \in L_g$, где:

$$N_g = \{n_g = n_m \forall n_m \in N_m\}; \quad (9)$$

$$L_g = \{l_g = (n_m, x) \forall f_m \in \{f_m = (n_m, _) \rightarrow (a, (_, x)) \in F_m : x \in N_m\}\},$$

где вершины соответствуют элементам семантической модели, а ребра – отношениям между элементами, заданным атрибутами.

Интерпретация семантической модели как семантической сети:

- S_m – система понятий, в терминах которой организована семантическая сеть, экземпляры которых представлены узлами семантической сети (1);
- T_s – понятия, виды сущностей;
- P_s – виды отношений между сущностями или значениями их атрибутов, если они лiteralьны;
- $n_m \in N_m$ – сущности (5);
- $a_m \in A_m$ – отношения между сущностями или их атрибуты;

- $V_m = \{(t, q), X_{v_m}\}$ – информационное содержимое атрибута (6)
 $\begin{cases} \text{литеральное значение атрибута, если } t \in T_0, \\ \text{целевая сущность отношения, если } t \in T_s, \end{cases}$
- $E_m = \{e_m = n_m \rightarrow (t_s, \{(p_s, a_m)\})\}$ – все атрибуты и отношения для сущности (7);
- $F_m = \{f_m = (n_m, p_s) \rightarrow (a_{n_m}, (v_s, x))\}$ – значения атрибута или цели отношения сущности (8).

На рисунке 5 изображено представление семантической модели M (5), метамоделью которой является спецификация S (1), где:

$$S = (\quad \quad \quad (10)$$

$$T_s = \{t1, t2, t3\},$$

$$P_s = \{p1, p2\},$$

$$V_s = \{v1 = (t2, false), v2 = (t3, true)\},$$

$$C_s = \{t1 \rightarrow \{p1\}, t2 \rightarrow \{p2\}\},$$

$$D_s = \{(t1, p1) \rightarrow v1, (t2, p2) \rightarrow v2\}$$

)

$$M = (\quad \quad \quad (11)$$

$$S,$$

$$N_m = \{n1, n2, n3, n4\},$$

$$A_m = \{a1, a2, a3\},$$

$$V_m = \{(v1, \{n2\}), (v2, \{n3, n4\})\},$$

$$E_m = \{$$

$$n1 \rightarrow (t1, \{(p1, a1)\}),$$

$$n2 \rightarrow (t2, \{(p2, a2), (p2, a3)\}),$$

$$n3 \rightarrow (t3, \emptyset), n4 \rightarrow (t3, \emptyset)$$

},

$$F_m = \{$$

$$(n1, p1) \rightarrow (a1, n2),$$

$$(n_2, p_2) \rightarrow (a_2, n_3),$$

$$(n_2, p_2) \rightarrow (a_3, n_4)$$

{}

}

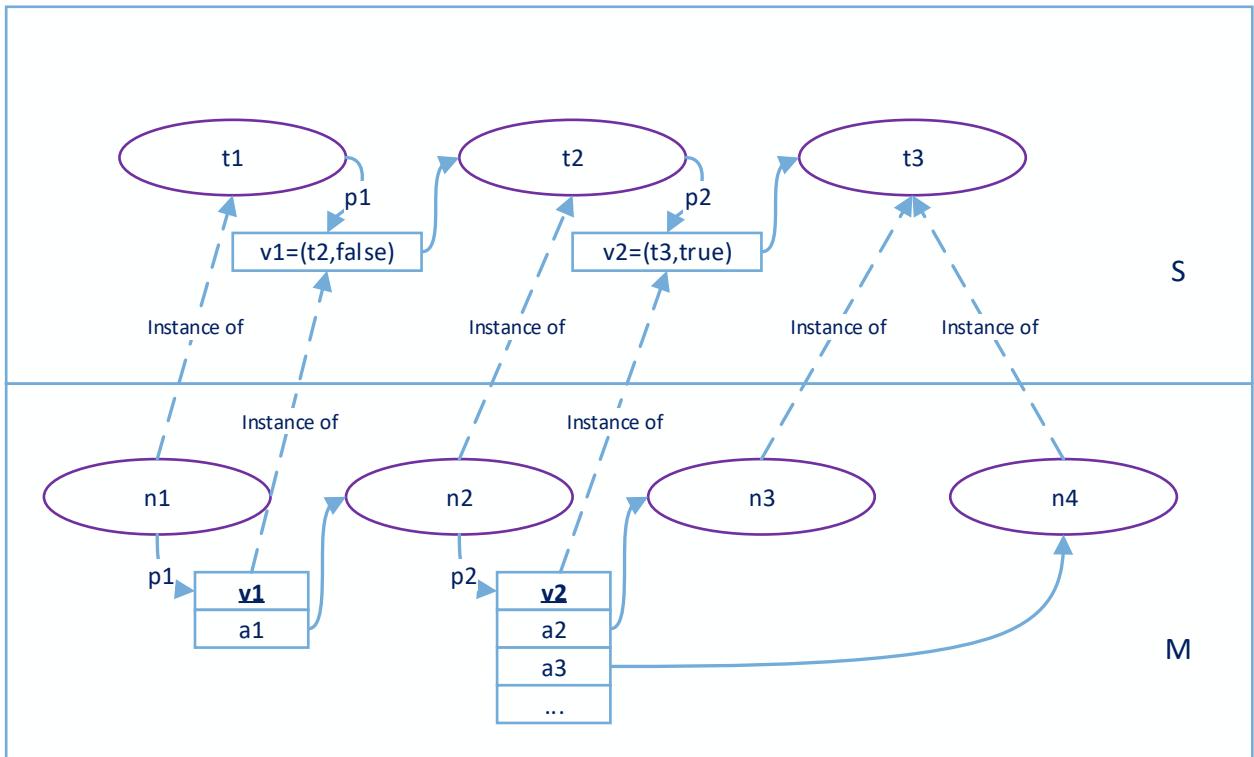


Рисунок 5 – иллюстрация формального представления семантической модели M (11) со спецификацией S (10)

Сущности семантической сети M имеют вид t и связаны отношениями p друг с другом. При этом каждый экземпляр отношения задан атрибутом a , несущим значение в виде связи с целевой сущностью данного отношения. Количество атрибутов для одного свойства узла соответствует кратности из схемы значения свойства: один – если $false$, несколько – если $true$. Сущность n_1 имеет вид t_1 и свойство p_1 , в качестве значения которого выступает атрибут a_1 , представляющий отношение с сущностью n_2 . Сущность n_2 вида t_2 , в свою очередь, имеет свойство p_2 , значениями которого являются атрибуты a_2 и a_3 , представляющие отношения с сущностями n_3 и n_4 .

Спецификация синтаксической модели – это частный случай спецификации семантической модели (1), расширенной контекстно-свободной грамматикой [62]:

$$S = (T_s, P_s, V_s, C_s, D_s, G_s), \quad (12)$$

где компоненты грамматики $G_s = (\Sigma_{G_s}, N_{G_s}, P_{G_s}, S_{G_s})$ следующие:

- Σ_{G_s} – набор терминальных символов;
- $n_{G_s} \in N_{G_s}$ – набор нетерминальных символов;
- P_{G_s} – набор правил вида $n_{G_s} \rightarrow w$, где n_{G_s} – нетерминал, w – любая последовательность терминалов и нетерминалов;
- S_{G_s} – стартовый нетерминал.

Тогда синтаксическая модель – частный случай семантической модели со спецификацией синтаксической модели в качестве S_m (5). Дополнительно к определению спецификации семантической модели, на спецификацию синтаксической модели накладывается ряд ограничений:

$$\begin{aligned} T_s &= N_{G_s} & (13) \\ P_s &= N_{G_s} \cup \Sigma_{G_s} \\ C_s &= \{c_s = t_s \rightarrow P_{t_s}: P_{t_s} \subseteq P_s \wedge t_s \rightarrow w \in P_{G_s} \wedge P_{t_s} \subseteq w\} \\ D_s &= \{d_s = (t_s, p_s) \rightarrow v_s: \\ &\quad t_s \rightarrow w \in P_{G_s}, \\ &\quad p_s \in w, \\ &\quad v_s = (t, q) \in V_s \\ &\quad \begin{cases} q = \text{false}, \text{ если } |\sigma_{p_s}(w)| = 1 \\ q = \text{true}, \text{ если } |\sigma_{p_s}(w)| > 1 \end{cases} \\ &\} \end{aligned}$$

Типы семантической модели T_s ставятся в соответствие нетерминалам грамматики N_{G_s} , свойства типов P_s ставятся в соответствие возможным дочерним узлам в дереве разбора по грамматике, которые могут быть представлены нетерминалами N_{G_s} или терминалами Σ_{G_s} , кратность q значений

этих свойств соответствует кратности дочерних узлов данного вида $|\sigma_{p_s}(w)|$, где $\sigma_{p_s}(w)$ – количество вхождений элемента p_s в цепочку w правила $t_s \rightarrow w$ грамматики. Таким образом обеспечивается отображаемость дерева разбора для грамматики G_s на семантическую модель в соответствии с графовой интерпретацией как абстрактного семантического графа.

Формальное представление спецификации семантической трансляции:

$$R_{1 \rightarrow 2} = (S_1, S_2, Q_{1 \rightarrow 2}, P_{1 \rightarrow 2}, I_{1 \rightarrow 2}), \quad (14)$$

где S_1 – спецификация исходной семантической модели (1), S_2 – спецификация целевой семантической модели (1), $Q_{1 \rightarrow 2}$ – набор контекстных отображений, $P_{1 \rightarrow 2}$ – набор свободных отображений, I – набор начальных отображений для исходной модели m_{s1} и целевой m_{s2} :

$$\begin{aligned} Q_{1 \rightarrow 2} &= \{q_{t_{s1} \rightarrow t_{s2}} = (n_{s1}, m_{s1}, q') \rightarrow m'_{s2}\} \\ P_{1 \rightarrow 2} &= \{p_{t_{s1} \rightarrow t_{s2}} = (n_{s1}, m_{s1}) \rightarrow m'_{s2}\} \\ I_{1 \rightarrow 2} &= \{\iota_{t_{s1} \rightarrow t_{s2}} = m_{s1} \rightarrow m'_{s2}\}, \end{aligned} \quad (15)$$

где каждая из частных трансляций q , p , и ι , описанных функциями $f_{t_{s1} \rightarrow t_{s2}}$, задает отображение из элемента типа t_{s1} исходной модели m_{s1} (5) в элемент типа t_{s2} целевой модели m_{s2} (5) через частичную модель m'_{s2} (5):

$$\begin{aligned} S_1 &= (T_1, P_1, V_1, C_1, D_1), m_{s1} = (S_1, N_1, A_1, V_1, E_1, F_1), \\ S_2 &= (T_2, P_2, V_2, C_2, D_2), m_{s2} = (S_2, N_2, A_2, V_2, E_2, F_2), \\ m'_{s2} &= (S_2, N'_2, A'_2, V'_2, E'_2, F'_2), \\ t_{s1} &\in T_1, t_{s2} \in T_2 \\ N'_2 &= \{n' \in N_2\}, \\ A'_2 &= \{a' \in A_2 : \exists(n' \rightarrow (t_{s1}, \{(., a')\})) \in E_2, n' \in N'_2\}, \\ V'_2 &= \{v' \in V_2 : \exists((n', .) \rightarrow (a', v')) \in F_2, n' \in N'_2, a \in A'_2\}, \\ E'_2 &= \{(n' \rightarrow .) \in E_2 : n' \in N'_2\}, \\ F'_2 &= \{((n', .) \rightarrow .) \in F_2 : n' \in N'_2\}. \end{aligned} \quad (16)$$

Начальные отображения $\iota_{t_{s1} \rightarrow t_{s2}}$ задают корневые трансляции, с которых начинается формирование зависимостей между элементами моделей. Они

могут включать обращения к свободным отображениям $p_{t_{s1} \rightarrow t_{s2}}$ и контекстным отображениям $q_{t_{s1} \rightarrow t_{s2}}$, задающим прямые зависимости между одним исходным элементом и одним целевым без дополнительных аргументов с учетом дополнительных аргументов q' соответственно. Каждое отображение связывает между собой структурные элементы моделей m_{s1} и m_{s2} посредством реляционных отображений. Начальные отображения не требуют дополнительных аргументов, кроме исходной модели, и могут выступать в качестве источников информации для инициации алгоритма, выполняющего семантическую трансляцию. Свободные отображения, применяющиеся к конкретным элементам исходной модели, и контекстные отображения, дополнительно требующие информации из контекста, могут использоваться для выполнения вторичных шагов алгоритма семантической трансляции, инициируемых косвенно при выполнении предшествующих шагов. В качестве узла аргумента и контекста при этом будут использоваться данные, полученные на предыдущих шагах, например, при применении первичных отображений.

Помимо непосредственной императивной интерпретации, функции отображений могут рассматриваться как декларативные источники информации о зависимостях между элементами разных моделей.

Реляционное отображение – композиция операций реляционной алгебры над множествами и операций над отдельными элементами или значениями, описывающая преобразование между элементами данных. В рамках реляционного представления в качестве элементов данных выступают записи таблиц. Каждая таблица рассматривается как множество записей, каждая запись – как кортеж полей. В соответствии с рассмотренной ранее реляционной интерпретацией семантической модели к описываемым ею данным могут быть применены реляционные операции, результатом чего будет формирование новых семантических моделей.

Функции отображений ι , p и q задаются как выражения реляционных отображений над элементами исходной семантической модели в соответствии её реляционной интерпретацией. Результирующие элементы при этом должны соответствовать спецификации целевой семантической модели. К операциям, композицией которых являются формулы отображений, относятся операции реляционных отображений, операции над литеральными элементами данных и обращения к смежным отображениям. Определим необходимые для выполнения анализа операции.

- Проекция, ограничивающая поля записей, составляющих набор:

$$\Pi_{a_1, \dots, a_n}(A), \quad (17)$$

где a_1, \dots, a_n – идентификаторы полей элементов множества A , подлежащих отображению в результирующее множество.

- Переименование, изменяющее идентификатор поля записи:

$$\rho_{a/b}(A), \quad (18)$$

где a – исходный идентификатор поля в составе элементов множества A , b – идентификатор того же поля в результирующем множестве.

- Выборка, отсеивающая записи в зависимости от удовлетворённости некоторого предиката, выраженного пропозициональной формулой:

$$\sigma_\varphi(A) = \begin{cases} \{x: x \in A \wedge \varphi(x)\}, & \text{для } p \\ \{x: x \in A \wedge \varphi(x, q')\}, & \text{для } q \end{cases}, \quad (19)$$

где φ – предикат над элементами множества A , состоящий из пропозициональных переменных (соответствующих идентификаторам полей или элементам контекста q') и n -арных функциональных символов $f(x_1, \dots, x_n)$, которыми выражаются как бинарные логические операторы \wedge , \vee , \neg , так любые другие над доступными аргументами, необходимые для определения истинности предиката для некоторого элемента данных.

- Пересечение, объединение и разность множеств элементов данных:

$$A \cap B = \{x = (a_1, \dots, a_n): x \in A \wedge x \in B\} \quad (20)$$

$$A \cup B = \{x = (a_1, \dots, a_n): x \in A \vee x \in B\}$$

$$A \setminus B = \{x = (a_1, \dots, a_n) : x \in A \wedge x \notin B\},$$

где A и B – множества элементов идентичной структуры (обладающие одинаковым набором полей).

- Произведение множеств записей

$$\begin{aligned} A \times B &= \{x = (a_1, \dots, a_n, a_{n+1}, \dots, a_m) : \\ &(a_1, \dots, a_n) \in A \wedge (a_{n+1}, \dots, a_m) \in B\}, \end{aligned} \quad (21)$$

где A и B – множества элементов с непересекающимся набором полей, а результирующее множество – с их объединением для каждого элементов из A и B .

- Естественное соединение множеств записей

$$A \bowtie_{\varphi} B = \begin{cases} \{x = a \cup b : a \in A \wedge b \in B \wedge \varphi(x)\}, \text{для } p \\ \{x = a \cup b : a \in A \wedge b \in B \wedge \varphi(x, q')\}, \text{для } q' \end{cases}, \quad (22)$$

где A и B – множества элементов с непересекающимся набором полей, а результирующее множество – с их объединением для каждого элементов из A и B , для которых выполняется предикат φ в форме, зависящей от наличия дополнительного локального контекста q' в данном отображении p или q (и так далее в двух вариантах для всех предикатов и функций над элементом модели).

- Внешнее соединение множеств записей

$$\begin{aligned} A \bowtie_{\varphi, \omega} B &= \\ &= \left\{ x = (a_1, \dots, a_n, a_{n+1}, \dots, a_m) : (a_1, \dots, a_n) \in A \wedge (a_{n+1}, \dots, a_m) \in B \right. \\ &\quad \left. \wedge x \in (A \bowtie_{\varphi} B) \cup ((A - \Pi_{\gamma_1, \dots, \gamma_n}(A \bowtie_{\varphi} B)) \times \{\omega\}) \right\}, \end{aligned} \quad (23)$$

где A и B – множества элементов с непересекающимся набором полей, a_i – поля записей, γ_i – их идентификаторы, φ – предикат, ω – синглтон с полями, аналогичными полям элементов множества. Таким образом, элементы результирующего множества имеют объединение полей каждого элементов из A и B , для которых предикат выполняется, и объединение полей элементов из A с полями ω для всех элементов A , для которых предикат не выполняется.

- Агрегация с произвольной функцией накопления в двух вариантах,

$$\begin{cases} G(A, \varphi) = g(2, a_1) \\ G(A, a_0, \varphi) = g(0, a_0) \end{cases} \text{ при } \begin{cases} g(n, x) = g(n + 1, \varphi(a_n, x)) \\ g(|A| + 1, x) = x \end{cases}, \quad (24)$$

где $A = \{a_n : 1 \leq n \leq |A|\}$, применимая так же для описания минимумов, максимумов, сумм, и так далее.

- Прямое отражение набора записей:

$$\Pi_\varphi(A) = \{x = \varphi(a) \forall a \in A\}, \quad (25)$$

где x – элемент результирующего набора, полученный с помощью функции φ из элемента a исходного набора A .

- Декомпозиция (выборка значений отдельных полей):

$$x_\gamma \in a, a \in A, \quad (26)$$

где x_γ – результирующее значение поля с идентификатором γ элемента a из набора A .

- Применение смежных функций отображений (15):

$$b = \begin{cases} p(a), p \in P_{1 \rightarrow 2} \\ q(a, q'), q \in Q_{1 \rightarrow 2} \end{cases}, \quad (27)$$

где a – элемент исходной модели по спецификации S_1 , b – элемент целевой модели по спецификации S_2 в соответствии с определением семантической трансляции, q' - любые дополнительные аргументы из локального контекста, в зависимости от текущего отображения p или q .

- Операции над литеральными значениями: любые чистые $\varphi(x)$ в отображении p или $\varphi(x, q')$ в отображении q , задающие способ формирования значения для некоторого элемента данных, внутренняя структура которых не зависит от других элементов моделей или спецификаций.

Также функции p и q могут содержать арифметические операции, операции выбора и строковые операции, позволяющие рекомбинировать литеральные значения в составе элементов данных. В целом структура этих

функций ограничена соображениями совместимости типов аргументов тех или иных операций и их результатов.

В формализованном виде разработанный метод можно записать следующим образом.

1. Пусть множество Ψ – тексты τ программной системы, написанные на языках с синтаксическими спецификациями S_τ :

$$(\tau_n, S_{\tau_n}) \in \Psi, 1 \leq n \leq |\Psi|, \quad (28)$$

где S_{τ_n} – синтаксическая спецификация текста программы τ_n , n – номер текста τ программной системы. Тогда результатом синтаксического анализа этих текстов будет множество синтаксических моделей Υ :

$$\Upsilon = \{v_{S_\tau} \forall (\tau, S_\tau) \in \Psi\}, |\Psi| = |\Upsilon|, \quad (29)$$

где множество Ψ – тексты τ программной системы, v_{S_τ} – синтаксическая модель по спецификации S_τ для текста программы τ .

2. Пусть Θ – предметные области, в рамках которых выполняется анализ текстов программной системы, заданные набором спецификаций семантических трансляций:

$$\Theta = \{R_n = (S_{1n}, S_{2n}, \dots)\}, 1 \leq n \leq |\Theta|, \quad (30)$$

где R_n – спецификация семантической трансляции, S_{1n} – спецификация исходной семантической модели, S_{2n} – спецификация целевой семантической модели.

3. Предметно-ориентированный анализ на основе семантических моделей для текста τ даёт набор семантических моделей Ω_τ , являющийся набором всех семантических моделей, полученных в результате трансляций $f(m, r)$, начиная с синтаксической модели v_{S_τ} прямо или косвенно для данной семантической модели $m = (S, \dots, \dots, \dots)$ при наличии подходящей спецификации семантической трансляции $r = (S, \dots, \dots, \dots) \in \Theta$ для спецификации семантической модели S :

$$\Omega_\tau = \bigcup_{\substack{m_0=v_{S_\tau} \\ m_n=(S,_,_,_,_) \\ \exists r_{n \rightarrow n+1}=(S,_,_,_,_) \in \theta}} \{m_n\}, m_{n+1} = f(m_n, r_{n \rightarrow n+1}) \quad (31)$$

где m_n – семантическая модель (5), спецификация семантической трансляции $r_{n \rightarrow n+1}$ (14), $f(m_n, r_{n \rightarrow n+1})$ – функция, осуществляющая трансляцию:

$$\begin{aligned} f(m_{S_a}, r_{a \rightarrow b}) &= \varsigma(S_b, \{m' = \iota(m_{S_a}) \forall \iota \in I, \\ &\quad m_{S_a} = (S_a, _, _, _, _) \\ &\quad r_{a \rightarrow b} = (S_a, S_b, Q, P, I)\}), \end{aligned} \quad (32)$$

где $r_{a \rightarrow b}$ – спецификация семантической трансляции (14) из семантической модели m_{S_a} по спецификации S_a в семантическую модель по спецификации S_b , m' – частичная целевая модель, получающаяся в результате применения трансляции $\iota \in I$ из спецификации $r_{a \rightarrow b}$ к модели m_{S_a} , ς – функция, объединяющая набор семантических моделей m'_Σ с общей спецификацией S в одну общую модель:

$$\begin{aligned} \varsigma &= (S, m'_\Sigma) \rightarrow (S, \\ &\quad N_b = \varsigma'(m'_\Sigma, (S, N_{m'_\Sigma}, _, _, _) \rightarrow N_{m'_\Sigma}), \\ &\quad A_b = \varsigma'(m'_\Sigma, (S, _, A_{m'_\Sigma}, _, _) \rightarrow A_{m'_\Sigma}), \\ &\quad V_b = \varsigma'(m'_\Sigma, (S, _, _, V_{m'_\Sigma}, _) \rightarrow V_{m'_\Sigma}), \\ &\quad E_b = \varsigma'(m'_\Sigma, (S, _, _, _, E_{m'_\Sigma}) \rightarrow E_{m'_\Sigma}), \\ &\quad F_b = \varsigma'(m'_\Sigma, (S, _, _, _, _, F_{m'_\Sigma}) \rightarrow F_{m'_\Sigma}) \\ &\quad), \end{aligned} \quad (33)$$

где $(S, N_b, A_b, V_b, E_b, F_b)$ – семантическая модель (5), ς' – функция, объединяющая результаты применения аргумента z к каждому из членов множества, заданного аргументом Y :

$$\varsigma' = (Y, z) \rightarrow \bigcup z(y) \forall y \in Y \quad (34)$$

4. M_Γ – все семантические модели (29) для всех текстов Υ (28) (первичные синтаксические и полученные из них трансляциями в совокупности):

$$M_\Gamma = \bigcup \Omega_\tau \forall v_{S_\tau} \in \Upsilon, \quad (35)$$

а Γ – итоговый набор объединённых семантических моделей:

$$\Gamma = \{\gamma_s = \varsigma(s, X) : \forall x = (s, _, _, _, _) \in X \subseteq M_\Gamma\}, \quad (36)$$

состоящее из объединённых функцией ς (33) семантических моделей x (5) по каждой из доступных спецификаций s (1).

Спецификации семантических моделей могут быть дополнены предикатами, заданными в любой из интерпретаций семантических моделей. Построение пространства Γ не требует выполнения всех ограничений и предикатов семантических и синтаксических моделей, так как они описывают формальные зависимости между элементами данных и их членами. По удовлетворённости этих ограничений и предикатов разработчик ПО может судить о соответствии текстов программной системы данным спецификациям.

2.3 Область применения и требования к реализации

После определения формального базиса разрабатываемого метода можно приступить к определению требований к его реализации с учётом особенностей программных проектов. Требования к структуре текста программы, специфические для программного проекта в конкретной области по определению формулируются в терминах этой предметной области.

Рассмотрим пример. Пусть дана программа на языке C#, использующая отображение данных из реляционной базы данных на классы языка C#. При использовании библиотек Linq2Database или EntityFramework возможность выполнения таких отображений зависит от правильной разметки классов специальными атрибутами, управляющими логикой объектно-реляционного отображения. При этом семантика языка C# не включает в свою предметную область правила объектно-реляционных отображений, специфичных для той или иной реализации таких отображений. То есть необходимый семантический анализ, который должен быть выполнен для статической

проверки корректности атрибутов объектно-реляционного отображения, не выполним средствами анализа кода общего назначения для программ на языке C#. Такой анализ является предметно-ориентированным и выполним либо специализированными инструментами, либо библиотекой, выполняющей объектно-реляционное отображение во время исполнения программы. В данном контексте подразумевается, что предметной областью является синтаксис и семантика языка программирования, так как они являются самодостаточной системой понятий.

Специфическую разметку классов для объектно-реляционного отображения, используемую при описании схемы базы данных, можно считать предметно-ориентированным языком. Такой язык зависит не только от библиотек, осуществляющих объектно-реляционное отображение, но и от способа, которым конкретный программный проект использует эти библиотеки. Например, средства разработки Linq2Database не позволяют автоматически создавать отдельные таблицы в существующей базе данных, а также не имеют поддержки миграций. Такая функциональность может быть реализована в различных программных проектах по-разному. При этом она требует дополнительной информации, описывающей объектно-реляционное отображение с помощью классов и свойств, отображаемых на понятия таблиц и записей посредством атрибутов языка C#.

В момент начала работы с кодом программы, должна быть загружена реализация рассматриваемого метода и сформированы необходимые семантические модели в соответствии с принципами, описанными в п. 2.1 и 2.2. В данном контексте в качестве предметных областей для анализа выступают синтаксис и семантика языка программирования, на котором написан текст программы и предметная область схемы базы данных, описанной с помощью исходного языка. Рисунок 6 иллюстрирует последовательность формирования моделей программы.

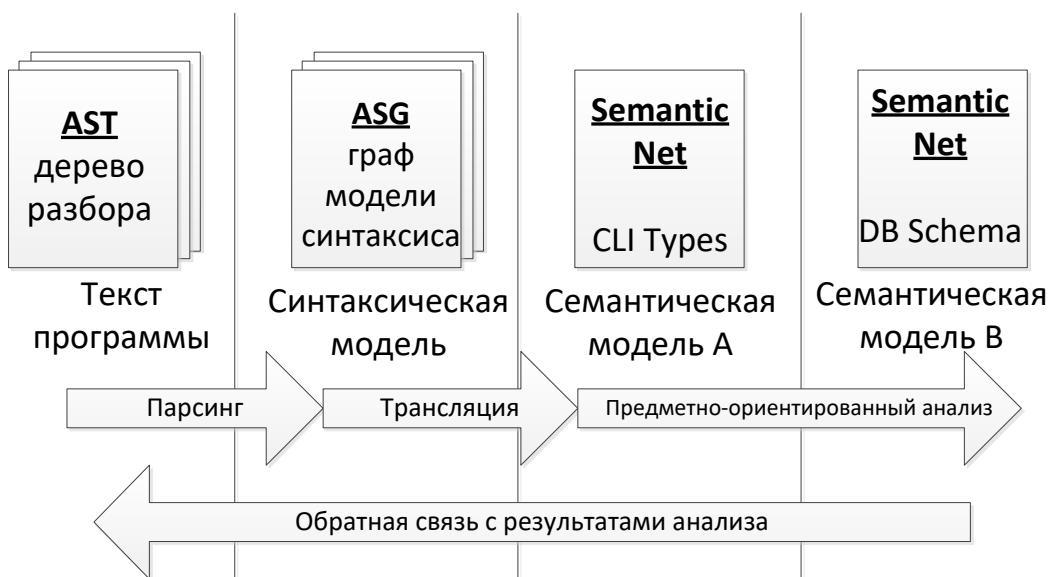


Рисунок 6 – Пример хода анализа для модели БД в программе на С#

На первом этапе выполняется синтаксический анализ (парсинг) текста на языке С#, с учетом информации внутренних структур данных текстового редактора. Например, может использоваться уже существующее дерево разбора. Формируется синтаксическая модель. На последующих этапах анализа выполняется семантическая трансляция из абстрактного синтаксического графа (ASG) синтаксической модели в семантическую модель языка С#, а затем в семантическую модель схемы базы данных. Собранная в ходе анализа информация, визуализируется.

Сформированные модели сохраняют взаимосвязи с текстовым редактором в составе среды программирования и продолжают использоваться по ходу редактирования текста. Учет зависимостей между элементами данных, описанных отображениями в составе спецификации семантической трансляции, позволяет обновлять только те части моделей, данные которых были затронуты в процессе редактирования ранее проанализированного текста. Так обеспечивается гранулярность и инкрементальность анализа.

На этапе синтаксического анализа для данного метода анализа текстов программ не имеет значения, к какому классу анализаторов принадлежит синтаксический анализатор, используемый для разбора исходных текстов. Принципиальным является отражение дерева разбора на семантическую сеть

так, что её фрагмент будет являться абстрактным синтаксическим графом. Узлы этого графа в дальнейшем трактуются как сущности предметной области синтаксической модели исходного текста.

На последующих этапах работы происходит с информацией, представленной с помощью семантической сети. С точки зрения реализации, она содержит совокупность моделей. Анализируемая информация является семантической моделью – экземпляром предметной области, сформулированной в терминах, заданных спецификацией семантической модели – схемой предметной области. Пример такой семантической сети представлен на рисунке 7.

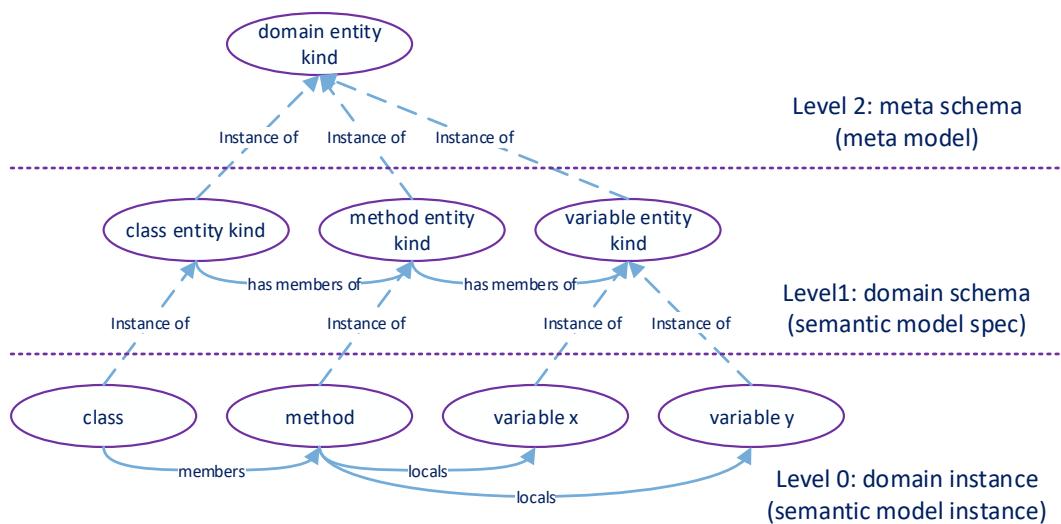


Рисунок 7 – Пример семантической метамодели (level 1) и модели (level 0) программного кода для языка C#

В процессе выполнения семантических трансляций между различными семантическими моделями и при учёте зависимостей между их элементами, используются реляционные отображения. Для семантической сети они являются запросами к графу, представляющему данную сеть. В ходе эксперимента, проводимого на примере представления информации о программной системе с помощью графовой базы данных применительно к статическому анализу веб-приложений, были проанализированы некоторые способы реализации запросов над графиками [81]. Исследование проводилось на

примере представления информации о программной системе с помощью графовой базы данных применительно к статическому анализу веб-приложений.

Интерпретация запросов к графикам при работе с графовыми представлениями может быть сформулирована как задача поиска путей. Путь в графике G – это последовательность ребер графа:

$$p = (v_0, a_1, v_1) \dots (v_{n-1}, a_n, v_n), \quad (37)$$

где каждое ребро представлено тройкой значений: v_0, v_1, v_n, v_{n-1} – вершины графа, a_n – символ алфавита, являющийся условием перехода между вершинами, n – количество ребер графа.

Условия (ограничения) поиска конкретного пути, связанные с интерпретацией реляционного отображения, могут быть представлены несколькими способами: в виде конъюнктивного запроса, запроса кратчайшего пути, регулярного запроса в терминах формальных (регулярных) языков.

Ограничения в терминах регулярных языков используются для поиска шаблонов по графу. Регулярные выражения для алфавита Σ объявляются в форме:

$$E ::= \emptyset \mid \varepsilon \mid \alpha \mid (E \circ E) \mid (E + E) \mid E^*, \quad (38)$$

где $\alpha \in \Sigma$, ε – эпсилон-переход, E – регулярное выражение.

Если E регулярное выражение, тогда $L(E)$ является регулярным языком. Регулярный запрос – это такое выражение вида $x \xrightarrow{r} y$, где x и y – переменные, и r – регулярное выражение над алфавитом Σ . Если r – это регулярное выражение, а G – это график, тогда путь p графа G соответствует регулярному выражению r если $a_1 a_2 \dots a_n \in L(r)$.

Для задания ограничений могут использоваться контекстно-свободные языки. Контекстно-свободная грамматика G может быть объявлена в виде четверки $G = (V, T, P, S)$, где V – конечное множество нетерминальных символов S , T – конечное множество терминальных символов, P – множество

порождающих правил в форме $\alpha \rightarrow \beta, \alpha \in V$ и $\beta \in (V \cup T)^*$, и S – стартовый (начальный) символ грамматики. Результатом запроса в данном случае является множество триплетов (A, m, n) такое, что существует путь от узла m к узлу n , условием перехода для которого является нетерминал A , где $A \in V$. Например, контекстно-свободная грамматика $S \rightarrow aSa$ и $S \rightarrow bSb$ порождает язык палиндромов четного количества символов. На рисунке 8 показан граф с маршрутом, определенный языком $L = \{ww^R, w \in \{a, b\}^*\}$, где w – набор слов, состоящих из цепочек символов a, b .

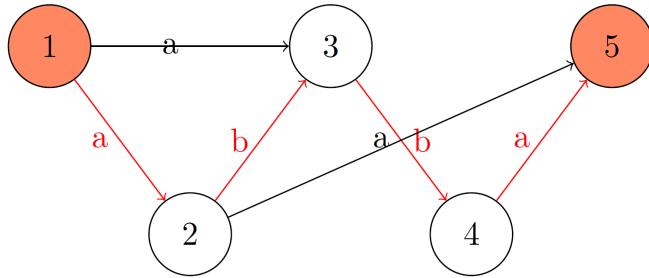


Рисунок 8. Граф с маршрутом, соответствующий языку
 $L = \{ww^R, w \in \{a, b\}^*\}$

Контекстно-свободные запросы более выразительны, чем регулярные запросы, но такой тип запросов требует большого количества памяти.

Другим способом повысить выразительные качества регулярных запросов является применение конъюнктивных регулярных запросов. Конъюнктивный регулярный запрос r_i над графом – это выражения вида

$$\exists \bar{z} \left(\left(x_1 \xrightarrow{a_1} y_1 \right) \wedge \dots \wedge \left(x_n \xrightarrow{a_n} y_n \right) \right), \quad (39)$$

где \bar{z} – кортеж переменных вида $\{x_1, \dots, x_n, y_1, \dots, y_n\}, \{a_1, \dots, a_n\} \in \Sigma, i \in [n]$. Конъюнктивные запросы (и даже запросы первого порядка) над графиками ограничены локальностью их применения, так как могут выражать только свойства конкретного графа, что не решает в полной мере задачу внутреннего представления исходного кода программ для одновременного анализа и редактирования текстов программ, разрабатываемых с использованием нескольких языков программирования.

Сформулируем требования к решению, позволяющему выполнять предметно-ориентированный анализ исходных текстов программ в процессе их редактирования на основе семантических моделей программ и предметных областей, затрагиваемых в рамках отдельных проектов.

1. Решение должно поддерживать анализ текстов программ на различных языках программирования одновременно.
2. Решение должно поддерживать анализ полного объёма текста программы или программной системы в начале своей работы.
3. Решение должно поддерживать гранулярность и итеративность анализа для скорейшего формирования актуального результата анализа.
4. Решение не должно вносить дополнительных накладных расходов на преобразование данных между различными формами представления.
5. Решение не должно требовать от использующего его разработчика изучения дополнительных предметных областей, не связанных с решаемыми им задачами по роду деятельности в программном проекте.

Для достижения поставленной в работе цели с учетом перечисленных выше требований был разработан метод предметно-ориентированного анализа исходных текстов программ. Метод основан на использовании семантических моделей, полученных в результате анализа текстов программ, разрабатываемых с использованием нескольких языков программирования. Основными операциями, выполняемыми в процессе анализа над семантическими моделями программ, являются операции семантической трансляции – отображение одной семантической модели на другую семантическую модель. Семантическая трансляция задаётся отображением элементов данных, представляющих информацию в терминах исходной предметной области одного языка программирования, в элементы данных в терминах целевой предметной области другого языка программирования или предметно-ориентированного языка. На рисунке 6 показано место семантической трансляции в процессе анализа. Элементы данных здесь

представляют информацию, извлеченную из исходного текста. Сами предметные области к этому моменту уже определены, исходя из контекста программного проекта. Необходимо определить способ семантической трансляции.

Языки и системы понятий, используемые для работы с семантическими представлениями в области инженерии знаний, в данном случае не подходят, так как требуют от разработчика изучения соответствующих дополнительных предметных областей, специфического синтаксиса, дополнительных внешних инструментов и так далее. В противовес этому методы и средства реляционного моделирования на сегодняшний день широко распространены, поэтому целесообразно использовать их.

Как уже упоминалось выше, информация представлена объектными моделями в памяти программы, а также в реляционной форме, где каждому типу объекта реляционной модели будет соответствовать таблица со схемой, описывающей поля, соответствующие свойствам объекта. То же действительно и для представлений данных в виде графа, снабжённых аналогичной схемой с классификацией узлов графа, подобной набору типов элементов объектной модели. С этой точки зрения семантическая трансляция данных из одной предметной области в другую будет соответствовать набору реляционных запросов из одной схемы данных в другую. Использование при этом гибридного или графового представления данных для материализации семантических моделей позволяет использовать графовые примитивы для описания нереляционных отношений, таких как пути в абстрактном синтаксическом графе. При этом графовое представление может быть использовано совместно с объектно-реляционными принципами для описания трансляций смежных сущностей. Такой подход позволяет снизить порог вхождения для разработчика, заинтересованного в использовании предметно-ориентированного анализа на базе семантических моделей.

Исходными данными для процедуры анализа являются:

- 1) исходные тексты программ на различных языках программирования – представляются наборами файлов или состояний текстового редактора;
- 2) спецификации синтаксических моделей, в терминах которых необходимо прочитать исходные тексты на первом шаге анализа – представляются в форме грамматик, описывающих структуру текста, правила которых трактуются также как схемы узлов абстрактного синтаксического графа, материализуемого на базе той же семантической сети, что и семантические модели;
- 3) спецификации семантических моделей, описывающие схему элементов предметных областей, в терминах которых необходимо выполнить анализ;
- 4) спецификации семантических трансляций, задающие отображения между семантическими моделями.

Для данного метода анализа текстов программ не имеет значения, к какому классу анализаторов принадлежит синтаксический анализатор, используемый для разбора исходных текстов. Принципиальным является отражение дерева разбора на семантическую сеть так, что её фрагмент будет являться абстрактным синтаксическим графом. Узлы этого графа в дальнейшем трактуются как сущности предметной области синтаксической модели исходного текста.

Ход анализа, в соответствии с предлагаемым методом, заключается в последовательном применении заданных семантических трансляций для перехода от одной предметной области к другой до тех пор, пока все интересующие проверки над семантическими моделями не будут выполнены. В результате выполнения анализа, будет сформирована общая семантическая сеть, включающая набор семантических моделей в терминах предметных областей, связанных отображениями, как показано на рисунке 9.

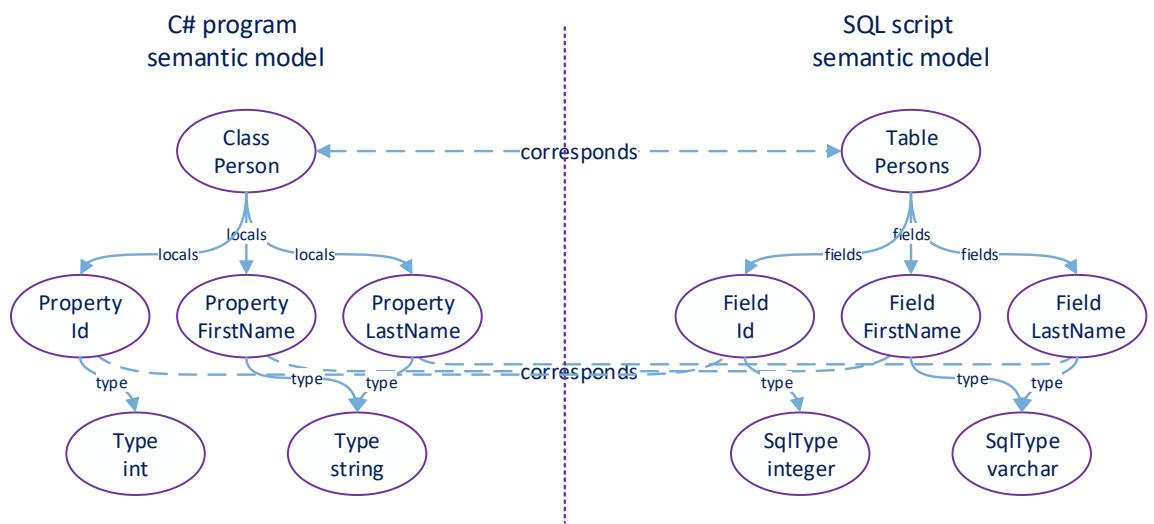


Рисунок 9 – Пример взаимного отображения семантических моделей для языков C# и SQL

Теперь из данной семантической сети можно извлекать информацию о сущностях, их идентификаторах в тексте программы, местах их определения или ссылках на них. И так далее в зависимости от конкретных предметных областей. Определения предметных областей могут быть дополнены различными ограничениями на структуру определений, такими как уникальность значений полей в пределах узлов семантической модели, объединённых каким-либо признаком, например, областью видимости. Представление элементов текстовых данных в составе семантической сети может использовать элементы данных из состава внутренней структуры данных текстового редактора. Это исключает дублирование данных и упрощает выявление частей семантических моделей, затрагиваемых изменениями текста. Таким образом, открывается возможность поддержки гранулярного и итеративного анализа на основе предложенного метода.

Рассмотрим вопрос использования предлагаемого метода и написания спецификаций заинтересованными в предметно-ориентированном анализе разработчиками.

Прежде всего широкое применение предлагаемого метода предполагает создание репозитория спецификаций синтаксических и семантических

моделей, а также семантических трансляций [2]. Такой репозиторий позволит совместно использовать единожды составленные спецификации для типовых этапов предметно-ориентированного анализа, таких как разбор исходных текстов программ и программных систем, написанных на распространённых языках программирования. В этом случае написание новых синтаксических спецификаций будет необходимо только тем разработчикам, которые используют подходы языково-ориентированного программирования с разработкой внешних DSL.

То же относится и к спецификациям семантических трансляций. Например, внутренние семантические модели языков программирования, используемые библиотеками и средствами рефлексии для осуществления ранее упомянутых объектно-реляционных отображений при взаимодействии с базами данных на языке программирования C# посредством библиотек EntityFramework, являются общими для всех сценариев с использованием языка программирования C#.

По мере накопления репозитория спецификаций синтаксических и семантических моделей общего назначения, а также семантических трансляций для этих моделей, конечным разработчикам необходимо будет создавать только небольшие спецификации, описывающие фрагменты предметных областей и применимые в рамках конкретных программных проектов.

Выводы по главе 2

1. Предложен метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей и дано его формальное описание.
2. В результате анализа особенности реализации текстовых редакторов, входящих в состав существующих интегрированных сред разработки, был выявлен ряд ограничений, осложняющих использование единого внутреннего представления текста программы редактором и

анализатором одновременно. Для решения проблемы необходима программная архитектура, допускающая интеграцию текстового редактора с анализатором.

3. Для одновременного использования внутреннего представления текста программы редактором и анализатором необходимо разработать модифицированные структуры данных.

Материалы, изложенные в данной главе, соответствуют первому положению, выносимому на защиту: «Метод предметно-ориентированного анализа исходных текстов программ, разрабатываемых с применением нескольких языков программирования». Основные результаты изложены в работах [3] и [81].

ГЛАВА 3 Язык описания семантических моделей программ

3.1 Описание синтаксических моделей программ

Синтаксический анализ необходим для определения принадлежности синтаксических конструкций заданному языку программирования. Анализатор выполняет синтаксический разбор поступающих на вход цепочек символов и формирует дерево разбора. Узел дерева разбора, соответствующий некоторому правилу, должен содержать дочерние узлы, соответствующие правилам, на которые ссылается цепочка данного правила. Такое представление дерева разбора обусловлено тем, что синтаксические модели используются не только в качестве грамматик для синтаксического анализа [46, 47, 83], но и как спецификации синтаксической структуры. Они используются для построения абстрактного семантического графа с узлами, схема данных которых должна соответствовать структуре цепочек правил. В этом случае такая спецификация синтаксической модели может интерпретироваться и как спецификация семантической модели. Тогда она будет соответствовать тому же самому представлению абстрактного синтаксического графа, но рассматриваемого в составе семантической сети. Такая двоякая интерпретация синтаксических моделей позволяет использовать задаваемые ими предметные области языков программирования как опору для первого шага семантической трансляции, использующей в качестве исходной семантической модели абстрактный синтаксический график. Имена правил соответствуют идентификаторам типов в формальной спецификации (1) и нетерминалам грамматики (12).

При таком способе использования синтаксических моделей и интерпретации грамматик стандартное представление грамматик в форме Бэкуса-Наура (БНФ-грамматики) [30] неприменимо. Представление грамматик должно быть модифицировано для того, чтобы привести класс конкретного варианта грамматики к классу конкретного синтаксического

анализатора. Такие трансформации должны быть скрыты и автоматизированы так, чтобы дерево разбора всегда соответствовало заданной структуре цепочек правил, описывающих фрагменты анализируемых текстов.

Использование данного подхода неспециалистами в области построения синтаксических анализаторов также предполагает более естественные способы описания повторов синтаксических конструкций, чем явным образом описанные альтернативные цепочки одного и того же правила. Данный способ описания повторов обладает недостатком – порождаемые при его использовании деревья разбора часто не соответствуют интуитивно понятной и желаемой структуре абстрактного синтаксического графа. Примером альтернативных способов описания повторов является использование кванторов Клини – символов '?', '*' '+', широко известных по регулярным выражениям. Здесь символ '?' обозначает вхождение некоторой цепочки один или нуль раз в данном месте синтаксической структуры, символ '*' – вхождение нуль или более раз, символ '+' – вхождение один или более раз.

Описание больших синтаксических спецификаций для реальных языков программирования общего назначения также может требовать средств декомпозиции и повторного использования ранее описанных синтаксических моделей внутри друг друга. Стандартные подходы к описанию формальных грамматик на основе БНФ-грамматик не обладают такими возможностями.

Далее будем рассматривать спецификации синтаксических моделей как частный случай семантических моделей. И те, и другие являются частными случаями схем предметных областей, подвергающихся семантическим трансляциям в ходе предметно-ориентированного анализа.

При описании семантических трансляций необходимо различать элементы предметных областей даже в том случае, если они идентичны по назначению. Например, синтаксические структуры арифметических выражений в языке C# и языке C++ очень похожи по набору доступных операторов, но трактуются совершенно различным способом, то есть их

семантика различается. По этой причине всякая схема предметной области задаёт пространство имён, которому принадлежат соответствующие виды существостей.

Предложенный ниже язык описания синтаксических спецификаций был представлен в работе [4] как язык описания грамматик для исследуемого в указанной работе языкового инструментария. Данный языковой инструментарий был использован в работе [63], которая потребовала улучшения синтаксического анализатора и развития на его основе семантического анализатора. Эти работы стали первыми итерациями в разработке метода предметно-ориентированного анализа исходных текстов программ на основе семантических моделей.

Пространства имён синтаксических определений формируются двумя способами – наборами правил и самими правилами. Правило синтаксической модели при этом рассматривается в двух аспектах – как цепочка, ставящаяся в соответствие фрагменту анализируемого текста, и как набор правил, принадлежащих дочерней лексической области, ограниченной данным правилом. Также как для лексических областей в языках программирования, ссылки на непосредственно вложенные правила текущего правила не требуют дополнительных квалификаторов. Достаточно имени самого целевого правила, к которому необходимо обратиться. То же самое действительно для вложенных правил более глубокого уровня по отношению к текущему. Обращение к правилу из скрытой ветви иерархии лексических областей видимости может быть выполнено посредством указания его полного имени. Полное имя состоит из имен всех правил или наборов правил по иерархии, разделённых точкой. Вместо полного имени может быть использовано относительное, если какая-либо часть дерева иерархии областей видимости является общей для ссылающегося и целевого правил.

Правила при этом могут описываться двумя способами – как простое правило или как расширяемое. Простое правило однозначно задаёт

соответствующее выражение цепочки грамматики. Расширяемое правило не задаёт цепочку в явном виде, вместо этого определяя набор дочерних правил, каждое из которых является возможной альтернативой по месту вхождения данного правила, как показано в таблице 3.

Таблица 3 – Примеры правил и наборов правил синтаксической модели

Элемент	Пример
Простое правило с дочерними правилами	<pre>ruleName: x y z; expr: s m n { s: expr '+' expr; m: expr '*' expr; n: "[0-9]+"; };</pre>
Расширяемое правило	<pre>expr: { s: expr '+' expr; m: expr '*' expr; n: "[0-9]+"; };</pre>
Набор правил	<pre>ruleSet { ruleX: "someth"; ruleY: "other"; subset { } ruleZ: "thing"; }</pre>

Расширяемые правила позволяют дополнить список альтернативных выражений, допустимых в том или ином контексте. Пусть расширяемое правило `expr` для целочисленных арифметических выражений из примера определено в наборе правил `intCalc`:

```
intCalc {
    expr: { ...|...|... };
}
```

Тогда мы можем описать синтаксическую модель десятично-дробных арифметических выражений следующим образом:

```
realCalc {
    expr: intCalc.expr; // ссылка на правило expr из набора
intCalc
    intCalc.expr: {      // расширение правила expr из набора
                        // intCalc
        |rn: "[0-9]+\\.\\[0-9]+"
    };
}
```

Другой возможный способ повторного использования синтаксических определений – это импортирование всех членов некоторого пространства имён. Например, для той же задачи:

```
realCalc {
    intCalc; // сделать видимыми все правила из intCalc
    expr: { // определить realCalc.expr
            // так как intCalc.expr тоже доступно,
            // оно расширяется
        |rn: "[0-9]+\\.\\[0-9]+"
    };
}
```

Важно, что при этом члены импортируемого пространства имён не вносятся в импортирующее пространство имен. Это обеспечивает независимость интерпретации соответствующих элементов синтаксической модели, построенной в процессе анализа.

Выражения цепочки правила, сопоставляемого фрагменту текста, могут состоять из различных элементов. Перечислим основные.

1. Цепочка последовательности вхождений элементов: a b c
2. Вхождение одной из набора альтернативных цепочек: a | b | c
3. N вхождений цепочки: a? b* c+
4. Дословный литерал, точное вхождение символов: ' abc '

5. Регулярный литерал, вхождение регулярной последовательности символов: "[0-9]+\. [0-9]+"
6. Скобочная группа последовательности: a (b c) d
7. Ссылка на правило: ruleName

Кроме перечисленных, правила синтаксических моделей могут быть расширены такими элементами, как опережающие проверки и интервальные квантификаторы повторов. Также могут быть введены шаблонные правила, ссылки на которые будут требовать подстановки цепочек в качестве аргументов (удобны для описания типовых списков различного вида в составе синтаксиса анализируемого текста). Синтаксис расширяемых правил может быть дополнен определением групп приоритетов и правилами полностью без тела в форме '`| name`', предназначенными для определения ключевых слов. Полный текст грамматики разработанного языка описания семантических моделей приведён в приложении А.

Данный синтаксис предлагает различные возможности трактовки рекурсии в синтаксической модели. Рассмотрение возможных техник синтаксического анализа, применимых для разбора текстов в соответствии с моделями в данной форме, выходит за рамки этой работы. Достаточно будет сказать, что наиболее перспективными являются такие алгоритмы как RNGLR, GLL, LL(*), но их использование сопряжено с рядом ограничений.

1. Существует необходимость предварительной генерации управляющих структур данных для синтаксического анализатора.
2. Сложно сделать анализатор гранулярным по отношению к последовательным изменениям ранее проанализированного текста.
3. Невозможно построить динамически расширяемый анализатор без повторного полного анализа исходной грамматики.

Более предпочтительным выглядит алгоритм «Марга» [5, 58] или обобщённые модификации алгоритма «Top-Down Operator-Precedence parsing» [34, 74], так как для них такие проблемы отсутствуют.

Таким образом, предлагаемый язык описания синтаксических моделей для предметно-ориентированного анализа текстов программ может использоваться в качестве синтаксического сахара для описания грамматик. В таком сценарии использования определения синтаксических моделей необходимо динамически транслировать в БНФ-подобную форму, подходящую для используемого синтаксического анализатора. Если дерево разбора, генерируемое используемым анализатором, соответствует ранее упомянутым ограничениям трактовки синтаксических моделей, или сводимо к необходимому представлению методом переписывания дерева разбора, то предлагаемый язык описания синтаксических моделей может использоваться как промежуточный слой абстрагирования от реализации синтаксического анализатора.

3.2 Описание семантических моделей программ

Так как описание семантической трансляции содержит указание всех элементов целевой семантической модели для каждого отображения, возможно использование одной и той же спецификации как источника информации не только о семантической трансляции, но и о схеме целевой предметной области. Такой подход был отвергнут по нескольким причинам.

1. Различные варианты предметно-ориентированного анализа могут связывать экземпляры одних и тех же предметных областей различающимися трансляциями. Спецификация самой предметной области при этом должна оставаться самотождественной, а не дублироваться, будучи производной от вариантов трансляций. При наличии трансляции из предметной области не важно, какая из множества доступных трансляций породит экземпляр этой предметной области, следовательно, ассоциировать это экземпляр с породившей его трансляцией некорректно.
2. Выделенная спецификация предметной области может использоваться для проверки описаний семантических трансляций на корректность. Это

относится и к исходным, и к целевой предметной области. Каждая спецификация предметной области связана отношением один ко многим спецификациям семантических трансляций.

3. Выделенная спецификация предметной области может быть дополнена ограничениями или проверками, представляющими интерес с точки зрения предметно-ориентированного анализа исходного текста. Такая информация по определению не является частью семантической трансляции.

Пространство имён спецификации семантической модели формируется аналогично пространству имён синтаксической модели. Рассмотрим синтаксическую структуру предлагаемых спецификаций семантических моделей.

Элементами пространства имён являются определения сущностей, составляющих предметную область. Имя каждого определения соответствует идентификатору типа (1). Всякое определение сущности задаёт тип, определяющий схему данных для экземпляров данной сущности. Всякая сущность может нести набор полей, описывающих либо ассоциацию литеральных значений (подобно семантике значимых типов), либо ассоциацию с другой сущностью или набором сущностей (2) (подобно ссылочной семантике ссылочных типов).

Рассмотрим простой пример.

```
syntax {
    scope: {
        name: string;
        rules: rule[];
        |namespace {
            namespaces: namespace[];
        }
        |rule {
            expr;
        }
    };
}
```

Данный пример демонстрирует описание сущности с именем scope, которая может быть представлена двумя вариантами – сущностью namespace и сущностью rule. Каждая из них будет нести атрибут name типа string и набор ссылок на сущности типа rule. При этом в варианте namespace будет также присутствовать список ссылок на другие namespace, а в варианте rule будет присутствовать ссылка на сущность типа expr.

Полями литературных типов задаются атрибуты сущностей, значения которых ассоциируются непосредственно с узлами семантической модели, а полями сложных типов задаются отношения между сущностями, представленные дугами ссылок между узлами семантической сети. Каждое значение определяется через 'k: v;', где k – имя поля, отношения или сущности (если отсутствует имя типа и отсутствует встроенное определение типа). Каждый тип сущности определяется через 'e: { ... };' для самостоятельных и встроенных, или '|e: { ... }' для алгебраических вариантов частных случаев содержащего типа, включая вариант без расширения набора полей '|e'.

Ниже приведены определения типов сущностей с учетом возможности описания подтипов.

```

e1: { ... };           // тип и/или отношение с именем e1
e2: e3 { ... };       // тип и/или отношение с именем e2,
                      // расширяющее тип e3
f: t;                 // поле или отношение с именем f и типом t
f: t[];               // отношение f отношение к набору сущностей типа t
t;                   // поле или отношение с именем t и типом t
|x                   // вариант содержащего типа с именем x
|x { ... }           // вариант содержащего типа с
                      // именем x и расширением

```

В качестве наглядного примера ниже приведена упрощенная спецификация предметной области типов Common Language Infrastructure

(ECMA-335) без учёта обобщенных типов и некоторых других аспектов метаданных.

```

cli {
    assembly: {
        name: string;
        types: type[];
    };
    member: {
        name: string;
        visibility: {
            |private| public|protected
        };
    };
    type: member {
        kind: {
            |class| delegate|struct
            |enum| interface
        };
        members: member[];
        parent: type;
    };
    field: member {
        type;
    };
    method: member {
        signature: {
            returns: type;
            args: type[];
        };
    };
    property: member {
        type;
    };
    event: member {
        type;
    };
}

```

Таким образом, разработанный язык описания спецификаций семантических моделей позволяет описывать составные типы, соответствующие сущностям предметной области. Спецификации задают

схемы данных, ассоциируемых с теми вершинами графа семантической сети, которые представляют экземпляр данной предметной области (5). Можно также сказать, что экземпляры таких сущностей (вершины графа) являются экземплярами данных составных типов. Сама система типов обладает набором встроенных литературальных типов данных (таких как `string`, `int`, и т.п.), а также поддержкой подтипов, вариантов типов и типов-множеств.

3.3 Описание семантических трансляций

Как уже упоминалось во второй главе, для описания семантических трансляций целесообразно использование объектного и реляционного подходов к трансляции данных между различными схемами их организации – различными предметными областями. Как обладающий чертами этих двух подходов, в качестве основы для языка спецификаций семантических трансляций был выбран синтаксис контекстных ключевых слов DSL LINQ из состава языка программирования C#.

LINQ содержит примитивы для взаимодействия с множествами сущностей, составляющих экземпляр модели предметной области в реляционном стиле, а элементы языка C# позволяют описывать как экземпляры, порождаемые в ходе семантической трансляции новых сущностей, так и выражения, задающие пути обращения к полям сущностей исходной семантической модели. При этом, поскольку синтаксис LINQ описывает именно способ связи элементов данных исходного и порожденного наборов сущностей, такая спецификация семантической трансляции не задаёт конкретного алгоритма, по которому должна следовать процедура трансляции, но декларативно описывает элементы предметных областей, являющиеся источниками данных и их получателями, подобно SQL-запросам.

Конкретная реализация механизма семантических трансляций может использовать предоставляемую в виде спецификаций декларативную информацию различными способами, например, для непосредственной интерпретации в ходе семантической трансляции или для генерации

статически типизированного программного кода, реализующего заданную семантическую трансляцию.

Предложенный язык описания семантических трансляций, представляет собой набор запросов к семантической модели, описывающих перенос данных в следующую модель в процессе трансляции. Синтаксически этот набор запросов организован аналогично спецификации предметной области, в которой каждой сущности заданы необходимые для её порождения параметры, а каждому полю сущности сопоставлено инициализирующее выражение, задающее источник данных для этого поля. Эти выражения могут рассматриваться как комбинация LINQ с элементами языка C# в окружении элементов спецификации предметной области. Рассмотрим фрагмент спецификации семантической трансляции из предметной области определений синтаксических моделей в предметную область семантики синтаксических моделей (см. пример описания целевой предметной области и грамматику из приложения А).

```

syntax {

    scope(s: scope) {
        parent = s;
        |namespace(s: scope, rs: pds1.ruleSet) base (s)
        from (null, rs) in
            pds1.definition.body.item.ruleSet {
                name = rs.complexName;
                namespaces = from crs in rs.body.item.ruleSet
                    select namespace(this, crs);
                rules = from r in rs.body.item.rule
                    select rule(this, r);
            }
        |rule(s: scope, r: pds1.rule) base (s)
        from (null, r) in
            pds1.definition.body.item.rule {

```

```

    name = r.complexName;
    rules = from cr in r.body.simple.rule
            select rule(this, cr);
    expr = syntax.expr(this, r.body.simple.expr);
}
};

}

```

Описание сущности целевой предметной области снабжается списком аргументов, который должен быть удовлетворён для её порождения при семантической трансляции. Этот список аргументов используется для проверки корректности частей спецификации, задающих порождение данной сущности. По ним можно также судить о корректности части текста, являющейся источником значений, связываемых с данными аргументами. В ходе трансляции создаётся узел семантической сети, значения полей и отношения которого установлены в соответствии с инициализирующими выражениями.

Для каждого поля сущности предметной области должно быть задано инициализирующее выражение. В качестве него выступает LINQ-запрос, описывающий перенос элементов данных в ходе семантической трансляции (15). Между списком аргументов конструктора сущности и списком инициализаторов её полей также может размещаться LINQ-запрос, используемый как корень процесса трансляции. Такие корневые запросы применяются на первой итерации алгоритма трансляции семантической модели.

В результате спецификация описывает семантическую трансляцию, начиная с корневых запросов, которые отвечают за материализацию первичных сущностей. Так как их материализация требует удовлетворения инициализирующих выражений, как следствие, будут выполнены трансляции, описанные этими выражениями, что вызовет порождение сущностей, на которые данная сущность должна содержать непосредственные ссылки. Затем

должны быть удовлетворены инициализирующие выражения этих вторичных сущностей, и так далее, пока вся целевая семантическая модель не будет материализована.

Выводы по главе 3

1. На основе анализа существующих языков описания спецификаций синтаксических и семантических моделей с учетом ограничений и специфики анализа текстов программ, написанных с применением нескольких языков программирования, был разработан предметно-ориентированный язык описания семантических моделей программ, позволяющий единожды выполнять описание спецификаций синтаксических и семантических моделей для последующего их применения при анализе исходного кода различных программных проектов.
2. Широкое применение предлагаемого метода в перспективе возможно при создании репозитория спецификаций синтаксических и семантических моделей, который позволит совместно использовать единожды составленные спецификации для типовых этапов предметно-ориентированного анализа, таких как разбор исходных текстов программ и программных систем, написанных на нескольких языках программирования, что позволит частично или полностью исключить необходимость написание новых спецификаций.

Материалы, изложенные в данной главе, соответствуют второму положению, выносимому на защиту: «Язык описания семантических моделей программ, позволяющий настраивать алгоритмы анализа текста под конкретные программные проекты». Основные результаты изложены в работе [63].

ГЛАВА 4 Алгоритм итеративного преобразования семантических моделей программ

Для проверки работоспособности предлагаемого метода и способов описания различных спецификаций были созданы необходимые алгоритмы и структуры данных. Рассмотрим некоторые аспекты программной реализации предлагаемых методов.

4.1 Разработка алгоритма и структур данных

Применение метода предметно-ориентированного анализа исходных текстов программ на основе семантических моделей требует автоматизированного выполнения семантических трансляций (32) в соответствии с принципами, обсуждаемыми во второй и третьей главах. В свою очередь, чтобы приступить к семантическим трансляциям, необходимо вначале прочитать заданные спецификации синтаксических и семантических моделей. Для этого были разработаны и реализованы объектные модели, предназначенные для размещения информации, извлекаемой из спецификаций в форме текстов на разработанном языке описания семантических моделей. На рисунке 10 представлена диаграмма классов объектной модели спецификаций синтаксических моделей.

Для работы синтаксического анализатора необходима синтаксическая спецификация. В момент начала работы прочтение синтаксических спецификаций невозможно, так как для этого должна быть загружена их синтаксическая модель. Поэтому инициализация анализатора требует другого источника спецификаций. Для этого была реализована поддержка представления синтаксических моделей в форме XML-документов. Такой XML-документ включен в состав программной библиотеки как ресурс. При инициализации библиотеки во время первой попытки её использования происходит десериализация данного XML-документа, из которого восстанавливается объектная модель грамматики. В дальнейшем она

используется для инициализации синтаксического анализатора и чтения спецификаций синтаксических моделей.

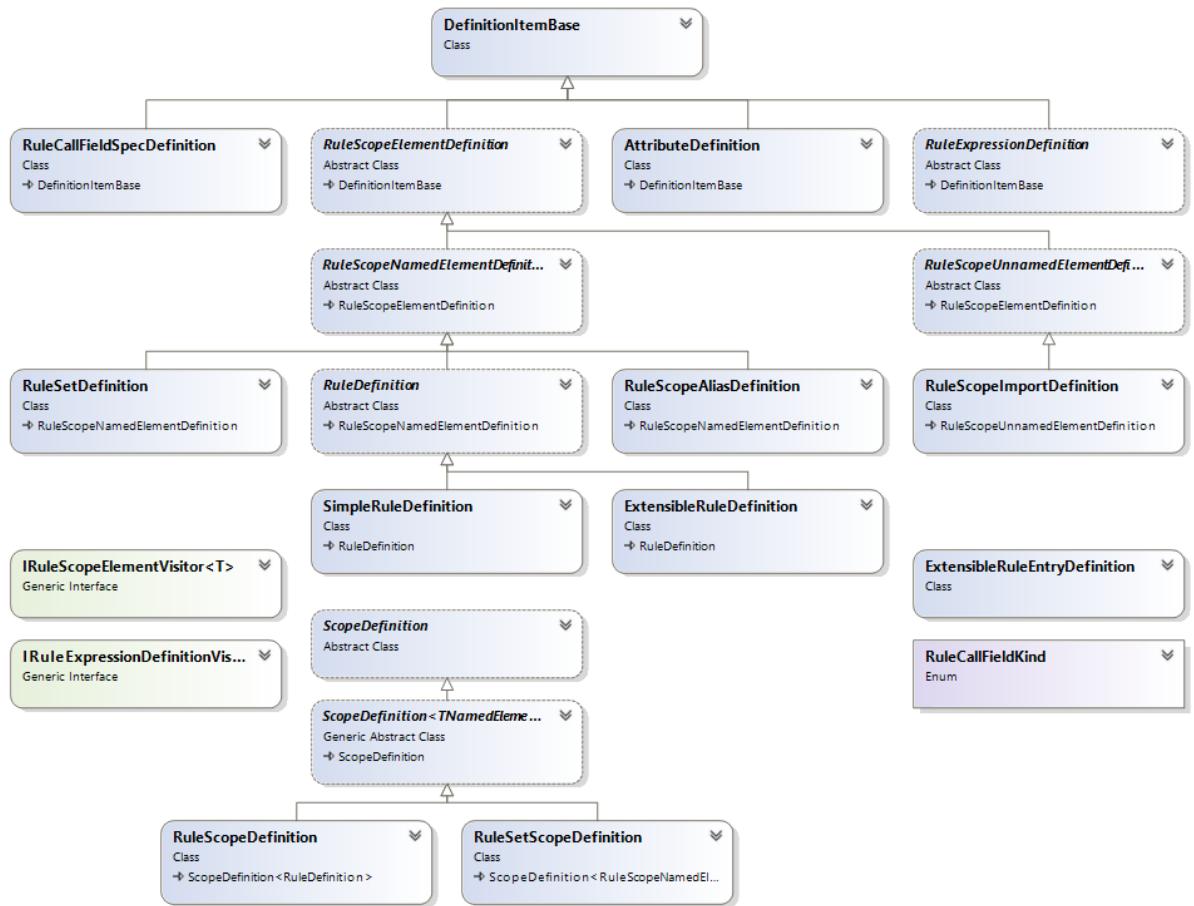


Рисунок 10 – Диаграмма классов объектной модели спецификаций синтаксических моделей

Семантическая сеть инициализируется с использованием наборов схем, задаваемых посредством спецификаций семантических моделей поэтапно. На основе спецификаций строится объектная модель схемы предметной области. Она содержит схемы для каждой сущности (1), входящей в состав предметной области. Все необходимые схемы вносятся в структуры данных семантической сети, отвечающие за уровень метамодели, где на их основе формируется словарь схем предметных областей. Далее загружаются спецификации необходимых семантических трансляций (14). Теперь

семантическая сеть готова к использованию для предметно-ориентированного анализа посредством итеративного преобразования семантических моделей.

Рассмотрим алгоритм итеративных преобразований.

1. На основе результатов синтаксического анализа текста или управляющих команд текстового редактора формируется набор мутаций, описывающих производимые над состоянием семантической сети изменения. На этом шаге посредством мутаций формируется абстрактный синтаксический граф для каждого входного файла в терминах соответствующей ему синтаксической спецификации.
2. Набор мутаций применяется к семантической сети, итеративно применяя каждую мутацию и накапливая информацию о фактически внесённых изменениях. Состояние фрагмента семантической сети при этом изменяется.
3. В результате формируется набор классифицированных изменений, отражающих фактически внесённые в состояние семантической сети изменения. Все мутации, попавшие в этот набор, принадлежат исходной предметной области.
4. Из известных запросов семантических трансляций выбираются те, исходная предметная область которых соответствует мутациям в текущем наборе изменений.
5. Из данного множества запросов посредством пересечения множества затрагиваемых ими сущностей-источников и множества сущностей из имеющегося набора изменений, затронутых мутациями, выделяется подмножество запросов, результаты выполнения которых в нынешнем состоянии семантической сети должны отличаться от предыдущего.
6. Выделенные запросы применяются, на основании их результатов:
 - 1) накапливается набор мутаций для внесения изменений в фрагмент семантической сети, соответствующий целевой семантической модели данного этапа трансляции;

- 2) формируется подмножество запросов, параметры или исходные данные которых были затронуты применёнными на данном шаге преобразованиями семантической модели.
7. Для нового подмножества запросов шаги пять и шесть повторяются до тех пор, пока подмножество подходящих запросов в текущей трансляции не пустое.
8. Шаги со второго по седьмой повторяются для накопленного набора мутаций.

В результате применения данного алгоритма осуществляется ряд преобразований семантических моделей. В ходе первой итерации данного алгоритма выполняется перенос данных от исходного абстрактного синтаксического графа в терминах синтаксической модели (12) к семантической модели (5), которая является предметом анализа (рисунок 6). В ходе последующих итераций выполняется перенос данных между семантическими моделями, для последовательности которых имеются спецификации семантических трансляций (31).

Поскольку перенос информации всегда осуществляется в направлении семантической трансляции от предыдущего экземпляра семантической модели к последующему, зацикливание на уровне шагов со второго по седьмой исключено. Зацикливание на уровне шагов пять и шесть исключено в силу логики обработки осуществляющего выборку данных запроса.

Интеграция данного алгоритма в процесс редактирования текста зависит от способа представления текста в текстовом редакторе. На рисунке 11 представлен обобщенный вид алгоритма семантического анализа текста программы в процессе его редактирования для случая использования совместных с текстовым редактором структур данных, как это предложено в части 2.2.

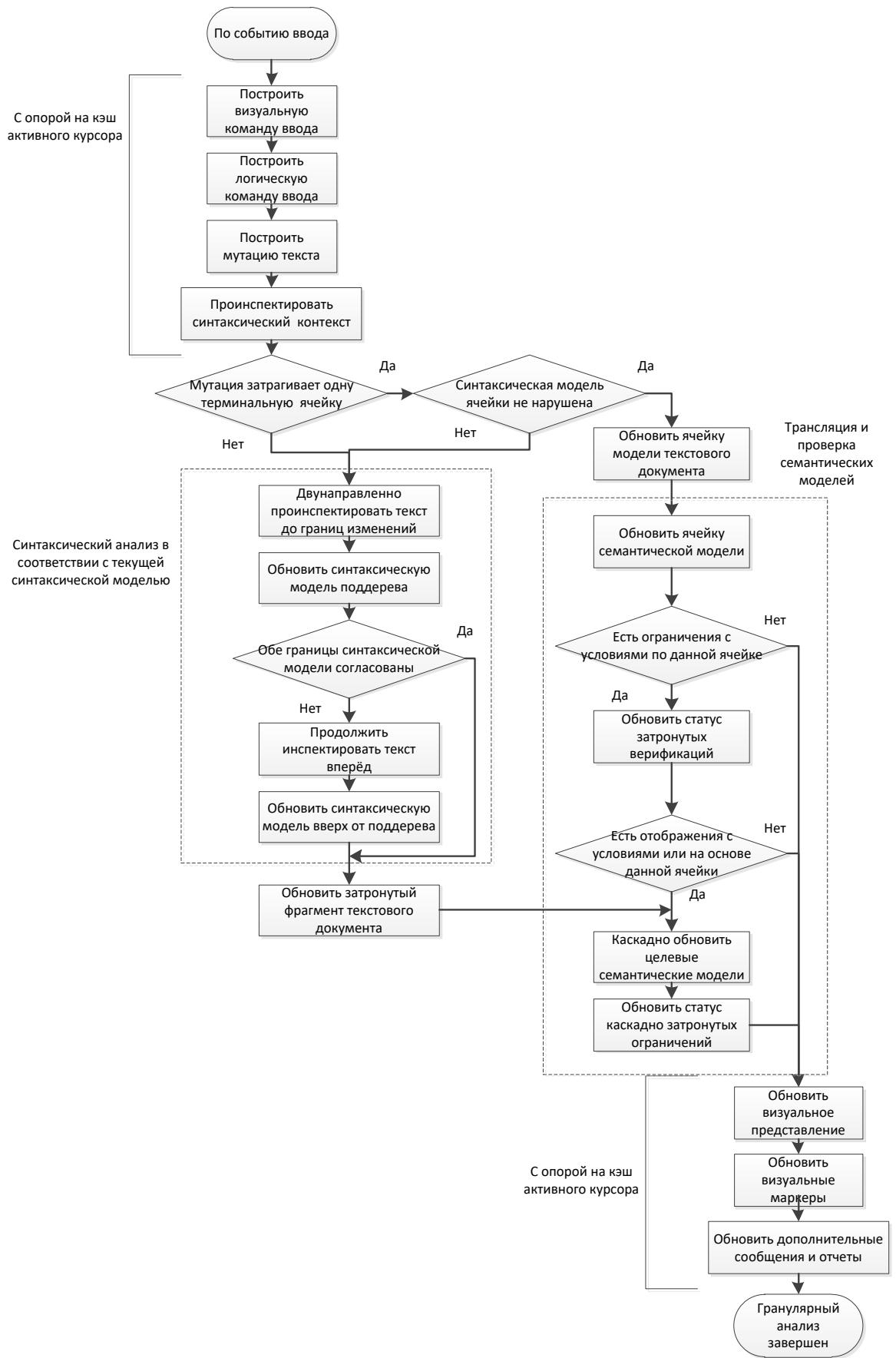


Рисунок 11 – Обобщенный вид алгоритма семантического анализа текста программ в процессе их редактирования

В качестве структуры данных, отвечающей за внутреннее представление текста, была реализована представленная на рисунке 12 гибридная структура данных, являющаяся комбинацией нескольких разных структур.

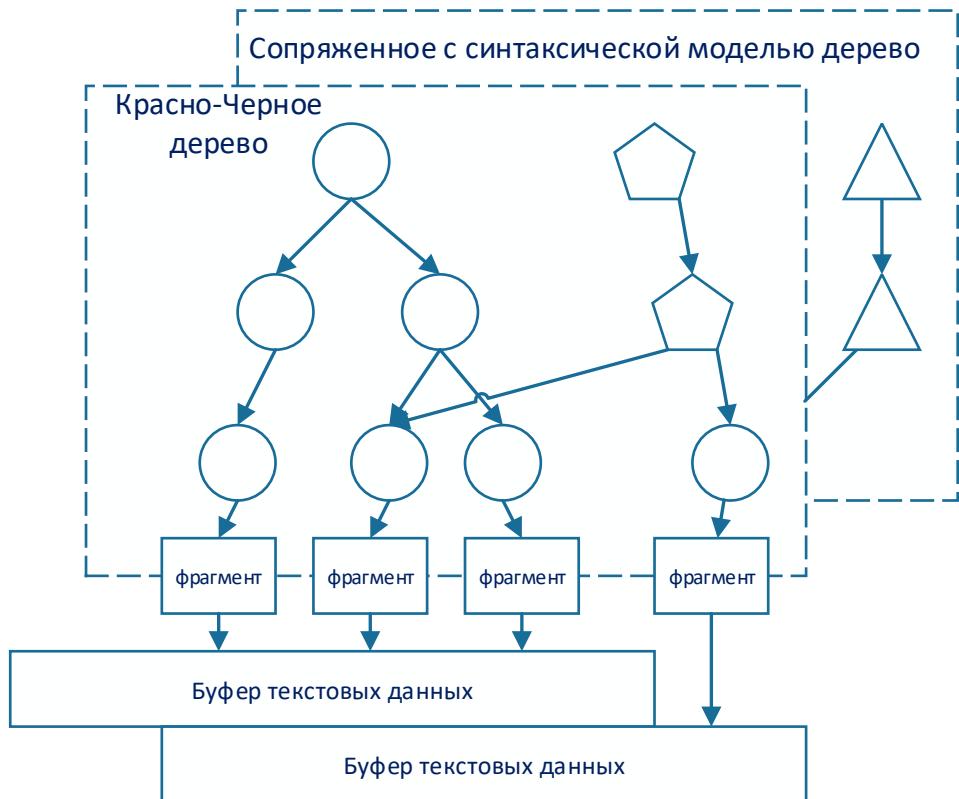


Рисунок 12 – Структура данных текстового документа внутри
редактора

Для размещения символьных данных, из которых составлен редактируемый текст, использованы выделенные буфера аналогично структуре данных Piece-Table. По мере редактирования и задействования новых фрагментов буфера в момент размещения нового фрагмента, слева и справа от него резервируется свободное место, которое далее используется при вставке символов в смежные с данным фрагментом позиции. Это обеспечивает сложность вставки символа в текст порядка $O(1)$ для изменений текста, не влияющих на его синтаксическую структуру.

Дробление текста на фрагменты выполняется с учётом лексем заданной синтаксической модели так, чтобы границы фрагментов соответствовали

границам лексем. Это обеспечивает возможность привязки стилевой информации к узлам фрагментов текста. Так устраняется необходимость перестроения этой информации при визуализации текста.

Фрагменты текста объединены в единую структуру данных одновременно иммутабельным красно-черным деревом и деревом на основе синтаксической модели, как предложено в части 2.2. Ключом для сортировки в красно-черном дереве выступает позиция в форме строка-колонка. Это позволяет сохранить быструю навигацию по тексту большого объёма.

Иммутабельные структуры данных имеют недостаток, заключающийся в необходимости частого клонирования их частей при внесении изменений. Это компенсируется за счет, использования «ленивого» клонирования пути. Положению курсора соответствует материализованный параллельно основному дереву фрагмент пути от коня дерева до узла, содержащего фрагмент, где в текущий момент расположен курсор. При внесении изменения в текст, если был запрос снимка текущей версии текста, порождается новое состояние фрагмента, включая лист дерева, при этом весь путь до корня не материализовывается. Если запроса снимка не было, то изменяется состояние текущего фрагмента и, возможно, листа дерева (если необходимо изменение его структуры). Клонирование части дерева при этом не требуется. Материализация пути, поддерево которого было изменено, выполняется по мере перемещения курсора за пределы отредактированного фрагмента, например, в момент перехода курсора от фрагмента текущей ветви к фрагменту текста, представленному листом другой ветви, в зависимости от текущей структуры дерева. Этот же подход применяется к дереву, отвечающему синтаксической структуре текста.

4.2 Интеграция решения в среды разработки

Были разработаны два основных варианта программной архитектуры для интеграции разработанных алгоритмов в среды разработки.

1. В качестве встраиваемого модуля.
2. В качестве внешнего сервиса.

При интеграции в качестве встраиваемого модуля реализация решения целиком загружается в процесс среды разработки или текстового редактора и функционирует в нём в режиме *in-proc*. Такой подход более эффективен с точки зрения производительности, так как не требует преобразования данных для передачи между процессом, импортирующим функциональность, и процессом, экспортирующим эту функциональность. Недостатком данного варианта программной архитектуры является необходимость использования независимых экземпляров синтаксического анализатора как в процессе с редактором, так и в процессе с семантическим анализатором. Данный вариант архитектуры представлен на рисунке 13. Основное отличие от существующих решений заключается во влиянии анализатора на внутреннее представление текста в текстовом редакторе. Такая интеграция связана с двумя особенностями.

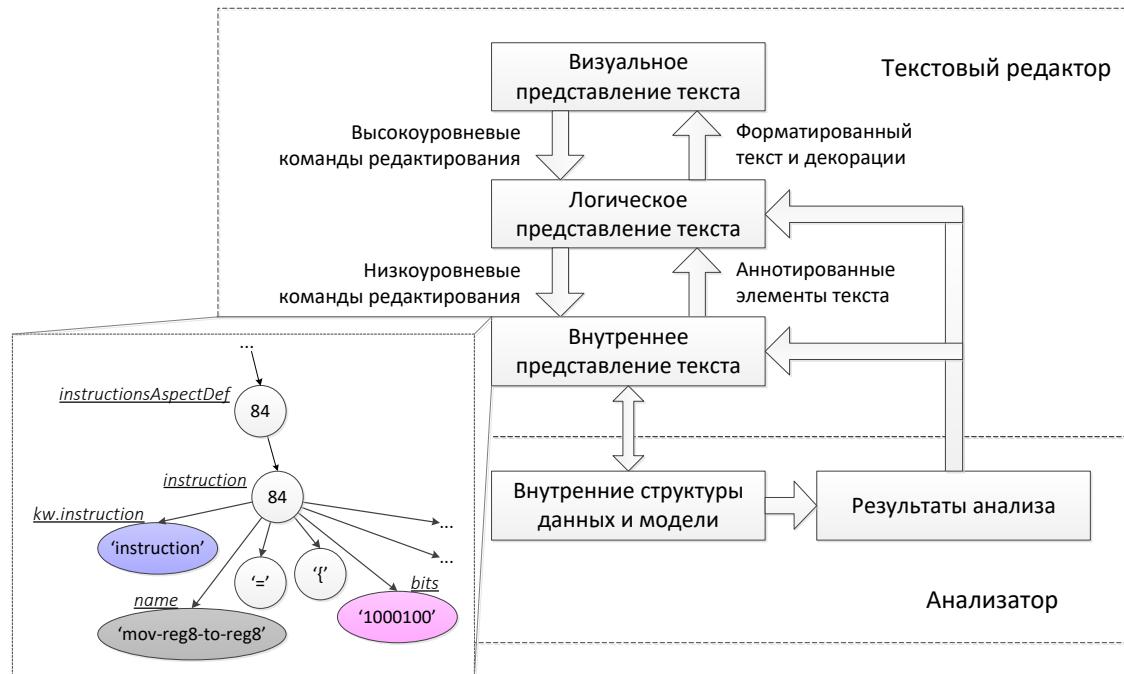


Рисунок 13 – Предлагаемая архитектура текстового редактора

Первая особенность заключается в модификации внутреннего представления текста в составе текстового редактора. Упомянутые во второй

главе структуры данных, используемые для представления текста, имеют лучшую асимптотику мутаций, чем текстовые строки, представленные массивами символов. Это достигается за счет разбиения текста на фрагменты и организацию этих фрагментов в структуру, операции редактирования которой выполнимы быстрее.

С точки зрения процесса синтаксического анализа, произвольное разбиение текста на фрагменты является причиной накладных расходов. Оно приводит к необходимости использования программного интерфейса, отображающего внутреннее представление текстового редактора на линейный текст, который анализатор должен разбить на лексемы, выделить синтаксическую структуру. При этом та же самая структура, формирующаяся в процессе анализа, отображается обратно в структуры данных, так как подсветка синтаксиса, блоки кода и аналогичная функциональность опираются на информацию о синтаксической и семантической структуре текста. По этой причине целесообразно использовать анализатор для разбиения текста на фрагменты в соответствии с правилами, определяемыми лексической и синтаксической моделями редактируемого текста. Это реализуемо при использовании информации об изменениях текста анализатором для выявления того, как должно измениться внутреннее представление, чтобы остаться соответствующим логической структуре текста.

Вторая особенность заключается в том, что результаты семантического анализа ассоциируются с теми же самыми синтаксическими единицами, что были выделены в тексте на первых этапах анализа. Целесообразно ассоциировать их не с дополнительными структурами данных, а с атрибутами, привязанными к элементам уже существующих представлений текста. Так, информация о форматировании и подсветке текста может ассоциироваться с внутренним представлением текста. В результате пропадает необходимость в повторном запросе этой информации или ее повторном вычислении в

процессе вывода текста на экран. То же относится и к другим операциям, связанным с извлечением данных из результатов семантического анализа текста одновременно с учётом позиций в тексте. Например, к операциям навигации по коду программы.

При интеграции в качестве внешнего сервиса, среда разработки или текстовый редактор взаимодействует с семантическим анализатором посредством механизмов межпроцессного взаимодействия (через разделяемую память, сетевое соединение или канал). Пример программной архитектуры представлен на рисунке 14.

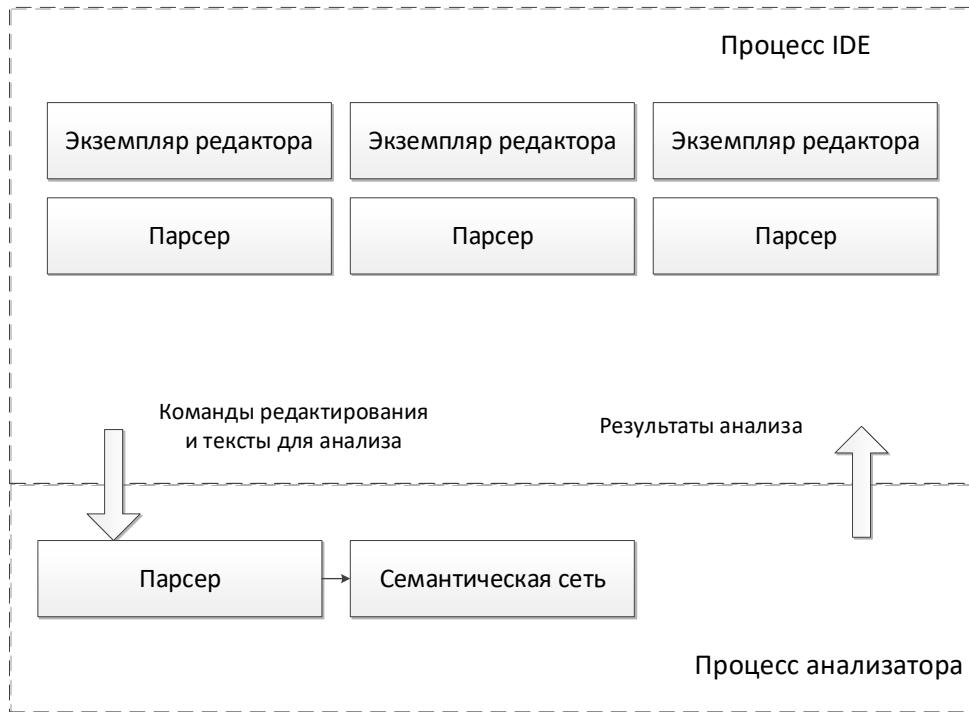


Рисунок 14 – Иллюстрация программной архитектуры для случая интеграции в качестве внешнего сервиса с текстовым редактором на основе предлагаемых структур данных

Главным преимуществом является то, что такая программная архитектура позволяет использовать один экземпляр семантического анализатора с несколькими экземплярами среды разработки одновременно. Это более эффективно с точки зрения потребляемой памяти, так как

отсутствует необходимость дублирования представлений моделей программ. Все заинтересованные в их анализе приложения (например, экземпляры среды разработки) могут использовать общую семантическую сеть. Кроме этого, такой подход позволяет исключить повторный анализ текста неизменённой программы в том случае, когда процесс среды разработки был аварийно завершен.

Оценка принципиальной возможности применения предлагаемого метода не зависит от объёма анализируемых данных. Вариант с анализатором в одном процессе проще и не обременён накладными расходами от межпроцессного взаимодействия, поэтому для создания экспериментального анализатора был выбран именно он.

4.3 Апробация разработанного решения

Для разработки экспериментального средства предметно-ориентированного анализа исходных текстов программ были выбраны язык программирования C# и IDE Visual Studio 2017 по двум причинам:

- 1) они входят в список наиболее широко используемых в соответствии с данными Google Trends языков программирования и интегрированных сред разработки;
- 2) автор владеет данными инструментами на профессиональном уровне, что позволяет сосредоточиться на решении задач исследования.

На базе представленных методов, алгоритмов и структур данных был создан конфигурируемый текстовый редактор (рисунок 15) с поддержкой анализа исходных текстов программ в процессе их редактирования. Данный редактор был интегрирован в среду IDE Visual Studio 2017 посредством создания расширения (рисунок 16).

```

calc.h.pDSL
coloring.pDSL coloring.h.pd 100%
1 !default {
2     color: #000000;
3     background: #ffffff;
4 }
5
6 num {
7     color: #0000ff;
8 }
9
10 sum, product {
11     color: #008800;
12 }
13
14 braces {
15     background: #00ffff;
16 }
17
18
19
20

calcgram-new.pDSL
defgrammar.pDSL (Portable.Parser\Def definition.h.pDSL (Po
1 [OmitPattern("[\s]*")]
2 [RootRule(expr)]
3 SimpleArithmetics {
4     /*Demonstration of the simple features*/
5     [RewriteRecursion]
6     #expr: {
7         |sum: expr ('+' | '-') expr;
8         |product: expr ('*' | '/') expr;
9         |[right]power: expr '^' expr;
10       |#braces: '(' expr ')';
11        |num: "[0-9]+";
12    };
13 }
14

screensample.pDSL
calcgram-new calc.h.pDSL (Pc 100%
1 1 + 2 ^ 3 + 5 * (2 - 9)

```

Рисунок 15 – Созданный текстовый редактор в среде Visual Studio 2017

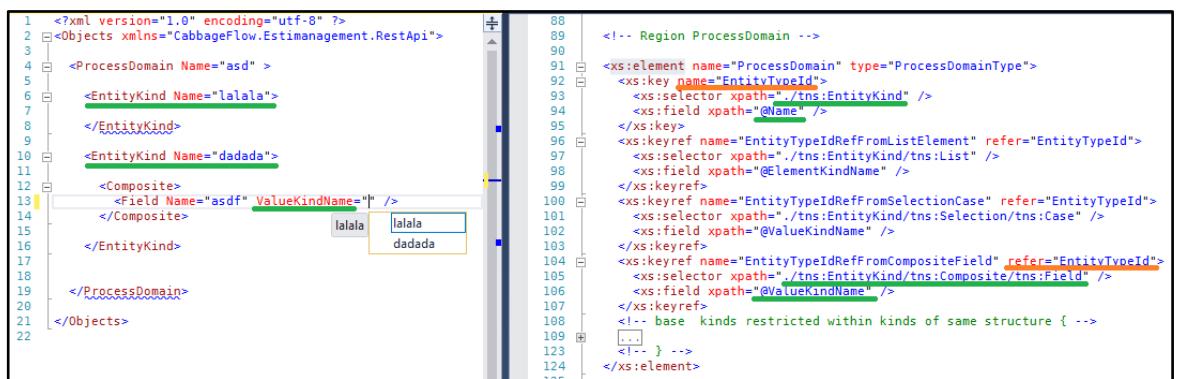


Рисунок 16 – Интеграция отдельной реализации предметно-ориентированного анализа на базе семантических моделей со встроенным редактором среды Visual Studio 2017

Созданное решение было апробировано в учебном процессе и научно-исследовательской деятельности на факультете программной инженерии и компьютерной техники Университета ИТМО, а также в рамках выполнения НИР-ФУНД № 619296 «Разработка методов создания и внедрения киберфизических систем» при разработке языков описания целевых архитектур для конфигурируемых компиляторов.

В ходе апробации было выполнено сравнение среднего времени между изменением текста и отображением результатов анализа на компьютере оснащенном Xeon E5-1620 3.6GHz 16GB RAM для имеющегося и предложенного решений в двух сценариях использования.

1. Проверка соответствия типов XML-схемы классам языка C#, используемым для автоматической сериализации в соответствии с данной схемой.
2. Проверка корректности ключей XML-документа относительно keyref-ограничений соответствующей XML-схемы.

Сравнение двух сценариев представлено на рисунке 17.

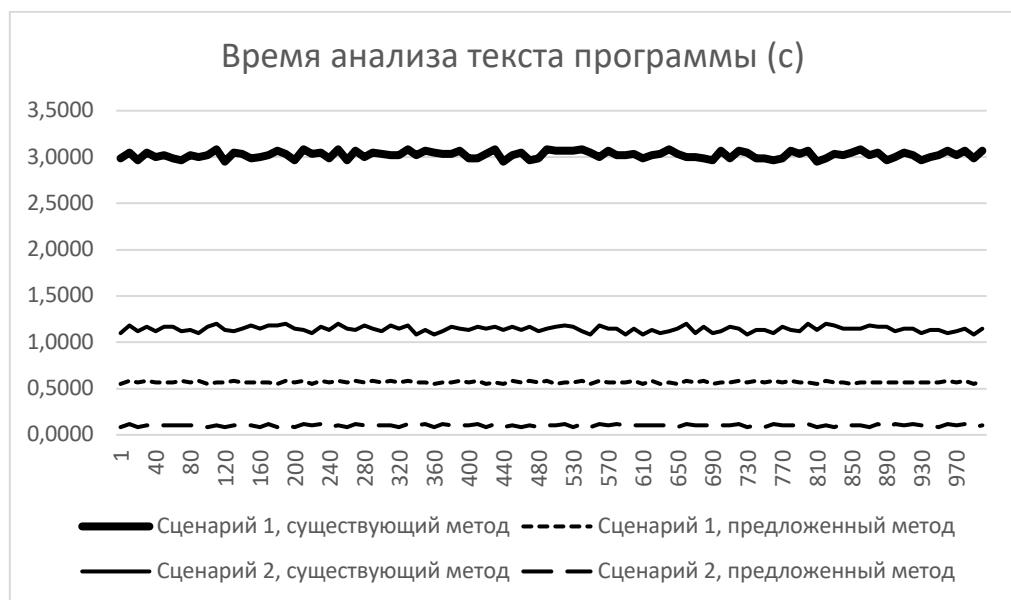


Рисунок 17 – Иллюстрация последовательности состояний среды разработки в ходе эксперимента для сценария 2

Ниже приведен порядок и условия проведения экспериментов:

- 1) в среде разработки создается проект с экспериментальными данными;
- 2) включается видеозапись с экрана в формате 60 кадров/с и запускается скрипт, эмулирующий сценарий внесения изменения в исходный текст программы 1000 раз;

- 3) в ходе эксперимента на экране в определённых местах изменяется подсветка синтаксиса, появляются и исчезают сообщения об ошибках;
- 4) после завершения эксперимента видеозапись покадрово анализируется и вычисляется интервал времени между вводом, запускающим анализ, и визуальной обратной связью, формирующейся в результате анализа и отображающейся на экране.

Для первого сценария проведения эксперимента пользователем выполнялись следующие действия:

- 1) добавление в код описания класса, соответствующего типу XML-схемы;
- 2) сборка проекта.

Для второго сценария проведения эксперимента пользователем выполнялись изменения в XML-документе значения для атрибута, схема которого имеет keyref-ограничение.

В обоих сценариях оценивалось время между началом ввода и появлением сообщения об ошибке в тексте программы.

Инструментальная погрешность измерения составляет $\delta = \frac{1\text{с}}{60*2} \sim 8.3333\text{мс.}$

Для каждого сценария оценивалось среднее время анализа и относительная погрешность измерений. Результаты экспериментов приведены в таблице 4.

Таблица 4 – Фрагмент результатов экспериментов (время, с)

Сценарий	Соответствие типов XMLSchema классам C# в Visual studio 2017		Корректность ключей XML-документа относительно keyref-ограничений XML-схемы	
Номер эксперимента	Существующее решение	Предложенное решение	Существующее решение	Предложенное решение
1	2.9833	0.5500	1.1000	0.0833
2	3.0500	0.5833	1.1833	0.1167
3	2.9667	0.5667	1.1167	0.0833
4	3.0500	0.5833	1.1667	0.1000
5	3.0000	0.5667	1.1167	0.1000
6	3.0167	0.5667	1.1667	0.1000
7	2.9833	0.5667	1.1667	0.1000
8	2.9667	0.5833	1.1167	0.1000
9	3.0167	0.5667	1.1333	0.1000
10	3.0000	0.5833	1.1000	0.1000
11	3.0167	0.5500	1.1667	0.0833
12	3.0833	0.5667	1.2000	0.1000
13	2.9500	0.5667	1.1333	0.0833
14	3.0500	0.5833	1.1167	0.1000
15	3.0333	0.5667	1.1500	0.1000
16	2.9833	0.5667	1.1833	0.1000
17	3.0000	0.5667	1.1500	0.0833
18	3.0167	0.5667	1.1833	0.1167
19	3.0667	0.5500	1.1833	0.0833
20	3.0333	0.5833	1.2000	0.1000
21	2.9667	0.5667	1.1500	0.0833
22	3.0833	0.5833	1.1333	0.1167
23	3.0333	0.5500	1.1000	0.1000
24	3.0500	0.5833	1.1667	0.1167
25	2.9833	0.5667	1.1333	0.0833
26	3.0833	0.5833	1.2000	0.1000
27	2.9667	0.5667	1.1500	0.0833
28	3.0667	0.5833	1.1333	0.1167
...
\bar{x}	3.0232	0.5693	1.1433	0.0997
$\Delta_{\Sigma}x$	0.0112	0.0086	0.0104	0.0087
δ	0.3710%	1.5146%	0.9104%	8.6801%

В ходе оценки использованы следующие соотношения:

- среднее арифметическое исследуемой величины

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n} \quad (40)$$

- среднеквадратическое отклонение среднего арифметического исследуемой величины

$$S_{\bar{x}} = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n(n-1)}} \quad (41)$$

- суммарная погрешность результатов измерений

$$\Delta_{\Sigma}x = \sqrt{\delta^2 + (t_{0.05,1000}S_{\bar{x}})^2} \quad (42)$$

- относительная погрешность результатов измерений

$$\delta = \frac{\Delta_{\Sigma}x}{\bar{x}} \quad (43)$$

где x_i – время между изменением текста и отображением результатов анализа в i -ом эксперименте.

С целью уменьшения погрешностей на время экспериментов все файлы тестового проекта помещались на виртуальный логический диск, расположенный в оперативной памяти. По результатам эксперимента было рассчитано среднее время выполнения каждого сценария для существующего и предложенного решения. Эксперимент показал пятикратное преимущество в пользу предложенного решения в случае первого сценария постановки эксперимента ($3,0232/0,5693 \sim 5$) и одиннадцатикратное в случае второго сценария ($1,1433/0,0997 \sim 11$).

На рисунке 18 приведена иллюстрация последовательности состояний среды разработки в ходе эксперимента для второго сценария. Показанные состояния соответствуют четырем моментам времени: 1 – перед началом ввода, 2 – через 0.02с после начала ввода, 3 – через ~ 0.1с после начала ввода, 4 – через ~ 1.1с после начала ввода. Видно, что в начале появляется подсветка синтаксиса, свидетельствующая о нарушении семантической модели предложенного анализатора, затем сообщение об ошибке от него, поясняющее причину этого нарушения. Через секунду после этого появляется сообщение об ошибке от встроенного анализатора среды разработки.

Изучение восприятия пользователем времени реакции программной системы не является предметом данной работы. Для разных приложений, контекстов использования, пользователей и их психофизиологических состояний заметным будет считаться различное время реакции. Например, в статье «How much faster is fast enough? user perception of latency & latency improvements in direct and indirect touch» [35] показано, что изменение времени реакции программной системы на 16.7мс будет заметно для пользователя.

Чем раньше пользователь заметит визуальный стимул – тем быстрее среагирует на него. В реальном случае работы с кодом программной системы время между вводом пользователя, вносящим в текст программы ошибку, и реакцией анализатора может увеличиваться за счет следующих факторов:

- 1) отложенный вызов анализатора во время заметной паузы во вводе пользователя (до нескольких секунд), из-за чего потребует вернуться в другую часть текста на экране для исправления ошибки;
- 2) отложенный вызов анализатора до времени сборки проекта может быть сопряжен с полной потерей программистом логического контекста, требуемого для коррекции ошибки;
- 3) формирование обратной связи от анализатора в незаметном месте экрана или, например, за пределами экранной области текста, редактируемой к моменту получения результатов анализа.

1

```

207 <Rule Name="sourceItem">
208   <Alts>
209     <Call RuleName="sourceItem.funcDef" />
210     <Call RuleName="sourceItem.classDef" />
211   </Alts>
212 </Rule>
213
214 <Rule Name="sourceItem.funcDef">
215   <Seq>
216     <Chars String="method" />
217     <Call RuleName="funcSignature" />
218     <Alts>
219       <Chars String=";" />
220       <Call RuleName="sourceItem.fun">
221       <Call RuleName="sourceItem.fun">
222     </Alts>
223   </Seq>

```

2

```

207 <Rule Name="sourceItem">
208   <Alts>
209     <Call RuleName="sourceItem.funcDef" />
210     <Call RuleName="sourceItem.classDef" />
211   </Alts>
212 </Rule>
213
214 <Rule Name="errsourceItem.funcDef">
215   <Seq>
216     <Chars String="method" />
217     <Call RuleName="funcSignature" />
218     <Alts>
219       <Chars String=";" />
220       <Call RuleName="sourceItem.fun">
221       <Call RuleName="sourceItem.fun">
222     </Alts>
223   </Seq>

```

3

```

207 <Rule Name="sourceItem">
208   <Alts>
209     <Call RuleName="sourceItem.funcDef" />
210     <Call RuleName="sourceItem.classDef" />
211   </Alts>
212 </Rule>
213
214 <Rule Name="errsourceItem.funcDef">
215   <Seq>
216     <Chars String="method" />
217     <Call RuleName="funcSignature" />
218     <Alts>
219       <Chars String=";" />
220       <Call RuleName="sourceItem.fun">
221       <Call RuleName="sourceItem.fun">
222     </Alts>
223   </Seq>

```

4

```

207 <Rule Name="sourceItem">
208   <Alts>
209     <Call RuleName="sourceItem.funcDef" />
210     <Call RuleName="sourceItem.classDef" />
211   </Alts>
212 </Rule>
213
214 <Rule Name="errsourceItem.funcDef">
215   <Seq>
216     <Chars String="method" />
217     <Call RuleName="funcSignature" />
218     <Alts>
219       <Chars String=";" />
220       <Call RuleName="sourceItem.fun">
221       <Call RuleName="sourceItem.fun">
222     </Alts>
223   </Seq>

```

Состояние	Код	Описание
1		Все решения 0 Ошибки 0 Предупреждения
2		Все решения 0 Ошибки 0 Предупреждения
3		Все решения 0 Ошибки 0 Предупреждения
4		Все решения 0 Ошибки 3 Предупреждения

Рисунок 18 – Иллюстрация последовательности состояний среды

разработки в ходе эксперимента для сценария 2

(1: 0с, 2: ~0.02с, 3: ~0.1с, 4: ~1.1с)

На рисунке 19 показано среднее время между изменением текста и отображением результатов анализа. Всё это делает целесообразным уменьшение времени между вводом и получением результатов анализа.

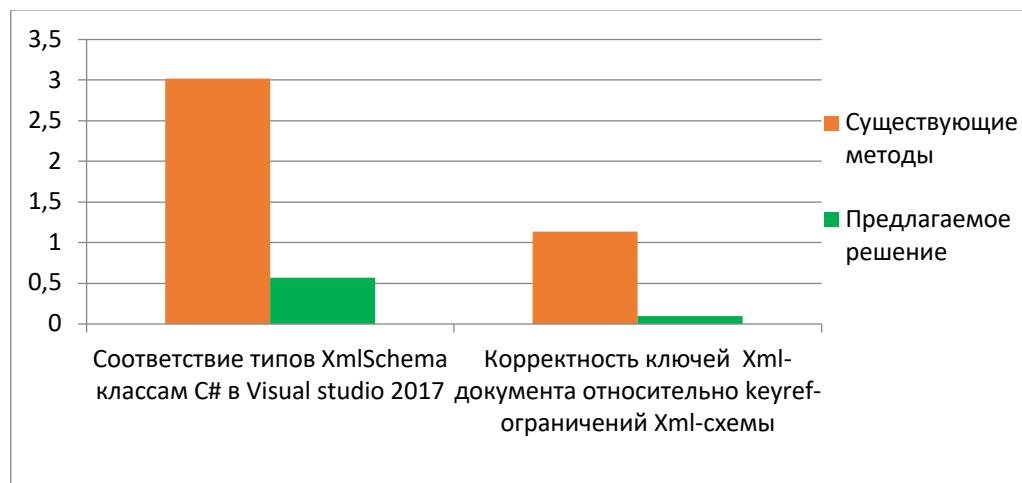


Рисунок 19 – Среднее время между изменением текста и отображением результатов анализа (сек.)

Таким образом, результаты сравнений и аprobации показали, что для рассмотренных случаев:

- 1) время проверки согласованности кода программ в мультиязыковых программных проектах уменьшено;
- 2) решена проблема неактуальной обратной связи от анализа;
- 3) расширен спектр доступного разработчику анализа семантики исходных текстов программ.

В целом, аprobация показала работоспособность предлагаемых методов.

Выводы по главе 4

1. Для реализации предложенного метода предметно-ориентированного анализа текста программ был разработан алгоритм итеративных преобразований семантических моделей программ, позволяющий выполнять анализ частей текстов, так как не требуется выявлять

измененные фрагменты текста, что в целом повышает быстродействие реализации текстового редактора.

2. Для представления динамических семантических моделей программ разработана программная архитектура и модифицированные структуры данных, которые были использованы при создании средства предметно-ориентированного анализа исходных текстов программ, реализованного на основе предложенного метода. В результате создан конфигурируемый текстовый редактор, встроенный в IDE Visual Studio с поддержкой реактивного анализа исходных текстов программ в процессе их редактирования.
3. Реализация программной архитектуры и модифицированных структур данных для единого внутреннего представления текста программы и сопутствующей информации, необходимой для анализа исходного кода, обеспечила одновременную работу редактора и анализатора текста, что позволило устранить определённый класс ошибок, связанных с некорректной ассоциацией фрагмента текста и сопутствующей информации, например, неправильная подсветка частей текста, ошибочные варианты автоматического дополнения.
4. Апробация полученных результатов показала перспективность их применения в текстовых редакторах и интегрированных средах разработки для анализа текстов программ и программных систем, разрабатываемых с использованием нескольких языков программирования, а также подтвердила теоретическую значимость в области развития новых подходов к созданию адаптируемых средств разработки программного обеспечения.

Материалы, изложенные в данной главе, соответствуют третьему положению, выносимому на защиту: «Алгоритм реактивного анализа и итеративного преобразования семантических моделей программ». Основные результаты изложены в работах [1, 3, 4, 63].

Заключение

Проведенное в рамках диссертационной работы исследование показало, что готовых решений для выполнения предметно-ориентированного анализа текстов программ и программных систем, разрабатываемых с использованием нескольких языков программирования, интегрированных в существующие текстовые редакторы исходного кода программ или интегрированные среды разработки, не существует. Для решения проблемы предложен метод предметно-ориентированного анализа на основе семантических моделей программ, разработан алгоритм итеративных преобразований и язык описания семантических моделей, создано экспериментальное программное средство редактирования и анализа текстов программ. Результаты работы внедрены в учебный процесс на факультете программной инженерии и компьютерной техники, а также использованы в научно-исследовательской деятельности в рамках выполнения НИР-ФУНД № 619296 «Разработка методов создания и внедрения киберфизических систем» при разработке языков описания целевых архитектур для конфигурируемых компиляторов.

В диссертационной работе были получены следующие результаты.

1. Разработан метод предметно-ориентированного анализа на основе итеративных преобразований семантических моделей программ, позволяющий осуществлять семантический анализ текстов программ, разрабатываемых с применением нескольких языков программирования.
2. Разработан язык описания семантических моделей программ, позволяющий адаптировать алгоритм анализа семантических моделей программ к специфике конкретных программных проектов.
3. Разработан алгоритм анализа семантических моделей программ, позволяющий уменьшить время отклика среды разработки или текстового редактора между вводом пользователя и отображением результатов анализа актуального состояния программного кода.

4. Разработана программная архитектура и структуры данных для представления текста и динамически изменяющихся семантических моделей программ, что позволяет интегрировать реализацию разработанного метода предметно-ориентированного анализа в различные среды разработки.
5. Разработан анализатор исходных текстов программ, допускающий его интеграцию с существующими средами разработки и прошедший апробацию в составе реализованного конфигурируемого текстового редактора при решении практической задачи анализа текстов программ, разрабатываемых с использованием нескольких языков программирования.

Дальнейшее развитие работы возможно в области создания глобально распределенной программно-информационной инфраструктуры поддержки программных проектов, включающей задачу создания репозитория синтаксических и семантических моделей для наиболее распространенных в индустрии разработки программного обеспечения сочетаний языков программирования. Это позволит обеспечить совершенствование предложенного метода и расширить возможности по практическому применению. Дальнейшее исследование алгоритмов синтаксического анализа в части возможности их настройки позволит повысить производительность программной реализации метода, что важно для пользователей, поскольку обеспечивает отзывчивость графического интерфейса среды разработки.

Таким образом, задачи диссертационного исследования выполнены в полном объеме, а поставленная цель достигнута.

Список сокращений и условных обозначений

БНФ – Бэкуса-Наура форма

ПО – программное обеспечение

API – Application Programming Interface

DSL – Domain-Specific Language

GCC – GNU Compiler Collection

IDE – Integrated Development Environment

LINQ – Language Integrated Query

LLVM – Low-Level Virtual Machine

MPS – Meta Programming System

p-invoke – Platform Invoke

RDF – Resource Description Framework

SPA – Single-Page Application

SQL – Structured Query Language

VB – Visual Basic

XAML – Extensible Application Markup Language

XML – Extensible Markup Language

Список литературы

1. Дергачев А.М., Садырин Д.С., Ильина А.Г., Логинов И.П., Кореньков Ю.Д. Методы и средства обнаружения уязвимостей аллокаторов динамической памяти библиотеки glibc // Научно-технический вестник Поволжья - 2020. - № 1. - С. 79-83
2. Кореньков Ю.Д. Итеративное преобразование семантических моделей программ // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. – СПб: Университет ИТМО, 2020. - Режим доступа: <https://kmu.itmo.ru/digests/article/4180>, своб. - 2020
3. Кореньков Ю.Д. Метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей // Научно-технический вестник Поволжья - 2020 - № 7 - С. 32-37
4. Кореньков Ю.Д., Логинов И.П. Исследование и разработка языкового инструментария на основе PEG-грамматики // Известия высших учебных заведений. Приборостроение - 2015. - Т. 58. - № 11. - С. 934-938
5. Кореньков Ю.Д., Логинов И.П., Павлова Е.В. Исследование и реализация алгоритма синтаксического анализа // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. – СПб: Университет ИТМО, [2020]. Режим доступа: <https://kmu.itmo.ru/digests/article/3956> - 2020
6. Хомский Н., Миллер Д. Введение в формальный анализ естественных языков/Пер. с англ. ЕВ Падучевой //М.: Едиториал УРСС. – 2003.
7. Adrion W. R., Branstad M. A., Cherniavsky J. C. Validation, verification, and testing of computer software //ACM Computing Surveys (CSUR). – 1982. – Т. 14. – №. 2. – С. 159-192.
8. Albahari J., Albahari B. C# 7.0 in a nutshell: The definitive reference. – "O'Reilly Media, Inc.", 2017.
9. Barrasa J., Corcho Ó., Gómez-Pérez A. R2O, an extensible and semantically based database-to-ontology mapping language. – SWDB, 2004.

10. Barzdins J. et al. Domain specific languages for business process management: a case study //Proceedings of DSM. – 2009. – T. 9. – C. 34-40.
11. Batini C., Lenzerini M. A methodology for data schema integration in the entity relationship model //IEEE transactions on software engineering. – 1984. – №. 6. – C. 650-664.
12. Becchi M. Data structures, algorithms and architectures for efficient regular expression evaluation. – Washington University in St. Louis, 2009.
13. Beelders T., du Plessis J. P. The influence of syntax highlighting on scanning and reading behaviour for source code //Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists. – 2016. – C. 1-10.
14. Binkley D. Source code analysis: A road map //Future of Software Engineering (FOSE'07). – IEEE, 2007. – C. 104-119.
15. Biron P. V. et al. XML schema part 2: Datatypes. – 2004.
16. Breslav A. DSL development based on target meta-models. Using AST transformations for automating semantic analysis in a textual DSL framework //arXiv preprint arXiv:0801.1219. – 2008.
17. Chebotko A. et al. Semantics preserving SPARQL-to-SQL query translation for optional graph patterns //Wayne State University, Tech. Rep. TR-DB-052006-CLJF. – 2006.
18. Chen H., Luo Y. Object Automatic Serialization and De-serialization in C++ //proceedings of 2010 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010 no. 2). – 2012.
19. Chisnall D. The challenge of cross-language interoperability //Communications of the ACM. – 2013. – T. 56. – №. 12. – C. 50-56.
20. Chomsky N. Studies on semantics in generative grammar. – Walter de Gruyter, 2013. – T. 107.
21. Chomsky N. Syntactic structures. – Walter de Gruyter, 2002.
22. Chowdhury K. Mastering Visual Studio 2017. – Packt Publishing Ltd, 2017.

- 23.Codd E. F. et al. Relational completeness of data base sublanguages. – IBM Corporation, 1972. – C. 65-98.
- 24.Codd E. F. Extending the database relational model to capture more meaning //ACM Transactions on Database Systems (TODS). – 1979. – T. 4. – №. 4. – C. 397-434.
- 25.Codd E. F. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate //<http://www.arborsoft.com/papers/coddTOC.html>. – 1993.
- 26.Codd E. F. The relational model for database management: version 2. – Addison-Wesley Longman Publishing Co., Inc., 1990.
- 27.Combemale B. et al. Essay on semantics definition in MDE. An instrumented approach for model verification. – 2009.
- 28.Cook P., Welsh J., Hayes I. J. Incremental semantic evaluation for interactive systems: inertia, pre-emption, and relations. – 2005.
- 29.Cooper K., Torczon L. Engineering a compiler. – Elsevier, 2011.
- 30.Crocker D., Overell P. Augmented BNF for syntax specifications: ABNF. – RFC 2234, November, 1997.
- 31.Crockford D. JavaScript: The Good Parts: The Good Parts. – " O'Reilly Media, Inc.", 2008.
- 32.Date C. J. An introduction to database systems. – Pearson Education India, 2004.
- 33.Date C. J. Database in depth: relational theory for practitioners. – " O'Reilly Media, Inc.", 2005.
- 34.De Bosschere K. An operator precedence parser for standard Prolog text //Software: Practice and Experience. – 1996. – T. 26. – №. 7. – C. 763-779.
- 35.Deber J. et al. How much faster is fast enough? user perception of latency & latency improvements in direct and indirect touch //Proceedings of the 33rd annual acm conference on human factors in computing systems. – 2015. – C. 1827-1836.

- 36.Diekmann L., Tratt L. Eco: A language composition editor //International Conference on Software Language Engineering. – Springer, Cham, 2014. – C. 82-101.
- 37.Earley J. An efficient context-free parsing algorithm //Communications of the ACM. – 1970. – T. 13. – №. 2. – C. 94-102.
- 38.Erdweg S. et al. The State of the Art in Language Workbenches //International Conference on Software Language Engineering. – Springer, Cham, 2013. – C. 197-217.
- 39.Ermilov I., Auer S., Stadler C. User-driven semantic mapping of tabular data //Proceedings of the 9th International Conference on Semantic Systems. – 2013. – C. 105-112.
- 40.European Computer Machinery Association et al. Standard ECMA-334: C# Language Specification. – 2005.
- 41.Evans D. E. Using specifications to check source code : дис. – Massachusetts Institute of Technology, 1994.
- 42.Eysholdt M., Behrens H. Xtext: implement your language faster than the quick and dirty way //Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. – 2010. – C. 307-309.
- 43.Fisher M., Dean M. Automapper: Relational database semantic translation using owl and swrl //Proceedings of the IASK International Conference E-Activity and Leading Technologies, Porto, Portugal. – 2007.
- 44.Flatt M. Creating languages in Racket //Communications of the ACM. – 2012. – T. 55. – №. 1. – C. 48-56.
- 45.Foster S. R., Griswold W. G., Lerner S. WitchDoctor: IDE support for real-time auto-completion of refactorings //2012 34th International Conference on Software Engineering (ICSE). – IEEE, 2012. – C. 222-232.
- 46.Geng T. et al. A practical GLR parser generator for software reverse engineering //Journal of Networks. – 2014. – T. 9. – №. 3. – C. 769.

47. Grigorev S., Kirilenko I. GLR-based abstract parsing //Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia. – 2013. – C. 1-9.
48. Hedin G. Incremental semantic analysis //Department of Computer Sciences. – 1992.
49. Hinkel G. et al. Using internal domain-specific languages to inherit tool support and modularity for model transformations //Software & Systems Modeling. – 2019. – T. 18. – №. 1. – C. 129-155.
50. Hinkel G. NMF: a multi-platform modeling framework //International Conference on Theory and Practice of Model Transformations. – Springer, Cham, 2018. – C. 184-194.
51. Holton J. A. The coding process and its challenges //The Sage handbook of grounded theory. – 2007. – T. 3. – C. 265-289.
52. Hyvönen E., Mäkelä E. Semantic autocompletion //Asian Semantic Web Conference. – Springer, Berlin, Heidelberg, 2006. – C. 739-751.
53. Jacobson G. Space-efficient static trees and graphs //30th annual symposium on foundations of computer science. – IEEE Computer Society, 1989. – C. 549-554.
54. Jézéquel J. M., Barais O., Fleurey F. Model driven language engineering with kermeta //International Summer School on Generative and Transformational Techniques in Software Engineering. – Springer, Berlin, Heidelberg, 2009. – C. 201-221.
55. Johanson A. N., Hasselbring W. Hierarchical combination of internal and external domain-specific languages for scientific computing //Proceedings of the 2014 European Conference on Software Architecture Workshops. – 2014. – C. 1-8.
56. Johnson G. F., Fischer C. N. Non-syntactic attribute flow in language based editors //Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 1982. – C. 185-195.

- 57.Jouault F., Bézivin J. KM3: a DSL for Metamodel Specification //International Conference on Formal Methods for Open Object-Based Distributed Systems. – Springer, Berlin, Heidelberg, 2006. – C. 171-185.
- 58.Kegler J. Marpa, A practical general parser: the recognizer //arXiv preprint arXiv:1910.08129. – 2019.
- 59.Keshishzadeh S., Mooij A. J., Hooman J. Industrial Experiences with a Formal DSL Semantics to Check Correctness of DSL Transformations //arXiv preprint arXiv:1511.08049. – 2015.
- 60.Kleene S. C. Recursive predicates and quantifiers //Transactions of the American Mathematical Society. – 1943. – T. 53. – №. 1. – C. 41-73.
- 61.Kleene S. C. Representation of events in nerve nets and finite automata //Automata studies. – 1956.
- 62.Knuth D. E. Semantics of context-free languages //Mathematical systems theory. – 1968. – T. 2. – №. 2. – C. 127-145.
- 63.Korenkov I., Loginov I., Dergachev A., Lazdin A. Declarative target architecture definition for data-driven development toolchain // 18th International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM-2018 - 2018, Vol. 18, No. 2.1, pp. 271-278
- 64.Kostaras I. et al. Apache NetBeans: New Features //Pro Apache NetBeans. – Apress, Berkeley, CA, 2020. – C. 63-72.
- 65.Kurtev I. et al. Model-based DSL frameworks //Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. – 2006. – C. 602-616.
- 66.Kurtev I. et al. Model-based DSL Frameworks //Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. – 2006. – C. 602-616.

67. Lavie A., Tomita M. 10. GLR*-AN EFFICIENT NOISE-SKIPPING PARSING ALGORITHM FOR CONTEXT-FREE GRAMMARS //Recent Advances in Parsing Technology. – 1996. – T. 1. – C. 183.
68. Louridas P. Static code analysis //Ieee Software. – 2006. – T. 23. – №. 4. – C. 58-61.
69. Mahmoud A., Niu N. Source code indexing for automated tracing //Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering. – 2011. – C. 3-9.
70. Matthews J., Findler R. B. Operational semantics for multi-language programs //ACM SIGPLAN Notices. – 2007. – T. 42. – №. 1. – C. 3-10.
71. Melton J., Simon A. R. Understanding the new SQL: a complete guide. – Morgan Kaufmann, 1993.
72. Mikowski M., Powell J. Single page web applications: JavaScript end-to-end. – Manning Publications Co., 2013.
73. Pech V., Shatalin A., Voelter M. JetBrains MPS as a tool for extending Java //Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. – 2013. – C. 165-168.
74. Pratt V. R. Top down operator precedence //Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. – 1973. – C. 41-51.
75. Radhakrishnan R. et al. Java runtime systems: Characterization and architectural implications //IEEE Transactions on computers. – 2001. – T. 50. – №. 2. – C. 131-146.
76. Rajlich V. T., Bennett K. H. A staged model for the software life cycle //Computer. – 2000. – T. 33. – №. 7. – C. 66-71.
77. Reps T. Optimal-time incremental semantic analysis for syntax-directed editors //Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 1982. – C. 169-176.

- 78.Reps T., Teitelbaum T., Demers A. Incremental context-dependent analysis for language-based editors //ACM Transactions on Programming Languages and Systems (TOPLAS). – 1983. – Т. 5. – №. 3. – С. 449-477.
- 79.Richter J. CLR via C#. Programming in .NET Framework 4.5 platform on in C# [CLR via C#. Programmirovanie na platforme Microsoft .NET Framework 4.5 na yazyke C#], Sankt-Petersburg: Piter. – 2013.
- 80.Sadilek D. A. Prototyping domain-specific language semantics //Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. – 2008. – С. 895-896.
- 81.Sadyrin D., Dergachev A., Loginov I., Korenkov I., Ilina A. Application of Graph Databases for Static Code Analysis of Web-Applications // CEUR Workshop Proceedings - 2020, Vol. 2590, pp. 1-9
- 82.Scholl M. H., Schek H. J. A relational object model //International Conference on Database Theory. – Springer, Berlin, Heidelberg, 1990. – С. 89-105.
- 83.Scott E., Johnstone A. GLL parsing //Electronic Notes in Theoretical Computer Science. – 2010. – Т. 253. – №. 7. – С. 177-189.
- 84.Slonneger K. Formal Syntax and Semantics of Programming Language. – Addison-Wesley Publishing Company, 1995.
- 85.Szabó T., Erdweg S., Voelter M. Inca: A DSL for the Definition of Incremental Program Analyses //Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. – 2016. – С. 320-331.
- 86.Thompson H. S. et al. XML schema part 1: Structures //W3C recommendation, May. – 2001. – Т. 21.
- 87.Tikhonova U. et al. Applying Model Transformation and Event-B for Specifying an Industrial DSL //MoDeVVa@ MoDELS. – 2013. – С. 41-50.
- 88.Tikhonova U. Reusable specification templates for defining dynamic semantics of DSLs //Software & Systems Modeling. – 2019. – Т. 18. – №. 1. – С. 691-720.

89. Tomita M. (ed.). Generalized LR parsing. – Springer Science & Business Media, 2012.
90. Tomita M. An efficient augmented-context-free parsing algorithm //Computational linguistics. – 1987. – T. 13. – C. 31-46.
91. Ulitin B., Babkin E., Babkina T. Ontology-based DSL development using graph transformations methods //Journal of Systems Integration. – 2018. – T. 9. – №. 2. – C. 37-51.
92. van der Storm T. The Rascal language workbench //CWI. Software Engineering [SEN]. – 2011. – T. 13. – C. 14.
93. Vasudevan N., Tratt L. Comparative study of DSL tools //Electronic Notes in Theoretical Computer Science. – 2011. – T. 264. – №. 5. – C. 103-121.
94. Vergu V., Néron P., Visser E. DynSem: A DSL for dynamic semantics specification //26th International Conference on Rewriting Techniques and Applications (RTA 2015). – Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
95. Wagner T. A., Graham S. L. Incremental analysis of real programming languages //ACM SIGPLAN Notices. – 1997. – T. 32. – №. 5. – C. 31-43.
96. Welsh J., Broom B., Kiong D. A design rationale for a language - based editor //Software: Practice and Experience. – 1991. – T. 21. – №. 9. – C. 923-948.
97. Wiegand J. et al. Eclipse: A platform for integrating development tools //IBM Systems Journal. – 2004. – T. 43. – №. 2. – C. 371-383.
98. Wilks Y. Machine translation: its scope and limits. – Springer Science & Business Media, 2008.
99. Wilks Y., Brewster C. Natural language processing as a foundation of the semantic web. – Now Publishers Inc, 2009.
100. Wirth N. Algorithms and data structures. – 1986.

Приложение А – расширенная грамматика спецификаций синтаксических моделей

Грамматика в предлагаемом формате:

```
[root(definition)]
[skip(commentsAndSpaces)]
pdsl {
    /* Portable Domain-Specific Language */
    /* keep it in sync with definition-grammar.xml */

    definition: ruleSet.body;
    ruleSet: attrs complexName '{' imports body '}' {
        body: item*;
        item: rule | ruleSet;
    };
    imports: ruleSetImport*;
    ruleSetImport: attrs alias complexName ';' {
        alias: (name '=')?;
    };
    rule: attrs complexName args ':' body ';' {
        args: ('<' (name (',' name))*? '>')?;
        body: {
            |simple: expr ('{' imports rule* '}')?;
            |cases: '{' imports literal* entry* rule* '}';
                literal: '|' name;
                entry: priority? '|' rule;
                priority: num|name;
            };
        };
    };
    commentsAndSpaces:
        "(([\\s]*)((/*(?>(?:(?>[^*]+)|\\*(?!/))*)\\*/[\\s]*)*";
    name: "[a-zA-Z_][a-zA-Z_0-9]*";
    number: "[0-9]+";
    hex: "0x[a-fA-F0-9]+";
    complexName: name ('.' name)*;
    attrs: collection* {
        usageArgList: ('(' expr (', ' expr)* ')')?;
        usage: complexName usageArgList;
        collection: '[' usage (', ' usage)* ']';
    };
}
```

```

expr: {
    |complex: {
        |sequence: item item+ {
            item: alternatives | repeat | simple;
        };
        |alternatives: item ('|' item)+ {
            item: repeat | simple;
        };
        |repeat: simple quantor {
            quantor: '*' | '+' | '?' | ('{' qnumbers '}');
            qnumbers: {
                |full: number ',' number;
                |max: ',' number;
                |min: number ',' ;
                |exact: number;
            };
        };
    };
    |simple: {
        |string: "\"[^\\\"\\\"]*(?:\\.\\[\\\"\\\"]*)*\"";
        |chars: "'[^']*'";
        |anyChar: '.';
        |charCode: hex;
        |group: '(' expr ')';
        |check: '&' simple;
        |checkNot: '!' simple;
        |usage: fieldSpec? flag complexName args {
            fieldSpec: name kind {
                kind: {
                    |collection: '+=';
                    |solitary: '=';
                };
            };
            flag: ('%' | '#')?;
            args: ('<' (expr (',' expr)*)? '>')?;
        };
    };
}

```

Та же грамматика, в плоском варианте, в форме XML-документа:

```

<?xml version="1.0" encoding="utf-8" ?>
<Grammar Name="pds1" StartRuleName="definition"
SkipRuleName="commentsAndSpaces" xmlns="PortableParserGrammar">
    <!-- keep it in sync with definition.pdsl -->
    <Rule Name="definition">
        <Call RuleName="ruleSet.body"/>
    </Rule>
    <Rule Name="ruleSet">
        <Seq>
            <Call RuleName="attrs"/>
            <Call RuleName="complexName"/>
            <Chars String="{" />
            <Call RuleName="imports"/>
            <Call RuleName="ruleSet.body"/>
            <Chars String="}" />
        </Seq>
    </Rule>
    <Rule Name="ruleSet.body">
        <Number Min="0">
            <Call RuleName="ruleSet.item"/>
        </Number>
    </Rule>
    <Rule Name="ruleSet.item">
        <Alts>
            <Call RuleName="rule"/>
            <Call RuleName="ruleSet"/>
        </Alts>
    </Rule>
    <Rule Name="imports">
        <Number Min="0">
            <Call RuleName="ruleSetImport"/>
        </Number>
    </Rule>
    <Rule Name="ruleSetImport">
        <Seq>
            <Call RuleName="attrs"/>
            <Call RuleName="ruleSetImport.alias"/>
            <Call RuleName="complexName"/>
            <Chars String=";" />
        </Seq>
    </Rule>

```

```

</Rule>
<Rule Name="ruleSetImport.alias">
  <Number Min="0" Max="1">
    <Seq>
      <Call RuleName="name" />
      <Chars String="=" />
    </Seq>
  </Number>
</Rule>
<Rule Name="rule">
  <Seq>
    <Call RuleName="attrs" />
    <Call RuleName="complexName" />
    <Call RuleName="rule.args" />
    <Chars String=":" />
    <Call RuleName="rule.body" />
  </Seq>
</Rule>
<Rule Name="rule.args">
  <Number Min="0" Max="1">
    <Seq>
      <Chars String="<" />
      <Number Min="0" Max="1">
        <Seq>
          <Call RuleName="name" />
          <Number Min="0">
            <Seq>
              <Chars String="." />
              <Call RuleName="name" />
            </Seq>
          </Number>
        </Seq>
      </Number>
      <Chars String=">" />
    </Seq>
  </Number>
</Rule>
<Rule Name="rule.body">
  <Alts>
    <Call RuleName="rule.body.simple" />
    <Call RuleName="rule.body.cases" />
  </Alts>

```

```

</Rule>
<Rule Name="rule.body.simple">
  <Seq>
    <Call RuleName="expr" />
    <Number Min="0" Max="1">
      <Seq>
        <Chars String="{" />
        <Call RuleName="imports" />
        <Number Min="0">
          <Call RuleName="rule" />
        </Number>
        <Chars String="}" />
      </Seq>
    </Number>
  </Seq>
</Rule>
<Rule Name="rule.body.cases">
  <Seq>
    <Chars String="{" />
    <Call RuleName="imports" />
    <Number Min="0">
      <Call RuleName="rule.body.cases.literal" />
    </Number>
    <Number Min="0">
      <Call RuleName="rule.body.cases.entry" />
    </Number>
    <Number Min="0">
      <Call RuleName="rule" />
    </Number>
    <Chars String="}" />
  </Seq>
</Rule>
<Rule Name="rule.body.cases.literal">
  <Seq>
    <Chars String="|" />
    <Call RuleName="name" />
  </Seq>
</Rule>
<Rule Name="rule.body.cases.entry">
  <Seq>
    <Number Min="0" Max="1">
      <Call RuleName="rule.body.cases.priority" />
  
```

```

    </Number>
    <Chars String=" | " />
    <Call RuleName="rule" />
</Seq>
</Rule>
<Rule Name="rule.body.cases.priority">
    <Alts>
        <Call RuleName="name" />
        <Call RuleName="number" />
    </Alts>
</Rule>
<Rule Name="commentsAndSpaces">
    <Regex
        Pattern="([\s]*)(/\*(?>(?:(>[^*]+)|\*(?!/))*)\*/[\s]*)*" />
</Rule>
<Rule Name="name">
    <Regex Pattern="[a-zA-Z_][a-zA-Z0-9_]*" />
</Rule>
<Rule Name="number">
    <Regex Pattern="[0-9]+" />
</Rule>
<Rule Name="hex">
    <Regex Pattern="0x[a-fA-F0-9]+" />
</Rule>
<Rule Name="complexName">
    <Seq>
        <Call RuleName="name"/>
        <Number Min="0">
            <Seq>
                <Chars String=". />
                <Call RuleName="name"/>
            </Seq>
        </Number>
    </Seq>
</Rule>
<Rule Name="attrs">
    <Number Min="0">
        <Call RuleName="attrs.collection" />
    </Number>
</Rule>
<Rule Name="attrs.usageArgList">
    <Number Min="0" Max="1">

```

```

<Seq>
  <Chars String="(" />
  <Call RuleName="expr" />
  <Number Min="0">
    <Seq>
      <Chars String="," />
      <Call RuleName="expr" />
    </Seq>
  </Number>
  <Chars String=")" />
</Seq>
</Number>
</Rule>
<Rule Name="attrs.usage">
  <Seq>
    <Call RuleName="complexName" />
    <Call RuleName="attrs.usageArgList" />
  </Seq>
</Rule>
<Rule Name="attrs.collection">
  <Seq>
    <Chars String="[" />
    <Call RuleName="attrs.usage" />
    <Number Min="0">
      <Seq>
        <Chars String="," />
        <Call RuleName="attrs.usage" />
      </Seq>
    </Number>
    <Chars String="]" />
  </Seq>
</Rule>
<Rule Name="expr">
  <Alts>
    <Call RuleName="expr.complex" />
    <Call RuleName="expr.simple" />
  </Alts>
</Rule>
<Rule Name="expr.complex">
  <Alts>
    <Call RuleName="expr.complex.alternatives" />
    <Call RuleName="expr.complex.sequence" />
  </Alts>
</Rule>

```

```
<Call RuleName="expr.complex.repeat" />
</Alts>
</Rule>
<Rule Name="expr.complex.sequence">
<Seq>
<Call RuleName="expr.complex.sequence.item" />
<Number Min="1">
<Call RuleName="expr.complex.sequence.item" />
</Number>
</Seq>
</Rule>
<Rule Name="expr.complex.sequence.item">
<Alts>
<Call RuleName="expr.complex.alternatives" />
<Call RuleName="expr.complex.repeat" />
<Call RuleName="expr.simple" />
</Alts>
</Rule>
<Rule Name="expr.complex.alternatives">
<Seq>
<Call RuleName="expr.complex.alternatives.item" />
<Number Min="1">
<Seq>
<Chars String=" | " />
<Call RuleName="expr.complex.alternatives.item" />
</Seq>
</Number>
</Seq>
</Rule>
<Rule Name="expr.complex.alternatives.item">
<Alts>
<Call RuleName="expr.complex.alternatives" />
<Call RuleName="expr.complex.repeat" />
<Call RuleName="expr.simple" />
</Alts>
</Rule>
<Rule Name="expr.complex.repeat">
<Seq>
<Call RuleName="expr.simple"/>
<Call RuleName="expr.complex.repeat.quantor" />
</Seq>
</Rule>
```

```

<Rule Name="expr.complex.repeat.quantor">
  <Alts>
    <Chars String="*" />
    <Chars String="+" />
    <Chars String="?" />
    <Seq>
      <Chars String="{" />
      <Call RuleName="expr.complex.repeat.qnumbers"/>
      <Chars String="}" />
    </Seq>
  </Alts>
</Rule>
<Rule Name="expr.complex.repeat.qnumbers">
  <Alts>
    <Call RuleName="expr.complex.repeat.qnumbers.full"/>
    <Call RuleName="expr.complex.repeat.qnumbers.max"/>
    <Call RuleName="expr.complex.repeat.qnumbers.min"/>
    <Call RuleName="expr.complex.repeat.qnumbers.exact"/>
  </Alts>
</Rule>
<Rule Name="expr.complex.repeat.qnumbers.full">
  <Seq>
    <Call RuleName="number" />
    <Chars String="," />
    <Call RuleName="number" />
  </Seq>
</Rule>
<Rule Name="expr.complex.repeat.qnumbers.max">
  <Seq>
    <Chars String="," />
    <Call RuleName="number" />
  </Seq>
</Rule>
<Rule Name="expr.complex.repeat.qnumbers.min">
  <Seq>
    <Call RuleName="number" />
    <Chars String="," />
  </Seq>
</Rule>
<Rule Name="expr.complex.repeat.qnumbers.exact">
  <Seq>
    <Call RuleName="number" />
  </Seq>

```

```

    </Seq>
</Rule>
<Rule Name="expr.simple">
    <Alts>
        <Call RuleName="expr.simple.string" />
        <Call RuleName="expr.simple.chars" />
        <Call RuleName="expr.simple.anyChar" />
        <Call RuleName="expr.simple.charCodeAt" />
        <Call RuleName="expr.simple.group" />
        <Call RuleName="expr.simple.check" />
        <Call RuleName="expr.simple.checkNot" />
        <Call RuleName="expr.simple.usage" />
    </Alts>
</Rule>
<Rule Name="expr.simple.string">
    <Regex
        Pattern="\"[^\"\\]*(:\\.\\[^\"\\])*\"/>
</Rule>
<Rule Name="expr.simple.chars">
    <Regex Pattern="'^[^']*'" />
</Rule>
<Rule Name="expr.simple.anyChar">
    <Chars String="." />
</Rule>
<Rule Name="expr.simple.charCodeAt">
    <Call RuleName="hex"/>
</Rule>
<Rule Name="expr.simple.group">
    <Seq>
        <Chars String="(" />
        <Call RuleName="expr" />
        <Chars String=")" />
    </Seq>
</Rule>
<Rule Name="expr.simple.check">
    <Seq>
        <Chars String="&" />
        <Call RuleName="expr.simple" />
    </Seq>
</Rule>
<Rule Name="expr.simple.checkNot">
    <Seq>

```

```

<Chars String="!" />
<Call RuleName="expr.simple" />
</Seq>
</Rule>
<Rule Name="expr.simple.usage">
<Seq>
<Number Min="0" Max="1">
<Call RuleName="expr.simple.usage.fieldSpec" />
</Number>
<Call RuleName="expr.simple.usage.flag" />
<Call RuleName="complexName" />
<Call RuleName="expr.simple.usage.args" />
</Seq>
</Rule>
<Rule Name="expr.simple.usage.fieldSpec">
<Seq>
<Call RuleName="name" />
<Call RuleName="expr.simple.usage.fieldSpec.kind" />
</Seq>
</Rule>
<Rule Name="expr.simple.usage.fieldSpec.kind">
<Alts>
<Call RuleName="expr.simple.usage.fieldSpec.kind.collection" />
<Call RuleName="expr.simple.usage.fieldSpec.kind.solitary" />
</Alts>
</Rule>
<Rule Name="expr.simple.usage.fieldSpec.kind.collection">
<Chars String="+=" />
</Rule>
<Rule Name="expr.simple.usage.fieldSpec.kind.solitary">
<Chars String="=" />
</Rule>
<Rule Name="expr.simple.usage.flag">
<Number Min="0" Max="1">
<Alts>
<Chars String "%" />
<Chars String="#" />
</Alts>
</Number>
</Rule>
<Rule Name="expr.simple.usage.args">
<Number Min="0" Max="1">

```

```
<Seq>
  <Chars String="&lt;" />
  <Number Min="0" Max="1">
    <Seq>
      <Call RuleName="expr" />
      <Number Min="0">
        <Seq>
          <Chars String="," />
          <Call RuleName="expr" />
        </Seq>
      </Number>
    </Seq>
  </Number>
  <Chars String="&gt;" />
</Seq>
</Number>
</Rule>
</Grammar>
```

Приложение Б – грамматика спецификаций семантических моделей

В предложенном формате:

```
[root(scope)]
[skip(commentsAndSpaces)]
pddl {
    /* Portable Domain Definition Language */
    common;
    scope: name '{' (types += value)* '}';
    value: name ':' typeRef|((typeParent = complexName)? typeDef);
    typeName: complexName|builtin {
        builtin: {
            |bool
            |char
            |byte
            |sbyte
            |short
            |ushort
            |int
            |uint
            |long
            |ulong
            |string
            |integer
            |decimal
        };
    };
    typeRef: typeName array? {
        array: '[' ']';
    };
    typeDef: '{' members cases '}';
    members: (member ';')*;
    member: {
        |explicit: value;
        |implicit: typeRef;
    };
    cases: ('|' case)*;
    case: name typeDef?;
}
}
```

Та же грамматика в ЕБНФ-подобном виде:

```

grammar Pddl
{
    options
    {
        Axiom = "scope";
        Separator = "SEPARATOR";
    }
    terminals
    {
        // A.1.1 Line terminators
        NEW_LINE      -> U+000D /* CR */
                        | U+000A /* LF */
                        | U+000D U+000A /* CR LF */
                        | U+0085 // Next line character
                        | U+2028 // Line separator character
                        | U+2029 ; //Paragraph separator
                        //character (U+2029)

        WHITE_SPACE   -> uc{Zs} | U+0009 | U+000B | U+000C ;
        COMMENT_LINE  -> '//' (.* - (.* NEW_LINE .*)) ;
        COMMENT_BLOCK -> '/*' (.* - (.* '*/' .*)) '*/' ;
        SEPARATOR     -> (NEW_LINE | WHITE_SPACE |
                            COMMENT_LINE | COMMENT_BLOCK)+;

        ESCAPEES-> '\\\\'           // Backslash
                        | '\\0'          // Unicode character 0
                        | '\\a'          // Alert (character 7)
                        | '\\b'          // Backspace (character 8)
                        | '\\f'          // Form feed (character 12)
                        | '\\n'          // New line (character 10)
                        | '\\r'          // Carriage return (character 13)
                        | '\\t'          // Horizontal tab (character 9)
                        | '\\v'          // Vertical quote (character 11)
                        | '\\u' [0-9a-fA-F]{4} // Unicode code point
                        | '\\u' [0-9a-fA-F]{8} ; // Unicode code point

        name -> [a-zA-Z_] [a-zA-Z0-9_]* ;

        typeBuiltin -> 'int'|'long'|'string'|'bool'|'real';
    }
}

```

```

rules
{
    complexName -> name ('.' name)*;

    scope      -> name '{' value* '}';
    value       -> name ':' (typeRef|(complexName? typeDef));
    typeName   -> complexName|typeBuiltin;

    typeRef     -> typeName typeRefArraySpec?;
    typeRefArraySpec -> '[' ']';

    typeDef     -> '{' typeDefMembers typeDefCases '}';
    // typeDefMembers-> (typeDefMember ';')*;
    // typeDefMember->
    //   typeDefMemberExplicit|typeDefMemberImplicit;
    typeDefMembers->
        ((typeDefMemberExplicit|typeDefMemberImplicit) ';' )*;
    typeDefMemberExplicit-> value;
    typeDefMemberImplicit-> typeRef;
    typeDefCases-> ('|' typeDefCase)*;
    typeDefCase-> name typeDef?;

}
}

```

Приложение В – акты о внедрении

**федеральное государственное автономное образовательное учреждение
высшего образования**

«Национальный исследовательский университет ИТМО»



АКТ

**о внедрении результатов диссертационной работы Коренькова Юрия Дмитриевича
«Методы и средства анализа исходных текстов программ и программных систем на
основе семантических моделей», представленного на соискание ученой степени
кандидата технических наук по специальности 05.13.11 Математическое и
программное обеспечение вычислительных машин, комплексов и компьютерных
сетей.**

Комиссия в составе

председателя: к.т.н., декана факультета программной инженерии и компьютерной
техники
Кустарева Павла Валерьевича,

и членов: к.т.н., доцента факультета программной инженерии и компьютерной
техники
Быковского Сергея Вячеславовича,

к.т.н., доцента факультета программной инженерии и компьютерной
техники
Муромцева Дмитрия Ильича

составила настоящий акт о том, что научно-практические результаты диссертационной
работы Коренькова Ю.Д. внедрены в учебный процесс на факультете программной
инженерии и компьютерной техники Университета ИТМО для студентов, обучающихся по
образовательной программе «Системное и прикладное программное обеспечение»
направления 09.03.04 «Программная инженерия» и «Технологии промышленного
программирования» направления 09.04.04 «Программная инженерия».

Внедрение заключается в следующем:

1. Теоретические результаты диссертационной работы, изложенные в главе 2 диссертации, используются в лекционном курсе дисциплины «Языково-ориентированное программирование» в части разработки встраиваемых предметно-ориентированных языков.
2. Научно-практические результаты диссертационной работы, изложенные в главах 3 и 4 диссертации, используются студентами при выполнении домашних заданий и лабораторных работ по дисциплине «Веб-программирование» и «Информационные системы и базы данных». Применение возможности анализа мультиязыковых проектов позволяет выявлять ошибки в коде программ, реализованных совместно на языках php, JavaScript и SQL.

Председатель комиссии

 Кустарев П.В.

Члены комиссии:

 Быковский С.В.

 Муромцев Д.И.



Общество с Ограниченной Ответственностью "Тюн-ит"
 197022, г. Санкт-Петербург , пр. Каменноостровский, дом № 27, литер Б, пом. 3-Н
 ИНН 7804088826 КПП 781301001 ОГРН 1037808021261
 Р/счет 40702810918000001813 Ф. ОПЕРУ БАНКА ВТБ (ПАО) В САНКТ-ПЕТЕРБУРГЕ
 БИК 044030704 к/счет 30101810200000000704
 тел.: +7(812) 325-44-40, факс: +7(812)325-44-40, e-mail: info@tune-it.ru

Дата : 08.10.2020
 Исх.номер : 20201008-01

АКТ О ВНЕДРЕНИИ
 научных и практических результатов
 кандидатской диссертации Коренькова Юрия Дмитриевича на тему
 «Методы и средства анализа исходных текстов программ и программных систем на
 основе семантических моделей»

Настоящий акт подтверждает использование в ООО «Тюн-ит» следующих результатов диссертационной работы Коренькова Ю.Д.

1. Метод предметно-ориентированного анализа исходных текстов программ, позволяющий осуществлять анализ кода, написанного с использованием нескольких языков программирования.
2. Реализация языка описания предметно-ориентированных семантических моделей программ, позволяющая выполнять анализ кода с учетом специфических требований в рамках отдельных программных проектов.
3. Реализация текстового редактора, позволяющая осуществлять кодирование программ с применением разработанного языка описания семантических моделей с целью задания ограничений для проверки корректности семантической модели программы.

Предложенный метод и разработанные программные средства применяются в процессе аудита исходного кода программ с целью выявления потенциально уязвимых программных компонентов, функциональности библиотек и их некорректного использования, приводящего к возникновению уязвимостей. Использование языка описания предметно-ориентированных семантических моделей осуществляется совместно с языками, на которых осуществляется разработка программных проектов. Применение разработанного редактора позволило повысить выявляемость потенциальных уязвимостей на уровне исходного кода на 10-12%.

Генеральный директор

С.В.Клименков

Главный инженер

Д.Б.Афанасьев



Приложение Г – тексты публикаций

Таблица Г.1 – перечень публикаций с указанием номеров страниц

Публикация	Номера страниц в документе
Sadyrin D., Dergachev A., Loginov I., Korenkov I., Ilina A. Application of Graph Databases for Static Code Analysis of Web-Applications // CEUR Workshop Proceedings - 2020, Vol. 2590, pp. 1-9	188-196
Korenkov I., Loginov I., Dergachev A., Lazdin A. Declarative target architecture definition for data-driven development toolchain // 18th International Multidisciplinary Scientific GeoConference Surveying Geology and Mining Ecology Management, SGEM-2018 - 2018, Vol. 18, No. 2.1, pp. 271-278	197-204
Дергачев А.М., Садырин Д.С., Ильина А.Г., Логинов И.П., Кореньков Ю.Д. Методы и средства обнаружения уязвимостей аллокаторов динамической памяти библиотеки glibc // Научно-технический вестник Поволжья - 2020. - № 1. - С. 79-83	205-209
Кореньков Ю.Д., Логинов И.П. Исследование и разработка языкового инструментария на основе PEG-грамматики // Известия высших учебных заведений. Приборостроение - 2015. - Т. 58. - № 11. - С. 934-938	210-214
Кореньков Ю.Д. Метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей // Научно-технический вестник Поволжья - 2020 - № 7 - С. 32-37	215-220

Application of Graph Databases for Static Code Analysis of Web-Applications

Daniil Sadyrin [0000-0001-5002-3639], Andrey Dergachev [0000-0002-1754-7120], Ivan Loginov [0000-0002-6254-6098], Iurii Korenkov [0000-0002-8948-2776], and Aglaya Ilina [0000-0003-1866-7914]

ITMO University, Kronverkskiy prospekt, 49, St. Petersburg, 197101, Russia
 dssadyrin@itmo.ru, dam600@gmail.com, ivan.p.loginov@gmail.com,
 ged.yuko@gmail.com, agilina@itmo.ru

Abstract. Graph databases offer a very flexible data model. We present the approach of static code analysis using graph databases. The main stage of the analysis algorithm is the construction of ASG (Abstract Source Graph), which represents relationships between AST (Abstract Syntax Tree) nodes. The ASG is saved to a graph database (like Neo4j) and queries to the database are made to get code properties for analysis. The approach is applied to detect and exploit Object Injection vulnerability in PHP web-applications. This vulnerability occurs when unsanitized user data enters PHP unserialize function. Successful exploitation of this vulnerability means building of “object chain”: a nested object, in the process of deserializing of it, a sequence of methods is being called leading to dangerous function call. In time of deserializing, some “magic” PHP methods (`__wakeup` or `__destruct`) are called on the object. To create the “object chain”, it’s necessary to analyze methods of classes declared in web-application, and find sequence of methods called from “magic” methods. The main idea of author’s approach is to save relationships between methods and functions in graph database and use queries to the database on Cypher language to find appropriate method calls. Also, some unobvious ways of calling other PHP “magic” methods, which help to find more appropriate “object chains” are considered. The approach was successfully tested on the vulnerability CVE-2014-1860 discovered in Contao CMS.

Keywords: static analysis · graph database · Cypher · PHP · Object injection

1 Introduction

Graph databases are used in many areas like bioinformatics [1], social networks [2], chemistry [3], and static code analysis [4]. A graph database (over a countably infinite set of labels Σ) is a pair $G = (V, E)$ where V is a finite set of nodes,

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

$E \subseteq V \times \Sigma \times V$ is a finite set of edges. There are two basic ways to explore and graphically depict connected data: Resource Description Framework (RDF) triple stores and labeled property graphs.

The abstract RDF syntax is a set of triplets called an RDF graph. An RDF triplet contains three components: a subject that is an URI reference or an empty node; predicate, which is an URI reference; an object that is an URI reference, literal, or empty node. RDF triplet written in order: subject, predicate and object. A predicate is also known as a triplet property. An RDF graph is a set of RDF triplets. A set of RDF graph nodes is a set of subjects and objects of triplets of a graph.

The property graphs are graphs in which attributes (properties) are assigned to edges and/or vertices of the graph. A database in a graph model is a graph whose vertices and edges are typed. A vertex or edge type is a collection of attributes (properties) attributed to a vertex or edge.

2 Graph search queries in terms of formal languages

One of the major problems, related with graph databases, is to find specific paths in it. A path in graph G is a sequence $p = (v_0, a_1, v_1) \dots (v_{n-1}, a_n, v_n)$ of edges of G . Constraints on the paths can be expressed in several ways: conjunctive queries, shortest path queries, in terms of formal languages.

Constraints in terms of regular languages are used to search patterns in graph. The regular expressions for an alphabet Σ are defined by the following form: $E ::= \emptyset \mid \varepsilon \mid a \mid (E \circ E) \mid (E + E) \mid E^*$, where $a \in \Sigma$. If E is a regular expression then $L(E)$ is a regular language. A regular path query (RPQ) is an expression of the form $x \xrightarrow{r} y$ where x and y are variables and r is regular expression over Σ . Let r be a regular expression and G be a graph. A path $p = (v_0, a_1, v_1) \dots (v_{n-1}, a_n, v_n)$ in G matches r , if $a_1 a_2 \dots a_n \in L(r)$.

Context-free languages also can be used as constraints. A context-free grammar G can be defined as 4-tuple: $G = (V, T, P, S)$ where V is a finite set of nonterminals containing S , T is finite set of terminals, P is a set of production rules in the form of $\alpha \rightarrow \beta$ where $\alpha \in V$ and $\beta \in (V \cup T)^*$, and S is start symbol. An answer to a context-free path query (CFPQ) is usually a set of triples (A, m, n) such that there is a path from the node m to the node n , whose labeling is derived from a non-terminal $A \in V$ of the given context-free grammar.

For example, let's consider the following context-free grammar: $S \rightarrow aSa$ and $S \rightarrow bSb$. This grammar generates the language of even-length palindromes.

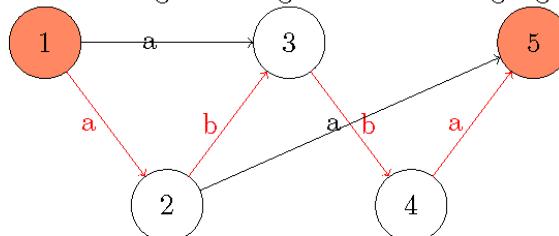


Fig. 1: Graph with path accepted by language $L = \{ww^R, w \in \{a,b\}^*\}$

Context-free path queries are more expressive, than regular path queries, but such type of queries is not real-time and requires large amounts of memory [12]. Another way to increase the expressiveness of regular path queries is conjunctive regular path queries. A conjunctive regular path query (CRPQ) is an expression of the form $\exists \bar{z} ((x_1 \xrightarrow{a_1} y_1) \wedge \dots \wedge (x_n \xrightarrow{a_n} y_n))$ where \bar{z} is a tuple of variables from $\{x_1, \dots, x_n, y_1, \dots, y_n\}$ and r_i is an RPQ over Σ for $i \in [n]$. A conjunctive query over graph is an expression of the form: $\exists \bar{z} ((x_1 \xrightarrow{a_1} y_1) \wedge \dots \wedge (x_n \xrightarrow{a_n} y_n))$ where \bar{z} is a tuple of variables from $\{x_1, \dots, x_n, y_1, \dots, y_n\}$ and $\{a_1, \dots, a_n\} \in \Sigma$. Conjunctive queries (and even first-order queries) on graphs are limited, they can only express "local" properties.

3 Path querying algorithms

Most existing graph querying languages, including SPARQL [5], Gremlin [6], Cypher support only regular languages as constraints. There are several approaches for evaluating RPQ: approach based on mapping to finite automaton [7] and on searching for rare labels and starting breadth-first search [8]. Algorithms for evaluating conjunctive regular path queries are studied in [9], [10]. cfSPARQL [11] is the single known graph query language to support context-free path constraints. The most of context-free path query evaluation algorithms are based on extending the known context-free parse techniques to the graph input. GSSLR algorithm [13] is based on Tomita recognizer, subgraph queries algorithm [14] uses Early parser, context-free path querying with structural representation of result [15] is based on generalized top-down parsing algorithm (GLL). Also, algorithm, introduced in [17], constructs annotated grammar in order to evaluate context-free path query. Algorithm in [16] is designed to use fast boolean matrix multiplication and GPU.

4 Graph databases in static code analysis

One of the important usages of graph data models is a static code analysis. Static analysis is a proven approach for detecting mistakes in the source code early in the development cycle. Since static analysis does not compile or run the code, it can be applied at an early state of development. This section investigates the usage of graph databases in static code analysis tools.

Graph-based analysis of JavaScript source code repositories [18] detects deadcode, potential division by zero, and other mistakes using Neo4j graph databases and openCypher for evaluating regular path queries.

GREENSPECTOR [21] uses Neo4j and Cypher query language for finding coding mistakes in a special graph data model called "call graph" via pattern-matching.

Wiggle [22] is a prototype graph-model code-query system. It performs such kinds of analysis like exploring type hierarchy, override hierarchy, type attribution, method call graph and data flow analysis.

Class-Graph [23] uses Neo4j and Cypher query language to collect structural insights about Java projects and to store, compute and visualize a variety of software metrics and other types of software analytics (method call hierarchies, transitive closure, critical path analysis, volatility and code quality).

Paper [24] presents an approach to detect behavioral design patterns from source code using static analysis techniques. This approach used Neo4j and uses graph query language Gremlin for doing graph matching to perform structural analysis, behavioral analysis, semantic analysis, Program Dependence Analysis, Control Dependence Analysis and Data Dependence Analysis.

jQAssistant [25] is a static code analysis tool using the graph database Neo4j and Cypher query language. It is used for detection of constraint violations, generating reports about user defined concepts and metrics, detecting common problems like cyclic dependencies or tests without assertions in Java projects. jQAssistant allows definition of rules and automated verification during a build process. Rules can be expressed as Cypher queries.

Joern [26] analyzes a code base using a robust parser for C/C++ and represents the entire code base by one large property graph stored in a Neo4j graph database. This allows code to be mined using complex queries formulated in the graph traversal languages Gremlin and Cypher. Exploring program structure, call graph, data flows, methods, types and other can be performed by Joern tool.

NAVEX [27] combines dynamic analysis that is guided by static analysis techniques in order to automatically identify vulnerabilities and build working exploits. It uses extension of Joern for PHP language to find vulnerabilities via searching the enhanced code property graph using Gremlin queries and Neo4j graph database.

5 Static code analysis problem

To extract information for static analysis, it is necessary to present the program code in the form of AST (Abstract Syntax Tree). Next, the syntax tree is translated into a graph representation - ASG (Abstract Semantic Graph), for this it is necessary to complete the information from AST with semantic information. The resulting graph structure that accommodates the information, would be a direct representation of packages, classes, interfaces, types, methods, fields and containing relationships like dependencies, containment, calls, coverage, etc. Queries are made to the constructed graph to retrieve the necessary code properties, for example, obtain functionDefinition relationships to get a program call stack.

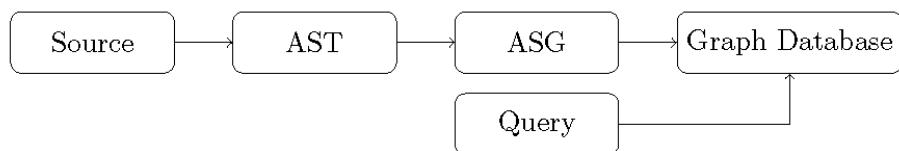


Fig. 2: Flowchart of static analysis engine

6 Proposed Approach

It is proposed to apply this approach for searching and exploiting of the Object Injection vulnerability in PHP web applications. Object Injection attack is part of the OWASP [19] vulnerability classification. The vulnerability arises when unsanitized user data enters the PHP unserialize function. The result of exploiting the vulnerability is to build a string with a chain of serialized PHP objects. In order to successfully exploit a PHP Object Injection vulnerability two conditions must be met:

- The web-application must have a class which implements a PHP magic method (such as __wakeup or __destruct) that can be used to carry out malicious attack
- All of the classes used during the attack must be declared when the unserialize() function is being called

To exploit this vulnerability, we need to do static analysis of declared classes in web-application code. During static analysis, we build a class hierarchy based on the inheritance of each class. All defined methods, properties and class constants are transformed to data symbols and stored in the analysis environment. When analyzing the code of declared methods, it is necessary to take into account these object-oriented features in PHP [20]:

- Object-sensitive Methods: __set_state(), __sleep(), __invoke(), __clone(), __toString(), __construct()
- Field-sensitive Methods: __get, __set, __isset, __unset
- Invocation-sensitive Methods: __call, __callStatic
- Calls using keywords: parent::, self::, static::

Let's consider an example of the class with __wakeup method, which has "new" operator in its code. __construct method of Vuln class is called and arbitrary file is unlinked.

```
class Vuln {
    public function __construct($file) {
        unlink($file);
    }
}
class test {
    public function __wakeup() {
        $this->a = new Vuln($this->test);
    }
}
unserialize('O:4:"test":1:{s:4:"test";s:13:"/tmp/test.php";}');
```

In the following example non-existent method call in `__destruct` leads to `__call` method being executed:

```
class method_test {
    public function __call($name, $arguments) {
        echo "call '$name' ". implode(' ', $arguments). PHP_EOL;
        unlink($this->file);
    }
    public function __destruct() {
        $this->notexisting(1,2,3);
    }
}
unserialize('O:11:"method_test":1:{s:4:"file";s:13:"/tmp/test.php"}');
```

For each method, it is performed a check for possible other method calls and "dangerous" functions. Using this information we can create object injection chains. It is necessary to take into account some unobvious ways to call PHP "magic" methods, when analyzing web-application source codes. It helps to find additional relationships between method calls and build proper "object chain". No one of the previously reviewed open-source projects solves the task of searching PHP object injection vulnerability.

7 Implementation

Constructing AST from source codes is done using nikic's PHP-Parser utility [28]. Each AST node is an object of class representing this node. Obtaining relationships is done by calling "traverse" method of an object of NodeTraverser class, declared in PHP-Parser.

```
$nodeTraverser = new PhpParser\NodeTraverser;
$nodeTraverser->addVisitor(new ChangeMethodNameNodeVisitor);
$traversedNodes = $nodeTraverser->traverse($nodes);
```

All constructed nodes are bypassed, and there is done a check whether the node is the object of some class:

- Class_ - represents AST of whole declared class.
- ClassMethod - represents AST of class method code.
- MethodCall - represents AST corresponding to method call inside other methods.
- FuncCall - represents AST corresponding to function call inside other methods.

Information is extracted from nodes, saved into CSV files and imported into Neo4j database. Methods and functions are represented using nodes labeled "Method" and "Function" and linked by relationship named "CALLS". Nodes labeled "Method" have properties: name - method name, class_name - name

of the class where method is declared. "Function" nodes have the following properties: name - function name, vuln (indicates that it is "dangerous" function, True or False). "CALLS" relationship stores class field that calls the method in its property. Relationships between method and function calls are created by the following queries written on Cypher language:

```
MATCH (n1:Method),(n2:Method) WHERE n1.class_name=line[0]
AND n1.name=line[1] AND n2.name=line[3]
MERGE (n1)-[r:CALLS {property:line[2]}]->(n2)'''
```

```
MATCH (n1:Method),(n2:Function) WHERE n1.class_name=line[0]
AND n1.name=line[1] AND n2.name=line[2]
MERGE (n1)-[r:CALLS {property:''}]->(n2)'''
```

To obtain a sequence of method calls, we execute a query to the database:

```
MATCH p = (a: Method) - [r: CALLS * 0..10] -> (b: Function {vuln: True})
WHERE a.name IN ['__wakeup', '__destruct'] RETURN p
```

This query returns all paths in graph database from nodes representing method with the "magic" name (`__destruct` / `__wakeup`) to nodes representing dangerous function (marked with property "vuln" equal to "True").

For demonstrating the approach, we took Contao CMS with vulnerability CVE-2014-1860 [29] and obtained a sequence of method calls leading to PHP unlink function being called with an arbitrary file name as an argument.

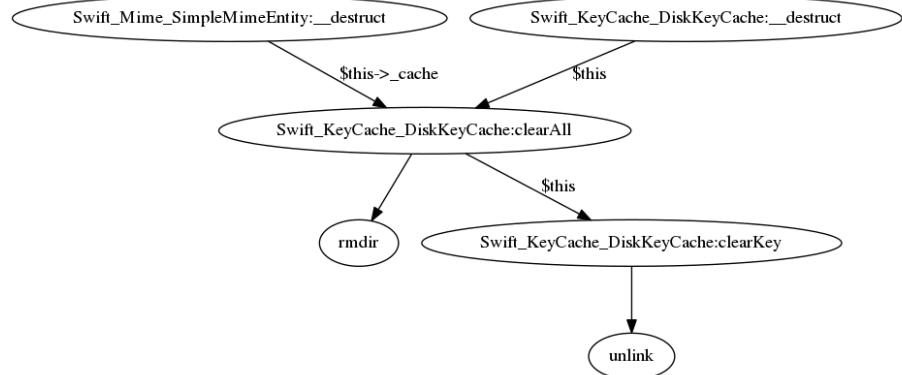


Fig. 3: CVE-2014-1860 methods call graph

8 Conclusion

Summing up, we propose method for static code analysis of scripts written in PHP programming language. We create graph database that stores relationships

between code properties. Using this information creation of object injection chains is done. Further work may consist in applying an approach to searching deserialization vulnerabilities in application code with frameworks in Java (Weblogic, Tomcat, Spring) or .NET. (Nancy, Breeze), and reducing time complexity of graph database queries using context-free path queries.

References

1. Fiannaca A. et al. BioGraphDB: a new GraphDB collecting heterogeneous data for bioinformatics analysis //Proceedings of BIOTECHNO. – 2016.
2. Cattuto C. et al. Time-varying social networks in a graph database: a Neo4j use case //First international workshop on graph data management experiences and systems. – ACM, 2013. – C. 11.
3. Hall R. J., Murray C. W., Verdonk M. L. The Fragment Network: A Chemistry Recommendation Engine Built Using a Graph Database //Journal of medicinal chemistry. – 2017. – T. 60.
4. Yamaguchi F. et al. Modeling and discovering vulnerabilities with code property graphs //2014 IEEE Symposium on Security and Privacy. – IEEE, 2014. – C. 590-604.
5. Prud E. et al. SPARQL query language for RDF.(2006). – 2006.
6. Rodriguez M. A. The Gremlin graph traversal machine and language (invited talk) //Proceedings of the 15th Symposium on Database Programming Languages. - ACM, 2015. - C. 1-10.
7. Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. SIAM J. Comput. 24, 6 (December 1995), 1235-1258. DOI=<http://dx.doi.org/10.1137/S009753979122370X>
8. Koschmieder, Andre and Leser, Ulf. (2012). Regular Path Queries on Large Graphs. 7338. 10.1007/978-3-642-31235-9_12.
9. Pablo Barceló Baeza. 2013. Querying graph databases. In Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems (PODS '13). ACM, New York, NY, USA, 175-188. DOI: <https://doi.org/10.1145/2463664.2465216>
10. Bienvenu, Meghyn and Ortiz, Magdalena and Šimkus, Mantas. (2013). Conjunctive Regular Path Queries in Lightweight Description Logics. IJCAI International Joint Conference on Artificial Intelligence. 761-767.
11. Zhang X. et al. Context-free path queries on RDF graphs //International Semantic Web Conference. - Springer, Cham, 2016. - C. 632-648.
12. Kuijpers J. et al. An experimental study of context-free path query evaluation methods //Proceedings of the 31st International Conference on Scientific and Statistical Database Management. – ACM, 2019. – C. 121-132.
13. Medeiros, Ciro and Musicante, Martin and Costa, Umberto. (2019). LL-based query answering over RDF databases. Journal of Computer Languages. 51. 75-87. 10.1016/j.jola.2019.02.002.
14. Sevon, Petteri and Eronen, Lauri. (2008). Subgraph Queries by Context-free Grammars. Journal of Integrative Bioinformatics. 5. 10.1515/jib-2008-100.
15. Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free path querying with structural representation of result. In Proceedings of the 13th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '17). ACM, New York, NY, USA, Article 10, 7 pages. DOI: <https://doi.org/10.1145/3166094.3166104>.

16. Rustam Azimov and Semyon Grigorev. 2018. Context-free path querying by matrix multiplication. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18), Akhil Arora, Arnab Bhattacharya, George Fletcher, Josep Lluis Larriba Pey, Shourya Roy, and Robert West (Eds.). ACM, New York, NY, USA, Article 5, 10 pages. DOI: <https://doi.org/10.1145/3210259.3210264>
17. Hellings, J. (2015). Querying for Paths in Graphs using Context-Free Path Queries.
18. Szárnyas, Gábor. Graph-based analysis of JavaScript source code repositories, FOSDEM, Graph devroom (Brussels, 2018)
19. OWASP T. Top 10-2017 The Ten Most Critical Web Application Security Risks
20. Azis I. M. F., Kom M. Object Oriented Programming Php 5. – Elex Media Komputindo, 2005.
21. GREENSPECTOR tool. Available: <https://greenspector.com/en/articles/2017-06-12-analyse-statique-code-bdd-orientee-graphe/>
22. Urma, Raoul-Gabriel and Mycroft, Alan. (2015). Source-code queries with graph databases - With application to programming language usage and evolution. Science of Computer Programming. 97. 10.1016/j.scico.2013.11.010.
23. Michael Hunger: Class-Graph, leverages Cypher to collect structural insights about your Java projects Available: <https://github.com/jexp/class-graph>
24. Abdelsalam, Khaled and Kamel, Amr. (2018). Reverse Engineering State and Strategy Design Patterns using Static Code Analysis. International Journal of Advanced Computer Science and Applications. 9. 10.14569/IJACSA.2018.090178.
25. jQAssistant tool. Available: <https://jqassistant.org>
26. Available: <https://github.com/ShiftLeftSecurity/joern>
27. Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V. N. Venkatakrishnan. 2018. NAVEX: precise and scalable exploit generation for dynamic web applications. In Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18). USENIX Association, Berkeley, CA, USA, 377-392.
28. A PHP parser written in PHP. Available: <https://github.com/nikic/PHP-Parser>
29. CVE-2014-1860. Available: <https://github.com/contao/core/pull/6730>

DECLARATIVE TARGET ARCHITECTURE DEFINITION FOR DATA-DRIVEN DEVELOPMENT TOOLCHAIN

Iuriii Korenkov¹, Ivan Loginov¹

Assoc. Prof. Andrey Dergachev¹, Arthur Lazdin¹

¹ ITMO University, Russia

ABSTRACT

Today retargetable and cross-platform compilers are mainstream, because variety of hardware platforms is very large, and it is required to support general-purpose programming languages for these platforms. But retargetable compiler development process has very high cost. The main criterion is development time (develop, debug and maintain time, high entrance level). Some of the most popular solutions on the market of these compilers are GCC, LLVM. Each of them contains (in implementation) platform-specific code like platform-specific functions' implementations [1].

In general, each platform (in this context – processor architecture or hardware platform in general case) requires development its own compiler, specific for this instruction set or/and memory model, etc. One of the most complicated aspect of compiler development is to make it modular. For example, it means, that it is possible to create a custom module for GCC to support some particular architecture. Development of such module among other tasks commonly incorporates hardcoding description of the instruction set, optimizations implementation, debugger support. The concept of our solution is introduced in this paper.

Keywords: compiler, DSL, language workbench, data-driven development

INTRODUCTION

The GCC compiler uses the Register Transfer Language (RTL) [2]. RTL is the form of a program intermediate representation (IR), that looks like abstract assembly for the abstract machine with registers and memory. Besides GCC, RTL is used by other compilers, for example, CompCert [3]. In addition to IR, by means of RTL it is possible to define descriptions of the software architectures of the target CPUs. Consider a small example of code in the GCC RTL definition in the Listing 1. This example shows definition of procedure of translation to code for the target platform.

Example code parts meanings: (1) Define instruction pattern, (2) Pattern name, (3) RTX (RTL eXpression) – target instruction semantics, (4) C boolean expression, if required; (5) Target instruction in target assembly syntax.

<pre>(define_insn /*(1)*/ "movsi" /*(2)*/ (set /*(3)*/ (match_operand 0 "register_operand" "r") (match_operand 1 "const_int_operand" "k")) /*(4)*/ "i%0,%1" /*(5)*/) </pre>	<p>Source: D.XXXX = YYYYY;</p> <p>Transformation:</p> <p>(set</p> <p style="margin-left: 20px;">(reg:SI 58 [D.XXXX])</p> <p style="margin-left: 20px;">(const_int YY: [YYYYYh])</p> <p>)</p> <p>Result: li \$t0, YYYYY</p>
--	--

Listing 1 Example of target arch instruction for GCC and results of translation of this code:

Architecture of a compiler to a certain extent is depicted in Figure 1.

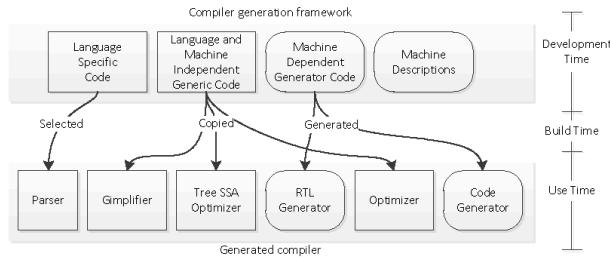


Figure 1. Brief GCC compiler architecture overview [4]

Hence, in the end of development cycle all those transformations lead to static code generation and linking. The other well-known retargetable toolchain to generate code for various target architectures is LLVM. It also uses IR as it outlined in listing 2.

```
define i32 @add2(i32 %a, i32 %b) {
entry:
%tmp1 = icmp eq i32 %a, 0
br i1 %tmp1, label %done, label %recurse
recurse:
%tmp2 = sub i32 %a, 1
%tmp3 = add i32 %b, 1
%tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
ret i32 %tmp4
done: ret i32 %b
}
```

Listing 2 Example of target arch instruction for LLVM [5]

Generally, the procedure of the source code translation to target platform is shown on Figure 2 [4].

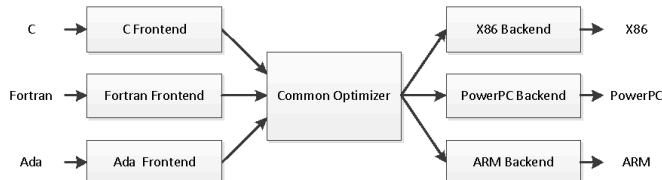


Figure 2. Overview of translation process in LLVM

Summing up, most of the retargetable solutions use hard-coded platform definitions. In some cases, compiler has modular structure and allows compiler developer to write own module, but it still contains hard-coded platform definition. To use this module, compiler developer needs to rebuild the toolchain (like in GCC). The support of this toolchain is quite complex as developer (or team) should think about a debugger, compiler (code-generator), optimizer, and the formal verification of the whole solution. Although performance of hard-coded solutions is high, a cost of development time is high as well. In modern business-realities, the result solution performance is not as important as iteration of the product lifecycle cost. So, the main purpose of this work is

to introduce the declarative target architecture definition syntax and API to reduce support time and domain entrance threshold.

PROPOSED NOTATION BASICS

Notation for target architecture definition should describe various aspects of the architecture, meaningful for an application developer. Minimum core subset of such aspects includes following: memory model structure (addressable regions and registers), opcode encodings for instructions and optionally corresponding mnemonic formats. As soon as main purpose is possibility to describe architectures intended to be a hardware computation models, this core features subset is sufficient for the first approach. Based on this, we are proposing extensible but not redundant format, consisting of architecture aspects enumeration, where each aspect defined in a form, which could be easily read, written and supported by a maintaining person without deep knowledge of the particular tool infrastructure, such as compiler-specific stages pipeline.

Top level of the proposed notation includes target architecture aspects enumeration. All aspects should be described inside an architecture block, as shown at Listing 3.

```
architecture TestArch {
    memory: /* addressable spaces information */
    registers: /* registers model information */
    instructions: /* part of instruction definitions */
    mnemonics: /* part of mnemonic definitions */
}
```

Listing 3 Top level of an architecture definition

Aspects definition can be split into multiple entries so that logically connected parts of various aspects could be described together but not interfering with each other.

Architecture definitions do not have to be complete in terms of functionality and computational completeness. They could be split into few fragments, where each block of functionality can be described as subset. Those subsets could be reused through import semantics in the particular complete architecture definition.

While proposed format and current implementation of the related instrumentation are designed in an extensible way, below is the brief overview of the core aspects.

A. Registers configuration

A part of a memory model, representing registers configuration, described in terms of physical data location cells (storages) and their addressable fragments (views).

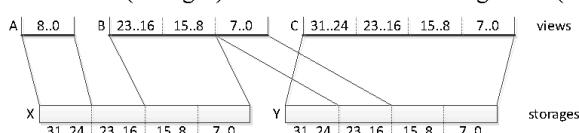


Figure 3. Red, blue and green line

```
registers:
storage X[32];
storage Y[32];
view A = X[24..31];
view B = { X[0..15], Y[16..24] };
view C = Y;
```

Listing 4 Registers storages and views relations example

Where X and Y are 32-bit register storages. Register view A provides access to the higher 8 bit for X, register view B provides access to combination of lower 16 bits of X and lower half of the higher word of Y (bits 16-23), register view C provides access to the whole content of storage Y.

B. Addressable spaces

Ranges of addressable memory or any other randomly addressable data holding entities are described with range blocks, see Listing 5.

```
memory:
range ram [0x0000..0xffff] {
    cell = 8;
    endianness = little-endian;
    granularity = 1;
}
```

Listing 5 Top level of an architecture definition

Here ram is name of the range, with which this range can be referred to. Numbers in the square brackets denotes valid address values for this range. cell element denotes size of range's cell in bits. endianness element denotes the order of the cells representing contiguous value of size more than one cell, when such value should be placed or taken from the range. It is also referred to as a byte order and can take one of the two values: little-endian or big-endian. The last part of the range description – granularity element denotes value, residue of division of effective address on which should be equal to zero for valid addressing operation.

C. Instruction set

Instruction is a minimum algorithmically meaningful encoding of the sequence of operations to be performed by an abstract interpreter. It is an atomic entity which is encoded by a bit pattern consisting of fields. Fields can denote either instruction parameters or fixed values, the purpose of the last ones is to distinguish various instructions from each other. Every possible instruction in a particular architecture should composed of such fields, that no other instruction can intersect with by a possible bit patterns of the whole opcode.

Description of every concrete instruction can be reflected by the following syntax representing an opcode encoding:

instruction <name> = { <list of fields> };

Besides, instruction definition can be followed by a function description:

instruction <name> = { <list of fields> } { <operations> };

Here <name> is a unique name for this instruction. List of fields is a comma-separated set of bit patterns and references to encodings for variable field entries. <operations> defines functional meaning of the instruction and describes operations with machine states in form similar to mix of RTL (register-transfer logic) and high-level programming language (see below). Table 7 represents possible entries of the opcode bit pattern encoding.

Table 7. Possible elements of the opcode fragment definition.

Representation	Meaning
<bit pattern with spaces>	Fixed parts of an opcode, represented by explicit bits
<field encoding name> as <value name>	Variable part of an opcode, represented by referred value encoding
<field encoding name>. <value name>	Fixed part of an opcode, represented by one element of the explicitly enumerated bit patterns
sequence <sequence encoding name>	Variable part of an opcode, represented by predefined subset of the opcode fields
<sequence encoding name>(<subfields>)	Same as previous with nested values renaming

Besides an instruction opcode pattern, there are simple and complex encodings for various parts of the opcode.

Simple encodings intended for description of immediate values, register references and predefined bit patterns:

```
encode <name> field = immediate[<bits count>];
encode <name> field = register {
    <reg view 1 name> = <bits>, <reg view 2 name> = <bits>, ...
};
encode <name> field = cases {
    <case 1 name> = <bits>, <case 2 name> = <bits>, ...
};
```

Complex encodings are sequence encodings and encodings of sequence alternatives:

```
encode <name> sequence = { <list of fields> };
encode <name> sequence = alternatives {
    <case 1 name> = { <list of fields> }, <case 2 name> = { <list of fields> }, ...
};
```

Listing 6 demonstrates complex example of sequence alternatives encoding definition.

```
instructions:
encode rmModByte sequence = alternatives {
    rm00reg = {00, rmReg as reg1, rm00AsReg as reg2},
    rm00sib = {00, rmReg as reg1, 100, sequence sibByte},
    rm00off = {00, rmReg as reg1, 101, off32 as off},
    rm01reg = {01, rmReg as reg1, rmXXAsReg as reg2, off8 as off},
    rm01sib = {01, rmReg as reg1, 100, sequence sibByte, off8 as off},
    rm10reg = {10, rmReg as reg1, rmXXAsReg as reg2, off32 as off},
    rm10sib = {10, rmReg as reg1, 100, sequence sibByte, off32 as off},
    rm11reg = {11, rmReg as reg1, rm11AsReg as reg2}
};
instruction adc3w1 = { 0001 0011, sequence rmModByte };
```

Listing 6 Top level of an architecture definition

Each instruction encoding can be followed by an instruction function definition. Such instruction function may be used in a number of ways: as compiler translation stage guidance, as interpreting machine state transition descriptions, as basis for complexity estimation of an instruction sequences and encoded algorithms, etc.

Each instruction function can be considered as C function without explicit resulting value and with variable opcode fields as arguments. While immediate and cases kind

values are treated as read-only by value arguments, register references are treated as writable by-reference arguments.

Body of an instruction function, as well as in C, is treated as block, that consists of sequence of nested statements. Basic statements are: expression statement, variable declaration statement, block statement, condition statements, loop statements.

Expressions inside these statements can be represented by the combination of various operator expressions (arithmetic operations, bit operations, comparisons, logical operations, assignment, memory indexing) and literals such as following: binary (0b00011010), decimal (12345), hexadecimal (0x78ab25FD).

For the various alternative encodings of one instruction, appearing when the sequence alternatives are used, there is a special condition syntax similar to if statement with keyword when instead of it. This when statement denotes logic branch selection depending on presence of the fields or their combination in the particular opcode instance. Examples of <condition> element see below in the description of the mnemonics format matching.

D. Mnemonics

Mnemonics' purpose is to allow user to interact with native code representation. They stand for human-readable form of native code in a linear form, directly inherited from underlying nature of native instruction sequences.

Mnemonic definitions describe matching between every instruction opcode alternative and combination of mnemonic name and arguments format string. Mnemonic variants are described in various ways depending on instruction arguments and structure of part of the particular opcode fields. It is decomposable into multiple entries with additional conditions eliminating intersections between concrete opcodes.

It is possible to describe format string of the arguments once and then use it for many mnemonics with simultaneous formats. If the accompanied by the same opcode structure conditions group of argument format strings occurs in many mnemonics for different instructions, this group together with conditions can also be described separately from the particular mnemonics. Here are how basic elements of the mnemonics aspect look like:

```
format <name> is "<format string>"; /* predefined format string */
format <name> of <list of format specs>; /* predefined group of format strings */
mnemonic <name> <list of format per instruction specs>; /* mnemonic definition */
```

Each format in <list of format per instruction specs> begins with list of arguments and ends with optional conditions on opcode fields presence for mnemonics-to-opcodes matching ambiguities resolution. Instead of format string there each format entry can refer to the predefined format string or to the predefined group of format strings. In the latter case there should be an '...' designation on the place of the arguments denoting that actual argument lists resided in the referenced group definition:

```
format <name> of (<arg list 1>) "<format string 1>",
(...)<format strings group name>,
(<arg list 2>) "<format string 2>" when <condition>,
(<arg list 3>) <format string name> when <condition>, ...
```

Note the difference between group of format strings and mnemonic with explicit multiple format strings:

```
mnemonic <name> for <insn name> (<arg names list 1>) "<format string 1>",
          (...) <format strings group name>,
          for <insn name> (<arg list 2>) "<format string 2>" when <cond>,
          (<arg list 3>) <format string name> when <condition>, ...
```

Each of the format spec entries in the mnemonic definition optionally preceded with instruction name so the following format strings would be matched to the variants of the referred instruction's opcode. So, the simples form of mnemonic definition is:

mnemonic nop ():

Such definition tells that there is instruction with name nop somewhere and its opcode has no arguments, thus no format string required. Here is complex example of the format definition for the adc3w1 instruction mentioned above demonstrating all the elements together:

```
format sib-to-plain of
  (reg1, reg2)    "{1}, [{2}]"      when rm00reg,
  (reg1, r32)     "{1}, [{2} * 1]"  when rm00sib and x1,
  (reg1, r32)     "{1}, [{2} * 2]"  when rm00sib and x2,
  (reg1, off)     "{1}, [{2}]"      when rm00off,
  (reg1, reg2, off) "{1}, [{2} + {3}]" when rmXXAsReg,
  (reg1, r32, off) "{1}, [{2} * 1 + {3}]" when x1,
  (reg1, r32, off) "{1}, [{2} * 2 + {3}]" when x2,
  (reg1, reg2)     "{1}, {2}"        when rm11reg;
mnemonic adc for adc3w1(...) sib-to-plain;
```

MANIPULATION API

A set of libraries was created for the proposed notation. These libraries expose few levels of API for manipulations with target architecture definition: definitions source text model, architecture definitions model, architecture information semantic model, native code and mnemonic operations.

When some tool uses target architecture definitions as its input for manipulations with native code, these APIs are used during this tool initialization.

At first, particular architecture definition should be read from one or multiple text files. As a result of their parsing, source text models are produced. Then during translation process these text models are gathered into one rich target definition model. Here is where aggregation of the information from multiple input definition files occurs. When all required aspects and subsets are introduced into definition model, next step is collection. Collection means selection from the introduced definitions of all information required for the concrete target architecture model to be functionally and computationally complete. During this step, a number of actions take place: merging of subsets, resolving of dependencies between definition parts and validation of the instruction encodings.

As a result of collection step, an object implementing IArchitectureInfo interface is produced. Besides information about current architecture, this object has two methods named CreateNativeCodeParser and CreateMnemonicsParser. They provide a way to perform conversions between mnemonic representation of instructions and native code

bytes. As every native instruction instance description associated with corresponding parts of the complete architecture info, robust opportunities for various code analysis, validation, generation and other tools creation exist.

In addition to basic scenario, this APIs can be used to generate architecture definitions programmatically: on source model level and on complete definition level. It makes architecture definitions itself a subject of manipulation. This is where metamodeling begins with all of its might in models' manipulation, transformations and such.

CONCLUSION

Developed declarative target platform architecture definitions format and supporting libraries demonstrate that model-based approach can be easily expanded to the low-level tools development bringing almost the same level of customizability, as in pure high-level modelling software such as Eclipse Modelling Framework.

For the evaluation of the proposed solution, few experimental tools were created: compact assembler with NASM-like syntax, disassembling tool as analogue to objdump utility, simple command-line debugger, syntax highlighting profile for our language workbench [7].

When the core functionality of the library was implemented, it was not a troublesome task to use its API to create an example tools and achieve consistency in their behaviour.

Further work would be concentrated primarily on the high-level code processing features: (1) ensure, that proposed notation allows describing the majority of actual hardware architectures on the market; (2) common intermediate models framework creation, intended to be used as target-architecture-independent basis for manipulations with algorithm intermediate representations of the programs; (3) generalized IR translation algorithms between various particular computational models; (4) complete modular compiler pipeline based on described principles and introduced libraries; (5) retargetable without toolchain recompilation compiler driven by proposed target architecture definitions.

REFERENCES

- [1] GCC, the GNU Compiler Collection website, Web: <https://gcc.gnu.org/onlinedocs/gcc/Target-Builtins.html#Target-Builtins>
- [2] GCC, the GNU Compiler Collection website, Web: <https://gcc.gnu.org/onlinedocs/gccint/RTL.html>
- [3] Kästner, Daniel, et al. "CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler." ERTS2 2018-Embedded Real Time Software and Systems. 2018.
- [4] The Conceptual Structure of GCC, Web: <https://www.cse.iitb.ac.in/grc/intdocs/gcc-conceptual-structure.html>
- [5] The Architecture of Open Source Applications, LLVM, Web: <http://aosabook.org/en/llvm.html>
- [6] Korenkov, Yuriy, Ivan Loginov, and Arthur Lazdin. "PEG-based language workbench." Open Innovations Association (FRUCT), 2015 17th Conference of. IEEE, 2015.

**ТЕХНИЧЕСКИЕ НАУКИ — МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ
ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ МАШИН, КОМПЛЕКСОВ И
КОМПЬЮТЕРНЫХ СЕТЕЙ (05.13.11)**

05.13.11

**А.М. Дергачев канд. техн. наук, Д.С. Садырин, А.Г. Ильина канд. техн. наук,
И.П. Логинов, Ю.Д. Кореньков**

Университет ИТМО,

Факультет программной инженерии и компьютерной техники,
Санкт-Петербург, dam600@gmail.com, cyberguru007@yandex.ru, agilina@itmo.ru,
ivan.p.loginov@gmail.com, ged.yuko@gmail.com

**МЕТОДЫ И СРЕДСТВА ОБНАРУЖЕНИЯ УЯЗВИМОСТЕЙ АЛЛОКАТОРОВ
ДИНАМИЧЕСКОЙ ПАМЯТИ БИБЛИОТЕКИ GLIBC**

Работа посвящена проблеме эксплуатации уязвимостей аллокаторов динамической памяти в библиотеке glibc. В работе рассмотрены современные способы обнаружения таких уязвимостей и представлен комплексный подход для решения проблемы автоматического выявления уязвимостей на разных стадиях жизненного цикла разработки программного обеспечения. Данный подход применим для верификации аналогичных реализаций аллокаторов динамической памяти, таких как ptmalloc, dmalloc, tcmalloc, jemalloc, musl.

Ключевые слова: *верификация, ошибки в ПО, символьное выполнение, проверка моделей, динамическая память, язык С.*

Введение

В настоящее время программные системы различного уровня и назначения используется практически во всех областях человеческой деятельности. Ошибки в системном программном обеспечении (ПО) могут привести к потерям данных, финансовому ущербу, нарушению производственных процессов, физическому уничтожению критически важных объектов [1]. Одним из важных этапов поиска ошибок является процесс верификации ПО, в том числе системного ПО, реализованного средствами языка С. В частности, можно выделить отдельную группу ошибок, связанных с реализацией аллокаторов (англ. allocator), занимающихся выделением динамической памяти и являющихся одной из причин уязвимости ПО. Для обнаружения такого рода ошибок необходимо проводить формальную верификацию в рамках жизненного цикла ПО.

В первой части статьи рассматриваются дефекты безопасности, относящиеся к использованию динамической памяти в языке С, а также методы и средства обнаружения связанных с этими дефектами уязвимостей. Во второй части предлагается комплексный подход, учитывающий недостатки существующих решений, который представляется перспективным в рамках решения задачи обнаружения рассматриваемых в статье уязвимостей ПО.

Методы и средства обнаружения уязвимостей аллокаторов динамической памяти

Для выявления дефектов допустимо применять как методы тестирования ПО, так и методы верификации. Среди них фаззинг [2,3], динамический анализ [4], верификация с использованием моделей [5,6], символьное выполнение [7], применение бинарных решающих деревьев – BDD (англ. Binary Decision Diagrams) [8], решение задачи пропозициональной выполнимости (англ. вариант – SATISFIABILITY, или SAT). Следует отметить, что применение каждого из методов подразумевают значительные накладные расходы на выполнение программы.

При верификации программ на языках C/C++ возможны нижеследующие варианты.

1. Эмуляция выполнения кода на виртуальной машине. Существенными недостатками этого метода являются высокие затраты при эмуляции больших программ и сложность эмуляции окружения и некоторых системных вызовов операционной системы. К эмуляторам, например, относится фреймворк Angr. В нем кроме символьного выполнения кода приложения производится символьное выполнение кода из разделяемых библиотек. К эмуляторам также относится фреймворк S2E (англ. Selective Symbolic Execution). Фреймворк S2E использует символьное выполнение на эмуляторе аппаратного обеспечения QEMU, исполняемом в KVM-режиме (англ. Kernel-based Virtual Machine).

2. Инструментация исполняемого файла после сборки и выполнение его на реальном процессоре. Примером фреймворка динамической инструментации является фреймворк Triton.

3. Инструментация исходного кода программы во время компиляции и последующее выполнение исполняемого файла. Данный подход представлен в работе [9].

4. Выполнение изменения исходных кодов программы отдельно перед компиляцией, например, средствами языка TXL (англ. Turing eXtender Language). Данный подход используется на работе [10].

5. Символьное выполнение внутреннего представления кода во время компиляции. Данный подход представлен в работе [11].

6. Выполнение с помощью отладчика после сборки.

На сегодняшний день существует ряд инструментальных средств, таких как CBMC, НАИТ, Heaphopper, ArcHeap, MOPS, посвященных поиску ошибок распределения динамической памяти следующих типов: Heap-based Buffer Overflow, Double Free, Use after free и NULL Pointer Dereference [12], но ни один из них не лишен недостатков.

CBMC

CBMC (англ. C Bounded Model Checker) [13] представляет собой верификатор, который обеспечивает возможность ограниченной проверки моделей для языков ANSI-C и C++. CBMC позволяет верифицировать уязвимости переполнения буфера, безопасность указателей, исключения и пользовательские утверждения (англ. user-specified assertions).

НАИТ

НАИТ (англ. Heap Analyzer with Input Tracing) [14] реализует подход автоматического сбора и визуализации информации о состоянии кучи и операциях, которые над ней выполняются. Прототип инструмента построен на основе фреймворка Triton для динамического бинарного анализа программ. С целью отслеживания операций кучи (англ. heap) используется динамический подход. Для анализа трассировки используется библиотека динамической бинарной инструментации Pintools от компании Intel, которая осуществляет интеграцию с Triton – обеспечивается привязка Python для непосредственного взаимодействия с трейсером и использования всех его функций, прежде всего механизмы перехвата событий.

На настоящий момент НАИТ не является конечным продуктом [14]. Достоинствами НАИТ являются быстрое прототипирование, качественная визуализация процессов выделения памяти, простая отладка. Требуется доработка для повышения эффективности и уменьшения накладных расходов, которые могут доходить до 500%.

Heap Hopper

В работе [15] на примере решения Heap Hopper, основанного на фреймворке Angr, демонстрируется применение символьного выполнения программного кода для поиска уязвимостей переполнения буфера [16]. На каждом шаге выполнения программы создается объект класса SimState, в котором хранится состояние регистров и памяти программы в данный момент. Регистры и память могут иметь конкретное, либо символьное значение. Каждая символьная переменная представляется в виде класса BitVectorSymbol. Также существует возможность вручную помечать необходимые входные данные как символьные – это может быть символьная память, представляемая в виде класса SimSymbolicMemory, или символьный файл, представляемый классом SimFile.

При достижении инструкции условного перехода, добавляется ограничение на символьную переменную. При вызове оператора malloc с символьным параметром размера выделяемой памяти, создается чанк (англ. chunk) памяти с символьными метаданными. На каждом шаге в классе SimState сохраняется состояние кучи.

При анализе алгоритмов Heap Hopper было выяснено, что при вызове malloc с символьным параметром возвращается символьный адрес памяти. Символьные адреса сохраняются в полях malloc_dict, free_dict объекта класса HeapConditionTracker. После каждого вызова malloc производится проверка условий для разных типов уязвимостей, таких как arbitrary write, overlapped allocations, non-heap address allocation, non-heap address free, double free [17]. При выполнении malloc не учитывается внутреннее состояние кучи, представленное структурой malloc_state (состояние полей bins, fastbins, bitmap), что говорит о приближенном варианте модели кучи.

ArcHeap

ArcHeap [18] – автоматический инструмент для обнаружения неисследованных техник эксплуатации кучи, независимо от их реализации. Для его работы необходимо описать параметры аллокатора памяти, а также задать возможные действия над кучей. Во время исследования ArcHeap проверяет, могут ли комбинации этих действий потенциально использоваться в качестве техник эксплуатации, таких как произвольная запись в память или перекрывающиеся чанки, и генерирует PoC (англ. Proof of concept) – доказательство осуществимости концепции, которое демонстрирует обнаруженную технику эксплуатации. В сравнении с Heap Hopper техника поиска уязвимостей в ArcHeap не позволяет проанализировать весь спектр их нежелательных эксплуатаций. В работе [18] авторы признают, что ArcHeap является “фундаментально неполным”.

Modelchecking Programs for Security properties (MOPS)

MOPS [19] - верификатор моделей, извлечённых из кода программ, написанных на языке C. Требования корректности задаются в специальном виде и соответствуют утверждениям так называемого «защитного» программирования (англ. defensive programming). Верификатор содержит готовую базу таких утверждений, планируется написание графического пользовательского интерфейса.

Во время компиляции производится анализ всех возможных путей исполнения без учета условий. Собираются все возможные трассы. Из них выделяются важные для безопасности операторы. Имея контекстно-свободную грамматику языка C, программа представляется в виде автомата с магазинной памятью. Модель безопасности представляется в виде конечного автомата, принимающего последовательность операций безопасности. Последовательность операторов безопасности является «подходящей», если принимается на вход конечным автоматом и переводит его в «безопасное» состояние.

Предлагаемое решение

Несмотря на достоинства существующих инструментов, нельзя считать работу по реализации верификатора для поиска уязвимостей аллокаторов памяти завершенной. Верификатор должен находить все возможные уязвимости, иметь низкие накладные расходы и не должен изменять код анализируемой программы. В настоящий момент инструмента, соответствующего таким требованиям, не существует.

В качестве решения предлагается осуществлять обнаружение уязвимостей, совмещая реальное исполнение программы с символьным исполнением. Динамическое символьное исполнение позволит применять техники исследования путей программы и, добавляя предикаты безопасности к ограничениям пути (англ. path constraint), проверять потенциально опасные операции на наличие реальных ошибок в программах.

Для построения более точного представления кучи (в сравнении с Heap Hopper) в процессе символьного выполнения предлагается работать напрямую со структурой malloc_state. Для каждого состояния символьного выполнения потребуется проверять возможность перезаписи top чанка или полей prev_size, size, fd, bk [20], возможность создания специального «подложного» чанка – участка памяти на стеке или куче, наличие

уязвимости double free, возможность высвобождения указателя по произвольному адресу (Arbitrary free).

Данный подход позволит выявить на стадии разработки ПО потенциальную возможность эксплуатации уязвимостей, вызванных ошибками работы с динамической памятью. С целью снижения накладных расходов вместо эмуляции планируется выполнение тестируемого приложения с использованием специального отладчика, находящегося в стадии разработки.

Заключение

В работе рассмотрены существующие методы и средства обнаружения уязвимостей аллокаторов динамической памяти, их достоинства и недостатки. Предложен подход к поиску ошибок в аллокаторах памяти на основе динамического символьного выполнения, который предполагает устранение указанных недостатков. По завершении реализации данного подхода после проведения качественного и количественного сравнения с существующими решениями предложенный подход может быть применен для верификации других видов аллокаторов динамической памяти. В дальнейшем также планируется исследовать методы поиска ошибок в программах с интерактивным вводом.

Список литературы

1. Андреев Ю.С., Дергачев А.М., Жаров Ф.А., Садырин Д.С. Информационная безопасность автоматизированных систем управления технологическими процессами // Известия высших учебных заведений. Приборостроение - 2019. - Т. 62. - № 4. - С. 331-339
2. Bhardwaj, M., Bawa, S. Fuzz testing in stack-based buffer overflow. Advances in Intelligent Systems and Computing, 2019, 759, pp. 23-36
3. Mouzaran, M., Sadeghiyan, B., Zolfaghari, M. A. Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes. Proceedings - 2015 IEEE 21st Pacific Rim International Symposium on Dependable, 2016
4. Program analysis. URL: https://en.wikipedia.org/wiki/Program_analysis (дата обращения 15.12.2019).
5. Кларк Э. М., Грамберг О., Пелед Д. Верификация моделей программ. Model Checking. М.: МЦНМО. 2002.
6. Karna, A.K., Chen, Y., Yu, H., Zhong, H., Zhao, J. The role of model checking in software engineering. Frontiers of Computer Science, 12 (4), 2018, pp. 642-668.
7. Dudina, I.A., Belevantsev, A.A. Using static symbolic execution to detect buffer overflows. Programming and Computer Software, 43 (5), 2017, pp. 277-288
8. Ebendt R., Fey G., Drechsler R. Advanced BDD optimization. Springer. 2005.
9. Jang, Y.-S., Choi, J.-Y. Automatic prevention of buffer overflow vulnerability using candidate code generation. IEICE Transactions on Information and Systems, E101D (12), 2018 pp. 3005-3018
10. Dahn, C., Mancoridis, S. Using program transformation to secure C programs against buffer overflows. Proceedings - Working Conference on Reverse Engineering, WCRE, 2003-January.
11. Loding H., Peleska J. Symbolic and Abstract Interpretation for C/C++ Programs / Electronic Notes in Theoretical Computer Science 217, 2008.URL: <https://www.sciencedirect.com/science/article/pii/S1571066108003885>. (дата обращения 15.12.2019).
12. Common Weakness Enumeration. URL: <https://cwe.mitre.org/index.html>. (дата обращения 15.12.2019).
13. CCBMC Homepage. URL: <http://www.cs.cmu.edu/~modelcheck/cbmc/>, (дата обращения 15.12.2019)
14. Atzeni, A., Marcelli, A., Muroni, F., Squillero, G. HAIT: Heap analyzer with input tracing ICETE 2017 - Proceedings of the 14th International Joint Conference on e-Business and Telecommunications, 4, 2017, pp. 327-334.
15. Eckert M, Bianchi A, Wang R, Shoshitaishvili Y, Kruegel C, Vigna G. Heap Hopper: Bringing Bounded Model Checking to Heap Implementation Security, 27th USENIX Security Symposium, 2018

16. *Dudina, I.A., Belevantsev, A.A.* Using static symbolic execution to detect buffer overflows. Programming and Computer Software, 43 (5), 2017, pp. 277-288
17. A repository for learning various heap exploitation techniques [Электронный ресурс]. Режим доступа: <https://github.com/shellphish/how2heap> Яз. анг. (дата обращения 15.12.2019).
18. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. URL: <https://arxiv.org/pdf/1903.00503.pdf>. (дата обращения 15.12.2019).
19. MOPS: an Infrastructure for Examining Security Properties of Software.URL: <http://web.cs.iastate.edu/~hridesh/teaching/610/06/02/papers/mops-ccs02.pdf> (дата обращения 15.12.2019).
20. Understanding glibc malloc. URL: <https://exploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/comment-page-1/>(дата обращения 15.12.2019).

ИССЛЕДОВАНИЕ И РАЗРАБОТКА ЯЗЫКОВОГО ИНСТРУМЕНТАРИЯ НА ОСНОВЕ PEG-ГРАММАТИКИ

Ю. Д. КОРЕНЬКОВ, И. П. ЛОГИНОВ

*Университет ИТМО, 197101, Санкт-Петербург, Россия
E-mail: ivan.p.loginov@gmail.com*

Рассматриваются средства создания предметно-ориентированных языков, так называемый языковой инструментарий, и их особенности. Предложено новое решение — прототип языкового инструментария, основанного на PEG-грамматике, который позволяет интуитивно понятным образом описывать грамматики предметно-ориентированных языков. Приведен сравнительный анализ ключевых возможностей разработанного прототипа и существующих решений.

Ключевые слова: языковой инструментарий, предметно-ориентированный язык, языко-ориентированное программирование.

Введение. Создание программного обеспечения необходимо во множестве отраслей промышленности и экономики. Примером могут служить задачи моделирования, для которых характерны большие объемы вычислений и данных, сложные логические структуры данных, построение сложных таблиц входных/выходных данных и пр. Ручное программирование подобных задач — процесс достаточно медленный и связанный с большим количеством ошибок.

Решить эту проблему позволяет автоматическое программирование. Для компьютерного генерирования программ необходимо создание формального языка, который может быть использован для описания решения задачи в терминах ее предметной области. Такие языки называются предметно-ориентированными [1] или DSL (Domain Specific Language).

Хорошо известным примером таких языков служит SQL (Structured Query Language). Этот язык был создан специально для работы с реляционными базами данных. На основе SQL были разработаны некоторые языки для манипуляции данными, например язык интегрированных запросов LINQ (Language Integrated Query) [2], встроенный в C# для выполнения запросов к коллекциям данных.

В настоящий момент создано множество DSL для использования в различных предметных областях: например, языки для описания аппаратуры интегральных схем (VHDL, Verilog), языки для символьных вычислений (Mathematica, Maple, Maxima) и др. С другой стороны, продолжается разработка новых решений по созданию средств языко-ориентированного (ЯО) программирования, так называемого языкового инструментария [3]. Собственно, новой разработке — созданию прототипа языкового инструментария — и посвящена настоящая статья.

Предлагаемый прототип позволит:

- повысить удобство использования создаваемых DSL за счет интеграции с существующими средствами разработки (например, Visual Studio, Eclipse) или создания нового stand-alone (автономного) средства для ЯО-программирования;
- повысить производительность труда разработчиков.

Состояние предметной области. Одним из самых известных примеров инструментария для языко-ориентированного программирования является система JetBrains MPS (Meta Programming System) [4]. В течение процесса разработки языка MPS предоставляет пользователю возможности, характерные для современных IDE, такие как: автоматическое завершение текста, выделение синтаксических конструкций в режиме реального времени, проверка

ошибок и др. MPS поставляется с большим количеством примеров, которые расширяют функциональность языка Java. Следует отметить, что MPS не зависит от языков программирования.

Другим примером языкового инструментария является проект Nitra [5]. В настоящее время Nitra поддерживает разработку генераторов синтаксических анализаторов (далее — парсеров, от англ. parser — производить грамматический разбор) для .NET Framework. Однако эта система спроектирована таким образом, что позволяет создавать парсеры и для других платформ — потенциально Nitra может использоваться для генерации кода парсеров на языке C или для виртуальных машин, таких как LLVM.

Решение, предлагаемое в данной статье, позволяет создавать предметно-ориентированные языки, и его можно использовать совместно с множеством языков программирования (более пятидесяти), таких как C#, F#, C++/CLI и др. Представленное решение не использует дополнительных зависимостей от других проектов, как, например, Nitra, который построен на базе языка Nemerle.

Конфигурируемый парсер. Для создания предметно-ориентированного языка необходимо средство, позволяющее анализировать исходный код, описываемый разными грамматиками. Таким образом, парсер должен быть гибко конфигурируемым.

Инструменты для построения классических парсеров не предоставляют достаточной свободы при конфигурировании анализатора, так как основаны на порождающих грамматиках (согласно иерархии Хомского), которые приводят к необходимости сложных преобразований формального языка и парсера для него.

Возможным решением проблемы может служить класс грамматик типа PEG (Parsing Expression Grammars) [6], т.е. грамматик, „разбирающих“ выражения. PEG-грамматики состоят из множества правил (множества нетерминалов, согласно Хомскому). Парсер для грамматики легко может быть построен как стековая машина, которую можно конфигурировать динамически. Кроме того, PEG-грамматики не требуют отдельной фазы разбора текста, так как правила грамматики, по сути, описывают сам процесс разбора.

Подход к реализации. В ходе исследования реализован анализатор, который обрабатывает PEG-грамматику, представленную как набор именованных правил, описывающих разбираемые выражения. Данный анализатор предоставляет возможности для описания правил грамматик различной сложности. Правила можно параметризовать при помощи выражений-аргументов.

Также для грамматики в целом возможно устанавливать шаблон пропуска символов. Такой шаблон определяет игнорируемые анализатором фрагменты текста, что позволяет различать эти фрагменты с точки зрения их необходимости для пользователя или модели анализа, представляющей результат разбора. Примером использования шаблона служат документирующие комментарии, которые должны быть особым образом выделены в текстовом редакторе для легкого визуального восприятия.

Результат анализа текста представляет собой дерево элементов, называемое StringTreeNode. Каждый элемент описывает фрагмент разбираемого текста в соответствии с правилами грамматики. По окончании анализа это дерево можно автоматически преобразовать в дерево любых объектов, например в абстрактное синтаксическое дерево компилятора.

Используемый в предлагаемом прототипе язык для описания грамматики предусматривает написание грамматики в интуитивно понятной форме. Поддерживаются регулярные выражения и атрибуты для правил (написанные в квадратных скобках, например: [right]power: expr '^' expr). Приведем список поддерживаемых атрибутов:

- left — помечает выражение как левоассоциативное;
- right — помечает выражение как правоассоциативное;
- OmitPattern — объявляет шаблон для пропуска;

`RootRule` — объявляет корневое правило;

`RewriteRecursion` — помечает правило, содержащее альтернативы, как необходимое для автоматического устранения рекурсии.

Альтернативные варианты в процессе разбора могут быть указаны следующим образом:

— как выражение типа „или“ при помощи символа ‘|’: `a|b|c`;

— как расширяемое правило, которое состоит из нескольких вложенных правил.

Расширяемые правила дают возможность расширить однажды описанную грамматику без необходимости ее изменения.

Отладчик грамматик. Для упрощения процесса разработки грамматик для DSL был создан специальный отладчик, позволяющий детализировать процесс анализа текста в соответствии с грамматикой, а также изучить деревья разбора. Окно приложения отладчика разделено на следующие области:

- 1) текстовое поле для ввода анализируемого текста;
- 2) дерево разбора с применением фильтров;
- 3) полное дерево разбора;
- 4) текстовое поле, содержащее объявление грамматики;
- 5) протокол процесса разбора;
- 6) использованные в процессе разборе правила (в виде дерева).

Дерево правил грамматики способствует поиску структурных ошибок в выражениях, описывающих правила, а подробный журнал процесса разбора текста — поиску логических ошибок. Применение полного дерева разбора очень полезно при анализе поведения парсера.

Генерация кода. Один из важных аспектов использования предметно-ориентированных языков — кодогенерация, за счет которой возможно взаимодействие кода, написанного вручную, и кода, полученного в результате генерации. При этом целесообразно избегать генерации промежуточного кода, а DSL интегрировать в язык программирования общего назначения.

При генерации используются разные подходы, специфические для языков программирования, например:

— в языке С возможно использование текстовых макросов [7], с помощью которых могут быть описаны термины предметной области; затем в процессе компиляции (точнее, работы транслятора макросов) таким образом описанные термины будут заменены на конструкции на языке С;

— во многих языках программирования присутствуют элементы „синтаксического сахара“, позволяющие скрыть сложные конструкции при помощи простых выражений, которые заменяются компилятором на полноценные языковые конструкции.

Кодогенерация — важная проблема в процессе интеграции разрабатываемого решения в IDE, поскольку генератор необходимо формировать по результатам анализа. В настоящее время существует возможность генерировать исходя из описания грамматик текстовую модель на языке C#. Исходный код может быть скомпилирован при помощи компилятора C# для дальнейшего использования с любыми инструментами и языками, поддерживаемыми CLI (Common Language Infrastructure — общеязыковая инфраструктура, широко распространенная в мире разработки программного обеспечения).

Поддержка интегрированной среды разработки. Инструментарий для поддержки процессов создания и использования DSL средой разработки должен предоставлять:

- текстовый редактор для создания DSL и редактирования текста с его использованием;
- способ интеграции созданного DSL в проект, для которого этот язык предназначен.

В качестве решения предлагается расширение для IDE Microsoft Visual Studio, которое предоставляет возможности для выделения синтаксиса грамматики, описывающей язык, а также для выделения фрагментов текста на созданном DSL. В текстовом редакторе возмож-

но установить произвольную схему выделения (подсветки) — набор стилей визуального представления текста для правил и терминальных символов; также существует возможность переключения между несколькими схемами при помощи выпадающего списка над окном текстового редактора.

Заключение. Разработанный прототип языкового инструментария предоставляет синтаксис, понятный пользователю на интуитивном уровне. В настоящий момент существует поддержка этого инструментария в Microsoft Visual Studio, также разработан специальный инструмент для отладки грамматик. В дальнейшем планируется существенно расширить функциональность этого прототипа, в частности, добавив возможности автоматического завершения текста в соответствии с разработанной грамматикой.

СПИСОК ЛИТЕРАТУРЫ

1. *Martin Fowler Website: Domain Specific Language* [Электронный ресурс]: <<http://martinfowler.com/bliki/DomainSpecificLanguage.html>>.
2. Pro LINQ: Language Integrated Query in C# 2010. M.: Williams, 2011. 656 p.
3. *Martin Fowler Website: Language Workbenches: The Killer-App for Domain Specific Languages?* [Электронный ресурс]: <<http://martinfowler.com/bliki/LanguageWorkbench.html>>.
4. *Martin Fowler Website: A Language Workbench in Action — MPS* [Электронный ресурс]: <<http://martinfowler.com/articles/mpsAgree.html>>.
5. An Introduction to Nitra [Электронный ресурс]: <<http://blog.jetbrains.com/blog/2013/11/12/an-introduction-to-nitra/>>.
6. Ford B. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. Cambridge, MA: Massachusetts Inst. of Technology, 2004.
7. International Standard ISO/IEC 9899:201x “Programming Languages — C”. 2000. 166 p.

Сведения об авторах

Юрий Дмитриевич Кореньков

— Университет ИТМО; кафедра информатики и прикладной математики-1; ассистент

Иван Павлович Логинов

— магистрант; Университет ИТМО; кафедра информатики и прикладной математики-1; E-mail: ivan.p.loginov@gmail.com

Рекомендована кафедрой
информатики и прикладной математики-1

Поступила в редакцию
31.08.15 г.

Ссылка для цитирования: Кореньков Ю. Д., Логинов И. П. Исследование и разработка языкового инструментария на основе PEG-грамматики // Изв. вузов. Приборостроение. 2015. Т. 58, № 11. С. 934—938.

PEG-BASED LANGUAGE WORKBENCH RESEARCH AND DEVELOPMENT

Yu. D. Korenkov, I. P. Loginov

ITMO University, 197101, St. Petersburg, Russia

E-mail: ivan.p.loginov@gmail.com

Tools for creation of a domain-specific language (called the language workbench) and their features are considered. A new solution based on the PEG-grammars is proposed – a prototype of language workbench that provides an intuitive way for description of a domain specific language grammar. A comparison of key features of the developed prototype with the opportunities provided by existing solutions is presented.

Keywords: language workbench, domain-specific language, language-oriented programming.

Data on authors

- Yury D. Korenkov* — ITMO University, Department of Computer Science and Applied Mathematics-1;
Assistant;
- Ivan P. Loginov* — Master Sci.; ITMO University, Department of Computer Science and Applied
Mathematics-1; E-mail: ivan.p.loginov@gmail.com

For citation: Korenkov Yu. D., Loginov I. P. PEG-based language workbench research and development // Izvestiya Vysshikh Uchebnykh Zavedeniy. Priborostroenie. 2015. Vol. 58, N 11. P. 934—938 (in Russian).

DOI: 10.17586/0021-3454-2015-58-11-934-938

05.13.11

Ю.Д. Кореньков

Университет ИТМО,
 Факультет программной инженерии и компьютерной техники,
 Санкт-Петербург, ged.yuko@gmail.com

**МЕТОД ПРЕДМЕТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА
 ИСХОДНЫХ ТЕКСТОВ ПРОГРАММ
 НА ОСНОВЕ СЕМАНТИЧЕСКИХ МОДЕЛЕЙ**

В работе рассматривается семантический анализ текстов программ и программных систем для проверки совместимости между их частями, написанными на различных языках программирования. Предложен язык описания спецификаций семантических моделей программ и их отображений, а также язык описания синтаксических моделей исходных текстов программ для построения абстрактных семантических графов в ходе семантического анализа текстов программ.

Ключевые слова: семантическая трансляция, предметно-ориентированные языки, инкрементальный анализ.

1. Введение

Разработка программ и программных систем на сегодняшний день зачастую сопряжена с декомпозицией проекта на несколько частей, таких как отдельные приложения или модули. При этом разные части программного проекта могут быть реализованы как с использованием одного языка программирования, так и различных языков программирования одновременно. В зависимости от применяемых методов организации программных интерфейсов между частями решения в их исходном коде могут выделяться фрагменты, отвечающие за сопряжение частей друг с другом. При этом, с точки зрения разработчика программного проекта, интерес представляет как можно более раннее выявление ошибок сопряжения элементов программной системы между собой.

Выявление и исправление ошибок в программном коде производится в составе двух основных процессов:

- 1) работы с исходным кодом программы;
- 2) сборки программного проекта, компиляции и компоновки всех его частей, их тестировании и верификации.

Исправление ошибок в программном коде, выявленных на любом из этапов жизненного цикла программного проекта, возвращает разработчика к исходному коду, требующему внесения изменений. По этой причине для программиста представляет интерес выявление как можно большего количества ошибок на этапе работы с исходным кодом программы. Такую задачу на сегодняшний день решают средства интегрированных сред разработки (IDE). Однако в силу специфики различных способов сопряжения программных компонент для различных программных проектов в общем случае такая задача может быть решена на уровне среды разработки только для небольшого числа предопределённых авторами IDE сценариев использования (например, частичная проверка корректности XAML-разметки относительно кода на языке C# в IDE Visual Studio).

2. Предметно-ориентированная семантика

Рассмотрим некоторые частные случаи, когда в составе компонент программного проекта возникают так называемые слои сопряжения:

- 1) при разработке веб-приложений пользовательский интерфейс может реализовываться на таких языках, как JavaScript или TypeScript, а служба веб-сервиса при этом может разрабатываться на C# или Java. При этом со стороны пользовательского интерфейса будут каким-то образом формироваться обращения к конечным точкам программного интерфейса службы;

2) при разработке распределённых приложений компоненты могут сопрягаться как с помощью готовых реализаций механизма удаленного вызова процедур на базе общих интерфейсов, так и с помощью специализированных протоколов или обёрток вокруг существующих библиотек, отвечающих за передачу данных и реализующих поведение, подобное удалённому вызову процедур;

3) при использовании баз данных код приложения будет содержать в том или ином виде слой доступа к данным, отвечающий за формирование запросов к базе данных с помощью построения запросов в виде объектных моделей или текстовых команд;

4) программные интерфейсы отдельных библиотек могут обладать особенностями поведения, которые требуют от импортируемого кода определённых паттернов использования;

5) явное использование методов языково-ориентированного программирования или (мета-)моделирования может требовать формирования или обработки структур данных, организованных предопределенным образом, спецификация которого не пересекается с семантикой языка программирования общего назначения.

Во всех этих случаях семантика языков программирования, на которых ведётся разработка проекта, недостаточна для статической проверки корректности программы. При этом сценарии 1-4 могут рассматриваться как применение внутренних предметно-ориентированных языков, а сценарий 5 – как применение внешних предметно-ориентированных языков.

Так как разнородные элементы в составе программной системы призваны решать разные задачи, примем, что контракты программного интерфейса каждого из них формулируются в терминах предметной области, соответствующей решаемой конкретным элементом задаче. Во всех рассматриваемых случаях программный код того или иного модуля можно разделить на слои в соответствии с выполняемой задачей (если разработчики не игнорировали принципы разработки ПО, такие как декомпозиция и принцип единственности ответственности). Рассмотрим слои, отвечающие за сопряжения со смежными компонентами (модулем, базой данных, службой, библиотекой): их код будет организован не произвольным образом для решения узкой задачи, но соответствовать ограничениям, диктуемым предметной областью соответствующего программного интерфейса. То есть семантика фрагмента текста программы расширяется посредством введения в неё понятий из предметной области смежного компонента.

3. Семантический анализ и предметно-ориентированные языки

На предмет возможности выполнения предметно-ориентированного анализа семантических моделей программ в процессе их редактирования был проанализирован ряд инфраструктур интегрированных сред разработки, наиболее популярных по данным Google Trends, основные из которых перечислены в таблице 1.

Таблица 1 – Популярные IDE

	Название IDE	Сайт проекта
1	Visual Studio	https://visualstudio.microsoft.com/ru/
2	Eclipse	https://www.eclipse.org/
3	Visual Studio Code	https://code.visualstudio.com/
4	IDEA	https://www.jetbrains.com/idea/
5	NetBeans	https://netbeans.org/
6	Xcode	https://developer.apple.com/xcode/

Во всех случаях единственным способом введения в процесс редактирования специфических для отдельного программного проекта проверок над исходным кодом является реализация специализированного расширения среды разработки. Это говорит о невозможности выполнения предметно-ориентированного семантического анализа встроенными средствами для поддерживаемых рас пространёнными средами разработки языков программирования. Единственным способом, автоматически решающим вопрос совместимости двух частей программного проекта, доступным без дополнительных усилий,

является генерация кода этих частей на основе общей спецификации. Что, по сути, является эквивалентом создания специализированного расширения IDE, но без возможности проверки текстов программ в процессе редактирования, так как для этого нужна интеграция решения с текстовым редактором IDE, а совместное использование семантической информации в существующих средах разработки либо не предусмотрено, либо затруднено.

Другим подходом может являться использование специализированных средств работы с предметно-ориентированными языками, позволяющих выполнять семантический анализ текстов программ, будучи изначально разработанными с возможностью настройки такого анализа.

Был рассмотрен ряд подходов и инструментов, позволяющих выполнять семантический анализ исходных текстов программ своими силами. Следует уточнить, что предметом интереса являются именно способы выполнения анализа семантической информации, полученной на основе текстов программ, а не только синтаксических структур в исходном тексте, что исключает из рассмотрения средства генерации парсеров, требующие самостоятельной реализации логики семантического анализа. Изучение существующих средств показало, что они могут быть разбиты на три группы:

- 1) не предназначенные для использования в реальном времени в процессе редактирования текста [1][2][3][4][5][6][7][8][9][10][11][12][13];
- 2) предназначенные для использования в процессе редактирования текста программы внутри своей специализированной среды разработки или специализированного режима редактирования [12][13][14][15];
- 3) предназначенные для использования в процессе редактирования моделей программ в терминах близких к модели абстрактного синтаксического дерева, не рассматривающие анализ текстов программ [12][16].

Таким образом, ни один из существующих инструментов, предназначенных для выполнения предметно-ориентированного семантического анализа, не может быть встроен в процесс редактирования текста в IDE, уже задействованной в ходе разработки программного проекта, если это не та же самая IDE, в рамках которой он изначально создан. Кроме этого, описание спецификаций, задающих необходимый семантический анализ, во всех случаях требует изучения специализированных предметных областей (языки и среды метамоделирования, синтаксического анализа, онтологического моделирования, и т.п.) [4][17][18][19], что повышает порог вхождения для решения этой задачи.

4. Предлагаемое решение

В результате анализа существующих решений была выявлена потребность в создании средства описания предметно-ориентированных проверок над семантической информацией о частях текстов программ в процессе их редактирования, которое минимизировало бы потребность изучения или использования средств, внешних по отношению к IDE, уже задействованной в разработке отдельного программного проекта.

Проверка условий корректности текста программы в отношении элементов семантики, не входящих состав исходного языка программирования, требует выделения соответствующей семантической информации из программы. При этом исходными данными для такой операции является синтаксическая или семантическая модель текста программы в терминах исходного языка программирования. Так, предметно-ориентированный анализ исходных текстов программ может быть сформулирован как последовательная семантическая трансляция информации о структуре текста программы из предметной области языка программирования в предметную область, в которой условия необходимых ограничений могут быть сформулированы без дополнительных усилий.

При этом система понятий предметной области на каждом из этапов анализа задаёт естественные ограничения для соответствующей части текста программы. При необходимости она также может быть расширена дополнительными условиями.

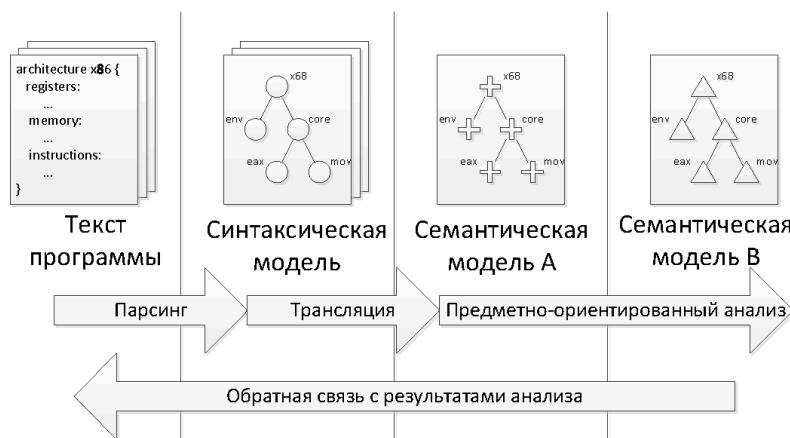


Рис. 1 – Последовательность этапов анализа текста программы

Исключение ошибок при семантической трансляции требует ограничений схемы представления данных на каждом из этапов. Первый шаг анализа при этом требует привести представление синтаксической модели к тому же обобщённому виду, что и семантические модели, трансляция между которыми будет выполняться в дальнейшем. Таким представлением синтаксической информации является абстрактный синтаксический граф (Abstract Syntax Graph – ASG), построение которого также требует однозначной спецификации элементов синтаксиса, которые должны соответствовать узлам графа. Это приводит к разделению описания семантического анализа текста программы на спецификации трёх видов: спецификации семантических моделей для анализа, спецификации семантических трансляций для перехода между разными моделями, и спецификации синтаксической модели для синтаксического анализа и построения первичного ASG, являющегося входными данными для первого этапа трансляций.

Для решения задачи декомпозиции спецификаций предлагается ввести широко распространённый механизм лексических областей в описание синтаксических моделей, на основе которых формируются как грамматики для синтаксического анализа, так и схемы данных ASG. Это делает описание синтаксических моделей более лаконичными, чем в плоской БНФ-подобной форме и фиксирует назначение элементов синтаксиса, значительно упрощая создание спецификаций. Например, в процессе разработки решения была составлена синтаксическая спецификация языка C# версии 4 объёмом 367 строк, средняя длина строки 23 символа. Идентичная БНФ-подобная грамматика 692 строки, средняя длина строки 46 символов.

Для описания самих семантических трансляций предлагается использовать синтаксис на основе языка LINQ, рассматривая при этом элементы семантических моделей как элементы наборов данных, обладающих полями, соответствующими атрибутам семантической модели. Таким образом, описание семантической трансляции между различными семантическими моделями становится подобным реляционному отображению между различными схемами данных. Написание спецификаций в таком виде не требует специализированных знаний, так как использование реляционных отображений при работе с базами данных на сегодняшний день является одной из распространённых практик.

Метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей посредством семантических трансляций, описанных в виде, подобном реляционным отображениям, позволяет абстрагироваться от логики самого процесса анализа, так как не задаёт последовательности применения трансляций, но описывает зависимости между ними. Это даёт возможность разработки и использования алгоритмов инкрементальной семантической трансляции.

Таблица 2 – Пример определения предметно-ориентированной модели

Модель	Пример фрагмента описания
Синтаксическая	<pre>defSyntax { definition: (ruleSet rule)*; ruleSet: name '{' imports item* '}' { item: rule ruleSet; }; ruleSetImport: attrs alias complexName ';'; rule: name '::' body ';'; }</pre>
Семантическая	<pre>defSemantic { scope: { name: string; rules: rule[]; namespace { namespaces: namespace[]; } rule { expr; } }; }</pre>
Отображение	<pre>defSemantic.scope(s: scope) { parent = s; namespace(rs: ruleSet) from rs in definition.ruleSet { name = rs.complexName; namespaces = from crs in rs.item.ruleSet select new namespace(this, crs); rules = from r in rs.item.rule select rule(this, r); } rule(r: defSyntax.rule) from r in definition.item.rule { name = r.complexName; rules = from cr in r.body.simple.rule select rule(this, cr); expr = syntax.expr(this, r.body.simple.expr); } };</pre>

Инкрементальность анализа позволяет минимизировать затрагиваемый в ходе анализа объём данных, а значит выполнять анализ значительно быстрее, вплоть до анализа в реальном времени в ходе редактирования исходного текста программы.

5. Заключение

В работе рассмотрен вопрос проверки корректности исходного текста программ в части сопряжения элементов программных систем в некоторых сценариях при разработке программных проектов. Предложен метод предметно-ориентированного анализа исходных текстов программ на основе семантических моделей посредством семантических трансляций на основе отображений, подобных реляционным. Предложен синтаксис описания

синтаксических и семантических моделей, трансляций между ними, понижающий порог вхождения для использования данного метода при разработке программных проектов. Приведены примеры описаний на основе предложенного синтаксиса. И сравнительная оценка объёмов грамматики на примере языка C#.

Список литературы

1. Vasudevan N., Tratt L. Comparative study of DSL tools //Electronic Notes in Theoretical Computer Science. – 2011. – Т. 264. – №. 5. – С. 103-121.
2. Kurtev I. et al. Model-based DSL frameworks //Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. – 2006. – С. 602-616.
3. Jouault F., Bézivin J. KM3: a DSL for Metamodel Specification //International Conference on Formal Methods for Open Object-Based Distributed Systems. – Springer, Berlin, Heidelberg, 2006. – С. 171-185.
4. Tikhonova U. et al. Applying Model Transformation and Event-B for Specifying an Industrial DSL //MoDeVVa@ MoDELS. – 2013. – С. 41-50.
5. Tikhonova U. Reusable specification templates for defining dynamic semantics of DSLs //Software & Systems Modeling. – 2019. – Т. 18. – №. 1. – С. 691-720.
6. Eakman G. et al. Practical formal verification of domain-specific language applications //NASA Formal Methods Symposium. – Springer, Cham, 2015. – С. 443-449.
7. Breslav A. DSL development based on target meta-models. Using AST transformations for automating semantic analysis in a textual DSL framework //arXiv preprint arXiv:0801.1219. – 2008.
8. Kurtev I. et al. Model-based DSL frameworks //Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. – 2006. – С. 602-616.
9. Sadilek D. A. Prototyping domain-specific language semantics //Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. – 2008. – С. 895-896.
10. Keshishzadeh S., Mooij A. J., Hooman J. Industrial Experiences with a Formal DSL Semantics to Check Correctness of DSL Transformations //arXiv preprint arXiv:1511.08049. – 2015.
11. Combemale B. et al. Essay on semantics definition in MDE. An instrumented approach for model verification. – 2009.
12. Erdweg S. et al. The state of the art in language workbenches //International Conference on Software Language Engineering. – Springer, Cham, 2013. – С. 197-217.
13. van der Storm T. The Rascal language workbench //CWI. Software Engineering [SEN]. – 2011. – Т. 13. – С. 14.
14. Diekmann L., Tratt L. Eco: A language composition editor //International Conference on Software Language Engineering. – Springer, Cham, 2014. – С. 82-101.
15. Vergu V., Néron P., Visser E. DynSem: A DSL for dynamic semantics specification //26th International Conference on Rewriting Techniques and Applications (RTA 2015). – Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
16. Szabó T., Erdweg S., Voelter M. Inca: A dsl for the definition of incremental program analyses //Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. – 2016. – С. 320-331.
17. Ehrig H., Prange U., Taentzer G. Fundamental theory for typed attributed graph transformation //International conference on graph transformation. – Springer, Berlin, Heidelberg, 2004. – С. 161-177.
18. Ulitin B., Babkin E., Babkina T. Ontology-based DSL development using graph transformations methods //Journal of Systems Integration. – 2018. – Т. 9. – №. 2. – С. 37-51.
19. Fisher M., Dean M. Automapper: Relational database semantic translation using owl and swrl //Proceedings of the IASK International Conference E-Activity and Leading Technologies, Porto, Portugal. – 2007.